POLITECNICO DI TORINO

01QWTBH – Operational Reasearch

MINIMUM COST FLOW PROBLEM – BHS

Authors:

Alessandro La Ciura s322200

Simone Terranova 331343

Marco Cutarelli 331196

A.Y. 2023/24

1. Problem Statement and definition

In the aeronautics industry, the optimal management of baggage is a critical challenge due to the increasing volume of air traffic and the corresponding rise in baggage handling demands. The primary objective of this project is to develop a Baggage Handling System (BHS) that minimizes the energy consumption required to efficiently transport luggage to designated collection points. Given a system of conveyor belts accessible through specific entry points, baggage is sent across the network by evaluating all possible routes, selecting the one that ensures delivery to the destination while accounting for the network's traffic conditions. Each conveyor belt has limitations in terms of capacity and supported weight, and these belts vary in size and energy consumption depending on the type employed. In this project, the model is designed to choose the conveyor belt for each luggage that results in the lowest energy consumption for each hop, while adhering to the constraints related to the belt's structure and congestion. During the model development process, certain assumptions were made to simplify the system's complexity, such as assigning a single corresponding exit to each entry point and reducing the internal network structure to a single series of intermediate nodes.

1.1 Instance Generation

The Python scripts *graph_gen.py* and *instance_gen.py* work together to simulate and analyze a baggage transport system modeled as a tripartite graph. This graph represents the flow of baggage from sources to destinations via intermediate nodes, with edges functioning as conveyor belts. It is important to notice that in this model intermediate nodes are not seen as collection points but just as switching points where baggage is correctly directed. For this reason, capacity values are referred to edges, meaning that on that conveyor belt there are currently a given amount of baggage passing through the edge output node and they cannot overtake the assigned capacity value.

The *instance_gen.py* script supports the graph generation process by defining the *Conveyor_belt* and *Baggage* classes, which model the system's behavior in greater detail. The *Conveyor_belt* class represents the conveyor belts in the network, tracking their capacity, weight limits, and power consumption. Each of them has attributes like *max_capacity*, *max_weight* and *power*, which influence how many items it can carries and how much energy it consumes. The *power_consumption_computation* method calculates the belt's energy usage, factoring in the weight of the baggage. Overall, the *max_capacity* value affects both weight and power coefficient of the conveyor belt, since it is reasonable to think that higher capacity means larger length and so more weight supported and more energy required.

The *Baggage* class defines individual instances of baggage with attributes such as weight, starting point, and destination. This baggage flows through the network, interacting with the conveyor belts and influencing their operational statistics.

Together, these two classes allow for detailed modeling of baggage transport, tracking the movement of each item and computing the system's power consumption.

As for *graph_gen.py* script, it focuses on generating and manipulating a tripartite graph using the *NetworkX* library. It defines sources (baggage origins), intermediates (transit points), and destinations (baggage endpoints). At each source is assigned a random supply of baggage, matched by a demand at the corresponding destination.

The function *create_tripartite_graph_nodes* creates nodes with attributes such as demand and baggage lists. More in detail, intermediate nodes have relevant edge properties related to output links connecting intermediate nodes with destination nodes. This, because in *MCF_heu.py* these properties are exploited to make some checks about output edge weights and capacities. These attributes are important for defining the flow and tracking the state of the system. Once the nodes are generated, the *generate_edges* function establishes directed edges between nodes. These edges represent conveyor belts and are assigned random capacities, weights, and power coefficients, which define the belt's ability to transport baggage.

Finally, the plot_graph function visualizes the generated graph, highlighting node supply, demand, and edge weights and capacities.

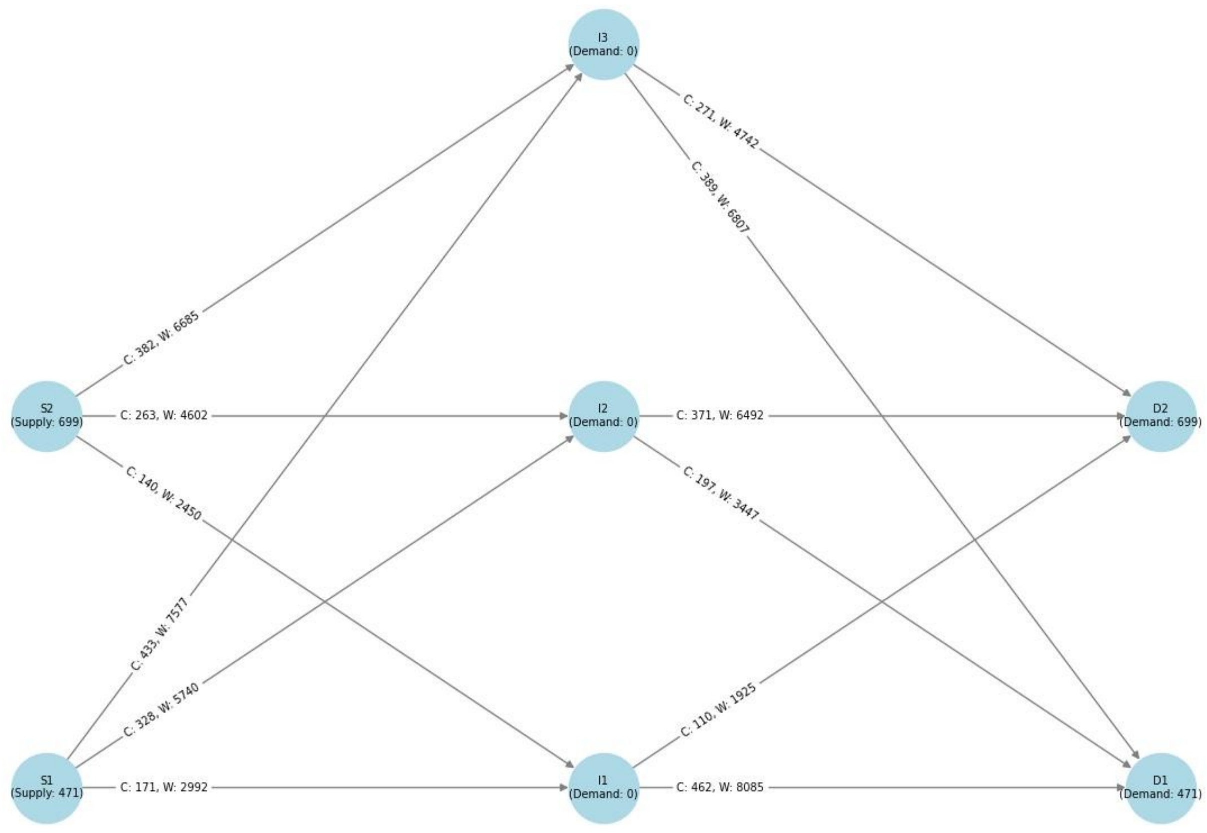


Figure 1:

Easy example of graph representing a network with two source nodes, three intermediate nodes and two destination nodes, each of them with their respective supply/demand. Moreover, each edge has specific weight and capacity values.

2. Heuristics

2.1 Heuristic references

- Min Cost Flow Heuristic:

The first heuristic algorithm is based on the concept of Min Cost Flow. This algorithm deals with finding the flow of material, such as luggage or goods, through a transportation network while minimizing the total costs. The network is represented as a graph with nodes (access points, intermediate points, and destinations) and arcs (conveyor belts). Each arc has a cost associated with transporting the flow and a maximum capacity and weights.

MCF_heu.py use a simplified version of this problem, where the central idea of the algorithm is to route flow (in this case baggage) through a network such that the overall cost (in this case, energy consumption) is minimized, while respecting constraints (edge capacity and weight limits).

- Local Neighbor Search Heuristic.

LNS_heu.py is based on Local Neighbor Search, a heuristic method widely used to solve optimization problems. In this case, the algorithm starts with an initial solution and tries to iteratively improve it by exploring neighboring solutions.

The procedure can be described as follows:

1. It starts with an initial solution, that is a feasible assignment of paths to the luggage in the network.
2. The algorithm explores neighboring solutions (e.g., by changing the path of one or more luggage) and evaluates whether these new solutions offer an improvement over the current one, in terms of lower energy consumption.
3. If a better solution is found, the algorithm adopts it and repeats the process. Otherwise, it continues to explore other nearby solutions.
4. The process repeats until a stopping criterion is reached, such as a maximum number of iterations or negligible improvement.

These two heuristic approaches offer a balance between solution quality and computational efficiency, the former being more structured and the latter more flexible and adaptive.

2.2 MCF heuristic

The file *MCF_heu.py* contains a heuristic approach whose goal is to optimize the energy consumption of conveyor belts as baggage moves from source nodes to their destinations, while also avoiding system congestion and ensuring the stability of the entire network. The code is structured around two key functions: *min_cost_computation and no_selected_edge_computation*.

- *min_cost_computation* function:

The core function of this code determines the optimal path for each baggage based on minimizing energy consumption across conveyor belts. The approach divides nodes into three categories: input, intermediate, and output nodes. The path computation occurs in two phases: first, from input nodes to intermediate nodes, and then from intermediate nodes to destination nodes. For each baggage, the algorithm evaluates all possible conveyor belts that it could traverse to move from its current node to the next. This process involves:

1. Checking the current capacity and weight limits of each conveyor belt.

2. Calculating the potential energy consumption if the baggage were to be sent through that belt.

3. Storing viable options in a dictionary (*possible_edge*) that tracks the total energy consumption for each potential path.

The function also includes a mechanism for handling congestion. It assigns almost full conveyor belts through a secondary dictionary (*secure_possible_edge*) and prioritizes these edges only if no better options are available. This allows the system to avoid bottlenecks while still functioning efficiently.

The function uses a First In, First Out (FIFO) policy for managing baggage at each node, ensuring that baggage is processed in the order it arrives. Once the best edge is selected for a baggage item, the system updates the baggage's path, the conveyor belt's statistics, and the node's baggage list. If no feasible path is found due to constraints on capacity or weight, the function returns an infeasible solution, halting the process.

In the second phase of the algorithm, baggage is moved from intermediate nodes to their destination. Here, only one edge exists for each intermediate-destination pair, simplifying the decision process.

- *no_selected_edge_computation* function:

This function is invoked when a conveyor belt (represented as an edge) is evaluated for carrying baggage but is ultimately not selected for that baggage's path. The purpose of the function is to revert any temporary updates to the belt's statistics, including total weight, total power consumption, and available capacity, to their previous state. This ensures that the condition of the conveyor belt remains consistent for other evaluations. The function accepts the edge ID, the list of all edges in the graph, and the baggage being evaluated as input. It is crucial to maintaining the integrity of the system during path evaluations, ensuring that the system's state is only updated for selected edges.

2.3 LNS heurisitc

The *LNS_heu.py* file implements a Large Neighborhood Search (LNS) heuristic algorithm whose main goal is to start with a feasible, though not necessarily optimal, solution and improve it through iterative exploration of nearby solutions. Also in this case, the key objective is to minimize the overall power consumption across the system.

- *first_feasible_solution_generator* function:

The purpose of this function is to generate a starting feasible solution that satisfies the capacity and weight constraints of the conveyor belts without necessarily focusing on optimality. This initial solution acts as a baseline for further improvement in the local search phase.

As for the previous approach, the function begins by organizing nodes into three categories: input, intermediate, and output nodes. It then iterates over the baggage at each input node, attempting to assign a path to each baggage item by sending it through available edges to intermediate nodes. However, with respect to *MCF_heu.py ,*the process of assigning baggage to intermediate nodes is performed assigning for each luggage item a path based on whether the conveyor belt's capacity and weight constraints are satisfied, without making any preliminary energy consumption consideration. Indeed, once a feasible edge is found, the baggage is moved along that edge, and then its power consumption is computed and stored. If no edge satisfies the constraints, the solution is marked as infeasible, and the function terminates. The same process is repeated for moving baggage from intermediate nodes to destination nodes. More in detail, following this approach, the algorithm would select always the first valid edge until it reaches its full capacity. For this reason, the selection process is randomized by shuffling the list of edges at each iteration, to prevent overloading any single intermediate node, thus improving fairness in the initial solution.

The function outputs a dictionary, *first_baggage_path_guess*, which tracks the path taken by each baggage, including the nodes it passes through, the conveyor belt used, and the associated power consumption. Furthermore, for each baggage path, also the baggage instance is appended to the list of information, since it is needed to make some computations exploiting its attributes in the *local_search_with_neighborhood* function.

- *calculate_total_consumption* function:

This function calculates the total power consumption of the current solution. It sums the power consumption of all baggage items as they move through their assigned paths. This total consumption serves as the objective function that the algorithm seeks to minimize during the local search process.

- *local_search_with_neighborhood* function:

The core function of the heuristic performs an iterative local search to explore and improve upon the initial feasible solution.

The process begins by generating the initial feasible solution using the *first_feasible_solution_generator* function. If no feasible solution is found, the function terminates. Otherwise, the current solution becomes the starting point for the local search. The total power consumption of this solution is computed, and it is treated as the best-known solution.

In each iteration of the local search, a neighboring solution is generated by modifying the paths of a portion of the baggage items. The function selects a random subset of baggage (one-third of the total) and attempts to reroute their paths by swapping the intermediate nodes through which they travel. For each baggage item in this subset, the algorithm looks for a new edge that satisfies the constraints of capacity and weight and leads to a different intermediate node than the one currently used. After adjusting the intermediate node, the path from the intermediate node to the final destination is also recalculated to ensure the feasibility of the new route.

The new solution is evaluated by recalculating the total power consumption. If this new solution is better than the current best solution, it is accepted as the new best. Otherwise, the original one is maintained. The search continues for a fixed number of iterations, with the goal of progressively improving the solution through small incremental changes.

3. Results

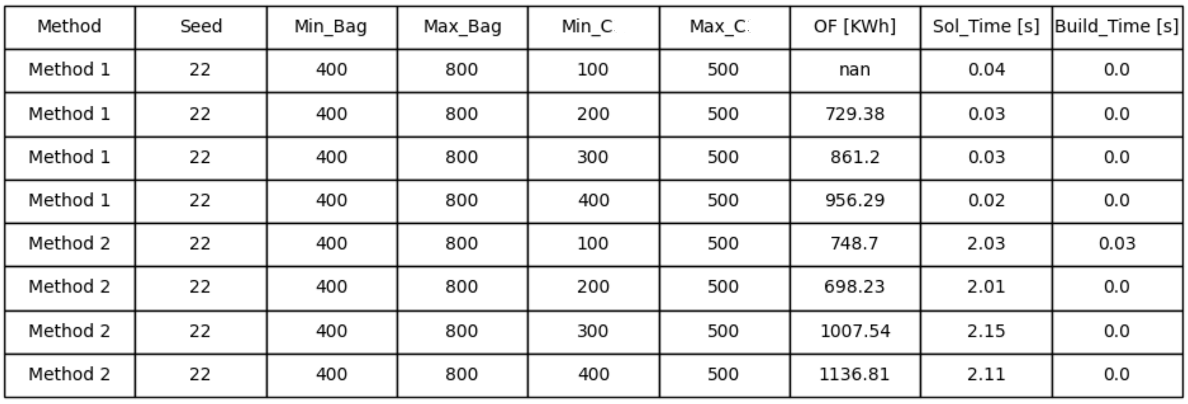
The *main.py* script is designed to compare the two discussed methods for solving the baggage-handling optimization problem. Method 1 and Method 2 are compared in terms of their solution quality and computation times, and the script calculates the optimality gap between them.

Several parameters are defined to control the generation of the graph and the simulation: - *num_sources*, *num_destinations*, and *num_intermediates* define the number of nodes in each category. - *min_c* and *max_c* control the capacity range for the conveyor belts. - *min_b* and *max_b* control the range for the number of baggage items. - *seed* is used for reproducibility. A range is created for the *min_c* parameter to iterate through different capacity values during the simulations.

Method 1 (MCF): As discussed before, the first method simulates the baggage transportation using a minimum-cost flow algorithm. The execution times for both the graph construction (*build_time*) and the solution computation (*sim_time1*) are measured. Once the baggage paths are computed, the total power consumption for each baggage's path is calculated. This power consumption (*of1*) is stored, as the solution for Method 1, in a list called *of_method_1*.

Method 2 (LNS): On the other hand, the second method employs a local search heuristic and like the previous one, it builds the graph and measures and stores the build time, the solution time (*sim_time2*) and the total power consumption (*of2*).

During each iteration on the values of *min_c*, the script compares the solutions from both methods using the optimality gap, which measures the relative difference between the two solutions. If both solutions are feasible (i.e., they return numeric *of* values), the optimality gap is calculated as the percentage difference between them.



Method	Seed	Min_Bag	Max_Bag	Min_C	Max_C	OF [KWh]	Sol_Time [s]	Build_Time [s]
Method 1	22	400	800	100	500	nan	0.04	0.0
Method 1	22	400	800	200	500	729.38	0.03	0.0
Method 1	22	400	800	300	500	861.2	0.03	0.0
Method 1	22	400	800	400	500	956.29	0.02	0.0
Method 2	22	400	800	100	500	748.7	2.03	0.03
Method 2	22	400	800	200	500	698.23	2.01	0.0
Method 2	22	400	800	300	500	1007.54	2.15	0.0
Method 2	22	400	800	400	500	1136.81	2.11	0.0

Table 1

Table 1 shows the objective function values (OF) and the solution times for the two methods across four different capacity settings (*min_c* values: 100, 200, 300, 400) and a fixed range for baggage (*min_b* = 400 and *max_b* = 800). Both methods use a seed value of 22 for results reproducibility on the graph shown in *figure 1*. Furthermore, “nan” OF value is reported if the solution is infeasible, indicating that no valid path was found to transport all baggage, and this depends on graph parameters like number of nodes, edge attributes and the amount of baggage in the system.

The OF values trend reflects that higher conveyor capacities allow for higher total consumption, potentially due to longer paths or more baggage being accommodated. The solution time remains low (around 0.03 seconds) for all cases generated by method 1. On the contrary, method 2 results suggest that the local search heuristic seems more effective at lower capacities but performs worse as capacity increases. The solution times for method 2 are significantly longer, ranging from 2.01 to 2.15 seconds, reflecting the complexity of the local search process.

However, with a more accurate analysis of the algorithms behaviors with small *min_c* step size of 2, it is possible to see that the MCF (method 1) is always better in terms of OF value, that are smaller except for small interval highlighted by green circles in Figure 2. Indeed, the mean value for method 1 is 837.32 kWh, while for method 2 it is 903.66 kWh. It is clear that the previous table scenario, in the first two iterations for both methods, is reporting results associated with the green circles below:

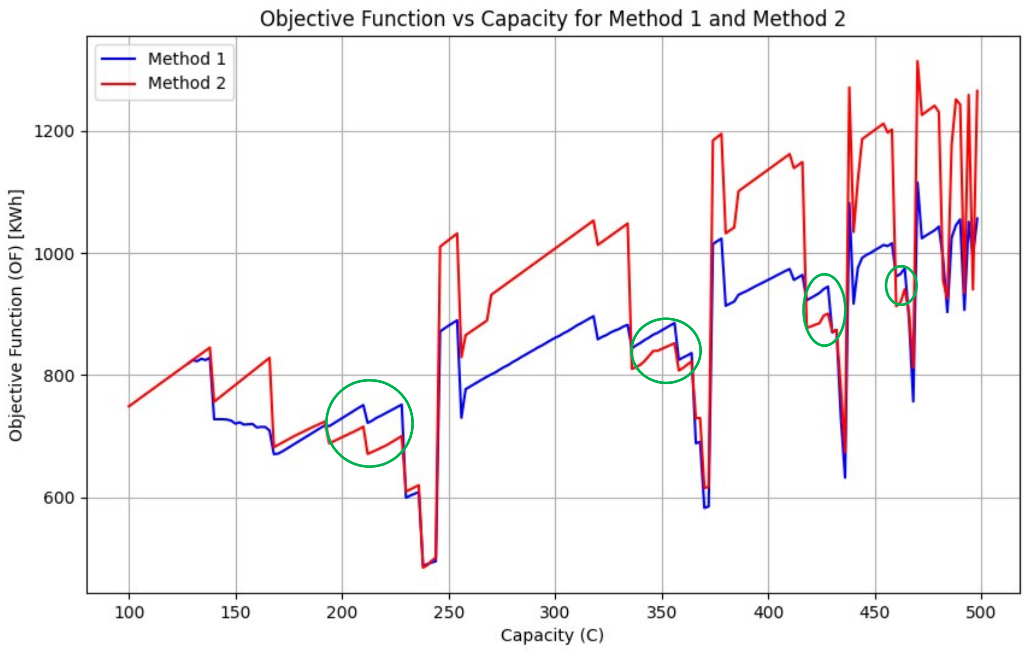
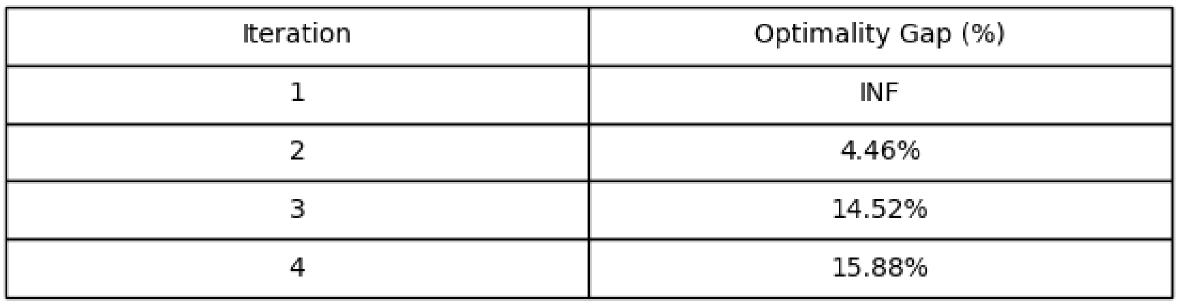


Figure 2:

OF value representation for method 1 and method 2 in function of the capacity range (start = 100, stop = 500, step = 2)

To sum up, the MCF objective function values and simulation times are almost always better than the LNS ones, and it is reasonable considering the iterative approach of the LNS. As confirmed by both *Table 1* and *Figure 2*, the general trend is of larger OF value for larger edge capacity. However, the result in terms of energy consumption is highly variable, probably related to the random assignment of the graph parameters. This last observation can be seen also by the optimality Gap statistics: it is often significant except for some specific capacity value where the OF value tends to be quite similar for both methods, as visible by the convex peaks of *Figure 2*.

As for the previous example of just four capacity iterations, the table below shows the optimality gap between the two methods:



Iteration	Optimality Gap (%)
1	INF
2	4.46%
3	14.52%
4	15.88%

Table 2:

Optimality gap table for Figure 1 graph example, related to Table 1 results