IND E 599 Data-Driven Optimization: An Implementation and Evaluation of Near-Optimal Online Optimization Algorithms

Due on June 5, 2020 at 5:00pm

Prof. Archis Ghate

Klaas Fiete Krutein

May 29, 2020

Problem Statement

This project implements the dynamic algorithms for online linear programs developed by Agrawal, Wang, and Ye (2014). The two algorithms presented by Agrawal et al. (2014) deliver near optimal solutions to linear problems where each column of the constraint matrix is revealed at a certain point in time. This problem is relevant for many situations in decision making when capacity is limited but the volume and reward of arriving requests is unknown but requires a decision. Thus, a decision variable needs to be set every time a new column of the constraint matrix is revealed without knowing what future columns will contain. In their paper, the authors introduce the One-Time Learning Algorithm (OLA) and the more advanced $1 - O(\epsilon)$ competitive Dynamic Learning Algorithm (DLA) that solve linear and integer programs of the above nature online. They analyze the competitive ratio and provide proofs for worst case bounds. However, the authors do not provide a data-backed example in their paper that shows the algorithms in practice. Thus, this project looks at reformulating the OLA and DLA algorithms for the 0/1 Knapsack Problem including implementation and testing. This problem was selected since it is well-known and its integer variables add another layer of complexity.

Data

The first two data sets used in this project were originally introduced by Kreher and Stinson (1999) for testing different models on combinatorics. They consist of knapsack problems of 15 and 24 items, respectively. Since the knapsack problem is an integer program and thus NP-hard, these datasets provide a decent size for offline optimization algorithms. However, since OLA and DLA are online algorithms and allow solving problems that evolve over many stages, a randomly generated synthetic dataset was designed. $1,000$ items were generated with uniformly drawn weights and profits between 50 and $5,000$, and between 100 and $200,000$, respectively. The knapsack capacity is fifty times the average item weight.

Methodology

The Offline 0/1 Knapsack Problem

The standard 0/1 knapsack problem is one of the most well-studied problems in combinatoric optimization and is presented in this section. Let n denote the number of items that the decision maker can choose to put in the knapsack. Further, let w_i denote the weight of item i, let r_i denote the revenues from selecting item i, and let C denote the capacity of the knapsack. The objective function seeks to select items such that revenues are maximized, subject to satisfying the capacity of the knapsack. Note that in the 0/1 knapsack items of equal weight and revenue are considered individually.

$$\max_{x \in X} \left\{ \sum_{i=1}^{n} r_i x_i \middle| \sum_{i=1}^{n} w_i x_i \leq C \wedge x_i \in \{0,1\}, \forall i \in I \right\} \tag{1}$$

The One-Time Learning Algorithm (OLA)

Moving to the online algorithms presented by Agrawal et al. (2014), we first introduce a reformulation of the 0/1 knapsack problem based on the simpler One-Time Learning Algorithm (OLA). Consider that, for the online algorithm, we do not have complete information about all items yet and assume that they get revealed randomly over time. Thus, the algorithm starts off with incomplete information, defined on the input until time $s = \epsilon N$. Note that $\epsilon \in [0, 1]$ is the determining parameter here that denotes the fraction of information of the total number of stages that is available, such that s is assumed to be integer. This provides the following relaxed partial primal formulation, where the discount of $1 - \epsilon$ is used to ensure that the decisions will not violate the constraints with high probability.

$$\max \quad \left\{ \sum_{t=1}^{s} r_t x_t \middle| \sum_{t=1}^{s} w_t x_t \leq (1-\epsilon)\frac{s}{n} C \wedge x_t \in [0,1], \forall t = 1, ..., s \right\} \tag{2}$$

The corresponding dual problem is thus defined accordingly.

$$\min \quad \left\{ (1-\epsilon)\frac{s}{n} Cp + \sum_{t=1}^{s} y_t \middle| w_t p + y_t \geq r_t, t = 1, ..., s, \wedge\ p, y_t \geq 0, t = 1, ..., s \right\} \tag{3}$$

Since this is the dual problem, we can observe that the optimal solution to $\hat{p}$ denotes the price for each item selected. The allocation rule for any future iteration $t > s$ is defined as:

$$x_t(p) = \begin{cases} 0, & \text{if } r_t \leq p^T a_t \\ 1, & \text{if } r_t > p^T a_t \end{cases} \tag{4}$$

With this decision rule, only items that cross a certain revenue threshold will be selected. Note that using the dual price vector obtained at s, each decision in future iterations is based on the threshold of this price vector. This enables the user to solve a problem of small size in the early stages to learn the decision vector. Agrawal et al. (2014)'s OLA algorithm then iterates as described in Algorithm 1. We can see that the learning period (all steps until s) does not contribute to the objective value and thus, the problem will only reach near-optimal status as proved in the paper. Note that integer problems like the 0/1 knapsack problem do not cause complications since OLA makes binary decisions.

Algorithm 1: OLA

```
Result: $x_t$
initialize $x_t = 0, \forall t \leq s$;
$\hat{p}$ := Solve partial dual problem for $t = 1, .., s$;
for $t = s+1, s+2, ..., n$ do
    if $w_t x_t(p) \leq C - \sum_{j=1}^{t-1} w_j x_j$ then
        $x_t = x_t(p)$;
    else
        $x_t = 0$;
end
```

Further, adapted from Agrawal et al. (2014)'s definitions on the random permutation model, the competitive ratio can only be ensured for this knapsack problem, if the following lower bound on C for the OLA is satisfied. Thus, the performance of the algorithm is dependent on the choice of ϵ.

$$C \geq \Omega\left(\frac{6\log(n/\epsilon)}{\epsilon^3}\right) \tag{5}$$

The Dynamic Learning Algorithm (DLA)

The improved Dynamic Learning Algorithm (DLA) differs from the OLA in that it allows for a lower bound on the RHS (C) and that it generally achieves better results. The DLA further updates the price vector every time the history doubles ($t = \epsilon n, 2\epsilon n, 4\epsilon n, ...$). We thus obtain $\hat{p}^l$ from the dual of the following updated primal problem:

$$\max \quad \left\{\sum_{t=1}^{l} r_t x_t \middle| \sum_{t=1}^{l} w_t x_t \leq (1-h_l)\frac{l}{n}C, x_t \in [0,1], \forall t = 1, ..., l\right\} \tag{6}$$

where $h_l = \epsilon\sqrt{\frac{n}{l}}$. For solving the DLA, the same allocation rule as before remains (4). The corresponding algorithm is described in Algorithm 2.

Algorithm 2: DLA

```
**Result:** $x_t$
initialize $t_0 = \epsilon n$, set $x_t = 0, \forall t \leq t_0$;
$\hat{p}$ := Solve partial dual problem (6) for $t = 1, .., t_0$;
for $t = t_0 + 1, t_0 + 2, ...$ do
    $\hat{x}_t = x_t(\hat{p}^l)$, $l = 2^r \epsilon n$, where $r$ is largest integer such that $l < t$;
    if $w_t \hat{x}_t(p) \leq C - \sum_{j=1}^{t-1} w_j x_j$ then
        $x_t = \hat{x}_t$;
    else
        $x_t = 0$;
end
```

In comparison to the OLA, we see that the DLA updates the dual price vector every time the history doubles. The authors further prove in their paper that DLA allows an improved lower bound on C compared to OLA.

$$C \geq \Omega\left(\frac{\log(n/\epsilon)}{\epsilon^2}\right) \tag{7}$$

In the following section of the report, we will show numerical results from both algorithms on the data sets presented above. The algorithms were implemented in python and the corresponding code is available at the end of this document.

Presentation and Discussion of numerical results

To investigate the sensitivity of the solution quality to the setting of ϵ, we first need to verify that the lower bounds on C are satisfied for each problem and accordingly can derive lower bounds for ϵ for each problem.

Table 1: Lower bounds on ϵ

Data set	No. of items (N)	Capacity (C)	ϵ-bound (OLA)	ϵ-bound (DLA)
Kreher15	15	750	$\epsilon \geq 0.3139$	$\epsilon \geq 0.0832$
Kreher24	24	6,404,180	$\epsilon \geq 0.0189$	$\epsilon \geq 0.0012$
Synthetic1000	1,000	126,050	$\epsilon \geq 0.0767$	$\epsilon \geq 0.0096$

Table 1 shows the importance of the RHS (C) to determine the lower bound for ϵ. The Kreher15 data set and its capacity are almost too small for a meaningful implementation of the OLA since for competitiveness of the solution, the size of the training problem needs to be almost a third of the size of the offline problem. We illustrate the effect of the ϵ choice on the objective function for the Kreher24 and the Synthetic problem in Figure 1. The results for OLA and DLA were obtained from implementations using the lower bounds from Table 1, and for the offline knapsack from an implementation using the cvxpy package.

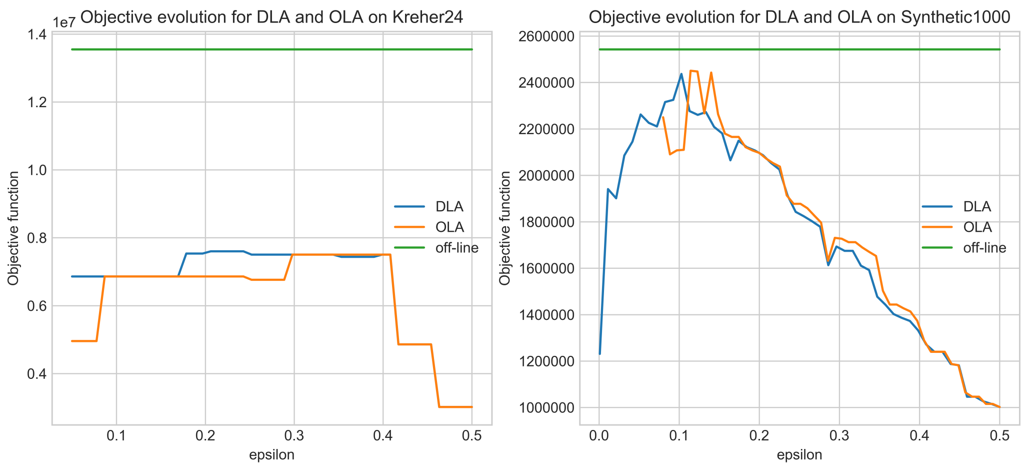


Figure 1: Effect of ϵ choice on objective function value

The graph for the the Kreher24 data set shows that the efficiency of both OLA and DLA are low for a problem with few stages, no matter the choice of ϵ. The reason for this is that the small problem size causes that only few decisions can be made after the training phase and thus, every decision heavily impacts the objective function and compensating for the items that are not selected during the training phase is almost impossible. However, the right graph shows a different picture for the Synthetic1000 data set. The algorithms shows best results for ϵ around 0.13 for OLA and 0.1 for DLA. However, despite that DLA allows for a lower bound on C, the difference in solution quality does not appear large. These results show that the online algorithms require a relatively high number of stages to actually approximate the competitiveness claimed through the proofs in the original paper.

Further, Figure 1 shows that, even for a larger problems the careful choice of ϵ is crucial to the algorithms success. We can further investigate the effect of the knapsack capacity on the solution quality for DLA and OLA for a fixed ϵ value. Figure 2 illustrates this for a range of knapsack capacities applied to the Synthetic1000 data set.

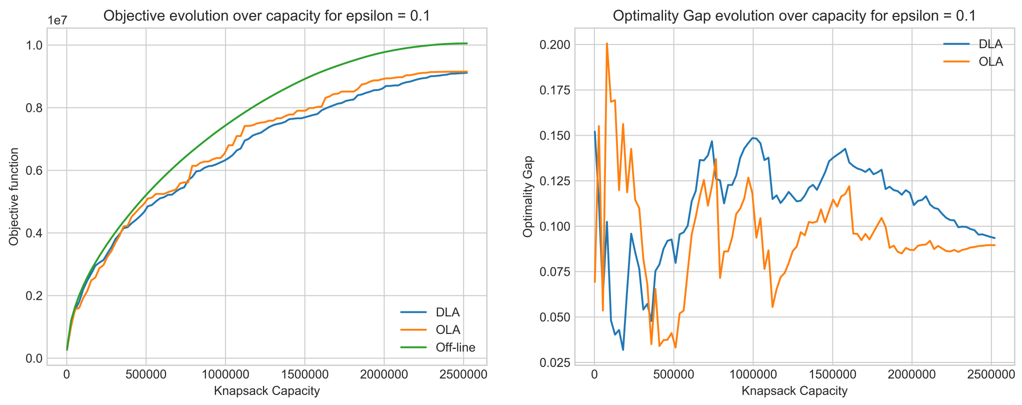


Figure 2: Effect of knapsack capacity on objective function value OLA vs. DLA

It can be observed that for large knapsack capacities, both the DLA and OLA approximate the $(1 - O(\epsilon))$ optimality gap. However, for a smaller knapsack capacity in this instance the DLA performs significantly better for this test problem than the OLA. This behavior shifts for larger knapsack capacities, even though the effect is not as extreme. We should mention here that the distribution of weights and profits is uniformly distributed over all stages to keep the algorithms comparable, since a shift in distribution parameters over time would be advantageous for the DLA since the OLA cannot update its price vector to consider this change. However, the size of the knapsack appears to have a strong influence on how either of the algorithms perform, extending further than the initially determined lower bound.

Conclusions

Agrawal et al. (2014) introduce their Dynamic Learning Algorithm as a solution to problems which get revealed over time with a $(1 - O(\epsilon))$ gap. The advantage of the Dynamic Learning Algorithm over alternative online algorithms is that it theoretically allows to reach a high competitive ratio of $(1 - O(\epsilon))$, where ϵ is a preset measure. In this study, we have shown an implementation of both the DLA and the simpler OLA to the 0/1-Knapsack problem and numerical tests on three instances of the problem. At first, we were able to confirm the strong influence of the performance of the algorithm on the choice of ϵ. We have further shown that while the optimality bounds shown in the paper can eventually be achieved for large RHS in the problem, the performance still fluctuates a lot, for any fixed ϵ value, when considering a range of values for the RHS. This illustrates that the choice of ϵ is not only imported considering a lower performance bound, but rather poses a new optimization problem by itself. This opens way for a future research direction. The problem of optimizing the ϵ value for the performance of a given problem and adjusting it accordingly is instrumental in training the model efficiently. This is particularly true since the column arrivals are supposedly random. Future research in this area would be valuable for use in practice.

References

Agrawal, S., Wang, Z., & Ye, Y. (2014). A Dynamic near-optimal algorithm for online linear programming. *Operations Research*, *62*(4), 876–890. doi: 10.1287/opre.2014.1289

Kreher, D. L., & Stinson, D. R. (1999). Combinatorial Algorithms: Generation, Enumeration, and Search. *CRC Press*, 33–40.

2020_05_15_Project_implementation

May 27, 2020

[1]:
```
# load data
filepath = "/Users/fietekrutein/Documents/University/University of Washington/
↪Courses/2020 Q2/IND E 599 Data-driven Optimization/homework/project/datasets/"
```

[2]:
```
# problem 1
cost1 = open(filepath + "p07_c.txt","r")
weights1 = open(filepath + "p07_w.txt", "r")
profits1 = open(filepath + "p07_p.txt", "r")
solution1 = open(filepath + "p07_s.txt", "r")

c1_data = cost1.readlines()
w1_data = weights1.readlines()
p1_data = profits1.readlines()
s1_data = solution1.readlines()
cost1.close()
weights1.close()
profits1.close()
solution1.close()

for i in range(len(c1_data)):
    c1_data[i] = c1_data[i].replace('\n', '')
for i in range(len(w1_data)):
    w1_data[i] = w1_data[i].replace('\n', '')
for i in range(len(p1_data)):
    p1_data[i] = p1_data[i].replace('\n', '')
for i in range(len(s1_data)):
    s1_data[i] = s1_data[i].replace('\n', '')
```

[3]:
```
# problem 2
cost2 = open(filepath + "p08_c.txt","r")
weights2 = open(filepath + "p08_w.txt", "r")
profits2 = open(filepath + "p08_p.txt", "r")
solution2 = open(filepath + "p08_s.txt", "r")

c2_data = cost2.readlines()
w2_data = weights2.readlines()
p2_data = profits2.readlines()
```

```
s2_data = solution2.readlines()
cost2.close()
weights2.close()
profits2.close()
solution2.close()

for i in range(len(c2_data)):
    c2_data[i] = c2_data[i].replace('\n', '')
for i in range(len(w2_data)):
    w2_data[i] = w2_data[i].replace('\n', '')
for i in range(len(p2_data)):
    p2_data[i] = p2_data[i].replace('\n', '')
for i in range(len(s2_data)):
    s2_data[i] = s2_data[i].replace('\n', '')
```

[4]:
```
c2_data
```

[4]:
```
['6404180']
```

[5]:
```
import numpy as np
import cvxpy as cp
import time
```

[6]:
```
# regular knapsack implementation
def knapsack(capacity_data, weights_data, profits_data):

    capacity = np.array(capacity_data)
    weights = np.array(weights_data)
    profits = np.array(profits_data)

    # The variable we are solving for
    selection = cp.Variable(len(weights), boolean=True)

    # The sum of the weights should be less than or equal to the capacity
    weight_constraint = weights * selection <= capacity

    # Our total profit is the sum of the item profits
    total_profit = profits * selection

    # Problem definition
    knapsack_problem = cp.Problem(cp.Maximize(total_profit), [weight_constraint])

    # Solving the problem
    knapsack_problem.solve()

    optimal_profit = total_profit.value
    optimal_selection = selection.value
    #print("Optimal solution:", optimal_profit)
    #print("Optimal set selection:", optimal_selection)

    return(optimal_profit, optimal_selection)
```

[7]:
```
### ONE TIME LEARNING ALGORITHM
```

[8]:
```
def dual_partial_knapsack(epsilon, capacity_data, weights_data, profits_data):

    # define time s = epsilon x n
    n = len(weights_data)
    s = int(np.floor(epsilon * n))

    # subsetting data until time s
    capacity = int(np.array(capacity_data))
    weights = np.array(weights_data)[0:s]
    profits = np.array(profits_data)[0:s]

    ## DUAL PROBLEM

    # The variable we are solving for
    p = cp.Variable(nonneg = True)
    y = cp.Variable(len(weights), nonneg = True)

    # the dual constraint for every time step t until s
    constraints = []
    for i in range(s):
        constraints += [
            weights[i] * p + y[i] >= profits[i]
        ]

    # dual objective function
    dual_obj = (1 - epsilon) * s/n * capacity * p + cp.sum(y)

    # dual partial knapsack problem
    dual_partial = cp.Problem(cp.Minimize(dual_obj), constraints)

    # Solving the problem
    dual_partial.solve(solver = cp.GUROBI) # dependence on solver

    optimal_dual = dual_obj.value
    optimal_p = p.value
    optimal_y = y.value

    #print("Optimal dual solution:", optimal_dual)
    #print("Optimal price:", optimal_p)
    #print("Optimal dual vars:", optimal_y)
    return(optimal_p)
```

[9]:
```
def allocation_rule(phi, p, a):
    if phi <= p * a:
        x = 0
    elif phi > p * a:
        x = 1
    return(x)
```

[10]:
```
def existing_weight(weights, x, t):
    prev = 0
    for i in range(t):
        prev += weights[i] * x[i]
    return(prev)
```

[11]:
```
# Implement One-time learning algorithm (OLA)

def OLA_knapsack(epsilon, capacity_data, weights_data, profits_data):

    # define time s = epsilon x n
    n = len(weights_data)
    s = int(np.floor(epsilon * n))

    # reading in data
    capacity = np.array(capacity_data)
    capacity = capacity.astype(np.float)
    weights = np.array(weights_data)
    weights = weights.astype(np.float)
    profits = np.array(profits_data)
    profits = profits.astype(np.float)

    start_time = time.time() # record start time

    ## OLA steps

    # 1. Initialize x_t = 0 for all t <= s, # with phat defined as above
    x = np.empty(n)
    phat = dual_partial_knapsack(epsilon, capacity_data, weights_data,␣
 ↪profits_data)
    for i in range(s):
        #if weights[i] * allocation_rule(profits[i], phat, weights[i]) <=␣
 ↪(capacity - existing_weight(weights, x, i)):
        #    x[i] = allocation_rule(profits[i], phat, weights[i])
        #else:
        x[i] = 0
    # 2. perform updates for
    for i in range(s,n):
        if weights[i] * allocation_rule(profits[i], phat, weights[i]) <=␣
 ↪(capacity - existing_weight(weights, x, i)):
            x[i] = allocation_rule(profits[i], phat, weights[i])
        else:
            x[i] = 0

    objective = np.dot(profits, x)
    total_weight = np.dot(weights, x)
    slack = float(capacity - total_weight)

    end_time = time.time() # record end time
    run_time = end_time - start_time

    return(objective, total_weight, slack, x, run_time)
```

[12]:
```
### DYNAMIC LEARNING ALGORITHM
```

[13]:
```
# Implement Dynamic learning algorithm (DLA)

def DLA_knapsack(epsilon, capacity_data, weights_data, profits_data):

    # define time s = epsilon x n
    n = len(weights_data)
    s = int(np.floor(epsilon * n))

    # reading in data
    capacity = np.array(capacity_data)
    capacity = capacity.astype(np.float)
    weights = np.array(weights_data)
    weights = weights.astype(np.float)
    profits = np.array(profits_data)
    profits = profits.astype(np.float)

    start_time = time.time() # record start time

    ## DLA steps

    # 1. Initialize x_t = 0 for all t <= s, # with phat defined as above
    x = np.empty(n)
    phat = dual_partial_knapsack(epsilon, capacity_data, weights_data,␣
 ↪profits_data)
    for i in range(s):
        #if weights[i] * allocation_rule(profits[i], phat, weights[i]) <=␣
 ↪(capacity - existing_weight(weights, x, i)):
        #    x[i] = allocation_rule(profits[i], phat, weights[i])
        #else:
        x[i] = 0
    r = 1 # initialize r
    l = (2 ** r) * epsilon * n # initialize l
    h = epsilon * np.sqrt(n/l) # initialize h

    # 2. perform updates for
    for i in range(s,n):
        if i >= l: # once t reaches l, re-solve linear program
            phat = dual_partial_knapsack(h, capacity_data, weights_data,␣
 ↪profits_data) # resolve the program
            r += 1 # update r
            l = (2 ** r) * epsilon * n # update l
            h = epsilon * np.sqrt(n/l) # update h
        if weights[i] * allocation_rule(profits[i], phat, weights[i]) <=␣
 ↪(capacity - existing_weight(weights, x, i)):
            x[i] = allocation_rule(profits[i], phat, weights[i]) # assign cost.
        else:
            x[i] = 0

    objective = np.dot(profits, x)
    total_weight = np.dot(weights, x)
    slack = float(capacity - total_weight)

    end_time = time.time() # record end time
    run_time = end_time - start_time

    return(objective, total_weight, slack, x, run_time)
```

[14]:
```
### SOLVE FOR EACH PROBLEM
```

[15]:
```
# Test data instances
# solutions from regular knapsack:
pr1, sol1 = knapsack(c1_data, w1_data, p1_data)
pr2, sol2 = knapsack(c2_data, w2_data, p2_data)
```

```
Using license file /Users/fietekrutein/gurobi.lic
Academic license - for non-commercial use only
```

[16]:
```
# OLA Results
# Test data 1
epsilon = 0.1
profit, weight, slack, x, runtime = OLA_knapsack(epsilon, c1_data, w1_data,␣
 ↪p1_data)
print("Total profit obtained:", profit)
print("Weight of knapsack:", weight)
print("Slack in knapsack:", slack)
print("Optimal selection:", x)
print("Run time:", runtime)
```

```
Total profit obtained: 994.0
Weight of knapsack: 507.0
Slack in knapsack: 243.0
Optimal selection: [0. 0. 1. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 1. 1.]
Run time: 0.024272918701171875
```

[17]:
```
# Test data 2
epsilon = 0.1
profit, weight, slack, x, runtime = OLA_knapsack(epsilon, c2_data, w2_data,␣
 ↪p2_data)
print("Total profit obtained:", profit)
print("Weight of knapsack:", weight)
print("Slack in knapsack:", slack)
print("Optimal selection:", x)
print("Run time:", runtime)
```

```
Total profit obtained: 6859456.0
Weight of knapsack: 3187471.0
Slack in knapsack: 3216709.0
Optimal selection: [0. 0. 0. 0. 0. 1. 0. 0. 0. 1. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0.
0. 1. 0. 1.]
Run time: 0.0332331657409668
```

[18]:
```
# DLA Results
# Test data 1
epsilon = 0.1
profit, weight, slack, x, runtime = DLA_knapsack(epsilon, c1_data, w1_data,␣
 ↪p1_data)
print("Total profit obtained:", profit)
print("Weight of knapsack:", weight)
print("Slack in knapsack:", slack)
print("Optimal selection:", x)
print("Run time:", runtime)
```

```
Total profit obtained: 1377.0
Weight of knapsack: 707.0
Slack in knapsack: 43.0
Optimal selection: [0. 0. 1. 0. 0. 0. 1. 1. 1. 0. 1. 0. 0. 1. 1.]
Run time: 0.08611106872558594
```

[19]:
```
# Test data 2
epsilon = 0.1
profit, weight, slack, x, runtime = DLA_knapsack(epsilon, c2_data, w2_data,␣
 ↪p2_data)
```

```
print("Total profit obtained:", profit)
print("Weight of knapsack:", weight)
print("Slack in knapsack:", slack)
print("Optimal selection:", x)
print("Run time:", runtime)
```

```
Total profit obtained: 6859456.0
Weight of knapsack: 3187471.0
Slack in knapsack: 3216709.0
Optimal selection: [0. 0. 0. 0. 0. 1. 0. 0. 0. 1. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0.
0. 1. 0. 1.]
Run time: 0.09398078918457031
```

[20]:
```
### EXPERIMENTS ON EPSILON VALUE CHOICE
```

[ ]:
```

```

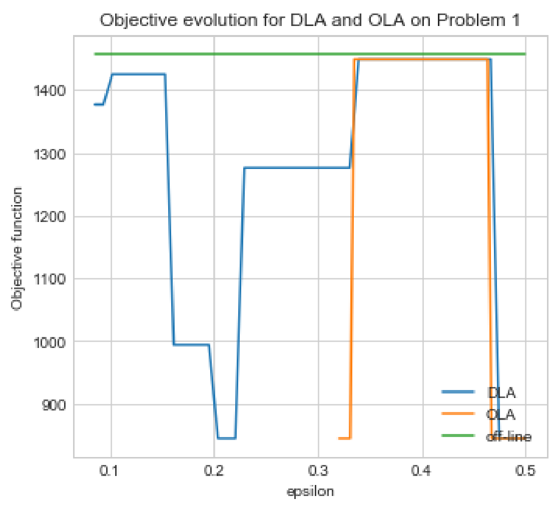
[21]:
```
# let's do it for our data examples
```

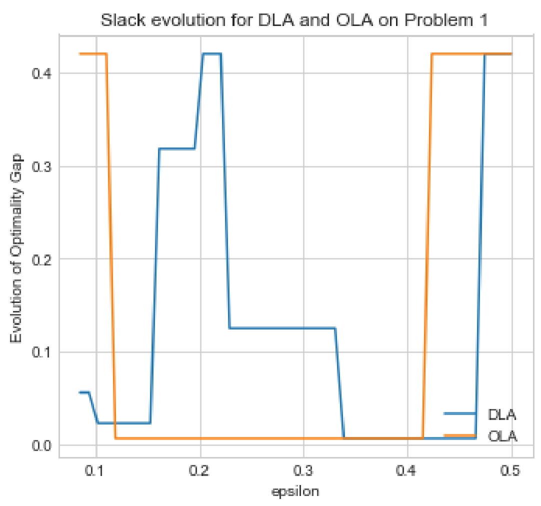
[22]:
```
# now run on OLA
total_profits = []
slack_evolve = []
weight_evolve = []
runtimes = []
epsilon_vals = np.linspace(0.32,0.5,50)

for k in epsilon_vals:
    profit, weight, slack, x, runtime = OLA_knapsack(k, c1_data, w1_data,␣
 ↪p1_data)
    #print("Epsilon value: ", k)
    #print("Total profit obtained:", profit)
    #print("Weight of knapsack:", weight)
    #print("Slack in knapsack:", slack)
    #print("")
    #print("Optimal selection:", x)
    total_profits.append(profit)
    slack_evolve.append(slack)
    weight_evolve.append(weight)
    runtimes.append(runtime)

OLA_results = np.array((epsilon_vals, np.array(total_profits), np.
 ↪array(slack_evolve), np.array(weight_evolve), np.array(runtimes)))
```

[23]:
```
# now run on DLA
total_profits = []
slack_evolve = []
weight_evolve = []
```

```
runtimes = []
epsilon_vals = np.linspace(0.085,0.5,50)

for k in epsilon_vals:
    profit, weight, slack, x, runtime = DLA_knapsack(k, c1_data, w1_data,␣
 ↪p1_data)
    #print("Epsilon value: ", k)
    #print("Total profit obtained:", profit)
    #print("Weight of knapsack:", weight)
    #print("Slack in knapsack:", slack)
    #print("Optimal selection:", x)
    #print("")
    total_profits.append(profit)
    slack_evolve.append(slack)
    weight_evolve.append(weight)
    runtimes.append(runtime)

DLA_results = np.array((epsilon_vals, np.array(total_profits), np.
 ↪array(slack_evolve), np.array(weight_evolve), np.array(runtimes)))
```

[24]:
```
from matplotlib import pyplot as plt

optimal = np.repeat(pr1, DLA_results.shape[1])
plt.style.use('seaborn-whitegrid')

plt.figure(figsize=[12,5])

plot1 = plt.subplot(121)
plt.plot(DLA_results[0], DLA_results[1], label = 'DLA')
plt.plot(OLA_results[0], OLA_results[1], label = 'OLA')
plt.plot(DLA_results[0], optimal, label = 'off-line')
plt.xlabel("epsilon")
plt.ylabel("Objective function")
plt.legend(loc="lower right")
plot1.set_title("Objective evolution for DLA and OLA on Problem 1")

plot2 = plt.subplot(122)
plt.plot(DLA_results[0], 1-(DLA_results[1]/optimal), label = 'DLA')
plt.plot(DLA_results[0], 1-(OLA_results[1]/optimal), label = 'OLA')
plt.xlabel("epsilon")
plt.ylabel("Evolution of Optimality Gap")
plt.legend(loc="lower right")
plot2.set_title("Slack evolution for DLA and OLA on Problem 1")

plt.savefig('Problem 1 epsilon.png', dpi = 800)

plt.show()
```





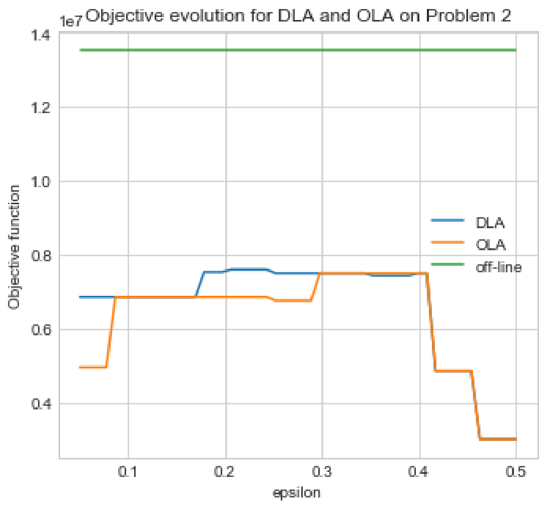
[25]:
```
# Now lets repeat this for problem 2
```

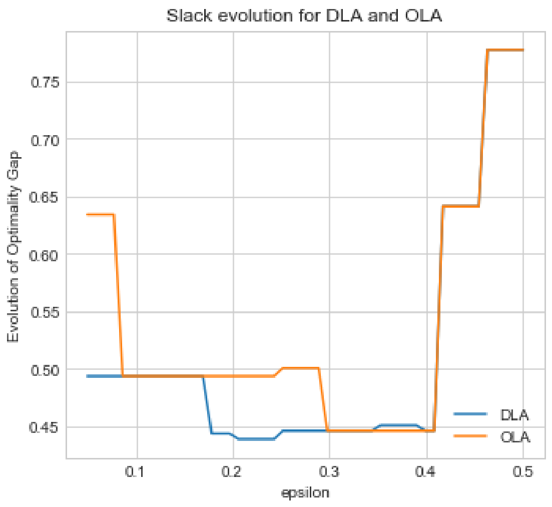
[26]:
```
# now run on OLA
total_profits = []
slack_evolve = []
weight_evolve = []
runtimes = []
epsilon_vals = np.linspace(0.05,0.5,50)

for k in epsilon_vals:
    profit, weight, slack, x, runtime = OLA_knapsack(k, c2_data, w2_data,␣
↪p2_data)
    #print("Epsilon value: ", k)
    #print("Total profit obtained:", profit)
    #print("Weight of knapsack:", weight)
    #print("Slack in knapsack:", slack)
    #print("")
    #print("Optimal selection:", x)
    total_profits.append(profit)
    slack_evolve.append(slack)
    weight_evolve.append(weight)
    runtimes.append(runtime)

OLA_results = np.array((epsilon_vals, np.array(total_profits), np.
↪array(slack_evolve), np.array(weight_evolve), np.array(runtimes)))
```

[27]:
```
# now run on DLA
total_profits = []
slack_evolve = []
weight_evolve = []
```

```
runtimes = []
epsilon_vals = np.linspace(0.05,0.5,50)

for k in epsilon_vals:
    profit, weight, slack, x, runtime = DLA_knapsack(k, c2_data, w2_data,␣
 ↪p2_data)
    #print("Epsilon value: ", k)
    #print("Total profit obtained:", profit)
    #print("Weight of knapsack:", weight)
    #print("Slack in knapsack:", slack)
    #print("Optimal selection:", x)
    #print("")
    total_profits.append(profit)
    slack_evolve.append(slack)
    weight_evolve.append(weight)
    runtimes.append(runtime)

DLA_results = np.array((epsilon_vals, np.array(total_profits), np.
 ↪array(slack_evolve), np.array(weight_evolve), np.array(runtimes)))
```

```
[28]: from matplotlib import pyplot as plt

optimal = np.repeat(pr2, DLA_results.shape[1])
plt.style.use('seaborn-whitegrid')

plt.figure(figsize=[12,5])

plot1 = plt.subplot(121)
plt.plot(DLA_results[0], DLA_results[1], label = 'DLA')
plt.plot(OLA_results[0], OLA_results[1], label = 'OLA')
plt.plot(DLA_results[0], optimal, label = 'off-line')
plt.xlabel("epsilon")
plt.ylabel("Objective function")
plt.legend(loc="center right")
plot1.set_title("Objective evolution for DLA and OLA on Problem 2")

plot2 = plt.subplot(122)
plt.plot(DLA_results[0], 1-(DLA_results[1]/optimal), label = 'DLA')
plt.plot(OLA_results[0], 1-(OLA_results[1]/optimal), label = 'OLA')
plt.xlabel("epsilon")
plt.ylabel("Evolution of Optimality Gap")
plt.legend(loc="lower right")
plot2.set_title("Slack evolution for DLA and OLA")

plt.savefig('Problem 2 epsilon.png', dpi = 800)

plt.show()
```





[29]:
```
epsilon_vals = np.linspace(0.01,0.5,50)
```

[30]:
```
# Now, let's generate our own data for a larger instance and test the algorithm␣
↪on this.
np.random.seed(123)

test_weights = np.round(np.random.uniform(50,5000, 1000),0)
test_profits = np.round(np.random.uniform(100,20000, 1000),0)
test_capacity = np.floor(np.mean(test_weights)) * 50
```

[31]:
```
test_capacity
```

[31]:
```
126050.0
```

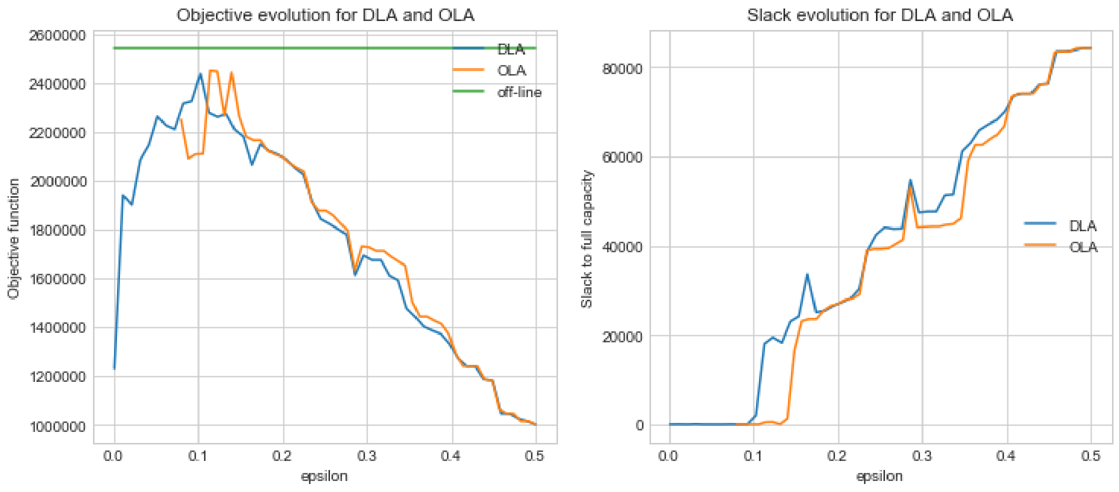
[32]:
```
# lets run it on the optimal knapsack
pr3, sol3 = knapsack(test_capacity, test_weights, test_profits)
```

[33]:
```
# now run on OLA
total_profits = []
slack_evolve = []
weight_evolve = []
runtimes = []
epsilon_vals = np.linspace(0.08,0.5,50)

for k in epsilon_vals:
    profit, weight, slack, x, runtime = OLA_knapsack(k, test_capacity,␣
↪test_weights, test_profits)
    #print("Epsilon value: ", k)
    #print("Total profit obtained:", profit)
    #print("Weight of knapsack:", weight)
    #print("Slack in knapsack:", slack)
    #print("")
    #print("Optimal selection:", x)
    total_profits.append(profit)
    slack_evolve.append(slack)
    weight_evolve.append(weight)
    runtimes.append(runtime)

OLA_results_S = np.array((epsilon_vals, np.array(total_profits), np.
 ↪array(slack_evolve), np.array(weight_evolve), np.array(runtimes)))
```

[34]:
```
# now run on DLA
total_profits = []
slack_evolve = []
weight_evolve = []
runtimes = []
epsilon_vals = np.linspace(0.001,0.5,50)

for k in epsilon_vals:
    profit, weight, slack, x, runtime = DLA_knapsack(k, test_capacity,␣
 ↪test_weights, test_profits)
    #print("Epsilon value:  ", k)
    #print("Total profit obtained:", profit)
    #print("Weight of knapsack:", weight)
    #print("Slack in knapsack:", slack)
    #print("Optimal selection:", x)
    #print("")
    total_profits.append(profit)
    slack_evolve.append(slack)
    weight_evolve.append(weight)
    runtimes.append(runtime)

DLA_results_S = np.array((epsilon_vals, np.array(total_profits), np.
 ↪array(slack_evolve), np.array(weight_evolve), np.array(runtimes)))
```

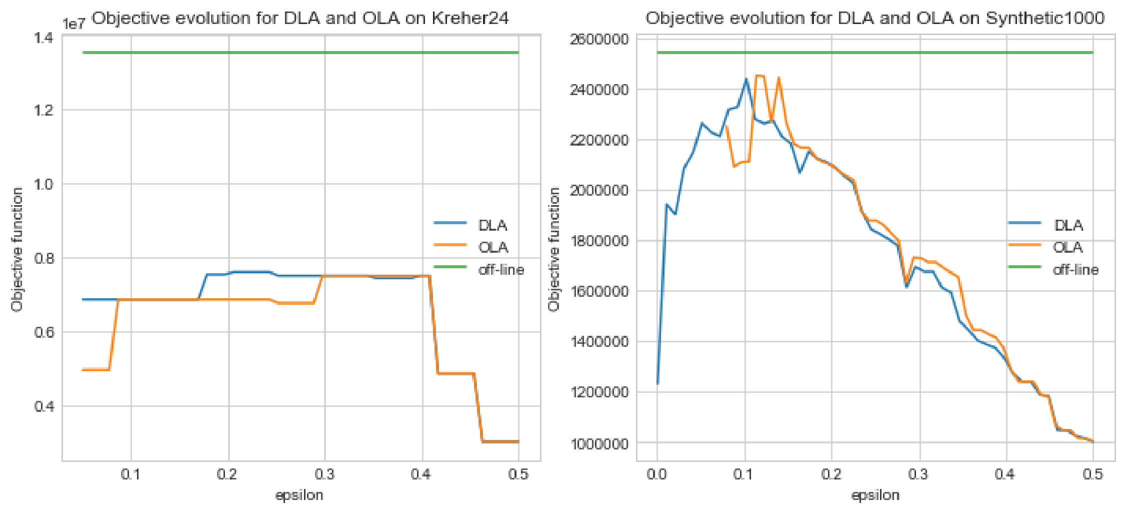
[35]:
```
from matplotlib import pyplot as plt

optimal_S = np.repeat(pr3, DLA_results.shape[1])
plt.style.use('seaborn-whitegrid')

plt.figure(figsize=[12,5])

plot1 = plt.subplot(121)
plt.plot(DLA_results_S[0], DLA_results_S[1], label = 'DLA')
plt.plot(OLA_results_S[0], OLA_results_S[1], label = 'OLA')
plt.plot(DLA_results_S[0], optimal_S, label = 'off-line')
plt.xlabel("epsilon")
```

```python
plt.ylabel("Objective function")
plt.legend(loc="upper right")
plot1.set_title("Objective evolution for DLA and OLA")

plot2 = plt.subplot(122)
plt.plot(DLA_results_S[0], DLA_results_S[2], label = 'DLA')
plt.plot(OLA_results_S[0], OLA_results_S[2], label = 'OLA')
plt.xlabel("epsilon")
plt.ylabel("Slack to full capacity")
plt.legend(loc="center right")
plot2.set_title("Slack evolution for DLA and OLA")

plt.savefig('Problem 3 epsilon.png', dpi = 800)

plt.show()
```



[36]:
```python
from matplotlib import pyplot as plt

optimal_S = np.repeat(pr3, DLA_results.shape[1])
plt.style.use('seaborn-whitegrid')

plt.figure(figsize=[12,5])

plot1 = plt.subplot(121)
plt.plot(DLA_results[0], DLA_results[1], label = 'DLA')
plt.plot(OLA_results[0], OLA_results[1], label = 'OLA')
plt.plot(DLA_results[0], optimal, label = 'off-line')
plt.xlabel("epsilon")
plt.ylabel("Objective function")
plt.legend(loc="center right")
```

```
plot1.set_title("Objective evolution for DLA and OLA on Kreher24")

plot2 = plt.subplot(122)
plt.plot(DLA_results_S[0], DLA_results_S[1], label = 'DLA')
plt.plot(OLA_results_S[0], OLA_results_S[1], label = 'OLA')
plt.plot(DLA_results_S[0], optimal_S, label = 'off-line')
plt.xlabel("epsilon")
plt.ylabel("Objective function")
plt.legend(loc="center right")
plot2.set_title("Objective evolution for DLA and OLA on Synthetic1000")

plt.savefig('Comparison epsilon.png', dpi = 800)

plt.show()
```



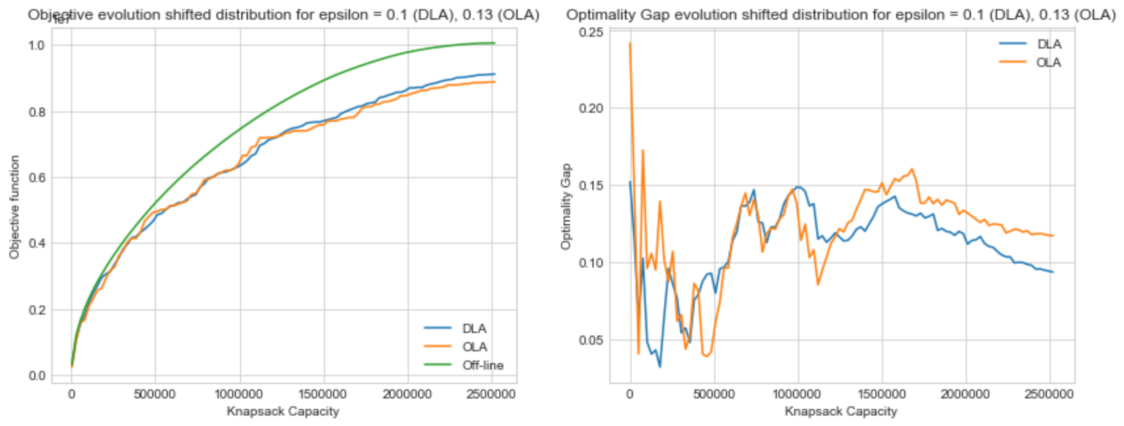
[37]:
```
### EXPERIMENTS ON CAPACITY BOUND OF RHS
```

[38]:
```
capacities = np.linspace(np.floor(np.mean(test_weights)), np.floor(np.
 ↪mean(test_weights)) * 1000, 100)
```

[39]:
```
# now run on OLA
total_profits = []
slack_evolve = []
weight_evolve = []
offline_optimal = []
runtimes = []
epsilon = 0.13

for k in capacities:
    pr, sol = knapsack(k, test_weights, test_profits)
    offline_optimal.append(pr)
    profit, weight, slack, x, runtime = OLA_knapsack(epsilon, k, test_weights,␣
 ↪test_profits)
    #print("Knapsack capacity: ", k)
    #print("Total profit obtained:", profit)
    #print("Weight of knapsack:", weight)
    #print("Slack in knapsack:", slack)
    #print("")
    #print("Optimal selection:", x)
    total_profits.append(profit)
    slack_evolve.append(slack)
    weight_evolve.append(weight)
    runtimes.append(runtime)

OLA_results = np.array((capacities, np.array(offline_optimal), np.
 ↪array(total_profits), np.array(slack_evolve), np.array(weight_evolve), np.
 ↪array(runtimes)))
```

[40]:
```
# now run on DLA
total_profits = []
slack_evolve = []
weight_evolve = []
offline_optimal = []
runtimes = []
epsilon = 0.1

for k in capacities:
    pr, sol = knapsack(k, test_weights, test_profits)
    offline_optimal.append(pr)
    profit, weight, slack, x, runtime = DLA_knapsack(epsilon, k, test_weights,␣
 ↪test_profits)
    #print("Knapsack capacity: ", k)
    #print("Total profit obtained:", profit)
    #print("Weight of knapsack:", weight)
    #print("Slack in knapsack:", slack)
    #print("")
    #print("Optimal selection:", x)
    total_profits.append(profit)
    slack_evolve.append(slack)
    weight_evolve.append(weight)
    runtimes.append(runtime)

DLA_results = np.array((capacities, np.array(offline_optimal), np.
 ↪array(total_profits), np.array(slack_evolve), np.array(weight_evolve), np.
 ↪array(runtimes)))
```

```
[41]: plt.figure(figsize=[14,5])
      plt.style.use('seaborn-whitegrid')

      plot1 = plt.subplot(121)
      plt.plot(DLA_results[0], DLA_results[2], label = 'DLA')
      plt.plot(DLA_results[0], OLA_results[2], label = 'OLA')
      plt.plot(DLA_results[0], DLA_results[1], label = 'Off-line')
      plt.xlabel("Knapsack Capacity")
      plt.ylabel("Objective function")
      plt.legend(loc="lower right")
      plot1.set_title("Objective evolution shifted distribution for epsilon = 0.1␣
       ↪(DLA), 0.13 (OLA)")

      plot2 = plt.subplot(122)
      plt.plot(DLA_results[0], 1-(DLA_results[2]/DLA_results[1]), label = 'DLA')
      plt.plot(DLA_results[0], 1-(OLA_results[2]/OLA_results[1]), label = 'OLA')
      plt.xlabel("Knapsack Capacity")
      plt.ylabel("Optimality Gap")
      plt.legend(loc="upper right")
      plot2.set_title("Optimality Gap evolution shifted distribution for epsilon = 0.1␣
       ↪(DLA), 0.13 (OLA)")

      #plt.savefig('Problem 3 RHS_diff_epsilon.png', dpi = 800)

      plt.show()
```



```
[ ]:
```