HEC MONTRÉAL

Monte Carlo Tree Search as a Primal Heuristic for Integer Programming

par

Vincent Fortin

Mémoire présenté en vue de l'obtention

du grade de maîtrise ès sciences en analytique d'affaires

Département de sciences de la décision

Janvier 2021

Résumé

Dans ce projet, nous explorons la technique de Recherche Arborescente Monte-Carlo (Monte-Carlo Tree Search) pour l’amélioration de la solution primale lors de la résolution de problèmes d’optimisation linéaire en nombres entiers.

Monte-Carlo Tree Search a eu beaucoup de succès récemment pour entraîner des agents à jouer à des jeux de société comme les échecs. Ces jeux ont une structure d’arbre similaire à celle de problèmes d’optimisation en nombres entiers.

Le but de notre travail n’est pas de surpasser la performance des solveurs modernes, mais plutôt d’explorer le potentiel de la méthode et de mieux comprendre son fonctionnement dans le cadre de problèmes d’optimisation en nombres entiers.

Nous utilisons des idées provenant de la technique de séparation et évaluation (branch and bound), cela permettra à nos modèles d’être utilisés pour différents problèmes d’optimisation en nombres entier.

Nous testons la capacité de nos modèles à trouver de bonnes solutions faisables rapidement, en mesurant l’intégrale de la borne primale que nous comparons avec SCIP, un solveur open source complet. Nous travaillons sur des problèmes pour lesquels il est difficile de trouver des solutions faisables, comme le problème d’ensemble indépendant généralisé. Ce choix est guidé par le fait que notre méthode converge lentement et qu'elle pourrait donc performer mieux comparée à des solveurs modernes sur des problèmes plus difficiles. Notre objectif est de trouver des bonnes solutions faisables rapidement. Dans le cadre de ce travail, nous ne travaillerons pas à prouver l’optimalité de ces solutions.

Mots clés : Recherche Arborescente Monte-Carlo, Optimisation linéaire en nombres entier, Séparation et Évaluation, Problème d’ensemble indépendant généralisé, SCIP

Abstract

In this work we are exploring Monte-Carlo Tree Search (MCTS) algorithms to quickly find good primal solutions in Mixed Integer linear optimization problems (MILP). Monte-Carlo Tree Search has had impressive results recently in the field of games like chess, which have a similar tree structure to optimization problems when formulated as integer programs.

Our work is not aimed at outperforming modern solvers, rather it is aimed at understanding how well the MCTS algorithm and its different extensions perform on MILP and if it is a good avenue for further research.

We will be working on problems for which good feasible solutions are hard to find, like the generalized independent set problem. We are working on this type of problem since the MCTS algorithm converges slowly, and we think that it will perform better on harder problems, when compared to commercial solvers like SCIP. Our goal with this algorithm is to find good feasible solutions quickly. For the purpose of this work, we won't focus on proving optimality.

We will be using ideas from the Branch and Bound framework and integer programming to train our Monte-Carlo Tree Search models. This means that our models are problem agnostic and can be trained on any type of integer programming problem.

We will test the capacity of our models to find good solutions quickly, by measuring the integral of the primal bound over the number of linear programs solved and will compare our results with those of SCIP, an extensive open-source solver.

Key words: Monte Carlo Tree Search, Mixed Integer Linear Problem, Branch and Bound, Generalized independent set problem, SCIP

Table of Contents

Index of figures

Index of tables

List of abbreviations

Monte-Carlo Tree Search:	MCTS
Branch and Bound:	B&B
Generalized independent set problem:	GISP
Mixed integer linear program:	MILP
Linear program:	LP
Upper confidence bound for trees:	UCT

Acknowledgements

Working on my master has been a very rewarding process. At first, there were a lot of doubt on my part regarding my ability to complete this project. Thanks to a lot of help from my family friends, and my supervisors, I was able to hand work I am proud of.

First, I would like to thank my supervisor, Laurent Charlin. His machine learning class opened my eyes to the different applications of machine learning, as well as the state-of-the-art methods in various fields. This class motivated me to take on this research project and further my knowledge of machine learning and integer programming. During this project, Laurent helped me see the big picture, always making sure implementation decisions were made with a precise goal in mind.

I would also like to thank Maxime Gasse for his knowledge of integer programming and SCIP. Learning to use SCIP was one of the harder parts of my project and it would have been a lot harder without Maxime. I also learned a lot talking to Maxime, about how to tackle a research problem.

Thank you to all of my friends who pushed me to go climbing, go on walks and generally kept me sane during this challenging year.

Finally, I would like to thank my girlfriend Marjolaine. She has been a large moral support from the beginning, always listening to my problems, pushing me to exercise and cooking for me in times of need.

1 - Introduction

Optimization problems have been studied for many years. They help provide business decisions [1], but are also of general interest to mathematicians and computer scientists [2]. Limited resources have always been at the heart of business decisions in the sense that decision makers need to make the most of the resources they have. Those decisions can often be translated into optimization problems. Before the 1950s [2], very simple hand-crafted heuristics were devised to solve optimization problems. Over time, theoretical methods were devised to help solve optimization problems, however they were impractical without modern computers. With the advent of computers, many challenging problems became solvable given the larger available number of computations per second. This approach of waiting for hardware to become faster only works up to a certain point. Many real-world problems are of combinatorial nature, meaning as they grow in size, the number of possible solutions grows exponentially. Even though computational power has increased in an exponential manner since the 1960s [3], many large optimization problems are still unsolvable and so sheer computation power isn't enough to solve them. We encounter such optimization problems every day, for example, scheduling, routing or packing problems are all part of this family of problems [4].

Our work focuses on using an algorithm which has had a lot of success recently in solving games like chess and Go and applying it to solving combinatorial optimization problems. This algorithm which will be explained in more detail in future sections is called Monte Carlo Tree Search (MCTS).

Along with a classical MCTS implementation, we will be testing different MCTS variations, notably, different diving heuristics for the simulation step of the algorithm. We will also be exploring different methods for candidate score initialization to see if the algorithm converges quicker when it is initialized to better values.

We will evaluate our MCTS algorithms on two types of Mixed Integer optimization problems, namely the set covering problem and the generalized independent set problem (GISP). We will compare the performance of our algorithms with SCIP, a powerful Mixed integer Program solver. Two different configurations of SCIP will be explored, one regular and one eager (aggressive). The eager configuration executes heuristics more aggressively, which increases the speed at which it finds the optimal solution, with less preoccupation to proving its optimality.

For the different diving heuristics, we will be testing random and fractional diving rollouts, as well as a diving policy which branches on variables based on an average of many diving scores. The idea behind different diving methods is that we are looking to speed up the convergence of the best actions for a given state and we are looking to understand which one performs best.

We will also be testing warm starting the node scores, meaning non-uniform initialization. The standard MCTS algorithm converges very slowly and thus giving it a good starting point might speed up the convergence.

In this MCTS implementation, we focus on the primal bound and do not solve the dual program. This means we have no mechanism to prove optimality, which is an integral part of modern solvers. This is why we are testing our algorithm as a primal heuristic, evaluating how quickly we can improve the primal bound.

In this project, we find that the MCTS architecture seems to be suited to finding good feasible solutions mixed integer linear programs. We find that the MCTS algorithm converges, in the sense that the objective function values improve on average, as the number of simulations increases. For the purpose of this project, we are only focussing on the improvement of the primal bound, without solving the dual program to prove optimality.

The convergence speed of the node statistics varies, depending on which rollout policy is used. This is an indication of the importance of using a good rollout policy which is suited to the problem at hand. We find that some rollout policies perform well on the set cover instances, but perform poorly on the GISP instances.

This being said, our implementation of MCTS fails to outperform modern solvers, which is to be expected, since modern solvers are based on decades of research and many years of engineering efforts. The scope of this project is also limited to a master's thesis, which limits the time available for the implementation.

We think that extending our implementation of MCTS could yield results closer to modern solvers like SCIP on specific problems, if more sophisticated techniques are used for different parts of the MCTS algorithm. This is especially true for hard MILP like GISP. Our experiments seem to show that the traditional heuristics used in modern solvers are not well suited to difficult (primal) problems, like GISP. Notably, we found that the margin of victory of the SCIP methods compared to a completely random branching policy (no MCTS) is quite small, on the instances we performed our tests on.

2 – Background

We now discuss the different components of our project, notably integer programming, branch and bound, and Monte-Carlo Tree Search.

2.1 - Combinatorial Optimization

In this section, we define integer programming, branch and bound as well as branching techniques. We also introduce SCIP, the MILP optimization library we use in this project.

2.1.1 - Integer programming

Integer programming is a framework for representing optimization problems which have binary or integer constraints. In this framework, a problem is defined by a list of constraints and an objective function. The canonical form of an integer linear program is:

$$\begin{aligned} &Maximize\ c^T x \\ &Subject\ to\colon Ax \le b, \\ &\qquad\qquad x \ge 0 \\ &\qquad\qquad x \in Z^n. \end{aligned}$$

Here, *c* is the array of coefficients of the objective function, *A* is the matrix of coefficients to the constraints, *b* is the right-hand side array of the constraints and *x* is the array of variables. This set of equations can be shown as a geometrical representation where the feasible solutions are inside the polygon delimited by the set of constraints, at the intersection of whole number lines.

When all of the variables have integer constraints, the problem is an integer optimization problem, while it is a mixed integer optimization problem when only a subset of the variables have integer constraints.

In the above formulation, the objective function $Maximize\ c^T x$ is linear, meaning each variable *x* is only multiplied with its associated coefficient *c*. Similarly, every constraint is also linear. In this case, since the objective function and every constraint are linear, the problem is a mixed integer linear program (MILP) or integer linear program (ILP).

There exist methods to solve non-linear integer programs, however, we won't be discussing those, as we will only work with integer programs.

2.1.2 - Linear programming

In this section we explain the Simplex, an algorithm for solving linear programs without integer constraints. We also describe the dual problem, which is used to prove optimality of MILP problems.

Simplex

The first popular modern method for solving linear optimization problems (without integer constraints) is called Simplex. It is still used today to solve linear problems. This method is attributed to George Dantzig [5]. The idea of the Simplex algorithm stems from the fact that at least one of the optimal solutions must be at one of the vertices (intersection of two or more constraints) of the feasible region. In the Simplex algorithm, starting at one of the vertices, the graph is searched by following neighboring vertices. The neighboring vertex is chosen in such a way to maximize the objective function's value. If none of the neighboring vertices have a better objective function value, the current vertex is proven to be optimal. The Simplex algorithm is at the core of most of the current techniques used to solve linear optimization problems.

Dual problem

The dual problem can be derived from the relaxed linear problem by doing the following:

- Each variable in the primal LP becomes a constraint in the dual LP
- Each constraint in the primal LP becomes a variable in the dual LP
- The objective function in the dual is the inverse direction from the primal (min becomes max and vice-versa)
- The coefficients of the dual objective function are the right-hand side of the constraints in the primal linear problem.

This dual problem has a lot of useful properties. It can be used to prove infeasibility or optimality of the primal (original) problem. We won't go into further detail on ways to prove optimality using the dual bound — the value of the dual solution — since our algorithm does not compute the dual bound.

2.1.3 - Branch and bound

Branch and bound is an algorithm used to solve linear integer optimization or Mixed Integer Linear Problems (MILP). Linear programming problems without any integer constraints can be solved with algorithms like the Simplex algorithm. The Simplex relies on the fact that at least one of the optimal solutions is guaranteed to be at the intersection of two constraints on the edge of the feasibility region. If a variable is restricted to be integer, the optimal solution is no longer guaranteed to be on one of the vertices of the polyhedron. MILP problems are hard to solve because of the fact that we need to find smart ways to work through integer solutions.

Branch and bound is a tree-based algorithm which starts by relaxing (removing) the integrality requirement and iteratively branches (see below) on variables to reduce the feasibility region. Once solutions are found, there is then the process of proving the solution is optimal, by closing the gap between the primal and dual bound. The primal bound is the current best feasible solution, while the dual bound is the optimal solution of the relaxed linear program. In other words, for a minimization problem, the primal bound is the upper bound, while the dual bound is the lower bound.

Branching

The branch part of the algorithm involves choosing a fractional variable to branch on and fixing it to either its floor or ceiling value, by adding an additional constraint of the form $x_i \geq \lceil x_i \rceil$ or $x_i \leq \lfloor x_i \rfloor$. To get all of the fractional variables, the relaxed linear program (along with all of the constraints added when branching) is solved.

The algorithm branches sequentially on open (feasible) nodes until either a pre-set time limit is hit or the primal and dual bounds are equal each another. We will not focus on the second case, since we are only working towards finding good primal bounds.

Each time the LP is solved for a new open node, the following instructions are executed in the 4 possible cases:

1- The current LP's solution is feasible in the original MILP. If the feasible solution is better than the current best feasible solution, the primal bound (if maximizing) is increased.
2- The current LP is infeasible. Nothing more needs to be done.
3- The current fractional LP's objective function is worse than the best feasible solution (primal bound). Nothing more needs to be done, since adding more constraints will necessarily worsen the value of the objective function.
4- The current fractional LP's objective function value is better than the primal bound. New open nodes are created from the current node's candidates

Many different methods exist to choose the open node and the variable to branch on, which won't be discussed further, since our project doesn't directly focus on selecting open nodes. In the future sections, we explain various branching techniques, as this is the main focus of our project.

One of the first mentions of the technique we now call branch and bound is from 1960 (Lang and Doig) [6]. There are two steps to the method they proposed. The first step is to relax all of the binary and integer constraints, meaning all of the binary and integer constraints are removed, allowing the variables to be continuous. The second step is to successively set bounds on each of the variables. To find the bounds for a variable, they proposed a method to create a 2d polygon using the possible integer values for that variable. The bounds are then set using the objective function value associated with those points. They show that by fixing variables in succession, the resulting linear program is either feasible (with all integer constraints) or infeasible (violating one of the original constraints). They also discuss the tree structure that is obtained from successively branching on different variables. Since the article was published in 1960, they comment on choices which need to be made regarding how far down the tree to search. This is because of their limited access to memory.

Following this article, many researchers used Branch and Bound to solve hard optimization problems which previously did not have good exact solution algorithms to. One such problem is the Traveling Salesman Problem (TSP), which is a well-known NP-Hard problem, with many real-world applications (vehicle routing, circuit drilling and more) [7]. The goal of the TSP is to find a path which minimizes the total distance traveled when visiting a list of cities, given the distance between each pair of cities. Researchers have worked on TSP and other NP-Hard problems for many decades, finding different heuristics based on the knowledge and available resources at the time.

Due to the limited computational resources at the time, early applications of the Branch and Bound for the TSP were based on human devised branching strategies. For example, using Branch and Bound, John D. C. Little found that TSPs of up to 40 nodes could be solved to optimality [8], as compared to TSPs of up to 12 nodes before Branch and Bound. This being said, at the time, no branching technique was general purpose enough to solve large instances of a variety of different types of problems.

2.1.4 - Branching techniques

Other than computational improvements, which help solve large MILPs, there are two major sources of improvements: 1) the branching method, the bounding method and the tree search and 2) the order in which the nodes are explored. Here, we focus on branching techniques, as they will be the most important part of the algorithm for our project.

In 1971, M. Benichou et al. [9] put forward a branching technique named *pseudo cost*, which tracks the change in objective function after a particular branching decision. Pseudo costs are initialized uniformly and will change based on how a branching decision affects the value of the objective function. More specifically, for each variable, Δ_j is the objective function value change per unit change of variable *j*, ς_j is calculated for every variable *j*. ς_j has both a positive and a negative value associated with branching on the ceiling and floor of variable *j* respectively.

$$\varsigma_j^- = \frac{\Delta_j^-}{f_j^-}, \varsigma_j^+ = \frac{\Delta_j^+}{f_j^+}$$

f_j represents the difference between the fractional variable and the floor or ceiling of variable *j*. The pseudo cost up ψ_j^+ and pseudocost down ψ_j^- is the average of ς_j^+ and ς_j^- respectively, over all branches done on variable *j*.

After a few branches, this will produce good branching decisions. This being said, at the start of the Branch and Bound algorithm, there are no branching decisions to base the pseudo cost statistic on.

In 1995, D. Applegate and al. [10] introduced *strong branching*, a branching strategy which tries to find the best branching candidate by testing all (in the case of full strong branching) or a subset of the candidates to find which gives the best dual bound improvement. This is done by simulating branching by fixing each of the variables to both of their bounds subsequently and solving all of the relaxed LPs. Strong branching has been shown to provide smaller trees (10), however, it requires solving a lot of LPs, which can be slow if used exclusively. Strong branching can be used in combination with other heuristics for improved results. For example, using strong branching early in the tree, for the most important branching decision and transitioning to faster strategies, like pseudocost, which becomes more useful as we gather more information on previous branching decisions.

Since 1995, new branching techniques have come out, however none of those techniques is strictly better in every situation. Recently machine learning approaches have tackled the branching problem, for example, Gasse et al. [11] introduced a *graph convolutional network* approach which learns to imitate strong branching statistics. This method outperforms state-of-the-art solvers, on a few benchmarks which use a variety of hand crafted heuristics.

2.1.5 - Diving heuristics

Primal heuristics can be seen as cheap search routines designed to quickly find feasible solutions during the branch-and-bound process and are typically called whenever a new node is created, based on a priority list as well as different statistics. One family of such heuristics are the so-called diving heuristics, which consist in expanding a single path down the branch and bound tree in the hope that it will lead into a feasible solution. Once a diving heuristic is executed from a certain node, branching decisions are made in succession, based on a certain criterion (fractionality, number of variable locks, etc.), which can be seen as an alternative to regular branching rules. This is done until a feasible or infeasible LP is hit, or until a pre-set criterion is hit (max depth, number of solved LPs, etc.). In our work we will be interested in diving heuristics for enhancing our MCTS algorithm during the rollin and rollout phases, as will be discussed in Sections 3.4 and 3.5.

There are many diving heuristics available in SCIP. Most of them are explained in Tobias Achterberg's thesis [12], which covers all of the components of SCIP in detail.

SCIP is implemented in such a way that it is not trivial to simply call a specific diving heuristic at a certain point in the branch-and-bound tree, which is required for our project. Instead of calling the heuristics implemented in SCIP, we implement some well-known diving heuristics, allowing us to execute them at any point in the tree. This however limits the number of diving heuristics which could be implemented and tested, since some of them are quite complex. The following section describes the diving heuristics we implemented and used.

For every diving heuristic below, the branching candidates are the variables with fractional variables in the current LP.

Random diving: This is the simplest diving policy. Both the variable and branching direction (up or down) are chosen randomly.

Fractional diving: This is another simple diving heuristic. The variable to branch on is chosen as the variable whose fractional value is closest to an integer. This variable is fixed to its closest integer. The idea behind this technique is that a variable whose fractional value is close to an integer is likely to end up at this integer value in the optimal solution. The following formula shows how the variable closest to an integer is chosen.

$$argmax_j \ (max(x_j - \lfloor x_j \rfloor, \lceil x_j \rceil - x_j))$$

In this formula, x_j represents a branching candidate variable.

Line search diving: The idea behind this heuristic is to compare a variable's value in the relaxed LP at a certain node in the tree with the same variable's value at the root of the tree, also in the relaxed LP. The goal of this heuristic is to continue pushing the variable in the direction it has been going since the root node. Mathematically, we would like to choose the variable that minimizes the following distance ratio.

$$\frac{x_j - \lfloor x_j \rfloor}{(x_R)_j - x_j}, if\ x_j < (x_R)_j \text{ and } \frac{\lceil x_j \rceil - x_j}{x_j - (x_R)_j}, if\ x_j > (x_R)_j$$

In those distance ratios, $(x_R)_j$ represents the value of variable *j* in the relaxed LP at the root of the tree, while x_j represents the value of the same variable in the current LP.

Coefficient diving: This heuristic branches on the variable which minimizes the number of variable locks, as long as it has at least one variable lock. A variable lock is the number of constraints which are violated by fixing a given variable to either its ceiling or floor value. For each variable, there is both a number of variables locks up and down. The number of variable locks associated with every branching candidate is available in SCIP.

Pseudocost diving: As noted above, the goal of this heuristic is to branch on the variable which has the greatest improvement of the objective function's value per unit change of the variable's value. The first step is to find in which direction to branch each variable if it had the highest pseudocost value. This is done by looking at different conditions in succession.

The first step is to compare the value of the variable at the current node with the value of the same variable at the root node (similar to line search diving). The variable is branched upwards (or downwards) if the variable shows a strong indication that it should be pushed upwards (or downwards). Here is the formula used to decide if the variable shows a strong indication in branching in either direction: branch upwards if $x_j > (x_R)_j + 0.4$ or branch downwards if $x_j < (x_R)_j - 0.4$. If $|x_j - (x_R)_j| < 0.4$, no branching decision is taken based on the above criteria.

If this criterion does not show a strong enough indication of branching either up or down, the second step is to look at the fractionality of the variable, branching upwards if $\lceil x_i \rceil - x_i > 0.7$ and downwards if $x_i - \lfloor x_i \rfloor < 0.3$.

If those the previous two conditions do not yield a decision, the third step is to evaluate the pseudocost value, branching up if the pseudocost up $\Psi_j^+ = \frac{\sigma_j^+}{\eta_j^+}$ is higher than the pseudocost down $\Psi_j^- = \frac{\sigma_j^-}{\eta_j^-}$ for variable *j*. σ_j^+ represents the sum over all branches of variable *j* of the objective function gain per unit change in variable *j,* as explained in *section 2.1.3.* η_j^+ represents the number of problems which includes the branching of *j* in the upwards direction.

This value is initialized uniformly and only starts making sense after statistics are gathered from branching decisions.

Diving statistics

When executing diving heuristics, a variable is chosen by minimizing (or maximizing) a certain criterion, like pseudocost, line search, etc. For our project, we will mostly be using the value of the statistic calculated (number of variable locks for example), as opposed to only being interested in which variable has the minimum value. We will call this value *diving statistic*. We will be creating a new branching rule by averaging different diving statistics. This will be explained in more detail in *section 3.4*.

2.1.6 - The SCIP solver

SCIP (Solving Constraint Integer Programs) is a very extensive optimization package for solving linear and nonlinear optimization programs. There are many options to create different branching rules, heuristics and more. It can also be used to solve the relaxed LP problem, find branching candidates and more. The goal of SCIP and other solvers is not only to find good solutions, but also to prove their optimality. We will explain how this fact affects the SCIP models we chose to compare our algorithm with.

In the following paragraphs, we will explain the main components of SCIP with an emphasis on how we will use each of them.

Given a MILP to solve, SCIP first runs a *presolving* phase. Presolving is the operation where the instance at hand is simplified, by removing redundant constraints. The aim of presolving is also to understand the structure of the problem, which could simplify future solving steps. This is an important step part of modern solvers, with many different presolving techniques, however, we will be disabling this operation, since we are only measuring the impact of the solving algorithm itself.

Heuristics are another big part of SCIP and solving MILPs. When selecting a node to be processed, SCIP needs to choose how it is processed. SCIP can either branch on one of the fractional variables, execute one of its many heuristics, or solve the dual problem, amongst others. The frequency at which it executes one of the heuristics depends on a priority list (which can be changed), as well as statistics calculated at the node being processed. SCIP has an *aggressive option* when it comes to heuristics, which means it will execute heuristics more often. For example, SCIP will call diving heuristics, instead of simple branching rules. This is presumably to improve the primal bound faster, with less intent on reducing the primal-dual gap.

Since we are only focusing on the primal bound for this project, we could expect our results to be closer to those of SCIP aggressive, since it assigns less importance to the dual side of the problem. This being said, we will be comparing our MCTS approach against both: 1) SCIP aggressive and 2) the regular version of SCIP (both without presolving).

Another component of SCIP which is important when solving hard problems is the frequency at which the relaxed LP problem is solved. This is mostly a concern when executing a diving heuristic, since we don't need as much information on nodes between the start of a dive and the end of the dive. A naïve approach would be to solve the LP every time we get to an intermediate node during the execution of the diving heuristic. SCIP does not do that. It instead solves the LP only once every few nodes. The idea behind the strategy of not solving the LP at every node is that branching on a variable once may have little impact of the relaxed LP problem. Furthermore, solving the LP is one of the most time-consuming steps in solving MILPs.

From the node where SCIP executes a dive, it will choose the 10 variables[1] to branch on, based on the statistics from the current node. This means that if it executes a fractionality diving heuristic, it will choose the 10 most fractional variables and branch on them in succession. It is possible that after all of those branching decisions, the new LP is infeasible since branching decisions are made without the information from intermediate nodes. We can see that there is a trade-off between the speed at which the algorithm runs and how often infeasible nodes are encountered. For example, solving the LP after every branching decision would mean infeasible nodes are rarely encountered, while at the opposite end of the spectrum, solving the LP very rarely would mean a very fast but very impractical algorithm. We implemented a similar mechanism for the rollout step of our MCTS algorithm. We discuss this mechanism in *section 3.3*, our choices regarding how often the LP is solved, as well as the impact on performance, both in terms of speed as well as the quality of the solutions found.

2.2 - Monte Carlo methods

Monte Carlo methods in general have been devised in the last 1930s and early 1940s [13], as a way to perform numerical approximations instead of finding exact solutions to complex problems, like differential equations. The general idea behind Monte Carlo methods is to perform many simulations based on random sampling in order to estimate a certain value.

We will mostly be discussing Monte Carlo methods in the context of games, from which we will be transitioning into the problem at hand, solving mixed integer optimization problems.

Since the end of 1990s, Monte Carlo based techniques have been used to find good policies in various two-player games. In 1999, researchers from the University of Alberta [14] modelled a reinforcement learning agent to play the game of poker (an imperfect information game). They used Monte Carlo simulations over the estimated probability distributions of the opponent's actions and cards to find policies that map each state to a distribution over the set of possible actions. This is done instead of an exhaustive search of all possible actions for the opponent since the search tree is too big and prohibitive to explore. Solving for likely actions will give the agent a good enough idea of the best action to take.

Using simulations to solve imperfect information games is natural, since it is not possible to find a single best action to take in a given state of the game. This is because the opponent's hand is unknown and so probability distribution needs to be assigned over possible hand combinations.

Other work around 2005 focused on the game of Go. Go has been the focus of much research since it is a very complex game, with a game tree orders of magnitude larger than other classic games such as chess. Work from Bruno Bouzy used Monte Carlo simulations from the current state of the board to the end of the game to select the best move. Because of the complexity of the game, they focused on pruning techniques [15], which are important when working with games which have a high branching factor. The idea behind their technique is to gather statistics on the moves which have been randomly selected and prune the moves which have a low

[1] Here the number 10 is an example of the number of branching decisions before solving the LP.

probability of leading to a good outcome. In their experiment, they don't actually calculate a probability of action A being better than action B, however they calculate an expected outcome for each move as well as a confidence interval. This confidence interval is calculated using the expected outcome and a confidence value which is set by the researchers. For example, for an average value of 5, and a confidence value of 2, the confidence interval would be 5 ± 2 = [3, 7]. They say a move is significantly worse and thus removed from the possible moves if the upper bound of this move is worse than the lower bound of every other move. Using this logic, parts of the game tree could be removed early even though they could be proven good once the tree is explored in more depth, which is an undesirable behavior.

In the work described above, there is no concept of exploration and exploitation. This could enable the agent to periodically try moves which have worse expected value in order to make sure their value has not changed, given new information available.

2.2.1 - Monte Carlo Tree Search

The Monte-Carlo-Tree-Search method introduced by Rémi Coulom in 2006 [16] as an improvement of previous Monte-Carlo-based methods. The goal of the MCTS algorithm is to find the best action to take, given state of the board (in a game). The idea behind this algorithm is that starting at a certain state of the board, many simulations of the possible actions are executed until the end of the game. Simulations can be random or based on more complex policies. Once the end of the game is reached, a statistic is calculated which represents the quality of each action taken.

Here are the 4 steps to the classical implementation of MCTS. We will be using the single player version of this algorithm; however, adapting it to 2-player games is straightforward.

1- **Selection**: From the root node of the MCTS tree, choose a child node from all possible candidate actions, using a statistic like the upper confidence bound for tree (UCT, explained in the next section). Execute the chosen action. Change the state of the board and repeat this selection step until a candidate which has not been created before is selected. We also call this step rollin.

2- **Expansion:** Create a node and initialize it.

3- **Simulation:** We use rollout and simulation interchangeably for this step. From the expanded node, play the game until the end. This rollout can be done using any policy (e.g., random, learned, or any other expert policy).

4- **Backpropagation:** Look at whether the simulation has ended in a win or a loss. Different games have different definitions for a win, but the result must be binary (for the classical implementation of MCTS) and can't simply be a continuous score like a number of points for example. If the game does not have a binary reward score, the score must be transformed to be binary. This information (win or loss) is then backpropagated up the tree, meaning every parent of the leaf node will have its statistics (number of visits, number of wins) incremented, based on the result of the simulation.

The following figure shows all of those steps for a simple game.

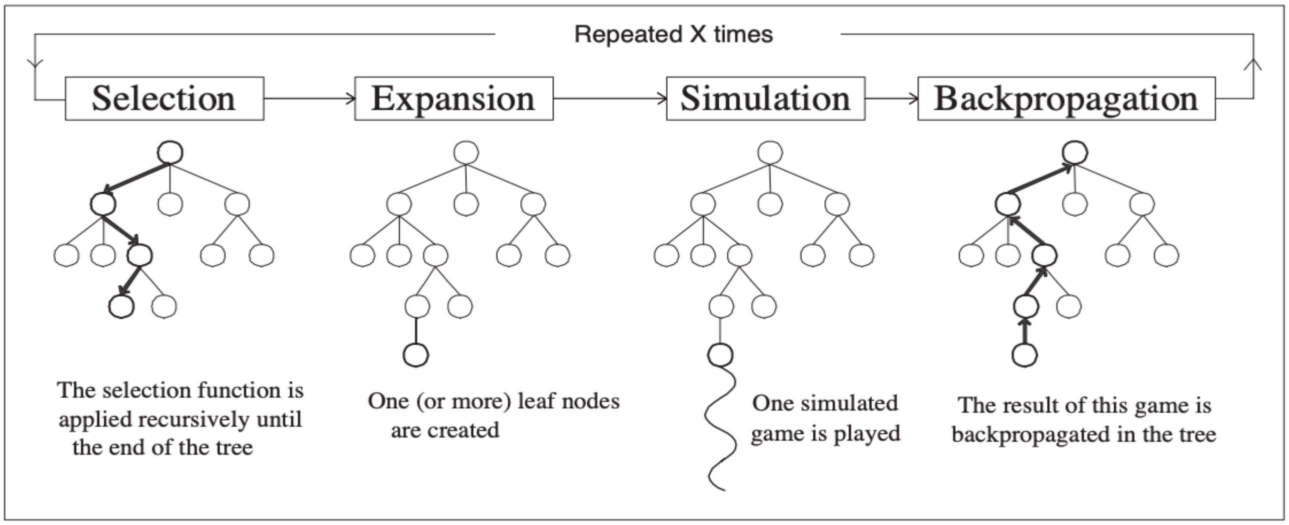
Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until the end of the tree
One (or more) leaf nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 1: MCTS algorithm steps from (16).

We discuss differences between the classical implementation of MCTS and our implementation in *section 3: Monte-Carlo tree search for integer programming*.

When Coulom introduced the MCTS algorithm for the game of Go in 2006, he proposed a method for node selection (in the selection step described above) based on the probability of each node being better than the current best node. The following equations represent this probability for each of the candidate moves:

$$u_i = exp\left(-2.4\frac{\mu_0-\mu_i}{\sqrt{2(\sigma_0^2+\sigma_i^2)}}\right)+\epsilon_i\,, \forall\; candidate\; move\; i\,.$$

In those equations, each move is ordered from best to worse, meaning $u_o > u_1 > u_2 > \cdots > u_N$. Each μ_i (including μ_0) represents the average score (end of game score) when move *i* is chosen, while σ_i^2 represents its variance. When we talk about the score in this context, we talk about the number of points[2] at the end of the game of Go.

The last part of the equation is what they call an urgency coefficient, $\epsilon_i = \frac{0.1+2^{-i}+a_i}{N}$, where a_i is a boolean representing "Atari" moves[3], which are a set of human-defined good moves requiring further actions to materialize their value. This urgency coefficient is added to avoid *good* moves from having a low probability of being the best move if they require additional moves to materialize their value.

The probability u_i of each move being better than the current best move (u_0) is calculated by sampling 1,500 board positions from self-play games (games played against itself), and randomly

[2] Score is calculated as the number of stones a player has on the board plus the number of board intersections surrounded by a player's stones.

[3] Here the name Atari is not related to the game console Atari

playing out the game many times to the end. Once a game is over, the score (number of points scored in the game of Go) is extracted, which then updates the mean μ_i and standard deviation σ_i^2 statistics. When updating the statistics for a given move using u_i, intermediate moves (between start and end of game) gather statistics representing an estimated mean and standard deviation, which will be used at game play.

Using this method, every simulation brings us closer to the optimal policy, since none of the moves are pruned. Because of memory limitations, it would be impossible to keep a large game tree in memory for a game like Go. This means that many states of the tree will be left unexplored. When put in front of an unexplored state, this MCTS method is left with no other choice but to select a move randomly.

Since this initial version of MCTS relies on random rollouts, it fails to outperform, on a large sample of games, the then-state-of-the-art Go program, GNU Go, which relies on human knowledge of the game.

The ultimate version of MCTS for games with high branching factor was introduced by DeepMind, with Alpha Go [17] and later Alpha Zero [18]. They use a MCTS-like approach to progressively update the probability distribution of each possible move at a given state of the board.

The main improvement from earlier methods is that they use a two-step process for node selection (both during selection and simulation step); first learning by imitating expert games and second learning through self-play.

In the first step, they train a neural-network-parametrized policy (*policy network*) to imitate expert players by predicting moves which are the most likely to be played by expert human players. This policy network is used for the simulation step of MCTS, described in *figure 1*.

The second step uses a value network which learns to predict the outcome at the end of a game, from any game state. This value network is used for node selection in the selection step of MCTS and is trained by playing against itself.

At prediction time (during a game, against itself or a human opponent), they run a MCTS from the current board position to select the best action. For the selection step of the MCTS algorithm, a node is chosen using a combination of the value network's prediction and the prior probability of each action (given by the policy network), normalized by the number of visits to favorize exploration. Here is the formula used to select the node.

$$a_t = argmax_a(Q(s,a) + u(s,a))$$

In this formula, *t* represents the iteration number of the MCTS algorithm, *s* and *a* represent the state and action respectively. $Q(s,a)$ is the prediction of the value network, while $u(s,a) = \frac{P(s,a)}{1+N(s,a)}$ with $P(s,a)$ being the prior probability of the action being chosen, given by the policy network and $N(s,a)$ being the number of visits of this state action pair. Since $u(s,a)$ incorporates the number of visits, many actions will be explored, since $u(s,a)$ will decrease as the number of visits increase, allowing worse nodes (according to the value network) to be selected.

After enough simulations, the estimated values of the candidate actions at the root of the tree (current state of the board) should converge to values close enough to the true (optimal) value of each action. This is not to say that AlphaGo plays optimally, however with a large number of simulations, the value network should make increasingly good predictions of the value of the state-action pair, which means a_t will converge to the best (theoretical) move very often.

Alpha Go beat the best human player in the world, proving the efficacy of this type of method for the game of Go. The game of Go was previously thought to be far from being played at a professional level by AI agents [17].

DeepMind followed AlphaGo with AlphaZero, which is similar, but does not use any prior knowledge of the game, meaning that there is no policy trained to imitate expert moves. This Alpha Zero outperforms prior versions of Alpha Go.

2.2.3 - Upper confidence bound for tree (UCT)

UCT is a statistic which can be used in the selection step of the MCTS algorithm. UCT was introduced by Levente Kocsis and Csaba Szepesvári in 2006 in the context of multi-armed bandit problems [19], a classical problem in mathematics where the goal is to maximize payout of “slot machines” with unknown expected payouts. The idea is to balance exploration and exploitation, meaning how much to explore the different to improve the expected payout estimates. Too much exploring reduces the number of exploitation moves and reduces the total payout. This trade-off is at the heart of the UCT formula, which is as followed, for a specific node.

$$UCT = \frac{w}{n} + c\sqrt{\frac{ln\,N}{n}}$$

In this formula: *w* represents the number of wins, *n* the number of visits, *c* is the exploration ratio and *N* is the number of visits of the parent node.

The first part of the UCT formula $\frac{w}{n}$ is the exploitation part of the algorithm. It relies on the percentage of wins, without taking into account the number of times other nodes have been visited, which is represented by *N*. The second part represents exploration, meaning a higher value of *c* favors exploration.

In the article, they talk about the best value in theory for the exploration ratio, which is $\sqrt{2}$. We will be using this ratio as the default value for our algorithm. We will talk more in detail about this decision and other values which were tested in *section 4.4*.

UCT initialization
UCT scores will be initialized uniformly at 0. We will explain the reasoning behind this initialization in the following paragraph.

Here is how the UCT statistics will look like for the first few visits:
First visit: When visiting a node for the first time, all of its children will be initialized with 0 wins and 0 visits, and the only way to avoid divisions by zero in the UCT formula is to initialize the number of visits to a small non-negative value (e.g., 10^{-16}). At this first visit, if N=0 (N is incremented after the dive), n= 10^{-16} and w=0, we will have problems because of $ln\ 0$ which is undefined. To mitigate this problem, we initialize N to 1, which naturally initializes $UCT_i = 0, \forall i \in candidates$. The first candidate to be chosen will be the first candidate in the list (or any other, since they all have the same value of UCT).

Second visit: After the first visit, the visited candidate will have the following values: N=2, n=1 and either w=0 or w=1, which means that this candidate will have $UCT\ = 1 + \sqrt{2ln2}\ if\ w = 0\ or\ UCT = \sqrt{2ln2}\ if\ w = 1$, while the rest of the candidates which have not been visited will have n=10^{-16}, w=0 and N=2, which means $UCT\ = \sqrt{\frac{2ln2}{10^{-16}}}$, which is larger than both $1 + \sqrt{2ln2}$ and $\sqrt{2ln2}$. Because of the initialization of n=10^{-16}, the value of UCT will always be larger for candidates which have not been visited, meaning that each candidate will always be visited at least once before visiting any other candidate more than once.

2.3 - Reinforcement learning and MCTS for Mixed Integer Linear problems
Recent work focuses on deploying machine learning to solve MILP problems. Here, we survey work that uses MCTS or reinforcement learning to solve integer optimization problems, since this is close to our work.

Some of the work focuses on using the structure of the problem to solve them using reinforcement-learning approaches. Irwan Bello et al. [20], focus on the TSP. For this specific application, the problem is formulated in such a way that the goal is to order a list of cities in such a way that the sum of the distances between each city is minimized. They used a pointer network [21], which is an encoder-decoder network which reads a sequence of cities and outputs them in an order that satisfies the problem definition. Using this architecture, they outperform other techniques on a variety of different sizes of TSP problems. This type of architecture is highly dependent on the type of problem and does not generalize to any other problem type. This is because pointer networks perform well on tasks which require arranging items in a specific order, which is exactly the problem definition of the TSP. Pointer networks would not be suited to other types of problems which require choosing elements from a set (set cover for example) instead of ordering them.

Other reinforcement-learning approaches leverage knowledge of integer programming instead of leveraging the structure of the problem, like Irwan Bello et al. did for the TSP. T. Yunhoa et al. [22] train an agent to choose which cutting plane method to use, given a certain state, which in this case is the feasible region. They used an attention network [23], which learns cuts in an order-agnostic manner. This is important because the order of the constraints or the candidate cuts does not affect the feasible space. This will also naturally solve the problem of the varying input size of the reinforcement learning agent.

For work on using MCTS for optimization problems, K. Abe et al. [24] introduced a method similar to Alpha Go Zero, which they call CombOpt Zero. They use the MCTS algorithm on a variety of graph-based problems, like Minimum Vertex Cover, Maximum Clique and others. Their algorithm uses the graph structure of the problems to choose possible actions for a given state. In this case, a possible action could be adding or removing an edge in the graph. Along with the MCTS algorithm, they use state-action pairs from all of the rollouts, along with their expected value to update the rollout policy. They also test the MCTS algorithm with a Graph Convolutional Neural Network, which they say improves performance compared to other state-of-the-art graph-based algorithms on most of the problems they studied. The idea behind this technique is to run MCTS on a variety of instances to learn a (generic) rollout policy and then, when testing on a new instance, execute the MCTS with the learned policy to find the optimal solution.

Our work is different from CombOpt Zero. We use the linear program and the branch-and-bound algorithm instead of using the graph-based structure of the problems. The main advantage of CombOpt Zero is that by using the graph structure of the problem, they can train models which are adapted to a specific class of problems (graph). The CombOpt Zero method works well on graph problems because the action state is easy to find for any graph (add or remove vertex, add or remove edge). This means no LP needs to be solved before every decision to get the state, which is computationally expensive.

The advantage of our method is that it works on any linear program, as opposed to only on graph-based optimization problems. The downside is that for each decision taken by our model, we need to solve the LP to get the list of candidates to branch on.

3 - Monte-Carlo tree search for integer programming

This section discusses our contributions, as well as some variations to the classical MCTS including different reward allocation, our own average diving statistic and methods for non-uniform node statistic initialization.

3.1 - State and action definition

When solving MILP using MCTS with Branch and Bound, we define the actions as the branching decisions at a certain node in the Branch and Bound tree and the state of the game as the current linear problem, along with all the constraints added from previous branching decisions. A branching decision includes both the choice of variable to branch on and the branching direction (that is, tightening the variable's upper or lower bound).

3.2 - Reward allocation

For the simulation step of the algorithm, we define the end of a game as reaching either a feasible integer solution or an infeasible LP. MILPs can be thought of as win/loss type of problems. A win comes from finding a feasible solution and a loss implies ending up at an infeasible node. This being said, on problems like set cover, we will practically always end up at a feasible node and never get to an infeasible node. This means that we would practically always assign a win at the end of a game. If we only allocate wins, the UCT statistics will stay close to a uniform distribution, and every node will have a 100% win percentage. Having uniform UCT statistics renders the UCT statistics useless. This is because none of the candidates will converge to a higher UCT statistic, which means no candidate will be chosen more often than any other candidate. Having a method which yields uniform UCT statistics would yield results similar to randomly choosing candidates.

The other difference between MILP and games like Go is that in Go, the score at the end of the game isn't meaningful, as long as the game ends in a win. In the case of MILP, finding a feasible solution is not enough. We want to find good feasible solutions, meaning the score (objective function's value) is important for us.

To mitigate those problems (uniform UCT statistics and importance of finding good feasible solutions), we decided to run steps 1-3 (selection, expansion, simulation) of the MCTS algorithm more than once (4 times in our case) before crediting a win and running step 4, backpropagation. After each iteration of step 1-3 of the MCTS algorithm, a temporary loss is assigned to the node being processed. In practice, this loss is assigned by incrementing the number of visits without incrementing wins. This loss is backpropagated up to the root of the tree.

Assigning a temporary loss avoids looping through the same node 4 times. This is because each loss will necessarily reduce the chosen node's UCT statistics. Once a node has been assigned a temporary loss, it will likely not be chosen again within the list of 4 rollins and rollouts.

In theory, it is possible for a node to be chosen for all 4 iterations of the MCTS. This happens when node statistics have converged. As the number of simulations tends to infinity, a single loss added to the best node does not affect the win ratio significantly enough to change which node is selected at the selection step of the MCTS algorithm. This does not happen in our case, since we are not reaching a large enough number of simulations. In rare instances, a node is chosen twice within the 4 MCTS iterations, however, this only happens once the total number of simulations is large. Having nodes be chosen more than once within a set of 4 nodes isn't an undesired mechanism, since only the best nodes will be chosen multiple times.

The win will be credited to the node with the best feasible solution. As with temporary losses, this win is backpropagated up the tree. Since all 4 nodes have been assigned a temporary loss by incrementing visits, only the number of wins for the winning node needs to be incremented and backpropagated.

This win assignment strategy mitigates the uniform UCT statistic problem because only 1 out of every 4 simulations will yield a win, which means the UCT statistics should converge to candidates which yield higher objective function values.

As stated previously, the simulation step of the MCTS algorithm can be done with many different branching policies. We will be testing different policies using diving heuristics.

The game of Go is a multiplayer game, which is not the case for mixed integer linear problems. Because of this, Alpha Go runs a MCTS every time its opponent has finished playing its move, meaning the goal of the algorithm is to find the single best move given the state of the board.
In our case, we will only run a single tree search from the root node until a pre-set amount of time or number of LP solves is reached. The goal of the MCTS in our case is twofold. First, we want to direct our search to the best areas in the search space, meaning over time, the average solutions are better and better and our probability of finding the optimal solution is higher. The other benefit of MCTS in our case is that from the start, we can have a non-zero probability to hit the optimal solution, even when using random rollouts.

3.3 - Early stopping

In our current implementation of MCTS, with 4 rollouts before assigning a win, we need to get to a feasible solution before assigning a win to one of the nodes. To speed up the algorithm, one workaround to the classical pruning of nodes in B&B would be to stop solving the LP if the relaxed solution is worse than the best feasible solution inside the list of 4 rollouts. For example, if the first rollout yields a feasible solution with an objective function value of 1,000 and during the second rollout, we are at a point where the LP has a worse fractional value than 1,000, we can stop solving the LP, since we can't get a better solution, which means that no win will be assigned to this node anyways. We will call the process described above of stopping the dive before reaching a feasible solution *early stopping*.

We implemented early stopping late in the process and only implemented it for the GISP (described in *section 4.1.1*) instances. This is because of the fact that set cover (described in *section 4.1.2*) instances results had already been aggregated and re executing the algorithms on the set cover instances would have too time consuming.

Anecdotally, it seems to increase the total number of MCTS iterations by around 2% for a given number of LP solved. A MCTS iteration is the execution of every step of the MCTS algorithm once. More MCTS iterations are executed since some simulation steps are stopped before finding a feasible solution. This was tested by running the same algorithm on the same problem with, one with early stopping and the other without. This is a pure improvement in performance since it saves unnecessary LP solves at a very minimal cost.

We decided to include it in this project even though it was only used for GISP and not set cover because it is an important feature for future implementation of MCTS using branch and bound, if using a similar win allocation.

3.4 - Average diving

We created this heuristic. It stems from the idea that we could use the diving statistics from a large number of diving heuristics and build a neural network which would create a composite score based on all diving scores. This neural network could predict the probability of a given branching decision leading to the optimal solution. We encountered two issues in creating this aggregated statistic.

First, as stated previously, gathering many diving statistics is far from easy, which limits the number of statistics to include in this model.

The second issue is the complexity of building a dataset to train such a network. One way to build this dataset is to solve a MILP and get the set of branching decisions which led to the optimal solution. In SCIP, getting intermediate nodes is not trivial and we did not have enough time to implement a workaround.

Instead of implementing a neural network, we decided to simply average a few diving scores and use them as a new diving heuristic. Along with fractional diving we implement line search, coefficient and pseudocost diving. We averaged all four of the previously mentioned heuristics. The following section explains different considerations when averaging diving statistics.

Considerations when averaging diving statistics

When averaging the diving statistics, we need to make sure every statistic is standardized. We wouldn't want for example to average a statistic where branching is done on high values with another statistic where branching is done on low values. This is the case for pseudocost, where we branching is done on the variable with the highest ratio of objective function improvement per unit change in the variable's value. For the other diving heuristics we implemented, we are looking to branch on the minimum value, which is the opposite of pseudocost. To eliminate this problem, we transform the pseudocost value by applying a multiplicative inverse, where $newPscost = \frac{1}{pseudocosts}$. This step will transform large values of pseudocost to a corresponding small value. This step would not be necessary if we were to train a neural network to assign weights to each of the diving statistics, since the network could predict weights which would reflect the fact that for certain statistics, high values are good, while for others, low values are preferred.

Further, we transform all of the statistics into a categorical distribution over the statistics. We do this so that there isn't more weight put on one of the statistics than another one. This can be done by either dividing by the sum of the diving statistics, by applying a softmax function ($standardizedStat_i = \frac{exp(divingStat_i)}{\sum_{i \in cands} exp(divingStat_i)}$) or any other function which transforms values to a list where each value is between 0 and 1 and the sum of all values is equal to 1.

The final consideration to keep in mind is regarding the coefficient diving statistic, for which we want to branch on the minimum non-zero value. In this case, we need to replace those zeros to a large value so that it is considered a poor branching decision, instead of a good one. We chose to change the values from zero to the maximum value of the coefficient diving statistic.

3.5 - Warm starting nodes

In the simplest version of the MCTS algorithm, when a new node is created, all of its children's statistics will be initialized uniformly. This works well if we can execute enough simulations for the nodes to converge, however, it can take quite a while before the statistics start converging. The convergence is slow because the algorithm needs to explore each node at least once (often more depending on the exploration ratio). Ideally, we would initialize the nodes using prior information on the state of the problem, which could speed up convergence. In theory, it would also be possible to initialize nodes in such a way that the worst candidate would not need to be visited.

Since the UCT statistics are only based on the number of visits and number of wins, we can't simply initialize the nodes with a list of statistics and update them in subsequent simulations without also initializing the number of wins and the number of visits. We first need to choose the number of visits to assign to the initialized nodes. This value represents how strong we want our prior to be, meaning how much weight we want to put on the initialization statistics. A large number of visits would represent a stronger confidence in the statistics. This value is a hyperparameter. We tried values of 1 and 5 for this prior. Those values are low because of the fact that the optimization problems we chose to work on have high branching factors and the UCT statistic is based not only on the number of visits at each of the candidate nodes, but also the total number of visits of the parent node. If initializing each node with 5 visits, we also need to initialize the parent's number of visits to the sum of all of those visits which would be 5000 if our problem has 1000 candidate at a particular node. A large prior means the UCT statistics will converge very slowly if the initialization is done poorly.

Since we need the number of wins to calculate the UCT score, we initialize the number of wins in such a way that $targetUCT = \frac{wins}{nbVisits}$ which means that $wins = targetUCT * nbVisits$, where *targetUCT* represents the values which we want to initialize the UCT statistics to. In this scenario, the number of wins would be fractional, but this does not change how UCT is calculated.

In our case, we will initialize the node statistics as the average diving statistic explained in *section 3.4*. This initialization relies on the fact that this average diving statistic is informative, which has yet to be proven. We will evaluate the value of this node initialization technique in the results section (*section 7*).

Because of the nature of diving statistics like pseudocost and linesearch, which are only non-uniform after the root node, the initialized statistics (using warm start initialization) will become more and more informative the deeper we get in the tree. This is because the node statistics are initialized as the average diving statistic, which include statistics (pseudocost and line search) which become more and more informative further from the root node.

The flip side is that the nodes closer to the root will be initialized closer to uniform and less informative, since 2 out of 4 of the diving statistics will be uninformative. The fact that the earlier nodes will get a lot more visits than the nodes deeper in the tree will mitigate the fact that they are initialized closer to a uniform distribution.

3.6 - Implementation details of MCTS in Python

In this section, we explain in more details how we implement the MCTS algorithm using python and SCIP.

The first step is to create the tree and node classes. Those are simple classes which could be used for any tree search algorithm. A few additional functions are added to initialize scores, increment wins and number of visits.

The tree search algorithm is implemented as an infinite loop where at each iteration, we start at the root node, optimize the problem (until we get a feasible solution or infeasible LP) and add a new node to the tree, as shown in the algorithm below. The following pseudocode shows every step of the process in a condensed form (including the best of 4 dives for node creation).

Algorithm 1: MCTS for integer programming with Branch and Bound

```
This algorithm optimizes an integer linear program, using MCTS. It
 takes as input the name of the MILP instance, the number of dives
 before adding a new node to the MCTS tree and the maximum
 number of LP to solve. It outputs the best feasible solution found.

 INPUT: instanceName, nbDivesToAddNode, allowedLPSolves;
 OUTPUT: bestFeasSol;

mctsTree = MCTSTree() // Initialize MCTS tree
// Initialize variables and read instance to optimize
diveNb = 1;
diveSols = [ ];
scip.readInstance(instanceName);

// Run MCTS algorithm for allowedLPSolves iterations
while $totalLPSolves \leq allowedLPSolves$ do
    mctsTree.selectionExpansion() // Will get to the leaf node of
        the MCTS tree, and branche in SCIP at the same time to
        get to the analogous node in SCIP
    feasSol = scip.executeDive() // Simulation step of MCTS.
        Execute dive until feasible sol or infeasible LP

    // Backpropagation step of MCTS
    mctsTree.incrementVisitToRoot();
    diveSols.append(feasSol);
    if $diveNb == nbDivesToAddNode$ then
        bestNode = getBestNode(diveSols);
        mctsTree.incrementWinsToRoot(bestNode);
        diveNb = 1;
        diveSols = [ ];
    else
        diveNb += 1;
    end

    // For maximization problem
    if $MAX(diveSols) > mctsTree.bestSol$ then
        mctsTree.updateBestSol(MAX(diveSols))
    end
    scip.reset() // Reset SCIP's BnB tree and remove it's
        information from memory
end
return mctsTree.bestSol
```

In the function *selectionExpansion(),* when we traverse the tree, we also branch on the same node in the SCIP tree at the same time. This can be done by setting the random number generator seed of SCIP, which means every time we read a certain problem and solve the relaxed LP at the root, all of the candidates will be the same, in the same order. This means that each node in the MCTS tree will have an analogous node in the SCIP tree which can be identified easily using attributes from the MCTS node.

Getting SCIP's branch and bound tree to the same node as the MCTS will not require us to solve the LP, since we know which variable to fix to which bound and we can fix multiple variables without solving the LP again.

Usually, when SCIP optimizes a problem, it does not stop at a feasible node, since its goal is to prove optimality, rather than simply finding feasible solutions. To stop at a feasible solution, we need to simulate branching by tightening the variable's upper or lower bound, instead of simply branching on the chosen variable, which keeps a list of open nodes to explore. Tightening variables tells SCIP that we found a way to prove that this variable is set at its optimal value. This means that as soon as a feasible solution is found, SCIP stops the optimization since it thinks the solution is optimal, since there are no open nodes to explore.

Once a feasible solution is found and a new node has been added to the MCTS tree, we delete all of the information that SCIP keeps on the problem including the feasible solution and all of the new bounds we added to simulate branching. We then read the instance file again, start at the root node of the MCTS tree, and execute another loop of the Monte-Carlo Tree search. This is done until a pre-set number of LPs have been solved.

The implementation described above was quite complex to implement because of the intricacies of SCIP. Many details which are not described above took a lot of work to implement. For example, making sure the LP is not solved unnecessarily is a high priority. With the complexity of the SCIP package, it took a lot of trial and error to find ways to traverse the branch and bound tree in the selection step of the MCTS without solving the LP again. Simply associating each node of the MCTS tree with a node in SCIP's branch and bound tree wasn't easy, as every time we reset SCIP, it removes all statistics gathered on different nodes. This means it is not possible to keep a copy of the node object, since it will be deleted on SCIP's reset. We decided to instead keep the name of the Branch and Bound node, which SCIP sets as the name of the variable. This is possible since we set the random number generator seed, which means each branch and bound node will have the same name before and after a SCIP reset.

The other detail which took more work than expected is regarding the statistics which SCIP can calculate. In SCIP's documentation (the C package), there is a lot of information on every function implemented, however, not all of them are implemented in PySCIPOpt. The documentation is not available on how to use the SCIP functions which are not implemented in Python's PySCIPOpt. We had to either implement them ourselves or modify the PySCIPOpt package to include different functions. This is the case for most diving statistics we use in this project.

4 – Experiments

In this section, we study the empirical performance of our proposed MCTS method on different optimization problems. We compare MCTS to several different models and justify each one. We end with a discussion on the different hyperparameters used in our models and justify their values.

4.1 - Optimization problems

Because of the nature of the MCTS algorithm which needs to explore every possible action before it starts converging, our main focus is on hard problems which are not easily solved using modern methods, especially on the primal side. This is the reasoning behind using GISP, which has been shown to be hard on the primal side [25].

We also wanted to test our algorithms on more than one problem, since it is possible that our algorithm performs well on a set of problems and not on another. The second problem we chose is set cover. Even though the set cover problem is different from GISP in the sense that it is quite easy to find feasible solutions, we chose this problem to confirm the hypothesis that our algorithm works better on problems which are hard on the primal side.

More information on how the instances were generated, as well as the parameters used can be found in the appendix section (*section 8.2*).

4.1.1 - Generalized independent set problem (GISP)

The Generalized independent set problem (GISP) builds on top of the independent set problem. The goal of the independent set problem is to find the largest independent set $S(V, E)$ from a graph $G(V, E)$. An independent set is a set where for each pair of vertices in *S*, there is no edge connecting the pair of vertices.

The GISP problem builds on top of the independent set problem by allowing to remove some of the edges of the original set at a certain cost. The goal is to maximize the net benefit of the independent set, where the removable edges have a removal cost and each vertex has a benefit if kept in the independent set.

This problem can be formulated as a linear programming problem, with binary variables for nodes and edges, and constraints for each edge in the graph. In this linear formulation *V* represents the set of vertex, E the set of edges, with *i,j* in $(i, j) \in E$ representing both vertices connected by edge e. x_i takes a value of 1 if vertex *i* is chosen to be in the subset S. $z_{ij}$is also a binary variable taking a value of 1 if edges *i* and *j* are in set S. w_i represents the value gained from including vertex *i* and c_{ij} represents the cost of removing edge *i,j* from the original set. The following equations represent the linear formulation of the generalized independent set problem.

$$Maximize \sum_{i \in V} x_i w_i - \sum_{(i,j) \in E} z_{ij} c_{ij}$$

Subject to:

$$x_i + x_j - z_{ij} \leq 1 \quad \forall (i,j) \in E$$
$$x_i, z_{ij} \in \{0,1\} \quad \forall i,j \in V$$

The number of variables and constraints grows quickly as the number of vertices goes up. When generating a graph, a density parameter is chosen, which represents how many edges are connected to each vertex, on average. For example, if there are 100 vertices, with a density of 0.5, an average of 50 edges would be connected to each vertex for a total of 5000 edges. With 1 variable per edge, the number of variables associated with the edges is equal to $density * nbVertices^2$. This means that by doubling the number of vertices, the number of constraints would roughly quadruple. There is only one variable associated with each node, which means the total number of variables is equal to $density * nbVertices^2 + nbVertices$. This makes this formulation of the problem impractical to solve optimally on large instances.

There exist more efficient non-linear formulations of the GISP problem, however we will be focusing on the linear formulation, since our goal is to test our method on hard MILPs.

4.1.2 - Set cover

The set cover problem is a NP-Complete problem. The goal is to cover all of the elements of a set U, by the union of the smallest subset of sets from another set S. For example,
U = {1,2,3,4,5,6} and S = {{1,2,4}, {5}, {3,4,5,6}, {5,6}}. In this case, the optimal solution would be Opt = {1,0,1,0}, with binary variables representing the inclusion or not of the sets S in the optimal solution. This means, we would include the sets {1,2,4} and {3,4,5,6} and not include {5} and {5,6}. Note that the union of all elements of S is always U.

Set cover is a linear programming problem where each element of the set U is represented by a constraint. The following linear program represents the formulation of the set cover problem.

$$Minimize \sum_{s \in S} x_s$$

Subject to:

$$\textstyle\sum_{S: e \in S} x_s \geq 1 \quad \forall\ e \in U \ \ (every\ element\ of\ U\ is\ covered)$$
$$x_s \ \in \{0,1\} \quad \forall\ s \in S$$

As opposed to the GISP problem, it is trivial to find primal solutions to the set cover problem. We can see that for any instance of set cover, we can simply take the union of every set inside S. We could hypothesize that, relative to standard MILP solvers such as SCIP, MCTS should perform better on GISP than on set cover instances. This is because MCTS may take many iterations before starting to converge to good primal solutions.

4.2 - Performance measure

To compare the performance of all of our models, we will be using the primal integral (more information in the appendix on how it is calculated), which measures how quickly the primal bound improves. We will be using this measure as opposed to solving time, since solving time is hardware and context (e.g. what else is the machine running) dependent and thus it is harder to get consistent estimates of solving time. The other reasoning for using the primal integral is that we are comparing our algorithm with SCIP, which is a highly optimized library, written in C. With the time allocated to this project, and my limited knowledge of the C programming language, I chose to use Python, which means my code will run slower than its C equivalent. Using the primal integral as our basis for comparison, means we will be evaluating the algorithm instead of the programming language used.

The final reason for not using time is that our goal is slightly different from SCIP. We are only focusing on tightening the primal bound (we don't measure the dual bound), while SCIP also focusses on reducing the primal-dual gap. This being said, every time SCIP solves the dual LP to prove optimality after having found the optimal value of the primal bound, the primal integral will not change. For example, if SCIP finds a primal bound with a value of 500, which is optimal (SCIP has not yet proven its optimality), and works on closing the primal-dual gap to prove optimality, every time SCIP solves the LP to prove optimality, the primal integral's value doesn't increase. This is because the primal integral calculates the area under the primal curve and over the optimal value, meaning once the optimal value has been found, all of the subsequent LP solves won't increase the primal integral. Using SCIP's aggressive mode, the optimal solution will probably be found before SCIP's non-aggressive counterpart, since it focuses on the primal bound more than the dual bound. This means that SCIP aggressive should have lower values of primal integral for a given problem, on average.

Since we are building a new primal heuristic, we could have compared our method against other primal heuristics implemented in SCIP. We did not find an efficient way to force SCIP to use certain heuristics and not others, which made it hard to compare our heuristic to existing ones. For this reason, as well as others listed above, we think the primal integral is a good measure to compare our heuristic with SCIP.

4.3 - Different tree exploration models tested

As stated above, we are testing MCTS with different rollout policies, as well as different methods for initialization of node statistics. Using all of the rollout methods, as well as warm starting, we tested 7 different combinations of models, along with two different parameters for the SCIP models. Here is the list of all of the models along with a quick explanation.

1- Random policy: This method does not use MCTS. It branches randomly from the root node until a feasible solution is found or a LP is infeasible. Being the simplest possible model, this is our baseline.
2- MCTS vanilla: This is the simplest of the MCTS variants, with random rollouts and no warm start initialization.

3- MCTS fractional diving: This is a MCTS model with fractional diving and no warm start. This model was added to test the impact of very simple rollout policy which is presumably better than random rollouts.
4- MCTS average diving: This MCTS model uses average diving explained in *section 3.4*. This model still doesn't have any warm start.
5- MCTS random diving warm start (prior: 1 visit): This is similar to MCTS vanilla, except we also initialize the nodes, using average diving statistics, with a prior of 1 visit per node. This will help us understand how important node initialization is.
6- MCTS average diving warm start (prior: 1 visit): MCTS with average diving used both for node selection and warm start initialization.
7- MCTS average diving warm start (prior: 5 visit): Same as above, with a stronger prior.
8- SCIP standard: This is the internal SCIP model without any presolving.
9- SCIP aggressive: Same as above except heuristics are set to aggressive, meaning it will execute heuristics more often.

SCIP with only selected heuristics
One of the models we wanted to test is SCIP which is only allowed to use the same branching rules used in our average diving rollout policy, meaning only the diving heuristics we implemented. The goal of this experiment was to see if we could match SCIP's node selection using MCTS. 1

Since there are hundreds, if not thousands of parameters in SCIP, it is impractical to simply disable the methods we don't want it to use. Instead, we enabled only the dives we want it to use, as well as all of the methods which were necessary, like solving the LP as well as the dual problem.

When testing SCIP with those parameters, we realized that SCIP was practically never executing diving heuristics; instead, it was only solving the dual. Because of this, we did not use those parameters of SCIP in our experiences. This would need to be explored in further detail.

4.4 - Hyperparameters of the model and settings of the experiment

Exploration ratio: As stated earlier, we will be using $\sqrt{2}$ for the exploration ratio in the UCT formula. This ratio dictates how much exploration we do when choosing nodes in the MCTS algorithm. We did not perform any extensive testing with different exploration ratios, however, anecdotally, a ratio 5 times smaller than $\sqrt{2}$ was tested, which practically removed exploration on the instances we tested on.

Good values for this ratio are problem-dependent, and extensive testing would be required to find a good value. For example, if we are using MCTS for a game like go, a higher value for exploration ratio could be beneficial. This is because when playing against an expert player, very few of the possible moves will lead to a win. This means that a move which is theoretically good but will almost never lead to a win could be exploited too early.

In our case, since many set of decisions can lead to an optimal solution, a lower value could drastically improve the speed of finding good solutions, however, it could also converge to worse parts of the search space if our rollout policy is poor.

Number of simulations before adding node: This represents how many MCTS simulations are performed (up until a feasible solution) before adding a new node in the MCTS tree. This node is added to the MCTS tree by selecting the node with the best feasible solution found. We chose to perform 4 dives before adding a new node. This is the only value that was tested. Performing more simulations before assigning a win would reduce the possibility of slightly worse moves being explored further, since the node associated with the particular move wouldn't be added as often. This is not necessarily bad in itself and would need to be explored further.

Total number of times the linear problem is solved
This represents how long we should leave our algorithms running for each of the different models chosen. We chose the value of 100 000, which is about 1 hour on the machine used to run the experiments. We wanted to run enough simulations starting at the root node to see if the root candidates started converging. The other motivation for choosing this one hour is that the instances we were running our algorithms on took from 1 to 2 hours for SCIP to solve[4] (with every parameter enabled, like presolving, etc.) on a standard machine[5] and we wanted to see how every algorithm performed on instances it shouldn't have enough time to solve to optimality in a given timeframe.

LP solves per branch: We chose to solve the LP after every branching decision. One of the reasons is explained in the next section in more detail, whereas increasing this value for random rollout policy yields unintended results. We tested on methods with non-random rollout policies with higher values of this parameter, like 5 or 10. This was pretty inconsistent, working pretty well (less LP solves on average to find optimal solutions) on some instances while leading to infeasible solutions often on other instances. It is to be noted that we did not test this value extensively and the results stated were simply observed on a limited number of samples.

Note on using more than 1 LP solves per branching for random policy
The idea behind branching more than once before solving the relaxed linear problem is that there will be few changes to the LP after branching 5 times for example. This means that if executing a fractional dive and branching on the 5 most fractional variables before solving the LP again, it is likely that the LP will still be feasible, since those variables were close to whole number values and fixing them to integer values will have little impact on the constraints.

This logic does not transfer very well to random diving. This is because the 5 variables to branch on will be chosen randomly. From a certain LP, each branching decision made without resolving the LP increases the odds of landing on an infeasible LP. This is augmented by the fact that the branching decisions are not based on any statistic or knowledge of the problem at hand.

[4] This was done to get the optimal value of the instance at hand to calculate the primal integral.
[5] 2.7 GHz Dual-Core Intel Core i5

This is not the case for other diving heuristics. For example, if we have a problem with 10 variables and after solving the relaxed linear problem at the root node, there are 5 fractional variables, each with values very close to their closest integer value. Fixing all of them to their closest integer has a higher probability of finding a feasible LP compared with branching randomly either up or down.

When testing random rollout policies, with 5 or 10 branching decisions per LP solves, on GISP instances, infeasible solutions were encountered for most of the rollouts that were performed. This makes it so either wins are rarely assigned, or wins are assigned randomly (none of them are better than one or another), which are both suboptimal ways to build the tree.

5 - Results

Since our goal is to explore different variations of MCTS with relatively limited computation power, we are not expecting to beat state-of-the-art solvers. Instead, our aim is to understand, through this preliminary study, if MCTS has potential and should be explored further for solving MILPs.

5.1 - Set cover instances

In the following sections, we go through results for the set cover instances, by looking at the LP integral results and the distribution of solutions for a single instance of set cover.

5.1.1 - Aggregated primal integral results

The following table represents the integral data for each of the instances on which we ran the different algorithms. The integral values have been divided by 10^5 for easier reading, with bolded values representing the best method for each of the instances

Instance	Rand_ policy	MC_std	MC_ fractdive	MC_ avdive	MC_randdive_ warm1	MC_avdive_ warm1	MC_avdive_ warm5	SCIP	SCIP_AGG
1	8.98	9.86	18.86	1.29	7.54	1.05	1.03	0.05	**0.03**
2	12.01	11.05	21.43	1.58	11.28	1.83	1.83	0.05	**0.04**
3	8.80	11.36	21.67	1.11	10.64	1.06	1.10	**0.06**	0.06
4	13.11	14.54	25.76	1.14	11.97	1.34	1.41	0.02	**0.02**
5	11.72	11.19	22.27	1.72	10.94	2.15	2.20	0.19	**0.17**
6	11.16	11.50	20.65	0.36	7.75	0.53	0.57	0.02	**0.02**
7	7.50	7.61	19.79	0.89	7.67	1.22	1.23	0.03	**0.02**
8	11.95	10.45	27.32	0.78	15.52	0.80	0.82	**0.06**	0.10
9	10.29	9.68	18.62	0.43	8.71	0.31	0.29	0.02	**0.02**
Wins	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**2**	**7**

Table 1: Primal integrals for set cover instances

The following figure represents the same data in boxplot form. This shows a better overview of how each method performed. The methods are in the same order as the table above.

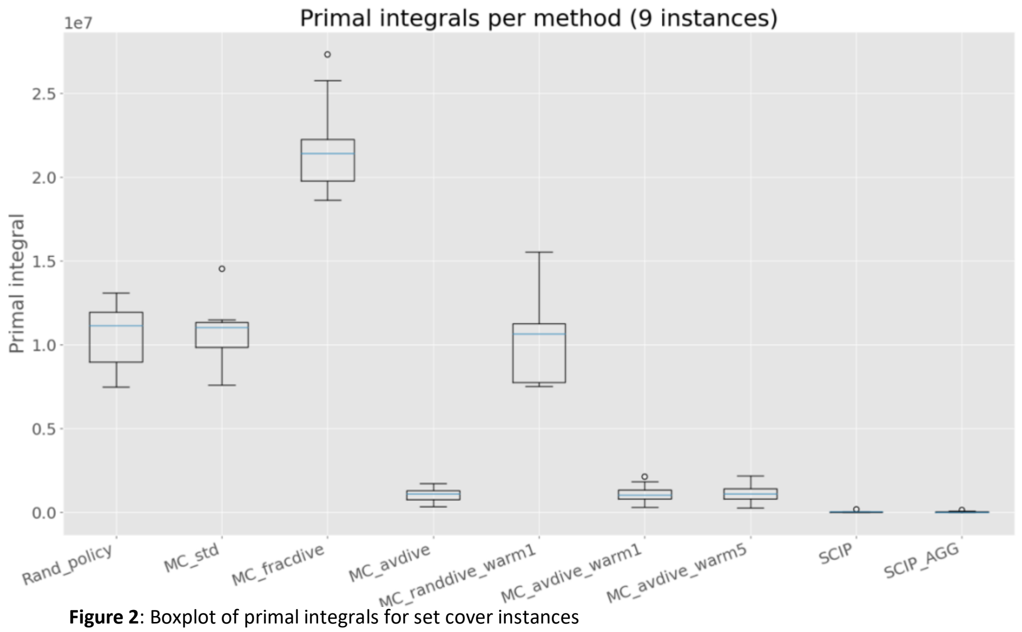

Figure 2: Boxplot of primal integrals for set cover instances

We notice that the SCIP-based methods significantly outperform other methods on the test instances. Those results are in line with what we expected for the set cover instances. As we stated previously, set cover instances are quite easy to solve on the primal side. We can also see that all of the MCTS models with average diving performed significantly better than the MCTS models without random diving.

Furthermore, both models with random dives perform about as well as the model that branches randomly. Looking more closely at the results from MCTS vanilla, and the random policy, which have the same rollout policy, we could think that the MCTS algorithm in itself does not outperform simply using diving heuristics until feasible solutions. This is not necessarily true, since the rollout policy itself could influence how impactful the MCTS algorithm is. In the next sections we discuss in more detail the convergence of the MCTS algorithm and how the solutions found change over time.

5.1.2 - Single problem analysis

The next figure represents all of the feasible solutions found for a single instance of set cover, with the red line representing the optimal solution. The table to the right represents the number of feasible solutions found as well as the frequency at which a feasible solution is found, which is simply $\frac{100\,000}{feasible\_solutions}$.

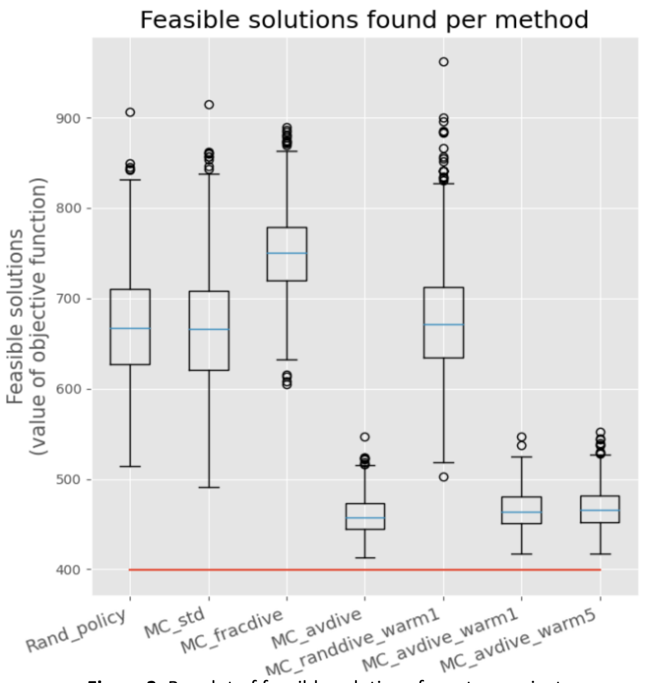

Figure 3: Boxplot of feasible solutions for set cover instance

Method name	Feasible solutions found (100 000 LP)	LP solver per feasible solution (average)
Rand_policy	1062	94
MC_std	1068	93
MC_fracdive	1766	56
MC_avdive	1309	76
MC_randdive_ warm1	1055	94
MC_avdive_ warm1	1295	77
MC_avdive_ warm5	1294	77

Table 2: Number of feasible solutions found for set cover instance

The first thing to note when looking at this graph is that the SCIP models are not included. This is because SCIP models do not keep the intermediate feasible solutions. SCIP only keeps the best solution and they often do not even get to feasible solutions because of pruning. As stated previously in *section 3.3*, we implemented a pruning feature for the GISP instances, in a slightly different manner.

The other notable information we can gather from the graph to the left is that the 3 MCTS models using average diving find solutions which are better than even the best solutions found from the other methods. This is a strong indication of the importance of the rollout policy, since the worst rollout policies do not come close to the optimal solution.

When looking at *Table 2*, we can see that the model with fractional diving finds close to twice as many feasible solutions compared to random-diving policies (in the same number of solved LPs). If the solutions found from each method all came from similar distributions, finding more of them in a certain time period would obviously be beneficial. This is however not the case as we saw earlier. This is especially true with the fractional diving method, which has a significantly worse median solutions found for this specific instance. On the other hand, methods with average dives have a higher average value of feasible solutions and find feasible solutions quicker on average, when comparing to the methods with random dives.

5.2 - GISP instances

In the following sections we will look at the GISP instance results from the same perspectives as in the previous sections on the set cover instances.

Primal integral results

The following table represents all of the LP integrals for 6 GISP instances (best value in bold) and the following graphic represents the same information as a boxplot. Values have also been divided by 10^5.

Instance	Rand_ policy	MC_std	MC_ fractdive	MC_ avdive	MC_randdive_ warm1	MC_avdive_ warm1	MC_avdive_ warm5	SCIP	SCIP_AGG
1	17.10	9.86	71.32	54.82	15.26	56.18	50.43	8.35	**8.15**
2	11.16	5.88	81.92	55.94	17.13	62.50	59.80	**2.27**	2.41
3	16.72	7.70	68.13	52.42	9.88	65.45	62.17	0.63	**0.60**
4	12.69	7.36	47.64	44.66	11.45	42.20	42.77	0.46	**0.46**
5	11.65	10.24	47.70	48.02	3.05	49.34	43.72	**1.00**	2.03
6	19.19	10.60	59.38	38.96	18.34	41.26	42.98	3.06	**2.75**
7	8.42	5.55	63.52	40.36	7.96	50.82	51.52	1.00	**0.91**
8	17.75	13.38	55.40	58.80	12.80	60.58	60.58	**3.96**	4.00
9	13.45	15.01	56.48	54.28	16.54	49.62	52.29	0.87	**0.54**
10	6.30	13.87	59.43	46.46	15.45	61.89	63.56	1.95	**1.90**
Wins	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**3**	**7**

Table 3: Primal integrals for GISP instance

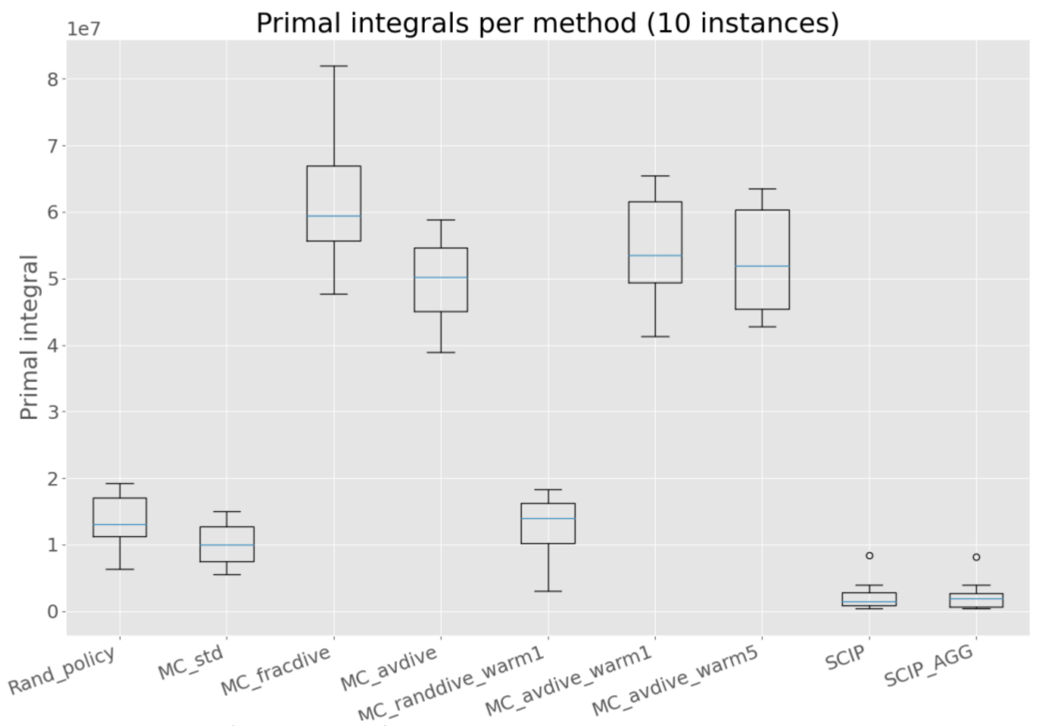

Figure 4: Boxplot of primal integrals for GISP instance

When looking at the boxplot, we can see that SCIP outperforms all of our MCTS models on every instance. We can also see that for the harder instances (those with higher LP integral for SCIP models), the gap between the SCIP models and the MCTS models is quite small, as compared to set cover instances. Also note that comparing the results of our MCTS models and those from SCIP is somewhat unfair since SCIP is also working on the dual side to prove optimality. We are simply noting the fact that on set cover instances, our best MCTS models had LP integral values between 10 and 20 times worse than SCIP. For GISP instances, our best MCTS models had a LP integral value from 1.25 to 3 times worse than SCIP. Furthermore, the results for MCTS with random diving are quite close to those of the SCIP models. Those two facts could be a confirmation of how hard GISP instances are on the primal side for solvers like SCIP.

We can also see that random diving methods performed significantly better than other MCTS methods. This is the opposite of the results found for the set cover instances, where random diving methods performed significantly worse than average diving methods. This could mean that the non-random diving methods we used are not adapted to GISP instances. Since we averaged 4 diving heuristics, it is possible that those heuristics are not well suited to the problem at hand.

When solving the relaxed LP, either at the root node, or at subsequent nodes, we found that all of the fractional variables had values of 0.5. This means that only two of the four diving statistics are providing information (fractional diving and line search diving are uniform if both the fractional variables at the root node and the fractional variables at the current node are 0.5). This puts more weight on the pseudocost and coefficient diving, any of which could not be suitable for GISP instances. This is one of the flaws of our approach of averaging the statistics, where some of the statistics may not be suited to the problem at hand. Using a simple average, there is no mechanism to vary the weight of each statistic based on their value, like there is for neural network based policies.

The other notable information we can gather from the above results is that the MCTS vanilla outperformed every other one of our models on 7 out of 10 instances, with the random policy being superior on 1 instance and MCTS with random diving and a prior of 1 being superior on 2 out of 10 instances. The fact that MCTS vanilla outperformed the random policy is an indication of the effectiveness of the MCTS algorithm, since the only difference between both policies is the branching decision before the rollout step. It is unlikely to be due to luck, since it outperformed on 9 out of the 10 instances we performed the test on.

The fact that MCTS vanilla outperformed the other MCTS policies is a confirmation of the importance of the rollout policy. In the case of the GISP instances, we can see that the diving heuristics we use are probably not adapted to the problem at hand, which is likely why their performance is poor. We can say this because they performed a lot better than the random diving methods, when testing on the set cover instances.

5.3 - MCTS convergence

In this section, we will try to understand how the MCTS converges after a large number of rollouts. We first look at all of the feasible solutions to see if they improve over time and analyze the win percentages at the root node to see if they seem to converge. We compare those statistics for the MCTS vanilla as well as the MCTS with average diving.

Due to time constraints, this analysis was done on an instance similar to GISP with less binary constraints. This allows us to be able to visualize the root node statistics, since with regular GISP instances, it is impractical to analyze the graphic with thousands of candidates.

In the following sections, we see that we have indications of convergence, for both the MCTS vanilla, as well as the MCTS with average diving.

5.3.1 - Random diving

In this section we look at the convergence of MCTS with random diving, the number of LP solved is large.

Moving average of objective function value

The first experience we performed is to run MCTS with random rollouts for a longer time period and see how the average solutions change over time. We ran the MCTS vanilla algorithm for 500 000 LP solves (5x more than the other tests) and recorded all of the feasible solutions. The graph on the right represents the moving average of all of those solutions. We are using the moving average to see the trend of all of the solutions found. For reference, the optimal objective function value for the instance used to create the graph below is 1553.

We can see a trend in the positive direction. It is hard to say if this trend is due to luck or the convergence of the MCTS statistics (especially at the early nodes). When looking at the range of y values, we see that it is quite compact, however, the moving average is over the last 2,000 values, meaning a small difference is quite significant. An improvement in the average solution found means that over time, we are exploring parts of the search space which have a higher likelihood of finding good solutions.

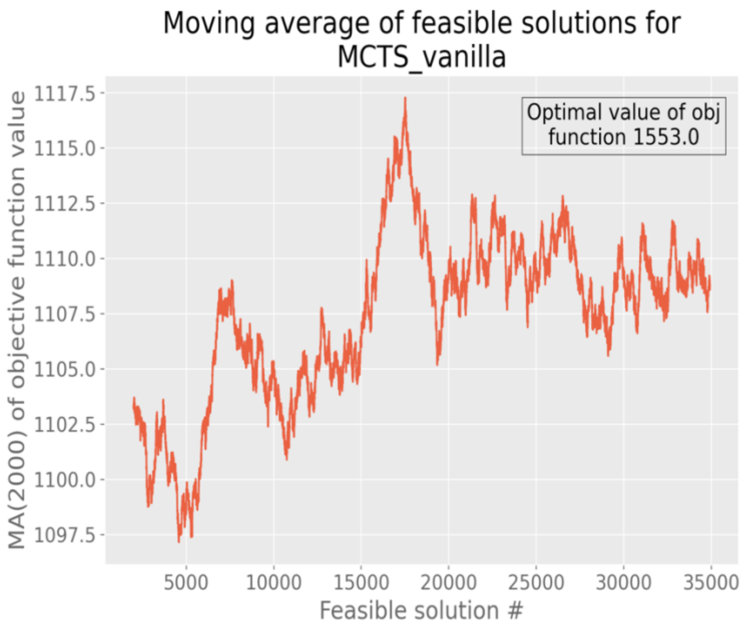

Figure 5: Moving average of feasible solutions MCTS vanilla

Win percentages at root node

We can also look at the win percentages at the end of this experience to see if the MCTS tree converges at the root node.

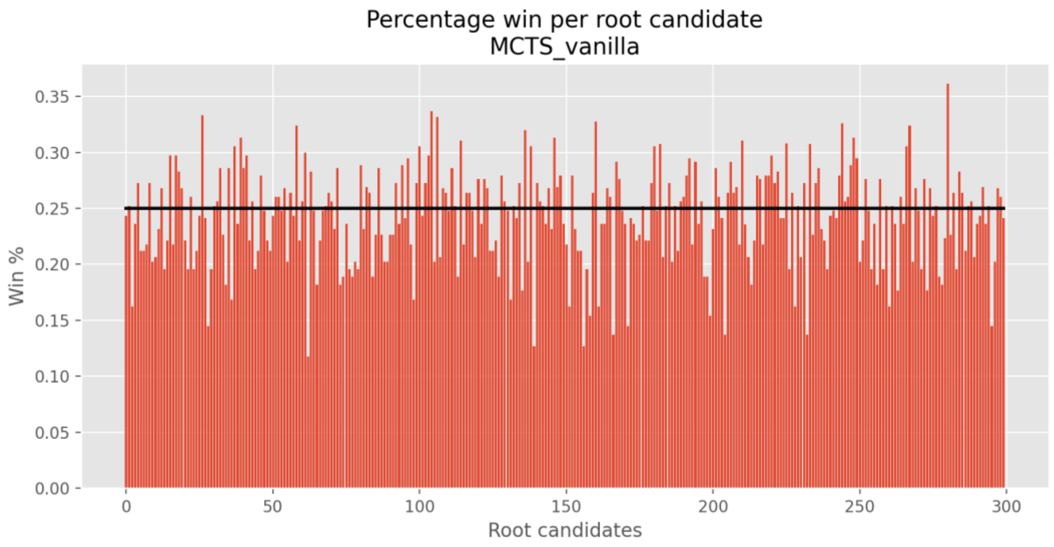

Figure 6: Win percentages at root nodes for MCTS vanilla compared to average win percentage

In this graph, every bar represents the win percentage when choosing one of the candidates at the root node of the MCTS tree. As $N \rightarrow \infty$, the UCT statistics will move up and down naturally. This means even the worst nodes will continue to get visited; however, the win percentage is not influenced at all by the exploring part of the UCT statistic. This is why we look at the win percentages and number of visits to understand convergence.

In *figure 6*, the black line represents the average of all win percentages. Since we are assigning a win every 4 dive we perform and each of those dive leads to a feasible solution, the average value is 0.25. We see that the range of win percentages is quite small, meaning that only few of the nodes have win percentages significantly above the average.

Number of visits at the root node

We also look at the number of visits for each node to see if the best nodes have significantly more visits, as well as the range of number of visits between the best and worse nodes.

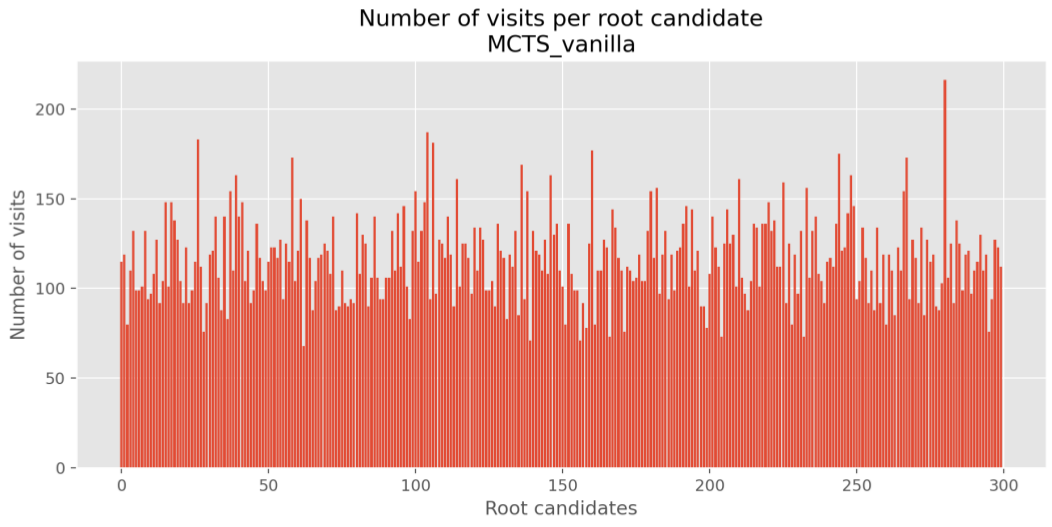

Figure 7: Number of visits at root nodes for MCTS vanilla

In *figure 7*, we see that the best candidates are chosen two to three times more often than the worse candidates. This is because of the UCT statistic which favors the visit of worse nodes periodically. When looking at the data from both of the previous graphics, we see that nodes with high win percentages also have high number of visits, which makes sense, since they are the nodes benefiting from more visits.

5.3.2 - Average diving

In this section, we look at the convergence of average diving methods, through the same lens as above for random diving methods.

Moving average of objective function value

We will now look at the results of feasible solutions and root statistics for the same instance as above (optimal objective function value is 1553), using the MCTS with average diving, without warm starts.

First, let's look at the moving average of the solutions found over the 500 000 LP solves.

There seems to be an upward trend in the first half of the graph, however, this trend does not continue for the second half of the feasible solutions. Using this information alone, it is hard to say if solutions are truly getting better over time. The other notable information from this graphic is the range of moving averages, which is about 100 higher. This emphasizes the fact that the diving heuristic or rollout policy is important in finding good solutions.

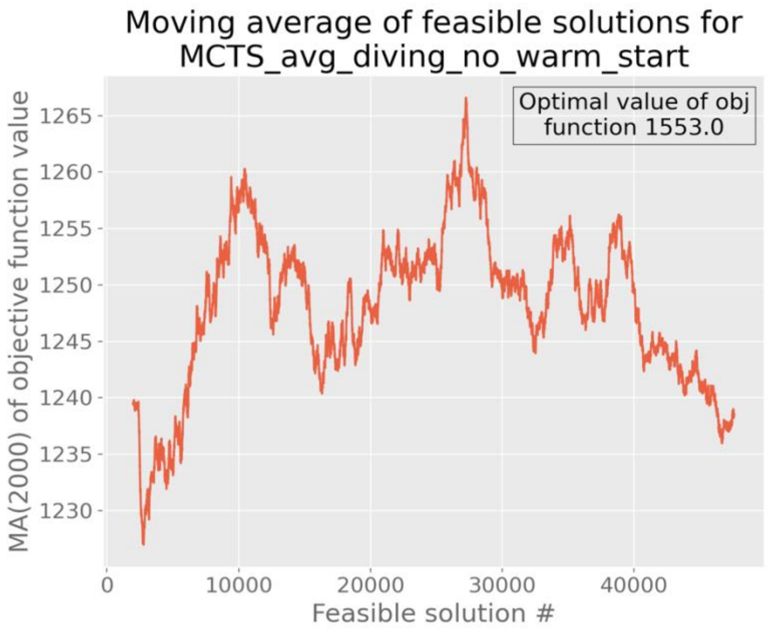

Figure 8: Moving average of feasible solutions MCTS average diving

Win percentages at root node

The following graph represents the win percentages for all the nodes, for the same MCTS model as above, also for 500 000 LP solves.

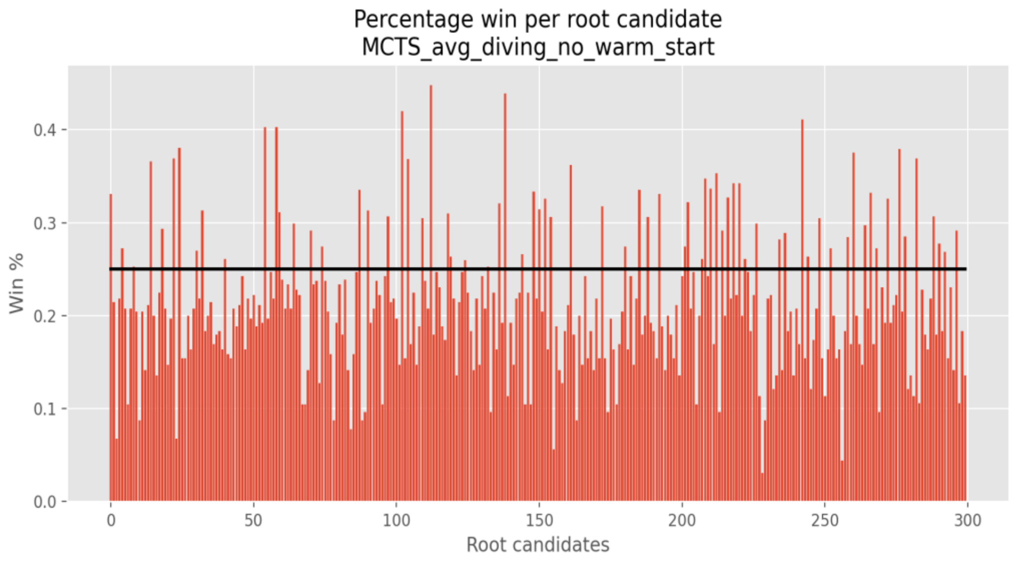

Figure 9: Win percentages at root node MCTS average diving compared to average win percentage

If we compare this graph with *figure 6*, which uses random diving, we see that there are more nodes which are further from the average win percentage. We see that there are many nodes with over 40% win percentage and a similar amount with less than 10% win percentage. This seems to be pointing in the direction of the MCTS converging to the best solutions. As stated previously, the UCT statistic does not converge to a single node, at least in a reasonable number of simulations, since the exploration part of the statistic will always increase over time, up to the point that it is the chosen candidate with the highest UCT.

The fact that the range of win percentages is larger than that of the MCTS vanilla algorithm is significant because the higher the win percentage, the higher the probability of the node being chosen. This is because the exploitation part of the UCT statistic is not affected by the total number of visits at the parent node. As long as a node is not chosen, the exploration part of its UCT statistic will naturally go up, until the point where it is chosen. Having a smaller win percentage means it will take more visits to other nodes before it is chosen because its starting point (win percentage) is smaller. The opposite is also true for the best nodes where it will take them less visits before their UCT increases to the maximum value of all candidates, since their win percentage is higher.

Number of wins at root node

If we look at the following graph, which represents the number of visits of all of the candidates at the root node, we can see that the intuition noted above is confirmed.

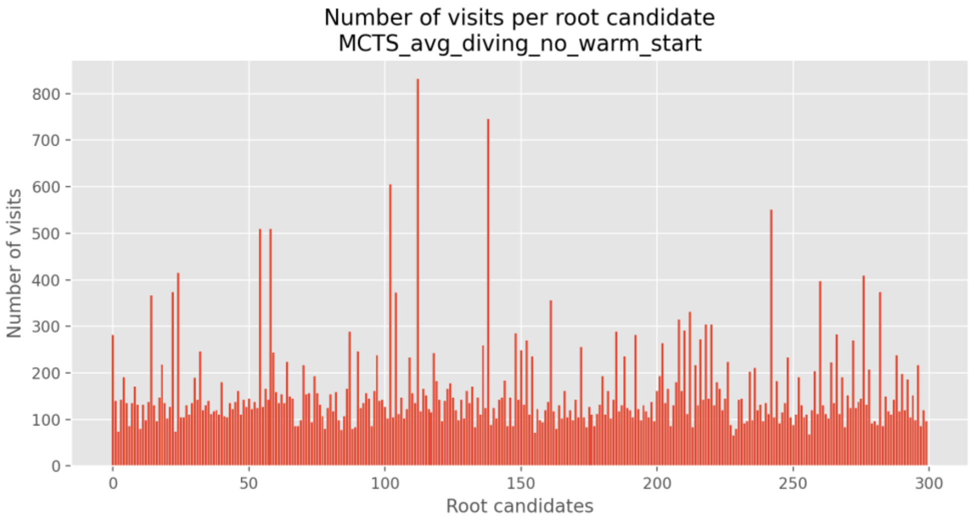

Figure 10: Number of visits at root nodes for MCTS average diving

In this case, the best nodes were visited over 10 times more than the worse candidates. This is an indication to the fact that there is more information to be gained in the average diving, when comparing to the random diving, for the specific instance we tested our algorithm on. The higher this ratio, the more confident we are in our decisions. For example, if each node is visited an equal number of times, it means that the algorithm failed to learn anything. On the opposite end of the spectrum, if some of the nodes are visited very often, while others are rarely visited, this is an indication of confidence in the learned statistics. This is because of the fact that over time, the win percentage is based on a larger sample size, which means that it is less likely to be due to randomness. It is to be noted that the ratio number of visits of the best versus the worse nodes is heavily influenced by the exploration ratio.

We can also see that even the worse candidates are visited a lot, which is not necessarily what we want, since they are very unlikely to lead us to the optimal solution. In the *model improvement* section (*section 6*), we will talk in more detail about this fact and possible ways to mitigate this problem.

6 - Possible model improvements

Using the results from the previous section, as well as the ideas from Alpha Go's implementation of MCTS, we will propose other avenues of research to improve our algorithms.

6.1 - Win assignment

One of the differences between our MCTS algorithms and Alpha Go's algorithm is due to the structure of the problem at hand. Rewards for the game of Go are binary (win/loss), while they are continuous for MILP (objective function's value). To mitigate this problem, we ran 4 rollins and rollouts before assigning a win to the node with the highest objective function value. This means we transformed the 1 player game of finding the best solution to the MILP to a 4 player game, where the goal is to find a solution which is better than the other 3 solutions. This may not be the best way to assign a win, since using this method doesn't exactly reward finding close to optimal solutions.

To improve on this, we could build two MCTS trees to play against each other, assigning a win to the node which provides the best feasible solution. This could improve on our method, since is likely that the agent which always finds the optimal solution is the agent which maximizes its win percentage.

We could also test different node statistics reflecting the continuous reward function of MILP. Some examples include Continuous Upper Confidence Trees [26].

6.2 - Node pruning

As we saw previously, it is likely that as the MCTS converges at the root node, solutions will improve on average, meaning that choosing the best move at the root node is valuable. This being said, for problems with binary variable constraints, any feasible solution will have a combination of 0s and 1s for all of those binary variables (every variable is either 0 or 1). This means that every other root node candidate of our MCTS model will lead to the optimal solution.

Using this information, we think pruning some of the nodes could be useful when the difference between the scores of the candidate which branches up and the corresponding candidate branching down is large. This is because once we are confident a variable should be fixed to a certain value; we could remove the candidate which branches in the opposite direction. Over time, we would progressively remove candidates which are unlikely to fix a variable on the optimal value for this variable.

This process would not completely remove the possibility for a certain variable to be fixed to either of its bounds. For example, if we have 4 candidates (2 variables): $\{x1_{up}, x1_{dn}, x2_{up}, x2_{dn}\}$, and we find, after many simulations that we should prune $x1_{up}$, which means that we think $x1 = 0$. This does not completely exclude the option that the optimal solution could include $x1 = 1$, since branching on $x2$ could still lead to a solution with $x1 = 1$. Pruning this variable will instead lead the search likely candidates more often than it already does. The UCT statistic already leads the search to the best possible actions, however, even the worse candidates are still chosen quite often, due to the exploration part of UCT.

We could also reduce the exploration ratio which would reduce the number of times the worse candidates are visited; however, it is hard to choose a ratio that achieves this without completely removing exploration between good candidates.

6.3 - General purpose branching policy

The other large improvement to this model would be a rollout policy which would be learned prior to the running the MCTS algorithm. We could use a neural network which uses diving statistics and any other statistic available to predict the probability of the optimal solution being in each of the candidates. This could be trained by creating a dataset of branching decisions, where we know in which candidate the optimal solution lies.

Other than using a simple feedforward neural network, we could also test a more complex architecture like Graph Convolutional network which uses the structure of the problem (constraints + variables) to make branching decisions.

Training a neural network based branching policy is time consuming prior to solving the instances, however, it would barely be more computationally intensive at test time when choosing branching candidates and could significantly improve the performance of the overall search algorithm.

6.4 - Directed acyclic graph (DAG)

As stated previously, when solving MILP with MCTS, many branching sequences can lead to a given state. For example, branching up on x_1 then up on x_2 yields the same result as branching on the same variables in the opposite order. In our implementation of the MCTS, a single state can be represented by multiple different nodes. In the above example, two different nodes would represent the state resulting from branching on $x_1$and x_2. Ideally, we would like for only one node to represent the aforementioned state.

To solve this problem, we could implement a directed acyclic graph (DAG), instead of the tree structure which we implemented for the MCTS. The difference between the tree we implemented and a DAG is that in a tree, each node only has one parent, which is not the case for DAGs. More precisely, a tree is a special case of a DAG.

Implementing a DAG would mean that a state of the LP could have multiple parents. Using this logic, meaning multiple paths could lead to a single state. Using a DAG instead of a tree structure, each unique state would be represented by only one node. This greatly reduces the size of the tree and likely improve the speed at which node UCT statistics converge.

Implementing a DAG to solve MILP would not be trivial, because of how nodes are encoded in SCIP. One workaround would be to build a tree as we did, but increment visits and wins for every node which represent the state we want to increment. In the above example with $x_1$and x_2, both nodes resulting from the branching on $x_1$and x_2 would be incremented instead of only the node resulting from branching on $x_1$followed by branching on x_2. This would mitigate the problem of multiple nodes representing a single state, since all of the nodes would share the same visit and win statistics.

7 - Conclusion

In this project, we worked on solving mixed integer linear programs using Monte Carlo Tree search. We used Branch and Bound, a popular framework for solving MILP problems and especially focussed on branching techniques. We tested our algorithms on two problems; set cover and generalized independent set problem in order to evaluate our algorithm performs on different types of problems.

We observed that on the set cover instances, MCTS models with the average diving method for the rollout step of the algorithm performed significantly better than other MCTS models, as well as the random policy. As expected, all MCTS models significantly underperformed SCIP models.

GISP instances gave different results. For GISP, we found that the best MCTS model was the one using a random rollout policy. This is likely an indication that the other diving policies we tested are not suited to the GISP instances. We also observe that on some instances, the MCTS vanilla algorithm underperformed SCIP models only slightly, which indicates that even for a strong solver, GISP instances can be hard on the primal side.

We then tested the convergence of two of our MCTS algorithms on a simplified version of the GISP problem to show that the nodes which were visited more often had a significantly higher win percentage. We also showed that the larger the gap in win percentages between the best and worse nodes, the more often the best nodes were visited compared to the worse nodes. This is an indication of the importance of a rollout policy which is informative, since we will be able to separate the best and worse nodes more clearly in this case.

We finally discussed the fact that several model improvements could be implemented in the future to test MCTS methods more thoroughly. We talked about ways to prune the worst nodes, when we have information that certain variables should not be fixed to either their floor or ceiling values. We then explained that the win assignment technique we used may not be best suited to the problem at hand and that better techniques could be used to calculate the node statistics which would be more appropriate to solving problems with continuous rewards, like mixed integer linear problems.

The other model improvement we proposed was a neural network based diving policy which could significantly improve the results of our algorithms by guiding the search to nodes which are more likely to find good solutions. As we saw earlier, choosing a good rollout policy is very important for the MCTS algorithm. Using existing diving heuristics as a rollout policy is limited, since there are a limited number of them in the literature and no heuristic is perfectly suited to all problems. We think that a neural network based rollout policy, which could be trained for a specific problem could significantly improve on the performance of general purpose rollout policies, like diving heuristics.

Finally, we suggested that computational gains might be achievable by exploiting the directed acyclic graph structure that emerges in solving MILPs in lieu the usual tree structure used for the

MCTS. The directed acyclic graph could greatly increase the speed at which node statistics converge, since each state of the LP would be associated with a single node in the DAG, as opposed to multiple nodes being associated with a state of the LP for the tree structure we use.

To conclude, even though the results found were not competitive with results from modern solvers, we think there is potential for this method to have some niche uses as a primal heuristic for problems with hard-to-find feasible solutions, as stated in *section 6.4*. Those problems are currently hard to solve with modern solvers and we are confident that combining new machine learning approaches with traditional methods could improve on the solver's results.

8 - Appendix

8.1 - Details on primal integral

When executing the algorithm, every time we solve the LP problem, we keep the primal bound along with the total number of LP solves so far. Using this set of points, we can calculate the integral (area under the curve) of the primal bound over the number of LP.

Since we are creating boxplots with the values of the primal integral of different instances, we need the values from different instances to be of similar magnitude. For example, if we minimize a problem and have an instance with an optimal value of 1000 and another with an optimal value of 200, we expect the LP integral of a method to be higher for the problem with the higher optimal value. To mitigate this problem, we only calculate the primal integral over the optimal value, meaning we are calculating the area under the curve of best objective values and over the line representing the optimal solution.

In this project, we will work with both minimization and maximization problems. Here is an example of all the curves used to calculate the primal integral for a minimization problem, with the optimal solution being the black line.

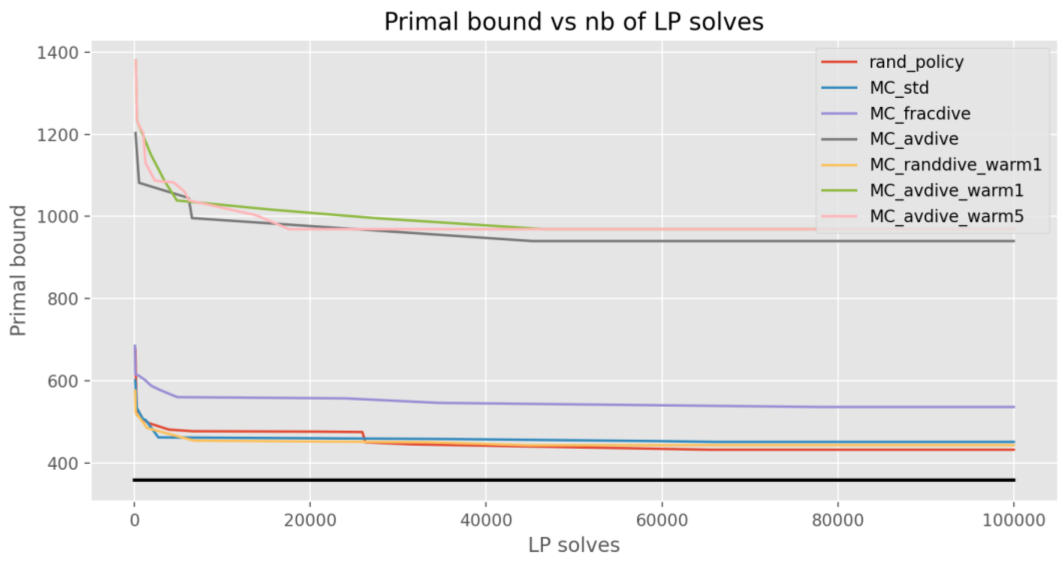

Figure 11: Primal integrals for minimization problem

In the following image, we can see the same curves for a maximization problem.

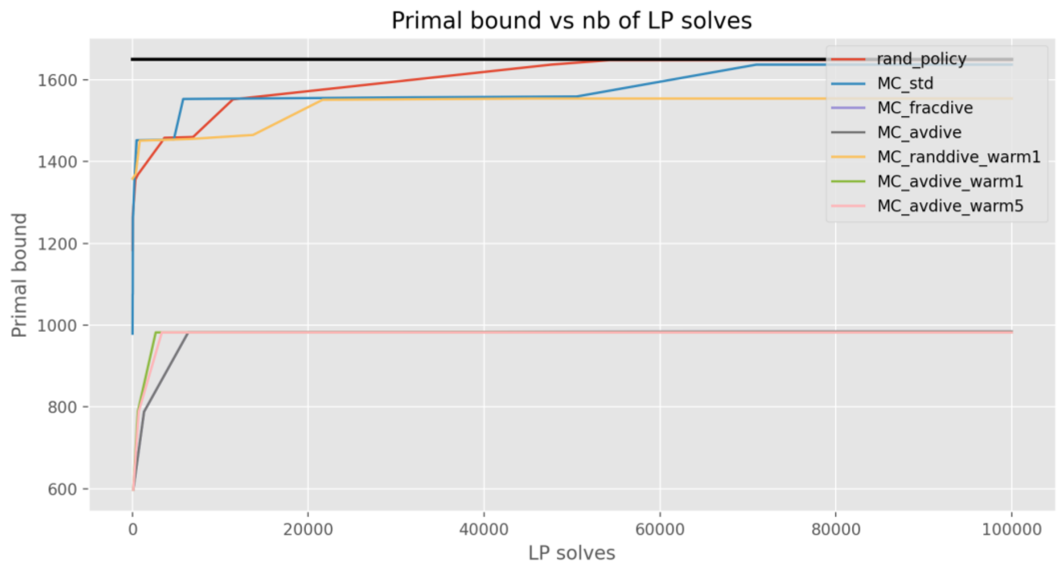

Figure 12: Primal integrals for maximization problem

For maximization problems, the primal integral is the area under the optimal value line and over the primal bound curve.

8.2 - Details on instances

Set cover: The generator for the set cover instances was taken from the code base of learn2branch [27]. Unless stated otherwise, instances of set cover were generated with 2000 rows and 1000 columns, with a density of 0.75.

GISP: The generator for the GISP instances was taken from the learn2selectnode's code base [28]. Instances were generated with between 125 and 150 nodes and an edge probability of 0.75. Those parameters were shown to lead to hard instances to solve on the primal side [25].

8.3 - Programs and libraries used

We used python 3.7 to program the algorithm, along with standard libraries to facilitate matrix operations.
We also used the open-source optimization package SCIP (Solving Constraint Integer Programs) as well as the python wrapper PySCIPOpt. Those packages were used to solve the LP problems, get the fractional candidates at a given node, find evaluate if a problem violates any constraint, and more.

8.4 - SCIP aggressive vs non-aggressive

By looking at *table 1* and *table 3*, we can see that SCIP aggressive outperforms SCIP's non-aggressive counterpart 14 out of 19 times, when measuring the LP integral. This confirms our intuition that SCIP aggressive tightens the primal side more aggressively than the non-aggressive version.

Bibliography

[1] G. L. Nemhauser, Integer Programming: the global impact, Georgia Tech, 2013.

[2] A. Schrijver, "On the history of combinatorial optimization (till 1960)," [Online]. Available: https://homepages.cwi.nl/~lex/files/histco.pdf.

[3] G. E. Moore, "Cramming more components onto integrated circuits," vol. 38, no. 8, 1965.

[4] P. Nunne, "University of Liverpool," [Online]. Available: https://cgi.csc.liv.ac.uk/~ped/teachadmin/COMP202/annotated_np.html.

[5] R. E. Bixby, "A Brief History of Linear and Mixed-Integer Programming Computation," *Documenta Math.,* pp. 107-121, 2012.

[6] L. A. H. and A. G. Doig, "An Automatic Method of Solving Discrete Programming Problems," *Econometrica vol.28, no. 3,* pp. 497-520, 1960.

[7] D. Davendra, Traveling Salesman Problem, Theory and Applications, InTech, 2010.

[8] J. D. C. Little, "Branch and Bound methods for combinatorial problems," Sloan school of Management, M.I.T., Cambridge, Massachusetts, 1966.

[9] M. Benichou and J. Gauthier, "Experiments in mixed-integer linear programming," *Mathematical Programming,* pp. 76-94, 1971.

[10] D. Applegate and R. E. Bixby, "Finding cuts in TSP," DIMACS, March 1995.

[11] M. Gasse and D. Chételat, "Exact Combinatorial Optimization with Graph Convolutional Neural Networks," in *NeurIPS*, 2019.

[12] T. Achterberg, "Constraint Integer Programming," 2007. [Online]. Available: https://opus4.kobv.de/opus4-zib/files/1112/Achterberg_Constraint_Integer_Programming.pdf.

[13] N. Metropolis, "THE BEGINNING of the MONTE CARL0 METHOD," no. Special Issue.

[14] D. Billings, L. Peña and J. Schaeffer, "Using Probabilistic Knowledge and Simulation to Play Poker," 1999.

[15] B. Bouzy and B. Helmstetter, "Monte-Carlo Go Developments," 2004.

[16] R. Coulom, "Efficient selectivity and backup operators in Monte-Carol tree search," 2006.

[17] D. Silver, A. Huang and C. J. Madisson, "Mastering the game of Go with deep neural networks and tree search," no. 529, pp. 484-489, 2016.

[18] D. Silver, J. Schrittwieser and K. Simonyan, "Mastering the game of Go without human knowledge," no. 550, pp. 354-359, 2017.

[19] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo Planning," Berlin, 2006.

[20] I. Bello, H. Pham and Q. V. Le, "Neural Combinatorial Optimization with Reinforcement Learning," 2017.

[21] O. Vinyals, M. Fortunato and N. Jaitly, "Pointer Networks," 2005. [Online]. Available: https://arxiv.org/pdf/1506.03134.pdf.

[22] Y. Tang, S. Agrawal and Y. Faenza, "Reinforcement Learning for Integer Programming: Learning to Cut," 2020. [Online]. Available: https://arxiv.org/pdf/1906.04859.pdf.

[23] V. Ashish, N. Shazeer and N. Parmar, "Attention Is All You Need," 2017.

[24] K. Abe, Z. Xu and M. Sugiyama, "Solving NP-Hard Problems on Graphs with Extended AlphaGo Zero," 7 March 2020. [Online]. Available: https://arxiv.org/pdf/1905.11623.pdf.

[25] M. Colombi, R. Mansini and M. Savelsbergh, "The generalized independant set problem: Polyhydral analysis and solution approaches," no. 260, 2017.

[26] A. Coutou, J.-B. Hooc and N. Sokolovsk, "Continuous Upper Confidence Trees," [Online]. Available: https://hal.archives-ouvertes.fr/hal-00542673v1/document.

[27] [Online]. Available: https://github.com/ds4dm/learn2branch.

[28] [Online]. Available: https://github.com/cromagnonninja/Learn2SelectNodes.