Dynamic Programming

June 15, 2020

1 Comparison between Dynamic Programming, Brute Force and Greedy algoritm on Knapsack problem

[11]:
```
import random
from itertools import product,combinations,compress
import numpy as np
import time
import matplotlib.pyplot as plt
import pandas as pd
import warnings
warnings.filterwarnings('ignore')
```

1.0.1 Function to generate the data

[12]:
```
def create_data(n):
    return [(random.randint(1,n*10),random.randint(1,n*10))for i in range(n)]
data = create_data(5)
print(data)
```

```
[(31, 28), (36, 8), (3, 15), (35, 43), (18, 13)]
```

1.0.2 Exhaustive search

[13]:
```
def brute_force(data,capacity):
    combinations = list(product([1,0],repeat = len(data)))
    weights,values = zip(*data)
    max_so_far = 0;
    for i in combinations:
        weight = np.dot(weights,i)
        value = np.dot(values,i)
        if weight <= capacity and value >= max_so_far:
            max_so_far = value
            combination = i
    return list(compress(data,combination))
```

[14]:
```python
capacity = 9
```

[15]:
```python
brute_force(data,capacity)
```

[15]:
```
[(3, 15)]
```

1.0.3 Dynamic Programming

[16]:
```python
def dynamic_programming(data,capacity):
    counter = 0
    table = np.zeros((len(data) +1,capacity +1))
    for i in range(1, len(data) + 1):
        value,weight = data[i-1][1],data[i-1][0]
        for j in range(1, capacity + 1):
            if weight > j:
                table[i,j] = table[i-1,j]
            else:
                table[i,j] = max(table[i-1,j], table[i-1, j-weight] + value)
    return table
print(dynamic_programming(data,9))
```

```
[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0. 15. 15. 15. 15. 15. 15. 15.]
 [ 0.  0.  0. 15. 15. 15. 15. 15. 15. 15.]
 [ 0.  0.  0. 15. 15. 15. 15. 15. 15. 15.]]
```

1.0.4 Greedy Algorithm - Approximation algorithm

[17]:
```python
def greedy(data,capacity):
    order = sorted(data,key = lambda t: t[0]/t[1])
    lst = []
    while capacity > 0 and order:
        item = order.pop(0)
        if capacity - item[0] >= 0:
            capacity = capacity - item[0]
            lst.append(item)
    return lst
```

[18]:
```python
greedy(data,capacity)
```

[18]:
```
[(3, 15)]
```

[19]:
```python
def evaluate(algorithms,minimum=1,maximum=20,jump=1):
    df = []
    for i in range(minimum,maximum,jump):
        data = create_data(i)
        capacity = sum([item[0] for item in data])
        lst = []
        for algorithm in algorithms:
            time1 = time.time()
            algorithm(data,capacity)
            lst.append(time.time() - time1)
        df.append(lst)
    return df
```

[20]:
```python
x = evaluate([greedy, brute_force,dynamic_programming])
df = pd.DataFrame(x, columns = ["Greedy","Brute Force","Dynamic Programming"])
df
```

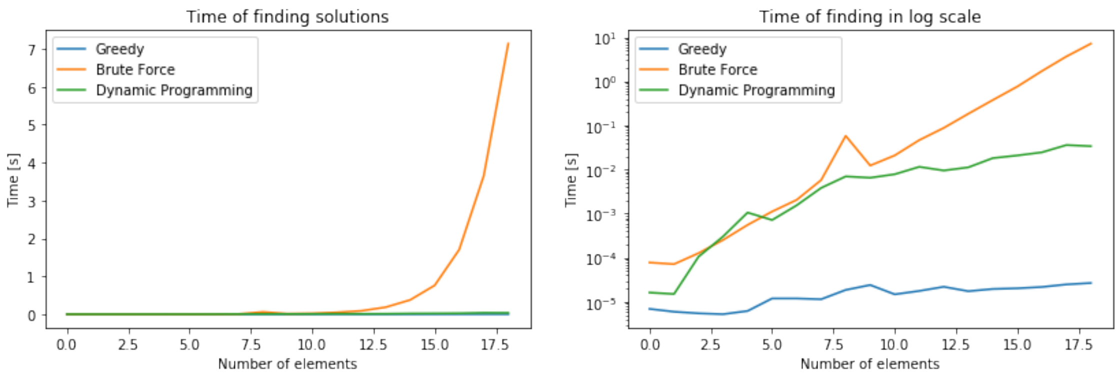
[20]:

	Greedy	Brute Force	Dynamic Programming
0	0.000007	0.000078	0.000016
1	0.000006	0.000072	0.000015
2	0.000005	0.000127	0.000105
3	0.000005	0.000252	0.000303
4	0.000006	0.000554	0.001056
5	0.000012	0.001114	0.000715
6	0.000012	0.002044	0.001539
7	0.000011	0.005763	0.003822
8	0.000019	0.057931	0.006960
9	0.000024	0.012288	0.006496
10	0.000015	0.020855	0.007800
11	0.000018	0.046382	0.011492
12	0.000022	0.087659	0.009484
13	0.000017	0.183370	0.011183
14	0.000020	0.375813	0.018119
15	0.000020	0.759866	0.020803
16	0.000022	1.703115	0.024482
17	0.000025	3.648328	0.035811
18	0.000027	7.135924	0.033750

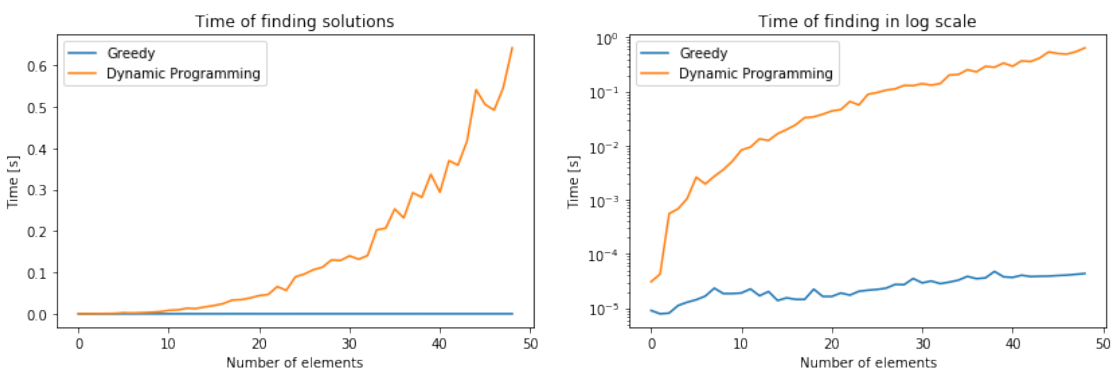
1.1 Time comparison graphs

[21]:
```python
def graphs(df):
    fig, (ax1,ax2) = plt.subplots(1,2, figsize=(14,4))
    ax1.set_xlabel("Number of elements")
    ax1.set_ylabel("Time [s]")
    ax2.set_xlabel("Number of elements")
    ax2.set_ylabel("Time [s]")
```

```python
    df.plot(kind = "line",ax=ax1,title = "Time of finding solutions")
    df.plot(logy=True,ax=ax2,title = "Time of finding in log scale")
    plt.show()
graphs(df)
```



[22]:
```python
x = evaluate([greedy,dynamic_programming],1,50)
df = pd.DataFrame(x, columns = ["Greedy","Dynamic Programming"])
df
graphs(df)
```



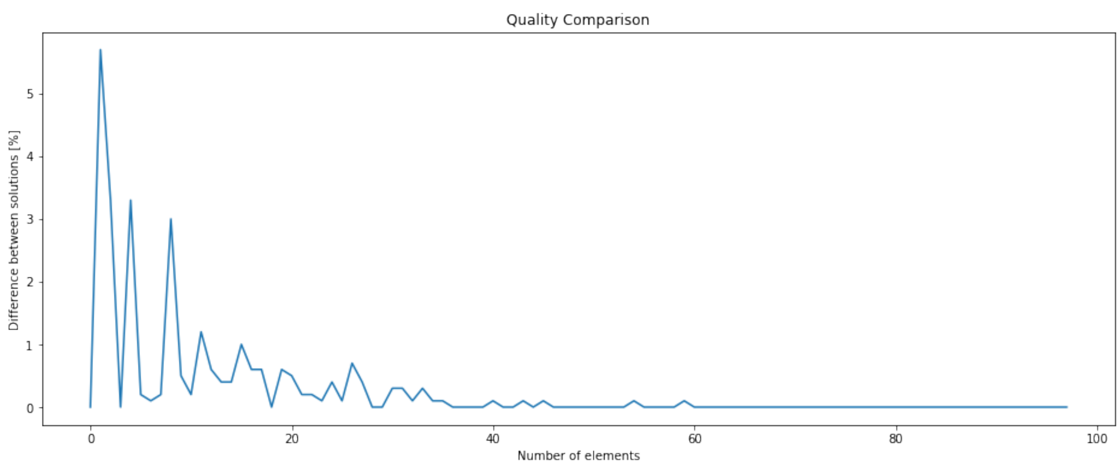
1.2 Quality Comparison graph

[8]:
```python
def evaluate1():
    df = []
    for i in range(2,100):
        lst = []
        for k in range(10):
            data = create_data(i)
            capacity = int(sum([item[0] for item in data])/2)
```

```
            greedy_solutions= greedy(data,capacity)
            aproximation = sum([item[1] for item in greedy_solutions])
            optimal =␣
↪int(dynamic_programming(data,capacity)[len(data)][capacity])
#          print(optimal,aproximation)
            comparison = int(((optimal- aproximation)/optimal)*100)
            lst.append(comparison)
        df.append(np.mean(lst))
#          print(i)
    return df
df = evaluate1()
# print(df)
```

[0.0, 5.7, 3.3, 0.0, 3.3, 0.2, 0.1, 0.2, 3.0, 0.5, 0.2, 1.2, 0.6, 0.4, 0.4, 1.0, 0.6, 0.6, 0.0, 0.6, 0.5, 0.2, 0.2, 0.1, 0.4, 0.1, 0.7, 0.4, 0.0, 0.0, 0.3, 0.3, 0.1, 0.3, 0.1, 0.1, 0.0, 0.0, 0.0, 0.0, 0.1, 0.0, 0.0, 0.1, 0.0, 0.1, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.1, 0.0, 0.0, 0.0, 0.0, 0.1, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

[9]:
```
plt.figure(figsize=(16,6))
plt.plot(df)
plt.title('Quality Comparison')
plt.xlabel('Number of elements')
plt.ylabel('Difference between solutions [%]')
plt.show()
```



1.2.1 Conclusions

Plots representing time comparison shows that both Dynamic Programming and Approximation Algorithm holds a huge advantage over exhaustive search.

Chart representing a quality comparison between Dynamic Programming and Approximation Algorithm can be split into two separate graphs. The first part of the graph with dense fluctuations describes strong dependence on a chance which is caused by a considerably small number of elements algorithms. Along with a small number of elements goes the smaller number of combinations which decreases the chance of obtaining the optimal solution. The second part could make one consider implementing the Approximation Algorithm for the relatively significant number of objects due to the fact that the algorithm nearly always obtains optimal solutions while maintaining better time complexity.

Knapsack problem as a decision problem belongs to NP-complete class which means that there does not exist algorithm providing correct optimal solution in polynomial time. Dynamic programing solves the problem in pseudo-polynomial time, however it depeneds on maximum capcity O(|N|* maximum capacity),so the 0-1 knapsack problem is a weakly NP-complete problem.

Time complexity of the algorithms: * **Dynamic Programming** time: O(|N|* CAPACITY) * **Exhaustive Search** time: O(2^|N|) * **Approximation Algorithm** time: O(|N|*log(|N|)

Created by: Maciej Filanowicz, AI, 145462