ERASMUS UNIVERSITY ROTTERDAM

Erasmus School of Economics

Bachelor Thesis Econometrics and Operational Research

Comparing Algorithms for Generating Multiple Solutions for Mixed Integer Programming Problems

Name student: Emma van der Leeden

Student ID number: 473936

Supervisor: R.S.H. Willemsen

Second assessor: Dr. T.A.B. Dollevoet

Date final version: 03/07/2022

Abstract

We focus on methods for finding multiple solutions that are optimal and near-optimal for Mixed Integer Programming (MIP) problems. Finding multiple solutions for this type of problem is relevant for decision makers who might need dissimilar solutions that provide optimal and near-optimal objectives, because MIP is often used to formulate real-world problems. We compare the one-tree and the sequential algorithm based on the time they need to find a predetermined amount of solutions and the diversity between these found solutions. We define four diversity measures, namely a binary Hamming, an integer Hamming, a general Hamming, and a variance-based diversity. We find that our sequential algorithm is always faster than our one-tree algorithm. Besides this we find that none of the diversity measures show a preference for one of the algorithms.

Contents

1 Introduction

Mixed integer programming (MIP) formulations are used for a variety of different problems. Some problems that can be formulated using MIP are knapsack, traveling salesman, bin packing, and plant location problems. All of these are common problems that need to be solved on a regular basis. In this paper we focus on the problem of finding multiple solutions for MIP problems, which can be both optimal and near-optimal. We expand on a paper by Danna, Fenelon, Gu, & Wunderling (2007), and use two of the algorithms they defined. They use a diversity measure based on binary variables to compare their algorithms, we extend on this by defining diversity measures that also include integer and continuous variables.

Researchers who develop meta-heuristics have stated their need for diverse solutions. Some of these meta-heuristics include genetic algorithms and memetic algorithms, where the first are algorithms based on natural selection and the second are evolutionary algorithms that use local search rather than global search (Dutta & Mahanand, 2022). Research into finding multiple solutions is therefore relevant for improving the performance of these metaheuristics. Besides, MIP is used more often to solve sub-problems within a branch-and-bound algorithm. An example is using an MIP formulation with an objective to maximize the cut violation to help define the cuts (Danna et al., 2007; Greistorfer et al., 2008).

There is also a more practical relevance to finding multiple solutions for MIP problems. Many problems are approximations of real world situations, which can change quickly. Therefore a solution that is found for the original problem can be infeasible due to changes in the real situation. Besides, some aspects of the problem may be difficult to add into an MIP formulation. It can then be practical for decision makers to choose between solutions that are dissimilar, but still provide optimal or near-optimal objectives (Danna et al., 2007; Greistorfer et al., 2008).

To find good methods for generating multiple solutions for MIP problems we analyse two algorithms. The first is the one-tree algorithm as introduced by Danna et al. (2007), the second is a sequential algorithm as defined by Greistorfer et al. (2008). We compare these algorithms based on two aspects, the running time of the algorithm when generating a predetermined amount of solutions, and the diversity between these generated solutions. We use the algorithms to solve a selection of problems gathered from the MIPLIB libraries: MIPLIB3.0 (Bixby et al., 1998), MIPLIB2003 (Achterberg et al., 2006), and MIPLIB2017 (Gleixner et al., 2021). Most of these problems are also used by Danna et al. (2007).

Our comparison of the algorithms gives us the following main findings. First, our sequential algorithm is on average 500 times faster than our one-tree algorithm. When comparing our running times with those found by Danna et al. (2007), we find that our one-tree algorithm is on average 2000 times slower than theirs, and our sequential algorithm is on average 8 times faster than theirs. Lastly, none of the diversity measures show a clear preference between the two algorithms.

This paper adds to the existing literature on finding multiple solutions for MIP problems in multiple ways. First of all, we perform another comparison between the one-tree algorithm and the sequential algorithm. With this comparison we show that the way an algorithm is implemented strongly affects the performance of the algorithm. Besides this, we also introduce various diversity measures that can be used to compare the performance of algorithms that find multiple solutions for MIP problems.

The remainder of the paper is structured as follows. First, we give a literature review in Section 2. Next, we define the methodology we use to analyse our problem in Section 3. After this, we give a description of the used data and show the results in Section 4. Lastly, we give a conclusion, discuss our findings and provide further research suggestions in Section 5.

2 Literature review

Various methods for finding multiple solutions for Mixed Integer Programming (MIP) problems have been published. An MIP problem is defined as an optimization problem with a linear objective and linear constraints where at least one variable is integer (Wolsey, 2020; Bixby et al., 2000).

MIP models are often described as being versatile (Danna et al., 2005; Rothberg, 2007). Some examples of problems that can be modelled using MIP include single commodity single echelon problems, inventory-routing problems, and combinatorial optimization problems such as knapsack and traveling salesman problems (Ghiani et al., 2013; Berthold, 2006). In many cases these problems are solved using heuristics, an example of this is presented by Fasano (2008). Here they use heuristics to solve non-standard 3D-packing problems which include additional conditions such as balancing and tetris-like items. Another example is by Ambrosino, Paolucci, & Sciomachen (2015), they use an MIP-based heuristic to solve a multi port stowage planning problem. They determine how to stow a set of containers of different types onto a container ship, where the entire route of the ship is taken into consideration.

Some researchers have focused on generating multiple solutions for various problems. Lee, Phalakornkule, Domach, & Grossmann (2000), for example, look into generating all different solutions that provide the optimum value in certain linear programming models for metabolic networks, which are biochemical networks that contain a collection of connected reactions. They use a recursive method to find these solutions. Greistorfer et al. (2008) investigate finding multiple solutions for MIP problems. They compare simultaneous and sequential generation for finding two solutions and conclude that sequential generation often takes less computation whilst producing solutions with at least the same quality as those produced with simultaneous generation.

When generating multiple solutions it is important to define the diversity between these solutions. This metric shows how the found solutions differ from each other based on the difference in variable values. Danna et al. (2007) define the diversity between solutions based on the Hamming distance between binary variables. This distance is defined as the amount of "bits" that are different between the solutions. In this case it is equal to the amount of binary variables that take on a different value in the solutions. Danna & Woodruff (2009) first define a diversity measure based on the scaled Hamming distance between binary variables. They define a second diversity measure based on the variances of all variables and lastly a diversity measure based on the variances of the variables that is scaled with the mean.

Contrary to these definitions, Glover et al. (2000) state that the Hamming distance is not an appropriate measure for diversity, because it is only based on binary variables and thus does not take regular integer or continuous variables into account. Besides this, they also state that the Euclidean distance is not appropriate because it is not scale invariant, which means that the results can be affected by the scales of the variables. Both of these distances also have the shortcoming of not taking correlations into account. Glover et al. (2000) conclude that the Mahalanobis distance would be the most representative measure because it is scale invariant and it can take correlations between the variables into account. Even so, they state that this diversity measure can be difficult to calculate.

3 Methodology

In this paper we analyse two algorithms, the one-tree algorithm and a sequential algorithm. Since both algorithms are based on the branch-and-bound algorithm, we first describe this. Next, we describe the one-tree and sequential algorithm and how we compare the performance of these algorithms. The problem on which we compare the algorithms is the MIP(p, q) problem. This is defined as finding p solutions for an MIP problem that are within $q\%$ of the optimum. When solving this problem it is not guaranteed that p solutions can be found, if that is the case we work with the amount of solutions that were found.

3.1 Branch-and-bound algorithm

The most-used algorithm to solve MIP problems is branch-and-bound. This algorithm enumerates over the possible solutions by creating a tree of subproblems. First, we create the root node, which is equal to the linear programming (LP) relaxation of the original problem, and solve this problem to optimality. The LP relaxation is obtained by relaxing the integrality constraints, it is thus a less restricted version of the original problem since all variables that were integer in the original problem can now take on continuous values. Next, we enumerate over all nodes in the tree where there are three possibilities in each node. First, if the found solution is greater than the best solution that was found at that point, we fathom the node. Second, if the found solution is smaller than the best solution at that point and the solution satisfies the integrality constraints, we set that solution as the new best solution and fathom the node. For all other solutions we split the node into two child nodes, fathom the current node, and explore these created child nodes (Wolsey, 2020; Bixby et al., 2000; Danna et al., 2005, 2007).

To split the problem, we take an integer variable that is fractional in value in the current solution and impose constraints to restrict this variable around its current value. Thus, if we have a variable x_j that is fractional in this solution with value x_j^*, we split the problem by creating two sub-problems. The first has the restriction $x_j \leq \left\lfloor x_j^* \right\rfloor$ added to the problem, the second has $x_j \geq \left\lceil x_j^* \right\rceil$ added to the problem. Then in each node we solve the LP relaxation of that node to optimality (Danna et al., 2005; Wolsey, 2020; Berthold, 2006; Bixby et al., 2000).

3.2 One-tree algorithm

The first algorithm is the one-tree algorithm as introduced by Danna et al. (2007), which contains two phases as described in Algorithm 1 and Algorithm 2. The main idea of this algorithm is to first explore and create the regular branch-and-bound tree, which is done in phase I. Instead of fathoming nodes as in branch-and-bound we store all explored nodes such that these can be used in phase II. Then in phase II the stored nodes from phase I are explored again such that we can collect multiple solutions. This is done by iterating over the nodes and solving the LP relaxation of the problem in this node to optimality. If the solution is outside $q\%$ of the optimum the node is fathomed, otherwise we do two things. First, we check if it is feasible and not yet present in the solution set. If that is the case we add the solution to this set. After this we branch on a variable that is not restricted by its bounds within this node. These phases are summarized in Algorithm 1 and Algorithm 2.

Algorithm 1 Outline of one-tree algorithm: Phase I

```
1: Create a set of open nodes and add the root node to this set: $N_{open} \leftarrow \{rootnode\}$.
2: Create an empty set for the stored nodes: $N_{stored} \leftarrow \emptyset$.
3: Set the objective value of the incumbent to infinity: $z^* \leftarrow +\infty$.
4: while $N_{open} \neq \emptyset$ do
5:     Select a node $n$ from $N_{open}$.
6:     Solve the LP relaxation at node $n$.
7:     if no optimal solution is found then
8:         Fathom the node: $N_{open} \leftarrow N_{open} \setminus n$.
9:         Continue to the next iteration.
10:    end if
11:    Store the solution of this relaxation as $x(n)$ with objective value $z(n)$.
12:    if $z(n) \geq z^*$ then
13:        Fathom the current node: $N_{open} \leftarrow N_{open} \setminus \{n\}$.
14:        Add the current node to the set of stored nodes: $N_{stored} \leftarrow N_{stored} \cup \{n\}$.
15:    else
16:        if x(n) is integer valued then
17:            x(n) becomes the new incumbent: $x^* \leftarrow x(n); z^* \leftarrow z(n)$.
18:            Fathom the current node: $N_{open} \leftarrow N_{open} \setminus \{n\}$.
19:            Add the current node to the set of stored nodes: $N_{stored} \leftarrow N_{stored} \cup \{n\}$.
20:        else
21:            Choose branching variable $i$ that is fractional in the current solution.
22:            Build child nodes $n_1 = n \cap \{x_i \leq \lfloor x_i(n) \rfloor\}$ and $n_2 = n \cap \{x_i \geq \lfloor x_i(n) \rfloor + 1\}$.
23:            Fathom the current node: $N_{open} \leftarrow N_{open} \setminus \{n\}$.
24:            Add the child nodes to the set of open nodes: $N_{open} \leftarrow N_{open} \cup \{n_1, n_2\}$.
25:        end if
26:    end if
27: end while
```

3.3 Sequential algorithm

The second algorithm is the sequential algorithm, for which an outline is given in Algorithm 3. The idea of this algorithm is to generate multiple feasible solutions for the problem in a sequential manner. Greistorfer et al. (2008) define a sequential algorithm with the following steps. First, we find the optimal solution x^* by solving the original problem using regular branch-and-bound. Next, we restrict the objective value to ensure that all next solutions are within $q\%$ of the optimal value. After this, we continue finding solutions until we have p different solutions. In each iteration we add a constraint to exclude the previous solution. This process is summarized in Algorithm 3.

Algorithm 2 Outline of one-tree algorithm: Phase II

```
1: Used the stored nodes from phase I as the open nodes for phase II: $N_{open} \leftarrow N_{stored}$.
2: Reuse incumbent, $x^*$ with $z^*$ from phase I.
3: Create a set of solutions and add the incumbent from phase I to this set: $S \leftarrow \{x^*\}$.
4: while $N_{open} \neq \emptyset$ and $|S| < p$ do
5:     Select a node $n$ from $N_{open}$.
6:     Solve the LP relaxation at node $n$.
7:     if no optimal solution is found then
8:         Fathom the node: $N_{open} \leftarrow N_{open} \setminus n$.
9:         Continue to next iteration.
10:    end if
11:    Store the solution of this relaxation as $x(n)$ with objective value $z(n)$.
12:    if $z(n) > z^* + q|z^*|/100$ then
13:        Fathom the current node: $N_{open} \leftarrow N_{open} \setminus \{n\}$.
14:    else
15:        if x(n) is integer valued then
16:            if $x(n) \notin S$ then
17:                Add x(n) to the set of solutions: $S \leftarrow S \cup \{x(n)\}$.
18:            end if
19:        end if
20:        Choose branching variable $i$ that is not fixed by its bounds in the current node: $lb_i(n) < ub_i(n)$.
21:        Build child nodes $n_1 = n \cap \{x_i \leq \lfloor x_i(n) \rfloor\}$ and $n_2 = n \cap \{x_i \geq \lfloor x_i(n) \rfloor + 1\}$.
22:        Fathom the current node: $N_{open} \leftarrow N_{open} \setminus \{n\}$.
23:        Add child nodes to the set of open nodes: $N_{open} \leftarrow N_{open} \cup \{n_1, n_2\}$.
24:    end if
25: end while
```

Algorithm 3 Outline of the sequential algorithm for MIP(p, q).

```
1: Solve the problem with standard branch-and-bound.
2: Store the found optimal solution as $x^*$ with objective value $z^*$.
3: Create a set of solutions and add the found solution to this set: $S \leftarrow \{x^*\}$.
4: Define the set of binary variables: $B$.
5: Add constraint on objective value to ensure that all solutions are within $q$% of the optimum: $X \leftarrow X \cap \{c^T x \leq z^* + q|z^*|/100\}$
6: while $|S| < p$ do
7:     Add constraint to exclude the previous solution: $X \leftarrow X \cap \{\sum_{i \in B: s_i = 0} x_i + \sum_{i \in B: s_i = 1} (1 - x_i) \geq 1\}$.
8:     Solve the new problem with standard branch-and-bound.
9:     Store the found optimal solution as $x^*$.
10:    Add the new solution to the set of solutions: $S \leftarrow S \cup \{x^*\}$.
11: end while
```

3.4 Performance measures

We compare the algorithms based on two aspects. The first is the running time of the algorithm when solving MIP(p, q), and the second is the diversity between the generated solutions for

MIP(p, q). To formulate the different diversity measures we use the following set definitions:

S	The set of all found solutions.
B	The set of all binary variables.
I	The set of all integer variables.
V	The set of all variables.

We define the diversity as

$$D(S) = \frac{1}{|S|^2} \sum_{s,s' \in S} d(s, s'), \tag{1}$$

which is the average pairwise distance. In this formulation we can adjust $d(s, s')$ to change the distance measure between two solutions s and s'.

Danna et al. (2007) define a binary Hamming diversity. This is thus based on the Hamming distance on the set of binary variables, which gives the definition

$$d(s, s') = \frac{1}{|B|} \sum_{i \in B} |s_i - s'_i|, \tag{2}$$

where s_i is the value of variable i in solution s. The larger the value of the diversity the more diverse the found solutions are. Danna et al. (2007) prove that $D(S) \leq \frac{1}{2}$ for all solution sets S. Thus, we look for a diversity as close as possible to 0.5.

We expand on the paper by Danna et al. (2007) by formulating a diversity measure that also includes integer variables compared to just the binary variables. We define an integer Hamming distance as follows,

$$d(s, s') = \frac{1}{|I|} \sum_{i \in I} \begin{cases} 1, & \text{if } |s_i - s'_i| > 0 \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

This defines the distance between two solutions for all integer variables. We count the amount of times that a variable takes on a different value in two solutions where each difference has weight one. The formulation simplifies to (2) in case it is a pure binary problem. For this definition it also holds that we look for the algorithm that has the highest value for this metric.

We also formulate two different diversity measures based on all types of variables. The first is similar to the previous two measures and is a general Hamming diversity, which is defined as

$$d(s, s') = \frac{1}{|V|} \sum_{i \in V} \begin{cases} 1, & \text{if } |s_i - s'_i| \geq 1 \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

This is equal to the differences between the solutions for all variables. We count two variables to be different if their values differ by at least 1. If two integer or binary variables are different this difference is always at least equal to 1, thus we also use this same threshold for the continuous variables. In this definition we use the same weight for all variables. This definition simplifies to (3) in case the problem does not contain continuous variables, and to (2) in case the problem is pure binary. For this measure we also look for the highest value.

Lastly, we have a variance-based diversity measure for all types of variables based on a definition by Danna & Woodruff (2009). Here the diversity is defined as

$$D(S) = \frac{1}{|V|}\sum_{i\in V}\sigma_i^2(S) \tag{5}$$

$$\sigma_i^2(S) = \frac{1}{|S|}\sum_{s\in S}\left(s_i - \frac{1}{|S|}\sum_{s'\in S}s'_i\right)^2. \tag{6}$$

This measure takes the average of the variance of the variables over the solutions. The variance of a variable i over the solution set is defined in (6). The main disadvantage of this measure is that all variances may be in a different scale, but they are all given the same weight in this definition. If the variance for a variable is larger, it means that the variable differs strongly in value over the solution set. Consequently, for this last diversity measure we also look for larger values.

4 Results

We compare our algorithms for the MIP(p, q) problem, and we compare them with the results found by Danna et al. (2007). To perform this last comparison we use the same parameters as Danna et al. (2007), which are $p = 10$ and $q = 1$. All results are obtained using CPLEX 20.1.0 in Java on a computer with an i7 core and 16 GB memory. There is a time limit of 15 minutes on each of the algorithms to ensure that the chosen problems can be run within a reasonable amount of time. We first give a description of the problems on which we test the algorithms. Next, we present our findings for each of the metrics on which we compare the algorithms. For some problems the algorithm gives a heap space error instead of producing results, this is indicated with $\sim$.

4.1 Problem set

This paper focuses on recreating some of the results found by Danna et al. (2007) and expanding on their ideas. They test their algorithms on 57 different problems. We also test our algorithms on these problems. These problems come from two MIPLIB libraries, namely from MIPLIB3.0

(Bixby et al., 1998) and MIPLIB2003 (Achterberg et al., 2006). They only test their algorithms on problems that can be solved to optimality within half an hour. The newest MIPLIB library is MIPLIB2017 (Gleixner et al., 2021), which has three categories. All the problems we use come from the category *easy*, which means that the problems can be solved to optimality in less than one hour with standard methods. We also add some problems to our analysis from this library that were not yet used by Danna et al. (2007).

Each of the problems contains a MIP formulation, which can contain integer, binary, and continuous variables in various combinations. The total set of problems is shown in Appendix A. Of the 57 problems there are 19 pure binary problems, 3 problems with binary and integer variables, 22 problems with binary and continuous variables, 1 problem with integer and continuous variables, and 12 problems with all types of variables. On average a problem contains 3663 variables. The amount of variables a problem contains ranges from 18 (*flugpl*) to 87482 (*nw04*). The amount of constraints a problem has ranges from 6 (*mod008*) to 10460 (*mzzv42z*), where a problem contains on average 852 constraints.

4.2 Time to solve MIP*(p,q)*

We first compare the algorithms based on how much time they need to solve MIP*(p,q)*. The running time in seconds for each of the problems per algorithm is shown in Table 1. We consider two times to be equal if the absolute difference is less than 3.601 seconds, this value is justified in Section B.1.

4.2.1 Comparison one-tree algorithm with sequential algorithm

The one-tree algorithm is unable to finish phase I within 15 minutes for most of the problems, however the sequential algorithm is always able to finish within this time. The bold values in Table 1 indicate which algorithm is faster, which is in all cases the sequential algorithm. For some problems the one-tree algorithm is unable to provide results due to heap space problems. These problems are *bell5, flugpl, gt2, lseu, mas76, mod008* and *pk1.*

The sequential algorithm is on average nearly 500 times faster than the one-tree algorithm, which is in this case equal to a difference of on average 800 seconds. This is however an underestimation of the actual value since the one-tree algorithm was often cutoff due to the time limit and in these cases a time of 900 seconds was used in the calculation of the average. This shows that in our case the sequential algorithm performs better than the one-tree algorithm based on the time it needs to solve MIP*(10,1)*. Danna et al. (2007) find the opposite, as their case the one-tree algorithm is faster than their sequential algorithm.

4.2.2 Comparison with Danna et al. (2007)

If we compare the times for each of the algorithms with the times that were found by Danna et al. (2007), as also shown in Table 1, we see a lot of differences. First, our one-tree algorithm is slower than theirs for all problems, with a difference of nearly 800 seconds on average. We can also state that their algorithm is at least 2000 times faster than ours. Both of these values are again underestimations due to the time cutoffs for our algorithm. The main explanation for these differences is the way the algorithm is implemented in Java. The algorithm can, for example, be implemented with regular loops, recursion, or callbacks. While we use regular loops to implement the algorithm, it may be the case that Danna et al. (2007) used one of the other methods which was more effective.

For the sequential algorithm we find that our algorithm mostly provides better times than theirs. Our algorithm is on average 8 times faster, which is in this case 300 seconds faster. Our algorithm is faster for 34 problems. If we only look at these problems, we find that they are on average 660 seconds faster. There are two problems for which our algorithm is slower, namely *air03* and *mitre*. For these problems our algorithm is 40 and 10 seconds slower respectively. Lastly, for the 21 other problems our algorithm differs at most 3.601 seconds from the results from Danna et al. (2007).

We thus find that in most cases our sequential algorithm performs faster than the sequential algorithm as implemented by Danna et al. (2007). Besides, for the problems where our algorithm is faster the difference is much more significant than the difference when our algorithm is slower. One explanation for finding these faster times is based on the difference in the used version of CPLEX. In our case we use version 20.1.0, and Danna et al. (2007) used version 10.1. The newer version that we use is more effective and faster than the previous version. Another explanation is that we use a computer that is in some aspects more efficient than the one used by Danna et al. (2007).

4.3 Number of found solutions

Next, we look at the amount of solutions each algorithm was able to find within the set time limit. Table 2 shows how many solutions each algorithm is able to find for each of the problems. Since the one-tree algorithm is often unable to finish phase I within the time limit we see that for these problems the algorithm is unable to find any solutions.

The one-tree algorithm is able to finish phase I for eight problems. For five of these problems the algorithm is able to find 10 solutions. For the problems *air03, enigma* and *khb05250* it finds 4, 1 and 5 solutions respectively. The inability to find 10 solutions can be explained by the speed

Table 1: Time in seconds to solve MIP*(10,1)*.

	One-tree			Sequential	
Problems	Phase I	Phase II	Total		Danna et al. (2007)
10teams	>900		>900	30.114	-
aflow30a	>900		>900	40.689	-
air03	44.291	855.762	>900	**64.273**	+
air04	>900		>900	122.208	-
air05	>900		>900	102.141	-
app3	329.973	408.238	738.211	**5.948**	
arki001	>900		>900	58.944	-
bell3a	>900		>900	0.270	-
bell5	∼	∼	∼	1.077	=
blend2	>900		>900	3.376	-
cap6000	>900		>900	15.087	=
dcmulti	>900		>900	2.417	-
disctom	>900		>900	18.249	-
dsbmip	>900		>900	2.869	=
egout	>900		>900	0.113	=
enigma	36.002	863.984	>900	**0.381**	=
fiber	>900		>900	2.864	-
fixnet6	>900		>900	2.090	-
flugpl	∼	∼	∼	0.091	-
gen	>900		>900	0.914	=
gesa2	>900		>900	3.252	=
gesa2_o	>900		>900	5.478	-
gesa3	>900		>900	26.259	-
gesa3_o	>900		>900	6.563	-
gt2	∼	∼	∼	0.630	=
khb05250	156.158	743.844	>900	**3.229**	=
l152lav	>900		>900	16.209	=
lseu	∼	∼	∼	2.110	=
mas76	∼	∼	∼	433.444	-
misc03	8.788	0.141	8.929	**1.483**	-
misc06	68.865	16.472	85.337	**2.870**	=
misc07	>900		>900	57.851	-
mitre	>900		>900	24.483	+
mod008	∼	∼	∼	1.721	=
mod010	>900		>900	3.662	=
mod011	>900		>900	848.052	-
modglob	>900		>900	6.110	=
mzzv11	>900		>900	299.610	-
mzzv42z	>900		>900	244.303	-
nw04	>900		>900	630.813	-
p0033	9.119	0.461	9.580	**0.516**	=
p0201	>900		>900	1.468	-
p0282	>900		>900	1.239	=
p0548	>900		>900	0.883	=
p2756	>900		>900	3.785	=
pk1	∼	∼	∼	55.529	-
pp08aCUTS	>900		>900	8.260	-
pp08a	>900		>900	9.581	-
qiu	>900		>900	185.245	-
qnet1	>900		>900	4.632	-
qnet1_o	>900		>900	3.674	-
rgn	>900		>900	1.321	-
rout	>900		>900	154.426	-
set1ch	>900		>900	8.742	=
stein27	26.823	0.076	26.899	**1.160**	-
stein45	>900		>900	41.059	-
vpm1	>900		>900	0.847	=
vpm2	>900		>900	6.170	-

+: Our algorithm is slower. =: The difference is at most 3.601. -: Our algorithm is faster. ∼: Heap space error.

Table 2: Number of solutions found within 15 minutes when solving MIP*(10,1)*.

Problems	One-tree	Sequential
10teams	0	10
aflow30a	0	10
air03	4	10
air04	0	10
air05	0	10
app3	10	10
arki001	0	10
bell3a	0	1
bell5	∼	10
blend2	0	10
cap6000	0	10
dcmulti	0	10
disctom	0	10
dsbmip	0	10
egout	0	1
enigma	1	2
fiber	0	10
fixnet6	0	10
flugpl	∼	1
gen	0	10
gesa2	0	10
gesa2_o	0	10
gesa3	0	10
gesa3_o	0	10
gt2	∼	10
khb05250	5	10
l152lav	0	10
lseu	∼	5
mas76	∼	10
misc03	10	10
misc06	10	10
misc07	0	10
mitre	0	10
mod008	∼	10
mod010	0	10
mod011	0	10
modglob	0	10
mzzv11	0	10
mzzv42z	0	10
nw04	0	10
p0033	10	10
p0201	0	10
p0282	0	10
p0548	0	10
p2756	0	10
pk1	∼	1
pp08aCUTS	0	10
pp08a	0	10
qiu	0	10
qnet1	0	10
qnet1_o	0	10
rgn	0	10
rout	0	10
set1ch	0	10
stein27	10	10
stein45	0	10
vpm1	0	10
vpm2	0	10

∼: Heap space error.

of the algorithm, which is quite slow as described in Section 4.2. For each of these problems the algorithm was cut off in phase II due to the time limit. The sequential algorithm is unable to find 10 solutions for six of the problems. These are *bell3a, egout, enigma, flugpl, lseu* and *pk1.* For the problems *lseu* and *pk1* we find the same amount of solutions as Danna et al. (2007), namely 5 and 1 respectively. This is also the proven maximum amount of solutions that can be found for MIP*(10,1)* (Danna et al., 2007).

For the four other problems, *bell3a, egout, enigma* and *flugpl*, our findings are particularly interesting. For *bell3a* and *flugpl* we find one solution. Danna et al. (2007) also find one solution with the sequential algorithm for these problems. However, they prove that there exist more solutions within 1% of the optimum. For *flugpl* it could be the case that the algorithm as implemented by Danna et al. (2007) would be able to find more solutions if given more time since it was cut off due to time restrictions.

These differences can be explained by the termination criteria of the sequential algorithm. This algorithm has three termination criteria: if it has passed the time limit of 900 seconds; if it has found 10 solutions; or if the problem is infeasible. For these problems it is thus the case that the problem is determined to be infeasible, this can happen due to the way the algorithm adds the restriction to exclude previous solutions. These restrictions are based on the binary variables, however the other solutions could have the same values for all binary variables, but differ for the other types of variables. In this case the sequential algorithm would not be able to find the other solutions. The *bell3a* problem contains, besides binary variables, integer and continuous variables. The *flugpl* problem contains only integer and continuous variables, which means that for this problem we would not be able to exclude solutions in general.

We find 1 solution for *egout* and 2 for *enigma.* Danna et al. (2007) find 2 and 3 solutions for these problems respectively, and also prove that those are the maximum amount of solutions that can be found within 1% of the optimum. We thus find one solution less with our implementation of the sequential algorithm. This means that for these problems the algorithm terminates too early. For *egout* we can offer the same explanation as for *bell3a* and *flugpl* because this problem also contains continuous variables as well as binary variables. However, *enigma* only contains binary variables and we would thus expect the algorithm to be able to find all solutions.

4.4 Performance comparison

Next, we compare the diversities that are found when solving MIP*(10,1).* The results for the different diversity measures are shown in Table 3. For the problems for which the algorithms were unable to find a solution we do not show the diversity, since the diversity is then not defined. In case the algorithm finds one solution the diversity is always zero. For each problem where we have a diversity for both algorithms we show the best, and thus highest, value in bold. For the binary Hamming diversity we also add columns to compare our results with the findings from Danna et al. (2007). For these columns we define two diversities to be the same if the difference is at most 0.003 (Section B.2).

4.4.1 Binary Hamming diversity

The first diversity measure is the binary Hamming diversity, as defined by (2) in Section 3.4. The results for this diversity are shown in the first two columns of Table 3. We first compare this measure between the two algorithms and see that for four problems the one-tree algorithm performs better and for three problems the sequential algorithm performs better. Because the algorithms have a different approach to finding the solutions, and thus find solutions in a different order, it is possible that this measure differs between the two algorithms. Based on this measure we do not find a preference for one of the algorithms.

Next, we compare our findings for this diversity with the findings from Danna et al. (2007), as also shown in Table 3. For the one-tree algorithm the diversity is equal for the problems *air03, misc03* and *misc06.* For *khb05250* and *stein27* our diversity is on average 0.075 larger. And lastly for *p0033* our diversity is 0.060 smaller. For the sequential algorithm this measure is equal for 37 of the problems. For six problems our findings are slightly larger than those found by Danna et al. (2007), as these are on average 0.022 larger. For 14 problems our findings are smaller. These are on average 0.023 smaller than the diversities found by Danna et al. (2007).

The differences in the values of the binary Hamming diversity for our results and the findings by Danna et al. (2007) can be explained by the possible set of solutions. For most problems there are multiple options available for the solutions that provide the same objective value. It is possible that CPLEX now iterates over the possible solutions in a different order and therefore finds different solutions that are within 1% of the optimal objective value. If the algorithms find a different set of solutions the diversity measure will then also take on a different value. However, these differences are mostly quite small.

4.4.2 Integer Hamming diversity

The second diversity measure is the integer Hamming diversity as defined by (3) in Section 3.4. The results for this measure are shown in the third and fourth column of Table 3. We see that the one-tree algorithm performs better for three problems, and the sequential performs better for four problems. These differences can be explained by the same reasons as in Section 4.4.1.

Table 3: Diversities for MIP*(10,1)* with comparison to the results from Danna et al. (2007) for the binary Hamming diversity.

	Binary Hamming				Integer Hamming		General Hamming		Variance-based	
Problems	One-tree		Sequential		One-tree	Sequential	One-tree	Sequential	One-tree	Sequential
10teams			0.038	=		0.038		0.033		0.017
aflow30a			0.036	=		0.044		0.043		5.719
air03	**0.001**	=	0.001	=	**0.001**	0.001	**0.001**	0.001	**0.000**	0.000
air04			0.001	=		0.001		0.001		0.000
air05			0.001	=		0.001		0.001		0.000
app3	**0.026**		0.018		**0.026**	0.018	**0.015**	0.001	**2601.337**	0.000
arki001			0.022	-		0.032		0.026		2.299
bell3a			0.000	=		0.000		0.000		0.000
bell5	∼		0.067	+	∼	0.047	∼	0.060	∼	5.645
blend2			0.010	=		0.009		0.015		275.836
cap6000			0.001	=		0.001		0.001		0.000
dcmulti			0.026	=		0.026		0.063		12.061
disctom			0.361	-		0.361		0.361		0.181
dsbmip			0.141	-		0.146		0.094		9374244.759
egout			0.000	-		0.000		0.000		0.000
enigma	0.000	-	0.020	-	0.000	0.020	0.000	0.020	0.000	0.010
fiber			0.004	=		0.004		0.006		0.714
fixnet6			0.028	=		0.045		0.049		86.932
flugpl	∼		0.000	=	∼	0.000	∼	0.000	∼	0.000
gen			0.039	=		0.055		0.030		2.312
gesa2			0.008	=		0.007		0.006		0.018
gesa2_o			0.005	=		0.009		0.006		0.018
gesa3			0.009	=		0.042		0.009		1.077
gesa3_o			0.005	=		0.011		0.009		1.235
gt2	∼		0.075	=	∼	0.093	∼	0.093	∼	0.189
khb05250	0.080	+	**0.137**	=	0.080	**0.137**	0.033	**0.042**	21333.208	**35382.647**
l152lav			0.003	=		0.003		0.003		0.002
lseu	∼		0.110	=	∼	0.110	∼	0.110	∼	0.055
mas76	∼		0.070	+	∼	0.070	∼	0.075	∼	31.287
misc03	0.072	=	**0.076**	=	0.072	**0.076**	0.071	**0.075**	0.036	**0.038**
misc06	**0.031**	=	0.025	=	0.031	**0.043**	**0.019**	0.008	**11.445**	0.460
misc07			0.059	=		0.059		0.059		0.030
mitre			0.003	=		0.004		0.003		0.002
mod008	∼		0.012	=	∼	0.012	∼	0.012	∼	0.006
mod010			0.005	=		0.005		0.005		0.003
mod011			0.039	+		0.039		0.018		191227.675
modglob			0.023	=		0.028		0.019		11129.413
mzzv11			0.005	-		0.006		0.006		0.006
mzzv42z			0.008	-		0.008		0.008		0.007
nw04			0.000	=		0.000		0.000		0.000
p0033	**0.218**	-	0.158	+	**0.218**	0.206	**0.218**	0.150	**0.109**	0.079
p0201			0.135	-		0.135		0.135		0.068
p0282			0.006	=		0.006		0.006		0.003
p0548			0.010	-		0.012		0.010		0.005
p2756			0.004	-		0.005		0.004		0.002
pk1	∼		0.000	=	∼	0.000	∼	0.000	∼	0.000
pp08aCUTS			0.127	+		0.207		0.179		709.104
pp08a			0.100	=		0.120		0.157		506.463
qiu			0.292	-		0.305		0.133		2.316
qnet1			0.003	=		0.003		0.003		0.001
qnet1_o			0.003	=		0.003		0.002		0.001
rgn			0.063	=		0.122		0.100		0.190
rout			0.073	-		0.091		0.062		0.130
set1ch			0.020	=		0.072		0.029		263.677
stein27	0.271	+	**0.439**	+	0.271	**0.439**	0.271	**0.439**	0.136	**0.219**
stein45			0.356	-		0.356		0.356		0.178
vpm1			0.048	=		0.072		0.036		26.738
vpm2			0.049	-		0.055		0.043		26.927

+: We find a larger value. =: The difference is at most 0.003. -: We find a smaller value. ∼: Heap space error.

4.4.3 General Hamming diversity

The third diversity measure is the general Hamming diversity as defined in (4) in Section 3.4. These values are shown in the fifth and sixth columns of Table 3. According to this measure the one-tree algorithm performs better for four problems, and the sequential algorithm performs better for three problems, the same results as we found for the binary Hamming diversity. For this measure we also find small differences between the two algorithms due to their different approaches and thus different order of finding the solutions.

For all pure binary problems we expect to find the same values for the binary, integer, and general Hamming diversity. The one-tree algorithm provides results within 15 minutes for three pure binary problems, namely *air03, p0033* and *stein27.* The one-tree algorithm finds the same values for the three Hamming diversity measure for all three problems.

However, when using the sequential algorithm we find different values for the four problems *mitre, p0033, p0548* and *p2756.* For *p0033* we find that all three values are different, and for the other three problems we find that the integer Hamming diversity is 0.001 or 0.002 larger than the other diversities. The smaller differences between the values can be due to numerical inaccuracies. The larger differences may be caused by the way the variables are defined in the problem, since the measures are implemented based on these definitions.

4.4.4 Variance-based diversity

The last diversity measure we look at is the variance-based diversity as defined by (5) and (6) in Section 3.4. The results for this measure are shown in the seventh and eighth column of Table 3. For this measure we find the same results as for the binary and general Hamming diversity. Namely, the one-tree algorithm performs best for four of the problems, and the sequential algorithm for three of the problems.

When we look at this diversity measure we can see that these values differ strongly in scale for each of the problems. This makes it more difficult to compare the difference in performance in each of the algorithms. For most problems we do see that the values for the two algorithms are within the same scale. This is not the case for the problems *app3* and *misc06.* This can be caused by the strong differences in the solutions that were found. In the sequential algorithm we add a restriction to exclude the previously found combination of binary variables. This algorithm will thus look for solutions that differ in these variables. The one-tree algorithm iterates over the nodes from the branch-and-bound tree and may thus find solutions that differ more in the integer and continuous variables, which tend to be on a larger scale than the binary ones.

5 Conclusion

In this paper we compared two algorithms for finding multiple solutions for mixed integer programming (MIP) problems. We compared the one-tree and the sequential algorithm based on how much time they need to find a predetermined amount of solutions and the diversity between these solutions.

First of all, we find that for our implementations the sequential algorithm is always faster, on average 500 times, than the one-tree algorithm. This is contradictory to the expected results based on the findings by Danna et al. (2007), who find that the one-tree algorithm is often faster than the sequential algorithm. This difference can be explained by the comparison of the run times of the algorithms to the run times found by Danna et al. (2007). Our one-tree algorithm is on average 2000 times slower than theirs, whilst our sequential algorithm is on average 8 times faster than theirs. Secondly, we find that based on various diversity measures we cannot indicate a clear preference for one of the algorithms.

Thus, when comparing these two algorithms to determine which is better for finding multiple solutions for MIP problems we conclude that the sequential algorithm is preferred. The algorithm performs similarly based on the diversity measures, but it is much faster in achieving this. Consequently, the sequential algorithm can already be used in practice to find multiple solutions.

Based on our findings we recommend some aspects that require further research. In this paper we used specific methods to implement the algorithms. When implementing the algorithms in a different manner one may find very different results. We have shown this with the differences in running times compared to Danna et al. (2007). Thus, looking into the effects of the various implementations on the performance of the algorithms would be a relevant extension.

Besides this, we find that our diversity measures do not show a lot of difference in the performance between the algorithms. It would thus be practical to look into various aspects of the diversity measures. Firstly, there should be more research into diversity measures and whether these can find a proper difference between the algorithms. Secondly, it would also be interesting to look at a diversity measure that takes all types of variables into account but is also properly scaled to ensure that all variables have the same effect on the value. Third, another practical direction for future research would be into more non-linear diversity measures. Lastly, it would be relevant to look more deeply into whether certain algorithms work better for certain problems, and thus in what sense the problem structure influences the performance of the algorithm.

References

Achterberg, T., Koch, T., & Martin, A. (2006). Miplib 2003. *Operations Research Letters*, *34*(4), 361–372.

Ambrosino, D., Paolucci, M., & Sciomachen, A. (2015). A mip heuristic for multi port stowage planning. *Transportation Research Procedia*, *10*, 725–734.

Berthold, T. (2006). *Primal heuristics for mixed integer programs* (Unpublished doctoral dissertation). Zuse Institute Berlin (ZIB).

Bixby, R., Ceria, S., McZeal, C. M., & Savelsbergh, M. W. (1998). *An updated mixed integer programming library: Miplib 3.0* (Tech. Rep.).

Bixby, R., Fenelon, M., Gu, Z., Rothberg, E., & Wunderling, R. (2000). Mip: Theory and practice—closing the gap. In *Ifip conference on system modeling and optimization* (pp. 19–49).

Danna, E., Fenelon, M., Gu, Z., & Wunderling, R. (2007). Generating multiple solutions for mixed integer programming problems. In *International conference on integer programming and combinatorial optimization* (pp. 280–294).

Danna, E., Rothberg, E., & Pape, C. L. (2005). Exploring relaxation induced neighborhoods to improve mip solutions. *Mathematical Programming*, *102*(1), 71–90.

Danna, E., & Woodruff, D. L. (2009). How to select a small set of diverse solutions to mixed integer programming problems. *Operations Research Letters*, *37*(4), 255–260.

Dutta, P., & Mahanand, B. (2022). Affordable energy-intensive routing using metaheuristics. In S. Mishra, H. Tripathy, P. Mallick, A. Sangaiah, & G.-S. Chae (Eds.), *Cognitive big data intelligence with a metaheuristic approach* (pp. 193–210). Elsevier.

Fasano, G. (2008). Mip-based heuristic for non-standard 3d-packing problems. *4OR*, *6*(3), 291–310.

Ghiani, G., Laporte, G., & Musmanno, R. (2013). *Introduction to logistics systems management.* John Wiley & Sons.

Gleixner, A., Hendel, G., Gamrath, G., Achterberg, T., Bastubbe, M., Berthold, T., ... Shinano, Y. (2021). MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. *Mathematical Programming Computation.* Retrieved from `https://doi.org/10.1007/s12532-020-00194-3` doi: 10.1007/s12532-020-00194-3

Glover, F., Løkketangen, A., & Woodruff, D. L. (2000). Scatter search to generate diverse mip solutions. In M. Laguna & J. L. G. Velarde (Eds.), *Computing tools for modeling, optimization and simulation* (pp. 299–317). Springer.

Greistorfer, P., Løkketangen, A., Voß, S., & Woodruff, D. L. (2008). Experiments concerning sequential versus simultaneous maximization of objective function and distance. *Journal of Heuristics*, *14*(6), 613–625.

Lee, S., Phalakornkule, C., Domach, M. M., & Grossmann, I. E. (2000). Recursive milp model for finding all the alternate optima in lp models for metabolic networks. *Computers & Chemical Engineering*, *24*(2-7), 711–716.

Rothberg, E. (2007). An evolutionary algorithm for polishing mixed integer programming solutions. *INFORMS Journal on Computing*, *19*(4), 534–541.

Wolsey, L. A. (2020). *Integer programming.* John Wiley & Sons.

Appendix A List of used problems

Table 4: An overview of the problems, with information on the variables and the constraints of each of the problems.

Problem	#Variables	#Binary Variables	#Integer Variables (not binary)	#Continuous Variables	#Constraints
10teams	2025	1800	0	225	230
aflow30a	842	421	0	421	479
air03	10757	10757	0	0	124
air04	8904	8904	0	0	823
air05	7195	7195	0	0	426
app3	3080	100	0	2980	766
arki001	1388	442	96	850	1048
bell3a	133	39	32	62	123
bell5	104	30	28	46	91
blend2	353	239	25	89	274
cap6000	6000	6000	0	0	2176
dcmulti	548	75	0	473	290
disctom	10000	10000	0	0	399
dsbmip	1886	192	0	1694	1182
egout	141	55	0	86	98
enigma	100	100	0	0	21
fiber	1298	1254	0	44	363
fixnet6	878	378	0	500	478
flugpl	18	0	11	7	18
gen	870	144	6	720	780
gesa2	1224	240	168	816	1392
gesa2_o	1224	384	336	504	1248
gesa3	1152	216	168	768	1368
gesa3_o	1152	336	336	480	1224
gt2	188	24	164	0	29
khb05250	1350	24	0	1326	101
l152lav	1989	1989	0	0	97
lseu	89	89	0	0	28
mas76	151	150	0	1	12
misc03	160	159	0	1	96
misc06	1808	112	0	1696	820
misc07	270	259	0	11	212
mitre	10724	10724	0	0	2054
mod008	319	319	0	0	6
mod010	2655	2655	0	0	146
mod011	10958	96	0	10862	4480
modglob	422	98	0	324	291
mzzv11	10240	9989	251	0	9499
mzzv42z	11717	11482	235	0	10460
nw04	87482	87482	0	0	36
p0033	33	33	0	0	16
p0201	201	201	0	0	133
p0282	282	282	0	0	241
p0548	548	548	0	0	176
p2756	2756	2756	0	0	755
pk1	86	55	0	31	45
pp08aCUTS	240	64	0	176	246
pp08a	240	64	0	176	136
qiu	840	48	0	792	1192
qnet1	1541	1288	129	124	503
gnet1_o	1541	1288	129	124	456
rgn	180	100	0	80	24
rout	556	300	15	241	291
set1ch	712	240	0	472	492
stein27	27	27	0	0	118
stein45	45	45	0	0	331
vpm1	378	168	0	210	234
vpm2	378	168	0	210	234
Average	3662	3149	37	476	852

Appendix B Comparing performance measures

B.1 Time

We compare the running times we find for the algorithms with the running times Danna et al. (2007) found for those same algorithms. In Table 5 we see that the range for the absolute difference in time is [0.001, 3599.909]. A histogram for the distribution within this range is shown in Figure 1. Due to the large spread of this value we want 99.9% of the interval to be considered different, and the other 0.1% can be considered equal. This gives the interval [0.001, 3.601] within which we state that the difference is negligible and we thus count the times as equal, otherwise we see them as different.

Table 5: Descriptive statistics on the absolute differences for the time and diversity measure.

	Mean	St.Dev.	Min.	Max.
Absolute Time Difference	371.576	847.782	0.001	3599.909
Absolute Diversity Difference	0.011	0.021	0.000	0.092

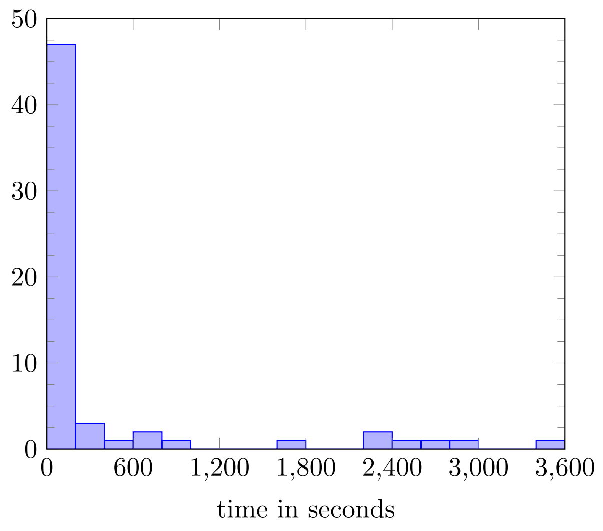

Figure 1: A histogram of the distribution of the absolute diversity differences.

B.2 Diversity

We compare the integer Hamming diversity results with the diversity measure results found by Danna et al. (2007). When we look at the range of the absolute differences of our values with theirs we see that this is in the range [0.000, 0.092], this is shown in Table 5. A histogram of the distribution of the absolute differences is shown in Figure 2. We define the difference such that values that are in the first 3% of the interval are considered equal, and the values in the other 97% of the interval are considered different. This then gives us the range of [0.000, 0.003] for which we state that we can neglect the difference and see the values as equal.

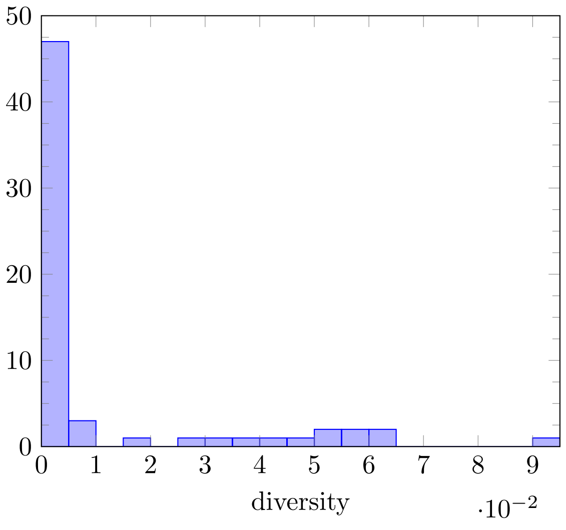

Figure 2: A histogram of the distribution of the absolute time differencs.

Appendix C Code description

For our research we programmed three different classes in Java. The first class is *SequentialAlgorithm*, this class performs the sequential algorithm on a specific problem and produces the values for the four diversity measures. This class thus consists of five different methods. One that runs the algorithm and one for each of the diversity measures.

The second class is *OneTreeAlgorithm7*, this class performs the one-tree algorithm on a specific problem. It also determines the values for the four diversity measures. The *OneTreeAlgorithm7* class consists of ten methods. The first method runs the actual algorithm. After this, we have a method that determines the LP relaxation of a problem, and a method that copies a cplex object. Next, we have a method that checks whether a certain solution is integer, a method that finds an integer variable that is fractional in the current solution, and a method that finds a variable that is not restricted by its bounds in the current problem. Lastly, we have one method for each of the diversity measures.

Since the one-tree algorithm is based on the branch-and-bound method we also define a class *Node2*. This last class defines the nodes within the branch-and-bound tree and is used by *OneTreeAlgorithm7*.

The methods for finding the diversity measure are defined for the following diversity measures: a binary Hamming diversity, an integer Hamming diversity, a general Hamming diversity, and a variance-based diversity.