IDWalk: A Candidate List Strategy with a Simple Diversification Device

Bertrand Neveu[1], Gilles Trombettoni[1], and Fred Glover[2]

[1] Projet COPRIN, CERTIS-I3S-INRIA
Route des lucioles, BP 93, 06902 Sophia Antipolis, France
{Bertrand.Neveu,Gilles.Trombettoni}@sophia.inria.fr
[2] Leeds School of Business, University of Colorado, Boulder
CO 80309-0419, USA
Fred.Glover@colorado.edu

Abstract. This paper presents a new optimization metaheuristic called IDWalk (Intensification/Diversification Walk) that offers advantages for combining simplicity with effectiveness. In addition to the number S of moves, IDWalk uses only one parameter Max which is the maximum number of candidate neighbors studied in every move. This candidate list strategy manages the Max candidates so as to obtain a good tradeoff between intensification and diversification.
A procedure has also been designed to tune the parameters automatically. We made experiments on several hard combinatorial optimization problems, and IDWalk compares favorably with correspondingly simple instances of leading metaheuristics, notably tabu search, simulated annealing and Metropolis. Thus, among algorithmic variants that are designed to be easy to program and implement, IDWalk has the potential to become an interesting alternative to such recognized approaches.
Our automatic tuning tool has also allowed us to compare several variants of IDWalk and tabu search to analyze which devices (parameters) have the greatest impact on the computation time. A surprising result shows that the specific diversification mechanism embedded in IDWalk is very significant, which motivates examination of additional instances in this new class of "dynamic" candidate list strategies.

1 Introduction

Local search is widely used in combinatorial optimization because it often yields a good solution in reasonable time. Among the huge number of metaheuristics that have been designed during the last decades, only a few can obtain a good performance on most problems while managing a small number of parameters.

The goal of our work was to obtain a new computationally effective metaheuristic by performing a study of the most intrinsic phase of the search process, the phase that examines a list of candidates (neighbors) for the next move. This study has led us to design a new, simple and very promising *candidate list strategy (CLS)* to provide a metaheuristic that implements local search devices in the neighborhood exploration phase.

Several CLS procedures have been designed in the past, particularly in connection with tabu search [8]. The IDWalk (Intensification/Diversification Walk)

M. Wallace (Ed.): CP 2004, LNCS 3258, pp. 423–437, 2004.

metaheuristic presented in this paper can be viewed as an extension of the `AspirationPlus` CLS approach [8] that is endowed with a simple and efficient diversification mechanism, called `SpareNeighbor` below, to exit from local minima.

Roughly, `ID Walk` performs S moves and returns the best solution found during the walk. Every time `ID Walk` selects a move, it examines at most `Max` candidate neighbors by selecting them randomly one by one. If the cost of a neighbor x' is *less than or equal to* the cost of the current solution x, then x' is chosen for the next move (rudimentary intensification effort). If no neighbor has been accepted among the `Max` examined, then one of these candidates, with a cost worse than the one of x, is chosen for the next move (rudimentary diversification device). Two variants perform this simple diversification process by setting a specific value to a parameter called `SpareNeighbor`. In the first variant `ID(any)`, where `SpareNeighbor` is set to `any`, *any* previously rejected candidate is randomly selected (among the `Max` visited neighbors). In the second variant `ID(best)`, where `SpareNeighbor` is set to `best`, a *best* (or rather less worsening, in terms of cost) previously rejected candidate is selected.

The first part of the paper introduces the `ID Walk` candidate list strategy. Section 2 gives a detailed description of the two variants of `ID Walk`. Performed on a large sample of benchmarks, `ID Walk` compares very favorably with correspondingly simple instances of leading metaheuristics, notably tabu search, simulated annealing [11] and Metropolis [2].

The second part of this paper tries to understand the role of key intensification and diversification parameters in the optimization process. Section 3 uses tabu search, several variants of `ID Walk`, and our automatic tuning tool to learn more about the impact of parameters on the computation time. Two CLS devices are studied along with the tabu list. This first analysis performed on numerous instances from different problem classes reveals that the `SpareNeighbor` diversification device used by `ID Walk` and tabu search has generally a crucial impact on performance.

2 Description of `ID Walk` and Comparison with Leading Metaheuristics

This section describes the two main variants of `ID Walk`, introduces a straightforward tool used to tune automatically easy to program metaheuristics and reports the experimental results performed on a large sample of problems.

2.1 Description of `ID Walk`

Without loss of generality, the following pseudo-code description assumes that `ID Walk` solves a combinatorial *minimization* problem.

The move selection is the main contribution of `ID Walk` and `Max` is a simple parameter for imposing a ratio between intensification and diversification efforts:

- First, the parameter is often useful to limit the number of neighbors visited in problems with large neighborhoods, to avoid an exhaustive search.

```
algorithm ID Walk (S: number of moves, Max: number of neighbors,
SpareNeighbor : type of diversification)
    Start with a configuration x
    BestConfiguration ← x
    for i from 1 to S do
        Candidate ← 1
        RejectedCandidates ← ∅
        Accepted? ← false
        while (Candidate ≤ Max) and (not Accepted?) do
            x' ← Select (Neighborhood(x), any)
            if cost(x') ≤ cost(x) then
                Accepted? ← true
            else
                RejectedCandidates ← RejectedCandidates ∪ {x'}
            end
            Candidate ← Candidate +1
        end
        if Accepted? then
            x ← x'
        else
            x ← Select (RejectedCandidates, SpareNeighbor)
        end
        if cost(x) < cost(BestConfiguration) then BestConfiguration ← x
    end.
    return BestConfiguration
end.
```

- Second, `Max` must be sufficiently large to allow the search to pursue better solutions in an aggressive way (intensification).
- Third, `Max` must be sufficiently small to allow the search to exit from local minima (diversification).

We have designed two variants of `ID Walk` that embedd two different ways for exiting from local minima, and thus two degrees of diversification. These variants differ only on the way a candidate is chosen when none of them has been accepted (in the `while` loop), that is, they differ on the `SpareNeighbor` parameter.

The Variant `ID(any)`

`ID(any)` (Intensification/Diversification Walk with *Any* "spare" neighbor) corresponds to the algorithm `ID Walk` called with `SpareNeighbor` equal to `any`. In this case, the `Select` function chooses **any** neighbor among the `Max` previously rejected candidates. This neighbor is randomly selected.

The Variant `ID(best)`

`ID(best)` (Intensification/Diversification Walk with *Best* "spare" neighbor) corresponds to the algorithm `ID Walk` called with `SpareNeighbor` equal to `best`. In

this case, the `Select` function chooses a **best** neighbor (i.e., with a lowest cost for the objective function) among the `Max` rejected candidates.

Note that a variant of tabu search also uses a parameter `SpareNeighbor` set to `best`. The behavior of the `TS` used in this paper is similar to the one of `ID(best)` in case all the studied candidates have not been accepted because they are all tabu and do not meet the aspiration criterion: instead of getting stuck, `TS` and `ID(best)` move to the best neighbor, in terms of cost. (More common variants of `TS` select a neighbor that has least recently or least frequently been selected in the past, breaking ties by reference to cost.)

2.2 Automatic Parameter Tuning Procedure

We have implemented a straightforward procedure for tuning the two parameters of `ID Walk`. In accordance with experimental observations, we have assumed, somewhat naively, that for a given walk length S, there exists one value for `Max` that maximizes the performance, i.e., that gives the best average cost of the solution. We suspected however that the best value of `Max` depends on S, so that the implemented procedure for tuning `Max` is called every time the number of moves is increased. The principle of the automatic tuning procedure is the following:

1. $S \leftarrow S_0$ (the walk length S is initialized)
2. Until a maximum running time is exceeded:
   (a) Tune `Max` by running the algorithm `ID Walk` on reduced walk lengths S/K. Let N_i be the value found for the parameter.
   (b) Run the algorithm `ID Walk`(S, N_i,`SpareNeighbor`).
   (c) $S \leftarrow F \times S$

In the experiments presented below, $S_0 = 10^6$ moves, $K = 50$, and we have chosen an increasing factor $F = 4$. Note that we restart from scratch (i.e., from a new configuration) when moving from S_j to S_{j+1}. Only the lattest value of `Max` is reused.

Thus, every phase i, performed with a given walk length S, includes a step (a) tuning `Max` and a solving step (b) keeping `Max` constant. Runs in steps (a) and (b) are performed with a given number of trials (e.g., 10 trials). In the tuning step (a), $P = 10$ different parameter values are tried for N_i in a dichotomous way. The number of moves of our tuning procedure is then: $\sum_{i=1}^{maxiter} S_0(1+P/K)F^i$

The tuning step (a) is perfomed as follows. Starting from an initial value for `Max` (depending on the metaheuristic), `Max` is divided by 2 or multiplied by 2 until a minimum is reached, in terms of cost. The value of `Max` is then refined in a dichotomous way.

Our automatic tuning procedure is also applied to other algorithms with one parameter such as Metropolis and simulated annealing with a linear temperature decrease. In this case, the (initial) temperature replaces the parameter `Max` in the above description.

This tuning procedure has also been extended to tune algorithms with two parameters (in addition to the number S of moves), such as the tabu search and more sophisticated variants of `ID Walk` that will be introduced in Section 3.

2.3 Experiments and Problems Solved

We have performed experiments on 21 instances issued from 5 categories of problems, generally encoded as weighted MAX-CSPs problems with two different neighborhoods, which yields in fact 35 instances. Graph coloring instances are proposed in the DIMACS challenge [16]. We have also tested CELAR frequency assignment problems [5] [1], a combinatorial game, called Spatially-balanced Latin Square, and random Constraint Satisfaction Problems (CSPs).

Several principles are followed in this paper concerning the experimental part. First, we compare metaheuristics that have at most two parameters. Indeed, the simple versions of the leading metaheuristics have only a few parameters and the procedure described above can tune them automatically. Second, for the sake of simplicity, we have not tested algorithms including restart mechanisms. This would make our automatic tuning procedure more complicated. More important, the restart device, although often useful, is in a sense orthogonal to the CLS mechanisms studied in this article that are applied during the move operation. Third, no clever heuristics have been used for generating the first configuration that is generally randomly produced, or only incorporates straightforward considerations[2]. In addition, for three among the five categories of tested problems, two different neighborhoods with specific degrees of intensification are used.

Random CSPs

We have used the generator of random uniform binary CSPs designed by Bessière [1] to generate 30 CSP instances with two different densities. All are satisfiable instances placed before the complexity peak. Ten (resp. twenty) instances in the first (resp. second) category have 1000 (resp. 500) binary constraints, 1000 variables with a domain size 15, and tightness 50 (resp. 88). A tightness 50 means that 50 tuples over 225 (15×15) do not satisfy the constraints.

These constraint satisfaction instances are handled as optimization MAX-CSPs: the number of violated constraints is minimized during the search and a solution is given by a configuration with cost 0.

The usual definition of neighborhood used for CSPs is chosen here: a new configuration x' is a neighbor of the current configuration x if both have the same values, except for one variable v which takes different values in both configurations. More precisely, we define two different neighborhoods:

- (`VarConflict`) Configurations x and x' are neighbors iff v belongs to a violated constraint.
- (`Minton`) Following the Min-conflict heuristics proposed by Minton et al. [15], v belongs to a violated constraint, and the new value of v in configuration x' is different than the old value and produces the lowest number of conflicts.

[1] Thanks to the "Centre d'électronique de l'Armement".

[2] For the latin square problem, a line contains the n different symbols (in any order); for the car sequencing, the initial assembly line contains the n cars (in any order).

Graph Coloring Instances
We have selected three graph coloring instances from the two most difficult categories in the catalogue: the `le450_15c` with 450 nodes and 16680 edges, the `le450_25c` with 450 nodes and 17425 edges, and the more dense `flat300_28` instance with 300 nodes and 21695 edges. All instances are embedded with specially constructed best solutions having, respectively, 15, 25 and 28 colors.

In this paper, graph coloring instances are encoded as MAX-CSP: variables are the vertices in the graph to be colored; the number d of colors with which the graph must be colored yields domains ranging from 1 to d; vertices linked by an edge must be colored with different colors: the corresponding variables must take different values. Coloring a graph in d colors amounts in minimizing the number of violated constraints and finding a solution with cost 0.

The two neighborhoods `VarConflict` and `Minton` defined above are used.

CELAR Frequency Assignment Instances
We have also selected the three most difficult instances of radio link frequency assignment [5]: `celar6`, `celar7` and `celar8`. These instances are realistic since they have all been built from different sub-parts of a real problem. The `celar6` has 200 variables and 1322 constraints; the `celar7` has 400 variables and 2865 constraints; the `celar8` has 916 variables and 5744 constraints.

The variables are the frequencies to be assigned a value which belong to a predefined set of allowed frequencies (domain size about 40). The constraints are of the form $|x_i - x_j| = \delta$ or $|x_i - x_j| > \delta$. Our encoding is standard and creates only the even variables in the CSP along with only the inequalities[3].

The objective function to be minimized is a weighted sum of violated constraints. Note that the weights of the constraints in `celar7` belong to the set $\{1, 10^2, 10^4, 10^6\}$, making this instance highly challenging. In addition to these problems, we have solved the `celar9` and `celar10` instances which have the same type of constraints and also unary soft constraints which assign some variables to given values. All the instances are encoded with the `VarConflict` neighborhood.

Spatially-Balanced Latin Square
The latin square problem consists in placing r different symbols (values) in each row of a $r \times r$ square (i.e., grid or matrix) such that every value appears only once in each row and in each column. We tried an encoding where the latin square constraint on a row is satisfied and a specific neighborhood: swap in a row two values which take part in a conflict in a latin square column constraint. A simple descent algorithm (with allowed plateaus) can quickly find a solution for a latin square of size 100. This suggests that there are no local minima.

The *spatially-balanced* latin square problem [9] must also solve additional constraints on every value pair: the average distance between the columns of two values in each row must be equal to $(r + 1)/3$. The problem is challenging for both exact and heuristic methods. An exact method can only solve the problem for sizes up to 8 and 9. A simple descent algorithm could not solve them. As shown in the experiments below, `TS` and `ID(best)` can solve them easily.

[3] A bijection exists between odd and even variables. A simple propagation of the equalities allows us to deduce the values of the odd variables.

Car Sequencing

The car sequencing problem deals with determining a sequence of cars on an assembly line so that predefined constraints are met. We consider here the nine harder instances available in the benchmark library CSPLib [7]. In these instances, every car must be built with predefined options. The permutation of the n cars on the assembly line must respect the following constraints: consider every option o_i; for any sequence of $q(o_i)$ consecutive cars on the line, *at most* $p(o_i)$ of them must require option o_i, where $p(o_i)$ and $q(o_i)$ are two integers associated to o_i. A neighbor is obtained by simply permuting two cars on the assembly line. Two neighborhoods have been implemented:

- (`NoConflict`) Any two cars can be permuted.
- (`''VarConflict''`) Two cars c_1 and c_2 are permuted such that c_2 is randomly chosen while c_1 violates the requirement of an option o_i, that is, c_1 belongs to a sub-sequence of length $q(o_i)$ containing more than $p(o_i)$ cars with o_i.

2.4 Compared Optimization Metaheuristics

We have compared `ID Walk` with correspondingly simple versions of leading optimization metaheuristics that manage only a few parameters. All algorithms have been developed within the same software system [17]. Our platform `INCOP` is implemented in `C++` and the tests have been performed on a `PentiumIII 935 Mhz` with a Linux operating system. All algorithms belong to a hierarchy of classes that share code, so that sound comparisons can be made between them.

Our Metropolis algorithm is standard. It starts with a random configuration and a walk of length S is performed as follows. A neighbor is accepted if its cost is lower than or equal to the current configuration. A neighbor with a cost higher than the current configuration is accepted with a probability function of a constant temperature. When no neighbor is accepted, the current configuration is not changed. Our simulated annealing `SA` approach follows the same schema, with a temperature decreasing during the search. It has been implemented with a linear decrease from an initial temperature (given as parameter) to 0.

Our simple `TS` variant is implemented as follows: a tabu list of recently executed moves avoids coming back to previous configurations. The aspiration criterion is applied when a configuration is found that is better than the current best cost. The two parameters of this algorithm are the tabu list length (which is fixed) and the size of the examined neighborhood. The best neighbor which is not tabu is selected.

2.5 Results

This section reports the comparisons between `ID(any)`, `ID(best)`, simulated annealing (`SA`), Metropolis and tabu search (`TS`) on the presented problems. 20 among the 35 instances make no significant difference between the tested algorithms and the corresponding results are thus reported in Appendix A.

Note that the goal of these experiments is to compare simple versions of the leading metaheuristics implemented in the same software architecture. We do

not compare with the best metaheuristics on every tested instance. In particular, ad-hoc metaheuristics obtain sometimes better results than our general-purpose algorithms do (see below). However, due to the efficient implementation of our library INCOP and due to the advances provided by ID Walk, very good results are often observed. More precisely:

- As shown below, ID(any) is excellent on CELAR instances and is competitive with state-of-the-art algorithms [12, 20, 13, 3]. The only slightly better general-purpose metaheuristic is GWW_idw, a more sophisticated population-based algorithm with four parameters [18].
- Several ad-hoc algorithms obtain very good results on the 3 tested graph coloring instances [16, 4, 6]. However, the results obtained by ID(best) and TS are impressive on le450_15c. Also, our TS, and our SA with more time [17], can color for the first time flat_300_28_0 in 30 colors.
- ID(best) and TS obtain even better results than the complicated variants of SA used by the designers of the balanced latin square problem [9].
- On car sequencing problems, we obtain competitive results as compared to the local search approach implemented in the COMET library [14] and the ant colony optimization approaches described in [10] (although the lattest seems faster on the easiest car sequencing instances).

CELAR Instances

Table 1. Comparisons between algorithms on CELAR instances. The first column contains the best bound ever found for the instance (not proven for celar7 and celar8). The second column reports the time per trial in minutes. For the other columns, each cell contains the average cost (left) on 10 or 20 trials, and the best cost (right). The numbers are reported minus the value of the best known bound, i.e., 0 means that the bound has been obtained.

	Bound	T	ID(any)		ID(best)		Metropolis		SA		TS	
celar6	3389	14	**58**	**0**	470	304	1659	517	778	150	389	227
celar7	343592	6	**29742**	**406**	$8.6\,10^5$	487406	$5.6\,10^6$	$2.5\,10^6$	$9\,10^5$	113301	$9\,10^5$	376787
celar8	262	50	29	5	131	73	108	38	**19**	**2**	84	38
celar9	15571	3	**0**	**0**	801	671	2188	416	69	0	644	249
celar10	31516	1	**0**	**0**	323	0	59	0	0	0	0	0

The results show that ID(any) is clearly superior to others. The only exception concerns celar8 for which SA is better than ID(any). The following remarks highlight the excellent performance of ID(any):

- ID(any) can reach the best kwown bound for all the instances. With more available time, the best bound 262 is reached for celar8 and bounds less than 343600 can be obtained on the challenging celar7 that has a very chahuted landscape (with constraint violation weights ranging from 1 to 10^6).
- Only a few ad-hoc algorithms can obtain such results on celar6 and celar7 [12, 20], while all the tested algorithms are general-purpose.
- The excellent result on celar9 (10 sucesses on 10 trials) is in fact obtained in 7 s, instead of 3 min for others. The excellent result on celar10 is in fact obtained in 1 s, instead of resp. 47 s and 34 s for SA and TS.

Graph Coloring Instances

Table 2. Comparisons between algorithms on graph coloring instances. For le450_15c, a cell contains the average time required per trial in seconds and the number of times (on 10 trials) the graph can be colored in 15 colors (into parentheses). For le450_25c, a cell contains the average cost (left) and the best cost (right) among the ten trials, obtained in 800 seconds per trial. The numbers are reported minus 25, i.e., 0 means that the graph has been colored in 25 colors.

	Neighborhood	#col	ID(any)	ID(best)	Metropolis	SA	TS
le450_15c	VarConflict	15	**99 (10)**	151 (8)	220 (0)	82 (3)	112 (10)
le450_15c	Minton	15	27 (10)	8 (10)	108 (10)	74 (6)	**3 (10)**
le450_25c	VarConflict	25	3.3 2	3.6 2	4.1 3	5.9 2	**2.3 0**
le450_25c	Minton	25	3.2 3	3.5 2	3.2 2	3.8 2	**2.6 1**

TS obtains generally the best results, especially on le450_25c. It can even color le450_25c once in 800 s with the VarConflict neighborhood.

ID(any) and ID(best) also obtain good results, especially on le450_15c.

Spatially-Balanced Latin Square Instances

Table 3. Comparisons between algorithms on spatially-balanced latin square instances. Each cell contains the average time in seconds per trial (over 10 trials). For blatsq8, all the algorithms always find a solution (10/10). For blatsq9, the number of successes (between 0 to 10) is indicated into parentheses.

	ID(any)	ID(best)	Metrop.	SA	TS
blatsq8	23	**1.5**	10	15	**2.8**
blatsq9	998 (6)	**5 (10)**	26 (10)	46 (10)	**9 (10)**

These tests show that ID(best) and TS clearly dominate the others.

Car Sequencing Instances

Table 4 collapses the results obtained on the two most difficult instances of car sequencing (in the CSPLib): pb10-93 and pb16-81.

The reader can first notice that the results obtained with the more "aggressive" neighborhood are better for all the metaheuristics. The trend is confirmed on the other instances in appendix, although this is not systematic.

Table 4. Comparisons between algorithms on car sequencing instances. Each cell contains the average time in seconds per trial (over 10 trials) and the number of successes into parentheses (between 0 to 10).

	Neighborhood	ID(any)	ID(best)	Metrop.	SA	TS
pb10-93	NoConflict	759 (0)	1842 (6)	**737 (6)**	697 (0)	5902 (4)
pb10-93	VarConflict	1330 (1)	**442 (10)**	509 (7)	709 (4)	1400 (9)
pb16-81	NoConflict	2450 (8)	**499 (10)**	945 (10)	592 (9)	580 (9)
pb16-81	VarConflict	603 (2)	188 (10)	677 (10)	1039 (9)	**99 (10)**

On these instances, ID(best) give the best results (twice) or is only twice slower than the best one, that is Metropolis or TS. ID(any) and SA are less effective.

Summary

On the 15 instances tested above (some of them being encoded with two different neighborhoods), we can conclude that:

- ID(any) dominates others on 4 CELAR and 1 graph coloring instances.
- ID(best) dominates others on 1 spatially-balanced latin square instance and 2 car sequencing instances. It is also generally good when TS is the best.
- Metropolis dominates others on only 1 car sequencing instance and is sometimes very bad.
- SA dominates others only on celar8 and is sometimes very bad.
- TS dominates others on 3 graph coloring instances, 1 spatially-balanced latin square instance and 1 car sequencing instance.

As a result, TS gives the best results on these instances, although it is bad on some CELAR problems, especially celar7.

We should highlight the excellent results obtained by the "best" metaheuristic among ID(any) and ID(best) on all the instances: one version of ID Walk is the best for 8 over the 15 tested instances, and is very efficient on 5 others (generally ID(best)). They are only clearly dominated by TS on the graph coloring instance le450_25c (with the 2 implemented neighborhoods).

2.6 Using the Automatic Tuning Tool in Our Experiments

Our tuning tool has allowed us to perform the large number of tests gathered above. The robustness of the tuning process depends on the tested problem and metaheuristic. Car sequencing instances seem more difficult to be tuned automatically. Also, the tool is less reliable when applied with SA and metaheuristics with two parameters (TS and more sophisticated variants of ID Walk), so that a final manual tuning was sometimes necessary to obtain reliable results. The complexity times reported above do not include the tuning time. However, note that more than 80% of them have been obtained automatically. Especially, Table 5 reports the overall time spent to obtain the results of ID(best) and ID(any) on the 15 instances above. This underlines that all the results, except 1, have been obtained automatically.

For readers who wish to investigate ID_Walk on their own, Table 6 gathers the values selected for Max in our experiments.

Table 5. Total time (tuning+solving) in minutes spent on the 15 instances by ID Walk. (N), (V) and (M) denote the different neighborhoods, resp., NoConflict, VarConflict, Minton.

Instance	celar6	celar7	celar8	celar9	celar10	blatsq8	blatsq9	pb10-93(N)
ID(any)	manual	147	666	36	2	7	311	142
ID(best)	414	200	702	45	51	2.5	4.5	524

Instance	pb10-93(V)	pb16-81(N)	pb16-81(V)	le_15c(V)	le_15c(M)	le_25c(V)	le_25c(M)	
ID(any)	295	611	164	117	24	429	223	
ID(best)	89	374	75	67	4	251	186	

Table 6. Values computed for the Max parameter by the automatic tuning tool (except for celar6).

Instance	celar6	celar7	celar8	celar9	celar10	blatsq8	blatsq9	pb10-93(N)
ID(any)	125	125	120	256	225	175	212	2800
ID(best)	15	7	29	45	16	125	71	1110

Instance	pb10-93(V)	pb16-81(N)	pb16-81(V)	le_15c(V)	le_15c(M)	le_25c(V)	le_25c(M)	
ID(any)	1200	900	562	40	4	100	6	
ID(best)	468	579	179	20	3	93	6	

3 Variants

Several variants of ID Walk have been designed to better understand the role of different devices on performance. Section 3.1 describes these variants and Section 3.2 perform some experiments that lead to significant results.

3.1 Description of Variants

In addition to the number S of moves, the variants ID(a,g) and ID(b,g) have only one parameter (like ID(any) or ID(best)), while ID(a,t) and ID(a,m) have two (like TS).

Variant ID(a,t) (ID(any) with a Tabu List)

ID(a,t) is ID(any) endowed with a tabu list of fixed length. One of the Max neighbor is accepted iff its cost is better than or equal to the current cost and is not tabu.

Variant ID(a,g) ("Greedy" Variant of ID(any))

At every move, ID(a,g) examines the Max candidates: it selects the best neighbor among the Max candidates if one of them improves or keeps the current cost; otherwise it randomly selects *any* of them.

Remark: This variant is allowed by the original move procedure[4] implemented in the INCOP library [17]. More precisely, INCOP allows the user to define a *minimum number* Min *of neighbors* that are visited at every move, among which the best accepted candidate is returned. Without going into details, Min is set to 0 (or 1) in the variants above and is set to Max in the "greedy" variants.

Variant ID(b,g) ("Greedy" Variant of ID(best))

ID(b,g) selects the best neighbor among the Max candidates (Min=Max). ID(b,g) is similar to a TS with no tabu list (or a tabu list of null length).

Other variants could be envisaged. In particular, many well known devices could enrich ID Walk, such as strategic oscillation (i.e., making Max vary with time). However, the aim of the next section is to compare the impact on performance of the following three mechanisms:

- the Min parameter,
- the SpareNeighbor diversification device,
- the tabu list.

[4] The Min parameter is also used in the Aspiration Plus strategy.

3.2 First Comparison Between Local Search Devices

There is no need to go into details to discover a significant result in the local search field. The impact of the `SpareNeighbor` parameter on performance is highly crucial, while it is unused in most metaheuritics and implicit (and fixed to `best`) in a simple form of tabu search. The result is clear on three categories of problems (among five): CELAR, latin square and car sequencing. Therefore we believe that this diversification device should be studied more carefully in the future and incorporated in more metaheuristics. This surprising result also explains the success of `ID(any)` and `ID(best)` (in disjoint cases especially).

On the opposite, we can observe that the impact of `Min` is very weak.

We can finally observe that the tabu list is very effective for graph coloring instances, but the effect on the other categories of problems is not clear.

Note that all the metaheuristics have a good behavior on the uniform random CSP instances. The results are thus reported in Appendix A.

To sum up, 1 category of problems does not discriminate the tested devices, 1 category takes advantage on the tabu list, and 3 categories are well handled by this new `SpareNeighbor` diversification device.

Impact of Parameter `SpareNeighbor`

Table 7 has been arranged so that columns on the left side correspond to metaheuristics with `SpareNeighbor=any`, while columns on the right side correspond to metaheuristics with `SpareNeighbor=best`. The impact of parameter `SpareNeighbor` is very significant on CELAR, latin square and car sequencing problems, for which several orders of magnitude can sometimes be gained by choosing `any` (for CELAR) or `best` (for latin square and car sequencing).

Table 7. Measuring the impact of `Min`, `SpareNeighbor` and the tabu list on performance. Every cell has the same content as described in the previous tables (only the average cost appears for `celar7`). The last column `p-q` gives the length p of the `TS` tabu list and the length `q` of the `ID(a,t)` tabu list.

Instance	Neigh.	ID(a)	ID(a,t)	ID(a,g)	ID(b)	ID(b,g)	TS	p-q
celar6	VarC	58 0	60 0	96 0	470 304	408 308	389 227	1-6
celar7	VarC	$3\,10^4$	$4\,10^4$	$4.8\,10^4$	$8.6\,10^5$	$8\,10^5$	$9\,10^5$	50-48
celar8	VarC	29 5	37 13	38 16	131 73	91 54	84 38	2-45
celar9	VarC	0 0	0 0	0 0	801 671	36 313	644 249	15-12
celar10	VarC	0 0	0 0	0 0	323 0	0 0	0 0	2-5
le_15c	VarC	99 (10)	18 (10)	92 (10)	151 (8)	152 (6)	112 (10)	72-56
le_15c	Mint.	27 (10)	1 (10)	4 (10)	8 (10)	14 (10)	3 (10)	45-10
le_25c	VarC	3.3 2	3.3 2	3.7 3	3.6 2	2.8 1	2.3 0	2-4
le_25c	Mint.	3.2 3	2.4 1	4.1 2	3.5 2	2.8 2	2.6 1	2-4
blatsq8	VarC	99	171	84	2	4	4	0-2
blatsq9	VarC	1410(5)	1581(5)	972(1)	40(10)	16(10)	16(10)	0-3
pb10-93	NoC	759(0)	4301(0)	5979(0)	1842(6)	1698(2)	5902(4)	1-1
pb10-93	VarC	1330(1)	5381(0)	1457(0)	442(10)	1264(10)	1400(9)	1-1
pb16-81	NoC	2450(8)	894(2)	1763(0)	499(10)	1182(10)	580(9)	1-2
pb16-81	VarC	603(2)	890(0)	862(1)	188(10)	236(10)	99(10)	1-4

On car sequencing instances, we can notice that a good performance is obtained by setting SpareNeighbor to best and by using a VarConflict neighborhood. Both trends indicate that the notion of intensification is very significant.

Impact of the Tabu List

The observations are especially based on the comparison between ID(b,g) and TS since ID(b,g) can be viewed as TS with a null tabu list. The comparison between ID(any) and ID(a,t) is informative as well. The interest of the tabu list is not clear on CELAR and car sequencing problems. The impact of the tabu list seems null on latin square when SpareNeighbor is set to best since the automatic tuning procedure selects a list of length 0. It is even slightly counterproductive when SpareNeighbor = any. On the opposite, the gain in performance of the tabu list is quite clear on graph coloring for which ID(a,t) and our TS variant obtain even better results than ID(any) and ID(b,g) resp.

Weak Impact of Parameter Min

The reader should first understand that the parameter Min set to Max allows a more aggressive search but is generally more costly since all the neighbors are necessarily examined.

The observations are especially based on the comparison between ID(any) (Min=0) and ID(a,g) (Min=Max) on one hand, and ID(best) and ID(b,g) on the other hand. First, the impact on performance of setting Min to 0 or Max seems negligible, except for 4 instances (among 15+15): celar7, le450_15c (VarConflict), pb10-93 (VarConflict) and pb16-81 (NoConflict). Second, it is generally better to select a null value for Min, probably because a large value is more costly. Third, we also made experiments with another variant of ID(any) where Min can be tuned between 0 and Max. This variant did not pay off, so that the results are not reported in the paper.

This analysis suggests to not pay a great attention to this parameter and thus to favor a null value for Min in metaheuritics.

4 Conclusion

We have presented a very promising candidate list strategy. Its performance has been highlighted on 3 over the 5 categories of problems tested in this paper. Moreover, a first analysis has underlined the significance of the SpareNeighbor diversification device that is ignored by most of the metaheuristics.

All the metaheuristics compared in this paper have two points in common with ID Walk. They are simple and have a limited number of parameters. Moreover, they use a specific mechanism to exit from local minima.

Our study could be extended by analyzing the impact of random restart mechanisms. In particular, it would allow us to compare ID Walk with the GSAT and the WALKSAT [19] algorithms used for solving the well-known SAT problem (satisfiability of logical propositional formula). Note that WALKSAT is equipped with specific intensification and diversification devices.

ID Walk can be viewed as an instance of the Aspiration Plus strategy, where parameters Min and Plus (see [8]) are set to 0, and where the aspiration level can

be adjusted *dynamically* during the search: the aspiration level (threshold) for `ID Walk` always begins at the value of the current solution, but when none of the `Max` candidates qualify, the aspiration level is increased to the value of "any" candidate (`SpareNeighbor=any`) or of the "best" one (`SpareNeighbor=best`). Since the value of `Min` is not important (with "static" aspiration criteria) and since we have exhibited a significant and efficient instance of a dynamic `Aspiration Plus` strategy, this paper strongly suggests the relevance of investigating additional dynamic forms in this novel and promising class of strategies.

In particular, the `SpareNeighbor` parameter can be generalized to take a value k between 1 (`any`) and `Max` (`best`), thus selecting the "best of k randomly chosen moves". Another variant would select any of the k best candidates.

Acknowledgments

Thanks to Pascal Van Hentenryck for useful discussions on preliminary works.

References

1. C. Bessière. Random Uniform CSP Generators. http://www.lirmm.fr/ bessiere/generator.html.
2. D. T. Connolly. An improved annealing scheme for the qap. *European Journal of Operational Research*, 46:93–100, 1990.
3. S. de Givry, G. Verfaillie, and T. Schiex. Bounding the optimum of constraint optimization problems. In *Proc. CP97*, number 1330 in LNCS, 1997.
4. R. Dorne and J.K. Hao. Tabu search for graph coloring, T-colorings and set T-colorings. In *Meta-heuristics: Advances and Trends in Local Search Paradigms for Optimization*, pages 77–92. Kluwer Academic Publishers, 1998.
5. A. Eisenblätter and A. Koster. FAP web - A website about Frequency Assignment Problems. http://fap.zib.de/.
6. P. Galinier and J.K. Hao. Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization*, 3(4):379–397, 1999.
7. I. Gent and T. Walsh. CSPLib: a benchmark library for constraints. In *Proc. of Constraint Programming CP'99*, 1999.
8. F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.
9. C. Gomes, M. Sellmann, C. van Es, and H. van Es. The challenge of generating spatially balanced scientific experiment designs. In *Proc. of the first CPAIOR conference, LNCS 3011*, pages 387–394, 2004.
10. J. Gottlieb, M. Puchta, and C. Solnon. A study of greedy, local search and ant colony optimization approaches for car sequencing problems. In *Proc. of the EvoCOP conference, LNCS 2611*, pages 246–257, 2003.
11. S. Kirkpatrick, C. Gellat, and M. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.
12. A. Kolen. A genetic algorithm for frequency assignment. Technical report, Universiteit Maastricht, 1999.
13. A. Koster, C. Van Hoesel, and A. Kolen. Solving frequency assignment problems via tree-decomposition. Technical Report 99-011, Universiteit Maastricht, 1999.
14. L. Michel and P. Van Hentenryck. A constraint-based architecture for local search. In *Proc. of the OOPSLA conference*, 2002.

15. S. Minton, M. Johnston, A. Philips, and P. Laird. Minimizing conflict: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.
16. C. Morgenstern. Distributed coloration neighborhood search. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, 1993*, volume 26, pages 335–357. American Mathematical Society, 1996.
17. B. Neveu and G. Trombettoni. INCOP: An Open Library for INcomplete Combinatorial OPtimization. In *Proc. Int. Conference on Constraint Programming, CP'03, LNCS 2833*, pages 909–913, 2003.
18. B. Neveu and G. Trombettoni. When Local Search Goes with the Winners. In *Int. Workshop CPAIOR'2003*, pages 180–194, 2003.
19. B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge. Theoretical Computer Science, vol. 26, AMS*, 2003.
20. C. Voudouris and E. Tsang. Solving the radio link frequency assignment problem using guided local search. In *Nato Symposium on Frequency Assignment, Sharing and Conservation in Systems(AEROSPACE)*, 1998.

A Results over Less Discriminating Benchmarks

Table 8. Comparing metaheuristics on the 20 remaining instances: 4 random CSPs, 2 graph coloring instances, and 14 car sequencing instances. Every cell has the same content as described in the previous tables. For random CSPs, the time includes the tuning step (see Section 2.2) and a run is interrupted as soon as a solution is found.

Instance	Neigh.	ID(a)	ID(a,t)	ID(a,g)	ID(a,m)	ID(b)	ID(b,g)	TS	Metr.	SA
csp1	VarC	91	110	228	88	165	–	121	200	105
csp1	Mint.	127	69	197	253	77	–	64	99	172
csp2	VarC	86	61	211	206	115	–	118	101	245
csp2	Mint.	49	76	126	161	98	–	67	42	43
flat_28	VarC	5.1 4	5.7 4	4.7 3	5.7 5	5.4 5	4.7 3	5 3	5.5 3	6.5 5
flat_28	Mint.	4.5 4	4.9 4	5 4	4.4 3	5.3 4	5 4	5.1 0	4.3 3	5.5 4
pb4-72	NoC	49	32	379	49	96	173	130	40	57
pb4-72	VarC	93	118	143	132	15	41	29	30	126
pb6-76	NoC	0.2	0.1	0.7	1.2	0.2	0.5	0.4	0.2	0.4
pb6-76	VarC	0.1	0.4	0.2	0.15	0.1	0.4	0.4	0.3	1.0
pb19-71	NoC	4	11	28	9	6	14	31	5	10
pb19-71	VarC	3	4	9	3	2	4	4	5	7
pb21-90	NoC	12	22	40	12	6	14	13	10	4
pb21-90	VarC	5	4	13	2	2	5	4	3	9
pb26-82	NoC	22	107	466	70	55	290	150	22	25
pb26-82	VarC	177	291	96	57	15	28	22	25	141
pb36-92	NoC	59	107	866	71	51	103	86	76	241
pb36-92	VarC	64	50	146	43	9	18	23	16	30
pb41-66	NoC	4	5	33	4	7	20	24	8	9
pb41-66	VarC	1.4	1.6	7.1	1.4	0.7	3.2	1.7	0.7	0.7