Analysis for Different Methods of Updating Upper Bound and Lagrangian Multiplier Based on UFLP

ISE 418: Integer Optimization Final Project

Tao Li and Minhan Li

Department of Industrial and Systems Engineering
Lehigh University
Spring 2017

Contents

List of Figures

Chapter 1

Introduction

In lecture 15, we discussed about the Lagrangian relaxation and we showed the textbook sub-gradient method to update the Lagrangian multiplier. The convergence rate of Lagrangian multiplier influence the solution time for MILP problem decomposition bound a lot. In this project we will discuss the methods for updating Lagrangian multiplier and try to improve the updating process under the frame of sub-gradient method.

For the testing problem, we choose the classical UFLP(Uncapacitated Facility Location Problem) model. Since this is NP hard problem and the gap of Lagrangian relaxation and the optimal solution is relatively small in most cases in this problem. We use Gurobi to solve for the optimal solution and we also code and test a heuristic method and compare it with Lagrangian bound.

Our code is in Matlab and we use a MILP class package for modelling. We test for iterations, solution time and the relative gap for each updating algorithm and compare with our improved method. We will introduce the model for UFLP, the algorithm used for updating multiplier, our improvement on sub-gradient method and the Heuristic method in the following chapter.

Chapter 2

Algorithm Descriptions

In this chapter we will discuss the algorithms that we have implemented, the UFLP model we test on and the improvement we make.

2.1 UFLP Model Formulation and Lagrangian Relaxation

The basic idea of UFLP is to minimize the total cost, including the fixed cost for opening a facility and the transportation cost. The UFLP problem can be solved by Lagrangian relaxation quite efficiently. We use the following notations:
Parameters:
I set of retailers, indexed by i,
J set of candidate DC sites, indexed by j,
f_j fixed (daily) cost of locating a DC at candidate site j, for each $j \in J$,
d_{ij} cost per unit to ship between retailer i and candidate DC site j, for each $i \in I$ and $j \in J$.
Decision Variables:
x_j decide whether a candidate facility j is chosen or not.
y_{ij} decide whether the demand at customer i is assigned by the facility j.
Then the model is formulated as follows.

$$\begin{aligned} \text{Minimize} \quad & \sum_{j\in J}\{f_j x_j + \sum_{i\in I} d_{ij} y_{ij}\} \\ \text{subject to} \quad & \sum_{j\in J} y_{ij} = 1 \quad \forall i \in I \\ & y_{ij} \le x_j \quad \forall i \in I, \forall j \in J \\ & x_j \in \{0,1\}, \quad \forall j \in J \\ & y_{ij} \in \{0,1\}, \quad \forall i \in I, \forall j \in J. \end{aligned}$$

we solve the UFLP by relaxing the assignment constraints equation $\sum_{j\in J} y_{ij} = 1$ for each

$i \in I$ to obtain the following Lagrangian Relaxation problem:

$$\begin{aligned} \max_{\lambda} \min_{x,y} \quad & \sum_{j \in J} \{f_j x_j + \sum_{i \in I} d_{ij} y_{ij}\} + \sum_{i \in I} \lambda_i (1 - \sum_{j \in J} y_{ij}) \\ \text{subject to } & y_{ij} \leq x_j \quad \forall i \in I, \forall j \in J \\ & x_j \in \{0, 1\}, \quad \forall j \in J \\ & y_{ij} \in \{0, 1\}, \quad \forall i \in I, \forall j \in J. \end{aligned}$$

For each given vector λ, we can get a corresponding lower bound and we want to get an optimal λ so that we can have the best bound generated by Lagrangian relaxation.

2.2 Generating Feasible Solution to Get Upper Bound

Now that we have obtained a lower bound from the Lagrangian sub-problem with a given λ, we need to get an upper bound by finding a feasible solution for the original problem from the Lagrangian relaxation optimal solution which is usually not feasible for the original problem.

In our test, we will consider two situations. If in the optimal solution for Lagrangian, none of the facilities is open, then we choose and open one facility by greedy method that minimize the cost. Else, We just open the facilities that are open in the solution to Lagrangian Relaxation and then assign each customer to its nearest open facility. The resulting solution is feasible and provides a best upper bound with fixed open facilities. It's common that the upper bound won't improve in each iteration, thus we will keep the best bound we ever find as the global upper bound we use for next iteration.

2.2.1 Improvement on Getting Better Upper Bound

We can also apply heuristic method to improve the upper bound. Swapping or neighbourhood searching algorithms can be applied to each feasible solution found, but this is may take some time to do the improvement. Since the updating of the Lagrangian multiplier is highly related on the upper bound, a good upper bound may result in less iteration to final optimal λ. Thus applying heuristic method to get better upper bound has both pros and cons. To avoid the additional cost for solving heuristic problem based on the same solution in different iterations, we improve this by keeping a set of used sub-problem solution. Then we can skip the updating of upper bound if we meet the same solution. We did experiment with customer from size 5 to 200. For this size of problem, in most cases, applying the heuristic method would cost us more time. However there do exists special cases that the performance with an improved upper bound is much better.

2.3 General Method for Updating the Multipliers

In each iteration, λ gives a corresponding lower bound and we can keep or update the upper bound. It is impractical to try every possible value of λ_i to find the optimal value, we want to update λ_i cleverly.

If λ_i is too small, there is no real incentive to set the y_{ij} to 1 since the penalty will be small. On the other hand, if λ_i is too large, there will be an incentive to set lots of y_{ij} to 1, making the term inside the parentheses negative and the overall penalty large and negative. By changing λ_i,

we'll encourage the sum of y_{ij} variables over j to get close to be 1. So:

$$\text{If } \sum_{j \in J} y_{ij} = 0, \text{ then lambda is too small; it should be increased.}$$

$$\text{If } \sum_{j \in J} y_{ij} > 1, \text{ then lambda is too large; it should be decreased.}$$

$$\text{If } \sum_{j \in J} y_{ij} = 1, \text{ then lambda is just right; it should not be changed.}$$

There are several ways to make these adjustments to . For UFLP, the step size at iteration t is given by:

$$\Delta^t = \frac{\alpha^t (UB - L^t)}{\sum_{i \in I} (1 - \sum_{j \in J} y_{ij})^2} \tag{2.1}$$

Where L^t is the lower bound found at iteration t and UB is the best upper bound we ever found. The step direction for iteration is simply given by:

$$1 - \sum_{j \in J} y_{ij} \tag{2.2}$$

So the computer update rule can be given by:

$$\lambda_i^{t+1} = \lambda_i^t + \Delta^t (1 - \sum_{j \in J} y_{ij}) \tag{2.3}$$

2.4 More Updating Rules

The updating rule is in fact very similar to that of gradient descent, if you treat Δ_i as the learning rate. However, one problem with gradient descent is that it is very easy to be trapped in the local minimum or saddle point. Currently, gradient descent is widely used in updating the parameters of the deep learning neural network during and there are several improved gradient descent methods being proposed to avoid trapping and accelerate the updating procedure. Some of them already showed great performance in searching of global or local but good enough minimum. So we incorporate those improved algorithms to the updating rule.

2.4.1 Momentum

The updating rule is given by:

$$v_{t+1} = \rho * \nabla f(\lambda_i^{t-1}) + f(\lambda_i^t) \tag{2.4}$$

$$\lambda_i^{t+1} = \lambda_i^t + \alpha * v_{t+1} \tag{2.5}$$

$Where,$

ρ *is the friction rate* (*set to be* 0.9 *by default*);

α *is the learning rate.*

In Momentum, the new path is the sum of the current path and previous path during each iteration.

Vanilla SGD has the problem that it will oscillate across the slopes of the ravine, while momentum helps accelerate SGD in the relevant direction and dampens oscillations

2.4.2 Nesterov Momentum

The updating rule is given by:

$$v_{t+1} = \rho v_t - \alpha \nabla f(\lambda_t + \rho v_t) \tag{2.6}$$
$$\lambda_i^{t+1} = \lambda_i^t + v_{t+1} \tag{2.7}$$

The intuition is similar to Momentum, but Nesterov Momentum calculats the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters

2.4.3 AdaGrad

The updating rule is given by:

$$grad_{squared} = grad_{squared} + \nabla f(\lambda_i^t)^2 \tag{2.8}$$
$$\lambda_i^{t+1} = \lambda_i^t + \alpha * \frac{\nabla f(\lambda_i^t)}{grad_{squared}^{0.5} + 10^{-7}} \tag{2.9}$$

Where,

$grad_{squared}$ *is the sum of all the previous gradient.*

It adapts its learning rate to each parameter, which performs larger updates for infrequent and smaller updates for frequent parameters. It is especially well-suited for sparse dataset.

2.4.4 RMSProp

The updating rule is given by:

$$grad_{squared} = decay_{rate} * grad_{squared} + (1 - decay_{rate}) * \nabla f(\lambda_i^t)^2 \tag{2.10}$$
$$\lambda_i^{t+1} = \lambda_i^t + \alpha * \frac{\nabla f(\lambda_i^t)}{grad_{squared}^{0.5} + 10^{-7}} \tag{2.11}$$

where,

$grad_{squared}$ *is set to be* 0 *at the first iteration by default;*
$decay_{rate}$ *is usually set to be* 0.9.

RMSProp is adaptive learning rate method and works well in on-line and non-stationary settings

2.4.5 Adam

The updating rule is given by:

$$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * \nabla f(\lambda_i^t) \tag{2.12}$$
$$v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * \nabla f(\lambda_i^t) * \nabla f(\lambda_i^t) \tag{2.13}$$
$$\lambda_i^{t+1} = \lambda_i^t + \frac{m_t}{\sqrt{v_t} + 10^{-7}} \tag{2.14}$$

Where m_t and v_t are set to be 0 at the first iteration by default.
β_1 and $beta_2$ are set to be 0.9 and 0.999 by default.

Adam computes adaptive learning rates for each parameter and it stores an exponentially decaying average of past squared gradients and an exponentially decaying average of past gradients, which RMSprop doesn't do.
Adam ensures that the magnitudes of parameter updates are invariant to rescaling of the gradient and a stationary objective is not required. Like Adagrad, it works well with sparse gradients, and it naturally performs a form of step size annealing.

Chapter 3

Numerical Results

We implement the original sub-gradients method and other 5 methods to update the Lagrangian multiplier on MATLAB and we also test the upper bound heuristic updating technology influence. The test data is from on line data facility location problem and we also generated some artificial data to test for different problem size. The artificial data is generated by excel from uniform distribution and we don't include them in the code file. For different size comparison, we test for all six algorithms as well as the optional upper bound heuristic method in problem scale from 5 to 200. The problem scale is the number of customers in the model. We used performance profile to show the result, but the result from it is too extreme and there is not much information from it so we decide not include in and compare the algorithm with the realistic data which can show more information and show the trend.

Figure 1 to 12 show the results for bound updating with iteration for each algorithm for two instance. From the results we can see that, at for this problem, sub-gradients shows the best performance. So we also compare it with Gurobi and the heuristic method result.

Table 3.1: Scale of problem

Problem ID	No. of EFs	No. of NFs
1	880	922
2	8	5

Table 3.2: Results compare

Methods	Total Time	Gap
Heuristic	7.04s	0.30%
Gurobi	176s	0.00%
Lagrangian	65s	1.86%

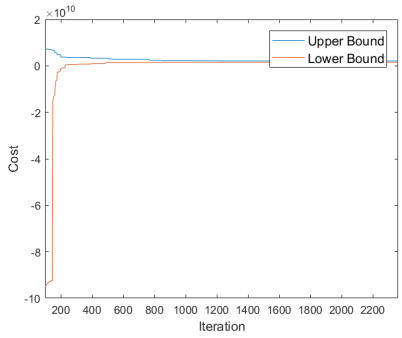

Figure 3.1: Lower and upper bound using sub-gradient

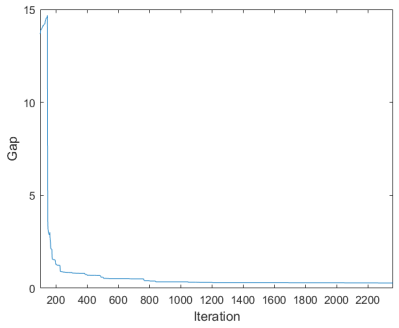

Figure 3.2: Gap using sub-gradient

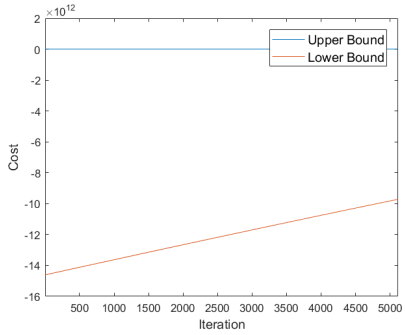

Figure 3.3: Lower and upper bound using Momentum

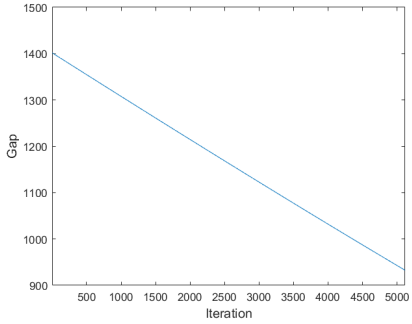

Figure 3.4: Gap using Momentum

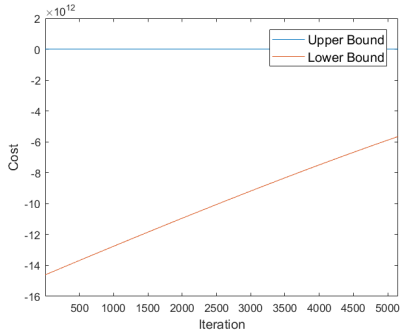

Figure 3.5: Lower and upper bound using Nesterov Momentum

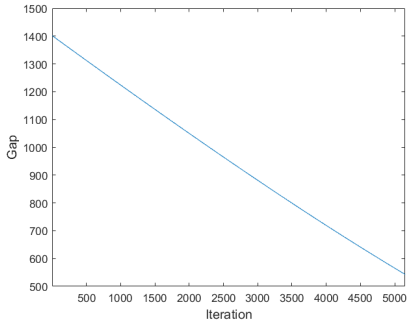

Figure 3.6: Gap using Nesterov Momentum

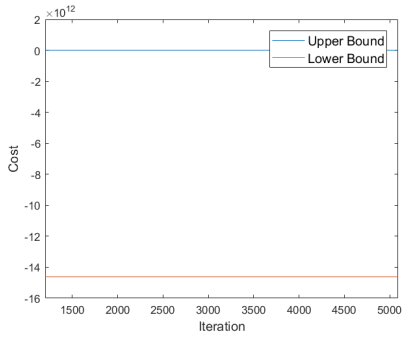

Figure 3.7: Lower and upper bound using AdaGrad

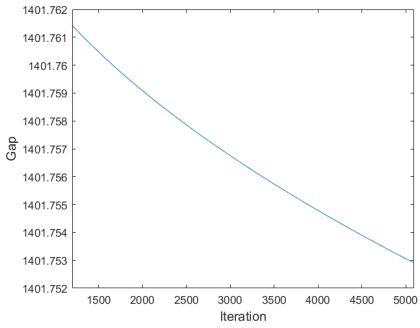

Figure 3.8: Gap using AdaGrad

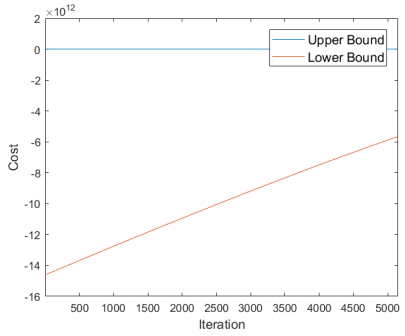

Figure 3.9: Lower and upper bound using RMSProp

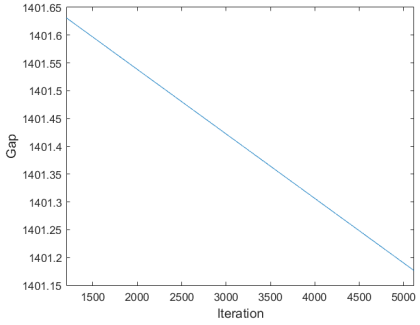

Figure 3.10: Gap using RMSProp

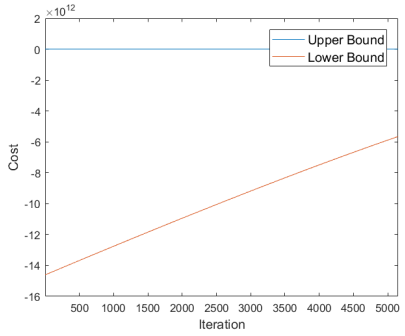

Figure 3.11: Lower and upper bound using Adam

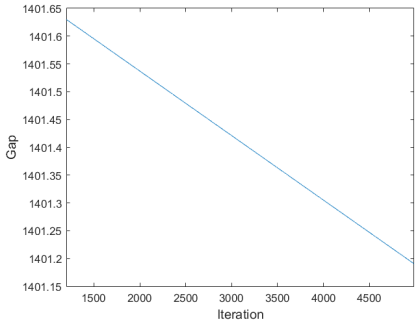

Figure 3.12: Gap using Adam

Chapter 4

Conclusion

In this paper, we discussed two parts of improvement in solving UFLP.

First we consider using heuristic method during the process of finding an feasible solution based on the optimal solution for Lagrangian sub-problem. To avoid the additional cost for solving heuristic problem based on the same solution in different iterations, we improve this by keeping a set of used sub-problem solution. Then we can skip the updating of upper bound if we meet the same solution. The improvement is not as efficient as we thought for UFLP. In most cases, the simple way of updating upper bound solves the problem faster. However, in some special instance, this method can get a good upper bound during the several start rounds performance much better than the simple way.

As for updating the Lagrangian multipliers, we implied 5 improved gradient descent methods and test them on a relatively large-scale UFLP model. The algorithms show pretty good performance in minimizing the gap, although still not as good as Gurobi.

In terms of future work, we can incorporate different methods together to explore a better one. For instance we can solve several iterations of Lagrangian relaxation and get the upper bound with our heuristic method as the upper bound. Then we can use the branch and cut method to solve for optimal. This is more like a pre-solve process. The limitation for this is the heuristic method is too specific for model and could not apply to general cases.