Algorithmic Programming
Project assignment

Two-machine scheduling

The two-machine scheduling problem

– A set of n parts needs to be processed in a workstation that has two parallel machines.

– The first machine is newer and faster than the second machine. However, since a single machine does not provide enough capacity, the older and slower machine is still in use.

– Each part $i \in \{1, \dots, n\}$ has a certain width wi and requires a processing time pik when handled by machine $k \in \{1,2\}$.

The two-machine scheduling problem

- To be able to process a part, the setting of the machine must be adjusted corresponding to the part’s width.

- The time required to adjust the machine setting is proportional to the difference in width between the consecutive parts.

- A schedule needs to be found that allocates each part to one of the machines and that sequences the parts for each machine such that the makespan to finish all the parts is minimized.

Two-machine scheduling: example

- Input: 4 parts

$w_1 = 200$mm, $p_{11} = 100$s, $p_{12} = 150$s.
$w_2 = 250$mm, $p_{21} = 120$s, $p_{22} = 180$s.
$w_3 = 180$mm, $p_{31} = 110$s, $p_{32} = 140$s.
$w_4 = 240$mm, $p_{41} = 140$s, $p_{42} = 180$s.
Machine 1 initially set at $w_{01} = 270$mm, machine 2 at $w_{02} =$ 250mm.
Machine adjustment rate: $a_1 = a_2 = 1$ mm/s

- Possible solution:

Machine 1: first part 2, then part 4 ➔ 290s
Machine 2: first part 1, then part 3 ➔ 360s
Overall makespan: 360s

Class ‘Instance’

Data:

- Part data: list of (w, p1, p2) tuples
- Machine data: list of (w0, a) tuples

Functions:

- setup(m, i, j): return the time required for machine m to adjust from part i to part j

You can define additional data structures and functions as much as you need, just make sure the code is efficient and elegant.

Class ‘Solution’

Data:

– Representation of a solution

Functions:

– eval(): evaluation function to check feasibility of a solution and calculate its makespan

You can define additional data structures and functions as much as you need, just make sure the code is efficient and elegant.

Exact solution methods

Solution methods to be implemented:

-enumerate():
brute-force method – enumerate all possible solutions

-LP_model():
Write a mathematical model and solve using a MIP solver (e.g., Gurobi)

-DP():
dynamic programming – define the problem recursively and solve with DP

Heuristic solution methods

-Greedy construction heuristics:

- Build a solution by inserting parts into the schedule one by one.

-Implement (at least) three versions of a greedy construction heuristic, i.e., using three different selection rules for choosing the next part to be inserted.

Heuristic solution methods

Local-search improvement heuristics:

-Improve a solution by making a local change to it.

-By making a small, ‘local’ change, any given solution can be changed into a different, so-called ‘neighbor’ solution

- A local-search heuristic evaluates neighbors of the current solution and ‘jumps’ to a neighbor if that leads to an improvement.

-Local-search finishes when no more improvements are found. Then the current solution is optimal in its neighborhood

- Implement (at least) three local-search improvement heuristics, i.e., define three neighborhoods.

Heuristic solution methods

Local-search neighborhoods:

– move():
  - take a part out of the solution and reinsert it in a different position

– swap():
  -swap the position of two parts in the solution

– Come up with at least one other local-search operators.

– Local-search can be implemented as ‘first-accept’ or ‘best-accept’:
  -First-accept immediately jumps to the first improving neighbor you encounter
  -Best-accept checks all the neighbors and jumps to the best neighbor

Performance evaluation

– Evaluate the solution quality and runtime by running your exact methods and heuristics on a large set of randomly generated instances with varying sizes.