Optimisation Problem: Variant of capacitated clustering problem (VCCP)

Optimization Algorithms and Techniques (CS-357)
Project Report

Team Members:

Param Bansal (220001056)

Parth Deshmukh (220001057)

Under The Guidance Of:
Dr. Kapil Ahuja Sir

Contents:

INTRODUCTION

In the world of growing economic development, there is also a need of efficiently utilizing resources. A need of sustainable development motivates us to device an optimisation algorithm which can allocate resources among factories efficiently.

Problem Statement:

We are given n possible sites, s suppliers and some constraints. We have to open p factories among these n potential sites (n≥p). Each factory's potential location has its corresponding demand and each supplier has its maximum supply volume which can be transferred. Each factory's location has its corresponding opening cost and for each pair of supplier and factory has its corresponding supply cost. Each supplier can be associated with atmost one factory. We have to open p factories fulfilling their demands by associating one or more suppliers to it while at the same time minimising the total cost of opening the factories and the supply cost.

Our approach uses Lagrangian Relaxation to get an approximate optimal feasible solution.

The original solution is first relaxed using lagrange multipliers and then the dual problem (described in later sections) is then solved to obtain the optimal lagrange multipliers.

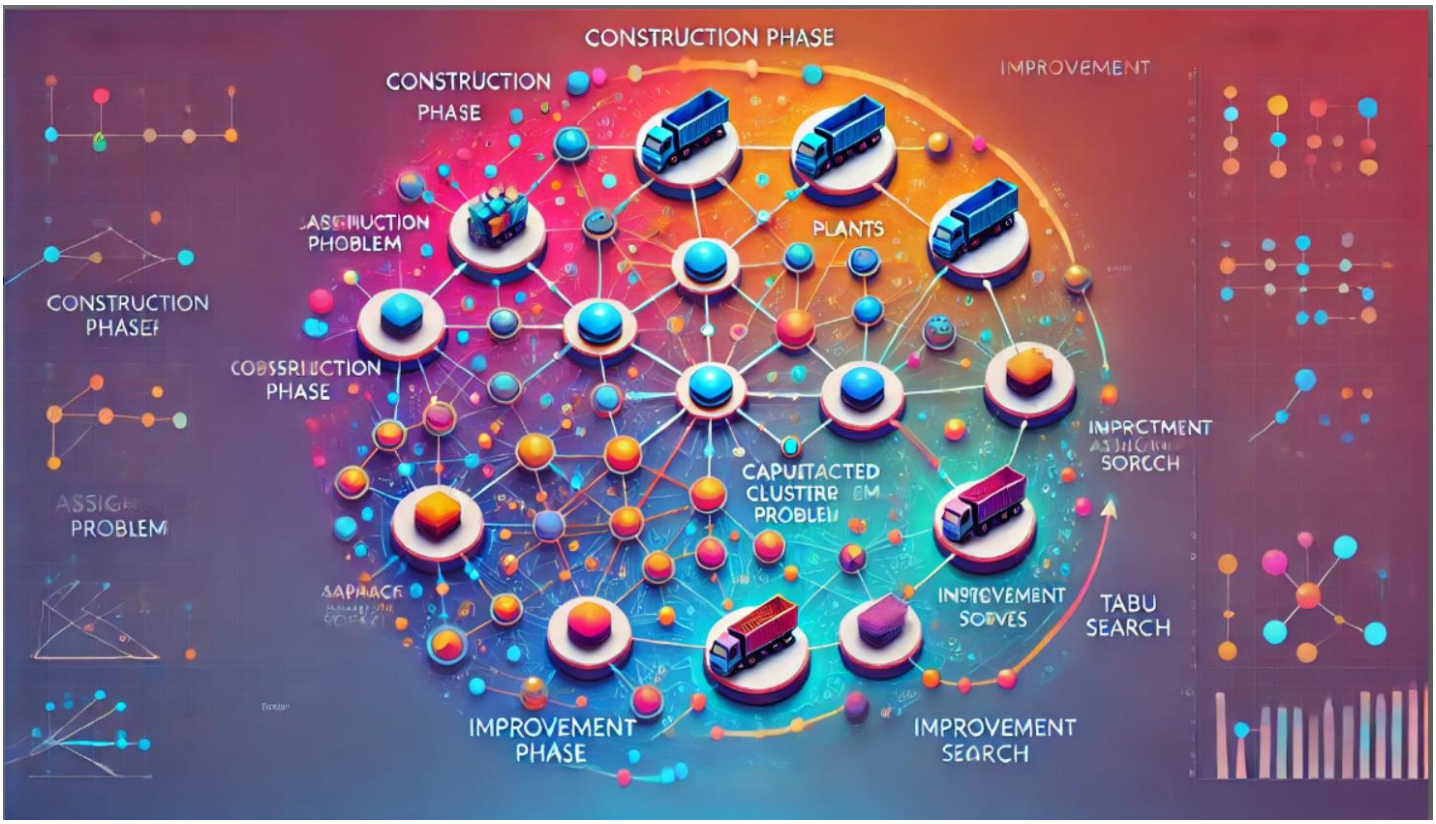
This dual problem is solved using:

1. Standard Subgradient method which is then further refined by
2. Subgradient Deflection method

This solution, obtained by lagrangian relaxation, is improved by two phases: first a construction phase which uses a greedy algorithm and an improvement phase which further refines the solution.

Thus, the algorithm which we describe consists of five phases:

1.Dual Problem Formulation, 2.Standard Subgradient,



PROBLEM FORMULATION

S: Set of suppliers

P: Set of potential sites

C_{ij}: Cost of moving supply from i^{th} supplier to j^{th} factory

SV_i: supply volume of i^{th} supplier

CR_j: Min capacity requirement of j^{th} factory

FP_j: Opening cost for j^{th} factory

Y_j: A binary variable which equals 1 if a factory is located at

site $j \in P$, otherwise 0

X_{ij}: A binary variable which equals 1 if i^{th} supplier is assigned to j^{th} factory, otherwise 0

$$\text{Model} \quad : z = \min \sum_{i \in S} \sum_{j \in P} c_{ij} \cdot x_{ij} + \sum_{j \in P} FP_j \cdot y_j \qquad (1)$$

$$\text{s.t.} \quad \sum_{j \in P} x_{ij} \leqslant 1, \quad \forall j \in S \qquad (2)$$

$$\sum_{i \in S} sv_i \cdot x_{ij} \geqslant CR_j \cdot y_j, \quad \forall j \in P \qquad (3)$$

$$\sum_{j \in P} y_j = p \qquad (4)$$

$$x_{ij},\ y_j \in \{0,1\}, \quad \forall i \in S,\ \forall j \in P \qquad (5)$$

Eq 1 minimises the cost of opening factories and assigning suppliers to them

Eq 2 make sures that each supplier is assigned to at most one factory

Eq 3 make sure that demand of every factory is satisfied

Eq 4 make sure that there are exactly p factories

WHAT IS THE NEED OF AN OPTIMAL ALGORITHM ?

Suppose there is only one potential site ($n=1$) and only one site to locate ($p=1$). There are 8 suppliers. Now we have to allocate these suppliers to satisfy the requirement of the plant & at the same time minimising the cost of allocation.
(Note: Since there is only one plant, no need to choose from plants)

Model :-

$$\hat{Z} = \min \sum_{i \in S} C_{ik}\, x_{ik} + FP_k$$

such that $\sum_{i \in S} s_{vi} \cdot x_{ik} \geq CR_k$

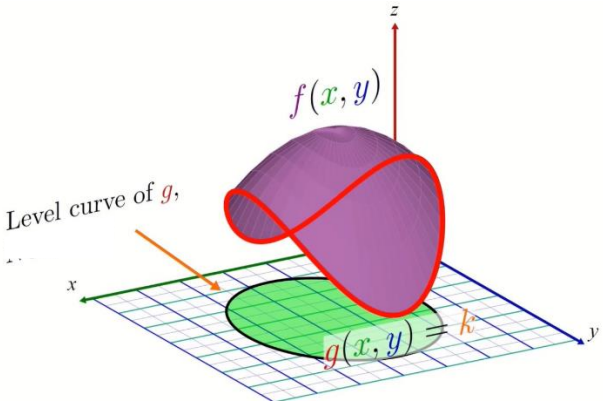
$x_{ik} \in \{0, 1\}$

It is a Knapsack problem in which we have to satisfy a threshold value while at the same time minimising the cost. It is proved that Knapsack problem is a NP Hard problem and has a time complexity which is pseudopolynomial. So for same input size and different threshold value the size of the problem can grow very fast. Thus, since an instance of our original problem is NP Hard, the original problem (which is an extension and more complex than the instance) is also NP Hard.

Thus there is a need to device an approximation algorithm for the problem which gives quite accurate result in feasible time.

LAGRANGIAN RELAXATION

Suppose there is a function f(x,y) which we have to minimise and x , y are constrained by equation g(x,y)=k. Now the minimiser can be at this boundary formed due to the equation g(x,y)<=k. Now this task can be quite difficult s we use Lagrange multipliers to approximate it.



We will relax the constraint g(x,y)<=k and our lagrangian equation becomes:

$$Z_{LR}(\lambda)=\min [\, f(x,y) + \lambda\{g(x,y)-k\} \,] \;, \lambda>=0$$

Note: If we go out of boundary then a penalty of $\lambda\{g(x,y)-k\}$ is incurred which is positive $\lambda>=0$. So if the solution of obtained of $Z_{LR}(\lambda)$ is feasible (i.e. $g(x,y)<=k$) it provides a lower bound on the original problem of min f(x,y)

i.e. $Z_{LR}(\lambda)<= \min f(x,y)$

Dual Problem:

To find the best lower bound, we need to solve the Lagragian Dual problem to determine the optimal lagrange multipliers.

Dual problem is defined as :

$$Z_D= \max Z_{LR}(\lambda) \;, \lambda>=0$$

APPLYING LAGRANGIAN RELAXATION ON OUR ORIGNAL PROBLEM

$$\text{Model} \quad : z = \min \sum_{i \in S} \sum_{j \in P} c_{ij} \cdot x_{ij} + \sum_{j \in P} FP_j \cdot y_j \qquad (1)$$

$$\text{s.t.} \quad \sum_{j \in P} x_{ij} \leqslant 1, \quad \forall j \in S \qquad (2)$$

$$\sum_{i \in S} sv_i \cdot x_{ij} \geqslant CR_j \cdot y_j, \quad \forall j \in P \qquad (3)$$

$$\sum_{j \in P} y_j = p \qquad (4)$$

$$x_{ij},\ y_j \in \{0,1\}, \quad \forall i \in S,\ \forall j \in P \qquad (5)$$

PHASE 1: Dual Problem Formulation

In our solution, we will relax the assignment constraints (2) and dualise them into the objective function (1) by introducing a set of non-negative lagrange multiplier (λ).

The lagragian relaxed problem Z_{LR} (λ) can be formulated as:

$$Z_{LR}(\lambda) = \min \sum_{i \in S} \sum_{j \in P} c_{ij} \cdot x_{ij} + \sum_{j \in P} FP_j \cdot y_j + \sum_{i \in S} \lambda_i \cdot \left(\sum_{j \in P} x_{ij} - 1 \right)$$

Z_{LR} (λ) provides a lower bound for our orignal problem for λ>=0 (same as proof done in previous page)

Thus our dual problem becomes :

$Z_D = \max Z_{LR}(\lambda)$, $\lambda >= 0$

PISINGER'S MINKNAP ALGORITHM

The Pisinger's MinKnap Algorithm is a crucial perquisite for our algorithm. This algorithm is named after David Pisinger. The $Z_{LR}(\lambda)$ is solved first taking $y_j=1$ for all j belonging to P. Then, $Z_{LR}(\lambda)$ can be decomposed into |P| independent 0-1 KnapSack problems such that for each j, the j_{th} KnapSack Problem becomes:

$$z_{KP}^{j}(\lambda) = \min \sum_{i \in S} c_{ij} \cdot x_{ij} + FP_j + \sum_{i \in S} \lambda_i \cdot x_{ij}$$

such that:

$$\sum_{i \in S} sv_i \cdot x_{ij} \geqslant CR_j \cdot y_j, \quad \forall j \in P$$

Since the normal Solution of KnapSack is pseudopolynomial, it is very inefficient. Thus we need a more faster approximation algorithm to solve this problem. For this, Pisinger's MinKnap is used.(Our code has implemented this algorithm)

Algorithm:

Each item has a **cost** (how much it "takes" to pick it) and a **value** (how much it "gives").

We're given a **threshold**: the minimum value you need to achieve.

Our objective is to select items so their combined value is at least the threshold, while keeping the total cost as low as possible.

First, maxvalue(maximum attainable value) is calculated by summing all the values of the suppliers.

Set Up the DP Table:

- dp[v] tells you the **minimum cost** to achieve an exact value vvv.
- Initially:
  - dp[0] = 0 (cost is zero to achieve zero value).
  - All other entries are set to infinity ,since those values are not achievable yet.

This table is the backbone of the algorithm.

Now, for each item, the algorithm checks how it can improve the table. If an item has a cost c[i] and a value v[i], it updates the table for all values v (from high to low) like this:

dp[v]=min(dp[v],dp[v−v[i]]+c[i])

If we can achieve a value of v−v[i] at some cost, then we can also achieve v by adding this item. The algorithm keeps the cheapest option.

By the end of this step, the table contains the minimum costs for every value up to the maximum.

The algorithm looks at all the entries in the table where v≥threshold. It picks the smallest cost among them. If all those entries are still infinity, it means there's no way to meet the threshold.

BackTracking:

Once the minimum cost is found, the algorithm retraces its steps to figure out which items contributed to this solution. It checks:

dp[v]=dp[v−v[i]]+c[i]

If this holds for an item, it means the item was included in the solution. The value v is reduced by v[i], and the process repeats until all items are identified.

The Pisinger algorithm ensures that every possible value is considered, and for each value, it keeps track of the cheapest way to achieve it. By working backward to find the items, it guarantees the solution is both optimal and traceable.

The Pisinger algorithm isn't just efficient; it's methodical and logical. It tackles the problem step by step, making sure nothing is overlooked. Whether we're minimizing costs in a project or optimizing resources in any scenario, it provides a reliable solution

This Pisinger algorithm is used to calcularte $Z^{j}_{KP}(\lambda)$ for each j and then these solutions are sorted in increasing order and first p solutions of these solutions are chosen. These p values construct the optimal solution of $Z_{LR}(\lambda)$.

Let π be a permutation of the numbers $1,2,\ldots,|P|$ such that

$$z_{KP}^{\pi(1)}(\lambda) \leqslant z_{KP}^{\pi(2)}(\lambda) \leqslant \cdots \leqslant z_{KP}^{\pi(|P|)}(\lambda)$$

$$z_{LR}(\lambda) = \sum_{i=1}^{p} z_{KP}^{\pi(i)}(\lambda) - \sum_{i \in S} \lambda_i$$

This step is done in each iteration of standard subgradient and subgradient deflection everytime changine the lagrange multipliers.

When the solution converge, the latest p subset obtained gives the set of sites which are actually chosen to be located.

Advancing to subsequent Phases

PHASE 2: Standard Subgradient

The subgradient method repeatedly solves the lagrangian relaxed problem and update the lagrange multipliers based on the subgradient $Z_{LR}(\lambda)$ at the current value of λ.

The multipliers for the next iteration λ_i^{k+1} are generated as follows:

$\lambda_i^{k+1} = \max(0, \lambda_i^k + t_k \cdot g_i^k) \; ; \forall i \in S$

where t_k is the step size and $g_i^k = \sum_{j \in P} x_{ij}^k - 1$ (the i^{th} component of the subgradient of $z_{LR}(\lambda)$ at $\lambda = \lambda^k$)

$t_k = \theta_k \left[Z_{UB} - Z_{LR}(\lambda^k) \right] / \|g^k\|^2$

Z_{UB}: upper bound

θ_k is parameter

$0 < \theta_k < 2$

ALGORITHM:

1. Initialise lagrange multipliers with 0 and upper bound with $+\infty$
2. Solve the relaxed problem using Pisinger MINKNAP algorithm (already discussed) and update the upper bound if necessary
3. Update t_K and lagrange multipliers using above mention procedure
4. Decrease the parameter θ if in a given number of iterations (in our code, 20) no improvement is observed.

Stopping Criteria:

Stop the iterations if any of the following criteria is met:

i. Required accuracy is achieved
ii. Lower bound is not improved for a given number of iterations
iii. Maximum number of iterations is reached

PHASE 3: SUBGRADIENT DEFLECTION

Subgradient Deflection can improve the convergence of subgradient algorithms.More refined lagrange multipliers are obtained after applying this phase. For this phase the lagrange multipliers obtained from phase 2 serve as an input. Here the parameter θ is initialised to 1.

This phase is same as phase 2 but the only difference is that this phase defines the moving direction as a combination of previous direction and current subgradient direction. Here, in updating the lagrange multiplers, instead of g^k, h^k is used where

$$h^k = \frac{g^k + 0.1h^{k-2} + 0.3h^{k-1}}{1.4}$$

$$h^{-2} \& h^{-1} = 0$$

Then, the Lagrange multipliers are updated as:

$$\lambda_i^{k+1} = \max(0, \lambda_i^k + t_k h_i^k)$$

PHASE 4: CONSTRUCTION PHASE

The optimal solution of the current lagrangian relaxed problem may be infeasible but it provides a crucial information for constructing a feasible solution: The locations of p open plants. Here it is possible that one supplier is assigned to more than one plant. To resolve this a greedy algorithm is used.

The process of assigning a supplier (i) to an open plant (j) is evaluated using a weight function f(i,j). For each open plant, the method calculates how much better the best option is compared to the second-best option. This difference helps prioritize which plants to supply first.

Key Points:

- For each plant (j), the desirability of assigning a supplier (i) is measured as:

$$\rho_j = \min_{s \neq i_j} f(s, j) - f(i_j, j)$$

- Here:
  - i_j is the supplier with the lowest weight (best option) for plant j.
  - s represents other suppliers being considered for the same plant.

Explanation:

- Plants are served in the order of how desirable they are, based on the difference between the best and second-best supplier weights.
- After deciding the locations of the plants and partially satisfying their demands, focus is only on the remaining suppliers and open plants.

Weight Function:

The weight f(i,j), which evaluates the suitability of a supplier for a plant, can be calculated as $f(i,j)=c_{ij}$

Greedy Algorithm:

Step 0: Set $y_j = y_j^k$ for all $j \in P$. If $\exists i \in S$, $l \in P$ such that $x_{il}^k = 1$ and $\sum_{j \in P} x_{ij}^k = 1$, then fix $x_{il} = 1$.

Step 1: Set $P = \{j | y_j = 1, j \in P\}$, $\widetilde{S} = \{i | \sum_{j \in P} x_{ij} = 0, i \in S\}$ and $\widetilde{CR}_j = CR_j - \sum_{i \in S \setminus \widetilde{S}} sv_i \cdot x_{ij}$, $\forall j \in \widetilde{P}$. Update $\widetilde{P} := \{j | \widetilde{CR}_j > 0, j \in \widetilde{P}\}$.

Step 2: Calculate $i_j = \arg\min_{i \in \widetilde{S}} f(i,j)$ and $\rho_j = \min_{s \in \widetilde{S},\ s \neq i_j} f(s,j) - f(i_j,j)$ for $j \in \widetilde{P}$.

Step 3: Calculate $\hat{j} = \arg\max_{j \in \widetilde{P}} \rho_j$, i.e., a supplier will be assigned to plant $\hat{j}$ next.

$$x_{i_{\hat{j}},\hat{j}} = 1$$

$$x_{i_j,\hat{j}} = 0 \quad \forall j \in \widetilde{P}, j \neq \hat{j}$$

$$\widetilde{CR}_{\hat{j}} := \widetilde{CR}_{\hat{j}} - sv_{i_j}$$

$$\widetilde{S} := \widetilde{S} \setminus \{i_j\}$$

Step 4: If $\widetilde{CR}_{\hat{j}} \leqslant 0$, then $\widetilde{P} := \widetilde{P} \setminus \{\hat{j}\}$. If $\widetilde{P} = \varnothing$: Stop, a feasible solution (x,y) of VCCP is found. If $\widetilde{P} \neq \varnothing$ and $\widetilde{S} = \varnothing$, then the greedy algorithm fails, Stop. Otherwise, return to Step 2.

In step 0, we set $X_{ij}=1$ only if supplier i supplies to no more than one factory.

In step 1, we are defining remaining suppliers and updating capacity

requirements of each factory

~P be a set of open plants

~S be a set of all suppliers not assign to any factory

$\sim CR_j$ is the updated capacity requirement of j^{th} factory

Now remove those factories from ~P whose capacity requirements are fully satisfied

In step 2, we are finding the supplier i ∈ ~S for each plant such that it minimizes the weight function f(i , j) where f(i , j)=C_{ij} and we are calling that supplier i_j.

Now we are calculating ρ_j which is the difference between the smallest and second smallest values of the weight function

In step 3,

$\hat{J}$ be the factory with highest value of ρ_j

Now assign i_j to $\hat{J}$ factory. Also modify capacity requirements according to it.

In step 4, we are removing the $\hat{J}$ factory from ~P if its capacity requirements are satisfied.

Stop if no open factories require suppliers otherwise jump to step 2

PHASE 5: IMPROVEMENT

It is possible that after applying the greedy algorithm, some suppliers are redundantly allocated to some sites(removing these sites still satisfy the requirement). Thus, in this phase, we will try to remove these redundancy most optimally to further minimise the cost.

For each chosen site, a knapsack problem is formulated. Let the chosen site be k.

Let Ω be the set of all suppliers allocated to k.

The problem is formulated as:

$$(KP) \quad \min \quad \sum_{i \in \Omega} c_{ik} S_i$$
$$\text{s.t.} \quad \sum_{i \in \Omega} sv_i \cdot s_i \geqslant CR_k$$
$$s_i \in \{0,1\} \quad \forall i \in \Omega$$

The new set obtained is the new allocation for the k^{th} site and thus a more feasible solution is found. Again, this problem can be solved using pisinger's MinKnap which is already discussed in previous pages.

Outcomes And Results:

```
PS C:\Users\ACER\Downloads> cd "c:\Users\ACER\Downloads\" ; if ($?) { g++ code.cpp -o code } ; if ($?) { .\code }
Enter Number Of Suppliers: 2
Enter Number Of potential Sites: 3
Enter supply volume for each supplier: 3 3
Enter requirement and cost of each potential site: 2 4 3 1 3 3
Enter cost of supplying for each supplier and potential site pair: 1 1 1 1 1 1
Enter number of facilities to open: 2

Minimum Cost after standard subgradient and subgradient deflection: 6
Allocation after standard subgradient and subgradient deflection
0 0
0 1
0 1
Minimum Cost after Greedy: 6
Allocation after greedy
0 0
1 0
0 1
FINAL MINIMUM COST AFTER Improvement Phase: 6
Allocation after improvement phase
0 0
1 0
0 1
```

From naïve algorithm (exact)

also the min cost is 6

```
PS C:\Users\ACER\Downloads> cd "c:\Users\ACER\Downloads\" ; if ($?) { g++ code.cpp -o code } ; if ($?) { .\code }
Enter Number Of Suppliers: 2
Enter Number Of potential Sites: 3
Enter supply volume for each supplier: 2 5
Enter requirement and cost of each potential site:
4 2
5 1
2 5
Enter cost of supplying for each supplier and potential site pair:
8 2 3
7 6 4
Enter number of facilities to open: 2

Minimum Cost after standard subgradient and subgradient deflection: 15
Allocation after standard subgradient and subgradient deflection
0 0
0 1
1 0
Allocation after greedy
0 0
0 1
1 0
FINAL MINIMUM COST AFTER Improvement Phase: 15
Allocation after improvement phase
0 0
0 1
1 0
```

From naïve

also, the

mincost is 15.

```
PS C:\Users\ACER\Downloads> cd "c:\Users\ACER\Downloads\" ; if ($?) { g++ code.cpp -o code } ; if ($?) { .\code }
Enter Number Of Suppliers: 8
Enter Number Of potential Sites: 4
Enter supply volume for each supplier:
10 9 5 6 4 9 7 5
Enter requirement and cost of each potential site:
12 5
9 5
6 7
16 2
Enter cost of supplying for each supplier and potential site pair:
1 2 3 4
5 6 7 8
9 1 2 3
4 5 6 7
8 9 1 2
3 4 5 6
7 8 9 1
2 3 4 5
Enter number of facilities to open: 3

Minimum Cost after standard subgradient and subgradient deflection: 22
Allocation after standard subgradient and subgradient deflection
1 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0
1 0 0 0 0 0 1 0
Minimum Cost after Greedy: 30
Allocation after greedy
1 0 0 0 0 0 0 1
0 0 1 0 0 1 0 0
0 0 0 0 0 0 0 0
0 0 0 1 1 0 1 0
FINAL MINIMUM COST AFTER Improvement Phase: 29
Allocation after improvement phase
1 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0
0 0 0 1 1 0 1 0
PS C:\Users\ACER\Downloads>
```

From Naïve

Algorithm,

Min cost is 25

```
PS C:\Users\ACER\Downloads> cd "c:\Users\ACER\Downloads\" ; if ($?) { g++ code.cpp -o code } ; if ($?) { .\code }
Enter Number Of Suppliers: 5
Enter Number Of potential Sites: 4
Enter supply volume for each supplier: 1 5 4 7 5
Enter requirement and cost of each potential site:
2 6
4 2
3 9
8 5
Enter cost of supplying for each supplier and potential site pair:
1 2 3 4
9 8 7 6
12 42 55 66
7 8 9 10
1 2 3 4
Enter number of facilities to open: 3

Minimum Cost after standard subgradient and subgradient deflection: 23
Allocation after standard subgradient and subgradient deflection
0 0 0 0 1
0 0 0 0 1
0 0 0 0 1
0 0 0 0 0
Minimum Cost after Greedy: 34
Allocation after greedy
1 0 0 0 1
0 0 0 1 0
0 1 0 0 0
0 0 0 0 0
FINAL MINIMUM COST AFTER Improvement Phase: 33
Allocation after improvement phase
0 0 0 0 1
0 0 0 1 0
0 1 0 0 0
0 0 0 0 0
PS C:\Users\ACER\Downloads>
```

In this case too, the naïve algorithm (exact solution) is 33. Thus our algorithm is very accurate and give quite accurate results.

Time Complexity of Our Algorithm

How much improvement we were able to do?

PISINGER'S MINKNAP:

Pisinger's MinKnap significantly improves the normal knapsack problem by giving an approximation algorithm within less time.

Let T=Threshold for the respective site (requirement)

And s be the number of suppliers.

The time complexity of normal (without Pisinger) knapsack is O(T*s)

Pisinger significantly improves it with:

1. Best Case: O(s)
2. Worst Case: O(T*s)
3. Average Case: O(s*log(t))

Although the worst case is same as normal knapsack, but in real world in most of the cases the performance is significantly better.

Apart from this, the space complexity required by normal solution is O(T*s) whereas in pisinger it is just O(T+s).

SubGradient and Subgradient Deflection:

Time complexity : O(maxiter*s*logT).

Here, maxiter is the maximum number of iterations permitted.

Greedy:

Worst case Time Complexity: $O(|N|^2.s)$

Average Case Time Complexity: $O(|N|.s.\log|N|)$

where $|N|$= total number of potential sites and s is total number of suppliers

Improvement Phase:

Time Complexity:

Worst Case: $O(P*s*T)$

Average Case: $O(P*s*\log T)$

Where P is the number of sites required and s is number of suppliers and T is the maximum threshold of a plant.

Overall Time Complexity:

Worst Case Time Complexity: $O(maxiter*s*T + P*s*T + |N|^2.s)$

Average Case Time Complexity: $O(maxiter*s*\log T + P*s*\log T + |N|.s.\log|N|)$

Thus, a problem which in general would have required exponential time complexity is solved very efficiently by this algorithm.

Conclusion And Summary:

Many state of the art tool have been developed but no one could solve the above problem with more than 400 suppliers. This algorithm is applied on several test cases and can solve those tests quite efficiently even for very large datasets.

Thus, this overall approach can be used more often in future to utilise resources in more efficient manners

.

Further Improvements:

The solution we obtained can be further refined by tabu search (which is a part of metaheuristics) and even a more optimal solution can be achieved.

Acknowledgements:

We are immensely grateful towards Dr. Kapil Ahuja Sir who taught us different optimisation techniques and gave us real life journals for us to explore the actual application of these optimisation techniques. His guidance fueled us with motivation and enthusiasm to give our best to this project.

We are glad that CS 357 course enabled us to study different optimisation techniques and applying them in a real life project. We are thankful to Dr. Kapil Ahuja Sir for this project as we have learnt a lot from it.

References:

1.Zhen Yang a, Haoxun Chen a, Feng Chu's: A Lagrangian relaxation approach for a large scale new variant of capacitated clustering problem

2.David Pisinger's: Where are the hard Knapsack Problems?

3.Serpentine Integral: Understanding Lagrange multipliers visually video

\

Code:

```
#include<bits/stdc++.h>
using namespace std;

int number_of_suppliers;
vector<int>supply_volume(number_of_suppliers);
int number_of_potential_sites;
vector<vector<int>>potential_sites(number_of_potential_sites
, vector<int>(2, 0));
vector<vector<int>>cost_of_supplying(number_of_suppliers,
vector<int>(number_of_potential_sites));
vector<int>supply_taken(number_of_suppliers, 0);
int p;

//allocations[i][j]-> if(jth supplier is assigned to ith
plant )then allocations[i][j]=1; else 0
//potential_sites[i][0] contain capacity requirement of
potential_sites[i]
//supply_volume[i] contains volume supplied by ith supplier
//cost_of_supplying[i][j] contains cost of supplying ith
supplier to jth site


pair<double, vector<int>>
pisingerMinimizeCost(vector<double>& costs, vector<int>&
values, int threshold) {
    int n = costs.size();
    int maxValue = 0;

    // Calculate the maximum possible value
    for (int value : values) {
        maxValue += value;
    }
```

```
    // Initialize DP table with high cost (INT_MAX for
unachievable states)
    vector<double> dp(maxValue + 1,
numeric_limits<double>::infinity());
    dp[0] = 0.0;

    // Fill the DP table
    for (int i = 0; i < n; i++) {
        for (int v = maxValue; v >= values[i]; v--) {
            if (dp[v - values[i]] !=
numeric_limits<double>::infinity()) {
                dp[v] = min((double)dp[v], dp[v - values[i]]
+ costs[i]);
            }
        }
    }

    // Find the minimum cost to satisfy the threshold value
    double minCost = numeric_limits<double>::infinity();
    int targetValue = -1;
    for (int v = threshold; v <= maxValue; v++) {
        if (dp[v] < minCost) {
            minCost = dp[v];
            targetValue = v;
        }
    }

    // Handle no feasible solution
    if (minCost == numeric_limits<double>::infinity()) {
        cout << "No feasible solution exists.\n";
        return { numeric_limits<double>::infinity(), {} };
    }

    // Trace back to find selected items
    vector<int> selected_items;
    int remainingValue = targetValue;
    for (int i = n - 1; i >= 0 && remainingValue > 0; i--) {
```

```
        if (remainingValue >= values[i] &&
dp[remainingValue] == dp[remainingValue - values[i]] +
costs[i]) {
            selected_items.push_back(i);
            remainingValue -= values[i];
        }
    }

    return { minCost, selected_items };
}

pair<double, vector<int>>
pisingerMinimizeCost2(vector<double>& costs, vector<int>&
values, int threshold) {
    int n = costs.size();
    int maxValue = 0;

    // Calculate the maximum possible value
    for (int value : values) {
        maxValue += value;
    }

    // Initialize DP table with high cost (INT_MAX for
unachievable states)
    vector<double> dp(maxValue + 1,
numeric_limits<double>::infinity());
    dp[0] = 0.0;

    // Fill the DP table
    for (int i = 0; i < n; i++) {
        for (int v = maxValue; v >= values[i]; v--) {
            if (dp[v - values[i]] !=
numeric_limits<double>::infinity()) {
                dp[v] = min((double)dp[v], dp[v - values[i]]
+ costs[i]);
            }
        }
```

```
    }

    // Find the minimum cost to satisfy the threshold value
    double minCost = numeric_limits<double>::infinity();
    int targetValue = -1;
    for (int v = threshold; v <= maxValue; v++) {
        if (dp[v] < minCost) {
            minCost = dp[v];
            targetValue = v;
        }
    }

    // Handle no feasible solution
    if (minCost == numeric_limits<double>::infinity()) {
        cout << "No feasible solution exists.\n";
        return { numeric_limits<double>::infinity(), {} };
    }

    // Trace back to find selected items
    vector<int> selected_items(costs.size());
    int remainingValue = targetValue;
    for (int i = n - 1; i >= 0 && remainingValue > 0; i--) {
        if (remainingValue >= values[i] &&
dp[remainingValue] == dp[remainingValue - values[i]] +
costs[i]) {
            selected_items[i] = 1;
            remainingValue -= values[i];
        }
    }

    return { minCost, selected_items };
}


vector<vector<int>> improvement_phase(vector<vector<int>>&
allocations) {
```

```
    // Step 0: Identify active plants based on initial
allocations.
    vector<int> plants_location(number_of_potential_sites,
0);
    for (int i = 0; i < allocations.size(); i++) {
        bool found = false;
        for (int j = 0; j < allocations[i].size(); j++) {
            if (allocations[i][j]) {
                found = true;
                break;
            }
        }
        if (found) plants_location[i] = 1;
    }

    double total_cost = 0;
    vector<vector<int>> allocations2;

    // Iterate through all plants to minimize cost and
satisfy requirements.
    for (int i = 0; i < allocations.size(); i++) {
        if (plants_location[i]) {
            total_cost += potential_sites[i][1];  // Add
fixed cost for active plant.

            // Prepare supplier data for the current plant.
            vector<double> costs;
            vector<int> values;
            vector<int> selected_j(number_of_suppliers, 0);
            for (int j = 0; j < allocations[i].size(); j++)
{
                if (allocations[i][j]) {
                    costs.push_back((double)cost_of_supplyin
g[j][i]);
                    values.push_back(supply_volume[j]);
                }
            }
```

```
            // Minimize cost using Pisinger's method for
knapsack problem.
            pair<double, vector<int>> improvement =
pisingerMinimizeCost2(costs, values, potential_sites[i][0]);
            int index = 0;
            for (int j = 0; j < number_of_suppliers; j++) {
                if (allocations[i][j]) {
                    selected_j[j] =
improvement.second[index++];
                }
            }

            total_cost += improvement.first;
            allocations2.push_back(selected_j);
        } else {
            // If plant is inactive, assign no suppliers.
            vector<int> temp(number_of_suppliers, 0);
            allocations2.push_back(temp);
        }
    }

    // Output the final minimized cost after improvement
phase.
    cout << "FINAL MINIMUM COST AFTER Improvement Phase: "
<< total_cost << endl;

    return allocations2;
}

vector<vector<int>> greedy(vector<vector<int>>& allocations)
{
    // Step 0: Identify which plants (potential sites) are
currently active based on initial allocations.
    vector<int> plants_location(number_of_potential_sites,
0);
    for (int i = 0; i < allocations.size(); i++) {
```

```
        bool found = false;
        for (int j = 0; j < allocations[i].size(); j++) {
            if (allocations[i][j]) {
                found = true;
                break;
            }
        }
        if (found) plants_location[i] = 1;
    }

    // Initialize a new allocation matrix and mark suppliers
already used.
    vector<vector<int>> allocations2(allocations.size(),
vector<int>(allocations[0].size(), 0));
    vector<int> suppliers_used(number_of_suppliers, 0);

    // Assign suppliers with a single connection directly to
the active plant.
    for (int i = 0; i < number_of_suppliers; i++) {
        int sum = 0, last_j = -1;
        for (int j = 0; j < number_of_potential_sites; j++)
{
            if (allocations[j][i]) {
                sum += allocations[j][i];
                last_j = j;
            }
            if (sum > 1) break;
        }
        if (sum == 1) {
            allocations2[last_j][i] = 1;
            suppliers_used[i] = 1;
        }
    }

    // Step 1: Filter active plants and unused suppliers.
    set<int> current_plants;
    for (int j = 0; j < number_of_potential_sites; j++) {
```

```
        if (plants_location[j]) {
            current_plants.insert(j);
        }
    }

    set<int> suppliers_not_used;
    for (int j = 0; j < number_of_suppliers; j++) {
        if (suppliers_used[j] == 0) {
            suppliers_not_used.insert(j);
        }
    }

    // Adjust plant capacities based on assigned suppliers.
    vector<int> to_erase;
    vector<vector<int>> potential_sites2 = potential_sites;
    for (auto& it : current_plants) {
        for (int i = 0; i < number_of_suppliers; i++) {
            if (allocations2[it][i]) {
                potential_sites2[it][0] -= supply_volume[i];
            }
        }
        if (potential_sites2[it][0] <= 0) {
            to_erase.push_back(it);
        }
    }
    for (int site : to_erase) {
        current_plants.erase(site);
    }

    // Step 2: Assign unused suppliers to minimize cost
iteratively.
    while (!current_plants.empty()) {
        vector<int> ij(number_of_potential_sites, -1);  //
Best supplier for each plant.
        for (auto& j : current_plants) {
            int min_cost = INT_MAX;
            for (auto& i : suppliers_not_used) {
```

```
                if (min_cost > cost_of_supplying[i][j]) {
                    min_cost = cost_of_supplying[i][j];
                    ij[j] = i;
                }
            }
        }

        vector<int> rho_j(number_of_potential_sites, 0);
        int j_cap = -1, max_rho = INT_MIN;
        for (auto& it : current_plants) {
            rho_j[it] = cost_of_supplying[ij[it]][it];  // 
Compute marginal cost.
            if (max_rho < rho_j[it]) {
                max_rho = rho_j[it];
                j_cap = it;
            }
        }

        // Update allocation and capacity for the selected 
plant.
        allocations2[j_cap][ij[j_cap]] = 1;
        potential_sites2[j_cap][0] -= 
supply_volume[ij[j_cap]];
        suppliers_not_used.erase(ij[j_cap]);

        // Remove plant if its capacity is fully utilized.
        if (potential_sites2[j_cap][0] <= 0) {
            current_plants.erase(j_cap);
        }

        // Break conditions: no plants left or no suppliers 
left.
        if (current_plants.empty() || 
suppliers_not_used.empty()) {
            break;
        }
    }
```

```
    // Compute and return final allocation.
    return allocations2;
}


pair<double, vector<pair<int, vector<int>>>>
subgradient_deflection_Method(
    function<pair<double, vector<pair<int,
vector<int>>>>(vector<double>)> objective, // Objective
function
    function<vector<double>(vector<double>, pair<double,
vector<pair<int, vector<int>>>>)> subgradient, //
Subgradient function
    vector<double> x0, // Initial guess
    double tolerance, // Convergence tolerance
    int max_iter, // Maximum number of iterations
    double upper_bound,
    double& f_best,
    vector<pair<int, vector<int>>>& ans_second
) {

    vector<double> x = x0;
    // Best objective value found
    double prev_f_val = numeric_limits<double>::infinity();
// Best objective value found

    int iter = 0;
    double theta = 1;




    prev_f_val = upper_bound;
    vector<double>prev_h(number_of_suppliers);
    vector<double>h(number_of_suppliers);
    while (iter < max_iter) {
        // Compute objective value
```

```
        pair<double, vector<pair<int, vector<int>>>>obj =
objective(x);
        double f_val = obj.first;
        if (f_val > f_best) {
            f_best = f_val;
            ans_second = obj.second;
        }
        f_best = max(f_best, f_val);

        // Check convergence
        if (abs(f_val - prev_f_val) / max(1.0, abs(f_val)) <
tolerance) {
            // cout << "Converged in " << iter << "
iterations." << endl;
            return { f_best,ans_second };
        }

        // Compute subgradient
        vector<double> g = subgradient(x, obj);
        for (int i = 0;i < number_of_suppliers;i++) {
            h[i] = (g[i] + 0.1 * prev_h[i] + 0.3 * h[i]) /
1.4;
        }
        double norm = 0.0;
        for (int i = 0;i < number_of_suppliers;i++)norm +=
g[i] * g[i];
        double st = theta * (upper_bound - f_val) / norm;
        // Update solution
        for (size_t i = 0; i < x.size(); i++) {
            x[i] = max((double)0, x[i] + st * h[i]);

        }

        if (iter % 20 == 0)theta /= 2;
        prev_f_val = f_val;
        upper_bound = min(upper_bound, f_val);
        iter++;
```

```
        if (iter == max_iter) {
            // cout << "Reached maximum iterations without
full convergence." << endl;

            return { f_best,ans_second };
        }
    }
    return { f_best,{{-1,{-1}} } };
}



pair<double, vector<pair<int, vector<int>>>>
subgradientMethod(
    function<pair<double, vector<pair<int,
vector<int>>>>(vector<double>)> objective, // Objective
function
    function<vector<double>(vector<double>, pair<double,
vector<pair<int, vector<int>>>>)> subgradient, //
Subgradient function
    vector<double> x0, // Initial guess

    double tolerance, // Convergence tolerance
    int max_iter // Maximum number of iterations
) {
    vector<pair<int, vector<int>>> ans_second;
    vector<double> x = x0;
    double f_best = numeric_limits<double>::infinity(); //
Best objective value found
    double prev_f_val = numeric_limits<double>::infinity();
// Best objective value found
    double upper_bound = 0.0;
    int iter = 0;
    double theta = 1;

    for (int i = 0;i < number_of_potential_sites;i++) {
```

```
        for (int j = 0;j <
number_of_suppliers;j++)upper_bound +=
cost_of_supplying[j][i];
        upper_bound += potential_sites[i][1];
    }

    f_best = 0;
    prev_f_val = upper_bound;
    while (iter < max_iter) {
        // Compute objective value
        pair<double, vector<pair<int, vector<int>>>>obj =
objective(x);
        double f_val = obj.first;
        if (f_val > f_best) {
            f_best = f_val;
            ans_second = obj.second;
        }
        f_best = max(f_best, f_val);

        // Check convergence
        if (abs(f_val - prev_f_val) / max(1.0, abs(f_val)) <
tolerance) {
            //cout << "Converged in " << iter << "
iterations." << endl;
            return subgradient_deflection_Method(objective,
subgradient, x, tolerance, max_iter, upper_bound, f_best,
ans_second);

        }

        // Compute subgradient
        vector<double> g = subgradient(x, obj);

        double norm = 0.0;
        for (int i = 0;i < number_of_suppliers;i++)norm +=
g[i] * g[i];
        double st = theta * (upper_bound - f_val) / norm;
```

```
        // Update solution
        for (size_t i = 0; i < x.size(); i++) {
            x[i] = max((double)0, x[i] + st * g[i]);
        }

        if (iter % 20 == 0)theta /= 2;
        prev_f_val = f_val;
        upper_bound = min(upper_bound, f_val);
        iter++;
        if (iter == max_iter) {
            //cout << "Reached maximum iterations without
full convergence." << endl;
            return subgradient_deflection_Method(objective,
subgradient, x, tolerance, max_iter, upper_bound, f_best,
ans_second);

        }
    }
    return { f_best,{{-1,{-1}} } };
}


// Objective function
pair<double, vector<pair<int, vector<int>>>> objective(const
vector<double>& x) {
    vector <pair< pair<double, vector<int>>,
int>>knapsacks(number_of_potential_sites);
    for (int i = 0;i < number_of_potential_sites;i++) {
        vector<double>costs(number_of_suppliers);
        for (int j = 0;j < number_of_suppliers;j++) {
            costs[j] = cost_of_supplying[j][i] + x[j];
        }
        knapsacks[i].first = pisingerMinimizeCost(costs,
supply_volume, potential_sites[i][0]);
        knapsacks[i].first.first += potential_sites[i][1];
        knapsacks[i].second = i;
    }
```

```
    sort(knapsacks.begin(), knapsacks.end());



    pair<double, vector<pair<int, vector<int>>>>ans;//zlr,
site id, supplies to site

    ans.first = -accumulate(x.begin(), x.end(), 0.0);
    for (int i = 0;i < p;i++) {
        ans.first += knapsacks[i].first.first;
        ans.second.push_back({
knapsacks[i].second,knapsacks[i].first.second });
        if (knapsacks[i].first.first ==
numeric_limits<double>::infinity()) {
            ans.first = numeric_limits<double>::infinity();
        }
    }

    return ans;

}

// Subgradient function
vector<double> subgradient(const vector<double>& x, const
pair<double, vector<pair<int, vector<int>>>>& obj) {
    vector<double> g(x.size(), -1);
    for (size_t i = 0; i < p; i++) {
        for (size_t j = 0;j <
obj.second[i].second.size();j++) {
            g[obj.second[i].second[j]]++;
        }
    }
    return g;
}
```

```
void lagrange_relaxation(
    function<pair<double, vector<pair<int,
vector<int>>>>(vector<double>)> objective, // Objective
function
    function<vector<double>(vector<double>, pair<double,
vector<pair<int, vector<int>>>>)> subgradient, //
Subgradient function

    double tolerance, // Convergence tolerance
    int max_iter
) {
    for (auto& x : supply_taken)x = 0;
    vector<double>x0(number_of_suppliers, 0);
    pair<double, vector<pair<int, vector<int>>>>ans =
subgradientMethod(objective, subgradient, x0, tolerance,
10);
    cout << endl;
    cout << "Minimum Cost after standard subgradient and
subgradient deflection: " << ans.first << endl;
    vector<vector<int>>sites_and_suppliers(number_of_potenti
al_sites, vector<int>(number_of_suppliers));
    for (auto& x : ans.second) {
        for (auto& y : x.second) {
            sites_and_suppliers[x.first][y] = 1;
        }
    }
    cout << "Allocation after standard subgradient and
subgradient deflection" << endl;
    for (auto& x : sites_and_suppliers) {
        for (auto& y : x)cout << y << ' ';
        cout << endl;
    }
    vector<vector<int>>new_sites_and_suppliers =
greedy(sites_and_suppliers);
    cout << "Allocation after greedy" << endl;
    for (auto& x : new_sites_and_suppliers) {
        for (auto& y : x)cout << y << ' ';
```

```
        cout << endl;
    }

    vector<vector<int>>new_sites_and_suppliers2 =
improvement_phase(new_sites_and_suppliers);
    cout << "Allocation after improvement phase" << endl;
    for (auto& x : new_sites_and_suppliers2) {
        for (auto& y : x)cout << y << ' ';
        cout << endl;
    }


}



int main() {


    cout << "Enter Number Of Suppliers: ";
    cin >> number_of_suppliers;
    cout << "Enter Number Of potential Sites: ";
    cin >> number_of_potential_sites;
    supply_volume.resize(number_of_suppliers);
    potential_sites.resize(number_of_potential_sites,
vector<int>(2));
    cost_of_supplying.resize(number_of_suppliers,
vector<int>(number_of_potential_sites));
    supply_taken.resize(number_of_suppliers);


    cout << "Enter supply volume for each supplier: ";
    for (auto& x : supply_volume)cin >> x;
    cout << "Enter requirement and cost of each potential
site: " << endl;
    for (auto& x : potential_sites)cin >> x[0] >> x[1];
    cout << "Enter cost of supplying for each supplier and
potential site pair: " << endl;
```

```
    for (auto& x : cost_of_supplying)
        for (auto& y : x)cin >> y;
    cout << "Enter number of facilities to open: ";
    cin >> p;

    int tolerance = 0.01;
    lagrange_relaxation(objective, subgradient, tolerance,
1000);

    return 0;
}
```