CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Computing@CIT

Decision Analytics

Lecture 5: NP-completeness

Decision problems

- Let Σ be a finite alphabet of symbols, i.e. $|\Sigma| \leq c$ for some $c \in \mathbb{N}$
- Then the set of strings over this alphabet is defined as all finite length sequences $\Sigma^* = \bigcup_{n \in \mathbb{N}_0} \Sigma^n$
- A decision problem is defined as a subset of strings $X \subset \Sigma^*$ for which the decision is “yes”
- An algorithm A solves the problem X, if given the input $s \in \Sigma^*$ the output is

$$A(s) = \text{"}yes\text{"} \Leftrightarrow s \in X$$

- The algorithm A runs in polynomial time if for every input string s the execution of $A(s)$ terminates in at most $p(|s|)$ “steps”, for some polynomial

$$p(n) = \sum_{i \leq d} a_i n^i$$

- A decision problem X is solvable in polynomial time, if such a polynomial time algorithm A exists ($X \in P$)

Example

- The decision problem, if a given sequence of digits is sorted is solvable in polynomial time

- Why?
- Because we know a linear time algorithm to check if this is the case or not

- We proof that a problem $X \in P$ by constructing a polynomial time algorithm to solve it

Decision problems

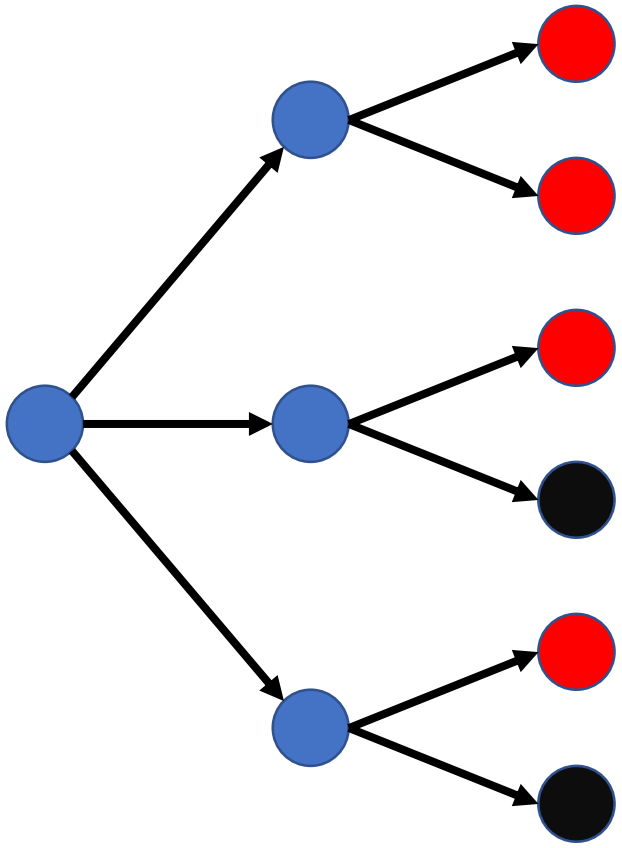
- For non decision algorithms it is often possible to phrase them as decision problems
- For example, instead of maximising the value of items to fit in the knapsack the problem can be restated as asking if it is possible to fit more than a specific given value into the knapsack?
- If we had a polynomial time algorithm for deciding this question, we could simply increase the target value step by step until the answer is no
- The resulting optimisation algorithm would also run in polynomial time then
- If on the other hand we had a polynomial time algorithm for finding the optimal value, we could immediately answer the decision problem as well
- Therefore the complexity of the two is the same, even through strictly speaking the complexity classes are only defined for decision problems

Non-deterministic computation

- Let's assume now that the decision algorithm needs to make a sequence of branching decisions
- Let's further assume that there are only polynomially many options for each of these decisions

- A **non-deterministic computation model** is now able to guess the "correct" decision leading to a "yes" answer <u>if any such option exists</u>

- Note the asymmetry in this, as a correct guess is <u>only</u> possible if there is a potential "yes" answer



Certification algorithms

- More formally we can collect all these branching decisions into a string $t \in \Sigma^*$, which we call a certificate or a proof
- An algorithm $C(s,t)$ is a certifier for a problem X if for every string $s \in X$ there exists a string $t \in \Sigma^*$ so that

$$C(s,t) = "yes" \Leftrightarrow s \in X$$

- Again, note the asymmetry that the certificate only needs to exist for $s \in X$
- A certifier runs in polynomial time, if $C(s,t)$ terminates in at most $p(|s|)$ "steps" and the certificate length is at most $|t| \leq q(|s|)$ for some polynomials p and q
- A decision problem is solvable in non-deterministic polynomial time, if such a polynomial time certifier C exists ($X \in NP$)

Example

- The decision problem, if a Boolean expression is satisfiable is solvable in non-deterministic polynomial time

- Why?
- Because if it is, we can guess a valid assignment and simply verify the solution in polynomial time

- We proof a problem $X \in NP$ by constructing a polynomial time certifier that can verify a correct solution

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4 \vee x_5) \wedge (x_1 \vee \neg x_5 \vee x_6)$$

$$\{x_1 = T, x_4 = T, x_6 = T\}$$

P and NP

- A decision problem $X \in P$ can be **solved** in polynomial time
- A solution for a decision problem $X \in NP$ can be **verified** in polynomial time
- Nobody knows, if these two complexity classes are the same or not ($P = NP$?)

Reduction

- A decision problem Y can be **reduced** to the decision problem X if there is a polynomial time algorithm R that transforms an input of Y into an input of X so that

$$y \in Y \Leftrightarrow R[y] \in X$$

- Therefore, if we know how to solve X we also know how to solve Y
- In particular, $X \in P \Rightarrow Y \in P$ and also $X \in NP \Rightarrow Y \in NP$
- Or in other words, X is at least as hard as Y

NP-hard & NP-complete

- A decision problem X is **NP-hard**, if every problem $Y \in NP$ reduces to X
- Therefore X is at least as hard as every problem that is solvable in non-deterministic polynomial time

- A decision problem X is **NP-complete**, if X is NP-hard and also $X \in NP$ itself
- This means, **all NP-complete problems are all equally hard** in the sense that every NP-complete problem can be reduced to every other NP-complete problem in polynomial time

Proving NP-completeness

- Because NP-complete problems are the hardest problems in NP, to prove that a new problem X is also NP-hard, all we need to do is find a reduction of X to any problem that we know is NP-complete
- If we can also show that $X \in NP$ we identified a new NP-complete problem

- Obviously, someone needed to find a first NP-complete problem to which <u>all</u> other problems in NP can be reduced in polynomial time to bootstrap this approach
- Cook's Theorem proved that the SAT problem is NP-complete, from which all other know NP-complete problems followed

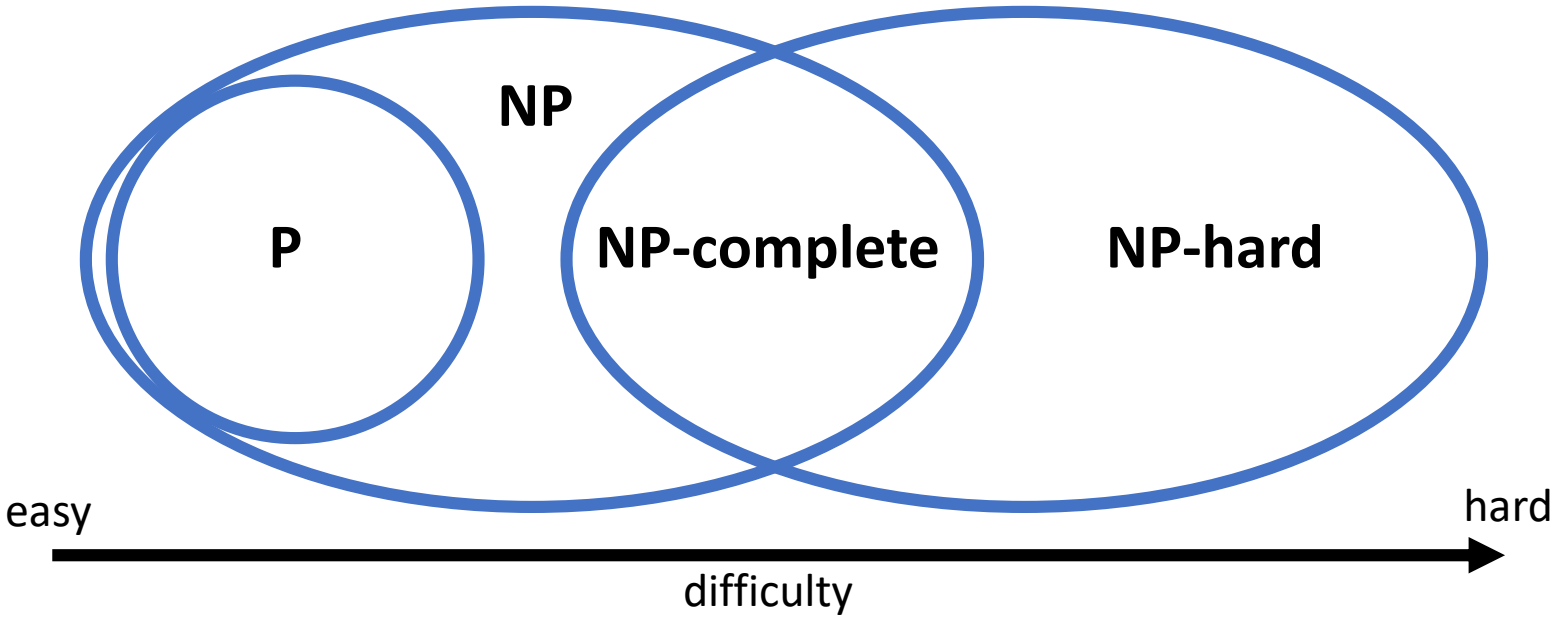
P=NP?

- Although many think it is unlikely, it is not known if the class of non-deterministic polynomial time decision problems can be solved in deterministic polynomial time

- If a polynomial time algorithm for any NP-complete problem X would be found, all problems in NP could be reduced to X and therefore solved in polynomial time ($P = NP$)

- However, if this is not the case ($P \neq NP$) then no NP-hard problem can be solved in deterministic polynomial time, i.e. $X \notin P$ for all problems X that are NP-hard

Complexity classes



- **P:** problems that can be solved in polynomial time
- **NP:** problems, for which the solution can be verified in polynomial time

- **NP-hard:** problems, to which all problems in NP can be reduced to in polynomial time
- **NP-complete:** problems in NP, to which all problems in NP can be reduced to in polynomial time

Implications for solving hard problems

- Many problems that we will be looking into in this lecture are NP-complete (e.g. the Knapsack problem)

- Unless P=NP (which is an unsolved problem of computer science) the best possible algorithm for any of these problems has **exponential running-time**, i.e. essentially requires to explore the complete search space
- For the Knapsack problem with 100 items this is $2^{100} > 10^{30}$

- If anybody found an algorithm that could solve a problem in NP, this would mean P=NP. As modern cryptography heavily depends on the assumption that this is not the case, we would have probably heard about this.

- The combination of good modelling choices, propagation and search strategies provide surprisingly good solutions for many real world problems
- **However: There can always be instances where this is not the case and where the approach we implemented works well for one instance and does not work for another.**

Soundness and completeness

- An algorithm is **sound**, if every result it produces is valid
- An algorithm is **complete**, if it is guaranteed to produce a result on all inputs

- A trivial sound algorithm can be achieved by simply never returning any result
- A trivial complete algorithm can be achieved by simply returning a wrong result regardless of the input

- Ideally, our algorithms should be both

Soundness and completeness

- An algorithm is **sound**, if every result it produces is valid
- An algorithm is **complete**, if it is guaranteed to produce a result on all inputs

- However, given the complexity of the problems we are trying to solve we might be willing to accept
  - Not sound: Having only an approximate result that is technically not 100% valid but only contradicts the constraints a little or isn't the global maximum of our objective function

  - Not complete: Having a solution that does not work on all input data, but works well on the typical input data we are seeing in the specific problem domain we are trying to solve

Limiting computation time

- There always exist instances on which we do not expect to find the solution in a reasonable time

- We can therefore limit computation time and work with the best result that we can obtain within this limit

```
solver.parameters.max_time_in_seconds = 10.0
```

- The algorithm in this case is
  - **neither sound,** because it might not be the optimal solution,
  - **nor complete,** because we might not have come to any solution in time

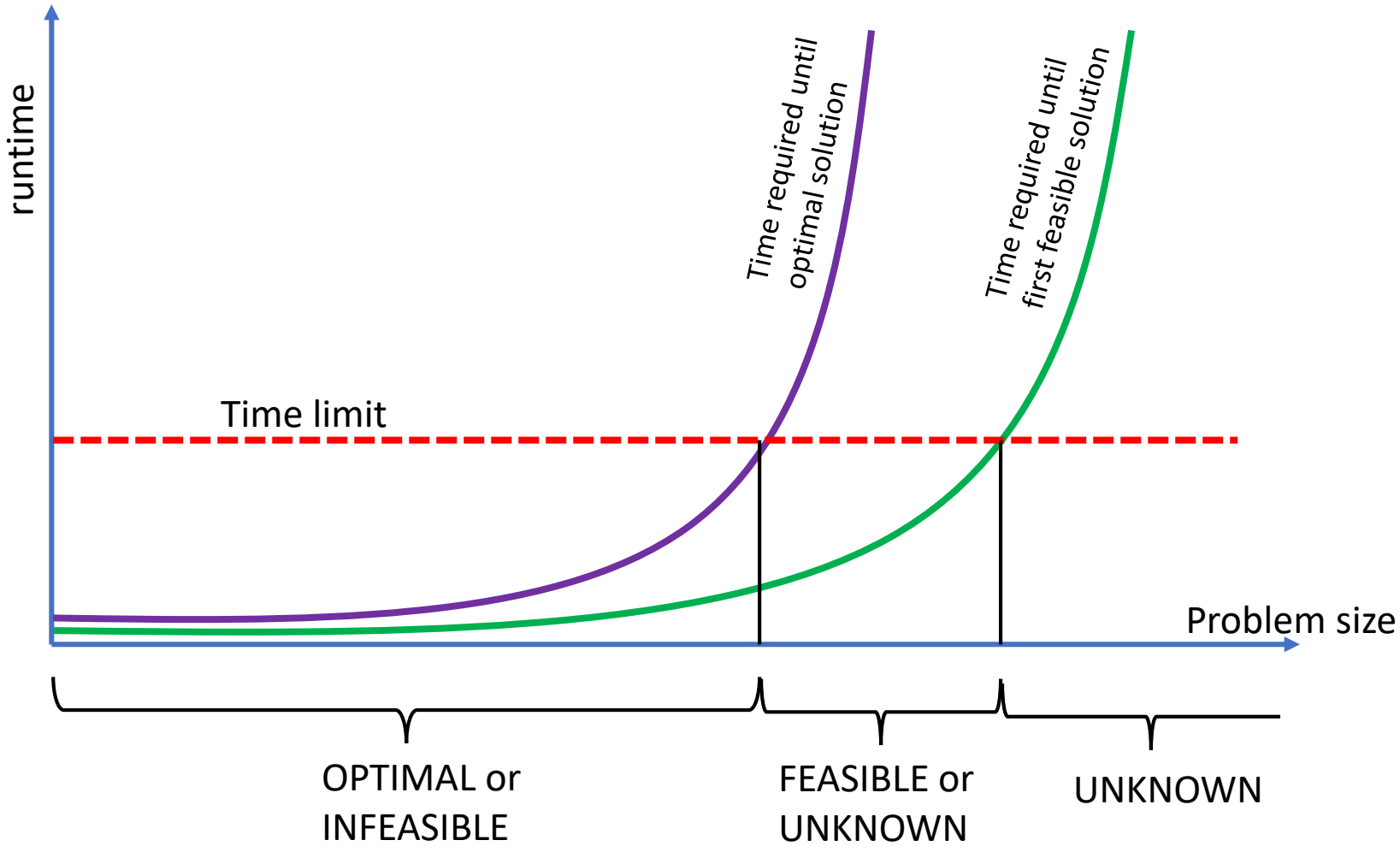
Limiting computation time

- We now need to check the status of the solution that we obtained

```
status = solver.Solve(model)
```

Status	Description
OPTIMAL	An optimal feasible solution was found.
FEASIBLE	A feasible solution was found, but we don't know if it's optimal.
INFEASIBLE	The problem was proven infeasible.
MODEL_INVALID	The given CpModelProto didn't pass the validation step. You can get a detailed error by calling ValidateCpModel(model_proto).
UNKNOWN	The status of the model is unknown because a search limit was reached.

Limiting computation time



Thank you for your attention!