T065001: Introduction to Formal Languages

Lecture 10: Turing machines (2)

Chapters 3.2 - 3.3 in Sipser's textbook

2025-06-23

(Lecture slides by Yih-Kuen Tsay)

(From Chapter 3.1)

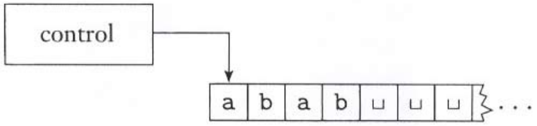


FIGURE 3.1
Schematic of a Turing machine

- The Turing machine model uses an infinite tape as its unlimited memory.
- It has a tape head that can read and write symbols and move around on the tape.
- Initially the tape contains only the input string and is blank everywhere else.
- If the machine needs to store information, it may write this information on the tape.

(From Chapter 3.1)

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ computes as follows.

- Initially, M receives input $w = w_1 w_2 \cdots w_n \in \Sigma^*$ on the leftmost n squares of the tape, and the rest of the tape is blank (i.e., filled with blank symbols).
- The head starts on the leftmost square of the tape.
- Note that Σ does not contain the blank symbol, so the first blank appearing on the tape marks the end of the input.
- Once M has started, the computation proceeds according to the rules described by the transition function.
- If M ever tries to move its head to the left off the left-hand end of the tape, the head stays in the same place for that move, even though the transition function indicates L.
- The computation continues until it enters either the accept or reject states, at which point it halts. If neither occurs, M goes on forever.

$\Rightarrow$ Three possible outcomes: Any computation will accept, reject, or **loop**.

Variants of Turing Machines

- Alternative definitions of Turing machines abound, including versions with *multiple tapes* or with *nondeterminism*. They are called *variants* of the Turing machine model.
- The original model and its reasonable variants all have the same power—*they recognize the same class of languages*.
- To show that two models are equivalent, we simply need to show that we can *simulate* one by the other.

Variants of Turing Machines

- Alternative definitions of Turing machines abound, including versions with *multiple tapes* or with *nondeterminism*. They are called *variants* of the Turing machine model.
- The original model and its reasonable variants all have the same power—*they recognize the same class of languages*.
- To show that two models are equivalent, we simply need to show that we can *simulate* one by the other.

Example:

- (The original TM model) $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\text{L}, \text{R}\}$

Variants of Turing Machines

- Alternative definitions of Turing machines abound, including versions with *multiple tapes* or with *nondeterminism*. They are called *variants* of the Turing machine model.
- The original model and its reasonable variants all have the same power—*they recognize the same class of languages*.
- To show that two models are equivalent, we simply need to show that we can *simulate* one by the other.

Example:

- (The original TM model) $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathsf{L}, \mathsf{R}\}$
- (TM with the ability to "stay put") $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathsf{L}, \mathsf{R}, \mathsf{S}\}$

Variants of Turing Machines

- Alternative definitions of Turing machines abound, including versions with *multiple tapes* or with *nondeterminism*. They are called *variants* of the Turing machine model.
- The original model and its reasonable variants all have the same power—*they recognize the same class of languages*.
- To show that two models are equivalent, we simply need to show that we can *simulate* one by the other.

Example:

- (The original TM model) $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathsf{L}, \mathsf{R}\}$
- (TM with the ability to "stay put") $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathsf{L}, \mathsf{R}, \mathsf{S}\}$

As we will see in this week's exercises, each of these two models can simulate the other. $\Rightarrow$ Robustness of the Turing machine model.

Multitape Turing Machines

- A *multitape Turing machine* is like an ordinary Turing machine with several tapes.
- Each tape has its own head for reading and writing. Initially the input appears on tape 1 and the others start out blank.
- The transition function is changed to allow for reading, writing, and moving the heads on all the tapes simultaneously. Formally,

  $$\delta : Q \times \Gamma^k \longrightarrow Q \times \Gamma^k \times \{L, R, S\}^k,$$

  where k is the number of tapes.

Multitape Turing Machines (cont.)

Theorem (3.13)

Every multitape Turing machine has an equivalent single-tape Turing machine.

Multitape Turing Machines (cont.)

Theorem (3.13)

Every multitape Turing machine has an equivalent single-tape Turing machine.

A single tape TM S can simulate a k-tape TM M as follows:

1. S "formats" its tape to represent all k tapes of M:

$$\# \overset{\bullet}{w_1}\, w_2 \cdots w_n \# \overset{\bullet}{\sqcup} \# \overset{\bullet}{\sqcup} \# \cdots \#$$

Multitape Turing Machines (cont.)

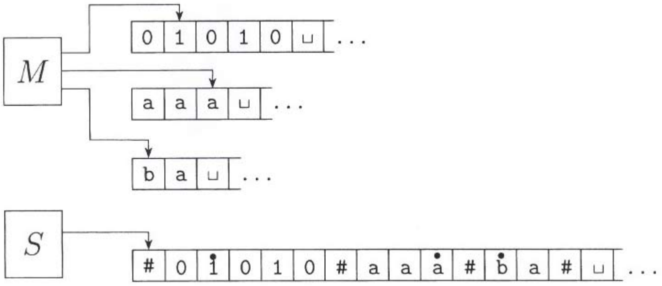


FIGURE 3.14
Representing three tapes with one

Multitape Turing Machines (cont.)

Theorem (3.13)

Every multitape Turing machine has an equivalent single-tape Turing machine.

A single tape TM S can simulate a k-tape TM M as follows:

1. S "formats" its tape to represent all k tapes of M:

$$\# \overset{\bullet}{w_1} w_2 \cdots w_n \# \overset{\bullet}{\sqcup} \# \overset{\bullet}{\sqcup} \# \cdots \#$$

2. To simulate a single move of M, S scans its tape to determine the symbols under the virtual heads. Then S makes a second pass to update the tapes according to M's transition function.
3. Whenever a virtual head is moved to the right onto a $\#$, S writes a blank symbol on this tape cell and shifts the tape contents from this cell one unit to the right.

Nondeterministic Turing Machines

- A **nondeterministic Turing machine** is defined in the expected way.
- The transition function of a nondeterministic TM has the form

$$\delta : Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\}).$$

- The computation of a nondeterministic TM is a tree whose branches correspond to different possibilities for the machine.
- If some branch of the computation leads to the accept state, the machine accepts its input.

Nondeterministic Turing Machines (cont.)

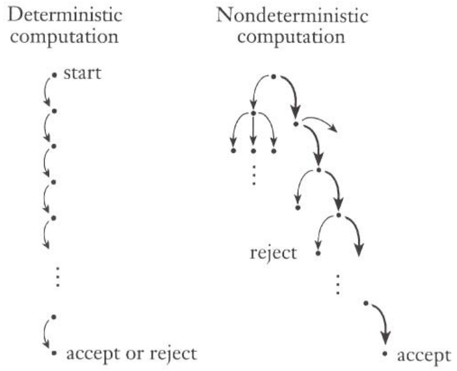


FIGURE 1.28
Deterministic and nondeterministic computations with an accepting branch

Nondeterministic Turing Machines (cont.)

Theorem (3.16)

Every nondeterministic TM has an equivalent deterministic TM.

Nondeterministic Turing Machines (cont.)

Theorem (3.16)

Every nondeterministic TM has an equivalent deterministic TM.

- The idea is to have a deterministic TM D try all possible branches of the given nondeterministic TM N's computation.
- D searches, in a breadth first manner, N's computation tree for an accepting configuration.

Nondeterministic Turing Machines (cont.)

D has three tapes:

- Tape 1 always contains the input string and is never altered.
- Tape 2 maintains a copy of *N*'s tape on some branch of its nondeterministic computation.
- Tape 3 keeps track of *D*'s location in *N*'s nondeterministic computation tree.

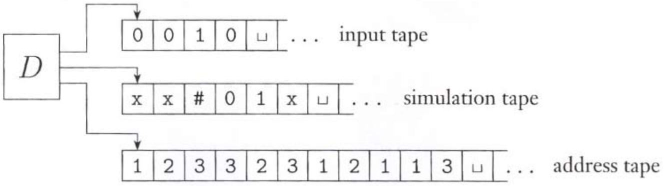

D
0 0 1 0 ␣ . . . input tape
x x # 0 1 x ␣ . . . simulation tape
1 2 3 3 2 3 1 2 1 1 3 ␣ . . . address tape

FIGURE 3.17
Deterministic TM *D* simulating nondeterministic TM *N*

Nondeterministic Turing Machines (cont.)

- Let's first consider the data representation on tape 3.
- Every node in the tree can have at most b children, where b is the size of the largest set of possible choices given by N's transition function.
- To every node in the tree we assign an address that is a string over the alphabet $\Gamma_b = \{1, 2, ..., b\}$.
- We assign the address 231 to the node we arrive at by starting at the root, going to its 2nd child, going to that node's 3rd child, and finally going to that node's 1st child.
- Each symbol in the string tells us which choice to make next when simulating a step in one branch in N's nondeterministic computation.
- The empty string is the address of the root of the tree.

Nondeterministic Turing Machines (cont.)

- Sometimes a symbol may not correspond to any choice if too few choices are available for a configuration.
- In that case, the address is invalid and doesn't correspond to any node.
- Tape 3 contains a string over Γ_b. It represents the branch of N's computation from the root to the node addressed by that string unless the address is invalid.

Nondeterministic Turing Machines (cont.)

- Sometimes a symbol may not correspond to any choice if too few choices are available for a configuration.
- In that case, the address is invalid and doesn't correspond to any node.
- Tape 3 contains a string over Γ_b. It represents the branch of N's computation from the root to the node addressed by that string unless the address is invalid.

Definition: A list of strings is in shortlex order if it is arranged so that shorter strings always come before longer strings and strings of equal length are sorted in lexicographic order.

Example: $\{0, 1\}^*$ listed in shortlex order is $\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots$

Nondeterministic Turing Machines (cont.)

To generate the strings in Γ_b^* in shortlex order on tape 3, where $\Gamma_b = \{1, 2, \ldots, b\}$, the Turing machine D can use the following strategy:

Nondeterministic Turing Machines (cont.)

To generate the strings in Γ_b^* in shortlex order on tape 3, where $\Gamma_b = \{1, 2, \ldots, b\}$, the Turing machine D can use the following strategy:

- Start with an empty tape.
- Whenever the next string is needed, check which of the following two cases holds and update the tape accordingly:
  - *If the tape currently contains a symbol from* $\{1, 2, \ldots, b-1\}$*:*
    Find the rightmost symbol on the tape that belongs to $\{1, 2, \ldots, b-1\}$ and increment it, i.e., if it was i then change it to $i+1$.
    Next, change all bs to the right of this position (if any) to 1s.
  - *Otherwise (the tape only contains bs and blanks):*
    Change all bs (if any) to 1s and change the first blank to 1.

Example: $(b = 3)$

$\varepsilon \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 21 \rightarrow \ldots \rightarrow 33 \rightarrow 111 \rightarrow 112 \rightarrow \ldots$

High-level description of the deterministic, three-tape Turing machine D that simulates the nondeterministic Turing machine N:

High-level description of the deterministic, three-tape Turing machine D that simulates the nondeterministic Turing machine N:

1. Initially, tape 1 contains the input w, and tapes 2 and 3 are empty.
2. Copy tape 1 to tape 2.
3. Use tape 2 to simulate N with input w on one branch of its nondeterministic computation. Before each step of N, consult the next symbol on tape 3 to determine which choice to make among those allowed by N's transition function. If no more symbols remain on tape 3 or if this nondeterministic choice is invalid, abort this branch by going to stage 4. Also go to stage 4 if a rejecting configuration is encountered. If an accepting configuration is encountered, *accept* the input.
4. Replace the string on tape 3 with the next string from Γ_b^* in shortlex order. Simulate the next branch of N's computation by going to stage 2.

High-level description of the deterministic, three-tape Turing machine D that simulates the nondeterministic Turing machine N:

1. Initially, tape 1 contains the input w, and tapes 2 and 3 are empty.
2. Copy tape 1 to tape 2.
3. Use tape 2 to simulate N with input w on one branch of its nondeterministic computation. Before each step of N, consult the next symbol on tape 3 to determine which choice to make among those allowed by N's transition function. If no more symbols remain on tape 3 or if this nondeterministic choice is invalid, abort this branch by going to stage 4. Also go to stage 4 if a rejecting configuration is encountered. If an accepting configuration is encountered, *accept* the input.
4. Replace the string on tape 3 with the next string from Γ_b^* in shortlex order. Simulate the next branch of N's computation by going to stage 2.

Finally, to conclude the proof of Theorem 3.16, use Theorem 3.13 which says that D has an equivalent deterministic, single-tape Turing machine S.

(From Chapter 3.1)

Definition (3.5)

A language is **Turing-recognizable** (also called *recursively enumerable*) if some Turing machine recognizes it.

- A Turing machine can fail to accept an input by entering the q_{reject} state and rejecting, or by looping (not halting).
- A machine is called a *decider* if it halts on all inputs. A decider that recognizes some language is said to *decide* the language.

Definition (3.6)

A language is **Turing-decidable**, or simply **decidable** (also called *recursive*), if some Turing machine decides it.

(From Chapter 3.1)

Definition (3.5)

A language is **Turing-recognizable** (also called *recursively enumerable*) if some Turing machine recognizes it.

Questions:

Where does the term "enumerable" come from?

What's the connection between recognizing and enumerating a language?

Enumerators

- An *enumerator* is a Turing machine with an attached printer. Every time the Turing machine wants to add a string to the output list, it sends the string to the printer.
- The language enumerated by an enumerator E is the collection of all the strings that E eventually prints out.
- Moreover, E may generate the strings of the language in any order, possibly with repetitions.

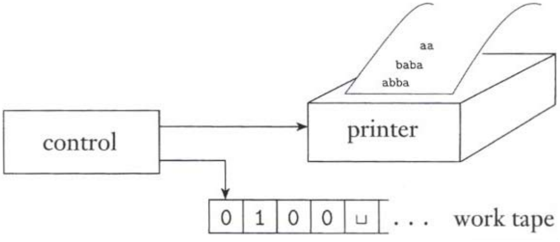


FIGURE **3.20**
Schematic of an enumerator

Enumerators (cont.)

One can think of an enumerator as a type of two-tape Turing machine that uses its second tape as the printer. The computation of an enumerator E is defined as for an ordinary TM, except that:

Enumerators (cont.)

One can think of an enumerator as a type of two-tape Turing machine that uses its second tape as the printer. The computation of an enumerator E is defined as for an ordinary TM, except that:

- E has two tapes, a work tape and a print tape, both initially blank.

Enumerators (cont.)

One can think of an enumerator as a type of two-tape Turing machine that uses its second tape as the printer. The computation of an enumerator E is defined as for an ordinary TM, except that:

- E has two tapes, a work tape and a print tape, both initially blank.
- Transitions are of the form $\delta(q, a) = (r, b, L, c)$ (or $\delta(q, a) = (r, b, R, c)$), meaning that if E is in state q and reads an a then it enters state r, writes a b on the work tape, and moves the work tape head left (or right).
  E also writes a c on the print tape, and if $c \neq \varepsilon$ then it moves the print tape head right.

Enumerators (cont.)

One can think of an enumerator as a type of two-tape Turing machine that uses its second tape as the printer. The computation of an enumerator E is defined as for an ordinary TM, except that:

- E has two tapes, a work tape and a print tape, both initially blank.
- Transitions are of the form $\delta(q, a) = (r, b, L, c)$ (or $\delta(q, a) = (r, b, R, c)$), meaning that if E is in state q and reads an a then it enters state r, writes a b on the work tape, and moves the work tape head left (or right).
  E also writes a c on the print tape, and if $c \neq \varepsilon$ then it moves the print tape head right.
- Whenever a special state q_{print} is entered, the print tape is reset to blank and the print tape head returns to the left-hand end.

Enumerators (cont.)

One can think of an enumerator as a type of two-tape Turing machine that uses its second tape as the printer. The computation of an enumerator E is defined as for an ordinary TM, except that:

- E has two tapes, a work tape and a print tape, both initially blank.
- Transitions are of the form $\delta(q, a) = (r, b, L, c)$ (or $\delta(q, a) = (r, b, R, c)$), meaning that if E is in state q and reads an a then it enters state r, writes a b on the work tape, and moves the work tape head left (or right).
  E also writes a c on the print tape, and if $c \neq \varepsilon$ then it moves the print tape head right.
- Whenever a special state q_{print} is entered, the print tape is reset to blank and the print tape head returns to the left-hand end.
- $L(E) = \{w \in \Sigma^* \mid$ at least once when entering q_{print}, the output tape contains $w\}$

Enumerators (cont.)

Theorem (3.21)

A language is Turing-recognizable if and only if some enumerator enumerates it.

Enumerators (cont.)

Theorem (3.21)

A language is Turing-recognizable if and only if some enumerator enumerates it.

$\Leftarrow$)

Let E be an enumerator.
To recognize the language enumerated by E, a TM M works as follows:

```
$M$ = "On input $w$:
    1. Run $E$. Every time that $E$ outputs a string, compare it with $w$.
    2. If $w$ ever appears in the output of $E$, accept."
```

Then M accepts those strings that appear somewhere in E's output.

Enumerators (cont.)

$\Rightarrow$)

Let M be a TM and let $s_1, s_2, s_3, \ldots$ be a list of all strings in Σ^* in shortlex order (i.e., arranged so that shorter strings always come before longer strings and strings of equal length are sorted in lexicographic order).

Enumerators (cont.)

$\Rightarrow$)

Let M be a TM and let $s_1, s_2, s_3, \ldots$ be a list of all strings in Σ^* in shortlex order (i.e., arranged so that shorter strings always come before longer strings and strings of equal length are sorted in lexicographic order).

To enumerate the language recognized by M, an enumerator E works as follows:

```
$E =$ "Ignore the input.
    1. Repeat the following for $i = 1, 2, 3, \ldots$.
    2.     Run $M$ for $i$ steps on each input, $s_1, s_2, \ldots, s_i$.
    3.     If any computations accept, print out the corresponding $s_j$."
```

Enumerators (cont.)

$\Rightarrow$)

Let M be a TM and let $s_1, s_2, s_3, \ldots$ be a list of all strings in Σ^* in shortlex order (i.e., arranged so that shorter strings always come before longer strings and strings of equal length are sorted in lexicographic order).

To enumerate the language recognized by M, an enumerator E works as follows:

```
E = "Ignore the input.
    1. Repeat the following for $i = 1, 2, 3, \ldots$.
    2.     Run $M$ for $i$ steps on each input, $s_1, s_2, \ldots, s_i$.
    3.     If any computations accept, print out the corresponding $s_j$."
```

(The above technique is known in the literature as dovetailing.)

This works because for any string s_i, if s_i is accepted by M after x steps for some positive integer x then s_i will eventually be printed by E.

Also, repetitions in the output are OK by the definition of an enumerator.

The Church-Turing thesis

- All models of a general-purpose computer turn out to be at best equivalent in power to the Turing machine, as long as they satisfy certain reasonable requirements.
- This has an important philosophical corollary: Even though there are many different computational models, *the class of algorithms that they describe is unique*.

The Church-Turing thesis

- All models of a general-purpose computer turn out to be at best equivalent in power to the Turing machine, as long as they satisfy certain reasonable requirements.
- This has an important philosophical corollary: Even though there are many different computational models, *the class of algorithms that they describe is unique*.
- The **Church-Turing thesis** says that *the intuitive notion of an algorithm corresponds to the formal definition of a Turing machine*.

  No proof, but is supported by the observation that no reasonable computational model that has been proposed so far is more powerful than the standard Turing machine model in terms of what can be computed.

Hilbert's Tenth Problem

10. Determination of the solvability of a Diophantine equation. Given a diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined by a finite number of operations whether the equation is solvable in rational integers.

Note: the kind of process that Hilbert looked after is “effective procedure” and is nowadays referred to as “computer algorithm” or simply “algorithm.”

Hilbert's Tenth Problem

- A *polynomial* is a sum of terms, where each term is a product of variables and a constant.
- For example, $6x^3yz^2 + 3xy^2 - x^3 - 10$ is a polynomial with four terms over variables x, y, and z.
- Let $D = \{p \mid p \text{ is a polynomial with a set of integer zeros}\}$
- Hilbert's tenth problem (rephrased): "Is there an algorithm for deciding D?"

  Hilbert implicitly assumed that the answer is "yes".

Hilbert's Tenth Problem

- A *polynomial* is a sum of terms, where each term is a product of variables and a constant.
- For example, $6x^3yz^2 + 3xy^2 - x^3 - 10$ is a polynomial with four terms over variables x, y, and z.
- Let $D = \{p \mid p \text{ is a polynomial with a set of integer zeros}\}$
- Hilbert's tenth problem (rephrased): "Is there an algorithm for deciding D?"

  Hilbert implicitly assumed that the answer is "yes".
- Proving that no algorithm exists for a particular task requires a precise definition of algorithm.

  $\Rightarrow$ Mathematicians of that generation failed to resolve the question.

  Much later, in 1970, Matiyasevich finally proved that no algorithm for deciding D exists.

Hilbert's Tenth Problem

- A *polynomial* is a sum of terms, where each term is a product of variables and a constant.
- For example, $6x^3yz^2 + 3xy^2 - x^3 - 10$ is a polynomial with four terms over variables x, y, and z.
- Let $D = \{p \mid p \text{ is a polynomial with a set of integer zeros}\}$
- Hilbert's tenth problem (rephrased): "Is there an algorithm for deciding D?"

  Hilbert implicitly assumed that the answer is "yes".
- Proving that no algorithm exists for a particular task requires a precise definition of algorithm.

  $\Rightarrow$ Mathematicians of that generation failed to resolve the question.

  Much later, in 1970, Matiyasevich finally proved that no algorithm for deciding D exists.

In chapters 4 and 5, we'll see many examples of **undecidable** languages.

Describing Turing Machines

- We now set up a format and notation for describing Turing machines.
- The input to a Turing machine is always a string. If we want to provide an object other than a string as input, we must first represent that object as a string.
- Strings can easily represent polynomials, graphs, grammars, automata, and any combination of those objects.
- A Turing machine may be programmed to decode the representation so that it can be interpreted in the way we intend. Our notation for the encoding of an object O into its representation as a string is $< O >$.
- If we have several objects $O_1, \cdots, O_k$, we denote their encoding into a single string $< O_1, \cdots, O_k >$.
- The encoding itself can be done in many reasonable ways. It doesn't matter which one we pick because a Turing machine can always translate one such encoding into another.

Describing Turing Machines (cont.)

Three different levels of detail:

- A *formal description* spells out in full the Turing machine's states, transition function, and so on.
- In an *implementation description*, we use natural language prose to describe the way that the Turing machine moves its head and the way that it stores data on its tape.
- In a *high-level description*, we use natural language prose to describe an algorithm, ignoring the implementation model.

An Example High-Level Description

An undirected graph is connected if every vertex can be reached from every other vertex by traveling along the edges of the graph.

An Example High-Level Description

An undirected graph is connected if every vertex can be reached from every other vertex by traveling along the edges of the graph.

Let $A = \{\langle G \rangle \mid G \text{ is a connected undirected graph}\}$. The following is a high-level description of a TM M that decides A:

An Example High-Level Description

An undirected graph is connected if every vertex can be reached from every other vertex by traveling along the edges of the graph.

Let $A = \{\langle G \rangle \mid G \text{ is a connected undirected graph}\}$. The following is a high-level description of a TM M that decides A:

$M =$ "On input $\langle G \rangle$, the encoding of a graph G:

1. Select the first node of G and mark it.
2. Repeat Step 3 until no new nodes are marked.
3. For each node in G, mark it if it is attached by an edge to a node that is already marked.
4. Scan all the nodes of G to determine whether they all are marked. If they are, *accept*; otherwise, *reject*."

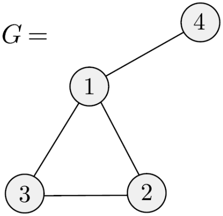
Implementation-Level Description

- Let's examine some implementation-level details of Turing machine M.
- First, we must understand how $< G >$ encodes the graph G as a string.

Implementation-Level Description

- Let's examine some implementation-level details of Turing machine M.
- First, we must understand how $< G >$ encodes the graph G as a string.
- Consider an encoding that is a list of the nodes of G followed by a list of the edges of G.
- Each node is a decimal number, and each edge is the pair of decimal numbers that represent the nodes at the two endpoints of the edge.
- The following figure depicts such a graph and its encoding.

$G =$



$\langle G \rangle =$

`(1,2,3,4)((1,2),(2,3),(3,1),(1,4))`

Implementation-Level Description (cont.)

- When M receives the input $< G >$, it first checks to determine whether the input is the proper encoding of some graph.

Implementation-Level Description (cont.)

- When M receives the input $< G >$, it first checks to determine whether the input is the proper encoding of some graph.
- To do so, M scans the tape to be sure that there are two lists and that they are in the proper form.
- The first list should be a list of distinct decimal numbers, and the second should be a list of pairs of decimal numbers.

Implementation-Level Description (cont.)

- When M receives the input $< G >$, it first checks to determine whether the input is the proper encoding of some graph.
- To do so, M scans the tape to be sure that there are two lists and that they are in the proper form.
- The first list should be a list of distinct decimal numbers, and the second should be a list of pairs of decimal numbers.
- Then M checks several things.
- First, the node list should contain no repetitions; and second, every node appearing on the edge list should also appear on the node list.
- If the input passes these checks, it is the encoding of some graph G.
- This verification completes the input check, and M goes on to stage 1.

Implementation-Level Description (cont.)

For stage 1, M marks the first node with a dot on the leftmost digit.

Implementation-Level Description (cont.)

For stage 1, M marks the first node with a dot on the leftmost digit.

For stage 2, M scans the list of nodes to find an undotted node n_1 and flags it by marking it differently—say, by underlining the first symbol. Then M scans the list again to find a dotted node n_2 and underlines it, too.

Implementation-Level Description (cont.)

For stage 1, M marks the first node with a dot on the leftmost digit.

For stage 2, M scans the list of nodes to find an undotted node n_1 and flags it by marking it differently—say, by underlining the first symbol. Then M scans the list again to find a dotted node n_2 and underlines it, too.

Now M scans the list of edges. For each edge, M tests whether the two underlined nodes n_1 and n_2 are the ones appearing in that edge. If they are, M dots n_1, removes the underlines, and goes on from the beginning of stage 2. If they aren't, M checks the next edge on the list. If there are no more edges, $\{n_1, n_2\}$ is not an edge of G. Then M moves the underline on n_2 to the next dotted node and now calls this node n_2. It repeats the steps in this paragraph to check, as before, whether the new pair $\{n_1, n_2\}$ is an edge. If there are no more dotted nodes, n_1 is not attached to any dotted nodes. Then M sets the underlines so that n_1 is the next undotted node and n_2 is the first dotted node and repeats the steps in this paragraph. If there are no more undotted nodes, M has not been able to find any new nodes to dot, so it moves on to stage 4.

Implementation-Level Description (cont.)

For stage 1, M marks the first node with a dot on the leftmost digit.

For stage 2, M scans the list of nodes to find an undotted node n_1 and flags it by marking it differently—say, by underlining the first symbol. Then M scans the list again to find a dotted node n_2 and underlines it, too.

Now M scans the list of edges. For each edge, M tests whether the two underlined nodes n_1 and n_2 are the ones appearing in that edge. If they are, M dots n_1, removes the underlines, and goes on from the beginning of stage 2. If they aren't, M checks the next edge on the list. If there are no more edges, $\{n_1, n_2\}$ is not an edge of G. Then M moves the underline on n_2 to the next dotted node and now calls this node n_2. It repeats the steps in this paragraph to check, as before, whether the new pair $\{n_1, n_2\}$ is an edge. If there are no more dotted nodes, n_1 is not attached to any dotted nodes. Then M sets the underlines so that n_1 is the next undotted node and n_2 is the first dotted node and repeats the steps in this paragraph. If there are no more undotted nodes, M has not been able to find any new nodes to dot, so it moves on to stage 4.

For stage 4, M scans the list of nodes to determine whether all are dotted. If they are, it enters the accept state; otherwise, it enters the reject state. This completes the description of TM M. ■