DATA STRUCTURES AND ALGORITHMS

Greedy Algorithm Analysis

Introduction

We have a scheduling problem where we are given a list of job offers and want to find a subset of jobs that do not conflict with each other and make the most amount of money as possible (where each job pays out $1). A job is compatible when it does not overlap with the others start times and end times. In order to find the optimal solution, each team member in the group implemented a separate strategy:

- Shortest interval
- Earliest finish time
- Fewest conflicts
- Earliest start time

It is hypothesized that each of the four greedy algorithms will not perform equally in returning a list of compatible job offers. Since greedy algorithms in essence are not brute-force search strategies that do not take into consideration each permutation of job offers. Keeping this in mind, the fewest conflicts as well as shortest interval strategies are predicted to be the highest performing out of the four strategies.

Methodology

Each strategy was tested with multiple different inputs in order to determine its efficiency. We attempted to identify edge cases for each method that would demonstrate a flaw in the strategy. A strategy is suboptimal if it returns less jobs than an optimal solution for the Job Scheduling problem. We found this by designing input sets that exposed the flaw in the strategy. We also ran our methods on large sets of data to broadly understand how well they operate.

These large random job lists were created using variations on the following script:

```
#! /usr/bin/env bash
time for i in `seq 1 100`;
do
        mod=$(( ( RANDOM % 77 )  + 1 ))
        a=$(( ( RANDOM % mod )  + 1 ))
        b=$(( ( RANDOM % mod )  + 1 ))
        c=$((a+b))
echo $i $a $c
done | sort --random-sort
```

This creates a pseudo random list of 100 jobs, including duplicates, with varying intervals between 1 ~ 100. An example of the output is provided in the graphs and data section. To find how our methods compared to the optimum solution for the job scheduling task, we tested our methods on job lists of 100, 1000, 1 000 and 20000 in size.

We ran five unique instances of each job list, for the different sizes. E.g. There are five unique job lists that are 100 jobs in length, five unique job lists that are 1000 jobs in length, etc. We recorded the results in a table and took means for each strategy.

We ran the same instances of input on an unknown benchmark we were given, which outputted an optimum solution but the implementation was not given to us. We used this benchmark as a baseline to compare our methods effectiveness and used this method:

$$\frac{mean\ jobs\ returned\ of\ \ given\ method}{mean\ jobs\ returned\ of\ \ optimum\ method} \times 100$$

This gives us a percentage representation of the effectiveness of our method, compared to the benchmark. E.g. for one set of tests by Earliest Start was ~40.5% as effective as the unknown benchmark over a list of 100 random jobs.

The raw data from our tests is included in the graphs and data section of this report.

Shortest Interval Time

Description of method

This method operates by calculating the difference of the finish and start times of the job and storing that value with the job class. The jobs are then sorted ascendingly by interval size. A list is created, containing all jobs that do not conflict with the shortest job in the list. In the event of a tie, where two jobs have the same interval time, they added into the list in the order they were entered at input.

We then compare each job in that list for conflicts with every other job in that list in a nested for-loop.

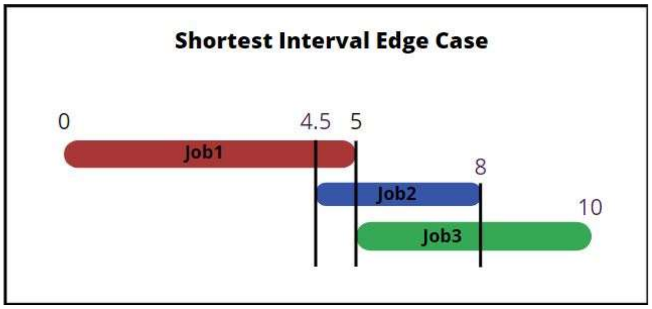
In pseudo code, we find conflicts by comparing two jobs thusly:

```
if(jobA == jobB) ignore
if( jobA.startTime <= jobB.startTime) AND jobA.finishTime >=  jobB.finishTime){
        jobA conflicts with jobB
}
if( jobA.startFinish <= jobB.startTime) AND jobA.startTime >=  jobB.finishTime){
        jobA conflicts with jobB
}
```

Evaluation

Worst Case:

This method performs suboptimally at the Job Scheduling task. This is due to a specific case that occurs when the start and finish time of a job lie within the interval of two other larger jobs. e.g:



When this situation occurs, we consider Job2 then Job3 then Job1. Since Job2 conflicts with Job1 and Job3, we ignore them as options. However Job1 and Job3 are compatible and would be included by an approach that found the optimum solution.

Best Case:

Otherwise this strategy would operate optimally on a list of jobs that does not contain instances of the edge case we have identified. Our largest test for this method (refer to graph G) found that this method was ~98.6% as efficient as a method that found the optimum solution. We can see the effect of the edge case over a large input over randomly generated jobs (refer to graph H).

Earliest Finish Time

Description of method

This method function compares the different finish time of jobs and would return a sorted list of jobs in ascending order by finish time. The final selection list would consist of all non-conflicting jobs and the selection is most earliest finish combination in considering the most greedy situation.

It would functional when there have two conflict jobs, then the method will identify which one finish earliest and choose to continue the following matching:

```
if (job_1.StratTime conflict job_2.FinishTime)  Then do if (job_1.FinishTime<job_2.FinishTime):
        Choose job_1 to continue - (converse);
```

The other possible coding (compare the size of list of job and the last job FinishTime):

```
 If (job_1. FinishTime < job_2.StartTime):
 Add job_2 after job_1;
While the List complete -previous combination ListA/ -current combination ListB
If (ListA.size<ListB.size){
Return ListB;} else if (ListA.size==ListB.size){
If (ListA.lastjob.FinishTime<ListB.lastjob.FinishTime){ return ListA;}
```

Evaluation

Best Case:

Referring to the test results, the earliest finish time method finds the optimal solution most times. In the small instance sizes, the output of the selected jobs were always the best compared with other methods. On the other hand, referring to each worst situation of the other strategies. The logical of earliest finish time can perfectly handle it, when the conflicts happen it will only consider the earliest finish one and continue. Therefore, in theory this method is the optimal solution with consider is the same as the benchmark which we are using as a baseline for the optimum. (see the graph section below)

Comparison to the other methods:

While the worst case occurs in the shortest interval strategy, referring to the diagram above, job1 and job3 will be ignored because they all conflict with job2. And the method will choose job2 to continue match the following jobs. And compare the worst case of fewest conflicts, the jobs are conflicting with each other. The method only would select the ones which have less conflict. Further comparison with special worst case of earliest start time strategy, as below diagram shows when some jobs conflict and will only return the earliest one even that is not the best selection. Assume those cases in earliest finish method, this strategy is always selecting the earliest finish job as continue then would have the optimal solution. Generally, there is no worst case that we can find in this strategy.

Fewest Conflicts

Description of method

This method works by ordering a list of jobs in ascending order from the least amount of conflicts to the most. Then systematically from the first to the last term in the list, jobs are added to a final configuration list. Noting that for a job to be added to this configuration it must not conflict with any of the jobs that it already contains (which by default have either a less than or equal number of conflicts)

Evaluation

Best Case:

The best case for this method is when the jobs sizes are relatively low, where there is a lesser likelihood of stacking of the jobs. Additionally, when there is some degree of uniformity within the jobs themselves, for instance the case where all the jobs had the same interval size or the same number of conflicts, this greedy algorithm performed optimally.

Worst Case:

The worst case for this job occurs when there is stacking in the job interval times. In one instance (see fig below), a job in the second row of one of the stacks has the least number for conflicts and is added to the final list first. However, the two jobs that this minimal conflicting job intersects with are in the row above and form the optimal task list of 4. Including the minimal conflicting job in the final task list is only able to provide three suitable jobs.

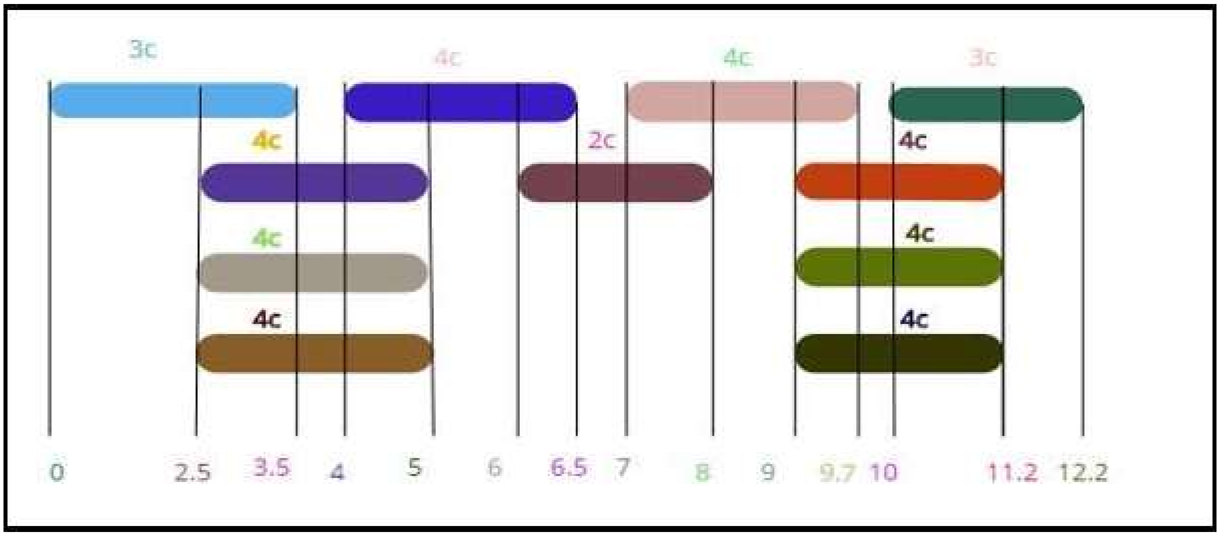


Fig: fewest conflicts worst case, ‘c’ represents the number of conflicts for that job

Earliest Start Time

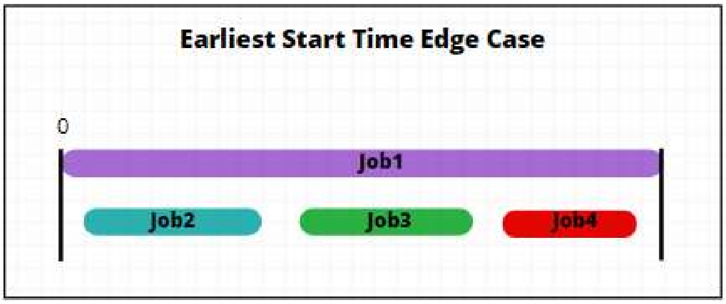
Description of method

For this strategy, the list of jobs is sorted by the earliest start times in ascending order with a priority queue. The jobs that have the same start times are ordered arbitrarily. The job at the top of the queue (the one with the earliest start time) is then added to the final list of compatible jobs. Jobs in the priority queue are only added to the final list if they do not conflict with the jobs in the final list.

Evaluation

Worst Case:

This method performs very poorly in most cases. In a list of jobs, it only takes one with a long interval and early start time to take over, making it impossible to include many jobs that are shorter in length but start later.



The diagram above shows the obvious problem this strategy has. The solution for earliest start time is 1, and for the optimal is 3.

Best Case:

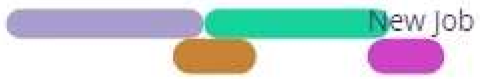
For small inputs, this strategy performs well when there are shorter intervals that do not conflict. However, those inputs are also the best cases for the other strategies. So, at best, this strategy's solution is the same as the other strategies, but not better.

Evaluation

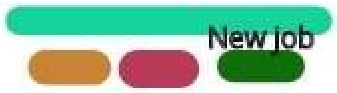
We evaluated the effectiveness of our methods in three separate ways. In the first, several random jobs were generated where the input list used was 100, 1000, 10000 and 20000 jobs. This was an initial tactic to see how well our algorithms scaled, as the four methods performed on par with one another with smaller input lists (20 or less). It was found that the earliest finish time was able to retrieve the maximum job list every time, with the shortest interval a very close second. The other two jobs were only able to pick a fraction of the list generated by earliest finish and shortest interval strategy.

In the next stage, generalized test cases were used for which the input list was kept to a constant number of 20. These included using lists where all the intervals had the same length, the job list provided was already sorted based on starting time, and the number of conflicts each job had was equal. The results for these tests were considerably better for the earliest start time and fewest conflicts. Either they were able to obtain the maximum or they were one or two shy. The other two strategies were able to consistently achieve the optimal number.

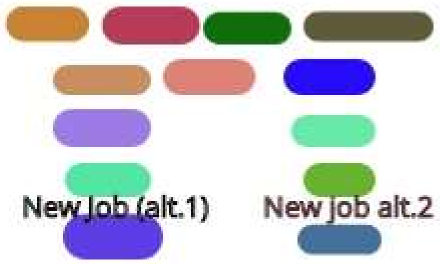
The final way these jobs were tested was by using specific instances that could potentially exploit flaws within the respective strategies. These instances were discovered through a combination of trial and error as well as intuition, where a small number of jobs usually three or four would be placed on a diagram and what the optimum find was compared with what the specific strategy would produce as a suitable job list. In this process no input configuration of jobs could be formulated for earliest finish time that would produce a different result to the optimum. For the other three strategies, specific instances were able to be designed to highlight flaws within the strategy. Focusing on the three non-optimal strategies, it was initially quite surprising how the shortest interval vastly outperformed the other two despites being a non-optimal solution. Upon closer inspection it becomes clear that the answer lies with just simple probability. The flaw for shortest interval is a short interval conflicting with two longer intervals. One must note that if another job was added to this configuration that didn't conflict with the shortest interval job, the difference between the optimum and the strategy would be the same.



In the case of the earliest start time, the flaw arises when a long interval at the start conflicts with multiple short intervals that do not conflict with one another but start at later times. If a new job was added to this list there is no guarantee it will return the optimum as if it conflicts with the longest interval job at the start, a sub-optimum solution would be the output.



In the case of fewest conflicts, that strategy is subject to arrangement bias where in many cases jobs that have the least number of conflicts, with whatever jobs that they do conflict with, hide the true solution. Where positioning of a new job unless it directly increases the number of conflicts for the least conflicting job, this flaw would be further exploited.



Thus, the only real difference between earliest finish and shortest interval is that shortest interval does have a flaw that comes up in a small number of instances but can be removed by the positioning of a new job.

Conclusion

After evaluating the strategies with various inputs, we can rank the strategies based on how close they are to the optimum solution, with the first being the optimum:

1. Earliest Finish Time
2. Shortest Interval
3. Fewest Conflicts
4. Earliest Start Time

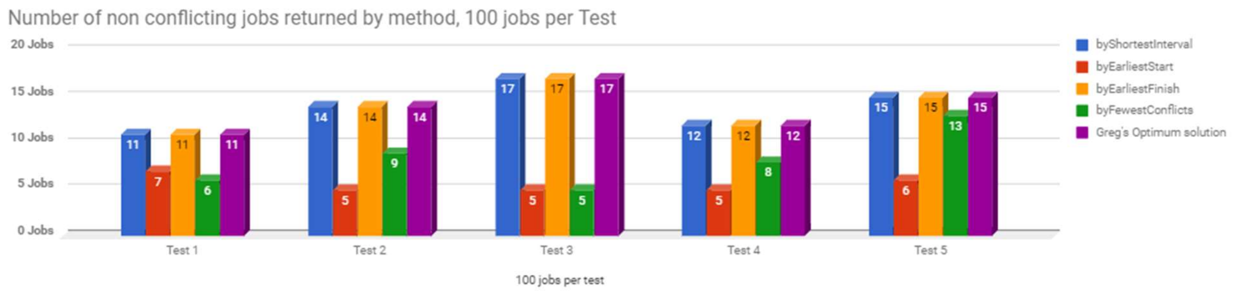
Earliest Finish always came out with the optimum, thus we can rank it as number 1. Shortest Interval was close to the optimum even with very large inputs, but due to the minor flaw described earlier, it is ranked as number 2. Fewest Conflicts performed well in some cases with small inputs, but not as well as the first two in larger inputs, so it gets ranked at number 3. And because Earliest Start performed very poorly on almost all inputs, it takes last place.

Graphs and Data

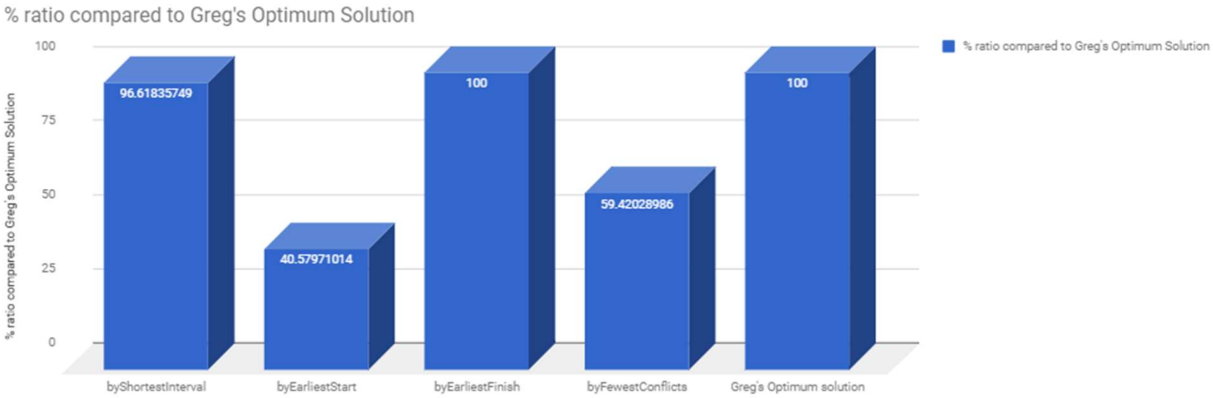
Lists of 100 randomly generated Jobs

	byShortestInterval	byEarliestStart	byEarliestFinish	byFewestConflicts	The unknown benchmark
Test 1	11	7	11	6	11
Test 2	14	5	14	9	14
Test 3	17	5	17	5	17
Test 4	12	5	12	8	12
Test 5	15	6	15	13	15
average	13.33333333	5.6	13.8	8.2	13.8
Percent compared to optimum	96.61835749	40.57971014	100	59.42028986	100

Graph A



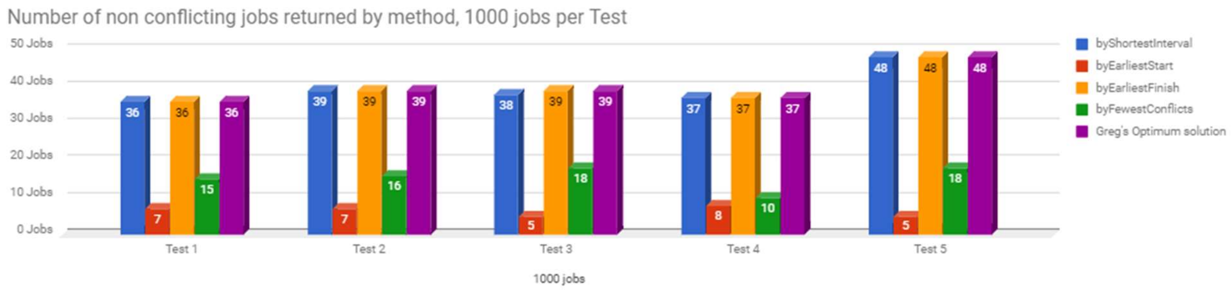
Graph B



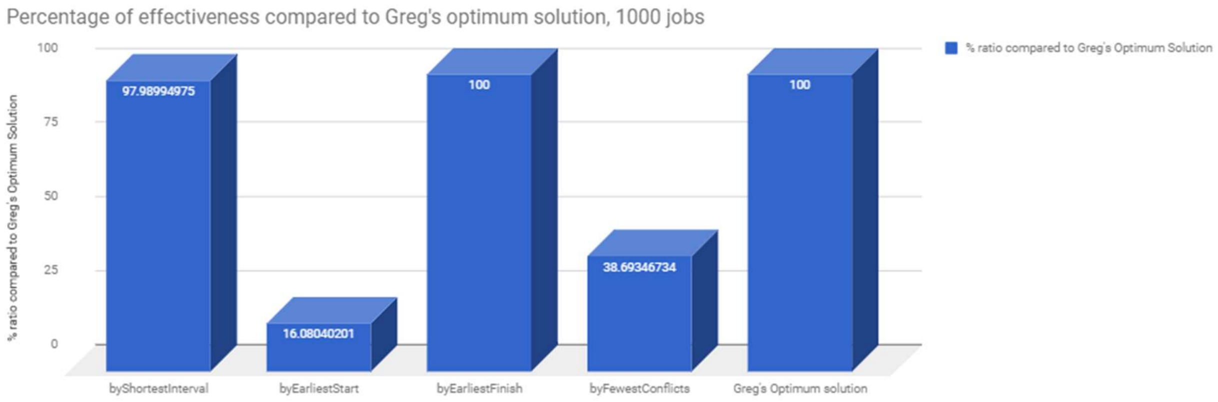
Lists of 1000 randomly generated Jobs

	byShortestInterval	byEarliestStart	byEarliestFinish	byFewestConflicts	The unknown benchmark
Test 1	36	7	36	15	36
Test 2	39	7	39	16	39
Test 3	38	5	39	18	39
Test 4	37	8	37	10	37
Test 5	48	5	48	18	48
Average	39	6.4	39.8	15.4	39.8
Percent compared to optimum	97.98994975%	16.08040201%	100%	38.69346734%	100

Graph C



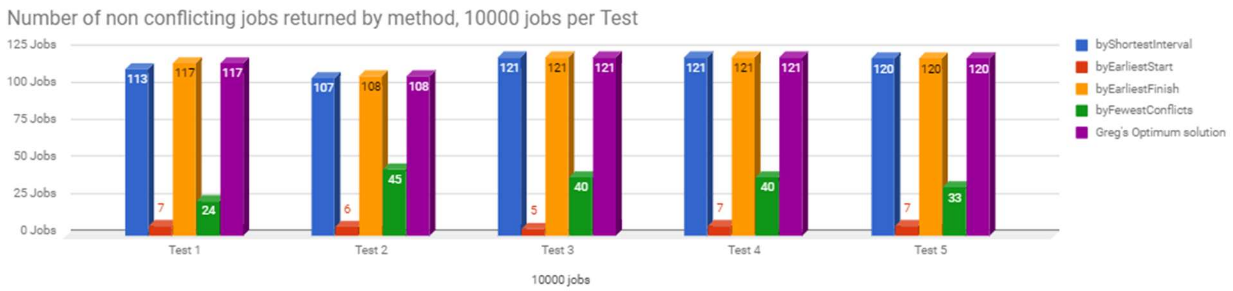
Graph D



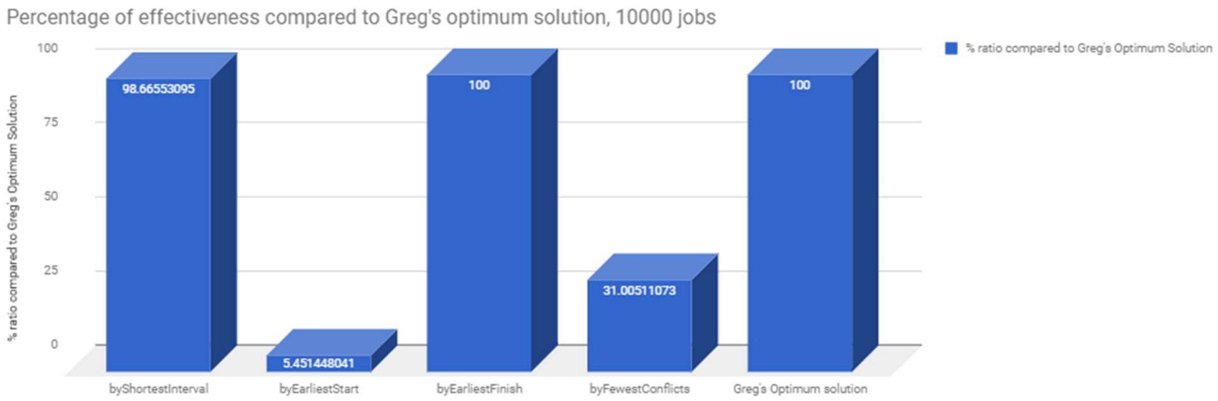
List of 10 000 Random Jobs

	byShortestInterval	byEarliestStart	byEarliestFinish	byFewestConflicts	The unknown benchmark
Test 1	113	7	117	24	117
Test 2	107	6	108	45	108
Test 3	121	5	121	40	121
Test 4	121	7	121	40	121
Test 5	120	7	120	33	120
average	115.8333333	6.4	117.4	36.4	117.4
Percent compared to optimum	98.66553095	5.451448041	100	31.00511073	100

Graph E



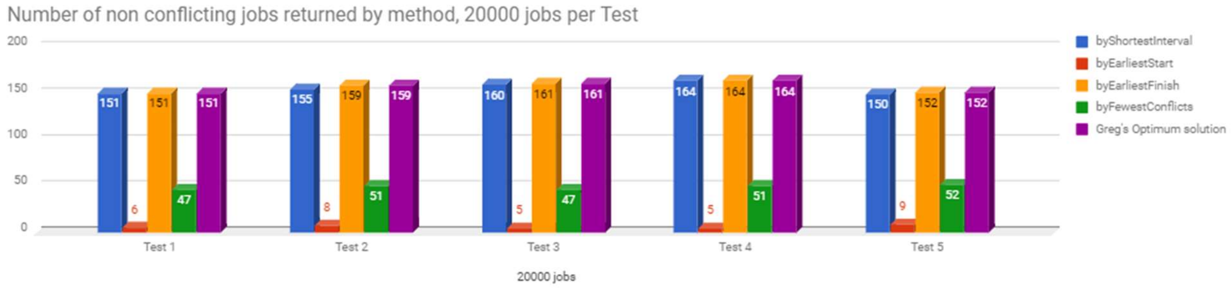
Graph F



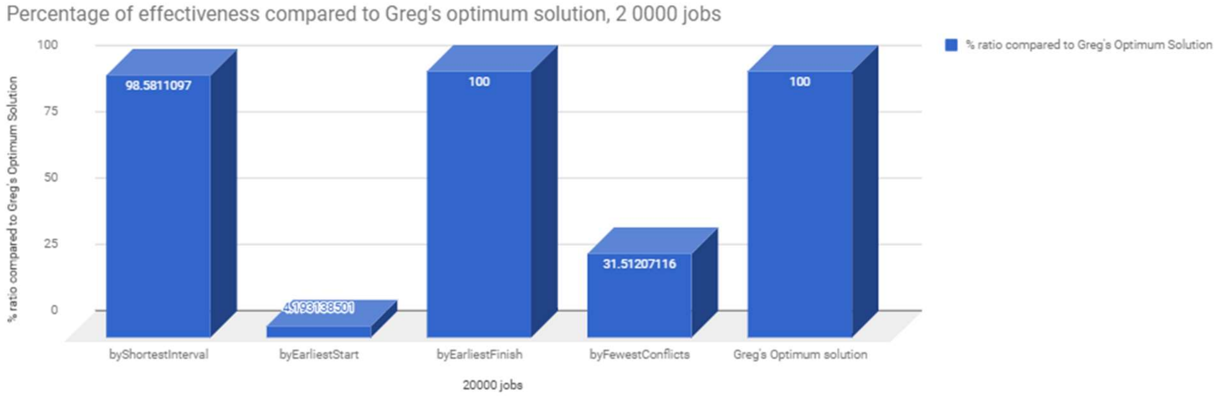
List of 20 000 Random Jobs

	byShortestInterval	byEarliestStart	byEarliestFinish	byFewestConflicts	The unknown benchmark
Test 1	151	6	151	47	151
Test 2	155	8	159	51	159
Test 3	160	5	161	47	161
Test 4	164	5	164	51	164
Test 5	150	9	152	52	152
average	155.1666667	6.6	157.4	49.6	157.4
Percent compared to optimum	98.5811097	4.193138501	100	31.51207116	100

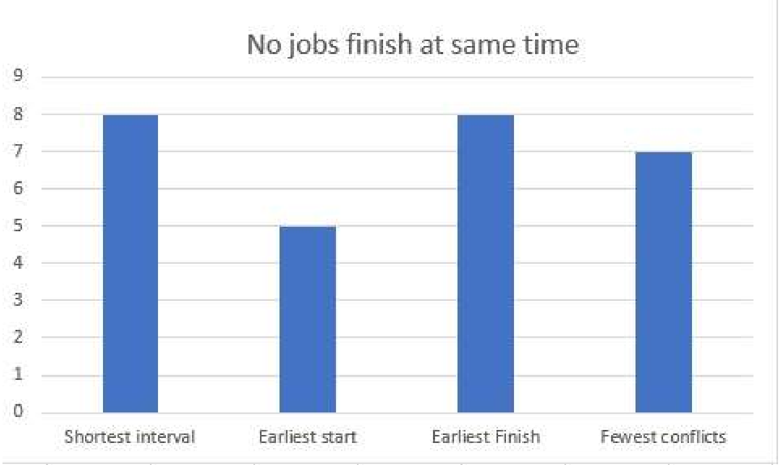
Graph G



Graph H



Graph I



Example output from pseudo random job generator:

```
#! /usr/bin/env bash
time for i in `seq 1 20`;
do
        mod=$(( ( RANDOM % 77 )  + 1 ))
        a=$(( ( RANDOM % mod )  + 1 ))
        b=$(( ( RANDOM % mod )  + 1 ))
        c=$((a+b))
echo $i $a $c
done | sort --random-sort
```

output:

```
17 21 22 | 16 2 4 | 4 33 43 | 12 26 78 | 11 4 7 | 8 2 3 | 2 7 13 | 15 1 60 | 10 17 40 | 5 3 6 |
1 13 33 | 7 44 70 | 13 40 82 | 18 28 52 | 19 1 3 | 6 13 31 | 20 11 22 | 3 32 63 | 14 24 48 | 9
26 41 |
```