CS 341: ALGORITHMS

Lecture 24: intractability VI – Decidability, more NPC transformations

Readings: see website

Trevor Brown
https://student.cs.uwaterloo.ca/~cs341
trevor.brown@uwaterloo.ca

1

COMPLEXITY CLASS **EXPTIME**

A very brief overview

(Non-core material)

EXPTIME is the set of all **decision problems** that can be solved in **exponential time**.
I.e., in time $O(2^{poly(n)})$ where $poly(n)$ is a polynomial in the input size.

Observe that $\boldsymbol{NP \subseteq EXPTIME}$

The idea is to generate all possible certificates of an appropriate length and check them for correctness using the given certificate verification algorithm. An an example, for **Hamiltonian Cycle**, we could generate all $n!$ certificates and check each one in turn.

$O(n!) \subseteq O(n^n) = O(2^{n \log n})$ time

We do not know if there are problems in **NP** that cannot be solved in polynomial time (because the **P** = **NP**? conjecture is not yet resolved). However, it is possible to prove that there exist problems in **EXPTIME** \ **P**.

One such problem is the **Bounded Halting** problem. Here an instance $I = (A, x, t)$, where A is a program, x is an input to A, and t is a positive integer (in binary). The question to be solved is if $A(x)$ halts after at most t computation steps.

The **Bounded Halting** problem can be solved in time $O(t)$, but this is not a polynomial time algorithm because size$(I) = |A| + |x| + \log_2 t$.

Actually, it can be proven that **Bounded Halting** is EXPTIME-complete. This implies that it is in **EXPTIME** \ **P**, since it is known that **EXPTIME** $\neq$ **P**.

***t* is exponential in $\log t$.**
(And $\log t$ might be the largest term in the input size, in which case $O(t)$ would be **exponential in the input size**.)

UNDECIDABILITY

Problems that are ***impossible*** to solve

5

DECIDABLE VS **UNDECIDABLE** PROBLEMS

We say an algorithm A "solves" a decision problem if, for **every** instance I, $A(I)$ has **finite** runtime and returns the correct answer

If an algorithm A **solves** decision problem Π, then we say Π is **decidable**.

Formally, Π is **decidable IFF** there exists some algorithm A such that, for **every** instance I, $A(I)$ returns the correct answer in **finite** time.

If it is ***not possible*** to design an algorithm A that **solves** decision problem Π, then we say Π is **undecidable**.

Formally, Π is **undecidable IFF** there **cannot exist** an algorithm A such that, for every instance I, $A(I)$ returns the correct answer in **finite time**.

Equivalently, Π is **undecidable IFF**, for every algorithm A, there exists some input I such that $A(I)$ **does not** return the correct answer in finite time.

I.e., for some input, $A(I)$ either runs forever or returns the wrong answer

6

HALTING: AN UNDECIDABLE PROBLEM

Problem 7.19
Halting
Instance: *A computer program A and input x for the program A.*
Question: *When program A is executed with input x, will it halt in finite time?*

For example, you could run $Halt(BFS, G)$ to determine whether, $BFS(G)$ will halt in finite time, which it will, so $Halt(BFS, G)$ returns yes.

The **Halting** problem is **decidable** **IFF** there **exists an algorithm** $Halt(I)$ that, for **every** instance $I = (A, x)$, $Halt(I)$ has **finite** runtime and correctly answers the question: "would a call to $A(x)$ halt in finite time?"

UNDECIDABILITY OF THE HALTING PROBLEM

Suppose that *Halt* is a program that solves the **Halting Problem**.

We suppose Halt **exists**, to obtain a **contradiction**…

The statement "*Halt* solves the Halting problem" means that *Halt* runs in finite time, and:

$$Halt(A, x) = \begin{cases} \textbf{true} & \text{if } A(x) \text{ halts} \\ \textbf{false} & \text{if } A(x) \text{ doesn't halt.} \end{cases}$$

Note that A (the "algorithm") and x (the "input" to A) are both strings over some finite alphabet.

Since A **is a string** (of code), and its input x **is also a string**… we ***could*** pass A as an argument to itself: $A(A)$

Then we could ask **if $A(A)$ halts**, by running $Halt(A, A)$…

Weird… Let's try to obtain a contradiction by doing this…

Consider the following algorithm *Strange*.

Algorithm: $Strange(A)$
external *Halt*
if not $Halt(A, A)$ — If **not $Halt(A,A)$**, then $A(A)$ **will run forever**
then return (!) — but $Strange(A)$ **terminates in finite time**
else $i \leftarrow 1$ — Else if **$Halt(A,A)$**, then $A(A)$ **will terminate in finite time**
while $i \neq 0$ **do** $i \leftarrow i + 1$ — But $Strange(A)$ **will run forever**

What happens when we run $Strange(Strange)$?

Two cases: $Strange(Strange)$ either **halts** or **does not halt**

Suppose $Strange(Strange)$ halts. Then, it must return. This means it sees *not* $Halt(A, A)$ just before returning.

But $A = Strange$, so it sees *not* $Halt(Strange, Strange)$.

So, **$Strange(Strange)$ does not halt --- contradiction!**

Suppose $Strange(Strange)$ does not halt. Then, it must spin in the while loop forever. This means $Halt(A, A) = true$.

But $A = Strange$, so $Halt(Strange, Strange) = true$.

So, **$Strange(Strange)$ halts --- contradiction!**

Both cases lead to a contradiction. So, our only assumption, **that Halt exists**, must be false!

Therefore, the Halting problem is **undecidable**.

Another Undecidable Problem

Here is another example of an undecidable problem. The problem **Halt-All** takes a program A as input and asks if A halts on all inputs x.

We describe a Turing reduction **Halting** $\leq^T$ **Halt-All**, which proves that **Halt-All** is undecidable.

Assume we have a program *HaltAllSolver*.

For a fixed program A and input x, let $B_x()$ be the program that executes $A(x)$ (so B_x has no input).

Here is the reduction:
Given A and x (an instance of **Halting**), construct the program B_x.
Run $HaltAllSolver(B_x)$.

We have

$$HaltAllSolver(B_x) = \textbf{true} \Leftrightarrow A(x) \text{ halts},$$

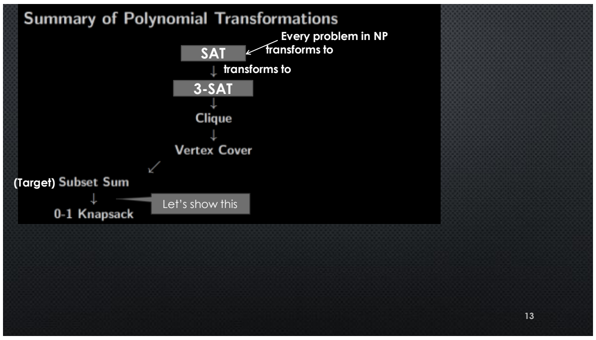
so we can solve the halting problem.

If we have *HaltAllSolver* then we have *Halt*, but this is impossible, so *HaltAllSolver* cannot exist, so **the Halt-All problem is undecidable**!

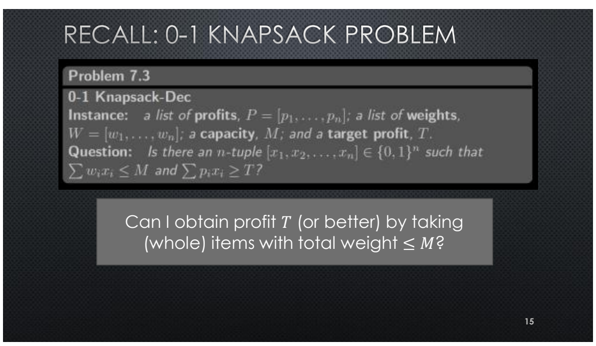
FINISHING NPC TRANSFORMATIONS/REDUCTIONS

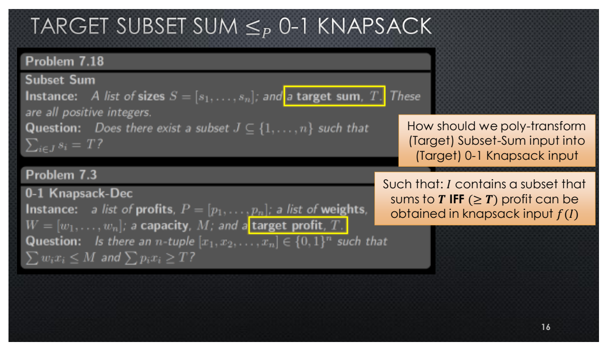
Summary of Polynomial Transformations

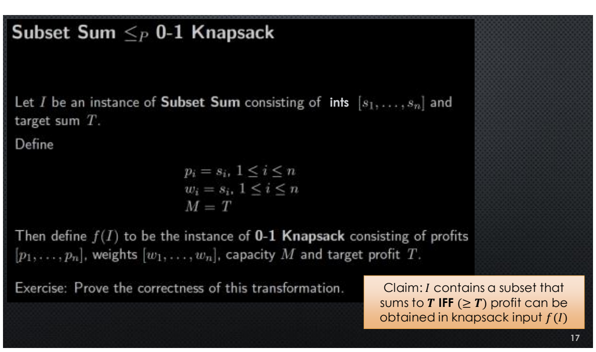
Every problem in NP transforms to
SAT
↓ transforms to
3-SAT
↓
Clique
↓
Vertex Cover
↙
Subset Sum

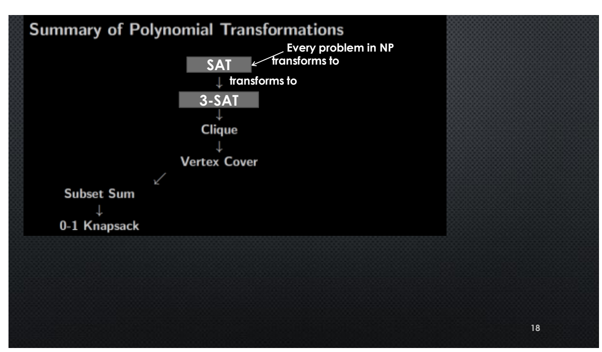

Summary of Polynomial Transformations
Every problem in NP transforms to
SAT
transforms to
3-SAT
Clique
Vertex Cover
(Target) Subset Sum
0-1 Knapsack
Let's show this
13

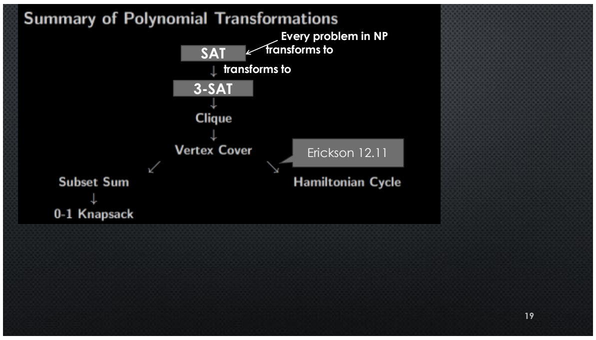

REDUCE TARGET SUBSET SUM
TO 0-1 KNAPSACK
14

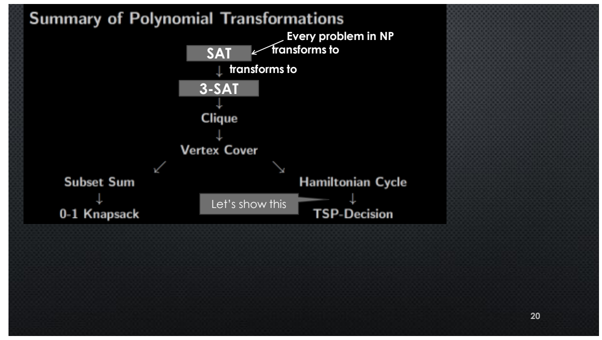

RECALL: 0-1 KNAPSACK PROBLEM
Problem 7.3
0-1 Knapsack-Dec
Instance: a list of profits, $P = [p_1, \ldots, p_n]$; a list of weights, $W = [w_1, \ldots, w_n]$; a capacity, M; and a target profit, T.
Question: Is there an n-tuple $[x_1, x_2, \ldots, x_n] \in \{0,1\}^n$ such that $\sum w_i x_i \leq M$ and $\sum p_i x_i \geq T$?
Can I obtain profit T (or better) by taking (whole) items with total weight $\leq M$?
15


TARGET SUBSET SUM $\leq_P$ 0-1 KNAPSACK
Problem 7.18
Subset Sum
Instance: A list of sizes $S = [s_1, \ldots, s_n]$; and a target sum, T. These are all positive integers.
Question: Does there exist a subset $J \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in J} s_i = T$?
How should we poly-transform (Target) Subset-Sum input into (Target) 0-1 Knapsack input
Problem 7.3
0-1 Knapsack-Dec
Instance: a list of profits, $P = [p_1, \ldots, p_n]$; a list of weights, $W = [w_1, \ldots, w_n]$; a capacity, M; and a target profit, T
Question: Is there an n-tuple $[x_1, x_2, \ldots, x_n] \in \{0,1\}^n$ such that $\sum w_i x_i \leq M$ and $\sum p_i x_i \geq T$?
Such that: I contains a subset that sums to T IFF ($\geq T$) profit can be obtained in knapsack input $f(I)$
16

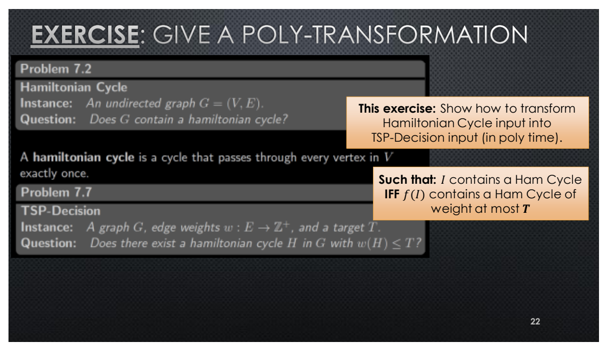

Subset Sum $\leq_P$ 0-1 Knapsack
Let I be an instance of Subset Sum consisting of ints $[s_1, \ldots, s_n]$ and target sum T.
Define
$p_i = s_i, 1 \leq i \leq n$
$w_i = s_i, 1 \leq i \leq n$
$M = T$
Then define $f(I)$ to be the instance of 0-1 Knapsack consisting of profits $[p_1, \ldots, p_n]$, weights $[w_1, \ldots, w_n]$, capacity M and target profit T.
Exercise: Prove the correctness of this transformation.
Claim: I contains a subset that sums to T IFF ($\geq T$) profit can be obtained in knapsack input $f(I)$
17

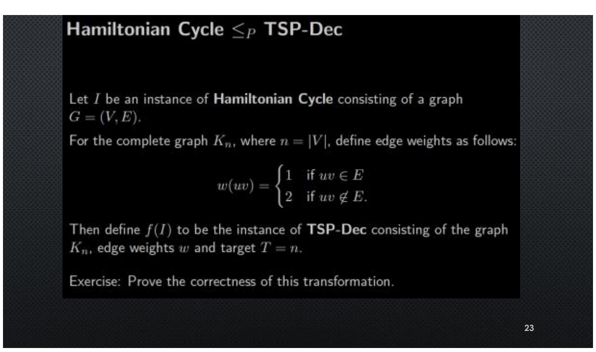

Summary of Polynomial Transformations
Every problem in NP transforms to
SAT
transforms to
3-SAT
Clique
Vertex Cover
Subset Sum
0-1 Knapsack
18

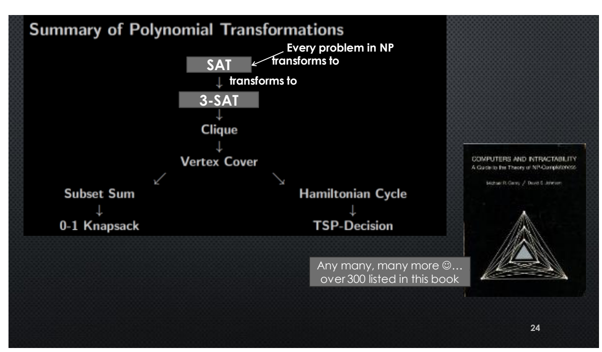

Summary of Polynomial Transformations
SAT
Every problem in NP transforms to
transforms to
3-SAT
Clique
Vertex Cover
Erickson 12.11
Subset Sum
Hamiltonian Cycle
0-1 Knapsack
19


Summary of Polynomial Transformations
SAT
Every problem in NP transforms to
transforms to
3-SAT
Clique
Vertex Cover
Subset Sum
Hamiltonian Cycle
0-1 Knapsack
Let's show this
TSP-Decision
20


REDUCE **HAMILTONIAN CYCLE** TO **TSP-DECISION**
21


EXERCISE: GIVE A POLY-TRANSFORMATION
Problem 7.2
Hamiltonian Cycle
Instance: An undirected graph $G = (V, E)$.
Question: Does G contain a hamiltonian cycle?
This exercise: Show how to transform Hamiltonian Cycle input into TSP-Decision input (in poly time).
A **hamiltonian cycle** is a cycle that passes through every vertex in V exactly once.
Problem 7.7
TSP-Decision
Instance: A graph G, edge weights $w : E \to \mathbb{Z}^+$, and a target T.
Question: Does there exist a hamiltonian cycle H in G with $w(H) \leq T$?
Such that: I contains a Ham Cycle **IFF** $f(I)$ contains a Ham Cycle of weight at most T
22


Hamiltonian Cycle $\leq_P$ TSP-Dec
Let I be an instance of **Hamiltonian Cycle** consisting of a graph $G = (V, E)$.
For the complete graph K_n, where $n = |V|$, define edge weights as follows:
$$w(uv) = \begin{cases} 1 & \text{if } uv \in E \\ 2 & \text{if } uv \notin E. \end{cases}$$
Then define $f(I)$ to be the instance of **TSP-Dec** consisting of the graph K_n, edge weights w and target $T = n$.
Exercise: Prove the correctness of this transformation.
23

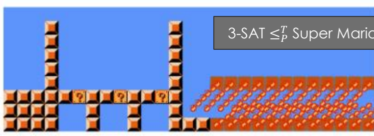

Summary of Polynomial Transformations
SAT
Every problem in NP transforms to
transforms to
3-SAT
Clique
Vertex Cover
Subset Sum
Hamiltonian Cycle
0-1 Knapsack
TSP-Decision
COMPUTERS AND INTRACTABILITY
Any many, many more ☺… over 300 listed in this book
24

FUN AND GAMES

https://arxiv.org/pdf/1203.1895.pdf

Abstract

We prove NP-hardness results for five of Nintendo's largest video game franchises: Mario, Donkey Kong, Legend of Zelda, Metroid, and Pokémon. Our results apply to generalized versions of Super Mario Bros. 1–3, The Lost Levels, and Super Mario World; Donkey Kong Country 1–3; all Legend of Zelda games; all Metroid games; and all Pokémon role-playing games. In addition, we prove PSPACE-completeness of the Donkey Kong Country games and several Legend of Zelda games.



3-SAT $\leq_P^T$ Super Mario

Figure 11: Clause gadget for Super Mario Bros.

25

(FACTUALLY INCORRECT) MEMES

- There's also an old video meme about proving that Super Mario Bros is NP complete
  - (Long before it was legitimately proved NP hard ☺)
- Whereas the stuff on the **previous slide is real math**, the stuff in this video is just a meme, and is **very wrong**. but you may find it funny…

26

SUMMARY OF COMPLEXITY CLASSES

See this slide's notes

- **P** (Poly-time) — E.g., (**decision** problem variants of:) BFS, Dijkstra's, **some** DP algorithms
  - **Decision** problems that can be solved by algorithms with runtime poly(input size)
- **NP** (Non-deterministic poly-time) — **All of P**, and e.g.,, vertex cover, clique, SAT, subset sum
  - **Decision** problems for which **certificates** can be **verified** in time poly(input size)
  - Equivalently: decision problems that can be solved in poly-time if you have access to a non-deterministic oracle that returns a yes-certificate if one exists
- **NPC** (NP-complete) — E.g., vertex cover, clique, SAT, subset sum, TSP-decision
  - **Decision** problems $\Pi \in \boldsymbol{NP}$ s.t. every $\Pi' \in NP$ can be **transformed** to Π in poly-time
- **NP-hard** (at least as hard as NPC) — **All of NPC**, and e.g., TSP-optimization, TSP-optimal value
  - problems Π s.t. every $\Pi' \in NP$ can be **reduced** to Π in poly-time
- Note: P, NP and NPC problems are **decidable**

27

POLYTIME 2-SAT

(IF WE HAVE TIME)

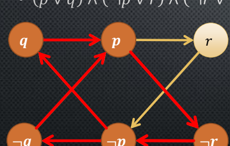
28

2-SAT EXAMPLES

- $(p \vee q) \wedge (\neg p \vee r) \wedge (\neg r \vee \neg p)$
  - Satisfiable: $p = 0, q = 1, r \in \{0,1\}$
- $(p \vee q) \wedge (\neg p \vee r) \wedge (\neg r \vee \neg p) \wedge (p \vee \neg q)$

Logical refresher: $p \Rightarrow q$ is **equivalent** to $\neg p \vee q$.

Therefore, $p \vee q$ is **equivalent** to $\neg p \Rightarrow q$ **and equivalent** to $\neg q \Rightarrow p$



Edges (implications of clauses)…

$\neg p \Rightarrow q$	$p \Rightarrow r$	$r \Rightarrow \neg p$	$\neg p \Rightarrow \neg q$
$\neg q \Rightarrow p$	$\neg r \Rightarrow \neg p$	$p \Rightarrow \neg r$	$q \Rightarrow p$

$q \Rightarrow p \Rightarrow \neg r \Rightarrow \neg p \Rightarrow \neg q$ … so q cannot be $true$

$\neg q \Rightarrow p \Rightarrow \neg r \Rightarrow \neg p \Rightarrow q$ … so q cannot be $false$

Therefore the formula **cannot** be satisfied!

29

(variable names are integers in 1..|X|)

2-SAT can be solved in polynomial time. Suppose we are given an instance I of **2-SAT** on a set of boolean variables $\boldsymbol{X = \{1..|X|\}}$

(1) For every clause $x \vee y$ (where x and y are literals), construct two directed edges $\overline{x}y$ and $\overline{y}x$. We get a directed graph on vertex set $X \cup \overline{X}$.

(2) Determine the strongly connected components of this directed graph.

(3) I is a yes-instance if and only if there is no strongly connected component containing x and $\overline{x}$, for any $x \in X$.

Suppose no variable x is in the same SCC as $\bar{x}$, then to get a satisfying assignment do the following:
For each x, if ∃ path from x to $\bar{x}$, then set $x = false$ else set $x = true$.

30