Turing Machines and Decidability

Purushothama B R

Department of Computer Science and Engineering
National Institute of Technology Goa ,INDIA

December 7, 2020

I want to Begin With... J K Rowling's Quote

It is impossible to live without failing at something, unless you live so cautiously that you might has well not have lived at all, in which case you have failed by default.

Investigate the Power of Algorithms

Investigate the Power of Algorithms

- I begin to investigate the power of algorithms to solve problems.

Investigate the Power of Algorithms

- I begin to investigate the power of algorithms to solve problems.
- I demonstrate certain problems that can be solved algorithmically and others that cannot.

Investigate the Power of Algorithms

- I begin to investigate the power of algorithms to solve problems.
- I demonstrate certain problems that can be solved algorithmically and others that cannot.
- My objective is to explore the limits of algorithmic solvability.

Investigate the Power of Algorithms

- I begin to investigate the power of algorithms to solve problems.
- I demonstrate certain problems that can be solved algorithmically and others that cannot.
- My objective is to explore the limits of algorithmic solvability.
- You are probably familiar with solvability by algorithms because much of computer science is devoted to solving problems.

Investigate the Power of Algorithms

- I begin to investigate the power of algorithms to solve problems.
- I demonstrate certain problems that can be solved algorithmically and others that cannot.
- My objective is to explore the limits of algorithmic solvability.
- You are probably familiar with solvability by algorithms because much of computer science is devoted to solving problems.
- The unsolvability of certain problems may come as a surprise.

Study of Unsolvability

Study of Unsolvability

- Why should you study unsolvability?

Study of Unsolvability

- Why should you study unsolvability?
- After all, showing that a problem is unsolvable doesnt appear to be of any use if you have to solve it.

Study of Unsolvability

- Why should you study unsolvability?
- After all, showing that a problem is unsolvable doesnt appear to be of any use if you have to solve it.
- You need to study this phenomenon for two reasons.

Study of Unsolvability

- Why should you study unsolvability?
- After all, showing that a problem is unsolvable doesnt appear to be of any use if you have to solve it.
- You need to study this phenomenon for two reasons.
    - First, knowing when a problem is algorithmically unsolvable is useful.

Study of Unsolvability

- Why should you study unsolvability?
- After all, showing that a problem is unsolvable doesnt appear to be of any use if you have to solve it.
- You need to study this phenomenon for two reasons.
  - First, knowing when a problem is algorithmically unsolvable is useful.
  - Then you realize that the **problem must be simplified or altered** before you can find an algorithmic solution.

Study of Unsolvability

- Why should you study unsolvability?
- After all, showing that a problem is unsolvable doesnt appear to be of any use if you have to solve it.
- You need to study this phenomenon for two reasons.
  - First, knowing when a problem is algorithmically unsolvable is useful.
  - Then you realize that the **problem must be simplified or altered** before you can find an algorithmic solution.
  - Like any tool, **computers have capabilities and limitations** that must be appreciated if they are to be used well.

Study of Unsolvability

- Why should you study unsolvability?
- After all, showing that a problem is unsolvable doesnt appear to be of any use if you have to solve it.
- You need to study this phenomenon for two reasons.
  - First, knowing when a problem is algorithmically unsolvable is useful.
  - Then you realize that the **problem must be simplified or altered** before you can find an algorithmic solution.
  - Like any tool, **computers have capabilities and limitations** that must be appreciated if they are to be used well.
- The second reason is cultural.

Study of Unsolvability

- Why should you study unsolvability?
- After all, showing that a problem is unsolvable doesnt appear to be of any use if you have to solve it.
- You need to study this phenomenon for two reasons.
  - First, knowing when a problem is algorithmically unsolvable is useful.
  - Then you realize that the **problem must be simplified or altered** before you can find an algorithmic solution.
  - Like any tool, **computers have capabilities and limitations** that must be appreciated if they are to be used well.
- The second reason is cultural.
  - Even if you deal with problems that clearly are solvable,

Study of Unsolvability

- Why should you study unsolvability?
- After all, showing that a problem is unsolvable doesnt appear to be of any use if you have to solve it.
- You need to study this phenomenon for two reasons.
  - First, knowing when a problem is algorithmically unsolvable is useful.
  - Then you realize that the **problem must be simplified or altered** before you can find an algorithmic solution.
  - Like any tool, **computers have capabilities and limitations** that must be appreciated if they are to be used well.
- The second reason is cultural.
  - Even if you deal with problems that clearly are solvable,
    - A glimpse of the unsolvable can stimulate your imagination and

Study of Unsolvability

- Why should you study unsolvability?
- After all, showing that a problem is unsolvable doesnt appear to be of any use if you have to solve it.
- You need to study this phenomenon for two reasons.
  - First, knowing when a problem is algorithmically unsolvable is useful.
  - Then you realize that the **problem must be simplified or altered** before you can find an algorithmic solution.
  - Like any tool, **computers have capabilities and limitations** that must be appreciated if they are to be used well.
- The second reason is cultural.
  - Even if you deal with problems that clearly are solvable,
    - A glimpse of the unsolvable can stimulate your imagination and
    - Help you gain an important perspective on computation.

DECIDABLE LANGUAGES

- I give some examples of languages that are decidable by algorithms.

DECIDABLE LANGUAGES

- I give some examples of languages that are decidable by algorithms.
- I focus on languages concerning automata and grammars.

DECIDABLE LANGUAGES

- I give some examples of languages that are decidable by algorithms.
- I focus on languages concerning automata and grammars.
- I present an algorithm that tests whether a string is a member of a context-free language (CFL).

DECIDABLE LANGUAGES

- I give some examples of languages that are decidable by algorithms.
- I focus on languages concerning automata and grammars.
- I present an algorithm that tests whether a string is a member of a context-free language (CFL).
- These languages are interesting for several reasons.

- I give some examples of languages that are decidable by algorithms.
- I focus on languages concerning automata and grammars.
- I present an algorithm that tests whether a string is a member of a context-free language (CFL).
- These languages are interesting for several reasons.
  - First, certain problems of this kind are related to applications.

DECIDABLE LANGUAGES

- I give some examples of languages that are decidable by algorithms.
- I focus on languages concerning automata and grammars.
- I present an algorithm that tests whether a string is a member of a context-free language (CFL).
- These languages are interesting for several reasons.
  - First, certain problems of this kind are related to applications.
  - This problem of testing whether a CFG generates a string is related

DECIDABLE LANGUAGES

- I give some examples of languages that are decidable by algorithms.
- I focus on languages concerning automata and grammars.
- I present an algorithm that tests whether a string is a member of a context-free language (CFL).
- These languages are interesting for several reasons.
  - First, certain problems of this kind are related to applications.
  - This problem of testing whether a CFG generates a string is relatedto the problem of recognizing and compiling programs in a programming language.

DECIDABLE LANGUAGES

- I give some examples of languages that are decidable by algorithms.
- I focus on languages concerning automata and grammars.
- I present an algorithm that tests whether a string is a member of a context-free language (CFL).
- These languages are interesting for several reasons.
  - First, certain problems of this kind are related to applications.
  - This problem of testing whether a CFG generates a string is relatedto the problem of recognizing and compiling programs in a programming language.
  - Second, certain other problems concerning automata and grammars are not decidable by algorithms.

DECIDABLE LANGUAGES

- I give some examples of languages that are decidable by algorithms.
- I focus on languages concerning automata and grammars.
- I present an algorithm that tests whether a string is a member of a context-free language (CFL).
- These languages are interesting for several reasons.
  - First, certain problems of this kind are related to applications.
  - This problem of testing whether a CFG generates a string is relatedto the problem of recognizing and compiling programs in a programming language.
  - Second, certain other problems concerning automata and grammars are not decidable by algorithms.
  - Starting with examples where decidability is possible helps you to appreciate the undecidable examples.

DECIDABLE PROBLEMS CONCERNING REGULAR LANGUAGES

ALgorithms for Regular Languages

ALgorithms for Regular Languages

- I begin with certain computational problems concerning finite automata.

ALgorithms for Regular Languages

- I begin with certain computational problems concerning finite automata.
- I give algorithms for testing

ALgorithms for Regular Languages

- I begin with certain computational problems concerning finite automata.
- I give algorithms for testing
  - Whether a finite automaton accepts a string.

ALgorithms for Regular Languages

- I begin with certain computational problems concerning finite automata.
- I give algorithms for testing
  - Whether a finite automaton accepts a string.
  - Whether the language of a finite automaton is empty.

ALgorithms for Regular Languages

- I begin with certain computational problems concerning finite automata.
- I give algorithms for testing
  - Whether a finite automaton accepts a string.
  - Whether the language of a finite automaton is empty.
  - Whether two finite automata are equivalent.

Preliminaries

Preliminaries

- I chose to represent various computational problems by languages.

Preliminaries

- I chose to represent various computational problems by languages.
- Doing so is convenient because we have already set up terminology for dealing with languages.

Preliminaries

- I chose to represent various computational problems by languages.
- Doing so is convenient because we have already set up terminology for dealing with languages.
- For example, **the acceptance problem** for DFAs

Preliminaries

- I chose to represent various computational problems by languages.
- Doing so is convenient because we have already set up terminology for dealing with languages.
- For example, **the acceptance problem** for DFAs of testing whether a particular deterministic finite automaton accepts a given string

Preliminaries

- I chose to represent various computational problems by languages.
- Doing so is convenient because we have already set up terminology for dealing with languages.
- For example, **the acceptance problem** for DFAs of testing whether a particular deterministic finite automaton accepts a given string can be expressed as a language, A_{DFA} .

Preliminaries

- I chose to represent various computational problems by languages.
- Doing so is convenient because we have already set up terminology for dealing with languages.
- For example, **the acceptance problem** for DFAs of testing whether a particular deterministic finite automaton accepts a given string can be expressed as a language, A_{DFA} .
- This language contains the encodings of all DFAs together with strings that the DFAs accept. Let

  $A_{DFA} = \{< B, w > | B \text{ is a } DFA \text{ that accepts input string } w\}$.

- I chose to represent various computational problems by languages.
- Doing so is convenient because we have already set up terminology for dealing with languages.
- For example, **the acceptance problem** for DFAs of testing whether a particular deterministic finite automaton accepts a given string can be expressed as a language, A_{DFA} .
- This language contains the encodings of all DFAs together with strings that the DFAs accept. Let

  $A_{DFA} = \{< B, w > | B \text{ is a } DFA \text{ that accepts input string } w\}$.
- The problem of testing whether a DFA B accepts an input w is the same as

Preliminaries

- I chose to represent various computational problems by languages.
- Doing so is convenient because we have already set up terminology for dealing with languages.
- For example, **the acceptance problem** for DFAs of testing whether a particular deterministic finite automaton accepts a given string can be expressed as a language, A_{DFA} .
- This language contains the encodings of all DFAs together with strings that the DFAs accept. Let

  $A_{DFA} = \{< B, w > | B \text{ is a } DFA \text{ that accepts input string } w\}$.
- The problem of testing whether a DFA B accepts an input w is the same as
  - The problem of testing whether $< B, w >$, is a member of the language A_{DFA}.

Contd..

Contd..

- Similarly, we can formulate other computational problems in terms of testing membership in a language.

- Similarly, we can formulate other computational problems in terms of testing membership in a language.
- Showing that the **language is decidable** is the same as

Contd..

- Similarly, we can formulate other computational problems in terms of testing membership in a language.
- Showing that the **language is decidable** is the same as showing that the **computational problem is decidable.**

Theorem

Theorem

Theorem 4.1

A_{DFA} is decidable language.

Theorem

Theorem 4.1

A_{DFA} is decidable language.

Proof Idea

We simply need to present a TM M that decides A_{DFA}.

Algorithm

$M =$ "On input $\langle B, w\rangle$, where B is a DFA and w is a string:

1. Simulate B on input w.
2. If the simulation ends in an accept state, *accept*. If it ends in a nonaccepting state, *reject*."

Proof

Proof

- I mention just a few implementation details of this proof.

Proof

- I mention just a few implementation details of this proof.
- For those of you familiar with writing programs in any standard programming lan- guage,

Proof

- I mention just a few implementation details of this proof.
- For those of you familiar with writing programs in any standard programming lan- guage,
- Imagine how you would write a program to carry out the simulation.

Proof

- I mention just a few implementation details of this proof.
- For those of you familiar with writing programs in any standard programming lan- guage,
- Imagine how you would write a program to carry out the simulation.
- First, lets examine the input $< B, w >$.

Proof

- I mention just a few implementation details of this proof.
- For those of you familiar with writing programs in any standard programming lan- guage,
- Imagine how you would write a program to carry out the simulation.
- First, lets examine the input $< B, w >$.
- It is a representation of a DFA B together with a string w.

Proof

- I mention just a few implementation details of this proof.
- For those of you familiar with writing programs in any standard programming lan- guage,
- Imagine how you would write a program to carry out the simulation.
- First, lets examine the input $< B, w >$.
- It is a representation of a DFA B together with a string w.
- One reasonable representation of B is simply

Proof

- I mention just a few implementation details of this proof.
- For those of you familiar with writing programs in any standard programming lan- guage,
- Imagine how you would write a program to carry out the simulation.
- First, lets examine the input $< B, w >$.
- It is a representation of a DFA B together with a string w.
- One reasonable representation of B is simply
  - A list of its five components: $Q, , , q_0$, and F .

Proof

- I mention just a few implementation details of this proof.
- For those of you familiar with writing programs in any standard programming lan- guage,
- Imagine how you would write a program to carry out the simulation.
- First, lets examine the input $< B, w >$.
- It is a representation of a DFA B together with a string w.
- One reasonable representation of B is simply
    - A list of its five components: $Q, , , q_0$, and F .
- When M receives its input,

Proof

- I mention just a few implementation details of this proof.
- For those of you familiar with writing programs in any standard programming lan- guage,
- Imagine how you would write a program to carry out the simulation.
- First, lets examine the input $< B, w >$.
- It is a representation of a DFA B together with a string w.
- One reasonable representation of B is simply
  - A list of its five components: $Q, , , q_0$, and F .
- When M receives its input,
  - M first determines whether it properly represents a DFA B and a string w.

Proof

- I mention just a few implementation details of this proof.
- For those of you familiar with writing programs in any standard programming lan- guage,
- Imagine how you would write a program to carry out the simulation.
- First, lets examine the input $< B, w >$.
- It is a representation of a DFA B together with a string w.
- One reasonable representation of B is simply
  - A list of its five components: $Q, , , q_0$, and F .
- When M receives its input,
  - M first determines whether it properly represents a DFA B and a string w.
  - If not, M rejects.

Contd..

Contd..

- Then M carries out the simulation directly.

Contd..

- Then M carries out the simulation directly.
- It keeps track of B's current state and

Contd..

- Then M carries out the simulation directly.
- It keeps track of B's current state and Bs current position in the input w by writing this information down on its tape.

Contd..

- Then M carries out the simulation directly.
- It keeps track of B's current state and Bs current position in the input w by writing this information down on its tape.
- Initially, B's current state is q_0 and

- Then M carries out the simulation directly.
- It keeps track of B's current state and Bs current position in the input w by writing this information down on its tape.
- Initially, B's current state is q_0 and
- B's current input position is the leftmost symbol of w.

Contd..

- Then M carries out the simulation directly.
- It keeps track of B's current state and Bs current position in the input w by writing this information down on its tape.
- Initially, B's current state is q_0 and
- B's current input position is the leftmost symbol of w.
- he states and position are updated according to the specified transition function δ.

Contd..

- Then M carries out the simulation directly.
- It keeps track of B's current state and Bs current position in the input w by writing this information down on its tape.
- Initially, B's current state is q_0 and
- B's current input position is the leftmost symbol of w.
- he states and position are updated according to the specified transition function δ.
- When M finishes processing the last symbol of w,

- Then M carries out the simulation directly.
- It keeps track of B's current state and Bs current position in the input w by writing this information down on its tape.
- Initially, B's current state is q_0 and
- B's current input position is the leftmost symbol of w.
- he states and position are updated according to the specified transition function δ.
- When M finishes processing the last symbol of w,
- M accepts the input if B is in an accepting state;

Contd..

- Then M carries out the simulation directly.
- It keeps track of B's current state and Bs current position in the input w by writing this information down on its tape.
- Initially, B's current state is q_0 and
- B's current input position is the leftmost symbol of w.
- he states and position are updated according to the specified transition function δ.
- When M finishes processing the last symbol of w,
- M accepts the input if B is in an accepting state;
- M rejects the input if B is in a nonaccepting state.

Theorem

Theorem

- Let

$$A_{NFA} = \{< B, w > | B \text{ is an } NFA \text{ that accepts input string } w\}.$$

Theorem

- Let
$$A_{NFA} = \{< B, w > | B \text{ is an } NFA \text{ that accepts input string } w\}.$$

Theorem 4.2

A_{NFA} is decidable language.

Proof

Proof

- We present a TM N that decides A_{NFA}.

Proof

- We present a TM N that decides A_{NFA}.
- We can design N to operate like M , simulating an NFA instead of a DFA.

Proof

- We present a TM N that decides A_{NFA}.
- We can design N to operate like M , simulating an NFA instead of a DFA.
- Instead, well do it differently to illustrate a new idea:

Proof

- We present a TM N that decides A_{NFA}.
- We can design N to operate like M , simulating an NFA instead of a DFA.
- Instead, well do it differently to illustrate a new idea:
  - Have N use M as a subroutine.

Proof

- We present a TM N that decides A_{NFA}.
- We can design N to operate like M , simulating an NFA instead of a DFA.
- Instead, well do it differently to illustrate a new idea:
  - Have N use M as a subroutine.
  - N first converts the NFA it receives as input to a DFA before passing it to M .

Algorithm

$N =$ "On input $\langle B, w\rangle$, where B is an NFA and w is a string:

1. Convert NFA B to an equivalent DFA C, using the procedure for this conversion given in Theorem 1.39.
2. Run TM M from Theorem 4.1 on input $\langle C, w\rangle$.
3. If M accepts, *accept*; otherwise, *reject*."

Regular Expression Decidability

Regular Expression Decidability

- We can determine whether a regular expression generates a given string.

Regular Expression Decidability

- We can determine whether a regular expression generates a given string.
- Let

$$A_{REX} = \{< R, w > | R \text{ is a regular expression that generates string } w\}.$$

Regular Expression Decidability

Regular Expression Decidability

- Let

$$A_{REX} = \{< R, w > | R \text{ is a regular expression that generates string } w\}.$$

- Let

$$A_{REX} = \{< R, w > | R \text{ is a regular expression that generates string } w\}.$$

Theorem 4.2

A_{REX} is decidable language.

Proof

Proof

- The following TM P decides A_{REX} .

Proof

- The following TM P decides A_{REX} .

$P =$ "On input $\langle R, w \rangle$, where R is a regular expression and w is a string:

1. Convert regular expression R to an equivalent NFA A by using the procedure for this conversion given in Theorem 1.54.
2. Run TM N on input $\langle A, w \rangle$.
3. If N accepts, *accept*; if N rejects, *reject*."

Different Problem

Different Problem

- I will turn to a different kind of problem concerning finite automata.

Different Problem

- I will turn to a different kind of problem concerning finite automata.
- Emptiness testing for the language of a finite automaton.

Different Problem

- I will turn to a different kind of problem concerning finite automata.
- Emptiness testing for the language of a finite automaton.
- In the preceding three theorems

Different Problem

- I will turn to a different kind of problem concerning finite automata.
- Emptiness testing for the language of a finite automaton.
- In the preceding three theorems we had to determine

Different Problem

- I will turn to a different kind of problem concerning finite automata.
- Emptiness testing for the language of a finite automaton.
- In the preceding three theorems we had to determine whether a finite automaton accepts a particular string.

Different Problem

- I will turn to a different kind of problem concerning finite automata.
- Emptiness testing for the language of a finite automaton.
- In the preceding three theorems we had to determine whether a finite automaton accepts a particular string.
- Now we must determine whether or not a finite automaton accepts any strings at all.

Emptiness Testing

Emptiness Testing

- Let

$$E_{DFA} = \{< A > | A \text{ is a DFA and } L(A) = \emptyset\}.$$

Emptiness Testing

- Let

$$E_{DFA} = \{< A > | A \text{ is a DFA and } L(A) = \emptyset\}.$$

Theorem 4.4

E_{DFA} is a decidable language.

Proof

Proof

- A DFA accepts some string iff

Proof

- A DFA accepts some string iff
  - Reaching an accept state from the start state,

Proof

- A DFA accepts some string iff
  - Reaching an accept state from the start state,
  - By traveling along the arrows of the DFA is possible.

Proof

- A DFA accepts some string iff
  - Reaching an accept state from the start state,
  - By traveling along the arrows of the DFA is possible.
- We can design a TM T that uses a marking algorithm that I had discussed earlier.

Proof

$T =$ "On input $\langle A \rangle$, where A is a DFA:

1. Mark the start state of A.

2. Repeat until no new states get marked:

3. Mark any state that has a transition coming into it from any state that is already marked.

4. If no accept state is marked, *accept*; otherwise, *reject*."

Another Decidable Problem

Another Decidable Problem

- We focus on determining whether two DFAs recognize the same language is decidable.

Another Decidable Problem

- We focus on determining whether two DFAs recognize the same language is decidable.
- Let

$$EQ_{DFA} = \{< A, B > | A \text{ and } B \text{ are DFA's and } L(A) = L(B)\}$$

Another Decidable Problem

- We focus on determining whether two DFAs recognize the same language is decidable.
- Let

$$EQ_{DFA} = \{< A, B > | A \text{ and } B \text{ are DFA's and } L(A) = L(B)\}$$

Theorem 4.4

EQ_{DFA} is a decidable language.

Proof

Proof

- I will make use of Theorem 4.4.

Proof

- I will make use of Theorem 4.4.
- We construct a new DFA C from A and B, where

Proof

- I will make use of Theorem 4.4.
- We construct a new DFA C from A and B, where
  - C accepts only those strings that are accepted by

Proof

- I will make use of Theorem 4.4.
- We construct a new DFA C from A and B, where
  - C accepts only those strings that are accepted by **either A or B but not by both.**

Proof

- I will make use of Theorem 4.4.
- We construct a new DFA C from A and B, where
  - C accepts only those strings that are accepted by **either A or B but not by both.**
- The language C is

$$L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$$

Proof

- I will make use of Theorem 4.4.
- We construct a new DFA C from A and B, where
  - C accepts only those strings that are accepted by **either A or B but not by both.**
- The language C is

$$L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$$

- The symmetric difference is useful here

Proof

- I will make use of Theorem 4.4.
- We construct a new DFA C from A and B, where
  - C accepts only those strings that are accepted by **either A or B but not by both.**
- The language C is

$$L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$$

- The symmetric difference is useful here because $L(C) = \emptyset$ iff $L(A) = L(B)$.

Proof

- I will make use of Theorem 4.4.
- We construct a new DFA C from A and B, where
  - C accepts only those strings that are accepted by **either A or B but not by both.**
- The language C is
$$L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$$
- The symmetric difference is useful here because $L(C) = \emptyset$ iff $L(A) = L(B)$.
- We can construct C from A and B with the constructions

Proof

- I will make use of Theorem 4.4.
- We construct a new DFA C from A and B, where
  - C accepts only those strings that are accepted by **either A or B but not by both.**
- The language C is

$$L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$$

- The symmetric difference is useful here because $L(C) = \emptyset$ iff $L(A) = L(B)$.
- We can construct C from A and B with the constructions
  - For proving the class of regular languages **closed under complementation, union, and intersection.**

Proof

- I will make use of Theorem 4.4.
- We construct a new DFA C from A and B, where
  - C accepts only those strings that are accepted by **either A or B but not by both.**
- The language C is
$$L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$$
- The symmetric difference is useful here because $L(C) = \emptyset$ iff $L(A) = L(B)$.
- We can construct C from A and B with the constructions
  - For proving the class of regular languages **closed under complementation, union, and intersection.**
- These constructions are algorithms that can be carried out by Turing machines.

Contd..

Contd..

- Once we have constructed C,

Contd..

- Once we have constructed C,we can use Theorem 4.4 to test whether $L(C)$ is empty.

Contd..

- Once we have constructed C,we can use Theorem 4.4 to test whether $L(C)$ is empty.
- If it is empty, $L(A)$ and $L(B)$ must be equal.

- Once we have constructed C,we can use Theorem 4.4 to test whether $L(C)$ is empty.
- If it is empty, $L(A)$ and $L(B)$ must be equal.

F = "On input $\langle A, B\rangle$, where A and B are DFAs:

1. Construct DFA C as described.
2. Run TM T from Theorem 4.4 on input $\langle C\rangle$.
3. If T accepts, *accept*. If T rejects, *reject*."

DECIDABLE PROBLEMS CONCERNING CONTEXT-FREE LANGUAGES

Focus

Focus

- I describe algorithms

Focus

- I describe algorithms
  - To determine whether a CFG generates a particular string w

Focus

- I describe algorithms
  - To determine whether a CFG generates a particular string w
  - To determine whether the language of a CFG is empty.

Focus

- I describe algorithms
  - To determine whether a CFG generates a particular string w
  - To determine whether the language of a CFG is empty.
- Let

$$A_{CFG} = \{< G, w > | G \text{ is a CFG that generates string } w\}.$$

Focus

- I describe algorithms
  - To determine whether a CFG generates a particular string w
  - To determine whether the language of a CFG is empty.
- Let

$$A_{CFG} = \{< G, w > | G \text{ is a CFG that generates string } w\}.$$

Theorem 4.7

A_{CFG} is a decidable language.

Proof Idea

Proof Idea

- For CFG G and string w, we want to determine whether G generates w.

Proof Idea

- For CFG G and string w, we want to determine whether G generates w.
- One idea is to

Proof Idea

- For CFG G and string w, we want to determine whether G generates w.
- One idea is to
  - Use G to go through all derivations to determine

Proof Idea

- For CFG G and string w, we want to determine whether G generates w.
- One idea is to
  - Use G to go through all derivations to determine whether any is a derivation of w.

Proof Idea

- For CFG G and string w, we want to determine whether G generates w.
- One idea is to
  - Use G to go through all derivations to determine whether any is a derivation of w.
- This idea doesnt work,

Proof Idea

- For CFG G and string w, we want to determine whether G generates w.
- One idea is to
  - Use G to go through all derivations to determine whether any is a derivation of w.
- This idea doesnt work,
  - As infinitely many derivations may have to be tried.

Proof Idea

- For CFG G and string w, we want to determine whether G generates w.
- One idea is to
  - Use G to go through all derivations to determine whether any is a derivation of w.
- This idea doesnt work,
  - As infinitely many derivations may have to be tried.
  - If G does not generate w,

Proof Idea

- For CFG G and string w, we want to determine whether G generates w.
- One idea is to
  - Use G to go through all derivations to determine whether any is a derivation of w.
- This idea doesnt work,
  - As infinitely many derivations may have to be tried.
  - If G does not generate w, this algorithm would never halt.

Proof Idea

- For CFG G and string w, we want to determine whether G generates w.
- One idea is to
  - Use G to go through all derivations to determine whether any is a derivation of w.
- This idea doesnt work,
  - As infinitely many derivations may have to be tried.
  - If G does not generate w, this algorithm would never halt.
- This idea gives a **Turing machine that is a recognizer,**

Proof Idea

- For CFG G and string w, we want to determine whether G generates w.
- One idea is to
  - Use G to go through all derivations to determine whether any is a derivation of w.
- This idea doesnt work,
  - As infinitely many derivations may have to be tried.
  - If G does not generate w, this algorithm would never halt.
- This idea gives a **Turing machine that is a recognizer,** but not a decider.

Proof Idea

Proof Idea

- To make this Turing machine into a decider,

Proof Idea

- To make this Turing machine into a decider,
  - We need to ensure that the algorithm tries

Proof Idea

- To make this Turing machine into a decider,
  - We need to ensure that the algorithm tries **only finitely many derivations.**

Proof Idea

- To make this Turing machine into a decider,
  - We need to ensure that the algorithm tries **only finitely many derivations.**

Note

If G were in Chomsky normal form,

Proof Idea

- To make this Turing machine into a decider,
  - We need to ensure that the algorithm tries **only finitely many derivations.**

Note

If G were in Chomsky normal form, any derivation of w has $2n - 1$ steps,

Proof Idea

- To make this Turing machine into a decider,
  - We need to ensure that the algorithm tries **only finitely many derivations.**

Note

If G were in Chomsky normal form, any derivation of w has $2n - 1$ steps,where n is the length of w.

Proof Idea

- To make this Turing machine into a decider,
  - We need to ensure that the algorithm tries **only finitely many derivations.**

Note

If G were in Chomsky normal form, any derivation of w has $2n - 1$ steps,where n is the length of w.

  - In that case, checking only derivations with $2n - 1$ steps to determine

Proof Idea

- To make this Turing machine into a decider,
  - We need to ensure that the algorithm tries **only finitely many derivations.**

Note

If G were in Chomsky normal form, any derivation of w has $2n - 1$ steps,where n is the length of w.

  - In that case, checking only derivations with $2n - 1$ steps to determine whether G generates w **would be sufficient.**

Proof Idea

- To make this Turing machine into a decider,
  - We need to ensure that the algorithm tries **only finitely many derivations.**

Note

If G were in Chomsky normal form, any derivation of w has $2n - 1$ steps,where n is the length of w.

- In that case, checking only derivations with $2n - 1$ steps to determine whether G generates w **would be sufficient.**
- Only finitely many such derivations exist.

Proof Idea

- To make this Turing machine into a decider,
  - We need to ensure that the algorithm tries **only finitely many derivations.**

Note

If G were in Chomsky normal form, any derivation of w has $2n-1$ steps,where n is the length of w.

- In that case, checking only derivations with $2n-1$ steps to determine whether G generates w **would be sufficient.**
- Only finitely many such derivations exist.
- We know how to can convert G to Chomsky normal form.

Proof

Proof

- The TM S for A_{CFG} follows.

Proof

- The TM S for A_{CFG} follows.

S = "On input $\langle G, w\rangle$, where G is a CFG and w is a string:

1. Convert G to an equivalent grammar in Chomsky normal form.
2. List all derivations with $2n-1$ steps, where n is the length of w; except if $n=0$, then instead list all derivations with one step.
3. If any of these derivations generate w, *accept*; if not, *reject*."

Discussion

Discussion

- The problem of determining whether a CFG generates a particular string

Discussion

- The problem of determining whether a CFG generates a particular string is related to the problem of compiling programming languages.

Discussion

- The problem of determining whether a CFG generates a particular string is related to the problem of compiling programming languages.
- The algorithm in TM S is very inefficient

Discussion

- The problem of determining whether a CFG generates a particular string is related to the problem of compiling programming languages.
- The algorithm in TM S is very inefficient and would never be used in practice,

Discussion

- The problem of determining whether a CFG generates a particular string is related to the problem of compiling programming languages.
- The algorithm in TM S is very inefficient and would never be used in practice,
- But it is easy to describe

Discussion

- The problem of determining whether a CFG generates a particular string is related to the problem of compiling programming languages.
- The algorithm in TM S is very inefficient and would never be used in practice,
- But it is easy to describe and we arent concerned with efficiency here.

Discussion

- The problem of determining whether a CFG generates a particular string is related to the problem of compiling programming languages.
- The algorithm in TM S is very inefficient and would never be used in practice,
- But it is easy to describe and we arent concerned with efficiency here.
- More efficient algorithm for recognizing general context-free languages exist.

Discussion

- The problem of determining whether a CFG generates a particular string is related to the problem of compiling programming languages.
- The algorithm in TM S is very inefficient and would never be used in practice,
- But it is easy to describe and we arent concerned with efficiency here.
- More efficient algorithm for recognizing general context-free languages exist.
- Even greater efficiency is possible for

Discussion

- The problem of determining whether a CFG generates a particular string is related to the problem of compiling programming languages.
- The algorithm in TM S is very inefficient and would never be used in practice,
- But it is easy to describe and we arent concerned with efficiency here.
- More efficient algorithm for recognizing general context-free languages exist.
- Even greater efficiency is possible for recognizing deterministic context-free languages.

Important

Important

- I am hopeful that you know how to convert back and forth between CFGs and PDAs.

Important

- I am hopeful that you know how to convert back and forth between CFGs and PDAs.
- Hence everything we say about the decidability

Important

- I am hopeful that you know how to convert back and forth between CFGs and PDAs.
- Hence everything we say about the decidability of problems concerning CFGs applies

Important

- I am hopeful that you know how to convert back and forth between CFGs and PDAs.
- Hence everything we say about the decidability of problems concerning CFGs applies equally well to PDAs.

Important

- I am hopeful that you know how to convert back and forth between CFGs and PDAs.
- Hence everything we say about the decidability of problems concerning CFGs applies equally well to PDAs.
- Let us focus on emptiness testing.

Important

- I am hopeful that you know how to convert back and forth between CFGs and PDAs.
- Hence everything we say about the decidability of problems concerning CFGs applies equally well to PDAs.
- Let us focus on emptiness testing.
- Let

$$E_{CFG} = \{< G > | G \text{ is a CFG and } L(G) = \emptyset\}.$$

Important

- I am hopeful that you know how to convert back and forth between CFGs and PDAs.
- Hence everything we say about the decidability of problems concerning CFGs applies equally well to PDAs.
- Let us focus on emptiness testing.
- Let

$$E_{CFG} = \{< G > | G \text{ is a CFG and } L(G) = \emptyset\}.$$

Theorem 4.8

E_{CFG} is a decidable language.

Proof Idea

Proof Idea

- To find an algorithm for this problem, we might attempt to use Theorem 4.7.

Proof Idea

- To find an algorithm for this problem, we might attempt to use Theorem 4.7.
- It states that we can test whether a CFG generates some particular string w.

Proof Idea

- To find an algorithm for this problem, we might attempt to use Theorem 4.7.
- It states that we can test whether a CFG generates some particular string w.
- To determine whether $L(G) = \emptyset$

Proof Idea

- To find an algorithm for this problem, we might attempt to use Theorem 4.7.
- It states that we can test whether a CFG generates some particular string w.
- To determine whether $L(G) = \emptyset$
  - The algorithm might try going through all possible $w's$, one by one.

Proof Idea

- To find an algorithm for this problem, we might attempt to use Theorem 4.7.
- It states that we can test whether a CFG generates some particular string w.
- To determine whether $L(G) = \emptyset$
  - The algorithm might try going through all possible $w's$, one by one.
  - But there are infinitely many $w's$ to try.

Proof Idea

- To find an algorithm for this problem, we might attempt to use Theorem 4.7.
- It states that we can test whether a CFG generates some particular string w.
- To determine whether $L(G) = \emptyset$
  - The algorithm might try going through all possible $w's$, one by one.
  - But there are infinitely many $w's$ to try.
  - But this method could end up running forever.

Proof Idea

- To find an algorithm for this problem, we might attempt to use Theorem 4.7.
- It states that we can test whether a CFG generates some particular string w.
- To determine whether $L(G) = \emptyset$
  - The algorithm might try going through all possible $w's$, one by one.
  - But there are infinitely many $w's$ to try.
  - But this method could end up running forever.
  - We need to take a different approach

Proof Idea

Proof Idea

- In order to determine whether the language of a grammar is empty,

Proof Idea

- In order to determine whether the language of a grammar is empty,
  - We need to test whether the start variable can generate a string of terminals.

Proof Idea

- In order to determine whether the language of a grammar is empty,
    - We need to test whether the start variable can generate a string of terminals.
- The algorithm does so by solving a more general problem.

Proof Idea

- In order to determine whether the language of a grammar is empty,
  - We need to test whether the start variable can generate a string of terminals.
- The algorithm does so by solving a more general problem.
- It determines for each variable,

Proof Idea

- In order to determine whether the language of a grammar is empty,
  - We need to test whether the start variable can generate a string of terminals.
- The algorithm does so by solving a more general problem.
- It determines for each variable,
  - whether that variable is capable of generating a string of terminals.

Proof Idea

- In order to determine whether the language of a grammar is empty,
  - We need to test whether the start variable can generate a string of terminals.
- The algorithm does so by solving a more general problem.
- It determines for each variable,
  - whether that variable is capable of generating a string of terminals.
- When the algorithm has determined

Proof Idea

- In order to determine whether the language of a grammar is empty,
  - We need to test whether the start variable can generate a string of terminals.
- The algorithm does so by solving a more general problem.
- It determines for each variable,
  - whether that variable is capable of generating a string of terminals.
- When the algorithm has determined that a variable can generate some string of terminals,

Proof Idea

- In order to determine whether the language of a grammar is empty,
  - We need to test whether the start variable can generate a string of terminals.
- The algorithm does so by solving a more general problem.
- It determines for each variable,
  - whether that variable is capable of generating a string of terminals.
- When the algorithm has determined that a variable can generate some string of terminals,
  - The algorithm keeps track of this information by placing a mark on that variable.

Proof Idea

Proof Idea

- First, the algorithm marks all the terminal symbols in the grammar.

Proof Idea

- First, the algorithm marks all the terminal symbols in the grammar.
- Then, it scans all the rules of the grammar.

Proof Idea

- First, the algorithm marks all the terminal symbols in the grammar.
- Then, it scans all the rules of the grammar.
- If it ever finds a rule that

Proof Idea

- First, the algorithm marks all the terminal symbols in the grammar.
- Then, it scans all the rules of the grammar.
- If it ever finds a rule that permits some variable to be

Proof Idea

- First, the algorithm marks all the terminal symbols in the grammar.
- Then, it scans all the rules of the grammar.
- If it ever finds a rule that permits some variable to be replaced by some string of symbols,

Proof Idea

- First, the algorithm marks all the terminal symbols in the grammar.
- Then, it scans all the rules of the grammar.
- If it ever finds a rule that permits some variable to be replaced by some string of symbols,
  - The algorithm knows that this variable can be marked, too.

Proof Idea

- First, the algorithm marks all the terminal symbols in the grammar.
- Then, it scans all the rules of the grammar.
- If it ever finds a rule that permits some variable to be replaced by some string of symbols,
  - The algorithm knows that this variable can be marked, too.
- The algorithm continues in this way

Proof Idea

- First, the algorithm marks all the terminal symbols in the grammar.
- Then, it scans all the rules of the grammar.
- If it ever finds a rule that permits some variable to be replaced by some string of symbols,
  - The algorithm knows that this variable can be marked, too.
- The algorithm continues in this way until it cannot mark any additional variables.

Proof

$R =$ "On input $\langle G \rangle$, where G is a CFG:

1. Mark all terminal symbols in G.
2. Repeat until no new variables get marked:
3. Mark any variable A where G has a rule $A \rightarrow U_1 U_2 \cdots U_k$ and each symbol $U_1, \ldots, U_k$ has already been marked.
4. If the start variable is not marked, *accept*; otherwise, *reject*."

Our Focus

Our Focus

- Consider the problem of determining

Our Focus

- Consider the problem of determining**whether two context-free grammars generate the same language.**

Our Focus

- Consider the problem of determining**whether two context-free grammars generate the same language.**
- Let

$$EQ_{CFG} = \{< G, H > | G \text{ and } H \text{ are CFGs and } L(G) = L(H).$$

Decidability of EQ_{CFG}

Decidability of EQ_{CFG}

- We had discussed algorithm that decides

Decidability of EQ_{CFG}

- We had discussed algorithm that decides the analogous language EQ_{DFA} for finite automata.

Decidability of EQ_{CFG}

- We had discussed algorithm that decides the analogous language EQ_{DFA} for finite automata.
- We used the decision procedure for E_{DFA}

Decidability of EQ_{CFG}

- We had discussed algorithm that decides the analogous language EQ_{DFA} for finite automata.
- We used the decision procedure for E_{DFA} to prove that EQ_{DFA} is decidable.

Decidability of EQ_{CFG}

- We had discussed algorithm that decides the analogous language EQ_{DFA} for finite automata.
- We used the decision procedure for E_{DFA} to prove that EQ_{DFA} is decidable.
- Because E_{CFG} also is decidable,

Decidability of EQ_{CFG}

- We had discussed algorithm that decides the analogous language EQ_{DFA} for finite automata.
- We used the decision procedure for E_{DFA} to prove that EQ_{DFA} is decidable.
- Because E_{CFG} also is decidable,
    - We might think that we can use

Decidability of EQ_{CFG}

- We had discussed algorithm that decides the analogous language EQ_{DFA} for finite automata.
- We used the decision procedure for E_{DFA} to prove that EQ_{DFA} is decidable.
- Because E_{CFG} also is decidable,
  - We might think that we can use a similar strategy

Decidability of EQ_{CFG}

- We had discussed algorithm that decides the analogous language EQ_{DFA} for finite automata.
- We used the decision procedure for E_{DFA} to prove that EQ_{DFA} is decidable.
- Because E_{CFG} also is decidable,
  - We might think that we can use a similar strategy to prove that EQ_{CFG} is decidable.

Decidability of EQ_{CFG}

- We had discussed algorithm that decides the analogous language EQ_{DFA} for finite automata.
- We used the decision procedure for E_{DFA} to prove that EQ_{DFA} is decidable.
- Because E_{CFG} also is decidable,
  - We might think that we can use a similar strategy to prove that EQ_{CFG} is decidable.
- But something is wrong with this idea!

Decidability of EQ_{CFG}

- We had discussed algorithm that decides the analogous language EQ_{DFA} for finite automata.
- We used the decision procedure for E_{DFA} to prove that EQ_{DFA} is decidable.
- Because E_{CFG} also is decidable,
  - We might think that we can use a similar strategy to prove that EQ_{CFG} is decidable.
- But something is wrong with this idea!
- The class of context-free languages is not closed under

Decidability of EQ_{CFG}

- We had discussed algorithm that decides the analogous language EQ_{DFA} for finite automata.
- We used the decision procedure for E_{DFA} to prove that EQ_{DFA} is decidable.
- Because E_{CFG} also is decidable,
    - We might think that we can use a similar strategy to prove that EQ_{CFG} is decidable.
- But something is wrong with this idea!
- The class of context-free languages is not closed under
    - complementation or

Decidability of EQ_{CFG}

- We had discussed algorithm that decides the analogous language EQ_{DFA} for finite automata.
- We used the decision procedure for E_{DFA} to prove that EQ_{DFA} is decidable.
- Because E_{CFG} also is decidable,
  - We might think that we can use a similar strategy to prove that EQ_{CFG} is decidable.
- But something is wrong with this idea!
- The class of context-free languages is not closed under
  - complementation or
  - intersection.

Decidability of EQ_{CFG}

- We had discussed algorithm that decides the analogous language EQ_{DFA} for finite automata.
- We used the decision procedure for E_{DFA} to prove that EQ_{DFA} is decidable.
- Because E_{CFG} also is decidable,
  - We might think that we can use a similar strategy to prove that EQ_{CFG} is decidable.
- But something is wrong with this idea!
- The class of context-free languages is not closed under
  - complementation or
  - intersection.
- In fact, EQ_{CFG} **is not decidable.**

Decidability of EQ_{CFG}

- We had discussed algorithm that decides the analogous language EQ_{DFA} for finite automata.
- We used the decision procedure for E_{DFA} to prove that EQ_{DFA} is decidable.
- Because E_{CFG} also is decidable,
  - We might think that we can use a similar strategy to prove that EQ_{CFG} is decidable.
- But something is wrong with this idea!
- The class of context-free languages is not closed under
  - complementation or
  - intersection.
- In fact, EQ_{CFG} **is not decidable.**
- I will prove it after going through the technique.

CFLs are Decidable

CFLs are Decidable

Theorem 4.9

Every context-free language is decidable.

Proof Idea

Proof Idea

- Let A be a CFL.

Proof Idea

- Let A be a CFL.
- My objective is to show that A is decidable.

Proof Idea

- Let A be a CFL.
- My objective is to show that A is decidable.
- One (bad) idea !!!

Proof Idea

- Let A be a CFL.
- My objective is to show that A is decidable.
- One (bad) idea !!!
    - Convert a PDA for A directly into a TM.

Proof Idea

- Let A be a CFL.
- My objective is to show that A is decidable.
- One (bad) idea !!!
  - Convert a PDA for A directly into a TM.
  - That isnt hard to do.

Proof Idea

- Let A be a CFL.
- My objective is to show that A is decidable.
- One (bad) idea !!!
  - Convert a PDA for A directly into a TM.
  - That isnt hard to do.
  - Because simulating a stack

Proof Idea

- Let A be a CFL.
- My objective is to show that A is decidable.
- One (bad) idea !!!
  - Convert a PDA for A directly into a TM.
  - That isnt hard to do.
  - Because simulating a stack with the TMs more versatile tape is easy.

Proof Idea

- Let A be a CFL.
- My objective is to show that A is decidable.
- One (bad) idea !!!
  - Convert a PDA for A directly into a TM.
  - That isnt hard to do.
  - Because simulating a stack with the TMs more versatile tape is easy.
  - The PDA for A may be nondeterministic.

Proof Idea

- Let A be a CFL.
- My objective is to show that A is decidable.
- One (bad) idea !!!
  - Convert a PDA for A directly into a TM.
  - That isnt hard to do.
  - Because simulating a stack with the TMs more versatile tape is easy.
  - The PDA for A may be nondeterministic.
  - But that seems okay because

Proof Idea

- Let A be a CFL.
- My objective is to show that A is decidable.
- One (bad) idea !!!
  - Convert a PDA for A directly into a TM.
  - That isnt hard to do.
  - Because simulating a stack with the TMs more versatile tape is easy.
  - The PDA for A may be nondeterministic.
  - But that seems okay because **we can convert it into a nondeterministic TM**.

Proof Idea

- Let A be a CFL.
- My objective is to show that A is decidable.
- One (bad) idea !!!
  - Convert a PDA for A directly into a TM.
  - That isnt hard to do.
  - Because simulating a stack with the TMs more versatile tape is easy.
  - The PDA for A may be nondeterministic.
  - But that seems okay because **we can convert it into a nondeterministic TM**.
  - And we know that **any nondeterministic TM can be converted into an equivalent deterministic TM.**

Proof Idea

- Let A be a CFL.
- My objective is to show that A is decidable.
- One (bad) idea !!!
  - Convert a PDA for A directly into a TM.
  - That isnt hard to do.
  - Because simulating a stack with the TMs more versatile tape is easy.
  - The PDA for A may be nondeterministic.
  - But that seems okay because **we can convert it into a nondeterministic TM**.
  - And we know that **any nondeterministic TM can be converted into an equivalent deterministic TM.**
  - Yet, there is a difficulty!!!

Contd..

Contd..

- Some branches of the PDAs computation may go on forever.

Contd..

- Some branches of the PDAs computation may go on forever.
- Go forever reading and writing the stack **without ever halting.**

Contd..

- Some branches of the PDAs computation may go on forever.
- Go forever reading and writing the stack **without ever halting.**
- The simulating TM

Contd..

- Some branches of the PDAs computation may go on forever.
- Go forever reading and writing the stack **without ever halting.**
- The simulating TM then would also have **some non-halting branches in its computation**

Contd..

- Some branches of the PDAs computation may go on forever.
- Go forever reading and writing the stack **without ever halting.**
- The simulating TM then would also have **some non-halting branches in its computation**
- So the TM would not be a decider.

Contd..

- Some branches of the PDAs computation may go on forever.
- Go forever reading and writing the stack **without ever halting.**
- The simulating TM then would also have **some non-halting branches in its computation**
- So the TM would not be a decider.
- A different Idea is necessary!!!

Proof

Proof

- Let G be a CFG for A.

Proof

- Let G be a CFG for A.
- Design a TM M_G that decides A.

Proof

- Let G be a CFG for A.
- Design a TM M_G that decides A.
- We build a copy of G into M_G .

Proof

- Let G be a CFG for A.
- Design a TM M_G that decides A.
- We build a copy of G into M_G .
- It works as follows.

Proof

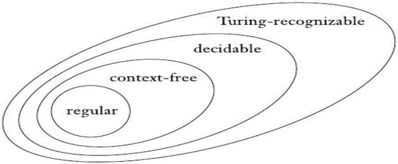
- Let G be a CFG for A.
- Design a TM M_G that decides A.
- We build a copy of G into M_G .
- It works as follows.

M_G = "On input w:

1. Run TM S on input $\langle G, w \rangle$.
2. If this machine accepts, *accept*; if it rejects, *reject*."

Relationships

Relationships



Undecidability

- I will prove one of the most philosophically important theorems of the theory of computation.
- There is a specific problem that is algorithmically unsolvable.
- Computers appear to be so powerful that you may believe that all problems will eventually yield to them.
- The theorem that I present strates that computers are limited in a fundamental way.

Unsolvable? What Sort of Problems?

- What sorts of problems are unsolvable by computer?
- Are they esoteric, dwelling only in the minds of theoreticians?
- **No!**
- Even some **ordinary problems turn out to be computationally unsolvable.**

Unsolvable Problem

- You are given a computer program and
- A precise specification of what that program is supposed to do
- For example, **sort a list of numbers**

Unsolvable Problem

- You are given a computer program and
- A precise specification of what that program is supposed to do
- For example, **sort a list of numbers**

Challenge

You need to verify that the program performs as specified (i.e., that it is correct).

What to Do?

- Both the program and the specification are mathematically precise objects.
- You hope to automate the process of verification by feeding these objects pause into a suitably programmed computer.
- However, you will be disappointed.

My Goal

- I will highlight several computationally unsolvable problems.
- We aim to help you develop a feeling for the types of problems that are unsolvable .
- To learn **techniques for proving unsolvability.**

Undecidability of a language

- Let us establishes the undecidability of a specific language.
- **The problem of determining whether a Turing machine accepts a given input string.**
- Let,

$$A_{TM} = \{< M, w > | M \text{ is a TM and } M \text{ accepts } w\}.$$

- Note: A_{DFA} and $A_C FG$ is decidable!!!.

Undecidability of a language

- Let us establishes the undecidability of a specific language.
- **The problem of determining whether a Turing machine accepts a given input string.**
- Let,

$$A_{TM} = \{< M, w > | M \text{ is a TM and } M \text{ accepts } w\}.$$

- Note: A_{DFA} and $A_C FG$ is decidable!!!. I have proved!!!

Theorem

Theorem 4.11

A_{TM} is undecidable.

Discussion

- Observe that A_{TM} is Turing-recognizable.
- It shows that recognizers are more powerful than deciders.
- Requiring a TM to halt on all inputs restricts

Discussion

- Observe that A_{TM} is Turing-recognizable.
- It shows that recognizers are more powerful than deciders.
- Requiring a TM to halt on all inputs restricts the kinds of languages that it can recognize.

Turing machine U to recognize A_{TM}

U = "On input $\langle M, w\rangle$, where M is a TM and w is a string:

1. Simulate M on input w.
2. If M ever enters its accept state, *accept*; if M ever enters its reject state, *reject*."

- If M loops on w, machine U loops on input $< M, w >$
- So machine does not decide A_{TM}.
- If the algorithm had some way to determine that M was not halting on w,
  - It could reject in this case.
- However, I will discuss that algorithm has no way to determine this.

- The Turing machine U is interesting.
- It is an example of the universal Turing machine first proposed by Alan Turing in 1936.
- This machine is called universal because
  - Tt is capable of simulating any other Turing machine from the description of that machine.
- The UTM played an important early role

Univeral Turing Machine

- The Turing machine U is interesting.
- It is an example of the universal Turing machine first proposed by Alan Turing in 1936.
- This machine is called universal because
  - Tt is capable of simulating any other Turing machine from the description of that machine.
- The UTM played an important early role in the development of **stored-program computers.**

THANK YOU