Untitled

February 10, 2018

Discrete Optimization course on Coursera : https://www.coursera.org/learn/discrete-optimization/home/welcome
Week 1:

1 Knap sack problem

```
In [15]: from time import time
         from IPython.display import Image
```

```
In [16]: # [value, weight]
         items = [ [1,2],[1,2],[1,2],[10,5],[10,5],[13,8],[7,3] ]
         K = 10
```

```
In [17]: n = len(items)
```

1.1 Greedy procedure

1. lightest first:

```
In [19]: def greedy1(K,items):
             n = len(items)
             # Sorting by ascending weight
             items.sort(key=lambda x: x[1])
             print("sorted by weight: %s" % items)
             selected_value = 0
             selected_weight = 0
             selected = []
             i = 0

             # Packing lightest items first
             while selected_weight < K:
                 if i>=n:
                     break;
                 if selected_weight+items[i][1] <= K:
                     selected.append(items[i])
                     selected_weight += items[i][1]
                     selected_value += items[i][0]
                 i += 1
```

```
                else:
                    break;
            return selected, selected_weight, selected_value

        selected, selected_weight, selected_value = greedy1(10,items)
        print("for a weight of %d" % selected_weight)
        print("for a value of %d" % selected_value)
        print("Selected items to pack are : \n %s " % selected)
```

```
sorted by weight: [[1, 2], [1, 2], [1, 2], [7, 3], [10, 5], [10, 5], [13, 8]]
for a weight of 9
for a value of 10
Selected items to pack are :
 [[1, 2], [1, 2], [1, 2], [7, 3]]
```

2. Most valuable first

```
In [20]: def greedy0(K,items):
             n = len(items)
             # Sorting by descending value
             items.sort(key=lambda x: x[0], reverse=True)
             print("sorted by value: %s" % items)
             selected_value = 0
             selected_weight = 0
             selected = []

             # Packing most valuable first if it still fits
             for i in range(n):
                 if selected_weight+items[i][1] <= K:
                     selected.append(items[i])
                     selected_weight += items[i][1]
                     selected_value += items[i][0]
             return selected, selected_weight, selected_value

         selected, selected_weight, selected_value = greedy0(10,items)
         print("for a weight of %d" % selected_weight)
         print("for a value of %d" % selected_value)
         print("Selected items to pack are : \n %s " % selected)
```

```
sorted by value: [[13, 8], [10, 5], [10, 5], [7, 3], [1, 2], [1, 2], [1, 2]]
for a weight of 10
for a value of 14
Selected items to pack are :
 [[13, 8], [1, 2]]
```

3. Value density

```
In [21]: def greedy(K,items):
             n = len(items)
             # Sorting by density : ie. value / weight
             items.sort(key=lambda x: x[0]/x[1], reverse=True)
             print("sorted by value density: %s" % items)
             selected_value = 0
             selected_weight = 0
             selected = []

             # Packing most valuable first if it still fits
             for i in range(n):
                 if selected_weight+items[i][1] <= K:
                     selected.append(items[i])
                     selected_weight += items[i][1]
                     selected_value += items[i][0]
             return selected, selected_weight, selected_value

         selected, selected_weight, selected_value = greedy(10,items)
         print("for a weight of %d" % selected_weight)
         print("for a value of %d" % selected_value)
         print("Selected items to pack are : \n %s " % selected)
```

```
sorted by value density: [[7, 3], [10, 5], [10, 5], [13, 8], [1, 2], [1, 2], [1, 2]]
for a weight of 10
for a value of 18
Selected items to pack are :
 [[7, 3], [10, 5], [1, 2]]
```

1.1.1 Overview:

- Depends on the chosen heuristic

- quick to design
  can be very fast to run

- no guarantee on quality
  feasibility needs to be easy
  quality varies between instances

1.2 Modeling

maximisation problem

- Decision variable:

$$\begin{cases} x_i = 1 := pack\ the\ i-th\ item x_i = 0 := Don't\ pack\ the\ i-th\ item \end{cases}$$

- Problem Constraint:

$$\sum_{i \in I} w_i x_i \leq K$$

- Objective function:

$$\sum_{i \in I} v_i x_i$$

1.3 Dynamic programming

- Finds Best Solution
- Divide and conquer / Bottom up computation technique
- We denote o(k,j) the optimal solution for the alternative problem with maximum capacity k and for the items 1..j to solve the original problem o(K,n)
- Procedure:
  - If item fits: O(k,j) = max{ O(k,j-1), v_j+O(k-w_j,j-1) }
  - If it doesn't O(k,j) = O(k,j-1)

  recursion starting from o(k,0)=0 for all k

1. Top down version : Inefficient and not a dynamic programming solution

```
In [7]: def o(k,j):
            if j==0: #first recursion term
                return 0
            elif items[j-1][1] <= k: #if item fits
                return max(o(k,j-1), items[j-1][0]+o(k-items[j-1][1],j-1) )
            else: #if item doesn't fit
                return o(k,j-1)

        start = time()
        print("Optimal reachable value : %d" % o(10,n))
        print("run time : %f ms" % (1000*(time()-start)))
```

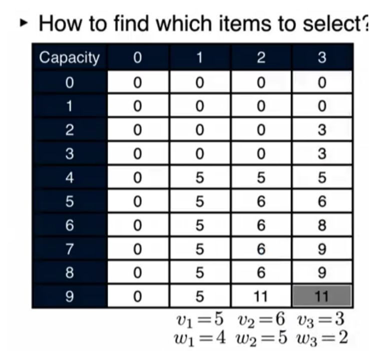
```
Optimal reachable value : 20
run time : 0.489473 ms
```

1. **Bottom up version :** O(Kn) solution but not polynomial because of the capacity K -> $log_2 K$ bits

   $\Rightarrow$ pseudo-polynomial

```
In [8]: Image("TableDynamicProgramming1.png")
```

Out[8]:

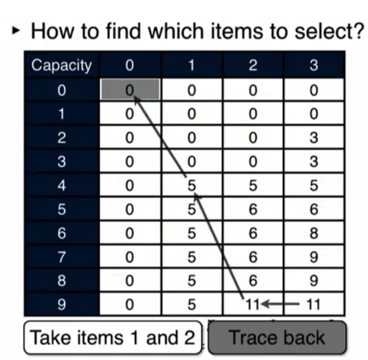


▸ How to find which items to select?

Capacity	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	3
3	0	0	0	3
4	0	5	5	5
5	0	5	6	6
6	0	5	6	8
7	0	5	6	9
8	0	5	6	9
9	0	5	11	11
		$v_1=5$ $w_1=4$	$v_2=6$ $w_2=5$	$v_3=3$ $w_3=2$

```
In [9]: Image("TableDynamicProgramming2.png")
```

Out[9]:



▸ How to find which items to select?

Capacity	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	3
3	0	0	0	3
4	0	5	5	5
5	0	5	6	6
6	0	5	6	8
7	0	5	6	9
8	0	5	6	9
9	0	5	11	11

```
In [10]: # A python improvement using memoization
         from functools import lru_cache

         @lru_cache(maxsize = 250)
         def dp_o(k,j,items):
             if j==0: #first recursion term
                 return 0
             elif items[j-1][1] <= k: #if item fits
                 return max(o(k,j-1), items[j-1][0]+o(k-items[j-1][1],j-1) )
             else: #if item doesn't fit
                 return o(k,j-1)

         start = time()
         print("Optimal reachable value : %d" % dp_o(K,n,items))
         print("run time : %f ms" % (1000*(time()-start)))
```

```
Optimal reachable value : 20
run time : 0.268459 ms
```

```
In [22]: def dynamic_prog(K,items):
             # Explicit dynamic programming implementation
             n = len(items)
             dp_table = []
             j_col = []

             # first column of zeros
             for k in range(K+1):
                 j_col.append(0)
             dp_table.append(j_col)
             # filling other columns
             for j in range(n):
                 j_col = []
                 for k in range(K+1):
                     if items[j][1] <= k: #item fits
                         j_col.append( max( dp_table[j][k], items[j][0]+dp_table[j][k-items[j][1
                     else:
                         j_col.append( dp_table[j][k] )
                 dp_table.append(j_col)

             return dp_table

         dp_table = dynamic_prog(10,items)
         start = time()
         print("Optimal reachable value : %d" % dp_table[n][K])
         print("run time : %f ms" % (1000*(time()-start)))
```

```
Optimal reachable value : 20
run time : 0.215054 ms
```

1.4 Branch and bound

In [24]:
```python
items2 = [[45,5],[48,8],[35,3]]
dynamic_prog(10,items2)[len(items2)][10]
```

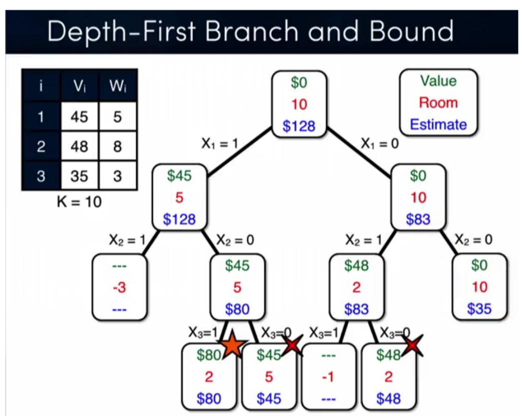
Out[24]: 80

maximize $45x_1 + 48x_2 + 35x_3$
subject to $\begin{cases} 5x_1 + 8x_2 + 3x_3 \leq 10 x_i \in \{0,1\} \end{cases} ,i \in \{1,2,3\}$
Branching: split problem into a number of subproblems ~ exhaustive search
Bounding: find an optimistic estimate; an upper bound for the maximization problems.

1. Relaxation: removing capacity constraint : Choosing all available items.

In [25]:
```python
Image("NoConstraint_B&B.png")
```

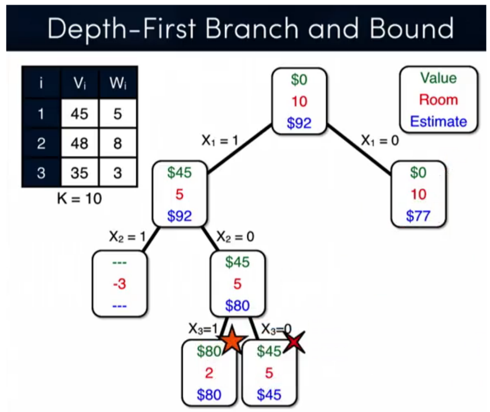
Out[25]:



1. Linear Relaxation: maximize $45x_1 + 48x_2 + 35x_3$
subject to $\begin{cases} 5x_1 + 8x_2 + 3x_3 \leq 10 0 \leq x_i \leq 1 \end{cases}$

In [27]:
```python
Image("LinearRelax_B&B.png")
```

Out[27]:



At each step, we bound by the value given by the linear relaxation which can be easily solved thanks to the greedy algorithm using density value and taking fractions of the objects if needed.

We stop the exploration as the estimate (77) is lower than the already retrieved solution (80)

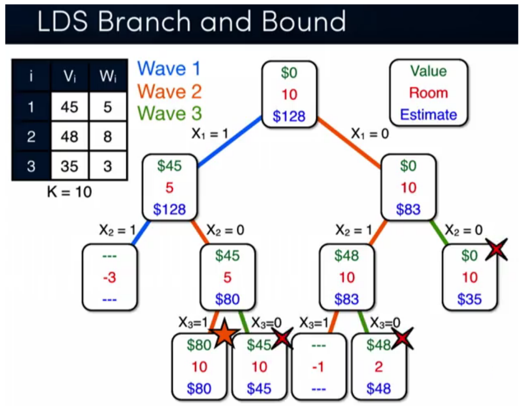
1.5 Search Strategies

Depth-first: go deeper in the tree and prune when a node's estimation is worse than the best solution found memory efficient : at most one whole branch so the number of the items #### Best-first: always explore the node with the best estimation #### Least discrepancy: - avoid mistakes (defined by trusting a greedy heuristic) - explore the space by allowing an increasing number of mistakes through waves

- probes the search space eg. heuristic is going left

```
In [28]: Image("LDS.png")
```

Out[28]:


LDS Branch and Bound
i
Vi
Wi
1
45
5
2
48
8
3
35
3
K = 10
Wave 1
Wave 2
Wave 3
Value
Room
Estimate
$0
10
$128
X1 = 1
X1 = 0
$45
5
$128
$0
10
$83
X2 = 1
X2 = 0
X2 = 1
X2 = 0

-3

$45
5
$80
$48
10
$83
$0
10
$35
X3=1
X3=0
X3=1
X3=0
$80
10
$80
$45
10
$45

-1

$48
2
$48