Erasmus University Rotterdam
Erasmus School of Economics
International Bachelor Econometrics and Operations Research

Multi-Neighborhood Simulated Annealing for the Machine Reassignment Problem

Donna Zegwaart (528847dz)

Supervisor:	P.J. Correia Duarte
Second assessor:	dr. T.A.B. Dollevoet
Date final version:	2nd July 2023

Abstract

The Machine Reassignment Problem (MRP) improves machine usage given an initial assignment of processes on machines. A new allocation is found that minimizes machine usage costs and process move costs, subject to several usage and assignment constraints. A more efficient allocation of processes leads to less electricity usage, and an increased capacity on software applications like Google Docs. This problem was first introduced in the ROADEF/EURO Challenge 2012 by Google, and as this problem is NP-hard this paper uses two heuristics to solve the MRP. In this research, we replicate Multi-Neighborhood Local Search (MNLS) by Wang, Lü and Ye (2016), and propose a new Multi-Neighborhood Simulated Annealing (MNSA) algorithm. MNLS finds the best solution 20% of the time, and outperforms MNSA which finds the best objective value in 8% of the cases.

1 Introduction

With the quick growth of IT platforms, like ChatGPT or Google Docs, cloud computing is becoming increasingly important in daily life. It is crucial to allocate the user requests of these softwares over the available computing resources in an efficient manner, to ensure low electricity costs and increase capacity (Canales, Rojas-Morales & Riff, 2020). This NP-hard allocation problem is the Machine Reassignment Problem (MRP), and it includes multiple NP-complete subproblems, like bin-packing (Gabay & Zaourar, 2016). To keep these services available to a growing amount of users, we need heuristics that relatively quickly move towards an optimal solution (one that sorts the user requests in the most cost-efficient way over all available computing resources). This problem is the topic of the ROADEF/EURO Challenge 2012 by Google, and many different heuristics related to neighborhood searches have been proposed to optimize the MRP (Murat Afsar, Artigues, Bourreau & Kedad-Sidhoum, 2016). Wang et al. (2016) introduce a new algorithm, Multi-Neighborhood Local Search (MNLS), using local and tabu search optimization, and three different neighborhoods. MNLS performs well when compared to existing literature, and this research replicates the algorithm and uses it as a benchmark. This paper introduces a new algorithm that performs Multi-Neighborhood Simulated Annealing (MNSA) using the promising neighborhoods from MNLS. This leads to the following research question: does MNSA outperform MNLS in the context of the MRP?

We find that MNLS is more robust than MNSA, and outperforms MNSA for most instances. However, both algorithms manage to find the best-known solution 20% and 8% of the time, for MNLS and MNSA respectively.

In this research, the existing literature is discussed in Section 2, the problem description in Section 3, the methodology in Section 4, the results in Section 5, and lastly a conclusion and discussion are given in Section 6 and 7.

2 Literature Review

The MRP is a relatively new problem, and only frequently appears in literature since the ROADEF/EURO challenge 2012 by Google. Many different algorithms have been proposed, most of which relate to various kinds of neighborhood searches. For instance Gavranović, Buljubašić and Demirović (2012) propose a Variable Neighborhood Search (VNS), Wang et al. (2016) present a Multi-Neighborhood Local Search algorithm (MLNS), and Masson et al. (2013) introduce a multi-start iterated local search using two neighborhoods. In literature, the two neighborhoods that are most often used are swap and shift (Canales et al., 2020) and (Murat Afsar et al., 2016). Swap switches two processes of different machines, and shift moves a process from one machine to another. Lopes, Morais, Noronha and Souza (2015) use variations of these two neighborhoods, namely

k–shift and k–swap, which are obtained by doing k successive shifts or swaps respectively. Others, like Turky, Sabar and Song (2017); Hoffmann, Riff, Montero and Rojas (2015), use double–shift and/or double–swap where only 2 processes are simultaneously shifted or swapped, with the MLNS from Wang et al. (2016) specifically using one–shift, two–swap, and a three–swap neighborhood where two processes on one machine are swapped with another process on another machine. VNS, which won the ROADEF/EURO challenge, also applies a Big Process Rearrangement (BRP) neighborhood, which is done by simultaneously shifting k processes to a specific machine, and shifting some processes from this specific machine to other machines. The authors state that this is especially useful for reassigning big processes in large instances (Gavranović et al., 2012).

In addition to different neighborhoods being used throughout the literature, several different local searches are also considered, like local search and simulated annealing. Lopes et al. (2015); Gavranović et al. (2012); Masson et al. (2013) all use local search effectively, while Wang et al. (2016) use a combination of local search and tabu search. Furthermore, Portal, Ritt, Borba and Buriol (2016) use simulated annealing (SA) in combination with the two-swap and one-shift neighborhoods, by selecting a random neighbor from one of the two neighborhoods with a given probability. Despite only using the two most rudimentary neighborhoods, they were fourth in the ROADEF/EURO challenge. This shows promise for future research of SA in combination with multiple neighborhoods. Butelle et al. (2016) propose a new method that runs Adaptive Variable Neighbourhood Search on one thread, while running a Simulated Annealing based Hyper-Heuristic on another, where the threads communicate the best-known solution to each other throughout the procedure. The Simulated Annealing based Hyper-Heuristic combines different heuristics and acceptance criteria, and considers the shift and swap neighborhoods. Turky, Sabar and Song (2018) introduce an algorithm that simultaneously runs SA on different initial solutions, using four different moves: shift, swap, double–swap, and double–shift.

3 Problem Description

The machine reassignment problem makes a more efficient allocation of processes over machines starting from a given initial allocation. This reallocation is optimized using five constraints, by minimizing five costs. The initial solution is denoted as Map_0 and holds a map connecting process $p \epsilon P$ to the machine $m \epsilon M$ this process is running on. Then $Map(p)$ equals the current machine m of a process p. The constraints and costs from the GOOGLE/ROADEF Challenge are described in Sections 3.2 and 3.3 respectively. Table 1 shows an overview of all the constants of this problem and can be found in the Appendix.

3.1 Variables

The two variables $U(m,r)$ and $TU(m,r)$ measure the usage and transient usage on a machine and make up the basis of the constraints and costs. Transient usage measures the resources still used on the initial machine of a process, after moving this process away. For instance, for some processes disk space is used on both machines when it's moved. For both variables, *bool* refers to an indicator function, which is one if the statement is true and zero otherwise.

U(m,r) The combined usage $R(p,r)$ of resource $r \epsilon R$ from processes $p \epsilon P$ running on machine $m \epsilon M$.
$U(m,r) = \sum_{p \epsilon P} bool(Map(p) = m) * R(p,r)$

TU(m,r) The transient usage of resource $r \epsilon R$ on machine $m \epsilon M$. The transient usage equals the requirement $R(p,r)$ if a moved process $p \epsilon P$ initially on machine $m \epsilon M$ has transient usage for a resource $r \epsilon TR$, and is zero when a resource $r \epsilon R - TR$ has no transient usage.

$$TU(m,r) = \begin{cases} \sum_{p \epsilon P} bool(Map_0(p) = m \wedge Map(p) \neq m) * R(p,r) & r \epsilon TR \\ 0 & r \epsilon R - TR \end{cases}$$

3.2 Constraints

Five constraints have to be satisfied in order for the MRP to be feasible, they are described in this section using the variables from Section 3.1. The five constraints are:

Capacity constraints A machine $m \epsilon M$ has to have enough capacity $C(m,r)$ for the total usage $U(m,r)$ of resources $r \epsilon R$ on m.
$\forall m \epsilon M, r \epsilon R, U(m,r) \leq C(m,r)$

Conflict constraints Processes $p \epsilon P$ of the same service $s \epsilon S$ must run on distinct machines in the current allocation Map.
$\forall s \epsilon S, (p_i, p_j) \epsilon s, p_i \neq p_j \Rightarrow Map(p_i) \neq Map(p_j)$

Spread constraints The number of different locations $l \epsilon L$ where processes $p \epsilon P$ of the same service $s \epsilon S$ run in the current allocation Map has to be greater than or equal to a given minimum S_{min}.
$\forall s \epsilon S, \sum_{l \epsilon L} \min(1, |\{p \epsilon s | Map(p) \epsilon l\}|) \geq S\_min(s)$

Dependency constraints If a service s^a depends on a service s^b, both in the service dependencies set SD, then each process $p^a \epsilon P$ of service s^a should run in the neighborhood NE of a process $p^b \epsilon P$ of service s^b in allocation Map.
$\forall (s^a, s^b) \epsilon SD, \forall p^a \epsilon s^a, \exists p^b \epsilon s^b \Rightarrow NE(Map(p^a)) = NE(Map(p^b))$

Transient usage constraints For resources $r \epsilon TR$ which need transient usage, the usage $U(m,r)$ already on machine $m \epsilon M$ plus the transient usage $TU(m,r)$ on m can not exceed the capacity $C(m,r)$ of machine m for resource r
$\forall m \epsilon M, tr \epsilon TR, U(m,tr) + TU(m,tr) \leq C(m,tr)$

3.3 Costs

Next, we explain the costs and objective function, again using variables from Section 3.1. First, the five elements of the objective are introduced separately, and next the full objective function is given. For each cost, $Map_0(p)$ denotes the initial machine of a process $p \epsilon P$, and the new machine p is moved to is specified as $Map(p)$. The five costs are:

Load cost The load cost of each resource $r \epsilon R$ is the usage $U(m,r)$ of a machine $m \epsilon M$ that exceed the given safety capacity $SC(m,r)$.
$f_1(r) = \sum_{m \epsilon M} \max(0, U(m,r) - S(m,r))$

Balance cost As it is useless to have for instance CPU available, but no RAM, there has to be a certain balance between resources $r_1^b, r_2^b \epsilon B$. This means the available capacity of resources r_1^b and r_2^b on machine $m \epsilon M$, i.e. capacity $C(m,r)$ minus the usage $U(m,r)$, has to be balanced with target ratio $target^b$.
$f_2(b) = \sum_{m \epsilon M} \max(0, target^b * (C(m,r_1^b) - U(m,r_1^b)) - (C(m,r_2^b) - U(m,r_2^b))$

Process move cost The total cost of moving processes $p \epsilon P$ from their initial machine $Map_0(p)$ to their new machine $Map(p)$. *bool* is an indicator function, where one corresponds to p being assigned to a new machine, and zero otherwise. $PMC(p)$ denotes the cost of moving p.
$f_3 = \sum_{p \epsilon P} bool(Map_0(p) \neq Map(p)) * PMC(p)$

Service move cost To balance moves among services $s \epsilon S$ the maximum number of moved processes $p \epsilon P$ of one service is punished.
$f_4 = \max_{s \epsilon S}(|\{p \epsilon S | Map_0(p) \neq Map(p)\}|)$

Machine move cost The total cost of moving processes $p \epsilon P$ from their initial machine $Map_0(p)$ to their new machine $Map(p)$, where MMC is the moving cost.
$f_5 = \sum_{p \epsilon P} MMC(Map_0(p), Map(p))$

Objective function All the costs are assigned a weight wt, which represents the punishment of the corresponding cost. The goal is to find the solution with the minimum objective value.
$f(Map) = \sum_{r \epsilon R} wt_1(r) * f_1(r) + \sum_{b \epsilon B} wt_2(b) * f_2(b) + wt_3 * f_3 + wt_4 * f_4 + wt_5 * f_5$

4 Methodology

4.1 Neighborhoods

This paper uses the following three neighborhoods: one-shift, two–swap and three–swap. These neighborhoods are also used in MNLS from Wang et al. (2016), which is a benchmark for this paper. All possible moves within a neighborhood with Map_0 as a starting solution are denoted by $MV(Assignment)$, with mv indicating a move within this set of moves. Then a neighborhood is defined as: $N(Map) = \{Map \oplus mv | mv \epsilon MV(Map)\}$. The neighborhoods are given below, using the formulation from Wang et al. (2016):

One-shift Moves a process $p \epsilon P$ to a machine $m \epsilon M$.
$MV_1(Map) = \{mv_1(p, m) | \forall p \epsilon P, m \epsilon M, Map(p) \neq m\}$

Two-swap Swaps a process $p_i \epsilon P$ with a process $p_j \epsilon P$ and reassigns these processes to machines $Map(p_j)$ and $Map(p_i)$ respectively.
$MV_2(Map) = \{mv_2(p_i, p_j) | \forall p_i, p_j \epsilon P, Map(p_i) \neq Map(p_j)\}$

Three-swap Here processes $p_i, p_j \epsilon P$ are swapped to machine $Map(p_k)$, and process p_k is swapped to machine $Map(p_i)$.
$MV_3(Map) = \{mv_3(p_i, p_j, p_k) | \forall p_i, p_j, p_k \epsilon P, Map(p_i) = Map(p_j) \wedge Map(p_i) \neq Map(p_k)\}$

The neighborhood partitioning consists of three different techniques by (Wang et al., 2016). Firstly, the neighborhoods are randomly partitioned into smaller parts of equal size. Then when researching a neighborhood, the optimal move is found by iterating over all the smaller partitioned neighborhoods and finding the best move within those. This means the algorithms lose some of their search effectiveness but gain efficiency. Therefore, this has to be a careful trade-off that is optimized using parameter tuning. Wang et al. (2016) find that the speed of local search is too low for neighborhoods larger than 10^6 moves, and that the search efficiency becomes too bad with neighborhoods smaller than 10^5 moves. Thus the partitioned neighborhoods of N_1, N_2 and N_3 have sizes $\max(1, |P|x|M|/10^5)$, $\max(1, |M|/100)$ and $\max(1, |M|/50)$ respectively. In N_1 the processes are partitioned, whereas in N_2 and N_3 the machines are split up.

Next to the neighborhood partitioning technique described above, we decrease N_2 and N_3 with two additional strategies. Firstly, at most ten processes are selected randomly on all machines within the partition. Furthermore, we exclude non-promising pairs of machines. Specifically, two machines are excluded if the lower bound introduced by Portal et al. (2016) of the load cost and balance cost is equal to the sum of the load cost and balance cost on these machines. As in this scenario, the odds of finding a good swap are quite low, the machine pairing is not considered within the neighborhoods.

4.2 Multi-Neighborhood Local Search

Since this paper uses MNLS by Wang et al. (2016) as a benchmark, this section shortly describes the method. A multi-neighborhood search is done using the neighborhoods from Section 4.1, where N_1, N_2 and N_3 are sequentially explored. The exploration of a neighborhood is stopped when the objective value stops making sizeable improvements, when this occurs the next neighborhood is investigated.

Within each neighborhood we perform a move selection heuristic, which first considers the best feasible move of a process that has not been moved in the past $|P|/100$ iterations, like tabu-search. However, if a move exists that moves a process which has been used in the past iterations, but gives a better objective than previously found in the entire local search algorithm, the move is accepted. If this is not the case, the algorithm checks whether the best feasible and non-tabu move has a better objective value than the best one found so far within the current neighbohood, if so this accepted and else we accept the best infeasible move. To fix this infeasible move one of two repair strategies is done. Specifically, these repair strategies can fix the capacity constraints and the transient usage constraints by moving processes away from a machine where the current usage exceeds the allowed usage. If no such fix exists, we choose a random feasible move within the neighborhood.

Once all neighborhoods have been explored, we perform a number of random moves and complete another round of neighborhood search. Wang et al. (2016) follow two different time-out conditions: a 300 second time-out as the Google ROADEF/EURO Challenge prescribes, and a 3600 second time-out constraint.

4.3 Multi-Neighborhood Simulated Annealing

A disadvantage of local search is that it tends to get stuck in local minima. MNLS tries to prevent this by making random moves at various points. Another way to work around this problem is to allow for a solution to get worse, in order to escape local minima by implementing Simulated Annealing. MNSA has different cycles of random moves and within each cycle the 'temperature' is lowered, this ensures that the objective value slowly converges (Johnson, Aragon, McGeoch & Schevon, 1989), given that the parameters have been tuned correctly. Similarly to Portal et al. (2016), we use a geometric cooling cycle, which holds a constant temperature for n iterations, and then lowers this temperature with a given cooling rate r. Similarly to MNLS, a random move is chosen from either the one-shift, two-swap or three-swap neighborhood. In their paper Wang et al. (2016) highlight the importance of including three neighborhoods in their search, by showing that the combination with all three outperforms any other combination of neighborhoods. MNSA chooses a random move from two-swap with probability α, a move from three-swap with probability β, and a move from one-shift with probability $1 - \alpha - \beta$. The implemented

algorithm is shown in Algorithm 1.

Algorithm 1 Simulated Annealing with a geometric cooling cycle

```
f_best ← f(Map_0)
while time limit is not reached do
    for cycle in 1:n do
        Map ← Map' ⊕ random feasible move
        Δ ← f(Map') − f(Map)
        if Δ ≤ 0 then
            Map' ← Map
            if f(Map') < f_best then
                f_best ← f(Map')
            end if
        else
            p ← random number between 0 and 1
            if p < exp(-Δ/T) then
                Map' ← Map
            end if
        end if
    end for
    T ← rT
    if f_best has not changed for 20n iterations & number of accepted moves < 0.1% then
        T ← T_0/100                          ▷ T_0 is the initial starting temperature
    end if
end while
return f_best
```

4.3.1 Fast moves

An issue of neighborhood searches is that the feasibility and cost have to be calculated for every possible move within a neighborhood, which leads to a slow running time. This is especially important for MNSA, as the strength of simulated annealing is in making many moves quickly. In order to speed this process up, instead of calculating the objective or feasibility from the entire proposed $Map \oplus mv$, where Map represents the current allocation of machines, it is quicker to calculate the effect of just the move. Specifically, the initial objective of Map_0 can be calculated using the objective function given in Section 3.3, then for all possible subsequent moves just the cost effect of that move can be calculated. The move which gives the biggest negative cost difference is chosen. In addition, as Map is already feasible (MNSA only accepts feasible moves), the only thing that needs to be checked is whether allocating the machines and processes of the given move leads to a feasible result.

5 Results

The most commonly used dataset of the MRP consists of three smaller datasets (A, B & X) from the ROADEF/EURO challenge 2012 by Google, the datasets are randomly generated according to real-life Google statistics.

Table 1: Overview of all instances with their characteristics.

Inst.	$\|R\|$	$\|TR\|$	$\|M\|$	$\|P\|$	$\|S\|$	$\|L\|$	$\|N\|$	$\|B\|$	$\|SD\|$
a1_1	2	0	4	100	79	4	1	1	0
a1_2	4	1	100	1000	980	4	2	0	40
a1_3	3	1	100	1000	216	25	5	0	342
a1_4	3	1	50	1000	142	50	50	1	297
a1_5	4	1	12	1000	981	4	2	1	32
a2_1	3	0	100	1000	1000	1	1	0	0
a2_2	12	4	100	1000	170	25	5	0	0
a2_3	12	4	100	1000	129	25	5	0	577
a2_4	12	0	50	1000	180	25	5	1	397
a2_5	12	0	50	1000	153	25	5	0	506
b_1	12	4	100	5000	2512	10	5	0	4412
b_2	12	0	100	5000	2462	10	5	1	3617
b_3	6	2	100	20,000	15,025	10	5	0	16,560
b_4	6	0	500	20,000	1732	50	5	1	40,485
b_5	6	2	100	40,000	35,082	10	5	0	14,515
b_6	6	0	200	40,000	14,680	50	5	1	42,081
b_8	3	1	100	50,000	45,030	10	5	0	15,145
b_9	3	0	1000	50,000	4609	100	5	1	43,437
x_1	12	4	100	5000	2529	10	5	0	4164
x_2	12	0	100	5000	2484	10	5	1	3742
x_3	6	2	100	20,000	14,928	10	5	0	15,201
x_4	6	0	500	20,000	1190	50	5	1	38,121
x_5	6	2	100	40,000	34,872	10	5	0	20,560
x_6	6	0	200	40,000	14,504	50	5	1	39,890
x_8	3	1	100	50,000	44,950	10	5	0	12,150
x_9	3	0	1000	50,000	4871	100	5	1	45,457

Dataset A consists of ten instances with the number of machines ranging from 4 to 100, and the number of processes spanning from 100 to 1000. Both B and X also have ten instances, with the number of machines ranging from 100 to 5000, and the number of processes varying from 5000 to 50000. Table 1 holds the most important characteristics of the instances, with the columns listing the instances, numbers of resources, resources needing

transient usage, machines, processes, services, locations, neighborhoods, balance 'triples', and service dependencies. Instances b_7,b_10,x_7 & x_10 were found to be infeasible, due to their particularly large sizes.

Both the MNLS and MNSA algorithms are programmed in Java on a Microsoft Windows 10 computer with Intel Core i5-10500 3.10GHz and 16 GB of RAM. Every algorithm is run on only one core, and two running times are tested: the original 300s time limit of the ROADEF/EURO Challenge 2012, and a 3600s time limit to see how the algorithms converge.

5.1 Parameter tuning

We use the same parameter settings as optimized by Wang et al. (2016) for MNLS, and we tune parameters for MNSA parameter ourselves. The MNSA algorithm is similar to the two-neighborhood SA algorithm proposed by Portal et al. (2016), so we use the same instances and values of parameters for tuning. All parameter combinations are tested on all of the following instances: a1_4, a2_1, a2_2, a2_3, a2_5, b_1, and b_3. The values we evaluate are $r\epsilon\{0.91, 0.95, 0.97\}$, $n\epsilon\{10^4, 10^5, 10^6\}$, $T_0\epsilon\{10^7, 10^8, 10^9\}$ and pairings $\alpha, \beta\epsilon\{\{0.2, 0.1\}, \{0.2, 0.3\}\}$. Due to time constraints only two α, β pairings are considered, we choose the values which keep the probability of choosing a one-shift move the highest as Portal et al. (2016) find the one-shift neighborhood to be more important than the two-swap neighborhood in simulated annealing. The parameter are tuned under the 300s time constraint. Next, the relative deviation with respect to the best-known value of each instance under 300s is calculated, and we choose the parameter setting with the lowest average relative deviation. Table 2 shows the average relative deviations of the best parameter settings for three different sets of tuning instances. The average relative deviations for all instances are much larger compared to the averages without a2_1. This is due to the especially slow convergence of this instance. To prevent the settings from being skewed toward this extreme case, it is taken out of consideration. In addition, the relative deviation of b_3 is about 40 times larger than the other considered instances. As the parameter settings with and without b_3 do not differ that greatly, we end up choosing $T_0 = 10^7$, $\alpha = 0.2$, $\beta = 0.3$, $n = 10^4$, $r = 0.91$.

Table 2: Average relative deviations with respect to the best-known value for three different parameter settings, the light gray coloured boxes correspond to the best setting for the instance in the column name.

	All instances[1]	Without a2_1	Without a2_1 & b_3
$T_0 = 10^7$, $\alpha = 0.2$, $\beta = 0.5$, $n = 10^5$, $r = 0.97$	121678.88	7.44	1.14
$T_0 = 10^7$, $\alpha = 0.2$, $\beta = 0.3$, $n = 10^6$, $r = 0.95$	228463.39	7.32	1.07
$T_0 = 10^7$, $\alpha = 0.2$, $\beta = 0.3$, $n = 10^4$, $r = 0.91$	211305.48	7.33	1.01

1: Only refers to the instances used for tuning mentioned in Section 5.1

5.2 Computational results

To assess the effectiveness of MNSA compared to MNLS, every instance is run five times and the best objective is compared to the best-known solution (BKS) from the literature. Tables 5 and 6 hold the BKS, best objective value (f_{best}), the average objective value (f_{avg}), and the coefficient of variance (c_v) for all instances and both optimization methods under time-out conditions 300s and 3600s respectively. For a full evaluation, Tables 3 and 4 present the initial cost in column 2, the BKS, the best objective value (f_{best}), the differential ratio between the best solution of the corresponding algorithm and the best-know solution (r%), i.e. $\frac{f_{best}-BKS}{BKS} * 100\%$, the differential ratio between the best solution of MNSA and the best solution of the benchmark MNLS (s%), i.e. $\frac{f_{best,MNSA}-f_{best,MNLS}}{f_{best,MNLS}} * 100\%$.

5.3 Comparison MNLS and MNSA with best-known results

Table 3: Computational results of MNLS and MNSA under the 300-second time-out condition.

			MNLS		MNSA		
Inst.	Init. Cost	BKS	f_{best}	r (%)	f_{best}	r (%)	s (%)
a1_1	49528750	44306501	44306501	0.00	44307915	0.00	0.00
a1_2	1061649570	777532177	810826359	4.28	887425469	14.13	9.45
a1_3	583662270	583005717	583006137	0.00	583662270	0.11	0.11
a1_4	632499600	244875206	291202548	18.92	525791949	114.72	80.56
a1_5	782189690	727578309	727578809	0.00	746311486	2.57	2.57
a2_1	391189190	161	17475691	10854366.46	156845733	97419609.94	797.51
a2_2	1876768120	720671548	1017016406	41.12	1281842544	77.87	26.04
a2_3	2272487840	1190713414	1495453534	25.59	1599217929	34.31	6.94
a2_4	3223516130	1680368578	1947384848	15.89	2338388083	39.16	20.08
a2_5	787355300	307150825	570346714	85.69	786237264	155.98	37.85
b_1	7644173180	3291069369	3631639759	10.35	6210331706	88.70	71.01
b_2	5181493830	1015496187	1373418522	35.25	3867606948	280.86	181.60
b_3	6336834660	156691279	4517824174	2783.26	6046849469	3759.08	33.84
b_4	9209576380	4677808036	7116516915	52.13	8775627983	87.60	23.31
b_5	12426813010	922944697	10997549814	1091.57	12047005890	1205.28	9.54
b_6	12749861240	9525851483	11249369781	18.09	12600973936	32.28	12.01
b_8	14068207250	1214291143	12991350689	969.87	13506272779	1012.28	3.96
b_9	23234641520	15885437256	21201173223	33.46	23019099273	44.91	8.57
x_1	7422426760	3044418078	3313077913	8.82	6127182421	101.26	84.94
x_2	5103634830	1002379317	1387717268	38.44	3566783214	255.83	157.03
x_3	6119933380	69970	4281486805	6118932.16	5925991719	8469232.17	38.41
x_4	9207188610	4721591023	7149085032	51.41	8904414850	88.59	24.55
x_5	12369526590	54132	10940061224	20209870.49	12079582925	22314950.11	10.42
x_6	12753566360	9546936159	11176251882	17.07	12552660920	31.48	12.32
x_8	11611565600	29193	10472774740	35874166.91	11001680751	37685923.19	5.05
x_9	23146106380	16125562162	21308367308	32.14	22788557085	41.32	6.95

As Table 3 shows, under the 300-second time-out condition, MNLS manages to get the best-known solution for three instances (a1_1,a1_3,a1_5). Whereas MNSA only reaches the BKS once, for the easiest and smallest instance (a1_1), and it comes close for instance (a1_3) with $r\% = 0.11\%$. The percental relative deviation ($s\%$) in Table 3 shows that MNSA never performs better than MNLS under this time limit. Only twice the difference is around 0%, and for 16 instances MNSA deviates with more than 10% from the optimal value of MNLS. This is likely due to the varying speeds at which the two algorithms converge toward an optimal solution. As Wang et al. (2016) prove, MNLS is able to quickly improve the objective by allowing for infeasible moves, whereas running time is one of the main bottlenecks of simulated annealing Johnson et al. (1989).

Table 4: Computational results of MNLS and MNSA under the 3600-second time-out condition. In order the columns hold: the instances, the initial cost, the best-known solution (BKS), f_{best} and the relative deviation of f_{best} compared to the BKS ($r\%$) for both MNLS and MNSA, the relative deviation of f_{best} from MNSA compared to MNLS.

			MNLS		MNSA		
Inst.	Init. Cost	BKS	f_{best}	r (%)	f_{best}	r (%)	s (%)
a1_1	49528750	44306501	44306501	0.00	44307420	0.00	0.00
a1_2	1061649570	77535597	823639248	962.27	789624150	918.40	-4.13
a1_3	583662270	583005717	583662270	0.11	583662270	0.11	0.00
a1_4	632499600	244875206	283456068	15.76	531697704	117.13	87.58
a1_5	782189690	727578309	727578809	0.00	741642454	1.93	1.93
a2_1	391189190	161	11246769	6985470.81	49993472	31051745.96	344.51
a2_2	1876768120	720671548	941631477	30.66	1107342354	53.65	17.60
a2_3	2272487840	1190713414	1491729660	25.28	1568206223	31.70	5.13
a2_4	3223516130	1680368578	1890221061	12.49	1890122424	12.48	-0.01
a2_5	787355300	307150825	523176061	70.33	714292244	132.55	36.53
b_1	7644173180	3291069369	3558337887	8.12	5727273157	74.02	60.95
b_2	5181493830	1015496187	1143086505	12.56	3217086224	216.80	181.44
b_3	6336834660	156691279	1011874011	545.78	6043580421	3757.00	497.27
b_4	9209576380	4677808036	4677947894	0.00	8224770673	75.83	75.82
b_5	12426813010	922944697	4995944394	441.30	11874964569	1186.64	137.69
b_6	12749861240	9525851483	9526175217	0.00	12475568573	30.97	30.96
b_8	14068207250	1214291143	6939131896	471.46	13280341370	993.67	91.38
b_9	23234641520	15885437256	15983033582	0.61	21855860021	37.58	36.74
x_1	7422426760	3044418078	3275896226	7.60	5554500054	82.45	69.56
x_2	5103634830	1002379317	1128791787	12.61	3080428279	207.31	172.90
x_3	6119933380	69970	687060809	981836.27	5946752043	8498902.49	765.54
x_4	9207188610	4721591023	4724203042	0.06	8224506183	74.19	74.09
x_5	12369526590	54132	5658248639	10452587.21	12083416237	22322031.52	113.55
x_6	12753566360	9546936159	9546971105	0.00	12516063223	31.10	31.10
x_8	11611565600	29193	4986243933	17080172.44	10469695170	35863617.91	109.97
x_9	23146106380	16125562162	16288944372	1.01	21676250921	34.42	33.07

To check if this discrepancy is indeed caused by the short running time, we also consider Table 4 which holds the same variables under the 3600-second time-out condition. In this case, MNLS gets the best-known solution five times and gets within 1% of the BKS eight times. Similarly to the 300-second time-out condition, MNSA only gets to the best solution once and comes close one other time with an $r\%$ of 0.11% for the same instances as under 300 seconds. Looking at $s\%$ we find that for three instances MNSA and MNLS perform similarly, with a deviation close to 0%, and MNSA once outperforms MNLS for case $a1_2$ with a differential ratio of 4.13%. Despite MNSA working a little better with the larger running time, MNLS still outperforms MNSA for 22 out of 26 instances.

Notably, the original MNLS by Wang et al. (2016) find $r\%$'s smaller than 1% for 18 instances under 300s and for 23 instances under 3600s. This contrast is likely partly due to the efficiency of the different programming languages (Java in this paper vs C++ in Wang et al. (2016)). Furthermore, both Tables 3 and 4 show that for instances a2_1, x_3, x_5 & x_8 that have to make the largest relative improvements from the initial cost to the final best-known objective value, both MNLS and MNSA struggle with getting near the BKS. For all these instances the $r\%$ is more than a million. This might be due to the need for bigger moves with these instances, both MNLS and MNSA move at most 3 processes at the same time, whereas other well-performing methods for these instances, like the Variable Neigborhood Search of Gavranović et al. (2012), have neighborhoods that shift many more processes at the same time.

5.4 Robustness of MNLS and MNSA

Table 5: Computational results of MNLS and MNSA under the 300-second time-out condition. In order the columns hold: the instances, the best-known solution (BKS), f_{best} & f_{avg} & the coefficients of variance (c_v) for both MNLS and MNSA.

		MNLS			MNSA		
Inst.	BKS	f_{best}	f_{avg}	c_v (%)	f_{best}	f_{avg}	c_v (%)
a1_1	44306501	44306501	44306501	0.00	44307915	44573058	0.42
a1_2	777532177	810826359	820503855	0.73	887425469	896240512	1.55
a1_3	583005717	583006137	583417107	0.06	583662270	583662270	0.00
a1_4	244875206	291202548	301758989	6.72	525791949	548412126	3.06
a1_5	727578309	727578809	727578889	0.00	746311486	761862040	1.74
a2_1	161	17475691	40706008	106.34	156845733	190364491	11.90
a2_2	720671548	1017016406	1039711327	1.55	1281842544	1327017972	2.53
a2_3	1190713414	1495453534	1516538688	0.95	1599217929	1653350564	3.01
a2_4	1680368578	1947384848	1971589484	1.10	2338388083	2366766601	1.12
a2_5	307150825	570346714	580503948	1.57	786237264	786676060	0.06
b_1	3291069369	3631639759	3676285799	1.03	6210331706	6323579584	1.31
b_2	1015496187	1373418522	1433572228	2.53	3867606948	3986151516	2.29
b_3	156691279	4517824174	4550545277	0.60	6046849469	6119573739	1.57
b_4	4677808036	7116516915	7145995318	0.33	8775627983	8833808526	0.60
b_5	922944697	10997549814	11034106340	0.20	12047005890	12168038889	0.77
b_6	9525851483	11249369781	11304227986	0.31	12600973936	12632150894	0.27
b_8	1214291143	12991350689	13024886819	0.40	13506272779	13627680640	1.25
b_9	15885437256	21201173223	21255195379	0.30	23019099273	23068011873	0.24
x_1	3044418078	3313077913	3409288097	3.01	6127182421	6307488880	3.20
x_2	1002379317	1387717268	1410785798	1.65	3566783214	3677898115	3.64
x_3	69970	4281486805	4323142002	0.88	5925991719	5985100756	0.84
x_4	4721591023	7149085032	7167743576	0.26	8904414850	8975572107	0.77
x_5	54132	10940061224	11032984833	0.85	12079582925	12154003669	0.56
x_6	9546936159	11176251882	11266354572	0.71	12552660920	12672270435	0.64
x_8	29193	10472774740	10534868122	0.69	11001680751	11198562719	1.06
x_9	16125562162	21308367308	21492746469	0.72	22788557085	22888058620	0.33

Next, we evaluate the robustness of the results with the coefficients of variance of both algorithms. In Table 5 MNLS and MNSA have 17 and 12 instances with a c_v under the 1% respectively. Under the 3600-second time-out condition, MNLS has 12 instances with a coefficient of variance lower than 1%, and MNSA has 8 instances, see Table 6. In both tables, MNLS has more instances with a low c_v, which is likely due to the influence of randomness on both algorithms. Though MNLS does make random moves at various points in the search, MNSA only makes random moves. In addition, the coefficients of variance in this paper are based on only five runs, which implies that the differing seeds

might still have an erratic effect.

Table 6: Computational results of MNLS and MNSA under the 3600-second time-out condition. In order the columns hold: the instances, the best-known solution (BKS), f_{best} & f_{avg} & the coefficients of variance (c_v) for both MNLS and MNSA.

		MNLS			MNSA		
Inst.	BKS	f_{best}	f_{avg}	c_v (%)	f_{best}	f_{avg}	c_v (%)
a1_1	44306501	44306501	44306501	0.00	44307420	45538687	2.44
a1_2	77535597	823639248	826535476	0.36	789624150	809429745	2.56
a1_3	583005717	583662270	583662270	0.00	583662270	583662270	0.00
a1_4	244875206	283456068	287252655	1.06	531697704	560352615	5.70
a1_5	727578309	727578809	727578909	0.00	741642454	749649236	1.14
a2_1	161	11246769	14993509	15.47	49993472	65773678	23.02
a2_2	720671548	941631477	971720641	3.25	1107342354	1155048098	3.89
a2_3	1190713414	1491729660	1512249759	0.97	1568206223	1624683393	2.98
a2_4	1680368578	1890221061	1938837296	2.10	1890122424	1958636036	4.39
a2_5	307150825	523176061	570943462	5.05	714292244	762267453	4.59
b_1	3291069369	3558337887	3728682071	3.44	5727273157	5945967454	2.76
b_2	1015496187	1143086505	1181836211	2.34	3217086224	3291500486	1.54
b_3	156691279	1011874011	1036476340	1.88	6043580421	6172632437	1.21
b_4	4677808036	4677947894	4678749048	0.03	8224770673	8341533435	1.34
b_5	922944697	4995944394	5083634804	1.21	11874964569	12097233821	1.24
b_6	9525851483	9526175217	9533828564	0.18	12475568573	12609101743	0.66
b_8	1214291143	6939131896	7013420470	1.05	13280341370	13466902525	0.87
b_9	15885437256	15983033582	16121235001	0.48	21855860021	21989369355	0.58
x_1	3044418078	3275896226	3413492334	2.28	5554500054	5753340305	2.24
x_2	1002379317	1128791787	1165299598	2.37	3080428279	3118163490	1.51
x_3	69970	687060809	747857383	8.28	5946752043	5988827640	0.67
x_4	4721591023	4724203042	4728218068	0.05	8224506183	8395914238	1.18
x_5	54132	5658248639	5682261256	0.46	12083416237	12131531840	0.28
x_6	9546936159	9546971105	9594649369	0.49	12516063223	12672182447	0.75
x_8	29193	4986243933	5028257538	1.36	10469695170	10981811247	2.92
x_9	16125562162	16288944372	16340115194	0.21	21676250921	21801946410	0.56

6 Conclusion

In this research we study the Machine Reassignment Problem which is the topic of the ROADEF/EURO Challenge 2012 by Google. We propose an algorithm, using Simulated Annealing (MNSA) with a geometric cooling cycle and three neighborhoods. In addition, we replicate Multi-Neighborhood Local Search (MNLS) by Wang et al. (2016) and use this as a benchmark for our own algorithm. MNLS makes use of a combination of local and tabu search optimization, with the same three neighborhoods as MNSA. This paper answers the question whether MNSA outperforms MNLS for the Machine Reassignment Problem. After tuning the parameters for MNSA, we run both algorithms five times with two time-out conditions, 300 seconds and 3600 seconds. We find that our implementation of MNLS either finds the best-known solution or gets close to it for about 20% of the time, whereas MNSA only finds the best objective or comes close about 8% of the time. Next to MNLS outperforming MNSA for most instances under both time conditions, we also find that MNLS is more robust than MNSA. Thus this leads to the conclusion that, in general MNLS does better than MNSA in the context of the Machine Reassignment Problem.

7 Discussion

Lastly, we discuss the ways our algorithm could be improved and give some ideas for future research. When comparing the original MNLS by Wang et al. (2016) with the MNLS replication in this paper, we find that the original MNLS gets closer to best-known solution. As this paper already implements efficient ways of checking the feasibility, calculating the costs, and making a move, the better performance likely has to do with the different choices for coding language. Wang et al. (2016) use C++ which tends to be faster compared to java which this paper uses.

Next, MNLS outperforms MNSA in this paper. As MNSA just consists of random moves, which can lead to large differences depending on the random seed, it might be smart to run every instance more than five times to get a more accurate average objective value. With the same reasoning it might be smart to use more random seeds in the tuning of the parameters, and run every combination of parameters with every instance multiple times. Since choosing the right parameters is essential in MNSA, it can be valuable to check how much every neighborhood adds. The chosen parameters lead to only 10% chance of choosing a three-swap move, which is the lowest setting tested. This might mean that three-swap is not a good fit for MNSA, thus MNSA may improve with more parameter tuning for the chances of choosing a neighborhood ($\alpha\&\beta$ in this paper), and with doing a careful analysis of how much a neighborhood adds by comparing solutions when leaving this neighborhood in MNSA versus taking it out of MNSA.

In addition, for future research the neighborhoods used in MNSA can be made smaller which increases the chance of picking a good move. For instance, the technique of only looking at promising machine pairs introduced by Wang et al. (2016) may be useful. Lastly, as neither MNLS nor MNSA performs well for instances which need big moves, like swapping 10 processes at the same time, in future research it can be interesting to look at neighborhoods that allow larger moves, like the Big Process Rearrangment from Gavranović et al. (2012).

References

Butelle, F., Alfandari, L., Coti, C., Finta, L., Létocart, L., Plateau, G., ... Calvo, R. W. (2016). Fast machine reassignment. *Annals of Operations Research*, *242*, 133–160.

Canales, D., Rojas-Morales, N. & Riff, M.-C. (2020). A survey and a classification of recent approaches to solve the google machine reassignment problem. *IEEE Access*, *8*, 88815–88829.

Gabay, M. & Zaourar, S. (2016). Vector bin packing with heterogeneous bins: application to the machine reassignment problem. *Annals of Operations Research*, *242*(1), 161–194.

Gavranović, H., Buljubašić, M. & Demirović, E. (2012). Variable neighborhood search for google machine reassignment problem. *Electronic Notes in Discrete Mathematics*, *39*, 209–216.

Hoffmann, R., Riff, M.-C., Montero, E. & Rojas, N. (2015). Google challenge: A hyper-heuristic for the machine reassignment problem. In *2015 ieee congress on evolutionary computation (cec)* (pp. 846–853).

Johnson, D. S., Aragon, C. R., McGeoch, L. A. & Schevon, C. (1989). Optimization by simulated annealing: An experimental evaluation; part i, graph partitioning. *Operations research*, *37*(6), 865–892.

Lopes, R., Morais, V. W., Noronha, T. F. & Souza, V. A. (2015). Heuristics and matheuristics for a real-life machine reassignment problem. *International Transactions in Operational Research*, *22*(1), 77–95.

Masson, R., Vidal, T., Michallet, J., Penna, P. H. V., Petrucci, V., Subramanian, A. & Dubedout, H. (2013). An iterated local search heuristic for multi-capacity bin packing and machine reassignment problems. *Expert Systems with Applications*, *40*(13), 5266–5275.

Murat Afsar, H., Artigues, C., Bourreau, E. & Kedad-Sidhoum, S. (2016). *Machine reassignment problem: the roadef/euro challenge 2012* (Vol. 242). Springer.

Portal, G. M., Ritt, M., Borba, L. M. & Buriol, L. S. (2016). Simulated annealing for the machine reassignment problem. *Annals of Operations Research*, *242*(1), 93–114.

Turky, A., Sabar, N. R. & Song, A. (2017). An evolutionary simulating annealing algorithm for google machine reassignment problem. In *Intelligent and evolutionary systems: The 20th asia pacific symposium, ies 2016, canberra, australia, november 2016, proceedings* (pp. 431–442).

Turky, A., Sabar, N. R. & Song, A. (2018). Cooperative evolutionary heterogeneous simulated annealing algorithm for google machine reassignment problem. *Genetic Programming and Evolvable Machines*, *19*, 183–210.

Wang, Z., Lü, Z. & Ye, T. (2016). Multi-neighborhood local search optimization for machine reassignment problem. *Computers & Operations Research*, *68*, 16–29.

A Code

The Appendix shortly describes the code used for the algorithms. The algorithms are written in Java, using IntelliJ. Now follows a brief summary of the classes and their most important methods.

A.1 DataInitialization

This class reads the data files and turns all constants into the data structures used in the code. All instances are described in two seperate data files: one holding the current allocation, and one holding all constants described in Table 1.

A.1.1 readAssignment

This method reads the current allocation of processes and machines.

A.1.2 readModel

This method transfers all constants into data structures by reading them out of an excell file.

A.2 AdditionalDataStructures

This class does three things: it initializes all variables used in the code, it updates the variables (the setters) and it returns the variables (the getters). The variables kept track of are $U(m, r)$, $TU(m, r)$ introduced in Section 3.1, and it holds additional data structures proposed by Portal et al. (2016) which ensure that the running time is as low as possible.

A.2.1 Initializers

Update the variables from the entire allocation of machines and processes.

A.2.2 Setters

All setters get the index of the moved process, the machine this process is moving from and the machine this process is moving towards. It updates the corresponding variable by just calculating the effect of this one move, instead of recalculating the variable for the entire new allocation.

A.2.3 Getters

Return the variables.

A.3 AdditionalNStrategies

This class does the neighborhood partitioning techniques of MNLS described in Section 4.1.

A.3.1 processesConsidered

Returns ten random processes on each machine.

A.3.2 machinePairsConsidered

Returns all machine pairs that are considered using the technique of finding promising machine pairs from Section 4.1. This is done by finding all possible 2-combinations and calculating the lower bound.

A.4 ConstraintsPortal

All constraints are verified in this class. To ensure quick running time, every constraint is checked by just considering if feasibility changes with the one move made within the neighborhood. As there are three neighborhoods, which move either one process, two processes or three processes at the same time, every neighborhood has their own constraint methods. An example of a method is: conflict constraints for the one-shift neighborhood. Every method returns a boolean, true if the constraint is not violated and false otherwise.

A.5 FasterCosts

All costs are calculated in this class. Like ConstraintsPortal, to ensure quick running time every neighborhood has its own methods for calculating costs. There is two kinds of methods within this class.

A.5.1 initializeCost

These methods compute the cost for the entire current allocation. Every cost explained in Section 3.3 has its own method.

A.5.2 differenceCostNeighborhood

Here the difference in cost a move causes is calculated. An example of such a class is differenceBalanceCostTwoSwap, which computes the difference in balance cost for making a given move from the two-swap neighborhood.

A.6 FirstRepairStrategy

This class tries to repair an infeasible move for MNLS, see Wang et al. (2016) for more details.

A.7 SecondRepairStrategy

This class tries to repair an infeasible move for MNLS, see Wang et al. (2016) for more details.

A.8 Neighborhood

This class holds three different methods, one for each neighborhood. Within each method a move from the neighborhood is calculated. These methods either return the best feasible move, the best infeasible move or a random move.

A.9 LocalSearch

This class implements the local search procedure of MNLS.

A.9.1 localSearchAlg

Performs a general local search of a given neighborhood. This method implements algorithm 5 from Wang et al. (2016). At every iteration method moveSelection within this same class is called. When the time from the time-out condition runs out, the best objective value is returned.

A.9.2 moveSelection

This method implements algorithm 6 from Wang et al. (2016). In this algorithm the class Neighborhood (see Section A.8) is called to return either a feasible move, an infeasible move or a random move from a given neighborhood. If a good infeasible move is found it calls either FirstRepairStrategy (Section A.6), or SecondRepairStrategy (Section A.7) to fix this move.

A.10 Extension

This class implements the simulated annealing algorithm from this paper.

A.10.1 simulatedAnnealing

Performs the algorithm described in Section 4.3. It does this by calling one of three methods within this same class: oneShift, twoSwap or threeSwap which all return a

random move from their neighborhood. When the time from the time-out condition runs out, the best objective value is returned.

B Constants in the MRP

Table 7: Description of all constants used in the MRP

Constants	Description
M	The set of machines, $M = m_1, m_2, .., m_k$ with $\|M\| = k$
P	The set of processes, $P = p_1, p_2, .., p_q$ with $\|P\| = q$
R	The set of resources, $R = r_1, r_2, .., r_d$ with $\|R\| = d$
TR	The set of resources which need transient usage, $TR = tr_1, tr_2, .., tr_h$ with $\|TR\| = h$ and $TR \subseteq R$
S	The set of services, $S = s_1, s_2, .., s_f$ with $\|S\| = f$. Processes are divided over services, services are disjoint
L	The set of locations, $L = l_1, l_2, ..., l_g$ with $\|L\| = g$. Machines are partitioned into locations, locations are disjoint
N	The set of neighborhoods, $N = n_1, n_2, .., n_v$ with $\|N\| = v$. Machines are partitioned into neighborhoods, all neighborhoods are disjoint
SD	The set of service dependencies, $SD = sd_1, sd_2, ..., sd_z$ with $\|SD\| = z$
B	The set of triples, $B = b_1, b_2, ..., b_e$ with $\|B\| = e$. A triple consists of three values: two resources and the target ratio of these two resources
$NE(m)$	The neighborhood of machine $m \epsilon M$
$C(m, r)$	The capacity of resource $r \epsilon R$ on machine $m \epsilon M$
$SC(m, r)$	The safety capacity of resource $r \epsilon R$ on machine $m \epsilon M$
$R(p, r)$	The requirement of resource $r \epsilon R$ for process $p \epsilon P$
$S_min(s)$	The minimum number of different locations where at least one process of service $s \epsilon S$ should run
$PMC(p)$	The cost of moving process $p \epsilon P$ from its original machine
$MMC(m, m')$	The cost of moving any process $p \epsilon P$ from machine $m \epsilon M$ to machine $m' \epsilon M$. $\forall m \epsilon M, MMC(m, m) = 0$
$wt_1(r)$	The weight of load cost for resource $r \epsilon R$
$wt_2(r)$	The weight of balance cost for triple $b \epsilon B$
wt_3, wt_4, wt_5	The weights of process move cost, service move cost and machine move cost in this order