For office use only
T1_____
T2_____
T3_____
T4_____
For office use only

F1_____
F2_____
F3_____
F4_____

2016

19th Annual High School Mathematical Contest in Modeling (HiMCM) Summary Sheet
(Please make this the first page of your electronic Solution Paper.)

Team Control Number:6829
Problem Chosen:B

As online shopping become more and more popular in contemporary society, the increase demand for larger quantity of production as well as faster delivery time lead the recreation equipment company to build more warehouses in the U.S. Its goal is to make those new warehouses cover all the area within one-day ground shipping.

The two of the biggest problems of this question lie in the data extraction from the URLs and data analysis, especially when there are a lot of mismatched zip-codes existing on this website. It is not difficult to find that this is a typical set covering problem. So a more efficient algorithm is needed when doing analysis: approximation algorithm.

The essence of this model is derived from "greedy algorithm": instead of considering from the overall perspective, the approximation algorithm only looks for the *current* maximum cover-area *increment* (warehouse's location that can radiate the largest area). We are clearly aware that it is impossible to come up with all the answers with high precision due to the large amount of data needed to address. So sacrificing a little bit of precision of the results (as well as the number of results) to save a huge amount of computational time is worthwhile and extremely beneficial. We admit that the error is bigger than the brute force search, but the time we saved is definitely worth the precision.

One significant advantage of this model is the high efficiency. Although approximation algorithm could not compute the optimal result, its algorithm complexity can decrease from $O(n! \cdot n)$ to $O(n^2 \cdot \log_2 n)$, as the time in other algorithms take up unimaginable amounts of time.

Furthermore, we optimized the approximation algorithm while solving part Ⅱ since tax rate had been taken into consideration. The same thinking pattern was employed in part Ⅲ.

Then we tested our model by calculating the land cover rate, as shown in table 3.

Lastly, the model analysis shows that our model has its stability when the tax rate of garment varies.

The paper will illustrate the ideas and results abovementioned specifically.

A Letter to Company's President

Dear President,

I am one of your company staffs. For several years, our company had always been the leading enterprise in recreation equipment businesses. As electronic marketing gradually becomes the dominating character, the demand from customers for a faster delivery speed and a lower, reasonable price becomes higher than it had never reached. Company executives had decided to enlarge our business by establishing more warehouses in different locations in the U.S., with the hope of minimizing both goods' delivery time from the customers' perspective and reducing tax cost. It is also acknowledged that adding clothes and apparel as new product lines are aside from that equally important. After deliberate analysis, we wrote this letter to produce you our perspective of view by giving you our model and results, sincerely hoping that our work could help solve your problem.

The results are, if we do not consider tax and new clothes business, result for 100% land cover rate needs 23 warehouses as shown in table 1. But for recommendation, 96.14% land cover rate is better and only need 19 warehouses. The locations can be seen in figure 4.3.1 and table 2.

Based on the given state sales tax rates, we optimized our approximation algorithm by considering the tax rate prior to incrementing the maximum coverage. (See table 4 for details) The adjusted number of warehouses are 22, covering 95.26% of land area. Total weight of taxes is 3.36157×10^6 dollars. Also, we list the land cover rate and correspoding tax and warehouses' number in table 3.

As for the result after the introduction of clothes and apparel businesses, we find that the result would not change unless the land cover rate varys from 90.7%-91.3%. Therefore, we decide that the adjusted number and location of warehosues are the same in part I (shown in figure 3).

We have faith in our model after deliberate testing. We sincerely hope that these results can contribute to your work, and make our company gain the upper ground in the competition in the digital age.

Sincerely,

Team #6829

Contents

1 Introduction and Problem Interpretation

As online shopping become more and more prevalent and important in people's everyday life, the seek for balance between the traditional brick and mortar stores and online stores are especially important for companies. Compared with shopping in person, online shopping is restricted to the geological locations not only for by the rare opportunity to actually experience the products, but also by the long time required to wait in order to receive the items from the limited number of warehouse(s).

Therefore, viewing from a long-term developing perspective, the solution that we are seeking is to look for appropriate sites for those warehouses in order to deliver goods within one-day ground shipping, so that companies can easily expand their business with agility.

In this question, we are ordered to solve this location choosing problem. After doing several transformations to this practical problem, we attributed it as a *set cover problem*.

- Shipping time and land-cover efficiency should be considered as important factors. We need to implement the utmost utilization of the resources given by the "Ground Transit Times Shipping From One Place to Another" maps.
- Meanwhile, as taxes are taken into consideration, our model optimization should estimate and compute each state's tax rate after selecting the minimum warehouse numbers and their corresponding locations.
- Following the same algorithmetic pattern, in order to add clothing and apparel to our line of product, we should analyze the effect of the respective taxes in the same way.

In the end, we finished our paper by writing a letter to our company's president including our results and conclusion, while also advising him or her on the choice of the warehouses' locations, how our model is best fitted in minimizing the tax liability and therefore benefits the whole business and the customers.

2 Simplifying Assumptions

2.1 Assumptions and Justifications

1. **Exceptions**

   Assumption: Inevitable factors and accidents do not exist in this model, which means delivery time can be accurately predicted by the "Ground Shipping" maps provided by UPS.

   Justification: First, inevitable factors such as rain, hurricane, earthquake and other forms of non-human activities, and events in emergence cannot be predicted. Secondly,

these small-probability events can be ignored with minor importance compared to the large scale scheme.

2. **City Difference**

Assumption: Each city is considered the same without functional difference.

Justification: We are given no information about cities' functional difference that will influence goods' delivering when choosing warehouses. There is no efficient and rapid way, beyond our reach, to analyze the effect of the difference. Therefore we treat every point of the U.S. land as equal.

3. **Time Difference**

Assumption: Everyday in a year is considered the same.

Justification: It would be a very complicated model if we take into consideration the daily differences. Also, it is not practically necessary to build our model precisely to whichever day in any year. For instance, business busy times such as Christmas, Thanksgiving can be overlooked and considered as ordinary days.

4. **ZIP Codes Retrieval**

Assumption: On the UPS site, if different ZIP codes represent a same map URL (maps), we assume that these ZIP codes of the cities are the same.

Justification: Due to the low precision data given by UPS, we are unable to distinguish between those cities linking to the same pictures. Under this condition we assume and believe that those cities are geographically close to each other and delivery-wise functionally similar to each other which lead to the same results.

5. **Customer Coverage**

Assumption: Goods' final destinations can be everywhere in the U.S. land, except the given states as exceptions. We only analyze the land that we covered.

Justification: The shipping time given by UPS describes the time directly from the source to the destination. Therefore the exact geographic points are analyzed. Doing this could also improve the model precision by taking more *sample points*. Therefore assuming city to be the smallest unit is logical and feasible.

6. **Purchasing Abilities**

Assumption: The purchasing ability of the state is proportional to its Gross State Product.

Justification: It would be meaningless to not retrieve data in this way. Since we are unable to get any actual data about our company, we would not estimate sales through actual data. Therefore we can only assume that the state's purchasing ability is proportional to its Gross State Product, which it produces.

2.2 Definitions

1. Sample Point

A *Sample Point* is a random point taken from the U.S. map, not regarding its actual meaning, such that it may not be an actual city, town or village, but shall be related to an actual geographic location. A sample point is represented on the map as a single pixel, in the following figure 3.1.2

The sample point method is crucial in defining how the U.S. land is covered by this one-day ground shipping implementation, and therefore is an important definition is this thesis.

2. Land Cover Rate

Land Cover Rate analyzes how the land of U.S. are covered when delivering goods from warehouses that we choose. It is noteworthy that due to the given pictures' unknown error, the overlapped radiated area can only cover 95.80% land (total rate). Therefore, the land cover rate that we defined is actually the relative rate: the actual rate divided by the total rate.

The higher the land cover rate, the better and faster the packages are delivered to the customers. Please note that we do not take into account the population coverage in this particular term.

Since this could evaluate the efficiency of the warehouse arrangement, this definition is critical in testing our model's efficiency, accuracy and feasible degree.

3. Taxity

Taxity is the annual amount multiplied by the tax rate, in terms of individual states.

Taxity estimates the purchasing power of each state. Except the tax rate, all other given information are retrieved from the U.S. Census and the Bureau of Labour Statistics (BLS) of America.

It is widely acknowledged and is logical to assume that the state's GSP (Gross State Product) is proportionate to its purchasing ability, therefore tax rate multiplies GSP is feasible.

2.3 Variables and Mathematic Symbols

- **Sample (Citites/States) Count:** n

  Our samples are the cities' of America, serving as potential which are 691 in total as given by the UPS server URLs, which includes invalid cities. After excluding those invalid cities we reached an agreement that there are 245 cities that satisfies our requirement. See 3.1.1 for more details.

- **Sample Points Count:** m

  By default, $m = 3000$. These sample points are the exact sample points as defined in 1. See 3.1.2 for more details.

- **Warehouses' Number:** r

  The number of all warehouses chosen as the result of the evaluation.

- **Warehouses' Zip-codes:** Zip_code

  The locations of warehouses that we are going to choose are reflected by the Zip-codes of the cities at which located. We can find each city's zip-code from the given URL (including invalid zip-codes). By choosing cities Zip-codes rather than using coordinates, we can lower the algorithm complexity efficiently.

- **Cities/States Information:** Q

  We use a 0-1 programming model (or a 0-1 Matrix, see 4.1) to store those information. Details of its principals would be illustrated later.

- **Sample Points:** S

  This defines the set of all sample points, whereas S_j $s.t.$ $j \in [1, n]$ is the sample points reachable by any individual warehouse $\#j$.

- **Chosen Warehouses:** J

  The set of the numbers of the warehouses chosen individually. We also assert that $|J| = r$. The final results should include the number of the warehouses, and also this is an important variable in our model.

- **A parameter in part II:** γ

  This parameter is used in part II, and we determine $\gamma = 0.5$. The reason for 0.5 is illustrated in 4.3.2.

- **The weight for purchases on garment:** η_{gar}

  According to the Bureau of Labor Statistics of U.S., the approximate allocation of the clothes and apparel in the overall consumption is about 3.3%. Therefore, we can assume that our company's clothing sales amount in each family is proportionate to that of the domestic, and use it to compute how its varation will affect the result.

- **Tax Rate (function):** T

  1. T is the tax rate of ordinary products.
  2. T_gar is the tax rate of garments.
  3. T_2 is the equivalent tax rate for problems in Part 3 (See 4.3.3), where:

$$T_2 = T \cdot (1 - \eta_{gar}) + T_{gar} \cdot \eta_{gar}$$

Note that the data of T and T_{gar} can be retrieved from the internet.

- **Affectiveness Weight:** ψ

  ψ_j $s.t.$ $j \in [1, n]$ infers the affective weight of the warehouse $\#j$, which defines how important a warehouse is. Normally in the following models it equals to the tax weight of the state which the warehouse is located in.

3 Model Description

This question is in fact a combination of a number of small problems, which can be solved one by one easier than solving as a whole. Aspects of mathematics and computer science involved in the question includes data retrieval, web crawlers, image processing, algorithms and data structures.

We shall retrieve the data from the references we had stated and the internet as is, and process the data into formats that we and our programs could understand. These formats are used to create results by the data processing programs, whose duties are resolving the results.

All data extraction utilities are used and functional, however only one of the three algorithms is correct and fully functional, id est the third one.

The model description in detail can be divided into two important sections, shown as the figure 1 below.

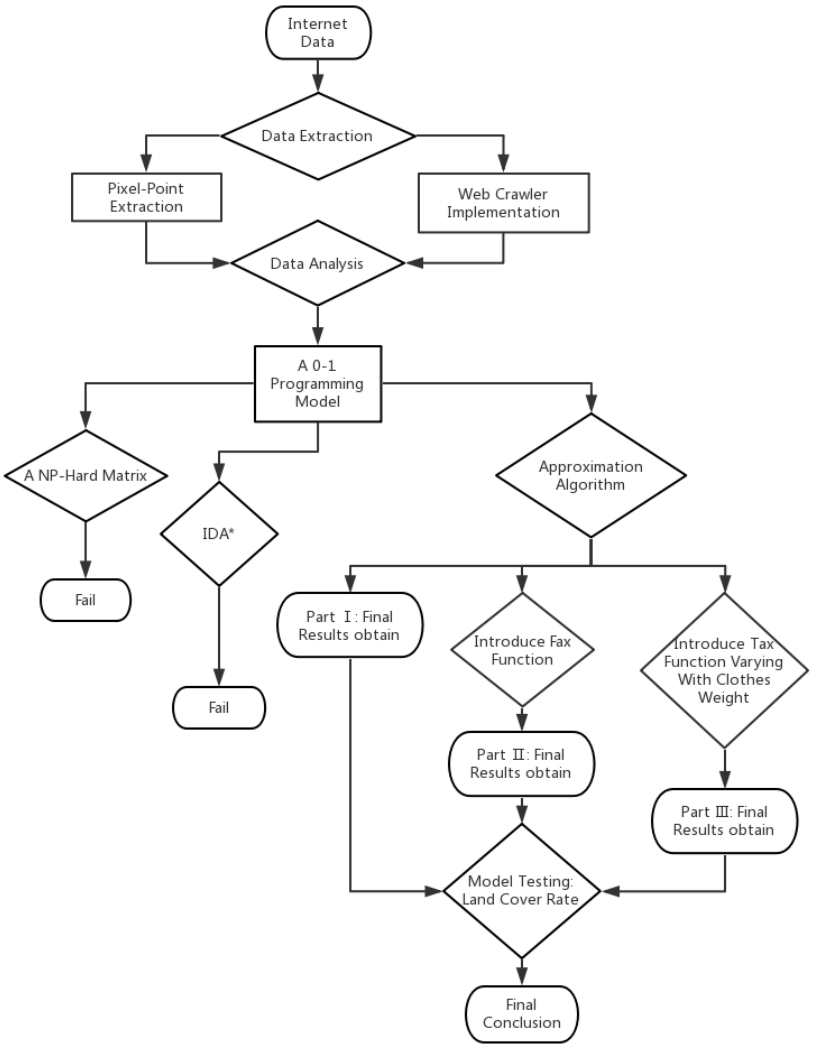

Figure 1: Flow Chart

3.1 Data Extraction

This section analyzes how we extract the data needed (the first step in the figure 1) on the Internet, in details and descriptions.

3.1.1 Web Crawler Implementation

Explicit data are not provided in the problem as input data, consequently we have to visit the website for details. The UPS official website [5] is not well formed, due to the obsolete website design. However, using Fiddler Web Debugger we are able to retrieve the image data

through the ZIP code.

The ZIP code is passed in with an HTTP request of the type `x-www-form-encoded`. The parameter named `zip` defines the ZIP code, in a string format. After passng in this data we shall use regular expressions to retrieve the image from the server and save it to a file. The code of implementation could be found in the appendix.

There are also a few packages in Python providing faster and easier implementations of HTTP requests manipulation support. Such packages include `urllib` and `requests` [6]. The pseudo code of this procedure is written as follows:

Algorithm 1 Web Crawler Algorithm

```
procedure DownloadImageByZip(zip)
    url ← "https://www.ups.com/maps/results"
    packet ← EmbedInformationOfZipCode(zip)
    request ← Requests.Post(url, packet)
    httpdata ← request.data
    if formatpattern not in httpdata then
        return false
    end if
    imagepath ← Regex.Match(formatpattern, imagepath)
    imagedata ← Requests.Get(imagepath)
    return imagedata
end procedure
```

3.1.2 Pixel-Point Extraction

We regard a set of sample points of a size 3000 (See 3.1.2), which serves as the sample of cities or towns. These towns, will cooperate with us in the procedure of evaluating the importance and coverage of individual warehose location selection candidates.

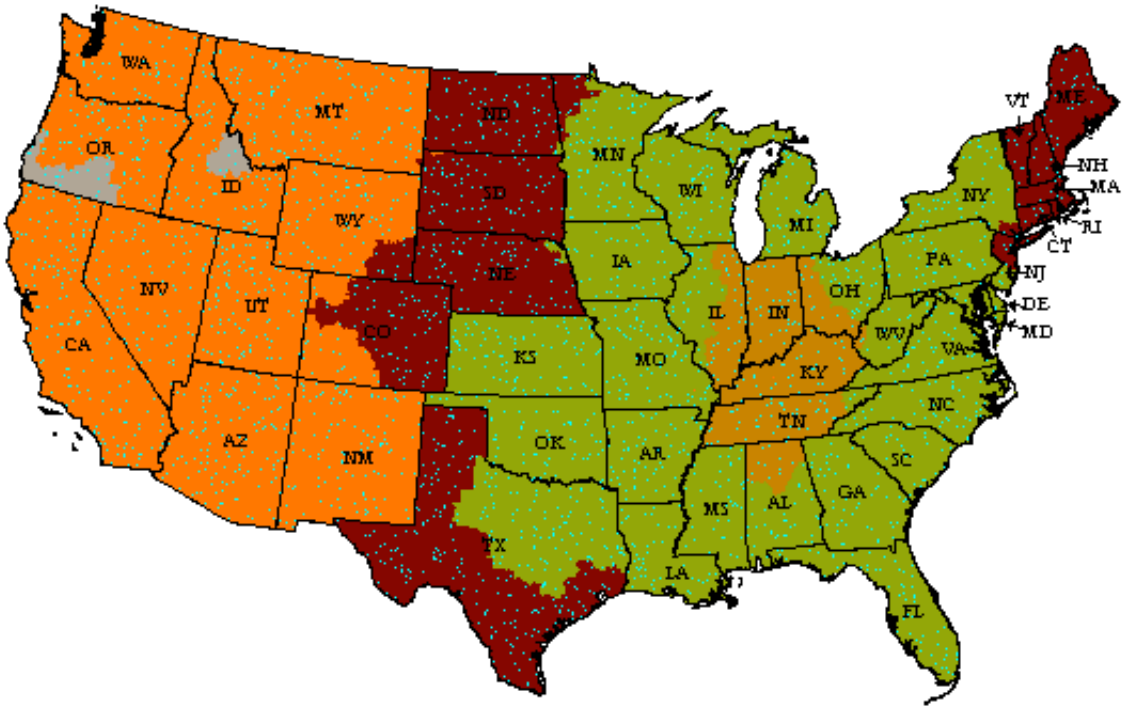

Figure 2: 3000 Sample Points

Using the builtin `Python Imaging Library` [7], we could extract the points of randomly chosen pixel locations. Due to the immense amount of selected sample points, we could gurantee that the map is or is near to evenly covered by the sample points, which, in return, guranteed our correctness in approximation of evaluating the locations of the warehouses.

The `getpixel()` function in `Python Imaging Library` retrieves the colour of pixels, which we use to test whether the chosen points are actually inside the states themselves. These points' coordinates are to be outputted to a given file handle, which is to be directed to a Python script, that determines the one-day ground shipping time between arbitrary warehouses and sample points.

Figure 3: Overall Potential Land Coverage: 95.80%

These data, namely the shipping time, will be transferred to the `C++` code that processes the decisive operations.

We have clearly demonstrated the sample points' randomizer in pseudo code, as stated follows:

Algorithm 2 Pixel Point Extraction Algorithm

```
procedure ExtractPixelPoints(image, n)
    acceptedcolours ← potential colours in all states
    markercolour ← rgb(0, 255, 255)
    markers ← ∅
    for i ∈ [0, n) do
        point ← (0, 0)
        while image[point.x][point.y] not in acceptedcolours do
            point ← Random((x, y)s.t.x ∈ [0, width), y ∈ [0, height))
        end while
        SetPixelData(image, point, markercolour)
        Append(markers, point)
    end for
    return markers, image
end procedure
```

3.2 Data Analysis: A 0-1 Programming Model

Following is the illustration of this 0-1 programming model in details.

Firstly, let us suppose that $S = \{e_1, e_2, ..., e_m\}$ is a universe set which is the set of sample points such that $|S| = m$ (see3.1.2) in problem B. S_1,S_2,...,S_n are subsets of S. If

$J \subseteq \{1, 2, ..., n\}$, and $\bigcup_{j \in J} S_j = S$, we call $\{S_j\}_{j \in J}$ is a set covering of S. The question is to find a set covering which has the minimum number of elements, which represents the minimum counts of warehouses in U.S. when applying in our question.

In fact, $\{S_j\}_{j \in J}$ is a set covering of S means that every element in S is included in at least one set $S_j (j \in J)$.

For each subset $S_j (j = 1, 2, ..., m)$, we introduces a decision variable:

$$x_j = \begin{cases} 1, \text{if } j \in J; \\ 0, \text{else} \end{cases}$$

Therefore we can build a 0-1 programming model A for this set covering problem:

$$min \sum_{j=1}^{m} x_j$$

$$s.t. \sum_{j:e_i \in S_j} x_j \geq 1, i = 1, 2, ..., n$$

$$x_j = 0, 1, j = 1, 2, ..., m$$

The constraint "$\sum_{j:e_i \in S_j} x_j \geq 1, i = 1, 2, ..., n$" can guarantee that each element e_i in S will at least be covered by one of the subset of $S_j (j \in J)$.

4 Algorithms

4.1 An NP-Hard 0-1 Matrix: *Failed*

To commence, we simplified the question by enlarging the scope of analysis from cities to states and assuming that the possible warehouses are located at their capital of state.

In this way, we can extract 48 points from the maps. Then we use these 48 points to build a 0-1 matrix of which "1" represents that online orders can arrive in one-day ground shipping, while "0" vice versa.

By analyzing through pixel-point extraction (see 3.1.2) those maps downloaded from URL employing web crawler implementation (see 3.1.1), we can obtain all the data and create this 0-1 matrix.

We now define each line in the 0-1 matrix as $S_i, i \in \{1, 2, ..., 48\}$, $S_i = \{x_{ia}, a \in N^* \cap [1, 48]\}$. Therefore the question is reduced to a form of:

$$\text{find } J \subset \{1, 2, 3, ..., n\} \ s.t. \ \forall j \in J, \bigcup_{i \in N^*, 1 \leq i \leq n} \{x_{ij}\} = \{1\}$$

4.2 Optimized Brute Force Search(IDA*): *Failed*

Due to the high proposed complexity of the first algorithm, we decided to reduce the constant, i.e. the running time by a large factor. If the factor is large enough, it could potentially reduce the new algorithm to a near-polynomial time. These optimizations can be described as follows:

When not using the sample points' algorithm, which can be understood as regarding the individual states as the only sample points, which held a stable time complexity of $O(n! \cdot m)$ in the first algorithm. In this algorithm, we deduce all states which can only be covered by one warehouse, and the respective warehouses also, before we begin the brute-force deep first search procedure. Upon this about one third of the points can be removed, which contributes greatly to the reduction of overall time.

Using branch cutting in the brute force search algorithm, after we found a solution, any solution with a depth larger than the solution with the minimum steps would be removed. This suggests that we may cut out a lot of other useless solutions during the search.

However, as we were about to implement this algorithm, we decided that it would not be appropriate to select only a few dozens of sample points, as this would deal a great amount of precision loss. Therefore this algorithmic optimization which mainly focused on constant factor reduction was actively dropped out of the discussion and would not evolve into actual code implementations.

4.3 Approximation Algorithm: *Succeeded*

Based on and inherited from the two aforementioned algorithms, we identified each of the disadvantages listed below:

The inefficient computational complexity: since we do not need to give all the possible locations of the warehouses, the algorithms designed through these direct and brute-force-based ideas are unnecessary. Therefore, improvements that should avoid non-polynomial complexity are preferred and required.

We discovered the reason why a perfect solution without a non-polynomial solution cannot be avoided is that the problem itself is in fact a set-cover problem. We can treat all cities in the United States as a set, and each warehouse covers a subset of cities, which should be time-wise, cost no longer than one day through means of ground shipping. These subsets are to be chosen, which should sum up to the entire set, i.e. the union of all sets should be or nearly approaching the set of all cities of the United States.

To reduce the general burden of this procedure and algorithm, we chose a set of 3000 points, randomly from the given ground shipping coverage maps, instead of all the cities of the Unitied States, which can be a large pack of data to be processed by general personal computers.

After all aforementioned assumptions we had given, we may come to a conclusion that a set cover problem can only be solved without guranteed solution perfectness, if a polynomial time complexity is required. We therefore discovered and developed a greedy algorithm, which is based on the general principles of Chvatal's algorithms. This algorithm can gurantee us a large amount of data with a high probability of closeness.

We used part of the approximation algorithm and derived from some of its core subroutines this partially new algorithm, which summarized and described as below: [2]

Instead of taking the whole situation into account, we decide the warehouses' locations one by one (as mentioned in the summary sheet): After finishing a single choice, we continue to search for the current optimal choice, not taking into account how this choice affects latter choices. This process would not require a time complexity higher than $O(m)$. Thereafter this choice is included in the final result, while these steps are repeated until the final result meet the termination requirement by the defined configuration.

In the following algorithm that approximates the solution, these variables are defined as follows:

$\widetilde{S}_j$is the set of the newly added sample points after selectingS_j.

$F$is the set of sample points that are already chosen.

$J$is the set of the numbers of chosen warehouses.

Algorithm 3 Approximation Algorithm

procedure FINDOPTIMALSOLUTION($m, \alpha, \psi, S, S_1, S_2, ..., S_n$)
  $F \leftarrow \emptyset$
  $J \leftarrow \emptyset$
  while $|F| < \alpha \cdot m$ **do**
    For every j, $\widetilde{S}_j \leftarrow S_j \setminus F$
    $j^* \leftarrow j \in [1, |S|] \cap N_+ \ s.t. \ max\{\frac{\widetilde{S_j}}{\psi_j^\gamma}\}$
    $J \leftarrow J \cap \{j^*\}$
    $F \leftarrow F \cap \{S_{j^*}\}$
  end while
  return J
end procedure

It is noteworthy that the result of approximation algorithm is not technically the global optimal solution but the local optimal solution. But in some situation, the global optimization is not needed, or its mathematic meaning is not the result we are looking

forward to. Therefore, the essence of approximation algorithm accurately solve our problem for it neither looks for the best allocation of the selected warehouses (which presumably have the same land cover rate), nor we need to provide with all the possible solutions.

By acting on the local optimization, we can significantly decrease the algorithm complexity which can be seen in table 5.

4.3.1 Part I

When land cover rate decreases, the requirement for warehouses decreases as well. In the approximation algorithm stated in 4.3, because that taxes are not taken into consideration in this part of the problem, we define the tax weight as:

$$\psi_j = 1, j \in \{1, 2, 3, ..., n\}$$

The following is the result when warehouses can radiate all area (relatively: see denifition in 2):

Zip_code	State	City
1841	MA	Lawrence
18041	PA	East Greenville
26241	WV	Elkins
29601	SC	Greenville
39841	GA	Damascus
46241	IN	Indianapolis
47721	IN	Evansville
49441	MI	Muskegon
57001	SD	Alcester
58041	ND	Hankinson
59001	MT	Absarokee
59841	MT	Pinesdale
64121	MO	Kansas City
69001	NE	McCook
71161	LA	Shreveport
76021	TX	Bedford
78841	TX	DelRio
82001	WY	Cheyenne
83321	ID	Castleford
85001	AZ	Phoenix
87001	NM	Algodones
93721	CA	Fresno
97281	OR	Portland

Table 1: Part I warehouses location: 23 in total, cover 100.00% of land area

But in real life situation, if we slightly cut down the land cover rate, the number of warehouses needed will decrease significantly. For instance, as table 2 and figure 4.3.1 had shown, if we decrease 3.86% on the demand for land cover rate, we can obtain a 17.4% decrease on the demand for warehouses.

Zip_code	State	City
1841	MA	Lawrence
18041	PA	East Greenville
29601	SC	Greenville
39841	GA	Damascus
46241	IN	Indianapolis
49441	MI	Muskegon
58041	ND	Hankinson
59001	MT	Absarokee
59841	MT	Pinesdale
64121	MO	Kansas City
71161	LA	Shreveport
76021	TX	Bedford
78841	TX	DelRio
82001	WY	Cheyenne
83321	ID	Castleford
85001	AZ	Phoenix
87001	NM	Algodones
93721	CA	Fresno
97281	OR	Portland

Table 2: Recommendation for part I warehouses location: 19 in total, cover 96.14% of land area

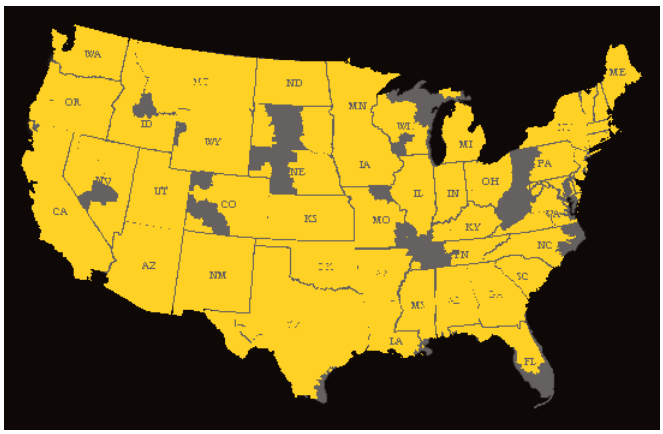

Figure 4: Radiated Area for warehouses in part I

CONCLUSION: Result for 100% land cover rate needs 23 warehouses as shown in table 1. But for recommendation, 96.14% land cover rate is better and only need 19 warehouses. The locations can be seen in figure 4.3.1 and table 2.

4.3.2 Part II: Effect brought by the Tax Rate

In this part, we are required to give the minimum customers' tax liability rather than the minimum warehouses' quantity. Therefore, we make slight change of the 0-1 programming model, named as model II.

To commence, we changed the definition of the decision variable in part I to the city *No.j* state tax rate: T_j. Then the decision changes into:

$$min \sum_{j \in J} T_j$$

Then, we assume that the total land cover area is α time(s) the total land area, and then we list different tax weight according to different α. Game can be applied in terms of reality to reach equlibrium. The constraint condition changes into:

$$s.t. \sum_{j:e_i \in S_j} x_j \geq 1, i \in I, \frac{|I|}{n} \geq \alpha$$

$$j = 1, 2, ..., m$$

As for how to realize the model II, we have two means:

1. **Change the constraint condition**

   It is not difficult to conclude that except for the none-tax state, all the other states' tax rates fluctuate between $4\% - 6\%$. The slight difference also appears in states' GSP (Gross State Product) [3]. Therefore, we deem the state's tax [4] rate is proportionate to the number of warehouses in each state. In precondition of a less number of warehouses, it is appropriate to select the state with the minimum tax rate. The rule for the selection of j^* is as follow:

   If

   $$|\bigcup_{j \in J} S_j| \geq \alpha |S|$$

   So we suppose:

   $$T_{j^*} = min\{T_j || \widetilde{S_j}| \geq \delta \times \max_{1 \leq j \leq m} |\{\widetilde{S_j}\}|\}$$

   $$J := J \cup \{j^*\}, \widetilde{S_j} := \widetilde{S_j} \backslash \widetilde{S_{j^*}}, (j = 1, 2, ..., m)$$

The merit of changing the constrait condition is that we can find the optimal δ by multiple computations, which efficiently avoids the none-tax rate problem (compare with the sencond means).

The disadvantages lie in the overlook of the difference of each state's GSP and tax rate and this means' result is unsatisfied.

2. **Change the decision variable**

We need to consider how to minimize the customers' tax liability when the total land cover rate is a constant, which means to minimize:

$$\frac{Total\_Tax^{\gamma}}{Total\_Land\_Cover\_Rate}$$

Therefore, we define Q_j as the value of the city $No.j$:

$$Q_j = \frac{|S_j|}{{T_j}^{\gamma}}, \gamma = a\_self\_defined\_constant \geq 0$$

The decision changes into:

$$max \sum_{j \in J} Q_j$$

The merits of changing the decision variable is that we can find the optimal γ through experiments with computer. Also, since we do not change the constraint condition, it is convenient to revise the code based on part I.

The disadvantage is that the Q_j do not have meaning if city $No.j$ is in none-tax rate state, since ${T_j}^{\gamma}$ is an denominator. To solve this problem, we laborly define tax rate in the none-tax rate state as 0.0000000000001.

When $\gamma = 0$, which means we do not consider the effect caused by the tax rate brought, all the data we get is the same in part I.

Therefore, in this model, considering not the affectiveness of garment taxes, we define the affectiveness weight as follows:

$$\psi_j = tax\_weight = T(j) \cdot GSP[j]$$

Whereas GSP is the Gross State Product of the state where the warehouse is located in.

Following is a part of the results, showing how this allocation effect the customers' tax liability. By analyzing this data, we can choose the best data for images and final results.

Land Cover Rate(%)	Total Count of Warehouses	Total Tax(million dollars)
91	17	3.39057
91.2	17	3.39057
91.4	17	3.39057
91.6	17	3.39057
91.8	17	3.39057
92	17	3.39057
92.2	17	3.39057
92.4	17	3.39057
92.6	18	3.67197
92.8	18	3.67197
93	18	3.67197
93.2	18	3.67197
93.4	18	3.67197
93.6	18	3.67197
93.8	18	3.67197
94	18	3.67197
94.2	18	3.67197
94.4	18	3.67197
94.6	18	3.67197
94.8	19	3.67197
95	19	3.67197
95.2	19	3.67197
95.4	19	3.67197
95.6	19	3.67197
95.8	19	3.67197
96	19	3.67197
96.2	20	3.89494
96.4	20	3.89494
96.6	20	3.89494
96.8	20	3.89494
97	20	3.89494

Table 3: part II land cover rate and correspoding tax and warehouses' number

In this model, in order to determine a appropriate γ, we select threee representives: 0.5, 1, 2 to make imgaes and then compare.

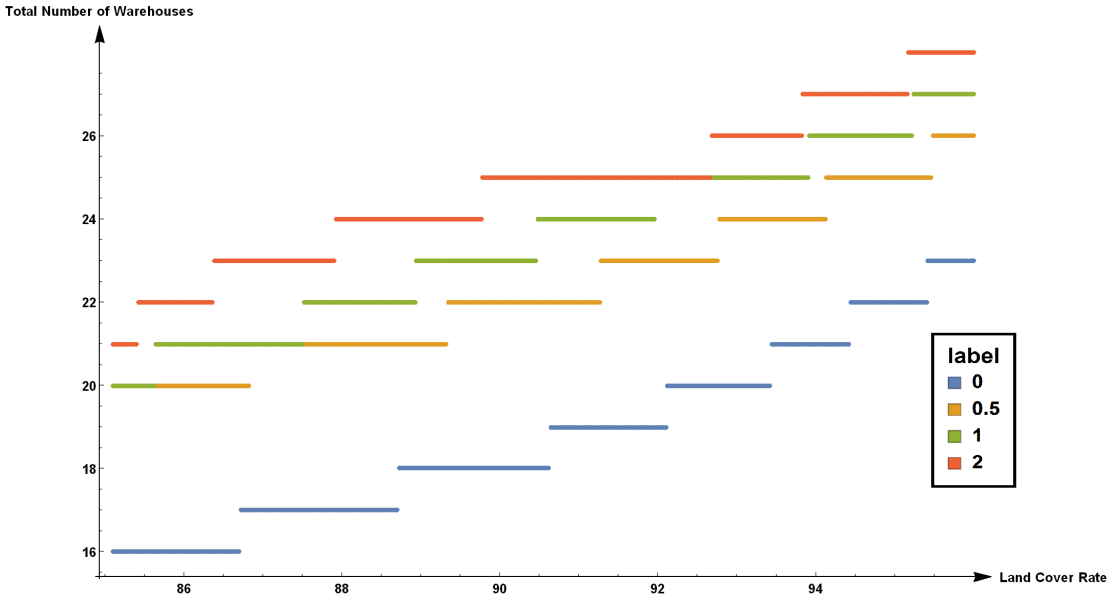

Figure 5: Relations between γ, Land Cover Rate and the result

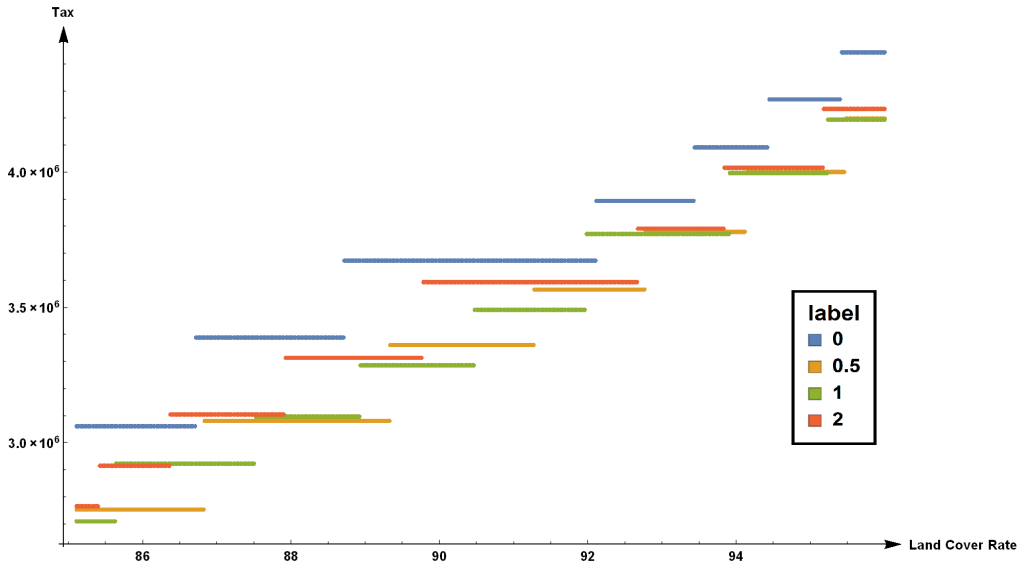

Figure 6: Relations between γ, Land Cover Rate and total tax weight

It is not difficult to find that once we take tax rate into consideration, the total tax decrease. Also, $\gamma = 2$ is worse than $\gamma = 1$ and $\gamma = 0.5$ and the difference between $\gamma = 1$ and $\gamma = 0.5$ is too small. The $\gamma = 0.5$ advantages over the other two data; therefore, we set $\gamma = 0.5$ in our model. The result is as following:

Zip_code	State	City
1841	MA	Lawrence
19961	DE	LittleCreek
27201	NC	Alamance
35201	AL	Birmingham
45041	OH	Miamitown
54201	WI	Algoma
57001	SD	Alcester
57481	SD	Westport
59001	MT	Absarokee
59841	MT	Pinesdale
64121	MO	KansasCity
71161	LA	Shreveport
76021	TX	Bedford
78841	TX	DelRio
80121	CO	Littleton
83321	ID	Castleford
85001	AZ	Phoenix
87001	NM	Algodones
93721	CA	Fresno
97281	OR	Portland
97401	OR	Eugene
97801	OR	Pendleton

Table 4: part II warehouses location: 22 in total, cover 95.26% of land area, total weight of taxes: 3.36157×10^6 million dollars

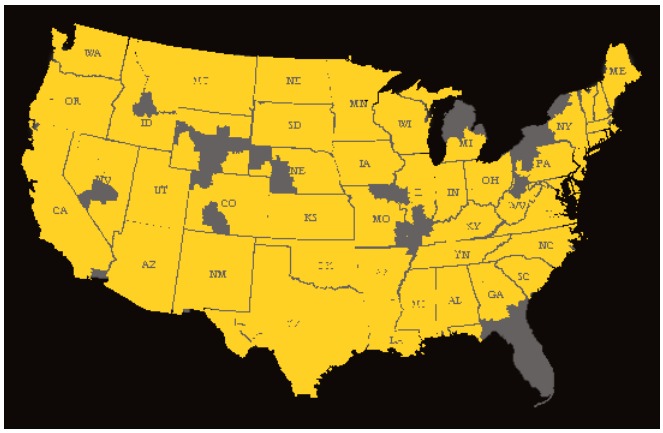

Figure 7: Radiated Area for warehouses in part II

CONCLUSION: The adjusted number of warehouses are 22, covering 95.26% of

land area (See table 4). Total weight of taxes is 3.36157×10^6 dollars. Also, we list the land cover rate and correspoding tax and warehouses' number in table 3 for sensitive testing.

4.3.3 Part III: Effect brought by the Emergent Clothes Business

As mentioned in 2.3, we defined the T_2 function (to substitute the tax rate function in part II) as following as:

$$T_2 = T \times (1 - \eta_{gar}) + T_{gar} \times \eta_{gar}$$

The affectiveness weight in Part ii can be minorly modified to satisfy the needs of this solution as:

$$\psi_j = T_2(j) \times GSP[j]$$

According to the Bureau of Labor Statistics of U.S.(see 2.3), $\eta = 3.3\%$:

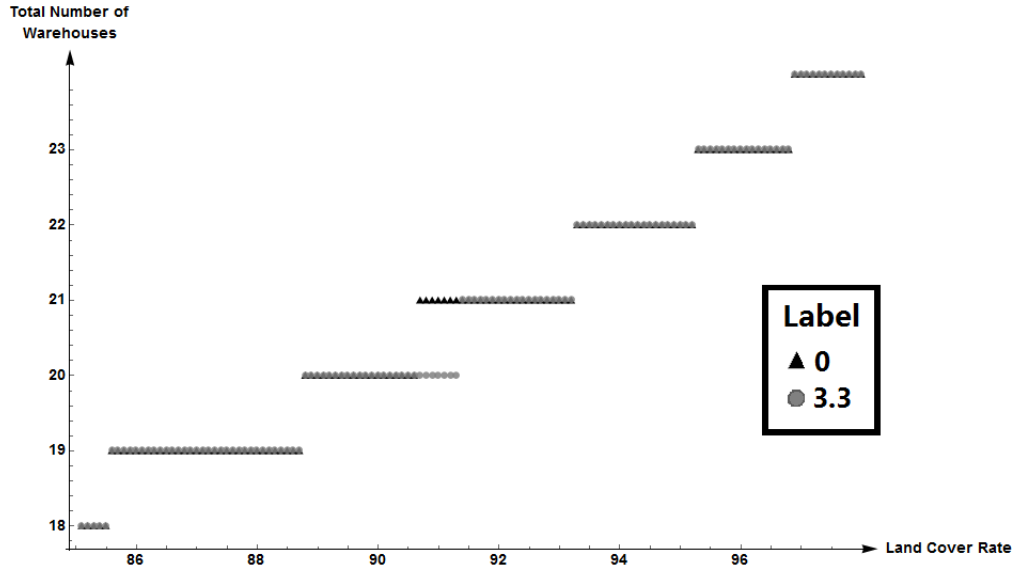

Figure 8: Relations between warehouses Land Cover Rate with/without η

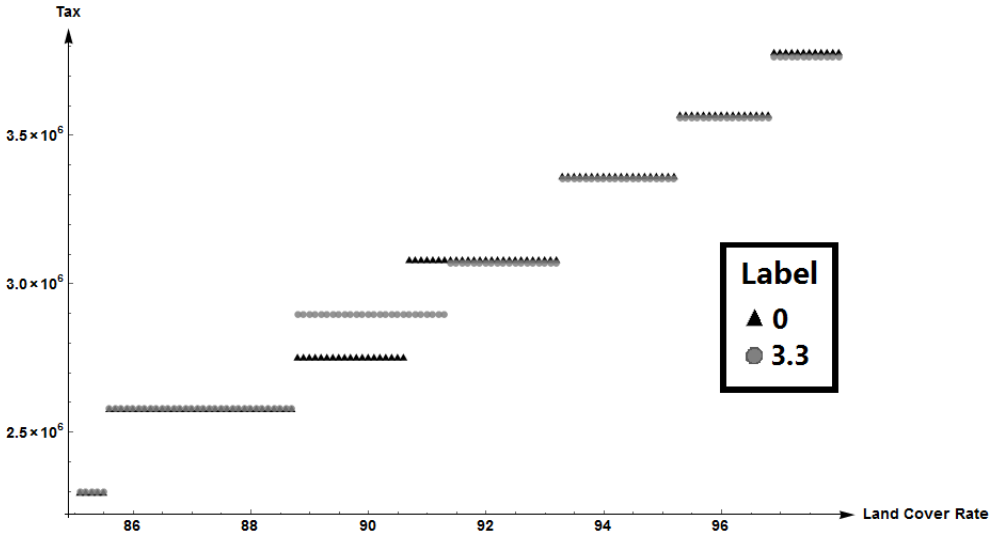

Figure 9: Optimized relations between warehouses Land Cover Rate with/without η

In this picture, we can see the difference about the existence of η_{gar}. In this way, both weight of taxes and number of warehouses will decrease in a small degree.

Next, we make the constant η_{gar} a variable with a scope from 2.1 through 4.4 to test the sensitiveness of η_{gar}.

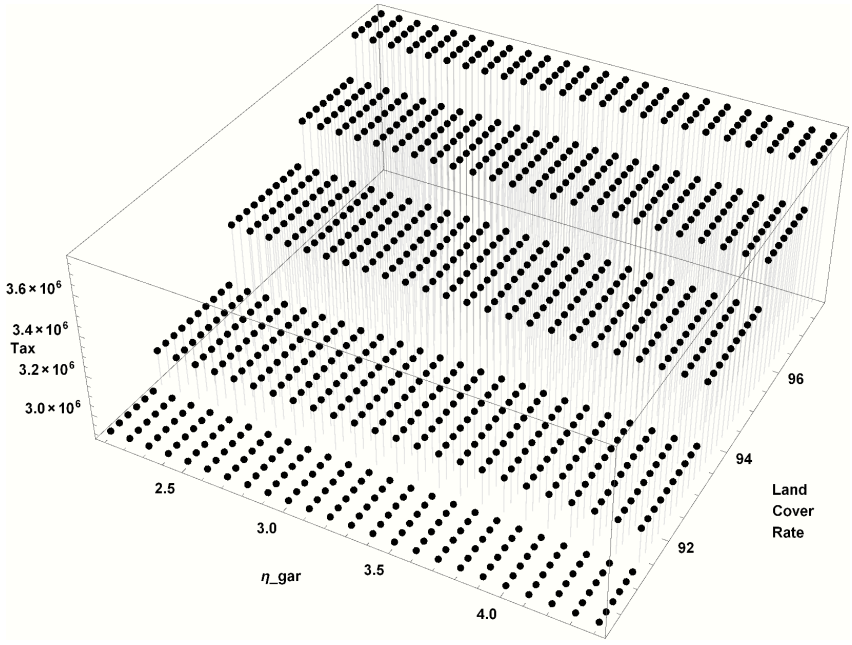

Figure 10: Tax-η_{gar} function A_1 in part III

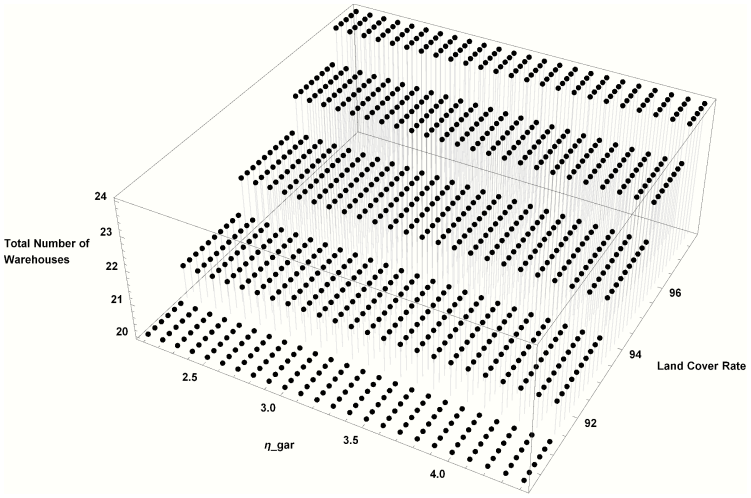

Figure 11: Warehouses numeber-η_{gar} function A_2 in part III

From the picture above, we can conclude that η_{gar} is not a sensitive factor to the model. (Obviusly, the change of η_{gar} will lead to the varying of tax rate in the same city.) Therefore, our model and method analyzed that it is stable. We altered η_{gar} from 1 through 9.8, which we believed was the appropriate and sufficient range for sensitivity testing.

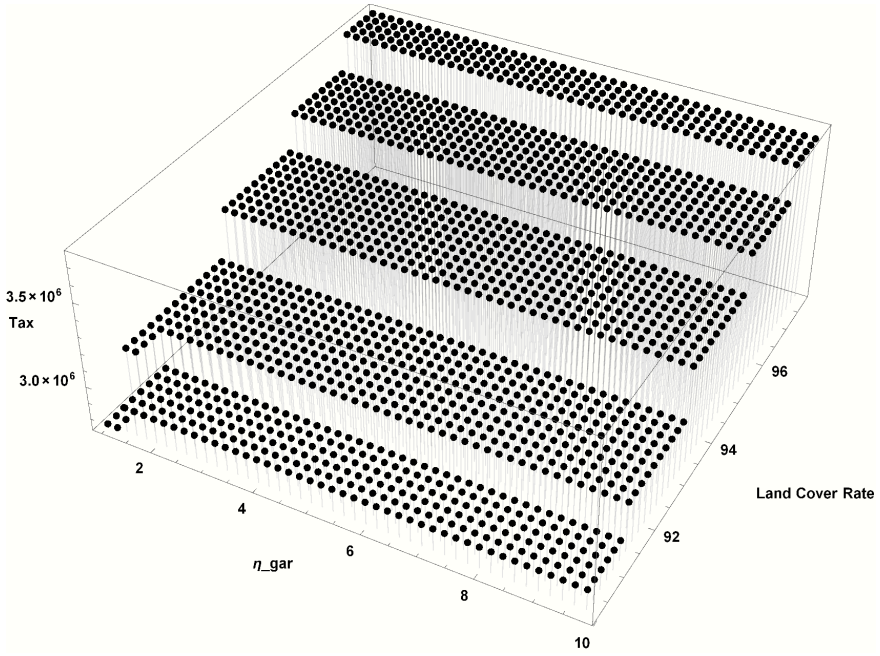

Figure 12: Warehouses number-η_{gar} function B_1 in part III

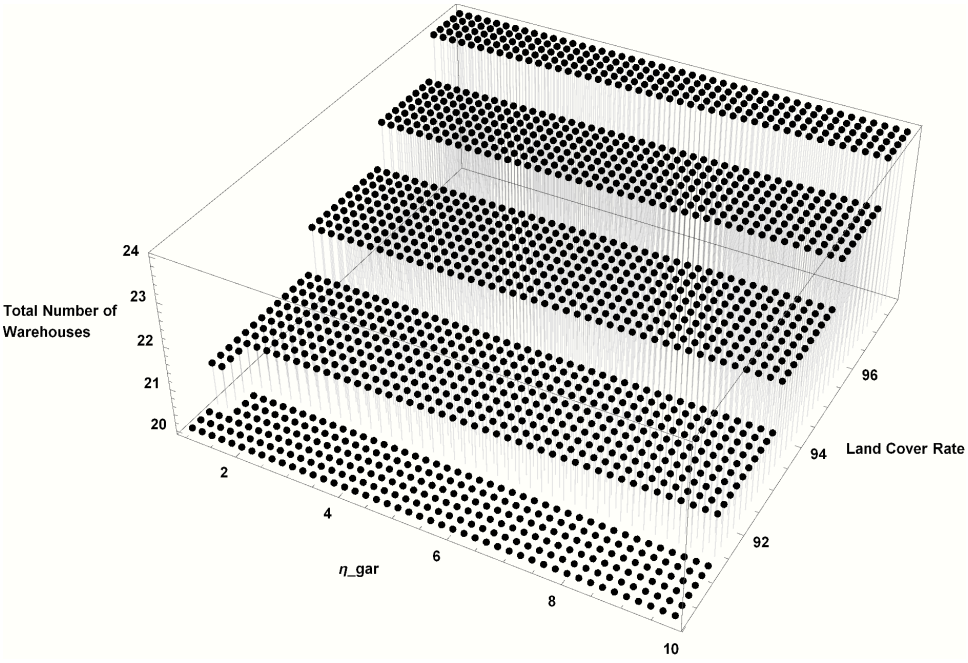

Figure 13: Warehouses number-η_{gar} function B_2 in part III

Result shows that, changes only occur while η_{gar} ranges from 1.0 to 1.2 affects the overall warehouse selection, which is relatively small to the actual data. When η_{gar} is large enough, proposedly larger than 3.3, no obvious changes are dealt to the evaluation result.

CONCLUSION: After the introduction of clothes and apparel businesses, we find that the result would not change unless the land cover rate varys from 90.7%-91.3%. Therefore, we decide that the adjusted number and location of warehosues are the same in part I (shown in figure 3).

5 Model Analysis

This section analyzes how the approximation algorithm as mentioned in 4.3 applys in Part I, II, III in details and descriptions.

5.1 Algorithm Complexity

Algorithm Complexity	n	n=300	Orders of Magnitude
An NP-Hard 0-1 Matrix	$n!$	300!	10^{614}
IDA*	$\frac{n}{k}!$	$\frac{300}{k}!$	$10^{262} - 10^{32}$
Approximation Algorithm	$n^3 log_2 n$ or $n^2 log_2 n$	$300^3 log_2 300$ or $300^2 log_2 300$	$10^6 - 10^8$

Table 5: Three Algorithm Complexities (generally $2 < k < 10$)

It is easily noticed that the approximation algorithm has outstand others remarkably.

6 Model Testing and Conclusion

6.1 Land Cover Rate

As shown in table 4, a fairly optimal solution is to achieve 95.26% of land area coverage with 22 warehouses, which poses a total weight of taxes of 3.36157×10^6 million dollars. This solution is locally optimal and covers a reasonable amount of area with a reasonable amount of warehouses, which is preferrable under most occasions.

The cities that are chosen are: Lawrence, Little Creek, Alamance, Birmingham, Miamitown, Algoma, Alcester, Westport, Absarokee, Pinesdale, Kansas City, Shreveport, Bedford, DelRio, Littleton, Castleford, Phoenix, Algodones, Fresno, Portland, Eugene and Pendleton (For details, see table 4).

6.2 Model advantages

This model holds a polynomial time complexity, which infers a rather faster run time, that gurantees that the selection of the result can be obtained immediately after the decision was made. In this way this model is way faster than most of the methods available in means of time.

In addition, one can simply change the constant m to a larger number, that will take more sample points into consideration, and greatly improve the precision. This, on one side can be applied to arbitrary countries and even single states. In this aspect, this model we designed can also be applied to other regions and countries if the company decided to enlarge its market, with minor modifications.

Thus, in comparatively ignorable time required, the company could quickly decide where to build its warehouses and immediately start its business. Since the need of financial

support is crucial in businesses, we should note that the faster a decision was made, the more beneficial it is to the company.

In addition, the ψ variable in this model can be greatly extended to other expertises and aspects. Extensions to more businesses and tax definitions can be made in short time.

We can conclude from the aforementioned arguments that this model is not only visibly fast, but also extremely extensible for further exploits in newly risen markets.

6.3 Model disadvantages

This model, however fast, is not a perfect model. We already argued that a perfect solution is not achievable with polynomial time complexity. Therefore this model could not retrieve the best solution existent. This might not gurantee the perfect minimization of taxes and minimization of warehouses' count.

In other words, this means the solutions we chosen are not actually the **best** solutions but clearly well-formed solutions we can obtain in limited time. The only defect in our model is that it cannot produce a perfect solution, in reasonable time, which is the major disadvantage of our model.

References

[1] `http://www.bls.gov/news.release/cesan.nr0.htm` Bureau of Labor Statistics of U.S, 2016.

[2] WANG Jiqiang, *Model and algorithm for set cover problem*, Computer Engineering and Applications, Volume 49, 2013.

[3] `http://www.bea.gov/newsreleases/regional/gdp_state/2012/pdf/gsp0612.pdf`, *Advance 2011 and Revised 19972010 GDP-by-State Statistics*, Bureau of Economic Analysis, U.S. Department of Commerce, 2012.

[4] `http://www.tax-rates.org/taxtables/sales-tax-by-state`, *Sales Tax Rates By State*, Tax-Rates.org, 2016.

[5] `https://www.ups.com/maps?loc=en_US&srch_pos=1&srch_phr=maps`, *Ground Time-in-Transit Maps*, United Parcel Services, 2016.

[6] `http://www.python-requests.org/en/master/user/quickstart/`, *More complicated POST requests*, Kenneth Reitz.

[7] `http://pillow.readthedocs.io/en/latest/reference/Image.html` *Pillow: Image Module.*

7 Appendixes

7.1 Web Crawler

data_processor/const.py

```
1
2  states = [
3      'AK', 'AL', 'AR', 'AZ', 'CA', 'CO', 'CT', 'DC', 'DE', 'FL',
4      'GA', 'HI', 'IA', 'ID', 'IL', 'IN', 'KS', 'KY', 'LA', 'MA',
5      'MD', 'ME', 'MI', 'MN', 'MS', 'MT', 'NC', 'ND', 'NE', 'NH',
6      'NJ', 'NM', 'NV', 'NY', 'OH', 'OK', 'OR', 'PA', 'RI', 'SC',
7      'SD', 'TN', 'TX', 'UT', 'VA', 'VT', 'WA', 'WI', 'WV', 'WY',
8  ]
9
10 # Range provided by http://www.structnet.com/instructions/
       zip_min_max_by_state.html
11 state_zip_idx = {
12     'AK': 99501, 'AL': 35004, 'AR': 71601, 'AZ': 85001, 'CA': 90001,
13     'CO': 80001, 'CT':  6001, 'DC': 20001, 'DE': 19701, 'FL': 32004,
14     'GA': 30015, 'HI': 96701, 'IA': 50001, 'ID': 83201, 'IL': 60001,
15     'IN': 46001, 'KS': 66002, 'KY': 40003, 'LA': 70001, 'MA':  1001,
16     'MD': 19701, 'ME':  3901, 'MI': 48001, 'MN': 55001, 'MS': 38601,
17     'MT': 59001, 'NC': 27006, 'ND': 58001, 'NE': 68001, 'NH':  3031,
18     'NJ':  7001, 'NM': 87001, 'NV': 88901, 'NY': 10001, 'OH': 43001,
19     'OK': 73001, 'OR': 97001, 'PA': 15001, 'RI':  2801, 'SC': 29001,
20     'SD': 57001, 'TN': 37010, 'TX': 75001, 'UT': 84001, 'VA': 20040,
21     'VT':  5001, 'WA': 98001, 'WI': 53001, 'WV': 24701, 'WY': 82001,
22 }
23
24 state_name_idx = {
25     'AK': 'Alaska', 'AL': 'Alabama', 'AR': 'Arkansas', 'AZ': 'Arizona',
26     'CA': 'California', 'CO': 'Colorado', 'CT': 'Connecticut', 'DC': 'Dist
       of Columbia',
27     'DE': 'Delaware', 'FL': 'Florida', 'GA': 'Georgia', 'HI': 'Hawaii', '
       IA': 'Iowa',
28     'ID': 'Idaho', 'IL': 'Illinois', 'IN': 'Indiana', 'KS': 'Kansas', 'KY'
       : 'Kentucky',
29     'LA': 'Louisiana', 'MA': 'Massachusetts', 'MD': 'Maryland', 'ME': '
       Maine',
30     'MI': 'Michigan', 'MN': 'Minnesota', 'MS': 'Mississippi', 'MT': '
       Montana',
31     'NC': 'North Carolina', 'ND': 'North Dakota', 'NE': 'Nebraska', 'NH':
       'New Hampshire',
32     'NJ': 'New Jersey', 'NM': 'New Mexico', 'NV': 'Nevada', 'NY': 'New
       York', 'OH': 'Ohio',
33     'OK': 'Oklahoma', 'OR': 'Oregon', 'PA': 'Pennsylvania', 'RI': 'Rhode
       Island',
34     'SC': 'South Carolina', 'SD': 'South Dakota', 'TN': 'Tennessee', 'TX':
       'Texas',
```

```
    'UT': 'Utah', 'VA': 'Virginia', 'VT': 'Vermont', 'WA': 'Washington',
    'WI': 'Wisconsin', 'WV': 'West Virginia', 'WY': 'Wyoming',
}

state_pixel_idx = {
    'AK': ( 94, 285), 'AL': (374, 235), 'AR': (314, 213), 'AZ': (118, 205), 'CA': ( 39, 155),
    'CO': (185, 158), 'CT': (493, 110), 'DC': (493, 122), 'DE': (477, 152), 'FL': (433, 283),
    'GA': (405, 231), 'HI': (207, 329), 'IA': (299, 126), 'ID': (112,  84), 'IL': (344, 143),
    'IN': (373, 146), 'KS': (254, 169), 'KY': (378, 186), 'LA': (317, 264), 'MA': (497, 101),
    'MD': (478, 158), 'ME': (511,  62), 'MI': (379, 105), 'MN': (294,  71), 'MS': (346, 241),
    'MT': (163,  52), 'NC': (437, 200), 'ND': (243,  71), 'NE': (242, 132), 'NH': (497,  85),
    'NJ': (479, 125), 'NM': (174, 210), 'NV': ( 79, 134), 'NY': (466 , 94), 'OH': (403, 140),
    'OK': (266, 206), 'OR': ( 46,  70), 'PA': (439, 132), 'RI': (503, 107), 'SC': (433, 219),
    'SD': (239,  92), 'TN': (366, 207), 'TX': (239, 253), 'UT': (129, 144), 'VA': (445, 172),
    'VT': (486,  80), 'WA': ( 61,  43), 'WI': (332,  90), 'WV': (427, 158), 'WY': (173, 103),
}

dist_colours_idx = [
    (-1 , -1,  -1 ), (255, 209, 36 ), (201, 132, 0  ), (147, 167, 8  ), (133, 6,   0  ),
    (255, 120, 0  ), (176, 166, 150), (0  , 129, 152), (184, 228, 212), (204, 234, 141),
]
```

data_processor/city_name_download.py

```
import requests
import re

def download_city_by_zip_code(zip_code):
    payload = {
        'loc': 'en_US',
        'usmDateCalendar': '11/13/2016',
        'zip': str(zip_code).rjust(5, '0'),
        'stype': '0', # 'D' for to, '0' for from
        'submit': 'Submit'
    }
    req = requests.post('https://www.ups.com/maps/results', data=payload, verify=True)
    html_data = req.text
    # Matching...
```

```
    m_rl = re.findall(r'<span class="bold">Business days in transit.*?from
    :</span>[ \t\r\n]*(.*?)[ \t\r\n]*<br>', html_data)
    m_r = m_rl[0]
    m_r = ' '.join(m_r.split(' ')[:-1])
    city, state = m_r.split(', ')
    city = ' '.join(i.title() for i in city.split(' '))
    return city, state

f = open('points.txt', 'r')
of = open('cities.csv', 'w')
of.write('ZIP code,State,City,Image ID\n')
for i in f.readlines():
    i = re.sub('[\r\n]*', '', i)
    a, b = i.split(' ')
    try:
        c, s = download_city_by_zip_code(b)
    except KeyboardInterrupt:
        break
    except:
        c = 'Unknown City'
        s = 'US'
    fs = '%s,%s,%s,%s' % (b, s, c, a)
    of.write(fs + '\n')
    print(fs)
```

data_processor/image_downloader.py

```
import requests
import PIL
import PIL.Image
import io
for i in range(1, 691):
    j = 'map_' + str(i).rjust(4, '0')
    k = 'https://www.ups.com/using/services/servicemaps/maps25/%s.gif' % j
    r = requests.get(k)
    s = r.content
    t = PIL.Image.open(io.BytesIO(s))
    t.save('%s.png' % j)
exit(0)
```

data_processor/main.py

```

import const
import io
import json
import PIL
import PIL.Image
import random
import re
import requests

```

```
def get_states():
    """ get_states() -- Generator, returns all states"""
    for i in const.states:
        yield i
    pass

def get_zip_code_by_state(state_id):
    """ get_zip_code_by_state(state_id) -- Gets valid ZIP code by state ID
    ."""
    return const.state_zip_idx[state_id]

def get_name_by_state(state_id):
    """ get_name_by_state(state_id) -- Gets human readable name by state
    ID."""
    return const.state_name_idx[state_id]

def get_days_by_colour(pix):
    """ get_days_by_colour(pix) -- Get deliver days by colour index."""
    for i in range(1, 9 + 1):
        clr = const.dist_colours_idx[i]
        if abs(pix[0] - clr[0]) > 10:
            continue
        if abs(pix[1] - clr[1]) > 10:
            continue
        if abs(pix[2] - clr[2]) > 10:
            continue
        return i
    return 10

def get_image_by_zip_code(zip_code, get_image=True):
    """ get_image_by_zip_code(zip_code) -- Downloads image through ZIP
    code,
    from the UPS official site. """
    # Downloading HTML
    payload = {
        'loc': 'en_US',
        'usmDateCalendar': '01/07/2016',
        'zip': str(zip_code).rjust(5, '0'),
        'stype': '0', # 'D' for to, '0' for from
        'submit': 'Submit'
    }
    req = requests.post('https://www.ups.com/maps/results', data=payload,
    verify=True)
    html_data = req.text
    # Retrieving image zip code.
    try:
        img_link = re.findall(r'<img id="imgMap" src="(.*?)"', html_data)
    [0]
    except Exception as err:
        raise Exception('Unable to receive data from UPS server.')
    img_link = 'https://www.ups.com' + img_link
```

```
57      if not get_image:
58          return img_link
59      # Retrieving image in binary
60      req = requests.get(img_link, verify=True)
61      img_data = req.content
62      return img_data
63
64  def get_image_by_state(state_id):
65      """ get_image_by_state(state_id) -- Gets valid image by state ID, in
        colours."""
66      zip_code = get_zip_code_by_state(state_id)
67      img_data = get_image_by_zip_code(zip_code)
68      return img_data
69
70  def get_dist_by_state(state_id, img_data=None):
71      """ get_dist_by_state(state_id) -- Gets distance to this state by
        state ID."""
72      if img_data == None:
73          img_data = get_image_by_state(state_id)
74      bytes_io = io.BytesIO(img_data)
75      image = PIL.Image.open(bytes_io)
76      image = image.convert(mode='RGB')
77      # The following array is the returning result
78      ret = {}
79      # For every state in mind...
80      for ns_id in get_states():
81          tup = const.state_pixel_idx[ns_id]
82          pix = image.getpixel(tup)
83          # print(pix)
84          days = get_days_by_colour(pix)
85          # print(get_name_by_state(ns_id), days) # Debugging purposes
86          ret[ns_id] = days
87      return ret
88
89  def get_dist_all_states():
90      """ get_dist_all_states() -- Get distance between all states, returns
        dict."""
91      res = {}
92      for state_id in get_states():
93          print('Processing data for "%s"...' % get_name_by_state(state_id))
94          s_res = get_dist_by_state(state_id)
95          res[state_id] = s_res
96      return res
97
98  def download_dist_all_states(f_handle):
99      """ download_dist_all_states -- Format get_dist_all_states() in an OI
        data
100     style, for input into C++ programs."""
101     ret = get_dist_all_states()
102     print('Saving data...')
103     for i in get_states():
```

```
104         for j in get_states():
105             f_handle.write('%s %s %d\n' % (i, j, ret[i][j]))
106     return f_handle
107 
108 def download_image_by_json(f_handle):
109     """ download_image_by_json(f_handle) -- Donwloads images of all
    appeared
110     data, with JSON opened handle. """
111     data = f_handle.read()
112     j_data = json.loads(data)
113     img_lst = sorted(list(set(j_data['image_index'][i] for i in j_data['
    image_index'])))
114     for i in range(0, len(img_lst)):
115         j = img_lst[i]
116         f_name = j.split('/')[-1:][0]
117         req = requests.get(j, verify=True)
118         img_data = req.content
119         image = PIL.Image.open(io.BytesIO(img_data))
120         image.save('%s.png' % f_name[:-4])
121         print('Downloaded image %s. Completed %f%%' % (f_name, float(i /
    len(img_lst) * 100)) )
122         pass
123     return
124 
125 def i_get_dist_by_city(city_id, img_idx, coord_index):
126     """ i_get_dist_by_city(city_id) -- Gets distance to this city by city
    #."""
127     img_name = '../../data/map_images/map_%s.png' % str(img_idx).rjust(4,
    '0')
128     image = PIL.Image.open(img_name)
129     image = image.convert(mode='RGB')
130     # The following array is the returning result
131     ret = {}
132     # For every state in mind...
133     for i in range(0, len(coord_index)):
134         tup = coord_index[i]
135         pix = image.getpixel(tup)
136         # print(pix)
137         days = get_days_by_colour(pix)
138         ret[i] = days
139     return ret
140 
141 def i_get_all_city_dists_offline():
142     fcsv = open('../../data/cities.csv', 'r')
143     cit_dat = []
144     for i in fcsv.read().split('\n'):
145         if not i:
146             break
147         i = i.split(',')
148         # # This was removed in the first place.
149         if i[1] in {'AK', 'HI', 'PR'}:
```

```
150             continue
151         cit_dat.append((i[3], i[4], i[5]))
152     cit_idx = []
153     for i in range(0, len(cit_dat)):
154         cit_idx.append((int(cit_dat[i][1]), int(cit_dat[i][2])))
155     for i in range(0, len(cit_dat)):
156         c_ret = i_get_dist_by_city(i, cit_dat[i][0], cit_idx)
157         for j in range(0, len(cit_dat)):
158             if c_ret[j] >= 8:
159                 c_ret[j] = 7
160             fmt_str = '%d %d %d' % (i + 1, j + 1, c_ret[j])
161             print(fmt_str)
162     return
163 
164 def choose_random_points(n):
165     """ choose_random_points(n) -- Choose n random points on map. """
166     img = PIL.Image.open('../../data/map_images/map.png')
167     res = set()
168     img = img.convert('RGB')
169     while len(res) < n:
170         tup = (0, 0)
171         col = (0, 0, 0)
172         while col not in const.dist_colours_idx:
173             tup = random.randrange(0, img.width), random.randrange(0, img.
    height)
174             col = img.getpixel(tup)
175         if tup not in res:
176             res.add(tup)
177         img.putpixel(tup, (0, 255, 255))
178     return sorted(list(res))
179 
180 def main():
181     """ main() -- called when opening in shell."""
182     rp = choose_random_points(3000)
183     fcsv = open('../../data/cities.csv', 'r')
184     cit_dat = []
185     for i in fcsv.read().split('\n'):
186         if not i: break
187         cit_dat.append(i.split(',')[3])
188     cit_idx = []
189     for i in range(0, len(cit_dat)):
190         c_ret = i_get_dist_by_city(i, cit_dat[i], rp)
191         for j in range(0, len(rp)):
192             if c_ret[j] > 1: continue
193             fmt_str = '%d %d' % (i + 1, j + 1)
194             print(fmt_str)
195     # Sample points' locations not saved.
196     fcsv.close()
197     return 0
198 
199 if __name__ == '__main__':
```

```
200     exit(main())
```

7.2 Data Processor

graph_solver/graph.h

```
1
2  #ifndef GRAPH_H_
3  #define GRAPH_H_
4
5  #include <iostream>
6  #include <fstream>
7  #include <sstream>
8  #include <stdexcept>
9
10 #include <cstdlib>
11 #include <cstdio>
12 #include <cstring>
13 #include <cmath>
14
15 #include <map>
16 #include <set>
17 #include <queue>
18 #include <stack>
19 #include <algorithm>
20
21 using namespace std;
22 // No more than 48 nodes and 48^2 edges.
23 // Using more memory for sake of avoiding SIGSEGVs.
24 const int maxn = 60;
25
26 template <typename _T>
27 std::vector< std::vector< _T > > make_quad_matrix(int size, _T zero)
28 {
29     std::vector< std::vector< _T > > ret_vec;
30     ret_vec.clear();
31     for (int i = 0; i <= size; i++) {
32         std::vector< _T >   vec;
33         vec.clear();
34         for (int j = 0; j <= size; j++)
35             vec.push_back(zero);
36         ret_vec.push_back(vec);
37     }
38     return ret_vec;
39 }
40
41 template <typename _T>
42 std::vector< std::vector< _T > > make_quad_matrix(int n, int m, _T zero)
43 {
44     std::vector< std::vector< _T > > ret_vec;
```

```
    ret_vec.clear();
    for (int i = 0; i <= n; i++) {
        std::vector< _T >   vec;
        vec.clear();
        for (int j = 0; j <= m; j++)
            vec.push_back(zero);
        ret_vec.push_back(vec);
    }
    return ret_vec;
}

#endif
```

graph_solver/descat_wrkr.cpp

```

/*
 * This class processes the scatterization of nodes. Every single node is processed
 * and sent to this class for distinguishment. This makes the need of std::string
 * amidst the process unnecessary.
 */
class DescatterizationWorker
{
protected:
    std::map<std::string, int>  state_to_id_map;
    std::map<int, std::string>  id_to_state_map;
public:
    DescatterizationWorker(void)
    {
        std::string state_ids[49] = { "",
            "AL", "AR", "AZ", "CA", "CO", "CT", "DC", "DE", "FL", "GA",
            "IA", "ID", "IL", "IN", "KS", "KY", "LA", "MA", "MD", "ME",
            "MI", "MN", "MS", "MT", "NC", "ND", "NE", "NH", "NJ", "NM",
            "NV", "NY", "OH", "OK", "OR", "PA", "RI", "SC", "SD", "TN",
            "TX", "UT", "VA", "VT", "WA", "WI", "WV", "WY"};
        // Pushing in data with Complexity O(n*logn)
        for (int i = 1; i <= 48; i++) {
            this->state_to_id_map[state_ids[i]] = i;
            this->id_to_state_map[i] = state_ids[i];
        }
        return ;
    }
    std::string convert(int in) {
        if (id_to_state_map.find(in) == id_to_state_map.end())
            return "NH";
        std::string out = id_to_state_map[in];
        return out;
    }
    int convert(std::string in) {
        if (state_to_id_map.find(in) == state_to_id_map.end())
```

```
            return 30; // NH
        int out = state_to_id_map[in];
        return out;
    }
} descat_wrkr;
```

graph_solver/graph_base.cpp

```

/*
 * This is a base class for all solutions, which holds edge manipulation and
 * taxes, where taxes are defaulted to 0.0% (As we omit taxes in Part I).
 * All solution classes must inherit this class with public properties.
 */
class DeliverGraph
{
public:
    // Using adjacent tables to store edges. Guranteed O(m) time complexity
    // and memory complexity for inserting all edges.
    struct edge
    {
        int     u, v; // From node #u to node #v
        int     len; // Distance between node #u and #v, also the days required to ship.
        edge    *next; // Saving adjacency
    };
    edge        *edges[maxn];
    long double taxes[maxn];
    long double gsp[maxn];
    // Implements an edge insertion.
    void add_edge(int u, int v, int len)
    {
        edge *p = new edge;
        if (u == v) len = 1; // Guranteed...
        p->u = u; p->v = v; p->len = len;
        p->next = edges[u]; edges[u] = p;
        return ;
    }
    // Set state taxes, in long double.
    void set_state_tax(int state, long double tax)
    {
        this->taxes[state] = tax;
        return ;
    }
    // Set state GSP, in long double (though it is commonly int)
    void set_state_gsp(int state, long double GSP)
    {
        this->gsp[state] = GSP;
        return ;
    }
```

```
};
```

graph_solver/graph_base_mat.cpp

```

/*
 * This is a base class for all solutions, only its styles are applied as
   matrix
 * styles.
 */
class MatrixGraph
{
public:
    int                                n;
    std::vector< std::vector<int> > dist;
    std::vector<long double>           taxes;
    std::vector<long double>           gsp;
    // Implements an edge insertion.
    void add_edge(int u, int v, int len)
    {
        if (u == v) len = 1; // Guranteed...
        dist[u][v] = len;
        return ;
    }
    // Set state taxes, in long double.
    void set_state_tax(int state, long double tax)
    {
        this->taxes[state] = tax;
        return ;
    }
    // Set state GSP, in long double (though it is commonly int)
    void set_state_gsp(int state, long double GSP)
    {
        this->gsp[state] = GSP;
        return ;
    }
    void init(int n)
    {
        this->n = n;
        this->dist = make_quad_matrix(n, 0);
        for (int i = 0; i <= n; i++) {
            this->taxes.push_back(0.0);
            this->gsp.push_back(0.0);
        }
        return ;
    }
};
```

graph_solver/sol_np.cpp

```

#include "graph.h"
```

```
3
4  #define DO_NOT_USE_LOW_PRECISION_SOLVER
5
6  /*
7   * This is a graph inherited from DeliverGraph.
8   * Will process data in a non-polynomial time complexity and linear memory
9   * omplexity.
10  * Is faster than pure brute force because of the use of IDA*.
11  */
12 class NonPolynomialGraphSolver : public DeliverGraph
13 {
14 public:
15     int n;
16     // Not using iterative depth for Brute Force Search
17     bool deep_first_search(
18             long long int   status,
19             long long int   matrix[],
20             int             selecting,
21             int             select_begin)
22     {
23         // Reached target, checking if satisfies
24         if (selecting == 0)
25             return status == (2ll << n) - 2ll;
26         // Logically select all possible inheritors
27         for (int i = select_begin; i <= n; i++) {
28             bool tmp_res = deep_first_search(
29                 status | matrix[i], // Logic or
30                 matrix, // Copy and send status
31                 selecting - 1, // Lower depth by 1
32                 i + 1 // Select new beginning position
33             );
34             if (tmp_res == true)
35                 return true;
36         }
37         return false;
38     }
39     // Whether choosing #size rows for matrix can satisfy the operation.
40     bool satisfiable_matrix(long long int matrix[], int size)
41     {
42         // Evaluates through dfs.
43         return deep_first_search(
44             0, // Original status: zero must be considered
45             matrix, // Initial data
46             size, // To select #size
47             1); // Start from first point
48     }
49     // Evaluates final result, returns an integer.
50     // Has no optimization.
51     int evaluate(void)
52     {
53         this->n = 48;
```

```
54          // We can convert this graph into a matrix, for details please
    read
55          // the paper on how to build this matrix.
56          int matrix[maxn][maxn];
57          memset(matrix, 0, sizeof(matrix));
58          for (int i = 1; i <= n; i++)
59              for (edge *ep = edges[i]; ep; ep = ep->next) {
60                  if (ep->len <= 1)
61                      matrix[i][ep->v] = 1;
62                  else
63                      matrix[i][ep->v] = 0;
64              }
65          // Compressing matrix for better memory complexity
66          long long int   matrix_c[maxn];
67          // // Print matrix, temporarily
68          for (int i = 1; i <= n; cout << endl, i++)
69              for (int j = 1; j <= n; j++)
70                  cout << (matrix[i][j] ? "X" : "-") << ' ';
71          memset(matrix_c, 0, sizeof(matrix_c));
72          for (int i = 1; i <= n; i++) {
73              matrix_c[i] = 0;
74              for (int j = 1; j <= n; j++)
75                  matrix_c[i] ^= (long long int)matrix[i][j] << j;
76          }
77          // Running graph with brute force search, evaluating if 10 is
    satisfiable
78          // for common run.
79          // This is a binary search function.
80          int bin_left = 1, bin_mid = 0, bin_right = 10;
81          int res = n;
82          while (bin_left < bin_right) {
83              bin_mid = (bin_left + bin_right) / 2;
84              // printf("Processing depth %d...\n", bin_mid);
85              bool bin_res = satisfiable_matrix(matrix_c, bin_mid);
86              if (bin_res) {
87                  bin_right = bin_mid;
88                  res = bin_mid;
89              } else {
90                  bin_left = bin_mid + 1;
91              }
92          }
93          return res;
94      }
95  };
96
97  #ifndef DO_NOT_USE_LOW_PRECISION_SOLVER
98
99  void solution_1_low_precision(void)
100 {
101     std::ifstream   f_handle("../../data/dist_stats.txt");
102     int             cnt = 0;
```

```
    graph_solve.init(48);
    while (!f_handle.eof()) {
        std::string from, to; // From node #... to #...
        int         dist; // Cost days
        f_handle >> from >> to >> dist;
        // Catch exception
        if (from.length() < 1 || to.length() < 1)
            break;
        // Ignored states
        if (from == "AK" || from == "HI" || to == "AK" || to == "HI")
            continue;
        // Adding state numbers
        int from_idx = descat_wrkr.convert(from),
            to_idx = descat_wrkr.convert(to);
        // Inserting into graph.
        graph_solve.add_edge(from_idx, to_idx, dist);
    }
    // Emulating taxes and GSP.
    for (int i = 1; i <= 48; i++) {
        graph_solve.set_state_tax(i, 0.00);
        graph_solve.set_state_gsp(i, 40000.00);
    }
    // Evaluating result
    graph_solve.n = 48;
    std::pair<int, long double> result = graph_solve.evaluate();
    std::cout << result.first << ' ' << result.second << std::endl;
    return ;
}

#endif
```

graph_solver/sol_np_opt.cpp

```

#include "graph.h"

/*
 * This is a graph inherited from DeliverGraph.
 * Will process data in a near polynomial time complexity with linear
   memory
 * complexity. This class is highly optimized with branch cutting.
 * This is unimplemented.
 */
class NonPolynomialGraphSolverOptimized : public DeliverGraph
{
public:
};
```

graph_solver/sol_poly.cpp

```

#include "graph.h"
```

```

#define DO_NOT_USE_HIGH_PRECISION_SOLVER

class PolynomialGraphSolver : public MatrixGraph
{
public:
    int n;
    // Using Chvatal algorithm to solve set covering problem, with a matrix,
    // described in vectors.
    std::vector<int>    eval_res;
    std::pair<int, long double> solve_set_cover(std::vector< std::vector<int> > matrices)
    {
        // Final result
        int                 set_count = 0;
        long double         total_cost = 0.0;
        std::vector<int>    final_res;
        // The final array, to be queried and indexed
        std::set<int>                   S;
        std::map<int, std::set<int> >   mat;
        // Loading matrices to mat [ O(n^2) ]
        for (int i = 1; i <= n; i++) {
            std::set<int> st;
            for (int j = 1; j <= n; j++)
                if (matrices[i][j])
                    st.insert(j);
            mat[i] = st;
        }
        // // Printing mat?
        // for (int i = 1; i <= n; i++) {
        //     printf("#%d: ", i);
        //     for (std::set<int>::iterator j = mat[i].begin(); j != mat[i].end(); j++)
        //         printf("%d ", *j);
        //     printf("\n");
        // }
        // S is not full, until now. [ Worst O(n) ]
        while (S.size() < n && !mat.empty()) {
            // Best status to add into S.
            int         best_pos = 0;
            int         best_count = 0;
            long double best_cost = 1e100; // Literally infinite.
            // Iterate through all unchosen maps [ O(n) ]
            for (std::map<int, std::set<int> >::iterator i = mat.begin(); i != mat.end(); i++) {
                int             union_count = 0;
                int             point_pos = i->first;
                long double     point_cost = gsp[point_pos] * taxes[point_pos];
                std::set<int>&  st = i->second;
```

```
                // Iterate through set, and find existence [ O(n * logn) ]
                for (std::set<int>::iterator    j = st.begin(); j != st.
    end(); j++)
                    if (S.find(*j) == S.end())
                        union_count++;
                // Updating best status
                if (union_count > best_count) {
                    best_pos = point_pos;
                    best_count = union_count;
                    best_cost = point_cost;
                } else if (union_count == best_count
                        && point_cost < best_cost) {
                    best_pos = point_pos;
                    best_cost = point_cost;
                }
            }
            // Update S by inserting elements [ O(n * logn) ]
            std::set<int>&  upd_st = mat[best_pos];
            for (std::set<int>::iterator    j = upd_st.begin(); j !=
    upd_st.end(); j++)
                if (S.find(*j) == S.end())
                    S.insert(*j);
            // Removing best status from all matrices [ O(logn) ]
            mat.erase(mat.find(best_pos));
            // Update final result
            set_count++;
            total_cost += best_cost;
            final_res.push_back(best_pos);
        }
        sort(final_res.begin(), final_res.end());
        this->eval_res = final_res;
        return make_pair(
                set_count,
                total_cost);
    }
    // Evaluate function by calling a set covering solution algorithm.
    std::pair<int, long double> evaluate(void)
    {
        // Building matrix with graph
        std::vector< std::vector<int> > graph_vec =
            make_quad_matrix(n, 0);
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                if (dist[i][j] <= 1)
                    graph_vec[i][j] = 1;
        // Getting result...
        return solve_set_cover(graph_vec);
    }
};

#ifndef DO_NOT_USE_HIGH_PRECISION_SOLVER
```

```
98
99  void solution_high_precision_import_graph(void)
100 {
101     std::ifstream   f_handle("../../data/dist_stats.txt");
102     int             cnt = 0;
103     graph_solve.init(245);
104     while (!f_handle.eof()) {
105         int from, to, dist; // From node #.. to #.., cost days
106         f_handle >> from >> to >> dist;
107         // Catch exception
108         if (from < 1 || to < 1)
109             break;
110         // Ignored states already removed in Python.
111         // Does not require state numbers.
112         // Inserting into graph.
113         graph_solve.add_edge(from, to, dist);
114     }
115     std::cout << "... Built graph.\n";
116     f_handle.close();
117     return ;
118 }
119
120 void solution_high_precision_output_result(void)
121 {
122     std::cout << "==> Evaluating result...\n";
123     std::pair<int, long double> result = graph_solve.evaluate();
124     std::cout << "... Total count of warehouses: " << result.first << "\n"
125               << "... Total weight of taxes: " << result.second << "\n"
126               << "... Selected ID of locations: ";
127     for (unsigned int i = 0; i < graph_solve.eval_res.size(); i++)
128         std::cout << "  > #" << graph_solve.eval_res[i] << "\n";
129     return ;
130 }
131
132 void solution_1_high_precision(void)
133 {
134     solution_high_precision_import_graph();
135     // Emulating taxes and GSP.
136     for (int i = 1; i <= 245; i++) {
137         graph_solve.set_state_tax(i, 0.00);
138         graph_solve.set_state_gsp(i, 40000.00);
139     }
140     graph_solve.n = 245;
141     // Evaluating result.
142     solution_high_precision_output_result();
143     return ;
144 }
145
146 void solution_2_high_precision(void)
147 {
148     solution_high_precision_import_graph();
```

```
    // Importing taxes and GSP.
    for (int i = 1; i <= 242; i++) {
        graph_solve.set_state_tax(i, 0.00);
        graph_solve.set_state_gsp(i, 40000.00);
    }
    graph_solve.n = 242;
    // Evaluating result.
    solution_high_precision_output_result();
    return ;
}

#endif
```

graph_solver/sol_poly_highprec.cpp

```

#include "graph.h"

class HighResPolynomialGraphSolver : public MatrixGraph
{
public:
    // To choose m items from n subsets.
    int         n, m;
    long double proximity_ratio;
    // Using Chvatal algorithm to solve set covering problem, with a matrix,
    // described in vectors.
    std::vector<int>    eval_res;
    std::pair<int, long double> solve_set_cover(
            std::vector< std::vector<int> > matrices)
    {
        // Final result
        long double         set_count = 0;
        long double         total_cost = 0.0;
        std::vector<int>    final_res;
        std::cout << "... Working at coverage precision " << proximity_ratio * 100.0  / 0.958 << "%.\n";
        // The final array, to be queried and indexed
        std::set<int>                   S;
        std::map<int, std::set<int> >   mat;
        // Loading matrices to mat [ O(n^2) ]
        for (int i = 1; i <= n; i++) {
            std::set<int> st;
            for (int j = 1; j <= m; j++)
                if (matrices[i][j])
                    st.insert(j);
            mat[i] = st;
        }
        // S is not full, until now. [ Worst O(n) ]
        while (S.size() < m * proximity_ratio && !mat.empty()) {
            // Best status to add into S.
            long double best_pos = 0;
```

```
36            long double best_count = 0;
37            long double best_cost = 1e100; // Literally infinite.
38            long double epsilon = 1;
39
40            // Iterate through all unchosen maps [ O(n) ]
41            for (std::map<int, std::set<int> >::iterator i = mat.begin();
    i != mat.end(); i++) {
42                double             union_count = 0;
43                int             point_pos = i->first;
44                long double     point_cost = gsp[point_pos] * taxes[
    point_pos];
45                std::set<int>&  st = i->second;
46                // Iterate through set, and find existence [ O(n * logn) ]
47                for (std::set<int>::iterator    j = st.begin(); j != st.
    end(); j++)
48                    if (S.find(*j) == S.end())
49                        union_count++;
50                union_count /= sqrt(point_cost); //If for Part1,delete
    this line
51                // Updating best status
52                if (union_count > best_count) {
53                    best_pos = point_pos;
54                    best_count = union_count;
55                    best_cost = point_cost;
56                } else if (union_count >= best_count * epsilon
57                        && union_count <= best_count
58                        && point_cost <= best_cost) {
59                    best_pos = point_pos;
60                    best_cost = point_cost;
61                }
62            }
63            // Update S by inserting elements [ O(n * logn) ]
64            std::set<int>&  upd_st = mat[best_pos];
65            for (std::set<int>::iterator    j = upd_st.begin(); j !=
    upd_st.end(); j++)
66                if (S.find(*j) == S.end())
67                    S.insert(*j);
68            // Removing best status from all matrices [ O(logn) ]
69            mat.erase(mat.find(best_pos));
70            // Update final result
71            set_count++;
72            total_cost += best_cost;
73            final_res.push_back(best_pos);
74        }
75        sort(final_res.begin(), final_res.end());
76        this->eval_res = final_res;
77        return make_pair(
78                set_count,
79                total_cost);
80    }
81    // Evaluate function by calling a set covering solution algorithm.
```

```
    std::pair<int, long double> evaluate(
            long double proximity_ratio_)
    {
        // Building matrix with graph
        std::vector< std::vector<int> > graph_vec =
            make_quad_matrix(n, m, 0);
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= m; j++) {
                if (dist[i][j] == 1)
                    graph_vec[i][j] = 1;
                else
                    graph_vec[i][j] = 0;
            }
        // Getting result...
        this->proximity_ratio = proximity_ratio_;
        return solve_set_cover(graph_vec);
    }
} graph_solve;
```

graph_solver/main.cpp

```

#include "graph.h"

#include "descat_wrkr.cpp"
#include "graph_base.cpp"
#include "graph_base_mat.cpp"

#include "sol_np.cpp"
#include "sol_np_opt.cpp"
#include "sol_poly.cpp"
#include "sol_poly_highprec.cpp"

void solution_very_high_precision_import_graph(void)
{
    std::ifstream   f_handle("../../data/dist_stats.txt");
    int             cnt = 0;
    int             n, m;
    f_handle >> n >> m;
    graph_solve.init(max(n, m));
    graph_solve.n = n;
    graph_solve.m = m;
    std::cout << "... Building graph:\n";
    while (!f_handle.eof()) {
        int from, to, dist; // From node #.. to #.., cost days
        f_handle >> from >> to;
        // Catch exception
        if (from < 1 || to < 1)
            break;
        // Ignored states already removed in Python.
        // Does not require state numbers.
        // Inserting into graph.
```

```
32          graph_solve.add_edge(from, to, 1);
33      }
34      std::cout << "... Built graph.\n";
35      f_handle.close();
36      return ;
37  }
38
39  void solution_very_high_precision_output_result(
40          long double proximity_ratio)
41  {
42      std::cout << "==> Evaluating result...\n";
43      std::pair<int, long double> result = graph_solve.evaluate(
    proximity_ratio);
44      std::cout << "... Selected ID of locations: ";
45      for (unsigned int i = 0; i < graph_solve.eval_res.size(); i++)
46          std::cout << "  > #" << graph_solve.eval_res[i] << "\n";
47      std::cout << "... Total count of warehouses: " << result.first << "\n"
48                << "... Total weight of taxes: " << result.second << "\n";
49      return ;
50  }
51
52  void solution_1_very_high_precision(
53          long double proximity_ratio)
54  {
55      solution_very_high_precision_import_graph();
56      // Emulating taxes and GSP.
57      for (int i = 1; i <= graph_solve.n; i++) {
58          graph_solve.set_state_tax(i, 0.00);
59          graph_solve.set_state_gsp(i, 40000.00);
60      }
61      // Evaluating result.
62      solution_very_high_precision_output_result(proximity_ratio);
63      return ;
64  }
65
66  void solution_2_very_high_precision(
67          long double proximity_ratio)
68  {
69      solution_very_high_precision_import_graph();
70      // Importing taxes and GSP.
71      std::ifstream   f_handle("../../data/cities.csv");
72      for (int i = 1; i <= graph_solve.n; i++) {
73          std::string line;
74          char        line_ch[1024];
75          const char* split_ch = ",";
76          getline(f_handle, line);
77          // Copying line to line_ch
78          memset(line_ch, 0, sizeof(line_ch));
79          for (unsigned int j = 0; j < line.length(); j++)
80              line_ch[j] = line[j];
81          // Splitting with commas
```

```
82          std::vector<std::string>    splitted;
83          char*       split_wrkr = NULL;
84          split_wrkr = strtok(line_ch, split_ch);
85          while (split_wrkr != NULL) {
86              std::string tmp = split_wrkr;
87              splitted.push_back(tmp);
88              split_wrkr = strtok(NULL, split_ch);
89          }
90          // Getting splitted[6], splitted[8] in int and float, respectively
91          std::stringstream   sstr;
92          int                 cur_gsp = 0;
93          float               cur_tax = 0.0;
94          sstr << splitted[6];
95          sstr >> cur_gsp;
96          sstr.clear();
97          sstr << splitted[8];
98          sstr >> cur_tax;
99          // Setting gsp and taxes' data
100         graph_solve.set_state_tax(i, cur_gsp);
101         graph_solve.set_state_gsp(i, (long double)cur_tax);
102     }
103     solution_very_high_precision_output_result(proximity_ratio);
104     return ;
105 }
106
107 void solution_3_very_high_precision(
108         long double proximity_ratio,
109         long double importance)
110 {
111     solution_very_high_precision_import_graph();
112     // Importing taxes and GSP.
113     std::ifstream   f_handle("../../data/cities.csv");
114     for (int i = 1; i <= graph_solve.n; i++) {
115         std::string line;
116         char        line_ch[1024];
117         const char* split_ch = ",";
118         getline(f_handle, line);
119         // Copying line to line_ch
120         memset(line_ch, 0, sizeof(line_ch));
121         for (unsigned int j = 0; j < line.length(); j++)
122             line_ch[j] = line[j];
123         // Splitting with commas
124         std::vector<std::string>    splitted;
125         char*       split_wrkr = NULL;
126         split_wrkr = strtok(line_ch, split_ch);
127         while (split_wrkr != NULL) {
128             std::string tmp = split_wrkr;
129             splitted.push_back(tmp);
130             split_wrkr = strtok(NULL, split_ch);
131         }
132         // Getting splitted[6], splitted[8] in int and float, respectively
```

```
        std::stringstream   sstr;
        int                 cur_gsp = 0;
        float               cur_tax = 0.0;
        float               gar_tax = 0.0; // Garments' taxes
        sstr << splitted[6];
        sstr >> cur_gsp;
        sstr.clear();
        sstr << splitted[8];
        sstr >> cur_tax;
        sstr.clear();
        sstr << splitted[9];
        sstr >> gar_tax;
        // Setting gsp and taxes' data
        graph_solve.set_state_tax(i, cur_gsp);
        graph_solve.set_state_gsp(i,
            (long double)cur_tax * (1.0 - importance) +
            (long double)gar_tax * (importance));
    }
    solution_very_high_precision_output_result(proximity_ratio);
    return ;
}

/*
 * Main function, only takes care of input and output. Task number must be added
 * upon compile time, or defined inside at the header.
 */
int main(int argc, char** argv)
{
    // Defining which task to take, in a verbose way.
    int         task_index = 0;
    std::cout << " # Enter task number, in an integer: ";
    std::cin >> task_index;
    std::cout << " - Processing task number " << task_index << "...\n";
    // Processing task Part I
    if (task_index == 1) {
        long double proximity_ratio = 0.0;
        long double proximity_ratio2 = 0.0;
        while (true) {
            std::cout << " # Enter proximity ratio to calculate (0 for stop): ";
            std::cin >> proximity_ratio2;
            proximity_ratio = proximity_ratio2 * 0.958;
            if (proximity_ratio <= 0.0)
                break;
            solution_1_very_high_precision(proximity_ratio / 100.0);
        }
    } else if (task_index == 2) {
        long double proximity_ratio = 0.0;
        long double proximity_ratio2 = 0.0;
        while (true) {
```

```
            std::cout << " # Enter proximity ratio to calculate (0 for stop): ";
            std::cin >> proximity_ratio2;
            proximity_ratio = proximity_ratio2 * 0.958;
            if (proximity_ratio <= 0.0)
                break;
            solution_2_very_high_precision(proximity_ratio / 100.0);
        }
    } else if (task_index == 3) {
        long double proximity_ratio = 0.0;
        long double proximity_ratio2 = 0.0;
        long double importance = 0.0; // Default should be 3.3%.
        while (true) {
            std::cout << " # Enter proximity ratio to calculate (0 for stop): ";
            std::cin >> proximity_ratio;
            proximity_ratio = proximity_ratio2 * 0.958;
            if (proximity_ratio <= 0.0)
                break;
            std::cout << " # Enter garment tax importance: ";
            std::cin >> importance;
            solution_3_very_high_precision(proximity_ratio / 100.0, importance / 100.0);
        }
    } else {
        throw std::runtime_error("A task is required.");
    }
    return 0;
}
```