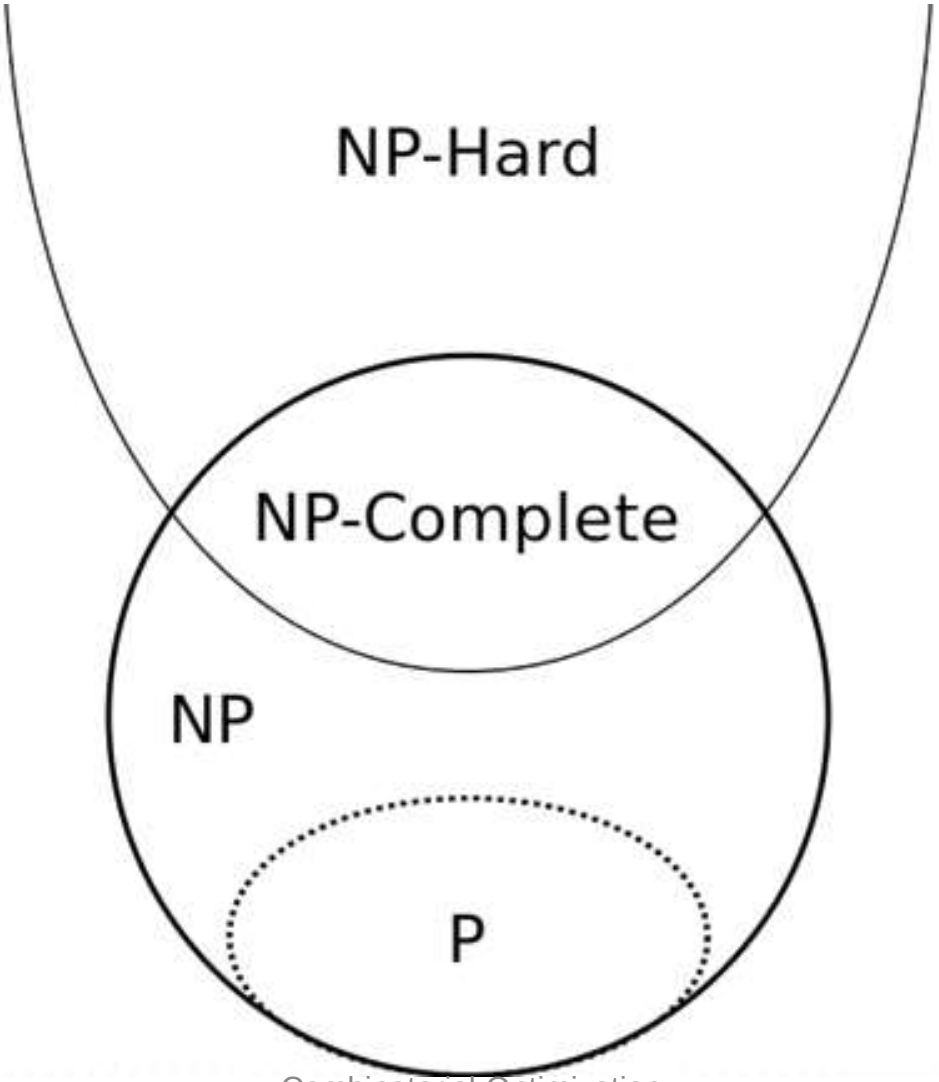
# Complexity and Complexity Classes

What is a hard problem?

# Today's contents



# Sources

This lecture includes hyperlinks to Wikipedia pages. These are mostly subsidiary to the course: relevant information but not necessarily what you may expect during the exam.

The bibliographic suggestions presented in the first lecture adequately cover the course contents.

Today we address the ‘prologue’ (chapter 0) from Dasgupta, in particular 0.3 the Big-O notation. Reading Chapter 1 (algorithms with numbers) is highly encouraged! We move towards chapter 8: NP-complete problems, as summarized in the course notes on Canvas.

# What is a computer?

A computer is a model of deterministic computation. Formally we can think of a computer as a Turing machine.

Turing machines are equivalent to machines that have an unlimited amount of storage space for their computations.

However, Turing machines are not intended to model computers, but rather they are intended to model computation itself.

Historically, computers, which compute only on their (fixed) internal storage, were developed later than Alan Turing defined his machine.

Turing called his concept a machine because it could actually be built. An algorithm (as we know them now) would be 'hard-wired' in the machine.

For us it is a formalism. Formalisms are needed to be able to proof theorems!

# You may have met Alan Turing already

https://www.wired.com/2012/06/lego-turning-machine-how-to/

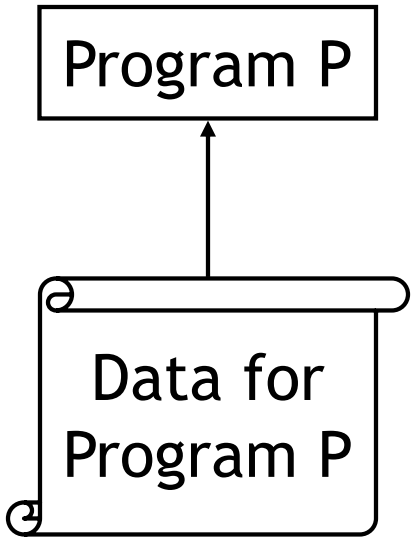
# In fact…

we can replace the concept of a Turing machine by any regular programming language running on a regular computer.

The key concept is *determinism*: the program does exactly what it is meant to do using basic steps of computation.

Let us meet a simple problem posed on deterministic programs and their input…

# Halting problem




Question: Given the input data, will Program P ever halt, or will it run forever?

Approach: Try running it

- If it halts, we know the answer
- If it hasn't halted *yet*, we *don't* know the answer

How long do we have to wait ?

Can we do better?

# Is undecidable!

See http://en.Wikipedia.org/wiki/Halting_problem

Suppose that `halt(p, i)` returns **true** if the string `p` describes a program that halts when given as input the string `i`, and returns **false** if `p` does not halt on `i`.
If two programs are realizable on a Turing machine, then executing both of them in sequence is also realizable, and so is executing one depending on a condition. Construct a program **`trouble(s)`** that does the following:

- **Call `halt(s,s)`**
- **If halt returned true, then loop forever.**

Since all programs have string descriptions, there is a string *t* that represents the program trouble. Does `trouble` halt when its input is `t`?
Consider both cases:

- If `trouble(t)` halts, it must be because `halt(t, t)` returned **false**, but that would mean that `trouble(t)` should not have halted.
- If `trouble(t)` runs forever, it is either because halt itself runs forever, or because it returned **true**. This would mean either that halt does not give an answer for every program and input, or that `trouble(t)` should have halted.

# Dealing with hard problems

There are some problems that **cannot** be solved by computers!


I couldn't find an algorithm,
because no such algorithm exists!

# Analysis of Algorithms that DO halt

Estimate the running time.
Estimate the memory space required.

NOTE: Time and space depend on the input size.
NOTE: We need to define input size!

Let us first be clear about what a combinatorial optimization problem is…

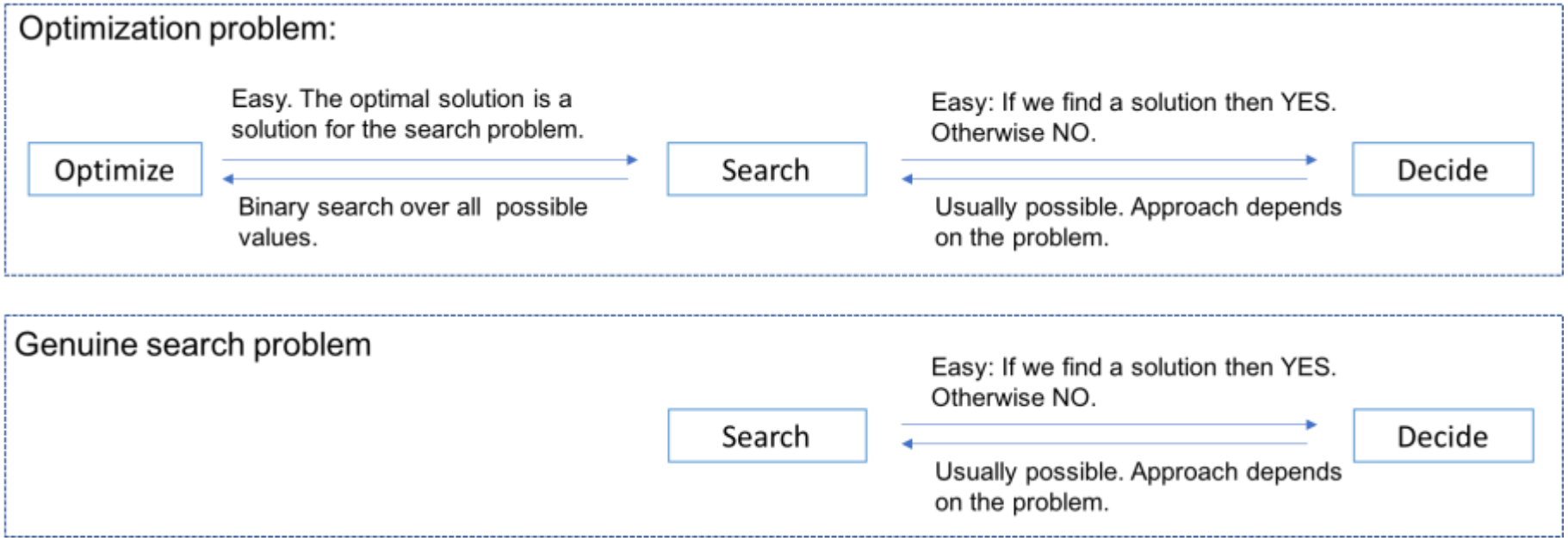
# What is a Combinatorial Optimization problem?

A combinatorial optimization problem consists in defining the following concepts:

- A set of *instances*.
- For each instance a set of candidate *solutions*.
- For each instance, a set of *valid solutions*. These are the candidate solutions with a certain *property* that we look for.

Sometimes referring to a **solution** means **optimal solution** because the problem in question defines optimality (e.g. max flow on a network) as the desired property. It should be obvious from the context.

# The family of combinatorial problems



# Example: the prefix averages

The very first example of the first lecture consisted of:

- Given a vector of $n$ numbers: the *instance* $x$
- Produce a new vector of $n$ numbers: the *solution* $a$
- Such that $a_i = \frac{1}{i}\sum_{j \leq i} x_i$: the *property*

# How do we solve a combinatorial optimization problem?

A combinatorial optimization problem is solved by an algorithm.

An algorithm is a set of computations that can be represented in a Turing machine.

For us an algorithm is anything that we can implement using a programming language on a deterministic (i.e. a *common*) computer.

# What is the size of an instance?

The size of an instance is the number of independent parameters necessary to define the instance.

In the prefix averages example that is $n$ since you need that many numbers to define and instance.

In the example of deciding if a graph admits an Eulerian cycle the instance is a graph and its size is the number of nodes $n$ and edges $m$. The size can therefore be given by more than a single number. This is also the case for the shortest path and for the max flow.

# Effort of an algorithm

The effort of an algorithm is measured by the number of elementary operations it requires when applied to a given instance.

This number is expressed as a function of the size of the instance.

Only the *smallest order* of the effort function matters.

# What are elementary operations?

An elementary operation is a piece of computation expressible in a programming language which takes some ***constant amount*** *of time*.

Examples are:

- Evaluate an expression consisting of variable values, constant values and arithmetic operations: `s/i`
- Dereference a variable: `a[i]`
- Assign the result of an expression to a variable: `a[i] = s/i`

Calling a function as `sin`, `sqrt`, `log`, etc.. also counts as elementary operation, while a function such as `sort` does not. These differences may be subtle at first… but should be clear after any course on data structures and algorithms.

# Big-O

$$f(n) = O(g(n))$$

If for $n$ above some number $k$ a constant $c$ exists for which:

$$f(n) \leq cg(n)$$

# Big-Omega

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

If :

$$g(n) = O(f(n))$$

# Big-Theta

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

If :

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

*and*

$$g(n) = O(f(n))$$

# Some Big-O examples from the definition

- $7n - 2$ is $O(n)$
  take $c = 7$ and $k = 1$.

- $3n^3 + 20n^2 + 5$ is $O(n^3)$
  take $c = 4$ and $k = 21$.

- $3 \log n + 5$ is $O(\log n)$
  take $c = 8$ and $k = 2$.

# Big-O rule of thumb

To simplify the running time estimation, for a function $f(n)$, we just ignore the constants and lower order terms.

Example: $10n^3 + 4n^2 - 4n + 5$ is $O(n^3)$.

# Big-O and Growth Rate

The Big-O notation gives an upper bound on the growth rate of a function.

The statement "$\boldsymbol{f(n)}$ is $\boldsymbol{O(g(n))}$" means that the growth rate of $\boldsymbol{f(n)}$ is no more than the growth rate of $\boldsymbol{g(n)}$.

We can use the Big-O notation to rank functions according to their growth rate.

# Common Growth Functions

- Constant $\approx 1$
- Logarithmic $\approx \log n$
- Linear $\approx n$
- Log Linear $\approx n \log n$
- Quadratic $\approx n^2$
- Cubic $\approx n^3$
- Polynomial $\approx n^k$
- Exponential $\approx 2^n$
- Factorial $\approx n!$

(aka super exponential, exhaustive, etc..)

# Growth Rates Compared

| | n=1 | n=2 | n=4 | n=8 | n=16 | n=32 |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| *logn* | 0 | 1 | 2 | 3 | 4 | 5 |
| *n* | 1 | 2 | 4 | 8 | 16 | 32 |
| *nlogn* | 0 | 2 | 8 | 24 | 64 | 160 |
| $n^2$ | 1 | 4 | 16 | 64 | 256 | 1024 |
| $n^3$ | 1 | 8 | 64 | 512 | 4096 | 32768 |
| $2^n$ | 2 | 4 | 16 | 256 | 65536 | 4294967296 |
| $n!$ | 1 | 2 | 24 | 40320 | 20.9T | Don’t ask! |

# Growth Rates Compared

| | n=1 | n=2 | n=4 | n=8 | n=16 | n=32 |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| *logn* | 0 | 1 | 2 | 3 | 4 | 5 |
| *n* | 1 | 2 | 4 | 8 | 16 | 32 |
| *nlogn* | 0 | 2 | 8 | 24 | 64 | 160 |
| $n^2$ | 1 | 4 | 16 | 64 | 256 | 1024 |
| $n^3$ | 1 | 8 | 64 | 512 | 4096 | 32768 |
| $2^n$ | 2 | 4 | 16 | 256 | 65536 | 4294967296 |
| $n!$ | 1 | 2 | 24 | 40320 | 20.9T | Don't ask! |

# notes

Running time is not a property of the problem, is a characteristic of algorithms and their implementation details!

It is common to say 'this problem can be solved in $O(f(n))$' but it is not always obvious how, even if one can easily think on a way to implement a solution procedure.

As we will saw last week, a roughly described algorithm leaves so many details to the implementation that the factual runtime may change orders!

# The most important question in combinatorial optimization

Given a problem, can we **find** a solution with the **desired property** in polynomial time?

All problems addressed last week belong to that category.

Today we will meet other type of problems and learn how to distinguish them.

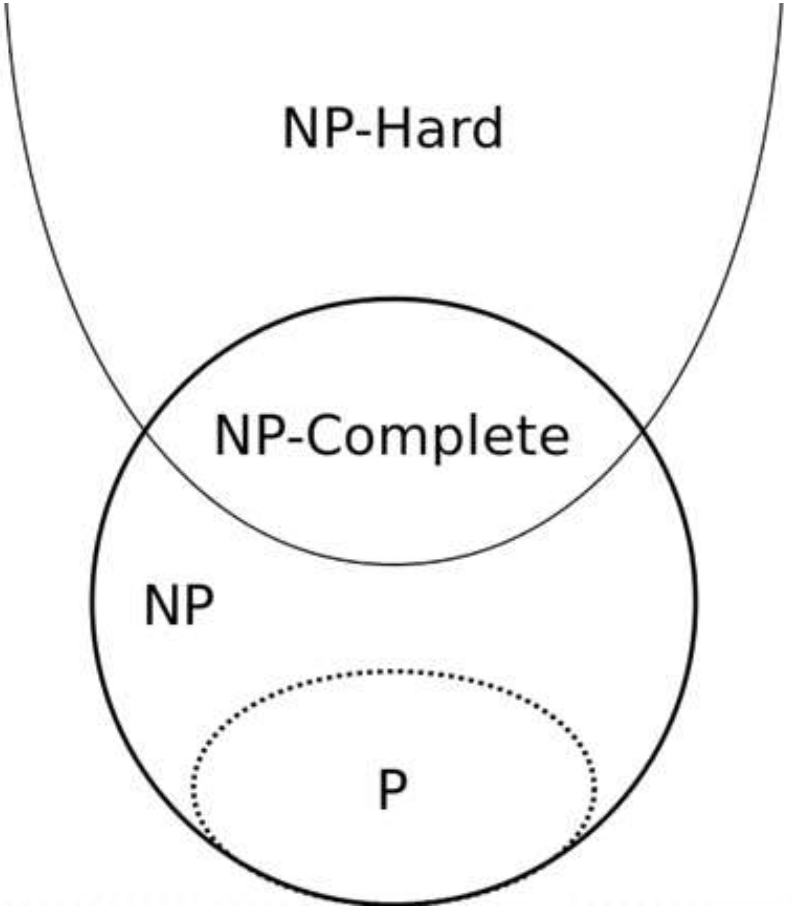
# Search problems

Remember that a combinatorial optimization problem consists in defining the following concepts:

- A set of *instances*.
- For each instance a set of *solutions*.
- A subset of the solutions: those with a desired *property*.

Now we need a new definition: our combinatorial problem is a **search problem** if given an instance, a solution and a property the **verification** that the solution is valid and satisfies the property can be done in polynomial time.

# Complexity classes: chapter 8 of Dasgupta



# Definitions

**NP** is the class of all **search problems**.
**P** is the class of all **NP** problems that can be **solved** in polynomial time.

**NP-Complete** are those problems in **NP** that are at least as hard as any other problem in **NP**.

Notice how **NP-hard** exceeds beyond **NP**: the halting problem is an example of an NP-hard problem that is not in NP. We will explain that later.

# At least as hard as…

A problem P is at least as hard as Q if P can be used to *efficiently* solve Q.

*Efficiency* is measured in terms of the computational effort required.

Efficient use of P to solve Q means that the *number of times* one uses P to solve Q is *bounded by a polynomial* and that the length of each instance solved by P is also *bound by a polynomial*.

# This leads to the notion of *polynomial reduction*

Reduction amounts to solving a problem by (repeatedly) creating instances of another problem which we know how to solve.

The reduction is polynomial whenever the number and size of the instances of the sub problem is polynomial on the size of the original problem.

# Examples of reduction

- The Ford-Fulkerson max flow method looks for an ‘augmenting path’ on each iteration.
- The (revised) simplex method looks for the non basic variable with the lowest reduced cost.

NOTE: both examples are **not polynomial**! The first became **polynomial** and led to the Edmonds-Karp-Dinitz algorithm. The simplex algorithm is proven not to be polynomial for each of the presently known pivot rules: Examples exist leading to an exponential number of iterations. Remains an open question whether a version of the simplex method exists which is polynomial.

# Transformations

If we find a reduction with length 1 then we call it a transformation.

Example: the Linear Assignment Problem (also known as bipartite matching) can be solved by transforming it into a min cost flow problem. The LAP consists of: given a square matrix $c$ of numbers $c_{ij}$ assign each row $i$ once to a unique column $j$ such that the sum of the corresponding $c_{ij}$ is minimal.

# Question

Explain how the transformation of the Linear Assignment problem to the min cost flow problem works.

# Size again…

We defined size independently of the magnitude of numbers in the problem instance.

This was important when we met the Ford-Fulkerson method: its effort depends on the capacities of the arcs.

In chapter 1 of the book Algorithms by Dasgupta we notice that the running time of algorithms may depend on how large numbers are!

One form of dependency is allowed while still placing algorithms in P.

# What if the actual symbols matter?

When the magnitude of the numbers matter (e.g. RELPRIME, LP, etc.) then we need to measure the length of the numeric representation.

Usually we adopt a binary representation of a number $x$ and take $n = \log_2 x$

NOTE: for a comprehensive explanation about the complexity of primality see http://en.Wikipedia.org/wiki/Primality_test.

# Let us recapitulate…

NP

P

NP-complete or NPC

NP-hard or NPH

# The class NP

The class NP is defined in terms of **yes** answers.

A recognition problem is in NP whenever a yes answer can be ***checked*** in polynomial time.

# The class P

A problem is in the class P when an yes answer can be ***found*** in polynomial time.

# NOTE

The statements

-***verifiable*** *in polynomial time by a* ***deterministic*** *Turing machine*

and

-***solvable*** *in polynomial time by a* ***non-deterministic*** *Turing machine*

mean exactly the same thing! That is actually the meaning of NP: Non-deterministic Polynomial.

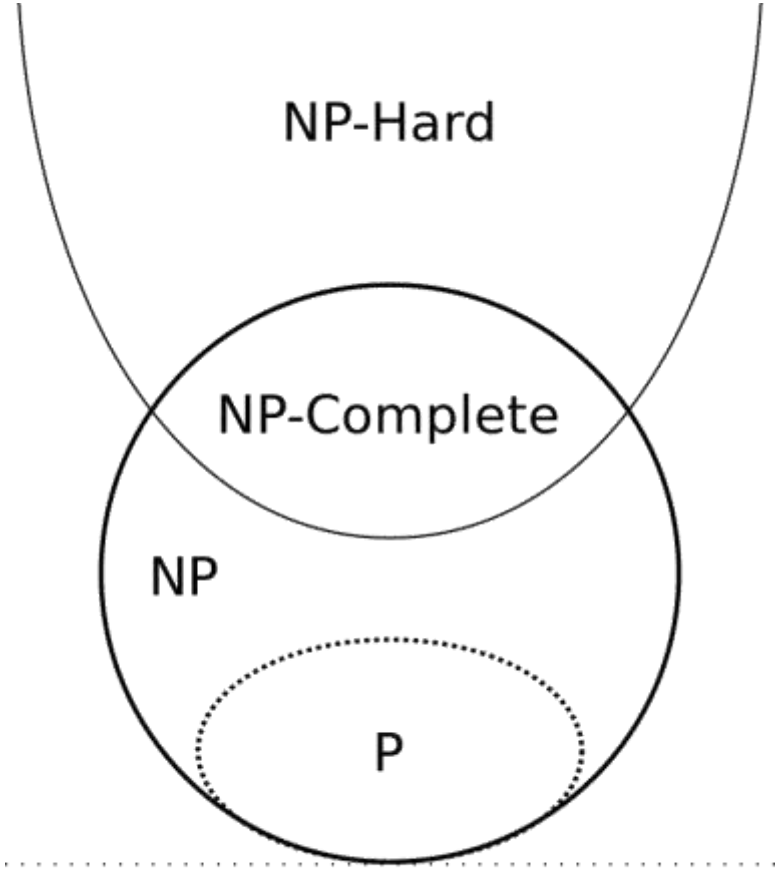
# The class NP-Complete

This may seem the most mysterious class.

In fact is quite simple: a problem is in NPC if

- It is in NP.
- *All* other problems in NP polynomially reduce to our problem.

# A slightly more accurate picture…




$$P \neq NP$$

# The previous picture is correct if…

The previous picture assumes that $P \neq NP$.

Despite being still unsolved (an you may win one million dollar by solving it, see https://en.Wikipedia.org/wiki/Millennium_Prize_Problems) there is at least strong evidence that this is the case.

How do we prove that NPC has at least one element?

# Boolean satisfiability

Suppose a Boolean expression in conjunctive normal form.
Example:

(~x | y | z ) & ( x | ~y | z ) & ( y | z ) & ( ~x | ~y | ~z)

The Boolean satisfiability problem amounts to finding truth values for the variables which make the expression become true.

Let us try…

| x | y | z | ~x\|y\|z | x\|~y\|z | y\|z | ~x\|~y\|~z | |
|---|---|---|---|---|---|---|---|
| FALSE | FALSE | FALSE | TRUE | TRUE | FALSE | TRUE | FALSE |

# Remember NP

A problem is in NP if it can be decided by a non-deterministic Turing machine in polynomial time.

*Another way to read the above definition:*

A problem is in NP if provided a candidate solution this can be verified in polynomial time by a deterministic algorithm in a regular computer.

# SAT is clearly in NP

Provided a truth assignment one can evaluate each clause in time proportional to the number of literals in the clause and hence all clauses in time proportional to the number of clauses.

# Suppose now a slightly different question…

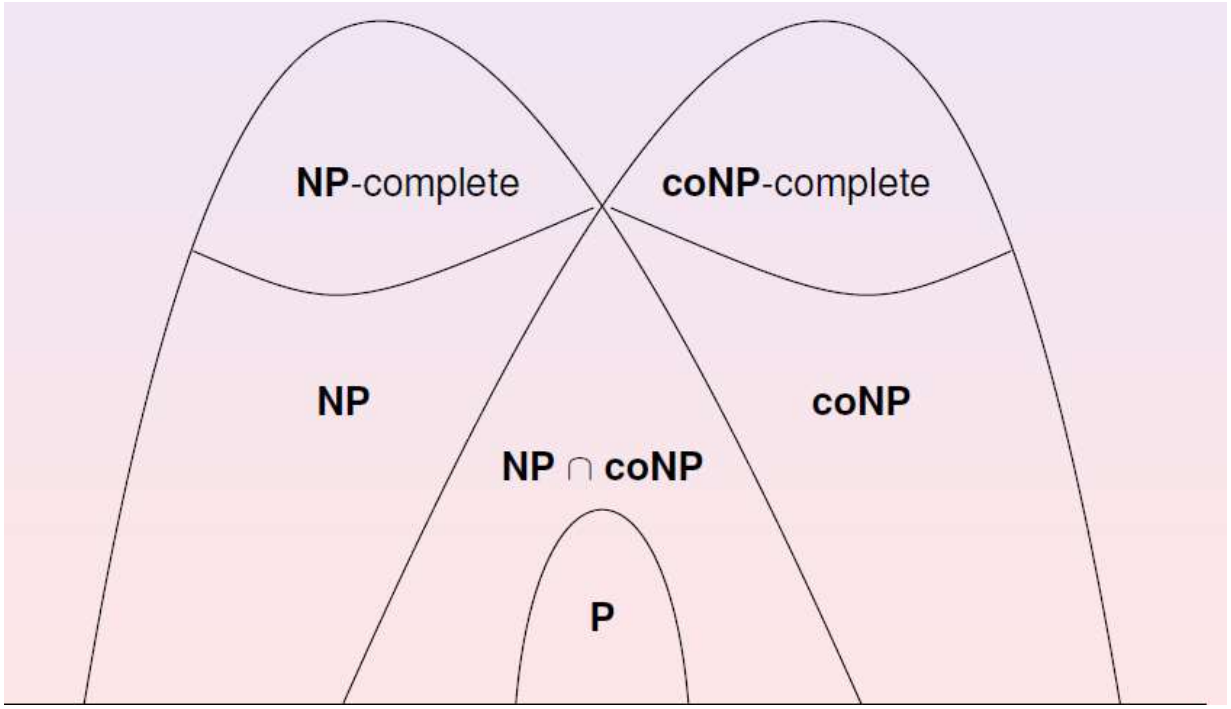
Given a Boolean expression in conjunctive normal form, just like in SAT, suppose now that your problem asks: is it unsatisfiable?

Is this problem in NP?

In this case what is ‘easy’ to check is the ‘no’ answer!

# It is in coNP!

There are many more complexity classes. These go beyond this course, but may be addressed at for instance a Master level course on combinatorial optimization.



# Is SAT in P?

Not known!

But is at least as hard as any problem in NP and therefore it is NPC!

# How do we lay the first egg?

*All* problems in NP polinomially reduce to SAT!

Proven (independently) by Stephen Cook (US, 1971) and Leonid Levin (USSR, 1973).

Idea of the proof:
Suppose that a given problem in NP can be solved by the nondeterministic Turing machine M.
M is polynomially transformed into a Boolean expression B.
The existence of an accepting computation for M is proven equivalent to satisfying B.

# consequence

If the Boolean satisfiability problem could be solved in polynomial time by a deterministic Turing machine, then all problems in NP could be solved in polynomial time by a deterministic Turing machine as well, and so the complexity class NP would be equal to the complexity class P.

# Born from the first egg: Karp's 21 problems as difficult as SAT

In 1972, one year after Cook's theorem, Karp published the following proofs via transformation to SAT:

* 0-1 INTEGER PROGRAMMING
* CLIQUE (see also independent set problem)
  - o SET PACKING
  - o VERTEX COVER
    - + SET COVERING
    - + FEEDBACK ARC SET
    - + FEEDBACK NODE SET
    - + DIRECTED HAMILTONIAN CIRCUIT
      - # UNDIRECTED HAMILTONIAN CIRCUIT
* 3-SAT
  - o CHROMATIC NUMBER
    - + CLIQUE COVER
    - + EXACT COVER
      - # 3-dimensional MATCHING
      - # STEINER TREE
      - # HITTING SET
      - # KNAPSACK
        - * JOB SEQUENCING
        - * PARTITION
          - o MAX-CUT

See also http://en.Wikipedia.org/wiki/Karp%27s_21_NP-complete_problems

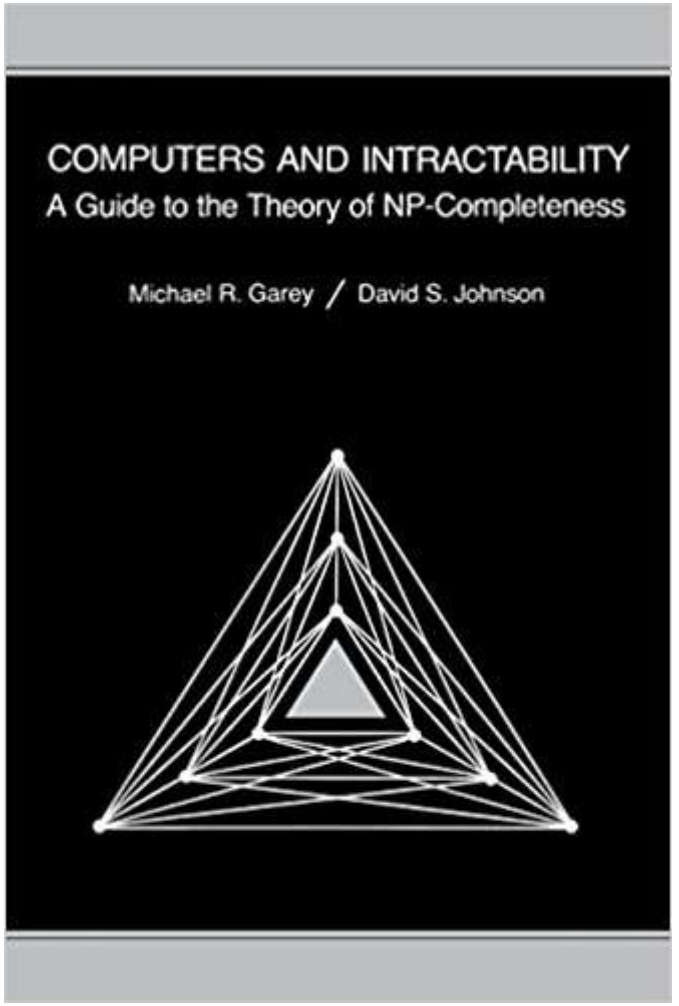
# Many more followed!

There are now hundreds or maybe thousands of distinct and relevant combinatorial problems known to be NPC.

A classical first reference is the book:

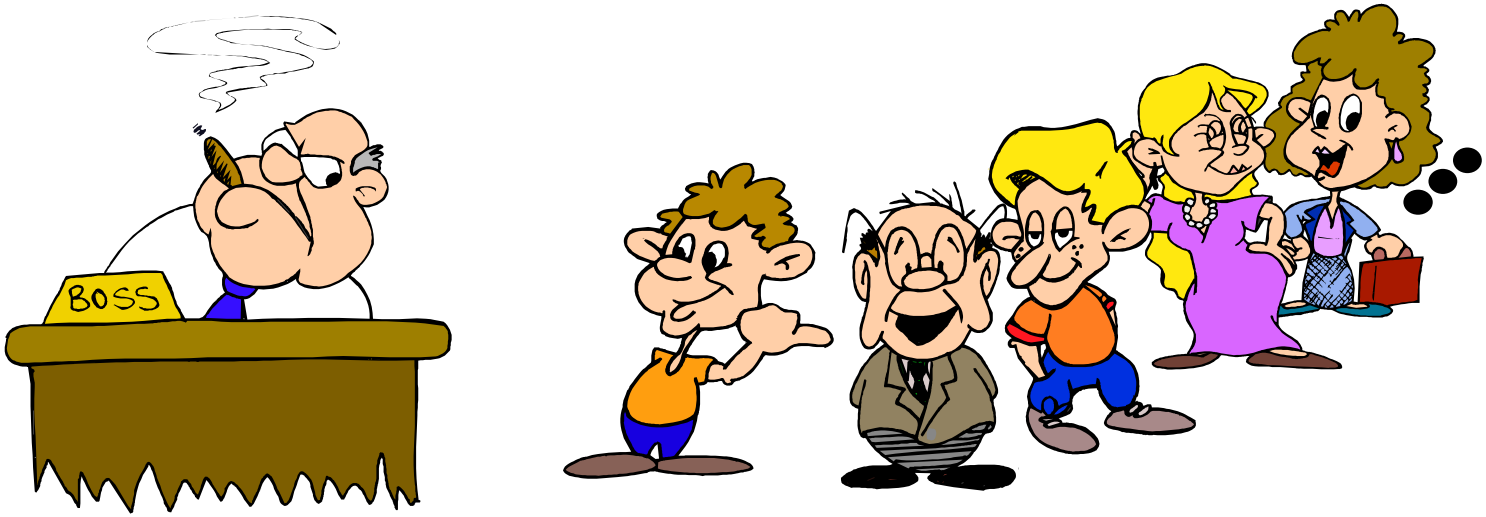
Computers and Intractability: A Guide to the Theory of NP-Completeness
(Series of Books in the Mathematical Sciences)
by Michael R. Garey and David S. Johnson

340 pages
W. H. Freeman publishers
(January 15, 1979)
ISBN-10: 0716710455
ISBN-13: 978-0716710455



# Dealing with hard problems

Aiming at solving some problems is still doomed to take too long.




I couldn't find an efficient algorithm,
but neither could all these other smart people.

# Another consequence

If you find an algorithm you can code on an existing computer able to solve any problem known to be NP complete then you are entitled to win one million dollar.

Another way to win the same million is to prove that such an algorithm does not exist!

Let us look into this matters from the perspective of a well-known computer game…

# All problems in NP reduce to mine sweeper!


http://web.mat.bham.ac.uk/R.W.Kaye/minesw/ordmsw.htm

# Question

Show that the halting problem is NP-hard by showing that 3SAT is reducible to it!

# Towers of Hanoi

**Goal:** transfer all $n$ disks from peg A to peg C

Rules:

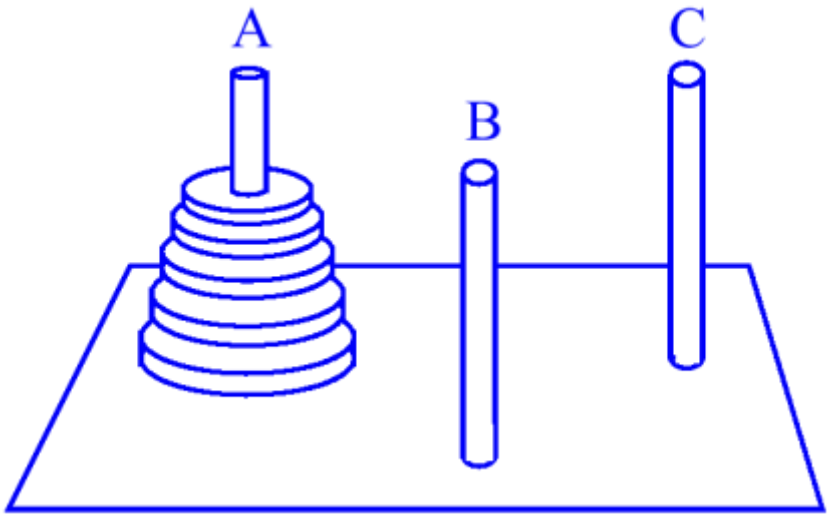
- move one disk at a time
- never place larger disk above smaller one

Recursive solution:

- transfer $n - 1$ disks from A to B
- move largest disk from A to C
- transfer $n - 1$ disks from B to C

Total number of moves:

- $T(n) = 2T(n - 1) + 1$



# Towers of Hanoi/2

Recurrence:

$T(n) = 2\,T(n-1) + 1$

$T(1) = 1$

Solution by repeated substitution:

$T(n) = 2\,(2\,T(n-2) + 1) + 1 =$

$= 4\,T(n-2) + 2 + 1 =$

$= 4\,(2\,T(n-3) + 1) + 2 + 1 =$

$= 8\,T(n-3) + 4 + 2 + 1 = \ldots$

$= 2^i\,T(n-i) + 2^{i-1} + 2^{i-2} + \ldots + 2^1 + 2^0$

The expansion stops with $n - i = 1$

$T(n) = 2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 2^1 + 2^0$

# Towers of Hanoi/3

This is a **geometric series** so that we have

$T(n) = 2^n - 1 = O(2^n)$

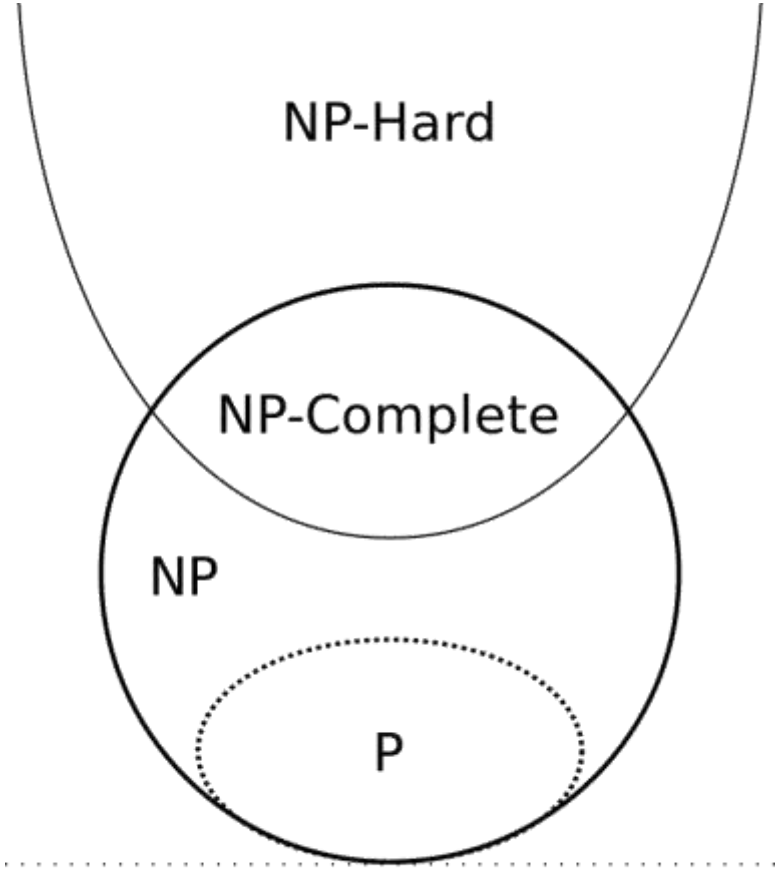
The running time of this algorithm is **exponential** ($k^n$).

***Is provable a lower bound!***

Good or bad news?

- the Tibetan priests were confronted with a tower problem of 64 rings...
- Assuming the priests move **one ring per second**, it would take ~**585 billion years** to complete the process!

# Where is the Towers of Hanoi problem?




$$P \neq NP$$

# Answer: is outside that picture!

Is it in NP?

Is it NP-hard?

Expensive computation is not the same as hard!!!

# Back to P

A problem is in P if it can be decided in polynomial time by a deterministic Turing machine.

*Another way to read the above definition:*

A problem is in P if a solution can either be found or proven non-existent in polynomial time by a deterministic algorithm in a regular computer.

# EULER is in P

The decision ‘is this graph Eulerian’ can be taken in polynomial time by testing if it is connected and computing the degree of each vertex and counting the number of vertices with odd degree. This can be done in time proportional to the number of edges in the graph.

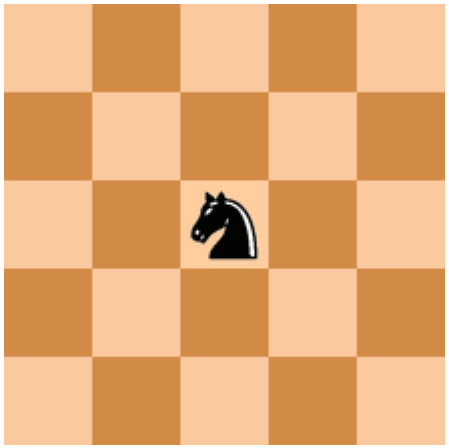
# Question

What is the complexity of deciding if a graph is connected?

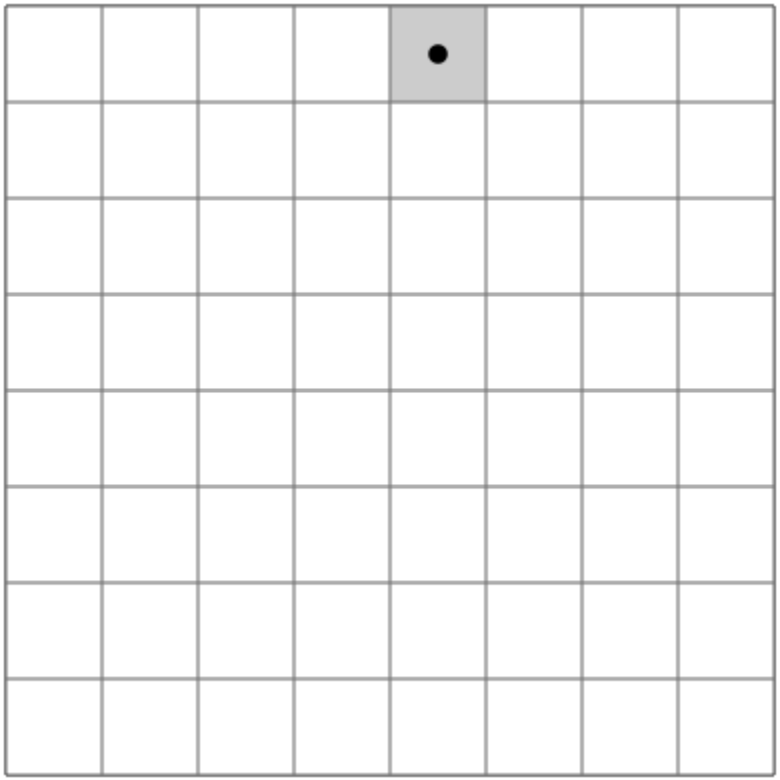
# Chess: knight's moves

# Is it possible to devise a knight's tour visiting every square once?

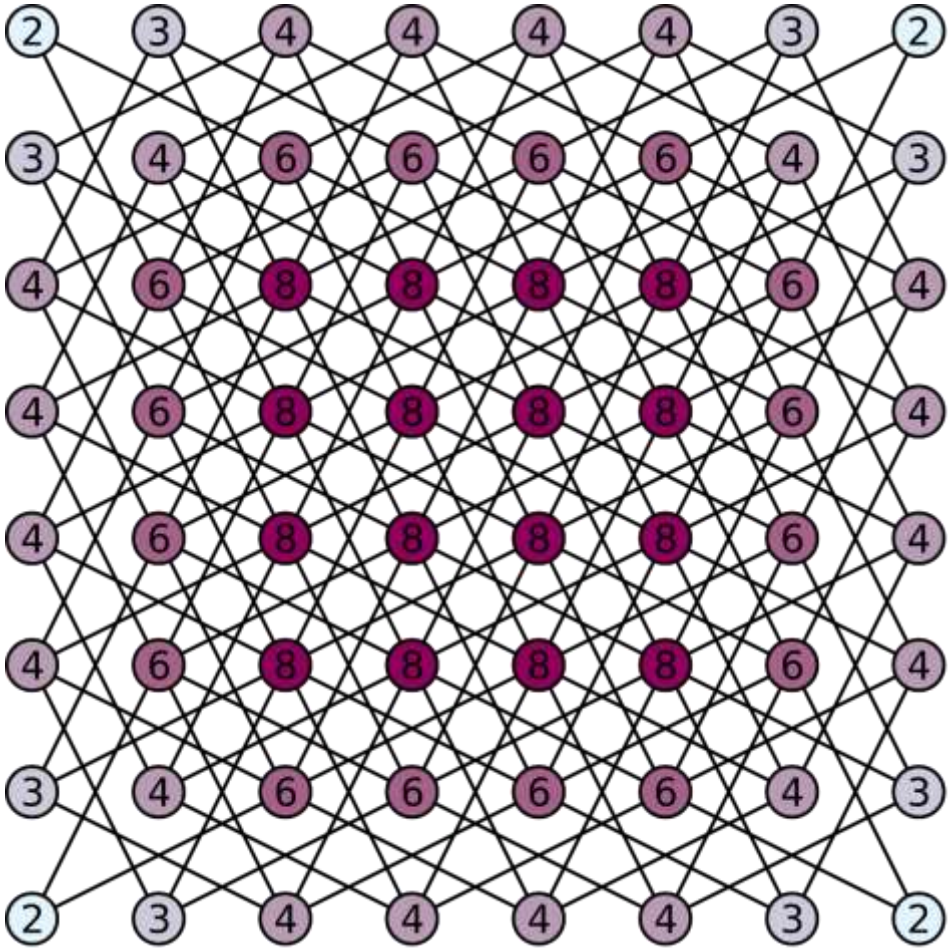
# Is it possible to devise a knight's tour visiting every square once?



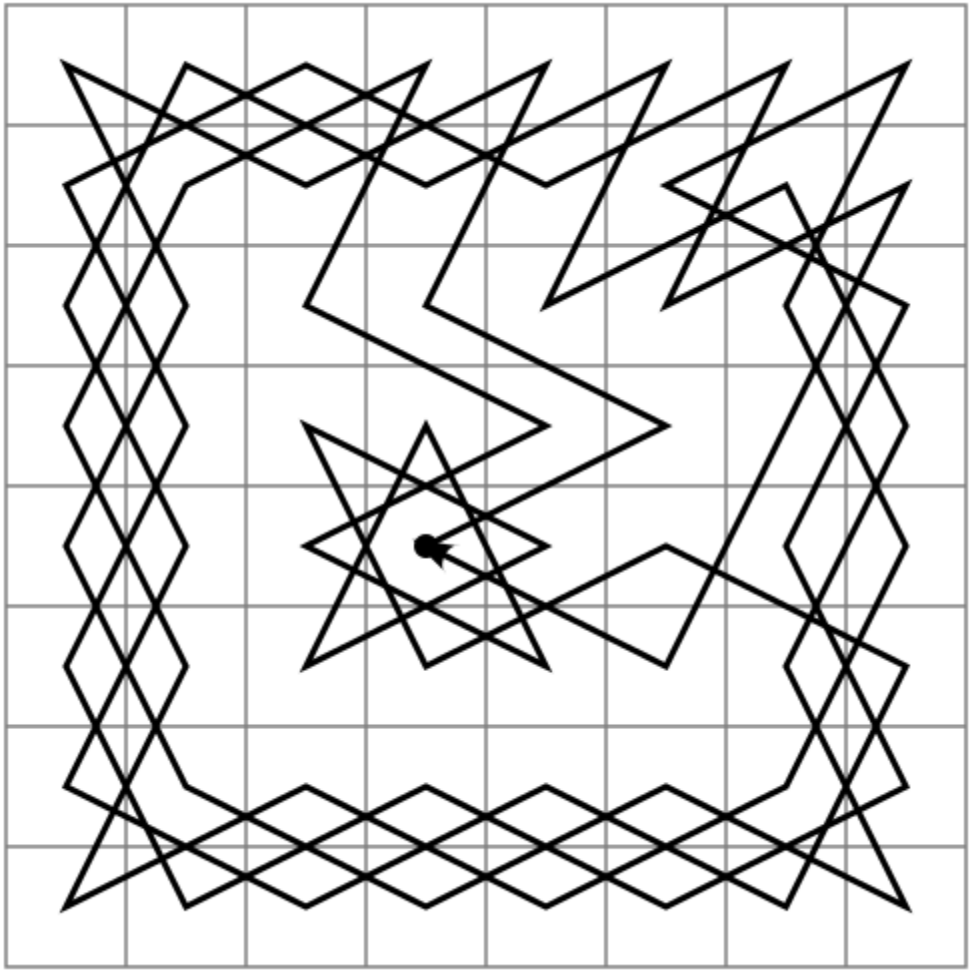
# What about a regular chessboard?



# Possible moves are edges of a graph



# Rudrata, X$^{th}$ century

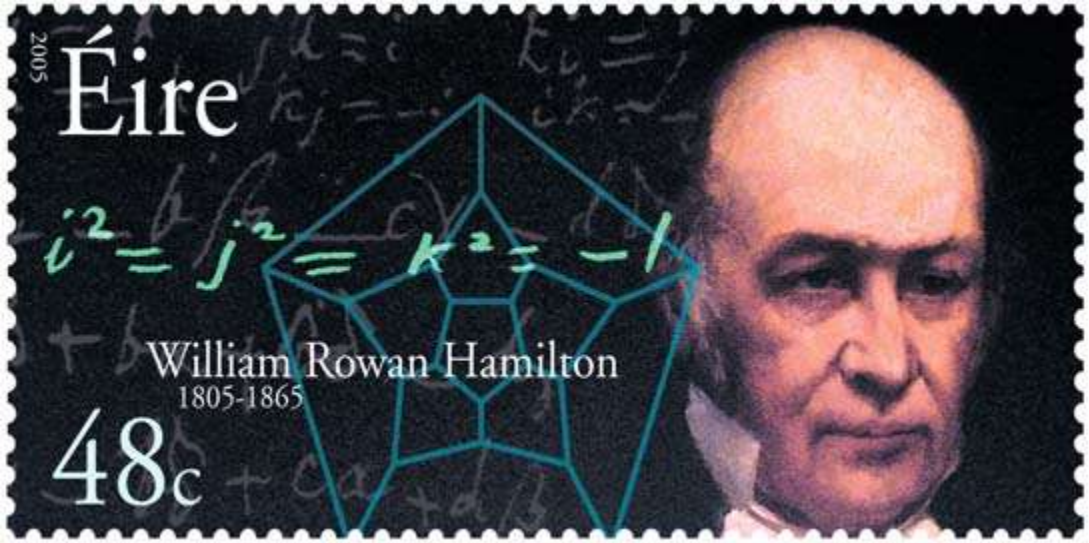


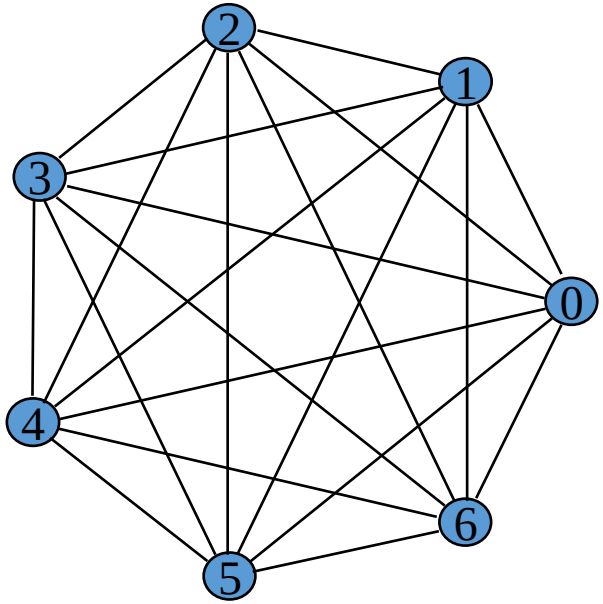
# Rudrata and Hamilthon

In the XIX$^{th}$ century Sir Rowan Hamilthon gained interest in the same problem as addressed by Rudrata almost thousand years before.

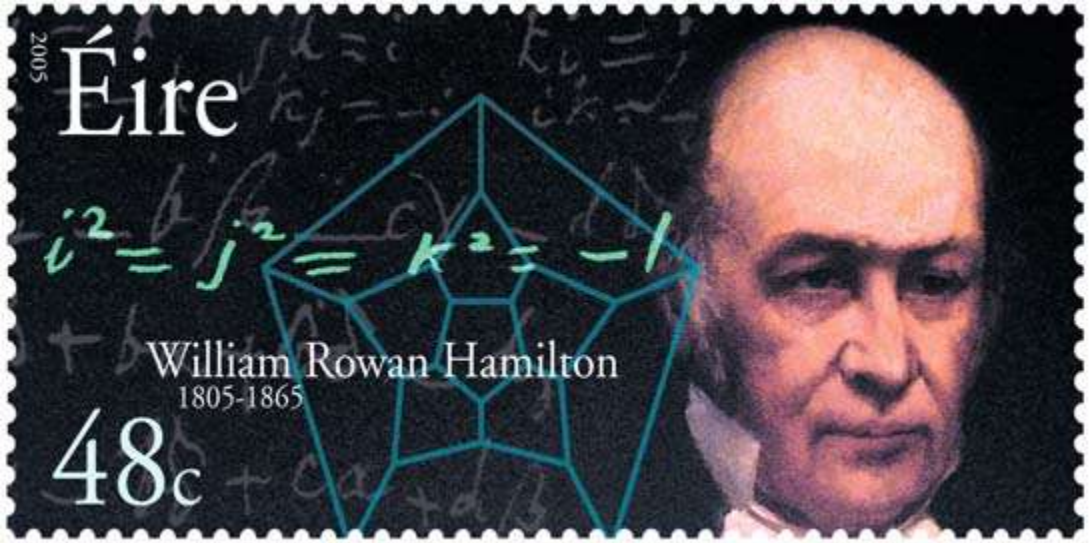
Nowadays you may find the problem named after Hamilthon more often than after Rudrata.

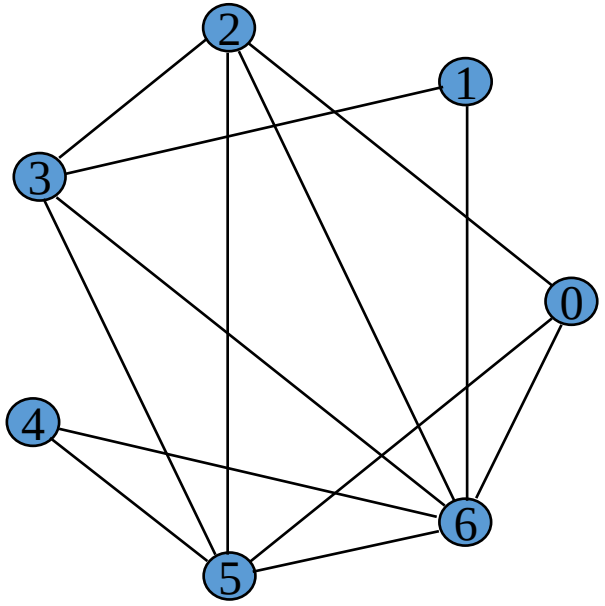
# Visit all nodes: the Hamiltonian journey



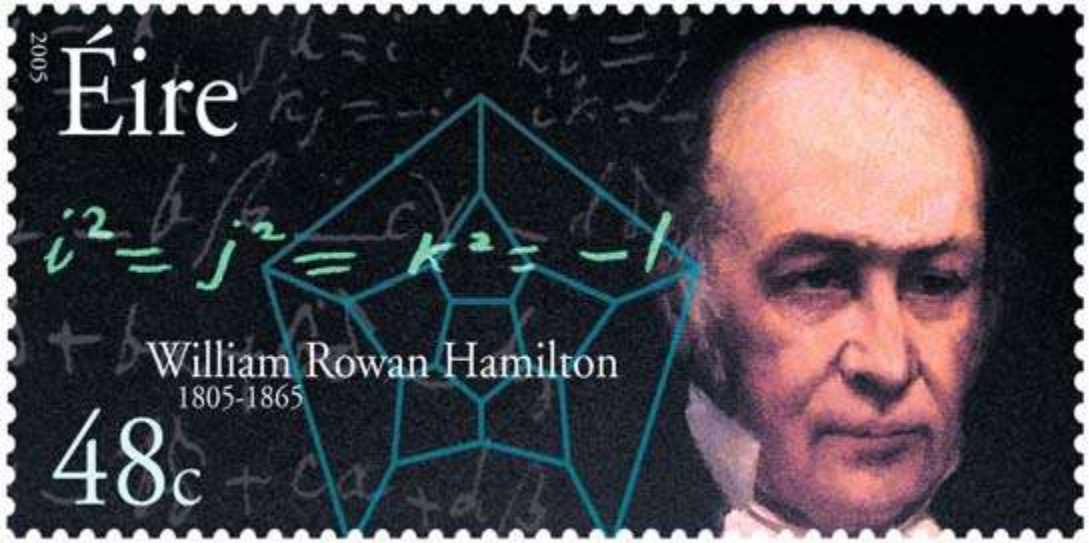



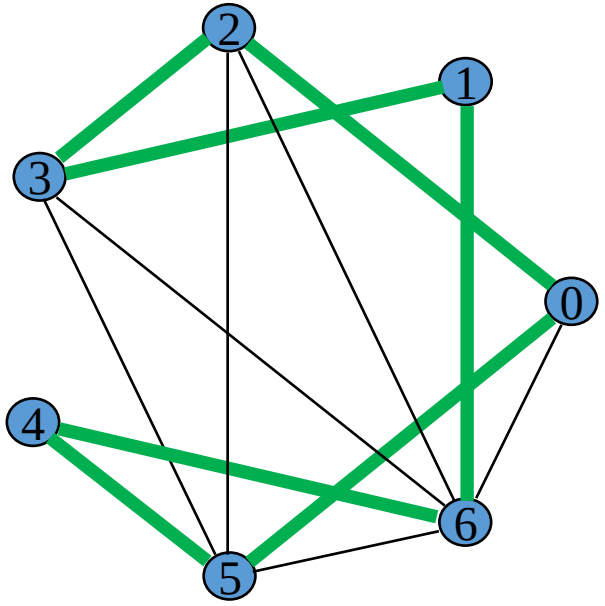
# Visit all nodes: the Hamiltonian journey



# Visit all nodes: the Hamiltonian journey



# a difficult decision

Much harder than Euler!

No efficient certificate known!

Suppose that one would be told the vertices in the correct sequence… then the ‘yes’ version of the certificate becomes easy!

We say that this problem is efficiently solved by a ‘non deterministic’ algorithm.

Such algorithms are unfortunately not implementable on ‘our’ computers…

# Optimization?

Suppose now that whether such a cycle exists is irrelevant (for instance when the graph is complete).

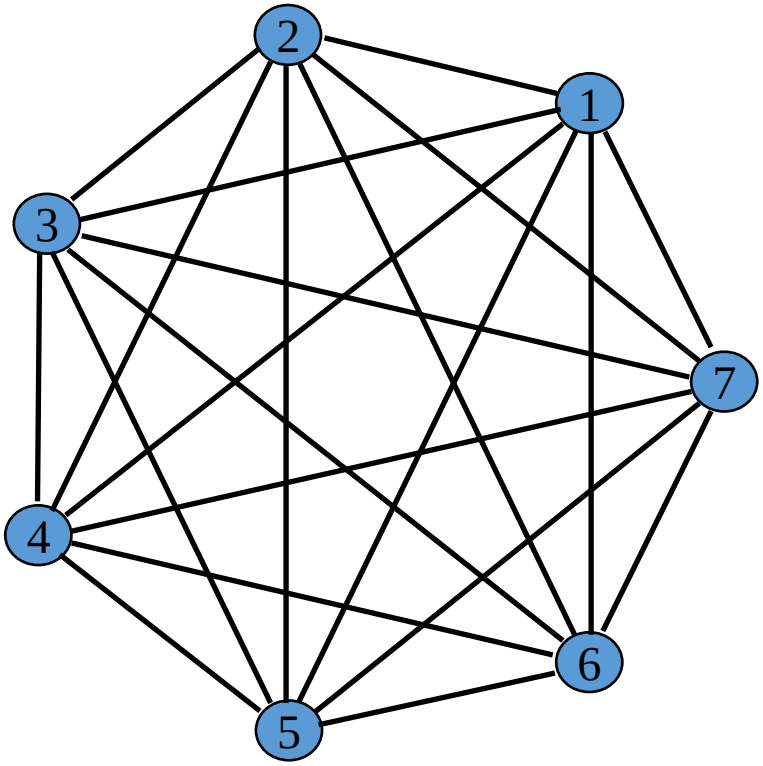
We need now to find the best (the shortest for instance) of such cycles.

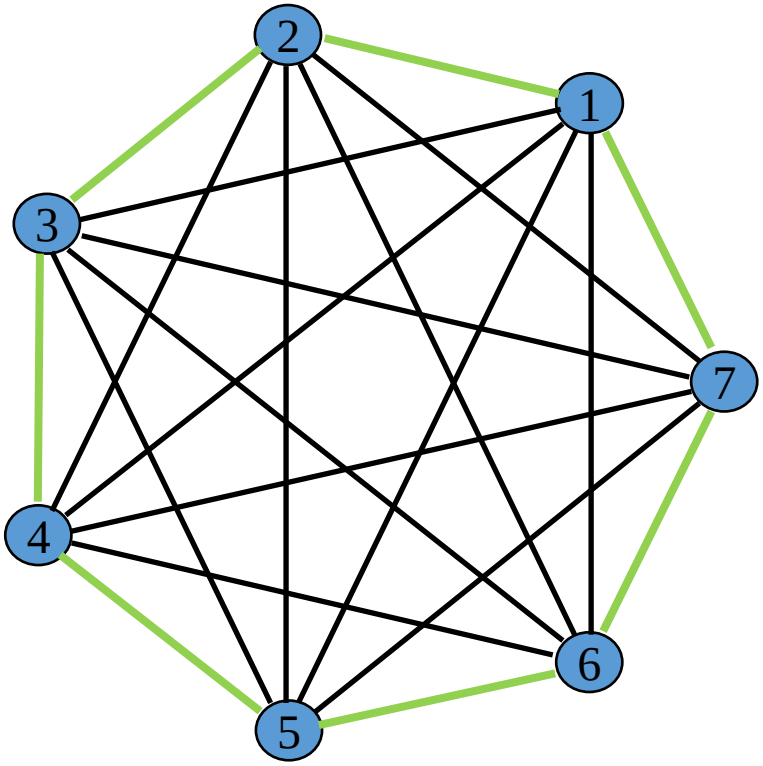
This is an optimization problem!

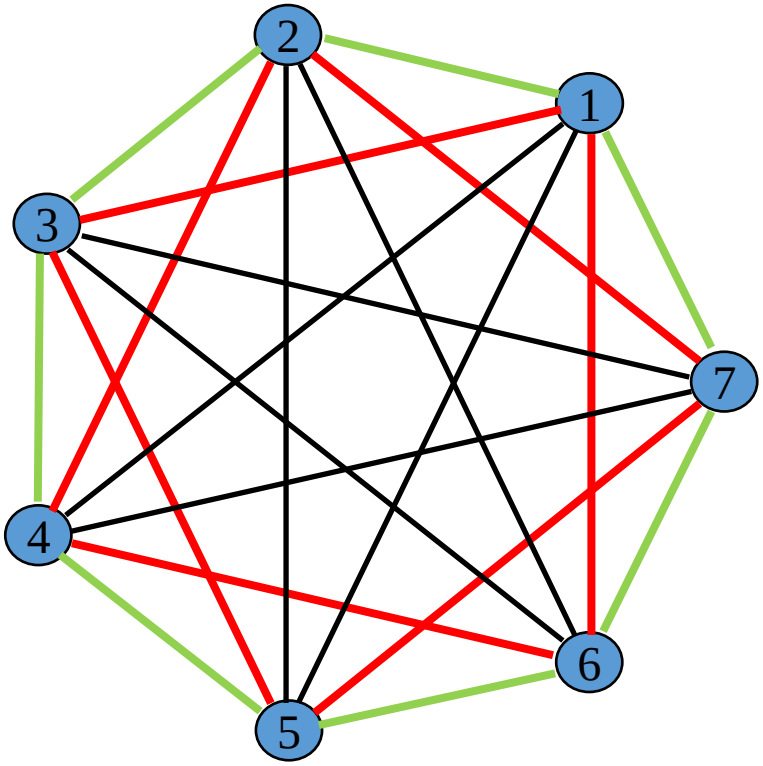
# Question

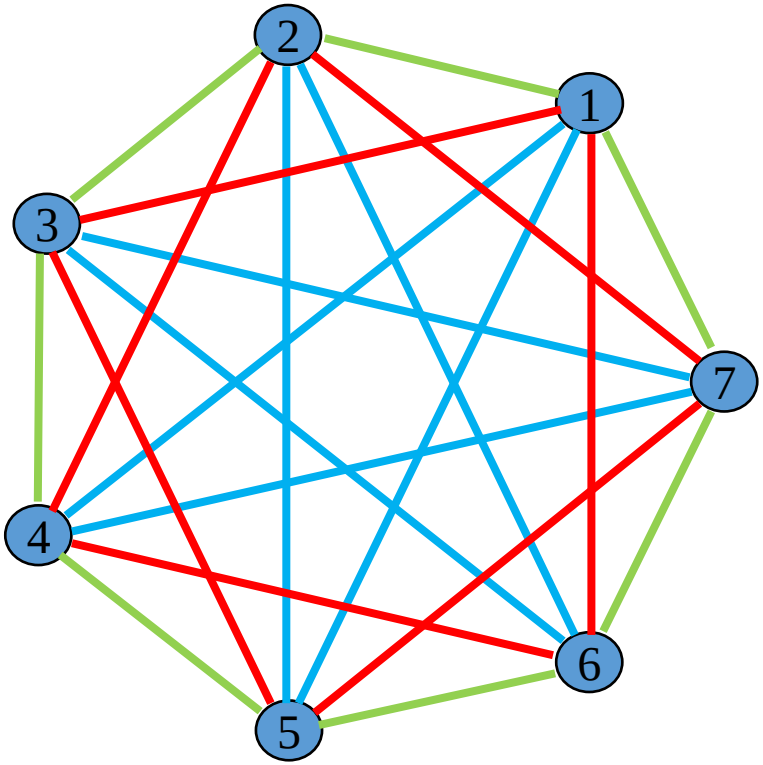
Show that if you could solve the optimization version then you could solve the decision problem!

You were just told that the decision problem is difficult… what does that mean for the difficulty of the optimization problem?


2
1
3
7
4
6
5


2
1
3
7
4
6
5


2
1
3
7
4
6
5


2
1
3
7
4
6
5

# Take 24978 cities in Sweden…

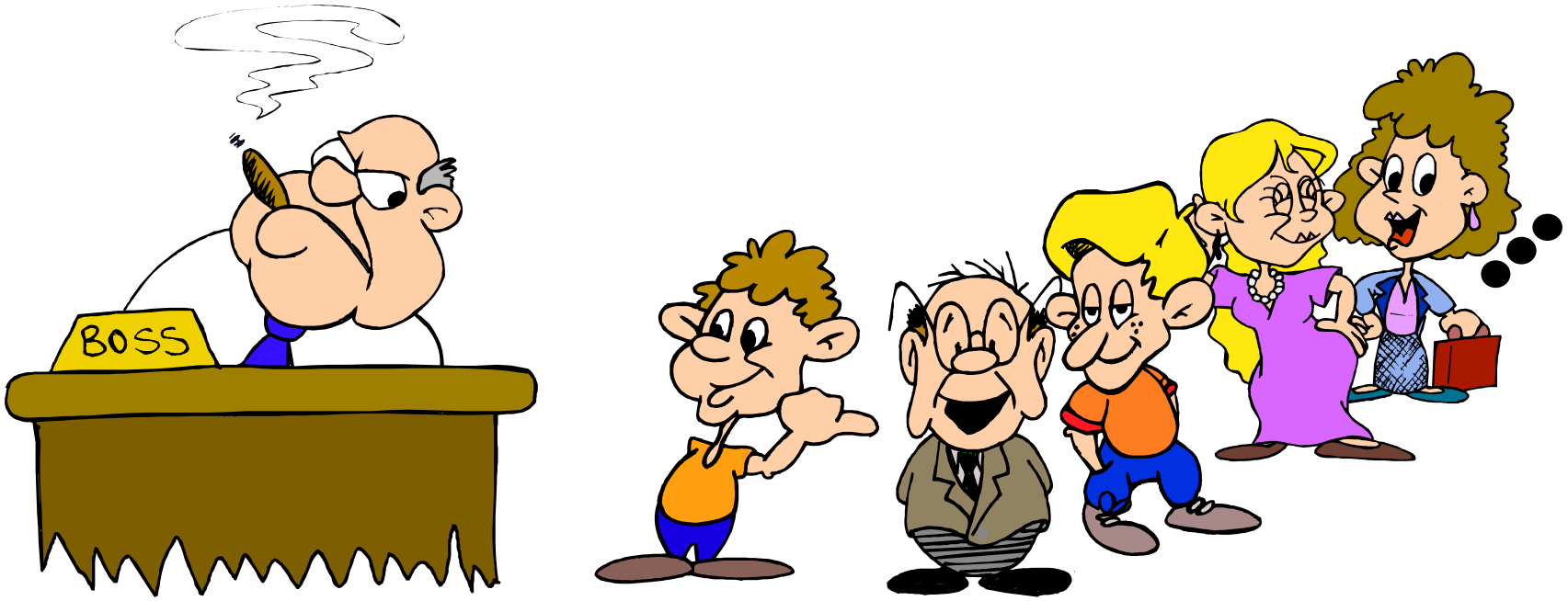
Give the distances among them to the best experts in the world
Add 96 dual processor computers
Wait for about one year (about 92 years of sequential computer power)

# Take 24978 cities in Sweden…

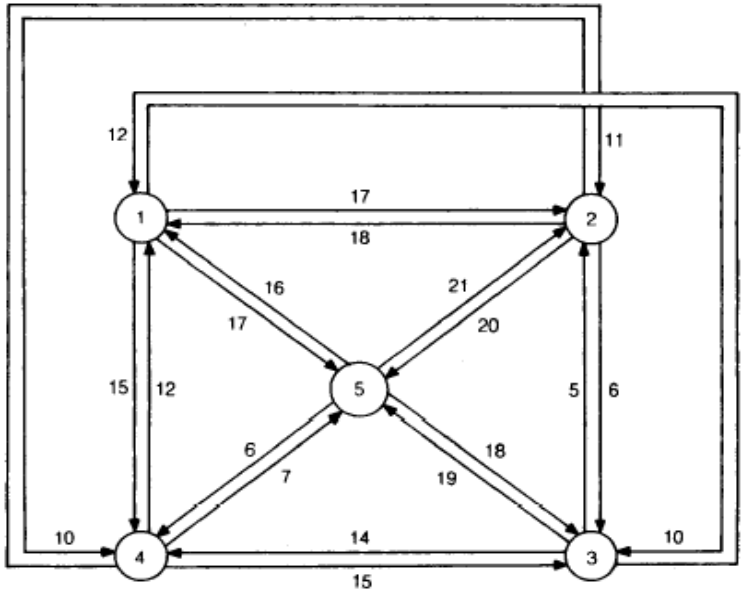
Give the distances among them to the best experts in the world
Add 96 dual processor computers
Wait for about one year (about 92 years of sequential computer power)

# I could not find an efficient algorithm…

# But so far no clever person in the world did!



# Travelling Salesman Problem




$$\mathbf{A} = \begin{bmatrix} M & 17 & 10 & 15 & 17 \\ 18 & M & 6 & 10 & 20 \\ 12 & 5 & M & 14 & 19 \\ 12 & 11 & 15 & M & 7 \\ 16 & 21 & 18 & 6 & M \end{bmatrix}$$

# TSP: mathematical program

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{1}$$

$$\text{st:} \sum_{i=1}^{n} x_{ij} = 1 \quad \forall j \in N \tag{2}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \quad \forall i \in N \tag{3}$$

$$\sum_{i \in S, j \notin S} x_{ij} \geq 1 \quad \forall S \subset N \tag{4}$$

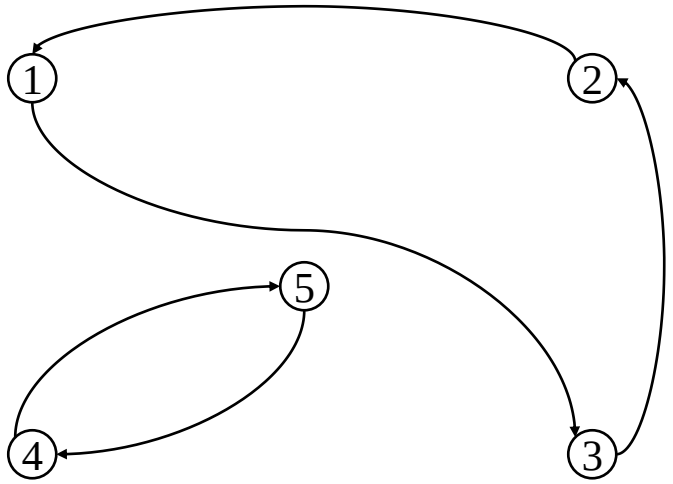
$$x_{ij} \in \{0, 1\} \tag{5}$$

# The linear assignment solution

$$1 \to 3, \;\; 2 \to 1, \;\; 3 \to 2, \;\; 4 \to 5, \;\; 5 \to 4$$

The weight $w(\mathbf{A}) = 10 + 18 + 5 + 6 + 7 = 46$

$$\mathbf{A} = \begin{bmatrix} M & 17 & \boxed{10} & 15 & 17 \\ \boxed{18} & M & 6 & 10 & 20 \\ 12 & \boxed{5} & M & 14 & 19 \\ 12 & 11 & 15 & M & \boxed{7} \\ 16 & 21 & 18 & \boxed{6} & M \end{bmatrix}$$




$$S_1 : \;\; 1 \to 3 \to 2 \to 1$$
$$S_2 : \;\; 4 \to 5 \to 4.$$

# Eliminate sub-cycles

$$\mathbf{A}(4) = \begin{bmatrix} M & 17 & 10 & 15 & \boxed{17} \\ 18 & M & \boxed{6} & 10 & 20 \\ 12 & \boxed{5} & M & 14 & 19 \\ \boxed{12} & 11 & 15 & M & \mathbf{M} \\ 16 & 21 & 18 & \boxed{6} & M \end{bmatrix}, \quad \mathbf{A}(5) = \begin{bmatrix} M & 17 & \boxed{10} & 15 & 17 \\ 18 & M & 6 & \boxed{10} & 20 \\ 12 & \boxed{5} & M & 14 & 19 \\ 12 & 11 & 15 & M & \boxed{7} \\ \boxed{16} & 21 & 18 & \mathbf{M} & M \end{bmatrix}.$$

$1 \rightarrow 5 \rightarrow 4 \rightarrow 1$

$2 \rightarrow 3 \rightarrow 2$

$w(\mathbf{A}(4)) = 46(NH)$ ?

$1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 1$

$w(\mathbf{A}(5)) = 48(H)$ !

# Repeat from A(4)

$$\mathbf{A}(4,2) = \begin{bmatrix} M & 17 & \boxed{10} & 15 & 17 \\ 18 & M & \mathbf{M} & 10 & \boxed{20} \\ 12 & \boxed{5} & M & 14 & 19 \\ \boxed{12} & 11 & 15 & M & M \\ 16 & 21 & 18 & \boxed{6} & M \end{bmatrix}, \quad \mathbf{A}(4,3) = \begin{bmatrix} M & 17 & 10 & 15 & \boxed{17} \\ 18 & M & \boxed{6} & 10 & 20 \\ \boxed{12} & \mathbf{M} & M & 14 & 19 \\ 12 & \boxed{11} & 15 & M & M \\ 16 & 21 & 18 & \boxed{6} & M \end{bmatrix}.$$

Hamiltonian cycle:

$1 \to 3 \to 2 \to 5 \to 4 \to 1$

$w(\mathbf{A}(4,2)) = 53(H)$

!

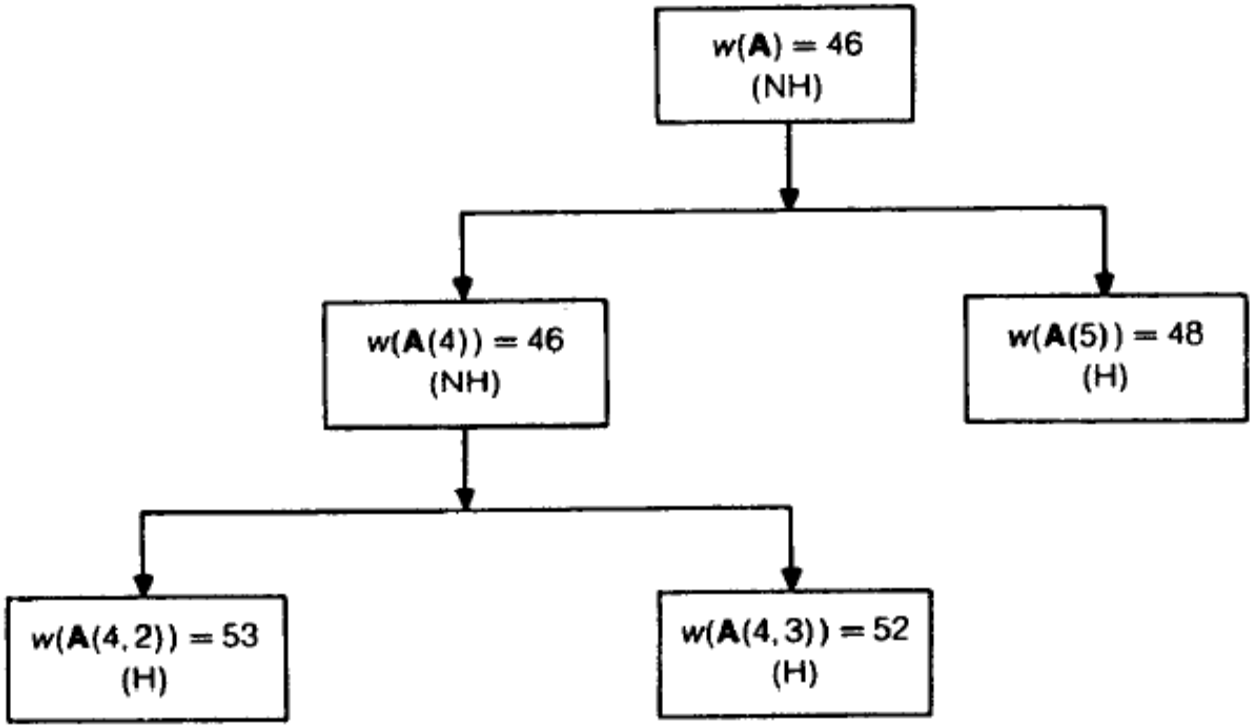
Hamiltonian cycle:

$1 \to 5 \to 4 \to 2 \to 3 \to 1$

$w(\mathbf{A}(4,3)) = 52(H)$

!

# Branch & Bound
# (first usage Little et al, 1963)



# Question

Does Branch & Bound prove that the Linear Assignment is as difficult as the Traveling Salesman Problem?

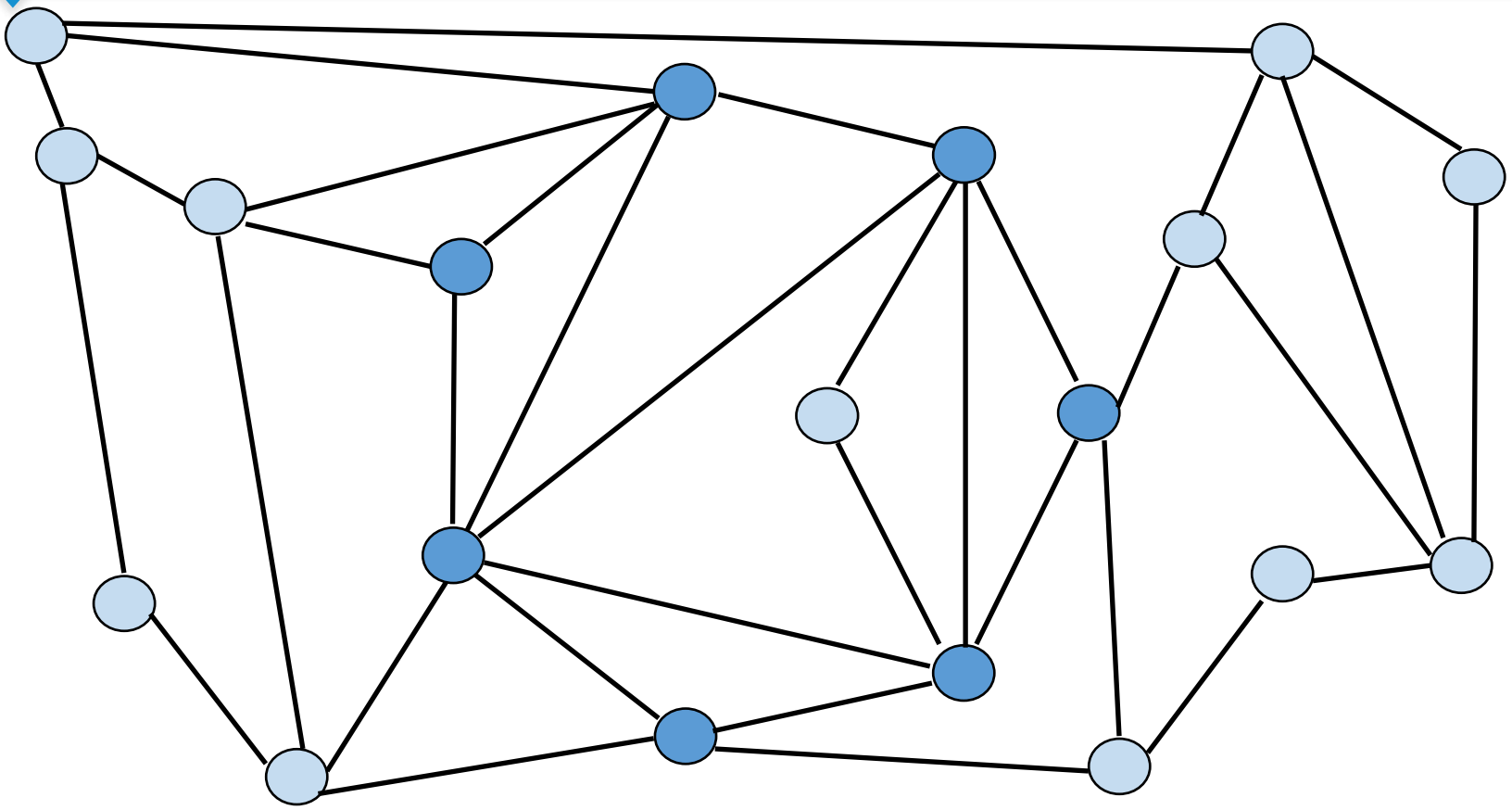
Or, same question in other words, does TSP efficiently reduce to LAP?

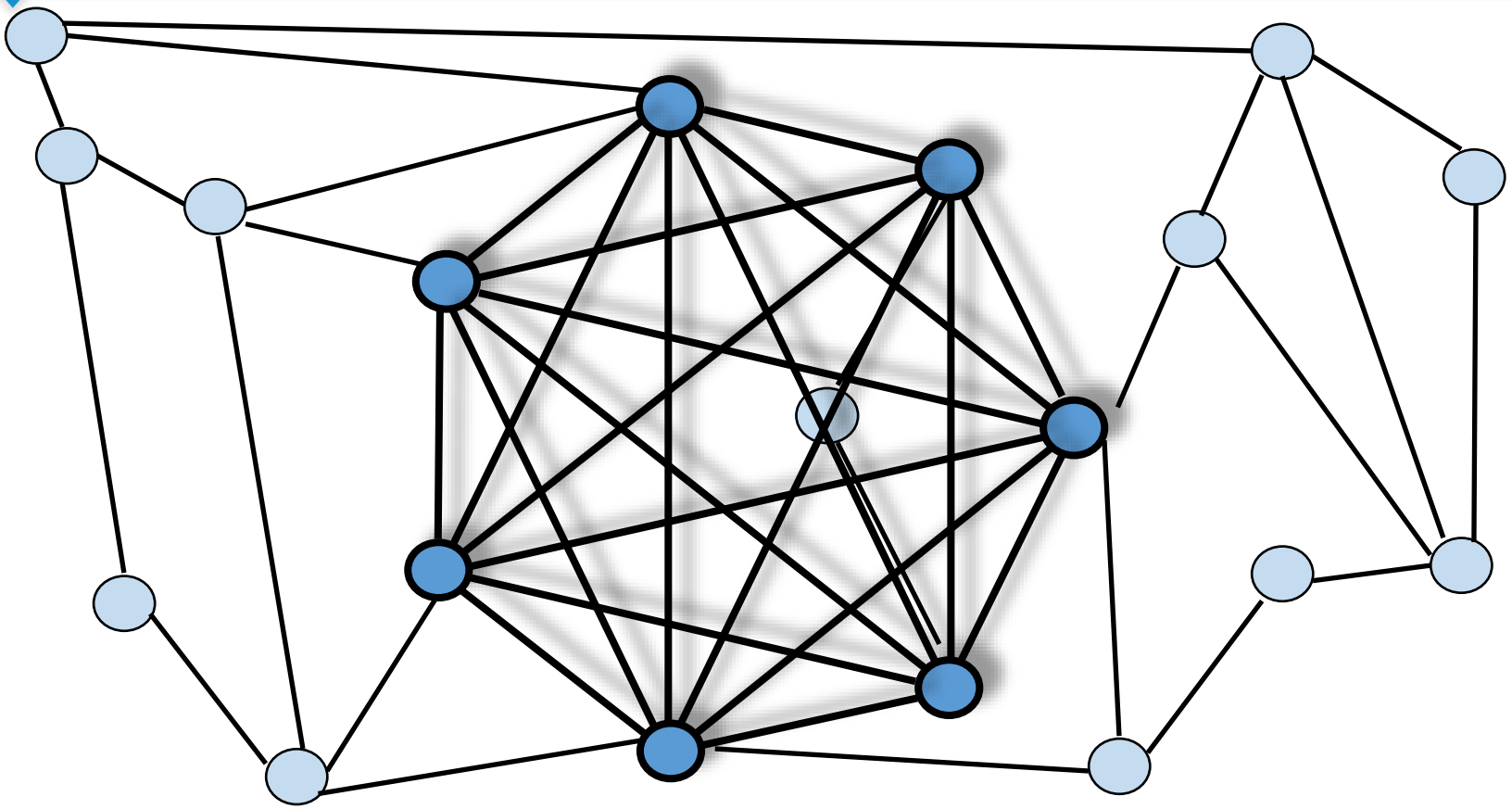
# note

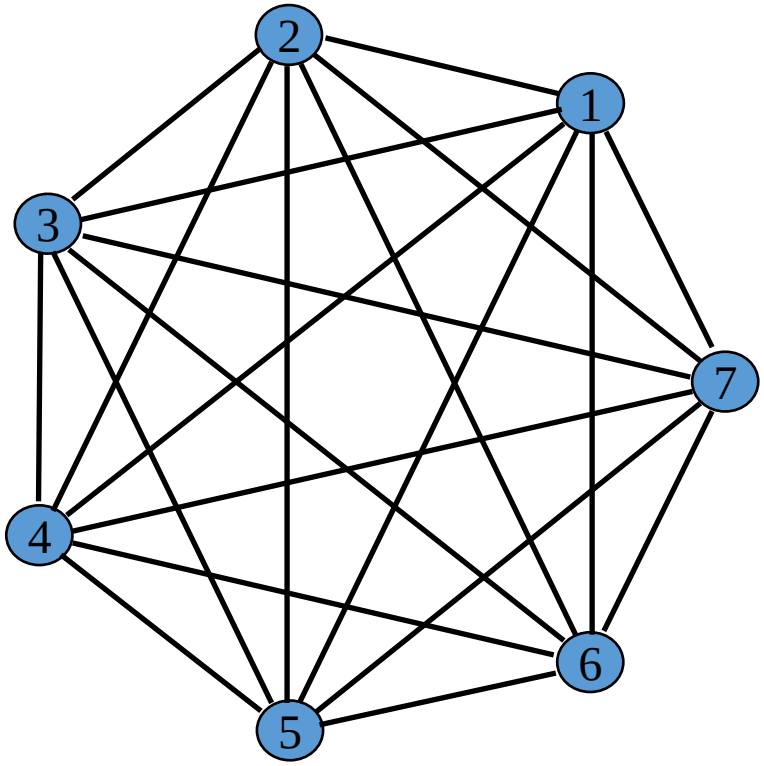
The first two assignments of this week's tutorials use TSP and Rudrata (or Hamilthon) Cycle to illustrate the relation between **search**, **decision** and **optimization**.

# Disclaimer on efficiency…

Suppose that a problem can efficiently be solved: is that always practical?





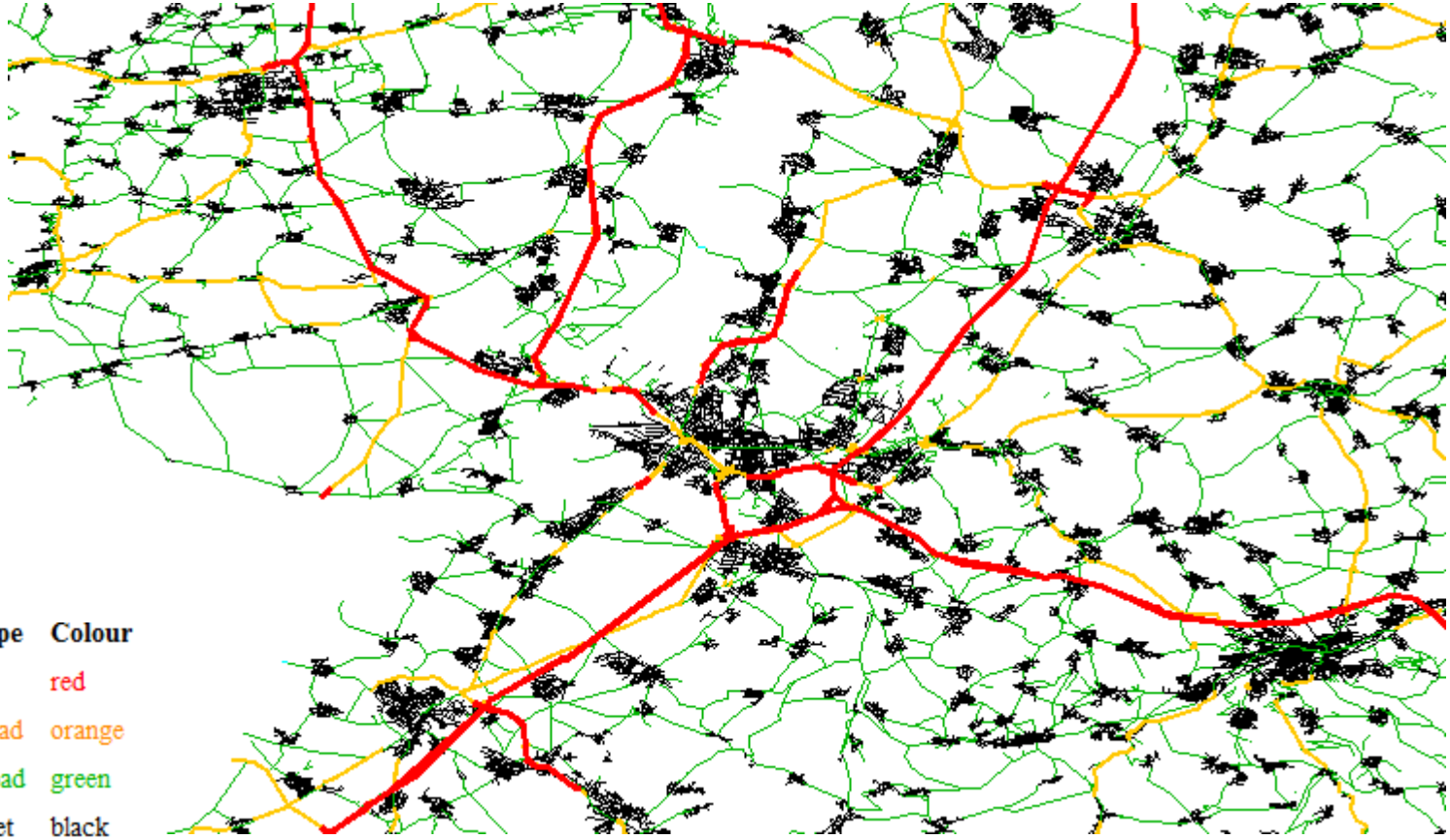

2
1
3
7
4
6
5

| n | n x n | Path per path | All from or all to |
|---|---|---|---|
| 10 | 100 | 0:01:40 | 0:00:20 |
| 50 | 2500 | 0:41:40 | 0:01:40 |
| 100 | 10000 | 2:46:40 | 0:03:20 |
| 1000 | 1000000 | 11 days 13:46:40 | 0:33:20 |
| 5000 | 25000000 | 289 days 8:26:40 | 2:46:40 |

# All about implementations…

Careful considerations about the type of graphs and about the mathematics behind the algorithm of Dykstra have led to huge improvements!

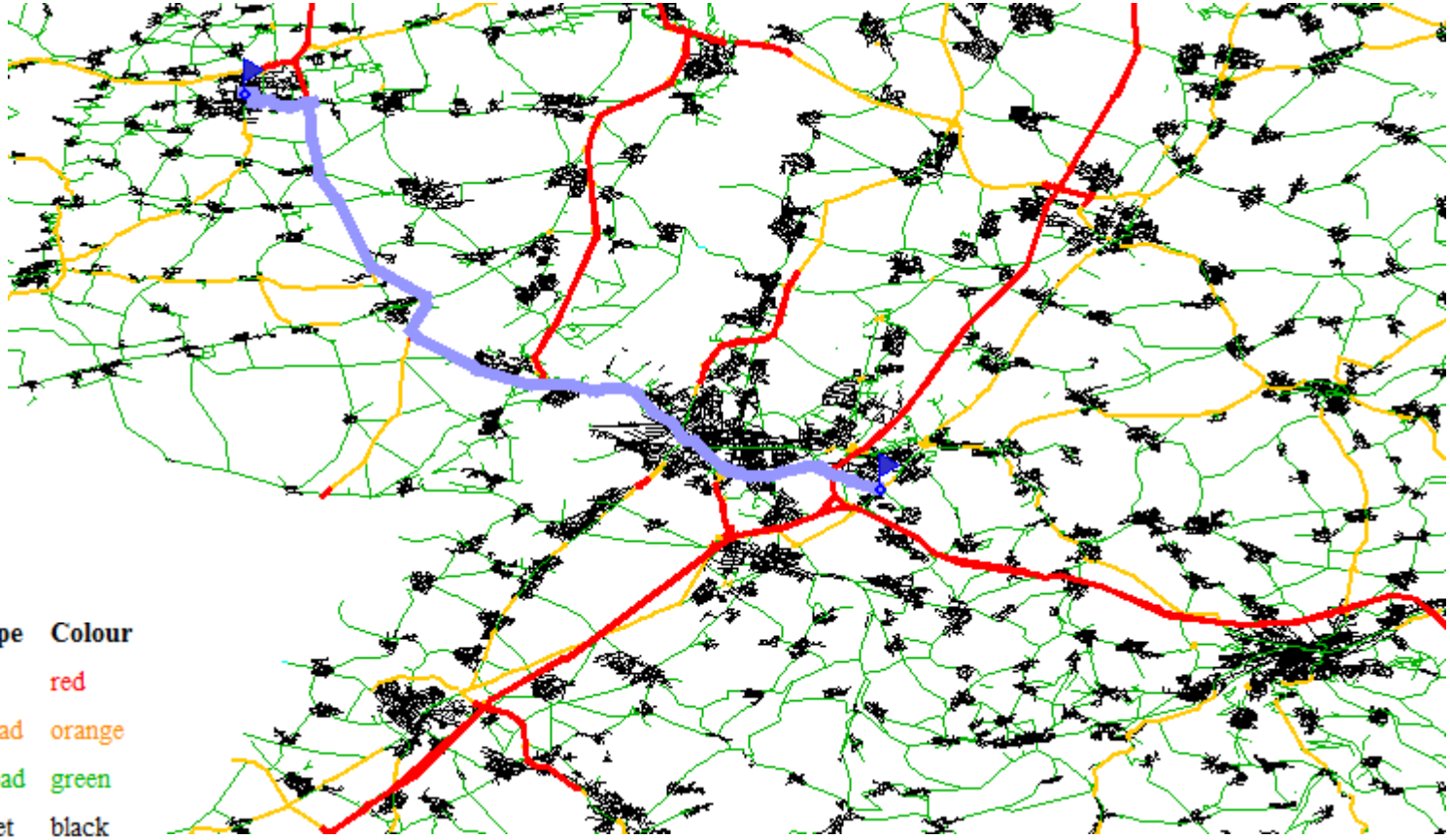
# We see roads as being of different types



| Road Type | Colour |
|---|---|
| motorway | red |
| national road | orange |
| regional road | green |
| urban street | black |
| cart-track | brown |
| ferry | cyan |

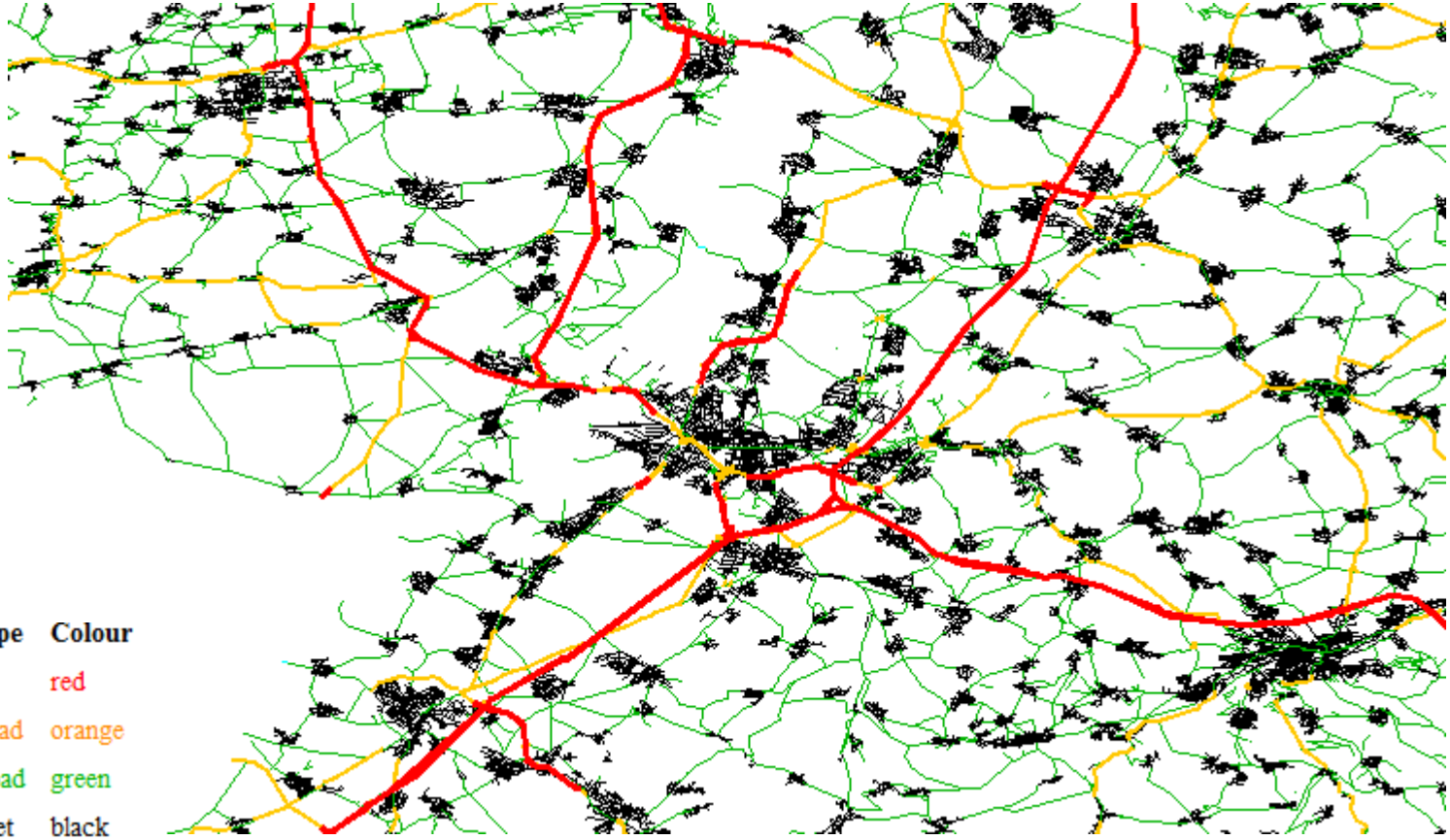
# For Dijkstra all arcs are of the same type

# Makes sense to take highways on long paths



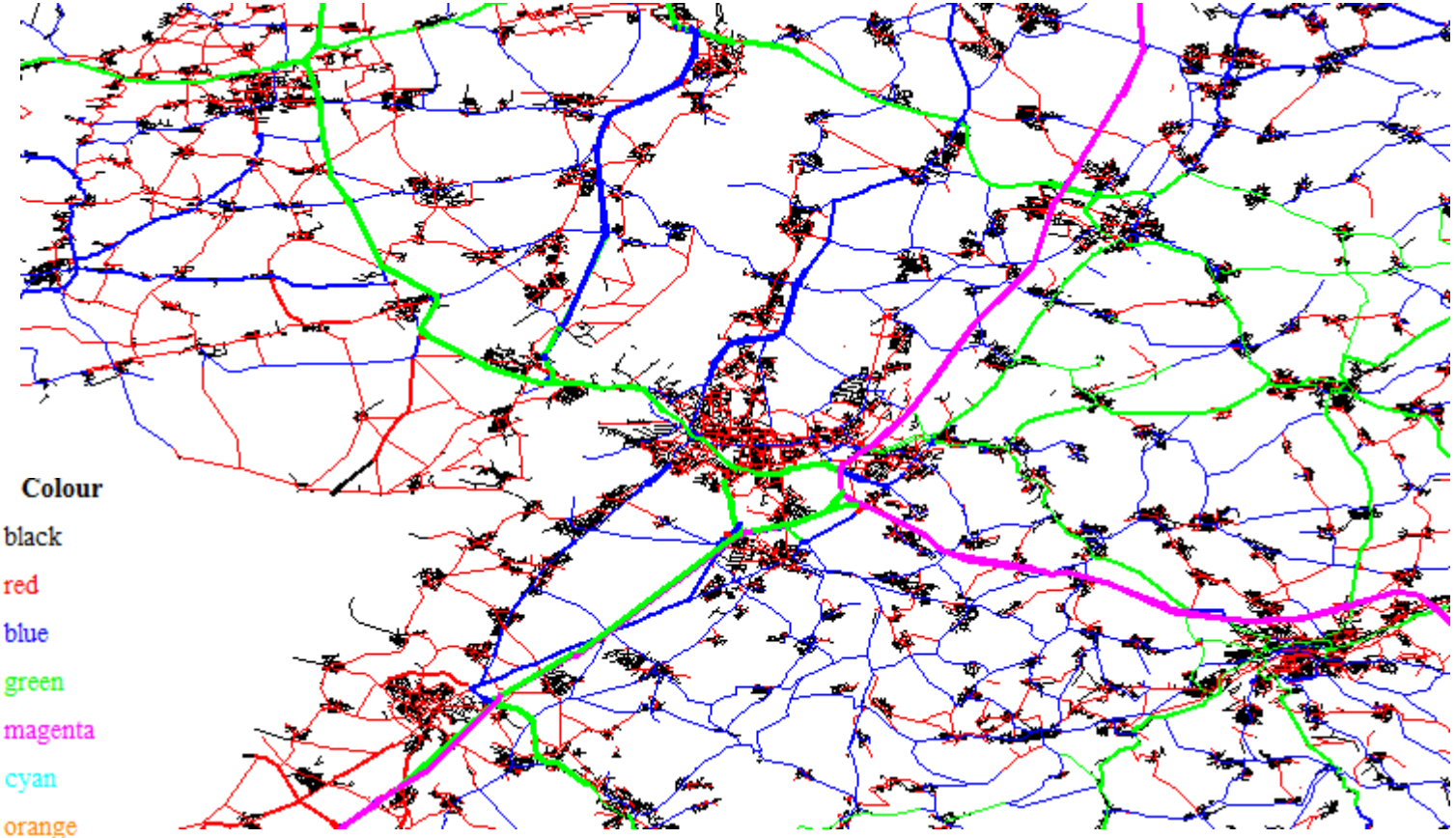
| Road Type | Colour |
|---|---|
| motorway | red |
| national road | orange |
| regional road | green |
| urban street | black |
| cart-track | brown |
| ferry | cyan |

# Administrative classification



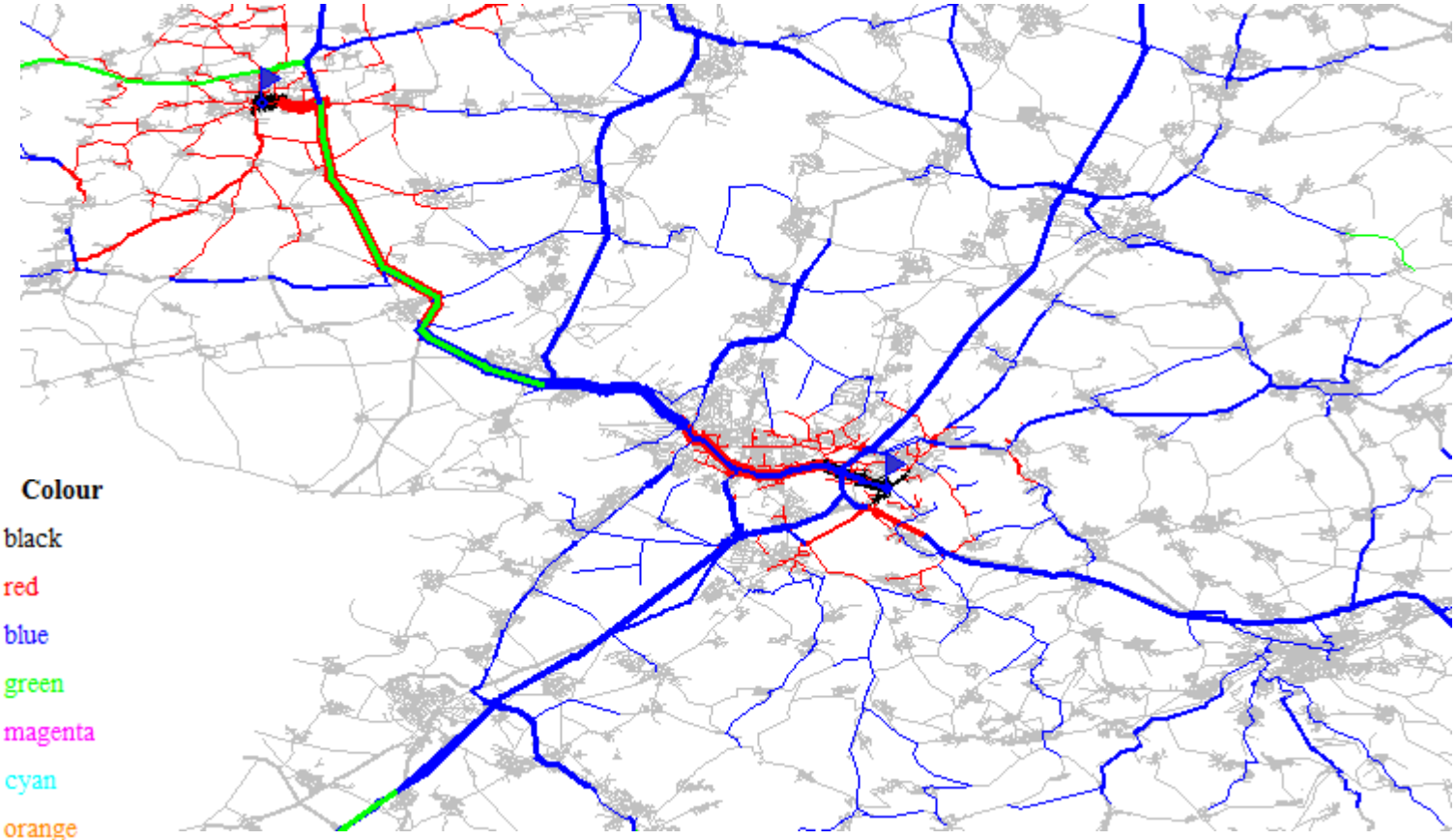
| Road Type | Colour |
|---|---|
| motorway | red |
| national road | orange |
| regional road | green |
| urban street | black |
| cart-track | brown |
| ferry | cyan |

# Rigorous classification



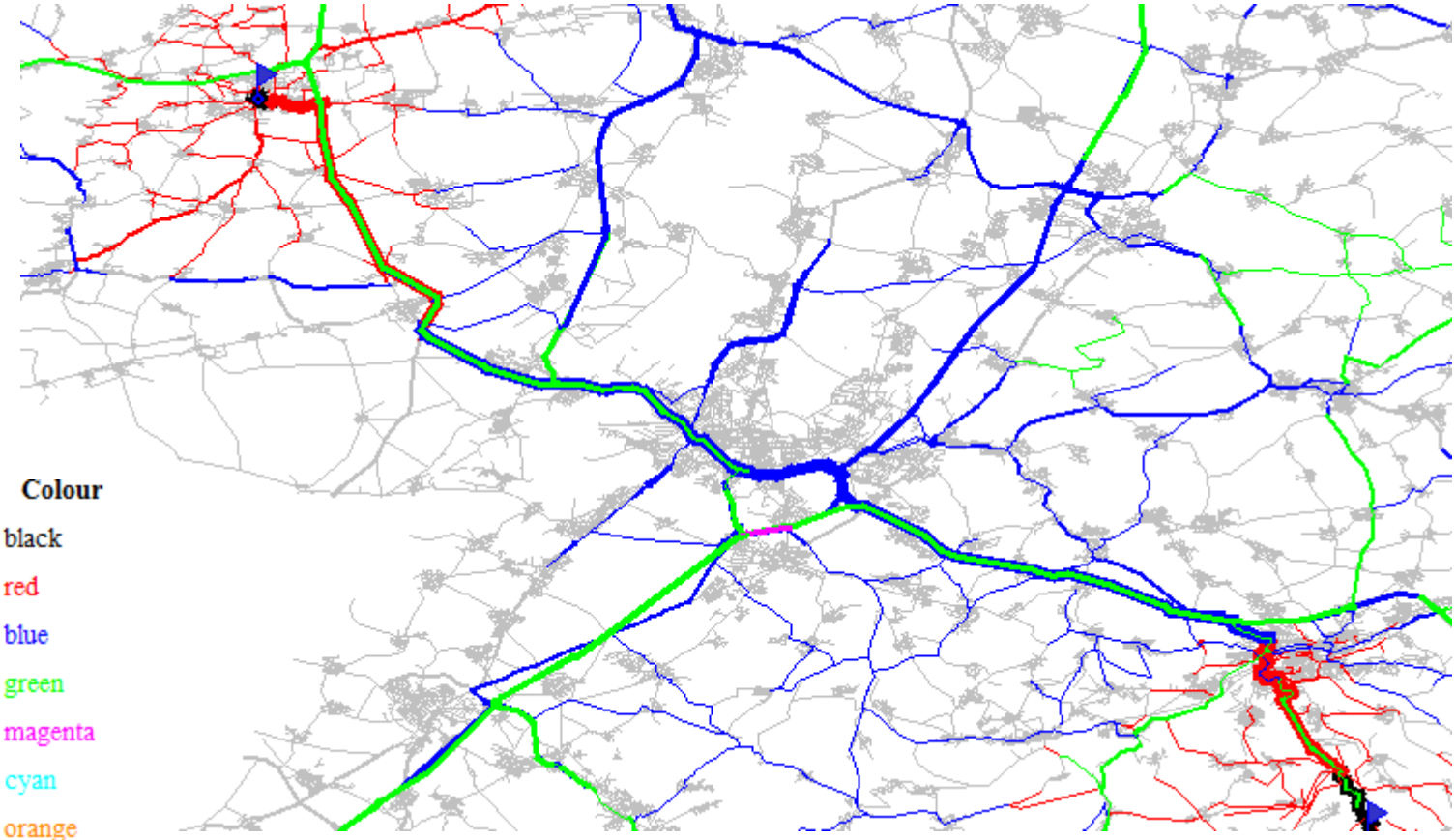
| Level | Colour |
|---|---|
| 0 | black |
| 1 | red |
| 2 | blue |
| 3 | green |
| 4 | magenta |
| 5 | cyan |
| 6 | orange |
| 7 | yellow |
| 8 | dark red |
| 9 | dark blue |
| 10 | dark green |

# Optimal queries become sparse!



| Level | Colour |
|---|---|
| 0 | black |
| 1 | red |
| 2 | blue |
| 3 | green |
| 4 | magenta |
| 5 | cyan |
| 6 | orange |
| 7 | yellow |
| 8 | dark red |
| 9 | dark blue |
| 10 | dark green |

# Effort almost independent of the length

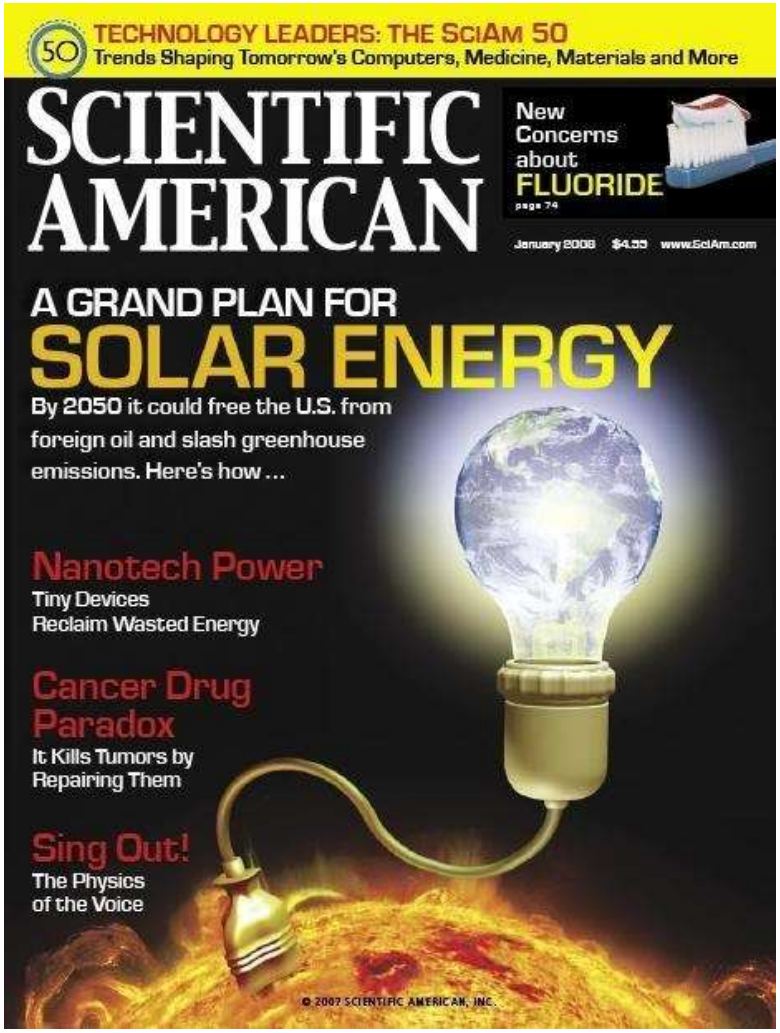


| Level | Colour |
|---|---|
| 0 | black |
| 1 | red |
| 2 | blue |
| 3 | green |
| 4 | magenta |
| 5 | cyan |
| 6 | orange |
| 7 | yellow |
| 8 | dark red |
| 9 | dark blue |
| 10 | dark green |

# Dominik Schultes (1980 - … )

| Size of the question | Conventional | HNR |
|---|---|---|
| From A to B | 2 to 30 seconds | 0.01 seconds |
| 100x100 | 10+ minutes | 1 second |
| 1000x1000 | 2+ hours | 20 seconds |
| 5000x5000 | 20+ hours | 2 minutes |

# Above Google on SciAm50 in 2008




**The Fastest Way to Get There**

Novel ways of calculating routes and predicting traffic jams promise less time in the car

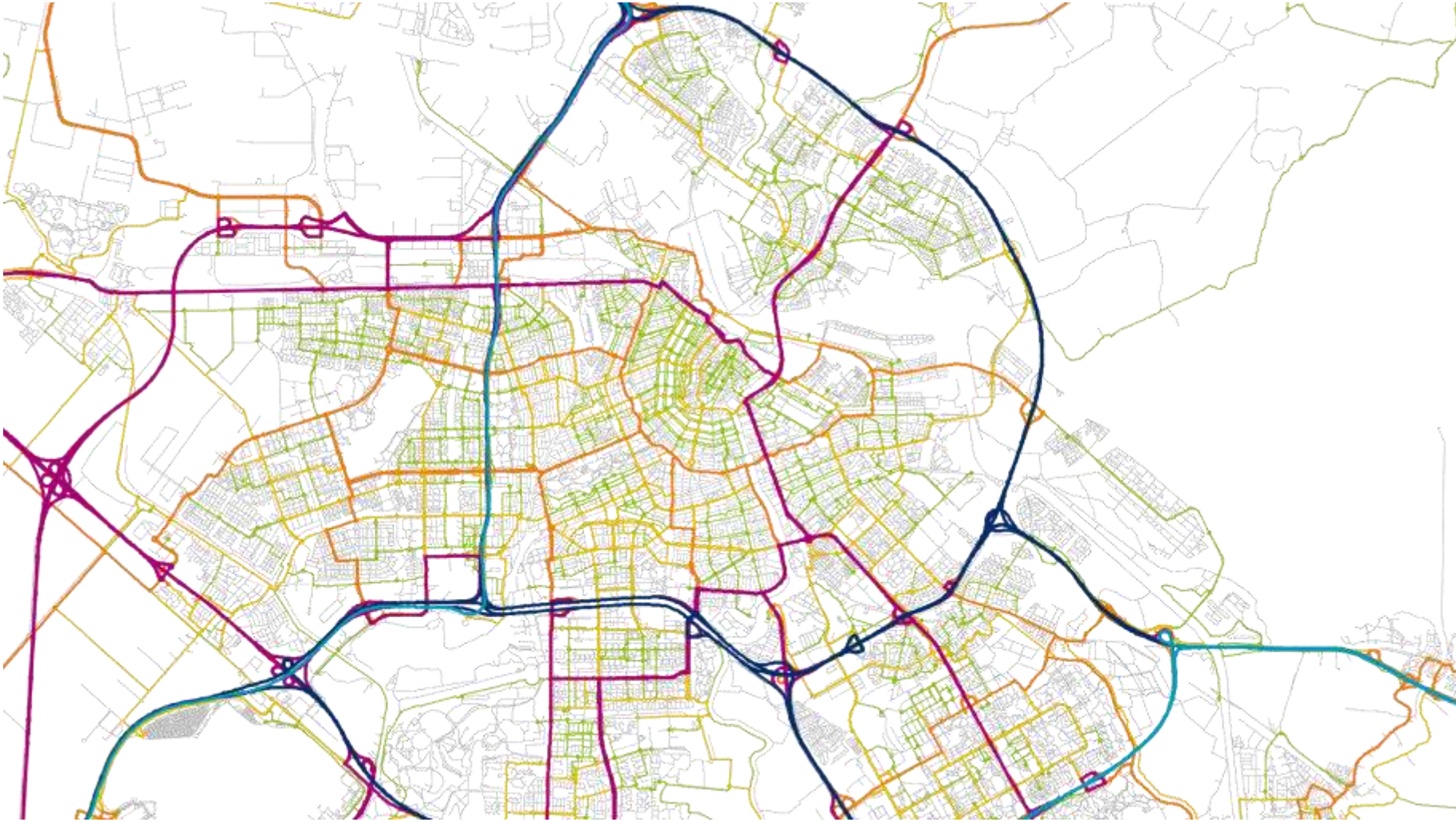
By Peter Sergo

Providing directions instantly online has until recently meant that navigational mapping programs, such as MapQuest and Google Maps, often simplify the problem by not considering every possible route to a destination. Scientists at the University of Karlsruhe in Germany have designed a computer application that can quickly calculate the most expedient of all possible driving routes without the need for excessive computation.

**Dominik Schultes**, one of the project's scientists, designed the program around a simple premise: driving somewhere usually requires crossing major intersections that are sparsely interconnected. Figuring the best route occurs by precomputing the connections between a starting point (or destination) and its nearest major intersections and between all locations where major routes cross each other's paths—so-called transit nodes.

# Amsterdam

# Amsterdam's HNR



# Amsterdam’s administrative

# LCS

Longest common subsequence

- x = "sariempiolcewe"
- y = "westigmupsalrte"

Solution: Proceed from the end of the strings and

- If $x_m = y_n$ append symbol to LCS($x_{m-1}$, $y_{n-1}$)
- If $x_m \neq y_n$
  - Skip last symbol from *x or*
  - last symbol from *y*
  - Decide which symbol to skip by comparing LCS($x_m$, $y_{n-1}$) and LCS($x_{m-1}$, $y_n$)

# LCS Implementation

```
int lcsRec(int i, int j) {
  if (i==0 || j==0) return 0;
  else if (x[i] == y[j])
    return lcsRec(i-1,j-1) + 1;
  else
    return max(lcsRec(i-1,j),lcsRec(i,j-1));
}
```

Recurrence for time complexity:

T(2n) = T(2n-2)+1 if x[n]=y[n]
T(2n) = 2T(2n-1) if x[n]!=y[n]
T(2n) = 1 if n=0

T(2n) = 2T(2n-1)
= 2(2T(2n-2))
= 4T(2n-2)
= 4(2T(2n-3))
= 8T(2n-3)
= ...
= $2^i T(2n-i)$
= $2^{2n} = 4^n$

# Notes about LCS

Previous implementation solves the Evaluation version of LCS (compute the optimal value)

**Question**: extend it to solve the optimization version (find the solution)

What about the Recognition version?

**Question**: Is it in NP?

**Question**: Is it hard?

Consider this as a cliffhanger for next week!