Unit 12 Greedy Method

Structure:

12.1 Introduction

In the previous units, we have discussed the different techniques of algorithms. Now we will study a well known method known as the greedy method. The greedy method is the most straightforward design technique, and it can be applied to a wide variety of problems. Most of these problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this is called an optimal solution. There is usually a simple way to determine a feasible solution but not necessarily an optimal solution.

Objectives:

After studying this unit, you should be able to:

- describe the greedy method approach
- apply the greedy method for optimistic storage on tapes, knapsack problem
- define the job sequencing problem using greedy method.

12.2 Greedy Method Strategy

The greedy method suggests that one can device an algorithm that works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this input is not added to the partial solution. Otherwise, it is added. The selection procedure itself is based on some optimization measure. This measure may be the objective function. This version of the greedy technique is called the subset paradigm.

We can describe the subset paradigm abstractly, but more precisely than above, by considering the control abstraction in Algorithm 12.1

Algorithm 12.1

```
1.  Algorithm Greedy (a,n)
2.  //a[1:n] contains the n inputs
3.  {
4.  Solution: =ϕ;//initialize the solution
5.  for i:=1 to n do
6.  {
7.  x: =select (a);
8.  if feasible (solution, x) then
9.  solution:=Union(solution, x);
10. }
11. return solution.
12. }
```

The function Select selects an input from a [ ] and removes it. The selected input's value is assigned to *x*. Feasible is a Boolean-valued function that determines whether *x* can be included into the solution vector. The function union combines x with the solution and updates the objective function. The function Greedy describes the essential way that a greedy algorithm will look, once a particular problem is chosen and the functions select, feasible and union are properly implemented.

Problems that do not call for the selection of an optimal subset, in the greedy method we make decisions by considering the inputs in some order. Each decision is made using an optimization criterion that can be computed using decisions already made. This version of the greedy method is called the ordering paradigm.

Self Assessment Questions

1. ———— is a Boolean-valued function that determines whether *x* can be included into the solution vector.

12.3 Optimistic Storage on Tapes

There are *n* programs that are to be stored on a computer tape of length *l*. Associated with each program *i* is a length l_i, $1 \le i \le n$. Clearly, all programs can be stored on the tape if and only if the sum of the length of the program is at most *l*.

We assume that whenever a program is to be retrieved from this tape, the tape is initially positioned at the front. Hence, if the programs are stored in the order $I=i_1, i_2 \text{-----} i_n$, the time t_j needed to retrieve program i_j is proportional to $\sum_{1\le k\le j} l_{ik}$. If all programs are retrieved equally often, then the expected or mean retrieval time (MRT) is $\left(\frac{1}{n}\right)\sum_{1\le j\le n} t_j$. In the optimal storage on tape problem, we are required to find a permutation for the n programs so that when they are sorted on the tape in this order the MRT is minimized. This problem fits the ordering paradigm.

Minimizing the MRT is equivalent to minimizing $d(I)=\sum_{1\le j\le n}\sum_{1\le k\le j} l_{ik}$

A greedy approach to building the required permutation would choose the next program on the basis of some optimization measure. One possible measure would be the *d* value of the permutation constructed so far. The next program to be stored on the tape would be one that minimizes the increase in *d*. If we have already constructed the permutation $i_1, i_2, \ldots.. i_r$, then appending program *j* gives the permutation $i_1, i_2, \ldots.., i_r, i_{r+1}=j$. This increases the d value by $\sum_{1\le j\le n} l_{ik} + l_j$. Since $\sum_{1\le k\le j} l_{ik}$ is fixed and independent of *j,* we trivially observe that the increase in *d* is minimized if the next program chosen is the one with the least length form among the remaining programs.

The greedy method simply requires us to store the programs in non-decreasing order of their lengths. This ordering can be carried out in O(n log n).

Theorem: If $l_1 \le l_2 \le \ldots\ldots \le l_n$, then the ordering $i_j=j,\ 1\le j \le n$, minimizes $\sum_{k=1}^{n} \sum_{j=1}^{k} l_{ij}$ over all possible permutations of the *ij.*

Proof: Let $I=i_1,i_2,\ldots..,i_n$ be any permutation of the index set $\{1, 2, \ldots..,n\}$.
Then

$$d(I) = \sum_{k=1}^{n} \sum_{j=1}^{k} l_{ij} = \sum_{k=1}^{n} (n-k+1) l_{jk}$$

If there exist *a* and *b* such that *a<b* and $l_{ia} > l_{ib}$, then interchanging i_a and i_b results in a permutation I' with

$$d(I') = \left(\sum_{\substack{k \\ k\ne a \\ k\ne b}} (n-k+1) l_{ik} \right) + (n-a+1) l_{ib} + (n-b+1) l_{ia}$$

Subtracting $d(I')$ from $d(I)$,we obtain

$$\begin{aligned} d(I) - d(I') &= (n-a+1)(l_{ia} - l_{ib}) + (n-b+1)(l_{ib} - l_{ia}) \\ &= (b-a)(l_{ia} - l_{ib}) \\ &> 0 \end{aligned}$$

Hence, no permutation that is not in non-decreasing order of the l_i*'s* can have minimum d. It is easy to see that all permutations in non-decreasing order of the l_i*'s* have the same *d* value. Hence, the ordering defined by $i_j=j$, $1\le j\le n$ minimizes the *d* value.

The tape storage problem can be extended to several tapes. If there are *m>1*, tapes, $T_0, \ldots.., T_{m-1}$, then the programs are to be distributed over these tapes. For each tape a storage permutation is to be provided If I_j is the storage permutation for the subset of programs on tape *j,* then $d(I_j)$ is as defined earlier. The total time $(TD)\ is \sum_{0\le j\le m-1} d(I_j)$. The objective is to store the programs in such a way as to minimize TD.

The obvious generalization of the solution for the one-tape case is to consider the program in non-decreasing order of l_i's. The program currently being considered is placed on the tape that results in the minimum increase in TD. This tape will be the one with the least amount of tape used so far. If there is more than one tape with this property, then the one with the smallest index can be used. If the jobs are initially ordered so that $l_1 \le l_2 \le ..... \le l_n$, then the first m programs are assigned to tapes $T_0, ...,T_{m-1}$ respectively. The next m programs will be assigned to tapes $T_0, ..., T_{m-1}$ respectively.

The General rule is that program i is stored on tape $T_{i \bmod m}$. On any given tape the programs are stored in non-decreasing order of their lengths. Algorithm 12.2 presents this rule in pseudocode. It assumes that the programs are ordered as above. It has a computing time of $\theta(n)$ and does not need to know the program lengths.

Algorithm 12.2

```
1.  Algorithm Store (n,m)
2.  //n is the number of programs and m the number
3.  // of tapes
4.  {
5.  j:=0; //Next tape to store on
6.  for i:=1 to n do
7.  {
8.  write(”append program”, i,”
9.  to permutation for tape”,j);
10. j:= (j+1)mod m,
11. }
12. }
```

Self Assessment Questions

2. A greedy approach to building the required permutation would choose the next program on the basis of some————————.
3. The ordering defined by $i_j=j$, $1 \le j \le n$, —————— the d value.

12.4 Knapsack Problem

Given *n* objects and a knapsack or bag. Object *i* has a weight w_i, and the knapsack has a capacity *m*. If a fraction x_i, $0 \le x_i \le 1$, of object *i* is placed into the knapsack, then a profit of $P_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is m, we require the total weight of all chosen objects to be atmost m. Formally, the problem can be stated as

Maxmise $\sum_{1 \le i \le n} P_i x_i$ -------------------- 1

Subject to $\sum_{1 \le i \le n} w_i x_i \le m$ ------------- 2

And $0 \le x_i \le 1, \ 1 \le i \le n$ ------------------ 3

The profits and weights are positive numbers. A feasible solution (or filling) is any set $(x_1, \ldots, x_n)$ satisfying 2 and 3 above. An optimal solution is a feasible solution for which 1 is maximized.

The Greedy knapsack algorithm requires *O(n)* time. The Greedy knapsack algorithm is given below.

Algorithm 12.3

```
Algorithm Greedyknapsack (m,n)
//p[1:n} and w[1:n] contain the profits and weights respectively
//of the n objects ordered such that p[i]/w[i]≥p[i+1]/w[i+1]
//m is the knapsack size and x[1:n] is the solution vector
{
for i:=1 to n do x[i]:=0.0;//Initialize x
u:=m;
for i:=1 to n do
{
if (w[i]>u) then break;
x[i]:=1.0; U:=U-w[i];
}
if (i ≤ n) then x[i]:=u/w[i];
}
```

Self Assessment Question

4. The profits and weights are ———— numbers.

12.5 Job Sequencing with Deadlines

We are given a set of *n* jobs. Associated with job *i* is an integer deadline $d_i \geq 0$ and a profit $p_i > 0$. For any job i the profit p_i is earned if the job in completed by its deadline. To complete a job, one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs. A feasible solution for this problem is a subset *J* of jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution *J* is the sum of the profits of the jobs in *J*, or $\sum_{i \in I} p_i$. An optimal solution is a feasible solution with maximum value. Since the problem involves the identification of a subset, it fits the subset paradigm.

Algorithm 12.4 demonstrates the greedy algorithm for sequencing unit time jobs with deadlines and profits

Algorithm 12.4

```
Algorithm JS (d,j,n)
//d[i]≥1, 1≤i≤n are the deadlines, n≥1. The jobs
// are ordered such that p[1] ≥p[2] ≥....≥ p[n].
//J[i] is the ith job in the optimal solution, 1≤i≤k.
//Also, at termination d [J[i]] ≤ d[J[i+1]], 1≤i≤k.
{
    d[0]:=J[0]:=0;//Initialize
    J[1]:=1 ;// include job 1
    K:=1;
    for i:=2 to n do
    {
        //Consider jobs in non-increasing order of p[i]
        //Find position for i and check feasibility for insertion
        r:=k;
        While((d[J[r]]>d[i]) and (d[J[r]]≠r)) dp r:= r – 1;
        if ((d[J[r]] ≤d[i]) and (d[i]>r)) then
        {
            // Insert i into J[ ].
            for q:=k to (r+1) step – 1 do J[q+1]:=J[q];
            J[r+1]:=i; k:=k+1;
        }
    }
    return k;
}
```

Theorem: If $p_1/w_1 \geq p_2/w_2 \geq \ldots . \geq p_n/w_n$. Then, Greedy knapsack algorithm generates an optimal solution to the given instance of the knapsack problem.

Proof: Let $x = (x_1, \ldots\ldots, x_n)$ be the solution generated by Greedy knapsack algorithm. If all the x_i equal one, then clearly the solution is optimal. So, let j be the least index such that $x_j \neq 1$. From the algorithm it follows that $x_i=1$ for $1 \leq i < j$, $x_i = 0$ for $j < i \leq n$ and $0 \leq x_j < 1$. Let $y=(y_1, \ldots\ldots, y_n)$ be an optimal solution we can assume that $\Sigma w_j y_i = m$. Let k be the least index such that $y_k \neq x_k$. Clearly, such a k must exist. It also follows that $y_k < x_k$. To see this, consider the three possibilities $k<j$, $k=j$ *or* $k>j$.

1. If $k<j$, then $x_k=1$, But, $y_k \neq x_k$, and so $y_k < x_k$.
2. $k=j$, then since $\Sigma w_i y_i = m$ and $y_i = x_i$ for $1 \leq i < j$. it follow that either $y_k < x_k$ or $\Sigma w_i y_i = m$
3. $k > j$, then $\Sigma w_i y_i = m$, and this is not possible.

Now, suppose we increase y_k to x_k and decrease as many of $(y_{k+1}, \ldots\ldots, y_n)$ as necessary so that the total capacity used is still on. This results in a new solution $z=(z_1, \ldots\ldots, z_n)$ with $z_i=x_i$, $1 \leq i \leq k$ and $\Sigma_{k<i\leq n} w_i\ (y_i - z_i) = w_k(z_k - y_k)$, Then, for z we have

$$\sum_{1\leq i\leq n} p_i z_i = \sum_{1\leq i\leq n} p_i y_i + (z_k - Y_k) w_k - \sum_{k\leq i\leq n} (y_i - z_i) w_i \frac{p_i}{w_i} \geq$$

$$\sum_{1\leq i\leq n} p_i y_i + \left[(z_k - y_k) w_k - \sum_{k\leq i\leq n} (y_i - z_i) \right] \frac{p_k}{w_k} = \sum_{1\leq i\leq n} p_i y_i$$

If $\sum p_i z_i > \sum p_i y_i$ then y could not have been an optimal solution. If these sums are equal, then either $z=x$ and x is optimal, or $z \neq x$. In the latter case, repeated use of the above argument will either show that y is not optimal or transform y into x and thus show that x too is optimal.

Greedy algorithm for sequencing unit time jobs with deadlines and profits For *JS,* there are two possible parameters in terms of which its complexity can be measured. We can use n, number of jobs and s, the number of jobs included in the solution J. The while loop of line 15 in Algorithm 12.4 is iterated at most K times. Each iteration takes $\theta(1)$ time. If the condition; of line 16 is true, then lines 19 and 20 are executed. These lines require $\theta(k-r)$

time to insert job *i*. Hence, the total number of time of each iteration, of for loop of line *10* is $\theta(k)$. This loop is iterated *n-1* times. If *s* is the final value of *k*, that is, *S* is the number of jobs in the final solution, then the total time needed by algorithm *JS* is $\theta\ (S_n)$. Since $S \leq n$, the worst-case time, as a function of *n* alone is $\theta(n^2)$. If we consider the job set $p_i = d_i = n - i + 1,\ 1 \leq i \leq n$, then algorithm *JS* takes $\theta(n^2)$ time to determine *J*. Hence. The worst-case computing time for *JS* is $\theta(n^2)$ In addition to the space needed for *d, JS* needs $\theta(S)$ amount of space for J.

Self Assessment Question

5. If ——————, then Greedy knapsack algorithm generates an optimal solution to the given instance of the knapsack problem.

12.6 Optimal Merge Pattern

When more than two sorted files are to be merged together, the merge can be accomplished by repeatedly merging sorted files in pairs. Thus, if files x_1, x_2, x_3, and x_4 are to be merged, we could first merge x_1 and x_2 to get a file y_i. then we could merge y_i and x_3 to get y_2. Finally, we could merge y_2 and x_4 to get the desired sorted file. Alternatively, we could first merge x_1 and x_2 getting y_i, then merge x_3 and x_4 and get y_2, and finally merge y_1 ad y_2 and get the desired sorted file. Given *n* sorted files, there are many ways in which we can pairwise merge them into a single sorted file. Different pairings require differing amounts of computing time. The problem we address ourselves to now is that of determining of an optimal way to pairwise merge *n* sorted files. Since this problem calls for an ordering among the pairs to be merged, it fits the ordering paradigm.

A greedy attempts to obtain an optimal merge path. Since merging of an *n*-record file and an *m*-record file requires possibly $n + m$ record moves, the obvious choice for a selection criterion is at each step merge the two smallest size files together. Thus, if we have files $(x_1, \ldots\ldots, x_5)$ with sizes *(20, 30, 10, 5, 30),* our greedy rule would generate the following merge pattern: merge x_4 and x_3 to get $z_1(|z_1|=15)$, merge z_1 and x_1 to get $z_2(|z_2| = 35)$ merge x_2 and x_5 to get $z_3\ (|z_3|=60)$ and merge z_2 and z_3 to get the answer z_4. The total number of record moves is *205*. One can verify that this is an optimal merge pattern for the given problem instance.

The merge pattern such as the one just described will be referred to as a two way merge pattern (each merge step involves the merging of two files). The two-way merge pattern can be represented by binary merge trees. Fig. 12.1 shows a binary merge tree representing the optimal merge pattern obtained for the above five files.

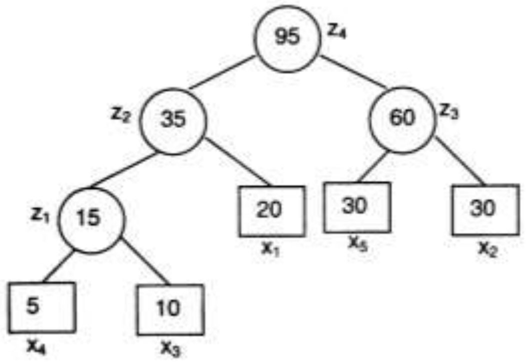


Fig. 12.1: Binary merge tree representing a merge pattern

The leaf nodes are drawn as squares and represent the given five files. These nodes are called external nodes. The remaining nodes are drawn as circles and are called internal nodes. Each internal node has exactly two children, and it represents the file obtained by merging the files represented by its two children. The number in each nodes is length (no. of records) of the file represented by that node.

The external node x_4 is at a distance of *3* from the root z_4 (a node at level *i* is at a distance of $i - 1$ from the root) Hence, the records of files x_4 are moved *3* times, once to get z_1, once again to get z_2, and finally, one more time to get z_4. If d_i is distance from the root to the external node for file x_i and q_i, the length of x_i is then the total number of record moves, for this binary merge tree it is $\sum_{i=1}^{n} d_i q_i$

This sum is called the weighted external path length of the tree.
tree node = record{
treenode*lchild; treenode * rchild;
integer weight;

In the algorithm below we discuss how to generate a two-way merge tree

Algorithm 12.5

```
1.  Algorithm Tree (n)
2.  //list is a global list of n single node
3.  //binary trees as described above
4.  {
5.  for i:=1 to n–1 do
6.  {
7.  pt:=new treenode; //Get a new tree node
8.  (pt→ /child):=Least(list);//Merge two trees with
9.  (pt.→ /child) :=Least(list) ;//smallest lengths
10. (pt→ weight):=((pt → lchild) → weight)
11. +((pt→rchild)→ weight);
12. Insert (list,pt):
13. }
14. return Least (list); //Tree left in list is the merge tree
15. }
```

An optimal two-way merge pattern corresponds to a binary merge tree with minimum weighted external path length. The function Tree of Algorithm 12.5 uses the greedy rule stated earlier to obtain a two-way merge tree for *n* files. The algorithm has as input a list of *n* trees.

Each node in a tree has three fields, /child, rchild and weight. Initially, each tree in list has exactly one node. This node is an external node and has *l*child and *r*child fields zero whereas weight is the length of one of *n* files to be merged. During the course of the algorithm, for any tree in list with root node *t, t*→ weight is the length of the merged file it represents (*t*→ weight equals the sum of the lengths of the external nodes in tree *t*). Function Tree uses two functions, Least (list) and Insert (list, *t*). Least (list) finds a tree in list whose root has least weight and returns a pointer to these trees. These trees are removed from list. Insert (list, *t*) inserts the tree with root *t* into list.

The main for loop in Algorithm 12.5 is executed $n - 1$ times. If list is kept in non-decreasing order according to the weight value in the roots, then Least (list) requires only $O(1)$ time and Insert (list, *t*) can be done in $O(n)$ time. Hence, the total time taken is $O(n^2)$.

Computing time for Tree is *O(n log n)*. Some speedup may be obtained by combining the Insert of line 12 with the Least of line 9.

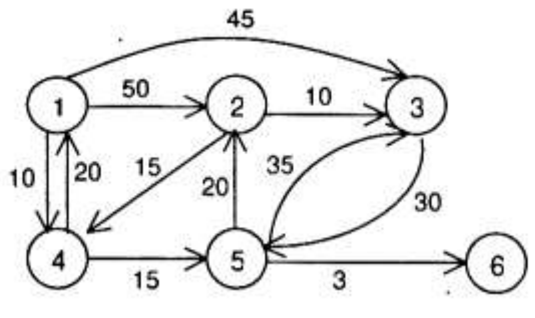
Self Assessment Questions

6. Each node in a tree has three fields ——, ——— and ——

12.7 Single Source Shortlist Paths

We are given a directed graph *G = (V, E),* a weighting function cost for the edges in a source vertex v_0. The problem is to determine the shortest paths from v_0 to all the remaining vertices of *G*. It is assumed that all the weights are positive. The shortest path between v_0 and some other node *v* is an ordering among a subset of the edges. Hence, this problem fits the ordering paradigm.

To formulate a greedy-based algorithm to generate the shortest paths, we must conceive of a multistage solution to the problem and also an optimization measure. One possibility is to build the shortest paths one by one. As an optimization measure, we can use the sum of the lengths of all paths so far generated. For this measure to be minimized, each individual path must be of minimum length. If we have already constructed *i* shortest path, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path. The greedy way to generate the shortest paths from v_0 to the remaining vertices is to generate these paths in non-decreasing order of path length. First, a shortest path to the nearest vertex is generated. Then a shortest path to the second nearest vertex is generated and so on.



Path	length
1) 1, 4	10
2) 1, 4, 5	25
3) 1, 4, 5, 2	45
4) 1, 3	45

Fig. 12.2: Graph & Shortest paths from vertex 1 to all destinations

For the graph of fig 12.2 the nearest vertex to v_0 = *1* is *4* (cost *[1,4] =10*). The path *1, 4* is the first path generated. The second nearest vertex to node

1 is *5* and the distance between *1* and *5* is *25*. The path *1, 4, 5* is the next path generated. In order to generate the shortest paths in this order, we need to be able to determine.

1) The next vertex to which a shortest path must be generated and
2) A shortest path to this vertex

Let S denote the set of vertices (including v_0) to which the shortest paths have already been generated. For *w* not in S, let *dist[w]* be the length of the shortest path starting from v_0, going through only those vertices that are in *S* and ending at *w*. We observe that:

1. If the next shortest path is to vertex *u*, then the path begins at v_0, and goes through only those vertices that are in *S*. To prove this, we must show that all the intermediate vertices on the shortest path to *u* are in *S*. Assume there is a vertex *w* on this path that is not in *S*. Then, the v_0 to *u* path also contains a path from v_0 to *w* that is of length less than the v_0 to *u* path. By assumption, the shortest paths are being generated in non decreasing order of path length, and so the shorter path v_0 to *w* must already have been generated. Hence, there can be no intermediate vertex that is not in *S*.

2. The destination of the next path generated must be that of vertex *u* which has the minimum distance, dist *[u]*, among all vertices not in *S*. This follows from the definition of dist and observation *1*. In case there are several vertices not in *S* with the same dist, then any of these may be selected.

3. Having selected a vertex *u* in observation *2* and generated the shortest v_0 to *u* path, vertex *u* becomes a member of *S*. At this point, the length of the shortest paths starting at v_0, going through vertices only in *S*, and ending at a vertex *w* not in *S* may decrease; that is, the value of dist *[w]* may change. If it does change, then it must be due to a shorter path starting at v_0 and going to *u* and then to *w*. The intermediate vertices on the v_0 to *u* path and the *u* to *w* path must all be in *S*. Further, the v_0 to *u* path must be the shortest such path; otherwise dist *[w]* is not defined properly. Also, the *u* to *w* path can be chosen so as not to contain any intermediate vertices. Therefore, we can conclude that if dist *[w]* is to change, then it is because of a path from v_0 to *u* to *w*, where, the path

from v_0 to *u* is the shortest path and the path from *u* to *w* is the edge *(u, w).* The length of this path is *dist [u] + cost [u, w].*

The above observations lead to a simple algorithm 12.6 for the single source shortest path problem. These algorithms (known as Dijkstra's algorithm), only determines the lengths of the shortest path from v_0 to all other vertices in *G*.

Algorithm 12.6

```
Algorithm shortest paths (v, cost dist, n)
// dist [j], 1 ≤ j ≤ n, is set to the length of the shortest
// path from vertex v to vertex in a digraph G with n
// vertices dist [v] is set to zero. G is represented by its
// cost adjacency matrix cost [1:n, 1:n]
{
    for i: = 1 to n do
    { // Initialise S
        S [i] : = false; dist [i] : = cost [v, i];
    }
    S[v]: = true; dist[v] : = 0.0; //put v in s
    for num:=2 to n – 1 do
    {
        // Determine n –1 paths from v
        choose u from among those vertices not
        in S such that dist [u] is minimum;
        S[u] : = true; //put u in S
        for (each w adjacent to u with S[w] = false) do
            // update distances
            if (dist [w] > (dist [u] + cost [u, w])) then
                dist [W]: = dist [u] + cost [u, W];
    }
}
```

Above algorithm is also known as Greedy algorithm to generate shortest paths.

In the function shortest paths (Algorithm 12.6) it is assumed that the *n* vertices of *G* are numbered *1* through *n*. The set *S* is maintained as a bit array with *S [i] = 0* if vertex *I* is not in *S* and *S [i] = 1*. It is assumed that the graph itself is represented by its cost adjacency matrix with cost *[i, j]'s* being the weight of the edge <*i, j*>. The weight cost *[i, I]* is set to some large number, ∞, in case the edge <*i, j*> is not in *E (G)*. For *i = j*, cost *[i, j]* can be set to any non negative number without affecting the outcome of the algorithm.

The time taken by the algorithm on graph with n vertices is $O(n^2)$. To see this, note that the for loop of line 7 in Algorithm 12.6 takes $\theta(n)$ time. The for loop of line *12* is executed *n – 2* times. Each execution of this loop requires *O(n)* time at lines 15 and 16 to select the next vertex and again at the for loop at line 18 to update dist. So the total time for this loop is $O(n^2)$. In case a list t of vertices currently not in S is maintained, then the number of nodes on this list would at any time be n-num. This would speed up lines 15 and 16 and the for loop of line 18, but the asymptotic time would remain $O(n^2)$.

Self Assessment Questions

7. The shortest path between v_0 and some other node *v* is an ——— among a subset of the edges.
8. The time taken by the algorithm on graph with n vertices is ————.

12.8 Summary

- In this unit we studied that the greedy method for optimally storage on tapes algorithm takes $\theta(n)$ computing time.
- The greedy method for knapsack problem takes $\theta(n^2)$ time to determine the solution.
- The greedy method for optimal merge pattern has O(nlogn) computing time for tree
- The greedy method for single source shortlist path problem has $\theta(n^2)$ asymptotic time.

12.9 Terminal Questions

1. Explain the Greedy Method
2. Explain the concept of Optimal Storage on Tapes
3. Briefly write in your own words the concept of Knapsack Problem
4. Explain the technique involved in Optimal Merge Pattern

5. Do you agree that the single source shortlist paths problem fits the ordering paradigm ? Justify your answer
6. Write an algorithm for single source shortest path problem
7. Write the Greedy-Knapsack algorithm

12.10 Answers

Self Assessment Questions

1. Feasible
2. Optimization measure
3. Minimizes
4. Positive
5. $p_1/w_1 \geq p_2/w_2 \geq \ldots.. \geq p_n/w_n$
6. lchild, rchild and weight.
7. ordering
8. O(n^2)

Terminal Questions

1. The greedy method suggests that one can device an algorithm that works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. (Refer Section 12.2)
2. We assume that whenever a program is to be retrieved from this tape, the tape is initially positioned at the front. Hence, if the programs are stored in the order $I=i_1$, i_2 -----i_n, the time t_j needed to retrieve program i_j is proportional to $\sum_{1 \leq k \leq j} l_{ik}$. If all programs are retrieved equally often, then the expected or mean retrieval time (MRT) is $\left(\frac{1}{n}\right)\sum_{1 \leq j \leq n} t_j$. (Refer Section 12.3)
3. Refer Section 12.4
4. Refer Section 12.6
5. Refer Section 12.7
6. Refer Section 12.7
7. Refer Section 12.4