CS253- Dynamic Programming vs Greedy Algorithms

It is a divide-and–conquer strategy

Chapter 15,16 Topics

Dynamic -- $16.2, $15.4

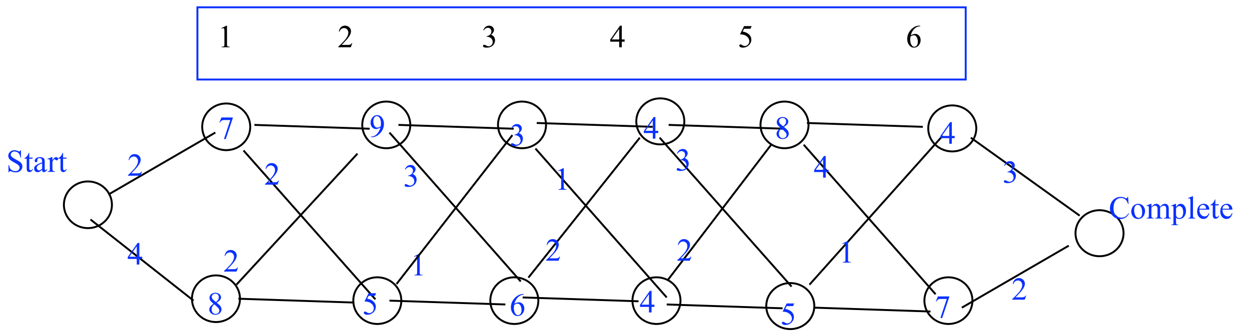
Greedy -- $15.1 -- assembly line scheduling, $16.1--- real time scheduling

- Principle of Optimality (PoO)
  - Know how a problem can be formulated so that it satisfies the PoO
  - Be able to develop a recurrence for a dynamic program
  - Show how a recurrence can be "unrolled" into a loop program for some problems.
- Example Problems
  - $15.1 Car Scheduling-- **dynamic 'programming'**
  - $15.4String matching: acd, adc -- **dynamic 'programming'**
  - $16.1 Real-Time deadline schedules – **greedy algorithm**
  - $16.2 0-1 Knapsack: 1,2,3,4 W=5 -- **dynamic 'programming'**
  - $16.2 fractional Knapsack: 1,2,3,4 W=5 -- **greedy algorithm**

Assembly-line Scheduling for manufacturing cars
A very practical problem
Problem Ingredients:
Chassis **entry station** and Auto **Completion Station**
Time to **enter** and time to **exit** assembly lines are denoted by : e_i , x_i
Two **assembly lines** with **n stations** on each line: $S_{i,j}$ i=1,2 , j=1,2,…..,n
Stations have different **assembly times** (due to different technologies): $a_{i,j}$
Time to move from one station to next station **on the same line** is negligible: 0 for $S_{i,j-1} \rightarrow S_{i,j}$
Time to move from one station to next station on the **different** line is : time from $S_{i,j-1} \rightarrow S_{i',j}$ is $t_{i,j-1}$
where i' = 1 if i=2 and i' = 2 if i=1, more precisely $i' = i - (-1)^i$
Note. If there are more than two lines, then $t_{i,\, i',j-1}$

I will prefer to use: time from $S_{i,j-1} \rightarrow S_{i',j}$ is $t_{i,j}$ **but be consistent with the book, I do not**

Time $f_i[j]$ to get past a station is time to reach the station $\min(f_i[j-1]\ ,\ f_{i'}[j-1] + t_{i',j-1})$ plus the assembly time $a_{i,j}$.
What is the line number of station $S_{?,j-1}$ used to reach $S_{i,j}$
$l_i[j]$ is the line number of station (j-1): $S_{?,j-1}$ to reach $S_{i,j}$
$f_i[j]$ is the time to get past a station $S_{i,j}$



Steps. Structure of solution

$S_{i,1}$ is fixed – only one way to go to station one.

To determine $S_{i,j}$ j =2,3,…,n, there are two choices See figure

The fastest way to reach $S_{1,j}$ is from $S_{1,j-1}$ or $S_{2,j-1} + t_{2,j-1}$

There was fastest way to $S_{1,j-1}$ or $S_{2,j-1}$. Now apply the same reason on $S_{1,j-1}$ or $S_{2,j-1}$

Similarly

The fastest way to reach $S_{2,j}$ is from $S_{2,j-1}$ or $S_{1,j-1} + t_{1,j-1}$

There was fastest way to $S_{1,j-1}$ or $S_{2,j-1}$. Now apply the same reason on $S_{1,j-1}$ or $S_{2,j-1}$

Therefore we consider both possibilities depending on which line we are.

The property of finding Optimal solution through optimal solutions of sub problems is called **optimal substructure.**

Recursive Solution Think recursively

The fastest way through the line is the fastest way through the stations j=1,2,3, …, n

Let $f_i[j]$ be the fastest time to move the chassis from the start point through station $S_{i,j}$

Let f* be the fastest time to move the chassis from the start point to the end point.

How to calculate the fastest time:

$f_i[1] = a_{i,j} + e_i$ j=1

$f_i[j] = a_{i,j} + \min(f_i[j-1] , f_{i'}[j-1] + t_{i',j-1})$, j>1

$f_i[j]$ for i=1,2; j=1,2,….,n gives the optimal solutions to sub problems

$f^* = \min(f_1[n] + x_1, f_2[n] + x_2)$

In order to keep track of the path we need for j=2,…,n

if $f_i[j-1] < f_{i'}[j-1] + t_{i',j-1}$

$l_i[j] = i$

$f_i[j] = a_{i,j} + f_i[j-1]$

otherwise

$l_i[j] = i'$

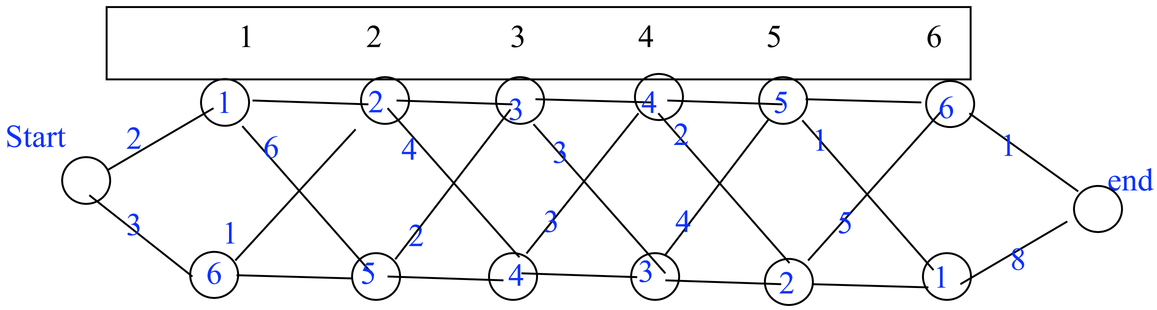
$f_i[j] = a_{i,j} + f_{i'}[j-1] + t_{i',j-1}$

$l^* = 1$ if $f_1[n] + x_1 \le f_2[n] + x_2$

$l^* = 2$ else $f_1[n] + x_1 > f_2[n] + x_2$

Example Illustrates that you cannot select stations with min $f_i[j]$

This the same as above with slight change at the end



j	1	2	3	4	5	6	exit
$f_1[j]$	**3**	**5**	**8**	**12**	**17**	**23**	**1**
$f_2[j]$	**9**	**14**	**13**	**14**	**16**	**17**	**8**

$f^* = 24$, $l^*=1$ – last station is $S_{1,6}$

The sequence of stations is $S_{1,1}$ **,** $S_{1,2}$ **,** $S_{1,3}$ **,** $S_{1,4}$ **,** $S_{1,5}$ **,** $S_{1,6}$

j	2	3	4	5	6
$l_1[j]$	1	1	1	1	1
$l_2[j]$	1,2	1	1	1,2	2

Complexity Brute force: If we know which stations be used then it can be computed in □(n) time.

There are 2^n possibilities for choosing a path. At each of n stations, there are two choices. Alternately, there are 2^n subsets if $\{1,2,3,\ldots,n\}$, take one subset from one line and the complement from the other line. Thus the fastest compute time □(n) which is not practical for large n.

Now an easier way to compute $f_i[j]$ in the form of table.

Note the $f_i[j]$ is computed in terms of $f_i[j-1]$

We can compute the fastest path in time □(n)

How to compute the fastest time?

Invariants

Algorithm

FastestPath(a,t,e,x,n)

for i=1,2

$f_i[1] = a_{i,1}+e_i$

for j=2 to n
Invariant: $f_{1..2}[j-1]$ is optimal time to go past station $S_{1..2j-1}$

for i=1,2
Invariant: $f_{i-1}[j]$ is optimal time to go past station $S_{i-1,j}$
if $f_i[j-1] < f_{i'}[j-1] + t_{i',j-1}$
$l_i[j] = i$
$f_i[j] = a_{i,j} + f_i[j-1]$
otherwise
$l_i[j] = i'$
$f_i[j] = a_{i,j} + f_{i'}[j-1] + t_{i',j-1}$
Invariant: $f_i[j]$ is optimal time to go past station $S_{i,j}$
l_{ij} is the line of station j-1 to reach station S_{ij}
Invariant: $f_{1..2}[j]$ is optimal time to go past station $S_{1..2j}$, $l_{1..2j}$ is the line number of station $S_{1..2,j-1}$ to reach station $S_{1..2j}$

if ($f_1[n] + x_1 \leq = f_2[n] + x_2$)
then $f^* = f_1[n] + x_1$, $l^*=1$
else $f^* = f_2[n] + x_2$, $l^*=2$

How to construct sequence of stations
We have constructed the times for the fastest path.
$f_i[j]$, f^*, $l_i[j]$, l^*

PrintStation (l*,n)

1. i= l* // here l is l*
2. print "line" i "station n
3. for j=n downto 2
   a. $i = l_i[j]$
   b. print "line" i "station j-1

Greedy Algorithm
Activity selection -- Scheduling

Elements of the Greedy Strategy

Greedy algorithm makes a choice among several possible subproblems and solves the subproblem recursively. This is fairly satisfactory but does not give the optimal solution all the time.

Strategy

1. determine the optimal structure
2. develop the recursive solution
3. at each stage of recursion make a greedy choice.
4. show greedy choice makes one of the subproblems empty(solved)
5. develop recursive algorithm implementing the greedy strategy
6. convert the recursive algorithm to iterative algorithm.

Greedy algorithm makes locally optimal choices. Greedy algorithms work well in most of the cases.

Applications:

Data compression(HuffmanCoding)
Minimum spanning trees.

Problem: Given a set of activities, maximize the set of mutually compatible activities.

$S = \{a_1, a_2, \ldots, a_n\}$

e.g. Optimize the use of a lecture hall, schedule classes in lecture hall.

Each activity has start time and finish time.

$a_i : s_i < f_i$ time interval $[s_i, f_i)$

Two activities a_i, a_j are mutually **compatible/disjoint** if $[s_i, f_i) \cap [s_j, f_j) = \emptyset$, i.e. $s_i > f_j$ or $s_j > f_i$

Note. Start and finish times are finite, start times are non-negative: $0 \le s_i < f_i < \infty$

Example

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

Note f_i are sorted in the ascending order. Mutually compatible activities subsets are:

$\{ a_1, a_4, a_8, a_{11}\}$
$\{ a_1, a_4, a_9, a_{11}\}$
$\{ a_2, a_4, a_9, a_{11}\}$
$\{ a_3, a_7, a_{11}\}$
if we start at a_4, there is not chance of optimizing because it has already been used.

Optimal size is unique, optimal set may not be unique. How to determine these?

What is the Greedy Strategy?

There are several choices: we choose greedy strategy: one of the subproblems becomes trivial, empty.
Convert it to recursive greedy algorithm
Convert the recursive algorithm to iterative algorithm

$S = \{a_1, a_2, \ldots, a_n\}$
Let
$S_{ij} = \{a_k: \mathbf{f_i} < s_k < f_k < \mathbf{s_j}\}$
be the set of all activities **start after a_i finishes** and **finish before a_j starts**.
To cover the entire spectrum of real line, let **$f_0 = 0$ and $s_{n+1} = \infty$**
$S = S_{0,n+1}$ $S_{ij} = \emptyset$ for $i \geq j$

Now if $a_k \in S_{ij}$ then it can be decomposed into $S_{ik} \cup \{a_k\} \cup S_{kj}$

Optimal Structure:
If A_{ij} is a solution to S_{ij}, and A'_{ij} another solution with more activities, we can replace A_{ij} with A'_{ij} to get a better solution.
So we construct optimal solutions.
If A_{ij} is an optimal solution to S_{ij}, then $S_{ik} \cup \{a_k\} \cup S_{kj}$ has an optimal solution $A_{ik} \cup \{a_k\} \cup A_{kj}$ if $a_k \in A_{ij}$

How to select k?
Recursive solution?
Recursively define optimal value of solution,
$c[i,j] =$ # of activities in max_size subset of activities in S_{ij}

Since $S_{ij} = \emptyset$ for $i \geq j$

$c[i,j] = 0$ for $i \geq j$
$c[i,j] = c[i,k] + c[k,j] + 1$
$c[i,j] = 0$ if $S_{ij} = \emptyset$
$c[i,j] = \max\{c[i,k] + c[k,j] + 1 \text{ for } a_k \in S_{ij}\}$ if $S_{ij} \neq \emptyset$

Greedy Algorithm
Theorem: If $f_m = \min\{f_k: a_k \in S_{ij}\}$, then

1. $a_m \in S_{ij}$ is part of **some max_size subset** of mutually compatible activities
Now $S_{ij} = S_{im} \cup \{a_m\} \cup S_{mj}$

2.subproblem is S_{im} empty

$f_i < s_m < f_m$ and f_m is the min

Now $S_{ij} = \{a_m\} \cup S_{mj}$

Now let a_k be the first activity in S_{ij}, if $a_m = a_k$?, trivial, if $a_m \neq a_k$? then a_k is the first activity in the solution of S_{mj}. Now from the solution of S_{ij} take out_cut a_k, and add_paste a_m to get same size optimal solution

Note. You may have an optimal solution without it also
Example. Here m =1, $\{a_1, a_4, a_8, a_{11}\}$, there is solution without using m $\{a_2, a_4, a_9, a_{11}\}$

Ordering finish times and choosing the first finish time activity is greedy choice that maximizes the remaining time.
Recursive Algorithm. Sorting is O(nlogn)
Recursive_Activity_Selector RAS
```
RAS(s,f,i,n)
        m=i+1;
        while(m≤n and s_m ≤ f_i ), m++;
        if (m≤n) return( { a_m } ∪ RAS(s,f,m,n) )
        else return 
```
Complexity O(n) – look at each activity only once.

Example

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

Iterative solution
```
RAS(s,f)
n = length(s)
A = {a_1}
i=1;
for m=2 to n
        if s_m <f_i
                ;
        else
                A = A U { a_m}
                i=m
return(A)
```

Complexity:
Sorting O(n lg n)
Selection O(n)
Overall O(n lg n)