Name: Licheng Yan
Date: 12th Mar 2023

Introduction

ANLS is a heuristic search algorithm that applies destroy and repair function to the current solution state. At the end of every iteration, the objective function will be evaluated. Should the value be better then the current solution, the current state will be accepted and a new iteration will begin. Should the solution be worse off, the current solution will be discarded and the iteration will run once more with a new destroy and repair function.

For every iteration, the destroy and repair function is selected based on a roulette possibility. The possibility changes with each iteration depending on the outcome and this is then adjusted accordingly. Moreover, we also have a control parameter "lambda" which indicates the learning rate of the ALNS. A value close to zero will mean that the possibility of the weights getting updates is small (history dependent). Making the system "rigid". A value of lambda closer to zero will mean that the possibility of having the weights updated is high, making the system softer (history independent).

Files submitted to elearn:

1. A2-2023-Licheng_Yan.docx
2. gsp.py
3. alns_main.py

Construction Heuristic

The purpose of a construction heuristic is meant to initialise the solution such that we can begin the ALNS search. Theoretically speaking, a random initialisation is fine since the ALNS will do the search and eventually come to a good result. This is also condition on having good destroy and repair functions.

However, having a good construction heuristic means that we start off with a better solution state and this also means that we will have to run less iterations of the ALNS subject on having good destroy and repair functions.

Construction Heuristic for assignment

The following construction heuristics is used:

1. Rearrange all the workers, with the cheapest worker on the very top.
2. Shuffle the existing unassigned tasks.
3. For every single task, try to assign to first worker at the top of the list (since he/she the cheapest).
    a. If "assign" is possible, assign the task to the worker.
    b. Update the schedule blocks of that worker.
    c. Update the total hour of the worker.
4. If the assign is not possible, then try to assign to the next cheapest worker in line.
5. If, after iterating through all the workers, we are still not able to find a worker to assign the task:
    a. Append the task to "unassigned" list.
6. Proceed to assign the next task on the list.
7. Repeat until there is not task left on the list.

The earlier mentioned construction heuristics is a naive form of greedy algorithm. This is because at any point of time, it assume that by selecting the cheapest worker to try and assign the task to, we will get the lowest objective value. However, it is naïve in the sense that it does not consider the time slot since all tasks have different time slot. Having a job on time slot 4 and 20 means that you have to pay the worker for 17 hours instead of just 2 hours. The above construction heuristics does not take that into consideration.

Still this naïve construction heuristics gives us a reasonably good start for the ALNS algorithm to perform its wonders on the solution.

Destroy Methods

The purpose of the destroy method is meant to alter the solution state such that it can then be repaired later by the repaired method in hopes of achieving a better solution state. Theoretically speaking with no limit on the iterations, we can simply just reply on the random destroy method and eventually, we will come to a good solution after numerous tries. The purpose of designing the destroy method however, is simply to provide the ALNS with a "compass" that "guides" it in the right direction which then saves us a lot of time to do fruitless search. Many of these destroy methods are actually based on intuition and some simple logic.

Degree of destruction

Taking motivation from the TSP example, we have implemented the degree of destruction. More details will be explained in the final section of the report. The reason for adding the degree of destruction is such that we don't need to always destroy a fixed number of workers, but instead, a percentage of workers. This will allow the ALNS to be more adaptive when handling different number of workers.

Method 1: Random worker removal

This is the simplest destroy method. **<u>We simple choose a random number of workers who have tasks assigned.</u>** Then we relieve them of all the tasks followed by a repair function which reassigns the tasks. This method works well and leads to the improvement of the solution simply because random tries sometimes leads to better solution.

Method 2: Highest rate worker removal

In this method, **<u>we target the workers with the highest rate</u>**. This is because tasks parked under them are charged high cost. Thus, intuitively we do not want to assign tasks to them if possible. For this method, we simply rearrange the most expensive workers at the top, then we select the top "x" number of workers which is determined by the degree of destruction.

<u>Note: In hindsight, this destroy operator may not be too productive since it always removes the same top few workers which are ranked the most expensive.</u>

Method 3: Four hours worker random removal

In this method, we will **<u>target workers with blocks less than 4 hours</u>**. This is because such cases are suboptimal since we pay them for the hours that they are not working. The purpose of this destroy operator is to totally remove the task of such workers and hopefully reassign them in a more optimal position during the repair operation. Each time the destroy operator is called, we will only destroy 1 worker by taking away all his tasks and freeing him up.

Method 4: Single task random removal

Earlier we have a random destroy operator that removes all the task from a worker. However, such large-scale operation may not result in good outcome since some tasks in optimal positions will then be rearranged. **Thus, instead of removing all the tasks of a worker, we will now randomly select workers and randomly choose a day and remove a random task from the worker**. The purpose of such heuristics is meant to rely on fine adjustments that may lead to a slightly better solution state.

At times, the random destroy will be able to lift me out of a local optimal due to the limitation and considerations of my destroy function.

Method 5: Shorten Block removal front

One of the interesting finding is that certain tasks can be reallocated to workers who have 1,2,3 hour blocks. This results in a reduction of cost since the worker has to be paid a minimum of 4 hours anyways. Furthermore, the worker with task removed from the front end will have their blocks shortened resulting in lower objective value.

Thus, in this destroy operator, we first identify all the workers who have blocks longer then 4 hours. Thereafter, **we will randomly select a worker and remove a single task with the earliest timing which shortens their block from the left-hand side**. We hope that the repair function will later assign this task to another worker who have 1,2,3 hour blocks.

Method 6: Shorten Block removal back

This method is the exact same as method 5. **Except in this case, we remove a task from the latest hour which then shorten the block from the right-hand side**. We then leave the single task to the repair function which hopefully assigns it at a more optimal place with no cost.

Repair Methods

Repair methods are meant to assign the tasks that are removed by the destroy operator to places that are hopefully more optimal and lower cost.

Method 1: Cheapest worker first repair

This is one of the most naïve approaches and follows the same idea as the construction heuristic. **We arrange the cheapest worker at the top** and then try to assign them tasks starting with the topmost worker. This is in hopes that tasks get assigned to workers with lower costs and results in a lower objective function.

Method 2: Lowest hour worker first repair

The naïve approach in the method 1 is not able to account for cases where we have workers who only have 2 tasks but are still paid 4 hours. These hours are already paid for but not utilised in anyway. One possible naïve simple heuristic to identify such worker is to look at the total hours of a worker. **We will now put workers with low total hours assigned at the top to give them priority for tasks assignment**. This is a simple operation and does not require a lot of computation power. Yet, it will give good opportunity to assign tasks to workers with little total hours.

Method 3: Lowest task worker first repair

In method 3 we look at the total hours of a worker. However, there are cases where the total hours are 14 but there are only 2 tasks assigned. This means that now we have a lot of gaps that are paid for but not utilised. **Thus, we will now arrange the workers with the lowest number of assigned task at the top**. We then follow this with the assignment of tasks.

Method 4: Highest available worker first repair

Independent of the method 1, 2 & 3 is the Method 4. In method 4, **<u>we do a simple count of the total available hours offered by each worker</u>**. We then rearrange the workers such that the worker with the highest number of available hours is at the top for task assignment later. This is a simple heuristic since such workers can accommodate more tasks and are more flexible. As much as possible we would like to make use of them instead of just assigning a single task to a work who might then be paid 4 hours.

Method 5: Random worker first repair

This is a very simple repair function in which I **<u>simply shuffle the workers randomly and just try to assign them tasks that were removed by the previous destroy operator</u>**.

<u>At times, the random repair will be able to lift me out of a local optimal due to the limitation and considerations of my repair function.</u>

Weight Adjustment Strategy

Learning rate adjustment

For the learning rate adjustment, after multiple experimentations, a value of 0.3 is ultimately chosen. This is because a high value of ~0.8 will result in a more rigid ALNS which gives less update of the weights further down the iterations. Since the nature of my repair and destroy functions are different, their utility will change as the solution state changes. Some operators that are greedy may be useful at the start of the iterations. However, when the solution plateaus, I would then like to give more attention to the random destroy & repair operators. Thus, I would like my ALNS to be more flexible in that sense. It should have the flexibility to change its preference of operators at any stage.

Furthermore, I am using hillclimbing(), therefore the state of my solution is “locked” in until a better result is obtained. This allows me the safety of making bad choices for exploration purpose.

However, if simulated annealing is used, having the low value of 0.3 would be risky since simulated annealing allows suboptimal solutions to be accepted at certain rates.

Weights adjustment

For the weights adjustment strategy, after multiple tries, [100, 14, 4, 1] is being chosen. A weight of 100 is assigned when a new global optimum is discovered. The difference in magnitude should allow my ALNS to respond faster when the utility of my destroy and repair function changes as the solution state evolves. For example, at later stages where random destroy and repair becomes more useful, I would like to quickly add weight to them.

Plots and results

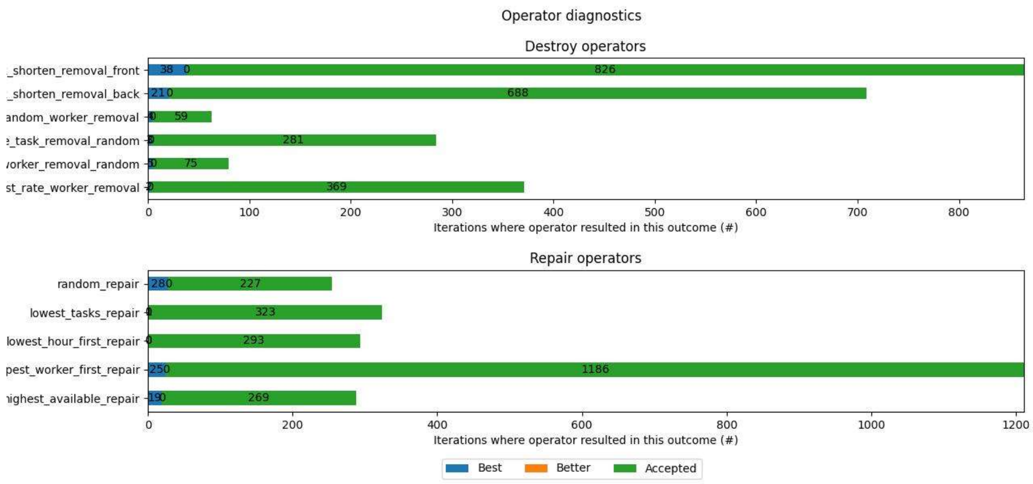
Below are the plots and results from the S3 instance:

Operator diagnostics

From the results show, it appears that not all of my operators are actually effective. This graph is particularly useful for me to diagnose and improve on the solution. If this is in a work environment, I will use this information to question why certain operators are not so useful and then modify them accordingly.

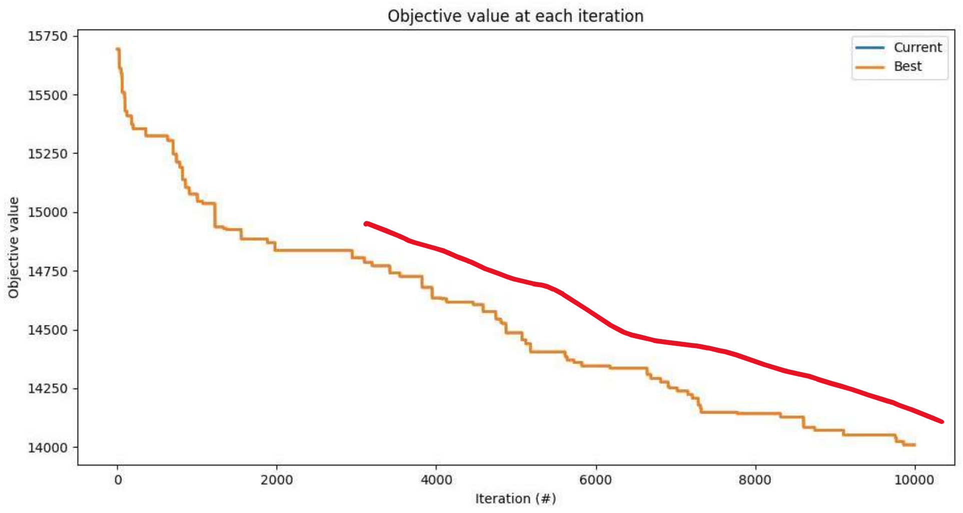
For the destroy, it seems that the "block shortening" operators are doing quite the good job.

For repair, it is interesting to note that "random repair" carries a significant proportion of credit alongside the "cheapest worker" and "most available worker" repair.



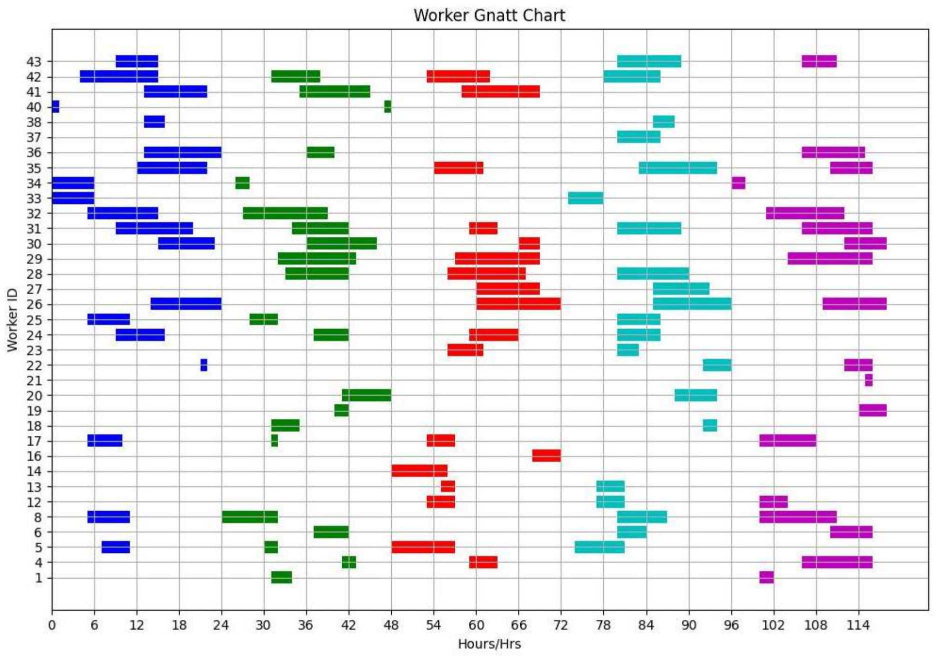
Objective value plot

Amongst the plots, this is one of the interesting one. Shown from the "near" linear line (in red) it shows that my ALNS is actually still capable of further decreasing the objective value if I allow it to run for even more iterations. However, it also shows that while the solution is capable of decreasing even further, I need to further improve the heuristics such that my search yield more frequent fruitful results which gets me optimal solutions faster.

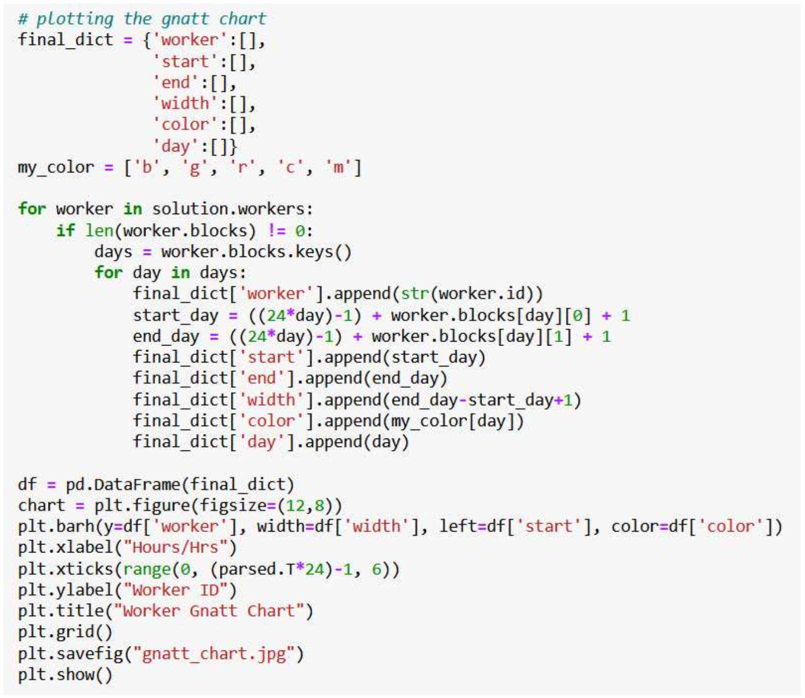


Gantt Chart

Here is the gnat chart plotted. It was not done with libraries. Instead, we use simple matplot lib.



This Gantt chart is actually another useful tool to help diagnose and inspire more formulations of destroy and repair operators. As shown from the image, there are still some instance where we see a single 1-2 hrs block which are paid with 4 hours of salary. These can either be removed or filled up which results in lower objective value. Given enough iterations, we will be able to reach the optimum. However, too many iterations take too much time, thus this is a signal for us to improve our destroy and repair operators.



```
# plotting the gnatt chart
final_dict = {'worker':[],
              'start':[],
              'end':[],
              'width':[],
              'color':[],
              'day':[]}
my_color = ['b', 'g', 'r', 'c', 'm']

for worker in solution.workers:
    if len(worker.blocks) != 0:
        days = worker.blocks.keys()
        for day in days:
            final_dict['worker'].append(str(worker.id))
            start_day = ((24*day)-1) + worker.blocks[day][0] + 1
            end_day = ((24*day)-1) + worker.blocks[day][1] + 1
            final_dict['start'].append(start_day)
            final_dict['end'].append(end_day)
            final_dict['width'].append(end_day-start_day+1)
            final_dict['color'].append(my_color[day])
            final_dict['day'].append(day)

df = pd.DataFrame(final_dict)
chart = plt.figure(figsize=(12,8))
plt.barh(y=df['worker'], width=df['width'], left=df['start'], color=df['color'])
plt.xlabel("Hours/Hrs")
plt.xticks(range(0, (parsed.T*24)-1, 6))
plt.ylabel("Worker ID")
plt.title("Worker Gnatt Chart")
plt.grid()
plt.savefig("gnatt_chart.jpg")
plt.show()
```

Additional points

I have a dynamic degree of destruction that decays with time. During the experimentation, I realised that a large value of destruction will give me good progress at the start, but eventually the objective value plateaus. I realised that a small destruction will give good end results but required more iterations (> 10000) since it is a form of fine tuning where we try smaller perturbations to the state.

To combat this problem, I created a function that returns the degree of destruction based on the time elapsed. At the start it returns 0.25 (25%), however, towards the ends, it returns (0.03%). This give me the advantage of having a fast exploration at the initial stages and then transits to fine and small perturbation at later stages that allows me to jump to better solution states.

When we destroy a significant proportion of tasks, we also need to assign them all back, this is something that is hard to control. Thus, with the fine adjustment using lower destruction, we have a better chance of getting better results.