MULTIOBJECTIVE MULTIDIMENSIONAL 0-1 KNAPSACK PROBLEM

Kelly Chang
1007000932

Ernesto Huergo
1006686178

Jonah Ernest
1007065275

Victoria Piroian
1006882090

1 INTRODUCTION

The multiobjective multidimensional 0-1 knapsack problem (m-KP-MOP) determines which items (i = 1,...,n) should be included to maximize the total utility across J objective functions without exceeding maximum capacities b_k, across k=1,...,m knapsack constraints. Each item has its own utility c_i, and weight a_{ik}. It can be formulated as an integer program as follows:

$$\begin{aligned} \text{Maximize } & \left(\sum_{i=1}^{n} c_{1i}x_i, \sum_{i=1}^{n} c_{2i}x_i, \ldots, \sum_{i=1}^{n} c_{Ji}x_i \right) \\ \text{subject to } & \sum_{i=1}^{n} a_{ik}x_i \leq b_k, \quad k = 1, \ldots, m \\ & x_i \in \{0, 1\}, \quad i = 1, \ldots, n \end{aligned}$$

In this model, the binary decision variable x_i is 1 if item i is selected, and 0 otherwise. Unlike the 0-1 knapsack problem, the optimal value is a set called the nondominated frontier or Pareto frontier, where $z \in \mathbb{R}^J$, and has the properties that it must be feasible and there does not exist any other feasible solution which dominates z. An element of this nondominated frontier is a nondominated point (NDP). Methods of finding NDPs vary, however some of the most common ones include using a weighted-sum method which solves for a single objective that is the combination of the multiobjective problem (the Supernal Method) or using lexicographic optimization to minimize one objective at a time, using the objectives from the solved problems as constraints in the next (such as the Rectangle Division Method). Both of these algorithms will be implemented and compared to the Brute-force enumeration method.

This report will begin by describing the need for each of the Brute-Force Enumeration, Rectangle Division, and Supernal methods. The implementation of each of these algorithms will then be explained, including the pseudocode and actual implementation in Python, as well as the theoretical complexity of each one. The process for creating instances of m-KP-MOP's will be explored to test the algorithms. With these new instances, the efficiency of each algorithm can be compared while varying the parameters of the m-KP-MOP such as the number of items, knapsack constraints, and objective functions. Next, improvements to each algorithm are proposed and then implemented. The theoretical complexity of these changes is then analyzed and compared to the complexity of the original methods, while also discussing any memory issues associated with the various algorithms and larger instances.

2 ALGORITHMS

The following section details three methods to solve the 0-1 multi-objective knapsack problem: Brute-Force Enumeration Method, Rectangle Division Method, and Supernal Method.

2.1 BRUTE-FORCE ENUMERATION METHOD (BF)

The following algorithm implements the most naive algorithm to find non-dominated points. The algorithm enumerates all feasible solutions in the decision space, finds their objective images, removes duplicate images, computes the non-dominated images, and returns the non-dominated images in a list.

2.1.1 BF ALGORITHM

Algorithm 1 Brute-Force Enumeration Method

Inputs: filename ▷ method = 1
Outputs: NDF

$n, b, c, a \leftarrow read\_input(filename)$ ▷ Read input file
$foundNDPs \leftarrow []$ ▷ Initialize list of found NDPs
$feas \leftarrow []$ ▷ Initialize list of feasible points
$arr \leftarrow np.empty(n)$ ▷ Initialize solution list of size n
$generateAllBinaryCombos(n, arr, 0, a, b, feas)$ ▷ Enumerate all feasible solutions
for $solution\ in\ feas$ **do**
    $Calculate\ z = objectiveimage$
    $foundNDPs.append(z)$
end for
$removeDuplicates(foundNDPs)$
$removeDominated(foundNDPs)$
return $foundNDPs$

Algorithm 2 Remove Dominated Points

Inputs: array
Outputs: 2d array with dominated points removed

for $z1\ in\ array$ **do**
    $dominated \leftarrow []$
    for $z2\ in\ array$ **do**
        if $z1 \neq z2\ (element - wise)$ **then**
            if $z1 \leq z2\ (element - wise)$ **then**
                $dominated.append(z2)$
            end if
        end if
    end for
    for $k\ in\ dominated$ **do**
        $array.remove(k)$ ▷ Remove dominated points
    end for
end for

2.1.2 BF COMPLEXITY

The Big-O complexity of the Brute-Force algorithm is dominated by the complexity of enumerating all feasible solutions. As such, we are interested in the complexity of the generateAllBinaryCombos function. GenerateAllBinaryCombos is a backtracking algorithm that makes use of recursion to enumerate all permutations and return those which satisfy the problem constraints. In the 0-1 Knapsack Problem, every item or variable, x, in the solution, can be either '0' or '1'. The number of ways to create an array of length n where each item can only be 0 or 1 is simply 2^n. This is because for each element in the array, there are two possible values it can take, and since there are n elements in each array, the total number of possible arrays is 2*2*2*...*2 (n times), which equals 2^n. Thus, the complexity of enumerating all solutions is exponential.

For each solution, we check whether it is feasible by comparing the dot product of the solution and the constraint coefficients with the right hand side of the constraint, for each constraint. Given each

solution has shape 1xn, and the coefficient matrix also has shape 1xn, the complexity of the dot-product for one constraint is $O(n^2)$. To perform this operation for each constraint would increase the complexity by a factor. Thus, the complexity for checking feasibility is polynomial.

The complexity to find feasible images is to perform the dot product operation between each feasible solution and the coefficients of each objective function. The dot product between a single feasible solution and the coefficients of a single objective function has complexity $O(n^2)$. To perform this operation for a single solution and each objective function increases the complexity by a factor of J, for each objective function. Furthermore, to perform this operation for each feasible solution increases the complexity by a factor, depending on the number of feasible solutions. Thus, in general, the complexity of finding feasible images is also polynomial.

Finally, we must remove dominated points. In the worst case scenario, all feasible points are NDPs. Thus, we would compare each feasible point with each other and no points will be dominated and removed from the list of foundNDPs. Thus, for each NDP, we must compare it to all other NDPs. Therefore, this step has a worst case quadratic complexity of $O(n^2)$.

Since exponential time dominates both polynomial and quadratic time, the task of enumerating all feasible solutions dominates any other task giving Brute-Force Enumeration an overall Big-O complexity of $O(2^n)$.

2.2 Rectangle Division Method (RDM)

The Rectangle Divison Method (RDM) searches for non-dominated points (NDPs) by continuously bisecting the feasible space into regions, i.e rectangles. The NDPs are found by respectively solving the two possible lexicographic optimizations within the given regions which are later used as new north-western and south-eastern points to further bisect the space.

2.2.1 RDM ALGORITHM

Algorithm 3 Rectangle Division Method

Inputs: filename ▷ method = 2
Outputs: NDF

1: $n, b, c, a \leftarrow read\_input(filename)$ ▷ Read input file
2: $foundNDPs \leftarrow []$ ▷ Initialize list of found NDPs
3: $model \leftarrow get\_model(n, m = len(b), J = 2, c, a, b)$ ▷ Define Gurobi model
4: $z\_nw \leftarrow lexmin(J, model, first\_obj = 1)$ ▷ Solve lex min for most NW point
5: $foundNDPs.append(z\_nw)$
6: $z\_se \leftarrow lexmin(J, model, first\_obj = 2)$ ▷ Solve lex min for most SE point
7: $foundNDPs.append(z\_se)$
8: $rectangles\_list \leftarrow [[z\_nw, z\_se]]$
9: **while** $len(rectangles\_list) \neq 0$ **do**
10: $R \leftarrow rectangles\_list[0]$
11: $Remove\ R\ from\ rectangles\_list$
12: $z1 \leftarrow R[0]$
13: $z2 \leftarrow R[1]$
14: $R2 \leftarrow [[z1[0], (z1[1] + z2[1])/2], z2]$ ▷ Bisect R to create bottom rectangle R2
15: $Solve\ lex\ min\ in\ R2\ for\ z\_hat$ ▷ Look for next most West point
16: **if** $z\_hat\ is\ not\ None$ **then**
17: **if** $z\_hat \neq z2$ **then**
18: $foundNDPs.append(z\_hat)$ ▷ A new NDP is found
19: $rectangles\_list.append([z\_hat, z2])$ ▷ A new rectangle is found
20: **end if**
21: $R3 \leftarrow [[z1, [z\_hat[0] - \epsilon$,(z1[1]+z2[1])/2]] ▷ Create refined top rectangle R3
22: $Solve\ lex\ min\ in\ R3\ for\ z\_squiggly$ ▷ Look for next most South point
23: **if** $z\_squiggly \neq None$ **then**
24: **if** $z\_squiggly \neq z1$ **then**
25: $foundNDPs.append(z\_squiggly)$ ▷ A new NDP is found
26: $rectangles\_list.append([z1, z\_squiggly])$ ▷ A new rectangle is found
27: **end if**
28: **end if**
29: **end if**
30: **end while**
31: **return** $foundNDPs$

2.2.2 RDM COMPLEXITY

The complexity of the RDM algorithm can be inspected in two separate parts. The first part involves the complexity of the initialization of the first rectangle which happens in lines 1 to 8. In this part, two lexmin operations are executed to find the north-western and south-eastern points of the first rectangle. Let G be the complextity of the lexmin algorithm; hence, the complexity of the first part of the RDM algorithm is 2G.

The second part of the complexity of the algorithm involves the operations inside the while loop (lines 9 to 30). Here, a rectangle R from the list containing the found rectangles is selected and bisected into R2 and R3 to find a maximum of 2 NDPs -z_hat and z_squiggly. There are two elementary operations needed to obtain R2 (an addition and a division) and 3 to obtain R3 (a subtraction, an addition and a division). Furthermore, two lexmin operations are executed to find z_hat and z_squiggly. Therefore, the complexity per rectangle R of the this part the algorithm is 2G + 5. The amount of rectangles that are added to the list of rectangles determines how many times the operations inside the while loop are executed, and the amount of those rectangles depends on the number of NDPs that exist. Let m represent this number. The following table shows the relationship between these values. Let O_i = 2G be the complexity of the first part of the algorithm and O_r = 2G + 5 be the complexity of the second part of the algorithm. Note that a new rectangle is generated when a new NDP is found, which means rectangles only start to be added to the rectangle list if the number

of existing NDPs is greater than 3 as the first 2 NDPs are found during the initialization part of the algorithm.

Table 1: Complexity of RDM with respect to the number of NDPs

# NDPs	**Complexity**
1	$O_i + O_r$
2	$O_i + O_r$
3	$O_i + 2O_r$
4	$O_i + 3O_r$
5	$O_i + 4O_r$
.	.
.	.
.	.
m	O_i + (m-1)O_r

The complexity of the RDM algorithm can thus be generalized as

O_m = O_i + (m-1)O_r = 2G + (m-1)(2G + 5) = 2Gm + 5m - 5

In the worst case, the initial rectangle would be a square because there could be an NDP on every integer point along its diagonal -that is, an NDP could be found for every vertical and horizontal integer axis contained in the initial region and every feasible image would lie along this diagonal. The algorithm would have to then generate a rectangle for every feasible point, i.e. for every point on the diagonal. In the worse case, this may be equivalent to the total number of feasible images, 2^n, as discussed in Section 2.1.2. The big-O complexity of the NDP could then be said to be O(2^n) in the worst case. Yet, this is highly unlikely to happen since there is usually not a high density of feasible points concentrated along a line; and based on the plot shown in Figure 7, the algorithm seems to have more of a quadratic or polynomial computational time.

2.3 SUPERNAL METHOD (SPM)

Unlike the other algorithms, SPM is applicable for MOP programs with more than two objectives. This method splits the criterion space into smaller regions, each of which is defined by its northeast corner. Unlike the RDM, this method uses a weighted-sum single-objective optimization problem as seen below:

$$\min_{z \in Z} \sum_{j=1}^{J} \lambda_j z_j = \min_{z \in Z} \lambda^T z$$

Once this NDP is found, if it is within any of the other regions, then remove those regions and split into J new regions, while checking for and removing dominated regions if J is greater than 2. This process is repeated until all the NDPs are found, and there are no more regions to search.

2.3.1 SPM ALGORITHM

Algorithm 4 Supernal Method

Inputs: filename, method ▷ method = 3
Outputs: run_time, foundNDPs, number of regions searched

Create an empty list called $FoundNDPs$
Initialize couter for number of regions searched ▷ Set num of regions searched = 0
Read passed input file : **filename** ▷ Retrieve values for n, a, b, and c
Create an empty list supernal_point
for $i \leftarrow 0$ **to** $J - 1$ **do**
    $supernal\_point \leftarrow 0$ *append* ▷ Initialize supernal point as worst possible solution
end for
Initialize J *as length of* **c** *and* M *as length of* **b**
Initialize regions list and set it equal to supernal_point
$reg \leftarrow regions[0]$ ▷ Select first region in the regions list
Randomly generate lambdas $\leftarrow$ *use np.random* ▷ Normalize generated values
Create min : $\lambda^T z$ **model** $\leftarrow$ GET_WEIGHTED_SUM_MODEL$(c, a, b, n, M, J, reg, \lambda)$
while *regions is not empty* **do**
    $Num\ of\ regions\ searched += 1$
    $reg \leftarrow regions[0]$ ▷ Select first region in the regions list
    for $i \leftarrow 0$ **to** $J - 1$, $r \in reg[i \leftarrow 0, J - 1]$ **do**
        model.$\_z[i].ub \leftarrow r$ ▷ Update UB of the model
    end for
    Update and optimize **model** ▷ Solve weighted sum problem in current region
    if model status $= feasible$ **then**
        $z^* \leftarrow$ GET_SUPERNAL_Z$(n, c,$ **model**$)$ ▷ Get supernal point
        Add z^* *to* $foundNDPs$ ▷ Add supernal point to list of found NPDs
        $regions\_temp \leftarrow []$ ▷ Initialize temporary list of new regions
        $regions\_to\_remove \leftarrow []$ ▷ Initialize list of regions to remove
        for *region* **in** *regions* **do**
            $count \leftarrow 0$ ▷ Set counter to 0
            for $i \leftarrow 0$ **to** $J - 1$ **do**
                if $z^* \leq region$; *for each dimension in* z^* **then**
                    $count += 1$
                end if
            end for
            if $count = J$ **then**
                Add region to list : regions_to_remove
                $new\_reg \leftarrow []$ ▷ Initialize list of new regions
                for $i \leftarrow 0$ **to** $J - 1$ **do**
                    Set $new\_reg \leftarrow region$ ▷ Assign region as the new region
                    for $j \leftarrow 0$ **to** $J - 1$ **do**
                        if $i = j$ **then**
                            Set $new\_reg[j] = z^* - 1$ ▷ Update dimension j of the new region
                        end if
                    end for
                    Add new_reg *to* : regions_temp ▷ Append new_reg to list of new regions
                end for
            end if
        end for
        for $r \in$ regions_to_remove **do**
            Remove r $from$ *regions* ▷ Update regions by removing all obsolete regions
        end for
        for $r \in$ regions_temp **do**
            Add r *to regions* ▷ Update regions by appending all new regions
        end for
        if $J \geq 3$ **then**
            REMOVEDOMINATED$(regions)$ ▷ Remove dominated regions
        end if
    else
        Remove reg $from$ *regions* ▷ reg is infeasible
    end if
end while

Algorithm 5 Get Weighted Sum Model

Inputs: C, A, B, n, M, J, region, lam
Outputs: model

Initialize the **model** ▷ Using gurobipy gp.Model
Initialize **x** *using* model.addVars() ▷ x is a binary decision variable with n dimensions
$z \leftarrow []$ ▷ Initialize variable for objectives
for $i \leftarrow 0$ **to** $J - 1$ **do**
    $z \leftarrow$ model.addVar() ▷ Add J integer variables
end for
for $i \leftarrow 0$ **to** $J - 1$ **do**
    $z[i] = sum(c_{ij} * x_j)\ for\ j \in [0, n-1]$ ▷ Add objective constraints to **model**
end for
for $i \leftarrow 0$ **to** $J - 1$ **do**
    $sum(A_{ij} * x_j) \leq B[i]\ for\ j \in [0, n-1]$ ▷ Add constraints to **model**
end for
for $i \leftarrow 0$ **to** $M - 1$ **do**
    $Set\ UB\ of\ z[i] = region[i]$ ▷ Add UB constraint to **model**
    $Set\ LB\ of\ z[i] = -\infty$ ▷ Add LB constraint to **model**
end for
for $i \leftarrow 0$ **to** $J - 1$ **do**
    $Set\ objective\ to : sum(lam_i * z_i) \leftarrow minimize$ ▷ Set **model** objective function
end for
$return$ **model**

Algorithm 6 Get Supernal Z

Inputs: n, C, **model**
Outputs: z

$Convert\ model\ output\ into\ optimal\ z - value$
solution-z $= x - var[i]\ for\ i \in [0, n-1]$
$return$ **solutoin-z**

2.3.2 SPM COMPLEXITY

The complexity of the SPM can be found by breaking down the algorithm into smaller tasks, the first being the complexity of calculating and randomly generating the lambdas. To generate a single lambda, we take the ratio of that lambda to the sum of all the weightings, which requires one multiplication and J-1 addition, which equates to a complexity of J per lambda. Since there are J lambdas that are created this step has a complexity of J^2.

Next, the weighted model is created using the helper function get_weighted_sum_model. To determine the complexity of calling this helper function, it is split into the complexity of creating the objective function and the constraints. Setting up the objective function requires J terms, where each term is the product of the z_i with its lambda, which requires J-1 additions and J multiplications. Further, each z_i requires one comparison, and is the sum of the n items multiplied by their utility. Therefore creating the objective function in this helper function requires $2Jn + 2J - 1$ elementary operations. There are M knapsack constraints, where each one requires one comparison, and is the sum of the n decision variables with the weight of each item. So the constraints require $M(1 + n + (n - 1)) = 2Mn$ elementary operations. Therefore, get_weighted_sum_model requires $2Jn + 2J - 1 + 2Mn$ elementary operations.

After initializing the model, there is a while loop that iterates until there are no more regions to search. Let R be the total number of regions needed to find all the NDPs to the problem. This means that there are R iterations in the while loop, so all elementary operations within the loop occur R times. Assume updating the model does not contribute to the complexity. Similarly to RDM, H will be the complexity of the gurobipy optimization, so add H to the complexity for each iteration. The feasibility check for this model requires one comparison. Next, finding z_* requires the get_supernal_z helper function, which converts the solution using n multiplications, n-1 additions, J times, for a complexity of $J(n + (n - 1)) = J(2n - 1)$.

The for loop through all the current regions is tricky to incorporate into the complexity analysis, however, in the worst case, this loop will occur R times for each of the R total regions being searched. In order to check for the condition in line 38 of the pseudocode, a counter is used. Therefore there is one check and one addition for each of the J dimensions, for a total of $2J + 1$ elementary operations to check if z_* is within the region. If it is within the region, then new regions are created, each of which requires JxJ comparisons to see if the indices, and one subtraction for the new region, due to the double nested loops through J, totaling to $J^2(1 + 1) = 2J^2$ elementary operations. Lastly, if J¿2, then removing dominated points requires one comparison plus n^2 elementary operations (see BF complexity in section 2.1.2 for the complexity of removing dominated points).

Therefore we can sum over all the elementary operations required in each of the small tasks discussed above while distributing the R regions throughout the while and for loops, providing the worst-case total number of elementary operations in SPM to be:

$J^2 + 2Jn + 2J - 1 + 2Mn + R(H + 1 + J(2n - 1) + R(2J + 1 + 2J^2) + 1 + n^2)$
$= RH + 2R + 2RJn - RJ + 2R^2J + R^2 + 2R^2J^2 + Rn^2 + J^2 + 2Jn + 2J - 1 + 2Mn$

Therefore, the big-O complexity of the supernal method in the worst case is n^2, assuming that H has a complexity of less than this. Note that the simplex method is used in H, which has a worst-case complexity of 2^n, making it the dominating term, and therefore SPM would have a big-O complexity of 2^n. However, for most problems, the simplex method has quadratic complexity.

3 DATA GENERATION

The following section of the report will describe the data generation process implemented in order to account test the algorithms with a variety of cases, as well as what the cases were and how they were used.

3.1 PROCESS

Various m-KP-MOP instances of different sizes were required to conduct tests and run the Brute Force, Rectangle, and Supernal Algorithms. In order to create these testing sets, a random instance generator was implemented that created integers for all parameters b,a, and c. These were used by

the algorithms mentioned above and were passed to them by being written to a file and given to the **SolveKnapsack** function. Given the values for n (number of items), m (number of constraints), and J (number of objectives), as well as a predetermined upper bound for the random number generation (40), the weights (a) and costs (c) for each item n were generated.

An array of integers $\in [1, 40]$ of size ***J by n*** were generated and used as the costs of each item n, with respect to each objective J. These values were then multiplied by -1 in order to be able to differentiate these values from other randomly generated number in the output file. A second array of integers $\in [1, 40]$ of size ***m by n*** were generated and used as the weights of each item n, with respect to each constraint m. The final parameter to be generated, b, was first initialized as an array of -1s of size m, and was later filled with the correct values. The upper bound of each constrain m was determined by $b_k = max(a_{1k}, a_{2k}, ..., a_{nk}, 0.5 * \sum_{i=1}^{n} a_{ik})$, $for\ k \in [1, m]$. The values for b were selected according to these conditions, after which a file was created and all values n,c,a, and b were written into it in this order. It is important to note that a random seed of 7 was implemented using *np.random.seed(7)*, through the numpy library, in order to keep the randomly generated numbers consistent when running the code several times.

Algorithm 7 Random Instance Generation

Inputs: n, m, J, u
Outputs: file_rig

$Generate$ **c** : $random\ uniform\ integers \in [1, u]\ , of\ size\ (J, n)$ ▷ Represents cost of items $\in [1, n]$ for each objective $\in [1, J]$ (multiply by -1)
$Generate$ **a** : $random\ uniform\ integers \in [1, u]\ , of\ size\ (m, n)$ ▷ Represents weights of items $\in [1, n]$ for each constraint $\in [1, m]$
$Initialize$ **b** : $[-1] * m$ ▷ Constraint array
for $k \leftarrow 0$ **to** $length\ of\ b - 1$ **do**
    $Set\ num = 0$
    for $i \leftarrow 0$ **to** $n - 1$ **do**
        $num+ = 0.5 * a_{ki}$ ▷ Sum over weights for each objective k
    end for
    $Take\ the\ ceiling\ of$: num
    $Set\ b[k] = max(a[k], num)$ ▷ Create UB for Knapsack constraints
end for
$Create\ file\ to\ write\ to$: **filename**
$Write\ n, b, c, a\ to$ **filename** $through$ **file_rig**
$Close\ file$
$return$ **file_rig**

3.2 Used Instances

Using the above described function, various random instances of the n, b, c, and a parameters were generated and written to files to use as tests for the three algorithms. The file names describe the n, m, and J values; these are the number that are followed by the letters. Files **inst_n5_m1_j2** , and **inst_n5_m1_j3** were generated to test the original Brute Force, Rectangle, and Supernal methods. The file **inst_n375_m2_j2** was also used to test the original algorithms for larger parameter sets. The instances in the files **inst_n5_m2_j2**, and **inst_n10_m4_j3** were generated and used to test the improved versions of the algorithms as well as the original versions. The NDPs for these instances were first found using the Brute Force algorithm, then compared with the output of all other algorithms to ensure their correctness.

4 Algorithm Comparison

4.1 Comparing the Efficiency of the Algorithms

The efficiencies of the Brute Force, Rectangle, and Supernal methods were assessed by running each algorithm on various randomly generated instances where either values of n or m were changed within a given range. This allowed for the visualization of each methods' performance over differing

amounts of items, constraints, and objectives. Random instances were generated using the **random_instance_generation** function to generate values of n, b, c, and a. These values were then passed to the **SolveKnapsack** function where all three algorithms were run, and the run-time was returned which was then plotted on a graph.

The first variation in generation parameters was J = 2, n = 18, u = 40, and m $\in [1, 15]$. The random instance generation function was used to create 15 different instances.The range of n and value of m was chosen as such because the Brute Force algorithm encountered memory issues and long run-times when running for lager values. This resulted in the inability to compare the algorithms over a larger set of parameters. Each algorithm was run for the given set of n, b, c, and a, and their run-times for each set were graphed in order to allow for the comparison between algorithms. Figure 1 shows the run-time for all three algorithms across m $\in [1, 15]$. It can be seen that the Brute Force algorithm has a much higher and increasing run-time compared to the Rectangle and Supernal methods. This is due to the exponential complexity of the Brute Force algorithm causing the longer run-times. The Supernal and Rectangle methods have a significantly lower run-times for the same sets of generated parameters. In addition, there are many variable values that affect the resulting complexities of these two values methods which may be the reason for their faster run-times. If the given set of parameters did not result in a worst case complexity for these method, the run-times would be significantly less than that of Brute Force which is demonstrated in the figure below.

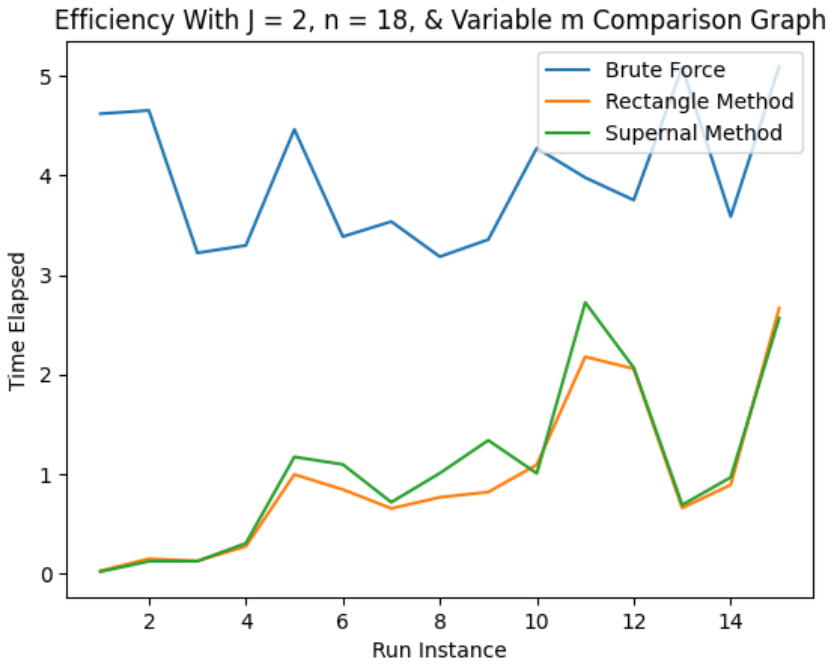

Figure 1: Efficiency of Brute Force, Rectangle, and Supernal methods with random instances generated using J = 2, n = 18, and a varying m

Parameters J = 2, m = 5, u = 40, and n $\in [1, 20]$ were then used to generate 20 different instances. Figure 2 shows the run-time for all three algorithms across n $\in [1, 20]$. It can be seen that all three algorithms have relatively similar run times until n = 17, when the run-time for the Brute Force rises exponentially. This once again can be supported by the exponential complexity of this method. The Supernal and Rectangle methods have similar run-times to the Brute Force method until n = 17, when the run-times rise slightly due to the increase in possible NPDs.

Next, parameters J = 3, m = 5, u = 40, and n $\in [1, 16]$ were used to generate 16 different instances of n, b, a, and c sets. These values were only tested on the Supernal algorithm as it is the only one that can solve a multi-objective optimisation problem for more than 3 objective functions. Figure 3 shows the run-time for the Supernal algorithm across n $\in [1, 16]$. It can be seen that the run-times are very small for lower values of n, but when n $\geq$ 14, the run time increases as a result of the variable

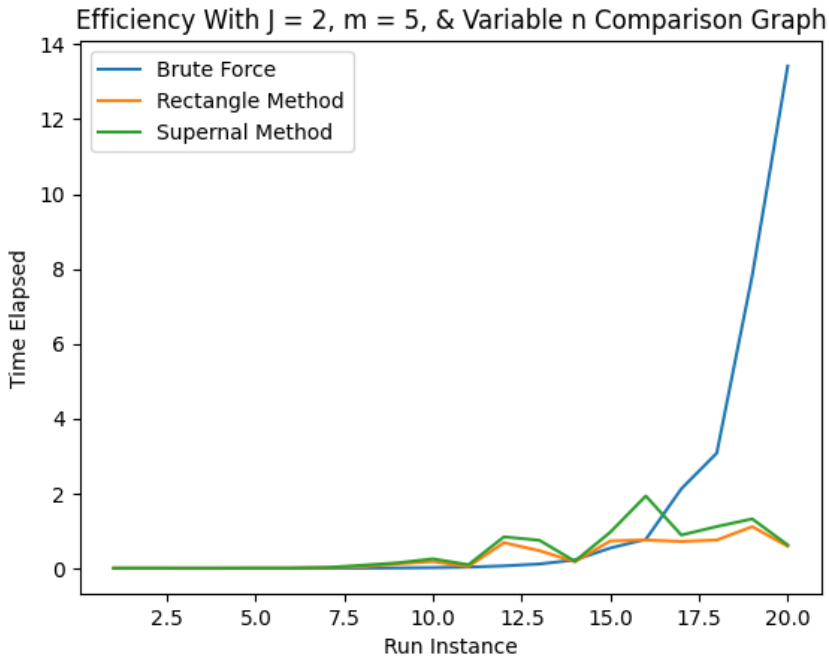

Figure 2: Efficiency of Brute Force, Rectangle, and Supernal methods with random instances generated using J = 2, m = 5, and a varying n

parameters, such as the number of regions to search, that contribute to the increase in complexity of the algorithm.

Similarly, parameters of J = 3, n = 13, u = 40, and m $\in [1, 25]$ were used to generate 25 different instances of n, b, a, and c sets. These values were once again only tested on the Supernal algorithm for the same reasons as previously stated. Figure 4 shows the run-time for the Supernal algorithm across m $\in [1, 25]$. It can be seen that the run-times are generally increasing for respectively increasing values of m. Although this trend is noticeable, the run-time values are still extremely varied for different amounts of m constraints. This suggests that the time complexity and the hard part of the algorithm comes from the number of items and objectives in the problem rather than the number of constraints.

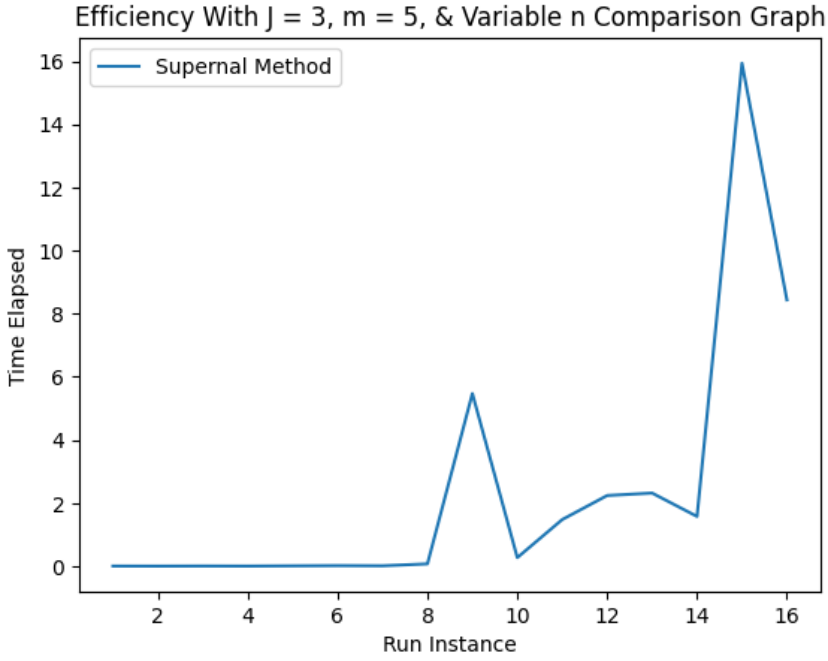

Figure 3: Efficiency of Supernal method with random instances generated using J = 3, m = 5, and a varying n

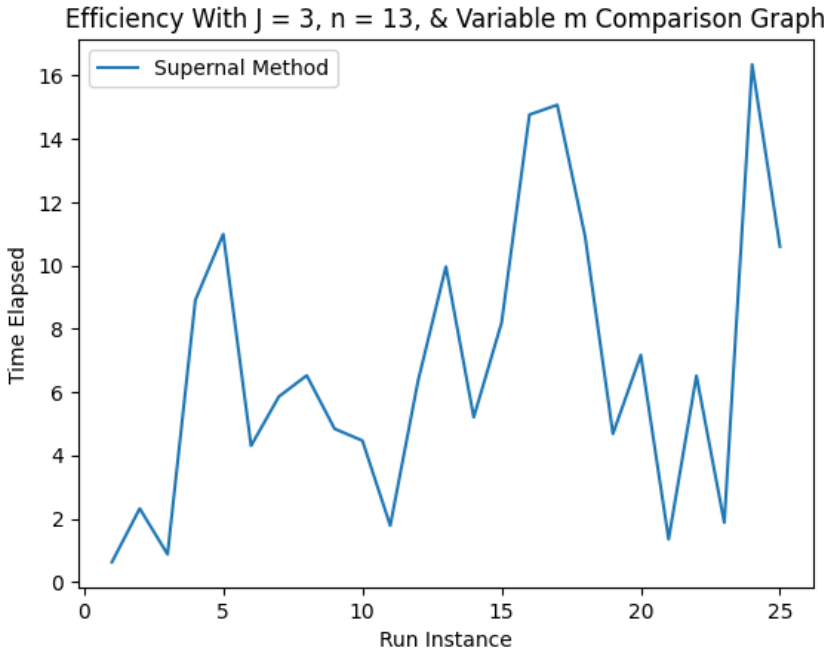

Figure 4: Efficiency of Supernal method with random instances generated using J = 3, n = 13, and a varying m

4.2 IMPROVEMENTS

4.2.1 BRUTE FORCE ENUMERATION IMPROVEMENTS AND COMPLEXITY

This algorithm implements an improved version of the most naive brute-force enumeration method. As before, the algorithm enumerates all the feasible solutions in the decision space. However, instead of finding all the objective images at once and comparing them with each other, it finds each objective image one at a time. More specifically, for each feasible solution, it finds it's objective image, and compares it with the images in the list of non-dominated points (which is initially empty). If the image is not dominated by any points in the list, it is added to the list. Any points in the list that are dominated by the new image are removed. Thus, we are only comparing each image to those which are not already dominated.

Algorithm 8 Improved Brute-Force Enumeration Method

Inputs: filename ▷ method = 1
Outputs: NDF

$n, b, c, a \leftarrow read\_input(filename)$ ▷ Read input file
$foundNDPs \leftarrow []$ ▷ Initialize list of found NDPs
$feas \leftarrow []$ ▷ Initialize list of feasible points
$arr \leftarrow np.empty(n)$ ▷ Initialize solution list of size n
$generateAllBinaryCombos(n, arr, 0, a, b, feas)$ ▷ Enumerate all feasible solutions
for $i\ in\ feas$ **do**
    $Calculate\ z = objective\ image$
end for
if $z\ not\ in\ foundNDPs$ **then**
    $dominated \leftarrow []$ ▷ Initialize list of dominated points
    for $ndp\ in\ foundNDPs$ **do**
        if $z \geq ndp\ (element\ wise)$ **then**
            $break\ outer\ for\ loop$ ▷ z is dominated and should not be considered
        end if
        if $z \leq ndp\ (element\ wise)$ **then**
            $dominated.append(ndp)$ ▷ ndp is dominated by z
        end if
    end for
    for $k\ in\ dominated$ **do**
        $foundNDPs.remove(k)$ ▷ Remove dominated points
    end for
    $foundNDPs.append(z)$ ▷ Append z to list of NDPs
end if
return $foundNDPs$

The improved brute-force algorithm focuses on the operation of removing dominated points. To do so, the algorithm iterates through each image and performs an element-wise comparison with each point in the current foundNDPs list. Only if an element is not dominated by any NDPs in the list it is added, and elements that are dominated by the new NDP are removed.

Worst case scenario, every point is an NDP, so on each iteration, no points are dominated and none are removed from the list of foundNDPs, we simply add the current NDP to the list. Subsequently, on each iteration, the number of comparisons is increasing by 1. As such, the task of removing dominated points has a final complexity of O_n = (n/2) * (1 + n) and a final Big-O complexity of O(n) = n, which is an improvement from the original Big-O complexity of O(n^2).

It should be noted that the Big-O complexity of the entire Brute-Force algorithm has a Big-O complexity of O(2^n) due to the task of enumerating all feasible solutions. This task has not been changed in the improved algorithm, thus the Big-O complexity of the improved algorithm is unchanged. However, the improvement to the task of removing dominated points reduces the exact complexity or runtime of the algorithm.

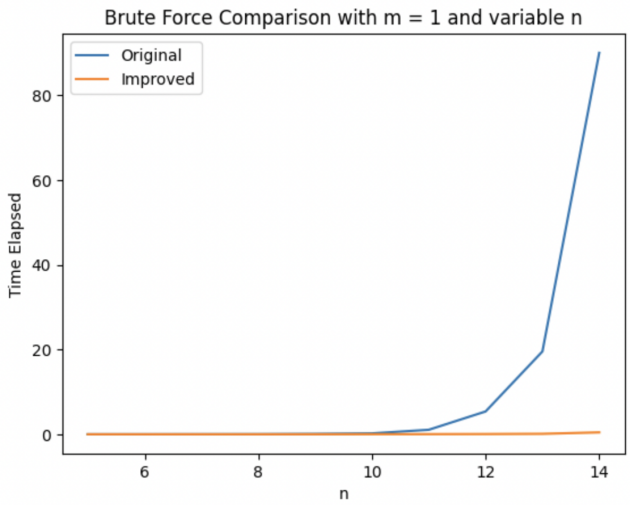

Figure 5: Efficiency of Original vs Improved Brute-Force Method

4.2.2 Rectangle Division Method Improvements and Complexity

The improved Rectangle Division Method aims to reduce the number of lexmin operations performed, and the number of rectangles processed. More specifically, we are only processing a rectangle (computing lex min) if we know it is possible for there to be a NDP inside. As such, we are eliminating the case when we solve the lex min, but it returns None (problem is infeasible). This is accomplished in two ways: 1. For each rectangle processed, search for a new northwest point and a new southeast point, and update the rectangle with the new points; 2. Consider the height of the rectangle. Namely, we will let the rectangle height be the difference between the y-values of the northwest and southeast points. Since each feasible point must be integer on all axes, the possible rectangle heights and corresponding actions are as follows (see Algorithms 9-11 for Python implementation):

- Rectangle height = 1: It is not possible for there to be an integer NDP that dominates the current northwest or southeast point in the rectangle. Thus, in this case we should not process any rectangle whose height is less or equal to 1. In other words, do nothing.
- Rectangle height = 2: There is at most one NDP in the rectangle. As such, searching for both a new northwest point and southwest point is redundant, as if either exists, they both exist, and are equal. Furthermore, since there can only be 1 NDP in the rectangle, we should not append a new rectangle to be processed.
- Rectangle height = 3: There can be up to two NDPs in the rectangle. As such, it is necessary to perform two lexmin operations to check if there is a new northwest point and/or southeast point. However, if two NDPs are found, it is not necessary to append a new rectangle because the new rectangle would have a height of 1, and as mentioned, it is not possible for there to be an NDP in a rectangle of height 1.
- Rectangle height = 4: There can be up to three NDPs in the rectangle. As such, we must bisect the rectangle and consider the top and bottom rectangles. Due to the integer constraints, if there are in fact three NDPs in the rectangle, there must be an NDP on the bisection line. We begin in the bottom rectangle and perform two lexmin functions to find a new northwest and southeast point (the northwest points would be on the bisection line). The top rectangle is refined and searched for an NDP. Again, it is not necessary to append a new rectangle because any rectangle would have a height of 1.
- Rectangle height = 5: There can be up to four NDPs in the rectangle. We must bisect the rectangle and solve lexmin in the top and bottom rectangles. There can be at most two NDPs in each rectangle. Again, it is not necessary to append a new rectangle because any rectangle would have a height of 1.

- Rectangle height = 6: There can be up to five NDPs in the rectangle. When we bisect the rectangle and solve lexmin in each. The bottom rectangle can have at most three NDPs and the top rectangle can have at most two NDPs. We must append the bottom rectangle to the list of rectangles to be processed again.
- Rectangle height ≥ 7: When the rectangle height is greater or equal to 7, there can be three or more NDPs in each bisected rectangle. As such, after solving lexmin in each bisected rectangle, we must append both rectangles to the list of rectangles to be processed again.

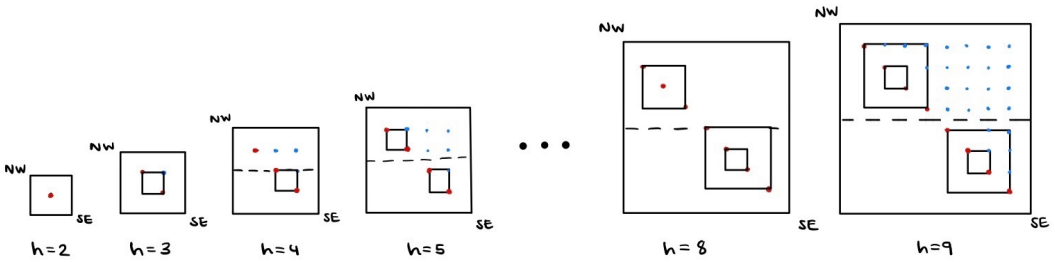

Figure 6: Rectangle Division for Different Size Rectangles

To evaluate the performance of the improved algorithm, we compare the runtime of the original rectangle division method with two improved methods. Improved1 implements the first improvement, whereby for each rectangle processed, we search for a new northwest point and a new southeast point and refine both corners of the rectangle (in comparison to only searching for a northwest or southeast point and refining one corner). Improved2 implements the above change, in addition to the height-based logic, as detailed in Algorithm 9-11. As demonstrated in Figure 7, both Improved1 and Improved2 perform much better than the Original. However, upon testing Improved1 and Improved2 on **inst_n375_m2_j2**, Improved1 took 843 seconds and Improved2 took 759 seconds. Therefore, it is evident that Improved2 performs slightly better for large values of n.

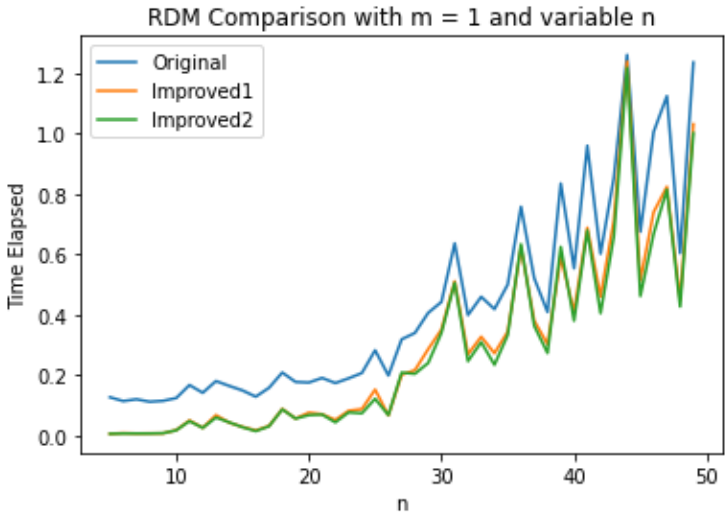

Figure 7: Comparison of Rectangle Division Methods (J=2)

In general, since we are usually concerned with the Big-O complexity for the worst case scenario, neither improvement would change the Big-O complexity. Both improvements aim to decrease computation time by reducing the number of times lex min must be computed. The first part of the improvement guarantees that no rectangle will be added to the list if its height is not equal or greater than 2, so the improved algorithm saves time by not iterating through rectangles where it is certain that no NDPs can be found. Moreover, rectangles are only bisected if their size is equal to or

greater than four which allows the algorithm to do less elementary operations. However, assuming the complexity of lex min is less than O(2^n), even reducing the complexity by a factor of lex min would not change the overall Big-O complexity.

Nonetheless, it is clear that the improvements discussed improve the exact complexity or runtime of the algorithm. As explained in Section 2.2.2, the worst case scenario involves having all of the NDPs aligned along the diagonal of the initial region, thus forming a square. This would mean that the maximum amount of NDPs that can be contained inside a rectangle is the height of the rectangle minus one (m = h-1). The following table shows how the two aforementioned improvements allow the algorithm to only do as many lexmin operations as there are NDPs inside the intial rectangle. This differs from the original RDM that requires twice as many lexmin operations as there NDPs inside the intial rectangle (see Section 2.2.2).

Table 2: Number of lexmin operations with respect to rectangle height r(h)

# Height (h)	**Max NDPs m(h)**	**Lexmin Operations r(h)**
1	0	0
2	1	1
3	2	2
4	3	3 = r(3) + r(2)
5	4	4 = r(3) + r(3)
6	5	5 = r(3) + r(3) + r(2)
.	.	
.	.	
.	.	
m	h-1	h-1 = r(3)(h-1)/2 when m is odd and = r(3)(h-2)/2 + r(2) when even

Algorithm 9 Improved Rectangle Division Method

Inputs: filename ▷ method = 2
Outputs: NDF

while $len(rectangles\_list) \neq 0$ **do**
  $R \leftarrow rectangles\_list[0]$
  $Remove\ R\ from\ rectangles\_list$
  $z1x \leftarrow x - coordinate\ of\ the\ NW\ point$
  $z1y \leftarrow y - coordinate\ of\ the\ NW\ point$
  $z2x \leftarrow x - coordinate\ of\ the\ SE\ point$
  $z2y \leftarrow y - coordinate\ of\ the\ SE\ point$
  $rec\_height = z1y - z2y$
  $z\_mid\_y = (z1y + z2y)/2$ ▷ Find midpoint of rectangle
  if rec_height = 2 **then** ▷ Due to integer constraint, only objective 1 can improve
    $R \leftarrow [[z1x, z1y - \epsilon], [z2x - \epsilon, z2y]]$
    $Solve\ lex\ min\ for\ z\_hat\_west\ in\ R$
    if $z\_hat\_west\ is\ not\ None$ **then**
      $append\ z\_hat\_west\ to\ foundNDPs$ ▷ A new NDP is found
    end if
  end if
  if rec_height = 3 **then**
    $R \leftarrow [[z1x, z1y - \epsilon], [z2x - \epsilon, z2y]]$
    $Solve\ lex\ min\ for\ z\_hat\_west\ and\ z\_hat\_south in\ R$
    if $z\_hat\_west\ is\ not\ None\ and\ z\_hat\_south\ is\ not\ None$ **then**
      if $z\_hat\_west = z\_hat\_south$ **then**
        $append\ z\_hat\_west\ to\ foundNDPs$
      end if
    else
      $append\ z\_hat\_west\ and\ z\_hat\_south\ to\ foundNDPs$
    end if
  end if
  if rec_height = 4 **then**
    $R2 \leftarrow [[z1x, z\_mid\_y], [z2x - \epsilon, z2y]]$
    $Solve\ lex\ min\ for\ z\_hat\_west\ and\ z\_hat\_south\ in\ R2$
    if $z\_hat\_west\ is\ not\ None\ and\ z\_hat\_south\ is\ not\ None$ **then**
      if $z\_hat\_west = z\_hat\_south$ **then**
        $append\ z\_hat\_west\ to\ foundNDPs$
      else
        $append\ z\_hat\_west\ and\ z\_hat\_south\ to\ foundNDPs$
      end if
      $R3 \leftarrow [[z1x, z1y - \epsilon], [z\_hat\_west[0] - \epsilon, z\_mid\_y]]$
    else
      $R3 \leftarrow [[z1x, z1y - \epsilon], [z2x - \epsilon, z\_mid\_y]]$
    end if
    $Solve\ lex\ min\ in\ R3\ for\ z\_squiggly\_west$
    if $z\_squiggly\_west \neq None$ **then**
      $foundNDPs.append(z\_squiggly\_west)$
    end if
  end if
  $ALGORITHM\ CONTINUED\ ON\ NEXT\ PAGE$
end while

Algorithm 10 [Continued] Improved Rectangle Division Method

Inputs: filename ▷ method = 2
Outputs: NDF

while $len(rectangles\_list) \neq 0$ **do**
  if rec_height = 5 **then**
    $R2 \leftarrow [[z1x, z\_mid\_y], [z2x - \epsilon, z2y]]$
    $Solve\ lex\ min\ for\ z\_hat\_west\ and\ z\_hat\_south\ in\ R2$
    if $z\_hat\_west\ is\ not\ None\ and\ z\_hat\_south\ is\ not\ None$ **then**
      if $z\_hat\_west = z\_hat\_south$ **then**
        $append\ z\_hat\_west\ to\ foundNDPs$
      else
        $append\ z\_hat\_west\ and\ z\_hat\_south\ to\ foundNDPs$
      end if
      $R3 \leftarrow [[z1x, z1y - \epsilon], [z\_hat\_west[0] - \epsilon, z\_mid\_y]]$
    else
      $R3 \leftarrow [[z1x, z1y - \epsilon], [z2x - \epsilon, z\_mid\_y]]$
    end if
    $Solve\ lex\ min\ in\ R3\ for\ z\_squiggly\_west\ and\ z\_squiggly\_south$
    if $z\_squiggly\_west\ is\ not\ None\ and\ z\_squiggly\_south\ is\ not\ None$ **then**
      if $z\_squiggly\_west = z\_squiggly\_south$ **then**
        $foundNDPs.append(z\_squiggly\_south)$
      else
        $append\ z\_squiggly\_west\ and\ z\_squiggly\_south\ to\ foundNDPs$
      end if
    end if
  end if
  if rec_height = 6 **then**
    $R2 \leftarrow [[z1x, z\_mid\_y], [z2x - \epsilon, z2y]]$
    $Solve\ lex\ min\ for\ z\_hat\_west\ and\ z\_hat\_south\ in\ R2$
    if $z\_hat\_west\ is\ not\ None\ and\ z\_hat\_south\ is\ not\ None$ **then**
      if $z\_hat\_west = z\_hat\_south$ **then**
        $append\ z\_hat\_west\ to\ foundNDPs$
      else
        $append\ z\_hat\_west\ and\ z\_hat\_south\ to\ foundNDPs$
        $append$ [z_hat_west,z_hat_south] $to\ rectangles\_list$
      end if
      $R3 \leftarrow [[z1x, z1y - \epsilon], [z\_hat\_west[0] - \epsilon, z\_mid\_y]]$
    else
      $R3 \leftarrow [[z1x, z1y - \epsilon], [z2x - \epsilon, z\_mid\_y]]$
    end if
    $Solve\ lex\ min\ in\ R3\ for\ z\_squiggly\_west\ and\ z\_squiggly\_south$
    if $z\_squiggly\_west\ is\ not\ None\ and\ z\_squiggly\_south\ is\ not\ None$ **then**
      if $z\_squiggly\_west = z\_squiggly\_south$ **then**
        $append\ z\_squiggly\_south\ to\ foundNDPs$
      else
        $append\ z\_squiggly\_west\ and\ z\_squiggly\_south\ to\ foundNDPs$
      end if
    end if
  end if
  $ALGORITHM\ CONTINUED\ ON\ NEXT\ PAGE$
end while

Algorithm 11 [Continued] Improved Rectangle Division Method

Inputs: filename ▷ method = 2
Outputs: NDF

while $len(rectangles\_list) \neq 0$ **do**
  if rec_height = 6 **then** ...
  else
    $R2 \leftarrow [[z1x, z\_mid\_y], [z2x - \epsilon, z2y]]$
    $Solve\ lex\ min\ for\ z\_hat\_west\ and\ z\_hat\_south\ in\ R2$
    if $z\_hat\_west\ is\ not\ None\ and\ z\_hat\_south\ is\ not\ None$ **then**
      if $z\_hat\_west = z\_hat\_south$ **then**
        $append\ z\_hat\_west\ to\ foundNDPs$
      else
        $append\ z\_hat\_west\ and\ z\_hat\_south\ to\ foundNDPs$
        $append$ [z_hat_west,z_hat_south] $to\ rectangles\_list$
      end if
      $R3 \leftarrow [[z1x, z1y - \epsilon], [z\_hat\_west[0] - \epsilon, z\_mid\_y]]$
    else
      $R3 \leftarrow [[z1x, z1y - \epsilon], [z2x - \epsilon, z\_mid\_y]]$
    end if
    $Solve\ lex\ min\ in\ R3\ for\ z\_squiggly\_west\ and\ z\_squiggly\_south$
    if $z\_squiggly\_west\ is\ not\ None\ and\ z\_squiggly\_south\ is\ not\ None$ **then**
      if $z\_squiggly\_west = z\_squiggly\_south$ **then**
        $append\ z\_squiggly\_south\ to\ foundNDPs$
      else
        $append\ z\_squiggly\_west\ and\ z\_squiggly\_south\ to\ foundNDPs$
        $append$ [z_squiggly_west,z_squiggly_south] $to\ rectangles\_list$
      end if
    end if
  end if
end while
return $foundNDPs$

4.2.3 SUPERNAL METHOD IMPROVEMENTS AND COMPLEXITY

The improved version of the supernal method is aiming to reduce the complexity of some of the repetitive tasks performed in the original version of the algorithm. The improvements include changing the way in which the next search region is selected, the process for which the weighting vector for the objective functions (lambda values) are changed both dynamically and statically, as well as removing dominated regions before refining and looping through the regions.

The first change that was implemented was to remove the dominated regions before iterating through all of the current regions. Note that this change only impacts instances where $J > 2$. The change to the code was simply moving the removeDominated function just above the loop through the regions, instead of calling it at the end of the loop. Essentially this adjustment means that the algorithm is removing any dominated regions before searching through all the regions and removing/creating new ones. In terms of the adjustment to the overall complexity of the SPM algorithm, this improvement does not directly impact the big-O theoretical complexity since the code is not being changed, only reordered. However, the efficiency of the algorithm is improved because there are potentially fewer regions to iterate through since the dominated regions are removed beforehand. Since fewer regions are being searched, the parameter R from the previous SPM complexity decreases as a result of this improvement, and so overall this change improves the runtime as can be seen in the figure below.

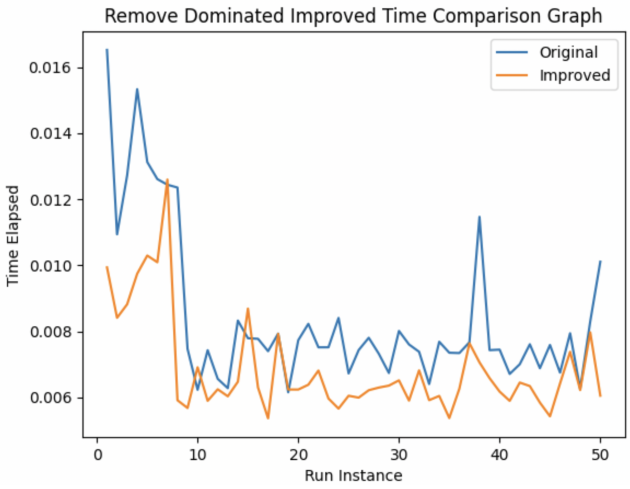

Figure 8: Efficiency of Original vs Improved Supernal Method using the Remove Dominated Change

The next change was to randomly select the next region to be processed rather than always picking the northeast region. This change can be seen in line fifteen of the improved supernal method pseudocode below. Although this alternate region selection method does not directly change the big-O complexity, it does improve efficiency because on average the random region selection is faster than picking the most northeast points, and in the worst case, it would be the same.

Finally, changes were made to determine the optimal lambda values to be used in the algorithm. The first adjustment to improve the SPM's weightings was to make lambda static and equal between objectives (see line three of the pseudocode). This change simplified the theoretical complexity of the algorithm as assigning the weightings previously had a $O_n = J$, from normalizing all the lambda's, and now has a complexity of zero as no elementary operations are performed. Therefore, this small change reduced the theoretical complexity of this algorithm by J. In addition to this change, a dynamic weighting strategy was used. Instead of establishing the lambdas at the beginning of the algorithm and remaining constant throughout, this technique changes the weightings for each objective as the model iterates to adapt to the more important objectives. This strategy was implemented by first resetting the weightings in every iteration of the outermost loop after regions were added and removed. The new lambdas for that iteration were then recalculated by considering which objectives were the most important, given the current regions. For an example where J=2, if the new NDPs were progressively getting more negative on the z_1 axis, then it is known that the state

of the optimization is prioritizing the first objective. To implement this logic, the new weightings were calculated based on the total distance of all NDPs from each of the axes, and the higher the total distance of NDPs to one of the axes, the more it was prioritized. To reflect this, the lambda for each of the objectives was then increased based on its total distance to give it more/less weight according to its proportion of the total distance to all NDPs (see lines nine to thirteen in the pseudocode). The weighted sum model is also updated to incorporate the new lambdas into the objective function each iteration. In terms of the impact of these lambda changes, the only difference in theoretical complexity comes from the updating of the lambdas in each iteration of the while loop. In order to update the lambdas, two nested loops are needed, the first through all of the NDPs and the second through J. Since in the worst case, there are R total NDPs, and each one is looped through J times, the complexity of the SPM increases by JR as a result of these lambda changes. Although this change seems to negatively impact the runtime of the algorithm as the theoretical complexity has worsened, it has in fact improved because the total number of regions that are being searched has decreased (R is decreasing). This is apparent in the figure below which reveals the much lower run time of the improved SPM for the instance where n=10, where m = 4 and J = 3.

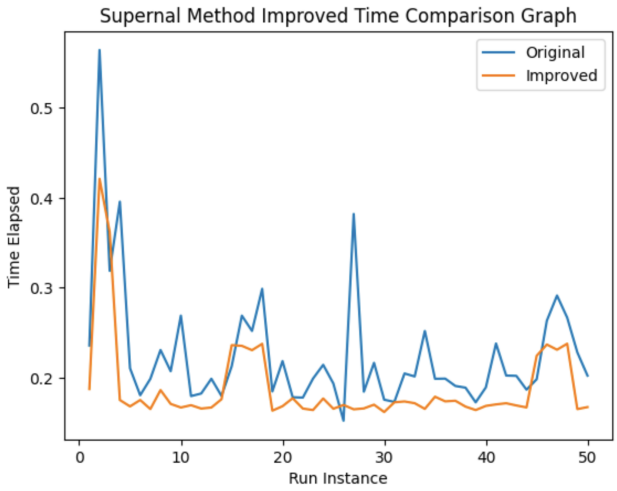

Figure 9: Efficiency of Original vs Improved Supernal Method using the Optimized Lambda Changes

Combining all these changes results in an overall increase in the theoretical complexity of the SPM by $JR - J$ since initializing the lambas improved it by J, and the dynamic lambda selection worsened the complexity by JR (even though runtime improved due to decreased R). The following chart compares the runtime of the original SPM with the improved version which contains all the changes to the lambda selection, random region selection, and removing dominated regions.

Algorithm 12 Improved Supernal Method

Inputs: filename, method ▷ method = 5
Outputs: run_time, foundNDPs, number of regions searched

Create an empty list called $FoundNDPs$
...
Create list of $\lambda \leftarrow [1] * J$: **lmbd** ▷ Initialize lambdas
Create min : $\lambda^T z$ **model** $\leftarrow$ GET_WEIGHTED_SUM_MODEL$(c, a, b, n, M, J, reg, \lambda)$
while *regions is not empty* **do**
  for $j \leftarrow 0$ **to** $J - 1$ **do**
    Reset $\lambda[j] \leftarrow [0]$ ▷ Reset the **lmbd**
  end for
  for $npd \in foundNDPs$ **do**
    for $j \leftarrow 0$ **to** $J - 1$ **do**
      $\lambda[j] += ndp[j]$ ▷ Calculate the distance of the NDPs from the origin
    end for
  end for
  $Num\ of\ regions\ searched += 1$
  Select $reg \leftarrow regions$ ▷ Randomly select the next region to be solved
  for $i \leftarrow 0$ **to** $J - 1$, $r \in reg[i \leftarrow 0, J - 1]$ **do**
    model.$\_z[i].ub \leftarrow r$ ▷ Update UBs of the model
    model.$\_lmbd\_new[i] \leftarrow \lambda[i]$ ▷ Update lambdas of the model
  end for
  Update and optimize **model** ▷ Solve weighted sum problem in current region
  if model status $= feasible$ **then**
    $z^* \leftarrow$ GET_SUPERNAL_Z$(n, c,$ **model**$)$ ▷ Get supernal point
    Add z^* *to* $foundNDPs$ ▷ Add supernal point to list of found NPDs
    $regions\_temp \leftarrow []$ ▷ Initialize temporary list of new regions
    $regions\_to\_remove \leftarrow []$ ▷ Initialize list of regions to remove
    if $J \geq 3$ **then**
      REMOVEDOMINATED$(regions)$ ▷ Remove dominated regions
    end if
    for $region$ **in** $regions$ **do**
      $count \leftarrow 0$ ▷ Set counter to 0
      for $i \leftarrow 0$ **to** $J - 1$ **do**
        if $z^* \leq region$; *for each dimension in* z^* **then**
          $count += 1$
        end if
      end for
      if $count = J$ **then**
        Add region to list : regions_to_remove
        $new\_reg \leftarrow []$ ▷ Initialize list of new regions
        for $i \leftarrow 0$ **to** $J - 1$ **do**
          Set $new\_reg \leftarrow region$ ▷ Assign region as the new region
          for $j \leftarrow 0$ **to** $J - 1$ **do**
            if $i = j$ **then**
              Set $new\_reg[j] = z^* - 1$ ▷ Update dimension j of the new region
            end if
          end for
          Add new_reg *to* : regions_temp ▷ Append new_reg to list of new regions
        end for
      end if
    end for
    for $r \in$ regions_to_remove **do**
      Remove r $from\ regions$ ▷ Update regions by removing all obsolete regions
    end for
    for $r \in$ regions_temp **do**
      Add r *to regions* ▷ Update regions by appending all new regions
    end for
  else
    Remove reg $from\ regions$ ▷ reg is infeasible
  end if
end while

5 CONCLUSION

The algorithms discussed in this report are all methods of solving multi-objective optimization problems. By analyzing their complexity, efficiency, and improvements, the comparison between these three algorithms yielded the Supernal method as the best algorithm for this purpose. From the charts and pseudo-code presented earlier in the report, it was seen the Brute Force method was found to have the worst run-time and efficiency with all variations of parameters. In addition, this algorithm is only capable of solving multi-objective optimization problems when there are two objectives. The Rectangle and Supernal method had similar run-time efficiencies and complexities. This similarity between the two methods and their significant improvement over Brute Force was surprising considering all three of them have a worst-case exponential complexity. The difference between these methods is that the Supernal algorithm is capable of solving multi-objective optimization problems when there are more than two objectives. It can be concluded, from these analyses, that the Supernal is the best method to solve an m-KP multi-objective optimization problem.