Algorithm Analysis and Design

(Assignment –I)

Submitted in partial fulfilment of the requirements for the degree of

Master of Technology in Information Technology

by

Vijayananda D Mohire

(Enrolment No.921DMTE0113)

Information Technology Department

Karnataka State Open University

Manasagangotri, Mysore – 570006

Karnataka, India

(2009)

Algorithm Analysis and Design

CERTIFICATE

This is to certify that the Assignment-I entitled (Algorithm Analysis and Design, subject code: MT13) submitted by Vijayananda D Mohire having Roll Number 921DMTE0113 for the partial fulfilment of the requirements of Master of Technology in Information Technology degree of Karnataka State Open University, Mysore, embodies the bonafide work done by him under my supervision.

Place: _____ **Signature of the Internal Supervisor**

Name

Date: _____ **Designation**

For Evaluation

Question Number	Maximum Marks	Marks awarded	Comments, if any
1	1		
2	1		
3	1		
4	1		
5	1		
6	1		
7	1		
8	1		
9	1		
10	1		
TOTAL	10		

Preface

This document has been prepared specially for the assignments of M.Tech – IT I Semester. This is mainly intended for evaluation of assignment of the academic M.Tech - IT, I semester. I have made a sincere attempt to gather and study the best answers to the assignment questions and have attempted the responses to the questions. I am confident that the evaluator's will find this submission informative and evaluate based on the provide content.

For clarity and ease of use there is a Table of contents and Evaluators section to make easier navigation and recording of the marks. A list of references has been provided in the last page – Bibliography that provides the source of information both internal and external. Evaluator's are welcome to provide the necessary comments against each response, suitable space has been provided at the end of each response.

I am grateful to the Infysys academy, Koramangala, Bangalore in making this a big success. Many thanks for the timely help and attention in making this possible within specified timeframe. Special thanks to Mr. Vivek and Mr. Prakash for their timely help and guidance.

Candidate's Name and Signature Date

Table of Contents

Table of Figures

ALGORITHM ANALYSIS AND DESIGN

RESPONSE TO ASSIGNMENT – I

Question 1 What is the difference between DFS & BFS?

Answer 1

Lets refer to Figure 1. A depth-first search traverses the graph in the following order: ABDBEBACF. If you are familiar with trees, you recognize this type of search as a variation of an inorder tree traversal. That is, the path goes left until a terminal node is reached or the goal is found. If a terminal node is reached, the path backs up one level, goes right, and then left until either the goal or a terminal node is encountered. This procedure is repeated until the goal is found or the last node in the search space has been examined. (Schildt, 2000)

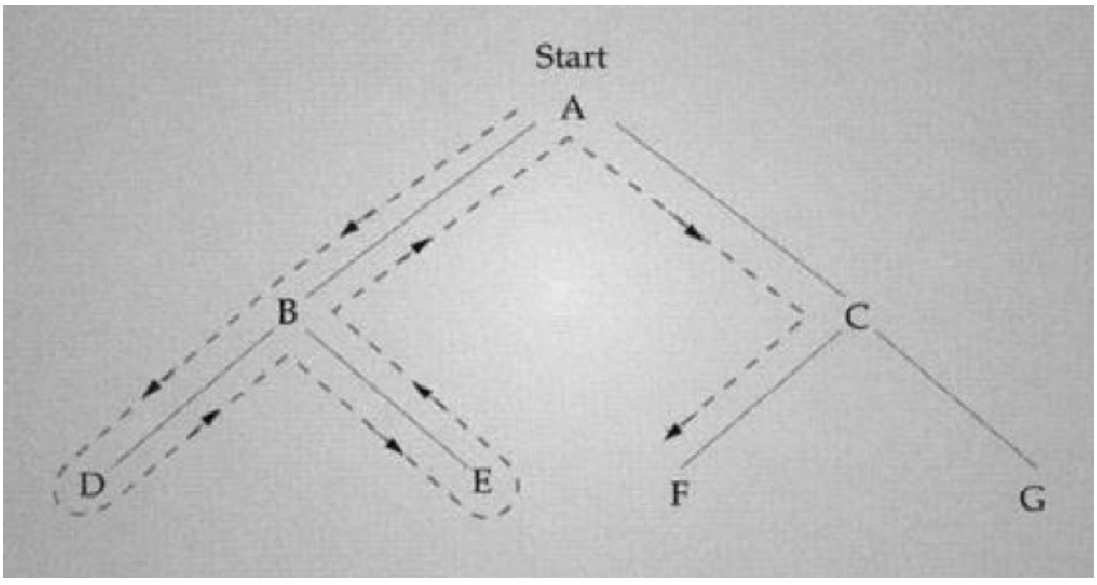

Figure 1 Depth-First Search (Schildt, 2000)

The opposite of the depth-first search is the *breadth-first search*. In this method,

each node on the same level is checked before the search proceeds to the next deeper level.

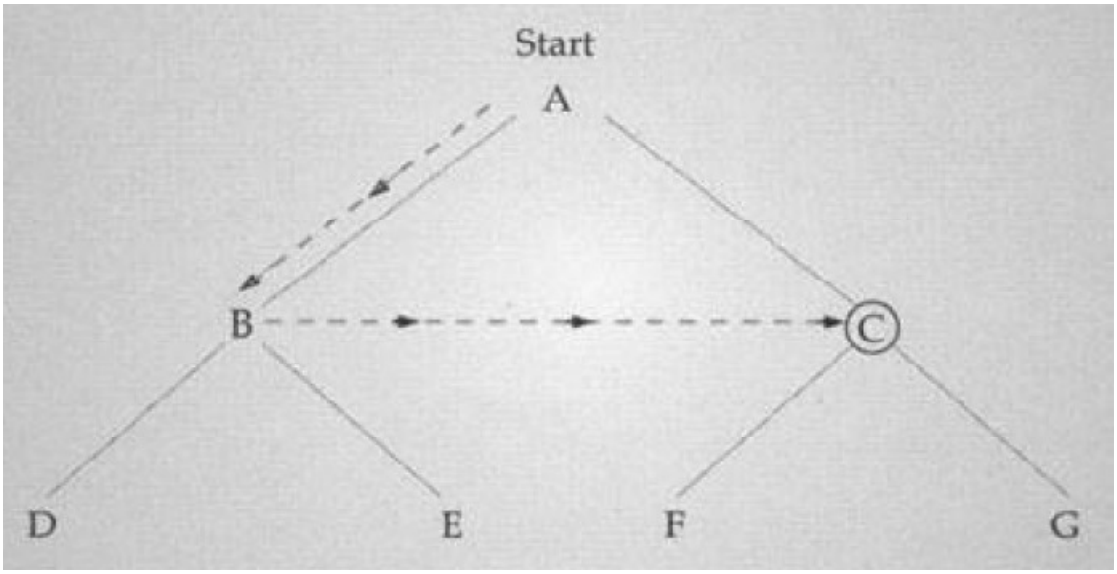

Figure 2 Breadth-First Search (Schildt, 2000)

Unlike breadth-first search, whose predecessor subgraph forms a tree, 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.

Evaluator's Comments if any:

Question 2 What are the worst, good/best and average cases regarding with algorithm?

Answer 2

- The worst-case running time gives a guaranteed upper bound for any input. For many algorithms, the worst case occurs often. For example, when searching, the worst case often occurs when the item being searched for is not present, and searches for empty items may be frequent.

- The average case is interesting and important because it gives a closer estimation of the realistic running time. However, its consideration usually requires more efforts (algebraic transformations, etc.). On the other hand, it is often about as “bad” as the worst case. Hence, it is often enough to consider the worst case

- The worst case is the most interesting. The average case is interesting, but often is as “bad” as the worst case and may be estimated by the worst case. The best case is the least interesting

- **Best case**: elements already sorted $t_i=1$, running time = $f(n)$, i.e., *linear* time. (Šaltenis, 2001)
- **Worst case**: elements are sorted in inverse order $t_j=j$, running time = $f(n^2)$, i.e., *quadratic* time
- **Average case**: $t_j=j/2$, running time = $f(n^2)$, i.e., *quadratic* time

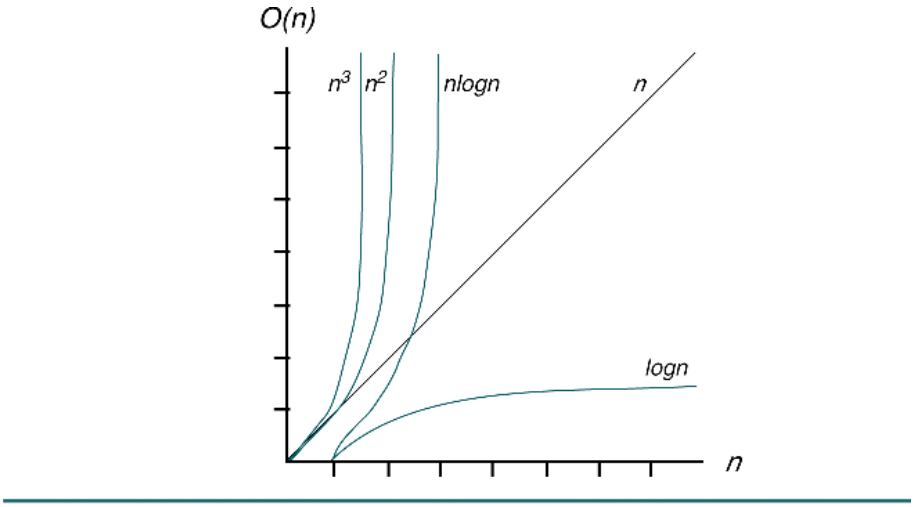

Figure 3 Plot of Efficiency Measures

Worst case is usually used:

- It is an upper-bound and in certain application domains (e.g., air traffic control, surgery) knowing the **worst-case** time complexity is of crucial importance
- For some algorithms **worst case** occurs fairly often
- The **average case** is often as bad as the **worst case**
- Finding the **average case** can be very difficult

Evaluator's Comments if any:

Question 3(i) Write a short note on –

Spanning tree

Answer 3(i)

In the mathematical field of graph theory, a spanning tree T of a connected, undirected graph G is a tree composed of all the vertices and some (or perhaps all) of the edges of G. Informally, a spanning tree of G is a selection of edges of G that form a tree spanning every vertex. That is, every vertex lies in the tree, but no cycles (or loops) are formed. On the other hand, every bridge of G must belong to T. (Anonymous, Spanning Tree, 2009)

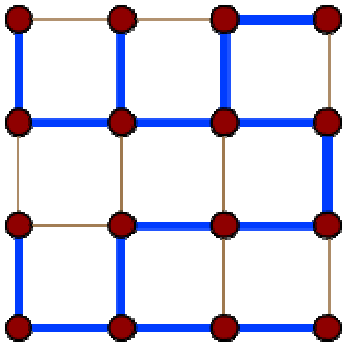

Figure 4 Spanning Tree (Anonymous, Spanning Tree, 2009)

A spanning tree (blue heavy edges) of a grid graph.

A spanning tree of a connected graph G can also be defined as a maximal set of edges of G that contains no cycle, or as a minimal set of edges that connect all vertices.

In certain fields of graph theory it is often useful to find a minimum spanning tree of a weighted graph. Other optimization problems on spanning trees have also been studied, including the maximum spanning tree, the minimum tree that spans at least k vertices, the minimum spanning tree with at most k edges per vertex (Degree-Constrained Spanning Tree), the spanning tree with the largest number of leaves (closely related to the smallest connected dominating set), the spanning tree with the fewest leaves (closely related to the Hamiltonian path problem), the minimum diameter spanning tree, and the minimum dilation spanning tree.

Adding just one edge to a spanning tree will create a cycle; such a cycle is called a fundamental cycle. There is a distinct fundamental cycle for each edge; thus, there is a one-to-one correspondence between fundamental cycles and edges not in the spanning tree. For a connected graph with V vertices, any spanning tree will have V-1 edges, and thus, a graph of E edges will have E-V+1 fundamental cycles. For any given spanning tree these cycles form a basis for the cycle space.

The classic spanning tree algorithm, depth-first search (DFS), is due to Robert Tarjan. Another important algorithm is based on breadth-first search (BFS).

Evaluator's Comments if any:

Question 3(ii) Write a short note on –

Dijkastra's algorithm

Answer 3(ii)

Dijkstra's algorithm, conceived by Dutch computer scientist Edsger Dijkstra in 1959, is a graph search algorithm that solves the single-source shortest path problem for a graph with nonnegative edge path costs, producing a shortest path tree. This algorithm is often used in routing. An equivalent algorithm was developed by Edward F. Moore in 1957. (Anonymous, Dijkstra's algorithm, 2009)

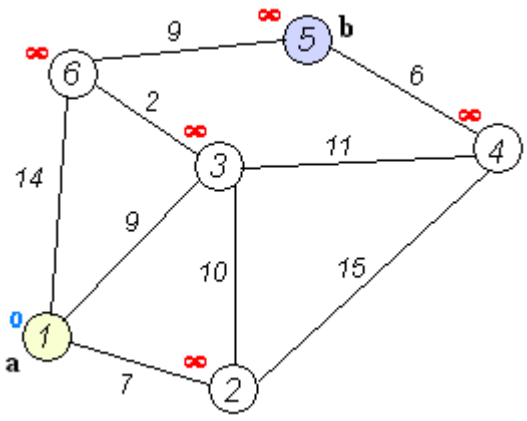

Figure 5 Dijkstra's algorithm (Anonymous, Dijkstra's algorithm, 2009)

For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the

destination vertex has been determined. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path first is widely used in network routing protocols, most notably IS-IS and OSPF (Open Shortest Path First).

In the following algorithm, the code u := vertex in Q with smallest dist[], searches for the vertex u in the vertex set Q that has the least dist[u] value. That vertex is removed from the set Q and returned to the user. dist_between(u, v) calculates the length between the two neighbor-nodes u and v. alt on line 13 is the length of the path from the root node to the neighbor node v if it were to go through u. If this path is shorter than the current shortest path recorded for v, that current path is replaced with this alt path. The previous array is populated with a pointer to the "next-hop" node on the source graph to get the shortest route to the source.

```
1  function Dijkstra (Graph, source):
2      for each vertex v in Graph: // Initializations
3          dist[v] := infinity          // Unknown distance function from source to v
4          previous[v] := undefined // Previous node in optimal path from source
5      dist[source] := 0                // Distance from source to source
```

```
6     Q := the set of all nodes in Graph
      // All nodes in the graph are unoptimized - thus are in Q
7     while Q is not empty:          // The main loop
8         u := vertex in Q with smallest dist[]
9         if dist[u] = infinity:
10            break           // all remaining vertices are inaccessible from source
11        remove u from Q
12        for each neighbor v of u: // where v has not yet been removed from Q.
13            alt := dist[u] + dist_between(u, v)
14            if alt < dist[v]:           // Relax (u,v,a)
15                dist[v] := alt
16                previous[v] := u
17    return dist[]
```

Evaluator's Comments if any:

Question 4 What are the properties of Binary Tree?

Answer 4

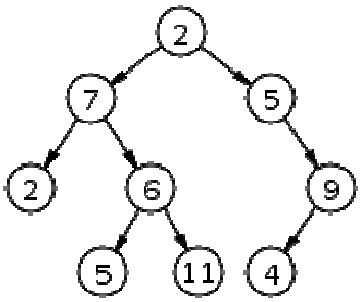

Figure 6 Binary Tree (Anonymous, Binary Tree, 2009)

In computer science, a binary tree is a tree data structure in which each node has at most two children. Typically the first node is known as the parent and the child nodes are called left and right. Binary trees are commonly used to implement binary search trees and binary heaps.

Properties of binary trees (Anonymous, Binary Tree, 2009)

- The number of nodes n in a perfect binary tree can be found using this formula: $n = 2^{h+1} - 1$ where h is the height of the tree.
- The number of nodes n in a complete binary tree is minimum: $n = 2^{h}$ and maximum: $n = 2^{h+1} - 1$ where h is the height of the tree.
- The number of nodes n in a perfect binary tree can also be found using this formula: $n = 2L - 1$ where L is the number of leaf nodes in the tree.
- The number of leaf nodes n in a perfect binary tree can be found using this formula: $n = 2^{h}$ where h is the height of the tree.

- The number of NULL links in a Complete Binary Tree of n-node is (n+1).
- The number of leaf node in a Complete Binary Tree of n-node is UpperBound(n / 2).
- For any non-empty binary tree with n_0 leaf nodes and n_2 nodes of degree 2, $n_0 = n_2 + 1$

Properties of binary trees (KSOU, 2009)

- The maximum number of nodes on Level i is 2^{i-1} , i >=1
- The maximum number of nodes in a binary tree of depth k is 2^{k-1} , k>=1
- For any non-empty tree T, if n_0 is the number of leaf nodes and n_2 is the number of nodes of degree 2, then $n_0=n_2+1$.

Evaluator's Comments if any:

Question 5 What is the need of an algorithm?

Answer 5

An algorithm is the formal declaration of a method of solving a particular problem. It is often used to describe a computer procedure, using words instead of computer instructions, that is computer language independent.

As an example, the binary search algorithm might be stated as...
Given an ordered list of elements; in order to find a particular element start by picking the middle element and comparing it against the search target. If found, the search is ended. If not found, subdivide the list into a smaller list, picking either the portion above the selected element or the portion below the selected element, the decision of which being dependent on the initial comparison, higher or lower. Once subdivided, iterate the algorithm from the top. You will eventually either find the element desired, or you will ultimately select an list of zero length, at which point you abandon the search, but also at which point you know where in the original list to insert the element should you choose to do so.

Algorithms are essential to the way computers process information. Many computer programs contain algorithms that specify the specific instructions a computer should perform (in a specific order) to carry out a specified task, such as calculating employees' paychecks or printing students' report cards. Thus, an algorithm can be considered to be any sequence of operations that

can be simulated by a Turing-complete system.

Typically, when an algorithm is associated with processing information, data is read from an input source, written to an output device, and/or stored for further processing. Stored data is regarded as part of the internal state of the entity performing the algorithm. In practice, the state is stored in one or more data structures.

For any such computational process, the algorithm must be rigorously defined: specified in the way it applies in all possible circumstances that could arise. That is, any conditional steps must be systematically dealt with, case-by-case; the criteria for each case must be clear (and computable).

Because an algorithm is a precise list of precise steps, the order of computation will always be critical to the functioning of the algorithm. Instructions are usually assumed to be listed explicitly, and are described as starting "from the top" and going "down to the bottom", an idea that is described more formally by flow of control.

Example:

Algorithm LargestNumber

Input: A non-empty list of numbers *L*.

Output: The *largest* number in the list *L*.

largest ← L_0

for each *item* **in** the list $L_{\geq 1}$, **do**

if the *item* > *largest*, **then**

largest ← the *item*

return *largest*

Evaluator's Comments if any:

Question 6 What is Buddy system?

Answer 6

The **buddy memory allocation** technique (Anonymous, Buddy memory allocation, 2009) is a memory allocation technique that divides memory into partitions to try to satisfy a memory request as suitably as possible. This system makes use of

splitting memory into halves to try to give a best-fit. According to Donald Knuth, the buddy system was invented in 1963 by Harry Markowitz, who won the 1990 Nobel Memorial Prize in Economics, and was first described by Kenneth C. Knowlton (published 1965)

Compared to the memory allocation techniques (such as paging) that modern operating systems use, the buddy memory allocation is relatively easy to implement, and does not have the hardware requirement of an MMU. Thus, it can be implemented, for example, on Intel 80286 and below computers.
In comparison to other simpler techniques such as dynamic allocation, the buddy memory system has little external fragmentation, and has little overhead trying to do compaction of memory.

However, because of the way the buddy memory allocation technique works, there may be a moderate amount of internal fragmentation - memory wasted because the memory requested is a little larger than a small block, but a lot smaller than a large block. (For instance, a program that requests 66K of memory would be allocated 128K, which results in a waste of 62K of memory). Internal fragmentation is where more memory than necessary is allocated to satisfy a request, thereby wasting memory. External fragmentation is where enough memory is free to satisfy a request, but it is split into two or more chunks, none of

which is big enough to satisfy the request.

The buddy memory allocation technique allocates memory in powers of 2, i.e 2x, where x is an integer. Thus, the programmer has to decide on, or to write code to obtain, the upper limit of x. For instance, if the system had 2000K of physical memory, the upper limit on x would be 10, since 210 (1024K) is the biggest allocatable block. This results in making it impossible to allocate everything in as a single chunk; the remaining 976K of memory would have to be taken in smaller blocks.

After deciding on the upper limit (let's call the upper limit u), the programmer has to decide on the lower limit, i.e. the smallest memory block that can be allocated. This lower limit is necessary so that the overhead of storing used and free memory locations is minimized. If this lower limit did not exist, and many programs request small blocks of memory like 1K or 2K, the system would waste a lot of space trying to remember which blocks are allocated and unallocated. Typically this number would be a moderate number (like 2, so that memory is allocated in 2^2 = 4K blocks), small enough to minimize wasted space, but large enough to avoid excessive overhead. Let's call this lower limit l.

Now that we have our limits, let us see what happens when a program makes requests for memory. Let's say in this system, l = 6, which results in blocks 26 =

64K in size, and u = 10, which results in a largest possible allocatable block, 210 = 1024K in size. The following shows a possible state of the system after various memory requests.

	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K
$t = 0$	1024K										
$t = 1$	A-64K	64K	128K		256K				512K		
$t = 2$	A-64K	64K	B-128K		256K				512K		
$t = 3$	A-64K	C-64K	B-128K		256K				512K		
$t = 4$	A-64K	C-64K	B-128K		D-128K		128K		512K		
$t = 5$	A-64K	64K	B-128K		D-128K		128K		512K		
$t = 6$	128K		B-128K		D-128K		128K		512K		
$t = 7$	256K				D-128K		128K		512K		
$t = 8$	1024K										

Figure 7 Buddy Memory Allocation (Anonymous, Buddy memory allocation, 2009)

This allocation could have occurred in the following manner

1. Program A requests memory 34K..64K in size
2. Program B requests memory 66K..128K in size
3. Program C requests memory 35K..64K in size
4. Program D requests memory 67K..128K in size
5. Program C releases its memory
6. Program A releases its memory
7. Program B releases its memory
8. Program D releases its memory

As you can see, what happens when a memory request is made is as follows:

- If memory is to be allocated

1. Look for a memory slot of a suitable size (the minimal 2k block that is larger or equal to that of the requested memory)
    1. If it is found, it is allocated to the program
    2. If not, it tries to make a suitable memory slot. The system does so by trying the following:
        i. Split a free memory slot larger than the requested memory size into half
        ii. If the lower limit is reached, then allocate that amount of memory
        iii. Go back to step 1 (look for a memory slot of a suitable size)
        iv. Repeat this process until a suitable memory slot is found

- If memory is to be freed

1. Free the block of memory
2. Look at the neighboring block - is it free too?
3. If it is, combine the two, and go back to step 2 and repeat this process until either the upper limit is reached (all memory is freed), or until a non-free neighbor block is encountered

This method of freeing memory is rather efficient, as compaction is done [illegible] number of compactions required equal to $\log_2(u/l)$ (i.e. $\log_2(u) - \log_2(l)$).

Typically the buddy memory allocation system is implemented with the use of

a binary tree to represent used or unused split memory blocks.

However, there still exists the problem of internal fragmentation. In many situations, it is essential to minimize the amount of internal fragmentation. This problem can be solved by slab allocation.

Evaluator's Comments if any:

Question 7 Describe the various issues with memory?

Answer 7

The basic problem in managing memory is knowing when to keep the data it contains, and when to throw it away so that the memory can be reused. This sounds easy, but is, in fact, such a hard problem that it is an entire field of study in its own right. In an ideal world, most programmers wouldn't have to worry about memory management issues. Unfortunately, there are many ways in which poor memory management practice can affect the robustness and speed of programs, both in manual and in automatic memory management. (The Memory Management Reference, 2009)

Typical problems include:

Premature free or dangling pointer

Many programs give up memory, but attempt to access it later and crash or behave randomly. This condition is known as premature free, and the surviving reference to the memory is known as a dangling pointer. This is usually confined to manual memory management.

Memory leak

Some programs continually allocate memory without ever giving it up and eventually run out of memory. This condition is known as a memory leak.

External fragmentation

A poor memory allocator can do its job of giving out and receiving blocks of memory so badly that it can no longer give out big enough blocks despite having enough spare memory. This is because the free memory can become split into many small blocks, separated by blocks still in use. This condition is known as external fragmentation.

Poor locality of reference

Another problem with the layout of allocated blocks comes from the way that modern hardware and operating system memory managers handle memory: successive memory accesses are faster if they are to nearby memory locations. If the memory manager places far apart the blocks a program will use together, then this will cause performance problems. This condition is known as poor locality of reference.

Inflexible design

Memory managers can also cause severe performance problems if they have been designed with one use in mind, but are used in a different way. These problems occur because any memory management solution tends to make assumptions about the way in which the program is going to use memory, such as typical block sizes, reference patterns, or lifetimes of objects. If these assumptions are wrong, then the memory manager may spend a lot more time doing bookkeeping work to keep up with what's happening.

Interface complexity

If objects are passed between modules, then the interface design must

consider the management of their memory.

A well-designed memory manager can make it easier to write debugging tools, because much of the code can be shared. Such tools could display objects, navigate links, validate objects, or detect abnormal accumulations of certain object types or block sizes.

Evaluator's Comments if any:

Question 8 What is Circuit satisfiability?

Answer 8

Circuit satisfiability (Saia, 2004) is a good example of a problem that we don't know how to solve in polynomial time. In this problem, the input is a boolean circuit: a collection of AND, OR, and NOT gates connected by wires. We'll assume there are no loops in the circuit (so no delay lines or flip-flops)

The circuit satisfiability problem asks, given a circuit, whether there is an input that makes the circuit output **True**. In other words, does the circuit always

output false for any collection of inputs. Nobody knows how to solve this problem faster than just trying all 2^m possible inputs to the circuit but this requires exponential time. On the other hand nobody has every proven that this is the best we can do!

Example:

Given a circuit with n-inputs and one output, is there a way to assign 0 1 values to the input wires so that the output value is 1 (true)?

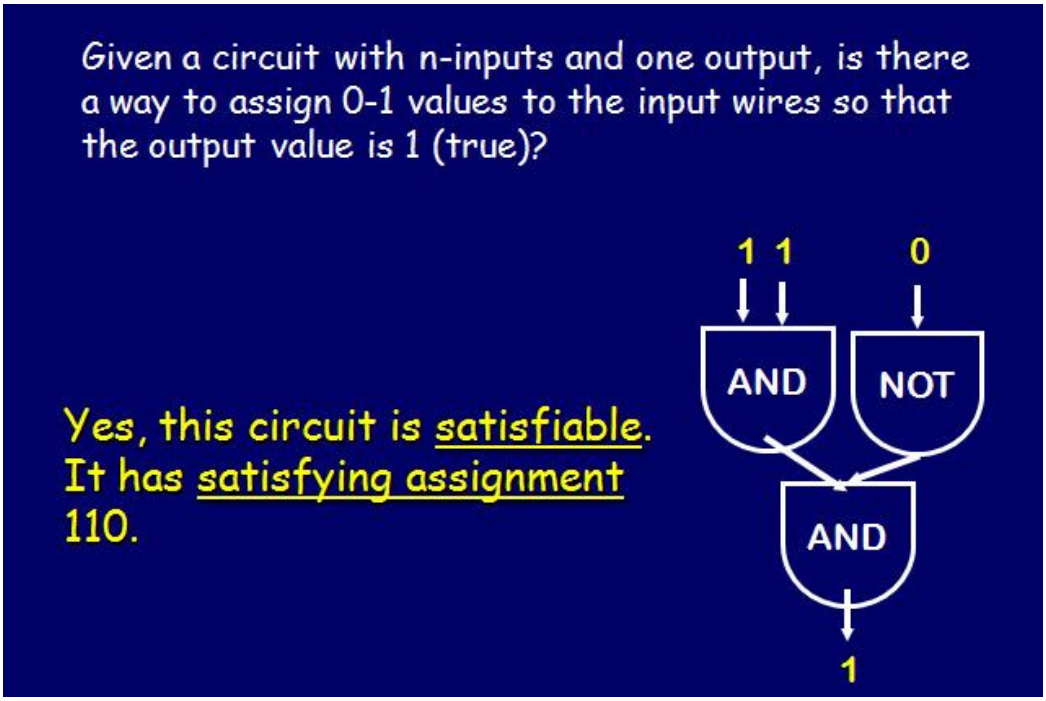

Figure 8 Circuit satisfiability

Evaluator's Comments if any:

Question 9 Describe the vertex cover problem?

Answer 9

A vertex cover for an undirected graph G=(V,E) is a subset S of its vertices such that each edge has at least one endpoint in S. In other words, for each edge ab in E, one of a or b must be an element of S.

Example; In the graph given below,(1,3,5,6) is an example of a vertex cover of size 4. However, it is not a smallest vertex cover since there exist vertex covers of size 3, such as (2,4,5) and (1,2,4).

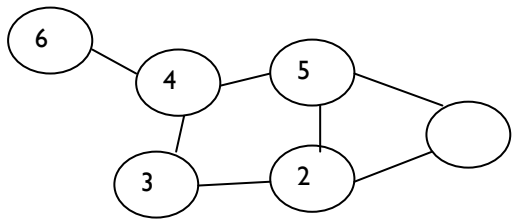

Figure 9 Vertex Cover problem (KSOU, Algorithm Analysis and Design, 2009)

The Vertex cover problem is the optimization problem of finding a vertex cover of size k in a given graph.

The description of the problem is given as follow:

Input Graph G.

Output: Smallest number K such that there is a vertex cover S for G of size K. Equivalently, the problem can be stated as a decision problem:

Instance : Graph G and positive integer K.

Question: Is there a Vertex cover S for G of size at most K.

Vertex cover is closely related to the independent set problem. A set of vertices S is a vertex cover if and only if its complement S' =V/S is an independent set. It follows that a graph with n vertices has a vertex cover of size K if and only if the graph has an independent set of size n-k. In this sense, the two problem are dual to each other.

Evaluator's Comments if any:

Question 10 Describe traveling salesmen problem?

Answer 10

The travelling sales man problem (TSP) is a problem in discrete or combinational optimization. It is a prominent illustration of a class of problems in computational complexity theory which are classified as NP-hard.

In the travelling salesman problem, which is closely related to the Hamiltonian cycle problem, a salesman must visit n cities. Modeling the

problem as a complete graph with n vertices, we can say that the salesman wishes to make a tour, visiting each city exactly once and to finishing at the city he started from. There is an integer cost c(I,j) to travel from city I to city j, and the salesman wishes to make the tour whose total cost is minimum, where the total cost is the sum 0f the individual costs along the edges of the tour.

Representing the cities by vertices and the roads between them by edges, we get a graph. In this graph, with every edge there is associated a real number such a graph is called a weighted graph, being the weight of edge.

In our problem, if each of the cities has a road to every other city, we have a complete weighted graph. This graph has numerous Hamiltonian circuits, and we are to pick the one that has the smallest sum of the distance.

The total number of different Hamilton circuits in a complete graph of n vertices can be shown to be (n-1)! /2. This follows from the fact that starting from any vertex we have n-1 edges to choose from the first vertex, n-2 from the second, n-3 from the third and so on, these being independent, result with (n-1) choices. Third number is, however, divided

by 2, because each Hamilton circuit has been counted twice.

Theoretically the problem of the travelling salesman can be always solved by enumeration all (n-1)!/2 Hamiltonian circuits, calculation the distance travelled in each, and then picking the shortest one. However for a large value of n, the labor involved is too great even for a digital computer.

The problem is to prescribe a manageable algorithm for finding the shortest route. No efficient algorithm for problems of arbitrary size has yet to be found ,although many attempts have been made. Science this problem has application in operations research, some specific large scale examples have been worked out. There are also available several heuristic methods of solution that give a route very close to the shortest one, but do not guarantee the shortest.

Evaluator's Comments if any:

Bibliography

Anonymous. (2009). *Binary Tree*. Retrieved 2009, from Wikipedia: http://en.wikipedia.org/wiki/Binary_Tree

Anonymous. (2009). *Buddy memory allocation*. Retrieved 2009, from Wikipedia: http://en.wikipedia.org/wiki/Buddy_memory_allocation

Anonymous. (2009). *Dijkstra's algorithm*. Retrieved 2009, from Wikipedia: http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Anonymous. (2009). *Spanning Tree*. Retrieved October 2009, from Wikipedia: http://en.wikipedia.org/wiki/Spanning_tree

KSOU. (2009). *Algorithm Analysis and Design.* Delhi: Virtual Education Trust.

KSOU. (2009). *Algorithm Analysis and Design.* Delhi: Virtual Education Trust.

Saia, J. (2004). *CS 362 Data Structures and Algorithms -II* . New Mexico: University of New Mexico.

Šaltenis, S. (2001). *Algorithms and Data Structures.* Denmark: Aalborg University.

Schildt, H. (2000). *C, The Complete Reference.* New York: McGraw-Hill.

The Memory Management Reference. (2009). Retrieved from The Memory Management Reference: http://www.memorymanagement.org/articles/begin.html