J Heuristics (2012) 18:353–374
DOI 10.1007/s10732-011-9190-2

Bounding, filtering and diversification in CP-based local branching

Zeynep Kiziltan · Andrea Lodi · Michela Milano · Fabio Parisini

Received: 18 May 2010 / Accepted: 2 September 2011 / Published online: 23 September 2011
© Springer Science+Business Media, LLC 2011

Abstract Local branching is a general purpose heuristic method which searches locally around the best known solution by employing tree search. It has been successfully used in Mixed-Integer Programming where local branching constraints are used to model the neighborhood of an incumbent solution and improve the bound. We propose the integration of local branching in Constraint Programming (CP). This integration is not simply a matter of implementation, but requires a number of significant extensions. The original contributions of this paper are: the definition of an efficient and incremental bound computation for the neighborhood, a cost-based filtering algorithm for the local branching constraint and a novel diversification strategy that can explore arbitrarily far regions of the search tree w.r.t. the already found solutions. We demonstrate the practical value of local branching in CP by providing an extensive experimental evaluation on the hard instances of the Asymmetric Traveling Salesman Problem with Time Windows.

Keywords Constraint Programming · Local search · Cost-based filtering · Additive bounding · Diversification · Traveling Salesman Problem with Time Windows

A shorter version of this paper appeared as Kiziltan et al. (2007).

Z. Kiziltan
Department of Computer Science, Università di Bologna, Bologna, Italy
e-mail: zeynep@cs.unibo.it

A. Lodi (✉) · M. Milano · F. Parisini
DEIS, Università di Bologna, Bologna, Italy
e-mail: andrea.lodi@unibo.it

M. Milano
e-mail: michela.milano@unibo.it

F. Parisini
e-mail: fabio.parisini2@unibo.it

1 Introduction

Local branching is a successful search strategy in Mixed-Integer Programming (MIP) (Fischetti and Lodi 2003) which is present in the current implementation of IBM-Cplex.[1] It is a general purpose heuristic method which searches locally around the best known solution by employing tree search. Local branching has similarities with Limited Discrepancy Search (LDS) (Harvey and Ginsberg 1995) and performs local search in an iterative way. The neighborhoods are obtained by linear inequalities in the MIP model so that MIP searches for an improving solution within a certain "distance" (Hamming distance in case of 0-1 problems) with respect to the incumbent solution. The linear constraints representing the neighborhood of incumbent solutions are called local branching constraints and are involved in the computation of the problem bound. Local branching is a general framework to effectively explore solution subspaces, making use of state-of-the-art MIP solvers. Even though the framework aims to improve the heuristic behavior of MIP solvers, it offers a complete method which can be integrated in any tree based search. In fact, local branching is a general technique applicable to any optimization problem modeled with constraints in contrast to many local search techniques designed and tailored only for specific problems.

In this paper, we propose the integration of the local branching framework in Constraint Programming (CP). The main motivation is to combine the power of constraint propagation and problem bound with a local search method which is applicable to any optimization problem and potentially complete, i.e., it is able to obtain the optimal solution and prove its optimality, if no fail/time limit is imposed. Integrating local branching in CP is not simply a matter of implementation but instead requires significant extensions to the original search strategy. We claim such modifications as the original contributions of this paper and list next.

- First, using a linear programming solver for computing the bound of each neighborhood is not computationally affordable in CP. We have therefore studied a lighter way to compute the bound of the neighborhood which is efficient, effective and incremental, using the additive bounding technique, see Fischetti and Toth (1989).
- Second, we developed a cost-based filtering algorithm for the local branching constraint by extracting reduced-costs out of additive bounding, see Focacci et al. (2002a).
- Third, we have studied a CP-tailored diversification technique that can push the search arbitrarily far from the current incumbent solution whenever needed.

All these aspects have been thoroughly tested on a set of instances of the Asymmetric Traveling Salesman Problem with Time Windows (ATSPTW) (Desrosiers et al. 1995). Our experimental results demonstrate the practical value of integrating local branching in CP. The results can be summarized as follows: (i) on small-size instances where pure CP proves optimality, we find the optimal solution in a shorter time and prove optimality quicker, (ii) on medium-size instances, we sometimes can

[1] IBM ILOG CPLEX: http://www.ibm.com/software/integration/optimization/cplex-optimizer/.

prove optimality where CP fails, (iii) on large-size instances, where both methods fail to prove optimality, we obtain a better solution quality within the same time limit. Moreover, we obtain even better results when compared with LDS (pure and enriched with bound computation) and with pure local search.

2 Formal background

A Constraint Programming model is defined on a set of variables $X = [X_1, \dots, X_n]$, each with a finite domain $\mathcal{D}(X_i)$ of values, and a set of constraints specifying allowed combinations of values for subsets of variables. A feasible solution $\bar{X} = [\bar{X}_1, \dots, \bar{X}_n]$ is an assignment of X satisfying the constraints. The CP solution process interleaves propagation and search: it explores the space of partial assignments using a backtrack tree search, enforcing a local consistency property using either specialised or general purpose propagation algorithms. In the following, we focus on minimization problems where a cost $C_{i,j}$ is associated to each variable value assignment $X_i = j$ and we want to minimize $C = \sum_{i \in N} C_{i,X_i}$ where $N = \{1, \dots, n\}$.

In optimization problems, even if a value in a domain can be part of a feasible solution, it can be pruned if it cannot be part of an optimal solution. This pruning technique is called cost-based filtering (Focacci et al. 2002a). Starting from a problem lower bound *LB* and from a gradient function $grad(i, j)$ that gives for each variable X_i the marginal increase in cost of assigning the value j in a solution, we can prune a value from the domain of a variable if $LB + grad(i, j)$ is greater than the value of the incumbent solution. A valid gradient function is represented by reduced-costs in linear programming. A reduced cost associated with a variable represents the additional cost to pay if such variable increases its value in the optimal solution of 1 unit.

It is a common practice to use tree search methods to solve optimization problems, exploiting carefully tuned successor ordering heuristics to guide the search towards promising regions. When the heuristics guide the search to a failure state, backtracking is performed up to an open decision point, where the choice performed by the heuristics is reverted. We call this alternate choice a "wrong turn". Limited Discrepancy Search (LDS) (Harvey and Ginsberg 1995) addresses the problem of limiting the number of "wrong turns" (i.e., discrepancies) along the way, exploring portions of the search space at increasing discrepancy values w.r.t. a given successor ordering heuristics. These regions at increasing discrepancy k are the so-called k-distance neighborhoods of the solution proposed by the heuristics.

When dealing with optimization problems for which we have an incumbent solution $\bar{X}$ having cost $C_{\bar{X}}$, LDS works by trying to reduce the cost of the incumbent solution, exploring the k-distance neighborhood of $\bar{X}$ via tree search. A main drawback of LDS is that, for large problems, exploring neighborhood at high discrepancy (high values of k) is very time consuming.

Finally, we introduce a mapping between CP and 0-1 Integer Programming (IP) models, which will be used to understand peculiarities of problem structure. We define a directed graph $G = (N, A)$ where arc $(i, j) \in A$ iff $j \in \mathcal{D}(X_i)$. We assume $\mathcal{D}(X_i) \subseteq N$. The corresponding IP model contains binary variables x_{ij} s.t. $x_{ij} = 1 \leftrightarrow X_i = j$; a feasible solution $\bar{X}$ is thus represented as $\bar{x}$, where $\bar{x}_{ij} = 1 \leftrightarrow \bar{X}_i = j$.

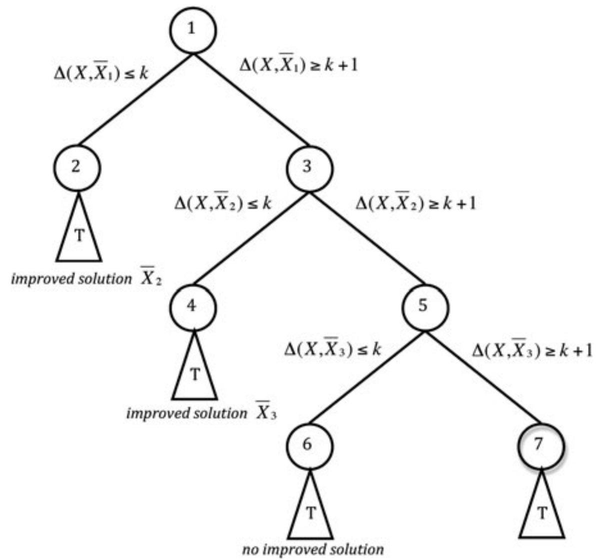

Fig. 1 The basic local branching framework

The cost to be minimized becomes $\sum_{i \in N} \sum_{j \in \mathcal{D}(X_i)} c_{ij} x_{ij}$, where $c_{ij} = C_{i,j}$ when $X_i = j$.

3 Local branching search strategy in MIP

Local branching is a successful search method proposed for 0-1 MIP (Fischetti and Lodi 2003) which is in the current implementation of IBM-Cplex.[2] The idea is that, given a reference solution to an optimization problem, its neighborhood is searched with the hope of improving the solution quality, using tree search. After the optimal solution in the neighborhood is found, the incumbent solution is updated to this new solution and the search continues from it. The basic machinery of the framework is depicted in Fig. 1. Assume we have a tree search method for solving an optimization problem P whose constrained variables are X. For a given positive integer parameter k and a given reference solution $\bar{X}$, the k-OPT neighborhood $\mathcal{N}(\bar{X}, k)$ is the set of feasible solutions of P satisfying the additional local branching constraint $\Delta(X, \bar{X}) \le k$. This constraint defines the neighborhood of $\bar{X}$ by the space of assignments which have at most k different values; the disjunction associated with belonging or not to the neighborhood is used as a branching criterion. More precisely, the solution space associated with the current branching node is partitioned by means of the disjunction $\Delta(X, \bar{X}) \le k \ \vee \ \Delta(X, \bar{X}) \ge k + 1$. In this way, the whole search space is divided into two, and thus exploring each part exhaustively would guarantee completeness. The neighborhood structure is general-purpose in the sense that it is independent of the problem being solved.

[2]Note that the way in which local branching is implemented in Cplex differs from the one described in the following, see Danna et al. (2004) for details.

In Fig. 1, each node of the resulting search tree is labeled with a number. The triangles marked by the letter "T" correspond to the branching sub-trees to be explored. In Fischetti and Lodi (2003), branch-and-bound MIP search is used as tree search method on the sub-trees. The neighborhoods are obtained by imposing linear inequalities in the MIP model.[3] The use of linear constraints in the MIP model clearly provides a tighter bound w.r.t. the one of the original problem.

4 CP-based local branching

The local branching framework is not specific to MIP. It can be integrated in any tree search strategy. We argue that integrating local branching in CP merges the advantages of the intensification and diversification mechanisms specific to local search methods, with constraint propagation that speeds up the neighborhood exploration by removing infeasible variable-value assignments.

The basic scheme of local branching in CP is similar to the one of MIP: a CP model is used to find a first reference solution $\bar{X}_1$. Then a neighborhood $\mathcal{N}(\bar{X}_1, k)$ is searched at the Hamming distance k w.r.t. the reference solution by adding to the CP model the local-branching constraint $\Delta(X, \bar{X}_1) \leq k$, where $\bar{X}_1$ is an assignment of X representing the current reference solution and k is a non-negative integer value. The constraint holds iff the sought assignment of X has at most k different values than $\bar{X}_1$. It can easily be encoded and propagated by exploting the concept of reification; a reified constraint $c \leftrightarrow (b = 1)$ reflects if the constraint c holds into a Boolean variable b, meaning that c holds iff $b = 1$. In our case we reify pairwise equivalences $X_i = \bar{X}_{1_i}$ to Boolean variables B_i and then we post a sum constraint on them. The CP model we consider for the neighborhood is:

$$\min \sum_{i \in N} C_{i,X_i} \tag{1}$$

$$\text{s.t.} \quad \mathit{AnySide_cst}(X) \tag{2}$$

$$\Delta(X, \bar{X}_1) \leq k \tag{3}$$

$$X_i \in \mathcal{D}(X_i) \quad \forall i \in N \tag{4}$$

where C_{i,X_i} is the cost of assigning variable X_i and $\mathit{AnySide_cst}(X)$ is the set of constraints of the original problem on domain variables X. As a consequence, neighborhood exploration via CP clearly benefits from constraint propagation.

The neighborhood can be explored either exhaustively, or up to the first feasible improving solution (if it exists) or stopped after a time or fail limit.

If the neighborhood is explored exhaustively and contains improving solutions, the best one is returned, namely $\bar{X}_2$. Then, the local branching constraint is reversed to $\Delta(X, \bar{X}_1) \geq k + 1$, as in the MIP scheme, and the search now proceeds around the neighborhood of $\bar{X}_2$ by imposing $\Delta(X, \bar{X}_2) \leq k$. Otherwise, if $\bar{X}_2$ is the first

[3] Note that this is trivial in the case the X variables are binary, while it requires much more work for general integer problems; see Fischetti and Lodi (2003) for details.

improving solution and the neighborhood has not been explored exhaustively, the reversed local branching constraint is simply a no-good imposing $[X_1, \ldots, X_n] \neq [\bar{X}_{1_1}, \ldots, \bar{X}_{1_n}]$ and the search again proceeds around the neighborhood of $\bar{X}_2$.

When the neighborhood is proved to contain no improving solution or a time or fail limit is reached without any improving solution, we diversify search and explore other parts of the search tree so as to find new feasible solutions and continue local branching thereafter. Within this schema, which resembles the local branching in MIP, we propose three original contributions that enable us to smoothly accommodate local branching in the CP machinery and to exploit its constraint propagation mechanism.

Bound of the neighborhood It is well known that having a tight problem bound enables the removal of sub-optimal regions of the search tree. Indeed, if the bound is not better than the best solution found so far, the exploration can be interrupted as the sub-tree contains no improving solution. We are interested in computing a bound for the original problem plus a local branching constraint. While in MIP we compute the linear relaxation of the problem within the neighborhood, we have found this is not computationally affordable in CP. We have therefore studied a computationally cheaper way to compute a bound of the neighborhood, based on additive bounding. The bound computation is efficient, effective and incremental and integrated into the local branching constraint. This contribution will be described in Sect. 5.

Cost-based filtering In CP, global constraints are in general used to remove provably infeasible variable-value assignments. Cost-based filtering (Focacci et al. 2002a) is an additional capability that enables the removal of provably suboptimal variable-value assignments. Cost-based filtering can be applied when reduced-costs are available. Its use in CP is more aggressive than that in MIP, where it is known as variable fixing. We show how we can extract reduced-costs from an additive bounding procedure applied to the local branching constraint. This contribution will be described in Sect. 6. On top of reduced costs, we have defined an incremental recomputation of the bound triggered by domain changes and assignments.

Diversification When a neighborhood is proved to contain no improving solution or a time or fail limit is reached without any improving solution, the search can be diversified so as to find a new solution and restart local branching thereafter. In Fischetti and Lodi (2003) the neighborhood is enlarged by changing k and/or by relaxing the constraint which enforces that new solutions should improve the best one. We cannot enlarge the neighborhood too much as the size of the search space, and thus the computational effort needed, would grow too quickly to explore it effectively. This is due to the fact that we explore the neighborhoods in an LDS fashion, and exploring high discrepancy neighborhoods with LDS is very time consuming when dealing with large-scale problems. We here propose a new CP-tailored diversification scheme which collects all the solutions found so far, selects heuristically a given percentage of variables and imposes constraints of difference on these variables. Such constraints enforce that the variables take values different from those they have taken in the current solutions. By playing with the heuristic choice of variables and the number of

variables chosen, it is possible to tune the technique in a number of different ways. Note that the use of constraints would not be effective in local branching in MIP, hence this simple technique is CP-tailored and enables us to search in the regions arbitrarily far from the set of already found solutions. This contribution will be described in Sect. 7.

5 Bound of the neighborhood

The purpose of this section is to exploit the local branching constraint, that implicitly represents the structure of the explored tree, so as to tighten the problem bound. To this purpose, we have developed a novel constraint called *lb_cst*$(X, \bar{X}, k, C)$ which combines together the discrepancy constraint $\Delta(X, \bar{X}_1) \leq k$ (Eq. 3) and the cost function $C = \min \sum_{i \in N} C_{i,X_i}$ (Eq. 1). Using this constraint alone provides very poor problem bounds, and thus poor filtering. If instead we recognize in the problem a combinatorial relaxation *Rel* which can be solved in polynomial time and provide a bound LB_{Rel} and a set of reduced-costs $\bar{c}$, we can feed the local branching constraint with $\bar{c}$ and obtain an improved bound in an additive bounding fashion.

Additive bounding is an effective procedure for computing bounds for optimization problems (Fischetti and Toth 1989). It consists in solving a sequence of relaxations of P, each producing an improved bound. Assume, we have a set of bounding procedures $B_1, \ldots, B_m$. We write $B_i(c)$ for the ith bounding procedure when applied to an instance of P with cost matrix c. Each B_i returns a lower bound LB_i and a reduced-cost matrix $\bar{c}$. This cost matrix is used by the next bounding procedure $B_{i+1}(\bar{c})$. The sum of the bounds $\sum_{i \in \{1,\ldots,m\}} LB_i$ is a valid lower bound for P. An example of the relaxation *Rel* is the Assignment Problem if the side constraints contain an *alldifferent* (Focacci et al. 2002a). Another example is a Network Flow Problem if the side constraints contain a global cardinality constraint (Regin 2002). Clearly, a tighter bound can always be found feeding an LP solver with the linear relaxation of the whole problem, including the local branching constraints, as done in Fischetti and Lodi (2003). We have experimentally noticed that this relaxation is computational too expensive to be used in a CP setting.

To explain how additive bounding can improve the bound obtained by local branching constraints, we use the mapping between CP and IP models described in Sect. 2. Note that this model is useful to understand the structure of the problems we are considering, but it is not solved by an IP solver. We devise a special purpose linear time algorithm as we explain next. The additive bounding procedure uses two relaxations, namely *Rel* and *Loc_Branch*. The latter considers the *lb_cst*$(X, \bar{X}, k, C)$. The solution of *Rel* produces the optimal solution X^*, with value LB_{Rel} and a reduced-cost matrix $\bar{c}$. We can feed the second relaxation *Loc_Branch* with the reduced-cost matrix $\bar{c}$ from *Rel* and we obtain:

$$LB_{Loc_Branch} = \min \sum_{(i,j) \in A} \bar{c}_{ij}\, x_{ij} \tag{5}$$

$$\text{s.t.} \quad \sum_{(i,j) \in S} (1 - x_{ij}) \leq k \tag{6}$$

$$x_{ij} \in \{0,1\}, \quad \forall(i,j) \in A \tag{7}$$

where $S = \{(i,j)|\bar{x}_{ij} = 1\}$. To have a tighter bound, we divide this problem into k problems corresponding to each discrepancy from 1 to k and transforming the constraint 6 into $\sum_{(i,j)\in S}(1 - x_{ij}) = d$ with $d = 1$ to k. The optimal solution of each of these sub-problems can be computed as follows. We start from $\bar{X} = [\bar{X}_1, \ldots, \bar{X}_n]$ and then extract and sort in non-decreasing order the corresponding reduced-costs $\bar{c}_{sorted} = sort(\bar{c}_{1\bar{X}_1}, \ldots, \bar{c}_{n\bar{X}_n})$. The new bound LB_{Loc_Branch} is the sum of the first $n - d$ smallest reduced-costs from $\bar{c}_{sorted}$. Overall, a valid bound for the problem is $LB = LB_{Loc_Branch} + LB_{Rel}$. One can see that only a subset of the domain variables is used to compute the solution; the model can be simplified, as we will show in Sect. 6. The use of the additive bounding here is different from its use in Lodi et al. (2003), because we are starting from a reference solution $\bar{X}$ which is not the optimal solution X^* computed by the first relaxation.

As an example, consider we have $[X_1, \ldots, X_5]$ each ranging on the domain $\{1, 2, 3, 4, 5\}$ with the constraint $alldifferent([X_1, \ldots, X_5])$. In this case, *Rel* is the Assignment Problem (AP). Assume the AP solution is [2, 1, 3, 5, 4] with the optimal value LB_1 and with the reduced-cost matrix $\bar{c}$:

$$\bar{c} = \begin{bmatrix} 3 & 0 & 2 & 4 & 5 \\ 0 & 3 & 4 & 1 & 2 \\ 2 & 1 & 0 & 2 & 1 \\ 5 & 2 & 3 & 2 & 0 \\ 4 & 1 & 3 & 0 & 5 \end{bmatrix}$$

One can see that the assignments corresponding to the AP solution have a reduced cost $\bar{c}_{ij} = 0$, while other assignments have a cost greater than zero. Consider an initial assignment $\bar{X} = [4, 2, 3, 1, 5]$ and that we have $d = 3$, which means the new solution must have three different assignments w.r.t. $\bar{X}$, and conversely keeping two variable assignments the same. Note that constraint 6 is conveniently rewritten as

$$\sum_{(i,j)\in S} x_{ij} = n - d = 2$$

which means that we "pay" in the objective function only the reduced costs of the (two) variables whose assignment is kept as it is, i.e., $x_{ij} = 1$. Then, we first extract the reduced costs of the values in this assignment, which is [4, 3, 0, 5, 5], and we sort this array $\bar{c}_{sorted} = [0, 3, 4, 5, 5]$. Because we need to keep two variable assignments of $\bar{X}$ the same, we choose those that give us the minimum increase in cost. So, we take the first two reduced-costs in $\bar{c}_{sorted}$, and add their sum (i.e., $0 + 3 = 3$) to LB_1 to obtain a valid lower bound. As for the remaining three variables, they can always be assigned to values having zero reduced cost, hence not contributing to the computation of the bound.

6 Cost-based filtering

After computing the lower bound *LB* in an additive way as $LB = LB_1 + LB_2$, we need an associated reduced-cost matrix so as to apply cost-based filtering (Focacci et al.

2002a), i.e., we have to compute the reduced-cost matrix of the problem:

$$LB_2 = \min \sum_{(i,j)\in S} \bar{c}_{ij}\, x_{ij} \tag{8}$$

$$\text{s.t.} \quad \sum_{(i,j)\in S} x_{ij} = n - d \tag{9}$$

$$x_{ij} \in [0, 1] \quad \forall (i, j) \in S \tag{10}$$

This problem is a simpler version of problem 5–7 obtained by removing variables $x_{ij}, \forall(i, j) \notin S$, and relaxing the integrality requirement on the remaining variables. (More precisely, it is a simpler version of its d-th sub-problem in which constraint 6 is written in equality form.) It is easy to see that the two problems are indeed equivalent. First, the integrality conditions are redundant as shown by the algorithm presented in Sect. 5 to solve the problem. Second, because the variables $x_{ij}, (i, j) \notin S$ do not contribute to satisfy constraint 9 they can be dropped to 0 because their reduced costs are non-negative. The dual of such a linear program is as follows:

$$LB_2 = \max \left[(n-d)\pi_0 + \sum_{(i,j)\in S} \pi_{ij} \right] \tag{11}$$

$$\pi_0 + \pi_{ij} \le \bar{c}_{ij}, \quad \forall (i, j) \in S \tag{12}$$

$$\pi_{ij} \le 0, \quad \forall (i, j) \in S \tag{13}$$

Let us partition set S into $S_{\min}$ containing the $n - d$ pairs (i, j) having the smallest reduced costs and $S_{\max}$ containing the remaining d pairs. We also define $\bar{c}_{\max} = \max_{(i,j)\in S_{\min}} \bar{c}_{ij}$. Then, it is not difficult to prove the following result.

Theorem 1 *The dual solution* (i) $\pi_0 = \bar{c}_{\max}$, (ii) $\pi_{ij} = 0, \forall (i, j) \in S_{\max}$, *and* (iii) $\pi_{ij} = \bar{c}_{ij} - \pi_0, \forall (i, j) \in S_{\min}$ *is optimal for system* 11–13.

Proof The above solution is trivially feasible because for $(i, j) \in S_{\max}$ all reduced costs are by definition greater or equal to the largest one in $S_{\min}$ and for $(i, j) \in S_{\min}$ we have an identity. Thus, both Eqs. 12 and 13 are satisfied. Moreover, we can see that our solution is optimal by comparing the objective function value of systems 11–13 and 8–10. Substituting our dual solution into Eq. 11 and noting that the cardinality of $S_{\min}$ is $n - d$, we see that the optimal objective function value is $\sum_{(i,j)\in S_{\min}} \bar{c}_{ij}$. This is the same as the optimal objective function values of system 8–10, as seen by setting x_{ij} to 1 for $(i, j) \in S_{\min}$ and 0 otherwise. □

The reduced-cost matrix $\hat{c}$ associated with the optimal solution of system 8–10 is then constructed as (i) $\hat{c}_{ij} = 0, \forall (i, j) \in S_{\min}$, and (ii) $\hat{c}_{ij} = \bar{c}_{ij} - \bar{c}_{\max}, \forall (i, j) \in S_{\max}$. The reduced costs of variables $x_{ij}, (i, j) \notin S$ do not change, i.e., $\hat{c}_{ij} = \bar{c}_{ij}$.

6.1 Incremental computation of the bound

While exploring the sub-trees, decisions are taken on variable instantiation and as a consequence constraints are propagated and some domains are pruned. We can there-

fore update the problem lower bound by taking into account the most recent situation of the domains. The challenge is to do this in an incremental way, not incurring much extra overhead. The first bound LB_1 can be updated in the traditional way: each time a value belonging to the optimal AP solution is removed, the AP is recomputed with an $O(n^2)$ algorithm (which consists in a single augmenting path) (Focacci et al. 2002a). Using the new AP solution, we can update the second bound LB_2 in a simple way by using some special data structures.

Consider the sets $S_{\min}$ and $S_{\max}$ defined previously. More precisely, $S_{\max}$ initially contains d ordered pairs (i, j) corresponding to the variable-value assignment of the d greatest reduced-costs from $\bar{c}_{sorted}$. Instead, $S_{\min}$ contains the $n-d$ ordered pairs (i, j) corresponding to the $n-d$ smallest reduced-costs from $\bar{c}_{sorted}$. Whilst $S_{\min}$ contains the assignments (variable index-value) that should remain the same w.r.t. $\bar{X}$ because they have the smallest reduced costs, $S_{\max}$ contains the assignments that should change w.r.t. $\bar{X}$ and that conceptually assume a value corresponding to a 0 reduced cost. Note that initially there are n pairs whose first index goes from 1 to n in $S_{\min} \cup S_{\max}$.

6.1.1 Assignment of j to X_i

We distinguish four cases:

1. $(i, j) \in S_{\max}$. A variable that was supposed to change w.r.t. $\bar{X}$ is instead assigned the value in $\bar{X}$. We must update $S_{\min}$ and $S_{\max}$, making sure they remain sorted, as well as update LB_2. To do this, we (a) remove (i, j) from $S_{\max}$; (b) remove $(h, k) = \arg\max_{(m,n)\in S_{\min}} \bar{c}_{mn}$ from $S_{\min}$ and add it ordered in $S_{\max}$; (c) $LB_2 = LB_2 + \bar{c}_{ij} - \bar{c}_{hk}$.
2. $(i, k) \in S_{\max}$ with $k \neq j$. A variable that was supposed to change w.r.t. $\bar{X}$, indeed changes. In the bound, this variable assumed a value corresponding to a 0 reduced cost, while now it assumes the value j whose reduced cost $\bar{c}_{ij}$ may or may not be 0. We update LB_2: $LB_2 = LB_2 + \bar{c}_{ij}$.
3. $(i, j) \in S_{\min}$. No changes are necessary because a variable that was supposed to remain the same w.r.t. $\bar{X}$ remains the same.
4. $(i, k) \in S_{\min}$ with $k \neq j$. A variable that was to remain the same w.r.t. $\bar{X}$ instead changes. We update $S_{\min}$ and $S_{\max}$ as well as LB_2. We (a) remove $(h, p) = \arg\min_{(m,n)\in S_{\max}} \bar{c}_{mn}$ from $S_{\max}$ and insert it in the last position of $S_{\min}$; (b) remove (i, j) from $S_{\min}$; (c) $LB_2 = LB_2 - \bar{c}_{ij} + \bar{c}_{hp}$.

Using the example at the end of Sect. 5, we analyze in depth the four cases. We are given the reduced-cost matrix $\bar{c}$ described in Sect. 5 and an initial assignment $\bar{X} = [4, 2, 3, 1, 5]$ having reduced costs $[4, 3, 0, 5, 5]$.

If we explore the neighborhood of $\bar{X}$ at discrepancy $d = 3$ the sets $S_{\min}$ and $S_{\max}$ are defined in the following way: $S_{\min} = \{(2, 2), (3, 3)\}$, corresponding to $X_2 = 2$ and $X_3 = 3$, $S_{\max} = \{(1, 4), (4, 1), (5, 5)\}$, corresponding to $X_1 = 4$, $X_4 = 1$ and $X_5 = 5$. We take into account the four possible situations depicted above:

1. $X_1 = 4$: X_1 was supposed to change but it is instead assigned the value in $\bar{X}$. We (a) remove $(1, 4)$ from $S_{\max}$; (b) remove $(2, 2) = \arg\max_{(m,n)\in S_{\min}} \bar{c}_{mn}$ from $S_{\min}$ and add it ordered in $S_{\max}$; (c) $LB_2 = LB_2 + 4 - 3$.

(2) $X_1 = 1$: X_1 was supposed to change and indeed changes. We update LB_2: $LB_2 = LB_2 + 3$.
(3) $X_2 = 3$. No changes are necessary because a variable in $S_{\min}$ remains the same.
(4) $X_2 = 4$: X_2 had to remain the same but instead changes. We (a) remove $(1,4) = \arg\min_{(m,n)\in S_{\max}} \bar{c}_{mn}$ from $S_{\max}$ and insert it in the last position of $S_{\min}$; (b) remove $(2,4)$ from $S_{\min}$; (c) $LB_2 = LB_2 - 3 + 4$.

6.1.2 Removal of j from $\mathcal{D}(X_i)$

The only important case is when $(i, j) \in S_{\min}$, where a variable that was supposed to remain the same w.r.t. $\bar{X}$ and that contributed to the computation of the bound changes. We assume X_i changes to a value corresponding to the smallest reduced cost (possibly 0) whose index is k, and then update $S_{\min}$ and $S_{\max}$ as well as LB_2. We (a) remove $(h, k) = \arg\min_{(m,n)\in S_{\max}} \bar{c}_{mn}$ from $S_{\max}$ and insert it in the last position of $S_{\min}$; (b) remove (i, j) from $S_{\min}$; (c) $LB_2 = LB_2 + \bar{c}_{hk} - \bar{c}_{ij}$. That is, the number of variables which must change is decreased from d to $d - 1$.

Again on the example of Sect. 5, we consider the removal of value 2 from the domain of X_2. We (a) remove $(1,4) = \arg\min_{(m,n)\in S_{\max}} \bar{c}_{mn}$ from $S_{\max}$ and insert it in the last position of $S_{\min}$; (b) remove $(2,2)$ from $S_{\min}$; (c) $LB_2 = LB_2 + 4 - 3$.

7 Diversification strategy

When a neighborhood is proved to contain no improving solution or a time limit is reached without any improving solution is found, one possible action to take is to stop local branching with the result of having a tree search-based hill climbing behavior which gets stuck at a local minimum. Another possibility is to escape from local minima, find a new (improving) solution and restart local branching thereafter. We can do so by inheriting from metaheuristics some successful diversification strategies, such as enlarging the neighborhood. This was done in Fischetti and Lodi (2003) by changing k to $k/2$ up to $3/2k$. In this case, one either finds an improving solution in the new neighborhood or applies another local minimum escape strategy, such as removing the upper bounding constraint and accepting non-improving solutions.

Another way to escape from local minima is to simply reverse the last local branching constraint (or imposing a no-good if the local branching constraint cannot be reversed, as it happens when a neighborhood has not been explored in a complete manner) and switching to the exploration of node 7 (Fig. 1) using pure CP. This strategy is not particularly efficient as the search space rooted at node 7 is extremely large. A variant of this strategy is to keep all the branching constraints $\Delta(X, \bar{X}_1) \geq k + 1$, $\Delta(X, \bar{X}_2) \geq k + 1$ and $\Delta(X, \bar{X}_3) \geq k + 1$ at node 7 and then find a new feasible solution $\bar{X}_1'$ which is enforced to be better than $\bar{X}_3$.

Finally, a different and effective diversification technique that we propose in this paper is to force the search towards regions of the search space that are arbitrarily far from the set of solutions found so far. This diversification technique is commonly used in metaheuristic methods, such as Rothberg (2007). Our original contribution consists in using it within a tree search strategy to exploit CP propagation mechanisms. Our experimental results show that such an aggressive diversification scheme

Algorithm 1 Diversification Algorithm

```
for all solutions X̄_j do
   for all variables i do
      L_i = L_i ∪ {X̄_{j_i}}
   end for
end for
populate X_change with p% of the n variables in X
for all X_k ∈ X_change do
   for all j ∈ L_k do
      X_k ≠ j
   end for
end for
```

tailored for CP is the most successful compared to the others described above when local branching in CP is stuck in a local minimum. Algorithm 1 briefly shows how our diversification technique works.

To implement this technique in CP, we benefit from the constraint of difference that simply removes, from the domain of the variables, the values that we do not want to assign any more. In particular, we collect all the solutions found so far in n sets: $L_1, \ldots, L_n$ where L_i is the set of values assigned to the variable X_i in the solutions found so far. We fix a percentage p of variables whose assignments we want to change and we select such variables heuristically. We therefore define a list of $p\%$ variables X_{change} and we impose constraints of difference on all of them. In particular, for each variable X_k in X_{change}, we impose $X_k \neq j$ for each $j \in L_k$. Depending on the percentage p, we can move arbitrarily far from the current set of solutions. Potentially, p could also be 100%.

As an example, if we have obtained the following solutions $[1, 2, 3, 4, 5]$, $[1, 2, 3, 5, 6]$, $[1, 2, 4, 6, 5]$, sets are $L_1 = \{1\}$, $L_2 = \{2\}$, $L_3 = \{3, 4\}$, $L_4 = \{4, 5, 6\}$, $L_5 = \{5, 6\}$. If the percentage p chosen is the 40% we have to change two variables. Suppose that we select X_2 and X_5. Then, we impose $X_2 \neq 2$, $X_5 \neq 5$ and $X_5 \neq 6$. The local branching performs a CP-based exploration of the search space, defined by including these constraints of difference. In case no feasible solution is found with such constraints, we choose other variables and perform a new search. We repeat this step until either we find a solution or the time limit is reached.

8 Experimental set-up

In this section, we describe the experimental setting we used to evaluate local branching in CP.

8.1 Traveling salesman problem with time windows

Given a set of cities and a cost associated to traveling from one city to another, the *Traveling Salesman Problem* (TSP) is to find a (unique) tour such that each city is

visited exactly once and the total cost of traveling is minimized. In Asymmetric TSP (ATSP), the cost from city i to city j may not be the same as the cost from j to i, in contrast with the classical (or symmetric) TSP. The TSP with *Time Windows* (TSPTW) is a time-constrained variant of the TSP in which each city has an associated visiting time and must be visited within a specific time window. TSPTW is a difficult problem which has applications in routing and scheduling. It is therefore extensively studied in Operations Research and CP (see, e.g., Desrosiers et al. 1995; Focacci et al. 2002b). In the following, we concentrate on the asymmetric TSPTW (ATSPTW).

The difficulty of ATSPTW stems on the fact that it involves both optimality and feasibility. ATSP calls for finding a minimum-cost Hamiltonian tour in a graph, which is of course strongly NP-hard. Moreover, scheduling with release and due dates are difficult satisfiability problems.

On the one side, scheduling problems are one of the most established application areas of CP (see, e.g., Baptiste et al. 2001). On the other hand, the use of local search techniques greatly enhances the ability of CP to tackle optimization problems (see, e.g., Milano 2003 for a survey). For this reason the ATSPTW is a good choice for testing the benefits of local branching in CP. Note that we do not aim at competing with the advanced techniques developed specifically to solve this problem. We will show that local branching in CP significantly helps in quickly finding good solutions for the ATSPTW, i.e., more generally for problems with difficult feasibility and optimization aspects together.

8.2 Local branching on ATSPTW

To solve ATSPTW using local branching in CP, we have adopted the model of Focacci et al. (2002b) in which each $Next_i$ variable indicates the city visited after city i. Each city i is associated with an activity and the variable $Start_i$ indicates the time at which the service of activity i begins. Apart from the other constraints, the model consists of the $alldifferent(X)$ constraint (posted on the *Next* variables) and the cost function C, therefore it can be propagated using the cost-based filtering algorithm described in Focacci et al. (2002a). Due to the existence of the local branching constraints (again posted on the *Next* variables) in addition to $alldifferent(X)$ and C, we can use the Assignment Problem as relaxation *Rel* and apply the cost-based filtering described in Sect. 5.

We test different variants of local branching in CP, each tuned for our experiments as we will describe later. We compare the results with a pure CP-based search using the same model except that additive bounding does not exist and thus $alldifferent(X)$ is propagated only by the cost-based filtering algorithm in Focacci et al. (2002a). The CP model is solved with depth-first search enhanced with constraint propagation using the primitives available in Solver and Scheduler. We improve CP search efficiency by using a cost-based scheduling heuristic. At each node of the search tree, the activity in the domain of $Next_i$ with the smallest reduced-cost value $\bar{c}_{ij}$ is chosen and assigned. In case of ties, the activity associated with the earliest start time and the latest end time is considered first. In the remainder of this section, we refer to this heuristic as H_1, the method using local branching in CP as *LBr*, and the pure

CP-based search as *CP*. To allow a fair comparison, heuristic H_1 is used in both *CP* and *LBr*.

The generic local branching framework is employed in the experiments as follows. An initial feasible solution $\overline{Next_1}$ is found using *CP* with heuristic H_1. At each iteration i the neighborhood of the incumbent solution $\overline{Next_i}$ is explored within a time-limited search sub-tree; each sub-tree is searched for increasing values of k with a heuristic exploiting the knowledge about the costs. While searching a sub-tree, we are exploring a limited discrepancy space, assigning some variables their values in the incumbent solution $\overline{Next_i}$ (fixing process) and changing the others. When choosing the variables to fix, at the i-th iteration, we take the incumbent solution $\overline{Next_i}$ and calculate δ_j for each variable $Next_j$, which is the minimum increase in the cost function by changing it from $\overline{Next_{i_j}}$ to a value in $\mathcal{D}(Next_j) \setminus \{\overline{Next_{i_j}}\}$. We then choose the variables with the highest δ, we fix them to the old value and change the others.

As discussed in the previous sections, the local branching algorithm allows a certain degree of flexibility, especially in terms of the definition and exploration of the neighborhoods. There are three main parameters that we can set. First of all, the discrepancy k at which neighborhoods are explored. Then, the time limits, both local, i.e., the time limit over neighborhood exploration and diversification, and global. Finally, the diversification step contains extra parameters to set, i.e., how to choose variables to set constraints of difference on and which percentage of variables to choose.

We performed preliminary testing on the time limits to be adopted, so we report here numbers only for the settings that we found most successful. We will discuss in detail the choice of the other parameters in the following sections.

9 Experimental results

In this section we report experimental results from testing CP-based local branching on ATSPTW. We evaluate both the performance of its components and its overall performance, which is compared against pure CP search. Experiments are conducted using ILOG Solver 6.3 and ILOG Scheduler 6.3 on a 2 GHz Pentium IV running Linux with 1 GB RAM.

9.1 Evaluation of CP-based local branching

The first set of experiments aimed at using the local branching approach in the easiest possible way. As we wanted to create sub-trees big enough to improve solution quality but small enough to be explored fast, we tested k equal either to 3 or to 5. A preliminary investigation showed that setting $k = 5$ defined neighborhoods which were computationally far more expensive to explore than setting $k = 3$, overcoming the benefit of searching a broader space. Consequently, the configuration $k = 3$ was chosen. Table 1 compares our first version of *LBr* with standard *CP* on a set of classical ATSPTW instances proposed by Ascheuer (1995).

The basic version of *LBr* whose results are reported in Table 1 explores each neighborhood stopping at the first solution with no time limit on the neighborhood, performing no diversification. We call this configuration LBr_1. Hence, when a neighborhood is proven to contain no improving solution, the last local branching constraint is

Table 1 Small instances, less than 50 cities, 600 CPU seconds time limit

Instance	Best value	Standard *CP*			LBr_1		
		Time	Time best	Value	Time	Time best	Value
rbg010a	149	0.03	0.01	149	0.03	0.02	149
rbg016a	179	0.20	0.08	179	0.15	0.13	179
rbg016b	142	2.63	1.67	142	1.06	0.70	142
rbg017.2	107	3.97	3.87	107	0.82	0.65	107
rbg017a	146	0.26	0.20	146	0.25	0.22	146
rbg017	148	1.14	0.85	148	0.64	0.42	148
rbg019a	217	0.25	0.20	217	0.25	0.23	217
rbg019b	182	2.59	0.24	182	1.53	1.37	182
rbg019c	190	1.00	0.82	190	0.93	0.59	190
rbg019d	344	600.00	0.05	344	0.44	0.10	344
rbg020a	210	0.25	0.11	210	0.59	0.52	210
rbg021.2	182	0.46	0.41	182	1.87	1.83	182
rbg021.3	182	5.14	4.52	182	3.72	3.50	182
rbg021.4	179	1.24	0.94	179	0.96	0.88	179
rbg021.5	169	1.65	1.34	169	2.92	2.32	169
rbg021.6	134	10.42	10.31	134	9.17	8.93	134
rbg021.7	133	120.73	120.49	133	2.82	2.58	133
rbg021.8	132	6.07	5.88	132	4.65	4.21	132
rbg021.9	132	9.68	9.31	132	5.18	4.42	132
rbg021	190	0.97	0.80	190	0.93	0.60	190
rbg027a	268	4.92	4.12	268	10.04	7.57	268
rbg031a	328	600.00	515.74	371	600.00	18.07	337
rbg033a	433	600.00	467.74	517	42.18	41.95	433
rbg034a	403	600.00	470.39	479	600.00	33.06	407
rbg035a.2	166	465.05	362.89	166	348.46	207.49	166
rbg035a	254	26.96	24.39	254	51.43	37.25	254
rbg038a	466	600.00	8.87	556	600.00	40.21	466
rbg040a	386	600.00	484.97	461	600.00	89.7	415
rbg041a	402	600.00	471.12	615	600.00	166.97	445
rbg042a	411	–	–	–	–	–	–
rbg048a	487	600.00	244.21	658	600.00	519.35	553
rbg049a	484	600.00	226.16	577	600.00	434.31	567

reversed and pure CP is applied. The aim of this experiment was to test on small (and relatively easy instances) the capability of *LBr* to compete with standard *CP* in terms of exact solution within a small global time limit. Table 1 reports for each instance, its name (Instance), the best known solution value (Best value) taken from Baldacci et al. (2010), the computing time (Time), the time to find the best solution (Time best) and the best solution value (Value) for both *CP* and *LBr*. For the instances in which the time limit of 600 CPU seconds is reached the solution value is most probably

not optimal (or at least is not proven to be optimal). The results show that basic *LBr* tends to be faster than standard *CP* both in terms of finding the best solution and of proving optimality. Two more instances are solved to optimality by *LBr* w.r.t. *CP*, namely `rbg019d` and `rbg033a`.

Few more details and comments are required.

- For instance `rbg042a` *CP* does not find any feasible solution within the time limit, thus *LBr* is not even started because it relies (in these test configurations) on *CP* for finding such a first solution.
- As mentioned, LBr_1 explores the neighborhood with no time limit but it stops as soon as a feasible improving solution is found. We, of course, tested the version in which the neighborhood exploration is exhaustive and the results are very similar for these instances while they are worse for larger ones (to be discussed later) and we preferred to keep one version only. Even by stopping at the first improving solution the algorithm remains exact; this is due to the fact that we do not reverse the local branching constraint as we would do when performing exhaustive neighborhood exploration. We instead remove the local branching constraint and add a no-good (see Sect. 4).
- The size of the neighborhood we considered is relatively small: the size of a classical 3-opt move in the Lin and Kernighan (1973) heuristic. In order to understand whether such a neighborhood could be explored more efficiently by a simple enumerative strategy as in classical local search methods we implemented and tested 3-opt. Results showed that such a version, lacking propagation, performs very poorly, much worse than standard *CP*.
- Another natural question concerns the use of Limited Discrepancy Search. We tested LDS by enhancing it with the bound improvement described in Sect. 5 and by using the same propagation we use for *LBr* but the results were disappointing. Note that in such a way we highlight the difference between LDS and *LBr* because the two algorithms are similar with the very relevant difference that the reference solution is (potentially) updated at each iteration in *LBr* while it is not in LDS. This is a strong point, in particular for those problems in which the reference (initial) solution can be poor which is definitely the case of the ATSPTW. This issue will be discussed again later.

Table 2 reports the results of the LBr_1 configuration on larger instances with up to 233 cities. The table introduces a new column, labeled Disc, that indicates the discrepancy separating the initial solution, which is the same for both *CP* and *LBr*, from the final solution, which depends on the configuration under examination. None of the considered approaches exhaustively explores the search space by increasing values of discrepancy, so it may happen that a configuration is able to find a solution of better quality having smaller discrepancy from the initial solution than the competing configuration. Both algorithms were tested with a longer time limit, namely 7,200 CPU seconds.

The interpretation of the results given for small instances in Table 1 is very similar here besides that, even allowing 2 hours of computing time, neither *CP* nor *LBr* are able to solve to optimality any of the large instances. However, *LBr* largely improves over *CP* on 14 of the 18 problems in terms of the best solution found. For these

Table 2 Big instances, 7,200 CPU seconds time limit

Instance	Best value	Standard *CP*				LBr_1			
		Time	Disc	Time best	Value	Time	Disc	Time best	Value
rbg050a	414	7,200	22	5,172.28	648	7,200	45	440.29	430
rbg050b	512	7,200	35	6,860.40	628	7,200	41	1,017.50	570
rbg050c	526	7,200	31	5,071.65	605	7,200	40	843.85	563
rbg055a	814	7,200	25	589.94	879	7,200	30	54.57	814
rbg067a	1048	7,200	26	3,732.71	1227	7,200	38	149.03	1056
rbg086a	1051	–	–	–	–	–	–	–	–
rbg092a	1093	7,200	30	68.71	1537	7,200	67	6,882.03	1159
rbg125a	1409	7,200	39	3,478.96	1991	7,200	76	7,197.21	1697
rbg132.2	1083	7,200	45	6,444.43	1980	7,200	61	7,169.71	2014
rbg132	1360	7,200	42	6,260.76	1933	7,200	60	7,200.00	1772
rbg152.3	1539	7,200	24	2,195.42	2652	7,200	57	7,037.57	2459
rbg152	1783	7,200	50	2,884.81	2532	7,200	69	6,739.56	2329
rbg172a	1799	7,200	51	5,666.87	2838	7,200	28	6,749.19	2930
rbg193.2	2017	7,200	35	333.49	3300	7,200	36	7,140.97	3200
rbg193	2414	7,200	37	3,415.06	3346	7,200	33	6,658.50	3294
rbg201a	2189	7,200	39	6,265.24	3585	7,200	26	7,200.00	3694
rbg233.2	2188	7,200	49	491.43	3909	7,200	32	6,381.93	4059
rbg233	2689	–	–	–	–	–	–	–	–

instances, the two algorithms never obtain the same solution value, thus the "Time best" column is not comparable. However, one can note that for *LBr* "Time best" is generally close to the time limit which indicates that *LBr* is still improving near the end of the available computing time. This is not the case for *CP* which seems to get often stuck in a local minimum early in the computation. In addition, one can note that on the 4 instances in which *LBr* is worse than *CP*, namely `rbg132.2`, `rbg172a`, `rbg233.2` and `rbg233`, the solution found by *CP* has a much higher discrepancy w.r.t. the one of *LBr*. This is a known issue of local branching approaches: if the initial solution is very poor the algorithm continuously improves it but can be slow to converge to a decent value because it explores small neighborhoods of poor solutions while the good ones might be quite far. In order to overcome such a difficulty, we tested the variant of the LBr_1 scheme described in Sect. 7 in which the search is carefully diversified. The results of this version, denoted as LBr_2, are reported in Table 3 where they are compared with standard *CP* and LBr_1 on instances with at least 100 cities.

More precisely, in the LBr_2 version we explore each neighborhood with a time limit of 180 CPU seconds or until the first improving solution is found. In case within the time limit no improving solution has been found, we tune our simple diversification technique by choosing 10% variables in a random way and impose constraints of difference on them w.r.t. their previous values. Such a new neighborhood is then explored with a time limit of 300 CPU seconds and, eventually, the process is iterated if no improving solution is obtained within such a time limit. The table reports, together

Table 3 Big instances with diversification, 7,200 CPU seconds time limit

Instance	Best value	Standard *CP*		LBr_1		LBr_2			
		Disc	Value	Disc	Value	Disc	#Div	Div time	Value
rbg125a	1409	39	1991	76	1697	71	1	1,771.21	1762
rbg132.2	1083	45	1980	61	2014	112	4	2,104.09	1883
rbg132	1360	42	1933	60	1772	36	1	5,393.15	1934
rbg152.3	1539	24	2652	57	2459	109	3	3,889.46	2397
rbg152	1783	50	2532	69	2329	120	1	1,719.33	2281
rbg172a	1799	51	2838	28	2930	129	3	2,948.26	2748
rbg193.2	2017	35	3300	36	3200	174	5	3,950.77	3143
rbg193	2414	37	3346	33	3294	161	5	4,271.16	3217
rbg201a	2189	39	3585	26	3694	179	6	3,390.61	3562
rbg233.2	2188	49	3909	32	4059	207	4	3,263.06	3886

with a selection of the usual information, the number of diversifications (#Div) and the computing time spent in diversification (Div time) for the LBr_2 version.

The results in Table 3 are quite encouraging: the unsatisfactory behavior of LBr_1 on the above mentioned four instances is largely improved with only slight deterioration on a couple of problems, namely `rbg125a` and `rbg132`. In particular, instance `rbg132` shows that the tuning of a diversification strategy is always a bit delicate: almost 5,400 CPU seconds are spent in a unique diversification phase with no effect. Despite this unlucky behavior, the value of the best solution found (1934) is only one unit worse than the one of *CP*, thus demonstrating that the overall behavior of the LBr_2 version is very effective and sufficiently stable.

Note that we have as well experimented with other tunings of our diversification strategy. We have explored changing the parameter p by setting it to 10, 20, 40 and 60. In addition, we have tried different heuristics to decide which $p\%$ of variables to choose by taking into account the way they have changed their values in the solutions found so far. In particular, we have selected the variables which changed their values either the most or the least in the current set of solutions. Unfortunately, none of the combinations of these tunings gave competitive results compared to the random selection of 10% variables.

9.2 Bounding evaluation

In this section we report results from tests aimed at evaluating the computational role of bounds in terms of pruning capabilities and computational speed up.

9.2.1 Impact of additive bounding and cost-based filtering

To test the impact of additive bounding and cost-based filtering on CP-based local branching, we defined two variants of the basic CP-based local branching configuration LBr_1, namely LBr_3 and LBr_4. In particular, LBr_3 behaves exactly like LBr_1, it computes the additive bound, as explained in Sect. 5, but it does not use such a bound

Table 4 Neighborhood exploration: LBr_1 against LBr_3

	Average	Minimum	Maximum
Time	0.78	0.29	0.99
Fails	0.84	0.23	1.00

Table 5 Neighborhood exploration: LBr_1 against LBr_4

	Average	Minimum	Maximum
Time	0.76	0.14	1.06
Fails	0.79	0.14	1.00

for the additional propagation described in Sect. 6. Instead, LBr_4 does not compute the additive bound at all, i.e., it does not use the results in Sects. 5 and 6.

We wanted to evaluate the impact of additive bounding and cost-based filtering over single neighborhood explorations of large ATSPTW instances. We chose the largest ATSPTW instances from Ascheuer (1995), i.e., instances having more than 50 nodes, and we compared the neighborhood exploration components from LBr_1, LBr_3 and LBr_4 on the partial exploration of a given solution neighborhood. For each instance the same initial solution was given to all configurations; the neighborhood exploration was stopped when a given feasible improving solution was found, the same for all configurations. This is obtained through the use of a simple static heuristic, i.e., a heuristic with both fixed variable and value ordering, to guide the search. More precisely, the ordering is that of the variable array in the model, and the smallest domain value of the chosen variable is selected.

We compared the time used from each configuration to perform the neighborhood exploration and the number of fails needed. For each instance a time ratio is computed as time needed from LBr_1 over time needed from the competing configuration, and similarly it is done for the number of fails. Tables 4 and 5 report the comparison between LBr_1 and LBr_3 and between LBr_1 and LBr_4, respectively, reporting the average, minimum and maximum of the time and fails ratio values over all the instances.

The results in Tables 4 and 5 clearly show how the adoption of both additive bounding and cost-based filtering brings a substantial gain to the single neighborhood exploration.

In order to confirm this observation in the real setting, we finally considered the 37 instances with up to 100 nodes (but `rbg042a` and `rbg086a` on which no initial solution is found, see Tables 1 and 2) and we ran the three versions LBr_1, LBr_3 and LBr_4 with the usual heuristic H_1 with a time limit of 600 CPU seconds. Clearly, because H_1 uses the bound information (see, Sect. 8.2) the evolution of the three versions of the code are generally completely different and the faster neighborhood exploration shown by Tables 4 and 5 above might, in principle, not correspond to better performance for LBr_1. However, version LBr_1 is clearly the best one:

- all versions could solve the 24 instances solved by LBr_1 (see, Table 1) in comparable computing times, LBr_1 being nevertheless the fastest;
- on the remaining 13 instances reaching the time limit, LBr_1 stops with a solution at least as good as that of LBr_3 and LBr_4 in all but 1 case and of strictly better quality

on 4 and 5 cases, respectively. When the solutions are identical LBr_1 is always faster.

9.2.2 *Impact of incremental bound computation*

A final note concerns the impact of the incremental bound recomputation. Needless to say, the incremental recomputation of the AP in $O(n^2)$ time is crucial to obtain competitive results. Concerning the incremental recomputation of the bound based on the local branching constraint, we extensively tested its impact in exploring the neighborhood of a given solution (with the same computational setting described in the previous section) w.r.t. a version that recomputes the additive bound from scratch when needed, i.e., building the $S_{\min}$ and $S_{\max}$ sets on the fly. The two versions are identical in terms of pruning capabilities (i.e., fails), while the incremental one is slightly (but consistently) faster, with up to a 5% time improvement in a single neighborhood exploration.

10 Related work

Integration of local search and CP aided tree search so as to tackle difficult constraint problems has been long advocated in the literature. See for instance (Shaw 2011) for a more recent survey which aggroups the proposed approaches. One group uses the modelings aspects of CP in combination with local search methods for finding solutions. The *min-conflicts* algorithm (Minton et al. 1992) and the *Comet* language (van Hentenryck and Michel 2005) are two examples of this approach. Another group employs local search to strengthen the pruning and propagation power CP. This is done for instance in Sellman and Harvey (2002) in the context of solving the social golfer's problem.

The group of integrations that are most related to our work use CP to evaluate the neighborhoods of a local search method. Specifically, Pesant and Gendreau (1999) proposed a framework for local search in which a new neighborhood constraint model is coupled with the original model of the problem. This new model represents the neighborhood of the current solution and is searched using CP aided tree search. This is exactly what our framework does though the way its constructs and searches the neighborhoods significantly differs from that proposed in Pesant and Gendreau (1999). Other related approaches are search methods like *large neighborhood search* (Shaw 1998) which iteratively relaxes a part of the current solution and then re-optimizes that part using for instance CP aided tree search. As introduced in Fischetti and Lodi (2003), local branching is a complete tree-search method designed for providing solutions of better and better quality in the early stages of search by systematically defining and exploring large neighborhoods. Although complete, the idea has been used mainly in an incomplete manner since Fischetti and Lodi (2003): the constraints defining large neighborhoods are iteratively added and the neighborhoods are explored, generally in a non-exhaustive way. When this is done within a local search method, the overall algorithm follows the spirit of both *large neighborhood search* and *variable neighborhood search* (Mladenovic and Hansen 1997). The

main peculiarity of local branching is that the neighborhoods and their exploration are general purpose.

Various other integrations have been proposed. Among them, Beck's *solution-guided multi-point constructive search* (Beck 2007) shows similarities to our framework as it makes use of the existing feasible solutions to guide search for optimal solution. The search algorithm initializes a set of elite solutions and enters in a while loop. In each iteration, with probability p, search is started from an empty solution or from a randomly selected elite solution r and continues until a fail limit. In the first case, if the best solution found s is better than the worst elite solution w, s replaces w. In the second case, r is used as a value selection heuristic and is replaced by s if s is a better solution. Local branching is fundamentally different from the solution-guided multi-point constructive search as it searches the neighborhood of the best known solution in a discrepancy-based fashion and applies diversification strategies in case stuck in local minima.

Related work also includes relaxations in propagation (Sellmann and Fahle 2003) and the use of Hamming distance to find similar and diverse solutions in CP (Hebrard et al. 2005). The local branching framework constitutes of many basic components that the list of related work is by no means complete.

11 Conclusions

We have shown that CP-based local branching raises a variety of issues which do not concern only the implementation. Even if the neighborhoods are not defined by linear inequalities and not explored by MIP techniques, both the definition and the exploration are as general as in the MIP context. The bound is applicable to any problem, as long as we can recognize in the problem a combinatorial relaxation which can be solved in polynomial time and provide a bound and a set of reduced-costs.

One could as well use a k-discrepancy constraint (Lodi et al. 2003) to define the neighborhood and a discrepancy-based technique (in the spirit of LDS (Harvey and Ginsberg 1995)) for its effective exploration. In this way, both the modeling and the search of neighborhoods would benefit from the important features of CP, as done in the MIP counterpart. CP-based local branching, however, benefits from diverse areas: power of propagation and effective heuristics of CP, cost-based filtering and additive bounding borrowed from MIP (for proving optimality), and intensification and diversification borrowed from local search (for finding good solutions earlier). Our experiments on the time-constrained variant of the classical ATSP show the benefits of local branching in CP. Although we believe the ATSPTW is a good representative problem to assert the effectiveness of the proposed approach in CP, clearly additional tests are needed to draw more complete conclusions. In particular, different problem classes and other search strategies should be taken into account.

Further work goes in the direction of developing an automatic tuning process of the CP-based local branching parameters; recent work (Hutter et al. 2010) shows that this is an important field to explore. Moreover, other research directions include the computation of a stronger and more sophisticated lower bound taking into account more than one reference solution and the analysis of problems involving different relaxations.

Acknowledgements We thank two anonymous referees for their detailed reading and useful remarks.

References

Ascheuer, N.: Hamiltonian path problems in the on-line optimization of flexible manufacturing systems. PhD thesis, Technische Universität Berlin (1995)

Baldacci, R., Mingozzi, A., Roberti, R.: New state space relaxations for solving the traveling salesman problem with time windows. Technical Report University of Bologna (2010)

Baptiste, P., Le Pape, C., Nuijten, W.: Constraint-based Scheduling. Kluwer Academic, Dordrecht (2001)

Beck, J.C.: Solution-guided multi-point constructive search for job shop scheduling. J. Artif. Intell. Res. **29**, 49–77 (2007)

Danna, E., Rothberg, E., Le Pape, C.: Exploring relaxation induced neighborhoods to improve mip solutions. Math. Program. **102**, 71–90 (2004)

Desrosiers, J., Dumas, Y., Solomon, M.M., Soumis, F.: Time constrained routing and scheduling. In: Network Routing, pp. 35–139 (1995)

Fischetti, M., Lodi, A.: Local branching. Math. Program. **98**, 23–47 (2003)

Fischetti, M., Toth, P.: An additive bounding procedure for combinatorial optimization problems. Oper. Res. **37**, 319–328 (1989)

Focacci, F., Lodi, A., Milano, M.: Optimization-oriented global constraints. Constraints **7**, 351–365 (2002a)

Focacci, F., Lodi, A., Milano, M.: A hybrid exact algorithm for the TSPTW. INFORMS J. Comput. **14**, 403–417 (2002b)

Harvey, W., Ginsberg, M.L.: Limited discrepancy search. In: Proc. of IJCAI-95, pp. 607–615. Morgan Kaufmann, San Mateo (1995)

Hebrard, E., Hnich, B., O'Sullivan, B., Walsh, T.: Finding diverse and similar solutions in constraint programming. In: Proc. of AAAI-05, pp. 372–377. AAAI Press/The MIT Press, Cambridge (2005)

Hutter, F., Hoos, H.H., Leyton-Brown, K.: Automated configuration of mixed integer programming solvers. In: CPAIOR, pp. 186–202 (2010)

Kiziltan, Z., Lodi, A., Milano, M., Parisini, F.: CP-based local branching. In: Proc. of CP-07. LNCS, vol. 4741, pp. 847–855. Springer, Berlin (2007)

Lin, S., Kernighan, B.W.: An effective heuristic algorithm for the traveling-salesman problem. Oper. Res. **21**, 498–516 (1973)

Lodi, A., Milano, M., Rousseau, L.M.: Discrepancy based additive bounding for the all different constraint. In: Proc. of CP-03. LNCS, vol. 2833, pp. 510–524. Springer, Berlin (2003)

Milano, M.: Constraint and Integer Programming: Toward a Unified Methodology. Kluwer Academic, Dordrecht (2003). Chap. 9.

Minton, S., Johnston, M.D., Philips, A.B., Laird, P.: Minimising conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. Artif. Intell. **58**, 161–205 (1992)

Mladenovic, N., Hansen, P.: Variable neighbourhood search. Comput. Oper. Res. **24**, 1097–1100 (1997)

Pesant, G., Gendreau, M.: A constraint programming framework for local search methods. J. Heuristics **5**, 255–279 (1999)

Regin, J.C.: Cost-based arc consistency for global cardinality constraints. Constraints **7**(3–4), 387–405 (2002)

Rothberg, E.: An evolutionary algorithm for polishing mixed integer programming solutions. INFORMS J. Comput. **19**, 534–541 (2007)

Sellman, M., Harvey, W.: Heuristic constraint propagation—using local search for incomplete pruning and domain filtering of redundant constraints for the social golfer problem. In: Proc. of CP-AI-OR, pp. 191–204 (2002)

Sellmann, M., Fahle, T.: Constraint programming based Lagrangian relaxation for the automatic recording problem. Ann. Oper. Res. **118**, 17–33 (2003)

Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: Proc. of CP-98. LNCS, vol. 1520, pp. 417–431. Springer, Berlin (1998)

Shaw, P.: Constraint programming and local search hybrids. In: Milano, M., van Hentenryck, P. (eds.) Hybrid Optimization: The Ten Years of CPAIOR, pp. 271–304. Springer, Berlin (2011)

van Hentenryck, P., Michel, L.: Constraint-based Local Search. MIT Press, Cambridge (2005)