A survey on the Minimum Linear Arrangement problem

George Kallitsis
Electrical and Computer Engineering department
National Technical University of Athens
Athens, Greece
el17051@mail.ntua.gr

Iliana Maria Xygkou
Electrical and Computer Engineering department
National Technical University of Athens
Athens, Greece
el17059@mail.ntua.gr

***Abstract*—The Minimum Linear Arrangement problem has been proven to be NP-complete for arbitrary graphs. However, it can be simplified by specifying the type of graph. There have been found efficient algorithms by applying restrictions to the input or the output, and there exist approximation techniques trying to achieve the optimal.**

***Index Terms*—MLA, optimal, trees, approximation**

I. Introduction

In general, the minimum linear arrangement problem is defined as follows. Given a graph $G(V,E)$ where E with $|E| = m$ and V with $|V| = n$ are the sets of edges and vertices respectively, a permutation π (or a linear ordering of the vertices or an one-dimensional layout) is defined as $\pi : V \to \{1,2,...,n\}$. The Minimum Linear Arrangement problem requires to find the permutation π s.t. the below objective function is minimized:

$$\sum_{(i,j)\in E} |\pi(i) - \pi(j)|$$

It's been proven that this problem is NP-complete for the general case. The solution is trivial when G is a complete graph, since every arrangement is optimal. However, when G is a rooted or undirected tree, complete bipartite, hypercube, rectangular or square mesh, etc, there exist algorithms of polynomial time.
Solving this problem can assist in designing of error-correcting codes, minimization of wire's length at the placement phase in VLSI design, biology, graph drawing and reordering of large sparse matrices.
In this paper, we are going to present some existing algorithms which solve the MLA problem, some approximation techniques which attain a near-optimal arrangement and finally we are going to measure the quality of some heuristics by executing experiments and simulations on random and real graphs.

II. Existing algorithms

We are going to order some existing algorithms with regards to their input.

A. Small number of vertices

The first algorithm [1] defines a formulation of the problem as a mathematical program with a linear objective and nonlinear equality constraints. The Linear Programming model and in particular a mixed-integer linear programming formulation is described below:

$$Minimize\{\sum_{p=1}^{n} \sum_{q=1,q>p}^{n} (q-p) \sum_{(i,j)\in E} z_{ipjq}\} \tag{1}$$

$$\sum_{i=1}^{n} \gamma_{ij} = 1, \quad (1 \le j \le n) \tag{2}$$

$$\sum_{j=1}^{n} \gamma_{ij} = 1, \quad (1 \le i \le n) \tag{3}$$

$$\gamma_{ij} \in \{0,1\} \tag{4}$$

$$z_{ipjq} \ge 0, \quad (1 \le i,p,j,q \le n; q > p; j \ne i) \tag{5}$$

$$\sum_{j=1,j\ne i}^{n} z_{ipjq} = \gamma_{ip}, \quad (1 \le i,p,q \le n; q > p) \tag{6}$$

$$\sum_{j=1,j\ne i}^{n} z_{jqip} = \gamma_{ip}, \quad (1 \le i,p,q \le n; q < p) \tag{7}$$

$$\sum_{q>p} z_{ipjq} + \sum_{q<p} z_{jqip} = \gamma_{ip}, \quad (1 \le i,p,j \le n; j \ne i) \tag{8}$$

where:

$$z_{ipjq} = \gamma_{ip}\gamma_{jq}$$

$$\gamma_{ij} = \begin{cases} 1, & \text{if label } j \text{ is assigned to vertex } i \text{, i.e. } \pi(i) = j \\ 0, & \text{otherwise} \end{cases}$$

This algorithm was applied to graphs derived from Nugent distance matrices where $|V| = n \in \{12,15,16,17,23\}$ and found to be time-efficient. However, as it is the case for almost every LP-model, the time complexity of this algorithm is exponential. As a result, it should be used when the number of vertices is relatively small.

B. Undirected trees (or rooted trees)

For this type of input we are going to order some existing algorithms with regards to their output.

1) No constraints in output: There are two widely-known algorithms which were devised during 1980-1990s.
The first one is by *Shiloach* [2] and exploits the ability of trees to recursively break down to smaller subtrees, with regards to the number of vertices, to which the same algorithm can be applied, and their optimal linear arrangements can be combined to compute the minimum linear arrangement of the original tree. The algorithm is structured as follows:

1) Find vertex v_* s.t. $n_i \leq \lfloor \frac{n}{2} \rfloor$ where T_i, $i \in \{0, ..., k\}$ is the subtree of $Tmodv_*$ i.e., the subtree that results from removing v_*. In an anchored tree, v_* is the vertex at which the anchor is connected to the tree.
2) Find minimum arrangements π_0 for $\overrightarrow{T}(v_0)$ and π_0' for $\overleftarrow{T - T_0}(v_*)$ (or for $T - T_0$ if T is anchored.).
3) Compute $C_\alpha(A)$:
$$C_\alpha(A) = C[\pi, T(\alpha)] =$$
$$\begin{cases} C[\pi, \overrightarrow{T}_0(v_0)] + C[\pi, \overleftarrow{T - T_0}(v_*)] + 1, & \text{if } \alpha = 0 \\ C[\pi, \overrightarrow{T}_0(v_0)] + C[\pi, T - T_0] + n - n_0, & \text{if } \alpha = 1 \end{cases}$$
4) Compute p_α: p_α is the greatest integer s.t.
$$n_i > \lfloor \frac{n_0 + 2}{2} \rfloor + \lfloor \frac{n_* + 2}{2} \rfloor \text{ for } i = 1, ..., 2p_\alpha - \alpha, \text{ where}$$
$$n_* = n - \sum_{i=0}^{2p_\alpha - \alpha} n_i$$
If $p_\alpha = 0$ go to step 9.
5) Find minimum arrangements $\pi_i, i = 1, .., 2p_\alpha - \alpha$, for $\overrightarrow{T}_i(v_i)$, $i = 1, 3, ..., 2p_\alpha - 1$, and for $\overleftarrow{T}_i(v_i)$, $i = 2, 4, .., 2p_\alpha - 2\alpha$, and a minimum arrangement π_* for $T - (T_1, ..., T_{2p_\alpha - \alpha})$.
6) Compute $C_\alpha(B)$:
$$C_\alpha(B) = C[\pi, T(\alpha)] =$$
$$\sum_{i=1, i \text{ is odd}}^{2p_\alpha - 1} C[\pi, \overrightarrow{T}_i(v_i)] + \sum_{i=1, i \text{ is even}}^{2p_\alpha - 2\alpha} C[\pi_i, \overleftarrow{T}_i(v_i)] +$$
$$C[\pi, T_*] + S_\alpha, \text{where}$$
$$S_0 = (n_3 + n_4) + 2(n_5 + n_6) + ... + (p_0 - 1)(n_{2p_0 - 1} + n_{2p_0}) + p_0(n_* + 1)$$
$$and$$
$$S_1 = (n_2 + n_3) + 2(n_4 + n_5) + ... + (p_1 - 1)(n_{2p_1 - 2} + n_{2p_1 - 1}) + p_1(n_* + 1) - 1$$
7) If $C_\alpha(A) \leq C_\alpha(B)$ go to step 9.
8) The arrangement π_m of type $(T_1, T_3, ..., T_{2p_\alpha - 1} | v_* | T_{2p_\alpha - 2\alpha}, ..., T_4, T_2)$ determined by π_i on T_i for $i = 1, ..., 2p_\alpha - \alpha$ and by π_* on $T - (T_1, ..., T_{2p_\alpha - \alpha})$ is a minimum arrangement of T, and $C[\pi, T(\alpha)] = C_\alpha(B)$. Stop.
9) The arrangement π_m of type $(T_0 | v_*)$ determined by π_0 on T_0 and by π_0' on $T - T_0$ is a minimum arrangement of T, and $C[\pi, T(\alpha)] = C_\alpha(A)$. Stop.

This algorithm has time complexity $O(n^{2.2})$.
Recently, an error was found in *Shiloach*'s algorithm by *J. L. Esteban* and *R. Ferrer-i-Cancho* [3]. During their experiments using the above algorithm, they observed that it was producing wrong results for complete binary trees with more than 5 levels. It's shown that there is a closed-form expression for the minimum value of the objective function for trees with $k \geq 1$ levels:
$$D_{min} = min \sum_{(i,j) \in E} |\pi(i) - \pi(j)| = 2^k (\frac{k}{3} + \frac{5}{18}) + (-1)^k \frac{2}{9} - 2 \quad (9)$$
According to equation 9, for $k = 5$ $D_{min} = 60$, but applying *Shiloach*'s algorithm indicates that $D_{min} = 46$. In order to address this issue and ensure that correct results would be produced in any case of undirected trees, they proposed two possible corrections:

1) By redefining S_0 and S_1:
$$S_0 = (n_3 + n_4) + 2(n_5 + n_6) + ... + (p_0 - 1)(n_{2p_0 - 1} + n_{2p_0}) + p_0(Z + 1)$$
$$and$$
$$S_1 = (n_2 + n_3) + 2(n_4 + n_5) + ... + (p_1 - 1)(n_{2p_1 - 2} + n_{2p_1 - 1}) + p_1(Z + 1) - 1$$
$$where$$
$$Z = n_* + n_0$$
2) By redefining n_* and subsequently p_α:
$$n_* = n - \sum_{i=1}^{2p_\alpha - \alpha} n_i$$
and p_α is the greatest integer s.t.
$$n_i > \lfloor \frac{n_0 + 2}{2} \rfloor + \lfloor \frac{n_* - n_0 + 2}{2} \rfloor \text{ for } i = 1, ..., 2p_\alpha - \alpha$$

Some years later, *Chung* [4] suggested two algorithms for computing the minimum linear arrangement of an undirected tree T or a rooted tree T^*, which were very similar to the previous one but managed to be more time-efficient.
The first one, which resembles *Shiloach*'s algorithm but improves it by conducting a more fastidious complexity analysis, is presented below:

1) If the tree has a root r, go to step 6.
2) Find a center u of T (same as vertex v_* in *Shiloach*'s algorithm).
3) Determine subtrees $T_0, T_1, ...,$ of $T - u$ (same as $Tmodv_*$ in *Shiloach*'s algorithm) where $|V(T_i)| = t_i$ and $t_0 \geq t_1 \geq ....$ Find the greatest positive integer q s.t.
$$t_{2q} \geq \lfloor \frac{t_0}{2} + 1 \rfloor + \lfloor \frac{z}{2} + 1 \rfloor$$

where

$$z = n - \sum_{i=0}^{2q} t_i.$$

If no q satisfying the above conditions is found, set $q = -1$.

4) If $q \neq -1$, go to step 5. Else, compute the minimum linear arrangements $\pi(T_0^*)$ and $\pi(T^* - T_0)$, and their costs $g(T_0^*)$ and $g(T^* - T_0)$ respectively. Then, set $g(T^*) = C(T_0 :) = g(T_0^*) + g(T^* - T_0) + 1$ and combine $\pi(T_0^*)$ and $\pi(T^* - T_0)$ to form $\pi(T) = \pi(T_0 :)$. Stop.
5) Compute the minimum linear arrangements and their costs of T_i^* and $T_i \cup Z$ where $Z = T - \bigcup_{i=0}^{2q} T_i$ for $i = 0, 1, ..., 2q$. Define $Q_i = \{0, 1, ..., 2q\} - \{i\}$ and define i_j as the jth smallest integer contained in Q_i. Determine:

$$f(T) = \min_{i=0,1...,2q} C(T_{i_2}, T_{i_4}, ..., T_{i_{2q}} : T_{i_{2q-1}}, ..., T_{i_1})$$

and corresponding linear arrangement $\pi(T)$. Stop.
6) Determine subtrees $T_0, T_1, ...,$ of $T^* - r$ where $|V(T_i)| = t_i$ and $t_0 \geq t_1 \geq ....$ Find the greatest integer p s.t.:

$$t_{2p+1} \geq \lfloor \frac{t_0}{2} + 1 \rfloor + \lfloor \frac{y}{2} + 1 \rfloor$$

where

$$y = n - \sum_{i=0}^{2p+1} t_i.$$

If no p satisfying the above conditions is found, set $p = -1$.
7) If $p \neq -1$, go to step 8. Else, compute the minimum linear arrangements $\pi(T_0^*)$ and $\pi(T - T_0)$, and their costs $g(T_0^*)$ and $f(T - T_0)$ respectively. Then, set $g(T^*) = C(: T_0)$ and $\pi(T^*) = \pi(: T_0)$. Stop.
8) Compute $g(T_i^*)$ and $f(T_i \cup Y)$, for $i = 0, ..., 2p + 1$, where $Y = T - \bigcup_{i=0}^{2p+1} T_i$. Define $P_i = \{0, 1, ..., 2p + 1\} - \{i\}$ and define i_j as the jth smallest integer contained in P_i. Determine:

$$g(T^*) = \min_{i=0,1,...,2p+1} C(T_{i_2}, T_{i_4}, ..., T_{i_{2p}} : T_{i_{2p+1}}, ..., T_{i_1})$$

and corresponding linear arrangement $\pi(T^*)$. Stop.

This algorithm has $O(n^2)$ time complexity.

Apart from the previous algorithm, *Chung* proposed a second one which improves the first one by changing the way the minimum linear arrangements of subtrees are used during recursion for the sake of efficiency, and is divided into three parts:

Part 1: Finding a minimum linear arrangement and its cost for an undirected tree T.

1) Find a center u of T.
2) Determine subtrees $T_0, T_1, ...,$ of $T - u$ where $|V(T_i)| = t_i$ and $t_0 \geq t_1 \geq ....$ Find the greatest positive integer $q = q(T)$ s.t.

$$t_{2q} \geq \lfloor \frac{t_0 + 2}{2} \rfloor + \lfloor \frac{z + 2}{2} \rfloor$$

where

$$z = n - \sum_{i=0}^{2q} t_i.$$

If no q satisfying the above conditions is found, set $q = -1$.
3) If $q \neq -1$, go to step 4. Else, compute the minimum linear arrangements $\pi(T_0^*)$ and $\pi(T^* - T_0)$, and their costs $g(T_0^*)$ and $g(T^* - T_0)$ respectively. Then, compute $C(T_0 :) = g(T_0^*) + g(T^* - T_0) + 1$, $f(T) = C(T_0 :)$ and $\pi(T) = \pi(T_0 :)$.Stop.
4) Find $h(T_i^*, Z^*)$ for $i = 0, 1, ..., 2q$, where $Z^* = T^* - \bigcup_{i=0}^{2q} T_i$. Define $Q_i = \{0, 1, ..., 2q\} - \{i\}$ and define i_j as the jth smallest integer contained in Q_i. Determine:

$$f(T) = \min_{i=0,1...,2q} C(T_{i_2}, T_{i_4}, ..., T_{i_{2q}} : T_{i_{2q-1}}, ..., T_{i_1})$$

and corresponding linear arrangement $\pi(T)$. Stop.

Part 2: Finding the minimum linear arrangement and its cost for a rooted tree T^* with root r.

1) Determine subtrees $T_0, T_1, ...,$ of $T^* - r$ where $|V(T_i)| = t_i$ and $t_0 \geq t_1 \geq ....$ Find the greatest integer $p = p(T^*)$ s.t.:

$$t_{2p+1} \geq \lfloor \frac{t_0 + 2}{2} \rfloor + \lfloor \frac{y + 2}{2} \rfloor$$

where

$$y = n - \sum_{i=0}^{2p+1} t_i.$$

If no p satisfying the above conditions is found, set $p = -1$.
2) If $p \neq -1$, go to step 3. Else, compute the minimum linear arrangements $\pi(T_0^*)$ and $\pi(T - T_0)$, and their costs $g(T_0^*)$ and $f(T - T_0)$ respectively. Then, set $g(T^*) = C(: T_0)$ and $\pi(T^*) = \pi(: T_0)$. Stop.
3) Find $h(T_i^*, Y^*)$ for $i = 0, 1, ..., 2p + 1$, where $Y^* = T^* - \bigcup_{i=0}^{2p+1} T_i$. Define $P_i = \{0, 1, ..., 2p + 1\} - \{i\}$ and define i_j as the jth smallest integer contained in P_i. Determine:

$$g(T^*) = \min_{i=0,1,...,2p+1} C(T_{i_2}, T_{i_4}, ..., T_{i_{2p}} : T_{i_{2p+1}}, ..., T_{i_1})$$

and corresponding linear arrangement $\pi(T^*)$. Stop.

Part 3: Finding the minimum linear arrangement and its cost for a rooted tree T^* and the tree $T \cup \overline{T}$ which results from joining the roots of T^* and $\overline{T}^*$ with an edge, where $|V(\overline{T})| \leq |V(T)|$.

1) Find a center u of the tree $T \cup \overline{T}$ in T.
2) Determine subtrees $X, T_1, T_2, ...,$ of $T \cup \overline{T} - u$ where $|V(T_i)| = t_i$ and $t_1 \geq t_2 \geq ....$ $|V(X)| = x, |V(T)| = n, |V(T')| = n' \leq n$ and X is the subtree in which $\overline{T}$ is contained.
3) Suppose $\overline{P}$ is the path between u and the root r of T^* and $\overline{P}$ contains $u = v_0, v_1, ..., v_s = r$. Suppose X_i is the subtree of $T \cup \overline{T} - v_i$ which contains $\overline{T}$. Set the

tree $R_i^* = (T \cup \overline{T} - X_i)^*$ with v_i as the root. Compute $q = q(T \cup \overline{T}), p_i = p(R_i^*)$ and $p' = p((T \cup \overline{T} - T_1)^*)$.

4) If $n' > \frac{n}{3}$, go to step 8. If $p_i = -1$ for all $0 \leq i \leq s$, go to step 5. If $q = 1$, go to step 8. If $x < t_1, q = -1, p' = 0$, go to step 8. Go to step 7.
5) If $x < t_2, q = -1, p' = -1$, go to step 6. Go to step 8.
6) If $p_0 \geq 0$, go to step 7. If $p_0 = -1$ and $t_2 \geq \sum_{i>2} t_i$, go to step 9. Go to step 10.
7) Compute $g(T^*), \pi(T^*)$ and remember the cost and minimum linear arrangements of the following trees, if any:
    a) R_0^*
    b) S_i^*, the second largest subtree of R_i^*
    c) T_i^*
    d) $T - T_1$

    Compute $f(T \cup \overline{T})$ based on the above data.
8) Compute $f(T \cup \overline{T}), \pi(T \cup \overline{T})$ and remember the cost and minimum linear arrangements of the following trees, if any:
    a) T_i^*
    b) $T_i \cup W$ where W is a subtree of $T \cup \overline{T}$ that doesn't contain vertices in any of the $2q+1$ largest subtrees of $T \cup \overline{T} - u$
    c) R_0^*
    d) $R_0 - T_1$
    e) $S_i, R_i - R_{i-1}$

    Compute $g(T^*)$ based on the above data.
9) Compute $g(T_1^*), h(T_2^*, (R_0 - T_1 - T_2)^*), f(T \cup \overline{T} - T_1 - T_2), f(X - R_0 - \overline{P})$. Set $g(T^*) = C(: T_1)$ and $f(T \cup \overline{T}) = C(T_1 : T_2)$.
10) If $q(R_0 - T_1) = -1$, compute $g(T_1^*), g(T_2^*), h(T_0^* - T_1 - T_2, X^*), f(X - R_0 - \overline{P})$. Else, compute $g(T_1^*), h(R_0^* - T_1 - T_2, X^*), h(T_2^*, (W'')^*), f(X - R_0 - \overline{P})$ where $W'' = R_0 - \bigcup_{i=1}^{2p''+2} T_i$ and $p'' = p(R_0^* - T_1 - T_2)$. Set $g(T^*) = C(: T_1)$ and $f(T \cup \overline{T}) = C(T_1 : T_2)$. Stop.

This algorithm has $O(n^\lambda)$ time complexity, where $\lambda > \frac{log3}{log2}$.

2) Planar graphs: The first approach for this constraint in the output was introduced by *Iordanskii* [5], who proposed the below algorithm:
Let tree $T = (V, E)$ with n vertices be represented by the list of its edges. The minimal planar numbering algorithm runs as follows:

1) Turn tree T into a rooted directed tree with its root in the centroid of T (centroid of a tree is a node which if removed from the tree would split it into a "forest", such that any tree in the forest would have at most half the number of vertices in the original tree.). Form from every vertex $v_i \in V$ the list $\Gamma_{v_i}^{-1}$ of its direct predecessors. Weight of a vertex v_i in the rooted directed tree is defined to be the number of all predecessors of v_i.
2) Order the lists $\Gamma_{v_i}^{-1}$ according to the weights of their vertices.
3) Separate the chains $\sigma_j, j = \{1...l\}$ s.t. they pass through vertices of tree along the branches with maximal numbers of vertices.
4) Number the vertices of chains $\sigma_j, j = \{1...l\}$ s.t. the corresponding numbering ϕ put in the class Φ^*.

This algorithm has time complexity $O(n)$.

3) Projective graphs: Two of the most recent algorithms for the MLA problem were proposed last year by *Lluís Alemany-Puig*, *Juan Luis Esteban* and *Ramon Ferrer-i-Cancho* [6]. Before describing the first one, which ends up in a projective graph, we firstly analyze the main components-functions of it (the numbering of the below sub-algorithms follows the numbering of the authors):

The algorithm below calculates recursively a displacement of all nodes with respect to the placement of the centroidal vertex of the whole tree in the linear arrangement.

```
Algorithm 3.2.
Function $EMBED\_BRANCH(L^c, v, base, dir, relPos)$
Input: $L^c$ is the sorted adjacency list for $T^c$, $v$ is
    the root of the subtree to be arranged, $base$ is
    the displacement for the starting position of
    the subtree arrangement, $dir$ is a boolean
    variable which defines whether $v$ is to the left
    or to the right of its parent.
Output: $relPos$ contains the displacement from the
    centroidal vertex of all nodes of the subtree.

$C_v \leftarrow L^c[v]$
$before \leftarrow after \leftarrow 0$
$under\_anchor \leftarrow 0$
for $i = 1$ to $|C_v|$ with step 2 do:
    $v_i, n_i \leftarrow C_v[i]$
    $under\_anchor \leftarrow under\_anchor + n_i$
$base \leftarrow base + dir * (under\_anchor + 1)$
for $i = |C_v|$ downto 1 do:
    $v_i, n_i \leftarrow C_v[i]$
    if $i$ is even then:
        $EMBED\_BRANCH(L^c, v, base - dir * before,$
        $-dir, relPos)$
        $before \leftarrow before + n_i$
    else:
        $EMBED\_BRANCH(L^c, v, base + dir * after,$
        $dir, relPos)$
        $after \leftarrow after + n_i$
$relPos[v] \leftarrow base$
```

The algorithm 3.2. has time and space complexity $O(n)$.

The algorithm below recursively calculates the size of subtrees for rooted trees. The term $s(u, v)$ refers to the size of the subtree T_v^u, which is the subtree of T^u rooted at v.

```
Algorithm 4.1: Calculation of size of subtrees for
    rooted trees.
Function $COMP\_S\_RT\_REC(T^r, (u, v))$
Input: $T^r$ is the rooted tree, $(u, v)$ is a directing
    edge.
Output: $s$ is the size of $T_v^r$ in vertices and
    $S = \{(u, v, s(u, v)) \| (u, v) \in E(T_v^r)\}$.

$s \leftarrow 1$
for $w \in$ neighbors of $v$ do:
        $(s, S') \leftarrow COMP\_S\_RT\_REC(T^r, (v, w))$
        $s \leftarrow s + s'$
        $S \leftarrow S \bigcup S'$
$S \leftarrow S \bigcup (u, v, s)$
return $(s, S)$
```

```
Function $COMPUTE\_S\_RT(T^r)$
Input: $T^r$ is the rooted tree
Output: $S = \{(u, v, s(u, v)) \| (u, v) \in E\}$

$S \leftarrow \emptyset$
for $v \in$ neighbors of $v$ do:
    $(\_, S') \leftarrow COMP\_S\_RT\_REC(T^r, (r, v))$
    $S \leftarrow S \bigcup S'$
return $S$
```

The algorithm 4.1. has time and space complexity $O(n)$.

The algorithm below constructs the rooted sorted adjacency list of a rooted tree T^r.

```
Algorithm 4.2: Calculation of the sorted adjacency
    list for rooted trees.
Function $SORTED\_ADJACENCY\_LIST\_RT(T^r)$
Input: $T^r$ is the rooted tree.
Output: $L$ is the decreasingly-sorted adjacency list
    of $T^r$.

$S \leftarrow COMPUTE\_S\_RT$(T)
Sort the tuples (u, v, s) in $S$ decreasingly by s
    using counting sort.
$L \leftarrow \{\emptyset\}^n$
for $(u, v, s) \in S$ do:
    $L[u] \leftarrow L[u] \bigcup (v, s)$
return $L$
```

The algorithm 4.2. has time and space complexity $O(n)$.

We now present the main algorithm for the projective case:

```
Algorithm 4.3: Linear time calculation of an optimal
    projective arrangement.
Function $HS\_PROJECTIVE(T^r)$
Input: $T^r$ is the rooted tree.
Output: $\pi$ is an optimal projective arrangement.

$L^r \leftarrow SORTED\_ADJACENCY\_LIST\_RT(T^r)$
$relPos \leftarrow \{0\}^n$
$leftSum \leftarrow rightSum \leftarrow 0$
for $i = k$ downto 1 do:
    if $i$ is even:
        $EMBED\_BRANCH(L^r, v_i, rightSum, 1,$
        $relPos)$
        $rightSum \leftarrow rightSum + n_i$
    else:
        $EMBED\_BRANCH(L^r, v_i, -leftSum, -1,$
        $relPos)$
        $leftSum \leftarrow leftSum + n_i$
$\pi \leftarrow \{0\}^n$
$\pi(r) \leftarrow leftSum + 1$
$relPos[r] \leftarrow 0$
for each vertex $v$ do:
    $\pi[v] \leftarrow \pi[r] + relPos[v]$
return $\pi$
```

The algorithm 4.3. has time complexity $O(n)$.

The authors also suggested one more algorithm which ends up in a planar graph. We will present it in the same way as above by describing the main functions of it. With a few changes, the modified algorithm can lead us to a projective graph as we describe below.
The algorithm below is quite similar to the Algorithm 3.2 we presented above with the only difference that it refers to free trees.

```
Algorithm 2.1: Calculation of directional sizes for
    free trees.
Function $COMP\_S\_FT\_REC(T, (u, v))$
Input: $T$ is a free tree, $(u, v)$ is a directing edge.
Output: $s$ is the size of $T^u_v*$ in vertices and
    $S = \{(u, v, s(u, v)), (v, u, s(v, u)) \| (u, v) \in E(T^u_v*)\}$
$s \leftarrow 1$
for $w \in$ neighbors of $v$ do
    if $w \neq u$ :
        $(s', S') \leftarrow COMP\_S\_FT\_REC($T$, ($v,w$))$
        $s \leftarrow s + s'$
        $S \leftarrow S \cup S'$
$S \leftarrow S \cup \{(u, v, s), (v, u, n - s)\}$
return $(s, S)$

Function $COMP\_S\_FT(T)$
Input: $T$ is a free tree.
Output: $S = \{(u, v, s(u, v)), (v, u, s(v, u)) \| (u, v) \in E\}$

$S \leftarrow \emptyset$
Choose an arbitrary vertex $u_*$.
for $v \in$ neighbors of $u_*$ do:
    $(\_, S') \leftarrow COMP\_S\_FT\_REC(T, (u_*, v))$
    $S \leftarrow S \cup S'$
return $S$
```

The algorithm 2.1. has time and space complexity $O(n)$.

Simirarly to the algorithm 4.2, the algorithm below calculates a sorted adjacency list for free trees.

```
Algorithm 2.2: Calculation of the sorted adjacency
    list for free trees.
Function $SORTED\_ADJACENCY\_LIST\_FT(T)$
Input: $T$ is a free tree.
Output: $L$ is the decreasingly-sorted adjacency list
    of $T$.

$S \leftarrow COMP\_S\_FT(T)$
Sort the tuples $(u, v, s)$ in $S$ decreasingly by $s$ using
    counting sort.
$L \leftarrow \{\emptyset\}^n$
for $(u, v, s) \in S$ do:
    $L[u] \leftarrow L[u] \cup (v, s)$
return $L$
```

The algorithm 2.2. has time and space complexity $O(n)$.

The algorithm below finds a centroidal vertex of the free tree.

```
Algorithm 2.3: Calculation of a centroidal vertex of
    a free tree.
Function $FIND\_CENTROIDAL\_VERTEX(T, L)$
Input: $T$ is a free tree, $L$ is the sorted adjacency
    list of $T$.
Output: The function finds a centroidal vertex of $T$.

Choose an arbitrary vertex $u$.
while $true$ do:
    $(v, s) \leftarrow$ largest entry in $L[u]$
    if $s > n/2$ then $u \leftarrow v$
    else return $u$

Function $FIND\_CENTROIDAL\_VERTEX(T)$
Input: $T$ is a free tree.
Output: The function finds a centroidal vertex of $T$.

$L \leftarrow SORTED\_ADJACENCY\_LIST\_FT(T)$
return $FIND\_CENTROIDAL\_VERTEX(T, L)$
```

The algorithm 2.3. has time and space complexity $O(n)$.

Table I
MLA ALGORITHMS

Algorithm Author	Characteristics		
	Input	***Output***	***Time complexity***
Amaral	Small number of vertices	-	exponential
Shiloach	undirected trees	-	$O(n^{2.2})$
Chung	undirected trees, rooted trees	-	$O(n^2)$ or $O(n^\lambda)$ with $\lambda > \frac{log3}{log2}$
Iordanskii	undirected trees	planar graph	$O(n)$
Alemany-Puig, Esteban, Ferrer-i-Canch	undirected trees, rooted trees	planar, projective graph	$O(n)$

We now present the main function for this algorithm. Algorithm 4.6. calculates the arrangement of the input tree T^r using intervals of integers [a, b], where $1 \leq \alpha \leq b \leq n$, that indicate the first and the last position of the vertices of a subtree in the linear arrangement. First of all, the first loop is responsible for arranging all immediate subtrees of T^r_u. Then, we check if the immediate subtree we examine right now is to be arranged in the available interval furthest to the left or to the right of its parent u and we change the interval accordingly. The variable $side$ is initialized according to the side to which u has been placed with respect to its parent and it gets the opposite value at the end of the loop. Before that, we need to update the limits of the arrangement of T^r_u by increasing or decreasing the left or the right limit accordingly, which is determined by the value of the variable $size$. In the end, when all immediate subtrees have been arranged, only the root node u has to be arranged. But for it, $\alpha = b$, so $\pi[u]$ = α.

```
Algorithm 4.6: Optimal arrangement of a tree
    according to its sorted adjacency list.
Function $ARRANGE(L^r, u, \tau, \alpha, b, \pi)$
Input: $L^r$ is a rooted sorted adjacency list, $u$ is
    the root of the subtree to be arranged, $\tau$ is the
    position of u with respect to its parent, $[\alpha, b]$
    is the interval of positions of the arrangement
    where to embed $T^r_u$, $\pi$ is the partially constructed
    arrangement.
Output: The output is $\pi$ updated with the optimal
    projective arrangement for $T^r_u$ in $[\alpha, b]$.

$C_u \leftarrow L^r[u]$
if $\tau$ is right then $side \leftarrow right$
else $side \leftarrow left$
for $i$ from 1 to $|C_u|$ do:
    $v, n_v \leftarrow C_u[i]$
    if $side$ is left:
        $\tau_{next} \leftarrow left$
        $\alpha_{next} \leftarrow \alpha$
        $b_{next} \leftarrow \alpha + n_v - 1$
    else:
        $\tau_{next} \leftarrow right$
        $\alpha_{next} \leftarrow b - n_v + 1$
        $b_{next} \leftarrow b$
    $ARRANGE(L^r, v, \tau_{next}, \alpha_{next}, b_{next}, \pi)$
    if $side$ is left:
        $\alpha \leftarrow \alpha + n_v$
    else:
        $b \leftarrow b - n_v$
    $side \leftarrow opposite\_side$
$\pi[u] \leftarrow \alpha$
```

The algorithm 4.6. has time and space complexity $O(n)$.
We finally present the 2 main algorithms for the problem. The first one ends up in a projective arrangement while the second one ends up in a planar arrangement:

The Algorithm 4.4. firstly finds the sorted adjacency list for rooted trees by calling Algorithm 4.2. and then it calls the Algorithm 4.6. by giving it an empty initial arrangement. The choice of the starting $side$ here is arbitrary, we could choose left as well.

```
Algorithm 4.4: Linear time calculation of an optimal
     projective arrangement.
Function $ARRANGE\_OPTIMAL\_PROJECTIVE(T^r)$
Input: $T^r$ is a rooted tree.
Output: $\pi$ is an optimal projective arrangement.

$L^r \leftarrow SORTED\_ADJACENCY\_LIST\_RT(T^r)$
$\pi \leftarrow \{0\}^n$
$ARRANGE(L^r, r, right, 1, n, \pi)$
return $\pi$
```

As for the planar case, the algorithm below firstly finds the sorted adjacency list for free trees by calling Algorithm 2.2. Then, it finds a centroidal vertex of T and as above, it calls the recursive Algorithm 4.6. with the suitable initial parameters (again the choice of the variable $side$ is arbitrary).

```
Algorithm 4.5: Linear time calculation of an optimal
     planar arrangement.
Function $ARRANGE\_OPTIMAL\_PLANAR(T)$
Input: $T$ is a free tree.
Output: $\pi$ is an optimal planar arrangement.

$L \leftarrow SORTED\_ADJACENCY\_LIST\_FT(T)$
$c \leftarrow FIND\_CENTROIDAL\_VERTEX(T, L)$
$L^c \leftarrow ROOT\_LIST(L, c)$
$\pi \leftarrow \{0\}^n$
$ARRANGE(L^c, c, right, 1, n, \pi)$
return $\pi$
```

Both algorithms have time and space complexity $O(n)$.

One very useful result which can be easily obtained from the above algorithms is that an optimal planar arrangement for T is an optimal projective arrangement for T^c, where c is a centroidal vertex of T (Algorithm 2.3.). But of course an optimal planar arrangement for T is not always an optimal projective arrangement for T^r where $r \neq c$, because r may be covered.
Table I summarizes the characteristics of all the aforementioned algorithms.

III. Approximation techniques (heuristics)

A. Spectral sequencing

The *Spectral sequencing* approximation technique utilizes the properties of the eigenvalues of the *Laplacian* matrix of a graph to provide valuable information about the latter.

First, we are going to state some definitions necessary for the approximate solution.
Given a simple and undirected graph $G(V, E)$ where $|V| = n$, we call ψ the bijective function and discrepancy $\sigma_p(G, \psi)$ s.t.

$$\psi : V(G) \rightarrow 1, 2, ..., n$$
$$\sigma_p(G, \psi) := (\sum_{(u,v)\in E} |\psi(u) - \psi(v)|^p)^{\frac{1}{p}}, \text{for } p \in (0, +\infty)$$
$$\sigma_\infty(G, \psi) := \max_{(u,v)\in E} |\psi(u) - \psi(v)|$$

Also, we call $min - p - sum$ the minimal value

$$\sigma_p(G) := \min_{\psi} \sigma_p(G, \psi), p \in (0, \infty]$$

It is readily understood that the minimum linear arrangement problem is identical to finding the $min - 1 - sum$ if one just specifies ψ as the arrangement π that has been used throughout this analysis. The $Laplacian$ matrix of graph G can be computed from the formula below:

$$L(G) := D(G) - A(G)$$

, where $A(G)$ is the adjacency matrix (square matrix of order n) s.t.

$$A(G)[u, v] = \begin{cases} 1, & \text{if nodes } u, v \text{ are neighbors} \\ 0, & \text{otherwise} \end{cases}$$

and $D(G)$ is the diagonal matrix (square matrix of order n) s.t.

$$D(G)[u, v] = \begin{cases} d(u) = \sum_{k\in V} A[u, k] & \text{if } u = v \\ 0 & \text{otherwise} \end{cases}$$

The *Laplacian eigenvalues* i.e., the eigenvalues of $L(G)$ are denoted as λ_i and are enumerated in an increasing order and repeated equal times to their multiplicity, that is $\lambda_1 \leq \lambda_2 \leq ... \leq \lambda_n$. According to *Juvan* [7], one can apply the following steps to find an approximate solution to our problem (and other optimal linear labeling problems):

1) Compute matrices $A(G), D(G)$ and subsequently the *Laplacian matrix* $L(G)$.
2) Find the *laplacian* eigenvalues and sort them in increasing order, $\lambda_1 \leq \lambda_2 \leq ... \leq \lambda_n$.
3) Compute the eigenvector $x^{(2)}$ (Fiedler vector) which corresponds to the second smallest *laplacian* eigenvalue λ_2.
4) Determine labeling ψ^e s.t.

$$\text{If } x_u^{(2)} \leq x_v^{(2)} \text{ then } \psi^e(u) \leq \psi^e(v)$$

In simple words, labeling ψ^e for node u is the ranking in ascending order of the value of the *laplacian* eigenvector $x^{(2)}$ for node u. In the case where there exist $u, v \in V$ s.t. $x_u^{(2)} = x_v^{(2)}$, one can simply choose arbitrarily their relative order. An approximate solution to the MLA problem is this labeling $\psi^e = \pi$ with a lower bound

$$LB_{SO} := \lceil \lambda_2(G) \frac{n^2 - 1}{6} \rceil$$

B. Random and Normal layouts

Maybe the simplest way to generate approximate solutions consists in returning a random feasible solution [8]. For example, one can assign a random label to each vertex (random layout) or preserve the label of each input (normal layout), i.e.

$$\pi[i] = i, \forall i \in \{1...n\} \quad (10)$$

It is reasonable to understand that such methods give us bad results in general. On the other hand, the main advantage of them is that they lead us to feasible and time negligible solutions.

C. Successive Augmentation heuristics

The main idea of this family of heuristics [8] is that a partial layout is extended, vertex by vertex, until all vertices have been labeled, at which point the output cannot be further improved. More precisely, the best possible free label is assigned to the current vertex at each iteration. In the beginning, we assign label 0 to an arbitrary vertex. Then, at each step, a new vertex is added to the partial layout, to its left or to its right, according to the way that minimizes the current partial cost of the solution. The new vertex will be placed either to the left or to the right extreme of the layout according to the result of a function called $Increment(G, \pi, i, v_i, x)$, which returns the increment of the partial cost of the layout if we assign label x to vertex v_i. The steps of the algorithm are shown below:

```
Function $Increment(G, \pi, i, v_i, x)$:
    $\pi[v_i] = x$
    $c = 0$
    for $j$ from 1 to $i$ do:
        if $(v_i, v_j) \in E$:
            $c = c + |\pi[v_i] - \pi[v_j]|$
    return $c$

Function $Successive\_Augmentation(G)$:
    Choose an arbitrary ordering of vertices
    $v_1, v_2, ..., v_n$
    $\pi[v_1] = 0$
    $left = -1$
    $right = 1$
    for $i$ from 2 to $n$ do:
        $left\_inc = Increment(G, \pi, i, v_i, left)$
        $right\_inc = Increment(G, \pi, i, v_i, right)$
        if $left\_inc < right\_inc$:
            $\pi[v_i] = left$
            $left = left - 1$
        else:
            $\pi[v_i] = right$
            $right = right + 1$
    for $i$ from 1 to $n$ do:
        $\pi[i] = \pi[i] - left$
    return $\pi$
```

As for the initial ordering of the vertices, apart from arbitrary, it can also be normal (i.e. according to the initial numbering of vertices) or it can even follow BFS or DFS of the graph. If we represent the graph using adjacency lists, the time complexity of Successive Augmentation heuristics is $O(n^2 logn)$.

D. Local Search heuristics

Local Search [8] is a very important tool to approximate many combinatorial problems, especially due to its simplicity

and performance. The main idea here is to iterately improve a (usually) random generated solution by performing local changes on its combinatorial structure.
In order to execute Local Search in an optimization problem, we need to define several things first. First of all, we need a set of feasible solutions ($S = \{\sigma_i\}$). Then, we need to specify which is the cost function, that is a function $f : S \rightarrow R^+$ which maps every feasible solution to a numerical value. Finally, we need to define the concept of "neighborhood", which is a relation between feasible solutions that are "close" in some sense.

A general aspect of the Local Search Algorithm is shown below:

```
Function $Local\_Search()$:
    Select an (arbitrary) feasible solution $\pi$ for
    the problem.
    while (a condition is not satisfied) do:
        Select a neighbor of $\pi, \pi'$
        Find the difference $\delta = f(\pi) - f(\pi')$
        if $\delta$ is acceptable, $\pi = \pi'$
    return $\pi$
```

For our problem, the set of feasible solutions S is the set of all permutations of size n, while the cost function f is the $LA(G, \pi)$. There are several ways in which we could define the concept of "neighborhood" in the MLA problem, some of which are the following:
Definition 1: Two layouts are neighbors if one can move from one to another by flipping the labels of any pair of nodes in the graph.
Definition 2: Two layouts are neighbors if one can move from one to another by flipping the labels of two adjacent nodes in the graph.
Definition 3: Two layouts are neighbors if one can move from one to another by rotating the labels of any triplet of nodes in the graph.
The selection of the definition of the neighborhood is of course a complicated subject, because there are trade-offs between the different plans. Some of them may be very time-consuming, while others may be stuck in moderate solutions because of a small number of available neighbors.

E. Hillclimbing

The hillclimbing heuristic [8] operates as follows: first, it selects an initial arbitrary layout at random. Then, it generates random moves from this state and the moves are accepted if they result in an improved cost. If no such moves are found after a specified max number of efforts, the algorithm terminates and returns the current solution. The steps of the hillclimbing heuristic are shown below:

```
Function $Hill\_Climbing(G, max\_iterations)$:
    $current\_tries = 0$
    Generate an initial random layout $\pi$.
    $current\_cost = LA(G, \pi)$
    while ($current\_tries < max\_iterations$) do:
        $current\_tries = current\_tries + 1$
        $u = random\_int(1, n)$
        $v = random\_int(1, n)$
        while ($v = u$), $v = random\_int(1, n)$
        $\pi' = flip(u, v, \pi)$
        /* it returns the permutation if we flip the
      labels of nodes $u, v$. */
        if $f(\pi') < f(\pi)$:
            $\pi = \pi'$
            $current\_tries = 0$
    return $\pi$
```

Apart from flipping the labels of just 2 arbitrary nodes, we could use the alternate definitions of neighborhood as we saw above and for example choose 3 random nodes and flip this trio. As it is understood, the parameter of $max\_iterations$ should be defined. If it is too small, the algorithm will terminate in negligible time but in general it will give us bad results. If it is too high, the algorithm's execution time rises quickly.

F. A logn-approximation algorithm

Here we present an approximation algorithm proposed by *Satish Rao* and *Andrea W. Richa* [9]. First of all, the algorithm which we will present solves a variation of the MLA problem we presented in the introduction, where the edges are weighted with weights greater than or equal to 1. There are no important changes in the description of the problem, except from the formula of the objective function, which now is:

$$\sum_{(i,j) \in E} |\pi(i) - \pi(j)| \cdot w(i,j) \tag{11}$$

where $w(i, j)$ represents the weight of the edge (i, j). We will firstly give some definitions before proceeding to the main algorithm.
The notion of a $level$ according to some l follows: given a node v, an edge (i, j) belongs to level x with respect to v if and only if $dist(i, v) \leq x$ and $dist(j, v) > x$ for any $x \in N$. According to the above definition, there might be edges which belong to more than one level, while other edges may not belong to any level. Moreover, the weight of a level i, i.e. ρ_i is defined as the sum of the weights of the edges at level i.
We are now ready to present the steps of algorithm, which are shown below:

1) Select any node v in the graph.
2) Assign the edges into levels: An edge (i, j) belongs to level x with respect to v if and only if $dist(i, v) \leq x$ and $dist(j, v) > x$ for any $x \in N$.
3) Partition levels according to their weights: Without loss of generality, we assume that $logW$ is an integer (if not, we can take $\lceil logW \rceil$). We partition the levels into $logW$ groups, according to indices assigned to the levels. Let a^k for all k in $[(logW) + 1]$. Level i has index k, k in $[logW]$ if and only if ρ_i belongs to the interval $(a^k, a^{k+1}]$. Select an index k s.t. there are $m \geq \frac{n}{4 * logW}$ levels with this index.
4) Cut along selected levels: For all i, let level a_i be the ith level of index k, in increasing order of distances to v. Let H_i be the subgraph induced by the nodes that are at distance greater than a_i and at most a_{i+1} from v (H_0 is the subgraph induced by the nodes that are at distance at most a_1 and H_m is the subgraph induced by

the nodes that are at distance greater than a_m) from v. Let n_i be the number of nodes in H_i.

5) Recursive step: We call the algorithm recursively on each H_i, obtaining a linear arrangement σ_i for the n_i nodes in this subgraph.
6) Combine solutions: Last but not least, we combine the linear arrangements obtained from each H_i and we end up in a solution as shown below:

$$(\sigma(1), ..., \sigma(n)) =$$
$$(\sigma_0(1), .., \sigma_0(n_0), \sigma_1(1), .., \sigma_1(n_1), .., \sigma_\kappa(1), .., \sigma_\kappa(n_\kappa))$$

The algorithm runs in polynomial time, since each recursive step runs in polynomial time and at each step, we decompose a connected component into at least 2 connected components. It can be shown that the cost of a solution in MLA problem obtained by the above algorithm is an $O(logn)$ factor of the cost of the optimal MLA of the graph.

G. Spreading metrics

The *Spreading metrics* approximation technique exploits the fact that the MLA problem can be analysed as a finite metric space of negative type and be addressed with semi-definite programs. In this case, the problem can be redefined through a *spreading metric* relaxation, that is given a weighted graph $G(V, E, w)$ we want to minimize the value below:

$$\sum_{(u,v)\in E} w(u, v) \cdot d(u, v)$$

with these constraints in mind:

- For $\forall(u, v)$ with $u, v \in V$, $d(u, v) \geq 1$
  For $\forall S \subseteq V$ with $|S| \geq 2$, and $\forall u \in S$,

$$\sum_{v\in S} d(u, v) \geq \frac{|S|^2}{5}$$

- (V, d) is a metric space which means that $\forall(u, v, w)$ with $u, v, w \in V$,

$$d(u, v) \leq d(u, w) + d(w, v)$$

If $d(u, v)$ is defined as $d(u, v) = ||x_u - x_v||_2^2$ (squared Euclidean norm, i.e. $||\mathbf{u}||_2^2 = \sum_i u_i^2$) with $x_u \in \mathbb{R}^n$ for $\forall u \in V$, then the program optimizing the *sum* can be solved with *Semidefinite programming* (SDP) and the metric space (V, d) is of negative type. Additionally, a metric space (V, d) is called $\epsilon - separable$ if for $\forall S \subseteq V$ with $|S| = k \geq 2$, there exist two $A, B \subseteq S$, s.t. $A, B \neq \emptyset$ and $|A|, |B| = \Omega(k)$, and $d(A, B) \geq \epsilon k$, where $d(A, B) = \min_{\alpha\in A, b\in B} d(\alpha, b)$. Moreover, we mention the definitions below:

$$W_S(d) = \sum_{(u,v)\in E: u,v\in S} w(u, v)d(u, v)$$

$$W(d) = W_V(d) = \sum_{(u,v)\in E} w(u, v)d(u, v)$$

$$W_k = \sum_{(u,v)\in E, u\in A_k, v\in B_k} w(u, v)$$

which is the cost of the edges that cross a cut C_k for which the following holds:

$$C_k \in \{C_0, ..., C_t\} \text{ for } t \geq \Omega(\epsilon n)$$

where C_i separates the vertices of V into two sets

$$A_i = \{u \in V : d(v, A) \leq i\}$$

and

$$B_i = V \setminus A_i$$

The algorithm [10] is described below based on two exclusive cases:

- There $\exists k \in \{0, ..., t\}$, s.t. $W_k \leq \frac{W(d)}{nlogn}$. The approximate solution to the minimum linear arrangement problem is found by following the steps below:
  1) Find recursively a linear arrangement for A_k.
  2) Find recursively a linear arrangement for B_k.
  3) Concatenate the results.

  The cost is computed as the sum of the cost of edges within A_k, the cost of edges within B_k and the concatenation cost of edges connecting A_k and B_k
- For $\forall k \in \{0, ..., t\}, W_k > \frac{W(d)}{nlogn}$. For this case an approximate solution can be determined as follows:
  1) Define buckets $B_0, ..., B_l$ with $l = O(loglogn)$ s.t. the bucket B_q contains all cuts C_k when

  $$W^q \leq W_k \leq 2W^q$$

  where

  $$W^q = 2^q \frac{W(d)}{nlogn}$$

  There $\exists B_q$ which contains at least $r = \Omega(\epsilon n/loglogn)$ cuts.
  2) By applying all the cuts in B_q to V, determine $V_1, V_2, ...$, s.t. $V_i \cap V_j = \emptyset$ and $\bigcup_i V_i = V$. The order of V_i's is natural linear according to the relative order of the corresponding cuts.
  3) Find minimum linear arrangement for $\forall V_i$ recursively.
  4) Concatenate the arrangements of the above step in the defined natural order.

  The cost is computed in a similar way as in the other case.

This algorithm produces solutions under an approximation ratio of $O(\sqrt{logn}loglogn)$ for general graphs $G(V, E)$ with $|V| = n$.

IV. Experimental evaluation

In this section, we present some experimental results aiming to evaluate the performance of some of the considered methods.

A. Experimental environment

The core of the programs has been written in Python programming language, using Anaconda 4.10.3 and more precisely Spyder 5.1.5. In order to obtain the optimal solution for each input graph, we used the Linear Arrangement

Library (LAL) implemented by *Lluís Alemany-Puig, Juan Luis Esteban* and *Ramon Ferrer-i-Cancho*. This library uses the algorithms of *Chung* [4] (the $O(n^2)$ algorithm) and *Shiloach* [2] (with its correction in [3]) which were discussed in Section II, in order to find the optimal solution for rooted trees. All experiments have been run on a PC with AMD Ryzen 7 3700U with Radeon Vega Mobile Gfx, 2.30 GHz processor and 12.0 GB (9.95 GB usable) memory with a Windows 11 Operating System.

B. Input Graphs

The input graphs are shortly shown in Table II. They are extracted from the GitHub repository (https://github.com/jordi-petit/graphs-minla), where *Jordi Petit* [8] has provided a set of graphs originally used to benchmark the MinLA problem. These graphs are very sparse and some of them originate from fields such as VLSI circuits and graph-drawing competitions. Apart from them, we also chose to generate random graphs, so we decided to generate Random Graphs based on the Gilbert Model, Random Graphs based on the Erdos-Renyi model, as well as Random Geometric Graphs. The reason for this decision is that such graphs might be amenable to a probabilistic analysis and as a result, general conclusions can be extracted from their study. Finally, we chose as an input graph the network of American football games between Division IA colleges during regular season Fall 2000, as compiled by M. Girvan and M. Newman as an alternate network from real life, as well as a 10-hypercube and a complete binary tree with 10 levels.

It should be noted that we processed all these graphs, in order to convert them into trees by producing their minimum spanning tree, so that the LAL can find the optimal solution for them.

All the aforementioned graphs are chosen for the reason below: First of all, the strong majority of them are "large", with the meaning that cannot be optimally solved by a brute-force algorithm in reasonable time. As a result, the chosen graphs have 500-1000 nodes (except from some graphs collected from the field of graph-drawing competitions). It was desirable to examine graphs with more than 1000 nodes but unfortunately our computational power was not enough for this; however, we strongly believe that even with these input graphs, we can extract some very useful conclusions regarding the quality of the solutions given by heuristics, as well as their time complexity.

In order to help to reproduce and verify the measurements and the code mentioned in this research, the code, instances and raw data are available on the Internet at our GitHub repository (https://github.com/IlianaXn/minimum-linear-arrangement-experiments).

C. Evaluation of the heuristics

The heuristics chosen to be implemented in our research are Random Layout, Normal Layout, Spectral Sequencing and the 3 versions of Local Search (with the 3 definitions of the term neighborhood as discussed in Section III). In order to evaluate the performance of these heuristics, summarized visualized results are given in Appendix. Here, we present some important observations that are worth discussing.

First of all, the results produced by *Random* and *Normal layout* are far from the optimal solution. This is obviously expected but we have to keep in mind that these results can be used in order to evaluate the quality of solutions given by the other heuristics.

Concerning the *Spectral Sequencing*, it can be easily observed that it is a method which produces moderate to good results in short time. It should also be mentioned that it is the only non-randomized algorithm which was implemented in this research, so it gives the same result whenever the input graph is the same. On the other hand, in our experiments we noticed that in some circumstances where the input graph is too large, it raises an Exception referring to lack of Memory. It is totally understood as it uses the Laplacian Matrix and works on eigenvalues on very large tables, so it needs strong computational power.

The most important observations concern the *Local Search* heuristic and the appropriate selection of the neighborhood of the current permutation. The simulations showed that the version of *Local Search* that gives the best result in the strong majority of graph instances is the version 2, that is the simple swap of 2 randomly chosen labels of the current permutation. The worst of all seems to be the version 2b, that is the swap of 2 labels whose nodes are adjacent in the graph. If we wanted to compare version 2 and version 3, we can conclude that the version 3 neighborhood will be stuck at fewer local minima than the version 2 neighborhood (it is easily shown with the time gap between these 2 versions). This is reasonable, because in version 3, it is more difficult to find good moves than in version 2.

The quality of solutions provided by *version 2 of Local Search* is quite satisfying. For all graphs, it provides the best approximation result among the implemented heuristics, and especially in smaller graphs the gap between its solution and the optimal solution is quite small. However, it should be emphasized that it demands the longest period of time in order to be executed among all heuristics here. Some executions of this algorithm took us hours to complete.

One important parameter in the *Local Search* algorithm is the stopping criterion. It determines when the algorithm should stop and return the best current result. Here, the stopping criterion refers to the maximum number of attempts where we didn't manage to find a better solution than the current solution. As it is obvious, the greater the max_tries is, the better the result that the algorithm will return will be. On the other hand, the greater the max_tries is, the longer the time period of execution will be. So here comes up an apparent trade-off. For our experiments, we tried to find the best balance between the quality of solution and the time complexity of it. As a result, we defined the maximum number of attempts equal to 5000. For smaller number of max_tries, e.g. 2000, the result provided by *Local Search* was worse even than the

Table II
INPUT GRAPHS

Name	**Characteristics**		
	Nodes	***Edges***	***Description***
bintree10	1023	1022	a complete binary tree with 10 levels
c1y	828	1749	graph from VLSI design
c2y	980	2102	graph from VLSI design
football	115	613	American Football Games
gd95c	62	144	graph from graph-drawing competitions
gd96b	111	193	graph from graph-drawing competitions
gd96c	65	125	graph from graph-drawing competitions
gd956d	180	288	graph from graph-drawing competitions
hc10	1024	5120	10-hypercube
randomA1	500	2454	Random Graph (Gilbert) with n=500 and p=0.02
randomA2	500	5004	Random Graph (Gilbert) with n=500 and p=0.04
randomA3	500	800	Random Graph (Erdos-Renyi) with n=500 and M=800
randomG1	500	2019	Random Geometric Graph with n=500 and radius=0.075
randomG2	500	5712	Random Geometric Graph with n=500 and radius=0.125

result of S*pectral Sequencing*. With the execution of *Local Search* with max_tries equal to 10000 to our small graphs, we noticed that the result provided especially from version 2 was approximately close to the optimal solution.

One last observation concerning *Local Search* is that it is a randomized algorithm, that is it depends on the initial permutation of labels. If we run twice the algorithm with the same graph as input, we will surely get different result, which is determined by the initial permutation. Furthermore, its execution time varies. If the initial permutation is close to the optimal, the algorithm will find a very good permutation in short time, but if the initial permutation is far from the optimal, it may run for hours by gradually improving the current solution. As we can see, the selection of a "good" initial permutation is very important. As a result, it could create thoughts for future work especially in the *Local Search Algorithm*: *How to find an initial permutation of labels (not at random as here) which will be "close enough" to the optimal permutation*.

As for the effect of the input graphs, it is totally reasonable that the larger the input graphs are, the greater the solution gap between optimal and heuristics is.

REFERENCES

[1] Amaral, A.R.S. ,"A mixed 0-1 linear programming formulation for the exact solution of the minimum linear arrangement problem", Optim Lett 3, 513–520, 2009.

[2] Y. Shiloach, "A minimum linear arrangement algorithm for undirected trees", SIAM J. Cam, 8, 15-32, 1979.

[3] Esteban J L and Ferrer-i-Cancho R, "A correction on Shiloach's algorithm for minimum linear arrangement of trees", SIAM J. Comput., 46, 1146–51, 2015.

[4] F.R.K. Chung, "An optimal linear arrangements of trees", Computers & Mathematics with Applications,10,43-60, 1984.

[5] M.A.Iordanskii, "Minimal numberings of the vertices of trees", Dokl. Akad. Nauk SSSR, 218, 2, 272-275, 1974.

[6] Lluís Alemany-Puig, Juan Luis Esteban, Ramon Ferrer-i-Cancho, "Minimum projective linearizations of trees in linear time", Information Processing Letters, Volume 174, 2022.

[7] M. Juvan, B. Mohar, "Optimal linear labelings and eigenvalues of graphs", Discr. Appl. Math. 36, 1992.

[8] Jordi Petit, "Experiments on the minimum linear arrangement problem". ACM J. Exp. Algorithmics 8, Article 2.3, 2003.

[9] Rao, S., Richa, A.W. "New approximation techniques for some linear ordering problems". SIAM J. Comput. 34(2), 388–404, 2004.

[10] Feige, Uriel & Lee, James, "An improved approximation ratio for the minimum linear arrangement problem", Information Processing Letter, 101, 26-29, 2007.

APPENDIX

Below, we present the plots produced based on the results of the experiments mentioned in section IV. Each figure is dedicated to one input graph and contains two plots. The first one (on the left) shows the ratio of the obtained result to the optimal result for each implemented heuristic. The second one (on the right) shows the required time for the execution of each heuristic, and additionally the execution time of the precise algorithms (*Chung*'s and *Shiloach*'s). It should be noted that sometimes the second plot seems to be missing several values. That is not the case; it simply implies that the execution time of these algorithms / heuristics is so smaller, and therefore negligible, than the others' ones that it cannot be depicted.

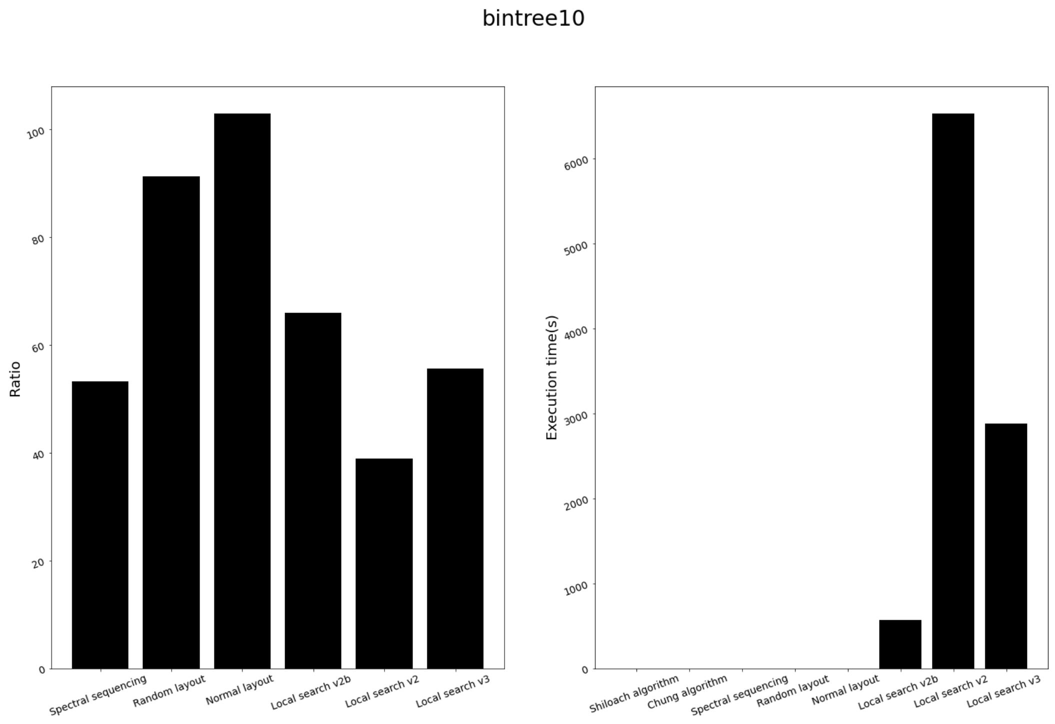

Figure 1. Bintree10

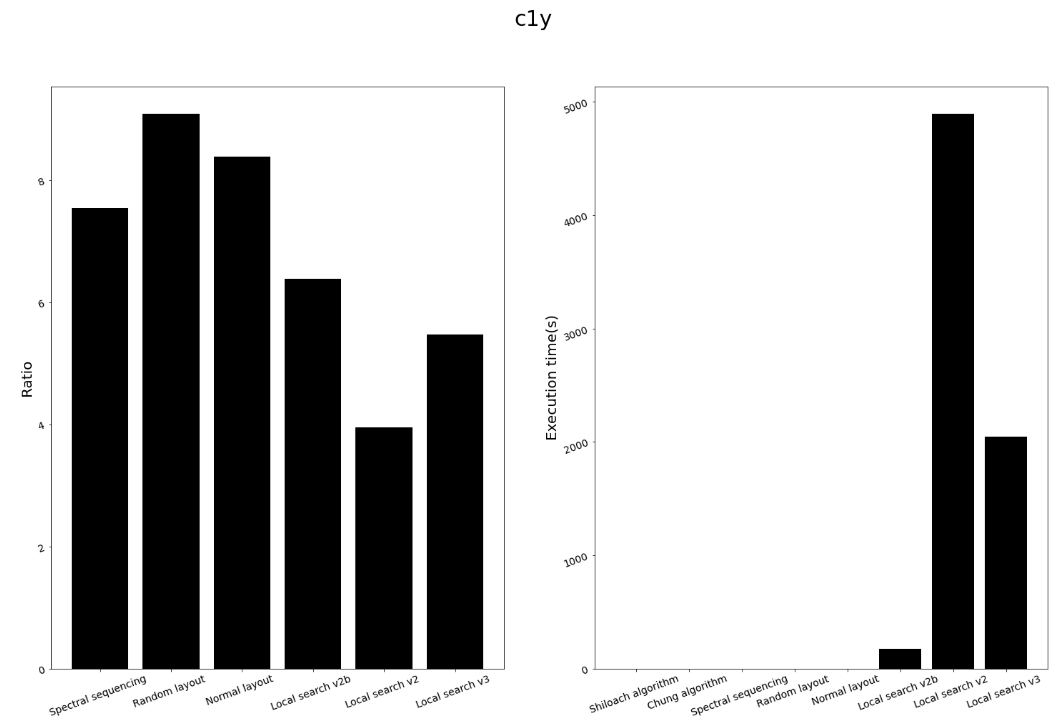

Figure 2. c1y

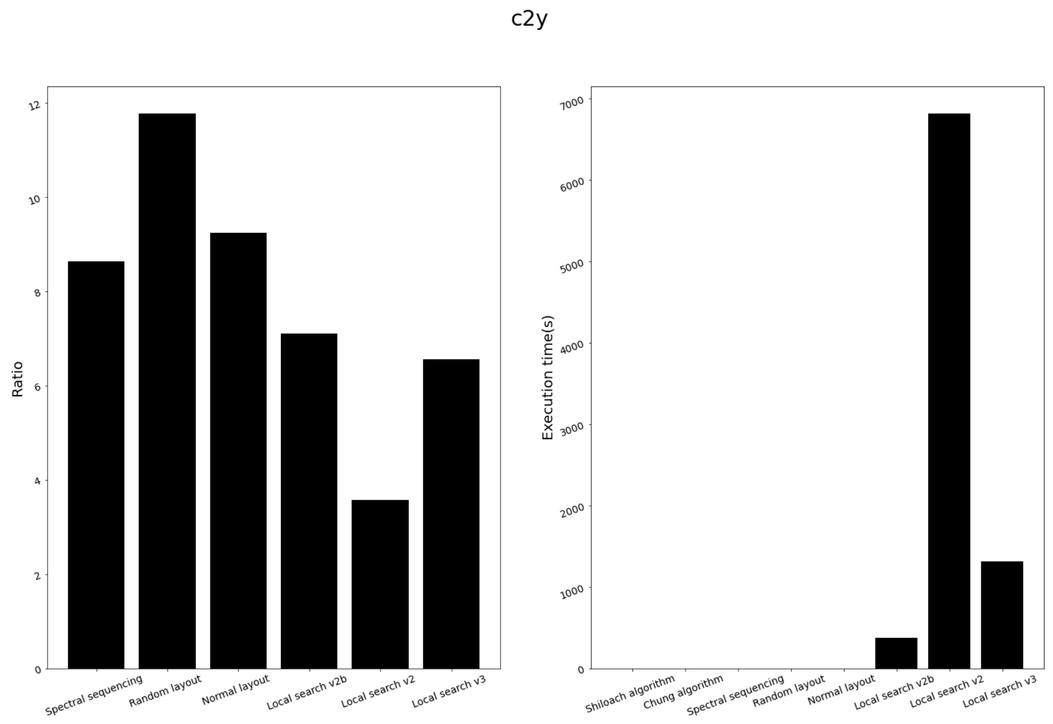

Figure 3. c2y

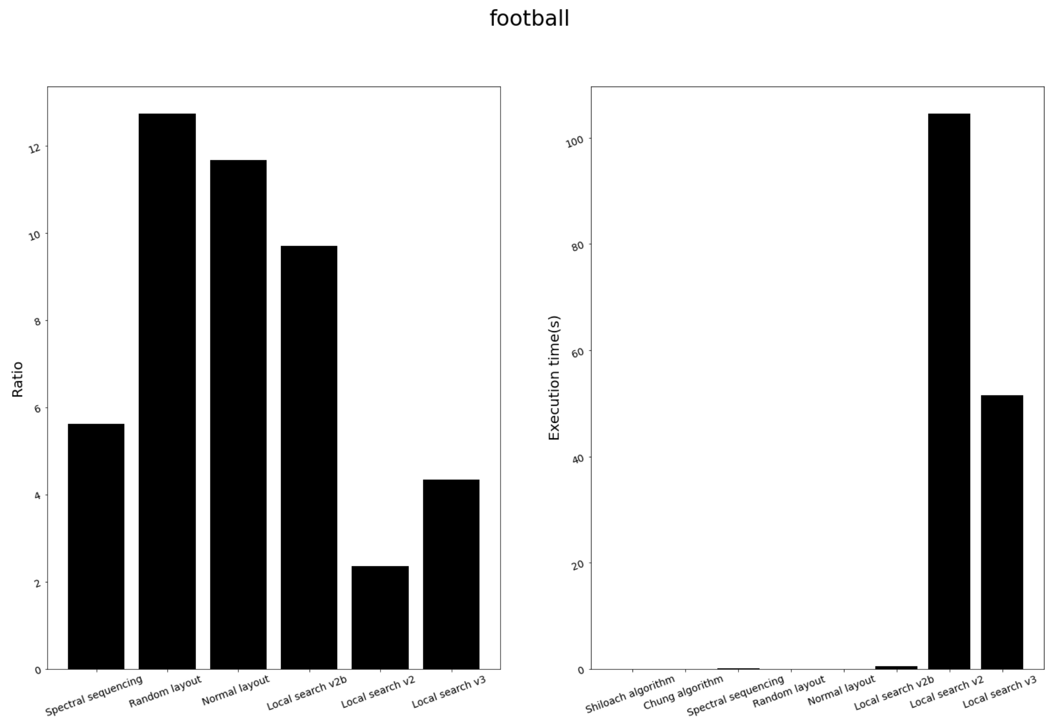

Figure 4. football

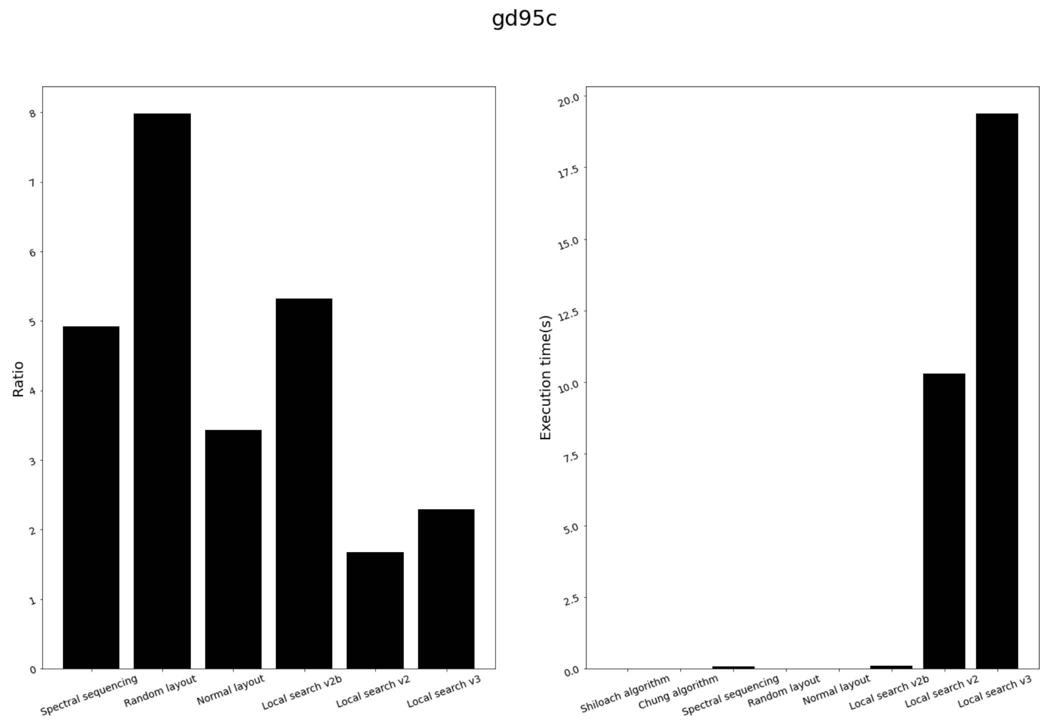

Figure 5. gd95c

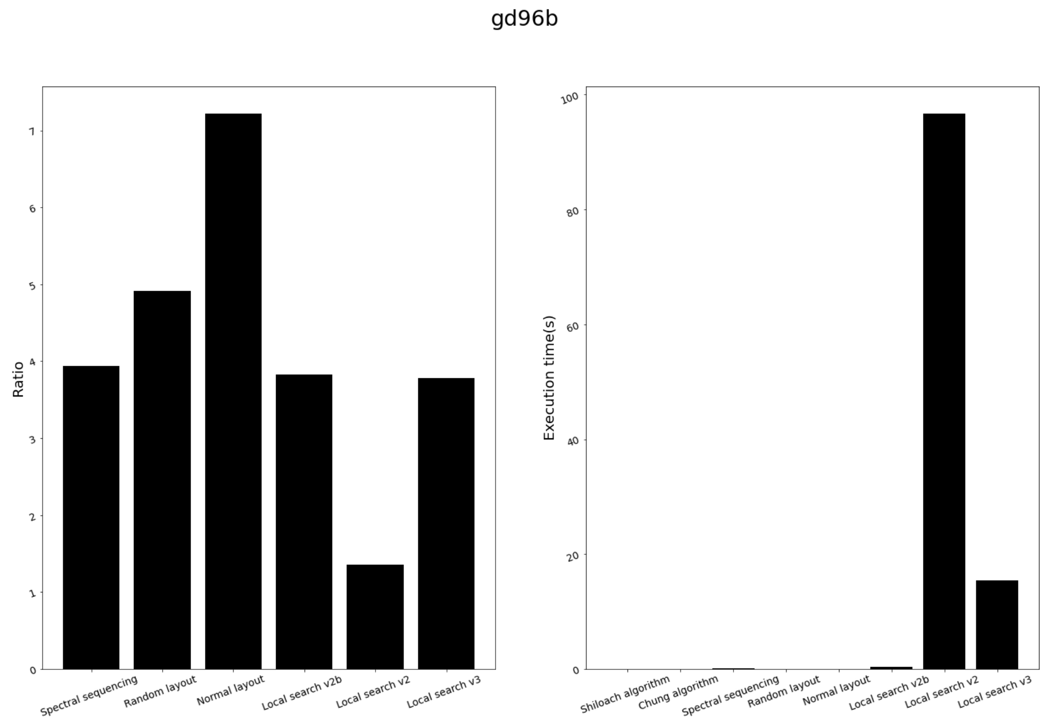

Figure 6. gd96b

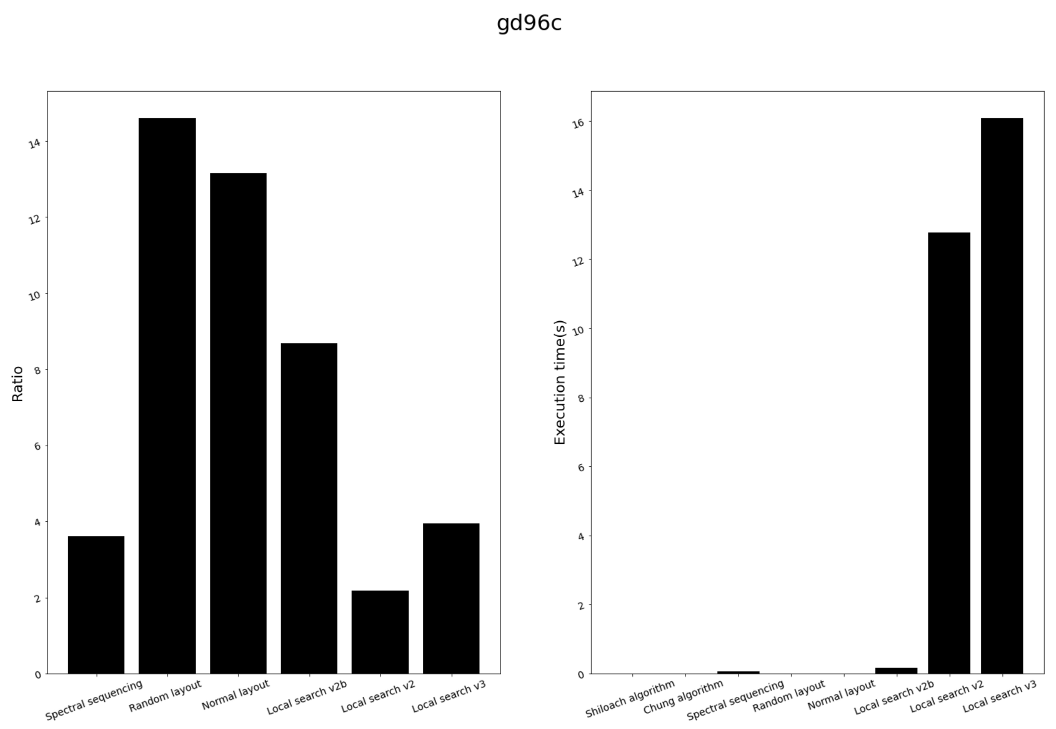

Figure 7. gd96c

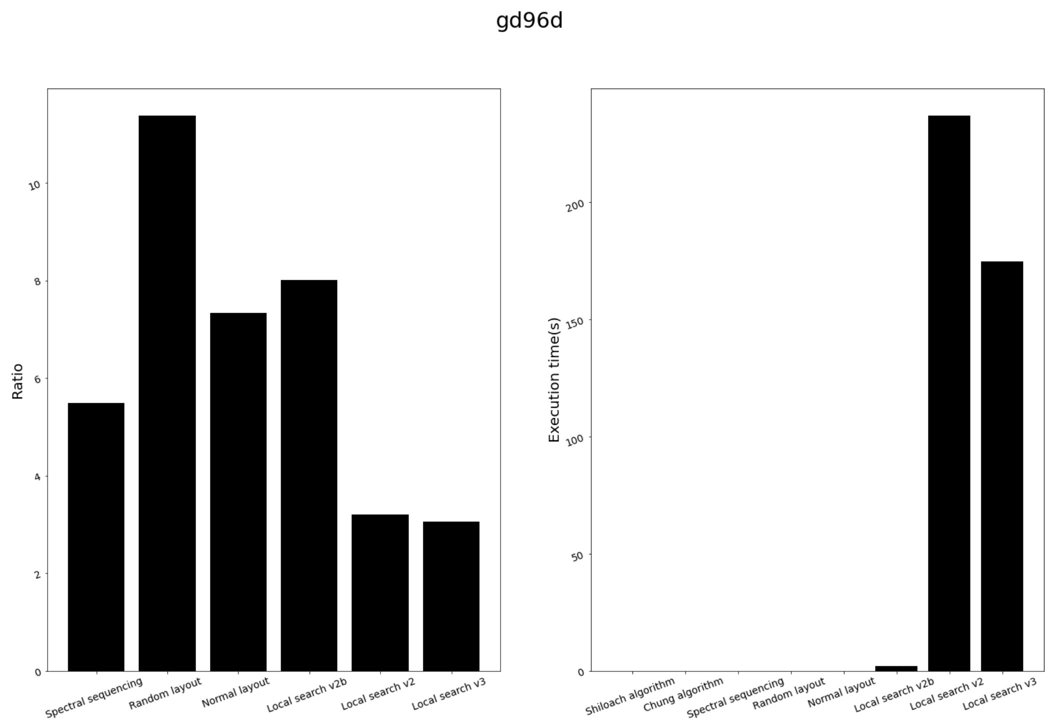

Figure 8. gd96d

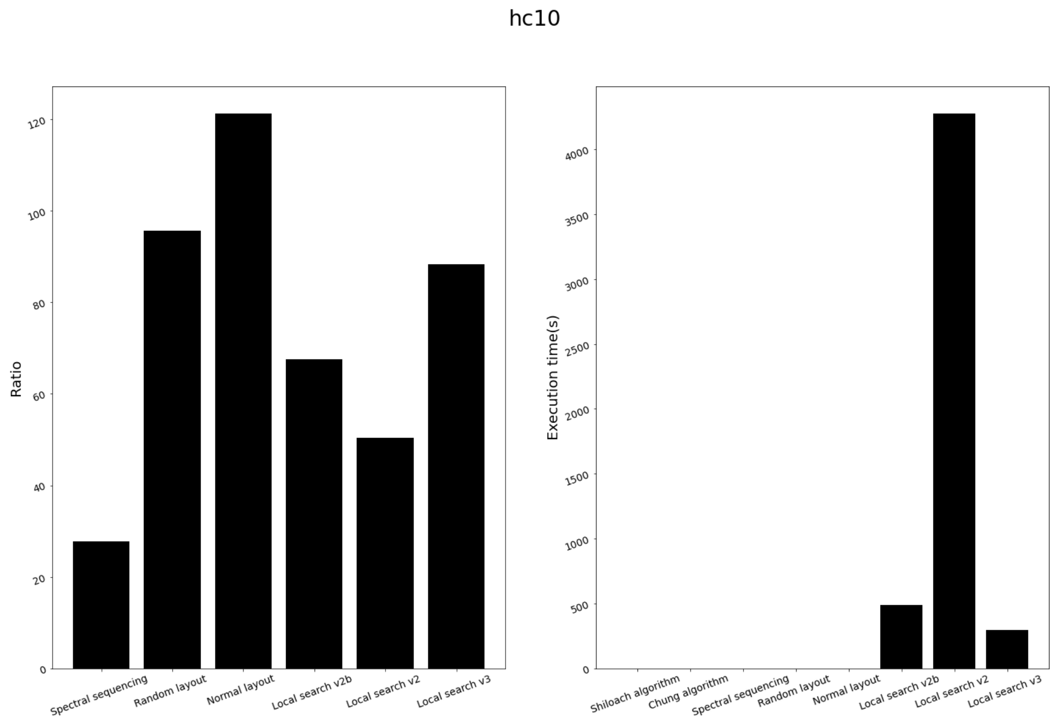

Figure 9. hc10

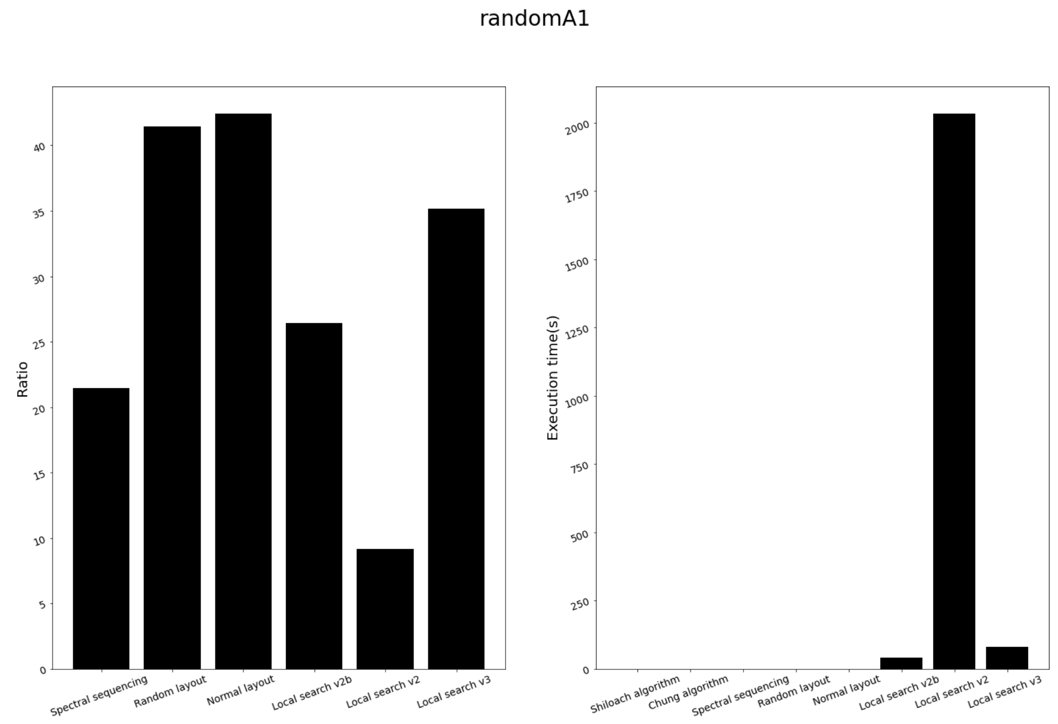

Figure 10. randomA1

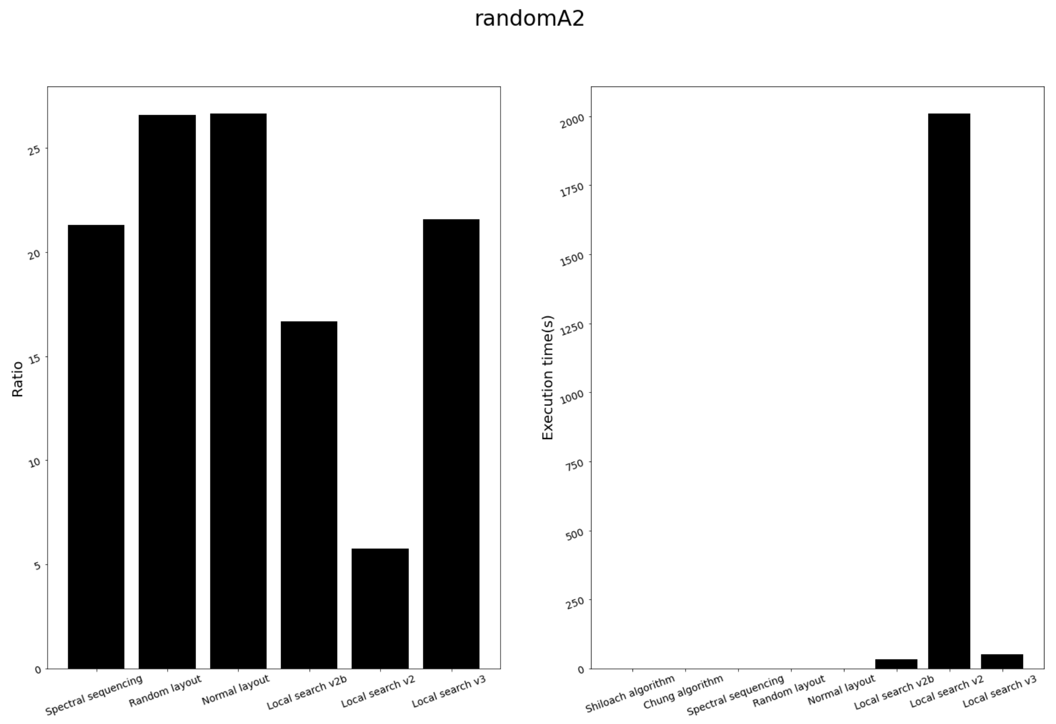

Figure 11. randomA2

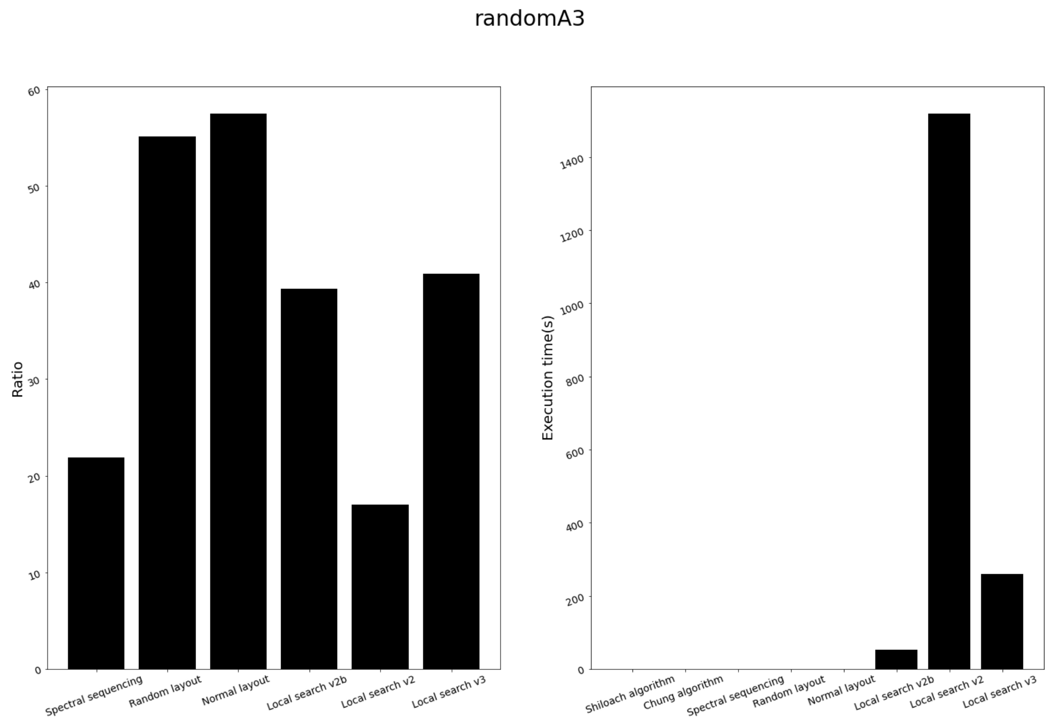

Figure 12. randomA3

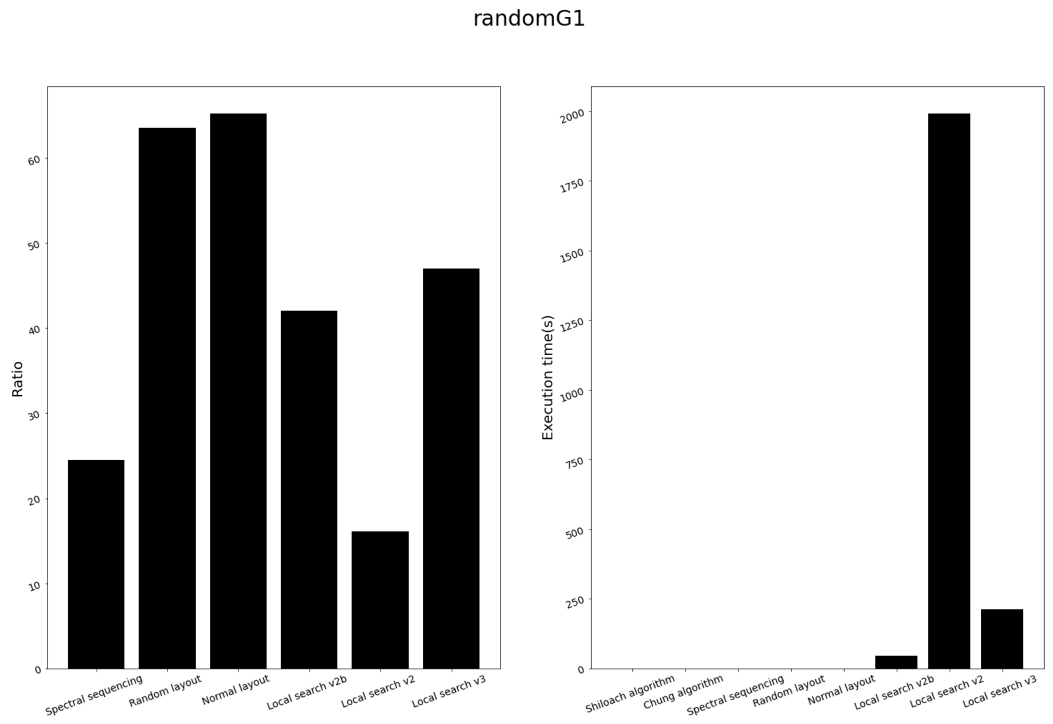

Figure 13. randomG1

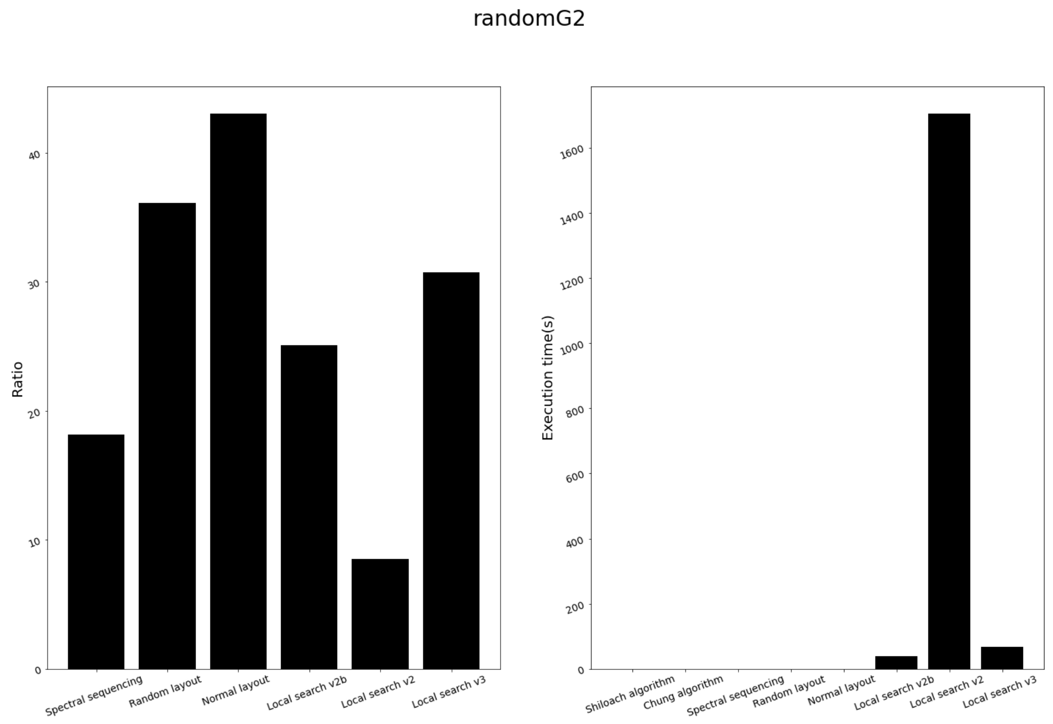

Figure 14. randomG2