Variable neighborhood search for binary integer programming problems

Håkon Bentsen and Lars Magnus Hvattum

Faculty of Logistics
Molde University College, Norway
hakon.bentsen@himolde.no, *hvattum@himolde.no*

October 5, 2021

Abstract

General solvers exist for several types of optimization problems, with the commercially available solvers for mixed integer programming (MIP) being a prime example. Although binary integer programming (BIP) can be used to model a wide variety of important combinatorial optimization problems, relatively few contributions have been made to develop heuristic algorithms for BIP. This paper examines whether variable neighborhood search can be successfully used to tackle BIP instances, when avoiding very large neighborhoods explored by the means of external MIP solvers. The results indicate that methods based on variable neighborhood search are more successful than exact and heuristic commercial solvers on certain types of instances, while the opposite holds true on others. A general variable neighborhood search proves very effective on instances with up to 200 variables, in particular some instances that are tightly constrained.

Keywords: black-box solver; 0-1 integer programming; variable neighborhood descent; mathematical programming.

Biographical notes: Håkon Bentsen is a PhD student at Molde University College, Norway. He has a Master of Science in Logistics from the Molde University College, from 2016.

Lars Magnus Hvattum is a Professor of Quantitative Logistics in the Faculty of Logistics at Molde University College. He has a PhD in Logistics from the Molde University College in 2007, and a Cand. Scient Degree in Informatics from the University of Bergen, Norway, in 2003.

1 Introduction

Binary integer programming (BIP) can be used to model a wide range of hard optimization problems with many important real world applications (Koch et al., 2011). Even so, there is a

relative scarcity of solvers targeting this particular class of problems (Bertsimas et al., 2013), with most efforts either focusing on specific binary optimization problems or even more general formulations, such as mixed integer programming (MIP).

Among heuristics developed specifically for BIP, it is worthwhile to mention the black box scatter search by Gortázar et al. (2010), and the local search by Bertsimas et al. (2013). Al-Shihabi (2021) created a matheuristic for BIP based on local search and the core concept (Huston et al., 2008), but tested the proposed method on a limited set of test instances. Other researchers have focused on specific aspects of solving binary optimization problems. Trapp and Konrad (2015) considered how to find multiple different optimal or near-optimal solutions by using fractional programming, while Glover et al. (2019) discussed how to systematically generate diverse solutions to binary optimization problems for use within heuristic solution methods. Exact algorithms for BIP also receive sporadic attention. Glover et al. (2021) recently investigated bookkeeping details of implicit enumeration, an algorithm suggested a long time ago by Balas (1965).

The motivation for this paper is to examine whether variable neighborhood search (VNS) is a viable metaheuristic for solving BIPs. In particular, the aim is to develop an understanding regarding which types of neighborhood structures can be useful within the VNS framework when the problem solved is of such a general character, without specific concrete problem structures to exploit. To this end, the focus is on the simplest form of VNS, namely variable neighborhood descent (VND). In addition, a general variable neighborhood search (GVNS) is examined, wherein the VND is used as a subroutine. Hansen et al. (2010) provide an extensive overview of VNS in many forms and for several applications.

When implementing VND, one seeks to define a hierarchy of neighborhoods, so that once the current solution is a local optimum with respect to the current neighborhood, the exploration continues with the next neighborhood in the hierarchy. If an improved solution is found in a neighborhood, the move is performed and the search moves back to the initial neighborhood. While VNS has been applied to several types of binary optimization problems, the neighborhoods are often explored by mathematical programming solvers. Puchinger and Raidl (2008) implemented a VNS for the *multidimensional knapsack problem* (MKP), with neighborhoods being explored using a MIP solver. The neighborhoods consisted of either removing or adding an exact number of items, and then adding or removing any number of other items to either improve the solution or regain feasibility. Hanafi et al. (2009) also considered the MKP and a MIP solver to explore neighborhoods. Turajlić and Dragović (2012) considered a *multiple-choice multidimensional knapsack problem* (MMKP) for selecting web services, and used tabu search as

the local search algorithm in a VNS. The neighborhoods considered were defined by the number of services for which the solutions can differ. Hansen et al. (2006) focused on more general MIP problems, but defined neighborhoods in terms of the binary variables of the problem. Again, a MIP solver was used to explore neighborhoods.

Rather than relying on external MIP solvers to explore neighborhoods, this paper focuses on defining neighborhoods that can be explored systematically using specially tailored code. This follows the structure of the improvement method by Gortázar et al. (2010), which consisted of two neighborhoods: one considering flipping the value of a single variable, and one considering swapping the values of two variables. Two different types of neighborhoods are considered for the VND here: one consists of simultaneously flipping a given number of variables, while the other consists of first flipping a given number of variables and then sequentially flipping additional variables as long as it benefits the resulting solution. For both types of neighborhoods, the motivation is to derive a hierarchy of neighborhoods of increasing complexity. The first generalizes standard neighborhoods for binary optimization problems, whereas the second is somewhat related to the concept of ejection chains (Rego et al., 2004).

Hierarchies of neighborhoods based on simultaneously flipping variables tend to grow very quickly in size, as a function of the number of variables flipped. Strategies to reduce the size of the neighborhoods are therefore relevant. In this work we investigate three strategies, based either on splitting neighborhoods into different parts or on avoiding the evaluation of certain moves based on the structure of the constraint matrix. Furthermore, besides techniques to reduce the size of neighborhoods, the efficiency of the VND may depend on the strategy chosen for exploring the neighborhoods, such as the sequence of neighborhoods or whether to use best improvement or first improvement when selecting moves.

The aim of the work is to examine whether VNS is a viable strategy for solving the general BIP. This can be summarized in three research questions to be addressed: 1) can a useful and meaningful hierarchy of neighborhoods be devised, 2) are methods proposed to reduce the neighborhood size effective when used within a VNS, and 3) are the obtained solutions competitive when compared with primal bounds obtained within the same time limits by available commercial software that can be used to solve BIP.

The remainder of this paper is structured as follows. Section 2 provides a formal definition of the BIP, as well as BIP formulations of three widely different binary optimization problems that are used in computational experiments to evaluate the research questions. Then, Section 3 describes in detail the implementations of the VND and GVNS and the different neighborhood structures

considered. The computational experiments, including initial experiments on a training set of instances and final experiments on a test set of instances, are described in Section 4. The conclusions regarding the research questions are presented in Section 5.

2 Problem description

In this section we first present the general formulation of a BIP, and then provide details on three types of specialized problems that are used to obtain test instances for the computational experiments.

2.1 Binary integer programming problem

The target is to create a solver for the BIP formulated w.l.o.g. as:

$$\max Z = \sum_{j=1}^{n} c_j x_j,$$

subject to

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i, \qquad i = 1, \ldots, m,$$

$$x_j \in \{0, 1\}, \qquad j = 1, \ldots, n,$$

where all a_{ij}, b_i, and c_j are assumed to be integers (not necessarily positive). This formulation does not exclude instances with equality constraints or $\geq$-constraints, as the former can be transformed into two inequalities of opposite directions and the latter can be transformed into a $\leq$-constraint by multiplying with -1. Non-integral rational coefficients can be converted into integers by scaling each constraint of the problem by the least common multiple of the denominators of the non-zero coefficients in the constraint. The same process can be applied if there are rational coefficients in the objective function.

In the following we describe three different combinatorial optimization problems that can be formulated as BIPs and that are used in the computational experiments presented later. The choice of problems is based on the diverse structure of the benchmark instances used in the literature, which is emphasized in the description below.

2.2 Optimum satisfiability problem

The *optimum satisfiability problem* (OptSAT) was put forth by Davoine et al. (2003), who referred to it as the Boolean optimization problem. The mathematical formulation used here was described by da Silva et al. (2020). Consider a set of m clauses forming a Boolean expression that must be satisfied. Let A_i be the set of non-negated variables appearing in clause i, B_i the set of negated variables, and c_j the profit obtained by setting variable x_j to true (equivalently to 1). The problem is then written as

$$\begin{aligned} \max\ z &= \sum_{j=1}^{n} c_j x_j, & \\ \sum_{j \in A_i} x_j - \sum_{j \in B_i} x_j &\leq |A_i| - 1, & i \in \{1, \ldots, m\} \\ x_j &\in \{0, 1\}, & j \in \{1, \ldots, n\}. \end{aligned}$$

2.3 Multidemand multidimensional knapsack problem

Cappanera and Trubian (2005) presented the most comprehensive early work on the *multidemand multidimensional knapsack problem* (MDMKP). The MDMKP combines knapsack constraints and covering constraints, and can be formulated as

$$\begin{aligned} \max\ z &= \sum_{j=1}^{n} c_j x_j, & \\ \sum_{j=1}^{n} a_{ij} x_j &\leq b_i, & i \in \{1, \ldots, m\}, \\ \sum_{j=1}^{n} a_{ij} x_j &\geq b_i, & i \in \{m+1, \ldots, m+q\}, \\ x_j &\in \{0, 1\}, & j \in \{1, \ldots, n\}, \end{aligned}$$

where $b_i > 0$ and $a_{ij} \geq 0$. The MDMKP includes as a special case the MKP when $q = 0$. Lai et al. (2019) provided a comprehensive overview of recent contributions for the MDMKP.

2.4 Multiple-choice multidimensional knapsack problem

In the MMKP, n disjoint groups of items, $G_1, G_2, \ldots, G_n$, are given. Exactly one item j from each group G_i should be selected. The profit from the selected item is c_{ij}. However, the selection is limited by m knapsack constraints. For knapsack k, a capacity of b_k is enforced, and the weight of item j from group i is a_{ijk}. This leads to the following formulation:

$$\begin{aligned}
\max\ z &= \sum_{i=1}^{n} \sum_{j \in G_i} c_{ij} x_{ij}, & \\
\sum_{i=1}^{n} \sum_{j \in G_i} a_{ijk} x_{ij} &\leq b_k, & k \in \{1, \ldots, m\}, \\
\sum_{j \in G_i} x_{ij} &= 1, & i \in \{1, \ldots, n\}, \\
x_{ij} &\in \{0, 1\}, & i \in \{1, \ldots, n\}, j \in G_i.
\end{aligned}$$

All of the a_{ijk}-parameters are non-negative and all of b_k and c_{ij} are strictly positive. As opposed to the OptSAT and the MDMKP, the MMKP includes equality restrictions that are non-trivial to satisfy in combination with the knapsack constraints.

3 Variable neighborhood search

The basic structure of the VND is taken from the introductory text by Hansen et al. (2010). As the resulting method is intended to be run until a given time limit, our implementation involves a random restart, where each time the VND is started from a randomly generated solution. Pseudo-code for the overall VND is given in Algorithm 1.

Algorithm 1 Pseudo-code for variable neighborhood descent.

Input: the number of neighborhoods k^{MAX}
Initialize x^{BEST} as an arbitrary solution
while time limit is not reached **do**
  Generate x such that each x_j is randomly set to 0 or 1
  Set $k \leftarrow 1$
  while time limit is not reached and $k \leq k^{MAX}$ **do**
    Let $x' \in N_k(x)$ be the best, or first improving, neighbor of x in N_k
    if x' is better than x **then**
      $x \leftarrow x'$
      $k \leftarrow 1$
    else
      $k \leftarrow k + 1$
    end if
  end while
  if x is better than x^{BEST} **then**
    $x^{BEST} \leftarrow x$
  end if
end while
Output: the best solution found x^{BEST}

A VND can be used as a part of a GVNS, in which the VND is used to improve solutions obtained by shaking the current solution by executing a random move from given neighborhoods. Algorithm 2 provides pseudo-code for this variant of VNS, which is based on the description by Hansen et al. (2010). The outer neighborhoods used to shake the solution are typically larger than the neighborhoods used within the VND.

Algorithm 2 Pseudo-code for general variable neighborhood search.

Input: the number of outer neighborhoods k^{MAX}
Initialize x^{BEST} as an arbitrary solution
while time limit is not reached **do**
  Generate x such that each x_j is randomly set to 0 or 1
  Set $k \leftarrow 1$
  while time limit is not reached and $k \leq k^{MAX}$ **do**
    Select at random $x' \in N_k(x)$
    Improve x' using VND, to obtain x''
    if x'' is better than x **then**
      $x \leftarrow x''$
      $k \leftarrow 1$
    else
      $k \leftarrow k + 1$
    end if
  end while
  if x is better than x^{BEST} **then**
    $x^{BEST} \leftarrow x$
  end if
end while
Output: the best solution found x^{BEST}

3.1 Problem representation

The internal representation of the problem is

$$\max Z = \sum_{j=1}^{n} c_j x_j,$$

subject to

$$\underline{b}_i \leq \sum_{j=1}^{n} a_{ij} x_j \leq \overline{b}_i, \qquad i = 1, \ldots, m,$$

$$x_j \in \{0, 1\}, \qquad j = 1, \ldots, n,$$

where $\underline{b}_i = -\infty$ for $\leq$-constraints, $\overline{b}_i = \infty$ for $\geq$-constrains, and $\underline{b}_i = \overline{b}_i$ for equality constraints.

Thus, for each constraint we store the values of $\underline{b}_i$ and $\overline{b}_i$, in addition to the current constraint values $L_i = \sum_{j=1}^{n} a_{ij}x_j$. The technology coefficients a_{ij} often represent a sparse matrix A. Therefore, it is inefficient to store the complete matrix, and instead we store, for a given constraint i, the following information:

- A vector of the indices j for which $a_{ij} \neq 0$.
- A vector of the values a_{ij} for the indices j in the previous vector.

We also store, for each variable j, the following information:

- A vector of the indices i for which $a_{ij} \neq 0$.
- A vector of the values a_{ij} for the indices i in the previous vector.

Define $\underline{M}(x) = \{i : \underline{b}_i > L_i\}$ as the set of constraints whose lower bound is violated, and define $\overline{M}(x) = \{i : \overline{b}_i < L_i\}$ as the set of constraints whose upper bound is violated. To facilitate the evaluation of neighbors, we consider normalized constraints. Let $\tilde{a}_i$ be the average absolute value of non-zero coefficients in constraint i. That is

$$\tilde{a}_i = \frac{\sum_{j=1}^{n} |a_{ij}|}{|\{j : a_{ij} \neq 0\}|}.$$

Then, the normalized constraints become

$$\underline{b}_i/\tilde{a}_i \leq L_i/\tilde{a}_i \leq \overline{b}_i/\tilde{a}_i,\ i = 1, \ldots, m.$$

3.2 Solution evaluation

To execute the VND, it must be clearly defined what is meant by stating that a solution x' is better than another solution x. Given a random starting solution, the choice here is to differentiate between solutions both based on their objective function values and their levels of infeasibility. A solution is better than another either if it has a better level of infeasibility, or it has the same level of infeasibility but a better objective function value. To express this formally,

for a given solution x, let the objective function value be denoted by

$$Z(x) = \sum_{j=1}^{n} c_j x_j.$$

The sum of constraint violations can be expressed as

$$V(x) = \sum_{i \in \underline{M}(x)} (\underline{b}_i - \sum_{j=1}^{n} a_{ij} x_j)/\tilde{a}_i + \sum_{i \in \overline{M}(x)} (\sum_{j=1}^{n} a_{ij} x_j - \overline{b}_i)/\tilde{a}_i.$$

In addition, the number of violated constraints is

$$W(x) = |\underline{M}(x)| + |\overline{M}(x)|.$$

We then create an overall measure of the infeasibility level as $A(x) = V(x) + \alpha W(x)$, where α is a parameter used to balance whether it is worse to violate few constraints by a large amount or many constraints by a small amount. Thus, if $W(x) = W(x') = 0$ and $Z(x) > Z(x')$, then x is a better solution than x'. Otherwise, if $W(x) > 0$, solution x is better than x' if $A(x) < A(x')$ or if both $A(x) = A(x')$ and $Z(x) > Z(x')$.

3.3 Neighborhood operators with simultaneous flips

Perhaps the most natural way to generate a hierarchy of neighborhoods for problems with only binary variables is to consider simultaneously flipping a given number $p = 1, 2, \ldots$ of variables. However, with increasing p, the size of the neighborhoods grows very quickly, with $n!/(p!(n-p)!)$ neighbors for an instance with n variables. To reduce the size of the resulting neighborhoods we also consider the sum of the variables that changes. That is, for a solution x, we consider neighborhoods $N_{pq}(x)$ consisting of flipping any p variables such that the sum of the values of the flipped variables is equal to either q or $-q$:

$$N_{pq}(x) = \{x' : \sum_{j=1}^{n} |x_j - x'_j| = p, |\sum_{j=1}^{n} (x_j - x'_j)| = q\}.$$

Table 1 lists the combinations of p and q that are considered in this paper. If we include neighborhoods with sufficiently large p, finding the optimal solution is theoretically guaranteed,

since the optimal solution must contain exactly k variables that are different from the current solution, with $0 \leq k \leq n$. However, given how the number of neighbors grow with p, it is unlikely that $p > 4$ can be used in practice. Already for $p > 2$, the size of the neighborhoods may become prohibitive for BIP instances with many variables. Although there are some simple strategies that can make neighborhood exploration more effective for higher p, such as using first improvement instead of best improvement, or by exploring the neighborhood only if the current solution is feasible, we also consider additional neighborhood reduction techniques.

Table 1: List of neighborhood operators considered for simultaneous flips of variables

p	q	Neighborhood description
1	1	Single-flip
2	0	Swap
2	2	Double-flip, but only with two variables from 1 to 0, or two from 0 to 1
3	1	Triple-flip, excluding moves flipping three variables from 1 to 0 or 0 to 1
3	3	Triple-flip, but only moves flipping three variables from 1 to 0 or 0 to 1
4	0	Double-swap
4	2	Quad-flip, but only with exactly three variables from 1 to 0 or vice versa
4	4	Quad-flip, but only with exactly four variables from 1 to 0 or vice versa

A basic idea to reduce the size of neighborhoods with $p > 1$ is based on considering a set of pairs $P \subseteq \{1, \ldots, n\} \times \{1, \ldots, n\}$ such that $(j, l) \in P$ if and only if there is a constraint i where $a_{ij} \neq 0$ and $a_{il} \neq 0$. That is, both variables j and l have non-zero coefficients in some constraint. We can then create reduced neighborhoods by considering

$$N_{pq}^{P}(x) = \{x' : \sum_{j=1}^{n} |x_j - x'_j| = p, |\sum_{j=1}^{n} (x_j - x'_j)| = q,$$
$$\forall j \exists l : l \neq j \wedge |x_j - x'_j + x_l - x'_l| = 2 \rightarrow (j, l) \in P\}.$$

That is, we only flip a pair of variables if they both appear together in at least one of the constraints. This basic idea can be taken further, by also considering that the variables should have opposite contributions to at least one of the constraint that they share. Let V be the set of variables to flip. Not only should we have $(j, l) \in P$ if $j, l \in V$, but we should also demand that there is at least one constraint where a variable in V has an opposite contribution to another variable in V: $\forall j \in V : \exists l \in V : \exists i : l \neq j \wedge (1 - 2x_j)(a_{ij})(1 - 2x_l)(a_{il}) < 0$. Note that $(1 - 2x_j)(a_{ij}) < 0$ if flipping variable x_j will decrease L_i, so that $(1 - 2x_j)(a_{ij})(1 - 2x_l)(a_{il}) < 0$ if and only if flipping x_j and flipping x_l have opposite effects on L_i. We can state this reduced

neighborhood as:

$$N^O_{pq}(x) = \{x' : \sum_{j=1}^{n} |x_j - x'_j| = p, |\sum_{j=1}^{n}(x_j - x'_j)| = q,$$
$$\forall j \exists l : l \neq j \wedge |x_j - x'_j + x_l - x'_l| = 2 \rightarrow \exists i : (1 - 2x_j)(a_{ij})(1 - 2x_l)(a_{il}) < 0\}.$$

Given these definitions, $N^O_{pq} \subseteq N^P_{pq} \subseteq N_{pq}$. Algorithm 3 provides pseudo-code for exploring the neighborhood N_{pq}. The pseudo-code provided assumes a best improvement search. It can be modified to a first improvement search by halting the while-loop as soon as a solution with $\Delta A > 0$ is found, or alternatively a solution with $\Delta A = 0$ and $\Delta Z > 0$.

Algorithm 3 Pseudo-code for exploring $N_{pq}(x)$.

Inputs: p, q, x, α
Initialize pointers to variables: $v(1) \leftarrow 1, v(2) \leftarrow 2, \ldots, v(p) \leftarrow p$
Calculate *currentW* and *currentV* as the infeasibility levels of x
bestmove $\leftarrow [v(1), v(2), \ldots, v(p)]$, *best*$\Delta Z \leftarrow -\infty$, *best*$\Delta W \leftarrow -\infty$, *best*$\Delta A \leftarrow -\infty$
if *currentW* $= 0$ **then**
  best$\Delta Z \leftarrow 0$, *best*$\Delta W \leftarrow 0$, *best*$\Delta A \leftarrow 0$
end if
while $v(p) \leq n$ **do**
  if $|\sum_{j=1}^{p}(2x_{v(j)} - 1)| = q$ **then**
    Calculate ΔZ as the change in objective function value when flipping $v(1), \ldots, v(p)$
    if *best*$\Delta W <$ *currentW* or (*best*$\Delta W =$ *currentW* and $\Delta Z >$ *best*ΔZ) **then**
      Calculate ΔV and ΔW for the current neighbor
      $\Delta A \leftarrow \Delta V + \alpha \Delta W$
      if $\Delta A >$ *best*ΔA or ($\Delta A =$ *best*ΔA and $\Delta Z >$ *best*ΔZ) **then**
        bestmove $\leftarrow [v(1), v(2), \ldots, v(p)]$
        best$\Delta Z \leftarrow \Delta Z$, *best*$\Delta W \leftarrow \Delta W$, *best*$\Delta A \leftarrow \Delta A$
      end if
    end if
  end if
  Move to next neighbor using Algorithm 4
end while
Output: *bestmove*, *best*ΔZ, *best*ΔW, *best*ΔA.

The exploration of $N_{pq}^P(x)$ and $N_{pq}^O(x)$ can be done very similarly to Algorithm 3, but with an added step just before Step 10. For $N_{pq}^P(x)$, an n by n matrix is precalculated, indicating whether or not variables j and l appear in any of the same constraints. Then, calculations can be skipped if $p > 1$ and for any variable involved in the move, there are no other variables flipped that appear in any of the same constraints. For $N_{pq}^O(x)$ two n by n matrices are precalculated: one for the case that x_j and x_l have the same value, and one for the case that x_j and x_l have different values in the current solution. Then, the matrices indicate whether or not there exists a constraint i with both x_j and x_l appearing, such that flipping x_j and x_l has opposite effects on L_i, given their current values. If $p > 1$ and there is a variable to be flipped that does not have any such opposite variable also being flipped, the move is skipped.

Algorithm 4 Find next neighbor.

1: Inputs: n, p, $v(1), v(2), \ldots, v(p)$.
2: $index = p$
3: **while** $index > 0$ **do**
4: $\quad v(index) \leftarrow v(index) + 1$
5: $\quad$ **for** $subindex = index + 1, \ldots, p$ **do**
6: $\quad\quad v(subindex) \leftarrow v(subindex - 1) + 1$
7: $\quad$ **end for**
8: $\quad$ **if** $v(p) > n$ and $index > 1$ **then**
9: $\quad\quad index \leftarrow index - 1$
10: $\quad$ **else**
11: $\quad\quad index = 0$
12: $\quad$ **end if**
13: **end while**

3.4 Neighborhood operators with sequential flips

In a second hierarchy of neighborhoods, some flipped variables are determined sequentially. For a given value of $r = 1, 2, \ldots$, a neighborhood $N_r(x)$ of x is defined consisting of $n - r + 1$ neighbors, where each neighbor is determined by first flipping r variables, and then sequentially considering each of the remaining variables and deciding whether or not to also flip each of them. First, sort variables according to $c_j(1 - 2x_j)$ in decreasing order. That is, in order of worse effect on the objective function if the variable is flipped. In the case of instances where $c_j = c_l$ for a large number of variables $j \neq l$, we use $c_j(1 - 2x_j) + \sum_{i=1}^m |a_{ij}/(m\tilde{a}_i)|$ as the sorting criterion,

which makes the magnitude of the technology coefficients a tie-breaker if two variables have equal objective function coefficients. Let $s(t)$ be the variable in position t of the sorted list, $t = 1, \ldots, n$.

For $r = 1, 2, \ldots$ we can now define a neighborhood N_r containing $n - r + 1$ solutions. Move number t can be described as first flipping variables $s(t), \ldots, s(t+r-1)$, which implies a total of r flips. Then, the variables $s(1), \ldots, s(t-1), s(t+r), \ldots, s(n)$ are considered sequentially. Each of them is flipped, in addition to any previously considered flips, if the additional flip myopically improves the solution implied by the previous set of flips.

Following the idea of N^O_{pq}, we speed up the neighborhood exploration by enforcing that the additionally flipped variables appear in at least one constraint together with one of the previously flipped variables, and that they also have opposite effects on at least one constraint. We refer to these variants of N_r as N^P_r and N^O_r, and they are implemented by checking the corresponding matrices before Step 10 of Algorithm 5.

Algorithm 5 Pseudo-code for exploring $N_r(x)$.

Inputs: r, x, α
Calculate a sorted list of variables, $s(1), \ldots, s(n)$
Calculate *currentW* and *currentV* as the infeasibility levels of x
bestmove $\leftarrow [v(1), v(2), \ldots, v(p)]$, *best*$\Delta Z \leftarrow -\infty$, *best*$\Delta W \leftarrow -\infty$, *best*$\Delta A \leftarrow -\infty$
for $t = 1, \ldots, n - r + 1$ **do**
  currentmove $\leftarrow [s(t), s(t+1), \ldots, s(t+r-1)]$
  Calculate ΔZ, ΔW, and ΔA for *currentmove*
  for $t' = 1, \ldots, n$ **do**
    if $s(t')$ not flipped in *currentmove* **then**
      if Adding $s(t')$ to *currentmove* leads to a better solution **then**
        Add $s(t')$ to *currentmove*
        Update ΔZ, ΔW, and ΔA for *currentmove*
      end if
    end if
  end for
  if $\Delta A >$ *best*ΔA or ($\Delta A =$ *best*ΔA and $\Delta Z >$ *best*ΔZ) **then**
    bestmove $\leftarrow$ *currentmove*
    best$\Delta Z \leftarrow \Delta Z$, *best*$\Delta W \leftarrow \Delta W$, *best*$\Delta A \leftarrow \Delta A$
  end if
end for
Output: *bestmove*, *best*ΔZ, *best*ΔW, *best*ΔA.

4 Results and discussion

To evaluate whether VNS is a viable strategy for solving the BIP, the VND and the GVNS were implemented in C++. Computational tests on instances based on the OptSAT, the MDMKP, and the MMKP were run using a PC with a 64 bit 3.5GHz i5-4690 CPU with 16 GB RAM, running Windows 10. For comparisons, the same instances were also solved using CPLEX version 12.10 and LocalSolver version 10.0. CPLEX is an exact mixed-integer programming solver using a branch-and-cut procedure (CPLEX, 2020), while LocalSolver is primarily based on a heuristic local search (Benoist et al., 2011), with a recent addition of the ability to find dual bounds (Boulmier, 2020). A third benchmark method is also used, consisting of a best-improvement local search (BILS) using the 1-flip neighborhood and restarting with new random solutions

when a local optimum has been reached.

In the computational testing we follow the protocol of selecting a training set of instances for tuning parameters of the heuristics and a test set of instances for evaluating the final settings. These sets of instances are non-overlapping, and are selected to be representative of the problem classes considered. While the results on the test set provide a basis for truthfully assessing the performance of the VND and the GVNS, we also provide results for an additional set of instances, as well as with additional parameter settings, to further highlight the benefits and the disadvantages of the VNS methodology for solving BIPs.

4.1 Instances

For the OptSAT we consider test instances created by Davoine et al. (2003) and da Silva et al. (2020), with the largest instances having up to $n = 3,000$ variables and $m = 15,000$ constraints (clauses). The instances typically have few non-zero coefficients in each constraint, leading to a sparse constraint matrix where the non-zero coefficients are all either 1 or -1. It is typically easy to find feasible solutions for these instances, but commercial mixed integer programming solvers are currently unable to solve the most difficult instances to proven optimality (da Silva et al., 2020; Hvattum et al., 2012).

Cappanera and Trubian (2005) generated test instances for the MDMKP. These instances have up to $n = 500$ variables, $m = 30$ knapsack constraints, and $q = 30$ covering constraints. A particular aspect of these instances is that the constraint matrix is full, that is, a_{ij} is strictly greater than 0 for all i and all j. For some of the instances, in particular those with many constraints and few variables, commercial mixed integer programming solvers may struggle to find any feasible solutions, as reported by Arntzen et al. (2006).

Test instances for the MMKP were presented by Khan et al. (2002) and Shojaei et al. (2013). However, some of these instances contain non-integer objective function coefficients, which we converted into integers by multiplying all coefficients by 10 until integrality was obtained. Although some of the a_{ijk} coefficients are zero, the part of the constraint matrix corresponding to the knapsack constraints is almost full.

To tune parameters of the VND and GVNS, we select a training set of 15 instances, five from each of the three problem classes used. For OptSAT, five instances generated by Davoine et al. (2003) are selected: rn200m1000t10s0c25num0 (class 27), rn500m1000t25s0c25num0 (class 29), rn500m2500t25s0c25num0 (class 31), rn500m2500t25s0c75num0 (class 49), and rn500m1000t25s0c0num0

(class 61). The five instances based on the MDMKP were created by Cappanera and Trubian (2005), and labeled 100-5-1-0-0 (class 1), 500-5-5-0-0 (class 3), 250-10-10-1-0 (class 5), 100-30-30-1-0 (class 7), and 500-30-1-0-0 (class 9). For the MMKP, the training set contains I3 and INST15 provided by Khan et al. (2002) and INST23, INST27, and RTI11 provided by Shojaei et al. (2013).

The evaluation of the final methods is done on a test set, which consists of 15 instances for each of the three problems. The names of the selected instances, as well as the number of variables, constraints, and non-zero elements of the constraint matrix, are given in Tables 2–4. The instances are selected manually, with the aim of obtaining a diverse set of instances from each problem class.

Table 2: Instances of OptSAT included in the test set.

Instance	Name	n	m	Non-zeros	Source
01	lmhn1000m5000num1	1000	5000	15000	(da Silva et al., 2020)
02	lmhn1500m7500num1	1500	7500	22500	(da Silva et al., 2020)
03	qn500m2500t2s0c0num0	500	2500	5000	(Davoine et al., 2003)
04	qn500m5000t2s0c0num0	500	5000	10000	(Davoine et al., 2003)
05	qn1000m10000t2s0c0num0	1000	10000	20000	(Davoine et al., 2003)
06	rn200m1000t10s0c0num0	200	1000	10000	(Davoine et al., 2003)
07	rn200m1000t10s0c25num4	200	1000	10000	(Davoine et al., 2003)
08	rn200m1000t40s20c0num0	200	1000	40187	(Davoine et al., 2003)
09	rn500m1000t25s0c0num4	500	1000	25000	(Davoine et al., 2003)
10	rn500m1000t25s0c25num4	500	1000	25000	(Davoine et al., 2003)
11	rn500m1000t25s0c50num0	500	1000	25000	(Davoine et al., 2003)
12	rn500m1000t100s50c0num0	500	1000	99511	(Davoine et al., 2003)
13	rn500m1000t100s50c25num0	500	1000	100798	(Davoine et al., 2003)
14	rn500m2500t25s0c25num4	500	2500	62500	(Davoine et al., 2003)
15	rn500m2500t25s0c50num0	500	2500	62500	(Davoine et al., 2003)

4.2 Results on the training set

The purpose of the tests on the training set is two-fold. First, we want to understand how the different neighborhoods behave on different types of instances. Second, we want to arrive at parameter settings for a VND and a GVNS that can be evaluated on the test set. To this end, a sequence of exploratory tests were executed. The first test included all neighborhoods N_{pq} listed in Table 1, with the goal of examining the relative usefulness of each neighborhood. Each run was limited to 1,800 seconds, using $\alpha = 1$ and best improvement local search, and statistics

Table 3: Instances of MDMKP included in the test set.

Instance	Name	n	m	Non-zeros	Source
16	100-5-5-1-0	100	10	1000	(Cappanera and Trubian, 2005)
17	100-10-5-1-0	100	15	1500	(Cappanera and Trubian, 2005)
18	100-30-15-1-10	100	45	4500	(Cappanera and Trubian, 2005)
19	100-30-30-0-1	100	60	6000	(Cappanera and Trubian, 2005)
20	100-50-10-1	100	51	5100	(Cappanera and Trubian, 2005)
21	100-50-q-1	100	51	5100	(Cappanera and Trubian, 2005)
22	100-100-25-1	100	101	10100	(Cappanera and Trubian, 2005)
23	100-100-q-1	100	101	10100	(Cappanera and Trubian, 2005)
24	250-5-2-0-0	250	7	1750	(Cappanera and Trubian, 2005)
25	250-10-1-0-14	250	11	2750	(Cappanera and Trubian, 2005)
26	250-30-30-0-0	250	60	15000	(Cappanera and Trubian, 2005)
27	500-5-5-0-14	500	10	5000	(Cappanera and Trubian, 2005)
28	500-10-10-1-0	500	20	10000	(Cappanera and Trubian, 2005)
29	500-30-15-1-0	500	45	22500	(Cappanera and Trubian, 2005)
30	500-30-30-0-0	500	60	30000	(Cappanera and Trubian, 2005)

Table 4: Instances of MMKP included in the test set.

Instance	Name	n	m	Non-zeros	Source
31	I5	250	35	2508	(Khan et al., 2002)
32	I9	2000	210	20009	(Khan et al., 2002)
33	I11	3000	310	30027	(Khan et al., 2002)
34	I13	4000	410	40050	(Khan et al., 2002)
35	INST01	500	60	5022	(Khan et al., 2002)
36	INST03	600	70	6024	(Khan et al., 2002)
37	INST07	800	90	8009	(Khan et al., 2002)
38	INST18	5600	290	61600	(Khan et al., 2002)
39	INST20	7000	360	77000	(Khan et al., 2002)
40	INST21	1076	210	10763	(Shojaei et al., 2013)
41	INST24	584	140	21675	(Shojaei et al., 2013)
42	INST28	1643	310	16439	(Shojaei et al., 2013)
43	RTI09	158	40	1568	(Shojaei et al., 2013)
44	RTI12	241	50	2448	(Shojaei et al., 2013)
45	RTI13	295	60	2954	(Shojaei et al., 2013)

regarding neighborhoods explored were collected during the runs.

Table 5 summarizes the behavior of the VND with all eight N_{pq} neighborhoods for the three different problem classes. The table reports the number of times each neighborhood is explored, the amount of time spent in each neighborhood, and the number of improvements found. Results are given as averages over all training instances, and using relative measures. The same test was also run using N^P_{pq} and N^O_{pq}, with an almost identical relative performance observed. Looking at the best solutions found, N^O_{pq} performs better than N_{pq} and N^P_{pq}, in that it either finds a better solution than the other two, or that the same best solution is found earlier. Certain neighborhoods very rarely lead to improvements. These are therefore eliminated in subsequent tests: $(p, q) \in \{(2, 2), (3, 3), (4, 2), (4, 4)\}$. The remaining large neighborhoods, with $p > 2$ consume a large portion of the running time, but leads to some improved solutions being found.

Subsequent tests therefore examined whether the large neighborhoods can be explored in an efficient manner by using a first improvement search, by exploring them only when the current solution is feasible, or by exploring them only for instances with relatively few variables. Using first improvement instead of best improvement for $p > 2$ leads to relatively less time spent in these neighborhoods, while finding relatively more improvements. Overall, the results found after 1,800 seconds are better. Further testing revealed that it is better to skip the largest neighborhoods, with $p > 2$, for large instances with $n \geq 600$ or $m \geq 100$. Attempting to further reduce running times by only exploring the larger neighborhoods when the current solution is feasible did not improve the results.

Table 5: Using the full set of neighborhoods N_{pq} from Table 1, running each instance for 1,800 seconds, and reporting the relative number of times each neighborhood is explored (C), the relative amount of time spent in each neighborhood (T), and the relative number of improvements found by each neighborhood (I). Numbers are averages over five instances in each problem class.

		MDMKP			MMKP			OptSAT		
p	q	C [%]	I [%]	T [%]	C [%]	I [%]	T [%]	C [%]	I [%]	T [%]
1	1	58.0	64.6	0.0	83.8	90.8	0.0	77.2	82.8	0.0
2	0	20.7	26.8	0.1	7.4	6.7	0.1	13.0	13.5	0.1
2	2	5.4	0.0	0.1	2.2	0.0	0.1	3.2	0.0	0.0
3	1	5.4	1.7	3.5	2.2	0.5	9.4	3.2	2.7	5.1
3	3	4.5	0.0	3.0	1.8	0.0	14.5	1.3	0.0	0.5
4	0	4.5	7.0	54.1	1.8	1.9	30.9	1.3	1.1	51.8
4	2	0.7	0.0	21.3	0.4	0.0	19.0	0.4	0.0	30.1
4	4	0.7	0.0	17.8	0.4	0.0	25.9	0.4	0.0	12.3

Table 6: Using a set of neighborhoods N_r with $r = 1, \ldots, 8$, running each instance for 1,800 seconds, and reporting the relative number of times each neighborhood is explored (C), the relative amount of time spent in each neighborhood (T), and the relative number of improvements found by each neighborhood (I). Numbers are averages over five instances in each problem class.

	MDMKP			MMKP			OptSAT		
r	C [%]	I [%]	T [%]	C [%]	I [%]	T [%]	C [%]	I [%]	T [%]
1	67.2	97.6	68.6	59.7	93.5	60.5	64.8	98.0	65.4
2	5.8	1.6	5.7	7.8	2.6	8.3	5.9	1.3	5.1
3	4.8	0.4	4.7	6.6	1.4	6.8	5.1	0.3	4.7
4	4.6	0.2	4.4	5.9	0.8	5.8	4.9	0.1	4.7
5	4.5	0.1	4.3	5.4	0.6	5.2	4.9	0.1	4.8
6	4.4	0.0	4.2	5.1	0.5	4.8	4.8	0.1	5.0
7	4.4	0.0	4.1	4.9	0.4	4.4	4.8	0.0	5.1
8	4.3	0.0	4.1	4.6	0.3	4.1	4.8	0.0	5.2

For the second hierarchy of neighborhoods, N_r, Table 6 shows the relative performance of neighborhoods with $r = 1, \ldots, 8$. These sequential-flip neighborhoods behave very differently from the neighborhoods based exclusively on simultaneous flips. The relative time used by each neighborhood is almost identical to the number of times the neighborhood is explored. The relative number of improvements found declines with increasing r, but this may partly be due to the sequence in which the neighborhoods are explored. Applying the techniques of neighborhood reduction is also found useful for this hierarchy, with N_r^O performing better than N_r. The neighborhoods are all explored relatively quickly, with similar complexity independent of r. The results indicate that the marginal improvement of additional neighborhoods by increasing r is quickly reduced. As for the large neighborhoods of N_{pq}^O, results are improved when using first improvement to explore the N_r^O neighborhoods.

Having analyzed the two neighborhood types separately, the next experiments were designed to find a joint set of neighborhoods that provides the best performance on the training instances. The final settings were decided by a process of manually exploring different alternatives and evaluating their average performance on the instances of the training set. Table 7 shows the final list of neighborhoods used in the VND, which includes further restrictions on the use of neighborhoods based on the size of the instance solved. In particular, the N_{pq}^O neighborhoods with $(p, q) = (3, 1)$ or $(4, 0)$ are not used when $n \geq 600$ or $m \geq 100$.

For the GVNS, the same settings are used for the VND, but in addition a set of neighborhoods for the shaking must be determined. Two options were tested, either using N_r for shaking, or

Table 7: Final settings for neighborhoods used in the VND.

k	type	size	exploration strategy	requirements
1	N^O_{pq}	$p = 1, q = 1$	best improvement	none
2	N^O_{pq}	$p = 2, q = 0$	best improvement	none
3	N^O_r	$r = 1$	first improvement	none
4	N^O_r	$r = 2$	first improvement	none
5	N^O_{pq}	$p = 3, q = 1$	first improvement	$n < 600$, $m < 100$
6	N^O_{pq}	$p = 4, q = 0$	first improvement	$n < 600$, $m < 100$

using $\bigcup_{q=0}^{p} N_p$. The latter, being the same as N_{pq} but without any restrictions on q, turned out to be the better choice, with $k^{MAX} = 16$ different neighborhoods from $p \in \{5, 6, \ldots, 20\}$. One further modification of the GVNS was made, wherein the VND used as a subroutine is halted immediately if the current solution becomes equal to the solution before shaking. For both the VND and the GVNS, the solution evaluation function $A(x) = V(x) + \alpha W(x)$ is found to work well with $\alpha = 1$, although the performance is not sensitive to the choice of α.

4.3 Results on the test set

This section presents results on the test set by using CPLEX 12.10, LocalSolver 10.0, BILS, the VND, and the GVNS. Each instance is solved using each method and two different time limits. Short runs are considered with a one minute time limit, and long runs are allowed a time limit of one hour. For the short runs, all methods except CPLEX are run 11 times and the median result is reported for each instance. When reporting the results, the objective function is reported relative to the best value found within all results obtained for each instance. When a method fails to find any feasible solutions within the time limit, the result is recorded as minus infinity. Results are grouped by the type of problem solved (OptSAT, MDMKP, MMKP), and for each group we report the minimum, median, and maximum performance over the 15 instances in the group, as well as the interquartile range (IQR).

The results for OptSAT are given in Table 8 and Table 9. For these instances, LocalSolver performs on average better than CPLEX, while the performance of GVNS is similar to that of the VND for the long runs. Comparing LocalSolver and GVNS is not entirely straightforward: there are three instances where the performance of GVNS and VND is relatively poor. These instances (01, 02, and 05) have the highest number of constraints among the OptSAT instances in the test set. Without these instances, the performance of GVNS would be dominating LocalSolver, as

seen when comparing the median performance over the 15 instances. The simple BILS benchmark method is the worst method on all instances, but is relatively close to the best solutions on some instances.

Table 8: Results for short runs (60 seconds) for instances of the OptSAT, reporting the objective function values relative to the best objective function value found in any test run. The best performances for each instance are highlighted in bold. CPLEX is run only once, whereas for the other methods the median result of 11 individual runs is given.

Instance	CPLEX	LocalSolver	BILS	VND	GVNS
01	0.910	**0.966**	0.867	0.941	0.946
02	0.933	**0.954**	0.865	0.939	0.945
03	0.975	**0.991**	0.930	0.978	0.978
04	0.967	0.975	0.900	0.970	**0.977**
05	**0.974**	0.974	0.863	0.947	0.941
06	0.996	0.995	0.956	0.995	**1.000**
07	0.994	0.992	0.969	0.996	**1.000**
08	1.000	0.997	0.996	**1.000**	**1.000**
09	0.997	0.995	0.988	0.998	**0.998**
10	0.999	0.997	0.992	0.998	**0.999**
11	0.998	0.998	0.992	0.999	**0.999**
12	0.999	0.999	0.998	**1.000**	**1.000**
13	1.000	0.999	0.998	**1.000**	**1.000**
14	0.996	0.990	0.986	0.998	**0.999**
15	0.995	0.991	0.987	0.997	**0.998**
Min	0.910	**0.954**	0.863	0.939	0.941
Median	0.996	0.992	0.986	0.997	**0.999**
Max	1.000	0.999	0.998	1.000	**1.000**
IQR	0.024	0.015	0.077	0.024	0.023

Table 10 and Table 11 provide the results for the 15 MDMKP instances in the test set. The performance of CPLEX is noteworthy, as it finds the best solution on all instances except two in the long runs. In the two cases where CPLEX does not perform well, there is one instance where no feasible solution is found, and one instance where the solution found is poor. Thus, CPLEX performs best on instances with 250 or more variables, whereas the best performance on the instances with 100 variables is by the GVNS, followed by the VND. Instance 19, where both CPLEX and LocalSolver has a poor performance, comes from a class of 30 instances with 100 variables and 60 constraints, and Section 4.4 contains additional results for these instances. BILS again has the worst performance on all the instances, and is relatively far away from the best found solutions on all instances.

Table 9: Results for long runs (3600 seconds) for instances of the OptSAT, reporting the objective function values relative to the best objective function value found in any test run. The best performances for each instance are highlighted in bold. Each method is run only once.

Instance	CPLEX	LocalSolver	BILS	VND	GVNS
01	0.980	**1.000**	0.868	0.954	0.955
02	0.984	**1.000**	0.869	0.955	0.950
03	**1.000**	**1.000**	0.930	0.992	0.992
04	0.998	**1.000**	0.900	0.998	1.000
05	**1.000**	0.992	0.860	0.969	0.964
06	0.998	**1.000**	0.961	**1.000**	**1.000**
07	**1.000**	0.998	0.962	**1.000**	**1.000**
08	**1.000**	1.000	0.996	**1.000**	**1.000**
09	1.000	0.997	0.988	0.999	**1.000**
10	0.999	0.998	0.992	**1.000**	**1.000**
11	0.999	0.999	0.992	**1.000**	**1.000**
12	**1.000**	1.000	0.999	**1.000**	**1.000**
13	**1.000**	1.000	0.999	**1.000**	**1.000**
14	0.998	0.998	0.986	1.000	**1.000**
15	0.999	0.997	0.988	0.999	**1.000**
Min	0.980	**0.992**	0.860	0.954	0.950
Median	0.999	1.000	0.986	1.000	**1.000**
Max	**1.000**	**1.000**	0.999	**1.000**	**1.000**
IQR	0.002	0.002	0.077	0.005	0.004

Table 10: Results for short runs (60 seconds) for instances of the MDMKP, reporting the objective function values relative to the best objective function value found in any test run. The best performances for each instance are highlighted in bold. CPLEX is run only once, whereas for the other methods the median result of 11 individual runs is given.

Instance	CPLEX	LocalSolver	BILS	VND	GVNS
16	**1.000**	**1.000**	0.805	0.995	0.990
17	**1.000**	0.990	0.788	0.990	0.991
18	**0.968**	0.881	$-\infty$	0.947	0.955
19	$-\infty$	0.898	$-\infty$	0.945	**0.959**
20	**1.000**	**1.000**	0.862	**1.000**	**1.000**
21	**1.000**	**1.000**	0.807	**1.000**	**1.000**
22	**1.000**	**1.000**	0.921	**1.000**	**1.000**
23	**1.000**	0.965	0.852	0.998	0.998
24	**1.000**	**1.000**	0.882	0.993	0.993
25	**1.000**	1.000	0.916	0.992	0.992
26	0.958	0.934	0.762	0.956	**0.965**
27	**1.000**	1.000	0.905	0.996	0.997
28	**0.996**	0.983	0.754	0.939	0.943
29	**0.978**	0.919	0.651	0.912	0.908
30	**0.946**	0.945	0.792	0.944	0.945
Min	$-\infty$	0.881	$-\infty$	**0.912**	0.908
Median	**1.000**	0.990	0.805	0.992	0.991
Max	**1.000**	**1.000**	0.921	**1.000**	**1.000**
IQR	0.027	0.060	0.114	0.051	0.041

Table 11: Results for long runs (3600 seconds) for instances of the MDMKP, reporting the objective function values relative to the best objective function value found in any test run. The best performances for each instance are highlighted in bold. Each method is run only once.

Instance	CPLEX	LocalSolver	BILS	VND	GVNS
16	**1.000**	**1.000**	0.805	**1.000**	**1.000**
17	**1.000**	**1.000**	0.788	**1.000**	**1.000**
18	0.975	0.974	$-\infty$	0.977	**1.000**
19	$-\infty$	0.902	$-\infty$	0.988	**1.000**
20	**1.000**	**1.000**	0.862	**1.000**	**1.000**
21	**1.000**	**1.000**	0.807	**1.000**	**1.000**
22	**1.000**	**1.000**	0.921	**1.000**	**1.000**
23	**1.000**	0.966	0.852	**1.000**	**1.000**
24	**1.000**	**1.000**	0.882	0.997	0.997
25	**1.000**	**1.000**	0.916	0.994	0.997
26	**1.000**	0.951	0.780	0.988	0.997
27	**1.000**	1.000	0.905	0.998	0.999
28	**1.000**	0.990	0.786	0.977	0.986
29	**1.000**	0.938	0.663	0.962	0.981
30	**1.000**	0.983	0.800	0.992	0.998
Min	$-\infty$	0.902	$-\infty$	0.962	0.981
Median	**1.000**	1.000	0.805	0.997	**1.000**
Max	**1.000**	**1.000**	0.921	**1.000**	**1.000**
IQR	0.000	0.030	0.089	0.012	0.003

Finally, the results for MMKP are provided in Table 12 and Table 13. For these instances, CPLEX provides the best performance, followed by LocalSolver. The GVNS performs better than the VND, but is only able to find the best solution on two of the 15 instances. The BILS benchmark performs poorly on most of the MMKP instances. For one instance, both GVNS and VND fails to find any feasible solution. Instance 41 is not particularly large, with 584 variables and 140 constraints, compared to the other instances of this class, and the performance on this instance is examined further in Section 4.4.

Table 12: Results for short runs (60 seconds) for instances of the MMKP, reporting the objective function values relative to the best objective function value found in any test run. The best performances for each instance are highlighted in bold. CPLEX is run only once, whereas for the other methods the median result of 11 individual runs is given.

Instance	CPLEX	LocalSolver	BILS	VND	GVNS
31	**1.000**	**1.000**	0.580	**1.000**	**1.000**
32	**1.000**	0.999	0.715	0.968	0.974
33	**1.000**	0.999	0.714	0.965	0.971
34	**1.000**	1.000	0.712	0.964	0.968
35	0.998	**0.999**	0.871	0.955	0.962
36	**0.999**	**0.999**	0.930	0.962	0.969
37	0.998	**0.998**	0.877	0.960	0.971
38	**1.000**	**1.000**	0.603	0.940	0.952
39	**1.000**	1.000	0.597	0.941	0.944
40	**1.000**	0.999	0.903	0.972	0.977
41	**0.997**	0.978	$-\infty$	$-\infty$	$-\infty$
42	**1.000**	0.999	0.875	0.971	0.977
43	**1.000**	**1.000**	0.728	1.000	1.000
44	**1.000**	0.998	0.931	0.956	0.961
45	**1.000**	**1.000**	0.665	0.948	0.968
Min	**0.997**	0.978	$-\infty$	$-\infty$	$-\infty$
Median	**1.000**	0.999	0.715	0.962	0.969
Max	**1.000**	**1.000**	0.931	**1.000**	**1.000**
IQR	0.001	0.001	0.243	0.017	0.014

While the presented results focus on the primal bounds obtained by the methods, both CPLEX and LocalSolver also provide dual bounds. Given sufficient running time, these methods can provide proofs of optimality. For long runs, CPLEX is able to prove that an optimal solution is found on 12 of the 45 instances tested: 16, 17, 20–24, 31, 34, and 43–45, while LocalSolver is able to prove optimality for eight of these instances. With a short time limit, the methods are able to provide proofs of optimality for seven and three of the instances, respectively.

Table 13: Results for long runs (3600 seconds) for instances of the MMKP, reporting the objective function values relative to the best objective function value found in any test run. The best performances for each instance are highlighted in bold. Each method is run only once.

Instance	CPLEX	LocalSolver	BILS	VND	GVNS
31	**1.000**	**1.000**	0.593	**1.000**	**1.000**
32	**1.000**	1.000	0.726	0.970	0.978
33	**1.000**	1.000	0.720	0.969	0.977
34	**1.000**	1.000	0.717	0.968	0.974
35	**1.000**	0.999	0.881	0.969	0.978
36	**1.000**	0.999	0.934	0.967	0.973
37	**1.000**	**1.000**	0.879	0.964	0.977
38	**1.000**	1.000	0.607	0.945	0.975
39	**1.000**	1.000	0.602	0.945	0.977
40	**1.000**	1.000	0.911	0.974	0.982
41	**1.000**	0.980	$-\infty$	$-\infty$	$-\infty$
42	**1.000**	1.000	0.883	0.973	0.981
43	**1.000**	**1.000**	0.728	**1.000**	**1.000**
44	**1.000**	**1.000**	0.947	0.974	0.985
45	**1.000**	**1.000**	0.675	0.987	0.994
Min	**1.000**	0.980	$-\infty$	$-\infty$	$-\infty$
Median	**1.000**	1.000	0.726	0.969	0.978
Max	**1.000**	**1.000**	0.947	**1.000**	**1.000**
IQR	0.000	0.000	0.241	0.008	0.008

In summary, the results on the test set are somewhat mixed. Compared to the commercially available solvers, CPLEX and LocalSolver, the new VNS methods perform well on certain types of instances, and less well on others. For OptSAT, GVNS is arguably producing the best results, followed by LocalSolver. However, on OptSAT instances with many constraints, this result is reversed. For MDMKP, GVNS is the best method on instances with fewer variables, whereas CPLEX is best on instances with more variables. On the MMKP, CPLEX is the best method, followed by LocalSolver. The heuristic commercial solver, LocalSolver, is the only method to find feasible solutions to all the instances in the test set, but additional tests reported in Section 4.4 provide a more nuanced conclusion.

Grouping the 45 instances in the test set together and sorting them by the number of variables leads to some general conclusions regarding the GVNS. When considering the long runs of 60 minutes, GVNS finds the best solution to all instances with $n \leq 200$. No other method is able to do this, as CPLEX and LocalSolver fail on three instances (06, 18, and 19) and VND on two (18 and 19). On the other hand, GVNS does not find the best solution for any instance with $n > 500$. Sorting the instances by the number of constraints or non-zeros does not lead to any similar patterns: GVNS finds the best solution for the two instances with the most non-zeros and the three instances with the fewest. In other words, it fails to find the best solution for the instances with the third most and the fourth fewest non-zeros, among the 45 instances. In addition, GVNS fails to find the best solution for both the instances with most ($m = 10,000$) and the instances with fewest ($m = 7$) constraints.

We end the comparison of the different methods by testing whether the differences in their performances are statistically significant. To this end we apply a sign-test, where we count the number of instances for which the performance is strictly better or strictly worse for each pair of methods, while ignoring instances where the performances are equal. The null-hypothesis is that there is no difference in performance, which leads to an assumption that the distribution of better and worse results is a binomial distribution with an equal probability for each outcome. In these tests we consider results from all 45 instances in the test set and note that for comparisons based on short runs, the median result for each instance is used, with all methods being run 11 times, except for CPLEX which is only run once.

Table 14 and Table 15 shows results from the sign tests on short and long runs, respectively. Methods highlighted in bold are significantly better than the opposing method when using a significance level of 0.05 and a two-sided probability. It is seen that CPLEX is significantly better than any other method, except the GVNS on short runs. LocalSolver is not significantly better than VND or GVNS. GVNS is significantly better than VND on both short and long runs,

and all other methods are better than BILS for both time limits.

Table 14: Statistical comparison of performances on short runs.

Method 1	Method 2	Method 1 wins	Method 2 wins	P-value
CPLEX	LocalSolver	28	7	0.001
CPLEX	BILS	44	0	<0.001
CPLEX	VND	27	13	0.038
CPLEX	GVNS	24	16	0.268
LocalSolver	BILS	45	0	<0.001
LocalSolver	VND	25	14	0.108
LocalSolver	GVNS	24	16	0.268
BILS	**VND**	0	44	<0.001
BILS	**GVNS**	0	44	<0.001
VND	**GVNS**	4	31	<0.001

Table 15: Statistical comparison of performances on long runs.

Method 1	Method 2	Method 1 wins	Method 2 wins	P-value
CPLEX	LocalSolver	27	5	<0.001
CPLEX	BILS	44	0	<0.001
CPLEX	VND	26	7	0.001
CPLEX	GVNS	24	9	0.014
LocalSolver	BILS	45	0	<0.001
LocalSolver	VND	22	15	0.324
LocalSolver	GVNS	22	15	0.324
BILS	**VND**	0	44	<0.001
BILS	**GVNS**	0	44	<0.001
VND	**GVNS**	4	25	<0.001

4.4 Additional results

Having observed the performance of the methods on the test instances, additional experiments were conducted to shed further light on the performance of the GVNS and the VND. First, we observed that CPLEX and LocalSolver performed poorly on one MDMKP instance with 100 variables, 30 knapsack constraints, and 30 cover constraints. In the original set of instances (Cappanera and Trubian, 2005), a set of 30 randomly generated instances of the same size are present. Table 16 shows the results for all of these 30 instances, out of which 100-30-30-1-0 was included in the training set and 100-30-30-0-1 in the test set.

The results indicate that the performance of GVNS and VND on instance 19 in the test set was not happenstance. The GVNS finds the best solution on 25 of the 30 instances, and the VND finds feasible solutions to all of the 30 instances. In comparison, CPLEX fails to find any feasible solution on 22 out of 30 instances, while LocalSolver fails to find feasible solutions on four of the instances. In addition, when LocalSolver does find feasible solutions, they are typically of much worse quality than the solutions found by GVNS and VND. The results of BILS are omitted from the table: running 11 times with different random seeds for the short time limit and once for the long time limit, the BILS was not able to find any feasible solution to any of the instances in any of the runs.

Second, the performance of the VND and the GVNS on the MMKP instances is worthy of further investigation. On instance 41, both of these methods fail to find any feasible solution after a full 60 minutes of running time. When starting from a random solution, the VND navigates towards a solution where the knapsack constraints are satisfied, but a number of the set partitioning constraints of the form $\sum_{j \in G_i} x_{ij} = 1$ are violated, with the left hand side being equal to 0. Changing any of the variables on the left hand side of these violated constraints from 0 to 1 leads to the violation of several knapsack constraints. Therefore, the moves considered are unable to find a path to a feasible solution. In addition, for those instances where a feasible solution is found, the VND does not easily find an improved feasible solution.

In an attempt to improve the performance of the GVNS on the MMKP, we introduced an alternative evaluation function, where each constraint has a separate weight. That is, with $w_i \geq 1$ being the weight of constraint $i = 1, \ldots, m$, the weighted sum of constraint violations is $V(x) = \sum_{i \in \underline{M}(x)} w_i(\underline{b}_i - \sum_{j=1}^{n} a_{ij}x_j)/\tilde{a}_i + \sum_{i \in \overline{M}(x)} w_i(\sum_{j=1}^{n} a_{ij}x_j - \overline{b}_i)/\tilde{a}_i$, and the weighted number of violated constraints is $W(x) = \sum_{i \in \underline{M}(x)} w_i + \sum_{i \in \overline{M}(x)} w_i$. A preprocessing step was used to identify set partitioning constraints and to set a higher weight for these, while keeping the weight equal to 1 for all other constraints.

If the weight of the set partition constraints is set sufficiently high, e.g., $w_i = 10$, the VND takes a different path, and more easily ends up in a feasible solution. For example, on instance 41, this new weighted version finds a solution with a relative objective function value of 0.983 after 60 minutes (and 0.979 after 60 seconds). However, even though this modification leads to feasible solutions on all instances in the test set, the quality of the feasible solution found is still somewhat arbitrary, and the median performance of the weighted version of GVNS becomes 0.974 and 0.981, on short and long runs, respectively, compared to 0.969 and 0.978 for the unweighted version.

Table 16: Results for additional MDMKP instances, reporting the objective function values relative to the best objective function value found among the four methods, after respectively 60 seconds (Short) and 3600 seconds (Long). For short runs, the median result of 11 runs is reported for LocalSolver, VND, and GVNS. The best performances for each instance and time limit are highlighted in bold. Instance names are abbreviated, and full names can be obtained by adding the prefix "100-30-30".

	Short (60 seconds)				Long (3600 seconds)			
Instance	CPLEX	LocalSolver	VND	GVNS	CPLEX	LocalSolver	VND	GVNS
0-0	$-\infty$	0.820	0.934	**0.974**	$-\infty$	0.705	1.000	**1.000**
0-1	$-\infty$	0.898	0.945	**0.959**	$-\infty$	0.902	0.988	**1.000**
0-2	$-\infty$	$-\infty$	0.883	**0.944**	$-\infty$	0.858	**1.000**	**1.000**
0-3	$-\infty$	0.838	0.911	**0.954**	$-\infty$	0.932	0.985	**1.000**
0-4	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	0.950	**1.000**
0-5	$-\infty$	0.941	0.950	**0.977**	$-\infty$	0.981	0.979	**1.000**
0-6	0.951	0.927	0.957	**0.975**	0.983	0.913	0.985	**1.000**
0-7	0.953	0.939	0.972	**0.984**	0.987	0.980	0.993	**1.000**
0-8	$-\infty$	0.880	0.943	**0.962**	$-\infty$	0.856	0.971	**0.987**
0-9	$-\infty$	0.931	0.962	**0.980**	**1.000**	0.889	0.986	0.994
0-10	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	**1.000**	**1.000**
0-11	$-\infty$	0.956	0.977	**0.989**	$-\infty$	0.979	0.997	**1.000**
0-12	$-\infty$	0.934	0.957	**0.969**	$-\infty$	0.963	0.971	**0.979**
0-13	0.971	0.964	0.981	**0.992**	1.000	0.980	**1.000**	**1.000**
0-14	$-\infty$	0.954	0.969	**0.979**	$-\infty$	0.962	0.988	**1.000**
1-0	$-\infty$	0.559	0.929	**0.966**	$-\infty$	0.507	**1.000**	**1.000**
1-1	$-\infty$	$-\infty$	0.734	**0.735**	$-\infty$	0.695	0.945	**1.000**
1-2	$-\infty$	$-\infty$	0.487	**0.612**	$-\infty$	0.738	0.988	**1.000**
1-3	$-\infty$	0.401	0.734	**0.794**	$-\infty$	0.542	0.925	**1.000**
1-4	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	0.465	**1.000**	**1.000**
1-5	$-\infty$	0.757	0.863	**0.938**	$-\infty$	0.805	0.990	**1.000**
1-6	0.788	0.714	0.823	**0.879**	0.916	0.917	0.883	**1.000**
1-7	0.652	0.809	0.858	**0.929**	0.991	0.944	0.983	**0.988**
1-8	$-\infty$	0.611	0.795	**0.893**	$-\infty$	0.589	0.955	**1.000**
1-9	$-\infty$	0.795	0.830	**0.907**	**1.000**	0.933	0.951	0.995
1-10	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	**1.000**	$-\infty$
1-11	$-\infty$	0.821	0.939	**0.958**	$-\infty$	0.754	0.998	**1.000**
1-12	$-\infty$	0.720	0.827	**0.899**	$-\infty$	0.907	0.942	**1.000**
1-13	0.834	0.801	0.868	**0.933**	0.991	0.991	0.965	**1.000**
1-14	$-\infty$	$-\infty$	0.868	**0.886**	$-\infty$	0.900	0.951	**1.000**
Min	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	0.883	$-\infty$
Median	$-\infty$	0.798	0.875	**0.941**	$-\infty$	0.894	0.986	**1.000**
Max	0.971	0.964	0.981	**0.992**	**1.000**	0.991	**1.000**	**1.000**
IQR	0.000	∞	0.147	0.092	∞	0.244	0.040	0.000

5 Concluding remarks

Binary integer programming (BIP) forms an important class of optimization problems, encompassing many more specialized structures, such as knapsack problems, arising when modelling real world applications. There have been few attempts at developing heuristics targeting the BIP. This paper investigated whether variable neighborhood search (VNS) is a viable technique for creating effective heuristic solvers, based on three research questions. In particular, a variable neighborhood descent (VND) and a general variable neighborhood descent (GVNS) were developed in this process.

The first research question posed was whether a useful and meaningful hierarchy of neighborhoods can be devised for the BIP. The premise was that these neighborhoods should be explored systematically without relying on external mixed integer programming solvers. Two hierarchies were suggested, one based on simultaneously flipping p variables such that the sum of the variables changed by q or $-q$, and the other based on first simultaneously flipping r consecutive variables from a sorted list, and then sequentially flipping additional variables following the same sorted list. Both types of neighborhoods were useful for solving BIP, and the best version of VND obtained contained a mix of neighborhoods from the two hierarchies.

The second research question pertained to a set of strategies for reducing the size of the explored neighborhoods. It was found to be efficient to flip several variables only when pairs of the variables appear in the same constraint but with opposite effects on the current value of the left hand side. Larger neighborhoods should be explored using a first improvement strategy, rather than a best improvement strategy, while the opposite is true for the smaller neighborhoods. The very largest neighborhoods should not be explored when the number of variables or constraints in the problem is too large. Limiting the exploration of any neighborhood to only improve feasible solutions did not have any positive effect on the performance of the search as a whole.

Finally, the third research question asked whether the obtained solutions compare well with primal bounds obtained by commercial software available for the BIP. The results were mixed, as the GVNS and the VND outperforms the two commercially available solvers CPLEX and LocalSolver on some instances, while the opposite holds for other instances. GVNS turns out to work particularly well for relatively small instances, with $n \leq 200$ variables, as well as for tightly constrained instances based on the *multidemand multidimensional knapsack problem*.

Hence, the main conclusion is that heuristics based on VNS for BIP may be particularly suitable as components of a heuristic that is based on variable fixing, so that subproblems with relatively

few variables are addressed by a VND or a GVNS. This can happen both in evolutionary algorithms, such as the one proposed by da Silva et al. (2020) for the *optimum satisfiability problem*, or when applying the core concept as outlined by Huston et al. (2008). Future research is needed to determine whether VNS can be successfully used as a subsolver in such circumstances. We also suggest for future research to test if the performance of VNS is dependent on the initial solution, and whether construction heuristics can be used instead of generating a random initial solution. Finally, we would like to investigate whether the random shaking step of the GVNS can be replaced by a systematic movement based on adaptive memory structures.

Acknowledgements

The authors wish to thank three anonymous reviewers for their helpful comments and suggestions.

References

S. Al-Shihabi. A novel core-based optimization framework for binary integer programs-the multidemand multidimensional knapsack problem as a test problem. *Operations Research Perspectives*, 8:100182, 2021.

H. Arntzen, L.M. Hvattum, and A. Løkketangen. Adaptive memory search for multidemand multidimensional knapsack problems. *Computers and Operations Research*, 33:2508–2525, 2006.

E. Balas. An additive algorithm for solving linear programs with zero-one variables. *Operations Research*, 13(4):517–546, 1965.

T. Benoist, B. Estellon, F. Gardi, R. Megel, and K. Nouioua. LocalSolver 1.x: a black box local-search solver for 0-1 programming. *4OR - A Quarterly Journal of Operations Research*, 9:299, 2011.

D. Bertsimas, D.A. Iancu, and D. Katz. A new local search algorithm for binary optimization. *INFORMS Journal on Computing*, 25(2):208–221, 2013.

S. Boulmier. *Optimisation globale avec LocalSolver (in French).* Phd dissertation, L'Université Grenoble Alpes, Grenoble, France, 2020.

P. Cappanera and M. Trubian. A local-search-based heuristic for the demand-constrained multidimensional knapsack problem. *INFORMS Journal of Computing*, 17:82–98, 2005.

CPLEX, 2020. https://www.ibm.com/support/knowledgecenter/en/SSSA5P_12.10.0/.

R.F. da Silva, L.M. Hvattum, and F. Glover. Combining solutions of the optimum satisfiability problem using evolutionary tunneling. *MENDEL*, 26:23–29, 2020.

T. Davoine, P.L. Hammer, and B. Vizvári. A heuristic for boolean optimization problems. *Journal of Heuristics*, 9:229–247, 2003.

F. Glover, G. Kochenberger, W. Xie, and J. Luo. Diversification methods for zero-one optimization. *Journal of Heuristics*, 25:643–671, 2019.

J. Glover, V. Quan, and S. Zolfaghari. Some new perspectives for solving 0–1 integer programming problems using Balas method. *Computational Management Science*, 2021. doi: https://doi.org/10.1007/s10287-021-00389-6. In press.

F. Gortázar, A. Duarte, M. Laguna, and R. Martí. Black box scatter search for general classes of binary optimization problems. *Computers and Operations Research*, 37:1977–1986, 2010.

S. Hanafi, J. Lazić, N. Mladenović, and C. Wilbaut. Variable neighborhood decomposition search with bounding for multidimensional knapsack problem. In *Proccedings of the 13th IFAC Symposium on Information Control Problems in Manufacturing*, volume 42, pages 2018–2022, Moscow, Russia, 2009.

P. Hansen, N. Mladenović, and D. Urošević. Variable neighborhood search and local branching. *Computers and Operations Research*, 33:335–350, 2006.

P. Hansen, N. Mladenović, J. Brimberg, and J.A. Moreno Pérez. Variable neighborhood search. In M. Gendreau and J.-Y. Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science*, pages 61–86. Springer, New York, NY, USA, 2010.

S. Huston, J. Puchinger, and P. Stuckey. The core concept for 0/1 integer programming. In *Fourteenth Computing: The Australasian Theory Symposium (CATS2008)*, pages hal–01299754, Wollongong, Australia, January 2008.

L.M. Hvattum, A. Løkketangen, and F. Glover. Comparisons of commercial MIP solvers and an adaptive memory (tabu search) procedure for a class of 0–1 integer programming problems. *Algorithmic Operations Research*, 7:13–21, 2012.

S. Khan, K.F. Li, E.G. Manning, and M.M. Akbar. Solving the knapsack problem for adaptive multimedia system. *Studia Informatica Universalis*, 2(1):157–178, 2002.

T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R.E. Bixby, E. Danna, G. Gamrath, A.M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D.E. Steffy, and K. Wolter. MIPLIB 2010 - mixed integer programming library version 5. *Mathematical Programming Computation*, 3:103–163, 2011.

X. Lai, J.-K. Hao, and D. Yue. Two-stage solution-based tabu search for the multidemand multidimensional knapsack problem. *European Journal of Operational Research*, 274:35–48, 2019.

J. Puchinger and G.R. Raidl. Bringing order into the neighborhoods: relaxation guided variable neighborhood search. *Journal of Heuristics*, 14:457–472, 2008.

C. Rego, T. James, and F. Glover. An ejection chain algorithm for the quadratic assignment problem. *INFORMS Journal on Computing*, 16(2):133–151, 2004.

H. Shojaei, T. Basten, M.C.W. Geilen, and A. Davoodi. A fast and scalable multi-dimensional multiple-choice knapsack heuristic. *ACM Transactions on Design Automation of Electronic Systems*, 18(4):Article 51, 32 pages, October 2013.

A.C. Trapp and R.A. Konrad. Finding diverse optima and near-optima to binary integer programs. *IIE Transactions*, 47(11):1300–1312, 2015.

N. Turajlić and I. Dragović. A hybrid metaheuristic based on variable neighborhood search and tabu search for the web service selection problem. *Electronic Notes in Discrete Mathematics*, 39:145–152, 2012.