Coupled task scheduling

Anna Markó
Supervisor: Györgyi Péter

2023. május 15.

1 Introduction

During the semester, I continued to study the version of coupled task scheduling problem where the value of L is constant, focusing on the objective function $\sum C_j$. My goal was to handle inputs consisting of as many jobs as possible. I based my work on the algorithm published last year by David Fisher and Péter Györgyi[**1**], and then attempted to improve it using various methods. To test these improvements, I used Python.

The algorithm first sorts the jobs in non-decreasing order of $a_j + b_j$, and then greedily schedules them in this order.

Algorithm 1:

Input : $(a_j, b_j) \quad j = 1 \dots n$, L
Output: $(s_j)_{j=1}^{n} \quad j = 1 \dots n$

Sort the jobs in non-decreasing order of $a_j + b_j$;
$s_1 := 0$;
for $j = 2 \dots n$ **do**
    if a_j *can be scheduled immediately after* a_{j-1} *without overlapping into the processing time of other tasks* **then**
        Schedule it this way;
        $s_j := s_{j-1} + a_{j-1}$
    else
        if b_j *can be scheduled immediately after* b_{j-1} *without overlapping into the processing time of other tasks* **then**
            Schedule it this way;
            $s_j := s_{j-1} + a_{j-1} + b_{j-1} - a_j$
        else
            Start a_j immediately after b_{j-1};
            $s_j := s_{j-1} + a_{j-1} + b_{j-1} + L$;

The algorithm can be reliably characterized as a 3-approximation. However, based on my observations from the previous semester, it performs significantly better in

practice. Unfortunately, I was only able to test it on inputs consisting of a maximum of 10 jobs due to limitations of the IP solver. The approximation algorithm swiftly schedules even large inputs, allowing me to compare the improved results with those generated by the algorithm itself.

2 Improvement by rearranging blocks

In a given schedule, blocks are formed by jobs whose starting times immediately follow each other (there are no task outside the block), and for any two directly following jobs, the later scheduled job starts before the earlier one would have finished. A scheduling arranges jobs into blocks. I examined the optimal scheduling of given blocks.

Each block can be described by three characteristics. Let B_j^n denote the number of jobs in the jth block, B_j^l represents the length of the block (the time elapsed from the start of the first job in the block to the completion of the last job in the block), and B_j^w indicates the sum of elapsed times between the completion of the jobs within the block and the start of the block.

Lemma 1. *Let $B = (B_1, \ldots, B_m)$ denote a set of blocks. Schedule them in non-decreasing order of $\frac{B_j^n}{B_j^l}$. This schedule is optimal.*

Proof. Suppose there exists an optimal block sequence where B_i is scheduled before B_j, and $\frac{B_j^n}{B_j^l} > \frac{B_i^n}{B_i^l}$. Then, there exist two adjacent blocks B_i' and B_j' such that B_i' is scheduled before B_j', and $\frac{B_{j'}^n}{B_{j'}^l} > \frac{B_{i'}^n}{B_{i'}^l}$. Let's exchange these two blocks and observe how the objective function changes. The completion times of the jobs in the other blocks remain unchanged, so we only need to consider the changes in the completion times of the jobs in $B_{i'}$ and $B_{j'}$. In the original order, this is $B_{i'}^w + B_{i'}^l B_{j'}^n + B_{j'}^w$, while after the swap, it becomes $B_{i'}^w + B_{j'}^l B_{i'}^n + B_{j'}^w$.

$$\frac{B_{j'}^n}{B_{j'}^l} > \frac{B_{i'}^n}{B_{i'}^l} \implies B_{i'}^w + B_{i'}^l B_{j'}^n + B_{j'}^w > B_{i'}^w + B_{j'}^l B_{i'}^n + B_{j'}^w$$

This contradicts the optimality of the order. □

Let's assume there are k blocks in a schedule. The objective function can be described using the characteristics of the blocks:

$$\sum_{i=1}^{k} C_j = \sum_{i=1}^{k} \left(B_i^w + B_i^n \sum_{j=1}^{i-1} B_j^l \right).$$

The value of $\sum_{i=1}^{k} B_i^w$ is constant. The scheduling of the blocks can be interpreted as the $1||w_j S_j$ problem, where $S_j = \sum_{i=1}^{j-1} B_j^l$ and $w_j = B_j^n$ for all j. Therefore, the start times of the blocks are weighted by the number of jobs in the blocks.

This method alone did not prove to be useful. I tested it on inputs consisting of 100 jobs with different values of L, but unfortunately, it did not improve by even half a percent on average. Nevertheless, I continued to use it in further analysis, as it runs in linear time with respect to the number of jobs.

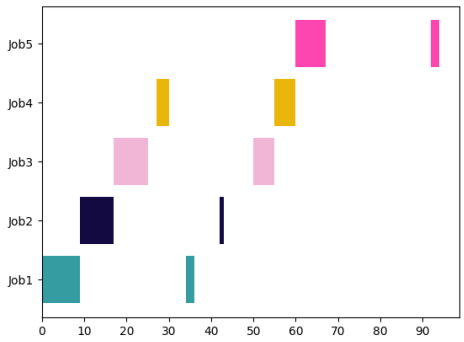
The result provided by the algorithm is denoted as C_{alg}, while the result improved by rearranging blocks is denoted by C_{bl}.

L	50	40	30	20	10
C_{bl}/C_{alg}	0.9984	0.9975	0.9964	0.9957	0.9951

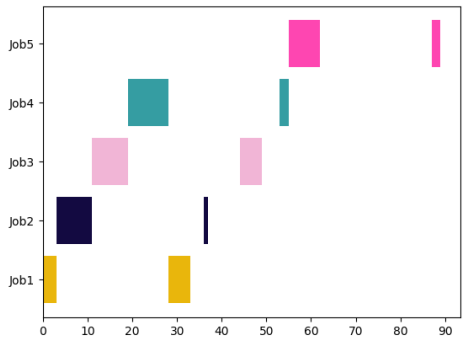
Table 1

3 Local search

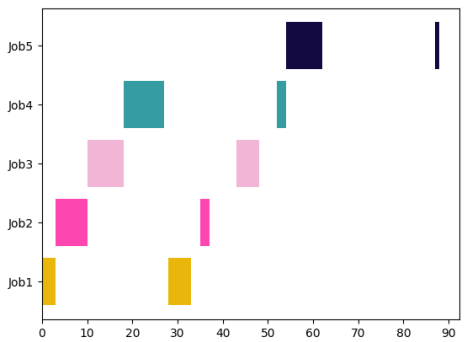
The local search is a heuristic method frequently utilized for solving NP-hard problems. The algorithm traverses the solution space according to specific operators until it reaches a local minimum or reaches the maximum iteration count. I introduced two simple operators: the interchange and the shift operator. The former moves into a new solution by swapping two jobs in the scheduling sequence, while the latter shifts a job within the sequence.



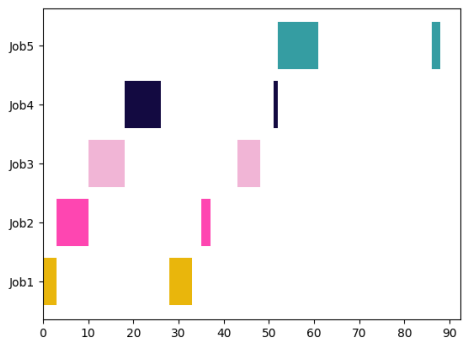
(a) Initial schedule



(b) 1. step

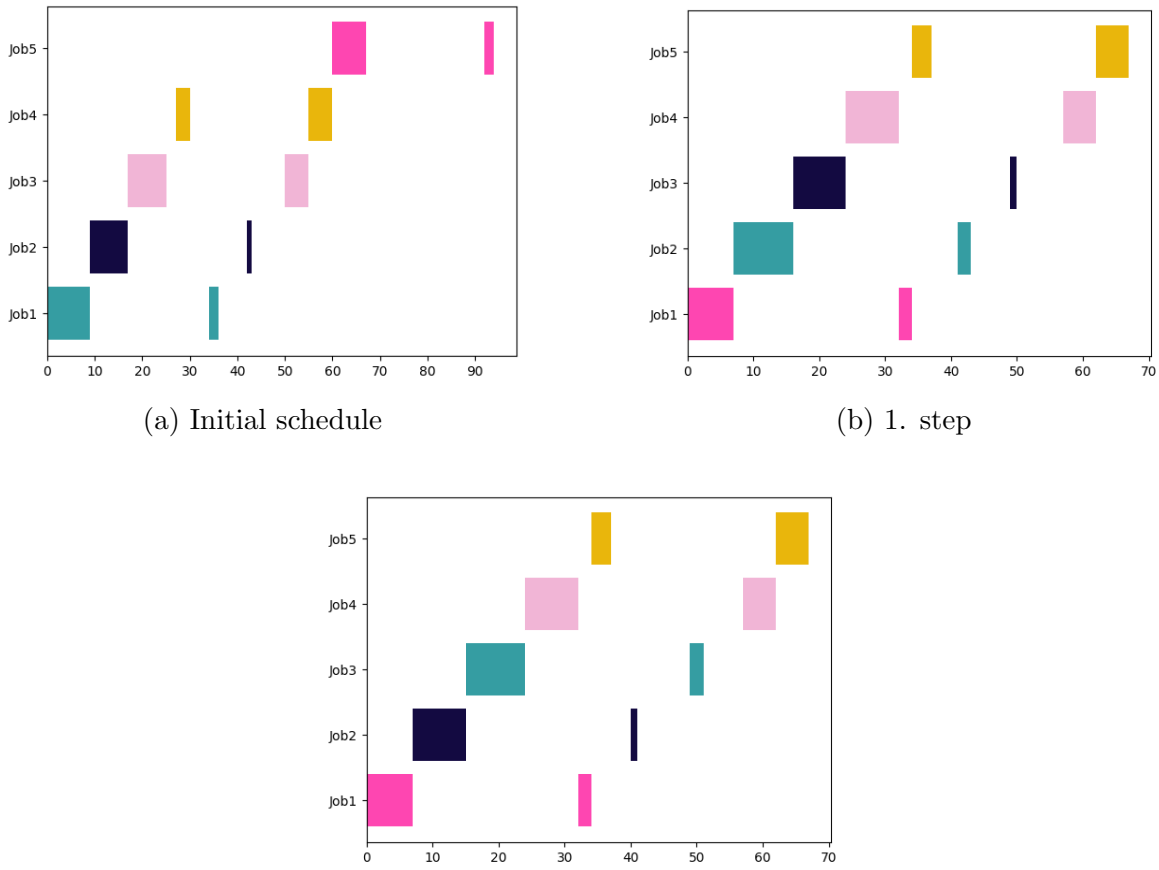


(c) 2. step



(d) 3. step

Figure 1: Local search with interchange operator



(a) Initial schedule

(b) 1. step

(c) 2. step

Figure 2: Local search with shift operator

In the first case, I used only one operator during each run. The operators were operated according to the previous figures. In one step, the operator performed the sequence rearrangement, and then evaluated the given sequence using the greedy scheduling employed in the approximation algorithm. I tested the two operators on 20 different inputs consisting of 50 jobs each. The values of a_j and b_j were randomly generated between 1 and 10, while L was set to 50. Initially, the values were sorted in non-decreasing order of $a_j + b_j$. The interchange operator performed slightly better, although not significantly. I denoted the results obtained using the shift operator as C_{shift}, and those obtained using the interchange operator as C_{ich}.

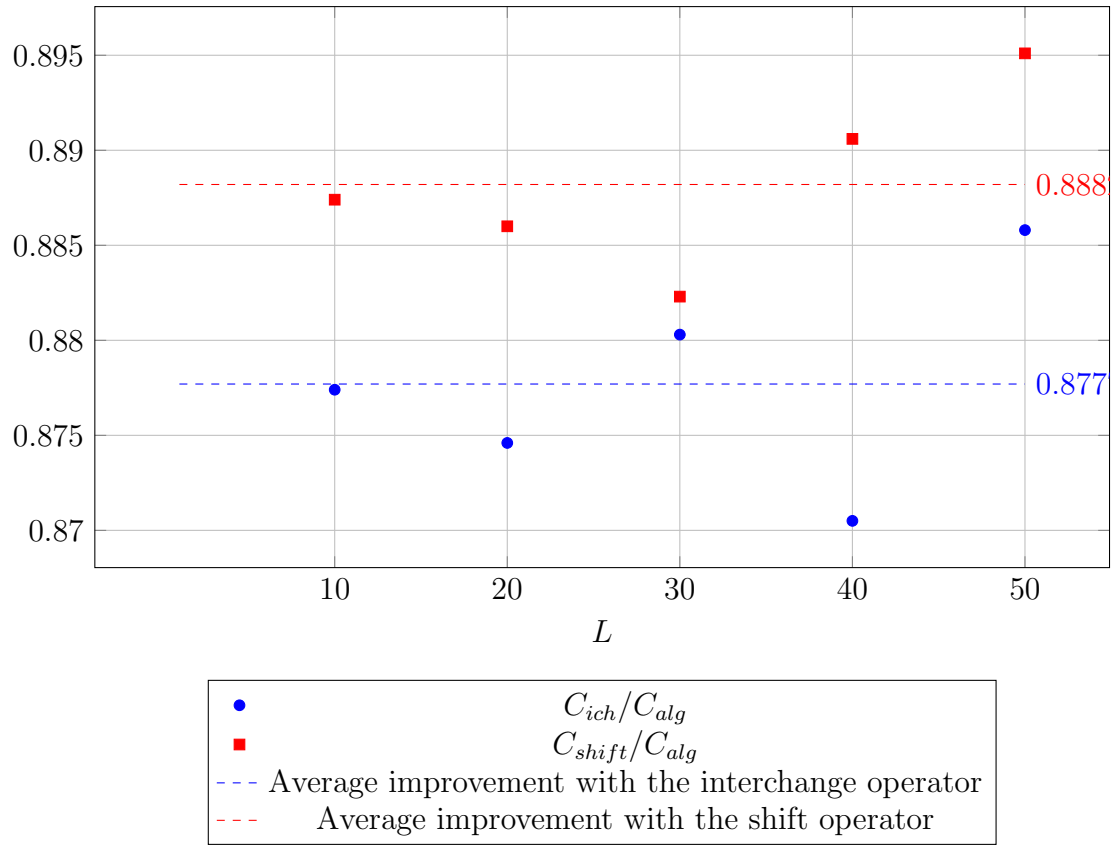


Figure 3: Improvement using local search with a single operator at a time

In the next case, I slightly modified the operation of the operators. In one step, they made a swap/shift in the sequence, evaluated it greedily, and then arranged the blocks in the optimal order. This resulted in a much larger change in the sequence of jobs in one step compared to the previous method. I tested the two operators on the same inputs as before. Here, too, I found that the interchange operator performed better. Overall, this method brought a slight improvement compared to the previous one. I denoted the result provided by the modified interchange operator as $C_{ich'}$ and the result provided by the modified shift operator as $C_{shift'}$.

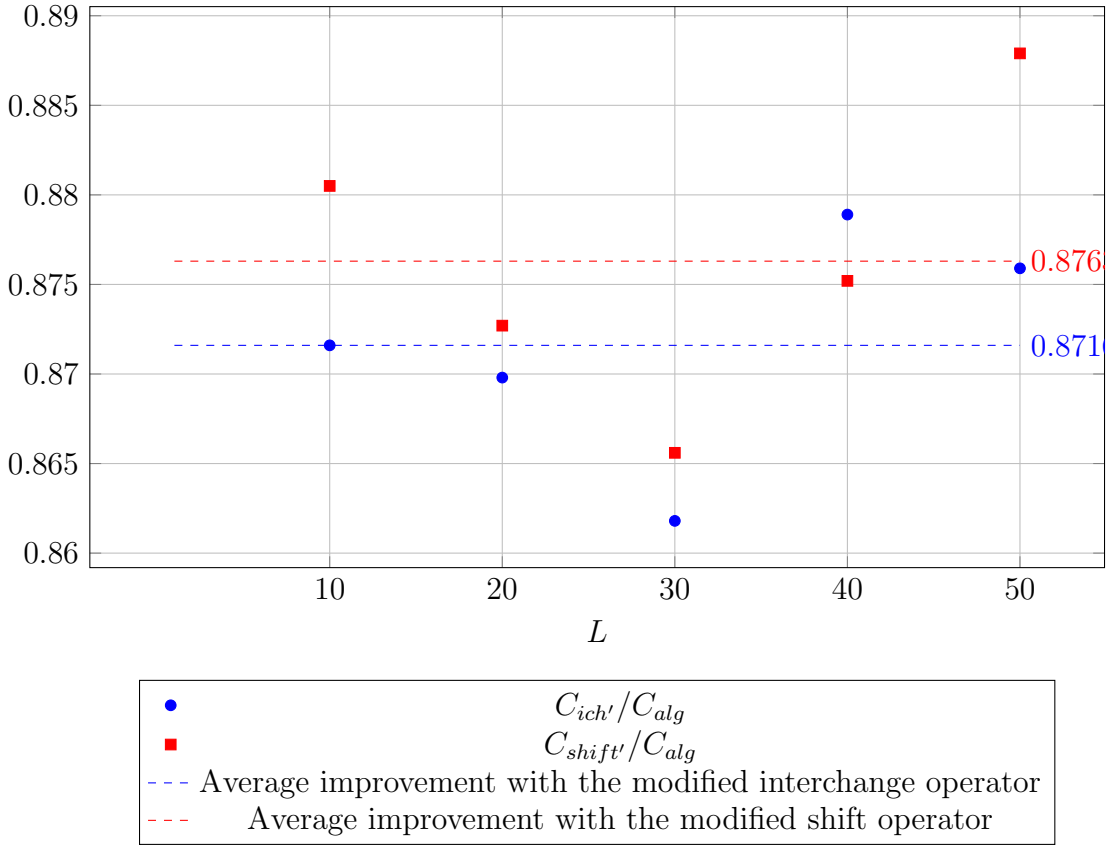


Figure 4: Improvement using local search with a single modified operator at a time

A characteristic improvement to local search is to alternate between using the operators. We continue improving with one operator until reaching a local minimum, then switch to the other operator. This process continues until neither operator can make further improvements.

Algorithm 2: Local search with alternating use of operators

Input : $(a_j, b_j) \quad j = 1 \dots n$, L
Output: $(s_j)_{j=1}^{n} \quad j = 1 \dots n$

```
while Schedule can be improved do
    while The interchange operator improves the schedule do
        Apply the interchange operator;
    while The shift operator improves the schedule do
        Apply the shift operator;
return best_solution
```

Since the modified operators performed better individually, I continued to use them in this method as well.

The performance of local search heavily depends on how the jobs are ordered in the input. I tested my program on the same tasks with different job orders. Generally, the approximation algorithm performs much better than local search starting from a

random order. Improvement was only observed for those orders where the input was sorted in non-decreasing order by $a_j + b_j$ or b_j, with $a_j + b_j$ proving to be more effective.

L	50	40	30	20	10
C	0.865	0.868	0.861	0.867	0.850

Table 2

I attempted to further improve the local search using tabu search. The essence of tabu search is that if no improving move is available, we allow a worsening move. However, this rule could easily lead us back to the same solution. To avoid this, we record a predefined number of the most recently evaluated states.

Algorithm 3: Local search with tabu search

Input : $(a_j, b_j) \quad j = 1 \dots n$, L, tabu_size, max_iter
Output: $(s_j)_{j=1}^{n} \quad j = 1 \dots n$

```
1  Initialize current solution, task list, and order;
2  Initialize tabu list as an empty queue with a maximum size of tabu_size;
3  iter_count ← 0;
4  while iter_count < max_iter do
5      iter_count ← iter_count + 1;
6      Initialize best solution, task list, and order to current solution, task list,
         and order;
7      for operator ∈ {interchange, shift} do
8          Apply operator to obtain new solution, task list, and order;
9          if improvement found and move not in tabu list then
10             Update best solution, task list, and order;
11     if improvement found then
12         Update current solution, task list, and order with best move;
13         Add best move to tabu list;
14         if tabu list exceeds tabu_size then
15             Remove the oldest move from tabu list;
16     else
17         Apply a random operator to get a new solution, task list, and order;
18         Add the new move to tabu list;
19         if tabu list exceeds tabu_size then
20             Remove the oldest move from tabu list;
21 return best_solution
```

In the case of tabu search, I also employed the modified operators. Unfortunately, compared to the previous results, I observed only very negligible improvement, typically around 0.1-0.2 percent, while the execution time significantly increased. Thus,

for now, local search with alternating use of operators proved to be the most effective approach.

My assumption is that further improvement would require a new evaluation algorithm. The greedy algorithm fails to handle cases where it's beneficial to delay a job in order to schedule another job efficiently within the same block. Therefore, the next goal is to find new evaluation algorithms that can schedule jobs once the order is given.

References

[**1**] D. Fischer, P. Györgyi: *Approximation algorithms for coupled task scheduling minimizing the sum of completion times* Ann. Oper. Res. 328(2): 1387-1408 (2023)