Job Shop Scheduling
Seminar Report for Topics in Algorithms Seminar

Matúš Harvan
m.harvan@iu-bremen.de

Spring Semester 2006

Instructor: Stefano Carpin

Abstract

The randomized version of the first polynomial-time polylogarithmic approximation algorithm for job shop scheduling by David B. Shmoys has been implemented. It has been applied to the problem of coordinating multiple robots moving along fixed paths. The results has been evaluated and compared with another randomized algorithm, the Distributed Motion Planning algorithm by Stefano Carpin.

1 Introduction

The job shop scheduling problem is an interesting, NP-hard problem with many applications in industry. The randomized version of the first polynomial-time polylogarithmic approximation algorithm for job shop scheduling by David B. Shmoys has been implemented. It has been applied to the problem of coordinating multiple robots moving along fixed paths. The results has been evaluated and compared with another randomized algorithm, the Distributed Motion Planning algorithm by Stefano Carpin.

The rest of the report is organized as follows. Section 2 formally defines the job shop scheduling problem. Our particular application of the job shop scheduling problem, the coordination of multiple robots moving along fixed paths is described in section 3. Section 4 describes the Shmoys algorithm and implementation issues are discussed in section 5. Results from running the implemented algorithm and comparisons to the DMP algorithm are discussed in section 6. The report concludes with section 7

2 Job Shop Scheduling

This section presents a description and a formal definition of the job shop scheduling taken from [4]. In the job shop scheduling problem we are given m machines and n jobs. Each job consists of a sequence of operations. Each operation must be processed on a specified machine. A job may contain more operations to be processed on the same machine. Operations must be processed in order specified by the sequence. Furthermore, at any point of time each machine can process at most one operation and each job can be processed at most on one machine at any point of time (i.e. only one operation of a job can be processed at any point of time). The goal is to produce a schedule of jobs on machines minimizing the time it takes to process all jobs, C_{max}.

In more detail, a set of machines $\mathcal{M} = \{M_1, M_2, \ldots, M_m\}$, a set of jobs $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ and a set of operations $\mathcal{O} = \{O_{ij} | i = 1, \ldots, \mu_j, j = 1, \ldots, n\}$ are given. K_{ij} indexes machine that has to process operation O_{ij}. μ_j is the number of operations of job J_j and $\mu = \max_j \mu_j$. Operation O_{ij} is the i-th operation of job J_j and requires uninterrupted processing time of p_{ij} on machine M_k with $k = K_{ij}$. The completion time of the whole schedule is the time when the last operation of each job is processed. This is $C_{\max} = \max_{i,j} C_{ij}$. The goal is to find the minimum completion time, $C^*_{\max}$.

Even for very restrictive cases the problem is NP-hard. Additionally, it is also a difficult optimization problem with even small instances being difficult to solve exactly. For example, a 10 jobs, 10 machines and 100 operations problem instance posted in 1963 remained unsolved for 23 years and several instances with 15 jobs, 15 machines, 225 operations are too hard to be solved by known methods (by 1994) and have been posed as open problems[4]. Hence, it seems reasonable to give up the exact solution and look for randomized approximation algorithms.

Two lower bounds on the length of an optimal schedule can easily be identified. One is due to the fact that all operations of each job must be processed and for each job not more

than one operation can be processed at any time. Thus the schedule has to be at least as long as the longest job. Hence, $C^*_{\max} \geq \max_{J_j} \sum_i p_{ij}$. This bound is called the *maximum job length* and is denoted $P_{\max}$. The other bound comes from the fact that each machine has to process's all its operations. This means $C^*_{\max} \geq \max_{M_k} \sum_{K_{ij}=k} p_{ij}$. This bound is called the *maximum machine load* and is denoted $\Pi_{\max}$.

3 Multiple robots with fixed paths

The particular scenario to which the job shop scheduling approach has been applied is the coordination of multiple robots moving along fixed paths. There are multiple robots which should move along specified paths from an initial position to a destination. The paths of the robots may (and do) intersect. Hence, the passes of robots via crossings of paths have to be scheduled in such a way that collisions among robots are avoided. The goal is to find a suitable schedule avoiding collisions such that the time it takes for all the robots to get to their final destinations is minimized. An example of this problem is shown in figure 1.

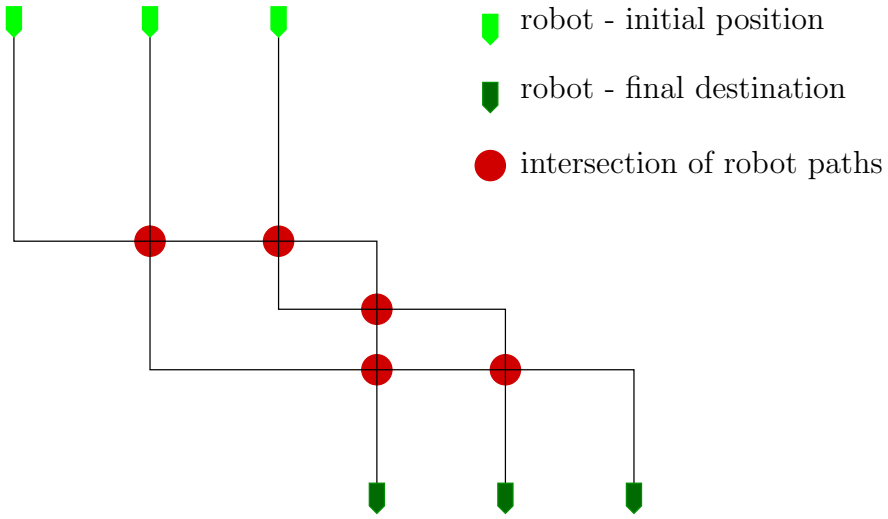

Figure 1: Multiple robots moving along fixed paths

An application of this problem would be scheduling and coordination of automated guided vehicles in harbors and airports or of automated vehicles in factories.

Clearly, the problem of scheduling multiple robots moving along fixed paths is an instance of the job shop scheduling problem. The intersections of robot paths represent machines. Robots represent jobs where a robot crossing an intersection is equivalent to an operation being processed on a machine. Therefore, a job shop scheduling algorithm will be used to solve this problem. Note that jobs do not necessarily have to be processed on every machine, i.e. robots do not necessarily cross every junction.

A similar problem is addresses in [3], but here only the start times of the robots can be modified. Once started, the robot has to follow a specified trajectory at specified velocities.

In our scenario, the velocity can also be changed, i.e. the robot can be stopped while on its path in order to wait for an intersection to become free.

4 Shmoys Algorithm

The algorithm presented in [4] is the first polynomial-time algorithm with a polylogarithmic bound on the approximation for solving the job shop scheduling problem. It will be referred to as the Shmoys algorithm. It comes in two versions, a randomized and a deterministic (derandomized) one. The randomized version will be described in more detail, closely following [4].

The algorithm first creates a schedule that may assign several operations to the same machine at the same time and later changes the schedule to obey the constraint of one operation per machine at any point of time. It works in three phases

1. Create an *oblivious* schedule, where each job starts at time 0 and runs continuously until all of it operations have been processed. It is possible that more than one operation is assigned to a single machine. The schedule is of length $P_{\max}$, the maximum job length.

2. Perturb the oblivious schedule by delaying the start of jobs by a random integral time chosen uniformly from the interval $[0, \Pi_{\max}]$. In this perturbed schedule there would be $O(\frac{\log(n\mu)}{\log\log(n\mu)}$ operations assigned to any machine at any time with high probability.

3. "Spread" the perturbed schedule so that at each point in time, all operations currently being processed have the same size. Afterwards, "flatten" the spread schedule to obey the constraint of at most one operation per machine at any point of time.

The spreading and flattening part will be explained in more detail. First the lengths of all operations are rounded up to the next power of two, so $p'_{ij} = 2^{\lceil \log_2 p_{ij} \rceil}$ The spreading of the schedule operates on blocks of the schedule. A *block* is an interval of the schedule where each operation that starts within the interval, is of length no more than the length of the interval. This does not imply that operations also end within the block, i.e. consider an operation as long as the block, but starting in the middle of the block. The schedule S from step 2 can be divided into $\lceil \frac{L}{p'_{\max}} \rceil$ blocks of size $p'_{\max} = \max_{ij} p'_{ij}$. On each block a recursive algorithm is run that spreads the block of size p into a sequence of fragments of total length $p \log p$. Operations in a *fragment* of length T are all of length T and start at the beginning. Clearly, if an operation of length p is scheduled to start in a block of length p, then that job is not scheduled on any other machine until after this block. Hence, such an operation can be scheduled at the end of the block after all smaller operations in the block have finished. To reschedule a block B of length $p'_{\max}$, the final fragment of length $p'_{\max}$ is constructed first. Afterwards, preceding fragments are constructed recursively. Each operation that begins in B and is of length $p'_{\max}$ is rescheduled into the final fragment and deleted from B. Afterwards, B has operations of length at most $\frac{p'_{\max}}{2}$ and so can be split into two

blocks, B_1, B_2 with half the original size. The algorithm then recurses on these smaller blocks. Total length of the fragments produced from a block of length L can be described by the recurrence relation $f(T) = 2f(\frac{T}{2}) + T; f(1) = 1$. Hence, $f(T) = \Theta(T \log T)$, so each block of length $p'_{\max}$ grows to $p'_{\max} \log p'_{\max}$. An example of spreading is presented in figure 2. After running the spreading algorithm on schedule S, a schedule S' with following

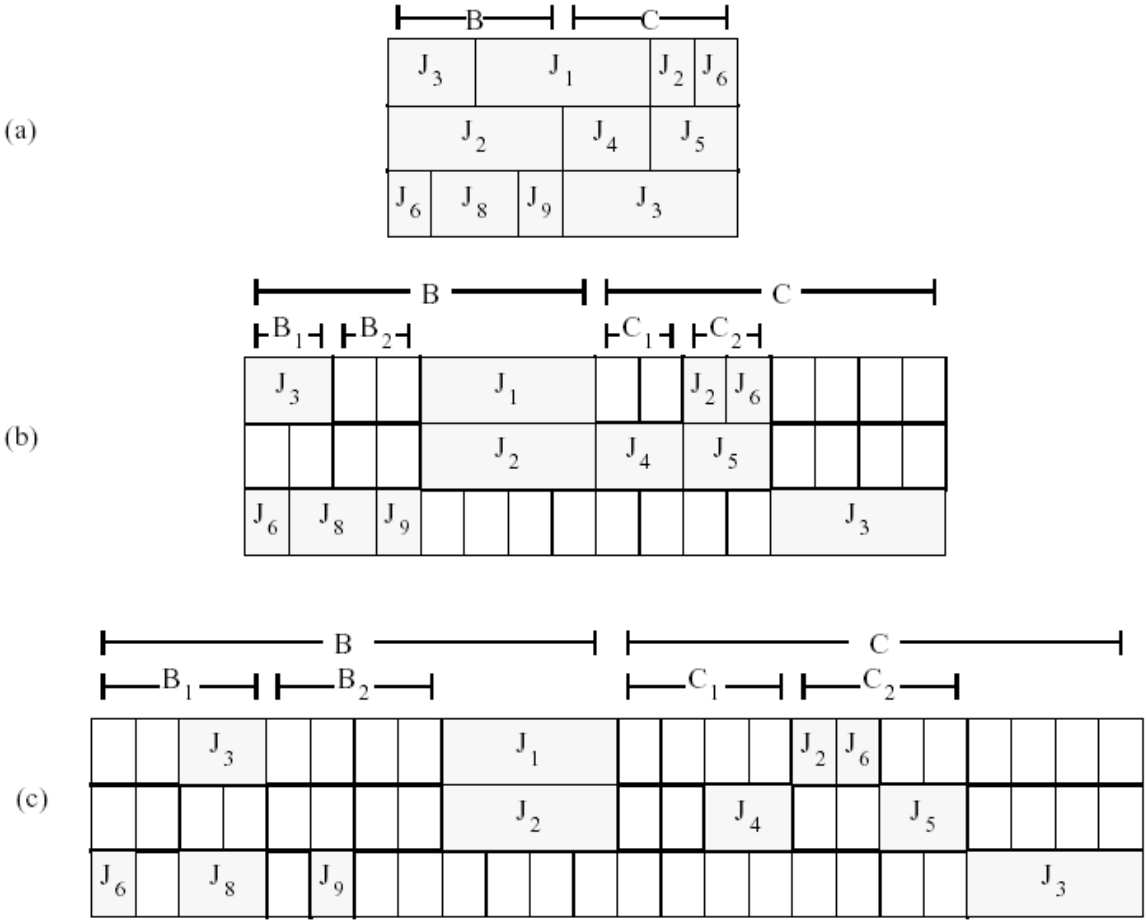

Figure 2: (a) The oblivious schedule of length 8. $p'_{\max} = 4$. (b)First spreading iterations. All jobs of length 4 moved to final fragments. Recursion on B_1, B_2 with $p'_{\max} = 2$. (c)Final schedule, with length $8 \log_2 8 = 24$. Figure taken from[4].

properties is obtained:

1. Operations scheduled at the same time are of the same length and any two operations either start at the same time or do not overlap.

   Proof. Satisfied by each pair of operations on the same machine by definition of spreading and on different machines because the division of time into fragments is the same for all machines.

2. If S has at most c jobs scheduled on one machine at the same time, then the same limit holds also for the new schedule S'.

   Proof. Operations of length T are scheduled at the same time on the same machine in S' if they started in the same block of size T on that machine in S. As they all must have been scheduled during the last unit of time in that block, there can be at most c of them.

3. Each job is scheduled on at most one machine at any point of time.

   Proof. If a job is scheduled in S' on two machines simultaneously, then in S' it must have been scheduled to start two operations of length T in the same block of length T on two different machines, i.e. it was scheduled in S' on two different machines during the last unit of time of that block. This would violate the properties of S'.

4. The i-th operation of a job is not scheduled until the first $i-1$ operations are completed.

   Proof. If two operations of a job are in different blocks of size $p'_{\max}$ in S then for sure they would be rescheduled in the correct order. For the case of schedule being produced from one block, if an operation is rescheduled to the final fragment, then it is the last operation for its job in given block (by definition of blocks). Therefore, order of operations is respected in S' also within blocks.

The spread schedule S' can trivially be flattened into a schedule obeying the constraint of only one operation per machine at any point of time. If c operations of length T start at the same, they can simply be executed one after the other, in total time cT.

The above described randomized algorithm has an approximation bound of $O(\frac{log^2(m\mu)}{loglog(m\mu)}C^*_{max})$ with high probability. It can be turned into a deterministic version by replacing the random assignment of initial delays with deterministic values as described in [4]. The deterministic version has a slightly worse bound of $O(log^2(m\mu)C^*_{max})$. Detailed proofs of the bounds can be found in [4]. The advantage is that this bound holds in any case, not just with high probability. Furthermore, the derandomization increases runtime of the algorithm.

5 Implementation

The randomized Shmoys algorithm has been implemented C. The `GLib` library[1] has been used for linked lists (`struct GList`) and `openssl`[2] for obtaining random values. Source code of the implementation is in `schedule.c`.

The input format is expected to be the same as for glpk and is specified in `input/modprob.mod`. Basically, it contains the number of machine, number of jobs, assignment of operations to machines and the processing time of jobs needed on each machine. Output is one line

containing the length of the schedule found, runtime spent in user space and runtime spent in system space.

The program starts by parsing the input for number of machines, number of jobs and number of operations per job. Then the input is read for the assignment of operations to machines. During this step the oblivious schedule is created. The schedule is represented as an array of linked lists `GList** sched`. Each list represents a job and contains elements representing operations. In this way operations within a job can be reordered efficiently. An operation is represented as a structure

```
typedef struct __op_t {
    int time_start;     /* when does this operation start */
    int p;              /* time it takes to process this operation */
    int p2;             /* p' = 2 ^ ceil(log_2 p) */
    int m;              /* machine to process operation */
    int j;              /* which operation is this within current job */
} op_t;
```

Initially, `time_start` and `p` are not filled in. Afterwards, the lengths of operations are read in. However, the input is indexed by job and machine, rather than job and sequence number of the operation. Therefore, the time taken to process an operation of each job on each machine is recorded in a 2-dimensional array indexed by job and machine number.

```
int** p;        /* p{j in J, a in M}, >= 0;
                 * processing time of j on a
                 */
```

After the input is parsed, the maximum job length $P_{\max}$ and maximum machine load $\Pi_{\max}$ are calculated.Random initial delays in the range $[0, Pi_{\max}]$ are generated for perturbing the schedule. The delays are stored as the `time_start` value of the first operation of each job, i.e. the first item in each list in `sched`. The random values are obtained using the function `RAND_bytes()` from the `openssl` project. In this way, "good" random values are obtained.

Afterwards, the schedule lists are walked starting at first operation and going through the list to the last operation for each job. The length of each operation is looked up and filled in the `op_t` structure. As we start with the first operation and iterate over through successive operations, `time_start` is filled in to indicate the global time when respective operation is scheduled to start. Operations of length zero are removed. These correspond to intersections which a robot does not cross.

The schedule is then split into blocks and each block is processed (spread and flattened) recursively. When this recursive processing finishes, total length of the schedule is found by looking at the finishing times (`time_start` + `p`) of the last operation of each job. The time spent in user space and in system space is obtained using the function `getrusage()` and output consisting of the schedule length and runtime is produced.

The recursive rescheduling of blocks is performed in the following function

```
/*
 * return time difference for duration of rescheduled block
 * sched         - complete schedule
 * block_start   - pointers to operations at which block starts
 *                 (first operation after end of previous block,
 *                  not neccessarily in this block)
 * t_block_start - time when the block starts within schedule
 * t_block_len   - length of block to process
 * t_new_block_start - time when the block starts within schedule
 *                     after the previous blocks have been spread
 *
 * block_end     - used to return pointers to operations which are
 *                 the first after this block (including its final fragments)
 * returns new length of the block
 *
 * Note that block_start and block_end have to be of length n.
 */
int reschedule_block(GList** sched, GList** block_start,
                     int t_block_start, int t_block_len, int t_new_block_start,
                     GList** block_end)
```

`block_start` is an array of pointers to operations which are the first ones for each job after the end of the previous block. Usually, this are the first ones in the current block, but they might be in a further block as well in case this job has no operation to be processed in current block. `block_end` is used in a similar way in order to determine `block_start` for rescheduling successive blocks.

x`t_block_start` and `t_block_len` are used to mark where the block starts and how long it is. These times are global times using the initial values from the perturbed schedule. Before being processed, the operations have `time_start` as set in the initial perturbed schedule. This is updated when they are moved into the final fragment. Note that operations in earlier blocks may already have been rescheduled, rendering the `start_time` of operations in this block incorrect. It would be possible to always update the start times of all operations after the one currently being moved to the final fragment, but many updates have to be performed, negatively impacting the runtime. Instead, we always know where is the boundary of already rescheduled and not yet rescheduled operations. Therefore, we know for which operations the `time_start` has already been updated. By inserting final fragments, the schedule is made longer by a certain amount. The parameter `t_new_block_start` is used for specifying exactly this difference, by which the schedule got longer, so that `time_start` values could correctly be updated for operations currently being processed. Although processed recursively, the schedule is still processed in a rather sequential manner with each operation being moved into a final fragment only once and hence it is possible to do these updates based on `t_new_block_start`.

In the function, large operations (operation length equal to length of the block) are

identified and moved to the final fragment. The block is then split into two smaller blocks, which are rescheduled by recursively calling the same function on them. The function returns the new length of the block, allowing to update further operation `start_time`s accordingly. After the two smaller blocks are rescheduled, the final fragment is inserted at the end of this block.

6 Results

The implementation of the Shmoys algorithm has been tested on randomly generated test cases. The test cases were representing grids of dimensions 20x20, 30x30 and 40x40. For each of the various grid dimensions, the test cases were further divided by the number of robots in the test case. For each combination of dimension and number of robots, there have been several different test cases. The test cases were provided by the instructor. A bash script (`do_tests.sh`) was written for automatically running the algorithm on all test cases and storing results in output files.

Besides test cases, output of the *Distributed Motion Planning* (DMP) algorithm has been provided as well. DMP is a randomized scheduling algorithm by Stefano Carpin. The algorithm is rather fast and usually finds the optimal solution. However, sometimes it does not find a solution at all. It is a randomized approximation algorithm and there is no known bound for the approximation it produces.

Furthermore, the *GNU Linear Programming Kit* (glpk) has been used to compute the optimal solution. Although glpk can be used to find the optimal solution, it can take very long to find it. The glpk solutions have been provided by the instructor as well. However, at the time of writing, optimal solutions have been computed only for part of the test cases.

As each of the three algorithms produces output in a different format, another bash script (`do_results.sh`) is used for analyzing the results. It calls several awk and perl scripts to process output from the various algorithms and calculate various statistics. More details about the scripts can be found in the comments at the beginning of each script. As the Shmoys algorithm is a randomized one, it has been run 10 times on each test case. The results (schedule length and runtimes) have then been averaged. Therefore, schedule lengths used in comparisons can can also be a non-integer value.

First the runtime was evaluated. It is plotted in figure 3 against the length of the schedule found. Clearly, the runtime of the algorithm is rather small, not taking more than 0.3s even on the largest test cases. Furthermore, there is a tendency of increasing runtime with increasing length of the schedule. The runtime is, however, clustered into several classes rather than increasing continuously with the length of the schedule. A possible explanation for this could be that operation times are rounded up to the next power of 2. The classes may represent thresholds, after which the next power of 2 is hit and more blocks have to be processed. As the observed runtimes are acceptably low and measuring such small runtimes naturally is less accurate, further reasons for the clustered runtime behavior have not been investigated.

A comparisons of the length of schedules found by the various algorithms can be found

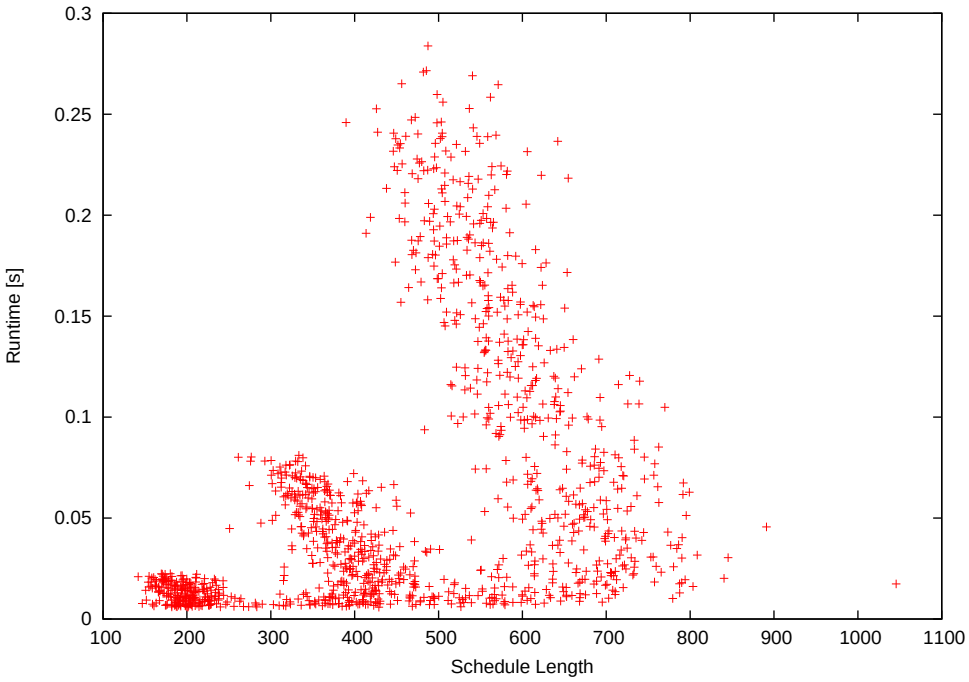

Figure 3: Runtime of the Shmoys algorithm

in figures 4, 5 and 6. The ratio of schedule length found by the Shmoys algorithm to glpk and to DMP algorithm is shown. It is plotted as percentage. For the case where glpk or DMP solution is missing, the corresponding ratio is set to zero. Although the Shmoys algorithm never found the optimal solution, it always has found a suboptimal one. Furthermore, we have a bound on how far the solution could be from the optimal one. From the plots it can be seen that the solution was within 5 times the optimal one for all test cases. Furthermore, there seems to be a pattern of getting closer to the optimal solution with increasing number of robots. However, the optimal solution has not yet been found with glpk for a large enough number of test cases to confirm this. As the DMP algorithm is expected to give worse approximations with increasing number of robots, the notion of increasing performance is probably fake.

It should be noted that DMP found the optimal solution for 97.52% cases of the ones for which we have a glpk solution and for 3 cases the solution was +1 worse than the optimal one. For all the 1220 test cases, DMP failed to find a solution 36 times, which corresponds to 2.95%. From these 36 cases, 3 were for the 30x30 dimension and the rest was for the 40x40 dimension.

7 Conclusion

The randomized version of the Shmoys algorithm (first polynomial-time polylogarithmic algorithm for job shop scheduling problem) has been implemented in C and tested on a

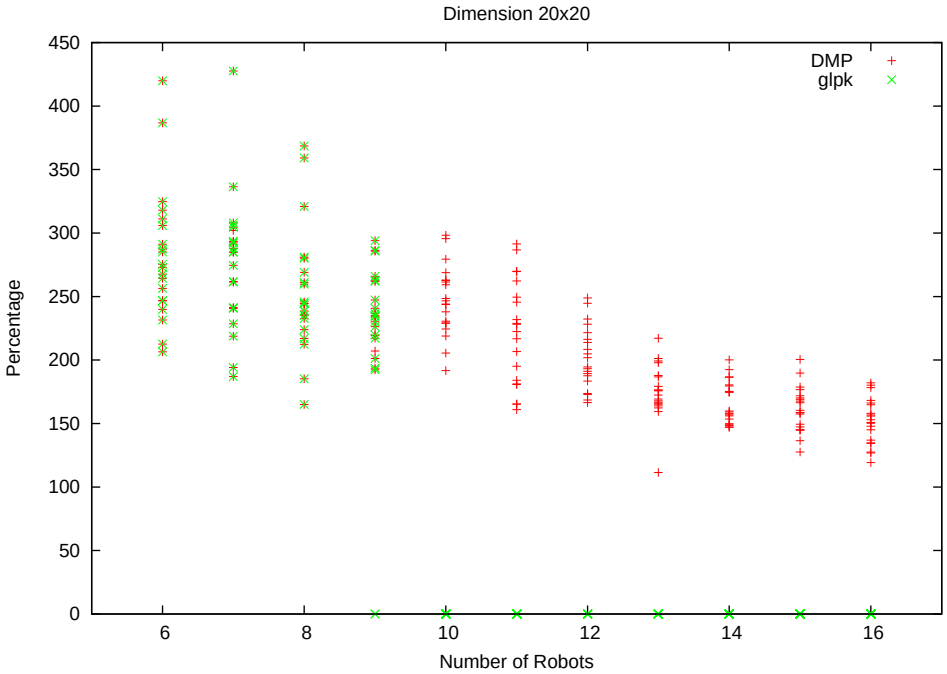

Figure 4: dimension 20x20, percentage – ratio of found schedule length to glpk and to DMP algorithm

number of random test cases. The randomized version has a bound of $O(\frac{log^2(m\mu)}{loglog(m\mu)}C^*_{max})$ on the approximation with high probability. The algorithm always finds a solution, though the solution is suboptimal. The runtime of the algorithm is relatively low, below 0.3s, so it may be viable to run the algorithm several times and pick the best solution. Furthermore, the DMP algorithm could be augmented in a way that if it does not find any solution, it would run the Shmoys algorithm to find a suboptimal solution.

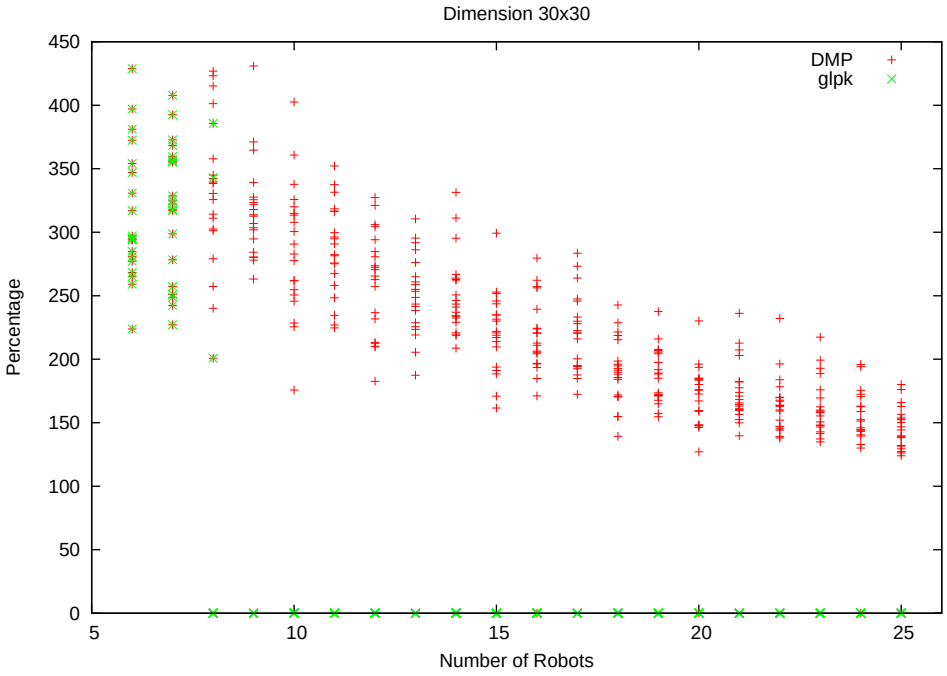

Figure 5: dimension 30x30, percentage – ratio of found schedule length to glpk and to DMP algorithm

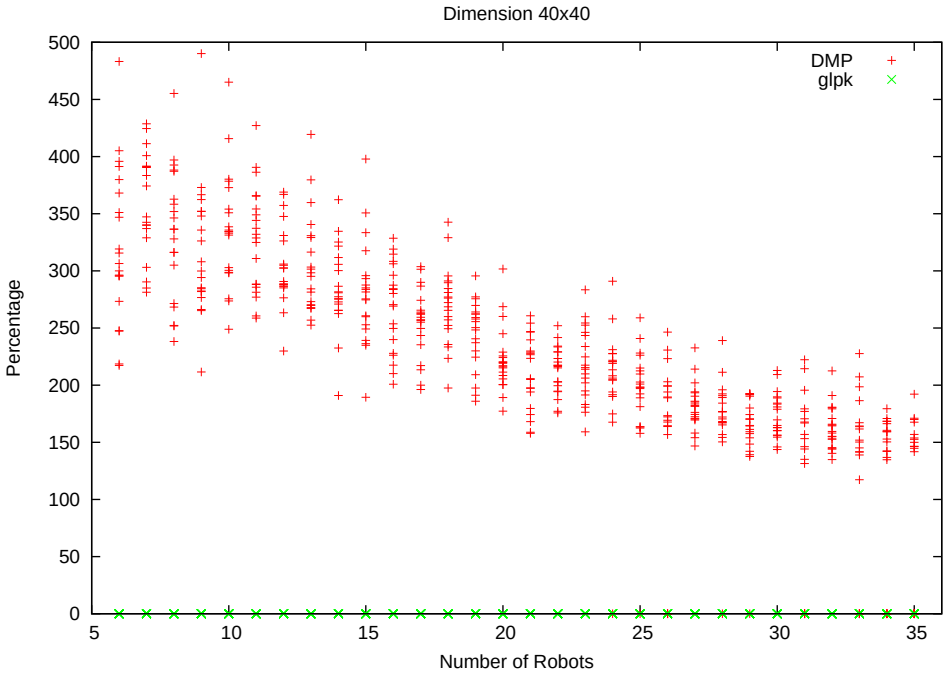

Figure 6: dimension 40x40, percentage – ratio of found schedule length to glpk and to DMP algorithm

References

[1] Glib. http://www.gtk.org/.

[2] openssl. http://www.openssl.org/.

[3] S. Akella and S. Hutchinson. Coordinating the motions of multiple robots with specified trajectories. In *Proceedings IEEE International Conference on Robotics & Automation*, pages 624–631, 2002.

[4] David B. Shmoys, Clifford Stein, and Joel Wein. Improved approximation algorithms for shop scheduling problems. *SIAM Journal on Computing*, 23(3):617–632, June 1994.