Scheduling and Resource Allocation Coursework

Jan Marczak
Rudolfs Grobins

November 17, 2022

Question 1

The processing times obtained with the gettimes.py script in Azure environment are displayed in Table 1.

Question 2

- We handled precedence constraints by considering the precedence matrix G. For a given partial schedule we loop over each already scheduled job j. Then we check each job i that points to j (elements of the j-th column of G). If all jobs that i points to (i-th row of matrix G) are in the partial schedule, time ordering is satisfied and we can add i to the list of candidate jobs to be scheduled next.

  We generate a full solution from a partial schedule by scheduling jobs in decreasing order of due dates, that is, largest due date to be scheduled last in the final schedule like in earliest due date (EDD) method.

- Table 2 shows the current node (its partial schedule) with its total tardiness (lower bound) in the first and the last 2 iterations of our branch and bound (BnB) algorithm [1].

- The best schedule found by our algorithm in 30 000 iterations has total tardiness of 381.08 and schedules jobs in the following order:

  30, 4, 3, 23, 22, 21, 10, 9, 8, 7, 6, 20, 19, 18, 17, 16, 14, 27, 29, 28, 26, 25, 24, 15, 13, 11, 12, 5, 2, 1, 31

  This schedule is not globally optimal. If we let branch and bound terminate on its own, it finds a globally optimal solution with a total tardiness of 352 in 40873 iterations.

- The code takes around 0.4 seconds to run on our machine (Apple M1). Largest list of pending nodes is 26167, however, this may include nodes that can be fathomed as we don't actively prune this list once we've identified a full schedule.

- Hu's algorithm produces the following schedule[2]:

  30, 4, 10, 23, 3, 9, 20, 22, 8, 19, 21, 7, 18, 29, 6, 17, 27, 28, 14, 16, 26, 13, 15, 25, 11, 12, 24, 5, 2, 1, 31

  On Azure, we obtained a total tardiness of 366.2614 ± 20.3944 s for our branch and bound method and 474.1927 ± 6.9760 s for Hu's.

Question 3

Whilst considering improvements to BnB, we examined the total tardiness while varying the maximum number of iterations as shown in Figure 1. We observed two facts: (1) "vanilla" BnB obtains a full schedule only when number of iterations is significantly beyond 30,000 (2) there is no reduction in total tardiness until approx. 25,000 iterations. This indicates that due to the breadth-first nature of BnB, we have to rely on our partial schedule completion heuristic but the EDD heuristic does not appear to produce good results when number of iterations is low and partial schedules are short. Therefore, we began by focusing on depth-first approaches.

[1] Our BnB algorithm breaks ties arbitrarily
[2] We break ties in Hu's method so that smaller indices come first in the *final* schedule

1. Node Selection Policy

Depth-First Search

At first we examined a depth-first search, by expanding the longest partial schedule at each iteration. DFS manages to outperform vanilla BnB by quickly producing full solutions, and therefore establishing upper bounds on total tardiness. We make use of this upper bound to fathom unnecessary non-optimal paths. As shown in Table 3 we see a significant improvement in both the number of iterations and a much shorter pending nodes list.

These promising results led us towards implementing a more guided search by using both a heuristic and the lower tardiness bound.

Projected Cost

The projected cost search can be thought of as a non-admissible A* search, with the evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost from the start node to the current node (lower tardiness bound) and $h(n)$ is the estimated cost from current node to the goal. This $h(n)$ estimation is done by taking the average cost of all nodes in the partial schedule and multiplying by the number of nodes needed to achieve a full schedule, i.e.

$$h(n) = \frac{g(n)}{len(n)} \times len(N) \text{ , where } N \text{ and } n \text{ are full schedule and current partial schedule respectively.}$$

As shown in Table 3, this method achieves the best overall performance for our workflow example. It displays both the space saving properties (smallest max pending nodes) and the fastest convergence to the optimal solution.

Although the results appear to be satisfactory, this node selection policy has its drawbacks. With this approach we assume the current cost per node is a good estimate for the remaining schedule. In schedules where bulk of the total tardiness arises at the end of the schedule $h(n)$ is likely to provide an overestimation of our actual cost function. This can lead the search space to undesirable branches. Whilst, this is not the case in our specific example we opted for an approach that generalises better.

2. Dominance

While the methods above produced good results for this problem, they might not be exploratory enough in general and do not offer improved performance guarantees. Here we consider fathoming nodes via a dominance relation.

Consider two partial schedules found by branch and bound, P and P' with total tardiness lower bounds T_P and $T_{P'}$. In this setup BnB starts from the end of the final schedule, so a full schedule is represented by $S = [R, P]$ where R is the remaining optimal schedule yet to be completed by BnB. Let's further assume $T_{P'} > T_P$ and $P' \subseteq P$, i.e. P contains *at least* all jobs of P' but possibly in a different order.

The total cost of a final schedule $T_S = T_R + T_P$. Since $P' \subseteq P$, it means $R \subseteq R'$ and thus $T_R \leq T_{R'}$ as both remaining schedules are optimal and R' is at least as long as R. Finally we can deduce $T_S - T_{S'} = (T_R - T_{R'}) + (T_P - T_{P'}) < 0$, meaning partial schedule P dominates P' and we can fathom the node representing P'.

We implement the dominance check during jumptracking: when we jump to a new node, we first check whether it's dominated. If not, then we branch, otherwise fathom it and repeat jumptracking.

There is a significant performance improvement with this method: it explores the whole search space and attains the optimal solution with 158 iterations and with 92 nodes fathomed via domination.

Of course, this method doesn't hold in general. For example, if we also penalised earliness then we would not be able to make strong assumptions about cost of R and R'.

Table 3 contrasts the different approaches. Overall the dominance relation appears to be the strongest solution as it provides optimality guarantees while requiring few iterations.

Appendix

Node type	Mean Time [s]	Standard Deviation [s]
vii	18.8609	0.7362
blur	5.5183	0.1368
night	22.4894	0.7375
onnx	2.8342	0.1242
emboss	1.7477	0.6833
muse	14.4640	0.5424
wave	10.0377	0.7100

Table 1: Processing times obtained with the gettimes.py script in the Azure virtual environment

Iteration Nr	Current Node	Total Tardiness Lower Bound
1	[31]	0
2	[1, 31]	31.4243
29999	[29, 15, 14, 27, 28, 26, 25, 13, 11, 12, 24, 5, 2, 1, 31]	333.2023
30000	[14, 13, 27, 11, 28, 26, 12, 25, 24, 5, 2, 1, 31]	333.2023

Table 2: Current node (with the partial schedule) and its total tardiness (lower bound) in the first 2 and the last 2 iterations of branch and bound using processing times from table 1.

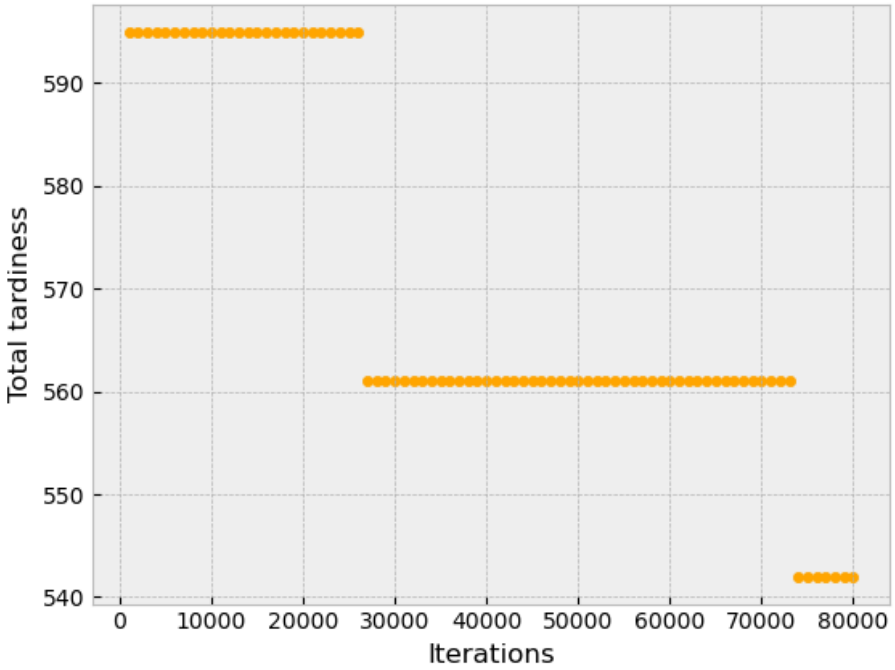

Figure 1: Total tardiness for vanilla branch and bound with varying max iterations limit.

Method	Nr of Iterations to Optimal Solution	Runtime	Max Pending Nodes
Vanilla BnB	73947	0.9	114399
Depth-First Search	3889	0.6	55
Projected Cost	88	0.7	45
Dominance	158	0.4	232

Table 3: Performance comparison of branch and bound methods using processing times from Q3. Runtime and maximum pending nodes refer to running each algorithm until it terminates (i.e. no max iteration limit).