Chapter 3

Generating sets for finding faults in the network

1. INTRODUCTION:

In this chapter we have introduced concept called generating sets the application intended for to find error and faults in the network. Now a day’s all networks are having more congestion than older generation networks. So to avoid congestion in the network we are implementing new approach to find errors and faults in the network. While forming reliable communication networks, we must guarantee that, after failure of a node or links, the surviving network still allows communication between all other nodes by choosing alternate path which gives strict requirement on the connectivity of the corresponding graph. A general network design problem which requires the underlying network to be resilient to link failures is known as the edge-connectivity survivable network design problem. In this chapter we are implementing finding the faults in network using divide and conquer technique, the given graph collects set of vertices and generates even number set of vertices and odd number set of vertices for some pre defined properties.

2. DIVIDE AND CONQUER METHOD

Divide and conquer method is a top-down technique for designing algorithms which consist of dividing the problem into smaller problems hoping that the solutions of the sub problems are easier to find. The solutions of all small problems are then combined to get a solution for the original problem.

The divide and conquer technique of solving a problem involves three steps at each level of the recursion:

**Divide:** The problem is divided into a number of sub problems

**Conquer:** If the problems are smaller in size, the problem can be solved using straightforward method. If the sub problems are larger in size, they are divided into number of sub-problems of the same type and size. Each of the sub-problems is solved recursively.
**Combine**: The solutions of sub problems are combined to get the solution for the larger problem.

![Diagram of Divide and Conquer technique](image)

**Fig.3.1 Divide and Conquer technique**

### 2.1 DIVIDE-AND CONQUER IS A GENERAL ALGORITHM DESIGN PARADIGM

**Divide**: divide the input data $S$ in two or more disjoint subsets $S_1, S_2,$

**Recursively**: solve the sub problems recursively

**Conquer**: combine the solutions for $S_1, S_2$... into a solution for $S$

The base case for the recursion is sub problems of constant size. Analysis can be done using Recurrence equations.
2.2 ALGORITHM

Algorithm D-and-C (n: input size)
{
    if n ≤ n0 /* small size problem*/
        Solve problem without further sub-division;
    Else
        {
            Divide into m sub-problems;
            Conquer the sub-problems by solving them independently and recursively; /* D-and-C(n/k) */
            Combine the solutions;
        }
}

Advantages

Solving difficult problems

- Divide and conquer is a powerful tool for solving conceptually difficult problems, such as the classic Tower of Hanoi puzzle: all it requires is a way of breaking the problem into sub-problems, of solving the trivial cases and of combining sub-problems to the original problem. Indeed, for many such problems the paradigm offers the only simple solution.

- Dividing the problem into sub-problems so that the sub-problems can be combined again is often the major difficulty in designing a new algorithm. Straight forward and running times are often easily determined.
Algorithm efficiency

Moreover, divide and conquer often provides a natural way to design efficient algorithms.

For example, if the work of splitting the problem and combining the partial solutions is proportional to the problem's size \( n \), there are a bounded number \( p \) of subproblems of size \( \sim n/p \) at each stage, and the base cases require \( O(1) \) (constant bounded) time, then the divide-and-conquer algorithm will have \( O(n \log n) \) complexity. This is used for problems such as sorting and FFTs to reduce the complexity from \( O(n^2) \), although in general there may also be other approaches to designing efficient algorithms.

Parallelism

Divide and conquer algorithms are naturally adapted for execution in multi-processor machines, especially shared-memory systems where the communication of data between processors does not need to be planned in advance, because distinct subproblems can be executed on different processors....

Memory access

Divide-and-conquer algorithms naturally tend to make efficient use of memory caches. The reason is that once a sub-problem is small enough, it and all its sub-problems can, in principle, be solved within the cache, without accessing the slower main memory. An algorithm designed to exploit the cache in this way is called cache oblivious, because it does not contain the cache size(s) as an explicit parameter.

Moreover, D&C algorithms can be designed for many important algorithms, such as sorting, FFTs, and matrix multiplication, in such a way as to be optimal cache oblivious algorithms—they use the cache in a provably optimal way, in an asymptotic sense, regardless of the cache size. In contrast, the traditional approach to exploiting the cache is blocking, where the problem is explicitly divided into chunks of the appropriate size—this can also use the cache optimally, but only when the algorithm
Disadvantages

- Recursion is slow
- Very simple problem may be more complicated than an iterative approach. Example: adding n numbers etc.
- One commonly argued disadvantage of a divide-and-conquer approach is that recursion is slow: the overhead of the repeated subroutine calls, along with that of storing the call stack (the state at each point in the recursion), can outweigh any advantages of the approach. This, however, depends upon the implementation style: with large enough recursive base cases, the overhead of recursion can become negligible for many problems.
- Another problem of a divide-and-conquer approach is that, for simple problems, it may be more complicated than an iterative approach, especially if large base cases are to be implemented for performance reasons. For example, to add N numbers, a simple loop to add them up in sequence is much easier to code than a divide-and-conquer approach that breaks the set of numbers into two halves, adds them recursively, and then adds the sums. (On the other hand, the latter approach turns out to be much more accurate in finite-precision floating-point arithmetic.)

2.3 DIVIDE-AND-CONQUER RECURRENCE RELATIONS

Suppose that a recursive algorithm divides a problem of size $n$ into parts, where each sub-problem is of size $n/b$. Also, suppose that a total number of $g(n)$ extra operations are needed in the conquer step of the algorithm to combine the solutions of the sub-problems into a solution of the original problem. Let $f(n)$ is the number of operations required to solve the problem of size $n$. Then $f$ satisfies the recurrence relation and it is called divide-and-conquer Recurrence relation.

$$F(n) = a \cdot f\left(\frac{n}{b}\right) + g(n)$$

The computing time of Divide and conquer is described by recurrence relation.

$$T(n) = g(n) \text{ where } n \text{ small}$$

$$T(n_3) + T(n_2) + \ldots + T(n_k) + f(n) \text{ other wise}$$
T (n) is the time for Divide and Conquer on any input of size n and g (n) is the time to compute the answer directly for small inputs. The function of f (n) is the time for dividing P combining solutions to sub problems. For divide-and-conquer-based algorithms that produce sub problems of the same type as the original problem, then such algorithm described using recursion.

The complexity of many divide-and-conquer algorithms is given by recurrence of the form.

\[
T(n) = \begin{cases} 
T (1) & \text{if } n = 1 \\
\frac{a T (n)}{b} + f(n) & \text{if } n > 1 
\end{cases}
\]

where a and b are known constants, and n is a power of b \((n = b^k)\).

One of the methods for solving any such recurrence relation is called substitution method.

**Examples**

If \(a=2\) and \(b=2\). Let \(T(1)=2\) and \(f(n)=n\). Than

\[
T(n) = 2T(n/2) + n \\
= 2[2T(n/4) + n/2] + n \\
= 4T(n/4) + 2n \\
= 4[2T(n/8) + n/4] + 2n \\
= 8T(n/8) + 3n
\]

In general, \(T(n) = 2^i T(n/2^i) + in\), for any log 2 \(n\) \(\geq i \geq 1\). In Particular, Then \(T(n) = 2^{log_2 n}T(n/2^{log_2 n}) + n log_2 n\) corresponding to choice of

\(i = log_2 n\). Thus, \(T(n) = nT(1) + n log_2 n = n log_2 n + 2n\).

### 2.3.1 DIVIDE AND CONQUER APPLICATIONS

#### 2.3.1.1 Min and Max

The minimum of a set of elements: The first order statistic \(i = 1\)

The maximum of a set of elements: The nth order statistic \(i = n\)

The median is the “halfway point” of the set \(l = (n+1)/2\), is unique

When \(n\) is odd
Finding Minimum or Maximum
Alg: MINIMUM (A, n)

\[
\begin{align*}
    &\text{min} \leftarrow A[1] \\
    &\text{for } i \leftarrow 2 \text{ to } n \\
    &\quad \text{do if } \text{min} > A[i] \text{ then } \text{min} \leftarrow A[i] \\
    &\text{return min}
\end{align*}
\]

How many comparisons are needed?

\( n - 1 \): each element, except the minimum, must be compared to a smaller element at least once. The same number of comparisons is needed to find the maximum. The algorithm is optimal with respect to the number of comparisons performed.

2.3.1.2 Simultaneous Min, Max

Find min and max independently

Use \( n - 1 \) comparisons for each \( \geq \) total of \( 2n - 2 \)

At most \( 3n/2 \) comparisons are needed. Process elements in pairs. Maintain the minimum and maximum of elements seen so far. Don’t compare each element to the minimum and maximum separately. Compare the elements of a pair to each other. Compare the larger element to the maximum so far, and compare the smaller element to the minimum so far. This leads to only 3 comparisons for every 2 elements.

2.3.1.3 Analysis of Simultaneous Min, Max

Setting up initial values:

\( n \) is odd: compare the first two elements, assign the smallest one to min and the largest one to max

\( n \) is even:

Total number of comparisons:

\( n \) is odd: we do \( 3(n-1)/2 \) comparisons

\( n \) is even: we do 1 initial comparison + \( 3(n-2)/2 \) more comparisons = \( 3n/2 - 2 \) comparisons
2.4 SETS:

A set is a collection of objects called elements or members. A set with no objects is called the empty set and is denoted by \( \emptyset \) (or sometimes by \{ \}).

Think of a set as a club with a certain membership. For example, the students who play chess are members of the chess club. However, do not take the analogy too far. A set is only defined by the members that form the set; two sets that have the same members are the same set. Most of the time we will consider sets of numbers.

For example, the set
\[ S = \{ 0, 1, 2 \} \]
is the set containing the three elements 0, 1, and 2. We write \( 1 \in S \) to denote that the number 1 belongs to the set S. That is, 1 is a member of S. Similarly we write \( 7 \notin S \) to denote that the number 7 is not in S. That is, 7 is not a member of S.

The elements of all sets under consideration come from some set we call the universe. For simplicity, we often consider the universe to be the set that contains only the elements we are interested in. The universe is generally understood from context and is not explicitly mentioned. In this course, our universe will most often be the set of real numbers.

While the elements of a set are often numbers, other objects, such as other sets, can be elements of a set.

A set may contain some of the same elements as another set.

For example,
\[ T = \{ 0, 2 \} \]
contains the numbers 0 and 2. In this case all elements of T also belong to S. We write \( T \subseteq S \). More formally we make the following definition.

2.4.1 Subset

- A set \( A \) is a subset of a set \( B \) if \( x \in A \) implies \( x \in B \), and we write \( A \subseteq B \). That is, all members of \( A \) are also members of \( B \).

- Two sets \( A \) and \( B \) are equal if \( A \subseteq B \) and \( B \subseteq A \). We write \( A = B \). That is, \( A \) and \( B \) contain exactly the same elements. If it is not true that \( A \) and \( B \) are equal, then we write \( A \neq B \).
A set $A$ is a proper subset of $B$ if $A \subseteq B$ and $A \neq B$.

We write $A \subseteq B$, when $A = B$, we consider $A$ and $B$ to just be two names for the same exact set.

For example, for $S$ and $T$ defined above we have $T \subseteq S$, but $T \neq S$. So $T$ is a proper subset of $S$. At this juncture, we also mention the set building notation,

$$\{ x \in A : P(x) \}.$$  

This notation refers to a subset of the set $A$ containing all elements of $A$ that satisfy the property $P(x)$. The notation is sometimes abbreviated (A is not mentioned) when understood from context. Furthermore, $x \in A$ is sometimes replaced with a formula to make the notation easier to read.

2.5 NETWORKS

A computer network consists of a collection of computers, printers and other equipment that is connected together so that they can communicate with each other. Fig 1 gives an example of a network in a school comprising of a local area network or LAN connecting computers with each other, the internet, and various servers.

Broadly speaking, there are two types of network configuration, peer-to-peer networks and client/server networks.
2.5.1 Peer-to-peer networks

Peer-to-peer networks are more commonly implemented where less than ten computers are involved and where strict security is not necessary. All computers have the same status, hence the term 'peer', and they communicate with each other on an equal footing. Files, such as word processing or spreadsheet documents, can be shared across the network and all the computers on the network can share devices, such as printers or scanners, which are connected to any one computer.

![Peer to Peer Network diagram](image-url)

Fig 3.3: Peer to Peer Network

2.5.2 Client/server networks

Client/server networks are more suitable for larger networks. A central computer, or 'server', acts as the storage location for files and applications shared on the network. Usually the server is a higher than average performance computer. The server also controls the network access of the other computers which are referred to as the 'client' computers. Typically, teachers and students in a school will use the client computers for their work and only the network administrator (usually a designated staff member) will have access rights to the server.
3 PROPOSED WORK

Fig 3.4: Client - Server Networking

Fig 3.5: Graph with weighted vertices
Collecting all the weights of the given graph consider as a universal set.

\[ U = \{0, 1, 2, 3, 4, 5, 6\} \]

3.1. Property:-

Select even weight and odd weight edges from universal set then prepare subsets as below.

**Even Set** = \{2, 4, 6\}

**Odd Set** = \{1, 3, 5\}

3.2. Generating sets for finding faults in the networks Algorithm as follows Algorithm:-

Step 1. Start
Step 2. InputGraph
Step 3. Collect set of all the weighted edges
Step 4. Count the number of Odd and even edges.
Step 5. if((n%2 == 0)&&(n == working))
    \[
    \text{even} = \text{even} + 1;
    \]
else
    \[
    \text{odd} = \text{odd} + 1
    \]
Step 6. display two subsets of odd and even
Step 7.
    if (even == count)
    {
        \text{no fault in even edge}
if (odd == count) {
    no fault in odd edge
} 

else {
    fault is odd edge so go to correct odd edge lines.
}

Correct the algorithm

Step 10. Stop

4. CONCLUSION:

We are representing the Generating sets for finding faults in the networks. This is very simple to understand and implement. This approach executes sequentially and here we are using sets to find faults also it is time saving, because we are implementing using divide and conquer technique so this algorithm takes half of the time comparing with existing algorithm.
REFERENCES:

[10] Dominik Alban Scheder,"Approaches to Approximating the minimum weight k-Edge 592

- 60 -