

## The Dynamic Programming Technique

### Preliminaries:

We define a *directed graph* to be a pair  $G = (V, E)$  where  $V$  is a set of vertices and  $E \subseteq V \times V$  is a set of (directed) edges.  $c(u, w) \in \mathbb{R}^{\geq 0} \cup \{\infty\}$  is the cost of edge  $(u, w)$ .  $c(u, w) = \infty$  if there is no edge between  $u$  and  $w$ .  $v_1, \dots, v_m$  is a (*directed*) *path from  $v_1$  to  $v_m$*  if  $\forall i \in \{1, \dots, m-1\} (v_i, v_{i+1}) \in E$ . The cost of a path  $v_1, \dots, v_m$  is  $c(v_1, \dots, v_m) = \sum_{i=1}^{m-1} c(v_i, v_{i+1})$ . In what follows, we will denote vertex  $v_i$  by  $i$  and assume that  $V = \{1, \dots, n\}$ . If  $p$  is a path and  $S$  is a set, then  $p \subseteq S$  means *all vertices in  $p$  are members of  $S$* .

We wish to solve the following search problem:

### ALL-PAIRS-CHEAPEST-PATH-SEARCH (APCPS):

**INPUT:** A weighted directed graph  $G = (V, E), c: E \rightarrow \mathbb{R}^{\geq 0} \cup \{\infty\}$

**OUTPUT:** For each pair of vertices  $u, v$ , a cheapest path from  $u$  to  $v$

We follow the dynamic programming technique

1. Consider the related value problem, *find the value of the optimal structure*

### ALL-PAIRS-CHEAPEST-PATH (APCP):

**INPUT:** A weighted directed graph  $G = (V, E), c: E \rightarrow \mathbb{R}^{\geq 0} \cup \{\infty\}$

**OUTPUT:** For each pair of vertices  $u, v$ , the cost of a cheapest path from  $u$  to  $v$

2. Define an array  $A$  (which would be useful in solving the value problem).

Let  $A[k, i, j]$  be the cost of the cheapest path from  $i$  to  $j$  which mentions only vertices in  $\{i, j, 1, 2, \dots, k\}$ .

So for  $k \in \{0, \dots, n\}, i, j \in \{1, \dots, n\}$

$$A[k, i, j] = \min\{c(p) : p \text{ is a path from } i \text{ to } j \text{ and } p \subseteq \{i, j, 1, 2, \dots, k\}\}$$

3. Determine how you would use  $A$  to solve the value problem

$A[n, i, j]$  is the cost of a cheapest path from  $i$  to  $j$  where any vertices are allowed as intermediates.

So  $A[n, *, *]$  solves APCP.

4. Write down a recurrence relation which defines  $A$

$$A[k, i, j] = \begin{cases} 0 & \text{if } k = 0, i = j \\ c(i, j) & \text{if } k = 0, i \neq j \\ \min\{A[k-1, i, j], A[k-1, i, k] + A[k-1, k, j]\} & \text{if } k > 0, i \neq j \end{cases}$$

The final part of the recurrence is understood as follows: Let  $p$  be a minimum cost path from  $i$  to  $j$  such that  $p \subseteq \{i, j, 1, \dots, k\}$ . There are 2 cases to consider

(a)  **$k$  appears in  $p$** , then it appears in  $p$  at most 1 time (since edges have positive weight).

So  $p = ip_1kp_2j$  for some paths  $p_1, p_2 \subseteq \{1, \dots, k-1\}$ . Then  $ip_1k$  is a minimum cost  $i$  to  $k$  path with vertices in  $\{i, k, 1, \dots, k-1\}$  so  $c(ip_1k) = A[k-1, i, k]$  (otherwise we could replace  $ip_1k$  with a smaller cost  $i$  to  $k$  path). Similarly  $c(kp_2j) = A[k-1, k, j]$  so  $A[k, i, j] = A[k-1, i, k] + A[k-1, k, j]$ .

(b)  **$k$  does not appear in  $p$** , then  $A[k, i, j] = A[k-1, i, j]$

The last line of the recurrence computes  $A[k, i, j]$  as the minimum of the above two cases.

5. Write an *iterative* algorithm to fill in  $A$

```

Initialize A[] to 0;
Initialize D[] to 0; // Added by steps 6,7
for(k=0;k<=n;k++){
    for(i=1;i<=n;i++){
        for(j=1;j<=n;j++){
            if(k==0){
                A[k,i,j]=c(i,j);
            } else if(A[k-1,i,k]+A[k-1,k,j]<A[k-1,i,j]){
                A[k,i,j]=A[k-1,i,k]+A[k-1,k,j];
                D[i,j]=k; // Added by steps 6,7
            } else {
                A[k,i,j]=A[k-1,i,j];
            }
        }
    }
}
} } }

```

Note: The above algorithm works since it computes new values of  $A$  based on previously computed values of  $A$ .

**6.** Define an array  $D[]$  which keeps track of critical decisions made in filling out  $A[]$   
For  $i, j \in \{1, \dots, n\}$   $D[i, j]$  = an intermediate vertex on some cheapest  $i$  to  $j$  path.  $D[i, j] = 0$  means that the edge  $(i, j)$  is a cheapest  $i$  to  $j$  path (so no intermediate vertices are required).

**7.** Modify the iterative algorithm which fills in  $A[]$  so that it also fills in  $D[]$   
The required modifications have been made in the code above.

**8.** Use  $D[]$  and  $A[]$  to obtain an optimal structure.  
Assuming that  $A[]$  and  $D[]$  are already filled in, the following recursive routine prints the vertices in a cheapest  $i$  to  $j$  path.

```

printCheapPath(i,j){
    if(i==j)return;
    k=D[i,j];
    if(k==0){
        print "edge from ", i, "to ", j;
        return;
    } else {
        printCheapPath(i,k);
        printCheapPath(k,j);
    }
}
}

```

**9.** Analyse the algorithm  
Initializing  $A$  and  $D$  takes time  $O(n^3) + O(n^2)$ . Filling  $A$  and  $D$  takes time  $O(n^3)$ . So the total time to compute  $A$  and  $D$  is  $O(n^3)$ . Finally, once  $A$  and  $D$  are computed, any call to `printCheapPath` involves a recursion that walks a tree with  $O(n)$  leaves (the edges in the  $i$  to  $j$  path). Each node in the recursion tree is visited a constant number of times, so walking the tree takes time  $O(n)$ .