# Genetic Branching: An Evolutionary Framework for Interpretable Branching Strategies

Simon Renard[1], Quentin Louveaux[2], and Bernard Fortz[3]

[1] Département d'informatique, Université libre de Bruxelles, Brussels, Belgium
simon.renard@ulb.be
[2] Institut Montefiore, Université de Liège, Liège, Belgium
q.louveaux@uliege.be
[3] HEC Liège, Université de Liège, Liège, Belgium
bernard.fortz@uliege.be

**Abstract.** We present Genetic Branching (GB), a framework based on genetic programming to automatically design branching strategies for the branch and bound algorithm. Unlike existing learning-based approaches, such as graph convolutional neural networks (GCNN), GB evolves interpretable symbolic scoring functions using only a small number of training instances. We evaluate GB on four benchmark families and compare it against SCIP's reliability branching, pseudocost branching, and a state-of-the-art GCNN model imitating strong branching. Our experiments show that, when applied on some specific problem classes, GB consistently matches or outperforms existing methods on easy and medium instances using only 10 training instances, and remains competitive on hard instances. This demonstrates that evolutionary computation can produce efficient and interpretable branching strategies tailored to specific problem classes.

**Keywords:** Branch and Bound · Genetic Programming · Mixed-Integer Linear Programming.

## 1 Introduction

Solving mixed-integer linear programs (MILPs) plays a crucial role in many sectors. Applications include planning the electricity production of different power plants [12, 26], routing [1, 11], and many others [28].

An MILP consists of minimizing a linear objective function subject to a set of linear constraints. The presence of integer variables in a MILP makes these problems NP-hard [20].

The most popular algorithm used to solve such MILPs is Branch and Bound (B&B) [22]. It is a divide and conquer approach that divides the solution space recursively into smaller subproblems and uses local bounds to prune the resulting search trees as much as possible [13]. The efficiency of B&B depends on several heuristics. The node selection strategy chooses the next subproblem to explore.

The branching strategy decides how the current problem is split into two subproblems. While having a significant impact on search tree size, and so the solving time, no universal method that is efficient for all MILPs is known [24].

This difficulty to establish a strategy that is proven to minimize the B&B tree size has led the researchers to use machine learning (ML) [29]. The main goal is to design either a branching strategy or a selection strategy using ML that leverages information from a specific set of instances of interest. This results in strategies that, ideally, are built specifically for the instances of interest.

In the area of using ML for the branching strategy, two main approaches exist. Either use imitation learning to learn how to imitate an existing strategy, or use reinforcement learning to discover new strategies from scratch.

In this paper, we propose a framework to build new branching strategies using genetic programming: Genetic Branching (GB). Our experiments show that in most cases, GB outperforms the default strategy of the SCIP [10] solver on different instance classes. In addition, it outperforms the state-of-the-art approach that uses graph convolutional neural networks to imitate strong branching (GCNN) [17] on these classes. Another advantage of GB is that it requires very few training instances compared to GCNN.

## 2 Background

### 2.1 MILP and Branch and Bound

A mixed-integer linear program (MILP) is an optimization problem where the objective is to minimize a linear function $c^\intercal x$ while satisfying a set of linear constraints $Ax \leq b$. In addition, some variables can be restricted to be integer: $x_i \in \mathbb{Z}\ \forall i \in \mathcal{I}$. This problem can be written as

$$MILP = \arg\min_x \{c^\intercal x | Ax \leq b, x_i \in \mathbb{Z}\ \forall i \in \mathcal{I}, x_i \in \mathbb{R}\ \forall j \notin \mathcal{I}\} \tag{1}$$

The most common algorithm to solve this MILP is branch and bound (B&B) [22]. It recursively divides the solution space into two subproblems and prunes nodes using bounds that prove the optimal solution is not in a given part of the tree.

To divide a node into two, the linear relaxation of the node is solved, yielding a fractional optimal solution $x^0$. This gives the following dual bound: $z^0 = c^\intercal x^0$. We denote by $C = \{i | i \in \mathcal{I},\ x_i^0 \notin \mathbb{Z}\}$ the set of variables that must be integer in the original MILP but are fractional in the solution of the relaxation. This set $C$ is the set of branching candidates.

While being a simple algorithm, B&B relies on several heuristics that can have a big impact on the final B&B tree size and the solving time [3]. The primal heuristics try to improve the primal bound which can lead to a smaller tree by allowing to prune several nodes. The node selection strategy decides which node to process next. A branching strategy chooses a variable $x_i, i \in C$, then two child nodes are created. On the left child the constraint $x_i \leq \lfloor x_i^0 \rfloor$ is added. On the right child the constraint $x_i \geq \lceil x_i^0 \rceil$ is added.

### 2.2 Branching strategies

Most of the branching strategies can be viewed as scoring functions that compute a score for each candidate variable $x_i \in C$. Then, the branching is performed on the variable with the highest score.

Strong branching (SB) [5] is a branching strategy that generally leads to very small B&B trees. However, the associated scoring function is computationally intensive, making this strategy uncompetitive. For a candidate variable $x_i, i \in C$, SB generates the two children corresponding to branching on this variable and solves their linear relaxations. This gives the solutions $x^{0-}$ and $x^{0+}$. The improvement in the dual bounds for the two children is given by $\Delta_i^- = c^\intercal x^{0-} - c^\intercal x^0$ and $\Delta_i^+ = c^\intercal x^{0+} - c^\intercal x^0$. Finally these two improvements $\Delta_i^-$ and $\Delta_i^+$ are combined together to compute the score of $x_i$. The computationally intensive part comes from computing the solution of the linear relaxations of both children.

On the other hand, pseudocost branching (PC) [8] is very fast to compute but often leads to larger trees than SB. PC tries to estimate the improvement in the dual bound after branching on a certain variable based on historical data. It branches on the variable with the highest estimated improvement. While fast to make a decision, PC suffers from the lack of data at the first nodes where no historical data is available.

To tackle this issue, hybrid strong/pseudocost branching [2] first uses SB in the first levels of the tree, for the deeper nodes, it uses PC scores. However, it is still possible, when treating deeper nodes, that a variable has no branching history and thus no reliable pseudocost estimate. That is why reliability branching (RB) [2] uses SB for the variables that are considered not reliable, i.e., that do not have a history of a certain size. If the variable is considered reliable, pseudocost is used. RB is the default strategy in SCIP [10].

## 3 Related work

Recently, the idea to use machine learning to improve the B&B algorithm has gained significant interest from the community. Either for designing branching strategies, node selection strategies or primal heuristics [29].

Two main approaches have been proposed for learning branching strategies. In [4, 17], the authors use imitation learning to train models that are able to imitate strong branching based on simple features. Trying to imitate SB is a natural choice as SB is known to lead to very small B&B trees but with a high computation time [2]. In this regard, the current best approach is to use graph convolutional neural network as models [17]. The authors managed to either compete or beat the default strategy of the SCIP [10] reliability branching when the model is trained on instances from one type of problem. We will refer later to this branching strategy as GCNN.

Another approach is to use reinforcement learning to discover new strategies. Therefore, the model is no longer limited by a hand crafted strategy and a better new strategy might be discovered. In [16, 27, 30], the authors use this

approach. While the results are encouraging, their model rarely performs better than GCNN. In fact, using reinforcement learning for this task suffers from several limitations. First, the action space for the learning agent is very large. Then, it is very difficult to distribute the rewards of the different actions of an episode. It is not well-known which decision was good or bad for reducing the B&B tree size.

In [15], the authors develop a GCNN to learn both a branching strategy and node selection strategy to benefit from their interaction in the B&B tree. This approach results in improved performance in terms of time compared to the approach in [17], but the model has only been tested on instances of the same size as the training ones.

To the best of our knowledge, the only work that uses genetic programming (GP) to improve B&B is [25]. The authors use GP to design a node selection strategy using very few operators and terminals. Their method leads to smaller solving time than using the default strategy of the SCIP solver.

## 4 Genetic Branching

In this section, we introduce Genetic Branching (GB), a GP based method that generates scoring functions that can be used as branching strategies. As explained in Section 2.2, a branching strategy can be summarized as computing a scoring function to assign a score to each candidate variable, then branching on the variable with the highest score.

These scoring functions are represented as symbolic expressions represented by an Abstract Syntax Tree (AST). In this tree-based representation, internal nodes correspond to mathematical operators and the leaf nodes correspond to the terminals, i.e. input features or constants. Figure 1 illustrates the reliability branching scoring function with its AST representation. The AST is built using different terminals and operators. $SB_{score}$ and $PC_{score}$ are respectively the strong branching and pseudocost score of the variable. $\eta^-$ and $\eta^+$ correspond to the number of pseudocost updates for the down and up directions so far during the solving process.

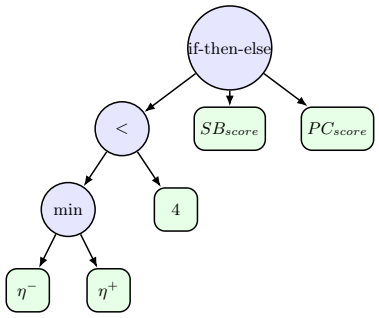
### 4.1 Operators and terminals

In order to have a scoring function that is able to make good decisions, it must take as input a vector of features that represent as much as possible the state of the solver and the corresponding variable. These features constitute the terminal set. We have selected 3 sets of features that together form the input of the function. Table 1 summarizes all the features used.

The first set is composed of *static features*. These features gather static information about the variable for the current instance. Therefore, it must be computed only once for each instance we are trying to solve. It contains information about the variable's objective coefficient, statistics about the variable in

Fig. 1: Representation of the reliability branching scoring function as an Abstract Syntax Tree.

$$f_{RB} = \begin{cases} SB_{score}, & \text{if } \min\left(\eta^-, \eta^+\right) < 4 \\ PC_{score}, & \text{otherwise} \end{cases}$$




the matrix $A$, and statistics about the constraints in which the variable is active. These features are used in [21].

The second set is composed of *tree features*. These features aim to gather information about the state of the solving tree. These features must be computed only once for each node in which we try to make a decision, and are the same for all the candidates. These features are used in [19].

The third set is composed of *dynamic features*. These features represent the state of the input variable. They must be computed before each evaluation of the score for a given variable. These features are used in [21].

These 3 feature sets compose the set of terminals that the GP algorithm can use. In addition, we add to the terminals a few constants: $\{2^k | k \in \{0, 1, 2, 3\}\}$.

These terminals are combined by the GP algorithm using the different operators. Table 2 lists all the operators used.

### 4.2 GP framework

The GP algorithm used is based on [6]. The evolution process iteratively modifies the population of trees to improve their fitness. It depends on several components. First, an initial population of size $n$ is generated. Then, for each generation, a selection is performed within the population to select good individuals. This selection is based on the evaluation of each individual. The selected individuals are paired together for crossover with a probability $p_{mate}$. Finally, each individual has a probability $p_{mut}$ to mutate. Figure 2 illustrates the evolution from one generation to the next one using all the presented genetic operators.

*Initial population* To ensure a diverse initial population, we employed the half-and-half method [6] to generate individuals. This technique combines two distinct tree-generation strategies: the grow method, which produces irregularly shaped trees with varying depths, and the full method, which constructs balanced trees where all branches extend to the maximum specified depth.

Table 1: List of features used for the genetic programming process.

(a) Static Features

| Name | Description | Size |
|---|---|---|
| $c$ | Variable objective's coefficient | 1 |
| $constr\_active\_count$ $constr\_active\_\{stat\}^{\{+,-\}}$ $stat \in \{mean, stdev, min, max\}$ | Report the number of constraints in which the variable is active and statistics on positive/negative coefficient (mean, stdev, min, max) | 9 |
| $constr\_deg\_\{stat\}$ $stat \in \{mean, stdev, min, max\}$ | Statistics about constraints' degrees in which the variable participates (mean, stdev, min, max) | 4 |

(b) Tree features

| Name | Description | Size |
|---|---|---|
| $gap$ | Gap | 1 |
| $leaf\_freq$ | Leaf frequency. Ratio between the number of open nodes in the B&B tree and the number total of nodes. | 1 |
| $tree\_weight$ | Tree weight. Sum of the weights $2^{-d(v)}$ for every node $v$ in the set of non open leaves of the tree where $d(v)$ represents the depth of the node [19]. | 1 |
| $compl$ | Completion. Estimation of the completion of the solving process [19]. | 1 |
| $depth$ | Depth of current node | 1 |

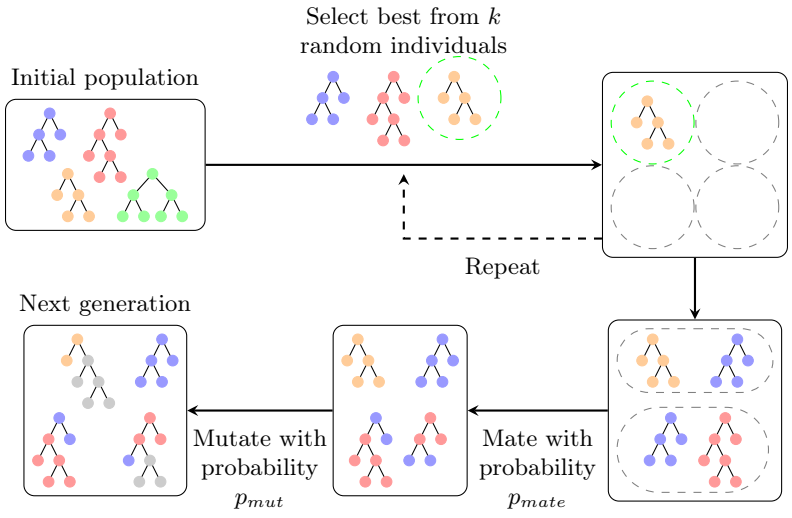
(c) Dynamic features

| Name | Description | Size |
|---|---|---|
| $min\_frac\_dist$ | The minimum distance of the variable's value to the nearest integer down or up. | 1 |
| $ceil\_frac\_dist$ | The distance of the variable's value to the nearest integer up. | 1 |
| $PC_{up}, PC_{down}$, $PC_{score}$ | Pseudocosts statistics. The estimated objective degradation for downward and upward branches, along with the aggregated selection score used to rank candidate variables | 3 |
| $ninf^{\{-,+\}}$ | Infeasibility statistics. Number of cutoffs branching for each direction. | 2 |
| $nbranch^{\{-,+\}}$ | Number of times, a bound of the variable was changed in given direction due to branching. | 2 |
| $down\_gain, up\_gain$, $SB_{score}$ | Strong Branching scores | 3 |
| $\eta^-, \eta^+$ | Number of pseudocost updates for the down and up directions | 2 |

Table 2: List of operators used for the genetic programming process

| Operator | Types | Description |
|---|---|---|
| + | $(\mathbb{R}, \mathbb{R}) \to \mathbb{R}$ | Addition of the inputs. |
| - | $(\mathbb{R}, \mathbb{R}) \to \mathbb{R}$ | Subtraction of the inputs. |
| * | $(\mathbb{R}, \mathbb{R}) \to \mathbb{R}$ | Multiplication of the inputs. |
| / | $(\mathbb{R}, \mathbb{R}) \to \mathbb{R}$ | Safe division of the inputs. If the denominator is 0, then the results is 0. |
| min | $(\mathbb{R}, \mathbb{R}) \to \mathbb{R}$ | The minimum of its inputs. |
| max | $(\mathbb{R}, \mathbb{R}) \to \mathbb{R}$ | The maximum of its inputs. |
| $\neg$ | $\mathbb{B} \to \mathbb{B}$ | The negation of its input. |
| round | $\mathbb{R} \to \mathbb{R}$ | Round the input |
| if_then_else | $(\mathbb{B}, \mathbb{R}, \mathbb{R}) \to \mathbb{R}$ | If the result of the first expression is true, then return the second expression, else return the third one. |
| $<$ | $(\mathbb{R}, \mathbb{R}) \to \mathbb{B}$ | True if the first input is smaller than the second one. |
| $>$ | $(\mathbb{R}, \mathbb{R}) \to \mathbb{B}$ | True if the first input is bigger than the second one. |
| and | $(\mathbb{B}, \mathbb{B}) \to \mathbb{B}$ | True if both inputs are true. |
| or | $(\mathbb{B}, \mathbb{B}) \to \mathbb{B}$ | True if at least one input is true. |

Fig. 2: Illustration of the genetic operators applied during the evolution process to get the next generation.



In addition to the randomly generated individuals, we added several individuals to the population that represent some known branching strategies. These individuals are the following:

- An individual that represents the pseudocost strategy
- Two individuals that represent the reliability branching strategy with a reliability threshold of 4 and 6
- Two individuals that represent the hybrid strong/pseudocost with a depth limit of 4 and 6.

*Evaluation* In order to evaluate an individual, i.e. a scoring function that can be used as a branching strategy, we solve $m$ instances of the focused problem using the individual as a branching strategy within a time limit $t$. For each instance, we record either the actual time required to solve it or an estimated completion time. If the solver reaches the time limit, the total time is estimated using the following formula:

$$\hat{T} = \frac{t}{n_{nodes}}.\hat{n}_{nodes}$$

where $t$ is the time elapsed, $n_{nodes}$ is the number of nodes processed so far, and $\hat{n}_{nodes}$ represents the estimated total number of nodes required to solve the instance optimally [19]. The final performance score for an individual is then calculated as the shifted geometric mean of these recorded times.

We have chosen to use the time as a metric rather than the number of nodes. This choice makes the method sensitive to the computation time of the different metrics. For instance, computing the SB score of a variable can give very good information to perform a good branching, but it is computationally intensive.

*Selection* To select the next generation, we use a $k$-tournament selection, where the best individual is chosen from $k$ randomly sampled candidates. This process is repeated to create a new generation of $n$ individuals. This method strikes a balance between exploitation and exploration: the former is achieved by selecting the best individuals, while the latter is achieved by retaining less fit individuals in the population.

*Crossover* The crossover between two individuals is a single point crossover. Two individuals exchange randomly selected sub-trees.

*Mutation* When a mutation is performed, a node is randomly selected from the individual and removed, along with its sub-tree. Then, a new sub-tree is grown randomly at this position.

### 4.3 Lazy Evaluation

As the goal of GB is to discover a branching strategy that minimizes the solving time of the instances of interest, it seems logical to only select features that can be computed quickly. However, the strength of state-of-the-art branching

strategies such as RB is that they find a balance between computing or not expensive features such as SB scores. Thus, not being able to use such features could heavily penalize our approach.

To enable the use of such features in our strategies while remaining competitive in terms of time, we have implemented a lazy evaluation of the features. A feature is only computed at the moment it is needed for the score computation, and not beforehand. So, even if an individual represents the reliability branching, the SB score will only be computed for the variables that are considered unreliable. This allows the individuals to compute and use expensive features only when a given condition is met.

## 5 Experimental results

In this section, we present an experiment to compare our approach to different state of the art branching strategies. All the code necessary to reproduce the results is available online at https://github.com/sirenard/genetic_branching.

### 5.1 Setup

To test and evaluate GB, we apply GB to 4 different sets of problems. For each problem, we randomly generate easy, medium and hard instances. The easy instances are used to evaluate the individuals during the evolution process. The problems used are: set covering problems (SC) [7], combinatorial auctions problems (CA) [23], capacitated facility location problems (CFL) [14] and maximum independent set instances (MIS) [9]. A description of each problem and their settings for each difficulty is given in Appendix A.

We compare our approach against 2 strategies of the SCIP solver [10]: RB and PC. In addition, we also test against the approach developed in [17] which uses a GCNN to imitate SB as it is the state of the art approach that uses machine learning to learn a branching strategy. We refer to this strategy as GCNN.

For each problem, we have applied GB with the following parameters. The evolution is performed with $n = 100$ individuals over 500 generations. To evaluate the individuals, $m = 10$ easy instances are solved with a time limit of $t = 240$ seconds. For the GP parameters, we choose $p_{mate} = 0.5$, $p_{mut} = 0.2$ and $k = 3$ for the $k$-tournament selection. This evolution process takes 48h with 100 Genoa @ 2.4 GHz CPUs.

For each type of problem, we solve 50 problems of each difficulty with each strategy with a 1800 seconds time limit. The evaluation is performed on a x86-64-v4 @ 3.25 GHz CPU and an NVIDIA RTX 6000 Ada GPU. The GPU is only used for the GCNN strategy.

### 5.2 Individuals

One major advantage of using GP is that its output is easily interpretable compared to a GCNN where it is very hard to distinguish the importance and impact

of each feature. In this section, we describe the resulting scoring functions that have been found for each class of problem.

The resulting scoring function for SC instances is:

$$f_{SC} = \begin{cases} PC_{down} + c, & \text{if } depth < 6 \\ PC_{score}, & \text{otherwise} \end{cases}$$

This function looks like hybrid branching, one of the individuals in the initial population, but instead of using SB score for the first nodes, the score is the addition of the down PC score and the variable's objective coefficient. Thus, when the pseudocosts are not well initialized, this function branches on the candidate which represents the subset with the highest cost.

The resulting scoring function for CFL instances is:

$$f_{CFL} = 3PC_{score} + nbranch^{+} + constr\_active\_mean^{+} + constr\_active\_stdev^{+}$$

This function depends mostly on the PC score and the number of times a variable's bound was changed due to branching down. Between two variables with similar PC scores, the number of changes due to branching is used as a tie breaker, as well as statistics on their positive coefficients. It can be seen as an improvement of pseudocost branching with a tie breaker for similar PC scores.

The resulting scoring function for CA instances is:

$$f_{CA} = \left[\min(constr\_active\_count, PC_{down}) + constr\_active\_min^{+}\right] * constr\_active\_count$$

This function has no similarities with any of the initial known individuals which suggests that GB finds a completely new scoring function. The function depends mostly on the number of times the variable appears in the constraints. In the MILP formulation of CA, a variable that appears in a lot of constraints corresponds to a bid with a lot of items. Thus, this new branching strategy chooses to separate the problem on the decision to take or not the biggest bid of the auction.

The resulting scoring function for MIS instances is:

$$f_{MIS} = constr\_active\_count$$

This function is also an entirely new one that has no similarities with the initial known individuals. It only depends on the number of constraints in which the variable is active. Regarding the MILP formulation, this corresponds to branching on the variable that corresponds to the vertex appearing in the highest number of cliques.

Looking at the resulting scoring functions, we observe 2 different behaviors. For SC and CFL instances, the resulting function is similar to one of the existing ones added to the population (hybrid branching and pseudocost branching). This shows the importance of these initial individuals. For CA and MIS, GB is able to find new scoring functions with no similarity with known used strategy.

### 5.3 Performances

Table 3 sums up the different results of the different strategies on the 4 types of problem. It reports the geometric mean of the number of nodes (s=10) and time (s=1) spent for each strategy. The wins metric corresponds to the number of times a solver is the fastest. The number of instances solved within the time limit is also reported.

Table 3: Results of the different strategies. Shifted geometric mean of the number of nodes and time are reported with $s = 10$ and $s = 1$. #wins refers to the number of times the solver is the fastest among the 4. #solved refers to the number of instances solved optimally within the time limit of 1800 seconds. For each test and metric, the best value is reported in bold.

| | Easy | | | | Medium | | | | Hard | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | #nodes | time | #wins | #solved | #nodes | time | #wins | #solved | #nodes | time | #wins | #solved |
| RB | 2456.86 | 62.21 | 0 | 50 | 16418.30 | 256.09 | 8 | **48** | 75160.60 | **997.33** | **15** | **28** |
| PC | 5044.58 | 73.91 | 0 | 50 | 28159.70 | 325.13 | 0 | 45 | 98023.16 | 1189.95 | 0 | 19 |
| GCNN | **2201.78** | 51.17 | 21 | 50 | **14034.87** | 238.28 | 10 | 47 | **60556.26** | 1100.33 | 1 | 21 |
| GB | 4552.64 | **48.78** | **29** | 50 | 24685.20 | **214.44** | **30** | 47 | 102260.27 | 1005.91 | 14 | 25 |
| Set Cover | | | | | | | | | | | | |
| RB | **1533.69** | 35.68 | 1 | 50 | **18418.23** | 195.69 | 20 | 47 | **81311.16** | **1097.05** | **19** | **25** |
| PC | 6210.41 | 66.36 | 0 | 50 | 51207.79 | 346.80 | 0 | 42 | 124105.52 | 1405.08 | 0 | 13 |
| GCNN | 1599.52 | 28.10 | 12 | 50 | 33910.63 | 310.74 | 0 | 42 | 110076.82 | 1499.25 | 0 | 12 |
| GB | 2172.90 | **23.78** | **37** | 50 | 30253.95 | **190.69** | **27** | **47** | 117322.84 | 1172.47 | 6 | 20 |
| Combinatorial Auction | | | | | | | | | | | | |
| RB | **135.31** | 100.00 | 7 | 50 | **109.59** | 728.18 | 4 | 46 | **44.38** | 1134.70 | 3 | 39 |
| PC | 325.27 | **88.91** | 14 | 50 | 264.18 | 633.65 | 16 | **49** | 119.09 | 983.20 | 14 | 41 |
| GCNN | 321.38 | 92.54 | 12 | 50 | 319.41 | 721.60 | 6 | 46 | 138.32 | 1045.74 | 8 | 41 |
| GB | 313.47 | 89.02 | **17** | 50 | 260.45 | **617.19** | **23** | **49** | 118.23 | **960.04** | **19** | **43** |
| Capacitated Facility Location | | | | | | | | | | | | |
| RB | **25.34** | 4.46 | 3 | 50 | 1125.59 | 83.78 | 0 | **50** | 11844.42 | 1149.35 | 0 | 26 |
| PC | 126.94 | 4.41 | 11 | 50 | 3438.39 | 161.37 | 0 | 49 | 18200.57 | 1518.83 | 0 | 11 |
| GCNN | 41.94 | 2.91 | 17 | 50 | 6315.98 | 191.49 | 0 | 47 | 25840.60 | 1510.25 | 0 | 5 |
| GB | 66.64 | **2.75** | **19** | 50 | **746.32** | **27.09** | **50** | **50** | **7387.20** | **308.58** | **47** | **47** |
| Max Independent Set | | | | | | | | | | | | |

For SC and CA, GB is the fastest and has the most wins on easy and medium instances. On hard instances, RB is the fastest but GB is faster than other baselines. This indicates that, while being competitive on instances of similar size as the training ones, the scoring function found does not scale well on hard instances. However, GB is still faster than GCNN and PC on the hard instances.

Regarding the number of nodes for SC, GB uses fewer nodes than PC although their scoring functions are very similar. This result highlights the benefit

of including the subset cost as a score for the first nodes when the PC score is not yet reliable.

For CFL, PC and GB exhibit similar performances, which is the expected result as the scoring function from GB depends mostly on the PC score. These two strategies outperform the other baselines RB and GCNN regarding the solving time for all categories of instances.

For MIS instances, where the scoring function found by GB is very simple, GB significantly outperforms the baseline strategies. While GB solves almost all the hard instances to optimality, RB solves half of them. On medium and hard instances, GB also uses significantly fewer nodes.

### 5.4 Discussion

To assess the robustness of GB, we repeated the training process several times with different training instances. The resulting scoring functions are always very similar to the ones presented in this paper. They differ only by a few terms or operations with a little impact on the final score. This indicates that the evolution converge to similar individuals without being sensitive to the random initialization of the population. As a consequence, the performances on the test instances are also very similar, confirming the robustness of the discovered branching strategies.

While working on distinct problems, we try to use GB on a heterogeneous set of training instances to discover a new scoring function that does not depend on the nature of the input instance to be efficient. We randomly selected 10 easy instances from MIPLIB 2017 [18] and test it against other MIPLIB instances. However, the resulting function is not competitive compared to SCIP's default strategy reliability branching. It even performs worse than pseudocosts. This shows a limitation of GB to adapt to a heterogeneous set of instances.

## 6 Conclusion

The experimental results show that GB is able to discover new branching strategies for the focused problems. This is the case for CA and MIS where the resulting functions are completely new branching strategies. For SC, while GB does not find a new strategy, it results in a customized version of an existing one: hybrid branching. Regarding CFL, although GB does not discover a new strategy, it stays competitive by selecting a variant of an existing strategy from the literature thanks to the known individuals added in the initial population.

A major improvement compared to using GCNN architecture to learn a strategy is that GB needs only 10 training instances compared to a few thousand. This makes GB a much more realistic approach as having easy instances similar to the ones of interest can be a hard task. In addition, GB leads to an easily interpretable function.

An interesting direction of development is to investigate how to use GB on heterogeneous instances.

## 7 Acknowledgments

Computational resources have been provided by the Consortium des Équipements de Calcul Intensif (CÉCI), funded by the Fonds de la Recherche Scientifique de Belgique (F.R.S.-FNRS) under Grant No. 2.5020.11 and by the Walloon Region.

## References

1. Abdullah, Z.N., Ahmad, I., Hussain, I.: Segment routing in software defined networks: A survey. IEEE Communications Surveys & Tutorials **21**(1), 464–486 (2018)
2. Achterberg, T.: Constraint Integer Programming. Ph.D. thesis, Technische Universität Berlin (2007)
3. Achterberg, T., Wunderling, R.: Mixed Integer Programming: Analyzing 12 Years of Progress, pp. 449–481 (01 2013)
4. Alvarez, A.M., Louveaux, Q., Wehenkel, L.: A machine learning-based approximation of strong branching. INFORMS Journal on Computing **29**(1), 185–195 (2017)
5. Applegate, D., Bixby, R.E., Chvátal, V., Cook, W.J.: Finding cuts in the TSP. Tech. Rep. 95–05, DIMACS Center, Rutgers University, Piscataway, NJ, USA (Mar 1995)
6. Bäck, T.: Evolutionary computation 1: Basic algorithms and operators. CRC press (2018)
7. Balas, E., Ho, A.: Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study. In: Combinatorial optimization, pp. 37–60. Springer (2009)
8. Bénichou, M., Gauthier, J.M., Girodet, P., Hentges, G., Ribière, G., Vincent, O.: Experiments in mixed-integer linear programming. Mathematical programming **1**(1), 76–94 (1971)
9. Bergman, D., Cire, A.A., Van Hoeve, W.J., Hooker, J.: Decision diagrams for optimization, vol. 1. Springer (2016)
10. Bolusani, S., Besançon, M., Bestuzheva, K., Chmiela, A., Dionísio, J., Donkiewicz, T., van Doornmalen, J., Eifler, L., Ghannam, M., Gleixner, A., Graczyk, C., Halbig, K., Hedtke, I., Hoen, A., Hojny, C., van der Hulst, R., Kamp, D., Koch, T., Kofler, K., Lentz, J., Manns, J., Mexi, G., Mühmer, E., Pfetsch, M.E., Schlösser, F., Serrano, F., Shinano, Y., Turner, M., Vigerske, S., Weninger, D., Xu, L.: The SCIP Optimization Suite 9.0. Technical report, Optimization Online (February 2024)
11. Callebaut, H., De Boeck, J., Fortz, B.: Preprocessing for segment routing optimization. Networks **82**(4), 459–478 (2023)
12. Carroll, P., Flynn, D., Fortz, B., Melhorn, A.: Sub-hour unit commitment milp model with benchmark problem instances. In: International Conference on Computational Science and Its Applications. pp. 635–651. Springer (2017)
13. Conforti, M., Cornuéjols, G., Zambelli, G.: Integer Programming. Graduate Texts in Mathematics, 271, Springer International Publishing, Cham, 1st ed. 2014. edn. (2014)
14. Cornuéjols, G., Sridharan, R., Thizy, J.M.: A comparison of heuristics and relaxations for the capacitated plant location problem. European journal of operational research **50**(3), 280–297 (1991)
15. Du, S., Tong, J., Fan, W.: Learning efficient branch-and-bound for solving mixed integer linear programs. Applied Soft Computing **172**, 112863 (2025)

16. Etheve, M., Alès, Z., Bissuel, C., Juan, O., Kedad-Sidhoum, S.: Reinforcement learning for variable selection in a branch and bound algorithm. In: International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research. pp. 176–185. Springer (2020)
17. Gasse, M., Chételat, D., Ferroni, N., Charlin, L., Lodi, A.: Exact combinatorial optimization with graph convolutional neural networks. Advances in neural information processing systems **32** (2019)
18. Gleixner, A., Hendel, G., Gamrath, G., Achterberg, T., Bastubbe, M., Berthold, T., Christophel, P.M., Jarck, K., Koch, T., Linderoth, J., Lübbecke, M., Mittelmann, H.D., Ozyurt, D., Ralphs, T.K., Salvagnin, D., Shinano, Y.: MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. Mathematical Programming Computation (2021)
19. Hendel, G., Anderson, D., Le Bodic, P., Pfetsch, M.E.: Estimating the size of branch-and-bound trees. INFORMS Journal on Computing **34**(2), 934–952 (2022)
20. Karp, R.M.: Reducibility Among Combinatorial Problems, pp. 219–241. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
21. Khalil, E., Le Bodic, P., Song, L., Nemhauser, G., Dilkina, B.: Learning to branch in mixed integer programming. In: Proceedings of the AAAI conference on artificial intelligence. vol. 30 (2016)
22. Land, A.H., Doig, A.G.: An Automatic Method for Solving Discrete Programming Problems, pp. 105–132. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
23. Leyton-Brown, K., Pearson, M., Shoham, Y.: Towards a universal test suite for combinatorial auction algorithms. In: Proceedings of the 2nd ACM conference on Electronic commerce. pp. 66–76 (2000)
24. Lodi, A.: The Heuristic (Dark) Side of MIP Solvers, pp. 273–284. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
25. Maudet, G., Danoy, G.: Search strategy generation for branch and bound using genetic programming (2024)
26. Montero, L., Bello, A., Reneses, J.: A review on the unit commitment problem: Approaches, techniques, and resolution methods. Energies **15**(4), 1296 (2022)
27. Parsonson, C.W., Laterre, A., Barrett, T.D.: Reinforcement learning for branch-and-bound optimisation using retrospective trajectories. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 37, pp. 4061–4069 (2023)
28. Petropoulos, F., Laporte, G., Aktas, E., Alumur, S.A., Archetti, C., Ayhan, H., Battarra, M., Bennell, J.A., Bourjolly, J.M., Boylan, J.E., et al.: Operational research: methods and applications. Journal of the Operational Research Society **75**(3), 423–617 (2024)
29. Scavuzzo, L., Aardal, K., Lodi, A., Yorke-Smith, N.: Machine learning augmented branch and bound for mixed integer linear programming. Mathematical Programming pp. 1–44 (2024)
30. Scavuzzo, L., Chen, F., Chetelat, D., Gasse, M., Lodi, A., Yorke-Smith, N., Aardal, K.: Learning to branch with tree mdps. In: Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., Oh, A. (eds.) Advances in Neural Information Processing Systems. vol. 35, pp. 18514–18526. Curran Associates, Inc. (2022)

# A Benchmark problems

## A.1 Set Cover

In the set cover problem, given a set $\mathcal{M}$ of $m$ objects and a collection $\mathcal{S}$ of $n$ subsets $s \subseteq \mathcal{M}\ \forall s \in \mathcal{S}$ with cost $c_s$ such that $\mathcal{M} = \bigcup_{s \in \mathcal{S}} s$, the goal is to find a

subset $\mathcal{S}^* \subseteq \mathcal{S}$ such that $\bigcup_{s \in \mathcal{S}^*} s = \mathcal{M}$ with a minimal cost. It is modeled as a MILP with binary variables $x_s$ that decide whether to select or not $s$ in $\mathcal{S}^*$.

$$\begin{aligned} \min & \sum_{s \in \mathcal{S}} c_s x_s & \\ s.t. & \sum_{s : e \in s} x_s \geq 1 & \forall e \in \mathcal{M} \\ & x_s \in \{0,1\} & \forall s \in \mathcal{S} \end{aligned}$$

All the benchmark instances have 1000 subsets. Easy instances have 1000 objects, medium instances have 1500 objects, and hard instances have 2000 objects.

### A.2 Combinatorial Auction

In the combinatorial auction problem, given $m$ items and $n$ bids, where each bid $j \in \{1, \ldots, n\}$ has a price $p_j$ for a subset of items $\mathcal{B}_i \subseteq \{1, \ldots, m\}$, the goal is to select a subset of bids such that each item is at most in one of these bids while maximizing the value. It is modeled as a MILP using binary variables $x_j$ that decide whether bid $j$ is selected.

$$\begin{aligned} \max & \sum_{j=1}^{n} p_j x_j & \\ s.t. & \sum_{j : i \in \mathcal{B}_j} x_j \leq 1 & \forall i \in \{1, \ldots, m\} \\ & x_j \in \{0,1\} & \forall i \in \{1, \ldots, n\} \end{aligned}$$

Easy instances have 200 items for 1000 bids, medium instances have 300 items for 1500 bids, and hard instances have 400 items for 2000 bids.

### A.3 Capacitated Facility Location

In the capacitated facility location problem, given a set $\mathcal{F}$ of $m$ facilities with fixed operating costs $f_i$ and capacity $u_i$, a set $\mathcal{C}$ of $n$ clients with demands $d_j$, and $c_{ij}$ the costs to serve client $j$ with facility $i$, the goal is to find how to serve all the clients while minimizing the costs. It is formulated as a MILP using two sets of decision variables. The binary variables $y_i$ that decide whether a facility is opened, and the binary variables $x_{i,j}$ that decides whether client $j$ is served by facility $i$.

$$
\begin{aligned}
\min & \sum_{i \in \mathcal{F}} y_i + \sum_{i \in \mathcal{F}} \sum_{j \in \mathcal{C}} x_{ij} & \\
s.t. & \sum_{j \in \mathcal{C}} d_j x_{ij} \leq u_i y_i & \forall i \in \mathcal{F} \\
& \sum_{i \in \mathcal{F}} x_{ij} \geq 1 & \forall j \in \mathcal{C} \\
& y_i \in \{0,1\} & \forall i \in \mathcal{F} \\
& x_{ij} \in \{0,1\} & \forall (i,j) \in \mathcal{F} \times \mathcal{C}
\end{aligned}
$$

Benchmark instances have 100 facilities. Easy instances have 200 customers, medium instances have 600 customers, and hard instances have 1000 customers.

### A.4 Maximum Independent Set

In the maximum independent set problem, given an undirected graph $G = (V, E)$ where $V$ is the set of vertices and $E$ the set of edges, the goal is to select a subset of vertices $V^* \subseteq V$ of maximal size such that there are no edges between each vertex of $V^*$ or $\{(i,j) | \forall (i,j) \in V^* \times V^*\} \cap E = \emptyset$. It is formulated as a MILP using a set of cliques $\mathcal{C}$ such that all edges $E$ are covered by at least one clique. It uses binary variables $x_v$ to decide whether a vertex $v \in V$ is selected in the solution $V^*$.

$$
\begin{aligned}
\max & \sum_{v \in V} x_v & \\
s.t. & \sum_{v \in C} x_v \leq 1 & \forall C \in \mathcal{C} \\
& x_v \in \{0,1\} & \forall v \in V
\end{aligned}
$$

Easy instances have 500 vertices, medium instances have 1000 vertices, and hard instances have 1500 vertices.