Single-machine Scheduling with Supporting Tasks (From the Perspective of Fixing the Setup Job Sequence)

Ian Lin, Jin-Liang Ma
Department of Infomation Management and Finance
National Yang Ming Chiao Tung University
ianlienfa.mg07@nycu.edu.tw

1 Keywords

Scheduling, $1|s-prec|\sum C_j$, Branch and Bound, $1|out-tree|\sum C_j$

2 Motivations

There is a commonly-seen situation in industries – For companies that hold a large number of computation units, setup tasks, for instance, container setups, package downloads, etc., should be finished before an active job could kick off on a machine.

We call these prerequisite jobs setup jobs $S = \{a_1, a_2, a_3, ...a_m\}$ and the jobs that follow test jobs $T = \{b_1, b_2, b_3, ...b_n\}$. Take figure 1, for instance; setup jobs a_1, a_2 should be finished before test job b_1's execution. The notation of this problem is $1|s-prec|\sum C_j$ where $s-prec$ denotes the dependencies between the setup and test jobs. C_j is the completion time of job j. $\sum C_j$ represents the objective that we want to minimize – the sum of the completion time of all jobs. The problem was proposed in [Kononov et al., 2015] and was proven to be an NP-hard problem.

3 Recent Research and Comparison

[Kononov et al., 2015] first prove that this problem is NP-hard based on a reduction from $1|prec, pj = 1|\sum w_j C_j$, then approaches this problem from two aspects, fixing the test-job sequence and fixing the setup-job sequence. Given a fixed job sequence, the optimal schedule can be found using a nested for loop, and the overall time complexity for it is O(mn), where m is the number of setup jobs and n is the number of test jobs. Given a fixed setup job sequence, the optimal schedule can be found by reducing the problem into $1|out-tree|\sum C_j$. The reduction process takes O(mn) and the solve of $1|out-tree|\sum C_j$ takes $O(nlogn)$, so problem is solvable in $O(n(\max(m, logn)))$ time.

4 Main Contributions

For this project, we've made following contributions:

1. Provides an realistically O(nlogn) algorithm for $1|out-tree|\sum C_j$ problem
2. Provide Integer Programming Formulation for $1|s-prec|\sum C_j problem$
3. Provide experiment results on the performance with Lagrangian Relaxation, SRPT as lower bound
4. Performance results on local search, generic algorithms, and branch and bound.

5 Research methods

We started the research with first formulating $1|s-prec|\sum C_j$ into the integer programming formulation , then run it on the gurobi solver for the baseline result.

Furthermore, to look into the problem's traits, we consider the problem with fixed setup job sequences. Once a complete setup jobs sequence is given, we can reduce the problem into $1|out-tree|\sum C_j$ and solve it in $O(n(\max(m, nlogn)))$ time. The problem is then about generating the setup jobs sequences. We approach it with Branch-and-Bound, local search, and the Generic algorithm. (Note that local search and generic search do not guarantee the finding of the global but a remarkable local minimum)

Moreover, for the Branch and Bound part, we tested out the effectiveness of using Lagrangian Relaxation and SRPT as lower bound.

6 Implementation and experiments

6.1 Integer Programming with Gurobi

Parameters

- s: the number of setup jobs
- t: the number of test jobs
- p: the number of the positions, which is equal to s+t
- time[]: the processing time for each job
- pre_list[]: all of the setup jobs which are the predecessors of a test job

Variables

- X_{ij} : boolean variable indicating whether job i runs on position j

Constraints

- C1 : Each position runs one job

$$\sum_{i=0}^{s+t} X_{ij} = 1, \forall j \in (0, p)$$

- C2: Each test job need to run once

$$\sum_{j=0}^{p} X_{ij} = 1, \forall i \in (s, s+t)$$

- C3: All of the setup jobs for a test job need to run before it kicks off

$$\sum_{i'in}^{\text{pre\_list[i]}} \sum_{j'=0}^{j} X_{i'j'} \leq X_{ij} \times \text{size (pre\_list[i])}, \forall i \in (s, s+t), \forall j \in (0, p)$$

Objective function

$$\text{Minimize} \sum_{i=s}^{s+t} \sum_{j=0}^{p} X_{ij} \times (\sum_{i'=0}^{s+t} \sum_{j'=0}^{j} X_{i'j'} \times \text{time[i']} + \text{time[i]})$$

6.2 Solving the problem given fixed setup job sequences

6.2.1 Reducing the $1|s-prec|\sum C_j$ problem to the $1|out-tree|\sum C_j$ problem

Given a fixed setup job sequence (see Fig 2), [Kononov et al., 2015] states that we can reduce the problem into $1|out-tree|\sum C_j$ with the following steps:

Figure 1: The relationships of setup and test jobs

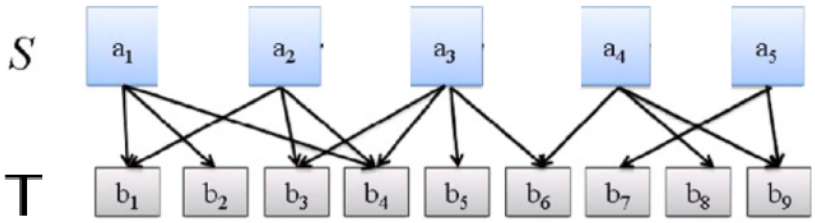
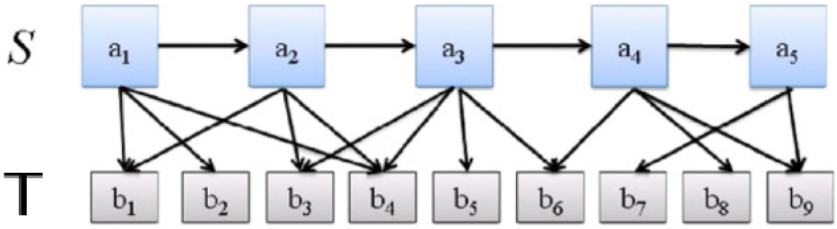


Figure 2: Fixing the setup jobs sequence



1. For a test job that is with more than one required setup jobs, retain the precedence constraint for the setup job with the latest completion time (see Fig 3)
2. Now, every test job will only have at most one setup job as its predecessor, therefore forming an out-tree (see Fig 4)

6.2.2 Solving $1|outtree|\sum C_j$ problem in $O(nlogn)$ time

The $1|out-tree|\sum C_j$ algorithm is proposed in Adolphson and Hu [1973] Here we use a special data structure called Uheap. The aim of Uheap is to retain the time complexity of tree merge to be O(logn) so that the total runtime is O(nlogn). The algorithm is shown in Figure 5, and Figure 6 shows how it works by repeatedly merging the node with the smallest q value into its parent node.

6.2.3 Algorithm explained

As illustrated in Figure 6, the algorithm starts with initializing the w, p, and q of the top node of the tree with negative infinite, ensuring that it will never be chosen to merge into its parent (who does not exist). Then, for each round, the algorithm picks out the node with the largest q value and merges it into its parent node, concatenating the merging child node to the parent nodes' job sequence and removing the child node from the tree. When only one node is left in the tree, the optimal sequence is found.

6.2.4 The Uheap data structure

An intuitive way to implement the above algorithm is first to construct a tree structure, then use a Depth-first Search for each round to find the node to merge. However, it produces a $O(n^2)$

Figure 3: Delegating the constraints to the setup job with the latest finish time

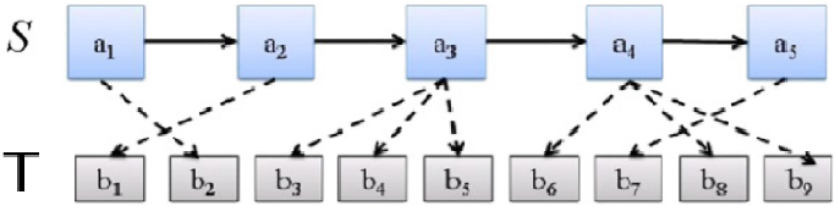
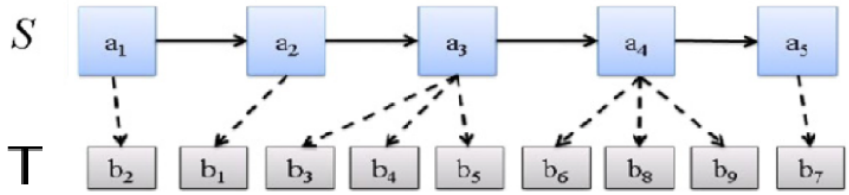


Figure 4: Reorder the test job to form an out-tree



runtime. One might consider using a maximum heap to store the tree nodes to improve the runtime. Nevertheless, saving the node solely using a heap structure causes a problem of information loss and, from time to time, makes it impossible to find a node's parent node. (See the example provided in Figure 6) Therefore, we use a unique data structure called Uheap, combining a heap and a union set. Uheap provides quick merging heap update and therefore keeps the algorithm runtime O(nlogn).

Figure 7 shows the Uheap data structure. Uheap combines a union set and a maximum heap, where union set tracks the tree structure and maximum heap helps pinpoint the maximum q value. In the union set part of Uheap, every index represents a job, *Seq* saves the job sequence of each node and *Set Parent* saves the parent job of one job, with this information we keep track of the merge status of jobs. The *Connect* boolean variable denotes whether a node has been merged into other nodes. The *heap pos* array stores the position of each node in the heap, connecting the two data structures.

The $1|out-tree|\sum C_j$ algorithm works with Uheap as follows (here we use heap node and set node to repectively denote the heap node and the union set node):

1. Initialize the Uheap with the top node having negative infinite w, p, and q values, and the rest of the nodes having the w, p, and q values of their corresponding jobs.
2. Build the heap and union set repectively based on the q value and the precedency of the jobs.
3. For every round (the number of jobs times), extract the heap node with the largest q value, and locate its position in corresponding position in the union set. (extract-max + heapify costs O(logn) and the locating process includes both random access and union-find (with Path compression implementaion) operations, which both costs O(1) time)
4. Merge the corresponding set node to its parent node (Flip *Connect* to true, and append job to sequence)
5. Update the heap node's q value and upfloat or heapify the heap (O(logn))

The data structure helps us to keep the runtime of the algorithm O(nlogn).

6.3 Searching for the optimal solution with Branch-and-Bound algorithm

With the help of the Uheap data structure, we are able to solve the $1|out-tree|\sum C_j$ problem in $O(nlogn)$ time. So to solve the original problem, we now have to search though all the possible setup job sequences to find the optimal one. The Branch-and-Bound algorithm is a good choice for this problem, it generates a tree of partial sequences, and with the provided lower bound, it can prune the branches that are not promising and only search through the promising branches to reduce the search space.

6.3.1 Lower Bound – General description

Lower Bound provides a conservative estimate of the ultimate solution, even with a partial set-up job sequence. During the searching process, we come across feasible solutions, and we often keep track of the best value (in our case, we wanted to minimize the objective, therefore it's the smallest feasible solution encountered) and call it the incumbent solution. If the lower bound is larger than the incumbent solution, we can prune the node and stop searching further, since it is impossible to find a better solution in this branch. Theorectically, the method can largely reduce the search space if the lower bound is tight enough.

Figure 5

Algorithm 1 | outtree | $\sum \mathbf{w_j C_j}$

1. $w(1) := -\infty$;
2. FOR $i := 1$ TO n DO
3. BEGIN $E(i) := i$; $J_i := \{i\}$; $q(i) := w(i)/p(i)$ END;
4. $L := \{1, \ldots, n\}$;
5. WHILE $L \neq \{1\}$ DO
   BEGIN
6. Find $j \in L$ with largest $q(j)$-value;
7. $f := P(j)$;
8. Find i such that $f \in J_i$;
9. $w(i) := w(i) + w(j)$;
10. $p(i) := p(i) + p(j)$;
11. $q(i) := w(i)/p(i)$;
12. $P(j) := E(i)$;
13. $E(i) := E(j)$;
14. $J_i := J_i \cup J_j$;
15. $L := L \backslash \{j\}$
    END

6.3.2 Lower Bound – Shortest-Remaining-Processing-Time-First (SRPT)

SRPT provides lower bound for $1|r_j|\sum C_j$, here we transform our problem into it. To trasform $1|s-prec|\sum C_j$ into $1|r_j|\sum C_j$, we can set the release time of each test job to be the summation of the processing time of all its setup jobs. As the search proceeds, the sequenced setup jobs will also postpone the release time of other test jobs, so their release time will be updated accordingly. According to Figure 8, the row on the top shows the processing times of the setup jobs, and the row on the bottom shows the corresponding release times of the test jobs. The middle subfigure shows if the third setup job is chosen to be done first, the release time of the other test job that it has no constraints on will be postponed by 3 time units.

To compute the lower bound using SRPT given a partial sequence of setup jobs, we can compute the release time of each test job by adding up the processing time of all the setup jobs that have no constraints on it. Then, we can compute the lower bound by sorting the test jobs by their release time and summing up the processing time of the test jobs in the order of their release time.

6.3.3 Lower Bound – Lagrangian Relaxation

For the computation of this Lower Bound, we formulated integer programming in another fashion. As shown in Figure 10, we have a form several constraints to solve the problem. The constraint that is being relaxed is the exclusivity constraints, where we allow jobs to be overlapped with each other with penalty.

6.4 Searching for decent feasible solutions with Local Search

The local search algorithm is presented in Figure 9. We use local search to generate fixed setup job sequences and compute the objective value of each sequence until the algorithm converges.

6.5 Searching for decent feasible solutions with Genetic algorithm

After local search, we wanted to further improve the solution by using genetic algorithm.

Figure 6

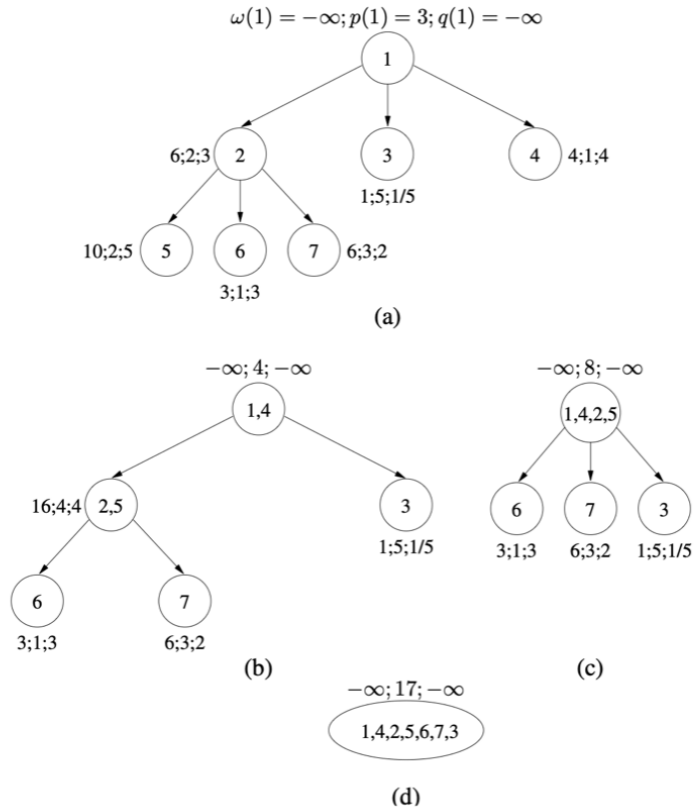


Figure 7

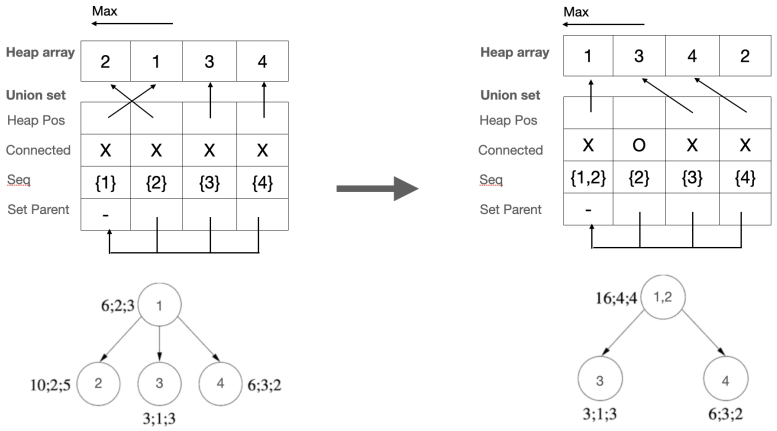


Figure 8

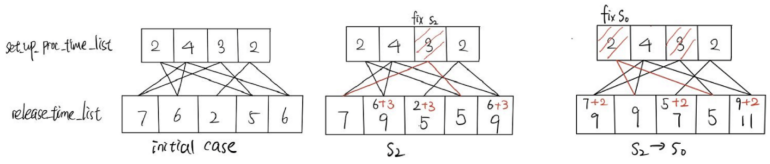


Figure 9

```
Local_search(Seq, Temp_opt) :
    FOR i :=1 TO n DO
        BEGIN
        Exchange  s_i with  s_{i+1}  ;
        IF  ∑C_j  < Temp_opt
            Temp_opt :=∑C_j  ;
        Exchange  s_i with  s_{i+1}  ;
        END
    Return Temp_opt, Seq
```

Figure 10

S: start time, C_p: completion time, W: work on machine(or not), Y: if Y_p[i][j] = 1, job i completes before j

---

- One-time constraint

  For every test job, $W[m] == 1$

- Dependency constraint

  For every pair of test & set-up job, Y_p[i][j] = 1

- Fixed sequence compliance constraint

  The sequence should follow the fixed set-up job

- Order clarity

  Y_p[i][j] + Y_p[j][i] = 1

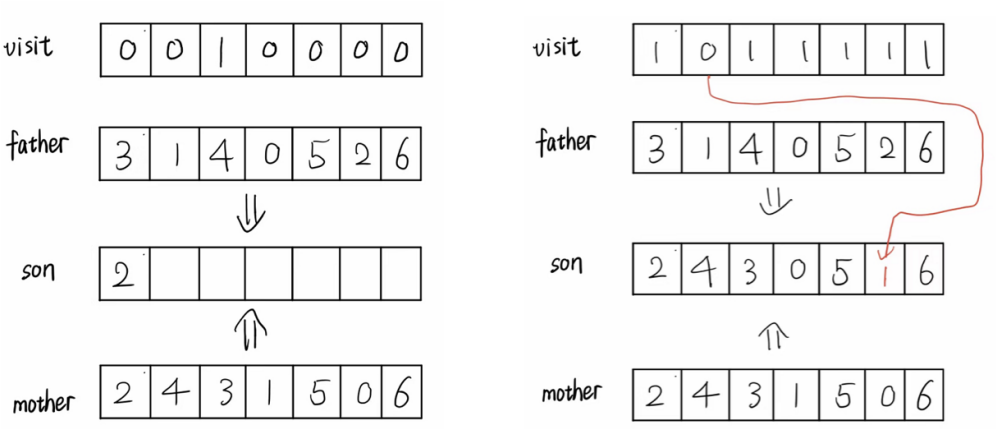
- Completion time definition

  For every test & set-up job, C[n] - (S[n] + P[n]) == 0, S[n] >= 0

- Exclusivity Constraint

  No job is allowed to overlap with any other jobs

Figure 11



Parameters

- G: generation
- P: population ‘
- S: survive rate
- M: mutation rate ’
- Fitness: every C_j computed given the corresponding fixed setup job sequence.

Cross over method is used for generating new population, Figure 11 shows an example of how a child is generated from two parents. For each round, the algorithm randomly chooses two person as father and mother. For every position of the job sequence, randomly chooses a job from either father or mother to be the gene of the child. The *Visit* array keeps track of the jobs that have been chosen, so that no job will be chosen twice.

7 Performance and Results

7.1 Lagrangian Relaxation Result

It turns out that Lagrangian Relaxation is not a good lower bound for the problem. There are two problems:

1. The Lagrangian Relaxation is fairly loose, for the instance showed in Figure 12, a good bound is around 15000, but the Lagrangian Relaxation bound is around 400. For the instance showed in figure 13, a good bound is around 15000, but the Lagrangian Relaxation bound is around 5000.
2. The Lagrangian Relaxation call from gurobi is too slow, and it takes too much time to run.

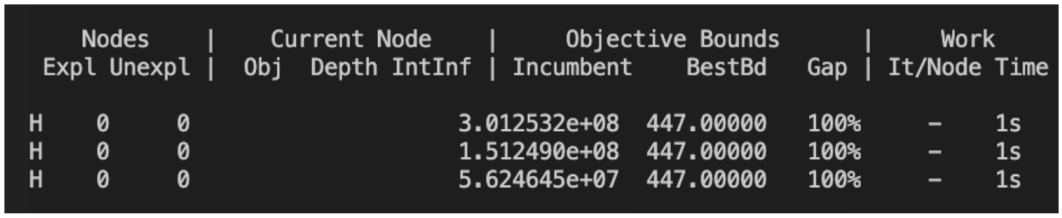
7.2 SRPT Result

SRPT provides better bound than the Lagrangian Relaxation, but it is still not good enough. From figure 14 we see that SRPT provides bounds around 10000, which is still far from 15000, a good bound for the problem.

7.3 Comparisons on the speed of convergence of different strategies

Figure 15 shows the speed of convergence of different strategies. The horizontal dashed line shows the best solution acheivable by the Gurobi solver with the integer programming method. Other shows the search progress of different strategies. We can see that generaly local search provides

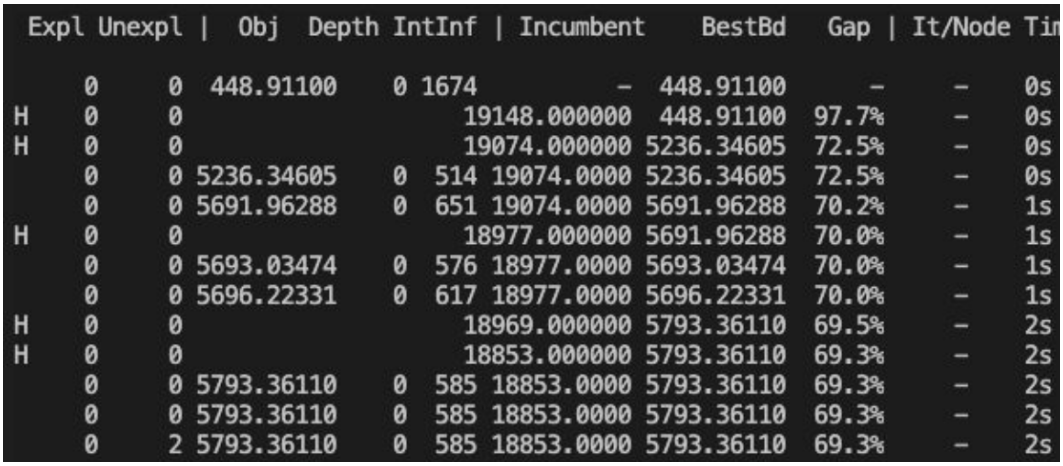
Figure 12



```
     Nodes    |    Current Node    |     Objective Bounds      |     Work
  Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

 H    0     0                    3.012532e+08  447.00000   100%     -    1s
 H    0     0                    1.512490e+08  447.00000   100%     -    1s
 H    0     0                    5.624645e+07  447.00000   100%     -    1s
```

Figure 13



```
  Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Tim

     0     0  448.91100    0 1674          -  448.91100      -     -    0s
 H   0     0                    19148.000000  448.91100  97.7%     -    0s
 H   0     0                    19074.000000 5236.34605  72.5%     -    0s
     0     0 5236.34605    0  514 19074.0000 5236.34605  72.5%     -    0s
     0     0 5691.96288    0  651 19074.0000 5691.96288  70.2%     -    1s
 H   0     0                    18977.000000 5691.96288  70.0%     -    1s
     0     0 5693.03474    0  576 18977.0000 5693.03474  70.0%     -    1s
     0     0 5696.22331    0  617 18977.0000 5696.22331  70.0%     -    1s
 H   0     0                    18969.000000 5793.36110  69.5%     -    2s
 H   0     0                    18853.000000 5793.36110  69.3%     -    2s
     0     0 5793.36110    0  585 18853.0000 5793.36110  69.3%     -    2s
     0     0 5793.36110    0  585 18853.0000 5793.36110  69.3%     -    2s
     0     2 5793.36110    0  585 18853.0000 5793.36110  69.3%     -    2s
```

best result when the number of jobs blows up. Genetic algorithm provides converges the fastest. Branch-and-Bound gurauntees the find of optimal solutions, but in practice is not fast enough to produce quality results when the number of jobs is large. For the Branch-and-Bound method, Cyclic best-first search provides better result than the best-first search and Depth-first search.

7.4 Comparisons on the number of nodes searched by different strategies

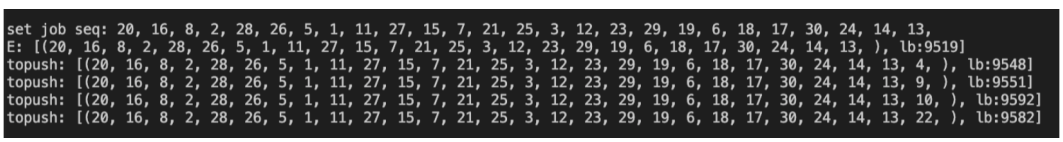
Figure 16 shows the number of nodes searched by different strategies. Here, only CBFS, DFS and Local search is shown in the figure. The BFS line is above the window. We can see that though CBFS can provide similar results searching less node than local search, local search still beats CBFS at finding a better solution.

8 Conclusion and Future Work

In the report, we present a way to solve $1|s-prec|\sum C_j$ from the perspective of integer programming. The integer programming formulation used with the Gurobi solver is able to find the optimal solution for instances with lower than respectively 7 setup and test jobs, and is able to acheive a 98.6 percent gap for respectively 30 setup and test jobs.

Additionally, we also research the perspective of solving the problem given fixed test job sequence. The problem can be further reduced into the $1|out-tree|\sum C_j$ and be solved in $O(nlogn)$ time. We proposed a data structure called Uheap to meet the time complexity requirement. With the Uheap,

Figure 14



```
set job seq: 20, 16, 8, 2, 28, 26, 5, 1, 11, 27, 15, 7, 21, 25, 3, 12, 23, 29, 19, 6, 18, 17, 30, 24, 14, 13,
E: [(20, 16, 8, 2, 28, 26, 5, 1, 11, 27, 15, 7, 21, 25, 3, 12, 23, 29, 19, 6, 18, 17, 30, 24, 14, 13, ), lb:9519]
topush: [(20, 16, 8, 2, 28, 26, 5, 1, 11, 27, 15, 7, 21, 25, 3, 12, 23, 29, 19, 6, 18, 17, 30, 24, 14, 13, 4, ), lb:9548]
topush: [(20, 16, 8, 2, 28, 26, 5, 1, 11, 27, 15, 7, 21, 25, 3, 12, 23, 29, 19, 6, 18, 17, 30, 24, 14, 13, 9, ), lb:9551]
topush: [(20, 16, 8, 2, 28, 26, 5, 1, 11, 27, 15, 7, 21, 25, 3, 12, 23, 29, 19, 6, 18, 17, 30, 24, 14, 13, 10, ), lb:9592]
topush: [(20, 16, 8, 2, 28, 26, 5, 1, 11, 27, 15, 7, 21, 25, 3, 12, 23, 29, 19, 6, 18, 17, 30, 24, 14, 13, 22, ), lb:9582]
```

Figure 15

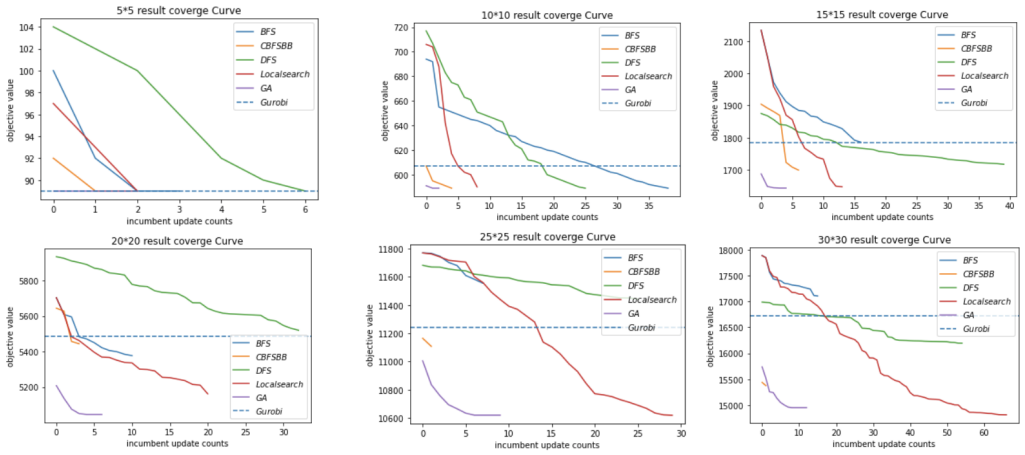
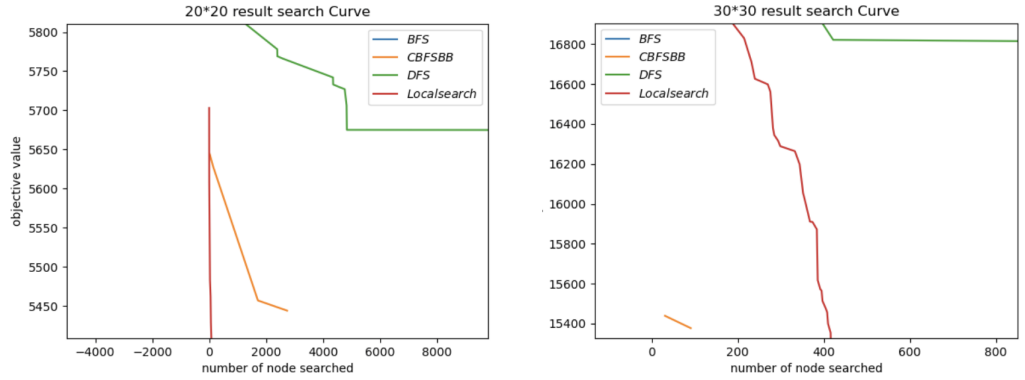


Figure 16



we are able to find optimal solution with Branch and Bound for repectively 10 setup and test jobs, and is able to find better result with local search on 30 * 30 jobs than the Gurobi solver with 98.6 percent gap

We also tried out SRPT and Lagrangian Relaxation as lower bounds for the problem and have shown their limitations for the probelms. Finally, we compared the speed of convergence of different strategies and the number of nodes searched by different strategies. We have shown that CBFS provides better result than DFS and BFS on this problem, and local search provides better result than CBFS when the number of jobs is large.

For the future works, we would like to try out other lower bounds for the problem and see if they can provide better results. We've shown that CBFS provides good architecture for Branch-and-Bound searching and pruning. However, local search provide good solution and small number of nodes searched. A good idea will be to combine both techniques together to provide a method with both benefits.

References

Alexander V. Kononov, Bertrand M.T. Lin, and Kuei-Tang Fang. Single-machine scheduling with supporting tasks. *Discrete Optimization*, 17:69–79, 2015. ISSN 1572-5286. doi: https://doi.org/10.1016/j.disopt.2015.05.001. URL `https://www.sciencedirect.com/science/article/pii/S1572528615000213`.

D. Adolphson and T. C. Hu. Optimal linear ordering. *SIAM Journal on Applied Mathematics*, 25(3):403–423, 1973. doi: 10.1137/0125042. URL `https://doi.org/10.1137/0125042`.