DYNAMIC PROGRAMMING

Dynamic programming is another algorithm design method which can be used for solving problems whose solution can be viewed as a result of a sequence of decisions.

Ex1. General knapsack problem. Notice that to find out the solution to a knapsack problem, we first have to make a decision on x_1, and then on x_2, then on x_3, etc.

Ex2. Find the shortest path from vertex i to vertex j in a directed graph G.
We have to decide which vertex should be the second vertex, which the third vertex and so on until vertex j is reached. An optimal sequence of decisions is one which results in a path of least length.

Ex3. 0/1 knapsack problem. To find out the solution to a 0/1 knapsack problem, we first have to make a decision on x_1, and then on x_2, then on x_3, etc.

For some of the above class of problems, an optimal sequence of decisions can be found out by making the decisions one at a time and never making an erroneous decision. E.g., an optimal solution to the knapsack problem (Ex1) can be found out by using the Greedy_Knapsack algorithm, which makes the best decision (based only on local information) at each step. In other words, at each step we know which object is to be considered so as to get an optimal solution finally.

There are, however, some other of the above class of problems (Ex2, Ex3), for which it is not possible, at each step, to make the best decision (based only on local information). E.g., example problem Ex2. One way to find the shortest path from vertex i to vertex j in a directed graph G is to decide which vertex should be the second vertex, which the third, and so on. Suppose, we are at vertex p in between vertex i and vertex j. Let there be edges from vertex p to vertex m and vertex n. There is no way of knowing at this point (i.e., at vertex p) whether choosing vertex m or vertex n would lead us to the shortest path.

One could use the brute force method to solve the above type of problems for which it is not possible to make a sequence of stepwise decisions leading to an optimal decision sequence. By brute force method, we mean enumerating all decision sequences and then selecting the optimal one. E.g., for the preceding problem, one could find out all the possible paths from vertex i to vertex j, and then choose the shortest among them. But this is an undesirable method because of the amount of enumeration that would be required. This is when **dynamic programming** steps in.

Dynamic programming makes use of the **principle of optimality** to arrive at an optimal sequence of decisions. This principle 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.

In other words, a problem can be solved by using dynamic programming if the principle of optimality holds for that problem. The **main difference** between the **greedy method** and **dynamic programming** is that in the greedy method only one decision sequence is ever generated, whereas in dynamic programming, many decision sequences may be generated.

The **main difference** between the **brute force method** and **dynamic programming** is that all possible sequences are generated in brute force method whereas sequences which cannot possibly be optimal are not generated in **dynamic programming**.

Show that the principle of optimality holds for the following problems:
Ex2. Finding shortest path from vertex i to vertex j.
Proof:
Assume: $i, i_1, i_2, \dots, i_k, j$ is a shortest path (SP) from i to j.
Initial problem state and decision: i, decision to go to vertex i_1.
To prove:
Sequence $i_1, i_2, \dots, i_k, j$ must be a SP from i_1 to j. If it is not, there is another path $i_1, r_1, r_2, \dots, r_q, j$ from i_1 to j. Then, $i, i_1, r_1, r_2, \dots, r_q, j$ is a path from i to j which is shorter than $i, i_1, i_2, \dots, i_k, j$. So, $i, i_1, i_2, \dots, i_k, j$ is not a shortest path, which contradicts our assumption. Hence, sequence $i_1, i_2, \dots, i_k, j$ must be a SP from i_1 to j. Hence, proved.

Ex3. O/1 Knapsack problem.
Proof:
Let KNAP(l,j,Y) represent the problem:

Maximize $\sum_{i=l}^{j}(p_i x_i)$

s.t. $\sum_{i=l}^{j}(w_i x_i) \leq Y$,
$x_i = 0$ or 1.

Then, the O/1 Knapsack problem is KNAP(1,n,M).

Let $y_1, y_2, \dots, y_n$ be an optimal sequence of 0/1 values for $x_1, x_2, \dots, x_n$ respectively.
Case a:
If $y_1=0$, then $y_2, \dots, y_n$ must constitute an optimal sequence for KNAP(2,n,M).
If it does not, then $y_1, y_2, \dots, y_n$ is not an optimal sequence for KNAP(1,n,M) which is a contradiction. Therefore, $y_2, \dots, y_n$ must constitute an optimal sequence for KNAP(2,n,M).
Case b:
If $y_1=1$, then $y_2, \dots, y_n$ must constitute an optimal sequence for KNAP(2,n,M-w_1).
If it does not, then there is another sequence $z_2, \dots, z_n$ such that

$\sum_{i=2}^{n}(w_i z_i) \leq M - w_1$, and $\sum_{i=2}^{n}(p_i z_i) > \sum_{i=2}^{n}(p_i y_i)$.

Hence, the sequence $y_1, z_2, \dots, z_n$ is a sequence which is better than $y_1, y_2, \dots, y_n$. Then, $y_1, y_2, \dots, y_n$ is not an optimal sequence which is a contradiction. Therefore, $y_2, \dots, y_n$ must constitute an optimal sequence for KNAP(2,n,M-w_1).

Forward DP vs. Backward DP
For a problem, we have to find the values of the sequence of variables $x_1, x_2, \dots, x_n$.
Forward DP: Then, the decision for x_i is based on optimal decision sequence for $x_{i+1}, \dots, x_n$.
Backward DP: Let the optimal sequence be $x_1, x_2, \dots, x_n$. Then, the decision for x_i is based on optimal decision sequence for $x_1, \dots, x_{i-1}$.

Building a Backward DP approach to the 0/1 Knapsack problem:

Let the O/1 Knapsack problem be denoted by KNAP(1,n,M):

Maximize $\sum_{i=1}^{n}(p_i x_i)$

s.t. $\sum_{i=1}^{n}(w_i x_i) \leq M$

x_i = 0 or 1.

We have to make an optimal sequence of decisions on the variables $x_1,x_2,\ldots,x_n$.

Assume that the decisions are made in the order $x_n,x_{n-1},\ldots,x_1$. Following an initial decision on x_n, the resulting state of the problem can be either of the following:

State 1: x_n=0. This means that object n is not inserted into the knapsack. Hence, the remaining capacity is M and the profit earned is 0.

State 2: x_n=1. This means that object n is inserted into the knapsack. Hence, the remaining capacity is M-w_n and the profit earned is p_n.

We have already proved that the principle of optimality holds for the 0/1 Knapsack problem. That is, the remaining decisions $x_{n-1},\ldots,x_1$ must be optimal w.r.t. to the problem state resulting from decision x_n.

Let $f_n(M)$ be the value of the optimal solution to KNAP(1,n,M).

Since the principle of optimality holds,

$f_n(M) = \max \{ f_{n-1}(M),\ \underline{p_n} + f_{n-1}(M-\underline{w_n)} \}$

Generalizing,

Let $f_i(X)$, i>0, be the value of the optimal solution to KNAP(1,i,X).

Since the principle of optimality holds,

$f_i(X) = \max \{ f_{i-1}(X),\ \underline{p_i} + f_{i-1}(X-\underline{w_i)} \}$ ---(1)

Eqn. (1) can be used to solve KNAP(1,i,X) knowing that $f_0(X)=0$ for all X and $f_i(X) = -\alpha$ for X<0.

Exercises:

1) Solve the 0/1 Knapsack instance: n=3, $(p_1,p_2,p_3)=(1,2,5)$, $(w_1,w_2,w_3)=(2,3,4)$, M=6. Solution: (1,0,1), Profit = 6.

2) Solve the 0/1 Knapsack instance: n=4, $(p_1,p_2,p_3,p_4)=(10,5,20,30)$, $(w_1,w_2,w_3,w_4)=(3,2,3,4)$, M=9.

DP Algorithm for solving O/1 Knapsack:

```
Opt(i,X) //Initial call is Opt(n,M)
// w are arrays which store the profits and weights respectively.
// This procedure returns the value of the optimal solution for KNAP(1,i,X).
if X < 0 then //this condition must be checked first before the second condition
        return -α  //return a large negative value
else if i==0 then
        return 0
     else
        val1 = Opt(i-1, X)
        val2 = p[i]+Opt(i-1, X-w[i])
        if val2 > val1 then
                return val2
        else
                return val1
        endif
     endif
endif
```

Another simpler and clearer way of solving O/1 Knapsack is by using the following recurrence:

If $X < w_i$

$f_i(X) = f_{i-1}(X)$ ---(2)

else

$f_i(X) = \max \{ f_{i-1}(X),\ p_i + f_{i-1}(X-w_i) \}$ -

$f_i(X) = 0$ for all $i = 0$.

```
Opt(i,X)
// p,w are arrays which store profits and weights respectively.
// This procedure returns the value of the optimal solution for KNAP(1,i,X).
if i==0 then
        return 0
else
        if  X < w[i] then
                val1 = Opt(i-1, X);
                return val1
        else
                val1 = Opt(i-1, X)
                val2 = p[i]+Opt(i-1, X-w[i])
                if val2 > val1 then
                        return val2
                else
                        return val1
                endif
        endif
endif
-----------
```

After solving the Recursive DP Algorithm for solving O/1 Knapsack, an algorithm can be used to find the solution, i.e., the values of x[1..n]. To do so, the result of the subproblems (intermediate results) have to be stored in a data-structure. We use a global integer array R[0..n][0..M] for the same.

Returns the value of the optimal solution and also stores the intermediate results:

```
M_Opt(i,X)  //Global array R[0..n][0..M] to store the intermediate values.
// R[i][X] denotes the value of f_i(X)
// p,w are arrays which store the profits and weights respectively.
//  This procedure returns the value of the optimal solution for KNAP(1,i,X).
if i==0 then
        R[i][X] = 0
        return 0
else
        if  X < w[i] then
                val1 = Opt(i-1, X);
                R[i][X] = val1
                return val1
        else
                val1 = Opt(i-1, X)
                val2 = p[i]+Opt(i-1, X-w[i])
                if val2 > val1 then
                        R[i][X] = val2
                        return val2
                else
                        R[i][X] = val1
                        return val1
                endif
        endif
endif
```

Finds the optimal solution x[1..n]:

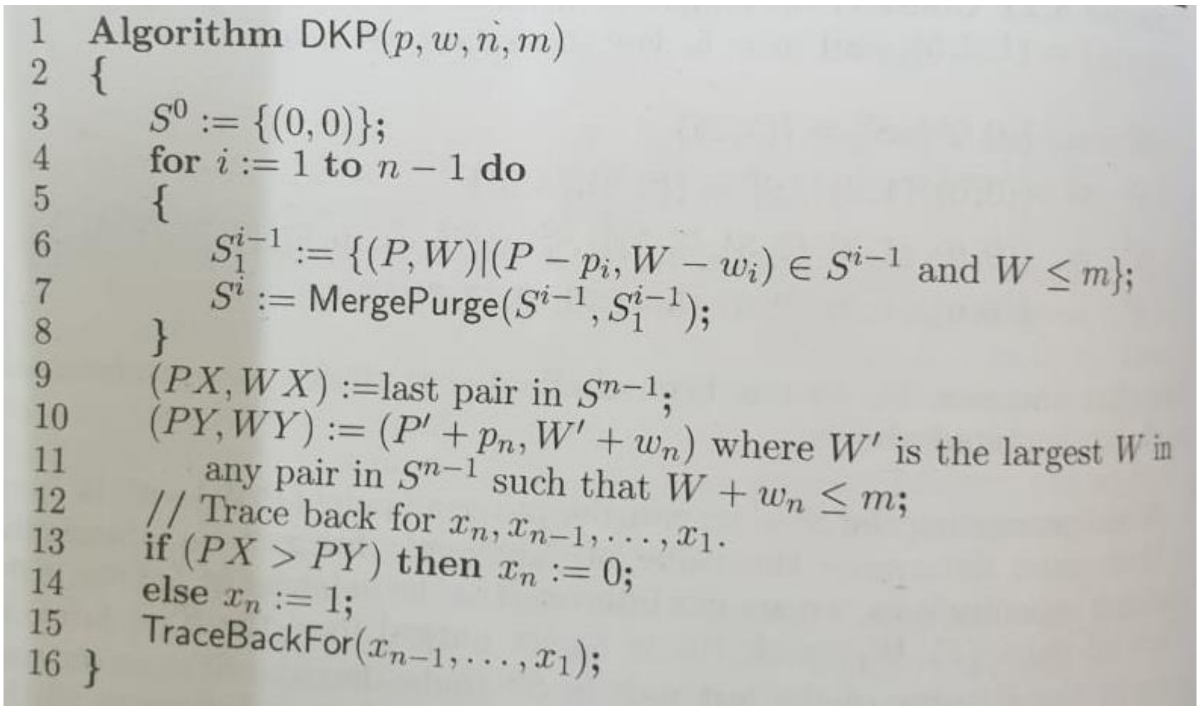
```
TraceBackSolution(i,X)  //Initial call is TraceBackSolution(n,M)
// R[i][X] denotes the value of f_i(X)
// p,w are arrays which store the profits and weights respectively.
// This procedure finds the solution
if i==0 then
        return
else
        if  X < w[i] then
                x[i] = 0
                TraceBackSolution(i-1, X);
        else
                if p[i]+R[i-1][X-w[i]] == R[i][X] then //object i was added
                        x[i] = 1
                        TraceBackSolution(i-1, X-w[i]);
                else
                        x[i] = 0
                        TraceBackSolution(i-1, X);
        endif
endif
```

Further discussions:

The above recursive procedures (Opt and M_Opt) take O(2^n) as discussed in the class. However, there exists an iterative procedure which takes O(nM) as given in 'Algorithm Design' by Klienberg and Tardos.

```
IterativeOpt(n,M)
for i =  0 to n do
        for X = 0 to M do
                .Use eqn. (2) to find $f_i(X)$
                R[i][X] = $f_i(X)$
        done
done
```

Another Iterative DP Algorithm for solving O/1 Knapsack is found in 'Fundamentals of Computer Algorithms - Ellis Horowitz and Sartaz Sahni'. However, this is for information only. Interested students may read up further.



```
Algorithm DKP($p, w, n, m$)
{
    $S^0 := \{(0,0)\}$;
    for $i := 1$ to $n-1$ do
    {
        $S_1^{i-1} := \{(P,W) | (P - p_i, W - w_i) \in S^{i-1}$ and $W \leq m\}$;
        $S^i :=$ MergePurge($S^{i-1}, S_1^{i-1}$);
    }
    $(PX, WX)$ :=last pair in $S^{n-1}$;
    $(PY, WY) := (P' + p_n, W' + w_n)$ where $W'$ is the largest $W$ in
        any pair in $S^{n-1}$ such that $W + w_n \leq m$;
    // Trace back for $x_n, x_{n-1}, \ldots, x_1$.
    if $(PX > PY)$ then $x_n := 0$;
    else $x_n := 1$;
    TraceBackFor($x_{n-1}, \ldots, x_1$);
}
```

S^i can be computed by merging the pairs in S^{i-1} and S_1^{i-1} together. If S^i contains two pairs (p_j,w_j) and (p_k,w_k) with the property that $p_k <= p_j$ and $w_k >= w_j$, then the pair (p_k,w_k) can be discarded (purged) due to the recurrence relation of Eqn. (1).

The Travelling Salesperson Problem (TSP)

Logistics (delivery), network design, robotics and optimization (robot arms to move to many nuts and tighten them)

The 0/1 Knapsack problem is a subset selection problem. Brute-force (a.k.a. Exhaustive) method will require examining 2^n subsets of n objects.

On the other hand, the TSP is a permutation problem. Brute-force (a.k.a. Exhaustive) method will require examining n! permutations of n objects.

Let G=(V,E) be a directed graph with edge costs c_{ij}. n=|V|. A tour of G is a directed cycle that includes every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour. The TSP is to determine a tour of minimum cost.
Suppose, the vertices are numbered from 1 to n. Then, a tour of G is a simple path that starts and ends at vertex 1.

Every tour consists of an edge <1,k> for some k ϵ 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.

For the tour to be optimal, the path from vertex k to vertex 1 must be a shortest path (SP) from k to 1 going through all the vertices in V-{1,k}.

Let g(i,S) be the length of a SP starting at vertex i, going through all the vertices in S and ending at vertex 1.

Then, g(1,V-{1}) is the length of a SP starting at vertex 1, going through all the vertices in V-{1} and ending at vertex 1. Thus, g(1,V-{1}) is the length of an optimal tour.

We have already proved that the principle of optimality holds for the solution of finding the shortest path from a vertex to another vertex in a graph.

Thus,
$g(1,V-\{1\}) = \min\{ c_{1k} + g(k,V-\{1,k\})$, for $2<=k<=n$

Generalizing,
$g(i,S)= \min\{ c_{ij} + g(j,S-\{j\})\}$, for $j \in S$

Start with S as an empty set. Find g(i,S) for all i not in S. Increase its size by 1 and repeat the same until S={2,3,..,n}. Then, value of g(1,S) is the cost of the optimal tour.

Exercise: Solve the TSP instance, where G consisting of 4 vertices is represented by the cost matrix:

0	10	15	20
5	0	9	10
6	13	0	12
8	8	9	0

Soln: The minimum cost tour is 1,2,4,3,1 with cost 35.

<u>Informal dynamic programming algorithm for TSP.</u>

```
procedure DTSP(COST, n, V)
// COST(n:n) denotes the cost matrix of the graph; cost(i,j)= +α if there is no edge <i,j>; //
// n is  the number of vertices, and V the set  of vertices in the graph; //

1.  for k ← 0 to n-2
2.       for all possible sets, S with size k, and all possible i's such that  i∈V,  i ≠ 1,  i∉S  and 1∉S, do
3.            Let j be a vertex such that j ∈S and  COST(i,j)+g(j, S-{j}) is minimum
4.            g(i, S) = COST(i,j)+g(j, S-{j}) //.. Update g(i, S)
5.            D(i,S) = j       // For tracing the tour  later on ..//
6.       repeat
7.  repeat
```

```
//.....Find out the cost of the minimum cost tour which is given by g(1, V-{1})..//
8.  Let j be a vertex such that j ∈V-{1} and  COST(1,j)+g(j, V-{1,j}) is minimum
9.  g(1, V-{1}) = COST(1,j)+g(j, V-{1,j});
10. D(1,V-{1}) = j
//...Find the minimum cost tour //
11.  S ← V-{1}; P(1) ← 1; P(2) ← D(1, S)
12.  for k ← 3 to n do
13.       P(k) ← D( P(k-1), S-P(k-1))  ; S ← S - P(k-1)
14.  repeat
15.  end DTSP
```

Solving TSP using DP requires O($n^2 2^n$).

Single-Source Shortest Paths: General Weights

We now consider the single-source shortest path problem when some or all of the edges of the directed graph G may have negative length (weight). We have seen that Djikstra's ShortestPaths algorithm does not necessarily give the correct results on such graphs.

We wish to solve the problem when negative edge lengths are permitted. However, it is required that the graph has no cycle of negative length. E.g, in the graph denoted by: c(1,2)=1, c(2,1)=-2 and c(2,3)=1, the length of the shortest path from vertex 1 to vertex 3 is $-\alpha$. The corresponding shortest path is 1,2,1,2,1,2,……,1,2,3. Thus, it is necessary to ensure that shortest paths consist of a finite number of edges.

When there is no cycle of negative length in a directed graph of n vertices, there is a shortest path between any two vertices of at most n-1 edges. Note that a path that has more than n-1 edges have a repetition of a least one vertex, and hence must contain a cycle. If it contains a cycle, this cycle can be deleted, which results in a path with the same source and destination. This path is then cycle-free and has a length that is no more than the original path, as the length of the eliminated cycle was non-negative (at least zero).

Let $\text{dist}^l[u]$ be the length of a shortest path from the source vertex v to vertex u containing at most l edges. Thus, $\text{dist}^1[u] = \text{cost}[v,u]$, 1<=u<=n. Since there are no cycles with negative length, we can limit our search for shortest paths to paths with at most n-1 edges. Thus, our goal is to determine $\text{dist}^{n-1}[u]$ for all u. There are two cases:

Case 1: The shortest path (SP) from v to u has exactly n-1 edges. Then, this SP is made up of a SP from v to some vertex j followed by the edge (j,u). All vertices i such that (i,u) is an edge are candidates for j. Thus, the candidate vertex i that minimizes the term $\text{dist}^{n-2}[i]+\text{cost}(i,u)$ is the value for j.

Case 2: The shortest path (SP) from v to u has at most n-2 edges. Then, $\text{dist}^{n-1}[u] = \text{dist}^{n-2}[u]$.

Thus, the following recurrence follows:
$\text{dist}^{n-1}[u] = \min \{ \text{dist}^{n-2}[u], \min \{ \text{dist}^{n-2}[i] + \text{cost}(i,u)\}\}$

Generalizing,
$\text{Dist}^{k}[u] = \min \{ \text{dist}^{k-1}[u], \min \{ \text{dist}^{k-1}[i] + \text{cost}(i,u)\}\}$

Thus, this recurrence can be used to compute dist^k from dist^{k-1} for k = 2, 3,.., n-1.

```
Algorithm BellmanFord(v,cost,dist,n)
//Single-source shortest paths with negative edge costs
begin
    for u = 1 to n do //initialize
        dist[u] = cost[v,u]  //calculating $dist^1[u]$
    endfor
    for k = 2 to n-1 do //calculating $dist^2[u]$, $dist^3[u]$,…, $dist^{n-1}[u]$
        for each u such that $u \neq v$ and u has at least one incoming edge do
            for each (i,u) in the graph do
                if dist[u] > dist[i] + cost[i,u] then
                    dist[u] = dist[i] + cost[i,u]
                endif
            endfor

        endfor

    endfor

end
```

Bellman Ford's algorithm takes $O(n^3)$ if adjacency matrices are used and O(ne) if adjacency lists are used, where *n* and *e* are the number of nodes and the number of edges respectively.

Dynamic Programming is a technique which can be used to solve a variety of problems. The following is a list of other problems (not discussed in class) which can be solved using Dynamic Programming. Interested students may read up about them in the books mentioned:

1) The Multi-Stage Graph problem (Fundamentals of Algorithms by Horowitz and Sahni)
2) All-Pairs Shortest Paths problem (Fundamentals of Algorithms by Horowitz and Sahni)
3) Optimal Binary Search Tree problem (Fundamentals of Algorithms by Horowitz and Sahni)
4) Flow Shop Scheduling problem (Fundamentals of Algorithms by Horowitz and Sahni)
5) Weighted Interval Scheduling problem (Algorithm Design by Kleinberg and Tardoss)
6) Segmented Least Squares problem (Algorithm Design by Kleinberg and Tardoss)
7) Subset Sum problem (Algorithm Design by Kleinberg and Tardoss)
