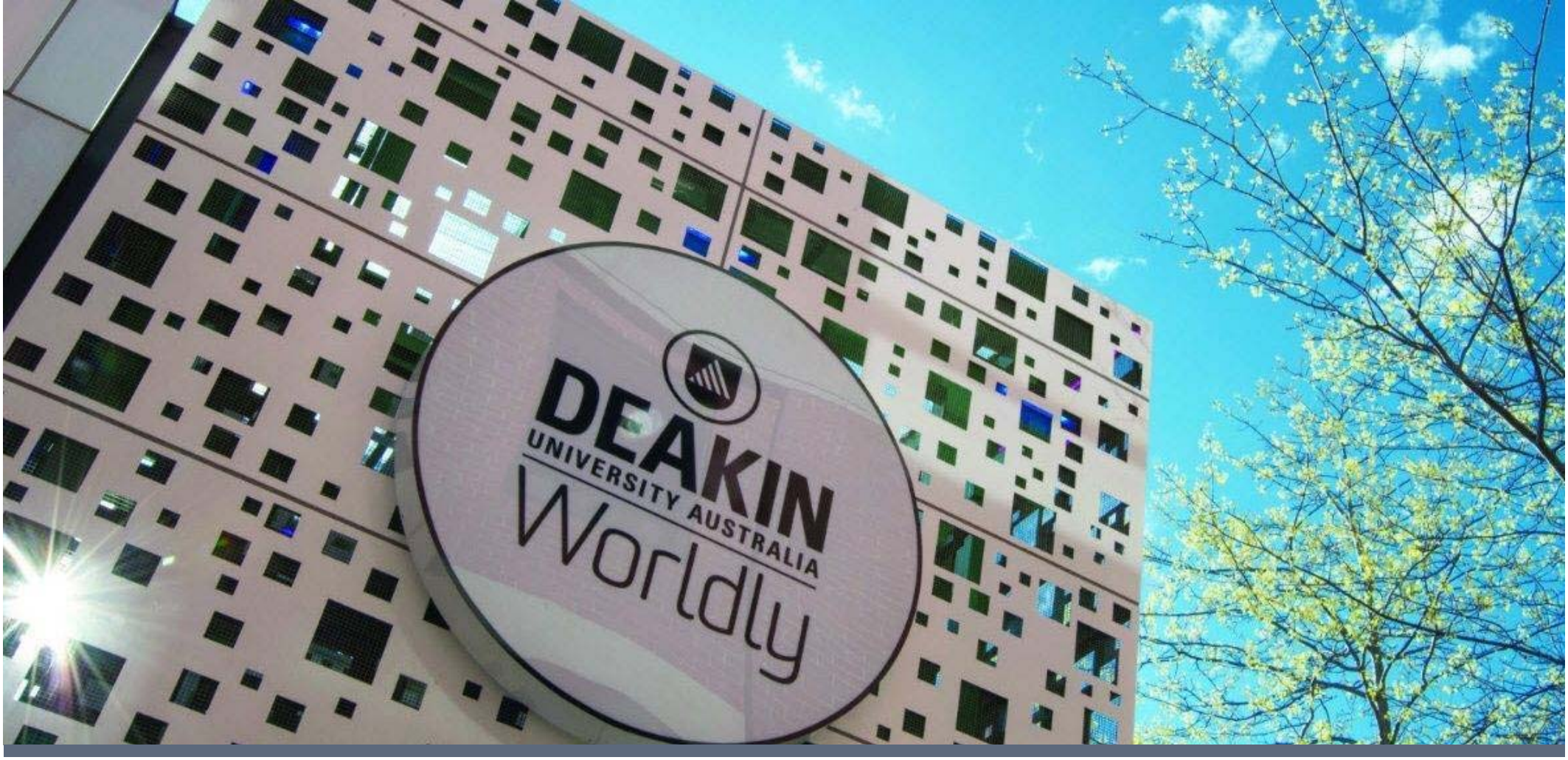


Lecture 11. Introduction to Complexity Theory

SIT221 Data Structures and Algorithms

Complexity Theory: Motivation

Algorithms provide a way to obtain a solution, but they assume that we are using the same underlying *model*.

How do we know how our algorithms will perform? If we can

- provide a model that performs in a similar way to real computation, and then
- apply our algorithms to the model,

then we can compare algorithms.

Complexity Theory: Motivation

Why a model is important?

There is scope for variations in:

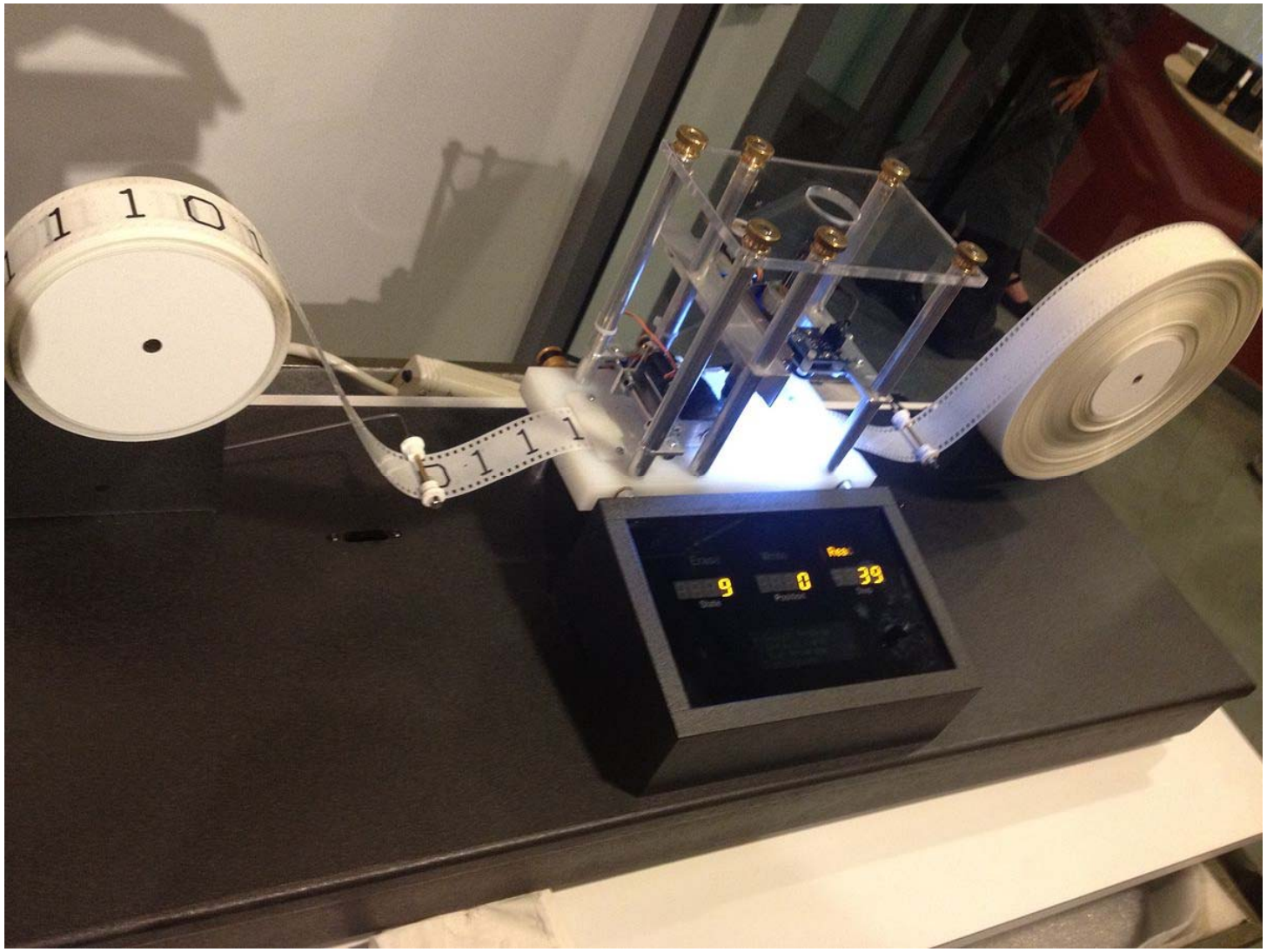
- Hardware
- Operating Systems
- Software

We need to be able to prove certain properties and compare algorithms generally to be useful.

Complexity Theory: Computational Basics

- Alonzo Church developed the untyped Lambda Calculus in the 1920s:
  - Everything reduced to the definition and application of functions
  - A simple programming language
  - A basis for proof
- Alan Turing developed a hypothetical device that could simulate the logic of any algorithm.
  - This machine is equivalent to the Lambda Calculus.
  - The Church-Turing thesis states that a function is algorithmically computable if and only if it is computable by a Turing Machine.

Complexity Theory: Turing Machine (TM)



See https://youtu.be/E3keLeMwfHY

Complexity Theory: Turing Machine (TM)

It is a mathematical model of a machine that operates on an infinitely long tape broken up into cells.

- The tape has symbols to be read and written by a tape head, one at a time.
- The machine has registers to keep track of its state.
- A table of instructions determining what to do based on the tape position and the current symbol.

Everything we can do on a computer, we can do on this model of a machine.

Complexity Theory: Computability

Every computable function has:

- A finite program that describes specifically how to compute the result.
- If a value is found, then it must be the correct value.

There are many types of problems and some of them cannot be computed (are undecidable).

The Halting Problem: Can we write a program that will look at any computer program and its input and decide if the program will halt (not run infinitely)?

$\Rightarrow$ Given a program P and input i for the Turing Machine, does P halt on i?

Complexity Theory: Computability

- A practical solution might be to run the program and if it halts, you have the answer. If after a given amount of time it does not halt, guess that it will not halt.

  However, you would not know if the program would eventually halt.

- For a problem to be **undecidable** you just have to prove that there is one case it can not produce an answer for.

- The case that Turing came up with that can never be solved involves giving a program itself as input.

Complexity Theory: Decision Problems

- A decision problem is a special type of problem where the answer is:
  - **yes**, or
  - **no**.
- Often used for:
  - membership
  - simplifying other problems to see if they are decidable
  - proving other properties
- Examples:
  - Does a given graph G have a cycle?
  - Are all the nodes of a given graph G reachable?
  - Does a text T contain a pattern P?
  - Does an instance of the Travelling Salesman Problem have a solution of length at most L?

Complexity Theory: Decision Problems

- Decision problems can be captured as languages.
- The Turing Machine leaves YES on the tape when the string of a language is acceptable, NO if not.
- A decision problem (or language) is decidable if there is a Turing Machine that solves that problem.

- Example of a language:
  - The problem of determining whether a number is divisible by 7 corresponds to the language of all strings that form numbers divisible by 7.
  - The Turing Machine for divisibility by 7 solves the problem and represents the language.

Complexity Theory: Types of the Turing Machine

Deterministic Turing Machine (DTM)

In a *deterministic* Turing machine, the set of rules prescribes **at most one action** to be performed for any given situation.

A deterministic Turing machine has a transition function that, for a given state and symbol under the tape head, specifies three things:

- the symbol to be written to the tape,
- the direction (left, right or neither) in which the head should move, and
- the subsequent state of the finite control.

For example, 1 on the tape in state 3 might make the DTM write 0 on the tape, move the head one position to the right, and switch to state 5.

Complexity Theory: Types of Turing Machine

Non-Deterministic Turing Machine (NTM)

In a *non-deterministic* Turing machine (NTM), the set of rules may prescribe more than one action to be performed for any given situation.

For example, 1 on the tape in state 3 might allow the NTM to:

- Write a 0, move right, and switch to state 5, or
- Write an 1, move left, and stay in state 3.

Resolution of multiple rules:

Whereas a DTM has a single "computation path" that it follows, an NTM has a "computation tree".

$\Rightarrow$ If at least one branch of the tree halts with an "accept" condition, we say that the NTM accepts the input.

Complexity Theory: Efficient Algorithms

- How do we compare two algorithms?

Complexity Theory: Efficient Algorithms

- How do we compare two algorithms?
- We can talk about how many steps it takes for them to reach a solution for a given input.
- We then start to worry about **how efficiently** we can solve the problem.
- We consider problems to be difficult if their solution requires a lot of resources.
- We use models of their resource usage to allow us to compare problems and various algorithms designed to solve them.

Complexity Theory: Easy Problems

- An algorithm A runs in **polynomial time** (is a polynomial time algorithm), if there is a polynomial $p(n)$ such that its execution time on inputs of size n is $O(p(n))$.
- A problem can be solved in polynomial time (and is 'easy') if there is a polynomial time algorithm that solves it.
- We call an algorithm **efficient** if and only if it runs in polynomial time.
- Problems that can be solved in polynomial time:
  - Integer Addition
  - Integer Multiplication
  - Test whether a graph is acyclic
  - Almost all problems that we consider in this course.

Complexity Theory: Difficult Problems

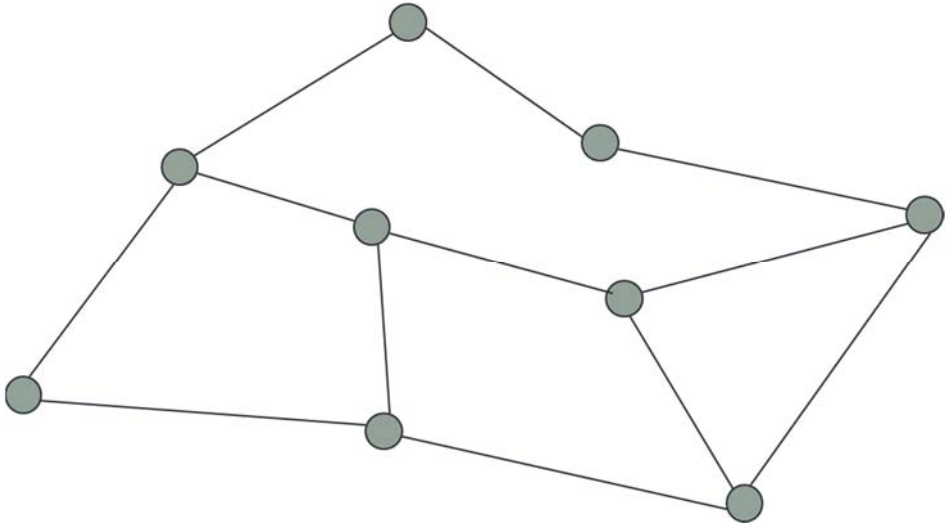
- There are many problems for which no efficient algorithm (i.e. polynomial-time) is known.
  - Hamiltonian Cycle Problem
  - Traveling Salesman Problem
  - Graph Coloring Problem
  - 0-1 Knapsack Problem
  - The Subset-Sum Problem

Complexity Theory: The Hamiltonian Cycle Problem

Given is an undirected graph $G = (V, E)$.

Decide whether it contains a Hamiltonian cycle.

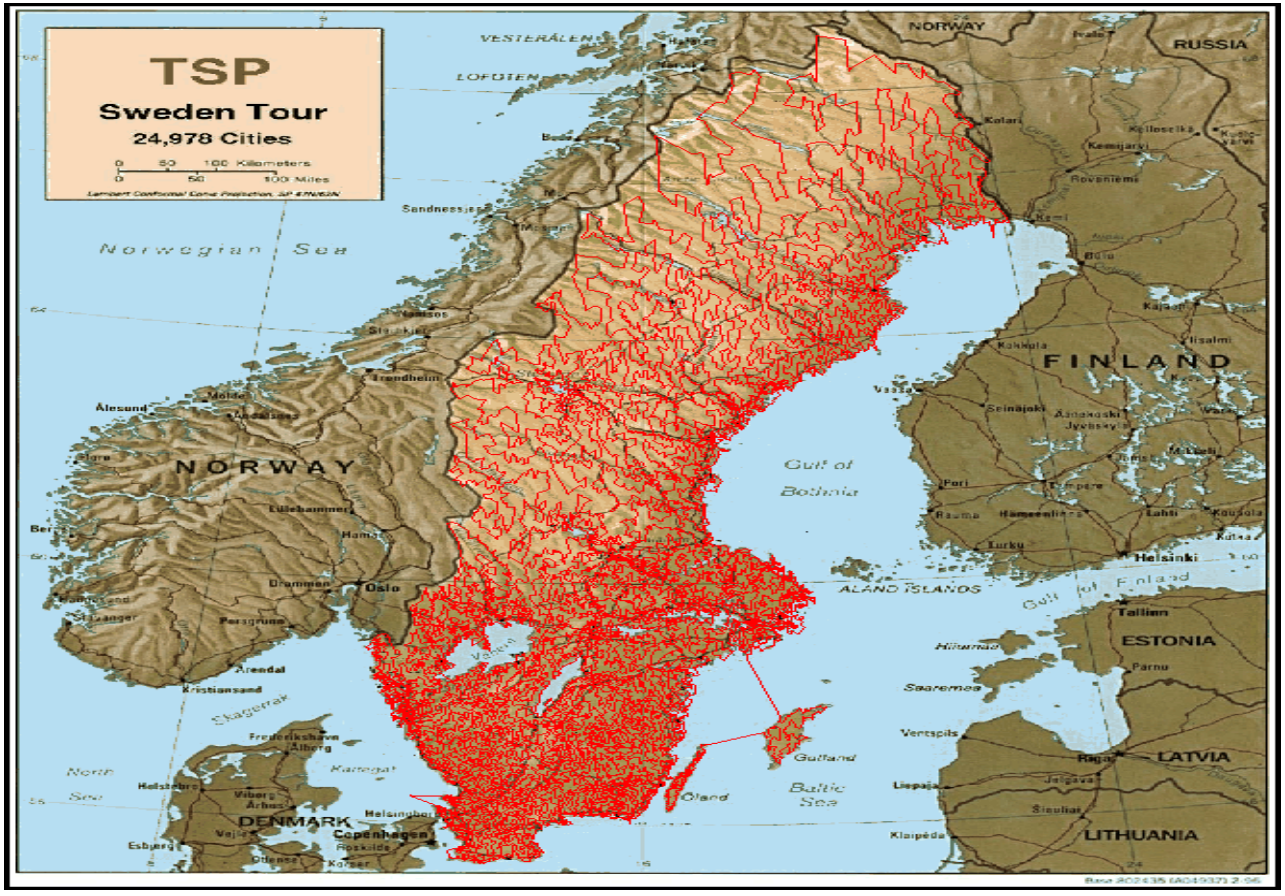
A Hamiltonian cycle is cycle that visits each node exactly once and returns to the start vertex.



Complexity Theory: The Traveling Salesman Problem

Given is complete edge-weighted undirected graph $G = (V, E)$ and an integer L.

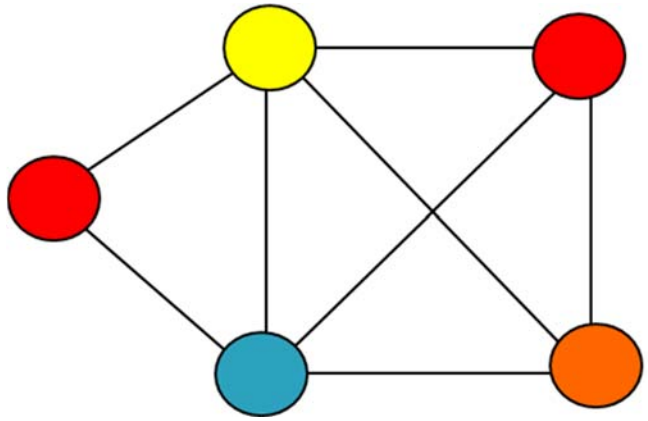
Decide whether G contains a Hamiltonian cycle of cost at most L.



Complexity Theory: The Graph Colouring Problem

Given is an undirected graph $G = (V, E)$ and an integer K.

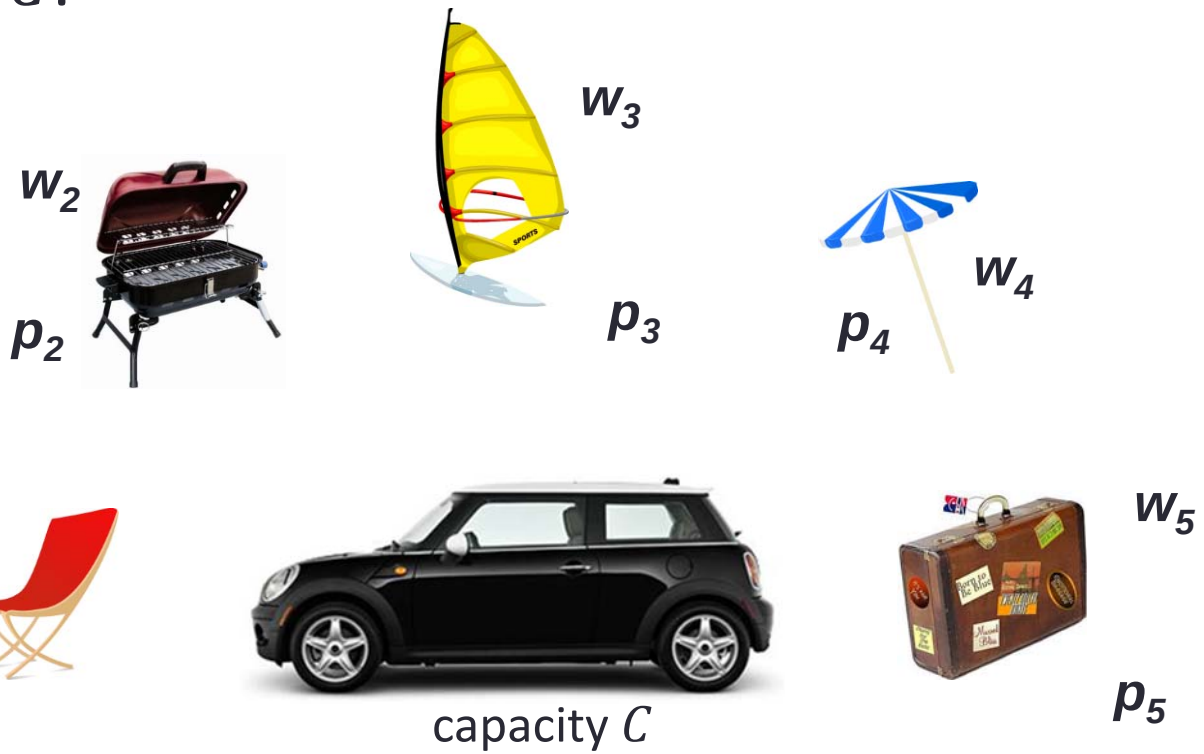
Decide whether there is a colouring of the nodes with K colour such that any two adjacent nodes are coloured differently.



Difficult Problems: The 0-1 Knapsack Problem

Given is a collection of items with weights $W = \{w_1, \dots, w_n\}$ and profits $P = \{p_1, \dots, p_n\}$.

Decide whether there exists a subset of the items giving the total profit of Z such that their total weight does not exceed the capacity C.



Difficult Problems: The Subset-Sum Problem

Given is a set $S = \{s_1, s_2, \dots, s_n\}$ of positive integers and a target value T.

Decide whether there exists a subset S that sums up the integers to the target value T.

Example:

- for $S = \{1,2,5,6, \dots, 8\}$ and $T = 9$,
- there are two solutions: $S = \{1,2,6\}$ and $S = \{1,8\}$.

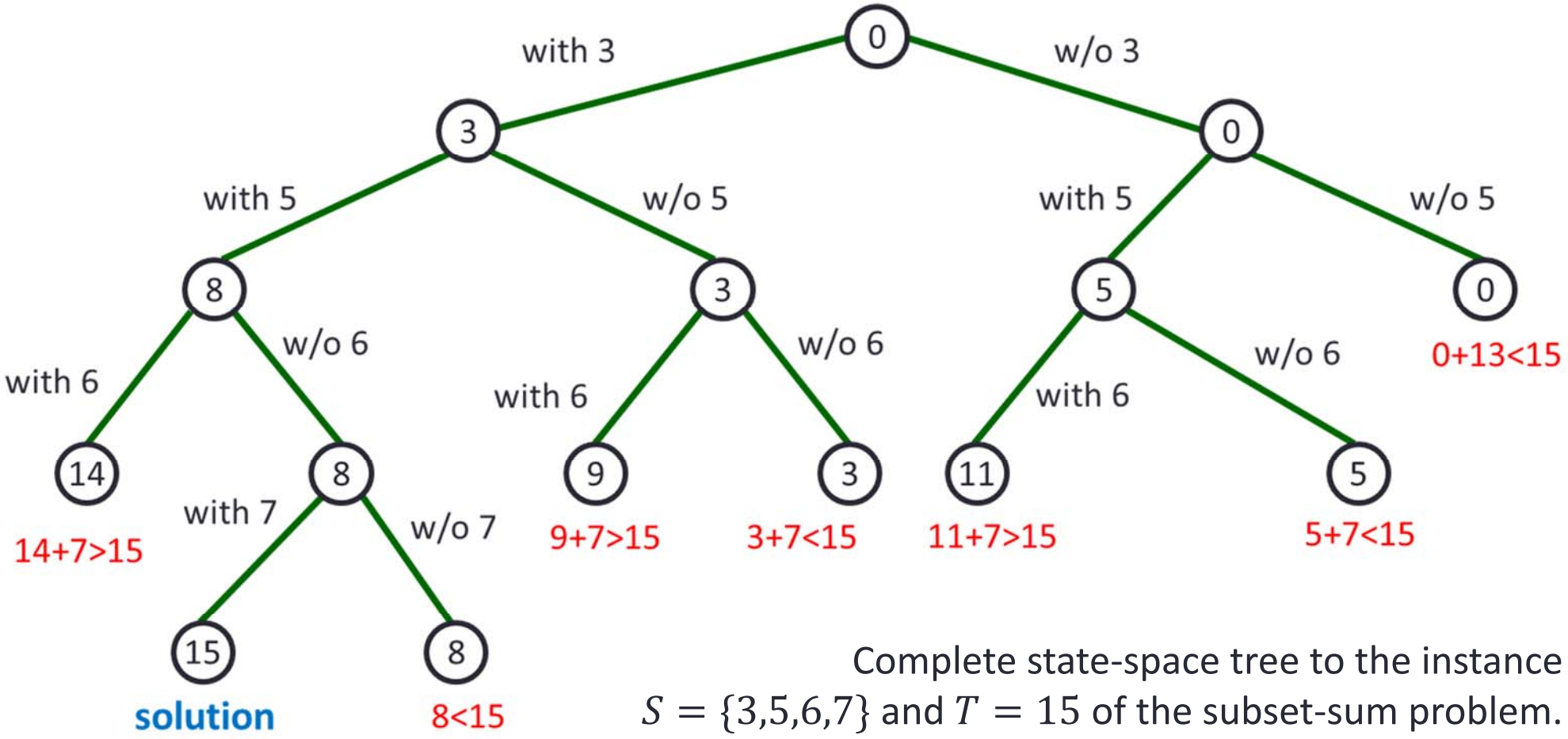
Worst-case: try all possible subsets and get solution in $O(2^n)$.

Best-case: must at least look at every element, thus $O(n)$.

Tight bound: No one knows! $\Rightarrow O(?)$.

Difficult Problems: The Subset-Sum Problem

- The state-space tree can be constructed as a binary tree.
- The root of the tree represents the starting point.
- Its left and right children represent, respectively, inclusion and exclusion of a number.



Complete state-space tree to the instance $S = \{3,5,6,7\}$ and $T = 15$ of the subset-sum problem.

Complexity Theory: Difficult Problems

- We do not know whether polynomial time algorithms exist for the mentioned difficult problems.
- It is very likely (and almost all people in computer science believe) that there are no polynomial time algorithms for these problems.
- They belong to a class of equivalent problems known as **NP-complete problems**.
  (NP stands for “nondeterministic polynomial time”)

Complexity Theory: Efficient Algorithms and the TM

- We can take a Turing Machine M that deals with a problem and give it input x.
- If the time taken to solve the problem is related to x, then we can express the time as $f(x)$.
- A problem can be solved in $f(x)$ if there exists a Turing Machine that can solve it in $f(x)$ time.
- The Turing machine represents the algorithm!

Complexity Theory: Complexity Classes P and NP

- **Class P** is the complexity class of decision problems that can be solved in $f(x)$, where $f(x)$ is polynomial, on a deterministic sequential Turing Machine.
- **Class NP** is the complexity class of decision problems that can be verified in polynomial time, yet they can be solved in polynomial time on a non-deterministic sequential Turing Machine.
- Are they equally difficult problems?

Complexity Theory: Classes P and NP (Formally)

- Decision problems can be captured as languages.
- A language L is a set of strings defined over some alphabet Σ.
- Every decision algorithm A defines a language L that is a set consisting of every string x such that A outputs "yes" on input x. We say "A accepts x" in this case.
- Then a **complexity class is a collection of languages**.
- The Turing Machine represents the algorithm.

- Class P is the complexity class consisting of all languages that are accepted by polynomial-time algorithms.
- Class NP is the complexity class consisting of all languages accepted by polynomial-time non-deterministic algorithms.

Complexity Theory: Example of NP-class Problem

The Subset-Sum Problem is in NP:

- Given is a set $S = \{s_1, s_2, \dots, s_n\}$ of positive integers and a target value T.
- Decide whether there exists a subset S that sums up the integers to the target value T.
- We can investigate all possible solutions in polynomial time assuming that the non-deterministic Turing Machine can explore the whole "computation tree" in parallel being in multiple states at once, i.e. it can parallelise the search without any limitation.
  Thus, we can solve the problem using the non-deterministic Turing Machine.
- Given an answer, we can check it in polynomial time just by summing up all the integers of the subset provided in the answer.
  Thus, we can verify the problem in polynomial time.

Complexity Theory: Example of NP-class Problem

The **decision** version of the Travelling Salesman Problem is in NP:

- Given is a complete edge-weighted undirected graph G and an integer L. Decide whether G contains a Hamiltonian cycle of cost at most L.
- We can investigate all possible solutions in polynomial time traversing the exponentially many (all) the feasible paths in parallel.
  Thus, we can solve the problem using the nNon-deterministic Turing Machine.
- Given an answer, we can check it in polynomial time just by summing up all the distances provided by the solution and comparing it to L.
  Thus, we can verify the problem in polynomial time.

Complexity Theory: Complexity Classes P and NP

- Obviously P is a subset of NP.

  What can be solved using the non-deterministic can be sure solved with the non-deterministic Turing Machine.

- One of the major open question in Computer Science: Is P=NP? Most people believe that P≠NP (see http://www.win.tue.nl/~gwoegi/P-versus-NP.htm).

Complexity Theory: Problems beyond the NP Class

The **optimisation** version of the Travelling Salesman Problem is **not** in NP:

- Given is a complete edge-weighted undirected graph G.
  Find a Hamiltonian cycle of the minimum cost.
- We can investigate all possible solutions in polynomial time traversing the exponentially many (all) the feasible paths in parallel.
  Thus, we can **still** solve the problem using the Non-Deterministic Turing Machine.
- Given an answer, we **can not** check it in polynomial time just by summing up all the distances provided by the solution as we do not know if the given cost is actually the minimum one. To verify the problem, we **have to solve it!**
  Thus, verification of an answer for the problem is hard as solving it.

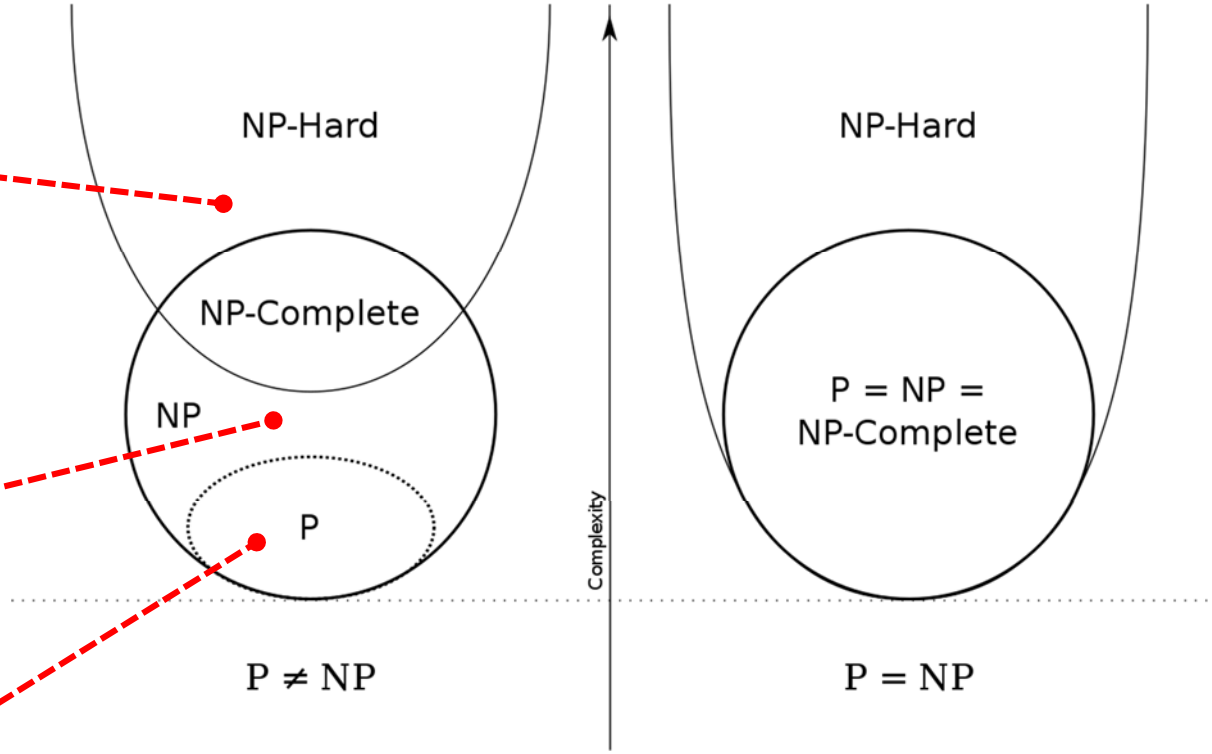
Complexity Theory: Complexity Classes Diagram

Problems beyond the NP class can not be verified in polynomial time time.

NP-hard: Problems that are at least as hard as problems of the NP class.

NP: Decision problems that can be can be verified in polynomial time, yet they can be solved in polynomial time on a non-deterministic Turing Machine

P: Decision problems that can be solved in polynomial time on a deterministic Turing Machine



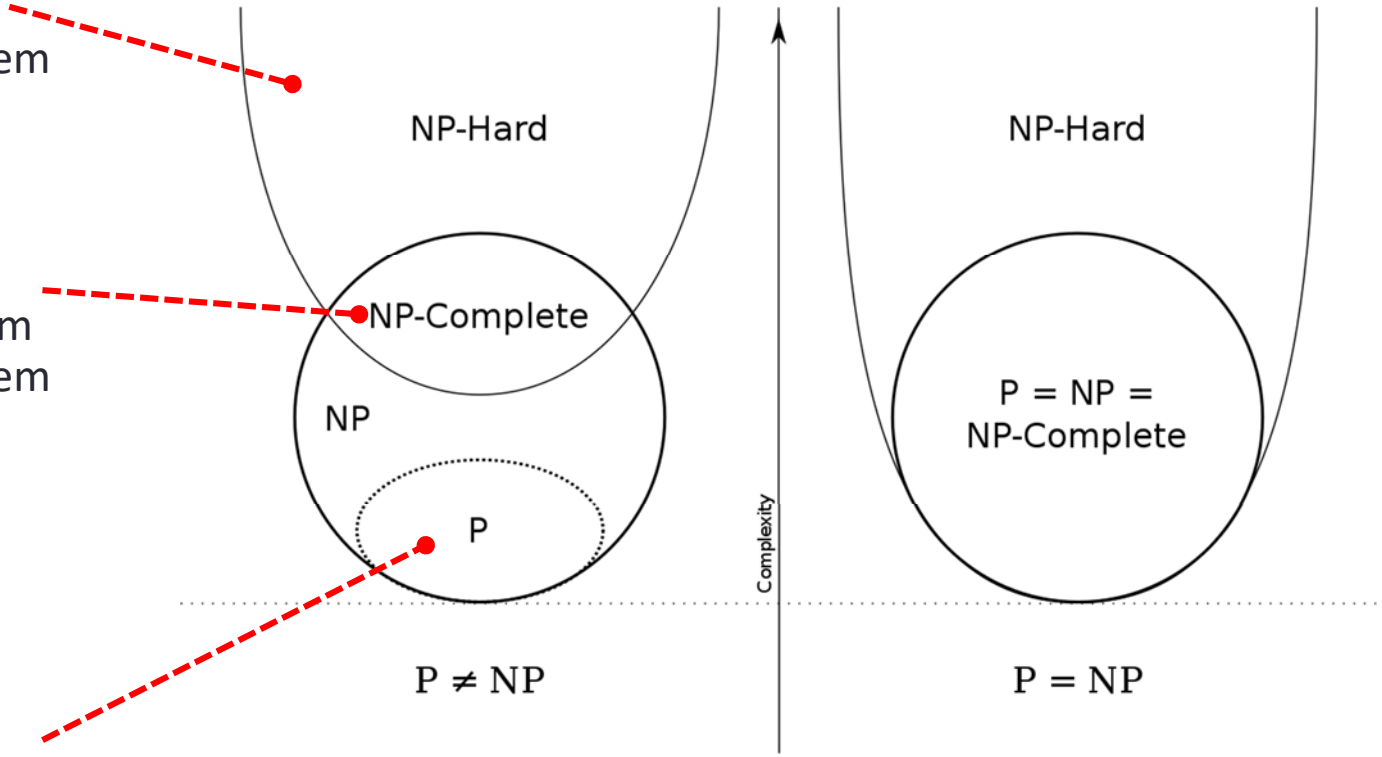
The left side is valid under the assumption that P≠NP, while the right side is valid under the assumption that P=NP

Complexity Theory: Complexity Classes Diagram

The Halting Problem,

Optimization problems:
- Traveling Salesman problem
- 0-1 Knapsack problem
- etc.

Decision problems:
- Hamiltonian Cycle problem
- Traveling Salesman problem
- Graph Colouring problem
- 0-1 Knapsack problem
- The Subset-Sum problem
- etc.



Polynomial time solvable problems:
- integer multiplication and addition
- searching and sorting
- The Depth-First and Breadth-First graph traversing
- the shortest path problem
- etc.

Complexity Theory: NP-hard problems

NP-hard class of problems represents those problems that

- are at least as hard as the hardest problems in the NP class
- a problem H is NP-hard when every problem L in NP can be reduced in polynomial time to H.

Implications:

- If we can solve NP-hard problem H, then we can solve problem L in the class NP as efficiently as problem H.
- Finding a polynomial algorithm to solve any NP-hard problem would give polynomial algorithms for all the problems in NP.

Complexity Theory: NP-Complete problems

A decision problem is NP-complete if and only if it is

- NP-hard, and
- in the class of NP problems.