Design and Analysis of Algorithm

Unit-5

String Matching

String Matching Problem

- We assume that the text is an array T[1..n] of length n and that the pattern is an array P[1..m] of length m $\leq$ n.
- We further assume that the elements of P and T are characters drawn from a finite alphabet $\sum$.
- Pattern P ***occurs with shift*** s in text **T** if 0 ≤ s $\leq$ n-m and T [s+1 .. s+m] = P[1..m].
- If **P** occurs with shift s in **T** , then we call s a ***valid shift***; otherwise, we call s an ***invalid shift***.
- The ***string-matching problem*** is the problem of finding all valid shifts with which a given pattern P occurs in a given text T .

String Matching Problem

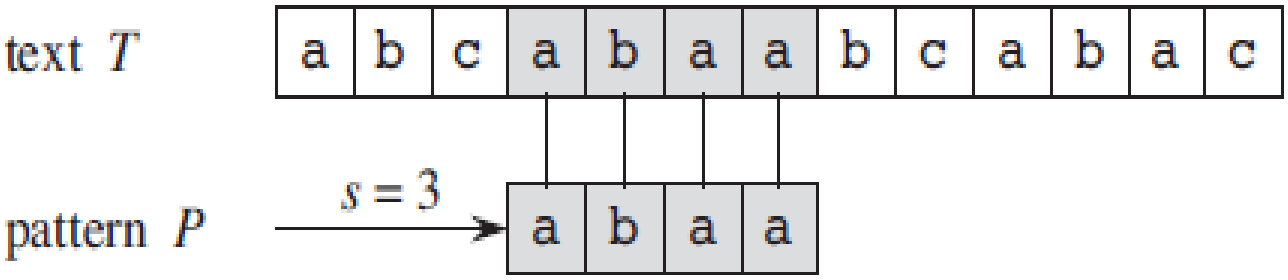
Example: Consider the text T and pattern P as following:-

T = abcabaabcabac

P= abaa

Find all valid shifts.

Solution:



Valid shift s = 3

There will be only one valid shift in this example.

Prefix and Suffix of a string

- **Prefix:** A string w is a ***prefix*** of a string x, denoted w $\sqsubset$ x,
  if x = wy for some string y $\in \sum^*$.

- **Suffix:** A string w is a ***suffix*** of a string x, denoted w $\sqsupset$ x,
  if x = yw for some string y $\in \sum^*$.

- Example: Clearly, ab $\sqsubset$ abcca and cca $\sqsupset$ abcca.

- The empty string ε is both a suffix and a prefix of every string.

The naive string-matching algorithm

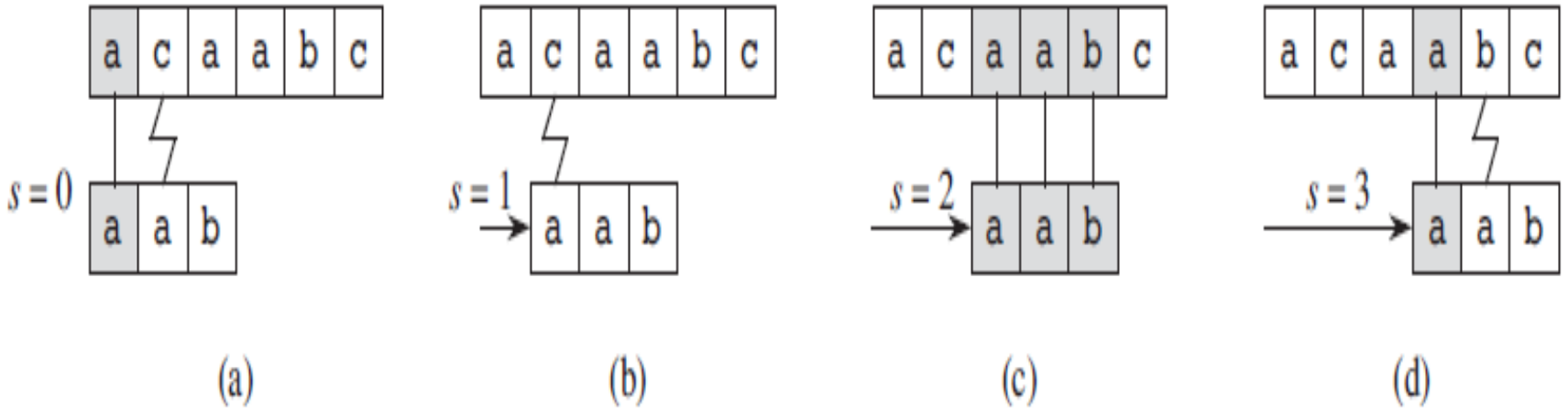
```
NAIVE-STRING-MATCHER(T, P)
1  n = T.length
2  m = P.length
3  for s = 0 to n − m
4      if P[1..m] == T[s + 1..s + m]
5          print "Pattern occurs with shift" s
```

- The worst-case running time is $\theta((n-m)m)$, which is $\theta(n^2)$ if $m = \lfloor n/2 \rfloor$.

The naive string-matching algorithm

Example: The operation of this algorithm is shown in the following:-

Here T = acaabc and P = aab



The Rabin-Karp algorithm

- Rabin and Karp proposed a string-matching algorithm that performs well.
- Given a pattern P{1..m], let p denote its corresponding decimal value. In a similar manner, given a text T[1..n], let t_s denote the decimal value of the length-m substring T [s+1 .. s+m] , for s= 0, 1,....., n-m.
- t_s = p iff T [s+1 .. s+m] = P[1..m]
- Therefore, s is a valid shift if and only if t_s = p.

The Rabin-Karp algorithm

Computation of p and t_s using Horner's rule:

- p = P[m]+10(P[m-1]+10(P[m-2]+10(P[m-3] + …. + 10(P[2]+10P[1]))))
- The value of t_0 can be computed similarly from T[1..m].
- To compute the remaining values t_1, t_2, t_3,……,t_{n-m} , t_{s+1} can be computed from t_s in the following way:-

$$t_{s+1} = 10\,(t_s - 10^{m-1}T[s+1]) + T[s+m+1] \quad \text{..... (1)}$$

- The only difficulty with this procedure is that p and t_s may be too large.

The Rabin-Karp algorithm

- To solve this problem, with d-ary alphabet {0,1,2,…., d-1}, we choose q so that dq fits with in a computer word and adjust the recurrence equation (1) to work modulo q, so that it becomes

  $$t_{s+1} = (d (t_s - T[s+1]h) + T[s+m+1]) \bmod q$$

  where $h = d^{m-1} \bmod q$

- The solution of working modulo q is not perfect, because:

$t_s \equiv p \bmod q$ does not imply that $t_s = p$. On the other hand, if $ts \not\equiv p \bmod q$, then we definitely have that $t_s \neq p$, so that shift s is invalid.

- Any shift s for which $t_s \equiv p \bmod q$ must be tested further to see whether s is really valid or we just have a ***spurious hit***. This additional test explicitly checks the condition

  P[1..m] = T[s+1……..s+m]

The Rabin-Karp algorithm

Example: Consider T and P as following:-

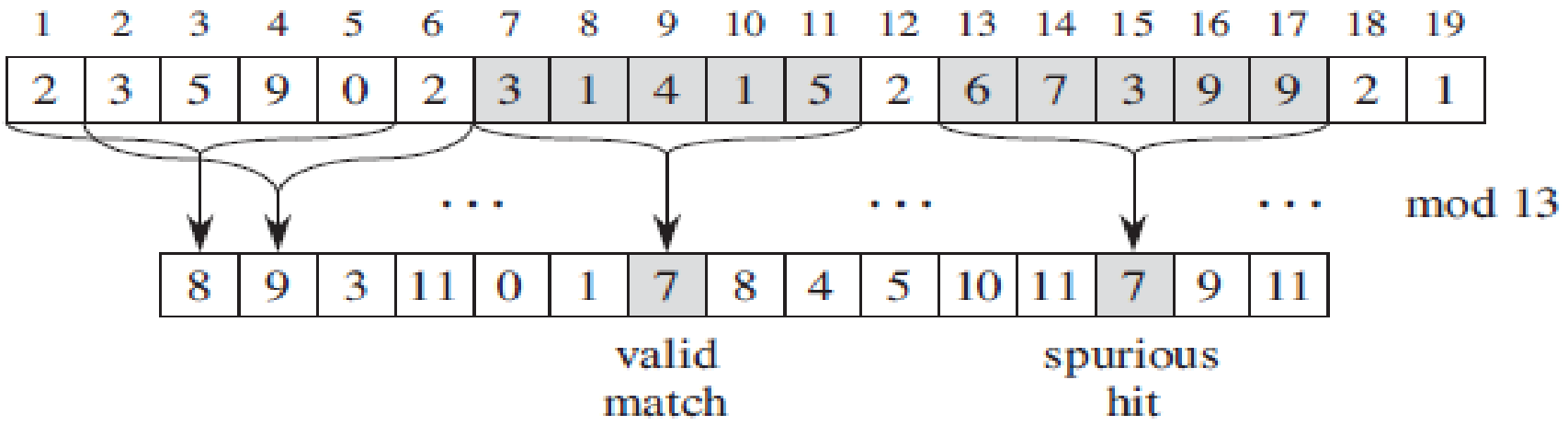
T= 2359023141526739921

P= 31415

q = 13

Find all valid shifts and spurious hit.

Solution:



The Rabin-Karp algorithm

```
RABIN-KARP-MATCHER(T, P, d, q)
 1  n = T.length
 2  m = P.length
 3  h = d^(m−1) mod q
 4  p = 0
 5  t_0 = 0
 6  for i = 1 to m                          // preprocessing
 7      p = (dp + P[i]) mod q
 8      t_0 = (dt_0 + T[i]) mod q
 9  for s = 0 to n − m                      // matching
10      if p == t_s
11          if P[1..m] == T[s + 1..s + m]
12              print "Pattern occurs with shift" s
13      if s < n − m
14          t_{s+1} = (d(t_s − T[s + 1]h) + T[s + m + 1]) mod q
```

The Rabin-Karp algorithm

Time complexity

- RABIN-KARP-MATCHER takes ,$\theta(m)$ preprocessing time, and its matching time is , $\theta((n-m+1)m)$ in the worst case.

Question: For q=11, how many spurious hits does the Robin-Karp matcher encounter in the text T = 3141592653589793 when looking for the pattern P= 26?

Solution:

Knuth-Morris-Pratt(KMP) algorithm

Prefix function for a pattern

Given a pattern P[1..m], the ***prefix function*** for the pattern P is the function $\pi : \{1,2,3,.....,m\} \rightarrow \{0,1,2,.....,m-1\}$ such that

$$\pi(q) = \max\{ k \mid K < q \text{ and } P_k \sqsupset P_q\}$$

$\pi(q)$ is the length of the longest prefix of P that is a proper suffix of P_q.

Knuth-Morris-Pratt(KMP) algorithm

Example: Compute the prefix function of the pattern

P = abababababca

Solution:

	1	2	3	4	5	6	7	8	9	10
i	a	b	a	b	a	b	a	b	c	a
P(i)	0	0	1	2	3	4	5	6	0	1
π(i)										

Knuth-Morris-Pratt(KMP) algorithm

```
KMP-MATCHER(T, P)
 1  n = T.length
 2  m = P.length
 3  π = COMPUTE-PREFIX-FUNCTION(P)
 4  q = 0                                  // number of characters matched
 5  for i = 1 to n                         // scan the text from left to right
 6      while q > 0 and P[q + 1] ≠ T[i]
 7          q = π[q]                       // next character does not match
 8      if P[q + 1] == T[i]
 9          q = q + 1                      // next character matches
10      if q == m                          // is all of P matched?
11          print "Pattern occurs with shift" i − m
12          q = π[q]                       // look for the next match
```

Knuth-Morris-Pratt(KMP) algorithm

```
COMPUTE-PREFIX-FUNCTION(P)
 1  m = P.length
 2  let π[1..m] be a new array
 3  π[1] = 0
 4  k = 0
 5  for q = 2 to m
 6      while k > 0 and P[k + 1] ≠ P[q]
 7          k = π[k]
 8      if P[k + 1] == P[q]
 9          k = k + 1
10      π[q] = k
11  return π
```

Knuth-Morris-Pratt(KMP) algorithm

Time complexity

Running time of compute-prefix-function is $\theta(m)$.
The matching time of KMP-Matcher is $\theta(n)$.

Question: Consider text and pattern as following:-

T = bacbababaabcbab

P = aba

Find all valid shifts using KMP algo.

Question: Compute the prefix function for the pattern ababbabbabbababbabb.

Knuth-Morris-Pratt(KMP) algorithm

Prefix function for a pattern

Given a pattern P[1..m], the ***prefix function*** for the pattern P is the function $\pi : \{1,2,3,.....,m\} \rightarrow \{0,1,2,.....,m-1\}$ such that

$$\pi(q) = \max\{ k \mid K < q \text{ and } P_k \sqsupset P_q\}$$

$\pi(q)$ is the length of the longest prefix of P that is a proper suffix of P_q.

Knuth-Morris-Pratt(KMP) algorithm

Example: Compute the prefix function of the pattern

P = abababababca

Solution:

	1	2	3	4	5	6	7	8	9	10
i	a	b	a	b	a	b	a	b	c	a
P(i)	0	0	1	2	3	4	5	6	0	1
π(i)										

Knuth-Morris-Pratt(KMP) algorithm

```
KMP-MATCHER(T, P)
 1  n = T.length
 2  m = P.length
 3  π = COMPUTE-PREFIX-FUNCTION(P)
 4  q = 0                                  // number of characters matched
 5  for i = 1 to n                         // scan the text from left to right
 6      while q > 0 and P[q + 1] ≠ T[i]
 7          q = π[q]                       // next character does not match
 8      if P[q + 1] == T[i]
 9          q = q + 1                      // next character matches
10      if q == m                          // is all of P matched?
11          print "Pattern occurs with shift" i − m
12          q = π[q]                       // look for the next match
```

Knuth-Morris-Pratt(KMP) algorithm

```
COMPUTE-PREFIX-FUNCTION(P)
 1  m = P.length
 2  let π[1..m] be a new array
 3  π[1] = 0
 4  k = 0
 5  for q = 2 to m
 6      while k > 0 and P[k + 1] ≠ P[q]
 7          k = π[k]
 8      if P[k + 1] == P[q]
 9          k = k + 1
10      π[q] = k
11  return π
```

Knuth-Morris-Pratt(KMP) algorithm

Time complexity

Running time of compute-prefix-function is $\theta(m)$.
The matching time of KMP-Matcher is $\theta(n)$.

Question: Consider text and pattern as following:-

T = bacbababaabcbab

P = aba

Find all valid shifts using KMP algo.

Question: Compute the prefix function for the pattern ababbabbabbababbabb.

AKTU Examination Questions

1. Write an algorithm for Naïve string matcher.
2. Write KMP algorithm for string matching. Perform the KMP algorithm to search the occurrences of the pattern abaab in the text string abbabaabaabab.
3. Write Rabin Karp string matching algorithm. Working modulo q=11, how many spurious hits does the Rabin karp matcher in the text T= 3141592653589793, when looking for the pattern P=26.
4. Explain and Write the Knuth-Morris-Pratt algorithm for pattern matching also write its time complexity.
5. Describe in detail Knuth-Morris-Pratt string matching algorithm. Compute the prefix function π for the pattern ababbabbabbababbabb when the alphabet is $\Sigma = \{a,b\}$.
6. Compute the prefix function π for the pattern P= a b a c a b using KNUTHMORRIS –PR

Approximation Algorithms

Approximation Algorithms

- An algorithm that returns near-optimal solutions is said to be approximation algorithm.
- This technique does not guarantee the best solution.
- The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at the most polynomial time.
- Approximation algorithms are designed to get the solution of NP-complete problems in polynomial time.
- If we work on an optimization problem where every solution carries a cost, then an approximation algorithm returns a legal solution, but the cost of that legal solution may not be optimal.

Approximation Ratio

- Let C be the cost of the solution returned by an approximate algorithm, and C* is the cost of the optimal solution.
- An algorithm for a problem has an **approximation ratio** of P(n) for any input of size n, if the cost C of the solution produced by the algorithm is within a factor of P(n) of the cost C* of an optimal solution i.e.

  $$\mathbf{max(C/C^*, C^*/C) \leq P(n)}$$

- If an algorithm achieves an approximation ratio of P(n), we call it a **P(n)-approximation algorithm**.

Approximation Ratio(cont.)

- The approximation ratio measures how bad the approximate solution is distinguished with the optimal solution. A large (small) approximation ratio measures the solution is much worse than (more or less the same as) an optimal solution.
- Observe that P(n) is always ≥ 1, if the ratio does not depend on n, we may write P. Therefore, a 1-approximation algorithm gives an optimal solution.
- For a minimization problem, $0 < C \leq C^*$, and the ratio C^*/C gives the factor by which the cost of the optimal solution is larger than the cost of approximate solution.
- Similarly, for a minimization problem, $0 < C^* \leq C$, and the ratio C/C^* gives the factor by which the cost of the approximate solution is larger than the cost of optimal solution.

Vertex Cover Problem

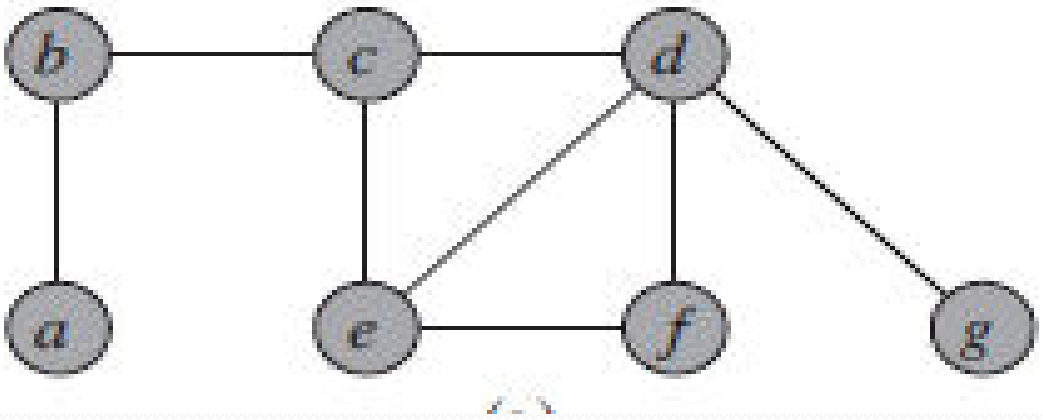
- A **vertex cover** of an undirected graph G =(V, E) is a subset V' $\subseteq$ V such that if (u,v) is an edge of G, then either u $\in$ V' or v $\in$ V' (or both).
- The size of a vertex cover is the number of vertices in it.
- The **vertex-cover problem** is to find a vertex cover of minimum size in a given undirected graph. We call such a vertex cover an optimal vertex cover.
- This problem is the optimization version of an NP-complete decision problem.

Approximation Algorithm for Vertex Cover Problem

```
APPROX-VERTEX-COVER(G)
1  C = ∅
2  E' = G.E
3  while E' ≠ ∅
4      let (u, v) be an arbitrary edge of E'
5      C = C ∪ {u, v}
6      remove from E' every edge incident on either u or v
7  return C
```

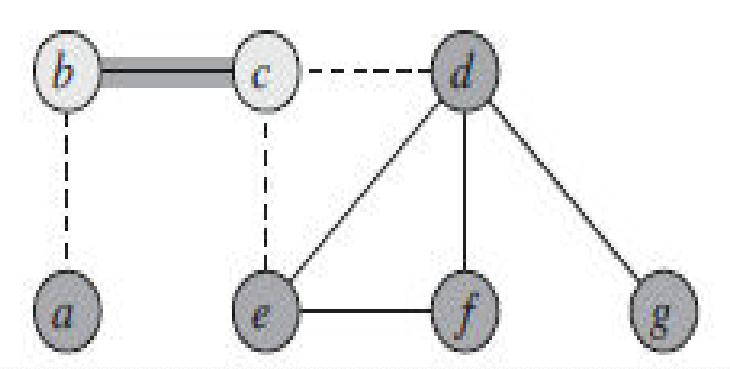
Approximation Algorithm for Vertex Cover Problem

Ex. Consider the following graph:-

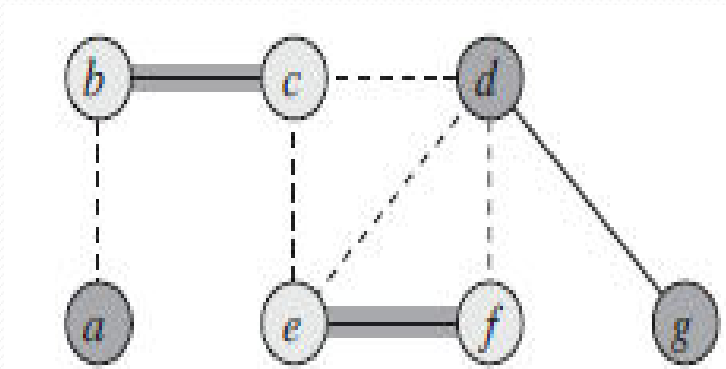


Find the optimal vertex cover of this graph.

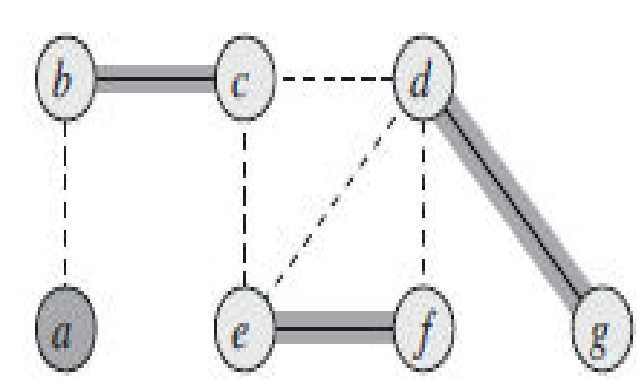
Solution



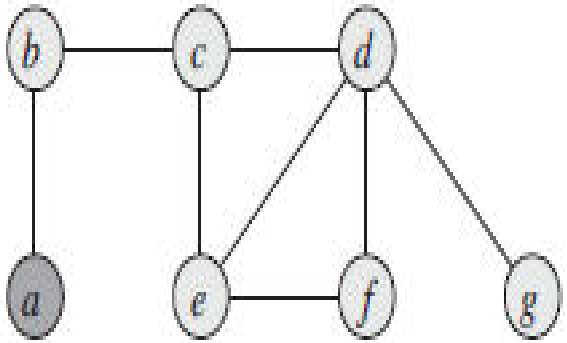
(a)



(b)

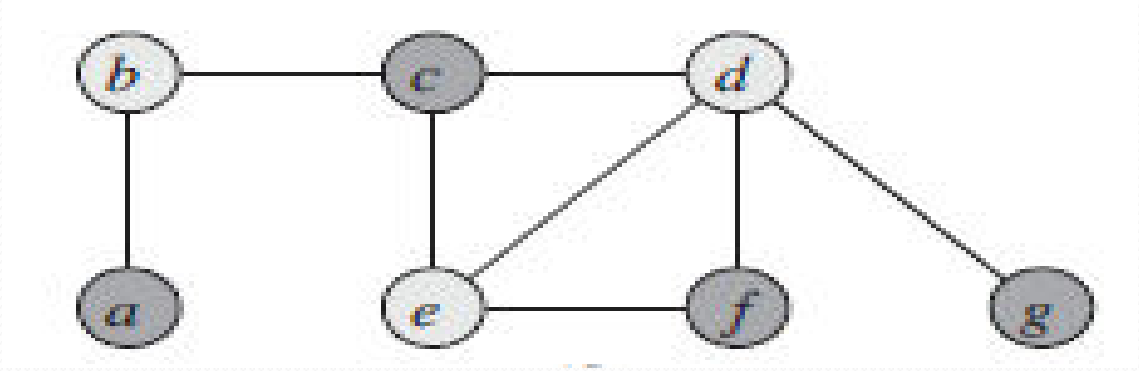


(c)



(d)

Approximation Algorithm for Vertex Cover Problem



Optimal vertex cover for this problem contains only three vertices: b, d, and e.

Note: The running time of this algorithm is O(E+V) using adjacency lists to represent E'.

Traveling-salesman problem

- In the traveling-salesman problem, we are given a complete undirected graph G(V,E) that has a nonnegative integer cost c(u,v) associated with each edge (u,v) ∈ E, and we must find a Hamiltonian cycle (a tour) of G with minimum cost.
- Let c(A) denote the total cost of the edges in the subset $A \subseteq E$.

$$c(A) = \sum_{(u,v) \in A} c(u,v)$$

- Cost function c satisfies the ***triangle inequality*** *if, for all vertices* u,v,w ∈*V,*

$$c(u,v) \leq c(u,v) + c(v,w)$$

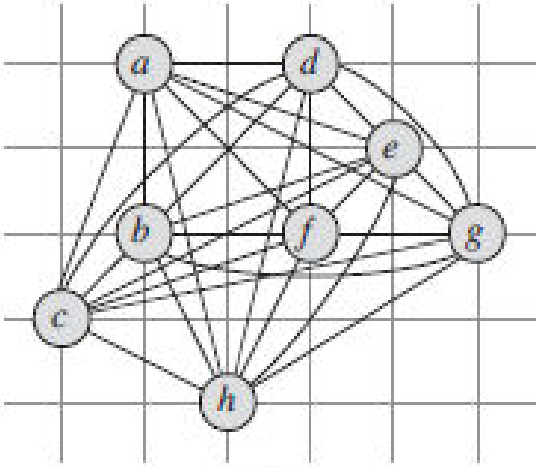
Traveling-salesman problem with the triangle inequality

- The following algorithm computes a near-optimal tour of an undirected graph G, using the minimum-spanning-tree algorithm MST-PRIM.
- Here, the cost function satisfies the triangle inequality.
- The tour that this algorithm returns is no worse than twice as long as an optimal tour.

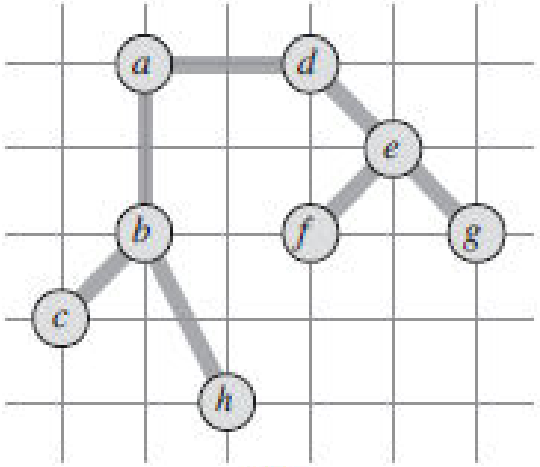
APPROX-TSP-TOUR(G, c)

1 select a vertex $r \in G.V$ 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 H be a list of vertices, ordered according to when they are first visited
  in a preorder tree walk of T
4 return the hamiltonian cycle H

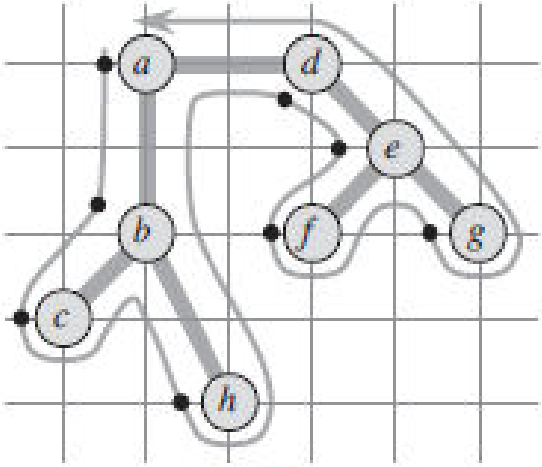
Traveling-salesman problem with the triangle inequality



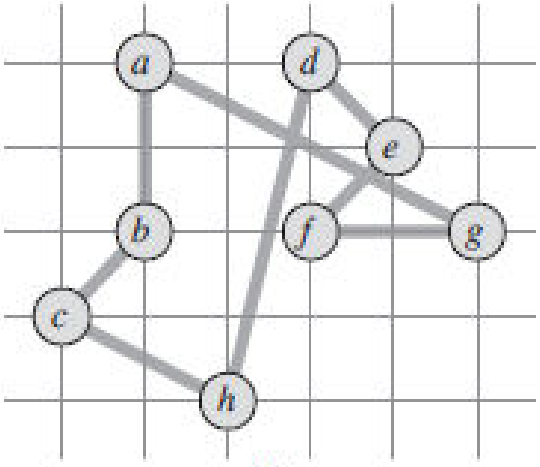
(a)



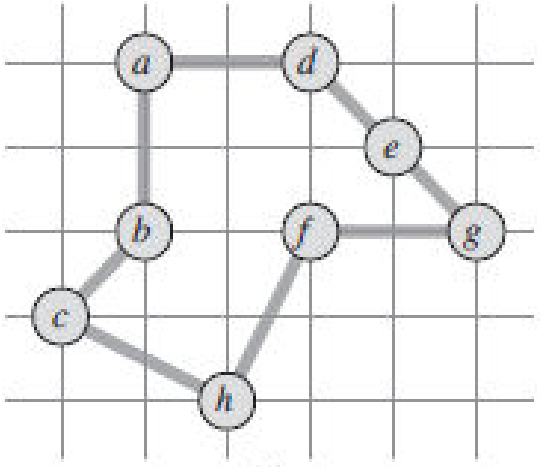
(b)



(c)



(d)



(e)

Traveling-salesman problem with the triangle inequality

(a) **A complete undirected graph. Vertices lie** on intersections of integer grid lines. For example, f is one unit to the right and two units up from h. The cost function between two points is the ordinary euclidean distance.

(b) **A minimum spanning** tree T of the complete graph, as computed by MST-PRIM. Vertex a is the root vertex. Only edges in the minimum spanning tree are shown. The vertices happen to be labeled in such a way that they are added to the main tree by MST-PRIM in alphabetical order.

(c) **A walk of T , starting at a. A** full walk of the tree visits the vertices in the order a, b, c, b, h, b, a, d, e, f, e, g, e, d, a. A preorder walk of T lists a vertex just when it is first encountered, as indicated by the dot next to each vertex, yielding the ordering a, b, c, h, d, e, f, g.

Traveling-salesman problem with the triangle inequality

(d) A tour obtained by visiting the vertices in the order given by the preorder walk, which is the tour H returned by APPROX-TSP-TOUR. Its total cost is approximately 19.074.

(e) An optimal tour H for the original complete graph. Its total cost is approximately 14.715.

Note: The running time of APPROX-TSP-TOUR is $O(V^2)$.

NP-Completeness

Classes P and NP

- The class P consists of those problems that are solvable in polynomial time.
- More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem.
- The class NP consists of those problems that are "verifiable" in polynomial time.
- What do we mean by a problem being verifiable? If we were somehow given a "certificate" of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the problem.

Classes P and NP

- For example, in the hamiltonian cycle problem, given a directed graph G(V,E), a certificate would be a sequence $< v_1, v_2, v_3, \ldots\ldots, v_n >$ of n vertices. We could easily check in polynomial time that $(v_i, v_{i+1}) \in E$ for i = 1, 2, 3,……, n-1 and that $(v_n, v_1) \in E$ as well.

- Any problem in P is also in NP, since if a problem is in P then we can solve it in polynomial time without even being supplied a certificate.

NP-Hard and NP-complete(NPC)

NP-Hard: A problem L is said to be NP-hard if every problems belong in to NP is polynomial time reducible to L.

That is,

Problem L is NP-hard if for all problems L' ϵ NP,

$$L' \leq_p L.$$

That is, if we can solve L in polynomial time, then we can solve all NP problems in polynomial time.

NP-Complete

NP-Complete: Problem L is NP-complete if

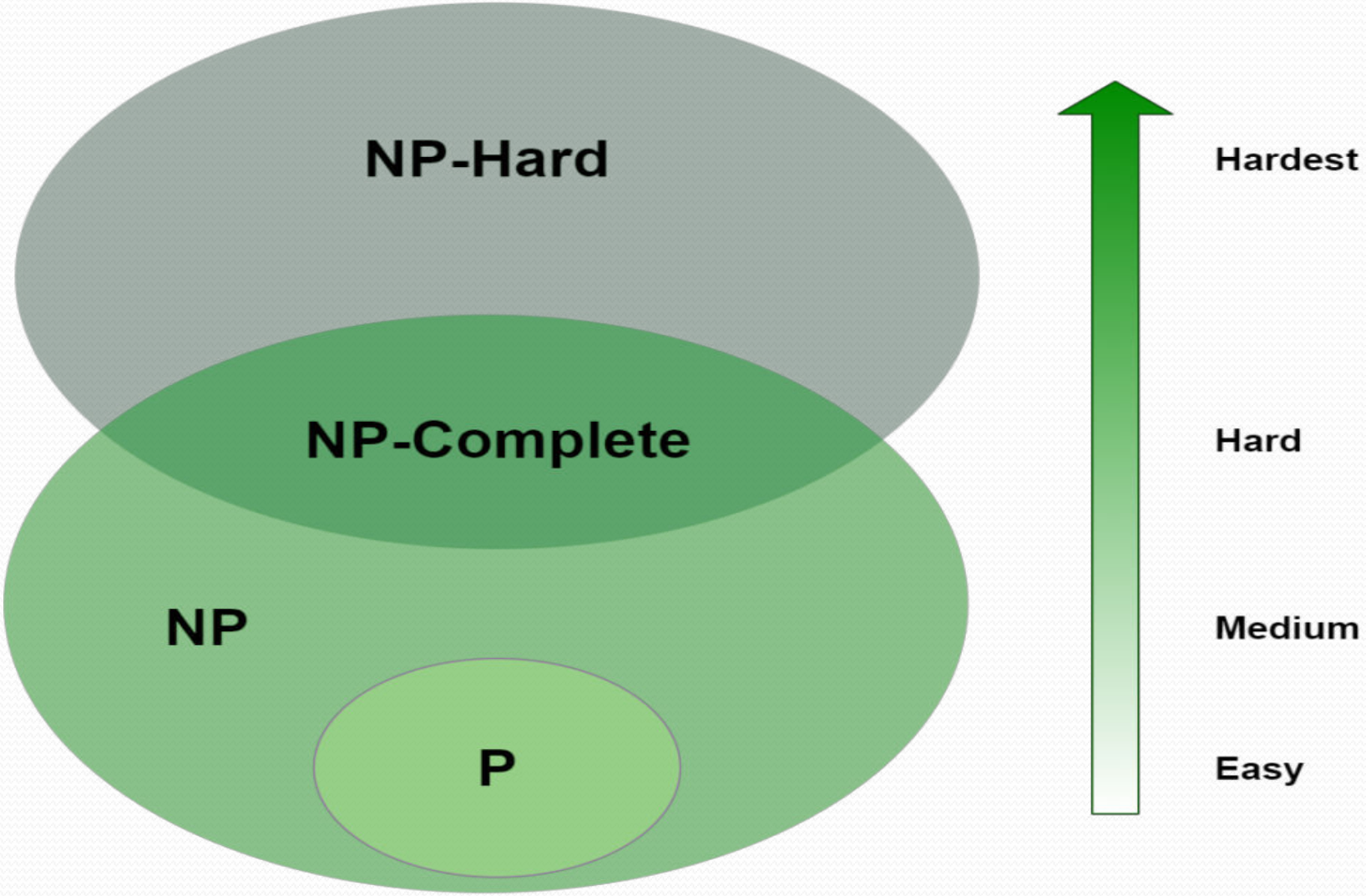
1. L ϵ NP and
2. L is NP-hard

NP-Complete problems:

- Boolean satisfiability problem (SAT)
- Hamiltonian cycle problem.
- Travelling salesman problem
- Vertex cover problem

Note: If any NP-complete problem is solvable in polynomial time, then every NP-Complete problem is also solvable in polynomial time.

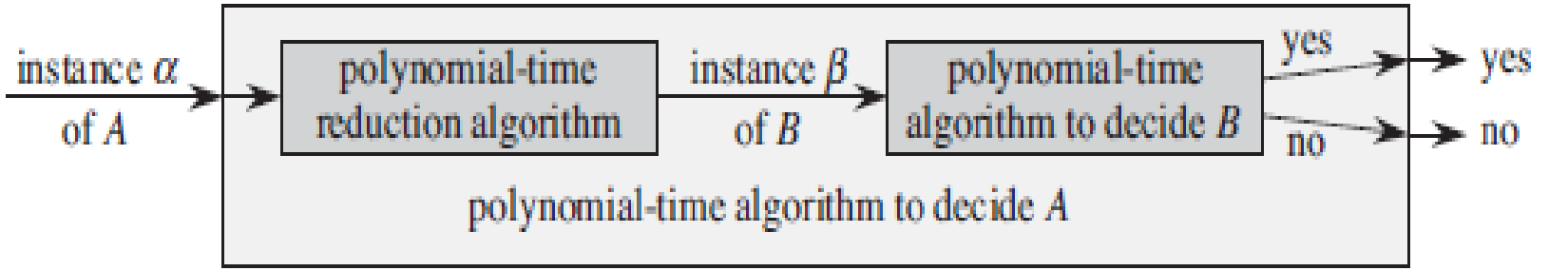
Relationship between P, NP, NP-hard and NPC



Polynomial-time ***Reduction Algorithm***

- Consider a decision problem A, which we would like to solve in polynomial time.
- We call the input to a particular problem an ***instance of that problem.***
- Suppose that we already know how to solve a different decision problem B in polynomial time.
- Finally, suppose that we have a procedure that transforms any instance α of A into some instance β of B with the following characteristics:
  - The transformation takes polynomial time.
  - The answers are the same. That is, the answer for α is "yes" if and only if the answer for β is also "yes."
- We call such a procedure a **polynomial-time *reduction algorithm.***

Polynomial-time ***Reduction Algorithm(cont.)***



Polynomial-time reduction algorithm provides us a way to solve problem A in polynomial time:

1. Given an instance α of problem A, use a polynomial-time reduction algorithm to transform it to an instance β of problem B.
2. Run the polynomial-time decision algorithm for B on the instance β.
3. Use the answer for β as the answer for α.

Lemma

If L is a language such that $L' \leq_P L$ for some $L' \in NPC$, then L is NP-hard. If, in addition, $L \in NP$, then $L \in NPC$.

Note: The hamiltonian cycle problem is NP-complete.

Note: Satisfiability of boolean formulas in 3-conjunctive normal form is NP-complete.

Note: The vertex-cover problem is NP-complete.

Theorem: Show that the traveling-salesman problem is NP-complete.

Proof:

We first show that TSP belongs to NP.

Given an instance of the problem, we use as a certificate the sequence of n vertices in the tour. The verification algorithm checks that this sequence contains each vertex exactly once, sums up the edge costs, and checks whether the sum is at most k. This process can certainly be done in polynomial time.

To prove that TSP is NP-hard, we show that HAM-CYCLE$\leq_p$TSP.

Let G=(V,E) be an instance of HAM-CYCLE. We construct an instance of TSP as follows.

We form the complete graph G'=(V,E'), where

$E' = \{(i,j) : i,j \in V \text{ and } i \neq j\}$ and

we define the cost function c by:-

$c(i,j) = 0 \quad \text{if} \quad (i,j) \in E$

$= 1 \quad \text{if} \quad (i,j) \notin E$

The instance of TSP is then $< G', c, 0 >$, which we can easily create in polynomial time.

We now show that graph G has a hamiltonian cycle if and only if graph G' has a tour of cost at most 0.

Suppose that graph G has a hamiltonian cycle h. Each edge in h belongs to E and thus has cost 0 in G'. Thus, h is a tour in G' with cost 0. Conversely, suppose that graph G' has a tour h' of cost at most 0. Since the costs of the edges in E' are 0 and 1, the cost of tour h' is exactly 0 and each edge on the tour must have cost 0. Therefore, h' contains only edges in E. We conclude that h' is a hamiltonian cycle in graph G.

AKTU Examination Questions

1. Write and explain the algorithm to solve vertex cover problem using approximation algorithm.
2. Explain NP-complete and NP-Hard.
3. Explain Randomized algorithm in brief.
4. What is an approximation algorithm? What is meant by P(n) approximation algorithms? Discuss approximation algorithm for Travelling Salesman Problem.
5. 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 Examination Questions

6. What are approximation algorithms? What is meant by P(n) approximation algorithms?
7. Define NP, NP hard and NP Complete Problems. Prove that Travelling Salesman Problem is NP-Complete.
8. What is the application of Fast Fourier Transform (FFT)? Also write the recursive algorithm for FFT.
9. Discuss the problem classes P, NP and NP –complete with class relationship.
10. Explain Approximation and Randomized algorithms.
11. What do you mean by polynomial time reduction?
12. Explain applications of FFT.