Algorithm Design

Dynamic Programming

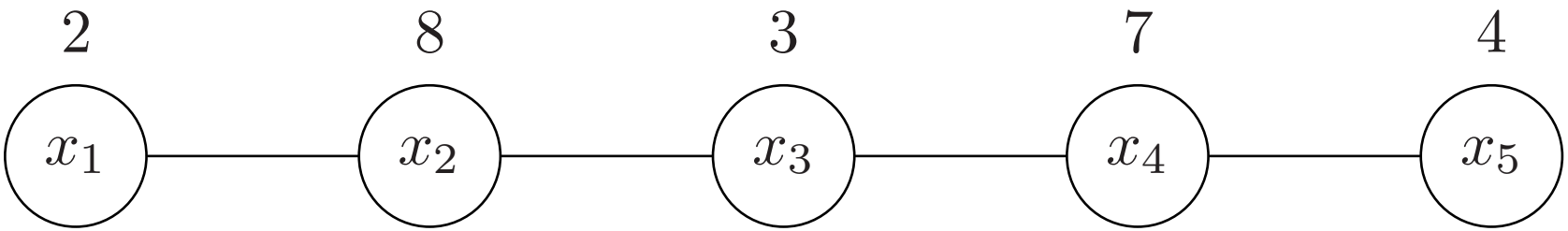
Hikmat Farhat

Independent Set (Linear)

- Given a graph $G = (V, E)$ an independent set is a subset of vertices $S \subseteq V$ such that for all $x, y \in S$, $(x, y) \notin E$.
- Usually we need to an independent set such that $|S|$ is maximum. This general problem is hard to solve in the general case(see NP chapter later)
- In this lecture we consider a special case of the independent set: when all vertices are on the same line.
- The solution for this particular case is trivial: select every other node.
- We make things interesting by adding a weight to each node and asking : what is the independent set which has the largest total weight?

Example

- An example of the (linear) independent set problem with weights is shown below. Clearly the solution in this case is $\{x_2, x_4\}$.



Dynamic Programming Solution

- Given a set of nodes $\{x_1, \ldots, x_n\}$ with associated weights $\{w_1, \ldots, w_n\}$ we need to find the independent set S such that $\sum_{x \in S}$ is maximum
- Assume that the solution $\mathcal{O}$ and the optimal value (which we don't know how to compute yet) is $opt(n)$. The dependence on n comes from the fact that we will express this value in terms of smaller subproblems.
- Consider the last node x_n. Either x_n contributed to $opt(n)$ (i.e. $x_n \in \mathcal{O}$) or not.
- In the first case $x_n \in \mathcal{O}$ implies that $x_{n-1} \notin \mathcal{O}$. If we can compute the optimal value for $\{x_1, \ldots, x_{n-2}\}$ then we add to it w_n to obtain the optimal value
- In the second case $x_n \notin \mathcal{O}$ then the optimal value is the same as the one obtained for $\{x_1, \ldots, x_{n-1}\}$

Dynamic Programming Solution

- From the analysis above we conclude that

$$opt(n) = \max \begin{cases} w_n + opt(n-2) \\ opt(n-1) \end{cases}$$

- Applying the above recursive formula to the previous example while noting that $opt[0] = 0, opt[1] = 2$ (only x_1 is included) we get

$$opt[5] = \max \begin{cases} 4 + opt[3] \\ opt[4] \end{cases}$$

$$opt[4] = \max \begin{cases} 7 + opt[2] \\ \quad\quad opt[3] \end{cases}$$

$$opt[3] = \max \begin{cases} 3 + opt[1] \\ \quad\quad opt[2] \end{cases}$$

$$opt[2] = \max \begin{cases} 8 + opt[0] \\ \quad\quad opt[1] \end{cases}$$

- Since $opt[0] = 0$ and $opt[1] = 2$ then $opt[2] = 8$, $opt[3] = 8$,$opt[4] = 15$,$opt[5] = 15$

Finding the solution

- Our method allowed us to compute the optimal value. What if we want the list of vertices for the optimal solution?
- In all dynamic programming problems the way to find the optimal solution from the optimal value is almost the same:walk backwards.
- In the example above: we start with x_5 is it in the solution?
- Since $opt[5] = 15$ and $opt[4] = 15$ then x_5 was **not** selected in the solution
- Since $opt[4] = 15$ and $opt[3] = 8$ then x_4 **was** selected in the solution, which means x_3 cannot be in the solution
- Since $opt[2] = 18$ and $opt[1] = 2$ then x_2 **was** selected in the solution which means x_1 cannot be in the solution
- from the above the optimal solution includes $\{x_2, x_4\}$.

Bottom up solution

- A quick look at the recursive solution we can see that the values of $opt[2]$ and $opt[3]$ were needed twice (for the computation of $opt[3], opt[4], opt[5]$).
- If the size of the problem was bigger then these values (among others) would have been needed even more.
- In fact this **top down** approach will lead to exponential complexity because many of the values have to be repeatedly computed.
- To avoid this exponential blowup we either have to use memoization (saving the computed values for later use) or solve the problem in a **bottom up** manner by using iteration.

Iterative Solution

- The iterative solution for the independent set (linear) problem can be written as

IS(G)

$opt[0] = 0$
$opt[1] = w_1$
for $2 = 1$ **to** n **do**
  $opt[i] = max(opt[i-1], w[i] + opt[i-2])$
return $opt[n]$

Weighted Interval Scheduling

- The weighted interval scheduling is a generalization of the interval scheduling we studied using greedy approach.
- In the greedy approach the intervals were considered to be equivalent.
- In this generalization each interval i has a weight v_i in addition to starting $e(i)$ and ending $f(i)$ time.
- A simple example shows that the greedy approach no longer works when the intervals have weights

Dyanmic Programming Solution

- Let I be a set of intervals where each interval i starts at $e(i)$, ends at $f(i)$ and has values $v(i)$.
- Our goal is to find a subset of non-overlapping intervals $S \subseteq I$ such that $V = \sum_{i \in S} v(i)$ is maximum.
- First we sort the intervals by ending time as we did in the case of greedy approach.
- Given interval i let $p(i)$ be the largest index $k < i$ such that intervals k and i do not overlap.
- As an example let $I = \{(1, 2, 2), (1, 5, 7), (4, 7, 3), (6, 8)\}$ where (a, b, c) denotes an interval that starts at a, ends at b and has value c.
- The greedy approach will select the smallest ending time first so the greedy solution gives

$$S = \{(1, 2, 2), (4, 7, 3)\} \text{ for a value of } 5$$

- Whereas the optimal solution is

$$S = \{(1, 5, 7), (6, 8, 4)\} \text{ for a value of } 11$$

- The function p in this example has the following values

$$p(1) = 0$$
$$p(2) = 0$$
$$p(3) = 1$$
$$p(4) = 2$$

- For the general problem where $I = \{i_1, \ldots, i_n\}$ let $opt(n)$ be the optimal solution and consider the last interval i_n.

- There are two cases
  1. $i_n \in opt(n)$. In this case all the intervals $i_{p(n)+1}, \ldots, i_{n-1}$ overlap with i_n and therefore cannot be in $opt(n)$
  2. $i_n \notin opt(n)$. In this case the solution with n intervals is the same as the solution for $n-1$ intervals since we know $i_n \notin opt$
  3. Since we don't know which case we have, we take the larger one. This means that we compute both possibilities and the larger of the two values is chosen as the optimal solution

$$opt(n) = \max \begin{cases} v_n + opt(p(n)) & \text{if } i_n \in opt(n) \\ opt(n-1) & \text{if } i_n \notin opt(n) \end{cases}$$

- The above is a recursive equation that will allow us to compute the optimal value.
- If we use the equation as it is it might lead to exponential blow up.
- Instead we do the computation bottom up

OPT(I)

$opt[0] = 0$
for $i = 1$ **to** n **do**
  $opt[i] = max(opt[i-1], v[i] + opt[p[i]])$
return $opt[n]$

- Clearly the complexity of the above algorithm is $\Theta(n)$.
- Also we need to sort the intervals by ending time which is $\Theta(n \log n)$.
- Once the intervals are sorted computing the $p[i]$'s can be done in $\Theta(\log n)$ for each i (similar to binary search) for a total of $\Theta(n \log n)$

Max subarray sum

- Recall: given an array of numbers $A_0, A_1, \ldots, A_{n-1}$ let

$$S_{ij} = \sum_{k=i}^{j} A_k$$

- Goal: find $\max_{ij} S_{ij}$
- Brute force (see first lecture): $\Theta(n^3)$
- Divide and conquer (see first lecture): $\Theta(n \log n)$
- Can we do better? yes.

Dyanmic programming solution

- Let $M_j = \max_i S_{ij}$. M_j is the maximum sum **ending** at j for all possible values of $0 \leq i \leq j$.(see next slide for details)
- The max subsequence sum can be written as $\max_j M_j$.
- The key idea is that the values M_j can be stored in an array and computed in $\Theta(n)$ time.
- What is the relationship between the different values of M_j?
- It is clear that if $M_{j-1} > 0$ then $M_j = M_{j-1} + A[j]$.
- Otherwise $M_{j-1} \leq 0$ then $M_j = A[j]$
- The above two cases can be combined to obtain

$$M_j = max(A[j], M_{j-1} + A[j])$$

$$M = \max_j \max_i \sum_{k=i}^{j} A[k] = \max_j \left(\max_i \sum_{k=i}^{j} A[k] \right)$$
$$= \max_j M_j$$

- Now

$$M_j = \max_i \sum_{k=i}^{j} A[k] = \max(\sum_{k=1}^{j} A[k], \sum_{k=2}^{j} A[k], \ldots, A[j])$$
$$= \max(\sum_{k=1}^{j-1} A[k] + A[j], \sum_{k=2}^{j-1} A[k] + A[j], \ldots, A[j])$$

$$= \max\left(\sum_{i=1}^{j-1} A[k], \sum_{i=2}^{j-1} A[k], \ldots, 0\right) + A[j]$$
$$= \max(M_{j-1} + A[j], A[j])$$

Example

- Consider the array

$$[2, -4, 6, 3, -7, 4, 5, -5, -6, 4, 6, -4, 3]$$

- One can guess by inspection that the subarray $[6, 3, -7, 4, 5]$ will give the largest sum.
- using the proposed algorithm we get

$$[2, -2, 6, 9, 2, 6, 11, 6, 0, 4, 10, 6, 9]$$

- By scanning the result we obtain the maximum 11 as it should be.
- Note that this method not only computes the result for $n = 13$ elements but also solves **all subproblems**

Code

```
MaxSubarraySum(A)
```

$M[0] \leftarrow 0$
$max \leftarrow 0$
for $i = 1$ **to** n **do**
  $M[i] \leftarrow max(A[i], A[i] + M[i-1])$
/* Now compute the maximum of all sums */
for $i = 1$ **to** n **do**
  if $M[i] > max$ **then**
    $max \leftarrow M[i]$
return max

Maximum Common Subsequence

- Consider two strings X and Y. Our goal is to find the longest sequence that is contained in both X and Y.
- For example: $BDBDC$ and $ADBC$ then the longest common sequence is DBC
- Note that the characters in the sequence do not have to be necessarily consecutive
- This problem is **different** from the **common substring** problem where the characters in the common sequence are required to be consecutive.
- In this example the longest common substring is DB

Dynamic Programming Solution

- Consider two strings $X = x_1 x_2 \dots x_{n-1} x_n$ and $Y = y_1 y2 \dots y_{m-1} y_m$. We denote the prefix of length k of X by $X_k = x_1 \dots x_k$ so that $X = X_n$ and $Y = Y_m$.
- let $LCS(X, Y)$ be the **length** of the longest common subsequence between X and Y then we can write $LCS(X, Y) = LCS(X_n, Y_m)$.
- If $x_n = y_m$ then the common subsequence must include x_n. In this case $LCS(X_n, Y_m) = LCS(X_{n-1}, Y_{m-1}) + 1$.
- If $x_n \neq y_m$ then $LCS(X_n, Y_m) = \max(LCS(X_n, Y_{m-1}), LCS(X_{n-1}, Y_m))$

Bottom up algorithm

- First we note that if one substring has 0 elements the length of the common subsequence is 0. Therefore $LCS[i][0] = 0$ and $LCS[0][j] = 0$ for all i, j

```
LCS(X, Y)
for i = 1 to n do
    for j = 1 to m do
        if X[i] = X[j] then
            LCS[i][j] ← 1 + LCS[i − 1][j − 1]
        else
            LCS[i][j] ← max(LCS[i − 1][j], LCS[i][j − 1])
return LCS[n][m]
```

Example

Consider the two strings $X = ABCD$ and $Y = AABDC$

Knapsack

- Given a knapsack of capacity W and n items $x_1, \ldots, x_n$ with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ we need to find a subset $S \subseteq \{x_1, \ldots, x_n\}$ such that

$$\sum_{x_i \in S} w_i \leq W$$

$$\sum_{x_i \in S} v_i \text{ is maximum}$$

Dynamic Programming

- Let S_n be the optimal solution when the problem contains n items and consider item x_n. There are two possibilities:
- Either $x_n \in S_n$. Suppose that we know S_{n-1} the optimal solution for the subproblem including the first $n-1$ items. Can one add to that solution x_n?No, because when the weight of x_n is added, the total weight might be greater than W. Instead we look for the optimal solution for the first $n-1$ items with a knapsack of capacity $W - w_n$ to leave room for item x_n. Therefore $S_n(W) = v_n + S_{n-1}(W - w_n)$
- Or $x_n \notin S_n$. Since $x_n \notin S_n$ then we don't need to leave room for it when we compute S_{n-1}. In this case $S_n(W) = S_n(W)$

- Because we don't know apriori if $x_n \in S_n$ we compute both values and take the largest.

$$S(n, W) = \max \begin{cases} v_n + S(n-1, W - w_n) \\ S(n-1, W) \end{cases}$$

- Before writing a bottom up solution we need to take care of the boundary conditions.
- Anytime the capacity of the knapsack $W = 0$ the solution is just 0.
- Also we need to take int account the case when $w_n > W$. In this case $S(n, W) = S(n-1, W)$

```
Knapsack(w,v,W)
/* if W = 0 then solution for any                    */
/* number of items is 0                              */
for i = 1 to n do
  | S[i][0]=0
/* for any W zero items gives 0                      */
for i = 1 to W do
  | S[0][i]=0
/* Now the solution                                  */
for j = 1 to W do
  |  for i = 1 to n do
  |   | S[i][j]=max(S[i − 1][j], v[i] + S[i − 1][j − w[i]])
```

Example

Consider the following instance of the knapsack problem where the knapsack size is $W = 13$.

$v_1 = 2$	$w_1 = 3$	$\frac{v_1}{w_1} = 0.66$
$v_2 = 4$	$w_2 = 6$	$\frac{v_2}{w_2} = 0.66$
$v_3 = 5$	$w_3 = 7$	$\frac{v_3}{w_3} = 0.71$
$v_4 = 6$	$w_4 = 8$	$\frac{v_4}{w_4} = 7.5$

Note that a greedy approach would choose items v_4 and v_1 for a total value of 8 whereas the optimal is 9 by choosing items 3 and 2.

$n \backslash w$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2	2	2	2	2	2
2	0	0	0	2	2	2	④	4	4	6	6	6	6	6
3	0	0	0	2	2	2	4	5	5	6	7	7	7	⑨
4	0	0	0	2	2	2	4	5	6	6	7	8	8	9

Subset sum

- We are given an array A of n elements and an integer value S and we ask if there is a subset of A whose sum is S
- We start with the last element. There are two choices
  1. Either the last element is in the sum in which case we have to find a subset of of the first $n-1$ elements whose sum is $S - A[n]$
  2. Or the last element is not in the sum in which case we have to find a subset of the first $n-1$ elements whose sum is S
- The above reasoning will lead to

```
subset(A,n,S){
  if(n==0) return false
  if(S==0) return true
  return subset(A,n-1,S)||subset(A,n-1,S-A[n])
```

Example

- Consider the array $A = 2, 7, 3, 1$ and $S = 6$.

```
subset(A,4,6)=subset(A,3,6)||subset(A,3,5)=false||true
subset(A,3,6)=subset(A,2,6)||subset(A,2,3)=false||false=fal
subset(A,3,5)=subset(A,2,5)||subset(A,2,2)=false||true=true
subset(A,2,6)=false
subset(A,2,3)=false
subset(A,2,5)=false
subset(A,2,2)=true
```

Bottom up

We use the recurrence relation that we have obtained to solve the subsetsum problem in a bottom up manner. First note that if the required sum is 0 then the answer is always true. Also if the number of elements is 0 then with the exception of sum=0 all the answers should be false.

SS(A,S)

```
/* if S = 0 then solution for any                    */
/* number of items is True                           */
for i = 0 to n do
  |  R[i][0]=True
/* for any S other than zero                         */
/* 0 items is False                                  */
for j = 1 to S do
  |  R[0][j]=False
/* Now the solution                                  */
for i = 1 to n do
  |  for j = 1 to S do
  |    |  R[i][j]=R[i − 1][j] or R[i − 1][j − A[i]]
```

Balanced Partition of a Set

- Suppose that you have a set of n integers $A = \{A_1, \ldots, A_n\}$.
- Let S_1 and S_2 be a partition of A i.e. $S_1 \cup S_2 = A$ and $S_1 \cap S_2 = \emptyset$.
- Let $sum_1 = \sum_{a \in S_1} a$ and $sum_2 = \sum_{a \in S_2}$ be the sum of elements in each partition.
- Our goal is to find S_1 and S_2 such that $|sum_1 - sum_2|$ is minimum.

- Let $\sigma = \sum_{a \in A} a$, i.e. the sum of all elements in A.
- Given A we know how to find if there is a subset of A whose sum is k.
- Let $p(i, j)$ be true if the first i elements of A have a subset whose sum is j.
- This is the subset sum problem we have studied previously.
- Computer $p(i, j)$ for all $1 \leq i \leq n$ and $0 \leq j \leq \sigma/2$.
- then consider the set

$$\{|j - \sigma/2| \mid p(n, j) = 1\}$$

- Our answer is

$$\min_j \{|j - \sigma/2| \mid p(n, j) = 1\}$$

Example

Consider the set $=\{1, 6, 11\}$ whose sum is 18 so $\sigma/2 = 9$. We compute the subset sum problem in a bottom up fashion we get

Sequence Alignment

- For example

$$ACTGG-ATT$$
$$ACGGGTATG$$

- We need to align two sequence to minimize the "cost".
- Cost the sum of costs where each dash incurs a cost α_{gap} and cost of mismatch α.
- Note that we are assuming that all mismatches are the same cost even though it is easy to modify the cost to depend on the type of mismatch

- Given strings $X = x_1 \dots x_n$ and $Y = y_1 \dots y_m$ we need to find the "best" alignment: the one that leads to the smallest cost
- Let X_i be the prefix of X of size i, i.e. $X_i = x_1 \dots x_i$.
- Let $opt(X_n, Y_m)$ be the optimal solution for prefixes $X_n = X$ and $Y_m = Y$. We can write

$$opt(X_n, Y_m) = \min \begin{cases} \alpha + opt(X_{n-1}, Y_{m-1}) \\ \alpha_{gap} + opt(X_{n-1}, Y_m) \\ \alpha_{gap} + opt(X_n, Y_{m-1}) \end{cases}$$

Base cases

- Before we implement the algorithm we need to determine the value of the base cases: $opt(X_i, 0)$ and $opt(0, Y_i)$.
- Note that when one string is empty we need i gaps to match the other so
- $opt(X_i, 0) = opt(0, Y_i) = i \cdot \alpha_{gap}$

Example

Consider the two strings $X =$ "CG" and $Y =$ "CA" with the cost of a gap α_g=3 and cost of mismatch $\alpha = 7$. The solution obtained from the table below is $CG-$ and $C-A$.

X\ Y	""	C	CA
""	0	3	6
C	3	$\nwarrow$ 0	3
CG	6	$\uparrow$ 3	$\leftarrow$ 6

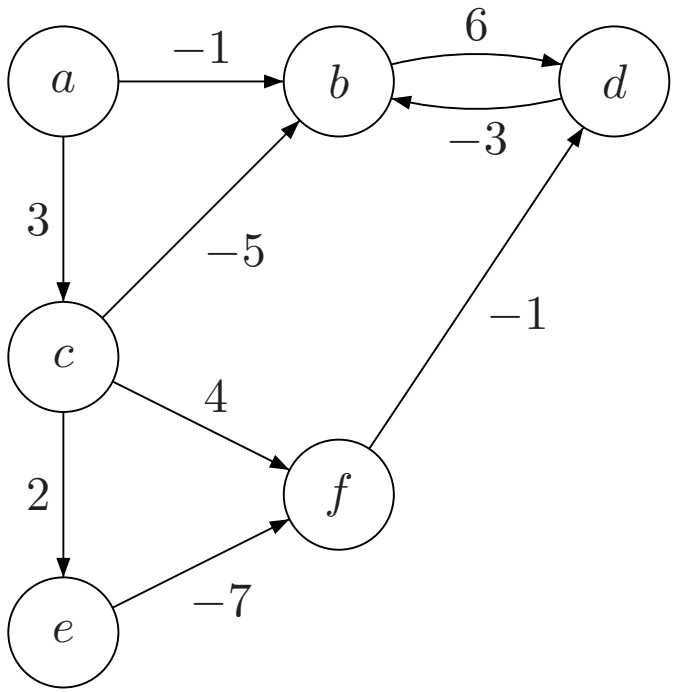
Bellman-Ford Single Source

- The Bellman-Ford algorithm uses dynamic programming to compute the shortest path in graph $G = \langle V, E \rangle$
- The quantity to be optimized is the shortest distance using *at most i edges*.
- Let $d[u, i]$ be the shortest path distance from some source s to destination u using *at most* i edges.
- Let $c_{v,u}$ be the cost of edge $(v, u) \in E$ or ∞ otherise.
- Then $d[u, i+1]$ can be written as

$$d[u, i+1] = \min(d[u, i], \min_{v \in V}(d[v, i] + c_{v,u}))$$

Example

In the graph below we would lile to use the Bellman-Ford algorithm to compute the shortest path from node a to all other nodes.



	0	1	2	3	4	5
a	0	0	0	0	0	0
b	∞	-1	-2	-2	-2	-6
c	∞	3	3	3	3	3
d	∞	∞	5	4	-3	-3
e	∞	∞	5	5	5	5
f	∞	∞	7	-2	-2	-2

Implementation

- Encoding the edge weights

```
0-1:-1
0-2:3
1-3:6
2-1:-5
2-4:2
2-5:4
3-1:-3
4-5:-7
5-3:-1
```

- We choose a large value compared to the edge weights in the graph, say 100.

Solution for the example

	0	1	2	3	4	5
a	0	0	0	0	0	0
b	100	-1	-2	-2	-2	-6
c	100	3	3	3	3	3
d	100	99	5	4	-3	-3
e	100	100	5	5	5	5
f	100	93	7	-2	-2	-2

```
1  import numpy as np
2  num=6
3  opt=np.full((num,num),100)
4  opt[0,0]=0
5  edge=np.full((num,num),100)
6
7  f=open("graph1.txt","r")
8  input=f.read()
9  lines=input.splitlines()
10 for line in lines:
11    x=line.split(":")
12    cost=x[1]
13    y=x[0].split("-")
14    s=y[0]
15    d=y[1]
16    edge[int(s),int(d)]=int(cost)
```

```
pred=np.full(num,-1)
for l in range(1,num): #iterate over length
  for n in range(0,num): #iterate over nodes
    opt[n,l]=opt[n,l-1]
    for m in range(0,num):#iterate over neighbors
      s=opt[m,l-1]+edge[m,n]
      if s<opt[n,l]:
        opt[n,l]=s
        pred[n]=m
```