This thesis was financed by a doctoral scholarship from the Portuguese Foundation for Science and Technology (reference SFRH/BD/66075/2009) under the POPH program, financed by the European Social Fund and the Portuguese Government.

Esta tese for financiada por uma bolsa de doutoramento da Fundação para a Ciência e Tecnologia (referência SFRH/BD/66075/2009) no âmbito do programa POPH, financiado pelo Fundo Social Europeu e pelo Governo Português.

Acknowledgements

First of all, I would like to thank my good friend and supervisor, Professor João Pedro Pedroso. He has been, for the good part of a decade now, the best supervisor a student can ask for. My growth as a Computer Scientist has to be credited to João Pedro, who has introduced me to the most interesting and difficult challenges I have come to know. He has always been available to discuss ideas or even help out reviewing papers, giving me freedom to pursue my own paths, and sharing his vast knowledge in Operations Research and Computer Science. It has been my pleasure and honor to work with him.

I dedicate this thesis to my wife, Sandra, who has been my pillar through difficult times. She kept believing in me regardless of failure, and pushed to me stand up to the most difficult challenge I have faced thus far.

Last, but not least, I would like to thank my family and friends. This thesis would not have been possible without the support and encouragement of all of the aforementioned people. It is thanks to them—and for them—that I now close this chapter of my life. I am forever indebted for everything they have done for me, and sure that these words cannot sufficiently express my gratitude.

Abstract

Tree search methods occupy a central role in the field of combinatorial optimization. Best-first based branch-and-bound algorithms are used by general-purpose solvers as the primary means to tackle combinatorial problems that can be formulated as mixed integer linear programs. These general solvers are extremely valuable because they are able to solve a wide range of problems by providing only a mathematical formulation.

However, there are many cases in practice where these solvers show poor performance: loose bounds prevent effective pruning; the search becomes trapped in a narrow region of the search space; leaf nodes are not reached within reasonable time; early decisions in the solution construction process can only be accurately assessed at the end of the construction. Additionally, the task of embedding domain-specific knowledge into general solvers is often non-trivial.

Monte Carlo tree search (MCTS), a method originating from game-playing, combines tree search with Monte Carlo simulations to quickly obtain full evaluations of partial states and guide the traversal of the search space. Due to its fundamental structure being extremely abstract and not requiring expert domain knowledge, the algorithm can perform reasonably in myriad problem domains with little implementation effort; furthermore, domain knowledge can be incorporated at several stages of the algorithm for problem-specific improvements. Since its inception, MCTS has attracted a lot of attention from the Artificial Intelligence community and many variants and enhancements have been developed. The Upper Confidence Bounds for Trees (UCT) algorithm is by far the most widely used variant, which improves upon the base algorithm by addressing the *exploitation-exploration* dilemma in an opti-

mal manner. MCTS/UCT has achieved considerable success in several game-playing tasks, and possesses characteristics which also make it suitable for optimization.

The goal of this thesis is to illustrate, through applications in different combinatorial problems, that MCTS/UCT is a powerful method for optimization. Given its qualities—general, aheuristic, anytime—this method is suitable for both approximative and exact use. The MCTS framework is also easily extendable, allowing the incorporation of problem-specific heuristics whenever these are available.

Due to the somewhat different nature of its original context of game-playing, we first examine the differences between game-playing and combinatorial optimization, to then propose adaptations of the UCT algorithm. The effectiveness of the proposed adjustments is subsequently assessed on a set of applications to combinatorial problems, including number partitioning, stacking, and recursive circle packing.

Keywords: tree search; branch-and-bound; Monte Carlo methods; combinatorial optimization.

Resumo

Os métodos de pesquisa em árvore ocupam um lugar de destaque no campo da optimização combinatória. Algoritmos de ramificação e limitação baseados em *best-first* são utilizados por resolutores genéricos como o principal método para abordar problemas combinatórios formulados como modelos de programação (linear) inteira mista. Estes resolutores genéricos são extremamente valiosos porque são capazes de resolver um vasto leque de problemas, necessitando apenas de uma formulação matemática.

No entanto, existem vários casos práticos em que estes resolutores têm fraco desempenho: majorantes/minorantes folgados previnem um uso efectivo de técnicas de poda; a pesquisa fica confinada a uma zona restrita do espaço de procura; as folhas da árvore não são visitadas em tempo útil; decisões tomadas no início da construção de uma solução só podem ser avaliadas com precisão no final da construção. Adicionalmente, a incorporação de conhecimento especializado nestes resolutores genéricos é frequentemente não-trivial.

Pesquisa em árvore de Monte Carlo (MCTS), um método originário da investigação em jogos, combina pesquisa em árvore com simulações de Monte Carlo para rapidamente obter avalições completas de estados parciais e com isso guiar a travessia do espaço de procura. Devido ao facto de a sua estrutura fundamental ser extremamente abstracta e não obrigar a conhecimento do domínio, o algoritmo consegue obter um desempenho razoável numa miríade de problemas com pouco esforço de implementação; além disso, conhecimento do domínio pode ser incorporado em vários pontos do algoritmo para obter melhorias específicas a cada problema. Desde o seu início, o método MCTS tem recebido bastante atenção da comunidade de

Inteligência Artificial e, consequentemente, muitas variantes e melhorias foram desenvolvidas. O algoritmo *Upper Confidence Bounds for Trees* (UCT) é de longe a variante mais utilizada, e melhora o MCTS resolvendo de forma óptima o dilema intensificação-exploração. O algoritmo MCTS/UCT obteve successos consideráveis em vários jogos, e possui características que o tornam apropriado também para optimização.

O objectivo desta tese é mostrar, através de aplicações em diferentes problemas combinatórios, que o algoritmo MCTS/UCT é uma poderosa ferramenta para optimização. Dadas as suas qualidades—geral, independente de heurísticas, interrompível a qualquer altura—este método é apropriado tanto para uso aproximativo como exacto. O algoritmo também é facilmente extensível, permitindo a incorporação de heurísticas específicas ao problema caso estas existam.

Devido ao contexto de onde o MCTS/UCT surge, primeiramente examinamos as diferenças entre os contextos de jogos e optimização, para então propôr algumas adaptações ao algoritmo UCT. A eficácia destas adaptações é avaliada num conjunto de aplicações em problemas combinatórios, que inclui o problema de particionamento de números, empilhamento, e empacotamento recursivo de círculos.

Palavras-chave: pesquisa em árvore; ramificação e limitação; métodos de Monte Carlo; optimização combinatória.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Context

Operations research (OR) is a sub-field of applied mathematics that is concerned with finding optimal values for a set of parameters controlled by a decision maker, under certain problem-specific constraints, such that a given function of those parameters is minimized (*e.g.*, cost) or maximized (*e.g.*, profit). Using mathematical notation, an optimization problem[1] can be formulated in a very general manner as:

$$\begin{aligned} &\text{minimize} && f(x) \\ &\text{subject to} && x \in \mathcal{X} \end{aligned}$$

where x represents the set of parameters that are controlled by the decision maker (known as *decision variables*), and the function $f(x)$ that is being optimized is called *objective function*. The domains of the decision variables and constraints of the problem are captured in the set $\mathcal{X}$ from which decision variables can take values; $\mathcal{X}$ is also known as the *feasible region*.

[1] Although the text refers specifically to minimization, maximization can be handled in the same manner by minimizing a function $g(x) = -f(x)$, where $f(x)$ is the original function we wish to maximize. Without loss of generality, we will only consider minimization problems in the discussion.

The objective of optimization is then to find $x^* \in \mathcal{X}$ such that

$$\forall x \in \mathcal{X}, f(x) \geq f(x^*)$$

or, equivalently,

$$x^* = \arg\min_{x \in \mathcal{X}} f(x)$$

The solution x^* is said to be an *optimal solution*.

A *combinatorial optimization* problem is an optimization problem in which the feasible region $\mathcal{X}$ is (or can be reduced to) a discrete set. Combinatorial optimization is an extremely important and extensively studied topic because many practical problems (scheduling, facility location, packing, vehicle routing, *etc*) can be formulated as combinatorial problems. In this thesis we will focus exclusively on combinatorial optimization problems.

Due to the so-called "combinatorial explosion", which is the exponential growth in the size of the search space arising from the product of the domains of decision variables, many problems cannot be solved exactly except for very small instance sizes. Many of these problems belong to the complexity class *NP-hard*[2], and it is widely believed that efficient (*i.e.*, polynomial-time) algorithms for such problems do not exist.

The approaches used to deal with combinatorial problems can be divided into *exact methods* and *approximative methods*. While an exact method may require exhaustive enumeration of the search space, an approximative method simply searches for the highest-quality solution possible within a given computational budget (*e.g.*, time or number of iterations). Steady progress in computer hardware and advances in solver software such as mixed integer linear programming solvers has pushed up the limits on the size of instances that can be solved exactly. However, in many real-life applications, that limit is still not high enough and practitioners are forced to resort to approximative methods. In such cases, obtaining good quality solutions in a reasonable amount of time is more valuable than obtaining optimal solutions,

[2] *NP-hard* is the class of optimization problems whose decision versions are *NP-complete* problems. These problems are informally said to be as *"at least as hard as NP-complete"*.

since the process of proving a solution is optimal may take an excessive amount of time.

Combinatorial problems naturally lend themselves to a *tree* structure. The search space can be seen as a rooted tree with nodes holding information about a particular partial/complete solution: the node at the root of the tree holds an empty solution, *i.e.*, having all variables unassigned; internal nodes contain partial assignments, *i.e.*, some variables are yet to be assigned; and leaf nodes contain complete solutions. *Tree search* algorithms provide a way to systematically traverse this tree structure without revisiting solutions[3] and guaranteeing that the full set of feasible solutions is covered. The latter property essentially ensures that, if the whole tree is explored, then the best solution found is optimal. A discussion of different tree search strategies and their trade-offs is given in section 2.1.

Branch and bound is an enhancement to tree search algorithms which relies on a bound function that, given a partial solution, can provide a bound on the best objective value that can be reached under the associated subtree. If the bound value is not better than the incumbent solution, then the node being examined can be discarded safely without sacrificing optimality. Depending on the tightness of the bound function, this technique may provide a dramatic reduction of the effective size of the search tree, thereby allowing larger instances to be solved.

In recent years, a lot of Artificial Intelligence research has been devoted to *Monte Carlo tree search* (MCTS). This has mainly been driven by its successes in computer Go [SHM+16, TKY08, GKS+12], but it has also been used successfully in other games and it has even seen a few applications in OR [Fin07, SSR12, RTC11]. The algorithm is based on the idea of using Monte Carlo simulations—randomized constructions starting from a given partial solution—to estimate nodes' game-theoretic values, and employing those estimates to drive the traversal of the search space. MCTS possesses interesting properties such as being a robust anytime algorithm, asymmetrically constructing the search tree, and most importantly, being capable of dealing with problems lacking good heuristics. The latter means that MCTS is well suited for domains where traditional methods usually fare poorly.

[3] Assuming the tree structure is free of symmetries.

Another advantage of MCTS lies in its generality and flexibility. In its simplest form, it can be used without any domain-specific knowledge—simulations can be done through uniform action selection at each step. However, should it exist, domain-specific knowledge can easily be leveraged to improve the algorithm's performance. OR research has consistently shown that, for many problems, state-of-the-art solution methods employ a great deal of specialized knowledge. Nonetheless, general problem solvers are valuable in that they can be used to tackle a wider range of problems and require significantly less development effort. A prime example of this is in mixed integer linear programming (MILP) solvers, which are used to tackle problems that can be modeled as mixed integer linear programs. In many cases, these solvers are even able to solve practical-sized instances, obviating the need for specialized solvers.

1.2 A general Monte Carlo tree search for combinatorial optimization

For the qualities mentioned above, we strongly believe that Monte Carlo tree search can occupy an important place in combinatorial optimization and OR in general. Its ability to cope with the lack of tight bounds or good heuristics can make it a relevant tool for the OR practitioner. Traditional methods applied to such problems are either unable to find feasible solutions or produce solutions of low quality. MCTS has already proven successful in problems with similar characteristics from other domains, and we attempt to transfer those successes into combinatorial optimization. We also show that, even outside of its comfort zone, in the realm of more "normal" problems for which good solution methods are available, MCTS can still be an effective alternative and provide competitive results. In short, the aim of this thesis is to illustrate the potential of MCTS as a general tool for combinatorial optimization.

As the algorithm is rooted in game-playing, we begin by presenting some adaptations for the context of optimization, and proceed to demonstrate its effectiveness through a few applications and computational experiments. We hope that, by the end of this thesis, the reader can view MCTS as a viable approach to some of the hardest

combinatorial problems and, owing to the algorithm's extreme flexibility, perhaps even consider its application in wildly different scenarios. In that sense, MCTS can be seen as a general framework for search and optimization problems.

1.3 Overview of the thesis

Chapter 2 presents concepts and terminology used throughout the remaining chapters. An overview of tree search methods is given, discussing desirable characteristics of an ideal method. Then, we introduce the Monte Carlo tree search algorithm and some of its most popular variants and enhancements, as a basis for our implementation.

In chapter 3, we describe adaptations done to the MCTS algorithm, and issues arising from its usage in the context of optimization. Since our aim is to create a general problem solver, we need to correctly handle problems where finding feasible solutions is in itself a difficult task. Also, we show how we can incorporate a pruning mechanism (as in branch-and-bound) that allows us to reduce the size of the search tree without sacrificing optimality in problems where good bounds are available.

We follow up with the presentation of a few applications of our adapted MCTS algorithm. As a starting point, chapter 4 presents the number partitioning problem, the well-known Karmarkar-Karp heuristic, and a basic depth-first tree search based on this heuristic. A computational experiment is conducted on a small set of benchmark instances, comparing the two tree search approaches.

Then, in chapter 5, we look at the stacking problem, a problem arising from a practical application in the storage and handling of steel, but which also has other applications such as container ship stowage planning and automated parking lots.

Another practical problem, the recursive circle packing problem is shown in chapter 6. This non-linear continuous packing problem originates from the tube industry, and is handled approximatively through a discretization technique that allows its treatment as a combinatorial problem.

These different applications are given in an effort to demonstrate the underlying

qualities of an MCTS approach: effective use of Monte Carlo simulations to find complete solutions early; generality/flexibility; simple application to new problem domains; viable in the absence of specialized knowledge, but capable of incorporating such domain-specific insights.

We conclude with chapter 7, where we summarize our results and contributions, and highlight possible directions for future research.

Chapter 2

Background

In this chapter, we introduce basic concepts and terminology related to tree search, branch-and-bound, and Monte Carlo tree search, presenting an overview of previous work on those topics. Popular enhancements of the basic MCTS algorithm are also described, as some of those techniques will be applied later in our own implementation.

2.1 Tree search

Tree search is a method for systematically exploring the search space of a combinatorial problem, with the aim of finding the best solution appearing therein. In its simplest form, all solutions are enumerated and verified, hence taking exponential time with respect to the number of variables. In optimization problems for which there is an appropriate mathematical programming formulation, it is usually easy to find bounds on the solutions that can be obtained from a node, which allows the search tree to be pruned without loss of optimality. For a minimization problem, if the lower bound in a given node is greater than the value of a known solution (*i.e.*, greater than an upper bound on the optimum), then the tree can be safely pruned at this node. This is the standard procedure in branch-and-bound (BB) [LD60, LW66]. Literature on BB is ample, as the technique is fully general

and can be applied in widely diverse areas. For a sample of recent applications of BB methods see *e.g.*, [BCL12, NWCL10, BLK+13, DGRW12].

For the problems we will be dealing with, formulations are very loose, rendering the provided bounds very ineffective and leading to very little or no pruning at all. This implies that, except for toy instances, exploration of the whole tree is unreasonable, either due to time limitations or because the size of the tree would grow unacceptably large. Consequently, the best solution found may not be optimal because the search space has not been fully explored. In this context, tree search may be used as an *approximative* algorithm.

Algorithm 2.1 presents a general tree search procedure using branch-and-bound, given objective function $f(x)$ and a bound function $\beta(x)$. For the sake of simplicity, in Algorithm 2.1 we use the notation $f(n)$ and $\beta(n)$ as a shorthand for $f(x_n)$ and $\beta(x_n)$, respectively, where x_n denotes the solution associated with a node n. Note that the order by which nodes are visited has a great impact on the performance of a tree search. To illustrate this idea, consider as a simple example a full (*i.e.*, exact) tree search: if an optimal solution is found immediately at the start of the search, then the maximum number of branches is pruned throughout the search and the effective size of the tree is minimum; if, on the other hand, the search first finds low-quality solutions, it will invest time searching branches that would otherwise already be pruned. Therefore, the time required for a complete search is directly determined by traversal path. When considering an incomplete (*i.e.*, approximative) search, visiting nodes containing good solutions early is obviously desirable and has a direct impact on the solution quality aspect of the algorithm's performance.

Below we discuss different strategies for traversal of the tree, some of which can be described as different implementations of the steps in lines 7 (dequeuing a node) and 18 (enqueuing a node) of algorithm 2.1. Note that, if the main loop terminates due to the queue becoming empty, then the solution x^* is optimal.

In *uninformed search* there is no use of information concerning the value of a node during tree exploration. For example, in *breadth-first search* all nodes in a level of the tree (*i.e.*, nodes that are equally distant from the root) are explored before proceeding to the next level. This can be implemented in the framework shown in

```
1  function TreeSearch (problem instance I)
2      r ← create root node with initial state from I
3      Q ← {r}                                              // queue of open nodes
4      n* ← undefined                          // node containing best solution found
5      z* ← ∞                                               // best objective value
6      while Q ≠ ∅ and computational budget is not depleted do
7          n ← take a node from Q
8          if n is a leaf then
9              z ← f(n)                                // objective function evaluation
10             if z < z* then                                          // new incumbent
11                 n* ← n
12                 z* ← z
13                 Q ← Q \ {n' ∈ Q : β(n') ≥ z*}                              // pruning
14         else
15             C ← children of node n                                    // expansion
16             for c ∈ C do
17                 if β(c) < z* then                                   // bound check
18                     insert c into Q
19     return n*
```

Algorithm 2.1: General branch-and-bound procedure for minimization problems.

algorithm 2.1 by taking an element from one end of the queue in the *dequeue* step, and appending new nodes to the opposite end of the queue in the *enqueue* step.

In *depth-first search* (DFS) each node is expanded down to the deepest level of the tree, where a node with no expansion (*i.e.*, a *leaf*) is found; search then backtracks until a node with unexplored children is found, which is again expanded until reaching a leaf, an so on, until the whole tree is explored. This can be achieved by both dequeuing from and enqueuing into the same end of the queue in algorithm 2.1. As only one path from the root to a leaf has to be stored at any given time, DFS has modest memory requirements. DFS can leverage problem-specific heuristics to order/score the actions available from a given node (equivalently, state). Building on such a heuristic, we can obtain a simple greedy construction algorithm by selecting at each step the action preferred by the heuristic. This is the base behavior for heuristically-guided DFS, which constructs the greedy solution on its first dive down the tree, and subsequently attempts variations on these greedy decisions by decreasing order of preference. As the heuristic considers only local information, *i.e.*, it sorts/scores only actions available from a given node, this is usually called *partially informed search*.

On *informed search*, a set of open nodes is used; the most common variant is *best-first search*, which selects the next node to expand based on problem-specific knowledge. To this end, an *evaluation function* is used which conveys information about the worth of each open node, and the one with the highest rating from all open nodes is selected at each iteration. Best-first search can also be implemented within the framework of algorithm 2.1 by taking the first open node from the queue in the *dequeue* step, and inserting new nodes into the queue according to the rating obtained from the evaluation function in the *enqueue* step.

When good guiding heuristics exist, DFS is usually very effective. When compared to greedy construction based on the same heuristics, DFS allows for substantial improvements; furthermore, these improvements can be obtained very quickly due to the simplicity of DFS, which imposes very little overhead over the greedy construction algorithm. For situations where exploration of the full tree is expected to be possible in a reasonable time, DFS is usually an appropriate choice. However, this is

not the case for most practical problems. For sufficiently large trees, DFS is unable to recover from poor decisions taken at the beginning of the search.

For trees with high branching factor, *iterative broadening*, proposed in [GH92], attempts to overcome the limitations of DFS by running a sequence of restricted depth-first searches. Each restricted DFS considers a larger subset of the tree than the previous: the first iteration examines only the heuristically preferred node, the second iteration examines the two top-ranked children of each node, and so on; in a tree of depth d, at iteration k, iterative broadening visits up to k^{d-1} leaves.

Best-first search takes a very different approach from DFS and considers, at each iteration, nodes from different levels in the tree. A seminal example is the $A*$ algorithm for the shortest-path problem [HNR68], where heuristic information that never overestimates the cost of the best solution reachable from a node is used to evaluate it. Best-first is not free of drawbacks though, as in the worst case it requires exponential space, thus becoming impractical in some scenarios. In fact, this feature is common to many tree search variants that maintain a list of open nodes, including MCTS. Also, best-first search may still be caught in a restricted region for an extended period of time, since traversal order is solely driven by the evaluation heuristic, which does not contain any built-in mechanism to prevent such occurrences.

A related algorithm taking only linear space is *iterative deepening* $A*$ (IDA*): each iteration is a DFS modified to use heuristic evaluation as in A*, and a limit on the heuristic value to interrupt the search; the interruption threshold is increased at each iteration [Kor85]. In practice, for many combinatorial optimization problems IDA* visits too many internal nodes before reaching its first leaf; this is mainly due to the underestimation provided by the heuristics being substantially different from the best value that can be reached from each node. Besides, IDA* does not cope well with the fact that in many optimization problems every node has a different heuristic value, only the best of which is expanded at each iteration. In combinatorial optimization problems the depth of the search tree is bounded, and reaching a leaf is usually inexpensive; this is not exploited in IDA*.

The most widely used tools for solving combinatorial problems that can be formu-

lated as mathematical programming models are mixed integer linear programming (MILP) solvers. These usually incorporate state-of-the-art tools in terms of pruning the search tree and adding cutting planes to the model, in a black-box solver targeted at obtaining the best performance in the widest range of problems possible. Still, deficiencies similar to those of DFS are often observed in practice, as the solvers may be stuck in the search for very long periods. Methods for overcoming this issue have been studied, following the observation of heavy-tailed phenomena in satisfiability and constraint satisfaction [GSCK00]. The basic idea is to execute the algorithm for a short time limit, possibly restarting it with some changes, usually with an increased time limit until a feasible solution is found. This has been recently addressed in [FM14], where variability of the solutions on which the MIP solver is trapped, for different initial random start conditions, is exploited. The proposed method consists in making a number of short runs with randomized initial conditions, bet on the most promising run, and bring it to completion, in an approach named *bet-and-run*. Diversified runs are produced in several fronts: exchanging rows and columns in the input instance, perturbing parameters of the MIP solver, and changing coefficients in the objective function. The choice of the candidate to bring to completion is based on 11 criteria, the most important of which are the number of open nodes, the lower bound improvement with respect to the root node, and number of integer-infeasible variables among all open nodes. Results are reported for a set of benchmark instances, showing that bet-and-run can lead to significant speedups.

We have looked at a few different strategies for tree search, each with its own advantages and disadvantages, and there are still plenty more strategies that could not be covered in this text. Next, we describe Monte Carlo tree search, an alternative that possesses interesting characteristics which position it as a promising candidate for combinatorial optimization.

2.2 Monte Carlo tree search

Monte Carlo tree search (MCTS) is a tree search algorithm that makes use of Monte Carlo simulations to estimate node values, and uses those estimates to drive the choice of next node to explore. The goal of MCTS is to approximate the game theoretic value of the actions that may be taken from the given initial state.

Although the idea of combining Monte Carlo evaluation with tree search had been studied before (see *e.g.*, [Bou06, Jui99]), it was not until the appearance of MCTS that it received greater scientific attention. Coulom [Cou07] proposed the initial version of MCTS and applied it with considerable success to the game of Go (9×9 board). Game-playing is still the area where the algorithm and its many variants are most commonly used [WBS08]. MCTS has had a particularly strong impact on games, where its application has led to the best game-playing software, most notably for the game of Go [TKY08, GKS+12]. The highest-profile feat of MCTS occurred very recently, with DeepMind's AlphaGo [SHM+16] program defeating the many-time world champion Lee Sedol in a five-game match on a full board with no handicap, with a final score of 4-1. The computer Go field has been dominated by MCTS since its inception, but AlphaGo took performance to the highest ever levels by combining MCTS with deep neural networks for position evaluation and action selection in simulations.

MCTS has also been applied on artificial intelligence approaches for other domains that can be represented as trees of sequential decisions and for planning problems (see [BPW+12] for a comprehensive survey).

2.2.1 Basic algorithm

Monte Carlo tree search is based on two fundamental concepts: the true value of a node may be estimated through randomized simulations (*i.e.*, Monte Carlo evaluation); and those estimates can be used to guide the traversal of the tree in a best-first manner. As more simulations are made, the accuracy of these estimates is expected to increase, hence performance will generally improve with additional computation

time.

MCTS starts with a tree consisting only of a root node created from a given initial state, and incrementally adds new nodes to it in a nonuniform manner, *i.e.*, new nodes can be added at any location in the current tree. During each iteration of the algorithm, a *tree policy* uses accumulated node statistics—these normally include at least the number of visits and the total reward obtained—to select the most "promising node" to explore. Then, a new child node[1]—created by randomly choosing an unexplored action from the selected node—is added to the tree. A *simulation* is run from the newly created node, making use of a *simulation policy*[2] to choose actions until a terminal state is reached. The simulation policy can select uniform random actions in the simplest case, or use more sophisticated rules to bias action selection favorably. Finally, once the simulation is completed, the terminal state reached is evaluated and backpropagated, *i.e.*, its end result is used to update the statistics of all nodes from the new node to the root of the tree. This process is iterated until either the full tree has been explored or, most likely, a computational budget is depleted, at which point the action leading to the "best child" of the root node is returned. As will be discussed later, both the notion of "promising node" and "best child" are open to different interpretations, and are left to the implementation.

The base MCTS described above is summarized in algorithm 2.2. Here, s_0 is the initial state from which the root node n_0 is created. The selection step is performed by the function *TreePolicy*, which returns an existing node n_1 that has one or more unexplored actions. From that set of actions, an action a is randomly selected and applied to obtain the new node n_2. Using the function *SimulationPolicy*, a Monte Carlo simulation is executed from n_2, yielding a reward Δ that is finally backpropagated, updating the statistics of all nodes from n_2 up to n_0.

Figure 2.1 illustrates the four main steps of an iteration of MCTS: *selection*, *expansion*, *simulation*, and *backpropagation*[3]. These steps are described in more detail in the following sections.

[1] In some implementations, multiple children may be added per iteration.
[2] The simulation policy is also called *default policy* in the literature.
[3] Image by *Mciura*, taken from https://en.wikipedia.org/wiki/Monte_Carlo_tree_search.

```
1 function MCTS(s_0)
2     n_0 ← root node with initial state from s_0
3     while within computational budget and there are open nodes do
4         n_1 ← TreePolicy(n_0)                          // selection
5         a ← random unexplored action from n_1          // expansion
6         n_2 ← copy of n_1 with a applied               // expansion
7         Δ ← SimulationPolicy(n_2)                      // simulation
8         Backpropagate(Δ, n_2)                          // backpropagation
9     return BestChild(n_0)
```

Algorithm 2.2: Basic Monte Carlo tree search.

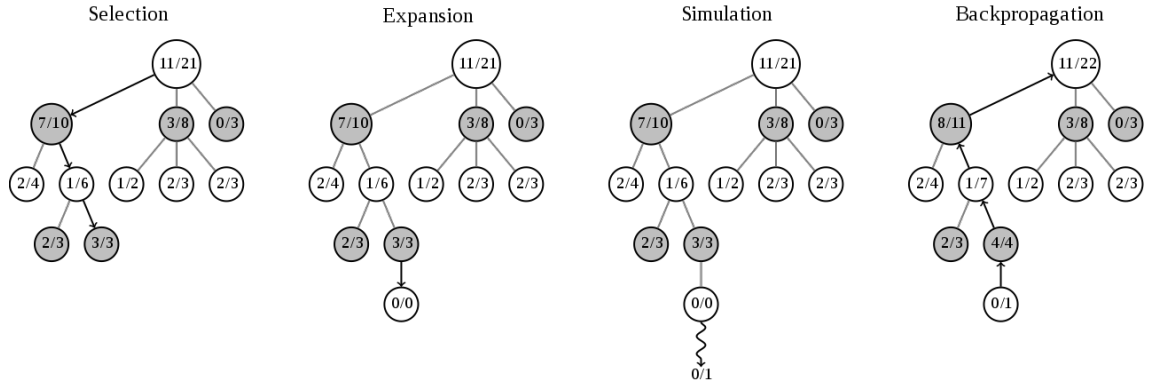

Figure 2.1: The four steps of a single iteration of Monte Carlo tree search. The numbers within the nodes represent the total reward (left) and number of visits to the node (right). The figure depicts a search tree for a two-player game.

2.2.1.1 Selection

Starting from the root node, the child which currently looks more "promising" is selected. This is done recursively until a node n which has not yet been fully expanded (*i.e.*, some of its children have not yet been added to the tree) is reached. The definition of promising is one of the key aspects determining the performance of MCTS, and can be represented by a utility function $U(n)$ that we wish to maximize at each level of the tree. In plain MCTS, the node with maximum average reward is selected, that is,

$$U(n) = \frac{1}{v_n} . \sum_{i=1}^{v_n} \Delta_{n,i} \tag{2.1}$$

where v_n is the number of visits to node n (or, equivalently, the number of simulations run from n), and $\Delta_{n,i}$ is the reward obtained in the i-th simulation run from node n. Note that the visits and rewards from all nodes n' in the subtree under node n also count towards the statistics of node n; in other words, when statistics are updated for a node, they are also updated for all its ancestors (this pertains to the backpropagation step and is detailed in section 2.2.1.4). The selection procedure is presented in pseudo-code form in algorithm 2.3 below.

```
1 function Selection(n)
2     if n has unexplored actions then
3         return n
4     else
5         C ← set of children of n
6         n' ← arg max_{c∈C} U(c)
7         return Selection(n')
```

Algorithm 2.3: Selection step in Monte Carlo tree search.

The observant reader might notice that, using equation (2.1) as measure of utility may result in an over-exploration of the region near a high-reward solution. If this phenomenon occurs for a prolonged period of time, a partial search may end up missing better alternatives. Section 2.2.2 presents the UCT variant, which employs a different utility function $U(n)$ that takes into consideration the *exploration-exploitation* dilemma. In other words, it balances exploitation of high-reward nodes with exploration of new nodes, in order to minimize the probability of missing the optimal solution.

2.2.1.2 Expansion

One or more children of the selected node n (the result of the *Selection* function in algorithm 2.3 above) are created by applying unexplored actions to the state associated with n. These new nodes are then attached to the tree, hence the asymmetrical construction of the search tree. The two main schemes for expansion are:

***Single expansion*:** a single child node is created using an unexplored action from

node n. The remaining unexplored actions are recorded for a later time when node n is again selected for expansion. The action to explore can be selected randomly (as in the base MCTS) or based on some heuristic evaluation.

***Full expansion*:** all children of node n are immediately created by generating all possible actions. Using this scheme, the order of creation of the child nodes is irrelevant, since all of them are created in the same iteration.

2.2.1.3 Simulation

Starting from the state of the node (or nodes) created in the expansion step—hereby called *initial node*—available actions are iteratively applied until a terminal state is reached. Various approaches can be taken in simulation, ranging from uniform random decisions—requiring nothing more than a generative model of the problem—to heuristic construction algorithms incorporating domain-specific knowledge. The latter approach typically allows for faster convergence at the expense of simplicity and generality. The general simulation process is presented in pseudo-code form in algorithm 2.4 below.

```
1 function Simulation(n)
2     while n is not terminal do
3         A ← set of available actions in n
4         a ← select action from A
5         n ← copy of n with a applied
6     return f(n)                          // reward/objective value
```

Algorithm 2.4: Simulation step in Monte Carlo tree search.

Note that the steps in lines 3 and 5 of the *Simulation* function require only a generative model of the problem domain, *i.e.*, how to enumerate available actions, and how to apply an action to modify a given state. Although this is evidently problem-dependent, this is not what is referred to when talking about *specialized knowledge*. Specialized knowledge refers to the utilization of heuristics or certain problem-specific properties to aid in the search for good solutions.

When incorporating specialized knowledge in MCTS, the foremost place where this

can be done is in the action selection of line 4. In addition to problem-specific construction heuristics, there are also some more general approaches that attempt to bias action selection to improve performance over uniform random choice. As an example, the AlphaGo program [SHM+16] uses a deep neural network to that end. Although the particular parameterization of the neural network is tied to the problem of playing Go, the deep learning technique is independent of the problem and no heuristics are hand-crafted.

2.2.1.4 Backpropagation

Once a simulation has reached a terminal state, its reward Δ is evaluated. Starting with the simulation's initial node n, and going up the tree all the way until the root node is reached, node statistics are updated to include the newly found observation Δ. Algorithm 2.5 below shows a simple backpropagation procedure.

```
1 function Backpropagation(Δ, n)
2     while n is not null do
3         increment visit count for n
4         add reward Δ to node n
5         n ← parent of node n                    // move up the tree
```

Algorithm 2.5: Backpropagation step in Monte Carlo tree search.

As will be discussed in later sections, the set of node statistics that we wish to keep track of can vary according to the application domain. In game-playing, the total reward obtained from a node is tracked in addition to its number of visits. Using different statistics, the backpropagation step must of course be adapted accordingly.

2.2.2 UCT

In this section, we describe the most popular variant of MCTS, named UCT—Upper Confidence Bound for Trees. We begin with a short introduction to the multi-armed bandit problem, the concept of regret, and the UCB1 policy.

2.2.2.1 Multi-armed bandit problem

The multi-armed bandit problem is a sequential decision problem where a player is presented with a row of k slot machines[4] and, at each step, must select a machine to play, with the goal of collecting the maximum total reward. It is assumed that each machine i has an unknown underlying distribution of rewards with unknown expected value μ_i.

The choice of machine to play is difficult since nothing is known *a priori* about the reward distributions; the player can only estimate future rewards based on past observations. In this scenario, the player is faced with the *exploitation-exploration dilemma*: he can either select the machine currently believed to be optimal (*i.e.*, the one with highest average reward), or a less explored machine that currently looks sub-optimal but may turn out to be superior. The player's strategy can be defined as a *policy* function that, given a history of previous plays, selects the next action.

Regret is defined as the expected loss incurred due to a sub-optimal decision. The regret R_t at step t is defined as

$$R_t = \mu^* - \sum_{i=1}^{k} \mu_i P_{i,t} \tag{2.2}$$

where μ^* is the best possible expected reward, and $P_{i,t}$ is equal to 1 if and only if machine i is selected at step t. The original objective of reward maximization can be equivalently expressed as regret minimization.

It has been shown in [LR85] that, for a wide class of reward distributions, there exists no policy whose regret grows slower than $O(\ln n)$. Therefore, a policy is deemed to "solve" the exploitation-exploration dilemma if it can attain a rate of regret growth that is within a constant factor of this theoretical bound. In [ACBF02], a family of policies is proposed, which make use of *upper confidence bounds* (UCB) that a given bandit is optimal. The simplest of these UCB policies—called UCB1—has an expected logarithmic growth of regret, requiring only that rewards take values in the interval $[0, 1]$. At step $t + 1$, UCB1 chooses the bandit which maximizes the

[4] Also known as one-armed bandits.

expression

$$\bar{\mu}_i + \sqrt{\frac{2 \ln t}{v_i}} \tag{2.3}$$

where $\bar{\mu}_i$ is the average reward from bandit i, t is the number of previous steps/plays, and v_i is the number of times bandit i was chosen. The two terms of this expression represent the conflicting goals of exploitation (first term) and exploration (second term). The exploration term is derived from a confidence bound for the bandit's true expected reward. It seems reasonable then, that whenever a bandit is picked, its exploration value decreases and, simultaneously, the exploration value of all other bandits slightly increases.

Note that, as t grows, the magnitude of the exploration term decreases (the denominator grows faster than the numerator) relative to the exploitation term. It is also worth noting that no bandit is ever permanently discarded because its exploration term monotonically increases with each step where another bandit is chosen; thereby, given enough time, all bandits are bound to be selected.

2.2.2.2 Upper Confidence Bound for Trees

The UCT algorithm is a variant of MCTS that essentially treats the selection step, at each level of the tree, as a multi-armed bandit problem. Child nodes are equated to slot machines, whose rewards correspond to the respective simulation results.

As we have seen in the previous section, this problem can be solved "optimally"—in the sense that it stays within a constant factor of the theoretical best—by employing UCB1 as a tree policy, as proposed in [KS06].

There is a subtle difference between the two problems: in the multi-armed bandit problem, reward distributions are assumed to be stationary; whereas in node selection the distribution of simulation results is constantly shifting due to the dynamic and asymmetric structure of the tree, which in turn affects sampling probabilities.

The UCT algorithm replaces the utility function $U(n)$ used in the selection step

with a modified UCB1 function:

$$\begin{aligned} U(n) &= X(n) + E(n) \\ X(n) &= \bar{\mu}_n \\ E(n) &= C\sqrt{\tfrac{\ln v_p}{v_n}} \end{aligned} \tag{2.4}$$

where $X(n)$ is an exploitation term defined as the average simulation result under node n, and $E(n)$ is an exploration term similar to UCB1, but multiplied by a parameter $C > 0$ that controls exploration weight. The numbers of visits to node n and its parent p are v_n and v_p, respectively. It is shown in [KS06] that using $C = \sqrt{2}$ satisfies the Hoeffding inequality, assuming rewards lie in the interval $[0, 1]$. Despite this theoretical result, the parameter C is usually adjusted empirically per application.

If multiple child nodes are tied for maximum $U(n)$, the tie is usually broken randomly. The UCT score of an unvisited node is considered to be ∞, ensuring that all children of a node are visited at least once before any child can be explored further; this behavior results in a form of local search.

Note that both the UCB1 and UCT score functions provide a balance between exploitation and exploration which depends on the value of the parameter C, and the assumption that rewards lie in $[0, 1]$. The exploration weight parameter may need to be adjusted for reward values outside this range, or when certain enhancements to the base MCTS/UCT are used.

In [KS06], in addition to showing that the logarithmic bound on regret of UCB1 is still valid for non-stationary reward distributions, it is also shown that, given enough time, UCT converges to the minimax (*i.e.*, game-theoretic) values and is thus asymptotically optimal.

2.2.3 Progressive bias

Progressive bias [CWH+08] is a natural and simple technique for adding domain specific knowledge into the selection step of MCTS. It consists in adding a new term

to the selection function, given by

$$f(n) = \frac{H(n)}{v_n + 1} ,$$

where $H(n)$ is a heuristic evaluation of node n, and v_n is the number of visits made to n. The denominator ensures that the weight of the heuristic decreases monotonically as the number of visits increases.

Note that, if using UCT, adding a new term to the selection function may require re-tuning of the exploration coefficient C.

2.2.4 History heuristic

The history heuristic [Sch89] is a domain-independent enhancement that uses global action value estimates gathered during the search. Whenever an action is taken during a simulation, regardless of depth, its global estimate is updated according to the result of the simulation. These estimates can be used in two ways:

- at the selection step, either as an additional fixed-weight term in the selection utility function, or as a replacement of the heuristic function $H(n)$ used in progressive bias; the latter combination is known as *progressive history* [NW10];
- at the simulation step, as an improvement over uniform random action selection.

2.2.5 All moves as first (AMAF)

Another general-purpose MCTS enhancement that is closely related to the history heuristic is a technique called *all moves as first* (AMAF) [GS07]. In this technique, the backpropagation step is modified to update the statistics of all nodes corresponding to any action that was part of the simulation as if it were the initial action. This attempts to reduce the initial warm-up period where actions are still inaccurately estimated

2.2.6 First play urgency

The UCT algorithm, by definition, visits all child nodes at least once (in random order) before any child can be visited again. In trees with high branching factors, this trait of UCT can prevent exploitation of good branches in deeper levels of the tree, because many iterations must be spent fully expanding all nodes in the particular path of interest.

First play urgency [GW06] addresses this issue by assigning a fixed finite UCT score to unexplored children (instead of ∞), and scoring already visited children normally with the UCB1-based utility function. This enhancement introduces a parameter which must be tuned carefully per application, in order to allow and even encourage exploitation of already visited child nodes, but not so much as to prevent exploration of possibly better alternatives.

2.3 Summary

Tree search algorithms allow exploration of the search space for problems that can be modeled as decision trees, *e.g.*, combinatorial games and combinatorial optimization. We have seen that traversal order is a key factor in the performance of a tree search: in a complete search, if pruning (BB) is used, then the number of nodes that must be explored is minimized if good/optimal solutions are found early; for partial/approximative searches, finding high-quality solutions as quickly as possible is precisely the main goal.

Some of the problems with the strategies discussed include:

- being completely oblivious of the problem being solved, *i.e.*, not leveraging specialized information, or even information collected during execution (BFS);
- heavily depending on heuristic functions, and performing poorly when those are unavailable (DFS, best-first search);
- becoming confined in restricted regions of the search space (DFS);
- taking too long to reach a leaf (BFS, IDA* and, possibly, best-first search);

- requiring exponential space (almost all except DFS-based algorithms).

MCTS is a tree search algorithm that utilizes randomized simulations to estimate node values, and uses those estimates to guide the search. Therefore, it is extremely suitable for problems where estimation of node values is hard and no good heuristics exist. When good evaluation heuristics are readily available, traditional algorithms such as minimax usually outperform MCTS; however, these heuristics can also be incorporated into MCTS to improve its performance.

UCT, a variant of MCTS, provides a utility function for node selection that balances exploitation and exploration, guaranteeing that all nodes are eventually considered, given enough time. This is very important, as the random nature of the simulations may conceal good branches which have poor initial estimates. UCT's utility function prevents the search from being permanently trapped in a restricted region. UCT is also shown to be optimal, asymptotically converging towards the minimax values.

MCTS has several important qualities. Due to its use of simulations till terminal states, it is an *anytime* algorithm—it can return a solution independently of how long it is run. Also, in general, it makes good use of extra computational resources; extended time leads to more accurate node estimates, and therefore better performance.

Because it can work "out of the box" without any problem-specific knowledge (taking uniform random actions in simulations), it is considered *aheuristic*. However, specialized knowledge can be embedded in the algorithm to improve its performance.

Of the problems listed above, MCTS does not address the exponential space requirement. This issue is inherent to all methods that make use of a pool/set of open nodes containing nodes from disparate locations in the tree. In practice, because the majority of time is spent on simulations, the growth of the tree does not become unwieldy and is normally not a serious concern.

Chapter 3

Monte Carlo tree search for combinatorial optimization

Applying Monte Carlo tree search to solve optimization problems has many similarities, but also significant differences to its application in game-playing. The size of the search trees in both domains is commonly large enough to prevent complete search within reasonable computational time. Also, for many optimization problems there are no efficient and accurate construction/evaluation heuristics; Monte Carlo simulations fill this gap, by providing both construction and evaluation of partial solutions. One major difference concerns the end product of the search: in game-playing, search is used to estimate the best action from the current game state, and when a reply from the adversary is received, the game state is updated and a new search is performed; in optimization, however, we wish to obtain a complete solution, *i.e.*, an assignment of decision variables that completely solves the problem at hand. Nevertheless, this essential difference does not entail any modification to the MCTS algorithm.

In this chapter, we describe some adaptations deemed necessary to enable MCTS as a tool for combinatorial optimization. We pinpoint the main differences to the algorithm's original domain of game-playing, and propose adjustments to accommodate for the new context. As our aim is to provide a general solver, none of the proposed adaptations have a particular optimization problem in mind.

As a starting point, we will use the ideas from the previous chapter, and build upon the UCT algorithm instead of plain MCTS. UCT is chosen because of its interesting properties, namely the equilibrium between exploitation and exploration, and the theoretical results on growth of regret and convergence to minimax.

An implementation in the Python programming language is described in detail in appendix A.

3.1 Average *vs* best solution

Arguably the most important difference to game-playing concerns the evaluation of nodes and their associated statistics. Whereas in game-playing a branch with a high average win rate is suggestive of a strong line of play, in optimization—since we are interested in finding extrema—the average solution under a node is definitely not a good estimator of the optimal solution to the node's underlying subproblem.

Instead, we propose the use of the best solution found under each node's subtree as a basis for determining its exploitation value, by replacing in equation (2.4) all occurrences of the average simulation result $\bar{\mu}_n$ with the best result b_n. Note that this implies a few rather simple changes to the node statistics and backpropagation step.

3.2 Reward ranges

In games, it is common to use a scoring scheme that assigns a value of 1 to a victory, 0 to a loss, and 0.5 to a draw. Not only is this a natural choice, it also is in line with the requirement of UCB1/UCT for rewards to lie in the $[0, 1]$ range.

In optimization, however, the objective function's range cannot be known in advance, and is normally not determined by the decision maker. Additionally, even for different instances of the same problem, the ranges of observed objective values are not related.

We can address this problem by keeping, at each node, a record of the best (minimum) and worst (maximum) objective values observed in the associated subtree, and using those values to normalize the rewards of their direct children to the interval $[0, 1]$. Given a parent node p and a child node n, the exploitation value of n is given by

$$X^{\diamond}(n) = \begin{cases} 0\,, & \text{if } b_p = w_p \\ \frac{b_n - w_p}{b_p - w_p}\,, & \text{otherwise.} \end{cases} \tag{3.1}$$

where b_p and w_p are, respectively, the best and worst simulation results under p, and b_n is the best simulation result under n.

In this manner, the best possible simulation outcome under node p is given an exploitation value of 1, the worst outcome is given value 0, and all other remaining outcomes are given values in-between. The interpolation of intermediate values is done linearly, but other functions could be used as well.

3.3 Infeasibility

Another problem related to the range of objective values is that of infeasibility. For some optimization problems, the mere task of finding a feasible solution might be very difficult. In such cases, a tree search may have to visit a large number of nodes before finding its first feasible solution.

The problem of finding a feasible solution can be posed as minimizing the degree of constraint violations. If the magnitude of constraint violations can be easily determined, then this information can be leveraged by MCTS in exactly the same way as the values of feasible solutions. However, it cannot be known in advance if it is hard to find feasible solutions for a problem.

We extend the ideas in the previous section and additionally keep track, at each node, of the best/worst (resp. least/most) infeasible solutions found, as well as the number of feasible/infeasible solutions observed in its subtree. Whenever a node n is being evaluated for selection, let b_n represent its best solution value: if b_n corresponds to a feasible solution, then a "raw" exploitation value is given by equation (3.1);

otherwise the raw exploitation value is given by

$$X^{\bullet}(n) = \begin{cases} 0\,, & \text{if } b_p^{\bullet} = w_p^{\bullet} \\ \frac{b_n^{\bullet} - w_p^{\bullet}}{b_p^{\bullet} - w_p^{\bullet}}\,, & \text{otherwise.} \end{cases} \tag{3.2}$$

where $b_p^{\bullet}$ and $w_p^{\bullet}$ correspond to the least and worst infeasible solutions under the parent node p, and $b_n^{\bullet}$ is the amount of constraint violations in the least infeasible solution under node n.

The above raw exploitation functions $X^{\diamond}(n)$ and $X^{\bullet}(n)$ both map to the $[0, 1]$ range. Ideally, we would like to assign higher exploitation values to feasible solutions, and prevent any overlap between the two types of solutions. This can be achieved using the tallies of feasible and infeasible solutions under node p, $t_p^{\diamond}$ and $t_n^{\bullet}$, respectively. The final *joint* exploitation function is given by

$$X^{\star}(n) = \begin{cases} \frac{t_p^{\bullet}}{t_p^{\diamond} + t_p^{\bullet} + 1} . X^{\bullet}(n)\,, & \text{if } b_n \text{ is infeasible} \\ \\ \frac{t_p^{\bullet}}{t_p^{\diamond} + t_p^{\bullet}} + \frac{t_p^{\diamond}}{t_p^{\diamond} + t_p^{\bullet}} . X^{\diamond}(n)\,, & \text{otherwise.} \end{cases} \tag{3.3}$$

This function assigns values in the interval $[\frac{t_p^{\bullet}}{t_p^{\diamond} + t_p^{\bullet}}, 1]$ to nodes whose best solution is feasible, and values in $[0, \frac{t_p^{\bullet}}{t_p^{\diamond} + t_p^{\bullet} + 1}]$ to nodes whose best solution is infeasible. Hence, any feasible value is always preferred, from an exploitation standpoint, to any infeasible value.

3.4 Branch-and-bound

For problems where useful bounds are available, we wish to make use of those bounds to reduce the number of nodes that have to be explored. As branch-and-bound is a fairly simple technique, its integration into the algorithm is straightforward. The addition of pruning to MCTS is done on two occasions:

- Every time a new node is to be added to the tree (expansion step), we first check if the node's bound is better than the incumbent solution. If not, then

the node is immediately discarded.

- Whenever the incumbent solution is updated, the entire tree is traversed top-down, comparing node bounds to the new best solution. As above, nodes whose bounds are not better than the incumbent solution are discarded. In this case, entire subtrees can be eliminated (*e.g.*, if a node close to the root is pruned), depending on the tightness of the bound and the quality of the incumbent.

  The removal of a node from the tree implies an update to its ancestors' best solutions. If node n is being pruned, and the best solution of an ancestor node a was obtained from n, then b_a must be updated using

  $$b_a \leftarrow \underset{c \in C_a \setminus \{n\}}{\arg\min}\, b_c \,,$$

  where C_a is the set of children of node a. This update must also be performed in other situations where a node is removed from the tree, *e.g.*, complete exploration of its subtree.

It should be noted that pruning introduces an overhead for the computation of bounds. However, this investment often pays off (in some cases, greatly) due to the reduction of tree size, allowing the remaining computational resources to be better focused. The reduced tree size can also help alleviate (but not eliminate) the space requirements of MCTS. To allow for a quick evaluation of the benefits of pruning in each application, our implementation provides a switch to turn pruning on or off.

3.5 Lazy expansion

For certain problems, the mere task of enumerating all actions available from a given state can be computationally expensive, either because there is a large number of actions available, or because computing this set involves complex calculations. Lazy expansion addresses this problem by computing actions on an as-needed manner.

Our implementation makes effective use of Python generators[1] in the enumeration of available actions from a node, during the expansion step of MCTS. Whenever a node is created, a generator of available actions is initialized, but only the first action is actually generated. The generator is paused, and resumed once the node is selected for expansion again, to produce the next available action.

This use of generators prevents wasted time producing actions that may not get to be explored anytime soon. For partial searches (by far the most common case), this guarantees that only actions that are explored are actually generated.

3.6 Selection interleaving

This enhancement is a slight variation on the ideas of first play urgency (FPU, section 2.2.6). It is focused on the same issue, which consists in the phenomenon of reduced exploitation observed in deeper levels of high-branching-factor trees (the common case in CO).

Our proposal is to introduce a parameter $\theta \in \mathbb{N}$ that controls, for nodes that still have unexplored children, the frequency of selection of already-visited children. In other words, given a node n with unexplored children, selection from n will pick an already-visited child once every θ times; the remaining $\theta - 1$ times, selection will stop at node n, and therefore one of its unexplored children gets generated and added to the tree.

Although the intended effect is somewhat similar to FPU, this implementation seems easier to grasp and tune, has a more predictable effect, and can be entirely disabled by setting $\theta = \infty$. Tuning of the parameter θ is also much less dependent on the application—for example a value of $\theta = 5$ seems generally reasonable.

Algorithm 3.1 below illustrates how interleaved selection can be implemented, requiring only a slight modification of algorithm 2.3.

[1] Generators can be very succinctly summarized as "resumable functions", in that they encapsulate the execution state of a function which can be paused (optionally producing a value) and resumed multiple times, each time continuing execution from the location where it last paused.

```
1 function InterleavedSelection(n)
2     C ← set of children of n
3     if n has unexplored actions and |C| mod θ ≠ 0 then
4         return n
5     else
6         n′ ← arg max_{c∈C} U(c)
7         return InterleavedSelection(n′)
```

Algorithm 3.1: Interleaved selection.

Chapter 4

Number Partitioning Problem

4.1 Introduction

The number partitioning problem (NPP) is a classical combinatorial optimization problem, with applications in public key cryptography and task scheduling. Given a set (or possibly a multiset) of N positive integers $\mathcal{A} = \{a_1, a_2, \ldots, a_N\}$, find a partition $\mathcal{P} \subseteq \{1, \ldots, N\}$ that minimizes the discrepancy

$$E(\mathcal{P}) = \left| \sum_{i \in \mathcal{P}} a_i - \sum_{i \notin \mathcal{P}} a_i \right| .$$

Partitions such that $E = 0$ or $E = 1$ are called perfect partitions.

A pseudo-polynomial time algorithm for the NPP is presented in [GJ79] for the case where all a_j are positive integers bounded by a constant A; for the general case of exponentially large input numbers (or exponentially high precision, if the a_j's are real numbers) the problem is NP-hard. If the numbers a_j are independently and identically distributed random numbers bounded by $A = 2^{\kappa N}$, the solution time abruptly rises at a particular value $\kappa = \kappa_c$; this is due to the high probability of having perfect partitions for $\kappa < \kappa_c$, and this probability is close to 0 for $\kappa > \kappa_c$ (see [Mer06, Mer98] for more details).

A direct application of the NPP occurs in load-balancing of two identical machines. The common two-way NPP, as well as a generalization to an arbitrary number of subsets (equivalently, machines) are tackled in [AGKR05] by recasting the problem as an unconstrained quadratic binary program (UQP); the UQP is then solved using a tabu search algorithm. Another application arises in high-performance multi-disk database systems. To promote parallelization of I/O and minimize query response times in such systems, data that is likely to be accessed by same queries is distributed across K disks—a process called *declustering*. This problem, equivalent to a multi-way NPP, is tackled in [KA05] using a two-phase approach: the first phase consists in recursive bipartitioning to obtain an initial K-way partition; in the second phase, the initial partition is improved through a refinement heuristic.

The best polynomial time heuristic known for the NPP is the differencing method of Karmarkar and Karp [KK82] (the KK heuristic). It consists of successively replacing the two largest numbers by the absolute value of their difference and placing those items in separate subsets, but without actually fixing the subset into which either number will go (see algorithm 4.1). When applied to the set $\mathcal{A} = \{8, 7, 6, 5, 4\}$, the KK heuristic leads to the partitions $\{8, 6\}$ and $\{7, 5, 4\}$ with discrepancy 2 (figure 4.2).

```
1 function KK (A ⊂ ℤ⁺)
2     ∀a_i ∈ A, create a vertex i with label l_i ← a_i
3     E ← {}
4     while there is more than one labeled vertex do
5         u, v ← vertices with the two largest labels
6         E ← E ∪ {{u, v}}
7         set label l_u ← l_u − l_v
8         remove label l_v from vertex v
9     return discrepancy (i.e., the only remaining label)
```

Algorithm 4.1: The Karmarkar-Karp heuristic.

Extensions of the KK heuristic for a complete search have been proposed in [Kor98, PK10]. In each step of the KK heuristic the two largest numbers are replaced by their difference; for a complete search, the alternative of replacing them by their sum—corresponding to placing the two numbers in the same subset—must also be

considered. For the previous example, the complete search tree is represented in figure 4.1.

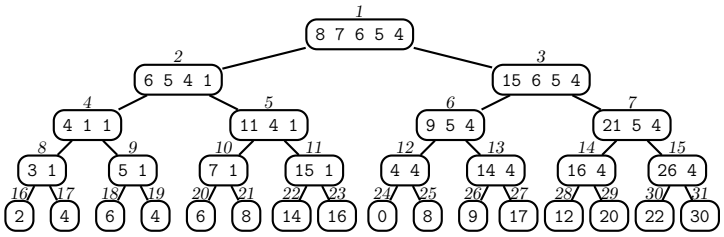

Figure 4.1: Search tree for the complete differencing method with the set $\mathcal{A} = \{8, 7, 6, 5, 4\}$.

Figures 4.2 and 4.3 show graphs representing solutions to the above example instance: straight edges connect differencing vertices (corresponding to the decision taken by the KK heuristic), that will be in different partitions; curly edges connect addition vertices (corresponding to the opposite decision to the KK heuristic), that will be in the same partition. Figure 4.2 represents the solution obtained by the KK heuristic, whereas figure 4.3 represents the optimal solution obtained by complete search. The optimal solution is the partition $\{8, 7\}$ and $\{6, 5, 4\}$, which is a perfect partition.

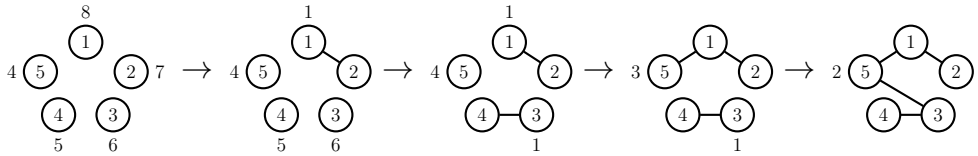

Figure 4.2: Graph created with the KK heuristics, corresponding to the path $1 \to 2 \to 4 \to 8 \to 16$ in figure 4.1.

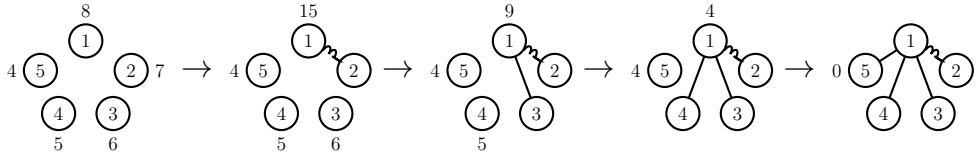

Figure 4.3: Graph created while applying complete search: steps followed for creating the optimal partition, *i.e.*, the path $1 \to 3 \to 6 \to 12 \to 24$ in figure 4.1.

In the worst case, the complete differencing method has exponential time complexity. Parts of the search tree may be pruned by observing that:

- the KK heuristic is exact for partitioning 4 or fewer numbers;
- the algorithm can be stopped when a perfect partition is found;
- when the difference between the largest remaining number and the sum of other remaining numbers is greater than 1, the best possible solution is to place the largest number in one set and all the other numbers in the other set.

Taking the above remarks into account, algorithm 4.2 introduces depth-first search for the complete KK.

```
1  function dfs (A ⊂ ℤ⁺)
2      if |A| ≤ 4 then return KK (A)
3      u, v ← largest and second-largest values in A
4      ū ← sum(A) − u
5      d ← u − ū
6      if |d| ≤ 1 then return |d|
7      if d > 1 then return d
8      A ← A \ {u, v}
9      ℓ ← dfs (A ∪ {u − v})
10     if ℓ ≤ 1 then return ℓ
11     r ← dfs (A ∪ {u + v})
12     return min(ℓ, r)
```

Algorithm 4.2: dfs($\mathcal{A}$) — depth-first search for the number partitioning problem.

4.2 Implementation details

We propose the application of MCTS to this problem, using the KK heuristic as the construction method in the simulation step. Since this heuristic is deterministic, running it from the differencing child (which coincides with the heuristic's choice) would lead to the exact same solution as that obtained for the parent node; thus, we can safely reuse the parent's solution, and therefore only one new construction is required for the two children of each node. For this reason, we use a full expansion strategy in the expansion step of MCTS, meaning that both children of each node are immediately generated once the node is selected.

Another advantage of using the KK heuristic is that repeated solutions are avoided altogether; such would not be the case with random or semi-greedy simulations. Finally, the KK heuristic allows the above pruning rules to be used, reducing the size of the search space without sacrificing completeness.

The careful reader may notice that this implementation represents a rather unorthodox use of MCTS, since Monte Carlo simulations are not actually being employed. However, our intention here is to demonstrate that UCT's selection utility function can effectively guide the search in an optimization context, even when estimation of node values is based on a deterministic algorithm.

4.3 Computational results

Table 4.1 presents results obtained with the KK heuristic and with time-limited DFS and MCTS, run on 10 hard instances from [PK10].

Instance	KK	DFS (60s)	MCTS (60s)	DFS (600s)	MCTS (600s)
hard0100	73.37	56.12	54.56	53.36	51.20
hard0200	171.81	151.91	150.88	151.35	149.79
hard0300	265.77	247.07	248.82	247.07	247.18
hard0400	366.18	343.36	347.55	339.07	344.84
hard0500	461.60	443.54	441.42	438.79	437.22
hard0600	557.09	540.22	542.59	539.13	540.66
hard0700	659.50	640.48	641.78	637.70	633.84
hard0800	751.27	737.65	738.49	731.98	735.67
hard0900	853.36	834.95	837.93	834.95	833.87
hard1000	952.47	932.55	938.05	932.55	930.23

Table 4.1: Results obtained for number partitioning with the KK heuristic, DFS, and MCTS on hard, large instances from [PK10]. DFS and MCTS results are reported at 60 seconds (center columns) and 600 seconds (right-hand side columns). For MCTS, the average best solution of 10 independent runs is shown. Values reported are $\log_2(n+1)$, where n is the discrepancy obtained (which for these instances is a very large integer).

As expected, both tree search variants provide very considerable improvements over

KK. It is well known that, for difficult instances, the optimum for NPP is very hard to find; DFS, being able to explore a much larger portion of the tree—hundreds to thousands of millions of nodes per hour, for these instances—is very difficult to beat. Also, as previously mentioned, the performance of DFS is highly dependent on the construction heuristic used, which in this case provides solutions of very good quality to begin with.

Although there is no clear indication of a winner from the results obtained, the observation of mixed results between DFS and MCTS is nonetheless impressive: MCTS was highly competitive despite exploring less than a thousandth of the nodes explored by DFS. This suggests that MCTS can effectively guide the search towards promising regions. In our view, these are very encouraging results.

These results indicate that MCTS's rate of convergence should improve over time. While DFS gains practically nothing as the search progresses, MCTS constantly accumulates knowledge of the search landscape, which should theoretically help convergence toward the optimum.

Chapter 5

Stacking Problem

5.1 Introduction

The stacking problem (SP) consists of a series of placement decisions for a set of items with known dates for entry into and exit from a storage area (*e.g.*, a warehouse), denoted by *release* and *due* dates, respectively. Items are placed in vacant positions, or on top of other items forming stacks, *i.e.*, last-in first-out queues (figure 5.1). At any given time, only the top item of each stack can be taken, so in order to take an item that is not at the top of a stack, it is necessary to first relocate all items above it onto other stacks. The objective is to store and then deliver all items with a minimum number of relocations.

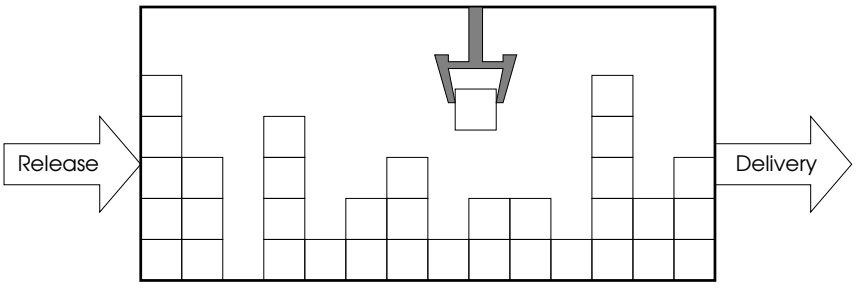

Figure 5.1: A warehouse where items are stored in stacks, using a stacking crane that can only handle one item at a time.

In the variant tackled in this work, we assume that there is no height limit on stacks (uncapacitated SP), item movements occur instantaneously, and release and delivery dates must be strictly respected, *i.e.*, releases and deliveries cannot be anticipated or delayed[1].

Stacking problems have evident practical importance in container port operations [DVA07, Har04] and ship stowage planning [APS00, APSW98]. Item stacking is also important in the steel industry [RP13], as finished products are stored in warehouses under very similar conditions to the SP.

The complexity class of the SP is studied in [RP13], where it is shown that the zero-relocation SP (decision problem) is NP-complete for any fixed number of stacks $W \geq 4$, by a polynomial reduction to the problem of coloring circle graphs. From this, it follows that the general r-relocation SP is also NP-complete, and the optimization version of the SP is NP-hard.

Problems with similar characteristics have been described and studied in the literature. One such problem is the Block Relocation Problem (BRP), where the initial state of the stacks is given as input data along with a (partial) order of retrieval of the items. The objective is to find the shortest sequence of movements such that items are retrieved in the given order. In [RP13], it is observed that the BRP is in fact a sub-problem of the SP, as it only considers the retrieval of items and does not take into account new arrivals. In [KH06], a branch-and-bound algorithm for the BRP, as well as a simple heuristic for real-time applications are proposed. The heuristic rule is based on an estimate of the number of additional relocations for each stack. In [CVS11], an algorithm for the BRP based on the corridor method is presented. The corridor method combines mathematical programming techniques with heuristics. The main idea is to exactly solve sub-problems where some variables are fixed, creating a "corridor-like" region where an item is allowed to go.

Another related problem is the container ship stowage problem (CSSP). The objective in the CSSP is to minimize the number of *shifts*, which consist in the temporary removal of containers from a ship's stacking bay when other containers below must

[1] Due to the assumption of instantaneous movements, item release and delivery dates can always be respected exactly.

be delivered at a port. Shifted containers are then reloaded in arbitrary order. An integer programming model for the CSSP is presented in [APSW98]; however, the model's usability is very limited due to the large number of binary variables and constraints. For this reason, the authors also propose a heuristic procedure.

For tackling the SP itself, in [RP13], a discrete-event simulation model of the warehouse is used, and construction of solutions is based on a semi-greedy heuristic. The heuristic is invoked when deciding the placement of an item during a release or *reshuffle*. Reshuffling is defined as the relocation of items that is necessary to reach an item deeper in the stack. For simplicity, the voluntary relocation of items in-between releases or deliveries—called *remarshalling*—is not considered during a simulation. Note, however, that this may potentially leave the optimal solution(s) out of the search space, therefore losing the guarantee of optimality even for a complete search.

5.2 Multiple simulation

A simulation consists in traversing a schedule of events (*i.e.*, , releases and deliveries) in chronological order and processing each event appropriately. When multiple events have the same date, deliveries are processed first, in increasing order of *item depth*, where the depth of an item is defined as the number of items above it in the same stack. Then, any releases are processed in inverse order of due date, that is, items with greater due date are released first. Ties are resolved randomly.

The probabilistic component of the construction heuristic is exploited by repetition of the process using different seeds for the pseudo-random number generator, in a simple method called Multiple Simulation (MS). This simple yet effective idea can also be exploited in Monte Carlo tree search, taking advantage of the tree structure to implicitly force different simulations to be executed.

In order to accelerate MS, a simulation is interrupted as soon as it is known that the number of relocations of the incumbent solution cannot be improved. This is actually a weak form of pruning, as seen in branch-and-bound algorithms. Whenever

a better solution is found, the cutoff value is updated, tightening the upper bound for future simulations.

5.3 Flexibility Optimization heuristic

We now describe the construction heuristic used in multiple simulation, called Flexibility Optimization (FO). FO will be used in the MS method, as well as in the simulation step of MCTS.

First, we define the concept of *stack movement date*: the earliest movement date of stack s is represented as

$$m_s = \min_{i \in I_s} D_i \,,$$

where I_s represents the set of items currently in stack s, and D_i is the due date of item i. When stack s is empty, then $m_s = \infty$ by definition.

A *due date inversion* occurs when an item i is placed (directly or indirectly) above an item j with $D_i > D_j$. In order to deliver j, item i will have to be relocated. Therefore, a future relocation is implicitly created whenever a movement creates an inversion. This fact can be used to obtain a loose lower bound on the number of relocations of a given partial solution, by observing that the final number of relocations will be at least equal to the number of relocations made so far plus the number of due date inversions in all stacks.

The stack movement date m_s can be seen as an indicator of the *flexibility* of stack s for receiving new items without creating inversions. To illustrate this idea, consider an empty stack s, with $m_s = \infty$; this stack has infinite flexibility, since any item can be added to it without creating a new inversion. On the other hand, if $I_s \neq \{\}$, then m_s is finite and only items with due date up to m_s can be added to s without creating an inversion.

When placing an item, the FO heuristic will prefer stacks where loss of flexibility is minimized, whenever this is possible without creating new inversions. If a new inversion is unavoidable, the heuristic places the item in the stack with the largest movement date, in order to postpone the item's future relocation as much as possible.

The heuristic makes use of a function associating to each placement decision $i \to s$ a score $f(i, s)$; this function embeds the above rules, and is defined as

$$f(i,s) = \begin{cases} \frac{1}{1+D_i-m_s} - 1 & \text{if } D_i > m_s, \\ \\ \frac{1}{1-D_i+m_s} & \text{otherwise.} \end{cases} \tag{5.1}$$

The graph for this function is shown in figure 5.2. Note that the top branch in equation (5.1) represents the creation of a new inversion (since $D_i > m_s$) and therefore corresponds to lower values of $f(i, s)$, in the interval $[-1, 0)$. The bottom branch represents the placement of an item without creating an inversion; any such movement is given a higher value than any inversion-inducing decision, in the interval $[0, 1]$.

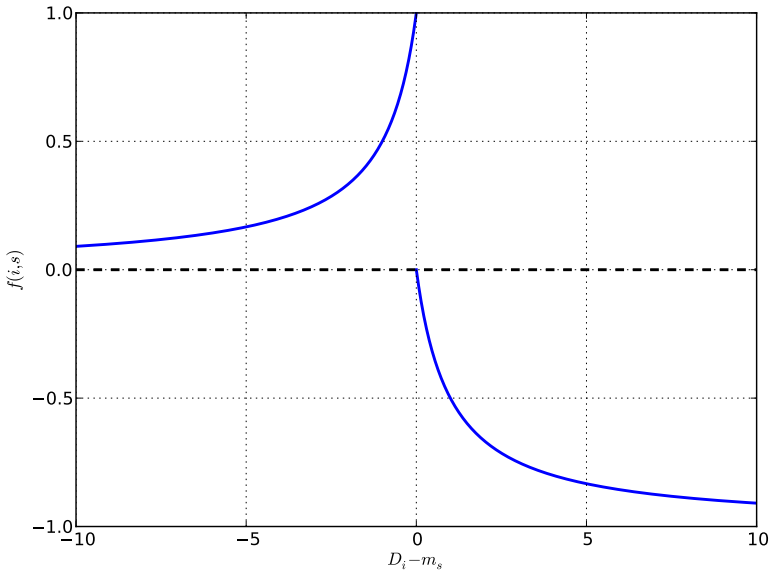

Figure 5.2: Score function $f(i, s)$ used by the FO heuristic.

Given an item i and a set of possible destination stacks T, the FO heuristic constructs a *restricted candidate list* of stacks $\text{RCL} = \{s \in T : f(i, s) \geq f(i, s'), \forall s' \in T\}$, and randomly selects a stack from the RCL as the destination of item i.

5.4 Computational results

A computational experiment was conducted on the 24 benchmark instances of [RP13], comparing MCTS with the MS method. MCTS uses the FO heuristic for construction of solutions in the simulation step. In this case, since the heuristic used is non-deterministic (as opposed to the KK algorithm in the number partitioning problem), a new simulation must be run for each node created in MCTS. As for the expansion strategy used, in this problem we choose single expansion due to the potentially high branching factor.

The two methods were run ten times on each instance, for 600 seconds, using different seeds for the pseudo-random number generator. Table 5.1 presents the average number of relocations of the best solution found after 60 seconds (left-hand side columns), and at the end of the 600-second period (right-hand side columns).

The tree search is naturally expected to perform better, but the reduced computational budget (especially the 60-second limit) presents some difficulties for MCTS. As MCTS uses the results of simulations from each node to estimate its worth, it has an initial warm-up period during which its decisions may be poor due to the lack of available information. As more time is allowed, node evaluation estimates improve and results are expected to be more consistent. The reduced time is an advantage for MS also because of its nonexistent overhead, as opposed to MCTS which must traverse the tree from the root to a non-expanded node at each iteration, create and maintain the tree structure, and backpropagate simulation results. Additionally, the early abortion of simulations used in MS further allows it to run more simulations than MCTS, where this technique cannot be used.

In short, MS is a brute force method that is capable of running many more simulations per second, but relies solely on the variability of the construction heuristic to obtain different/better solutions. In comparison, MCTS has a significant overhead and warm-up period, but incorporates sophisticated rules to explore the search space in a cleverer manner.

The results indicate that, even with 60 seconds, MCTS performs better than MS in most instances; this shows that the search is able to quickly focus on the more

Instance	MS (60s)	MCTS (60s)	MS (600s)	MCTS (600s)
2-A	52872.6	52358.1	52794.3	51854.5
2-B	50820.6	50760.2	50799.1	50727.3
2-C	49450.7	48880.0	49249.2	47514.3
2-D	50227.2	50230.6	50058.9	49090.4
3-A	23148.2	23124.3	22568.4	22502.2
3-B	24608.7	24665.6	24137.5	24201.0
3-C	24627.9	24738.3	23915.3	24036.8
3-D	24130.7	24065.4	23349.3	23510.7
4-A	14439.4	14441.0	14159.9	14132.4
4-B	13355.9	13429.9	13204.6	13150.5
4-C	13981.8	14050.8	13720.9	13854.0
4-D	14796.0	14865.8	14501.4	13975.5
10-A	3524.9	3484.1	3494.1	3289.5
10-B	4031.3	3790.9	4013.4	3751.4
10-C	3644.3	3497.1	3606.1	3421.7
10-D	3416.3	3437.9	3390.3	3410.4
20-A	883.9	887.5	882.6	885.3
20-B	1029.4	950.6	1022.8	842.2
20-C	1098.0	977.1	1098.0	975.7
20-D	1178.0	1159.2	1178.0	1158.9
40-A	79.0	53.7	79.0	53.4
40-B	59.0	30.5	59.0	9.0
40-C	41.0	32.1	41.0	29.3
40-D	154.0	99.7	154.0	94.3

Table 5.1: Results for the stacking problem: average number of relocations over 10 runs with the MS and MCTS algorithms, for large instances from [RP13], with a time limit of 60 seconds (left-hand side columns) and 600 seconds (right-hand side columns).

promising branches. As expected, performance is further improved with the increased time limit of 600 seconds.

Chapter 6

Recursive Circle Packing

6.1 Introduction

The recursive circle packing problem (RCPP) originates from the tube industry, where shipping costs represent an important fraction of the total cost of product delivery [PCT14]. Tubes are produced in a continuous extraction machine and cut to the length of the container inside of which they will be shipped. Before being placed in the container they may be inserted into other, wider tubes—a process called *telescoping*—so that usage of container space is maximized. As all the tubes occupy the full length of the container, maximizing container load is equivalent to maximizing the area filled with circles (or, more precisely, rings/*annuli*) in a cross-section of the container.

This problem is evidently more general than circle packing, which is known to be NP-complete (see, *e.g.*, [LRK79]). We describe a heuristic method for tackling it, which has proven to be able to produce very good solutions for practical purposes.

A non-technical, general overview of circle packing is presented in [Ste03]; for a bibliographic review article see [HM09], which surveys the most relevant literature on efficient models and methods for packing circular items into regions in the Euclidean plane; items and regions considered are either 2- or 3-dimensional. A survey of industrial applications of circle packing and of methods for their solution, both

exact and heuristic, is presented in [CKP08].

In the base RCPP, a number A of tubes is available for packing in a container of width W and height H, in such a way that the value of the packing is maximum. Let $\mathcal{A} = \{1, \ldots, A\}$ be the index set of the tubes; each tube $i \in \mathcal{A}$ is characterized by an external radius r_i^{ext} and an internal radius r_i^{int}, and can either be included in or left out of the packing. A formulation in mixed integer non-linear programming, provided in [PCT14], considers:

- binary variables w_i for all $i \in \mathcal{A}$, where $w_i = 1$ if tube i is placed directly inside the container, $w_i = 0$ otherwise;
- binary variables u_{ki} for $k, i \in \mathcal{A}$ such that $r_k^{\mathsf{int}} \geq r_i^{\mathsf{ext}}$, where $u_{ki} = 1$ if tube i is placed directly inside tube k, $u_{ki} = 0$ otherwise (only required if $r_k^{\mathsf{int}} \geq r_i^{\mathsf{ext}}$; other pairs (k, i) are not excluded for facilitating notation);
- position variables (x_i, y_i) of the center of tube i, for all $i \in \mathcal{A}$ (only relevant if i is packed).

Each loaded tube is placed within the bounds of a container, which we assume to be a rectangle with vertices $(0, 0)$, $(W, 0)$, $(0, H)$ and (W, H). The constraints are the following:

- inserted tubes must be completely inside the container;
- loaded tubes may be placed either directly in the container or inside other tubes;
- for each pair of tubes (i, j) directly placed in the container, the distance between them must be greater than or equal to the sum of their external radii;
- the above constraint is likewise applied for each pair of tubes (i, j) directly placed inside the same tube k;
- if tube i is placed directly in tube k, their centers must be close enough for i to remain completely inside k.

The objective of this problem is to maximize the value of the packing, *i.e.*, the sum

of a user-defined value v_i for loaded tubes:

$$\text{maximize } V = \sum_{i \in \mathcal{A}} v_i \left(w_i + \sum_{k \in \mathcal{A}} u_{ki} \right). \tag{6.1}$$

6.2 Heuristic construction

We now describe a heuristic method for quickly constructing a solution to the RCPP. The method begins with an empty container and iteratively inserts new tubes either directly into the container or into a previously packed tube. An auxiliary set $\mathcal{O}$ of *open objects* is used, which initially has the container as its only element. An object (a tube or the container) is said to be open while it is possible to insert at least one of the remaining tubes into it. Whenever a new tube is packed, it is added to $\mathcal{O}$; and when it is found that no tubes can be inserted into an object, it is removed from $\mathcal{O}$. The algorithm packs a new tube per iteration until either all available tubes have been packed or $\mathcal{O}$ becomes empty.

In order to prioritize telescoping—which seems intuitively advantageous because value is gained without further occupying the container—we choose, if possible, to insert the next tube into the open object $o \in \mathcal{O}$ with minimum free space. It is then checked if at least one tube can be inserted into o: if it is possible, we move on to the next step in the algorithm; otherwise, o is removed from $\mathcal{O}$ and the object having the next minimum free space is checked. If during this selection $\mathcal{O}$ becomes empty, the algorithm stops and the current solution is returned.

After selecting the object o into which a tube will be inserted, the actual tube to be inserted is selected. In this step, the algorithm greedily chooses the tube t which has the maximum estimated *value-to-area ratio*. This ratio is an estimate of the total value of a tube and all tubes that can potentially be telescoped into it, divided by its area. It is approximated in a manner similar to the greedy heuristic for the knapsack problem, which is based on the value-to-weight ratio of items.

Finally, from the set of positions of tube t in the open object o, a position p with minimum ordinate is chosen; for tie-breaking, the position with smallest abscissa is

selected. An iteration is concluded by inserting tube t into object o at position p, and updating $\mathcal{O}$ to include t.

6.2.1 Discretization of tube positions

Since the position variables (x_i and y_i) in the mathematical model are real variables, the set of positions for a tube is often infinite. In the computation of candidate positions for tube t, we reduce this possibly infinite set to a finite set through a number of simple rules. When inserting a new tube t directly into the container (figure 6.1), the candidate positions considered are:

- the two positions placing t at the bottom corners of the container;
- for each tube u already packed directly into the container, include all positions where t is tangent to u and to any wall of the container;
- for each pair of tubes (v, w) already packed directly into the container, include all positions where t is tangent to both v and w.

Similarly, when t is being inserted into a wider, previously packed tube t' (figure 6.2), the set of candidate positions includes:

- the position placing t at the bottom center of t';
- for each tube u packed directly into t', include all positions that are tangent to both t' (from the inside) and u (from the outside);
- for each pair of tubes (v, w) already packed directly into t', include all positions that are tangent to both v and w;

After the set of candidate positions is computed, positions violating any constraint (*e.g.*, positions where t overlaps with a packed tube) are discarded. It must be noted that with this simplification we are excluding the majority of the original problem's search space, so the property of proven optimality is lost even when this restricted search space is fully explored. Figure 6.3 shows an example of an optimal solution which cannot be generated by the described method; whichever the first tube is, it is not at a corner of the container.

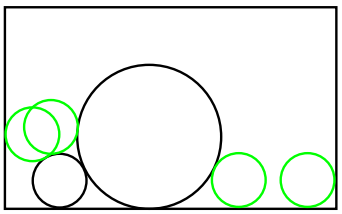

Figure 6.1: Circle packing inside a rectangle: positioning possibilities given previously placed, fixed circles (in black).

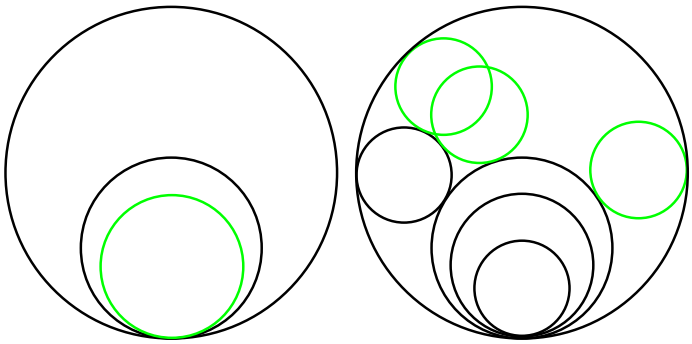

Figure 6.2: Circle packing inside another circle (telescoping): positioning possibilities given previously placed, fixed circles (in black).

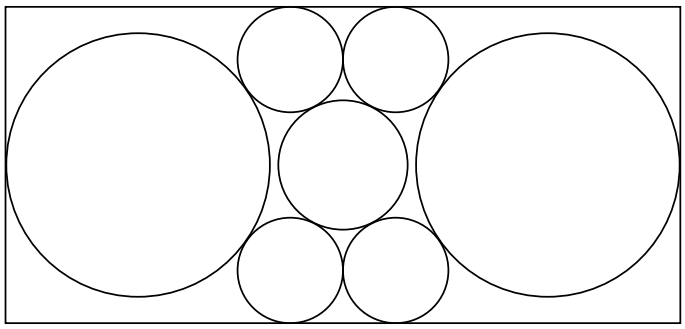

Figure 6.3: An optimal solution which is not contained in the restricted search space.

6.2.2 Semi-greedy construction

The above construction method is purely greedy, leading to a single solution if the same data is provided multiple times. The method is converted into a semi-greedy algorithm by introducing a probabilistic component into one of the choices; in our implementation, the position into which a new tube is inserted is randomly selected, giving higher probability to positions closer to the greedy decision. The full construction method is summarized in algorithm 6.1.

```
1  function SG(container C, set of available tubes 𝒜)
2      𝒮 ← {} (set of (tube, position) pairs)
3      𝒪 ← {C} (set of open objects)
4      while 𝒪 ≠ {} and 𝒜 ≠ {} do
5          o ← element of 𝒪 with minimum unused area
6          foreach t ∈ 𝒜 (from largest to smallest value-to-area ratio) do
7              𝒫 ← positions for t inside o
8              if 𝒫 ≠ {} then
9                  choose position p ∈ 𝒫 (semi-greedy choice)
10                 𝒮 ← 𝒮 ∪ {(t, p)}
11                 𝒪 ← 𝒪 ∪ {t}
12                 𝒜 ← 𝒜 \ {t}
13                 break
14         if could not place any tube inside o then
15             𝒪 ← 𝒪 \ {o}
16     return 𝒮
```

Algorithm 6.1: Semi-greedy heuristic construction for recursive tube packing.

This semi-greedy variant is used in both algorithms compared in the computational experiment. The multiple simulation (MS) algorithm consists in repeating constructions with different pseudo-random number generator seeds, whereas MCTS uses it for the simulation step. As in the stacking problem, MCTS uses a single-expansion strategy.

6.3 Computational results

The computational experiment included 6 instances of the RCPP, adapted from [PCT14]. The two algorithms were run 10 times on each instance, with a time limit of 600 seconds. table 6.1 reports the average total value of the best solution found after 60 seconds, and at the end of the full 600-second period.

Instance	MS (60s)	MCTS (60s)	MS (600s)	MCTS (600s)
large03	3660023.3	3660021.8	3660024.2	3660023.5
large05	4140042.1	4140044.0	4140043.0	4140047.1
large16	30311008.0	30625928.6	30311016.6	31192827.3
small03	938000.0	950000.0	940000.0	957000.0
small05	1090000.0	1120000.0	1090000.0	1120000.0
small16	10438035.5	10388039.2	10450036.6	10534032.5

Table 6.1: Average value packed in a container, for 10 independent observations, for the RCPP with multiple simulation and with Monte Carlo tree search, with a time limit of 60 seconds (left-hand side columns) and 600 seconds (right-hand side columns). Instances adapted from [PCT14].

Although the number of instances is small, the results indicate a superior performance of MCTS after 60 seconds, further widening the gap when the full 600 seconds are allowed. This is an expected consequence of allowing extra time, as more information is gathered and MCTS's estimates of node values are refined.

Chapter 7

Conclusion

Tree search methods are an essential part of combinatorial optimization (CO). As they represent elegant ways to completely enumerate the search space, tree search algorithms are used in general-purpose integer programming and constraint satisfaction solvers providing exact solutions.

In chapter 2 we looked at various tree search strategies, observing their strengths and weaknesses. The main issues detected are: inability to reach leaves (*i.e.*, complete solutions) in reasonable time; stagnation of the search in a limited part of the search space; exponential space requirements; and heavy reliance on heuristics.

Monte Carlo tree search is a fairly recent addition to the tree search family of methods, which has seen extreme growth in research interest due to considerable successes in different fields (most notably, computer Go). The results in this thesis indicate that MCTS can be a powerful tool for *general* combinatorial optimization. The base MCTS algorithm provides a solution to most issues observed in other tree search methods. A variant of MCTS called UCT (upper confidence bounds for trees) addresses one other crucial issue—the *exploitation-exploration dilemma*—in an optimal way.

Chapter 3 builds upon the UCT algorithm and proposes a number of adaptations for CO taking into account differences from game-playing. Worthy of notice are the different set of node statistics (*i.e.*, best *vs* average solution), normalization of objective

function values, dealing with infeasibility, pruning, and selection interleaving.

The remaining chapters (chapters 4 to 6) then proceed to demonstrate that this method can provide competitive results in a variety of problems. The difficulty in solving the target problems normally stems from the size of the search space, the absence of tight bounds and usable mathematical programming formulations, or lack of effective heuristics.

Chapter 4 presents the number partitioning problem, a hard problem with application in *e.g.*, load balancing. MCTS exploits the quality of the well-known Karmarkar-Karp (KK) construction heuristic in its simulation step, and is compared with a KK-based depth-first search (DFS) approach. MCTS obtains encouraging results, that are comparable in quality to DFS despite being able to examine a far smaller number of solutions.

The second application is the stacking problem (chapter 5), common in container port operations and in handling warehouse storage, for which full simulation is necessary to evaluate even the earliest placement decisions. A semi-greedy construction heuristic (FO) is described for this problem. A basic repeated construction method (MS) is compared with MCTS, with both methods utilizing the FO heuristic for construction of solutions. Although the MS method is capable of completing a higher number of constructions, it is generally outperformed by MCTS except for a few instances. This is attributed to the ability of the FO heuristic to produce a wider range of solutions in these instances.

The final case study is the recursive circle packing problem (chapter 6)—a generalization of circle packing in rectangles—where, once more, early decisions during construction may not be accurately assessed before the solution is complete. A construction heuristic is presented that discretizes the set of candidate positions for a tube. Although this simplification entails that the full search space is no longer covered, it allows the problem to be tackled as a combinatorial problem. MCTS is compared with repeated semi-greedy construction, obtaining higher quality solutions overall.

Even in the presence of good heuristics, MCTS is able to achieve high-quality results, outperforming a simpler method of repeated construction and obtaining comparable

results to a low-overhead DFS method in the NPP. Therefore, MCTS shows great promise as a candidate for many other problems. Its implementation requires only a generative model of the problem (usually already available); if specialized knowledge is unavailable, simulations can be made through uniform random action selection. Nonetheless, problem-specific knowledge can be easily integrated, usually within the simulation step, to improve the algorithm's performance and/or shorten its warm-up period.

Future work in this area may include techniques for accelerating the warm-up period of MCTS (*e.g.*, rapid action value estimation), and parallelization of the Python implementation. Deep learning is a recent trend in Artificial Intelligence (AI) research, and its combination with MCTS has already produced impressive results in the field of computer Go. We believe that this and other general techniques from AI can be brought into the OR field and provide significant performance improvements.

We sincerely hope that the studies in this thesis will contribute to the practice of OR by expanding its reach to previously unapproachable problems, and by sparking interest in MCTS from other OR researchers and practitioners.

Appendix A

Simple MCTS: a Python implementation of Monte Carlo tree search for combinatorial optimization

This appendix presents a Python implementation of the UCT algorithm for optimization, including the adaptations described in chapter 3. The source code is available at [Rei16a], and accompanying up-to-date user documentation can be found at [Rei16b].

A.1 Introduction

Monte Carlo tree search (MCTS) is a very suitable algorithm to tackle difficult problems for which there are no known good (value or policy) heuristics. The algorithm builds a search tree asymmetrically, using relatively simple rules to decide what nodes to explore next.

The core concept of MCTS is that of Monte Carlo simulations, which are used to estimate the value of tree nodes. The term "Monte Carlo" is used here to indicate

that some degree of probabilistic behavior is included in the simulation, and that repetition is used to improve the accuracy of the method, as in the traditional meaning of Monte Carlo methods. In an extreme case, actions are randomly selected with uniform probability at each step of the simulation until a leaf is reached – in the context of optimization, this is either a feasible or infeasible solution.

As more time is allotted to the algorithm, a larger part of the search space is covered and the quality of the node value estimates is improved, especially for nodes closer to the root. This focuses the search on the most promising regions of the tree, and in turn leads to better allocation of computational resources. However, the node selection strategy typically incorporates a term to promote exploration of unknown regions. This term is combined with an exploitation-oriented term, which tends to narrow the scope of the search to the areas around known good solutions.

Below is a highly simplified pseudo-code version of the algorithm for maximization problems:

```
def mcts(root):
    # initialization
    best = simulation(root)
    backprogation(best, start_at=root)
    # main loop
    while time_remaining() and not exhausted(root):
        node = selection(start_at=root)
        for child in expansion(node):
            leaf = simulation(start_at=child)
            if value(leaf) > value(best):
                best = leaf
            backpropagation(leaf, start_at=child)
    return best
```

The code shows the four main components of MCTS – selection, expansion, simulation, and backpropagation – which are described in greater detail in the following sections.

A.1.1 Selection

The role of the selection policy is to traverse the tree and pick the node which seems most promising to be expanded in the current iteration. As the notion of "promising" varies according to the current node value estimates and search history, the tree normally grows in an asymmetric manner.

Selection starts with the root as the current node and, at each level, picks the best child of the current node with respect to some measure of node potential. As mentioned above, this is typically achieved through a combination of exploration- and exploitation-oriented terms.

A.1.2 Expansion

Node expansion determines the structure of the tree, *i.e.* how child nodes are obtained from the selected node. The same problem can be seen from multiple perspectives and lead to different tree structures, *e.g.* binary *vs* n-ary trees. This decision can have a great impact on the algorithm's performance, though this is practically impossible to ascertain.

A.1.3 Simulation

The objective of the simulation step is to make a quick dive and attempt to find complete (feasible) solutions. Even if that cannot be achieved, the degree of infeasibility of a solution found during a simulation can be used to guide subsequent iterations of the algorithm.

Ideally, simulations should be computationally cheap in order to afford as many simulations as possible within the allotted computational budget. Expensive simulations would not allow much information to be gathered and value estimates to improve.

A.1.4 Backpropagation

Each node in the tree keeps a list of simple statistics and data that are used to compute the node's value estimate for the selection step. These data can include the number of simulations run under that node's subtree, the best solution found therein, and possibly other information. The backpropagation step consists in updating these statistics for all nodes between the root and the child which originated the simulation. This means that all nodes in the selection path will have their value estimates updated, hopefully increasing their accuracy.

A.2 Using `rr.opt.mcts.simple`

The *rr.opt.mcts.simple* module provides a simple, self-contained[1] implementation of Monte Carlo tree search, and a framework for users to define their own `TreeNode` classes for specific problems.

To use the framework, a user should simply define their own tree node class by subclassing `TreeNode` . The `TreeNode` base class defines some internal attributes that are used to manage parent-child connections and keep track of simulations. Node objects can define their own internal structure freely, with the exception of the names `path` , `parent` , `children` , `sim_count` , `sim_sol` and `sim_best` , as these are used for the aforementioned purposes. While the base `TreeNode` class takes care of general MCTS-related operations, problem-specific logic must be implemented in subclasses by defining a few methods. These methods are documented below.

class `rr.opt.mcts.simple. TreeNode`

Base class for tree nodes. Subclasses should define:

Tree management methods

- *root()*
- *copy()*

[1] The module depends only on the `future` library for cross-version compatibility with Python 2 and 3.

- *branches()*
- *apply()*

MCTS-related methods

- *simulate()*

Branch-and-bound related methods

- *bound()* *[optional]*

classmethod `root ( instance)`

Given a problem instance, create the root node for the associated search tree.

Normally this method should create an empty node using `root = cls()` and then proceed to add the attributes necessary to fully represent a node in the tree.

Parameters `instance` – an object representing an instance of a specific problem. For example, for the knapsack problem, an instance would contain a list of item weights, a list of item values, and the knapsack's capacity. This could be a 3-tuple, a namedtuple, or even an instance of a custom class. The internal structure of the instance object is not dictated by the framework.

Returns the root of the search tree for the argument instance.

Return type *TreeNode*

`copy ( )`

Create a new node representing a copy of the node's state.

This method should create a new "blank" node using `clone = TreeNode.copy(self)` , which takes care of copying generic MCTS node attributes, and should then fill in the domain-specific data by shallow- or deep-copying the custom attributes that were previously defined in *root()* . Note that some attributes should be unique for each node (hence copied deeply), while

others can (and should, if possible) be shared among all nodes. This should be analyzed on a case-by-case basis.

Returns a clone of the current node.

Return type *TreeNode*

`branches ( )`

Generate a collection of branch objects that are available from the current node.

This method should produce a collection (*e.g* list, tuple, set, generator) of branch objects. A branch object is an object (of any type, and with any internal structure) which carries enough information to apply a modification (through *apply()*) to a copy of the current node and obtain one of its child nodes. In some cases, a branch object may be something as simple as a boolean value (see *e.g.* the knapsack example).

Returns collection of branch objects.

`apply ( branch)`

Mutate the node's state by applying a branch (as produced by *branches()*).

The logic in this method is highly dependent on the internal structure of the nodes and branch objects that are returned by *branches()* .

Note: This method should operate in-place on the node. The `expand()` method will take care of creating copies of the current node and calling *apply()* on the copies, passing each branch object returned by *branches()* and thereby generating the list of the current node's children.

Parameters `branch` – an object which should contain enough information to apply a local modification to (a copy of) the

current node, such that the end result represents descending one level in the tree.

`simulate ( )`

Run a simulation from the current node to completion or infeasibility.

This method defines the simulation strategy that is used to obtain node value estimates in MCTS. It should quickly descend the tree until a leaf node (solution or infeasibility) is reached, and return the result encountered.

Smarter simulation strategies incorporate more domain-specific knowledge and normally use more computational resources, but can dramatically improve the performance of the algorithm. However, if the computational cost is too high, MCTS may be unable to gather enough data to improve the accuracy of its node value estimates, and will therefore end up wasting time in uninteresting regions. For best results, a balance between these conflicting goals must be reached.

Returns object containing the objective function value (or an *Infeasible* value) and *optional* solution data.

Return type *Solution*

`bound ( )`

Compute a lower bound on the current subtree's optimal objective value.

A *bound()* method should be defined in subclasses intending to use pruning. By default, pruning will be automatically activated if the root node defines a *bound()* method different from the one defined in the base *TreeNode* class.

Returns a lower bound on the optimal objective function value in the subtree under the current node.

class `rr.opt.mcts.simple. Solution ( value, data=None)`

Base class for solution objects. The `simulate()` method of *TreeNode* objects

should return a *Solution* object. Solutions can have solution data attached, but this is optional. The solution's value, however, is required.

class `rr.opt.mcts.simple.Infeasible(` *infeas=inf* `)`

Infeasible objects can be compared with other objects (such as floats), but always compare as greater (*i.e.* worse in a minimization sense) than those objects. Among infeasible objects, they compare by value, meaning that they can be used to represent different degrees of infeasibility.

To inform the MCTS framework that a solution is infeasible, set an *Infeasible* object as the solution's `value` attribute, like so:

```
class MyNode(mcts.TreeNode):
    # (...)

    def simulate(self):
        # (...)
        infeas = len(self.unassigned_vars)
        if infeas > 0:
            return mcts.Solution(value=mcts.Infeasible(infeas))
        # (...)

    # (...)
```

A.2.1 Defining custom node attributes

Problem-specific node attributes should not be added in the regular Python object initializer method `__init__()`. In fact, the `__init__()` method should preferably not be redefined by subclasses, as nodes are automatically created by the MCTS framework through the *TreeNode.copy()* method. The recommended way to define additional instance attributes is to add them in the *TreeNode.root()* class method, and replicate the node's custom structure appropriately in *TreeNode.copy()*.

```
class FooNode(mcts.TreeNode):
    @classmethod
```

```
def root(cls, instance):
    root = cls()
    root.instance = instance
    root.shared_state = ExampleSharedState()
    root.bar = 42
    root.ham = ["spam"]
    return root

def copy(self):
    clone = mcts.TreeNode.copy(self)
    clone.instance = self.instance
    clone.shared_state = self.shared_state
    clone.bar = self.bar
    clone.ham = list(self.spam)
    return clone
```

A.2.2 Writing a simulate method

The simulation algorithm lies at the heart of MCTS, and its goal is to reach a full solution or infeasibility by quickly diving down the search tree in a possibly (semi-)randomized manner. In the *rr.opt.mcts.simple* framework, the simulation algorithm should be defined as a `simulate()` method within your *TreeNode* subclass. The only requisite on the `simulate()` method is that it must return a *Solution* object.

Solution objects contain a value representing the solution's objective function value for feasible solutions, or its degree of infeasibility (see *Infeasible*) otherwise. Also, the constructor of the *Solution* object can take a `data` argument, which is an object of any type that is meant to represent the actual solution, *i.e.* a complete set of decision variable assignments. This is helpful if something is to be done with the solutions found, after the algorithm has finished running. The *rr.opt.mcts.simple* framework does not use solution data for any purpose, therefore attaching solution data to a *Solution* object is entirely optional. However, the solution value **must**

be present.

A.2.3 Running the algorithm

Once a custom *TreeNode* class has been defined, MCTS can be run as in the following example:

```
import rr.opt.mcts.simple as mcts
import myproblem

instance = myproblem.load("./instance_01.json")
root = myproblem.TreeNode.root(instance)
sols = mcts.run(
    root=root,
    iter_limit=1e10,
    time_limit=3600,
    rng_seed=42,
)
print(sols.best.value)
if sols.best.is_feas:
    print(sols.best.data)
    print(sols.best.is_opt)
```

The parameters and return value of the *rr.opt.mcts.simple.run()* function are documented below:

`rr.opt.mcts.simple.run` (*root*, *time_limit=inf*, *iter_limit=inf*, *pruning=None*, *rng_seed=None*, *rng_state=None*, *log_iter_interval=1000*, *sols=None*)

Monte Carlo Tree Search for **minimization** problems.

Note: Objective functions (and bounds) for maximization problems must be multiplied by -1.

Parameters

- `root` (`TreeNode`) – the root of the search tree.
- `time_limit` (*float*) – maximum CPU time allowed.
- `iter_limit` (*int*) – maximum number of iterations.
- `pruning` (*bool or None*) – make the search use/not use pruning if true/false. If *None* is given (default), auto-detects pruning settings from root node.
- `rng_seed` – an object to pass to *random.seed()*. Does not seed the RNG if no value is given.
- `rng_state` – an RNG state tuple, as obtained from *random.getstate()*. Can be used to set a particular RNG state at the start of the search.
- `log_iter_interval` (*int*) – interval, in number of iterations, between automatic log messages.
- `sols` (*Solutions*) – a Solutions object obtained from a previous run of MCTS. If this argument is provided, a previous search can be resumed from the point where it stopped.

Returns *Solutions* object containing the best solution found by the search, as well as the list of incumbent solutions during the search.

A.2.4 Caveat: solving maximization problems

It should be noted that this simple MCTS implementation can only deal with minimization problems. However, it is easy to work around this limitation and deal with maximization problems by multiplying all objective function and bound values by -1 (see, *e.g.* the Knapsack example).

A.3 Knapsack example

This section explains the example implementation of MCTS for the knapsack problem. This example can be found within the git repository at `src/examples/knapsack.py` . The file begins with some auxiliary function definitions which are used to run some basic tests. We will skip those here and take a deeper look at the `KnapsackTreeNode` implementation, which defines one possible structure and operations on tree nodes for this particular problem.

A.3.1 `root()`

We begin with the `KnapsackTreeNode.root()` class method, where the internal structure of tree nodes is defined:

```
class KnapsackTreeNode(mcts.TreeNode):
    @classmethod
    def root(cls, instance):
        items, capacity = instance
        root = cls()
        root.items_left = list(sorted(items, key=lambda i: i.ratio))
        root.items_packed = []
        root.capacity_required = sum(i.weight for i in items)
        root.capacity_left = capacity
        root.total_value = 0
        root.upper_bound = None
```

Each tree node will have several domain-specific attributes:

`items_left` A list of `Item` objects ordered by value-to-weight ratio. Items will be considered for packing in decreasing order of value-to-weight ratio.

`items_packed` List of items that have been added to the knapsack. Naturally, at the root node this is initialized as an empty list.

`capacity_required` The total weight of items that are still available to pack. We

detect that a leaf node was reached when `capacity_left` is greater than or equal to `capacity_required`.

`capacity_left` The unused capacity in the knapsack. Since at the root we have an empty solution with no items packed, this is initialized as the total knapsack capacity. Whenever an item is packed, this value will be reduced by the item's weight.

`total_value` This is the sum of the values of all packed items. Similar to `capacity_left`, when packing a new item, `total_value` will be increased by the item's value.

`upper_bound` This is a cached result of the best possible value that is obtainable in the node's subtree. This is computed in the `bound()` method on its first call, and subsequent calls will return the value stored in `upper_bound`.

A.3.2 `copy()`

The next method we look at is `copy()`. This method should simply create a copy of a given node:

```
    def copy(self):
        clone = mcts.TreeNode.copy(self)
        clone.items_left = list(self.items_left)
        clone.items_packed = list(self.items_packed)
        clone.capacity_required = self.capacity_required
        clone.capacity_left = self.capacity_left
        clone.total_value = self.total_value
        clone.upper_bound = None
        return clone
```

The only noteworthy aspect in this method is that the attributes `items_left` and `items_packed` are copied using `list()` in order to create new lists with the same elements for the child node. Otherwise, both the parent and child nodes would refer to the same lists, which would lead to incorrect results. More generally, all data that should be local to each node must be deep-copied in some way, *i.e.* a simple

assignment like `clone.x = self.x` does not suffice. In this particular example, the `Item` objects themselves are not copied because they are immutable data which can (and should) be shared by all nodes.

A.3.3 `branches()`

The next method is `branches()` , defining our options to generate the children of a given node. The structure of the tree is defined by this method. In this example, it defines a binary tree, where each internal node has exactly two children – pack or ignore the next item – and leaf nodes have no children.

```
def branches(self):
    return (True, False) if len(self.items_left) > 0 else ()
```

A.3.4 `apply()`

The `apply()` method takes an element from the list of branches returned by the `branches()` method, and modifies the current node in-place. For the knapsack example, a branch object is simply a boolean value indicating whether or not to pack the highest-value-to-weight-ratio available item.

```
def apply(self, pack_item):
    item = self.items_left.pop()
    self.capacity_required -= item.weight
    if pack_item:
        self.items_packed.append(item)
        self.total_value += item.value
        self.capacity_left -= item.weight
        self.items_left = [
            i for i in self.items_left
            if i.weight <= self.capacity_left
        ]
        self.capacity_required = sum(i.weight for i in self.items_left)
```

```
        if self.capacity_required <= self.capacity_left:
            self.total_value += sum(i.value for i in self.items_left)
            self.capacity_left -= sum(i.weight for i in self.items_left)
            self.capacity_required = 0
            self.items_packed.extend(self.items_left)
            self.items_left = []
```

The first half of this method should be fairly straightforward: the highest value-to-weight ratio item is removed from the list of available items and then, if that item is to be packed, the node's attributes are updated appropriately. In the second `if` statement, the node is recognized as a leaf if all remaining items can fit in the available space.

A.3.5 `simulate()`

The simulation strategy is defined by this method. Here, the simulation performs a dive in the tree using a simple uniform random selection of the branch to follow at each step. Then, the objective value of the solution obtained is multiplied by -1 because the *`rr.opt.mcts.simple`* implementation deals with minimization problems only, and the knapsack problem is a maximization problem.

```
    def simulate(self):
        node = self.copy()
        while len(node.items_left) > 0:
            node.apply(random.choice([True, False]))
        return mcts.Solution(
            value=(node.total_value * -1),
            data=node.items_packed,
        )
```

A.3.6 `bound()`

Implementing a `bound()` method allows the MCTS algorithm to use pruning as in traditional branch-and-bound algorithms. The value returned by this method *must* represent a lower bound on the best objective function value obtainable in the current subtree. If this best-case scenario value is not better than the best value found so far in the search, the entire subtree can be safely discarded, as it is guaranteed not to contain any improving solution. Depending on the "tightness" of this bound, this simple pruning idea can lead to a significant reduction of the size of the tree that must be explored, usually resulting in better performance.

In the knapsack example, we compute a simple yet tight bound that is equivalent to the linear relaxation of the mixed-integer programming formulation, *i.e.* we allow the last item packed in the knapsack to be partially packed in order to obtain maximum value from it. Like in the `simulate()` method above, we must multiply the bound by -1.

```
    def bound(self):
        if self.upper_bound is None:
            bound = self.total_value
            capacity = self.capacity_left
            for item in reversed(self.items_left):
                if item.weight <= capacity:
                    bound += item.value
                    capacity -= item.weight
                else:
                    bound += item.value * capacity / item.weight
                    break
            self.upper_bound = bound
        return self.upper_bound * -1
```

References

[ACBF02] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

[AGKR05] Bahram Alidaee, Fred Glover, Gary A Kochenberger, and Cesar Rego. A new modeling and solution approach for the number partitioning problem. *Journal of Applied Mathematics and Decision Sciences*, 9(2):113–121, 2005.

[APS00] Mordecai Avriel, Michal Penn, and Naomi Shpirer. Container ship stowage problem: Complexity and connection to the coloring of circle graphs. *Discrete Applied Mathematics*, 103(1-3):271–279, 2000.

[APSW98] Mordecai Avriel, Michal Penn, Naomi Shpirer, and Smadar Witteboon. Stowage planning for container ships to reduce the number of shifts. *Annals of Operations Research*, 76:55–71, 1998.

[BCL12] Christoph Buchheim, Alberto Caprara, and Andrea Lodi. An effective branch-and-bound algorithm for convex quadratic integer programming. *Mathematical Programming*, 135(1-2):369–395, 2012.

[BLK+13] J Bazin, Hongdong Li, In So Kweon, Cédric Demonceaux, Pascal Vasseur, and Katsushi Ikeuchi. A branch-and-bound approach to correspondence and grouping problems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(7):1565–1576, 2013.

[Bou06] Bruno Bouzy. Associating shallow and selective global tree search with Monte Carlo for 9× 9 Go. In *Computers and Games*, pages 67–80. Springer, 2006.

[BPW+12] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[CKP08] Ignacio Castillo, Frank J. Kampas, and János D. Pintér. Solving circle packing problems by global optimization: Numerical results and industrial applications. *European Journal of Operational Research*, 191(3):786–802, 2008.

[Cou07] Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *Proceedings of the 5th International Conference on Computers and Games*, CG'06, pages 72–83, Berlin, Heidelberg, 2007. Springer-Verlag.

[CVS11] Marco Caserta, Stefan Voß, and Moshe Sniedovich. Applying the corridor method to a blocks relocation problem. *OR Spectrum*, 33:915–929, 2011.

[CWH+08] Guillaume M JB Chaslot, Mark HM Winands, H JAAP VAN DEN HERIK, Jos WHM Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(03):343–357, 2008.

[DGRW12] Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. Exact combinatorial branch-and-bound for graph bisection. In *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, pages 30–44. Society for Industrial and Applied Mathematics, 2012.

[DVA07] Rommert Dekker, Patrick Voogd, and Eelco Asperen. Advanced methods for container stacking. In Kap Hwan Kim and Hans-Otto

Günther, editors, *Container Terminals and Cargo Systems*, pages 131–154. Springer Berlin Heidelberg, 2007.

[Fin07] Hilmar Finnsson. Cadia-player: A general game playing agent, 2007.

[FM14] Matteo Fischetti and Michele Monaci. Exploiting erraticism in search. *Operations Research*, 62(1):114–122, 2014.

[GH92] M L Ginsberg and W D Harvey. Iterative broadening. *Artificial Intelligence*, 55(2-3):367–383, June 1992.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman, New York, 1979.

[GKS+12] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michele Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. The grand challenge of computer Go: Monte Carlo tree search and extensions. *Communications of the ACM*, 55(3):106–113, 2012.

[GS07] Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In *Proceedings of the 24th international conference on Machine learning*, pages 273–280. ACM, 2007.

[GSCK00] Carla P Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1-2):67–100, 2000.

[GW06] Sylvain Gelly and Yizao Wang. Exploration exploitation in go: Uct for monte-carlo go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*, 2006.

[Har04] Sönke Hartmann. A general framework for scheduling equipment and manpower at container terminals. *OR Spectrum*, 26:51–74, 2004.

[HM09] Mhand Hifi and Rym M'Hallah. A literature review on circle and sphere packing problems: Models and methodologies. *Advances in Operations Research*, 2009(4):1–22, 2009.

[HNR68] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[Jui99] Hugues Rene Juille. *Methods for Statistical Inference: Extending the Evolutionary Computation Paradigm*. PhD thesis, Waltham, MA, USA, 1999.

[KA05] Mehmet Koyutürk and Cevdet Aykanat. Iterative-improvement-based declustering heuristics for multi-disk databases. *Information Systems*, 30(1):47–70, 2005.

[KH06] Kap Hwan Kim and Gyo-Pyo Hong. A heuristic rule for relocating blocks. *Computers and Operations Research*, 33:940–954, 2006.

[KK82] N. Karmarkar and R. Karp. The differencing method of set partitioning. Technical Report UCB/CSD 82/113, University of California - Berkeley, Computer Science Division, 1982.

[Kor85] RE Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[Kor98] Richard E. Korf. A complete anytime algorithm for number partitioning. *Artificial Intelligence*, 106(2):181–203, 1998.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer Berlin Heidelberg, 2006.

[LD60] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[LR85] Tze Leung Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6(1):4–22, 1985.

[LRK79] J.K. Lenstra and A.H.G. Rinnooy Kan. Complexity of packing, covering, and partitioning problems. In A. Schrijver, editor, *Packing and Covering*

in Combinatorics, pages 275–291. Mathematisch Centrum, Amsterdam, 1979.

[LW66] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.

[Mer98] Stephan Mertens. Phase transition in the number partitioning problem. *Physical Review Letters*, 81(20):4281–4284, November 1998.

[Mer06] Stephan Mertens. The easiest hard problem: Number partitioning. In A.G. Percus, G. Istrate, and C. Moore, editors, *Computational Complexity and Statistical Physics*, pages 125–139, New York, 2006. Oxford University Press.

[NW10] J Pim AM Nijssen and Mark HM Winands. Enhancements for multi-player monte-carlo tree search. In *International Conference on Computers and Games*, pages 238–249. Springer, 2010.

[NWCL10] CT Ng, J-B Wang, TC Edwin Cheng, and LL Liu. A branch-and-bound algorithm for solving a two-machine flow shop problem with deteriorating jobs. *Computers & Operations Research*, 37(1):83–90, 2010.

[PCT14] João Pedro Pedroso, Sívia Cunha, and João Nuno Tavares. Recursive circle packing problems. *International Transactions in Operational Research*, 2014. (Accepted for publication).

[PK10] João P. Pedroso and Mikio Kubo. Heuristics and exact methods for number partitioning. *European Journal of Operational Research*, 202:73–81, 2010.

[Rei16a] Rui Rei. Simple MCTS—a Python implementation of MCTS for optimization. https://github.com/2xR/rr.opt.mcts.simple, 2016. Source code repository.

[Rei16b] Rui Rei. Simple MCTS—a Python implementation of MCTS for optimization. http://rroptmctssimple.readthedocs.io/en/latest/, 2016. User documentation.

[RP13] Rui Jorge Rei and João Pedro Pedroso. Tree search for the stacking problem. *Annals of Operations Research*, 203(1):371–388, 2013.

[RTC11] Arpad Rimmel, Fabien Teytaud, and Tristan Cazenave. Optimization of the nested Monte-Carlo algorithm on the traveling salesman problem with time windows. pages 501–510, 2011.

[Sch89] Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE transactions on pattern analysis and machine intelligence*, 11(11):1203–1212, 1989.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[SSR12] Ashish Sabharwal, Horst Samulowitz, and Chandra Reddy. Guiding combinatorial optimization with UCT. In *Proceedings of the 9th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2012)*, pages 356–361. Springer, 2012.

[Ste03] Kenneth Stephenson. Circle packing: A mathematical tale. *Notices of the American Mathematical Society*, 50(11):1376–1388, 2003.

[TKY08] Shogo Takeuchi, Tomoyuki Kaneko, and Kazunori Yamaguchi. Evaluation of Monte Carlo tree search and the application to Go. In *2008 IEEE Symposium On Computational Intelligence and Games (CIG)*, pages 191–198. IEEE, 2008.

[WBS08] Mark HM Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte-Carlo tree search solver. In *Computers and Games*, pages 25–36. Springer, 2008.