SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

FUNDAMENTALS OF OPTIMIZATION

COURSE: IT3052E

Harvest Planning Problem

Author

LE NHAT QUANG - 20225522

TRAN CHI HIEU - 20225572

NGUYEN NGHIA HA – 20225570

PHAM TRAN TUAN KHANG - 20225503

Mentor

BUI QUOC TRUNG

January, 2024

Table of Contents

Harvest Scheduling Problem

January 2024

Abstract

There are no known polynomial-time algorithms for solving NP-hard problems in general. The Harvest Scheduling Problem is one of them. The purpose of this report is to reaffirm this statement by tackling this problem by several methods, as well as to compare them to further enhance understandings for different algorithms. The algorithms used here are Branch and Bound, Contraint Programming (CP), Integer Programming (IP), Greedy, Local Search and Metaheuristic. The results show that as the input's size increases, the time taken to compute solutions rises. The Branch and Bound algorithm couldn't produce solutions for large inputs in acceptable time. All solutions (except when the input size is small) are near-optimal. In conclusion, it shows that the most advanced algorithms used here can only provide good enough solutions but not optimal. In a limited amount of time, some algorithms outperform others as we will see.

1. Introduction

The Havesting Planning Problem is stated as follow: there are N fields 1, 2, …, N. Each field has its own amount of crops to be collected and a harvesting period. We are

asked to schedule them for harvest to achieve some objectives.

Notable features:

- The amount of crops and harvesting days are positive integers.
- Harvesting constraint: Every field can only be harvested once and this can only be done in a designated harvesting period.
- Factory capacity constraint: In any day, the processing factory will only work if the total amount of crops collected is no less than m and no more than M.
- The two objectives are to maximize the total amount of crops collected is all harvesting days and to minimize the differences in the amount of crops collected in each harvesting days.

Here are steps that our team implemented the project:

- Phase 1: Mathematical modelling of the problem and analysis of the optimal solution.
- Phase 2: Algorithm Testing and Refinement.
- Phase 3: Performance Evaluation.
- Phase 4: Final Testing and Project Wrap-up.

2. Methodology

2.1. Mathematical modelling

There are N fields $1, 2, 3, \dots, N$ growing the same agricultural product. The field i has the amount of product $d(i)$ and needs to be harvested in the period from $s(i)$ to $e(i)$. The processing factory has capacity M, which is the maximum amount of product it can process each day. Moreover, if the total amount of product available is less than m, the factory will not work (as the efficiency is not good). Each field can only be harvested once.

Let x_{ij} be the decision variable for the harvest of field i in the day j. Then $x_{i_j} = 0$ if the field i

is not harvested on the day j and $x_{i_j} = 1$ otherwise.

Then the factory capacity would be:

$$m \leq \sum_{i=1}^{N} d_i \times x_{ij} \ \forall \, day \ j \leq M$$

Each field can only be harvested once:

$$\sum_{j = min(s(i))}^{max(e(i))} x_{ij} = 1 \ for \ all \ field \ i$$

MAXIMAL TOTAL AMOUNT OF PRODUCT HARVESTED

$$f(x) = \sum_{i=1}^{N} \left(\sum_{j=startDay}^{endDay} d_i \times x_{ij} \right)$$

2.2. Constraint Programming (CP)

2.2.1. Introduction

Constraint programming (CP) is a paradigm within the field of computer science and artificial intelligence that deals with solving problems by expressing relationships among variables in the form of constraints. In constraint programming, you model a problem as a set of variables, each with a domain of possible values, and a set of constraints that define relationships and restrictions among these variables.

The goal of constraint programming is to find a solution that satisfies all the specified constraints. It's particularly useful for solving combinatorial problems where the search space is vast and exploring all possible combinations is impractical.

Key components of constraint programming include:

- **Variables:** These represent the unknowns in the problem, and each variable has a domain of possible values.
- **Constraints**: These are relationships or restrictions between variables that must be satisfied for a solution to be valid. Constraints define the feasible solutions to the problem.
- **Objective Function**: In some cases, there may be an associated objective function that you want to optimize or minimize while still satisfying the constraints.
- **Search and Propagation**: Constraint programming often involves a search process to explore the solution space systematically. Propagation techniques help reduce the search space by eliminating inconsistent values for variables based on the constraints.
- **Solver:** A constraint solver is a software tool or system that implements algorithms to efficiently search for solutions that satisfy the constraints.

Constraint Programming can be applied in:

- **Scheduling Problems** When you have a set of jobs that need to be scheduled with time constraints.
- **Bin Packing Problem** If you want to pack objects into bins so that space is maximized.
- **Production Scheduling** When you want to determine the production schedule so that constraints such as material availability, production time, and production capacity are met.
- **Timetabling** If you are trying to define schedules for events or tasks with constraints such as events cannot happen at the same time.
- **Resource Scheduling** When you need to distribute resources such as machinery, personnel, and raw materials in an efficient way.

2.2.2. Problem Adaptation

Mathematical modelling is the same with the maximal amount of product harvested objective function above.

The structure of the program:

```
# Defining variables: Whether each field is harvested on a particular day

Pseudocode:

 Initialize an empty dictionary named "harvest".

 For each field index 'i' from 0 to N-1:

   . Loop through each day starting from the start day of field 'i' until the end day of field 'i' (inclusive).

      i. For each day in this range:

         - Create a boolean variable representing whether field 'i' is harvested on this day.

         - The name of the boolean variable is formatted as 'harvest_field_{i}_day_{day}'.

         - Store this boolean variable in the "harvest" dictionary with a key that is a tuple consisting of the field index 'i' and the current day.

# constraints: Each field must be harvested exactly once within its available days

Pseudocode:

 For each field from 0 to N-1

    Create a sum variable for the current field

    For each day from the start day to the end day of the field

        Add the 'harvest' variable for the current field and day to the sum

   Impose the constraint: the sum must equal 1 (ensuring the field is harvested exactly once)


# constraints for total harvested amount each day to be within the minimum and maximum limits

Pseudocode:

For each day from 1 to the maximum of the end dates in the fields array (inclusive):

    Initialize daily_harvest to 0

    For each field i from 0 to N-1:

        If the day is within the start and end dates for field i:

            Add the product of the harvest variable for field i on this day and the value at fields[i][0] to daily_harvest

    Add a constraint to the model that daily_harvest must be at least m

    Add a constraint to the model that daily_harvest must be no more than M


# Objective: Maximize the total amount harvested

# Calculate the total harvest by summing the product of harvest, field size, and the number of days harvested

total_harvest = sum(harvest[(i, day)] * fields[i][0] for i in range(N) for day in range(fields[i][1], fields[i][2] + 1))

# Define the objective function to maximize the total harvest

model.Maximize(total_harvest)
```

Advantages:

- **Easy to read and understand:** The source code is clear and easy to understand with the use of CP. The structure of the code often directly reflects the constraints of the problem.
- **High overall:** CP helps to model the problem holistically and establish constraints logically. This can reduce complexity compared to traditional search methods.
- **Automatic optimization:** CP often automatically performs efficient optimization, searching for the best solution in the variable space.
- **High adjustability:** You can easily change and test different constraints to see if the results are as desired.

Disadavantages:

- **Performance depends much on the nature of the problem:** CP's performance can be greatly influenced by the specific natire of the problem. In some cases, traditional search methods may be more effective.
- **Difficulties with large-sized problems:** In some cases, when the problem is large in size, the search space may become large and reduce the performance of the CP.
- **Limited persistent binding ability:** CP is usually well-suited to problems with discrete constraints, but can have difficulty dealing with continous constraints.
- **Difficulty in handling complexed constraints:** Some complicated constraints can increase the complexity of the problem and make finding a solution difficult.

2.3. Interger Programming (IP)

2.3.1. Introduction

Integer programming is an optimisation method that involves the use of integer variables, or "whole numbers". It is a type of mathematical programming problem used to find optimal solutions for certain types of problems. The goal of this technique is to maximise profit while minimising cost and resources. Integer programming differs from linear programming in its use of integer constraints, instead of real number constraints. Linear programming techniques can be applied to solve all kinds of problems, but when it comes to dealing with binary (0/1) and mixed-integer decisions, they are not always suitable.

Key concepts of integer programming include:

- **Decision variables:** Integer programming involves decision variables that represent the quantities to be determined. Unlike linear programming, which allows variables to take any real value, IP requires these variables to be integers.
- **Objective function:** The objective function in IP defines the goal of the optimization problem, whether it is to maximize or minimize a certain quantity. This function is typically a linear combination of the decision variables.
- **Constraints:** Constraints are conditions or limitations that the solution must satisfy. These constraints are linear relationships involving the decision variables and constants. Integer programming adds the requirement that some or all decision variables must be integers.

2.3.2. Problem adaptation

Mathematical modelling for IP is the same as that for CP.

Advatages:

- More modelling power
- More readily implementable answers

Disadvantages:

- More complexed models
- Much longer to get answers.

2.4. Greedy Algorithm

Heuristic:

We want to maximize the total productivity of the scheme → We should collect the most productive fields as much as possible while satisfying the constraint → We should prioritize setting the schedule for the high-productive fields in the way satisfying the constraint first to get rid of the case we miss any high-productive fields while accepting other lower-productive one.

Idea:

First, we sort the fields based on their productivity (from high to low). Then in each field (denote field a), we consider which is the most suitable day ranging from the start day to the end day we can assign the field a to. It's quite simple to find, just scan the day sequentially from start to end. If we find a day whose total productivity plus the productivity of the field a is equal or smaller than M (upper bound condition), then we just assign the field a to that day. Else we just keep scanning until the upper bound condition is satisfied.

After that we have a complete scheme. But be careful! We've just implemented the heuristic obeying the upper bound constraint while we haven't mentioned anything about the lower bound constraint (m). So, we must cut off the day whose productivity is smaller than m.

Finally, we get the optimal harvest scheme of the algorithm.

Pseudocode:

```
Function Find_the_best_harvest_scheme(fields,m,M):
Sort the fields according to the productivity of each field
        For each field in fields:
                For each day range from start day and end day:
                        If the total prod. of the day + prod. of the field <= M:
                                Assign the field to the day
                                Update the total productivity of the day
                end For
        end For
        Scan the harvest scheme to cut off the day whose total productivity < m
        Return the harvest scheme
end Function
```

Time and space complexity of Greedy Approach:

- Time complexity: O(N*d)
- Space complexity: O(N)

Advantages:

- **Simplicity:** Greedy algorithms are often straightforward and easy to understand, making them easy to implement.
- **Efficiency:** In many cases, greedy algorithms provide reasonably efficient solutions with a relatively simple implementation.
- **Optimality for Certain Problems:** Greedy algorithms are optimal for some problems where making locally optimal choices at each step leads to a globally optimal solution.

Disadvantages:

- **No Backtracking:** Greedy algorithms make locally optimal choices at each step without considering the global picture. This lack of backtracking may lead to suboptimal solutions.
- **Not Always Optimal:** Greedy algorithms do not guarantee globally optimal solutions in all cases. In some scenarios, the locally optimal choice at each step may not result in the best overall solution.
- **Dependency on Heuristic:** The effectiveness of a greedy approach often relies on the chosen heuristic. If the heuristic is not well-designed, the algorithm might not perform optimally.
- **Sensitivity to Input:** The greedy approach may be sensitive to the order of input data. Different orderings of input could lead to different solutions.

In the context of the harvest scheme problem, the greedy approach seems reasonable and efficient, we tried it on Hustack and it ran quite fast and produced acceptable answers. However, it's essential to validate its performance on various test cases and be aware of its limitations, especially when dealing with different types of constraints and input distributions. It can run very poorly when dealing with some tricky input.

2.5. Local Search

Idea:

First, we find a brute force solution, this is the initial scheme we based on to implementing the local search. Because the effectiveness of the local search is based on the quality of the start config, we should take the brute force solution as good as possible. Instead of finding a brute force solution one time only, we randomize the input of the brute force function (to have different start of the brute force) to implement the brute force. We need to choose the best config returned by the randomized brute force algorithms.

After finding the best brute force plan, Iterate 10000 times:

We randomly (random walk) modify the harvest day of 1 field in that plan to the other interval-satisfied day.

Compare the previous scheme with the shifted scheme. Update if finding the better scheme.

After the iterations, we should find the ultimate scheme.

Pseudocode:

```
Func best_brute_force(Fields, m, M, iterations=100):
        Initiate empty best scheme
                For each iteration:
                        Shuffle the fields
                        For each Field in Fields:
                                Initiate best day value
                                For each day ranging from the start day and the end day of the
                        Fields:
                                        If the day have potential to be the best day while
                                        satisfying the upper bound:
                                                Assign the day to the best day
                                        end For
                                If the best day is already set:
                                        Update the total product of the best day
                                        Update the harvest scheme
                                end For
                        Drop off the day whose productivity is smaller than m in the harvest
                        scheme
                        end For
        Update the best scheme by comparing the previous best scheme and current scheme
        Return the best scheme
        end Func
Func evaluate_plan(plan, fields):
        Initiate the total harvested is zero
        For each field and its assigned day in plan:
                if the day is not Null:
                        Add the productivity of the field to the total harvested
                        end If
                end For
        Return  the total harvest of the plan
        end Func
Func is_valid_plan(plan, fields, m, M):
        For each field and its assigned day in plan:
                if the day is not null:
                        Calculate the total productivity of the day
                if the day does not satisfy the constraint (<= M and >= m):
                        return False
                end For
        Return True
        end Func

Func Local_Search(N, m, M, fields):
        Harvested = best_brute_force(Fields, m M)
        Initiate current plan is empty array, current_plan[i] indicates the day assigned to the
        field number i.
        Sort the harvested according to the order of the fields
        For each scheme in harvested:
                Update the current plan
                end For
        While iteration < 10000 times:
                Randomize the field to change
                Create a neighbor plan, which is the copy of the current plan
                Navigate the randomized field in the neighbor plan, then modify the day of it to
        another day between the       start day and the end day of the randomized field.
                If evaluate_plan(neighbor_plan) > evaluate_plan(current_plan) and
                is_valid_plan(neighbor_plan):
                        Update the current plan to the neighbor plan
                        end If
        Return the current plan
        end Func
```

Time and space complexity of this Local Search algorithm:

- Time complexity: $O(N \times D \times Iteration)$
- Space complexity: $O(N)$

Advantages:

- **Complete exploration of the solution space**: The local search efficiently explores the solution space by thousand iteratively making small change to the current solution. This is particularly advantageous for large problem instances like this problem where exhaustive search may be impractical.

- **Adaptability to the constraints**: Because we check the valid plan every time a new plan is updated, the algorithm can adapt to various constraints and objectives.
- **Randomization for diversity**: the use of randomization in the local search algorithm introduces diversity in the exploration, helping to avoid getting stuck at local maxima. This can lead to a more robust and varied search, further global maxima.

Disadvantages:

- **Dependency of the initial solution**: The quality of the initial solution obtained from the brute-force search can influence the final solution. If the initial solution is suboptimal, the algorithm might struggle to improve it.
- **Dependency of the randomization**: We tried to give the best initial solution from doing hundred times of the brute force search, but if the randomization was not good enough, we had got a not good initial state and then probably not the best solution after performing local search.

MINIMAL DIFFERENCES IN THE AMOUNT OF PRODUCT COLLECTED BETWEEN HARVESTING DAYS

$$f(x) = max(plan) - min(plan) \rightarrow min$$

Our solution can be represented by a list of length N called **plan** where plan[i] is the day the field[i] is harvested. As such, the state space of this problem is a set of "complete" configurations of the list plan where $s[i] \leq plan[i] \leq e[i]$.

This makes Branch-and-Bound and metaheuristic algorithms suitable for the task. The metaheuristic algorithm discussed here is the Simulated Annealing algorithm.

2.6. Branch-and-Bound

2.6.1. Introduction

The branch-and-bound (B&B) framework is a fundamental and widely-used methodogy for producing exact solutions to NP-hard optimization problems. The algorithm's procedure implicitly enumerates all possible solutions to the problem under consideration by storing partial solutions called subproblems in a tree structure.

Unexplored nodes in the tree generate children by partitioning the solution space into smaller regions that can be solved recursively, and rules are used to prune off regions of the search space that are provably suboptimal. Once the entire tree has been explored, the best solution found in the search is returned.

In the above framework, there are three components having significant impacts on the performance of the algorithm. These components are the **search strategy** (*i.e., the order in which subproblems in the tree are explored*), the **branching strategy** (*i.e, how the solution space is partitioned to produce new subproblems in the tree*), and the **pruning rules** (i.e*, rules that prevent exploration of sub-optimal regions of the tree*).

Pseudocode for the generic B&B procedure is given as follow:

Algorithm 1 Get Path Function

```
procedure GetPath(forward_parents, backward_parents, meeting_node)
    path ← []
    node ← meeting_node
    while node ≠ None do
        path.insert(0, node)
        node ← forward_parents[node]
    end while
    node ← backward_parents[meeting_node]
    while node ≠ None do
        path.append(node)
        node ← backward_parents[node]
    end while
    return path
end procedure
```

2.6.2. Problem adaptation

The search and branching strategies:

Iterating through the list **plan,** we assign plan[i] with values in the increasing order from s[i] to e[i] for i = 1, …, N and consider the partial solutions as the computer moves from one to another.

The pruning rules:

Let k be the objective value we wish to get ($k \in N$) and $S = \sum_{i=1}^{N} d(i)$ be the total amount of crops harvested in all days. Then S is fixed, and k is arbitrarily chosen.

During the search and branching strategies, there might be fields that haven't been scheduled. Consider the fields that have been scheduled and their harvesting days, it is possible to generate a list consisting of the amount of crops collected each day (only days with a positive amount of crops harvested are taken into account). Assume that the elements of that list are $a_i \in N^* \forall i = 1, \dots, n$ where $n \leq N$ and $a_i \leq a_{i+1} \forall i = 1, \dots, n-1$.

We will show that $\boldsymbol{a_n \leq \frac{S+(n-1)\times k}{n}}$. The pruning rule will not consider partial solutions where $a_n > \frac{S+(n-1)\times k}{n}$ where $S = \sum_{i=1}^{N} d(i)$.

From the formulation, we have:

$$\sum_{i=1}^{n} a_i \leq S$$

$$a_i \leq a_{i+1} \forall i = 1, \dots, n-1$$

$$a_n - a_1 \leq k$$

Then, it can be shown that:

$$a_n - a_{n-1} \leq \dots \leq a_n - a_2 \leq a_n - a_1 \leq k$$

As such,

$$a_i \geq a_n - k \; \forall i = 1, \ldots, n-1$$

Therefore,

$$S \geq \sum_{i=1}^{n} a_i \geq (n-1) \times (a_n - k) + a_n$$

From this, we can conclude that $a_n \leq \frac{S+(n-1)\times k}{n}$.

It can be checked that as **n increases, this upper bound decreases** with k fixed. So, consider a partial solution where $n < N$, assume that $a_n > \frac{S+(n-1)\times k}{n}$, then we would obtain the following inequalities:

$$\frac{S+(n-1)\times k}{n} < a_n \leq a_N \leq \frac{S+(N-1)\times k}{N}$$

$$\frac{S+(n-1)\times k}{n} > \frac{S+(N-1)\times k}{N} \; since \; n < N$$

This is contradictory. We can conclude that this **pruning rule is valid** in the sense that it will not omit partial solutions that might develop into the optimal solution**.**

Another constraint to be considered is the bounds for the factory capacity for each day. A partial solution must satisfy both of these constraints to be expanded.

Advantage: If the problem is not large and if we can do the branching in a reasonable amount of time, the algorithm finds the optimal solution.

Disadvantages:

- It can be very time-consuming and computationally expensive, especially for large and complex problems.
- The design of a good bounding strategy that can provide accurate and reliable bounds for the subproblems.

2.7. Simulated Annealing

2.7.1. Introduction

Simulated Annealing is a method for solving unconstrained and bound-constrained optimization problems. The method models the physical process of heating a material and then slowly lowering the temperature to decrease the defects, thus minimizing the system energy.

An annealing schedule is selected to systematically decrease the temperature as the algorithm proceeds. As the temperature decreases, the algorithm reduces the extent of its search to converge to a minimum.

The procedure starts from an initial solution x_0. At each iteration with each temperature fixed, it moves to other neighbouring solutions, as determined by a neighbourhood function $neighbour(x)$. Typically, the procedure only moves to solutions that are better than the current

one. Howerver, it also accepts, with a certain probability, configurations that raise the objective. By doing so, the algorithm avoids being trapped in a local minimum, and can explore globally for more possible solutions.

Here is the general Pseudocode for the temperature stopping condition.

$T = T_0$; $x = x_0$ $(\in S)$; $min = \infty$;
while $(T > T_1)$ {
  for $i = 1, N$ {
    $x' = neighbor(x)$;
    if $(cost(x') < cost(x))$ $x = x'$;
    else if $(rand() < exp\{(cost(x) - cost(x'))/T\})$ $x = x'$;
    if$(cost(x) < min)$ $min = cost(x)$, $s = x$;
  }
  if (Min was not modified in the above loop) $T = T * \alpha$;
}

where α is the cooling constant, x_0 is the initial solution. For different stopping condition like time limitation, we just need to calculate the time taken to execute the algorithm and change the condition of the while loop shown in the figure above.

2.7.2. Problem adaptation

Neighbourhood structure:

The aim is to generate as many neighbouring solutions satisfying the factory capacity constraint as possible, while still retaining the randomness in the generation to explore for more possible configurations.

Given a solution, we would know the amount of crops collected each day (only harvesting day are considered). Let F_i and f_j be fields in days with the maximum and minimum produce respectively. Then, D_i and d_j are $d(F_i)$ and $d(f_i)$. We choose the field with $max\big(d(F_i)\big)$ and change its schedule randomly. At the same time, choose randomly any field in the solution and re-schedule its harvesting day to be the same as that with the $min(f_i)$.

Intuitively saying, after the modification, D_i is decreased while d_i is increased. Thus $D_i - d_i$ is smaller than its original value, note that the objective is to minimize this difference.

Here is the Pseudocode for this neighbourhood structure:

```
Function neighbour(currentSol: List)
distinct = Sort and Remove Duplicates from currentSol
days = amountCrops(currentSol, amount)
largest_day = distinct[IndexOf Maximum Value in days]
smallest_day = distinct[IndexOf Minimum Value in days]
largestFields = List of (index, amount) for each field if harvested on
largest_day
smallestFields = List of (index, amount) for each field if harvested on
smallest_day
Sort largestFields in Descending Order by amount
Sort smallestFields in Ascending Order by amount
```

```
        largest_index = First Index from largestFields
        smallest_index = First Index from smallestFields
        a = Random Integer between lowerBound[largest_index] and
upperBound[largest_index]
        b = Random Integer between 0 and N-1
        Initialize newSol as an Empty List
        For each i in range N:
                If i equals largest_index:
                        Append a to newSol
                Else:
                        Append currentSol[i] to newSol
        newSol[b] = newSol[smallest_index]
        Return newSol
```

After creating the new neighbour, the algorithm shifts it, if necessary, to stay within bounds $[s(i), e(i)]\ \forall i = 1, N$. Each infeasible component of the new neighbour is shifted to a value chosen uniformly at random between the violated bound and the (feasible) value at the previous iteration.

Initial solution:

Experiments with different ways to generate neighbours like randomly chosen a harvesting day to assign for each field have shown the following. With a large enough input and a badly chosen initial solution, the algorithm couldn't explore any configurations that meet the given constraints. Therefore, no improvements are achieved. This calls for the design of the initial solution that will, in tandem with the neighbourhood structure, give us configurations satisfying the constraints.

We will use Contraint Programming to create a valid initial solution that meets the constraint requirements. The modelling of the problem for this design is the same as that in the Contraint Programming section of this report.

Other main components:

- The acceptance function:

$$P(\text{accept new}) = \begin{cases} \exp(-\frac{\Delta}{T_k}), & \text{if } \Delta \geq 0 \\ 1, & \text{if } \Delta < 0 \end{cases},$$

where $\Delta = f(\text{new}) - f(\text{old})$ is the change in objective function f which is to be minimised and T_k is a strictly decreasing positive sequence with $\lim_{k\to\infty} T_k = 0$.

- The cooling schedule:

$$T = T_0 \times 0.95^k$$

Where T_0, T are the temperature in the $k-1\ and\ k$ iterations respectively and 0.95 is the cooling constant.

- The stopping condition: Time limit.

- Number of attempts for each iteration where the current temperature is fixed: This depends on the scope of the input, the larger this number is, the wider the range of solutions the algorithm can explore.

Interactions between components

Let the computing time be 6s. We will look at how the algorithm works for a input of 100 fields with different initial temperatures, cooling schedules and number of attemps for each iteration.

- *Initial temperature = 100, cooling constant = 0.95, number of attemps = (25, 100)*

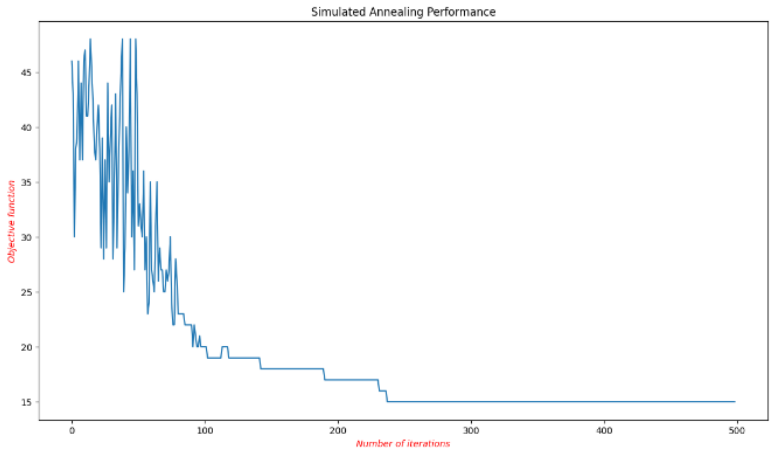

Figure 1: no_attemps = 25

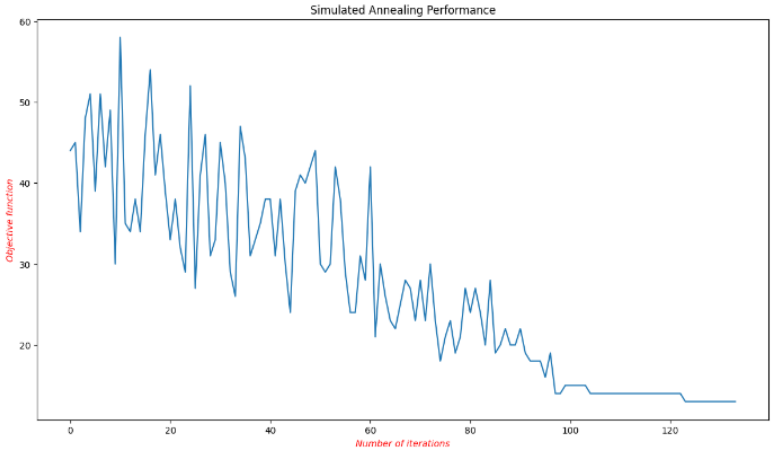

Figure 2: no_attempts = 100

The two graphs show that with a smaller number of attemps, the speed of convergence increases. As the computer can iterate several times more before the time limit is met, the temperature drops much faster. Still, the values of the objective function found are 15 and 13 for figure 1 and figure 2 respectively. This indicates that with a larger number of attempts, the algorithm can explore more solutions and produce better results even with the same time limit, while it is also unclear whether convergence is achieved or not due to the time limitation. Better solutions await.

- *Initial temperature = (25, 100), cooling constant = 0.95, number of attemps = 100*

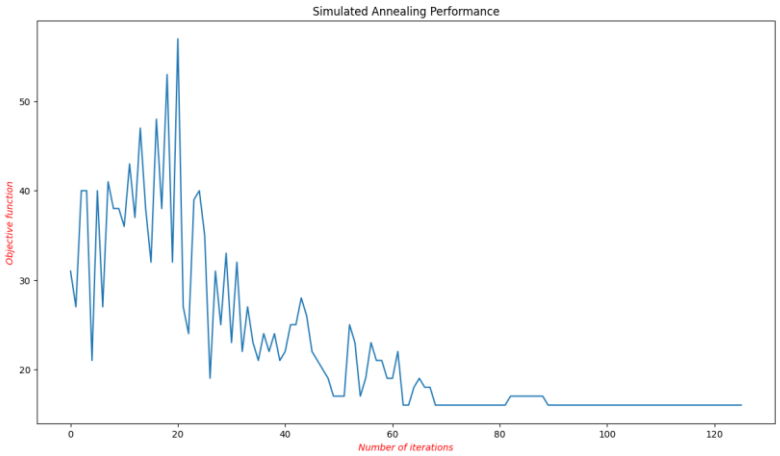

Figure 3: intial_temp = 25

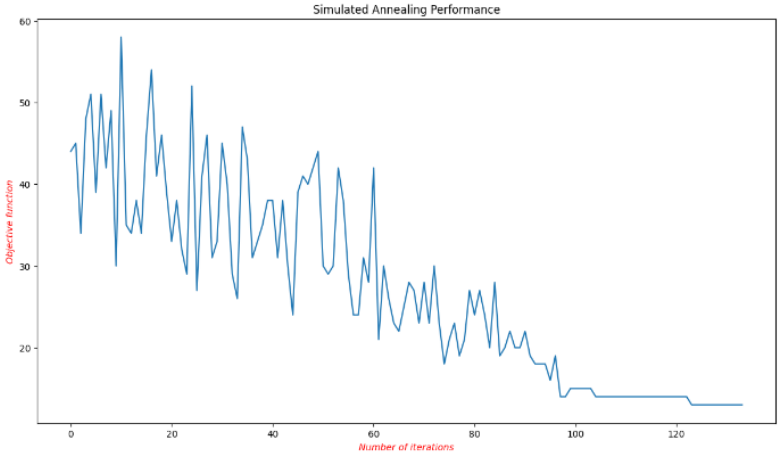

Figure 4: initial_temp = 100

The smaller the initial temperature is, the less time it takes for the algorithm to reach convergence. By setting it small, we could achieve results relatively fast. However, the solution isn't better when compared to when the temperature is set larger, as shown by the objective values in the two figure (16 and 13 for figure 3 and figure 4, respectively).

- *Initial temperature = 100, cooling constaint = (0.5, 0.95), no_attempts = 100*

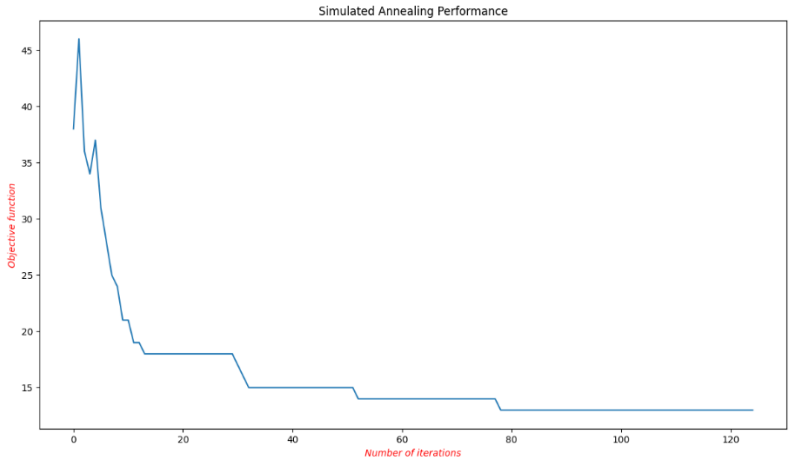

Figure 5: cooling_const = 0.5

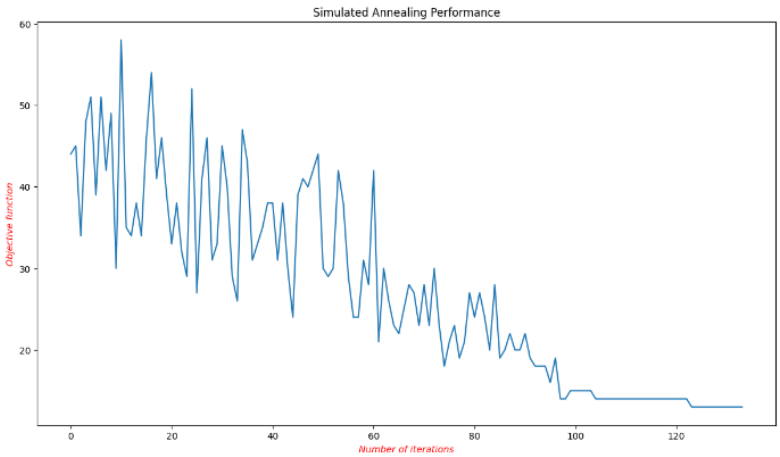

Figure 6: cooling_const = 0.95

The speed of convergence increases as does the decreasing rate of the temperature.

Advantages:

- The algorithm can deal with highly nonlinear models with many constraints. It is a robust and general technique.
- The flexibity of the algorithm and its ability to escape local optima to approach global optimum.

Disadvantages:

- The tradeoff between the quality of the solutions and the time required to compute them.
- The precision of the components used in implementation of the algorithm for a particular problem can have a significant effect upon the quality of the outcome.

3. Results

MAXIMAL TOTAL AMOUNT OF PRODUCT HARVESTED

Performance Analysis (Optimal Solution)

Size of test case	10	100	1000	5000	10000
Greedy Approach	58	1020	48905	128545	254857
Local Search	58	1020	49589	128868	255386
Constraint Programming	58	1020	49589	128868	255386
Integer Programming	58	1020	49589	128868	255386

Performance Analysis (Time Taken)

Size of test case	10	100	1000	5000	10000
Greedy Approach	0.49s	0.53s	1.38s	5.63s	11.03s
Constraint Programming	0.32s	0.49s	3.23s	9.8s	21.08s
Integer Programming	0.68s	1.21s	2.97	17.63s	65.78s
Local Search	0.76s	1.33s	4.77s	46.22s	125.40s

MINIMAL DIFFERENCES IN AMOUNT OF PRODUCT COLLECTED BETWEEN HARVESTING DAYS

Performace Analysis (Optimal Solution)

Size of test case	10	100	1000	5000	10000
Branch and Bound	1	NaN	NaN	NaN	NaN
Meta-Heuristic	1	13	9	89	54

Performance Analysis (Time Taken)

Size of test case	10	100	1000	5000	10000
Branch and Bound	0.01s	Exceeded	Exceeded	Exceeded	Exceeded
Meta-Heuristic	6s	120s	1000s	6000s	10800s

4. Conclusion

MAXIMAL TOTAL AMOUNT OF PRODUCT HARVESTED

Local Search takes a lot of time to execute. However, it is relatively stable and provides us with a very good solution because it did thousands of randomizations.

When the size of the field is small enough, it's hard to see the difference between the algorithms, but when we increase the size of the test case, the difference becomes visible.

The Greedy Algorithm, although produces the best scheme for input of size 10, performs worse when the size increases. However, it runs very fast.

The Constraint Programming and Integer Programming algorithms share a relatively equivalent running time. Both perform quite good to find the best scheme.

If you need a quite good solution or a reasonable plan in a short time owhile not considering too much about the constraint, then you should choose the Greedy Algorithm.

If you need a stable solution and don't care too much about the execution time, you should choose the Local Search, which can always give you near-best solutions.

If you need a near-optimal solution in an acceptable interval of time and the test cases are not very large, then you should go for CP and IP.

MINIMAL DIFFERENCES IN AMOUNT OF PRODUCT COLLECTED BETWEEN HARVESTING DAYS

Due to the nature of the pruning rules, the Branch-and-Bound algorithm fails to eliminate partial solutions not leading up to the optimal objective value. As a result, it couldn't handle inputs with size $\geq \mathbf{100}$.

The Simulated Annealing algorithm proves to be highly flexible and can produce near-optimal solutions to all inputs. Still, further adjustments should be made to lower the trade-off between the quality of solutions and time taken to compute them.

5. References

[1] David R.Morrison, Sheldon H.Jacobson, Jason J.Sauppe, Edward C.Sewell; "*Branch-and-bound algorithms: A survey of recent advances in searching, branching and pruning*.", Discrete Optimization, Vol.19, pp. 79 – 102, February 2016.

[2] Lyndon Martin Wendell Beharry, "*What are the limitations and challenges of branch and bound?"*

URL: *https://www.linkedin.com/advice/1/what-limitations-challenges-branch-bound-skills-linear-programming#what-are-the-limitations-of-branch-and-bound*

[3] Subham Datta, "Branch and Bound Algorithm", August 2023

URL: https://www.baeldung.com/cs/branch-and-bound

[4] Yoichiro Nakakuki and Norman M.Sadek, *"Increasing the efficiency of simulated annealing seach by learning to recognize (un)promising runs"*, Twelfth National Conference on Artificial Intelligence, pp. 1316-1322, September 1994.

[5] Franco Busetti, *"Simulated Annealing Overview"*, 2003

[6] MathWorks, "*What is Simulated Annealing?" and other Related Topics.*

URL: *https://www.mathworks.com/help/gads/what-is-simulated-annealing.html*