Scheduling and Conflict Resolution of Television Commercials

Avi Prasanna

Department of Mathematics

Indian Institute of Science Education and Research, Pune

A thesis submitted towards partial fulfillment of

B.S. - M.S. dual degree programme

April, 2011

SUPERVISOR

Prof. Jaikumar Radhakrishnan

Tata Institute of Fundamental Research, Mumbai

I would like to dedicate this thesis to my loving parents...

Certificate

This is to certify that this dissertation entitled " Scheduling and Conflict Resolution of Television Commercials " towards the partial fulfillment of the 5 year integrated BS-MS programme at the Indian Institute of Science Education and Research Pune, represents original research carried out by Avi Prasanna at School of Technology and Computer Science, Tata Institute of Fundamental Research Mumbai, under the supervision of Prof. Jaikumar Radhakrishnan during the academic year 2010-2011.

Name and Signature of the Candidate

Supervisor: Prof. Jaikumar Radhakrishnan
(Professor and Dean, STCS, TIFR Mumbai)

Date: 8-April-2011
Place: TIFR, Mumbai

Head:Mathematical Science

Date: 8-April-2011
Place: IISER, Pune

Acknowledgements

First of all I would like to thank my guide Prof. Jaikumar Radhakrishnan for helping me through out the project. I must say that this insight and ideas were beautiful and very interesting. I had a wonderful time working under his supervision. Then I would like to thank my Local mentor, Dr. Soumen Maity, who provided me all the possible help and assistance. I would like to thank Dr. Kavitha Tellikapalli for allowing me to attend the course "Selected Topics in Algorithms" at TIFR, which was of great help to me. I would also like to thank all my friends from TIFR Mumbai and IISER Pune, without them this project would not have been be possible.

Finally, the most important of all I would like to thank God for his blessings and everything...

Abstract

During this project, I studied the basics of Algorithms and ideas in a much more concrete way. By attending a course on "Selected Topics in Algorithm", I got a chance to study the subject in depth, which helped me to get the insight into the subject. I learnt basic algorithms like max-cut problems, balanced max - cut, max k-cut, 2-SAT, 3-SAT etc. Then I studied stuffs related to Steiner trees. Finally in the end I studied an algorithm written by Gaur et al. for the Scheduling and Conflict Resolution of Television commercials. We tried to solve this problem in a different and simple way by following and using a different algorithm that gives us the same performance ratio, in other words we solved this problem by using a simple algorithm which gave us the same optimal result as in the work done by Gaur et al.

Contents

List of Figures

Chapter 1

Introduction

Algorithms and Graph Theory are two very versatile and dynamic fields of study. These techniques are widely used these days and have numerous applications. Graphs can be used to represent almost any kind of network problem. Graph theory is used to understand and simplify the real life situation and help us design a method for solving it. Graph theory is used on Physics, Chemistry and biology to study the interaction of various different things, their interaction, their force of attraction or the force of repulsion etc. Apart from all these things it is also in sociology, a practical application of this is in the construction of social network analysis software. Algorithm also has lots and lots of applications. It is used to describe problems of all kinds. Start from a simple task of *Making Tea* to the complicated and big tasks of handling *Robots* and controlling the *Satellite* etc. Everywhere, algorithms are used to solve our problems.

In this project, I tried to study various topics starting from Algorithms and Graph theory. First of all I learnt the basics of algorithms, the different data structures, studied about the running time of an algorithm and how do we analyze the running time. Then we studied briefly about some general time bounds for e.g.- Linear, Quadratic, polynomial etc. In Graph theory, the topics studied were representations of graphs, and the two very important search tools, the Breadth-first search and the Depth-first search. These two search algorithms are at the heart of graph theory. Then we studied two very useful, fundamental and versatile techniques of solving Algorithms. They are the Greedy algorithms and the Divide

and Conquer rule. We studied these two techniques with the help of an example which illustrated the usefullness of these two techniques.

Then we studied about Maximum Cut and tried to study a recent topic regarding the conflict resolution in the scheduling of Television Commercials. We studied the work done by Daya Ram Gaur, Ramesh Krishnamurti and Rajeev Kohli. They designed an algorithm to resolve the conflict. We then tried to find a simpler algorithm which does the job in quadratic time with the same performance ratio.

Chapter 2

Basics of Algorithm

An algorithm is a step wise well-defined computational procedure that takes a set of value(s), as input and produces a set of value(s), as output. An algorithm is thus a sequence of computational steps that transform the input into the output. An algorithm can be defined as *"A finite set of well-defined instructions for accomplishing some task which, given an initial state, will terminate in a defined end-state."* Algorithms are a very effective methods to solve a problem. It requires time and memory space. In this chapter we will discuss and analyze the important characteristics of an algorithm.

2.1 Efficient Algorithms

Many different algorithms can be designed to solve a particular problem. Let us take an example of sorting a set of say 10 numbers, we can design different algorithms for solving this problem. For obvious reasons we would like to use the one which does the sorting in less time and which uses less memory and space to do it, in other words the one which uses less resource. Hence, our goal is to find an efficient algorithm. Let us now try to define the notion of an efficient algorithm.
A few proposed definitions of an efficient algorithm could be the following

> Definition of Efficiency (1): *An algorithm is said to be efficient if, when implemented, it runs quickly on real input instances.*[6]

Definition of Efficiency (2): *An algorithm is efficient if it achieves qualitatively better worst-case performance, at an analytical level, than brute-force search.*[6]

Definition of Efficiency (3): *An algorithm is efficient if it has a polynomial running time.*[6]

2.2 Asymptotic bounds

An important characteristic of an efficient algorithm is the time taken to solve the problem. Let us now study briefly the notation of asymptotic bounds for calculating and studying the running times of an algorithm.

O - notation

This notation is used to denote ***asymptotic upper bound***. For a function $\psi(n)$, $O(\psi(n))$ denotes the set of functions which satisfy the following criteria,

$$O(\psi(n)) = \{\phi(n) : \text{there exist positive constants } k \text{ and } n_0 \\ \text{such that } 0 \leq \phi(n) \leq k\psi(n) \text{ for all } n \geq n_0\}.$$

The figure 2.1 illustrates the intuition behind this O - notation

Ω - notation

This notation is used for ***asymptotic lower bound***. For a function $\psi(n)$, $O(\psi(n))$ denotes the set of functions which satisfy the following criteria,

$$\Omega(\psi(n)) = \{\phi(n) : \text{there exist positive constants } k \text{ and } n_0 \\ \text{such that } 0 \leq k\psi(n) \leq \phi(n) \text{ for all } n \geq n_0\}.$$

The figure 2.2 illustrates this notation, and the intuition behind Ω - notation

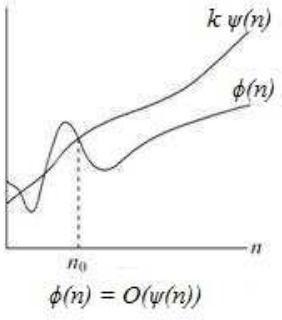

Figure 2.1: O - notation

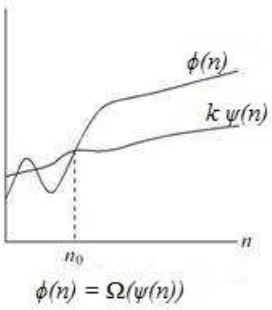

Figure 2.2: Ω - notation

Θ - notation

Let us define what this notation means. For a given function $\psi(n)$, we denote by $\Theta(\psi(n))$ the set of functions,

$$\Theta(\psi(n)) = \{\phi(n) : \text{there exist positive constants } k_1 \text{ , } k_2 \text{ and } n_0 \text{ such that } 0 \leq k_1\psi(n) \leq \phi(n) \leq k_2\psi(n) \text{ for all } n \geq n_0\}.$$

The figure 2.3 illustrates this notation, and the intuition behind this notation

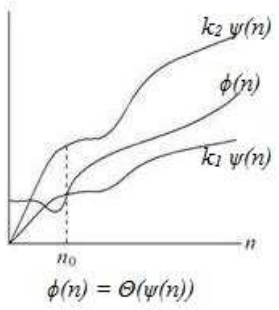

Figure 2.3: Θ - notation

2.3 Properties of Asymptotic bounds

Let us now just have a look at the properties of these asymptotic functions

Transitivity[6]

- If $\phi = O(\psi)$ and $\psi = O(\sigma)$, then $\phi = O(\sigma)$.
- If $\phi = \Omega(\psi)$ and $\psi = \Omega(\sigma)$, then $\phi = \Omega(\sigma)$.
- If $\phi = \Theta(\psi)$ and $\psi = \Theta(\sigma)$, then $\phi = \Theta(\sigma)$.

Sums of Functions[6]

- Suppose that ϕ and ψ are two functions such that for some other function σ, we have $\phi = O(\sigma)$ and $\psi = O(\sigma)$. Then $\phi + \psi = O(\sigma)$.
- Let k be a fixed constant, and let $\phi_1, \phi_2, \cdots, \phi_k$ and σ be functions such that $\phi_i = O(\sigma)$ for all i. Then $\phi_l + \phi_2 + \cdots + \phi_k = O(\sigma)$.
- Suppose that ϕ and ψ are two functions (taking nonnegative values) such that $\psi = O(\phi)$. Then $\phi + \psi = \Theta(\phi)$. In other words, ϕ is an asymptotically tight bound for the combined function $\phi + \psi$.

2.4 Some General Functions and their Bounds

2.4.1 Polynomial

A function which can be written in the form $g(n) = c_0 + c_1 n + c_2 n^2 + \cdots + c_k n^k$ is known as a polynomial function, for some integer constant $k > 0$, where the coefficient of the highest power, c_k is nonzero. This value k is called the degree of the polynomial. For example, the functions of the form $an^2 + bn + c$ (with $a \neq 0$) is a polynomial of degree 2.

Proposition 2.1: *Let ϕ be a polynomial of degree k, in which the coefficient c_k is positive. Then $\phi = \mathrm{O}(n^k)$.*

Proof. Suppose $\phi(n) = c_0 + c_1 n + c_2 n^2 + \cdots + c_k n^k$, where $c_k n^k > 0$. For all coefficients c_j, we have $c_j n^j \leq |c_j| n^k$ for all $n \leq 1$. Thus each term in the above polynomial is $O(n^d)$. Since ϕ is a sum of a constant number of functions, each of which is $O(n^k)$, hence from the above property ϕ is $O(n^k)$ ■

2.4.2 Logarithmic

If $\log_b a$ is equal to a number say, c, then from the definition of logarithm $b^c = a$. Logarithmic functions are nothing but inverse of exponential functions. An approximate sense of how $\log_b a$ grows is to note that, if we round it down to the

nearest integer, it is one less than the number of digits in the base-b representation of the number a.

Proposition 2.2: *For every $b > 1$ and every $c > 0$, we have $log_b\ a = O(a^c)$* ■

2.4.3 Exponential

Functions of the form $f(n) = k^n$ for some constant base k are known as exponential functions. Throughout our discussion we will consider the case in which $k > 1$. If we take $k = 1$, then $f(n)$ equals 1 for any value of n. When $k > 1$, the function $f(n)$ the growth of the function is very fast.

Proposition 2.3: *For every $k > 1$ and every $d > 0$, we have $n^d = 0(k^n)$* ■

2.5 Study of Common Running Times

2.5.1 Linear Time

An algorithm that runs in some constant multiple of the size of the input is known as a Linear time algorithm, it is represented by *O(n)*. An algorithm requires linear time when it spends a constant time on each of its input item. Consider an algorithm which processes the input in a single pass, and spends a certain constant amount of time on each item of the input encountered. Depending on the way a problem is solved decides the time bound of the algorithm, other algorithms achieve a linear time bound for other reasons. A simple example of such an algorithm would be to find the highest element from an array of numbers.

2.5.2 *O(nlogn)* Time

O(nlogn) is also a very common running time, generally it is the running time of any algorithm that splits its input into two equal-sized pieces, solves each piece recursively, and then combines the two sorted parts into one in linear time. *Merge-sort algorithm* is an example which *O(nlog n)* time. In case of *Merge-sort*

algorithm, the input is divided into two equal-sized pieces. These two sets are sorted independently and then merged into a single sorted output in linear time.

2.5.3 Quadratic Time

Quadratic time arises naturally in case of *nested loops*, Consider a basic problem: given n points in the plane, such that each of the vertex is specified by (x, y) coordinates, the task is to find the pair of points that are closest together. The natural brute-force algorithm for this problem would enumerate all pairs of points and compute the distance between each pair. The last step would be to find the smallest distance, and the corresponding points would give us the desired pair. This problem can be solved by writing an algorithm with two nested loops for calculating the distance between each pair of points. This would require quadratic time.

2.5.4 Cubic Time

From the above idea, we can guess that more elaborate sets of nested loops often lead to algorithms that run in $O(n^3)$ time. Consider, for example, the following problem. We are given sets V_1, V_2, $\cdots$, V_n, each of which is a subset of $\{1, 2, \cdots, n\}$. We would like to know whether some pair of these sets is disjoint in other words, has no elements in common. This problem can be solved by an algorithm which has three nested loops, and each loops runs in *O(n)* time.

2.5.5 $O(n^k)$ Time

In a similar fashion, the running time of $O(n^k)$ for any constant k, can be obtained. We obtained a running time of $O(n^2)$ by performing brute-force search over all pairs formed from a set of n items. This would require a running time of $O(n^k)$ for any constant k when we search over all subsets of size k.

2.5.6 Beyond Polynomial Time

In general, not all algorithms have polynomial bounds, two kinds of bounds that come up very frequently are 2^n and $n!$. Let us now try to understand why this is

so. They grow much faster compared to the polynomial type algorithm.

Suppose, we are given a graph and we have to find an independent set of maximum size, instead of testing the existence of one with a given number of nodes. This is similar to the brute-force algorithm for k-node independent sets, except that now we are iterating over all the possible subsets of the graph. We know that the total number of subsets of an n-element set is 2^n, so the outer loop in this algorithm will have 2^n number of iterations as it tries all these subsets. Inside the loop, we are checking all pairs from a set S that can be as large as n nodes, so each iteration of the loop takes at most $O(n^2)$ time. Multiplying these two together, we get a running time of $O(n^2 2^n)$. This is an example of beyond polynomial time algorithm.

2.5.7 Sublinear Time

A algorithm whose execution time, $f(n)$, grows asymptotically slower than the size of the problem, n, for processing the input and giving us approximate correct answer forms the example of a Sublinear Time algorithm. Consider the cases in which the running times that are asymptotically smaller than linear time. Since, reading the input itself takes linear time in the size of the input, these situations tend to arise in a model of computation where the input can be read indirectly rather than completely. The goal in such cases is to minimize the amount of querying (the input) that must be done.

2.6 Data Structures

2.6.1 Arrays

An array is a basic data structure which stores a set of values in the same name. Arrays are amongst the oldest and most important data structures. It is represented by <Array name> <[number of values to be stored]> e.g. A[200]. Consider a situation in which we need to store the name of students of a class given that the number of students is 200. If we use different variable for storing all these 200 different names, it will be a very cumbersome process, and confusing at

the same time. Here, we feel the need of such a data structure which can store all the names one after the other in the same name and such that it can be processed easily. The entries of the array can also be located and found relatively each. In an array the ith boy/girl can be found just by finding the entry A[i].

2.6.2 Linked List

A similar data structure is a *LinkedList*, the difference between linked list and array is that, unlike arrays in addition to data storage it has a link which points to the next element. It comes handy when we have a dynamic list of entries. Inserting and deleting any element becomes much easy compared to array. Consider a set of numbers sorted in increasing order, if we store these numbers in an array, inserting and deleting becomes a little clumsy, because after finding the location of the element to be inserted we need to shift all the later elements by one place to accommodate the new element. In case of a linked list we don't have to do this shifting business, we only need to point the pointer of the one after which it has to be inserted to the new one and the pointer of the new element to the one before which it has to be located. Linked list can be of two type, one which has a single pointer and another which is doubly linked. A doubly linked list has two pointers with each element, one points to the next element and the another one points to the previous one.

2.6.3 Priority Queue

A priority queue is a data structure that maintains a set of elements K, such that each element $e \in K$ has an associated value $key(v)$ which indicates the priority of the element e; the smaller key value represent element with higher priorities. Priority queues can be used to insert and delete the elements in the set. The element with the highest priority i.e. the smallest key value is the first one to get selected.

Consider a general situation, in a public function there is a priority given to the VVIP's, VIP's and other special guests. This can be a perfect example of a priority Queues in a real life situation. A motivating application for priority queues, that is useful to keep in mind when considering their general function,

is the problem of managing real-time events such as the scheduling of tasks on a machine. Each process has a priority, or urgency, but processes do not arrive in order of their priorities. At any time, we have a current set of active processes, and we want to be able to extract the one with the currently highest priority and run it. We can be done with the help of a priority Queue, maintain the set of processes, according to the key of a process representing its priority value. Scheduling the highest-priority process corresponds to selecting the element with minimum key from the priority queue; at the same time, new processes will arrive and we will need to insert the new process according to their priority values.

Chapter 3

Graphs

The study of Graphs started with the famous problem of "The bridges of Königsberg". This problem was solved for the first time by Leonhard Euler, and in the process of finding the solution began this branch of Mathematics, which is now known as *Graph Theory*. A graph is a figure consisting of points (called vertices) and lines (or curves) connecting these vertices (called edges). The more one works with graphs, the more often we find it real life situations and its application everywhere. It has a lot of applications and has turned out to be a very important tool for Mathematics.

3.1 Definition and Some Examples

A graph G is simply a way of encoding pairwise relationships among a set of objects: it consists of a collection V of nodes (vertices) and a collection E of edges, each of which "joins" two of the nodes. We thus represent an edge $e \in E$ as a two-element subset of $V : e = \{u, v\}$ for some $u, v \in V$, where we call u and v the ends of edge e [6]. These edges could be directed or undirected, depending upon the situation we consider. For example, if we consider a group of cities connected by roads, it represents an undirected graph, whereas consider a network of pipes supplying water in any city, in this case the water flows from the reservoir(s) to all the parts of the city, which is an example of a directed graph. If $e = \{u, v\}$ is a directed edge, then the vertex u is called the *tail* and v is called the *head*.

Examples of Graphs Let us have a look at a few examples which will demonstrate us the usefulness and versatility of graphs. We will see how it can be used in different situations to study the problem.

Transportation Networks The maps of the airline routes could be an example of a graph. The airports can be represented by vertices and the routes could form the edges. Since if their is a route from one airport to another, then aeroplanes travel both ways, start from *city* 1 and end at *city* 2 or start from *city* 2 and end at *city* 1. Hence, this would be an example of a undirected graphs. Similarly, train routes and railway station are other examples of undirected graph.

Communication Networks A collection of terminals connected to each other through some communication channel can be modeled as a graph in a few different ways. "First, we could have a node for each computer and an edge joining u and v if there is a direct physical link connecting them. Alternatively, for studying the large-scale structure of the Internet, people often define a node to be the set of all machines controlled by a single Internet service provider, with an edge joining u and v if there is a direct *peering relationship* between themroughly, an agreement to exchange data under the standard BGP protocol that governs global Internet routing. Note that this latter network is more "virtual" than the former, since the links indicate a formal agreement in addition to a physical connection." [6]

Information Networks The World Wide Web can be viewed as a natural example of a directed graph. Suppose a Web page represents a node, and it has many different links to different web page, consider this link as an edge from, say l to m, where l represents the web page which has a link to the other web page, say m. This is clearly an example of a directed graph, in case the new page does not have a link to the previous web page. There are types of web pages which have links both ways, these kind of pages represent undirected graphs. Most of the search engines use this kind of structure to link and infer to important pages on the World Wide Web.

Social Networks In a social group, we can easily find the application of a graph. In any collection of people, for example, people living in a society, the employees of a company, the members of a committee, the residents of a city, etc., we can form a graph in which the people represent the nodes and there exist an edge between them depending on the condition if they are friends or if they share some kind of a relationship with each other. Different relationships will have different representations. An edges could mean many different things instead of friendship: the undirected edge (u, v) could mean that u and v have had a romantic relationship or a financial relationship; the directed edge (u, v) could mean that u is the mentor of v, or that u lists v in his or her e-mail address book. There can also be examples in which we could imagine bipartite social networks based on a notion of affiliation: given a set X of companies and a set Y of products, we could define an edge between $u \in X$ and $v \in Y$ if a company u manufactures the product v.

3.2 Representation of Graphs

Let us describe the two standard ways to represent a graph $G = (V, E)$: either as a collection of adjacency lists or as an adjacency matrix. Both the ways can be used for both directed and undirected graphs. The adjacency-list representation is generally preferred over the adjacency-matrix representation. The graphs for which $|E|$ is much less compared $|V|^2$ uses the list form of representation, because it requires much less space for storing the network connections. Whereas, in case when the graph has number of edges comparable to $|V|^2$, the matrix representation is used, because in such a representation it is very easy and at the same time very less time consuming to find if any pair of vertices are connected or not. The Figure 3.1 shows an undirected graph and the Figure 3.2 shows the pictorial representations of the these two different forms.

The ***adjacency-list representation*** of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V. For each $u \in V$, the array $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$. That is, $Adj[u]$ consists of all the vertices adjacent to $u \in G$. In other words the adjacency list

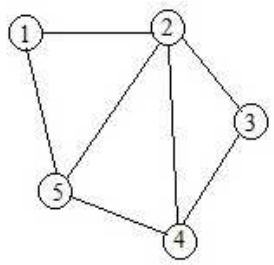

Figure 3.1: An Undirected Graph

of each of the vertex contains a pointer to its neighbour. The vertices in each adjacency list have no particular order, they are stored in any order. Figure 3.2(a) is an adjacency-list representation of the undirected graph in Figure 3.1.

So, the adjacency list of all the vertices will have a pointer to all the vertices it is connected, hence in a directed graph, the sum of the lengths of all the adjacency lists, in a graph G is $|E|$. Note that in a directed graph, an edge of the form (u, v) is represented by having v appear in $Adj[u]$. If G is an undirected graph, then an edge will appear twice in the adjacency list of all the vertices, and hence the sum of the lengths of all the adjacency lists would be $2|E|$. Let us now try to analyze the amount of memory it requires, since for both directed and undirected graphs, the adjacency-list representation has to pass through all the edges for each of the vertex, the space needed to store this information would be of the order of $\Theta(V + E)$. Adjacency lists can also be used to represent ***weighted graphs***, in such a case we would store the weight of the edge along with the vertex v. In a weighted graphs, each edge has an associated ***weight***, typically given by a ***weight function*** $\rho : E \rightarrow N$. For example, let $G = (V, E)$ be a weighted graph with weight function ρ. The weight $\rho(u, v)$ of the edge $(u, v) \in E$ is stored in $u^{'}$s adjacency list along with the vertex v. The kind of representation is quite robust, it can be used in many different situations.

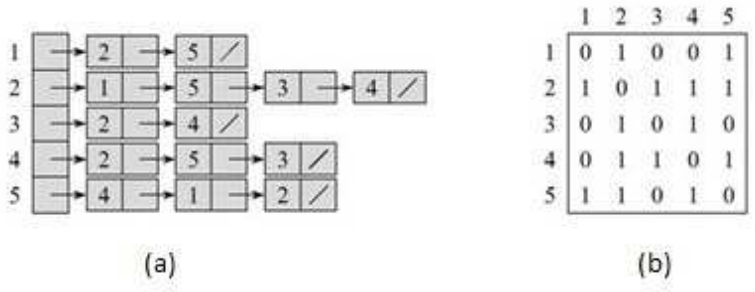

Figure 3.2: (a) The adjacency list representation (b) The adjacency matrix representation of the undirected graph in fig 3.1[11]

But if we want to find whether a particular edge between the vertices say u and v, is present or not. The only way to find this is to search if v is present in the $Adj[u]$ list or not. This is one disadvantage of $adjacency-list\ representation$ as finding whether the two vertices are connected or not is not so quick. This disadvantage can be resolved by using an $adjacency-matrix\ representation$ of the graph, but at the price of using more memory than the previous represenattion.

The ***adjacency-matrix representation*** of a graph $G = (V, E)$, with vertices numbered $1, 2, ..., |V|$ in some arbitrary manner, consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in \text{E}, \\ 0 & \text{otherwise.} \end{cases}$$

Figure 3.2(b) shows the adjacency-matrix of the undirected graph. Clearly, the adjacency matrix of a graph requires $\Theta(V^2)$ memory, since the matrix has (V^2) number of memory units containing the information, whether there exists an edge or not.

In case of an undirected graph, there is a symmetry in the matrix representation along the main diagonal of the adjacency matrix, for example in Figure 3.2(b). The transpose of a matrix $A = (a_{ij})$ is defined to be the matrix $A^T = (a_{ij}^T)$

given by $a_{ij}^T = a_{ji}$. Since in an undirected graph, (u, v) and (v, u) represent the same edge, the adjacency matrix A of an undirected graph is its own transpose: $A = A^T$. Hence, we do not actually need to store all the entries in the matrix, it is prudent to store only the entries on and above the diagonal of the adjacency matrix, and hence cutting the memory required to store the graph almost in half.

The adjacency-matrix representation can clearly be used for weighted graphs. For example, if $G = (V, E)$ is a weighted graph with edge-weight function ρ, the weight $\rho(u, v)$ of the edge $(u, v) \in E$ can be simply stored as the a_{uv} entry in the adjacency matrix. If an edge does not exist, a NIL value can be stored as its corresponding matrix entry, however for many problems it is better to use either 0 or ∞.

Even though, the adjacency-list representation is asymptotically at least as efficient as the adjacency-matrix representation, the adjacency matrix representation is used because of its ability to store the weight of the edges rather easily, and hence preferable even when graphs are reasonably small. Moreover, if the graph is unweighted, the additional advantage in using the adjacency-matrix representation is that, instead of using one word of computer memory for each matrix entry, the adjacency matrix uses only one bit per entry.

3.3 Search Algorithms

This section presents two basic and important methods for searching a graph. It means systematically following the edges of the graph so as to visit all the vertices of the graph. Searching techniques for a graph is at the heart of graph algorithms.

3.3.1 Breadth First Search

Breadth-first search is one of the simplest algorithms for searching a graph and it serves as a model or a basis for many important graph algorithms. Prims minimum-spanning-tree algorithm and Dijkstra′s single-source shortest-paths algorithm are based on the ideas similar to those in breadth-first search.

Given a graph $G = (V, E)$ and a ***source*** vertex s, breadth-first search system-

atically explores the edges of G to discover every vertex that can be reached from s. And in doing so, it computes the distance from s to each reachable vertex, which turns out to be the smallest number of edges from s to v. For any vertex v reachable from s, the path in the breadth-first tree from s to v, gives us the shortest path, in other words a path containing the smallest number of edges. This algorithm works for both directed and undirected graphs.

Breadth-first search is so named because of the way it discovers the vertices of the graph. It expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices which are at a the minimum distance from s and then discovers all the vertices which are at a further distance.

In the process we obtain a tree, which we call a breadth-first tree, initially containing only its root, which is the source vertex s. Initially all the vertex are coloured white, when a white vertex v is discovered in the course of scanning the adjacency list of an already discovered vertex u, this new vertex v and the corresponding edge (u, v) are added to the tree. We say that u is the ***predecessor*** or ***parent*** of v in the breadth-first tree. In this algorithm each vertex is discovered at most once, and hence it has at most one parent. "Ancestor and descendant relationships in the breadth-first tree are defined relative to the root s as usual: if u is on a path in the tree from the root s to vertex v, then u is an ancestor of v and v is a descendant of u." [11]

The breadth-first-search algorithm described below assumes that the input graph $G = (V, E)$ is represented using adjacency lists. The variable $color[u]$ stores the color of each vertex $u \in V$, and the variable $\pi[u]$ stores the predecessor of u. In case u has no predecessor (for example, if $u = s$ or u has not been discovered), then $\pi[u] =$ NIL. The distance from the source s to vertex u computed by the algorithm is stored in $d[u]$. The algorithm also uses a first-in, first-out queue to manage the set of gray vertices.

BFS(G, s) [11]

```
1    for each vertex u ∈ V[G] − s
2        do color[u] ← WHITE
3           d[u] ← ∞
```

```
4            π[u] ← NIL
5     color[s] ← GRAY
6     d[s] ← 0
7     π[s] ← NIL
8     Q ← φ
9     ENQUEUE (Q, s)
10     while Q ≠ φ
11         do u ← DEQUEUE(Q)
12             for each v ∈ Adj[u]
13                 do if color[v] = WHITE
14                     then color[v] ← GRAY
15                         d[v] ← d[u] + 1
16                         π[v] ← u
17                         ENQUEUE(Q, v)
18             color[u] ← BLACK
```

Analysis of the running time

Now, let us analyze the running time of the algorithm on the input graph $G = (V, E)$. The process starts with initializing white color to all the vertex (no vertex is ever whitened), which means that each vertex is enqueued. The step number 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once. The process of enqueuing or dequeuing takes $O(1)$ time, hence the total time taken for queue operations is $O(V)$. According to the algorithm, it is clear that each adjacency list is scanned at most once. Since the sum of the adjacency list is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(E)$. Hence, the total running time of the ***Breadth first search*** is $O(V + E)$. Thus, in other words the ***BFS*** runs in linear time in the size of the adjacency-list representation of G.

Shortest Paths

The ***breadth-first search*** finds the distance to each reachable vertex in a graph $G = (V, E)$ from the given source vertex $s \in V$. The ***shortest-path distance*** $\delta(s, v)$ from s to v is defined as the minimum number of edges in any path from

vertex s to vertex v; in case there is no path from s to v, then $\delta(s, v = \infty)$. A path of length $\delta(s, v)$ from s to v is said to be a shortest path from s to v, and a ***BFS*** actually computes these ***shortest-path distances***.

The ***BFS*** algorithm builds a breadt-first tree in the process of searching the graph. The tree is represented by the π field in each vertex. The following algorithm prints out the vertices on a shortest path from s to v, assuming that ***Braedth-first search*** has already been run to compute the shortest-path tree.

```
PRINT-PATH(G, s, v) [11]
1    if v = s
2        then print s
3        else if π[v] = NIL
4                then print "no path from" s "to" v "exists"
5                else PRINT-PATH(G, s, π[v])
6                     print v
```

This procedure takes linear time in the number of vertices in the path printed.

3.3.2 Depth First Search

This search algorithm follows a different procedure for finding the graph vertices. This algorithm is so named because of its property to search "deeper" in the graph whenever possible. The underlying idea behind ***depth-first search***, is to explore the edges out of the most recently discovered vertex v that still has unexplored edges leaving it. When all edges of a vertex is explored, the search backtracks to explore edges leaving the vertex from which v was discovered. This process continues until we discover all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then one of them is selected as a new source and the search procedure is repeated. This entire process is repeated until all vertices are discovered.

As in breadth-first search, whenever a vertex v is discovered during a scan of the adjacency list of an already discovered vertex u, depth-first search records

this event by setting $v^{'}$s predecessor field $\pi[v]$ to u. The difference between the two is that in breadth-first search the predecessor subgraph forms a tree, whereas the predecessor subgraph produced by a depth-first search may be composed of several trees, because the search may be repeated from multiple sources. The ***predecessor subgraph*** of a depth-first search is therefore defined slightly differently from that of a breadth-first search: we let $G_\pi = (V, E_\pi)$, where
$E_\pi = \{(\pi[v], v) : v \in V and\ \pi[v]\ \neq \text{NIL}\}$.
The predecessor subgraph of a depth-first search forms a ***depth-first forest*** composed of several ***depth-first trees***. The edges in E_π are called ***tree edges***.

In both the cases, the color of the vertex indicates its state. In the depth first search, each vertex is initially white, is grayed when it is discovered in the search, and is blackened when it is finished, i.e. when its adjacency list has been examined completely. The trees formed in this way are disjoint and each vertex ends up in exactly one depth-first tree

The procedure DFS below records the time when it discovers vertex u in the variable $d[u]$ and also when it finishes vertex u in the variable $f[u]$. These timestamps are integers between 1 and $2|V|$, since there is one discovery event and one finishing event for each of the $|V|$ vertices. For every vertex u,

$$d[u] < f[u]. \tag{3.1}$$

Vertex u is WHITE before time $d[u]$, GRAY between time $d[u]$ and time $f[u]$, and BLACK thereafter.

The following algorithm below is the basic depth-first search algorithm. This algorithm also works for both undirected or directed graphs. The variable *time* is a global variable that we use for timestamping.

DFS(G) [11]
1 **for** each vertex $u \in V[G]$
2 **do** $color[u] \leftarrow$ WHITE
3 $\pi[u] \leftarrow$ NIL
4 $time \leftarrow 0$
5 **for** each vertex $u \in V[G]$

```
6       do color[u] = WHITE
7          then DFS-VISIT(u)
```

DFS-VISIT(u) [11]

```
1   color[u] ← GRAY      ▷ White vertex u has just been discovered.
2   time ← time + 1
3   d[u] ← time
4   for each v ∈ Adj[u]      ▷ Explore edge (u, v).
5      do if color[v] = WHITE
6         then π[v] ← u
7              DFS-VISIT(v)
8   color[u] ← BLACK      ▷ Blacken u; it is finished.
9   f[u] ← time ← time + 1
```

Note that the results of depth-first search may depend upon the order in which the vertices are examined in line 5 of DFS, and upon the order in which the neighbours of a vertex are visited in line 4 of DFS-VISIT. These different visitation orders tend not to cause problems in practice, as any depth-first search result can usually be used effectively, with essentially equivalent results.

Analysis

The loops on lines 1-3 and lines 5-7 of DFS takes time $\Theta(V)$, exclusive of the time to execute the calls to DFS-VISIT. The procedure DFS-VISIT is called exactly once for each vertex $v \in V$, since DFS-VISIT is invoked only on white vertices and the first thing it does is paint the vertex gray. During an execuation of DFS-VISIT(v), the loop on lines 4-7 is executed $|Adj[v]|$ times. Since

$$\sum_{v \in V} |Adj[v]| = \Theta(E)$$

the total cost of executing lines 4-7 of DFS-VISIT is $\Theta(E)$. the running time of DFS is therefore $\Theta(V + E)$.

Chapter 4

Algorithmic Techniques

In this chapter we will discuss the technique of *Greedy algorithms*. With the help of examples we will illustrate this technique and observe its advantage over other standard techniques. The other very important tool is the method of *Divide and Conquer*. These techniques are very easy to understand but at the same time very powerful. Let us now study the *Greedy algorithms*.

4.1 Greedy Algorithms

Somebody said and I quote, "Greed $\cdots$ is good. Greed is right. Greed works." It is hard, to define precisely what is meant by a *greedy algorithm*. An easy way to understand this would be to think of an algorithm which tries to be better than other algorithms by greedily utilizing its resources and trying to give the best result. One can design many different greedy algorithms for the same problem. A greedy algorithm builds up in small steps, by choosing the option which optimizes some of the underlying criteria desired in the output.

Greedy algorithm helps us to solve some of the nontrivial problems optimally, which implies something very interesting and useful about the structure of the problem itself. there is a local decision rule that one can use to construct optimal solutions. It is relatively easy to invent greedy algorithms for almost any kind of problem; finding cases in which they work well, and proving that they work well, is the interesting challenge.

In this chapter we will discuss a simple approach, for proving that a greedy algorithm produces an optimal solution to a problem. By this we mean that if one measures the greedy algorithm's progress in a step-by-step fashion, one sees that it does better than any other algorithm at each step; it then follows that it produces an optimal solution. Another approach to find a greedy algorithm is a more general one. The second approach is known as an exchange argument: one considers any possible solution to the problem and gradually transforms it into the solution found by the greedy algorithm without hurting its quality. Again, it will follow that the greedy algorithm must have found a solution that is at least as good as any other solution. Some of the most well-known applications of greedy algorithms are: shortest paths in a graph, the Minimum Spanning Tree Problem, and the construction of Huffman codes for performing data compression[6].

In the Figure 4.1, the three instances show that(a), the algorithm fails if we

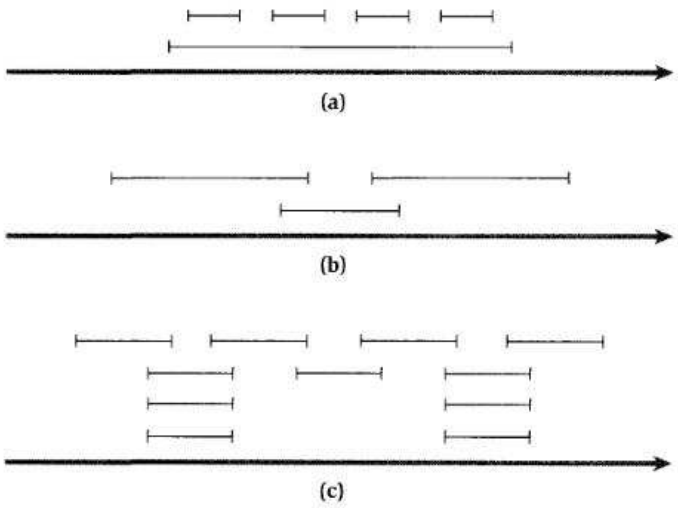

Figure 4.1: Instances of Scheduling Problem [6]

select the event that starts earliest; in (b), similarly selecting the shortest interval doesn't work; and in (c), if we try to select the ones which have the lowest conflict

it does not work.

4.1.1 Interval Scheduling Problem

Consider the following very simple scheduling problem. You have a resource- it may be a auditorium, a supercomputer, or an electron microscope and many people request to use the resource for periods of time. A request takes the form: Can I reserve the resource starting at time s, until time f? We will assume that the resource can be used by at most one person at a time. A scheduler wants to accept a subset of these requests, rejecting all others, so that the accepted requests do not overlap in time. Let us say, we have a set of requests {1,2,..., n}; the ith request corresponds to an interval of time starting at $s(i)$ and finishing at $f(i)$. We'll say that a subset of the requests is *compatible* if no two of them overlap in time, and our goal is to accept as large a *compatible* subset as possible. Compatible sets of maximum size will be called *optimal*.

Designing a Greedy Algorithm

The basic idea in a greedy algorithm for interval scheduling is to use a simple rule: select a first request i_1. Once a request i_1 is accepted, reject all requests that overlap with the request i_1. Then select the next request i_2 to be accepted, and similarly reject all the requests that are not compatible with i_2. Repest this process until all the requests are processed. It might seem quite obvious but designing a good greedy algorithm is a challenging task. There are many natural rules which may not give us the optimum result.

Let us now try to study some of the rules and understand how they work.

- The first thing that comes to the mind is to select a request which starts at the earliest, that is, the one with minimal start time $s(i)$. In this way our resource starts being used as quickly as possible.

  A look at the situation depicted in the Figure 4.1(a) shows a similar situation. But this method does not give an optimal result. If the first request to be accepted is for a very large interval, then this would lead to rejection of many other request which could have been accepted, if we had not accepted the first request. Consider the situation shown in figure, by this rule we

would be able to accept only one single request, while the optimal could have many requests.

- This might suggest that we should start out by accepting the request that requires the smallest interval of timenamely, the request for which $f(i)s(i)$ is as small as possible. It is a somewhat a better rule than the previous one, but still this rule does not give us the desired result. For example, in Figure 4.1(b), accepting the short interval, prevents us from accepting two request which overlaps with the middle one. Hence, this rule also fails to the optimal result.

- In the third attempt, we could design a greedy algorithm that is based on this idea: for each request, we count the number of other requests that are not compatible, and accept the request that has the fewest number of non-compatible requests. This greedy choice would give us the optimum solution in the previous example. But consider the situation shown in Figure 4.1(c). The unique optimal solution is to accept the four requests in the top row. The greedy method used in this case accepts the middle request in the second row and thereby gives an output of size no greater than three.

After having analyzed at the three greedy rule, let us now look at the fourth one: in this case we accept first the request that finishes first, that is, the request i for which $f(i)$ is as small as possible. This is also quite a natural idea: this would ensure that our resource becomes free as soon as possible while still satisfying one request. In this way we can maximize the time left to satisfy other requests. Let us state the algorithm a bit more formally. We will use R to denote the set of requests that we have neither accepted nor rejected yet, and use A to denote the set of accepted requests. For an example of how the algorithm runs, see Figure 4.2.

Interval Scheduling [6]

```
1    Initialize R = 1, 2, 3, ···, n and A = φ
2        While R =≠ φ
3            Choose a request i ∈ R with the smallest finishing time
4            Add request i to A
5            Delete all the request from R that are not compatible with i
```

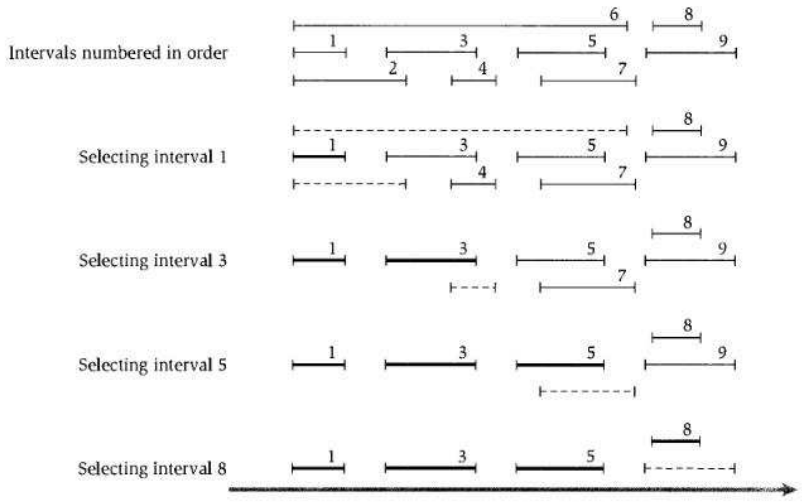

Figure 4.2: Sample run of the Interval Scheduling Algorithm. At each step the selected intervals are darker lines, and the intervals deleted at the corresponding step are indicated with dashed lines.[6]

6 **EndWhile**

7 Return the set A as the set of accepted requests.

Analyzing the Algorithm

While this greedy method is quite natural, it is certainly not obvious that it returns an optimal set of intervals. Indeed, it would only be sensible to reserve judgment on its optimality: the ideas that led to the previous non-optimal versions of the greedy method also seemed promising at first.

As a start, we can immediately declare that the intervals in the set A returned by the algorithm are all compatible. Now, we need to show that the solution obtained is optimal. So, for purposes of comparison, let $\mathcal{O}$ be an optimal set of intervals. Ideally one might want to show that $A = \mathcal{O}$, but this is too much to ask: there may be many optimal solutions, and at best A is equal to a single one of them. So instead we will simply show that $|A| = |\mathcal{O}|$, that is, that A contains the same number of intervals as $\mathcal{O}$ and hence is also an optimal solution.

The idea underlying the proof, as we suggested initially, will be to find a sense in which our greedy algorithm "stays ahead" of this solution 0. We will compare the partial solutions that the greedy algorithm constructs to initial segments of the solution $\mathcal{O}$, and show that the greedy algorithm is doing better in a step-by-step fashion.

We introduce some notation to help with this proof. Let i_1, ..., i_l be the set of requests in A in the order they were added to A. Note that $|A| = l$. Similarly, let the set of requests in $\mathcal{O}$ be denoted by j_1,... ,j_m. Our goal is to prove that $l = m$. Assume that the requests in $\mathcal{O}$ are also ordered in the natural left-to-right order of the corresponding intervals, that is, in the order of the start and finish points. Note that the requests in $\mathcal{O}$ are compatible, which implies that the start points have the same order as the finish points. Our intuition for the greedy method came from wanting our resource to become free again as soon as possible after satisfying the first request. And indeed, our greedy rule guarantees that $f(i_1) \leq f(j_1)$. This is the sense in which we want to show that our greedy rule "stays ahead"that each of its intervals finishes at least as soon as the corresponding interval in the set 0. Thus we now prove that for each k > 1, the kth accepted request in the algorithm's schedule finishes no later than the kth request in the

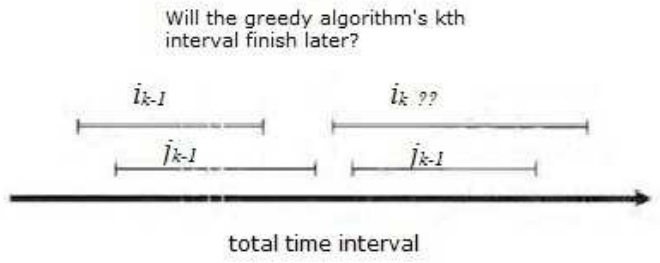

Figure 4.3: The inductive step in the proof that the greedy algorithm stays ahead.

optimal schedule.

(4.1) *For all vertices $k \leq l$ we have $f(i_k) \leq f(j_k)$*

Proof. We prove this statement by the use of induction. For k = 1 the statement is clearly true: the algorithm starts by selecting the request i_1 with minimum finish time.

Now let $k > 1$. From the induction hypothesis assume that the statement is true for $k - 1$, and we will try to prove it for k. As shown in Figure 4.3, the induction hypothesis lets us assume that $f(i_{k-1}) \leq f(j_{k-1})$.

We know (since $\mathcal{O}$ consists of compatible intervals) that: $f(j_{k-1}) \leq s(j_k)$. Combining this with the induction hypothesis $f(i_{k-1}) \leq f(j_{k-1})$, we get $f(j_{k-1}) \leq s(j_k)$. Thus the interval j_k is in the set R of available intervals; at the time when the greedy algorithm selects i_k. The greedy algorithm always selects the available interval with smallest finish time; since interval j_k is one of these available intervals, we have $f(i_k) \leq f(j_k)$. This completes the induction step ■

In this way we have designed a greedy method which selects the interval which finishes at least as soon as the kth interval in $\mathcal{O}$. We now see why this implies the optimality of the greedy algorithms set A.

(4.2) *The greedy algorithm returns an optimal set A.*

Proof. We will prove the statement by contradiction. If A is not optimal, then an optimal set $\mathcal{O}$ must have more requests, that is, we must have $m > l$. Applying (4.1) with $k = l$, we get that $f(i_l) < f(j_l)$. Since $m > l$, there is a request j_{l+1} in $\mathcal{O}$. This request starts after request j_l ends, and hence after i_l ends. So after deleting all requests that are not compatible with requests $i_1, ..., i_l$, the set of possible requests R still contains j_{l+1}. But the greedy algorithm stops with request i_l, and it is only supposed to stop when R is emptya contradiction■

Implementation and Running Time We can make our algorithm run in time *O(n log n)* as follows. We begin by sorting the n requests in order of finishing time and labeling them in this order; that is, we will assume that $f(i) \leq f(j)$ when $i < j$. This takes time *0(n log n)*. In an additional *O(n)* time, we construct an array $S[l...n]$ with the property that $S[i]$ contains the value $s(i)$.

We now select requests by processing the intervals in order of increasing $f(i)$. We always select the first interval; we then iterate through the intervals in order until reaching the first interval j for which $s(j) \geq f(l)$; we then select this one as well. More generally, if the most recent interval we've selected ends at time f, we continue iterating through subsequent intervals until we reach the first j for which $s(j) \geq f$.In this way, we implement the greedy algorithm analyzed above in one pass through the intervals, spending constant time per interval. Thus this part of the algorithm takes time *O(n)*.

Chapter 5

Maximum Cut

For a graph, a ***maximum cut*** is a cut whose size is greater than the size of any other cut, i.e. to say the maximum number of edges we need to remove to divide the graph in two complimentary sets. The problem of finding a maximum cut in a graph is known as the max-cut problem. The problem can be stated simply as follows.

> One wants a subset K of the vertex set such that the number of edges between K and the complementary subset is as large as possible.

Consider an advanced version of this problem, suppose that each edge of the graph has some weight, then the problem is called weighted max-cut. In this version each edge has a real number, its weight, and the objective is to maximize not the number of edges but the total weight of the edges between K and its complement. The weighted max-cut problem is often, but not always, restricted to non-negative weights, because negative weights can change the nature of the problem.

Computational Complexity

The following decision problem related to maximum cuts has been studied widely in theoretical computer science: Given a graph G and an integer k, determine whether there is a cut of size at least k in G. This problem is known to be NP-complete. It is easy to see that problem is in NP: a yes answer is easy to prove by presenting a large enough cut. The NP-completeness of the problem

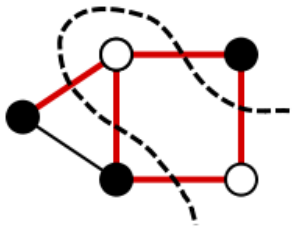

Figure 5.1: A maximum cut

can be shown, for example, by a transformation from maximum 2-satisfiability (a restriction of the maximum satisfiability problem)[3].

The canonical optimization variant of the above decision problem is usually known as the maximum cut problem or max-cut problem and is defined as: Given a graph G, find a maximum cut.

Polynomial time Algorithm

As the max-cut problem is NP-hard, no polynomial-time algorithms for max-cut in general graphs are known. However, a polynomial-time algorithm to find maximum cuts in planar graphs exists.

Approximation Algorithms

There is a simple randomized 0.5-approximation algorithm: for each vertex flip a coin to decide to which half of the partition to assign it[7] [8]. In expectation, half of the edges are cut edges. This algorithm can be derandomized with the method of conditional probabilities; therefore there is a simple deterministic polynomial-time 0.5-approximation algorithm as well[7] [9]. One such algorithm is: given a graph $G = (V, E)$ start with an arbitrary partition of V and move a vertex from one side to the other if it improves the solution until no such vertex exists. The number of iterations is bounded by because the algorithm improves the cut value by at least 1 at each step and the maximum cut is at most . When the algorithm

terminates, each vertex has at least half its edges in the cut (otherwise moving v to the other subset improves the solution). Therefore the cut is at least.

The best known max-cut algorithm is the 0.878-approximation algorithm by Goemans and Williamson using semidefinite programming and randomized rounding [4] [5]. It has been shown by Khot et al [10] that this is the best possible approximation ratio for Max-Cut assuming the unique games conjecture.

Chapter 6

Simple Algorithm for Conflict Resolution in the Scheduling of Television Commercials

Advertisements of products/services are aired on the television/radio shows during the program breaks. The advertisements (also called commercials) are given different slots according to the advertisers choice/preferrence. These slots are hence sold to the advertisers, who select according to their preference the percentage of slots of their choice, where they want their commercials to be aired. Earlier, this task was done manually. With the increase in the number of commercials and demand for the best possible conflict free scheduling, that too within a time constraint, this task turns out to be very complicated, tedious and error prone. Models have been designed to solve this effectively using programming techniques. This problem is known to be NP-hard. The model proposed by Gaur et al. [2] gives us a model to solve this in polynomial time. We extend their model for scheduling commercial advertisements during breaks in television programming. In this chapter we propose an algorithm, which gives us the partition of m vertices with a performance ratio, which is the same as that of the algorithm designed by Gaur et al. in quadratic time. We have used the formulation as a capacitated generalization of the max k-cut problem in which the vertices of a graph correspond to commercial insertions and the edge weights of the conflicts

between pairs of insertions. The vertices are partitioned into k capacitated sets and the objective is to maximize the sum of the conflict weights across partition and also to design an efficient algorithm.

6.1 Scheduling of Television Commercials

The television broadcasts have breaks between the programs where the commercials are aired. The program breaks are hence sold to the advertisers. Certain locations are more desirable to the clients than others. Generally the most coveted are the first and the last slots. So, the advertisers choose the slots according to their need and preference. It is desired by the clients that the commercials of the competing brands/products are separated as far as possible. They also have the freedom to decide their competitors and which specific brands, products or type of products not to be aired along with their products. Bollapragada and Garbiras[1] studied this problem and designed a method to solve this problem algorithmically. Gaur et al. [2] studied the model and came up with a generalized model, which allows differential weighting of conflicts between pairs of insertions. They have discussed the problem and found a polynomial time algorithm with an improved performance ratio. In this chapter we have proposed a quadratic time algorithm with the same performance ratio for the generalized model of Bollapragada and Garbiras[1]. The generalization model makes a distinction between multiple insertions of the same commercials, which are generally prohibited in the same break. The algorithm proposed by Gaur et al. [2], solves this problem in polynomial time. In this paper we propose an algorithm which does this conflict resolution in quadratic time with better optimization. We use the same model as proposed by Gaur et al. [2], with a slight change in the way we partition the set of advertisements which changes the optimal value. The advantage of this model is to be able to accommodate all the possible types and pairs of combinations of advertisements with the capacity to properly partition and differentiate properly. Finally, differences in the conflict weights can be used to represent varying degrees to which comparing commercials, or classes of commercials, are in conflict. For example, it is likely to be less desirable that the same program break contain commercials of directly competing brands (e.g., Dove, Liril, Cinthol and Dettol

soaps) than the brands that are more distant competitors (e.g., Dove soaps and Dettol antiseptic liquid). The special case in which all conflict weights are zeroes or ones corresponds to the assumption of equal conflict weights by Bollapragada and Garbiras (2004) [1].

6.2 The Problem

This problem is defined separately for each of the television program. We have a set of advertisements that are to be arranged during the breaks as per the requirement of their clients. A 1 hour show generally has six to ten breaks. As used by Gaur et al., we refer to these commercials as *insertions*, which occupies a slot in a break. We assume that these commercials are some integer multiple of some minimum length. Any practical time duration can serve this purpose - say for that matter 5 seconds, 10 seconds etc. A program break also is an integer multiple of this specified or chosen minimum duration. The problem we are studying is to find an efficient way to schedule these advertisements such that the conflict between them is as small as possible. By conflict, we mean the commercials of the competing products. They should not be placed in the same break or one after the other, they should be placed as far as possible. The buyers may specify the competing brands which should not be aired in the same break. Some competing brands might be produced by the same firm. For example, Luvs and Pampers are brands of diapers that are both manufactured by Procter and Gamble, but that should typically not be advertised in the same break. Other competing brands might be produced by different firms. For example, Acura, Lexus, and Sonata are three competing cars manufactured by Honda, Lexus, and Hyundai, respectively. However, Acura and Lexus compete more closely against each other than they do against Sonata. So, we assign very high weights to closely competing commercials in such a way that the chance of those two coming in the same break is very less.

We associate a nonnegative conflict weight with each pair of insertions. A small value of the conflict weight signifies that those two commercials have small conflict between them, and similarly a large value means the two are close competitors. To do this we represent these *insertions* as vertices and is connected

to all the remaining of the vertices in the graph. Every edge is assigned a weight which is a measure of the conflict between the two advertisements. The next section shows the formal representation of the problem.

Formulation of the problem[2]

Consider a graph G(V,E) and $|V| = m$ vertices and $|E|$ edges. An insertion is represented by a vertex and the conflict weight between two advertisements is the edge weight between those two respective vertices. We assign a conflict weight w_e to edge $e \in E$. A k-cut of the graph places the vertices into k mutually exclusive sets, V_1, V_2, $\cdots$, V_k, and each V_i correspond to the ith program break. And without any loss of generality, we consider that all these breaks have m insertions according to the size of the break and the number of insertions it can accommodate.

The conflict-resolution problem is to find an assignment of the insertions to the k program breaks so that the sum of the conflict weights across all pairs of program breaks is as large as possible. Let w_{uv} denote the conflict weight associated with insertions u and v for all $u, v \in V$. Let x_{ui} denote a 0-1 integer variable that takes a value of one only if insertion u is in program break i. Let y_{uivj} denote a 0-1 integer variable that takes a value of one only if insertion u is in program break i and insertion v is in a different program break j. The conflict-resolution problem can be formulated as the following 0-1 integer program:

Maximize

$$\sum_{(u,v)\in E} \sum_{i,j\in\{1,\cdots,k\},i\neq j} w_{uv} y_{uivj}$$

subject to

$$\begin{aligned} y_{uivj} \leq \quad & 1/2(x_{ui} + x_{vj}) \\ & \text{for all } (u,v) \in E, i, j \in \{1, \cdots, k\}, \\ \sum_{i=1}^{k} x_{ui} \leq \quad & 1 \text{ for all } u \in V, \end{aligned}$$

$$\begin{aligned}\sum_{u\in V} l_u x_{ui} \leq \quad & n_i \text{ for all } i \in \{1,\cdots,k\},\\ x_{ui} \in \quad & \{0,1\} \text{ for all } u \in V, i \in \{1,\cdots,k\},\\ y_{uivj} \in \quad & \{0,1\} \text{ for all } (u,v) \in E, i,j \in \{1,\cdots,k\}.\end{aligned}$$

The first constraint restricts the summing of conflict weights in the objective function to only those cases where the associated insertions are in different program breaks. The second constraint requires that each insertion should be assigned to only one program break. The third constraint specifies the capacity restriction for each program break: n_i denotes the number of units of the minimum unit length. If we change the third constraint accordingly then we can accommodate insertions and programme break of different length.

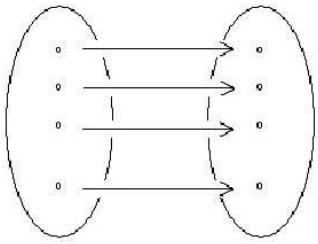

Figure 6.1: Representation of Matching 1

6.3 Algorithm

The idea behind the algorithm is to first partition all the m advertisements in n sets of size k (since $n = m/k$), in such a way that the conflict within the set is maximum. Once we get such a partition we separate each element of these sets in k different sets. In other words, we place all the elements which where in the same set, in different sets. Since these n sets had maximum conflict, when we separate all the elements of these sets, the conflict among them gets minimized.

Consider a situation, where we want to map k elements to k different sets. There can be many different ways to do it. Let us consider the following k mappings

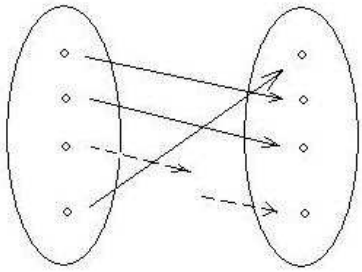

Figure 6.2: Representation of Matching 2

$$\begin{aligned} M_0 &: i \longrightarrow i, \\ M_1 &: i \longrightarrow i+1, \\ M_2 &: i \longrightarrow i+2, \\ &\vdots \\ M_{k-1} &: i \longrightarrow (i+k-1) \bmod k \end{aligned}$$

The *Matching* 0 maps ith element of the domain to the ith element of the co-domain. Similarly, *Matching* 1 maps ith element to the $i+1$th element and so on. If there is a weight attached to all these mappings, say $W_1, W_2, \cdots, W_k$, then there will exist a map amongst these maps, say M_j, such that

$$\text{weight of } M_j \leq \frac{\sum W_i}{k}$$

and for the remaining of the matching taken, the sum of the conflict weights will be $\geq (W - W/k)$, where $W = \sum W_j$. In our problem, we have a total $m(= kn)$ number of advertisements and k number of breaks. The distribution of these kn elements are done as follows.

Step 1

In this step we want to divide these $m = nk$ advertisements in sets of size k. So the number of such sets would be n. The process of dividing these advertisements is described below.

Suppose there are n sets and i elements have been already inserted in these sets. Let us now define a few notations as follows

W	being the weight of all the edges
W_0	denotes the sum of all the edges formed by the remaining $kn - i$ vertices to be inserted amongst themselves
v	be the $i + 1th$ element to be inserted
w_0	be the sum of the edges with respect to the remaining $kn - (i + 1)$ vertices that are to be inserted
C	be the sum of all the edge weights saved, and
W_j	denotes the total weight of the edges coming into S_j.

We would now try to figure out where to put in the next i.e.- the $i+1$th element. Let these n sets be denoted by S_1, S_2, $\cdots$, S_n. After i elements have been inserted suppose set S_1 has k_1 empty slots, S_2 has k_2 empty slots to put in the advertisements and so on S_j has k_j slots available and let the sum of these slots be denoted by M, i.e.

$$\sum k_j = kn - i = M(\text{say})$$

The $i + 1$th vertex, v can be inserted into any one of the sets S_1, S_2, $\cdots$, S_n. And depending on the number of elements present in these corresponding sets, or rather the number of remaining slots we will get the probability of the vertex v being inserted in any particular set. So, at any given time the ***Expectation value, say A*** would be equal to

$$A = W_0 \sum_{j=1}^{n} (\frac{k_j}{M})(\frac{k_{j-1}}{M-1}) + \sum_{j=1}^{n} (\frac{k_j}{M}) W_j + C$$

The *expectation value* is the averagre value of the weight that will be added to the total conflict weight. This value is calculated by using the probability of a particular advertisement being inserted in a break and the sum of the weight of the edges in that particular break. When no element was inserted in the sets S_1, S_2, $\cdots$, S_n, then

$$\begin{aligned} W_0 &= W \\ k_j &= k, \text{ for all } j \\ W_i &= 0, \text{ for all } i, and \\ C &= 0 \end{aligned}$$

This, give us the ***initial expectation value***, A_{init} equal to

$$A_{init} = W\frac{k-1}{M-1}$$

Fix v, now suppose v is placed in some S_l for $(k_l \geq 1)$, where $0 \leq l \leq n$. The new ***expectation value*** assuming v is placed in S_l would be, say B_l equal to

$$B_l = (W_0 - w_0)(\sum_{j=1,j\neq l}^{n} \frac{k_j}{M-1}\frac{k_j \dot{-} 1}{M-2} + \frac{k_l - 1}{M-1}\frac{k_l \dot{-} 2}{M-2}) + w_l + C + w_0\frac{k_l - 1}{M-1} + \sum_{j=1,j\neq l}^{n} \frac{(k_j)(W_j - w_j)}{M-1} + (W_l - w_l\frac{k_l \dot{-} 1}{M-1})$$

where w_l indicates the total weight of the edges from v to S_l. The conflict weight is contributed by the weights of the edges already inserted, the weights amongst the edges which are yet to be inserted and the conflict weight contribution from the individual breaks. Note that, the *expectation value* of the vertex v being put anywhere in S_j is A, so the weighted average of the B_j's i.e. $\sum_j B_j\frac{k_j}{M} = A$. Hence there exists atleast one j say, B_l such that a $B_l \geq A$, then we place v in S_l and we update the values of different parameters as follows

$$\begin{aligned} W_j &\leftarrow W_j - w_j \\ M &\leftarrow M - 1 \end{aligned}$$

$$\begin{aligned} A &\leftarrow (A - \sum_{j=1}^{n} \frac{w_j k_j}{M}) \frac{M}{M-1} \\ k_j &\leftarrow k_j - 1 \\ W_0 &\leftarrow W_0 - w_0 \end{aligned}$$

where, w_0 is the weight of the edges from v to all the vertices which were not in any of the S_j's.

Step 2

After the first step we have, n mutually exclusive sets such that the conflict amongst them is more than the expectation value. Now, we simply need to separate these k elements in each of the n sets so as to give us the desired output. The sets are S_1, S_2, $\cdots$, S_n. Let L_i = total weight of edges from S_i to (S_1, S_2, $\cdots$, S_{i-1}). When S_i is inserted into V_1, V_2, $\cdots$, V_k, the total weight of the conflicting edges is at most $\frac{L_i}{k}$. Recall, the matching argument, when we have k matchings then there is always a matching which is less than or equal to average of all the matchings. Similarly, when we repeat this argument over L_i's, the total conflict weight after all S_i are inserted is at most

$$\sum_{i=2}^{n} \frac{L_i}{k}$$

From the first step we obtain the inequality,i.e.- the sum of the weights (conflicts) within the sets S_i's would be more than or equal to the expectation value, i.e. - $W(\frac{k-1}{kn-1})$, and the conflict weight lost across the sets would be at most $W(1 - \frac{k-1}{kn-1})\frac{1}{k}$. Hence the total weight saved both within and across the sets would be

$$W\frac{k-1}{kn-1} + W(1 - \frac{k-1}{kn-1})(1 - \frac{1}{k})$$

So, the performance ratio would be the total weight saved divided by the total weight of the edges, W, i.e. -

$$\frac{k-1}{kn-1} + (1 - \frac{k-1}{kn-1})(1 - \frac{1}{k})$$

According to the theorem proved by Gaur et al. they found that the per-

formance ratio $'\phi'$ according to their algorithm is greater than or equal to $1 - \frac{1}{k+(k-1)/(n-1)}$, i.e.

$$\begin{aligned} \phi &\geq 1 - \frac{1}{k + (k-1)/(n-1)} \\ &= \frac{k-1}{k - \frac{1}{n}} \end{aligned}$$

Comparison of the two performance ratio

The performance ratio according to our algorithm gives us the performance ratio as

$$\begin{aligned} & (\frac{k-1}{kn-1}) + (1 - \frac{k-1}{kn-1})(1 - \frac{1}{k}) \\ = \; & (\frac{k-1}{kn-1}) + 1 - \frac{1}{k} - (\frac{k-1}{kn-1}) + \frac{1}{k}(\frac{k-1}{kn-1}) \\ = \; & 1 - \frac{1}{k}(1 - \frac{k-1}{kn-1}) \\ = \; & 1 - \frac{n-1}{kn-1} \end{aligned}$$

The performance ratio according to Gaur et al. had a lower bound equal to

$$\begin{aligned} & 1 - \frac{1}{k + (k-1)/(n-1)} \\ = \; & \frac{k-1}{k - \frac{1}{n}} \\ = \; & \frac{k - \frac{1}{n} + \frac{1}{n} - 1}{k - \frac{1}{n}} \\ = \; & 1 - \frac{1 - \frac{1}{n}}{k - \frac{1}{n}} \\ = \; & 1 - \frac{n-1}{kn-1} \end{aligned}$$

Hence, the ratio calculated by our approach is the same compared to the algorithm discussed by Gaur et al.

Running time

Let us now discuss the running time of our algorithm. Our algorithm has two major steps, *step* 1 and *step* 2. So the total time taken in the first step and the second step, would give us the total running time of our algorithm. We will use the Adjacency-list representation of the graphs. In the *step* 1, calculating the ***expectation value*** time requires the sum of all the edges in the graph, which requires linear time of the order of the number of the edges i.e. $O(|E|)$. Then the next step is calculation of B_j for each $j \leq n$. For B_l, we need the sum of the edge weights coming to a particular set S_j, which we have initially. For the first calculation we need some constant time, but for the calculation of other B_j's for any v can be done in constant time (once the w_j's have been calculated), as it will require addition and subtraction of some of the terms to get the value for the B_j. The calculation of the v_j's will take time proportional to the order of the vertices, i.e. the degree of v. So, the total time taken for calculating all the w_j will be $O(|E|)$. Hence the time needed to calculate the *conditional expectation value* would be a constant. But we need to do this for all j's so identifying the set where we need to put in the $i+1$th element will require $O(n)$. We repeat this step for all the vertices, hence the order of the *step* 1 will be $O(|V|)$ time $O(n)$. Hence the total time taken in the *step* 1 would be $O(|V|n + |E|)$, i.e. quadratic in the number of advertisements. In *step* 2 we need to find k mapping from each setS_i having k elements to sets V_1, V_2, $\cdots$, V_k. For this we need to have one pass aver the adjacency list of all the vertices, hence the time required would be $|E|$ for all the vertices, i.e. $O(|V| + |E|)$. Hence the total running time would be quadratic in the size of m.

A Linear Time Algorithm

Suppose if we skip the first step of our algorithm and we arbitrarily form n sets with k elements. Then the performance ration would be simply -

$$\phi \geq \frac{W(1-\frac{1}{k})}{W}$$

$$= \quad (1 - \frac{1}{k})$$

and from the above analysis the running time of the second step is $O(|V| + |E|)$. Hence, we can use this linear time algorithm to solve this problem with just a little worse performance ratio. The performance ratio has k in the denominator instead of $k - \frac{1}{n}$. Hence, we get a much more efficient algorithm with just a little smaller performance ratio.

6.4 Conclusion

The algorithm described in the paper by Gaur et al. uses a procedure to partition the set of kn advertisements into k sets of size n which has a running time which is a polynomial time running algorithm, whereas our algorithm is a quadratic time in the size of the input and gives us the same optimal performance ratio.Hence we could say that our algorithm is more efficient compared to the one designed by Gaur et al. [2] with the same performance ratio.

References

[1] Srinivas Bollapragada and Marc Garbiras. Scheduling commercials on broadcast television. *Operations Research*, **52, No. 3**:337–345, May-June 2004. 36, 37

[2] Ramesh Krishnamurthy Daya Ram Gaur and Rajiv Kohli. Conflict resolution in the scheduling of television commercial. *Operations Research*, **57, No. 5**:1098–1105, September-October 2009. 35, 36, 38, 46

[3] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979. 33

[4] Daya Ram Gaur and Ramesh Krishnamurthy. *LP rounding and extensions*. 34

[5] Daya Ram Gaur and Ramesh Krishnamurthy. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer, 2003. 34

[6] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Pearson Education, Inc., 2006. vii, 3, 4, 6, 7, 13, 14, 25, 27, 28

[7] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge, 2005. 33

[8] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge, 1995. 33

[9] Balaji Raghavachari Samir Khuller and Neal E. Young. 33

[10] Elchanan Mossel Subhash Khot, Guy Kindler and Ryan ODonnell. Optimal inapproximability results for max-cut and other 2-variable csps? *Electronic Colloquium on Computational Complexity*, **Report No. 101**, 2005. 34

[11] Ronald Rivest Thomas H. Cormen, Charles E. leiserson and Clifford Stein. *Introduction to Algorithms*. McGraw Hill Book Company, 2001. vii, 17, 19, 21, 22, 23