POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Informatyka (INF)
SPECJALNOŚĆ: Advanced Informatics and Control (AIC)

PRACA DYPLOMOWA MAGISTERSKA

Jednoprocesorowy problem szeregowania zadań przy kryterium minimalizacji ważonej sumy opóźnień zadań – algorytmy i badania eksperymentalne

The single-machine total weighted tardiness problem – algorithms and experimental investigations

AUTOR:
Krystian Suliński

PROWADZĄCY PRACĘ:
dr inż. Leszek Koszałka

OCENA PRACY:

WROCŁAW 2016

WROCŁAW UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF ELECTRONICS

FIELD OF STUDY: Computer Science (INF)
SPECIALIZATION: Advanced Informatics and Control (AIC)

MASTER THESIS

Jednoprocesorowy problem szeregowania zadań przy kryterium minimalizacji ważonej sumy opóźnień zadań – algorytmy i badania eksperymentalne

The single-machine total weighted tardiness problem – algorithms and experimental investigations

AUTHOR:
Krystian Suliński

SUPERVISOR:
dr inż. Leszek Koszałka

GRADE:

WROCŁAW 2016

Table of Contents

Chapter 1. Introduction

The Single Machine Total Weighted Tardiness Problem (SMTWTP) is a well-known optimization problem in the field of engineering, operation research and computer science. It is computationally challenging strongly NP-hard problem that can be solved effectively by using heuristic methods.

The SMTWTP is a universal, critical tool and its solutions have been implemented widely in a plethora of domains, including computer multitasking, manufacturing enterprises, robotics engineering or batch manufacturing processes in chemistry. It impacts the productivity of processes and optimizes production as well as reduces costs, aiming to maximize the efficiency through outperforming older manual methods of job sequencing.

1.1 Aim and Scope of the Work

This thesis introduces implementation of metaheuristic algorithms along with experimental investigations in an attempt to enhance current solutions for the SMTWTP. Even though computers are becoming more and more sophisticated, the best known solutions have not been improved since 2002 [4]. However, despite the problem's complexity, there have been raised voices that the SMTWTP is already resolved for metaheuristics [9], although these voices must be proved [12]. Nevertheless, as the problem is evergreen and well-known in the broad field, therefore there is a continuous demand for a constant improvement.

There have been implemented four algorithms to deal with it – Exhaustive Search (ES), Simulated Annealing (SA), Tabu Search (TS) and Iterated Dynasearch (ID) – and their results have been compared. Design of the experimental system is the most essential part of the project as the quality of the results ought to be considered against their performance time and accuracy. The datasets used in the referenced literature are the same here so that the outcomes have been tested unbiasedly. An extensive descriptions of the algorithms have been made as well as the explanation of the implementation itself.

1.2 Problem definition

The SMTWTP can be defined as follows − each of n jobs is to be computed without an interruption on one machine that can process no more than one job at a time. Each of *n* jobs becomes available for processing at time zero, has its own processing time *p,* weight *w* and due date *d.* Job *j* is delayed, when finishes itself after its due date d_j. The tardiness, which is the maximum from zero and the completion time of job *j,* has an economic interpretation, which symbolizes the costs of production. The tardiness must be minimized as far as it is only possible. The problem is to find combination of jobs, for which the weighted tardiness is minimized.

1.2.1 Mathematical model

Each of instances consist of a number of jobs. The characteristic of one instance is presented below.

Given:

- job $j = 1, 2, \ldots, n$
- processing time $p_j > 0$
- weight $w_j > 0$
- due date $d_j > 0$

To find:

- tardiness, $T(j) = max\{C_j - d_j, 0\}$
- where C_j is the earliest completion time

Such that:

- weighted tardiness $\sum_{j=0}^{n} w(j) * T(j)$ is minimized

1.2.2 Computational complexity

As the SMTWTP is NP-complete hard (non-deterministic polynomial-time hard) [13, 14], so that the complexity solving this problem using all feasible solution if factorial. Therefore, there is no existing tool to work this out in reasonable time [5]. For instance, if we had a computer that could compute one million operations per second, solving one instance of 50 jobs would

take 10^{40} *13.8 billion years. The implementation of ES confirms that it does not compute even 9 job's instance in acceptable time.

1.3 Benchmark instances

The benchmark instanced used to test the solutions are taken from OR-Library [13]. They are commonly used to test optimization approaches for this problem across literature. Their author provides 375 data sets parameterized in different ways. There are 3 big sets, 125 instances in one. Each instance has different number of jobs – 40, 50 and 125. Processing time is randomly taken from a uniform distribution as well as the weights – [1, 100] and [1, 10], respectively. Due dates are also drawn randomly as numeral values from:

$$[P\left(1 - TF - \frac{RDD}{2}\right), P\left(1 - TF + \frac{RDD}{2}\right)]$$

where:

$$P = \sum_{j=1}^{n} p(j)$$

Relative Range of Due Dates (RDD) and average Tardiness Factor (TF) are:

$$\text{RDD } \epsilon \{0.2, 0.4, 0.6, 0.8, 1.0\}$$
$$\text{TF } \epsilon \{0.2, 0.4, 0.6, 0.8, 1.0\}$$

For each out of the 25 pairs from Relative Range of Due Dates and Tardiness Factory, five instanced were computed, producing 125 instances for each value of job size (40, 50 and 125). Optimal solutions are available for job number 124 of the smallest instance (40 jobs) and for 125 of the medium instance (50 jobs). Unresolved remains job number 19 for 40-job instance and 11, 12, 14, 19, 36, 44, 66, 87, 88, 11 for the 50-job instance [13]. The rest of the instances from these two data sets have been solved and their optimal solutions are known. However, for the job size n = 100, only best-known solutions are known and without their final proof.

Chapter 2. Related Work

This annotated bibliography, so called related work, provides an essential insight into this problem by looking from different perspectives, different author's view and different solutions on the SMTWTP. First, I run through Iterated Dynasearch, which is implemented further, then describe Local Search Heuristics and finally, present new discovery on this field – The Breakout Dynasearch.

2.1 Iterated Dynasearch

This new neighbourhood search technique (ID) was introduced in [4], which is the state-of-the-art solution for the SMTWTP. This algorithm for this domain performs better results than the state-of-the-art TS known for [6], in terms of not only computational time, but more importantly, of the solution quality. [5] depicts convincing evidence that this algorithm is a powerful new local search technique for the SMTWTP and also for other optimization problems.

The authors describe briefly the first-improve and best-improve descent algorithms against the ID, present this algorithm along with the speedups for the Total Weighted Tardiness Comparisons and the dynamic program in dynasearch.

The implementation and computational experience has been described fully and in details so that the experimental design was adopted in this thesis. The computational results for the iterated local search algorithms and multi-start TS are proceeded on the datasets provided by the OR-Library [13]. which are the same datasets for experimental investigations done in this paper. The ID features dynamic programming in the local search and the exponential size of a neighbourhood is explored in polynomial time. This is accomplished by finding the best permutation of moves where each of the neighbours refer to one or more moves.

They alternate the possibility of choosing the neighbourhood from independent insert moves to independent swap moves to state at the end, that these both methods can be used together, which finally distinctly proves this approach's flexibility.

2.2 Local Search Heuristics

[6] describes several local search heuristics for the problem. They demonstrate new binary encoding scheme to represent solutions, which is compared with the known permutation for TS, SA, evolutionary algorithms and Threshold Accepting. They conclude, that with the binary-coded features the algorithm is more likely to produce the optimal solution, however they are not as robust as the classical (natural permutation).

This article is an extension of [15], supplementing their descent and SA by giving a complete comparison of multi-start version not only for these, but also or the mentioned Threshold Accepting, evolutionary algorithm and the TS.

2.3 Breakout Dynasearch

Breakout Dynasearch (BDS) algorithm is the newest and ground-breaking game changer for solving the SMTWTP, inspired by [8] and presented in [7], which has been available since the 7th of May 2016. This algorithm solves all the standard benchmark instances from [13] in 0,1s. On top of that, for a half thousand larger instances with 150, 200, 250 and 300 jobs it produces all the upper bounds on average in 250s, whereas the objective function remains not changed. After 18 years, the unresolved solutions have been not only solved and obtained in very impressive time – one tenth of a second – but also with a hit ratio 100%. This algorithm is based upon the merits of breakout local search and ID. Its core is presented in 'Algorithm 1'.

```
1    Input: Processing time, weight and due time of each job
2    Output: The best scheduled sequence S* found so far
3    S* ← GenerateInitialSolution()
4    L ← L0
5    while stopping condition not reached do
6        S' ← Dynasearch(S*)
7        L ← DetermineJumpMagnitude(L, S', history)
8        T ← DeterminePerturbationType(S', history)
9        if f(S') < f(S*) then
10           S* = S'
11       end if
12       S* ← Perturbation(L, T, S*, history)
13   end while
```

Algorithm 1. Pseudo code of BDS algorithm for SMTWTP.

As we can notice in 'Algorithm 1', the breakout dynasearch determines jump magnitude and perturbation type based on the solutions from dynasearch and taking into account its history as well.

```
1    Input: An initial sequence S0
2    Output: The local optimal sequence S0 found so far
3    S0 ←S0
4    M ← two_swap(S0)
5    while dynamic_programming(M) < 0 do
6        S0 ← backtracking(S0)
7        M ←two_swap(S0)
8    end while
9    return S0
```

Algorithm 2. Pseudo code of the dynasearch.

The dynasearch in 'Algorithm 2' allows for two swaps at a time and also for a backtracking – this concept is developed widener in Chapter 3.

```
1   Input: Local optimum S returned by DS, current jump magnitude L and history information
    including the hash table HT of previously encountered local optimum, the descent-phase number lc
    when the last cycle was encountered, the current descent-phase number iter_cur, and the number w of
    consecutive iterations of non-improving global optima.
2   Output: Jump magnitude L for the next perturbation phase
3   wc ← 10, num_nc ←1
4   prev_visit ← PreViousEncounter(HT, f(S))      //Check if f(S) was encountered before
5   if f(S) < f(S*) || w > T0 then
6       w ← 0
7   else
8       w ← w + 1
9   end if
10  if prev_visit != -1 then
11      wv ← wc + iter_cur – lc          //A cycle is encountered
12      num_nc ← num_nc + 1
13      lc ← iter_cur
14      L ← L – 1
15  else if (iter_cur – lc) > (wc/num_nc) * β then
16      L ← L – 1
17  end if
18  if L > Lmax then
19      L ← Lmax
20  else if L ← Lmin then
21      L ← Lmin
22  end if
```

Algorithm 3. Pseudo code of the DetermineJumpMagnitude.

BDS is a general stochastic local search solution which reflects the main scheme of iterated search [9]. It makes use from the dynasearch to intensify the neighbourhood search in specific area by applying perturbations when a local minimum has been found, to move easily to another area. The pseudo code presented in [7] is shown in ‘Algorithm 1’, which is complemented in ‘Algorithm 2’ and ‘Algorithm 3’ right abo

Chapter 3. Algorithms

Metaheuristic algorithms are not designed to provide the optimal solution. Their main purpose is to deliver solution that is just good enough. By manipulating algorithms' parameters, it is possible to adjust the quality of their final computation. This chapter provides description of four algorithms that are implemented in Chapter 4. They have been chosen due to their usefulness shown in literature for this particular problem.

3.1 Exhaustive Search

Exhaustive Search produces always exact solution – if it exists. It costs proportionally to the number of candidates, which grows very fast as the problem inclines. This brute force method is usually applied when a problem's size is small or when an exact solution is needed.

There has been selected Heap's algorithm to produce all possible permutations of S elements. This algorithm generates permutations based always on the previous one by swapping two elements, while the other S-2 remains untouched. There are two popular implementations of this algorithm – recursive and non-recursive format. The second one have been chosen because of its easier implementation compared to the non-recursive one and better clarity plus readiness of the code. Its pseudo code is presented in 'Algorithm 4'.

```
procedure generate(int N, array A):
    if n == 1 then output(A)
    else for (i = 0; i <= n; ++i) do
            generate(n - 1, A)
            if n is even then swap(A[i], A[n-1])
            else swap(A[0], A[n-1])
         generate(n - 1, A)
```

Algorithm 4. Pseudo code of the Heap's algorithm [11]

3.2 Simulated Annealing

The SA algorithm is derived from its use in metallurgy – it is based upon natural annealing processes. The first step is to heat up the material up to some point, so that the elements are redistributed equally, then lower the temperature little by little. In the application of this algorithm, we set the starting and the end temperature – the high and the low one, respectively. This approach to manipulate with the temperature is named the *cooling method*.

The temperature reflect the chances of setting a worse than the current solution. This is to assure that we avoid trapping in *local minimum*, which may prevent from obtaining improved results.

3.2.1 Parameters

SA is a complex algorithm in terms of calibrating its parameters, which are as follows:

- *cooling scheme* – it describes the number of big iterations of the algorithm and the speed of temperature drop
- *starting and ending temperature* – determining the possibility of choosing a worse solution
- *small iterations* – allow to search for random solution with no impact on the temperature changes

3.3 Tabu Search

TS is based on searching neighbourhood, created by all feasible solution by a move sequence [2]. There are exist forbidden moves, i.e. tabu moves in this sequence. The algorithm avoids oscillations around local optimum thanks to storing information about already checked solutions in the TS list. This list contains the freshest moves to avoid cycling while looking for new solutions.

This algorithm has been used for various scheduling problems, nonetheless it cannot search space of the solution using concurrency [3]. TS is characterized by a set of parameters that can be calibrated and leading to various versions and strategies of this algorithm, like maximum elite count and window, stopping condition and the tenure multiplier.

3.4 Iterated Dynasearch

This algorithm has been first described in [4], which is a neighbourhood search technique that allows to perform a sequence of moves at one go. Therefore, it avoids the main handicap of descent techniques: looking only one single move ahead – but from the other hand this is much harder to implement. Overview of this algorithm is presented in 'Figure 1' on the next page.

First off, it finds and updates a local optimum in the generated initial solution. It allows backtracking in order to assure that most of the search is made in valid areas of the solution space, then after backtracking, it either applies a kick or adopts new local optimum.

The 'Table 2' and 'Table 3' demonstrate that the Dynasearch Swap outperforms Best-Improve Descent Swaps due to its ability to make many multiple swaps at one go using methods of the dynamic programming.

Table 1. Data for the Problem Instance as in [4].

Job j	1	2	3	4	5	6
Processing time	3	1	1	5	1	5
Weight	3	5	1	1	4	4
Due date	1	5	3	1	3	1

Table 2. Best-Improve Descent Swaps as in [4].

Iteration	**Current Sequence**	**Total Weighted Tardiness**
	1 2 3 **4 5** 6	109
1	1 2 3 5 **4 6**	90
2	**1** 2 3 **5** 6 4	75
3	5 2 3 1 6 4	70

Table 3. Dynasearch Swap as in [4].

Iteration	**Current Sequence**	**Total Weighted Tardiness**
	1 **2 3** 4 5 6	109
1	1 **3** 2 **5** 4 6	89
2	**1 5** 2 3 6 4	68
3	5 1 2 3 6 4	67

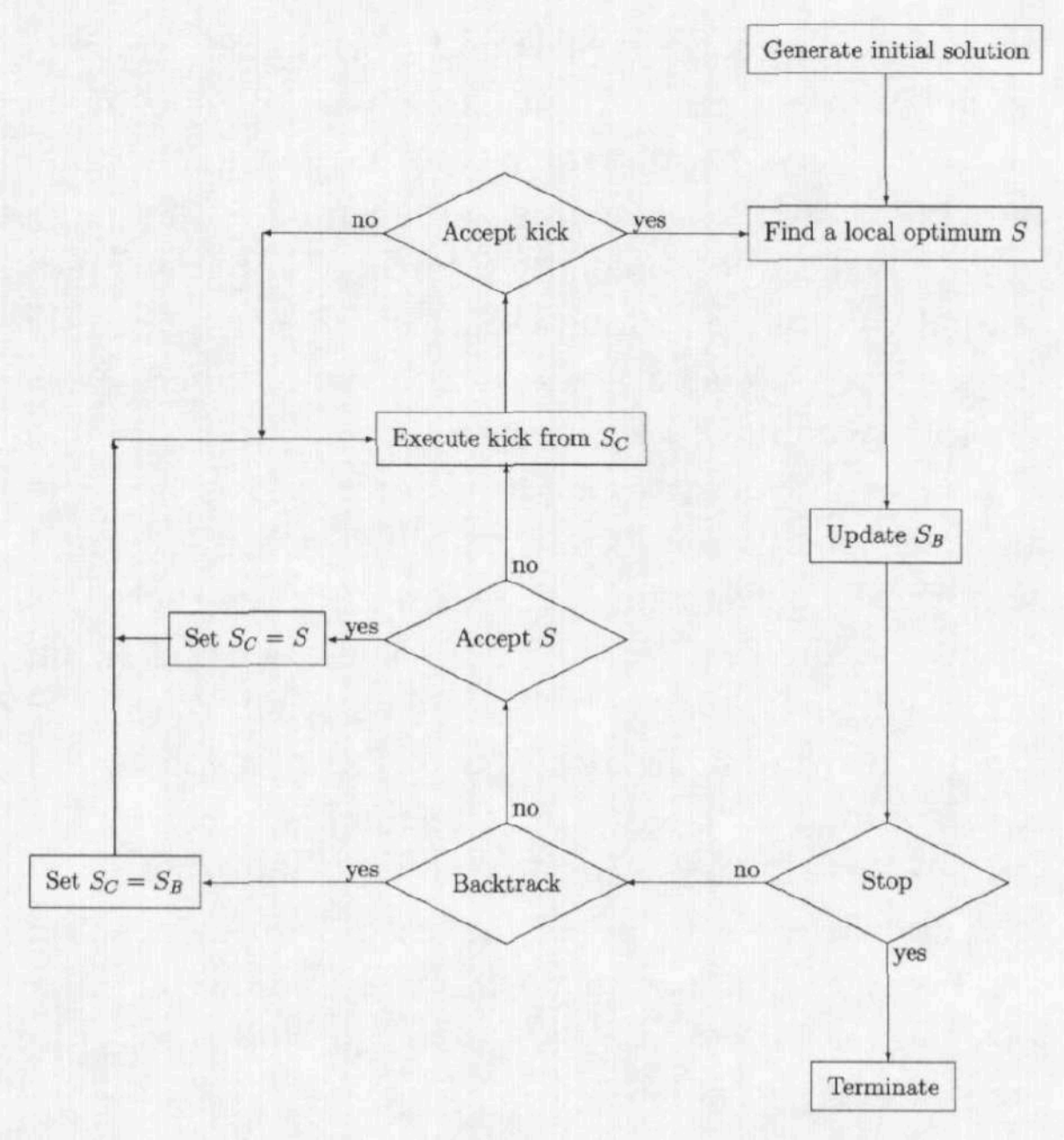

Figure 1. Overview of ID.

Chapter 4. Experimentation System

Results have been averaged by repeating computation one hundred times. Only the most significant have been provided and analysed. Classic algorithm, the SA has performed really well compared to the state-of-the-art ID, while TS was not good enough to obtain competitive results.

4.1 Implementation of Exhaustive Search

The implementation of ES is comprised in one class (as shown in 'Algorithm 5'). This algorithm does not have any parameter to calibrate, so the only fields it has are *size*, *solution* and *elapsedTime* to measure its effectiveness. Its main method – *run*, implements the Heap's algorithm in its recursive form. It takes *dataSet* as the input and produces permuted *solution* as the output.

```
public class ExhaustiveSearch {
    private Map<ArrayList<ArrayList<Integer>>, ArrayList<Integer>> solution;
    private long size;
    private long elapsedTime;

    public ExhaustiveSearch(final ArrayList<Order> dataSet) {
        solution = new HashMap<>(dataSet.size());
        run(dataSet);
    }

    private void run(final ArrayList<Order> dataSet) {
        ArrayList<ArrayList<Integer>> indices;
        ArrayList<Integer> totalWeightedTardinesses;

        final int numberOfJobs = dataSet.get(0).jobsSize();
        Permutations permutations = new Permutations(numberOfJobs);

        int i = 0; for (Order jobs : dataSet) {
            indices = new ArrayList<>();
            totalWeightedTardinesses = new ArrayList<>();
            for (ArrayList<Integer> jobOrder : permutations.getPermutations()) {
                Order permutedOrder = new Order(jobs, jobOrder);

                indices.add(jobOrder);
                totalWeightedTardinesses.add(permutedOrder.getTardiness());
            }
            solution.put(indices, totalWeightedTardinesses);
        }
        size = solution.size();
        elapsedTime = permutations.getElapsedTime();
    }
```

Algorithm 5. ES class.

Of course, this class uses *getPermutations()* function – a crucial for its behaviour, that is demonstrated as the function *permuteFromZeroToN* of *Permutation* class in ‘Algorithm 6’.

```
class Permutations {
    private ArrayList<ArrayList<Integer>> permutations;
    private static long elapsedTime;
    private static boolean addedFirstPermutation = false;

    Permutations(final int n) {
        permuteFromZeroToN(n);
    }
    private void permuteFromZeroToN(final int n) {
        permutations = new ArrayList<>();
        long startTime = System.currentTimeMillis();
        permuteFromZeroToN(n, createIntegersFromOneToN(n));

        elapsedTime = System.currentTimeMillis() - startTime;
    }
    private ArrayList<Integer> createIntegersFromOneToN(final int lastValue) {
        ArrayList<Integer> list = new ArrayList<>();
        for (int i = 0; i < lastValue; ++i) {
            list.add(i);
        } return list;
    }
    private void permuteFromZeroToN(final int n, ArrayList<Integer> permutable) {
        if (!addedFirstPermutation) {
            permutations.add(getDeepCopy(permutable));
            addedFirstPermutation = true;
        }
        if (n != 1) {
            for (int i = 0; i < n - 1; ++i) {
                permuteFromZeroToN(n - 1, permutable);
                int temp;
                if (n % 2 == 0) {
                    temp = permutable.get(i);
                    permutable.set(i, permutable.get(n - 1));
                } else {
                    temp = permutable.get(0);
                    permutable.set(0, permutable.get(n - 1));
                }
                permutable.set(n - 1, temp);
                permutations.add(getDeepCopy(permutable));
            }
            permuteFromZeroToN(n - 1, permutable);
        }
    }
```

Algorithm 6. Permutations.

4.2 Implementation of Simulated Annealing

The implementation of the SA is given below in ‘Algorithm 7’. The constructor takes jobs, the temperature and cooling rate and runs the algorithm returning the sum of the best tardiness.

In the function *run*, we first set the temperature and cooling rate, generate random solution and assigns it to the best solution. Then, if the set temperature is greater than 1, we generate new order and swap two jobs by random. After computation of the new tardiness, we run acceptance probability function on these two solutions. If the new solution is better, the previous one is being replaced. We update the temperature by multiplying it by *1 – cooling rate* and star over again.

```
public class SimulatedAnnealing {
    private final double temperature;
    private final double coolingRate;
    private Map<ArrayList<ArrayList<Integer>>, ArrayList<Integer>> solution = new
HashMap<>();

    public SimulatedAnnealing(final ArrayList<Order> jobs, final double
temperature, final double coolingRate) {
        this.temperature = temperature;
        this.coolingRate = coolingRate;
        run(jobs);
    }

    private void run(final ArrayList<Order> jobs) {
        ArrayList<ArrayList<Integer>> theBestOrders = new ArrayList<>();
        ArrayList<Integer> theBestTardinesses = new ArrayList<>();

        long sumOfBestTardinesses = 0;
        int i = 0;
        for (Order j : jobs) {
            double temperature = this.temperature;
            double coolingRate = this.coolingRate;

            Order currentOrder = new Order(j.getOrder());
            currentOrder.generateRandomSolution();
            Order theBestOrder = new Order(currentOrder.getOrder());
```

```
            while (temperature > 1) {
                Order newOrder = new Order(currentOrder.getOrder());
                newOrder.swapTwoJobsRandomly();

                int currentTardiness = currentOrder.getTardiness();
                int neighbourTardiness = newOrder.getTardiness();

                double random = SimulatedAnnealingUtility.randomDouble();
                if
(SimulatedAnnealingUtility.acceptanceProbability(currentTardiness,
neighbourTardiness, temperature) > random) {
                    currentOrder = new Order(newOrder.getOrder());
                }
                if (currentOrder.getTardiness() < theBestOrder.getTardiness()) {
                    theBestOrder = new Order(currentOrder.getOrder());
                }
                temperature *= 1 - coolingRate;
            }
            theBestTardinesses.add(theBestOrder.getTardiness());
            theBestOrders.add(theBestOrder.getOrderInIndexes());
            int solution = theBestOrder.getTardiness();
            sumOfBestTardinesses += theBestOrder.getTardiness();
        }
        this.solution.put(theBestOrders, theBestTardinesses);
    }
}
```

Algorithm 7. Implementation of the SA algorithm.

The *acceptance probability* function is shown below in ‘Algorithm 8’.

```
public class SimulatedAnnealingUtility {
    public static double acceptanceProbability(final int currentTardiness, final
int newTardiness, final double temperature) {
        if (newTardiness < currentTardiness) {
            return 1.0;
        }
        return Math.exp(currentTardiness - newTardiness)/temperature;
    }
}
```

Algorithm 8. Implementation of the acceptance probability function.

4.3 Implementation of Tabu Search

TS was implemented as a single class that has two fields – number of iterations and size of the tabu list, which is presented in ‘Algorithm 9’. The main function *run* takes the inputs and returns computed solution.

The TS runs as follows: it creates an initial random solution and solves it, if the stop condition is fulfilled (as in ‘Algorithm 10’), it proceeds to the main part of it. It creates a new solution neighbourhood, and iterates through solution candidates. If the solution candidate is not already on the tabu list, it adds it to the candidate list. Then it gets the best out of them and adds to the tabu list. If the tabu list is full, it removes the first element to free a place for another one.

```
public class TabuSearch {
    private int numberOfIterations;
    private int maxTabuSize;

    private Map<ArrayList<ArrayList<Integer>>, ArrayList<Integer>> solution =
new HashMap<>();

    public TabuSearch(final ArrayList<Order> orders, final int
numberOfIterations, final int maxTabuSize) {
        this.numberOfIterations = numberOfIterations;
        this.maxTabuSize = maxTabuSize;
        run(orders);
    }

    private void run(final ArrayList<Order> orders) {
        ArrayList<ArrayList<Integer>> bestOrders = new ArrayList<>();
        ArrayList<Integer> bestTardinesses = new ArrayList<>();
        BigInteger sumOfBestTardinesses = new BigInteger("0");
        int solutionCount = 0;
        for (Order order : orders) {
            Order initialSolution = new Order(order);
            initialSolution.generateRandomSolution();
            initialSolution.solve();
            Order bestSolution = initialSolution;
            ArrayList<Order> tabuList = new ArrayList<>(maxTabuSize);

            while (!stopCondition()) {
                ArrayList<Order> candidateList = new ArrayList<>();
                ArrayList<Order> solutionNeighbourhood =
```

```
      generateNeighbourhood(bestSolution);

                      for (Order solutionCandidate : solutionNeighbourhood) {
                          if (!tabuList.contains(solutionCandidate)) {
                              candidateList.add(solutionCandidate);
                          }
                      }
                      Order solutionCandidate = locateBestCandidate(candidateList);

                      if (solutionCandidate.getTardiness() <
      bestSolution.getTardiness()) {
                          bestSolution = solutionCandidate;
                      }
                      tabuList.add(bestSolution);
                      if (tabuList.size() > maxTabuSize) {
                          tabuList.remove(0);
                      }
                  }
                  bestOrders.add(bestSolution.getOrderInIndexes());
                  bestTardinesses.add(bestSolution.getTardiness());
                  sumOfBestTardinesses = sumOfBestTardinesses.add(new
      BigInteger(Integer.toString(bestSolution.getTardiness())));

                  //System.out.println("TS solution of Instance nr " +
      solutionCount++ + ": " + initialSolution.getTardiness());
              }
              System.out.printf("Sum of TabuSearch all solutions: %,d\n",
      sumOfBestTardinesses);
              this.solution.put(bestOrders, bestTardinesses);
          }
      }
```

Algorithm 9. TS implementation.

```
private Order locateBestCandidate(ArrayList<Order> candidateList) {
    int max = 0;
    for (int index = 1; index < candidateList.size(); ++index) {
        if (candidateList.get(index).getTardiness() <
candidateList.get(max).getTardiness()) {
            max = index;
        }
    }
    return candidateList.get(max);
}
```

```
private ArrayList<Order> generateNeighbourhood(final Order bestSolution) {
    ArrayList<Order> neighbourhood = new ArrayList<>();
    Order copyOfBestSolution = new Order(bestSolution);

    int i = 10;
    while (i > 0) {
        copyOfBestSolution.swapTwoJobsRandomly();
        neighbourhood.add(copyOfBestSolution);
        copyOfBestSolution = new Order(copyOfBestSolution);
        i--;
    }
    return neighbourhood;
}

private boolean stopCondition() {
    return this.numberOfIterations-- <= 0 ? true : false;
}
```

Algorithm 10. TS componenets - locate candidate and generate beighbourhood.

4.4 Implementation of Iterated Dynasearch

The implementation of ID is overviewed in ‘Figure 1’ in Chapter 2. Exploration of multiple local minimas was implemented as a standard local search algorithm that start off with many different beginning solutions. As in SA and TS, they are generated randomly and do not have anything in common with the previous runs. However, the author chosen both riskier and more promising solution to generate the starting solutions by restarting near local minimum. Thanks to that, the next solution is taken from the current local optimum (which is the best local optimum discovered up to now or lastly generated). This is done by using a specified in advance mode of a random kick.

So an iterated local search algorithm runs a local search within the local optimas. Subsequently, there is either the traditional descent or dynasearch used to generate the solution N that is the new optimum. If needed, the new solution is updated.

4.5 Graphic User Interface

The entire application has been implemented in the Java 8 language and in the IntellJ IDEA IDE (Integrated Development Environment). It has more than two thousand line of code and 20 classes. The main class is the *Facade*, from which the application can be run – this is also the name of the used design-pattern. The class *Factory* is the Factory design pattern – it encapsulates all the implementation which is hidden in various packages like algorithms, GUI, problem and utilities.

This software is implemented according to high-level standards in the industry. The GUI has been made in FXML technology, which corresponds with a controller that handles events. The main window possesses four tabs – ES, SA, TS and ID and each of them are distinguished due to their different characteristics.

We can see in ‘Figure 3’, that the parameters to fill are placed on the left-hand side, while an appropriate graph is generated dynamically on the right-hand side. Incidentally, on this figure there are 7! instances and all their possible solutions. (7! equals 5040 and the axis X ends at that point).

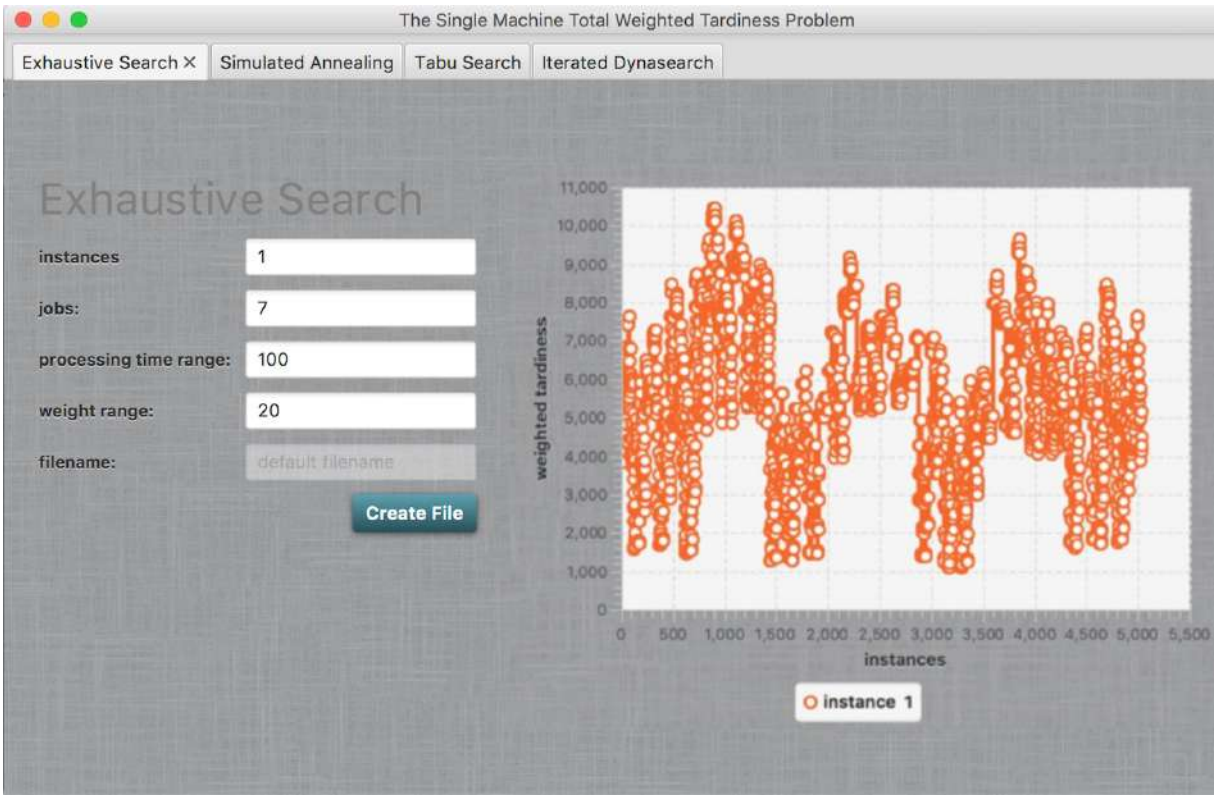

Figure 2. Graphic User Interface of the experimentation system.

Chapter 5. Investigations

5.1 Environment

All experimental investigations have been conducted within the same environment, on the same processor, memory size and on the same data sets. The machine that has been used has following attributes:

- **Processor:** 2.7 GHz Intel Core i5
- **Memory:** 8 GB 1867 MHz DDR3
- **Computer**: MacBook Pro
- **Operation System**: OS X El Capitan

5.2 Exhaustive Search

ES has been made as the proof of concept for the first implementation and also the confirmation, that a bit more sophisticated solutions are needed. Basically, the computer run out of a memory when tried to compute just only 11 jobs – 11!, which is approximately 4 million combinations. This situation is pictured in both ‘Table 4’ and ‘Figure 4’.

Table 4. ES.

Time [ms]	Number of jobs
86	5
156	6
302	7
537	8
1998	9
20990	10
java.lang.OutOfMemoryError	11

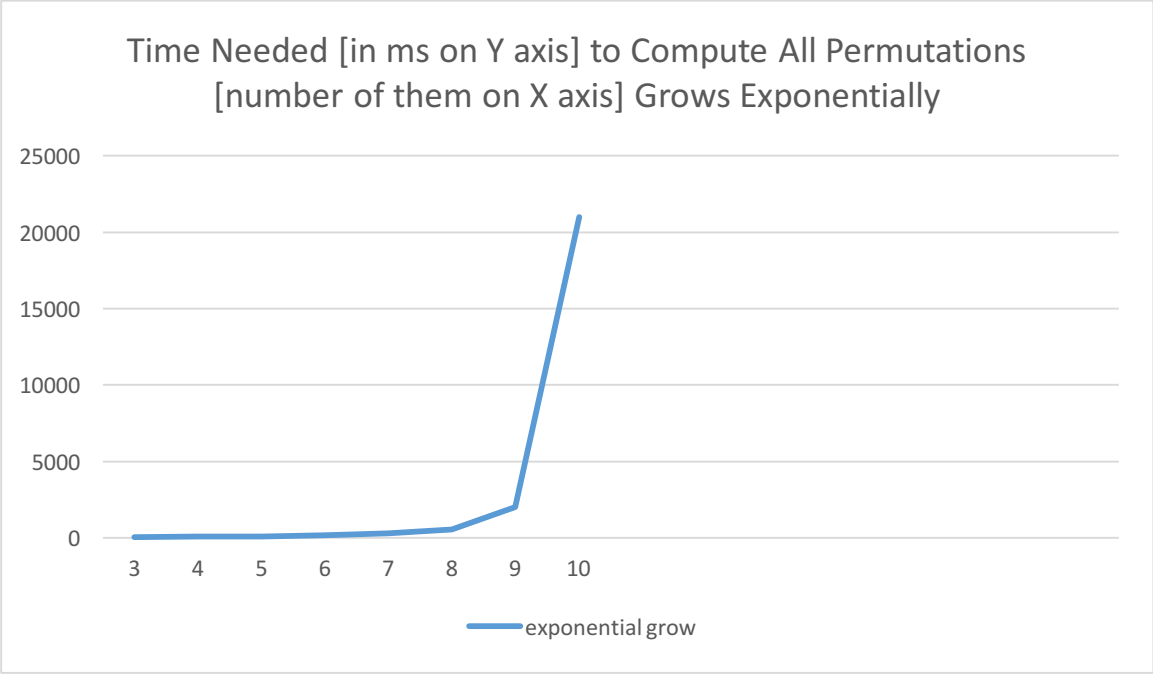

Figure 3. Exponential grow of the ES.

5.3 Simulated Annealing vs Iterated Dynasearch

Obtaining satisfying results out of the SA was not straightforward as it required tuning parameters – the starting and end temperatures plus the cooling rate. The best results were given with the initial temperature set on 100.000 and the cooling rate on the level 0,003. The GUI for that is shown below in 'Figure 5'.

Experimental investigations, made under these conditions, have been repeated one hundred times to get the right and averaged results. 'Figure 6' represents the outcomes from testing instance of 40 jobs produced by SA against ID. Even though in many cases SA has produced the same results as the second algorithm, the discrepancies for some jobs are significant.

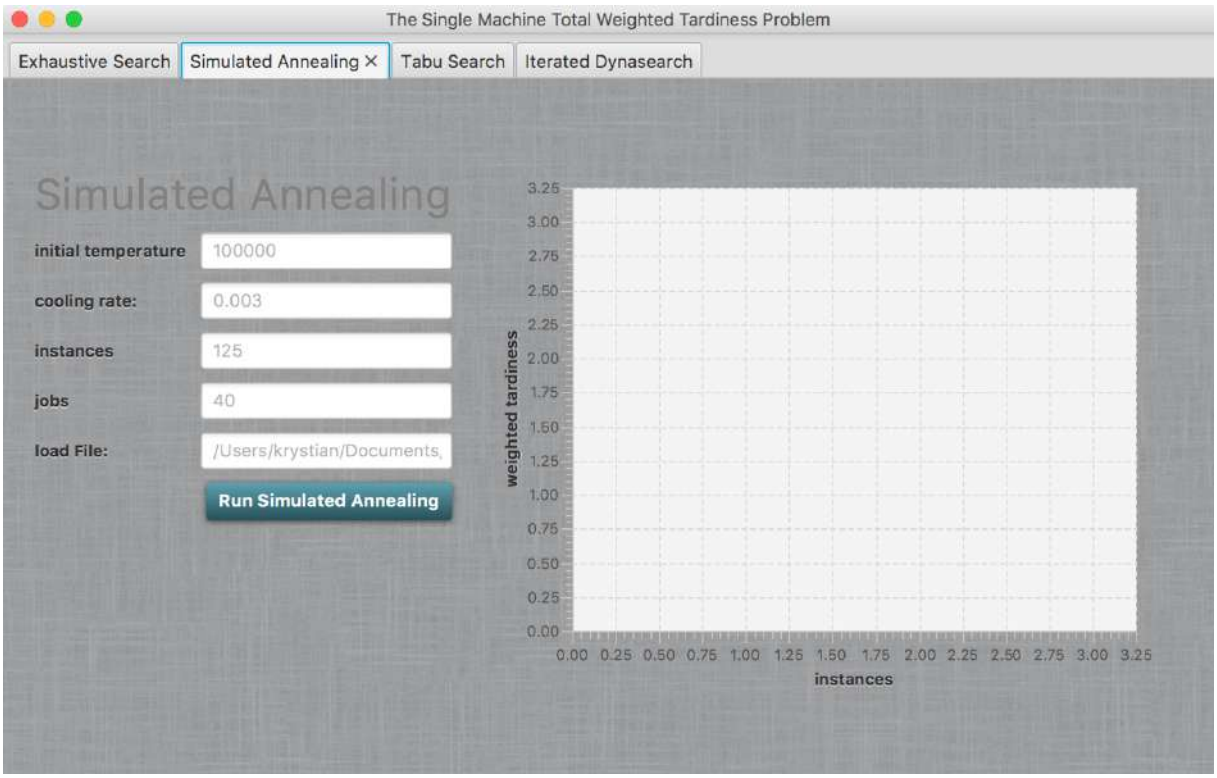

Figure 4. Graphic User Interface for SA.

The same results are presented in 'Figure 8'. 125 instances, 40 jobs each are grouped in 10 containers. The lower the tardiness the better the result. This bar graph clearly shows, that the biggest difference is for container 9, which correspond to the outcomes seen on the 'Figure 7' (15,750 tardiness against 12,740).

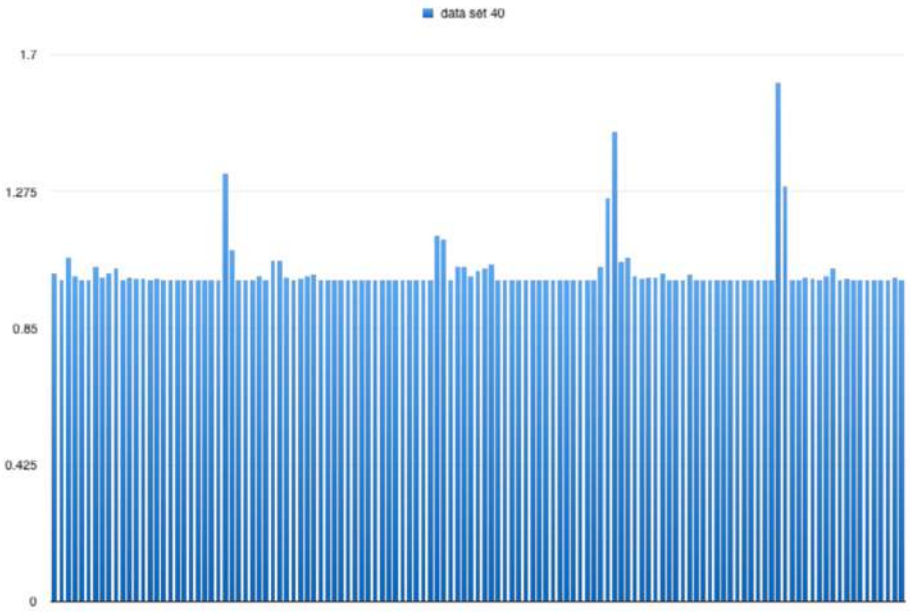

Figure 5. ID divided by SA

*(initial temp = 100 000, cooling rate = 0.003) * 100.*

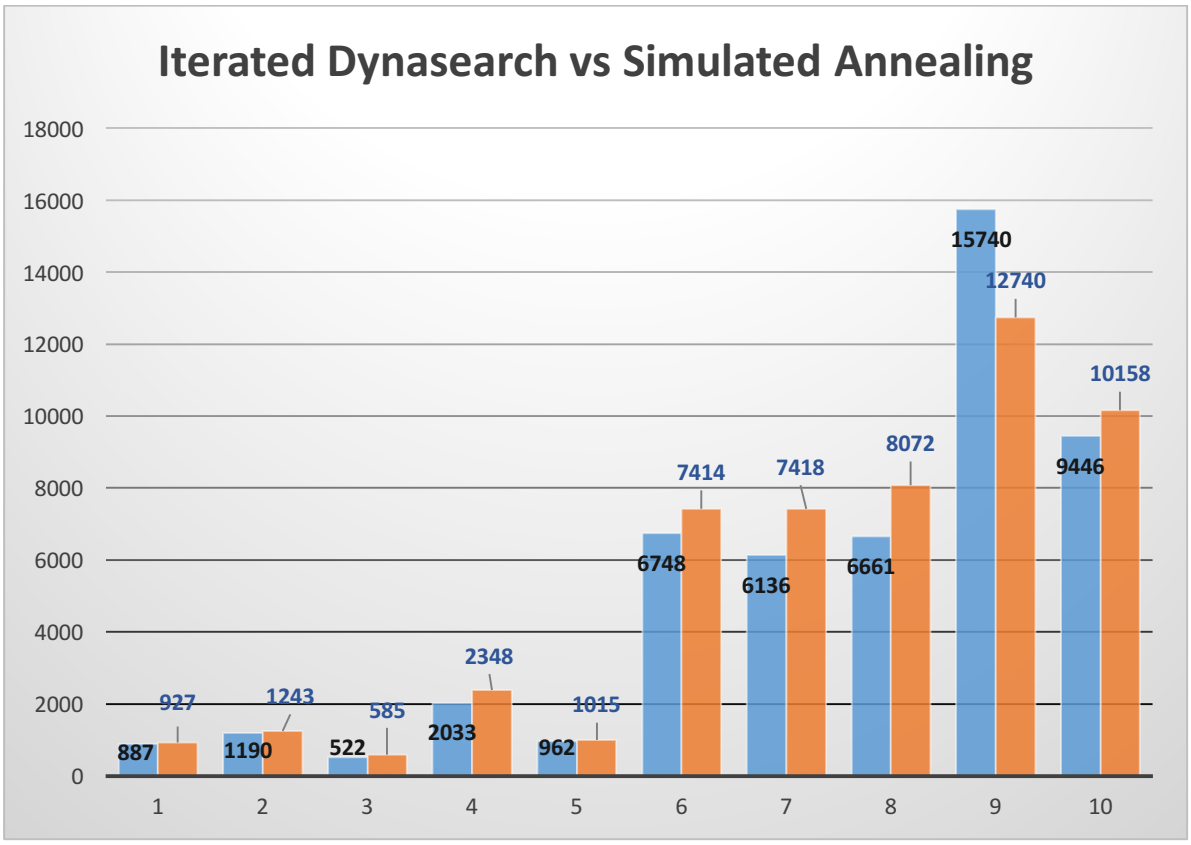

Figure 6. Figure 4 revisited. Jobs are grouped in 10 containers. SA is blue, ID orange.

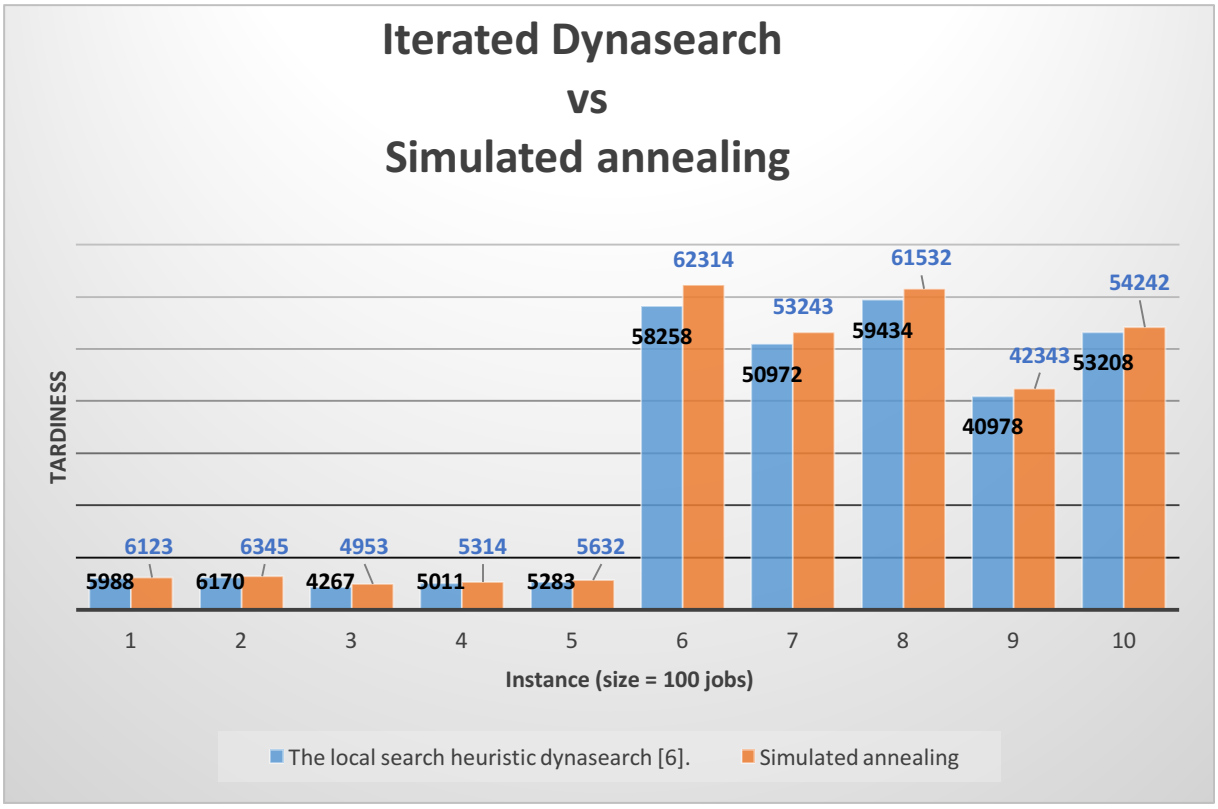

Figure 7. ID vs SA for 100 jobs.

‘Figure 7’ and ‘Figure 8’ present comparisons of SA and ID. The 125 instances are grouped in 10 containers (visible on the horizontal axis), each represents 12/13 jobs – this is applied in order to avoid blurriness.

The situation keeps steady while we have to compare ID with SA for the instance of 100 jobs. The result presented in ‘Figure 8’ shows, that ID outperforms remarkably the first algorithm for the second half of the data sets in every respect. This bar graph skyrockets for instances 6-10 due to specific values in the benchmark data.

To conclude, ID remains unbeaten by the SA no matter its parameters. The results are similar for some instances, but vary greatly for the considerate number of them as well. We can observe the used data sets are of some sort biased and that the random values are not distributed equally.

5.4 Investigations of Tabu Search against Iterated Dynasearch

TS was implemented based on the guidelines given in [2, 3]. The aspiration criterion states, that when the tardiness of the tabu solution is not worse than the tardiness of the best-known, this is accepted even so it is in the tabu. The termination criterion is set for 50 iterations with no improvement of the cost. Its interface is presented in ‘Figure 9’.

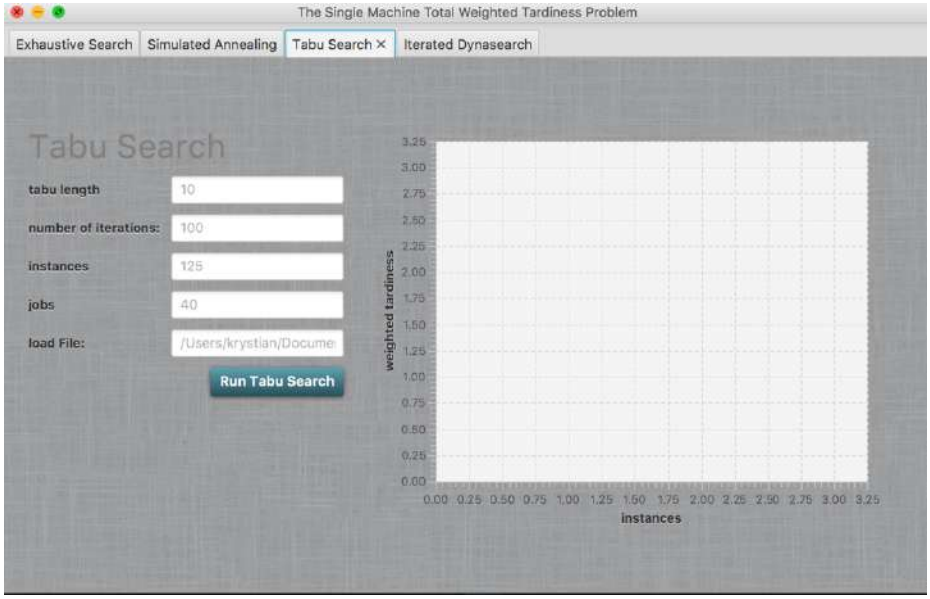

Figure 8. Graphic User Interface for TS.

The results provided in ‘Figure 10’ document, that the TS has not computed all solutions since we can see some lack of data in regular gaps. Also in this case the ID surpassed TS evidently, which calculated a bit worse outcomes then the SA.

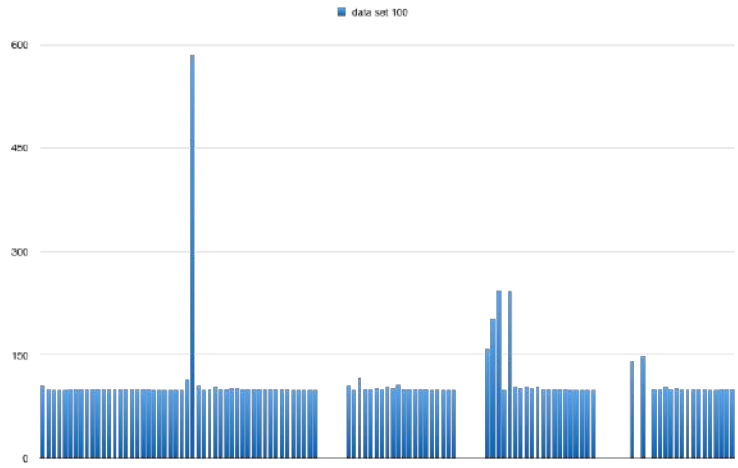

Figure 9. TS divided by ID.

Chapter 6. Conclusion

An attempt to find better solutions than [6] have been failing since 1998 including this project. Nevertheless, the newest research of Breakout Dynasearch [7] states, that it solves all the benchmark in less than 0,1 second providing exact solutions. However, there is a need for repeating their investigations and confirming the results by many science bodies before applying it in industries [1].

Conducted experimental investigations have proven that the SA with very precise calibrated parameters can be as good as the best algorithms solving this problem (ID) in many cases, while the TS algorithm – in its rudimentary form – does not perform that well. Only with applied special speedups and manipulating with the acceptance probability might provide promising results.

To conclude, ID finds optimal solutions and therefore the better results cannot be discovered. After almost two decades, that was proven by the newest breakthrough on this field [7]. On top of that, the benchmark sets that have been used in the literature are becoming out-of-date and do not pose a big challenge any more. An interesting point of view about these benchmarks is provided in [10], which may become a new standard and replacement for the sets from OR-Library [13].

The SMTWTP has been solved by four different methods leading to different solution's quality. The comparisons of the used algorithm prove its effectiveness in this domain and demonstrate their flexibility and the importance of tuning their parameters. Further research is needed to face the implemented algorithms with the BDS along with using the new standard datasets [10

References

[1] Benlic, U., & Hao, J. K. (2013). Breakout local search for the max-cut problem. Engineering Applications of Artificial Intelligence, 26(3), 1162-1173.

[2] Bilge, Ü., Kurtulan, M., & Kıraç, F. (2007). A tabu search algorithm for the single machine total weighted tardiness problem. European Journal of Operational Research, 176(3), 1423-1435.

[3] Cesaret, B., Oğuz, C., & Salman, F. S. (2012). A tabu search algorithm for order acceptance and scheduling. Computers & Operations Research, 39(6), 1197-1205.

[4] Congram, R. K., Potts, C. N., & van de Velde, S. L. (2002). An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. INFORMS Journal on Computing, 14(1), 52-67.

[5] Cormen, T. H. (2009). Introduction to algorithms. MIT press.

[6] Crauwels, H. A. J., Potts, C. N., & Van Wassenhove, L. N. (1998). Local search heuristics for the single machine total weighted tardiness scheduling problem. INFORMS Journal on computing, 10(3), 341-350.

[7] Ding, J., Lü, Z., Cheng, T. C. E., & Xu, L. (2016). Breakout dynasearch for the single-machine total weighted tardiness problem. Computers & Industrial Engineering, 98, 1-10.

[8] Fu, Z. H., & Hao, J. K. (2014). Breakout local search for the Steiner tree problem with revenue, budget and hop constraints. European Journal of Operational Research, 232(1), 209-220.

[9] Geiger, M. J. (2009). The Single Machine Total Weighted Tardiness Problem-Is it (for Metaheuristics) a Solved Problem?. arXiv preprint arXiv:0907.2990.

[10] Geiger, M. J. (2010). New instances for the single machine total weighted tardiness problem.

[11] Heap's algorithm – Details of the algorithm. (2016, August 28). Retrieved from https://en.wikipedia.org/wiki/Heap%27s_algorithm

[12] Huang, W. H., Chang, P. C., Lim, M. H., & Zhang, Z. (2012). Memes co-evolution strategies for fast convergence in solving single machine scheduling problems. International Journal of Production Research, 50(24), 7357-7377.

[13] J.E. Beasley, Operations Research Library, Brunel University, West London (2005). Retrieved from http://people.brunel.ac.uk/~mastjjb/jeb/info.html [Available online: August 2016].

[14] Lenstra, J. K., Kan, A. R., & Brucker, P. (1977). Complexity of machine scheduling problems. Annals of discrete mathematics, 1, 343-362.

[15] Potts, C. N., & van Wassenhove, L. N. (1991). Single machine tardiness sequencing heuristics. IIE transactions, 23(4), 346-354.

List of figures

List of tables

List of algorithms

List of abbreviations

SMTWTP	Single Machine Total Weighted Tardiness Problem
ES	Exhaustive Search
SA	Simulated Annealing
TS	Tabu Search
ID	Iterated Dynasearch
BDS	Breakout Dynasearch

POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Informatyka (INF)
SPECJALNOŚĆ: Advanced Informatics and Control (AIC)

PRACA DYPLOMOWA MAGISTERSKA

Jednoprocesorowy problem szeregowania zadań przy kryterium minimalizacji ważonej sumy opóźnień zadań – algorytmy i badania eksperymentalne

The single-machine total weighted tardiness problem – algorithms and experimental investigations

AUTOR:
Krystian Suliński

PROWADZĄCY PRACĘ:
dr inż. Leszek Koszałka

OCENA PRACY:

WROCŁAW 2016

Streszczenie pracy w języku polskim

Niniejsza praca dyplomowa (magisterska) przedstawia możliwe rozwiązania problemu optymalizacyjnego z dziedziny algorytmów szeregowania zadań. Jednoprocesorowy problem szeregowania zadań przy kryterium minimalizacji ważonej sumy opóźnień zadań to problem szczególnie popularny, z którego doświadczenia często czerpią korzyści inne dziedziny nauki, jak np. chemia, ogólnie pojęta inżynieria czy produkcja.

Praca składa się z sześciu rozdziałów: wstępu, przedstawienia relewantnej literatury, opisu algorytmów, zaprezentowania systemu eksperymentacyjnego, przeprowadzenia badań wydajnościowych oraz wniosków. Całość wsparta jest piętnastoma pozycjami fachowej literatury, w tym przełomowego artykułu sprzed kilku miesięcy, który zmienia bieg rzeczy w całej dziedzinie problemu – mianowicie po prawie 20 latach badań, ostatecznie wynaleziono algorytm, który rozwiązuje rozważane tutaj instancje problemów w sposób optymalny – jednakowoż obecnie naukowcy oczekują na powtórzenie badań oraz potwierdzenie ich niewiarygodnych wręcz wyników.

Wstęp przedstawia problem i ukazuje jego aktualne zastosowania w szeroko pojętym przemyśle. Jest w nim również zawarty cel i zakres pracy. Głównym motywem pracy jest próba podjęcia wyzwania i rozwiązania nierozwiązanych dotychczas (z zastrzeżeniem z akapitu powyżej) instancji problemu, a jej zakres obejmuje implementację czterech popularnych algorytmów – przeglądu zupełnego, symulowanego wyżarzania, przeszukiwania tabu oraz iteracyjnego przeszukiwania dyna. Jest w nim również precyzyjnie sformułowany problem w zapisie matematycznym oraz szczegółowo opisanie dane użyte do testów.

Rozdział drugi dostarcza opisu trzech znakomitych pozycji literaturowych wraz z uzasadnieniem ich wyboru – artykuł o najlepszym algorytmie dla tego problemu (iteracyjne przeszukiwanie dyna), lokalne heurystyczne przeszukiwanie i jego wykorzystanie oraz breakout przeszukiwanie dyna, które rzuca świeże i odważne spojrzenie na dziedzinę problemów NP-trudnych.

Kolejny rozdział traktuje o implementowanych algorytmach, wyjaśniając przy okazji dlaczego metaheurystyka jest tu niezbędna. Jako dowód przedstawiony jest przegląd zupełny, który zawsze znajduje optymalne rozwiązania – o ile tylko one jednak istnieją. Symulowane wyżarzanie zostało odniesione do jego pochodzenia z metalurgii oraz przedstawiono jego trzy parametry do kalibracji – sposób chłodzenia, startowa i końcowa temperatura oraz małe iteracje. Podczas opisu przeszukiwania tabu i rozpisania jego parametrów również nadmieniono, że algorytmy metaheurystyczne wymagają niezwykłej delikatność i precyzji w kalibracji ich parametrów. Iteracyjne przeszukiwanie dyna zostało zaprezentowane m. in. w trzech tabelach – danych instancji, najlepszej-poprawy wymiany oraz przeszukiwania dyna wymiany, które ukazują rdzeń algorytmu i dowodzą jego pożyteczności.

Zawarty w rozdziale czwartym system eksperymentacyjny został podzielony na dwie części – implementację czterech wspominanych algorytmów – wraz z pseudo kodem – oraz graficzny interfejs użytkownika. Algorytmy zostały zaimplementowane i testowane w języku Java 8, a ich najważniejsze elementy kodu opisane. Ten rozdział jest kluczowy dla całej pracy, ponieważ spaja wcześniejsze informacje zaczerpnięte z literatury oraz nadchodzące badania eksperymentacyjne.

Rozdział piąty przedstawia środowisko testowe i jego cechy oraz opisuje wyniki przeprowadzonych badań. Przegląd zupełny doskonale sobie poradził z instancjami mniejszymi aniżeli 10, ponieważ już z większymi wyrzucał wyjątek przekroczenia pamięci komputera. Symulowane wyżarzanie odniosło w wielu aspektach podobne wyniki co iteracyjne przeszukiwanie dyna, jednakże to drugie znakomicie lepiej sobie poradziło dla większych instancji problemu. Przeszukiwanie tabu nie rozwiązało wszystkich instancji, a te co rozwiązało zostało pokonane przez symulowanie wyżarzanie.

Rozdział szósty podsumowuje pracę, że podjęta próba odniesienia lepszych rezultatów niż te sprzed prawie 20 lat nie została zakończona sukcesem, niemniej jednak zaimplementowane algorytmy spisały się zgodnie z oczekiwaniami. Nadmienione zostało również, że oczekuje się potwierdzenia przełomowego przeszukiwania breakout oraz że wykorzystanie bardziej skrupulatnych danych testowych mogło by przynieść lepsze rezultaty, nie obarczając przy tym środowiska eksperymentacyjnego żadnym błędem.