5

UNIT

Selected Topics

CONTENTS

PART-1

Algebraic Computation, Fast Fourier Transform.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.1. **What is FFT (Fast Fourier Transformation) ? How the recursive FFT procedure works ? Explain.**

Answer

1. The Fast Fourier Transform (FFT) is a algorithm that computes a Discrete Fourier Transform (DFT) of n-length vector in $O(n \log n)$ time.
2. In the FFT algorithm, we apply the divide and conquer approach to polynomial evaluation by observing that if n is even, we can divide a degree $(n - 1)$ polynomial.

$$A(x) = a_0 + a_1x + a_2x^2 + ... + a_{n-1}\, x^{n-1}$$

into two degree $\left(\frac{n}{2} - 1\right)$ polynomials.

$$A^{[0]}(x) = a_0 + a_2x + a_4x^2 + ... + a_{n-2}\, x^{n/2-1}$$

$$A^{[1]}(x) = a_1 + a_3x + a_5x^2 + ... + a_{n-1}\, x^{n/2-1}$$

Where $A^{[0]}$ contains all the even index coefficients of A and $A^{[1]}$ contains all the odd index coefficients and we can combine these two polynomials into A, using the equation,

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) \qquad ...(5.1.1)$$

So that the problem of evaluating $A(x)$ at ω^k_n where $k = 0, 1, 2, ..., n-1$ reduces to,

i. Evaluating the degree $\left(\frac{n}{2} - 1\right)$ polynomial $A^{[0]}(x)$ and $A^{[1]}(x)$ at the point $(\omega^k_n)^2$ *i.e.*,

$$(\omega^0_n)^2, (\omega^1_n)^2, ..., (\omega^{n-1}_n)^2$$

because we know that if ω^k_n is a complex n^{th} root of unity then $(\omega^k_n)^2$ is a complex $\frac{n^{th}}{2}$ root of unity. Thus, we can evaluate each $A^{[0]}(x)$ and $A^{[1]}(x)$ at $(\omega^k_n)^2$ values.

ii. Combining the results according to the equation (5.1.1). This observation is the basis for the following procedure which computes the DFT of an n-element vector $a = (a_0, a_1,..., a_{n-1})$ where for sake of simplicity, we assume that n is a power of 2.

FFT (a, w) :

```
1.  n ← length [a]          n is a power of 2.
2.  if n = 1
3.  then return a
4.  $\omega_n \leftarrow e^{2\pi i/n}$
5.  $x \leftarrow \omega^0$              x will store powers of ω initially x = 1.
6.  $a^{[0]} \leftarrow (a_0, a_2,...a_{n-2})$
7.  $a^{[1]} \leftarrow (a_1, a_3, ... a_{n-1})$
8.  $y^{[0]} \leftarrow \text{FFT}(a^{[0]}, \omega^2)$     Recursive calls with $\omega^2$ as $(n/2)^{th}$ root of unity.
9.  $y^{[1]} \leftarrow \text{FFT}(a^{[0]}, \omega^2)$
10. for $k \leftarrow 0$ to $(n/2) - 1$
11. do $y_k \leftarrow y_k^{[0]} + x\, y_k^{[1]}$
12. $y_{k+(n/2)} \leftarrow y_k^{[0]} - x y_k^{[1]}$
13. $x \leftarrow x\omega_n$
14. return y
```

Line 2-3 represents the basis of recursion; the DFT of one element is the element itself. Since in this case

$$y_0 = a_0\, \omega_1^{\,0} = a_0\, 1 = a_0$$

Line 6-7 defines the recursive coefficient vectors for the polynomials $A^{[0]}$ and $A^{[1]}$. $\omega = \omega^k_n$

Line 8-9 perform the recursive DFT$_{n/2}$ computations setting for

$$k = 0, 1, 2, ..., \frac{n}{2} - 1 \text{ i.e.,}$$

$$y_k^{[0]} = A^{[0]}(\omega_n^{2k}), \quad y_k^{[1]} = A^{[1]}(\omega_n^{2k})$$

Lines 11-12 combine the results of the recursive DFT$_{n/2}$ calculations.

For $y_0, y_2, ... y_{(n/2)-1}$, line 11 yields.

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) = A(\omega_n^k) \quad \text{using equation (5.1.1)} \end{aligned}$$

For $y_{n/2}, y_{(n/2)+1} ... y_{n-1}$, line 12 yields.

$$\begin{aligned} y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} = y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]} \qquad [\because \omega_n^{k+(n/2)} = -\omega_n^k] \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) \\ &= A^{[0]}(\omega_n^{2k}\, \omega_n^n) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}\, \omega_n^n) \qquad [\because \omega_n^n = 1] \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) \\ &= A(\omega_n^{k+(n/2)}) \qquad \text{using equation (5.1.1)} \end{aligned}$$

each $k = 0, 1, 2, \ldots, (n/2) - 1$.

Thus, the vector y returned by the FFT algorithm will store the values of $A(x)$ at each of the roots of unity.

—

Que 5.2. **What is the application of Fast Fourier Transform (FFT) ? Also write the recursive algorithm for FFT.**

AKTU 2018-19, Marks 10

Answer

Application of Fast Fourier Transform :

1. Signal processing.
2. Image processing.
3. Fast multiplication of large integers.
4. Solving Poisson's equation nearly optimally.

Recursive algorithm : Refer Q. 5.1, Page 5–2B, Unit-5.

PART-2

String Matching.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.3. **What is string matching ? Discuss string matching problem. Also define string, substring and proper substring.**

Answer

String matching is a process of finding one or more occurrences of a pattern in a text.

String matching problem :

Given a text array $T[1 .. n]$ of n character and a pattern array $P[1 .. m]$ of m characters.

The problem is to find an integer s, called valid shift where $0 \le s < n - m$ and $T[s + 1 .... s + m] = P[1 ... m]$.

We further assume that the elements of P and T are characters drawn from a finite alphabet Σ such as $\{0, 1\}$ or $\{A, B, ... Z, a, b, ..., z\}$.

String : A string is traditionally a sequence of character, either as a literal constant or as some kind of variable.

Substring : Given a string $T[1 .. n]$, the substring is defined as $T[i .. j]$ for some $0 \le i \le j \le n - 1$, that is, the string formed by the characters in T from index j, inclusive. This means that a string is a substring of itself (simply take $i = 0$ and $j = m$).

Proper substring : The proper substring of string $T[1 .. n]$ is $T[i .. j]$ for some $0 \le i \le j \le n - 1$, that is, we must have either $i > 0$ or $j < m - 1$.

Using these definition, we can say given any string $T[1 .. n]$, the substring are

$$T[i .. j] = T[i]\ T[i + 1]\ T[i + 2]\ ...\ T[j]$$

for some $0 \leq i \leq j \leq n - 1$.
And proper substrings are

$$T[i .. j] = T[i]\ T[i + 1]\ T[i + 2]\ ...\ T[j]$$

for some $0 \leq i \leq j \leq n - 1$.
Note that if $i > j$, then $T[i .. j]$ is equal to the empty string or null, which has length zero. Using these notations, we can define of a given string $T[1 .. n]$ as $T[0 .. i]$ for some $0 \leq i \leq n - 1$ and suffix of a given string $T[1 - n]$ as $T[i .. n - 1]$ for some $0 \leq i \leq n - 1$.

Que 5.4. **What are the different types of string matching ? Explain one of them.**

Answer

Basic types of string matching algorithms are :

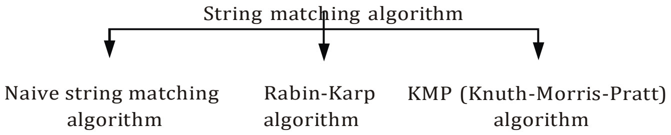

String matching algorithm
Naive string matching algorithm
Rabin-Karp algorithm
KMP (Knuth-Morris-Pratt) algorithm

Fig. 5.4.1.

Naive string matching :

The Naive approach simply test all the possible placement of pattern $P[1 .. m]$ relative to text $T[1 .. n]$. Specifically, we try shifts $s = [0, 1, ..., n - m]$, successively and for each shift, s, compare $T[s + 1 .. s + m]$ to $P[1 .. m]$.

Naive string matcher (*T*, *P*)

```
1.  $n \leftarrow$ length $[T]$
2.  $m \leftarrow$ length $[P]$
3.  for $s \leftarrow 0$ to $n - m$
4.  do if $P[1 .. m] = T[s + 1 .. s + m]$
5.  then print "pattern occurs with shift" $s$.
```

The Naive string matching procedure can be interpreted graphically as a sliding a pattern $P[1 .. m]$ over the text $T[1 .. m]$ and noting for which shift all of the characters in the pattern match the corresponding characters in the text.

To analyze the time of Naive matching, the given algorithm is implemented as follows, note that in this implementation, we use notation $P[1 .. j]$ to denote the substring of P from index i to index j. That is, $P[1 ... j] = P[i]\ P[i + 1]\ ...\ P[j]$.

Naive string matcher (*T*, *P*)

```
1.  $n \leftarrow$ length $[T]$
2.  $m \leftarrow$ length $[P]$
3.  for $s \leftarrow 0$ to $n - m$ do
```

```
4.          $j \leftarrow 1$
```

```
5.          while $j \le m$ and $T[s + j] = P[j]$ do
6.              $j \leftarrow j + 1$
7.          if $j > m$ then
8.          return valid shift $s$
9.      return no valid shift exist // i.e., there is no substring of $T$ matching $P$.
```

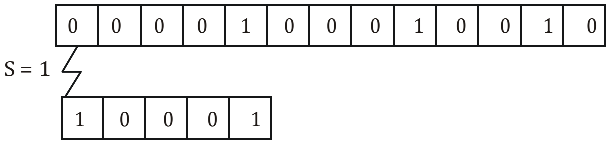
Que 5.5. **Show the comparisons that Naive string matcher makes for the pattern $P = \{10001\}$ in the text $T = \{0000100010010\}$**

Answer

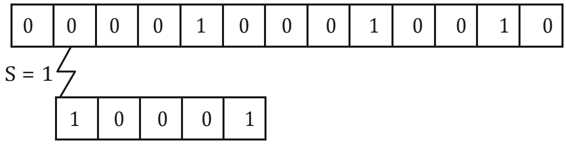
Given, $P = 10001$

$T = 0000100010010$

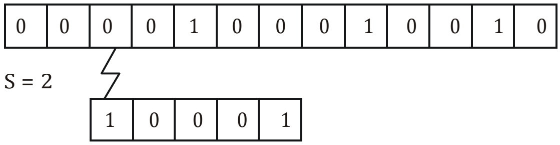
i.



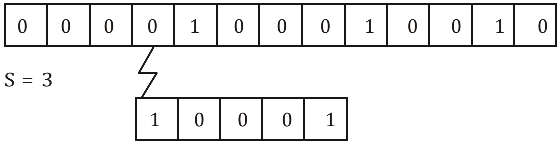
ii



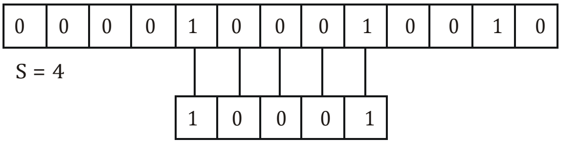
iii.



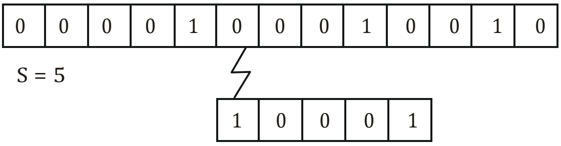
iv.



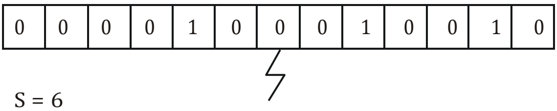
v.



vi.

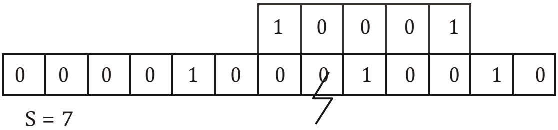


vii.



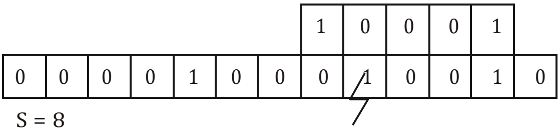
S = 6

viii.

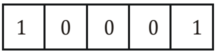


S = 7

ix.



S = 8



Que 5.6. **Write down Knuth-Morris-Pratt algorithm for string matching. Find the prefix function of the string *ababababca*.**

Answer

Knuth-Morris-Pratt algorithm for string matching :

COMPUTE-PREFIX-FUNCTION (*P*)

```
1.  $m \leftarrow$ length $[P]$
2.  $\pi[1] \leftarrow 0$
3.  $k \leftarrow 0$
4.  for $q \leftarrow 2$ to $m$
5.      do while $k > 0$ and $P[k + 1] \neq P[q]$
6.          do $k \leftarrow \pi[k]$
7.      if $P[k + 1] = P[q]$
8.          then $k \leftarrow k + 1$
9.      $\pi[q] \leftarrow k$
10. return $\pi$
```

KMP-MATCHER calls the auxiliary procedure COMPUTE-PREFIX-FUNCTION to compute π.

KMP-MATCHER (*T*, *p*)

```
1.  $n \leftarrow$ length $[T]$
2.  $m \leftarrow$ length $[P]$
3.  $\pi \leftarrow$ COMPUTE-PREFIX-FUNCTION $(P)$
4.  $q \leftarrow 0$
5.  for $i \leftarrow 1$ to $n$
6.      do while $q > 0$ and $P[q + 1] \neq T[i]$
7.          do $q \leftarrow \pi[q]$
```

```
8.        if $P[q+1] = T[i]$
```

```
9.          then $q \leftarrow q + 1$
10.  if $q = m$
11.  then print "pattern occurs with shift" $i - m$
12.  $q \leftarrow \pi [q]$
```

Prefix function of the string *ababababca* :

$m \leftarrow$ length $[P]$

$\therefore$ $m = 10$

Initially, $P[1] = 0$, $k = 0$

for $q \leftarrow 2$ to 10

for $q = 2$, $k \not> 0$

& $P[0 + 1] = P[2]$

$\therefore$ $\pi[2] = 0$

for $q = 3$, $k \not> 0$

& $P[0 + 1] = P[3]$

$\therefore$ $k \leftarrow k + 1 = 1$

& $\pi[3] \leftarrow 1$

for $q = 4$, $k > 0$

& $P[1 + 1] = P[4]$

$\therefore$ $k \leftarrow 1 + 1 = 2$

& $\pi[4] \leftarrow 2$

for $q = 5$, $k > 0$

& $P[2 + 1] = P[5]$

$\therefore$ $k \leftarrow 2 + 1 = 3$

& $\pi[5] \leftarrow 3$

for $q = 6$, $k > 0$

& $P[3 + 1] = P[6]$

$\therefore$ $k \leftarrow 3 + 1 = 4$

& $\pi[6] \leftarrow 4$

for $q = 7$, $k > 0$

& $P[4 + 1] = P[7]$

$\therefore$ $k \leftarrow 4 + 1 = 5$

& $\pi[7] \leftarrow 5$

for $q = 8$, $k > 0$

& $P[5 + 1] = P[8]$

$\therefore$ $k \leftarrow 5 + 1 = 6$

& $\pi[8] \leftarrow 6$

for $q = 9$, $k > 0$

& $P[6 + 1] = P[9]$

$\therefore$ $k \leftarrow \pi[k] = 6$

& $\pi[9] \leftarrow 6$

for $q = 10$, $k > 0$

& $P[6 + 1] = P[10]$

$\therefore$ $k \leftarrow 6 + 1 = 7$

& $\pi[10] \leftarrow 7$

String	*a*	*b*	*a*	*b*	*a*	*b*	*a*	*b*	*c*	*a*
$P[i]$	1	2	3	4	5	6	7	8	9	10
$\pi[i]$	0	0	1	2	3	4	5	6	6	7

Que 5.7. **Compute the prefix function π for the pattern *P* = *abacab* using KNUTH-MORRIS-PRATT algorithm. Also explain Naive string matching algorithm.**

AKTU 2017-18, Marks 10

Answer

Prefix function of the string *abacab* :

$m \leftarrow$ length $[P]$

$\therefore$ $m = 6$

Initially, $\pi[1] = 0$, $k = 0$

for $q \leftarrow 2$ to 6

for $q = 2$, $k \not> 0$

& $P[0 + 1] \neq P[2]$

$\therefore$ $\pi[2] = 0$

for $q = 3$, $k \not> 0$

& $P[0 + 1] = P[3]$

$\therefore$ $k = k + 1 = 1$

& $\pi[3] = 1$

for $q = 4$, $k > 0$

& $P[1 + 1] \neq P[4]$

$\therefore$ $k \leftarrow \pi[1] = 0$

$P[1] \neq P[4]$

& $\pi[4] = 0$

for $q = 5$, $k > 0$

& $P[0 + 1] = P[5]$

$\therefore$ $k \leftarrow 0 + 1 = 1$

& $\pi[5] = 1$

for $q = 6$, $k > 0$

& $P[1 + 1] = P[6]$

$\therefore$ $k \leftarrow 1 + 1 = 2$

& $\pi[6] = 2$

String	*a*	*b*	*a*	*c*	*a*	*b*
$P[i]$	1	2	3	4	5	6
$\pi[i]$	0	0	1	0	1	2

Naive string matching algorithm : Refer Q. 5.4, Page 5–5B, Unit-5.

Que 5.8. **Describe in detail Knuth-Morris-Pratt string matching algorithm. Compute the prefix function π for the pattern *ababbabbabbababbabb* when the alphabet is Σ = {*a*, *b*}.**

AKTU 2019-20, Marks 07

Answer

Knuth-Morris-Pratt string matching algorithm : Refer Q. 5.6, Page 5–7B, Unit-5.

Numerical :

pattern = ababbabbabbababbabb

length 19

Initially, $\pi(1) = 0$ and $k = 0$

For $q \leftarrow 2$ to 9

For $q = 2$ and $k \not> 0$

$P[0 + 1] \neq P[2]$.

$\pi[2] = 0$

For $q = 3$

$P[0 + 1] = P[3]$

$k = k + 1 = 1$

$\pi[3] = 1$

For $q = 4$ $k > 0$

$P[1 + 1] = P[4]$

$P[2] = P[4]$

$k = k + 1 = 2$

$\pi[4] = 2$

For $q = 5$ $k > 0$

$P[2 + 1] \neq P[5]$

$\pi[5] = 0$

For $q = 6$

$P[0 + 1] = P[6]$

$k = k + 1 = 1$

$\pi[6] = 1$

For $q = 7$ $k > 0$

$P[1 + 1] = P[7]$

$P[2] = P[7]$

$k = k + 1 = 2$

$\pi[7] = 0$

For $q = 8$ $k > 0$

$P[2 + 1] = P[8]$

$P[3] \neq P[8]$

$\pi[8] = 0$

For $q = 9$

$P[0 + 1] = P[9]$

$k = k + 1$

$\pi[9] = 2$

For $q = 10$ $k > 0$

$P[2 + 1] \neq P[10]$

$\pi[10] = 0$

For $q = 11$ $k > 0$

$P[0 + 1] \neq P[11]$

$\pi[11] = 0$

For $q = 12$ $k > 0$

$P[0 + 1] \neq P[12]$

$k = k + 1$

$\pi[12] = 2$

For $q = 13$ $k > 0$

$P[2 + 1] \neq P[13]$

$\pi[13] = 0$

For $q = 14$

$P[0 + 1] = P[14]$

$k = k + 1$

$\pi[14] = 2$

For $q = 15$ $k > 0$

$P[2 + 1] \neq P[15]$

$\pi[15] = 0$

For $q = 16$ $k > 0$

$P[0 + 1] \neq P[16]$

$\pi[16] = 0$

For $q = 17$

$P[0 + 1] \neq P[17]$

$k = k + 1$

$\pi[17] = 2$

For $q = 18$ $k > 0$

$P[2 + 1] \neq P[18]$

$\pi[18] = 0$

For $q = 19$

$P[0 + 1] \neq P[19]$

$\pi[19] = 0$

String	*a*	*b*	*a*	*b*	*b*	*a*	*b*	*b*	*a*	*b*	*b*	*a*	*b*	*a*	*b*	*b*	*a*	*b*	*b*
$P[i]$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$\pi[i]$	0	0	1	2	0	1	2	0	2	0	0	2	0	2	0	0	2	0	0

Que 5.9. **Write algorithm for Rabin-Karp method. Give a suitable example to explain it.**

OR

What is string matching algorithm ? Explain Rabin-Karp method with examples.

AKTU 2015-16, Marks 10

Answer

String matching algorithm : Refer Q. 5.3, Page 5–4B, Unit-5.

The Rabin-Karp algorithm :

The Rabin-Karp algorithm states that if two strings are equal, their hash values are also equal. This also uses elementary number-theoretic notions such as the equivalence of two numbers module a third.

Rabin-Karp-Matcher (*T*, *P*, *d*, *q*)

```
1.  $n \leftarrow$ length $[T]$
2.  $m \leftarrow$ length $[P]$
3.  $h \leftarrow d^{m-1}$ mod $q$
4.  $p \leftarrow 0$
5.  $t_o \leftarrow 0$
6.  for $i \leftarrow 1$ to $m$
7.      do $p \leftarrow (dp + p[i])$ mod $q$
8.      $t_0 \leftarrow (dt_0 + T[i])$ mod $q$
9.  for $s \leftarrow 0$ to $n-m$
10. do if $p = t_s$
11.         then if $p[1....m] = T[s + 1 ......s + m]$
12.         then "pattern occurs with shift" $s$
13.     if $s < n - m$
14. then $t_{s+1} \leftarrow (d(t_s - T[s + 1]\ h) + T[s + m + 1])$ mod $q$
```

Example of Rabin-Karp method : Working modulo $q = 11$, how many spurious hits does the Rabin-Karp matcher encounter in the text $T = 3141592653589793$ when looking for the pattern $p = 26$

Given, $p = 26$ and $q = 11$

Now we divide 26 by 11 *i.e.*,

Remainder is 4 and $m = 2$.

We know m denotes the length of p.

T																
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3

Now we divide 31 by 11, and get remainder is 9.

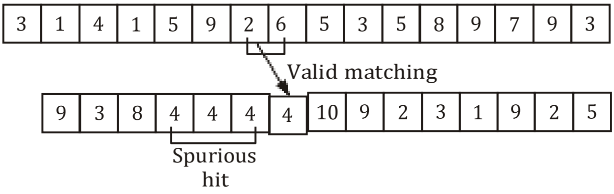
Similarly, 14 by 11 and get remainder is 3.

So, continue this step till last *i.e.*, 93 is divided by 11 and get remainder is 5.

After that we will store all remainder in a table.

9	3	8	4	4	4	4	10	9	2	3	1	9	2	5

Now we find valid matching.



The number of spurious hits is 3.

Que 5.10. **Give a linear time algorithm to determine if a text *T* is a cycle rotation of another string *T*. For example : RAJA and JARA are cyclic rotations of each other.**

AKTU 2018-19, Marks 10

Answer

Knuth-Morris-Pratt algorithm is used to determine if a text *T* is a cycle rotation of another string *T*′.

Knuth-Morris-Pratt algorithm : Refer Q. 5.6, Page 5–7B, Unit-5.

PART-3

Theory of NP – Completeness.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.11. **Discuss the problem classes P, NP and NP-complete.**

Answer

P : Class P are the problems which can be solved in polynomial time, which take time like O(n), O(n^2), O(n^3).

Example : Finding maximum element in an array or to check whether a string is palindrome or not. So, there are many problems which can be solved in polynomial time.

NP : Class NP are the problems which cannot be solved in polynomial time like TSP (travelling salesman problem).

Example : Subset sum problem is best example of NP in which given a set of numbers, does there exist a subset whose sum is zero, but NP problems are checkable in polynomial time means that given a solution of a problem, we can check that whether the solution is correct or not in polynomial time.

NP-complete : The group of problems which are both in NP and NP-hard are known as NP-complete problem.

Now suppose we have a NP-complete problem *R* and it is reducible to *Q* then *Q* is at least as hard as *R* and since *R* is an NP-hard problem, therefore *Q* will also be at least NP-hard, it may be NP-complete also.

Que 5.12. **Discuss the problem classes *P*, *NP* and NP-complete with class relationship.**

AKTU 2017-18, Marks 10

Answer

1. The notion of NP-hardness plays an important role in the relationship between the complexity classes *P* and *NP*.

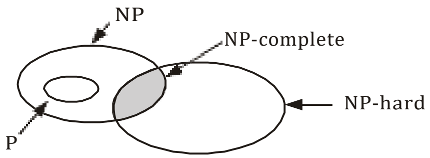


Fig. 5.12.1. Relationship among P, NP, NP-complete and NP-hard problems.

2. It is also often used to define the complexity class NP-complete which is the intersection of *NP* and NP-hard.
3. Consequently class NP-hard can be understood as the class of problems that are NP-complete or harder.
4. There are no polynomial time algorithms for NP-hard problems.
5. A problem being in *NP* means that the problem is "easy" (in a certain specific sense), whereas a problem being NP-hard means that the problem is "difficult" (in another specific sense).
6. A problem being in *NP* and a problem being NP-hard are not mutually exclusive. When a problem is both in *NP* and NP-hard, we say that the problem is NP-complete.
7. All problems in *NP* can be solved deterministically in time O(2*n*).
8. An example of an NP-hard problem is the decision problem subset-sum. Given a set of integers, does any non-empty subset of them add up to zero ? *i.e.*, a yes or no question, and happens to be NP-complete.
9. There are, also decision problems that are NP-hard but not NP-complete.
10. For example, in the halting problem "given a program and its input, will it run forever" *i.e.*, yes or no question, so this is a decision problem. It is case to prove that the halting problem is NP-hard but not NP-complete.

Que 5.13. **What is NP-completeness ?**

Answer

A language $L \subseteq \{0,1\}^*$ is NP-complete if it satisfies the following two properties :

i. $L \in$ NP ; and

ii. For every $L' \leq_p L$

NP-hard : If a language *L* satisfies property (*ii*), but not necessarily property (*i*), we say that *L* is NP-hard.

NP-complete : We use the notation $L \in$NPC to denote that *L* is NP-complete.

Theorem : If any NP-complete problem is polynomial time solvable, thenP = NP. If any problem in NP is not polynomial time solvable, then all NP complete problems are not polynomial time solvable.

Proof : Suppose that $L \in$ P and also that $L \in$ NPC. For any $L' \in$ NP, we have $L' \in_p L$ by property (*ii*) of the definition of NP-completeness. We know if $L' \leq_p L$ then $L \in P$ implies $L' \in$ P, which proves the first statement.

To prove the second statement, suppose that there exists and $L \in$ NP such that $L \notin$ P. Let $L' \in$ NPC be any NP-complete language, and for the purpose of contradiction, assume that $L' \in$ P. But then we have $L \leq_p L'$ and thus $L \in$ P.

Que 5.14. Explain NP-hard and NP-complete problems and also define the polynomial time problems and write a procedure to solve NP-problems.

OR

Write short note on NP-hard and NP-complete problems.

OR

Define NP-hard and NP-complete problems. What are the steps involved in proving a problem NP-complete ? Specify the problems already proved to be NP-complete.

AKTU 2019-20, Marks 07

Answer

NP-hard problem :

1. We say that a decision problem P_i is NP-hard if every problem in NP is polynomial time reducible to P_i.
2. In symbols,
   P_i is NP-hard if, for every $P_j \in$ NP, $P_j \xrightarrow{\text{Poly}} P_i$.
3. This does not require P_i to be in NP.
4. Highly informally, it means that P_i is 'as hard as' all the problem in NP.
5. If P_i can be solved in polynomial time, then all problems in NP.
6. Existence of a polynomial time algorithm for an NP-hard problem implies the existence of polynomial solution for every problem in NP.

NP-complete problem :

1. There are many problems for which no polynomial time algorithms is known.
2. Some of these problems are travelling salesman problem, optimal graph colouring, the Knapsack problem, Hamiltonian cycles, integer programming, finding the longest simple path in a graph, and satisfying a Boolean formula.
3. These problems belongs to an interesting class of problems called the "NP-complete" problems, whose status is unknown.
4. The NP-complete problems are traceable *i.e.*, require a super polynomial time.

Polynomial time problem :

An algorithm is said to be solvable in polynomial time if the number of steps required to complete the algorithm for a giveninput is $O(n^k)$ for some non-negative integer k where n is the complexity of input.

Polynomial time verifiable algorithm : A polynomial time algorithm *A* is said to be polynomial time verifiable if it has following properties :

1. The input to *A* consists of an instance *I* of *X* (*X* is a decision problem) and a string *S* such that the length of *S* is bounded by some polynomial in the size of *I*.
2. The output of *A* is either yes or no.
3. If *I* is a negative instance of *X*, then the output of *A* is "no" regardless of the value of *S*.
4. If *I* is a positive instance of *X*, then there is at least one choice of *S* for which *A* output "yes".

Procedure to solve NP-problems :

1. The class NP is the set of all decision problems that have instances that are solvable in polynomial time using a non-deterministic turing machine.
2. In a non-deterministic turing machine, in contrast to a deterministic turing machine, for each state, several rules with different actions can be applied.
3. Non-deterministic turing machine branches into many copies that are represented by a computational tree in which there are different computational paths.
4. The class NP corresponds to a non-deterministic turing machine that guesses the computational path that represents the solution.
5. By doing so, it guesses the instances of the decision problem.
6. In the second step, a deterministic turing machine verifies whether the guessed instance leads to a "yes" answer.
7. It is easy to verify whether a solution is valid or not. This statement does not mean that finding a solution is easy.

Que 5.15. **Differentiate NP-complete with NP-hard.**

AKTU 2016-17, Marks 10

Answer

S. No.	NP-complete	NP-hard
1.	An NP-complete problems is one to which every other polynomial-time non-deterministic algorithm can be reduced in polynomial time.	NP-hard problems is one to which an NP- complete problem is Turing-reducible.
2.	NP-complete problems do not corresponds to an NP-hard problem.	NP-hard problems correspond to an NP-complete problem.

3.	NP-complete problems are exclusively decision problem.	NP-hard problems need not to be decision problem.
4.	NP- complete problems have to be in NP-hard and also in NP.	NP-hard problems do not have to be in NP.
5.	**For example :** 3- SAT vertex cover problem is NP-complete.	**For example :** Halting problem is NP-hard.

Que 5.16. **Discuss NP-complete problem and also explain minimum vertex cover problem in context to NP-completeness.**

Answer

NP-complete problem : Refer Q. 5.14, Page 5–15B, Unit-5.

Minimum vertex cover problem :

1. A vertex cover of a graph is a set of vertices such that each edge of the graph is incident to at least one vertex of the set.
2. The problem of finding a minimum vertex cover is a classical optimization problem in computer science and is a typical example of an NP-hard optimization problem that has an approximation algorithm.
3. Its decision version, the vertex cover problem, was one of Karp's 21 NP-complete problems and is therefore a classical NP-complete problem in computational complexity theory.
4. Furthermore, the vertex cover problem is fixed-parameter tractable and a central problem in parameterized complexity theory.
5. The minimum vertex cover problem can be formulated as a half-integral linear program whose dual linear program is the maximum matching problem.
6. Formally, a vertex cover V' of undirected graph $G = (V, E)$ is a subset of V such that $uv \in V' \vee v \in V'$ *i.e.*, it is a set of vertices V' where every edge has at least one endpoint in the vertex cover V'. Such a set is said to cover the edges of G.

   Example : Fig. 5.16.1 shows examples of vertex covers, with some vertex cover V' marked in dark.

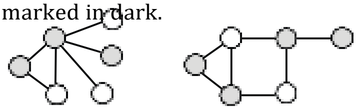


Fig. 5.16.1.

7. A minimum vertex cover is a vertex cover of smallest possible size.
8. The vertex cover number τ is the size of a minimum cover, *i.e.*, $\tau = |V'|$. The Fig. 5.16.2 shows examples of minimum vertex covers in the previous graphs.

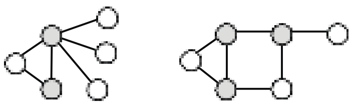


Fig. 5.16.2.

Que 5.17. **Discuss different types of NP-complete problem.**

Answer

Types of NP-complete problems :

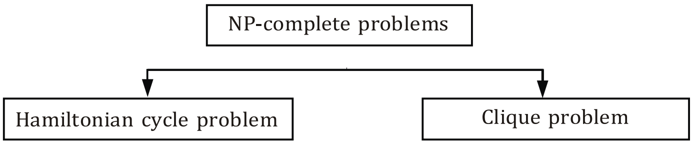


Fig. 5.17.1.

Hamiltonian cycle problem :

1. A Hamiltonian cycle of an undirected graph $G = (V, E)$ is a simple cycle that contains each vertex in V.

2. A graph that contains a Hamiltonian cycle is said to be Hamiltonian, otherwise it is said to be non-Hamiltonian.

The clique problem :

1. A clique in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E.

2. The size of a clique is the number of vertices it contains.
   CLIQUE = $\{<G, K> : G$ is a graph with a clique of size $K\}$

3. The clique problem is the optimization problem of finding a clique of maximum size in a graph.

4. As a decision problem, we ask simply whether a clique of a given size k exists in the graph.

Que 5.18. **Show that Hamiltonian circuit is NP-complete.**

Answer

Theorem : Hamiltonian circuit (HC) is NP-complete.

Proof :

1. Let us define a non-deterministic algorithm A that takes, as input, a graph G encoded as an adjacency list in binary notation, with the vertices numbered 1 to N.

2. We define A to first iteratively call the choose method to determine a sequence S of $N + 1$ numbers from 1 to N.

3. Then, we have A to check that each number from 1 to N appears exactly once in S (for example, by sorting S), except for the first and last numbers in S, which should be the same.

4. Then, we verify that a sequence S defines a cycle of vertices and edges in G.
5. A binary encoding of the sequence S is clearly of size at most n, where n is the size of the input. Moreover, both of the checks made on the sequence S can be done in polynomial time in n.
6. Observe that if there is a cycle in G that visits each vertex of G exactly once; returning to its starting vertex, then there is a sequence S for which A will output "yes."
7. Likewise, if A outputs "yes," then it has found a cycle in G that visits each vertex of G exactly once, returning to its starting point. Hence, Hamiltonian circuit is NP-complete.

Que 5.19. **Show that CLIQUE problem is NP-complete.**

Answer

Problem : The CLIQUE problem is defined as {< G, k >|, G is a graph with a k-clique}. Show that CLIQUE is NP-complete.

Proof :

1. First, to show that CLIQUE is in NP. Given an instance of < G, k > and a k-clique we can easily verify in $O(n^2)$ time that we do, in fact, have a k-clique.
2. Now, we want to show that 3-SAT is CLIQUE. Let F be a boolean formula in CNF.
3. For each literal in F we will make a vertex in the graph *i.e.*,

   $(x_1 + x_2 + x_3)$ $(x_1 + x_2 + x_3)$ has 6 vertices. Let

   k be the number of clauses in F.
4. We will connect each vertex to all of the other vertices that are logically compatible except for the ones that are in the same clause.
5. Now, if we have a satisfiable assignment we will have a k-clique because the satisfying vertices will all be connected to one another.
6. Thus, we can use CLIQUE to solve 3-SAT so CLIQUE is NP-complete.

Que 5.20. **Define different complexity classes in detail with suitable example. Show that TSP problem is NP-complete.**

Answer

Different complexity classes :

There are some complexity classes involving randomized algorithms :

1. **Randomized polynomial time (RP) :** The class RP consists of all languages L that have a randomized algorithm A running in worst case polynomial time such that for any input x in Σ^*

$$x \in L \Rightarrow P[A(x) \text{ accepts}] \geq \frac{1}{2}$$
$$x \notin L \Rightarrow P[A(x) \text{ accepts}] = 0$$

Independent repetitions of the algorithms can be used to reduce the probability of error to exponentially small.

2. **Zero-error probabilistic polynomial time (ZPP) :** The class ZPP is the class of languages which have Las Vegas algorithms running in expected polynomial time.

$$\text{ZPP} = \text{RP} \cap \text{co-RP}$$

where a language L is in co-X where X is a complexity class if and only if its complement $\Sigma^* - L$ is in X.

3. **Probabilistic polynomial time (PP) :** The class PP consists of all languages L that have a randomized algorithm A running in worst case polynomial time such that for any input x in Σ^*.

$$x \in L \Rightarrow P[A(x) \text{ accepts}] \geq \frac{1}{2}$$
$$x \notin L \Rightarrow P[A(x) \text{ accepts}] < \frac{1}{2}$$

To reduce the error probability, we cannot repeat the algorithm several times on the same input and produce the output which occurs in the majority of those trials.

4. **Bounded-error probabilistic polynomial time (BPP) :** The class BPP consists of all languages that have a randomized algorithm A running in worst case polynomial time such that for any input x in Σ^*.

$$x \in L \Rightarrow P[A(x) \text{ accepts}] \geq \frac{3}{4}$$
$$x \notin L \Rightarrow P[A(x) \text{ accepts}] \leq \frac{1}{4}$$

For this class of algorithms, the error probability can be reduced to $1/2n$ with only a polynomial number of iterations.

For a given a graph $G = (V, E)$ and a number k, does there exist a tour C on G such that the sum of the edge weights for edges in C is less than or equal to k.

Proof :

Part 1 : TSP is in NP.

Proof :

1. Let a hint S be a sequence of vertices $V = v_1, \ldots, v_n$.
2. We then check two things :
   a. First we check that every edge traversed by adjacent vertices is an edge in G, such that the sum of these edge weights is less than or equal to k.
   b. Secondly we check that every vertex in G is in V, which assures that every node has been traversed.
3. We accept S if and only if S satisfies these two questions, otherwise reject.
4. Both of these checks are clearly polynomial, thus our algorithm forms a verifier with hint S, and TSP is consequently in NP.

Part 2 : TSP is NP-Hard.

Proof :

1. To show that TSP is NP-Hard, we must show that every problem y in NP reduces to TSP in polynomial time.
2. To do this, consider the decision version of Hamiltonian Cycle (HC).
3. Take $G = (V, E)$, set all edge weights equal to 1, and let $k = |V| = n$, that is, k equals the number of nodes in G.
4. Any edge not originally in G then receives a weight of 2 (traditionally TSP is on a complete graph, so we need to add in these extra edges).
5. Then pass this modified graph into TSP, asking if there exists a tour on G with cost at most k. If the answer to TSP is YES, then HC is YES. Likewise if TSP is NO, then HC is NO.

First direction : HC has a YES answer => TSP has a YES answer.

Proof :

1. If HC has a YES answer, then there exists a simple cycle C that visits every node exactly once, thus C has n edges.
2. Since every edge has weight 1 in the corresponding TSP instance for the edges that are in the HC graph, there is a Tour of weight n. Since $k = n$, and given that there is a tour of weight n, it follows that TSP has a YES answer.

Second direction : HC has a NO answer => TSP has a NO answer.

Proof :

1. If HC has a NO answer, then there does not exist a simple cycle C in G that visits every vertex exactly once. Now suppose TSP has a YES answer.
2. Then there is a tour that visits every vertex once with weight at most k.
3. Since the tour requires every node be traversed, there are n edges, and since $k = n$, every edge traversed must have weight 1, implying that these edges are in the HC graph. Then take this tour and traverse the same edges in the HC instance. This forms a Hamiltonian Cycle, a contradiction.

   This concludes Part 2. Since we have shown that TSP is both in NP and NP-Hard, we have that TSP is NP-Complete.

Que 5.21. Prove that three colouring problem is NP-complete.

AKTU 2016-17, Marks 10

Answer

1. To show the problem is in NP, let us take a graph $G(V, E)$ and a colouring c, and checks in $O(n^2)$ time whether c is a proper colouring by checking if the end points of every edge $e \in E$ have different colours.
2. To show that 3-COLOURING is NP-hard, we give a polytime reduction

from 3-SAT to 3-COLOURING.

3. That is, given an instance ϕ of 3-SAT, we will construct an instance of 3-COLOURING (*i.e.*, a graph $G(V, E)$) where G is 3-colourable iff ϕ is satisfiable.
4. Let ϕ be a 3-SAT instance and $C_1, C_2, ..., C_m$ be the clauses of ϕ defined over the variables $\{x_1, x_2, ..., x_n\}$.
5. The graph $G(V, E)$ that we will construct needs to capture two things :
   a. Somehow establish the truth assignment for $x_1, x_2, ..., x_n$ via the colours of the vertices of G; and
   b. Somehow capture the satisfiability of every clause C_i in ϕ.
6. To achieve these two goals, we will first create a triangle in G with three vertices $\{T, F, B\}$ where T stands for True, F for False and B for Base.
7. Consider $\{T, F, B\}$ as the set of colours that we will use to colour (label) the vertices of G.
8. Since this triangle is part of G, we need 3 colours to colour G.
9. Now we add two vertices $v_i, \overline{v}_i$ for every literal x_i and create a triangle B, v_i , $\overline{v}_i$ for every $(v_i , \overline{v}_i)$ pair, as shown in Fig. 5.21.1.

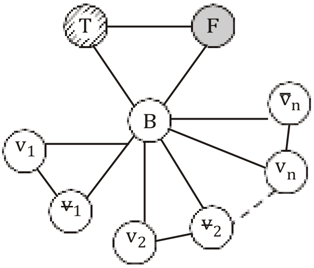


Fig. 5.21.1.

10. This construction captures the truth assignment of the literals.
11. Since if G is 3-colourable, then either v_i or $\overline{v}_i$ gets the colour T, and we interpret this as the truth assignment to v_i.
12. Now we need to add constraints to G to capture the satisfiability of the clauses of ϕ.
13. To do so, we introduce the Clause Satisfiability Gadget, (the OR-gadget). For a clause $C_i = (a \vee b \vee c)$, we need to express the OR of its literals using our colours $\{T, F, B\}$.
14. We achieve this by creating a small gadget graph that we connect to the literals of the clause. The OR-gadget is constructed as follows :

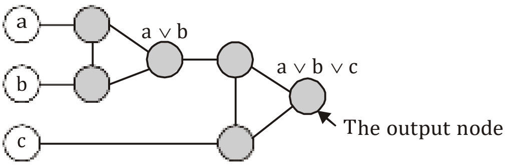


Fig. 5.21.2.

15. Consider this gadget graph as a circuit whose output is the node labeled $a \vee b \vee c$. We basically want this node to be coloured T if C_i is satisfied and F otherwise.
16. This is a two step construction : The node labelled $a \vee b$ captures the output of $(a \vee b)$ and we repeat the same operation for $((a \vee b) \vee c)$. If we play around with some assignments to a, b, c, we will notice that the gadget satisfies the following properties :
    a. If a, b, c are all coloured F in a 3-colouring, then the output node of the OR-gadget has to be coloured F. Thus capturing the unsatisfiability of the clause $C_i = (a \vee b \vee c)$.
    b. If one of a, b, c is coloured T, then there exists a valid 3-colouring of the OR-gadget where the output node is coloured T. Thus again capturing the satisfiability of the clause.
17. Once we add the OR-gadget of every C_i in ϕ, we connect the output node of every gadget to the Base vertex and to the False vertex of the initial triangle, as follows :

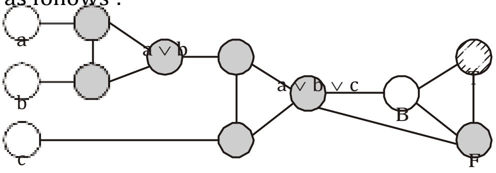


Fig. 5.21.3.

18. Now we prove that our initial 3-SAT instance ϕ is satisfiable if and only the graph G as constructed above is 3-colourable. Suppose ϕ is satisfiable and let $(x_1^*, x_2^*, ..., x_n^*)$ be the satisfying assignment.
19. If x_i^* is assigned True, we colour v_i with T and $\bar{v}_i$ with F (recall they are connected to the Base vertex, coloured B, so this is a valid colouring).
20. Since ϕ is satisfiable, every clause $C_i = (a \vee b \vee c)$ must be satisfiable, *i.e.*, at least of a, b, c is set to True. By the property of the OR-gadget, we know that the gadget corresponding to C_i can be 3-coloured so that the output node is coloured T.
21. And because the output node is adjacent to the False and Base vertices of the initial triangle only, this is a proper 3-colouring.
22. Conversely, suppose G is 3-colourable. We construct an assignment of the literals of ϕ by setting x_i to True if v_i is coloured T and vice versa.
23. Now consider this assignment is not a satisfying assignment to ϕ, then this means there exists at least one clause $C_i = (a \vee b \vee c)$ that was not satisfiable.
24. That is, all of a, b, c were set to False. But if this is the case, then the output node of corresponding OR-gadget of C_i must be coloured F.
25. But this output node is adjacent to the False vertex coloured F; thus contradicting the 3-colourability of G.
26. To conclude, we have shown that 3-COLOURING is in NP and that it is

NP-hard by giving a reduction from 3-SAT.

27. Therefore 3-COLOURING is NP-complete.

Que 5.22. **Prove that *P* is the subset of NP.**

Answer

To prove : *P* is the subset of NP.

Proof :

1. If $L \in P$, then $L \in$ NP, as there is a polynomial time algorithm to decide L, this algorithm can easily be converted into a row argument verification algorithm that simply ignores any exception and accepts exactly those input strings it determines to be in L.
2. Thus, $P \subseteq$ NP.

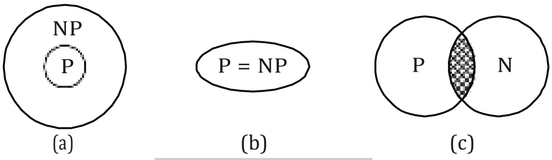


Fig. 5.22.1. *P* Vs NP.

PART-4

Approximation Algorithm.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.23. **Describe approximation algorithm in detail. How it differ with deterministic algorithm. Show that TSP is 2 approximate.**

OR

Explain approximation algorithms with suitable examples.

AKTU 2015-16, 2017-18; Marks 10

Answer

Approximation algorithm :

1. An approximation algorithm is a way of dealing with NP-completeness for optimization problem. This technique does not guarantee the best solution.
2. The best of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at most polynomial time.

3. Let $c(i)$ be the cost of solution produced by approximate algorithm and $c^*(i)$ be the cost of optimal solution for some optimization problem instance i.
4. For minimization and maximization problem, we are interested in finding a solution of a given instance i in the set of feasible solutions, such that $c(i) / c^*(i)$ and $c^*(i) / c(i)$ be as small as possible respectively.
5. We say that an approximation algorithm for the given problem instance i, has a ratio bound of $p(n)$ if for any input of size n, the cost c of the solution produced by the approximation algorithm is within a factor of $p(n)$ of the cost c^* of an optimal solution. That is
$$\max(c(i) / c^*(i), c^*(i) / c(i)) \leq p(n)$$
The definition applies for both minimization and maximization problems.
6. $p(n)$ is always greater than or equal to 1. If solution produced by approximation algorithm is true optimal solution then clearly we have $p(n) = 1$.
7. For a minimization problem, $0 < c^*(i) < c(i)$, and the ratio $c(i) / c^*(i)$ gives the factor by which the cost of the approximate solution is larger than the cost of an optimal solution.
8. Similarly, for a maximization problem, $0 < c(i) \leq c^*(i)$, and the ratio $c^*(i) / c(i)$ gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution.

Difference between deterministic algorithm and approximation algorithm :

S. No.	Deterministic algorithm	Approximation algorithm
1.	It does not deal with optimization problem.	It deals with optimization problem.
2.	It has initial and final step.	It does not have initial or final state.
3.	It require finite state machine.	It does not require finite state machine.
4.	It fails to deliver a result.	It gives an optimal result.
5.	It does not apply to maximization or minimization problem.	It applies to maximization and minimization problem.

Proof :

TSP is 2-approximate :

Let H^* denote the optimal tour. Observe that a TSP with one edge removed is a spanning tree (not necessarily MST).

It implies that the weight of the MST 'T' is in lower bound on the cost of an optimal tour.
$$c(T) \leq c(H^*)$$
A "Full" walk, W, traverse every edge of MST, T, exactly twice. That is,
$$c(W) = 2c(T)$$
which means
$$c(W) \leq 2c(H^*)$$

and we have

$$c(W) / c(H^*) \leq p(n) = 2$$

That is, the cost of walk, $c(W)$, of the solution produced by the algorithm is within a factor of $p(n) = 2$ of the cost $c(H^*)$ of an optimal solution.

Que 5.24. **Explain vertex cover problem with algorithm and analysis.**

Answer

A vertex cover of an undirected graph $G = (V, E)$ is a subset of $V' \subseteq V$ such that if edge $(u, v) \in G$ then $u \in V'$ or $v \in V'$ (or both).

Problem : Find a vertex cover of maximum size in a given undirected graph. This optimal vertex cover is the optimization version of an NP-Complete problem but it is not too hard to find a vertex cover that is near optimal.

Approx-vertex-cover (*G* : Graph)

```
1.  $c \leftarrow \phi$
2.  $E' \leftarrow E[G]$
3.  while $E'$ is not empty
4.  do Let $(u, v)$ be an arbitrary edge of $E'$
5.  $c \leftarrow c \cup \{u, v\}$
6.  Remove from $E'$ every edge incident on either $u$ or $v$
7.  return $c$
```

Analysis : It is easy to see that the running time of this algorithm is $O(V + E)$, using adjacency list to represent E'.

Que 5.25. **Describe approximation algorithm in detail. What is the approximation ratio ? Show that vertex cover problem is2-approximate.**

Answer

Approximation algorithm : Refer Q. 5.23, Page 5–24B, Unit-5.

Proof :

Vertex cover problem is 2-approximate :

Goal : Since this is a minimization problem, we are interested in smallest possible c / c^*. Specifically we want to show $c / c^* = 2 = p(n)$.

In other words, we want to show that Approx-Vertex-Cover algorithm returns a vertex-cover that is almost twice the size of an optimal cover.

Proof : Let the set c and c^* be the sets output by Approx-Vertex-Cover and Optimal-Vertex-Cover respectively. Also, let A be the set of edges.

Because, we have added both vertices, we get $c = 2|A|$ but Optimal-Vertex-Cover would have added one of two.

$$c / c^* \leq p(n) = 2.$$

Formally, since no two edge in A are covered by the same vertex from c^* and the lower bound : $|c^*| \geq A$ ...(5.25.1)

on the size of an Optimal-Vertex-Cover.

Now, we pick both end points yielding an upper bound on the size of Vertex-Cover :

$$|c| \leq 2|A|$$

Since, upper bound is an exact in this case, we have

$$|c| = 2|A| \quad \text{...(5.25.2)}$$

Take $|c|/2 = |A|$ and put it in equation (5.25.1)

$$|c^*| \geq |c|/2$$

$$|c^*|/|c| \geq 1/2$$

$|c^*|/|c| \leq 2 = p(n)$ Hence the theorem proved.

Que 5.26. **Explain Travelling Salesman Problem (TSP) with the triangle inequality.**

Answer

Problem : Given a complete graph with weights on the edges, find a cycle of least total weight that visits each vertex exactly once. When the cost function satisfies the triangle inequality, we can design an approximate algorithm for TSP that returns a tour whose cost is not more than twice the cost of an optimal tour.

APPROX-TSP-TOUR (*G*, *c*) :

1. Select a vertex $r \in V[G]$ to be a "root" vertex.
2. Compute a minimum spanning tree T for G from root r using MST-PRIM (G, c, r).
3. Let L be the list of vertices visited in a pre-order tree walk of T.
4. Return the Hamiltonian cycle H that visits the vertices in the order L.

Outline of an approx-TSP tour : First, compute a MST (minimum spanning tree) whose weight is a lower bound on the length of an optimal TSP tour. Then, use MST to build a tour whose cost is no more than twice that of MST's weight as long as the cost function satisfies triangle inequality.

Que 5.27. **Write short notes on the following using approximation algorithm with example.**

i. Nearest neighbour

ii. Multifragment heuristic

Answer

i. Nearest neighbour :

The following well-known greedy algorithm is based on the nearest-neighbour heuristic *i.e.*, always go next to the nearest unvisited city.

Step 1 : Choose an arbitrary city as the start.

Step 2 : Repeat the following operation until all the cities have been visited : go to the unvisited city nearest the one visited last (ties can be broken arbitrarily).

Step 3 : Return to the starting city.

Example :

1. For the instance represented by the graph in Fig. 5.27.1, with a as the starting vertex, the nearest-neighbour algorithm yields the tour (Hamiltonian circuit) s_a: $a - b - c - d - a$ of length 10.

2. The optimal solution, as can be easily checked by exhaustive search, is the tour s^*: $a - b - d - c - a$ of length 8. Thus, the accuracy ratio of this approximation is

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{10}{8} = 1.25$$

i.e., tour s_a is 25 % longer than optimal tour s^*.

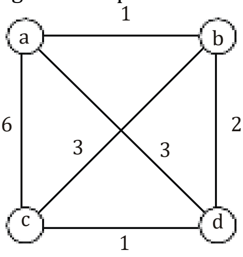


Fig. 5.27.1. Instance of the traveling salesman problem.

3. Unfortunately, except for its simplicity, not many good things can be said about the nearest-neighbour algorithm.
4. In particular, nothing can be said in general about the accuracy of solutions obtained by this algorithm because it can force us to traverse a very long edge on the last leg of the tour.
5. Indeed, if we change the weight of edge (a, d) from 6 to an arbitrary large number $w \geq 6$ in given example, the algorithm will still yield the tour $a - b - c - d - a$ of length $4 + w$, and the optimal solution will still be $a - b - d - c - a$ of length 8. Hence,

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{4 + w}{8}$$

which can be made as large as we wish by choosing an appropriately large value of w. Hence, $RA = \infty$ for this algorithm.

ii. Multifragment heuristic :

Another natural greedy algorithm for the traveling salesman problem considers it as the problem of finding a minimum-weight collection of edges in a given complete weighted graph so that all the vertices have degree 2.

Step 1 : Sort the edges in increasing order of their weights. (Ties can be broken arbitrarily.) Initialize the set of tour edges to be constructed to the empty set.

Step 2 : Repeat this step n times, where n is the number of cities in the instance being solved : add the next edge on the sorted edge list to the set of tour edges, provided this addition does not create a vertex of degree 3 or a cycle of length less than n; otherwise, skip the edge.

Step 3 : Return the set of tour edges.

Example :

1. Applying the algorithm to the graph in Fig. 5.27.1 yields $\{(a, b), (c, d), (b, c), (a, d)\}$.

2. There is, however, a very important subset of instances, called Euclidean, for which we can make a non-trivial assertion about the accuracy of both the nearest-neighbour and multifragment-heuristic algorithms.
3. These are the instances in which intercity distances satisfy the following natural conditions :
   - **a. Triangle inequality :** $d[i, j] \le d[i, k] + d[k, j]$ for any triple of cities i, j, and k (the distance between cities i and j cannot exceed the length of a two-leg path from i to some intermediate city k to j).
   - **b. Symmetry :** $d[i, j] = d[j, i]$ for any pair of cities i and j (the distance from i to j is the same as the distance from j to i).
4. A substantial majority of practical applications of the traveling salesman problem are its Euclidean instances.
5. They include, in particular, geometric ones, where cities correspond to points in the plane and distances are computed by the standard Euclidean formula.
6. Although the performance ratios of the nearest-neighbour and multifragment-heuristic algorithms remain unbounded for Euclidean instances, their accuracy ratios satisfy the following inequality for any such instance with $n \ge 2$ cities :

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{4+w}{8}$$

where $f(s_a)$ and $f(s^*)$ are the lengths of the heuristic tour and shortest tour.

Que 5.28. What is an approximation algorithm ? What is meant by $p(n)$ approximation algorithms ? Discuss approximation algorithm for Travelling Salesman Problem.

AKTU 2019-20, Marks 07

Answer

Approximation algorithm : Refer Q. 5.23, Page 5–24B, Unit-5.

$p(n)$ approximation algorithm : A is a $p(n)$ approximate algorithm if and only if for every instance of size n, the algorithm achieves an approximation ratio of $p(n)$. It is applied to both maximization $(0 < C(i) \le C^*(i))$ and minimization $(0 < C^*(i) \le C(i))$ problem because of the maximization factor and costs are positive. $p(n)$ is always greater than 1.

Approximation algorithm for Travelling Salesman Problem (TSP) :

1. The key to designing approximation algorithm is to obtain a bound on the optimal value (OPT).
2. In the case of TSP, the minimum spanning tree gives a lower bound on OPT.
3. The cost of a minimum spanning tree is not greater than the cost of an optimal tour.

The algorithm is as follows :

1. Find a minimum spanning tree of G.

2. Duplicate each edge in the minimum spanning tree to obtain a Eulerian graph.
3. Find a Eulerian tour (*J*) of the Eulerian graph.
4. Convert *J* to a tour *T* by going through the vertices in the same order of *T*, skipping vertices that were already visited.

PART-5

Randomized Algorithm.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.29. **Write short notes on randomized algorithms.**

OR

Explain approximation and randomized algorithms.

AKTU 2017-18, Marks 10

Answer

Approximation algorithm : Refer Q. 5.23, Page 5–24B, Unit-5.

Randomized algorithm :

1. A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased bits and it is then allowed to use these random bits to influence its computation.
2. An algorithm is randomized if its output is determined by the input as well as the values produced by a random number generator.
3. A randomized algorithm makes use of a randomizer such as a random number generator.
4. The execution time of a randomized algorithm could also vary from run to run for the same input.
5. The algorithm typically uses the random bits as an auxiliary input to guide its behaviour in the hope of achieving good performance in the "average case".
6. Randomized algorithms are particularly useful when it faces a malicious attacker who deliberately tries to feed a bad input to the algorithm.

Randomized algorithm are categorized into two classes :

i. **Las Vegas algorithm :** This algorithm always produces the same output for the same input. The execution time of Las Vegas algorithm depends on the output of the randomizer.

ii. **Monte Carlo algorithm :**

a. In this algorithm output might differ from run to run for the same input.

b. Consider any problem for which there are only two possible answers, say yes and no.
c. If a Monte Carlo algorithm is used to solve such a problem then the algorithm might give incorrect answers depending on the output of the randomizer.
d. Then the requirement is that the probability of an incorrect answer from a Monte Carlos algorithm be low.

Que 5.30. Write a short note on randomized algorithm. Write its merits, and applications.

Answer

Randomized algorithm : Refer Q. 5.29, Page 5–30B, Unit-5.

Merits :

1. Simple.
2. High efficiency.
3. Better complexity bounds.
4. Random selection of good and bad choices.
5. Cost efficient.

Applications :

1. Randomized quick sort algorithm
2. Randomized minimum-cut algorithm
3. Randomized algorithm for N-Queens problem
4. Randomized algorithm for majority element

Que 5.31. Write the EUCLID'S GCD algorithm. Compute gcd (99, 78) with EXTENDED-EUCLID.

Answer

Euclid's GCD algorithm :

The inputs a and b are arbitrary non-negative integers.

```
EUCLID (a, b) {
if (b = = 0)
then return a;
else return EUCLID (b, a mod b); }
EXTEUCLID (a, b) {
//returns a triple (d, x, y) such that d = gcd (a, b)
//d = = (a × x + b × y)
if (b = = 0) return (a, 1, 0);
(d_1, x_1, y_1) = EXTEUCLID (b, a % b);
d = d_1;
x = y_1;
y = x_1 – (a div b) × y_1;    //div = integer division
return (d, x, y);
}
```

Numerical :

Let a = 99 and b = 78

a	b	$[a/b]$	d	x	y
99	78	1	3	– 11	14
78	21	3	3	3	– 11
21	15	1	3	– 2	3
15	6	2	3	1	– 2
6	3	2	3	0	1
3	0	–	3	1	0

i. In the 5th receive (a = 6, b = 3), values from the 6th call (b = 0) has $d_1 = 3$, $x_1 = 1$ and $y_1 = 0$. Still within the 5th call we calculate that $d = d_1 = 3$, $x = y_1 = 0$ and $y = x_1 - (a \text{ div } b) \times y_1 = 1 - 2 \times 0 = 1$.

ii. In the 4th receive (a = 15, b = 6), the values $d_1 = 3$, $x_1 = 0$ and $y_1 = 1$ from the 5th call, then compute $x = y_1 = 1$ and
$y = x_1 - (a \text{ div } b) \times y_1 = 0 - 2 \times 1 = -2$.

iii. In the 3rd receive (a = 21, b = 15), $x = -2$ and
$y = x_1 - (a \text{ div } b) \times y_1 = 1 - (1 \times -2) = 3$.

iv. In the 2nd receive (a = 78, b = 21), x = 3 and
$y = x_1 - (a \text{ div } b) \times y_1 = (-2) - 3 \times 3 = -11$.

v. In the 1st receive (a = 99, b = 78), $x = -11$ and
$y = x_1 - (a \text{ div } b) \times y_1 = 3 - 1 \times (-11) = 14$.

vi. The call EXTEUCLID (99, 78) return (3, – 11, 14), so gcd (99, 78) = 3 and gcd (99, 78) = 3 = 99 × (–11) + 78 × 14.

VERY IMPORTANT QUESTIONS

Following questions are very important. These questions may be asked in your SESSIONALS as well as UNIVERSITY EXAMINATION.

Q. 1. What is Fast Fourier Transformation and how it works ?

Ans. Refer Q. 5.1.

Q. 2. Explain the following string matching algorithms :

a. Naive string matching

b. Rabin-Karp algorithm

c. Knuth-Morris-Pratt algorithm

Ans.

a. Refer Q. 5.4.

b. Refer Q. 5.8.

c. Refer Q. 5.6.

Q. 3. Discuss the problem classes P, NP and NP-complete.
Ans. Refer Q. 5.11.

Q. 4. Differentiate NP-complete with NP-hard.
Ans. Refer Q. 5.15.

Q. 5. Show that CUQUE NP-complete.
Ans. Refer Q. 5.19.

Q. 6. Explain the following :
a. Approximation algorithm
b. Randomized algorithm

Ans.
a. Refer Q. 5.23.
b. Refer Q. 5.29.

Q. 7. Describe in detail Knuth-Morris-Pratt string matching algorithm. Compute the prefix function π for the pattern *ababbabbabbababbabb* when the alphabet is Σ = {*a*, *b*}.
Ans. Refer Q. 5.8.

☺