Toward a general understanding of P=NP

P and NP, informally

- P and NP are two classes of problems that can be solved by computers.

- P problems can be solved ***quickly.***

    - Quickly means seconds or minutes, maybe even hours.

- NP problems can be solved ***slowly.***

    - Slowly can mean hundreds or thousands or years.

An equivalent question

- Is there a clever way to turn a ***slow*** algorithm into a ***fast*** one?
  - If P=NP, the answer is yes.
  - If P≠NP, the answer is no.

- Most people think P≠NP.

Back to P and NP

- P and NP are classes of ***solvable*** problems.

- *Solvable* means that there's a program that takes an input, runs for a while, but eventually stops and gives the answer.

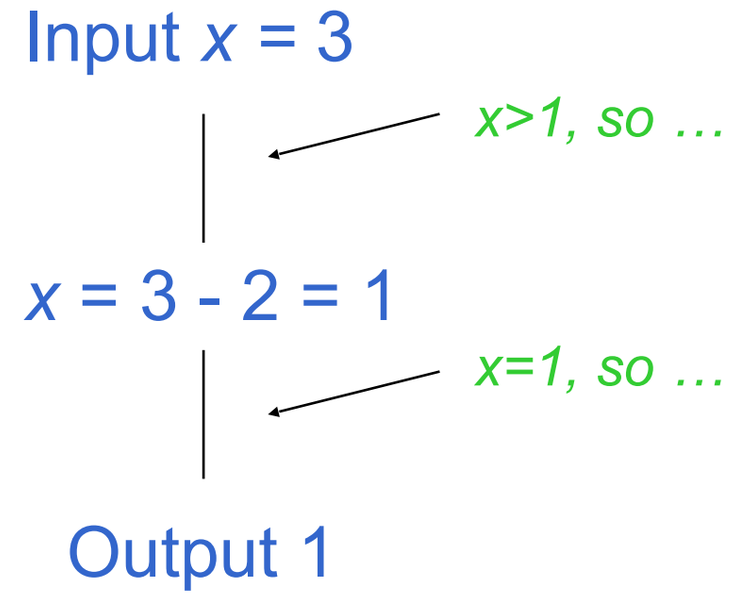
Computation "trees" for solvable problems

Program:

Input *x*

L1. If *x* > 1,
  set *x* = *x-2,*
  and GoTo L1.
If *x* = *0*,
  output *0.*
If *x* = *1,*
  output 1.

Example computation:



Solvability versus Tractability

- A problem is solvable if there is a program that always stops and gives the answer.
- The number of steps it takes depends on the input.
- A problem is **tractable or in the class P** if it is solvable and we can say Time(x)≤(some polynomial).
  - P stands for 'Polynomial-time computable'
- The description of a problem in **NP involves non-deterministic programming**.
  - NP stands for "non-deterministic, polynomial-time computable"

Class P

P is the complexity class containing decision problems which can be solved by a Deterministic Turing machine (DTM) using a polynomial amount of computation time, or polynomial time.

P is often taken to be the class of computational problems which are "efficiently solvable" or "tractable“.

But P also contains problems which are intractable in practical terms; for example, some require at least $n^{1000000}$ operations

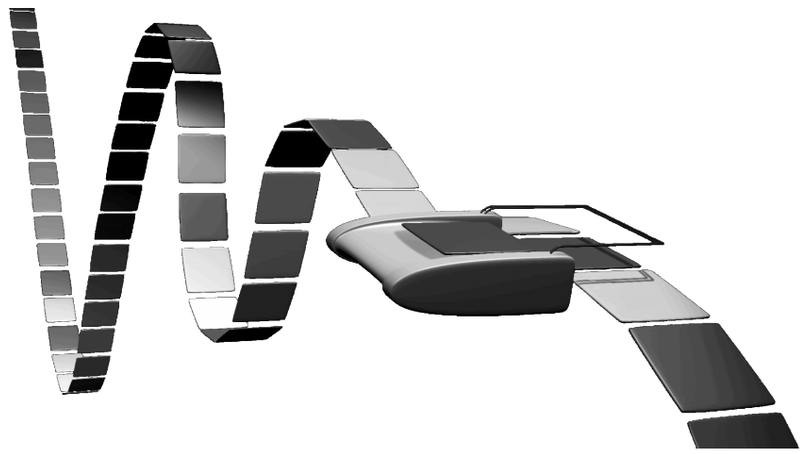
.

Deterministic Turing machine (DTM)

- DTM is:

- A Turing machine model (basic symbol-manipulating device which, despite its simplicity, can be adapted to simulate the logic of any computer).

- Consists of a finite state control, a read/write head, and a tape made up of a two-way infinite sequence of tape squares (or cells).



Source Wikipedia

Example of a problem in P

- Given a connected graph G, can its vertices be coloured using two colours so that no edge is monochromatic?

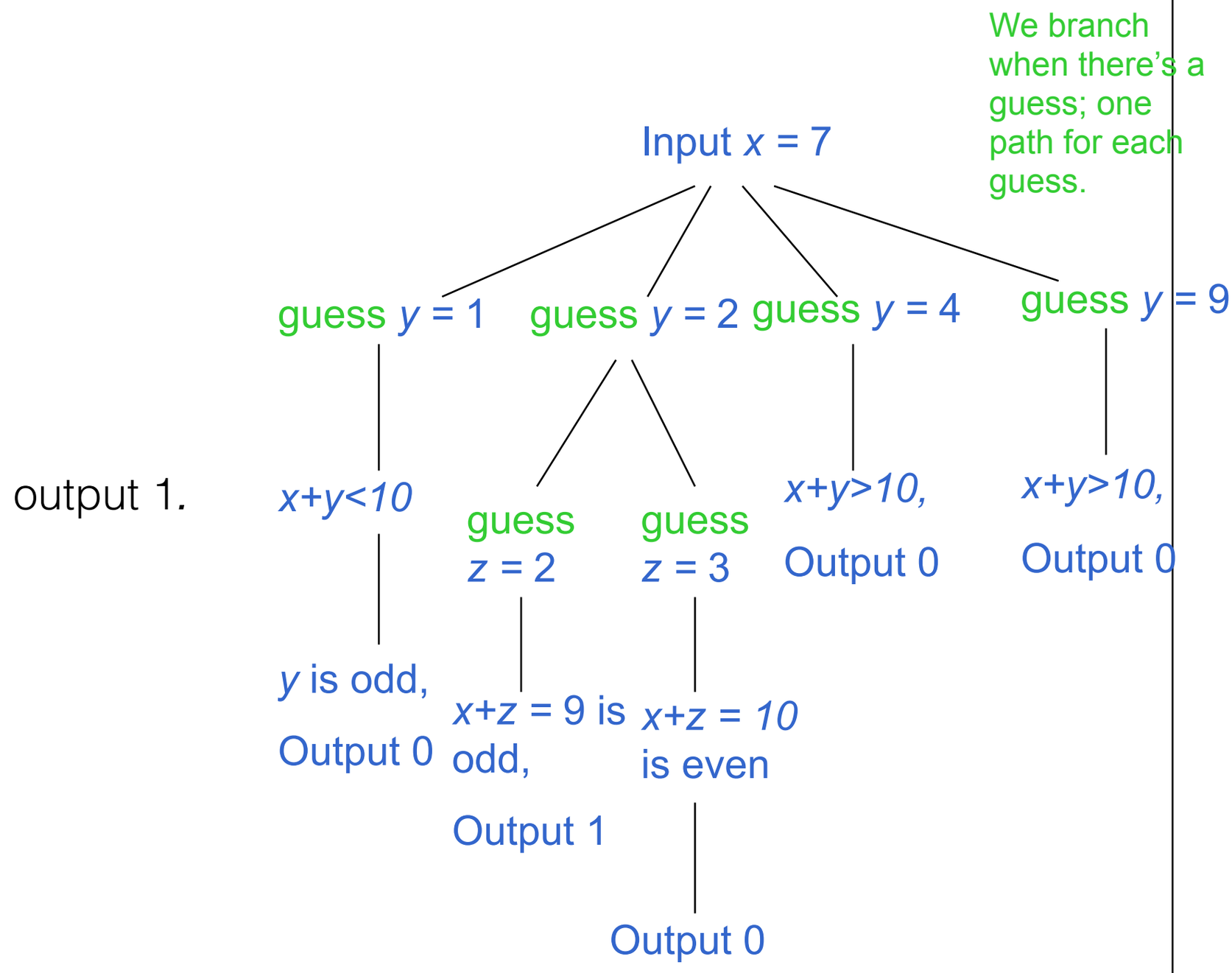
- Algorithm: start with an arbitrary vertex, color it red and all of its neighbours blue and continue. Stop when you run out of vertices or you are forced to make an edge have both of its endpoints be the same color.

Class NP

- The class NP is defined informally to be the class of all decision problems which, under reasonable encoding schemes, can be solved "verified" by polynomial time using Non-Deterministic Turing machines (NDTM)
    - NDTM model has exactly the same structure of DTM except it's featured with a guessing module that has its own write only head.
    - The guessing module's main purpose is providing the means for writing down the "guess" into the tape.
    - Polynomial time verifiability doesn't imply polynomial time solvability.
- Non-deterministic programs use a new kind of command that normal programs can't really use.
- Basically, they can guess the answer and then check to see if the guess was right.
- And they can guess all possible answers simultaneously (as long as it's only finitely many).

An example of non-deterministic programming

Program:
Input ***x.***
Guess ***y*** in {1, 2, 4, 9}.
If ***x+y*** > 10,
    stop and output 0.
Otherwise,
If ***y*** is even,
    Guess ***z*** in {2, 3},
    If ***x+z*** is odd, stop, output 1.
Otherwise, output 0.



Non-deterministic programming

- Convention:
  - If any computation path ends with a 1, the answer to the problem is 1 (we count this as “yes”).
  - If all computation paths end with a 0, the answer to the problem is 0 (we count this as “no”).
  - Otherwise, we say the computation does not converge.
- Again, we’re only interested in problems where this third case never happens – solvable problems.

The class NP

- If the computation halts on input ***x***, the length of the longest path is *NTime(x).*

- A problem is NP if it is solvable and there is a non-deterministic program that computes it so that *NTime(x)≤(Some polynomial).*

Exemple of problem in NP

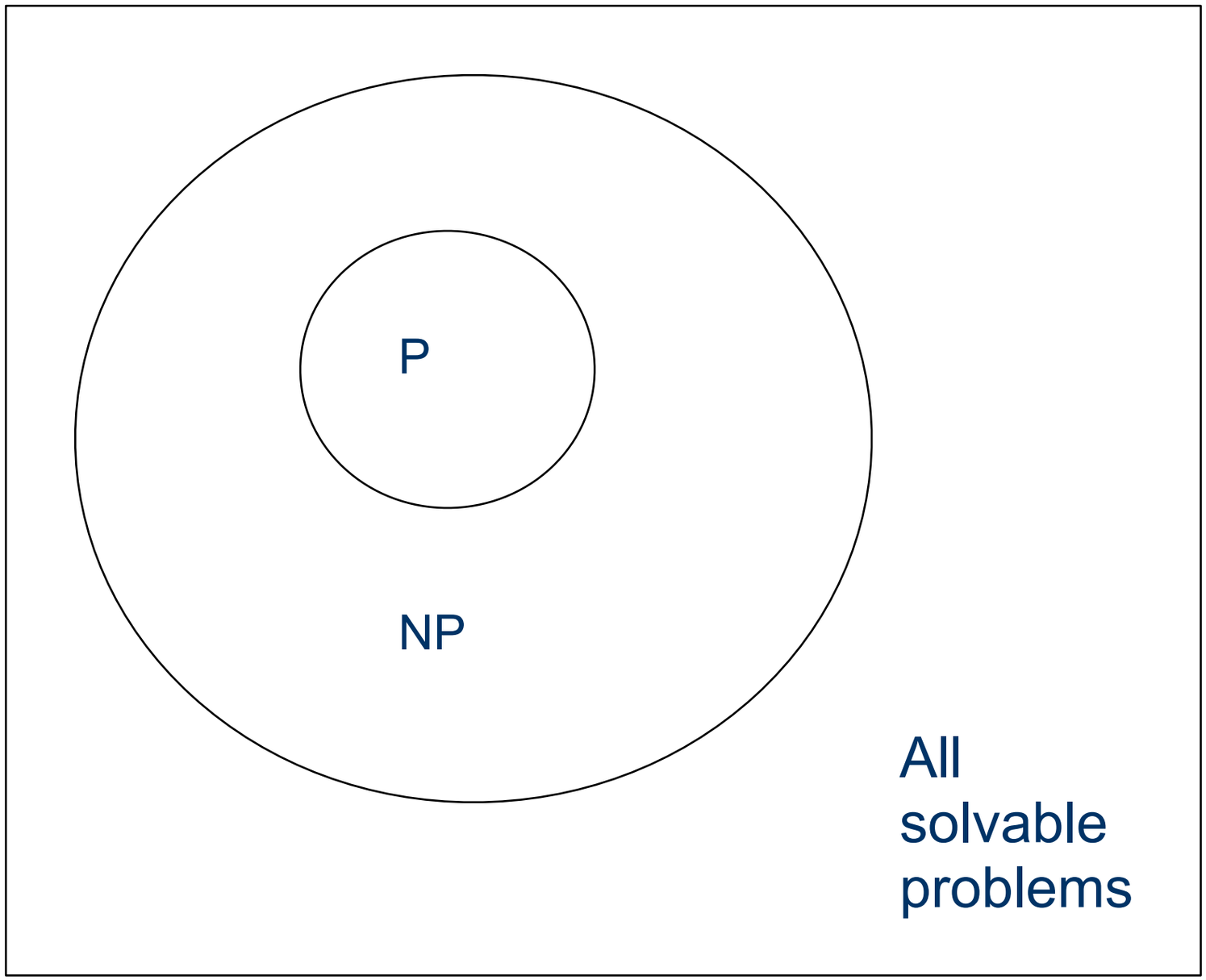
- Integer factorisation is in NP. This is the problem that given integers n and m, is there an integer f with 1 < f < m, such that f divides n (f is a small factor of n)?

- This is a decision problem because the answers are yes or no. If someone hands us an instance of the problem (so they hand us integers n and m) and an integer f with 1 < f < m, and claim that f is a factor of n (the certificate), we can check the answer in polynomial time by performing the division n / f.

Non-deterministic → Deterministic

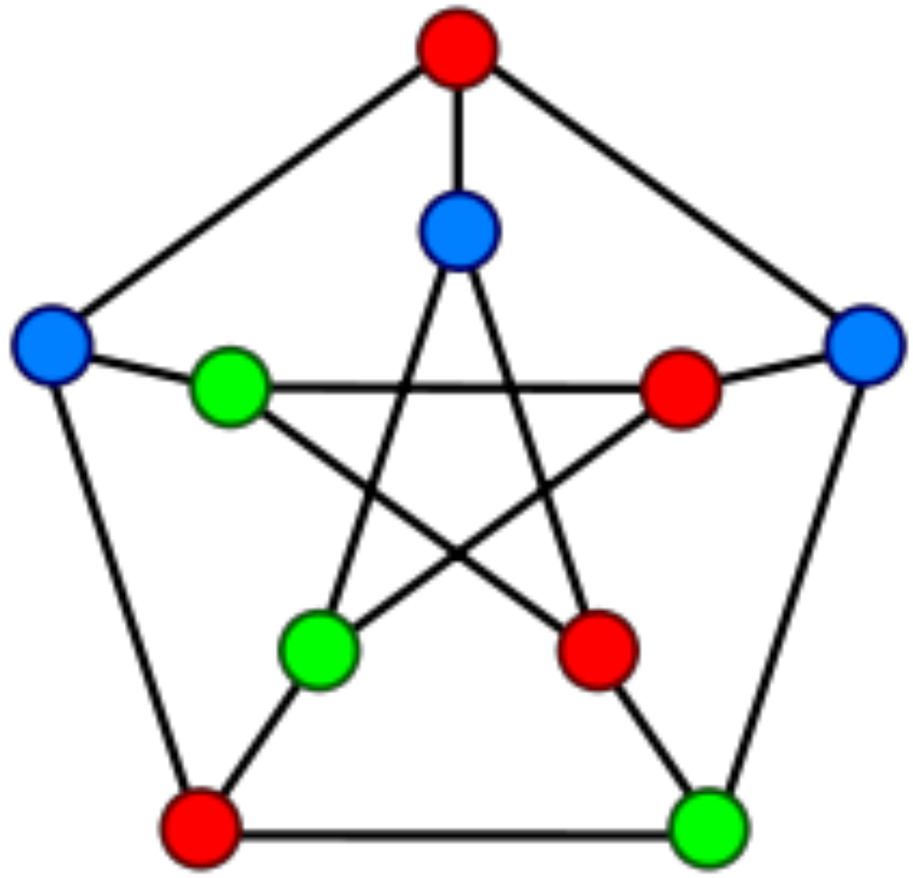
- A non-deterministic algorithm can be converted into a deterministic algorithm at the cost of ***time.***
- Usually, the increase in computation time is exponential.
- This means, for normal computers, (deterministic ones), NP problems are ***slow***.

The picture so far…



Graph coloring with 3 colors

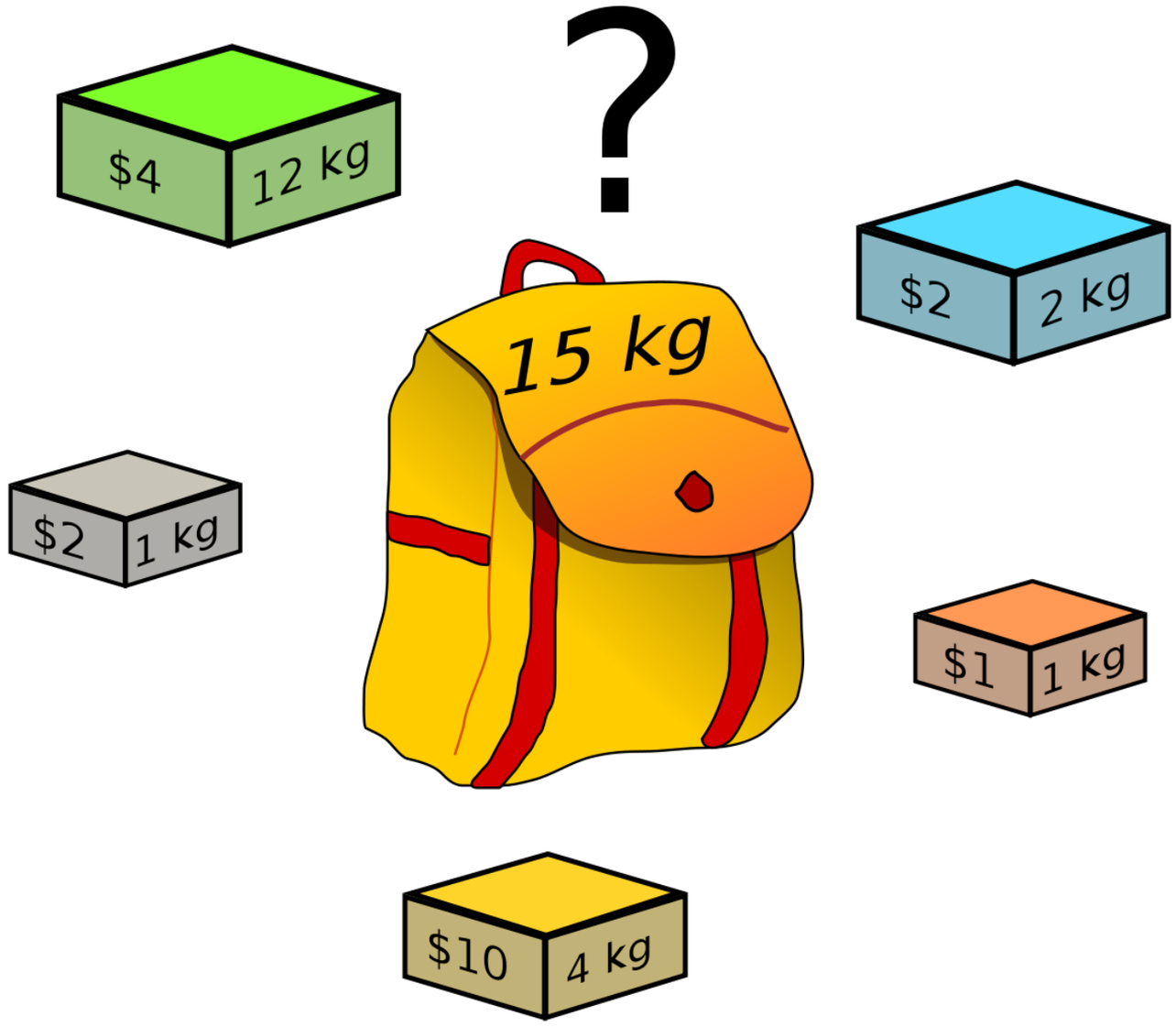
- The coloring problem with 3 colors is a NP problem



Source: Wikipedia

Knapsack problem

- Linear optimization problem with integer constraints !



Source: Wikipedia

Heuristic approaches to solve this kind of problem

- Solve for the linear problem by relaxing the integer constraint
    - i.e. solve for rationals
- Round rational solutions to lower integer values
- You get a (not so bad) solution, and a bound on maximum expected value

Method Branch-and-bound

- The problem solutions are represented as a tree
  - At each node, one possible solution is chosen
  - Final solution is supported by leaves
- The branch and bound method consists in pruning some branches of the tree when we know that the corresponding solution is worse than an already found solution
- Example on the knapsack problem

Dynamic programing

- Cutting the problem in sub-problems
- Storing intermediate results
- Example with the knapsack problem

SAT (The problem of satisfiability)

- Take a statement in propositional logic, (like $p \vee q \rightarrow p \wedge q$, for example).
- The problem is to determine if it is satisfiable. (In other words, is there a line in the truth table for this statement that has a “T” at the end of it.)
- This problem can be solved in polynomial time with a non-deterministic program.

SAT, cont.

- We can see this by thinking about the process of constructing a truth table, and what a non-deterministic algorithm would do:

SAT, cont.

- The length of each path in the computation tree is a polynomial function of the length of the input statement.

- SAT is an NP problem.

NP completeness

- If P≠NP, SAT is a witness of this fact, that is, SAT is NP but not P.
- It is among the "hardest" of the NP problems: any other NP problem can be coded into it in the following sense.

If R is a non-deterministic, polynomial-time algorithm that solves another NP problem, then for any input, ***x***, we can quickly find a formula, ***f***, so that ***f*** is satisfiable when R halts on ***x*** with output 1, and ***f*** is not satisfiable when R halts on ***x*** with output 0.

NP complete problems

- Problems with this property that all NP problems can be coded into them are called *NP-hard.*

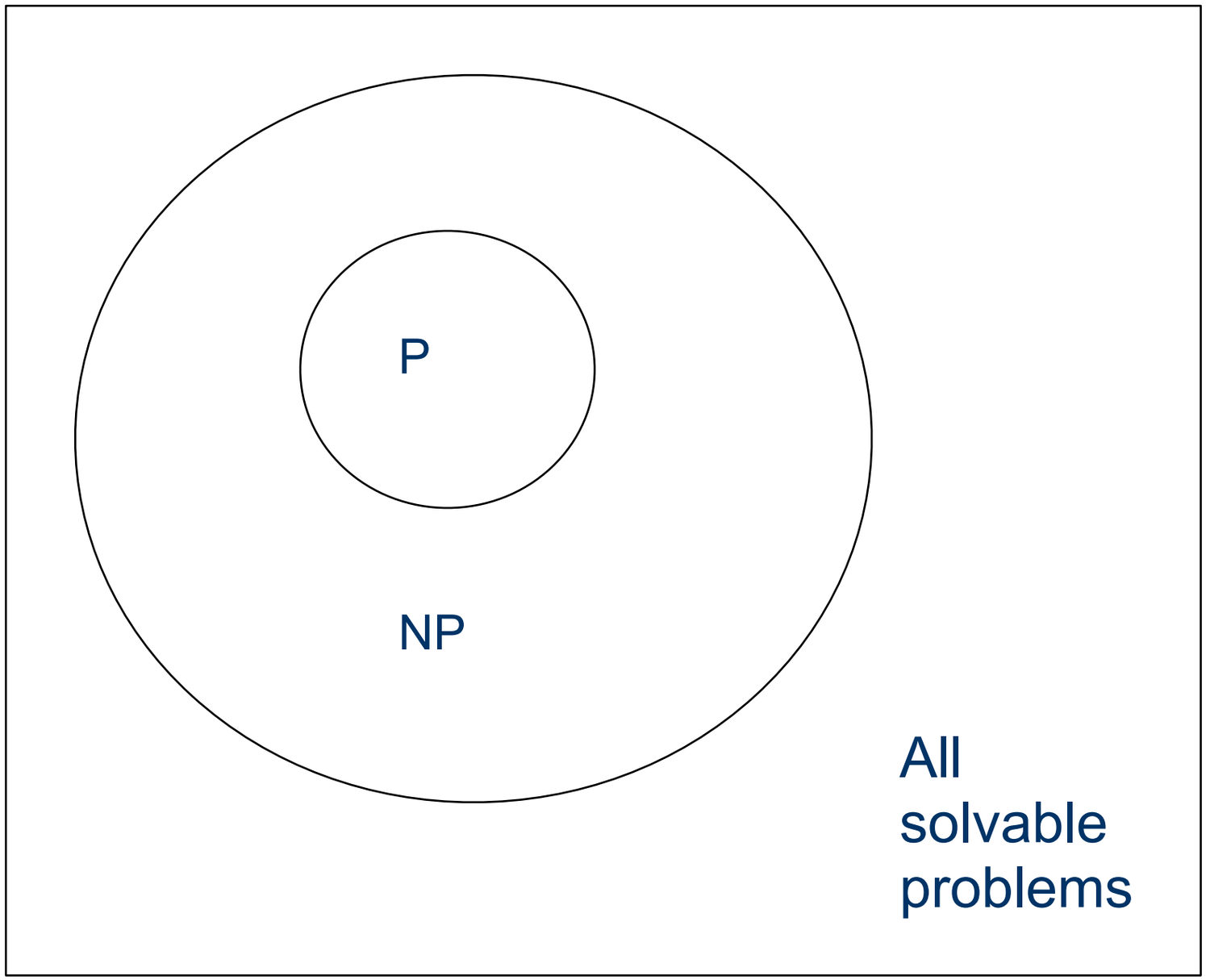
- If they are also NP, they are called *NP-complete.*

- If P and NP are different, then the NP-complete problems are NP, but not P.

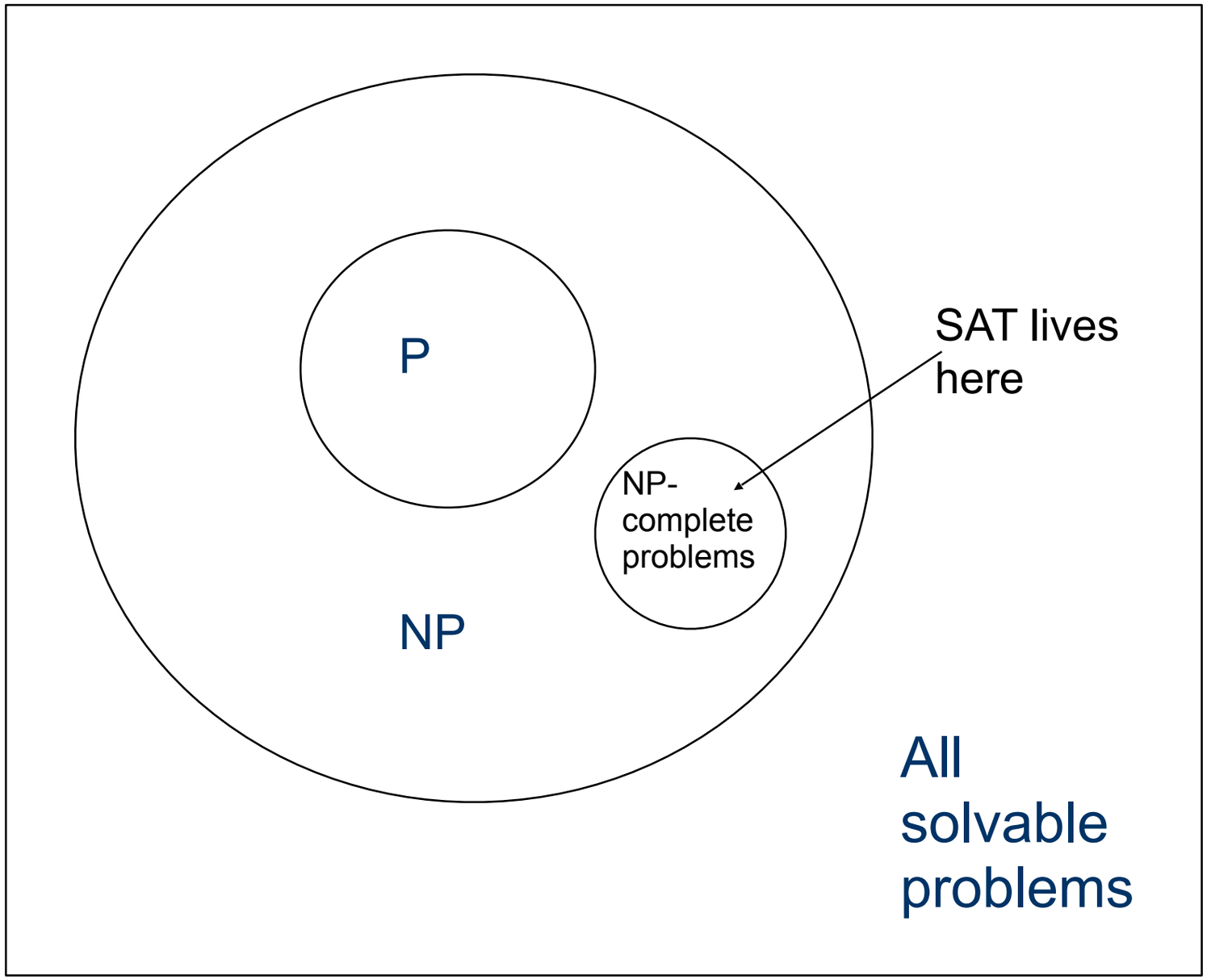
Equivalence between coloring graph problems and SAT

- Let's show that a SAT problem can be converted into a graph coloring problem

The picture so far…



The picture so far…

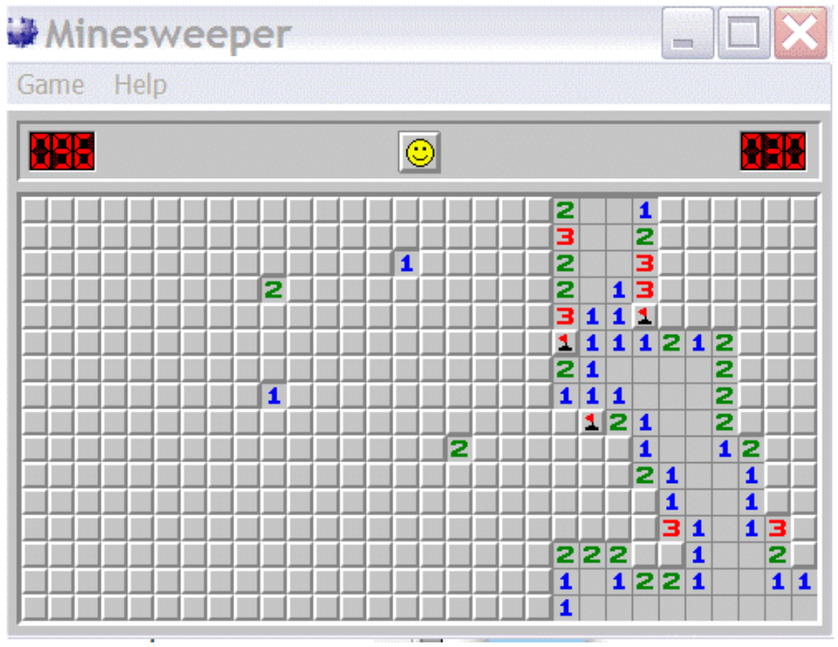


Question:

- Is there a clever way to change a non-deterministic polynomial time algorithm into a deterministic polynomial time algorithm, without an exponential increase in computation time?
- If we can compute an NP complete problem quickly then ***all*** NP problems are solvable in deterministic polynomial time.

Minesweeper

- The Minesweeper consistency problem:
    - Given a rectangular grid partially marked with numbers and/or mines – some squares being left blank – determine if there is **some** pattern of mines in the blank squares that give rise to the numbers seen. That is, determine if the grid is consistent for the rules of the game.



Minesweeper

- This is certainly an NP problem:
  - We can guess all possible configurations of mines in the blank squares and see if any work.
- To see that it is NP complete is much harder.
  - The trick is to code logical expressions into partially filled minesweeper grids. Then we will have demonstrated that SAT can be coded into Minesweeper, so Minesweeper is also NP-complete.

Sudoku

- Also NP-complete

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

Beyond NP

- Chess and checkers are both EXPTIME-complete – solvable with a deterministic program with computation bounded by $2^{p(x)}$ for some polynomial $p(x)$.
- Go (with the Japanese rules) is as well.
- Go (American rules) and Othello are PSPACE-hard, and Othello is PSPACE-complete (PSPACE is another complexity class that considers memory usage instead of computation time).

Hierarchy of complexity

- EXPSPACE and PSPACE consider exponential and polynomial solvability in space (instead of time)