

## Discrete Maths – Definitions

### Set Theory

A relation  $R$  on a set  $A$  is called:

- Reflexive: iff  $\forall x \in A, xRx$
- Symmetric: iff  $\forall x, y \in A, xRy \rightarrow yRx$
- Transitive: iff  $\forall x, y, z \in A, xRy \wedge yRz \rightarrow xRz$
- Anti-Symmetric: iff  $\forall x, y \in A$  s.t.  $xRy \wedge yRx$ , then  $x = y$

An equivalence relation on  $A$  is a relation that is reflexive, symmetric and transitive.

An partial order on  $A$  is a relation that is reflexive, anti-symmetric and transitive.

---

### Functions

Certain functions have different properties based on their characteristics:

- Surjective: iff  $f(x) = f(y) \Rightarrow x = y$
  - Injective: A function  $f: A \rightarrow B$  is injective iff  $\forall z \in B \exists x \in A$  s.t.  $f(x) = z$
  - Bijjective: A function  $f: A \rightarrow B$  is called bijective if  $f$  is both injective and surjective
- 

### Abstract Algebra

A binary operation  $*$  on  $A$  is an operation applied to any two elements  $x, y \in A$  that yields an element  $x * y$  in  $A$

A binary operation  $*$  on a set  $A$  is called:

- Commutative: if  $\forall x, y \in A, x * y = y * x$
- Associative: if  $\forall x, y, z \in A, (x * y) * z = x * (y * z)$

A Semigroup is a set endowed with an associative binary operation. We denote the semigroup  $(A, *)$

An Identity Element is an element  $e \in A$  s.t.  $e * x = x * e = x, \forall x \in A$

- e.g.
1.  $(\mathbb{R}, +)$  has 0 as the identity element.
  2.  $(\mathbb{R}, \times)$  has 1 as the identity element.

An **Monoid** is a set  $A$  endowed with an associative binary operation  $*$  that has an identity element  $e$ . In other words, a monoid is a semigroup  $(A, *)$ , where  $*$  has an identity element  $e$

- e.g
1.  $(\mathbb{R}, +)$  is a commutative monoid with  $e = 0$ .
  2.  $(\mathbb{R}, \times)$  is a commutative monoid with  $e = 1$ .

An **Group** is a monoid in which every element is invertible. A group  $(A, *, e)$  is called commutative or **Abelian** if its operation  $*$  is commutative.

An **Homomorphism** is a function  $f : A \rightarrow B$  which acts on two semigroups, monoids, or groups  $(A, *)$  and  $(B, *)$  where  $f(x * y) = f(x) * f(y) \forall x, y \in A$ . In other words, if  $f$  is a function that respects (behaves well with respect to) the binary operation.

Let  $(A, *)$  and  $(B, *)$  both be semigroups, monoids or groups. A function  $f : A \rightarrow B$  is called an **isomorphism** if  $f : A \rightarrow B$  is both bijective AND a homomorphism.

---

## **Formal Languages**

Let  $A$  be a finite set. When studying formal languages, we call  $A$  an **alphabet** and the elements of  $A$  **letters**.

$\forall n \in \mathbb{N}^*$ , we define a **word** of length  $n$  in the alphabet  $A$  as being any string of the form  $a^1, a^2, \dots, a^n$  s.t.  $a^i \in A \forall i, 1 \leq i \leq n$

Let  $A$  be a finite set. A **language** over  $A$  is a subset of  $A^*$ . A language  $L$  over  $A$  is called a formal language if  $\exists$  a finite set of rules or algorithm that generates exactly  $L$ , i.e. all words that belong to  $L$  and no other words.

A (formal) grammar is a set of production rules for strings in a language.

To generate a language we use:

1. the set  $A$ , which is the **alphabet** of the language;
2. a **start symbol**  $\langle s \rangle$ ;
3. a set of **production rules**.

Example:

$A = \{0, 1\}$   
Start Symbol  $\langle s \rangle$

2 production rules given by:

1.  $\langle s \rangle \rightarrow 0\langle s \rangle 1$
2.  $\langle s \rangle \rightarrow 01$

We saw 2 types of strings that appeared in this process of generating L:

1. Terminals, i.e. the elements of A
2. Nonterminals, i.e. strings that don't consist solely of 0's and 1's such as  $\langle s \rangle$ ,  $0\langle s \rangle 1$ ,  $00\langle s \rangle 11$ , etc.

Grammars come in two flavours:

1. Context-free grammars where we can replace any occurrence of  $\langle T \rangle$  by  $w$  if  $\langle T \rangle \rightarrow w$  is one of our production rules.
2. Context-sensitive grammars only certain replacements of  $\langle T \rangle$  by  $w$  are allowed, which are governed by the syntax of our language L.

**Notation:**  $( \overset{V}{\text{set of terminals and non terminals}}, \overset{A}{\text{set of terminals}}, \overset{\langle s \rangle}{\text{start symbol}}, \overset{P}{\text{set of production rules}} )$

A language L generated by a context-free grammar is called a context-free language.

A regular language  $\Leftrightarrow$  it can be recognized by a finite state acceptor, which is a type of finite state machine.

A finite state acceptor (S, A, i, t, F) consists of:

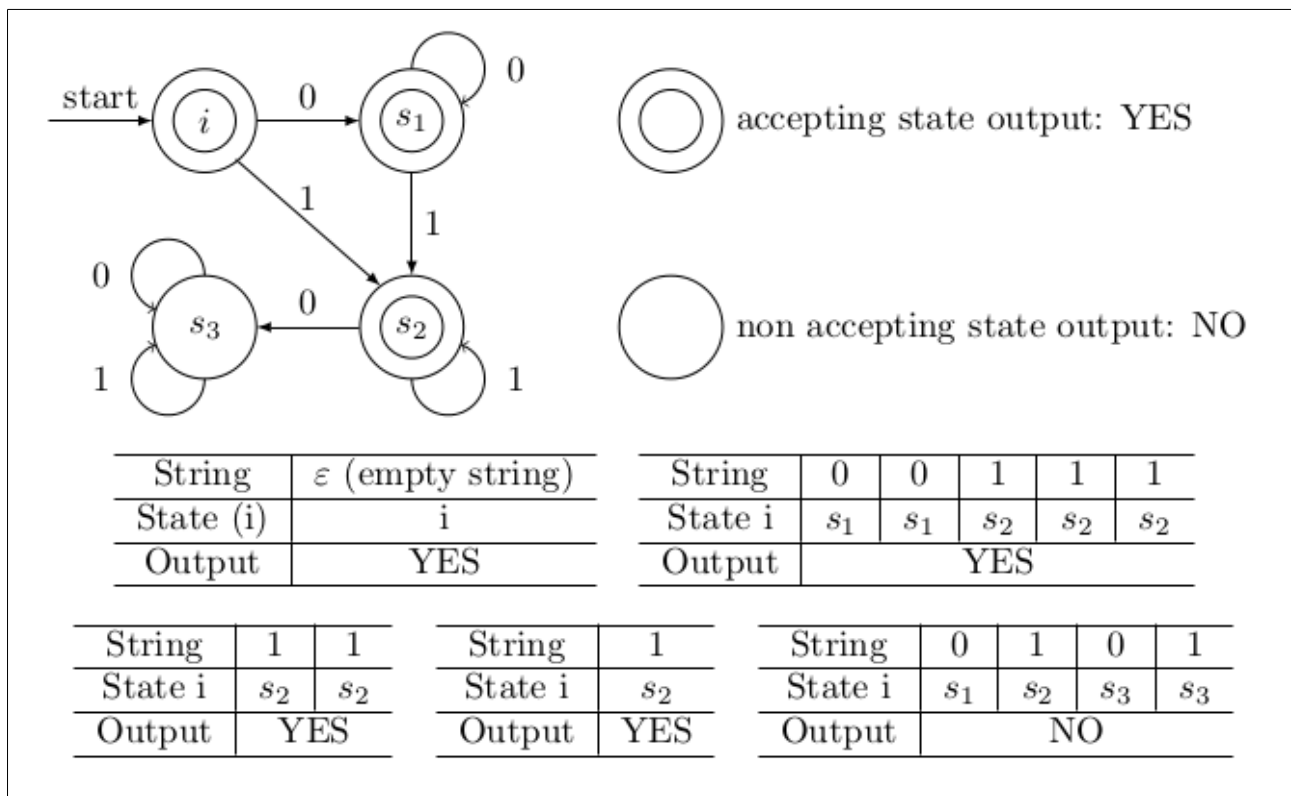
- S = A finite set S of states
- A = A finite set A that is the input alphabet
- i = A starting state  $i \in S$
- t = A transition mapping  $t : S \times A \rightarrow S$
- F = A set F of finishing states, where  $F \subseteq S$

Example:

*Build a deterministic finite state acceptor for the regular language*

$L = \{0^m 1^n \mid m, n \in \mathbb{N}, m \geq 0, n \geq 0\}$

- Accepting states: i, S<sub>1</sub>, S<sub>2</sub>
- Non accepting states: S<sub>3</sub>
- Start states: i
- S = {i, S<sub>1</sub>, S<sub>2</sub>, S<sub>3</sub>}
- F = {i, S<sub>1</sub>, S<sub>2</sub>}
- A = {0, 1}
- t :  $S \times A \rightarrow S$
- $t(i, 0) = S_1$
- $t(i, 1) = S_2$
- $t(S_1, 0) = S_1$
- $t(S_1, 1) = S_2$
- $t(S_2, 0) = S_3$
- $t(S_2, 1) = S_2$



A context-free grammar  $(V, A, \langle s \rangle, P)$  is called a regular grammar if every production rule in  $P$  is of one of the three forms:

- (i)  $\langle A \rangle \rightarrow b \langle B \rangle$
- (ii)  $\langle A \rangle \rightarrow b$
- (iii)  $\langle A \rangle \rightarrow \epsilon$

Recall the regular language  $L = \{0^m 1^n \mid m, n \in \mathbb{N}, m \geq 0, n \geq 0\}$ . We can generate it from the regular grammar in normal form given by production rules:

1.  $\langle s \rangle \rightarrow 0 \langle A \rangle$
2.  $\langle A \rangle \rightarrow 0 \langle A \rangle$
3.  $\langle A \rangle \rightarrow \epsilon$
4.  $\langle s \rangle \rightarrow \epsilon$
5.  $\langle A \rangle \rightarrow 1 \langle B \rangle$
6.  $\langle B \rangle \rightarrow 1 \langle B \rangle$
7.  $\langle S \rangle \rightarrow 1 \langle B \rangle$
8.  $\langle B \rangle \rightarrow \epsilon$

All regular languages can be given by a regular expression.

Examples:

(1)  $A = \{0, 1\}$

$$\begin{aligned}
 1^* \circ 0 &= \{w \in A^* \mid w = 1^m \circ 0 \text{ for } m \in \mathbb{N}, m \geq 0\} \\
 &= \{0, 10, 110, 1110, \dots\} \\
 &= 1^*0.
 \end{aligned}$$

(2)  $A = \{0, 1\}$

$$\begin{aligned} A^* \circ 1 \circ A^* &= \{w \in A^* \mid w \text{ contains at least one } 1\} \\ &= \{u \circ 1 \circ v \mid u, v \in A^*\} \\ &= A^* 1 A^* \end{aligned}$$

### The Pumping Lemma

If A is a Regular Language, then A has a Pumping Length 'P' such that any string 'S' where  $|S| \geq P$  may be divided into 3 parts  $S = xyz$  such that the following conditions are true

1.  $x y^i z \in A$  for every  $i \geq 0$
2.  $|y| \geq 0$
3.  $|xy| < P$

### To prove that a language is NOT REGULAR using the Pumping Lemma:

1. Assume that A is regular
2. A must then have a Pumping Length P
3. All strings longer than P can be pumped  $|S| \geq P$
4. Now find a string 'S' in A such that  $|S| \geq P$
5. Divide S into x y and z
6. Show that  $x y^i z \notin A$  for some i
7. Then consider all ways that S can be divided into x y and z
8. Show that none of these satisfy all 3 pumping conditions at the same time
9. S cannot be Pumped == CONTRADICTION

### The Pumping Lemma – Example 1

Using the Pumping Lemma prove that the language  $A = \{a^n b^n \mid n > 0\}$  is not regular.

#### **Proof:**

Assume A is a regular language, therefore it has a pumping length p.

$$S = a^p b^p$$

Divide S into x y and z

Let the Pumping Length = 7

Therefore  $S = aaaaaa bbbbbb$

Explore the different possibilities of breaking up S...

**Case 1 (The y is in the a part):**

$x \quad y \quad z$   
[aa][aaaa][abbbbbbb]

**Case 2 (The y is in the b part):**

$x \quad y \quad z$   
[aaaaaabb][bbbb][b]

**Case 3 (The y is in the a and the b part):**

$x \quad y \quad z$   
[aaaa][aabb][bbbb]

Show that  $x y^i z \notin A$  for some  $i$   
Let  $i = 2$ , therefore test  $S = x y^2 z$

$x \quad y^2 \quad z$   
**Case 1:**  $S = [aa][aaaaaaa][abbbbbbb]$

This  $S \notin A$  since there are a total of 11 a's and 7 b's which does not satisfy  $A = \{ a^n b^n \}$

$x \quad y^2 \quad z$   
**Case 2:**  $S = [aaaaaabb][bbbbbbbb][b]$

This  $S \notin A$  since there are a total of 7 a's and 11 b's which does not satisfy  $A = \{ a^n b^n \}$

$x \quad y^2 \quad z$   
**Case 3:**  $S = [aaaa][aabbaabb][bbbb]$

This  $S \notin A$  since it does not follow the pattern as described under  $A = \{ a^n b^n \}$

**\*\*REVERT BACK TO CONDITIONS 1-3 AND SHOW HOW THEY ARE NOT PROVED\*\***

Condition 1 proven false above.

Condition 2 is true.

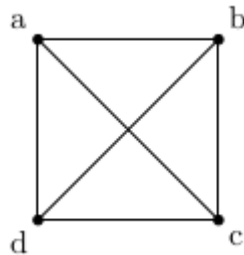
Condition 3 -  $|xy| < P$  – is false in all of the above cases.

Therefore using the Pumping Lemma it is not possible for all strings under L to satisfy the conditions of the Pumping Lemma, therefore the language L is not regular.

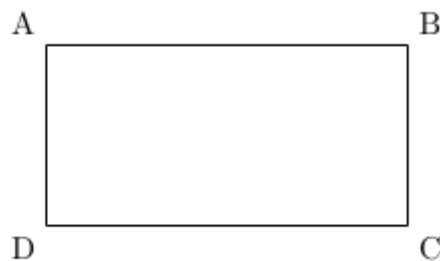
---

## **Graph Theory**

**Complete Graph:** A complete graph is a graph containing the highest number of edges possible for its given set of vertices.



**Regular Graph:** A regular graph is a graph where every vertex has the same degree. A graph is considered to be k-regular if every vertex has a degree of k.



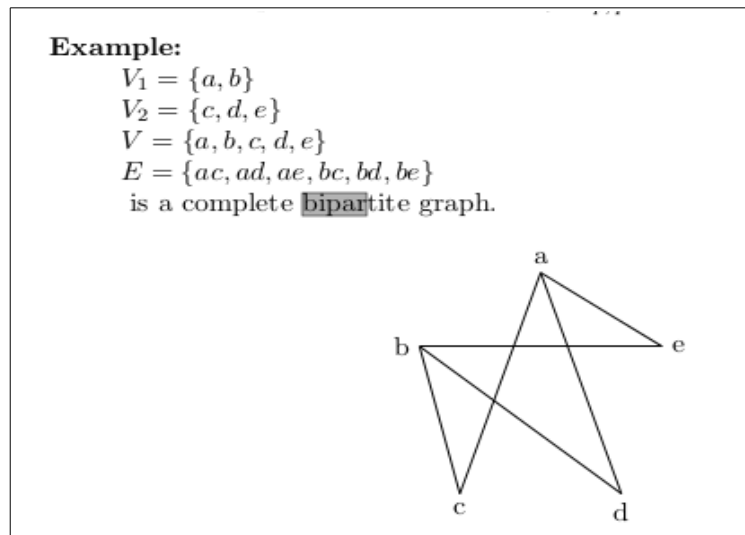
**Incidence Table:** A graph's incidence table is the table of Edges vs Vertices marking incidence.

	ac	ad	be	bd	ce
a	1	1	0	0	0
b	0	0	1	1	0
c	1	0	0	0	1
d	0	1	0	1	0
e	0	0	1	0	1

**Adjacency Table:** A graph's adjacency table is the table of Edges vs Vertices marking adjacency.

	a	b	c	d	e
a	0	1	1	0	1
b	1	0	0	1	1
c	1	0	0	1	0
d	0	1	1	0	0
e	1	1	0	0	0

**Bipartite Graph:** A bipartite graph is a graph that contains two separate unique subsets of vertices that when taking every edge into account, every edge will be of the form  $E = vw$  where  $v$  is from subset one and  $w$  is from subset 2.



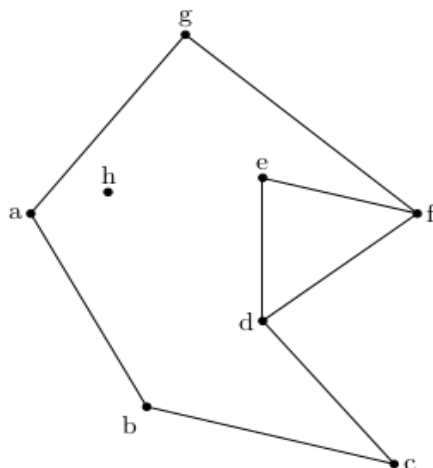
**Isomorphism:** An isomorphism between two graphs  $(V, E)$  and  $(V', E')$  is a bi-jective function  $\phi : V \rightarrow V'$  satisfying that  $\forall a, b \in V$  with  $a \neq b$  the edge  $ab \in E \Leftrightarrow$  the edge  $\phi(a)\phi(b) \in E'$

**Walk:** A walk in a graph is a defined route that traverses a given set of edges passing through a given set of vertices.

**Trail:** A trail in a graph is a walk that traverses **edges** at most once.

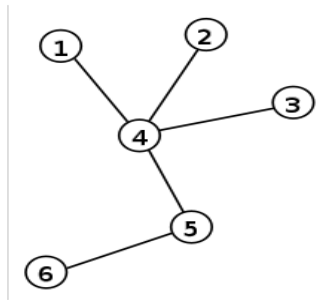
**Path:** A path in a graph is a walk that passes through **vertices** at most once.

1.  $h$  is a trivial walk/trail/path
2.  $defd$  is a trail, but not a path because we pass through the vertex  $d$  twice.
3.  $def$  is a path
4.  $gfdefdc$  is a walk but not a trail or a path





**Connected Graph:** A connected graph is a graph where there exists a path in the graph from every u to every v.



**Circuit:** A circuit is a nontrivial closed trail i.e. a closed walk with no repeated edges passing through at least two vertices.

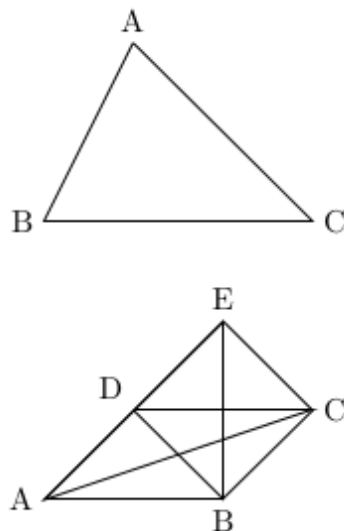
**Eulerian Trail:** An Eulerian trail in a graph is a trail that traverses every edge of that graph. In other words, an Eulerian trail is a walk that traverses every **edge** of the graph **exactly once**.

Trail  $\Rightarrow$  an **edge** is traversed **at most once**.

Eulerian  $\Rightarrow$  every **edge** is traversed.

**Examples:**

1.  $ABCA$  is an Eulerian circuit. The triangle is  $K_3$ .
2. Consider  $K_5$ , the complete graph with 5 vertices.  
 $EABECDBCADE$  is an Eulerian circuit.



In both cases, the degree of the vertices is even for all vertices. We'll see this property is important and derive other necessary and sufficient conditions for the existence of Eulerian trails and circuits.

**Hamiltonian Path:** A Hamiltonian path in a graph is a path that passed **exactly once** through every **vertex** of a graph.

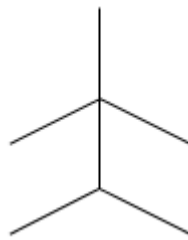
Path  $\Rightarrow$  we pass through a **vertex at most once** (no repeated vertices)

Hamiltonian  $\Rightarrow$  we pass through every **vertex**.

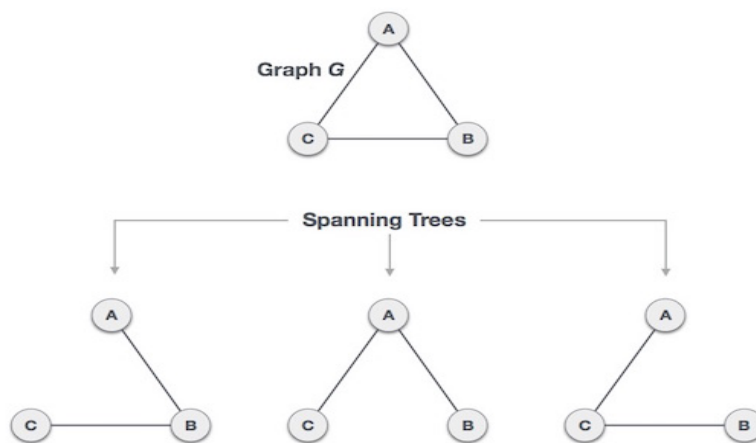
**Hamiltonian Circuit:** A Hamiltonian circuit in a graph is a simple **circuit** that passes through every vertex of the graph.

**A Tree:** A tree is a graph defined by the formula:

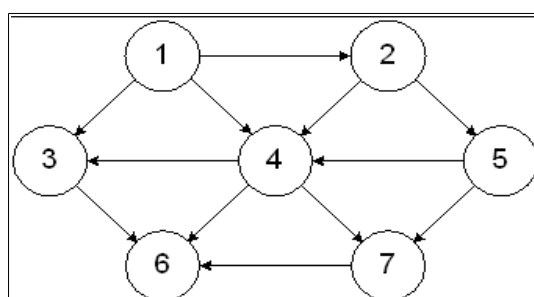
$$\#E = \#V - 1$$



**Spanning Tree:** A spanning tree is a subset of the graph G which has all of the vertices covered with a minimum number of edges.



**Directed Graphs:** A directed graph is a graph  $(V, E)$  where the an edge  $E = vw$  is not the same as the edge  $D = wv$ . In other words the order in which an edge is listed depicts its direction.



## Graph Theory - Algorithms

Algorithms for detecting the minimal spanning tree within a connected graph are as follows:

**Kruskals** → Edges in ordered queue.

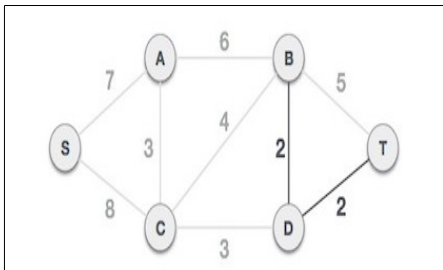
**Prims** → Repeated adjacency from a given vertice.

- Kruskals Algorithm:**

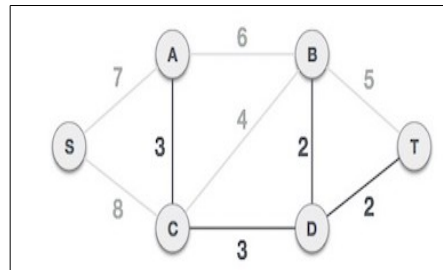
1. Arrange edges in increasing order of weight.
2. Add edge with the lowest weight.
3. Ensure that this **does not create a circuit**
4. If it does, skip, else continue

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

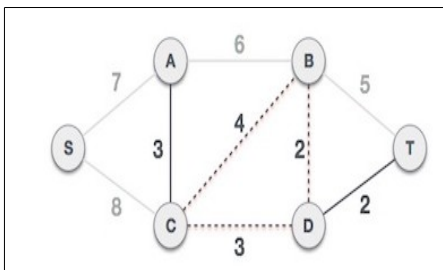
1) Add edge BD and DT



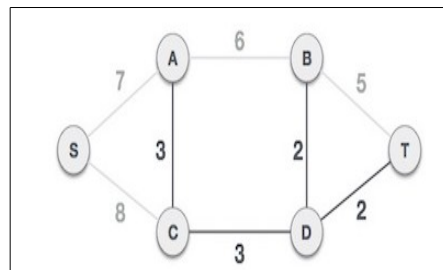
2) Add edge AC



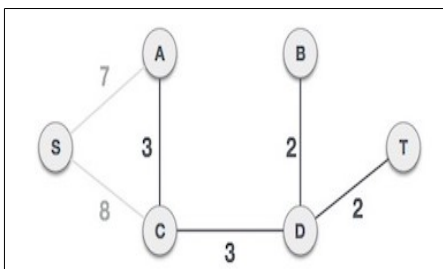
3) Add edge CB (creates circuit)



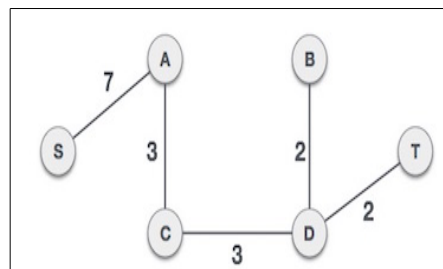
4) Ignore CB since circuit



5) Ignore edge BT and AB (circuit)



6) Add edge AS (FINISHED)

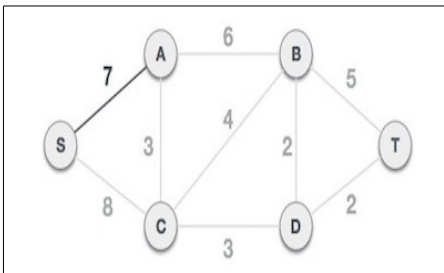


- Primms Algorithm:

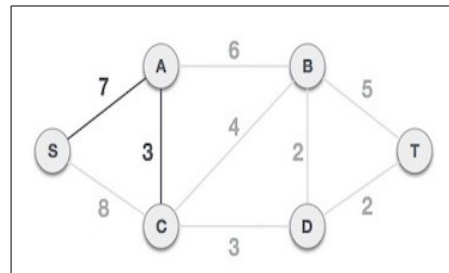
1. Choose any arbitrary vertice S as root.
2. Check adjacent edges to S and add least cost (e.g B).
3. Check edges adjacent to both S **and** B, choose least cost.
4. Continue until all vertices added.

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

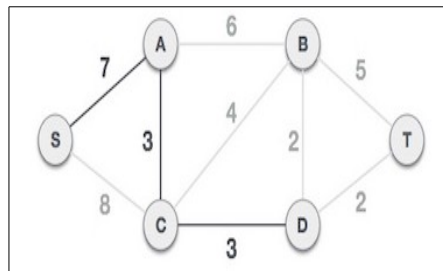
1) Choose S as root node, add SA



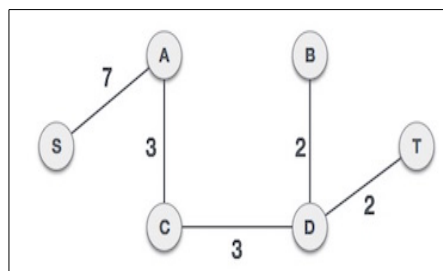
2) Check adjacent edges to S and A, choose AC



3) Check adjacent edges to S,A and C choose CD



4) Check adjacent edges to S,A,C and D choose BD and DT (both 2)



## Countability of Sets

Since a set  $A$  can be denoted as  $f: A \rightarrow \{1, 2, \dots, n\}$

Let  $J_n = \{1, 2, \dots, n\}$

**Countably Infinite:** A set  $A$  is countably infinite if  $A \sim J$  (bijective mapping).

**Uncountably:** A set  $A$  is uncountably infinite if  $A$  is not finite or countably infinite.

- A set  $A$  is countably infinite if its elements can be arranged in a sequence  $\{x_1, x_2, \dots\}$ . This is another way of saying  $A$  is in bijective correspondence with  $J$ , i.e.

$\exists f: A \rightarrow J$  a bijection, namely  $A \sim J$

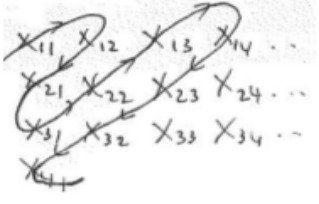
### Examples

- $\pi \simeq 3.1415\dots$  i.e. instead of  $\pi$  we can work with the following sequence of rational numbers:  $x_1 = 3, x_2 = 3.1, x_3 = 3.14, x_4 = 3.141, x_5 = 3.1415, \dots \lim_{n \rightarrow \infty} x_n = \pi$ .  $\pi$  is irrational, i.e.,  $\pi \in \mathbb{R} \setminus \mathbb{Q}$ .
- $\frac{1}{3} \simeq 0.333\dots$  means we can set up the sequence of rational numbers  $x_1 = 0, x_2 = 0.3, x_3 = 0.33, x_4 = 0.333, x_5 = 0.3333$  etc. such that  $\lim_{n \rightarrow \infty} x_n = \frac{1}{3}$ . Note that  $\frac{1}{3} \in \mathbb{Q}$ .

- Note:** Every subset of a countably infinite set is itself a countably infinite set

**Notation:** A sequence  $\{x_1, x_2, \dots\}$  can be denoted as  $\{x_i\}_{i=1,2,\dots}$ .

- Let  $\{A_n\}_{n=1,2,\dots}$  be a sequence of countably infinite sets. Let  $S = \{A_1 \cup A_2 \cup \dots \cup A_{\text{infinity}}\}$ . Then  $S$  is countably infinite.



$\{x_{11}, x_{12}, x_{21}, x_{31}, x_{22}, x_{13}, x_{14}, x_{23}, x_{32}, x_{41}, \dots\}$

$= \{A_1 \cup A_2 \cup \dots \cup A_n\} = S$  is countably infinite because even if some  $x_i$ 's are the same.  
 $A_n \subseteq S \forall n \geq 1$  and  $A_n \sim J$ .

## Countability of Sets to Formal Languages

- Theorem 1:** If  $A$  is a finite alphabet then the set of all words over  $A$  ( $A^*$ ) is countably infinite.
- Theorem 2:** If  $A$  is a finite alphabet then the set of all languages over  $A$  ( $A^*$ ) is countably infinite.

*Recall: A language over a finite alphabet is regular  $\iff$  it is given by a regular expression.*

- **Theorem 3:** The set of all regular languages over a finite alphabet A is countably infinite.
- 

## **Turing Machines**

A Turing machine is similar to a finite state acceptor but it has **unlimited memory** given by an infinite tape (countably infinite). The tape is divided into **cells** each of which contains a character of a **tape alphabet**.

The Turing machine is equipped with a **tape head** that can read and write symbols on the tape, and move left (back) or right (forward) on the tape. Initially, the tape contains only the input string and is blank everywhere else.

To store information, the Turing machine can write this information on the tape. To read information, the Turing machine can move its head back over it.

The Turing machine continues computing until it decides to produce an output. The outputs “**accepts**” and “**rejects**” are obtained by entering accepting or rejecting states. It is also possible for the Turing machine to go on forever if it does not enter either an accepting nor rejecting state.

[ 0 ][ 1 ][ 0 ][ \_ ][ \_ ][ ...

The **blank symbol** ( \_ ) is part of the tape alphabet.

### Example 1

Let  $A = \{ 0, 1 \}$  and  $L = \{ 0^m 1^m \mid m \in \mathbb{N}, m \geq 1 \}$

We know L is **not a regular language**, so there is no finite state acceptor that can recognise it, but there is a Turing machine that can.

**Initial State of the Tape:** Input string of 0s and 1s, then infinitely many blanks

**Machine Method:** Change a 0 to an X, and a 1 to a Y until either:

- a) All 0s and 1s have been matched – ACCEPT
- b) The 0s and 1s do not match or the string does not have the form  $0^*1^*$  - REJECT

**Machine Algorithm:** The tape head is initially positioned over the first cell.

1. If anything other than 0 in the first cell – REJECT
2. Is 0 in the cell, then change 0 to X
3. Move right to the first 1. If none – REJECT
4. Change 1 to Y
5. Move left to leftmost 0. If none, move right looking for either a 0 or a 1. If either a 0 or 1 is found before the first blank symbol – REJECT. Otherwise – ACCEPT.
6. Go to step 2.

Examples of processing strings: (Tape Head)

0011_ X011_ X011_ X0Y1_ X0Y1_ XXY1_ XXY1_ XXYY_ XXYY_ XXYY_ ACCEPT (STEP 5)	001_ X01_ X01_ X0Y_ X0Y_ XXY_ XXY_ REJECT (STEP 3)	011_ X11_ X11_ XY1_ XY1_ XY1_ REJECT (STEP 5)	010_ X10_ X10_ XY0_ XY0_ XY0_ REJECT (STEP 5)
---	---	---	---

**Definition:** A Turing machine is a 7-tuple  $(S, A, \tilde{A}, t, i, S_{\text{accept}}, S_{\text{reject}})$ .

- $S$  = Set of States
- $A$  = Input Alphabet (*without the blank symbol*)
- $\tilde{A}$  = Tape Alphabet
- $t$  = Transition Mapping
- $i$  = Initial State
- $S_{\text{accept}}$  = Accept State
- $S_{\text{reject}}$  = Reject State

Recall that a finite state acceptor is given by  $(S, A, i, t, F)$  where:

- $S$  = Set of States
- $A$  = Alphabet
- $i$  = Initial State
- $t$  = Transition Mapping
- $F$  = State of Finishing States

The transition mapping is given by  $t : S \times A \rightarrow S$

By contrast, a Turing machine's transition mapping is of the form  $t : S \times \tilde{A} \rightarrow S \times \tilde{A} \times \{L, R\}$

- $\tilde{A}$  = Indicates what the Turing machine can write
- $\{L, R\}$  = Indicates the Turing machine can move left or right

**Configurations:** We represent a configuration as  $U, S_i, V_i$  where  $U, V$  are strings in the tape alphabet and  $S_i$  is the current state of the machine.

e.g  $\epsilon i 001$  is the configuration  $[0][0][1][\_][\dots]$

There are several types of configurations:

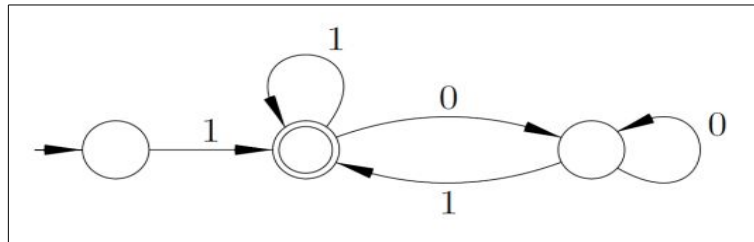
1. **Initial Configuration:** Starting configuration.
2. **Accepting Configuration:** Configuration of the machine in an accepting state.
3. **Rejecting Configuration:** Configuration of the machine in a rejecting state.
4. **Halting Configuration:** Yields no further configurations.

A Turing Machine  $M$  accepts input  $w \in A^*$  (string over the input alphabet  $A$ ) if  $\exists$  a sequence of configurations  $C_1, C_2, \dots, C_k$  such that:

1.  $C_1$  is the start configuration with input  $w$
2. Each  $C_i$  yields  $C_{i+1}$
3.  $C_k$  is an accepting state.

(a) Let  $L$  be the language consisting of the binary representations of all odd natural numbers. Write down the algorithm of a Turing machine that recognizes  $L$ . Process the following strings according to your algorithm: empty string, 0, 11, and 100.

Binary representations of odd natural numbers can be described under the following expression  $1 \cup 1(0 \cup 1)^*1$ . An example of a finite state acceptor that accepts binary representations of odd natural numbers can be found below. This representation does not account for leading zeros.



This can be described in the form of a Turing Machine M which performs the following algorithm:

1. If anything other than 1 is in the first cell **REJECT**.
2. Move to the end of the string (the next `_` character), then **move left**.
3. If 1 is in the current cell **ACCEPT**. Else **REJECT**.

<u>Processing Empty Word</u>	<u>Processing '0'</u>	<u>Processing '11'</u>	<u>Processing '100'</u>
<p>↓</p> <p>[ _ ] <b>REJECT</b> - Step 1</p>	<p>↓</p> <p>[ 0 ] <b>REJECT</b> - Step 1</p>	<p>↓</p> <p>[ 1 ] [ 1 ] [ _ ] Step 1</p> <p>↓</p> <p>[ 1 ] [ 1 ] [ _ ] Step 2</p> <p>↓</p> <p>[ 1 ] [ 1 ] [ _ ] Step 2</p> <p>↓</p> <p>[ 1 ] [ 1 ] [ _ ] <b>ACCEPT</b> Step 3</p>	<p>↓</p> <p>[ 1 ] [ 0 ] [ 0 ] [ _ ] Step 1</p> <p>↓</p> <p>[ 1 ] [ 0 ] [ 0 ] [ _ ] Step 2</p> <p>↓</p> <p>[ 1 ] [ 0 ] [ 0 ] [ _ ] Step 2</p> <p>↓</p> <p>[ 1 ] [ 0 ] [ 0 ] [ _ ] <b>REJECT</b> Step 3</p>



(b) Write down the transition diagram of the Turing machine from part (a) carefully labelling the initial state, the accept state, the reject state, and all the transitions specified in your algorithm.

The Turing Machine above can be described under the 7-tuple  $(S, A, \tilde{A}, t, i, S_{\text{accept}}, S_{\text{reject}})$ :

$$\underline{S} = \{S_1, S_2\}$$

$$\underline{A} = \{0, 1\}$$

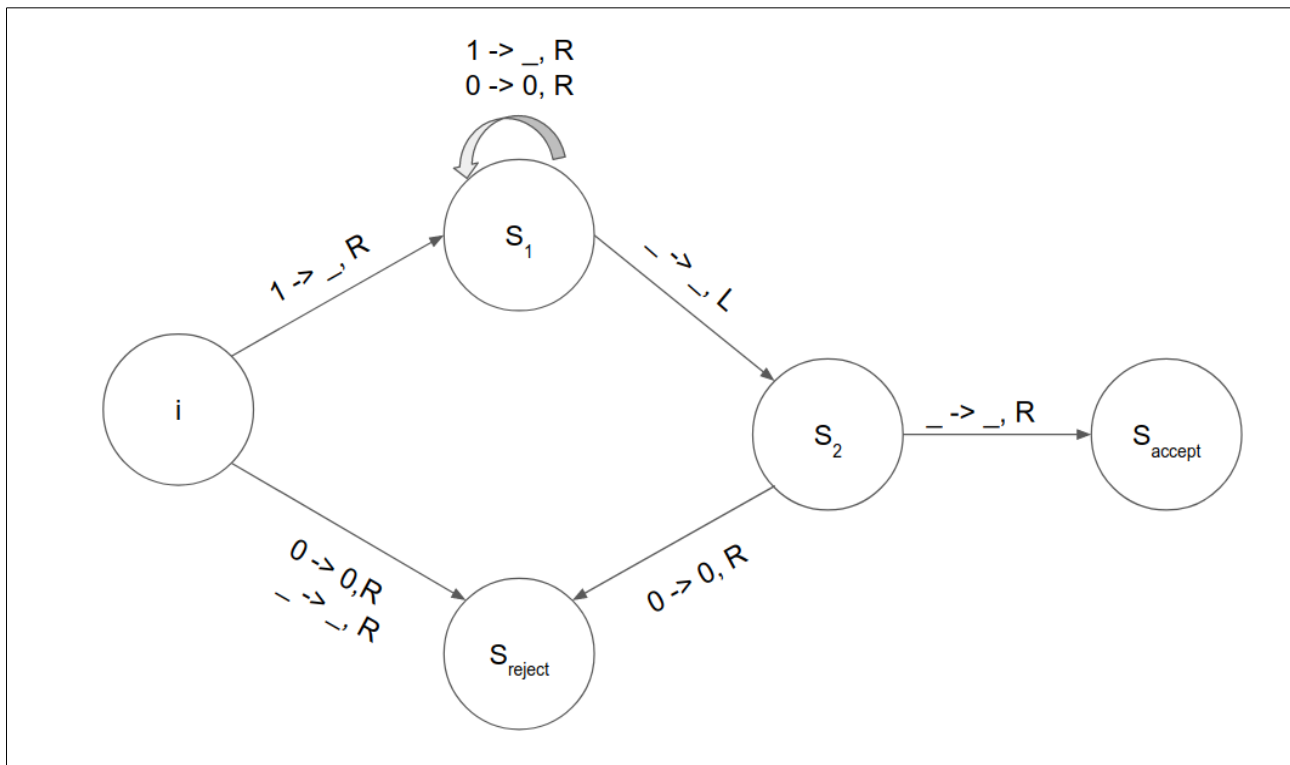
$$\underline{\tilde{A}} = \{\_ \}$$

$$\underline{t} = t : S \times \tilde{A} \rightarrow S \times \tilde{A} \times \{L, R\}$$

$$\underline{I} = S_i$$

$$\underline{S_{\text{accept}}} = S_{\text{accept}}$$

$$\underline{S_{\text{reject}}} = S_{\text{reject}}$$



(b) Write down the algorithm of an enumerator that prints out EXACTLY ONCE every odd natural number divisible by 5. (Hint: Order the words on the input tape.)

-The Turing Machine as described above needs to be somewhat changed as more states needed to be added. The new and updated Turing Machine M is described under the following model:

$$\underline{S} = \{S_1, S_2, S_3, S_4, S_5\}$$

$$\underline{A} = \{0, 1\}$$

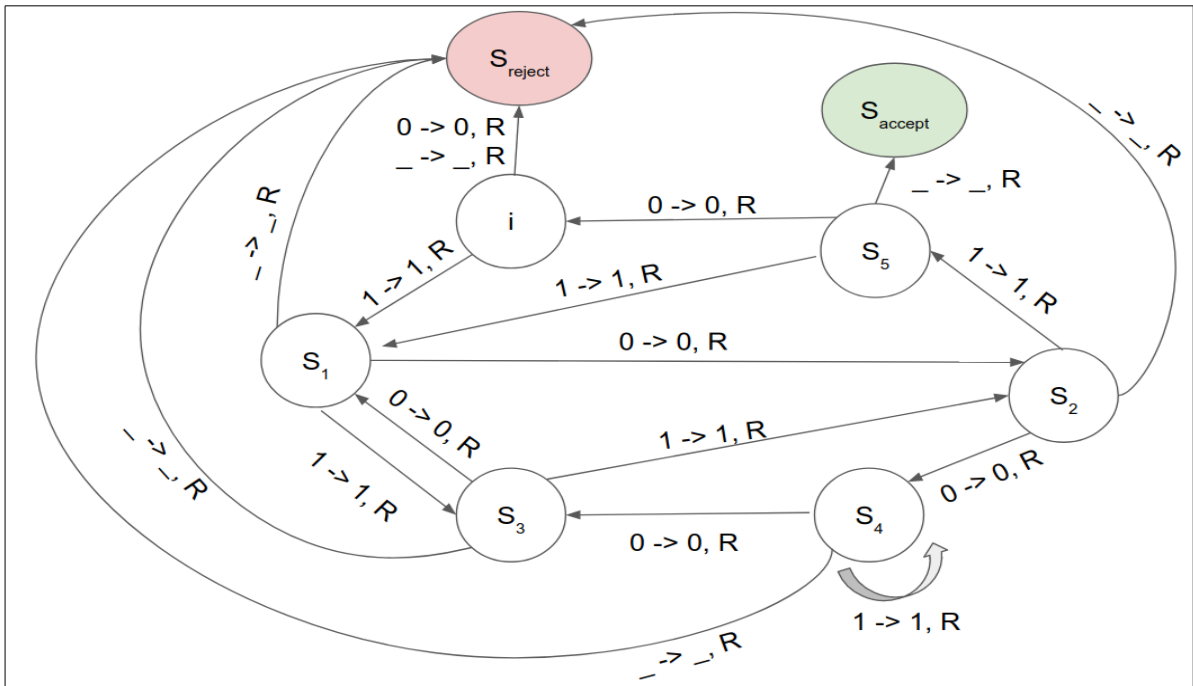
$$\underline{\tilde{A}} = \{\_ \}$$

$$\underline{t} = t : \underline{S} \times \underline{\tilde{A}} \rightarrow \underline{S} \times \underline{\tilde{A}} \times \{L, R\}$$

$$\underline{I} = S_i$$

$$\underline{S}_{\text{accept}} = S_{\text{accept}}$$

$$\underline{S}_{\text{reject}} = S_{\text{reject}}$$



### Algorithm of Enumerator:

Let L be the language  $L = 1 \cup 1(0 \cup 1)^* 1$ . Let M be the above Turing Machine that recognizes all words under L that are binary representations of odd numbers divisible by 5. We construct an enumerator E that outputs these words.

Let A be the alphabet of L  $\{0, 1\}$ .  $A^*$  has an enumeration as a sequence  $A^* = \{w_1, w_2, \dots\}$ .

1. Repeat the following for  $k = 1, 2, 3, \dots$
2. Run M, passing it the input word  $w_k$
3. If any computations accept, print out the corresponding  $w_k$ .

Every string accepted by M (every binary representation of every odd natural number) will appear on the list of E.