

# Judging System

[Judging system](#)

[Introduction](#)

[Pairwise Comparisons](#)

[A note on applications](#)

[Inspiration](#)

[Algorithmic Approach](#)

[MFAS is NP-Hard proof](#)

[Approximating MFAS - Key Results](#)

[Creating Synthetic Data](#)

**[Explaining the lolib dataset and its origin](#)**

**[Experience with the dataset](#)**

**[The need to change normalized matrices and the dataset originally used](#)**

**[The normalized version of LOlib dataset can be found here :](#)**

[Exact Methods](#)

[Brute Force](#)

[A Dynamic Programming Approach](#)

[Code:](#)

[Improvement:](#)

[Mixed Integer Programming \(MIP\) Formulation](#)

[Heuristics](#)

[2-OPT](#)

**[Naive with insertion as a local move](#)**

[Kernighan-Lin 2](#)

[The Heuristic of Chanas & Kobylanski](#)

**[SORT](#)**

**[Reverse](#)**

[These are values produced by naive insertion heuristic, KL2, Chanas on Lolib initial dataset](#)

[Papers read \(references\):](#)

[Sample Order \(for n = 40\)](#)

## Judging system

### Introduction

There are countless events organised across the world which have entries that need to be evaluated subjectively. This could range in scale and style from a course project to an online meme contest. It is inherently difficult to judge such contests, different people view the results differently. In the status quo, they rely on having certain official judges scoring each project on a scale (say 1 to 100), and then ranking the projects by average score.

There are 2 key issues in this approach:

1. It is difficult to extend it to scenarios with thousands of entries (common in international contests, or even national art contests etc.). This is because each judge cannot be expected to go through all entries, and here arises the fundamental problem of normalising across judges. This can be a difficult task, especially considering typical normalization mechanisms are not robust to a judge simply receiving a bad batch of projects, which will then get normalized up. So for statistical significance in normalization, the random batch of projects assigned needs to be large enough. Moreover, since judges have an idea of only a very small sample space of projects, it is difficult for them to assign scores with a global perspective. They also only see projects one by one, so their metrics and leniency might change as they gain more information. This approach is thus clearly imperfect from a practical implementation perspective.
2. However, even if you manage to eliminate such errors, there is a more fundamental problem at play. Humans are inherently bad at regression, that is assigning a rating over a range of values. Their estimates often contradict themselves and vary based on not only the information they currently have (which naturally changes over the process of judging), but also their mood. [Cite].

### Pairwise Comparisons

However, one thing humans are good at is comparisons. When shown 2 "objects" simultaneously, humans can be consistent with their preference. Keeping in mind the efficacy of pairwise judgements as a metric for evaluation, it is useful to create software that can automate the process of both providing pairs for comparisons and resolving these comparisons into a final ranking. Most of our project focuses on the latter, as it has an interesting reduction to one of Karp's 21 NP-Complete problems. The first part seems harder, for it requires analysis on what type of data a given algorithm performs well on. For now, we've mostly avoided this consideration and left it for future work.



## A note on applications

It is important to note that a pairwise-comparisons based approach works best when comparing pairs requires less time. This is not true when comparing large reports (say 50+ pages) or presentations as in such cases the number of comparisons doable by judges would be low. It is however useful in quicker to evaluate, and more subjective cases such as digital art contests, beauty pageants (as much as we condemn them) and meme contests where a judge can be shown 2 objects and can provide a comparison in seconds. As an aside, the same algorithm can be used as a more efficient way of ranking sports teams. Eg. the total points ranking (as used in IPL, EPL, and many other leagues) is actually a greedy heuristic for MFAS which we describe later. What if we could use the more general MFAS based ranking instead? This can be especially used for cricket / football national team rankings as they play each other in a non-league, sporadic format. In theory, our MFAS based approach can be a direct competitor for the more popular ELO rating.

## Inspiration

Note that we got the inspiration of the problem from [Gavel](#), an expo judging system designed for HackMIT and now used by many other hackathons. Gavel uses pairwise comparisons, but for the ranking algorithm it initially used a probabilistic approach based on Thurnstone's statistical model of judgements and Tikhonov regularization. It eventually shifted to the more sophisticated (and also probabilistic) [CrowdBT algorithm](#), which not only provides a ranking, but also provides a framework for collecting data. CrowdBT was originally meant for crowdsourcing data but is adapted for judging by Gavel. We have modelled the problem as a deterministic problem instead.

## Algorithmic Approach

So given pairwise comparisons (judging data) how can we generate a final ranking? We can model this problem as a graph where each object is a node and a directed edge from project  $u$  to  $v$  with weight  $w$  represents that  $v$  was preferred over  $u$  by  $w$  more judges than  $u$  over  $v$ . Now, we want to find an ordering (ranking) of all nodes such that the number of 'disagreements' in the ranking are minimized. A disagreement is if  $u$  gets ranked above  $v$  but a judge found  $v$  better in a comparison. Notice that if our graph is acyclic, finding a topological ordering suffices and gives 0 disagreements, and there are simple  $O(M)$  algorithms to do so using a *queue* or a *DFS*. Moreover, it is impossible to find an ordering with 0 disagreements if there is a directed cycle in the graph. This creates a bijection between ranking with minimum disagreements and the following problem:

Given a directed acyclic graph, remove the minimum number of edges such that the graph becomes acyclic.

This is the unweighted version of the minimum feedback arc set problem, which is known to be NP-complete. Our formulation also adds weights to each edge, which is also the classic Weighted Minimum Feedback Arc Set problem. This too is NP-complete, naturally. An important thing to note though is that as not many judges will be provide judgement on the same pair of objects, the edge-weights will be small.

## MFAS is NP-Hard proof

We can prove MFAS is NP-Hard by reducing Vertex Cover to Feedback Arc Set. As Vertex Cover is NP-Complete, FAS will thus be NP-Hard by definition as we can reduce any problem in NP to Vertex Cover, which can in-turn be reduced to FAS. FAS can be reduced to MFAS thus making it NP-Hard.

The feedback arc set, given a digraph  $G$ , asks whether the removal of  $\leq k$  edges can make  $G$  acyclic. Notice that FAS is in NP as given a subset of edges  $E'$  for removal, we can check if  $G(V, E/E')$  is acyclic using topological sort, which takes  $O(|V| + |E|)$  (linear) time.

### Reduction 1: Vertex Cover to Feedback Arc Set

We have to construct a graph  $G(V', E')$  such that  $\exists$  a feedback arc set of size  $\leq k$  iff  $\exists$  a vertex cover of size  $\leq k$  in  $G(V, E)$ .

1.  $\forall v \in G$  add  $v_{in}, v_{out}$  to  $G'$  with an edge from  $v_{in}$  to  $v_{out}$ .
2.  $\forall \{u, v\} \in G$  add  $\{u_{out}, v_{in}\}, \{v_{out}, u_{in}\}$  to  $G'$

This completes the construction. Proof:

In the construction, there is a cycle corresponding to each edge in the graph. If we have a vertex cover  $K \in G$  we can remove the corresponding  $v_{in}, v_{out}$  edges and as it the vertex covers all edges, all cycles corresponding to these edges will

be broken. This is because the cycle requires the existence of both  $\{v_{in}, v_{out}\}$  and  $\{u_{in}, u_{out}\}$ , one of which will get removed in any vertex cover. Conversely, given a feedback arc set  $K' \in G'$ , replace any edge of the form  $\{u_{out}, v_{in}\}$  in it with  $\{u_{in}, u_{out}\}$  and this would maintain the invariant of no cycles on removing. Then, we can simply add the corresponding  $u$  to construct a vertex cover of  $G$  of the same size.

## Reduction 2: Feedback Arc Set to Minimum Feedback Arc Set

1. Compute the minimum feedback arc set
2. Check if it has  $\leq k$  edges.

Thus, Minimum Feedback Arc Set is NP-Hard. This means it's a fertile problem for applying the discrete optimization techniques we've learn, but before that we explore some approximation / greedy algorithms which can also be used as initialisations for local search.

## Approximating MFAS - Key Results

Weighted MFAS is fixed parameter tractable. There exists a  $O(4^k * k! * n^{O(1)})$  algorithm for Minimum feedback vertex set (which in-turn can directly be used to find a Minimum Feedback Edge set). Here  $k$  is the size of the solution, i.e. the edge-set to be removed.

MFAS is also APX-Hard. This means that there is a proven lower-bound to which MFAS can be approximated using a polynomial-time algorithm. Note that since the approximation lower-bound on Vertex cover is 1.3606, the same can be said for MFAS. Further, if the Unique Games Conjecture is true, this lower-bound can be taken to arbitrarily close to 2.

A 2-approximation to MFAS is quite simple to state, so the above would be quite a significant result. The 2-approximation is as follows:

1. Fix an arbitrary permutation labelling of the vertices from 1 through  $n$
2. Construct two subgraphs  $G_L$  containing the edges  $\{u, v\}$  where  $u < v$ , and  $G_R$  containing those where  $u > v$ .
3. Now both  $G_L$  and  $G_R$  are acyclic subgraphs of  $G$ , and at least one of them is at least half the size of the maximum acyclic subgraph.

Note that in practice, a non-constant approximation may work better, as the best known ratio is  $O(\log(n)\log(\log(n)))$ .

A popular heuristic is the sorting in decreasing order of *indegree* — *outdegree*. This is similar to what is used in most sports tournaments today, when they rank teams in decreasing order of points won (*indegree*), without penalty based on losses (*outdegree*). It's amusing that sports with massive fan followings and money invested use such a naive greedy algorithm to achieve ranking, but it makes sense to make it simpler for fans to view and coaches/players to understand.

## Creating Synthetic Data

While we did find a popular benchmark dataset for MFAS called LOLIB, and compared our approaches to existing research papers, it is important to note that the data we're solving for has small edge-weights as described before. It thus felt natural to create synthetic data emulating the judging scenario as well. Note that having real-world judging data is less helpful than synthetic data, because there is no 'correct ranking' even for real-world data, because the existing judging systems are flawed in the first place. Synthetic data allows us to *plant* an ordering and hope the data generated using it allows it to be achieved, which it might not as we randomly add noise (disagreements). This we describe in further detail in the following generation algorithm.

### Initializations

```
const int N = 2005, T = 100, V = 15, C = 5, WL = 3, WU = 10;
//N is an upper bound on number of judges (taken in input).
//T is an upper bound on score (different for different files, always > input N)
//V is the category-score variance from the base score of a project
//C is the number of judging criteria / categories.
//WL and WU are bounds on how much a judge can *weigh* a particular category.
struct project{
    int idx;
    int score[C];
    int tot = 0;
} prj[N];

struct judge{
    int weight[C];
} juj[N];

int n, numj, jpp, mat[N][N], inp[N][N];

//project score assignment
for(int i = 1; i <= n; i++){
    int base = rng(V, T); //assign a base score to project b/w V and T
    prj[i].idx = i;
    for(int j = 0; j < C; j++){
        prj[i].score[j] = base + rng(-V, V); //In each category, project's score is base+R, R \in [-V, V] randomly
        prj[i].tot += prj[i].score[j]; //total score = sum across categories
    }
}
```

```

//Judge preference assignment
for(int i = 1; i <= numj; i++){
    for(int j = 0; j < C; j++){
        juj[i].weight[j] = rng(WL, WU); //judge i weighs category j as R, R \in [WL, WU] randomly
    }
}

//Perform comparisons for each judge
for(int i = 1; i <= n; i++){
    for(int j = i+1; j <= n; j++){
        for(int k = 1; k <= jpp; k++){ //do #judges-per-pair times
            int nj = rng(1, numj); //pick a random judge

            int si = 0; //score of ith project
            for(int it = 0; it < C; it++){
                si += prj[i].score[it] * juj[nj].weight[it];
            }
            int sj = 0; //score of jth project
            for(int it = 0; it < C; it++){
                sj += prj[j].score[it] * juj[nj].weight[it];
            }

            //increment and decrement matrix edge-weight for better and worse project
            if(si>sj){
                mat[i][j] += 1; mat[j][i] -= 1;
            }
            else if(sj>=si){
                mat[i][j] -= 1; mat[j][i] += 1;
            }
        }
    }
}

```

Essentially there can be multiple (set 5) 'criteria' or 'categories' a 'project' can be judged on. In the real-world these would be metrics like 'creativity', 'implementation', 'presentation' etc. Initially each project is assigned a base score. Then scores for each category are assigned uniformly between  $base - V$  and  $base + V$ , where  $V$  is a variation constant. The total score is computed as the sum of scores across categories and the projects are sorted in descending order of total score (which is printed as planted solution).

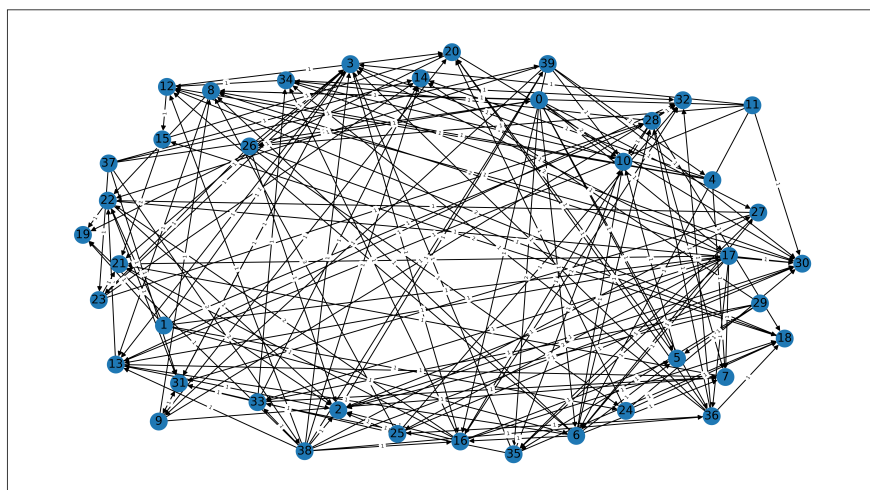
Then, each judge is given a characterisation based on weightage assigned to each category. This reflects the real world as judges have biases towards specific kinds of work. So their weighting constants for each category are randomized.

Finally, all pairs are compared by  $jpp$  random judges. They do this by  $\sum_{i \in categories} Weight_i * Score_i$  for both items. The item with more weighted score is adjudged the winner by the judge.

Clearly, the matrix so produced a) has  $jpp$  comparisons between each pair and b) might deviate from the planted solution due to the varying weightages of judges. b) is accepted as a consequence of having to add disagreements (and hence cycles). a) leads to an unrealistic number of judgements expected, so for our results we just take data from 5% of the judge comparisons and show that we get a similar ranking using just these as in the 100% data matrix closer-to-ideal matrix. Note that such variations obviously also happen in the real-world, and we don't claim our model to be perfect, just better than a regression system.

We create 3 data instances representing different scenarios:

1. AAD Project Judgements - 40 projects, 6 judges, 5 comparisons per pair with 5% random comparisons chosen finally.  $T = 100$ ,  $V = 10$
2. Hackathon - 250 projects, 10 judges, 5 comparisons per pair with 5% random comparisons chosen finally.  $T = 1,000$   $V = 500$  (high)
3. Digital Art Contest - 1000 projects, 30 judges, 5 comparisons per pair with 5% random comparisons chosen finally.  $T = 10,000$   $V = 2,000$



## Explaining the lolib dataset and its origin



The official non-judging specific datasets we used for testing our algorithms in the initial stage was the LOLIB dataset : <http://comopt.ifl.uni-heidelberg.de/software/LOLIB/>

LOLIB is a library of sample instances for the linear ordering problem. LOLIB includes data as well as optimum solution values.

### Bifurcation of the dataset

It contains a set of benchmark problems including all problem instances which have so far been used for conducting computational experiments by majority of research papers available online.

### Matrix Notation of the input dataset

A set of  $n$  objects is given which have to be ordered in a linear sequence. For every pair  $i$  and  $j$  of objects there are coefficients  $c_{ij}$  ( $c_{ji}$ ) expressing the preference for having  $i$  before  $j$  ( $j$  before  $i$ ) in this sequence. The task is to find a linear sequence such that the sum of the coefficients that are compatible with this ordering is maximized.

- **Input/Output matrices:**

This is a well-known set of instances that contains 50 real-world linear ordering problems generated from input-output tables from various sources

- **SGB instances:**

These instances are taken from the *Stanford GraphBase* and consist of input-output tables from sectors of the economy of the United States (Knuth 1993). The set has a total of 25 instances with 75 sectors.

- **Random instances of type A:**

This is a set with 175 random problems that has been widely used for experiments. Problems of type I (called *RandomAI*), are generated from a [0,100] uniform distribution. Sizes are 100, 150, 200 and 500 and there are 25 instances in each set for a total of 100.

## Experience with the dataset

The most puzzling thing we faced while analyzing the dataset was the **presence of non-zero values in the diagonals of the matrices**. After all, in linear ordering, the presence of a vertex  $i$  coming in the sequence before itself seems to make no sense and the having a non-zero penalty cost associated with this seems to be absurd. After spending much time on figuring out this anomaly, we realized the reason for this: a part of the dataset was originally collected for analysis by economists.

In this application, the economy (regional or national) is first subdivided into sectors. Then, an input/output matrix is created, in which the entry  $(i, j)$  represents the flow of money(or raw materials) from sector  $i$  to sector  $j$ . Economists are often interested in ordering the sectors so that suppliers tend to come first followed by consumers. It is not uncommon for industries to use their own end-product as raw material for some other application within the same industry (and hence the presence of diagonal entries). More info about this can be found here :

#### Input-output model

In economics, an input-output model is a quantitative economic model that represents the interdependencies between different sectors of a national economy or different regional economies. Wassily Leontief (1906-1999) is credited with developing this type of analysis and earned the Nobel Prize in Economics for his development of this model.

W [https://en.wikipedia.org/wiki/Input%E2%80%93output\\_model#Basic\\_derivation](https://en.wikipedia.org/wiki/Input%E2%80%93output_model#Basic_derivation)

### The need to change normalized matrices and the dataset originally used

In their original definition, some problem instances are not in normal form. For the computations documented here, all instances have been transformed to normal form.

Note, that if a constant is added to both entries  $c_{ij}$  and  $c_{ji}$  or if we take diagonal entries into account, the optimality of an ordering is not affected by this transformation. However, the quality of bounds does change. If we add large constants, then every feasible solution is close to optimal and no real comparison of qualities is possible (which is very relevant when we are comparing heuristics). Therefore it makes sense to transform every problem matrix  $C$  to its normal form satisfying the following conditions:

1. All entries of  $C$  are integral and nonnegative,
2.  $c_{ii} = 0$ , for all  $i = 1, \dots, n$
3.  $\min(c_{ij}, c_{ji}) = 0$ , for all  $1 \leq i < j \leq n$ .

**Note that this normal form for a specific input matrix is unique.**

**The normalized version of LOLib dataset can be found here :**

 [http://grafo.etsii.urjc.es/opticom/lolib/lop/lolib\\_bestvalues.zip](http://grafo.etsii.urjc.es/opticom/lolib/lop/lolib_bestvalues.zip)

## Exact Methods

## Brute Force

The most naive method to solve this problem is by forming all the permutations of  $\{1, \dots, n\}$  and iterating over them to calculate the sum of backward arcs. This runs in  $O(N! \cdot N^2)$  time which is feasible for  $N \leq 9$ .

## A Dynamic Programming Approach

While generating permutations for the brute force model, notice the fact that we create the *precise order* of already placed elements while only the **set** of elements matters. This allows for a natural dynamic programming model which can be viewed as building a permutation — placing nodes one by one and minimizing the sum of backward arcs to already placed elements. It has a similar flavor to the dynamic programming method for TSP where we keep track of already placed elements (or elements left to place) and the last placed vertex so we can obtain the cost of the whole tour by summing the cost of consecutive pairs.

We build the permutation progressively. Let *mask* denote a 0/1 bit-vector of size  $N$  where a 1 at the  $i^{th}$  position means that the  $i^{th}$  node has been placed while 0 means it is yet to be placed. Let  $dp[mask]$  denote the cost of backward arcs in the permutation already built using the nodes placed in *mask*.

Let's assume we plan to place  $j$  next. The the new mask would be `mask | (1 << j)` i.e. setting the  $j^{th}$  bit to 1. To calculate the contribution from  $j$ , we just loop from  $i = 1 \dots N$  and add the sum of backward arcs. ie.  $cost[j][i]$  if the  $i^{th}$  bit of *mask* is on. We do this for all valid  $j$  and take the minimum. The following presents a recursive dynamic programming code for the ease of understanding and implementation:

### Code:

```
int solveDP(int mask = 0)
{
    if(__builtin_popcount(mask) == k) return dp[mask] = 0; // base case (all bits are already placed)
    if(dp[mask] != OFFSET) return dp[mask] ;                // value pre-computed so use that
    int ans = INT_MAX ;                                     // infinity

    for(int i = 0; i < k; i++)
        if(!(mask & (1 << i))) { // checking if the i^th bit is empty. If so, try placing it next
            int backward_cost = 0;
            for(int j = 0; j < k; j++)
                if(mask & (1 << j)) // j^th bit already set (ie. place in permutation)
                    backward_cost += cost[i][j] ;

            ans = min(ans, backward_cost + solveDP(mask | (1 << i)));
        }
    return dp[mask] = ans ;
}
```

### Improvement:

The current complexity is  $O(2^N \cdot N^2)$ . This can be improved to  $O(2^N \cdot N)$  by not calculating for each  $(i, mask)$  the sum of backward edges from  $i$  to already placed nodes in *mask* rather saving time by pre-calculating them. This sub-problem can itself be formulated as another dynamic programming task, which can be solved in  $O(2^N \cdot N)$ . For each  $i$ , traversing the bitmasks in way that ensures that only 1 bit (ie. a node, say  $j$ ) changes in current bitmask to previous bitmask, we can just add/subtract the cost of that edge  $i \rightarrow j$  (which got added/removed).

Fortunately, there is a way for traversing bits in precisely this fashion: **Graycode**! Once we have done all the required pre-calculations, the final dynamic programming code turns out to be extremely simple! And, faster too.

```
void setSubmaskCosts() {
    for(int j = 0; j < k; j++) {
        int prev = 0 ; maskCost[j][prev] = 0 ;
        for(int i = 1; i < (1 << k); i++) {
            int mask = i ^ (i >> 1); // graycode
            // find the bit which changed (note: only 1 bit changes in graycode)
            int dif = mask ^ prev ;
            int pos = __builtin_ctz(dif); // finds the position of the first on bit

            int on = mask & (1 << pos); // to add or subtract
            if(on) maskCost[j][mask] = maskCost[j][prev] + cost[j][pos] ;
            else maskCost[j][mask] = maskCost[j][prev] - cost[j][pos] ;
            prev = mask ;
        }
    }
}

int solveDP(int mask = 0)
{
    if(__builtin_popcount(mask) == k) return dp[mask] = 0;
    if(dp[mask] != OFFSET) return dp[mask] ;
    int ans = INT_MAX ;
```

```

for(int i = 0; i < k; i++)
    if(!(mask & (1 << i)))
        ans = min(ans, maskCost[i][mask] + solvedP(mask | (1 << i)));

return dp[mask] = ans ;
}

```

## Mixed Integer Programming (MIP) Formulation

While we can solve small instances ( $N \leq 23$ ) using dynamic programming, we present another method to solve the original problem *exactly*. Just like with TSP, we seek a minimum cost ordering  $\pi$  of the nodes of the given graph  $G = (V, E)$  using mixed-integer programming.

Let  $c_{i,j}$  denote the cost associated with the directed edges  $(i, j) \in E$ , and let  $c_{i,j} = 0$  if  $(i, j) \notin E$ . If the cardinality of the (*unweighted*) feedback edge set is to be minimized, then for each  $(i, j) \in E$  we have  $c_{i,j} = 1$ . If the weighted minimum feedback edge set problem is to be solved, then all  $c_{i,j}$  associated with a directed edge equals the weight of the corresponding edge  $(i, j)$ . Furthermore, let the *binary decision variables*  $y_{i,j}$  associated with a given ordering  $\pi$  encode the following:  $y_{i,j} = 0$  if node  $i$  precedes  $j$  in  $\pi$ , and let  $y_{i,j} = 1$  otherwise. Any ordering  $\pi$  uniquely determines a corresponding  $y$ . This results in the following integer programming formulation:

$$\begin{aligned}
 \min_y \quad & \sum_{j=1}^n \left( \sum_{k=1}^{j-1} c_{k,j} \cdot y_{k,j} + \sum_{l=j+1}^n c_{l,j} \cdot (1 - y_{j,l}) \right) \\
 \text{such that} \quad & y_{i,j} + y_{j,k} - y_{i,k} \leq 1, & 1 \leq i < j < k \leq n \\
 & y_{i,k} \leq y_{i,j} + y_{j,k}, & 1 \leq i < j < k \leq n \\
 & y_{i,j} \in \{0, 1\}, & 1 \leq i < j \leq n
 \end{aligned}$$

Any  $y$  that satisfies the **triangle inequalities** above must correspond to a unique ordering. Note that, there are  $O(n^2)$  binary variables ( $y$ ) and  $O(n^3)$  constraints in the formulation above.

While custom-tailored cutting plane algorithms have been developed to solve this integer program (and the linear ordering problem in general), a *naïve* implementation of this using an industrial solver (Google's OR-Tools specifically) suffers even for small values. Testing for randomly generated cost matrices for  $n \approx 20$ , Google's MIP solver took few seconds while for  $n = 30$  it took *several hours*.

As we don't know the advanced theories for cutting plane techniques to optimize our MIP formulation, we present below local search methods instead to approach larger instances of the problem.

## Heuristics

### 2-OPT

A simple method to improve a bad solution is to do 2-OPT similar to the way used canonically, say in TSP. We find the best pair of nodes  $(i, j)$  swapping which improves the score largest. If the neighborhood  $\sim O(N^2)$  is too big, then we use the *break first* rule of finding the first swap which improves the score and thereafter finding the next local swap to perform.

This does improve the solution but faces the common problem with all local search algorithms — getting stuck in local minima! The fact that it reaches local minima very fast since generally  $N$  is not very large enables us to perform multiple restarts with random initialization leading to pretty good score in small instances like  $N \leq 50$ .

### Naive with insertion as a local move

This heuristic checks whether the objective function can be improved if the position of an object in the current ordering is changed. All possibilities for altering the position of an object are checked and the method stops when no further improvement is possible this way.

**Local move:** **INSERT\_MOVE**( $O_j, i$ ) includes deletion of  $O_j$  from its current position  $j$  in permutation  $O$  and its insertion at position  $i$  (i.e., between the objects currently in positions  $i-1$  and  $i$ ). Now, the insertion heuristic tries to find improving moves examining eventually all possible new positions for all objects  $O_j$  in the current permutation  $O$ .

The evaluation of increase/decrease in optimisation function all  $N$  possible insertion local moves for a vertex  $O_j$  is  $O(N)$ . So, overall decision of which local move to make is  $O(N^2)$ .

**INSERT\_MOVE**( $p_j, i$ )

$$\begin{aligned}
 p'_j &= (p_1, \dots, p_{i-1}, p_j, p_i, \dots, p_{j-1}, p_{j+1}, \dots, p_m) & \text{if } i < j \\
 p &= (p_1, \dots, p_{j-1}, p_{j+1}, \dots, p_{i-1}, p_j, p_i, \dots, p_m) & \text{if } i > j
 \end{aligned}$$

Even on switching to the total points based greedy initialisation as described earlier, the improvement in score was not significant.

```
void perform_change(int idx, int seeked_idx)
{
    //shifting the elements of the ordering based on whether the insertion is in the beginning or in the end
    int node = v[idx];
    if (seeked_idx < idx)
    {
        for (int i = idx; i > seeked_idx; i--)
        {
            v[i] = v[i - 1];
        }
        v[seeked_idx] = node;
    }
    else
    {
        for (int i = idx; i < seeked_idx; i++)
        {
            v[i] = v[i + 1];
        }
        v[seeked_idx] = node;
    }
}

bool find_best_local_move()
{
    LL i, j, k, t, temp;
    LL idx_of_best_node = -1;
    LL seeked_pos_best = -1;
    LL best_improvement = -1;
    for (i = 0; i < n; i++)
    {
        LL node_now = v[i];
        LL personal_best_pos = -1;
        LL personal_best_improvement = -1;
        LL curr_change = 0;
        for (j = i - 1; j >= 0; j--)
        {
            LL ver = v[j];
            //maintaining prefix sum array
            curr_change += cost[ver][node_now] - cost[node_now][ver];
            if (curr_change < 0)
            {
                if (fabs(curr_change) > personal_best_improvement)
                {
                    personal_best_improvement = fabs(curr_change);
                    personal_best_pos = j;
                }
            }
        }
        curr_change = 0;
        for (j = i + 1; j < n; j++)
        {
            LL ver = v[j];
            //maintaining prefix sum array
            curr_change += cost[node_now][ver] - cost[ver][node_now];
            if (curr_change < 0)
            {
                if (fabs(curr_change) > personal_best_improvement)
                {
                    personal_best_improvement = fabs(curr_change);
                    personal_best_pos = j;
                }
            }
        }

        if (personal_best_improvement > best_improvement)
        {
            best_improvement = personal_best_improvement;
            idx_of_best_node = i;
            seeked_pos_best = personal_best_pos;
        }
    }
    //returning false if no possible improvement found
    if (idx_of_best_node == -1)
    {
        return false;
    }
    else
    {
        //improvement found by changing vertex at position 'idx_of_best_node' to position 'seeked_pos_best'
        //performing the change to reflect the insertion
        perform_change(idx_of_best_node, seeked_pos_best);
        violations_now -= best_improvement;
        return true;
    }
}
```

## Kernighan-Lin 2

The main problem with local improvement heuristics is that they very quickly get trapped in a local optimum.



In contrast to pure improvement heuristics, it allows that some of the simple moves are not improving. In this way the objective can decrease locally, but new possibilities arise for escaping from the local optimum.

- (1) Compute some linear ordering  $O$ .
- (2) Let  $m = 1$ ,  $S_m = \{1, 2, \dots, n\}$ .
- (3) Among all possibilities for inserting an object of  $S_m$  at a new position determine the one leading to the largest increase  $g_p$  of the objective function (increase may be negative). Let  $s$  be this object and  $p$  the new position.
- (4) Move  $s$  to position  $p$  in the current ordering. Set  $s_m = s$  and  $p_m = p$ .
- (5) If  $m < n$ , set  $S_{m+1} = S_m \setminus \{s\}$  and  $m = m + 1$ . Goto (3).
- (6) Determine  $1 \leq k \leq m$ , such  $G = \sum_{i=1}^k g_i$  is maximum.
- (7) If  $G \leq 0$  then **Stop**, otherwise, starting from the original ordering  $O$ , successively move  $s_i$  to position  $p_i$ , for  $i = 1, 2, \dots, k$ . Let  $O$  denote the new ordering and goto (2).

## The Heuristic of Chanas & Kobylanski

The algorithms make use of two main modifications to the current linear ordering : **SORT AND REVERSE**.

### SORT

So, essentially we traverse through each element of the ordering and see if the ordering can be improved by a possible insertion of that element in a position to the left of its current position.

Formally, we make a single pass through the **nodes from the left to the right**. As each node is considered, it is moved to the position to the left of its current position that minimises the number of backward edges(if that number of back arcs is fewer than its current position).

$$SORT(t_1, \dots, t_k) = \begin{cases} t_1 & k = 1 \\ INSERT\_MOVE(t_k, (SORT(t_1, \dots, t_{k-1}))) & k > 1 \end{cases}$$

### Reverse

Reverse is defined as just reversing the current ordering.

$$REVERSE(t_1, t_2, \dots, t_k) = (t_k, t_{k-1}, t_{k-2}, \dots, t_1)$$

$SORT^*$  = repeatedly apply SORT until there is no improvement in the number of back-arcs.

The Chanas algorithm is defined as:

$$(SORT^* \circ REVERSE)^*$$

Here, the symbol  $\circ$  stands for the operation of super-position of two functions  $f \circ g(x) = f(g(x))$ .


If the permutation  $O = \langle O_1, O_2, \dots, O_n \rangle$ , is an optimum solution to the maximization problem, then an optimum solution to the minimization problem is  $O^* = \langle O_n; O_{n-1}, \dots, O_1 \rangle$ . The method is used to escape local optimality: once a local optimum solution  $O$  is found, the process is re-started from the permutation  $O^*$  (REVERSE operation).

In a global iteration, the algorithm tries to find a better position for each element to its left. When no further improvement is possible, it generates a new solution by applying the REVERSE operation from the last solution obtained, and performs a new global iteration. The method halts when the best solution found cannot be improved further in the current global iteration.

```
pair<LL, LL> sort_star()
{
    LL violations_at_beginning = violations_now;
    for (int i = 0; i < n; i++)
    {
        int idx_to_be_moved = i;
        LL lb = 0, ub = i - 1;
        LL best_diff = 0, curr_diff = 0;
        int best_pos = i;
        int n1 = v[i];
        for (int j = ub; j >= 0; j--)
        {
            int n2 = v[j];
            curr_diff += cost[n2][n1] - cost[n1][n2];
            if (curr_diff < best_diff)
            {
                best_diff = curr_diff;
                best_pos = j;
            }
        }
        if (best_pos != i)
        {
            //shift elements and update violations now
```

```
        perform_change(i, best_pos);
        violations_now += best_diff;
    }
}
// debug(violations_at_beginning);
// debug(violations_now);
return {violations_at_beginning, violations_now};
}
```

These are values produced by naive insertion heuristic, KL2, Chanas on Lolib initial dataset







 finalized\_MFAS\_LOLIB\_performance\_anm

[https://docs.google.com/spreadsheets/d/1rQZ5g2iYe5gPERX0\\_V4\\_MxW82a94UxRoOzA9UTxx0o/edit?usp=drive](https://docs.google.com/spreadsheets/d/1rQZ5g2iYe5gPERX0_V4_MxW82a94UxRoOzA9UTxx0o/edit?usp=drive)

ol

esdk

Benchmark

 Synthetic Data Files	 Total edge costs sum	 Anmol's Naive insertion Running time (less than a sec)	 Anmol's Lin Kernighan 2 (Running time: less than a sec)	 Chanas (Running time: less than a sec)	 Nikhil's (5mins to run each)
100.in	177	<u>Forward: 176 Backward: 1</u>	Forward: 176 Backward: 1	<b>Forward: 176 Backward: 1</b>	77
250.in	7641	<u>Forward: 7565 Backward: 76</u>	Forward: 7565 Backward: 76	<b>Forward: 7602 Backward: 39</b>	7349
1000.in	74757	<u>Forward:73891 Backward:866</u>	Forward:73891 Backward:866	<b>74070</b>	46229
100big	3828	=	-	<b>3826</b>	<b>3826</b>
250big	150063	=	-	<b>149691</b>	149670
1000big	1455192	=	-	<b>1447610</b>	982012

Papers read (references):

1. [https://www.mat.univie.ac.at/~neum/ms/minimum\\_feedback\\_arc\\_set.pdf](https://www.mat.univie.ac.at/~neum/ms/minimum_feedback_arc_set.pdf) - A survey of weighted minimum feedback arc set approaches along with their own approach [MIP]
2. <https://www.fim.uni-passau.de/fileadmin/dokumente/fakultaeten/fim/forschung/mip-berichte/mip1104.pdf> - Applying conventional (quick-sort, insertion-sort and modifications) sorting algorithms to the minimum feedback arc set problem. Also discusses an extension of 1-opt/2-opt to this. Gives some nice pointers to keep in mind for the problem.
3. **An efficient method for finding a minimal feedback arc set in directed graphs, Sungju Park and Sheldon B. Akers** - On calculating minimum feedback arc sets on unweighted graphs. Crucial insight: Finding SCCs, and the biconnected components within each SCC, and then doing DFS using the heuristic of picking the next node with the most outgoing edges [minimize depth → minimize back-edges → minimize cycles]. There's also a final part called 'cut checking' which I didn't really get.
4. **A Fast and Effective Algorithm for the Feedback Arc Set Problem, Youssef Saab** - Primarily, mentions method(s) on how to generate testcases for MFAS. Presents datasets and their results in terms of (number of edges in MFAS /total number of edges).
5. <https://pdf-drive.com/pdf/28Applied20Mathematical20Sciences201752920Rafael20MartC3AD2C20Gerhard20Reinelt2028auth.2920-20The20Linear20Ordering20Problem20Exact20and20Heuristic20Methods20in20Combinatorial20Optimization-Springer-Verlag20Berlin20Heidelberg202820.pdf> - Mentions heuristics for linear ordering problem, mentions some possible local search approaches + results obtained.
6. <http://comopt.ifi.uni-heidelberg.de/software/LOLIB/> - Contains dataset for Linear ordering problem with values of optimal solns or upper bounds (useful for testing)
7. <https://www.uv.es/~rmarti/paper/docs/lop5.pdf> - contains data related to performance of several LS techniques on several benchmark datasets
8. CrowdBT algorithm - Crucial insight taken: Produce entire matrix of comparisons and then take x% of it to show similar results with less comparisons are possible.

Sample Order (for n = 40)

Order (from 100% judgement results):

38 12 30 2 1 18 39 7 28 27 40 36 5 24 37 29 34 26 4 10 17 35 25 19 13 9 16 14 8 23 22 32 6 15 33 21 11 3 31 20

Order (from using just 5% of the judgement results):

12 38 1 2 18 7 30 27 24 25 39 40 5 28 37 29 26 36 4 17 19 6 11 34 35 13 16 15 10 21 9 14 23 8 22 20 3 31 32 33