UNIVERSITY OF CALIFORNIA SAN DIEGO

Exploring Monte Carlo Tree Search for Combinatorial Optimization Problems

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science

in

Computer Science

by

Ye Jin Jeon

Committee in charge:

Professor Sicun Gao, Chair
Professor Chung Kuan Cheng
Professor Jun-kun Wang

2024

The Thesis of Ye Jin Jeon is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2024

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Sicun Gao for his support as the chair of my committee. I would also like to acknowledge Eric Yu and Zhizhen Qin for their support as my research coworkers.

ABSTRACT OF THE THESIS

Exploring Monte Carlo Tree Search for Combinatorial Optimization Problems

by

Ye Jin Jeon

Master of Science in Computer Science

University of California San Diego, 2024

Professor Sicun Gao, Chair

This thesis proposes a general search algorithm design for hard combinatorial optimization problems. Monte Carlo Tree Search(MCTS) method is a heuristic search method that partitions the search space while balancing exploration and exploitation. We develop an MCTS framework to navigate the combinatorial input domain that uses existing mixed integer linear programming solvers as subroutines. We design different ways to partition the input regions with MCTS and analyze their search efficiency based on experimental results with three combinatorial problems. Our experiments show that this framework can outperform other search frameworks and state-of-the-art commercial solvers such as Gurobi by finding more optimized solutions in a shorter wall-clock time.

Introduction

Combinatorial optimization (CO) problems are prevalent in the real world with ongoing research in problems such as employee scheduling, supply chain planning, job machine planning, box packing, and vehicle routing. These problems can typically be formulated as mixed integer linear programs (MILP) which are known to be NP-hard. One type of approach to solve CO is an exact method approach which can guarantee to find the optimal feasible solution. Such approaches include Branch and Bound (BB) and Cutting Plane. Not surprisingly, however, exact methods are often impacted the most by the "NP-hardness" of the MILP problem [9], as they fail to scale in the number of system variables and constraints.

A second type of approach is the heuristic method which attempts to find a high-quality feasible solution with little to no guarantees on its optimality. There are two common types of heuristic methods. The first approach is a learning-based approach to improve inner policies of BB algorithm [5, 7]. These methods takes much engineering effort on interfacing with inner process of solvers. The second approach involve external searching techniques solving sub-problems with MILP solver. Such approaches include Large Neighborhood Search (LNS) [12, 14] which can often deal better with large MILP problems. However, they still need problem-dependent knowledge to define a "neighborhood" and locally search for solutions without exploring other regions of search space to escape converging to local minima.

Since we are aiming to develop a general search algorithm that can be used in various CO problems considering solvers as black-box models, we choose to follow the second heuristic method to devise new searching techniques that are more scalable and efficient than existing methods.

Chapter 1

Combinatorial Optimization

1.1 Preliminaries

While we consider more general formulations of the combinatorial optimization problem, one commonly used form is the mixed integer linear programming (MILP) problem. This problem optimizes a continuous function for a set of variables that can be both continuous or integer. The typical formulation for a MILP problem is

$$\begin{aligned} \min_{x} \quad & c^\top x \\ \text{subject to} \quad & Ax \geq b \\ & c, x \in \mathbb{R}^n \\ & A \in \mathbb{R}^{m \times n} \\ & b \in \mathbb{R}^m \\ & x_j \in \mathbb{Z} \text{ for } j \in J \end{aligned}$$

where $c^\top x$ is the objective function to minimize, $Ax \geq b$ are the inequality constraints, and for all $j \in J$, x_j must be an integer. In general, MILP problems are non-convex and belong in the class of NP-hard problems due to the discrete nature of the integer constraints.

1.2 Branch and Bound for MILP

Branch and Bound (BB) is a standard method for solving MILP problems. The core idea behind it is to split the input domain into smaller search spaces, and recursively "branch" onto each subspace. It first solves the linear relaxed solution of the MILP. The linear relaxation of the MILP removes the integrality constraints $x_j \in Z$ for $j \in J$, thus turning the problem into a convex optimization problem which can be solved in polynomial time. The solution found for the linear relaxation is globally optimal and provides a lower bound to the function value of feasible solutions. If the relaxed solution satisfies all integrality constraints as well, then the solution is feasible and the optimal MILP solution has been found. Otherwise, these approaches will attempt to recover a feasible solution in the neighborhood of the relaxed solution, with some approaches guaranteeing the optimality of such a feasible solution or the non-existence of such a solution. To avoid a pure brute-force search, BB keeps track of a "bound" on the function value of the optimization problem, to prune entire search subspaces if they exceed the bound threshold. By combining this branch and bound method, BB can find a provably optimal solution faster than simply performing a brute force search over the entire input domain.

1.3 Large Neighborhood Search

Large Neighborhood Search(LNS) is an improvement heuristic that stochastically optimizes solutions by iteratively destroying and repairing the current solution. [12] introduces a general LNS framework for mixed integer programming. It starts with an initial feasible solution and improves the solution by optimizing its partial solution using MILP solver. LNS adapts the destroy-and-repair method or fix-and-optimize method to define a notion of a neighborhood, and search the neighborhood for a better solution. First, it decomposes a set of integer variables X in the solution into a disjoint union $X_1 \bigcup X_2 \bigcup \ldots \bigcup X_k$. It views each subset X_i of variables as a local neighborhood for search. In the destroy-and-repair method, it fixes integers in $X \backslash X_i$ with their values in the current solution and reoptimizes for variables in X_i solving sub-MILP with

Algorithm 1. Large neighborhood search

```
function LNS(the degree of decomposition: k , an initial solution: S)
    for step = 1,...,t do
        X_t ← Random Decomposition(k)
        for i = 1,...,k do
            S ← Fix and Optimize(S, X_t)
        end for
    end for
    return S
end function=0

function FIX AND OPTIMIZE(an initial solution: S_X, a decomposition: X = X_1 ⋃ X_2 ... ⋃ X_k)
    for i = 1,...,k do
        S_X ← Fix and Optimize(S_X, X_i)
    end for
    return S_X
end function=0
```

Gurobi.

Forming the decomposition step of destroy-and-repair with the current solution is an important part of LNS heuristics. The degree of decomposition determines the size of the local neighborhood, which can become a problem for the sub-MILP problem. On one hand, if only a small part of the solution is destroyed, then the neighborhood can be too small and the heuristic may result in a redundant solution. On the other hand, if a very large part of the solution is destroyed, then the neighborhood may be too huge and forces a global reoptimization with the solver. Thus, the design of the destroy-and-repair heuristic can have a large impact on the quality of downstream solutions found within the neighborhood of existing solutions.

1.4 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a general search framework that balances exploitation and exploration traversing tree-like combinatorial spaces. Unlike traditional search algorithms that rely upon exhaustive exploration of the entire search area, MCTS specializes in sampling and exploring only promising areas of the search area. MCTS has been commonly

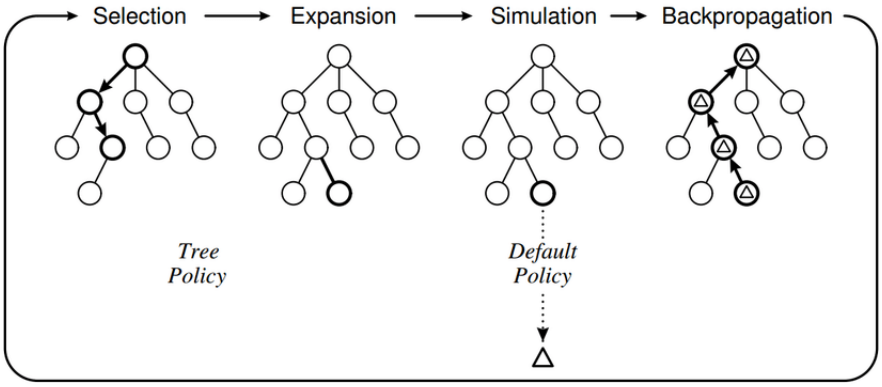

Figure 1.1. Monte Carlo Tree Search [13]

applied to combinatorial games and even some real-time video games and has achieved safe performance. Recently, it is combined with aritificial intelligence(AI) taking advantage of heuristics to make it more efficient and improve the convergence such AlphaGo Zero by Google DeepMind [11]. Driven by successes in games, MCTS has been increasingly often applied in domains outside the game AI such as planning [10], scheduling [3], control and combinatorial optimization [4, 8].

There are typically four MCTS stages: Selection, Expansion, Simulation, and Backpropagation. In Selection, the MCTS algorithm recursively traverses on child nodes with the highest confidence value until the current node is either unexpanded or terminal. If the selected node is unexpanded or not terminal, then it will be expanded during the expansion stage and added as part of the MCTS tree. In the simulation stage, a rollout occurs which can take on many different interpretations based on the MCTS application. For example, in blackbox optimization, a rollout would be a function call. In reinforcement learning, it would be simulating a trajectory starting from the selected node state. Finally, in the back propagation stage, the sampled value from the rollout is propagated from the selected node back up to the root. MCTS repeats this process until it reaches some termination condition.

Chapter 2

MCTS for Combinatorial Optimization

We propose a new MCTS approach for combinatorial optimization that complements the limitations of existing methods such as exhaustive search in BB and small search area in LNS. Start from an initial solution, the goal is to find a globally optimal solution by efficiently exploiting and exploring promising regions of the input domain. To exploit, the algorithm partitions the search space by fixing a variable's value at each level, and vertically expanding the tree while preserving the assignments of previous levels. To explore, the algorithm keeps track of less frequently visited parts of the input domain and expands intermediate nodes in the MCTS tree. Figure 2.1 compares our approach with BB and LNS through a visualization.

We note that the selection of variables to branch on, and the values we assign to them, are critical for an efficient search. We construct a problem-driven method to choose variables and their values for a fix-and-optimize method. First, our algorithm selects variables to fix based on each of their gradients with respect to the objective function. Second, depending on whether the variable is discrete or continuous, we branch using one of two methods, Point-MCTS and Interval-MCTS, which we describe in greater detail below.

2.1 Overall algorithm

The algorithm starts by constructing a root node with an initial solution provided by the solver. In each iteration of the algorithm, we perform four operations sequentially. First, we

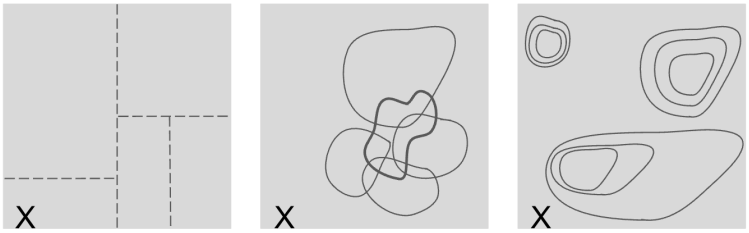

Figure 2.1. Searching pattern in input domain X. BB(left), LNS(mid), and our approach(right)

perform branch selection starting from the root node, and then expands a new node from the leaf node on the selected branch. After every expansion of new node, it performs solving sub-problem with a MILP solver in the simulation stage and back-propagates the simulation result to the root node to keep track of how many times each node has visited and the best objective value found in this branch. The details of the algorithm are explained in the following sections.

Algorithm 2. Monte Carlo Tree Search

```
0: function MCTS(objective: f, domain: Ω)
1: for step = 1,...,t do
       n ← Select(n_0)
       reward ← Simulation(n)
       Backpropagation(n, reward)
2: end for
3: return y_0
```

2.2 Node Configuration

A node consists of its state, untried actions, value, the number of visits, best value, and solution. The state consists of a decision variable and the assigned value to the variable determined in the expansion stage. The untried action is the set of possible values of the variable that has not been explored. The value and solution of the node are assigned according to the result of sub-routine MILP and the number of visits is increased by one every time the node is visited.

Table 2.1. Node configuration

Attributes	Description
State	a list of (a decision variable, its value)
Untried actions	a list of untried values of a variable in its child nodes
Solution	the feasible solution found by sub-MILP solver
Value	a the objective value of the solution
Best value	the best objective value found along the path where node is in
Parent	the immediate parent of the node
Children	the list of its child nodes
Visit	the number of visits

2.3 Selection

MCTS selects a leaf node to expand a new child node determining the path from the root to the leaf. When selecting nodes at every level, there are two choices we can do. If the current node has less child nodes than the possible maximum number of child nodes *MC* and untried decision variables, it expands a new child node from the node. Otherwise, it keeps traversing the tree choosing best child node among its child nodes using UCT v. The UCT is computed by:

$$v(n_i) = R_i/N_i + C \cdot \sqrt{2 \cdot \log(N_b/N_i)} \qquad \text{(Eq.1)}$$

in which R_i is the values in n_i; N_i and N_b denote the number of visits on n_i and its parent node n_b; C is a constant to balance between exploitation and exploration. At each branch node n_b, its immediate children who has highest v value is selected. When selecting nodes at every level, there are two choices we can do. *MC* is a hyperparameter we can set that indicates the maximum number of child node each node can have.

2.4 Expansion

After selecting a node n_i, it chooses a decision variable x_{action} among the untried variables of the selected node n_i that has highest gradient value. The gradients are determined by a coeffi-

Algorithm 3. Selection and Best Child

```
0: function SELECT(node: n_i)
0:     while n_i is not terminal do
0:         if n_i has untried variables and |child| < MC then
0:             return Expand(n_i)
0:         else
0:             n_i ← Best Child(n_i)
1: return n_i

0: function BEST CHILD(node: n_i)
0:     for child node n_bi do
0:         Compute v(n_bi) by Eq.1
0:     end for
0:     b̂ ← argmax_iv(n_bi)
0:     n_b ← n_bb̂
0:     return n_b
```

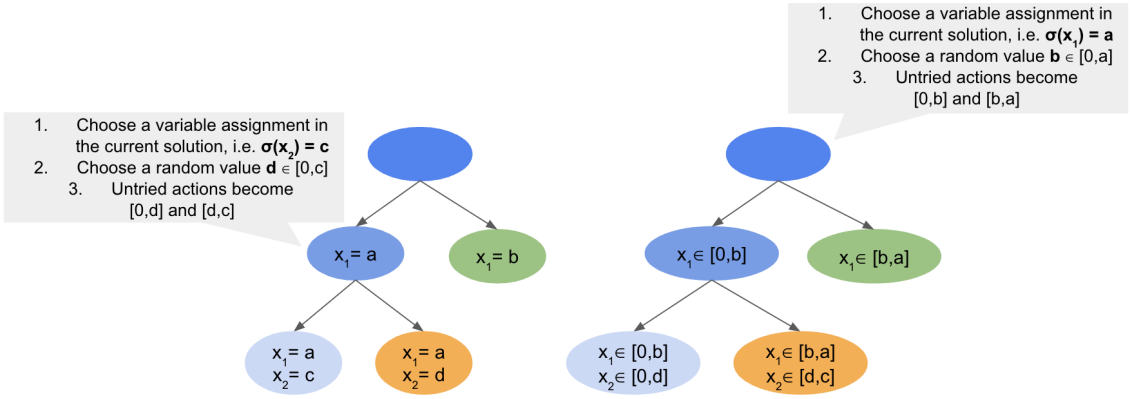

Figure 2.2. Point-MCTS (left) and Interval-MCTS (right)

cient of each variable in the objective function and they are usually static in the combinatorial optimization problem. Next, it expands new child nodes from the selected node n_i.

Point-MCTS If the node n_i was not expanded, it creates a child node with the state which has the variable x_{action} with the value $v_{current}$ in the solution of n_i by default. For the next child nodes, it expands a child node that has the same variable x_{action} with the different value v_{new}. If the decision variable is binary with its value 0 in the current solution, the new value will be 1. If the decision variable is integer or real, the new value will be integer or continuous value that has not been tried.

Interval-MCTS It expands two child nodes at each expansion. Assuming the upper bound of x_{action} is the value $v_{current}$ in the solution of n_i the lower bound of x_{action} is 0. It chooses a random value v_{new} in the original feasible range [0, $v_{current}$] and divides the range into two intervals using the value. One child node will have [0, v_{new}] and the other child node will have [v_{new}, $v_{current}$].

Algorithm 4. Expansion

0: **function** EXPAND(node: n_i)
0: $x_{action}, v_{current} \leftarrow$ choose variable from n_i.untried actions given gradients
0: $node_{new}$ = []
0: **if** is first expansion of n_i **then**
0: $n_{cd} \leftarrow$ Node(state = $[x_{action}, v_{current}]$, value = $n_i.value$, parent = n_i)
0: n_i.children.append(n_{cd})
0: $node_{new}$.append(n_{cd})
0: **end if**
0: **for** i=required number of child node **do**
0: $v_i \leftarrow$ choose new value for x_{action}
0: $n_{ci} \leftarrow$ Node(state = $[x_{action}, v_i]$, parent = n_i)
0: n_i.children.append(n_{ci})
0: $node_{new}$.append(n_{ci})
0: **end for**
0: **return** $node_{new}$

2.5 Simulation

A simulation is represented as a single solver call using the constraints specified by the current node. This solver is black boxed and can thus be represented by several different models, including Gurobi [2] and Z3 [1]. Recall that in our algorithm each branched node contains an additional assignment or interval that we impose as an added constraint on a particular variable. The constraints for the current node is composed of the conjunction of the original constraints and all assignments made at the current node and parent nodes, reaching back up to the root node. Thus, the solver is tasked to solve a modified variation of the original MILP and find a new solution. Finally, to limit the amount of time spent optimizing on each node, we restrict the

solver to run within a time limit of 5 seconds.

2.6 Backpropagation

Once a feasible solution has been obtained or the solver times out during simulation, the result is back propagated to the root node starting from the current simulated node. First, increment the number of visits to the current node. Then, in the case where a feasible solution was found, the node saves this solution if the objective value obtained is lower than that of all other solutions this node has observed in the past, assuming a minimization optimization problem. If no feasible solution was found during simulation (i.e. timeout), do nothing more. Finally, propagate this feasible solution to the parent node to repeat these steps, until the parent node is null and we have reached the root node.

Algorithm 5. Simulation and Backpropagation

```
0: function SIMULATION(node: n_i)
0:     ctrs ← n_i.state
0:     S ← FIX AND OPTIMIZE(ctrs)
0:     reward ← S objective value
1: return reward

0: function BACKPROPAGATION(node: n_i, reward: r)
   parent ← n_i.parent
0:     if r is timeout then
0:         n_i.value ← 0
0:     else if r is infeasible then
0:         parent.children.remove(n_i)
0:     else
0:         n_i.value ← r
0:         while n_i is not None do
0:             n_i.visits += 1
0:             if r < n_i.best value then
0:                 n_i.best value ← r
0:                 n_i = n_i.parent
```

Chapter 3

Experiments and Evaluation

We evaluate the performance of our approach with 3 benchmark MILP problems. The first two problems, job shop problem(JSP) and set covering (SC), are classic combinatorial optimization problems. JSP includes both continuous and discrete variables and SC has only binary variable. Next, we perform our method on a common real-world problem of worker scheduling (WS). We instantiate our framework in two ways: MCTS with value assignment (Point-Interval) and MCTS with interval assignment (Interval-MCTS).

Baselines The baselines are Gurobi and LNS with random decomposition and Gurobi. We use Gurobi 11.0 as the underlying solver for sub-routine problems in both LNS and our framework and set the time limit to 5 seconds for running each sub-problem. For Gurobi, we set the total time limit as 1000 seconds.

Hyperparameter Configuration For LNS method, we set the total iteration number of LNS as 100 and randomly generate 10 decompositions for each iteration. For our framework, there are two hyperparameters we have to choose: the maximum number of child node and the constant c to calculate UCT value. These hyperparameters decide the degree of exploration and exploitation in MCTS; as we set them with big value, it will explore new regions than exploit good regions it has found. In the experiments, we set 6 for the maximum number of child node and use 1 for c.

Evaluation Metrics For each benchmark problems, we run baselines and our methods

by five different random seeds. We compare the best-found value at the end of the run of all runs and the time spent to first reach that optimal value.

3.1 Job-Shop Problem

3.1.1 Problem Definition

Job-Shop problem is a classic NP-hard scheduling problem which is defined by a finite set J of n jobs and a finite set M of m machines. For each job j J, we are given a list $(\sigma_1^j, ..., \sigma_h^j, ..., \sigma_m^j)$ of the machines which represents the processing order of j through the machines and a list of their durations $(d_1^j, ..., d_h^j, ..., d_m^j)$. The objective is to find a schedule of J on M that minimizes the makespan, i.e., the maximum completion time of the last operation of any job in J.

Table 3.1. Problem instances

Problem	Problem size	Variables	Constraints
ft06	6 x 6	222 (42 continuous, 180 binary)	396
ft10	10 x 10	1010 (110 continuous, 900 binary)	1900
ft20	20 x 5	2020 (120 continuous, 1900 binary)	3900
ta09	15 x 15	3390 (240 continuous, 3150 binary)	6525
swv03	20 x 10	4020 (220 continuous, 3800 binary)	7800
ta13	20 x 15	6020 (320 continuous, 5700 binary)	11700
ta21	20 x 20	8020 (420 continuous, 7600 binary)	15600
ta32	30 x 15	13530 (480 continuous, 13050 binary)	26550
ta56	50 x 15	37550 (800 continuous, 36750 binary)	74250

3.1.2 Results

Our approach and other baselines could find the global optimal solution in a reasonable time for instances as small as ft06 and ft10. Interval-MCTS can find more optimal solutions than Gurobi for the bigger problems, i.e., ft20 and ta09. It found better solutions six times faster than Gurobi for ft20. There are some bottlenecks for the problem size that our approach can not scale. Gurobi discovers better solutions than our approach with the problem instances whose sizes are bigger than ta09. However, our approach could still find good solutions for some big instances

Table 3.2. Results of job shop problem

Problem	**Gurobi**		**LNS**		**Point-MCTS**		**Interval-MCTS**	
JxM	Objective	Time(s)	Obj	Time	Obj	Time	Obj	Time
6x6	**265**	0.1	**265**	0.2	**265**	0.2	**265**	0.2
10x10	**7460**	2.6	7734	5.0	**7460**	5.0	**7460**	5.0
20x5	14002	408.9	15000	5.0	14216	45.2	**13840**	70.3
15x15	17415	309.9	80833	5.1	17878	264.7	**17324**	666.1
20x10	**21534**	271.6	47017	5.1	24513	413.0	23624	130.8
20x15	**23788**	1000	33973	5.3	27140	51.2	24847	61.2
20x20	**29554**	985.1	-	-	30646	75.6	31295	826.8
30x15	**46137**	1000	-	-	60235	226.8	58230	206.8
50x15	-	-	-	-	-	-	-	-

where LNS results in solutions that have huge gaps with the best solution found by Gurobi.

3.2 Set Cover

3.2.1 Problem Definition

$\phi\bigcap$ The Set Cover problem is a NP-hard problem that involves finding the minimum number of sets that cover all elements in a given a universe U of n elements and a collection of subsets of U say $S = \{S_1, S_2, ..., S_m\}$ where every subset S_i has an associated cost. The goal is to find a subset C of S such that every element in U is contained in at least one set in C and the size of C is minimized.

Table 3.3. Set cover problem instances

J(n) x M(m)	Variables	Constraints
5000 x 1000	1000 (1000 binary)	5000
5000 x 2000	2000 (2000 binary)	5000
5000 x 4000	4000 (4000 binary)	5000

3.2.2 Results

As the problem consists of only binary variables, we use Point-MCTS and compare it with Gurobi baseline. As a result, our approach could find the optimal variable faster than Gurobi

and even found better objective values for some problem instances.

Table 3.4. Results of set cover problem

Problem size	**Gurobi**		**Point-MCTS**	
U x S	Objective	Time(s)	Objective	Time(s)
5000*x*1000	503	443.158	**500**	**250.46**
5000*x*2000	329	414.275	329	**5.037**
5000*x*4000	169	46.499	169	**5.043**

3.3 Worker Scheduling

3.3.1 Problem Definition

Worker scheduling is a problem that finds the weekly schedules of N employees that satisfy a store requirements and their working hour requirements. The store has the required number of workers needed each hour during its operation hours. Each worker should work consecutive hours in one's possible hour range. The workers also have the preferred number of days and hours to work and the goal is to find an optimal schedule that satisfies users' preferences as many as possible.

Table 3.5. Worker scheduling problem instances

N	Variables	Constraints
20	2568 (2568 binary)	1316
100	13836 (13836 binary)	6086
300	40272 (40272 binary)	17369
1000	132684 (132684 binary)	56797

3.3.2 Results

Similar to the set cover, it has only binary variables and the result shows Point-MCTS can find the optimal solution faster than Gurobi.

Table 3.6. Results of worker scheduling problem

Problem size	**Gurobi**		**Point-MCTS**	
N	Objective	Time(s)	Objective	Time(s)
20	29.085	0.016	29.085	0.002
100	125.80	0.62	125.803	0.001
300	328.14	5.37	328.14	0.03
1000	2434.942	21.36	2434.94	0.12

3.4 Analysis

Point-MCTS and Interval-MCTS From the performance perspective, Interval-MCTS could discover a more optimal solution than Point-MCTS for most of the instances as shown in Chapter 3. One of the possible factors that might affect Point-MCTS negatively is the small maximum number of child nodes *MC*. We use 6 for *MC* which means Point-Interval can search only 6 possible values of a variable in each level of the tree while Interval-MCTS could maintain the entire possible range of its value no matter how many children can be generated in each level. Furthermore, we also analyze the time each iteration took to re-optimize the solution to compare their time efficiency. As shown in Figure 3.1, the search time the solver took in each iteration of Point-Interval decreases consistently as the search space is partitioned into smaller sizes growing the MCTS tree. This shows the potential time efficiency of Point-MCTS.

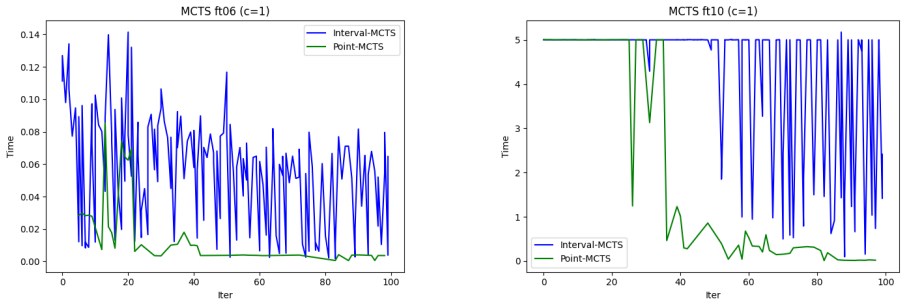

Figure 3.1. Time efficiency between Point-MCTS and Interval-MCTS

What makes problem hard Intuitively the complexity of the problem increases if a problem has more constraints and variables. However, the worker scheduling(WS) problem,

which had five times more variables than the job shop problem(JSP), was able to find a solution faster. This observation implies some factors make a problem inherently hard. One factor is that JSP has a much larger search space as it has some continuous values to predict. In addition, we focus on observing the correlation of variables between the constraints of their MILP problems. We visualize constraint graphs of similar size instances of each problem. The graph is constructed by connecting variables if they are in the same constraint. As you can see in Figure 3.2, JSP has more correlations between variables which means a change in one variable induces the changes in the majority of the variables.

Table 3.7. JSP instance and WS instance

Problem	Problem size	Variables	Constraints
JSP	6 x 6	222 (42 continuous, 180 binary)	396
WS	10	396 (264 continuous, 132 binary)	284

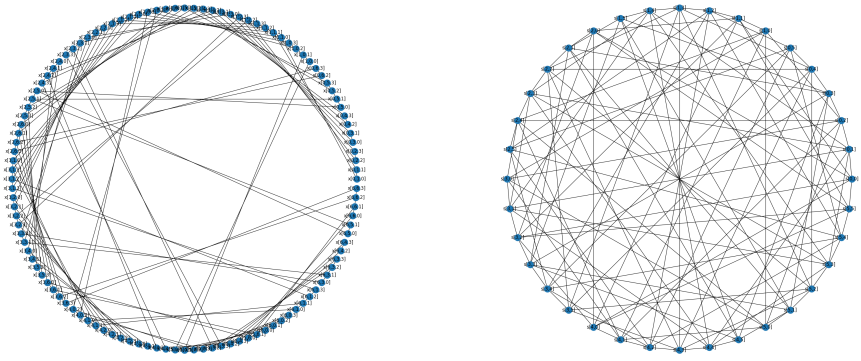

Figure 3.2. Constraint graph of instances above. WS (left) and JSP (right)

Chapter 4

Conclusion and Future work

We have presented a general MCTS framework for solving combinatorial optimization problems. The experiments show that the proposed method discovers better optimal solutions faster than the best commercial MIP solver Gurobi. Our method can also be applied to search in the continuous domain of variables unlike other approaches [6] that only work in discrete domains and are more scalable than the existing general framework for MILP [12]. We believe this approach has many interesting directions to be improved in the future. As we have many random components in MCTS design, we can incorporate a learning-based method to choose variable assignments to branch in the tree. Also, we can study how to efficiently balance exploration and exploitation of the tree through modeling data-driven UCT value formulation and a selection rule.

Bibliography

[1] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[2] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023.

[3] Zhiming Hu, James Tu, and Baochun Li. Spear: Optimized dependency-aware task scheduling with deep reinforcement learning. In *2019 IEEE 39th international conference on distributed computing systems (ICDCS)*, pages 2037–2046. IEEE, 2019.

[4] Angel A Juan, Javier Faulin, Josep Jorba, Jose Caceres, and Joan Manuel Marquès. Using parallel & distributed computing for real-time solving of vehicle routing problems with stochastic demands. *Annals of Operations Research*, 207:43–65, 2013.

[5] Elias Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[6] Elias B Khalil, Pashootan Vaezipoor, and Bistra Dilkina. Finding backdoors to integer programs: A monte carlo tree search framework. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 3786–3795, 2022.

[7] Sirui Li, Wenbin Ouyang, Max Paulus, and Cathy Wu. Learning to configure separators in branch-and-cut. *Advances in Neural Information Processing Systems*, 36, 2024.

[8] Jacek Mańdziuk and Maciej Świechowski. Uct in capacitated vehicle routing problem with traffic jams. *Information Sciences*, 406:42–56, 2017.

[9] Alexander Schrijver. *Combinatorial optimization: polyhedra and efficiency*, volume 24. Springer, 2003.

[10] Feng Shi, Ranjith K Soman, Ji Han, and Jennifer K Whyte. Addressing adjacency constraints in rectangular floor plans using monte-carlo tree search. *Automation in Construction*, 115:103187, 2020.

[11] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, and Marc

Lanctot. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[12] Jialin Song, Yisong Yue, and Bistra Dilkina. A general large neighborhood search framework for solving integer linear programs. *Advances in Neural Information Processing Systems*, 33:20012–20023, 2020.

[13] Karol Waldzik and Jacek Mańdziuk. Applying hybrid monte carlo tree search methods to risk-aware project scheduling problem. *Information Sciences*, 460:450–468, 2018.

[14] Yaoxin Wu, Wen Song, Zhiguang Cao, and Jie Zhang. Learning large neighborhood search policy for integer programming. *Advances in Neural Information Processing Systems*, 34:30075–30087, 2021.