Unit 13 Dynamic Programming

Structure:

13.1 Introduction

In the previous unit, we discussed the concept of greedy method but when the optimal solutions to subproblems are to be retained in order to avoid recomputing their values, we use dynamic programming. Dynamic programming is an algorithm design method that can be used when the solution to problem can be viewed as the result of sequence of decisions.

The principle of optimality states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.

The essential difference between the greedy method and dynamic programming is that in the greedy method only one decision sequence is ever generated. In dynamic programming, many decision sequences may be generated. However, sequences containing suboptimal subsequences cannot be optimal and so not be generated.

Objectives

After studying this unit, you should be able to:

- apply Dynamic Programming in Multistage graphs
- solve all pair shortlist paths using Dynamic Programming
- solve Traveling Salesman Problems using Dynamic Programming.

13.2 Dynamic Programming Strategy

Because of the use of the principle of optimality, decision sequences containing subsequences that are suboptimal are not considered. Although the total number of different decision sequences is exponential to the number of decisions (if there are d choices for each of the n decisions to be made then there are d^n possible decision sequences). Dynamic programming algorithms often have a polynomial complexity.

Another important feature of the dynamic programming approach is that optimal solutions to subproblems are retained so as to avoid recomputing their values. The use of these tabulated values makes it natural to recast the recursive equations into an iterative algorithm.

Self Assessment Question

1. Dynamic programming algorithms often have a ————————

13.3 Multistage Graphs

A multistage graph $G = (V,E)$ is a directed graph in which the vertices are partitioned into $K \geq 2$ disjoint sets V_i, $1 \leq i \leq k$. In addition, if $<u, v>$ is an edge in E, then $u \in v_i$, and $v \in v_{i+1}$ for some i, $1 \leq i < k$. The sets V_1 and V_k are such that $|V_1| - |V_k| = i$ Let s and t, respectively, be the vertices in V_1 and V_k. The vertex s is the source, and t the link. Let $c(i, j)$ be the cost of edge $<i, j>$. The cost of a path from s to t is the sum of the costs of the edges on the path. The multistage graph problem is to find a minimum-cost path from s to t. Each set V_i defines a stage in the graph. Because of the constraints, on E, every path from s to t starts in stage 1, goes to, stage 2, then to stage 3, then to stage 4, and so on, and eventually terminates in stage K.

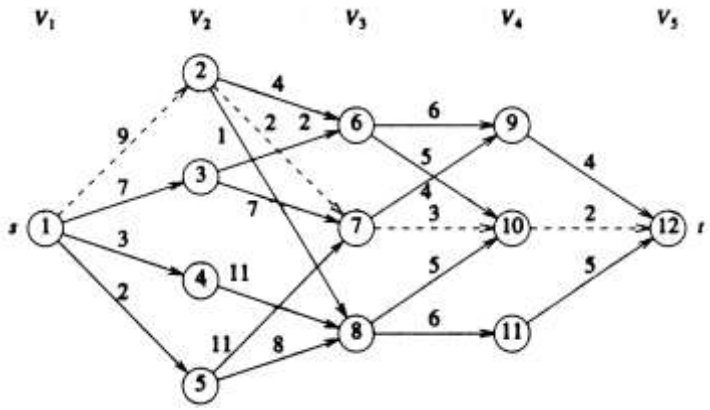

Fig. 13.1: A minimum-cost *s* to *t* path is indicated by the broken edges.

Many problems can be formulated as multistage graph problems. We give only one example. Consider a resource allocation problem in which *n* units of resource are to be allocated to *r* projects. If *j* $0 \le j \le n$ units of the resource are allocated to project *i*, then the resulting net profit is *N(i,j).* The problem is to allocate the resource to the *r* projects in such a way as to maximize total net profit. This problem can be formulated as an *r+1* stage graph problem as follows:

Stage *i*, $1 \le i \le r$, represents project *i*. There are *n+1* vertices *V(i, j), 0* $\le j \le n$, associated with stage *i,* $2 \le i \le r$. Stages *1* and *r+1* each have one vertex, *V(1,0) = s* and *V(r+1,n)=t*, respectively. Vertex *V(i,j),* $2 \le i \le r$, represents the state in which a total of *j* units of resource have been allocated to projects *1,2, ….. ,i –1*. The edges in G are of the form <*V(i,j),V(i+1, l)*> for all $j \le l$ and $1 \le i < r$. The edge <*V(i,j, V(i+1, l)>, j<l,* is assigned a weight or cost of *N(i, l-j)* and corresponds to allocating *l-j* units of resource to project *i,* $1 \le i < r$. In addition, *G* has edges of the type <*v(r,j), v(r+1,n)>.* Each such edge is assigned a weight of $\max_{0 \le p \le n-j} \{N(r,p)\}$. It should be easy to see that an optimal allocation of resources is defined by a maximum cost *s* to *t* path. This is easily converted into a minimum-cost problem by changing the sign of all the edge costs.

A dynamic programming formulation for a k-stage graph problem is obtained by first noticing that every *s* to *t* path is the result of a sequence of *k – 2*

decisions. The i^{th} decision involves determining which vertex in V_{i+1}, $1 \le i \le k - 2$, is to be on the path. It is easy to see that the principle of optimality holds. Let *p(i,j)* be a minimum-cost path from vertex *j* in V_i of vertex *t*. Let cost*(i,j)* be the cost of this path. Then, using the forward approach, we obtain

$$cost(i,j) = \min_{\substack{l \in V_{j+1} \\ \langle j,l \rangle \in E}} \{c(j,l) + cost(i+1,l)\} \ldots\ldots\ldots\ldots 13.1$$

Before writing an algorithm for a general *k*-stage graph, let us impose an ordering on the vertices in *V*. This ordering makes it easier to write the algorithm. We require that the *n* vertices in *V* are indexed *1* through *n*. Indices are assigned in order of stages. First, *s* is assigned index *1*, then vertices in V_2 are assigned indices, then vertices from V_3, and so on. Vertex *t* has index *n*. Hence, indices assigned to vertices in V_i. As a result of this indexing scheme, cost and *d* can be computed in the order *n-1, n-2..., 1*. The first subscript in cost, *p* and *d* only identifies the stage number and is omitted in the algorithm. The resulting algorithm in pseudocode is *F* Graph, given in Algorithm 13.1

Algorithm 13.1

```
1.  Algorithm FGraph (G,k,n,p)
2.  //The input is a K-stage graph G=(V,E) with n vertices
3.  //indexed in order of stages. E is a set of edges and c[i,j]
4.  //is the cost of <i, j> p[1:k] is a minimum-cost path.
5.  {
6.  cost[n] :=0.0;
7.  for j:=n-1 to 1 step – 1do
8.  {//compute cost[j]
9.  Let r be a vertex such that <j,r> is an edge
10. of G and c[j,r]+cost[r] is minimum:
11. cost[j] :=c[j, r]+cost[r]:
12. d[j]:r;
13. }
14. //Find a minimum-cost path
15. p[i]:=1; p[k]:=n;
16. for j:=2 to k-1 do p[j]:=d [p-1]];
17. }
```

In the above algorithm, we study multistage graph pseudocode corresponding for the forward approach.

The complexity analysis of the function *F* Graph is fairly straightforward. If *G* is represented by its adjacency lists, then *r* in line 9 of Algorithm 13.1 can be found in time proportional to the degree of vertex *j*, Hence, if *G* has *IEI* edges, then time for the for loop of line 7 is θ*(IVI+IEI)*. The time for the loop of line 16 is θ*(K)*. Hence, the total time is θ*(IVI+IEI)*. In addition to the space needed for input, space is needed for cost*[ ]*, *d[ ],* and *p[ ].*

The multistage graph problem can also be solved using the backward approach. Let *bp(i,j)* be a minimum-cost path from vertex *s* to a vertex *j* in V_i. Let bcost *(I,j)* be the cost *bp(i,j).* From the backward approach we obtain.

$$bcost(i,j) = \min_{\substack{l \in V_{i-1} \\ \langle l,j \rangle \in E}} \{bcost(i-1,l) + c(l,j)\} \quad \text{.....13.2}$$

Algorithm 13.2

```
1.  Algorithm Bgraph (G, k, n, p)
2.  // same function as Fgraph
3.  {
4.  bcost[1]:= 0.0;
5.  for j:=2 to n do
6.  {// compute bcost [j]
7.  let r be such that <r,j> is an edge of
8.  G and bcost [r] + c [r,j] is minimum;
9.  bcost [j]: = bcost [r] + c[r,j];
10. d[j]:=r;
11. }
12. // Find a minimum-cost path
13. p[1]:=1;p[k]:=n;
14. for j:=k-1 to 2 do p[j]:=d[p [j+1]];
15. }
```

In the above algorithm we study, the multistage graph pseudocode complexity to backwards approach

This algorithm has the same complexity as *F* graph provided *G* is now represented by its inverse adjacency lists (i.e., for each vertex *v,* we have a list of vertices *w* such that $<w,v> \in E$).

It should be easy to see that both *F* graph and *B* graph work correctly even on a more generalized version of multistage graphs. In this generalization, the graph is permitted to have edges $<u.v>$ such that $u \in V_i$, $V \in V_1$ and $i<j$.

Self Assessment Question

2. A multistage graph *G = (V,E)* is a directed graph in which the vertices are ————————into $K \geq 2$ disjoint sets V_i, $1 \leq i \leq k$.

13.4 All Pair Shortest Paths

Let *G = (V, E)* be a directed graph with *n* vertices and let cost be a cost adjacency matrix for *G* such that cost *(i, i) = 0*, $1 \leq i \leq n$. Then cost (I,j) is the length (or cost) of edge $<i,j>$ if $<i,j> \in E(G)$ and cost *(i,j)* = ∞ if $i \neq j$ and $<i,j> \notin E(G)$. The all pairs shortest path problem is to determine a matrix *A* such that *A (i,j)* is the length of a shortest path from *i* to *j*.

Let us examine a shortest *i* to *j* path in $G_i \neq j$. This path originates at vertex *i* and goes through some intermediate vertices (possibly none) and terminates at vertex *j*. We can assume that this path contains no cycles for if there is a cycle, and then this can be deleted without increasing the path length. If *k* is an intermediate vertex on this shortest path, then the subpaths from *i* to *k* and from *k* to *j* must be shortest paths from *i* to *k* and *k* to *j* respectively. Otherwise, the *i* to *j* path is not of minimum length. So the principle of optimality holds. This alerts us to the prospect of using dynamic programming. If *k* is the intermediate vertex with highest index, then the *i* to *k* path is a shortest *i* to *k* path in *G* going through no vertex with index greater than *k–1*. Similarly, the *k* to *j* path is a shortest *k* to *j* path in *G* going through no vertex of index greater than *k–1*. We can regard the construction of a shortest *i* to *j* path as first requiring a decision as to which is the highest indexed intermediate vertex *k*. once this decision has been made, we need to find two shortest paths, one from *i* to *k* and the other from *k* to *j*. Neither of these may go through a vertex with index greater than *k–1*.Using $A^k(i,j)$ to represent the length of a shortest path from *i* to *j* going through no vertex of index greater than *k*, we obtain

$$A(i,j) = min\left\{ \min_{1 \le k \le n} \{ A^{k-1}(i,k) + A^{k-1}(k,j), cost(i,j) \right\} \ldots\ldots\ldots 13.3$$

Clearly, $A°(i,j)$ = cost (i, j), $1 \le i \le n$, $1 \le j \le n$. We can obtain a recurrence for $A^k(j,j)$ using argument similar to that used before. A shortest path from i to j going through no vertex higher than k either goes through vertex k or it does not. If it does, $A^k(i,j) = A^{k-1}(i,k) + A^{k-1}(k, j)$. If it does not, then no intermediate vertex has index greater than $k-1$. Hence,

$A^k(i,j)=A^{k-1}(i, j)$. Combining, we get

$A(i,j)=min\{A^{k-1}(i, j), A^{k-1}(i,k)+A^{k-1}(k,,j)\}, k>1$ ----------13.4

Recurrence (5,4) can be solved for A^n by first computing A^1, then A^2, then A^3 and so on. Since there is no vertex in G with index greater than n, $A(i,j) = A(i,j)$

Algorithm Allpaths (cost, A, n)

Algorithm 13.3

```
1.  //cost[1:n, 1:n] is the cost adjacency matrix of a graph
2.  //with n vertices; A[I, j] is the cost of a shortest path from
3.  //vertex i to vertex j. cost [I, j] = 0.0, for 1 ≤ i ≤ n.
4.  {
5.      for i:=1 to n do
6.          for j:=1 to n do
7.              A[i, j]:=cost [i, j]; //copy cost into A.
8.      for k:=1 to n do
9.          for i:=1 to n do
10.             for j:=1 to n do
11.                 A[i, j]:=min(A[i, j], A[i, k]+A[k, j];)
12. }
```

Above algorithm functions to compute the length of shortest paths

Function All paths computer $A^n(i.j)$. The computation is done in place so the superscript on A is not needed. The reason this computation can be carried out in-place is that $A^k(i,k) = A^{k-1}(i, j)$ and $A^k(k, j) = A^{k-1}(k, j)$. Hence, when A^k is formed, the k^{th} column and row do not change. Consequently, when $A^k(i,j)$ is computed in line II of Algorithm 13.3, $A(i,k) = A^{k-l}(i,k) = A^k(i, k)$ and $A(k,j) = A^{k-l}(kJ) = A^k(k,j)$. So, the old values on which the new values are based do not change on this iteration.

Let $M = max \{cost (i,j) | <i,j> \in E (G)\}$. It is easy to see that $A^n (i,j) \leq (n-1)M$. From the working of all paths, it is clear that if $<i,j> \notin E (G)$ and $i \neq j$, then we can initialize cost $(i,,j) > (n-I) M$,. If, at termination, $A(i,j) > (n-1)$, then there is no directed path from i to j in G.

The time needed by All paths (Algorithm 13.3) is especially easy to determine because the looping is independent of the data in the matrix A. Line 11 is iterated n^3 times, and so the time for all paths is $\theta (n^3)$.

Self Assessment Question

3. The time needed by All paths is easy to determine, because the looping is ———————— of the data in the matrix .

13.5 Travelling Salesman Problem

A tour of G is a directed simple cycle that includes every vertex in V. The cost of the tour is the sum of the cost of the edges on the tour. The travelling salesperson problem is to find a tour of minimum cost.

Let $G=(V,E)$ be a directed graph with edges cost c_{ij}. The variable c_{ij} is defined such that $c_{ij} = 0$, $c_{ij} > o$ for all i and j and $c_{ij} = \infty$ if $< i,j> \notin E$. Let $IVI = n$ and assume $n>1$. A tour is a simple path that starts and ends at vertex 1. Every tour consists of an edge $< 1, k>$ for some $k \in V — \{1\}$ and a path from vertex k to vertex 1. The path from vertex k to vertex 1 goes through each vertex in $V-\{1,k\}$ exactly once. It easy to see that if the tour is optimal, then the path from k to 1 must be a shortest k to 1 path going through all vertices in $V– \{1,k\}$. Hence, the principle of optimality holds. Let $g (i,s)$ be the length of a shortest path starting at vertex i, going through all vertices in S, and terminating at vertex 1. The function $g(1, V-\{1\})$ is the length of an optimal salesperson tour. From the principal of optimality if follows that

$$g(1,V-\{1\}) = \min_{2 \leq k \leq n} \{c_{1k} + g(k,V-\{j,k\})\} \ldots\ldots\ldots 13.5$$

Generalizing (13.5), we obtain (for $i \notin s$)

$$g(i,S) = \min_{j \in S} \{c_{ij} + g(j,S-\{j\})\} \ldots\ldots\ldots\ldots 13.6$$

Equation 13.5 can be solved for $g (1, V- \{1\})$ if we know $g (k, V-\{1,k\})$ for all choices of k. The g values can be obtained by using (13.6). Clearly, $g (i,\phi) = C_{i1}$, $i \leq 1 \leq n$.

Hence, we can use (13.6) to obtain *g(i,S)* for all *S* of size *1*. Then, we can obtain *g (i, s)* for *S* with |*S*|=*2*, and so on. When |*S*|< *n-1*, the values of *i* and *S* for which *(i, S)* is needed are such that
$i \neq 1$, $1 \notin S$, and $i \notin S$.

Let *N* be the number of *g (i, S)'s* that have to be computed before (13.6) can be used to compute *g (i,v-{i}).* For each value of |*S*| there are *n-1* choices for i. The number of distinct sets *S* of size *k* not including *l* and *i* is (k^{n-2}). Hence,

$$N = \sum_{k=0}^{n-2} (n-1)\left(k^{n-2}\right) = (n-1)2^{n-2}$$

An algorithm that proceeds to find an optimal tour by using (13.5) and (13.6) will require $\theta(n^2 2^n)$ time by using computation of *g(i,s)* with |*S*|=*k* ,require *k-1* comparisons when solving (13.6). This is better than enumerating all n! different tours to find best one. The most serious drawback of this dynamic programming solution is, $O(n2^n)$. This is too large even for modest values of n.

Self Assessment Question

4. A ———— of *G* is a directed simple cycle that includes every vertex in *V.*

13.6 Summary

- In this unit, we studied that Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence decisions.
- Optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision, this is called as the principle of optimality.
- We also studied the concept of travelling salesmen problems using the concept of graphs.

13.7 Terminal Questions

1. Explain the concept of Multistage Graphs
2. Explain the Concept of travelling Salesman Problem
3. Write an Algorithm to obtain a minimum cost s-t path

13.8 Answers

Self Assessment Questions

1. Polynomial complexity
2. partitioned
3. independent
4. tour

Terminal Questions

1. A multistage graph $G = (V,E)$ is a directed graph in which the vertices are partitioned into $K \geq 2$ disjoint sets V_i, $1 \leq i \leq k$. In addition, if $<u, v>$ is an edge in E, then $u \in v_i$, and $v \in v_{i+1}$ for some i, $1 \leq i < k$. The sets V_1 and V_k are such that $|V_1| - |V_k| = i$ Let s and t, respectively, be the vertices in V_1 and V_k. The vertex s is the source, and t the link. Let $c(i, j)$ be the cost of edge $<i, j>$. The cost of a path from s to t is the sum of the costs of the edges on the path. The multistage graph problem is to find a minimum-cost path from s to t. Each set V_i defines a stage in the graph. (Refer Section 13.3)
2. Refer Section 13.5
3. Refer Section 13.3