Proceedings of the Federated Conference on Computer Science and Information Systems pp. 561–570 DOI: 10.15439/2016F201 ACSIS, Vol. 8. ISSN 2300-5963

Heuristics for Job Scheduling Reoptimization

Elad Iwanir, Tami Tamir
School of Computer Science
The Interdisciplinary Center, Herzliya, Israel

***Abstract*—Many real-life applications involve systems that change dynamically over time. Thus, throughout the continuous operation of such a system, it is required to compute solutions for new problem instances, derived from previous instances. Since the transition from one solution to another incurs some cost, a natural goal is to have the solution for the new instance close to the original one (under a certain distance measure). We study reoptimization problems arising in scheduling systems. Formally, due to changes in the environment (out-of-order or new machines, modified jobs' processing requirements, etc.), the schedule needs to be modified. That is, jobs might be migrated from their current machine to a different one. Migrations are associated with a cost – due to relocation overhead and machine set-up times. In some systems, a migration is also associated with job extension. The goal is to find a good modified schedule, with a low transition cost from the initial one.**

We consider reoptimization with respect to the classical objectives of minimum makespan and minimum total flow-time. We first prove that the reoptimization variants of both problems are NP-hard, already for very restricted classes. We then develop and present several heuristics for each objective, implement these heuristics, compare their performance on various classes of instances and analyze the results.

I. Introduction

REOPTIMIZATION problems arise naturally in dynamic scheduling environments, such as manufacturing systems and virtual machine managers. Due to changes in the environment (out-of-order or new resources, modified jobs' processing requirements, etc.), the schedule needs to be modified. That is, jobs may be migrated from their current machine to a different one. Migrations are associated with a cost due to relocation overhead and machine set-up times. In some systems, a migration is also associated with job extension. The goal is to find a good modified schedule, with a low transition cost from the initial one.

This work studies the reoptimization variant of two classical scheduling problems in a system with identical parallel machines: (i) minimizing the total flow-time (denoted in standard scheduling notation by $P||\Sigma C_j$ [13]), and (ii) minimum makespan (denoted by $P||C_{max}$).

The minimum total flow-time problem for identical machines can be solved efficiently by the simple greedy Shortest Processing Time algorithm (SPT) that assigns the jobs in non decreasing order by their length. The minimum makespan problem is NP-hard, and has several efficient approximation algorithms, as well as a polynomial-time approximation scheme [15]. These algorithms, as many other algorithms for combinatorial optimization problems, solve the problem from scratch, for a single arbitrary instance without having any constraints or preferences regarding the required solution - as long as they achieve the optimal objective value. However, many of the real-life scenarios motivating these problems involve systems that change dynamically over time. Thus, throughout the continuous operation of such a system, it is required to compute solutions for new problem instances, derived from previous instances. Moreover, since the transition from one solution to another consumes energy (used for the physical migration of the job, for warm-up or set-up of the machines, or for activation of the new machines), a natural goal is to have the solution for the new instance close to the original one (under certain distance measure). Solving a reoptimization problem involves two challenges:

1) Computing an optimal (or close to the optimal) solution for the new instance.
2) Efficiently converting the current solution to the new one.

Each of these challenges, even when considered alone, gives rise to many theoretical and practical questions. Obviously, combining the two challenges is an important goal, which shows up in many applications.

A. Problem description

An instance of our problem consists of a set J of n jobs and a set M_0 of m_0 identical machines. Denote by p_j the processing time of job j. An initial schedule S_0 of the jobs is given. That is, for every machine, it is specified what are the jobs it processes. At any time, a machine can process at most one job and a job can be processed by at most one machine. We consider the scenario in which a change in the system occurs. Possible changes include addition or removal of machines and/or jobs, as well as modification of processing times of jobs in J. We denote by M the set of machines in the modified instance and let $m = |M|$.

Our goal is to suggest a new schedule, S, for the modified instance, with good objective value and small transition cost form S_0. Assignment of a job to a different machine in S_0 and S is denoted *migration* and is associated with a cost. Formally, we are given a *price list* $\theta_{i,i',j}$, such that it costs $\theta_{i,i',j}$ to migrate job j from machine i to machine i'. Moreover, in some systems job migrations are also associated with an extension of the job's processing time. Formally, in addition to the transition costs, we are given a job-extension penalty list $\delta_{i,i',j} \geq 0$, such that the processing time of job j is extended to $p_j + \delta_{i,i',j}$ when it is migrated from machine i to machine i'.

For a given schedule, let C_j denote the *flow-time* (also known as 'completion time') of job j, that is, the time when the process of j completes. The *makespan* of a schedule is defined as the maximal completion time of a job, that is, $C_{max} = \max_j C_j$.

While the schedule does not specify the internal order of jobs assigned to a machine, we assume throughout this work that jobs assigned to a specific machine are always processed in SPT-order (Shortest Processing Time), that is, $p_1 \leq p_2 \leq \ldots$. For a given set of jobs, SPT-order is known to achieve the minimal possible total flow-time, $\min \sum_j C_j$ [22], [8]. Clearly, the internal order has no effect on the makespan.

Given S_0, J and M, the goal is to find a good schedule for J that is close to the initial schedule S_0. We consider two problems:

1) Rescheduling to an optimal schedule using the minimal possible transition cost.
2) Given a budget B, find the best possible modified schedule that can be achieved without exceeding the budget B.

If the modification includes machines' removal, and the budget is limited, then we assume that a feasible solution exists. That is, the budget is at least the cost of migrating the jobs on the removed machines to some machine. Formally, $B \geq \sum_{j \ on \ i \in M_0 \setminus M} \min_{i' \in M} \theta_{i,i',j}$.

Applications: Our reoptimization problems arise naturally in manufacturing systems, where jobs may be migrated among production lines. Due to unexpected changes in the environment (out-of-order or new machines, timetables of task processing, etc.), the production schedule needs to be modified. Rescheduling tasks involves energy-loss due to relocation overhead and machine set-up times. In fact, our work is relevant to any dynamic scheduling environment, in which migrations of jobs are allowed though associated with an overhead caused due to the need to handle the modification and to absorb the migrating jobs in their new assignment.

With the proliferation of cloud computing, more and more applications are deployed in the data centers. Live migration is a common process in which a running virtual machine (VM) or application moves between different physical machines without disconnecting the client or application [7]. Memory, storage, and network connectivity of the virtual machine are transferred from the original host machine to the destination. Such migrations involve a warm-up phase, and a memory-copy phase. In pre-copy memory migration, the Hypervisor typically copies all the memory pages from source to destination while the VM is still running on the source. Alternatively, in post-copy memory migration the VM is suspended, a minimal subset of the execution state of the VM (CPU state, registers and, optionally, non-pageable memory) is transferred to the target, and the VM is then resumed at the target. Live migration is performed in several VM managers such as Parallels Virtuozzo [18] and Xen [24]. Sequential processing of jobs that might be migrated among several processors is performed also in several implementations of MapReduce (e.g., [3]), and in RPC (Remote Procedure Call) services, in which virtual servers can be temporarily rented [4].

B. Related Work

The work on reoptimization problems started with the analysis of dynamic graph problems (see e.g. [10], [23]). These works focus on developing data structures supporting update and query operations on graphs. A different line of research, deals with the computation of a good solution for an NP-hard problem, given an optimal solution for a close instance. Among the problems studied in this setting are TSP [1], [5] and Steiner Tree on weighted graphs [11].

The paper [21] suggests the framework we adopt for this work, in which the solution for the modified instance is evaluated also with respect to its difference from the initial solution. This framework is in use also in [20], to analyze algorithms for data placement in storage area network. Job scheduling reoptimization problems with respect to the total flow-time objective was studied in [2], were algorithms for finding an optimal scheduled using the minimal possible transition cost and algorithms for optimal utilization of a limited number of migration were described.

Lot of attention, in both the industry and the academia is given recently to the problem of minimizing the overhead associated with migrations (see e.g., [7], [14]). Using our notations, this refers to minimizing the transition costs and the job-extension penalties associated with rescheduling a job. Our work focuses in determining the best possible schedule given these costs.

C. Our Contribution

Our study includes both theoretical and comprehensive experimental results. We consider reoptimization with respect to the two classical objectives of minimum makespan and minimum total flow-time, and distinguish between instances with unlimited and limited budget. Our results are presented in Table I. For completeness, we include in the table previous results regarding the minimum total flow-time with unlimited budget [2].

We first analyze the computational complexity status of these problems. While the hardness result for the minimum makespan problem is straightforward, the hardness for the minimum total flow-time problem with limited budget is complex and a bit surprising - given that the minimum flow-time problem is solvable even on unrelated machines

[6], [16], and that the dual variant of achieving an optimal reschedule using minimum budget is also solvable [2]. Our hardness results are valid already for very restricted classes, with a single added machine and no job-extension penalties.

In order to be able to evaluate our heuristics against an optimal solution, we develop and implement an optimal solver based on *branch and bound* technique. Naturally, the solver could not handle very large instances, but we were able to run it against small but diverse instances to compare the different heuristics to the optimum. For example, problems instances with 4 machines and 20 jobs were easily solved.

We then present several heuristics for each objective function. Some of the heuristics distinguish between modifications that involve addition or removal of machines. For both objectives we also developed and applied a genetic algorithm [9], [19]. All the heuristics were implemented, their performance on various classes of instances have been compared, and the results were analyzed. Our experimental study concludes the paper.

II. Computational Complexity

In this section we analyze the computational complexity of reoptimization scheduling problems. We distinguish between the minimum makespan and the minimum total flow problems, as well as between the problem of finding an optimal solution using minimum budget and finding the best solution that can be obtained using limited budget. Due to space constraints, some of the proofs in this section are omitted.

We use the following notations: For a multiset $A = \{a_1, a_2, \ldots, a_{|A|}\}$ of integers, let $MAX(A) = \max_{j=1}^{|A|} a_j$ and $SUM(A) = \sum_{j=1}^{|A|} a_j$. Also, let $\vec{A}$ denote the vector consisting of the elements of A in non-decreasing order and define $SPT(A) = \vec{A} \cdot (|A|, \ldots, 2, 1)$. For example, for $A = \{5, 3, 1, 5, 8\}$ it holds that $SUM(A) = 22$, $\vec{A} = (1, 3, 5, 5, 8)$ and $SPT(A) = (1, 3, 5, 5, 8) \cdot (5, 4, 3, 2, 1) = 5 + 12 + 15 + 10 + 8 = 50$. Note that $SPT(A)$ is the value of an optimal solution for the minimum total-flow problem on a single machine for an instance consisting of $|A|$ jobs with lengths in A.

For the minimum makespan reoptimization problem, our result is not surprising - the classical load-balancing problem $P||C_{max}$ is known to be NP-complete even with no transition costs or extensions. For completeness we show that this hardness carry over to the simplest class of the reoptimization variant.

Theorem 2.1: The minimum makespan reoptimization problem is NP-complete even with a single added machine, unlimited budget, and no job-extension penalty.

The analysis of the minimum total flow problem is more involved. The corresponding classical optimization problem $P||\sum C_j$ is known to be solvable in polynomial time by the simple SPT algorithm. For the reoptimization problem, an efficient optimal algorithm for finding an optimal solution using minimum budget is presented in [2]. The algorithm is based on a reduction to a minimum weighted perfect matching in a bipartite graph. This reduction cannot be generalized to consider instances with limited budget, and the complexity status of the problem of finding the best solution that can be obtained using limited budget remains open in [2]. We show that this problem is NP-complete, even with no job-extension penalties and a single added machine.

Our proof refers to the compact representation of the problem. In a compact representation, the jobs assigned on each machine are given by a set of pairs $\langle p_j, n_j \rangle$, where n_j is the number of jobs of length p_j assigned on the machine. We first prove two simple observations:

Observation 2.2: Let A' be a subset of A then $SPT(A') + SPT(A \setminus A') < SPT(A)$.

Given a multiset A, and an integer Z, let $x > MAX(A)$. Extend A to a multiset A^* by adding to it Z elements of value x.

Observation 2.3: $SPT(A^*) = \frac{Z(Z+1)}{2}x + Z \cdot SUM(A) + SPT(A)$.

Using the above observations, we are now ready to prove the hardness result.

Theorem 2.4: The minimum total flow-time reoptimization problem with limited budget is NP-complete even with a single added machine, and no job-extension penalty.

Proof: The reduction is from the *Partition* problem: given a set $A = \{a_1, a_2, \ldots, a_n\}$ of integers, whose total sum is $2B$, the goal is to decide whether A has a subset $A' \subset A$ such that $SUM(A') = SUM(A \setminus A') = B$. The Partition problem is known to be NP-complete [12].

Given an instance of Partition $A = \{a_1, a_2, \ldots, a_n\}$, whose total sum is $2B$, let $Z = SPT(A)$. We construct the following instance for the reoptimzation problem: The initial schedule, S_0, consists of a single machine and $n+Z$ jobs. The first n jobs correspond to the Partition elements, that is, for $1 \leq j \leq n$ let $p_j = a_j$. Each of the additional Z *dummy jobs* have length x for some $x > B$. Assume that one machine is added, and that the transition cost of migrating job j from the initial machine to the new machine is p_j. Assume also that the budget is B.

Since the budget is B and each dummy jobs have length more than B, none of these jobs can be migrated to the new machine. Thus, a modified schedule S is characterized by a subset $A' \subset A$ of jobs corresponding to the partition elements, whose total length is at most B. These jobs are migrated to the new machine and assigned in SPT order on it. The remaining jobs are be assigned in SPT order on the initial machine. since x is larger than any a_i, the Z jobs of length x will be assigned after the jobs corresponding to $A \setminus A'$.

Finally, we note that the reduction is polynomial. Calculating $SPT(A)$ requires sorting and is performed in

TABLE I
SUMMARY OF OUR RESULTS.

Objective Function	Optimal Reschedule Using Minimum Budget			Best Reschedule Using Given Limited Budget		
	NP-hard	Optimal Solution	Heuristics	NP-Hard	Optimal Solution	Heuristics
$\min C_{max}$	Yes	Branch and Bound	• LPT-based greedy • Loads-based greedy • Genetic algorithm	Yes	Branch and Bound	• LPT-based greedy • Loads-based greedy • Genetic algorithm
$\min \sum_j C_j$	No [2]	Reduction to min-weight perfect matching [2]	–	Yes	Branch and Bound	• Greedy reversion • Cyclic reversion • SPT-like

time $O(n \log n)$. The instance constructed in the reduction consists of $n + Z$ jobs where $Z = SPT(A)$. The compact representation of this instance includes at most $n + 1$ different pairs, where all $\langle p_j, n_j \rangle$ values are polynomial in n.

Claim 2.5: The minimum total flow-time in an optimal modified schedule is less than $\frac{Z(Z+1)}{2}x + (B+1)Z$ if and only if a partition of A exists.

The above claim completes the hardness proof. ∎

Remark: It is possible that two multisets A, B will have the same cardinality, and that $SUM(A) > SUM(B)$ while $SPT(A) < SPT(B)$. For example, for $A = \{1, 2, 10\}$ and $B = \{3, 4, 5\}$, we have $|A| = |B| = 3$, $SUM(A) = 13 > 12 = SUM(B)$ while $SPT(A) = 15 < 24 = SPT(B)$. This emphasis an additional challenge of the reoptimization problem: an optimal solution may not use the whole budget. Such anomalies also explains why our hardness proof cannot be a simple reduction from the subset-sum problem, and the dummy jobs are required.

III. Optimal Algorithms

A. A Brute-force Solver based on Branch and Bound

Our brute-force solver was designed to utilize high performance multi-core machines in order to find optimal solutions for the problems that were shown to be NP-complete. The solution space for a scheduling problem can be described by a tree of depth n, where depth k corresponds to the assignment of job k, for $1 \leq k \leq n$. Specifically, the root (depth 0) corresponds to an empty schedule - none of the jobs were assigned; at level 1 there are m nodes, representing job 1 being assigned to each of the m machines. At level k there are m^k nodes, corresponding to all possible assignment of the first k jobs. This implies that the brute-force solver may need to consider m^n assignments find the optimal one.

We note that, as detailed in Section I-A, once the partition of jobs among the machines is determined, the internal job order on each machine either has no effect on the solution (in the $\min C_{max}$ problem), or is the unique SPT-order (in the $\min \sum_j C_j$ problem). Thus, the solution space need not distinguish between assignments with different internal order of the same set of jobs on every machine.

Obviously, even without considering different internal orders, iterating over all of the m^n configurations is not feasible when dealing with large instances. Our solver uses a *branch and bound* technique combined with other optimizations to effectively trim tree-branches that are guaranteed not to yield an optimal solution.

In particular, the solver keeps in memory the best solution it found so far (its objective value and its transition cost). When processing a tree node if the already accumulated transition cost is larger than the budget or if the objective-function value is larger than the current best, then the solver can safely discard this tree branch as it is guaranteed not to yield a feasible optimal solution. For partial assignments, the objective-value is calculated by combining the value (makespan or total flow-time) of the already assigned jobs, and a lower bound on the yet-to-be-assigned jobs. For the minimum makespan problem, the lower bound is calculated by assuming perfect load-balancing ($\sum_j p_j/m$), and for total flow-time the lower bound is calculated by assuming SPT-order with no job-extension penalties.

In addition, we find out that considering the jobs from longest to shortest, that is, depth 1 corresponds to the longest job in the initial assignment, etc.) drastically helps in trimming branches earlier in the process.

The solver was designed to use multi-core machines in order to shorten the run time, by doing the work in parallel on the different cores. In the heart of the design stands a concurrent queue to which tasks are enqueued. Different threads concurrently dequeue these tasks, in a consumer-producer like mechanism. A 'task' for that matter is a request to process a tree node. That is, when the solver starts the queue is empty and a task to process the root node is added, which ignites the process. The solver is done when the queue is empty.

The solver's ability to solve problems instances of different sizes is determined by the given machine, to be more specific, by the CPU's clock speed, the number of available cores and sufficient memory (as the entire process is in memory). For example, the solver was able to handle a problem with 9 machines and 20 jobs in about 100 minutes, when ran on a machine with 8 cores and 32GB of RAM memory.

B. An Optimal Algorithm for $\Sigma_j C_j$

An algorithm for finding an optimal reschedule with respect to the minimum total flow-time objective is presented in [2]. The algorithm returns an optimal modified schedule using the minimal possible budget. The algorithm is based on reducing the problem to a minimum-weight complete-matching problem in a bipartite graph. The algorithm fits the most general case - arbitrary modifications, arbitrary transition costs and arbitrary job-extension penalties.

We have implemented this algorithm, and use its results as a benchmark so we can evaluate how well our heuristics perform. The algorithm is based on matching the jobs with possible slots on the machines. For completeness, we give here the technical details that are relevant to its implementation. Recall that n and m represent, respectively, the number of jobs and machines in the modified instance. Let $G = (V, E)$, where $V = J \cup U$. The vertices J correspond to the set of n jobs (a single node per job). The set U consists of mn nodes, q_{ik}, for $i = 1, \dots, m$ and $k = 1, \dots, n$, where node q_{ik} represents the k^{th} from last position on machine i. The edge set E includes an edge (v_j, q_{ik}) for every node in J and every node in U (a complete bipartite graph). The edge weights consist of two components: a dominant component corresponding to the contribution of a job assigned in a specific position to the total flow-time, and a minor component corresponding to the associated transition cost. Both components are combined to form a single weight. Formally, for a large constant Z,

- For every job that is assigned to i in S_0, let $w(v_j, q_{ik}) = Zkp_j$.
- For every $i' \neq i$, let $w(v_j, q_{i'k}) = Zk(p_j + \delta_{i,i',j}) + \theta_{i,i',j}$.

These weights are based on the observation that a job assigned to the k-th from last position, contributes k times its processing-time to the total flow-time (see details in [2]). We implemented the algorithm by using the Hungarian method [17], a combinatorial optimization algorithm that solves the assignment problem in polynomial time. The solver's run time is $O(|V|^3)$, where $|V| = n(m + 1)$. In practice, this optimal solver can easily handle instances with 30 machines and 300 jobs.

IV. Our Heuristics

In this section we describe the heuristics we have designed and implemented. Some heuristics were designed for specific modification (e.g. machines removal, limited budget), or for specific objective function, while some are general and fit all our reoptimization variants.

A. Heuristics for Minimum Makespan

We suggest two greedy heuristics, both intended to solve the *minimal makespan* problem ($\min C_{max}$). In the first, we select the next migration to be performed according to the job's processing times, while in the second, we select the next migration according to the loads on the machines. Both algorithms begin with S_0 as the initial configuration. If the modification involves machines' removal, we first perform migrations of jobs assigned to the removed machines and migrate each such job j assigned to a removed machine i, to a machine $i' \in M$ for which $\theta_{i,i',j}$ is minimal. Ties are broken in favor of short extension penalty. As mentioned in the introduction, we assume that the budget is sufficient for this reschedule, as otherwise no feasible solution exists. Following the above preprocessing, we perform the following:

1. LPT-Based: In every iteration we consider the jobs in non-increasing processing-time order. When considering job j, we check whether migrating it to one of the two least loaded machines increases the load-balancing, formally, assume j is assigned to machine i, and we consider moving it to machine i', we check whether $p_j + \theta_{i,i',j} + L_{i'} < L_i$. If the answer is positive and the remaining budget allows, the migration is performed. We repeat the iterations until a complete pass over the jobs yields no migration.

2. Loads-Based: In every iteration we try to migrate some job out of the most loaded machine. We first consider the pair of most-loaded and least-loaded machines. Denote these machines by i and i'. We consider jobs on machine i according to order $\theta_{i,i',1} \leq \theta_{i,i',2} \leq \dots$. When considering job j we check whether migrating it to machine i' increases the load-balancing, that is, $p_j + \theta_{i,i',j} + L_{i'} < L_i$. If the answer is positive and the remaining budget allows, the migration is performed, and a new iteration begins (maybe with a different pair of most- and least-loaded machines). If the answer is negative for all the jobs on i, we move to the next candidate for target machine i' - the second least-loaded machine, etc. The iteration ends when some beneficial migration from the most-loaded machine is detected. If none such migration exists, the algorithm terminates.

B. Heuristics for Minimum Total Flow-time

The minimum total flow-time reoptimization problem can be solved optimally assuming unlimited budged. While the optimal algorithm presented in Section III-B solves optimally the $\Sigma_j C_j$ problem using the minimal possible budget, it cannot be modified to solve the problem when the budget is limited. In fact, as shown in Section II, this variant is NP-hard. We propose two heuristics that use the optimal algorithm as a first step and then each, in its own way, change the assignment to reach a feasible solution which obeys the budget constraints. A third algorithm that we propose, tries to reach an SPT-like schedule.

1. Greedy Reversion: The optimal algorithm returns an assignment S minimizing the total flow-time, which might not conform to the budget limitation. The following steps are performed to reach a feasible solution. First, we sort all the jobs which migrated in the transition from S_0 to S in non-increasing order according to the transition cost their migration caused.

We then distinguish between two cases:

1) The modification consists of only machines' *addition*. We revert the transitions one by one until we reach an allowed budget.
2) The modification includes machines' *removal*. We revert the transitions of jobs which do not originate from a removed machine, one by one until we reach an allowed budget. If after all possible reverts were done, the budget is still not met, we continue to the next step: 'Handling jobs of removed machines'.

Handling jobs of removed machines: This step is performed only when removed machines are involved, and all the jobs assigned to remaining machines are back in their initial machines. We sort the jobs originated from removed machines in non-increasing order according to the transition cost their migration (determined by the optimal algorithm) caused. Job after job, we migrate a job j assigned in S_0 to a removed machine $i \in M_0 \setminus M$ to the machine $i' \in M$ for which $\theta_{i,i',j}$ is minimal, breaking ties in favor of better objective value. As explained in the introduction, we assume that the budget is sufficient to complete all these migrations.

2. Cyclic Reversion: The optimal algorithm returns an assignment S minimizing the total flow-time, which might not conform to the budget limitation. Similar to the previous heuristic, we choose the most expensive transition involved, denote by j the corresponding job. We migrate job j back to its origin machine, $M_{0,j}$. Since we wish to keep jobs distributed as evenly as possible, we now choose a job that migrated to $M_{0,j}$ and migrate it back to its initial machine. We choose the job whose migration to $M_{0,j}$ was most expensive. We keep these cyclic reverts until one of following conditions holds: a) We made a complete loop and reached back the machine from which we started. b) We have reached a machine to which no job was migrated. If the budget conforms to the limitation, we stop, Otherwise we choose a job with the most expensive transition cost and start a new revert cycle.

If the modification includes machines' *removal*, jobs originated from the removed machines cannot be selected to migrate back to their original machine. If the budget is not met after all the allowed reverts were performed, we continue to the step 'Handling jobs of removed machines', as described in the greedy heuristic.

3. SPT-like: It is known that SPT ordering is optimal for $P||\Sigma_j C_j$. We therefore try to reach a schedule that fulfils the following basic properties of an SPT schedule:

1) In any optimal schedule the number of jobs on any machine is either $\left\lfloor \frac{n}{m} \right\rfloor$ or $\left\lceil \frac{n}{m} \right\rceil$.
2) The jobs can be partitioned into $\left\lfloor \frac{n}{m} \right\rfloor$ rounds, such that the k-th round consists of all the jobs that are k from last on some machine. In an SPT schedule, each of the jobs in the k-th round is not shorter than each of the jobs in the $k+1$st round.

This heuristic consists of three stages. The first stage, applied when machines were removed, is to move to some feasible schedule - each of the jobs assigned to a removed machine, is migrated into a machine for which the transition cost is the lowest.

In the second stage, we try to make the machines as balanced as possible in terms of number of jobs. While the budget allows and while there exists a pair of machines, one with more than $\left\lceil \frac{n}{m'} \right\rceil$ jobs and the other with less than $\left\lfloor \frac{n}{m'} \right\rfloor$ jobs, we migrate the cheapest-to-move job on the first machine, to the second one.

In the third phase, we try to make our solution as close to the SPT ordering as possible, we compare the solution round after round to the desired SPT ordering, and switch jobs whenever required, as long as the budget allows.

C. Genetic algorithm

In the Genetic algorithm the idea is for the solution to be obtained in a way that resembles a biological evolution process [9], [19]. That is, we let the method's evolutionary process find the solution by itself. The idea is to define the *Genome* of a single solution and a *Ranking* method $Rank : Genome \rightarrow \mathbb{R}$. The genome of a single solution represents and gives all the needed information regarding the solution. In our case the Genome is simple, for a problem instance with m machines and n jobs, a solution genome id, g, defined as $g = (g_1, g_2, ..., g_n)$, where $g_i \in [1, m]$ and $g_i \notin \{x | \text{x is a removed machine id}\}$, in other words each cell with index i represents the $i - th$ jobs and the cell's value is the machine this job is assigned to in the modified schedule. The ranking method is used to define how good is a given solution. In our case the ranking method sorts the different genomes first by the objective method value (C_{max} or $\Sigma_j C_j$) and then by the transition cost. We create a $population$ (generation 1), which is a collection of $genomes$, we rank each member of the population and sort them from best to worst.

When solving a reoptimization problem with limited budget, to guarantee the algorithm end up with a feasible solution, we create at least one feasible genome in generation1. In the case of 'machines addition', this solution will be S_0 as it is both valid and requires no transition cost. In the case of 'machines removal', a job j assigned in S_0 to a removed machine $i \in M_0 \setminus M$ is assigned in the feasible genome to the machine $i' \in M$ for which $\theta_{i,i',j}$ is minimal. We assume that the budget is sufficient for this reschedule, as otherwise no feasible solution exists.

The next step is the evolutionary-like step, in which we create the next generation according to the following methods:

1) Elitism mechanism: We take the best 5% genomes and move them 'as-is' to the next generation. This guarantee that the next generation will be at least as good as the current one. To deal with the case of limited budget, we also pass 'as-is' the best solution which meets the given budget. This must be done since the genetic process is pushing the genomes population for a better objective values as a primary goal and to minimize the transition cost as a secondary goal.
2) Cross over: From the entire genomes population, we choose randomly 2 elements, denoted g_x and g_y, we choose a pivot point from the range of $ind \in [1, n-1]$, and create two new items:

$$newItem1_i = \begin{cases} g_{x_i} & \text{if } i \leq ind \\ g_{y_i} & \text{if } i > ind \end{cases}$$

and

$$newItem2_i = \begin{cases} g_{y_i} & \text{if } i \leq ind \\ g_{x_i} & \text{if } i > ind \end{cases}$$

43% of the next generation is a result of this operator.
3) Mutate: We choose a random genome, we choose from its genome a random cell and change its value. 43% of the next generation is a result of this operator.
4) Fresh Items: We generate new genomes. These new elements have the potential of shifting the results in a new direction and to help avoid local optimum. 9% of the next generation is be a result of this operator.

Each genome in the newly created population is then re-evaluated, meaning, its score is computed. The process repeats itself until it fails to improve any further and the genome with the best ranking is selected as output from the most recent generation. Obviously if we examine the best solution from generation $i+1$ compared to that of generation i we will notice that the 'quality' of the best solution is non decreasing over the generations as we have the 'Elitism mechanism' which ensures us that the best individuals will survive to the next generation.

V. Experimental results

The datasets for our experiments were created using our own data generator which supports any parameters combination of number of machines and jobs, number of added or removed machines, number of added or removed jobs, as well as the distributions of job lengths, job-extension penalty, and transition costs.

Instances on which we run our heuristics could be very large (hundreds of jobs, and several dozens of machines). Instances on which we run our brute-force solver (introduced in Section III-A) had to be smaller. In particular, we ran the brute-force solver on instances with 20 jobs and 4 machines. We find out that even such small instances can provide a good comparison between different heuristics; therefore, the brute-force solver is helpful for concluding how far from the optimum our heuristics perform. The optimal algorithm for $\min \Sigma_j C_j$, based on a reduction to perfect matching (introduced in Section III-B), was able to handle relatively large instances of 15 machines and 300 jobs. In our basic template for job creation, the jobs' lengths were uniformly distributed in $[1, 20]$. The job-extension penalty of job j was uniformly distributed in $[1, \frac{p_j}{2}]$, and the transition costs were uniformly distributed in $[1, 5]$.

To generate problems instances for a specific experiment we took a template instance, decided on one parameter that will vary in the different runs, and set the rest of the parameters to basic values. For example, to understand how the number of added machines affects the heuristics performance, we fixed all the other parameters (jobs' lengths, transition costs, etc.), and run the heuristics on multiple instances which vary only in the number of added machines. To avoid anomalies, we have generated for each experiment 5 instances with the same parameters, based on 5 templates instances (same configuration, different instance) and considered the average performance as the result.

Another parameter that could affect the performance of our heuristics is the initial assignment - which may be random, SPT or LPT. We found out that in practice the initial assignment does not affect the results significantly and the results we present in the sequel were all generated with a randomize initial assignment - which is a bit more 'challenging' and therefore emphasizes the differences among the heuristics.

The results of our experiments are presented and analyzed below. In all the figures, the bars show the objective value ($\Sigma_j C_j$ or C_{max}), and the lines show the corresponding transition cost.

A. Results for the Minimal Total Flow-Time problem

1) Machines' Addition: The template for heuristics that analyze the $\min \Sigma_j C_j$ problem consists of 15 machines and 300 jobs. We start by showing how the different heuristics performs on instances with both transition costs and job-extension penalties, where the number of added machines was set to $m/2$. As shown in Fig. 1, with unlimited budget the genetic algorithm is the closest to the known optimum, calculated by the perfect matching algorithm result. As budget is limited, the performance of the genetic algorithm drops. As expected, the lower the budget, the higher the objective value. Also, the differences

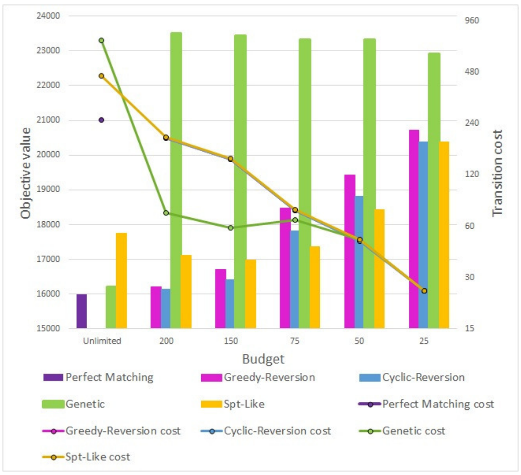

Fig. 1. Results for $\min \Sigma_j C_j$ with $m/2$ added machines and variable budget.

between the heuristics are less significant as the budget decreases, as less transitions are allowed. Interesting fact is that the genetic algorithm has a slight improvement as the limitation is getting stricter. We explain this by the fact that before starting the algorithm we include in the population items that obey the allowed budget. Later, these items are influencing the genetic process and are helping the algorithm to converge to a good solution, better than with a more relaxed budget limitation.

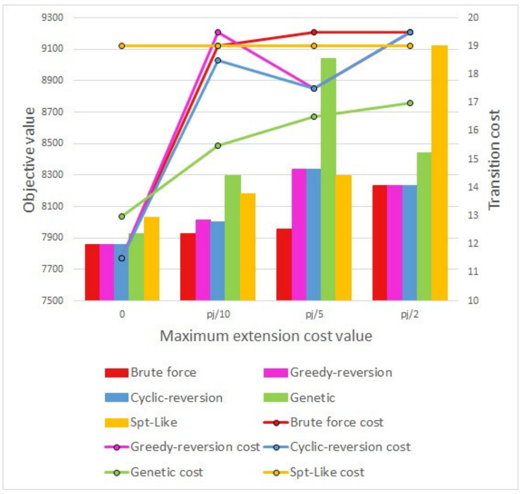

Fig. 2. Results for $\min \Sigma_j C_j$ with variable extension penalty.

The goal of our next experiment was to see how close the heuristics get to the actual optimum. For this test we have used our brute-force solver on relatively small problem instances (4 machines and 20 jobs), two machines were added and the budget was set to 20. The results for various extension penalties are shown in Fig. 2. We observe that both 'Greedy-reversion' and 'Cyclic-reversion' heuristics were very close to the optimum.

2) Machines' Removal: Fig. 3 presents the performance of the different heuristics when the modification is machines' removal and the budget is limited to 150. According to our parameters, this budget is expected to be sufficient for the migration of about 20% of the jobs. The initial assignment was of 300 jobs on 15 machines. Not surprisingly, we see an increase of the objective value as the number of removed machines increases. All of the heuristics performed more or less the same, both in terms of the achieved objective value and in term of the budget utilization, with an exception of the Genetic algorithm which manage to use a significantly lower budget.

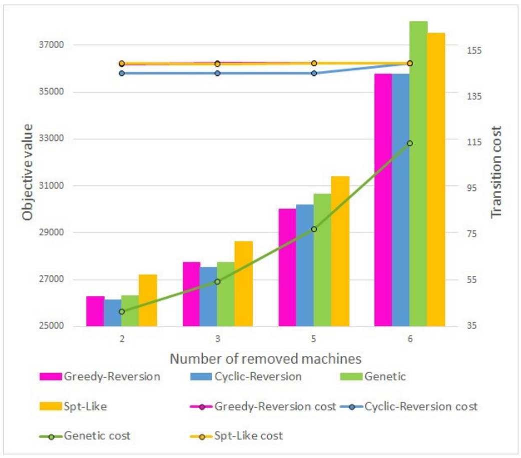

Fig. 3. Results for $\Sigma_j C_j$ for machines' removal and limited budget.

Fig. 4 presents the results for the same instance and the same modification only with unlimited budget. This problem is the one for which we have an efficient optimal algorithm (see Section III-B). The genetic algorithm perform very close to the optimum for every number of removed machines. In fact, for two removed machines its transition cost is lower than the optimum and only slightly higher in the total flow-time (recall that the optimal algorithm 'insists' on finding a reschedule that minimizes the total flow-time). On the other hand the SPT-Like heuristic is both very costly and gives poor results. This can be explained by the fact that insisting on a complete SPT order requires many transition, and involves many job-extension penalties.

B. Results for the Minimum Makespan problem

1) Machines' Addition: Our template for experiments analyzing the $\min C_{max}$ problem consists of 30 machines and 500 jobs. Fig. 5 presents results for adding 15 machines and variable budget. The 'Loads-based' heuristic is the best heuristic. The genetic algorithm perform poorly compared to the other heuristics, but on the other hand, it does not utilize the whole budget.

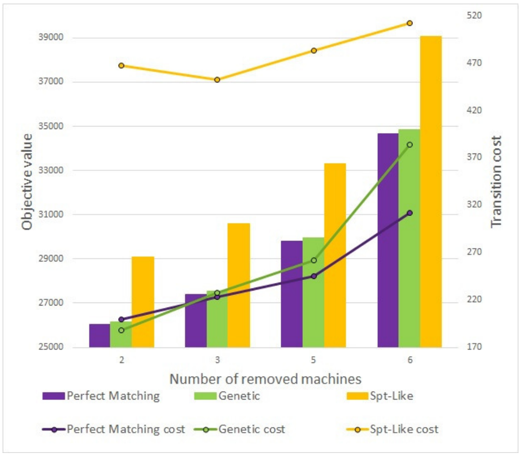

Fig. 4. Results for $\Sigma_j C_j$ for machines' removal and unlimited budget.

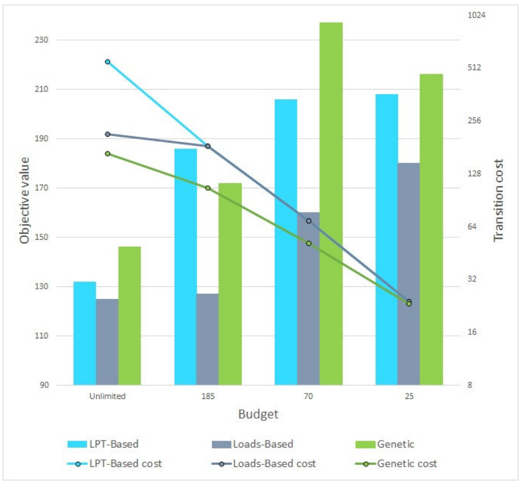

Fig. 5. Results for $\min C_{max}$ with $m/2$ added machines and variable budget.

In the our next experiment we measure how close the heuristics get to the optimum. We have used our brute-force solver on relatively small problem instances, consisting of 4 machines, 20 jobs, and a limiting budget of 20. The results for various extension penalties are shown in Fig. 6. Once again, the 'Loads-based' heuristic outperform the others, and is relatively close to the optimum. The 'LPT-based' heuristic seems to perform (relatively) better as the job-extension penalty increases.

2) Removing machines: Our next experiments compare the performance of the different heuristics when the modification is machines' removal. We performed two experiments - with budget limited to 250 and with unlimited budget. Due to space constraints, the figures are omitted. Our results show that with unlimited budget, all three heuristics (LPT-based, Loads-based and genetic) perform more or less the same. While a similar makespan is achieved, the

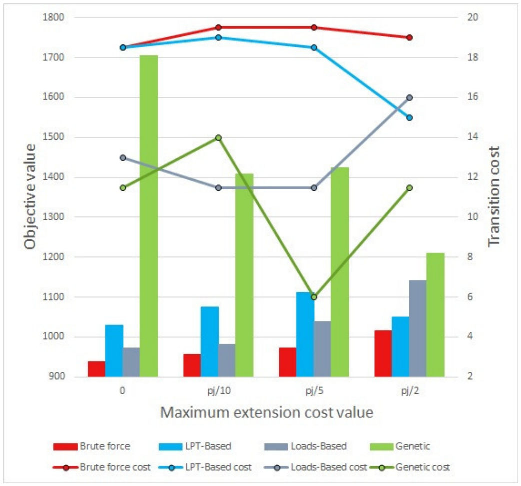

Fig. 6. Results for $\min C_{max}$ with variable extension penalty.

Loads-based heuristic requires the lower transition cost, then the LPT-based (that needs 10% higher cost) and the genetic ($15-20\%$ higher than Loads-based). With limited budget, the Loads-based and LPT-based heuristics perform significantly better than the genetic algorithm, but they also require a much higher budget.

VI. Conclusions and future work

We presented theoretical and experimental results for the reoptimization variant of job scheduling problems. We have shown that the problems of finding the minimum total flow-time with limited budget, finding the minimum makespan with limited budget and finding the minimum makespan with unlimited budget are NP-Complete. We have designed and implemented several heuristics for each of these problems, and performed a comprehensive empirical study to analyze their performance. To see how well these heuristics perform compared to the actual optimum, an efficient branch-and-bound brute-force solver was designed and implemented. An optimal algorithm for the minimum total-flow problem with unlimited budget was also implemented.

In general, our experiments reveal that while the problems are NP-hard, heuristics whose time complexity is polynomial in the number of jobs and machines, perform very well in practice. Simple algorithms, that are based on adjustment of known heuristics for the one-shot problem (with no modifications) are both simple to implement and provide results that are, on average, within $10\%-15\%$ from the optimum. More complex algorithms, that are based on a preprocessing in which a perfect matching algorithm is implemented, perform on average even better.

We have observed that while the Genetic algorithm does not perform well when given a limited budget, it performs relatively well with unlimited budget for the minimum C_{max} problem, and close to optimum for $\Sigma_j C_j$. It also takes a considerable amount of time to run. In some scenarios, its objective value may not be competitive compared to other heuristics, however, its budget utilization is impressively good (see for example Fig. 3). A known issue of genetic algorithm is that the parameters must be carefully tuned in order for the algorithm to converge into a good solution. Future work on this algorithm might refactor the population size or operators we have used (Elitism, Crossover, Mutation) by adding new operators, modify the existing or change the possibilities of each to create a more optimized genetic algorithm. Another direction to explore is to create a dedicated score method for each variant of the problems. For real life applications where budget utilization is a big concern, we are sometimes allowed to use lot of budget but strive to use as less as possible. The genetic algorithm is a good choice for such scenarios.

Our greedy heuristics performed well, both on the $\Sigma_j C_j$ and the C_{max} problems. We have observed that the 'Loads-based' heuristic outperformed the 'LPT-based' both in terms of objective value and budget utilization. With unlimited budget, the budget utilization difference was more significant. For $\min \Sigma_j C_j$, the 'Cyclic-reversion' showed better performance compared to the 'Greedy-reversion', This result does not surprise us as a more balanced solution is expected to yield better results for the minimum total flow-time problem. In terms of budget utilization, it was expected that this greedy, budget oriented method will utilize as mush budget as possible.

An additional direction for future work is to develop algorithms for the minimum makespan problem with a guaranteed approximation ratio. While tuning existing approximation-algorithm for the classical one-shot problem seems to be a promising direction, the presence of transition-costs and job-extension penalties give rise to new challenges and considerations. Finally, it would be interesting to consider different objective functions, or different scheduling environments, for example, jobs with deadlines or precedence constraints, as well as unrelated or restricted machines.

References

[1] G. Ausiello, B. Escoffier, J. Monnot, and V. Th. Paschos. Reoptimization of minimum and maximum traveling salesmans tours. *J. of Discrete Algorithms* 7(4):453–463, 2009.

[2] G. Baram and T. Tamir, Reoptimization of the Minimum Total Flow-Time Scheduling Problem. *Sustainable Computing, Informatics and Systems*. vol. 4(4):241–251, 2014.

[3] J. Berlinskaa and M. Drozdowskib. Scheduling divisible MapReduce computations. In *Journal of Parallel and Distributed Computing*. vol.71(3):450-459, 2011.

[4] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems* 2:39–59 ,1984.

[5] H. J. Bockenhauer, L. Forlizzi, J. Hromkovic, J. Kneis, J. Kupke, G. Proietti, and P. Widmayer. On the approximability of TSP on local modifications of optimally solved instances. *Algorithmic Operations Research* 2(2), 2007.

[6] J. L. Bruno, E.G Coffman, and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 17:382–387, 1974.

[7] C. Clark, K. Fraser, S.Hand, J.G. Hansen, E. Jul, C. Limpach, I. Pratt, A. Warfield. Live migration of virtual machines. The 2nd *Symp. on Networked Systems Design and Implementation* (NSDI). 2005.

[8] R.W. Conway, W.L. Maxwell, and L.W. Miller. Theory of Scheduling. *AddisonWesley*, 1967.

[9] A. E. Eiben and J. E. Smith. Introduction to Evolutionary Computing. *Springer*, 2007

[10] D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms, Chapter 8. In *CRC Handbook of Algorithms and Theory of Computation*, ed. M. J. Atallah, 1999.

[11] B. Escoffier, M. Milanic, and V. Th. Paschos. Simple and fast reoptimizations for the Steiner tree problem. *DIMACS Technical Report* 2007-01.

[12] M. R. Garey and D.S. Johnson. *Computers and Intractability: a guide to the theory of NP-completeness*. W. H. Freeman and Co., New York, 1979.

[13] R. L. Graham, E.L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, Optimization and approximation in deterministic sequencing and scheduling: a survey, *Ann. Discrete Math.* vol. 5:287–326, 1979.

[14] S. Hacking and B. Hudzia. Improving the live migration process of large enterprise applications. The 3rd *international workshop on Virtualization technologies in distributed computing* (VTDC), 2009.

[15] D. S. Hochbaum and D. B. Shmoys. Using dual approximation algorithms for scheduling problems: Practical and theoretical results. *Journal of the ACM*, 34(1):144–162, 1987.

[16] W. Horn. Minimizing average flow-time with parallel machines. *Operations Research*, 21:846–847, 1973.

[17] Harold W. Kuhn. The Hungarian Method for the assignment problem, *Naval Research Logistics Quarterly*, vol. 2: 83–97, 1955.

[18] Parallels Virtuozzo *http://www.parallels.com/products/pvc*.

[19] L. M. Schmitt. Theory of Genetic Algorithms, *Theoretical Computer Science* vol. 259:1-61, 2001.

[20] H. Shachnai, G. Tamir, and T. Tamir, Minimal cost reconfiguration of data placement in storage area network. *Theoretical Computer Science*. vol. 460:42–53, 2012.

[21] H. Shachnai, G. Tamir, and T. Tamir. A theory and algorithms for combinatorial reoptimization. *In Proc. of 10th LATIN*, 2012.

[22] W.E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*. vol. 3:59–66, 1956.

[23] M. Thorup, and D.R. Karger. Dynamic graph algorithms with applications. *In Proc. of 7th SWAT*, 2000.

[24] Xen Project, *http://www.xenproject.org/*.