Intractability and Reductions

Lecture 19

April 15, 2021

Course Outline

- Part I: models of computation (reg exps, DFA/NFA, CFGs, TMs)
- Part II: (efficient) algorithm design
- Part III: intractability via reductions
  - Undecidablity: problems that have no algorithms
  - NP-Completeness: problems unlikely to have efficient algorithms unless $\boldsymbol{P} = \boldsymbol{NP}$

Part I

Intractability and Lower Bounds

Turing Machines and Church-Turing Thesis

Turing defined TMs as a machine model of computation

Church-Turing thesis: any function that is computable can be computed by TMs

Efficient Church-Turing thesis: any function that is computable can be computed by TMs with only a polynomial slow-down

Computability and Complexity Theory

- What functions can and *cannot* be computed by TMs?
- What functions/problems can and cannot be solved *efficiently*?

Why?

- Foundational questions about computation
- Pragmatic: Can we solve our problem or not?
- Are we not being clever enough to find an efficient algorithm or should we stop because there isn't one or likely to be one?

Lower Bounds and Impossibility Results

Prove that given problem cannot be solved (efficiently) on a TM. Informally we say that the problem is "hard".

Generally quite difficult: algorithms can be very non-trivial and clever.

Example: The famous $P \neq NP$ conjecture.

Reductions to Prove Intractability

A general methodology to prove impossibility results.

- Start with some *known* hard problem X
- *Reduce* X to your favorite problem Y

If Y can be solved then so can $X \Rightarrow Y$ is also *hard*

Reductions to Prove Intractability

A general methodology to prove impossibility results.

- Start with some *known* hard problem X
- *Reduce* X to your favorite problem Y

If Y can be solved then so can $X \Rightarrow Y$ is also *hard*

Caveat: In algorithms we reduce new problem to known solved one!

Reductions to Prove Intractability

A general methodology to prove impossibility results.

- Start with some *known* hard problem X
- *Reduce* X to your favorite problem Y

If Y can be solved then so can X $\Rightarrow$ Y is also *hard*

Caveat: In algorithms we reduce new problem to known solved one!

Who gives us the initial hard problem?

- Some clever person (Cantor/Gödel/Turing/Cook/Levin ...) who establish hardness of a fundamental problem
- Assume some core problem is hard because we haven't been able to solve it for a long time. This leads to *conditional* results

Reductions to Prove Intractability

A general methodology to prove impossibility results.

- Start with some *known* hard problem X
- *Reduce* X to your favorite problem Y

If Y can be solved then so can $X \Rightarrow Y$ is also *hard*

Caveat: In algorithms we reduce new problem to known solved one!

Who gives us the initial hard problem?

- Some clever person (Cantor/Gödel/Turing/Cook/Levin ...) who establish hardness of a fundamental problem
- Assume some core problem is hard because we haven't been able to solve it for a long time. This leads to *conditional* results

Reduction is a powerful and unifying tool in Computer Science

Decision Problems, Languages, Terminology

When proving hardness we limit attention to *decision* problems

- A decision problem Π is a collection of instances (strings)
- For each instance I of Π, answer is YES or NO
- Equivalently: boolean function $f_\Pi : \Sigma^* \to \{0, 1\}$ where $f(I) = 1$ if I is a YES instance, $f(I) = 0$ if NO instance
- Equivalently: language $L_\Pi = \{I \mid I \text{ is a YES instance}\}$

Decision Problems, Languages, Terminology

When proving hardness we limit attention to *decision* problems

- A decision problem Π is a collection of instances (strings)
- For each instance I of Π, answer is YES or NO
- Equivalently: boolean function $f_\Pi : \Sigma^* \to \{0, 1\}$ where $f(I) = 1$ if I is a YES instance, $f(I) = 0$ if NO instance
- Equivalently: language $L_\Pi = \{I \mid I \text{ is a YES instance}\}$

Notation about encoding: distinguish I from encoding $\langle I \rangle$

- n is an integer. $\langle n \rangle$ is the encoding of n in some format (could be unary, binary, decimal etc)
- G is a graph. $\langle G \rangle$ is the encoding of G in some format
- M is a TM. $\langle M \rangle$ is the encoding of TM as a string according to some fixed convention

Examples

- Given directed graph G, is it strongly connected? $\langle G \rangle$ is a YES instance if it is, otherwise NO instance
- Given number n, is it a prime number?
  $L_{PRIMES} = \{\langle n \rangle \mid n \text{ is prime}\}$
- Given number n is it a composite number?
  $L_{COMPOSITE} = \{\langle n \rangle \mid n \text{ is a composite}\}$
- Given $G = (V, E), s, t, B$ is the shortest path distance from s to t at most B? Instance is $\langle G, s, t, B \rangle$

Part II

(Polynomial Time) Reductions

Reductions for decision problems/languages

For languages L_X, L_Y, a **reduction from L_X to L_Y** is:

1. An algorithm ...
2. Input: $w \in \Sigma^*$
3. Output: $w' \in \Sigma^*$
4. Such that:

$$\boxed{w \in L_Y} \iff \boxed{w' \in L_X}$$

Reductions for decision problems/languages

For languages L_X, L_Y, a **reduction from L_X to L_Y** is:

1. An algorithm ...
2. Input: $w \in \Sigma^*$
3. Output: $w' \in \Sigma^*$
4. Such that:

$$\boxed{w \in L_Y} \iff \boxed{w' \in L_X}$$

(Actually, this is only one type of reduction, but this is the one we will use for hardness.) There are other kinds of reductions.

Reductions for decision problems/languages

For decision problems X, Y, a **reduction from X to Y** is:

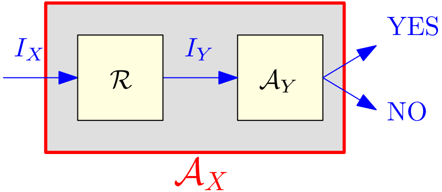
1. An algorithm ...
2. Input: I_X, an instance of X.
3. Output: I_Y an instance of Y.
4. Such that:

$$\boxed{I_Y \text{ is YES instance of } Y} \iff \boxed{I_X \text{ is YES instance of } X}$$

Reductions

1. $\mathcal{R}$: Reduction $X \to Y$
2. $\mathcal{A}_Y$: algorithm for Y:
3. $\implies$ New algorithm for X:

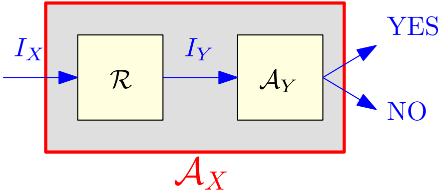
$\mathcal{A}_X(I_X)$:
  // I_X: instance of X.
  $I_Y \Leftarrow \mathcal{R}(I_X)$
  return $\mathcal{A}_Y(I_Y)$



Reductions

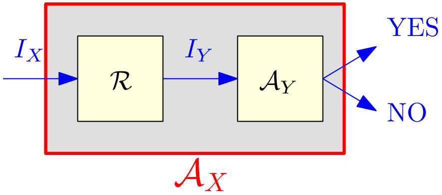
1. $\mathcal{R}$: Reduction $X \to Y$
2. $\mathcal{A}_Y$: algorithm for Y:
3. $\implies$ New algorithm for X:

```
A_X(I_X):
        // I_X: instance of X.
        I_Y ⇐ R(I_X)
        return A_Y(I_Y)
```



If $\mathcal{R}$ and $\mathcal{A}_Y$ polynomial-time $\implies$ $\mathcal{A}_X$ polynomial-time.

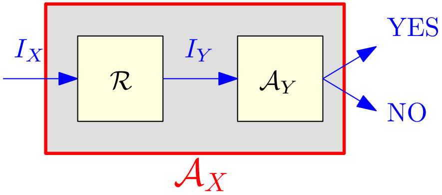
Reductions and running time



$R(n)$: running time of $\mathcal{R}$
$Q(n)$: running time of $\mathcal{A}_Y$

Question: What is running time of $\mathcal{A}_X$?

Reductions and running time



$R(n)$: running time of $\mathcal{R}$
$Q(n)$: running time of $\mathcal{A}_Y$

Question: What is running time of $\mathcal{A}_X$? $O(R(n) + Q(R(n))$.
Why?

- If I_X has size n, $\mathcal{R}$ creates an instance I_Y of size at most $R(n)$
- $\mathcal{A}_\mathcal{Y}$'s time on I_Y is by definition at most $Q(|I_Y|) \leq Q(R(n))$.

Example: If $R(n) = n^2$ and $Q(n) = n^{1.5}$ then $\mathcal{A}_X$ is $O(n^3)$

Notation and Implication of Reductions

1. If Problem X reduces to Problem Y we write $X \leq Y$
2. If Problem X reduces to Problem Y where reduction $\mathcal{R}$ is an efficient (polynomial-time algorithm) we write $X \leq_P Y$.

Notation and Implication of Reductions

1. If Problem X reduces to Problem Y we write $X \leq Y$
2. If Problem X reduces to Problem Y where reduction $\mathcal{R}$ is an efficient (polynomial-time algorithm) we write $X \leq_P Y$.

Algorithmic implication:

Lemma

- *If $X \leq Y$ and Y has an algorithm then X has an algorithm.*
- *If $X \leq_P Y$ and Y has a polynomial-time algorithm then X has a polynomial-time algorithm.*

Hardness Implications of Reductions

1. Problem X reduces to Problem Y: $X \leq Y$
2. Problem X efficiently reduces to Problem Y: $X \leq_P Y$.

Hardness implication:

Lemma

- *If $X \leq Y$ **and** X does not have an algorithm then Y does not have an algorithm.*
- *If $X \leq_P Y$ **and** X does not have a polynomial-time algorithm then Y does not have a polynomial-time algorithm.*

Hardness Implications of Reductions

1. Problem X reduces to Problem Y: $X \leq Y$
2. Problem X efficiently reduces to Problem Y: $X \leq_P Y$.

Hardness implication:

Lemma

- *If $X \leq Y$ **and** X does not have an algorithm then Y does not have an algorithm.*
- *If $X \leq_P Y$ **and** X does not have a polynomial-time algorithm then Y does not have a polynomial-time algorithm.*

Proof.

Suppose Y has an algorithm. Then X does too since $X \leq Y$. But contradicts assumption that X does not have an algorithm. Similarly for efficient reduction. □

Transitivity of Reductions

Proposition

$X \leq Y$ and $Y \leq Z$ implies that $X \leq Z$. Similarly $X \leq_P Y$ and $Y \leq_P Z$ implies $X \leq_P Z$.

Note: $X \leq Y$ does not imply that $Y \leq X$ and hence it is very important to know the FROM and TO in a reduction.

Proving Correctness of Reductions

To prove that $X \leq Y$ you need to give an **algorithm** $\mathcal{A}$ that:

1. Transforms an instance I_X of X into an instance I_Y of Y.
2. Satisfies the property that answer to I_X is YES iff I_Y is YES.
   1. typical easy direction to prove: answer to I_Y is YES if answer to I_X is YES
   2. typical difficult direction to prove: answer to I_X is YES if answer to I_Y is YES (equivalently answer to I_X is NO if answer to I_Y is NO).
3. To prove $X \leq_P Y$, additionally show that $\mathcal{A}$ runs in **polynomial** time.

Remember, remember, remember

- Algorithm design: reduce new problem X to *known easy* problem Y
- Hardness: reduce *known hard* problem X to new problem Y

Tools to remember:

- Am I trying to design algorithm or prove hardness?
- What do I know about some standard problems? Easy or hard?

Part III

Examples of Reductions

Independent Sets and Cliques

Given a graph G, a set of vertices V' is:

Independent Sets and Cliques

Given a graph G, a set of vertices V' is:

1. **independent set**: no two vertices of V' connected by an edge.

Independent Sets and Cliques

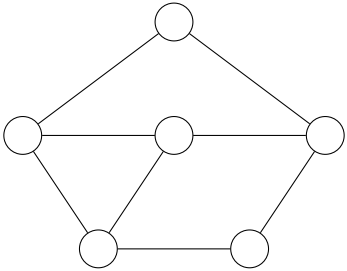
Given a graph G, a set of vertices V' is:

1. **independent set**: no two vertices of V' connected by an edge.
2. **clique**: *every* pair of vertices in V' is connected by an edge of G.

Independent Sets and Cliques

Given a graph G, a set of vertices V' is:

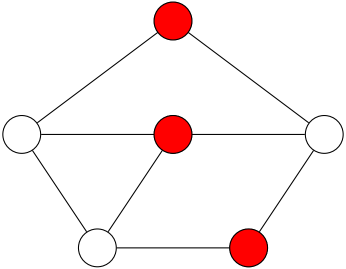
1. **independent set**: no two vertices of V' connected by an edge.
2. **clique**: *every* pair of vertices in V' is connected by an edge of G.



Independent Sets and Cliques

Given a graph G, a set of vertices V' is:

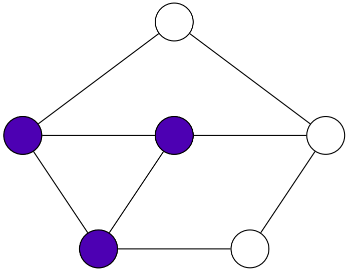
1. **independent set**: no two vertices of V' connected by an edge.
2. **clique**: *every* pair of vertices in V' is connected by an edge of G.



Independent Sets and Cliques

Given a graph G, a set of vertices V' is:

1. **independent set**: no two vertices of V' connected by an edge.
2. **clique**: *every* pair of vertices in V' is connected by an edge of G.



The Independent Set and Clique Problems

Problem: Independent Set

Instance: A graph G and an integer k.
Question: Does G has an independent set of size $\geq k$?

The Independent Set and Clique Problems

Problem: Independent Set

Instance: A graph G and an integer k.
Question: Does G has an independent set of size $\geq k$?

Problem: Clique

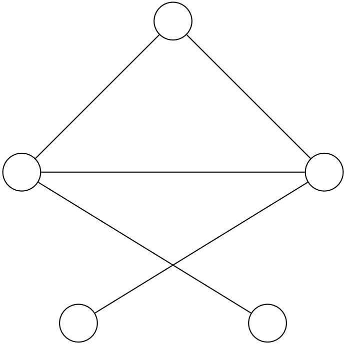
Instance: A graph G and an integer k.
Question: Does G has a clique of size $\geq k$?

Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph G and an integer k.

Reducing **Independent Set** to **Clique**

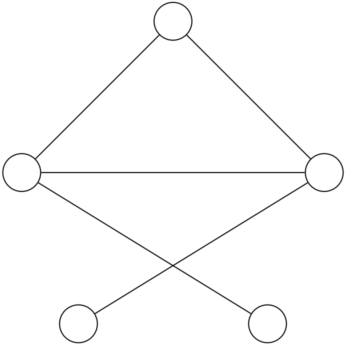
An instance of **Independent Set** is a graph G and an integer k.



Reducing Independent Set to Clique

An instance of **Independent Set** is a graph G and an integer k.

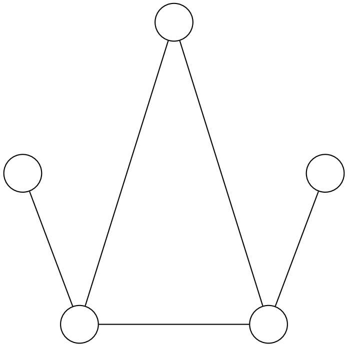
Reduction given $\langle G, k \rangle$ outputs $\langle \overline{G}, k \rangle$ where $\overline{G}$ is the *complement* of G. $\overline{G}$ has an edge (u, v) if and only if (u, v) is not an edge of G.



Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph G and an integer k.

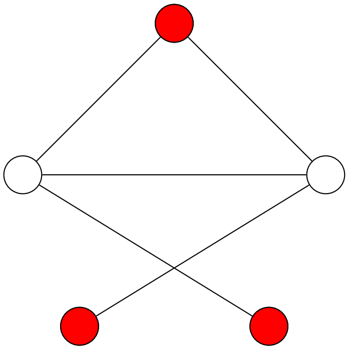
Reduction given $\langle G, k \rangle$ outputs $\langle \overline{G}, k \rangle$ where $\overline{G}$ is the *complement* of G. $\overline{G}$ has an edge (u, v) if and only if (u, v) is not an edge of G.



Reducing Independent Set to Clique

An instance of **Independent Set** is a graph G and an integer k.

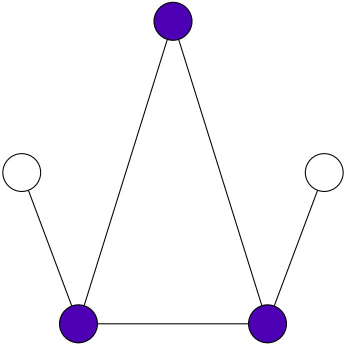
Reduction given $\langle G, k \rangle$ outputs $\langle \overline{G}, k \rangle$ where $\overline{G}$ is the *complement* of G. $\overline{G}$ has an edge (u, v) if and only if (u, v) is not an edge of G.



Reducing Independent Set to Clique

An instance of **Independent Set** is a graph G and an integer k.

Reduction given $\langle G, k \rangle$ outputs $\langle \overline{G}, k \rangle$ where $\overline{G}$ is the *complement* of G. $\overline{G}$ has an edge (u, v) if and only if (u, v) is not an edge of G.



Correctness of reduction

Lemma

G has an independent set of size k if and only if $\overline{G}$ has a clique of size k.

Proof.

Need to prove two facts:

G has independent set of size at least k implies that $\overline{G}$ has a clique of size at least k.

$\overline{G}$ has a clique of size at least k implies that G has an independent set of size at least k.

Easy to see both from the fact that $S \subseteq V$ is an independent set in G if and only if S is a clique in $\overline{G}$. □

Independent Set and Clique

Independent Set $\leq_P$ **Clique**. What does this mean?

Independent Set and Clique

Independent Set $\leq_P$ **Clique**. What does this mean?

1. If have an algorithm for **Clique**, then we have an algorithm for **Independent Set**.
2. The reduction is efficient. Hence, if we have a poly-time algorithm for **Clique**, then we have a poly-time algorithm for **Independent Set**.
3. **Clique** is *at least as hard as* **Independent Set**.

Independent Set and Clique

Independent Set $\leq_P$ **Clique**. What does this mean?

1. If have an algorithm for **Clique**, then we have an algorithm for **Independent Set**.
2. The reduction is efficient. Hence, if we have a poly-time algorithm for **Clique**, then we have a poly-time algorithm for **Independent Set**.
3. **Clique** is *at least as hard as* **Independent Set**.

Also... **Clique** $\leq_P$ **Independent Set**. Why?

Independent Set and Clique

Independent Set $\leq_P$ **Clique**. What does this mean?

1. If have an algorithm for **Clique**, then we have an algorithm for **Independent Set**.
2. The reduction is efficient. Hence, if we have a poly-time algorithm for **Clique**, then we have a poly-time algorithm for **Independent Set**.
3. **Clique** is *at least as hard as* **Independent Set**.

Also... **Clique** $\leq_P$ **Independent Set**. Why?
Caveat: in general $X \leq Y$ does not mean that $Y \leq X$.

Vertex Cover

Given a graph $G = (V, E)$, a set of vertices S is:

Vertex Cover

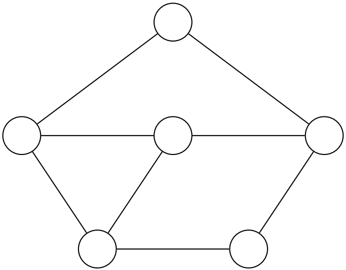
Given a graph $G = (V, E)$, a set of vertices S is:

1. A **vertex cover** if every $e \in E$ has at least one endpoint in S.

Vertex Cover

Given a graph $G = (V, E)$, a set of vertices S is:

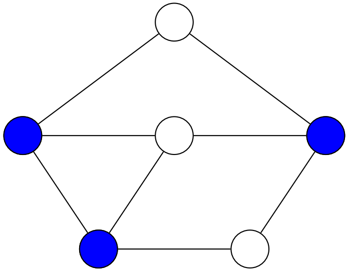
1. A **vertex cover** if every $e \in E$ has at least one endpoint in S.



Vertex Cover

Given a graph $G = (V, E)$, a set of vertices S is:

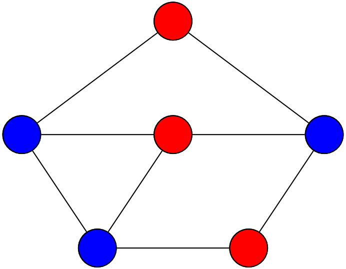
1. A **vertex cover** if every $e \in E$ has at least one endpoint in S.



Vertex Cover

Given a graph $G = (V, E)$, a set of vertices S is:

1. A **vertex cover** if every $e \in E$ has at least one endpoint in S.



The Vertex Cover Problem

Problem (Vertex Cover)

Input: *A graph G and integer k.*
Goal: *Is there a vertex cover of size $\leq k$ in G?*

The Vertex Cover Problem

Problem (Vertex Cover)

Input: *A graph G and integer k.*
Goal: *Is there a vertex cover of size $\leq k$ in G?*

Can we relate **Independent Set** and **Vertex Cover**?

Relationship between...

Vertex Cover and Independent Set

Proposition

Let $G = (V, E)$ be a graph. S is an independent set if and only if $V \setminus S$ is a vertex cover.

Proof.

($\Rightarrow$) Let S be an independent set

1. Consider any edge $uv \in E$.
2. Since S is an independent set, either $u \notin S$ or $v \notin S$.
3. Thus, either $u \in V \setminus S$ or $v \in V \setminus S$.
4. $V \setminus S$ is a vertex cover.

($\Leftarrow$) Let $V \setminus S$ be some vertex cover:

1. Consider $u, v \in S$
2. uv is not an edge of G, as otherwise $V \setminus S$ does not cover uv.
3. $\implies$ S is thus an independent set. □

Independent Set $\leq_P$ Vertex Cover

1. G: graph with n vertices, and an integer k be an instance of the **Independent Set** problem.
2. Reduction: given (G, k), an instance of **Independent Set** , ouput $(G, n - k)$ as an instance of **Vertex Cover**.
3. G has an independent set of size $\geq k$ iff G has a vertex cover of size $\leq n - k$ which proves correctness.
4. Easy to see reduction is efficient.
5. Therefore, **Independent Set** $\leq_P$ **Vertex Cover**. Also **Vertex Cover** $\leq_P$ **Independent Set**.

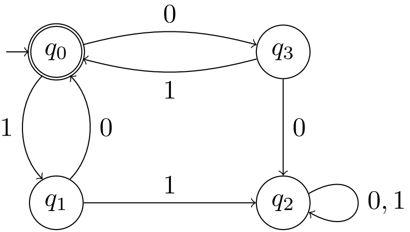
Part IV

Reasoning about Programs

DFA Accepting a String

Given DFA M and string $w \in \Sigma^*$, does M accept w?

- Instance is $\langle M, w \rangle$
- Algorithm: given $\langle M, w \rangle$, output YES if M accepts w, else NO



Does above DFA accept 0010110?

DFA Accepting a String

Given DFA M and string $w \in \Sigma^*$, does M accept w?

- Instance is $\langle M, w \rangle$
- Algorithm: given $\langle M, w \rangle$, output YES if M accepts w, else NO

Question: Is there an (efficient) algorithm for this problem?

DFA Accepting a String

Given DFA M and string $w \in \Sigma^*$, does M accept w?

- Instance is $\langle M, w \rangle$
- Algorithm: given $\langle M, w \rangle$, output YES if M accepts w, else NO

Question: Is there an (efficient) algorithm for this problem?

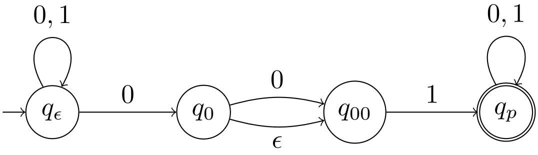
Yes. Simulate M on w and output YES if M reaches a final state.

Exercise: Show a linear time algorithm. Note that linear is in the input size which includes both encoding size of M and $|w|$.

NFA Accepting a String

Given NFA N and string $w \in \Sigma^*$, does N accept w?

- Instance is $\langle N, w \rangle$
- Algorithm: given $\langle N, w \rangle$, output YES if N accepts w, else NO



Does above NFA accept 0010110?

NFA Accepting a String

Given NFA N and string $w \in \Sigma^*$, does N accept w?

- Instance is $\langle N, w \rangle$
- Algorithm: given $\langle N, w \rangle$, output YES if N accepts w, else NO

Question: Is there an algorithm for this problem?

NFA Accepting a String

Given NFA N and string $w \in \Sigma^*$, does N accept w?

- Instance is $\langle N, w \rangle$
- Algorithm: given $\langle N, w \rangle$, output YES if N accepts w, else NO

Question: Is there an algorithm for this problem?

- Convert N to equivalent DFA M and use previous algorithm!
- Hence a reduction that takes $\langle N, w \rangle$ to $\langle M, w \rangle$
- Is this reduction efficient?

NFA Accepting a String

Given NFA N and string $w \in \Sigma^*$, does N accept w?

- Instance is $\langle N, w \rangle$
- Algorithm: given $\langle N, w \rangle$, output YES if N accepts w, else NO

Question: Is there an algorithm for this problem?

- Convert N to equivalent DFA M and use previous algorithm!
- Hence a reduction that takes $\langle N, w \rangle$ to $\langle M, w \rangle$
- Is this reduction efficient? No, because $|M|$ is exponential in $|N|$ in the worst case.

Exercise: Describe a polynomial-time algorithm.
Hence reduction may allow you to see an easy algorithm but not necessarily best algorithm!

DFA Universality

A DFA M is universal if it accepts every string.
That is, $L(M) = \Sigma^*$, the set of all strings.

DFA Universality

A DFA M is universal if it accepts every string.
That is, $L(M) = \Sigma^*$, the set of all strings.

Problem (DFA universality)

Input: *A* DFA $\langle M \rangle$.
Goal: *Is* M *universal?*

DFA Universality

A DFA M is universal if it accepts every string.
That is, $L(M) = \Sigma^*$, the set of all strings.

Problem (DFA universality)

Input: *A DFA $\langle M \rangle$.*
Goal: *Is M universal?*

How do we solve **DFA Universality**?

DFA Universality

A DFA M is universal if it accepts every string.
That is, $L(M) = \Sigma^*$, the set of all strings.

Problem (DFA universality)

Input: *A DFA* $\langle M \rangle$*.*
Goal: *Is* M *universal?*

How do we solve **DFA Universality**?
We check if M has *any* reachable non-final state.

NFA Universality

An NFA N is said to be universal if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.

Problem (NFA universality)

Input: *A NFA M.*
Goal: *Is M universal?*

How do we solve **NFA Universality**?

NFA Universality

An NFA N is said to be universal if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.

Problem (NFA universality)

Input: *A NFA M.*
Goal: *Is M universal?*

How do we solve **NFA Universality**?
Reduce it to **DFA Universality**?

NFA Universality

An NFA N is said to be universal if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.

Problem (NFA universality)

Input: *A* NFA M.
Goal: *Is* M *universal?*

How do we solve **NFA Universality**?
Reduce it to **DFA Universality**?
Given an NFA N, convert it to an equivalent DFA M, and use the **DFA Universality** Algorithm.

NFA Universality

An NFA N is said to be universal if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.

Problem (NFA universality)

Input: *A NFA M.*
Goal: *Is M universal?*

How do we solve **NFA Universality**?
Reduce it to **DFA Universality**?
Given an NFA N, convert it to an equivalent DFA M, and use the **DFA Universality** Algorithm.
The reduction takes exponential time!
NFA Universality is known to be PSPACE-Complete and we do not expect a polynomial-time algorithm.

Reasoning about TMs/Programs

- $\langle M \rangle$ is encoding of a TM M.
- Equivalently think of $\langle M \rangle$ as the code of a program in some high-level programming language

Reasoning about TMs/Programs

- $\langle M \rangle$ is encoding of a TM M.
- Equivalently think of $\langle M \rangle$ as the code of a program in some high-level programming language

Three related problems:

- Given $\langle M \rangle$ does M halt on blank input? (Halting Problem)
- Given $\langle M, w \rangle$ does M halt on input w?
- Given $\langle M, w \rangle$ does M accept w? (Universal Language)

Question: Do any of the above problems have an algorithm?

Reasoning about TMs/Programs

- $\langle M \rangle$ is encoding of a TM M.
- Equivalently think of $\langle M \rangle$ as the code of a program in some high-level programming language

Three related problems:

- Given $\langle M \rangle$ does M halt on blank input? (Halting Problem)
- Given $\langle M, w \rangle$ does M halt on input w?
- Given $\langle M, w \rangle$ does M accept w? (Universal Language)

Question: Do any of the above problems have an algorithm?

Theorem (Turing)

All the three problems are undecidable! No algorithm/program/TM.

CS 125 assignment

Write a program that prints “Hello World”

```
main() {
    print(‘‘Hello World’’)
}
```

CS 125 assignment

Write a program that prints "Hello World"

```
main() {
    print(''Hello World'')
}
```

Question: Can we create an autograder?

CS 125 assignment

Write a program that prints "Hello World"

```
main() {
    print(''Hello World'')
}
```

Question: Can we create an autograder? No! Why?

```
main() {
    stealthcode()
    print(''Hello World'')
}
stealthcode() {
    do this
    do that
    viola
}
```

Reducing Halting to Autograder

- **Halting problem:** given arbitrary program foo(), does it halt?

Reducing Halting to Autograder

- **Halting problem:** given arbitrary program foo(), does it halt?
- **Reduction to CS125Autograder**: given foo() output foobar()

```
main() {
    foo()
    print(``Hello World'')
}
foo() {
    line 1
    line 2
    ...
}
```

Note: Reduction only needs to add a few lines of code to foo()

Reducing Halting to Autograder

- **Halting problem:** given arbitrary program foo(), does it halt?
- **Reduction to CS125Autograder**: given foo() output foobar()

```
main() {
    foo()
    print(``Hello World'')
}
foo() {
    line 1
    line 2
    ...
}
```

Note: Reduction only needs to add a few lines of code to foo()

- foobar() prints "Hello World" **if and only if** foo() halts!
- If we had CS125Autograder then we can solve Halting. But Halting is hard according to Turing. Hence ...