Efficient Adaptive Implementation of the Serial Schedule Generation Scheme using Preprocessing and Bloom Filters

Daniel Karapetyan[1] and Alexei Vernitski[2]

[1] Institute for Analytics and Data Science, University of Essex, Essex, UK
daniel.karapetyan@gmail.com
[2] Department of Mathematical Sciences, University of Essex, Essex, UK
asvern@essex.ac.uk

Abstract. The majority of scheduling metaheuristics use indirect representation of solutions as a way to efficiently explore the search space. Thus, a crucial part of such metaheuristics is a "schedule generation scheme" – procedure translating the indirect solution representation into a schedule. Schedule generation scheme is used every time a new candidate solution needs to be evaluated. Being relatively slow, it eats up most of the running time of the metaheuristic and, thus, its speed plays significant role in performance of the metaheuristic. Despite its importance, little attention has been paid in the literature to efficient implementation of schedule generation schemes. We give detailed description of serial schedule generation scheme, including new improvements, and propose a new approach for speeding it up, by using Bloom filters. The results are further strengthened by automated control of parameters. Finally, we employ online algorithm selection to dynamically choose which of the two implementations to use. This hybrid approach significantly outperforms conventional implementation on a wide range of instances.

Keywords: resource-constrained project scheduling problem, serial schedule generation scheme, Bloom filters, online algorithm selection

1 Introduction

Resource Constrained Project Scheduling Problem (RCPSP) is to schedule a set of jobs J subject to precedence relationships and resource constraints. RCPSP is a powerful model generalising several classic scheduling problems such as job shop scheduling, flow shop scheduling and parallel machine scheduling.

In RCPSP, we are given a set of resources R and their capacities c_r, $r \in R$. In each time slot, c_r units of resource r are available and can be shared between jobs. Each job $j \in J$ has a prescribed consumption $v_{j,r}$ of each resource $r \in R$. We are also given the duration d_j of a job $j \in J$. A job consumes $v_{j,r}$ units of resource r in every time slot that it occupies. Once started, a job cannot be interrupted (no preemption is allowed). Finally, each resource is assigned a set $\mathit{pred}_j \subset J$ of jobs that need to be completed before j can start.

There exist multiple extensions of RCPSP. In the multi-mode extension, each job can be executed in one of several modes, and then resource consumption and duration depend on the selected mode. In some applications, resource availability may vary with time. There could be set-up times associated with certain jobs. In multi-project extension, several projects run in parallel sharing some but not all resources. In this paper we focus on the basic version of RCPSP, however some of our results can be easily generalised to many of its extensions and variations.

Most of the real-world scheduling problems, including RCPSP, are NP-hard, and hence only problems of small size can be solved to optimality, whereas for larger problems (meta)heuristics are commonly used. Metaheuristics usually search in the space of feasible solutions; with a highly constrained problem such as RCPSP, browsing the space of feasible solutions is hard. Indeed, if a schedule is represented as a vector of job start times, then changing the start time of a single job is likely to cause constraint violations. Usual approach is to use indirect solution representation that could be conveniently handled by the metaheuristic but could also be efficiently translated into the direct representation.

Two translation procedures widely used in scheduling are *serial schedule generation scheme* (SSGS) and *parallel schedule generation scheme* [9]. Some studies conclude that SSGS gives better performance [7], while others suggest to employ both procedures within a metaheuristic [10]. Our research focuses on SSGS.

With SSGS, the indirect solution representation is a permutation π of jobs. The metaheuristic handles candidate solutions in indirect (permutation) form. Every time a candidate solution needs to be evaluated, SSGS is executed to translate the solution into a schedule (t_j, $j \in J$), and only then the objective value can be computed, see Figure 1.

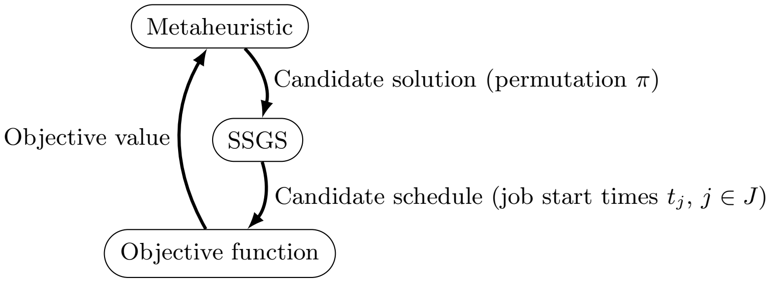
Metaheuristic
Candidate solution (permutation π)
Objective value
SSGS
Candidate schedule (job start times t_j, $j \in J$)
Objective function

Fig. 1: Classic architecture of a scheduling metaheuristic. To obtain objective value of a candidate solution, metaheuristic uses SSGS to translate the candidate solution into a candidate schedule, which is then used by objective function.

SSGS is a simple procedure that iterates through J in the order given by π, and schedules one job at a time, choosing the earliest feasible slot for each job. The only requirement for π is to respect the precedence relations; otherwise SSGS produces a feasible schedule for any permutation of jobs. The pseudo-

Algorithm 1: Serial Schedule Generation Scheme (SSGS). Here T is the upper bound on the makespan.

```
input  : Solution $\pi$ in permutation form, respecting precedence relations
output : Schedule $t_j$, $j \in J$
$A_{t,r} \leftarrow c_r$ for every $t = 1, 2, \ldots, T$ and $r \in R$;
for $i = 1, 2, \ldots, |J|$ do
    $j \leftarrow \pi(i)$;
    $t^0 \leftarrow \max_{j' \in pred_j} t_{j'} + d_{j'}$;
    $t_j \leftarrow find(j, t^0, A)$ (see Algorithm 2);
    $update(j, t_j, A)$ (see Algorithm 3);
return $t_j$, $j \in J$;
```

Algorithm 2: Conventional implementation of *find*(j, t^0, A) – a function to find the earliest feasible slot for job j.

```
input  : Job $j \in J$ to be scheduled
input  : Earliest start time $t^0$ as per precedence relations
input  : Current availability $A$ of resources
output : Earliest feasible start time for job $j$
$t_j \leftarrow t^0$;
$t \leftarrow t_j$;
while $t < t_j + d_j$ do
    if $A_{t,r} \geq v_{j,r}$ for every $r \in R$ then
        $t \leftarrow t + 1$;
    else
        $t_j \leftarrow t + 1$;
        $t \leftarrow t_j$;
return $t_j$;
```

code of SSGS is given in Algorithm 1, and its two subroutines *find* and *update* in Algorithms 2 and 3.

Commonly, the objective of RCPSP is to find a schedule that minimises the makespan, i.e. the time required to complete all jobs; however other objective functions are also considered in the literature. We say that an objective function of a scheduling problem is *regular* if advancing the start time of a job cannot worsen the solution's objective value. Typical objective functions of RCPSP, including makespan, are regular. If the scheduling problem has a regular objective function, then SSGS guarantees to produce *active* solutions, i.e. solutions that cannot be improved by changing t_j for a single $j \in J$. Moreover, it was shown [8] that for any active schedule S there exists a permutation π for which SSGS will generate S. Since any optimal solution S is active, searching in the space of feasible permutations π is sufficient to solve the problem. This is an important property of SSGS; the parallel schedule generation scheme, mentioned above,

Algorithm 3: Procedure $update(j, t_j, A)$ to update resource availability A after scheduling job j at time t_j.

```
input : Job $j$ and its start time $t_j$
input : Resource availability $A$
for $t \leftarrow t_j, t_j + 1, \ldots, t_j + d_j - 1$ do
    $A_{j,r} \leftarrow A_{j,r} - v_{j,r}$ for every $r \in R$;
```

does not provide this guarantee [8] and, hence, may not in some circumstances allow a metaheuristic finding optimal or near-optimal solutions.

The runtime of a metaheuristic is divided between its search control mechanism that modifies solutions and makes decisions such as accepting or rejecting new solutions, and SSGS. While SSGS is a polynomial algorithm, in practice it eats up the majority of the metaheuristic runtime (over 98% as reported in [1]). In other words, by improving the speed of SSGS twofold, one will (almost) double the number of iterations a metaheuristic performs within the same time budget, and this increase in the number of iterations is likely to have a major effect on the quality of obtained solutions.

In our opinion, not enough attention was paid to SSGS – a crucial component of many scheduling algorithms, and this study is to close this gap. In this paper we discuss approaches to speed up the conventional implementation of SSGS. Main contributions of our paper are:

- A detailed description of SSGS including old and new speed-ups (Section 2).
- New implementation of SSGS employing Bloom filters for quick testing of resource availability (Section 3).
- A hybrid control mechanism that employs intelligent learning to dynamically select the best performing SSGS implementation (Section 4).

Empirical evaluation in Section 5 confirms that both of our implementations of SSGS perform significantly better than the conventional SSGS, and the hybrid control mechanism is capable of correctly choosing the best implementation while generating only negligible overheads.

2 SSGS implementation details

Before we proceed to introducing our main new contributions in Sections 3 and 4, we describe what state-of-the-art implementation of SSGS we use, including some previously unpublished improvements.

2.1 Initialisation of A

The initialisation of A in line 1 of Algorithm 1 iterates through T slots, where T is the upper bound on the makespan. It was noted in [1] that instead of initialising A at every execution of SSGS, one can reuse this data structure between the

Algorithm 4: Enhanced implementation of *find*(j, t^0, A)

```
input  : Job $j \in J$ to be scheduled
input  : Earliest start time $t^0$ as per precedence relations
input  : Current availability $A$ of resources
output : Earliest feasible start time for job $j$
$t_j \leftarrow t^0$;
$t \leftarrow t_j + d_j - 1$;
$t^{\text{test}} \leftarrow t_j$;
while $t \geq t^{test}$ do
    if $A_{t,r} \geq V_{j,r}$ for every $r \in R$ then
        $t \leftarrow t - 1$;
    else
        $t^{\text{test}} \leftarrow t_j + d_j$;
        $t_j \leftarrow t + 1$;
        $t \leftarrow t_j + d_j - 1$;
return $t_j$;
```

executions. To correctly initialise A, at the end of SSGS we restore $A_{t,r}$ for each $r \in R$ and each slot where some job was scheduled: $A_{t,r} \leftarrow c_r$ for $r \in R$ and $t = 1, 2, \ldots, M$, where M is the makespan of the solution. Since $M \leq T$ and usually $M \ll T$, this notably improves the performance of SSGS [1].

2.2 Efficient search of the earliest feasible slot for a job

The function *find*(j, t^0, I, A) finds the earliest slot feasible for scheduling job j. Its conventional implementation (Algorithm 2) takes $O(T|R|)$ time, where T is the upper bound of the time horizon. Our enhanced implementation of *find* (Algorithm 4), first proposed in [1], has the same worst case complexity but is more efficient in practice. It is inspired by the Knuth-Morris-Pratt substring search algorithm. Let t_j be the assumed starting time of job J. To verify if it is feasible, we need to test sufficiency of resources in slots $t_j, t_j + 1, \ldots, t_j + d_j - 1$. Unlike the conventional implementation, our enhanced version tests these slots in the reversed order. The order makes no difference if the slot is feasible, but otherwise testing in reversed order allows us to skip some slots; in particular, if slot t is found to have insufficient resources then we know that feasible t_j is at least $t + 1$.

A further speed up, which was not discussed in the literature before, is to avoid re-testing of slots with sufficient resources. Consider the point when we find that the resources in slot t are insufficient. By that time we know that the resources in $t+1, t+2, \ldots, t_j + d_j - 1$ are sufficient. Our heuristic is to remember that the earliest slot t^{test} to be tested in future iterations is $t_j + d_j$.

2.3 Preprocessing and Automated Parameter Control

We observe that in many applications, jobs are likely to require only a subset of resources. For example, in construction works, to dig a hole one does not need cranes or electricians, hence the 'dig a hole' job will not consume those resources. To exploit this observation, we pre-compute vector R_j of resources used by job j, and then iterate only through resources in R_j when testing resource sufficiency in *find* (Algorithm 4, line 5) and updating resource availability in *update* (Algorithm 3, line 2). Despite the simplicity of this idea, we are not aware of anyone using or mentioning it before.

We further observe that some jobs may consume only one resource. By creating specialised implementations of *find* and *update*, we can reduce the depth of nested loops. While this makes no difference from the theoretical point of view, in practice this leads to considerable improvement of performance. Correct implementations of *find* and *update* are identified during preprocessing and do not cause overheads during executions of SSGS.

Having individual vectors R_j of consumed resources for each job, we can also intelligently learn the order in which resource availability is tested (Algorithm 4, line 5). By doing this, we are aiming at minimising the expected number of iterations within the resource availability test. For example, if resource r is scarce and job j requires significant amount of r, then we are likely to place r at the beginning of R_j. More formally, we sort R_j in descending order of probability that the resource is found to be insufficient during the search. This probability is obtained empirically by a special implementation of SSGS which we call $\texttt{SSGS}^{\texttt{data}}$. $\texttt{SSGS}^{\texttt{data}}$ is used once to count how many times each resource turned out to be insufficient during scheduling of a job. (To avoid bias, $\texttt{SSGS}^{\texttt{data}}$ tests every resource in R_j even if the test could be terminated early.) After a single execution of $\texttt{SSGS}^{\texttt{data}}$, vectors R_j are optimised, and in further executions default implementation of SSGS is used.

One may notice that the data collected in the first execution of SSGS may get outdated after some time; this problem in addressed in Section 4.

Ordering of R_j is likely to be particularly effective on instances with asymmetric use of resources, i.e. on real instances. Nevertheless, we observed improvement of runtime even on pseudo-random instances as reported in Section 5.

3 SSGS Implementation using Bloom filters

Performance bottleneck of an algorithm is usually its innermost loop. Observe that the innermost loop of the *find* function is the test of resource sufficiency in a slot, see Algorithm 4, line 5. In this section we try to reduce average runtime of this test from $O(|R|)$ to $O(1)$ time. For this, we propose a novel way of using a data structure known as Bloom filter.

Bloom filters were introduced in [2] as a way of optimising dictionary lookups, and found many applications in computer science and electronic system engineering [3,13]. Bloom filters usually utilise pseudo-random hash functions to encode

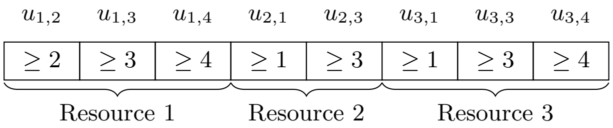

Fig. 2: Example of a Bloom filter structure for a problem with 3 resources, each having capacity 4.

data, but in some applications [6] non-hash-based approaches are used. In our paper, we also use a non-hash-based approach, and to our knowledge, our paper is the first in which the structure of Bloom filters is chosen dynamically according to the statistical properties of the data, with the purpose of improving the speed of an optimisation algorithm.

In general, Bloom filters can be defined as a way of using data, and they are characterised by two aspects: first, all data is represented by short binary arrays of a fixed length (perhaps with a loss of accuracy); second, the process of querying data involves only bitwise comparison of binary arrays (which makes querying data very fast).

We represent both each job's resource consumption and resource availability at each time slot, by binary arrays of a fixed length; we call these binary arrays Bloom filters. Our Bloom filters will consist of bits which we call *resource level bits*. Each resource bit, denoted by $u_{r,k}$, $r \in R$, $k \in \{1, 2, \dots, c_r\}$, means "$k$ units of resource r" (see details below). Let U be the set of all possible resource bits. A *Bloom filter structure* is an ordered subset $L \subseteq U$, see Figure 2 for an example. Suppose that a certain Bloom filter structure L is fixed. Then we can introduce $B^L(j)$, the Bloom filter of job j, and $B^L(t)$, the Bloom filter of time slot t, for each j and t. Each $B^L(j)$ and $B^L(t)$ consists of $|L|$ bits defined as follows: if $u_{r,k}$ is the ith element of L then

$$B^L(j)_i = \begin{cases} 1 & \text{if } v_{j,r} \geq k, \\ 0 & \text{otherwise,} \end{cases} \qquad \text{and} \qquad B^L(t)_i = \begin{cases} 1 & \text{if } A_{t,r} \geq k, \\ 0 & \text{otherwise.} \end{cases}$$

To query if a job j can be scheduled in a time slot t, we compare Bloom filters $B^L(j)$ and $B^L(t)$ bitwise; then one of three situations is possible, as the following examples show. Consider a job j and three slots, t, t' and t'', with the following resource consumption/availabilities, and Bloom filter structure as in Figure 2:

$$\begin{array}{rrrr} v_{j,1} = 3 & v_{j,2} = 2 & v_{j,3} = 0 & B^L(j) = (110\,10\,000) \\[1ex] A_{t,1} = 2 & A_{t,2} = 3 & A_{t,3} = 4 & B^L(t) = (100\,11\,111) \\ A_{t',1} = 3 & A_{t',2} = 1 & A_{t',3} = 4 & B^L(t') = (110\,10\,111) \\ A_{t'',1} = 3 & A_{t'',2} = 2 & A_{t'',3} = 2 & B^L(t'') = (110\,10\,100) \end{array}$$

For two bit arrays of the same length, let notation '$\leq$' mean bitwise less or equal. By observing that $B^L(j) \not\leq B^L(t)$, we conclude that resources in slot t

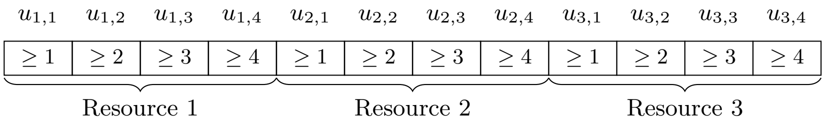

Fig. 3: Example of a Bloom filter structure for a problem with 3 resources, each having capacity 4, with $L = U$.

are insufficient for j; this conclusion is guaranteed to be correct. By observing that $B^L(j) \leq B^L(t')$, we conclude tentatively that resources in slot t' may be sufficient for j; however, further verification of the complete data related to j and t' (that is, $v_{j,\cdot}$ and $A_{t',\cdot}$) is required to get a precise answer; one can see that $v_{j,2} \geq A_{t',2}$, hence this is what is called a *false positive*. Finally, we observe that $B^L(j) \leq B^L(t'')$, and a further test (comparing $v_{j,\cdot}$ and $A_{t'',\cdot}$) confirms that resources in t'' are indeed sufficient for j.

Values of $B^L(j)$, $j \in J$, are pre-computed when L is constructed, and $B^L(t)$, $t = 1, 2, \ldots, T$, are maintained by the algorithm.

3.1 Optimisation of Bloom Filter Structure

The length $|L|$ of a Bloom filter is limited to reduce space requirements and, more importantly for our application, speed up Bloom filter tests. Note that if $|L|$ is small (such as 32 or 64 bits) then we can exploit efficient bitwise operators implemented by all modern CPUs; then each Bloom filter test takes only one CPU operation. We set $|L| = 32$ in our implementation. While obeying this constraint, we aim at minimising the number of false positives, because false positives slow down the implementation.

Our L building algorithm is as follows:

1. Start with $L = U$, such as in Figure 3.
2. If $|L|$ is within the prescribed limit, stop.
3. Otherwise select $u_{r,k} \in L$ that is least important and delete it. Go to step 2.

By 'least important' we mean that the deletion of it is expected to have minimal impact of the expected number of false positives. Let $L = (\ldots, u_{r,q}, u_{r,k}, u_{r,m}, \ldots)$. Consider a job j such that $k \leq v_{j,r} < m$ and a slot t such that $q \leq A_{t,r} < k$. With L as defined above, Bloom filters correctly identify that resources in slot t are insufficient for job j: $B^L(j) \not\leq B^L(t)$. However, without the resource level bit $u_{r,k}$ we get a false positive: $B^L(j) \leq B^L(t)$. Thus, the probability of false positives caused by deleting $u_{r,k}$ from L in is as follows:

$$\left(\sum_{k=i}^{m-1} D_k^r\right) \cdot \left(\sum_{k=q}^{i-1} E_k^r\right) \ ,$$

where D_k^r is the probability that a randomly chosen job needs exactly k units of resource r, and E_k^r is the probability that a certain slot, when we examine it for scheduling a job, has exactly k units of resource r available. The probability distribution D^r is produced from the RCPSP instance data during preprocessing.[3] The probability distribution E^r is obtained empirically during the run of $\texttt{SSGS}^{\texttt{data}}$ (see Section 2.2); each time resource sufficiency is tested within $\texttt{SSGS}^{\texttt{data}}$, its availability is recorded.

3.2 Additional Speed-ups

While positive result of a Bloom filter test generally requires further verification using full data, in some circumstances its correctness can be guaranteed. In particular, if for some $r \in R$ and $j \in J$ we have $u_{r,k} \in L$ and $v_{j,r} = k$, then the Bloom filter result, whether positive or negative, does not require verification.

Another observation is that updating $B^L(t)$ in *update* can be done in $O(|R_j|)$ operations instead of $O(|R|)$ operations. Indeed, instead of computing $B^L(t)$ from scratch, we can exploit our structure of Bloom filters. We update each bit related to resources $r \in R_j$, but we keep intact other bits. With some trivial pre-processing, this requires only $O(|R_j|)$ CPU operations.

We also note that if $|R_j| = 1$, i.e. job j uses only one resource, then Bloom filters will not speed up the *find* function for that job and, hence, in such cases we use the standard *find* function specialised for one resource (see Section 2.2).

4 Hybrid Control Mechanism

So far we have proposed two improved implementations of SSGS: one using Bloom filters (which we denote $\texttt{SSGS}^{\texttt{BF}}$), and the other one not using Bloom filters (which we denote $\texttt{SSGS}^{\texttt{NBF}}$). While it may look like $\texttt{SSGS}^{\texttt{BF}}$ should always be superior to $\texttt{SSGS}^{\texttt{NBF}}$, in practice $\texttt{SSGS}^{\texttt{NBF}}$ is often faster. Indeed, Bloom filters usually speed up the *find* function, but they also slow down *update*, as in $\texttt{SSGS}^{\texttt{BF}}$ we need to update not only the values $A_{t,r}$ but also the Bloom filters $B^L(t)$ encoding resource availability. If, for example, the RCPSP instance has tight precedence relations and loose resource constraints then *find* may take only a few iterations, and then the gain in the speed of *find* may be less than the loss of speed of *update*. In such cases $\texttt{SSGS}^{\texttt{BF}}$ is likely to be slower then $\texttt{SSGS}^{\texttt{NBF}}$.

In short, either of the two SSGS implementations can be superior in certain circumstances, and which one is faster mostly depends on the problem instance. In this section we discuss how to adaptively select the best SSGS implementation. Automated algorithm selection is commonly used in areas such as sorting, where multiple algorithms exist. A typical approach is then to extract easy to compute input data features and then apply off-line learning to develop a predictor of which algorithm is likely to perform best, see e.g. [5]. With sorting, this seems to be the most appropriate approach as the input data may vary significantly between executions of the algorithm. Our case is different in that the most

[3] In multi-mode extension of RCPSP, this distribution depends on selected modes and hence needs to be obtained empirically, similarly to how we obtain E^r.

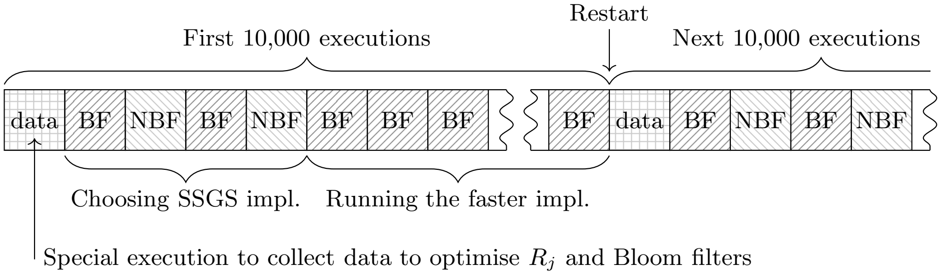

Fig. 4: Stages of the Hybrid control mechanism. Each square shows one execution of SSGS, and the text inside describes which implementation of SSGS is used. $\mathtt{SSGS}^{\mathtt{data}}$ is always used in the first execution. Further few executions (at most 100) alternate between $\mathtt{SSGS}^{\mathtt{BF}}$ and $\mathtt{SSGS}^{\mathtt{NBF}}$, with each execution being timed. Once sign test shows significant difference between the $\mathtt{SSGS}^{\mathtt{BF}}$ and $\mathtt{SSGS}^{\mathtt{NBF}}$ implementations, the faster one is used for the rest of executions. After 10,000 executions, previously collected data is erased and adaptation starts from scratch.

crucial input data (the RCPSP instance) does not change between executions of SSGS. Thus, during the first few executions of SSGS, we can test how each implementation performs, and then select the faster one. This is a simple yet effective control mechanism which we call Hybrid.

Hybrid is entirely transparent for the metaheuristic; the metaheuristic simply calls SSGS whenever it needs to evaluate a candidate solution and/or generate a schedule. The Hybrid control mechanism is then intelligently deciding each time which implementation of SSGS to use based on information learnt during previous runs. An example of how Hybrid performs is illustrated in Figure 4. In the first execution, it uses $\mathtt{SSGS}^{\mathtt{data}}$ to collect data required for both $\mathtt{SSGS}^{\mathtt{BF}}$ and $\mathtt{SSGS}^{\mathtt{NBF}}$. For the next few executions, it alternates between $\mathtt{SSGS}^{\mathtt{BF}}$ and $\mathtt{SSGS}^{\mathtt{NBF}}$, measuring the time each of them takes. During this stage, Hybrid counts how many times $\mathtt{SSGS}^{\mathtt{BF}}$ was faster than the next execution of $\mathtt{SSGS}^{\mathtt{NBF}}$. Then we use the sign test [4] to compare the implementations. If the difference is significant (we use a 5% significance level for the sign test) then we stop alternating the implementations and in future use only the faster one. Otherwise we continue alternating the implementations, but for at most 100 executions. (Without such a limitation, there is a danger that the alternation will never stop – if the implementations perform similarly; since there are overheads associated with the alternation and time measurement, it is better to pick one of the implementations and use it in future executions.)

Our decision to use the sign test is based on two considerations: first, it is very fast, and second, it works for distributions which are not normal. This makes our approach different from [12] where the distributions of runtimes are assumed to be normal. (Note that in our experiments we observed that the distribution of running times of an SSGS implementation resembles Poisson distribution.)

As pointed out in this and previous sections, optimal choices of parameters of the SSGS implementations mostly depend on the RCPSP instance – which does not change throughout the metaheuristic run; however solution π also affects the performance. It should be noted though that metaheuristics usually apply only small changes to the solution at each iteration, and hence solution properties tend to change relatively slowly over time. Consequently, we assume that parameters chosen in one execution of SSGS are likely to remain efficient for some further executions. Thus, `Hybrid` 'restarts' every 10,000 executions, by which we mean that all the internal data collected by SSGS is erased, and learning starts from scratch, see Figure 4. This periodicity of restarts is a compromise between accuracy of choices and overheads, and it was shown to be practical in our experiments.

5 Empirical Evaluation

In this Section we evaluate the implementations of SSGS discussed above. To replicate conditions within a metaheuristic, we designed a simplified version of Simulated Annealing. In each iteration of our metaheuristic, current solution is modified by moving a randomly selected job into a new randomly selected position (within the feasible range). If the new solution is not worse than the previous one, it is accepted. Otherwise the new solution is accepted with 50% probability. Our metaheuristic performs 1,000,000 iterations before terminating.

We evaluate $\texttt{SSGS}^{\texttt{BF}}$, $\texttt{SSGS}^{\texttt{NBF}}$ and `Hybrid`. These implementations are compared to 'conventional' SSGS, denoted by $\texttt{SSGS}^{\texttt{conv}}$, which does not employ any preprocessing or Bloom filters and uses conventional implementation of *find* (Algorithm 2).

We found that instances in the standard RCPSP benchmark set PSPLIB [11] occupy a relatively small area of the feature space. For example, all the RCPSP instances in PSPLIB have exactly four resources, and the maximum job duration is always set to 10. Thus, we chose to use the PSPLIB instance generator, but with a wider range of settings. Note that for this study, we modified the PSPLIB instance generator by allowing jobs not to have any precedence relations. This was necessary to extend the range of network complexity parameter (to include instances with scarce precedence relations), and to speed up the generator, as the original implementation would not allow us to generate large instances within reasonable time.

All the experiments are conducted on a Windows Server 2012 machine based on Intel Xeon E5-2690 v4 2.60 GHz CPU. No concurrency is employed in any of the implementations or tests.

To see the effect of various instance features on SSGS performance, we select one feature at a time and plot average SSGS performance against the values of that feature. The rest of the features (or generator parameters) are then set as follows: number of jobs 120, number of resources 4, maximum duration of job 10, network complexity 1, resource factor 0.75, and resource strength 0.1.

These values correspond to some typical settings used in PSPLIB. For formal definitions of the parameters we refer to [11].

For each combination of the instance generator settings, we produce 50 instances using different random generator seed values, and in each of our experiments the metaheuristic solves each instance once. Then the runtime of SSGS is said to be the overall time spent on solving those 50 instances, over 50,000,000 (which is the number of SSGS executions). The metaheuristic overheads are relatively small and are ignored.

From the results reported in Figure 5 one can see that our implementations of SSGS are generally significantly faster than $\mathtt{SSGS}^{\mathtt{conv}}$, but performance of each implementation varies with the instance features. In some regions of the instance space $\mathtt{SSGS}^{\mathtt{BF}}$ outperforms $\mathtt{SSGS}^{\mathtt{NBF}}$, whereas in other regions $\mathtt{SSGS}^{\mathtt{NBF}}$ outperforms $\mathtt{SSGS}^{\mathtt{BF}}$. The difference in running times is significant, up to a factor of two in our experiments. At the same time, Hybrid is always close to the best of $\mathtt{SSGS}^{\mathtt{BF}}$ and $\mathtt{SSGS}^{\mathtt{NBF}}$, which shows efficiency of our algorithm selection approach. In fact, when $\mathtt{SSGS}^{\mathtt{BF}}$ and $\mathtt{SSGS}^{\mathtt{NBF}}$ perform similarly, Hybrid sometimes outperforms both; this behaviour is discussed below.

Another observation is that $\mathtt{SSGS}^{\mathtt{NBF}}$ is always faster than $\mathtt{SSGS}^{\mathtt{conv}}$ (always below the 100% mark) which is not surprising; indeed, $\mathtt{SSGS}^{\mathtt{NBF}}$ improves the performance of both *find* and *update*. In contrast, $\mathtt{SSGS}^{\mathtt{BF}}$ is sometimes slower than $\mathtt{SSGS}^{\mathtt{conv}}$; on some instances, the speed-up of the *find* function is overweighed by overheads in both *find* and *update*. Most important though is that Hybrid outperforms $\mathtt{SSGS}^{\mathtt{conv}}$ in each of our experiments by 8 to 68%, averaging at 43%. In other words, within a fixed time budget, an RCPSP metaheuristic employing Hybrid will be able to run around 1.8 times more iterations than if it used $\mathtt{SSGS}^{\mathtt{conv}}$.

To verify that Hybrid exhibits the adaptive behaviour and does not just stick to whichever implementation has been chosen initially, we recorded the implementation it used in every execution for several problems, see Figure 6. For this experiment, we produced three instances: first instance has standard parameters except Resource Strength is 0.2; second instance has standard parameters except Resource Factor is 0.45; third instance has standard parameters except Maximum Job Duration is 20. These parameter values were selected such that the two SSGS implementations would be competitive and, therefore, switching between them would be a reasonable strategy. One can see that the switches occur several times throughout the run of the metaheuristic, indicating that Hybrid adapts to the changes of solution. For comparison, we disabled the adaptiveness and measured the performance if only implementation chosen initially is used throughout all iterations; the results are shown on Figure 6. We conclude that Hybrid benefits from its adaptiveness.

6 Conclusions and Future Work

In this paper we discussed the crucial component of many scheduling metaheuristics, the serial schedule generation scheme (SSGS). SSGS eats up most

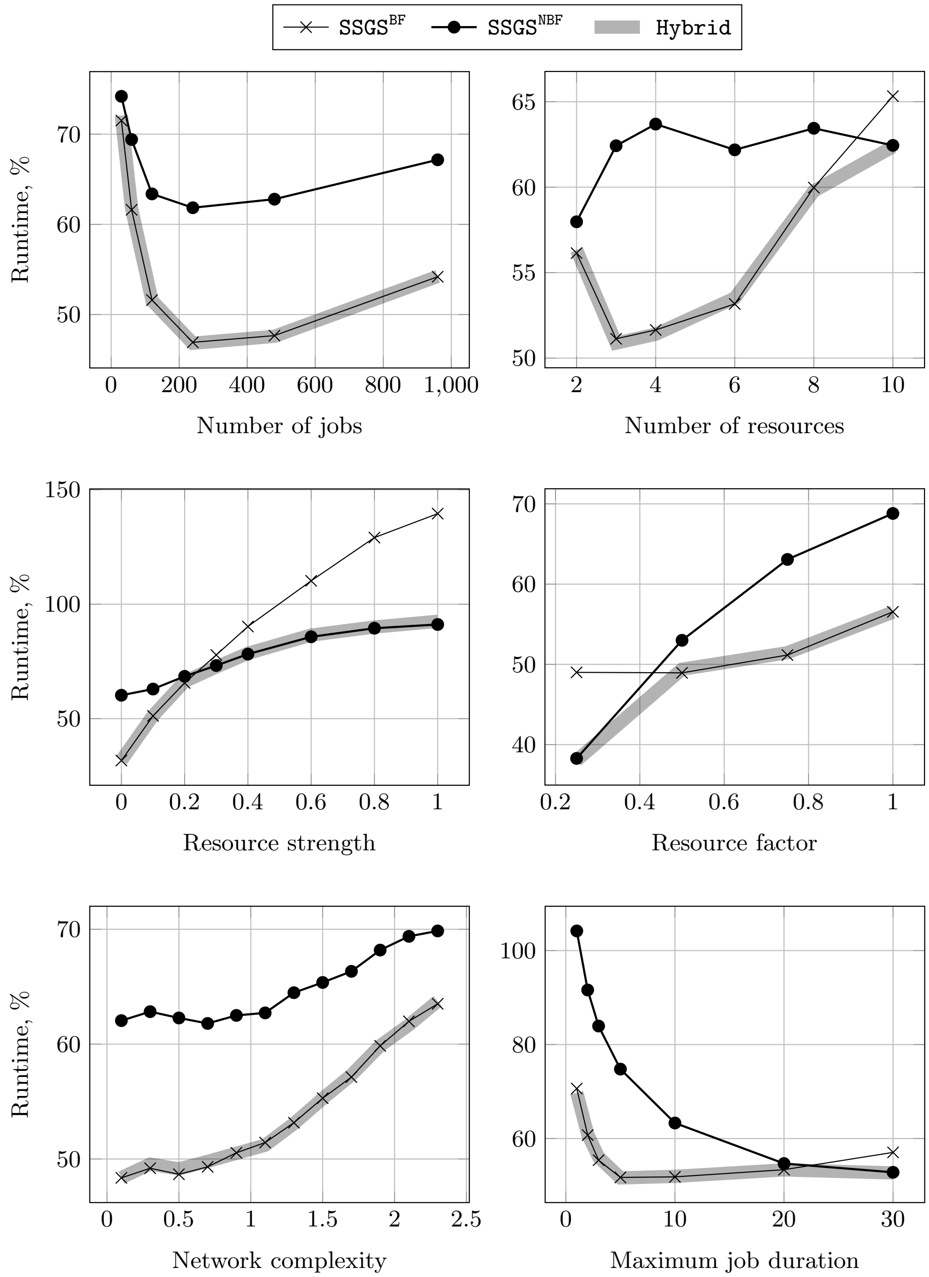

Fig. 5: These plots show how performance of the SSGS implementations depends on various instance features. Vertical axis gives the runtime of each implementation relative to $\texttt{SSGS}^{\text{conv}}$. ($\texttt{SSGS}^{\text{conv}}$ graph would be a horizontal line $y = 100\%$.)

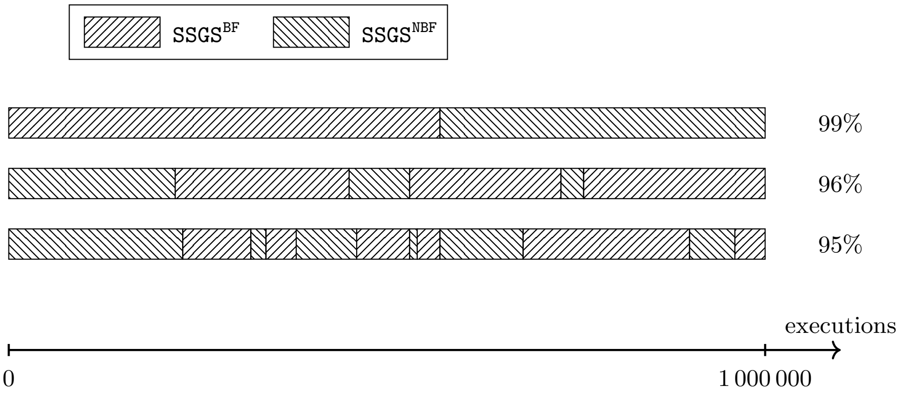

Fig. 6: This diagram shows how `Hybrid` switches between SSGS implementations while solving three problem instances. The number on the right shows the time spent by `Hybrid` compared with the time that would be needed if only the implementation chosen at the start would be used for all iterations.

of the runtime in a typical scheduling metaheuristic, therefore performance of SSGS is critical to the performance of the entire metaheuristic, and thus each speed-up of SSGS has significant impact. We described existing and some new speed-ups to SSGS, including preprocessing and automated parameter control. This implementation clearly outperformed the 'conventional' SSGS in our experiments. We further proposed a new implementation that uses Bloom filters, particularly efficient in certain regions of the instance space. To exploit strengths of both implementations, we proposed a hybrid control mechanism that learns the performance of each implementation and then adaptively chooses the SSGS version that is best for a particular problem instance and phase of the search. Experiments showed that this online algorithm selection mechanism is effective and makes `Hybrid` our clear choice. Note that `Hybrid` is entirely transparent for the metaheuristic which uses it as if it would be simple SSGS; all the learning and adaptation is hidden inside the implementation.

The idea behind online algorithm selection used in this project can be further developed by making the number of executions between restarts adaptable. To determine the point when the established relation between the SSGS implementations may have got outdated, we could treat the dynamics of the implementations' performance change as two random walks, and use the properties of these two random walks to predict when they may intersect.

All the implementations discussed in the paper, in C++, are available for downloading from `http://csee.essex.ac.uk/staff/dkarap/rcpsp-ssgs.zip`. The implementations are transparent and straightforward to use.

While we have only discussed SSGS for the simple RCPSP, our ideas can easily be applied in RCPSP extensions. We expect some of these ideas to work particularly well in multi-project RCPSP, where the overall number of resources is typically large but only a few of them are used by each job.

Acknowledgements We would like to thank Prof. Rainer Kolisch for providing us with a C++ implementation of the PSPLIB generator. It should be noted that, although the C++ implementation was developed to reproduce the original Pascal implementation, the exact equivalence cannot be guaranteed; also, in our experiments we used a modification of the provided C++ code.

References

1. Shahriar Asta, Daniel Karapetyan, Ahmed Kheiri, Ender Özcan, and Andrew J. Parkes. Combining Monte-Carlo and hyper-heuristic methods for the multi-mode resource-constrained multi-project scheduling problem. *Information Sciences*, 373:476–498, 2016.
2. Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
3. Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.
4. Louis Cohen and Michael Holliday. *Practical statistics for students: An introductory text*. Paul Chapman Publishing Ltd, 1996.
5. Haipeng Guo. *Algorithm selection for sorting and probabilistic inference: A machine learning-based approach*. PhD thesis, Kansas State University, 2003.
6. Gökçe Caylak Kayaturan and Alexei Vernitski. A way of eliminating errors when using Bloom filters for routing in computer networks. In *ICN 2016, The Fifteenth International Conference on Networks*, pages 52–57, 2016.
7. Jin-Lee Kim and Ralph D. Ellis Jr. Comparing schedule generation schemes in resource-constrained project scheduling using elitist genetic algorithm. *Journal of construction engineering and management*, 136(2):160–169, 2010.
8. Rainer Kolisch. Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation. *European Journal of Operational Research*, 90(2):320–333, 1996.
9. Rainer Kolisch and Sönke Hartmann. Heuristic algorithms for the resource-constrained project scheduling problem: Classification and computational analysis. In *Project scheduling*, pages 147–178. Springer, 1999.
10. Rainer Kolisch and Sonke Hartmann. Experimental investigation of heuristics for resource-constrained project scheduling: An update. *European Journal of Operational Research*, 174(1):23–37, 2006.
11. Rainer Kolisch and Arno Sprecher. PSPLIB – a project scheduling problem library: OR software – ORSEP operations research software exchange program. *European Journal of Operational Research*, 96(1):205–216, 1997.
12. Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. Online performance auditing: using hot optimizations without getting burned. *SIGPLAN Not.*, 41(6):239–251, 2006.
13. Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials*, 14(1):131–155, 2012.