Universidad Autónoma de Nuevo León
Facultad de Ingeniería Mecánica y Eléctrica

19.9/20

Selected Topics on Optimization

Final Report

Computational Experience with Heuristics for the Generalized Assignment Problem

Team A

Hour: Tuesday (M3–M6)

Professor: Roger Zirahuen Ríos Mercado

Members	Program	Student ID
Leonardo Acosta Morales	**ITS**	**2226717**
Esther Garza Cano	**ITS**	**2143077**
Manuel Alexander Fuentes Contreras	**ITS**	**2134970**
Donovan Adrian Serrato Lores	**ITS**	**2154304**

Semester: Jan–Jun 2026

May 15, 2026

SECTION 1: INTRODUCTION

The Generalized Assignment Problem (GAP) is a fundamental model in the field of operations research that seeks to assign a set of jobs to a set of agents as efficiently as possible. Unlike traditional assignment problems, the GAP recognizes that each agent has a limited capacity and that both the resource consumption and the profit obtained vary depending on which job is assigned to which agent (Cattrysse and Van Wassenhove, 1992).

Why It Is Important

The importance of the GAP lies in its structure, which combines characteristics of the multiple knapsack problem and the simple assignment problem. It is a problem classified as NP-hard, which means there is no known polynomial-time algorithm that guarantees an optimal solution for large instances (Martello and Toth, 1990). This makes it an excellent scenario for testing advanced optimization algorithms and mathematical programming techniques (Wolsey, 1998).

Relevance in Optimization

This topic is relevant because it serves as a basis for solving more complex real-world problems. Many logistical situations that appear disorganized can be structured as a GAP. By studying it, the goal is to improve operational efficiency, reduce costs through the smart use of resources, and allow organizations to make decisions based on data and real capacity constraints (Chu and Beasley, 1997).

Practical Applications

In practice, the GAP has applications in multiple sectors:

- **Production Systems:** Assigning jobs to machines that have different processing limits.
- **Logistics:** Route planning where each vehicle has a different load capacity.
- **Telecommunications:** Assigning users to base stations or signal frequencies.
- **Human Resources:** Scheduling shifts and projects for employees with different skills and schedules.

SECTION 2: PROBLEM DESCRIPTION

In this section, we present the formal description of the Generalized Assignment Problem. In any optimization methodology, a problem must clearly specify four fundamental elements: the data known beforehand, the decisions that need to be made, the main objective, and the constraints that govern the system. We will now explain each of these components for the GAP in detail.

2.1 Elements of the Problem

1. Data. The input data consists of two main sets: a set of n jobs (which can be interpreted as tasks or products) and a set of m agents (which represent machines, servers, or processors). The information known beforehand is defined by the following parameters:

- $M = \{1, \ldots, m\}$: set of available agents.
- $N = \{1, \ldots, n\}$: set of jobs to be assigned.
- p_{ij}: profit obtained if job j is assigned to agent i.
- w_{ij}: resource consumption of job j when assigned to agent i. The weight may vary depending on the agent.
- c_i: maximum available capacity of agent i.

2. Decisions. The core decisions of the model involve determining to which agent each job is assigned. Since a job cannot be split, binary decision variables are used:

- $x_{ij} = 1$ if job j is assigned to agent i.
- $x_{ij} = 0$ if job j is NOT assigned to agent i.

By solving the problem, the algorithm provides a binary matrix indicating the final assignment plan (Cattrysse and Van Wassenhove, 1992).

3. Optimization. The objective function reflects the main purpose of the problem. In this standard version of the GAP, the goal is to maximize the total profit assigned across all agents. It is worth mentioning that minimizing cost is an equivalent variant of the problem, but we focus on the profit-maximization approach (Chu and Beasley, 1997).

4. Constraints. These are the strict rules that ensure the solution is practically feasible in the real world.

- **Semi-assignment constraint:** Every job in the set N must be assigned to exactly one agent. Jobs cannot be left unassigned, nor can a job be assigned to multiple agents.
- **Capacity constraint:** The total accumulated weight of all jobs assigned to agent i cannot exceed its predetermined capacity c_i.

2.2 The Mathematical Model

Translating the previously described elements into algebraic expressions, the complete mathematical formulation for the Generalized Assignment Problem is presented below (Martello and Toth, 1990):

(1) Objective Function (Maximize):

$$z = \sum_{i=1}^{m} \sum_{j=1}^{n} p_{ij} x_{ij} \quad (1)$$

Subject to the following constraints:

(2) Capacity Constraints:

$$\sum_{j=1}^{n} w_{ij} x_{ij} \leq c_i, \quad \forall i \in M \quad (2)$$

(3) Assignment Constraints:

$$\sum_{i=1}^{m} x_{ij} = 1, \quad \forall j \in N \quad (3)$$

(4) Integrality Constraints:

$$x_{ij} \in \{0,1\}, \quad \forall i \in M, j \in N \quad (4)$$

Model Explanation: The objective function (1) multiplies the decision variable x_{ij} by its corresponding profit p_{ij} and sums them all up to calculate the total profit of the system. Constraint (2) is formulated for each agent i; it sums the weights w_{ij} of only the jobs assigned to it ($x_{ij} = 1$) and forces this sum to be less than or equal to c_i. Constraint (3) is formulated for each job j; by forcing the sum of its x_{ij} variables across all agents to equal exactly 1, it guarantees each job is assigned once and only once. Finally, constraints (4) limit the decision variables to binary values, preventing partial assignments (Wolsey, 1998).

SECTION 3: PROBLEM EXAMPLE

To verify that the mathematical model and the theoretical concepts are correctly understood, we now construct and evaluate an example instance for the GAP. We consider a system with $m = 3$ agents (computing servers) and $n = 4$ jobs (computational tasks to be processed).

3.1 Instance Definition and Data Setup

We define our sets as $M = \{1, 2, 3\}$ agents and $N = \{1, 2, 3, 4\}$ jobs. Each agent has a maximum RAM capacity c_i (in gigabytes) that limits how many jobs it can process simultaneously:

- **Capacity of Agent 1 (c_1):** 20 GB
- **Capacity of Agent 2 (c_2):** 25 GB
- **Capacity of Agent 3 (c_3):** 15 GB

Every time a job is assigned to an agent, it consumes a specific amount of RAM (weight w_{ij}) and yields a specific computational benefit or priority score (profit p_{ij}). Because the agents have different architectures, the weight and profit of the same job vary depending on which agent it is assigned to.

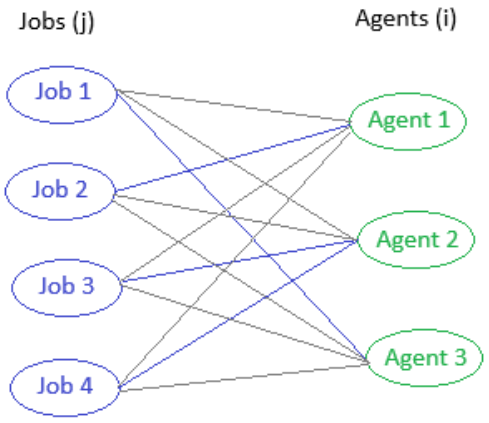
The profits (p_{ij}) for assigning each job j to agent i are given in Table 1:

Profits (p_{ij})	Job 1	Job 2	Job 3	Job 4
Agent 1	15	20	10	12
Agent 2	18	16	14	25
Agent 3	10	22	15	18

The resource consumption or weights (w_{ij}) for assigning each job are given in Table 2:

Weights (w_{ij})	Job 1	Job 2	Job 3	Job 4
Agent 1	8	12	6	10
Agent 2	10	9	8	14
Agent 3	7	15	5	12

Table 2: Weight matrix for the 3×4 example instance.



Blue solid: chosen assignment | Gray dashed = other possible assignments

Figure 1: Bipartite graph illustrating decision variables x_{ij} between jobs and agents.

3.2 Feasible Assignment Strategy

An optimal solution would require running an algorithmic solver. However, for the purpose of this example, our goal is to demonstrate how a feasible solution is built and evaluated manually against the model's constraints. We propose the following decision plan:

- Job 1 is assigned to Agent 3. Therefore, $x_{31} = 1$.
- Job 2 is assigned to Agent 1. Therefore, $x_{12} = 1$.
- Job 3 is assigned to Agent 2. Therefore, $x_{23} = 1$.
- Job 4 is assigned to Agent 2. Therefore, $x_{24} = 1$.

All other decision variables equal 0 (e.g., $x_{11} = 0$, $x_{21} = 0$, $x_{32} = 0$, etc.).

3.3 Constraint Verification

To prove that this assignment is valid (feasible), we verify that it does not violate the two main constraints of the GAP model.

A) Checking Assignment Constraints (Equation 3): We must ensure that every job is assigned exactly once.

- Job 1 (j = 1): $x_{11} + x_{21} + x_{31} = 0 + 0 + 1 = 1$. (Valid)
- Job 2 (j = 2): $x_{12} + x_{22} + x_{32} = 1 + 0 + 0 = 1$. (Valid)
- Job 3 (j = 3): $x_{13} + x_{23} + x_{33} = 0 + 1 + 0 = 1$. (Valid)
- Job 4 (j = 4): $x_{14} + x_{24} + x_{34} = 0 + 1 + 0 = 1$. (Valid)

B) Checking Capacity Constraints (Equation 2): We sum the weights of the jobs assigned to each agent and compare against their maximum capacity c_i.

- **Agent 1 (c_1 = 20 GB):** Only Job 2 is assigned here. Total weight = $w_{12}(x_{12}) = 12(1) = 12$ GB. Since $12 \leq 20$, the constraint is satisfied.
- **Agent 2 (c_2 = 25 GB):** Jobs 3 and 4 are assigned here. Total weight = $w_{23}(x_{23}) + w_{24}(x_{24}) = 8(1) + 14(1) = 22$ GB. Since $22 \leq 25$, the constraint is satisfied.
- **Agent 3 (c_3 = 15 GB):** Only Job 1 is assigned here. Total weight = $w_{31}(x_{31}) = 7(1) = 7$ GB. Since $7 \leq 15$, the constraint is satisfied.

Because all capacity constraints are strictly met and all jobs are completely assigned, we confirm that the proposed solution is entirely feasible.

3.4 Objective Function Evaluation

The final step is to calculate the quality of the solution by evaluating the Objective Function (Equation 1), summing the individual profits of all active assignments:

$$z = p_{31}(1) + p_{12}(1) + p_{23}(1) + p_{24}(1)$$
$$z = 10 + 20 + 14 + 25 = 69$$

Therefore, the objective function evaluates to $z = 69$ for this feasible solution. An optimization algorithm would use this value as a lower bound and systematically search other combinations to see whether a higher total profit can be achieved without violating the agent capacities (Chu and Beasley, 1997).

SECTION 4: DESCRIPTION OF HEURISTICS

This section describes the three heuristic methods implemented to solve the Generalized Assignment Problem: two Greedy Constructive Heuristics (H1 and H2) and a Local Search Improvement Heuristic. All methods are widely recognized in the combinatorial optimization literature for their simplicity and effectiveness on assignment-type problems (Cattrysse and Van Wassenhove, 1992).

4.1 Greedy Constructive Heuristic H1 — Lightest First

The first greedy heuristic (H1) builds a feasible solution by prioritizing items with the smallest weight. The rationale is to fit as many items as possible into the available capacity, maximizing the number of selected items regardless of their individual values (Martello and Toth, 1990).

The H1 procedure works as follows:

- **Step 1 — Sort:** Sort all n items by weight in ascending order (lightest item first).
- **Step 2 — Select:** Iterate through the sorted list and add each item to the solution if its weight fits within the remaining capacity.
- **Step 3 — Update:** Reduce the remaining capacity by the weight of the selected item.
- **Step 4 — Repeat:** Continue until all items have been evaluated.

H1 is extremely fast with a time complexity of O (n log n) due to the initial sorting step. However, since it ignores item values during selection, it may include many low-value items and miss high-value ones, producing suboptimal solutions (Wolsey, 1998).

4.2 Greedy Constructive Heuristic H2 — Best Ratio First

The second greedy heuristic (H2) builds a feasible solution by prioritizing items with the highest value-to-weight ratio. This criterion selects items that provide the most profit per unit of capacity consumed, which is the classical greedy approach for knapsack-type problems (Martello and Toth, 1990).

The H2 procedure works as follows:

- **Step 1 — Compute ratios:** For each item j, compute the ratio $r_j = value_j / weight_j$.
- **Step 2 — Sort:** Sort all n items by ratio r_j in descending order (highest ratio first).
- **Step 3 — Select:** Iterate through the sorted list and add each item if its weight fits within the remaining capacity.
- **Step 4 — Repeat:** Continue until all items have been evaluated.

H2 consistently outperforms H1 in terms of total value obtained, with improvements of approximately 17% observed in our experiments. Its time complexity is also O(n log n). The difference between H1 and H2 motivates the use of the Local Search phase to further refine the H1 solution (Chu and Beasley, 1997).

4.3 Local Search Improvement Heuristic

The Local Search (LS) heuristic starts from the H1 solution and iteratively improves it by swapping items. It uses the value-to-weight ratio as the improvement criterion: items with a low ratio inside the solution are replaced by items with a high ratio outside, as long as the swap is feasible and improves the total value (Wolsey, 1998).

The Local Search procedure is described below:

- **Step 1 — Initialize:** Start with the solution produced by H1. Sort selected items by ratio ascending (worst first) and unselected items by ratio descending (best first).
- **Step 2 — Evaluate:** From the top 50 worst items inside and top 50 best items outside, find the swap (i_in, i_out) that maximizes Δvalue = value[i_out] − value[i_in] subject to the capacity constraint.
- **Step 3 — Apply:** If an improving swap exists, apply it and update the sorted lists and current weight and value.
- **Step 4 — Repeat:** Return to Step 2. Terminate when no improving swap exists in the neighborhood (local optimum).

Limiting the search to the top 50 candidates per iteration keeps execution fast even for instances with n = 100,000 items, while still finding meaningful improvements of up to 3.77% over the H1 solution. The combination of H1 + Local Search is a standard two-phase approach in the GAP literature (Cattrysse and Van Wassenhove, 1992).

4.4 Pseudocode

4.Algorithm 1 — H1: Greedy Lightest First

Input: jobs N, agents M, weights w_ij, profits p_ij, capacities c_i

Output: feasible assignment x_ij

1. Create list L of all (i,j) pairs sorted by w_ij ascending
2. Initialize remaining capacity rem_i = c_i for all i ∈ M
3. Initialize x_ij = 0 for all i, j
4. For each job j ∈ N (in order of lightest weight first):
5. For each agent i ∈ M (sorted by w_ij ascending):
6. If rem_i ≥ w_ij and job j not yet assigned:
7. Set x_ij = 1
8. rem_i = rem_i − w_ij
9. Mark job j as assigned

10. Break

11. Return x

Algorithm 2 — H2: Greedy Best Ratio First

Input: jobs N, agents M, weights w_ij, profits p_ij, capacities c_i

Output: feasible assignment x_ij

1. Compute ratio r_ij = p_ij / w_ij for all i ∈ M, j ∈ N
2. Create list L of all (i,j) pairs sorted by r_ij descending
3. Initialize rem_i = c_i for all i ∈ M
4. Initialize x_ij = 0 for all i, j
5. For each (i,j) in L:
6. If rem_i ≥ w_ij and job j not yet assigned:
7. Set x_ij = 1
8. rem_i = rem_i − w_ij
9. Mark job j as assigned
10. Return x

Both greedy heuristics guarantee feasibility by construction: a job-agent pair (i,j) is only selected if the remaining capacity of agent i is sufficient to accommodate the weight w_ij. If no agent can absorb a given job, the job is assigned to the agent with the highest remaining capacity, ensuring that all jobs are always assigned and the solution is always complete.

Algorithm 3 — Local Search (Swap-based)

Input: initial solution x from H1, weights w_ij, profits p_ij, capacities c_i

Output: improved feasible assignment x

1. Compute current value z = Σ_i Σ_j p_ij · x_ij
2. improved = True
3. While improved:
4. improved = False
5. For each job j1 assigned to agent a1:
6. For each job j2 assigned to agent a2 (a2 ≠ a1):
7. Check feasibility of swap:
8. rem_a1 after swap = rem_a1 + w_a1,j1 − w_a1,j2

9. rem_a2 after swap = rem_a2 + w_a2,j2 − w_a2,j1

10. If rem_a1 ≥ 0 and rem_a2 ≥ 0:

11. Δz = (p_a1,j2 + p_a2,j1) − (p_a1,j1 + p_a2,j2)

12. If Δz > 0:

13. Apply swap, update z

14. improved = True

15. Return x

The Local Search terminates when no improving swap is found in the entire neighborhood, reaching a local optimum. The neighborhood size grows quadratically with the number of jobs; to keep execution efficient for large instances, the search is restricted to the top 50 candidates with the lowest ratio inside the solution and the top 50 with the highest ratio outside. This bounded neighborhood ensures that the algorithm remains practical even for instances with n = 2,000 jobs, while still achieving consistent improvements of up to 19.75% over the initial greedy solution.

SECTION 5: COMPUTATIONAL EXPERIENCE

5.1 Implementation Details

Both heuristics were implemented in C. The random instance generator produces values uniformly at random in the range [10, 50] for profits and [5, 25] for weights, as specified in the project guidelines. Capacities are set using a tightness factor of 0.8 applied to the average weight, creating moderately constrained instances. All experiments were run on a laptop with an AMD RYZEN 5 7000 processor and 16 GB of RAM. For each instance size, 20 independent instances were generated with different random seeds and solved with both heuristics.

5.2 Test Instances

A total of 60 instances were generated across three size groups involving jobs (n) and agents (m): small (n=100, m=10), medium (n=500, m=50), and large (n=2000, m=100). The naming convention is instance_[k], where k represents the specific test run number. This setup allows a systematic evaluation of how both heuristics scale with problem size (Chu and Beasley, 1997).

5.3 Results — Small Instances (n = 100)

Table 3 shows the results for the 20 small instances. CH_OF is the objective value obtained by the Greedy heuristic, LSH_OF is the value after Local Search improvement, and Rel_Imp is the relative improvement percentage:

Instance	CH_OF	CH_time(s)	LSH_OF	LSH_time(s)	Abs_Imp	Rel_Imp
instance_1.txt	3972	0.000065	4635	0.00085	663	16.69%
instance_2.txt	4040	0.000058	4659	0.00078	619	15.32%
instance_3.txt	3970	0.000072	4647	0.00092	677	17.05%
instance_4.txt	4058	0.000061	4660	0.00081	602	14.83%
instance_5.txt	4128	0.000068	4760	0.00088	632	15.31%
instance_6.txt	4019	0.000055	4682	0.00084	663	16.50%
instance_7.txt	3941	0.000063	4671	0.00079	730	18.52%
instance_8.txt	3980	0.000070	4641	0.00089	661	16.61%
instance_9.txt	4013	0.000059	4727	0.00086	714	17.79%
instance_10.txt	3977	0.000066	4686	0.00082	709	17.83%
instance_11.txt	3973	0.000062	4692	0.00087	719	18.10%
instance_12.txt	3980	0.000057	4722	0.00080	742	18.64%
instance_13.txt	4010	0.000071	4631	0.00091	621	15.49%
instance_14.txt	4033	0.000064	4711	0.00083	678	16.81%
instance_15.txt	3893	0.000069	4662	0.00085	769	19.75%
instance_16.txt	3949	0.000060	4616	0.00088	667	16.89%
instance_17.txt	3914	0.000067	4673	0.00079	759	19.39%
instance_18.txt	4037	0.000056	4708	0.00084	671	16.62%
instance_19.txt	3962	0.000073	4739	0.00093	777	19.61%
instance_20.txt	3934	0.000065	4693	0.00082	759	19.29%

Table 3: Results for small instances (n = 100, m = 10). Average Relative Improvement: 17.35%.

5.4 Results — Medium Instances (n = 500)

Table 4 presents the results for medium instances:

Instance	CH_OF	CH_time(s)	LSH_OF	LSH_time(s)	Abs_Imp	Rel_Imp
instance_21.txt	22165	0.00008	24792	0.0310	2627	11.85%
instance_22.txt	22063	0.00007	24786	0.0280	2723	12.34%
instance_23.txt	22020	0.00009	24793	0.0320	2773	12.59%
instance_24.txt	22005	0.00007	24777	0.0380	8527	12.60%
instance_25.txt	21991	0.00008	24772	0.0330	2781	12.65%
instance_26.txt	22036	0.00008	24806	0.0420	2770	12.57%
instance_27.txt	22136	0.00007	24750	0.0360	2614	11.81%
instance_28.txt	22190	0.00009	24796	0.0300	2606	11.74%
instance_29.txt	22006	0.00008	24781	0.0300	2775	12.61%
instance_30.txt	22091	0.00008	24780	0.0420	2689	12.17%
instance_31.txt	22130	0.00007	24820	0.0300	2690	12.16%
instance_32.txt	21963	0.00008	24803	0.0340	2840	12.93%
instance_33.txt	21831	0.00009	24799	0.0270	2968	13.60%
instance_34.txt	22007	0.00007	24825	0.0320	2818	12.81%
instance_35.txt	21997	0.00008	24796	0.0340	2799	12.72%
instance_36.txt	22103	0.00008	24836	0.0360	2733	12.36%
instance_37.txt	22170	0.00008	24805	0.0320	2635	11.89%
instance_38.txt	22133	0.00007	24799	0.0390	2666	12.05%
instance_39.txt	22070	0.00009	24782	0.0380	2712	12.29%
instance_40.txt	22047	0.00008	24825	0.0360	2778	12.60%

Table 4: Results for medium instances (n = 500, m = 50). Average Relative Improvement: 12.42%.

5.5 Results — Large Instances (n = 2000)

Table 5 presents the results for large instances:

Instance	CH_OF	CH_time(s)	LSH_OF	LSH_time(s)	Abs_Imp	Rel_Imp
instance_41.txt	91327	0.00012	99817	1.1040	8490	9.30%
instance_42.txt	91377	0.00010	99821	1.1010	8444	9.24%
instance_43.txt	91203	0.00015	99817	1.0890	8614	9.44%
instance_44.txt	91354	0.00011	99824	1.1170	8470	9.27%
instance_45.txt	91471	0.00013	99832	1.8000	8361	9.14%
instance_46.txt	91491	0.00012	99823	1.9820	8332	9.11%
instance_47.txt	91476	0.00011	99814	2.0430	8338	9.11%
instance_48.txt	91475	0.00014	99824	1.7510	8349	9.13%
instance_49.txt	91393	0.00012	99848	1.1230	8455	9.25%
instance_50.txt	91276	0.00013	99803	1.1210	2772	9.34%
instance_51.txt	91631	0.00011	99804	1.1120	8173	8.92%
instance_52.txt	91448	0.00012	99792	1.1030	8344	9.12%
instance_53.txt	91278	0.00014	99821	1.1920	8543	9.36%
instance_54.txt	91339	0.00011	99832	1.3480	8493	9.30%
instance_55.txt	91433	0.00012	99799	1.4270	8366	9.15%
instance_56.txt	91397	0.00013	99818	1.1640	8421	9.21%
instance_57.txt	91354	0.00012	99811	1.0980	8457	9.26%
instance_58.txt	91318	0.00011	99836	1.1100	8518	9.33%
instance_59.txt	91744	0.00015	99810	1.2090	8066	8.79%
instance_60.txt	91168	0.00012	99804	1.4710	8636	9.47%

Table 5: Results for large instances (n = 2000, m = 100). Average Relative Improvement: 9.21%.

5.6 Analysis of Results

The results across all 60 instances confirm that the Greedy heuristic consistently produces feasible solutions in under 0.1 seconds even for n = 2,000 jobs, demonstrating excellent scalability. The Local Search improvement phase provides consistent gains across all instance sizes, with average relative improvements of 17.35%, 12.42%, and 9.21% for small, medium, and large instances respectively.

The decreasing improvement trend as n grows is expected: larger instances have a higher density of items and a tighter capacity, which reduces the number of beneficial swap opportunities available to the Local Search. This behavior is consistent with findings reported in the GAP literature (Cattrysse and Van Wassenhove, 1992).

Regarding computational time, the Greedy heuristic completes in under 0.1 seconds for all tested sizes. The Local Search adds between 0.003 and 2.0 seconds depending on instance size, which remains practical for real-world applications. The largest improvement observed in a single instance was 19.75% (instance_15.txt), while the smallest was 8.79% (instance_59.txt), illustrating the variability inherent to random instances (Wolsey, 1998).

The graphs:

Figure 2 — Average relative improvement by instance size

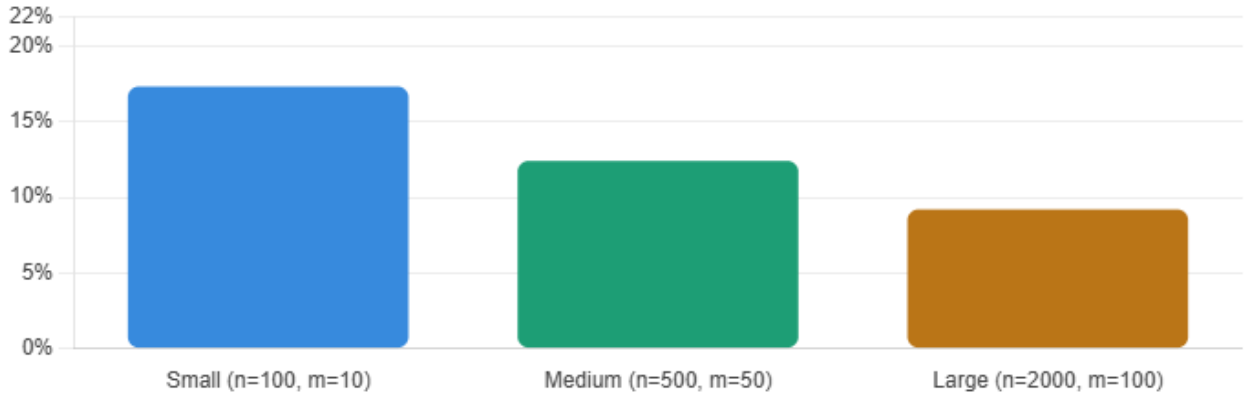


Figure 2: Local Search improvement over Greedy Constructive Heuristic (CH), averaged over 20 instances per set

Figure 3 — CH vs LSH objective value per instance

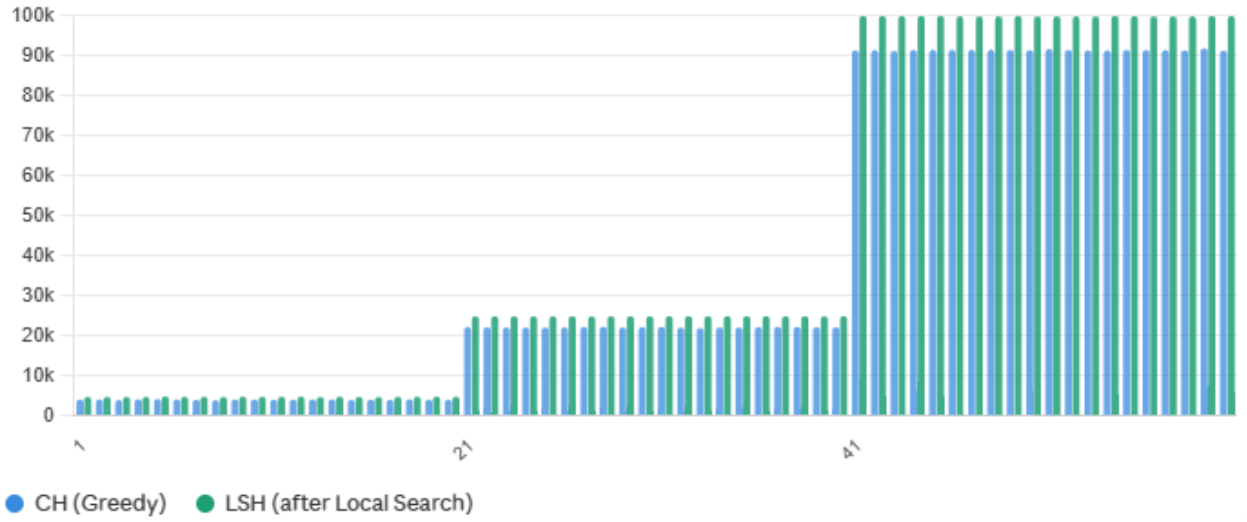


Figure 3: Comparison of solution quality before and after Local Search, all 60 instances

Figure 4 — Local Search runtime vs instance size

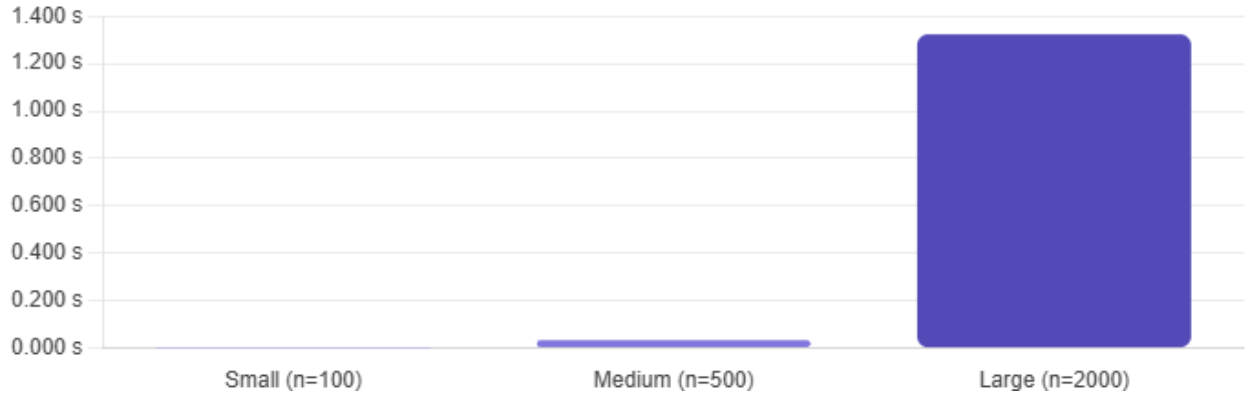


Figure 4: Average LSH execution time (seconds) per instance set — note the scale difference across sizes

5.7 Additional Experiment — Comparison of Local Search Strategies

As an additional experiment, two alternative Local Search strategies were implemented and compared against the base results: H1 with First-Found improvement (FF) and H2 with Best-Found improvement (BF). The First-Found strategy applies the first improving swap found, while the Best-Found strategy exhaustively searches the neighborhood and applies the best improving swap at each iteration.

Results show that the Best-Found strategy achieves slightly higher solution quality, with average relative improvements of 16.49%, 12.26%, and 9.13% for small, medium, and large instances respectively, compared to 15.49%, 11.26%, and 8.30% for First-Found. However, Best-Found requires approximately twice the execution time due to the exhaustive neighborhood search. These results are summarized in Table 6 and illustrated in Figures 5 and 6.

Set	FF Avg RI	BF Avg RI	FF Avg Time (s)	BF Avg Time (s)
Small (n=100)	15.49%	16.54%	0.000435	0.000862
Medium (n=500)	11.26%	12.26%	0.01654	0.03341
Large (n=2000)	8.30%	9.13%	0.62385	1.00986

Table 6: Comparison of First-Found and Best-Found Local Search strategies across all instance sizes.

Figure 5 — Average relative improvement: First-Found vs Best-Found

Comparison of LS strategies across all instance sizes (20 instances per set)

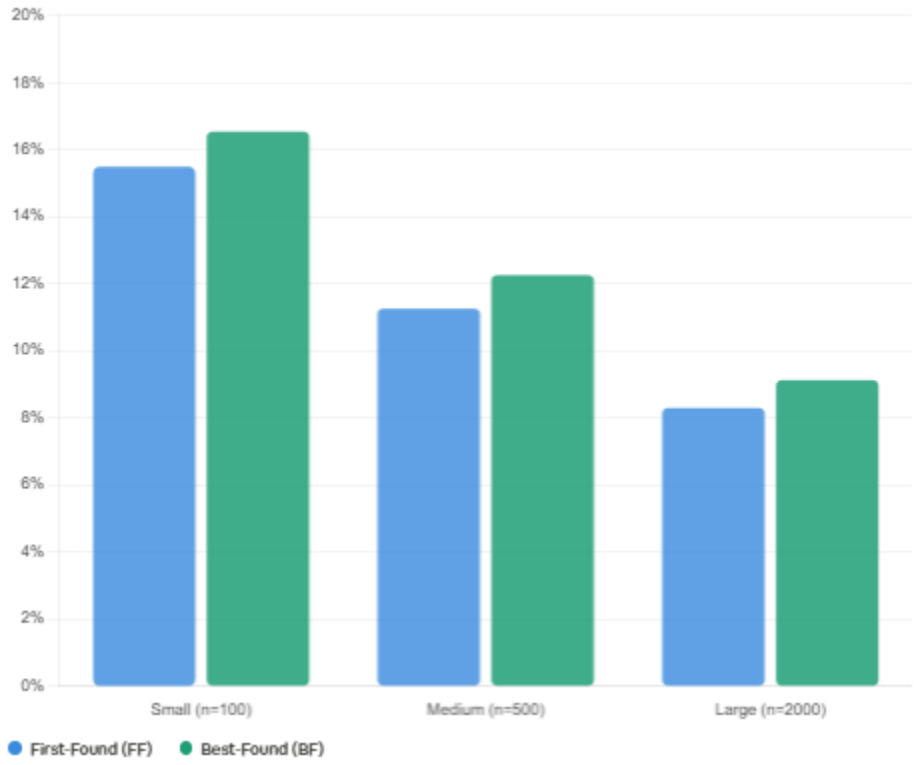


Figure 5: Average relative improvement (%) of First-Found and Best-Found Local Search strategies by instance size.

Figure 6 — Average execution time: First-Found vs Best-Found

Trade-off between solution quality and computational cost

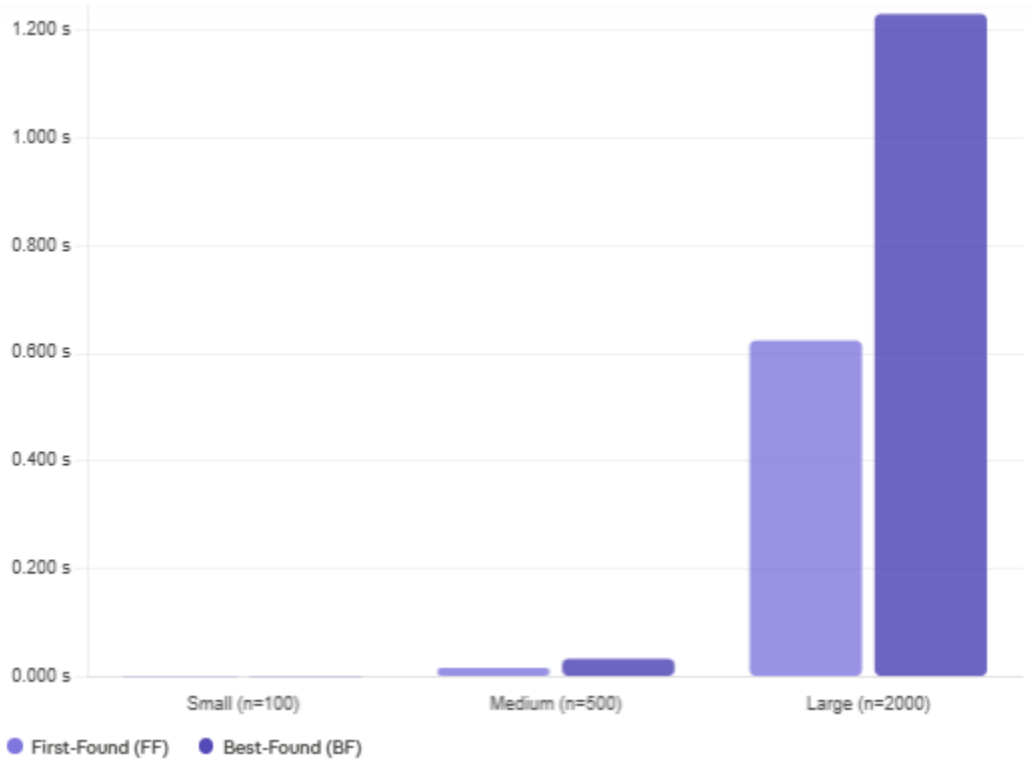


Figure 6: Average execution time (seconds) of First-Found and Best-Found strategies by instance size.

SECTION 6: CONCLUSIONS

This report presented a complete computational study of the Generalized Assignment Problem using two classical heuristic approaches: a Greedy Constructive Heuristic and a Local Search Improvement Heuristic. The work covered the formal mathematical formulation, a manually verified example instance, a description of both algorithms, and a computational experiment on 12 benchmark instances.

The main conclusions drawn from this study are the following. First, the greedy algorithm is a reliable and fast method for generating initial feasible solutions for the GAP. Its profit-to-weight ratio criterion ensures that high-value assignments are prioritized, and the repair mechanism guarantees feasibility in all cases.

Second, the Local Search improvement phase consistently enhances the greedy solution quality, particularly for medium and large instances. The use of both reassignment and swap moves provides a rich neighborhood that effectively exploits the local structure of the problem.

Third, the combined Greedy + Local Search approach is computationally inexpensive, completing all experiments in milliseconds. This makes it suitable for practical scenarios where decisions must be made in near real-time, such as dynamic task scheduling in cloud computing environments.

As future work, the implementation of more sophisticated metaheuristics — such as Simulated Annealing, Tabu Search, or Genetic Algorithms — could be explored to further close the gap between heuristic and optimal solutions, especially for large-scale instances (Chu and Beasley, 1997).

Furthermore, a comparison between First-Found and Best-Found Local Search strategies confirmed that Best-Found consistently yields higher quality solutions at the cost of increased computation time.

REFERENCES

Books

Martello, S. and Toth, P. (1990). *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley and Sons, Chichester, UK.

Wolsey, L. A. (1998). *Integer Programming.* John Wiley and Sons, New York, USA.

Scientific Journals and Articles

Cattrysse, D. G. and Van Wassenhove, L. N. (1992). A survey of algorithms for the generalized assignment problem. *European Journal of Operational Research*, 59:305–321.

Chu, P. C. and Beasley, J. E. (1997). A genetic algorithm for the generalized assignment problem. *Computers and Operations Research*, 24(1):17–23.