On the relation between complete and incomplete search: an informal discussion

Michela Milano
DEIS - Università di Bologna
Viale Risorgimento, 2
40136 Bologna - Italia
email: `mmilano@deis.unibo.it`

Andrea Roli
DEIS - Università di Bologna
Viale Risorgimento, 2
40136 Bologna - Italia
email: `aroli@deis.unibo.it`

Abstract

In this paper, we propose a variant of the LDS strategy, called Climbing Discrepancy Search (CDS) by underlying similarities and differences with a well known metaheuristic strategy called Variable Neighborhood Descent (VND). LDS in this setting can be seen as a bridge between tree and local search. We then argue that this process of comparing LDS with metaheuristics can lead to many different variants of LDS that can be used to exhaustively explore different neighborhoods and different regions of the search landscape. A discussion on open perspectives is presented.

1 Introduction

Many combinatorial optimization problems can be solved using tree search methods. Among them, Limited Discrepancy Search (LDS), proposed by Harvey and Ginsberg [23], has been effectively used. LDS starts from the solution suggested by the search heuristic (the *reference* solution) and successively explores solutions with increasing discrepancies, i.e., distances, from it.

Often, problem hardness and size make it impractical to exhaustively explore the search space and thus metaheuristic algorithms are applied. Metaheuristics [21, 4, 20, 1, 3] are approximate algorithms, as they do not guarantee to find the optimal solution in bounded time. However, they effectively and efficiently exploit search space characteristics to guide the search toward promising search space regions.

The motivations of this work originate from the intent to define a general framework to investigate similarities and differences among complete and incomplete search. This enables us to combine the strengths of both kinds of techniques and to introduce ideas from one field to the other and vice versa. Moreover, recent work on finite convergence of metaheuristc algorithms [9] can be easily included in our general discussion and finite convergence proofs may be provided for many local search algorithms.

In this paper, we introduce a variant of Limited Discrepancy Search, called Climbing Discrepancy Search (CDS) useful in optimization problems. In optimization problems, as soon as a solution with a better value of the objective function is found, the reference solution can be updated so as to visit nodes at increasing discrepancies from the new reference solution. CDS embodies this idea by performing an LDS with dynamic update of the reference solution.

CDS is directly comparable with a well known metaheuristic strategy called Variable Neighborhood Descent (VND, [10, 11]). Roughly speaking, they explore solutions in the same order. Thus, this comparison can be seen as a starting point toward a definition of a bridge between complete and incomplete search.

We have considered other metaheuristics and applied their basic principles to define other variants of LDS. In particular, for each variant, we have to define the discrepancy and how to explore nodes at a given discrepancy. In this paper, beside VND, we revisit First and Best Improvement, Simulated Annealing and Tabu Search in a discrepancy-based fashion.

It is important to stress the fact that by using a tree search to explore leaves in the same order as done by metaheuristics, we have many advantages: first, in a sense, we can *remember* which leaves have been explored; second, we can either stop the LDS variant as soon as a stopping condition is met, as done in metaheuristic algorithms, or we can exhaustively explore the search tree; third, exploring a search tree, we can exploit constraint propagation to prune infeasible states, and lower and upper bound calculation to prune provably sub-optimal nodes.

This paper is organized as follows: we first present CDS, then we discuss its similarities with VND and we generalize this comparison to other metaheuristics. Finally, we mention related work and discuss open directions.

2 Climbing Discrepancy Search

Let us consider the leaves of the search tree explored by the Limited Discrepancy Search (LDS). The first leaf explored is the one suggested by the heuristic. Then, nodes at discrepancy equal to one are explored, then those at discrepancy equal to two and so on. The discrepancy can be seen as a sort of *distance* from a *reference* solution.

In local search and metaheuristic techniques, starting from a node (the *reference* solution), a neighborhood is explored to visit a portion of the search space "near" the initial solution. The idea behind metaheuristics is that if the reference solution has a good objective function value, solutions similar to it are supposed to be good as well. The reference solution is changed as soon as a solution with better value is found, and the process of exploring the neighborhood starts again.

As argued in [23], LDS can be seen as a way of exploring the neighborhood of a given solution. In the original LDS definition, the reference solution is never changed. In addition, the neighborhood is successively enlarged since nodes at increasing discrepancy are visited.

In optimization problems, we can change the reference solution as soon as a leaf with an improving objective function is found. We suppose to deal with a problem defined

Algorithm 1 Climbing Discrepancy Search

$k \leftarrow 1$ {k is the discrepancy}
$k_{max} \leftarrow n$
$s^* \leftarrow$ InitialSolution() {s^* is the reference solution proposed by the heuristic}
while $k \leq k_{max}$ **do**
  Generate all leaves at discrepancy k w.r.t. s^*: $Leaves(s^*, k)$
  $s' \leftarrow$ BestOf($Leaves(s^*, k)$)
  if $f(s') < f(s^*)$ **then**
    $s^* \leftarrow s'$ {the new solution substitutes the reference one}
    $k \leftarrow 1$
  end if
  $k \leftarrow k + 1$
end while

over binary variables $x_1, \ldots, x_n$ and a solution s corresponds to a feasible assignment to all variables. In this setting, we can describe the modified LDS as shown in Algorithm 1.

Note that we can implement an incomplete version of CDS by setting a $k_{max} < n$. The process of changing neighborhoods in case of no improvements is exploited by a local search called Variable Neighborhood Descent (VND). In section 3.1 we will compare the two strategies.

A straightforward variant of CDS can be obtained by picking each leaf in $Leaves(s^*, k)$, compare it with the reference solution and, if it is better, immediately changing the reference solution (instead of exhaustively explore $Leaves(s^*, k)$).

3 Local Search and Tree Search: Convergences

In the previous section we have described an LDS variant which combines tree search with some characteristics of local search. In this section we push this combination further, showing how CDS can be seen as a bridge between tree search and local search. This process will be split into two steps: first we will show the strong similarities between CDS and a local search method called Variable Neighborhood Descent, then we will extend and generalize the comparison.

3.1 CDS and Variable Neighborhood Descent

CDS can be seen a kind of reformulation of the local search algorithm Variable Neighborhood Descent [10, 11] in terms of tree search. Local Search (LS) algorithms require the definition of *states*, which are feasible configurations of variables, a *neighborhood function*, which assigns to each state s a set of feasible states $\mathcal{N}(s)$ and the objective function to be minimized[1]. The neighborhood function $\mathcal{N}$ implicitly defines an *operator* φ which takes a state s_1 and transforms it into another state $s_2 \in \mathcal{N}(s_1)$. Dually, given an operator φ, it is possible to define a neighborhood of a variable $s_1 \in S$: $\mathcal{N}_\varphi(s_1) = \{s_2 \in S \setminus \{s_1\} \mid s_2$ can be obtained by one application of φ on $s_1\}$. VND belongs, with Variable Neighborhood Search [10, 11], to the family of LS algorithms with dynamic neighborhood structures. These algorithms explicitly deal with dynamically changing neighborhood

[1]Equivalently, the problem can be defined over an objective function to be maximized.

Algorithm 2 Variable Neighborhood Descent

```
Select a set of neighborhood structures 𝒩_k, k = 1,...,k_max
s* ← InitialSolution()
while termination conditions not met do
  k ← 1
  while k ≤ k_max do
    s' ← BestOf(𝒩_k(s*))
    if f(s') < f(s*) then
      s* ← s'
      k ← 1
    else
      k ← k + 1
    end if
  end while
end while
```

structures $\mathcal{N}_1, \ldots, \mathcal{N}_{k_{max}}$. The search process switches from one neighborhood $\mathcal{N}_k$ to the following one $\mathcal{N}_{k+1}$ whenever the search needs to be moved to other regions of the search space (diversification). In particular, the choice of neighborhoods of increasing cardinality yields a progressive diversification. The effectiveness of dynamic neighborhood strategies can be explained by the fact that a "bad" place on the search landscape given by one neighborhood could be a "good" place on the search landscape given by another neighborhood[2]. Moreover, a solution which is locally optimal with respect to a neighborhood is probably not locally optimal with respect to another neighborhood [16, 15, 13, 7, 19].

VND directly exploits these properties, as it exhaustively searches the neighborhood $\mathcal{N}_k$ of the current solution s^* and it picks the best found neighbor s'. If s' is better than s^* (i.e., if $f(s') < f(s^*)$), it is chosen as current solution and the search proceeds by exploring the neighborhood of s', $\mathcal{N}_k(s')$. Otherwise the neighborhood is switched to $\mathcal{N}_{k+1}$, usually of greater cardinality than the previous one or, if a *distance* between solutions is defined, containing solutions more and more distant from s^*. When the neighborhood structures are somehow related (e.g., of increasing cardinality or distance), the algorithm should be changed a bit, with respect to the original formulation [10, 11], in order to make the algorithm more effective. Instead of continuing the search from s' with the *k-th* neighborhood, the neighborhood sequence is restarted and $\mathcal{N}_1(s')$ is explored[3]. The VND high level algorithm is sketched in Algorithm 2.

It is important to note that VND is an incomplete algorithm: it moves from a solution to a better one, without storing information about the visited search space regions. Therefore it cannot know whether it is revisiting solutions or it has entirely explored the search space.

The VND's search process is almost identical to the search process of CDS. This fact is particularly apparent in the case of search on binary trees, but it holds also for n-ary trees. We suppose to deal with a problem defined over binary variables $x_1, \ldots, x_n$ and a solution s corresponds to a feasible assignment to all variables. The neighborhoods are

[2]With "good" place in the search space we mean an area from which a good local optimum can be easily reached.

[3]In the following, we will refer to this latter formulation of VND.

defined by means of Hamming distance d_H between solutions: the *k-th* neighborhood $\mathcal{N}_k$ of a solution s is defined as

$$\mathcal{N}_k(s) = \{s' \text{ s.t. } s' \text{ is feasible and } d_H(s, s') = k\}.$$

Observe that the whole search space $S = \{0,1\}^n$ can be partitioned with n neighborhoods, i.e., $\forall s\ S = \mathcal{N}_0(s) \cup \mathcal{N}_1(s) \cup \mathcal{N}_2(s) \cup \ldots \cup \mathcal{N}_n(s)$ and $\bigcap_{k=0}^{n} \mathcal{N}_k = \emptyset$, where we have defined $\mathcal{N}_0 := \{s\}$.

The algorithm starts with an initial solution s and exhaustively explores neighborhoods containing solutions more and more distant from s. The exploration of each neighborhood $N_k(s)$ terminates returning the best solution it contains: s'_k *s.t.* $f(s'_k) < f(z)\ \forall z \in N_k(s) \setminus \{s'_k\}$. If $f(s'_k) < f(s)$, then the search starts again from $N_1(s'_k)$. If $k < k_{max}$ it increments k and explore the new neighborhood. Otherwise, if $k = k_{max}$ and no better solution has been found, the algorithm stops. Note that also VND could be extended to a complete algorithm. Indeed, once it has found the optimum s^*, it explores all the possible solutions contained in the neighborhoods of increasing cardinality up to $k_{max} = n$. However, since no memory of the visited space is kept, the search will revisit all the solutions already visited.

3.1.1 Example

For example, let consider a simple problem with three binary variables x_1, x_2, x_3, where every possible assignment is feasible and the objective function is

$$f = 3 - \sum_{i=1}^{3} x_i, \text{ i.e., the greater the number of 1s, the better } f.$$

The optimal solution s^* is the assignment $\{(x_1,1),(x_2,1),(x_3,1)\}$, with $f(s^*) = 0$. With $k_{max} = 3$ VNS finds the optimum in finite number of steps (proportional to 2^3). Supposing to start at solution $\{(x_1,0),(x_2,0),(x_3,0)\}$ (for brevity written as $(0,0,0)$), the sequence of visited solutions is the following:

1. Start at solution $(0,0,0)$.
2. Visit all solutions at $d_H = 1$ from $(0,0,0)$: $(0,0,1),(0,1,0),(1,0,0)$.
3. Pick the best (ties are randomly broken): $(0,1,0)$.
4. Since $(0,1,0)$ is better than $(0,0,0)$, start searching in the smallest neighborhood of $(0,1,0)$.
5. Visit all solutions at $d_H = 1$ from $(0,1,0)$: $(0,1,1),(\mathbf{0},\mathbf{0},\mathbf{0}),(1,1,0)$.
6. Pick the best (ties are randomly broken): $(1,1,0)$.
7. Since $(1,1,0)$ is better than $(0,1,0)$, start searching in the smallest neighborhood of $(1,1,0)$.
8. Visit all solutions at $d_H = 1$ from $(1,1,0)$: $(1,1,1),(\mathbf{1},\mathbf{0},\mathbf{0}),(\mathbf{0},\mathbf{1},\mathbf{0})$.
9. Pick the best: $(1,1,1)$.

10. Since $(1,1,1)$ is better than $(1,1,0)$, start searching in the smallest neighborhood of $(1,1,1)$.
11. Visit all solutions at $d_H = 1$ from $(1,1,1)$: $(\mathbf{1},\mathbf{1},\mathbf{0}),(\mathbf{1},\mathbf{0},\mathbf{1}),(\mathbf{0},\mathbf{1},\mathbf{1})$.
12. None is better than the current one, then $k \leftarrow k+1$
13. Visit all solutions at $d_H = 2$ from $(1,1,1)$: $(\mathbf{1},\mathbf{0},\mathbf{0}),(\mathbf{0},\mathbf{1},\mathbf{0}),(\mathbf{0},\mathbf{0},\mathbf{1})$.
14. None is better than the current, then $k \leftarrow k+1$
15. Visit all solutions at $d_H = 3$ from $(1,1,1)$: $(\mathbf{0},\mathbf{0},\mathbf{0})$.
16. Now $k = k_{max} = 3$, therefore stop and return $(1,1,1)$.

Note that the algorithm revisits a lot of solutions (in bold) and, once the optimum is reached (step 9), all the search space is visited again, because the search in not structured enough to efficiently maintain memory of the visited regions of the search space.

As can be easily observed, the search process generated is the same as the CDS search process. Indeed, CDS explores neighborhoods composed of solutions at constant Hamming distance from the reference solution. Nevertheless, since the search is based on a tree structure, the visited solutions are not visited again. Supposing to start at the same initial solution $(0,0,0)$ as VND and using the same seed for the random number generator, the sequence of visited solutions (leaves of the search tree) is the following:

1. Visit the solution (leaf) at discrepancy 0, i.e., start at solution $(0,0,0)$.
2. Visit all solutions at $d_H = 1$ from $(0,0,0)$: $(0,0,1),(0,1,0),(1,0,0)$.
3. Pick the best (ties are randomly broken): $(0,1,0)$.
4. Since $(0,1,0)$ is better than $(0,0,0)$, set $(0,1,0)$ as reference solution (discrepancies are now evaluated from it).
5. Visit all solutions at $d_H = 1$ from $(0,1,0)$: $(0,1,1),(1,1,0)$ (note that $(0,0,0)$ is not visited, as it has been previously visited).
6. Pick the best (ties are randomly broken): $(1,1,0)$.
7. Since $(1,1,0)$ is better than the current, set $(1,1,0)$ as reference solution.
8. Visit all solutions at $d_H = 1$ from $(1,1,0)$: $(1,1,1)$ (only one is not yet visited).
9. Pick the best: $(1,1,1)$.
10. Since $(1,1,1)$ is better than $(1,1,0)$, set $(1,1,1)$ as reference solution.
11. Visit all solutions at $d_H = 1$ from $(1,1,1)$: $(1,0,1)$ (only one is not yet visited).
12. No better solution has been found and the search tree has been completely visited. Stop and return $(1,1,1)$.

The example shows that the difference between VND and CDS is in the way the search process is generated. In the first case, the smallest amount of information (one solution) is stored and from it an exploration step is performed. In the latter case, the amount of information is higher (a list of open nodes), but it permits to perform an exhaustive search and to avoid revisiting regions of the search space.

Algorithm 3 Local Search inner cycle

$s^{(0)} \leftarrow$ InitialSolution()
$t \leftarrow 0$
while termination conditions not met **do** {Inner cycle}
  $s' \leftarrow$ Choose($\mathcal{N}(s^{(t)}), history$){choose a solution in the neighborhood of the current solution}
  if Accept(s') **then** {if the new solution is accepted}
    $t \leftarrow t+1$
    $s^{(t)} \leftarrow s'$ {the new solution is added to the sequence}
  end if
end while

3.2 Trajectory Methods

In the previous section we have shown how a local search algorithm like VND can be described in the CDS framework. In this section we extend this observation by considering how the most important local search techniques can be performed as (incomplete) variants of CDS and, conversely, how these algorithms can be defined in the frame of tree search (and therefore, in principle, complete).

In the following we will use the expression "local search" with the meaning of *trajectory method*, that is, search process characterized by a trajectory (continuous or not) in the search space $s^{(0)}, s^{(1)}, \ldots, s^{(t)}, \ldots$. The following discussion will consider only binary search trees, thus only problems formulated on binary variables.

The general algorithmic scheme of a trajectory method is composed of two nested cycles. The inner cycle describes the actual local search, where the sequence of solutions composing the trajectory is the result of the iterative application of local moves. The external cycle determines the search strategy (e.g., restarts, perturbations). In this section we focus on the inner cycle, described in Algorithm 3. The algorithm visits a sequence of states by moving from a solution to a neighboring one. The function Choose() selects a neighbor of $s^{(t)}$ according to the designed strategy. The function Accept() applies some criteria to decide whether the chosen neighbor has to be added to the sequence of states. The two extreme cases of acceptance are: either accept every solution or accept s' only if $f(s') < f(s^{(t)})$. Several combinations of the two functions are possible; two of them are largely diffused and represent the core for two well known metaheuristics: Tabu Search [8] and Simulated Annealing [17]. In the first, the function Choose() exhaustively explores the neighborhood of the current solution and picks the best neighbor, while the function Accept() just accepts all the solutions provided by Choose(). In Simulated Annealing, the function Choose() randomly picks a solution in the neighborhood of the current one and Accept() applies a probabilistic criterion to decide if the solution has to be accepted[4].

The function Choose() generally uses some kind of information collected during the search (the *search history*). The search history is usually stored in an intensive way and it is often compacted and incomplete. For example, in Tabu Search a number of last moves is stored in the *tabu list* and used to avoid choosing always the same small set of

[4]The acceptance probability depends, on a parameter called *temperature*: the higher the temperature, the higher the probability to accept a worse solution. The temperature is decreased during the search.

Algorithm 4 High level generalized CDS

$k \leftarrow 1$ {k is the discrepancy}
$s^* \leftarrow$ InitialSolution() {s^* is the reference solution proposed by the heuristics}
while Termination condition not met **do**
  Generate all leaves at discrepancy k w.r.t. s^*: $Leaves(s^*, k)$
  $s' \leftarrow$ Choose($Leaves(s^*, k), history$)
  if Accept(s') **then** {if the new solution is accepted}
    $s^* \leftarrow s'$ {the new solution substitutes the reference one}
    $k \leftarrow 1$
  end if
  $k \leftarrow k + 1$
end while

solutions. Simulated Annealing, on the opposite, does not use any kind of memory.

Trajectory methods are often very effective (on the average) since they try to exploit as much as possible search space (called landscape) properties. Fundamental properties are clustering of good solutions and correlations between objective function and distance from good local optima [16, 13, 7].

3.3 Discrepancy-based Trajectory Methods

CDS can be generalized by allowing more complex and even non-deterministic criteria (see Algorithm 4). Trajectory methods can be implemented in the discrepancy-based framework from the general CDS algorithm by specializing the functions Choose() and Accept() and the termination condition[5]. We have kept the same terminology as in the description of trajectory methods, since the two main functions perform conceptually the same actions. Observe that the definition of a neighborhood structure in trajectory methods corresponds to the generation of leaves at given discrepancy in CDS.

Algorithm 5 Discrepancy-based Best and First Improvement

Choose():
  $s' \leftarrow$ BestOf($Leaves(s^*, k), \{\}$)){for First Improvement replace BestOf with FirstBest}

Accept():
  if $f(s') < f(s^*)$ **then**
    $s^* \leftarrow s'$ {the new solution substitutes the reference one}
  end if

Starting from the simplest case, let us consider how to implement discrepancy-based Best and First Improvement. The specification of Choose() and Accept() is given in Algorithm 5 and the termination condition is $k = k_{max} = 1$. These components can be directly inserted into the general CDS algorithm to obtain *Discrepancy-based Best* and *First Improvement*.

Note that, in this case, a tree structure is the basis of search, which permits to intensionally store the visited regions and the inherent Best (First) Improvement incomplete-

[5]The termination condition is usually $k = k_{max}$.

Algorithm 6 Simulated Annealing

Choose():
$s' \leftarrow$ PickAtRandom($\mathcal{N}(s^*)$)

Accept():
if $f(s') < f(s^*)$ **then**
$s^* \leftarrow s'$ {s' replaces s^*}
else
Accept s' with probability $p(T, s', s^*) = exp(-\frac{f(s')-f(s^*)}{T})$
end if
Update(T)

ness can be overcome by using dynamic neighborhoods (as in VND), by systematically changing the reference solution or by considering a different notion of discrepancy that includes at discrepancy one all the leaves of the search tree.

Among the most known and used trajectory methods, there are Simulated Annealing and Tabu Search. These algorithms are also representative in the choice of functions Choose() and Accept(). Simulated Annealing is described in Algorithm 6.

In the inner cycle of the *metaheuristic version* of Simulated Annealing, a solution in the neighborhood of the current solution is randomly chosen and always accepted if it is better than the current one, or probabilistically accepted if it is worse. It is easy to see that the Choose() function can be implemented in CDS by randomly sampling (probing) a leaf at discrepancy one with respect to the reference solution. Instead of changing the new reference solution only if it is better than the current one, a probabilistic acceptance criterion can be applied so as to possibly accept even worse solutions. Nevertheless, few changes are needed for this implementation to take advantage of tree search structure. First: since CDS does not revisit already visited leaves, when no more leaves to visit in the current neighborhood are left, it is necessary to allow searching at greater discrepancies. Second: a non-deterministic mechanism to randomly sample the leaf at fixed discrepancy has to be provided. This can be done by exploring all leaves at discrepancy k and then randomly pick one of them (and discarding it), or by randomly building a path in the tree (random probing). For the sake of readability, a discrepancy-based Simulated Annealing algorithm is entirely described in Algorithm 7.

The second metaheuristic we discuss is Tabu Search (TS). TS core idea is to apply a Best Improvement local search by using a dynamically changing neighborhood. During the search the most recently visited solutions are stored in a list (called *tabu list*) and deleted from the neighborhood to explore. This mechanism prevents from revisit solutions and thus cycling indefinitely in a small set of solutions. However, the implementation of tabu list as a list of visited solutions is not practical, because managing a list of solutions is highly inefficient. Therefore, instead of solutions themselves, usually *moves* from a solution to one of its neighbors are stored. We call *allowed set* the portion of neighborhood not cut by tabu criteria[6]. The specific functions describing such algorithm are reported in Algorithm 8.

TS can be obtained as a variant of CDS. The neighborhood corresponds to leaves at unitary discrepancy and the best solution in the restricted neighborhood is chosen as

[6]TS uses also *aspiration conditions* which can promote a candidate solution from tabu to allowed.

Algorithm 7 Discrepancy-based Simulated Annealing

$k \leftarrow 1$
Set k_{max}
$s^* \leftarrow$ InitialSolution()
while $k \leq k_{max}$ **do**
  Generate all not yet explored leaves at discrepancy k w.r.t. reference solution s^*: $Leaves(s^*, k)$
  while $Leaves(s^*, k) \neq \emptyset$ and No solution is accepted **do**
    $s' \leftarrow$ PickAtRandom($Leaves(s^*, k)$)
    $q = random(0, 1)$, $p = exp(-\frac{f(s')-f(s^*)}{T})$
    if $f(s^*) < f(s')$ or $q \leq p$ **then**
      $s^* \leftarrow s'$
      $k \leftarrow 1$
    end if
  end while
  $k \leftarrow k + 1$
end while

Algorithm 8 Tabu Search

Choose():
 $s' \leftarrow$ BestOf(s^*,$AllowedSet(s^*, k)$)

Accept():
 $s^* \leftarrow s'$
 Update Tabu Lists and Aspiration Conditions

new reference solution. Also in this case, we can extend the algorithm and relying on the tree search structure. The discrepancy-based TS is sketched in Algorithm 9.

Concluding, we can generalize the examples given above asserting that, with appropriate definitions for the criteria used to explore leaves of the tree and accept the new reference solution, it is possible to rewrite trajectory methods in terms of tree search. This process gives rise to interesting comparisons and considerations on the relations between tree and local search.

We have also experimentally tested the relative performance of LDS, CDS and CDS with first improvement variant (CDS-first). The algorithms have been tested on Weighted MAXSAT instances [7]. The algorithm performance has been evaluated as the number of leaves of the search tree visited before reaching an optimal solution. This parameter can be used to estimate the relative computation cost needed for the algorithms to find the optimum. In Table 1 data on the ratio between the number of visited leaves are reported (results are averaged over 10 instances for each class). We observe that CDS and CDS-first seem to perform considerably better than LDS. We also performed a non-parametric analysis, especially useful in this case, since the computational cost distribution is not normal and the size of samples is small. We compared the three algorithms by ranking

[7] We have generated two sets of random instances. One set with problems with 10 variables and 60 clauses (wmax3sat10-60) and 15 variables and 90 clauses (wmax3sat15-90) for the other one. Clauses have 3 literals each and clause weights follow a uniform distribution in the interval [1, number of clauses].

Algorithm 9 Discrepancy-based Tabu Search

$k \leftarrow 1$
Set k_{max}
$s^* \leftarrow$ InitialSolution()
while $k \leq k_{max}$ **do**
  Generate all not yet explored leaves at discrepancy k w.r.t. reference solution s^*: $Leaves(s^*, k)$
  $AllowedSet \leftarrow \{z \in Leaves(s^*, k)\}$
  if $AllowedSet \neq \emptyset$ **then**
    $s^* \leftarrow$ BestOf(s^*,$AllowedSet$) {the new solution substitutes the reference one}
    $k \leftarrow 1$
  else
    $k \leftarrow k + 1$
  end if
end while

Table 1: Ratio of numbers of visited leaves.

	wmax3sat10-60	wmax3sat15-90
CDS vs LDS	Average ratio: 0.662 Standard deviation: 0.820 Maximum: 2.59 Minimum: 0.078	Average ratio: 0.259 Standard deviation: 0.323 Maximum: 1.00 Minimum: 0.015
CDS-first vs LDS	Average ratio: 0.413 Standard deviation: 0.416 Maximum: 1.39 Minimum: 0.043	Average ratio: 0.247 Standard deviation: 0.426 Maximum: 1.18 Minimum: 0.009

them for each instance on the base of the computation cost [8]. From the results shown in Table 2 we can observe that CDS and CDS' perform significantly better than DS and CDS' performs slightly better than CDS.

4 Related work

This work is inspired by the considerations on variants of LDS discussed in [23]. Discrepancy search has been presented in some variations, usually aimed at improving effectiveness and efficiency. Among them we mention Improved Limited Discrepancy Search [18] which eliminated much of the redundancy in the original algorithm and Depth-bounded Discrepancy Search [22] which first explores discrepancy at lower tree levels, where the probability of wrong heuristic decisions is higher. A very interesting variant of LDS is Weighted Discrepancy Search [2] which extends the concept of discrepancy to a continuum of values between 0 and 1. Moreover, in [14] the relation between LDS and local search is explicitly treated.

[8]Rank 1 for the best, 3 for the worse and 2 for the other.

Table 2: Non-parametric analysis: mean and median rank.

		LDS	**CDS**	**CDS-first**
wmax3sat10-60	Mean	2.4	1.8	1.5
	Median	3	2	1
wmax3sat15-90	Mean	2.9	1.5	1.6
	Median	3	1.5	1.5

LDS is also strongly related to non-systematic backtracking techniques [12] and restart strategies, which are designed to focus the search toward the most promising regions of the search space, or, at least, far away from bad ones.

Finally, we would like to outline that LDS basic ideas have some similarity with the probabilistic construction of solutions in Ant Colony Optimization [6, 5], where a colony of agents constructs a solution by following a heuristic function and the past experience. The agents (called ants) construct a solution as a path in a search tree, by using a probabilistic criterion which leads them to the construction of solutions at some degree of discrepancy from the reference solution. The constructed solutions are not all at the same discrepancy, but they have a discrepancy which depends on the actual values of parameters and random values: the probabilistic construction is such that the greater the discrepancy of a solution, the lower the probability to construct it.

5 Conclusion and Future Work

The LDS variant we have proposed is a bridge between complete search (relying on a tree structure) and local search. These two complementary search methods can be combined so as to take advantage from the strengths of both. One of the advantages is the possibility to improve tree search by exploiting search space properties (clustering of solutions and fitness/distance correlations). Moreover, CDS permits to apply LDS-based search even when a good heuristic function is not available. On the side of trajectory methods, it may be possible to incorporate search tree pruning techniques into local search algorithms. Finally, CDS may provide an important basis for finite convergence proofs of trajectory methods. Furthermore, it also makes it possible to outline limits and weaknesses of such proofs in terms of memory space required by the algorithms.

This is just a preliminary and informal analysis of the subject and several points have to be further investigated. The first question concerns the actual efficiency of trajectory methods implemented as discrepancy search algorithms: Does the efficiency of local search get penalized? Does the biased search of trajectory method actually show its effectiveness? What kind of problems are more suited for these algorithms? Which are less? Furthermore, theoretical questions arise about similarities and differences between local and tree search: Is the edge between the two defined by the kind of information stored during the search? Has it to be structured to guarantee completeness? And, conversely, does unstructured information always lead to incomplete search?

References

[1] E.H.L. Aarts and J.K. Lenstra, editors. *Local Search in Combinatorial Optimization*. Wiley-Interscience, 1997.

[2] T. Bedrax-Weiss. *Weighted Limited Discrepancy Search*. PhD thesis, University of Oregon, 1998.

[3] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. Technical Report IRIDIA/01-13, IRIDIA, Université Libre de Bruxelles - Belgium, 2001.

[4] D. Corne, M. Dorigo, and F. Glover, editors. *New Ideas in Optimization*. Advanced topics in computer science series. McGraw-Hill, 1999.

[5] M. Dorigo, G. Di Caro, and L. M. Gambardella. Ant algorithms for discrete optimization. *Art. Life*, 5(2):137–172, 1999.

[6] M. Dorigo, V. Maniezzo, and A. Colorni. Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man and Cybernetics - Part B*, 26(1):29–41, 1996.

[7] C. Fonlupt, D. Robilliard, P. Preux, and E.G. Talbi. Fitness landscapes and performance of meta-heuristics. In Stefan Voss, Silvano Martello, Ibrahim Osman, and Catherine Roucairol, editors, *Meta-heuristics: advances and trends in local search paradigms for optimization*. Kluwer Academic, 1999.

[8] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[9] S. Hanafi. On the convergence of tabu search. *Journal of Heuristics*, 7:47–58, 2000.

[10] P. Hansen and N. Mladenović. An introduction to variable neighborhood search. In Stefan Voss, Silvano Martello, Ibrahim Osman, and Catherine Roucairol, editors, *Meta-heuristics: advances and trends in local search paradigms for optimization*, chapter 30, pages 433–458. Kluwer Academic Publishers, 1999.

[11] P. Hansen and N. Mladenović. Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130:449–467, 2001.

[12] W. D. Harvey. *Nonsystematic Backtracking Search*. Phd thesis, CIRL, University of Oregon, 1269 University of Oregon; Eugene, OR USA 97403-1269, 1995.

[13] W. Hordijk. A measure of landscapes. *Evolutionary Computation*, 4(4), 1996.

[14] W. K. Jackson, M. Irgens, and W. S. Havens. A re-examination of limited discrepancy search. In *Proceedings of the Constraint Reasoning Workshop at FLAIRS-96*, 1995.

[15] T. Jones. One operator, one landscape. Santa Fe Institute Technical Report 95-02-025, Santa Fe Institute, 1995.

[16] T. Jones and S. Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In L. J. Eshelman, editor, *Proceedings of the 6th Int. Conference on Genetic Algorithms*, pages 184–192. Kaufman, 1995.

[17] S. Kirkpartick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science, 13 May 1983*, 220(4598):671–680, 1983.

[18] Richard E. Korf. Improved limited discrepancy search. In *AAAI/IAAI, Vol. 1*, pages 286–291, 1996.

[19] P. Merz and B. Freisleben. Fitness landscapes and memetic algorithm design. In David Corne, Marco Dorigo, and Fred Glover, editors, *New Ideas in Optimization*, Advanced topics in computer science series. McGraw-Hill, 1999.

[20] T. Stützle. *Local Search Algorithms for Combinatorial Problems - Analysis, Algorithms and New Applications*. DISKI - Dissertationen zur Künstliken Intelligenz. infix, 1999.

[21] S. Voss, S. Martello, I.H. Osman, and C. Roucairol, editors. *Meta-Heuristics - Advances and Trends in Local Search Paradigms for Optimization*. Kluwer Academic Publishers, 1999.

[22] Toby Walsh. Depth-bounded discrepancy search. In *IJCAI*, pages 1388–1395, 1997.

[23] Matthew L. Ginsberg William D. Harvey. Limited discrepancy search. In Chris S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95); Vol. 1*, pages 607–615, Montréal, Québec, Canada, August 20-25 1995. Morgan Kaufmann, 1995.