Final Term CS-502

By:- MUDASSAR Iqbal
0340-6829977

بسم اللہ الرحمن الرحیم

Recite Darood Pak!

Chapter 7

Greedy Algorithms

ارتقا، بہتری، ترقی / improvement of efficiency.

An *optimization problem* is one in which you want to find, not just a solution, but the best solution. Search techniques look at many possible solutions. E.g. dynamic programming or backtrack search. A "greedy algorithm" sometimes works well for optimization problems

A greedy algorithm works in phases. At each phase:

اگر آپ نے ایک Node کو choose کیا تو وہ Best تھی اور آگے جا کر آپ کو پتہ چلا کہ وہ Best نہیں تھی تو

Back Track search

- You take the best you can get right now, without regard for future consequences.
- You hope that by choosing a local optimum at each step, you will end up at a global optimum.

For some problems, greedy approach always gets optimum. For others, greedy finds good, but not always best. If so, it is called a greedy heuristic, or approximation. For still others, greedy approach can do very poorly.

7.1 Example: Counting Money

Suppose you want to count out a certain amount of money, using the fewest possible bills (notes) and coins. A greedy algorithm to do this would be: at each step, take the largest possible note or coin that does not overshoot.

```
while (N > 0){
    give largest denomination coin ≤ N
    reduce N by value of that coin
}
```

N → Number of rupees:-

Good

Consider the currency in U.S.A. There are paper notes for one dollar, five dollars, ten dollars, twenty dollars, fifty dollars and hundred dollars. The notes are also called "bills". The coins are one cent, five cents (called a "nickle"), ten cents (called a "dime") and twenty five cents (a "quarter"). In Pakistan, the currency notes are five rupees, ten rupees, fifty rupees, hundred rupees, five hundred rupees and thousand

If count small then optimal solution.
If count large then then not optimal solution / just solution

rupees. The coins are one rupee and two rupees. Suppose you are asked to give change of \$6.39 (six dollars and thirty nine cents), you can choose:

- a \$5 note
- a \$1 note to make \$6
- a 25 cents coin (quarter), to make \$6.25
- a 10 cents coin (dime), to make \$6.35
- four 1 cents coins, to make \$6.39

greedy solution. but not optimal.

optimal solution is best solution.

Notice how we started with the highest note, \$5, before moving to the next lower denomination.

Formally, the Coin Change problem is: Given k denominations $d_1, d_2, \ldots, d_k$ and given N, find a way of writing

$$\boxed{N = i_1 d_1 + i_2 d_2 + \cdots + i_k d_k}$$

such that

$$i_1 + i_2 + \cdots + i_k \text{ is minimized.}$$

The "size" of problem is k.

The greedy strategy works for the coin change problem but not always. Here is an example where it fails. Suppose, in some (fictional) monetary system, "krons" come in 1 kron, 7 kron, and 10 kron coins Using a greedy algorithm to count out 15 krons, you would get A 10 kron piece Five 1 kron pieces, for a total of 15 krons This requires six coins. A better solution, however, would be to use two 7 kron pieces and one 1 kron piece This only requires three coins The greedy algorithm results in a solution, but not in an optimal solution

The greedy approach gives us an optimal solution when the coins are all powers of a fixed denomination.

$$N = i_0 D^0 + i_1 D^1 + i_2 D^2 + \cdots + i_k D^k$$

Note that this is N represented in based D. U.S.A coins are multiples of 5: 5 cents, 10 cents and 25 cents.

7.1.1 Making Change: Dynamic Programming Solution

Not imp

The general coin change problem can be solved using Dynamic Programming. Set up a Table, $C[1..k, 0..N]$ in which the rows denote available denominations, d_i; $(1 \le i \le k)$ and columns denote the amount from $0 \ldots N$ units, $(0 \le j \le N)$. $C[i, j]$ denotes the minimum number of coins, required to pay an amount j using only coins of denominations 1 to i. $C[k, N]$ is the solution required.

To pay an amount j units, using coins of denominations 1 to i, we have two choices:

1. either chose NOT to use any coins of denomination i,
2. or chose at least one coin of denomination i, and also pay the amount $(j - d_i)$.

To pay $(j - d_i)$ units it takes $C[i, j - d_i]$ coins. Thus,

$$C[i, j] = 1 + C[i, j - d_i]$$

Since we want to minimize the number of coins used,

$$C[i, j] = \min(C[i-1, j], 1 + C[i, j - d_i])$$

Here is the dynamic programming based algorithm for the coin change problem.

```
COINS(N)
d[1..n] = {1, 4, 6}   // (coinage, for example)
for i = 1 to k
do c[i, 0] ← 0
for i = 1 to k
do for j = 1 to N
    do if (i = 1 & j < d[i])
        then c[i, j] ← ∞
        else if (i = 1)
            then c[i, j] ← 1 + c[1, j − d[1]]
            else if (j < d[i])
                then c[i, j] ← c[i − 1, j]
                else c[i, j] ← min (c[i − 1, j], 1 + c[i, j − d[i]])
return c[k, N]
```

pseudo code

7.1.2 Complexity of Coin Change Algorithm

MCQ's

omega

Greedy algorithm (non-optimal) takes $O(k)$ time. Dynamic Programming takes $O(kN)$ time. Note that N can be as large as 2^k so the dynamic programming algorithm is really exponential in k. Lect 23

Lec #24

7.2 Greedy Algorithm: Huffman Encoding

The Huffman codes provide a method of encoding data efficiently. Normally, when characters are coded using standard codes like ASCII. Each character is represented by a fixed-length codeword of bits, e.g., 8 bits per character. Fixed-length codes are popular because it is very easy to break up a string into its individual characters, and to access individual characters and substrings by direct indexing. However, fixed-length codes may not be he most efficient from the perspective of minimizing the total quantity of data.

ASCII → 8 bit code

Consider the string " abacdaacac". if the string is coded with ASCII codes, the message length would be $10 \times 8 = 80$ bits. We will see shortly that the same string encoded with a variable length Huffman encoding scheme will produce a shorter message.

ایک Message کو دوسری جگہ پہنچانے سے پہلے اس message کو compress کر دیتا ہے

یعنی Number of Bits کو کم کر دیتا ہے

جیسا کہ آپ جانتے ہیں کہ کپڑے ایک Bag میں ڈالتے ہیں تو جتنے بھی وہ Bag میں ہوتے ہیں وہ Compress ہو جاتے ہیں۔ اور آپ اگلی جگہ پہ ڈالیں گے اس کو 100 والے ہی حاصل کرتے ہیں۔

7.2.1 Huffman Encoding Algorithm

Here is how the Huffman encoding algorithm works. Given a message string, determine the frequency of occurrence (relative probability) of each character in the message. This can be done by parsing the Freq. message and counting how many time each character (or symbol) appears. The probability is the number of occurrence of a character divided by the total characters in the message. The frequencies and probabilities for the example string " abacdaacac" are

By: Mudassar Iqbal 0340-6829977.

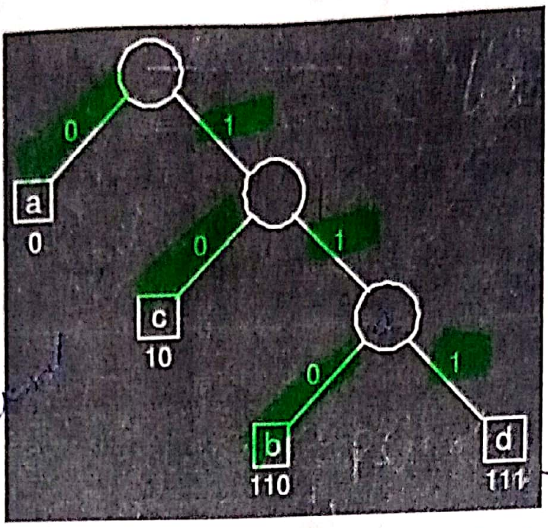
character	a	b	c	d
frequency	5	1	3	1
probability	0.5	0.1	0.3	0.1

Next, create binary tree (leaf) node for each symbol (character) that occurs with nonzero frequency Set node weight equal to the frequency of the symbol. Now comes the greedy part: Find two nodes with smallest frequency. Create a new node with these two nodes as children, and with weight equal to the sum of the weights of the two children. Continue until we have a single tree.

Finding two nodes with the smallest frequency can be done efficiently by placing the nodes in a heap-based priority queue. The min-heap is maintained using the frequencies. When a new node is created by combining two nodes, the new node is placed in the priority queue. Here is the Huffman tree building algorithm.

```
HUFFMAN(N, symbol[1..N], freq[1..N])
  for i = 1 to N
  do t ← TreeNode(symbol[i], freq[i])
     pq.insert(t, freq[i]) // priority queue
  for i = 1 to N − 1
  do x = pq.remove(); y = pq.remove()
     z ← new TreeNode
     z.left ← x; z.right ← y
     z.freq ← x.freq + y.freq
     pq.insert(z, z.freq);
  return pq.remove(); // root
```

Figure 7.1 shows the tree built for the example message "abacdaacac"



left sid → 0
Right sid → 1

[illegible] Frequency [illegible] code word [illegible]

Figure 7.1: Huffman binary tree for the string "abacdaacac"

Prefix Property:

The codewords assigned to characters by the Huffman algorithm have the property that no codeword is a prefix of any other:

character	a	b	c	d
frequency	5	1	3	1
probability	0.5	0.1	0.3	0.1
codeword	0	110	10	111

The prefix property is evident by the fact that codewords are leaves of the binary tree. Decoding a prefix code is simple. We traverse the root to the leaf letting the input 0 or 1 tell us which branch to take.

Expected encoding length:

If a string of n characters over the alphabet $C = \{a, b, c, d\}$ is encoded using 8-bit ASCII, the length of encoded string is $8n$. For example, the string "abacdaacac" will require $8 \times 10 = 80$ bits. The same string encoded with Huffman codes will yield

a	b	a	c	d	a	a	c	a	c
0	110	0	10	111	0	0	10	0	10

This is just 17 bits, a significant saving!. For a string of n characters over this alphabet, the expected encoded string length is

$$n(0.5 \cdot 1 + 0.1 \cdot 3 + 0.3 \cdot 2 + 0.1 \cdot 3) = 1.7n$$

In general, let $p(x)$ be the probability of occurrence of a character, and let $d_T(x)$ denote the length of the codeword relative to some prefix tree T. The expected number of bits needed to encode a text with n characters is given by

$$\boxed{B(T) = \overset{\text{No. of bits.}}{n} \sum_{x \in C} p(x) d_T(x)}$$

Bit-Tree. depth of Tree.

Lecture [illegible] Register [illegible]

Lect #25

7.2.2 Huffman Encoding: Correctness

Huffman algorithm uses a greedy approach to generate a prefix code T that minimizes the expected length B(T) of the encoded string. In other words, Huffman algorithm generates an optimum prefix code. The question that remains is that *why is the algorithm correct?*

Recall that the cost of any encoding tree T is

$$B(T) = n \sum_{x \in C} p(x) d_T(x)$$

Our approach to prove the correctness of Huffman Encoding will be to show that any tree that differs from the one constructed by Huffman algorithm can be converted into one that is equal to Huffman's tree without increasing its costs. Note that the binary tree constructed by Huffman algorithm is a full binary tree.

Claim:

Node

Consider two characters $\boxed{x \text{ and } y}$ with the smallest probabilities. Then there is optimal code tree in which these two characters are siblings at the maximum depth in the tree.

بھائی سب سے نچلی سطح

Proof:

Let T be any optimal prefix code tree with two siblings b and c at the maximum depth of the tree. Such a tree is shown in Figure 7.2Assume without loss of generality that

$$p(b) \le p(c) \quad \text{and} \quad p(x) \le p(y)$$

Probability less so it is written to right

Doe

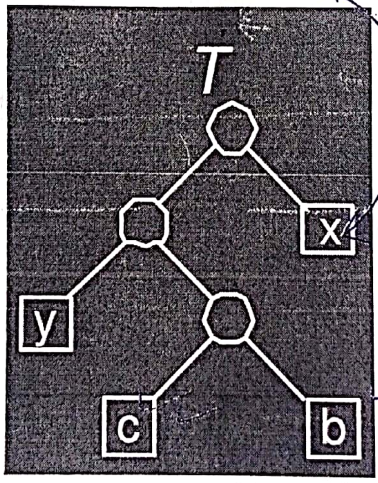


Figure 7.2: Optimal prefix code tree T

Small will change to Small/or left to left

اس میں x اور y علیحدہ علیحدہ ہیں۔ ہم نے ان کو sibling بنانا ہے۔ مطلب ان کا Parent ایک ہی ہو، [illegible]

Since x and y have the two smallest probabilities (we claimed this), it follows that

$$\boxed{p(x) \le p(b)} \quad \text{and} \quad \boxed{p(y) \le p(c)}$$

(Right to Right)

Since b and c are at the deepest level of the tree, we know that

$$d(b) \geq d(x) \quad \text{and} \quad d(c) \geq d(y) \quad \text{(d is the depth)}$$

Thus we have

$$p(b) - p(x) \geq 0$$

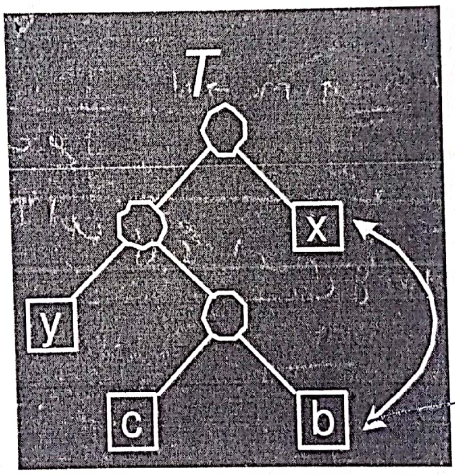
and

$$d(b) - d(x) \geq 0$$

Hence their product is non-negative. That is,

$$(p(b) - p(x)) \cdot (d(b) - d(x)) \geq 0$$

Now swap the positions of x and b in the tree



ٹری میں x کو نیچے لا
سکتے ہیں.

P کی Probability زیادہ ہو ←
تو ٹری میں اس کو
اوپر رکھیں.

Figure 7.3: Swap x and b in tree prefix tree T

This results in a new tree T′

کیونکہ Node برابر ہونے سے اس کی Probability کم نہیں ہوگی

depth والے نیچے کم Probability
ہمیشہ نیچے ہیں
زیادہ Probability والے اوپر ہیں

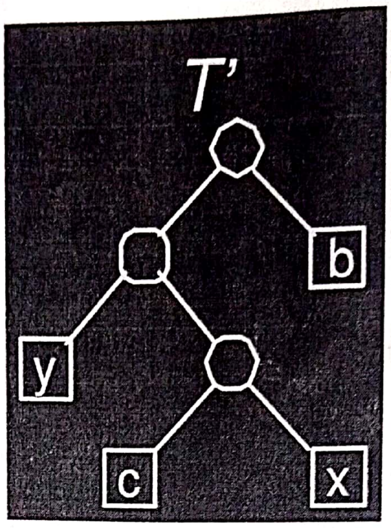


Figure 7.4: Prefix tree T′ after x and b are swapped

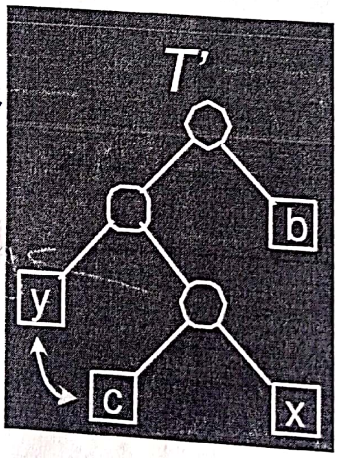
Let's see how the cost changes. The cost of T′ is

$$\begin{aligned} \mathrm{B}(\mathrm{T}') &= \mathrm{B}(\mathrm{T}) - p(x)d(x) + p(x)d(b) - p(b)d(b) + p(b)d(x) \\ &= \mathrm{B}(\mathrm{T}) + p(x)[d(b) - d(x)] - p(b)[d(b) - d(x)] \\ &= \mathrm{B}(\mathrm{T}) - (p(b) - p(x))(d(b) - d(x)) \\ &\leq \mathrm{B}(\mathrm{T}) \quad \text{because} \quad (p(b) - p(x))(d(b) - d(x)) \geq 0 \end{aligned}$$

Thus the cost does not increase, implying that T′ is an optimal tree.

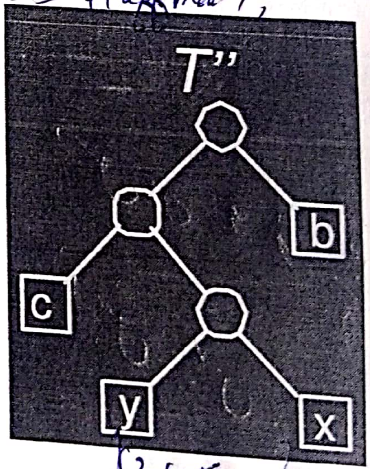
By switching y with c we get the tree T″. Using a similar argument, we can show that T″ is also optimal.

اس میں y کو نیچے لے کر گئے ہیں۔



⇒

یہ Huffman کی ٹری ہے۔



The final tree T″ satisfies the claim we made earlier, i.e., consider two characters x and y with the

Sibling.

اس کے اندر آپ دیکھ سکتے ہیں x اور y بھائی بھائی بن گئے ہیں
مطلب ان کا parents ایک ہی ہے

smallest probabilities. Then there is optimal code tree in which these two characters are siblings at the maximum depth in the tree.

The claim we just proved asserts that the first step of Huffman algorithm is the proper one to perform (the greedy step). The complete proof of correctness for Huffman algorithm follows by induction on n.

Claim: Huffman algorithm produces the optimal prefix code tree. → code distinct.

Proof: The proof is by induction on n, the number of characters. For the basis case, $n = 1$, the tree consists of a single leaf node, which is obviously optimal. We want to show it is true with exactly n characters.

induction method.

Suppose we have exactly n characters. The previous claim states that two characters x and y with the lowest probability will be siblings at the lowest level of the tree. Remove x and y and replace them with a new character z whose probability is $\boxed{p(z) = p(x) + p(y)}$. Thus $n - 1$ characters remain.

Consider any prefix code tree T made with this new set of $n - 1$ characters. We can convert T into prefix code tree T' for the original set of n characters by replacing z with nodes x and y. This is essentially undoing the operation where x and y were removed an replaced by z. The cost of the new tree T' is

z-Parents Node.

$$\begin{aligned}
B(T') &= B(T) - p(z)d(z) + p(x)[d(z)+1] + p(y)[d(z)+1] \\
&= B(T) - (p(x)+p(y))d(z) + (p(x)+p(y))[d(z)+1] \\
&= B(T) + (p(x)+p(y))[d(z)+1-d(z)] \\
&= B(T) + p(x) + p(y)
\end{aligned}$$

Binary Tree/Parents node.

The cost changes but the change depends in no way on the structure of the tree T (T is for $n - 1$ characters). Therefore, to minimize the cost of the final tree T', we need to build the tree T on $n - 1$ characters optimally. By induction, this is exactly what Huffman algorithm does. Thus the final tree is optimal.

Lec#26

7.3 Activity Selection

The activity scheduling is a simple scheduling problem for which the greedy algorithm approach provides an optimal solution. We are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of n activities that are to be scheduled to use some resource. Each activity a_i must be started at a given start time s_i and ends at a given finish time f_i.

An example is that a number of lectures are to be given in a single lecture hall. The start and end times have be set up in advance. The lectures are to be scheduled. There is only one resource (e.g., lecture hall) Some start and finish times may overlap. Therefore, not all requests can be honored. We say that two activities a_i and a_j are non-interfering if their start-finish intervals do not overlap. I.e, $(s_i, f_i) \cap (s_j, f_j) = \emptyset$. The activity selection problem is to select a maximum-size set of mutually non-interfering activities for use of the resource.

So how do we schedule the largest number of activities on the resource? Intuitively, we do not like long

مثال کے طور پر کالج میں ایک Function ہوتا ہے۔ کوئی کہتا ہے کہ میں نے ۱۰ سے ۱۱ بجے تقریر کرنی ہے۔ کوئی کہتا ہے میں نے 2 سے 3 بجے تک یہ کام کرنا ہے تو سارے Function کے لیے ایک مکمل Time بنا لیا جاتا ہے۔ تو شروع سے end تک ایک مکمل scedual میں جائے گا Initial Time کو Ti اور Final Time کو Tf کہیں گے۔ اگر کسی کی activity کے ساتھ بھی ہو جائے تو اسے over lap activities Because they occupy the resource and keep us from honoring other requests. This suggests the greedy strategy: Repeatedly select the activity with the smallest duration $(f_i - s_i)$ and schedule it, provided that it does not interfere with any previously scheduled activities. Unfortunately, this turns out to be non-optimal

Here is a simple greedy algorithm that works: Sort the activities by their finish times. Select the activity that finishes first and schedule it. Then, among all activities that do not interfere with this first job, schedule the one that finishes first, and so on.

```
SCHEDULE(a[1..N])
1  sort a[1..N] by finish times
2  A ← {a[1]};   // schedule activity 1 first
3  prev ← 1;   // most recently scheduled
4  for i = 2 to N
5  do if (a[i].start ≥ a[prev].finish)
6         then A ← A ∪ a[i];  prev ← i
```

Figure 7.5 shows an example of the activity scheduling algorithm. There are eight activities to be scheduled. Each is represented by a rectangle. The width of a rectangle indicates the duration of an activity. The eight activities are sorted by their finish times. The eight rectangles are arranged to show the sorted order. Activity a_1 is scheduled first. Activities a_2 and a_3 interfere with a_1 so they ar not selected. The next to be selected is a_4. Activities a_5 and a_6 interfere with a_4 so are not chosen. The last one to be chosen is a_7. Eventually, only three out of the eight are scheduled.

Timing analysis: Time is dominated by sorting of the activities by finish times. Thus the complexity is $\boxed{O(N \log N)}$

Time → N

چونکہ Hall ایک ہی ہے so ایک ہی Activity ایک وقت میں ہو سکتی ہے۔

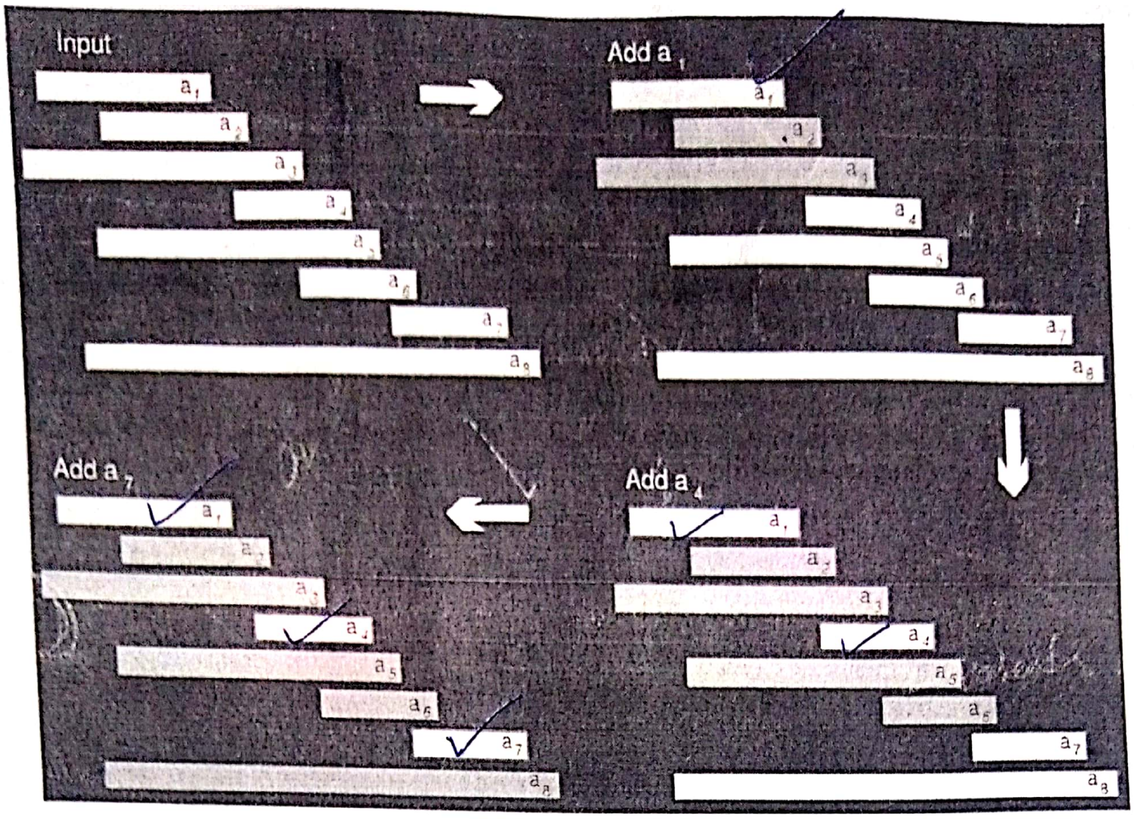


Figure 7.5: Example of greedy activity scheduling algorithm

3-activities

7.3.1 Correctness of Greedy Activity Selection

مطلب زیادہ سے زیادہ Activity کو پورا کرنا

Our proof of correctness is based on showing that the first choice made by the algorithm is the best possible. And then using induction to show that the algorithm is globally optimal. The proof structure is noteworthy because many greedy correctness proofs are based on the same idea: Show that any other solution can be converted into the greedy solution without increasing the cost.

Big Solution.

Claim:

Time.

Let $S = \{a_1, a_2, \ldots, a_n\}$ of n activities, sorted by increasing finish times, that are to be scheduled to use some resource. Then there is an optimal schedule in which activity a_1 is scheduled first.

Proof:

Let A be an optimal schedule. Let x be the activity in A with the smallest finish time. If $x = a_1$ then we are done. Otherwise, we form a new schedule A' by replacing x with activity a_1.

Activity should Not be overlapp.

The greedy algorithm gives an optimal solution to the activity scheduling problem.

Proof:

The proof is by induction on the number of activities. For the basis case, if there are no activities, then the greedy algorithm is trivially optimal. For the induction step, let us assume that the greedy algorithm is optimal on any set of activities of size strictly smaller than $|S|$ and we prove the result for S. Let S' be the set of activities that do not interfere with activity a_1. That is

$$S' = \{a_i \in S | s_i \geq f_1\}$$

Any solution for S' can be made into a solution for S by simply adding activity a_1, and vice versa. Activity a_1 is in the optimal schedule (by the above previous claim). It follows that to produce an optimal schedule for the overall problem, we should first schedule a_1 and then append the optimal schedule for S'. But by induction (since $|S'| < |S|$), this exactly what the greedy algorithm does.

آپ کے پاس ایک Bag ہوتا ہے اس میں آپ زیادہ سے زیادہ چیزیں ڈالنے کی کوشش کرتے ہیں اس کے وزن اور جگہ کا خیال رکھتے ہوئے۔

Lect#(27)

7.4 Fractional Knapsack Problem

Earlier we saw the 0-1 knapsack problem. A knapsack can only carry W total weight. There are n items; the ith item is worth v_i and weighs w_i. Items can either be put in the knapsack or not. The goal was to maximize the value of items without exceeding the total weight limit of W. In contrast, in the fractional knapsack problem, the setup is exactly the same. But, one is allowed to take *fraction* of an item for a fraction of the weight and fraction of value. The 0-1 knapsack problem is hard to solve. However, there is a simple and efficient greedy algorithm for the fractional knapsack problem.

item → value → weight

Let $\rho_i = v_i/w_i$ denote the *value per unit weight* ratio for item i. Sort the items in decreasing order of ρ_i. Add items in decreasing order of ρ_i. If the item fits, we take it all. At some point there is an item that does not fit in the remaining space. We take as much of this item as possible thus filling the knapsack completely.

اگر کوئی قیمتی چیزیں ہوں اور وہ ان کو اپنے Bag میں ڈالنا چاہے تو ان قیمتی چیزوں کے حصے کر لیے جا سکیں۔

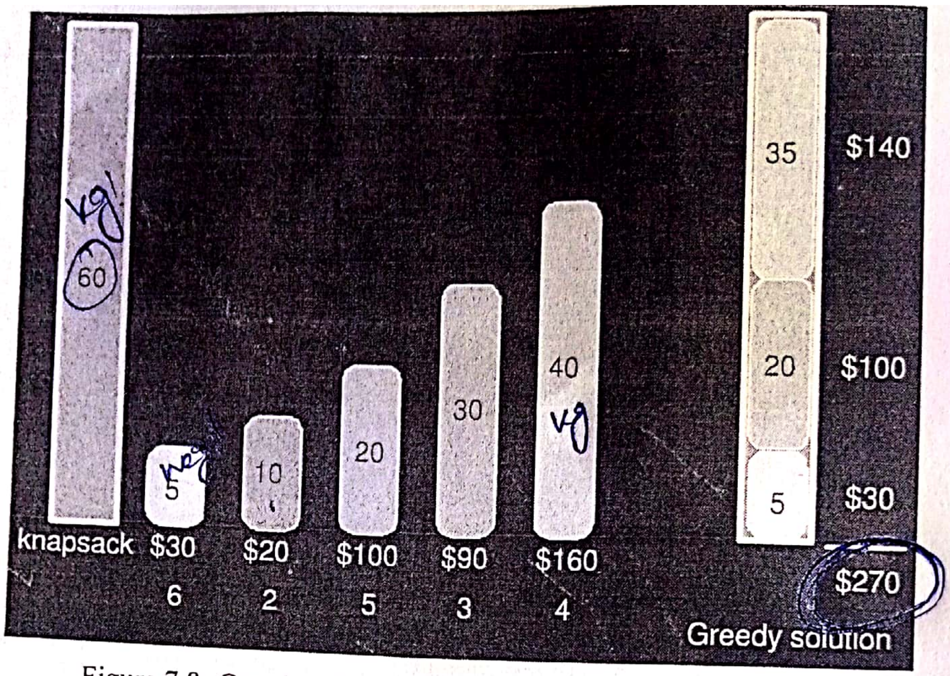


Figure 7.8: Greedy solution to the fractional knapsack problem

40kg $160 والے Bag بہت قیمتی ہے. اس میں سے 35kg نکال لیں گے.
باقی 25kg bag میں بچی. اب ہم $100 والے 20kg والے Bag میں سارے لیں گیں
باقی 5 کلو کی جگہ بچے گی تو ہم $30 والا پورا سامان ڈال لیں گیں.

It is easy to see that the greedy algorithm is optimal for the fractional knapsack problem. Given a room with sacks of gold, silver and bronze, one (thief?) would probably take as much gold as possible. Then take as much silver as possible and finally as much bronze as possible. It would never benefit to take a little less gold so that one could replace it with an equal weight of bronze.

We can also observe that the greedy algorithm is not optimal for the 0-1 knapsack problem. Consider the example shown in the Figure 7.9. If you were to sort the items by ρ_i, then you would first take the items of weight 5, then 20, and then (since the item of weight 40 does not fit) you would settle for the item of weight 30, for a total value of $30 + $100 + $90 = $220. On the other hand, if you had been less greedy, and ignored the item of weight 5, then you could take the items of weights 20 and 40 for a total value of $100+$160 = $260. This is shown in Figure 7.10.

Lect# 28

Chapter 8

Graphs

For use to solve any problem.

We begin a major new topic: Graphs. Graphs are important discrete structures because they are a flexible mathematical model for many application problems. Any time there is a set of objects and there is some sort of "connection" or "relationship" or "interaction" between pairs of objects, a graph is a good way to model this. Examples of this can be found in computer and communication networks transportation networks, e.g., roads VLSI, logic circuits surface meshes for shape description in computer-aided design and GIS precedence constraints in scheduling systems.

ہر چیز اس طرح 2 ہوتی ہے جیسے وہ گراف بنا رہے ہوں

A *graph* G = (V, E) consists of a finite set of *vertices* V (or nodes) and E, a binary relation on V called *edges*. E is a set of pairs from V. If a pair is *ordered*, we have a *directed* graph. For *unordered* pair, we have an *undirected* graph.

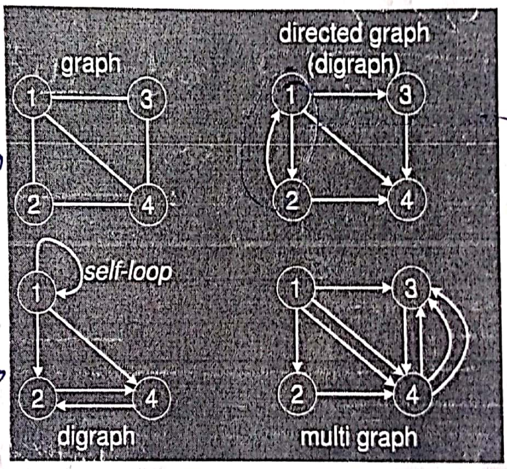
Roads

V → روڈز

Undirected

ہر vertex 1, 2, 3, 4 ہیں۔

اگر ایک گھر سے سفر شروع ہو گیا اور گھوم گھما کر وہیں واپس آ گئے تو وہ self loop ہو گا



→ double

Figure 8.1: Types of graphs

A vertex *w* is *adjacent* to vertex *v* if there is an edge from *v* to *w*.

By:- Mudassar Iqbal!

undirected → graph.

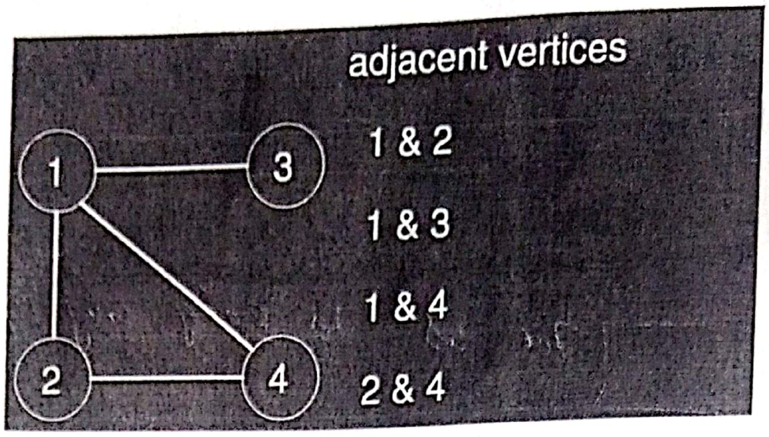


Figure 8.2: Adjacent vertices

In an undirected graph, we say that an edge is *incident* on a vertex if the vertex is an endpoint of the edge. of the edge

1 — 3 منسلک ہوتے

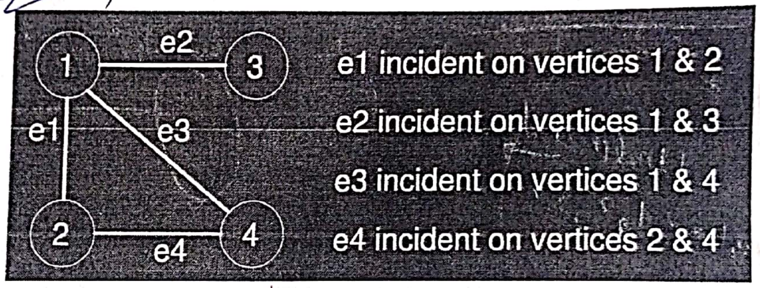

e1 incident on vertices 1 & 2
e2 incident on vertices 1 & 3
e3 incident on vertices 1 & 4
e4 incident on vertices 2 & 4
e1
e2
e3
e4
1
2
3
4

Figure 8.3: Incidence of edges on vertices

جب ہم کسی vertex میں موجود ہوتے ہیں وہ in-degree کہلاتا ہے

In a digraph, the number of edges coming out of a vertex is called the *out-degree* of that vertex. Number of edges coming in is the *in-degree*..In an undirected graph, we just talk of degree of a vertex. It is the number of edges incident on the vertex.

جب ہم اس سے نکل جاتے ہیں وہ out degree

directed →

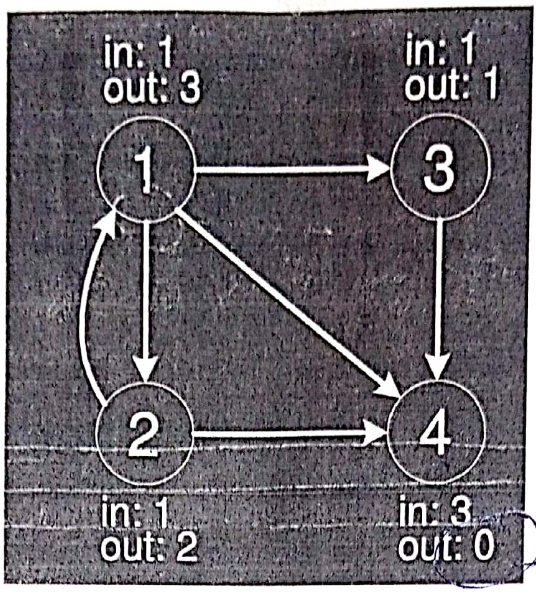


Figure 8.4: In and out degrees of vertices of a graph

0 اس لیے کہ یہاں سے واپس کوئی سڑک نہیں آتی اس لیے ہم out میں ہو گئے

For a digraph G = (V, E),

$$\boxed{\sum_{v \in V} \text{in-degree}(v) = \sum_{v \in V} \text{out-degree}(v) = |E|} \rightarrow \text{edges.}$$

where |E| means the cardinality of the set E, i.e., the number of edges.

For an undirected graph G = (V, E),

$$\sum_{v \in V} \text{degree}(v) = 2|E| \rightarrow$$

where |E| means the cardinality of the set E, i.e., the number of edges.

A *path* in a directed graphs is a sequence of vertices $\langle v_0, v_1, \ldots, v_k \rangle$ such that (v_{i-1}, v_i) is an edge for $i = 1, 2, \ldots, k$. The *length* of the paths is the number of edges, k. A vertex w is *reachable* from vertex u is there is a path from u to w. A path is simple if all vertices (except possibly the fist and last) are distinct.

A *cycle* in a digraph is a path containing at least one edge and for which $v_0 = v_k$. A *Hamiltonian* (vertex.) cycle is a cycle that visits every vertex in a graph exactly once. A *Eulerian* (→ edges.) cycle is a cycle that visits every edge of the graph exactly once. There are also "path" versions in which you do not need return to the starting vertex.

اگر ایک vertex سے شروع کریں اور گھوم گھما کر واپس اسی vertex پر واپس آ جائیں اس Hamiltonian cycle کہیں گے

اگر ایک جگہ سے edges سے چلے ہوئے دوبارہ اسی edges پر واپس آئے تو اسے Eulerian cycle کہیں گے

اگر ہم ایک ہی node کو ایک دفعہ سے زیادہ visit کر رہے ہیں تو وہ
3 cycle.

جہاں یہ نہ ہو acyclic graph ہوگا

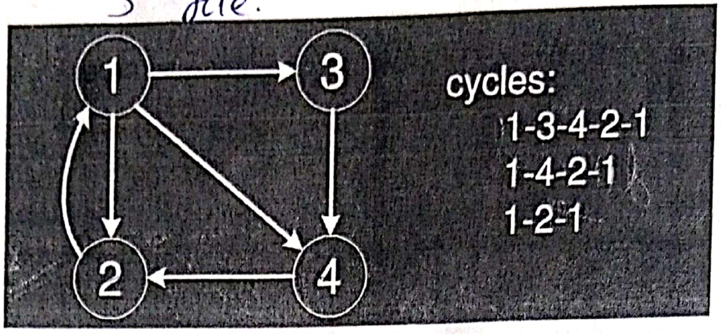


Figure 8.5: Cycles in a directed graph

A graph is said to be *acyclic* if it contains no cycles. A graph is *connected* if every vertex can reach every other vertex. A directed graph that is acyclic is called a *directed acyclic graph (DAG)*.

There are two ways of representing graphs: using an adjacency matrix and using an adjacency list. Let $G = (V, E)$ be a digraph with $n = |V|$ and let $e = |E|$. We will assume that the vertices of G are indexed $\{1, 2, \ldots, n\}$.

An *adjacency matrix* is a $n \times n$ matrix defined for $1 \le v, w \le n$.

$$A[v, w] = \begin{cases} 1 & \text{if } (v, w) \in E \\ 0 & \text{otherwise} \end{cases}$$

An *adjacency list* is an array $Adj[1..n]$ of pointers where for $1 \le v \le n$, $Adj[v]$ points to a linked list containing the vertices which are adjacent to v

Adjacency matrix requires $\Theta(n^2)$ storage and adjacency list requires $\Theta(n + e)$ storage. list
Θ link list

اگر راستہ موجود ہو تو 1 ہے اگر نہ ہو تو 0 ہے

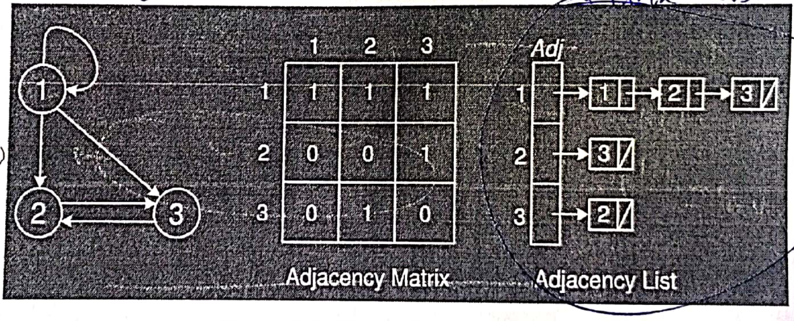


Figure 8.6: Graph Representations

اگر ہمیں کسی نفل کے [illegible] کسی کو approach کرنا ہو تو 1 سے شروع کر کے dense کریں گے
اگر نہ ہو تو 0 سے شروع کر کے dense کریں گے

8.1 Graph Traversal

To motivate our first algorithm on graphs, consider the following problem. We are given an undirected graph $G = (V, E)$ and a *source vertex* $s \in V$. The *length* of a path in a graph is the number of edges on

link list میں وہی جو ہوں گے جو راستہ موجود ہو گا اگر راستہ میں
موجود ہو گا تو ہم اس کو link list میں استعمال میں کریں گے

the path. We would like to find the shortest path from s to each other vertex in the graph. The final result will be represented in the following way. For each vertex $v \in V$, we will store $d[v]$ which is the *distance* (length of the shortest path) from s to v. Note that $d[s] = 0$. We will also store a predecessor (or parent) pointer $\pi[v]$ which is the first vertex along the shortest path if we walk from v backwards to s. We will set $\pi[s] = \text{Nil}$.

There is a simple brute-force strategy for computing shortest paths. We could simply start enumerating all simple paths starting at s, and keep track of the shortest path arriving at each vertex. However, there can be as many as n! simple paths in a graph. To see this, consider a fully connected graph shown in Figure 8.7

آپ ایک سادہ سے طریقہ سے vertex [illegible] vertex [illegible] جب آپ [illegible]

[illegible] vertex [illegible] vertex [illegible]

Vertex میں جو [illegible] shortest راستہ دیکھو [illegible] 2 [illegible]

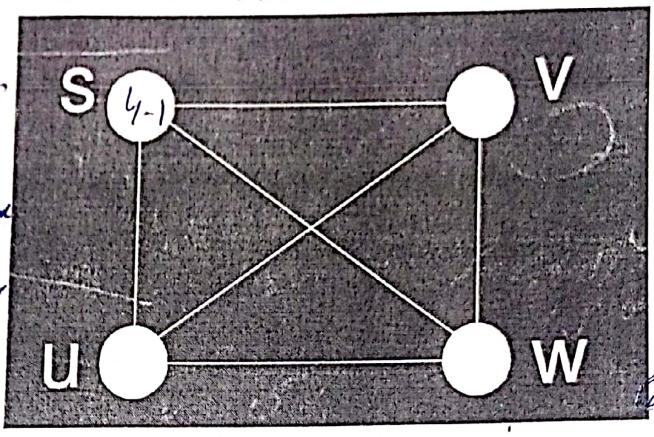


Figure 8.7: Fully connected graph

4 vertex.

There n choices for source node s, $(n-1)$ choices for destination node, $(n-2)$ for first hop (edge) in the path, $(n-3)$ for second, $(n-4)$ for third down to $(n-(n-1))$ for last leg. This leads to n! simple paths. Clearly this is not feasible.

Lect # 29.

8.1.1 Breadth-first Search

like BST $\uparrow$ Binary search Tree.

(Leaf Node) [illegible] visit [illegible] nodes [illegible] Parents [illegible]

Here is a more efficient algorithm called the *breadth-first search* (BFS) Start with s and visit its adjacent nodes. Label them with distance 1. Now consider the neighbors of neighbors of s. These would be at distance 2. Now consider the neighbors of neighbors of neighbors of s. These would be at distance 3. Repeat this until no more unvisited neighbors left to visit. The algorithm can be visualized as a *wave front* propagating outwards from s visiting the vertices in bands at ever increasing distances from s.

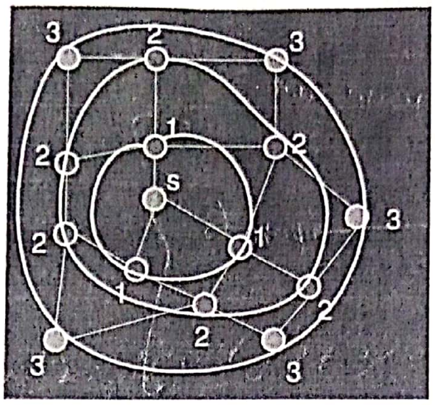


Figure 8.11: Wave reaching distance 3 vertices during BFS

8.1.2 Depth-first Search

مثال کے طور پر آپ ایک نئی Building میں گئے اور آپ کو پتا نہیں

Breadth-first search is one instance of a general family of *graph traversal algorithms*. Traversing a graph means visiting every node in the graph. Another traversal strategy is *depth-first search* (DFS). DFS procedure can be written recursively or non-recursively. Both versions are passed s initially.

[illegible] یہ کہ اس کے ساتھ کون کون سے کمرے منسلک ہیں

```
RECURSIVEDFS(v)
if (v is unmarked )
    then mark v
        for each edge (v, w)
        do RECURSIVEDFS(w)
```

vertex → connected room.

ہر کمرے سے دروازے نکلتے ہیں اب آپ سارے کمروں کو visit کر لیتے ہو۔ ابھی کچھ ایسا ہی ہو رہا ہے Main دروازے سے شروع کرتا ہے

```
ITERATIVEDFS(s)
PUSH(s)
while stack not empty
do v ← POP()
    if v is unmarked
        then mark v
            for each edge (v, w)
            do PUSH(w)
```

نکالنا

جب آپ ایک کمرے کو visit کر لیتے ہو تو اس کو Label کر دیتے ہیں تاکہ دوبارہ visit نہ کرو

8.1.3 Generic Graph Traversal Algorithm

The *generic graph traversal* algorithm stores a set of candidate edges in some data structures we'll call a *"bag"*. The only important properties of the "bag" are that we can put stuff into it and then later take stuff back out. Here is the generic traversal algorithm.

```
TRAVERSE(s)  → Main room
1  put (∅, s) in bag
2  while bag not empty
3  do take (p, v) from bag
4     if (v is unmarked )  → parents
5        then mark v
6             parent (v) ← p
7             for each edge (v, w)
8             do put (v, w) in bag
```

Notice that we are keeping edges in the bag instead of vertices. This is because we want to remember, whenever we visit v for the first time, which previously-visited vertex p put v into the bag. The vertex p is call the *parent of* v.

The running time of the traversal algorithm depends on how the graph is represented and what data structure is used for the bag. But we can make a few general observations:

Lect#30

- Since each vertex is visited at most once, the for loop in line 7 is executed at most V times.
- Each edge is put into the bag exactly twice; once as (u, v) and once as (v, u), so line 8 is executed at most 2E times.
- Finally, since we can't take out more things out of the bag than we put in, line 3 is executed at most 2E + 1 times.
- Assume that the graph is represented by an adjacency list so the overhead of the for loop in line 7 is constant per edge.

If we implement the bag by using a *stack*, we have *depth-first* search (DFS) or traversal.

```
TRAVERSE(s) → Stack
1  push(∅, s)        Trace 5
2  while stack not empty
3  do pop(p, v)
4     if (v is unmarked )
5        then mark v
6             parent (v) ← p
7             for each edge (v, w)
8             do push(v, w) → connected room
```

[illegible]

Figures 8.12 to 8.20 show a trace of the DFS algorithm applied to a graph. The figures show the content of the stack during the execution of the algorithm.

or line 8 still takes constant time. So overall running time is still O(E).

```
TRAVERSE(s)
1  enqueue(∅, s)
2  while queue not empty
3  do dequeue(p, v)
4      if (v is unmarked )
5        then mark v
6             parent (v) ← p
7             for each edge (v, w)
8             do enqueue(v, w)
```

قطار کے حساب سے
[illegible]

If the graph is represented using an *adjacency matrix*, the finding of all the neighbors of vertex in line 7 takes $O(V)$ time. Thus depth-first and breadth-first take $O(V^2)$ time overall.

Either DFS or BFS yields a spanning tree of the graph. The tree visits every vertex in the graph. This fact is established by the following lemma:

Lemma:

The generic TRAVERSE(s) marks every vertex in any connected graph exactly once and the set of edges $(v, \text{parent}(v))$ with $\text{parent}(v) \neq \varnothing$ form a spanning tree of the graph.

Proof:

First, it should be obvious that no vertex is marked more than once. Clearly, the algorithm marks s. Let $v \neq s$ be a vertex and let $s \to \cdots \to u \to v$ be a path from s to v with the minimum number of edges.

Since the graph is connected, such a path always exists. If the algorithm marks u, then it must put (u, v) into the bag, so it must take (u, v) out of the bag at which point v must be marked. Thus, by induction on the shortest-path distance from s, the algorithm marks every vertex in the graph.

Call an edge $(v, \text{parent}(v))$ with $\text{parent}(v) \neq \varnothing$, a *parent edge*. For any node v, the path of parent edges $v \to \text{parent}(v) \to \text{parent}(\text{parent}(v)) \to \ldots$ eventually leads back to s. So the set of parent edges form a connected graph.

Clearly, both end points of every parent edge are marked, and the number of edges is exactly one less than the number of vertices. Thus, the parent edges form a *spanning tree*.

Lect#31 depth first search.

8.1.4 DFS - Timestamp Structure

As we traverse the graph in DFS order, we will associate two numbers with each vertex. When we first discover a vertex u, store a counter in d[u]. When we are finished processing a vertex, we store a counter in f[u]. These two numbers are *time stamps*.

Consider the *recursive* version of depth-first traversal depth first search.

```
DFS(G)
1  for (each u ∈ V)
2  do color[u] ← white still not visit.
3     pred[u] ← nil
4  time ← 0
5  for each u ∈ V          vertex کے [illegible]
6  do if (color[u] = white)
7        then DFSVISIT(u)
```

The DFSVISIT routine is as follows:

```
DFSVISIT(u)
1  color[u] ← gray;  // mark u visited
2  d[u] ← ++ time
3  for (each v ∈ Adj[u])        جو کہ visit نہیں ہوئے ان کو white
4  do if (color[v] = white)
5        then pred[v] ← u       جو کہ visit ہو گئے ان کو Black کر دیں.
6             DFSVISIT(v)
7  color[u] ← black;  // we are done with u
8  f[u] ← ++ time;
```

Figures 8.21 through 8.25 present a trace of the execution of the time stamping algorithm. Terms like "2/5" indicate the value of the counter (time). The number before the "/" is the time when a vertex was discovered (colored gray) and the number after the "/" is the time when the processing of the vertex finished (colored black).

Main room

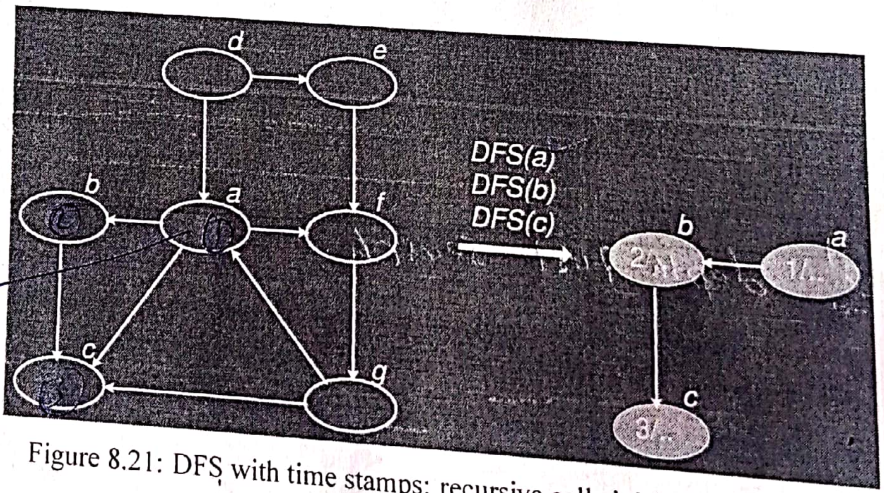


Figure 8.21: DFS with time stamps: recursive calls initiated at vertex 'a'

Start Time.

Forward edge: (u, v) where v is a proper descendent of u in the tree.

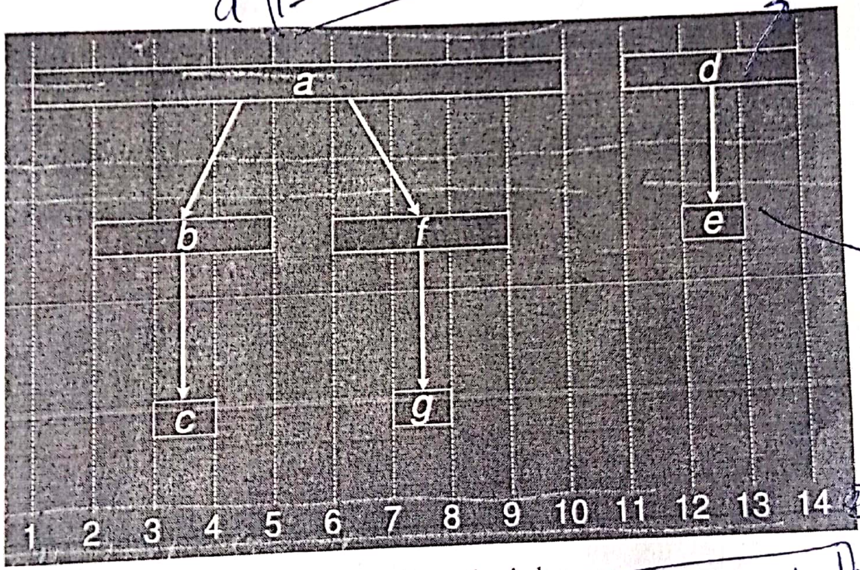
Cross edge: (u, v) where u and v are not ancestor or descendent of one another. In fact, the edge may go between different trees of the forest.

اس میں کوئی relation نہیں ہے

The ancestor and descendent relation can be nicely inferred by the *parenthesis* lemma. u is a descendent of v if and only if [d[u], f[u]] ⊆ [d[v], f[v]]. u is a ancestor of v if and only if [d[u], f[u]] ⊇ [d[v], f[v]]. u is unrelated to v if and only if [d[u], f[u]] and [d[v], f[v]] are disjoint. The is shown in Figure 8.26. The width of the rectangle associated with a vertex is equal to the time the vertex was discovered till the time the vertex was completely processed (colored black). Imagine an opening parenthesis '(' at the start of the rectangle and and closing parenthesis ')' at the end of the rectangle. The rectangle (parentheses) for vertex 'b' is completely enclosed by the rectangle for 'a'. Rectangle for 'c' is completely enclosed by vertex 'b' rectangle.

a $\boxed{1 — 10}$

ences



Trees کو الگ کرتے ہیں دو

d,e are $\boxed{\text{cross edges}}$

کیونکہ e نہ d کا

d کا des

ancestor ہے

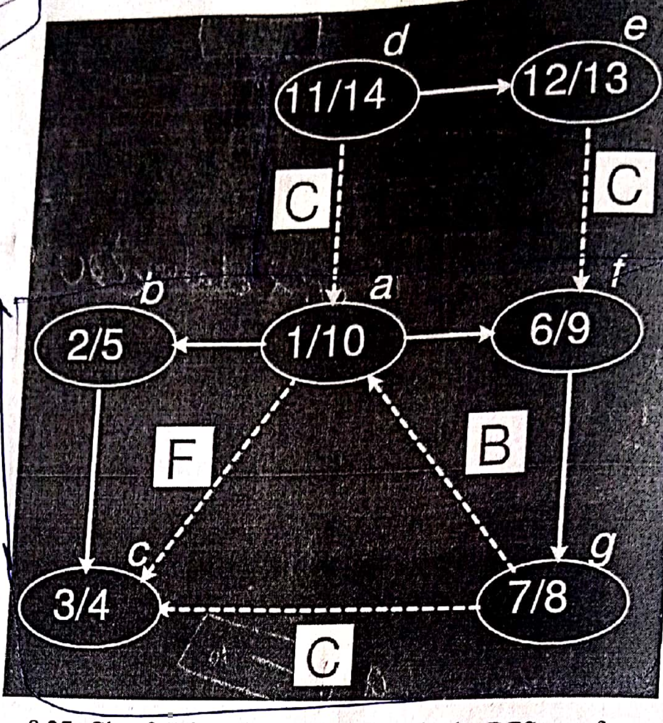
Figure 8.26: Parenthesis lemma

$\boxed{\text{b, 7, c, g are descendent}}$

$\boxed{\text{a is ancestor}}$

Figure 8.27 shows the classification of the non-tree edges based on the parenthesis lemma. Edges are labelled 'F', 'B' and 'C' for forward, back and cross edge respectively.

Lect #32



→ Cross edge.

اس گراف میں سے Forward, Backward اور Cross edges کو تلاش کریں

Figure 8.27: Classfication of non-tree edges in the DFS tree for a graph

For *undirected* graphs, there is no distinction between forward and back edges. By convention they are all called back edges. Furthermore, there are no cross edges (can you see why not?)

8.1.5 DFS - Cycles

The time stamps given by DFS allow us to determine a number of things about a graph or digraph. For example, we can determine whether the graph contains any *cycles*. We do this with the help of the following two lemmas.

Lemma: Given a digraph $G = (V, E)$, consider any DFS forest of G and consider any edge $(u, v) \in E$. If this edge is a tree, forward or cross edge, then $f[u] > f[v]$. If this edge is a back edge, then $f[u] \leq f[v]$.

Proof: For the non-tree forward and back edges the proof follows directly from the parenthesis lemma. For example, for a forward edge (u, v), v is a descendent of u and so v's start-finish interval is contained within u's implying that v has an earlier finish time. For a cross edge (u, v) we know that the two time intervals are disjoint. When we were processing u, v was not white (otherwise (u, v) would be a tree edge), implying that v was started before u. Because the intervals are disjoint, v must have also finished before u.

Lemma: Consider a digraph $G = (V, E)$ and any DFS forest for G. G has a cycle if and only if the DFS forest has a *back* edge.

اگر ہم گراف میں واپس اسی جگہ آ جائیں تو وہ Back edge ہوگا اور اسے cycles کہتے ہیں

Proof: If there is a back edge (u, v) then v is an ancestor of u and by following tree edge from v to u, we get a cycle.

We show the contrapositive: suppose there are no back edges. By the lemma above, each of the remaining types of edges, tree, forward, and cross all have the property that they go from vertices with higher finishing time to vertices with lower finishing time. Thus along any path, finish times decrease monotonically, implying there can be no cycle.

The DFS forest in Figure 8.27 has a back edge from vertex 'g' to vertex 'a'. The cycle is 'a-g-f'.

Beware: No back edges means no cycles. But you should not infer that there is some simple relationship between the number of back edges and the number of cycles. For example, a DFS tree may only have a single back edge, and there may anywhere from one up to an exponential number of simple cycles in the graph.

A similar theorem applies to undirected graphs, and is not hard to prove.

اگر کسی گراف میں Back edge ایک ہو تو اسکا مطلب ہرگز نہیں کہ cycle ایک ایک ہی ہے cycle اور بھی ہو سکتے ہیں

8.2 Precedence Constraint Graph

ہر کام ایک طریقے سے ہوگا۔ آپ ورک forward ہی جا سکتے ہیں Backward نہیں جا سکتے۔

A *directed acyclic graph* (DAG) arise in many applications where there are precedence or ordering constraints. There are a series of tasks to be performed and certain tasks must precede other tasks. For example, in construction, you have to build the first floor before the second floor but you can do electrical work while doors and windows are being installed. In general, a *precedence constraint graph* is a DAG in which vertices are tasks and the edge (u, v) means that task u must be completed before task v begins.

For example, consider the sequence followed when one wants to dress up in a suit. One possible order and its DAG are shown in Figure 8.28. Figure 8.29 shows the DFS with time stamps of the DAG.

جس گراف میں Cycle موجود نہیں ہوتا اسے (DAG) کہتے ہیں۔
اس میں Back edge نہیں ہوتا۔

اس کے اندر آپ پہلے والے کورس مکمل کریں گے پھر Next کے پروگرام شروع ہوں

C1	Introduction to Computers	
C2	Introduction to Computer Programming	
C3	Discrete Mathematics	
C4	Data Structures	C2
C5	Digital Logic Design	C2
C6	Automata Theory	C3
C7	Analysis of Algorithms	C3, C4
C8	Computer Organization and Assembly	C2
C9	Data Base Systems	C4, C7
C10	Computer Architecture	C4, C5,C8
C11	Computer Graphics	C4,C7
C12	Software Engineering	C7,C11
C13	Operating System	C4,C7,C11
C14	Compiler Construction	C4,C6,C8
C15	Computer Networks	C4,C7,C10

Table 8.1: Prerequisites for CS courses

The prerequisites can be represented with a precedence constraint graph which is shown in Figure 8.30

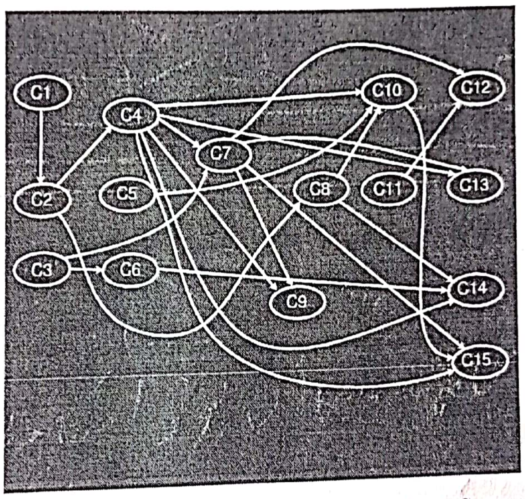


Figure 8.30: Precedence constraint graph for CS courses

8.3 Topological Sort

Directed acyclic graph changes into linear ordering.

A topological sort of a DAG is a linear ordering of the vertices of the DAG such that for each edge (u, v), u appears before v in the ordering.

Computing a topological ordering is actually quite easy, given a DFS of the DAG. For every edge (u, v) in a DAG, the finish time of u is greater than the finish time of v (by the lemma). Thus, it suffices to output the vertices in the reverse order of finish times.

Lect 33.

8.4 Strong Components

Example of DFS

We consider an important connectivity problem with digraphs. When diagraphs are used in communication and transportation networks, people want to know that their networks are *complete*. Complete in the sense that that it is possible from any location in the network to reach any other location in the digraph.

A digraph is *strongly connected* if for every pair of vertices $u, v \in V$, u can reach v and vice versa. We would like to write an algorithm that determines whether a digraph is strongly connected. In fact, we will solve a generalization of this problem, of computing the *strongly connected components* of a digraph. We partition the vertices of the digraph into subsets such that the induced subgraph of each subset is strongly connected. We say that two vertices u and v are *mutually reachable* if u can reach v and vice versa. Consider the directed graph in Figure 8.32. The strong components are illustrated in Figure 8.33.

Backward edges are strong connected.

Not Strong.

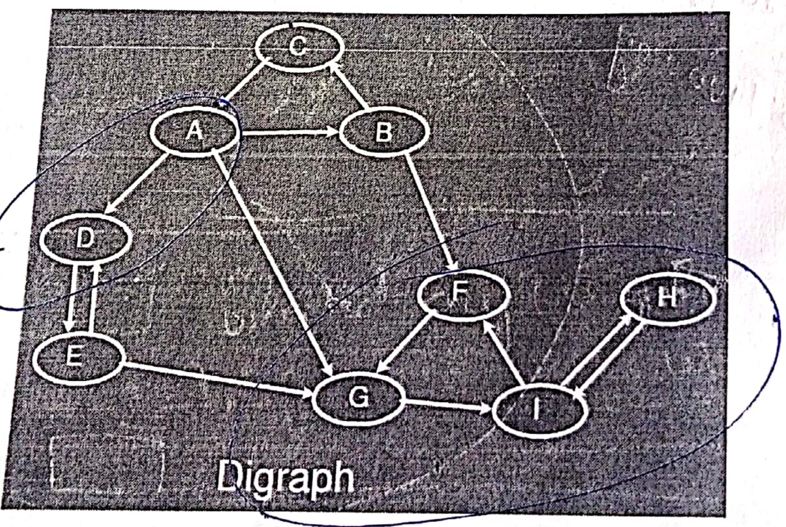


Figure 8.32: A directed graph

Strong connectivity

Strong component وہ ہوتے ہیں جن میں ہر کسی Node سے جا کے واپس بھی آ سکتے ہیں

Strong

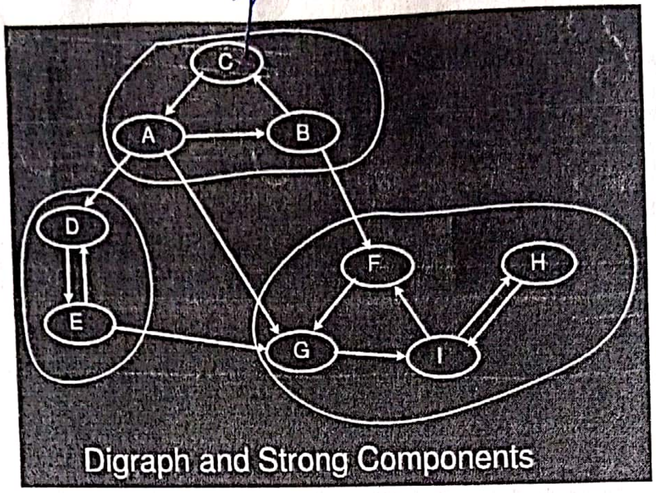


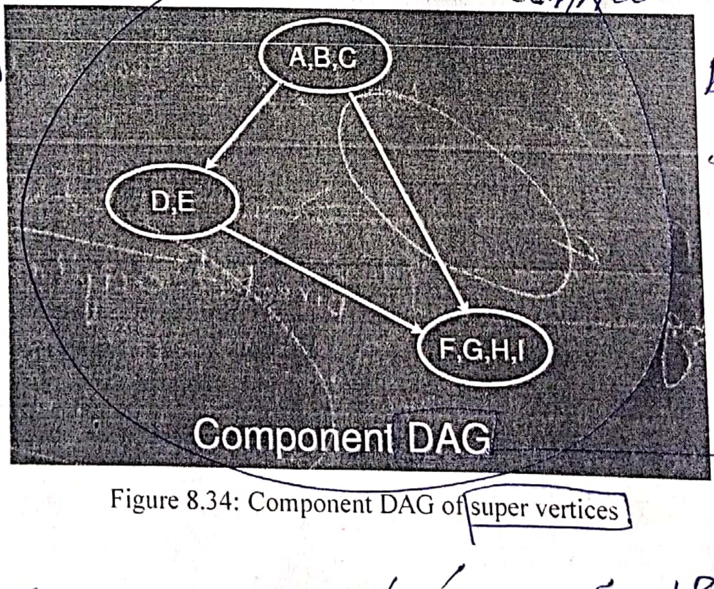
Figure 8.33: Digraph with strong components

It is easy to see that mutual reachability is an *equivalence relation*. This equivalence relation partitions the vertices into equivalence classes of mutually reachable vertices and these are the strong components.

If we merge the vertices in each strong component into a single *super vertex*, and join two super vertices (A, B) if and only if there are vertices $u \in A$ and $v \in B$ such that $(u, v) \in E$, then the resulting digraph is called the *component digraph*. The component digraph is necessarily acyclic. The is illustrated in Figure 8.34.

Strong connectivity میں ہوتی ہیں Two way ہوتا

connectivity but not Strong connectivity.



ڈائریکٹ directed acyclic graph

Figure 8.34: Component DAG of super vertices

جب ہم vertex کو ایک دوسرے میں merge کر دیتے ہیں تو وہ super vertex کہلائے گا

Is there a way to order the DFS such that it true? Fortunately, the answer is "yes". Suppose that you knew the component DAG in advance. (This is ridiculous, because you would need to know the strong components and this is the problem we are trying to solve.) Further, suppose that you computed a reversed topological order on the component DAG. That is, for edge (u, v) in the component DAG, then v comes before u. This is presented in Figure 8.37. Recall that the component DAG consists of super vertices.

Forward.

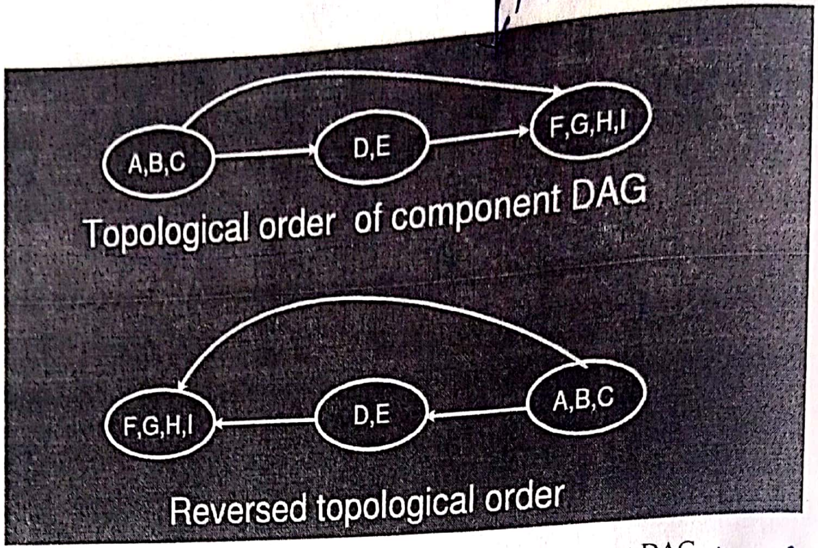

A,B,C
D,E
F,G,H,I
Topological order of component DAG
F,G,H,I
D,E
A,B,C
Reversed topological order

Figure 8.37: Reversed topological sort of component DAG

Topology, move from right to left

Lecture#34

Now, run DFS, but every time you need a new vertex to start the search from, select the next available vertex according to this reverse topological order of the component digraph. Here is an informal justification. Clearly once the DFS starts within a given strong component, it must visit every vertex within the component (and possibly some others) before finishing. If we do not start in reverse topological, then the search may "leak out" into other strong components, and put them in the same DFS tree. For example, in the Figure 8.36, when the search is started at vertex 'a', not only does it visit its component with 'b' and 'c', but it also visits the other components as well. However, by visiting components in reverse topological order of the component tree, each search cannot "leak out" into other components, because other components would have already have been visited earlier in the search.

This leaves us with the intuition that if we could somehow order the DFS, so that it hits the strong components according to a reverse topological order, then we would have an easy algorithm for computing strong components. However, we do not know what the component DAG looks like. (After all, we are trying to solve the strong component problem in the first place). The trick behind the strong component algorithm is that we can find an ordering of the vertices that has essentially the necessary property, without actually computing the component DAG.

We will discuss the algorithm without proof. Define G^T to be the digraph with the same vertex set at G but in which all edges have been reversed in direction. This is shown in Figure 8.38. Given an adjacency list for G, it is possible to compute G^T in $\Theta(V + E)$ time.

G Simple ہے، ہو گا۔ اب اگر ہم نے edges کو Reverse کر دیا

direction change کر دیں گے تو وہ G^{t} (t→Transpose) ہو گا۔

مثال کے طور پر آپ ایک شہر میں Cable بچھانا چاہتے ہیں آپ اس Cable کا ایک Head office بنائیں گے اس کے بعد اس کو گھر گھر بھی پہنچانا ہوگا تو اس کیلئے کم سے کم کتنا خرچہ 4+2 کا لگے تار کتنی کم سے کم لگے گی اس کو آپ calculate کریں گے

8.5 Minimum Spanning Trees

MST

A common problem is communications networks and circuit design is that of connecting together a set nodes by a network of total minimum length. The length is the sum of lengths of connecting wires. Consider, for example, laying cable in a city for cable t.v.

The computational problem is called the *minimum spanning tree* (MST) problem. Formally, we are given a connected, undirected graph $G = (V, E)$ Each edge (u, v) has numeric weight of cost. We define the cost of a spanning tree T to be the sum of the costs of edges in the spanning tree

① تھوڑے خرچ سے کیا

② Time اور Space کم لگے

$$\boxed{w(T) = \sum_{(u,v)\in T} \underline{w}(u, v)}$$

weight of Tree

اس بات کو ترجیح دے الگورتھم پہلے کام کر رہا تھا نہیں

A minimum spanning tree is a tree of minimum weight.

Figures ??, ?? and ?? show three spanning trees for the same graph. The first is a spanning tree but is not a MST; the other two are.

جن راستوں سے گزریں گے ان کو ہم add کریں گے وزن بھی لوں گا اور cost بھی ہم

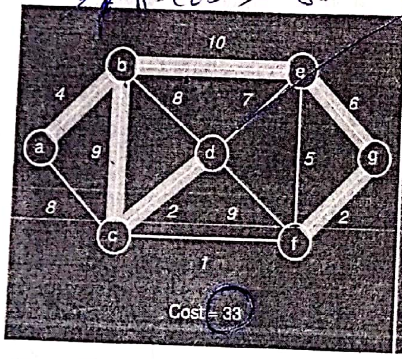


Figure 8.42: A spanning tree that is *not* MST

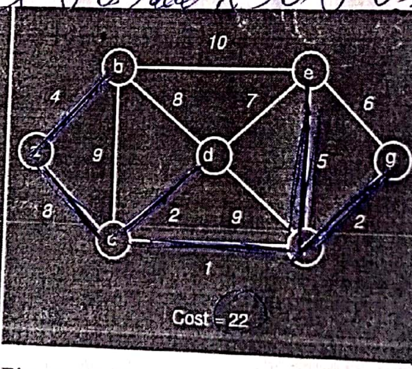


Figure 8.43: A minimum spanning tree

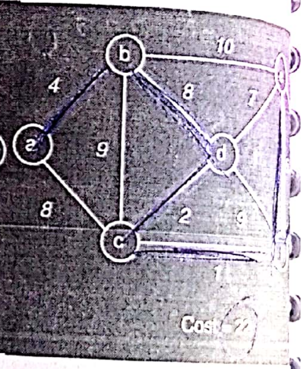


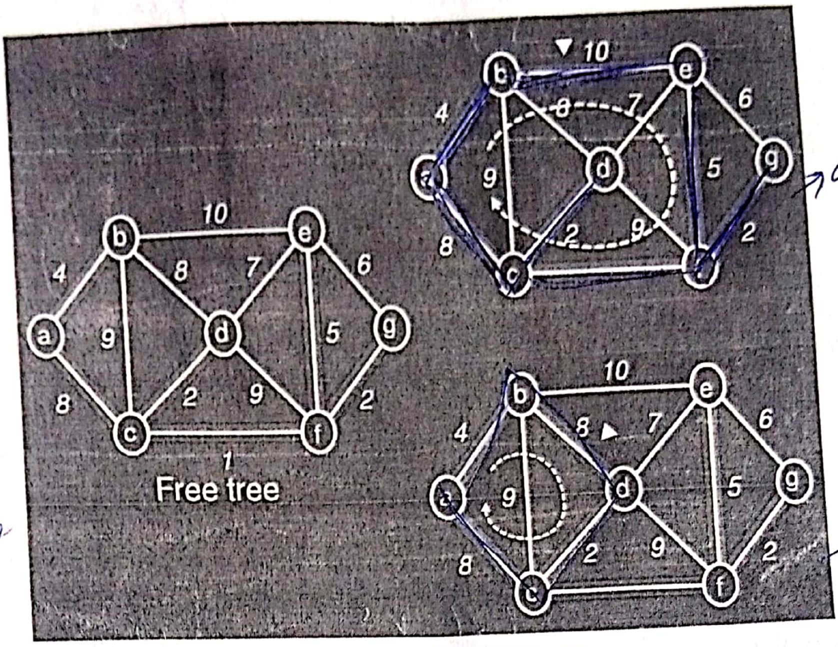
Figure 8.44: Another minimum spanning tree

We will present two *greedy* algorithms (Kruskal's and Prim's) for computing MST. Recall that a greedy algorithm is one that builds a solution by repeatedly selecting the cheapest among all options at each stage. Once the choice is made, it is never undone.

Before presenting the two algorithms, let us review facts about *free trees*. A free tree is a tree with no vertex designated as the root vertex. A free tree with n vertices has exactly n − 1 edges. There exists a unique path between any two vertices of a free tree. Adding any edge to a free tree creates a unique cycle. Breaking any edge on this cycle restores the free tree. This is illustrated in Figure 8.45. When the edges (b, e) or (b, d) are added to the free tree, the result is a cycle.

جس vertex میں Root کا ہونا ضروری نہیں اس میں cycle نہیں بنا.

ایسا Tree جس میں ہم تمام vertices کو visit کر سکیں اسے Spaning Tree کہتے ہیں



→cycle

→cycle

Free Tree جس میں کوئی cycle نہیں ہوتی

Figure 8.45: Free tree facts

اس میں کوئی cycle نہیں بنتا

Root ایسا choose نہیں کرنا جس میں

8.5.1 Computing MST: Generic Approach

Let G = (V, E) be an undirected, connected graph whose edges have numeric weights. The intuition behind greedy MST algorithm is simple: we maintain a subset of edges E of the graph . Call this subset A. Initially, A is empty. We will add edges one at a time until A equals the MST.

A subset A ⊆ E is *viable* if A is a subset of edges of *some* MST. An edge (u, v) ∈ E − A is *safe* if A ∪ {(u, v)} is viable. In other words, the choice (u, v) is a safe choice to add so that A can still be extended to form a MST.

جس کا ویٹ کم ہو گا وہ safe ہو گا

Note that if A is viable, it cannot contain a cycle. A generic greedy algorithm operates by repeatedly adding any *safe* edge to the current spanning tree.

بار بار اضافہ کرنا

When is an edge safe? Consider the theoretical issues behind determining whether an edge is safe or n Let S be a subset of vertices S ⊆ V. A *cut* (S, V − S) is just a partition of vertices into two disjoint subsets. An edge (u, v) *crosses* the cut if one endpoint is in S and the other is in V − S. Given a subset of edges A, a cut *respects* A if no edge in A crosses the cut. It is not hard to see why respecting cuts are important to this problem. If we have computed a partial MST and we wish to kn which edges can be added that *do not* induce a cycle in the current MST, any edge that crosses a respecting cut is a possible candidate.

→ greedy Technique:-

8.5.3 Kruskal's Algorithm

Lect# 36

جب ہم Tree بنا رہے ہوں گے تو کوشش کریں گے کہ cycle نہ بنے
سب سے پہلے جو کم light weight والے ہوں گے ان کو اٹھا کر Tree میں ڈالا جائے گا۔

Kruskal's algorithm works by adding edges in increasing order of weight (lightest edge first). If the next edge does not induce a cycle among the current set of edges, then it is added to A. If it does, we skip it and consider the next in order. As the algorithm runs, the edges in A induce a *forest* on the vertices. The trees of this forest are eventually merged until a single tree forms containing all vertices.

The tricky part of the algorithm is how to detect whether the addition of an edge will create a cycle in A. Suppose the edge being considered has vertices (u, v). We want a fast test that tells us whether u and v are in the same tree of A. This can be done using the *Union-Find* data structure which supports the following $O(\log n)$ operations:

Create-set(u): Create a set containing (Union) a single item u.

Find-set(u): Find the set that contains u

Union(u,v): merge the set containing u and set containing v into a common set.

اگر کسی cycle بننے کی Tension ہو گی تو وہ
اس edge کو ختم کر دیں گے اس کو وہاں رکھیں گے نہیں

In Kruskal's algorithm, the vertices will be stored in sets. The vertices in each tree of A will be a set. The edges in A can be stored as a simple list. Here is the algorithm: Figures 8.51 through ?? demonstrate the algorithm applied to a graph.

```
KRUSKAL(G = (V, E))  → Graph.
1  A ← {}
2  for ( each  u ∈ V)
3  do create_set(u)
4  sort E in increasing order by weight w
5  for ( each (u, v) in sorted edge list)
6  do if (find(u) ≠ find(v))
7        then add (u, v) to A
8             union(u, v)
9  return A
```

جب vertices کی ایک ٹانگ ایک side پہ ہو
اور دوسری ٹانگ دوسری side پر ہو
تو کہہ سکتے ہیں Cross Tree نہیں کریں گے۔

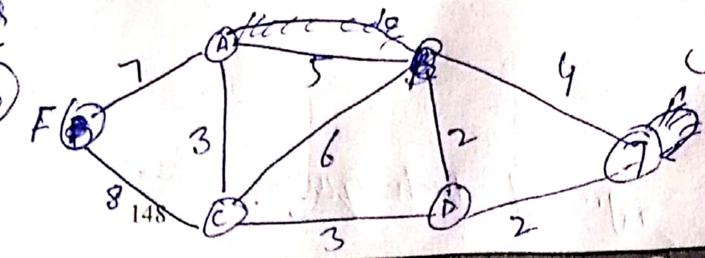
Good! Cos [illegible] 2:30

[illegible]

Graphs

[illegible]

Vertices ⑥



① سب سے پہلے جو گراف آپ کو دیا جاتا ہے اس میں سے Parallel loop کو ختم کریں

② loop itself کو بھی ختم کریں

③ آپ دو loop ختم کریں گے جس کا وزن زیادہ ہو گا

③ del loop and Parallel edges.

④
BD = 2
DE = 2
AC = 3
CD = 3
BE = 4 X
AB = 5 X
BC = 6 X
⑤ AF = 7
FC = 8 X

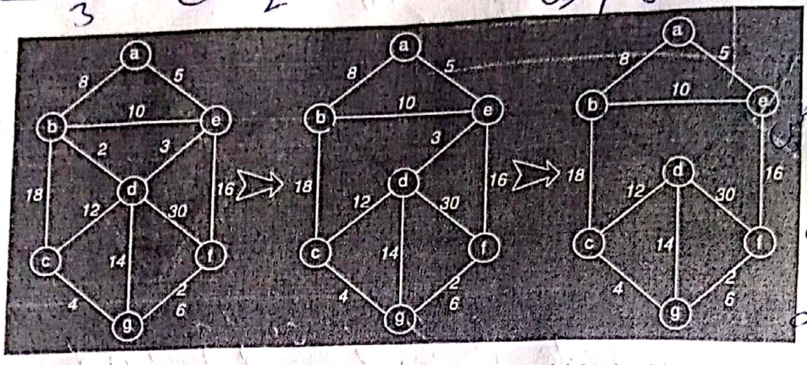


Figure 8.51: Kruskal algorithm: (b, d) and (d, e) added

اس کے بعد ہم نے اس میں سب سے تھوڑا وزن والا Choose کرنا ہے اس طرح سے

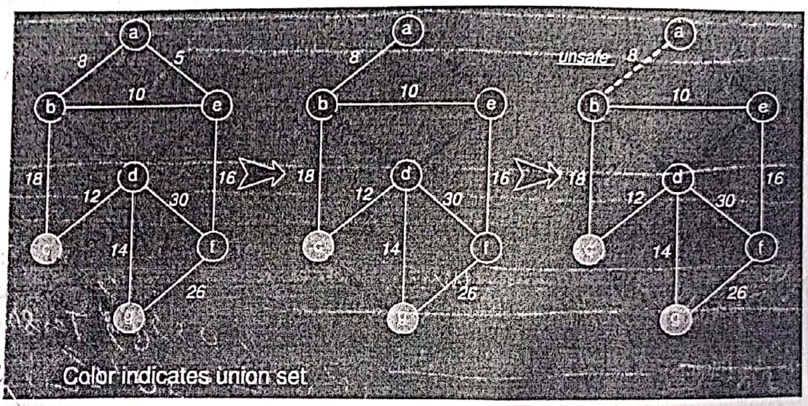


Figure 8.52: Kruskal algorithm: (c, g) and (a, e) added

This graph is also minimum Spanning Tree.

جب آپ نے ان کو دوبارہ ترتیب دے لی ہے تو دیکھیں کہ کوئی cycle نہ بنا سکیں

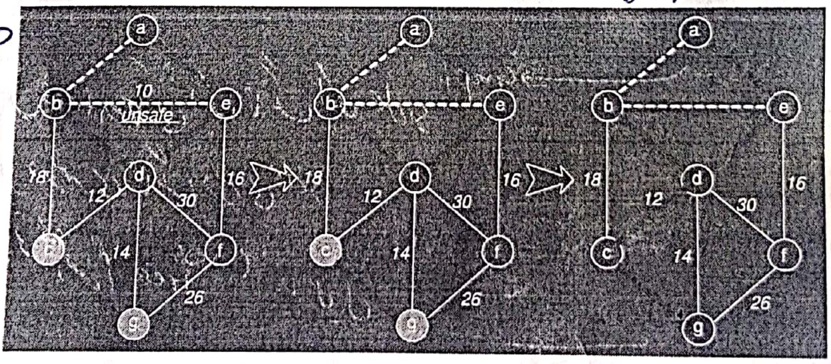
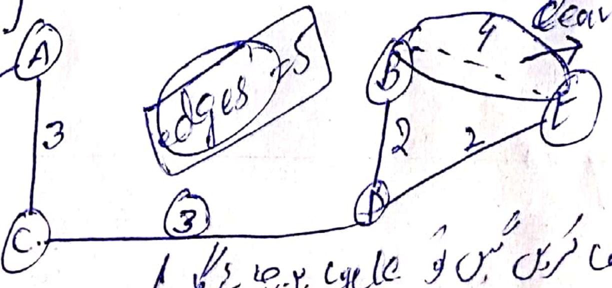


Figure 8.53: Kruskal algorithm: unsafe edges

ان کو ہم نہیں ملائیں گے کیونکہ ان کو ملانے سے cycle بن جاتا
{BE, AB, BC, FC}



آپ اس کو الگ (dotted) line سے بھی connect کر سکتے ہیں کیونکہ اگر solid line سے contact کریں گے تو cycle بن جائے گا

MST میں vertices با تو same ہوں گی، لیکن n-1 edges ہوں گے

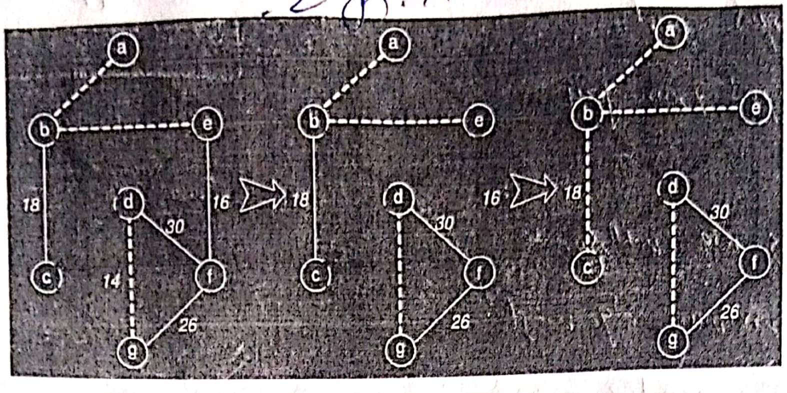


Figure 8.54: Kruskal algorithm: (e, f) added

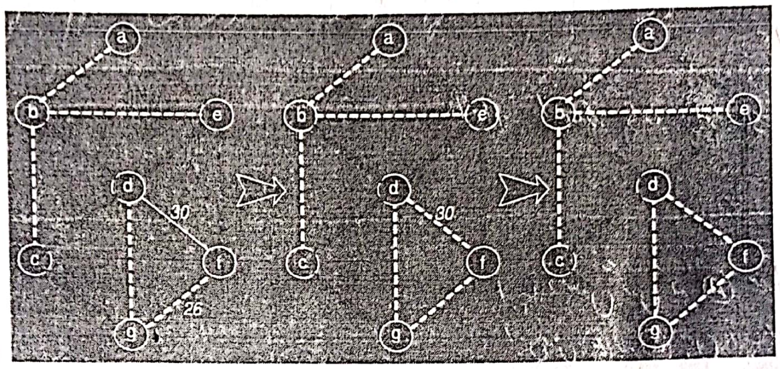


Figure 8.55: Kruskal algorithm: more unsafe edges and final MST

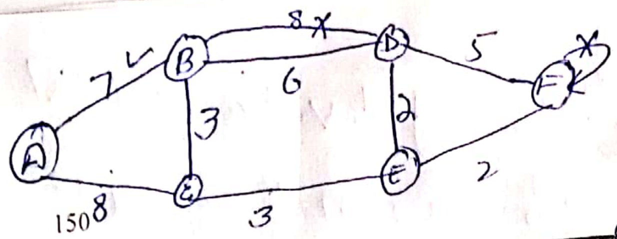
Analysis:

Since the graph is connected, we may assume that $E \geq V - 1$. Sorting edges (*line 4*) takes $\Theta(E \log E)$. The for loop (*line 5*) performs $O(E)$ `find` and $O(V)$ `union` operations. Total time for `union` $-$ `find` is $O(E\alpha(V))$ where $\alpha(V)$ is the inverse Ackerman function. $\alpha(V) < 4$ for V less the number of atoms in the entire universe. Thus the time is dominated by sorting. Overall time for Kruskal is $\Theta(E \log E) = \Theta(E \log V)$ if the graph is sparse.

Lect#37 8.5.4 Prim's Algorithm

Kruskal's algorithm worked by ordering the edges, and inserting them one by one into the spanning tree, taking care never to introduce a cycle. Intuitively Kruskal's works by merging or splicing two trees together, until all the vertices are in the same tree.

In contrast, Prim's algorithm builds the MST by adding leaves one at a time to the current tree. We start with a root vertex r; it can be any vertex. At any time, the subset of edges A forms a single tree (in Kruskal's, it formed a forest). We look to add a single vertex as a leaf to the tree.



⇒ vertices = 6

vertex.

(———) edges.

[illegible]

① Remove loops and parallel edges from your graph.

② By choosing we will remove that have maximum edge weight

③ Choose any ~~[illegible]~~ vertex as head. Note.

④ Suppose Choose (A) as root Note.

G(V,E) → given

after Prim's alg. MST G' is (V', E')

$V' = V \Rightarrow V = 6$

$E' \subset E$.

$E' = V - 1$

$E' = 6 - 1$

$E' = 5$

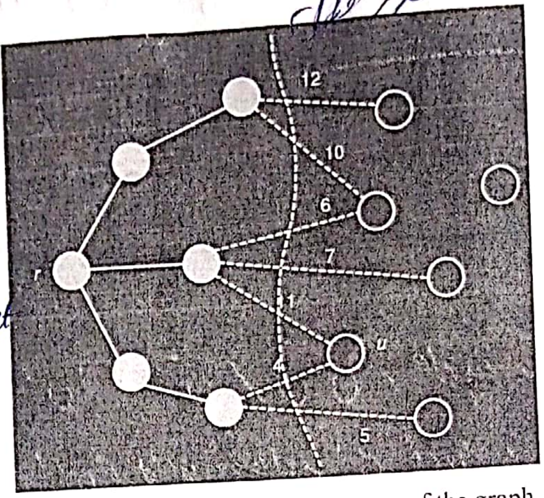
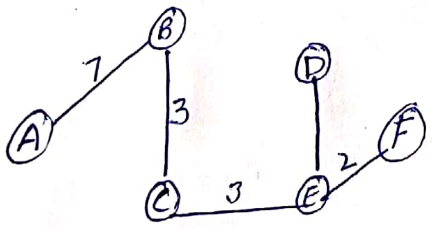


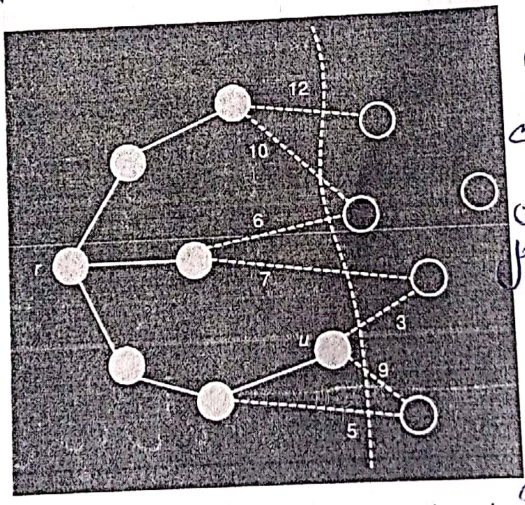
Figure 8.56: Prim's algorithm: a *cut* of the graph

Consider the set of vertices S currently part of the tree and its complement (V – S) as shown in Figure 8.56. We have *cut* of the graph. Which edge should be added next? The greedy strategy would be to add the lightest edge which in the figure is edge to 'u'. Once u is added, Some edges that crossed the cut are no longer crossing it and others that were not crossing the cut are as shown in Figure 8.57

آپ دیکھیں گیں کہ (A) سے کتنے Node منسلک ہیں۔
اس میں (B) اور (C) ہیں آپ اس میں سے Minimum select کریں۔



Vetecees = 6



جب آپ (B) پر پہنچ گیں تو آپ دیکھیں گیں کہ (B) سے کتنے ~~[illegible]~~ vetecees منسلک ہیں۔ آپ نے (A) کو بھی ساتھ شامل کرنا ہے۔

جس edge پر آپ ہوں گے اس سے باقی edges کو بھی دیکھیں گیں ان سب میں جو سب سے چھوٹا ہوا اس کو attach کرتے جائیں گیں

Figure 8.57: Prim's algorithm: u selected

We need an efficient way to update the cut and determine the light edge quickly. To do this, we will make use of a *priority queue*. The question is what do we store in the priority queue? It may seem logical that edges that cross the cut should be stored since we choose light edges from these. Although possible, there is more elegant solution which leads to a simpler algorithm.

جب آپ کے پاس 2 ویلیو same آجائیں تو آپ کوئی بھی choose کر سکتے ہیں۔

can also be used, e.g., time, cost, penalties and loss.

Similar scenarios occur in computer networks like the Internet where data packets have to be routed. Th vertices are *routers*. Edges are communication links which may be be wire or wireless. Edge weights ca be distance, link speed, link capacity link delays, and link utilization.

The breadth-first-search algorithm we discussed earlier is a shortest-path algorithm that works on un-weighted graphs. An un-weighted graph can be considered as a graph in which every edge has weig one unit.

BFS

There are a few variants of the shortest path problem. We will cover their definitions and then discuss algorithms for some.

Single-source shortest-path problem: Find shortest paths from a given (single) *source* vertex $s \in V$
every other vertex $v \in V$ in the graph G.

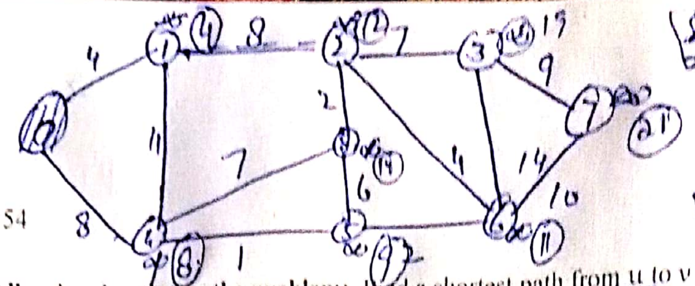
Single-destination shortest-paths problem: Find a shortest path to a given destination vertex t from
each vertex v. We can reduce the this problem to a single-source problem by reversing the direc
of each edge in the graph.

Single-pair shortest-path problem: Find a shortest path from u to v for given vertices u and v. If w
solve the single-source problem with source vertex u, we solve this problem also. No algorithn
for this problem are known to run asymptotically faster than the best single-source algorithms i
the worst case.

35000

14000 → ~~[illegible]~~

lec



Single Source Shortest Problem.

its undirected

All-pairs shortest-paths problem: Find a shortest path from u to v for *every pair* of vertices u and v. Although this problem can be solved by running a single-source algorithm once from each vertex, it can usually be solved faster.

① From one source you are supposed to find the shortest source.

8.6.1 Dijkstra's Algorithm

② Dijkstra's algorithm is a simple *greedy* algorithm for computing the **single-source shortest**-paths to all other vertices. Dijkstra's algorithm works on a weighted directed graph $G = (V, E)$ in which all edge weights are non-negative, i.e., $w(u, v) \geq 0$ for each edge $(u, v) \in E$.

Negative edges weights maybe counter to intuition but this can occur in real life problems. However, we will *not allow negative cycles* because then there is no shortest path. If there is a negative cycle between, say, s and t, then we can always find a shorter path by going around the cycle one more time.

Working principles

① 0 is source vertex.

② 0 to 0 distance is zero. zero to zero

③ We give the name from zero to other vertices as infinity ∞.

④ zero is connected with 1 and 4. Now we will find distance 0 to 1 then we give name zero as u and 1 as v $\Rightarrow d(u) + c(u,v) < d(v)$

$0 + (4) \quad \infty$

$4 < \infty$

⑤ after finding the value from (0-1) and (0-4) then we see the shortest value in whole graph.

⑥ Now ① as source.

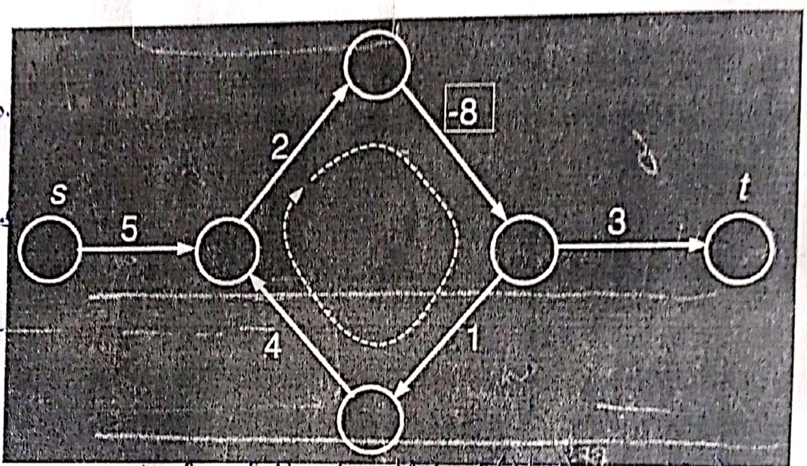


Figure 8.61: Negative weight cycle

lec#3

The basic structure of Dijkstra's algorithm is to maintain an *estimate* of the shortest path from the source vertex to each vertex in the graph. Call this estimate d[v]. Intuitively, d[v] will the length of the shortest path *that the algorithm knows of* from s to v. This value will always be greater than or equal to the true shortest path distance from s to v. I.e., $d[v] \geq \delta(u, v)$. Initially, we know of no paths, so $d[v] = \infty$. Moreover, $d[s] = 0$ for the source vertex.

As the algorithm goes on and sees more and more vertices, it attempts to update d[v] for each vertex in the graph. The process of updating estimates is called *relaxation*. Here is how relaxation works.

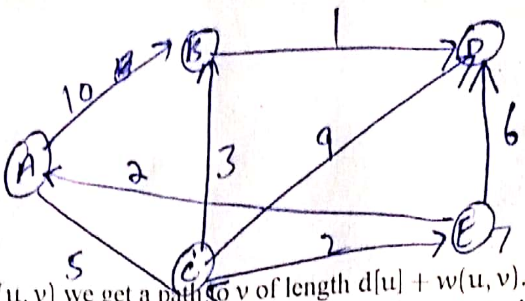
Intuitively, if you can see that your solution is not yet reached an optimum value, then push it a little closer to the optimum. In particular, if you discover a path from s to v shorter than d[v], then you need to update d[v]. This notion is common to many optimization algorithms.

Consider an edge from a vertex u to v whose weight is $w(u, v)$. Suppose that we have already computed current estimates on d[u] and d[v]. We know that there is a path from s to u of weight d[u]. By taking

$$\left\{ \begin{array}{l} d(u) + c(u,v) < dv \\ d(v) = d(u) + c(u,v) \end{array} \right\}$$

Formula.

A is source vertex.



this path and following it with the edge (u, v) we get a path to v of length $d[u] + w(u, v)$. If this path is better than the existing path of length $d[v]$ to v, we should take it. The relaxation process is illustrated in the following figure. We should also remember that the shortest way back to the source is through u by updating the predecessor pointer.

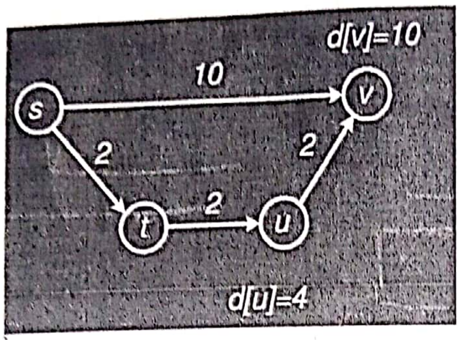


Figure 8.62: Vertex u relaxed

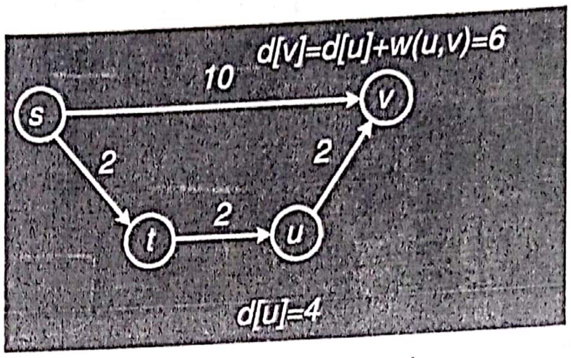


Figure 8.63: Vertex v relaxed

```
RELAX((u, v))
1  if (d[u] + w(u, v) < d[v])
2     then d[v] ← d[u] + w(u, v)
3          pred[v] = u
```

Observe that whenever we set $d[v]$ to a finite value, there is always evidence of a path of that length. Therefore $d[v] \geq \delta(s, v)$. If $d[v] = \delta(s, v)$, then further relaxations cannot change its value.

It is not hard to see that if we perform RELAX(U,V) repeatedly over all edges of the graph, the $d[v]$ values will eventually converge to the final true distance value from s. The cleverness of any shortest path algorithm is to perform the updates in a judicious manner, so the convergence is as fast as possible.

Dijkstra's algorithm is based on the notion of performing repeated relaxations. The algorithm operates by maintaining a subset of vertices, $S \subseteq V$, for which we claim we *know* the true distance, $d[v] = \delta(s, v)$.

Initially $S = \emptyset$, the empty set. We set $d[u] = 0$ and all others to ∞. One by one we select vertices from $V - S$ to add to S.

Source ?. vertex [illegible]

How do we select which vertex among the vertices of $V - S$ to add next to S? Here is *greediness* comes in. For each vertex $u \in (V - S)$, we have computed a distance estimate $d[u]$.

The greedy thing to do is to take the vertex for which $d[u]$ is minimum, i.e., take the unprocessed vertex that is closest by our estimate to s. Later, we justify why this is the proper choice. In order to perform

	A	B	C	D	E
A	$\boxed{0}$	∞	$\infty \leftarrow$	∞	∞ ⇒ minimum
C		$10 \leftarrow$	$\boxed{5}$	∞	∞
E		$8 \leftarrow$		14	$\boxed{7}$
B		$\boxed{8} \leftarrow$ [illegible]		13 ← [illegible] Change	
D				$\boxed{9} \leftarrow$	

$\boxed{\text{distance}}$ $d(u) + c(u,v) < d(v)$

$0 + 10 < \infty$

The shortes distance between is $(A-B) = 8$

$(A-C) = 5$

$(A-D) = 9$

$A-E = 7$

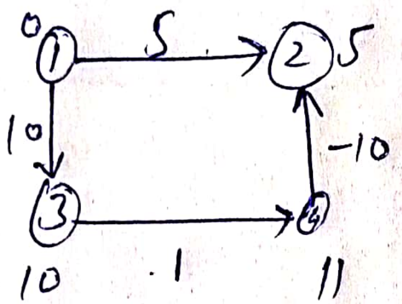
$\boxed{\text{Path}}$ A – D

D B C A

A C B D

$= 5 + 3 + 1 = 9$

Figures 8.64 through ?? demonstrate the algorithm applied to a directed graph with no negative weight edges.



According to dijkstra Algorith, we cannot change ② because we already solved.

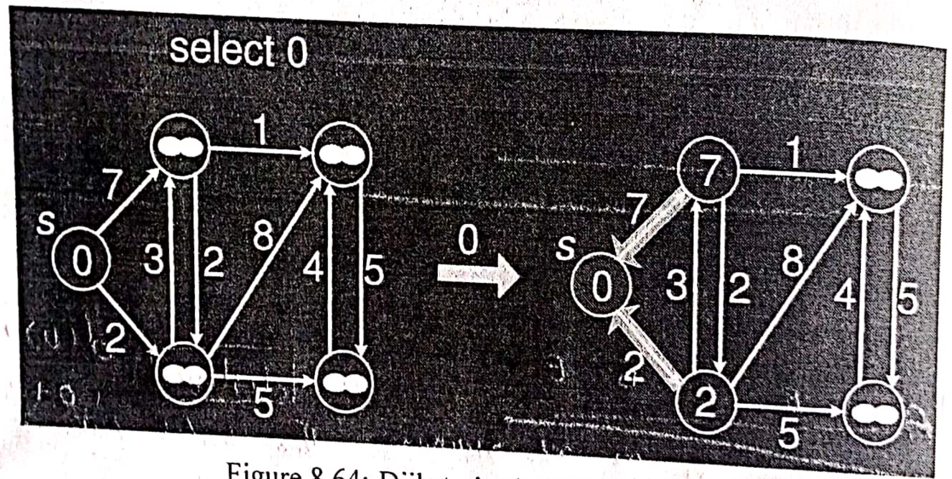


Figure 8.64: Dijkstra's algorithm: select 0

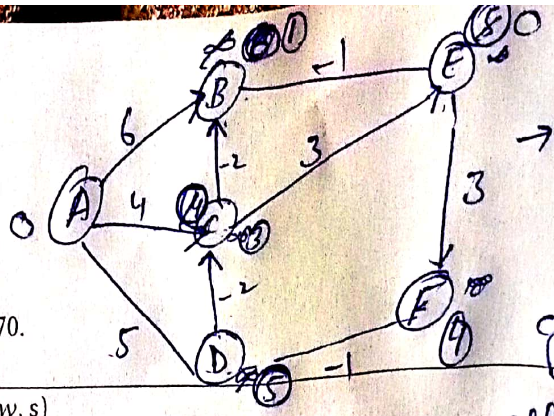
→ It may or may not be work with when side is negitive

→ also it will not work if graph is many cycle is -ve.

8.6.3 Bellman-Ford Algorithm Single Source Shortest Path

Dijkstra's single-source shortest path algorithm works if all edges weights are non-negative and there are no negative cost cycles. Bellman-Ford *allows* negative weights edges and no negative cost cycles. The algorithm is slower than Dijkstra's, running in $\Theta(VE)$ time.

Like Dijkstra's algorithm, Bellman-Ford is based on performing repeated relaxations. Bellman-Ford applies relaxation to *every edge* of the graph and repeats this $V - 1$ times. Here is the algorithm; its is



→ directed graph.

illustrated in Figure 8.70.

```
BELLMAN-FORD(G, w, s)
1  for ( each u ∈ V)
2  do d[u] ← ∞
3      pred[u] = nil
4
5  d[s] ← 0;
6  for i = 1 to V − 1
7  do for ( each (u, v) in E )
8      do RELAX(u, v)
```

NO of vertices ⇒ 6

go on relaxing all the edges (n−1) Times:

n = number of vertices

$\boxed{\text{if } d(u)+c(u,v) < d(v) \quad d(v)=d(u)+c(u,v)}$

A as source vertex.

edges.

(A,B), (A,C), (A,D), (B,E)

(C,E), (D,C), (D,E), (E,F)

(C,B)

1st iteration ✓

2nd iteration.

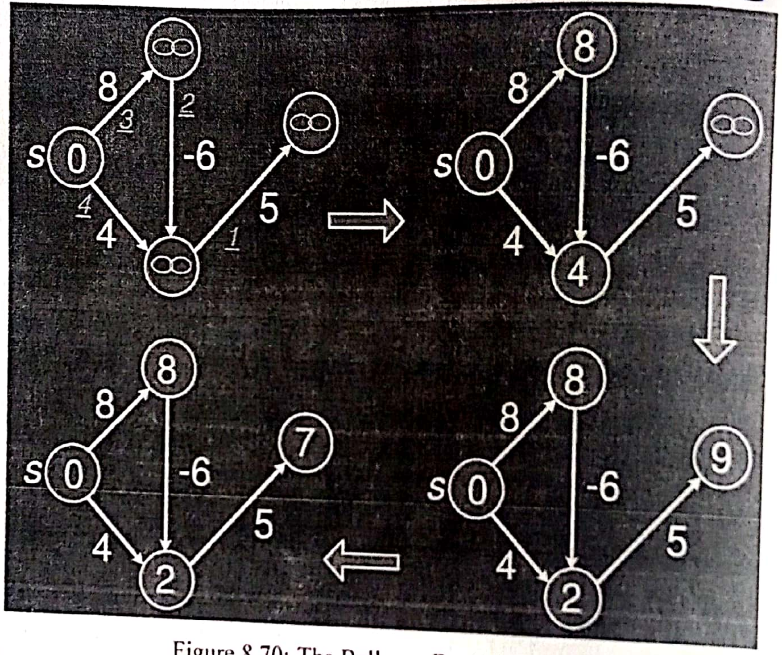


Figure 8.70: The Bellman-Ford algorithm

اگر Total (Vertices) 6 ہوں تو آپ انہیں 5 دفعہ calculate کرتا ہے۔ مطلب 5 دفعہ (iteration) نکالنی ہے۔

Lect# 41

8.6.4 Correctness of Bellman-Ford

Think of Bellman-Ford as a sort of bubble-sort analog for shortest path. The shortest path information propagated sequentially along each shortest path in the graph. Consider any shortest path from s to some other vertex u: $\langle v_0, v_1, \ldots, v_k \rangle$ where $v_0 = s$ and $v_k = u$.

Since a shortest path will never visit the same vertex twice, we know that $k \leq V - 1$. Hence the path consists of at most $V - 1$ edges. Since this a shortest path, it is $\delta(s, v_i)$, the true shortest path cost

ہر (iteration) میں آپ نے وہی پرانا طریقہ لگانا ہے اور فارمولا لگاتے جانا ہے۔

جب تک فارمولا لگتا نوڈز کو بار بار iteration سے change کرنا ہے

جب (iteration) ختم ہو جائے تو.

Vertices

A	= 0
B	= 1
C	= 3
D	= 5
E	= 0
F	= 3

$O(E(|V|-1))$

$O(E.V)$ Time complexity

$O(n^2)$

to v_i that satisfies the equation:

Time Complexity $= \frac{V(V-1)}{2} \times (V-1)$

$$\delta(s, v_i) = \delta(s, v_{i-1}) + w(v_{i-1}, v_i)$$

Claim: $\boxed{\text{We assert that after the } i^{th} \text{ pass of the "for-i" loop, } d[v_i] = \delta(s, v_i)}$

Proof: The proof is by induction on i. Observe that after the initialization (pass 0), $d[v_1] = d[s] = 0$.

In general, prior to the i^{th} pass through the loop, the induction hypothesis tells us that $d[v_{i-1}] = \delta(s, v_{i-1})$. After the i^{th} pass, we have done relaxation on the edge (v_{i-1}, v_i) (since we do relaxation along all edges). Thus after the i^{th} pass we have

(It will not work if it have -ve weight cycle. whose edge sum is -ve)

$$\begin{aligned} d[v_i] &\le d[v_{i-1}] + w(v_{i-1}, v_i) \\ &= \delta(s, v_{i-1}) + w(v_{i-1}, v_i) \\ &= \delta(s, v_i) \end{aligned}$$

جتنی Vertices ہوں گے No. of iteration اس سے ایک کم ہوں گی

Recall from Dijkstra's algorithm that $d[v_i]$ is never less than $\delta(s, v_i)$. Thus, $d[v_i]$ is in fact equal to $\delta(s, v_i)$. This completes the induction proof.

In summary, after i passes through the for loop, all vertices that are i edges away along the shortest path tree from the source have the correct values stored in d[u]. Thus, after the $(V-1)^{st}$ iteration of the for loop, all vertices u have correct distance values stored in d[u].

8.6.5 Floyd-Warshall Algorithm — dynamic Programming

We consider the generalization of the shortest path problem: to compute the shortest paths between all pairs of vertices. This is called the all-pairs shortest paths problem.

Let $G = (V, E)$ be a directed graph with edge weights. If $(u, v) \in E$ is an edge then $w(u, v)$ denotes its weight. $\delta(u, v)$ is the distance of the minimum cost path between u and v. We will allow G to have negative edges weights but will not allow G to have negative cost cycles. We will present an $\Theta(n^3)$ algorithm for the all pairs shortest path. The algorithm is called the *Floyd-Warshall algorithm* and is based on *dynamic programming*.

We will use an adjacency matrix to represent the digraph. Because the algorithm is matrix based, we will employ the common matrix notation, using i, j and k to denote vertices rather than u, v and w.

The input is an $n \times n$ matrix of edge weights:

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{if } i \ne j \text{ and } (i,j) \in E \\ \infty & \text{if } i \ne j \text{ and } (i,j) \notin E \end{cases}$$

The output will be an $n \times n$ distance matrix $D = d_{ij}$, where $d_{ij} = \delta(i, j)$, the shortest path cost from vertex i to j.

جتنے (vertices) ہوں گیں اتنے ہی (A^0) کی value نکلے گی

Lect# 42

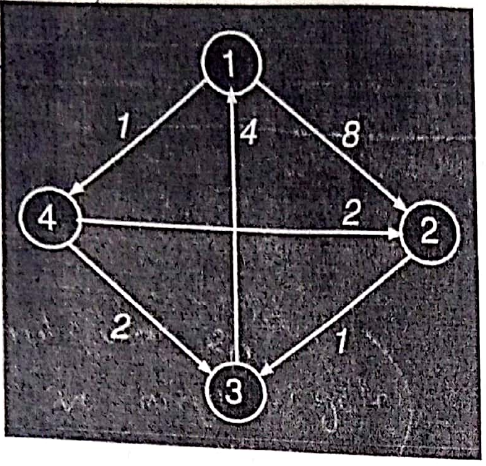


Figure 8.72: k = 0, $d_{3,2}^{(0)} = \infty$ (no path)

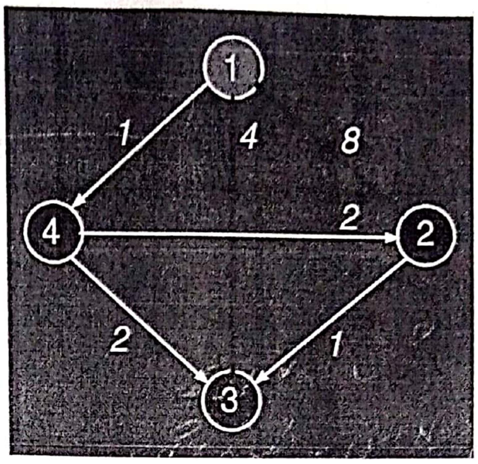


Figure 8.73: k = 1, $d_{3,2}^{(1)} = 12$ $(3 \to 1 \to 2)$

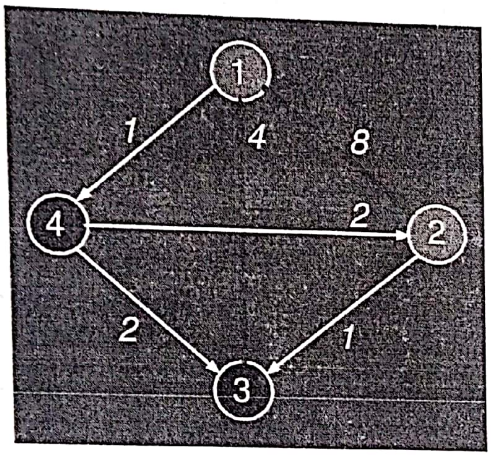


Figure 8.74: k = 2, $d_{3,2}^{(2)} = 12$ $(3 \to 1 \to 2)$

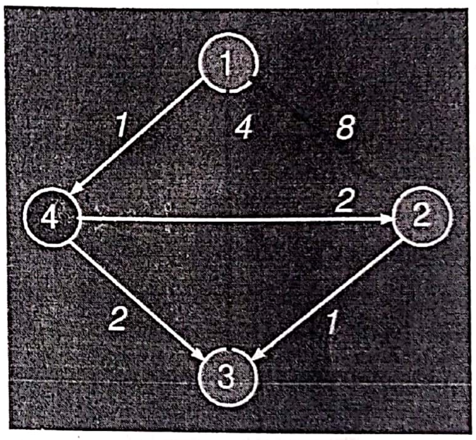
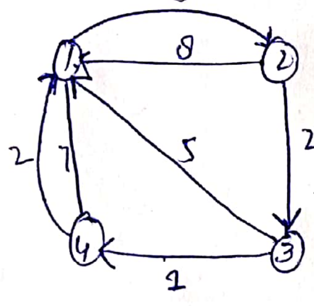


Figure 8.75: k = 3, $d_{3,2}^{(3)} = 12$ $(3 \to 1 \to 2)$



اس کیس میں (4 Verticest) ہیں تو A^4 نکالیں گیں

★ A_1 نکالتے وقت پہلی Row اور پہلا کالم بالکل Same as A^0 ہی ہوگی

★ A_1 کی باقی ویلیوز A^0 سے $\overset{2}{A}$ کی ویلیوز A^1 سے Backward Technique کہلاتی ہے۔

$$A^0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 7 \\ 2 & 8 & 0 & 2 & \infty \\ 3 & 5 & \infty & 0 & 1 \\ 4 & 2 & \infty & \infty & 0 \end{array} \Rightarrow A^1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 7 \\ 2 & 8 & 0 & \boxed{2} & 15 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & \infty & 0 \end{array}$$

$A^0[2,4] = A^0[\underline{2,1} + 1,4]$

$\infty = 8 + 7$

$\infty > 15$

$A^0[3,2] = A^0[3,1] + [1,2]$

$\infty = 5 +$

آپکو $A^1(2\text{-}3)$ میں یہ قیمت نکالنی ہے تو آپ A^0 میں جائیں گیں:

$A^0[2,3] = A^0[2,1 + 1,3]$

$2 = 8 + \infty$, 3 چھوٹی ہوگی

$A^2 =$

Chapter 9

Lect #43.

~~C~~omplexity Theory

BY:- MUDASSAR Iqbal 0340-6829977

So far in the course, we have been building up a "bag of tricks" for solving algorithmic problems. Hopefully you have a better idea of how to go about solving such problems. What sort of design paradigm should be used: divide-and-conquer, greedy, dynamic programming.

What sort of data structures might be relevant: trees, heaps, graphs. What is the running time of the algorithm. All of this is fine if it helps you discover an acceptably efficient algorithm to solve your problem.

The question that often arises in practice is that you have tried every trick in the book and nothing seems to work. Although your algorithm can solve small problems reasonably efficiently (e.g., $n \leq 20$), for the really large problems you want to solve, your algorithm never terminates. When you analyze its running time, you realize that it is running in exponential time, perhaps $n^{\sqrt{n}}$, or 2^n, or 2^{2^n}, or $n!$ or worse!.

By the end of 60's, there was great success in finding efficient solutions to many combinatorial problems. But there was also a growing list of problems for which there seemed to be no known efficient algorithmic solutions.

People began to wonder whether there was some unknown paradigm $\overset{\text{example:-}}{}$ that would lead to a solution to there problems. Or perhaps some proof that these problems are inherently hard to solve and no algorithmic solutions exist that run under exponential time.

Near the end of the 1960's, a remarkable discovery was made. Many of these hard problems were interrelated in the sense that if you could solve any one of them in polynomial time, then you could solve all of them in polynomial time. this discovery gave rise to the notion of NP-completeness.

This area is a radical departure from what we have been doing because the emphasis will change. The goal is no longer to prove that a problem *can* be solved efficiently by presenting an algorithm for it. Instead we will be trying to show that a problem *cannot* be solved efficiently.

Up until now all algorithms we have seen had the property that their worst-case running time are bounded above by some *polynomial* in n. A *polynomial time algorithm* is any algorithm that runs in $O(n^k)$ time. A problem is solvable in polynomial time if there is a polynomial time algorithm for it.

Some functions that do not look like polynomials (such as $O(n \log n)$ are bounded above by polynomials (such as $O(n^2)$). Some functions that do look like polynomials are not. For example, suppose you have an algorithm that takes as input a graph of size n and an integer k and run in $O(n^k)$ time. Is this a polynomial time algorithm? No, because k is an input to the problem so the user is allowed to choose $k = n$, implying that the running time would be $O(n^n)$. $O(n^n)$ is surely not a polynomial in n. The important aspect is that the exponent must be a constant independent of n.

ایک مسئلہ اس وقت فیصلہ کن مسئلہ ہوگی اگر اس کی Yes output یا Not ہو

9.1 Decision Problems

Most of the problems we have discussed involve optimization of one form of another. Find the shortest path, find the minimum cost spanning tree, maximize the knapsack value. For rather technical reasons, the NP-complete problems we will discuss will be phrased as *decision problems*.

A problem is called a *decision problem* if its output is a simple "yes" or "no" (or you may this of this as true/false, 0/1, accept/reject.) We will phrase may optimization problems as decision problems. For example, the MST decision problem would be: Given a weighted graph G and an integer k, does G have a spanning tree whose weight is at most k?

This may seem like a less interesting formulation of the problem. It does not ask for the weight of the minimum spanning tree, and it does not even ask for the edges of the spanning tree that achieves this weight. However, our job will be to show that certain problems cannot be solved efficiently. If we show that the simple decision problem cannot be solved e¡ciently, then the more general optimization problem certainly cannot be solved efficiently either.

9.2 Complexity Classes

Before giving all the technical definitions, let us say a bit about what the general classes look like at an intuitive level.

Solved **Class P:** This is the set of all decision problems that can be *solved* in polynomial time. We will generally refer to these problems as being "easy" or "efficiently solvable".

Varific **Class NP:** This is the set of all decision problems that can be *verified* in polynomial time. This class contains P as a subset. It also contains a number of problems that are believed to be very " hard" to solve.

Class NP: The term "NP" does not mean "not polynomial". Originally, the term meant " non-deterministic polynomial" but it is a bit more intuitive to explain the concept from the perspective of verification.

Class NP-hard: In spite of its name, to say that a problem is NP-hard does not mean that it is hard to solve. Rather, it means that if we could solve this problem in polynomial time, then we could solve *all* NP problems in polynomial time. Note that for a problem to NP-hard, it does not have to be in the class NP.

اگر ہم NP Hard کو Polynomonial Hard کے اندر حل کر دیں تو ہم تمام NP Problem کو حل کر سکتے ہیں۔

Class NP-complete: A problem is NP-complete if (1) it is in NP and (2) it is NP-hard.

The Figure 9.1 illustrates one way that the sets P, NP, NP-hard, and NP-complete (NPC) might look. We say might because we do not know whether all of these complexity classes are distinct or whether they are all solvable in polynomial time. The Graph Isomorphism, which asks whether two graphs are identical up to a renaming of their vertices. It is known that this problem is in NP, but it is not known to be in P. The other is QBF, which stands for Quantified Boolean Formulas. In this problem you are given a boolean formula and you want to know whether the formula is true or false.

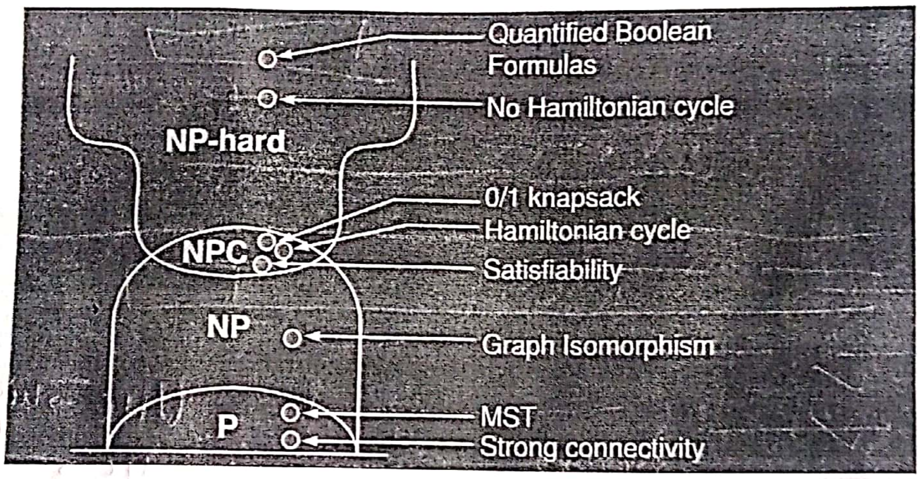


Figure 9.1: Complexity Classes

9.3 Polynomial Time Verification

Before talking about the class of NP-complete problems, it is important to introduce the notion of a *verification algorithm*. Many problems are hard to solve but they have the property that it easy to verify the solution if one is provided. Consider the Hamiltonian cycle problem.

Given an undirected graph G, does G have a cycle that visits every vertex exactly once? There is no known polynomial time algorithm for this problem.

معلوم کرنا کہ وہ صحیح ہیں verification کہا جاتا ہے

certifica

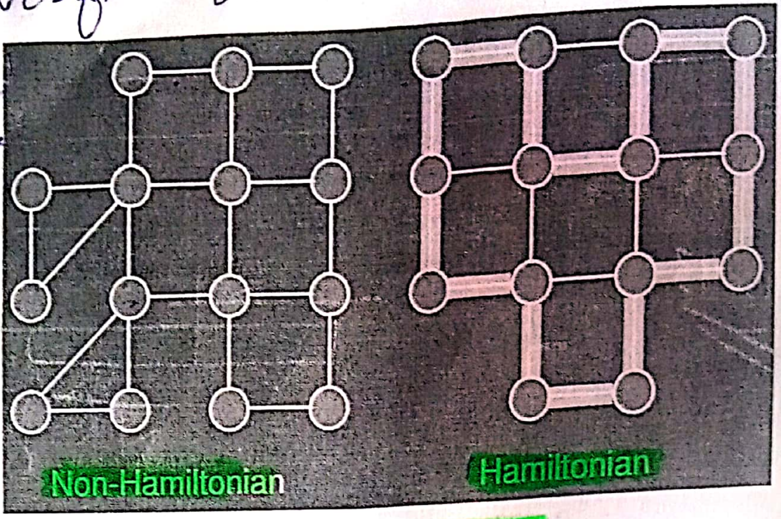


Figure 9.2: Hamiltonian Cycle

However, suppose that a graph did have a Hamiltonian cycle. It would be easy for someone to convince of this. They would simply say: "the cycle is $\langle v_3, v_7, v_1, \ldots, v_{13}\rangle$" We could then inspect the graph and check that this is indeed a legal cycle and that it visits all of the vertices of the graph exactly once. Thus, even though we know of no efficient way to *solve* the Hamiltonian cycle problem, there is a very efficient way to *verify* that a a given cycle is indeed a Hamiltonian cycle.

The piece of information that allows verification is called a *certificate*. Note that not all problems have the property that they are easy to verify. For example, consider the following two:

1. UHC = {⟨G⟩|G has a unique Hamiltonian cycle}
2. $\overline{\text{HC}}$ = {⟨G⟩|G has no Hamiltonian cycle}

UHC → unique Hamilton- cycle

HC →

Suppose that a graph G is in UHC. What information would someone give us that would allow us to verify this? They could give us an example of the unique Hamiltonian cycle and we could verify that it is a Hamiltonian cycle. But what sort of certificate could they give us to convince us that this is the *only* one?

They could give another cycle that is *not* Hamiltonian. But this does not mean that there is not another cycle somewhere that is Hamiltonian. They could try to list every other cycle of length n, but this is not efficient at all since there are n! possible cycles in general. Thus it is hard to imagine that someone could give us some information that would allow us to efficiently verify that the graph is in $\overline{\text{UHC}}$.

9.4 The Class NP

The class NP is a set of all problems that can be verified by a polynomial time algorithm. Why is the set called "NP" and not "VP"? The original term NP stood for *non-deterministic polynomial time*. This referred to a program running on a non-deterministic computer that can make guesses. Such a computer could non-deterministically guess the value of the certificate, and then verify it in polynomial time. We have avoided introducing non-determinism here; it is covered in other courses such as automata or complexity theory.

Observe that P ⊆ NP. In other words, if we can solve a problem in polynomial time, we can certainly verify the solution in polynomial time. More formally, we do not need to see a certificate to solve the problem; we can solve it in polynomial time anyway.

However, it is not known whether P = NP. It seems unreasonable to think that this should be so. Being able to verify that you have a correct solution does not help you in finding the actual solution. The belief is that P ≠ NP but no one has a proof for this.

9.5 Reductions

P⊆NP

NPC → Non-deterministic Polinomial complete

The class NP-complete (NPC) problems consists of a set of decision problems (a subset of class NP) that no one knows how to solve efficiently. But if there were a polynomial solution for even a single NP-complete problem, then ever problem in NPC will be solvable in polynomial time. For this, we need the concept of *reductions*.

Consider the question: Suppose there are two problems, A and B. You know (or you strongly believe at least) that it is impossible to solve problem A in polynomial time. You want to prove that B cannot be solved in polynomial time. We want to show that

$$(A \notin P) \Rightarrow (B \notin P)$$

How would you do this? Consider an example to illustrate reduction: The following problem is well-known to be NPC:

3-color: Given a graph G, can each of its vertices be labelled with one of 3 different colors such that two adjacent vertices have the same label (color).

Coloring arises in various partitioning problems where there is a constraint that two objects cannot be assigned to the same set of partitions. The term "coloring" comes from the original application which was in map drawing. Two countries that share a common border should be colored with different colors.

It is well known that planar graphs can be colored (maps) with *four colors*. There exists a polynomial time algorithm for this. But determining whether this can be done with 3 colors is hard and there is no polynomial time algorithm for it. In Figure 9.3, the graph on the left can be colored with 3 colors while the graph on the right cannot be colored.

Definition: L is NP-complete if

1. $L \in NP$ and
2. $L' \leq_P L$ for some known NP-complete problem L'.

Given this formal definition, the complexity classes are:

P: is the set of decision problems that are solvable in polynomial time.

NP: is the set of decision problems that can be verified in polynomial time.

NP-Hard: L is NP-hard if for all $L' \in NP$, $L' \leq_P L$. Thus if we could solve L in polynomial time, we could solve all NP problems in polynomial time.

NP-Complete L is NP-complete if

1. $L \in NP$ and
2. L is NP-hard.

The importance of NP-complete problems should now be clear. If any NP-complete problem is solvable in polynomial time, then every NP-complete problem is also solvable in polynomial time. Conversely, if we can prove that any NP-complete problem cannot be solved in polynomial time, the every NP-complete problem cannot be solvable in polynomial time.

9.8 Boolean Satisfiability Problem: $\boxed{\text{Cook's Theorem}}$

We need to have at least one NP-complete problem to start the ball rolling. Stephen Cook showed that such a problem existed. He proved that the *boolean satisfiability problem* is NP-complete. A boolean formula is a logical formulation which consists of variables x_i. These variables appear in a logical expression using logical operations

1. negation of x: $\overline{x}$
2. boolean or: $(x \vee y)$
3. boolean and: $(x \wedge y)$

$\vee \rightarrow$ or
$\wedge \rightarrow$ AND

For a problem to be in NP, it must have an efficient verification procedure. Thus virtually all NP problems can be stated in the form, "does there exists X such that P(X)", where X is some structure (e.g. a set, a path, a partition, an assignment, etc.) and P(X) is some property that X must satisfy (e.g. the set of objects must fill the knapsack, or the path must visit every vertex, or you may use at most k colors and no two adjacent vertices can have the same color). In showing that such a problem is in NP, the certificate consists of giving X, and the verification involves testing that P(X) holds.

In general, any set X can be described by choosing a set of objects, which in turn can be described as choosing the values of some boolean variables. Similarly, the property P(X) that you need to satisfy, can be described as a boolean formula. Stephen Cook was looking for the most general possible property he could, since this should represent the hardest problem in NP to solve. He reasoned that computers (which represent the most general type of computational devices known) could be described entirely in terms of boolean circuits, and hence in terms of boolean formulas. If any problem were hard to solve, it would be one in which X is an assignment of boolean values (true/false, 0/1) and P(X) could be any boolean formula. This suggests the following problem, called the *boolean satisfiability problem*.

SAT: Given a boolean formula, is there some way to assign truth values (0/1, true/false) to the variables of the formula, so that the formula evaluates to true?

A boolean formula is a logical formula which consists of variables x_i, and the logical operations $\overline{x}$ meaning the *negation* of x, *boolean-or* $(x \vee y)$ and *boolean-and* $(x \wedge y)$. Given a boolean formula, we say that it is satisfiable if there is a way to assign truth values (0 or 1) to the variables such that the final result is 1. (As opposed to the case where no matter how you assign truth values the result is always 0.) For example

$$(x_1 \wedge (x_2 \vee \overline{x_3})) \wedge ((\overline{x_2} \wedge \overline{x_3}) \vee \overline{x_1})$$

is satisfiable, by the assignment $x_1 = 1$, $x_2 = 0$ and $x_3 = 0$. On the other hand,

$$(\overline{x_1} \vee (x_2 \wedge x_3)) \wedge (x_1 \vee (\overline{x_2} \wedge \overline{x_3})) \wedge (x_2 \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})$$

is not satisfiable. Such a boolean formula can be represented by a logical circuit made up of OR, AND and NOT gates. For example, Figure 9.9 shows the circuit for the boolean formula

$$((x_1 \wedge x_4) \vee x_2) \wedge ((x_3 \wedge \overline{x_4}) \vee \overline{x_2}) \wedge \overline{x_5}$$

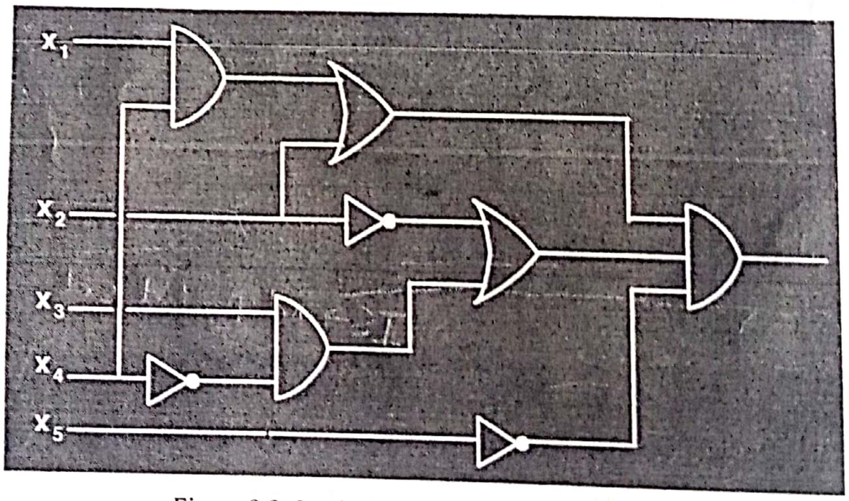


Figure 9.9: Logical circuit for a boolean formula

10th class Physics
ch#16
Read

To clasify this
Topic

3 CNF → Form.
↓ → Normal
Conjuctive

Bolean Satisfaction Problem is Np complete.

Cook's Theorem: SAT is NP-complete

We will not prove the theorem; it is quite complicated. In fact, it turns out that a even more restricted version of the satisfiability problem is NP-complete.

A *literal* is a variable x or its negation $\overline{x}$. A boolean formula is in *3-Conjunctive Normal Form* (3-CNF) if it is the boolean-and of clauses where each clause is the boolean-or of exactly three literals. For example,

$$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_3 \vee x_4) \wedge (x_2 \vee \overline{x_3} \vee x_4)$$

is in 3-CNF form. 3SAT is the problem of determining whether a formula is 3-CNF is satisfiable. 3SAT is NP-complete. We can use this fact to prove that other problems are NP-complete. We will do this with the *independent set problem*.

Independent Set Problem: Given an undirected graph $G = (V, E)$ and an integer k, does G contain a subset V' of k vertices such that no two vertices in V' are adjacent to each other.

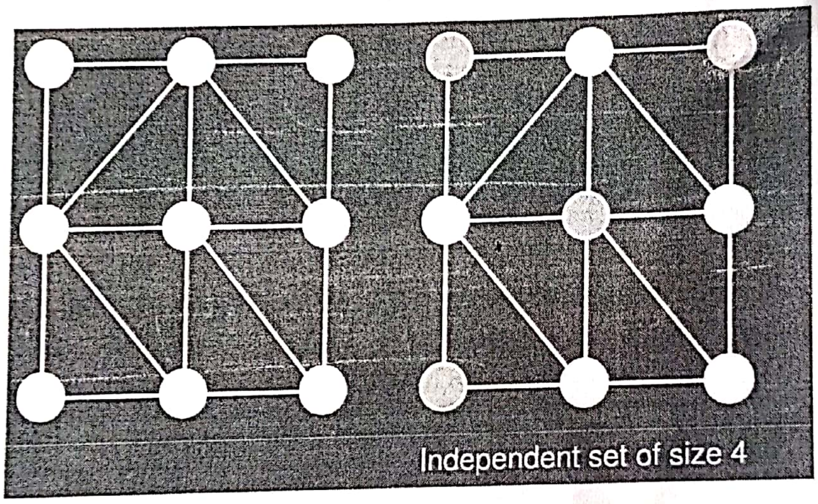


Figure 9.10:

The independent set problem arises when there is some sort of selection problem where there are mutual restrictions pairs that cannot both be selected. For example, a company dinner where an employee and his or her immediate supervisor cannot both be invited.

Claim: IS is NP-complete

IS → independent set.

The proof involves two parts. First, we need to show that IS ∈ NP. The certificate consists of k vertices of V'. We simply verify that for each pair of vertices $u, v \in V'$, there is no edge between them. Clearly, this can be done in polynomial time, by an inspection of the adjacency matrix.

Second, we need to establish that IS is NP-hard This can be done by showing that some known NP-compete (3SAT) is polynomial-time reducible to IS. That is, $3SAT \leq_P IS$.

An important aspect to reductions is that we do not attempt to solve the satisfiability problem. Remember: It is NP-complete, and there is not likely to be any polynomial time solution. The idea is to translate the similar elements of the satisfiable problem to corresponding elements of the independent set problem.

What is to be selected?

3SAT: Which variables are to be assigned the value true, or equivalently, which literals will be true and which will be false.

IS: Which vertices will be placed in V'.

Requirements:

3SAT: Each clause must contain at least one true valued literal.

IS: V' must contain at least k vertices.

Restrictions:

3SAT: If x_i is assigned true, then $\overline{x_i}$ must be false and vice versa.

IS: If u is selected to be in V' and v is a neighbor of u then v cannot be in V'.

We want a function which given any 3-CNF boolean formula F, converts it into a pair (G, k) such that the above elements are translated properly. Our strategy will be to turn each literal into a vertex. The vertices will be in clause clusters of three, one for each clause. Selecting a true literal from some clause will correspond to selecting a vertex to add to V'. We will set k equal to the number of clauses, to force the independent set subroutine to select one true literal from each clause. To keep the IS subroutine from selecting two literals from one clause and none from some other, we will connect all the vertices in each clause cluster with edges. To keep the IS subroutine from selecting a literal and its complement to be true, we will put an edge between each literal and its complement.

A formal description of the reduction is given below. The input is a boolean formula F in 3-CNF, and the output is a graph G and integer k.

```
3SAT-TO-IS(F)
 1  k ← number of clauses in F                    ✓
 2  for ( each clause C in F )                     ✓
 3  do create a clause cluster of                  ✓
 4       3 vertices from literals of C             ✓
 5  for ( each clause cluster ($x_1, x_2, x_3$) )  ✓
 6  do create an edge ($x_i, x_j$) between
 7       all pairs of vertices in the cluster      ✓
 8  for ( each vertex $x_i$ )
 9  do create an edge between $x_i$ and            ✓
10       all its complement vertices $\overline{x_i}$
```

```
11  return (G, k)   // output is graph G and integer k
```

If F has k clauses, then G has exactly 3k vertices. Given any reasonable encoding of F, it is an easy programming exercise to create G (say as an adjacency matrix) in polynomial time. We claim that F is satisfiable if and only if G has an independent set of size k.

Example: Suppose that we are given the 3-CNF formula:

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee x_3)$$

The following series of figures show the reduction which produces the graph and sets $k = 4$. First, each of the four literals is converted into a three-vertices graph. This is shown in Figure 9.11

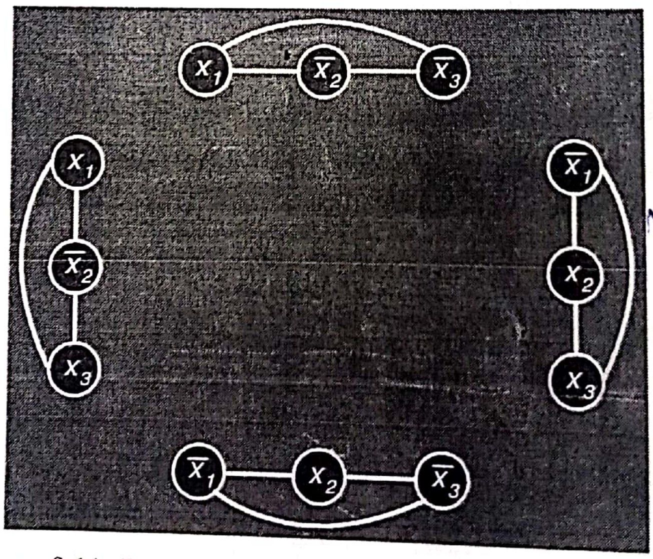


Figure 9.11: Four graphs, one for each of the 3-terms literal

Next, each term is connected to its complement. This is shown in Figure 9.12.

9.9 Coping with NP-Completeness

With NP-completeness we have seen that there are many important optimization problems that a to be quite hard to solve exactly. Since these are important problems, we cannot simply give up a point, since people do need solutions to these problems. Here are some strategies that are used to with NP-completeness:

Use brute-force search: Even on the fastest parallel computers this approach is viable only for smallest instance of these problems.

Heuristics: A heuristic is a strategy for producing a valid solution but there are no guarantees ho it to optimal. This is worthwhile if all else fails.

General search methods: Powerful techniques for solving general combinatorial optimization problems. Branch-and-bound, A*-search, simulated annealing, and genetic algorithms

Approximation algorithm: This is an algorithm that runs in polynomial time (ideally) and produc solution that is within a guaranteed factor of the optimal solution.

دعاؤں میں یاد رکھیں !!!

MUDASSAR Iqbal!

0340-6829977