278 DAA Complete - sub

Design and analysis of algorithm (Lovely Professional University)

Scan to open on Studocu

Studocu is not sponsored or endorsed by any college or university
Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

DESIGN AND ANALYSIS OF ALGORITHM

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

LECTURE NOTES

ON

DESIGN AND ANALYSIS OF ALGORITHMS

Prepared by

Dr. Subasish Mohapatra

Department of Computer Science and Application
College of Engineering and Technology, Bhubaneswar
Biju Patnaik University of Technology, Odisha

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

CONTENTS:

MODULE – I
Lecture 1 - Introduction to Design and analysis of algorithms
Lecture 2 - Growth of Functions (Asymptotic notations)
Lecture 3 - Recurrences, Solution of Recurrences by substitution
Lecture 4 - Recursion tree method
Lecture 5 - Master Method
Lecture 6 - Worst case analysis of merge sort, quick sort and binary search
Lecture 7 - Design and analysis of Divide and Conquer Algorithms
Lecture 8 - Heaps and Heap sort
Lecture 9 - Priority Queue
Lecture 10 - Lower Bounds for Sorting
MODULE-II
Lecture 11 - Dynamic Programming algorithms
Lecture 12 - Matrix Chain Multiplication
Lecture 13 - Elements of Dynamic Programming
Lecture 14 - Longest Common Subsequence
Lecture 15 - Greedy Algorithms
Lecture 16 - Activity Selection Problem
Lecture 17 - Elements of Greedy Strategy
Lecture 18-19 - Fractional Knapsack Problem
Lecture 20 - Huffman Codes
Lecture – 21 Disjoint Set Data Structure
Lecture 22 - Disjoint Set Operations, Linked list Representation
Lecture 23 - Disjoint Forests
MODULE – III
Lecture 24 - Graph Algorithm - BFS and DFS
Lecture 25 - Minimum Spanning Trees
Lecture 26 - Kruskal algorithm
Lecture 27 - Prim's Algorithm
Lecture 28 -30 Single Source Shortest paths, Dijkstra's Algorithm
Lecture 31 – All pairs shortest paths algorithms.
Lecture 32 - Backtracking And Branch And Bound
Lecture 33 - Fourier transforms and Rabin-Karp Algorithm
Lecture 34 - NP-Hard and NP-Complete Problems
Lecture 35 - Approximation Algorithms(Vertex-Cover Problem)
Lecture 36 - NP-Complete Problems (without proofs)
Lecture 37 - Traveling Salesman Problem

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Module-I

Lecture 1 - Introduction to Design and analysis of algorithms

MOTIVATION

The advancement in science and technology enhance the performance of processor, which proportionally affect the characteristics of computer system, such as security, scalability and reusability. Important problems such as sorting, searching, string processing, graph problems, Combinational problems, numerical problems are basic motivations for designing algorithm.

DESIGN GOAL

The Basic objective of solving problem with multiple constraints such as problem size performance and cost in terms of space and time. The goal is to design fast, efficient and effective solution to a problem domain. Some problems are easy to solve and some are hard. Quite cleverness is required to design solution with fast and better approach. Designing new system need a new technology and background of the new technology is the enhancement of existing algorithm. The study of algorithm is to design efficient algorithm not only limited in reducing cost and time but to enhance scalability, reliability and availability.

The main concern of the course ensures:

i) Correctness of solution
ii) Decomposition of application into small and clear units which can be maintained precisely
iii) Improving the performance of application

INTRODUCTION TO ANALYSIS OF ALGORITHM

A lay man perceives that a computer perform anything and everything. It is very difficult to ensure that it is not really the computer but the man behind computer who does the whole thing.

For example users just enter their queries and can get information as he/she desire. A common man rarely understands that a man made procedure called search has done the entire task and the only support provided by the computer is the execution speed and organized storage information.

'Algorithm' is defined after the name of **Abu Ja' far Muhammad ibn Musa Al-Khwarizmi**, Ninth century , **al-jabr** *means* **"restoring"** *referring to the process of moving a subtracted quantity to other side of an equation;* **al-muqabala** *is* **"comparing"** *and refers to subtracting equal quantities from both sides of an equation.*

Definition of Algorithm

- An algorithm is a set of rules for carrying out calculation either by hand or on a machine.
- An algorithm is a sequence of computational steps that transform the input into the output.
- An algorithm is a sequence of operations performed on data that have to be organized in the data structures.
- A finite set of instruction that specify a sequence of operations to be carried out in order to solve a specific problem or class of problems is called an algorithm.
- An algorithm is an abstraction of a program to be executed on a physical machine (model computation).

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

- An algorithm is defined as set of instructions to perform a specific task within finite no. of steps.
- Algorithm is defined as a step by step procedure to perform a specific task within finite number of steps.
- It can be defined as a sequence of definite and effective instructions, while terminates with the production of correct output from the given input.

Example of algorithm

Let us assume can common example of ATM withdrawal system .To present the scenario of a person under goes the following steps.

1. SWAP the card
2. Enter the PIN
3. Put the amount
4. Withdraw amount
5. Check balance
6. Cancel/clear.

If we go through these above 6 steps without considering the statement of the problem, we should assume that this is the algorithm for clearing the toilet. As of several ambiguities arises there while comprehending every step. The step 1 may imply toothbrush, paintbrush, toilet brush etc. Such an ambiguous doesn't an instruction an algorithmic step. Thus every step should be made unambiguous step is called '**definite instruction'**. Even if the step 2 is rewritten as apply the toothpaste, to eliminate ambiguities yet the conflicts such as, where to apply the toothpaste and where the source of the toothpaste is, need to be resolved. Therefore, the act of applying the toothpaste is not mentioned. Although unambiguous, such unrealizable steps can't be included as algorithmic instruction as they are not effective.

The definiteness and effectiveness of an instruction implies the successful termination of that instruction. Hence that above two may not be sufficient to guarantee the termination of the algorithm. Therefore, while designing an algorithm care should be taken to provide a proper termination for algorithm.

CHARACTERISTICS OF AN ALGORITHM

Every algorithm should have the following five characteristic features:

a) Input
b) Output
c) Finiteness
d) Definiteness
e) Effectiveness
f) Precision
g) Determination
h) Correctness
i) Generality

a) Input

An algorithm may have one or more inputs. The inputs are taken from a specified set of subjects. The input pattern may be texts, images or any type of files.

b) Output

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

An algorithm may have one or more outputs. Output is basically a quantity which has a specified relation with the input.

c) Finiteness

An algorithm should terminate after a countable number of steps. In some cases the repetition of steps may be larger in number. If a procedure is able to resolve in finite number of execution of steps, then it is referred to be computational method.

d) Definiteness

Each step of an algorithm must be precisely defined. The action to be carried out must be on ambiguously specified for each case. Due to the lack of understandability one may think that the step might be lacking definiteness. Therefore in such cases mathematically expressions are written, so that it resembles the instruction of any computer language.

e) Effectiveness

An algorithm is generally expected to effective. Means the steps should be sufficiently basic, so that it may be possible for a man to resolve them.

f) Precision

The steps are precisely stated.

g) Determination

The intermediate results of each step of execution are unique and are determined only by input and output of the preceding steps.

h) Correctness

Output produced by the algorithm should correct.

i) Generality

Algorithm applies to set of standard inputs.

What is Computer algorithm?

‘’a set of steps to accomplish or complete a task that is described precisely enough that a computer can run it’’.

Described precisely: very difficult for a machine to know how much water, milk to be added etc. in the above tea making algorithm.

These algorithms run on computers or computational devices. for example, GPS in our smart phones, Google hangouts.

GPS uses shortest path algorithm. Online shopping uses cryptography which uses RSA algorithm.

Characteristics of an algorithm:-

Must take an input.

Must give some output(yes/no, value etc.)

Definiteness –each instruction is clear and unambiguous.

Finiteness –algorithm terminates after a finite number of steps.

Effectiveness –every instruction must be basic i.e. simple instruction.

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Expectation from an algorithm

Correctness:-

Correct: Algorithms must produce correct result.

Produce an incorrect answer: Even if it fails to give correct results all the time still there is a control on how often it gives wrong result. Eg. Rabin- Miller PrimalityTest (Used in RSA algorithm): It doesn't give correct answer all the time.1 out of 250 times it gives incorrect result.

Approximation algorithm: Exact solution is not found, but near optimal solution can be found out. (Applied to optimization problem.)

Less resource usage:

Algorithms should use less resources (time and space).

Resource usage:

Here, the time is considered to be the primary measure of efficiency .We are also concerned with how much the respective algorithm involves the computer memory.But mostly time is the resource that is dealt with. And the actual running time depends on a variety of backgrounds: like the speed of the Computer, the language in which the algorithm is implemented, the compiler/interpreter, skill of the programmers etc.

So, mainly the resource usage can be divided into: 1.Memory (space) 2.Time

Time taken by an algorithm?

performance measurement or Apostoriori Analysis: Implementing the algorithm in a machine and then calculating the time taken by the system to execute the program successfully.

Performance Evaluation or Apriori Analysis. Before implementing the algorithm in a system. This is done as follows

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

How long the algorithm takes :-will be represented as a function of the size of the input.

f(n)→how long it takes if 'n' is the size of input.

How fast the function that characterizes the running time grows with the input size.

"Rate of growth of running time".

The algorithm with less rate of growth of running time is considered better.

How algorithm is a technology?

Algorithms are just like a technology. We all use latest and greatest processors but we need to run implementations of good algorithms on that computer in order to properly take benefits of our money that we spent to have the latest processor. Let's make this example more concrete by pitting a faster computer(computer A) running a sorting algorithm whose running time on n values grows like n2 against a slower computer (computer B) running assorting algorithm whose running time grows like n lg n. They each must sort an array of 10 million numbers. Suppose that computer A executes 10 billion instructions per second (faster than any single sequential computer at the time of this writing) and computer B executes only 10 million instructions per second, so that computer A is1000 times faster than computer B in raw computing power. To make the difference even more dramatic, suppose that the world's craftiest programmer codes in machine language for computer A, and the resulting code requires 2n2 instructions to sort n numbers. Suppose further that just an average programmer writes for computer B, using a high- level language with an inefficient compiler, with the resulting code taking 50n lg n instructions.

Computer A (Faster) Computer B(Slower)

Running time grows like n2. Grows inkling.

10 billion instructions per sec. 10million instruction per sec

2n2 instruction. 50 long instruction.

Time taken= $\frac{2\times(10^7)^2}{10^{10}} = 20{,}000$ $\frac{50\times10^7\times\log 10^7}{10^7} \approx 1163$

It is more than 5.5hrs it is under 20 mins.

So choosing a good algorithm (algorithm with slower rate of growth) as used by computer B affects a lot.

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 2-Growth of Functions (Asymptotic notations)

Before going for growth of functions and asymptotic notation let us see how to analyses an algorithm.

How to analyses an Algorithm

Let us form an algorithm for Insertion sort (which sort a sequence of numbers).The pseudo code for the algorithm is give below.

Pseudo code:

for j=2 to A length ---------- C1

key=A[j]----------C2

//Insert A[j] into sorted Array A[1.....j-1]----------C3 i=j-1----------C4

while i>0 & A[j]>key----------C5

A[i+1]=A[i]----------C6

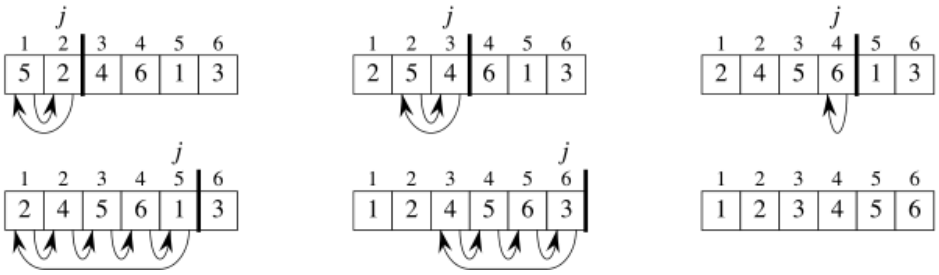
i=i-1----------C7

A[i+1]=key----------C8

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Example:



Let Ci be the cost of ith line. Since comment lines will not incur any cost C3=0. Cost No. Of times Executed

C1n C2 n-1

C3=0 n-1 C4n-1 C5 $\sum_{j=2}^{n-1} t_j$

C6 $\sum_{j=2}^{n} (t_j -$) 1

C7 $\sum_{j=2}^{n} (t_j - 1)$ C8n-1

Running time of the algorithm is:

T(n)=C1n+C2(n-1)+0(n-1)+C4(n-1)+C5$(\sum_{j=2}^{n-1} t_j)$+C6($\sum_{j=2}^{n} t_j -$)+C7($\sum_{j=2}^{n} t_j -$)+ C8(n-1)

Best case:

It occurs when Array is sorted.

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

All devalues are 1.

T(n)=C1n+C2(n-1)+0 (n-1)+C4(n-1)+C5 $(\sum_{j=2}^{n-1} 1)$+C6($\sum_{j=2}^{n} 0$)+C7($\sum_{j=2}^{n} 0$)+ C8(n-1)

=C1n+C2 (n-1) +0 (n-1) +C4 (n-1) +C5 $(n-1)$+ C8 (n-1)

= (C1+C2+C4+C5+ C8) n-(C2+C4+C5+ C8)

Which is of the forman+b?

Linear function of n. so, linear growth.

Worst case: It occurs when Array is reverse sorted, and tj =j

T(n)=C1n+C2(n-1)+0 (n-1)+C4(n-1)+C5 $(\sum_{j=2}^{n-1} j)$+C6($\sum_{j=2}^{n} j -$)+C7($\sum_{j=2}^{n} j -$)+C8(n-1)

=C1n+C2(n-1)+C4(n-1)+C5 $(\frac{n(n-1)}{2} - 1)$+C6($\sum_{j=2}^{n} \frac{n(n-1)}{2}$)+C7($\sum_{j=2}^{n} \frac{n(n-1)}{2}$)+ C8(n-1)

which is of the form an2+bn+c

Quadratic function. So in worst case insertion set grows in n2. Why we concentrate on worst-case running time?

The worst-case running time gives a guaranteed upper bound on the running time for any input.

For some algorithms, the worst case occurs often. For example, when searching, the worst case often occurs when the item being searched for is not present, and searches for absent items may be frequent.

Why not analyze the average case? Because it's often about as bad as the worst case.

Order of growth:

It is described by the highest degree term of the formula for running time. (Drop lower-order terms. Ignore the constant coefficient in the leading term.)

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Example: We found out that for insertion sort the worst-case running time is of the form an2 + bn + c.

Drop lower-order terms. What remains is an2.Ignore constant coefficient. It results in n2.But we cannot say that the worst-case running time T(n) equals n2 .Rather It grows like n2 . But it doesn't equal n2.We say that the running time is Θ (n2) to capture the notion that the order of grow this n2.

We usually consider one algorithm to be more efficient than another if its worst-case running time has a smaller order of growth.

Asymptotic notation

It is a way to describe the characteristics of a function in the limit.

It describes the rate of growth of functions.

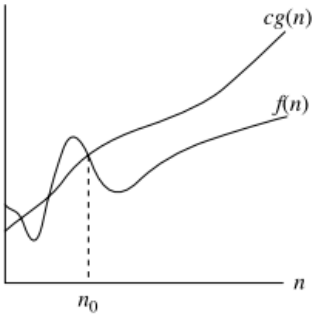
Focus on what's important by abstracting away low-order terms and constant factors.

It is a way to compare "sizes" of functions: O≈ ≤

Ω≈ ≥ Θ ≈ = o ≈ < ω ≈ >

O-notation

$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$.



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Example: $2n^2 = O(n^3)$, with $c = 1$ and $n_0 = 2$.

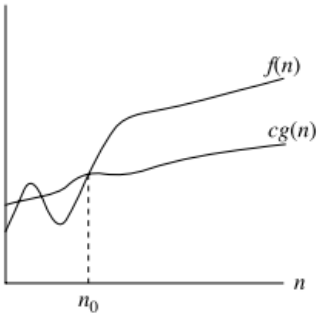
Examples of functions in $O(n^2)$:

n^2
$n^2 + n$
$n^2 + 1000n$
$1000n^2 + 1000n$
Also,
n
$n/1000$
$n^{1.99999}$
$n^2/\lg\lg\lg n$

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Ω-notation

$\Omega(g(n)) = \{f(n) :$ there exist positive constants c and n_0 such that
$0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$.



$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Example: $\sqrt{n} = \Omega(\lg n)$, with $c = 1$ and $n_0 = 16$.

Examples of functions in $\Omega(n^2)$:

n^2
$n^2 + n$
$n^2 - n$
$1000n^2 + 1000n$
$1000n^2 - 1000n$
Also,
n^3
$n^{2.00001}$
$n^2 \lg\lg\lg n$
2^{2^n}

Θ-notation

$\Theta(g(n)) = \{f(n) :$ there exist positive constants c_1, c_2, and n_0 such that
$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.

Example: n2 /2 − 2n = Θ (n2), with c1 = 1/4, c2 = 1/2, and n0 = 8.

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

o-notation

$o(g(n)) = \{f(n) :$ for all constants $c > 0$, there exists a constant
$n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$.

Another view, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

$n^{1.9999} = o(n^2)$
$n^2/\lg n = o(n^2)$
$n^2 \neq o(n^2)$ (just like $2 \not< 2$)
$n^2/1000 \neq o(n^2)$

ω-notation

$\omega(g(n)) = \{f(n) :$ for all constants $c > 0$, there exists a constant
$n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$.

Another view, again, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

$n^{2.0001} = \omega(n^2)$
$n^2 \lg n = \omega(n^2)$
$n^2 \neq \omega(n^2)$

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 3-5: Recurrences, Solution of Recurrences by substitution, Recursion Tree and Master Method

Recursion is a particularly powerful kind of reduction, which can be described loosely as follows:

• If the given instance of the problem is small or simple enough, just solve it.

• Otherwise, reduce the problem to one or more simpler instances of the same problem.

Recursion is generally expressed in terms of recurrences. In other words, when an algorithm calls to itself, we can often describe its running time by a recurrence equation which describes the overall running time of a problem of size n in terms of the running time on smaller inputs.

E.g. the worst case running time T(n) of the merge sort procedure by recurrence can be expressed as

$T(n) = \Theta(1)$;if n=1

$2T(n/2) + \Theta(n)$;if n>1 whose solution can be found as $T(n)=\Theta(n \log n)$

There are various techniques to solve recurrences.

1. SUBSTITUTION METHOD:

The substitution method comprises of 3 steps

Guess the form of the solution

Verify by induction

Solve for constants

We substitute the guessed solution for the function when applying the inductive hypothesis to smaller values. Hence the name “substitution method”. This method is powerful, but we must be able to guess the form of the answer in order to apply it.

e.g. recurrence equation: $T(n)=4T(n/2)+n$

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

step 1: guess the form of solution T(n)=4T(n/2)

- F(n)=4f(n/2)
- F(2n)=4f(n)
- F(n)=n2

So, T(n) is order of n2 Guess T(n)=O(n3)

Step 2: verify the induction

Assume T(k)<=ck3 T(n)=4T(n/2)+n

<=4c(n/2)3 +n

<=cn3/2+n

<=cn3-(cn3/2-n)

T(n)<=cn3 as (cn3/2 –n) is always positive So what we assumed was true.

- T(n)=O(n3)

Step 3: solve for constants Cn3/2-n>=0

- n>=1
- c>=2

Now suppose we guess that T(n)=O(n2) which is tight upper bound Assume(k)<=ck2

so, we should prove that T(n)<=cn2

T(n)=4T(n/2)+n

= 4c(n/2)2+n

=cn2+n

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

So, T(n) will never be less than cn2. But if we will take the assumption of T(k)=c1 k2-c2k, then we can find that T(n) = O(n2)

2. BY ITERATIVE METHOD:

e.g. T(n)=2T(n/2)+n

=> 2[2T(n/4) + n/2]+n

=>22T(n/4)+n+n

=> 22[2T(n/8)+ n/4]+2n

=>23T(n/23) +3n

After k iterations ,T(n)=2kT(n/2k)+kn----------(1) Sub problem size is 1 after n/2k=1 => k=logn So, after logn iterations ,the sub-problem size will be 1. So, when k=long is put in equation 1 T(n)=nT(1)+nlogn

= nc+nlogn (say c=T(1))

= O(nlogn)

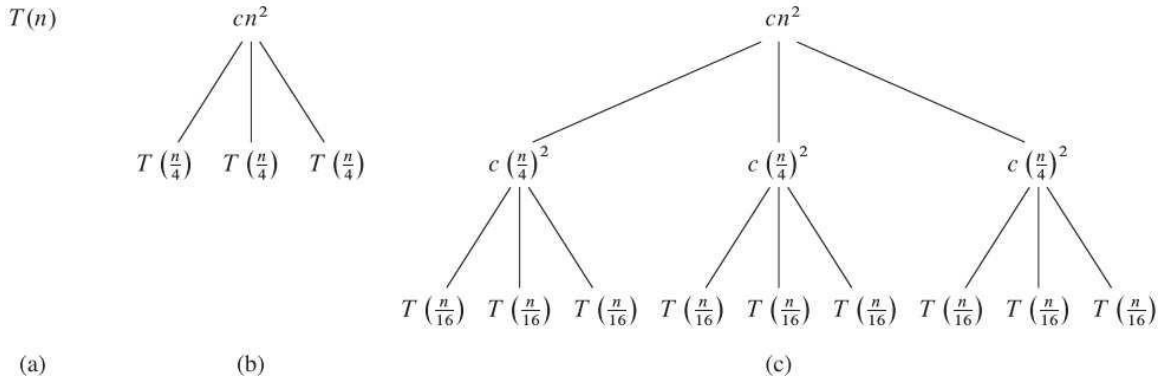
3.BY RECURSSION TREE METHOD:

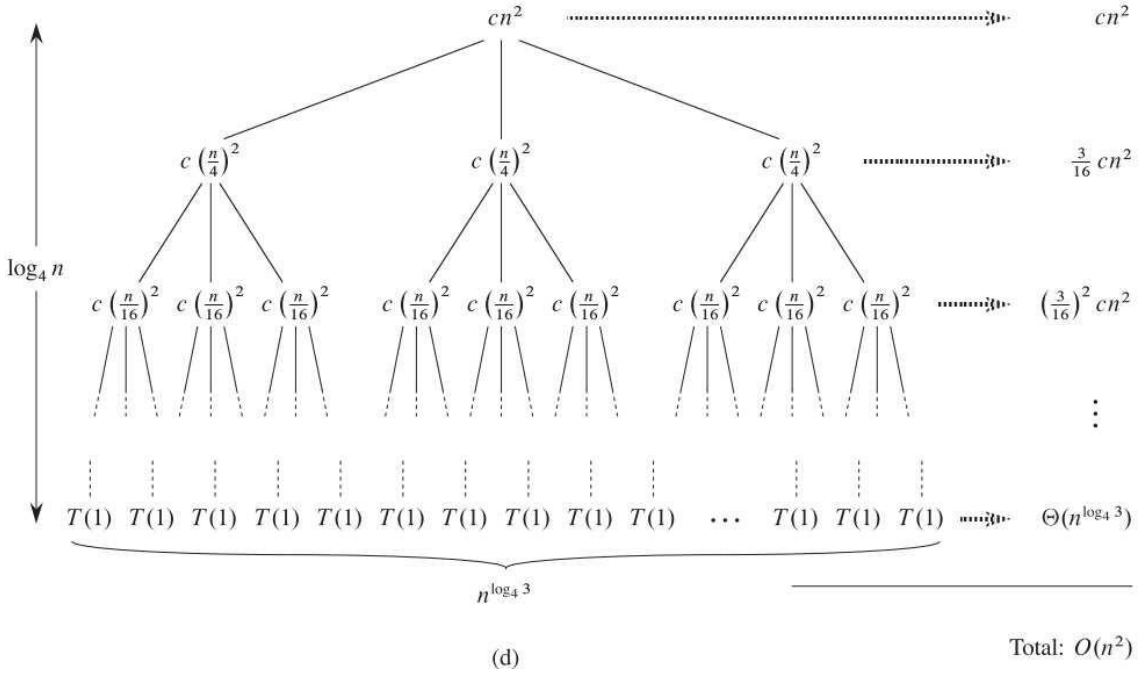
In a recursion tree ,each node represents the cost of a single sub-problem somewhere in the set of recursive problems invocations .we sum the cost within each level of the tree to obtain a set of per level cost, and then we sum all the per level cost to determine the total cost of all levels of recursion .

Constructing a recursion tree for the recurrence T(n)=3T(n/4)+cn2

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)





Constructing a recursion tree for the recurrence T (n)= 3T (n=4) + cn2.. Part (a) shows T (n), which progressively expands in (b)–(d) to form the recursion tree. The fully expanded tree in part

(d) has height log4n (it has log4n + 1 levels).

Sub problem size at depth i =n/4i

Sub problem size is 1 when n/4i=1 => i=log4n So, no. of levels =1+ log4n

Cost of each level = (no. of nodes) x (cost of each node)

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

No. Of nodes at depth i=3i

Cost of each node at depth i=c (n/4i)2

Cost of each level at depth i=3i c (n/4i)2 = (3/16)icn2 T(n)= i=0∑log4n cn2(3/16)i

T(n)= i=0∑log4n -1 cn2(3/16)i + cost of last level Cost of nodes in last level =3iT(1)

- c3log n (at last level i=log n)
- cnlog 3
  4

$$T(n)=\sum_{i=0}^{\log_4 n-1} cn^2(3/16)^i + c\,n^{\log 3}_{4}$$

<= cn2

$$\sum_{i=0}^{\infty} (3/16)^i + cn^{\log_4 3}$$

- <= cn2*(16/13)+ cnlog 3 =>T(n)=O(n2)
  4

4.BY MASTER METHOD:

The master method solves recurrences of the form

T(n)=aT(n/b)+f(n)

where a>=1 and b>1 are constants and f(n) is a asymptotically positive function . To use the master method, we have to remember 3 cases:

If f(n)=O(nlogba - Ɛ) for some constants Ɛ >0,then T(n)=Θ(nlogba)

If f(n)=Θ(nlogba) then T(n)=Θ(nlogbalogn)

If f(n)=Ω(nlogba+Ɛ) for some constant Ɛ>0 ,and if a*f(n/b)<=c*f(n) for some constant c<1 and all sufficiently large n,then T(n)=Θ(f(n))

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

T(n)=2T(n/2)+nlogn

ans: a=2 b=2

f(n)=nlogn

using 2nd formula f(n)=Θ(nlog 2logkn)

=>Θ(n1 logkn)=nlogn =>K=1

T(n)=Θ(nlog 2 log1n)

=>Θ(nlog2n)

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 6 - Worst case analysis of merge sort, quick sort

Merge sort

It is one of the well-known divide-and-conquer algorithm. This is a simple and very efficient algorithm for sorting a list of numbers.

We are given a sequence of n numbers which we will assume is stored in an array A [1...n]. The objective is to output a permutation of this sequence, sorted in increasing order. This is normally done by permuting the elements within the array A.

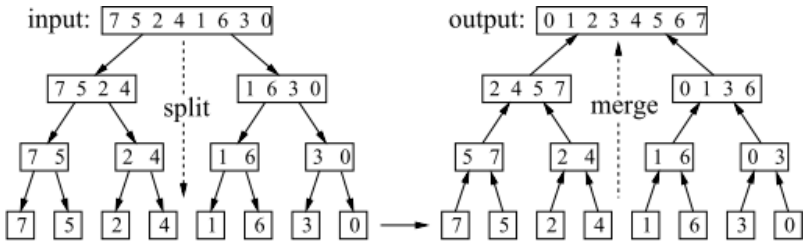
How can we apply divide-and-conquer to sorting? Here are the major elements of the Merge Sort algorithm.

Divide: Split A down the middle into two sub-sequences, each of size roughly n/2 . Conquer: Sort each subsequence (by calling Merge Sort recursively on each).

Combine: Merge the two sorted sub-sequences into a single sorted list.

The dividing process ends when we have split the sub-sequences down to a single item. A sequence of length one is trivially sorted. The key operation where all the work is done is in the combine stage, which merges together two sorted lists into a single sorted list. It turns out that the merging process is quite easy to implement.

The following figure gives a high-level view of the algorithm. The “divide” phase is shown on the left. It works top-down splitting up the list into smaller subsists. The “conquer and combine” phases are shown on the right. They work bottom-up, merging sorted lists together into larger sorted lists.



Merge Sort

Designing the Merge Sort algorithm top-down. We’ll assume that the procedure that merges two sorted list is available to us. We’ll implement it later. Because the algorithm is called recursively on sublists,in addition to passing in the array itself, we will pass in two indices, which indicate the first and last indices of the sub array that we are to sort. The call Merge Sort(A, p, r) will sort the sub-array [p..r] and return the sorted result in the same sub array.

Here is the overview. If r = p, then this means that there is only one element to sort, and we may return immediately. Otherwise (if p < r) there are at least two elements, and we will invoke the divide-and-conquer. We find the index q, midway between p and r, namely q = (p + r) / 2 (rounded down to the nearest integer). Then we split the array into sub arrays A [p..q] and A [q

+ 1 ..r] . Call Merge Sort recursively to sort each sub array. Finally, we invoke a procedure (which we have yet to write) which merges these two sub arrays into a single sorted array.

Merge Sort(array A, into p, int r) {

if (p < r) { // we have at least 2 items q = (p + r)/2

This document is available for free with registration on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

```
Merge Sort(A, p, q)  // sort A[p..q]

Merge Sort(A, q+1, r)          // sort A[q+1..r]

Merge(A, p, q, r)      // merge everything together

}     }
```

Merging: All that is left is to describe the procedure that merges two sorted lists. Merge(A, p, q, r)assumes that the left sub array, A [p..q] , and the right sub array, A [q + 1 ..r] , have already been sorted. We merge these two sub arrays by copying the elements to a temporary working array called B. For convenience, we will assume that the array B has the same index range A, that is, B [p..r] . We have to indices i and j, that point to the current elements of each sub array. We move the smaller element into the next position of B (indicated by index k) and then increment the corresponding index (either i or j). When we run out of elements in one array, then we just copy the rest of the other array into B. Finally, we copy the entire contents of B back into A.

```
Merge(array A, int p, int q, int r) {     // merges A[p..q] with A[q+1..r] array B[p..r]

i = k = p       //initialize          pointers

j = q+1

while (i <= q and j <= r) {       // while both sub arrays are nonempty

if (A[i] <= A[j]) B[k++] = A[i++]           // copy from left sub array

else B[k++] = A[j++]  // copy from right sub array

}

while (i <= q) B[k++] = A[i++]// copy any leftover to B while (j <= r) B[k++] = A[j++]

for i = p to r do A[i] = B[i]      // copy B back to A        }
```

Analysis: What remains is to analyze the running time of Merge Sort. First let us consider the running time of the procedure Merge(A, p, q, r). Let n = r − p + 1 denote the total length of both the left and right sub arrays. What is the running time of Merge as a function of n? The algorithm contains four loops (none nested in the other). It is easy to see that each loop can be executed at most n times. (If you are a bit more careful you can actually see that all the while-loops

together can only be executed times in total, because each execution copies one new element to the array B, and B only has space form elements.) Thus the running time to Merge n items is Θ (n) . Let us write this without the asymptotic notation, simply as n. (We'll see later why we do this.)

Now, how do we describe the running time of the entire Merge Sort algorithm? We will do this through the use of a recurrence, that is, a function that is defined recursively in terms of itself. To avoid circularity, the recurrence for a given value of n is defined in terms of values that are strictly smaller than n. Finally, a recurrence has some basis values (e.g. for n = 1), which are defined explicitly.

Let's see how to apply this to Merge Sort. Let T (n) denote the worst case running time of Merge Sort on an array of length n. For concreteness we could count whatever we like: number of lines of pseudocode,number of comparisons, number of array accesses, since these will only differ by

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

a constant factor. Since all of the real work is done in the Merge procedure, we will count the total time spent in the Merge procedure.

First observe that if we call Merge Sort with a list containing a single element, then the running time is constant. Since we are ignoring constant factors, we can just write T (n) =1 . When we call Merge Sort with a list of length n >1 , e.g. Merge(A, p, r), where r − p +1 = n, the algorithm first computes q = (p + r) / 2 . The sub array A [p..q] , which contains q − p + 1 elements. You can verify that is of size n/ 2 . Thus the remaining sub array A [q +1 ..r] has n/ 2 elements in it. How long does it take to sort the left sub array? We do not know this, but because n/ 2< n for n >1 , we can express this as T (n/ 2) . Similarly, we can express the time that it takes to sort the right sub array as T (n/ 2).

Finally, to merge both sorted lists takes n time, by the comments made above. In conclusion we have

T (n) =1 if n = 1 ,

2T (n/ 2) + n otherwise.

Solving the above recurrence we can see that merge sort has a time complexity of Θ (n log n) .

QUICKSORT

Worst-case running time: O (n2).

Expected running time: O (n lgn).

Sorts in place.

Description of quick sort

Quick sort is based on the three-step process of divide-and-conquer.

To sort the subarea[p . . r]:

Divide: Partition A[p . . r], into two (possibly empty) subarraysA[p . . q − 1] and

A[q + 1 . . r], such that each element in the ÞrstsubarrayA[p . . q − 1] is ≤ A[q] and

A[q] is ≤ each element in the second subarrayA[q + 1 . . r].

Conquer: Sort the two sub arrays by recursive calls to QUICKSORT.

Combine: No work is needed to combine the sub arrays, because they are sorted in place.

Perform the divide step by a procedure PARTITION, which returns the index q that marks the position separating the sub arrays.

QUICKSORT (A, p, r)

ifp < r

thenq ←PARTITION(A, p, r)

QUICKSORT (A, p, q − 1) QUICKSORT (A, q + 1, r)

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Initial call is QUICKSORT (A, 1, n)

Partitioning

Partition subarrayA [p . . . r] by the following procedure: PARTITION (A, p, r)

x ← A[r]

i ← p –1

for j ← p to r –1

do if A[j] ≤ x, then i ← i + 1

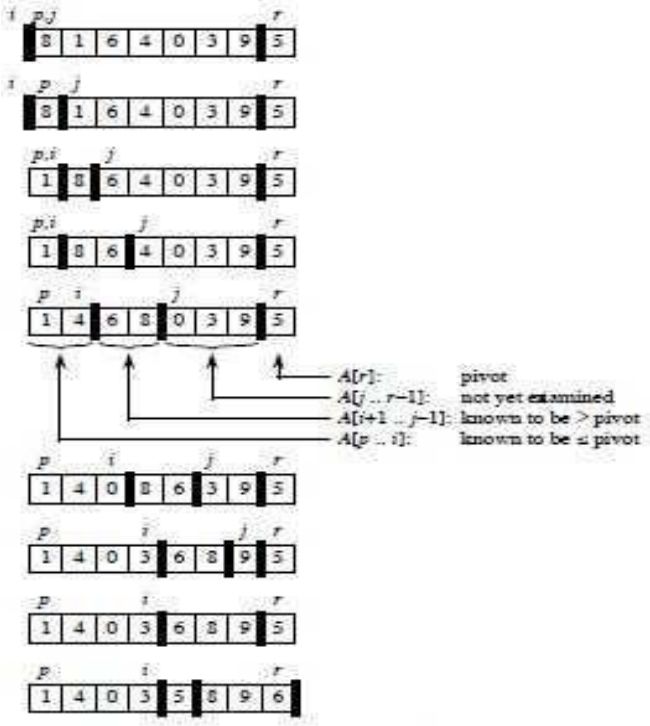
exchange A[i] ↔ A[j]

exchange A[i + 1] ↔ A[r]

return i + 1

PARTITION always selects the last element A[r] in the subarrayA[p . . r] as the pivot the element around which to partition.

As the procedure executes, the array is partitioned into four regions, some of which may be empty:



[The index j disappears because it is no longer needed once the for loop is exited.]

Performance of Quick sort

The running time of Quick sort depends on the partitioning of the sub arrays:

• If the sub arrays are balanced, then Quick sort can run as fast as merge sort.

• If they are unbalanced, then Quick sort can run as slowly as insertion sort.

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Worst case

• Occurs when the sub arrays are completely unbalanced.

• Have 0 elements in one sub array and $n - 1$ elements in the other sub array.

• Get the recurrence

$T(n) = T(n - 1) + T(0) + \Theta(n)$

$= T(n - 1) + \Theta(n)$

$= O(n2)$.

• Same running time as insertion sort.

• In fact, the worst-case running time occurs when Quick sort takes a sorted array as input, but insertion sort runs in $O(n)$ time in this case.

Best case

• Occurs when the sub arrays are completely balanced every time.

• Each sub array has $\leq n/2$ elements.

• Get the recurrence

$T(n) = 2T(n/2) + \Theta(n) = O(n \lg n)$.

Balanced partitioning

• QuickPort's average running time is much closer to the best case than to the worst case.

• Imagine that PARTITION always produces a 9-to-1 split.

• Get the recurrence

$T(n) \leq T(9n/10) + T(n/10) + _(n) = O(n \lg n)$.

• Intuition: look at the recursion tree.

• It's like the one for $T(n) = T(n/3) + T(2n/3) + O(n)$.

• Except that here the constants are different; we get log10 n full levels and log10/9 n levels that are nonempty.

• As long as it's a constant, the base of the log doesn't matter in asymptotic notation.

• Any split of constant proportionality will yield a recursion tree of depth $O(\lg n)$.

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 7 - Design and analysis of Divide and Conquer Algorithms

DIVIDE AND CONQUER ALGORITHM

- In this approach ,we solve a problem recursively by applying 3 steps
  1. DIVIDE-break the problem into several sub problems of smaller size.
  2. CONQUER-solve the problem recursively.
  3. COMBINE-combine these solutions to create a solution to the original problem.

CONTROL ABSTRACTION FOR DIVIDE AND CONQUER ALGORITHM

Algorithm D and C (P)
{
if small(P)
then return S(P)
else

-
- { divide P into smaller instances P1 ,P2 .....Pk Apply D and C to each sub problem
- Return combine (D and C(P1)+ D and C(P2)+.....+D and C(Pk)

}

}

Let a recurrence relation is expressed as T(n)=
Θ(1), if n<=C
aT(n/b) + D(n)+ C(n) ,otherwise
then n=input size a=no. Of sub-problems /b= input size of the sub-problems

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

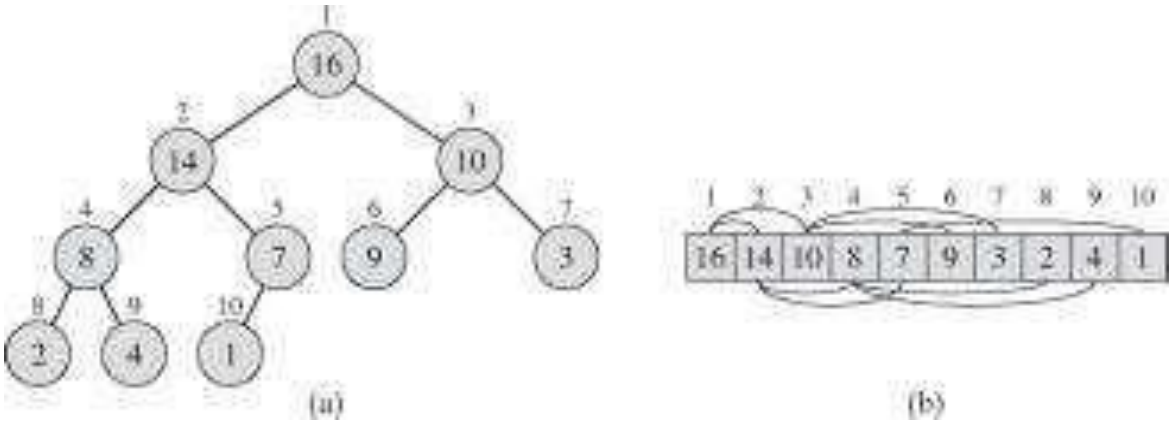
Lecture 8-Heaps and Heap sort

HEAPSORT

In place algorithm
Running Time: O(n log n)

Complete Binary Tree

The (binary) heap data structure is an array object that we can view as a nearly complete binary tree. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.
The root of the tree is A[1], and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:



PARENT (i) => return [i / 2]
LEFT (i) => return 2i
RIGHT (i) => return 2i+ 1

On most computers, the LEFT procedure can compute 2i in one instruction by simply shifting the binary representation of i left by one bit position.
Similarly, the RIGHT procedure can quickly compute 2i + 1 by shifting the binary representation of i left by one bit position and then adding in a 1 as the low-order bit.

The PARENT procedure can compute [i/2] by shifting i right one bit position. Good implementations of heap sort often implement these procedures as "macros" or "inline" procedures.
There are two kinds of binary heaps: max-heaps and min-heaps.

In a max-heap,the max-heap property is that for every node i other than the root, A[PARENT(i)] >= A[i] ,that is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself.
A min-heap is organized in the opposite way; the min-heap property is that for every node i other than the root, A[PARENT(i)<=A[i] ,
The smallest element in a min-heap is at the root.

The height of a node in a heap is the number of edges on the longest simple downward path from the node to a leaf and
The height of the heap is the height of its root.
Height of a heap of n elements which is based on a complete binary tree is O(log n).

studocu
This document is available free of charge on
Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Maintaining the heap property

MAX-HEAPIFY lets the value at A[i] "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.
MAX-HEAPIFY(A,i)

l □ LEFT(i)
r □RIGHT(i)
if A[l] > A[i]
largest □l
if A[r] > A[largest]
Largest □r
if largest != i
Then exchange A[i] □ □A[largest]

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 9-MAX-HEAPIFY(A, largest)

At each step, the largest of the elements A[i], A[LEFT(i)], and A[RIGHT(i)] is determined, and its index is stored in largest. If A[i] is largest, then the subtree rooted at node i is already a max-heap and the procedure terminates. Otherwise, one of the two children has the largest element, and A[i] is swapped with A[largest], which causes node i and its children to satisfy the max-heap property. The node indexed by largest, however, now has the original value A[i], and thus the subtree rooted at largest might violate the max-heap property. Consequently, we call MAX-HEAPIFY recursively on that subtree.

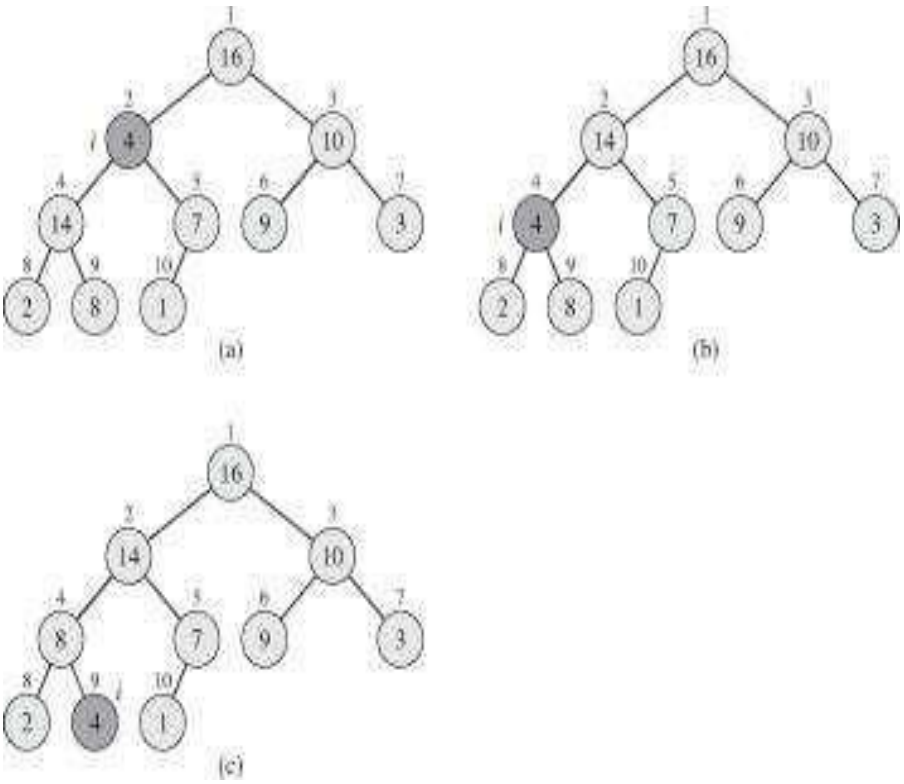


Figure: The action of MAX-HEAPIFY (A, 2), where heap-size = 10. (a) The initial con-figuration, with A [2] at node i = 2 violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in (b) by exchanging A [2] with A[4], which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY (A,4)

This document is available free of charge on studocu

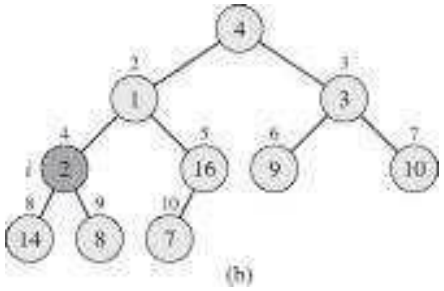
Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

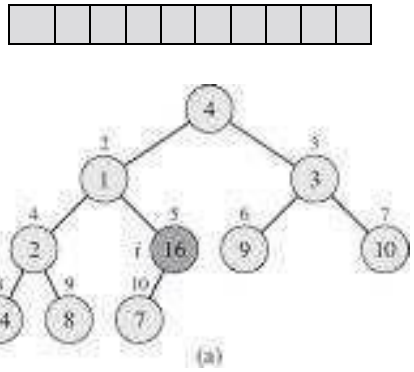
now has i = 4. After swapping A[4] with A[9], as shown in (c), node 4 is fixed up, and the recursive call MAX-HEAPIFY(A, 9) yields no further change to the data structure.
The running time of MAX-HEAPIFY by the recurrence can be described as T (n) < = T (2n/3) + O (1)
The solution to this recurrence is T(n)=O(log n)

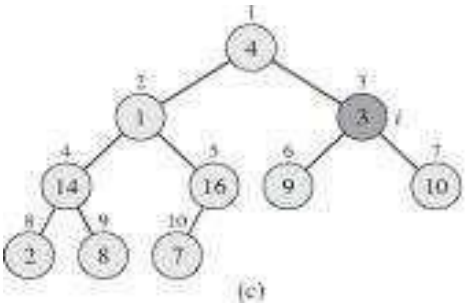
Building a heap

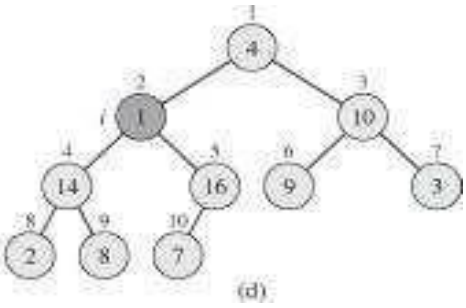
Build-Max-Heap(A)

for i□[n/2] to 1
do MAX-HEAPIFY(A,i)

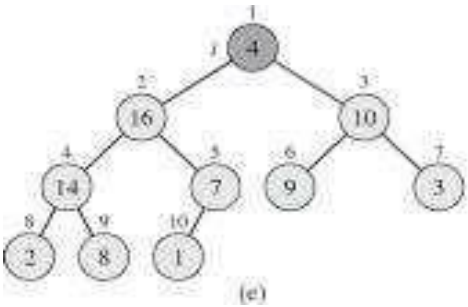


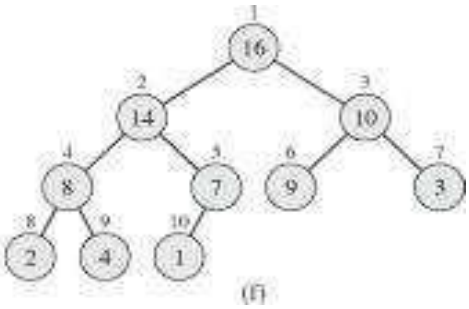






•





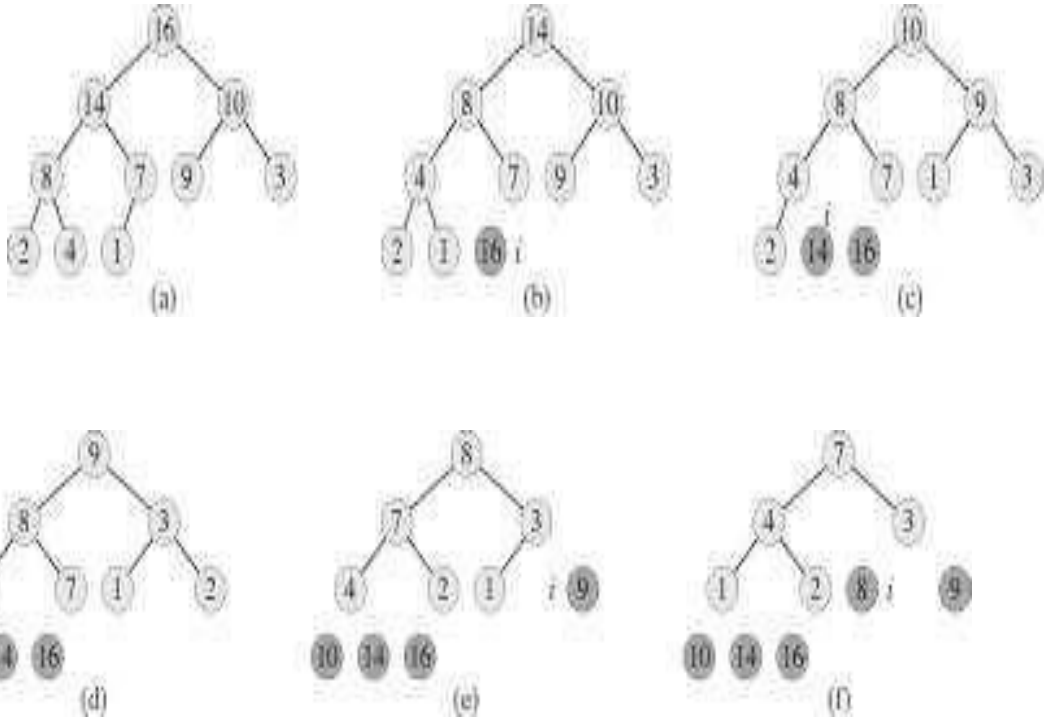
Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

- We can derive a tighter bound by observing that the time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small. Our tighter analysis relies on the properties that an n-element heap has height [log n] and at most [n/2h+1] nodes of any height h.
- The total cost of BUILD-MAX-HEAP as being bounded is T(n)=O(n)

The HEAPSORT Algorithm

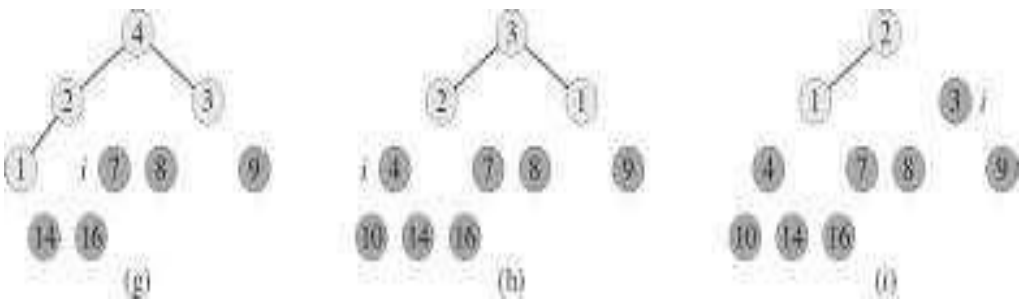
- HEAPSORT(A)

- BUILD MAX-HEAP(A)
- for i=n to 2
- exchange A[1] with A[i]
- MAX-HEAPIFY(A,1)



A

1	2	3	4	7	8	9	10	14	16



The HEAPSORT procedure takes time O(n log n), since the call to BUILD-MAX- HEAP takes time O(n) and each of the n - 1 calls to MAX-HEAPIFY takes time O(log n).

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 10-Lower Bounds For Sorting

Review of Sorting: So far we have seen a number of algorithms for sorting a list of numbers in ascending order. Recall that an in-place sorting algorithm is one that uses no additional array storage (however, we allow Quick sort to be called in-place even though they need a stack of size O(log n) for keeping track of the recursion). A sorting algorithm is stable if duplicate elements remain in the same relative position after sorting.

Slow Algorithms: Include Bubble Sort, Insertion Sort, and Selection Sort. These are all simple Θ (n2)in-place sorting algorithms. Bubble Sort and Insertion Sort can be implemented as stable algorithms, but Selection Sort cannot (without significant modifications).

Merge sort: Merge sort is a stable Θ(n log n) sorting algorithm. The downside is that Merge Sort is the only algorithm of the three that requires additional array storage, implying that it is not an on-place algorithm

Quick sort:
Widely regarded as the fastest of the fast algorithms. This algorithm is O(n log n) in the expected case, and O(n2) in the worst case. The probability that the algorithm takes asymptotically longer (assuming that the pivot is chosen randomly) is extremely small for large n. It is an(almost) in-place sorting algorithm but is not stable.
Heap sort: Heap sort is based on a nice data structure, called a heap, which is a fast priority queue. Elements can be inserted into a heap in O(log n) time, and the largest item can be extracted in O (log n) time. (It is also easy to set up a heap for extracting the smallest item.) If you only wantto extract the k largest values, a heap can allow you to do this is O(n + k log n) time. It is anon-place algorithm, but it is not stable.
Lower Bounds for Comparison-Based Sorting: Can we sort faster than O(n log n) time?
We will give an argument that if the sorting algorithm is based solely on making comparisons between the keys in the array, then it is impossible to sort more efficiently than (n log n) time. Such an algorithm is called comparison-based sorting algorithm, and includes all of the algorithms given above. Virtually all known general purpose sorting algorithms are based on making comparisons, so this is not a very restrictive assumption. This does not preclude the possibility of a sorting algorithm whose actions are determined by other types of operations, for example, consulting the individual bits of numbers, performing arithmetic operations, indexing into an array based on arithmetic operations onkeys.We will show that any comparison-based sorting algorithm for a input sequence ha1; a2; : : : ; animist make at least (n log n) comparisons in the worst-case. This is still a difficult task if you think about it. It is easy to show that a problem can be solved fast (just give an algorithm). But to show that a problem cannot be solved fast you need to reason in some way about all the possible algorithms that might ever be written. In fact, it seems surprising that you could even hope to prove such a thing. The catch here is that we are limited to using comparison-based algorithms, and there is a clean mathematical way ofcharacterizing all such algorithms.
Decision Tree Argument: In order to prove lower bounds, we need an abstract way of modeling “any possible “comparison-based sorting algorithm, we model such algorithms in terms of an abstract model called a decision tree. In a comparison-based sorting algorithm only comparisons between the keys are used to determine the action of the algorithm. Let ha1; a2; : : : ; anibe the input sequence. Given two elements, aiandaj, their relative order can only be determined by the results of comparisons like ai<aj, ai<=aj,ai=aj, ai>=aj, and ai>aj.A decision tree is a mathematical representation of a sorting algorithm (for a fixed value of n). Each node of the decision tree represents a comparison made in the algorithm (e.g., a4 : a7), and the two branches represent the possible results, for example, the left sub tree consists

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

of the remaining comparisons made under the assumption that a4 _ a7 and the right sub tree for a4 > a7. (Alternatively, one might be labeled with a4 < a7 and the other with a4 _ a7.)Observe that once we know the value of n, then the "action" of the sorting algorithm is completely determined by the results of its comparisons. This action may involve moving elements around in the array, copying them to other locations in memory, performing various arithmetic operations on non- key data. But the bottom-line is that at the end of the algorithm, the keys will be permuted in the final array in some way. Each leaf in the decision tree is labeled with the final permutation that the algorithm generates after making all of its comparisons. To make this more concrete, let us assume that n = 3, and let's build a decision tree for Selection Sort. Recall that the algorithm consists of two phases. The first finds the smallest element of the entire list, and swaps it with the first element. The second finds the smaller of the remaining two items, and swaps it with the second element. Here is the decision tree (in outline form). The first comparison is betweena1 and a2. The possible results are:

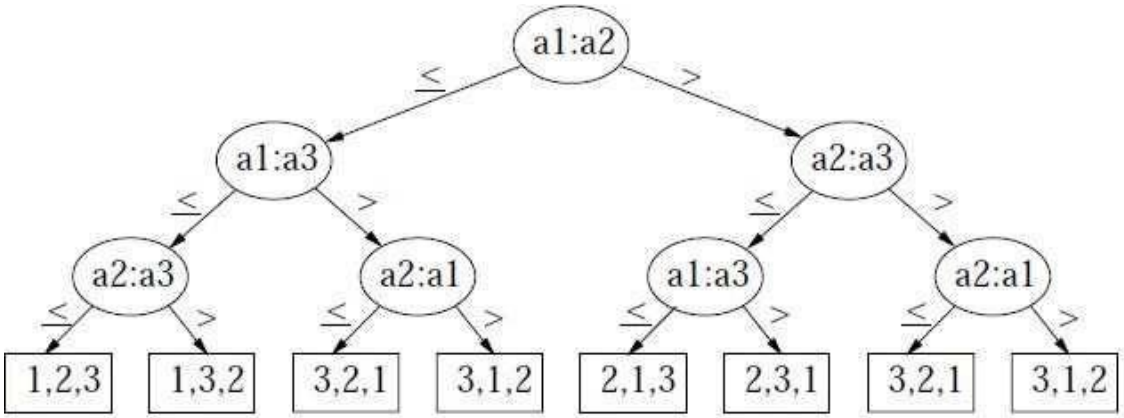
a1 <= a2: Then a1 is the current minimum. Next we compare a1 with a3 whose results might be either:
a1 <=a3: Then we know that a1 is the minimum overall, and the elements remain in their original positions. Then we pass to phase 2 and compare a2 with a3. The possible results are:
a2 <=a3: Final output is ha1; a2; a3i.
a2 > a3: These two are swapped and the final output is ha1; a3; a2i.
a1 > a3: Then we know that a3 is the minimum is the overall minimum, and it is swapped witha1. The we pass to phase 2 and compare a2 with a1 (which is now in the third position of the array) yielding either:
a2 <=a1: Final output is ha3; a2; a1i.
a2 > a1: These two are swapped and the final output is ha3; a1; a2i.
a1 > a2: Then a2 is the current minimum. Next we compare a2 with a3 whose results might be either:
a2 <=a3: Then we know that a2 is the minimum overall. We swap a2 with a1, and then pass to phase 2, and compare the remaining items a1 and a3. The possible results are:
a1 <=a3: Final output is ha2; a1; a3i.
a1 > a3: These two are swapped and the final output is ha2; a3; a1i.
a2 > a3: Then we know that a3 is the minimum is the overall minimum, and it is swapped witha1. We pass to phase 2 and compare a2 with a1 (which is now in the third position of the array) yielding either:
a2<= a1: Final output is ha3; a2; a1i.

a2 > a1: These two are swapped and the final output is ha3; a1; a2i.

The final decision tree is shown below. Note that there are some nodes that are unreachable. For example, in order to reach the fourth leaf from the left it must be that a1 _ a2 and a1 > a2, which cannot both be true. Can you explain this? (The answer is that virtually all sorting algorithms, especially inefficient ones like selection sort, may make comparisons that are redundant, in the sense that their outcome has already been determined by earlier comparisons.) As you can see, converting a complex sorting algorithm like Heap Sort into a decision tree for a large value of n will be very tedious and complex, but I hope you are convinced by this exercise that it can be done in a simple mechanical way

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)



(Decision Tree for Selection Sort on 3 keys.

Using Decision Trees for Analyzing Sorting: Consider any sorting algorithm. Let T(n) be the maximum number of comparisons that this algorithm makes on any input of size n. Notice that the running time of the algorithm must be at least as large as T(n), since we are not counting data movement or other computations at all. The algorithm defines a decision tree. Observe that the height of the decision tree is exactly equal to T(n), because any path from the root to a leaf corresponds to a sequence of comparisons made by the algorithm.

As we have seen earlier, any binary tree of height T (n) has at most 2T(n) leaves. This means that this sorting algorithm can distinguish between at most 2T (n) different final actions. Let's call this quantity (n), for the number of different final actions the algorithm can take. Each action can be thought of asa specific way of permuting the original input to get the sorted output. How many possible actions must any sorting algorithm distinguish between? If the input consists of indistinct numbers, then those numbers could be presented in any of n! Different permutations. For each different permutation, the algorithm must "unscramble" the numbers in an essentially different way, that is it must take a different action, implying that A(n) >= n!. (Again, A (n) is usually not exactly equal to n! because most algorithms contain some redundant unreachable leaves.)

- Since A(n) ≤ 2T(n) we have 2T(n) ≥ n!, implying that
- T(n) ≥ lg(n!):

- We can use Sterling's approximation for n! Yielding:

- $n! \geq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$
- $T(n) \geq \log\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right)$
- $= \log\sqrt{2\pi n} + n\log n - n\log e \qquad \Omega(n\log n)$ ∈

- Thus we have the following theorem.

- Theorem: Any comparison-based sorting algorithm has worst-case running time (n log n).
- This can be generalized to show that the average-case time to sort is also (n log n) (by arguing about the average height of a leaf in a tree with at least n! leaves). The lower bound on sorting can be generalized to provide lower bounds to a number of other problems as well.

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 11 - Dynamic Programming algorithms

Introduction

The Dynamic Programming (DP) is the most powerful design technique for solving optimization problems. It was invented by mathematician named Richard Bellman inn 1950s. The DP in closely related to divide and conquer techniques, where the problem is divided into smaller sub-problems and each sub-problem is solved recursively. The DP differs from divide and conquer in a way that instead of solving sub-problems recursively, it solves each of the sub-problems only once and stores the solution to the sub-problems in a table. The solution to the main problem is obtained by the solutions of these sub- problems.

The steps of Dynamic Programming technique are:

- **Dividing the problem into sub-problems**: The main problem is divided into smaller sub- problems. The solution of the main problem is expressed in terms of the solution for the smaller sub-problems.
- **Storing the sub solutions in a table**: The solution for each sub-problem is stored in a table so that it can be referred many times whenever required.
- **Bottom-up computation**:

  The DP technique starts with the smallest problem instance and develops the solution to sub instances of longer size and finally obtains the solution of the original problem instance.

The strategy can be used when the process of obtaining a solution of a problem can be viewed as a sequence of decisions. The problems of this type can be solved by taking an optimal sequence of decisions. An optimal sequence of decisions is found by taking one decision at a time and never making an erroneous decision. In Dynamic Programming, an optimal sequence of decisions is arrived at by using the principle of optimality. *The principle of optimality states that whatever be the initial state and decision, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision*.

A fundamental difference between the greedy strategy and dynamic programming is that in the greedy strategy only one decision sequence is generated, wherever in the dynamic programming, a number of them may be generated. Dynamic programming technique guarantees the optimal solution for a problem whereas greedy method never gives such guarantee.

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 12 - Matrix Chain Multiplication

Let, we have three matrices A_1, A_2 and A_3, with order (10 x 100), (100 x 5) and (5 x 50) respectively. Then the three matrices can be multiplied in two ways.

(i) First, multiplying A_2 and A_3, then multiplying A_1 with the resultant matrix i.e. $A_1(A_2 A_3)$.
(ii) First, multiplying A_1 and A_2, and then multiplying the resultant matrix with A_3 i.e. $(A_1A_2) A_3$.

The number of scalar multiplications required in case 1 is 100 * 5 * 50 + 10 * 100 * 50 = 25000 + 50,000 = 75,000 and the number of scalar multiplications required in case 2 is 10 * 100 * 5 + 10 * 5 * 50 = 5000 + 2500 = 7500

To find the best possible way to calculate the product, we could simply parenthesize the expression in every possible fashion and count each time how many scalar multiplications are required. Thus the matrix chain multiplication problem can be stated as "*find the optimal parenthesisation of a chain of matrices to be multiplied such that the number of scalar multiplications is minimized*".

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 13 - Elements of Dynamic Programming

Dynamic Programming Approach for Matrix Chain Multiplication

Let us consider a chain of n matrices $A_1, A_2 \ldots\ldots\ldots A_n$, where the matrix A_i has dimensions $P[i-1] \times P[i]$. Let the parenthesisation at k results two sub chains $A_1 \ldots\ldots A_k$ and $A_{k+1} \ldots\ldots\ldots A_n$. These two sub chains must each be optimal for $A_1 \ldots\ldots A_n$ to be optimal. The cost of matrix chain $(A_1 \ldots A_n)$ is calculated as $cost(A_1 \ldots\ldots A_k) + cost(A_{k+1} \ldots\ldots A_n)$ + cost of multiplying two resultant matrices together i.e.

$cost(A_1 \ldots\ldots A_n) = cost(A_1 \ldots\ldots A_k) + cost(A_{k+1} \ldots\ldots A_n)$ + cost of multiplying two resultant matrices together.

Here, the cost represents the number of scalar multiplications. The sub chain $(A_1 \ldots A_k)$ has a dimension $P[0] \times P[k]$ and the sub chain $(A_{k+1} \ldots\ldots A_n)$ has a dimension $P[k] \times P[n]$. The number of scalar multiplications required to multiply two resultant matrices is $P[0] \times P[k] \times P[n]$.

Let $m[i, j]$ be the minimum number of scalar multiplications required to multiply the matrix chain $(A_i \ldots\ldots\ldots A_j)$. Then

(i) $m[i, j] = 0$ if $i = j$

(ii) $m[i, j]$ = minimum number of scalar multiplications required to multiply $(A_i \ldots A_k)$ + minimum number of scalar multiplications required to multiply $(A_{k+1} \ldots A_n)$ + cost of multiplying two resultant matrices i.e.

$$m[i, j] = m[i, k] + m[k, j] + P[i-1] \times P[k] \times P[j]$$

However, we don't know the value of k, for which $m[i, j]$ is minimum. Therefore, we have to try all $j - i$ possibilities.

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j} \{ m[i,k] + m[k, j] + P[i-1] \times P[k] \times P[j] \} & \text{Otherwise} \end{cases}$$

Therefore, the minimum number of scalar multiplications required to multiply n matrices $A_1 A_2 \ldots\ldots A_n$ is

$$m[1, n] = \min_{1 \le k < n} \{ m[1, k] + m[k, n] + P[0] \times P[k] \times P[n] \}$$

The dynamic programming approach for matrix chain multiplication is presented in Algorithm

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Algorithm MATRIX-CHAIN-MULTIPLICATION (P)

// P is an array of length n+1 i.e. from $P[0]$ to $P[n]$. It is assumed that the matrix A_i has the dimension $P[i-1] \times P[i]$.

```
{
    for(i=1; i<=n; i++)
        m[i,i] = 0;

    for(l = 2; l<=n; l++){

        for(i = 1; i<=n-(l-1); i++){ j
            = i + (l-1);

            m[i, j] = ∞;

        for(k = i; k<=j-1; k++)

                q = m[i, k] + m[k+1, j] + P[i-1] P[k] P[j] ;

                if (q<m [i, j]){


                    m[i, j] = q;

                    s[i, j] = k;

                }
            }
        }
    }
    Return m and s.

}
```

Algorithm Matrix Chain multiplication algorithm.

Now let us discuss the procedure and pseudo code of the matrix chain multiplication. Suppose, we are given the number of matrices in the chain is n i.e. $A_1, A_2 \ldots\ldots A_n$ and the dimension of matrix A_i is $P[i-1] \times P[i]$. The input to the matrix-chain-order algorithm is a sequence $P[n+1] = \{P[0], P[1], \ldots\ldots P[n]\}$. The algorithm first computes $m[i, i] = 0$ for $i = 1, 2, \ldots\ldots n$ in lines 2-3. Then, the algorithm computes $m[i, j]$ for $j - i = 1$ in the first step to the calculation of $m[i, j]$ for $j - i = n - 1$ in the last step. In lines 3 – 11, the value of $m[i, j]$ is calculated for $j - i = 1$ to $j - i = n - 1$ recursively. At each step of the calculation of $m[i, j]$, a calculation on $m[i, k]$ and $m[k+1, j]$ for $i \leq k < j$, are required, which are already calculated in the previous steps.

To find the optimal placement of parenthesis for matrix chain multiplication $A_i, A_{i+1}, \ldots.A_j$, we should test the value of $i \leq k < j$ for which $m[i, j]$ is minimum. Then the matrix chain can be divided from $(A_1 \ldots\ldots A_k)$

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

and (A_{k+1} ……. A_j).

Let us consider matrices $A_1, A_2 \ldots\ldots A_5$ to illustrate MATRIX-CHAIN-MULTIPLICATION algorithm. The matrix chain order $P = \{P_0, P_1, P_2, P_3, P_4, P_5\} = \{5, 10, 3, 12, 5, 50\}$. The objective is to find the minimum number of scalar multiplications required to multiply the 5 matrices and also find the optimal sequence of multiplications.

The solution can be obtained by using a bottom up approach that means first we should calculate m_{ii} for $1 \le i \le 5$. Then m_{ij} is calculated for $j - i = 1$ to $j - i = 4$. We can fill the table shown in Fig. 7.4 to find the solution.

The value of m_{ii} for $1 \le i \le 5$ can be filled as 0 that means the elements in the first row can be assigned 0. Then

For j – i = 1

$m_{12} = P_0 P_1 P_2 = 5 \times 10 \times 3 = 150$

$m_{23} = P_1 P_2 P_3 = 10 \times 3 \times 12 = 360$

$m_{34} = P_2 P_3 P_4 = 3 \times 12 \times 5 = 180$

$m_{45} = P_3 P_4 P_5 = 12 \times 5 \times 50 = 3000$

For j – i = 2

$m_{13} = \min \{m_{11} + m_{23} + P_0 P_1 P_3, m_{12} + m_{33} + P_0 P_2 P_3\}$

$= \min \{0 + 360 + 5 * 10 * 12, 150 + 0 + 5*3*12\}$

$= \min \{360 + 600, 150 + 180\} = \min \{960, 330\} = 330$

$m_{24} = \min \{m_{22} + m_{34} + P_1 P_2 P_4, m_{23} + m_{44} + P_1 P_3 P_4\}$

$= \min \{0 + 180 + 10*3*5, 360 + 0 + 10*12*5\}$

$= \min \{180 + 150, 360 + 600\} = \min \{330, 960\} = 330$

$m_{35} = \min \{m_{33} + m_{45} + P_2 P_3 P_5, m_{34} + m_{55} + P_2 P_4 P_5\}$

$= \min \{0 + 3000 + 3*12*50, 180 + 0 + 3*5*50\}$

$= \min \{3000 + 1800 + 180 + 750\} = \min \{4800, 930\} = 930$

For j – i = 3

$m_{14} = \min \{m_{11} + m_{24} + P_0 P_1 P_4, m_{12} + m_{34} + P_0 P_2 P_4, m_{13} + m_{44} + P_0 P_3 P_4\}$

$= \min \{0 + 330 + 5*10*5, 150 + 180 + 5*3*5, 330 + 0 + 5*12*5\}$

$= \min \{330 + 250, 150 + 180 + 75, 330 + 300\}$

$= \min \{580, 405, 630\} = 405$

$m_{25} = \min \{m_{22} + m_{35} + P_1 P_2 P_5, m_{23} + m_{45} + P_1 P_3 P_5, m_{24} + m_{55} + P_1 P_4 P_5\}$

$= \min \{0 + 930 + 10*3*50, 360 + 3000 + 10*12*50, 330 + 0 + 10*5*50\}$

$= \min \{930 + 1500, 360 + 3000 + 6000, 330 + 2500\}$

$= \min \{2430, 9360, 2830\} = 2430$

For j - i = 4

$m_{15} = \min \{m_{11} + m_{25} + P_0 P_1 P_5, m_{12} + m_{35} + P_0 P_2 P_5, m_{13} + m_{45} + P_0 P_3 P_5, m_{14} + m_{55} + P_0 P_4 P_5\}$

$= \min \{0 + 2430 + 5*10*50, 150 + 930 + 5*3*50, 330 + 3000 + 5*12*50,$
$405 + 0 + 5*5*50\}$

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

= min {2430+2500, 150+930+750, 330+3000+3000, 405+1250}

= min {4930, 1830, 6330, 1655} = 1655

Hence, minimum number of scalar multiplications required to multiply the given five matrices in 1655.

To find the optimal parenthesization of A_1…….A_5, we find the value of *k* is 4 for which m_{15} is minimum. So the matrices can be splitted to (A_1….A_4) (A_5). Similarly, (A_1….A_4) can be splitted to (A_1A_2) (A_3 A_4) because for *k* = 2, m_{14} is minimum. No further splitting is required as the sub chains (A_1A_2) and (A_3 A_4) has length 1. So the optimal paranthesization of A_1 …….A_5 in ((A_1 A_2) (A_3 A_4)) (A_5).

Time complexity of multiplying a chain of *n* matrices

Let *T*(*n*) be the time complexity of multiplying a chain of *n* matrices.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ \Theta(1) + \sum_{k=1}^{n-1} [T(k) + T(n-k) + \Theta(1)] & \text{if } n > 1 \end{cases}$$

$$\Rightarrow T(n) = \Theta(1) + \sum_{k=1}^{n-1} [T(k) + T(n-k) + \Theta(1)] \quad \text{if } n > 1$$

$$= \Theta(1) + \Theta(n-1) + \sum_{k=1}^{n-1} [T(k) + T(n-k)]$$

$$\Rightarrow T(n) = \Theta(n) + 2[T(1) + T(2) + \text{LL} + T(n-1)] \text{LLL} (7.1)$$

Replacing *n* by *n*-1, we get

$$T(n-1) = \Theta(n-1) + 2[T(1) + T(2) + \text{LL} + T(n-2)] \text{LLL} (7.2)$$

Subtracting equation 7.2 from equation 7.1, we have

$$T(n) - T(n-1) = \Theta(n) - \Theta(n-1) + 2T(n-1)$$

$$\Rightarrow T(n) = \Theta(1) + 3T(n-1)$$

$$= \Theta(1) + 3[\Theta(1) + 3T(n-2)] = \Theta(1) + 3\Theta(1) + 3^2 T(n-2)$$

$$= \Theta(1)[1 + 3 + 3^2 + \text{LL} + 3^{n-2}] + 3^{n-1} T(1)$$

$$= \Theta(1)[1 + 3 + 3^2 + \text{LL} + 3^{n-1}]$$

$$= \frac{3^n - 1}{2} = O(2^n)$$

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Longest Common Subsequence

The longest common subsequence (LCS) problem can be formulated as follows "Given two sequences $X = \langle x_1, x_2 \ldots\ldots\ldots x_n \rangle$ and $Y = \langle y_1, y_2 \ldots\ldots\ldots y_n \rangle$ and the objective is to find the LCS $Z = \langle z_1, z_2 \ldots\ldots\ldots z_n \rangle$ that is common to x and y".

Given two sequences X and Y, we say Z is a common sub sequence of X and Y if Z is a subsequence of both X and Y. For example, $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$, the sequence $\langle B, C, A \rangle$ is a common subsequence. Similarly, there are many common subsequences in the two sequences X and Y. However, in the longest common subsequence problem, we wish to find a maximum length common subsequence of X and Y, that is $\langle B, C, B, A \rangle$ or $\langle B, D, A, B \rangle$. This section shows that the LCS problem can be solved efficiently using dynamic programming.

Theorem.4.1.(Optimal Structure of an LCS)

Let $X = \langle x_1, x_2 \ldots\ldots\ldots x_n \rangle$ and $Y = \langle y_1, y_2 \ldots\ldots\ldots y_n \rangle$ be sequences and let $Z = \langle z_1, z_2 \ldots\ldots\ldots z_n \rangle$ be any LCS of X and Y.

Case 1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an *LCS* of X_{m-1} and Y_{n-1}.
Case 2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an *LCS* of X_{m-1} and Y.
Case 3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Proof The proof of the theorem is presented below for all three cases.

Case 1. If $x_m = y_n$ and we assume that $z_k \neq x_m$ or $z_k \neq y_n$ then $x_m = y_n$ can be added to Z at any index after k violating the assumption that Z_k is the longest common subsequence. Hence $z_k = x_m = y_n$. If we do not consider Z_{k-1} as *LCS* of X_{m-1} and Y_{n-1}, then there may exist another subsequence W whose length is more than k-1. Hence, after adding $x_m = y_n$ to the subsequence W increases the size of subsequence more than k, which again violates our assumption.

Hence, $Z_k = x_m = y_n$ and Z_{k-1} must be an LCS of X_{m-1} and Y_{n-1}.

Case 2.If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y_n. If there were a common subsequence W of $X_{m-1}$and Y with length greater then k than W would also be an LCS of X_m and Y_n violating our assumption that Z_k is an LCS of X_m and Y_n.

Case 3. The proof is symmetric to case-2.

Thus the LCS problem has an optimal structure.

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Overlapping Sub-problems

From theorem 4.1, it is observed that either one or two cases are to be examined to find an LCS of $X_m$and Y_n. If $x_m = y_n$, then we must find an LCS of X_{m-1} and Y_{n-1}. If $x_m \neq y_n$, then we must find an LCS of X_{m-1} and Y_n and LCS of X_m and Y_{n-1}. The LCS of X and Y is the longer of these two LCSs.

Let us define *c*[*m, n*] to be the length of an LCS of the sequences X_m and Y_n. The optimal structure of the LCS problem gives the recursive formula

$$c[m, n] = \begin{cases} 0 & \text{if } m = 0 \text{ or } n = 0 \\ c[m-1, n-1]+1 & \text{if } x_m = y_n \\ \max\{c[m-1, n], c[m, n-1]\} & \text{if } x_m \neq y_n \end{cases} \quad \text{.....(7.1)}$$

Generalizing equation 7.1, we can formulate

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1]+1 & \text{if } x_i = y_j \\ \max\{c[i-1, j], c[i, j-1]\} & \text{if } x_i \neq y_j \end{cases} \quad \text{.....(7.2)}$$

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Algorithm LCS_LENGTH (*X*, *Y*)

```
{
    m=length [X]

    n=length [Y]

    for(i=1;i<=m;i++)
        c[i,0] = 0;

    for(j=0; j<n; j++)
        c[0, j]=0;

    for(i=1; i< m; i++){

        for(j = 1; j <= n; j++){

            if(x[i]==y[j]){

                c[i, j]=1+c[i-1, j-1];
                b[i,j] = ' ';


            else{

                if(c[ i-1, j] ≥ c[i, j-1] )

                    c[i, j] = c[i-1, j];

                    b[i,j] = '↑';

                else

                    c[i,j]=c[i, j-1];
                    b[i, j] = '←'

            }

    return c and b ;

}
```

Algorithm 7.3 Algorithm for finding Longest common subsequence .

Constructing an LCS

The algorithm LCS_LENGTH returns *c* and *b* tables. The *b* table can be used to construct the LCS of *X* and *Y*

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

quickly.

Algorithm PRINT_LCS (*b, X, i, j*)

```
{
    if (i==0 || j==0)
        return;
    if (b[i, j] == '↖'){
        PRINT_LCS (b, X, i-1, j-1)
        Print x_i
    }
    else if (b[i, j] == '↑')
        PRINT_LCS (b, X, i-1, j)
    else
        PRINT_LCS (b, X, i, j-1)
}
```

Algorithm 7.4 Algorithm to print the Longest common subsequence .

Let us consider two sequences $X = \langle C, R, O, S, S \rangle$ and $Y = \langle R, O, A, D, S \rangle$ and the objective is to find the LCS and its length. The c and b table are computed by using the algorithm LCS_LENGTH for X and Y that is shown in Fig.7.5. The longest common subsequence of X and Y is $\langle R, O, S \rangle$ and the length of LCS is 3.

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 15 - Greedy Algorithms

Greedy Method

Introduction

Let we are given a problem to sort the array $a = \{5, 3, 2, 9\}$. Someone says the array after sorting is $\{1, 3, 5, 7\}$. Can we consider the answer is correct? The answer is definitely "**no**" because the elements of the output set are not taken from the input set. Let someone says the array after sorting is $\{2, 5, 3, 9\}$. Can we admit the answer? The answer is again "**no**" because the output is not satisfying the objective function that is the first element must be less than the second, the second element must be less than the third and so on. Therefore, the solution is said to be a feasible solution if it satisfies the following constraints.

(i) **Explicit constraints**: - The elements of the output set must be taken from the input set.
(ii) **Implicit constraints**:-The objective function defined in the problem.

The best of all possible solutions is called the optimal solution. In other words we need to find the solution which has the optimal (maximum or minimum) value satisfying the given constraints.

The Greedy approach constructs the solution through a sequence of steps. Each step is chosen such that it is the best alternative among all feasible choices that are available. The choice of a step once made cannot be changed in subsequent steps.

Let us consider the problem of coin change. Suppose a greedy person has some 25p, 20p, 10p, 5paise coins. When someone asks him for some change then be wants to given the change with minimum number of coins. Now, let someone requests for a change of top then he first selects 25p. Then the remaining amount is 45p. Next, he selects the largest coin that is less than or equal to 45p i.e. 25p. The remaining 20p is paid by selecting a 20p coin. So the demand for top is paid by giving total 3 numbers of coins. This solution is an optimal solution. Now, let someone requests for a change of 40p then the Greedy approach first selects 25p coin, then a 10p coin and finally a 5p coin. However, the some could be paid with two 20p coins. So it is clear from this example that Greedy approach tries to find the optimal solution by selecting the elements one by one that are locally optimal. But Greedy method never gives the guarantee to find the optimal solution.

The choice of each step is a greedy approach is done based in the following:

- It must be feasible
- It must be locally optimal
- It must be unalterable

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 16 - Activity Selection Problem

Activity Selection Problem

Suppose we have a set of activities S = {a_1, a_2 .....a_n} that with to use a common resource. The objective is to schedule the activities in such a way that maximum number of activities can be performed. Each activity a_i has a start times, and a finish time fi, where $0 \leq S_i < f_i < \infty$. The activities a_i and a_j are said to compatible if the intervals [s_i, f_i] and [S_j, f_j] do not overlap that means $s_i \geq f_j$ *or* $s_j \geq f_i$.

For example, let us consider the following set S of activities, which are sorted in monotonically increasing order of finish time.

i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	72	13	14

For this example, the subsets {a_3, a_9, a_{11}}, {a_1, a_4, a_8, a_{11}} and {a_2, a_4, a_9, a_{11}} consist of mutually compatible activities. We have two largest subsets of mutually compatible activities.

Now, we can devise greedy algorithm that works in a top down fashion. We assume that the n input activities are ordered by monotonically increasing finish time or it can be sorted into this order in O($nlog_2n$) time. The greedy algorithm for activity selection problem is given below.

Algorithm ACTIVITY SELECTION (S, f)

```
{
    n = LENGTH (S) ; // n is the total number of activities //
    A = {a1} ; // A is the set of selected activities and initialized to a1)//
    i = 1; // i represents the recently selected activity //
    for (j = 2 ; j< = n ; j++)

    {
        if (sj ≥ fi) {
            A = A∪ {am} ;
            i = j ;

        }
```

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

```
    }
    Return A ;
}
```

Algorithm 3. Algorithm of activity selection problem.

The algorithm takes the start and finish times of the activities, represented as arrays s and f , length (s) gives the total number of activities. The set A stores the selected activities. Since the activities are ordered with respect to their finish times the set A is initialized to contain just the first activity a_1. The variable i stores the index of the recently selected activity. The for loop considers each activity and adds to the set A if it is mutually compatible with the previously selected activities. To see whether activity a_j is compatible with every activity assumingly in A, it needs to check whether the short time of a_j is greater or equal to the finish time of the recently selected activity a_i.

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 17 - Elements of Greedy Strategy

Greedy strategy I: In this case, the items are arranged by their profit values. Here the item with maximum profit is selected first. If the weight of the object is less than the remaining capacity of the knapsack then the object is selected full and the profit associated with the object is added to the total profit. Otherwise, a fraction of the object is selected so that the knapsack can be filled exactly. This process continues from selecting the highest profitable object to the lowest profitable object till the knapsack is exactly full.

Greedy strategy II: In this case, the items are arranged by fair weights. Here the item with minimum weight in selected first and the process continues like greedy strategy-I till the knapsack is exactly full.

Greedy strategy III: In this case, the items are arranged by profit/weight ratio and the item with maximum profit/weight ratio is selected first and the process continues like greedy strategy-I till the knapsack is exactly full.

Therefore, it is clear from the above strategies that the ***Greedy method*** generates optimal solution if we select the objects with respect to their profit to weight ratios that means the object with maximum profit to weight ratio will be selected first. Let there are *n* objects and the object *i* is associated with

profit p_i and weight w_i. Then we can say that if $p_1 / w_1 \geq p_2 / w_2 \geq \ldots \geq p_n / w_n$, the solution

$(x_1, x_2, x_3 \ldots x_n)$ generated by greedy method is an optimal solution. The proof of the above statement is left as an exercise for the readers. The algorithm 6.1 describes the greedy method for finding the optimal solution for fractional knapsack problem.

Algorithm FKNAPSACK (*p, w, x, n, M*)

// *p*[1:*n*] and *w*[1:*n*] contains the profit and weight of *n* objects. *M* is the maximum capacity of knapsack and *x*[1:*n*] in the solution vector.//

```
{
    for (i = 1; i<= n; i ++)
        x[i] =0 ;                    // initialize the solution to 0 //
        cu = M                       // cu is the remaining capacity of the knapsack//
    for (i =1; i<= n ; i ++){
        if(w[i] >cu )
            break;
        else{
            x[i] = 1 ;
```

This document is available free of charge on studocu
Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

```
            cu = cu – w[i] ;

        }
    }
    if( i<= n){

        x[i] = cu/w[i] ;

    return x;

}
```

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Fractional Knapsack Problem

Let there are *n* number of objects and each object is having a weight and contribution to profit. The knapsack of capacity *M* is given. The objective is to fill the knapsack in such a way that profit shall be maximum. We allow a fraction of item to be added to the knapsack.

Mathematically, we can write

$$\text{maximize} \sum_{i=1}^{n} p_i x_i$$

Subject to

$$\sum_{i=1}^{n} w_i x_i \leq M$$

$$1 \leq i \leq n \text{ and } 0 \leq x_i \leq 1.$$

Where p_i and w_i are the profit and weight of i^{th} object and x_i is the fraction of i^{th} object to be selected.

For example

Given $n = 3$, $(p_1, p_2, p_3) = \{25, 24, 15\}$

$(w_1, w_2, w_3) = \{18, 15, 10\}$ $M = 20$

Solution

Some of the feasible solutions are shown in the following table.

Solution No	x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
1	1	2/15	0	20	28.2
2	0	2/3	1	20	31.0
3	0	1	1/2	20	31.5

These solutions are obtained by different greedy strategies.

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 20 - Huffman Codes

Huffman Coding

Each character is represented in 8 bits when characters are coded using standard codes such as ASCII. It can be seen that the characters coded using standard codes have fixed-length code word representation. In this fixed-length coding system the total code length is more. For example, let we have six characters (a, b, c, d, e, f) and their frequency of occurrence in a message is {45, 13, 12, 16, 9, 5}. In fixed-length coding system we can use three characters to represent each code. Then the total code length of the message is (45+13+12+16+9+5) x 3 = 100 x 3 = 300.

Let us encode the characters with variable-length coding system. In this coding system, the character with higher frequency of occurrence is assigned fewer bits for representation while the characters having lower frequency of occurrence in assigned more bits for representation. The variable length code for the characters are shown in the following tableThe total code length in variable length coding system is $1 \times 45 + 3 \times 12 + 3 \times 16 \times 4 \times 9 + 4 \times 5 = 224$. Hence fixed length code requires 300 bits while variable code requires only 224 bits.

a	b	c	d	e	f
0	101	100	111	1101	1100

Prefix (Free) Codes

We have seen that using variable-length code word we minimize the overall encoded string length. But the question arises whether we can decode the string. If *a* is encoded 1 instead of 0 then the encoded string "111" can be decoded as "d" or "aaa". It can be seen that we get ambiguous string. The key point to remove this ambiguity is to use prefix codes. Prefix codes is the code in which there is no codeword that is a prefix of other codeword.

The representation of "decoding process" is binary tree whose leaves are characters. We interpret the binary codeword for a character as path from the root to that character, where

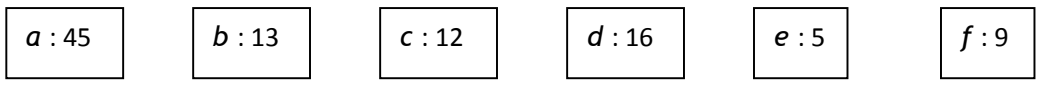
⇒ "0" means "go to the left child"
⇒ "1" means "go to the right child"

Greedy Algorithm for Huffman Code:

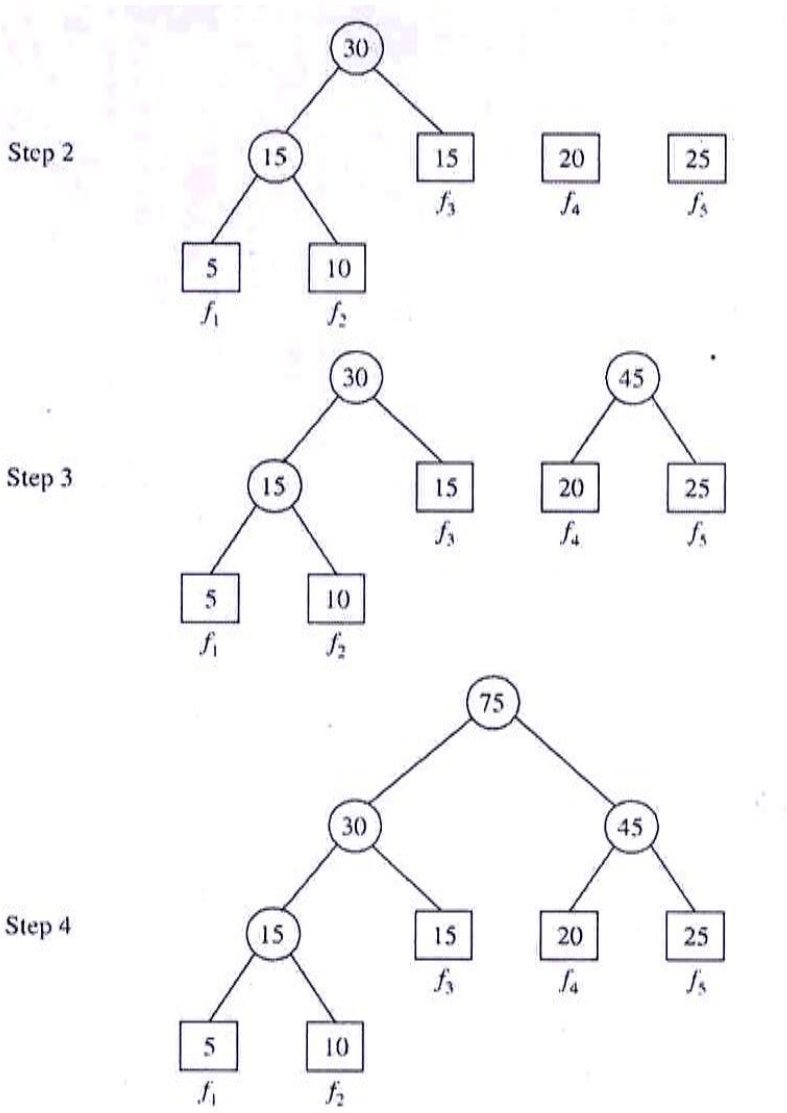
According to Huffman algorithm, a bottom up tree is built starting from the leaves. Initially, there are *n* singleton trees in the forest, as each tree is a leaf. The greedy strategy first finds two trees having minimum frequency of occurrences. Then these two trees are merged in a single tree where the frequency of this tree is the total sum of two merged trees. The whole process is repeated until there in only one tree in the forest.

Let us consider a set of characters $S=<a, b, c, d, e, f>$ with the following frequency of occurrences $P=$

< 45, 13, 12, 16, 5, 9 >. Initially, these six characters with their frequencies are considered six singleton trees in the forest. The step wise merging these trees to a single tree is shown in Fig. 6.3. The merging is done by selecting two trees with minimum frequencies till there is only one tree in the forest



Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)



Step wise merging of the singleton trees.

Now the left branch is assigned a code “0” and right branch is assigned a code “1”. The decode tree after assigning the codes to the branches.

The binary codeword for a character is interpreted as path from the root to that character; Hence, the codes for the characters are as follows

a = 0

b=101

c=100

d=111

e=1100

f=1101

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Therefore, it is seen that no code is the prefix of other code. Suppose we have a code 01111001101. To decode the binary codeword for a character, we traverse the tree. The first character is 0 and the character at which the tree traversal terminates is *a*. Then, the next bit is 1 for which the tree is traversed right. Since it has not reached at the leaf node, the tree is next traversed right for the next bit 1. Similarly, the tree is traversed for all the bits of the code string. When the tree traversal terminates at a leaf node, the tree traversal again starts from the root for the next bit of the code string. The character string after decoding is "*adcf*".

Algorithm HUFFMAN(*n, S*)

{

// *n* is the number of symbols and *S* in the set of characters, for each character $c \in S$, the frequency of Occurrence in $f(c)$ //

Initialize the priority queue;

Q = *S* ; // Initialize the priority Q with the frequencies of all the characters of set S//

for(*i* =1 ; *i*<= *n*-*i*, *i*++){

z=CREAT_NODE(); // create a node pointed by z; //

// Delete the character with minimum frequency from the Q and store in node ***x***//

x= DELETE _MIN (Q);

// Delete the character with next minimum frequency from the Q and store in node ***y***//

y = DELETE_MIN(Q);

$z \rightarrow left = x$; // Place x as the left child of *z*//

$z \rightarrow right = y$; // Place y as the right child of z//

//The value of node *z* is the sum of values at node x and node y//

$f(z) = f(x) + f(y)$;

//insert z into the priority Q//

INSERT(*Q*,*z*);

}

Return DELETE_MIN (*Q*)

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture – 21 Disjoint Set Data Structure

In computing, a **disjoint-set data structure**, also called a **union–find data structure** or **merge–find set**, is a data structure that keeps track of a set of elements partitioned into a number

of disjoint (non-overlapping) subsets.

It supports the following useful operations:

- *Find*: Determine which subset a particular element is in. *Find* typically returns an item from this set that serves as its "representative"; by comparing the result of two *Find* operations, one can determine whether two elements are in the same subset.
- *Union*: Join two subsets into a single subset.
- *Make Set*, which makes a set containing only a given element (a singleton), is generally trivial. With these three operations, many practical partitioning problems can be solved.

In order to define these operations more precisely, some way of representing the sets is needed. One common approach is to select a fixed element of each set, called its *representative*, to represent the set as a whole. Then, *Find*(x) returns the representative of the set that *x* belongs to, and *Union* takes two set representatives as its arguments.

Example :

Make Set creates 8 singletons.

After some operations of *Union*, some sets are grouped together.

Applications:

- partitioning of a set
- Boost Graph Library to implement its Incremental Connected Components functionality.
- Implementing Kruskal's algorithm to find the minimum spanning tree of a graph.
- Determine the connected components of an undirected graph.

```
CONNECTED-COMPONENTS(G)
1. for each vertex v∈V[G]
2. do MAKE-SET(v)
3. for each edge (u,v) ∈E[G]
4. do if FIND-SET(u) ≠ FIND-SET(v)
5. then UNION(u,v)
```

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

- 6. **for** each vertex $v \in V[G]$
  7. **do** MAKE-SET(v)
  8. **for** each edge $(u,v) \in E[G]$
  9. **do if** FIND-SET(u) $\neq$ FIND-SET(v)
  10. **then** UNION(u,v)

SAME-COMPONENT (u,v)

1. **if** FIND-SET(u)=FIND-SET(v)
2. **then return** TRUE

else

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 22 - Disjoint Set Operations, Linked list Representation

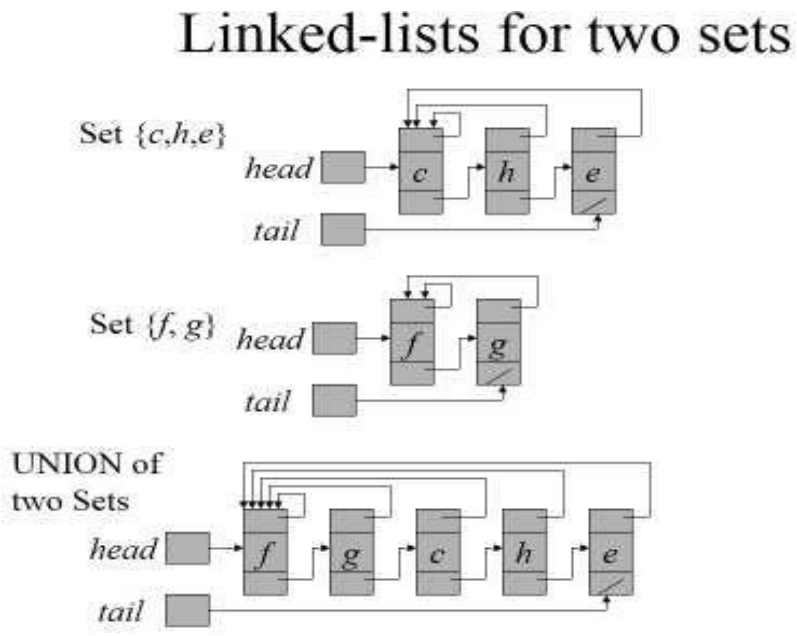
- A disjoint-set is a collection ©={S_1, S_2,…, S_k} of distinct dynamic sets.
- Each set is identified by a member of the set, called *representative*.

Disjoint set operations

- MAKE-SET(x): create a new set with only x. assume x is not already in some other set.
- UNION(x, y): combine the two sets containing x and y into one new set. A new representative is selected.
- FIND-SET(x): return the representative of the set containing x.

Linked list Representation

- Each set as a linked-list, with head and tail, and each node contains value, next node pointer and back-to-representative pointer.
- Example:
- MAKE-SET costs $O(1)$: just create a single element list.
- FIND-SET costs $O(1)$: just return back-to-representative pointer.



This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

UNION Implementation

- A simple implementation: UNION(x,y) just appends x to the end of y, updates all back-to-representative pointers in x to the head of y.
- Each UNION takes time linear in the x's length.
- Suppose n MAKE-SET(x_i) operations ($O(1)$ each) followed by n-1 UNION
  - UNION(x_1, x_2), $O(1)$,
  - UNION(x_2, x_3), $O(2)$,
  - …..
  - UNION(x_{n-1}, x_n), $O(n-1)$
- The UNIONs cost $1+2+\ldots+n-1=\Theta(n^2)$

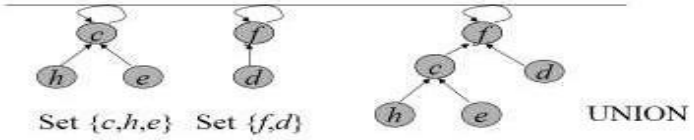
So $2n-1$ operations cost $\Theta(n^2)$, average $\Theta(n)$ each

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 23 - Disjoint Forests

Disjoint-set Implementation: Forests

- Rooted trees, each tree is a set, root is the representative. Each node points to its parent. Root points to itself.

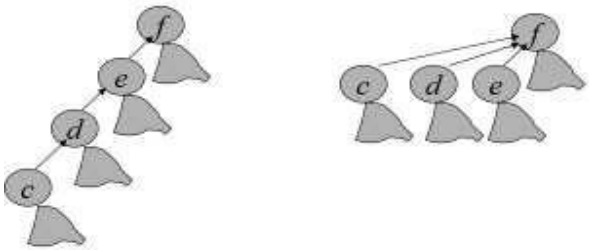


- Three operations
  - MAKE-SET(x): create a tree containing x. $O(1)$
  - FIND-SET(x): follow the chain of parent pointers until to the root. O(height of x's tree)
  - UNION(x, y): let the root of one tree point to the root of the other. $O(1)$
- It is possible that n-1 UNIONs results in a tree of height n-1. (just a linear chain of n nodes).
- So n FIND-SET operations will cost $O(n^2)$.

Union by Rank & Path Compression

- **Union by Rank:** Each node is associated with a rank, which is the upper bound on the height of the node (i.e., the height of sub tree rooted at the node), then when UNION, let the root with smaller rank point to the root with larger rank.
- **Path Compression:** used in FIND-SET(x) operation, make each node in the path from x to the root directly point to the root. Thus reduce the tree height.

Path Compression



This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Algorithm for Disjoint-Set Forest

```
MAKE-SET(x)
1.  p[x]←x
2.  rank[x]←0
```

```
UNION(x,y)
1. LINK(FIND-SET(x),FIND-SET(y))
```

```
LINK(x,y)
1.  if rank[x]>rank[y]
2.  then p[y] ←x
3.  else  p[x] ←y
4.        if rank[x]=rank[y]
5.        then rank[y]++
```

```
FIND-SET(x)
1.  if x≠ p[x]
2.     then p[x] ←FIND-SET(p[x])
3.  return p[x]
```

Worst case running time for m MAKE-SET, UNION, FIND-SET operations is: $O(m\alpha(n))$ where $\alpha(n)\leq 4$. So nearly linear in m.

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Module-III

Lecture 24 - Graph Algorithm - BFS and DFS

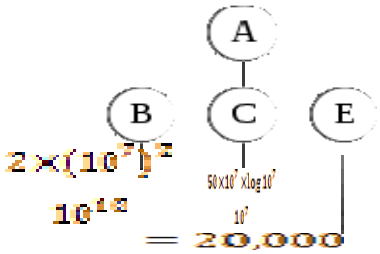
In graph theory, **breadth-first search** (**BFS**) is a strategy for searching in a graph when search is limited to essentially two operations:

(a) visit and inspect a node of a graph;
(b) gain access to visit the nodes that neighbor the currently visited node.

- The BFS begins at a root node and inspects all the neighboring nodes.
- Then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on.
- Compare BFS with the equivalent, but more memory-efficient.

Historical Background

- BFS was invented in the late 1950s by E. F. Moore, who used to find the shortest path out of a maze,
- discovered independently by C. Y. Lee as a wire routing algorithm (published 1961).



Example

A BFS search will visit the nodes in the following order: A, B, C, E, D, F, G

BFS Algorithm

The algorithm uses a queue data structure to store intermediate results as it traverses the graph, as follows:

1. Enqueue the root node
2. Dequeue a node and examine it
   - If the element sought is found in this node, quit the search and return a result.
   - Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.
3. If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
4. If the queue is not empty, repeat from Step 2.

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Applications

Breadth-first search can be used to solve many problems in graph theory, for example:

- Copying Collection, Cheney's algorithm
- Finding the shortest path between two nodes *u* and *v* (with path length measured by number of edges)
- Testing a graph for bipartiteness
- (Reverse) Cuthill–McKee mesh numbering
- Ford–Fulkerson method for computing the maximum flow in a flow network
- Serialization/Deserialization of a binary tree vs serialization in sorted order, allows the tree to be re-constructed in an efficient manner.

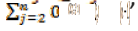
Pseudo Code

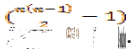
```
1  procedure BFS(G,v) is
2      create a queue Q
3      create a set V
4      add v to V
5      enqueuev onto Q
6      whileQ is not empty loop
7          t ← Q.dequeue()
8          ift is what we are looking for then
9              returnt
10         end if
11         for all edges e in G.adjacentEdges(t) loop
12             u ← G.adjacentVertex(t,e)
13             ifu is not in Vthen
14                 add u to V
15                 enqueueu onto Q
16             end if
17         end loop
18     end loop
19     return none
20 end BFS
```

Input: A graph *G* and a root *v* of G

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Time and space complexity

The time complexity can be expressed as since every vertex and every edge will be explored in the worst case. Note: may vary between and  , depending on how sparse the input graph is.

When the number of vertices in the graph is known ahead of time, and additional data structures are used to determine which vertices have already been added to the queue, the space complexity can be expressed as where is the cardinality of the set of vertices. If the graphis represented by an Adjacency list it occupies space in memory, while an Adjacency matrix representation occupies  .

Depth-first search (**DFS**) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

Historical Background

A version of depth-first search was investigated in the 19th century by French mathematician

Charles Pierre Trémaux

Example

A DFS search will visit the nodes in the following order: A, B, D, F, E, C, G

Pseudo Code

Input: A graph *G* and a vertex *v* of G

Output: All vertices reachable from *v* labeled as discovered

A recursive implementation of DFS

```
1  procedure DFS(G,v):
2      label v as discovered
3      for all edges from v to winG.adjacentEdges(v) do
4          if vertex w is not labeled as discovered then
5              recursively call DFS(G,w)
```

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

```
1  procedure DFS-iterative(G,v):
2      let S be a stack
3      S.push(v)
4      whileS is not empty
5            v ← S.pop()
6            ifv is not labeled as discovered:
7                label v as discovered
8                for all edges from v to winG.adjacentEdges(v) do
9                    S.push(w)
```

A non-recursive implementation of DFS

Applications

- Finding connected components.
- Topological sorting.
- Finding the bridges of a graph.
- Generating words in order to plot the Limit Set of a Group.
- Finding strongly connected components.
- Planarity testing
- Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)
- Maze generation may use a randomized depth-firstsearch.
- Finding bi-connectivity in graphs.

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Minimum Cost Spanning Tree

Let $G = (V, E)$ be the graph where V is the set of vertices, E is the set of edges and $|V| = n$. The spanning tree $G' = (V, E')$ is a sub graph of G in which all the vertices of graph G are connected with minimum number of edges. The minimum number of edges required to correct all the vertices of a graph G in $n - 1$. Spanning tree plays a very important role in designing efficient algorithms.

Let us consider a graph shown in Fig 6.6(a). There are a number of possible spanning trees that is shown in Fig 6.6(b).

If we consider a weighted graph then all the spanning trees generated from the graph have different weights. The weight of the spanning tree is the sum of its edges weights. The spanning tree with minimum weight is called minimum spanning tree (MST). Fig. 6.7 shows a weighted graph and the minimum spanning tree.

A greedy method to obtain the minimum spanning tree would construct the tree edge by edge, where each edge is chosen accounting to some optimization criterion. An obvious criterion would be to choose an edge which adds a minimum weight to the total weight of the edges selected so far. There are two ways in which this criterion can be achieved.

1. The set of edges selected so far always forms a tree, the next edge to be added is such that not only it adds a minimum weight, but also forms a tree with the previous edges; it can be shown that the algorithm results in a minimum cost tree; this algorithm is called Prim's algorithm.
2. The edges are considered in non decreasing order of weight; the set T of edges at each stage is such that it is possible to complete T into a tree; thus T may not be a tree at all stages of the algorithm; this also results in a minimum cost tree; this algorithm is called Krusleat's algorithm.

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 26 - Kruskal algorithm

This algorithm starts with a list of edges sorted in non decreasing order of weights. It repeatedly adds the smallest edge to the spanning tree that does not create a cycle. Initially, each vertex is in its own tree in the forest. Then, the algorithm considers each edge ordered by increasing weights. If the edge (u, v) connects two different trees, then (u, v) is added to the set of edges of the MST and two trees connected by an edge (u, v) are merged in to a single tree. If an edge (u, v) connects two vertices in the same tree, then edge (u, v) is discarded.

The Krushal's algorithm for finding the MST is presented as follows. It starts with an empty set A, and selects at every stage the shortest edge that has not been chosen or rejected regardless of where this edge is situated in the graph. The pseudo code of Kruskal's algorithm is given in Algorithm .

The operations an disjoint sets used for Krushal's algorithm is as follows:

Make_set(v) : create a new set whose only member is pointed to v.
Find_set(v) : returns a pointer to the set containing v.
Union (u, v) : unites the dynamic sets that contain u and v into a new set that is union of these two sets.

Algorithm KRUSKAL (V, E, W)

// V is the set of vertices, E is the set of edges and W is the adjacency matrix to store the weights of the links. //

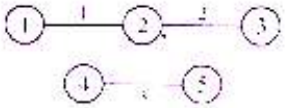
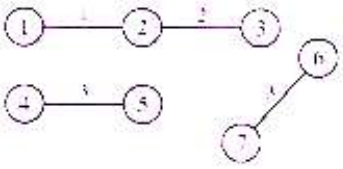
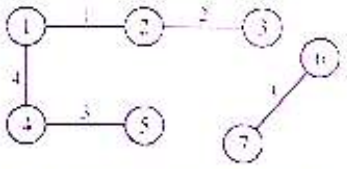
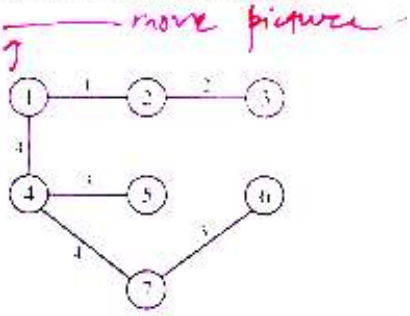
```
{
    A = Φ ;

    for (each vertex u in V)
        Make set(u)

    Create a min heap from the weights of the links using procedure heapify.

    for (each least weight edge (u, v) in E) //least weight edge is the root of the heap//
        if (Find set(u) ≠ Find_set(v)){ // u and v are in two different sets //
            A = A ∪ {u, v}

            Union (u, v)


        }

    }

    return A ;

}
```

Algorithm. 5 Kruskal's algorithm for finding MST.

Let us consider the graph shown in Fig.6.10 to illustrate Kruskal's algorithm.

The step wise procedure to construct MST by following the procedure presented given below.

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Edge selected	*Disjoint sets*	*Spanning tree after the edge included*
—	{1} {2} {3} {4} {5} {6} {7}	Φ
(1, 2)	{1, 2} {3} {4} {5} {6} {7}	
(2, 3)	{1, 2, 3} {4} {5} {6} {7}	
(4, 5)	{1, 2, 3} {4, 5} {6} {7}	
(6, 7)	{1, 2, 3} {4, 5} {6, 7}	
(1, 4)	{1, 2, 3, 4, 5} {6, 7}	
(2, 5)	Since 2 and 5 belong to the same set, the edge (2, 5) is discarded	
(4, 7)	{1, 2, 3, 4, 5, 6, 7}	
(3, 5) (2, 4) (3, 6) (5, 6)	Discarded	

Step wise construction of MST by Kruskal's algorithm

Time complexity of Kruskal's Algorithm

The Kruskal's algorithm first creates *n* trees from *n* vertices which is done in O(*n*) time. Then, a heap is created in O(*n*) time using heapify procedure. The least weight edge is at the root of the heap. Hence, the edges are deleted one by one from the heap and either added to the MST or discarded if it forms a cycle. This deletion process requires O($nlog_2n$). Hence, the time complexity of Kruskal's algorithm is O($nlog_2n$).

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 27 - Prim's Algorithm

This algorithm starts with a tree that has only one edge, the minimum weight edge. The edges (j, q) is added one by one such that node j is already included, node q is not included and weight $wt(j, q)$ is the minimum amongst all the edges (x, y) for which x is in the tree and y is not. In order to execute this algorithm efficiently, we have a node index $near(j)$ associated with each node j that is not yet included in the tree. If a node is included in the tree, $near(j) = 0$. The node $near(j)$ is selected into the tree such that $wt(j, near(j))$ in the minimum amongst all possible choices for $near(j)$.

Algorithm PRIM (E, wt, n, T)

//E is the set of edges, $wt(n, n)$ is the weight adjacency matrix for G, n is the number of nodes and $T(n–1, 3)$ stores the spanning tree.

```
{
    (k, l) = edge with minimum wt.
    minwt = wt[k, l] ;

    T[1, 1] = k, T[1, 2] = l ;
    for(i = 1; i<=n; i++){

        if(wt[i, k] <wt [i, l] )

            near[i] = k;

        else

            near[i] = l;

    }
    near[k] = near[l] = 0;

    for(i = 2; i≤=n-1 ; i++)

    {
        let j be an index such that near[j] ≠ 0 and wt[j, near[j]] is minimum.

        T[i, 1] = j; T[i, 2] = near[j];
        minwt = minwt + wt[j, near[j]] ;
        near[j] = 0;

        for(k = 1; k<= n ; k++){

            if (near[k]≠0 and wt[k, near[k]] >wt[k, j])
                near[k] = j;

        }
```

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

```
    }
    if(minwt == ∞)
        print("No spanning tree");
return minwt;
}
```

Algorithm 4. Prim's algorithm for finding MST.

Fig 6. The weighted undirected graph to illustrate Prim's algorithm

Let us consider the weighted undirected graph shown in Fig.6.8 and the objective is to construct a minimum spanning tree. The step wise operation of Prim's algorithm is described as follows.

Step 1 The minimum weight edge is (2, 3) with weight 5. Hence, the edge (2, 3) is added to the tree. *near*(2) and *near*(3) are set 0.

Step 2 Find near of all the nodes that are not yet selected into the tree and its cost.

near(1) = 2 *weight* = 16

near(4) = 2 *weight* = 6
near(5) = - *weight* = ∞
near(6) = 2 *weight* = 11

The node 4 is selected and the edge (2, 4) is added to the tree because

weight(4, *near*(4)) is minimum. Then near(4) is set 0.

Step 3 *near*(1) = 2 *weight* = 16

near(5) = 4 *weight* = 18

near(6) = 2 *weight* = 11

As *weight*(6, *near*(6)) is minimum, the node 6 is selected and edge (2, 6) is added to the tree. So *near*(6) is set 0

Step 4 near (1) = 2 weight = 16

near (5) = 4 weight = 18

Next, the edge (2, 1) is added to the tree as *weight*(1, near(1)) is minimum. So *near*(1) is set 0.

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Step 5 near (5) = 1 weight 12

The edge (1, 5) is added to the tree. The Fig. 6.9(a) to 6.9(e) show the step wise construction of MST by Prim's algorithm.

Fig. 6.9 Step wise construction of MST by Prim's algorithm

Time complexity of Prim's Algorithm

Prim's algorithm has three for loops. The first *for* loop finds the near of all nodes which require O(n) time. The second *for* loop is to find the remaining n-2 edges and the third *for* loop updates near of each node after adding a vertex to MST. Since the third for loop is within the second for loop, it requires O(n^2) time. Hence, the overall time complexity of Prim's algorithm is O(n^2).

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Shortest Path Problem

Let us consider a number of cities connected with roads and a traveler wants to travel form his home city A to the destination B with a minimum cost. So the traveler will be interested to know the following:

- Is there a path from city A to city B?
- If there is more than one path from A to B, which is the shortest or least cost path?

Let us consider the graph $G = (V, E)$, a weighting function $w(e)$ for the edges in E and a source node v_0. The problem is to determine the shortest path from v_0 to all the remaining nodes of G. The solution to this problem is suggested by E.W. Dijkstra and the algorithm is popularly known as Dijustra's algorithm.

This algorithm finds the shortest paths one by one. If we have already constructed i shortest paths, then the next path to be constructed should be the next shortest path. Let S be the set of vertices to which the shortest paths have already been generated. For z not in S, let $dist[z]$ be the length of the shortest path starting form v_0, going through only those vertices that are in S and ending at z. Let u is the vertex in S to which the shortest path has already been found. If $dist[z] > dist[u] + w(u,z)$ then $dist[z]$ is updated to $dist[u] + w(u,z)$ and the predecessor of z is set to u. The Dijustra's algorithm is presented in Algorithm 6.6.

Algorithm Dijkstra (v_0, *W*, *dist*, *n*)

// v_0 is the source vertex, *W* is the adjacency matrix to store the weights of the links, *dist[k]* is the array to store the shortest path to vertex *k*, *n* is the number of vertices//

```
{
    for (i = 1 ; i < = n ; i ++){
        S[i] = 0;                    // Initialize the set S to empty i.e. i is not inserted into the set//
        dist[i] = w(v0,i) ; //Initialize the distance to each node
    }
    S[v0] = 1; dist[v0] = 0;
    for(j = 2; j <= n; j++){
        choose a vertex u from those vertices not in S such that dist [u] is minimum.
        S[u] = 1;
        for (each z adjacent to u with S[z] = 0) {
```

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

```
            if(dist[z] >dist [u] + w[u, z])

             dist[z] = dist [u] + w[u, z];

        }

    }

}
```

Algorithm 6.6. Dijkstra's algorithm for finding shortest path.

Let us consider the graph. The objective is to find the shortest path from source vertex 0 to the all remaining nodes.

Iteration	Set of nodes to which the shortest path is found	Vertex selected	Distance				
			0	1	2	3	4
Initial	{0}	-	0	10	∞	5	∞
1	{0, 3}	3	0	8	14	5	7
2	{0, 3,4}	4	0	8	13	5	7
3	{0, 3, 4, 1}	1	0	8	9	5	7
4	{0, 3, 4, 1, 2}	2	0	8	9	5	7

The time complexity of the algorithm is $O(n^2)$.

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 31 - ALL PAIRS SHORTEST PATHS ALGORITHMS

Shortest paths computation is one of the most fundamental problems in graph theory. The huge interest in the problem is mainly due to the wide spectrum of its applications, ranging from routing in communication networks to robot motion planning, scheduling, sequence alignment in molecular biology and length-limited Huffman coding, to name only a very few. The problem divides into two related categories: single-source shortest-paths problems and all-pairs shortest-paths problems. The single-source shortest-path problem in a directed graph consists of determining the shortest path from a fixed source vertex to all other vertices. The all-pairs shortest-distance problem is that of finding the shortest paths between all pairs of vertices of a graph.

APSP ALGORITHMS

The APSP problem is: Given a weighted & directed graph G = (V,E) (where V is the set of vertices and E is the set of edges) with a weight function {w : E

à R}, that maps edges to real valued weights, we wish to find, for every pair of vertices u, v (-V, a shortest (least-weight)) path from u to v, where the weight of a path is the sum of the weights of its constituent edges. Here we assume that there are no cycles with zero or negative weights. The weight function :(W matrix)

Given the fundamental nature of the APSP problem, it is important to consider the desirability of implementing the algorithms in practice. The quest for faster algorithms has led to a surge of interest in APSP.

We take a step in this direction and present a slightly modified Floyd- War shall algorithm that runs faster when applied on a machine. Its asymptotic order remains the same but it decreases the number of computations. Our main interest in this paper will be

A. APSP via Matrix Multiplication It's a dynamic programming algorithm for the APSP problem on a directed graph G= (V, E). It uses the adjacency matrix representation of the graph. Each major loop of the dynamic program will invoke an operation that is very similar to repeated matrix multiplication. The running time of this algorithm improves to O(n3 log n) by using the technique of "repeated squaring".

B. Johnson's Algorithm It finds shortest paths between all pairs in O(V2 lg V + VE) time. For sparse graphs, it is asymptotically better than either repeated squaring of matrices or the Floyd-War shall algorithm. The algorithm either returns a matrix of shortest-path weights for all pairs of vertices or reports that the input graph contains a negative-weight cycle. Johnson's algorithm uses as subroutines both Dijkstra's algorithm and the Bellman-Ford algorithm. Johnson's algorithm uses the technique of re-weighting.

Johnson's algorithm consists of the following steps: 1.First, a new node q is added to the graph, connected by zero-weight edge to each other node.

RISHABH CHAUDHARY Bhagwan Parshuram Institute of Technology, Delhi, India

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

ANKIT SABLOK Bhagwan Parshuram Institute of Technology, Delhi, India

DEEPAK GUPTA Bhagwan Parshuram Institute of Technology, Delhi, India

2. Second, the Bellman-Ford algorithm is used, starting from the new vertex q, to find for each vertex v the least weight h(v) of a path from q to v. If this step detects a negative cycle, the algorithm is terminated.

3.Next the edges of the original graph are reweighted using the values computed by the Bellman-Ford algorithm: an edge from u to v, having length w(u,v), is given the new length w(u,v) + h(u) −h(v). (h : V -> R be any function mapping vertices to real numbers.)

4.Finally, for each node s, Dijkstra's algorithm is used to find the shortest paths from s to each other vertex in the reweighted graph

C. Floyd – Warshall Algorithm The Floyd–Warshall algorithm (sometimes known as the WFI Algorithm or Roy–Floyd algorithm, since Bernard Roy described this algorithm in 1959) is a graph analysis algorithm for finding shortest paths in a weighted, directed graph. A single execution of the algorithm will find the shortest paths between all pairs of vertices. The Floyd–Warshall algorithm is named after Robert Floyd and Stephen Warshall; it is an example of dynamic programming

The algorithm considers the "intermediate" vertices of a shortest path, where an intermediate vertex of a simple path p = v1, v2,..., vl is any vertex of p other than v1 or vl, that is any vertex in the set {v2, v3,..., vl-1}.The Floyd-Warshall algorithm is based on the following observation. The vertices of G are V = {1, 2,..., n}, let us consider a subset {1, 2,..., k} of vertices for some k. For any pair of vertices i, j (- V, consider all paths from i to j whose intermediate vertices are all drawn from {1, 2,..., k}, and let p be a minimum weight path from among them. • If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set {1, 2,..., k - 1}. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set {1, 2,..., k - 1} is also a shortest path from i to j with all intermediate vertices in the set {1, 2,..., k}. • If k is an intermediate vertex of path p, then we break p down into i

à (p1)k

à (p2)j. p1 is a shortest path from i to k with all intermediate vertices in the set {1, 2,..., k}. Because vertex k is not an intermediate vertex of path p1, we see that p1 is a shortest path from i to k with all intermediate vertices in the set {1, 2,..., k - 1}. Similarly, p2 is a shortest path from vertex k to vertex j with all intermediate vertices in the set {1, 2,..., k - 1}. Let dij(k) be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set {1, 2,..., k}. When k = 0, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. Such a path has at most one edge, and hence dij(0) =wij . A recursive definition is:

Because for any path, all intermediate vertices are in the set {1, 2,..., n}, the matrix gives the final answer: dij(n)=shortest path between i and j for all i, j (- V. The following returns the matrix D(n) of shortest-path weights.

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

FLOYD-WARSHALL (W) 1 n ← rows [W] 2 D (0) ← W 3 for k ← 1 to n do 4 for i ← 1 to n do 5 for j ← 1 to n do 6 7 return D(n)

The running time of the Floyd-War shall algorithm is determined by the triply nested for loops of lines 3-6. Because each execution of line 6 takes O (1) time, the complexity of the algorithm is Θ (n3) and can be solved by a deterministic machine in polynomial time. To find all n2 of D (k) from those of D (k-1) requires 2n2 bit operations. Since we begin with D(0) =W and compute the sequence of n matrices the total number of bit operations used is 2n2 * n= n3

MODIFIED FLOYD WARSHALL ALGORITHM

Now we present a slightly modified Floyd-War shall Algorithm which has the same asymptotic running time as of the Floyd-War shall Algorithm. However it involves less number of computations. The modification is achieved by observing and applying a very simple logic: At the kith iteration the values in the ith row (when i= =k) and jth (when j= =k) column do not change in the D(k) matrix i.e. they are the same as in the D(k-1) matrix. This means we need not update the D matrix for the kith row or column as either i = k or j = k during the kth iteration. This saves us a considerable amount of computations thereby saving time when applied on a machine. Also during the kth iteration if any entry of the kth row (say (i = = k) , jth entry) or column(say i , (j = = k) entry) is ∞ then the that jth column or that ith row(respectively) is preserved in the D(k) matrix from the D(k-1) matrix. This also contributes to a lesser number of computations.

The reason for this peculiar property will be given by first considering specific case of D(1) matrix and then it can be understood for any D(k) matrix owing to recursive nature of the solution . Its explained as:

We have D(1) as the matrix where the values give the minimum weight between all vertices i,j (- V for which all intermediate vertices are in the set {1}. So, the weights of the shortest paths involving vertices 1 to j (1..n) and I (1…n) to 1 will be same as there in D(0) matrix (= W) because we can use only vertex 1 as the intermediate vertex here.

Now for any entry of ∞ (means (i,j) !(-(does not belong to)E) in the 1,j(1..n) or (i(1..n),1) th element of the matrix means (1,j) (or (i,1)) !(- E. Now for this jth column (or the ith row) the entries(in the D(1) matrix) will be the same as in the D(0) matrix because now vertex 1 cannot be used as the intermediate vertex for this jth column (or the ith row) in the D(1) matrix.

The Modified Floyd Warshall's algorithm is given below.

Modified-FLOYD-WARSHALL (W) 1 n ← rows [W] 2 D (0) ← W 3 for k ← 1 to n do 4 for i ← 1 to n do 5 If(dki==∞ || i==k) do 6 continue; 7 else 8 { 9 for j ← 1 to n do 10 If (j==k || djk==∞) do 11 continue; 12 else 13 14 } 15 return D (n)

COMPARISON OF APSP ALGORITHMS A. Comparison of Floyd-Warshall, Johnson & APSP via MM, modified floyd-warshalls..

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

The running time of APSP via Matrix Multiplication is O(n3 log n) by using the technique of “repeated squaring”. The time complexity of Johnson’s algorithm, using Fibonacci heaps in the implementation of Dijkstra's algorithm, is O(V2log V + VE); the algorithm uses O(VE) time for the Bellman-Ford stage of the algorithm, and O(V log V + E) for each of V instantiations of Dijkstra's algorithm.

F-W and Mod F-W both run in Θ(n3) time.

We have plotted a graph between the order of complexity and the number of vertices in the graph for all the APSP algorithms as shown below.

Fig 1.1 Comparison of all APSP Algorithms

(Note: For n = = 0, Johnson’s and APSP via MM are not defined & Johnson’s for dense graph has the same complexity as that of FW.)

B. Comparison of F-W and its Modified version

We applied the Floyd Warshall Algorithm and its modified version to a number of weighted, directed graphs and analyzed the times taken for the computations in both the cases. One such example is:

Floyd-Warshall solves this in ≈ 0.16s while Modified FloydWarshall solves this in ≈ 0.10 s. Modified version solved this problem, saving time of ≈ 33.33% over Floyd-Warshall.

CONCLUSION Among Floyd Warshall ,Johnson’s, and APSP via MM ,Floyd War shall is quite simple , efficient and achieves a good running time of Θ(n3),but when the graph is sparse, the total time for Johnson’s is faster than the Floyd-Warshall algorithm. Modified F-W has the same order as that of F-W but runs faster when implemented on a machine owing to the less number of computations.

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 32 - Backtracking And Branch And Bound

Subset & Permutation Problems

•Subset problem of size n.

Nonsystematic search of the space for the answer takes O(p2n)time, wherep is the time needed to evaluate each member of the solution space. •Permutation problem of size n. •Permutation problem of size n.

Nonsystematic search of the space for the answer takes O(pn!)time, wherep is the time needed to evaluate each member of the solution space. •Backtracking and branch and bound perform a systematic search; often taking much less time than taken by a nonsystematic search.

Tree Organization Of Solution Space •Set up a tree structure such that the leaves represent members of the solution space. •For a size nsubset problem, this tree structure has 2n leaves. •For a sizenpermutation problem, this tree structure has n! leaves. •The tree structure is too big to store in memory; it also takes too much time to create the tree structure. •Portions of the tree structure are created by the backtracking and branch and bound algorithms as needed.

Subset Problem

•Use a full binary tree that has 2n leaves. •At level i the members of the solution space are partitioned by their xivalues. are partitioned by their xivalues. •Members with xi = 1 are in the left subtree. •Members with xi = 0 are in the right subtree. •Could exchange roles of left and right subtree.

Subset Tree For n = 4

x1=1x1= 0

x=1x= 0 x 2=1x2= 0x2=1x2= 0

x3=1x3= 0

x4=1x4=0

1110101101110001

Permutation Problem

•Use a tree that has n!leaves. •At level i the members of the solution space are partitioned by their xivalues. are partitioned by their xivalues. •Members (if any) with xi = 1 are in the first subtree. •Members (if any) with xi = 2 are in the next subtree. •And so on.

Permutation Tree For n = 3

x1=1x1=2x1= 3

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

x= 2x= 3x= 1x= 3x= 1x= 2 x2= 2x2= 3x2= 1x2= 3x2= 1x2= 2

x3=3x3=2x3=3x3=1x3=2x3=1

123132213231312321

Backtracking

•Search the solution space tree in a depth first manner. •May be done recursively or use a stack to •May be done recursively or use a stack to retain the path from the root to the current node in the tree. •The solution space tree exists only in your mind, not in the computer.

Backtracking Depth-First Search

x1=1x1= 0

x=1x= 0 x 2=1x2= 0x2=1x2= 0

Backtracking Depth-First Search

x1=1x1= 0

x=1x= 0 x 2=1x2= 0x2=1x2= 0

Backtracking Depth-First Search

x1=1x1= 0

x=1x= 0 x 2=1x2= 0x2=1x2= 0

Backtracking Depth-First Search

x1=1x1= 0

x=1x= 0 x 2=1x2= 0x2=1x2= 0

Backtracking Depth-First Search

x1=1x1= 0

x=1x= 0 x 2=1x2= 0x2=1x2= 0

O(2n) Subet Sum & Bounding Functions

x1=1x1= 0

x=1x= 0

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

{10, 5, 2, 1}, c = 14

x2=1x2= 0x2=1x2= 0

Each forward and backward move takes O(1) time.

Bounding Functions •When a node that represents a subset whose sum equals the desired sum c, terminate. •When a node that represents a subset whose sum exceeds the desired sum c, backtrack. I.e., do not enter its sub trees, go back to parent node. enter its sub trees, go back to parent node. •Keep a variable r that gives you the sum of the numbers not yet considered. When you move to a right child, check if current subset sum + r >= c. If not, backtrack.

Backtracking

•Space required is O(tree height). •With effective bounding functions, large instances can often be solved. •For some problems (e.g., 0/1 knapsack), the •For some problems (e.g., 0/1 knapsack), the answer (or a very good solution) may be found quickly but a lot of additional time is needed to complete the search of the tree. •Run backtracking for as much time as is feasible and use best solution found up to that time.

Branch And Bound

•Search the tree using a breadth-first search (FIFO branch and bound). •Search the tree as in a bfs, but replace the FIFO queue with a stack (LIFO branch and bound). •Replace the FIFO queue with a priority queue (least-cost (or max priority) branch and bound). The priority of a node p in the queue is based on an estimate of the likelihood that the answer node is in the sub tree whose root sip.

Branch And Bound

•Space required is O(number of leaves). •For some problems, solutions are at different levels of the tree (e.g., 16 puzzle).

1234 5678 9101112 131415

1

3 2

4

5 6 13 14

15

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

12 1110 97 8

Branch And Bound

FIFO branch and bound finds solution closest to root.

Backtracking may never find a solution because tree depth is infinite (unless repeating configurations are eliminated). •Least-cost branch and bound directs the search to •Least-cost branch and bound directs the search to parts of the space most likely to contain the answer. So it could perform better than backtracking

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 33 - Fourier transforms and Rabin-Karp Algorithm

In the previous section, great care was taken to restrict our attention to particular spaces of functions for which Fourier transforms are well-defined. That being said, it is often necessary to extend our definition of FTs to include “non-functions”, including the Dirac “delta function”. In this section, we also show, very briefly, the importance of the delta function in the analysis of functions that are defined on the entire real line R.

Recall that the delta function δ(x) is not a function in the usual sense. It has the following properties:

δ(x) =□ □ □

0, x 6= 0, ∞, x = 0,

(1)

with the additional feature that

Z ∞ −∞

δ(x) dx = 1. (2)

Actually, the Dirac delta function is an example of a distribution – distributions are defined in terms of their integration properties. For any function f(x) that is continuous at x = 0, the delta distribution is defined as

Z ∞ −∞

f(x)δ(x) dx = f(0). (3)

If f(x) is continuous at x = x0, then Z ∞ −∞

f(x)δ(x−x0) dx = f(x0). (4)

The Fourier transform of the Dirac distribution is easily calculated from the above property. For the distribution positioned at x = 0:

F(ω) = F(δ(x)) =

1 2πZ ∞ −∞

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

δ(x)eiωx dx = 1 2π

. (5)

With reference to the sketches below, note that the delta function δ(x) is a perfect “spike”, i.e., it is concentrated at x = 0, whereas its Fourier transform is a constant function for all x ∈ R, i.e., it is “spread out” as much as possible.

This illustrates the complementarity between the “spatial domain”, i.e., x-space (or “temporal

domain,” i.e., t-space, if x is replaced by t) and the “frequency domain, i.e., ω-space, Note also that

223

x ω 00

f(x) = δ(x) F(ω) = 1 2π

1 2π

neither δ(x) nor its Fourier transform F(ω) = 1/(2π) belong to L2(R), the space of square integral

Functions on R. Now suppose that the delta function is translated to x = x0, i.e., we replace x with x−x0: F(ω) = F(δ(x−x0)) = 1 2πZ ∞ −∞ δ(x−x0)eiωx dx = 1 2π eiωx0. (6) Note that in all cases, F(ω) is not square-integrable. But, then again, the delta function δ(x −x0) does not belong to L2(R) as well.

Let’s now return to the formal definition of the Fourier transform of a function f(x) for this course

(assuming that the integral exists),

F(ω) =

1 2πZ ∞ −∞

f(x)eiωx dx, (7)

and the associated inverse Fourier transform, f(x) =Z ∞ −∞

F(ω)e−iωx dω. (8)

As we discussed earlier, in Eq. (8), the terms F(ω) may be regarded as the expansion coefficients of

f(x) in terms of the basis functions

φω(x) = e−iωx. (9)

In other words, we may view Eq. (8), when written as follows, f(x) =Z ∞ −∞ F(ω)φω(x) dω, (10)

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

as a continuous version of the following expansion of a function g(x) that is defined over a finite interval [a,b] and expressed in terms of a discrete set of functions ϕn(x), n = 1,2,⋯, that form a basis on [a,b]:

g(x) = ∞

X n=1

cnϕn(x). (11)

224

The discrete summation over the integer-valued index n in Eq. (11) has been replaced by a continuous integration over the real-valued index ω in Eq. (8). We'll have more to say about Eq. (8) later.

Let us now substitute our results for the Dirac delta function and its Fourier transform, i.e.,

f(x) = δ(x), F(ω) =

1 2π

, (12)

into Eq. (8):

δ(x) =

1 2πZ ∞ −∞

e−iωx dω. (13)

If we replace x with x−x0, this equation becomes δ(x−x0) = 1 2πZ ∞ −∞

e−iω(x−x0) dω. (14)

Eqs. (13) and (14) are known as the "integral representations" of the Dirac delta function. Note that the integrations are performed over the frequency variable ω.

Let us now consider the following case,

F(ω) = δ(ω). (15)

We wish to find the inverse Fourier transform of the Dirac delta function in ω-space. In other words, what is the function f(x) such that F(f) = δ(ω)? If we substitute F(ω) = δ(ω) into Eq. (8), then f(x) =Z ∞ −∞ δ(ω)e−iωxd ω. (16)

But recall that an integration of the Dirac delta function δ(whatever) yields f(whatever = 0). This means that

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

f(x) = e−i·0·x = 1. (17)

Therefore, the inverse Fourier transform of δ(ω) is the function f(x) = 1. This time, the function δ(ω) in frequency space is spiked, and its inverse Fourier transform f(x) = 1 is a constant function spread over the real line, as sketched in the figure below.

Let us now substitute this result into Eq. (7), i.e., f(x) = 1 and F(ω) = δ(ω). We then have

δ(ω) =

1 2πZ ∞ −∞

eiωx dx. (18)

Now let ω = ω1 −ω2 so that the above equation becomes δ(ω1 −ω2) = 1 2πZ ∞ −∞ ei(ω1−ω2)xdx. (19)

00

F(ω) = δ(ω)

x

f(x) ω

1

Let us rewrite this result slightly, i.e.,

δ(ω1 −ω2) =

1 2πZ ∞ −∞

eiω1xe−iω2xdx. (20)

Eq. (20) may be viewed as an orthogonality relation for the functions ϕω(x) = e−iωx defined earlier. Recalling the definition of the inner product in the Hilbert space L2(R), we may rewrite Eq. (20) as

1 2πhϕω1,ϕω2i = δ(ω1 −ω2). (21)

We may go one step further and define the functions

ψω(x) =

1 √2πϕω(x) =

1 √2πe−iωx, ω ∈ R, (22)

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

so that Eq. (21) becomes

hψω1,ψω2i= δ(ω1 −ω2). (23) Note that this may be viewed as a continuous version of the relations encountered for functions{χn}∞ n=1 that form an orthonormal set on a finite interval [a,b], i.e., hχn,χmi=⍰ ⍰ ⍰ 1, n = m 0, n 6= m. (24) In going from Eq. (24) to Eq. (23), the discrete indices n and m become the continuous indices ω1 and ω2, and the constant 1 on the RHS becomes ∞ because of the Dirac delta function. This is the price to be paid in going from the discrete case to the infinite case. Moreover, note once again that the functions ψω(x) are not elements of L2(R) even though they form a basis for the space! Physicists

refer to these functions as (one-dimensional) plane waves.

Plane waves are very important in physics. It's not too hard to imagine that they would be

important in classical optics or electromagnetism, where light or electromagnetic radiation is viewed

in terms of waves. In quantum mechanics, plane waves are important in "scattering theory:" For

example, a particle X travels toward another particle Y and interacts with it, thereby being "scattered"

or deflected. The quantum mechanical wavefunction of the particle, before and after the interaction,

may be expressed in terms of plane waves.

This agrees with comments made earlier in this lecture: If we choose to "expand" a function f(x)

defined over the real line R in terms of plane waves, integrating over its ω index, i.e., f(x) = Z ∞ −∞ A(ω)ψω(x) dω

=

1 √2πZ ∞ −∞

A(ω)e−iωx dω, (25)

then the "coefficients" A(ω) of the expansion comprise, up to a constant, the Fourier transform of

f(x), cf. Eq. (8).

We conclude this section with a couple of intriguing results. Let us return to the integral representation of the Dirac delta function δ(x−x0) in Eq. (14): δ(x−x0) = 1 2πZ ∞ −∞ e−iω(x−x0) dω. (26)

We'll first rewrite this equation as follows, δ(x−x0) = Z ∞ −∞

1 √2πeiωx0 1 √2πe−iωx dω

= Z ∞ −∞

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

ψ∗ ω(x0)ψω(x) dω. (27)

Note that this is an integration over ω involving functions evaluated at x and x0 (which may be the

same, or may not). We claim that the above equation is the continuous analogue of the following result: Let {χn}∞ n=1 be a set of orthonormal basis functions on an interval [a,b], as given by Eq. (24). Then for any point x0 ∈(a,b), δ(x−x0) = ∞ X n=1 χ∗ n(x)χn(x0), (28) This may be viewed as an eigenfunction expansion of the Dirac delta function δ(x − x0). It is an important result that has applications in the solution of ODEs and PDEs, in context of Green's

functions.

227

We leave the proof of this result as an exercise. Hint: Multiply each side of Eq. (28) by a continuous

function f(x) and consider the integral of each side over R.

The two-dimensional Fourier transform

Relevant section of text: 10.6.5

The definition of the Fourier transform for a function of two variables, i.e., f : R2 → R, is a rather straightforward extension of the one-dimensional FT. It is notationally convenient to let

(x1,x2) represent the two spatial variables, i.e., f(x1,x2). Since there are two spatial variables, we

must have two frequency variables, w1 and w2, which will also be written as a pair: ω = (ω1,ω2).

The Fourier transform of a function f(x1,x2) is defined as

F(ω1,ω2) =

1 (2π)2 Z ∞ −∞Z ∞ −∞

f(x1,x2)eiω1x1eiω2x2dx1 dx2. (29)

Note that each variable introduces a factor of 1/(2π). The above equation can be written in a more

compact way using vector notation, i.e.,

~ω = (ω1,ω2), ~r = (x1,x2), so that ω~·~r = ω1x1 + ω2x2. (30)

Then

F(~ω) =

1 (2π)2 ZR2 f(~r)eiω~·~rd~r, (31)

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

where d~r = dx1dx2 or dx2dx1.

The 2D inverse Fourier transform will be defined as f(~x) =Z ZR2 F(~ω)e−iω~·~rdω~. (32)

We now consider the heat equation on R2,

∂u ∂t

= ∇2u = ∂2u ∂x2 1

+

∂2u ∂x2 2

(33)

with initial condition

u(x1,x2,0) = f(x1,x2). (34)

228

The (unique) solution u(x1,x2) to this problem may be produced using the FT in the same way as

was done in 1D. Very briefly, we take 2D FTs of both sides of Eq. (33) – the rules for FTs of partial

derivatives will apply again – to arrive at the following PDE for the FT U(ω1,ω2) of u:

∂U ∂t

= kω2U, where ω2 = ω2 1 + ω2 2. (35)

Once again, since only the partial time derivative appears in the equation, it may be solved as an

ODE. The solution is easily found to be

U(ω1,ω2,t) = F(ω1,ω2)e−kω2t, (36)

where

F(ω1,ω2) = F(f(x1,x2)). (37) We now take the inverse Fourier transform of each side to obtain u(x1,x2). The IFT of the RHS is

obtained from a two-dimensional Convolution Theorem (see text, p. 498). First of all, the IFT of the

Gaussian on the RHS is assisted by the fact that it is separable, i.e.,

G(ω1,ω2) = e−kω2t = e−kω2 1te−kω2 2t

= G1(ω1)G2(ω2). (38)

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

The inverse FT of this function is then a product of the IFTs of G1 and G2, which we know from the one-dimensional case,

g1(x1) =rπ kt

e−x2 1/4kt, g2(x2) =rπ kt

e−x2 2/4kt. (39)

As a result, the solution will be a convolution of these functions and the initial data function

f(x1,x2). The final result is u(x1,x2,t) = Z ZR2 f(s1,s2) 1 4πkt

e−[(x1−s1)2+(x2−s2)]/(4kt)ds1 ds2 = Z ZR2 f(s1,s2)ht(x1 −s1,x2 −s2)ds1 ds2. (40)

Here, ht(x1,x2) is the two-dimensional heat kernel, a two-dimensional, normalized Gaussian distribute tin. It is the product of the one-dimensional heat kernels in the x and y directions.

In the special case that the initial condition is concentrated at a point (x0,y0), i.e.,

f(x1,x2) = δ(x1 −a1,x2 −a2), (41)

then the solution u(x,t) to the heat equation becomes u(x1,x2,t) = Z ZR2 δ(s1 −a1,s2 −a2) 1 4πkt

e−[(x1−s1)2+(x2−s2)]/(4kt)ds1 ds2

=

1 4πkt

e−[(x1−a1)2+(x2−a2)2]/(4kt), t > 0. (42)

At time t > 0, the temperature function u(x1,x2,t) is a Gaussian function centered at (a1,a2) which spreads out with increasing time.

String Matching

CLRS Chapter 32

- Defination of string matching
- Naive string-matching algorithm
- String matching algorithms – an overview

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

• Rabin-Karp algorithm

• Finite automata

• Linear time matching using finite automata

• (Knuth-Morris-Pratt algorithm)

Martin Zachariasen, DIKU

May 18, 2009

1

String-matching problem

Given:

• Text T[1..n]

• Pattern P[1..m], where $m \leq n$

Characters of text and pattern are drawn from a common finite alphabet Σ: $T \in \Sigma*$ and $P \in \Sigma*$.

Find: All occurrences of pattern P in T, that is, all valid shifts s, where $0 \leq s \leq n - m$, such that

$T[s + 1..s + m] = P[1..m]$

or

$T[s + j] = P[j], j = 1, ..,m$

2

Naive string-matching algorithm

Iterate through all shifts s, and for each of these check if the shift is valid: $T[s + j] = P[j], j = 1, ..,m$.

Clearly takes time $\Theta((n - m + 1)m)$, or $\Theta(n2)$ if $m = \lfloor n/2 \rfloor$.

More clever algorithms use information obtained when checking one value of s in the following iteration(s).

3

String-matching algorithms — an overview

Divide running time into preprocessing and matching time.

Preprocessing: Setup some data structure based on pattern P.

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Matching: Perform actual matching by comparing characters from T with P and precomputed data structure.

String-matching algorithms considered:

Algorithm Preprocessing time Matching time Naive 0 Θ((n − m + 1)m) Rabin-Karp Θ(m) Θ((n − m + 1)m) Finite automation O(m|Σ|) Θ(n) (Knuth-Morris-Pratt) Θ(m) Θ(n)

Note: Rabin-Karp uses O(n) expected matching time.4

Rabin-Karp algorithm

Consider (sub) strings as numbers. Characters in a string correspond to digits in a number written in radix-d notation (where d = |Σ|).

Numerical value p corresponding to pattern P[1..m]: p = P[1]dm−1 + P[2]dm−2 + ... + P[m − 1]d + P[m]

or by using Horner's rule:

p = P[m]+d(P[m−1]+d(P[m−2]+...+d(P[2]+dP[1])...))

Let ts correspond to the decimal value of T[s + 1..s + m].

Main observation: Valid shift s is obtained if and only if p = ts.

5

Fast computation of text string numbers

Assume that we have computed t0,...,ts.

Question: How can we compute ts+1 efficiently?

Answer: Just need to drop the most significant digit from ts and append the least significant digit from ts+1. Let h = dm−1. Then we have:

ts+1 = d(ts − T[s + 1]h) + T[s + m + 1]

Thus given ts we can compute ts+1 in constant time — assuming that arithmetic operations take constant time.6

Reducing the size of decimal numbers

Problem: Numbers p and ts cannot be computed or compared in constant time!

Solution: Compute all numbers modulo some (small) number q.

Basic facts on modulus computations:

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

• Remainder/residue of a division: The number

r = a mod q is the remainder of the integer division a/q, or the unique number $0 \leq r < q$ such that $a = kq + r$ (where k is the result of the integer division).

• Equivalence classes modulo q: For two integers a and b we have that $a \equiv b \pmod{q}$ if and only if there exists some number k such that

$a - b = kq$

7

Properties of modified algorithm

New main observation: When

$p \equiv ts \pmod{q}$ then we either have a valid shift s or a so-called spurious hit.

Need to check every such hit explicitly. Takes O(m) time for each hit.

The expected number of spurious hits is O(n/q). If v is the number of valid shifts, the expected matching time is

$O(n) + O(m(v + n/q))$ which is O(n) if v is a constant and $q \geq m$.

8

Finite automata

We may build a finite automaton that recognizes pattern P. More precisely, the automaton should recognize all strings x such that P is a suffix of x: $P \sqsupset x$.

Why? A shift s is valid if and only if $P \sqsupset T[1..m + s]$.

Note that the automaton is built in the preprocessing phase and uses only the pattern P as input.

Some notation on finite automata:

• Q is the set of states (where $q0 \in Q$ is the start state),

• $A \subseteq Q$ is the set of accepting states,

• δ is the transition function,

• ϕ is the (implicit) final-state function: $\phi(x)$ is the state that the automation ends in after scanning string x.

9

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Building a string-matching automation

Need to define the so-called suffix function σ which maps any string $x \in \Sigma*$ to the set {0,1,...,m} according to

$\sigma(x) = \max\{k : P_k \sqsupset x\}$ where P_k is the prefix P[1..k].

The finite automation will have the following properties:

• Set of states Q = {0,1,...,m}, start state q0 = 0, and only accepting state A = {m}.

• Transition function: $\delta(q,a) = \sigma(Pqa)$

• Invariant maintained while reading the text T is

$\phi(T_i) = \sigma(T_i)$ where T_i is the prefix T[1..i]. The state number should be equal to the length of the longest prefix of P that is a suffix of T_i.

10

Correctness of finite automation

Need to prove that the machine is in state $\sigma(T_i)$ after scanning character T[i].

Proceed in two steps:

1. (Lemma 32.3) For any string T_i and character a, if $q = \sigma(T_i)$, then $\sigma(T_ia) = \sigma(Pqa)$

2. (Theorem 32.4) For all i = 0,1,...,n, we have

$\phi(T_i) = \sigma(T_i)$

11

Proof of property 1

For any string T_i and character a, if $q = \sigma(T_i)$, then

$\sigma(T_ia) = \sigma(Pqa)$

We cannot have $\sigma(T_ia) > q + 1$ since this would imply that $\sigma(T_i) > q$.

However, if $P_{q+1} \sqsupset T_ia$ then $\sigma(T_ia) = q + 1$.

Since $q = \sigma(T_i)$ we have $P_q \sqsupset T_i$. Thus computing $\sigma(T_ia)$ is the same as computing $\sigma(Pqa)$ since $\sigma(T_ia) \leq q + 1$.

12

Proof of property 2

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

For all i = 0,1,...,n, we have

$\phi(T_i) = \sigma(T_i)$

Proof by induction on i.

Basis: $\phi(T_0) = 0 = \sigma(T_0)$, since T_0 is the empty string.

Inductive step: Assume that $\phi(T_i) = \sigma(T_i)$. Would like to prove that $\phi(T_{i+1}) = \sigma(T_{i+1})$.

Let $\phi(T_i) = q$. By induction we have $\sigma(T_i) = q$, and hence by property 1 that $\sigma(T_ia) = \sigma(P_qa)$ for any character a.

Let $a = T[i + 1]$. Now we have

$\phi(T_{i+1}) = \phi(T_ia)$ [by the definition of a] $= \delta(\phi(T_i),a)$ [by the definition of ϕ] $= \delta(q,a)$ [by the definition of q] $= \sigma(P_qa)$ [by the definition of δ] $= \sigma(T_ia)$ [as argued above] $= \sigma(T_{i+1})$ [by the definition of T_{i+1}]

13

Computing the transition function

May use a straight-forward $O(m^3|\Sigma|)$ time algorithm:

For each state $q \in Q$ and character $a \in \Sigma$ find the maximum $k \in \{0,1,...,m\}$ such that

$P_k \sqsupset P_qa$

The result is defined to be the value of $\delta(q,a)$.

There are m+1 states, $|\Sigma|$ characters, at most m+1 possible values of k and at most m characters to check for the condition $P_k \sqsupset P_qa$.

Possible to devise an algorithm that runs in $O(m|\Sigma|)$ time.

14

(Knuth-Morris-Pratt algorithm)

Similar to finite automation, but avoids explicit computation of $\delta(q,a)$.

Only needs one auxiliary function $\pi[1..m]$ that can be computed from P in $\Theta(m)$ time:

$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$

We compute $\delta(q,a)$ iteratively by using the function π.

The amortized cost is $\Theta(m)$ for preprocessing and $\Theta(n)$ for matching.

15

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Applied Algorithms Date: January 24, 2012

Rabin-Karp algorithm

Professor: Antonina Kolokolova Scribe: Md. Asaduzzaman

1 String Matching

1.1 Rabin-Karp algorithm

Rabin-Karp string searching algorithm calculates a numerical (hash) value for the pattern p, and for each m-character substring of text t. Then it compares the numerical values instead of comparing the actual symbols. If any match is found, it compares the pattern with the substring by naive approach. Otherwise it shifts to next substring of t to compare with p. We can compute the numerical (hash) values using Horner's rule. Lets assume, h0 = k h1 = dk−p[1].dm−1+ p[m + 1] Suppose, we have given a text t = [3, 1, 4, 1, 5, 2] and m = 5, q = 13; t0 = 31415 So t1 = 10(31415 - 105−1.t[1]) + t[5+1] = 10(31415−104.3) + 2 = 10(1415) + 2 = 14152 Here p and substring ti may be too large to work with conveniently. The simple solution is, we can compute p and the ti modulo a suitable modulus q. So for each i, hi+1 = (dhi −t[i + 1].dm−1+ t[m + i + 1]) mod q The modulus q is typically chosen as a prime such that d.q fits within one computer word. Algorithm Compute hp (for pattern p) Compute ht (for the first substring of t with m length) For i = 1 to n−m If hp = ht Match t[i....i + m] with p, if matched return 1 Else ht = (dht −t[i + 1].dm−1+ t[m + i + 1]) mod qEnd Suppose, t= 2359023141526739921 and p = 31415, Now, hp = 7 (31415 = 7 (mod 13)) substring beginning at position 7 = valid match

This algorithm has a significant improvement in average-case running time over naive approach

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 34 - NP-Hard and NP-Complete Problems((Vertex-Cover Problem))

Aims:

• To describe SAT, a very important problem in complexity theory; • To describe two more classes of problems: the NP-Hard and NP Complete problems.

33.1. SAT

33.1.1. Description of SAT • We start by looking at a decision problem that plays a major role in complexity theory. The nice thing is that it gives us another example of a problem that is in NP. • Revision – A wff in propositional logic comprises propositional symbols (e.g. p, p1, p2,..., q, q1, q2) combined using connectives (¬, ∧, ∨, ⇒, ⇔). – An interpretation, I, stipulates the truth-values of propositional symbols (e.g. p is true, q is false, etc.) – A wff W is satisfiable iff there is at least one interpretation that makes W true. • Now here is the important problem we mentioned above:

Problem 33.1. SAT Parameters: A wff of propositional logic, W. Returns: YES if W is satisfiable; NO otherwise.

• What would SAT return for these instances? – p∨q – p∧¬p – p∨¬p

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

– p1 ∨(p1 ⇒((p2 ⇒(p3 ∧¬p4)⇔ p5)) – p • (Textbook presentations of SAT are sometimes slightly deferent from this one. Sometimes they only allow a subset of the connectives, typically just ∧, ∨ and ¬. Often they insist that the wff be in a special format called conjunctive normal form. This can make some proofs easier because it gives fewer connectives to consider. But none of it makes any difference to what we're doing.) • SAT is a problem for which we know no polynomial-time algorithm. • Yet, we have no proof that it is intractable (i.e. no proof that there cannot be a polynomial-time algorithm). • The only algorithms we have take worst-case exponential time in n, where n is the number of propositional symbols. • Here is an outline of one obvious exponential-time algorithm: while there are untried interpretations { generate the next interpretation, I; if I satisfies W { return YES; } } return NO;

• Class Exercise: Why does this take worst-case exponential-time?

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

• We can also show that SAT is in NP. • Assume we have a wff W containing n distinct propositional symbols. Then here's an ND-DECAFF algorithm:

// The guessing part for each distinct propositional symbol in W { v := choose(0,1); if v = 0 { Assign false to the propositional symbol; } else { Assign true to the propositional symbol; } } // The checking part Evaluate W using the truth-values from above and the truth-tables for ¬, ∧, ∨, ⇒, ⇔;

• Both parts of the algorithm (the guessing and the checking) take polynomial time. • This shows that SAT is in NP.

33.2. NP-Hard and NP-Complete Problems

33.2.1. NP-Hard Problems • We say that a decision problem Pi is NP-hard if every problem in NP is polynomial time reducible to Pi. • In symbols, Pi is NP-hard if, for every Pj ∈ NP, Pj poly $-\!\!\rightarrow$ Pi. • Note that this doesn't require Pi to be in NP. • Highly informally, it means that Pi is 'as hard as' all the problems in NP. – If Pi can be solved in polynomial-time, then so can all problems in NP. – Equivalently, if any problem in NP is ever proved intractable, then Pi must also be intractable.

33.2.2. NP-Complete Problems • We say that a decision problem Pi is NP-complete if – it is NP-hard and – it is also in the class NP itself. • In symbols, Pi is NP-complete if Pi is NP-hard and Pi ∈NP • Highly informally, it means that Pi is one of the hardest problems in NP.

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

• So the NP-complete problems form a set of problems that may or may not be intractable but, whether intractable or not, are all, in some sense, of equivalent complexity. • If anyone ever shows that an NP-complete problem is tractable, then – every NP-complete problem is also tractable – indeed, every problem in NP is tractable and so P = NP. • If anyone ever shows that an NP-complete problem is intractable, then – every NP-complete problem is also intractable and, of course, P6= NP. • So there are two possibilities:

P

NP–complete NP P = NP

We don’t know which of these is the case. • But this gives Computer Scientists a clear line of attack. It makes sense to focus efforts on the NP-complete problems: they all stand or fall together. • So these sound like very significant problems in our theory. But how would you show that a decision problem is NP-complete?

SAT

NP-Hard and NP-...

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

• How to show a problem Pi is NP-complete (Method 1, from the definition) – First, confirm that Pi is a decision problem. – Then show Pi is in NP. – Then show that Pi is NP-hard by showing that every problem Pj in NP is polynomial-time reducible to Pi. ∗ You wouldn't do this one by one! ∗ You would try to make a general argument.

33.2.3. An NP-Complete Problem • Definitions are all very well. But has anyone ever found an actual NP-complete problem? Yes! • SAT is NP-complete. • How was this proved? By method 1. • First, SAT is a decision problem. • Second, SAT is in NP. – We proved this earlier. • Then it was shown SAT is NP-hard by showing that every problem in NP is polynomial-time reducible to SAT – This wasn't done one by one. – It was done by a general argument

SAT

NP-Hard and NP-...

poly SAT All problems in NP

• The proof is beyond the scope of this course and the result goes by the name of the Cook-Levin Theorem

33.2.4. How to Show Other Problems are NP-Complete • We have one problem that is proven to be NP-complete, where the proof is done generically and 'from scratch'. Showing that other problems are NP-complete is easier. • How to show decision problem Pi is NP-complete (Method 2) – First, confirm it is a decision problem. – Then show Pi is in NP. – Then show that Pi is NP-hard by taking just one problem Pj

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

that is already known to be NP-complete and showing that Pj poly −→ Pi • Why does the latter show Pi to be NP-hard? If Pj is NP-complete, then we know that Pj is NP-hard (by the definition of NP-complete), i.e. every problem in NP is polynomially-reducible to Pj. But, if every problem in NP is polynomially-reducible to Pj and Pj is polynomially-reducible to Pi then, by transitivity, every problem in NP is polynomially-reducible to Pi.

SAT

• Starting with SAT and using Method 2, numerous problems have been shown to be NP-complete. • Without going into the details of the problems or the reductions themselves, here is a picture that shows a few of the polynomial-time reductions that have been found.

poly poly

poly

poly

SAT 3SAT VertexCover

Clique

SetCover

SubsetSum

Hamiltonian Cycle Decision Problem

Knapsack

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

TSP DP

poly poly poly

poly

• Here’s a picture showing some actual decision problems.

ECDP Membership of finite−lenght list Non−membership of finite−length list P

Graph−isomorphism

TSPDP HCDP SAT

NP

NP−complete

SAT

NP-Hard and NP-...

Acknowledgements:

This material owes a little to [GJ79], [GT02] and [Man89]. Clip Art (of head with bomb) licensed from the Clip Art Gallery on DiscoverySchool.com.

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

References

[GJ79] M.R. Garey and D.S. Johnson. Computers and Intractability. W.H. Freeman, 1979. [GT02] M. T. Goodrich and R. Tamassia. Algorithm Design: Foundations, Analysis, and Internet Examples. Wiley, 2002. [Man89] U. Member. Introduction to Algorithms: A Creative Approach. Addison Wesley,

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 35 - Approximation Algorithms(Vertex-Cover Problem)

Overview

Suppose we are given an NP-complete problem to solve. Even though (assuming P 6= NP) we can't hope for a polynomial-time algorithm that always gets the best solution, can we develop polynomial-time algorithms that always produce a "pretty good" solution? In this lecture we consider such approximation algorithms, for several important problems. Specific topics in this lecture include:

• 2-approximation for vertex cover via greedy matchings.

• 2-approximation for vertex cover via LP rounding.

• Greedy O(logn) approximation for set-cover.

• Approximation algorithms for MAX-SAT.

Introduction

Suppose we are given a problem for which (perhaps because it is NP-complete) we can't hope for a fast algorithm that always gets the best solution. Can we hope for a fast algorithm that guarantees to get at least a "pretty good" solution? E.g., can we guarantee to find a solution that's within 10% of optimal? If not that, then how about within a factor of 2 of optimal? Or, anything non-trivial? As seen in the last two lectures, the class of NP-complete problems are all equivalent in the sense that a polynomial-time algorithm to solve any one of them would imply a polynomial-time algorithm to solve all of them (and, moreover, to solve any problem in NP). However, the difficulty of getting a good approximation to these problems varies quite a bit. In this lecture we will examine several important NP-complete problems and look at to what extent we can guarantee to get approximately optimal solutions, and by what algorithms.

VERTEX COVER 92

Vertex Cover

Recall that a vertex cover in a graph is a set of vertices such that every edge is incident to (touches) at least one of them. The vertex cover problem is to find the smallest such set of vertices.

Definition 21.1 Vertex-Cover: Given a graph G, find the smallest set of vertices such that every edge is incident to at least one of them. Decision problem: "Given G and integer k, does G contain a vertex cover of size ≤ k?"

For instance, this problem is like asking: what is the fewest number of guards we need to place in a museum in order to cover all the corridors. As we saw last time, this problem is NP-hard. However, it

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

turns out that for any graph G we can at least get within a factor of 2. Let's start first, though, with some strategies that don't work.

Straw man Alg #1: Pick an arbitrary vertex with at least one uncovered edge incident to it, put it into the cover, and repeat.

What would be a bad example for this algorithm? [Answer: how about a star graph]

Straw man Alg #2: How about picking the vertex that covers the most uncovered edges. This is very natural, but unfortunately it turns out this doesn't work either, and it can produce a solution $\Omega(\log n)$ times larger than optimal.1

How can we get factor of 2? It turns out there are actually several ways. We will discuss here two quite deferent algorithms. Interestingly, while we have several algorithms for achieving a factor of 2, nobody knows if it is possible to efficiently achieve a factor 1.99.

Algorithm 1: Pick an arbitrary edge. We know any vertex cover must have at least 1 endpoint of it, so let's take both endpoints. Then, throw out all edges covered and repeat. Keep going until there are no uncovered edges left.

Theorem 21.1 The above algorithm is a factor 2 approximation to Vertex-Cover.

1The bad examples for this algorithm are a bit more complicated however. One such example is as follows. Create a bipartite graph with a set SL of t nodes on the left, and then a collection of sets SR,1,SR,2,... of nodes on the right, where set SR,i has $\lfloor t/i \rfloor$ nodes in it. So, overall there are $n = \Theta(t \log t)$ nodes. We now connect each set SR,i to SL so that each node $v \in$ SR,i has i neighbors in SL and no two vertices in SR,i share any neighbors in common (we can do that since SR,i has at most t/i nodes). Now, the optimal vertex cover is simply the set SL of size t, but this greedy algorithm might first choose SR,t then SR,t−1, and so on down to SR,1, finding a cover of total size n−t. Of course, the fact that the bad cases are complicated means this algorithm might not be so bad in practice.

Proof: What Algorithm 1 finds in the end is a matching (a set of edges no two of which share an endpoint) that is "maximal" (meaning that you can't add any more edges to it and keep it a matching). This means if we take both endpoints of those edges. we must have a vertex cover. In particular, if the algorithm picked k edges, the vertex cover found has size 2k. But, any vertex cover must have size at least k since it needs to have at least one endpoint of each of these edges, and since these edges don't touch, these are k different vertices. So the algorithm is a 2-approximation as desired.

Here is now another 2-approximation algorithm for Vertex Cover:

Algorithm 2: First, solve a fractional version of the problem. Have a variable xi for each vertex with constraint $0 \leq x_i \leq 1$. Think of $x_i = 1$ as picking the vertex, and $x_i = 0$ as not picking it, and in-between as "partially picking it". Then for each edge (i,j), add the constraint that it should be covered in that we

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

require xi + xj ≥ 1. Then our goal is to minimize Pi xi. We can solve this using linear programming. This is called an “LP relaxation” because any true vertex cover is a feasible solution, but we’ve made the problem easier by allowing fractional solutions too. So, the value of the optimal solution now will be at least as good as the smallest vertex cover, maybe even better (i.e., smaller), but it just might not be legal any more. [Give examples of triangle-graph and star-graph] Now that we have a super-optimal fractional solution, we need to somehow convert that into a legal integral solution. We can do that here by just picking each vertex i such that xi ≥ 1/2. This step is called rounding of the linear program (which literally is what we are doing here by rounding the fraction to the nearest integer — for other problems, the “rounding” step might not be so simple).

Theorem 21.2 The above algorithm is a factor 2 approximation to Vertex-Cover.

Proof: Claim 1: the output of the algorithm is a legal vertex cover. Why? [get at least 1 endpt of each edge] Claim 2: The size of the vertex cover found is at most twice the size of the optimal vertex cover. Why? Let OPTfrac be the value of the optimal fractional solution, and let OPTV C be the size of the smallest vertex cover. First, as we noted above, OPTfrac ≤ OPTV C. Second, our solution has cost at most 2·OPTfrac since it’s no worse than doubling and rounding down. So, put together, our solution has cost at most 2·OPTV C. Interesting fact: nobody knows any algorithm with approximation ratio 1.9. Best known is 2 − O(1/√logn), which is 2−o(1). Current best hardness result: Hasted shows 7/6 is NP-hard. Improved to 1.361 by Dinur and Safra. Beating 2-epsilon has been related to some other open problems (it is “unique games hard”), but is not known to be NP-hard.

21.4 Set Cover

The Set-Cover problem is defined as follows:

Definition 21.2 Set-Cover: Given a domain X of n points, and m subsets S1,S2,...,Sm of these points. Goal: find the fewest number of these subsets needed to cover all the points. The decision problem also provides a number k and asks whether it is possible to cover all the points using k or fewer sets.

Set-Cover is NP-Complete. However, there is a simple algorithm that gets an approximation ratio of lnn (i.e., that finds a cover using at most a factor lnn more sets than the optimal solution).

Greedy Algorithm (Set-Cover): Pick the set that covers the most points. Throw out all the points covered. Repeat.

What’s an example where this algorithm doesn’t find the best solution?

Theorem 21.3 If the optimal solution uses k sets, the greedy algorithm finds a solution with at most klnn sets.

Proof: Since the optimal solution uses k sets, there must some set that covers at least a 1/k fraction of the points. The algorithm chooses the set that covers the most points, so it covers at least that many.

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Therefore, after the first iteration of the algorithm, there are at most n(1−1/k) points left. Again, since the optimal solution uses k sets, there must some set that covers at least a 1/ kfraction of the remainder (if we got lucky we might have chosen one of the sets used by the optimal solution and so there are actually k −1 sets covering the remainder, but we can't count on that necessarily happening). So, again, since we choose the set that covers the most points remaining, after the second iteration, there are at most n(1−1/k)2 points left. More generally, after t rounds, there are at most n(1−1/k)t points left. After t = k lnn rounds, there are at most n(1−1/k)k lnn < n(1/e)lnn = 1 points left, which means we must be done.

Notice how the above analysis is similar to the analysis we used of Edmonds-Karp #1. Also, you can get a slightly better bound by using the fact that after kln(n/k) rounds, there are at most n(1/e)ln(n/k) = k points left, and (since each new set covers at least one point) you only need to go k more steps. This gives the somewhat better bound of kln(n/k) + k. So, we have:

Theorem 21.4 If the optimal solution uses k sets, the greedy algorithm finds a solution with at most kln(n/k) + k sets.

In fact, it's been proven that unless everything in NP can be solved in time nO(loglogn), then you can't get better than (1−ϱ)ln(n) for any constant ϱ > 0 [Feige].

21.5 Max-SAT

The Max-SAT problem is defined as follows:

Definition 21.3 Max-SAT: Given a CNF formula (like in SAT), try to maximize the number of clauses satisfied.

To make things cleaner, let's assume we have reduced each clause [so, (xVxVy) would become just (xVy), and (xV¬x) would be removed]

Theorem 21.5 If every clause has size exactly 3 (this is sometimes called the MAX-exactly-3SAT problem), then there is a simple randomized algorithm can satisfy at least a 7/8 fraction of clauses. So, this is for sure at least a 7/8-approximation.

Proof: Just try a random assignment to the variables. Each clause has a 7/8 chance of being satisfied. So if there are m clauses total, the expected number satisfied is (7/8)m. If the assignment satisfies less, just repeat. Since the number of clauses satisfied is bounded (it's an integer between 0 and m), with high probability it won't take too many tries before you do at least as well as the expectation.

How about a deterministic algorithm? Here's a nice way we can do that. First, let's generalize the above statement to talk about general CNF formulas.

Claim 21.6 Suppose we have a CNF formula of m clauses, with m1 clauses of size 1, m2 of size 2, etc. (m = m1 + m2 + ...). Then a random assignment satisfies Pk mk(1 − 1/2k) clauses in expectation.

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Proof: linearity of expectation.

Theorem 21.7 There is an efficient deterministic algorithm that given a CNF formula of m clauses, with m1 clauses of size 1, m2 of size 2, etc. (m = m1 + m2 + ...) will find a solution satisfying at least Pk mk(1−1/2k) clauses.

Proof: Here is the deterministic algorithm. Look at x1: for each of the two possible settings (0 or 1) we can calculate the expected number of clauses satisfied if we were to go with that setting, and then set the rest of the variables randomly. (It is just the number of clauses already satisfied plus Pk mk(1−1/2k), where mk is the number of clauses of size k in the "formula to go".) Fix x1 to the setting that gives us a larger expectation. Now go on to x2 and do the same thing, setting it to the value with the highest expectation-to-go, and then x3 and so on. The point is that since we always pick the setting whose expectation-to-go is larger, this expectation-to-go never decreases (since our current expectation is the average of the ones we get by setting the next variable to 0 or 1).

This is called the "conditional expectation" method. The algorithm itself is completely deterministic — in fact we could rewrite it to get rid of any hint of randomization by just viewing Pk mk(1−1/2k) as a way of weighting the clauses to favor the small ones, but our motivation was based on the randomized method. Interesting fact: getting a 7/8 + ǫ approximation for any constant ǫ > 0 (like .001) for MAXexactly-3-SAT is NP-hard. In general, the area of approximation algorithms and approximation hardness is a major area of algorithms research. Occupies a good fraction of major algorithms conferences.

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Lecture 37 - Traveling Salesman Problem

The Traveling Salesman Problem

Introduction

The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one (e.g. the hometown) and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip.

The traveling salesman problem can be described as follows: TSP = {(G, f, t): G = (V, E) a complete graph, f is a function V×V Z, → t ∈ Z, G is a graph that contains a traveling salesman tour with cost that does not exceed t}.

Example:

Consider the following set of cities:

5

12 2 3 10

8 4 3

A

E

B

D

C

Figure 10.1 A graph with weights on its edges.

The problem lies in finding a minimal path passing from all vertices once. For example the path Path1 {A, B, C, D, E, A} and the path Path2 {A, B, C, E, D, A} pass all the vertices but Path1 has a total length of 24 and Path2 has a total length of 31.

Definition:

A Hamiltonian cycle is a cycle in a graph passing through all the vertices once.

Example:

A

E

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

B

D

C

Figure 10.2 A graph with various Hamiltonian paths.

P = {A, B, C, D, E} is a Hamiltonian cycle. The problem of finding a Hamiltonian cycle in a graph is NP-complete.

Theorem 10.1: The traveling salesman problem is NP-complete.

Proof: First, we have to prove that TSP belongs to NP. If we want to check a tour for credibility, we check that the tour contains each vertex once. Then we sum the total cost of the edges and finally we check if the cost is minimum. This can be completed in polynomial time thus TSP belongs to NP. Secondly we prove that TSP is NP-hard. One way to prove this is to show that Hamiltonian cycle TSP (given that the Hamiltonian cycle problem is NP-complete). Assume G = (V, E) to be an instance of Hamiltonian cycle. An instance of TSP is then constructed. We create the complete graph = (V, P≤ G' E'), where E' = {(i, j):i, j ∈ V and i ≠ j. Thus, the cost function is defined as:

t(i ,,j) = 10.1 ∉ ∈ . j) (i, if1 , j) (i, if0 E E

Now suppose that a Hamiltonian cycle h exists in G. It is clear that the cost of each edge in h is 0 in G as each edge belongs to E. Therefore, h has a cost of 0 in G ' '. Thus, if graph G has a Hamiltonian cycle then graph G has a tour of 0 cost. ' Conversely, we assume that G' has a tour h' of cost at most 0. The cost of edges in E' are 0 and 1 by definition. So each edge must have a cost of 0 as the cost of h' is 0. We conclude that h' contains only edges in E. So we have proven that G has a Hamiltonian cycle if and only if G' has a tour of cost at most 0. Thus TSP is NP-complete.

10.2 Methods to solve the traveling salesman problem 10.2.1 Using the triangle inequality to solve the traveling salesman problem

Definition: If for the set of vertices a, b, c ∈ V, it is true that t (a, c) ≤ t(a, b) + t(b, c) where t is the cost function, we say that t satisfies the triangle inequality.

First, we create a minimum spanning tree the weight of which is a lower bound on the cost of an optimal traveling salesman tour. Using this minimum spanning tree we will create a tour the cost of which is at most 2 times the weight of the spanning tree. We present the algorithm that performs these computations using the MST-Prim algorithm.

Approximation-TSP Input: A complete graph G (V, E) Output: A Hamiltonian cycle

1.select a "root" vertex r ∈ V [G]. 2.use MST-Prim (G, c, r) to compute a minimum spanning tree from r. 3.assume L to be the sequence of vertices visited in a preorder tree walk of T. 4.return the Hamiltonian

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

cycle H that visits the vertices in the order L. The next set of figures show the working of the proposed algorithm. A B E C

(a)

(b) (c)

D

A

B

D

C

A

B

D

E

C

E

Figure 10.3 A set of cities and the resulting connection after the MST-Prim algorithm has been applied..

In Figure 10.3(a) a set of vertices is shown. Part (b) illustrates the result of the MST-Prim thus the minimum spanning tree MST-Prim constructs. The vertices are visited like {A, B, C, D, E, A) by a preorder walk. Part (c) shows the tour, which is returned by the complete algorithm.

Theorem 10.2: Approximation-TSP is a 2-approximation algorithm with polynomial cost for the traveling salesman problem given the triangle inequality.

Proof: Approximation-TSP costs polynomial time as was shown before.

Assume H* to be an optimal tour for a set of vertices. A spanning tree is constructed by deleting edges from a tour. Thus, an optimal tour has more weight than the minimum-spanning tree, which means that the weight of the minimum spanning tree forms a lower bound on the weight of an optimal tour.

$c(t) \leq c(H^*)$. 10.2

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Let a full walk of T be the complete list of vertices when they are visited regardless if they are visited for the first time or not. The full walk is W. In our example: W = A, B, C, B, D, B, E, B, A,. The full walk crosses each edge exactly twice. Thus, we can write:

$c(W) = 2c(T)$. 10.3

From equations 10.2 and 10.3 we can write that

$c(W) \leq 2c(H^*)$, 10.4

Which means that the cost of the full path is at most 2 time worse than the cost of an optimal tour. The full path visits some of the vertices twice which means it is not a tour. We can now use the triangle inequality to erase some visits without increasing the cost. The fact we are going to use is that if a vertex a is deleted from the full path if it lies between two visits to b and c the result suggests going from b to c directly. In our example we are left with the tour: A, B, C, D, E, A. This tour is the same as the one we get by a preorder walk. Considering this preorder walk let H be a cycle deriving from this walk. Each vertex is visited once so it is a Hamiltonian cycle. We have derived H deleting edges from the full walk so we can write: $c(H) \leq c(W)$ 10.5

From 10.4 and 10.5 we can imply:

$c(H) \leq 2\ c(H^*)$. 10.6

This last inequality completes the proof.

10.2.2 The general traveling salesman problem

Definition: If an NP-complete problem can be solved in polynomial time then P = NP, else $P \neq NP$.

Definition: An algorithm for a given problem has an approximation ratio of $\rho(n)$ if the cost of the S solution the algorithm provides is within a factor of $\rho(n)$ of the optimal S* cost (the cost of the optimal solution). We write:

$\max(S/S^*, S^*/S) \leq \rho(n)$. 10.7

If the cost function t does not satisfy the triangle inequality then polynomial time is not enough to find acceptable approximation tours unless P = NP.

Theorem 10.3: If $P \neq NP$ then there is no approximation algorithm with polynomial cost and with approximation ratio of ρ for any $\rho \geq 1$ for the traveling salesman problem.

Proof: Let us suppose that there is an approximation algorithm A with polynomial cost for some number $\rho \geq 1$ with approximation ratio ρ. Let us assume that ρ is an integer without loss of generality. We are going to try to use A to solve Hamiltonian cycle problems. If we can solve such NP-complete problems then P = NP. Let us assume a Hamiltonian-cycle problem G = (V, E). We are going to use algorithm A to determine whether A contains Hamiltonian cycles. Assume G' = (V,E') to be the complete graph on V. Thus: $E' = \{(a, b): a, b \in V \text{ and } a \neq b\}$ 10.8 Each edge in E' is assigned an integer:

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

t(a, b) = 10.9 ∈Velsep E |1| , b) (a, if1

Consider the traveling salesman problem (G', t). Assume there is a Hamiltonian cycle H in the graph G. Each edge of H is assigned a cost of 1 by the cost function t. Hence (, t) has a tour of cost |V|. If we had assumed that there is not a Hamiltonian cycle in the graph G, then a tour in G G' ' must contain edges that do not exist in E. Any tour with edges not in E has a cost of at least

(ρ |V| + 1) + (|V| - 1) = ρ |V| + |V| 10.10 > ρ |V|

Edges that do not exist in G are assigned a large cost the cost of any tour other than a Hamiltonian one is incremented at least by |V|. Let us use the approximation algorithm described in 10.2.1 to solve the traveling salesman problem (G , t). ' A returns a tour of cost no more than ρ times the cost of an optimal tour. Thus, if G has a Hamiltonian cycle then A must return it. If G does not have a Hamiltonian cycle, A returns a tour

whose cost is more than ρ |V|. It is implied that we can use A to solve the Hamiltonian-cycle problem with a polynomial cost. Therefore, the theorem is proved by contradiction. 10.2.3 A heuristic solution proposed by Karp

According to Karp, we can partition the problem to get an approximate solution using the divide and conquer techniques. We form groups of the cities and find optimal tours within these groups. Then we combine the groups to find the optimal tour of the original problem. Karp has given probabilistic analyses of the performance of the algorithms to determine the average error and thus the average performance of the solution compared to the optimal solution. Karp has proposed two dividing schemes. According to the first, the cities are divided in terms of their location and only. According to the second, the cities are divided into cells that have the same size. Karp has provided upper bounds of the worst-case error for the first dividing scheme, which is also called Adaptive Dissection Method. The working of this method is explained below. Let us assume that the n cities are distributed in a rectangle. This rectangle is divided in B = 2k sub rectangles. Each sub rectangle contains at most t cities where k = log2[(N-1)/(t-1)]. The algorithm computes an optimum tour for the cities within a sub-rectangle. These 2k optimal tours are combined to find an optimal tour for the N cities. Let us explain the working of the division algorithm. Assume Y to be a rectangle with num being the number of cities. The rectangle is divided into two rectangles in correspondence of the [num/2] th city from the shorter side of the rectangle. This city is true that belongs to the common boundary of the two rectangles. The rest of the division process is done recursively. The results for this algorithm are presented below. Assume a rectangle X containing N cities and t the maximum allowed number of cities in a subrectangle. Assume W to be the length of the walk the algorithm provides and Lopt to be the length of the optimal path. Then the worst-case error id defined as follows: W-Lopt ≤ 3/2 ∑2ki=1 Per(Yi) 10.11 Where Per (Yi) is the perimeter of the ith rectangle. If a,b are the dimensions of the rectangle we can imply an upper bound for ∑2ki=1 Per(Yi) : ∑2ki=1 Per(Yi) ≤ 3/2 2*2a+b/2 (2⌈k/2⌉ + ⌈ k/2⌉) 10.12

Now we can write: W-Lopt ≤ 3/2 2*2a+b/2 (2⌈k/2⌉ + ⌈ k/2⌉) 10.13 Where 2α = a and 2β = b.

If a = b then we can imply that: W-Lopt ≤ 3/2[2a(2⌈k/2⌉ + ⌈ k/2⌉)] 10.14.

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

There are two possibilities for k: If k even W-Lopt ≤ 3a2k/2 +1 10.15 If k odd W-Lopt ≤ 3a3/ 2 *2k/2 10.16

Assuming that log2(N-1)/(t-1) is an integer we can express this equation in terms of N and t:

If k even W-Lopt ≤ 3a2 1))/((1 −− Nt 10.17 If k odd W-Lopt ≤ 3a 2 3 1))/((1 −− Nt 10.18

Observation: The points distribution does not affect the result. It should be noted however that these results only hold for uniform distributions. We now assume random distributions to generalize the results. Let us assume a rectangle X of area v(X), within which there are randomly distributed N cities, following a uniform distribution. Let us denote the length of an optimal tour through the N cities to a random variable TN(X).Thus there exists a positive constant β such as that ∀ε>0

Prob limNÆ∞(TN(X)/ () NvX β >ε = 0 10.19

This result from equation 10.12 shows that the relative error between a spanning walk and an optimal tour can be estimated as: ∀ε>0 Prob limNÆ∞(W- TN(X)/ TN(X) – S/ t)>ε =0 10.20 Where S>0.

Let us assume a rectangle X[a, b] with ab = 1. Let Tt(X) be an optimal tour through t<N cities in the rectangle. We are going to compare the average length of this tour to the average length of an optimal tour through all the N cities.

βx(t) is defined as: βx(t) = E(Tt(X))/ t .

From equation 10.20 we get that limt ∞ x(t) = β . Æ Thus, there exists the following bound: β

βx(t) - β ≤ 6(a+b)/ t . We can say that the length of a tour through t<N cities tends to be almost the same as the length of the optimal tour.

10.2.4 Trying to solve the traveling salesman problem using greedy algorithms

Assume the asymmetric traveling salesman problem. We use the symbol of (Kn,c) whre c is the weight function and n is the number of vertices. We assume the symmetric traveling salesman problem to be defined in the same way but Kn symbols a complete undirected graph. If we try to find an approximate solution to an NP-hard problem using heuristics, we need to compare the solutions using computational experiments. There is a number called domination number that compares the performance of heuristics. A heuristic with higher domination number is a better choice than a heuristic with a lower domination number.

Definition: The domination number for the TSP of a heuristic A is an integer such as that for each instance I of the TSP on n vertices A produces a tour T that is now worse than at least d(n) tours in I including T. If we evaluate the greedy algorithm and the nearest neighbor algorithm for the TSP, we find that they give good results for the Euclidean TSP but they both give poor results for the asymmetric and the symmetric TSP. We analyze below the performance of the greedy algorithm and the nearest neighbor algorithm using the domination number.

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Theorem 10.4: The domination number of the greedy algorithm for the TSP is 1.

Proof: We assume an instance of the ATSP for which the greedy algorithm provides the worst tour. Let nmin{i, j} be the cost of each arc(i, j). We assume the following exceptions: c(i, i+1) = in, for i = 1,2,..,n-1, c(i, 1) = n2 –1for i = 3,4,…,n-1 and c(n,1) = n3 . We observe that the cheapest arc is (1,2). Thus the greedy algorithm returns the tour (1,2,…,n,1). We can compute the cost of T as: ∑ −

=

1

1

n

i

in+c(n,1). 10.21

Assume a tour H in the graph such as that c(H) ≥ c(T). The arc (n,1) must be contained within the tour h as

c(n,1) > n max{c(i,j) : 1≤i≠j ≤n,(i,j)≠(n,1)}. 10.22 It is implied that there is a Hamiltonian path P from 1 to n with a cost of ∑ − = 1 1 n i in Let ei be an arc of P with a tail i. It is true that c(ei) ≤ in+1. P mut have an arc (ek) that goes to an edge with an identification number smaller than the number of the edge it starts from. We can now write:

c(ek) ≤ (k-1)n + 1 and 10.23 c(P) ≤ +(n-1)-n. 10.24 ∑ − = 1 1 n i in

The theorem is proven by contradiction.

Theorem 10.5: Let n ≥ 4. The domination number of the nearest neighbor algorithm for the asymmetric traveling salesman problem is at most n-1 and at least n/2.

Proof: Assume all arcs such as that (i,i+1) 1≤ i <n, have a cost of iN, all arcs such as that (i,i+2) 1≤ i ≤ n2 have a cost of iN+1, all the other arcs that start from an edge with an identification number smaller than that of the edge hey end(forward arcs) have a cost of iN+2 and all the other arcs that start from an edge with a greater identification number than that of the edge they end (backward arcs) have a cost of (j-1)N. If nearest neighbor starts at i, which is neither 1 nor n, it has a cost of

l = ∑ -N+1. 10.25 kN n k − = 1 1 If the algorithm starts at 1 we have a cost of > l kN n k ∑ − = 1 1 Any tour has a cost of at least l. Let us define the length of a backward arc as i-j. Let F be the set of tours described above and T1 a tour not in F. T1 is a tour, so the cost of T1 is at most 2 1 1 +∑ − = iN n k -qN -|B|N , 10.26 where B is the set of backward arcs and q is the sum of length of the arcs in B. We conclude that the cost of T1 is less than l, which would mean that T1 belongs to F. So all cycles that do not belong to F have a cost less than those who belong to F. We assume that nearest neighbor does not have a domination number of at least n/2. Nearest neighbor constructs n tours. By assumption the number of

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

cities is at least 4, so we have at least 3 tours that may coincide. Let F = x1x2xnx1 be a tour such as that F = Fi=Fj= Fk. We could assume that i=1 and 2<j≤1+(n/2). Foe every m with j<m≤n let Cm be the tour provided by deleting consecutive arcs and adding backward arcs. We should note that c(Cm)≥ c(Cf) since c(xi,xi+1)≤ c(xj,xj+1). This is true as we used nearest neighbor from xj to construct Fj. and c(xm,xm+1)≤ c(xm,xi+1) since nearest neighbor chose the (xm,xm+1) on Fj when the arc (xm,xi+1) was available. We can state now that the domination number is at least n-j+1≤ n/2. The theorem is proven by contradiction.

Definition: A tour x1 x2 x3 xn x1 , x1 = 1 in a symmetric traveling salesman problem is called pyramidal if x1<x2<xn<x1≤xk+1 >…>xn. The number of pyramidal tours in a symmetric traveling salesman problem is: 2n-3.

Theorem 10.6: Let n ≥ 4. The domination number of nearest neighbor for the symmetrical traveling salesman problem is at most 2n-3.

Proof: We consider an instance of symmetric traveling salesman problem, which proves that nearest neighbor has a domination number of 2n-3. Let all edges {ι, ι+1}, 1≥ i < n have cost of iN. Let all edges {i,i+2} have a cost of iN+1 Let all the remaining edges {i,j}, i<j , cost iN+2. Let us assume that CNN is the cost of the cheapest tour provided by the nearest neighbor algorithm. It is then clear that

CNN = c(12…n1) = +N+2. 10.27 ∑ − = 1 1 n i iN

Assume a tour on the graph. Let it be x1 x2 xn x1 . We assume a directed cycle T’ which is constructed by orienting all the edges on the tour. For a backward arc in the cycle e(j,i), we define its length as q(e) = j-i. We express the sum of the lengths of the backward arcs in the cycle as q(T’). Assume the most expensive non-pyramidal tour T. Let Cmax- be the cost of this tour. We have to show that

Cmax < C nn, 10.28 as there are 2n-3 pyramidal tours. It is true that q(T’) ≥ n for every T’. Assume a non pyramidal tour H of cost Cmax and ei = (i,j) be an arc of H’. If e1 is forward then c (e1) ≤ iN +2. If e1 is backward then c(e1) ≤ jN +2 – q(ei)N. Thus we can write: Cmax ≤∑ - q(H’)N ≤ +2n 10.29 = + n i iN 1 2)(∑ − = 1 1 n i iN

As q(H’) ≥ n. From the preceding equations, we conclude that indeed

Cmax < C nn

Thus, the theorem is proven.

10.2.5 The branch and bound algorithm and its complexity

The branch and bound algorithm converts the asymmetric traveling salesman problem into an assignment problem. Consider a graph V that contains all the cities. Consider Π being the set of all the permutations of the cities, thus covering all possible solutions. Consider a permutation of this set π∈Π in which each city is assigned a successor, say i,for the π i city. So a tour might be (1, π (1), π (π (1)),…,1). If the number of the cities in the tour is n then the permutation is called a cyclic permutation. The

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

assignment problem tries to find such cyclic permutations and the asymmetric traveling salesman problem seeks such permutations but with the constraint that they have a minimal cost. The branch and bound algorithm firstly seeks a solution of the assignment problem. The cost to find a solution to the assignment problem for n cities is quite large and is asymptotically O(n3). If this is a complete tour, then the algorithm has found the solution to the asymmetric traveling salesman problem too. If not, then the problem is divided in several sub-problems. Each of these sub-problems excludes some arcs of a sub-tour, thus excluding the sub-tour itself. The way the algorithm chooses which arc to delete is called branching rules. It is very important that there are no duplicate sub-problems generated so that the total number of the sub-problems is minimized. Carpaneto and Toth have proposed a rule that guarantees that the sub-problems are independent. They consider the included arc set and select a minimum number of arcs that do not belong to that set. They divide the problem as follows. Symbolize as E the excluded arc set and as I the included arc set. The I is to be decomposed. Let t arcs of the selected sub-tour x1x2 ...xn not to belong to I. The problem is divided into t children so that the jth child has Ej excluded arc set and Ij included arc set. We can now write:

□□□ □ Ⅶ∪ x}x,...,{x, I -Ii {x} E =Ej -12j1 j k = 1,2,..,j 10.30

But xj is an excluded arc of the jth sub-problem and an included arc in the (j+1)st problem. This means that a tour produced by the (j+1)st problem may have the xj arc but a tour produced by the jth problem may not contain the arc. This means that the two problems cannot generate the same tours, as they cannot contain the same arcs. This guarantees that there are no duplicate tours.

Complexity of the branch and bound algorithm.

There has been a lot of controversy concerning the complexity of the branch and bound algorithm. Bellmore and Malone have stated that the algorithm runs in polynomial time. They have treated the problem as a statistical experiment assuming that the ith try of the algorithm is successful if it finds a minimal tour for the Ith sub-problem. They assumed that the probability of the assignment problem to find the solution to the asymmetric traveling salesman problem is e/n where n is the number of the cities. Under other assumptions, they concluded that the total number of sub-problems is expected to be:

∑∞ =1i ipi ∏ − = − 1 0

1

i

j

pj≤ (1-p ∑ ∞ =1i ipo o)i-1 = 1\po = O(n). 10.31

Smith concluded that under some more assumptions the complexity of the algorithm is O(n3ln(n))

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

The assumptions made to reach this result are too optimistic. Below it will be proven that they do not hold and that the complexity of the branch and bound algorithm is not polynomial.

Definition: The assignment problem of a cost matrix with ci,j = ∞ is called a modified assignment problem. Lemma 10.1: Assume a n x n random cost matrix. Consider the assignment problem that has s<n excluded arcs and t included arcs. Let q(n,s,t) be the probability that the solution of the assignment problem is a tour. Then q(n,s,t) is asymptotically

e/n – o(1/n) < q(n,s,t) + o(1/n) if t=0, 10.32

q(n,s,t) - λ /(n-t) + o(1/n) if t>0, 10.33 where λ , < λ <e is a constant.

Proof: See [7] Lemma 10.2: Consider branch and bound select two nodes that are independent. Assume that the probability that a non-root node in the search tree is a leaf is p. Let po be the possibility that the node is the root. There exists a constant 0< δ <l-1/e for a non-root node so that if t< δ n then p<po, where n is the number of cities.

Proof: Assume the search tree and a node of it say Y, which is constructed from 10.33.Y, has some included and some excluded arcs. Suppose the number of included arcs is t and the number of excluded arcs is s, Observe that s is the depth of the node in the search tree. Let the path from the root to the node is is Y0,Y1,Y2,…,Y. We can say that Yi has I excluded arcs. The probability of Y creating a complete tour is that none of its ancestors provides a solution to the assignment problem thus they do not provide a complete tour—but Y does. The probability that Y's parent does not provide a complete tour is (1-q(n,s-1,ts-1)). Consequently the probability that p and Y exists and is a leaf taking into account the independence assumption is:

p = q(n.s.t) = ∏ . 10.34 − = − 1 0 ,))(,(1 s i itiqn

Using lemma 10.1, we find that:

p = (λ /(n-t) + o(1/n)) ∏ = − = − − 1 0 1 s i nti λ

nt − λ

(1-∑ + o(1/n), 10.35 − = − 1 0 s i nti i λ

where λ o > λ 1 > λ 2 > λ s-1 > λ >1 are constants. We can now show that ∑ −

= −

1

0

s i nti i λ

=

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

'

' nt s − λ

10.36

where λ’ = ½ ∑ − = 1 0 s i i λ

and t’ = ∑ ∑ −

= −

=

1 0

1 0

s i

s i

i iti λ λ

It is true that 0 < t’ < t. Now we can write the probability as:

p =

nt − λ

(1

'

' nt s − λ

) + o(1/n) 10.37

Now we assume that the lemma does not hold thus p≥po where po = e/n. Let δ = (eλ)/e. we know that 1 < λ < e and 0 < δ < 1-1/e. Now it can be shown that: t’ > n + nete sn −− () ' λ λλ >n 10.38

but n≥t so t’>t which is a contradiction. Thus, the lemma is proven. Lemma 10.3: Assume a n x n random matrix. Assume a solution in a modified assignment problem. The expected number of sub-tours is asymptotically less than ln(n) and the expected number of arcs in a sub-tour is greater than n/ln(n).

Proof: See [7]

The number of children constructed when we choose a sub-tour with the minimum number of arcs is O(n/ln(n)), as is proven in lemma 10.1. The nodes at the first depth have t = O(n/ln(n)) included arcs. But

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

as it it shown above $t < \delta n$. This means that all nodes on the first depth asymptotically follow the inequality: $t < \delta n$. Equally all nodes in the ith depth follow the same inequality except the ones that I is greater than O(ln(n)). Assume a node with no included arcs. Its ancestors do not have included arcs either. Using lemma 10.1 we can state that the probability that the node or one of its ancestors being a solution to the assignment problem is e/n. We can now generalize this and say that the probability that a node with no included arcs exists at dth depth and is a leaf node is: $p = e/n(1-e/n)d$. 10.39

Observe that this probability is less than e/n or the probability that the root node is a leaf.. It is therefore true that if we consider nodes whose depth is no greater than O(ln(n)) the probability that they are leaf nodes is less than the probability of the root being a leaf itself. The probability that a sub-problem chosen by branch and bound will be solved by the assignment problem and will produce an optimal tour is less than the probability that the search will become a leaf node. Consider the node that generates the optimal tour. If its depth is greater than ln(n) then we have to expand ln(n) nodes each one of which has a probability of producing the optimal tour less than po. If the depth of the node is lesser than ln(n) and if we need to expand only a polynomial number of nodes according to Bellmore and Malone, then the expanded nodes have a probability less than po of creating the optimal tour. This statement contradicts the polynomial assumption. Therefore we can state that the branch and bound algorithm expands more than ln(n) nodes to find the optimal tour. This means that the algorithm cannot finish in polynomial time. 10.2.6 The k-best traveling salesman problem, its complexity and one solution using the branch and bound algorithm Consider a graph $G = (V, E)$. The number of the n cities in the graph has to be at least 3. Consider an edge $e \in E$. The length of this edge is described by l(e). Consider a vector that contains the lengths of the edges of the initial graph. Let us call this vector l. We can now create a weighted graph, which consists of the pairs (G, d). Consider a set S of edges. The length of this set is described as Ll(S). Consider the set H of all the Hamiltonian tours in the G. We assume that G has at least on Hamiltonian tour. Definition: Let $1 \leq k \leq |H|$. Any set H(k) satisfying $Ll(H1) \leq Ll(H2) \leq \ldots \leq Ll(Hk) \leq Ll(H)$ for all H is called a set of k-best tours.

In other words the k-best tour problem is the problem of finding a set of k tours such that the length of each tour is at least equal to the length of the greater tour in the set.

Complexity of the k-best traveling salesman problem

Theorem 10.7: The k-best TSP is NP-hard for $k \geq 1$ regardless if k is fixed or variable.

Proof: Consider the case that k is variable. This means that k is included in the input. We have to solve TSP itself to find a solution to the k-best TSP. Since the TSP which is NP-hard is part of the kbest TSP then the k-best TSP is NP-hard too. Consider the case that k is fixed. This means that k is not included in the input. It is clear that a shortest path can be determined if we know k-best tours. So we can conclude that k-best TSP is NP-hard itself too.

Solutions of the k-best TSP

Solutions provided by partitioning algorithms

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

Definition: For I, O ∈ E, the set {H:I ⊆ H ⊆ E\O} is called a restricted set of tours. To solve the problem using partitioning algorithms we use the following idea. We partition the tours into sets such as that each set is a restricted one. We apply algorithms for solving the traveling salesman problem for each one of these restricted sets. We combine the optimal solutions for the sets to find the k-best solution. There are two known partitioning algorithms for the k-best TSP. The one of them is the Lawler algorithm and the other one is the Hamacher and Queyranne algorithm. The difference in these two algorithms lies in the way they partition the solutions. They both call an algorithm to solve the problem for a restricted set. Their complexity cannot be easily determined since the k-best TSP is NP-hard. A clue to figure out which algorithm is the best of the two would be to check how many times they call the algorithm to find a solution for the restricted set.

Using the branch and bound method to derive solutions for the k-best traveling salesman problem

Since the branch and bound method is used for solving the classic traveling salesman problem (although in greater time than polynomial) it is worthy to modify it to solve the k-best TSP. Initially the branch and bound tree contains only one node, the root node. As the algorithm proceeds each node of the tree expands taking into computation edges from the graph. At a given moment the branch and bound tree contains information about what are the best tours so far. As an analogue to the original branch and bound, which contains information, what is the best candidate for the optimal path. When the algorithm ends, the initially empty tree has information about the set of k-best tours. We considered a restricted set of tours as is defined above. Let us assume a node in the branch and bound tree with this restricted set. First of all the algorithm determines a lower bound which we will express as LB(I, O) : Ll(H) ≥ LB(I, O) for every tour H. It is true that if LB(I, O) ≥ U then we should not take into account any tour that exists in the tree. The algorithm continues until the above holds for all the tours ; that is to say that we cannot take into account any tour in the tree. At that moment we should say that the tree is equal to H(k). If LB(I, O) < U then we have to distinguish two cases. If the graph contains k tours, then the longest of them is removed. The tour from the tree is removed from the tree and added to the

graph. At that point, the information about the longest tour is updated. If the graph contains less than k tours, then we do not have to remove any tour. The longest tour from the tree is added to the graph and the information about the longest tour is updated. Below is the formal expression of the algorithm. It uses a recursive procedure named EXPLORE that runs through all the nodes in the branch and bound tree and performs the computations we have explained. It searches the tree at a depth-first way. It has three parameters I, O and the graph. The I is the partial tour. The algorithm starts by taking the empty sets I ,O and the graph and calling the procedure EXPLORE for these sets.

Modified Branch and bound for finding k-best tours for the traveling salesman problem Input: (G, d), the set of tours H and an integer so that 1 ≤ k ≤ |H| Output: A set H(k) of best tours in the (G, d).

Procedure EXPLORE (I, O, G) 1 Begin 2 Find (LB(I, O)) for (G, d) 3 If (LB(I, O) <U)) 4 Then if |I| = n-1 5 Then begin 6 H is the completion of the partial tour I 7 If(|G| = k) 8 Then 9 Remove one of the longest tours in G 10 G = G ∪ {H} 11 If (|G| = k) 12 Then 13 U is the length of a longest tour in G 14 End 15 Else begin 16 Find a

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

branching edge E 17 EXPLORE (I ∪ {e},O,G) 18 Find LB(I, O ∪ {e} taking into account (G, d) 19 If (LB(I, O ∪ {e}) <U) 20 Then 21 EXPLORE (I, O ∪ {e}, G) 22 End 23 Begin 24 U <= ∞ 25 H(k) <= 0 26 EXPLORE (0, 0, H(k)) 27 End

It has been shown by experiments that the complexity of the branch and bound algorithm increases dramatically as k gets larger. It is a result that we should expect as the branch and bound algorithm gets much slower as the number of cities increase in the classic traveling salesman problem.

10.3 Geometric solutions 10.3.1 A heuristic solution based on continuous approximation

We assume that the cities are distributed over a region that has the shape of a circular sector. The starting point is on the sector vertex. It is not obligatory that this point is actually the point at which the traveling salesman starts the tour, but we need to ensure that the tour visits the vector. We also assume that the distribution of the cities is uniform to be able to state that the probability of finding a city does not depend on the location of the city. Let us assume that we have N points C of which are cities the salesman visits and N = C + 1 are the cities plus the starting city. The way the tour is constructed is explained below. We divide the circular vector into three parts. There are two inner circular sectors that share the vertex as the border and the remaining ring sector. Figure 10.4 shows the division of the circular sector.

B R

A R’ C

u

Figure 10.4 The circular sector after it has been divided in three regions.

We name the inner sectors A and C and the ring sector B. The division can be described completely by R’. We can write now: p = R R' 10.40

We visit each one of the regions and construct paths in these regions. In the final step, we combine the paths. Figures 10.5 and 10.6 show the working of the algorithm.

Figure 10.5 The cities in the three sections have been connected

Figure 10.6 All the cities have been connected

From the preceding figures, one can see only the tour and not the starting point. This depends entirely on the partition. The average length of the tour is estimated below. The distance between two points of polar coordinates (r, u) is:

d(P1,P2) = min(r1,r2)abs(u1-u2)+abs(r1-r2) = du + dr 10.41

We assume the length of the radial links and the length of the ring links. We express the total tour length as the sum of all the path lengths either radial or ring on all sections. We can now write:

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

l = 2(lA,r + lA,u) + lB,r + lB,,u 10.42

The two multiplier is there because section A and C are the same. We approximate the length of the radial links as lA,r = p. We can also approximate the length of the ring links in A or C by the number of the points in these two regions times the average length of the ring between two points. So we can write:

lA,u = nAdA,u 10.43

By assumption the point are distributed uniformly in all the regions so we can state that: nA = Cp2/2 10.44 The average ring length for all the points can be approximated by:

dA,u = dr rup ∫ 0 6

= pu/12 10.45

So the length of the ring in A or C becomes: lA,u = p3uC/24 10.46

Taking into consideration that the radial distribution has a probability density of f(r) = 2r we conclude that the average distance in sector B is:

pavg = = ∫ 1 () p rdrrf

1)3(1)2(p ppp + ++

10.47

let us make an assumption that we have a uniform radial distribution so as to simplify the above expression. Now we can write:

p =

2 1 p +

10.48

Now we can compute the length of the ring links in B. We find that

LB,u =

2 1)(pu+

10.49

So the expected radial distance between two points found in B can be expressed as:

dB,r = 2 = ∫∫ − 1 ()()() p r p sdsdrrfsfr

1)1)(15(1))(3(14 pp pppp ++ ++−

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

10.50

Once again, we assume a uniform radial distribution and can write that:

dB,r =

3 1 p −

10.51 We compute the length of the radial links in B as the number of points times the expected distance between two points. Thus we can write: lB,r = nB,dB,r = C(1- p2) (1-p)/3 10.52

From the above expressions, we can find the average tour length:

l = 2p +

12 pppuC

+

2 1)(pu+

+

3 C

(1-p2)(1-p). 10.53

The above expression has a drawback. The results it produces are pessimistic if p is close to 1. This is the case when the circular sector is divided into two identical sections, thus having an angle of u/2. Now there is no ring B but there still exists a ring that connects the outermost paths of A and C. This is the cause that makes the estimation pessimistic. We can instead substitute the expression that gives the length of the radial links in B by the more accurate: lB,u = 2 1)(pu+ (1-p/2) 10.54

We can take more precise estimations for the total length of the tour by this expression:

l = 2p +

12 pppuC

+

2 1)(pu+

(1-p/2) +

3 C

(1-p2)(1-p). 10.55

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

This expression is immune to very low values of p (approaching 1) and gives a very accurate estimation of the total tour length. 10.3.2 Held – Karp lower bound

Definition: A 1-tree problem on n cities is the problem of finding a tree that connects n cities with the first city connecting to two cities.

When we try to find a lower bound for the 1-tree problem we try to find a minimum 1-tree. We apply the 1-tree problem to the traveling salesman problem by considering that a tour is a tree whose each vertex has a degree of two. Then a minimum 1-tree is also a minimal path. Figure 10.7 shows a simple 1-tree.

1

Figure 10.7 A simple 1-tree.

Let us consider the geometric traveling salesman problem. We denote to eij the length of the path from the i city to the j city. We assume that each city has a weight of πi. So we can say that each edge has a cost of

cij = eij + πi πj 10.56

We now compute a new minimum 1-tree taking into account the edge costs. It is clear that the new 1-tree we will construct id different from the original 1-tree. Let us consider a set of different tours V. Let U be the set of 1-trees constructed by each tour from V. Recall that a tour is a 1-tree with each vertex having a degree of 2. This means that the set of tours is included in the set of 1trees. Let us express a tour with T and the cost of a tour as L(cij , T) if we take in account the cost of the edges. Therefore, it is true that

min LT ∈

U(cij, T) ≤ min LT ∈

V(cij, T) 10.57

From equation 3.17 we can write that: L(cij , T) = L(eij , T) + ∑ π = n i 1

idTi 10.58

With dTi we symbol the degree of i vertex in the 1-tree. Consider T to be a tour. This means that dTi = 2. So we can now write that:

L(cij , T) = L(eij , T) + π ∑ = n i 1

i 2 10.59

We assume a minimal tour T'. Equation 3.18 is then transformed as:

min LT ∈

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)

U(cij, T) ≤ min L(cij, T') - π ∑ = n i 1

i 2 10.60

Let us express the length of the optimal tour as c' = L(eij , T').Then from equation 10.59 and 10.60 we can get:

minT∈U{c + π ∑ = n i 1

idTi } ≤ c' + ∑ π = n i 1

i 2 10.61

This is transformed into:

minT∈U{c + π ∑ = n i 1

i (dTi -2)} ≤ c' 10.62

We can finally write that:

w(π) = minT∈U{c + π ∑ = n i 1

i (dTi -2)} 10.63

Hence the lower bound for Held-Karp is

Held-Karplower-bound = max (w(π)). 10.64

It has been shown that Held Karp is a very good estimate for the minimum tour length although it does not give the exact result.

This document is available free of charge on studocu

Downloaded by Shushant Kumar Awasthi (sushantawasthi000@gmail.com)