Some introductory notes on Design and Analysis of Algorithms

Venkatesh Raman
The Institute of Mathematical Sciences
C. I. T. Campus
Chennai - 600 113.
email: vraman@imsc.res.in

1 Introduction to Algorithm Design and Analysis

1.1 Introduction

An algorithm is a receipe or a systematic method containing a sequence of instructions to solve a computational problem. It takes some inputs, performs a well defined sequence of steps, and produces some output. Once we design an algorithm, we need to know how well it performs on any input. In particular we would like to know whether there are better algorithms for the problem. An answer to this first demands a way to analyze an algorithm in a machine-independent way. Algorithm design and analysis form a central theme in computer science.

We illustrate various tools required for algorithm design and analysis through some examples.

1.2 Select

Consider the problem of finding the smallest element from a given list A of n integers. Assuming that the elements are stored in an array A, the pseudocode[1] *Select*$(A[1..n])$ below returns the smallest element in the array locations 1 to n.

```
function Select(A[1..n])
begin
  Min := A[1]
  for i=2 to n do
    if A[i] < Min then Min := A[i]
  endfor
  return (Min)
end
```

It is easy to see that finally the variable Min will contain the smallest element in the list.

Now let us compute the number of steps taken by this algorithm on a list of n elements in the worst case. The algorithm mainly performs two operations: a comparison (of the type $A[i] < Min$) and a move (of the type $Min := A[i]$). It is easy to see that the algorithm performs $n-1$ comparisons and *at most* n moves.

Remarks:

1. If we replace '$Min := A[i]$' by $Min := i$ and $A[i] < Min$ by $A[i] < A[Min]$ after replacing the initial statement to $Min := 1$ and finally return $A[Min]$, the algorithm will still be correct. Now we save on the kind of moves $Min := A[i]$ which may be expensive in reality if the data stored in each location is huge.

2. Someone tells you that she has an algorithm to select the smallest element that uses at most $n-2$ comparisons always. Can we believe her?

   No. Here is why. Take any input of n integers and run her algorithm. Draw a graph with n vertices representing the given n array locations as follows. Whenever algorithm

[1] A pseudocode gives a language-independent description of an algorithm. It will have all the essence of the method without worrying about the syntax and declarations of the language. It can be easily converted into a program in any of your favourite language.

detects that $A[i] < A[j]$ for some pair of locations i and j, draw a directed edge $i \leftarrow j$ between i and j. Now at the end of the execution of the algorithm on the given input, we will have a directed graph with at most $n-2$ edges. Since the graph has at most $n-2$ edges, the underlying undirected graph is disconnected. Due to the transitivity of the integers, each connected component of the graph does not have a directed cycle. Hence each component has a sink vertex, a vertex with outdegree 0. Each such vertex corresponds to a location having the smallest element in its component. Note also that the algorithm has detected no relation between elements in these sink locations (otherwise there will be edges between them). Let x and y be integers in two such locations. If the algorithm does not output x or y as the answer, then the algorithm is obviously wrong as these are the smallest elements in their components. If the algorithm outputs x as the answer, we decrease the value of y to some arbitrarily small number less than x. Even now the execution of the algorithm would be along the same path with the same output x which is a wrong answer. In a similar way, we can prove the algorithm wrong even if it outputs y as the answer.

An alternate way to see this is that an element has to win a comparison (i.e. must be larger than the other element) before it can be out of consideration for the smallest. In each comparison, at most one element can win, and to have the final answer, $n-1$ elements must have won comparisons. So $n-1$ comparisons are necessary to find the smallest element.

This fact also indicates that proving a lower bound for a problem is usually a nontrivial task. This problem is one of the *very few* problems for which such an exact lower bound is known.

1.3 Selectionsort

Now suppose we want to sort in increasing order, the elements of the array A. Consider the following pseudocode which performs that task. The algorithm first finds the smallest element and places it in the first location. Then repeatedly selects the smallest element from the remaining elements and places it in the first remaining location. This process is repeated $n-1$ times at which point the list is sorted.

```
Procedure Selectsort(A[1..n])
begin
  for i=1 to n-1 do
      Min := i
      for j= i+1 to n do
        if A[j] < A[Min] then Min := j
      endfor
      swap (A[i], A[min])
  endfor
end
```

Here *swap*(x, y) is a procedure which simply swaps the two elements x and y and which can be implemented as follows.

```
Procedure swap(x,y)
```

```
begin
  temp := x
  x := y
  y := temp
end
```

Note that the inner for loop of the above sorting procedure is simply the *Select* procedure outlined above. So the code can be compactly written as

```
Procedure Selectsort(A[1..n])
begin
 for i=1 to n-1 do
   min := Select(A[i,n])
   swap(A[i], A[min])
 endfor
end
```

Now let us analyse the complexity of the algorithm in the worst case.

It is easy to see that the number of comparisons performed by the above sorting algorithm is $\sum_{i=1}^{n-1}(n-i)$ as a call to $Select(A[i,n])$ takes $n-i$ comparisons. Hence the number of comparisons made by the above algorithm is $n(n-1)/2$. It is also easy to see that the number of moves made by the algorithm is $O(n)$ [2]. Thus Selectsort is an $O(n^2)$ algorithm.

1.4 Merge

In this section, we look at another related problem, the problem of merging two sorted lists to produce a sorted list. You are given two arrays $A[1..n]$ and $B[1..n]$ of size n each, each containing a sequence of n integers sorted in increasing order. The problem is to merge them to produce a sorted list of size $2n$.

Of course, we can use the Selectsort procedure above to sort the entire sequence of $2n$ elements in $O(n^2)$ steps. But the goal is to do better using the fact that the sequence of elements in each array is in sorted order. The following procedure merges the two arrays $A[1..n]$ and $B[1..n]$ and produces the sorted list in the array $C[1..2n]$.

```
Procedure Merge(A[1..n], B[1..n], C[1..2n])
begin
 i := 1; j := 1; k := 1;
 while i <= n and j <= n do
    if A[i] < B[j] then
        C[k] := A[i]; i := i+1
    else
        C[k] := B[j]; j := j+1
    endif
    k := k+1
 endwhile
 if i > n then
     while j <= n do
          C[k] := B[j]; k := k+1; j := j+1
```

[2] A function $f(n)$ is $O(g(n))$ if there exists constants c and N such that $f(n) \leq cg(n)$ for all $n \geq N$

```
            endwhile
        else
            while i <= n do
                C[k] := A[i]; k := k+1; i := i+1
            endwhile
        endif
end
```

It is easy to see that finally the array $C[1..2n]$ contains the sorted list of elements from A and B.

It is also easy to see that the algorithm performs at most $2n-1$ comparisons and $2n$ moves. Can you find out when the algorithm will actually perform $2n-1$ comparisons?

Recall that initially we were toying around with an $O(n^2)$ algorithm (Selectsort) and now we have managed to complete the task in $O(n)$ time simply using the fact that each list is sorted already.

1.5 Mergesort

The following algorithm gives another method to sort an array of n integers. This method uses the Merge routine described above. The algorithm calls itself several times (with different arguments) and hence it is a recursive algorithm. Contrast with *Selectsort* which called some other algorithm.

Recursion is a powerful tool both in algorithm design and also in algorithm description. As we will see later, the following recursive algorithm is significantly better than the *Selectsort* algorithm described earlier.

```
Procedure Mergesort(A[1..n])
begin
if n <> 1 then
    Mergesort(A[1..n div 2])
    Mergesort(A[n div 2 + 1.. n])
    Merge(A[1..n div 2], A[n div 2 +1.. n], B[1..n])
    for i=1 to n do A[i] := B[i]
endif
```

The algorithm basically divides the given array into two halves. Recursively sorts the two halves and then uses the Merge routine to complete the sort. The Merge routine actually produces the merged list into a different array B. Hence in the last statement, we copy back the elements of B into the array A.

This method is also an example of the classic 'divide and conquer' algorithm design technique. The idea here is to divide the problem into many parts, solve each part separately and then combine the solutions of the many parts.

Note that though the actual execution of the recursive calls go through many more recursive calls, we don't have to worry about them in the description. Thus a recursive program usually has a compact description. Note also that the first step in the program is the end (or tail) condition. It is important to have tail condition in a recursive program. Otherwise the program can get into an infinite loop.

Now let us get into the analysis of the algorithm. Again for simplicity let us count the number of comparisons made by the algorithm in the worst case. Note that the recursion introduces some complication as we cannot do the analysis as we did so far. But it is easy to write what is called a *recurrence relation* for the number of comparisons performed. Let $T(n)$ be the number of comparisons made by the algorithm. Then it is easy to see that

$$T(1) = 0$$

and

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n - 1, \ \textit{for } n \geq 2$$

since the Merge routine performs at most $n-1$ comparisons to merge two lists one of size $\lfloor n/2 \rfloor$ and the other of size $\lceil n/2 \rceil$ (Note that the way we described, the Merge routine merges two lists each of size n using at most $2n - 1$ comparisons. But it can be easily modified to merge a list of size m and one of size n using at most $m + n - 1$ comparisons.)

Now, how do we solve such a recurrence relation? The study of recurrence relations forms a major part in the analysis of algorithms. Note that, for now, we are not interested in solving the recurrence relation exactly. We will be happy with a tight enough estimate which will help us compare with *Selectsort*. To solve the recurrence relation, let us first assume that $n = 2^k$ for some integer k, i.e. n is an exact power of 2. Then happily, we will have

$$T(n) = 2T(n/2) + n - 1, \ \textit{for } n \geq 2.$$

Now expand the recurrence relation using the fact that $T(n/2) = 0$ if $n = 2$ or it is $2T(n/4) + n/2 - 1$ otherwise. If we keep expanding the relation all the way until we get the term in the right hand side to be $T(1)$ for which we know the answer, we will have

$$T(n) = (n-1) + (n-2) + (n-4) + ...(n - 2^{k-1})$$

Since $k = \log_2 n$ [3], we have $T(n) = nk - n + 1 = n \log n - n + 1$.

What if n is not a power of 2? Let $2^k < n < 2^{k+1}$. Using our *Select* routine, first find the largest element in the given array. Then add $2^{k+1} - n$ dummy elements equal to a number larger than the largest to the array make the number of elements in the array 2^{k+1}. Now we apply the Mergesort routine to this array. Clearly the first n elements of the sorted list form the required sorted output.

Now from our earlier analysis, we have that the algorithm performs at most $2^{k+1}(k + 1) - 2^{k+1} + 1 + n - 1$ comparisons (the $n - 1$ comparisons are to find initially the largest element in the list). Now since $k < \log n$, we have that the algorithm performs at most $2n(\log n + 1) - n + 1 + n - 1$ which is $2n(\log n + 1)$ comparisons. So Mergesort is an $O(n \log n)$ algorithm.

Recall that *Selectsort* performed $n(n-1)/2$ comparisons in the worst case. It is easy to see that *Mergesort* outperforms *Selectsort* for $n \geq 32$. You should note that *Selectsort* performs significantly fewer moves than *Mergesort*. Also *Mergesort* uses extra storage space of up to n locations (for the B array) whereas *Selectsort* uses very few extra variables. Despite all this, *Mergesort* performs significantly better compared to *Selectsort*.

It is also known that any sorting algorithm by comparisons must perform at least $n \log n$ comparisons in the worst case. So Mergesort is, in some sense, an optimal sorting algorithm.

Exercise 1:

[3]hereafter we will omit the base 2 when we talk about $\log n$ and assume that all logarithms are to the base 2

1. Given an array of n integers, suppose you want to find the largest as well as the smallest in the array. Give an algorithm (a simple pseudocode) to perform this task. How many comparisons does your algorithm use in the worst case?

   [There is a simple divide and conquer algorithm that performs $3\lceil n/2 \rceil - 1$ comparsons in the worst case. But as a warm up, you may want to start with a simpler algorithm which could perform more comparisons.]

2. Given a sorted array A of integers and an integer x, suppose you want to search whether x is in the given array.

   (a) One method (which does not use the fact that the given array is sorted) is to simply scan the entire array checking whether any element is equal to x. Write a simple pseudocode for this method. How many comparisons does your algorithm use?

   (b) Another popular method called binary search works as follows.

      Compare x with the middle element $A[n\ div\ 2]$. If $x \leq A[n\ div\ 2]$ then recursively search in the first half of the array. Otherwise recursively search for x in the second half of the array.

      Write a proper pseudocode (with tail conditions etc) for this method, and find the number of comparisons performed by this algorithm. Can you modify the algorithm into a non-recursive one?

1.6 Summary

In this section, we have dealt with various algorithm design and analysis tools through some simple but important data processing problems: Selecting the largest element from a list, Sorting a list and Merging two sorted lists. Through these examples, we introduced the 'big Oh' (O) notation, recursive algorithms, divide and conquer algorithm design technique, recurrence relations, algorithm analysis and lower bounds.

2 Introduction to Data Structures

The way in which the data used by the algorithm is organized will have an effect on how the algorithm performs. In some algorithms a certain kind of operations on the input are more frequently used. So a clever method to organize the data to support those operations efficiently will result in the overall improvement of the algorithms. What is the efficient way to organize the data to support a given set of operatons is the question addressed in the study of Data Structures.

2.1 Arrays and Linked Lists

Throughout the last section, we assumed that the input is an *array* where the elements are in contiguous locations and any location can be accessed in constant time. Sometimes it may be an inconvenience that the elements have to be in contiguous locations. Consider the binary search example in the last section. Suppose you discover that the given element x is not in the sorted array, and you want to *insert* it in the array. Suppose you also find that x has to be in the fourth location to maintain the sorted order. Then you have to move all elements from the fourth location onwards one position to their right to make room for x. It would have been nice if we can simply grab an arbitrary empty cell of the array, insert the new element and somehow logically specify that that element comes after the third location and before the present fourth location. A *linked list* has the provision to specify next and previous elements in such a way.

A *linked list* is a data structure in which the objects are arranged in a linear order. Unlike in an array where the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object. So each element x of a linked list has two *fields*, *a data* field $x.key$ and *a pointer* field $x.next$. The pointer field points to the location containing (i.e. gives the address of) the next element in the linear order. The *next* pointer of the last element is NIL. An attribute $head[L]$ points to the first element of the list L. If $head[L] = NIL$ then the list is empty.

The following psuedocode gives a procedure to search for an element x in a given linked list L.

```
Procedure listsearch(L,x)
begin
    p := head[L]
    while p <> NIL and p.key <> x
      p := p.next
    endwhile
    if p.key = x return(p)
    else return('x not found')
end
```

The following psuedocode gives a procedure to insert an element x not present in the given sorted list of elements arranged in a linked list.

```
Procedure sortedlistinsert(L,x)
begin
    p := head[L]
    if p.key >= x then
        new(cell)
        cell.key := x; cell.next := p;
        head[l] := cell
    else
     while p <> NIL and p.key < x
       q := p
       p := p.next
     endwhile
     new(cell)
     cell.key := x; cell.next := p;
     q.next := cell
    endif
 end
```

It is easy to see that the insert procedure takes only constant time after the position to be inserted is found by the initial scan (the while loop).

Coming back to the binary search example, we observed that if the sorted list is arranged in a linked list then insertion is easy after the position to be inserted is found. But note that doing a binary search in a linked list is significantly harder. This is because the access to any element in the linked list is always through the header $head[L]$. So if we have to search as well as insert, then a more sophisticated data structure is required to perform these operations in $O(\log n)$ time. We will see more about this later.

With each element x in the list, we could also have another pointer $x.prev$ pointing to the location of the previous element. Such a list is called doubly linked list. If we have previous pointers available, we need not have used the variable q in the above code. We could have obtained the final q using $p.prev$.

2.1.1 Graph Representations

In this section we will give two representations to represent a graph and discuss their advantages and disadvantages for various problems.

Adjacency Matrix

The standard way to represent a graph G is by its adjacency matrix A. If G has n vertices 1 to n, then A is an n by n matrix with $A[i, j] = 1$ if the vertex i and j are adjacent and 0 otherwise. If there are m edges in the graph, then the adjacency matrix will have 1 in $2m$ locations.

Adjacency matrix can be represented easily as a two dimensional array as an array $[1..n]$ of arrays $[1..n]$ of integers or equivalently as an array $[1..n, 1..n]$ of integers. We can access $A_{i,j}$ by calling $A[i, j]$.

Adjacency List

In an adjacency list, we simply have an array A[1..n] of linked lists. The linked list $A[i]$ has a list of vertices adjacent to the vertex i, arranged in some order (say, increasing order).

Essentially each linked list in an adjacency list corresponds to the list of entries having 1 in the corresponding row of the adjacency matrix.

There are several graph operations for which an adjacency matrix representation of a graph is better and there are some for which an adjacency list representation is better. For example, suppose we want to know whether vertex i is adjacent to vertex j. In an adjacency matrix, this can be performed in constant time by simply probing $A[i, j]$. In an adjacency list, however, we have to scan the i-th list to see whether vertex j is present or scan the j-th list to see whether vertex i is present. This could potentially take $O(n)$ time (if for example, the graph is $n/2$-regular).

On the other hand, suppose you want to find all the neighbours of a vertex in a graph. In an adjacency matrix, this takes $O(n)$ time whereas in an adjacency list this takes time proportional to the degree of that vertex. This could amount to a big difference in a sparse graph (where there are very few edges) especially when the neighbours of all vertices have to be traversed. We will see more examples of the use of adjacency list over an adjacency matrix later as well as in the exercises below.

Exercise 2.1:

1. It is well known that a graph is Eulerian if and only if every vertex has even degree. Give an algorithm to verify whether a given graph on n vertices and m edges is Eulerian. How much time does your algorithm take in the worst case if (a) the graph is given by its adjacency matrix and (b) graph is given by its adjacency list?

2. A Directed graph can be represented by an adjacency matrix A in a similar way:

$$A[i,j] \begin{array}{l} = 1 \text{ if there is an edge from } i \text{ to } j \\ = -1 \text{ if there is an edge from } j \text{ to } i \text{ and} \\ = 0 \text{ if there is no edge between } i \text{ and } j. \end{array}$$

Tournaments form a special class of directed graphs where there is a directed edge between every pair of vertices. So we can modify the definition of an adjacency matrix of a tournament as

$$A[i,j] \begin{array}{l} = 1 \text{ if there is an edge from } i \text{ to } j \\ = 0 \text{ if there is an edge from } j \text{ to } i \end{array}$$

In a very similar way, we can also represent a tournament by an adjacency list as an array of linked lists where list i contains all vertices j to which there is a directed edge from i.

Given a tournament, your problem is to find a hamiltonian path (a directed path passing through all vertices exactly once) in the tournament by the following three algorithms.

For each of the three algorithms, write a psuedocode and analyse the running time of your algorithm assuming (a) the tournament is represented by an adjacency matrix as above and (b) the tournament is represented by an adjacency list.

(a) (By induction and linear search) Remove any vertex x of the tournament. Find a hamiltonian path P in the resulting tournament recursively. Find, in P, two consecutive vertices i and j such that there is an edge from i to x and from x to j by finding the edge direction between x and every vertex of P. Insert x in P between i and j to obtain a hamiltonian path in the original graph.

(b) (By induction and binary search) The algorithm is the same as above except to find the consecutive vertices i and j, try to do a binary search.

(c) (By divide and conquer technique) It is known that every tournament on n vertices has a *mediocre* vertex, a vertex whose outdegree and indegree are at least $n/4$.

Find a mediocre vertex v in the given tournament (use any naive and simple method for this). Find the set of vertices I from which there is an edge to v and the set of vertices O to which there is an edge from v. I and O individually induce a tournament. Find hamiltonian path P and Q recursively in I and in O respectively. Insert v between P and Q to obtain a hamiltonian path in the given tournament.

How much time does your algorithm take if a mediocre vertex can be found in a tournament in $O(n)$ time?

2.2 Trees

In a linked list, we saw that though the actual memory locations may be scattered, pointers can be used to specify a linear order among them. It turns out that using pointers one can represent structures that form more than a linear order.

A binary tree, or more specifically a rooted binary tree is another fundamental and useful data structure. A binary tree T has a $root[T]$. Each node or an element x in the tree has several fields including the data field $key[x]$ along with several pointers. The pointer $p[x]$ refers to the parent of the node x ($p[x]$ is NIL for the root node), $l[x]$ is a pointer to the left child of the node x, and similarly $r[x]$ is a pointer to the right child of the node x. If node x

has no left child then $l[x]$ is NIL and similarly for the right child. If $root[T]$ is NIL then the tree is empty.

There are also several ways of representing a rooted tree where each node can have more than two children. Since we will not be needing them in these lectures, we won't deal with them here.

2.3 Stacks and Queues

From the fundamental data structures arrays and linked list, one can build more sophisticated data structures to represent the input based on the operations to be performed on the input. Stacks and Queues are data structures to represent a set of elements where elements can be inserted or deleted but the location at which an element is inserted or deleted is prespecified.

Stack: In a stack, the element to be deleted is the one which is recently inserted. Insert into a stack is often called a *push* operation and deletion from a stack is called a *pop* operation. Essentially the push and the pop operation are performed at one end of the stack. The stack implements the last-in-first-out or LIFO policy.

Queue: In a queue, the element to be deleted is the one which has been in the set for the longest time. Insert into a queue is often called an *enqueue* operation and deletion from a queue is called a *dequeue* operation. If the enqueue operation is performed at one of the queue the dequeue operation is performed at the other end. The queue implements the first-in-first-out or FIFO policy.

Implementation of Queues and Stacks: It is easy to implement Queues and Stacks using arrays or linked list. To implement a stack, we will use a special variable to keep track of the top of the list. During a push operation, the top pointer is incremented and the new element is added. Similarly during a pop operation, the element pointed to by the top pointer is deleted and the top pointer is decremented.

Similarly to implement a queue, we need to keep track the head and tail of the list. During an enqueue operation, the tail pointer is incremented and the new element is inserted. During a dequeue operation, the element pointed to by the head pointer is deleted and the head pointer is decremented.

2.4 Binary Search trees and Heaps

The structures we have defined so far simply give an implementation of a specific arrangement of memory locations. In a typical usage of these structures, there is also some order among the data fields of the elements in those locations. Binary Search Trees and Heaps are binary trees where the data fields of the nodes in the binary tree satisfy some specific properties.

2.4.1 Binary Search Trees

A binary search tree is a binary tree where each node x has a key $key[x]$ with the property that $key[x]$ is larger than or equal to $key[y]$ for all nodes y in the left subtree of x, and less than or equal to $key[z]$ for all nodes z in the right subtree of x. There is a pointer to the root of the binary search tree.

Suppose all the given n integers are arranged in a binary search tree, there is a clever way to search for an element y. We compare with y with the $key[root]$. If $y = key[root]$ we stop. Otherwise if $y < key[root]$ then we can abandon searching in the right subtree of root and search only in the left subtree and similarly if $y > key[root]$. This procedure can be applied recursively for the appropriate subtree of the root and the entire search will involve traversing a root to leaf path. If we reach a NIL pointer without finding y, then y is not in the list. So the time taken by the above algorithm is proportional to a root to leaf path. This is the height of the tree in the worst case.

If the binary tree is a complete binary tree where the tree is filled on all levels except possibly the lowest, which is filled from the left up to a point, then it is easy to see that such a binary tree on n nodes has height $O(\log n)$. So if the given n elements are arranged in a binary search tree on a complete binary tree, then searching for an element y in that list takes $O(\log n)$ time. In fact, if we reach a NIL pointer without finding y, the NIL pointer specifies the location where y has to be if it were there in the list. So we can simply insert y in that location (updating appropriate pointers) as we would insert an element in a linked list. So insert also takes $O(\log n)$ time in such a binary search tree.

There is one catch. Suppose we start from a binary search tree where the underlying binary tree is complete and keep inserting elements. After a few insertions, the tree will no longer remain complete and so height may no longer be $O(\log n)$. The solution is to keep the tree 'balanced' after every insertion. There are several balanced binary search trees in the literature – implementations of insert or delete on these trees involve more operations to keep the tree balanced after a simple insert or delete.

2.4.2 Heaps

A (min) heap is a complete binary tree (where all levels of the tree are filled except possibly the bottom which is filled from the left up to a point) where each node x has a key $key[x]$ with the property that $key[x]$ is less than or equal to the key value of its children. There is a pointer to the root as well as the first empty (NIL) location at the bottom level of the heap.

It follows that the root of the heap contains the smallest element of the list. The key values in the tree are said to satisfy the heap order.

Since a heap is stored in a complete binary tree, the key values of all the nodes can be stored in a simple array by reading the key values level by level.

To insert an element into a heap, insert it at the first empty location at the bottom level. Compare it with the parent and swap it with the parent if it doesn't satisfy the heap property with its parent. Then this procedure is recursively applied to the parent until the heap property is satisfied with the node's parent. The execution simply involves traversing a root to leaf path and hence takes $O(\log n)$ time since the height of the tree is $O(\log n)$. To delete the minimum element from the heap, simply delete the root and replace it with the element at the last nonempty node of the bottom level. Also the pointer to first empty location at the bottom level is pointed to this node. Now there may be a heap order violation at the root. Again this is fixed by traversing down the tree up to a leaf. This process also takes $O(\log n)$ time.

Thus heap is also an efficient data structure to support a priority queue where elements are inserted and only the smallest element is deleted. We will come across graph problems where the only operations performed on the data list are to insert an integer into the list and to delete the smallest element in the list. For such applications a heap will be a useful data structure to represent the data list.

Exercise 2.2:

1. Given a binary tree, there are various ways in which we can traverse all nodes of the tree.

   A *preorder traversal* of a binary tree visits the root, followed by a preorder traversal of the left subtree of the root, followed by a preorder traversal of the right subtree.

   An *inorder traversal* of a binary tree recursively performs an inorder traversal of the left subtree of the root, followed by the root, followed by an inorder traversal of the right subtree of the root.

   A *postorder traversal* of a binary tree recursively performs a postorder traversal of the left subtree of the root, followed by a postorder traversal of the right subtree of the root, followed by the root.

   Write psuedocodes for each of the three traversals of a binary tree (with proper end condition etc), and analyse their running time.

2. Given n integers arranged in a binary search tree, how can we produce the integers in sorted order in $O(n)$ time?

3. Given an array of n integers, it can be considered as the key values of a complete binary tree read level by level from left to right. Such an arrangement in a complete binary tree can be made into a heap as follows:

   > Starting from the bottom level to the root level, make all subtrees rooted at every node in that level a heap.

   The subtree rooted at a node x in a level can be made into a heap as follows. Both subtrees of x form heaps by themselves. If $key[x]$ is smaller than or equal to the key values of its children, then the subtree rooted at x is already a heap. Otherwise, swap $key[x]$ with the key value of the smaller child and continue down this process from the node containing the smaller child.

   Write a proper psuedocode for this algorithm. Prove that this algorithm takes $O(n)$ time in the worst case. (Note that in the array representation of the complete binary tree, the left child of a node at position x is at position $2x$ and the right child at position $2x + 1$. Similarly the parent of a node at position x is at position $\lfloor x/2 \rfloor$.)

4. Give an $O(\log n)$ algorithm to perform the following operations on a heap.

   (a) To delete an arbitrary element specified by its location in the tree.

   (b) To replace the value of an element by a smaller value.

5. Given an array of n integers, give an algorithm to sort them into increasing order using the heap data structure in $O(n \log n)$ time. (Hint: In the first step, convert the array into a representation of a heap in $O(n)$ time using the previous exercise. The smallest element will be in the first location. Then if you perform some heap operation n times, you get a sorted list.)

2.5 Summary

In this section, we had an introduction to fundamental data structures like arrays, linked lists, trees, stacks, queues, heaps and binary search trees. In graph representations, we saw an application of arrays versus linked lists. We will see applications of the other data structures in subsequent sections.

3 Graph Search Algorithms

In this section, we will look at algorithms for several fundamental problems on graphs. In the initial step of these algorithms, the vertices and edges of the graph are traversed in a specific way. There are two basic methods to systematically visit the vertices and edges of a graph: breadth-first-search and depth-first-search. We begin with the description of these methods.

3.1 Breadth First Search

Here, we visit the vertices of the graph in a breadthwise fashion. The idea is to start from a vertex, and then visit all its neighbours and then visit the neighbours of its neighbours and so on in a systematic way. A queue will be an ideal data structure to record the neighbours of a vertex (at the other end) once a vertex is visited.

See pages 469-472 of [3] for a more detailed description.

Note that the way the BFS is described in those pages, the search will visit only vertices reachable from the vertex s. If the graph is disconnected, then one can pick an arbitrary vertex from another component (i.e. a vertex which is still coloured white at the end of BFS(s)) and grow a BFS starting at that vertex. If this process is repeated until no vertex is coloured white, the search would have visited all the vertices and edges of the graph.

3.1.1 Applications

The following properties of the BFS algorithm are useful.

1. For vertices v reachable from s, $d[v]$ gives the length of the shortest path from s to v. Furthermore, a shortest path from s to v can be obtained by extending a shortest path from s to $\pi[v]$ using the edge $(\pi[v], v)$.

   Though this result is quite intuitive from the way a BFS algorithm works, it can be proved rigorously using induction.

2. For a graph $G = (V, E)$ with source s, the predecessor graph $G_\pi = (V_\pi, E_\pi)$ defined by $V_\pi = \{v \in V : \pi[v] \neq NIL\} \cup \{s\}$ and $E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$ is a tree called the *breadth-first tree*. Furthermore, for all vertices reachable from s, and for all vertices $v \in V_\pi$, the unique simple path from s to v in G_π is a shortest path from s to v in G.

3. If $(i, j) \in E$ is not an edge in the breadth first tree, then either $d[i] = d[j]$ or $d[i] = d[j] + 1$ or $d[i] = d[j] - 1$.

Connectivity: To verify whether the given graph G is connected, simply grow a BFS starting at any vertex s. If any vertex is still coloured white after $BFS(s)$, then G is disconnected. Otherwise G is connected. Note that by growing a BFS starting at white vertices, we can obtain also the connected components of the graph. Thus in $O(n+m)$ time it can be verified whether a given graph on n vertices and m edges is connected and if it is not, the connected components of the graph can also be obtained in the same time.

Shortest Paths: As we have observed in properties 1 and 2 above, the length of the shortest path[4] from a given vertex s can be obtained simply from a BFS search. In addition, BFS also outputs a breadth-first tree from which one can also obtain a shortest path from s to every vertex reachable from s.

Spanning Forest: To find a spanning tree of a connected graph, simply perform a breadth-first search starting at any vertex. The breadth first tree gives a spanning tree of the graph if the graph is connected. If the graph is disconnected, we obtain a breadth-first forest which is a spanning forest.

Exercise 3.1:

1. What is the running time of BFS if its input graph is represented by an adjacency matrix and the algorithm is modified to handle this form of input?
2. Give an efficient algorithm to determine if an undirected graph is bipartite. (Hint: Property 3 discussed above may be useful here.)
3. Give an efficient algorithm to find the shortest cycle in the graph. How much time does your algorithm take?

3.2 Depth First Search

Here the idea is to visit the vertices in a depthwise fashion. Starting from a vertex, we first visit ONE of its neighbours, then one of ITS neighbours and so on until the current vertex has no unvisited neighbours. At that point, we backtrack and continue visiting the next neighbour of the earlier vertex and so on. A stack (Last in First out) is a good data structure to record the neighbours of the visted vertices by this procedure. See pages 477-479 of [3] for more details and some applications of DFS (pages 485-487).

One of the early applications of DFS is to obtain strongly connected components of a graph. The readers are directed to the books in the References section for details of this.

Exercise 3.2:

1. Give an algorithm that determines whether or not a given undirected graph on n vertices contains a cycle. Your algorithm should run in $O(n)$ time (independent of the number of edges in the graph).
2. Another way to perform topological sorting on a directed acyclic graph on n vertices and m edges is to repeatedly find a vertex of in-degree 0, output it, and remove it all of its outgoing edges from the graph. Explain how to implement this idea so that the algorithm runs in $O(n+m)$ time. What happens to this algorithm if the graph has cycles?

[4] Here, we assume that the edges of the graph are unweighted. The shortest path problem on weighted graphs is discussed in the next section.

3.3 Summary

In this section, we looked at the basic search procedures in a graph, Breadth First Search (BFS) and Depth First Search (DFS). We saw applications of the data structures Stacks and Queues in these procedures. As applications of BFS and DFS, we described algorithms to test whether a given graph is connected, and if so to find the spanning tree of the graph; to find the connected components and/or the spanning forest of the graph if the graph is disconnected; to find the length of the shortest path from a source vertex to every vertex reachable from it; and to perform a topological sort of a directed acyclic graph. Using BFS and DFS, we observed that these problems can be solved in $O(m+n)$ time on a graph with n vertices and m edges.

4 Efficient Algorithms on Weighted graphs

Graph theory, and graph algorithms in particular, have several real-world applications. In modeling some of these applications as graphs, typically the edges of the graph have some real weights - these could denote distances between two points and/or some cost. The goal is to find some optimum structure (with respect to the weights) in the weighted graph.

For example, in the last section we saw that in a connected graph, a spanning tree as well as the shortests paths to all vertices from a source can be found in $O(m+n)$ time. However, when the edges have weights, the shortest path problem gets a little complicated. Similarly, instead of any spanning tree, a spanning tree with minimum weight (the total weight of all edges in the tree) is of interest in some applications. We will look at algorithms for these problems in this section. Though we will not specifically mention about the representation of the graph, it sufficies to represent it by an adjacency list where the weight of an edge (i,j) is kept with the vertex j in the adjacency list i as well as with the vertex i in the adjacency list j.

4.1 Minimum Spanning Tree

Given an undirected graph with weights on the edges, the problem here is to find, among all spanning trees, one with minimum (total) weight. Note that there may be several spanning trees with the same weight. We give two classical algorithms which follow a 'greedy' approach to find the minimum spanning tree.

Greedy heuristic is a well known heuristic to solve optimization problems. It advocates making the choice that is the best at the moment - i.e. a locally optimum choice. Such a strategy is not generally guaranteed to find globally optimum solution. For the minimum spanning tree problem, however, we can prove that certain greedy heuristics do yield a spanning tree with minimum weight.

4.1.1 Kruskal's algorithm

An obvious greedy strategy is to sort the edges by their weights and to construct a spanning tree by considering edges in that order and adding those that do not form a cycle to the already constructed graph (forest).

This is the essence of Kruskal's algorithm. This algorithm actually produces a minimum spanning tree due to the following reason (which can be easily proved rigorously).

- Let $G=(V,E)$ be a connected undirected graph with each edge having a real valued integer as its weight. Let A be a subset of E that is in some minimum spanning tree for

G. Let C be a connected component (a tree) in the forest $G_A = (V, A)$, and let (u, v) be an edge with minimum weight among those connecting C to some other component in G_A. Then $A \cup \{(u, v)\}$ is included in some minimum spanning tree for G.

The first sorting step can be performed in $O(m \log m)$ time using any of the efficient sorting algorithms described in the first section. After that at any stage we have a forest of edges spread over several components. When a new edge is considered, it is added to the forest only if it joins vertices from two different components of the forest. There are efficient data structures to maintain such a structure (forest) in which one can check whether or not an edge is inside one component in $O(\log m)$ time. Thus the overall algorithm can be implemented in $O(m \log m)$ time.

4.1.2 Prim's algorithm

Kruskal's algorithm constructs a spanning tree starting from a forest of n trees (vertices), by adding at every iteration a minimum weight edge connecting two components. At any stage of the Kruskal's algorithm, there is a forest.

Prim's algorithm, on the other hand, always has a connected component (a tree) C of edges and grows the tree by adding a minimum weight edge connecting C to the rest of the graph. Initially C is a single vertex s from which the tree is grown.

The correctness of Prim's algorithm also follows from the reason mentioned above for the correctness of Kruskal's algorithm.

To implement Kruskal's algorithm, at every stage we need to find a minimum weight edge connecting C to the rest of the graph. Initially this is the edge that has the minimum weight among those incident on the starting vertex s. But at any intermediate stage C has several vertices, and each of those vertices may have several edges incident with it.

Suppose, for every vertex u not in C, we keep track of the weight $d[u]$ (and also the other end point) of the minimum weight edge connecting u to a vertex in C ($d[u]$ will be ∞ if u is not adjacent to any vertex in C). Then the vertex p (and the corresponding edge) with the minimum $d[p]$ among all those vertices not in C is the next vertex (and the corresponding edge) to be included in C according to Kruskal's algorithm. Once the vertex p is included in C, the $d[]$ values of some vertices (those adjacent to p) need to be updated (more specifially decreased). Thus we need a data structure to maintain a set of at most n integers where the operations we will perform at any stage are *a delete the minimum element - deletemin*, and *decrease the value - decreasekey* for some elements. It is easy to see that on a graph on n vertices and m edges, overall there will be $n-1$ deletemin operations and m decreasekey operations (why?).

A heap is an appropriate data structure to implement these operations. As we have already seen a deletemin or a decreasekey operation can be performed in $O(\log n)$ time in a heap. Thus Kruskal's algorithm can be implemented to take $O(m \log n)$ time.

There are more sophisticated variations of the heap (Fibonacci Heap) that takes $O(m)$ time to perform a sequence of m decreasekey operations while performing $O(n \log n)$ time to perform n deletemin operations. Thus Kruskal's algorithm can be made to run in $O(m + n \log n)$ time. Note that for a dense graph where m is, say, $n^2/4$, the former implementation takes $O(n^2 \log n)$ time whereas the latter implementation takes $O(n^2)$ algorithm.

Notice the role played by the data structures in the implementation of algorithms. Both implementations above implement the same algorithm, but improvements are obtained by using more efficient data structures.

4.2 Shortest Paths

As we saw in the last section, the breadth-first-search method gives the shortest path from a source vertex in an unweighted graph (we could view the weight of each edge to be unit). There the length of a path is simply the number of edges in the path. In a weighted graph, however, the length of a path is the sum of the weights of the edges in the path.

It turns out that Prim's algorithm to find the minimum spanning tree starting from the vertex s, actually constructs the shortest path tree from the vertex s (if the edge weights are non-negative). That is, the unique path from s to a vertex v in the minimum spanning tree constructed by Prim's algorithm is the shortest This is essentially Dijkstra's algorithm to find the shortest paths from a source.

There are several variants to the problem. If some edge weights are negative, there is an algorithm due to Bellman and Ford that finds the shortest paths from a single source. Sometime we may be interested in the shortest paths between every pair of vertices in the graph. This can be solved by applying the single source shortest path algorithm once for each vertex; but it can usually be solved faster, and its structure is of interest on its own right.

See the references for more details of the algorithms for the variants.

4.3 Summary

In this section, we looked at algorithms for Shortest Paths and Minimum Spanning Tree on weighted graphs. Along the way we also saw the role of data structures in designing efficient algorithms; in particular, we saw an application of heaps. We also came across the greedy heuristic - an algorithm technique useful for designing exact or approximate solutions for optimization problems.

5 NP-completeness and Coping with it

Each of the graph (and even other) problems we have seen so far has a very efficient algorithm - an algorithm whose running time is a polynomial in the size of the input.

Are there problems that don't have such an efficient algorithm? For example, we saw in the second section that we can test whether a graph on n vertices and m edges is Eulerian in $O(m+n)$ time. How about checking whether a given graph is Hamiltonian? I.e. whether the graph has a simple path which visits all vertices exactly once? The lack of a nice characterization (as for Eulerian graphs) for Hamiltonian graphs apart, there is no efficient algorithm for testing whether a graph is hamiltonian. All known algorithms are essentially brute force, checking all possible paths in the graph, and take roughly $O(n^n)$ time. This essentially means that to solve this problem for a graph even on 50 vertices on today's powerful computers will take several centuries. Hence exponential time algorithms are considered infeasible or impractical in Computer Science. In practice, it also turns out that problems having polynomial time algorithms have a very low degree (< 4) in the runtime of their algorithms. So one is always interested in designing polynomial time algorithms.

Can we show that the Hamiltonicity problem cannot have a polynomial time algorithm? Answering such questions takes us to the arena of lower bounds which we briefly alluded to in the first section. Unfortunately proving that a problem does not have an efficient algorithm is more difficult than coming up with an efficient algorithm.

Fortunately some *evidence* to the fact the Hamiltonian problem cannot have a polynomial time algorithm is known. In 1971, Cook[2] identified a class of decision problems[5] called NP-complete problems. The class NP consists of decisions problems for which there is a polynomial time algorithm to verify the YES answer given a witness/proof for the answer. The class of NP-complete problems are those NP problems for which if there is a polynomial time algorithm to compute the answer, then all problems in the class NP have polynomial time algorithms. Cook identified the (first) problem of Boolean CNF formula satisfiability as an NP-complete problem proving from the first principles. Karp[9] in 1972 proved several graph problems NP-complete reducing from the Satisfiability problem as well as from the problems he proved NP-complete. Now there is a list of thousands of NP-complete problems. A polynomial time algorithm for any one of them would imply a polynomial time algorithm for all NP problems. Similarly a proof that any one of the NP-complete problems is not polynomial time solvable would imply a proof that all the NP-complete problems are not polynomial time solvable. Whether all NP problems have polynomial time algorithms is an important open problem in the area of Algorithms and Complexity. Proving a problem NP-complete is considered as an evidence that the problem is not efficiently solvable.

It is known that the decision version of the Hamiltonian Path problem (given an integer k, is there a path of length at least k in the graph), along with Independent Set problem (given an integer k, is there an independent set of size k?), Clique problem (is there a clique of size k in a graph) and Dominating Set problem, is NP-complete.

Since the decision version of the Hamiltonian Path problem and that of the longest path problem are the same, it follows that there is no efficient algorithm to find the longest path in the graph. Contrast this to the problem of finding the shortest path in the graph (Lecture 2).

An Example Reduction:
Proving a problem NP-complete involves first proving that the problem is in NP and then efficiently (in polynomial time) reducing an instance of a known NP-complete problem to an instance of this problem in such a way that the original instance has a YES answer if and only if the instance of this problem has a YES answer.

Suppose the problem, 'Given a graph and an integer k, does it have a clique of size at least k' is already proved to be NP-complete. Using that result we will show that the Vertex Cover problem - 'Given a graph and an integer k, does it have a vertex cover (a set S of vertices such that for every edge in the graph, one of its end points is in S) of size at most k' is NP-complete.

The fact that the vertex cover problem is in NP is easy to show. To verify that the given graph has a vertex cover of size at most k, the witness is a set S of at most k vertices and the verification algorithm has to simply go through all edges to make sure at least one of its end points is in S.

To prove it is NP-complete, we reduce the Clique problem to this. Given a graph G on n vertices (in which we are interested in solving the clique problem), construct its complement G^c. G has a clique of size at least k if and only if G^c has an independent set of size at least k. G^c has an independent set of size at least k if and only if G^c has a vertex cover of size at most $n-k$ (since the complement of an independent set is a vertex cover).

Thus G has a clique of size at least k if and only if G^c has a vertex cover of size at most $n-k$. This shows that the Vertex Cover problem is NP-complete.

Essentially we have shown that if the Vertex Cover problem has a polynomial time algorithm, then we can use that as a *subroutine* to solve the clique problem which has been

[5]For some technical reason, this entire NP-completeness theory deals only with decisions problems - problems for which only a YES or NO answer is required. This is really not a constraint as most of the optimization problems can be solved by calling an appropriate decision version a few times.

proved to be NP-complete already.

Note that quite often, the NP-complete reductions are much more complicated than this.

5.1 Coping with NP-completeness

Once a problem has been shown to be NP-complete, one stops the search for an efficient polynomial time algorithm. However, the optimization versions of most of the NP-complete problems are real world problem and so they have to be solved. There are various approaches to deal with these problems. We will describe a few approaches here.

5.1.1 Approximate Algorithms

One approach to solve the optimization versions of the NP-complete problems is to look for polynomial time approximate algorithms. One is interested in an approximate algorithm whose solution quality can be guaranteed to be reasonably close to the optimal. The *approximation ratio* of an approximate algorithm A for a minimization problem is defined to be $Max\{A(I)/opt(I)\}$ where $A(I)$ is the value of the solution produced by the approximate algorithm A on the input sequence I, and $opt(I)$ is the value of the optimal solution on the input sequence I and the Max is taken over all inputs to the problems I of size n. The approximation ration for a Maximization problem is defined as $Max\{opt(I)/A(I)\}$ where $A(I)$ and $opt(I)$ are defined as above. Note that the approximation ratio is always greater than 1, and one is interested in an approximation algorithm with approximation ratio close to 1.

Approximate Algorithms for Vertex Cover:
The vertex cover problem asks, for a given graph G, the minimum number of vertices needed to cover all edges of G.

One greedy approach is to repeatedly pick a vertex to the vertex cover that covers the maximum number of uncovered edges (once a vertex is picked to the vertex cover, all its incident edges are covered). It can be shown that the approximate ratio of such an algorithm is $O(\log n)$.

Consider the following simple approximate algorithm. Find a maximal (not maximum) matching in the graph[6]. Let the matching have k edges and hence $2k$ vertices. Output the set S of $2k$ end points of the edges in the matching as the vertex cover.

The fact that S forms a vertex cover is clear. For, an edge with both its end points outside S can be added to the maximal matching contradicting the fact that the matching is maximal. Hence every edge has at least one end point in S.

Computing the approximation ratio is usually difficult as one doesn't know the optimal value for the given input. We can get around it by using some bounds for the optimal solution. Note, in this case, that k is a lower bound on the optimal value of the vertex cover for the given graph. This is because, for each edge in the matching, at least one end point must be in the vertex cover. Thus the approximation ratio of this algorithm is $A(G)/opt(G) \leq 2k/k = 2$.

Thus this is a 2-approximation algorithm. The startling fact is that no significant improvement on this approximation ratio is known and it is a challenging open problem to design an approximate algorithm for Vertex Cover with approximation ratio better than 2.

[6]A maximal matching can be found in a graph by repeatedly picking an edge, and deleting all edges incident on both its end points.

Approximate Algorithms for the Euclidean Traveling Salesperson Problem:
The traveling saleperson problem is essentially a variation of the Hamilton cycle problem on weighted graphs. Given an undirected graph with non-negative weights on the edges (the weight of an edge (a, b) is denoted by $wt(a, b)$), the problem is to find a Hamiltonian cycle with minimum total weight. [7]

It is known that it is NP-complete to find even a polynomial time, constant ratio approximate algorithm for the general version of the problem. Here we will look at approximate algorithm for the problem for the special case when the distances (the weights) on the edges satisfy the triangle inequality (for every triple of vertices a, b, c, $wt(a, b) + wt(b, c) \geq wt(a, c)$. This is also known as the Euclidean Traveling Salesperson Problem and its decision version can also be shown to be NP-complete.

Consider the following approximate algorithm for the problem.

1. Find a minimum spanning tree T of the weighted graph.

2. Duplicate every edge of the spanning tree to get an Eulerian graph T'. Compute the Euler Tour E of T'.

3. From the Euler Tour E, obtain a Hamiltonian tour H of G as follows: whenever a sequence of edges $(a, b_1), (b_1, b_2), ...(b_{l-1}, b_l)$ is repeated in the Euler tour, simply remove those edges and replace by the edge (a, b_l) (the edge (a, b_l) would not be present in the Euler tour since the edges of the Euler tour are the tree edges and in a tree there is only one path between a pair of vertices). Output the resulting Hamiltonian tour.

That the output produced is a hamiltonian tour of the graph is clear. Let $W(F)$ is the total weight of a collection of edges in a subgraph F. $W(T)$ is the total weight of the minimum spanning tree T, and $W(E) = 2W(T)$. Also $W(H) \leq W(E)$ due to the triangle inequality (the new edges used have a total weight less than or equal to the total weight of the edges removed from the Euler tour). Thus $W(H) \leq 2W(T)$. Also $W(T)$ is a lower bound for the optimal hamiltonian tour since if T' is an optimal hamiltonian tour of G, then removing an edge from T' gives a spanning tree of G and so $W(T) \leq W(T')$.

Thus the approximation ratio of the above algorithm is $A(I)/opt(I) \leq 2W(T)/W(T) = 2$. That is the above algorithm is a 2-approximate algorithm.

Instead of duplicating every edge of the minimum spanning tree T to make it Eulerian, if we add to T a perfect matching on its odd degree vertices and then obtain an Eulerian tour and a hamiltonian tour as above, one can obtain a ratio 3/2-approximate algorithm. This is the best known approximate algorithm for the Euclidean Traveling Salesperson problem where the edge weights satisfy triangle inequality.

Exercise 5.1:
Given a graph G with p vertices, the following algorithm colours the vertices with positive integers in such a way that for each edge in the graph, the endpoints are coloured differently:

> Order the vertices of G in an arbitrary way. Colour the first vertex 1. For $i = 2$ to p do: Pick the first uncoloured vertex v (this will be the i-th vertex in the order chosen). Colour v with the smallest possible colour c such that no neighbour of v is already coloured c.

[7]This is a popular NP-complete problem due to its practical application. It gets this name, because it models the problem of finding a minimum distance tour for a traveling salesperson wanting to visit all cities of a state exactly once.

Note that this algorithm may use different number of colours on the same graph, depending on the order chosen in the first step.

1. Given a complete bipartite graph $K(n, n)$, how many colours will this algorithm use? Justify.

2. Consider a graph G constructed from $K(n, n)$ by removing a perfect matching (i.e. a set of n edges in which no two edges share vertices). What is the maximum number of colours the above algorithm will use to colour G? How should the vertices be ordered in the first step for these many colours to be used?

5.1.2 Efficient Algorithms in Special Graphs

Typically when a problem is proved NP-complete, only the general version of the problem is shown to be NP-complete (Actually there are several results showing problems NP-complete even for special cases of the problem.). For some special classes of graphs, the special structure of the class can be taken advantage to design polynomial time algorithms.

For example, the NP-completeness result of the hamiltonian path problem for undirected graphs can be extended to general directed graphs also. However, for the special class of directed graphs, tournaments, we have already seen polynomial time algorithms for finding hamiltonian paths.

Problems known to be NP-complete are studied in special classes of graphs (Bipartite graphs, Chordal graphs, Interval graphs, Perfect Graphs, Planar graphs to name a few classes). This is a potential area of current research. For example, the Vertex Cover problem has a polynomial time algorithm in Bipartite graphs. Similarly the clique problem is known to be polynomial time solvable in bipartite graphs, chordal graphs, interval graphs etc.

5.1.3 Parameterized Complexity

This is a very recent direction of research to deal with parameterized versions of hard problems.

Consider the decision version of the Vertex Cover problem: Given a graph G on n vertices and m edges and an integer k, is there a vertex cover of size at most k. Since the problem is NP-complete we don't expect an algorithm polynomial in n, m and k.

However there is an $O(n^k m)$ algorithm to solve this problem: Simply try all possible k element subsets of the vertex set, for each such subset check whether it is a vertex cover. If some such subset is a vertex cover, then the answer is YES and is NO otherwise. It is easy to see that the algorithm answers the question correctly in $O(n^k m)$ time.

This algorithm becomes impractical even for $k = 10$ or so for a reasonably sized graph. So the question addressed by this new area of Parameterized Complexity is to ask whether there is an algorithm with running time $O(f(k)p(n, m)$ where $f(k)$ is an aribitrary (typically an exponential) function of k, and $p(n, m)$ is a polynomial function of n and m. In particular, the parameter k is not an exponent of function of n or m. It turns out that it is easy to come up with an $O(2^k m)$ algorithm for Vertex Cover (and even this bound can be improved). The idea is to pick an edge, say (a, b) and try including either of its end points in a proposed vertex cover (one of them must be in ANY vertex cover). Once we pick a vertex a in the vertex cover, we an delete that vertex and all edges incident on that vertex to get a new graph. Recursively find whether the new graph has a vertex cover of size $k - 1$. If either $G - a$ or $G - b$ has a vertex cover of size $k - 1$, then we say YES else say No. It is easy to see that the algorithm takes $O(2^k m)$ time and it produces a vertex cover of size k if there is one in the graph. Such algorithms are very useful for a large value of the parameter k. In practice, such values of the parameter seems to cover practical ranges of the input and so such an approach attains practical importantance.

Classifying which of the parameterized problems have such an efficient (fixed parameter tractable) algorithm is the main goal of this new area of research. See [4] [12] for more details.

5.2 Summary

In this section, we had a glimpse of problems hard for algorithmic approach. Specifically we had a brief introduction to NP-complete problems and a few approaches to cope with NP-completeness. The approaches to deal with hard problems we saw are through approximate algorithms and parameterized tractability. We also observed that there are several classes of graphs in which some of the problems that are NP-complete in general graphs are polynomial time solvable.

6 Conclusions

Through these sections, we had a brief introduction to algorithm design and analysis concepts and some basic graph algorithms. We also looked at some recent advanced topics. Other advanced topics in Algorithms include a study of randomized algorithms (where an algorithm can make some careful random choices and one is interested in the expected runtime of the algorithm, as opposed to the worst case) and parallel algorithms (where the algorithm can make use of several available processors simultaneously).

The topics we have seen are by no means exclusive. Algorithms continue to be a goldmine of problems and a hot area of current research due to its wide applicability and interesting theoretical insights. Designing better algorithms or lower bounds form an open problem even for some of the classical problems like Minimum Spanning Tree and Maximum weighted Matching.

There are several good books on Algorithms and we list a few below. The reader is encouraged to look at them to get a more complete picture on some of the algorithms discussed here, and also to find about other topics.

References

[1] A. V. Aho, J.E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974).

(One of the classics and is popularly used as a text in courses. Though it is old and some recent results are missing, some of the classical topics are dealt with thoroughly here.)

[2] S. A. Cook, 'The complexity of theorem-proving procedures", Proceedings of the 3rd ACM Symposium on Theory of Computing, New York (1971) 151-158.

[3] T. H. Cormen, C. L. Leiserson and R. L. Rivest, C.Stein *Introduction to Algorithms*, Second Edition, MIT Press, Prentice-Hall (2002).

(A comprehensive book on Algorithms covering old and recent topics in detail. A good book to have in the library. The Indian edition is available for about 300Rs. There is an old edition without the last author which is also a reasonable book)

[4] R. D. Downey and M. R. Fellows, *Parameterized Complexity*, Springer Verlag, (1999).

[5] S. Even, *Graph Algorithms*, Computer Science Press (1979).

(A good reference for Graph Algorithms. One of the rare books having a good treatment of Planarity. For example, it gives efficient algorithms to test whether a given graph is planar or not.)

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, Freeman and Company (1979).

(A fascinating book on NP-completeness and coping strategies. It has a comprehensive list of NP-complete problems. As this is also an old book, some recent results are missing. In paritcular, a problem listed as an open problem here is probably already solved.)

[7] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press (1980)

(A good book describing efficient algorithms for several hard problems in various subclasses of Perfect Graphs.)

[8] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press (1978).

(A reasonable introductory book. There is an Indian edition.)

[9] R. M. Karp, 'Reducibility among combinatorial problems", in R. E. Miller and J. W. Thatcher (eds.), *Complexity of Computer Computations*, Plenus Press, New York (1972) 85-103.

[10] D. E. Knuth, *Sorting and Searching*, Volume 3 of the *Art of Computer Programming*, Addison-Wesley (1973).

(A classic and a thorough treatment of Sorting and Searching, the kind of problems dealt with in the first section. There is a recent second edition available. Kunth also has two other volumes in the series, one on *Fundamental Algorithms* and the other on *Seminumerical Algorithms*. He is also planning on a Volume 4 on Combinatorial Algorithms; that should be relevant to the topic of the present lectures.)

[11] K. Mehlhorn, 'Data Strucutures and Algorithms, Volume 2: Graph Algorithms and NP-Completeness', EATCS monograph on Theoretical Compuer Science, Springer Verlag (1984).

(There are two other volumes of this book: Volume 1 is about Sorting and Searching and Volume 3 is about Computational Geometry.)

[12] R.Niedermier, *An Invitation to Parameterized Complexity*, Springer Verlag, (2006).

[13] C. H. Papadimitrious and K. Steiglitz, 'Combinatorial Optimization, Algorithms and Complexity', Prentice-Hall (1982).

(Several graph problems like Shortest Paths, Network Flows, Matching have deep connections with Linear Programming and Integer Programming. This book is a good introductory book for Linear Programming and Integer Programming and graph algorithms that apply these techniques.)

[14] R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics (SIAM) (1983).

(A good book on heaps, binary search trees, minimum spanning trees, shortest paths and network flows.)