AMMM Project

Matthias Nadal, Virginia Nicosia

10/12/2024

https://github.com/abrowng/cpds-opty

Contents

Formal Problem Statement

The goal is to form a committee of faculty members that satisfies specific requirements while maximizing compatibility. The problem can be formally described as follows:

Inputs

- Total number of faculty members: $N \in \mathbb{Z}^+$.
- Total number of departments: $D \in \mathbb{Z}^+$.
- d_i: Department to which faculty member i belongs, $d_i \in \{1, 2, \ldots, D\}$.
- n_p: Number of participants required from department p, $p = 1, 2, \ldots, D$.
- m_{ij}: Compatibility matrix where m_{ij} represents the compatibility between faculty members i and j ($0 \leq m_{ij} \leq 1$, with $m_{ii} = 1$ for all i).

Outputs

- A set of faculty members forming the committee, satisfying the constraints.

Objective Function

Maximize the average compatibility among all pairs of participants in the committee.

Problem Description

The problem is to select a subset of faculty members to form a committee such that:

- Each department p contributes exactly n_p members.
- No two members i and j with $m_{ij} = 0$ are included.
- If two members with compatibility $0 < m_{ij} < 0.15$ are in the committee there must be a third member k such that $m_{ik} > 0.85$ and $m_{kj} > 0.85$.

The objective is to maximize the average compatibility of the selected members.

Integer Linear Programming Model

Decision Variables

- x_i: Binary decision variable that is equal to 1 if member i is included in the committee and 0 otherwise .
- x_{ij}:Binary decision variable that is equal to 1 if both members i and j are included in the committee and 0 otherwise.
- y_{ip}: Binary decision variable that is equal to 1 if member i is a part of the department p and 0 otherwise.

Mathematical Formulation

Objective Function:

$$maximize \frac{1}{\sum_{i=1}^{N}\sum_{j=i+1}^{N} x_{ij}} \sum_{i=1}^{N}\sum_{j=i+1}^{N} m_{ij}x_{ij} \tag{1}$$

Subject to:

1. $y_{ip}$is setted to one only when member i belongs to department p

$$y_{ip} = \begin{cases} 1, & \text{if } d[i] = p \\ 0, & \text{otherwise} \end{cases}, \quad \forall i \in N,\, p \in D \tag{2}$$

2. x_{ij} is setted to one only when both members i and j belong to the committee

$$x_{ij} \leq x_i, \quad \forall i, j \in N,\, i \neq j \tag{3}$$
$$x_{ij} \leq x_j, \quad \forall i, j \in N,\, i \neq j \tag{4}$$
$$x_{ij} \geq x_i + x_j - 1, \quad \forall i, j \in N,\, i \neq j \tag{5}$$

3. Total number of committee members must be equal to the total numbers of members that represents every department:

$$\sum_{i \in N} x_i = \sum_{p \in D} n_p \tag{6}$$

4. Each department must have n_p representatives in the committee:

$$\sum_{i \in N} y_{ip} \cdot x_i = n_p, \quad \forall p \in D \tag{7}$$

5. If two members are incompatible, at most 1 of them can be in the commission.:

$$x_i + x_j < 1, \quad \forall i, j \in N, \text{ such that } m_{ij} = 0 \tag{8}$$

6. If two members get along poorly there must be a third one that goes along well with both of them:

$$\sum_{\substack{k \in N, \\ k \neq i, k \neq j}} \left(x_k \cdot \mathbb{1}_{m[i,k] \geq 0.85} \cdot \mathbb{1}_{m[j,k] \geq 0.85} \right) \geq 1, \quad \forall i, j \in N, \text{ where } 0 < m[i,j] < 0.15 \tag{9}$$

Meta-Heuristics Algorithms

Greedy Algorithm

The greedy algorithm is a simple and intuitive approach that incrementally constructs a solution by making the best local choice at each step. In this problem, the goal is to maximize the compatibility of the selected committee members while adhering to constraints such as departmental quotas and pairwise compatibility restrictions. At each step, the algorithm evaluates all feasible candidates based on their compatibility with the already selected committee members; the candidate with the highest compatibility score, who also satisfies the constraints, is selected. The process continues until the required number of members has been selected or no feasible candidates remain. The greedy algorithm is computationally efficient and provides a good initial solution but it may not handle all constraints optimally that is why his solutions are often refined using local search or GRASP

Algorithm: Greedy Construction

- **Input:** A problem instance.
- **Output:** A feasible committee solution.

```
Method _selectCandidate(teachers):
If config.solver == "Greedy":
Sort teachers by descending compatibility
Return teacher with highest compatibility
Else:
Return random.choice(teachers)
```

```
Method construction():
solution = instance.createSolution()
sizeCommittee = instance.getSizeCommittee()

For i = 1 to sizeCommittee:
    candidateList = solution.getFeasibleTeachers()
    If candidateList is empty:
        solution.makeInfeasible()
        Break
    candidate = _selectCandidate(candidateList)
    solution.assign(candidate)
Return solution
```

Greedy + Local Search

After constructing an initial solution using the greedy algorithm, we can improve it by applying a local search procedure. The greedy algorithm makes decisions step by step, focusing on the best immediate option, but it does not account for potential improvements by exploring alternative configurations; the local search algorithm addresses this limitation by exploring small changes, refining the solution.

Algorithm: Local Search Improvement

- **Input:** A feasible committee solution.
- **Output:** An improved feasible solution.

```
Method exploreExchange(solution):
neighbor = None
_, currentFitness = solution.calculateFitness()
nonAssignedTeachers = solution.getNonAssignedTeachers()

For teacher in solution.getAssignedTeachers():
    neighborSolution = solution.clone()
    neighborSolution.unassign(teacher)

    For candidate in nonAssignedTeachers:
        If not neighborSolution.isFeasible(candidate):
            Continue

        neighborSolution.assign(candidate)
        _, newFitness = neighborSolution.calculateFitness()

        If newFitness > currentFitness:
            neighbor = neighborSolution
            currentFitness = newFitness
            Break
Return neighbor
```

```
Method solve():
initialSolution = kwargs.get("solution")
If not initialSolution.isFeasible():
Return initialSolution

incumbent = initialSolution
While not timeLimitExceeded():
    neighbor = exploreExchange(incumbent)
    If neighbor is None:
        Break
    incumbent = neighbor
Return incumbent
```

GRASP Algorithm

The GRASP algorithm is an iterative optimization technique that alternates between two key phases to solve combinatorial problems: construction and local search. In the construction phase, the algorithm creates a feasible solution using a randomized version of the greedy algorithm and a Restricted Candidate List (RCL) is created based on compatibility scores, controlled by the alpha-parameter, and candidates are randomly selected from this list to introduce diversity into the solution. After constructing an initial solution, the local search phase refines it by exploring its neighborhood: this involves making small changes, such as swapping or replacing members, to maximize compatibility while respecting all constraints. The local search continues until no further improvements can be made, reaching a local minimum. These two phases are repeated iteratively for a predefined number of iterations or until a time limit is reached (for us 30 seconds); at the end of the process, the best solution found is returned as the output. This approach balances exploration and exploitation, allowing the algorithm to effectively navigate the solution space.

Algorithm: GRASP

- **Input:** A problem instance, an alpha parameter, and a boolean flag `localSearch`.
- **Output:** A constructed solution.

```
Method _selectCandidate(teachers, alpha):
Sort teachers by descending compatibility
Define min and max compatibility
Set boundary compatibility using alpha
Create RCL with teachers below boundary
Return random.choice(RCL)

Method _greedyRandomizedConstruction(alpha):
solution = instance.createSolution()
sizeCommittee = instance.getSizeCommittee()

For i in range(sizeCommittee):
    candidateList = solution.getFeasibleTeachers()
```

```
    If candidateList is empty:
        solution.makeInfeasible()
        Break
    candidate = _selectCandidate(candidateList, alpha)
    solution.assign(candidate)
Return solution

Method stopCriteria():
elapsedTime = currentTime - startTime
Return elapsedTime > config.maxExecTime

Method solve():
Start timer
incumbent = instance.createSolution()
incumbent.makeInfeasible()
bestFitness = incumbent.getFitness()

While not stopCriteria():
    solution = _greedyRandomizedConstruction(alpha)
    If solution is feasible:
        fitness = solution.getFitness()
        If fitness > bestFitness:
            incumbent = solution
            bestFitness = fitness
Return incumbent
```

Execution Time Analysis

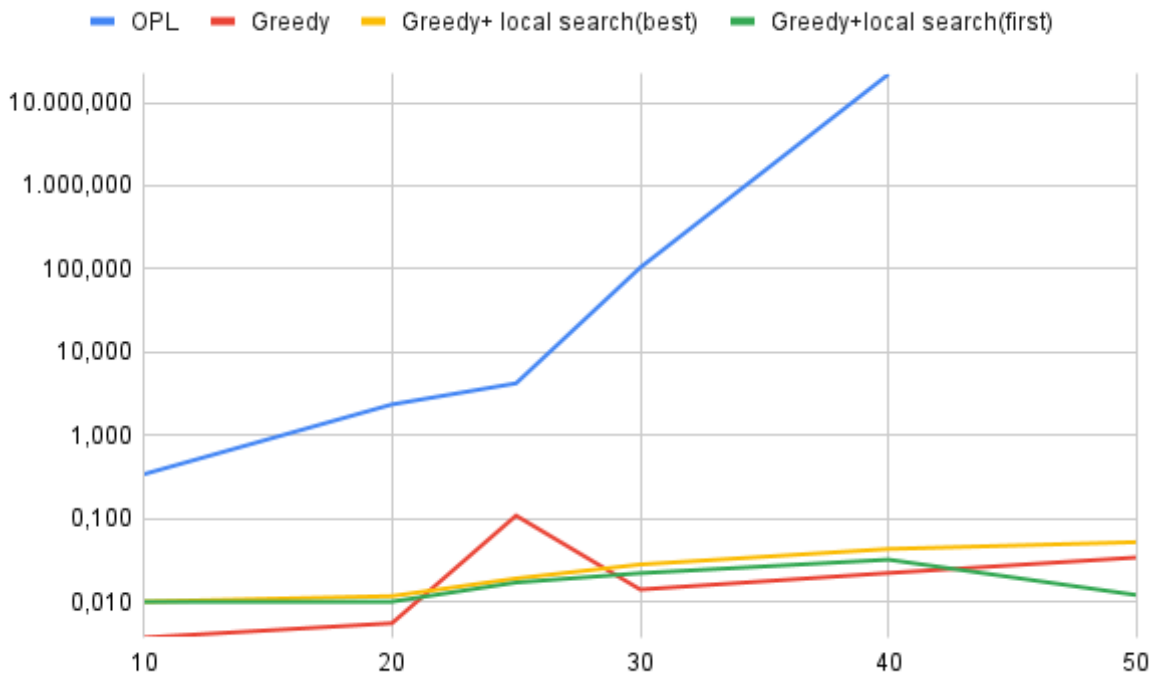
The study revealed significant differences in execution times among the algorithms tested:

- **CPLEX Solver:** An exact method producing optimal solutions. However, for instances with 50 members or more, the execution time often exceeded 20 minutes, making it impractical for large-scale problems.
- **Heuristic Approaches:** Algorithms such as Greedy and Local Search produced results in less than a second. The GRASP algorithm, while slower, was capped at 30 seconds to ensure a balance between computational time and solution quality.

Solution Quality and Trade-offs

An essential aspect of this study was the evaluation of solution quality relative to computation time. The findings include:

- **CPLEX:** Predictably provided the highest-quality solutions due to its exact nature. However, the increasing computational cost with problem size makes it unsuitable for instances exceeding 50 members.



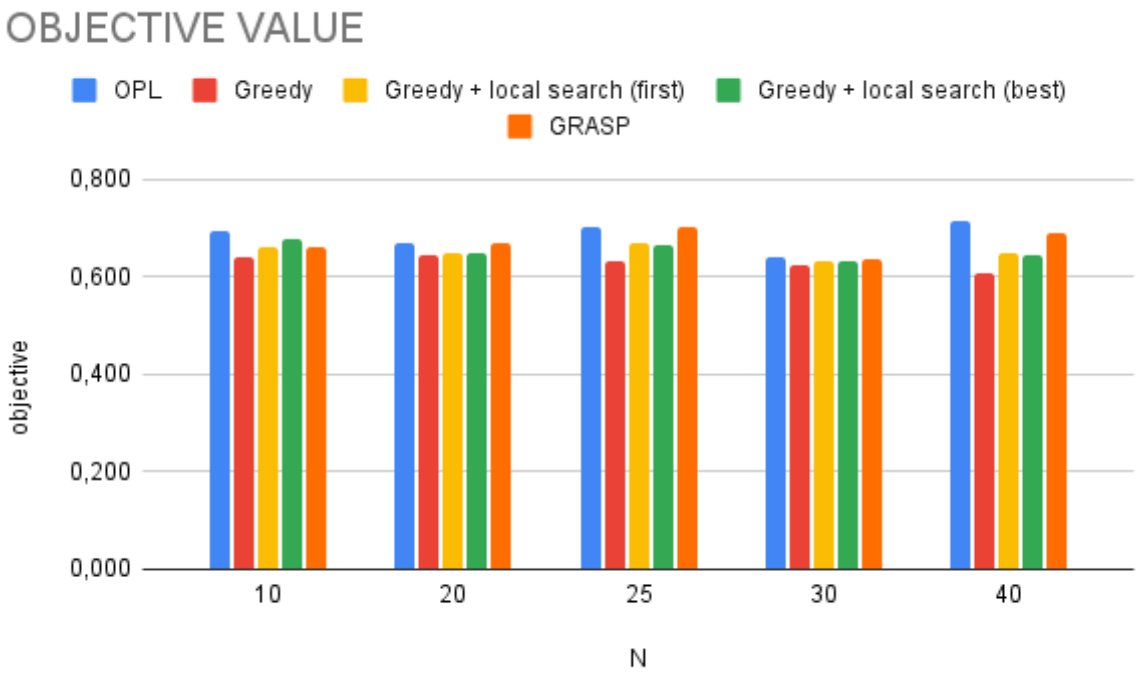
- **Greedy Algorithm and Local Search:** While Greedy is the fastest, its solutions are less accurate. Local Search enhances the Greedy algorithm's results, consistently achieving better quality with a proportional increase in computation time. This balance makes Local Search a preferred heuristic approach.
- **GRASP:** Aimed to improve solution quality further, with a cap on computation time to maintain practical usability.

Comparison of Best Results Found

The analysis also considered the differences in the best results found by each algorithm, as illustrated by the accompanying chart. The key observations are:

- **CPLEX:** Consistently produced the most accurate solutions, validating its suitability for small to medium-sized problems where computation time is not a critical constraint.
- **Greedy Algorithm:** While fast, its results showed a noticeable gap compared to the optimal solutions from CPLEX.
- **Local Search:** Demonstrated a significant improvement over Greedy, narrowing the gap with CPLEX without incurring excessive computation time.
- **GRASP:** Provided high-quality solutions, often comparable to Local Search, within the designated time cap, making it a strong candidate for larger problem instances.

The chart highlights the trade-off between solution quality and computation time, reinforcing the value of heuristic methods for practical applications.



Challenges in Problem Instances

The complexity of problem instances significantly affects computation time. Notable challenges include:

- **Feasibility:** Instances with infeasible solutions, such as when representation exceeds capacity, are particularly challenging.
- **Random Generation:** Randomly generated instances often produce unfavorable cases, emphasizing the need for reliable instance generation methods to ensure balanced and solvable problems.

Resource Limitations and Implications

The analysis was constrained by the available computational resources. Larger problem instances remain inaccessible, underscoring the necessity of efficient heuristics like Local Search and GRASP. These algorithms offer scalable solutions, enabling the handling of more complex inputs within practical limits.

Conclusion

In summary, while the CPLEX solver provides the best solutions, its computational demands make it impractical for large problem sizes. Heuristic methods, particularly Local Search, strike an effective balance between solution quality and computation time. This makes them ideal for tackling combinatorial optimization problems under real-world

constraints. Future efforts should focus on developing robust instance generators and exploring scalable heuristic approaches to address these challenges more effectively.