

Distance-Guided Local Search

Daniel Porumbel · Jin-Kao Hao

the date of receipt and acceptance should be inserted later

Abstract We present several techniques that use distances between candidate solutions to achieve intensification in Local Search (LS) algorithms. One of the main drawbacks of classical LS is the fact that after visiting a very high-quality solution, an LS can often “forget it” and continue by exploring other very different areas. We propose a method that works on top of a given LS to equip it with a form of memory so as to record the highest-quality visited areas (spheres). More exactly, this new method uses distances between candidate solutions to perform a coarse-grained recording of the LS trajectory, *i.e.*, it records a number of discovered spheres. The (centers of the) spheres are kept sorted in a priority queue in which new centers are continually inserted as in insertion-sort algorithms. After thoroughly investigating a sphere, the proposed method resumes the search from the first best sphere center in the priority queue. The resulting LS trajectory is no longer a continuous path, but a tree-like structure, with closed branches (already investigated spheres) and open branches (as-yet-unexplored spheres). We also explore several other techniques relying on distances, *e.g.*, in Section 2.3, we show how to use distance information to prevent the search from looping indefinitely on large (quasi-)plateaus. Experiments on three problems based on different encodings (partitions, vectors and permutations) confirm the intensification potential of the proposed ideas.

Keywords meta-heuristic methodologies · local search · distance between solutions · intensification

D. Porumbel
CEDRIC, Conservatoire National des Arts et Métiers, 292, rue Saint Martin, Paris
daniel.porumbel@cnam.fr

J-K Hao
LERIA, Université d’Angers, 2 Boulevard Lavoisier, 49045 Angers, France
Institut Universitaire de France, 1, rue Descartes, 75231 Paris Cedex 05, France
jin-kao.hao@univ-angers.fr

1 Introduction

Local Search (LS) is one of the most popular methods for solving large and hard optimization problems in many fields of science. A drawback of many classical LS algorithms is that they lack a “global vision” over the search trajectory and evolution. Typically, even if an LS algorithm visits a very high-quality solution s at a given moment, it might often not intensify the search in the proximity of s , thus easily missing better solutions close to s .

This paper proposes Distance-Guided Local Search (DGLS), an algorithmic framework that operates on top of a given LS; the goal is to improve the underlying LS by introducing different intensification techniques that rely on a distance measure defined over the space of candidate solutions. The distance between two solutions s_1 and s_2 is measured as the minimum number of neighborhood transitions (moves) required to reach s_2 from s_1 . Using such a metric, we conveniently define the notion of $sphere(c, r)$: the set of solutions situated within a certain radius r from the sphere center c . In DGLS, each LS run launched from a center c is stopped as soon as the distance $d(s, c)$ between the current solution s and c reaches a maximum radius value.

The proposed method guides the underlying LS to intensively examine a part of the search space, *i.e.*, it selects certain spheres that are thoroughly investigated by launching a number of `runsPerSphere` LS runs from their centers. Each of the `runsPerSphere` LS runs launched from a center leads to the discovery of a new sphere center. All sphere centers are recorded in a priority queue that is sorted according to the objective values of the centers and also according to the distances between them. After launching these `runsPerSphere` LS runs from a center, DGLS can resume from a very distant center, *i.e.*, from the first center in the priority queue.

1.1 Context and Related Literature

Generally speaking, distances have already been used in the meta-heuristic literature, but in rather disparate research threads, with limited common objectives. We discuss below several such research subjects, most of them from the literature of evolutionary or memetic algorithms. To the best of our knowledge, there are very few systematic studies of the potential use of distances to improve LS.

The Tabu Search (TS) algorithm from [5] uses distances to make each step of the TS move to “solutions with increasing distance from the center solution”. The main idea is to prevent the search from coming back to a center solution, and to force the search to move away from it until a “prespecified search depth” is reached. When this depth is reached, the current iteration is finished; the search is then resumed to start a new iteration by selecting another center solution from the best solutions considered during the last iteration. Compared to our proposal, the TS from [5] is also more complex: it “resembles a genetic algorithm because a population of K members is main-

tained during the iteration”. Finally, the distances in this TS are not mainly used for intensification reasons, but rather for diversification, by forcing “the search away from previous solutions”.

In *Spacing Memetic Algorithms* (SMA) [16] one can find a formal evolutionary model devoted to a systematic control of the spacing (distances) between individuals in genetic algorithms. This framework uses distances to choose which individuals to insert into the population, which individuals to remove from the population, and when to perform mutations. However, the main purpose of SMA is diversification rather than intensification.

In *Geometric Genetic Algorithms* [12], the main evolutionary operators (mutation and crossover) are interpreted in the light of topological and geometric terms. We notice, for instance, that the definition of the notion of “closed ball” [12, §2.3] corresponds to a sphere in our study. However, this line of research is focused on evolutionary or genetic algorithms rather than local search.

More distantly related, standard genetic algorithms can (try to) locate the multiple global optima of a continuous multi-modal function. In this context, each optimum can be sought by a sub-population (niche) and one can promote crossover only inside subpopulations [11, 2] (“intra-niche” crossover), so as to “crowd” new individuals on the same niches. One popular method in this (continuous optimization) area is crowding [3, 2]; it attempts to induce “niches by forcing new individuals to replace individuals that are similar genomically” [17]. For this purpose, the eliminated individual is selected from among the closest individuals to the offspring solution in terms of distance.

Distances are also used for solution ranking in multi-objective optimization [4]. However, such diversity measures are typically calculated in the *objective function* space and they rely on fitness differences—not meaningful in our single-objective context.

The notion of distance is also considered in the population-based scatter search and path-relinking methods [7]. To generate new solutions from the existing ones, both solution quality and distances among solutions are taken into account to ensure the diversity of newly generated solutions.

The “limited discrepancy search” approach is often used in connection with exact methods to find the best solution within a certain “discrepancy” from a reference solution. The notion of “discrepancy” can be seen as a particular type of distance. The principle is also applied to design “limited-discrepancy” heuristics which take as input a rounded solution resulting from some relaxed formulations or column generation models. This is motivated by the fact that a high-quality starting solution could be obtained by rounding the fractional optimal solution generated by exact methods [9]. The idea is relevant for our study, because we will often launch DGLS from solutions not very far from an optimal solution.

We can conclude that the potential of distances to improve intensification in LS is not fully exploited; the existing studies that address this subject are rather scattered amongst the wider optimization literature. The most related

study is our TS-INT algorithm [14] designed for the graph k -coloring problem. In fact, the current study is based on and generalizes ideas from [14, §5].

1.2 Paper Organization

The remaining of the paper is organized as follows. Section 2 describes the proposed approach and the associated pseudo-code, considering the distance function as an external routine. Section 3 presents a distance function for each of the three problems considered in this paper. Section 4 is devoted to numerical results, followed by conclusions in the last section. In appendix, we provide more details on the underlying LS used for the k -coloring, the k -cluster, and the capacitated arc-routing problem. A second appendix provides 2D visualizations for three DGLS trajectories observed on the k -coloring problem. A final appendix is devoted to parameter tuning.

2 Distance-Guided Local Search

We now present the general Distance-Guided Local Search (DGLS) framework and its pseudo-code. We assume we are given a distance measure and an LS algorithm, which represent together the foundation upon which DGLS is built.

2.1 Main Principles: a Local Search with a Tree-like Trajectory

As hinted above, the proposed DGLS uses a distance measure d to compare candidate solutions, such that $d(s_1, s_2)$ is the minimum number of neighborhood transitions (moves) required to reach s_2 from s_1 . The notion of sphere is based on the given distance and it is defined in a straightforward manner.

Definition 1 Given a distance measure d in the search space, a candidate solution (center) c and an integer (radius) r , the sphere (c, r) is the set of all candidate solutions s such that $d(c, s) \leq r$.

If a sphere with numerous high-quality solutions is visited at a given moment, a classical LS could spend very limited time inside it and rapidly continue towards other areas of the search space. If the sphere is not examined intensively at the given moment, the opportunity of finding better solutions inside the sphere can pass by. To overcome such issues, DGLS will ensure an intensive examination of each sphere associated to high-quality solutions. This is achieved by performing several *LS runs* (parameter `runsPerSphere`) launched from the center of such a sphere. Each run is stopped as soon as it goes beyond the sphere boundary. This leads to a tree-like search trajectory: each investigated sphere center has `runsPerSphere` (child) branches.

The best solution visited during each LS run launched from a sphere center becomes a future center itself and it is inserted into a priority queue. This

best solution is simply the solution of minimum objective value, breaking ties according to the distance from the current center (the furthest solution is better).

The spheres in the priority queue can be sorted according to different (quality or diversity) criteria. The most frequently-used criterion is the objective value of the center, but one can also take into account the sum of the distances from the center to all other recorded spheres in the priority queue.

2.2 The General Pseudo-code of Distance-Guided Local Search

By putting together all the general principles from Section 2.1 above, we obtain the pseudo-code of the Distance-Guided Local Search in Algorithm 1; the goal is to minimize the objective value. The innermost `repeat-until` loop launches an LS run from the sphere center c . The outer loop at Lines 5–21 performs a *sphere examination* by launching `runsPerSphere` LS runs.

Algorithm 1 Distance-Guided Local Search (DGLS)

```

1:  $c \leftarrow \text{initial-candidate-solution}()$ 
2:  $\mathcal{Q}_{spheres} \leftarrow \{c\}$  ▷ the first sphere in the queue  $\mathcal{Q}_{spheres}$ 
3: repeat
4:    $c \leftarrow \text{dequeue}(\mathcal{Q}_{spheres})$ 
5:   loop runsPerSphere times ▷ This loop performs a sphere examination
6:      $s \leftarrow c$ 
7:      $bst \leftarrow c, \text{distBst} \leftarrow 0$  ▷ best solution of current run with  $d(bst, c) = 0$ 
8:     repeat
9:        $s \leftarrow \text{LS-Step}(s)$  ▷ increase an iteration counter here
10:       $\text{distToCenter} \leftarrow 0$ 
11:      if need-calc-dist() ▷ It might not be necessary to calculate the
12:         $\text{distToCenter} = d(c, s)$  ▷ distance at each iteration, see point 6 below
13:      end if
14:      if ( $\text{obj}(s) < \text{obj}(bst)$ ) or
15:        ( $\text{obj}(s) = \text{obj}(bst)$  and  $\text{distToCenter} > \text{distBst}$ )
16:         $bst \leftarrow s$ 
17:         $\text{distBst} \leftarrow \text{distToCenter}$ 
18:      end if
19:      until  $\text{distToCenter} > \text{maxRadius}$  or inner-stop-condition()
20:       $\text{insert}(bst, \mathcal{Q}_{spheres})$  ▷ insert it in the queue at the appropriate position
21:    end loop
22: until general-stop-condition() ▷ return best objective value ever reached

```

This pseudo-code relies on several external routines that we discuss below. The ideas presented next are actually general guidelines for implementing a DGLS; our goal is not to specify a would-be perfect DGLS variant, but to propose a set of distance-guided tools that could be mixed together in different ways to improve the intensification of a given LS. The implementation of the external routines below can depend substantially on the considered problem.

1. `initial-candidate-solution()` is a routine that provides the search process with an initial candidate solution that is either obtained by external means or generated at random.
2. `dequeue($Q_{spheres}$)` returns the center of the first sphere and removes it from the priority queue.
3. `LS-step(s)` calls the underlying LS operator to move from the current solution s to a neighboring solution s_{next} , returning s_{next} . By sequentially calling $s \leftarrow \text{LS-step}(s)$ multiple times, one actually executes the underlying LS. This LS should incorporate techniques to avoid getting blocked on a unique local optimum, *e.g.*, one should not use a simple deterministic Steepest Descent (or First Improvement) that is very prone to looping by visiting and revisiting the same local minimum again and again.
4. `insert($bst, Q_{spheres}$)` is a routine that establishes bst as a sphere center and inserts it at the appropriate position in $Q_{spheres}$. We recall that $Q_{spheres}$ is a priority queue that can be sorted according to different characteristics of the spheres. For instance, one can sort $Q_{spheres}$ lexicographically using two (minimization) criteria: (i) the objective value of the center; and (ii) the sum of distances to all other existing sphere centers. This approach seems well suited for graph coloring and arc-routing. For the k -cluster problem, we preferred to replace the above criterion (ii) with a First In First Out order.
5. `general-stop-condition()` and `inner-stop-condition()` indicate when a number of iterations (or a time limit) is reached, *e.g.*, one could use the iteration counter incremented at Line 9. We ask that `inner-stop-condition()` be stronger than `general-stop-condition()`, in the sense that it has to return `true` whenever `general-stop-condition()` returns `true`. One could make `inner-stop-condition()` return `true` after reaching a maximum number of iterations inside the current sphere, to avoid stagnation – see also Section 2.3 below.
6. $d(s, c)$ returns the distance from s to c . We mentioned at Line 12 that it is *not* necessary to compute this distance at every single iteration. Indeed, after computing a distance $d(c, s)$ at some iteration, the distance calculation can be skipped for the next $\text{maxRadius} - d(c, s)$ iterations, because, in the worst case, each iteration increases the distance to the center by one. As such, after $\text{maxRadius} - d(c, s)$ iterations, the distance to the center can become at most $d(c, s) + (\text{maxRadius} - d(c, s)) = \text{maxRadius}$, enough to be sure that the condition $(\text{distToCenter} > \text{maxRadius})$ at Line 19 is false, *i.e.*, the innermost `repeat-until` loop can not be broken. During these $\text{maxRadius} - d(c, s)$ iterations, `need-calc-dist()` can return `false`.¹ Finally, the condition $(\text{distToCenter} > \text{distBst})$ at Line 15 is not needed at each iteration, but only when the best objective value $obj(bst)$ is rediscovered; one can also imagine DGLS implementations that completely

¹ For instance, in practical cases for graph coloring, one can have $\text{maxRadius} = 100$ and if $d(c, s) = 20$ at some iteration, then the distance calculation can be skipped 80 iterations!

skip testing this condition at Line 15, so that *bst* simply becomes the last visited best solution.

The evolution of the general DGLS trajectory is controlled by the way spheres are sorted in the priority queue $\mathcal{Q}_{spheres}$. The sorting criteria determine how DGLS selects each new sphere to resume the search, which has an important impact on the search trajectory.

As long as the distance can be computed within a similar running time as an iteration of the underlying LS, the total distance calculation overhead can be kept within reasonable limits. While LS algorithms often use incremental (streamlined) objective function evaluations, this could also be done for the distance function. For example, we do perform such a streamlined distance calculation for the k -cluster (k -clique) problem, as described in Appendix A.2.

2.3 Using the Distance Mechanisms to Avoid Stagnation

It is well-acknowledged that an undesirable behavior of any heuristic algorithm is to be stuck looping on plateaus around a local optimum. Distance based-mechanisms could be very useful for detecting and tackling such issues; We propose the following mechanisms to be used alongside Algorithm 1:

1. Fix a maximum number of iterations per sphere, to ensure that DGLS can not stagnate looping indefinitely on a plateau inside a sphere. It is enough to make the function `inner-stop-condition()` stop the *sphere examination* after a number of iterations.
2. If the best solution *bst* visited by the current run launched from center c is too close to c (e.g., if $d(bst, c) < \frac{1}{2}\text{maxRadius}$), do *not* establish *bst* as a sphere center and do not insert it in the priority queue (i.e., skip Line 20). Choose instead the best visited solution situated at more than a threshold (e.g., $\frac{3}{4}\text{maxRadius}$) from the center c . Notice that by forbidding new sphere centers at less than $\frac{1}{2}\text{maxRadius}$ from c , we actually exclude a relatively small volume, e.g., a mini-sphere of $(\frac{1}{2})^k$ the volume of a standard sphere for the k -cluster problem (see Section 4.2).
3. If after a number of iterations `maxIterCheck` (e.g., use `maxIterCheck` $\in [2n, 3n]$, where n is the number of variables), the best solution *bst* visited by an LS run satisfies $obj(bst) = obj(c)$ and $d(bst, c) < \frac{1}{2}\text{maxRadius}$, then the current LS run might be stagnating by looping on a plateau around the center c . The algorithm should apply repulsion mechanisms to make this run leave the sphere. For example, on the k -cluster problem, we chose to increase the Tabu List length for: (i) the vertices selected by the current solution s but not selected by c , and (ii) the vertices not selected by s but selected by c . As such, the vertices that contribute to the Hamming distance $d(s, c)$ are fixed for a longer time. This naturally repulses the search from the center.

3 Problem Examples and Associated Distances

In this section, we introduce three distance measures for the following three well-known combinatorial optimization problems: graph k -coloring, k -cluster (or k -clique) and the capacitated arc-routing problem. These can be considered as representatives of three large classes of problems that require partition, binary and permutation representations.

3.1 A General Neighborhood Distance

We first provide the most general definitions of the distance function.

Definition 2 Given a set of candidate solutions (search space) S , an objective function and a neighborhood function $N : S \rightarrow 2^S$, the *landscape* $\mathcal{L} = (\mathcal{S}, E_N)$ is an attributed graph such that: (1) the vertex set \mathcal{S} is the set of candidate solutions, (2) there is an edge between two vertices (solutions) if and only if they are neighbors according to N , (3) each vertex (solution) is labeled with the objective value of the solution.

Definition 3 The *Neighborhood Distance* $d(s_1, s_2)$ is the shortest path between s_1 and s_2 in the landscape (\mathcal{S}, E_N) .

The distance $d(s_1, s_2)$ is an indicator of the minimum number of local search steps needed to reach solution s_2 from s_1 . This correlation is important, because without it an LS process could reach very distant solutions in a few steps, reducing the relevance of the distance value.

3.2 Distance Measures: Partitions, Arrays and Permutations

3.2.1 Graph k -Coloring

Given an input graph $G = (V, E)$ and a number of colors k , this problem asks to color V with k colors so as to minimize the number of conflicting edges (edges with both end vertices of the same color). The candidate solutions of this problem can be seen as partitions of the vertex set into k subsets. The distance is given by the *transfer partition distance* [8]. We recall [15] that the distance between partitions (colorings) C_a and C_b is $|V| - s(C_a, C_b)$, where s is a measure of similarity defined as follows:

$$s(C_a, C_b) = \max_{\sigma \in \Pi} \sum_{1 \leq i \leq k} M_{i, \sigma(i)},$$

where Π is the set of all bijections from $\{1, 2, \dots, k\}$ to $\{1, 2, \dots, k\}$ and M is a matrix such that M_{ij} indicates the cardinal of the intersection between the i^{th} color class of C_a and the j^{th} color class of C_b , *i.e.*, $M_{ij} = |C_a^i \cap C_b^j|$.

In most cases, the computation of this distance requires an asymptotic running time of $O(k^2 + |V|)$. In few other cases discussed in [15], the distance calculation can require $O(k^3 + |V|)$ time. However, both asymptotic running times are relatively large comparing to the complexity of a neighborhood evaluation. On the other hand, the distance does not need to be computed every single iteration, as we discussed at point 6 of Section 2.2.

3.2.2 k -cluster and k -clique

Given an input graph, this problem requires finding the densest induced subgraph with k vertices, *i.e.*, the induced subgraph with the maximum number of edges. The candidate solutions are represented as 0/1 arrays with exactly k ones corresponding to the selected vertices. The distance between two arrays can thus be given by the Hamming distance. In fact, it is the halved Hamming distance that constitutes a neighborhood distance in the sense of Definition 3 (with regard to the bit swap neighborhood). However, we generally prefer to express our ideas in terms of the standard Hamming distance – all calculations could have been equally done using the halved Hamming distance. It is worth noticing a particular aspect of this distance measure: when k is less than $n/2$, there might exist numerous pairs of solutions at distance $2 \cdot k$, *i.e.*, pairs of solutions corresponding to disjoint selections.

3.2.3 Capacitated Arc-Routing (CARP)

Given an input graph $G(V, E)$ with a set of required edges (clients) $E_R \subseteq E$, this problem asks to find the least-cost set of routes that service (visit) all edges E_R [6]. It is the edge-focused counterpart of the celebrated vehicle routing problem. Using the approach from [13], this problem is cast in the space of permutations, and so, it can be considered as a permutation problem [1]. More exactly, this approach uses a decoder that transforms any permutation (of the client set E_R) into a set of routes.

The metric used to evaluate the distance between two permutations is the *Kendal tau rank distance* [10]. In general terms, this counts the number of pairwise disagreements between the two permutations. The Kendall tau distance is also called the bubble-sort distance since it is equivalent to the number of swaps that the bubble sort algorithm would perform to place one permutation in the same order as the other. Technically, the distance between permutations τ_1 and τ_2 is:

$$d(\tau_1, \tau_2) = |\{(i, j) : i < j, (\tau_1(i) < \tau_1(j) \wedge \tau_2(i) > \tau_2(j)) \vee (\tau_1(i) > \tau_1(j) \wedge \tau_2(i) < \tau_2(j))\}| \quad (3.1)$$

We observe that this satisfies the properties of a neighborhood distance from Definition 3 if one uses a neighborhood defined by adjacent transpositions (*e.g.*, the adjacent interchange neighborhood). The calculation of this distance can be realized by comparing $\frac{n \cdot (n-1)}{2}$ pairs, *i.e.*, it requires an asymptotic running time of $O(n^2)$.

4 Numerical Experiments

Here, we report computational results on the three considered problems. We demonstrate that the numerical results of a given LS can be improved if this LS is embedded into the DGLS framework. This improvement is mainly due to the increased capacity of intensification induced by the distance mechanisms.

4.1 Graph k -Coloring Experiments

The underlying LS for graph k -coloring is the Tabu Search (TS) from [14, §2.2]. Essentially, this TS moves from solution to solution by changing the color of a vertex v in conflict (sharing its color with a neighbor). After replacing the current color of v by a new color, v can not receive again the lost color for the next $\mathbf{random}(10) + 0.6 \cdot \mathit{obj}(s) + \left\lfloor \frac{\mathit{iter}_{\mathbf{splat}}}{1000} \right\rfloor$ iterations, where $\mathbf{random}(10)$ returns a random integer in $[0..10]$, $\mathit{obj}(s)$ is the current objective value (*i.e.*, the number of conflicting edges), and $\mathit{iter}_{\mathbf{splat}}$ is the number of last iterations with no objective value variation.

The last term aims at keeping certain moves Tabu for a longer time when the TS is blocked looping on a plateau with no objective function variation. Each series of consecutive 1000 moves on such plateau lead to incrementing all subsequent Tabu list lengths. In the worst case, most moves that keep the TS on the plateau become Tabu for a long time, forcing the TS to choose other moves and stop looping. The algorithm also uses streamlining calculations to rapidly find the best neighbor, as described in Appendix A.1.

4.1.1 General results on graph coloring

The number of iterations for both LS and DGLS is set at $\mathbf{maxIter} = 300000 \left\lceil \frac{k}{100} \right\rceil$ (the last term allows more iterations for larger instances). The radius value is set at $\mathbf{maxRadius} = 0.2 \cdot n$ and the number of runs per sphere is $\mathbf{runsPerSphere} = 3$. These are two important parameters of DGLS which merit particular attention. Appendix C presents a test that evaluates the success rate of DGLS for different values of $\mathbf{maxRadius}$ and $\mathbf{runsPerSphere}$. Generally speaking, this test indicates that there is an interval of safe values for each parameter: any $\mathbf{maxRadius}$ value between $0.1 \cdot n$ and $0.6 \cdot n$ and any $\mathbf{runsPerSphere}$ value between 3 and 5 may lead to relatively good DGLS results. We did not use any of the stagnation avoidance techniques from Section 2.3.

Since DGLS is designed for improving the intensification potential of a given LS, it makes sense to first compare DGLS and LS by launching them from a coloring which is relatively close to an optimal solution (a legal k -coloring). We consider the following protocol. For each graph, we take the best legal coloring reported in our previous paper [14],² we modify a number

² Colorings available on line at cedric.cnam.fr/~porumbed/graphs/tsdivint/

Graph, k	Algo- rithm	Start dist.	Succes rate	Final objective values			Iterations to success			
				avg (std)	min	max	avg (std)	min	max	
1e450.25c, 25	DGLS	150	5/10	1.1 (1.1)	0	2	69901 (97969)	774	262654	
1e450.25c, 25	LS	150	2/10	2 (1.5)	0	5	68060 (66132)	1928	134192	
1e450.25d, 25	DGLS	190	5/10	2 (2.3)	0	6	40906 (46484)	3468	122994	
1e450.25d, 25	LS	190	1/10	2.9 (1.8)	0	5	70035 (0)	70035	70035	
flat300.28, 28	DGLS	200	5/10	12.3 (15.3)	0	36	72331 (84670)	2577	211600	
flat300.28, 28	LS	200	0/10	35.9 (2)	31	38	– (–)	–	–	
dsjc250.5, 28	DGLS	140	4/10	0.7 (0.6)	0	2	148541 (74090)	80443	273258	
dsjc250.5, 28	LS	140	0/10	1.5 (0.5)	1	2	– (–)	–	–	
dsjc500.1, 12	DGLS	300	6/10	1.1 (1.9)	0	6	31430 (38929)	1794	112656	
dsjc500.1, 12	LS	300	0/10	3.3 (1.3)	1	5	– (–)	–	–	
dsjc500.5, 48	DGLS	230	5/10	0.8 (0.9)	0	2	66981 (55317)	20765	173355	
dsjc500.5, 48	LS	230	3/10	3.8 (3.8)	0	11	4900 (3779)	1985	10237	
dsjc500.9, 126	DGLS	150	10/10	0 (0)	0	0	20398 (36871)	933	125062	
dsjc500.9, 126	LS	150	4/10	0.7 (0.6)	0	2	61623 (87229)	1022	210958	
dsjc1000.1, 21	DGLS	800	4/10	1.3 (1.4)	0	4	153604 (42788)	80134	185540	
dsjc1000.1, 21	LS	800	1/10	2.4 (1.1)	0	4	257524 (0)	257524	257524	
dsjc1000.5, 85	DGLS	450	4/10	8.1 (7.8)	0	25	116658 (43243)	53091	160702	
dsjc1000.5, 85	LS	450	0/10	15.3 (7.2)	3	25	– (–)	–	–	
dsjc1000.9, 223	DGLS	250	10/10	0 (0)	0	0	13631 (35213)	511	119226	
dsjc1000.9, 223	LS	250	4/10	0.9 (0.8)	0	2	5904 (8547)	642	20701	

Table 1: Comparison of DGLS and standard LS launched from a coloring obtained by randomly modifying a number of colors (“Start dist.” in Column 3) of a legal coloring. DGLS has significantly higher success rates.

of colors (at least $\frac{|V|}{3}$) and we launch DGLS and standard LS from the resulting modified coloring.

Table 1 presents this comparison of DGLS and LS, reporting the instance in Column 1 (the graph and the number of colors), the algorithm version in Column 2 (one row on DGLS, one row on LS), the above number of modified colors of a legal coloring (Column 3), the number of successful executions (finding a legal coloring) out of 10 (Column 4), followed by statistical results on the final objective values reported at the end of the 10 executions (Columns “Final objective values”) and statistical results on the number of iterations needed by the successful executions (last 4 columns). The statistical results include: the average value (columns “avg”), the standard deviation (columns “std”), the minimum value (columns “min”) and the maximum (columns “max”).

Table 1 shows that DGLS is able to find the path towards an optimal solution twice or three times more often than the standard LS. Notice that DGLS does not find the optimum only in the beginning of the search (by directly re-constructing the original optimal solution). It might need sometimes more than 150000 iterations to reach an optimal solution, after having examined tens or hundreds of spheres. We will see in Section 4.1.2 below that a *sphere examination* can often take less than 1000 iterations.

We now consider a different experimental protocol, using the same parameters as above. We execute 5 times 300000 iterations of the underlying LS and we take the best solution ever visited. Then, we launch from this solution 10 times DGLS and 10 times the underlying LS. Table 2 compares the

k -coloring instance Graph, k	LS			DGLS		
	bst	avg	worst	bst	avg	worst
1e450.25c.col, 24	21	22.7	24	20	21	22
1e450.25d.col, 24	20	21.5	23	20	21.1	22
flat300.28.0.col, 30	29	31	32	24	26.4	28
dsjc250.5.col, 27	6	6.5	7	6	6.1	7
dsjc500.1.col, 11	27	27.1	28	27	27	27
dsjc500.5.col, 47	19	23.2	26	19	22	24
dsjc500.9.col, 125	4	4.1	5	4	4	4
dsjc1000.1.col, 19	34	34.6	35	34	34.1	35
dsjc1000.5.col, 82	82	91.3	98	82	86.7	93
dsjc1000.9.col, 221	9	10.7	13	7	9.3	11

Table 2: Graph k -coloring result on overly-difficult instances; both LS and DGLS were launched from the best solution visited by 5 LS runs of 300000 iterations.

results obtained by DGLS and LS following this protocol on a set of overly difficult instances (the number of colors one unit lower than the best known upper bound). For both algorithms, Table 2 reports the minimum (bst), average (avg) and maximum (worst) number of conflicting edges (edges with both end vertices of the same colors) obtained over 10 executions. Notice DGLS achieves improved results on all instances, with regards to all three criteria.

4.1.2 Insights into the sphere examinations

Natural questions regarding DGLS include:

- What does the global trajectory of DGLS looks like?
- How many candidate solutions are usually visited during a *sphere examination*?
- How many iterations can take a run launched from a center, or equivalently how long is the innermost `repeat-until` loop of Algorithm 1 in Section 2.2 ?
- What is the average distance from the center c to the best solution found by a run launched from c ? Do different runs launched from c lead to finding similar best solutions?

To provide an intuitive answer such questions, we propose using a Multidimensional Scaling (MDS) procedure that creates a 2D visualization (projection) of the visited sphere centers and of the distances between them. This MDS procedure³ takes as input a matrix of distances (between colorings) and generates a set of Euclidean points such that the distances between these points represent an approximation of the initial distances. The quality of this approximation can be evaluated using a loss function (the Kruskal stress). In our cases, the value of this loss function is usually between 0.2 and 0.3.

³ We used the tool MDSJ “Java Library for Multidimensional Scaling (Version 0.2)” from University of Konstanz, available on-line at <http://algo.uni-konstanz.de/software/mdsj/>

Regarding the quality of the MDS projections, we can discuss an example on Figure 1. The table on the right provides the *real* distances between the points **START**, 1, 2, 3, 4 and 5. One could check the Euclidean distances in the figure are approximately not far from the real distances in the table.

	START	1	2	3	4	5
START	0	50	50	50	96	97
1	50	0	33	36	69	71
2	50	33	0	22	50	50
3	50	36	22	0	60	62
4	96	69	50	60	0	26
5	97	71	50	62	26	0

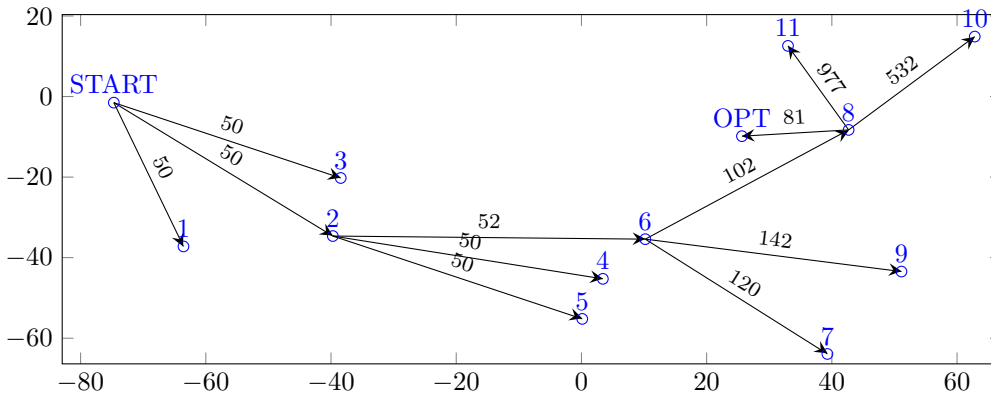


Fig. 1: MDS plot of the running profile of a short successful DGLS execution on `dsjc250.5` with `maxRadius = 50`. Each point represents a sphere center; each arrow $i \xrightarrow{\text{iters}} j$ indicates that the sphere center j was discovered in iters iterations by a run launched from i . The starting point labelled **START** was generated by randomly modifying 100 colors of a legal coloring. **OPT** is the optimal solution found by DGLS.

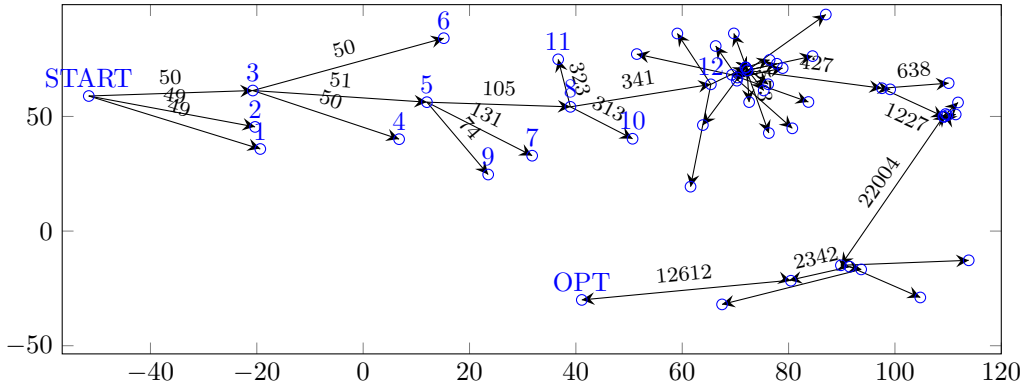


Fig. 2: MDS plot of the running profile of a longer DGLS execution on `dsjc250.5` with `maxRadius = 50`, using the same starting point as in Figure 1. After a long intensified search close to sphere center 12, DGLS eventually finds its way towards an optimal solution.

Figures 1-2 plot the MDS representations of two DGLS executions on the smallest random graph `dsjc250.5` with $k = 28$ colors. Each arrow represents a

run launched from a sphere center (start point). The end point of the arrow is the best solution visited by the run (that also becomes a future sphere center). The labels in blue indicate the order of the discovery of the centers and the figures above each arrow indicate the run length in iterations.

We can safely conclude from Figures 1-2 that the number of iterations of a run can vary from $50 = \text{maxRadius}$ to values of hundreds or thousands. In the beginning, the starting solution has many conflicts that can be solved directly, making the search rapidly leave the proximity of the starting solution. Naturally, DGLS finds sphere centers of increasingly improved quality over the time, and so, the search process spends more iterations on plateaus close to such centers; thus, later runs require more iterations. The distance between the sphere center and the best coloring reported by a run can evolve from very large values in the beginning (close to maxRadius) to values close to zero (this can be seen in Figure 2, starting with center 12).

We also notice that, in the beginning, the three runs launched from a center follow quite similar paths, *i.e.*, observe the three arrows originating from the point **START** in both figures. The underlying Tabu Search is basically executing three times a similar Steepest Descent, as the center **START** has many conflicts that can be easily solved. However, towards (the middle and) the end of the DGLS execution, we observe the opposite behaviour: we notice a star-like shape of three arrows originating at each point, *i.e.*, the three runs launched from the same center can seriously diverge in all directions.

As expected from theory, Figures 1-2 suggest that DGLS does follow a tree-like trajectory. The execution in Figure 2 is more challenging: there are quite numerous arrows pointing towards the top of the figure, representing runs could lead DGLS away from the optimal solution (see centers 6, 11, and those above 12). These branches were fortunately cut by DGLS; its intensification mechanism managed to keep the main search process on a region not far from the optimal solution.

The above conclusions are generally confirmed by other MDS representations for DGLS executions on different graphs. We refer the reader to Appendix B for more MDS figures of other DGLS trajectories.

4.1.3 Comparing to random restarts and other sphere ranking criteria

Let us explore other DGLS variants, to gain more insight into the impact of the individual components that constitute DGLS. We will also compare these DGLS flavors with two standard LS methods that do use restart mechanisms as well. Specifically, we consider the following four algorithms:

- 1 A DGLS version in which the second criterion for ranking spheres (see point 4 of the list below Algorithm 1 in Section 2.2) is replaced by a First In First Out (FIFO) policy. The sphere centers in the priority queue are still sorted by their objective values, but this DGLS variant breaks ties using the FIFO (arrival) order.

Graph, k	Algorithm	Start dist.	Succes rate	Final objective values			Restarts avg
				avg (std)	min	max	
1e450.25c, 25	DGLS-standard	150	5/10	1.1 (1.1)	0	2	9.8
flat300.28, 28		200	5/10	12.3 (15.3)	0	36	203.8
dsjc250.5, 28		140	4/10	0.7 (0.6)	0	2	49.3
1e450.25c, 25	DGLS with FIFO sphere ranking (second criterion)	150	5/10	1.3 (1.6)	0	5	6.4
flat300.28, 28		200	4/10	18.6 (15.3)	0	35	162
dsjc250.5, 28		140	4/10	1 (1)	0	3	51.9
1e450.25c, 25	DGLS that computes distances only every 200 iterations	140	6/10	0.9 (1.1)	0	3	6
flat300.28, 28		200	1/10	30.1 (10.4)	0	37	122.8
dsjc250.5, 28		140	4/10	0.8 (0.7)	0	2	36.1
1e450.25c, 25	Standard LS with a restart applied every 30000 iterations	150	3/10	1.3 (1.5)	0	5	7.7
flat300.28, 28		200	1/10	28.3 (9.6)	0	35	9
dsjc250.5, 28		140	2/10	1 (0.6)	0	2	8.3
1e450.25c, 25	Standard LS with a restart applied every 100000 iterations	150	3/10	1.7 (1.3)	0	4	2.1
flat300.28, 28		200	0/10	33 (2)	30	36	3
dsjc250.5, 28		140	1/10	1.3 (0.6)	0	2	2.7

Table 3: Comparison of 3 DGLS variants with 2 LS variants with restarts

- 2 A DGLS version that calculates the distance value only every 200 iterations. Recall that Algorithm 1 uses a function `need-calc-dist()` that is generally used to skip computing distances when exact distance values are not needed. For instance, if the current solution is at distance $0.2 \cdot \text{maxRadius}$ from the center, the next $0.8 \cdot \text{maxRadius}$ iterations can not lead to distances larger than `maxRadius`. But if the distance calculation is skipped for 200 iterations, an LS run *can* leave the sphere during these iterations. In such cases, the sphere examination is not really confined to a sphere of radius `maxRadius` as usually. However, this is not always so bad and it might not necessarily happen very often in practice.
- 3 A standard LS algorithm that applies a restart from the best-known solution every 30000 iterations.
- 4 A standard LS algorithm that applies a restart from the best-known solution every 100000 iterations.

Table 3 presents a comparison of the DGLS and LS variants presented above, using three rows for each variant. The columns of this table are exactly the same as those of Table 1, except for the fact that we replaced the last columns with the number of restarts. For DGLS, this number of restarts in the last column actually signifies the number of centers from which DGLS launched LS runs. By dividing the number of iterations by this number of restarts, one can form an opinion of the average number of iterations executed by an individual run inside a sphere.

Table 3 shows that the success rate of an LS method with restarts is only about half of the success rate of a DGLS variant, even if an LS with restarts can perform better than a pure LS without restarts (compare with the LS data from Table 1).

Comparing the three DGLS variants among them lead to more mixed conclusions. For example, the results of the DGLS version with a FIFO sphere

ranking criterion are very similar to those of the standard DGLS, which hints the second criterion for ranking spheres is not essential. The DGLS version that computes distances every 200 iterations generates slightly worse results. On the other hand, this DGLS variant computes less distances.

Recall that DGLS was deliberately designed to support a variety of (ways of combining the presented) intensification techniques, rather than a would-be “unique DGLS way”. For instance, preliminary experiments suggest that it could be useful to make DGLS even more aggressive as follows: allow DGLS to switch to a new center s immediately after finding a solution s of better quality than the current center c . One would need to modify Algorithm 1 to make it break the loop starting at Line 5 whenever it finds a solution s better than the current center c . As such, DGLS could (temporarily) abandon the goal of performing all `runsPerSphere` runs from c . However, after finishing exploring the sphere of s , DGLS could later come back to c (if c is at the beginning of the queue).

4.2 The k -clique and the k -cluster Problem

The goal of the k -cluster problem with unitary edge weights is to maximize the number of edges in an induced subgraph of size k . In fact, we will present results with regards to the minimization version of this problem, *i.e.*, minimize the number of non-edges (missing edges) in an induced subgraph of size k . We will actually only test the k -clique version of the problem, *i.e.*, we always choose values of k for which we know there exists at least one complete k -cluster (perfect clique) with k vertices.

We prefer to evaluate DGLS using a relatively basic canonical Tabu Search (TS) algorithm as the underlying LS. This TS encodes candidate solutions as 0/1 arrays with exactly k ones representing k selected vertices. At each iteration, it chooses the best vertex swap: remove a selected vertex v^{in} from the current solution and replace it with some non-selected vertex v^{out} . The best swap is the one that leads to the highest objective value improvement, breaking ties randomly in case of equality. The implemented TS uses incremental streamlined calculations to rapidly evaluate the objective value variation of each swap, see Appendix A.2.

After de-selecting v^{in} , this vertex becomes Tabu for $10 + \text{random}(5)$ moves. Despite this Tabu mechanism, our TS is more prone to stagnation than the LS for graph coloring from Section 4.1. It could sometimes loop for a long time on a plateau or on a quasi-plateau, *i.e.*, on a set of connected solutions with the same or very close⁴ objective values. Our TS uses the following technique to prevent such looping. After the first 1000 iterations, the TS counts the number $iters_{\text{plat}}$ of last consecutive iterations spent on a quasi-plateau. It then increases the above Tabu list length by $iters_{\text{plat}}$ for all moves that keep

⁴ We chose to consider two objective values obj_1 and obj_2 to be very close if and only if $|obj_1 - obj_2| \leq \Delta$, where Δ is the difference between the best and the third best objective value ever discovered by the current run.

the search on the current quasi-plateau, similarly to what we did using the term $\left\lfloor \frac{iters_{\text{plat}}}{1000} \right\rfloor$ in the Tabu list length for the graph coloring TS. The more iterations are spent on a quasi-plateau, the longer the Tabu status of many vertices typically selected by solutions of the quasi-plateau. This eventually imposes the selection of other non-Tabu vertices, leading the search to new areas.

To avoid slowing down DGLS with distance calculations, we also perform an incremental calculation of the distance from the current solution to the sphere center. This is relatively straightforward, because it is not difficult to update the distance (to the center) value after swapping vertices v^{in} and v^{out} —see exact calculation details in Appendix A.2.

The C++ source code of both LS and DGLS for the k -cluster problem are publicly available on-line at cedric.cnam.fr/~porumbed/dgls/. We can say it is a “human-size” code of about 1200 lines; the fact that the underlying LS is canonical TS with few fancy features may simplify reading the code.

4.2.1 General results on k -clique instances

We will compare DGLS with LS using a total number of iterations of `maxIter` = 1.000.000. The spheres are sorted according to the objective value of the center, breaking ties according to the FIFO order. We set the number of runs per sphere at `runsPerSphere` = 3 as in the graph coloring case. The radius value has been set to `maxRadius` = $1.5 \cdot k$, because we observed that `maxRadius` = k does not seem enough, *i.e.*, our TS can often reach a distance of k in only $\frac{1}{2}k$ iterations by simply changing $\frac{1}{2}k$ vertices. More generally, Appendix C presents a test that evaluates the success rate of DGLS for different values of `runsPerSphere` and `maxRadius`. Generally speaking, this test indicates that $[0.75 \cdot k, 1.75 \cdot k]$ is an interval of safe values for `maxRadius` and $[2, 5]$ is an interval of safe values for `runsPerSphere`.

We apply all three techniques for avoiding stagnation from Section 2.3 and they are instantiated as follows. In technique 1, the maximum number of iterations per sphere is set at $10 \cdot n$. Technique 2 is instantiated with no modification. To implement technique 3, we associate to each run a repulsion force f that can increase along the iterations depending on the distance from the visited solutions to the center. More exactly, for each visited solution s such that $obj(s) = obj(c)$ and $d(s, c) < \frac{1}{2} \text{maxRadius}$, we increase f by a $\Delta_{s,c}$ value inversely proportional⁵ to $d(s, c)$. This repulsion force f acts as follows: we increase the Tabu list length by f for all vertices v that contribute to the Hamming distance $d(s, c)$, *i.e.*, such that (i) $s[v] = 1$ and $c[v] = 0$ or (ii) $s[v] = 0$ and $c[v] = 1$. This progressively repulses the run from the center, because a high repulsion force encourages fixing vertices that contribute to

⁵ We uses $\Delta_{s,c} = \frac{1}{d(s,c)+3}$. For example, if the search revisits 30 times the center c , then we obtain a total repulsion force of $30 \cdot \frac{1}{3} = 10$. As such, the currently selected vertices that do *not* belong to the center stay Tabu 10 iterations more. This encourages DGLS to deselect vertices that do belong to the center, thus repulsing the search away from it.

the distance to c . If f is non-zero at the end of a run launched from c , we then insert in the priority queue the best solution `bstFar` that satisfies $d(\text{bstFar}, c) \geq \frac{9}{10} \text{maxRadius}$.

For many k -clique instances, the TS implemented in this section reports the same result over all executions. One can also observe this phenomenon for the faster TS from [18], where Table 1 announces a success rate of 100% for all but three graphs. Our TS has less fancy features and allows a larger variation of the final best objective values. However, we did need to restrict the study to several graphs on which our TS does report significantly different final results. We also introduce two new instances `keller4+1` and `keller4+2` obtained by modifying the `keller4` instance. The original `keller4` instance is not very difficult, because it contains numerous perfect cliques of size 11. We took one of these cliques of size 11 and removed some of the edges linking it to the rest of the graph, so as to isolate (hide) it; finally, we added an artificial vertex that is only linked to the chosen clique of size 11. The maximum clique in the resulting instance is thus 12, but it is more difficult to find it.⁶

Since DGLS is primarily designed to achieve intensification, it makes sense to evaluate it by launching DGLS from a solution moderately close to an optimal clique, as in Section 4.1.1. For this purpose, we took a perfect clique for each graph, we replaced a number of vertices with vertices outside the clique and we launched both DGLS and LS from the resulting perturbed solution.

Table 4 (previous page) presents this comparison of DGLS and LS, reporting the instance in Column 1 (the graph, n and k), the algorithm version in Column 2 (one row on DGLS, one on the underlying LS), the above number of vertices from a perfect clique replaced with other vertices (Column 3), the

⁶ These two instances are publicly available on-line, along with the LS/DGLS source code in C++ at <http://cedric.cnam.fr/~porumbel/dgls/>.

Graph, n, k	Algo- rithm	Disloca- ted vtx	Succes rate	Final objective values			Iterations to success			
				avg (std)	min	max	avg (std)	min	max	
<code>C1000.9</code> , 1000, 68	DGLS	40	10/10	0 (0)	0	0	5061 (13185)	157	44605	
<code>C1000.9</code> , 1000, 68	LS	40	9/10	0.1 (0.3)	0	1	294406 (191687)	166278	800226	
<code>C500.9</code> , 500, 57	DGLS	40	10/10	0 (0)	0	0	7394 (11984)	379	36801	
<code>C500.9</code> , 500, 57	LS	40	10/10	0 (0)	0	0	131209 (120215)	346	437266	
<code>MANN.a27</code> , 378, 126	DGLS	21	10/10	0 (0)	0	0	26756 (37578)	14	133534	
<code>MANN.a27</code> , 378, 126	LS	21	5/10	0.7 (0.8)	0	2	280488 (299632)	14	800954	
<code>c-fat500-2</code> , 500, 26	DGLS	14	10/10	0 (0)	0	0	29084 (20134)	15	62210	
<code>c-fat500-2</code> , 500, 26	LS	14	4/10	7.2 (5.9)	0	12	200016 (244950)	15	600018	
<code>c-fat500-5</code> , 500, 64	DGLS	34	10/10	0 (0)	0	0	18864 (2651)	16359	23982	
<code>c-fat500-5</code> , 500, 64	LS	34	0/10	31 (0)	31	31	- (-)	-	-	
<code>keller4⁺¹</code> , 172, 12	DGLS	5	8/10	0.2 (0.4)	0	1	231362 (167365)	6	415401	
<code>keller4⁺¹</code> , 172, 12	LS	5	1/10	0.9 (0.3)	0	1	6 (0)	6	6	
<code>keller4⁺²</code> , 172, 12	DGLS	6	5/10	0.5 (0.5)	0	1	29525 (58946)	30	147417	
<code>keller4⁺²</code> , 172, 12	LS	6	2/10	0.8 (0.4)	0	1	101532 (27384)	74148	128917	

Table 4: Comparison of DGLS and standard LS launched from a solution obtained by dislocating a number (“Dislocated vtx” in Column 3) of vertices from a perfect clique.

number of successful executions (finding a perfect clique) out of 10 (Column 4), followed by statistical results on the final objective values reported by the 10 executions (Columns “Final objective values”) and statistical results on the number of iterations needed by the successful executions (last 4 columns). The statistical results include: the average value (columns “avg”), the standard deviation (columns “std”), the minimum value (columns “min”) and the maximum value (columns “max”).

Table 4 shows that DGLS can indeed achieve stronger intensification. Except for the first two graphs, it finds the path towards an optimal solution twice or three times more often than the standard LS. Even for the first two graphs, it needs far less iterations than the underlying LS to reach the optimum.

4.2.2 Insights into the sphere examinations

All questions regarding the graph coloring DGLS from Section 4.1.2 are equally relevant for the k -clique problem. We thus use the same Multidimensional Scaling procedure from Section 4.1.2 to provide an intuitive visualization of the DGLS trajectory, so as to (try to) offer an answer to such questions.

Figures 3 and 4 confirm that DGLS follows a tree-like trajectory as expected from theory. In Figure 3, one notices many arrows (runs) that point towards the optimal solution, without directly reaching it. However, it is clear that DGLS can find an optimal solution virtually with probability 100%, by taking as starting center any of the end points of these arrows. On the other

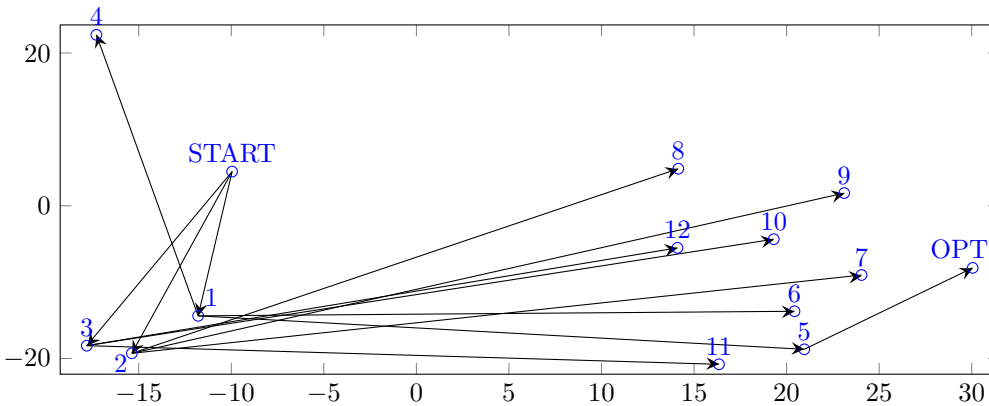


Fig. 3: MDS representation of the running profile of a short successful DGLS execution on `c-fat500-2` with `maxRadius = 40`. The points represent sphere centers and the associate labels indicate the order of the discovery of these centers; each arrow points to the best solution (future center) reported by a run launched from a center. The starting point `START` was generated by dislocating 14 selected vertices from a perfect clique, *i.e.*, `START` is at distance 28 from an optimal solution. However, the optimal solution `OPT` discovered by DGLS is at distance 40 from `START`. The three solutions discovered from `START` are at distance 28, *i.e.*, DGLS “repaired” the 14 dislocated vertices at each run from `START`.

hand, a standard LS could also follow a path towards a point like 4 and thus miss the region at the right of the figure with optimal solutions.

Figure 4 shows a more challenging DGLS execution. One can notice that many arrows do not point at all towards the optimal solution, and so, certain runs could easily lead DGLS away from interesting areas. These branches were fortunately cut by DGLS and its strong intensification mechanism managed to lead the main search process to a region that does contain an optimal solution.

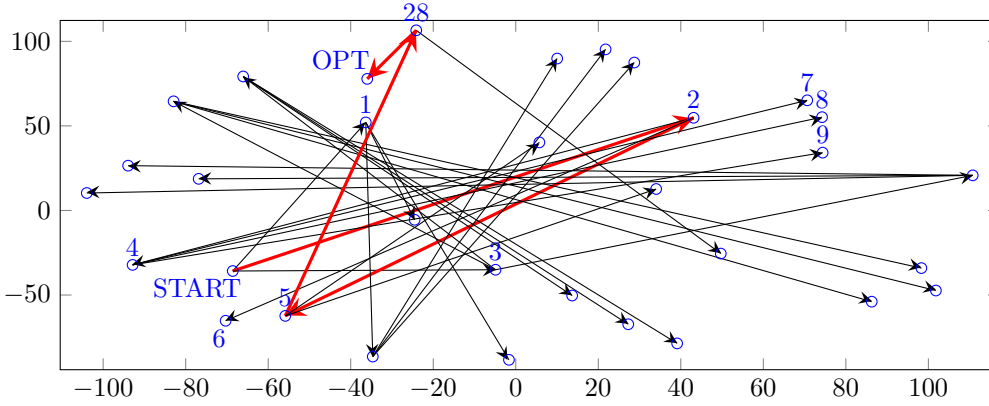


Fig. 4: Running profile of a more challenging successful DGLS execution on MANN_a27 with $\text{maxRadius} = 189$. Each point represents a sphere center. The path from the starting point **START** to the optimum solution **OPT** is depicted in red; **OPT** is at distance 114 from **START**. DGLS started out by visiting a quite far point 2, at distance 148 from **START**. It then came back closer to **START** at point 5, before eventually finding a way towards **OPT**.

4.2.3 Comparing to other random restarts or sphere ranking criteria

As in Section 4.1.3 on graph coloring, we now investigate other DGLS and LS variants. This will also be very useful for evaluating the contribution of the different techniques incorporated into DGLS and LS. More exactly, we will compare the standard DGLS with the following four algorithms:

1. DGLS with particularly small spheres, using $\text{maxRadius} = 0.25 \cdot k$ instead of $\text{maxRadius} = 1.5 \cdot k$.
2. DGLS with a maximum number of iterations per sphere of $1000 \cdot n$ instead of $10 \cdot n$.
3. DGLS with none of the stagnation avoidance techniques from Section 2.3.
4. LS with 10 restarts during the $\text{maxIter} = 1.000.000$ iterations.

Table 5 compares these algorithms. The second block or rows (rows 6-8) suggest that using a very small radius $\text{maxRadius} = 0.25 \cdot k$ leads to weaker DGLS flavor. Such DGLS can end up generating (a web of) thousand of small spheres (see the last column) associated to small-length runs that do not have enough intensification strength. The third block of rows (rows 9-11) shows

Graph, n, k	Algorithm	Start dist.	Success rate	Final objective values			Average restarts
				avg (std)	min	max	
C500.9, 500, 57	DGLS-standard	40	10/10	0 (0)	0	0	7
MANN_a27, 378, 126		21	10/10	0 (0)	0	0	10.3
c-fat500-2, 500, 26		14	10/10	0 (0)	0	0	3.9
C500.9, 500, 57	DGLS with a small $maxRadius = 0.25k$ instead of $1.5k$	40	4/10	1.3 (1.2)	0	3	24265.2
MANN_a27, 378, 126		21	9/10	0.1 (0.3)	0	1	4068.5
c-fat500-2, 500, 26		14	5/10	6 (6)	0	12	595.7
C500.9, 500, 57	DGLS with max $1000 \cdot n$ (100x more) iterations per sphere	40	10/10	0 (0)	0	0	23.5
MANN_a27, 378, 126		21	10/10	0 (0)	0	0	12.6
c-fat500-2, 500, 26		14	2/10	9.6 (4.8)	0	12	1.8
C500.9, 500, 57	DGLS with no stagnation avoidance	40	7/10	0.6 (0.9)	0	2	871.5
MANN_a27, 378, 126		21	10/10	0 (0)	0	0	254.2
c-fat500-2, 500, 26		14	3/10	8.4 (5.5)	0	12	4
C500.9, 500, 57	Standard LS with a restart every 100000 iterations (max 10)	40	10/10	0 (0)	0	0	2.2
MANN_a27, 378, 126		21	6/10	0.6 (0.8)	0	2	7.7
c-fat500-2, 500, 26		14	5/10	3.6 (5.5)	0	12	6.3

Table 5: Comparison of different DGLS and LS variants

that imposing a maximum number of iterations per sphere is not always necessary. Using a very large value for this parameter, DGLS could still solve two instances with a 100% success rate, but fail 8 times on `c-fat500-2`.

The impact of the stagnation avoidance techniques from Section 2.3 can be evaluated using the fourth block of rows (rows 12-14) of Table 5. We notice that by removing these techniques, the success rate is reduced for two graphs. Even if the success rate for `MANN_a27` remains the same, the number of runs launched from sphere centers is much larger, which suggests that this DGLS variant needed more effort to find the optimum. The reason for the failures of this DGLS variant on `c-fat500-2` comes from the fact that the search process is actually blocked looping on a plateau around a local optimum (inside a sphere). Indeed, notice that this DGLS launched in average *only* 4 runs (see last column) from sphere centers during all 1.000.000 iterations.

Finally, the last three rows concern an LS variant that executes 10 random restarts during the `maxIter = 1.000.000` iterations. This LS variant does not reach results that can change our main conclusions. For example, it fails almost half of the time on `MANN_a27`, while this instance is solved with a 100% success rate even by the simplest DGLS variants.

4.3 The Capacitated Arc Routing Problem (CARP)

In this section, the underlying LS is a simplified version of the Iterated Local Search (ILS) from [13]. We recall that this LS works with permutations of the set E_R of edges requiring service; any permutation is decoded into explicit routes by applying a decoder based on dynamic programming. The main simplifications compared to [13] come from the fact that we use no Column Generation and no local search on explicit (decoded) routes. Additionally, our

CARP instance Graph, best	LS			DGLS		
	bst	avg	worst	bst	avg	worst
eg1-S1-A, 5018	5154	5249.5	5336	5050	5180.8	5276
eg1-S1-B, 6388	6454	6584	6658	6473	6599.7	6658
eg1-S1-C, 8518	8725	8778.6	8852	8616	8710	8917
eg1-S2-A, 9884	11057	11166.8	11379	10993	11148.9	11379
eg1-S2-B, 13100	16251	16677.1	16861	16140	16602.9	16895
eg1-S2-C, 16425	18998	19582.1	19868	19309	19568.9	19807
eg1-S3-A, 10220	11236	11334.1	11391	11236	11289.9	11342
eg1-S3-B, 13682	16251	16677.1	16861	15468	16007.1	16251
eg1-S3-C, 17188	19392	19581.3	19650	19306	19460	19627

Table 6: Results of LS and DGLS on CARP considering a time limit of 300 seconds. For each row, we execute 10 times LS and DGLS.

neighborhood only consists of adjacent swaps on permutations. More details on the algorithm are provided in Appendix A.3 or directly in [13].

Since the evaluation of each permutation requires a decoder that is relatively computationally intensive,⁷ there is no important slowdown induced by a straightforward distance calculation. Recalling the distance definition (3.1) from Section 3.2.3, we notice this distance calculation requires $\frac{n(n-1)}{2}$ comparisons. Finally, the sphere radius is set at $\text{maxRadius} = 5 \cdot n$ and the number of runs per sphere is $\text{runsPerSphere} = 3$ as for k -coloring and k -cluster.

Table 6 compares LS and DGLS on several CARP instances on which the difference between the results of LS and DGLS are relatively large. For both methods, we allow 300 seconds per execution. Columns 3 and 6 show that DGLS obtains a better minimum objective value with only one exception (eg1-S1-B). Columns 4 and 7 show that DGLS obtains a lower average objective value in all instances but one (eg1-S1-B).

Finally, all results from this section were obtained on an Intel Xeon CPU (E5-2630) clocked at 2.4GHz. The k -cluster and k -coloring algorithms were implemented in C++ and compiled by `gnu g++` with `-O3` optimization option. The CARP algorithm was implemented in Java, version 1.7. Notice there exists a benchmark for comparing coloring algorithms on different instances,⁸ useful for providing a hardware-independent measure of CPU speed. This benchmark leads the following user times on our machine: 5.05s for `r500.5.b`, 1.33 for `r400.5.b`, 0.28 for `r300.5.b`, and 0.05 for `r200.5.b`.

⁷ For the k -coloring and k -clique problems, the evaluation of each neighbor requires $O(1)$ time, *i.e.*, strong streamlining routines are used. In CARP, the evaluation of each neighbor is linear in the number $|E_R|$ edges (clients), in the number of vehicles and in the size of the longest route.

⁸ See <http://mat.gsia.cmu.edu/COLOR03/> or more exactly the benchmark in the tar archive available for download at mat.gsia.cmu.edu/COLOR03/BENCHMARK/benchmark.tar.

5 Conclusions and Prospects

Distance measures have been used relatively rarely in local search algorithms and usually for diversification reasons rather than for intensification. In this work, we have demonstrated that distances can be used to increase the intensification potential of a given LS. The proposed distance-guided local search framework operates on top of an underlying local search to equip it with a number of intensification mechanisms discussed throughout the paper. An important change is that the trajectory of the resulting DGLS algorithm is no longer a continuous path of visited solutions, but a tree-like structure composed of examined spheres and non-examined spheres. Experiments on three representative problems (k -coloring, k -clique and Capacitated Arc-Routing) show that DGLS can improve the underlying LS.

The proposed DGLS is not an exact recipe which must be closely followed in any attempt to improve an existing LS, but rather a synthesis of convergent ideas related to distances. One could use only some of the distance-based techniques presented throughout the paper, *i.e.*, the ones that prove to be the most effective for the considered problem. For instance, it might not always be necessary to record the spheres in a priority queue. Without resorting to Algorithm 1, one could only use the stagnation avoidance techniques from Section 2.3 which can enable the given LS to detect when it is stuck looping on a plateau around a center, so as to change its trajectory accordingly.

Finally, the distance calculation overhead could always be kept within reasonable limits, using a different idea for each of the three considered problems. For graph coloring, the distance has to be computed only once in tens of iterations, using arguments from point 6 of Section 2.2. For the clique problem, the distance to the center can be incrementally calculated in constant time at each iteration, see Appendix A.2. For the CARP, the objective function evaluation requires running a permutation decoder based on dynamic programming and this is a more important computational bottleneck than the distance calculation.

Acknowledgments

We are grateful to the reviewers for their valuable comments which helped us to improve the paper. The idea of using MDS to represent intuitively the trajectory of an LS was originally proposed by Pascale Kuntz during the PhD thesis of the first author and this contribution is acknowledged.

References

1. Vicente Campos, Manuel Laguna, and Rafael Martí. Context-independent scatter and tabu search for permutation problems. *INFORMS Journal on Computing*, 17(1):111–122, 2005.
2. Walter Cedeño and Rao V. Vemuri. Analysis of speciation and niching in the multi-niche crowding GA. *Theoretical Computer Science*, 229(1):177, 1999.

3. Kenneth A. De Jong. *An analysis of the behavior of a class of genetic adaptive systems*. PhD thesis, University of Michigan Ann Arbor, MI, USA, 1975.
4. Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.
5. Zvi Drezner. A new heuristic for the quadratic assignment problem. *Advances in Decision Sciences*, 6(3):143–153, 2002.
6. Moshe Dror. *Arc routing: theory, solutions and applications*. Springer Science & Business Media, 2012.
7. Fred Glover, Manuel Laguna, and Rafael Martí. Fundamentals of scatter search and path relinking. *Control and Cybernetics*, 29(3):653–684, 2000.
8. Dan Gusfield. Partition-distance: A problem and class of perfect graphs arising in clustering. *Information Processing Letters*, 82(3):159–164, 2002.
9. Cédric Joncour, Sophie Michel, Ruslan Sadykov, Dmitry Sverdlov, and François Vanderbeck. Column generation based primal heuristics. *Electronic Notes in Discrete Mathematics*, 36:695–702, 2010.
10. MG Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.
11. Brand L. Miller and Michael J. Shaw. Genetic algorithms with dynamic niche sharing for multimodal function optimization. In *Proceedings of the IEEE International Conference on Evolutionary Computation*, pages 786–791, 1996.
12. Alberto Moraglio and Riccardo Poli. Topological interpretation of crossover. In Kalyanmoy Deb, editor, *Genetic and Evolutionary Computation Conference*, pages 1377–1388. Springer, 2004.
13. Daniel Porumbel, Gilles Goncalves, Hamid Allaoui, and Tienté Hsu. Iterated local search and column generation to solve arc-routing as a permutation set-covering problem. *European Journal of Operational Research*, 256(2):349–367, 2017.
14. Daniel Cosmin Porumbel, Jin-Kao Hao, and Pascale Kuntz. A search space “cartography” for guiding graph coloring heuristics. *Computers and Operations Research*, 37(4):769–778, 2010.
15. Daniel Cosmin Porumbel, Jin-Kao Hao, and Pascale Kuntz. An efficient algorithm for computing the distance between close partitions. *Discrete Applied Mathematics*, 159(1):53–59, 2011.
16. Daniel Cosmin Porumbel, Jin-Kao Hao, and Pascale Kuntz. Spacing memetic algorithms. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation (GA track)*, pages 1061–1068. ACM, 2011.
17. Robert E. Smith, Stephanie Forrest, and Alan S. Perelson. Searching for diverse, cooperative populations with genetic algorithms. *Evolutionary Computation*, 1(2):127–149, 1993.
18. Qinghua Wu and Jin-Kao Hao. An adaptive multistart tabu search approach to solve the maximum clique problem. *Journal of Combinatorial Optimization*, 26(1):86–108, 2013.

A The underlying local searches and their streamlined calculations

A.1 Graph k -Coloring

The underlying LS for graph k -coloring is the Tabu Search (TS) from [14]. A solution s is represented as an array of length n such that s_v is the color of vertex v . A neighboring solution can be obtained by simply changing the color s_v of any conflicting vertex v to some s'_v . By focusing on conflicting vertices, this neighborhood helps the search process to concentrate on influential moves and to avoid irrelevant ones, because changing the color of a non-conflicting vertex would not directly improve the objective function.

After executing a move and assigning a new color to a vertex v , v can not receive again the lost color for the next T_ℓ iterations. The value of T_ℓ is set at $\text{random}(10) + 0.6 \cdot \text{obj}(s) + \left\lfloor \frac{\text{iters}_{\text{plat}}}{1000} \right\rfloor$, where $\text{iters}_{\text{plat}}$ is the number of last moves with no objective function variation.

The last term is only introduced to change T_ℓ when the algorithm is blocked looping on a plateau and the objective value does not change for 1000 moves. Each series of consecutive 1000 moves with no objective function variation triggers the increment of all subsequent tabu list lengths, which encourages TS to choose more and more moves that have not been executed in the past, until the objective changes again and TS leaves the plateau. This additional term prevents the search process from getting blocked looping on a plateau while not affecting its behavior outside plateaus.

To rapidly choose the best neighbor of s , this TS uses a $n \times k$ table Γ such that Γ_{v,s'_v} indicates the number of conflicts of v if v received color s'_v . As such, $\Gamma_{v,s'_v} - \Gamma_{v,s_v}$ represents the objective function variation associated to the move that changes the color of v from s_v into s'_v . After performing a move, Γ can be updated in $O(n)$ time (because only columns s_v and s'_v might require updating).

A.2 k -cluster: incremental calculations of objective value and distance

The main ideas of the k -clique Tabu Search (TS) algorithm were presented in the first paragraphs of Section 4.2. We here describe how it uses incremental calculations to rapidly find the best swap of vertices at each iteration. For this, the TS uses a table that associates to each non-selected vertex v^{out} the number of edges that it can bring to the current solution. For a selected vertex v^{in} , this table records the number of edges linked to v^{in} in the current solution. To find the best swap, it is enough to consider each selected vertex v^{in} and each non-selected one v^{out} and to calculate (in constant time using the above table!) the objective function variation of swapping v^{in} with v^{out} . After executing the move, the table values of v^{in} and v^{out} are quite easily updated, by scanning their neighbors modified by the last move. For a more complex and faster calculation streamlining scheme, we refer the reader to [18]. However, using a slower (and more pedagogical) algorithm poses no problem for the empirical evaluations needed in this paper.

The calculation of the distance from the current solution s to the current center c is also incremental. If s' is obtained from s by swapping a and b , then

$$d(s', c) = d(s, c) - \underbrace{\left([s_a \neq c_a] + [s_b \neq c_b] \right)}_{\text{old contribution to the Hamming distance}} + \underbrace{\left([s_b \neq c_a] + [s_a \neq c_b] \right)}_{\text{new contribution to the Hamming distance}},$$

where $[S]$ is the Iverson bracket, *i.e.*, $[S]$ is 1 when the statement S is true and 0 otherwise. If the move consists of deselecting a selected vertex $a = v^{\text{in}}$ and of selecting a non-selected vertex $b = v^{\text{out}}$, the above formula becomes

$$d(s', c) = d(s, c) - \left([1 \neq c_a] + [0 \neq c_b] \right) + \left([0 \neq c_a] + [1 \neq c_b] \right).$$

One can check all possible cases of c_a and c_b to see this leads to the following simpler formula:

$$d(s', c) = d(s, c) + 2 \cdot c_a - 2 \cdot c_b.$$

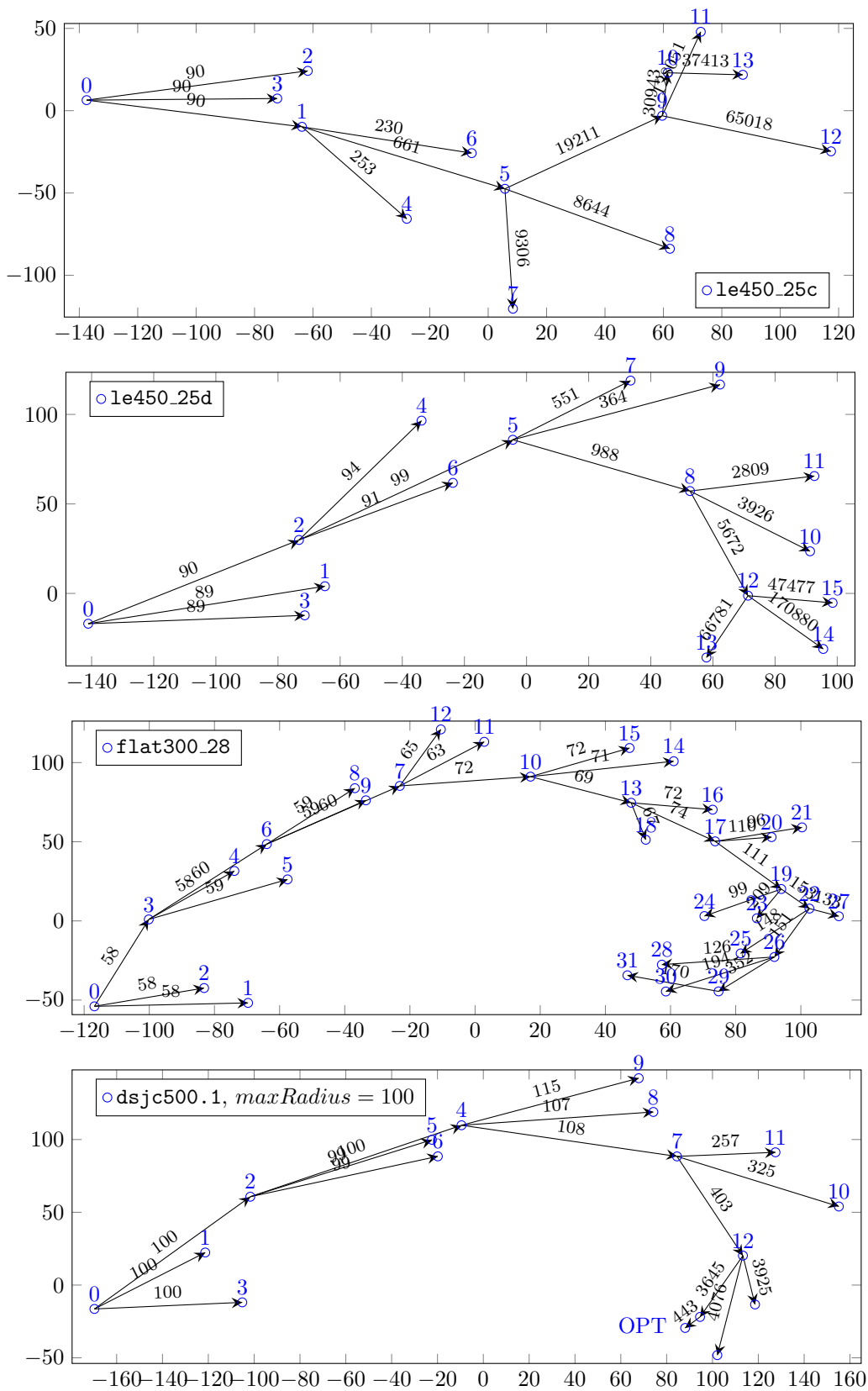
A.3 Capacitated Arc-Routing (CARP)

The underlying LS for CARP is based on a simplification of the Iterated Local Search (ILS) from [13]. The original ILS considers a search space of permutations that are decoded into explicit routes using a decoder (see below). The main simplifications are the following. First, all Column Generation (CG) components of the algorithm from [13] are removed, allowing one to more easily compare LS with DGLS, using less external components. Secondly, the

neighborhood is restricted to only use adjacent transpositions (swaps), *i.e.*, a neighbor permutation is constructed by swapping consecutive elements of the current permutation. This allows one to achieve a correlation between a distance $d(s_a, s_b)$ and the number of LS moves needed to reach s_a from s_b . We do not use any post-decoder as in [13].

The perturbation operator of this ILS consists of inserting in the current solution a route (sequence) discovered earlier during the search [13, §2.1]. More exactly, to perturb the current permutation s , we extract a route r from a pool, we inject r at the beginning of s and we remove from s any duplicate element of r . The pool is continually updated throughout the search, by adding routes discovered by the ILS at different moments of the search.

Finally, the decoder consists of a dynamic programming routine of linear complexity in terms of the number of clients $|E_R|$, *i.e.*, the complexity is $O(|E_R|)$. More precisely, given input permutation $s = (s_1, s_2, \dots, s_m)$, the decoder determines a set of routes of minimum total cost that service all required edges in the order s_1, s_2, \dots, s_m . Since the decoder is relatively computationally intensive, the distance calculations do not introduce an important slowdown in the search.

B MDS plots of other DGLS trajectories for graph k -coloring

C The success rate of DGLS for different values of runsPerSphere and maxRadius

Here, we analyze the effectiveness of DGLS over several values of `runsPerSphere` and `maxRadius` on the graph coloring and the k -clique problem (in Table 7 and resp. Table 8). The caption of Tables 7-8 is self explanatory.

Graph, k	Start dist.	maxRadius										
		$0.05 \cdot n$	$0.1 \cdot n$	$0.2 \cdot n$	$0.3 \cdot n$	$0.4 \cdot n$	$0.5 \cdot n$	$0.6 \cdot n$	$0.7 \cdot n$	$0.8 \cdot n$	$0.9 \cdot n$	$1 \cdot n$
1e450_25c, 25	150	4/10	4/10	5/10	5/10	8/10	5/10	4/10	0/10	0/10	0/10	0/10
1e450_25d, 25	190	3/10	3/10	5/10	2/10	1/10	4/10	2/10	3/10	3/10	2/10	2/10
flat300_28, 28	200	3/10	4/10	5/10	3/10	1/10	2/10	4/10	6/10	3/10	0/10	0/10
dsjc250_5, 28	140	3/10	4/10	4/10	5/10	4/10	4/10	6/10	4/10	4/10	2/10	2/10
dsjc500_1, 12	300	7/10	4/10	6/10	2/10	3/10	2/10	7/10	6/10	4/10	2/10	2/10
dsjc500_5, 48	230	1/10	7/10	5/10	5/10	3/10	4/10	6/10	3/10	2/10	0/10	0/10
dsjc500_9, 126	150	9/10	10/10	10/10	8/10	9/10	10/10	8/10	6/10	6/10	6/10	6/10
dsjc1000_1, 21	800	2/10	2/10	4/10	0/10	2/10	3/10	5/10	3/10	4/10	2/10	4/10
dsjc1000_5, 85	450	2/10	2/10	4/10	1/10	0/10	1/10	2/10	0/10	1/10	0/10	0/10
dsjc1000_9, 223	250	10/10	10/10	10/10	10/10	9/10	8/10	8/10	8/10	8/10	8/10	8/10

Graph, k	Start dist.	runsPerSphere						
		1	2	3	4	5	6	7
1e450_25c, 25	150	3/10	0/10	5/10	6/10	2/10	1/10	6/10
1e450_25d, 25	190	1/10	0/10	5/10	3/10	2/10	2/10	4/10
flat300_28, 28	200	1/10	3/10	5/10	3/10	4/10	3/10	4/10
dsjc250_5, 28	140	5/10	5/10	4/10	3/10	5/10	2/10	4/10
dsjc500_1, 12	300	4/10	6/10	6/10	2/10	2/10	2/10	5/10
dsjc500_5, 48	230	2/10	4/10	5/10	6/10	4/10	5/10	2/10
dsjc500_9, 126	150	9/10	8/10	10/10	9/10	9/10	9/10	9/10
dsjc1000_1, 21	800	0/10	4/10	4/10	2/10	1/10	0/10	3/10
dsjc1000_5, 85	450	0/10	0/10	4/10	2/10	1/10	1/10	0/10
dsjc1000_9, 223	250	8/10	7/10	10/10	8/10	7/10	7/10	10/10

Table 7: The success rate of DGLS for different value of `maxRadius` (above table) or `runsPerSphere` (below table) on graph coloring. See also Table 1 (p. 11) for the main results with `maxRadius` = $0.2 \cdot n$ and `runsPerSphere` = 3 that correspond to the bold column in both tables above. The first three columns indicate the instance and they are the same as in Table 1. Recall DGLS is launched from a solution situated at a given distance (Column “Start dist.”) from an optimal solution.

Instance, n, k	Dislocated vertices	maxRadius							
		$0.25 \cdot k$	$0.5 \cdot k$	$0.75 \cdot k$	$1 \cdot k$	$1.25 \cdot k$	$1.5 \cdot k$	$1.75 \cdot k$	$2 \cdot k$
C1000.9, 1000, 68	40	6/10	9/10	8/10	10/10	10/10	10/10	10/10	10/10
C500.9, 500, 57	40	7/10	9/10	7/10	7/10	10/10	10/10	10/10	6/10
MANN_a27, 378, 126	21	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10
c-fat500-2, 500, 26	14	0/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10
c-fat500-5, 500, 64	34	0/10	0/10	10/10	10/10	10/10	10/10	10/10	10/10
keller4 ⁺¹ , 172, 12	5	1/10	0/10	10/10	9/10	9/10	8/10	7/10	0/10
keller4 ⁺² , 172, 12	6	2/10	4/10	2/10	1/10	10/10	5/10	10/10	7/10

Instance, n, k	Dislocated vertices	runsPerSphere						
		1	2	3	4	5	6	7
C1000.9, 1000, 68	40	10/10	8/10	10/10	8/10	10/10	10/10	10/10
C500.9, 500, 57	40	10/10	9/10	10/10	10/10	7/10	0/10	0/10
MANN_a27, 378, 126	21	2/10	10/10	10/10	10/10	10/10	10/10	10/10
c-fat500-2, 500, 26	14	10/10	10/10	10/10	10/10	10/10	10/10	10/10
c-fat500-5, 500, 64	34	10/10	10/10	10/10	10/10	10/10	10/10	10/10
keller4 ⁺¹ , 172, 12	5	5/10	7/10	8/10	9/10	10/10	10/10	10/10
keller4 ⁺² , 172, 12	6	6/10	2/10	5/10	3/10	2/10	2/10	2/10

Table 8: The success rate of DGLS for different value of `maxRadius` (above table) or `runsPerSphere` (below table) on the k -clique problem. See also Table 4 (p. 18) for the main results with `maxRadius` = $1.5 \cdot k$ and `runsPerSphere` = 3 that correspond to the bold column in both tables above. Recall DGLS is launched from a solution obtained by dislocating a number (Column “Dislocated vertices?”) of vertices from a perfect clique.