AMMM Final Project Report

Xin Tong[1] and Qiuchi Chen[2]

[1]MIRI xin.tong@estudiantat.upc.edu
[2]MIRI qiuchi.chen@estudiantat.upc.edu

May 22, 2025

Contents

1 Conflict Resolution Problem Statement

The Conflict Resolving problem can be formally stated as follows:

Given:

- The set $Members$ of N members.
- The matrix $m(n, n)$ represents the pairwise bids between members.

Find:

- $Priority(n, n)$ represents the assignment of priorities to ordered member pair $< i, j >$ subject to the following constraints:
  - A priority is assigned to one member of each unordered pair in case of a clash between the two members.
  - Members and Solution (set of member pairs, for each pair $< i, j >$ where member i prioritizes member j) form a directed acyclic graph, with members as vertices and solution as edges of the graph.

Objective:

- Maximize the total sum of bids taken from the higher priority pairwise members.

2 ILP Model Formulation

The conflict resolving problem can be modeled as an Integer Linear Program. To achieve this, the following sets and parameters are defined:

$Members$	Set of N members, index n.
m_{ij}	Member i places bids over member j.

The following decision variables are also defined:

$Priority_{ij}$	Binary. Equal to 1 if member i prioritizes member j; 0 otherwise.
O_n	Topological sort of the directed graph formed with members as vertices and $\{< i, j > \forall i, j \in Members \quad and \quad Priority_{ij} = 1\}$ as edges.
$Income$	Positive real. The total sum of money taken from a member with higher priority from each unordered pair.

$$Income = \sum_{i,j \epsilon Members} Priority_{ij} \cdot m_{ij}$$

Finally, the ILP model for the conflict resolving problem is as follows:

$$\text{maximize} \quad Income \tag{1}$$

subject to:

$$Priority_{ij} + Priority_{ji} = 1 \qquad \forall \quad i, j \in Members \quad \text{and} \quad m_{ij} + m_{ji} \neq 0 \tag{2}$$

$$O_i + 1 \leq O_j + (1 - Priority_{ij}) \cdot N \qquad \forall \quad i, j \in Members \quad \text{and} \quad m_{ij} + m_{ji} \neq 0 \tag{3}$$

The equation (3) can be derived through the following reasoning.
If the directed graph is acyclic, we can obtain a topological ordering of the vertices. This ordering is an array in which each vertex appears before all the vertices it points to. Our objective is to achieve

the topological order of the directed priority graph. If we successfully obtain this order, we can ensure that no cycles are formed in the graph.
$Priority_{ij}$ is used to indicate whether or not vertex i is before vertex j. If $Priority_{ij} = 1$ we wish to have $O_i + 1 \leq O_j$, i.e. if $(1 - Priority_{ij}) = 0$ we wish to have $O_i + 1 \leq O_j$. This condition is imposed if $O_i + 1 - O_j \leq M(1 - Priority_{ij})$ where M is a sufficiently large number. In order to find how large M must be we consider the case $Priority_{ij} = 0$ giving $O_i + 1 - O_j \leq M$. M must be chosen to be an upper bound for the expression $O_i + 1 - O_j$, which is $M = N + 1 - 1 = N$. Then we obtain (3).
We do not need to consider the reverse of (3), which states that $O_i < O_j \rightarrow Priority_{ij} = 1$. This means that if $Priority_{ij} = 0$, then $O_i \geq O_j$. However, equation (2) ensures that if $Priority_{ij} = 0$, then $Priority_{ji} = 1$, which leads to the conclusion that $O_j < O_i$.

3 Heuristics Method

3.1 Greedy Method

3.1.1 Variable Declaration

- $Members$: Set of members.
- $Bids_{ij}$: The bid member i places over member j.
- $CandidatePairs$: Set of all pairs of members $< i, j >$.
- $Priority(i, j)$: Binary. Equal to 1 if member i prioritizes member j, 0 otherwise.
- $Solution$: Set of member pairs. For each pair $< i, j > Priority[i][i] = 1$. k
- $CoveredMembers$: Set of members covered in the $Solution$.

3.1.2 Greedy Cost Function

We always consider the pair $< i, j >$ with the highest bid. If adding $< i, j >$ to the partial solution makes it infeasible, then we swap $< i, j >$ for $< j, i >$ in the solution. Whenever we add a pair to the partial solution, we assess its feasibility. If the pair $< i, j >$ is infeasible, we can be certain that $< j, i >$ is feasible.
For example, if the pair (2,3) results in an infeasible solution, we assume that (2,3), (3,1), and (1,2) create a priority cycle. Similarly, if (3,2) leads to an infeasible solution, we assume that (3,2), (2,4), and (4,3) also form a priority cycle. In this case, (3,1), (1,2), (2,4), and (4,3) would then constitute another priority cycle. However, we've already assessed the feasibility of (3,1), (1,2), (2,4), and (4,3), which presents a contradiction.

$$q(< i, j >, S) = \begin{cases} -1 & if \quad G(S, < i, j >) \text{ is not a DAG} \\ max(Bids_{ij}, Bids_{ji}) & if \quad G(S, < i, j >) \text{ is a DAG} \end{cases}$$

3.1.3 Pseudocode

Algorithm Greedy Algorithm

function ValidateCandidatePair($partialSolution$, $< i, j >$)
  $topologicalOrder, residualEdges \leftarrow TopologicalSort(partialSolution \cup \{i, j\})$
  if $residualEdges = \phi$ **then**
    return True
  else
    return False

$Solution \leftarrow \phi$
$CoveredMembers \leftarrow \phi$
$CandidatePairs \leftarrow \{< i, j > |\ i, j \in Members \quad \text{and} \quad Bids_{ij} + Bids_{ji} \neq 0\}$
$Priority(i, j) \leftarrow 0 \quad \forall < i, j > \in CandidatePairs$
while $CandidatePairs \neq \phi$ **do**
  Evaluate $q(< i', j' >, Solution) \quad \forall < i', j' > \in CandidatePairs$
  $< i, j > \leftarrow argmax\{q(< i', j' >, Solution) | < i', j' > \in CandidatePairs\}$
  $CandidatePairs \leftarrow CandidatePairs \setminus \{< i, j >, < j, i >\}$
  $CoveredMembers \leftarrow CoveredMembers \cup \{i, j\}$
  $Solution \leftarrow Solution \cup \{< i, j >\}$
  $Priority(i, j) \leftarrow 1$
return $Solution$

3.2 Local Search Method

3.2.1 Neighborhood

The neighborhood of this conflict resolution problem is defined as the solution obtained by replacing the member pair $< i, j >$ currently not included in the solution (indicated by $Priority(< i, j >) = 0$) with its flipped counterpart $< j, i >$ included in the solution (indicated by $Priority(< j, i >) = 1$), if $Bids_{ij} > Bids_{ji} \quad \forall i, j \in Members$.

3.2.2 Exploration Strategy

Based on the neighborhood mentioned above, the replacing procedure may necessitate the flipping of other pairs as a knock-on effect. If the total decrease in bids from passively flipped pairs (defined as ***knock-on flipped pairs***, represented by $\boldsymbol{edges^{flip}}$) is less than the increase from the actively flipped pair $< i, j >$, we achieve a better solution.
For example, if we have ($Bids_{ij} > Bids_{ji}$ and $< j, i > \in Solution$ (which indicates that $Priority(< j, i >) = 1$), when we consider replacing $< j, i >$ with $< i, j >$ in the current solution, some priority cycles may be formed. To address this, we need to develop a strategy for identifying pairs to flip within this directed graph, ensuring that no cycles remain.
First, we reverse the pair (representing the edges in the directed graph) from $< j, i >$ to $< i, j >$ in the current solution (representing the vertices in the directed graph). We then proceed to eliminate all edges with an indegree of 0 or an outdegree of 0, ensuring that all remaining edges in the graph are part of a cycle.
Once we have the remaining edges $edges^{remain}$, we utilize equation $max(q(< i, j' >, edges^{flip}))$ to continuously select edge $< i', j' >$ and added to $edges^{flip}$ until no cycles are left in the graph. We prioritize edges $< i', j' >$ with higher indegree and outdegree, as we believe such edges are more capable of eliminating multiple cycles compared to others. If multiple edges are tied, we will select the edge with the lowest bid decrease ($Bids_{i',j'} - Bids_{j'i'}$).The $edges^{flip}$ cost function is defined as follows.

$q(< i', j' >, edges^{flip})=$

$$\begin{cases} -1 \\ \qquad \text{if} \quad Bids_{i'j'} - Bids_{j'i'} + \sum_{<i'',j''>\in edges^{flip}} Bids_{i''j''} - Bidsj''i'' \geq Bids_{ij} - Bids_{ji} \\ \qquad \text{or } edges^{remain} \mid edges^{flip} = edges^{remain} \mid edges^{flip} \cup \{< i', j' >\}) \\ \\ \sum_{k'\in\{i',j'\}} indegree(k') + outdegree(k') \\ \\ \qquad \text{if} \quad Bids_{i'j'} - Bids_{j'i'} + \sum_{<i'',j''>\in edges^{flip}} Bids_{i''j''} - Bidsj''i'' < Bids_{ij} - Bids_{ji} \\ \qquad \text{and } edges^{remain} \mid edges^{flip} > edges^{remain} \mid edges^{flip} \cup \{< i', j' >\} \end{cases}$$

Finally, we check whether flipping every knock-on edge $< i', j' > \in edges^{flip}$ will create any cycle in the graph. If there are no cycles formed, we can replace the pair $< i, j >$ in the current solution. This change will yield a better solution with the objective value calculated as follows: $Objective + (Bids_{ij} - Bids_{ji}) - \sum_{<i',j'>\in edges^{flip}} Bids_{i'j'} - Bids_{j'i'}$.

We continue executing the above procedure until no improvements are reached.

3.2.3 Variable Declaration

Alongside the variables established in the Greedy Method 3.1.1, additional auxiliary variables for the local search procedure are defined as follows:

- $pairsWithHigherBid$: Member pair $< i, j >$ not included in solution but having higher bid than their flipped counterpart $< j, i >$ included in solution.
- $edges^{flip}$: Set of member pairs needed to be flipped if member pair in $pairsWithHigherBid$ requires flipping.

3.2.4 Pseudocode

Algorithm Local Search Algorithm (Part 1: Auxiliary functions)

function FLIPPAIRWITHIMPROVEMENT($< i, j >$)
  $bidIncrease \leftarrow Bids_{ij} - Bids_{ji}$,
  $potentialSolution \leftarrow solution$
  $potentialSolution[potentialSolution.\text{INDEX}(< j, i >)] \leftarrow < i, j >$
  $edges, indegree, outdegree \leftarrow TrimGraph(potentialSolution)$
  if $edges = \phi$ **then**
    return True, $bidIncrease, \phi$
  $edges^{flip}, bidDecrease \leftarrow GetKnockonFlippedPairs(bidIncrease, edges, indegree, outdegree)$
  if $bidDecrease = -1$ **then**
    return False, 0, ϕ
  if $CanBeFlipped(< i, j >, edges^{flip})$ =True **then**
    return True, $bidIncrease - bidDecrease, edges^{flip}$
  return False, 0, ϕ

function CanBeFlipped($< i, j >, edges^{flip}$)
  $potentialSolution \leftarrow currentSolution$
  $potentialSolution[potentialSolution.\text{INDEX}(< j, i >)] \leftarrow < i, j >$
  for each $< i', j' > \in edges^{flip}$ **do**
    $potentialSolution[potentialSolution.\text{INDEX}(< i', j' >)] \leftarrow < j', i' >$
  $edges \leftarrow TrimGraph(potentailSolution)$
  if $edges \neq \phi$ **then**
    return False
  return True

function GetKnockonFlippedPairs($bidIncrease$, $solution$, $indegree$, $outdegree$)
  $totalBidDecrease, edges^{flip} \leftarrow 0, \phi$
  while $edges^{remain} \neq \phi$ **do**
    $edges^{candidate} \leftarrow \{< i, j > \in edges^{remain} \mid Bids_{ij} - Bids_{ji} + totalBidDecrease < bidIncrease\}$
    if $edges^{candidate} = \phi$ **then**
      return $\phi, -1$
    $vertices \leftarrow GetVerticesFromEdges(edges)$
    $edgeDegrees(< i', j' >) \leftarrow \sum_{k' \in i', j'} indegree(k') + outdegree(k') \quad \forall < i', j' > \in edges^{remain}$
    $sortedEdges^{candidate} \leftarrow$ SORT(
      $edges^{candidate}, (vertexDegrees(< i', j' >), Bids_{j'i'} - Bids_{i'j'}), DESC)$
    $foundCandidate \leftarrow$ FALSE
    for each $< i', j' > \in edges^{candidate}$ **do**
      $edges^{remain'} \leftarrow edges^{remain}$
      $edges^{remain'}[edges^{remain'}.\text{INDEX}(< i', j' >)] \leftarrow < j', i' >$
      $edges^{remain'}, indegree, outdegree \leftarrow TrimGraph(edges^{remain'})$
      if $edges^{remain'} \subset edges^{remain}$ **then**
        $totalBidDecrease \leftarrow totalBidDecrease + Bids_{i'j'} - Bids_{j'i'}$
        $edges^{remain} \leftarrow edges^{remain'}$
        $edges^{flip} \leftarrow edges^{flip} \cup \{i', j'\}$
        $foundCandiate \leftarrow$ True
        break
    if $foundCandidate =$ FALSE **then**
      return $\phi, -1$
  return $edges^{flip}, totalBidDecrease$

Algorithm Local Search Algorithm (Part 2: Main Algorithm)

$Solution \leftarrow GreedyConstructionPhase()$
$improved \leftarrow$ True
while $improved$ **do**
  $improved \leftarrow$ False
  $pairsWithHigherBids \leftarrow \{< i, j > | < j, i > \in Solution$ and $Bids_{ij} > Bids_{ji}\}$
  for each $< i, j > \in pairsWithHigherBids$ **do**
    $canBeFlipped, bidIncrease, edges^{flip} \leftarrow FlipPairWithImprovement(< i, j >)$
    if $canBeFlipped =$ True **then**
      $improved \leftarrow$ True
      for each $< i^{'}, j^{'} > \in edges^{flip} \cup \{< j, i >\}$ **do**
        $Solution[Solution.\text{INDEX}(< i^{'}, j^{'} >)] \leftarrow < j^{'}, i^{'} >$
        $Priority(< i', j' >) \leftarrow 0$
        $Priority(< j', i' >) \leftarrow 1$
return $Solution$

3.3 GRASP Method

3.3.1 Variable Declaration

The same as 3.2.3.

3.3.2 Pseudocode

Algorithm GRASP Construction Phase

$Solution \leftarrow \phi$
$CoveredMembers \leftarrow \phi$
$CandidatePairs \leftarrow \{< i, j > | \ i, j \in Members \quad \text{and} \quad Bids_{ij} + Bids_{ji} \neq 0\}$
$Priority(i, j) \leftarrow 0 \quad \forall < i, j > \in CandidatePairs$
while $CandidatePairs \neq \phi$ **do**
  Evaluate $q(< i', j' >, Solution) \quad \forall < i', j' > \in CandidatePairs$
  $CandidatePairs \leftarrow \{< i', j' > \in CandidatePairs \mid q(< i', j' >, S) > 0\}$
  $q^{min} \leftarrow \min\{q(< i', j' >, Solution) \mid < i', j' > \in CandidatePairs\}$ ▷ RCL Procedure
  $q^{max} \leftarrow \max\{q(< i', j' >, Solution) \mid < i', j' > \in CandidatePairs\}$
  $\text{RCL}_{max} \leftarrow \{< i', j' > \in CandidatePairs \mid q(< i', j' >, Solution) \geq q^{max} - \alpha(q^{max} - q^{min})\}$
  $< i, j > \leftarrow$ select $< i', j' > \in$ RCL at random
  $CandidatePairs \leftarrow CandidatePairs \setminus \{< i, j >, < j, i >\}$
  $CoveredMembers \leftarrow CoveredMembers \cup \{i, j\}$
  $Solution \leftarrow Solution \cup \{< i, j >\}$
  $Priority(< i, j >) \leftarrow 1$
return $Solution$

Algorithm GRASP Procedure

$Object^{best}, Solution^{best} \leftarrow 0, \phi$
for $retry = 1$ to MaxIterations **do**
  $Obejctive, Solution \leftarrow 0, \phi$
  $Solution \leftarrow doConstructionPhase()$
  $Solution \leftarrow doLocalSearchPhase(Solution)$
  if $Obejctive > Objective^{best}$ **then**
    $Objective^{best} \leftarrow Obejctive$
    $Solution^{best} \leftarrow Solution$
return $Solution^{best}$

4 Experiment Result Analysis

4.0.1 Experiment Dataset And Environment

16 data files are generated in the folder *src/project/testdata* with prefix *project.{39-44}-1* and *project.45-{1-10}*, containing 39 to 45 members for each dataset.
These experiments were conducted on an Apple MacBook Pro laptop with an Apple M2 chip and 8GB of memory. Solving times may vary slightly on other computers, particularly for the ILP model.

4.1 Parameter Tuning For GRASP

We tested $\boldsymbol{\alpha}$ from 0 to 1 in increments of 0.1, using datasets with 45 members without conducting the local search procedure, and the maximum iterations for the constructive phase is set to 100.

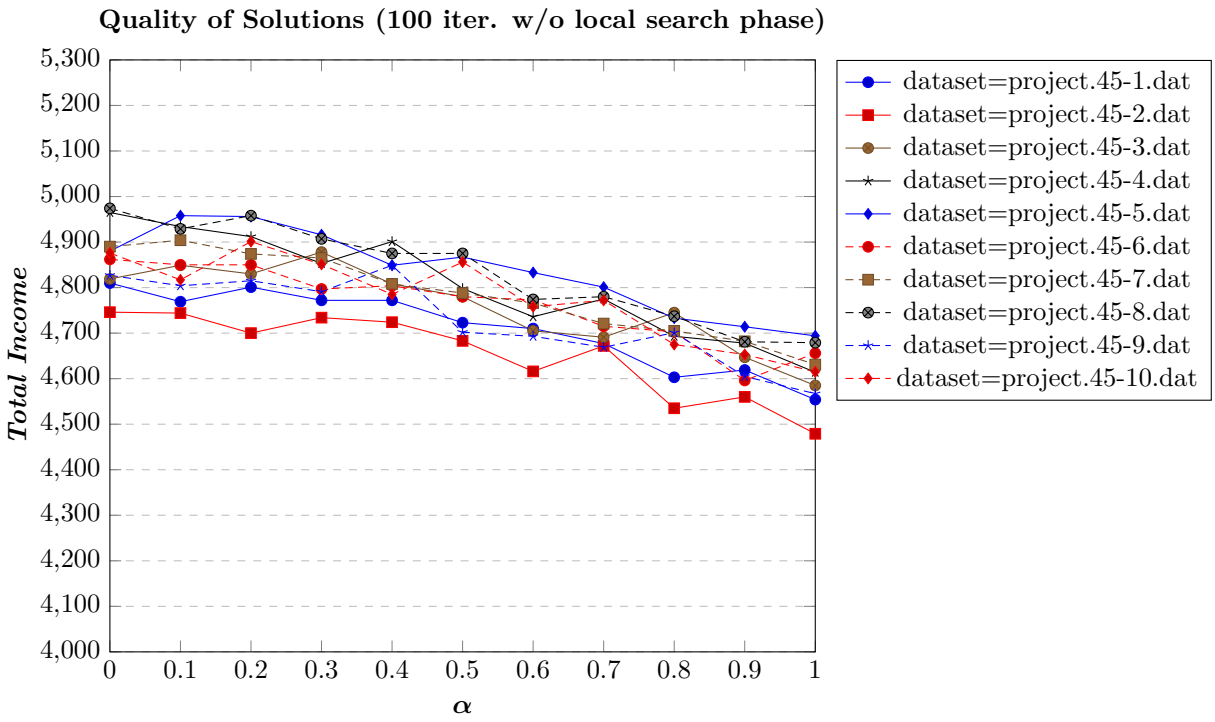

Figure 4.1

Dataset	α for $Solution^{best}$
project.45-1.dat	0
project.45-2.dat	0
project.45-3.dat	0.3
project.45-4.dat	0
project.45-5.dat	0.1
project.45-6.dat	0
project.45-7.dat	0.1
project.45-8.dat	0
project.45-9.dat	0.4
project.45-10.dat	0.2

Table 4.1

As shown in figure Figure 4.1 and table Table 4.1, a better solution is most likely achieved when α is set from 0 to 0.1. **In the following experiments, we will set α equal to 0 when conducting the GRASP method**.

4.2 Solving Time

As shown in Figure 4.2 the ILP model takes approximately 11 to 75 minutes to reach the optimal solution, while the Greedy method requires less than 200 milliseconds. This means the Greedy method is roughly 10000 times faster than the ILP model.
The Local Search method takes about 20 seconds to solve the problem. Additionally, when GRASP is executed with 10 iterations and followed by the Local Search procedure, it takes about ten times longer than the Local Search method alone.
Furthermore, when GRASP is executed with 100 iterations and not followed by the Local Search procedure, it takes around 25 seconds to solve the problem.
As demonstrated in Figure 4.3, the time required to solve the ILP model varies significantly. At

times, we may be fortunate enough to obtain the optimal solution quickly. Specifically, if the number of members is small (fewer than 42), we can effectively use the ILP model to achieve the optimal solution.

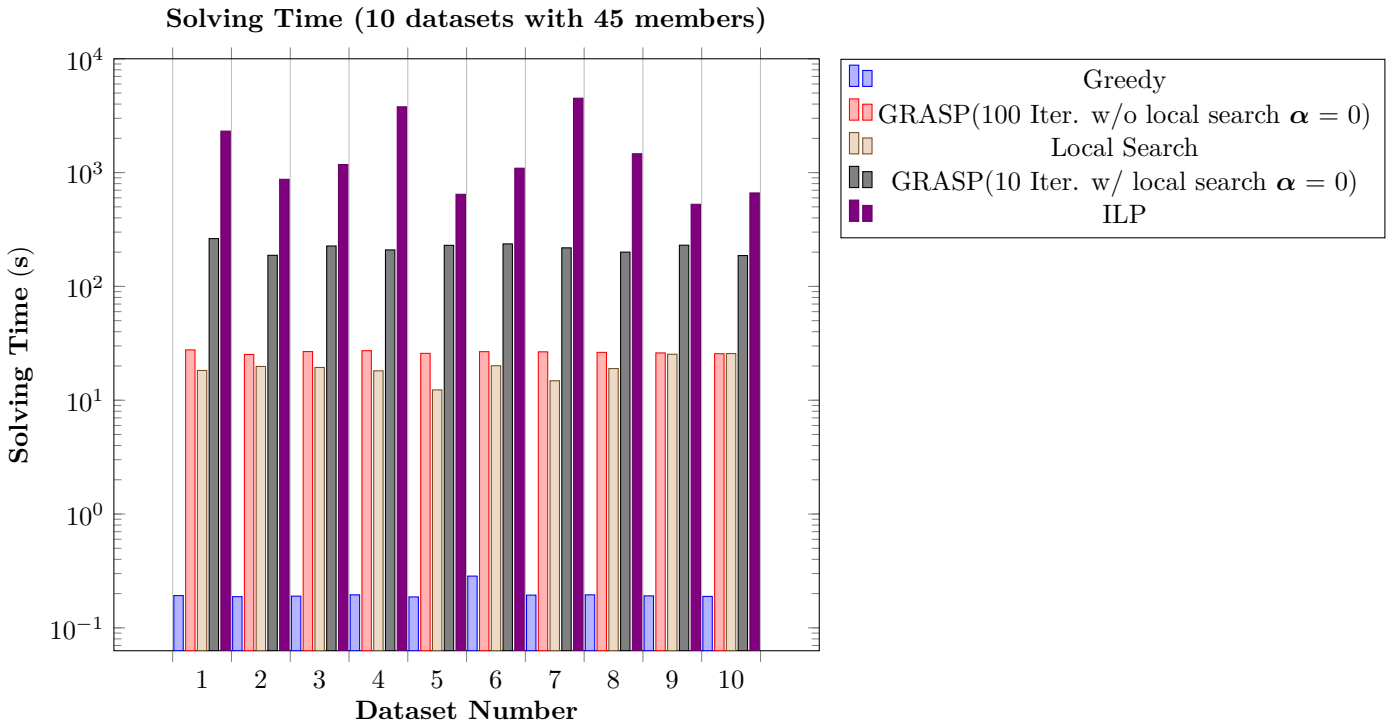

Figure 4.2

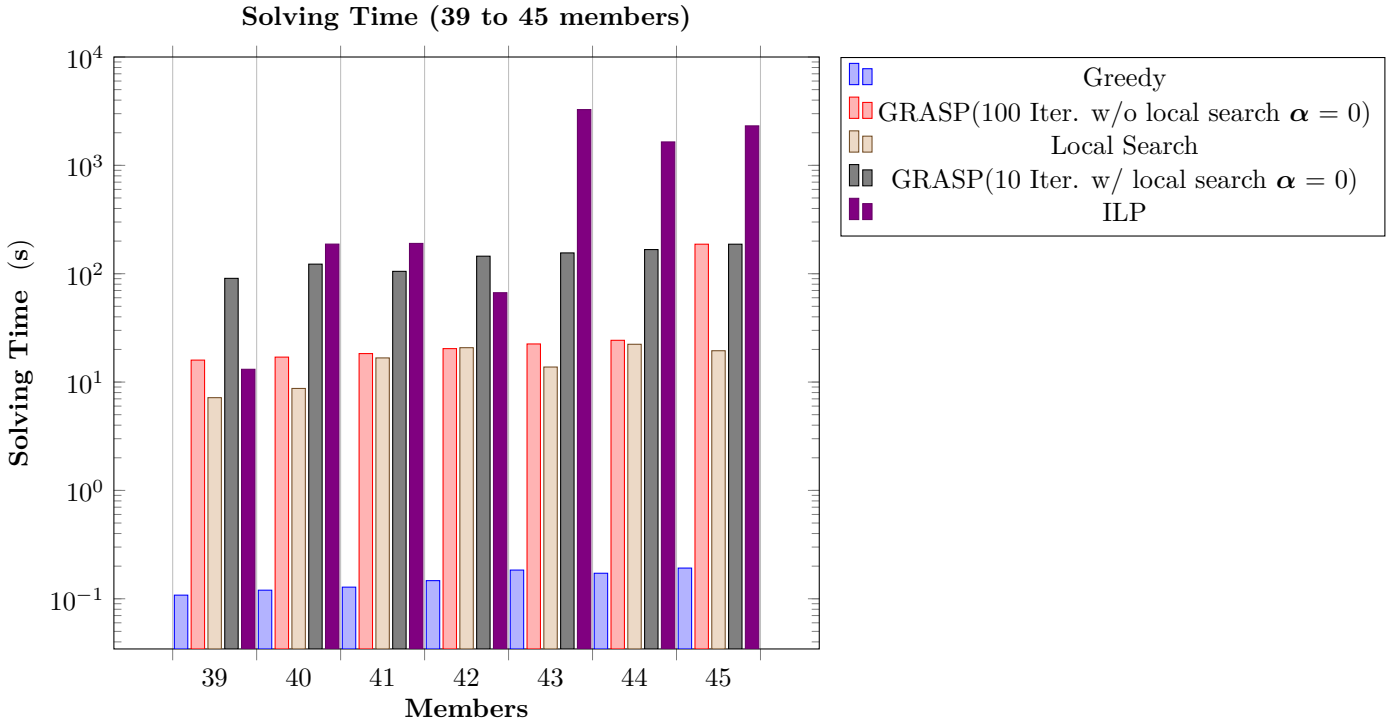

Figure 4.3

4.3 Solving Quality

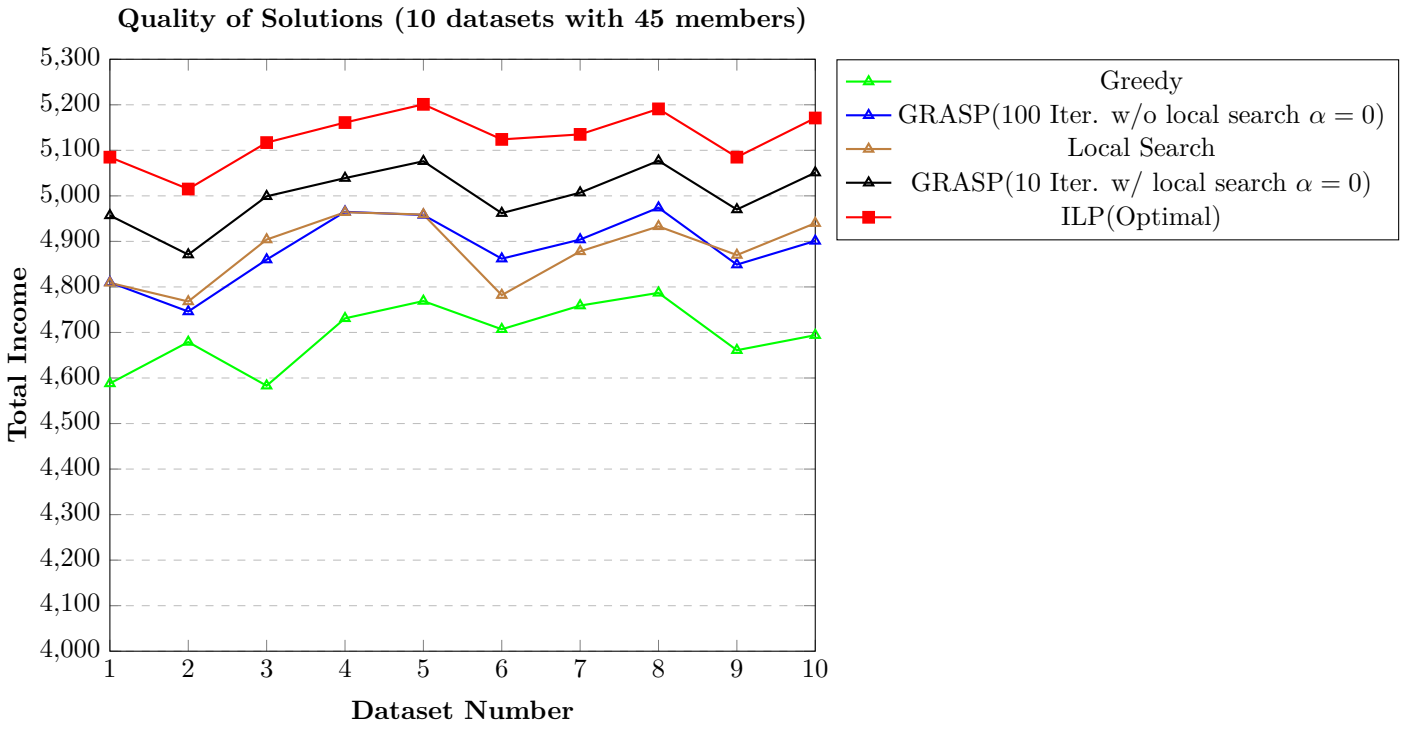

Figure 4.4

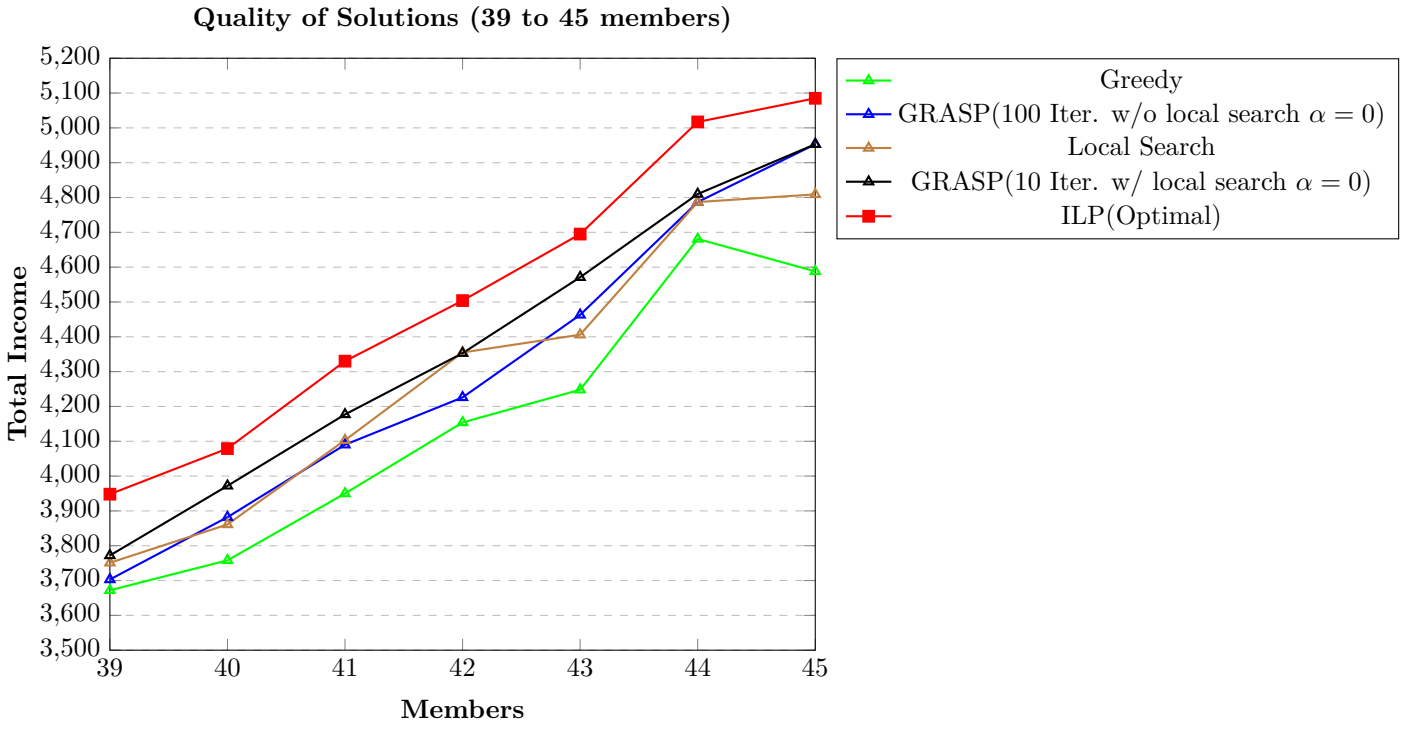

Figure 4.5

Figure 4.4 and Figure 4.5 shows that the Greedy approach yields the worst solution. In contrast, both GRASP (100 iterations without local search) and Local Search provide similar results. Among the

heuristic methods, GRASP (10 iterations with local search) offers the best solution.

4.4 Experiment Conclusion

If the number of members is small (less than 42), we should always apply ILP model to achieve the optimal solution. GRASP with 10 iterations and local search yields the best solution among the heuristic methods, but it requires a significant amount of time to compute compared with other methods. To balance solution quality and solving time, we can use GRASP with 100 iterations and no local search, utilizing an $\boldsymbol{\alpha}$ value between 0 and 0.1 to address the problem, or employ only the local search method.

5 Personal Reflections

Initially, we had limited knowledge of graph theory and were unfamiliar with concepts like topological sorting or directed acyclic graphs. We approached the subject incorrectly at first, but after receiving guidance from our professors and exploring graph theory in more depth, a new world opened up for us. We were amazed by the efficiency of graph algorithms.
For us, the optimization methods used in linear programming represent a significant shift in thinking. These methods can be elegant, and we always strive to pursue optimal solutions in theory. However, in practice, solving problems can sometimes be quite costly. Therefore, for these problems we should employ heuristic methods as a tradeoff. It is essential to experiment and iterate in order to discover better solutions.
We will continue to refine this project by optimizing the heuristic methods, which we believe will reduce solving time and improve solution quality.

6 Acknowledgements

We want to give a big thank you to Prof. Enric Rodriguez and Prof. Luis Velasco for all their guidance and support throughout this report and during the whole course. Your help has truly meant a lot to us!

A Pseudocode of Auxiliary Functions for Directed Graph

Some auxiliary functions are utilized to process directed graphs for the *Greedy*, *Local Search*, and *GRASP* methods.

- The function $TopologicalSort$ implements ***Kahn's algorithm*** to obtain the topological sort of a directed graph, which has $CoveredMembers$ as vertices and $Solution$ (in Greedy method, it is the partial solution) as edges. If there are edges remaining, this indicates that a cycle exists in the graph.
- The function $TrimGraph$ continues to eliminate all vertices with zero indegree or zero outdegree until every remaining edge is part of a cycle.

Algorithm Auxiliary functions For Directed Graph

```
function GetVerticesFromEdges(edges)
    vertices ← ϕ
    for each < i, j >∈ edges do
        vertices ← vertices ∪ i, j
    return vertices

function ConstructNeighbors(vertices, eges)
    neighbors ← ϕ
    for each < i, j >∈ edges do
        neighbors(i) ← neighbors(i) ∪ j
    return neighbors

function TopologicalSort(edges)
    topologicalOrder, indegree ← ϕ, ϕ
    neighbors ← ConstructNeighbors(vertices, edges)
    for each vertex ∈ vertices do
        for each neighbor ∈ neighbors(vetex) do
            indegree(neighbor) ← indegree(neighbor) + 1
    nodesWithZeroIndegree ← {vertex ∈ vertices|indegree(vertex) = 0}
    while nodesWithZeroIndegree ≠ ϕ do
        node ← nodesWithZeroIndegree.POP()
        topologicalOrder ← topologicalOrder ∪ {node}
        for each neighbor ∈ neighbors(node) do
            indegree(node) ← indegree(node) − 1
            if indegree(neighbor) = 0 then
                nodesWithZeroIndegree ← nodesWithZeroIndegree ∪ {neighbor}
    return topologicalOrder
```

```
function TrimGraph(edges)
    indegree, outdegree, previousEdges ← φ, φ, φ
    vertices ← GetVerticesFromEdges(edges)
    neighbors ← ConstructNeighbors(vertices, edges)
    for each vertex ∈ vertices do
        outdegree(vertex) ← |neighbors(vertex)|
        for each neighbor ∈ neighbors(vetex) do
            indegree(neighbor) ← indegree(neighbor) + 1
    nodesWithZeroIndegree ← {vertex ∈ vertices|indegree(vertex) = 0}
    nodesWithZeroOutdegree ← {vertex ∈ vertices|outdegree(vertex) = 0}
    while edges ⊂ previousEdges do
        previousEdges ← edges
        while nodesWithZeroIndegree ≠ φ do
            node ← nodesWithZeroIndegree.POP()
            for each neighbor ∈ neighbors(node) do
                indegree(node) ← indegree(node) − 1
                outdegree(node) ← outdegree(node) − 1
                if < node, neighbor > then
                    edges.REMOVE(< node, neighbor >)
                if indegree(neighbor) = 0 then
                    nodesWithZeroIndegree ← nodesWithZeroIndegree ∪ {neighbor}
                if outdegree(node) = 0 then
                    nodesWithZeroOutdegree ← nodesWithZeroOutdegree ∪ {node}
        while nodesWithZeroOutdegree ≠ φ do
            node ← nodesWithZeroOutdegree.POP()
            for each node′ ∈ neighbors do
                if outdegree(node′) > 0 AND node ∈ neighbors(node′) then
                    outdegree(node′) ← outdegree(node′) − 1
                    indegree(node) ← indegree(node) − 1
                    if < node′, node >∈ edges then
                        edges.REMOVE(< node′, node >)
                    if outdegree(node′) = 0 then
                        nodesWithZeroOutdegree ← nodesWithZeroOutdegree ∪ {node′}
                    if indegree(node) = 0 then
                        nodesWithZeroIndegree ← nodesWithZeroIndegree ∪ {node}
    return edges, indegree, outdegree
```

B Source Code

All code implementations of heuristic methods and ILP model are available at Github Repo. The README provides the explanation of how to use the code to solve the problem.