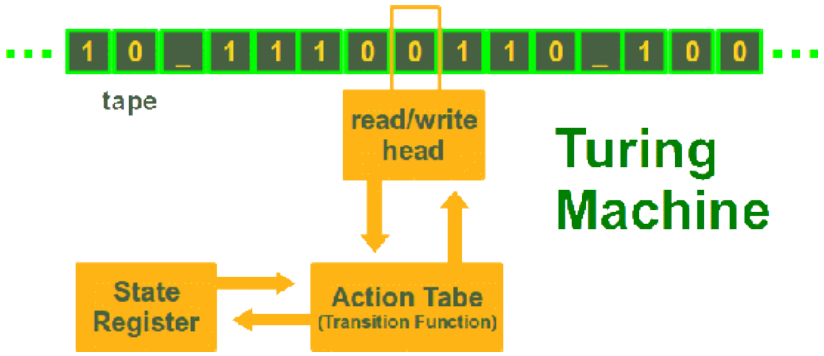
L-Università ta' Malta

Data Structures And Algorithms 2

Prof. John Abela
Department of CIS
Faculty of ICT
University of Malta

john.abela@um.edu.mt

+ 365 79367936
Room 1A/27
FICT Building

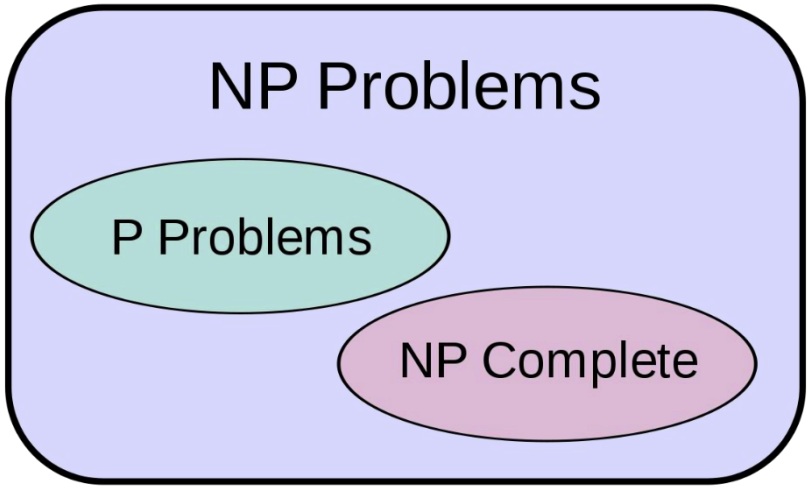


Turing Machine

Module 1

Introduction to NP-Completeness

2021-2022



Faculty of ICT

Department of Computer Information Systems

Data Structures and Algorithms 2

ICS2210 - 5 ECTS Credits

Data Structures and Algorithms 2

ICS2212 - 4 ECTS Credits

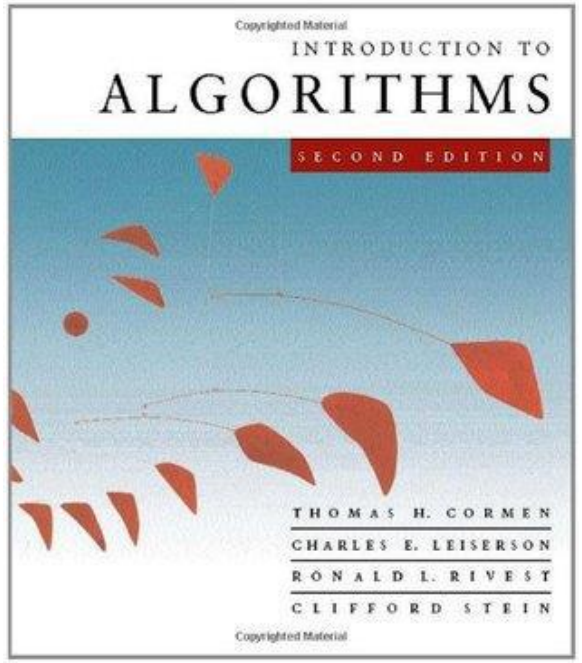
Data Structures and Algorithms 2

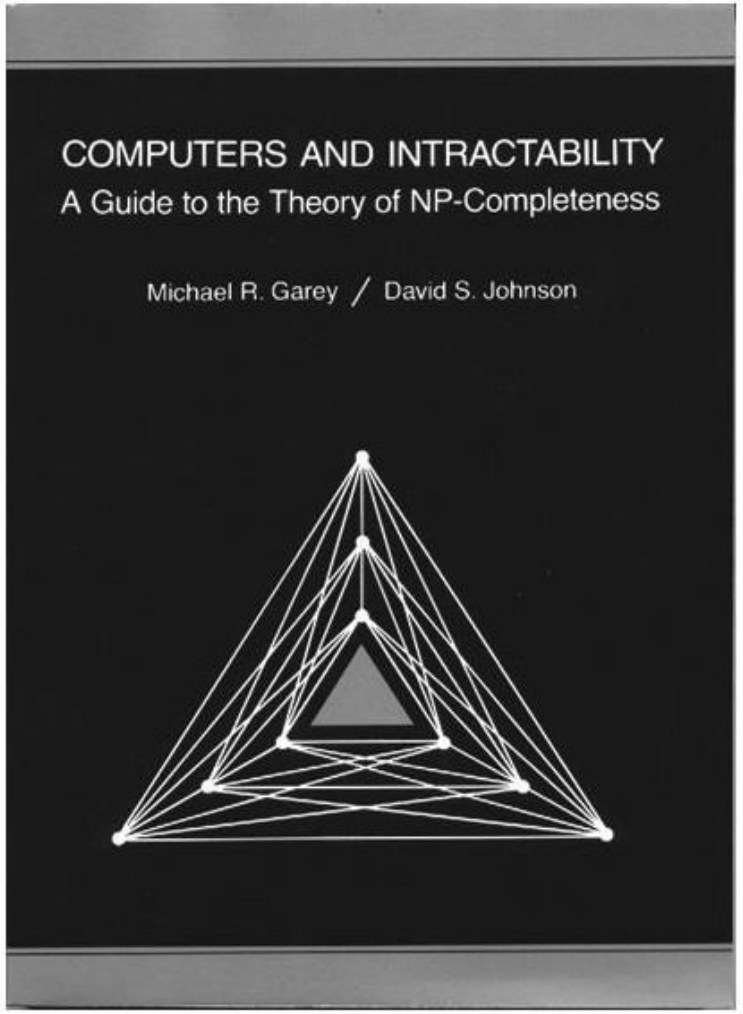
Also, **graduate students** in a **qualifying** or **preparatory** year.

- Quick Revision of Algorithmic Complexity
- The Theory of NP-Completeness
  - Not formal language theoretic
- Turing Reductions
- Greedy Algorithms and Dynamic Programming
- Approximation Algorithms
- Case Studies
  - The Stable Marriage Problem
  - The Task Assignment Problem

Data Structures and Algorithms 2

- **Compulsory** for **second year undergraduate** for **CS**, **AI**, and **CIS** students.
- **Post-grad** for M.Sc. students in **Computational Physics**, **ICT**, and **Bioinformatics**.
- Main **textbooks** are **CLRS** and **Garey** & **Johnson**.





January, 1979

Quick mathematical Preliminaries

Remember the **following definitions** from **Discrete Mathematics**:

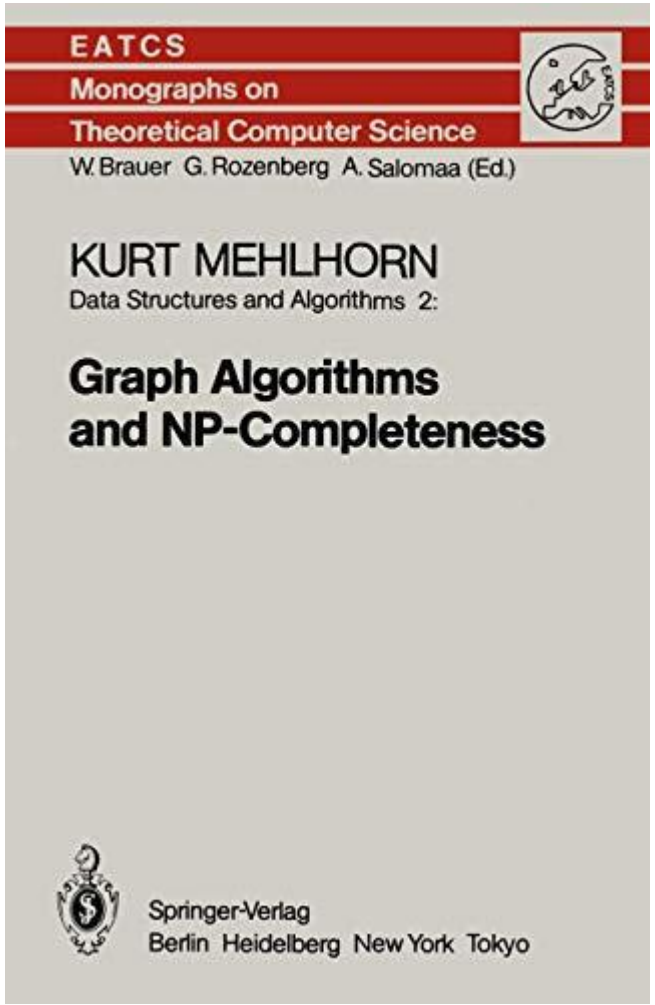
- **Graphs** and their **Representation**.
- **Relations** on Sets.
- **Equivalence Classes**.
- **Operators** and **Closure**.
- **Functions**.
- **Cartesian Products**.

Quick Revision

- **Algorithms** & **Programs**.
- **Iteration** and **Recursion**.
- **Computable** and **Non-Computable** Problems.
- **Asymptotic** Complexity.
- **Big O** Notation.

We then proceed directly to the:

Theory of NP-Completeness

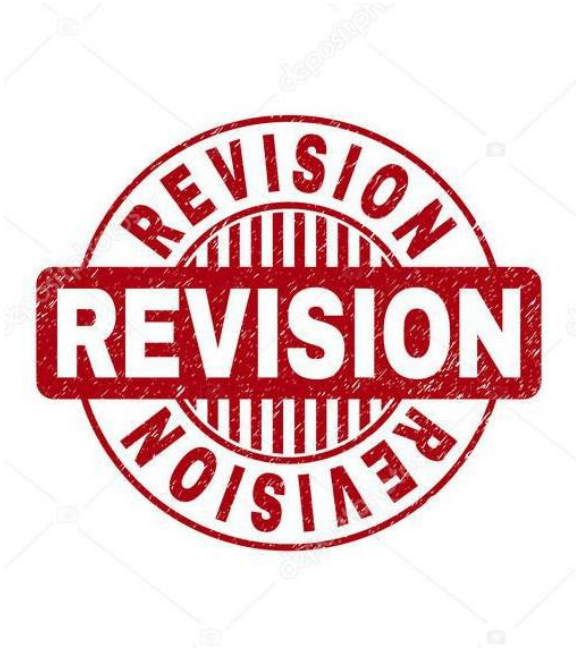


Quick Revision

Algorithms and Programs

- An **algorithm** is a **finite number** of **computational steps** that **terminate**.

- If the **algorithm** does **not terminate** then it is **called** a **program**.



Quick Revision

Time and Space Complexity

- The ***time complexity*** of an **algorithm** is a **measure** of the **time taken** by an **algorithm** to **execute**. This is **expressed** as a **function of the size of the input**.
- The **time** is **actually** '**steps**'.

- The ***space complexity*** of an **algorithm** is a **measure** of the **memory required** by an **algorithm** to **execute**. This is also **expressed** as a **function of the size of the input**.

Quick Revision

Recursion vs Iteration

- ***Iteration*** and ***Recursion*** are **two approaches** to **solving** a **problem**.
- **Iteration** is, in **general**, more **efficient**.
- **Recursion** often **results** in more **elegant code** but requires a ***recurrence relation***.
- Both are ***Turing Powerful*** but **some problems** are more **elegantly solved** using **recursion** and **vice versa**.

Quick Revision

Algorithm Analysis

- We **keep** must **track** of the '**growth rates**' of the **computational workload** as a **function** of the **problem size**/**dimension**.
- This **area** is very **important** and **subject** to **intense study** and **research**.
- If we **do not** do this we **will** make **stupid mistakes** such as **writing exponential time** algorithms to **solve** problems in **business**, **industry**, and **academia**.
- This is not **very clever**.

Quick Revision

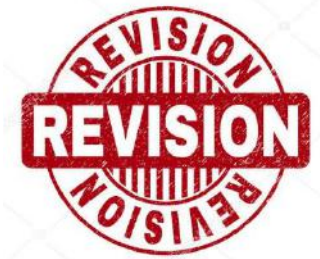
Eight functions *g(n)* that occur **frequently** in the **analysis** of **algorithms** (in order of **increasing** rate of **growth** relative to **n**):

- Constant ≈ ***1***
- Logarithmic ≈ $\log_2 n$
- Linear ≈ ***n***
- Log Linear ≈ $n \log_2 n$
- Quadratic ≈ n^2
- Cubic ≈ n^3

Feasible or Tractable

- Exponential ≈ 2^n
- Super Exponential ≈ $n!$

Intractable

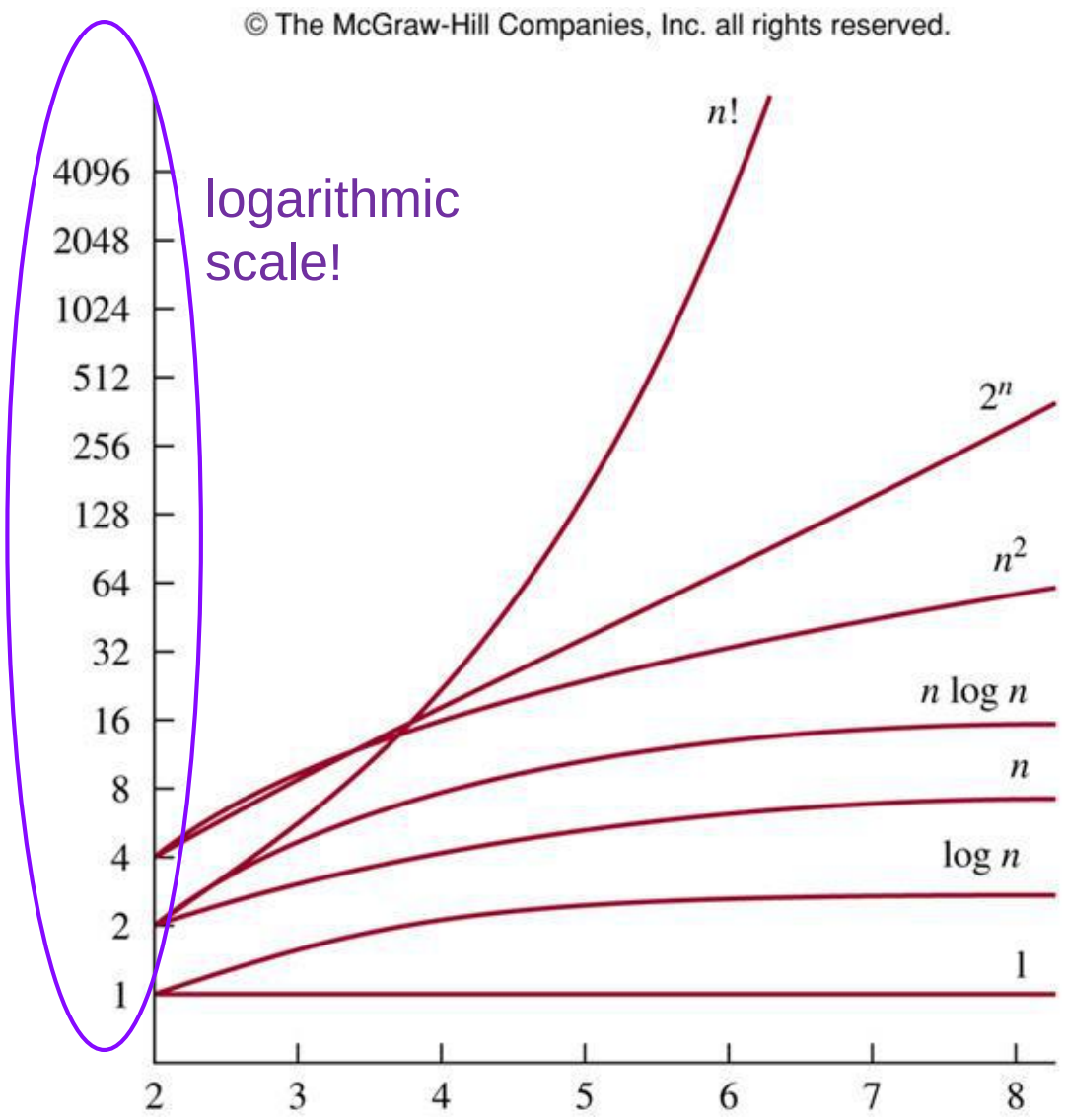


Quick Revision

Comparing Algorithm Growth Rates

Input Size: n	(1)	log n	n	n log n	n^2	n^3	2^n
5	1	3	5	15	25	125	32
10	1	4	10	33	100	10^3	10^3
100	1	7	100	664	10^4	10^6	10^{30}
1000	1	10	1000	10^4	10^6	10^9	10^{300}
10000	1	13	10000	10^5	10^8	10^{12}	10^{3000}

Quick Revision



Comparing Algorithm Growth Rates

Quick Revision

- **Algorithms** with **exponential** or **super-exponential** time complexity are **considered intractable**.

- **Number** of **atoms** in **universe** is **approximately** 10^{80} or $2^{265.7}$

- These **running time functions** grow **very fast**. Writing an algorithm with these **time complexities** is **not practical** (or **very clever**).

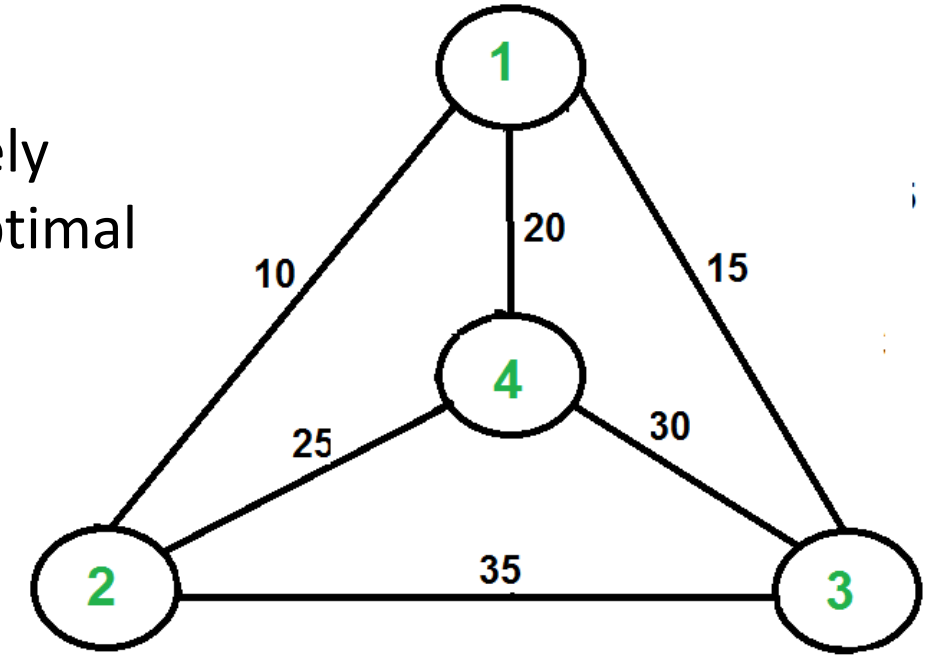
- You are **now** in **second year**. You **must understand**, and give **due importance** to, the **time complexity classes** of the **algorithms** you **work** with – **whether** your **own** or from **others**.

The Travelling Salesman Problem

- Instance: a complete **weighted undirected** graph **G=(V,E)** (all **weights** are **non-negative**).
- Problem: to find a **Hamiltonian** cycle (**tour**) of **minimal cost**.

A very **simple problem** to **describe**. In approximately **70 years** nobody has **found** an **polynomial-time** optimal algorithm.

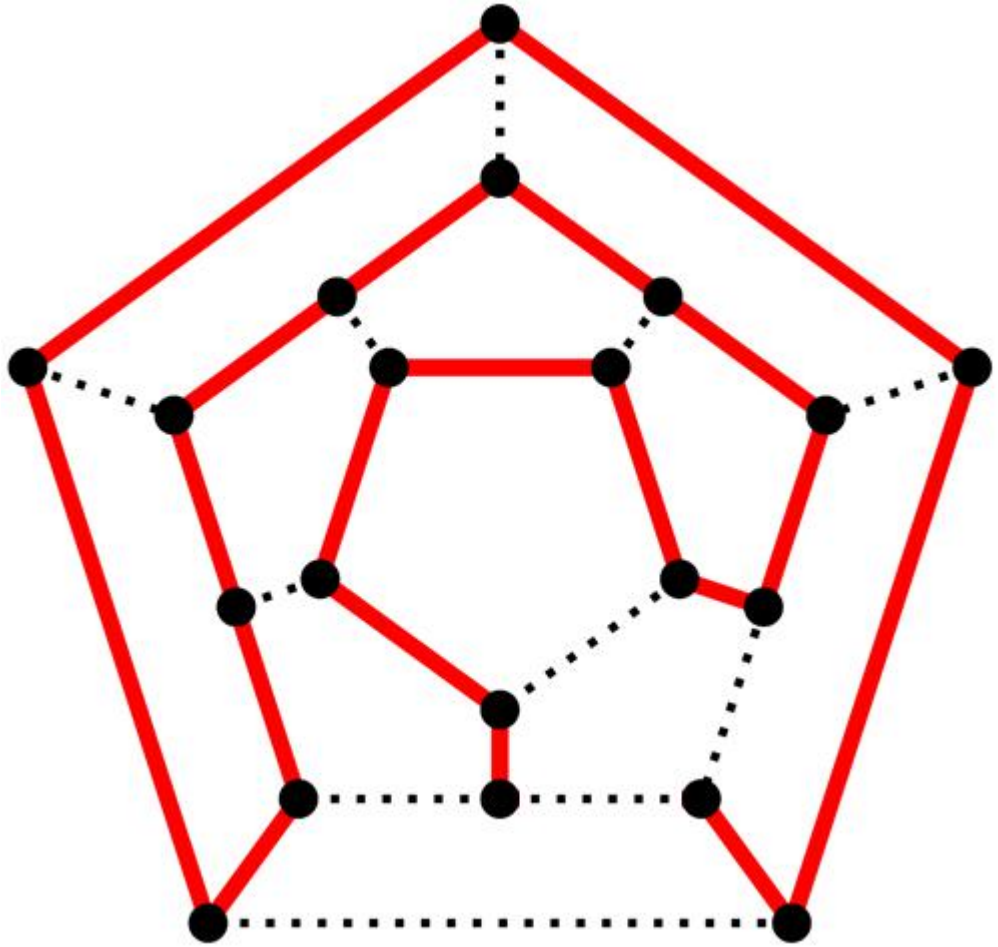
All known optimal algorithms require **at least exponential time**.



Hamiltonian Cycles and Paths

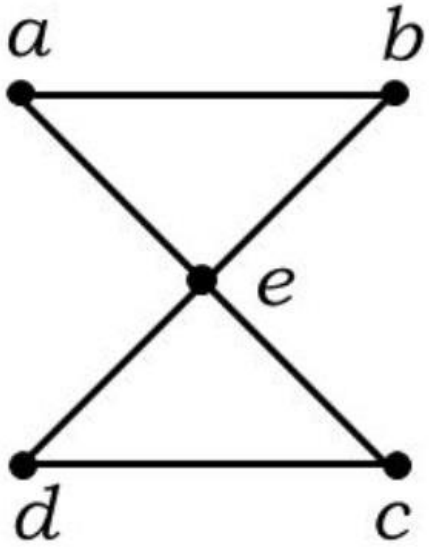
Given a graph **G** = (**V**, **E**).

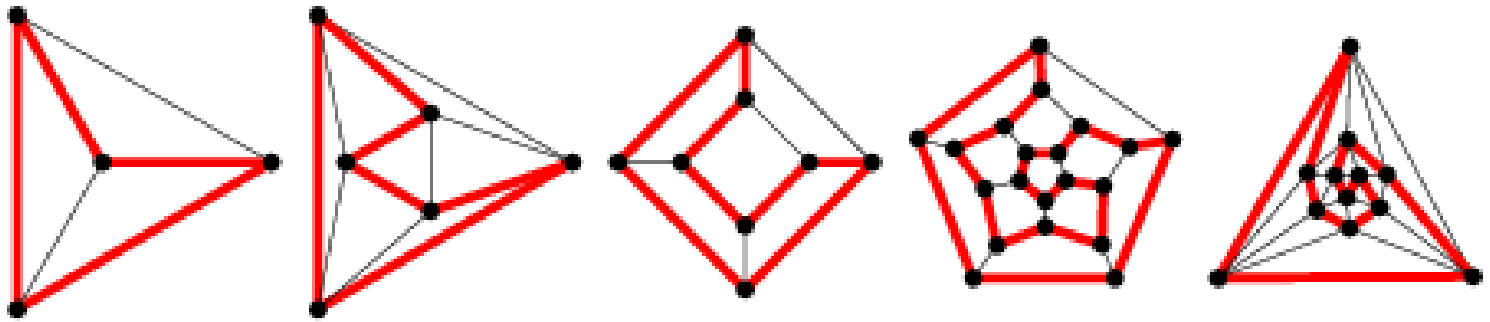
- A **Hamiltonian Path** is a **path**, a **sequence** of **connected** nodes (**vertices** or **cities**), that **includes** every **node**. Every **node** is **visited exactly** once.
- A **Hamiltonian Cycle** is a **Hamiltonian Path** that is a **cycle**. This **means** it **starts** from a **source** node **returns** to this **source** node.
- To **obtain** a **Hamiltonian Path** from a **Hamiltonian Cycle** just **remove** any **edge**.
- To **determine** if a **graph contains** at **least** one **Hamiltonian Cycle** is a **hard problem**.
- To **find** the **Hamiltonian Cycle** with the **lowest cost** is an **even harder** problem.



Hamiltonian Cycles and Paths

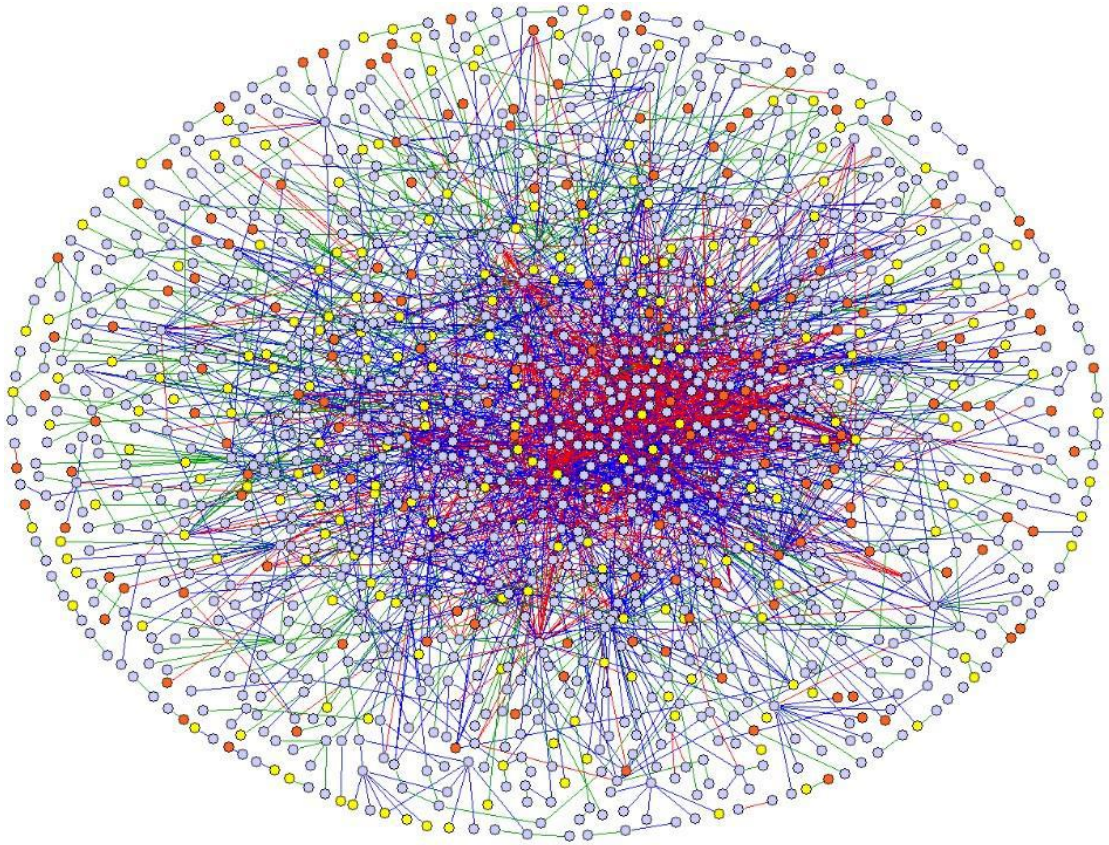
- Graphs **may have** a **Hamiltonian Cycle**.
- Graphs may **not have** a Hamiltonian Cycle.
- Note that **Complete Graphs** always have at **least one Hamiltonian Cycle**.
- **TSP** is **about finding** the **shortest Hamiltonian Cycle**.





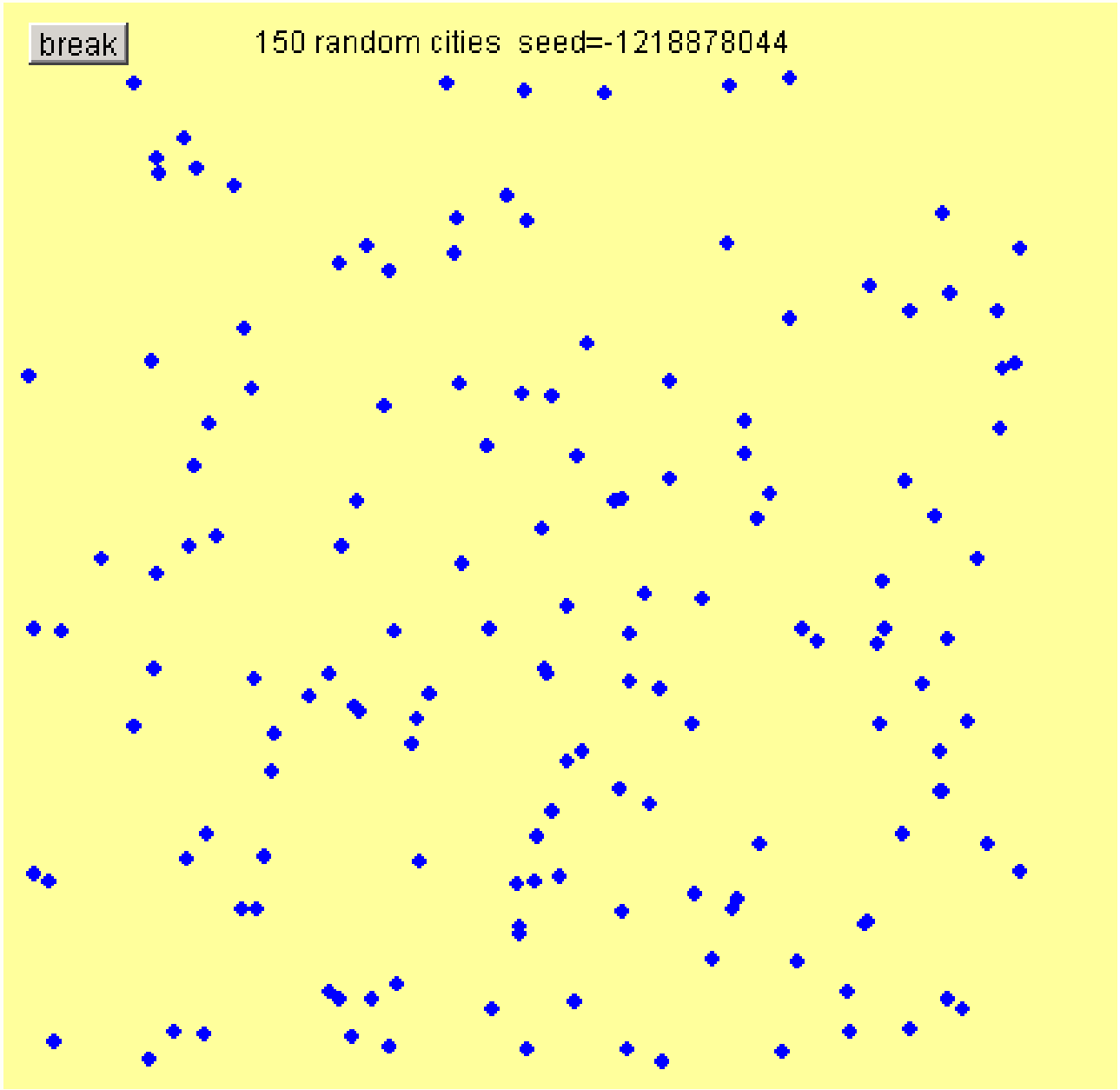
Hamiltonian Cycles and Paths

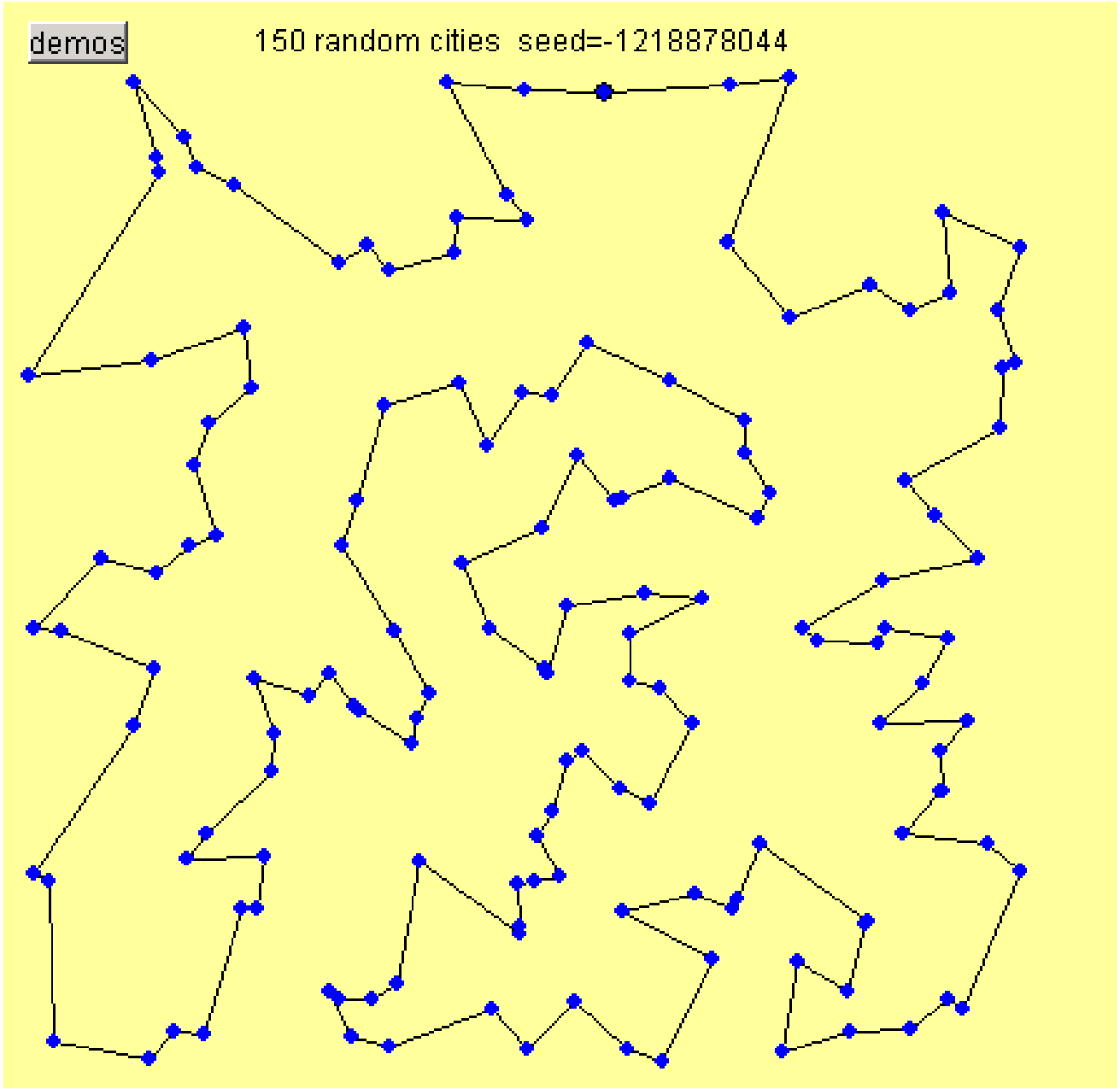
- For **small** (**less than** around **10-12** nodes) a **Hamiltonian Cycle** can **probably** be **identified** by **inspection**.
- In **general**, finding a (**single**) **Hamiltonian Cycle** is **computationally** a **difficult** problem.
- Finding the **shortest** **Hamiltonian Cycle** is **therefore**, if **anything**, **harder**.



The Travelling Salesman Problem

- 33 Node Traveling Salesperson Problem
- **8,683,317,618,811,886,495,518,194, 401,280,000,000** possible routes.
- **ASCI White**, an **IBM supercomputer** being used by **Lawrence Livermore National Labs** to model **nuclear explosions**, is **capable** of 12 trillion **operations** per **second** (**TeraFLOPS**) peak **throughput**.
- ASCI **White** would take **11.7 billion** years to **exhaustively search** all the **solution** space.


break
150 random cities seed=-1218878044


demos
150 random cities seed=-1218878044

The Travelling Salesman Problem

- For **150** cities the **number** of **possible tours** is **150!**
- This is **more** than the **number** of **atoms** in the entire **universe**.
- If a **computer** checked **1000 tours** per **second** and **started** at the **time** of the **Big Bang** (**14.7 billion years** ago) **then** by now it **would** have **checked less** than **1%** of the **search space**.
- This is **precisely** why we **call** these **problems** **intractable**.

Problems and their Instances

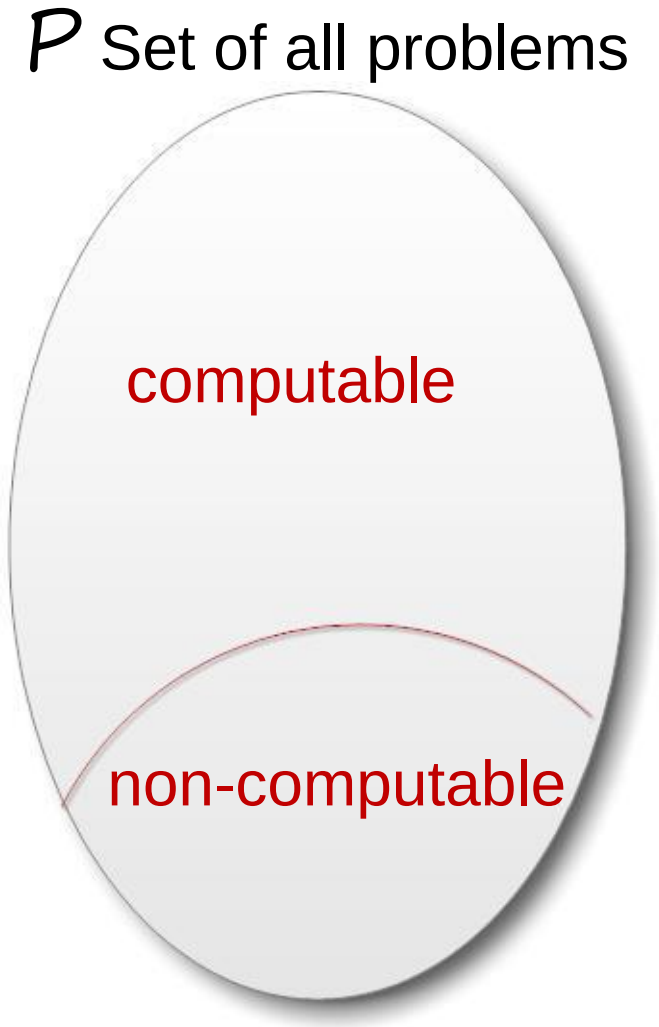
- We must **distinguish** between a **problem** and **instances** of that problem.
- By '**a problem**' we mean the **definition** of a problem – **such** as the **definition** of **TSP**.
- By '**an instance**' we mean a **particular input** of a problem such as a set of **cities** and the **inter-distances** for **TSP**.
- Each **problem therefore** has **infinitely many instances** just like there are **infinitely many instances** of **TSP**.
- The **problem** of **sorting** a **list** of **numbers** is **easy** to **define**. **Note** that there are **infinitely many instances** of **sorting**.
- Note **that** we are **talking about** the **sorting problem** and **not** a **sorting algorithm**. A **problem** might **have many** algorithms.

Why Polynomial Algorithms

- When we say **polynomial** we usually mean **polynomial** or **sub-polynomial** (**logarithmic**, **log-linear**, or **linear**).
- Polynomial **algorithms** are **considered tractable** or **feasible**.
- **Polynomials** are **closed** under **composition** and **addition**.
- All **digital computers** are **polynomially related**.
- This is a **very important** concept. It **means** that given a **problem instance** of **size n**, then the **actual running times** of **any two computers** will be **polynomial related**.
- If the **fastest** of the **two computers** requires **k seconds** to **solve** the **problem** then the **slower** one will **require** a **number** of **seconds** that is **polynomial** in **k**, for **example** $3k^2 - 4k$.

Types of Problems

- **Computable** problems. Those for which we can write an algorithm.
- **Non-computable** problems. Those for which we (provably) cannot write an algorithm.



Interesting Questions

- Can we **prove** that there are **problems** that are **non-computable**?
- Do **non-computable** problems exist because our **computers** are not **powerful** enough?
- Cannot we just **build** a more **powerful** computer?
- What is **computation** anyway?

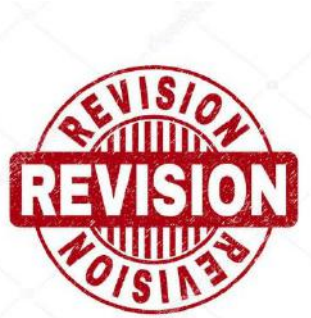
The Halting Problem

Is there a function that takes as input a program and the input (bit string) to that program, and the function determines if that program terminates on that input?

H(P,I) where P is a program and I is the input.

Posed by Alan Turing in 1936 to prove that there are ***undecidable*** problems.

Note that it is not possible to consider programs without their inputs.



Proof of Halting Problem (1)

Suppose that ***H***(***P***,***I***) exists and if ***P*** halts on ***I***

- then it returns **TRUE** (*terminate*)
- or **FALSE** (*loop forever*)

A program is a bit-string and can therefore take itself as input. A call ***H***(***H***,***H***) is therefore allowed.

Construct a new function ***K***(***P***) that calls ***H***:

- if ***P*** terminates then 'loop forever'
- if ***P*** loops forever then '*terminate*'

K(P) does exactly the opposite of what H(P,P) says!

Proof of Halting Problem (2)

The trick is now to make a call ***K***(***K***)!

This makes a call to ***H***(***K***,***K***).

- If ***H***(***K***,***K***) loops forever then ***K***(***K***) will ***terminate***.
- If ***H***(***K***,***K***) terminates then ***K***(***K***) will ***loop forever***.

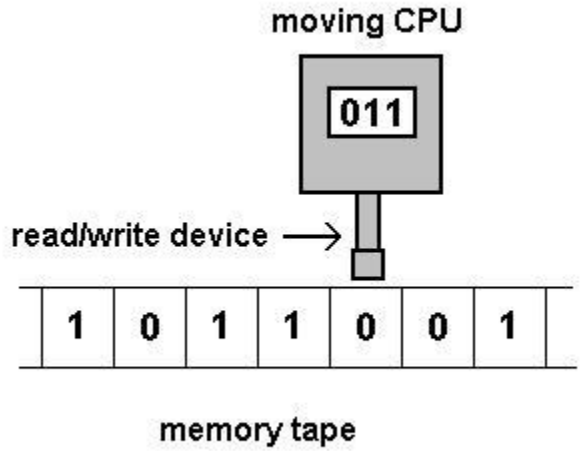
This is contrary to what ***H***(***K***,***K***) tells us and therefore is a contradiction (***reductio ad absurdum***).

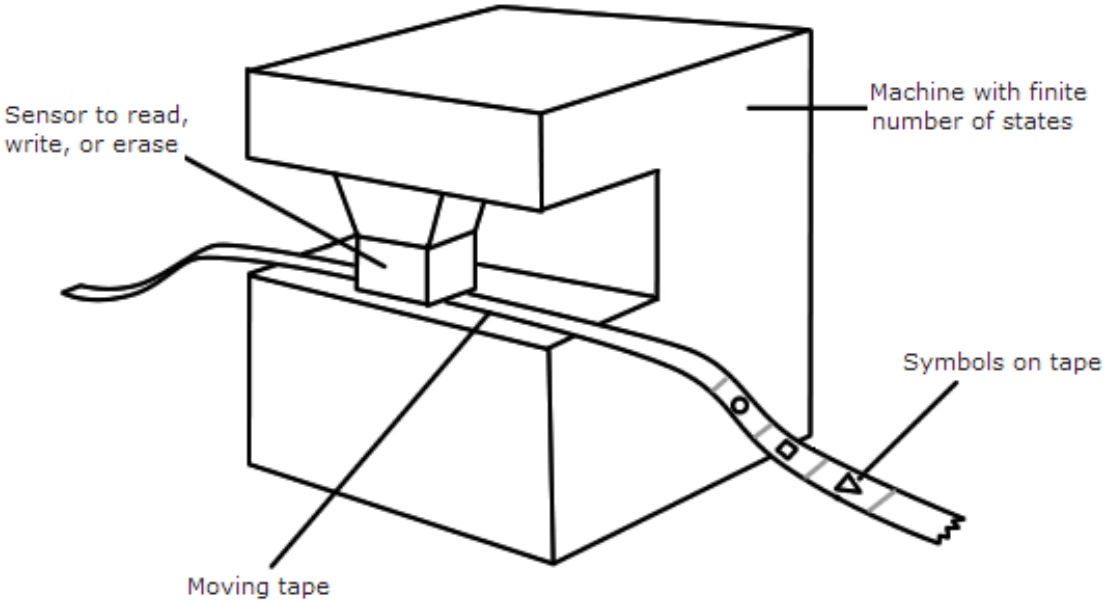
H(***P***,***I***) therefore cannot exist ■

Can we just build more powerful computers?

- Alan Turing invented the Turing Machine.
- This is a very simple device that performs computation.
- The Turing Machine is as powerful as the most powerful computer we have today.
- Nobody has managed to build a computing device more powerful than a Turing Machine.
- The Church-Turing Thesis states that this is (probably) not possible.

Turing Machine





A Turing Machine

Turing Machine (formal definition)

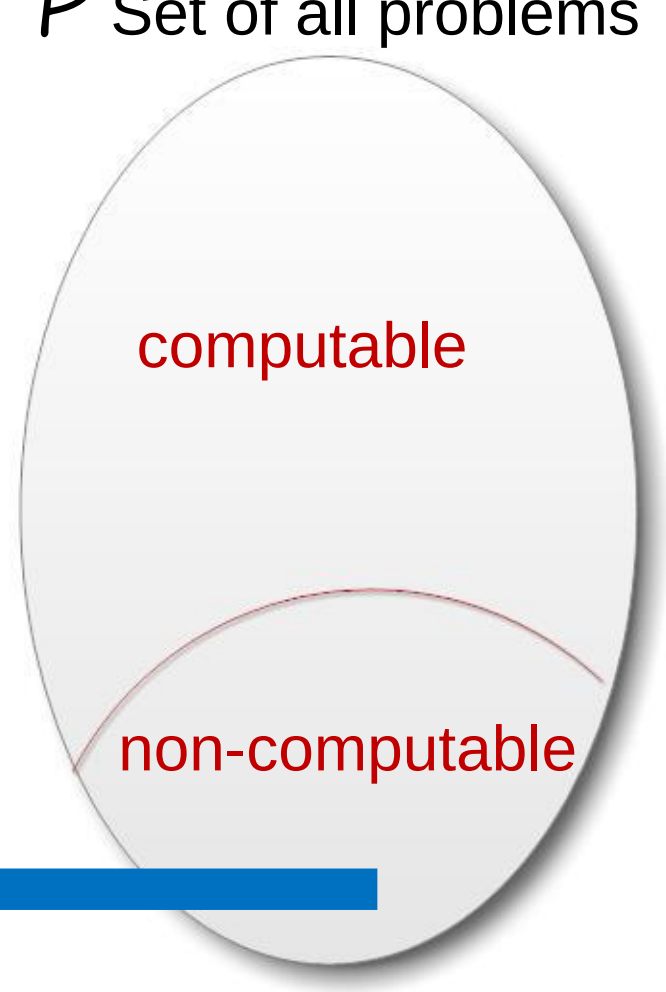
- A Turing Machine is represented by a 7-tuple $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$
- Q is a finite set of ***states***
- Σ is the ***input alphabet***, where $\square \notin \Sigma$
- Γ is the ***tape alphabet***, a superset of Σ; $\square \in \Gamma$
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the ***transition function***
- $q_0 \in Q$ is the ***start state***
- $q_{accept} \in Q$ is the ***accept state***
- $q_{reject} \in Q$ is the ***reject state***, and $q_{reject} \neq q_{accept}$

Church-Turing Thesis

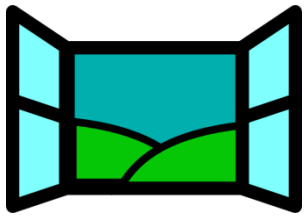
- Can be **paraphrased** as **"Anything that is 'computable' can be computed using a Turing Machine"**
- In **other words**, the **Turing Machine** is the **most powerful computing** device.
- Many **people believe** the **Church-Turing** thesis to be **true** (including **myself**).
- Many people have to tried to find other computational models: **Lambda Calculus** (**Alonzo Church**), **Post Production Systems** (**Emil Post**).
- All **turned** out **equivalent** to a Turing **Machine**!

This is not good!

- **Unless** we **find** a **model** of **computation** that is **super** Turing powerful we **might** have to **accept** that there are **non-computable** (or **undecidable**) **problems**.
- It is also **possible** that we ***may*** never **build** a **computer** more **powerful** than a **Turing Machine**.
- **Another example** of an **non-computable** problem is the **Unbounded Tiling Problem**.

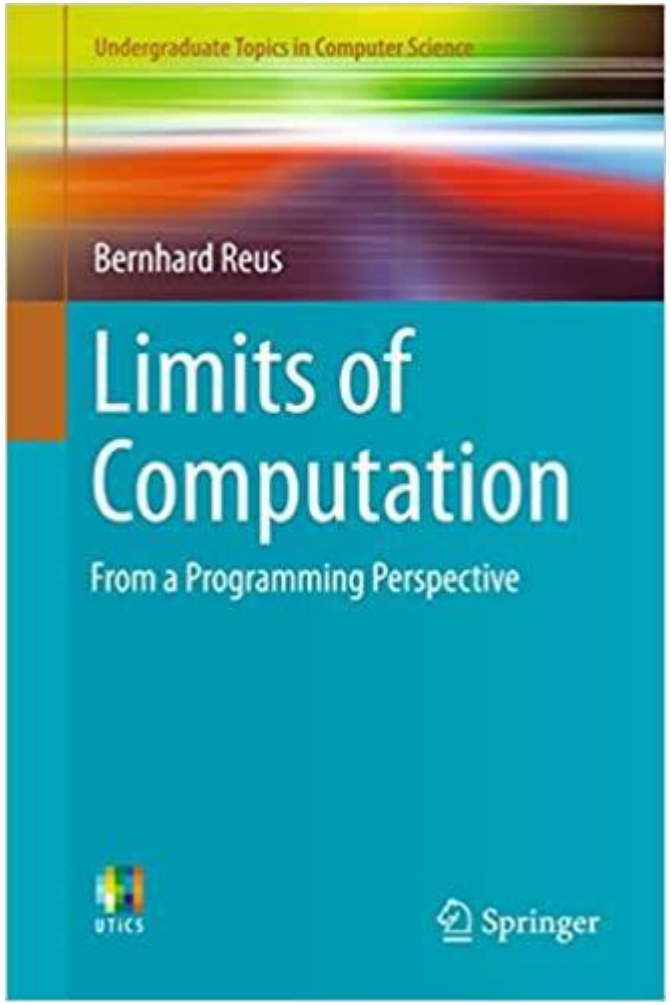


Throw them out of the window!



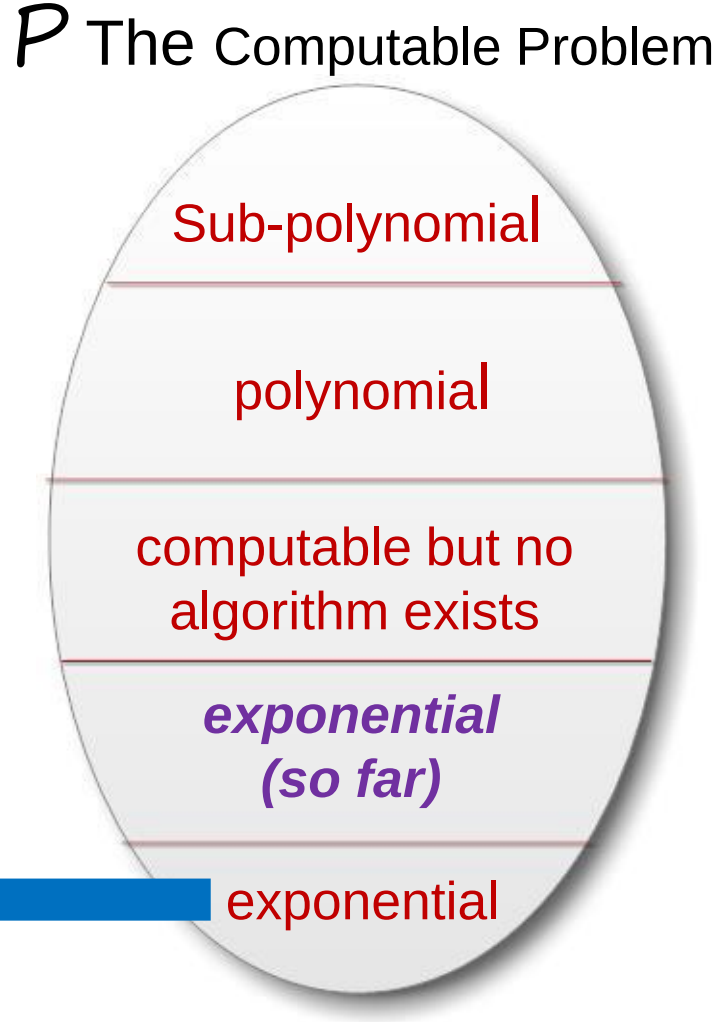
Undecidabilty

- To say that a **problem** is '**non-computable**' or '**undecidable**' **means** that there is **no known way**, even given **unlimited computing** resources and an **infinite amount** of **time**, that the **problem** can be **decided** by **algorithmic** means.
- In **other words**, you **cannot write** an **algorithm** to **solve** (**decide**) **such** a **problem**.
- All **known CPUs** and **programming languages** are **Turing powerful** and, therefore, **equivalent** (in **expressive power**) to a **Turing Machine**.
- Do **not try** to **change** the **CPU** or the **programming language**.

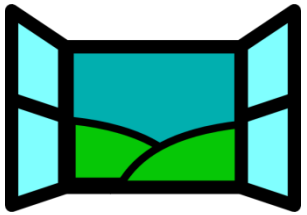


Let us Consider only Computable Problems

There are problems that are **provably exponential** and (**many**) others for which we have **only exponential** algorithms (**so far**).



Throw these out of the window too!



An Important Point

Many **problems** such as **TSP**, **time tabling**, **scheduling**, **MSA**, etc. only **have** (so **far**) *exponential* or *super-exponential* **optimal** algorithms.

However

This **does not mean** that a **polynomial** or **sub-polynomial** time **algorithm** does **not exist** for these problems. Only **that**, **so far**, one **has not** yet **been found**.

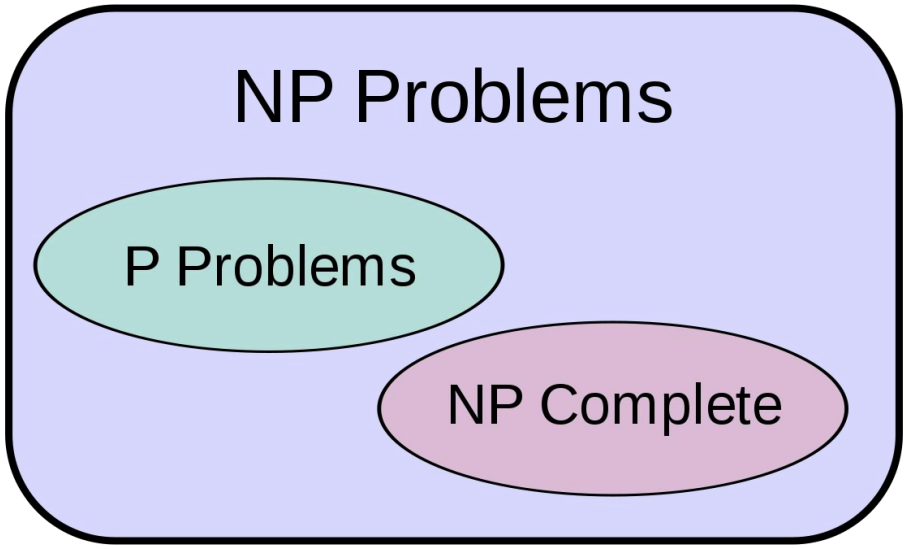
An Interesting Question

Is **there** is **polynomial time algorithm** for **TSP** and **other problems** that, **so far**, only **have exponential time** algorithms?

Answer

Nobody **really knows**. Many **people** have **tried** to find **efficient algorithms** but **failed**.

However, the **CS community** has **developed** a **wonderful theory** called the **Theory of NP-Completeness** which we **now discuss**.



Running Time Functions (RTFs)

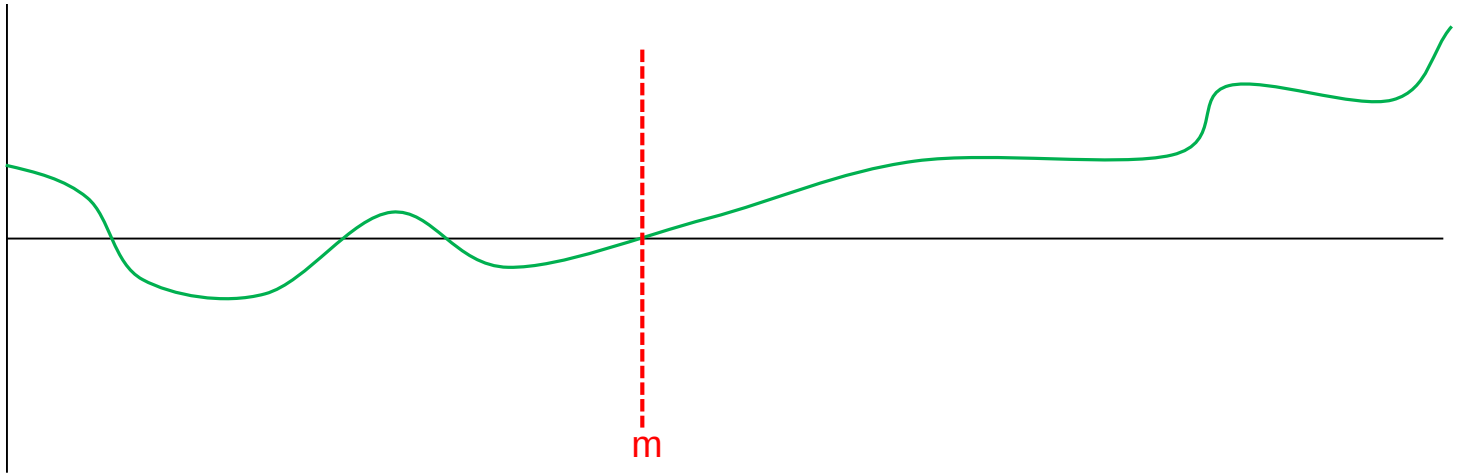
- A function f from $Z^+ \rightarrow R$ is called a **Running Time Function** (**RTF**)
  - If and only if
    - there exists a constant $m > 0$ *(depending on f)*
    - such that $f(n) > 0$ for all $n \geq m$,
    - $\exists\ m > 0$ s.t. $\forall\ n \geq m$ then $0 < f(n)$

∃ there exists

∀ for all

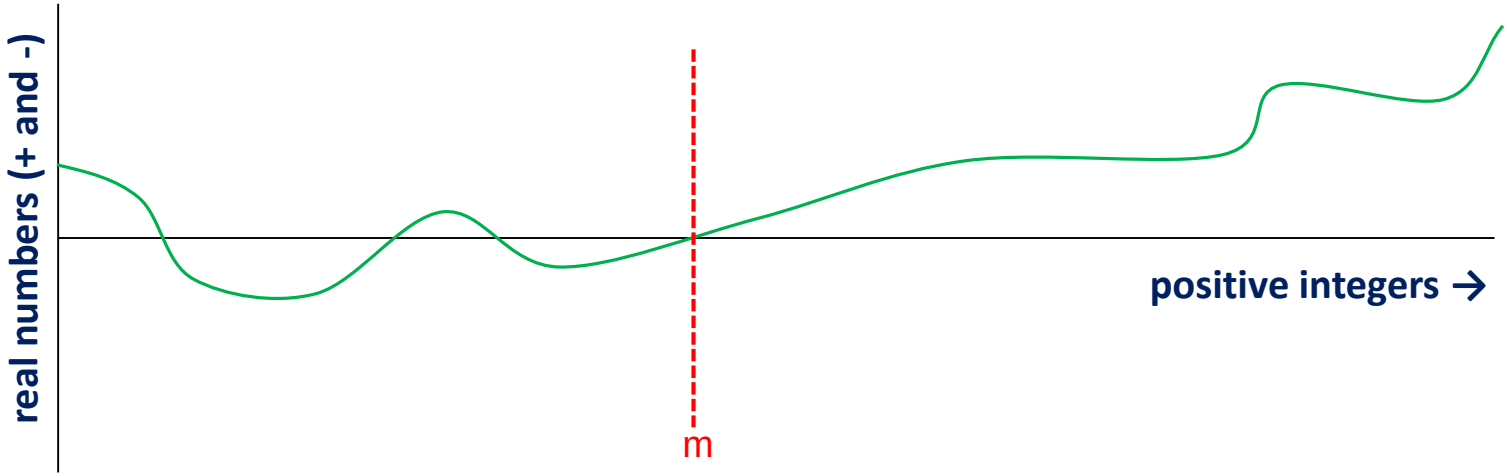
iff If and only if

s.t. such that



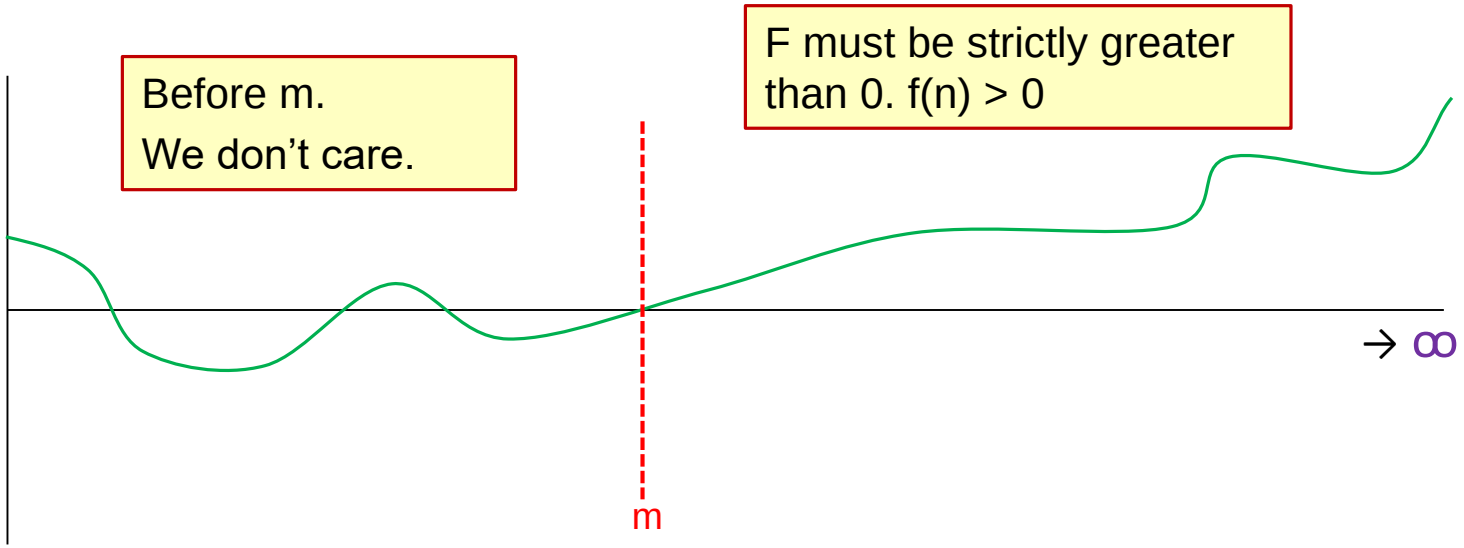
Running Time Functions (RTFs)

- **Note that** an **RTF** is **just *any function*** that **accepts** as **input** a **positive integer *n*** (**1** and **above**) and **outputs** a **real number** (**negative** or **positive**) ***f(n)*** **such that *f(n)*** is **strictly positive** after some ***m***. Each **RTF** has **its own *m***.
- **Note also**, **that** for **all** $n \geq m$, **then** $f(n) > 0$.
- This **means** that ***f(n)*** can be **negative** in the beginning but, after **some *m***, it **must** be **strictly positive**.
- The **constant *m* depends** on ***f***. In **other words**, each **running time function *f*** has its **own *m***.



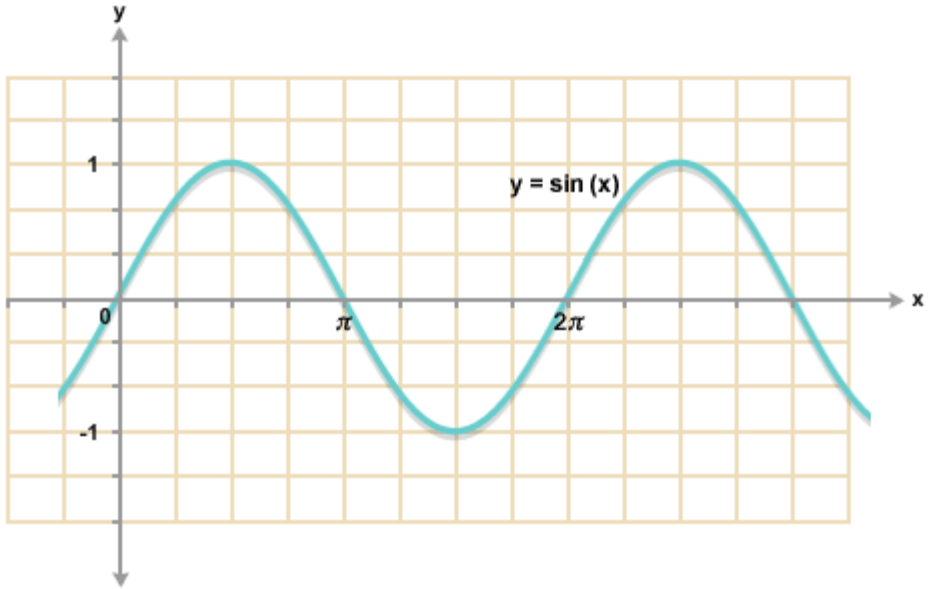
Running Time Functions (RTFs)

- RTFs are used in the study of the time complexity and the space complexity of algorithms.
- The input to the RTF is a positive integer since the number of 'steps' in an algorithm has to be a positive number.
- Note that the x axis (the domain of the function) is discrete.
- Note also that we do not care what the function f does before m as long as it is strictly positive from m onwards (towards infinity).



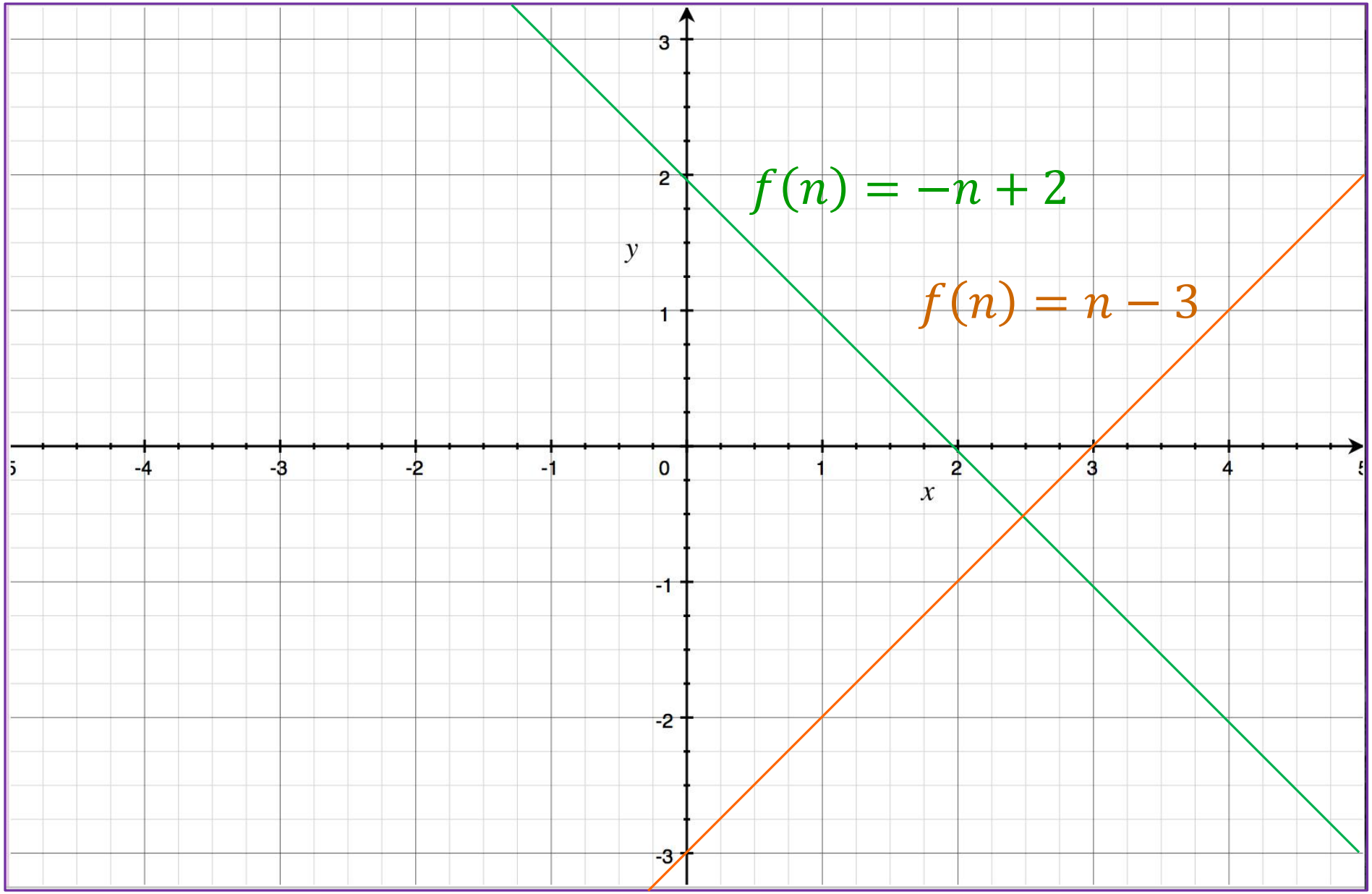
Running Time Functions (RTFs)

- The following are not running time functions:
- *sin() and cos()*
- *$-n^2$*
- *-n + 4*



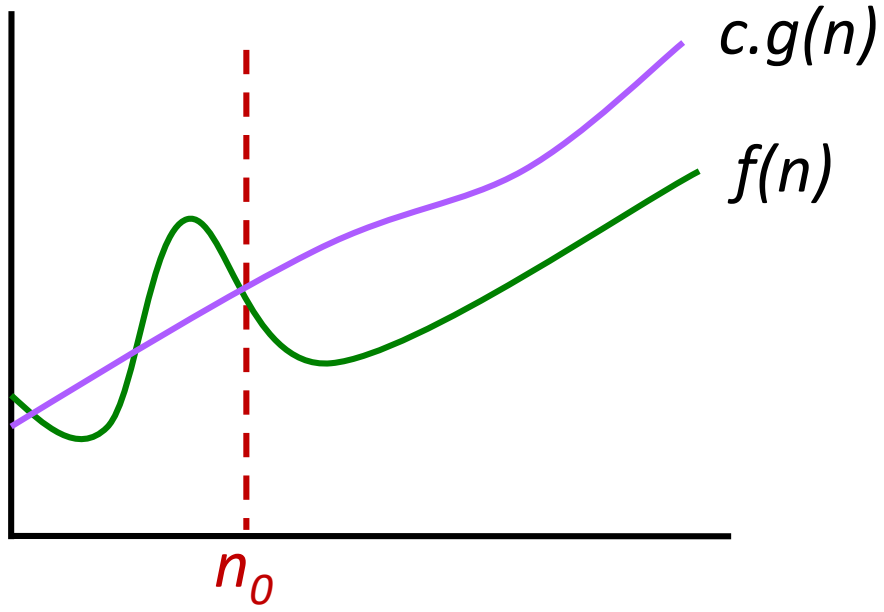
- What about:
- *tan()?*
- *sin() + 2?*
- *n – 3?*

Running Time Functions (RTFs)



Big O

- For **two** RTFs ***f*** and ***g*** we **say** that **f(n) = O(g(n))**
  - If and only if:
    - there **exist two** constants **c > 0** and **$n_0 > 0$**,
    - such that **$f(n) \leq c.g(n)$** for all **$n \geq n_0$**
  - iff **$\exists\ c, n_0 > 0$** s.t. **$\forall\ n \geq n_0 : 0 \leq f(n) \leq c.g(n)$**



f(n) is eventually **upper-bounded** by **c.g(n)**

Big O - Notes

Big O is a **relation** on RTFs.

$3n^2$ is $O(n^2)$

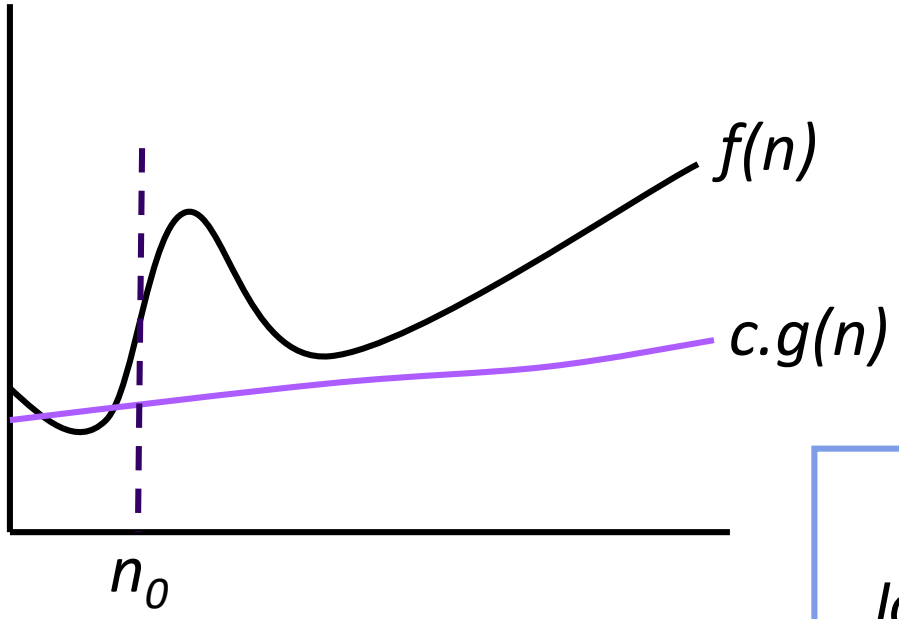
However,

n^3 is not $O(n^2)$ since we cannot find a constant c such that $n^3 \leq c.n^2$

Omega Notation

$f(n) = \Omega(g(n))$

iff $\exists\, c, n_0 > 0$ **s.t.** $\forall\, n \geq n0\,,\ 0 \leq c.g(n) \leq f(n)$

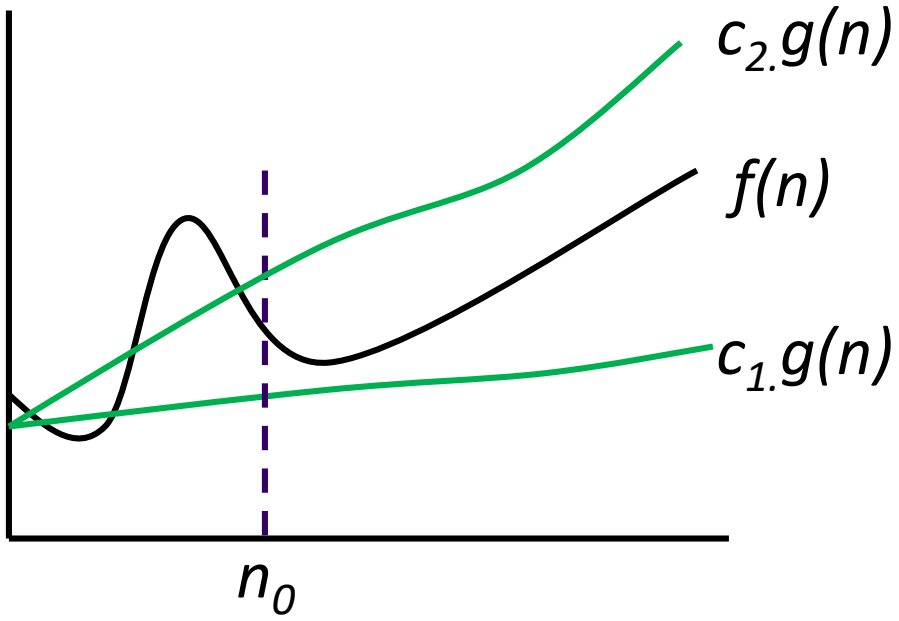


f(n) is eventually lower-bounded by g(n)

Theta Notation

$f(n) = \Theta(g(n))$

iff $\exists\, c_1, c_2, n_0 > 0$ **s.t.** $0 \le c_1 g(n) \le f(n) \le c_2 g(n), \quad \forall n > n_0$



$f(n)$ has exactly the same long-term growth rate of $g(n)$

Some Observations

Big O is used to define **upper bounds**.

Omega **Ω** is used to define **lower bounds**.

Theta **Θ** is used to **compare functions** that **grow** at **exactly** the **same** rate – have the same **order**.

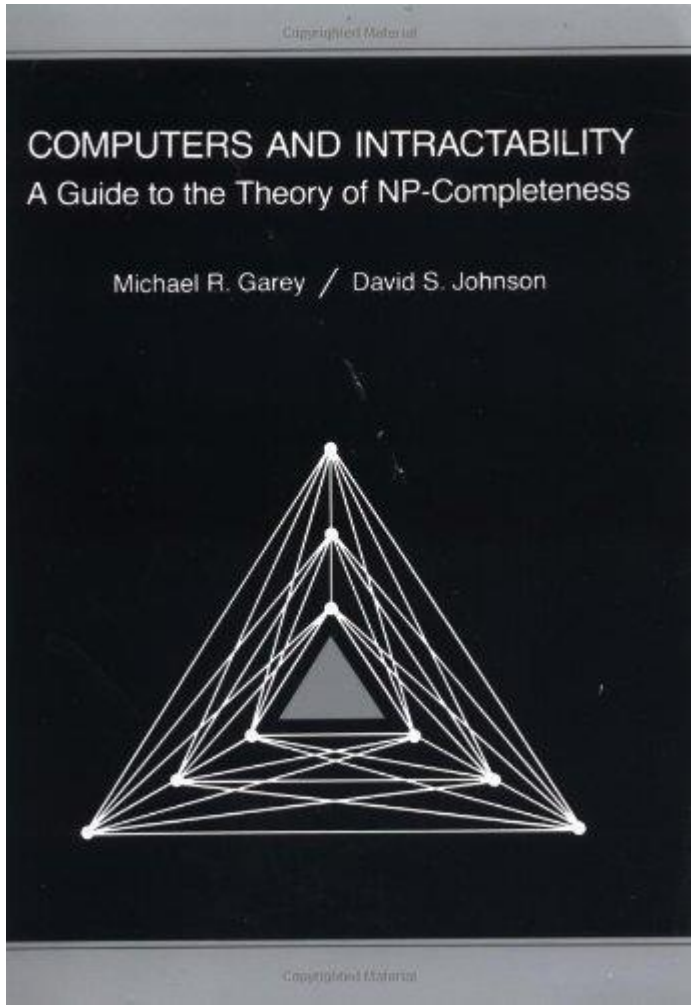
Note that Theta is an equivalence relation on the set of RTFs.

We can now proceed to NP-Completeness.

Best NP-Completeness Book

Computers and Intractability: A Guide to the Theory of NP-Completeness

by Garey and Johnson,
W.H. Freeman and Company, 1979.



NP-Completeness Theory

Before we **proceed** with a **definition** of **NP-Completeness** we first **start** with some **amusing** definitions.

These **definitions** are **very important** for **understanding NP-Completeness** and we will **revisit** these definitions **many times**.

i. **The Oracle**
ii. **Decision Problems**
iii. **Polynomial Time Verification**

The Oracle

- The **Oracle** is a **hypothetical** device. This **means** that it **does not**, and **cannot**, **exist**.
- It is often **represented** by a **large rectangular stone** with a **huge eye** on the **front**.
- The **Oracle talks** and **answers questions**.
- It **always** tells the **truth** when **asked** a **question**.
- It will **always** return the **answer** in **one time step O(1)** – no matter **how difficult** the question.
- The **Oracle** is **female**!
- The **Oracle never** gets **tired** or **angry**.

Consulting the Oracle

The Oracle

The **Oracle** quite **obviously cannot exist** but it is very **useful** in the **Theory** of **NP-Completeness**.

When **asked** a **question**, the Oracle **always returns** an **answer** in **O(1) time** as follows:

1. A **Yes** of a **No answer**. It **does not return** a **number** or **some** other **data structure**.
2. Something **else** called a **certificate**.

Decision Problems

- A **Decision Problem** is any **problem** whose **expected output** is a **1** or **0** (**Yes** or **No**).

- We **shall** see **that** most **optimization problems**, such as **TSP**, can **easily** be **posed** as **decision problems** by just **adding** an **extra parameter**.

- The **decision** problem **must** be **as hard** as the **original optimization** problem.

Decision Problems

- It is **important** to **understand** (and **remember**) that in **NP-Completeness** we **only deal** with **Decision Problems**.

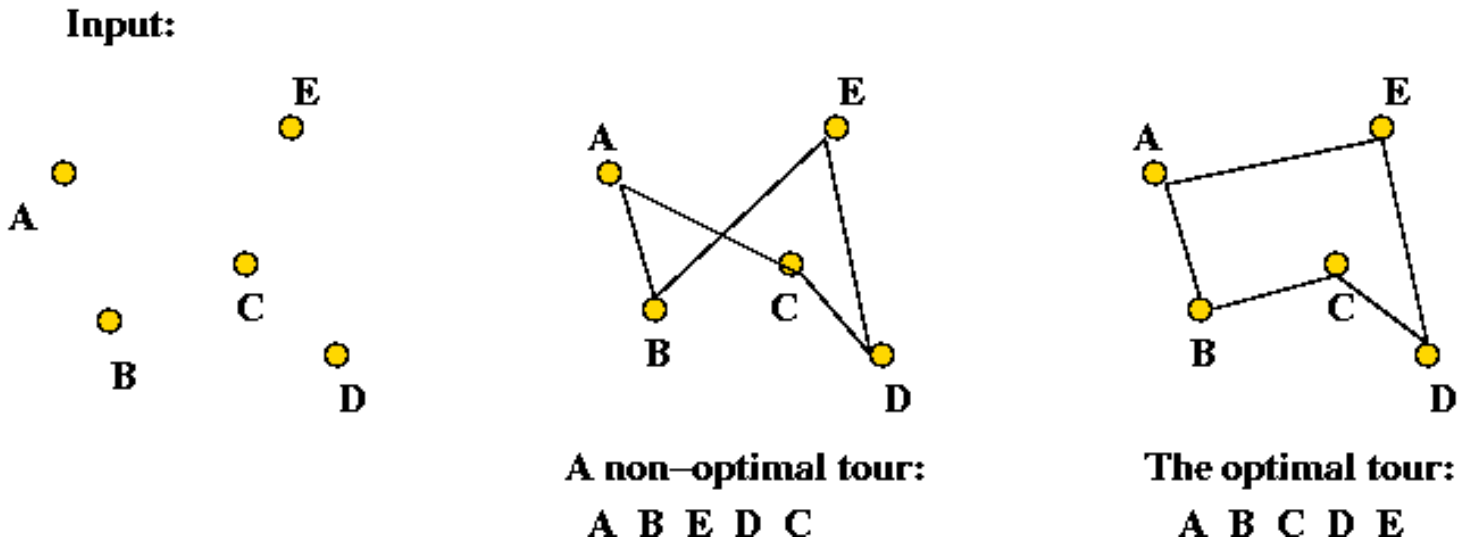
- Do not **worry** about this. All **problems**, such as **TSP**, can be **posed** as **equivalently hard decision problems** simply by **adding** an **extra parameter** - ***k***.

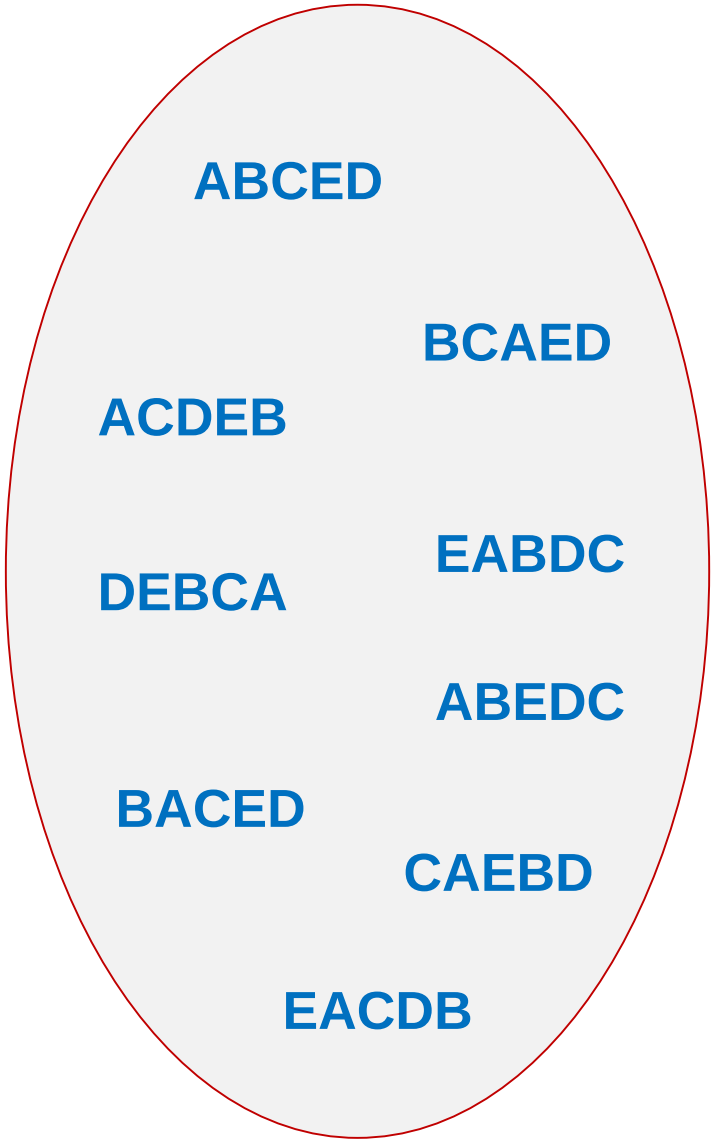
- We shall present **many examples** of how to **change** a normal **optimization problem** to a **decision** problem.

Optimization Problems

- Many **real-world** problems involve **maximizing** or **minimizing** a value.
- We normally **cannot compute** the value **directly** and **have** to **search** a **large space** of **candidate** solutions – the **search space**.
  - **Playing Chess**.
  - **Bin Packing**.
  - The **Travelling Salesman Problem**.
  - How **can** a **car manufacturer** get the **most parts** out of a **piece** of **sheet metal**?
  - How **can** a **moving company** fit the **most furniture** into a **truck** of a **certain** size?
  - How **can** the **phone company** route calls to get the **best use** of its **lines** and **connections**?
  - How **can** a university **schedule** its **classes** to **make** the **best use** of **classrooms** without **conflicts**?

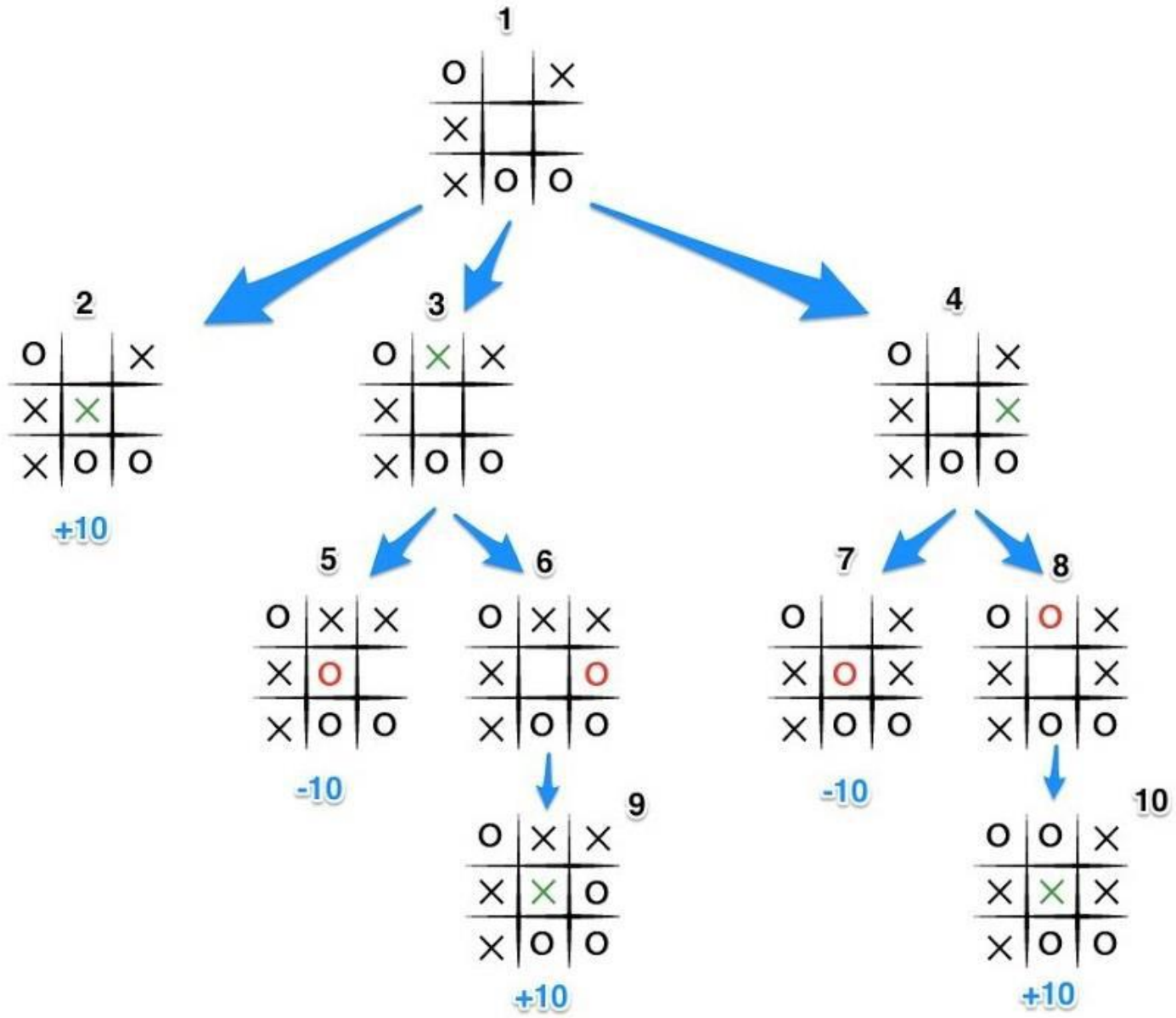
TSP Search Space



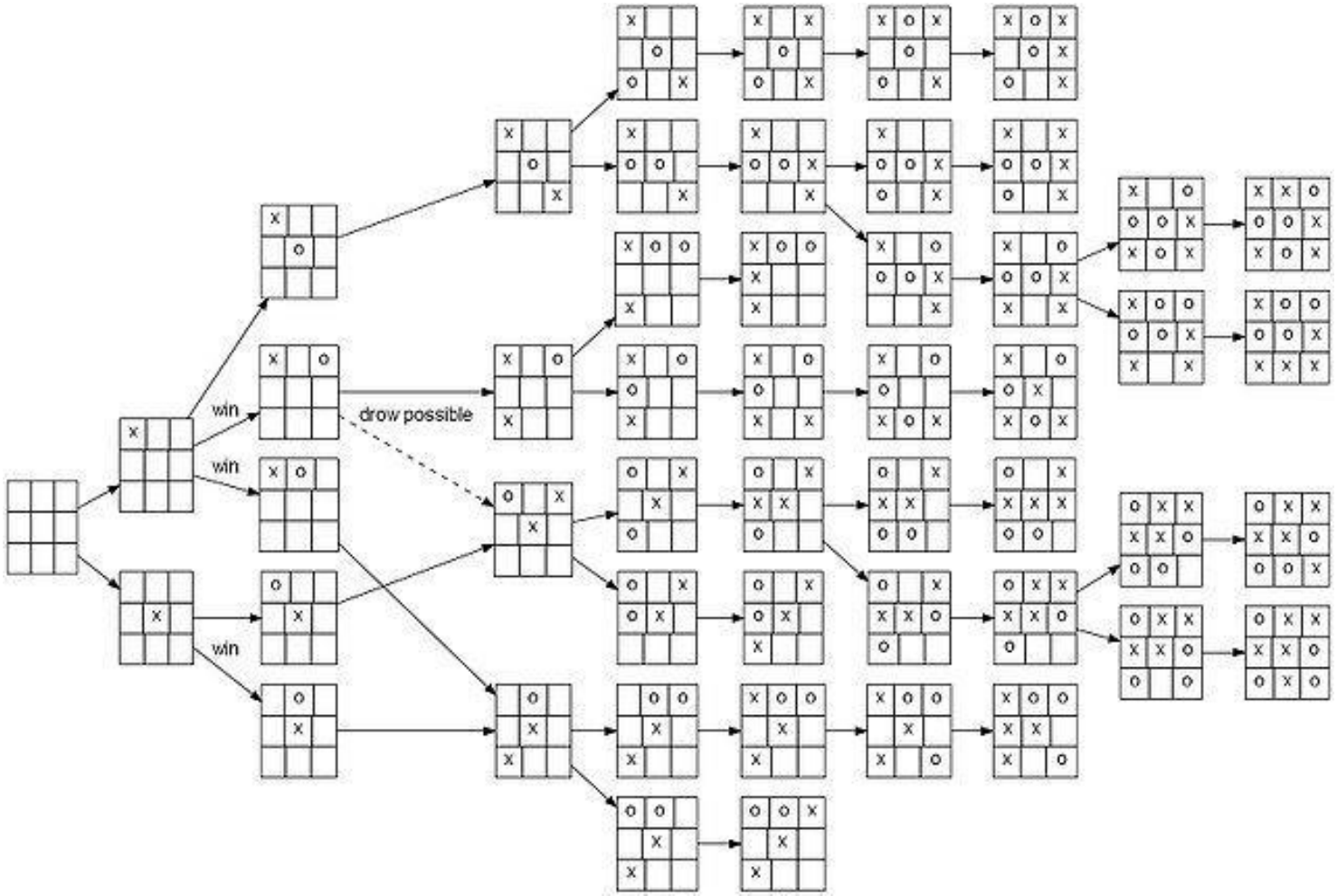


- **Search Space** is the **set** of **all permutations** of the **cities**.
- **Brute Force** algorithms **search all** of the **search space** – **even** if this is **exponential** or **super exponential** in the **size** of the **problem**.

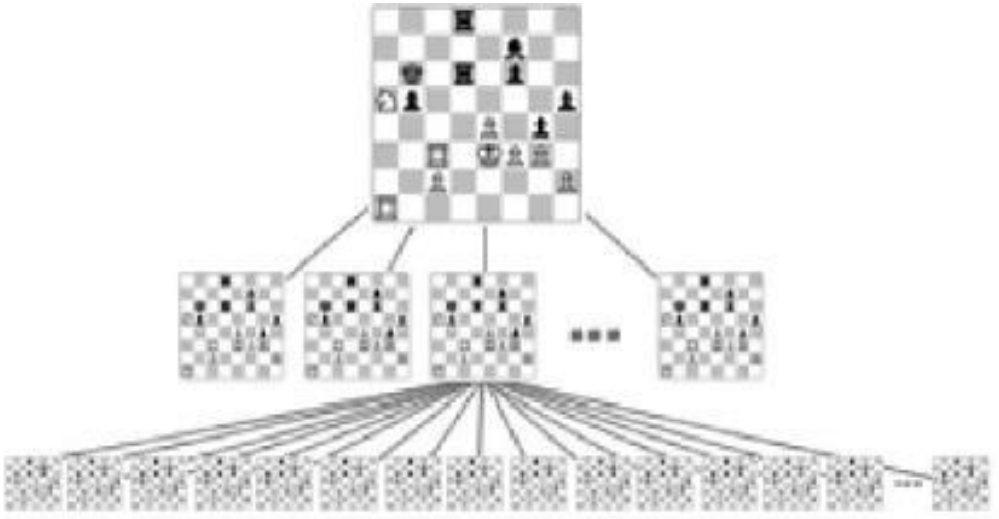
Tic-Tac-Toe (OXO) Search Space



Tic-Tac-Toe (OXO) Search Space



Chess Search Space 10^{38}



Optimal vs Approximate Solutions

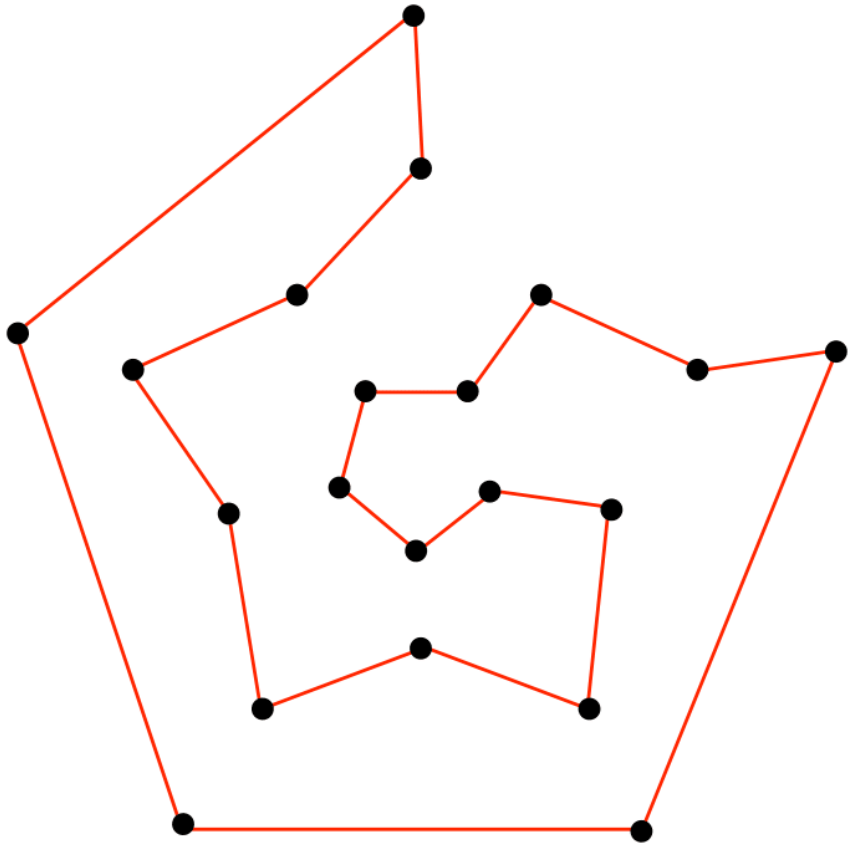
- It **appears** that **many optimization** problems, such as **TSP**, ***may*** require **exponential** time to **solve optimally** (finding the **shortest tour**). **However**, there is **no proof** yet that a **polynomial time algorithm** does **not exist** for **TSP** and **many** other **optimization** problems.
- **Often**, we **have** to **make** a **choice**.
- Do we **want** the **guaranteed optimal solution** to the problem, even **though** it might **take** a **lot** of **work**/**time** to find?
- Do we **want** to spend **less time/work** and find an **approximate solution** (which is **near optimal**)?
- For the **Theory** of **NP-Completeness** we shall **always assume** that we **have** to **find** the **optimal solution** to the problem.

Polynomial Time Verification

- Are we ***able*** to **check** if the **answer** to a **decision problem** is **true** (when Oracle answers ***yes***) in **polynomial** time?
- The **verification** time must be a **polynomial** function (or less) in the **size** of the ***input*** and in the **size** of the ***certificate*** **returned** by the **Oracle**.
- We give an **example** soon.
- Note that the '**size**' of the **certificate** itself must be **polynomial** in the **input** size ***n***.
- This is a **very important** concept in **Computer Science**.
- Can **problems** whose **answer** (**solution**) can be **verified** in **polynomial time** also **produce** an **answer** in **polynomial time**?

Some Problems and their Certificates

- The **certificate** is **essentially** the **output** of the **corresponding optimization** problem.
- Consider the **TSP decision** problem.
- Given a **weighted undirected** graph G, is there a **Hamiltonian Cycle** of **length k** or **less**?
- If the **Oracle** answers **Yes** the **certificate** is the **Hamiltonian Cycle found** by the Oracle.
- We can **quickly check** if the **Hamiltonian Cycle** in the **certificate** is **indeed less than** or **equal** to **k**.



Important Note re TSP

- For the **normal TSP problem** we have to **find** the **shortest Hamiltonian Cycle**. This is an **optimization problem**.
- For the **TSP Decision Problem** we have to **find** a **Hamiltonian Cycle** with **length less** than some ***k***.
- Both are **equally hard**.
- Note that, in **general**, finding a **single Hamiltonian Cycle** is **hard** let **alone** finding **all** the **Hamiltonian Cycles** and then **returning** the **shortest** one.
- By **hard** we mean **computationally hard**. **Writing** an **algorithm** to **solve** TSP is **trivial**. The **problem** is that **all** these **trivial algorithms** will **run** in **exponential time**.

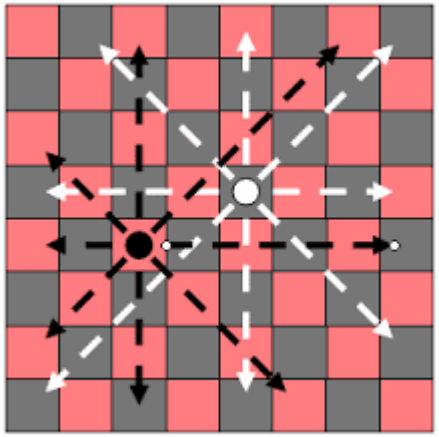
Some Problems and their Certificates

What is the **certificate** for:

- **Sorting**?
- **N-Queens** Problem?
- **Scheduling** and **Timetabling**?

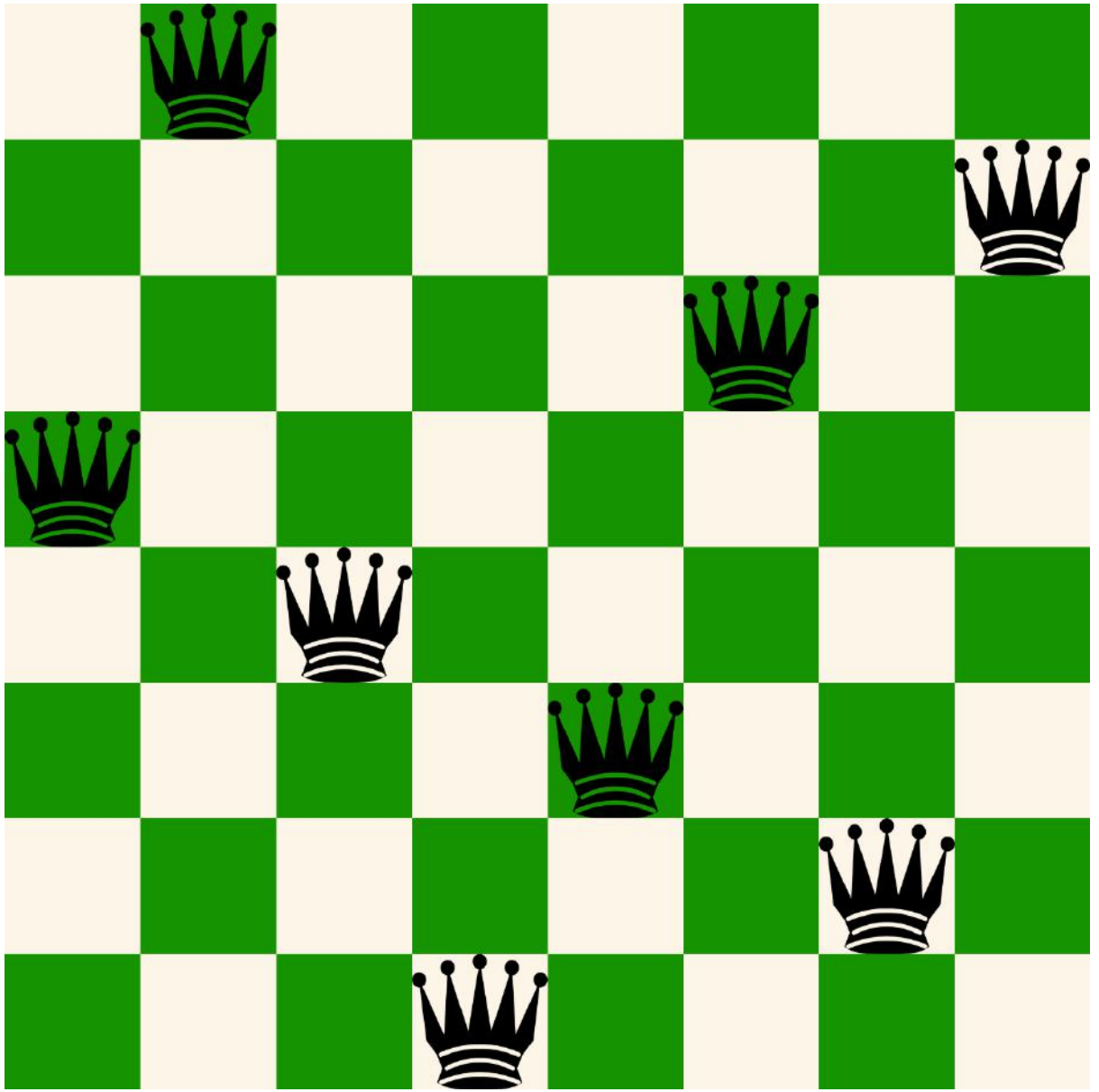
Polynomial time verification '**forces**' the Oracle to **find** a **solution** (**output**) for the **problem** where the **size** of the **solution** is **polynomial** in the **size** of the **input** and also **that** the **solution** can be **polynomially verified**.

n-Queens Problem Completion



The ***n*-Queens Problem** is the **problem** of **placing *n* chess queens** on an ***n* × *n*** chessboard so that **no two** queens can **capture** each other.

We can **verify** the solution **very quickly**.



The Class P

This is **the** set of **all problems** which can be **solved** by a ***deterministic polynomial-time algorithm***.

- **Sorting**.
- **Tree Traversal**.
- **LCD**.
- **Pairwise Sequence Alignment**.
- Building a **MST**.
- **Multiplying** 2 numbers.
- Plus **many** others.

The Class NP

This is the **set** of **all problems** which:

- ***Can be changed to (posed as) a Decision Problem.***
- ***Passed to the Oracle. This will return an answer in O(1) time and a certificate.***
- ***The certificate from the Oracle must be verified in polynomial time.***

If a **problem *X* satisfies** the **above** then **it is** in **NP**.

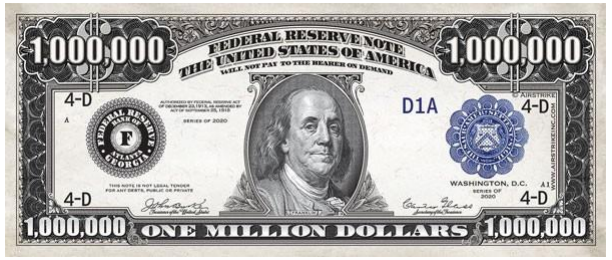
Formally, we **say** that $X \in NP$.

Showing that a **problem** is **in NP** is **usually** very **easy**.

Showing that a **problem** is **NP-Complete** is **more difficult**.

Problems in **NP** can be **solved** in **polynomial time non-deterministically using** an **Oracle**.

Can **all problems** in **NP** also be **solved** in **polynomial time** using a **deterministic** algorithm? Is $P = NP$?



The Class NP

The **answer** from the **Oracle** is **essentially**:

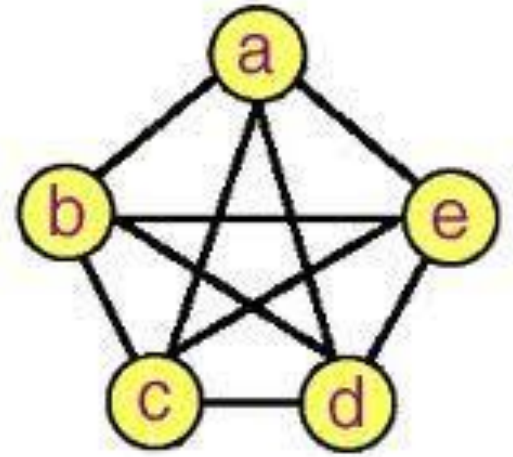
- A **Yes** or a **No**
- A **certificate**

The **certificate** must be **verified** in **polynomial time** in the **size *m*** of the **certificate itself** and in the **size *n*** of the **input** to the **decision problem**.

The **class NP** is **therefore exactly** those **problems** whose **answers** (**certificates**) can be **verified** in **time polynomial** in the **size** of the **input** and in the **size** of the **certificate**.

Decision Problem – Example
The Travelling Salesman Problem TSP

- Instance: a complete ***weighted*** undirected graph G=(V,E) (all weights are non-negative).
- Problem: to **find** a Hamiltonian cycle (tour) of minimal cost.



Change to a **decision problem** as follows:

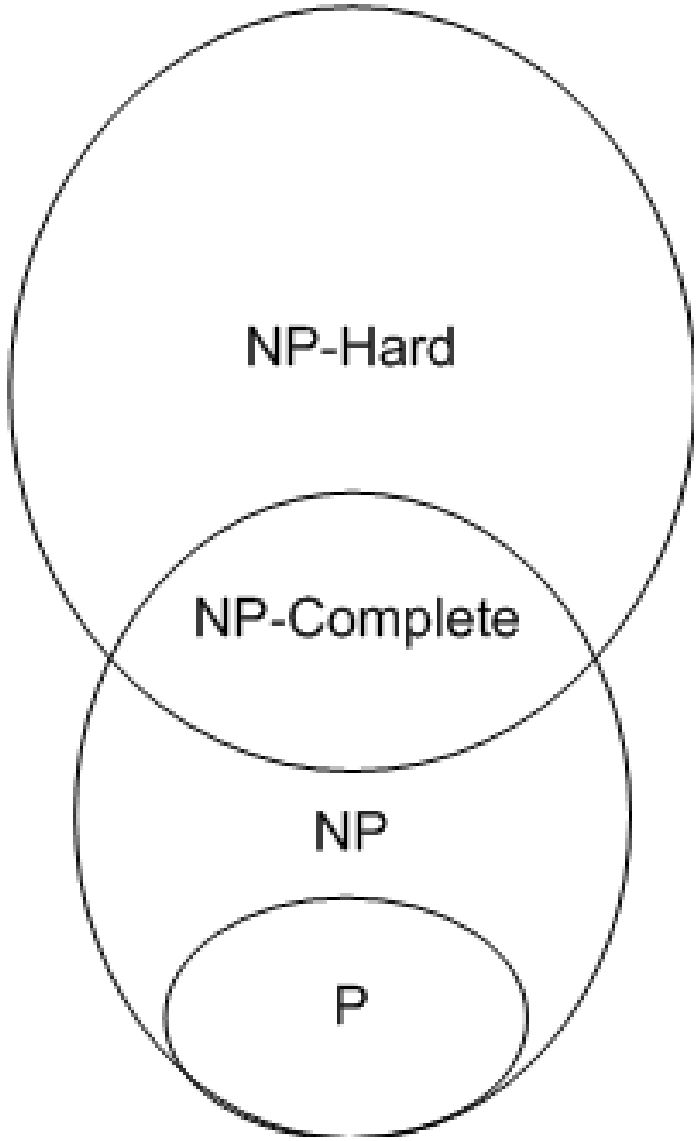
- Problem: Is there a Hamiltonian cycle (tour) of length *k* or less?

Class NP – Example
The Travelling Salesman Problem TSP

- We **give** the **instance** (***cities***, ***inter-distances***, and ***k***) to the **Oracle**.
- **Suppose** that the **Oracle** answers ***yes*** and **gives** us a **tour** (in the **certificate**).
- We can **check** in **polynomial time** if the **tour** is **less** than ***k*** by **adding** all the **distances between** the **cities** in the **tour**.
- Therefore ***TSP ∈ NP !***
- *Showing that problems are in NP is (usally) easy and straightforward.*

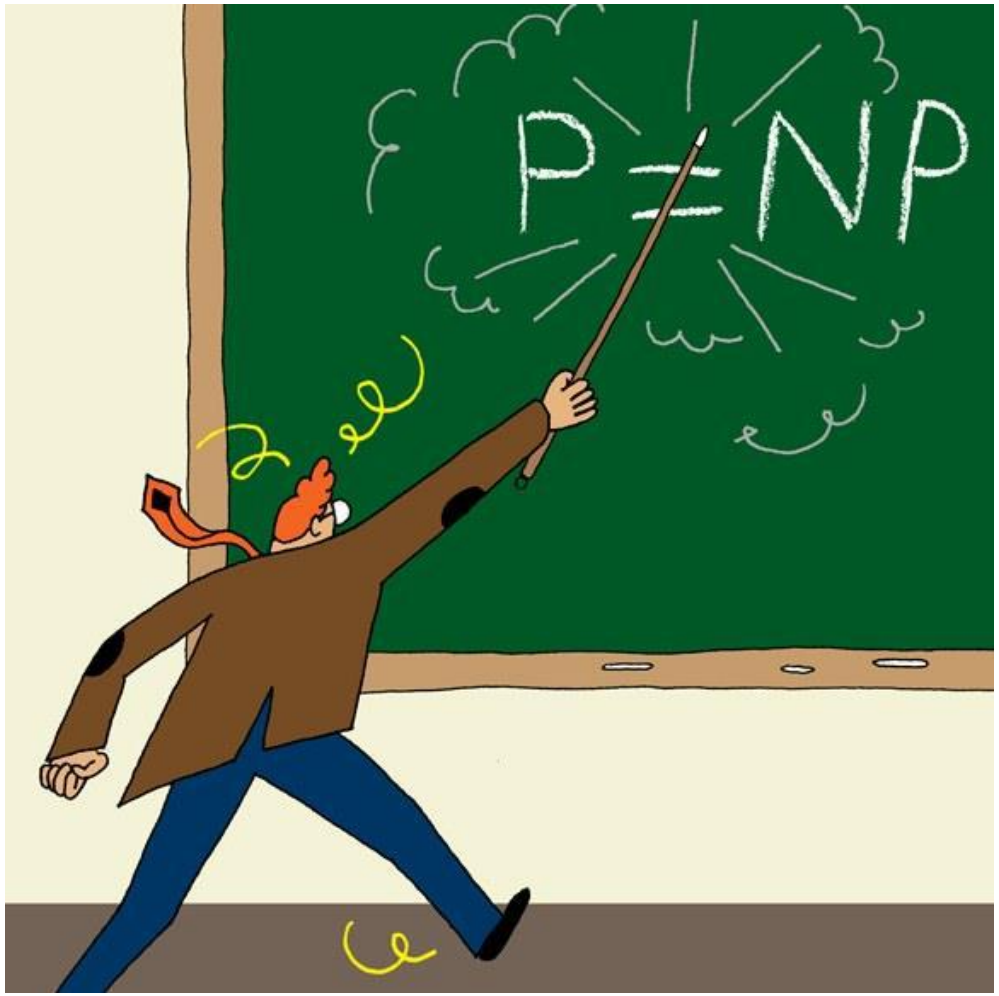
The Class NP

- NP means ***Non-Deterministic Polynomial***.
- This is **because** all the **problems** in **NP** can be **solved** in ***polynomial time*** using a ***non-deterministic machine*** (the **Oracle**).
- **Of course, the Oracle does not exist but we shall use the Class NP to prove some wonderful results.**



The Million Dollar Question

Is P = NP?



The Classes P and NP

- It is **clear** that the **class P** is a **subset** of the **Class NP**. **Why**?
- **The million dollar question, literally, is whether P = NP**.
- The ***Clay Mathematics Institute*** has **offered** a **million US** dollars **prize** to **anyone** in the **World** who can **prove** that **P = NP** or that **P ≠ NP**.
- **Many**, **many**, **people** have **tried** (and **failed**) so **far**.

CLAY MATHEMATICS INSTITUTE

The Millennium Problems

The $1M Questions

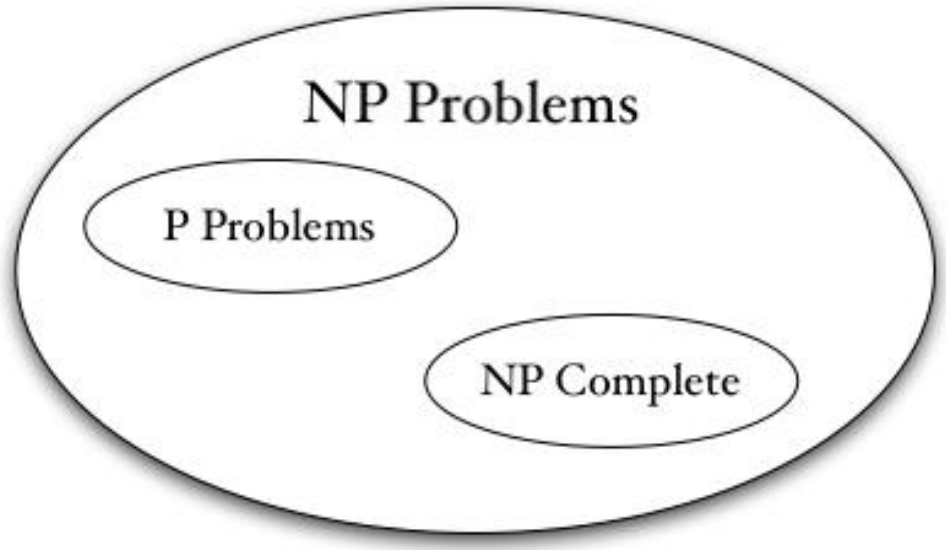
The Clay Mathematics Institute
Millenium Prize Problems

1. Birch and Swinnerton-Dyer Conjecture
2. Hodge Conjecture
3. Navier-Stokes Equations
4. P vs NP
5. Poincaré Conjecture ← solved!
6. Riemann Hypothesis
7. Yang-Mills Theory

CLAY MATHEMATICS INSTITUTE

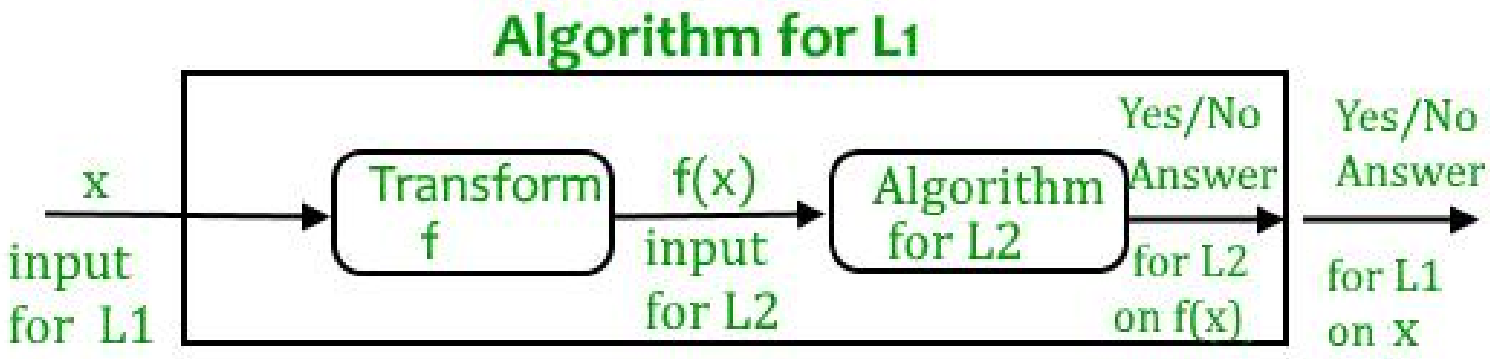
The Class NP

- It is **believed** that ***P*** is a **proper subset** of ***NP***.
- There are **many** (**thousands**) of **problems** that **are** in ***NP*** but **cannot**, **so far**, be **shown** to **be** in ***P***.
- In **other words**, there are **problems** for which we can ***verify*** the **answer** in **polynomial time** **but** for **which** we **cannot** ***compute*** the **answer** in **polynomial time**.
- These **problems** include **TSP**, **SAT**, **Vertex Cover**, **MSA**, etc.
- **What** do **we do**? Give **up**?



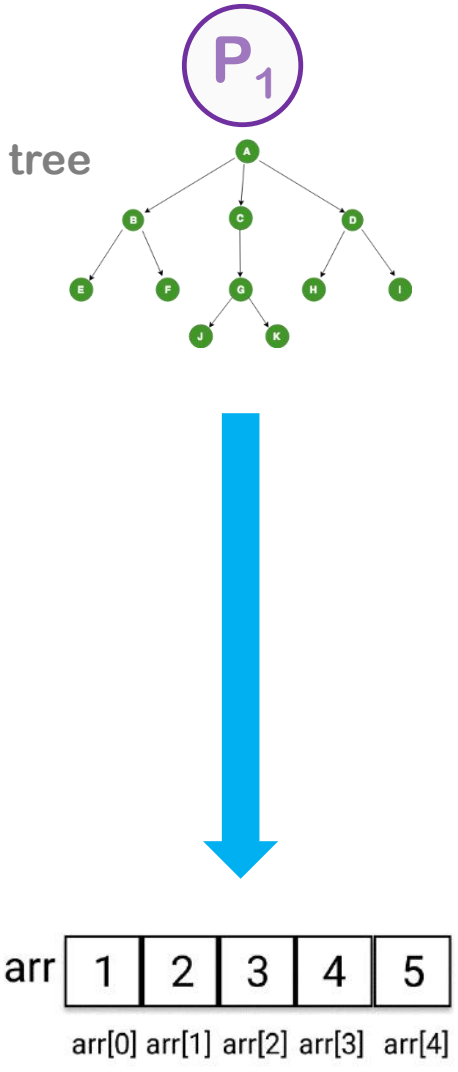
Polynomial Time Reductions

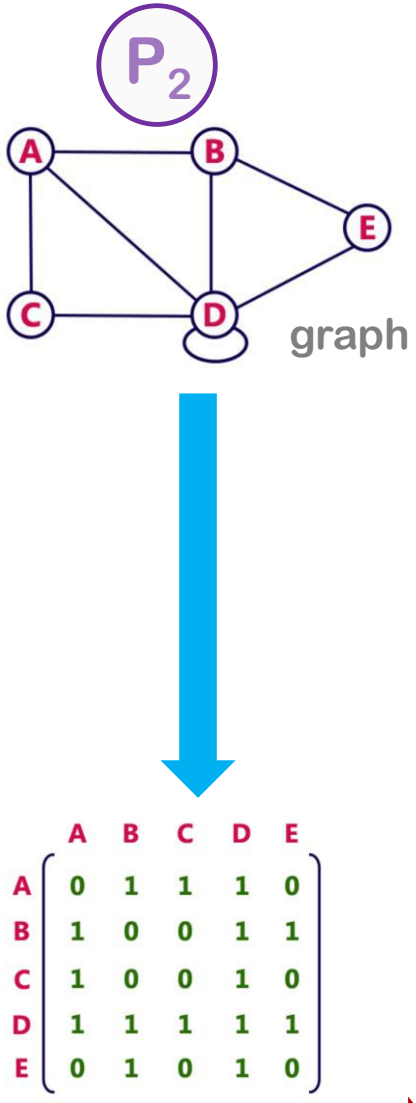
- **Suppose** we **could take** an **instance** I_1 of a **decision problem** P_1 and **transform** (**reduce**) it to an **instance** I_2 of **decision problem** P_2 such **that** if we **then solve** P_2 we would **also** have **solved** P_1.
- If this can be **done** in **polynomial time** it is called a **Turing Reduction** and **denoted** by the **symbol** $\propto$.
- We **write** $P_1 \propto P_2$.



Polynomial Time Reductions

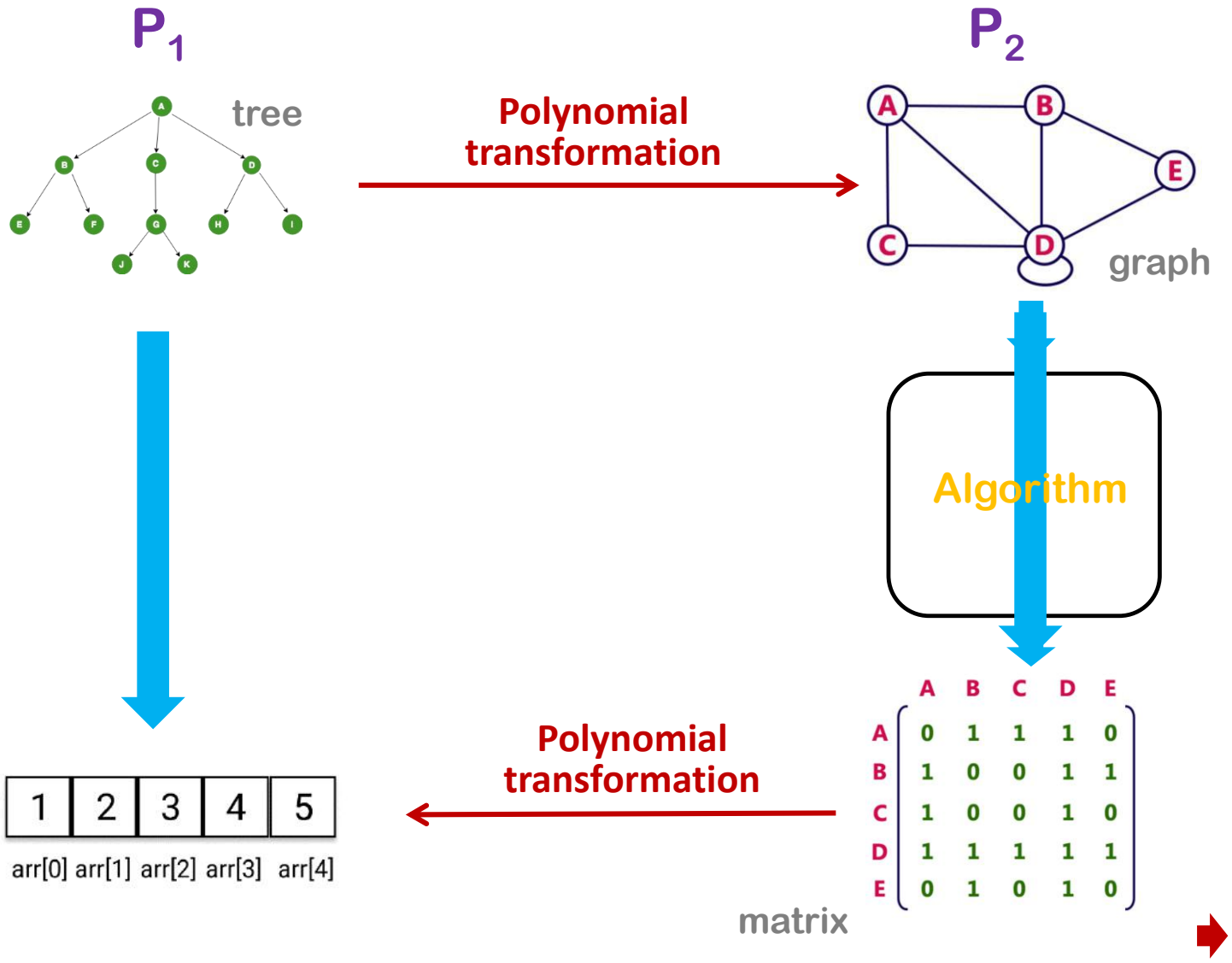
- **Suppose** we **have 2 problems** P_1 and P_2.
- At this **point** it is **not important**, actually **irrelevant**, **whether** or **not** we **have algorithms** to **solve** these **problems**.
- For P_1 the **input** is a **tree** and for P_2 the **input** is a **graph**.
- The **output** for P_1 is an **array and** the **output** for P_2 is a **matrix**.
- These are **two completely different**, **unrelated**, **problems** with **different inputs** and **outputs**.
- **Question:**
  Can we **use** P_2 to **solve** P_1?





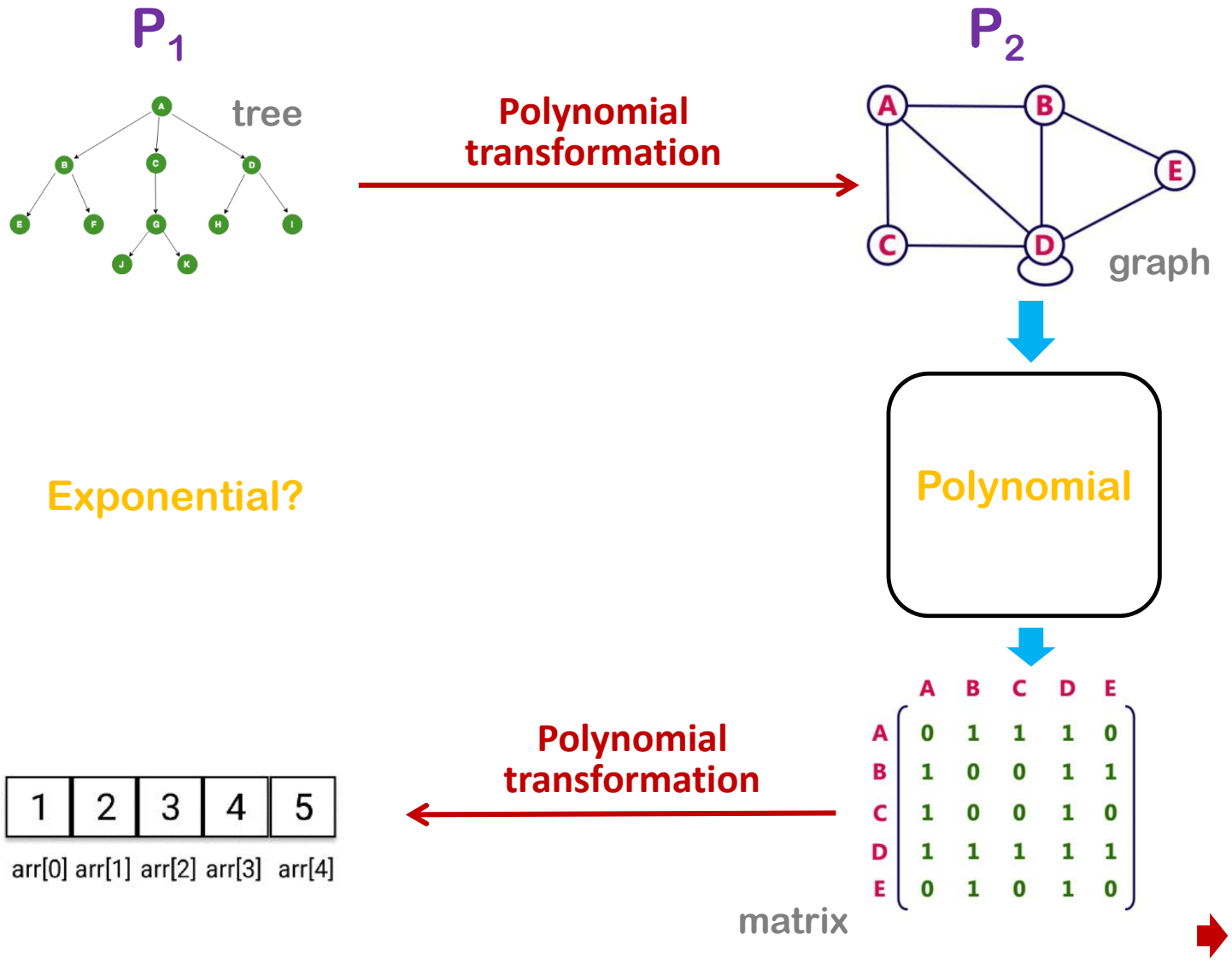
Polynomial Time Reductions

- **Suppose** we **could transform** the **input** of P_1 **into** the **input** of P_2.
- And **also** be **able transform** the **output** of P_2 into the **output** of P_1.
- We **do this** in **such** a **way** so **that** if we **use** an **algorithm** for P_2 to **produce** the **output** we **can** then **transform** the **output** of P_2 **into** the **output** of P_1.
- In this **way** we **would have solved** P_1 by **using** P_2.
- We **can** do **all this** without **even having** an **algorithm** for P_1.



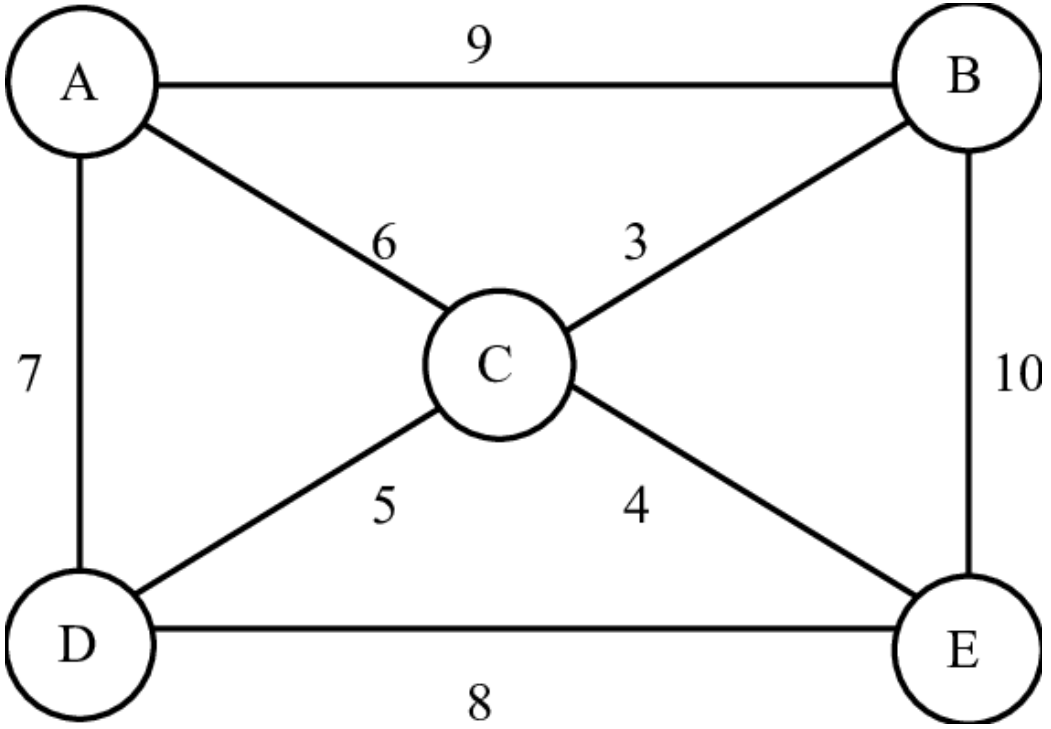
Polynomial Time Reductions

- Now **suppose** that we **have** an **algorithm** for P_2 that **runs** in **polynomial** time.
- Can we **then claim** that P_1 can only **have** an **exponential algorithm**?
- This will **lead** to a **contradiction**.
- If P_2 has a **polynomial time algorithm** then **so does** P_1.
- We **say** that P_1 is **not asymptotically harder** than P_2.
- In this **case** the **only asymptotical classes** we **consider** are **polynomial** and **super-polynomial**.



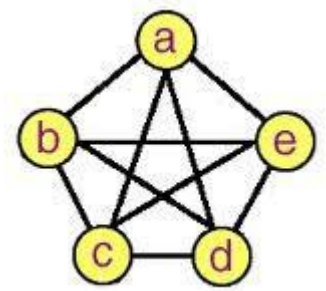
A Stupid Example to Illustrate

- **Suppose** P_1 is **TSP** and the **instance** I_1 is a **set** of **five cities** *a,b,c,d,e*.
- **Suppose** that P_2 is the **problem** of **sorting** a **list** of **strings**.
- Let us **transform** an **instance** I_1 into the **instance** I_2.



Stupid Example of Turing Reduction

- **Create** a **list** of **all permutations** of the **cities** together with the **length** of **each tour**.
- **Sort** the **list** by **length** (this **takes** $m.log_2 m$ **time**) where m is the **number** of **city permutations**.
- We have **solved TSP**!
- Not **only** that, we **have solved TSP** in **polynomial time**.



abcde,12
abced,34
abdce,9
abdec,14
abecd,22
abedc,19
acbde,11
acbed,39
acdbe,10
acdeb… etc.

The problem is that creating the list of the permutations of the cities takes ***n!* time** where n is the number of cities.

This is therefore **not** a Turing Reduction.

A Turing Reduction **must** be a polynomial time transformation.

Reasonable Encodings

- The previous example was '**stupid**' because it is **not reasonable** to define a **TSP** instance by the set of **all permutations** of the **cities**.
- All **reasonable** encodings of **instances** of a problem are ***polynomially*** related.
- The normal **adjacency list** encoding of a **graph** is a **reasonable** encoding.
- Using '**unreasonable encodings**' allows one to '**cheat**'.

Reasonable Encodings

Suppose we have two different encodings for the input to a given problem ***X***.

For a given input size for ***X***:

- Encoding ***A*** requires ***n*** bits, and
- Encoding ***B*** requires $3n^2-6n$ bits.

These encodings are ***polynomially*** related.

Suppose if:

- Encoding ***A*** requires ***n*** bits, and
- Encoding ***B*** requires 2^n bits?
- Note **that** have a '**larger**' **input** size **makes** an algorithm **appear faster**.

Turing Reductions

- **Surprise**!
  In general, **Turing Reductions** are **possible**.
- If a problem **P_1** can be **Turing reduced** to another problem **P_2** then **what** does this **tell us**?

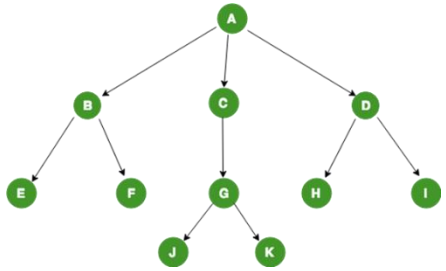
If $P_1 \propto P_2$ then P_2 is at least as hard as P_1.

Why?

We will look at a number of important Lemmas.

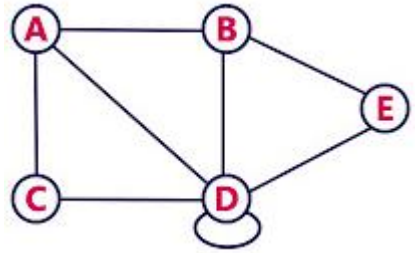
Turing Reductions

Problem *A*



$\propto$

Problem *B*



Algorithm for Problem *B*

	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	1	1
E	0	1	0	1	0

$\propto$

Solution for Problem *A*

$$\begin{bmatrix} 17 \\ 56 \\ 25 \\ 45 \\ 16 \\ 29 \\ 64 \\ 21 \\ 15 \\ 83 \\ 95 \\ 98 \\ 85 \\ 107 \\ 07 \\ 78 \end{bmatrix}$$

Important Lemmas

Suppose $P_1 \propto P_2$

Lemma 1

If P_2 **can** be **solved** in **polynomial time** then so **can** P_1.

Lemma 2

If P_1 **cannot** be **solved** in **polynomial** time than **neither** can P_2.

Important Lemmas

Lemma 3

Suppose $P_1 \propto P_2$ and $P_2 \propto P_3$ then $P_1 \propto P_3$

The above shows that **Turing Reduction** is a **transitive relation**.

Easy and Hard Problems

- In the **context** of **NP-Completeness**, when **comparing two problems** P_1 and P_2 we **say** that:
  - P_1 is **easier than** P_2 if P_1 **belongs** to a **lower asymptotic class** than P_2.
  - Likewise, we **say** that P_1 is **harder** than P_2 if P_1 **belongs** to a **higher asymptotic class** than P_2.
- The only **asymptotic classes** we are **interested** in are the **ones** on the **right**.
- The **order** of the **polynomials** is **not important**. All **problems** with **polynomial algorithms** are equally '**hard**' or '**easy**'.
- We are only **interested** when **problems** are in **different asymptotic** classes.
- The **above** is, **perhaps**, an **over simplification** of the **actual case** but it will **help** you **understand** what is **meant** by '**easy**' and '**hard**'.
- Note that, when **estimating** the **time** (and **space**) **complexity** of **algorithms** we **consider** many **more asymptotic classes**.

Polynomial

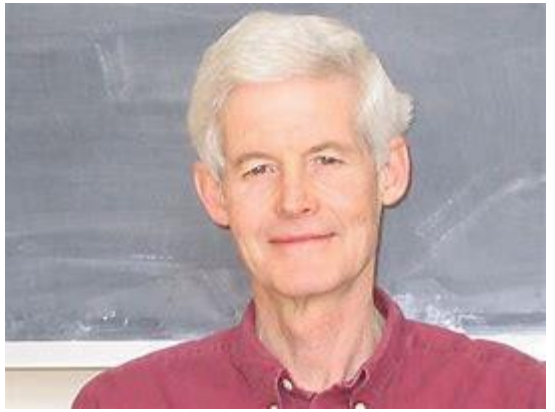
- Constant ≈ 1
- Logarithmic ≈ $\log_2 n$
- Linear ≈ n
- Log Linear ≈ $n \log_2 n$
- Quadratic ≈ n^2
- Cubic ≈ n^3

Super polynomial

- Exponential ≈ 2^n
- Super Exponential ≈ $n!$

NP-Complete Problems

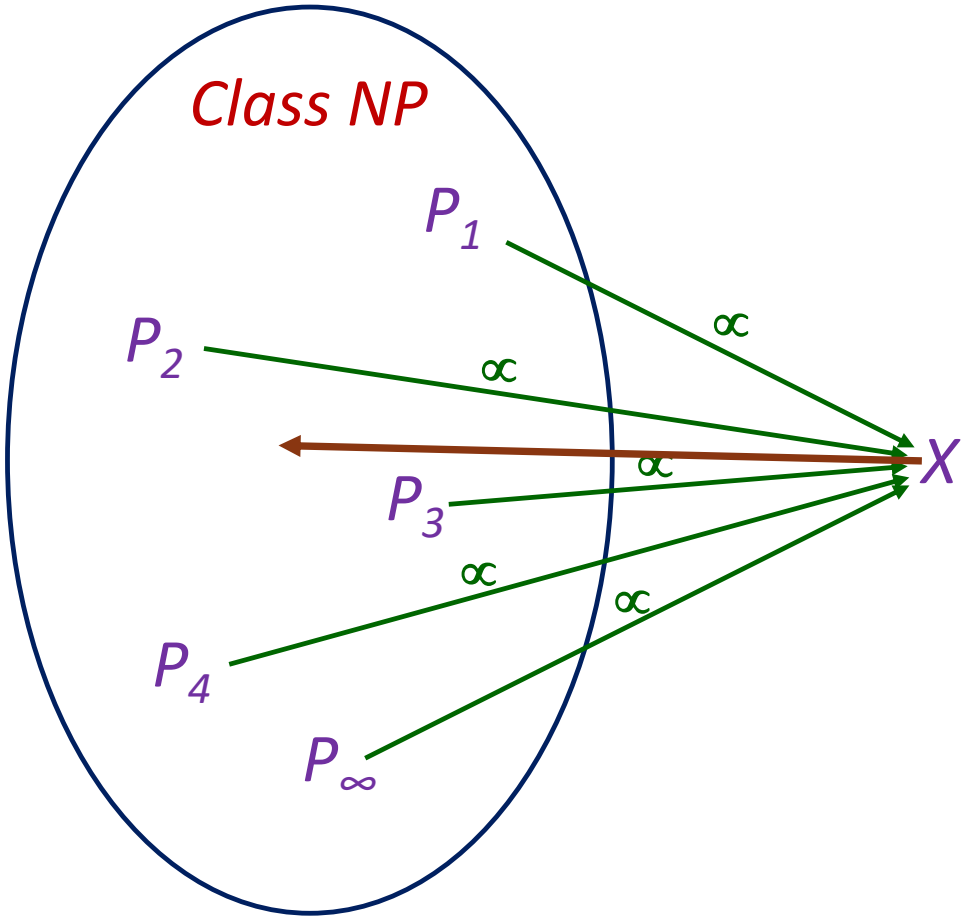
- We have seen examples of problems for which we (the computer science community) have not been able to find efficient (polynomial) algorithms.
- By the early 1970s, literally thousands of problems were stuck in this limbo.
- The theory of NP-Completeness, developed by Stephen Cook and Richard Karp, provided the tools to show that all of these problems were, actually, the same problem.
- We call these problems NP-Complete.
- What is astounding is that if you find a polynomial (optimal) algorithm for one of these problems you have found a polynomial algorithm for all of them and P = NP.
- You can then collect your Turing Award.



Stephen Cook

Richard Karp

The Class NP-Complete



- In **general**, **Turing Reductions** are **possible**.
- A problem **X** is called ***NP-Complete*** if
  a) **X belongs** to **NP and**;
  b) **all instances** of **all problems** in **NP** can be **Turing reduced** to **instances** of **X**.
- This **means** that we **can transform all instances** of **any problem** in **NP** to an instance of **X**.
- If all **problems** in **NP** can be **reduced** to **X** then **X** is at **least** as **hard** as **any problem** in **NP**. But **why**?
- **X cannot** be **easier** than any **other problem Y** in **NP since otherwise** we **could solve Y** by **Turing Reducing Y** to **X** (in **polynomial** time), **using** the '**easier**' **algorithm** for **X** and **then transforming** the **output** of **X** into the **output** of **Y** (**again**, in **polynomial** time).

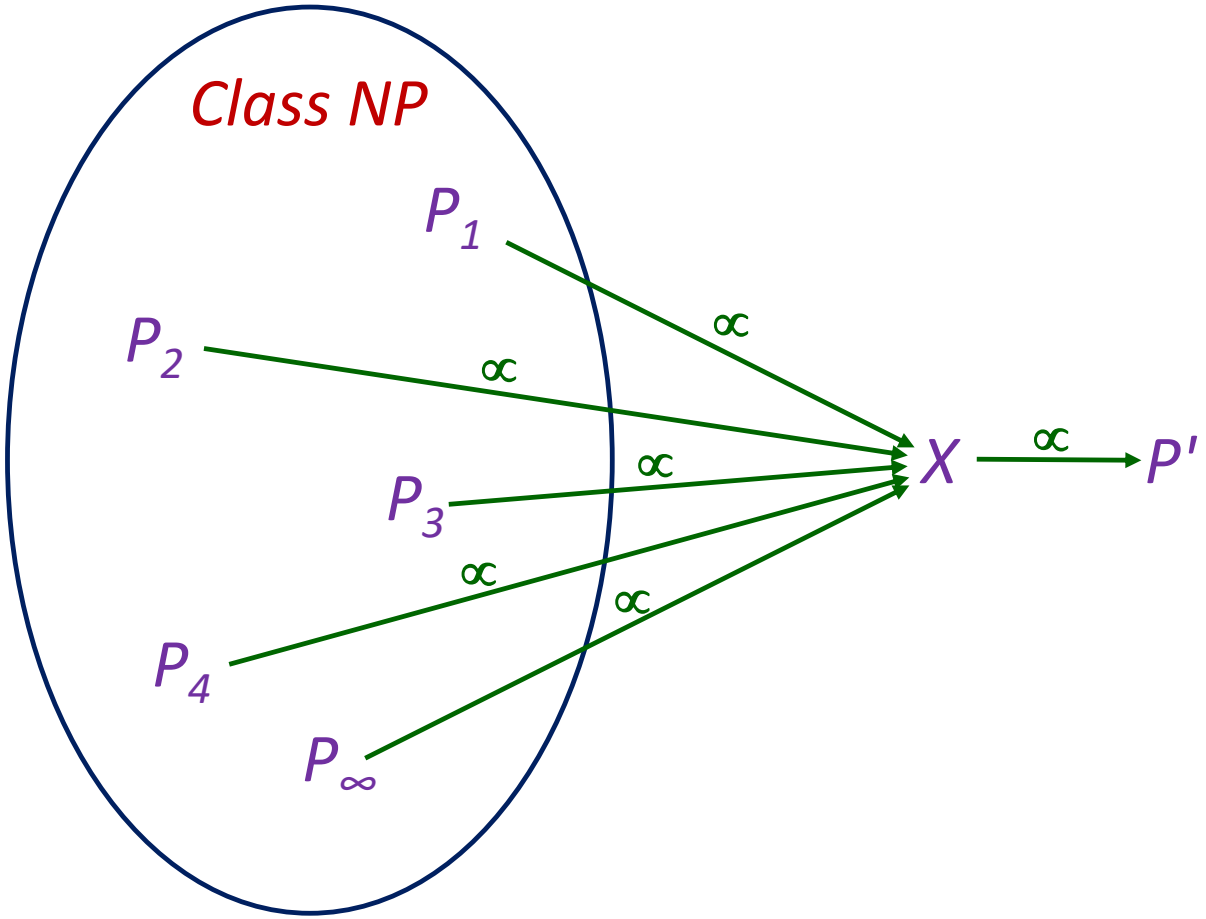
Important Lemma

Lemma 4

If P_1 and P_2 belong to NP and $P_1 \propto P_2$ then if P_1 is NP-Complete then so is P_2.

This follows from the transitivity of Turing Reduction and serves that to show that a given problem X is NP-Complete all we need to do is to reduce a known NP-Complete problem to X.

The Class NP-Complete



- From **Lemma 4** (previous **slide**), if **all problems P_i** in **NP can** be **Turing Reduced** to **X** and **X can**, **itself**, be **Turing Reduced** to a **given problem NP problem P'** then **P'** is **also NP-Complete**.
- These **results** follow from the **transitivity** of **Turing Reduction**.
- If we **have** a **problem P** and we **want** to **show** that **P** is **NP-Complete** then **we have** to:
  1) First **show** that **P is** in **NP** (very **easy**), and;
  2) then **show** that a **known NP-Complete problem X** can be **Turing Reduced** to **P**. This **part** involves **constructing** the **Turing Reduction** and is **not always** easy.
- We **shall** only **look** at some **easy examples**.
- Note that **proving** that a **problem P** is **NP-Complete** is **not** the **same** as **proving** the **P** is in **NP**.

The (Boolean) Satisfiability Problem (SAT)

Given a Boolean formula, does there exist a truth assignment to the variables to make the expression true?

Boolean variables: $x_1, \ldots, x_n$.

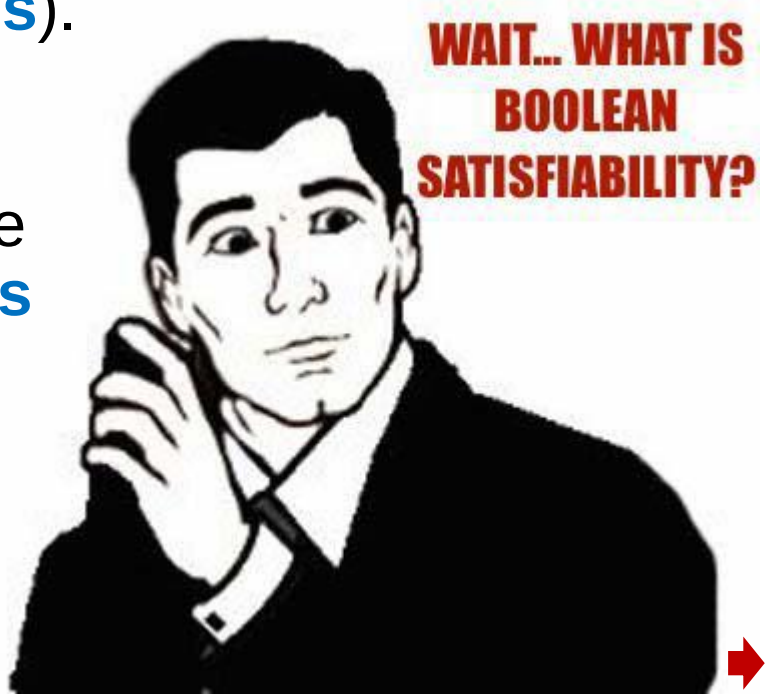
Term: x_i or its **negation** $!x_i$ (called a **literal**).

A **clause** is **simply** a **disjunction** of **terms** t_1 or t_2 or … t_j.

We are **given** a **conjunction** of **disjunctions** (the **clauses**).

Problem:

- Given a **collection** of **clauses** $C_1, \ldots, C_k$, does there **exist** a **truth assignment** that **makes** all the **clauses true**?
- $(x_1$ **or** $!x_2)$ **and** $(!x_1$ **or** $!x_3)$ **and** $(x_2$ **or** $!x_3)$



The (Boolean) Satisfiability Problem (SAT)

- Given the **Boolean variables** x_1, x_2, and x_3 and the **set** of **clauses** shown on the **right**, can we **assign True** or **False** to **each** of the **variables** such that the **all** the **clauses evaluate** to **True**?
- **Note** that this is a **conjunction** (**ANDs**) of **disjunctions** (**ORs**).
- Note that **assigning**:
  - x_1 = **1**
  - x_2 = **0**, and
  - x_3 = **1 satisfies** the **set** of **clauses**.

$(x_1 \text{ or } x_2 \text{ or } \overline{x_3})$

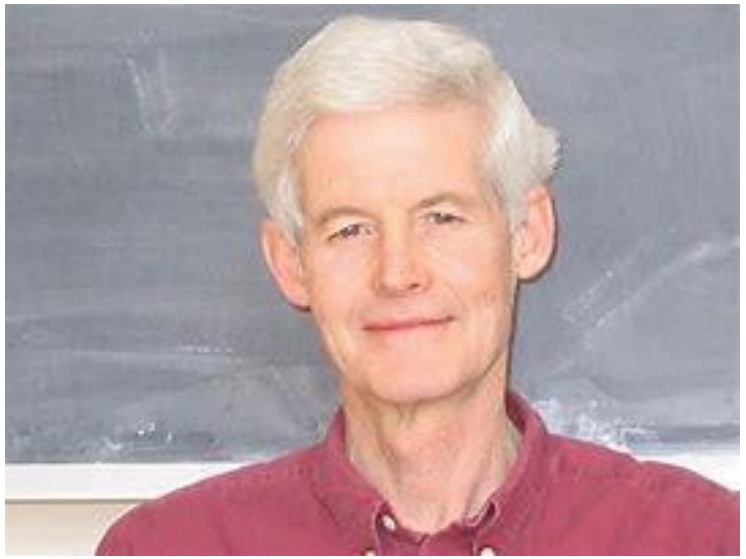
$(\overline{x_1} \text{ or } \overline{x_2} \text{ or } x_3)$

$(\overline{x_1} \text{ or } \overline{x_2} \text{ or } \overline{x_3})$

$(\overline{x_1} \text{ or } x_2 \text{ or } x_3)$

The Class NP-Complete

- In the early **1970s Stephen Cook** proved that **Satisfiability** (**SAT**) is **NP-Complete**.
- There is **Cook's proof** in the **CLRS** and also in **G&J** book.
- In essence, he **proved** that **all problems** in **NP** can be **Turing reduced** to **Satisfiability**.
- Since then **many other** problems have been **shown** to be **NP-Complete**.
- Some **reductions** are **profound** and **some** are **relatively easy**.



Stephen Cook

Cook-Levin Theorem

Theorem: **Satisfiability (SAT)** is NP-Complete

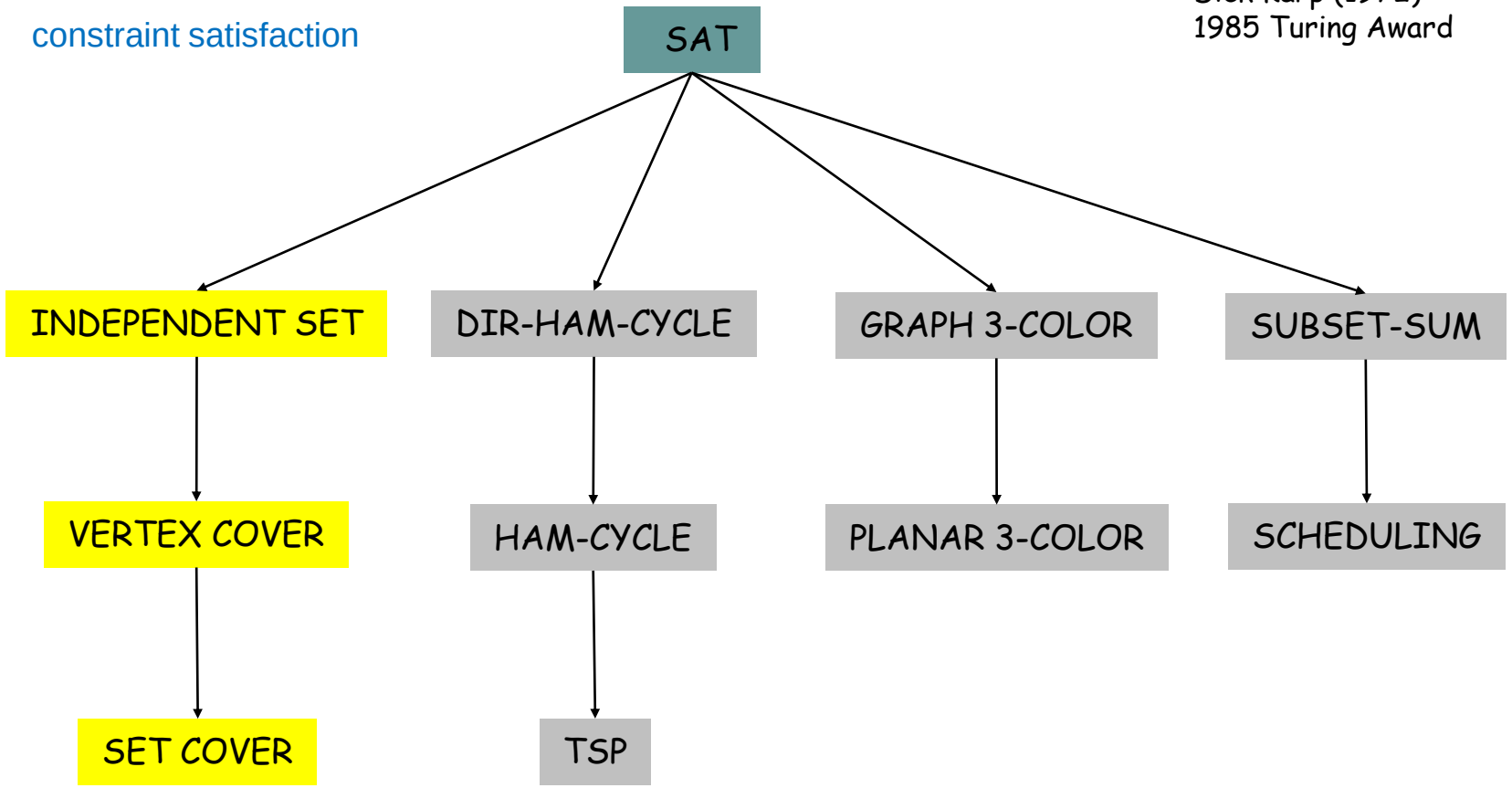
Corollary: If SAT $\in$ *P then P = NP.*

(proof is very easy)

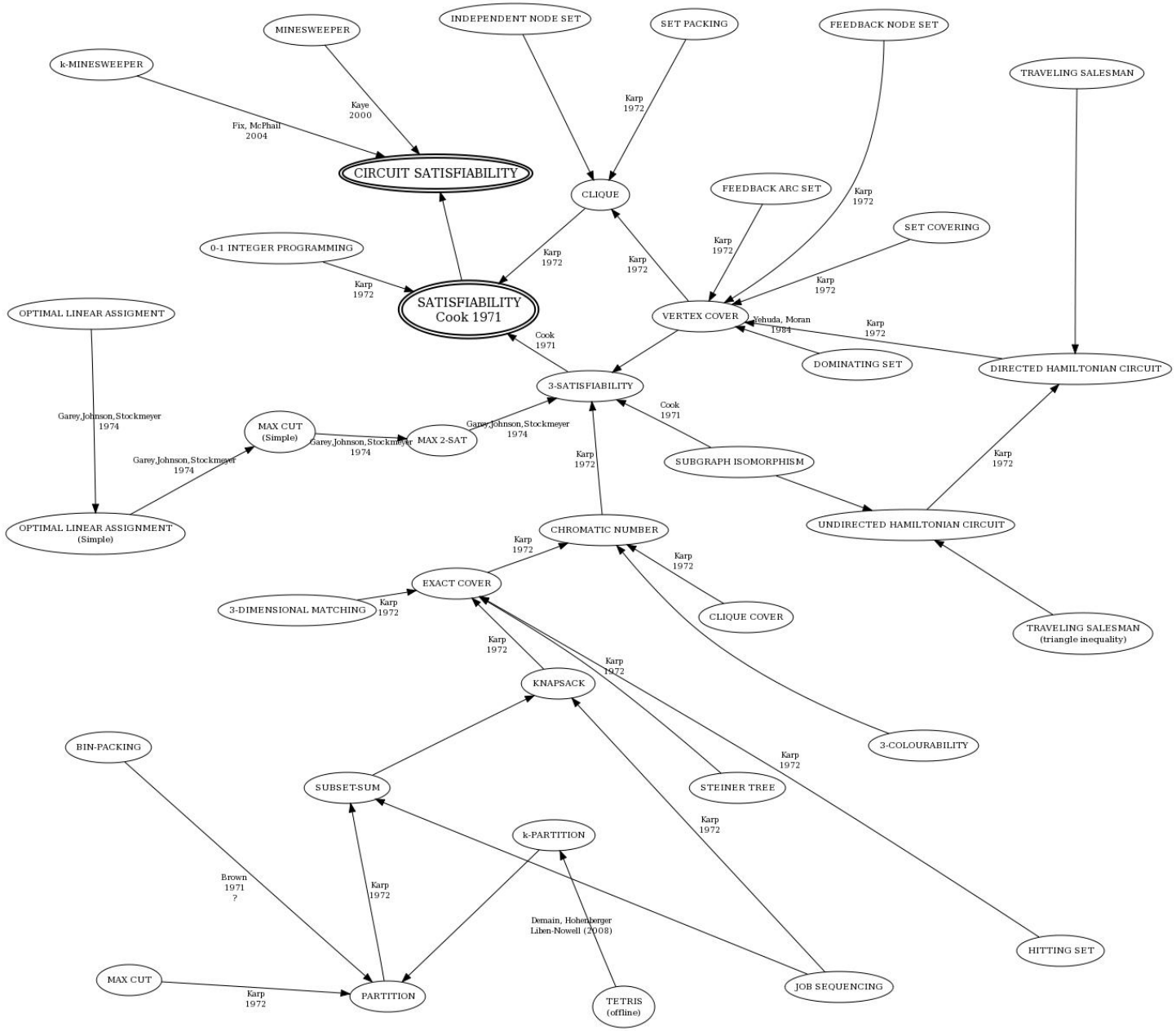
The **Cook-Levin Theorem** tells us that **SAT** is the **hardest problem** in **NP**. If **SAT** can be **solved** in **polynomial time** then so **can *ALL*** other **problems** in **NP**.

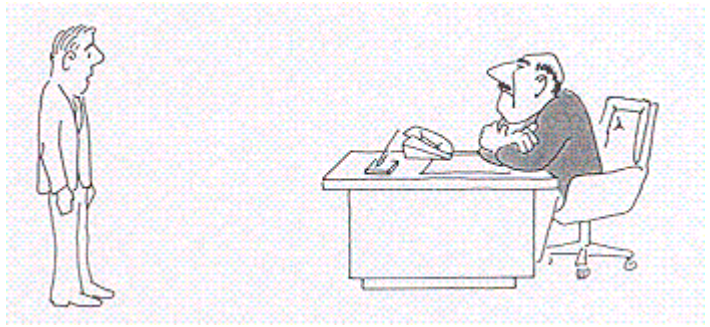
Dick Karp's Work

Dick Karp (1972)
1985 Turing Award

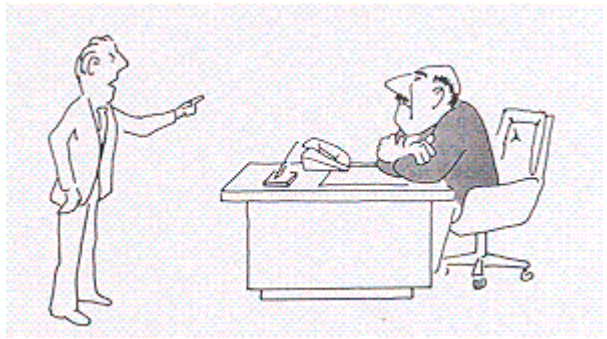


The Turing Reductions Web

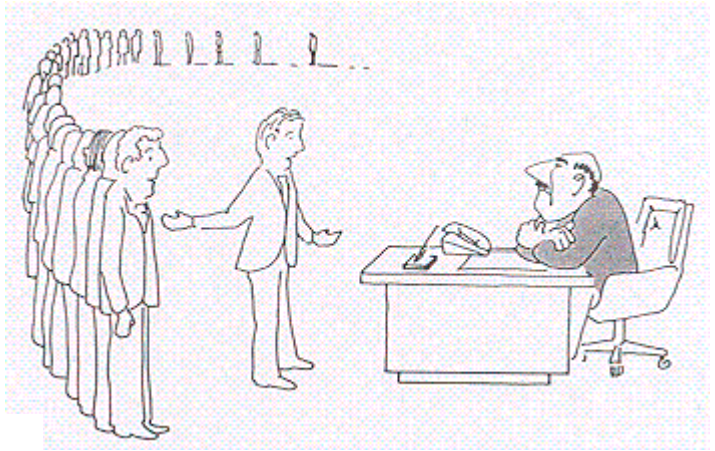




Suppose your Boss asks you to solve a difficult problem ***X*** in polynomial time.



You can tell your Boss that he is **very stupid** and that there are no polynomial solutions for ***X***.



Reduce a known NPC problem to ***X***. Tell your Boss that many clever people could not find a polynomial time solution.

Reduce Vertex Cover to Set Cover

Vertex Cover (VC): given a graph G of ***n* vertices** and **parameter *k***, is there a set of ***k* vertices** or less such that every edge has at **least one** end among selected nodes?

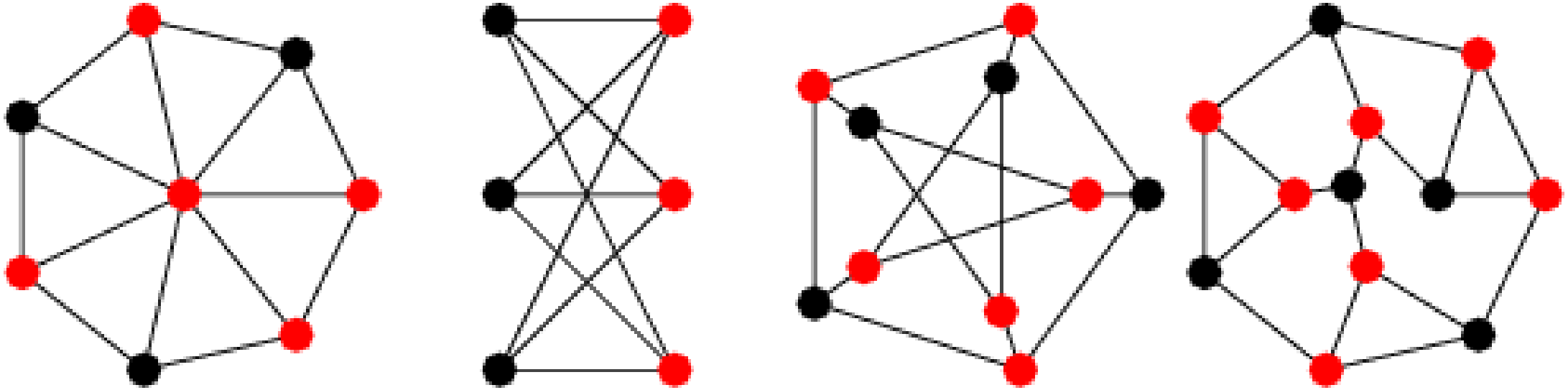
Set Cover (SC): given ***n* elements**, ***m* sets** which cover the set of elements, and **parameter *k***, is there a **family** of ***k* sets** or less that **covers** all ***n* elements**?

- We know that **Vertex Cover** is **NP-Complete**.
- We will now show the **Set Cover** is also **NP-Complete** by **reduction** from **Vertex Cover**.
- *Note that we transform VC into SC and not the other way around! This is a very common mistake by students in exams.*

Vertex Cover

Vertex Cover (VC): given a graph G of ***n* vertices** and **parameter *k***, is there a set of ***k* vertices** or less such that every edge has at **least one** end among selected nodes?

The above is a **decision problem**. In the equivalent VC optimization problem we have to find the **smallest** vertex cover.



Set Cover

Set Cover (SC): given ***n* elements**, ***m* sets** which cover the set of elements, and **parameter *k***, is there a **family** of ***k* sets** that **covers** all ***n* elements**?

Let the set of n elements be **T** = {a, b, c, d, e, f, g, h, i, j}. Note the **n** = 10

Let the subsets be:

S_1 = {a, c, e, g, i}

S_2 = {c, b, f, g}

S_3 = {c, e, f, g, j}

S_4 = {b, d, f, h, j}

S_5 = {d, g, l, j}

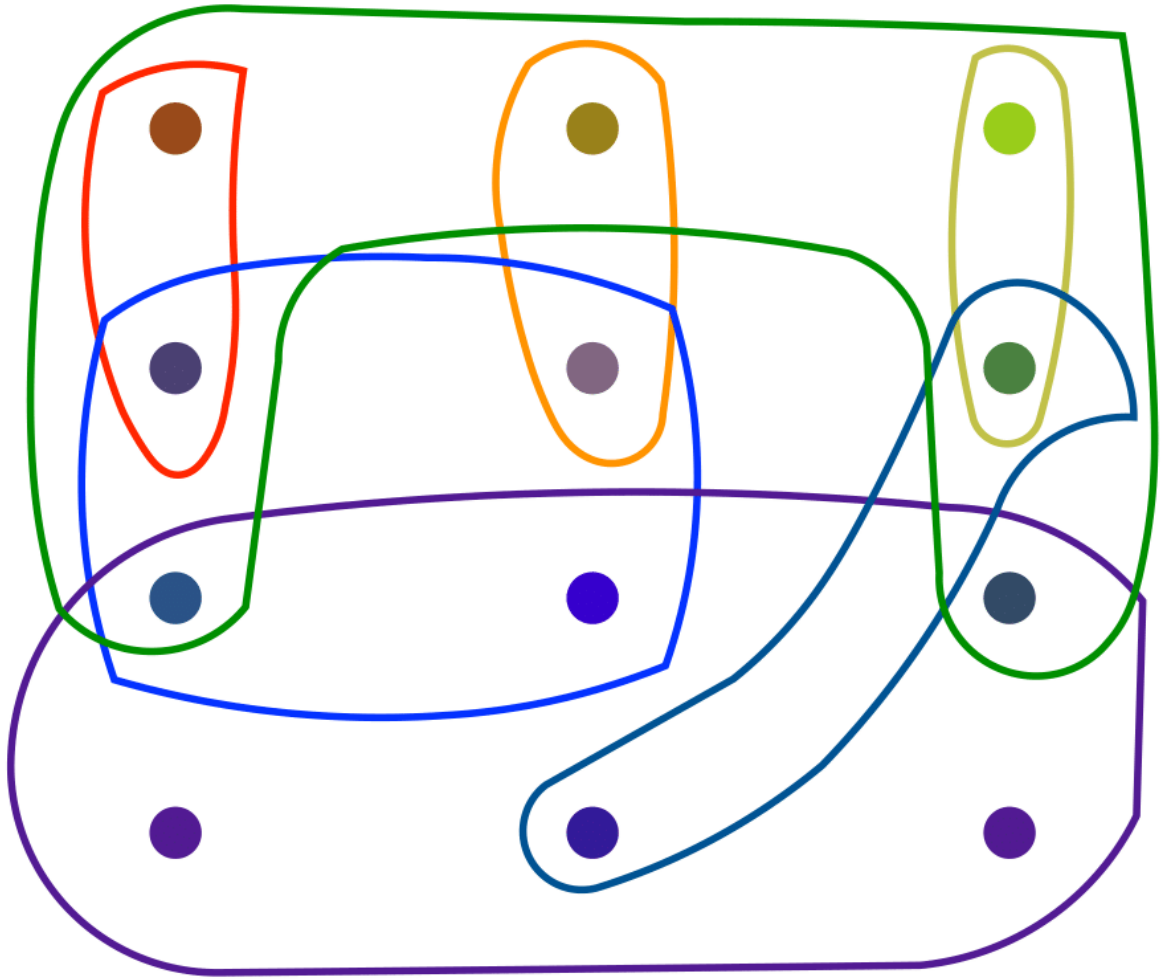
S_6 = {a, d, d, h}

We **have** to **choose** the **smallest number** of **subsets** such that their **union** is **T**.

An optimal cover is S_1 and S_4.

Visualizing Set Cover

Set Cover (SC): given ***n* elements**, ***m* sets** which cover the set of elements, and **parameter *k***, is there a **family** of ***k* sets** or less that **covers** all ***n* elements**?



Reduce Vertex Cover to Set Cover

Polynomial Reduction (VC to SC):

- Let **each edge** from the **graph** *G* **correspond** to the **elements** for the **Set Cover** C instance.
- For each **vertex** in the **graph create** a **set** of **incident edges** in the **Set Cover** instance.
- This **completes** the **Turing reduction** which **clearly** be **done** in **polynomial** time.

Proof:

Each **node-edge** is **covered** by at **least** one **set-vertex** since each **node** is **covered**.

This **covering** is **therefore** minimal.

NP-Hard Problems

Remember - A **problem X** is called **NP-Complete** if:

- **X** is in **NP**, and
- all **instances** of **all problems** in **NP** can be **Turing reduced** to **instances** of **X**.

A **problem X** is called **NP-Hard** if:

- all **instances** of **all problems** in **NP** can be **Turing reduced** to **instances** of **X**.

- Note that **every NP-Complete problem** is also **NP-Hard** and that **NP-Complete** is a **subset** of **NP-Hard**.
- An **NP-Hard problem**, **therefore**, does **not necessarily** have to be **also** in **NP**.

NP-Hard Problems

- An **NP-Hard** problem is '**hard**' because it **cannot** have a **polynomial** solution **unless**, of course, **P = NP**.
- Of **course**, if a **problem X** in **NP** and it is **NP-Hard** then it is also **NP-Complete**.
- Many of the **non-decision** problem **versions** of **problems** in **NP** are **NP-Hard**. An **example** is **TSP**.
- ***The only difference between an NP-Complete problem and an NP-Hard problem is that an NP-Complete must be a decision problem and be in NP. An NP-Hard problem does not necessarily have to be a decision problem and does not necessarily have to be in NP.***
- Every **NP-Complete** Problem is also **NP-Hard**. **NP-Complete** is a **subset** of **NP-Hard**.
- You **must know** the **definitions** of **both classes**!

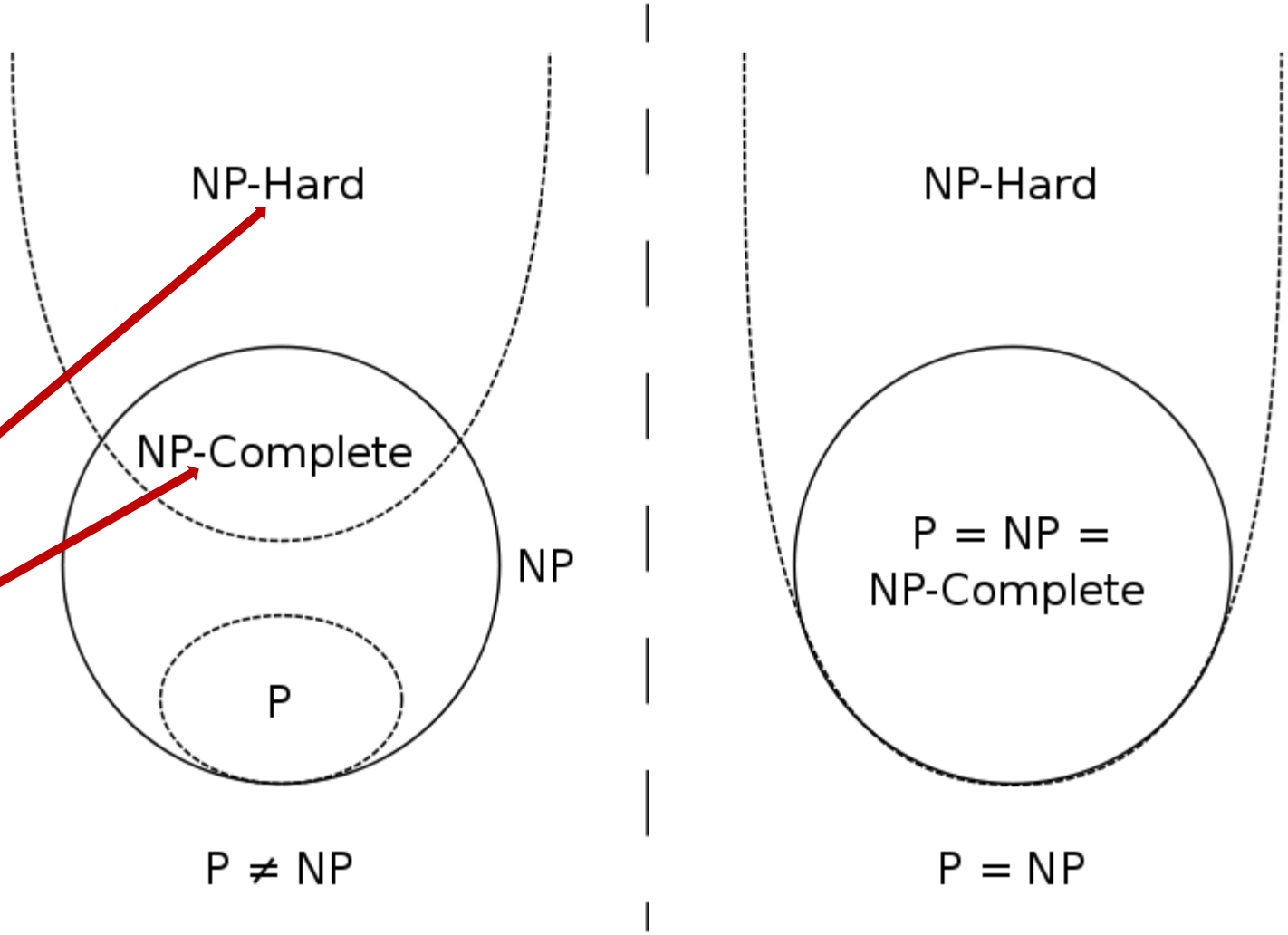
The Current Situation

Remember that **NP-Complete** is a **subset** of **NP-Hard**.

N**P-Hard** problems **may** or **may not** be **decision problems**.

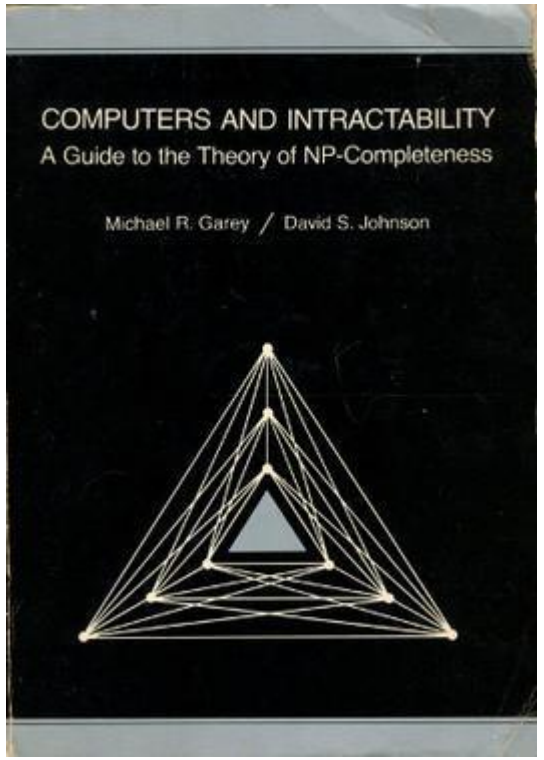
Optimization problems

Decision problems



Examples of NP-Hard and NP-Complete Problems

- **Time-Tabling**
- **Production** Scheduling
- **Airline Crew** Scheduling
- Multiple **Bio-Sequence** Alignment
- Many **Search Problems**
- Many **Optimization Problems**
- Problems in **Astrophysics**
- **Graph Theoretic** Problems
- **Design** Problems
- **Game Theory**



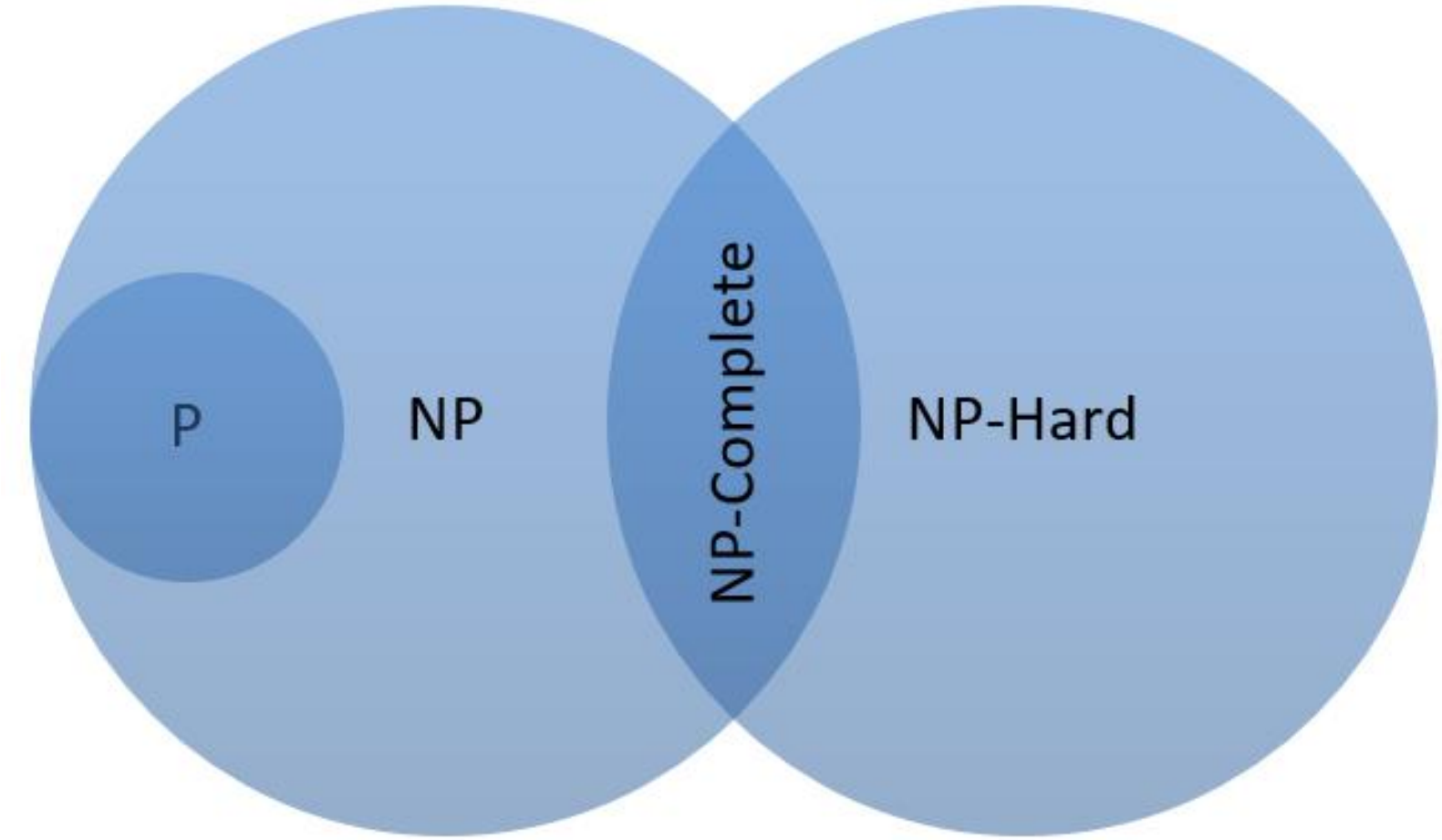
The Garey & Johnson book contains 100's of NP-Complete and NP-Hard problems.

NP-Hard and NP-Complete Problems

Do **not forget** that:

- **NP-Complete** problems are **always** **decision problems**.
- **NP-Hard** problems are **usually optimization** problems were we **want** to **maximize** or **minimize** a **function**. **NP-Hard** problems can **also be decision** problems. Every **NP-Complete** problem is also **NP-Hard**!.
- Most **search problems** can be **posed** as **optimization** problems.

NP-Hard and NP-Complete Problems



Importance of NP-completeness
Importance of "Is P=NP" Question

- Theoretician's view
  - NP is exactly the set of problems that can be "verified" in polynomial time
  - Thus "Is P=NP?" can be rephrased as follows:
    - Is it true that any problem that can be "verified" in polynomial time can also be "solved" in polynomial time?
- Hardness Implications
  - It seems unlikely that all problems that can be verified in polynomial time also can be solved in polynomial time
  - If so, then P≠NP
  - Thus, proving a problem to be NP-complete is a hardness result as such a problem will not be in P if P≠NP.

Some Notes.

- The **Gary** and **Johnson** book is the **most cited book** in **all** of **computer science**.
- After **many decades**, and in **spite** of a **million dollar prize** offer and **intensive study** by **many** of the **best minds** in **computer science**, **no one managed** to **either**:
  - **Prove** that there **are problems** in **NP** that **cannot** be **solved** in **polynomial time** (which **would mean P ≠ NP**), or
  - Find a **polynomial time solution** for a **single NP Complete Problem** (which would **mean P=NP**).
- In **1979 Garey** & **Johnson** wrote, **"The question of whether or not the NP-complete problems are intractable is now considered to be one of the foremost open questions of contemporary mathematics and computer science."**
- **Problems** that look **very similar** to **each other** may **belong** to **different complexity classes** (if P ≠ NP), for **example**:
  - The **shortest path/s** in a **graph** can be **found** in **O(*VE*)** or **better**, but the **problem** of **finding** the **longest simple paths** is **not known** to be **polynomial-time**.

IS P = NP?

- The **current conventional wisdom** is that **P ≠ NP**.
- Many **people**, often **enthusiasts**, have **published claims**, or '**proofs**' that **P = NP** or **P ≠ NP**.
- All **these claims** were **rejected** after **academic scrutiny**.
- A **proof** that **P = NP** could have **far-reaching practical consequences** if the **proof leads** to **efficient methods** for **solving** some of the **important problems** in **NP**.
- The **potential consequences**, both **positive** and **negative**, arise since **various NP-complete** problems are **fundamental** in **many fields**.
- The **link below provides *116*** '**proofs**' or **claims** about the **problem**. Only **one** was **published** in an **academic journal** and it **simply showed** that a **certain approach** would **not work**.
- https://www.win.tue.nl/~gwoegi/P-versus-NP.htm

More Notes and Questions

What are some examples of NP-Complete problems whose real-life instances are not particularly hard to solve?

- It is common for NP-hard problems to have ***some*** real-life instances that are completely feasible, and others that are not. The former may be feasible simply because the parameters are small enough, or because we have efficient algorithms that solve the problem approximately.
- For example, one of the grand-daddies of all NP-complete problems is Boolean satisfiability, or SAT. Many problems can be efficiently and naturally reduced into SAT, and the fact that it is hard does not cause us to throw our hands in the air and declare any SAT instance as hopeless.

Quora

More Notes and Questions

Why is it that all NP-Complete problems don't have a unifying approximation algorithm?

- One intuitive reason is that approximations do not necessarily survive reductions.
- Take two **NP-Complete** problems: P_1 and P_2. The fact that they are **NP-Complete** implies that we can transform instances of P_1 into instances of **P2**, in polynomial time, in such a way that an exact solution to P_2 maps to a corresponding exact solution to P_1. (We say that we have reduced P_1 to P_2.)
- But let's say that S_2 is an approximate solution to P_2. If we do the same transformation of P_1 to an instance of P_2, we have no guarantee that **S2**, when transformed back into the problem space of P_1, will be a good approximate solution to P_1.

Quora

More Notes and Questions

What are some of the hardest NP-complete problems?

They are all equally "hard", in the most fundamental sense.

If you could solve ***any*** of them efficiently, you could solve ***all*** of them efficiently. That is the astonishing theorem of Cook and Levin that creates the concepts of NP-Hardness and NP-Completeness in the first place.

Are all NP problems decidable?

Yes. An NP-problem is one solvable by a nondeterministic Turing machine in polynomial time. But you can simulate a nondeterministic Turing machine on a deterministic Turing machine with, at worst, exponential slowdown just by walking through the different possible choices of transition, so NP problems are not only decidable, they are decidable in, at most, exponential time.

Quora

More Notes and Questions

What are the current approaches for approximating NP complete problems?

1) Narrowing the problem space

For instance, if we cannot solve TSP on general graphs, let us try to just solve it for graphs obeying a Euclidean distance metric.

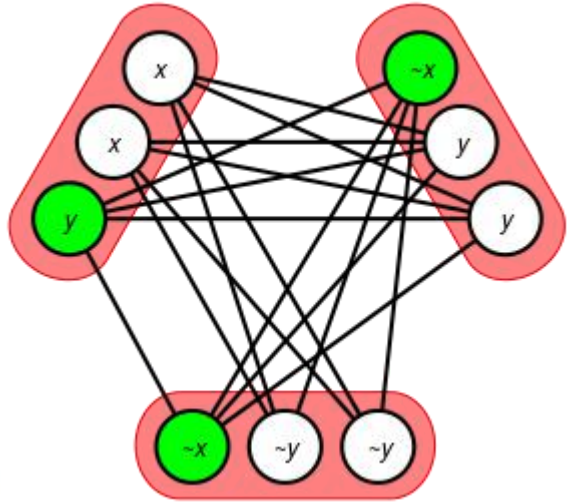
2) Approximation Algorithms

Some NP-Hard problems admit polynomial time approximation algorithms. Sometimes this give only a constant factor approximation at best, such as MAX-CUT or Metric TSP, and sometimes it yields a Polynomial Time Approximation Scheme, whereby one can trade additional running time for a better approximation. (It's polynomial time in n with 1/epsilon in the exponent.) In the case of Euclidean TSP, we have had such an algorithm since 2010.

Quora

Using SAT Solvers.

- Since all NP-Complete problems can be Turing reduced to each other, we then can use an algorithm for one NP-Complete problem to solve all other NP-Complete problems.
- This is a very powerful idea and has many practical applications.
- If is common in academia, and even industry, to Turing reduce hard problems to SAT and then to use a SAT Solver.
- SAT Solvers have been around for many years and use very sophisticated techniques.
- https://en.wikipedia.org/wiki/SAT_solver
- DFA Learning is one example.
- One of the best SAT Solvers is MiniSAT.



The MiniSat Page

by Niklas Eén, Niklas Sörensson

Main
MiniSat
MiniSat+
SatELite
Papers
Authors
Links

Introduction

MiniSat is a minimalistic, open-source SAT solver, developed to help researchers and developers alike to get started on SAT. It is released under the MIT licence, and is currently used in a number of projects (see "Links"). On this page you will find binaries, sources, documentation and projects related to MiniSat, including the Pseudo-boolean solver MiniSat+ and the CNF minimizer/preprocessor SatELite. Together with SatELite, MiniSat was recently awarded in the three industrial categories and one of the "crafted" categories of the SAT 2005 competition (see picture).

Some key features of MiniSat:

- **Easy to modify.** MiniSat is small and well-documented, and possibly also well-designed, making it an ideal starting point for adapting SAT based techniques to domain specific problems.
- **Highly efficient.** Winning all the industrial categories of the SAT 2005 competition, MiniSat is a good starting point both for future research in SAT, and for applications using SAT.
- **Designed for integration.** MiniSat supports incremental SAT and has mechanisms for adding non-clausal constraints. By virtue of being easy to modify, it is a good choice for integrating as a backend to another tool, such as a model checker or a more generic constraint solver.

We would like to start a community to help distribute knowledge about how to use and modify MiniSat. Any questions, comments, bugreports, bugfixes etc. should be directed to minisat@googlegroups.com. The archives can be browsed here. The source code repository for MiniSat 2.2 can be found at github.

the polytechnic

Search... Subscribe

News Features **Opinion** Sports | Staff Contact About Archives Submit

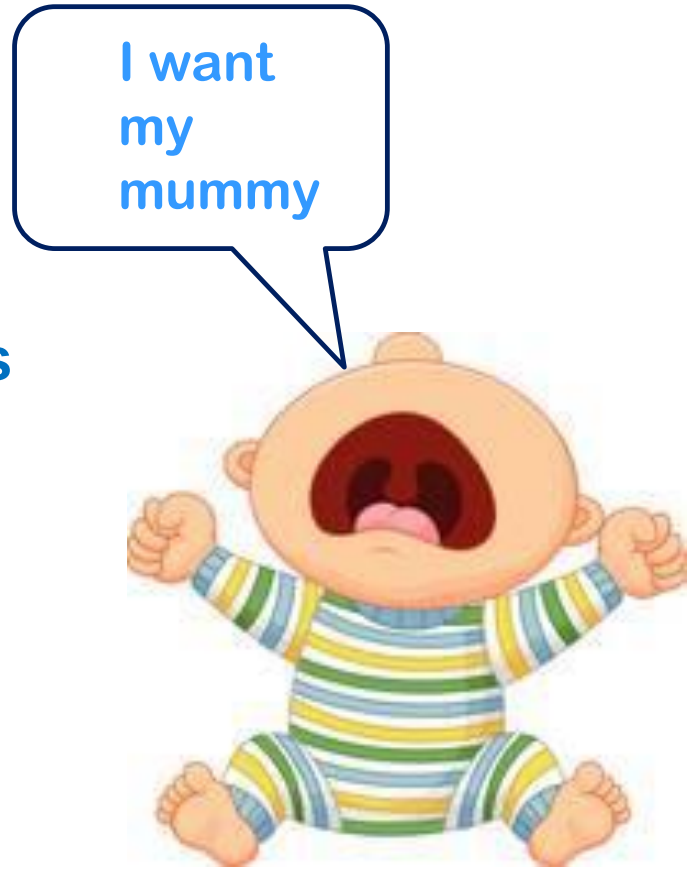
EDITORIAL NOTEBOOK

The wonderful world of SAT solvers

				1		2	6	5
					6		4	3
			2			8		
	8		5	6		9		4
	4						5	
5		9		4	8		2	
		1			3			
8	7		1					
4	9	6		5				

What Do We Do Next?

- Are there **NP-Complete** problems that **provably require super-polynomial time**?
- The **results** from **NP-Completeness** Theory **are** a little **depressing**.
- We can **sit** in a **corner** and ***cry*** **or** we **can do something** about it.
- What we **must *NEVER*** do for **large problem instances** is to write **brute-force** algorithms. This is **extremely stupid** (and **unproductive**).
- The **answer** is to use **polynomial time Approximation Algorithms**.
- Some **NP-Complete** problems are **hard** to **approximate**. These **are APX-Hard**.



NP-Completeness

Slides will be Uploaded to the VLE

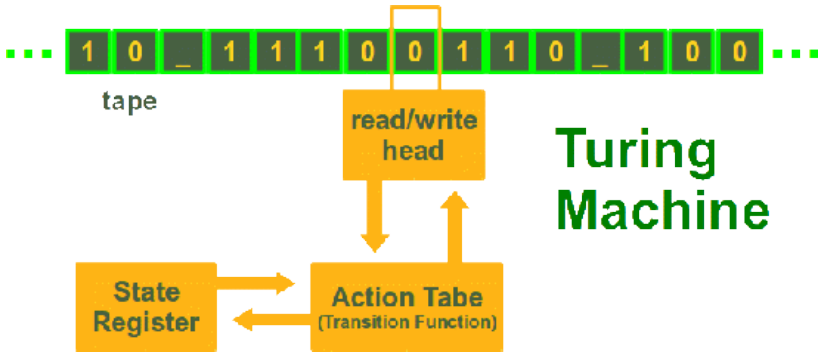
L-Università ta' Malta
Faculty of Information & Communication Technology

Data Structures And Algorithms 2

John Abela
Department of CIS
Faculty of ICT
University of Malta

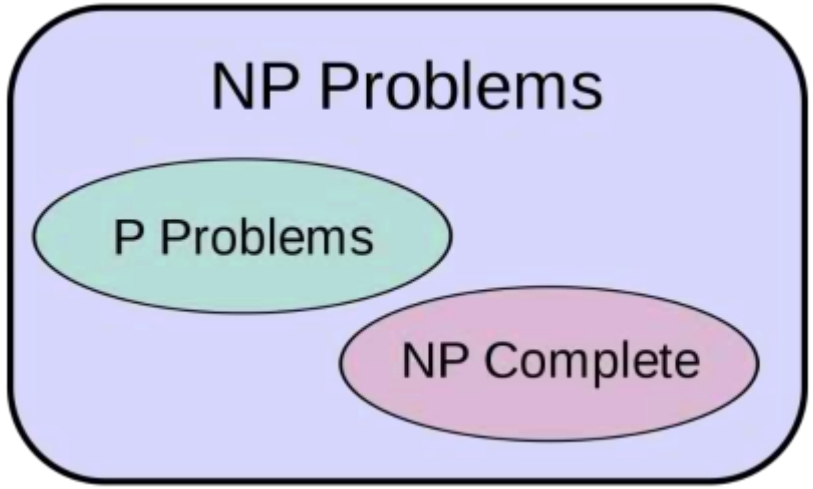
john.abela@um.edu.mt

+ 365 79367936
Room 1A/27
FICT Building



Module 2

Turing Reductions & Approximation



Faculty of ICT

Department of Computer Information Systems

Last Updated 14-Feb-2022

Module Overview

- **Present Books used for Material**
- **Re-visit VC-IS-Clique Problem**

  Present general strategies for Turing Reduction
- **Approximation Algorithms**
- **Genetic Algorithms**

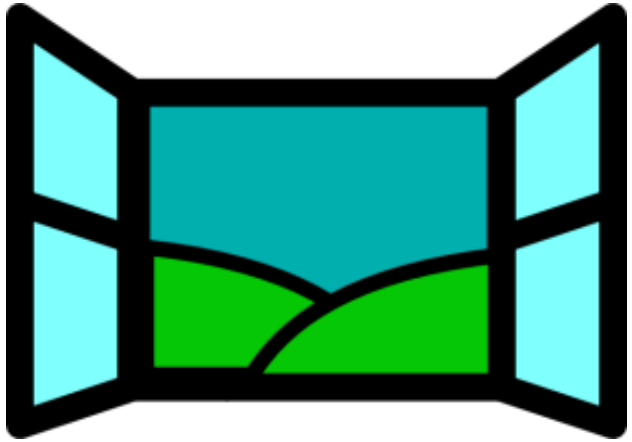
  with applications in various domains

https://en.wikipedia.org/wiki/List_of_NP-complete_problems

We Cannot Throw NP-Complete and NP-Hard Problems out of the Window

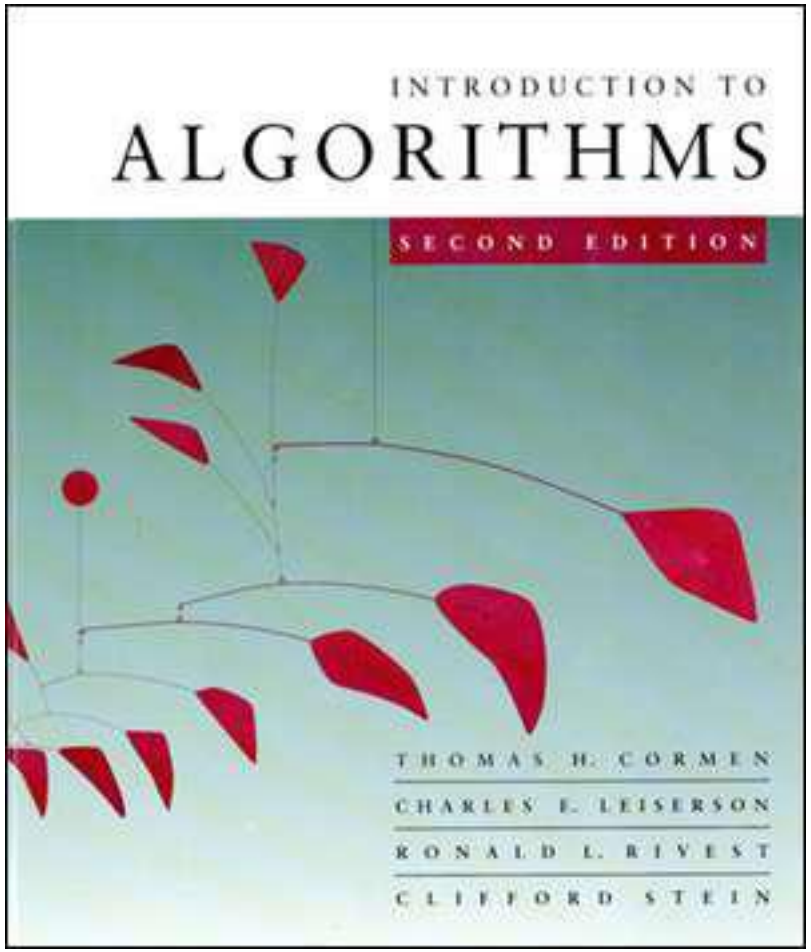
Exponential Problems?

Non-Computable Problems?

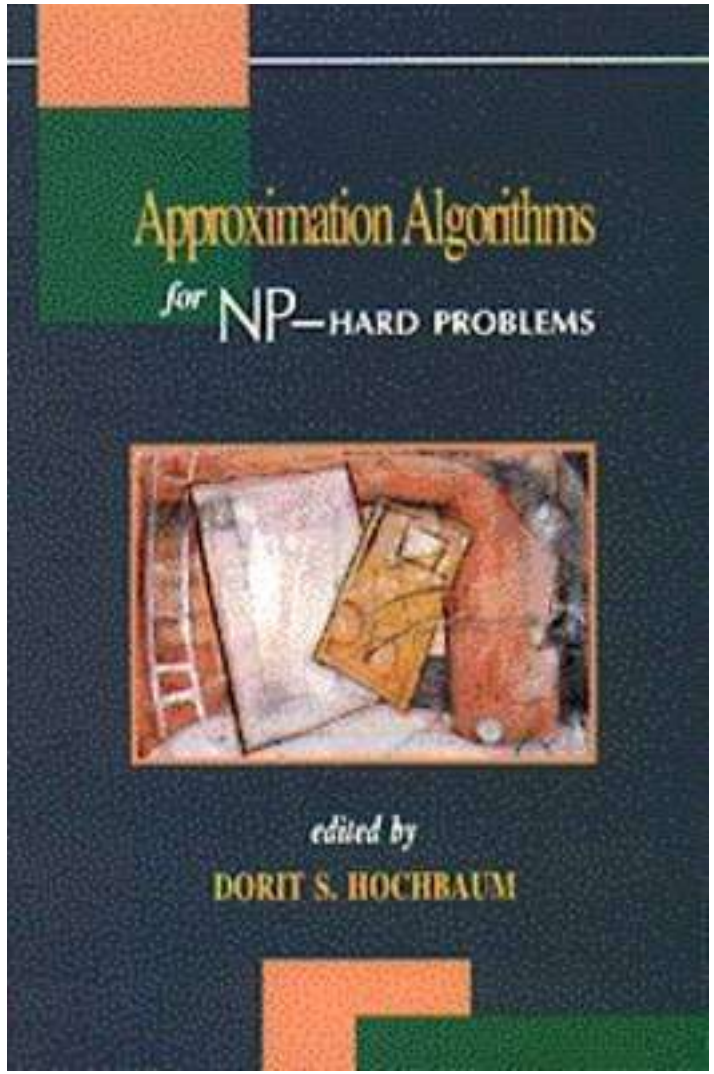


- There are, **literally**, **thousands** of **important search**, **optimization**, and **decision** problems that **have been shown** to be **NP-Hard** or **NP-Complete**.
- We **cannot throw** these **problems out** of the **window**. **Many** are **everyday problems** like **scheduling**, **routing**, etc.
- We **cannot** use **brute force**!
- The **only available** option is to **find polynomial-time approximations**.
- Some **NP-Complete** problems **cannot** be **approximated** in **polynomial time**.

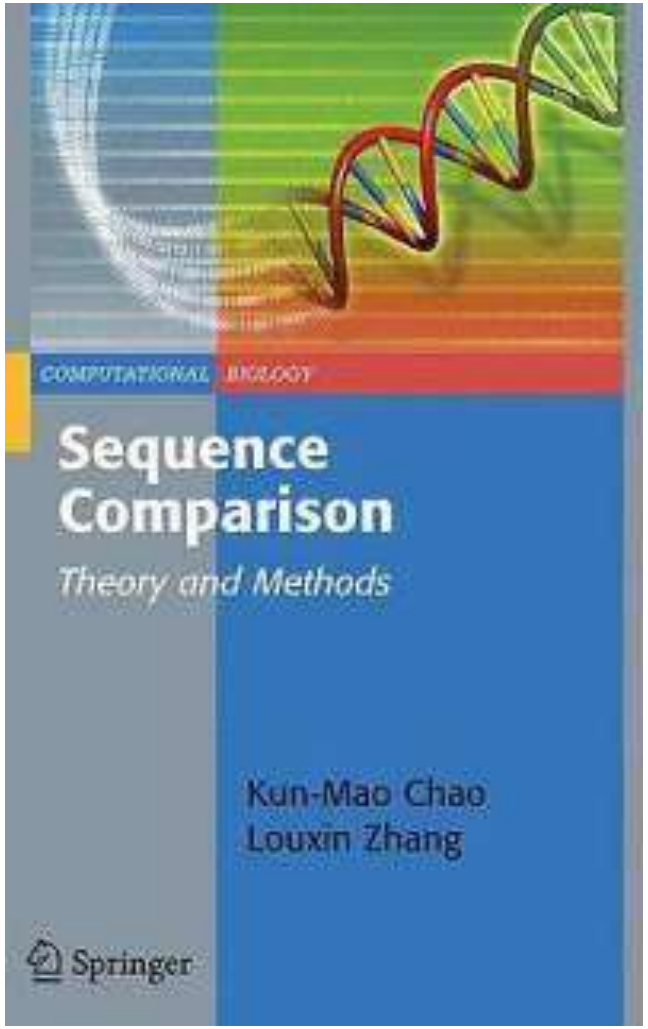
Cormen, Leiserson, Ronald, Stein



Book by Dorit S.Hochbaum

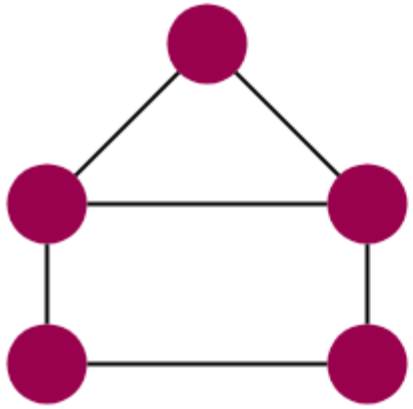


Sequence Comparison - Chao, Zhang

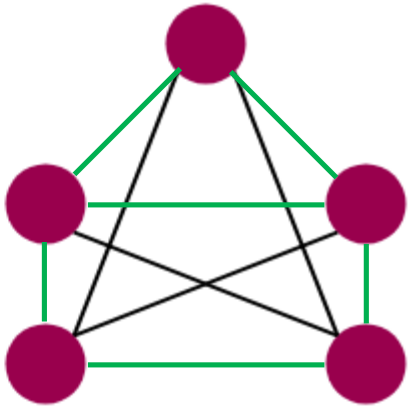


Complement of a Graph

- The **complement** of a **G**=(**V**,**E**) **denoted** by **G**^{c}=(**V**,**E**^{c}) is the **graph** with **exactly** the **same vertices** of **G** but **such that** any **two distinct vertices** in **G**^{c} are **adjacent** if and **only** if they are **not adjacent** in **G**.
- Sometimes **called** the '**inverse**' of **G**. Sometimes **also denoted** by $\overline{G}$
- To **obtain G**^{c} from **G simply delete** all **existing edges** from **G** and **add edges** between **non-adjacent** vertices.
- Remember that a **graph** is **complete** if **every pair** of **vertices** are **adjacent** – i.e. **there** is an **edge joining** them.

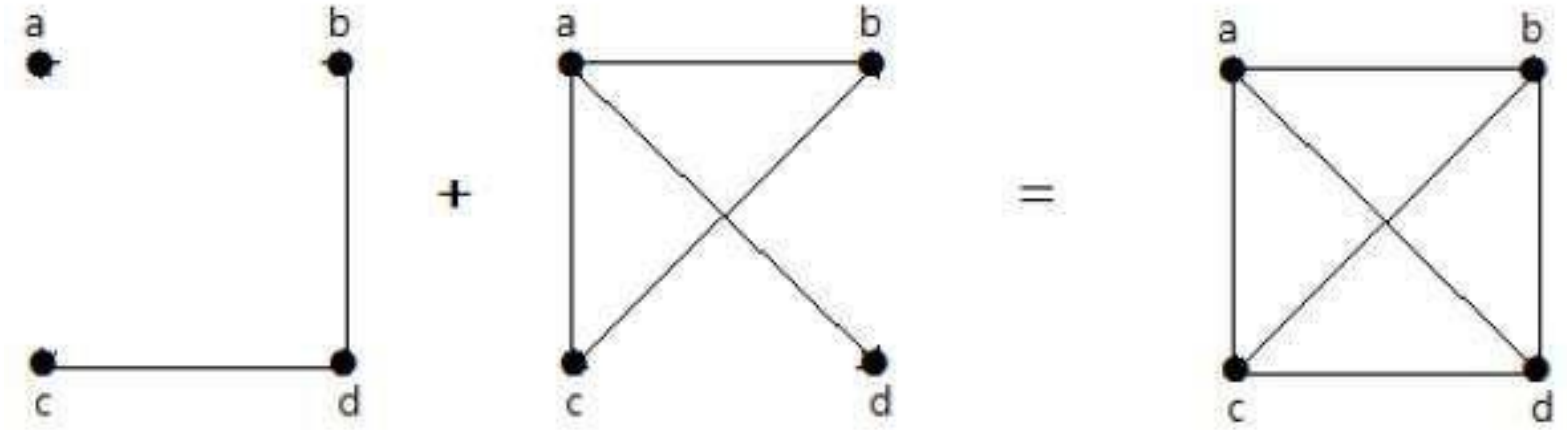


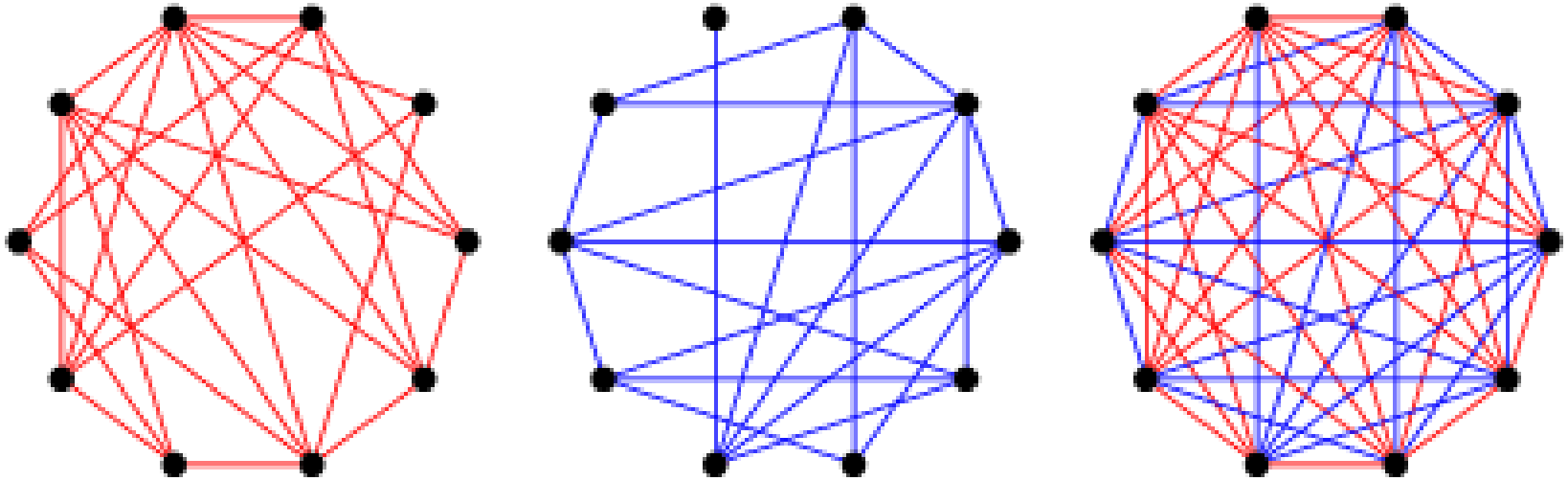
Graph G



Complement Graph $\overline{G}$

Examples of Graph Complement

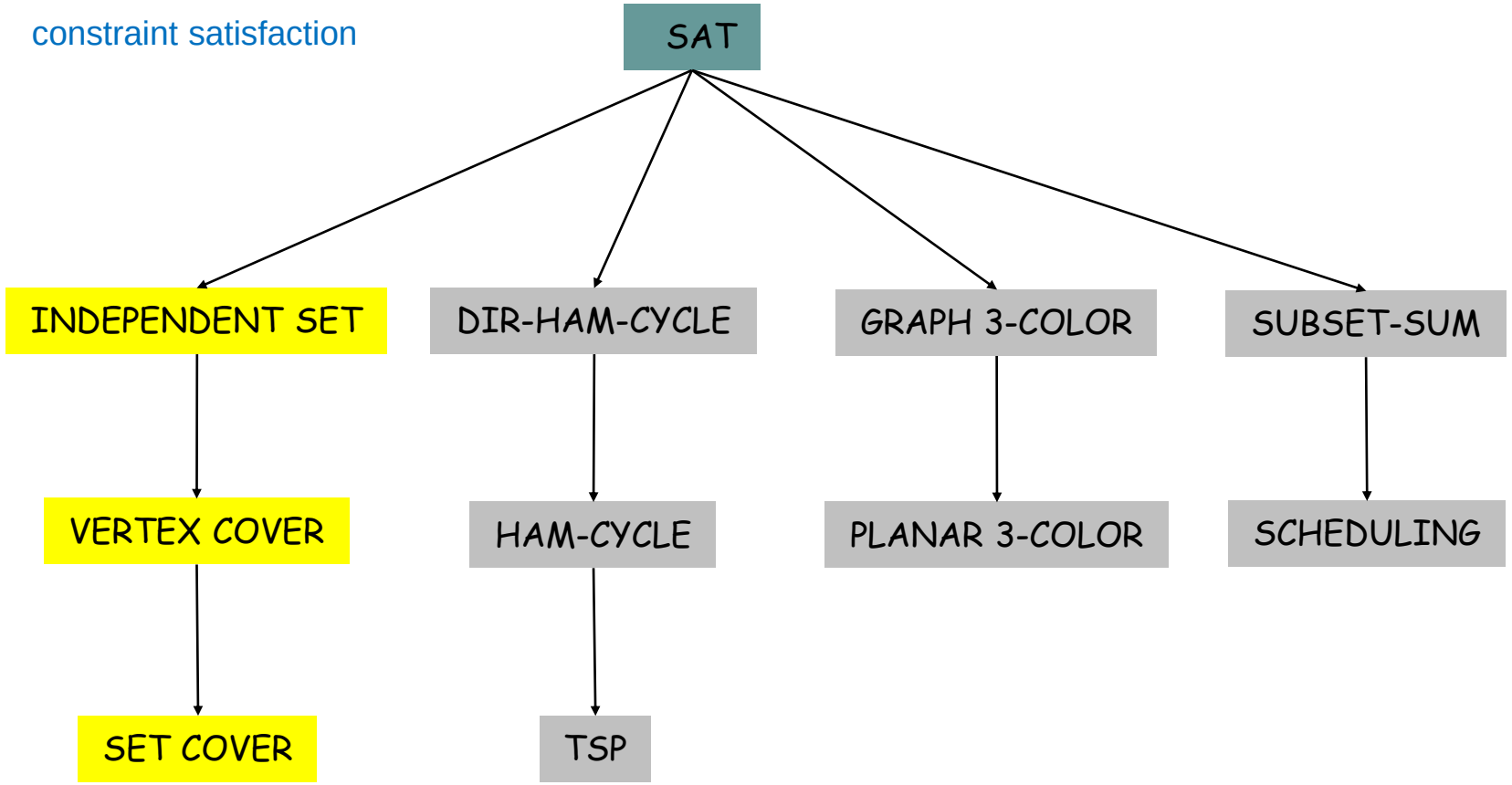




Dick Karp's Work

Remember this?

Dick Karp (1972)
1985 Turing Award



Dick Karp's Work

- After **Stephen Cook** had **proved** that **SAT** is **NP-Complete**, **Dick Karp** (who **would later** on **win** the **Turing Award**) **reduced SAT** to many other **problems** and **thus proved** (by the **transitivity** of **Turing Reduction**) that these **problems** were **also NP-Complete**.

- **Question 1**
  Is it **true** that for **any two NP-Complete problems A** and **B** then **A** can be **Turing Reduced** to **B** and **vice versa**?
- **Question 2**
  What **about** if **A** is **NP-Complete** and **B** is **simply** in **NP** but **not known** to **be** NP-Complete?
- **Question 3**
  What if I **can Turing Reduce** a **known NP-Complete problem A** into a **problem B** which is **known** to **be** in **P**?

Question 1 Answer

- Question 1
  Is it **true** that for **any two NP-Complete problems A** and **B** then **A** can be **Turing Reduced** to **B** and **vice versa**?

- If **A** and **B** are **NP-Complete then**, by **definition**, they are **also** both in **NP**.
- If **A** is **NP-Complete then** all **problems** in **NP** (**including B**) **can** be **Turing Reduced** to **A**.
- If **B** is **NP-Complete then** all **problems** in **NP** (**including A**) **can** be **Turing Reduced** to **B**.
- **Therefore**, **A** $\propto$ **B** and **also B** $\propto$ **A**. **QED**

- '**QED**' is ***quod erat demonstrandum***, Latin for "**I proved what I was trying to prove**". This is **used** to **denote** the **end** of the **proof**.

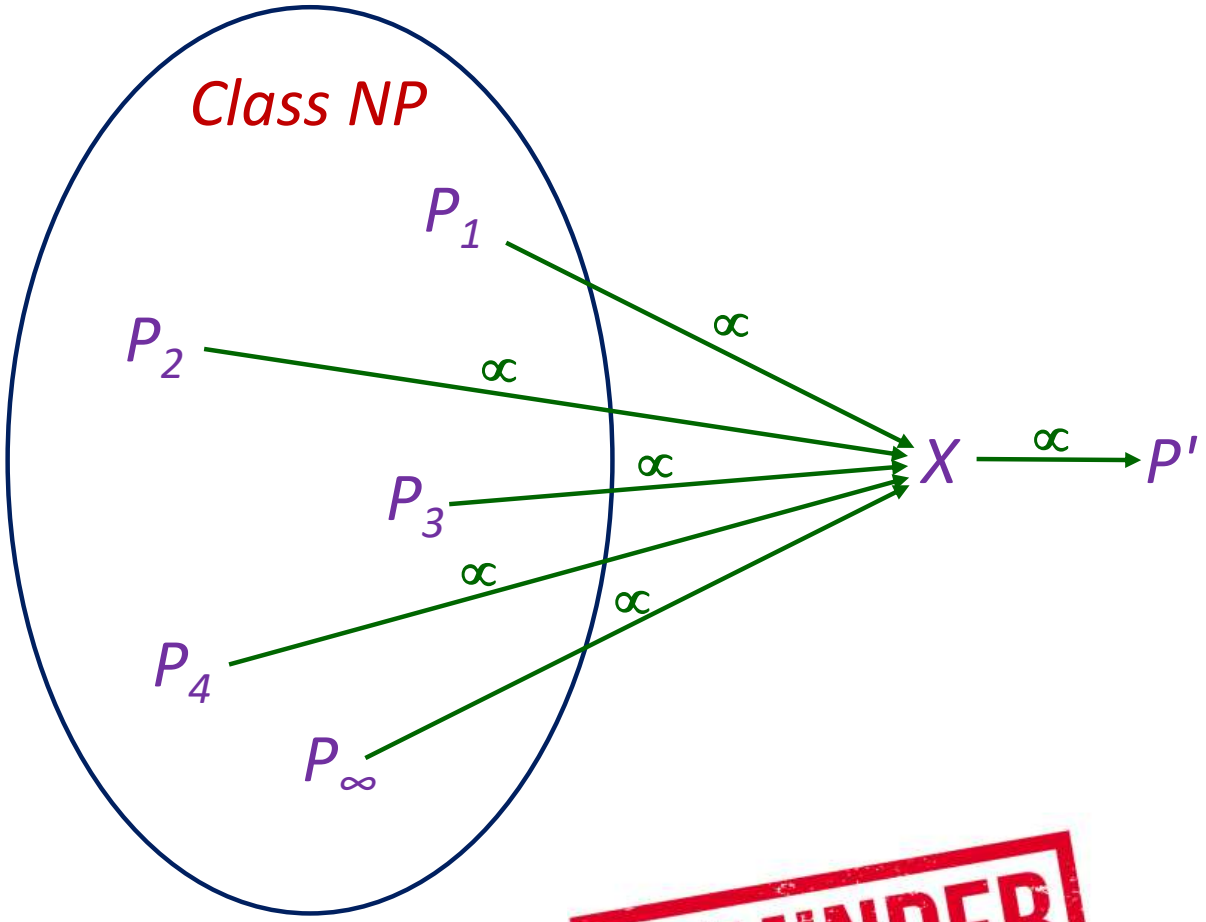
The Transitivity Argument

Lemma 4

If P_1 and P_2 belong to NP and $P_1 \propto P_2$ then if P_1 is NP-Complete then so is P_2.

This follows from the transitivity of Turing Reduction and serves that to show that a given problem *X* is NP-Complete all we need to do is to reduce a known NP-Complete problem to *X*.

The Transitivity Argument



- From **Lemma 4** (previous **slide**), if **all problems P_i** in **NP can** be **Turing Reduced** to **X** and **X can**, **itself**, be **Turing Reduced** to a **given problem NP problem P'** then **P'** is **also NP-Complete**.
- These **results** follow from the **transitivity** of **Turing Reduction**.
- If we **have** a **problem P** and we **want** to **show** that **P** is **NP-Complete** then **we have** to:
  1) First **show** that **P is** in **NP** (very **easy**), and;
  2) then **show** that a **known NP-Complete problem X** can be **Turing Reduced** to **P**. This **part** involves **constructing** the **Turing Reduction** and is **not always** easy.
- We **shall** only **look** at some **easy examples**.
- Note that **proving** that a **problem P** is **NP-Complete** is **not** the **same** as **proving** the **P** is in **NP**.

Question 2 Answer

- **Question 2**
  What **about** if **A** is **NP-Complete** and **B** is **simply** in **NP** but **not known** to **be** NP-Complete?

- If A can be Turing Reduced to B then B is also NP-Complete by the transitivity of Turing Reduction.
- If B can be Turing Reduced to A than this tells us that A is at least as hard as B. This does not tell us much about B. In fact, B could even be in P.
- If A can be Turing Reduced to B and vice versa (both of the above) then this means that B is also NP-Complete using the transitivity argument from Question 1.

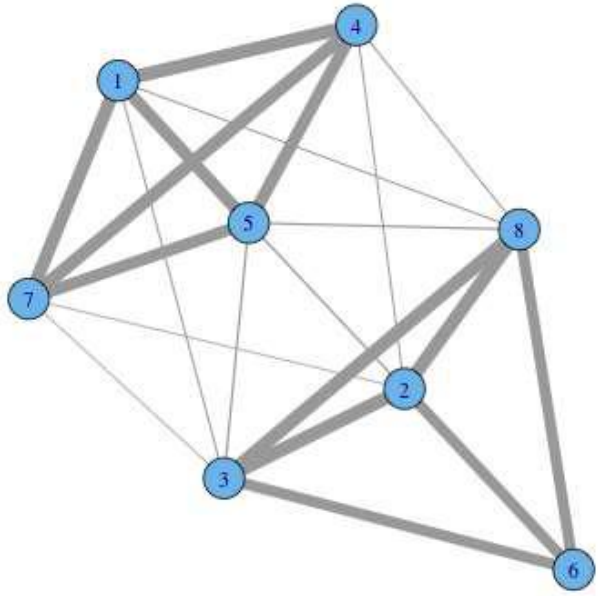
Question 3

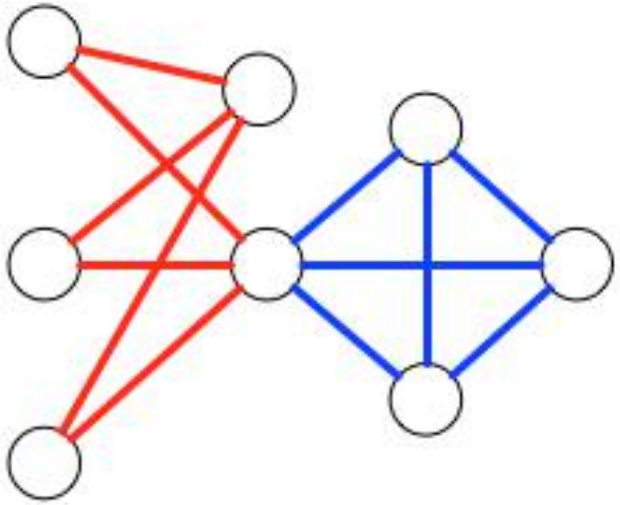
- **Question 3**
  What if I **can Turing Reduce** a **known NP-Complete problem A** into a **problem B** which is **known** to **be** in **P**?

- This is **left** as **an exercise**.
  Hint. You will be **1 million dollars** richer.

What is a Clique?

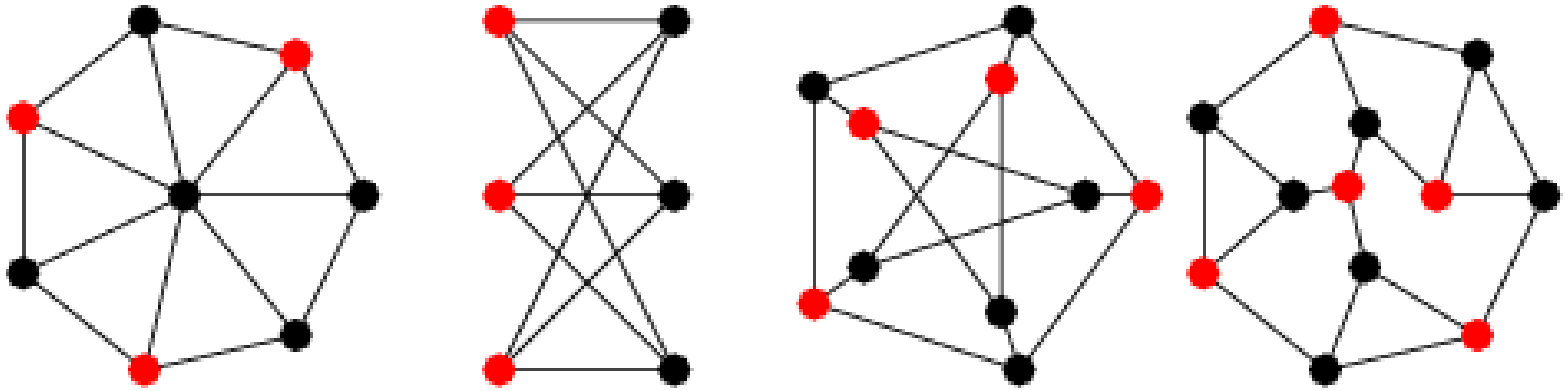
- **CLIQUE**: given a graph G=(V,E) of ***n* vertices** and **parameter *k***, is there a set of ***k* vertices V'** such that **every two** vertices in **V'** are **connected**?
- A **Clique** is **therefore** a **set** of **vertices** that are **pairwise adjacent**. A **pair** of **vertices** (**a**,**b**) is **adjacent** (**connected**) if **there** is an **edge joining** them.





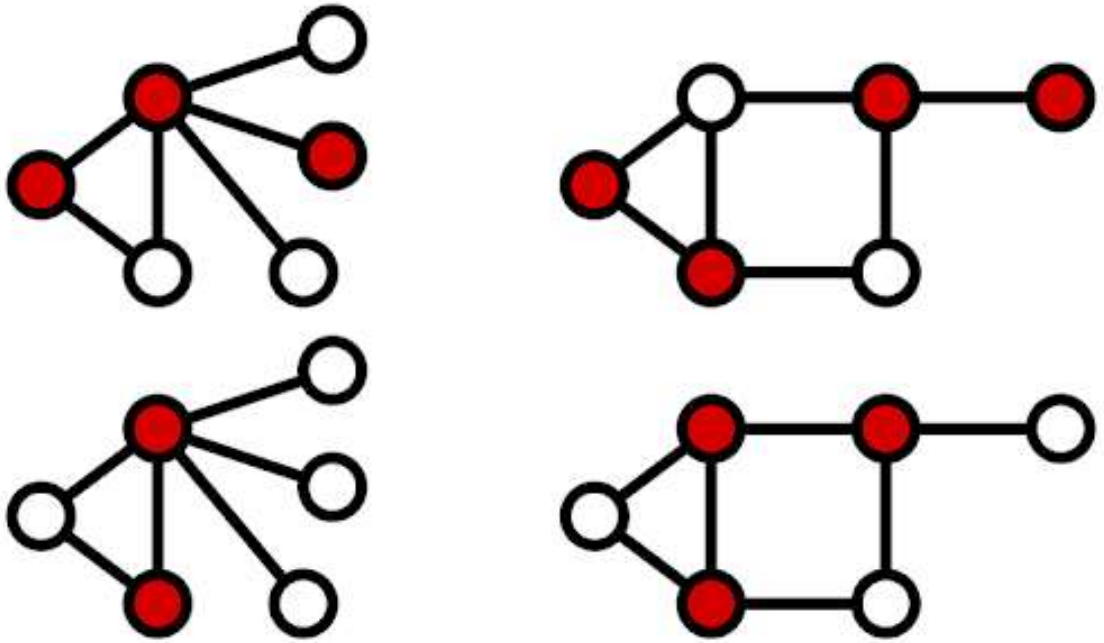
What is an Independent Set?

- **INDEPENDENT SET**: given a graph G=(V,E) of *n* **vertices** and **parameter *k***, is there a set of *k* **vertices V'** such that **no two vertices** in **V'** are **connected**?
- An **independent set** is **therefore** a **set** of **vertices** that are **pairwise not adjacent** (**connected**). For any **pair** of **vertices** (**a**,**b**) in the **independent set** there is **no edge between a** and **b**.
- One can **say** that an **independent set** is the **converse** of a **Clique**.



What is a Vertex Cover?

- **Vertex Cover (VC)**: given a graph G=(V,E) of ***n*** **vertices** and **parameter *k***, is there a set of ***k*** **vertices V'** such that **every edge** has at **least** one end **among** selected **nodes**?
- A **vertex cover** is a **set** of **vertices** such that **every edge** in the **graph** is **adjacent** (**connected to**) at **least one** of the **vertices** in the **Vertex Cover**.
- Note that, for **every edge** in the graph, at **least** one of the **adjacent vertices** must be in a **Vertex Cover**.
- **Can** you **see** that the **vertices** that are **not** in the **Vertex Cover form** an **Independent Set**?
- **Why**?



VC-IS-Clique Problems – Simple but Cool

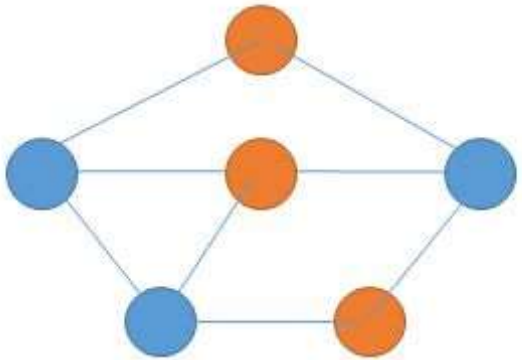
- **Vertex Cover (VC)**: given a graph G=(V,E) of *n* **vertices** and **parameter** *k*, is there a set of *k* **vertices V'** such that **every edge** has at **least** one end **among** selected **nodes**?
  NP-Hard Optimization problem: **Minimize**

- **CLIQUE**: given a graph G=(V,E) of *n* **vertices** and **parameter** *k*, is there a set of *k* **vertices V'** such that **every two** vertices in V' are **connected**?
  NP-Hard Optimization problem: **Maximize**

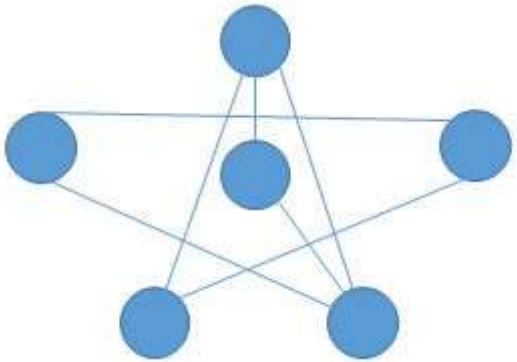
- **INDEPENDENT SET**: given a graph G=(V,E) of *n* **vertices** and **parameter** ***k***, is there a set of *k* **vertices V'** such that **no two vertices** in V' are **connected**?
  NP-Hard Optimization problem: **Maximize**

Reducing Independent Set to Clique

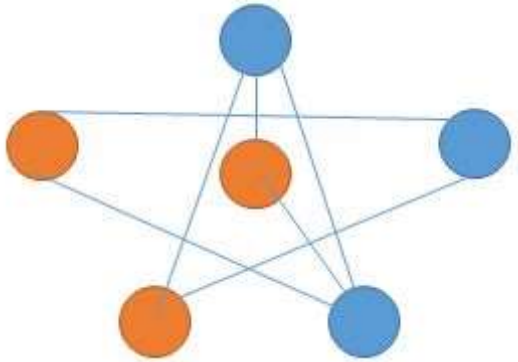
Independent Set
in the Graph



Complimentary of
Independent Graph

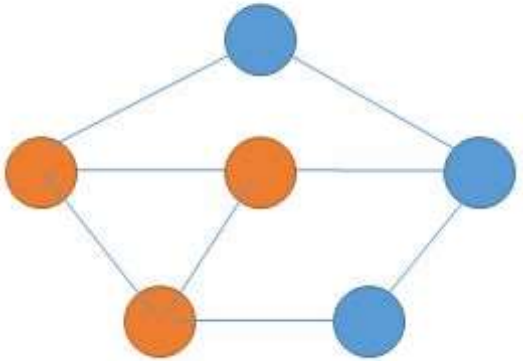


Independent Set In
Complimentary of
Independent Graph



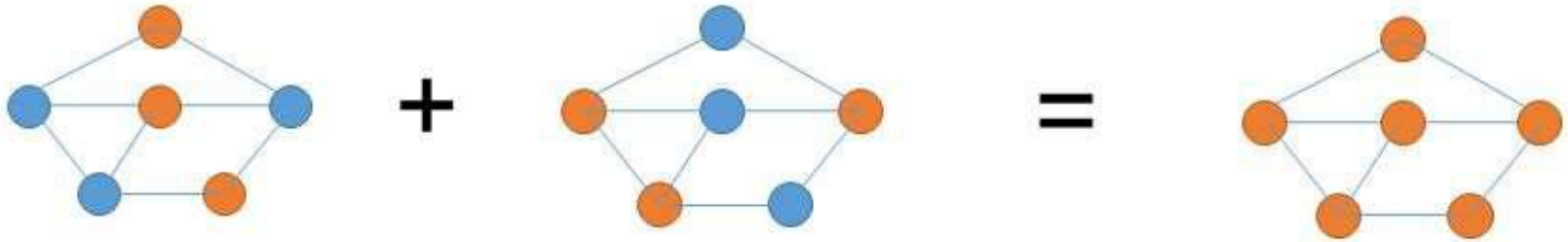
= Clique

Actual Clique
in the Graph

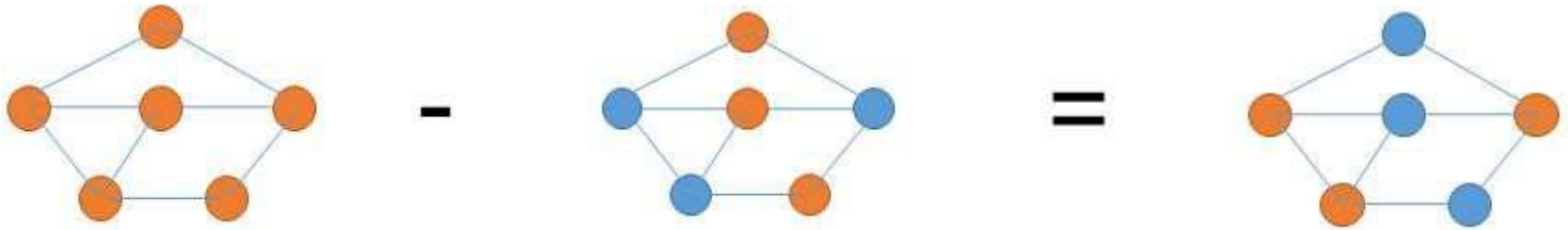


Clique is as hard as Independent Set

Independent Set to Vertex Cover



Independent Set Vertex Cover



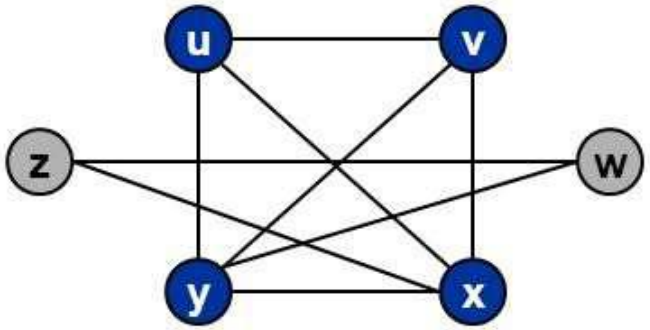
Independent Set Vertex Cover

Vertex Cover to Clique (homework)

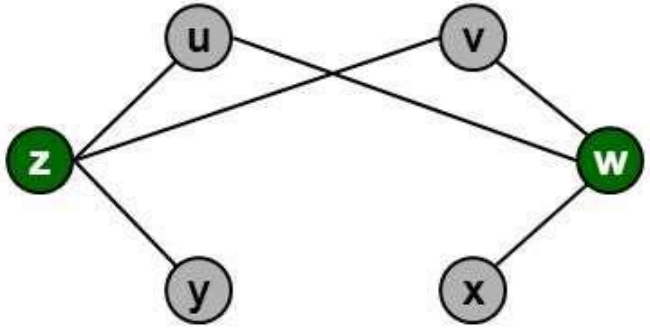
Vertex Cover and Clique

Claim. VERTEX COVER $\equiv_P$ CLIQUE.

- Given an undirected graph G = (V, E), its complement is G' = (V, E'), where E' = { (v, w) : (v, w) $\notin$ E}.
- G has a clique of size k if and only if G' has a vertex cover of size |V| - k.



G
Clique = {u, v, x, y}



G'
Vertex cover = {w, z}

Strategies for Turing Reduction

To show that P_1 Turing reduces to P_2

Restriction

This is the easiest reduction. Show that P_2 is a special case of P_1.

Local Replacement

More difficult but relatively uncomplicated. Ex. see proof of reduction of SAT to 3-SAT.

Component Design

The most difficult type of reduction. Ex. 3-SAT to VC or VC to Hamiltonian Circuit.

Approximation Algorithms

The area of **Approximation Theory** is a **very large** area so we will **introduce** the **basic theory** and then **proceed** to **discuss** a **number** of **approximation techniques**.

One **technique** with **no provable bounds** will be **Genetic Algorithms**.

Approximation Algorithms

We are **going** to **see** that there are **many different ways** to **deal** with **NP-Complete** and **NP-Hard** problems.

We will **also see** that some **NP-Hard** and **NP-Complete** problems can be **approximated** but others **cannot** be **approximated**.

If fact for some problems finding an *bounded* polynomial-time approximation is NP-Hard!

Dealing with NP-Hard/Complete Problems

1. Sit **down** and **Cry** like a **baby**! ✘
2. If **instance** is **small** then use **brute-force**. ✓
3. Find **certain conditions** under which **problem** becomes **tractable** (**feasible**). ✓
4. Try to **find near-optimal** solutions in **polynomial** time. ✓

The **last approach** uses an **approximation algorithm**.

Some Approximation Techniques

1. **Greedy** Algorithms
2. **Dynamic** Programming
3. **Evolutionary** Computation
4. **Polynomial Algorithms** with **approximation bounds**.
5. **Neural Networks** and other **Vector Space** methods such as **SVMs**

The **last approach** includes a '**learning**' component.

Some Definitions

Approximation algorithm: An algorithm that returns *near-optimal* solutions (in polynomial time), i.e. is "provably good" is called an ***approximation algorithm***.

Performance Ratio (Ratio Bound): We say that an approximation algorithm for a problem *P* has a **ratio bound** of $\rho(n)$ if for any instance of size *n*, the cost *C* of the solution produced by the approximation algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$max(C/C^*, C^*/C) \leq \rho(n)$$

Performance Ratios

In **Maximization** problems:

$$0<C\leq C^*\ ,\ \rho(n) = C^*/C$$

In **Minimization** Problems:

$$0<C^*\leq C\ ,\ \rho(n) = C/C^*$$

- ρ(n) is never less than 1.
- A **1-approximation** algorithm is the **optimal solution**.
- The **goal** is to find a **polynomial-time approximation** algorithm with **small constant approximation** ratios.

Performance Ratios

Minimization	Maximization
Travelling Salesman Problem (**TSP**) Find shortest Hamiltonian Tour. **Vertex Cover** (**VC**) Least number of vertices that cover all the edges. **Set Cover** (**SC**) Least number of subsets whose union is the set T. **Bin Packing** (**BP**)	**Clique** Largest number of pair-wise connected vertices. **Independent Set** (**IS**) Largest number of pair-wise non-adjacent vertices.
The **estimate** C will be **larger** than the **optimum** C*.	The **estimate** C will be **smaller** than the **optimum** C*

Approximation Schemes

Approximation scheme is an approximation algorithm that takes $\epsilon > 0$ as an input such that for any fixed $\epsilon > 0$ the scheme is a $(1+\epsilon)$-approximation algorithm.

Polynomial-Time Approximation Scheme is an approximation algorithm that runs in time polynomial in the size of the input and also ϵ .

- As the ϵ decreases the running time of the algorithm may increase rapidly:

  For example it might be $O(n^{2/\epsilon})$

Approximation Schemes

We have a **Fully Polynomial-Time Approximation Scheme** when the running time is polynomial not only in n but also in $1/\epsilon$

For example, it could be $O((1/\epsilon)^3 n^2)$

This means the smaller ϵ becomes the longer the algorithm takes to run.

Approximation Schemes

We shall later look at algorithms that find approximate solutions in polynomial time using a Fully Polynomial Time Approximation Algorithm

We shall discuss one NP-Hard problem for which even approximation is NP-Hard.

Finally we will look at a problem where the running time of its approximation algorithms increases as ϵ decreases.

Some Algorithmic Paradigms

- Greedy algorithms
- Dynamic programming

What is a Greedy Algorithm?

Constructs a solution to an *optimization problem* step by step through a sequence of choices that are:

- ***Feasible**, i.e. satisfying the constraints.*
- ***Locally Optimal** (with respect to some neighborhood definition).*
- ***Greedy** (in terms of some measure), and irrevocable.*

For some problems, it yields a ***globally optimal*** solution for every instance. For most, it does not but it can be useful for fast and efficient approximations.

The Greedy Technique

Optimal solutions

- Giving change “normal” coin denominations
- Minimum Spanning Tree (MST)
- Single-source shortest paths in graphs
- Simple scheduling problems
- Huffman Codes

Approximations & Heuristics

- Traveling Salesman Problem (TSP)
- Bin-packing & Knapsack problems
- Many other combinatorial optimization problems

Giving Change

Given unlimited amounts of coins of denominations.
$d_1 > \ldots > d_m$, give change for an amount n with the least number of coins:

- Example: $d_1 = 25c$, $d_2 = 10c$, $d_3 = 5c$, $d_4 = 1c$ and $n = 48c$

Greedy solution is $<1,2,0,3>$
Greedy solution is optimal for any amount and "normal" set of denominations.

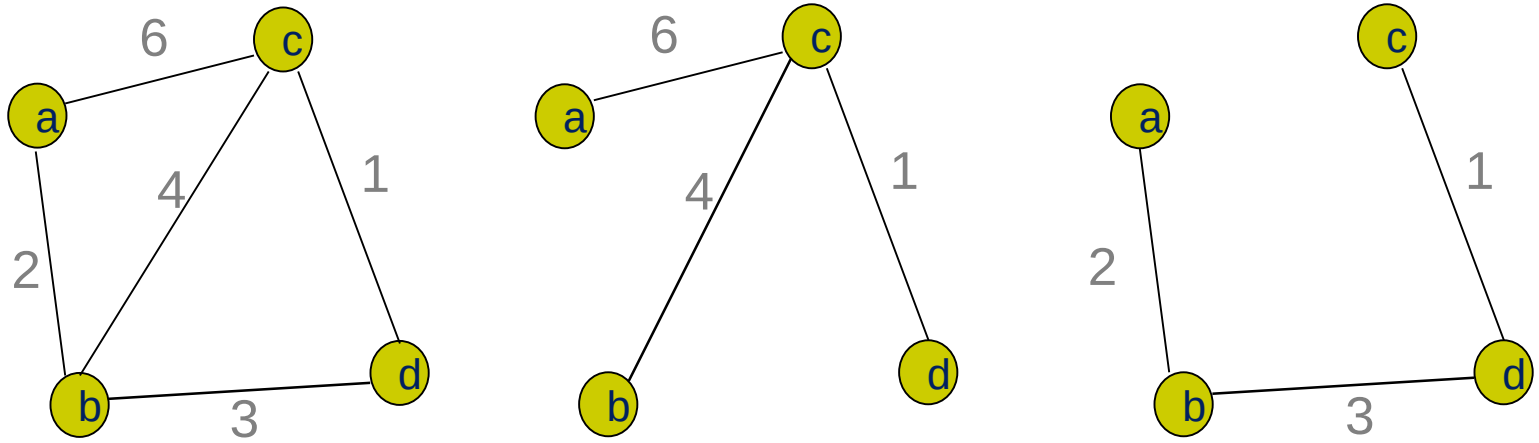
It, however, may not be optimal for arbitrary coin denominations.

- Example: $d_1 = 25c$, $d_2 = 10c$, $d_3 = 1c$, and $n = 30c$

Minimum Spanning Trees (MSTs)

- ***Spanning tree*** of a connected graph *G*: a connected acyclic subgraph of *G* that includes all of *G*'s vertices

- ***Minimum spanning tree*** of a weighted, connected graph *G*: a spanning tree of *G* of the minimum total weight

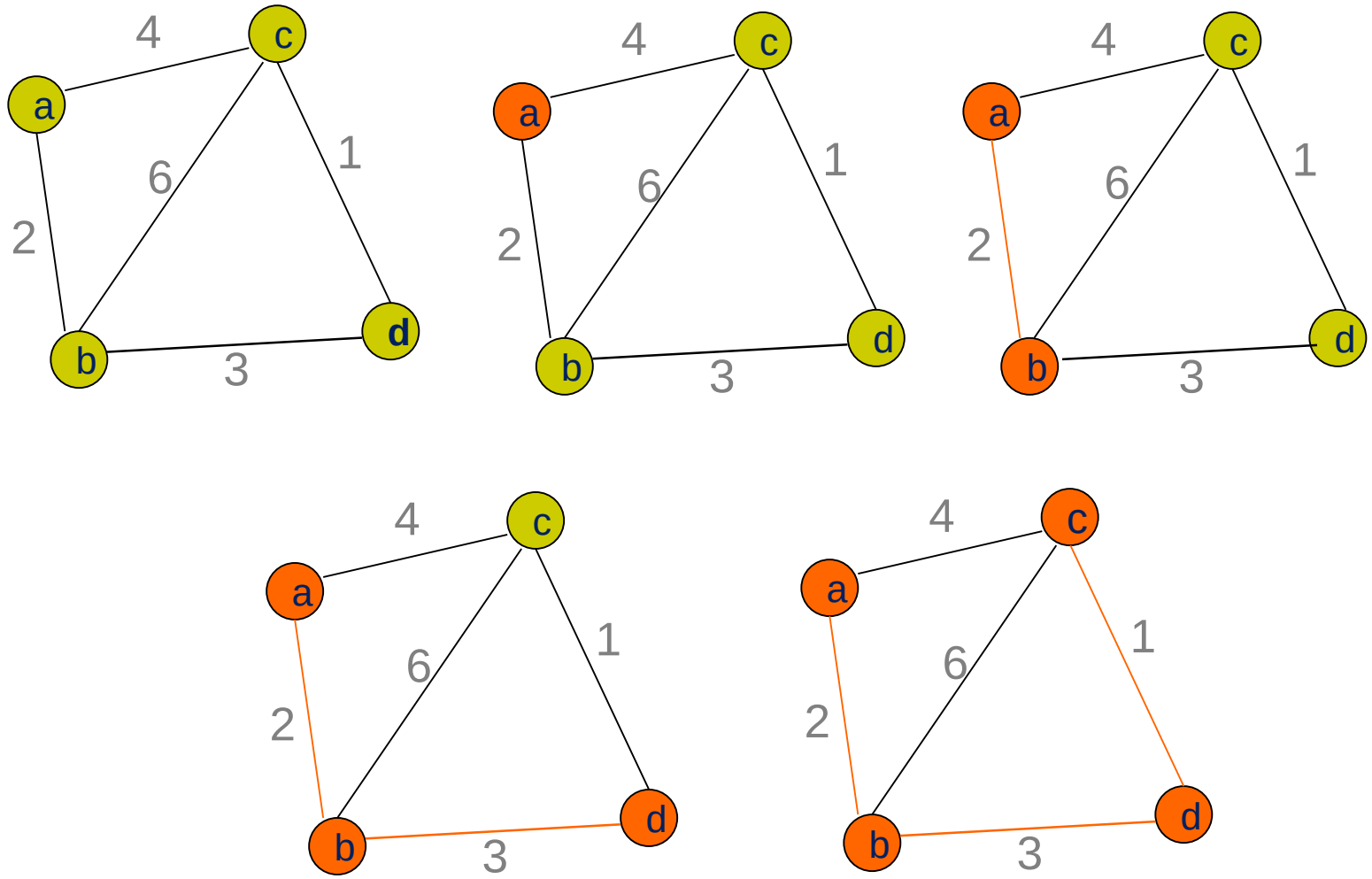


Prim's Greedy Algorithm for MST

- Start with tree T_1 consisting of one (any) vertex and "grow" the tree one vertex at a time to produce MST through ***a series of expanding subtrees T1, T2, …, T_n***

- On each iteration, ***construct Ti+1 from Ti*** by adding vertex not in ***Ti that is*** closest to those already in ***Ti*** (this is a ***greedy*** step!)

- Stop when all vertices of the tree are included.



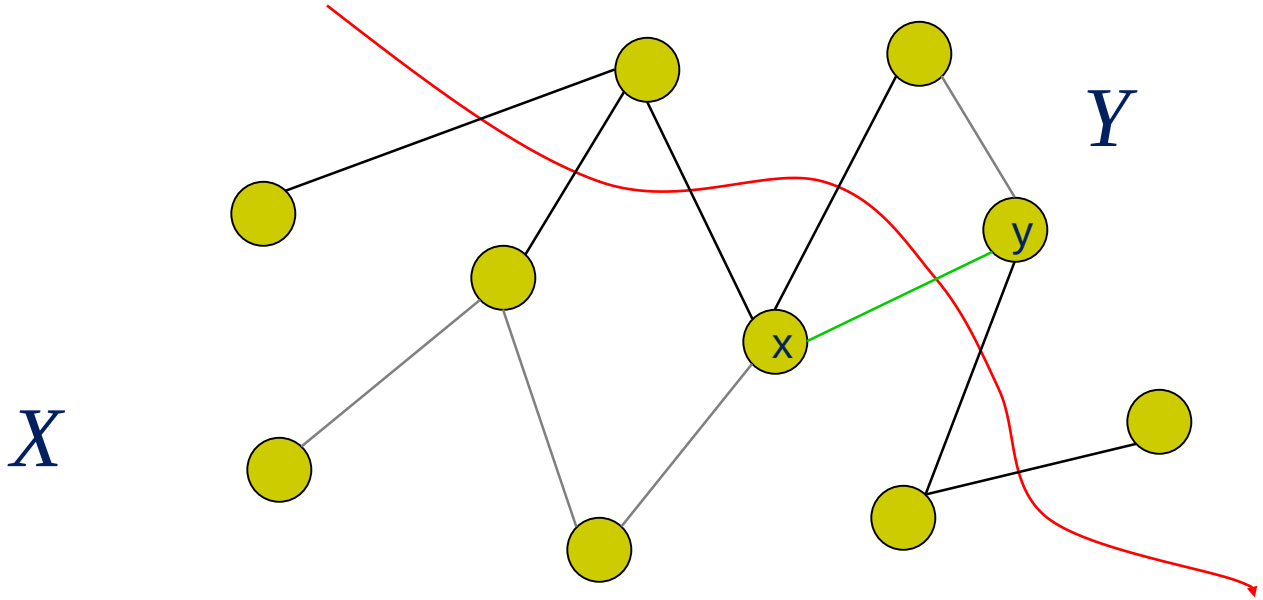
Prim's Greedy Algorithm for MST

Remarks on Prim's Algorithm

- There is a proof by induction that this construction actually yields an MST (CLRS, Ch. 23.1). This proof relies on the ***Cut Theorem***.
- Needs a ***priority queue*** for locating closest fringe vertex.
- **Efficiency**
  - $O(n^2)$ for weight matrix representation of graph and array implementation of priority queue.
  - $O(m \log_2 n)$ for adjacency lists representation of graph with n vertices and m edges and Min-Heap implementation of the priority queue.

The *Cut* Theorem for MSTs

Claim: Let $G = (V,E)$ be a weighted graph and (X,Y) be a partition of V (called a *cut*). Suppose that $e = (x,y)$ is an edge of E across the cut, where x is in X and y is in Y, and e has the minimum weight among all such crossing edges (called a light edge). Then there is an MST containing e.



The Coding Problem

***Coding*: the assignment of bit strings to alphabet characters.**

E.g. We can code {a,b,c,d} as {00,01,10,11} or {0,10,110,111} or {0,01,10,101}.

Codewords: bit strings assigned for characters of alphabet

Two types of codes

- *fixed-length encoding* (e.g., ASCII)
- *variable-length encoding* (e,g., Morse code)

Prefix-free codes (or prefix-codes) - no codeword is a prefix of another codeword since otherwise the receiver will not be able to decode the message. For above we can use the following {0,10,110,111}.

Problem: If frequencies of the character occurrences are known, what is the best binary prefix-free code?

The one with the shortest average code length. The average code length represents on the average how many bits are required to transmit or store a character.

Building a Huffman Code

Building a Huffman Code to solve the coding problem can be done using a very simple polynomial-time greedy algorithm.

An example is here HuffmanSample.ppt

Try at home and then we can discuss.

Dynamic Programming

Dynamic Programming is a general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping sub-instances.

Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems.

"***Programming***" here means "***planning***"

Dynamic Programming

Main Idea

- set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
- solve smaller instances once
- record solutions in a table
- extract solution to the initial instance from that table

Dynamic Programming Examples

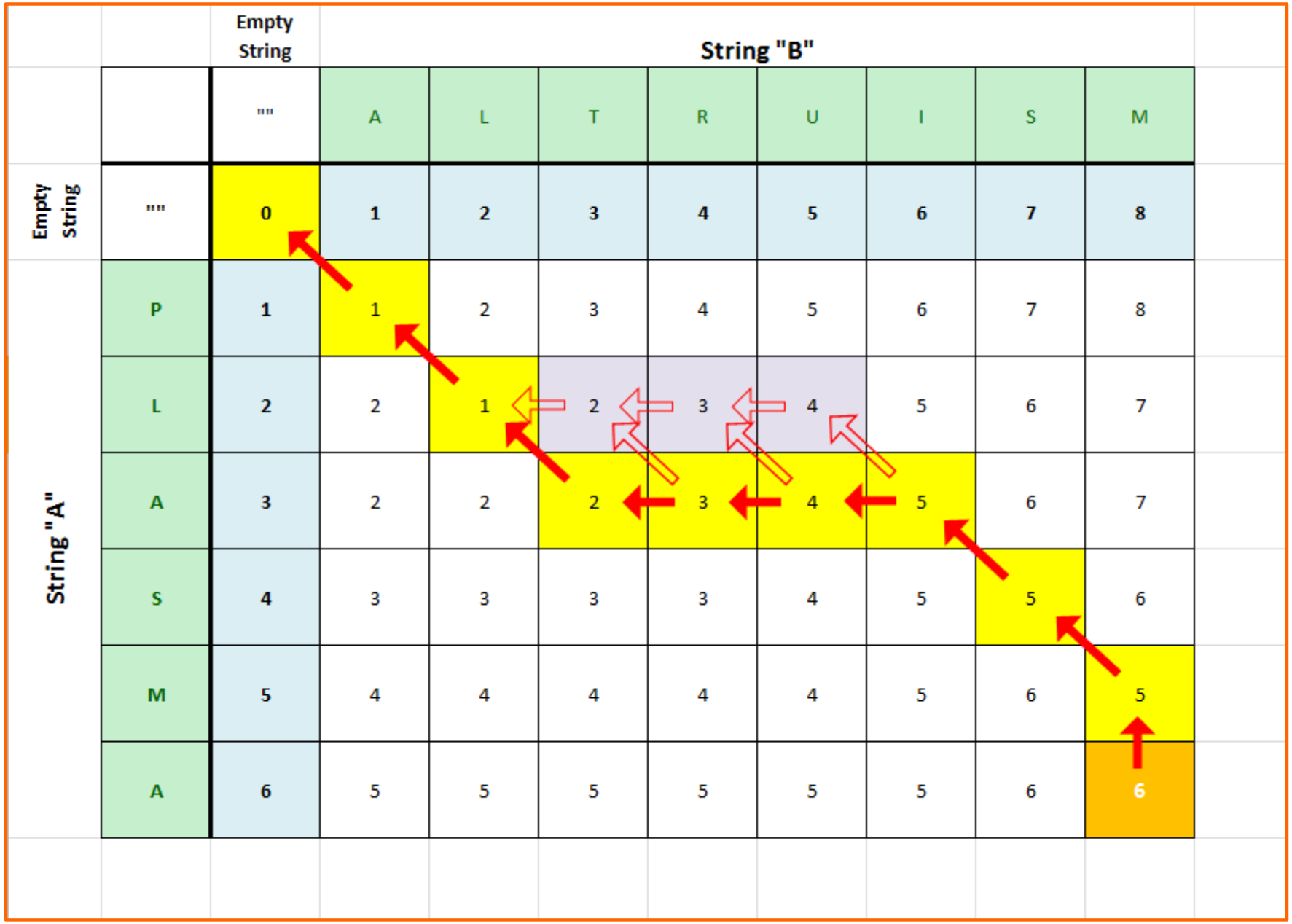
- Computing a binomial coefficient
- Longest Common Subsequence (LCS)
- Warshall's algorithm for transitive closure
- Floyd's algorithm for all-pairs shortest paths
- Constructing an optimal Binary Search Tree BST
- Some instances of difficult discrete optimization problems such as TSP
- Computing Fibonacci Numbers
- ***Levenshtein String-Edit Distance (using Wagner-Fischer Algorithm)***

Main Loop of Wagner-Fischer Alg.

```
for i=2:Len(S1)
  for j=2:Len(S2)
    if S1[i] = S2[j] then
       M[i,j]=M[i-1,j-1]+2
    else
       M[i,j]=min(M[i-1,j]+1,
                  M[i,j-1]+1,
                  M[i-1,j-1]+2)
  end
end
```

The above code segment requires n^2 steps.

Levensthein Distance



		Empty String	String "B"							
		""	A	L	T	R	U	I	S	M
Empty String	""	0	1	2	3	4	5	6	7	8
String "A"	P	1	1	2	3	4	5	6	7	8
	L	2	2	1	2	3	4	5	6	7
	A	3	2	2	2	3	4	5	6	7
	S	4	3	3	3	3	4	5	5	6
	M	5	4	4	4	4	4	5	6	5
	A	6	5	5	5	5	5	5	6	6

Some Examples of Approximation Algorithms

- Vertex Cover (VC)

  2-approximation algorithm

- Traveling Salesman Problem (TSP)

  2-Approximation for Euclidean TSP only.

- Set Cover (SC)

  Time complexity increases with instance size.

The Vertex Cover Problem

A ***vertex-cover*** of an undirected graph G is a subset of its vertices such that it includes at least one vertex from each edge.

The problem is to find ***minimum size*** of vertex-cover of the given graph.

As explained before this problem is an NP-Hard optimization problem.

The Vertex Cover Problem

Finding the **optimal** solution is **hard** (its **NP-Hard**!) but finding a **near-optimal** solution is **easy**.

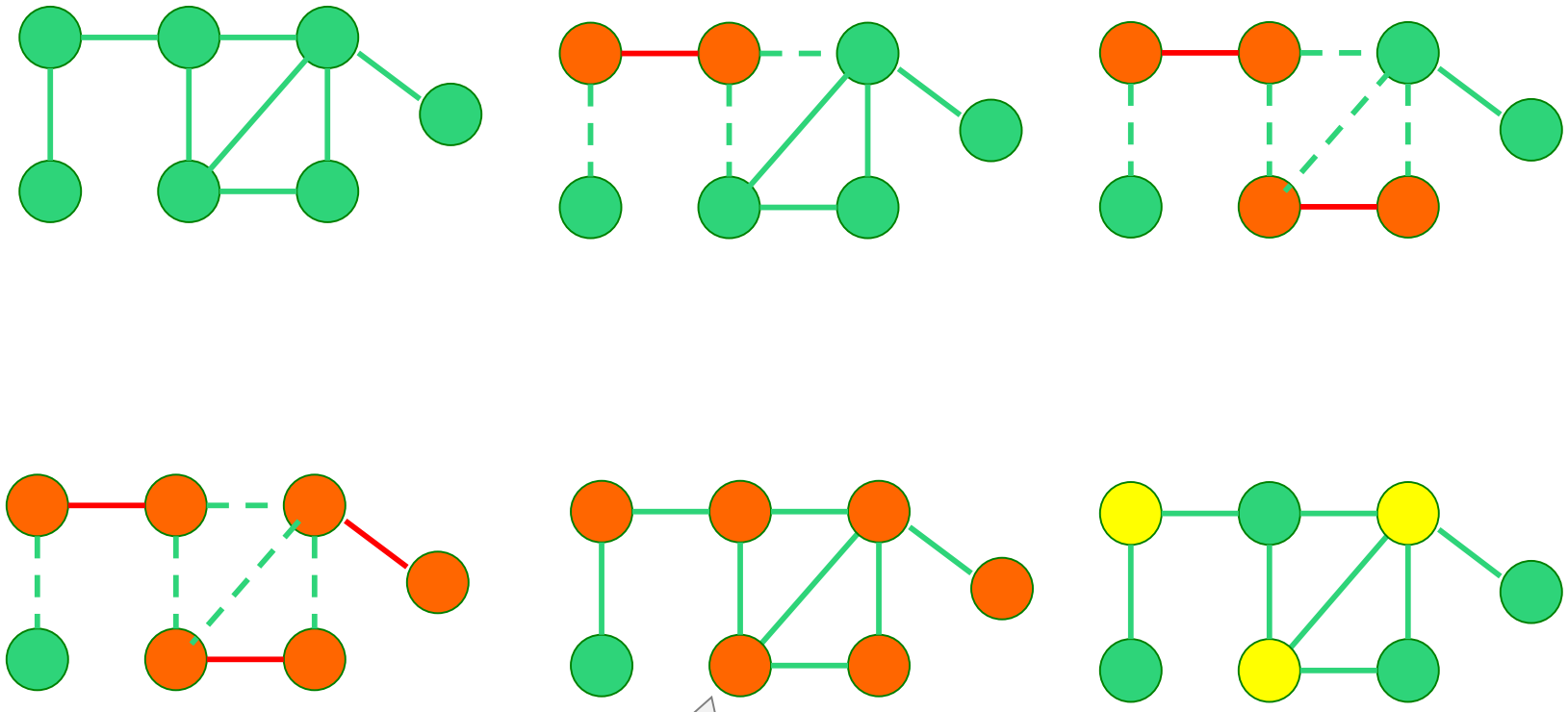
There is a **very simple** greedy **2-approximation** algorithm that returns a vertex-cover that is **not more** than **twice** of the **size optimal** solution.

The Vertex Cover Problem

APPROX-VERTEX-COVER(G)

```
1       C ← Ø
2       E′ ← E[G]
3       while E′ ≠ Ø
4               do let (u, v) be an arbitrary edge of E′
5                   C ← C U {u, v}
6                   remove every edge in E′ incident on u or v
7 return C
```


The Vertex Cover Problem



The Vertex Cover Problem

This is a polynomial-time 2-approximation algorithm. (**Why**?)

Because:

APPROX-VERTEX-COVER is O(V+E)

$|C^*| \geq |A|$

$|C| = 2|A|$

$|C| \leq 2|C^*|$

Optimal

Selected Edges

Selected Vertices

All Edges must be covered

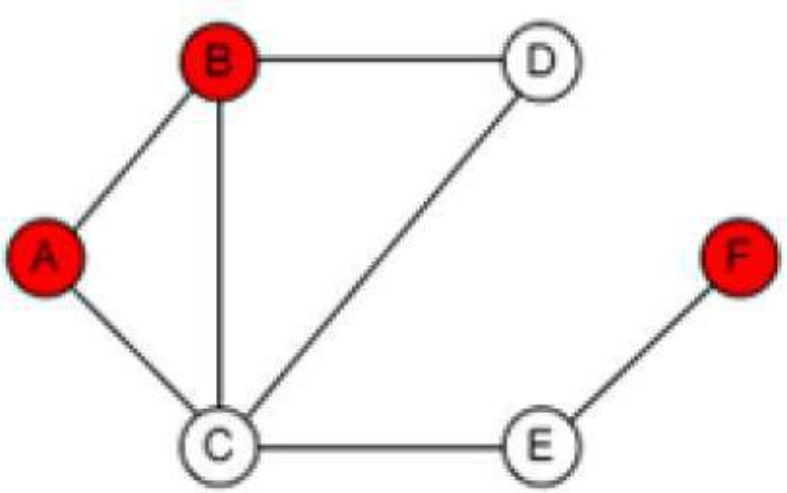


Figure 1

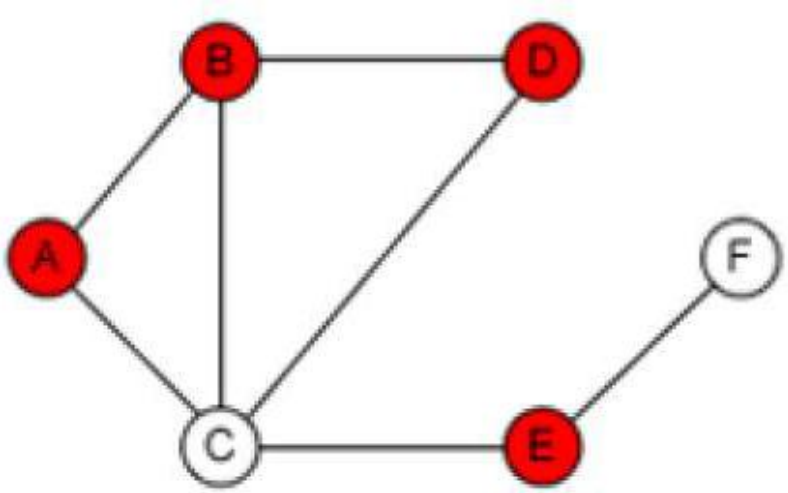


Figure 2

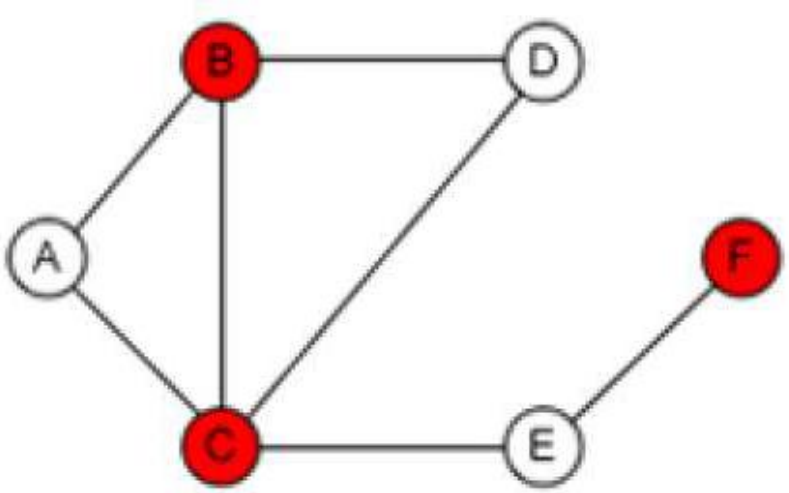


Figure 3

Explaining the 2-Approximation Bound

APPROX-VERTEX-COVER is O(V+E)

|C*| ≥ |A|

|C| = 2|A|

|C| ≤ 2|C*|

Optimal

Selected Edges

Selected Vertices

TSP Approximation

Given an complete, undirected, weighted Graph ***G*** we are to find a ***minimum cost Hamiltonian cycle***.

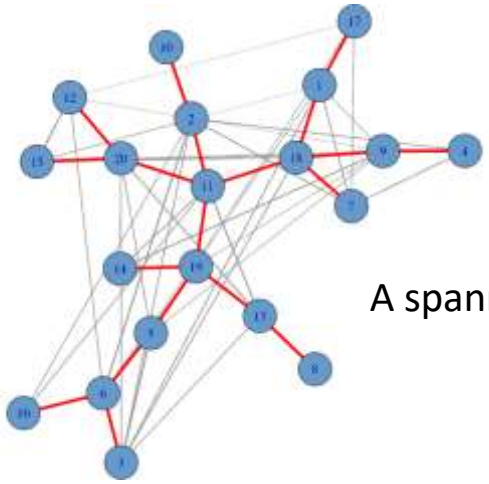
Satisfying the triangle inequality or not this problem is NP-Hard. If the graph satisfies the triangle inequality the problem is called ***Euclidean TSP.***

We now present a simple greedy polynomial-time 2-approximation algorithm based on Prim's MST algorithm.

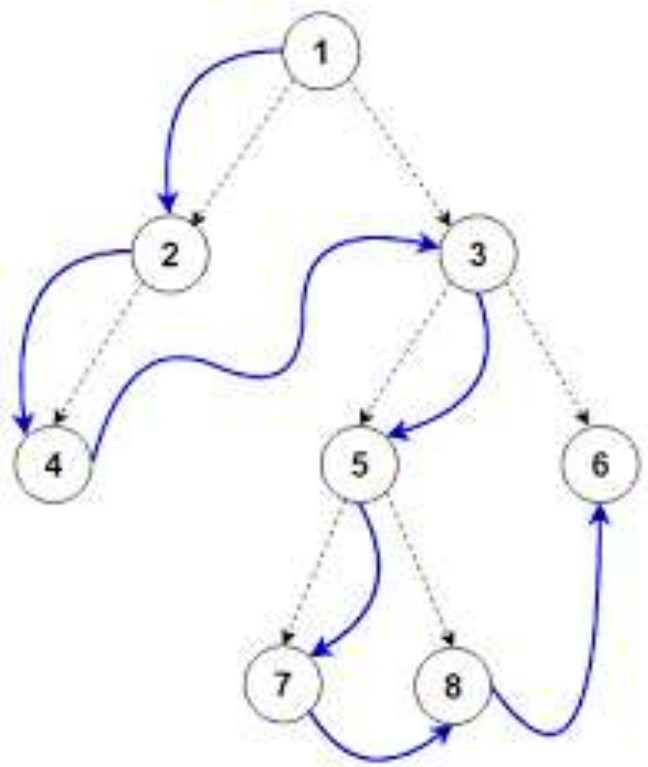
Euclidean TSP Approximation

APPROX-TSP-TOUR(G, c)

- Select a vertex ***r*** ϵ *V[**G**]* to be root.
- Compute a **MST** for ***G*** from root r using ***Prim's Alg***.
- ***L***=list of vertices in ***preorder*** walk of that **MST**.
  Preorder means you **visit** the **root first** and then the **children left** to **right**.
- **Return** the Hamiltonian cycle ***H*** in the order ***L***.

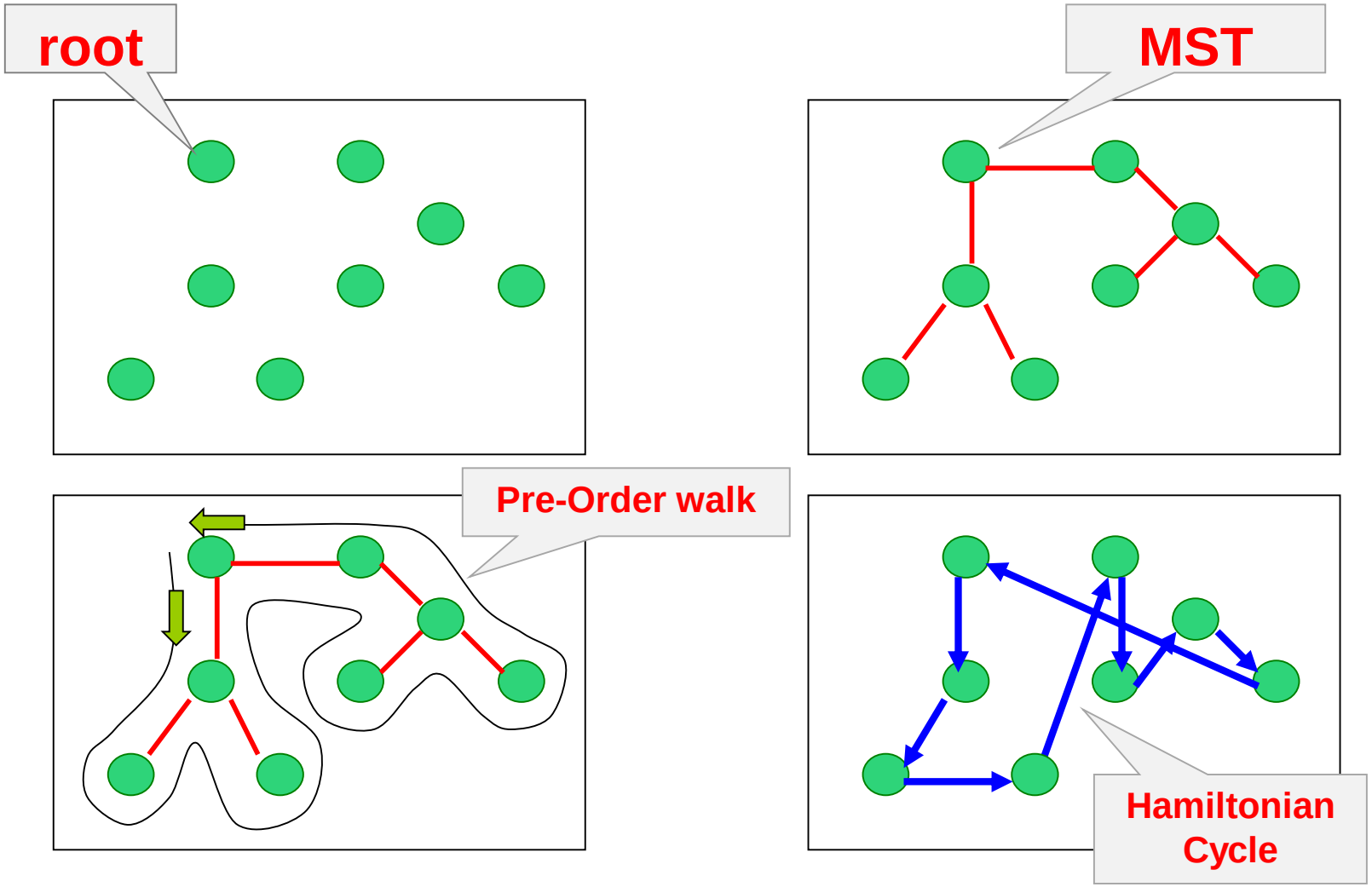


A spanning tree does not have to be binary



Preorder: 1, 2, 4, 3, 5, 7, 8, 6

Euclidean TSP Approximation



TSP Approximation in General

Theorem

If P ≠ NP, then for any constant ρ≥1, there is no polynomial time ρ-approximation algorithm for TSP.

The proof is by contradiction. If this was true for general TSP (not necessary complete) then such an algorithm could be used to find a Hamiltonian Cycle which itself is NP-Complete and then P = NP!

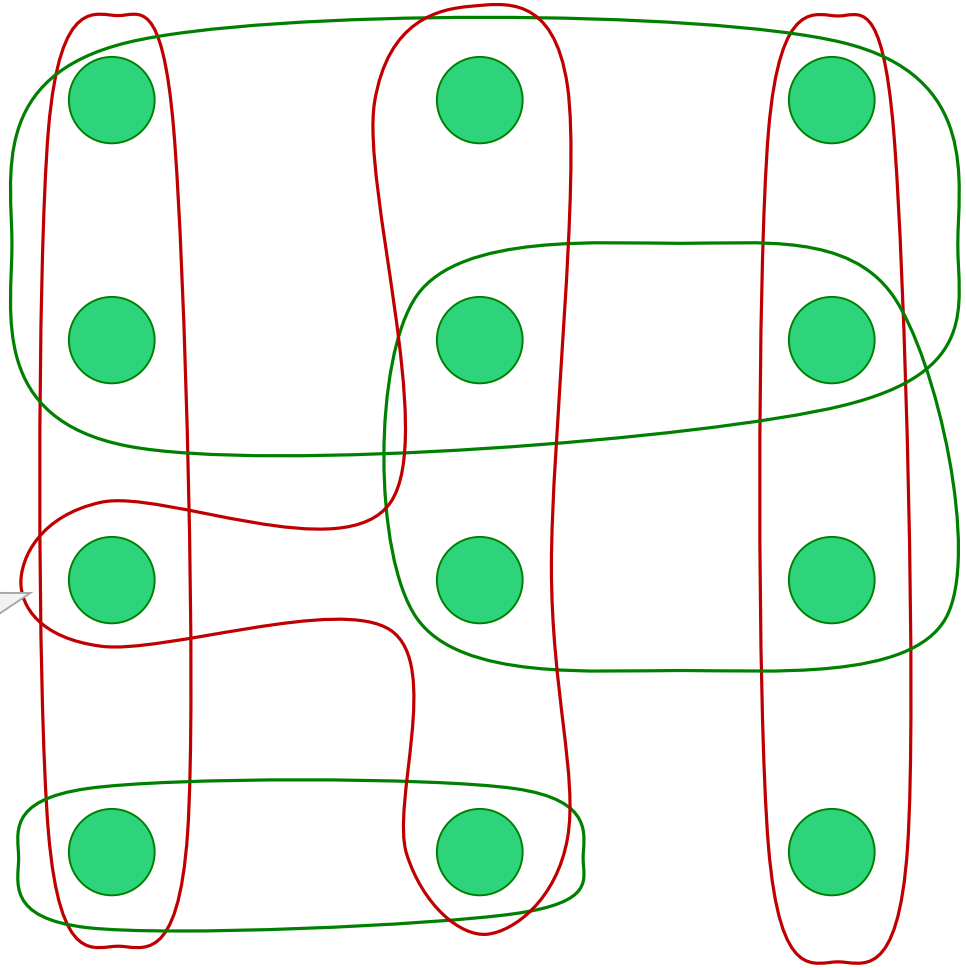
Approximaton for Set Cover (SC)

We have given ***(X,F)*** :

- ***X*** : a finite set of elements.
- ***F*** : family of subsets of ***X*** such that every element of ***X*** belongs to at least one subset in ***F***.
- Solution ***C*** : subset of ***F*** that Includes all the members of ***X***.

The ***greedy approximation algorithm*** we shall present has a time complexity that increases with the size of the instance.

An Instance of Set Cover



Minimal Covering set

size=3

A Greedy Set Cover Algorithm

GREEDY-SET-COVER(X,F)

```
1 M ← X
2 C ← Ø
3 while M ≠ Ø do
4         select an S ϵ F that maximizes |S ∩ M|
5         M ← M – S
6         C ← C U {S}
7 return C
```

Greedy (Non-Optimal) Set Cover

Greedy Set Cover – Time Complexity

This greedy algorithm is polynomial-time ρ(n)-approximation algorithm

- **ρ(n)=H(max{|S| : S Є F})**

The proof is beyond of scope of this course.

H(x) denotes the x^{th} Harmonic number.

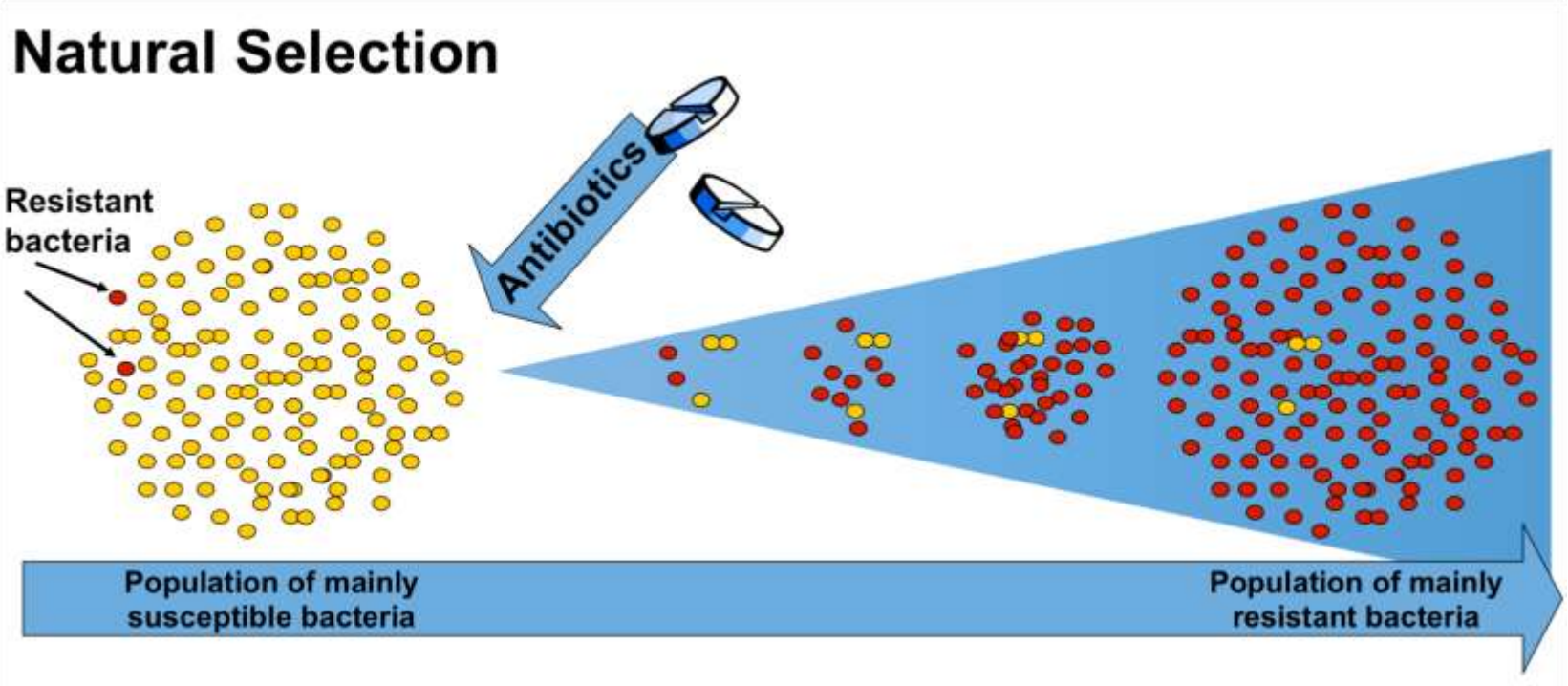
A complete exposition of the proof can be found in CLRS.

Genetic Algorithms

- A **genetic algorithm** (or **GA**) is a search technique used in computing to find optimal or approximate solutions to NP-Hard optimization and search problems.
- Genetic algorithms are categorized as **global search heuristics**.
- Genetic algorithms are a particular class of **evolutionary algorithms** that use techniques inspired by evolutionary biology such as inheritance, mutation, selection, and crossover (also called recombination).

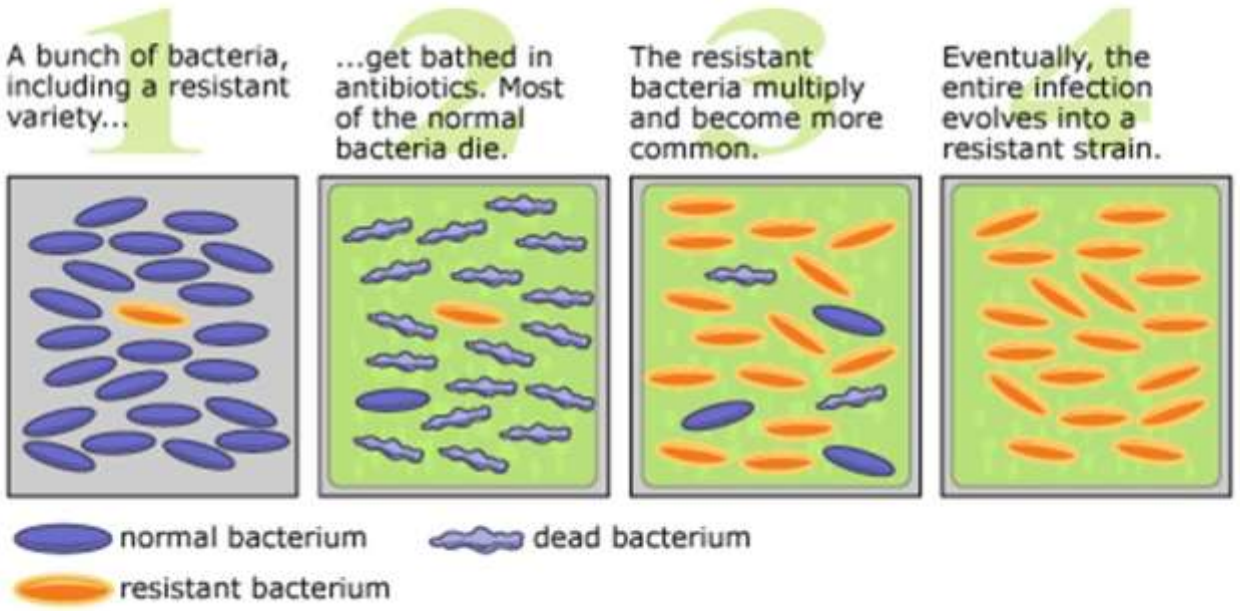
Genetic Algorithms - Motivation

Genetic Algorithms - Motivation



Genetic Algorithms - Motivation

Natural selection leads to the development of antibiotic resistant bacteria

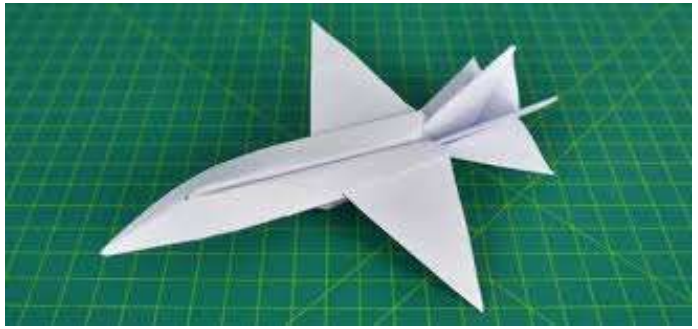


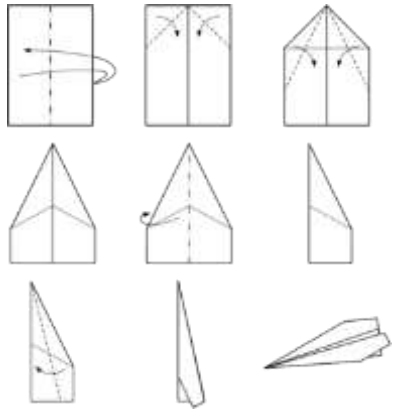
Genetic Algorithms - Motivation

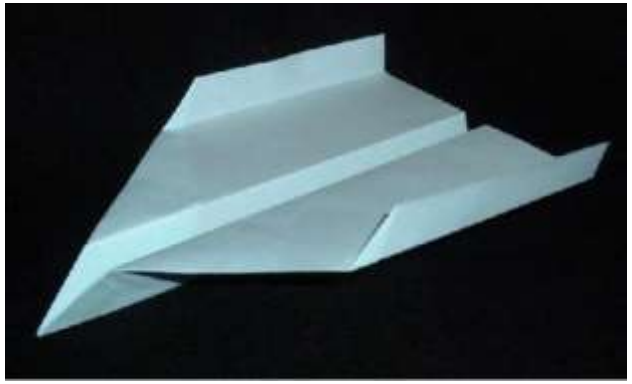
- How do bacteria modify their genome to become resistant to antibiotics?
- A genome has billions of DNA bases. How do you modify a genome to make a species ‘better’ – react to a changing environment?
- Darwin explained how this happens in his The Origin of Species.
- Species, Darwin argued, evolve in order to adapt to a changing environment.

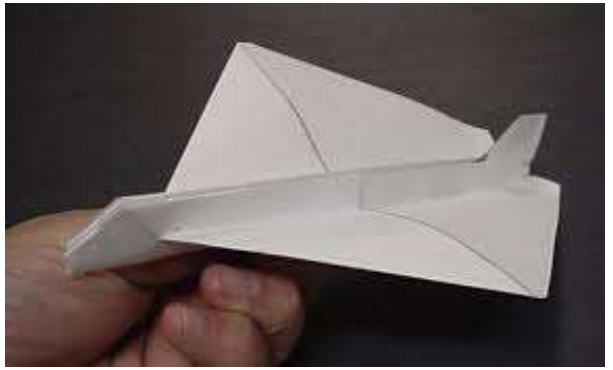
Genetic Algorithms - Motivation

- How does one design a paper plane?
- Try many (billions) of different designs and choose the best one.







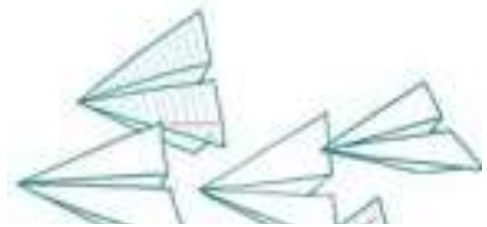


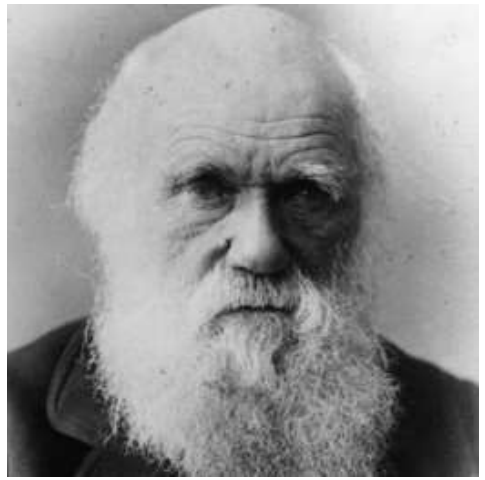
Genetic Algorithms

- Genetic Algorithms (GAs) are now a very popular approximation technique.
- GAs have a stochastic component and can therefore be, in some sense, considered ***non-deterministic***.
- GAs are often referred to as a ***meta-heuristic***. This is because they employ general, non-problem specific, heuristics.
- Unlike Neural Networks, there is no general GA algorithm. One usually designs a GA for the task at hand.
- Covered in more depth in **ICS2207** / **ICS3222**

Genetic Algorithms

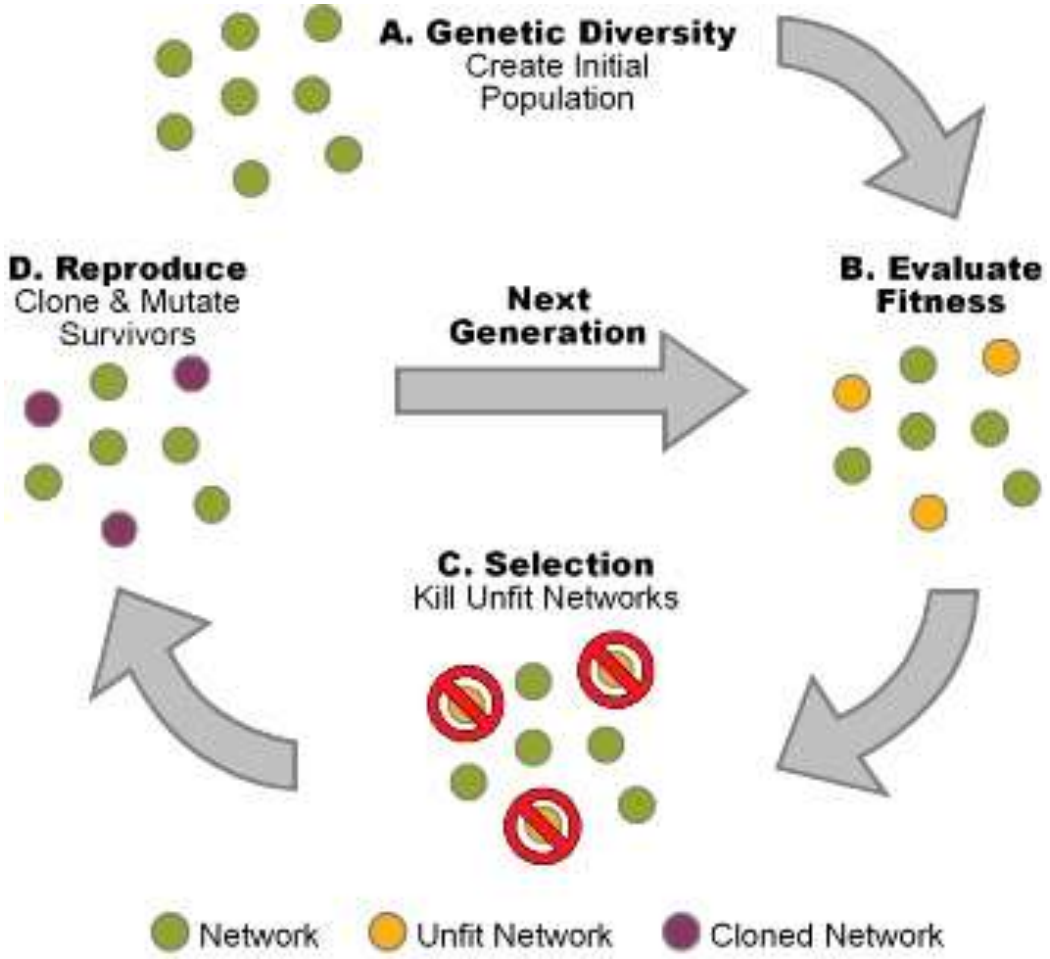
- **Genetic Algorithms** are a **popular metaheuristic** (**search technique**). based on **Darwinian evolution**.
- **Suppose** you **want** to **design** a **paper plane**.
- Represent a **paper plane** by a **chromosome** (a **string**).
- **Create** an **initial random population** of chromosomes.
- Use **cross-over** & **mutation** to **increase** the **population**.
- **Apply** a **fitness function** to **select** the **best individuals**.
- **Repeat**.

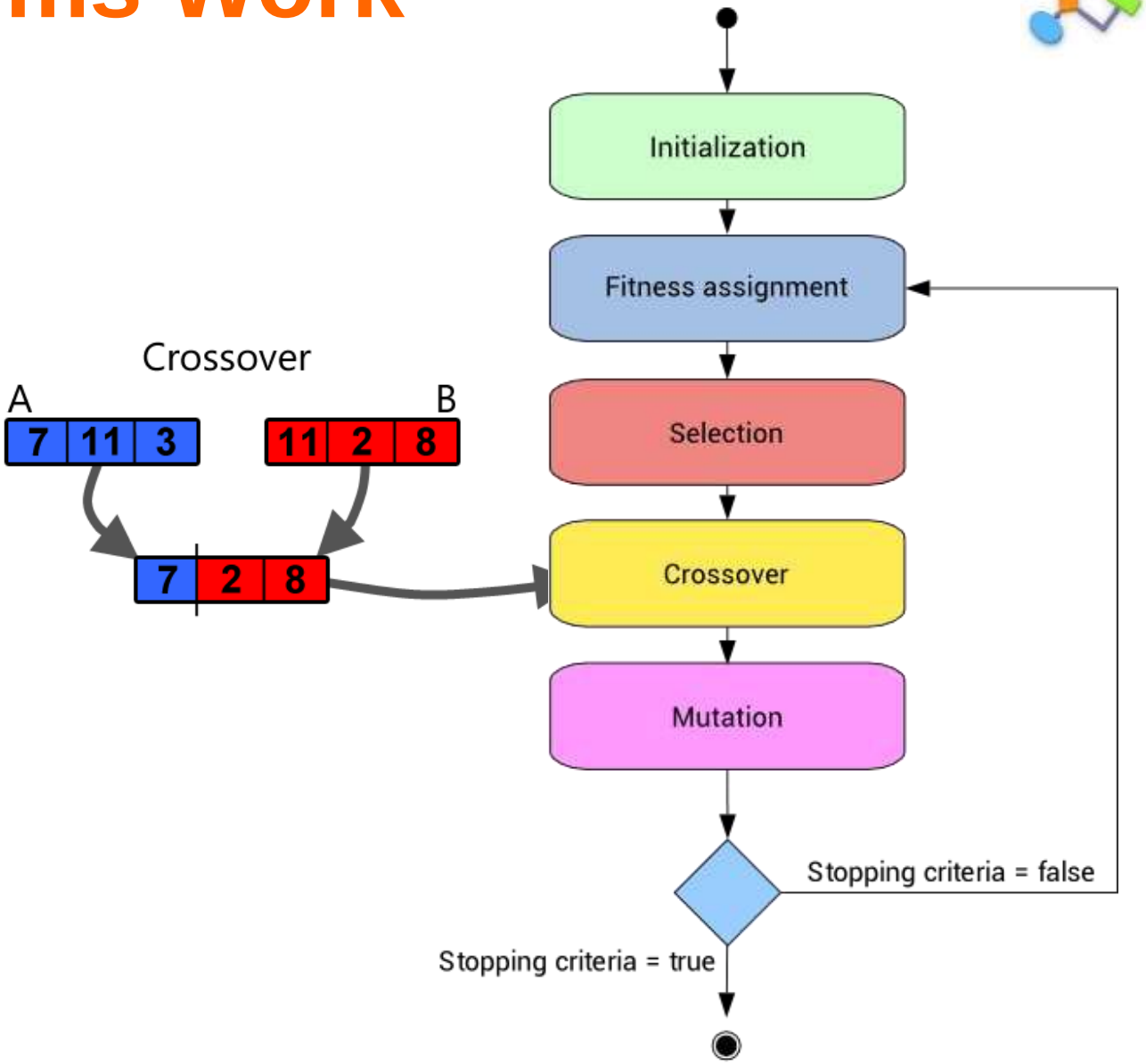




Charles Darwin

How Genetic Algorithms Work

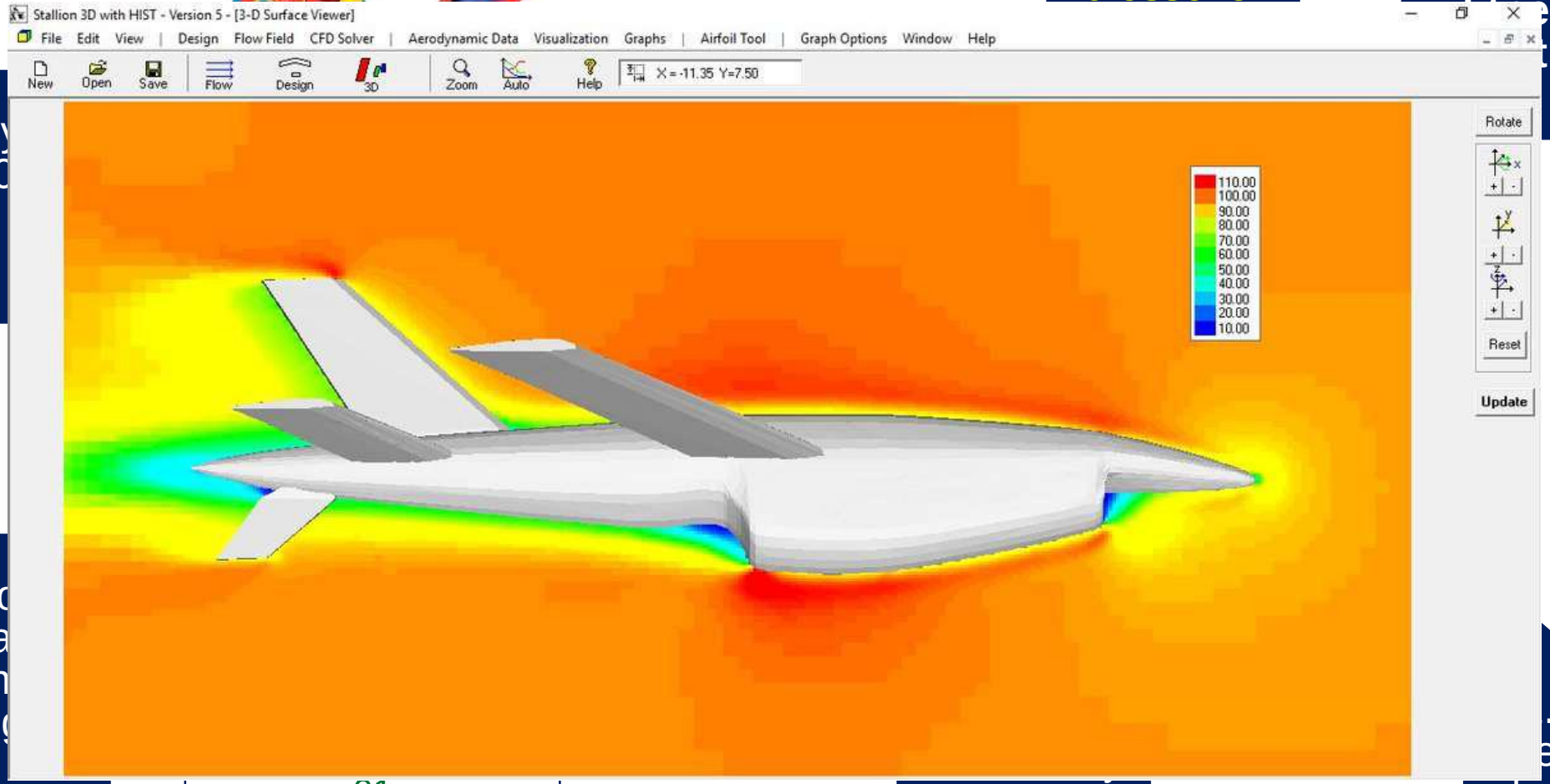




How Genetic Algorithms Wo

- Randomly create 100 paper planes

- Enco the pla as bin strin



01

- Create 'baby' planes using crossover

- planes using mutation

- 1. Apply ess tion

- . Select est 100 planes and repeat

Approximation of NP-Hard Problems

L-Università ta' Malta

Data Structures And Algorithms 2

John Abela
Department of CIS
Faculty of ICT
University of Malta

john.abela@um.edu.mt

+ 365 79367936
Room 1A/27
FICT Building

Module 4

The

Stable Marriage Problem

Academic Year 2020-21

Faculty of ICT

Department of Computer Information Systems

The Only Algorithm to ever win a Nobel Prize

The Stable Marriage Problem

Lloyd Shapley

Alvin Roth

Partly based on slides by Safa S. Mahmoodian

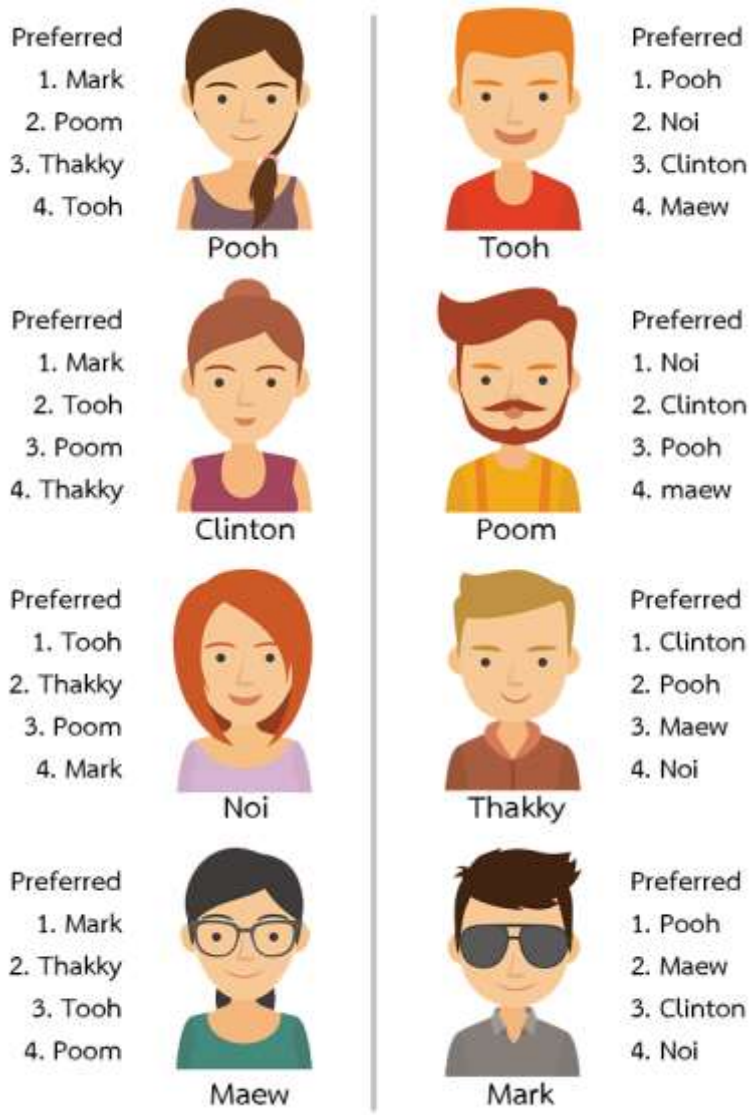
The Stable Marriage Problem

In **mathematics**, **economics**, and **computer science**, the **stable marriage problem** (also **stable matching problem** or **SMP**) is the **problem** of **finding** a **stable matching** between **two equally** sized sets of **elements Y** & **X** (**Y** are **boys** and **X** are **girls**) given an **ordering** of **preferences** for each **element**.

- A **matching** is a **mapping** from the **elements** of **one set** to the **elements** of the **other set**. A **matching** is **not stable** if:
- There is an **element *A*** of the **first matched set** which **prefers** some **given element *B*** of the **second matched set** over the **element** to **which *A*** is **already matched**, and
- ***B*** also **prefers *A*** over the **element** to **which *B*** is **already matched**.

In other words, a **matching** is **stable** when there does **not exist** any **match** (***A***, ***B***) by which ***both A*** and ***B*** would be **individually better off** than they are with the **element** to which they are **currently matched**.

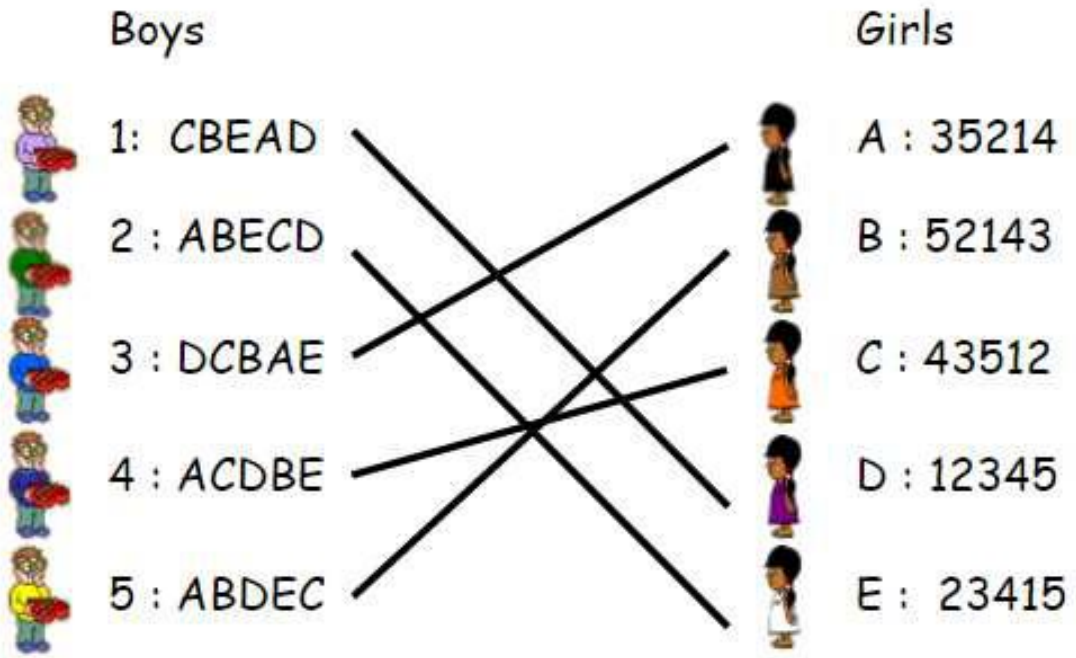
Assuming **heterosexual** pairings:
Given ***n*** **boys** and ***n*** **girls**, where each **person** has **ranked** all **members** of the **opposite sex** in **order** of **preference**, **pair** the **boys** and **girls** together such that there are **no two people** of **opposite** sex who **would both rather** have **each other** than their **current partners**. When there are **no such pairs** of **people**, the set of **marriages** is deemed **stable**.



WIKIPEDIA

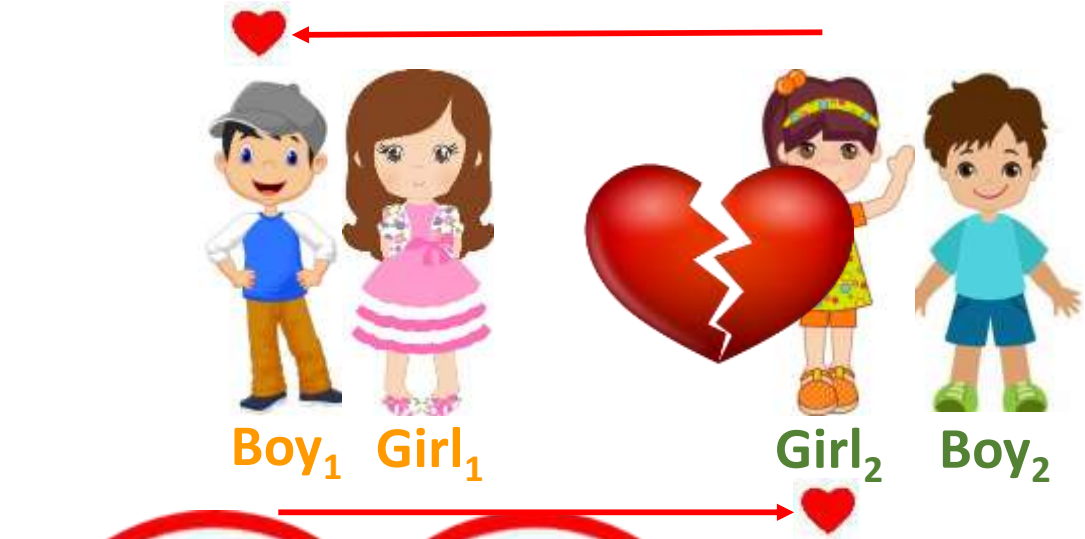
More formally…

- **Consider** a set **Y** of with ***n* boys** (denoted by **numbers**) and a set **X** of ***n* girls** (denoted by **letters**).
- Every **boy** has a **preference list** of the **girls organized** in a **decreasing** order of **desirability** or **preference**.
- Each **girl** has a **similar list** for the **boys**.
- A **marriage**, or a **matching**, ***M*** is a **1-1** and **onto mapping** between the **boys** and the **girls**.
- This will **give** a **complete bi-partite graph** with ***2n* nodes**.
- The **total number** of **persons** (**boys** and **girls**) is **always even**.



Unstable Marriages

- A **marriage** is **said** to be **unstable** if there **exist 2 marriage** couples:
- **Boy_1** – **$Girl_1$**, and
  Boy_2 – **$Girl_2$**, such that:
- **Boy_1** desires **$Girl_2$** more than **$Girl_1$** and
- **$Girl_2$** desires **Boy_1** more than **Boy_2**.
- The pair **Boy_1** - **$Girl_2$** is said to be an '**unstable couple**'. They **both prefer** each **other** over their **assigned partners**.
- A marriage ***M*** is called a '**stable marriage**' if there are **no unstable couples**.

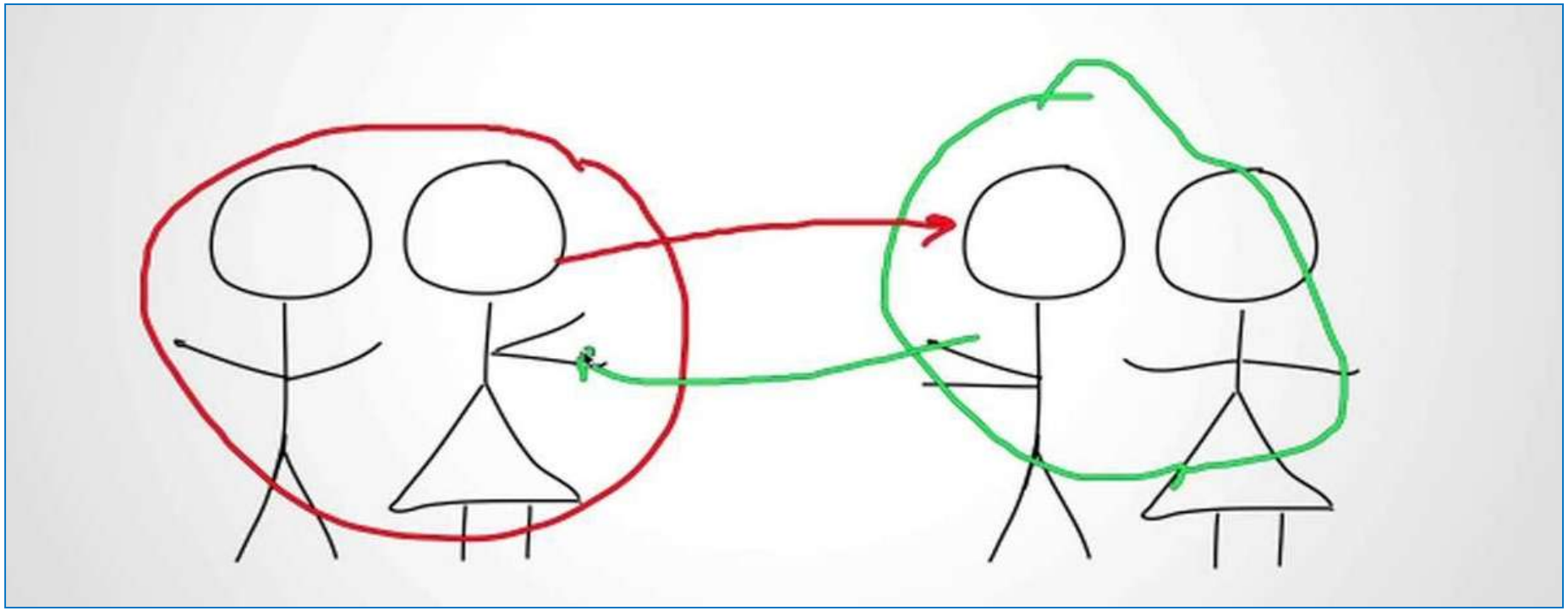


Important Notes

- The **number** of **boys** and **girls** must be **equal**.
- All **boys rank** all the **girls**, and **vice versa**.
- **Rank-ordered lists** are in **decreasing** order of **preference**.
- Each **boy** is **paired** with **one girl** and vice versa.
- Every person **appears** in **exactly one** pair.
- Each **boy** or **girl** is **assigned** by a **deterministic algorithm** – they **cannot make choices** beyond **submitting** their **list** of **preferences**.
- The **algorithm** must **assign** each **boy** to a **girl** in such a **way** that **there** are **no unstable** couples.

Remember – Unstable Marriage

- A **boy prefers** another **girl** (over his **assigned girlfriend**) who, in **turn**, **prefers** the same **boy** (over her **assigned boyfriend**).



The Stable Marriage Problem

The problem is:

"Given the preference rankings of a group of *k* boys and *k* girls, construct a stable marriage arrangement."

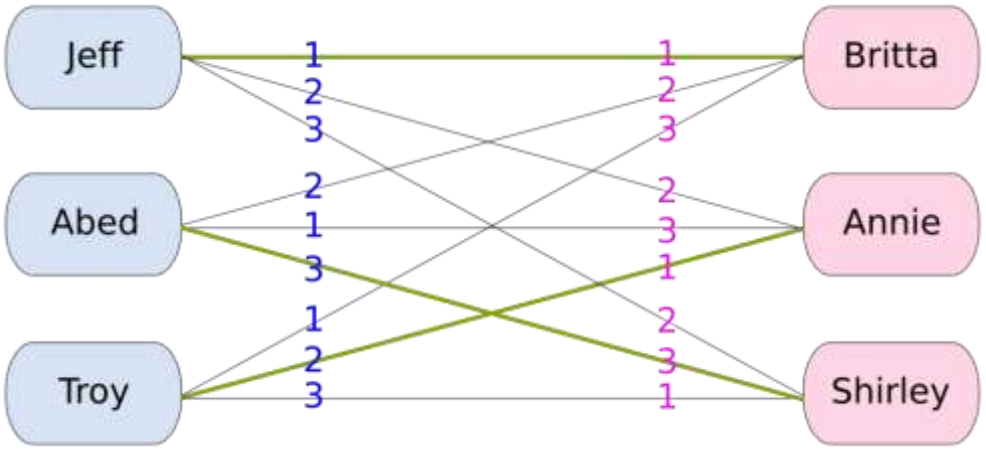
Stable Marriage Problem

- At **least** one **stable** marriage is **always guaranteed** to exist.
- **Boy-optimal** or **Girl-optimal** matchings (**marriages**) can be **found** using the **Gale-Shapely Algorithm**.
- The **Gale-Shapely Algorithm** is **guaranteed** to find **one stable marriage** (**there may be many**) in polynomial time.
- Used for **student-doctor** assignment to **hospitals**, or assigning **applicants** to **universities**, etc.
- **Time complexity** is $O(n^2)$.
- Do **not assume** a **problem** is **NP-Hard**. You **have** to **prove** it. The **SMP fooled** some people.
- Do **not confuse** with the **Stable Roommate Problem** which **can also** be **solved** in **polynomial time** (**Irving**, $O(n^2)$, **1985**).

Formal Definition – Perfect Matching

Suppose we **have** a **set B** of **n boys** and a **set G** of **n girls**.

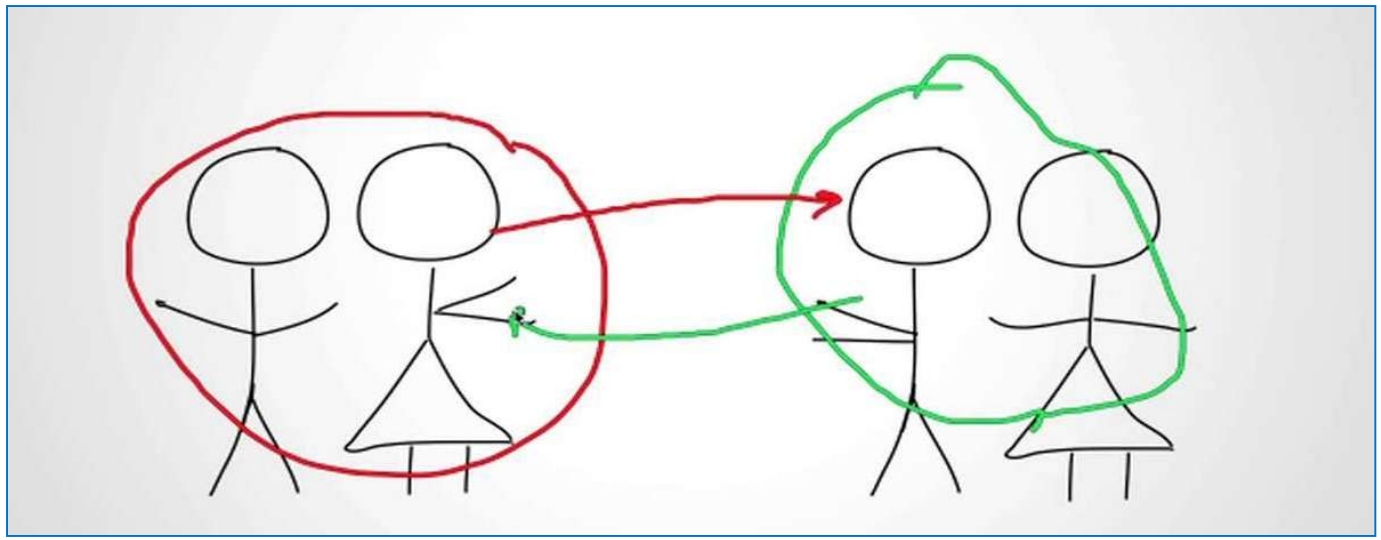
- **Definition**
  A **matching M** is a **set** of **ordered pairs** (**b**, **g**) with **b** ∈ **B** and **g** ∈ **G** **such that**:
  - **Each** boy **b** ∈ **B appears** in at **most one** pair of **M**.
  - **Each** girl **g** ∈ **G appears** in at **most one** pair of **M**.
- **Definition**
  A **matching M** is **perfect** if |**M**| = |**B**| = |**G**| = **n**.



Complete bi-partite graph.

Formal Definition – Unstable Pair

- **Definition**
  Given a **perfect matching *M***, boy ***b*** and girl ***g*** **form** an **unstable pair** if both the **following** are **true**:
  - ***b* prefers *g*** to his **matched girl** in ***M***.
  - ***g* prefers *b*** to her **matched boy** in ***M***.
- **Key Point**
  An **unstable pair *b*** and ***g*** could **each improve** by **being assigned** each **other**.



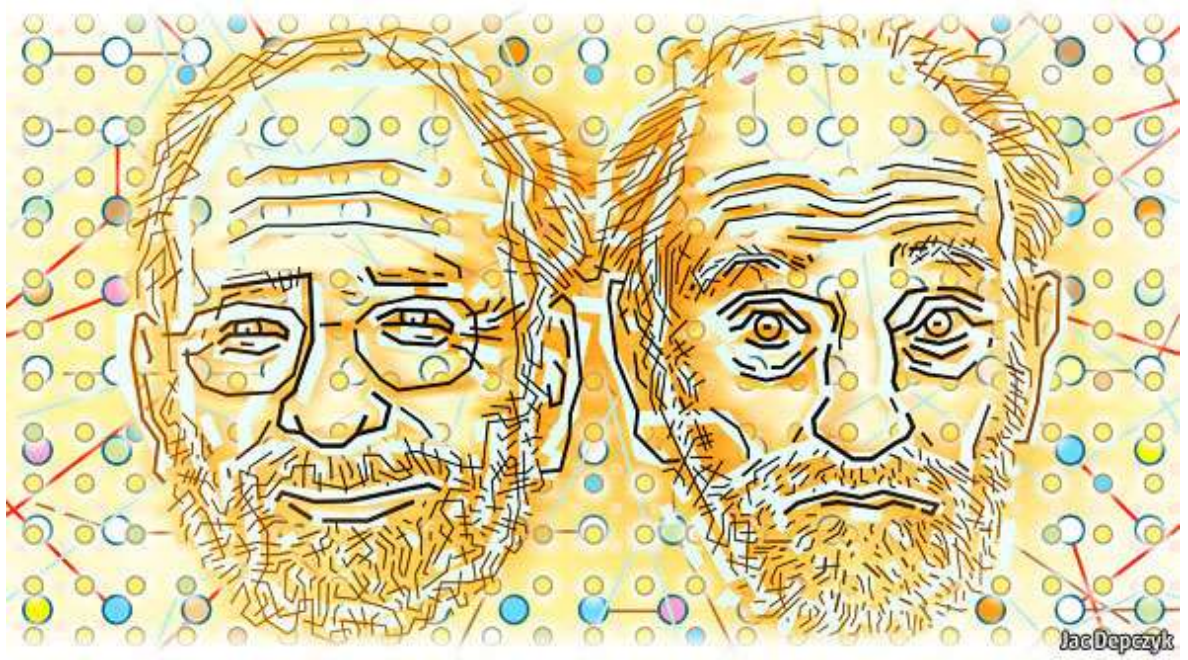
Formal Definition – Stable Matching

- **Definition**
  A **stable matching** is a **perfect matching** with **no unstable pairs**.
- **The Stable Marriage Problem**
  Given the **preference lists** of ***n*** **boys** and ***n*** **girls**, find a **stable matching** (if **one exists**).

- Note that a **stable matching** is **guaranteed** to **exist** and also the **Gale-Shapley algorithm** is **guaranteed** to find a **stable matching**.
- There **may**, **however**, be **more** than one **stable matching**.

The Gale Shapely Algorithm

- In **mathematics**, **economics**, and **computer science**, the **Gale–Shapley algorithm** (also known as the **deferred acceptance algorithm**) is an **algorithm** for **finding** a **solution** to the **stable marriage problem**, **named** after **mathematicians David Gale** and **Lloyd Shapley**.
- **Published** by **Gale** and **Shapely** in **1962**.
- In **2012**, **Shapely** was a **joint recipient** of the **Noble Prize** in **Economics** (with **Alvin Roth**).
- **David Gale passed** away **before** the **award** was **given**.



How it Works

The **method** for **solving** this **problem works something** like this:

- **Initialize** each **boy** or **girl** to be '**free**'
- As **long** as **there** is a **free boy*** do **following**:
- The **first free boy** (in some **order**) will **propose** to the **first girl** on his **list**.
- The **girl**, having **no better offer** at this point, will **accept**.
- The **second boy** will then **propose** to his **first** choice, and **so on**.
- **Eventually** it may **happen** that a **boy proposes** to a girl **who already** has a **partner**.

* Boy optimal pairings

How it Works

- She **will compare** the **new offer** to her **current partner** and will **accept** whoever is **higher** on her **list**.
- The **boy** she **rejects** will **then** go **back** to his **list** and **propose** to his **second** choice, **third** choice, and so **forth until** he **comes** to a **girl** who **accepts** his **offer**.
- If this **girl already** had a **partner**, her old **partner** gets **rejected** and he, in **turn**, starts **proposing** to **girls** further **down** his **list**.
- Eventually **everything** gets **sorted out**.

The Gale-Shapely Algorithm

Algorithm:

- Each **person** starts with **no people** '***cancelled***' from **his** or **her** list (of **preferences**).
- **People** will be **cancelled** from **lists** as the **algorithm** progresses.
- For **each boy *m***, call **propose**(***m***), as **defined** on the **next slide**.
- The **boys** go **first** – more **about** this **later**.

The Gale-Shapely Algorithm

propose(***b***)*:*

- Let ***G*** be the ***first*** uncancelled girl on boy ***b's*** preference list.
- Do: **refuse**(***G,b***), as defined below.

refuse(***G,b***)*:*

- Let ***b'*** be ***G*** **'s** current partner (**if any**).
- **If** ***G*** prefers ***b'*** to ***b***, then she rejects ***b***, in which case:
  cancel ***b*** off ***G*** 's list and ***G*** off ***b***'s list;
  call **propose**(***b***) (**now *b* must now propose to someone else.**)
- **Otherwise**, ***G*** accepts ***b*** as her new partner, in which case:
  cancel ***b'*** off ***G*** 's list and ***G*** off ***b'*** 's list;
  do: **propose**(***b'***). (**since *n'* must now propose to someone else.**)

The Stable Marriage Problem – Stability

- How do we know the final arrangement will be stable?
- Suppose there exists a boy Y and girl X such that they prefer each other over their own spouses. Since Y prefers X over his own spouse, then he must have proposed to her before proposing to his own spouse. So there are two possibilities of what happened at the time of that proposal.
- Then either:
  a) X rejected Y, which means that she was already engaged to someone she preferred over Y (she had scratched Y off her list). Therefore, she must either be married to that person or someone else who was on her list at the time of the proposal. Therefore, she could not prefer Y to her own spouse, contradicting our assumption.
  b) X accepted Y, but dumped him later. Since she would have dumped him only in order to become engaged to someone higher ranked on her list, she must not prefer Y over her own spouse, again contradicting the assumption.

The Stable Marriage Problem – Stability

- In **both** cases, the **assumption** is **contradicted**. So, there **cannot** exist a **boy Y** and **girl X** such that they **prefer each other** over their **own spouses**.

  ➔ ***The arrangement must therefore be stable***.

- How **do** we **know everyone** gets **paired** off?
  - Since the number of **boys** and **girls** are equal, the only **way** for a **boy** to **remain eligible** at the end is for **there** to be an **unengaged** girl who **rejects** him. However, the only **rejections** come **only** from **girls** who are **engaged**.

Lemma 1 – The Improvement Lemma

Improvement Lemma: If a **girl** is **engaged** to a **boy**, then **she** will **always** be **engaged** (or **married**) to **someone** at **least** as **good**.

This is **because** she **would** only **let go** of **him** in **order** to **accept** the **proposal** of someone **better** (**higher** up her **list**).

Lemma 2 – Boys Always Get Accepted

Lemma: No **boy** can be **rejected** by **all** the **girls**.

Proof is by **contradiction**.

Suppose boy **b** is **rejected** by **all** the **girls**. At that point:

- Each **girl** must have a **suitor other** than **b.**
  (By **Improvement Lemma**, **once** a **girl** has a **suitor** she will **always** have at **least one**).
- The **n** girls have **n** suitors, **b** not **among** them. Thus, **there are** at least **n+1** boys.

Corollary: Each **girl** will **marry** her **absolute favorite** of the **boys** who **visit her** during the **GS Algorithm**.

Unstable Couples – An Example

Boys' Preference List

Boy	1st	2nd	3rd
Xavier	A	B	C
Yancey	B	A	C
Zeus	A	B	C

Girls' Preference List

Girl	1st	2nd	3rd
Amy	Y	X	Z
Bertha	X	Y	Z
Clare	X	Y	Z

The lavender assignment is a perfect matching.
Are there any unstable pairs?

Yes. Bertha and Xavier form an unstable pair.
They would prefer each other to current partners.

Stable Matchings – Examples

Boys' Preference List

Boy	1st	2nd	3rd
Xavier	A	B	C
Yancey	B	A	C
Zeus	A	B	C

Girls' Preference List

Girl	1st	2nd	3rd
Amy	Y	X	Z
Bertha	X	Y	Z
Clare	X	Y	Z

Boys' Preference List

Boy	1st	2nd	3rd
Xavier	A	B	C
Yancey	B	A	C
Zeus	A	B	C

Girls' Preference List

Girl	1st	2nd	3rd
Amy	Y	X	Z
Bertha	X	Y	Z
Clare	X	Y	Z

The Gale-Shapely Algorithm - Example

- We have a **list** of **4 boys** and **4 girls** and their **preference list** as **below**:
- **Zeppi**: Mary Anita Eileen Benna
- **Xandru**: Anita Eileen Mary Benna
- **Kalanc**: Eileen Mary Benna Anita
- **Dru**: Mary Anita Eileen Benna

- **Anita**: Zeppi Kalanc Dru Xandru
- **Benna**: Dru Zeppi Xandru Kalanc
- **Eileen**: Kalanc Xandru Dru Zeppi
- **Mary**: Xandru Dru Kalanc Zeppi

The Gale-Shapely Algorithm - Example

- **Zeppi**: ~~Mary~~ Anita Eileen Benna
- **Xandru**: ~~Anita~~ ~~Eileen~~ Mary Benna
- **Kalanc**: Eileen Mary Benna Anita
- **Dru**: ~~Mary~~ ~~Anita~~ ~~Eileen~~ Benna

- **Anita**: Zeppi ~~Kalanc Dru Xandru~~
- **Benna**: Dru Zeppi Xandru Kalanc
- **Eileen**: Kalanc ~~Xandru Dru Zeppi~~
- **Mary**: Xandru ~~Dru Kalanc Zeppi~~

Zeppi → ~~Mary~~ Anita

Xandru → ~~Anita~~ ~~Eileen~~ Mary

Kalanc → Eileen

Dru → ~~Mary~~ ~~Anita~~ ~~Eileen~~ Benna

Stable Marriages!

Zeppi ♥ Anita
Xandru ♥ Mary
Kalanc ♥ Eileen
Dru ♥ Benna

The Gale-Shapely Algorithm - Example

- **Who** gets a **better deal** in this **algorithm**, the **boys** or the **girls**?
- It **turns** out that **since** the **unengaged** girls in this **algorithm accept** any **proposal**, the **boys** end up **happier** on **average**!
- For **example**, suppose there are **5 boys** and **5 girls**, and suppose **every boys** has a **different** first choice. Then they'll all **get** their **first choice**, **regardless** of the **preferences** of the **girls**!
- You can, of **course**, let the **girls** go **first**.
- This problem is **typically** applied to **binary relationships** somewhat more **practical** than **marriage**, such as **dormitory room assignment**, **university admissions**, **matching graduating medical students to hospitals**, etc.

The Gale-Shapely Algorithm - Pseudocode

```
function stableMatching {
        Initialize all m ∈ M and w ∈ W to free
        while ∃ free man m who still has a woman w to propose to {
          w = first woman on m's list to whom m has not yet proposed
          if w is free
            (m, w) become engaged
          else some pair (m', w) already exists
            if w prefers m to m'
              m' becomes free
              (m, w) become engaged
            else
              (m', w) remain engaged
        }
}
```

The Gale-Shapely Algorithm - Notes

- **Note** that the **master** list of **k boys** (and **k girls**) has **k^2 entries** since **each** of the **k boys ranks** each of the **k girls**. The **space complexity** is therefore O(**k^2**) **where k** is the **number** of **boys** (or **girls**).
- At **every step** where at **least one boy** gets a '**No**', at **least one girl** gets **crossed off** the boys' **master** list.
- Since the **number** of **steps** is **bounded** by the **original size** of the **master** list, then the **number** of **steps** is **bounded** by: **n**.(**n-1**) = **n^2** – **n which** is O(**n^2**) – a **quadratic polynomial**.
- In the **early 1970s**, **McVitie** and **Wilson**, however, **formulated** a **faster**, **straight-forward**, **backtracking**, **scheme** for the **stable marriage problem**. **Look** it **up**.

Google

Stable Marriage Problem Questions

- **Suppose** the a **boy** and a **girl rank** each **first** on their **respective** lists. Are they **guaranteed** that **they** will **get each** other?
- Is it **possible** for a **boy** or a **girl** to remain **unassigned**?
- Is the **Gale-Shapley** algorithm a **greedy** algorithm?
- When a **boy proposes** to a **free** (**available**) **girl** will she **accept** his **proposal**?
- **Suppose** all the boys **propose** to a **different** girl. How many **iterations** are **required** for the **algorithm** to **converge**?
- How do you get **worst** case **complexity** for the **Gale-Shapley algorithm**?
- In **general**, is there **usually** only **one stable marriage**?
- Does the **order** that the **boys** are **visited affect** the **matching** found?

Stable Marriage Problem Applications

- Matching **kidney donors** to **patients** awaiting a **transplant**.
- Match college **applicants** to **Universities**.
- Assigning of **user queries** to **web servers**.
- Matching **peers** in a **P2P** network.
- Resource **allocation** in **5G** networks.
- Assigning **seamen** to **navy** ships.
- Assigning **students** to **public** schools.
- **Algorithmic Game Theory**.

Imaged by Heritage Auctions, HA.com

The **Gale-Shapley algorithm** did not win the **Nobel Prize** because of its complexity but **probably** because it was so **ubiquitous** and so **wide** in **scope**.

Prologue

- Please **send** me **feedback** and/or **errata** to john.abela@um.edu.mt
- **Feedback helps** me **improve** the **slide decks** for **future** students.
- **Online lectures** may be **recorded**.
- **Lecture slides** are also to the **VLE**.

L-Università ta' Malta

Data Structures & Algorithms 2

John Abela
Department of CIS
Faculty of ICT
University of Malta
john.abela@um.edu.mt
+ 365 79367936
Room 1A/27
FICT Building

Academic Year 2021-22

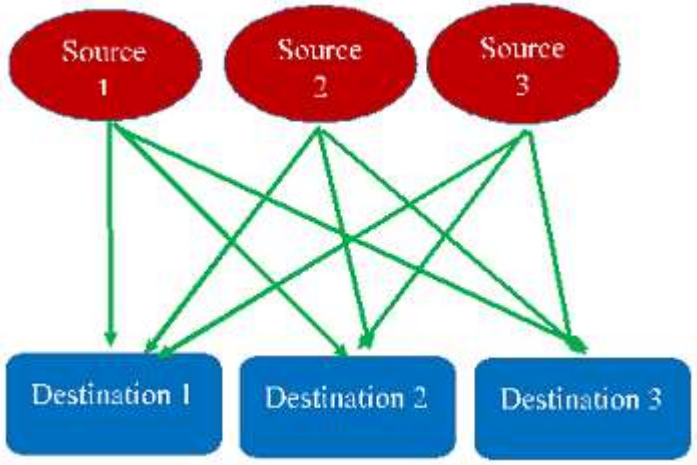
Module 5

The

Assignment Problem

&

The Hungarian Algorithm



The Hungarian Algorithm:
1 Start with an initial feasible labeling to get an equality graph G_l
and find a matching M in G
2 **while** M is not a perfect matching
3 Find an augment path in G
which can increase the size of the matching
4 If no above augment exists,
update labeling and update the equality graph G to become G'_l

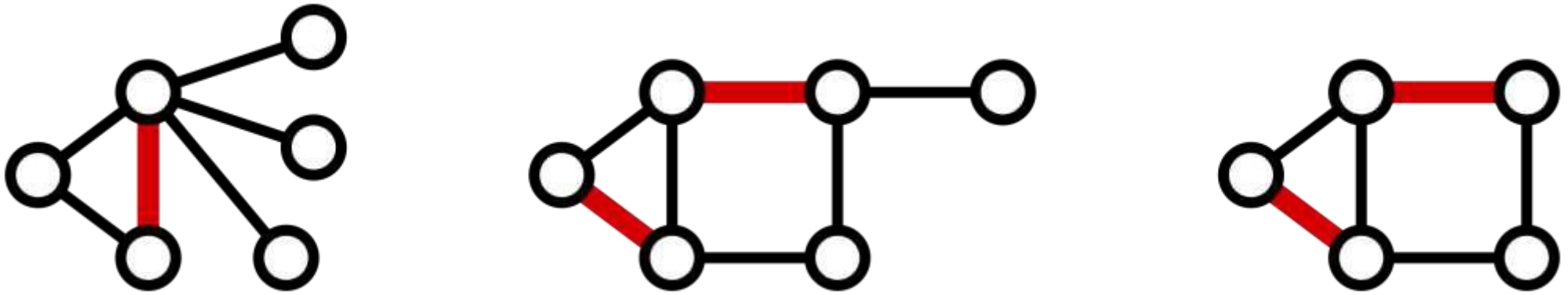
Faculty of ICT

Department of Computer Information Systems

Some Useful Definitions

WIKIPEDIA

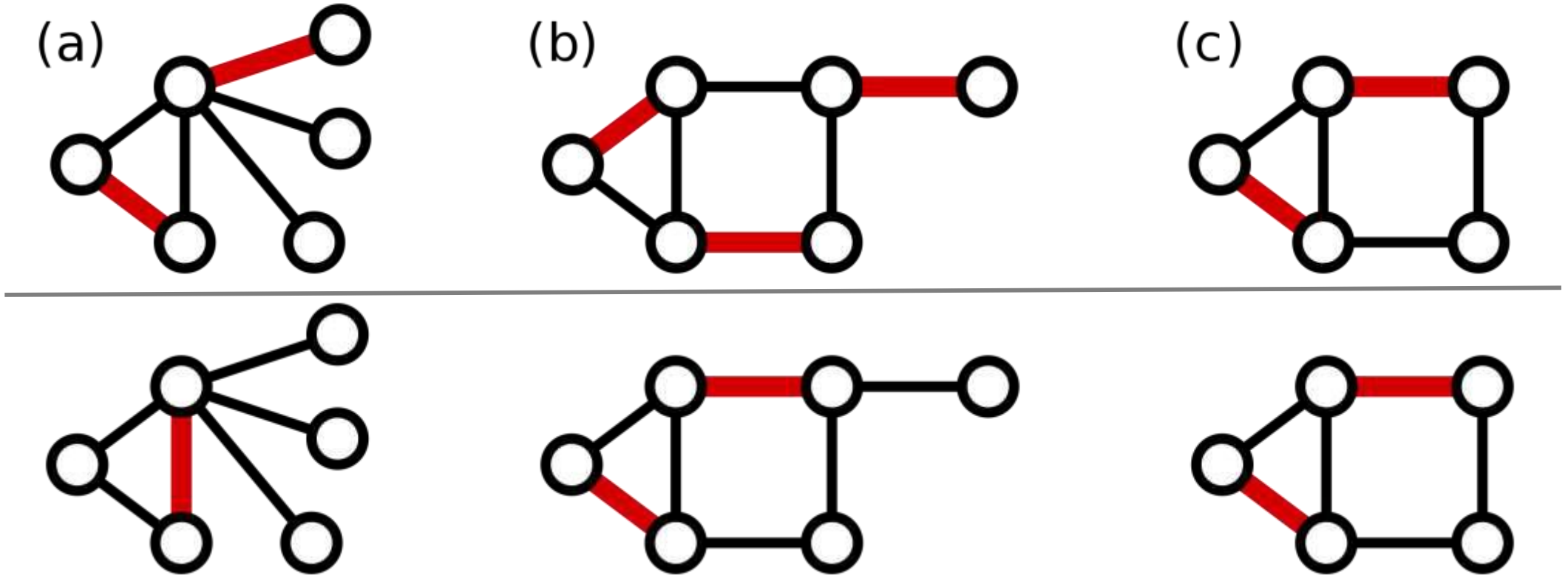
- Given a **graph G** = (**V**,**E**), a **matching M** in **G** is a set of **pairwise non-adjacent** edges, **none** of **which** are **loops**.
- That is, no **two edges share** a **common vertex**.
- A **vertex** is **matched** (or **saturated**) if it is an **endpoint** of **one** of the **edges** in the **matching**. Otherwise the **vertex** is **unmatched**.
- A **maximal matching** is a matching **M** of a **graph G** with the **property** that if **any edge not** in **M** is **added** to **M**, it is no **longer** a **matching**, that is, **M** is **maximal** if it is not a **subset** of any other **matching** in graph **G**.
- In **other words**, a **matching M** of a **graph G** is **maximal** if **every edge** in **G** has a **non-empty intersection** with at **least one** edge in **M**. The figure **below** shows **examples** of **maximal matchings** (**red**) in **three** graphs.



More Useful Definitions

WIKIPEDIA

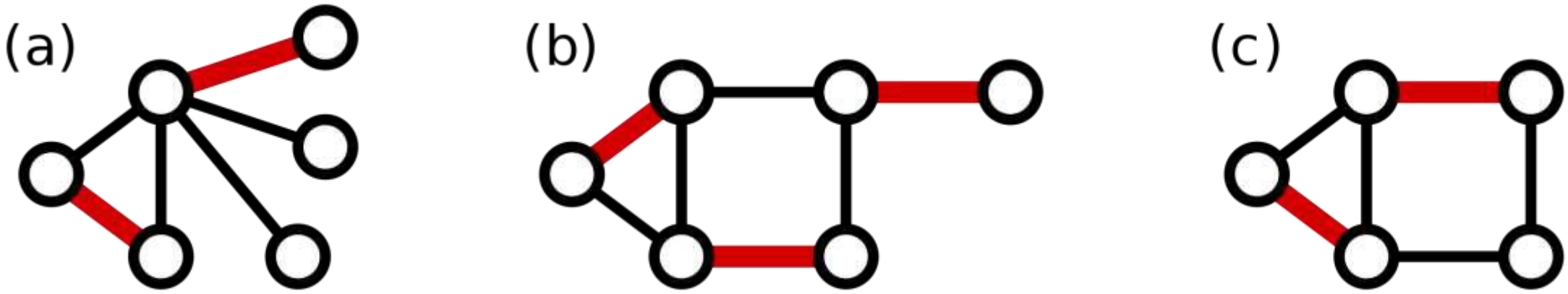
- A **maximum matching** (also **known** as **maximum-cardinality** matching) is a **matching** that **contains** the **largest possible** number of **edges**. There **may** be many **maximum** matchings. The **matching number** of a **graph** is the **size** of a **maximum** matching. **Note** that every **maximum matching** is **maximal**, but not **every maximal matching** is a **maximum** matching. The **figure** below shows **examples** of **maximum matchings** in the **same three** graphs.



More Useful Definitions

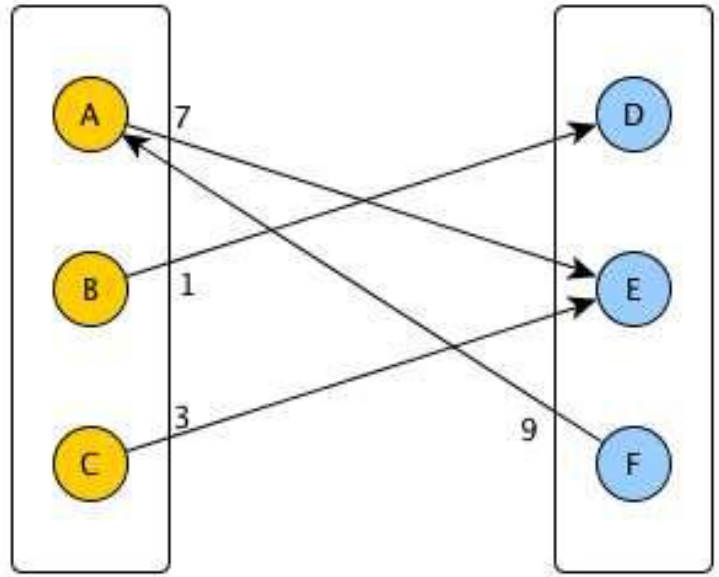
WIKIPEDIA

- A **perfect matching** is a **matching** which **matches** all **vertices** of the **graph**. That is, every **vertex** of the graph is **incident** to exactly one **edge** of the **matching**. Every **perfect matching** is **maximum** and hence **maximal**. In some **literature**, the term **complete matching** is used. In the **figure** below, only **graph** (**b**) shows a **perfect matching**.



The Assignment Problem

- The **assignment problem** is a fundamental **combinatorial optimization** problem. It consists of **finding**, in a **weighted bipartite graph**, a **matching** in which the **sum** of **weights** of the **edges** is as **large** as **possible**. A **common variant** consists of **finding** a **minimum-weight perfect matching**.
- It is a **specialization** of the **maximum weight matching** problem for **bipartite** graphs.
- In its **most general form**, the **problem** is as **follows**: The problem **instance** has a **number** of **agents** and a number of **tasks**. Any **agent** can be **assigned** to perform any **task**, incurring some **cost** that may **vary** depending on the **agent-task** assignment. It is **required** to **perform** all **tasks** by **assigning** exactly **one agent** to each **task** and exactly **one task** to each **agent** in such a **way** that the **total cost** of the **assignment** is **minimized**.



The Assignment Problem - Example

- **Three** workmen **A**, **B**, and **C** are each to be **assigned** one of three **tasks**. The **cost matrix** is **shown** below **right**.
- Note that the **number** of **workers** is **equal** to the **number** of tasks.
- Each worker has a **different cost** for each **task** depending on the **worker's skills** and the **time required** to **complete** the **task**.
- To which **worker** should **each task** be **assigned**?
- Note that **there** are **O(*n!*)** ways of **assigning tasks** to **workers**.
- Do we **have** to use **brute-force**?

	Task1	Task2	Task3
A	53	96	37
B	47	87	41
C	60	92	36

Using Brute Force

	Bldg1	Bldg2	Bldg3
A	53	96	37
B	47	87	41
C	60	92	36

Total Cost = 53 + 87+ 36 = **176**

	Bldg1	Bldg2	Bldg3
A	53	96	37
B	47	87	41
C	60	92	36

Total Cost = 53 + 92+ 41 = **186**

	Bldg1	Bldg2	Bldg3
A	53	96	37
B	47	87	41
C	60	92	36

Total Cost = 47 + 96 + 36 = **179**

	Bldg1	Bldg2	Bldg3
A	53	96	37
B	47	87	41
C	60	92	36

Total Cost = 47 + 92 + 37 = **176**

	Bldg1	Bldg2	Bldg3
A	53	96	37
B	47	87	41
C	60	92	36

Total Cost = 47 + 96 + 41 = **197**

	Bldg1	Bldg2	Bldg3
A	53	96	37
B	47	87	41
C	60	92	36

Total Cost = 60 + 87 + 37 = **184**

Using Brute Force

- Brute force gives us **2 minimum cost assignments**.
- How does the **time-to-solution vary** with problem size?
- It is not **very clever** to use a **factorial time** brute force algorithm.
- Neither is it **very clever** to assume that **since** the **brute force** algorithm is **factorial** then the **problem** must be **NP-Hard**.
- **Sorting** can be **implemented** as **brute-force**. This does **not mean** that **sorting** is **NP-Hard**!

	1	2	3
A	**53**	96	37
B	47	**87**	41
C	60	92	**36**

	1	2	3
A	**53**	96	37
B	47	87	**41**
C	60	**92**	36

	1	2	3
A	53	**96**	37
B	**47**	87	41
C	60	92	**36**

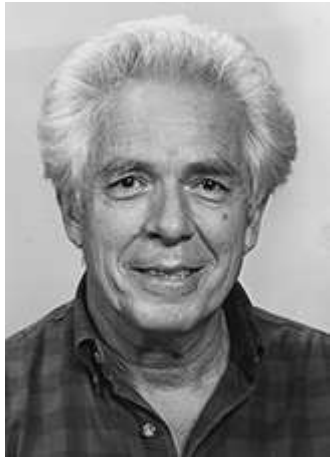
	1	2	3
A	53	96	**37**
B	**47**	87	41
C	60	**92**	36

	1	2	3
A	53	**96**	37
B	47	87	**41**
C	**60**	92	36

	1	2	3
A	53	96	**37**
B	47	**87**	41
C	**60**	92	36

The Hungarian Algorithm

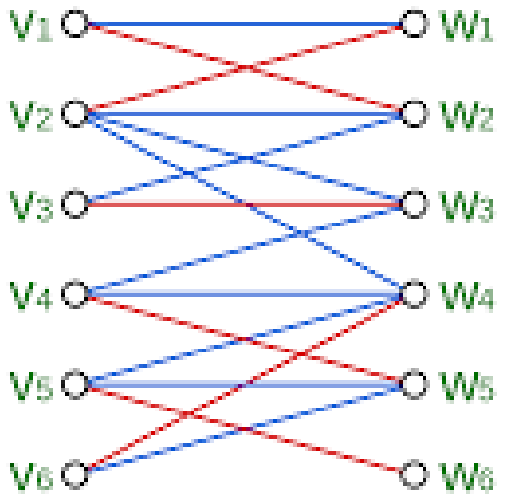
- The **Hungarian algorithm** is a **combinatorial optimization** algorithm **that solves** the **assignment problem** in **polynomial** time.
- It was **developed** and **published** in **1955** by **Harold Kuhn**, who **modestly gave** it the **name** '**Hungarian method**' because the **algorithm** was largely **based** on the **earlier works** of two **Hungarian** mathematicians - **Dénes Kőnig** and **Jenő Egerváry**.
- **James Munkres** reviewed the **algorithm** in **1957** and **observed** that it is **low order polynomial**. The **algorithm** has **also** been **known** also as the **Kuhn–Munkres algorithm**.
- The **time complexity** of the **original** algorithm was ***O*(n^4)**, however **Edmonds** and **Karp**, and independently **Tomizawa**, **noticed** that it **can** be **modified** to **achieve** an ***O*(n^3) running time**.
- One of the **most popular *O*(n^3) variants** is the **Jonker-Volgenant algorithm**.



Problem Specification

- **Suppose** we have ***n*** **resources** to **which** we **want** to **assign** to ***n*** **tasks** on a **one-to-one** basis. Suppose **also** that we **know** the **cost** of **assigning** a given **resource** to a **given** task.
- We **represent** the **costs** in an ***n*** X ***n*** **cost matrix**.
- We **wish** to **find** an **optimal assignment** – one which **minimizes** or **maximizes** the **total cost**.
- Can **also** be **equivalently** posed as a **weighted bipartite graph** problem.

	w_1	w_2	w_3	w_4	w_5	w_6
v_1	1	1	0	0	0	0
v_2	1	1	1	1	0	0
v_3	0	1	1	0	0	0
v_4	0	0	1	1	1	0
v_5	0	0	0	1	1	1
v_6	0	0	0	1	1	0



Cost Matrix

- Let $c_{i,j}$ be the **cost** of **assigning** the *i*th **resource** to the *j*th **task**. We **define** the **cost matrix** to be the $n \times n$ **matrix**:

$$C = \begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,n} \\ \vdots & \vdots & & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,n} \end{bmatrix}$$

- An **assignment** is a **set** of **n entry positions** in the **cost matrix**, no **two** of which **lie** in the same **row** or **column**. The **sum** of the **n entries** of an **assignment** is its **cost**. An **assignment** with the **smallest** or **largest possible cost** is called an **optimal assignment**.

The Hungarian Algorithm

- The **Hungarian Algorithm** is an **algorithm** which **finds** an **optimal assignment** for a given **cost** matrix.
- We **illustrate** the **algorithm** with **two examples**.

Example 1

You work as a **sales manager** for a **toy manufacturer**, and you **currently** have **three salespeople** on the **road** meeting **buyers**. Your **salespeople** are in **Austin**, TX; **Boston**, MA; and **Chicago**, IL. You **want** them to **fly** to three **other cities**: **Denver**, CO; **Edmonton**, Alberta; and **Fargo**, ND. The **table below** shows the **cost** of **airplane tickets** in **dollars** between **these cities**.

From \ To	Denver	Edmonton	Fargo
Austin	250	400	350
Boston	400	600	350
Chicago	200	400	250

Example 1 – Minimizing Airfares

- **Where** should you **send each** of your **salespeople** in **order** to **minimize** airfares?
- We can **represent** the **table** as a **cost matrix**.

From \ To	Denver	Edmonton	Fargo
Austin	250	400	350
Boston	400	600	350
Chicago	200	400	250

$$\begin{bmatrix} 250 & 400 & 350 \\ 400 & 600 & 350 \\ 200 & 400 & 250 \end{bmatrix}$$

Example 1 – Minimizing Airfares

- Let us **look** at one **possible assignment**.
- The **total cost** of this **assignment** is $**250** + $**600** + $**250** = $**1100**.
- Note that **small instances** can be **solved** by **inspection**.

$$\begin{bmatrix} \boxed{250} & 400 & 350 \\ 400 & \boxed{600} & 350 \\ 200 & 400 & \boxed{250} \end{bmatrix}$$

Example 1 – Minimizing Airfares

- Let us **look** at **another possible** assignment.
- The **total cost** of this assignment is $**250** + $**350** + $**400** = $**1000**.
- If your **pick** this **assignment** then your **salespeople** should **travel** from **Austin** to **Edmonton**, **Boston** to **Fargo**, and **Chicago** to **Denver**.

$$\begin{bmatrix} \boxed{250} & 400 & 350 \\ 400 & 600 & \boxed{350} \\ 200 & \boxed{400} & 250 \end{bmatrix}$$

Example 1 – Minimizing Airfares

- **Trial** and **error** works **well enough** for this problem, but **suppose** you had **twenty salespeople** flying to **twenty cities**? How many **trials** would this **take**?
- There are ***n!*** ways of **assigning *n* resources** to ***n* tasks**.
- That **means** that as ***n*** gets **large**, we have **too many** trials to **consider**.
- Using **brute force** is not **very clever**.
- Also, do not **automatically assume** that the **problem** is **NP-Hard**.
- The **lesson** here is that **simply because** the **search space** of an **optimization problem** is **factorial** in the **size** of the **problem** does **not mean** that the **problem** is **NP-Hard**.

Example 1 – Minimizing Airfares

Important Theorem

If a **number** is **added** to or **subtracted** from **all** of the **entries** of any **one row** or **column** of a cost matrix, then an **optimal assignment** for the **resulting** cost matrix is also an **optimal assignment** for the **original** cost matrix.

This **theorem** is **indeed** very **useful**!

The Hungarian Algorithm

Step 1

Subtract the **smallest** entry in each **row** from **all** the **entries** of its **row**.

Step 2

Subtract the **smallest entry** in each **column** from **all** the **entries** of its **column**.

Step 3

Draw **lines** through **appropriate rows** and **columns** so that all the **zero** entries of the **cost matrix** are **covered** and the **minimum number** of such **lines** is used.

Step 4 - Test for Optimality

(i) If the **minimum number** of **covering lines** is **n**, an **optimal assignment** of zeros is **possible** and we are **finished**.

(ii) If the **minimum number** of **covering** lines is **less** than **n**, an **optimal assignment** of zeros is **not yet possible**. In that case, **proceed** to **Step 5**.

Step 5.

Determine the **smallest entry** not **covered** by any **line**. **Subtract** this **entry** from each **uncovered row**, and then add it to each **covered column**. Return to **Step 3**.

Example 1 – Minimizing Airfares

Step 1 - Subtract **250** from **Row 1**, **350** from **Row 2**, and **200** from **Row 3**.

$$\begin{bmatrix} 250 & 400 & 350 \\ 400 & 600 & 350 \\ 200 & 400 & 250 \end{bmatrix} \sim \begin{bmatrix} 0 & 150 & 100 \\ 50 & 250 & 0 \\ 0 & 200 & 50 \end{bmatrix}$$

Step 2 - Subtract **0** from **Column 1**, **150** from **Column 2**, and **0** from **Column 3**.

$$\begin{bmatrix} 0 & 150 & 100 \\ 50 & 250 & 0 \\ 0 & 200 & 50 \end{bmatrix} \sim \begin{bmatrix} 0 & 0 & 100 \\ 50 & 100 & 0 \\ 0 & 50 & 50 \end{bmatrix}$$

Example 1 – Minimizing Airfares

Step 3 - Cover **all** the **zeros** of the **matrix** with the **minimum number** of **horizontal** or **vertical** lines.

$$\begin{bmatrix} \cancel{0} & \cancel{0} & \cancel{100} \\ \cancel{50} & \cancel{100} & \cancel{0} \\ \cancel{0} & 50 & 50 \end{bmatrix}$$

Step 4 - Since the **minimal number** of **lines** is 3, an **optimal assignment** of **zeros** is **possible** and we are **finished**.

Example 1 – Minimizing Airfares

Since the **total cost** for this **assignment** is **0**, it must be an **optimal assignment**.

$$\begin{bmatrix} 0 & \boxed{0} & 100 \\ 50 & 100 & \boxed{0} \\ \boxed{0} & 50 & 50 \end{bmatrix}$$

Here is the **same assignment** made to the **original cost matrix**.

$$\begin{bmatrix} 250 & \boxed{400} & 350 \\ 400 & 600 & \boxed{350} \\ \boxed{200} & 400 & 250 \end{bmatrix}$$

Example 1 – Minimizing Airfares

- The **first** example was **easy**.
- We got the **minimal number** of **crossing lines** to be **3** in the first **iteration**.
- **Real-world** instances are **not always** so **easy** and **straight-forward**.
- We **may** have to do **more iterations** to get the **number** of **crossing** lines to be ***n***.
- We **illustrate** with **another** example.

Example 2- Moving Bulldozers

Example 2

A **construction company** has **four large bulldozers** located at **four different** garages. The **bulldozers** are to be **moved** to **four different construction** sites. The **distances** in **miles** between the **bulldozers** and the **construction** sites are given below.

Bulldozer \ Site	A	B	C	D
1	90	75	75	80
2	35	85	55	65
3	125	95	90	105
4	45	110	95	115

Example 2- Moving Bulldozers

- How **should** the **bulldozers** be **moved** to the **construction** sites in **order** to **minimize** the **total distance** travelled?

Step **1** - **Subtract 75** from **Row 1**, **35** from **Row 2**, **90** from **Row 3**, and **45** from **Row 4**.

$$\begin{bmatrix} 90 & 75 & 75 & 80 \\ 35 & 85 & 55 & 65 \\ 125 & 95 & 90 & 105 \\ 45 & 110 & 95 & 115 \end{bmatrix} \sim \begin{bmatrix} 15 & 0 & 0 & 5 \\ 0 & 50 & 20 & 30 \\ 35 & 5 & 0 & 15 \\ 0 & 65 & 50 & 70 \end{bmatrix}$$

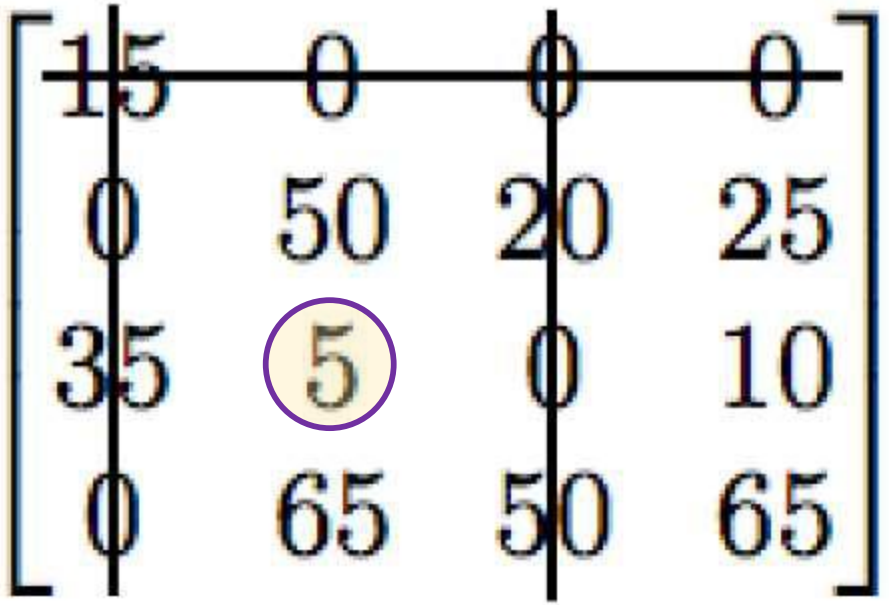
Example 2- Moving Bulldozers

Step 2 - Subtract 0 from Column 1, 0 from Column 2, 0 from Column 3, and 5 from Column 4.

$$\begin{bmatrix} 15 & 0 & 0 & 5 \\ 0 & 50 & 20 & 30 \\ 35 & 5 & 0 & 15 \\ 0 & 65 & 50 & 70 \end{bmatrix} \sim \begin{bmatrix} 15 & 0 & 0 & 0 \\ 0 & 50 & 20 & 25 \\ 35 & 5 & 0 & 10 \\ 0 & 65 & 50 & 65 \end{bmatrix}$$

Example 2- Moving Bulldozers

Step **3** - Cover all the zeros of the **matrix** with the **minimum number** of **horizontal** or **vertical** lines.



Step **4** - Since the **minimal number** of **lines** is **less** than 4, we **have** to **proceed** to **Step 5**.

Example 2- Moving Bulldozers

Step 5 - Note that 5 is the smallest entry not covered by any line. Subtract 5 from each uncovered row.

$$\begin{bmatrix} 15 & 0 & 0 & 0 \\ 0 & 50 & 20 & 25 \\ 35 & 5 & 0 & 10 \\ 0 & 65 & 50 & 65 \end{bmatrix} \sim \begin{bmatrix} 15 & 0 & 0 & 0 \\ -5 & 45 & 15 & 20 \\ 30 & 0 & -5 & 5 \\ -5 & 60 & 45 & 60 \end{bmatrix}$$

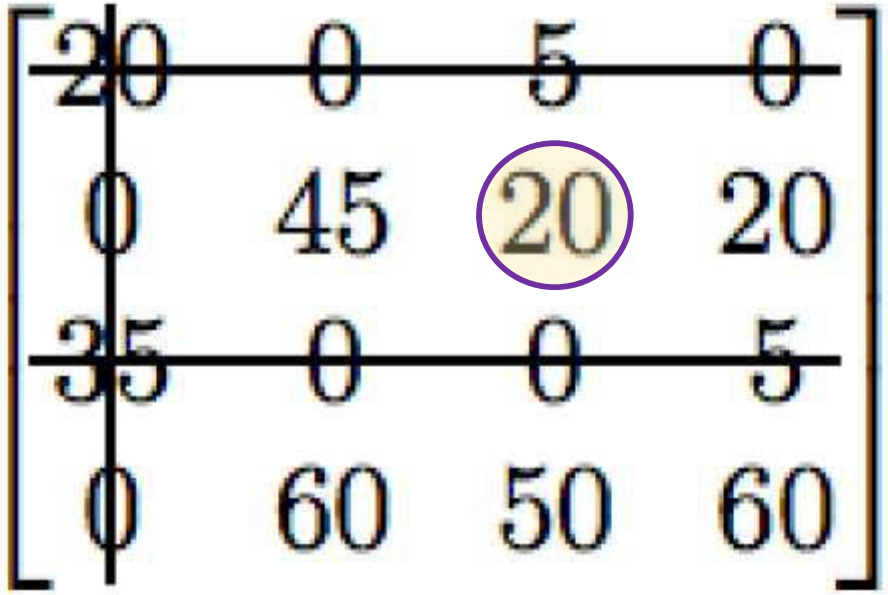
Now add 5 to each covered column.

$$\begin{bmatrix} 15 & 0 & 0 & 0 \\ -5 & 45 & 15 & 20 \\ 30 & 0 & -5 & 5 \\ -5 & 60 & 45 & 60 \end{bmatrix} \sim \begin{bmatrix} 20 & 0 & 5 & 0 \\ 0 & 45 & 20 & 20 \\ 35 & 0 & 0 & 5 \\ 0 & 60 & 50 & 60 \end{bmatrix}$$

Example 2- Moving Bulldozers

Now **return** to **Step 3**.

Step **3** - **Cover** all the **zeros** of the **matrix** with the **minimum number** of **horizontal** or **vertical** lines.



Step **4** – **Again**, **since** the **minimal number** of **lines** is less than **4**, we **have** to **proceed** to **Step 5**.

Example 2- Moving Bulldozers

Step 5 - Note that **20** is the **smallest entry** not **covered** by **any line**. **Subtract 20** from each **uncovered row**.

$$\begin{bmatrix} 20 & 0 & 5 & 0 \\ 0 & 45 & 20 & 20 \\ 35 & 0 & 0 & 5 \\ 0 & 60 & 50 & 60 \end{bmatrix} \sim \begin{bmatrix} 20 & 0 & 5 & 0 \\ -20 & 25 & 0 & 0 \\ 35 & 0 & 0 & 5 \\ -20 & 40 & 30 & 40 \end{bmatrix}$$

Now **add** 20 to each **covered column**.

$$\begin{bmatrix} 20 & 0 & 5 & 0 \\ -20 & 25 & 0 & 0 \\ 35 & 0 & 0 & 5 \\ -20 & 40 & 30 & 40 \end{bmatrix} \sim \begin{bmatrix} 40 & 0 & 5 & 0 \\ 0 & 25 & 0 & 0 \\ 55 & 0 & 0 & 5 \\ 0 & 40 & 30 & 40 \end{bmatrix}$$

Example 2- Moving Bulldozers

Now **return** to **Step 3**.

Step **3** - **Cover** all the **zeros** of the **matrix** with the **minimum** number of **horizontal** or vertical lines.

$$\begin{bmatrix} \cancel{40} & \cancel{0} & \cancel{5} & \cancel{0} \\ \cancel{0} & \cancel{25} & \cancel{0} & \cancel{0} \\ \cancel{55} & \cancel{0} & \cancel{0} & \cancel{5} \\ \cancel{0} & \cancel{40} & \cancel{30} & \cancel{40} \end{bmatrix}$$

Step **4** - Since the **minimal number** of **lines** is **4**, an **optimal assignment** of **zeros** is **possible** and we are **finished**.

Example 2- Moving Bulldozers

Since the total cost for this assignment is 0, it must be an **optimal assignment**.

$$\begin{bmatrix} 40 & 0 & 5 & \boxed{0} \\ 0 & 25 & \boxed{0} & 0 \\ 55 & \boxed{0} & 0 & 5 \\ \boxed{0} & 40 & 30 & 40 \end{bmatrix}$$

Example 2- Moving Bulldozers

Here is the **same assignment** applied to the **original** cost matrix.

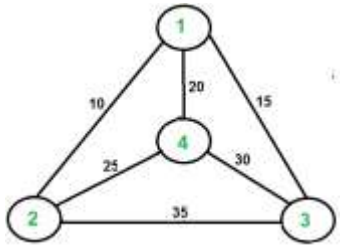
$$\begin{bmatrix} 90 & 75 & 75 & \boxed{80} \\ 35 & 85 & \boxed{55} & 65 \\ 125 & \boxed{95} & 90 & 105 \\ \boxed{45} & 110 & 95 & 115 \end{bmatrix}$$

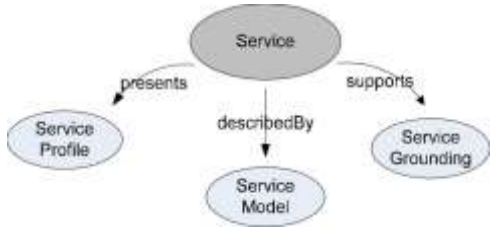
So we **should send** Bulldozer **1** to Site **D**, Bulldozer **2** to Site **C**, Bulldozer **3** to Site **B**, and Bulldozer **4** to Site **A**.

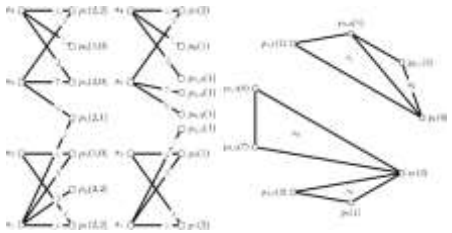
Bulldozers are now happy!

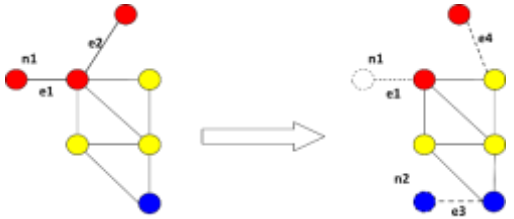
Applications of the Hungarian Method

- **Travelling Sales Problem** (**approximation**).
- **Employee** Assignment.
- **Geospatial** Analytics.
- **Baseball Team** Selection.
- **Semantic Web Services** Optimization.
- **Examination Timetabling** (**approximation**).
- **Student-Project** Allocation.
- **Graph Edit Distance** (**approximation**).
- **Transportation**.
- **Manufacturing Logistics**.





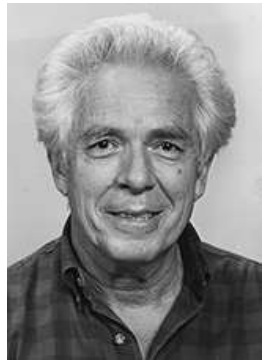




Some Notes

- The **Hungarian method** is a **combinatorial optimization algorithm** that **solves** the **assignment problem** (**optimally**) in **polynomial** time.
- It was **developed** and **published** in **1955** by **Harold Kuhn**.
- Kuhn **gave** it the **name** '**Hungarian method**' because the **algorithm** was **largely** based on the **earlier works** of two **Hungarian mathematicians** - **Dénes Kőnig** and **Jenő Egerváry**.
- Today it is **often called** the **Munkres** algorithm or the **Kuhn-Munkres** algorithm after **James Munkres** who **reviewed** the **algorithm** in **1957** and **observed** that it is **strongly polynomial**. **Look** this **up** on **Google**.


James Munkres


Harold Kuhn


Dénes Kőnig


Jenő Egerváry

Additional Notes

- The **number** of **agents** and the **number** of **tasks** must be **equal** - ***n***.
- The **algorithm only** works if the **cost matrix** is a **square *n*** X ***n* matrix**.
- If the **number** of **tasks** is, for **instance**, ***n***-1 then we just **add** a **column** with **entries equal** to the **largest value** in the **matrix**.
- If the **number** of **agents** is, for **instance**, ***n***-1 then we just **add** a **row** with **entries equal** to the **largest value** in the **matrix**. **Proof** in the **literature**.
- You **can also** add a **row** with all **entries equal** to 0. This **works too** but ***may* require** more **iterations**. **Proof** in the **literature**.

Prologue

- Please **send** me **feedback** and/or **errata** to john.abela@um.edu.mt
- **Feedback helps** me **improve** the **slide decks** for **future** students.
- **Online lectures** are **recorded** and will be **available** on the **VLE**.
- **Lecture slides** are also **uploaded** to the **VLE**.

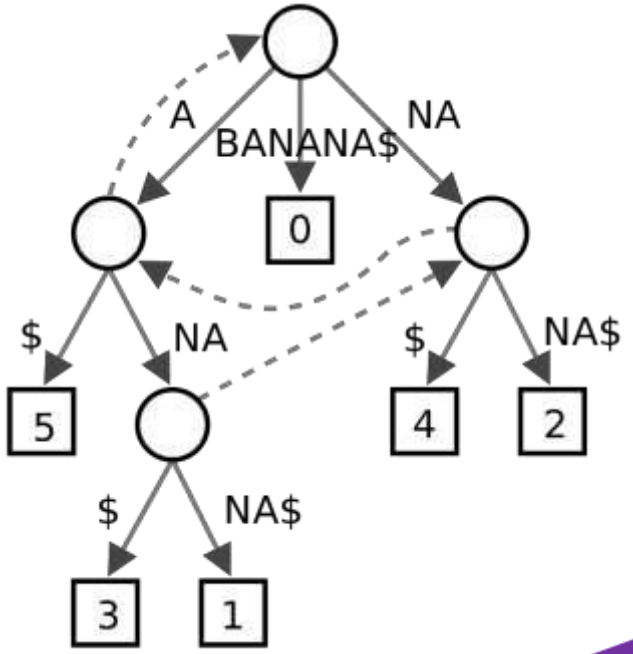
L-Università ta' Malta

Data Structures And Algorithms 2

John Abela
Department of CIS
Faculty of ICT
University of Malta

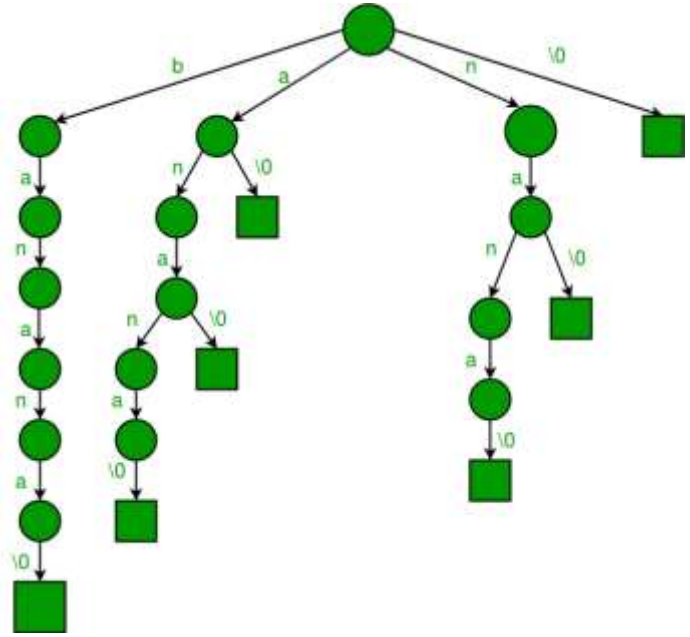
john.abela@um.edu.mt

+ 365 79367936
Room 1A/27
FICT Building



Module 5

Suffix Tries & Trees

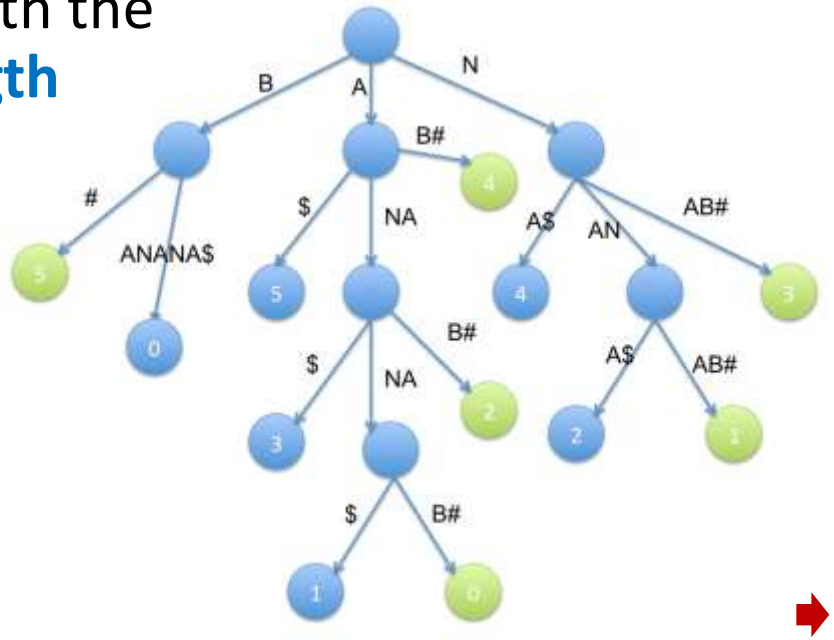


Faculty of ICT

Department of Computer Information Systems

Background

- **Suffix tries** and **trees** are **data structures** that are **used** to **store**, **index**, and **process strings** (**words**, **biological sequences**, **codes**, etc.)
- They are **very important** data structures in **Natural Language Processing** (**NLP**), **text processing**, **cryptography**, **bioinformatics**, etc.
- **Suffix tries** and **trees** are **extremely versatile**, **ubiquitous**, and **powerful** tools in string processing.
- Have you **ever considered** why **search engines** are able to **index billions** of **documents** and still **return** query **results** in **milliseconds**?
- Suffix **tries** and **trees** allow the **indexing** of **documents** with the **query time complexity** bounded (from **above**) in the **length** of the **longest** string.
- This is **irrespective** of how **many documents** are **indexed**.



Some Preliminary Definitions

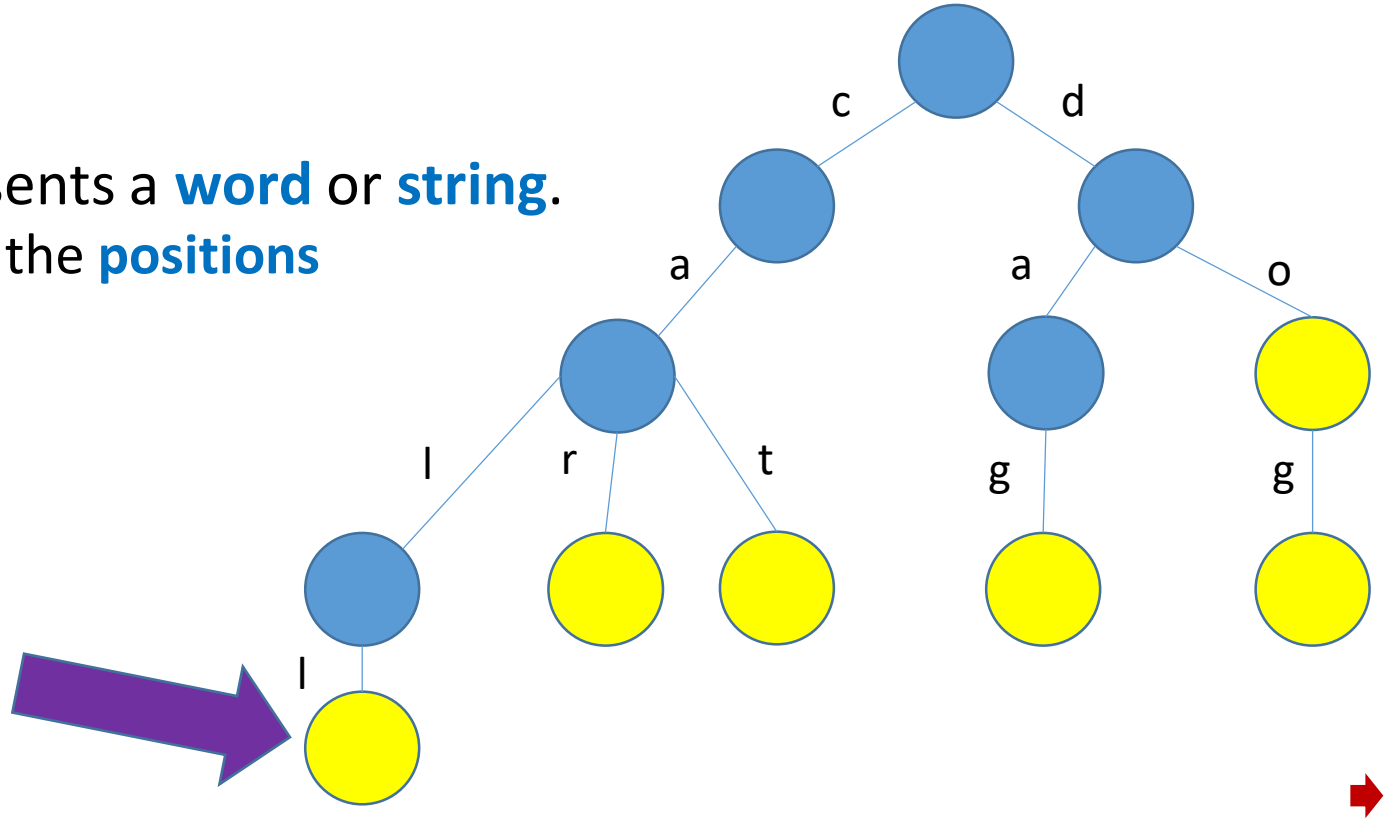
- Let Σ be finite alphabet of characters. For example $\Sigma = \{a, b, c\}$.
- Let T be a string (contiguous sequence of characters) from Σ.
- A substring is a contiguous sequence of characters within a string. For instance, “ana" is a substring of “banana".
- Do not confuse substring with subsequence, which is a generalization of substring.
  - For example, ‘bnn’ is a subsequence of ‘banana’, but not a substring.
- Prefix and suffix are special cases of substring.
  - A prefix of a string T is a substring of T that occurs at the ***beginning*** of T.
  - A suffix of a string T is a substring that occurs at the ***end*** of T.
- A substring of a string $T = t_1 \dots t_n$ is a string $\hat{T} = ti \dots tj$, where $0 \leq i < j \leq n$.

WIKIPEDIA
The Free Encyclopedia

A Trie

- A **Trie**, **pronnouced** '**try**', is a **tree** that is **used** to **store strings**.
- The **edges** are **labelled** with **characters** from the **alphabet** Σ.
- A **yellow** node is **used** to **denote** the **end** of **words**. All **leaves must** be **yellow**.
- The **trie shown** on the **bottom right** stores the **words**:
  - *cat*
  - *car*
  - *call*
  - *dog*
  - *dag*
  - *do*
- Every **path** of **labelled edges** represents a **word** or **string**.
- A **yellow node** can, for **instance**, **store** the **positions** where the **word occurs** in a **text**/**s**.

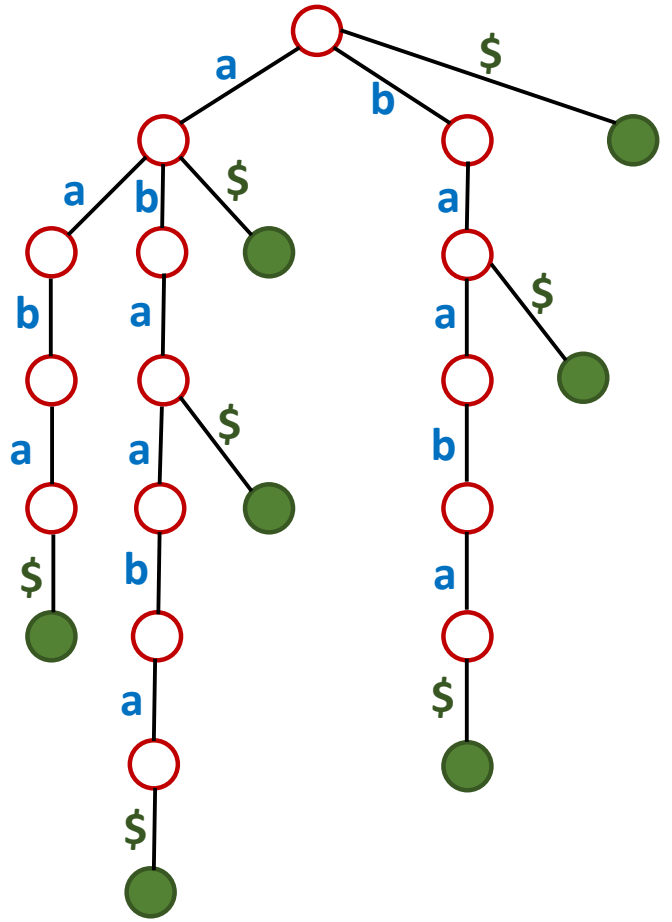
call
Document A, Page 3, Word 61
Document A, Page 5, Word 12
Document B, Page 14, Word 103
Document D, Page 27, Word 44



Suffix Tries

- A **Suffix Trie** is a **Trie** that **stores** all the **suffixes** of a **word**.
- Suppose the string **T** = **abaaba** → then **T$** = **abaaba$**
- Note that we **add** a **sentinel character**, usually **$**, just in **case** there are **suffixes** that are **prefixes** of other **suffixes**. Eg. **ba** is a **prefix** of **baaba**.
- The **suffixes** are:

abaaba$
baaba$
aaba$
aba$
ba$
a$
$



- Note that **every path** from the **root** to a **leaf** represents a **suffix** of **T**.
- What **would** have **happened** if we had **not added** a **$** to the **end** of the **string T**?
- How **many nodes** & **leaves** do we **have** in **such** a **trie**?
- How **many children** can a **node** have?
- The **sentinel character** is sometimes **\0** or **CTRL-0**.
- How **does** one **check** for **suffixes** and **substrings** in a **Suffix Trie**?

Global Suffix Tries (GSTs)

- A **Suffix Trie** can be **used** to **store more** than **just** a **word**.
- Such a **trie** is **called** a **Global Suffix Trie**.
- The **starting position** (**offset**) of **each word** or **suffix** is **stored** in the **leaves**.
- This **essentially** creates an **index** of **all** the **words** in a **document** (or **documents**) and **allows** for **very fast searching**.
- A **GST** can be **used** to index **hundreds**, **thousands**, **millions**, or even **billions** of **documents**.
- The **search time** complexity of a **GST** (like **ALL** trees) is **linear** in the **height** of the **tree**.
- The **height** of the **GST** is **bounded** by the **length** of the **longest word** in the **documents**.
- The **search time complexity** does **not change** when more **documents** are **added**.
- **GSTs** are **sometimes** called **Generalized Suffix Trees**.

Global Suffix Trie

Construction

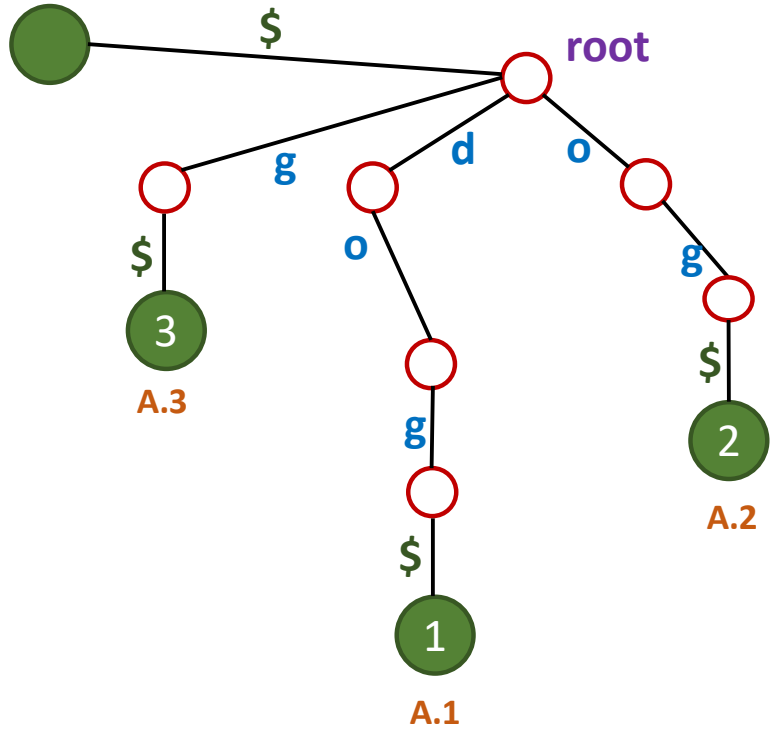
Document **A**
dog dig dot
12345678911
01

Document **B**
dob dig doc
12345678911
01

offset

Document **A**
Word **1**

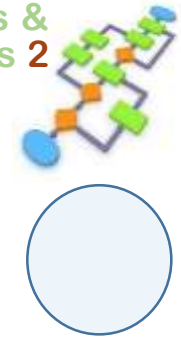
dog$
og$
g$



- The **Global Suffix Trie** (**GST**) is **built** one **suffix** at a **time**.
- The **start location** for each **suffix** is **stored** in the **corresponding leaf**.
- In a **typical application**, the **suffix locations** are **stored** in **indexed lists**.
- The **time complexity** for **searching** for a **suffix** is **O(|*s*|)** where ***s*** is the **longest word** in the **document corpus**.
- The GST **grows very quickly** at **first** but then **settles down**.
- The **indexed lists** keep **growing** as more **documents** are **added** but the **size** of the **GST** remains **stable**.
- **Note** that **no new nodes** are **added** for **words** that are **already** in the **GST**. The **indexed lists** are **updated** for every **word occurrence**.

Global Suffix Trie

Construction



offset

Document **A**

dog dig dot

12345678911

01

Document **B**

dob dig doc

12345678911

01

Document **A**

Word **1**

---

dog$

og$

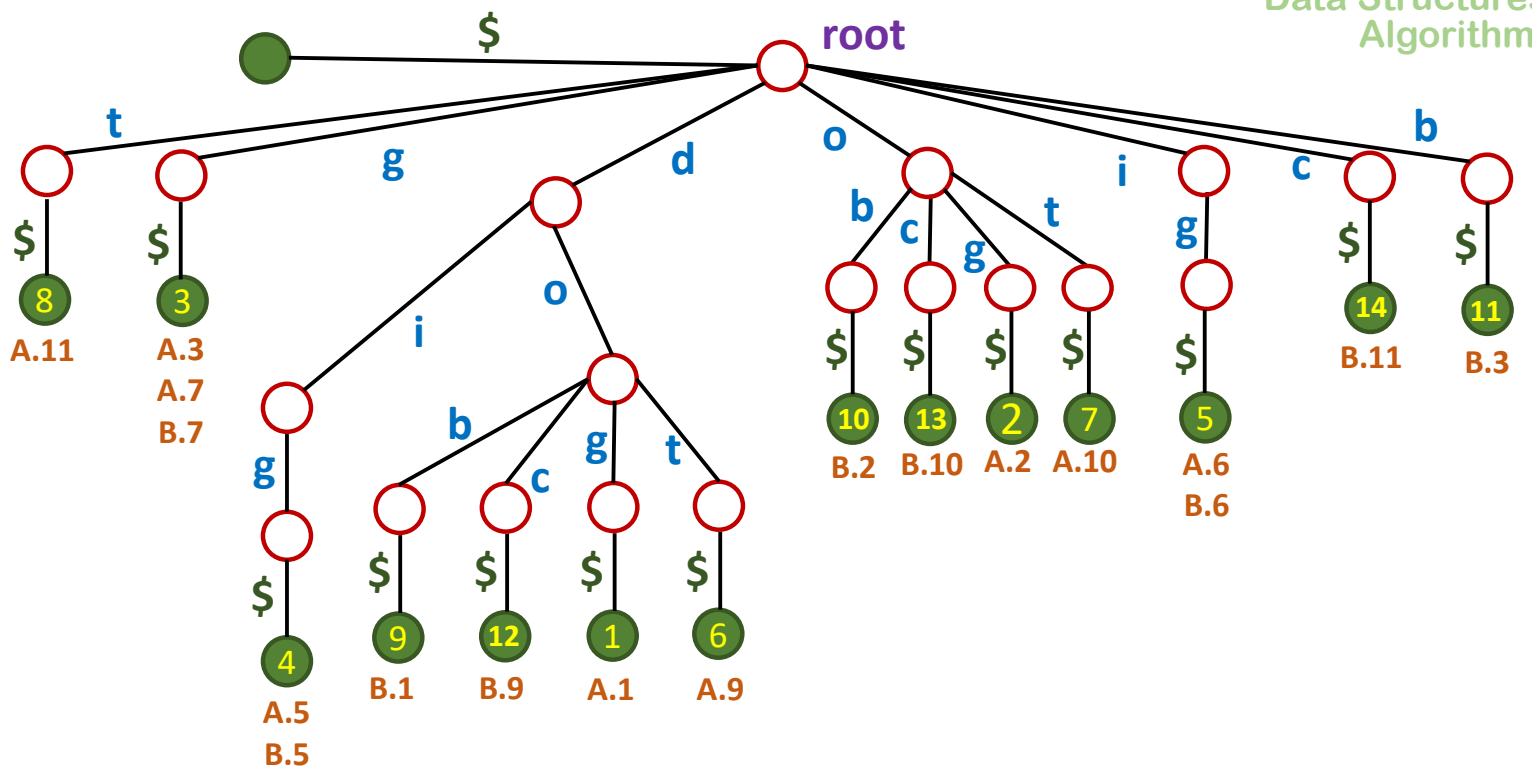
g$

Global Suffix Trie

Construction

Document **A**
dog dig dot
12345678911
01

Document **B**
dob dig doc
12345678911
01



- The **words** (and **suffixes**) in **each document** are **added one** by **one** to the **GST**.
- The GST **stabilizes** (in **size**) and **changes** only when **new words** (**never seen** before) are **encountered**.
- When **all** the documents are **processed** it will be **possible** to **search** for any **word** (and its **prefixes**, **substrings**, and **suffixes**) **very quickly** using the **GST**.
- The **locations** of the **suffixes** are **not really stored** in the **leaves**. They are **stored** in **indexed arrays**. The **leaves** will **contain** only **pointers**.
- In **theory**, one can **add** an **unlimited number** of **documents**.

Global Suffix Trie

Construction

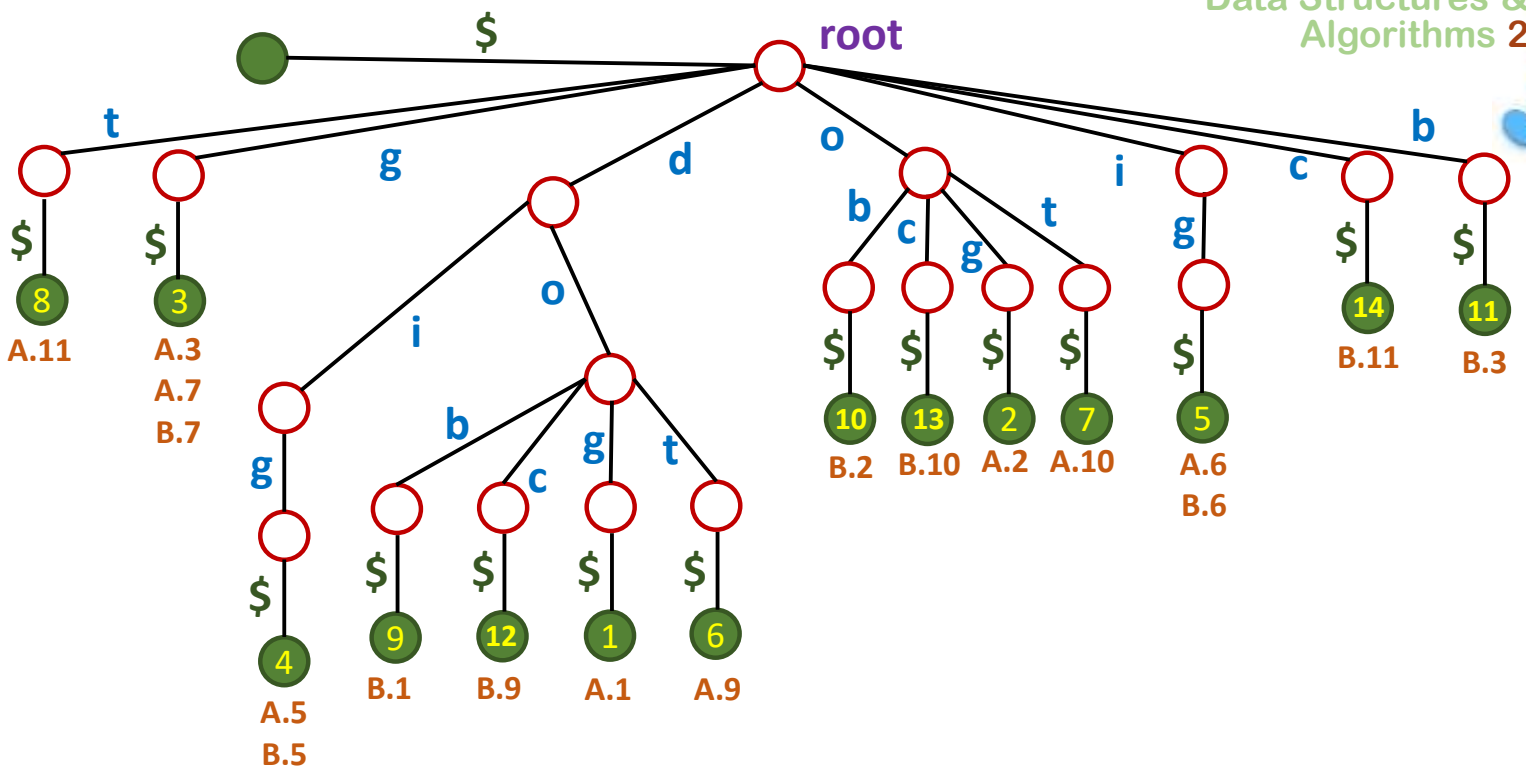
Node	Links
1	A.1
2	A.2
3	A.3, A.7, B.7
4	A5, B.5
5	A.6, B.6
6	A.9
7	A.10
8	A.11
9	B.1
10	B.2
11	B.3
12	B.9
13	B.10
14	B.11

Document **A**

```
dog dig dot
12345678911
          01
```

Document **B**

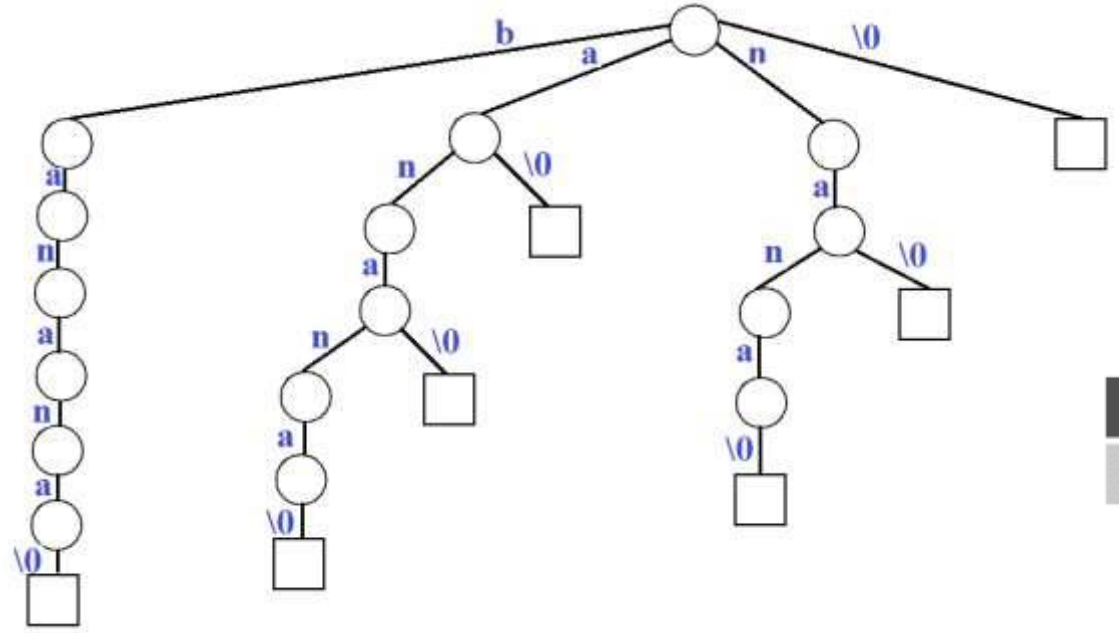
```
dob dig doc
12345678911
          01
```



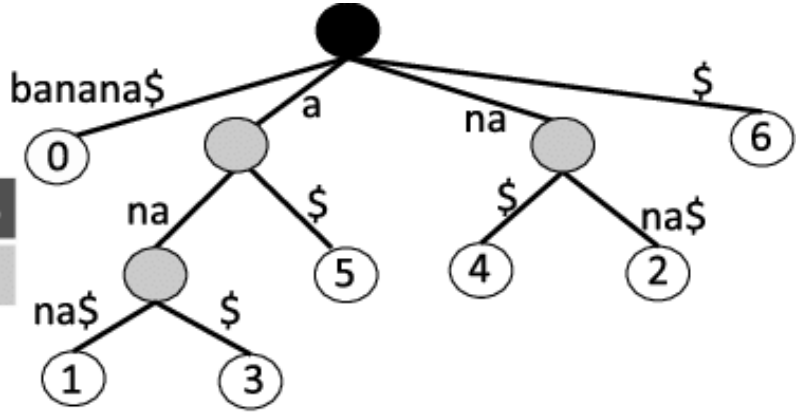
- In the **example above**, the **leaves** of the Trie **contains** the **links** (**pointers** to the **offsets** of each **suffix**).
- **In practice**, the **links** are **stored** in a **separate data structure** – often an **indexed database** table.
- It is **common** to **store** the **suffix trie** (actually this is **normally** a **suffix tree**) in **main memory**.
- The **links** are **typically stored** in a **very fast relational database**.

Suffix Trees

- A **Suffix Tree** is a **compressed** Suffix Trie.
- The **number** of **nodes** in a **Suffix Trie** can be **quadratic O(n^2)** in the **length** of the **indexed string**.
- It is **not efficient** to **retain** internal **nodes** with **only one** child.
- The **suffix trie** can be **compressed** into a **suffix tree** by **merging edges** by **removing internal nodes** with **only one** child.
- This can **drastically reduce** the **number** of **nodes** as can be **seen** for the **suffix trie** and **suffix tree** for the string **banana$**.

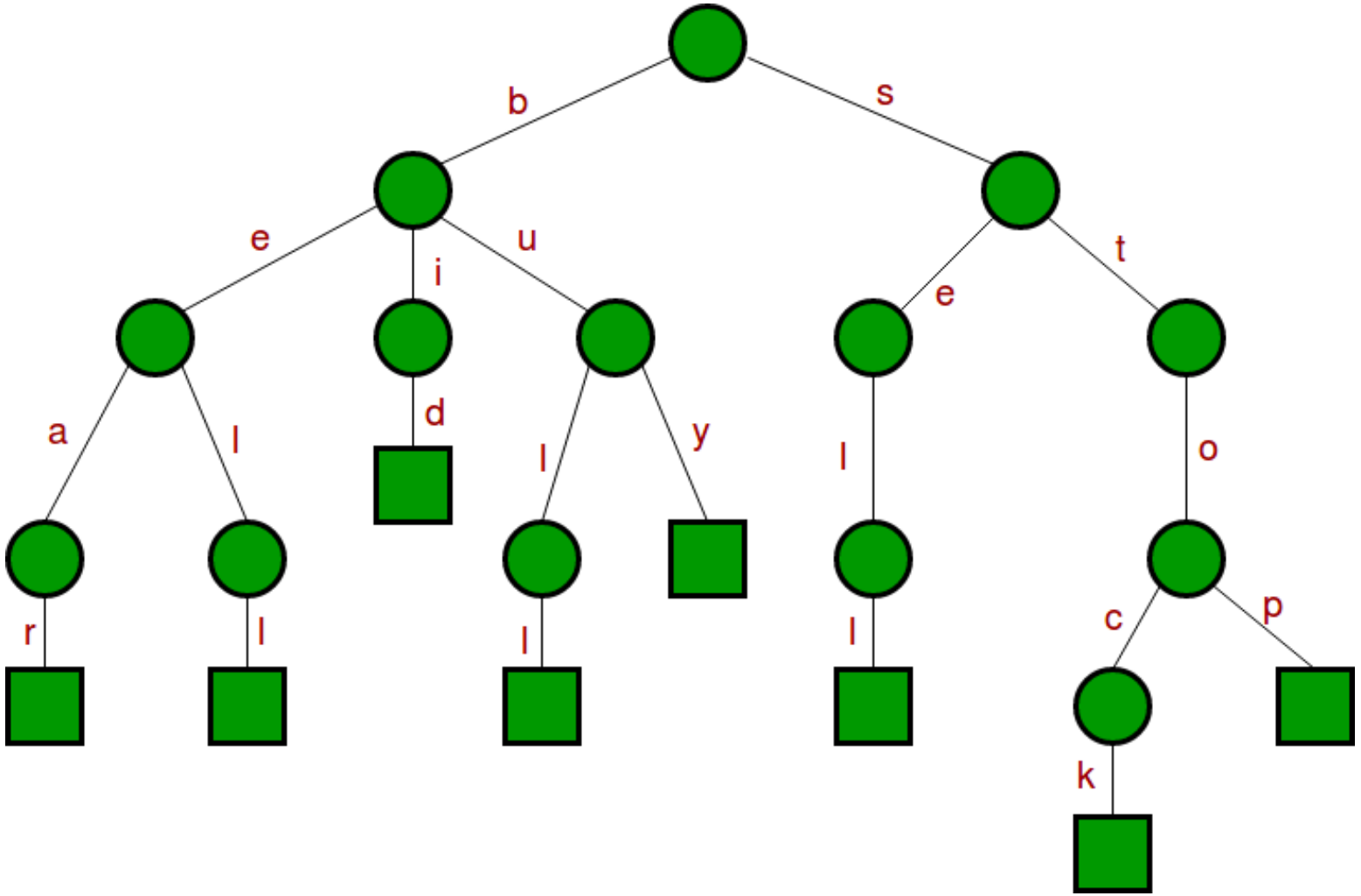


0	1	2	3	4	5	6
b	a	n	a	n	a	$



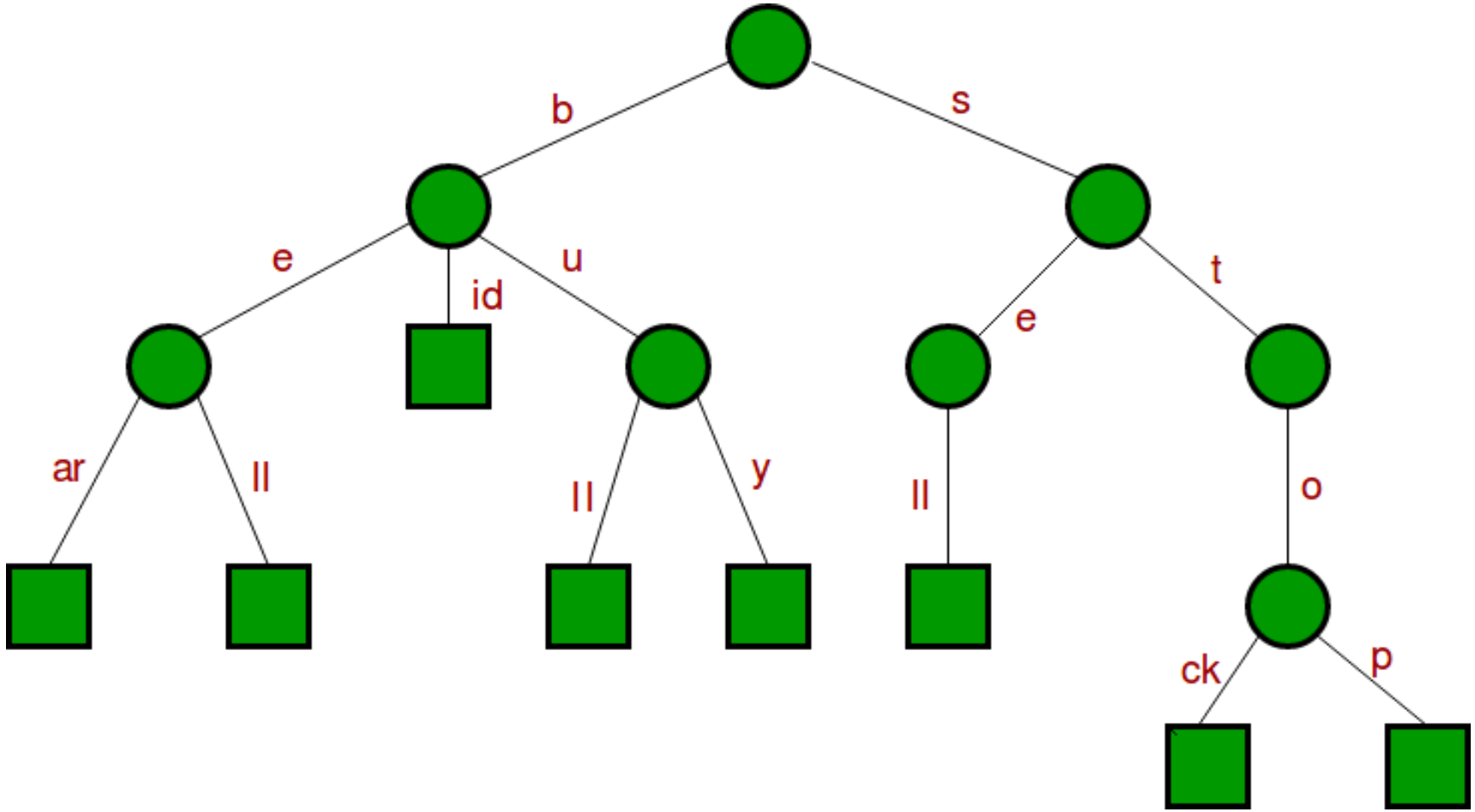
Suffix Trees

- **Consider** the **set** of **strings** {bear, bell, bid, bull, buy, sell, stock, stop}.



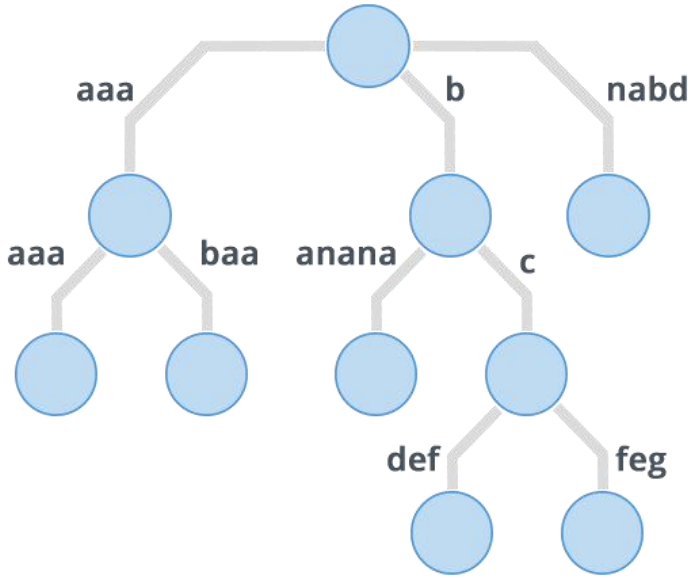
Suffix Trees

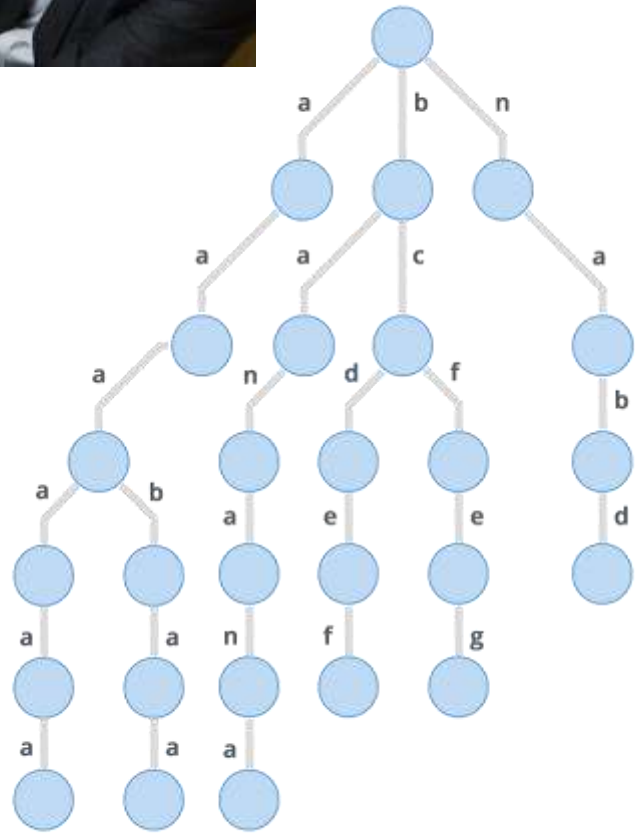
- After **compression**.
- A **suffix trie becomes** a **suffix tree**.



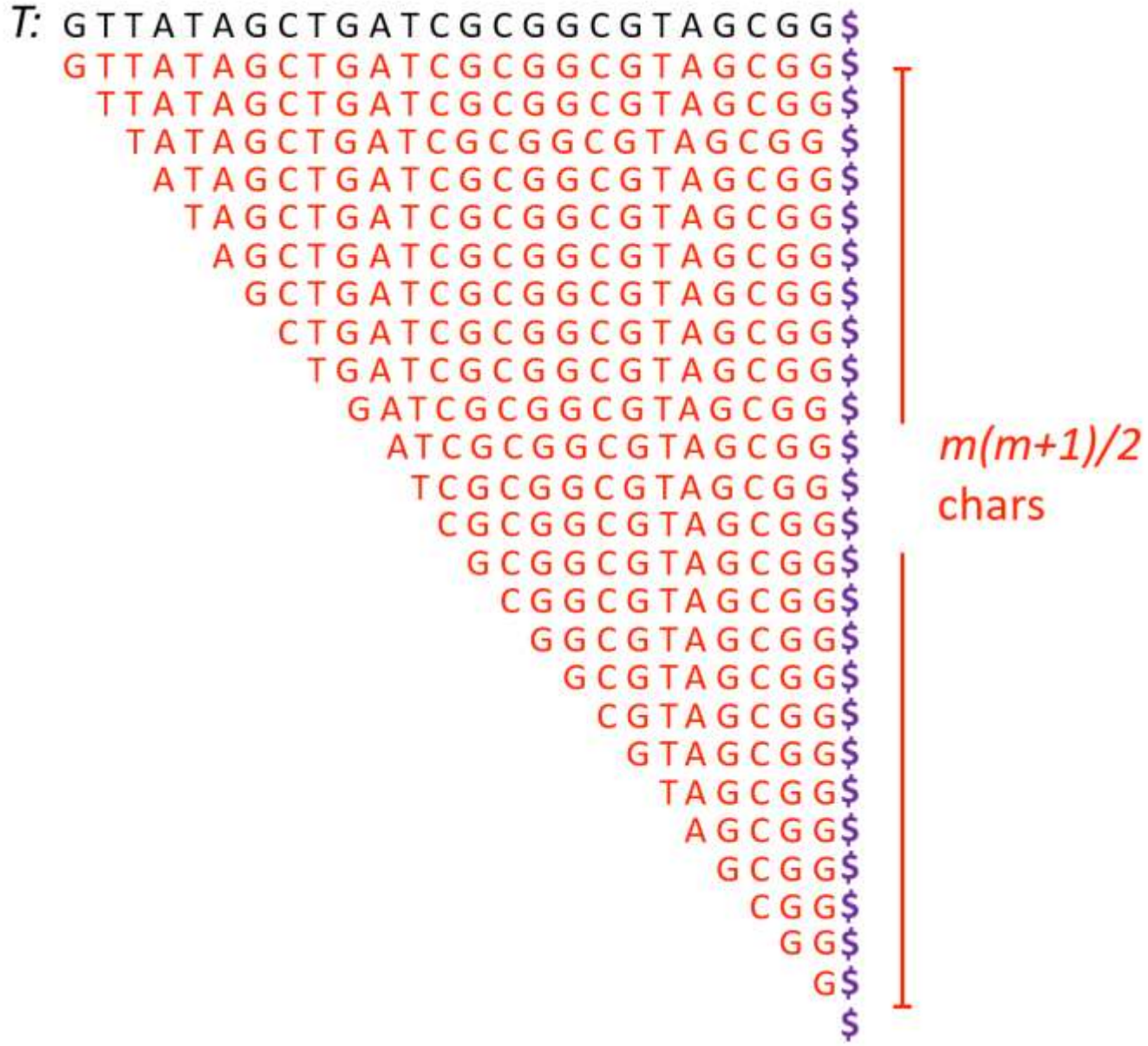
Suffix Trees

- A **Suffix Tree** is, **therefore**, nothing **more** (or **less**) **than** a **compressed Suffix Trie**.
- One does **not build** a **suffix trie** and then **compress** into a **suffix tree**.
- A **suffix** tree **can** be **built directly** from a **set** of **strings** (**words**) in **linear time** using **Ukkonen's algorithm** (**1995**).
- The **tree space complexity** is **O(*n*)** instead of **O(n^2)** in a **normal trie**.
- The **total space complexity** for **construction** is still **quadratic**.



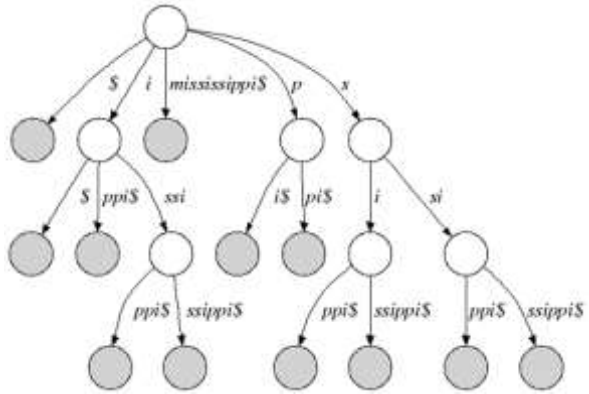


Suffix Trees Storage Space

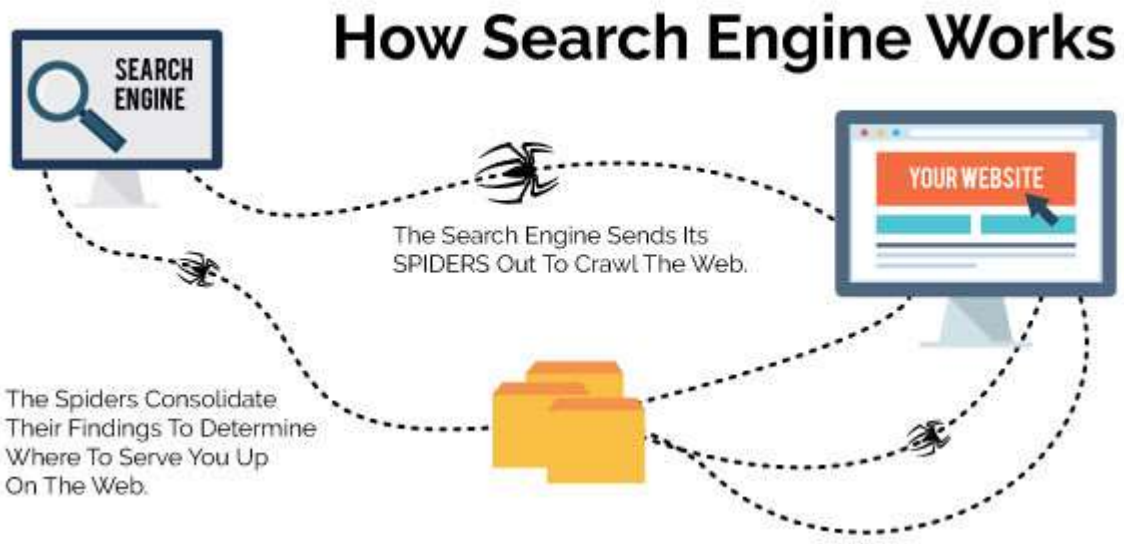


How (we think) Google Works.

- A **Suffix Trie** and **Trees** are the **fundamental technology** used in **document indexing**.
- The **time complexity** for **building** the **index** is **linear** in the **length** of the **words**. **Ukkonen's** algorithm is **used**.
- To **index** a **large corpus** of **documents** it may **take** some **time**.
- For these **reasons indexing** (**building** the **suffix** trees) and **querying** (**searching** the **suffix** trees) are **two separate** processes.
- Once the **index** is **built** then **searching** for a **string** is **almost instantaneous**.
- Can be **extended** to **search** for **regular expressions**.

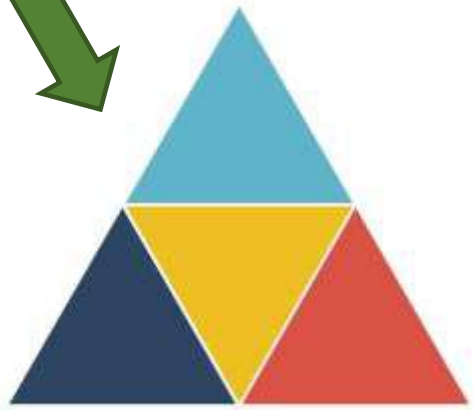


How (we think) Google Works
Indexing



Indexing Bot Farm

- May contain 1000s of indexing bots.
- Each bot augments the Master GST with new documents and web content.
- Every ***n*** minutes the updated Master GST is sent to the Search Bots.



Master Index (GST)

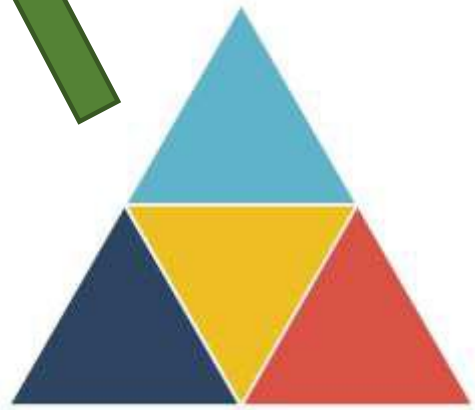
- Continuously updated by the indexing bots.

How (we think) Google Works
Searching

Web User makes query.

Query Broker
- Finds a ‘free’ search bot and sends query.
- The bot uses the GST to perform the search.

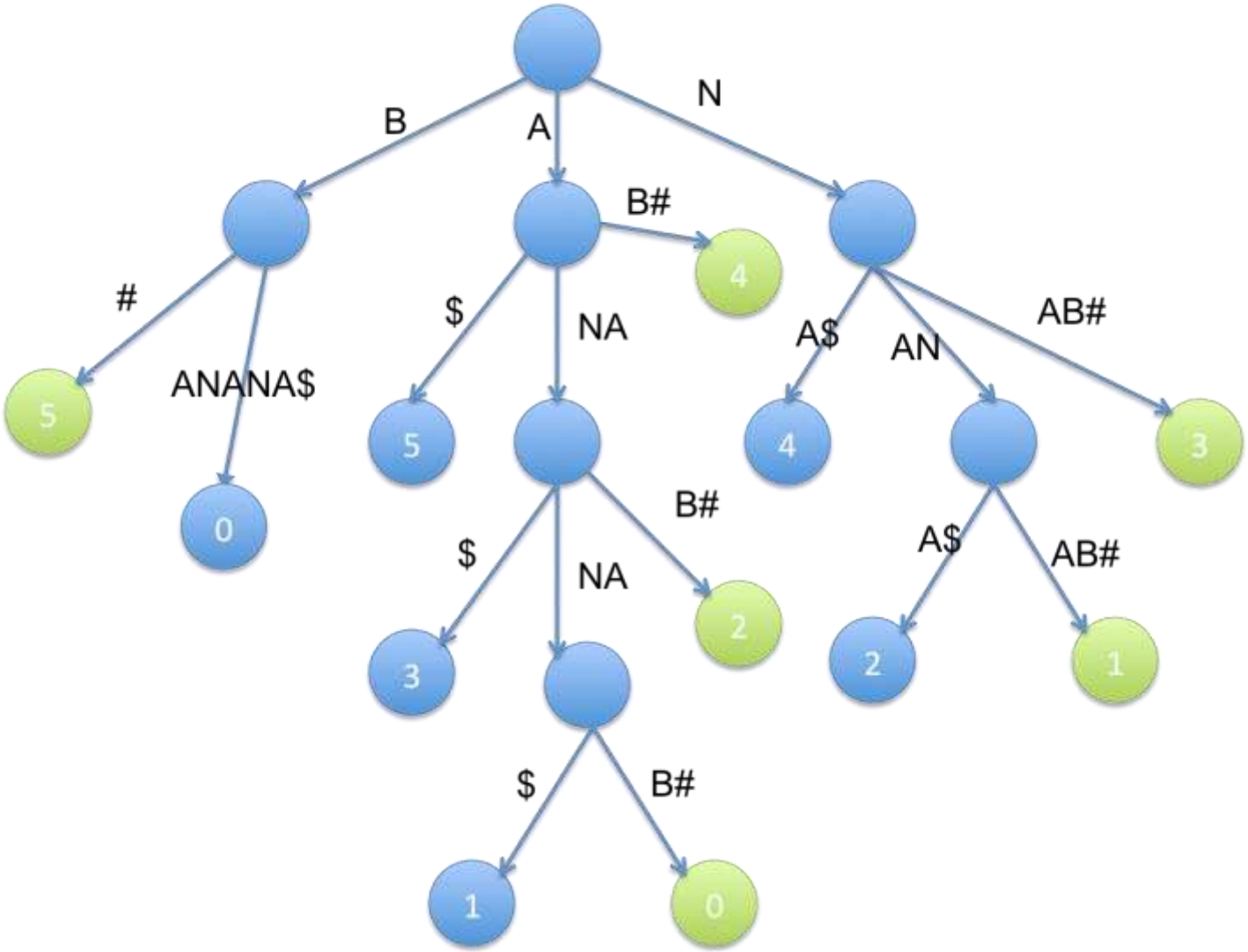
Search Bot Farm
- May contain 1000s of search bots.
- Each bot has a copy of the Master GST.
- Bots stop servicing queries now and again to update the GST.



Master Index (GST)

Continuously updated by the indexing bots.

End of Module



END