CS430

LECTURE NOTE NO.4

DYNAMIC POGRAMMING

The technique of Dynamic Programming is based on the following principle:

Principle of Optimality: An optimal sequence of decisions has the property that whatever the decision until the current state the remaining decisions must constitute an optimal decision sequence with respect to the current state.

Consider the Search Tree approach to solving problems. At every node of the tree we have a partial solution, say $PS(v)$. Each node may be characterized by a problem state, i.e $\mathcal{S}(v)$. For example, in the Subset-Sum problem, the state of the problem is characterized by the partial solution and the remainder set of choices. By the princple of optimality, the optimal solution to be found in the subtree $ST(v)$ is dependant only on the current state and is independant of how we arrived at the state, $\mathcal{S}(v)$. This fact may be used to improve the efficiency of the search algorithm as follows: If we encounter a state for which we have already computed the optimal decision sequence then we need not recompute this decision. In the search tree, if we encounter two nodes v_1 and v_2 such that $\mathcal{S}(v_1) = \mathcal{S}(v_2)$ with v_1 occuring before v_2 then we need not explore the search tree $ST(v_2)$.

We use this strategy to devise efficient algorithms. In the designs we will use a recursive definitions of the problem solution.

1 Knap-Sack Problem

Consider the Knap-Sack problem, *Knap-Sack*$(\{a_1 \dots a_n\}, M, S, P)$ where S is the size function and P the profit function on the set of elements $\{a_1.a_2 \dots a_n\}$. The state at any step is characterized by the elements left for consideration and the amount of space left in the knapsack.

We denote the solution of the problem, *Knap-Sack*$(\{a_1 \dots a_n\}, M, S, P)$ by $KP(1, M)$. At a node v of the search tree, the problem required to be solved is the Knap-Sack problem with a set of elements $\{a_i ... a_n\}$ and a remainder size M of the Knapsack. The partial solution $PS(v)$ uses elements in $\{a_1 .. a_{i-1}\}$. This problem is represented by $KP(i, M')$. It suffices to solve the various problems $KP(j, w), 1 \leq j \leq n, 1 \leq w \leq M$ that arise.

The solution of a problem at a node in the search tree may be recursively expressed as:

$$KP(i, M') = \max(KP(i+1, M' - s_i) + p_i, KP(i+1, M'))$$

$$KP(n, M') = p_n \quad if s_n \leq M'$$

$$KP(n, M') = 0 \quad if s_n > M'$$

To use the principle of optimality effectively, we construct a 2-dimensional table, $\mathcal{T}$, to store solutions of problems already considered. The rows of the table are indexed by $i, 1 \leq i \leq n$ and the columns by $w, 1 \leq w \leq M$. The entry in the cell $\mathcal{T}(i,j)$ is the solution of the problem $KP(i,j)$. The entry in $\mathcal{T}(i,j)$ is computed from entries in $\mathcal{T}(i+1, M'-s_i)$ and $\mathcal{T}(i+1, M')$ using the recurrence above. The initial entries corresponding to $\mathcal{T}(n,j), 1 \leq j \leq M$ can be obtained by inspection.

This method of computing the solution can be interpreted as evaluation of the search tree in a bottom-up fashion, i.e. the solution at nodes at distance $i, i = 1, 2\ldots$ from the leaf nodes is computed in that sequence.

The time complexity of the algorithm is the time required to fill up the table. Thus

Lemma 1.1. *The Knap-Sack Problem can be solved, using Dynamic Programming, in $O(nM)$ steps.*

2 Travelling Salesman Problem

We assume a labeling of the vertices of the graph $G = (V, E)$ which is a complete graph with a weight function $w : E \Rightarrow Z^+$. Further let us assume that $|V| = n$.

We assume that our solution starts and end at vertex 1. For this problem, the state at every node of the search tree can be characterized by the problem $TSP(i, V' = \{v_{i1}, v_{i2}..v_{ij}\})$ which requires the finding of a tour from i to 1 using vertices only in the set V'. We call the set V' the *working set.* We let the cost of the optimal solution be represented by the function $TSP(i, V' = \{v_{i1}, v_{i2}..v_{ij}\})$ itself.

In order to find the optimal tour which starts with vertex v_i and ends at the vertex labelled 1, using vertices only in the set V', we investigate the choices available at the first step. The first vertex after v_i is one of the vertices in the set V'. There are $|V'|$ choices for this vertex. A choice, say v', generates a subproblem where we have to start with v' and reach vertex 1 using vertices only in $V' - \{v'\}$.

The following recurrence characterizes the solution to the problem.

$$\begin{aligned} TSP(i, V' = \{v_{i1}, v_{i2}\ldots v_{ij}\}) \quad = \min \quad & \{TSP(v_{i1}, \{v_{i2}, ..v_{ij}\}) + w(v_i, v_{i1}) \\ & TSP(v_{i2}, \{v_{i1}, v_{i3}, ..v_{ij}\}) + w(v_i, v_{i2}), \ldots \\ & TSP(v_{ij}, \{v_{i1}, v_{i2}....v_{i(j-1)}\}) + w(v_i, vij)\} \end{aligned}$$

$$TSP(i, V' = \Phi) = w(i, 1)$$

Note that the solution to a problem is obtained only from the solutions to problems with smaller size working sets.

The Dynamic Programming implementation of the above scheme starts with the solution of subproblems with working set of size 0.

```
Algorithm $DP\text{-}TSP(i, V')$
    If $|V'| = 0$ Then return cost of the edge $(i, 1)$
        Else
          $result \leftarrow 0$
          Let $V' = \{v_{i1}, v_{i2}\ldots v_{ij}\}$;
        For k = 1 to j do
            $result \leftarrow \min(result, DP\text{-}TSP(v_{ik}, V' - \{v_{ik}\}))$
        return $result$
End.
```

To estimate the time complexity, we need to compute the total number of subproblems that arise during the computations. Each subproblem is characterized by a pair with one element being a vertex and the other being a set $V', V' \subseteq V$. The number of subsets are $O(2^n)$. And thus there are $O(n2^n)$ subproblems.

Moreover, to compute the solution of one subproblem requires the solution of $O(n)$ smaller subprblems and can be accomplished in $O(n)$ time given the solutions to the smaller subproblems. Thus

Lemma 2.1. *The Travelling Salesman Problem can be solved using Dynamic Programming in $O(n^2 2^n)$ steps.*

3 Optimal Triangulations

Consider the problem of triangulating a convex polygon of n vertices with $n-3$ non-intersecting diagonals.

Let $C = (v_1, v_2 \dots v_n)$ be the convex polygon. We first present a recurrence which allows us to compute the value of the optimal triangulation, $OPT(C)$, using optimal solutions to subpolygons.

Consider the edge $e = (v_1, v_n)$ in the convex polygon. In any triangulation, e is part of one of the triangles $(v_1, v_2, v_n), (v_1, v_3, v_n) \dots (v_1, v_{n-1}, v_n)$. Since the optimal triangulation comprises one of these triangles, consider the choice of one such triangle (v_1, v_i, v_n). This divides the polygon into two smaller convex polygons, $C_1 = (v_1, v_2 \dots v_i)$ and $C_2 = (v_i, v_{i+1} \dots v_n)$. Using the principle of Dynamic Programming, these subproblems must be solved optimally to get an optimal solution given the choice of this first triangle. Thus

$$OPT(C) = \min_{2 \le i \le n-1} \{OPT(C_1) + OPT(C_2) + d(1,i) + d(i,n)\}$$

$$OPT(C) = 0 \quad ifn = 1, 2, 3$$

where $d(i,j)$ is the length of the diagonal $d(i,j)$ and $d(i,i+1) = 0$

Again we have defined subproblems, characterized by a sequence of vertices $(v_i, v_{i+1} \dots v_j)$, the edge (v_j, v_i) being the last edge in the convex subpolygon. The number of such subproblems is $O(n^2)$, the number of contiguous subsequences obtained by a pair of vertices (i,j). A subproblem with a convex polygon of size s is solved using solutions to subproblems with convex polygons of smaller size. We thus start constructing solutions of problems with convex polygons which are of 4 vertices. At each phase subproblems with convex polygons of size one greater are solved. The solution sequencing is illustrated in figure 1 where entries in the upper triangular portion of the table are to be filled. At each step we process table squares which lie on the diagonal $i - j = c$ having processed table entries such that $i - j < c$. This diagonal represents subproblems of size $c + 1$. For $c = 0, 1, 2$ the entries are already determined.

Each subproblem can be solved in $O(n)$ steps by evaluating the above recurrence given the solution of the smaller subproblems. Since there are $O(n^2)$ subproblems to be solved, the total time required to solve this problem is $O(n^3)$.

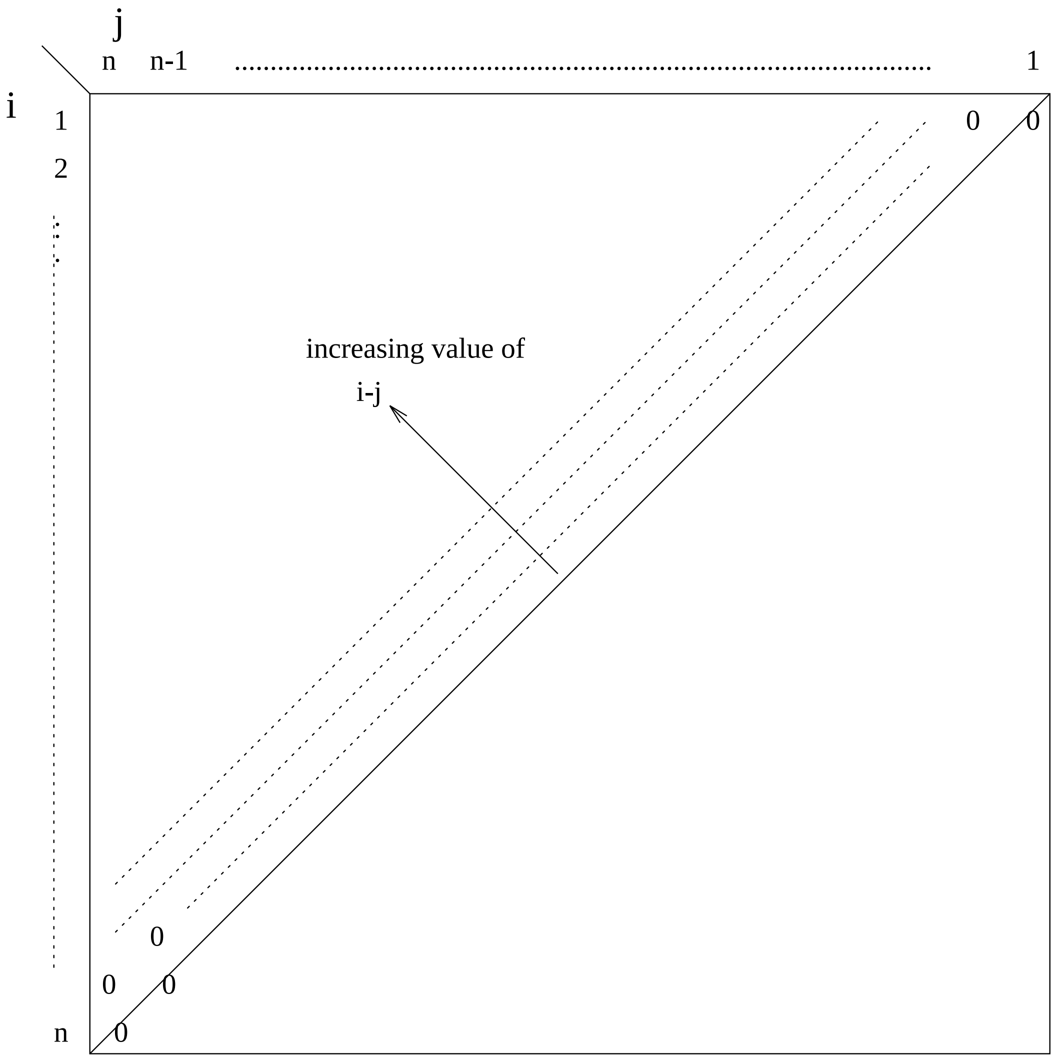

Figure 1: Dynamic Programming Method to solve the Triangulation Problem

4 Shortest Paths

Given a directed graph $G(V, \uparrow E)$ with a cost function on the edges $c : E \to Z$, we define the cost of a path $P = (e_1, e_2 \dots e_k)$ to be $\Sigma_i c(e_i)$. The following shortest path problems are of interest:

- Single Source Shortest Path (SSSP) Problem: Given a start vertex $s \in V$ find the least cost path from s to all other vertices in the graph.
- All Pairs Shortest Path (APSP) Problem: Find the shortest path between all pairs of vertices in V.

We will use dynamic programming principles to solve this problem. The fundamental recurrence for this problem may be expressed as follows:

$$SP(u,v) = min_{w \in Adj(v)}\{SP(w,v) + c(u,w)$$

where $SP(u,v)$ is the shortest path from u to v and $c(u,w)$ the cost of the edge (u,v).

4.1 SSSP Problem

In this problem we will consider two cases; one where all the edge costs are positive and the other where no restriction is imposed on the edge costs. Applying the fundamental recurrence to this problem we get the following useful recurrences:

$$SP(s,v) = min_{w \in Adj(v)}\{SP(s,w) + c(w,v)$$

$$SP(s,v) = min_{w \in Adj(s)}\{c(s,w) + SP(w,v)$$

In order to determine $SP(s,v)$ by dynamic programming we would require $SP(s,w)$. A priori there is no way of ordering the vertices so as to be able to solve the above recurrence. However, the following lemma is of use:

Lemma 4.1. *Let $l(w) = c(s,w)$ be a label of vertex w where $(s,w) \in E$. Furthermore $l(w) = \infty$ if no edge (s,w) exists. Then $\exists w$ such that $SP(s,w) = l(w)$.*

Proof: Consider a vertex w' and the shortest path to w'. This path must use a vertex, say v, adjacent to s as the first vertex on the path. Since the shortest path to w' must use the shortest path to w the edge (s,v) is the shortest path to v from s. The vertex v is the required vertex w. ■

Non-Negative Cost Edges

We will now consider the case when all the weights in the graph are non-negative.

Lemma 4.2. *Let $l(w) = c(s,w)$ be a label of vertex w where $(s,w) \in E$. Furthermore $l(w) = \infty$ if no edge (s,w) exists. Let w be the vertex with the smallest label. Then $SP(s,w) = l(w)$.*

The lemma follows from the fact that the label value asssigned to the vertex with the smallest label will not decrease since all edge costs are non-negative.

This lemma can be extended to the following by an analogous proof:

Lemma 4.3. *Let S be a set of vertices to which the shortest path from s has been found. Let $l(v), v \in S$ denote the cost of the shortest path $SP(s,v)$. Let $l(w), w \in V - S$ be the label of vertices such that $l(v)$ is the cost of the shortest path to v from s using vertices in S. Let $w \in V - S$ be the vertex with the smallest label. Then $SP(s,w) = l(w)$.*

This leads to the following algorithm developed by Dijkstra:

```
Algo ShortP(G,s)
    $S \leftarrow \emptyset$;
    $l(s) \leftarrow 0$; $l(v) \leftarrow \infty, v \in V - \{s\}$
    Repeat
        begin
            Find vertex $v$ with least label in $V - S$;
            $SP(s,v) \leftarrow l(v)$;
            $S \leftarrow S + \{v\}$;
            /* Update labels of vertices in $V - S$ */
            $l(w) \leftarrow min\{l(w), c(v,w) + SP(s,v)\}$;
        end
    Until $|S| = n$;
```

An implementation of this scheme using Fibonacci heaps gives an $O(E + VlogV)$ algorithm for finding single source shortest paths in graphs where the edge costs are non-negative.

Arbitrary Edge Costs

For graphs with arbitrary edge costs Lemma 4.1 assures us that at least one vertex has $Sp(s,v)$ determined after labels have been computed from the source. Determining this vertex is not an easy task. The algorithm developed by Bellman-Ford does not even attempt to do this. Instead it relies on the fact that vertices w which have v as the adjacent vertex in $SP(w)$ will have their shortest paths determined after $SP(v)$ has been found. Thus at successive steps the number of vertices to which the shortest path has been found increases.

The algorithm is simple to state: Apply the following relaxation step n times:

$$\forall e = (u,v) \in E \;\; bfdol(v) \leftarrow min\{l(v), l(u) + c(u,v)\}$$

At some stage in the algorithm the label will be set to the shortest path cost and would not decrease further. The following lemma helps us prove the correctness of the algorithm:

Lemma 4.4. *Suppose $SP(u,s)$ has k edges in it. Then $l(u) = SP(u,s)$ in k relaxation steps.*

Proof: The proof is by induction. For vertices which are distance 1 from s the label is set correctly at the first step itself. This label will not change at successive relaxation steps. Consider a vertex w whose shortest paths are of edge length k. Assuming that the lemma is true for vertices with shortest path edge lengths less than k, we note that the shortest path to w uses as its last intermediate vertex a vertex whose shortest path edge length is $k-1$. The label of this vertex will be set to its shortest path cost at the $k-1$st relaxation step. Thus at the kth relaxation step the label of $w, l(w)$ will be set to $SP(s,w)$.

Consequently the algorithm requires at most $n-1$ relaxation steps provided the graph does not contain a negative cycle. Negative cycle are detected if any label shows a decrease even at the nth relaxation step.

4.2 APSP Problem

For the All Pairs problem it is considerably easier to establish a recurrence which can be solved efficiently via dynamic programming.

Suppose we have numbered the vertices $v_1, v_2 \ldots v_n$. This labelling is arbitrary. We define $SP(i,j,k)$ to be the length of the shortest path from v_i to v_j using vertices in the set $\{v_1, v_2 \ldots v_k\}$. Thus $SP(i,j,n)$ gives us the length of the desired shortest path between v_i and v_j. The following recurrence characterizes $SP(i,j,k)$ (assuming no self-loops of negative cost):

$$SP(i,j,k) = min\{SP(i,j,k-1), SP(i,k,k-1) + SP(k,j,k-1)\}$$

$$SP(i,j,0) = c(i,j)$$

$$SP(i,i,k) = 0$$

This recurrence is "convergent" in a simple way.

In fact it is easy to solve the recurrence in $O(n^3)$ steps by maintaining an $O(n^2)$ matrix of shortest path lengths (Details are left to the reader). In this matrix the (i,j)th entry will represent $SP(i,j,k)$ for some k. Moreover negative cycles are easily detected since a diagonal entry of the matrix will become negative at the discovery of a negative cycle.