CS583 Lecture 11

Jana Kosecka

Dynamic Programming
Greedy Algorithms

Many slides here are based on D. Luebke slides

Review: Dynamic Programming

- A meta-technique, not an algorithm (like divide & conquer)
- Applicable when problem breaks down into recurring small sub-problems

Review: Dynamic Programming

- Problem solving methodology (as divide and conquer)
- Idea: divide into sub-problems, solve sub-problems
- Applicable to optimization problems
- Ingredients

1. Characterize the optimal solution
2. Recursively define a value of the optimal solution
3. Compute values of optimal solution bottom up
4. Construct an optimal solution from computed inf.

Greedy Algorithms

- A *greedy algorithm* always makes the choice that looks best at the moment
- The hope: a locally optimal choice will lead to a globally optimal solution
- Minimum weight spanning tree, Dijstra's algorithm (greedy)
- For many problems dynamic programming can be overkill; greedy algorithms tend to be easier to code

Activity-Selection Problem

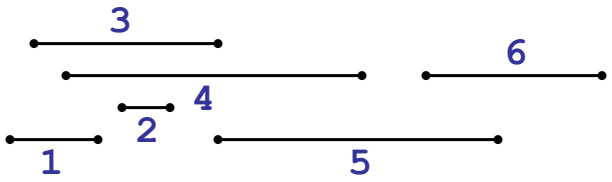
- Problem: get your money's worth out of a carnival

- Buy a wristband that lets you onto any ride
- Lots of rides, each starting and ending at different times
- Your goal: ride as many rides as possible
- Another, alternative goal that we don't solve here: maximize time spent on rides
- Welcome to the *activity selection problem*

- *General: how to schedule activities which require use of a common resource – goal select maximal set of compatible activities*

Activity-Selection

- Formally:
- Given a set S of n activities
  s_i = start time of activity I
  f_i = finish time of activity I
  Find max-size subset A of compatible activities



- Assume (wlog) that $f_1 \leq f_2 \leq \ldots \leq f_n$
- Final times are sorted
- Compatible activities - if their intervals do not overlap

Activity Selection

- Example

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

a_3, a_9, a_{11} Compatible activities

a_1, a_4, a_8, a_{11}

a_2, a_4, a_9, a_{11}

Activity Selection

- Optimal substructure (as in dynamic programming)
- Set of activities compatible with f_i, s_j

$$S_{ij} = \{a_k \in S; f_i \le s_k < f_k \le s_j\}$$

- Need to find a maximal set
- Suppose the set contains activity a_k

$$S_{ij} = S_{ik} + S_{kj} + 1$$

- Recursive definition

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = 0 \\ \max_{i<k<j}\{c[i,k] + c[k,j] + 1\} & \text{otherwise} \end{cases}$$

Activity Selection: Optimal Substructure

- Suppose A is the solution set of the problem
- Let k be the minimum activity in A (i.e., the one with the earliest finish time). Then $A' = A - \{k\}$ is an optimal solution to $S' = \{i \in S: s_i \geq f_k\}$
- In words: once activity #1 is selected, the problem reduces to finding an optimal solution for activity-selection over activities in S compatible with #1
- Proof: if we could find optimal solution B' to S'
- with $|B| > |A - \{k\}|$,
- Then $B \cup \{k\}$ is compatible
- And $|B \cup \{k\}| > |A|$ → contradition since we said A is the optimal solution to the problem

Activity Selection

- Dynamic Programming Strategy

1. Identify sub-problems
2. Recursively define the cost
3. Fill in the cost table in the tabular form

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = 0 \\ \max_{i<k<j}\{c[i,k] + c[k,j] + 1\} & \text{otherwise} \end{cases}$$

4. Need to solve all sub-problems

Activity Selection

Converting dynamic programming to greedy solution
Greedy choice property:
Observation: given a_m activity with the earliest finishing time,
In S_{ij} then that activity will be in some maximal size subset of mutually compatible activities of S_{ij}
(sketch the proof: more details in the book)

Conclusion:

- The activity we choose is always the one with earliest finishing time
- Greedy choice
- Show that it always will maximize the amount of scheduled activities

Recursive Alg.

```
RecursiveActivitySelect(s,f,k,n)
1. m = k+1
2. while m < n and s_m < f_k
3.    do m = m+1
4. if m < n
5. then return
   a_m ∪ RecursiveActivitySelector(s,f,m,n)


Call RecursiveActivitySelector(s,f,0,n)
```

Recursive Alg.

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

s_m

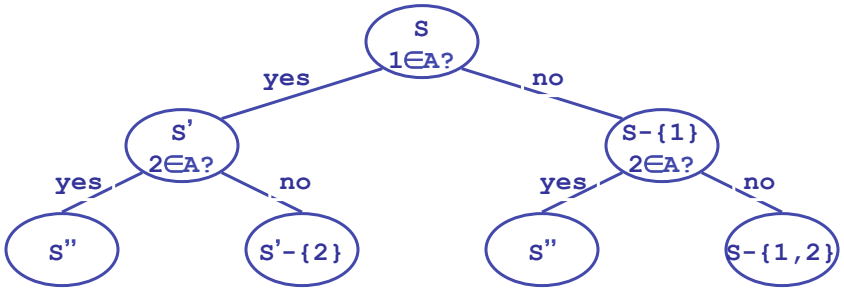
```
RecursiveActivitySelector(s,f,0,n)

RecursiveActivitySelect(s,f,k,n)
1. m = k+1
2. While m =< n and s_m < f_k  % m=1
3.    do m = m+1
4. if m =< n
5. then return
   a_m ∪ RecursiveActivitySelector(s,f,m,n)

Selected:  a_1, a_4, a_8, a_11
```

Activity Selection: Repeated Subproblems

- Consider a recursive algorithm that tries all possible compatible subsets to find a maximal set, and notice repeated subproblems:



Greedy Choice Property

- Dynamic programming? Memoize? Yes, but…
- Activity selection problem also exhibits the *greedy choice* property:
- Locally optimal choice ⇒ globally optimal sol'n

- Them 16.1: if S is an activity selection problem sorted by finish time, then ∃ optimal solution $A \subseteq S$ such that $\{1\} \in A$

- Sketch of proof: if ∃ optimal solution B that does not contain {1}, can always replace first activity in B with {1} (*Why?*). Same number of activities, thus optimal.

Activity Selection:A Greedy Algorithm

- So actual algorithm is simple:

- Sort the activities by finish time

- Schedule the first activity
- Then schedule the next activity in sorted list which starts after previous activity finishes
- Repeat until no more activities

- Easy iterative algorithm

- Intuition is even more simple:
  Always pick the shortest ride available at the time

Huffman coding

- Design of optimal codes
- Example:

- Idea how to design optimal code ?
- Notion of prefix code
- Greedy Algorithm for constructing optimal codes

Huffman coding

Algorithm:
1. Keep the frequencies in Priority Queue (build heap)
2. Take two minimal elements (extract min)
3. Insert their sum to queue
4. Until queue is empty

Running time *O(n lgn)*

Huffman coding

- What is the optimal substructure and greedy choice property ?
- Given alphabet C each character has frequency f[c]
- Suppose x and y are characters with lowest frequencies
- Then there exist an optimal code where x and y have same length and differ only in last bit.

- Optimal substructure property
- Given C and C' with the x and y removed and new symbol
- Added where f[z] = f[x]+f[y]. If we have a tree T' which represents optimal code for C' then replacing node z with two children x and y will yield optimal code for C

Review: The Knapsack Problem

- The famous *knapsack problem*:
  A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market. But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry. What items should the thief take to maximize the haul?

Review: The Knapsack Problem

- More formally, the *0-1 knapsack problem*:
  - The thief must choose among n items, where the ith item worth v_i dollars and weighs w_i pounds
  - Carrying at most W pounds, maximize value
    Note: assume v_i, w_i, and W are all integers
    "0-1" b/c each item must be taken or left in entirety
- A variation, the *fractional knapsack problem*:
  Thief can take fractions of items
  Think of items in 0-1 problem as gold ingots, in fractional problem as buckets of gold dust

Solving The Knapsack Problem

- The optimal solution to the fractional knapsack problem can be found with a greedy algorithm
  How?
- The optimal solution to the 0-1 problem cannot be found with the same greedy strategy

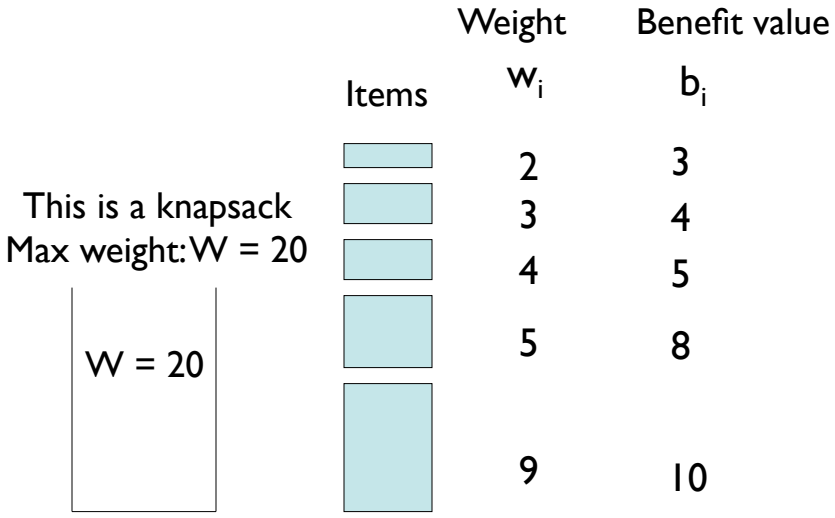
- Greedy strategy: take in order of dollars/pound
  Example: 3 items weighing 10, 20, and 30 pounds, knapsack can hold 50 pounds

  Suppose item 2 is worth $100. Assign values to the other items so that the greedy strategy will fail

The Knapsack Problem: Greedy Vs. Dynamic

- The fractional problem can be solved greedily
- The 0-1 problem cannot be solved with a greedy approach
- 0-1 can be solved with dynamic programming

0-1 Knapsack problem: a picture



0-1 Knapsack problem

- Problem, in other words, is to find

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

- The problem is called a *"0-1"* problem, because each item must be entirely accepted or rejected.

0-1 Knapsack problem: brute-force approach

Let's first solve this problem with a straightforward algorithm

- Since there are n items, there are 2^n possible combinations of items.
- We go through all combinations and find the one with the most total value and with total weight less or equal to W
- Running time will be $O(2^n)$

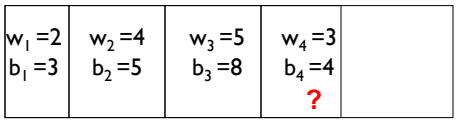
- Can we do better?
- Yes, with an algorithm based on dynamic programming
- We need to carefully identify the subproblems

Defining a Subproblem

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for S_k = *{items labeled 1, 2, .. k}*

- This is a valid subproblem definition.
- The question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?
- Unfortunately, we can't do that. Explanation follows….

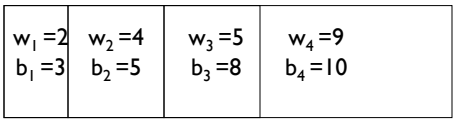
Defining a Subproblem



Max weight: W = 20

For S_4:

Total weight: 14;
total benefit: 20



For S_5:

Total weight: 20
total benefit: 26

S_5

S_4

Item #	Weight w_i	Benefit b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

Solution for S_4 is not part of the solution for S_5!!!

The Knapsack Problem And Optimal Substructure

- To show this for the 0-1 problem, consider the most valuable load weighing at most W pounds
- *If we remove item j from the load, what do we know about the remaining load?*
- A: remainder must be the most valuable load weighing at most $W - w_j$ that thief could take from museum, excluding item j

Defining a Subproblem (continued)

- As we have seen, the solution for S_4 is not part of the solution for S_5
- So our definition of a subproblem is flawed and we need another one!
- Let's add another parameter: w, which will represent the exact weight for each subset of items
- The subproblem then will be to compute $B[k,w]$

Recursive Formula for subproblems

- Recursive formula for subproblems:

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- It means, that the best subset of S_k that has total weight w is one of the two:

1) the best subset of S_{k-1} that has total weight w, **or**
2) the best subset of S_{k-1} that has total weight w-w_k plus the item k

Recursive Formula

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight w, either contains item k or not.
- First case: $w_k > w$. Item k can't be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable
- Second case: $w_k <= w$. Then the item k can be in the solution, and we choose the case with greater value

0-1 Knapsack Algorithm

```
for w = 0 to W
  B[0,w] = 0
for i = 0 to n
  B[i,0] = 0
  for w = 0 to W
       if w_i <= w // item i can be part of the solution
             if b_i + B[i-1,w-w_i] > B[i-1,w]
                   B[i,w] = b_i + B[i-1,w- w_i]
             else
                   B[i,w] = B[i-1,w]
       else B[i,w] = B[i-1,w]  // w_i > w
```

Running time

```
for w = 0 to W                    O(W)
  B[0,w] = 0
for i = 0 to n                    Repeat n times
  B[i,0] = 0
  for w = 0 to W                  O(W)
      < the rest of the code >
```

What is the running time of this algorithm?

O(nW)

Remember that the brute-force algorithm takes $O(2^n)$

Example

Let's run our algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

Example (2)

W \ i	0	1	2	3	4
0	0				
1	0				
2	0				
3	0				
4	0				
5	0				

for w = 0 to W
B[0,w] = 0

Example (3)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0				
2	0				
3	0				
4	0				
5	0				

for i = 0 to n
B[i,0] = 0

Example (4)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0 →	0			
2	0				
3	0				
4	0				
5	0				

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i=1
b_i=3
w_i=2
w=1
$w-w_i$ =-1

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

Example (5)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	**3**			
3	0				
4	0				
5	0				

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i=1
b_i=3
w_i=2
w=2
w-w_i =0

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	**3**			
4	0				
5	0				

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i=1
b_i=3
w_i=2
w=3
w-w_i=1

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (7)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	**3**			
5	0				

i=1
b_i=3
w_i=2
w=4
w-w_i=2

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (8)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	3			
5	0	**3**			

i=1
b_i=3
w_i=2
w=5
w-w_i=2

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (9)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	**0**		
2	0	3			
3	0	3			
4	0	3			
5	0	3			

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i=2
b_i=4
w_i=3
w=1
w-w_i=-2

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

Example (10)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	**3**		
3	0	3			
4	0	3			
5	0	3			

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i=2
b_i=4
w_i=3
w=2
w-w_i=-1

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

Example (11)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	**4**		
4	0	3			
5	0	3			

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

$i=2$
$b_i=4$
$w_i=3$
$w=3$
$w-w_i=0$

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (12)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	**4**		
5	0	3			

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

$i=2$
$b_i=4$
$w_i=3$
$w=4$
$w-w_i=1$

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (13)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	4		
5	0	3	**7**		

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i=2
b_i=4
w_i=3
w=5
w-w_i=2

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (14)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	**0**	
2	0	3	3	**3**	
3	0	3	4	**4**	
4	0	3	4		
5	0	3	7		

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i=3
b_i=5
w_i=4
w=1..3

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

Example (15)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	**5**	
5	0	3	7		

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i=3
b_i=5
w_i=4
w=4
w- w_i=0

```
if w_i <= w // item i can be part of the solution
   if b_i + B[i-1,w-w_i] > B[i-1,w]
      B[i,w] = b_i + B[i-1,w- w_i]
   else
      B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (15)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	5	
5	0	3	7	→ **7**	

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i=3
b_i=5
w_i=4
w=5
w- w_i=1

```
if w_i <= w // item i can be part of the solution
   if b_i + B[i-1,w-w_i] > B[i-1,w]
      B[i,w] = b_i + B[i-1,w- w_i]
   else
      B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (16)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0 →	**0**
2	0	3	3	3 →	**3**
3	0	3	4	4 →	**4**
4	0	3	4	5 →	**5**
5	0	3	7	7	

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

$i=3$
$b_i=5$
$w_i=4$
$w=1..4$

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

Example (17)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	3	3	3	3
3	0	3	4	4	4
4	0	3	4	5	5
5	0	3	7	7 →	**7**

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

$i=3$
$b_i=5$
$w_i=4$
$w=5$

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Comments

- This algorithm only finds the max possible value that can be carried in the knapsack
- To know the items that make this maximum value, an addition to this algorithm is necessary
- Please see LCS algorithm from the previous lecture for the example how to extract this data from the table we built