Accelerating Dynamic Linear Assignment Problem solver using Primal Algorithms

Utsav Majumdar, Samiran Kawtikwar, Rakesh Nagi
Department of Industrial and Enterprise Systems Engineering, 117 Transportation Building, University of Illinois at Urbana-Champaign, Urbana, IL 61801. utsavm2@illinois.edu, nagi@illinois.edu.

This report delves into the realm of Linear Assignment Problems (LAPs) within dynamic environments, wherein costs fluctuate across iterations. Unlike conventional approaches, we investigate a primal algorithmic strategy that leverages previous iteration's assignments and duals to expedite the solution process. By harnessing insights from prior solutions, this method optimizes computational efficiency while maintaining accuracy in addressing dynamically evolving assignment scenarios. The primal algorithm is built by modifying the standard Hungarian algorithm with minimum changes in order to integrate it into the codebase and parallelize it at a later stage. Through rigorous examination and experimentation, we elucidate the efficacy and potential advantages of this accelerated dynamic linear assignment problem solver.

Key words: Linear Assignment Problem; Primal algorithm; Dynamic setting

1. Introduction

Assignment problems, including the Linear Assignment Problem (LAP), are crucial in various fields, optimizing resource allocation to minimize costs or maximize efficiency (Lawler (1963)). However, some real-world scenarios introduce additional constraints, like the Resource Capacitated Assignment Problem. This variation necessitates complex solving approaches such as sub-gradient algorithms with a branch and bound scheme. Here, multiple linear assignment problems are iteratively solved with small changes in the cost matrix.

To address these challenges, a primal algorithm has been developed. Unlike traditional methods requiring fresh solves for each iteration, like the Hungarian algorithm (Munkres (1957)), this algorithm leverages the solution from the previous iteration. By doing so, it accelerates the solving process for the new costs, enhancing efficiency in tackling Resource Capacitated Assignment Problems and similar constrained variants.

The LAP's widespread applications, from scheduling to image matching, underscore its importance. Yet, as industries demand real-time decision-making and solving more constrained problems, dynamic algorithms are essential.

2. Literature Review

The Linear Assignment Problem (LAP) has been extensively studied in the optimization literature due to its broad applicability in diverse fields such as operations research, computer vision, and bioinformatics. This problem, which involves optimally assigning a set of agents to a set of tasks with associated costs or profits, has spurred the development of numerous algorithms over the years.

A groundbreaking algorithm that revolutionized LAP solving is the Hungarian algorithm, also known as the Kuhn-Munkres algorithm (Munkres (1957)), proposed by Kuhn in the mid-20th century. This algorithm, based on the primal-dual method, guarantees optimality and has a polynomial time complexity, making it highly efficient for practical applications.

In the realm of primal approaches to LAP solving, the Balinski-Gomory algorithm stands out. Balinski and Gomory (Balinski and Gomory (1964)) introduced this method, which keeps the solution primal feasible at all times and focuses to achieve dual feasibility, making it suitable for dynamic settings where costs change over time. By leveraging assignments and dual variables from previous iterations, the Balinski-Gomory algorithm efficiently adapts to changing problem instances without the need for a fresh solve.

Another notable contribution to LAP solving is the auction algorithm proposed by Bertsekas (Bertsekas (1990)). This algorithm, inspired by auction theory, offers an efficient approach to finding optimal assignments by iteratively updating prices and allocations.

Dantzig's algorithm, also known as the simplex method, has been adapted to solve LAP by reformulating it as a linear program (Dantzig et al. (1954)) . This algorithm, originally designed for general linear programming, provides another avenue for LAP solving.

Kennington's branch-and-bound algorithm (Kennington and Wang (1991)) is another prominent method for LAP solving. By employing a branch-and-bound scheme, this algorithm systematically explores the solution space to find an optimal assignment.

In summary, LAP has been a subject of intense research, leading to the development of various algorithms catering to different problem settings and requirements. Especially accelerating classical approaches like Hungarian algorithm using GPUs (Date and Nagi (2016)) has been a leading direction of research.

The algorithm discussed in this paper is integerated into the codebase developed by Kawtikwar and Nagi (2024).

3. Formulations

The Linear Assignment Problem can be formulated in terms for a linear program. As with any linear programs, we can have primal and dual formulations.

3.1. Primal Formulation

The Linear Assignment Problem (LAP) can be formulated as follows:

Given:

- n tasks to be assigned.
- n agents to perform the tasks.
- A cost matrix C, where c_{ij} represents the cost of assigning task i to agent j.

The goal is to find the assignment that minimizes the total cost.

The LAP can be mathematically expressed as the following optimization problem:

$$\text{Minimize} \quad \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}$$

subject to the following constraints:

$$\begin{aligned}
&\text{subject to}\\
&\sum_{j=1}^{n} x_{ij} = 1, \quad \text{for } i = 1,2,\ldots,n \quad \text{(Each task is assigned to exactly one agent)},\\
&\sum_{i=1}^{n} x_{ij} = 1, \quad \text{for } j = 1,2,\ldots,n \quad \text{(Each agent is assigned to exactly one task)},\\
&x_{ij} \in \{0,1\}, \quad \text{for } i,j = 1,2,\ldots,n.
\end{aligned}$$

Here, x_{ij} is a binary decision variable, which equals 1 if task i is assigned to agent j, and 0 otherwise.

3.2. Dual Formulation

Duality is an important concept in the theory of optimization. Duality in Linear Assignment Problems (LAPs) reveals a fascinating relationship between two optimization perspectives: the primal and the dual. It ensures that optimal solutions to both problems have equivalent objective function values. This insight not only guarantees solution optimality but also guides the development of efficient algorithms like the Hungarian method. Moreover, duality offers a deeper understanding of LAPs' structure and provides bounds on optimal solutions, making it a vital tool for optimization. The dual formulation of the Linear Assignment Problem (LAP) can be given as follows:

Let u_i and v_j be dual variables associated with the constraints of the LAP.

The dual problem is to maximize the dual objective function subject to certain constraints:

$$\text{Maximize} \quad \sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j$$

subject to the following constraints:

$$\begin{aligned} &\text{subject to} \\ &u_i + v_j \leq c_{ij}, \quad \text{for } i, j = 1, 2, \ldots, n, \\ &u_i, v_j \in \mathbb{R}, \quad \text{for } i, j = 1, 2, \ldots, n. \end{aligned}$$

Here, u_i and v_j represent dual variables corresponding to the constraints of the LAP, and c_{ij} is the cost associated with assigning task i to agent j.

3.3. Complementary Slackness

The complementary slackness conditions for the Linear Assignment Problem (LAP) can be given as follows:

Let x_{ij} be the decision variable representing whether task i is assigned to agent j, and let u_i and v_j be the corresponding dual variables.

The complementary slackness conditions are as follows:

$$\begin{aligned} x_{ij} = 1 \quad \text{implies} \quad & u_i + v_j = c_{ij}, \\ & u_i, v_j \in \mathbb{R}, \quad \text{for } i, j = 1, 2, \ldots, n. \end{aligned}$$

These conditions essentially state that if a task is assigned to an agent, then the sum of the corresponding dual variables should equal the cost of that assignment. If a task is not assigned to an agent, then the sum of the corresponding dual variables should be greater than the cost of that assignment.

By duality theory, a pair of solutions respectively feasible for the primal and the dual is optimal if and only if (complementary slackness)

$$x_{ij}(c_{ij} - u_i - v_j) = 0 \quad (i, j = 1, 2, \ldots, n). \tag{1}$$

The values

$$\bar{c}_{ij} = c_{ij} - u_i - v_j \quad (i, j = 1, 2, \ldots, n) \tag{2}$$

are the linear programming reduced costs. This transformation from C to $\bar{C}$ is a special case of what is known as “admissible transformation”. Indeed, for any feasible primal solution X, the transformed objective function is

$$\begin{aligned} &\sum_{i=1}^{n}\sum_{j=1}^{n}(c_{ij} - u_i - v_j)x_{ij} \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} - \sum_{i=1}^{n} u_i \sum_{j=1}^{n} x_{ij} - \sum_{j=1}^{n} v_j \sum_{i=1}^{n} x_{ij} \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} - \sum_{i=1}^{n} u_i - \sum_{j=1}^{n} v_j, \end{aligned} \tag{3}$$

i.e., for any feasible solution X the objective function values induced by C and $\bar{C}$ differ by the same constant $\sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j$.

4. Approaches to solve the LAP

Two main approaches have been explored for addressing Linear Assignment Problems (LAPs):

- Primal-Dual Hungarian Algorithm (Munkres (1957))
- Primal Balinski-Gomory Approach (Balinski and Gomory (1964))

4.1. Hungarian Algorithm

4.1.1. Overview The Hungarian algorithm, pioneered by Kuhn (1955) and independently by Munkres (1957), stands as one of the cornerstone methods for solving LAPs. Its name stems from the Hungarian mathematicians who made significant contributions to its development.

The Hungarian algorithm operates as a generative algorithm, iteratively refining both primal and dual solutions until an optimal assignment is reached. It begins with no assignments and gradually increases the number of assignments with each iteration. The process continues until all nodes are assigned, resulting in an optimal solution.

One of the distinctive features of the Hungarian algorithm is its efficiency. It is renowned for its speed and widespread applicability in solving LAPs. Moreover, its inherent structure lends itself well to parallelization, facilitating accelerated computation. Ongoing research has further explored its performance characteristics and optimization strategies.

4.1.2. Algorithm As part of this project, a codebase has been developed to implement the sequential Hungarian algorithm using Python and C++. The classical variant of Hungarian algorithm is explored. In this approach, the reduced cost (slack) $c_{ij} - u_i - vj$ is utilized. Based on complementary slackness criteria, we build assignment edges where the reduced cost is 0.

$$x_{ij}(c_{ij} - u_i - v_j) = 0 \quad (i, j = 1, 2, \ldots, n). \tag{4}$$

For $c_{ij} - u_i - v_j = 0$, we can conclude that $x_{ij} = 1$. In other words we can form an assignment. The pseudo-code for the $O(n^2)$ implementation is as follows : The following pseudocode outlines the implementation of the classical Hungarian algorithm.

Parallelize-able setup

It is crucial to make sure that the code is parallelizable. We plan to parallelize this on CUDA. As a result, we have to simplify the code, for it to be compatible with CUDA framework. This necessitates elimination of dynamic memory allocation like using vectors, queues, stacks etc. Also, all of the 2D matrices are flattened into 1D array to simplify the process.

Pre-processing

The bi-partite graph for Cost matrix C can be interpreted as the following :

- **Rows** : Nodes on the Left (L.H.S) of the bi-partite graph
- **Columns** : Nodes on the Right (R.H.S) of the bi-partite graph

Of course we flatten this 2D matrix into 1D. Before the iterations start, the Cost matrix C is pre-processed with row and column reduction to obtain the intial set of assignments.

First we initialize the arrays $rows$ and $\phi\dot{}$The assignments are stored in the array $rows$ where:

- **Index of array** $rows$: R.H.S of the bipartite graph or the column node
- **Value of array** $rows$: L.H.S of the bipartite graph or the row node

Conversely for backtracking purpose we also store the assignments in ϕ.

- **Value of array** ϕ: R.H.S of the bipartite graph or the column node
- **Index of array** ϕ: L.H.S of the bipartite graph or the row node

Initially, we start with no assignments. Hence, the arrays $rows$, $pred$, ϕ are initialized to their sentinel value = -1.

Algorithm 1: Initialization of Arrays

```
• Initialize arrays:
• For $i = 0$ to $SIZE - 1$
•     $rows[i] \leftarrow -1$
•     $pred[i] \leftarrow -1$
•     $\phi[i] \leftarrow -1$
```

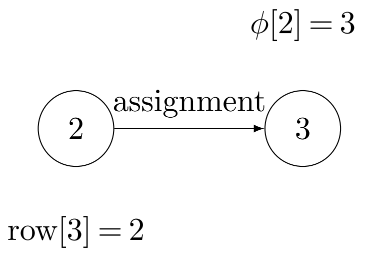


Figure 1 **Assignment with** $rows$ **and** ϕ

We first initialize the labels to their sentinel values.

Algorithm 2: Initialization of Labels

```
• Initialize labels to their sentinel values:
•     $U', SU, SV, LV \leftarrow$ arrays of boolean values of size SIZE
• For $i = 0$ to $SIZE$
•         $U'[i], SU[i], SV[i], LV[i] \leftarrow 0$
```

U', SU, SV, LV will be used for labelling the nodes for the alternating tree.

We start with the pre-processing step, which involves initial assignments. This is done by row and column reduction.

Algorithm 3: Classical Hungarian Algorithm pre-processing

```
Input: Matrix $C$
Output: Optimal assignments $Ar$ and $Ac$
Function classical_hungarian($C$):
    // Initial reduction;
    foreach $i \in \{1, \ldots, n\}$ do
        $Dr[i] \leftarrow \min_j \{C[i,j]\}$;
    end
    foreach $j \in \{1, \ldots, n\}$ do
        $Dc[j] \leftarrow \min_i \{C[i,j] - Dr[i]\}$;
    end
```

In the pre-processed matrix, the edges where reduced cost $c_{ij} - u_i - v_j = 0$ are used as the initial set of assignments. U' denotes the row nodes on L.H.S that are already assigned and needs to be skipped for further iterations. Once the initial set of assignments are obtained, we check for unassigned nodes. The algorithm iterates until all of the unassigned nodes are assigned. We choose an arbitrary unassigned node k on the L.H.S of the bipartite graph.

Algorithm 4: Pre-processed assignments

```
Data: $rows$, $\phi$, $U'$
Result: Result
for $int\ i = 0; i < SIZE; i++$ do
    for $int\ j = 0; j < SIZE; j++$ do
        if $rows[j] == -1$ and $slack[i * SIZE + j] == 0$ then
            $rows[j] = i$;
            $\phi_i = j$;
            $U'[i] = 1$;
            break;
        end
    end
end
```

Algorithm 5: Over-arching classical Hungarian iterations

```
Data: $C$
Result: assigned $rows$
while $|U \cup U'| > 0$ do
    Start with the first unassigned node $k$ where $U'[k] == 0$;
    while $U'[k] == 0$ do
        Run alternating tree;
        int sink = augmenting_path($k, C, u, v, rows, pred, SU, SV, LV, SIZE$);
        if Valid augmenting path found then
            Run backtracking;
        end
        else
            Run Dual Updates;
        end
    end
end
```

Algorithm 6: Augmenting path

```
Function augmenting_path($k, C, u, v, rows, pred, SU, SV, LV, SIZE$):
    for $j \leftarrow 0$ to $SIZE - 1$ do
        $SU[j] \leftarrow SV[j] \leftarrow LV[j] \leftarrow 0$;
    end
    $fail \leftarrow$ false , $sink \leftarrow -1$ , $i \leftarrow k$;
    while $!fail$ and $sink = -1$ do
        $SU[i] \leftarrow 1$;
        for $j \leftarrow 0$ to $SIZE - 1$ do
            if $LV[j] = 0$ and $(C[i * SIZE + j] - u[i] - v[j]) = 0$ then
                $pred[j] \leftarrow i$ , $LV[j] \leftarrow 1$;
            end
        end
        if there is no node $j$ where $LV[j] = 1$ and $SV[j] = 0$ then
            $fail \leftarrow$ true;
        end
        else
            for $j \leftarrow 0$ to $SIZE - 1$ do
                if $LV[j] = 1$ and $SV[j] = 0$ then
                    $SV[j] \leftarrow 1$;
                    if $rows[j] = -1$ then
                        $sink \leftarrow j$;
                    end
                    else
                        $i \leftarrow rows[j]$ , break;
                    end
                end
            end
        end
    end
    return $sink$;
```

Algorithm 7: Dual update

```
Data: $SU, LV, C, u, v, SIZE$
Function dual_update($SU, LV, C, u, v, SIZE$):
    Save $dmin$ as Most minimum reduced cost where SU=1 and LV=0
    $delta \leftarrow dmin(SU, LV, C, u, v, SIZE)$;
    for $i \leftarrow 0$ to $SIZE - 1$ do
        if $SU[i] = 1$ then
            $u[i] \leftarrow u[i] + delta$;
        end
    end
    for $j \leftarrow 0$ to $SIZE - 1$ do
        if $LV[j] = 1$ then
            $v[j] \leftarrow v[j] - delta$;
        end
    end
```

Algorithm 8: Back tracking

```
Data: $U', sink, pred, rows, \phi, k$
Function back_tracking($U', sink, pred, rows, \phi, k$):
    $U'[k] \leftarrow 1$;
    $b \leftarrow sink$;
    $a \leftarrow -1$;
    while true do
        $a \leftarrow pred[b]$;
        $rows[b] \leftarrow a$;
        $h \leftarrow \phi[a]$;
        $\phi[a] \leftarrow b$;
        $b \leftarrow h$;
        if $a = k$ then
            break;
        end
    end
```

Steps on the algorithm after pre-processing

1. **Augmenting path** : An arbitary unassigned node k is chosen from L.H.S of the bipartite graph. The goal is to find alternate assigned and unassigned edges until a node in R.H.S is reached. This node is the *sink* and indicates the existence of an augmenting path.

2. **Back-tracking** : If an augmenting path is found that ends in a sink, we can trace back from the sink to the source while assigning and unassigning altenate edges. This would lend us an extra assigned edge

3. **Dual update** : If an augmenting path is not found, we update the duals. Considering all the edges that connect scanned nodes in L.H.S and unscanned nodes in R.H.S, we consider the most minimum reduced cost as *delta* . We add *delta* for duals u in all scanned L.H.S nodes and subtract *delta* for duals v in all scanned R.H.S nodes.

4.2. Balinski-Gomory Algorithm

4.2.1. Overview The Balinski-Gomory algorithm developed by Balinski and Gomory (1964) is an approach that always maintains primal feasibility. The algorithm strives to make the infeasible dual feasible reach optimality. This is not a generative algorithm. Here all nodes are assigned at all times and the algorithm works by interchanging the assignments among the nodes until dual feasibility is met. Through our study, we notice that Balinski-Gomory algorithm and Hungarian algorithm have similarities that can be leveraged. This is especially useful since research has already accelerated Hungarian algorithm in the parallel domain (Date and Nagi (2016)). Our objective is to make sure we obtain the same results as the Balinski algorithm with minimum changes in the Hungarian algorithm, so that Balinski algorithm can also be parallelized.

4.2.2. Algorithm We start the initial assignment arbitrarily. Here, we have considered the diagonal where $rows[i] = i$ and each element in L.H.S is assigned to the same cardinal element in R.H.S.

Algorithm 9: Initialization

```
Data: $C, SIZE$
Result: $v, u, rows, pred$
for $i \leftarrow 0$ to $SIZE - 1$ do
    $v[i] \leftarrow C[i][i]$;
    $u[i] \leftarrow 0$;
    $rows[i] \leftarrow pred[i] \leftarrow i$;
end
```

Starting edge from node k to node l corresponds to the most negative reduced cost $c_{ij} - u_i - v_j$.

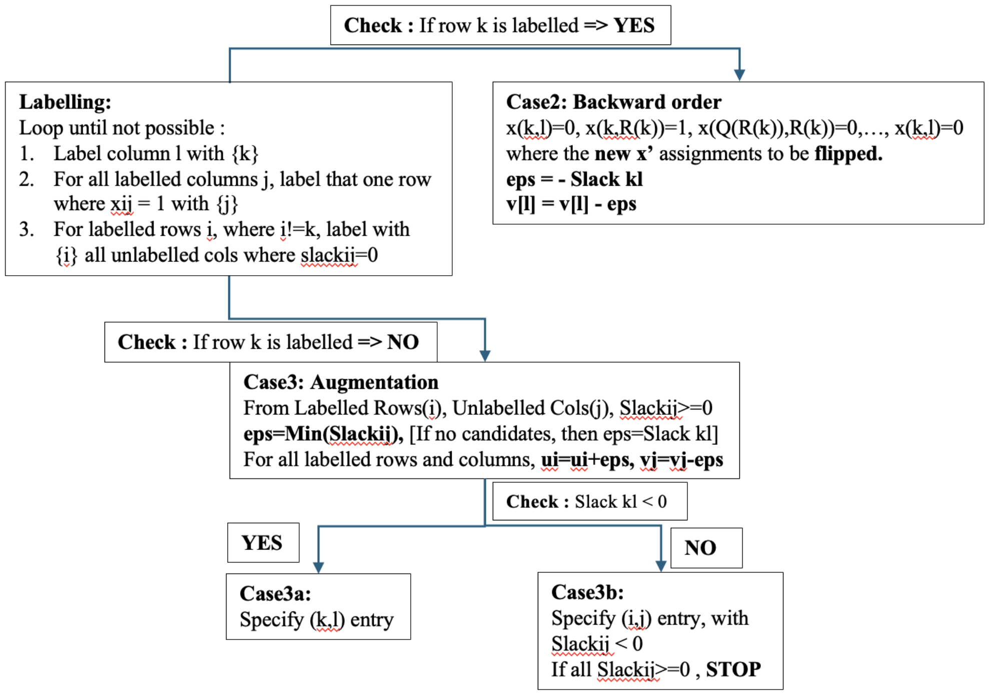

Check : If row k is labelled => **YES**
Labelling:
Loop until not possible :
1. Label column l with {k}
2. For all labelled columns j, label that one row where xij = 1 with {j}
3. For labelled rows i, where i!=k, label with {i} all unlabelled cols where slackii=0
Case2: Backward order
x(k,l)=0, x(k,R(k))=1, x(Q(R(k)),R(k))=0,…, x(k,l)=0
where the **new x'** assignments to be **flipped.**
eps = - Slack kl
v[l] = v[l] - eps
Check : If row k is labelled => **NO**
Case3: Augmentation
From Labelled Rows(i), Unlabelled Cols(j), Slackij>=0
eps=Min(Slackij), [If no candidates, then eps=Slack kl]
For all labelled rows and columns, **ui=ui+eps, vj=vj-eps**
Check : Slack kl < 0
YES
NO
Case3a:
Specify (k,l) entry
Case3b:
Specify (i,j) entry, with Slackij < 0
If all Slackij>=0 , **STOP**

Figure 2 **Balinski Algorithm flow**

Our approach differs from the method used by Balinski and Gomory (1964) in the sense that we do not use the Q, R labelling system. Our approach tries to stick closely with the established Hungarian codebase's labelling system and build the algorithm with as few changes as possible. Hence we would want to recycle some of the approaches of Hungarian algorithm like Augmenting path, Backtracking and Dual update with minimum modifications.

Below we notice that the only change in the over-arching algorithm, compared to Hungarian algorithm are :

- Termination criteria
- Augmenting cycle formation instead of Augmenting path

Algorithm 10: Over-arching Balinski-Gomory iterations

```
Data: $C$
Result: assigned $rows$
while there exists a negative reduced cost do
    Start with the first unassigned node $k$ where $U'[k] == 0$;
    Run alternating cycle;
    int sink = augmenting_cycle($k, C, u, v, rows, pred, SU, SV, LV, SIZE$);
    if Valid augmenting cycle found then
        Run backtracking;
    end
    else
        Run Dual Updates;
    end
end
```

For the alternating cycle, we first start from node k in L.H.S and reach node l in R.H.S. We traverse the assignment from node l to a node k' in L.H.S.

We run a modified B.F.S algorithm at this stage where we jump across alternate nodes through assignments based on the *rows* array. Basically, the R.H.S nodes act as jumping points to the next assignment in L.H.S automatically. Our aim in the B.F.S is to search return from k' to k.

Once this is achieved we will end up with a cycle that starts and ends in k. This can be used to alternately assign and unassign edges in the cycle to get new sets of assignments.

We explore the algorithms below and highlight their similarities as compared to the Hungarian algorithm.

Algorithm 11: Augmenting cycle

```
Function augmenting_cycle($k, l, C, u, v, rows, pred, SU, SV, LV, SIZE$):
    for $j \leftarrow 0$ to $SIZE - 1$ do
        $SU[j] \leftarrow SV[j] \leftarrow LV[j] \leftarrow 0$;
    end
    $fail \leftarrow$ false , $sink \leftarrow -1$ , $i \leftarrow k$ , $bool * visited =$ new bool[$SIZE$]; ;
    for $i \leftarrow 0$ to $SIZE - 1$ do
        $visited[i] \leftarrow 0$;
    end
    while $!fail$ and $i \neq k$ do
        $SU[i] \leftarrow 1$;
        for $j \leftarrow 0$ to $SIZE - 1$ do
            if $LV[j] = 0$ and $(C[i * SIZE + j] - u[i] - v[j]) = 0$ then
                $pred[j] \leftarrow i$ , $LV[j] \leftarrow 1$;
            end
        end
        if there is no node $j$ where $LV[j] = 1$ and $SV[j] = 0$ then
            $fail \leftarrow$ true;
        end
        else
            for $j \leftarrow 0$ to $SIZE - 1$ do
                if $LV[j] = 1$ and $SV[j] = 0$ then
                    $SV[j] \leftarrow 1$;
                    $i \leftarrow rows[j]$, break;
                end
            end
        end
    end
    $path\_found \leftarrow bfs(start, end, visited, C, u, v, rows, pred, SIZE)$; ;
    if $i == k$ then
        $path\_found \leftarrow$ true;;
    end
    return $path\_found$
```

Algorithm 12: Modified BFS

```
Function bfs($start, end, visited, C, u, v, rows, pred, SIZE$):
    $path\_found \leftarrow$ false; , $int * phi =$ new int[$SIZE$]; , for $i \leftarrow 0$ to $SIZE - 1$ do
        $phi[rows[i]] \leftarrow i$;
    end
    $intcurrent\_index \leftarrow 0 intnext\_index \leftarrow 0 int * nodes\_to\_visit =$ new int[$SIZE$];
    $nodes\_to\_visit[next\_index++] \leftarrow start$ , $visited[start] \leftarrow$ true;;
    while $current\_index < next\_index$ and $path\_found ==$ false do
        $int\ node \leftarrow nodes\_to\_visit[current\_index++]$; , for $j \leftarrow 0$ to $SIZE - 1$ do
            if !visited[rows[j]] then
                if near_zero($C[node][j] - u[node] - v[j]$) == 1 then
                    $visited[rows[j]] \leftarrow$ true; , $pred[j] \leftarrow node$; , $nodes\_to\_visit[next\_index +$
                    $+] \leftarrow rows[j]$; , if $rows[j] == end$ then
                        $path\_found \leftarrow$ true; , break;;
                    end
                end
            end
        end
    end
    $bool * change =$ new bool[$SIZE$]; for $i \leftarrow 0$ to $SIZE - 1$ do
        $change[i] \leftarrow$ false;;
    end
    if $path\_found ==$ true then
        $int\ j \leftarrow phi[end]$; while $pred[j]! = start$ do
            $change[j] \leftarrow$ true; , $j \leftarrow phi[pred[j]]$;;
        end
        $change[j] \leftarrow$ true; for $i \leftarrow 0$ to $SIZE - 1$ do
            if !change[i] then
                $pred[i] \leftarrow rows[i]$;;
            end
        end
    end
    if $path\_found ==$ false then
        for $i \leftarrow 0$ to $SIZE - 1$ do
            $pred[i] \leftarrow rows[i]$;;
        end
    end
    return path_found;;
```

The Dual update and Back-tracking are also quite similar to the Hungarian algorithm.

Algorithm 13: Dual update

```
Data: $SU, LV, C, u, v, SIZE$
Function dual_update($SU, LV, C, u, v, SIZE$):
    Save $dmin$ as Most minimum POSITIVE reduced cost where SU=1 and LV=0
    $delta \leftarrow dmin(SU, LV, C, u, v, SIZE)$;
    for $i \leftarrow 0$ to $SIZE - 1$ do
        if $SU[i] = 1$ then
            $u[i] \leftarrow u[i] + delta$;
        end
    end
    for $j \leftarrow 0$ to $SIZE - 1$ do
        if $LV[j] = 1$ then
            $v[j] \leftarrow v[j] - delta$;
        end
    end
    if $slack[k][l] >= 0$ then
        $(k, l)$ as the nodes along the edge with the most negative reduced cost
    end
```

Algorithm 14: Back tracking

```
Data: $U', sink, pred, rows, \phi, k$
Function back_tracking($U', sink, pred, rows, \phi, k$):
    $rows[i] \leftarrow pred[i] \forall i$;
    $delta \leftarrow slack[k][l]$;
    $v[l] \leftarrow v[l] - delta$;
    $(k, l)$ as the nodes along the edge with the most negative reduced cost
```

5. Comparison

We find that there is a lot of equivalence between Hungarian algorithm and Balinski-Gomory algorithm. This includes :

- Same mechanism in alternating augmenting path
- Same backtracking mechanism
- Same dual update procedure

However, each of these is punctuated by small differences. Below we have mentioned a list of differences :

Table 1 Comparison

Hungarian Algorithm	**Balinski-Gomory Algorithm**
Starts with no assignment, followed by pre-processing to assign few nodes. (Refer Algorithm 3)	Starts with complete assignment with every node assigned arbitrarily.
Every node once assigned, does not change its assignment and it concretized.	Every node once assigned, does not change its assignment and it concretized.
Primal and Dual Feasibility maintained throughout the algorithm.	Only Primal Feasibility maintained throughout the algorithm.
Termination occurs when all nodes are assigned. (Refer Algorithm 5)	Termination occurs when there is no negative reduced cost. (Refer Algorithm 10)
Reduced cost is always positive.	Reduced cost can be negative.
Need to find the minimum reduced cost in Dual update step. (Refer Algorithm 7)	Need to find the minimum positive reduced cost in Dual update step. (Refer Algorithm 13)
Augmenting path consists of alternating assigned and unassigned edges.(Refer Algorithm 6)	Augmenting path is a cycle with alternate assigned and unassigned edges.(Refer Algorithm 11)
Starting node for augmenting path procedure can be any unassigned node.	Starting node for augmenting path procedure is preferably the one with the most negative reduced cost.
Fast algorithm and converges in fewer iterations	Slower algorithm and converges after more iterations
Cannot leverage previous solution in dynamic setting	Capable of leveraging previous solution in dynamic setting

6. Dynamic Setting

Our main idea is to run the LAP on a dynamic setting. Since at every iteration the cost matrix changes slightly, there is a possibility of leveraging a previous iteration's solutions to get faster convergence.

We have tested the Balinski and Hungarian algorithms and tested how each perform from scratch. We notice that Hungarian is around 25x faster thatn Balinski at smaller matrices of size 50 and 300x faster than Balinski for matrices of size 2500. This makes the Balinski algorithm's fresh solve unviable. In the figure below, we notice how much better Hungarian performs comapred to Balinski at higher sizes and ranges of matrix. This is due to the fact that Hungarian concretizes assignments in every iteration, which allows it to have less nodes to work with in subsequent steps, in contrast with the Balinski algorithm, which goes through a larger set of nodes in every iteration.

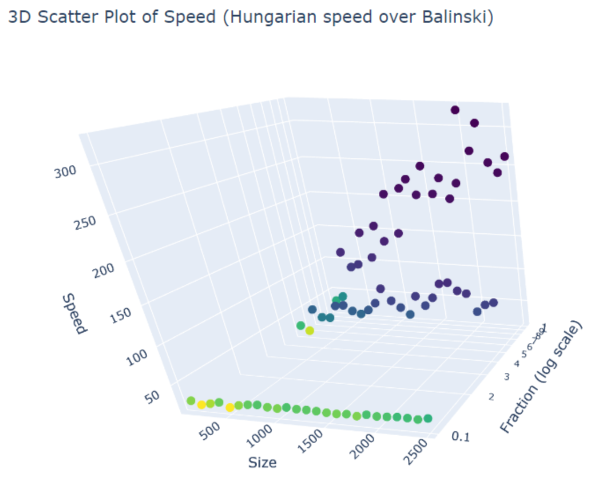


Figure 3 Hungarian vs Balinski Initial solve speed

However, we try to take the best of both worlds by combining an initial fresh solve of Hungarian algorithm followed by a re-solve by Balinski algorithm.

6.1. Approach

1. Solve LAP for cost matrix C using Hungarian algorithm
2. Record the assignment stored in $rows$ array and the duals u and v
3. Develop a noise matrix. Each value in the noise matrix is random a percentage change on the cost matrix. We have also included a noise sparsity matrix that dictates the percentage of randomly chosen values in the cost matrix that will change.

4. Integrate the noise matrix to change values in the cost matrix to get new costs.

5. Use the assignment and duals in the previous solution to use as the starting condition for the new LAP solve using Balinski-Gomory algorithm. We also update the duals to maintain primal feasibility, based on δ change in the Cost matrix.

Algorithm 15: Updating initial duals based on noise

```
• Input: Array $u$ of size $n$, array $v$ of size $m$, array $rows$ of size $SIZE$, matrix $C$
• Output: Updated arrays $u$ and $v$
• For $i = 0$ to $SIZE - 1$
•     $u[rows[i]] \leftarrow u[rows[i]] + \lceil C_{\delta_{rows[i]}[i]}/2 \rceil$
•     $v[i] \leftarrow v[i] + \lfloor C_{\delta_{rows[i]}[i]}/2 \rfloor$
```

6. Solve the modified cost using Hungarian algorithm fresh solve to use as a control benchmark to compare speeds.

7. We iterate this over multiple sizes of matrices, with multiple range of values and different noise magnitudes and sparsity of noise induced.

8. The trend of solving speeds are compared to check the conditions for which each algorithm performs better than the other.

6.2. Experiment

The code developed is integrated into the same codebase developed by Samiran (Kawtikwar and Nagi (2024) to ensure parity. The cost generator with the fraction range used by Kawtikwar and Nagi (2024) has been used for the randomly generated cost matrices. We run the algorithm for the following conditions :

- Cost matrices C of sizes ranging from 100 to 4000, with steps of increase of 100
- Fraction Range for the values {0.01, 0.05, 0.1, 0.5, 1, 5, 10}
- Noise Magnitude range $\{U(-0.05, 0.05), U(-0.1, 0.1), U(-0.15, 0.15), (-0.2, 0.2)\}$
- Noise Sparsity $\{\ 0\%, 20\%, 40\%, 60\%, 80\%, 100\%\ \}$

All of these parameter combinations are iterated over to test performance of both algorithms. The product of noise magnitude and noise sparsity is considered as the parameter *noise* in this case.

7. Computational Results

Comparing Balinski Resolve and Hungarian fresh solve : We plot the difference in time for solving the LAPs. $\delta = Time_{Hungarian} - Time_{Balinski}$ We notice that for increase in fraction range, Hungarian algorithm is out-performing Balinski algorithm. In the figure below Hungarian is represented by 'x' and Balinski by 'o'.

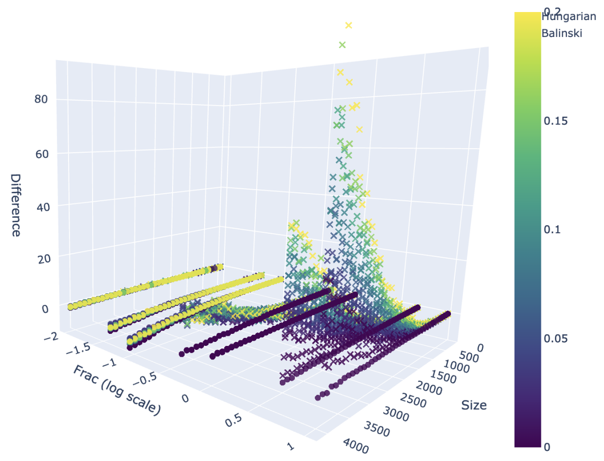


Figure 4 Performance of the algorithms

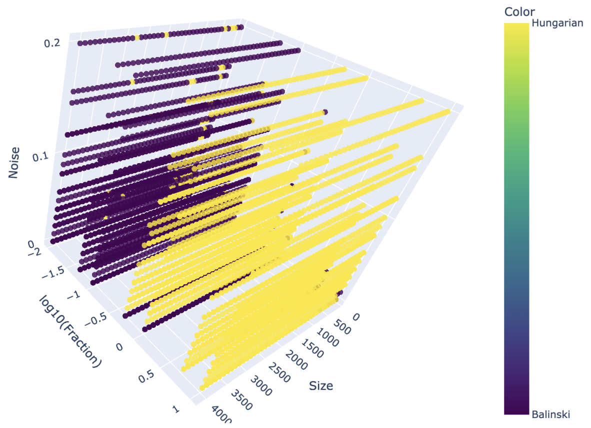


Figure 5 Another representation of cases out-performance by each algorithm

To elucidate, below is a grid of performances of algorithm for a fraction range of 0.01 and varying noises. In the case below, we notice that Balinski outperforms Hungarian in general. For higher sizes of matrix, the δ of outperformance increases for Balinski.

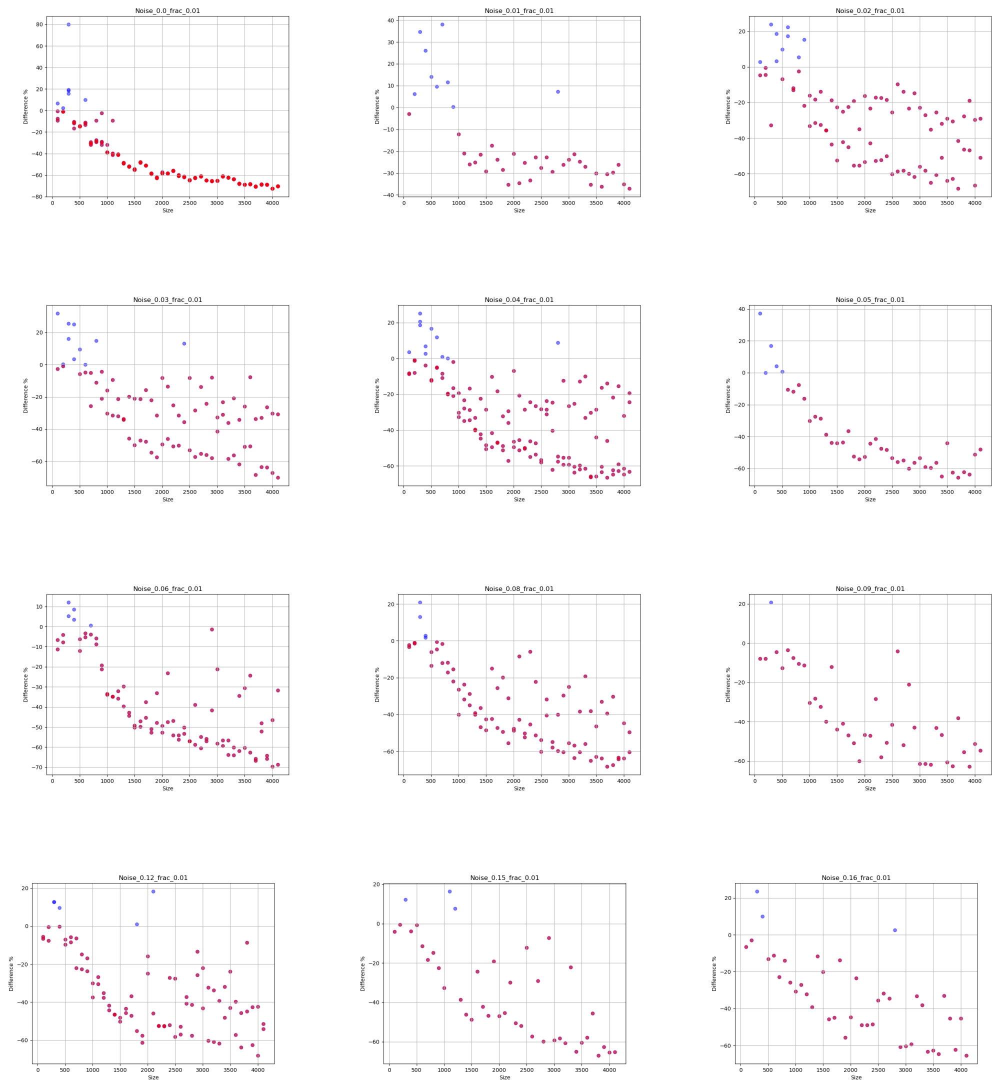


Figure 6 Grid for Frac 0.01 for different noises

We also notice that for higher size of matrices, δ in the performance's susceptibility to noise increases drastically.

Now, we focus on the Balinski's performance.

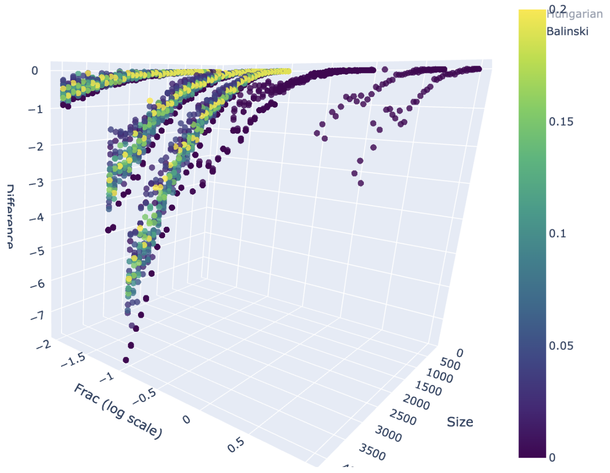


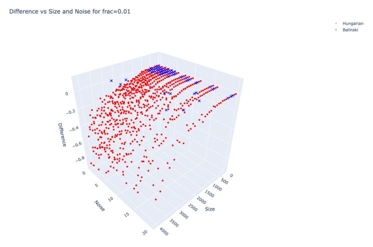
Figure 7 Performance of the Balinski

The speedup factor for Balinski is relatively less compared to Hungarian algorithm. For higher ranges of fraction and larger matrix sizes, the Hungarian is $80secs$ faster than Balinski. However, as per our experiments, we notice a maximum of $7secs$ speed up for Balinski algorithm compared to Hungarian.

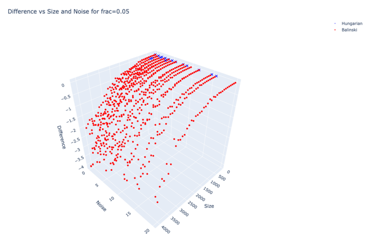
We notice that Balinski is out-performing Hungarian algorithm especially well in fraction range 0.1. We also notice that for less fraction range, Balinski is more resilient to high noises and out-performs Hungarian. The δ of out-performance increases till fraction range 0.1 and falls off quickly.

Hypothesis : For higher matrix sizes, δ in Balinski's out-performances drastically increases compared to Hungarian. However the frequency of this occurring decreases with increase in fraction range. Had we tested for matrices larger than 4000, we might have witnessed few cases where Balinski outperforms Hungarian by a difference $> 7s$.

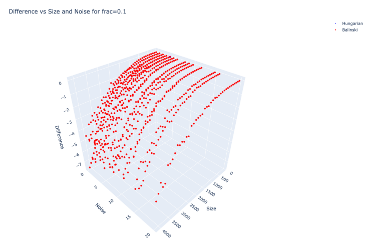
Here is another representation :



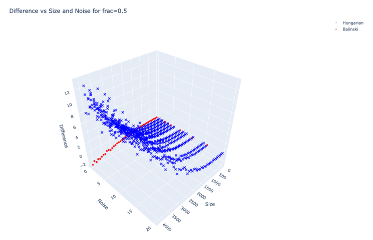
(a) Frac:0.01



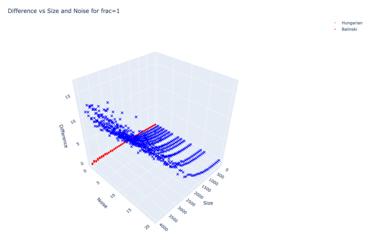
(b) Frac:0.05



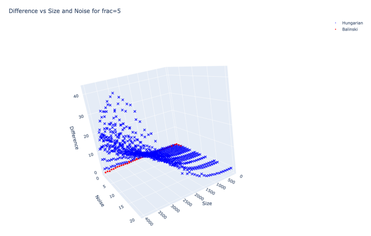
(c) Frac:0.1



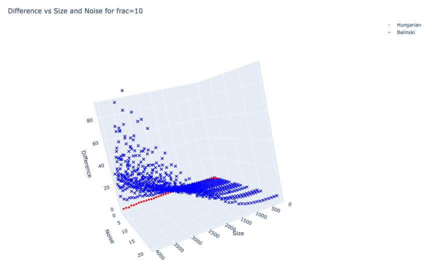
(d) Frac:0.5



(e) Frac:1



(f) Frac:5



(g) Frac:10

Figure 8 Difference in speed for different fraction ranges

Since there seems to be an abrupt shift from 0.1 to 0.5, we break down the transition to understand the change. It is noticed that the transition is noticed at fraction range 0.4.

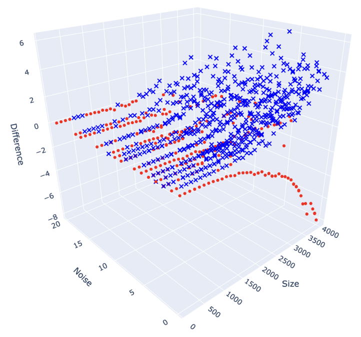


Figure 9 Fraction range 0.4

Investigating further in fraction 0.4 with different noises.

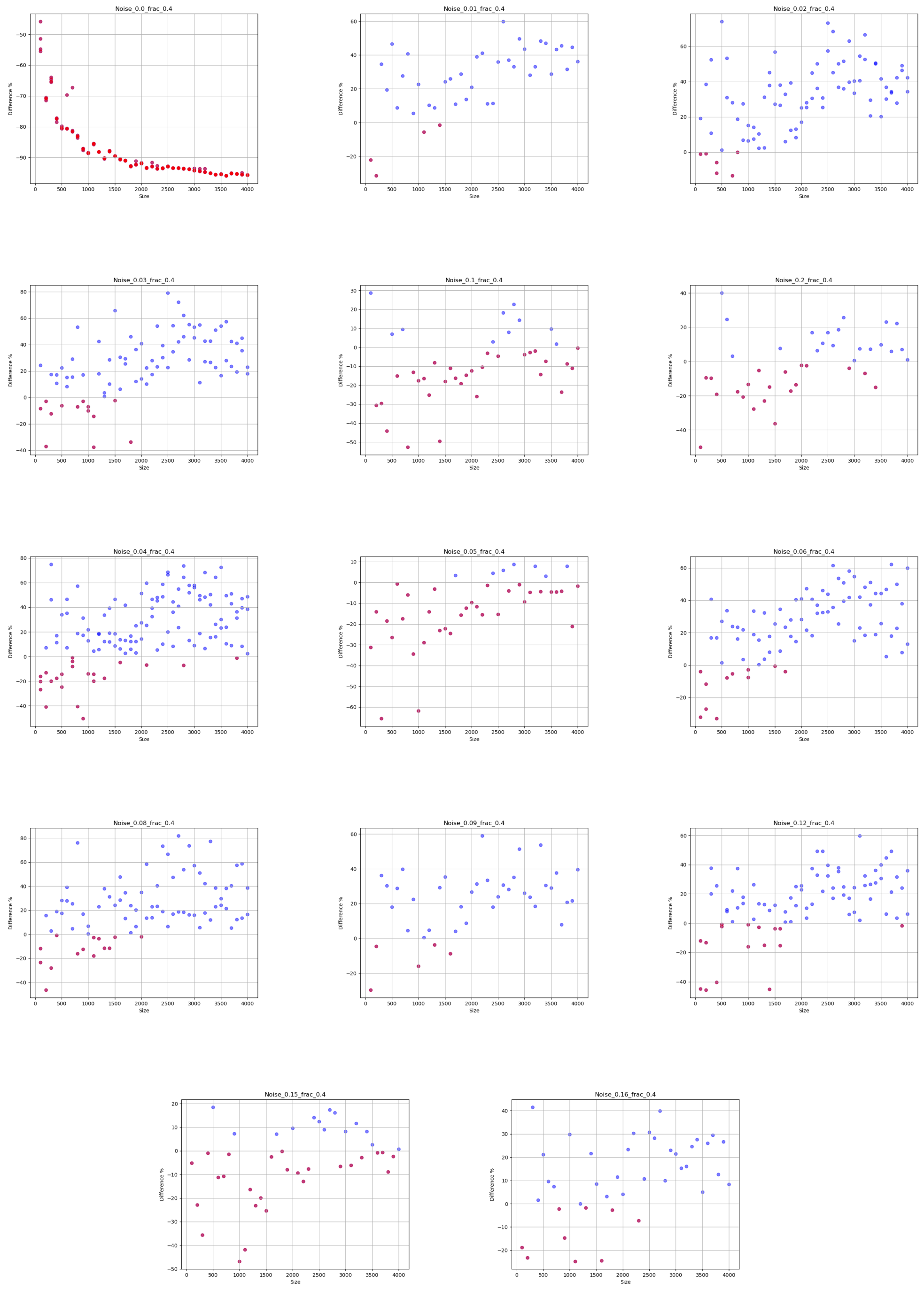


Figure 10 **Grid for Frac 0.4 for different noises**

8. Conclusion

The strategy of solving LAP initially by the Hungarian algorithm and re-solving it by the Balinski-Gomory algorithm is a valid approach for certain size and ranges of matrices. This would depend on the magnitude and sparsity of δ throughout the cost matrix at each iteration. We notice that especially for fraction range of 0.1 there is a considerable increase in speedup factor $(Time_{Hungarian}/Time_{Balinski})$

Table 2 Balinski Speed comparison with Hungarian

Size	Frac	Noise range	Noise Density	Balinski Time	Hungarian Time	SpeedFactor
3600	0.1	10%	100%	0.29	4.77	16.48
3500	0.1	10%	60%	0.33	4.53	13.67
3400	0.1	10%	80%	0.29	3.86	13.48
3700	0.1	5%	60%	0.41	5.38	13.20
3700	0.1	5%	100%	0.39	4.92	12.60

The code built for the Balinski re-solve is integrated into the code base by Kawtikwar and Nagi (2024) and is a modified version of the already established Hungarian algorithm. Since the Hungarian algorithm is already parallelized in CUDA, our future work would focus on parallelizing the Balinski-Gomory algorithm as well. Currently in the sequential paradigm the performances for Balinski-Gomory and Hungarian is comparable for specific cases of size and range of matrices. This shows promise that if it is transformed into the parallel domain and tested, Balinski-Gomory re-solve strategy can be utilized to accelerate Linear Assignment problems. This would be highly beneficial since constrained LAP solvers that use algorithms like sub-gradient branch and bound schemes, with requirements of multiple iterative LAP solutions, can be accelerated to a large extent.

References

Balinski, ML, RE Gomory. 1964. A primal method for the assignment and transportation problems. *Management Science* **10** 578–593.

Bertsekas, Dimitri P. 1990. The Auction algorithm for assignment and other network flow problems: A tutorial. *Interfaces* **20** 133–149.

Dantzig, G., R. Fulkerson, S. Johnson. 1954. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America* **2** 393–410. URL `http://www.jstor.org/stable/166695`.

Date, Ketan, Rakesh Nagi. 2016. GPU-accelerated Hungarian algorithms for the Linear Assignment Problem. *Parallel Computing* **57** 52–72. doi: http://dx.doi.org/10.1016/j.parco.2016.05.012.

Kawtikwar, Samiran, Rakesh Nagi. 2024. Hylac: Hybrid linear assignment solver in cuda. *Journal of Parallel and Distributed Computing* **187** 104838. doi: https://doi.org/10.1016/j.jpdc.2024.104838. URL `https://www.sciencedirect.com/science/article/pii/S0743731524000029`.

Kennington, Jeffery L, Zhiming Wang. 1991. An empirical analysis of the dense assignment problem: Sequential and parallel implementations. *ORSA Journal on Computing* **3** 299–306.

Kuhn, Harold W. 1955. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly* **2** 83–97.

Lawler, Eugene L. 1963. The Quadratic Assignment Problem. *Management Science* **9** 586–599.

Munkres, James. 1957. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial & Applied Mathematics* **5** 32–38.