EE 5239 2018 Fall
Introduction to Nonlinear Optimization
Final Project Report
Santa Gift Matching Challenge

Tiancong Chen Chunni Zhao

December 20, 2018

1 Introduction

When it comes to the end of the year, one of the biggest holiday, Christmas, is around the corner. It's the time that Santa starts to worry about the gift matching again. The dilemma is: each child has his/her own preference about the ideal gift, but at the same time, each gift also has a list that contains the favorable kids it should be assigned to. It is always a tough question that makes Santa's life harder during the Christmas time. This year, Santa needs those smart Optimization guys to come up with a perfect gift matching plan in order to make both Santa and kid happy, which means Santa hopes to drop gifts to those children who are on the wish list of gift. Meanwhile, the gifts given to kids are also supposed to be the ones the kids have been desiring. By maximizing the happiness, one toy matching algorithm is supposed to solve Santa's problem to pair kids with gifts they want.

1.1 Data Description

This dataset is taken from the "Santa Gift Matching Challenge" on Kaggle. For this challenge, our goal is to build an algorithm that maximizes the total happiness of both Santa and children by paring children with gifts they want. This includes two parts of measurement of happiness: one is to please Santa by assigning gifts to kids that gifts want to. For kids, the higher the gift is on their wish list , the happier they are. The other one is to make children satisfied by giving them the gifts they like. As for Santa and his gifts, the higher the child is on the good kids list, the happier Santa is.

1.1.1 File description

The dataset that this Kaggle competition provides contains two different sub-datasets. One is called "*child_wishlist*", in which we have a list of one million unique children ID and their wish lists of 100 gifts.

While in this dataset, there are a few details need us to notice: First, the first 0.5% (ChildId 0-5000) children are triplets. More particularly, 0,1,2 are triplets, 3,4,5 are triplets $\cdots$ 4998,4999,5000 are triplets. Triplets need to be given the same gift even though they might have different preferences. Second, the next 4% (ChildId 5001-45000) children are twins. More particularly, 5001 and 5002 are twins, 5003 and 5004 are twins, $\cdots$ 44999 and 45000 are twins. Twins need to be given the same gift even though they might have different preferences.

The other is known as "$gift\_goodkids$", in which we are given a list of one thousand gifts and their list of one thousand good kids that they prefer to give to.

In this dataset, the only constraint is that for each GiftId, there are 1000 of them available. There are exactly the same number of gifts available (1000 * 1000 = 1,000,000). We shall not exceed the quanlity of 1000 for each GiftId.

1.1.2 Evaluation

Our goal is to maximize:

$$\begin{gathered} AverageNormalizedHappiness \text{ (ANH)} = \\ (Average\_Normalized\_Child\_Happiness \text{ (ANCH)})^3 + \\ (Average\_Normalized\_Santa\_Happiness \text{ (ANSH)})^3 \end{gathered}$$

where $Normalized\_Child\_Happiness$ is the happiness of each child, divided by the maximum possible happiness, and $Normalized\_Santa\_Happiness$ is the happiness of each gift, divided by the maximum possible happiness.

In the equation form:

$$ANCH = \frac{1}{n_c} \sum_{i=0}^{n_c - 1} \frac{ChildHappiness}{MaxChildHappiness}$$

$$ANSH = \frac{1}{n_g} \sum_{i=0}^{n_g - 1} \frac{GiftHappiness}{MaxGiftHappiness}$$

n_c is the number of children.
n_g is the number of gifts.
$MaxChildHappiness$ = len($ChildWishList$) * 2 = 100 * 2
$MaxGiftHappiness$ = len($GiftGoodKidsList$) * 2 = 1000 * 2
$ChildHappiness$ = 2 * $GiftOrder$ (if the gift is found in the wish list of the child)
$ChildHappiness$ = -1 (if the gift is out of the child's wish list)
$GiftHappiness$ = 2 * $ChildOrder$ (if the child is found in the good kids list of the gift)
$GiftHappiness$ = -1 (if the child is out of the gift's kids list)
For example, if a child has a preference of gifts [5, 2, 3, 1, 4], and is given $gift_3$, then $ChildHappiness$ = [len($Wishlist$) - indexOf($gift_3$)] * 2 = [5 - 2] * 2 = 6

1.1.3 Data Visualization

To get a basic idea of how our dataset looks like, we explored data visualization by plotting several graphs to show the information about how they get preferred by each other. It's easy to see that there are 1667 groups of triplets in our dataset, and their preferences are displayed in Figure 1.

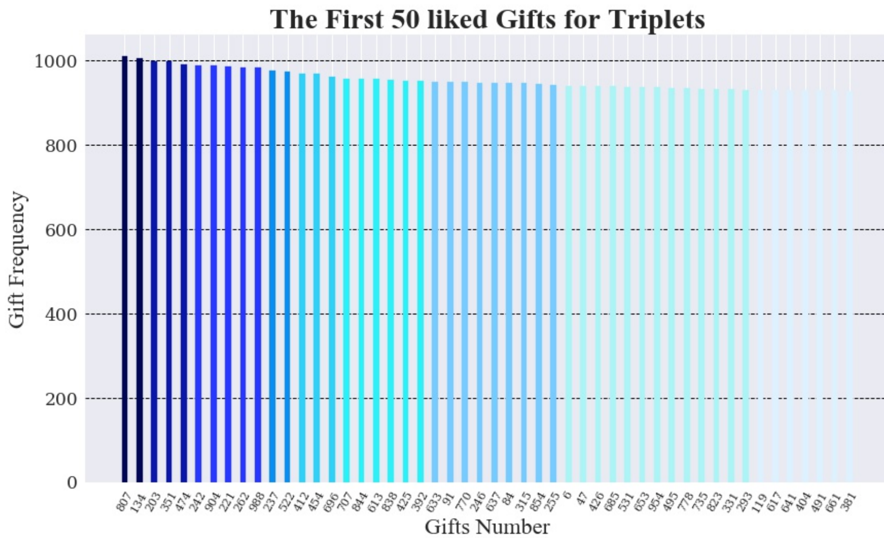

Figure 1: The Frist 50 Gifts Liked by Triplets

What's more, the number of twin couples is 20000 in total in our dataset, and their tastes of gifts choosing is shown in Figure 2.

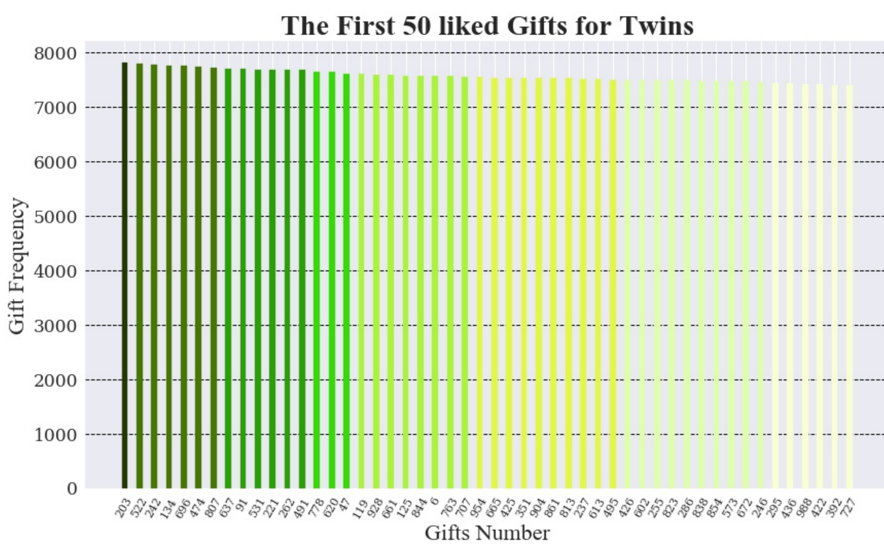

Figure 2: The First 50 Gifts Liked by Twins

As for the distribution for the whole dataset, the first 50 liked gifts is shown in Figure 3.

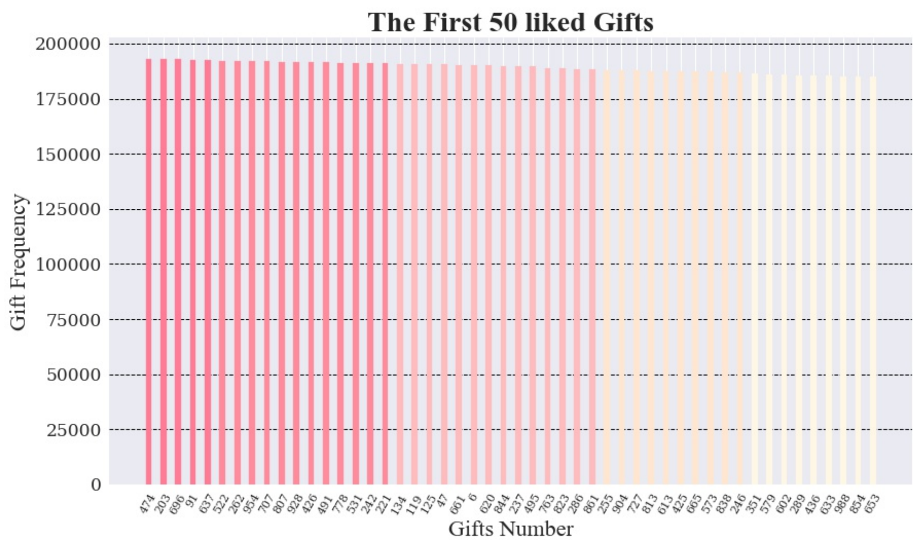

Figure 3: The First 50 Gifts Liked by Children

1.1.4 Feature Engineering

To obtain the latent information of the measurement of the happiness of both Santa and children, we compute ChildHappiness given any gift and GiftHappiness given any child as well, which also is the edge weight in our algorithm shown in the following section.
Listing 1 is how we construct the measurement of child happiness: we first treat the triplets as three kids with the same preference for gifts although most of them have different interests. Then we also relax the twins constraints, we treat the twins as two same kids with same taste in gift choosing. However, the error due to this relaxation could be negligible in the end. From the final result we obtained, for each present, except for some edge cases, there are at most one twin and one triplet which don't follow the twins/triplets constraints. What's more, considering the limited computing resources, we did some approximation for the real happiness for each subgroup of children. What we chose here is 60 for the sake of computing limit.

Listing 1: ChildHappiness

```
# Hapiness of Child
h4c = dict()
n = child.shape[1]
limt = 60
# for triplets
for i in range(0, 5001):
    node = i - (i % 3)
    for j in range(limt):
        if (node, child[i][j]) in h4c:
            h4c[(node, child[i][j])] += 10*(1+(n-j)*2)
        else:
            h4c[(node, child[i][j])] = 10*(1+(n-j)*2)
# for twins
for i in range(5001, 45001):
    node = i + (i % 2)
    for j in range(limt):
        if (node, child[i][j]) in h4c:
            h4c[(node, child[i][j])] += 10*(1+(n-j)*2)
        else:
            h4c[(node, child[i][j])] = 10*(1+(n-j)*2)
# for single
for i in range(45001, 1000000):
    for j in range(limt):
        h4c[(i, child[i][j])] = 10*(1+(n-j)*2)
```

As for the GiftHappiness, here is how it works:

Listing 2: GiftHappiness

```
# Happiness of Santa
h4g = dict()
for i in range(gift.shape[0]):
  for j in range(gift.shape[1]):
    cur_child = gift[i][j]
    # triplets
    if cur_child < 5001:
      cur_child -= cur_child % 3
    # twins
    elif cur_child < 45001:
      cur_child += cur_child % 2
```

```
    h4g[(cur_child, i)] = (1+(gift.shape[1]-j)*2)
```

To make it more efficient, we only take the cases that both children and gift are interested to each other, and just discard the cases that only one of them is on the wish list of others, which is named as "positive_cases" in our codes. We also did a linear approximation of the original nonlinear object function, which is $(ANCH))^3 + (ANSH))^3$, but we'll explain the reason in the Algorithm section.

Listing 3: TotalHappiness

```
# for cutting some edges,
# if they are not liked by each other.
  positive_cases = list(set(h4c.keys())|set(h4g.keys()))
  # final happiness dictionary
  h = dict()
  for p in positive_cases:
      h[p] = 0
      if p in h4c:
          a = h4c[p]
          h[p] += int((a**3)*4)
      if p in h4g:
          b = h4g[p]
          h[p] += int((b**3)/4)
```

2 Algorithm

With the relaxation we made in the previous section, we can formulate our algorithm easily. If we know how many singles, twins and triplets receive each present, the original Santa problem can be formulated as a Min-Cost Flow Problem.

2.1 Min-Cost Flow Problem

A flow network is a directed graph $G = (V, E)$ with a source vertex $s \in V$, where each edge $(u, v) \in E$ has capacity $c(u, v) > 0$, flow $f(u, v) \geq 0$ and cost $a(u, v)$, with most minimum-cost flow algorithms supporting edges with negative costs. The cost of sending this flow along an edge (u, v) is $f(u, x) \cdot a(u, v)$. The problem requires an amount of flow d to be sent from

source s to sink t.
The definition of the problem is to minimize the total cost of the flow over all edges:

$$\sum_{(u,v)\in E} a(u,v)\cdot f(u,v)$$

with the constraints

Capacity constraints: $f(u,v) \le c(u,v)$
Skew symmetry: $f(u,v) = -f(v,u)$
Flow conservation: $\sum_{w\in V} f(u,w) = 0$ for all $u \ne s,t$
Required flow: $\sum_{w\in V} f(s,w) = d$ and $\sum_{w\in V} f(w,t) = d$

The minimum cost flow problem can be solved by linear programming, since we optimize a linear function, and all constraints are linear. Therefore, we need to make our object function in a somehow linear form.
What we did in our final model is to use $\frac{1}{n_c}\sum_{i=0}^{n_c-1}(\frac{ChildHappiness}{MaxChildHappiness})^3$ to replace $(\frac{1}{n_c}\sum_{i=0}^{n_c-1}\frac{ChildHappiness}{MaxChildHappiness})^3$.
Similarly, we use $\frac{1}{n_c}\sum_{i=0}^{n_c-1}(\frac{GiftHappiness}{MaxGiftHappiness})^3$ instead of $(\frac{1}{n_c}\sum_{i=0}^{n_c-1}\frac{GiftHappiness}{MaxGiftHappiness})^3$.
In application, we have to drop some edges between gifts and children due to limited computational resources. One natural idea is dropping all edges between gifts and children that gifts not interested in children or children not interested in gifts. But it's still out of resource to solve this simplified version. Then we cut the edges whose ChildHappiness that less than 2×50. Then the algorithm becomes runnable. However, this also raises another problem: term $\frac{GiftHappiness}{MaxGiftHappiness}$ range in $[1/1000, 2/1000, \cdots, 1000/1000]$, and $\frac{ChildHappiness}{MaxGiftHappiness}$ range in $[51/100, 52/100, \cdots, 100/100]$, this implies we need a scale parameters for these two linearizations.

2.2 Minimum Weight Bipartite Matching

For our specific case, we use one application of min-cost flow problem, called minimum weight bipartite matching.
Given a bipartite graph $G = (A\cup B, E)$, the goal is to find the maximum cardinality matching in G that has minimum cost. Let $w : E \to R$ be a weight function on the edges of E. The minimum weight bipartite matching

problem or assignment problem is to find a perfect matching $M \subseteq E$ whose total weight is minimized. The idea is to reduce this problem to a network flow problem.
Let $G' = (V' = A \cup B, E' = E)$. Assign the capacity of all the edges in E' to 1. Add a source vertex s and connect it to all the vertices in A' and add a sink vertex t and connect all vertices inside group B' to this vertex. The capacity of all the new edges is 1 and their cost is 0. It is proved that there is minimum weight perfect bipartite matching in G if and only if there is a minimum cost flow in G'.

2.3 Implementation

2.3.1 Ortools

To solve the algorithm we applied to this case, we use this solver called "Ortools". To call this method, we need to construct specific inputs corresponding to the method inside this solver.
Following is the description of the min cost flow solver in ortools:
The min cost flow problem has special nodes, called supply nodes or demand nodes, which are similar to the source and sink in the max flow problem. Material is transported from supply nodes to demand nodes.
At a supply node, a positive amount the supply is added to the flow. A supply could represent production at that node, for example. At a demand node, a negative amount the demand is taken away from the flow. A demand could represent consumption at that node, for example. For convenience, we'll assume that all nodes, other than supply or demand nodes, have zero supply (and demand).
For the min cost flow problem, we have the following flow conservation rule, which takes the supplies and demands into account:
At each node, the total flow leading out of the node minus the total flow leading in to the node equals the supply (or demand) at that node.
The graph below shows a min cost flow problem. The arcs are labeled with pairs of numbers: the first number is the capacity and the second number is the cost. The numbers in parentheses next to the nodes represent supplies or demands. Node 0 is a supply node with supply 20, while nodes 3 and 4 are demand nodes, with demands -5 and -15, respectively.

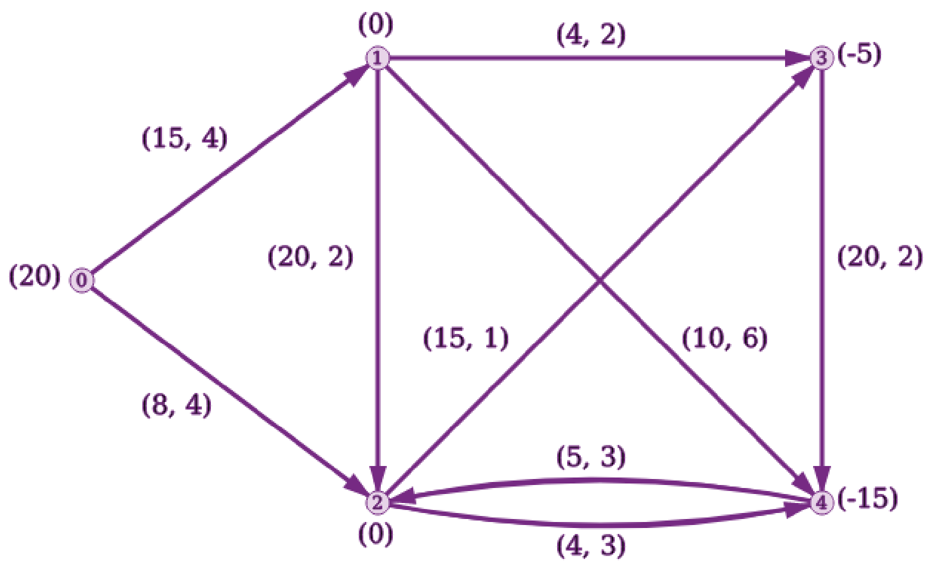

Figure 4: Example

The following code defines the data for the problem. In this case, there are four arrays for the start nodes, end nodes, capacities, and unit costs. Again, the length of the arrays is the number of arcs in the graph.

Listing 4: Sample Definition

```
# Define four parallel arrays: start_nodes, end_nodes,
# capacities, and unit costs between each pair.
# For instance, the arc from node 0 to node 1 has a
# capacity of 15 and a unit cost of 4.

start_nodes = [ 0, 0,  1, 1,  1,  2, 2,  3, 4]
end_nodes   = [ 1, 2,  2, 3,  4,  3, 4,  4, 2]
capacities  = [15, 8, 20, 4, 10, 15, 4, 20, 5]
unit_costs  = [ 4, 4,  2, 2,  6,  1, 3,  2, 3]

# Define an array of supplies at each node.

supplies = [20, 0, 0, -5, -15]
```

For our problem, we assign each gift nodes with supply 1000, child nodes with supplies -1,-2, or -3, depending on they are single, double or triplets, capacity of each edges is 1, 2, or 3, corresponding cost is -ChildHappiness -GiftHappiness. Then we can apply this tool to our problem.

2.3.2 Parameter Tuning

Among all the methods contained inside the solver, wo chose "SolveMaxFlowWithMinCost()". It is same as another method, "Solve()", but does not have the restriction that supply must match the demand or that the graph has enough capacity to serve all the demand or use all the supply. What it will compute is a maximum-flow with minimum cost. The main reason to use this method rather than "Solve()" is that the "Solve()" method has so strict constraints that it's hard for us to run a relaxation version of the original problem by cutting the number of gift on the each ChildWishList to a smaller one, which is a necessary compromise to the limited computing resource.
What we came up to solve this problem is to use this relaxed method "SolveMaxFlowWithMinCost()", and try to find the smallest number of gift on the each ChildWishList that makes our optimal solution feasible.

3 Results

Since what we are trying to solve is a relaxed version problem of the original one, the solution set does have some elements that do not satisfy the twins/triplets constraints. Hence, we need to recover a feasible solution. We make a heuristic exchange with a random non-twin child, if there is one couple of twins who end up with different gifts. Similar process also applied to the triplets as well. As a matter of fact, this operation could bring in some error. It turns out that there are still 2 triplets and 42 doubles assigned by different types of gifts, which is contracted to the problem settings. So we swap with single to make them satisfied the setting, and since it's 200 children compared to total 1000000 children, this doesn't affect the result enormously.
Our result is 0.9287240888.

4 Conclusion

We view the problem as a large-scale matching problem with cubic cost function, then use some approximation and simplification to convert it into a solvable min cost flow problem, and finally swap remained mismatching nodes. There are some further work that maybe can improve the result:
1. Use piecewise linear approximation instead of general linear approximation, which may improve the accuracy.
2. Use greedy algorithm to swap the mismatching nodes instead of swapping it casually.
3. View the problem as a integer programming, use MIP solver in Ortools instead of min cost flow solver to address the problem.