Monte Carlo Tree Search on Perfect Rectangle Packing Problem Instances

Igor Pejic
Faculty of Science, University of Amsterdam,
The Netherlands
igorpejicw@gmail.com

Daan van den Berg
Faculty of Science, University of Amsterdam,
The Netherlands
D.vandenBerg@uva.nl

ABSTRACT

We explore the possibilities of Monte Carlo tree search (MCTS), immensely successful in games such as Go and Chess, to solve the perfect rectangle packing problem. Experiments are done on two differently generated problem sets of 1,000 instances each, and we explore six different rollout numbers and two different action-selection strategies for MCTS. We compare the algorithm's performance to an exact depth-first algorithm equipped with efficient pruning techniques. By rating the number of solutions found against the total number of tiles placed, we define a 'computationally economic tradeoff'. Different rollout numbers and strategies lead to different results, both within and between the two problem sets. We discuss these results in context of other heuristic algorithms on this problem and closely related areas.

CCS CONCEPTS

• **Computing methodologies** → **Heuristic function construction**; *Game tree search*; • **Mathematics of computing** → **Permutations and combinations**;

KEYWORDS

MCTS, Monte Carlo, Tree Search, Perfect, Rectangle, Packing Problem

ACM Reference Format:
Igor Pejic and Daan van den Berg. 2020. Monte Carlo Tree Search on Perfect Rectangle Packing Problem Instances. In *Genetic and Evolutionary Computation Conference Companion (GECCO '20 Companion), July 8–12, 2020, Cancún, Mexico.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3377929.3398115

1 INTRODUCTION

Having a long history in combinatorial optimization, modern-day packing problems are more acute and challenging than ever. Large instances are found in industries such as warehousing and data center management [10][15]. The slightly more specific rectangle packing problem, which involves orthogonally arranging rectangular objects in a rectangular container of minimal dimensions, is known to be NP-hard [23]. Practical examples can be found in high-end manufacturing industries such as VLSI as well as applications in job scheduling and pallet loading tasks [13][25][9].

The *perfect* rectangle packing problem (PRPP) is in some ways the somewhat simpler cousin of the rectangle packing problem. It entails orthogonally packing two-dimensional rectangular objects ('tiles') into a rectangular container ('frame'), but the combined surface area of the tiles is identical to the area of the frame so any solution, if existent at all, is a perfect fit without vacant space or overlapping tiles. Consequently, any given solution is polynomially verifiable, but since no subexponential solving algorithm is known, the problem is NP-complete even if rotation is not allowed [18][14][24]. A number of clever strategies have been devised to tackle the PRPP, sometimes reducing runtimes considerably [19, 20, 26, 34]. While these relatively efficient algorithms are 'complete' (they will always return the solution if it exists, or "no solution" otherwise), their worst-case run time is still (worse than) exponential.

It is no big surprise therefore, that a number of heuristic non-exact[1] methods such as genetic algorithms and tabu search have been applied to 2D packing problems [22][31] [2] [28]. Simulated annealing has also been used; a study by Katheryn Dowsland on non-perfect rectangle packing contains many valuable lessons, such as choosing sensible insertion points, or neighbourization through admissible operators[9] (also see [31]). But as an ominous omen to this and other studies on PRPP, Dowsland also warns for "expecting too much [from heuristic algorithms] in requiring an optimal solution", even on a singular objective value such as non-overlap [9]. It should be carefully recognized though, that the ordeal of finding good (or perfect) solutions for these optimization problems might not be caused by the non-existence of a subexponential algorithm alone. The intrinsic properties of the instances can make a big difference.

In this paper, we explore whether the Monte Carlo tree search (MCTS) could provide a reasonable alternative to forementioned heuristic algorithms. The algorithm's main idea is to find solutions by randomly sampling paths down the search tree that would otherwise be traversed by some exhaustive algorithm. By generating 'rollouts', random sequences of legal actions (i.e. tile placements) departing from the current configuration, it aggregates the objective values of potential future scenarios and chooses the candidate action which has the best outlook.

In recent years, Monte Carlo tree search has made its claim to fame by fulfilling a pivotal role in AlphaGo, the first ever algorithm to beat both the European and the world champion in the board

[1] We explicitly make the distinction, as exact algorithms may still contain heuristics for expected speedups.

GECCO '20 Companion, July 8–12, 2020, Cancún, Mexico

ACM ISBN 978-1-4503-7127-8/20/07…$15.00
https://doi.org/10.1145/3377929.3398115

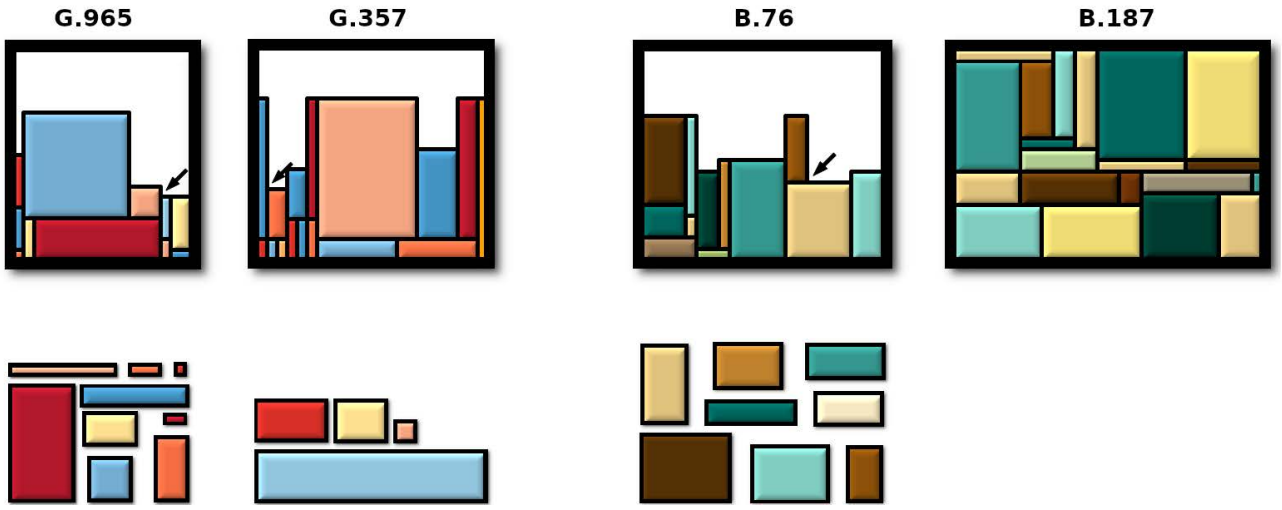

Figure 1: Four PRPP-instances from our study in various stages of the solving process. The G-instances (left) were created by iteratively 'guillotining' through tiles, the B-instances (right) by generating tiles of random dimensions and matching a frame afterwards. Every instance is known to have at least one solution, arrows point towards the bottom-left insertion point and overhead numbers are indices used for identification.

game of Go [32] [33] [4]. Having a search tree of $\approx 4.91 \cdot 10^{25,900}$ nodes (compared to chess: $\approx 3.35 \cdot 10^{122}$ nodes), AlphaGo's victories are a monumental achievement in combinatorial optimization [1]. For playing Go, the MCTS algorithm was equipped with two convolutional neural networks; a 'policy network', which selects candidate moves from which the rollouts are to be initiated and a 'value network' which evaluates the board situation of a game in progress. Go is a two-player turn-based game, but MCTS has also proven successful in single player Atari arcade game classics. Being too slow for real-time execution, a team from the University of Michigan moved MCTS offline to generate training data for a deep-learned classifier used in a real-time game playing agent [17]. In all these situations, the *best possible* action might be unknown but what is certain, is that MCTS plays well enough to beat the best adversarial algorithms and top-ranked human champions alike.

In light of these dazzling successes, it is easy to forget that in its most elementary form, MCTS is still a general tree search algorithm, and thereby potentially suitable to a much wider range of combinatorial problems than 'just' games. Relevant to our study, one investigation successfully applies MCTS to a large-scale robotized three-dimensional industrial packing problem [11]. The authors are explicitly not concerned with *perfect* solutions but with optimization which, considering the heuristic history on decision problems, might be a wise approach. In another study on MCTS for 3D packing, apparently related to shipping containers, an interesting observation is made. Reminiscent of earlier work on optimization problems, the results show that the instance properties of the boxes to be packed (the 'entropy' of the box types and numbers, loosely said[2]) play an important role in the quality of the found solution [27]. Similar phenomena have previously been demonstrated in other optimization problems, such as the traveling salesman problem (in planar and non-planar versions) [37] [12]. In this study too, the intrinsic properties of the PRPP-instances themselves will substantially influence the results.

All previously mentioned applications allow for some of the heuristic suboptimality MCTS inevitably brings along. This has proven not to be a problem; even if AlphaGo did not select *the absolute best* move, it was still good enough, as are shipping containers with a few inches of unused space left. For decision problems however, things are a little different as good solutions are not good enough – they need to be *perfect*. This requirement apparently makes MCTS an unappealing candidate for this category but still, its resounding successes in other areas warrant an exploration into its (im)possibilities. There have been some previous explorations in this direction though. In one study, the authors successfully apply MCTS to the generalized quantified constraint satisfaction problem (QCSP), but only when extended with constraint propagation [3]. QCSP is known to be PSPACE-complete, but it is unknown whether their QCSP-instances are in some way or another easier or harder to solve than regular CSPs. For SameGame, a study on MCTS by a team from Maastricht University exists[30]. Although SameGame's NP-completeness is debatable (we think it is NP-hard, even if the related "Clickomania" *is* NP-complete), MCTS was successfully applied. An optimization problem which is 'closer' to a decision problem is minimizing the number of colours needed to colour a graph (its 'chromatic number'). Strictly speaking, this too is an optimization problem, but as the number of applicable values is both discrete and usually limited, an iterative lowering of the upper bound can easily transform this problem into a short sequence of NP-complete decision problem instances [21]. In the end, the only two *pure* NP-complete decision problems we found were in two brief studies by Tristan Cazenave from Paris-Dauphine. In solving

[2]The authors themselves use the term 'heterogeneity'.

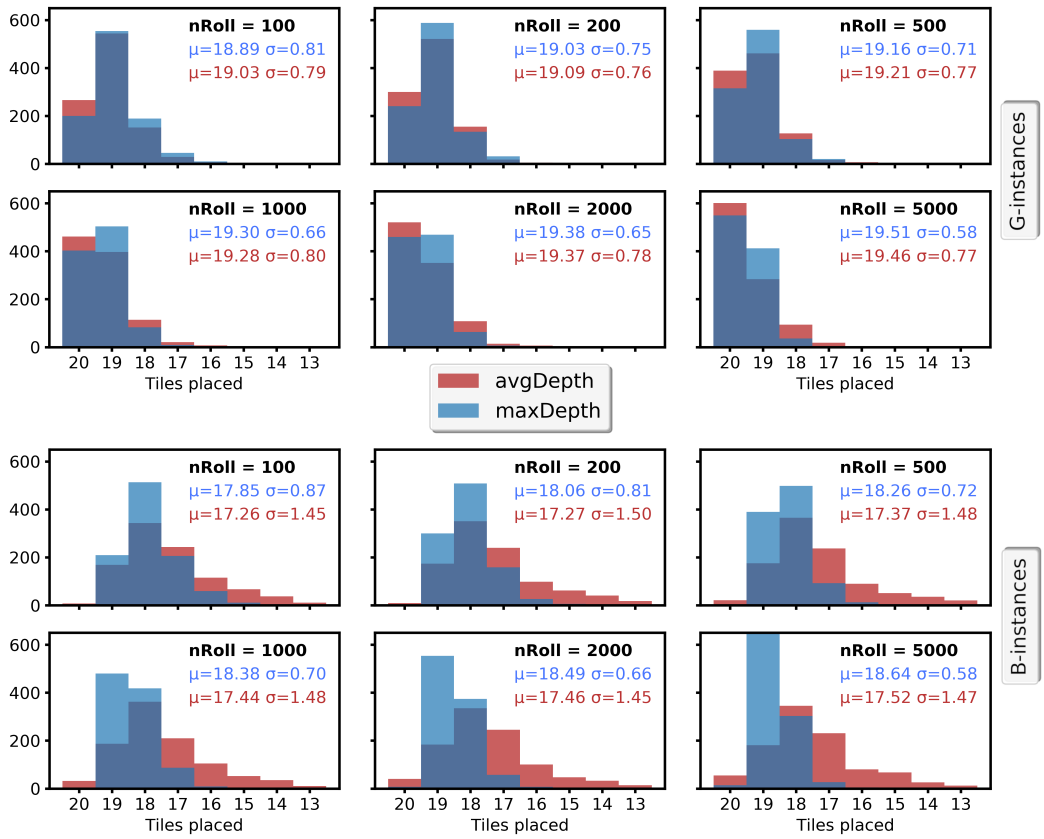

Figure 2: Numbers of placed tiles by Monte Carlo tree search per perfect rectangle packing problem instance. Generally, more rollouts increase the number of solutions, an effect that was stronger in the G-set than in the B-set. The *avgDepth* strategy for aggregating rollouts finds more solutions in all of the 12 subexperiments (look closely in the 6^{th} to 9^{th} subfigures), but counterintuitively, the *maxDepth* strategy places more tiles.

sudoku and related kakuro, he applies "nested Monte-Carlo search", which is close if not identical to MCTS, and outperforms depth-first search on a time-bound cutoff [7][6][5]. We have elaborated on this approach for PRPP, supplying somewhat more expansive data on partial solutions in Section 4. Cazenave's method of scoring rollouts through the depth of the sampled subtree, is comparable to ours. A comparison to an exact algorithm is also made.

Summarizing, it seems that the applications of MCTS to pure NP-complete decision problems are few and far between. In this study, we will explore 12 parameter settings for MCTS, and the effects on solving two different sets of 1,000 perfect rectangle problem instances each (Section 2). The algorithm will be outlined in Section 3, after we will present the results (Section 4) and discuss their significance in light of earlier findings and subtleties of instance hardness (Section 5).

2 2,000 PROBLEM INSTANCES TO SOLVE

To obtain a first foothold on the capabilities of the MCTS-algorithm on the perfect rectangle packing problem, we deploy two sets of instances, both containing 1,000 instances of 20 tiles. First, the *guillotineable instances* (G-set) is a set generated by iteratively cutting a tile in two smaller rectangular tiles. Starting off with a single tile enclosed in a frame of which both sides are random integers between 10 and 40, the generator selects a random point on a side and cuts straight to the other side, dividing the tile in two. Then, one of the tiles already in the set is chosen at random, again cutting it perpendicularly from a random point on either side. In total, this cutting procedure is executed 19 times, resulting in a 20-tile PRPP-instance which is then added to the G-set. Note that by this procedure, every instance is guaranteed to be solvable – there is at least one perfect packing inside the frame from which it was generated.

The second set consists of 1,000 PRPP-instances of 20 tiles from a collection created by Florian Braam at the University of Amsterdam in an ongoing (and yet unpublished) study. Braam chose an approach for generating instances which is sort-of-inverse to the guillotining approach. The process entails randomly generating 20 tiles of dimensions $1 \leq x, y \leq 30$, which are then retroactively matched to an available frame of integer dimensions by adding up the tiles' area, and listing all pairs of integers whose product equals

that area. If more such integer pairs exist, one is chosen at random. Instances with square or duplicate tiles were discarded, as were frames which are too eccentric to accommodate all tiles in both horizontal and vertical orientations. For this 'B-set', we randomly selected 1,000 of these instances which were proofed to have at least one solution by a run of an exhaustive recursive algorithm. By the enforced requirements, the combinatorial upper bound of all 20-tile PRPP instances in both sets is equal to the number of permutations multiplied by their orientational options, in this case $20! \cdot 2^{20} \approx 2.25 \cdot 10^{24}$. Not quite the search space of Go or Chess, but still quite sizeable and *just* unlarge enough to solve-proof with a good exhaustive algorithm.

So although the G-set and the B-set were generated by very different procedures, both sets consisted of 1,000 instances containing 20 tiles, all of which have least one perfect packing inside their respective frame, and all of which have an unpruned combinatorial state space of $\approx 2.25 \cdot 10^{24}$. It should be noted though, that the B-set, being more general in its generation procedure, *could* contain guillotineable instances too. All sets are publicly available for replication or inspection through our publicly accessible repository [29].

3 THE ALGORITHM

For solving the 20-tile perfect rectangle packing problems instances, 1,000 from the B-set and 1,000 from the G-set, we use the most basic form of Monte Carlo tree search[3]. There are no neural networks, no dynamic programming partial search trees stored in a priority queue, and not even a policy for limiting the width of the tree. The only 'bias' we deploy, is to use the 'bottom-left heuristic', which is widely used on this kind of packing problem[8]. Its quality is certainly debatable, as the 'border-first' heuristic has given some impressive results recently [34], but the motivation is to keep things as general as possible for now. Furthermore, the early heuristic lessons from Dowsland and others taught us that choosing a mechanism for considering only legal actions greatly reduces the action space but does not limit the reachability of possible solutions[9][31]. It is trivial to see that the root configuration C_{root} has exactly 40 legal actions: each of the 20 tiles can be placed either horizontally or vertically in the bottom left corner of the empty frame.

Each round, our MCTS considers the placement of every remaining tile in both orientations in the bottom-left insertion point of a base configuration C of earlier placed tiles. Any placement that fits without overlapping other tiles or the frame is considered a *legal action* from configuration C. Placing the tile from the first legal action results in configuration C_1, which is assigned an empty score list. Then, it will commence the first rollout from C_1: consecutively perform randomly chosen legal actions, placing one tile after another in the respective bottom-left insertion point, until no more tiles can be legally placed. The number of tiles that gets placed (eq.: the depth of the subtree) is the score of the rollout, so if 14 tiles are inside the frame before getting stuck, 14 is added to the score list of C_1. Then, backing up to configuration C_1, it will perform another rollout of random placements anew, adding another score to the score list of C_1, and so on until *nRoll* rollouts have been successfully executed. The final score of configuration C_1 is then aggregated by taking either the maximum from all the rollout scores, or averaging over all rollout scores, depending on which of the strategies *maxDepth* or *avgDepth* is used.

After *nRoll* rollouts have been completed and a score to C_1 has been aggregated by the strategy, configuration C_2 corresponding to the second legal placement from C will be scored in the same way. After all configurations $C_1 ... C_n$ emanating from all n legal actions from C have been scored, the configuration with the highest score is chosen, and the corresponding tile placement in the bottom-left insertion point is done, resulting in a new base configuration C after which a new round of MCTS rollouts begins. If a solution is

[3]At the time of writing, one reviewer at GECCO2020 suggested that professional opinions in the field may differ.

Algorithm 1 MCTS for Perfect Rectangle Packing

```
▷ Main procedure
state ← (nullMatrix(width, height), tiles)
while hasLegalMoves(State) do
    bestAction ← NULL
    bestMaxDepth, bestAvgDepth ← 0, 0
    for legalAction in getLegalActions(state) do
        newState ← applyAction(state, legalAction)
        actionDepths ← List()
        for nRoll do
            depth ← performSimulation(newState)
            actionDepths.append(depth)
        end for
        maxDepth ← max(actionDepths)
        avgDepth ← average(actionDepths)
        if strategy = 'maxDepth' and maxDepth > bestMaxDepth then
            bestMaxDepth ← maxDepth
            bestAction ← legalAction
        else if strategy = 'avgDepth' and avgDepth > bestAvgDepth then
            bestAvgDepth ← avgDepth
            bestAction ← legalAction
        end if
    end for
    state ← applyAction(state, bestAction)
end while
if tiles.length() = 0 then
    solutionFound ← true
else
    solutionFound ← false
end if

function performSimulation(state)
    depth ← 0
    while hasLegalMoves(state) do
        action ← random(getLegalActions(state))
        state ← applyAction(state, action)
        depth ← depth + 1
    end while
    return depth
end function
```

found during any rollout phase on any level of the tree, the whole process stops and the found solution is recorded.

The keen reader may already have noticed the chances of finding solutions might be greatly influenced by the two parameters which have been furtively introduced in this section: *nRoll*, being the number of rollouts per legal placement, and the strategy for aggregating the rollout scores, being either *maxDepth* or *avgDepth*. Both of these parameter settings are extensively tested in our experiments' section, which comes next.

4 EXPERIMENTS & RESULTS

Both the 1,000 G-instances and the 1,000 B-instances are run through our MCTS on both *maxDepth* and *avgDepth* strategies, with $nRoll \in \{100, 200, 500, 1000, 2000, 5000\}$ totalling to 24,000 runs. For both strategies, the number of found solutions increases with *nRoll*, as can be expected from a randomized heuristics algorithm. Quite consistently, *avgDepth* was the better strategy for finding solutions in all 12 parameter combinations, both on the G-instances and on the B-instances (Fig. 2).

More counterintuitively however, the mean number of placed tiles before getting stuck (μ) was higher for the *maxDepth* strategy on all B-instance experiments. This effect is witnessed even for low rollout numbers, but becomes more prominent as *nRoll* increases. This is also true for the G-instances, but only for values of $nRoll \geq 1000$. The effect appears closely tied to the standard deviation on *maxDepth*'s number of placed tiles, which monotonically decreases for both the G-instances ($0.81 \geq \sigma_{maxDepth(G)} \geq 0.56$) and the B-instances ($0.87 \geq \sigma_{maxDepth(B)} \geq 0.59$). This is not true for *avgDepth*'s standard deviation which hovers around 0,78 and 1,51 for the G-instances and B-instances respectively; a difference in distribution wideness that can be clearly seen from Figure 2. Summarizing, *avgDepth* is the better strategy for finding solutions regardless of the instance type. But paradoxically enough though, the *maxDepth* strategy gets more tiles placed – regardless of the instance type, given it does 1000 or more rollouts.

For an exact reference, we ran an exhaustive bottom-left backtracking algorithm with pruning and symmetry-breaking to compare yield/effort ratios, which is measured as the number of solved instances divided by the total number of legal actions required to reach that number. The result is multiplied by 10^9 for reasons of readability.

Even though the algorithm is complete and exact, it is a well-known fact that instance hardness can vary a great deal in NP-complete problems, even within the same instance size [16]. In most randomized ensembles, the vast majority of the instances is relatively easy, but a few are extremely hard to decide upon [36]. It is therefore common practice to define a *cutoff point*, either in wall clock time or in number of recursions, to prohibit these algorithms from unfeasible runtimes even though this *technically* means it releases them from their exactness[4] and turns them into non-randomized heuristic algorithms [36][35]. In our experiment, the maximum calculation time for the exact algorithm was set to one hour of processing time (approximately 24 million tile placements) and typically, a small number of instances was not solved within the time limit, even though a solution is guaranteed for every instance.

[4] equivalently: '*completeness*'

The deployment of the bottom-left tile insertion action for both the heuristic MCTS algorithm and the exhaustive solver enables us to perform a comparison of computational economy, in terms of solutions found within a 1000-instance set versus the total number of performed legal actions (tiles placed) (Fig. 3). Placement of a single tile took approximately 0.15 milliseconds on one fully utilized core on an Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz processor. Note that this comparison is rather coarse; the backtracking procedure of the exhaustive solver requires it to remove a tile for every tile it places, and data structures, memory management and stack size may all matter to some degree, even for a single instance size of 20 tiles. Still, it does give a sense of proportionality.

On these instance sets, MCTS does remarkably well, economically speaking. The *avgDepth* strategy outperformed the exhaustive solver on 5 out of 6 *nRoll* parameterizations on B-instances, and even 6 out of 6 for the G-instances. Surprisingly, the lower *nRoll*, the better the results. The *maxDepth* strategy, that places more tiles but finds fewer solutions, performs worse than the exhaustive solver on B-instances, but outpaces it by a huge margin on the G-instances. Again, smaller numbers of rollouts perform better for both sets, amounting to a whopping 1270-fold outperformance of the exhaustive solver on the G-set with $nRoll = 100$.

Finally, the yield/effort curve to number of rollouts on G-sets is tightly characterized by a power law for both *maxDepth* and *avgDepth* strategies ($12542 \cdot nRoll^{-0,61}$ and $20287 \cdot nRoll^{-0,65}$ respectively, both with $R^2 > 0.99$), as a straight line trend on the logarithmic vertical scale of Fig. 3. The significance of these tight fit power laws is yet to be determined, but might be uniquely related to the interplay between MCTS and the guillotine generation procedure.

5 DISCUSSION AND FUTURE WORK

In the end, it seems that the old dichotomy of decision problems versus optimization problems plays out for Monte Carlo tree search just as much as it does for other heuristical algorithms, but in a quite peculiar way. In the most general sense, the algorithm is not very good at finding solutions for the perfect rectangle packing problem, but it nonetheless does pretty well on placing many tiles, especially using the *avgDepth* strategy. But if one is prepared to waive the absolute guarantee of finding a solution if it exists, MCTS has a good economic tradeoff of computational effort versus solving probability. It should be noted though, that its results might be flattered a bit by the smallness of the instances, or the generation procedures. How these factors play out in larger NP-complete problem instances is yet to be seen. All in all, for these rectangle packing problems, MCTS might be a typical heuristic algorithm: better suited for finding good but non-optimal solutions in very reasonable time.

Like the many heuristic and exact applications prior to this study, a lot depends on the type of instances MCTS is required to solve. Upon posterior inspection, 937 of the G-instances contained at least one square tile, and no less than 999 had at least one duplicate tile, reducing an instance's potential search space. But even *within* the respective sets, great differences exist. Going back to figure 1, the left-hand instance G.965 took 42,167 legal actions to solve, the right hand side G.357 a whopping 22,190,041 legal actions. The same holds for both the two B-instances, whose 17,444 and 53,421,223

	B-Instances			G-Instances		
	#actions	#found	yld/eff	#actions	#found	yld/eff
AVG100	256,278,482	8	31.22	250,534,847	266	1061.73
AVG200	518,082,673	9	17.37	486,259,739	300	616.95
AVG500	1,304,127,579	21	16.10	1,095,306,143	389	355.15
AVG1000	2,607,250,826	32	12.27	1,987,119,667	461	231.99
AVG2000	5,203,417,411	40	7.69	3,577,742,373	521	145.62
AVG5000	12,905,554,577	55	4.26	7,460,459,402	601	80.56
MAX100	225,010,379	1	4.44	258,060,301	200	775.01
MAX200	451,921,492	0	0.00	495,013,900	241	486.86
MAX500	1,133,944,051	2	1.76	1,137,401,307	315	276.95
MAX1000	2,266,621,092	4	1.76	2,089,319,415	403	192.89
MAX2000	4,533,940,985	7	1.54	3,755,116,831	463	123.30
MAX5000	11,311,869,847	14	1.24	7,973,408,744	549	68.85
Exhaustive	210,476,678,804	985	4.68	1,293,569,739,654	816	0.63

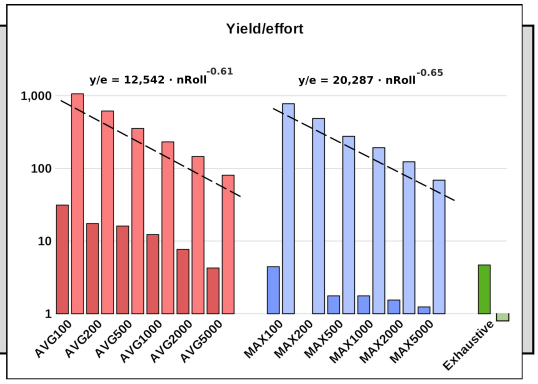

Figure 3: If one is prepared to waive absolute guarantees, MCTS can be an economically rewarding algorithm for finding solutions to perfect rectangle packing problems, especially using *avgDepth* strategy or on guillotined problem instances. On the latter, the economic tradeoff curve shows a strict power-law relation to the number of rollouts.

legal actions for B.76 and B.187 respectively, represent a tremendous difference in computational cost.

Another point of interest is the potential optimal number of *nRoll* in terms of computational costs, which confidently sits on the lower end at 100. It is easy to suspect this has to do with the relatively low number of tiles, but this is yet to be investigated. It could also be that, in accordance with other NP-complete problems, the vast majority of instances is simply dead easy, and thereby a low rollout number has a relative high immediate payoff [36][16].

And then there is the power law relationship between the number of rollouts and the yield/effort when solving G-instances. These relationships are usually associated with 'scale-free phenonema' that arise on every size magnitude. How would this phenomenon arise in G-sets? A possible explanation might be in the relative distribution of solutions along the fringe of the search tree. A result from the guillotining procedure, in which every step cuts a tile in two, is that every two simultaneously created tiles in the final solution can also be invertedly inserted, and thereby represent an associated *partially inverted* solution. As this is true on every level of cutting, each guillotined instance of m cuts has exactly 2^m 'partially reverted' equivalent solutions, which are fractally distributed among the search tree's leaf nodes. The two big open questions however, are whether the G-set contains no significant numbers of solutions other than the root solution and it partial-flips, and whether or why the B-set would not have an equivalent solution distribution among its search tree fringe. A final note might be that this relation only appears in true random (or maybe *unbiased*) MCTS-implementations, and might disappear otherwise.

In future work, the MCTS implementation might be improved with upper confidence bounds to guide the search process. Another improvement would be to temporarily store a bigger part of the search tree obtained through rollouts to allow for backtracking. Finally, a (deep) machine learning model could be used to estimate the score (reward) of a state and replace the *rollout* step with an estimated evaluation which could lead to better action selection.

ACKNOWLEDGMENTS

Maarten van Someren (1955-2019) had been involved in a great number of research and teaching activities with us at the University of Amsterdam. Moreover, he was one of my (Daan's) first teachers from back then, and in many ways the beginning of much of my own research & teaching. An incredibly soft-spoken man, kind to students and colleagues alike, I am sad to miss him, but he will be remembered with warmth in my heart. Rest in peace, Maarten, and thanks for everything. When I come up too, I'll show you my new chess algorithm.

It will beat yours for sure.

REFERENCES

[1] Louis Victor Allis et al. 1994. *Searching for solutions in games and artificial intelligence.* Ponsen & Looijen Wageningen.
[2] Ramón Alvarez-Valdés, Francisco Parreño, and José Manuel Tamarit. 2007. A tabu search algorithm for a two-dimensional non-guillotine cutting problem. *European Journal of Operational Research* 183, 3 (2007), 1167–1182.
[3] Satomi Baba, Yongjoon Joe, Atsushi Iwasaki, and Makoto Yokoo. 2011. Real-Time Solving of Quantified CSPs Based on Monte-Carlo Game Tree Search. *IJCAI International Joint Conference on Artificial Intelligence* (07 2011), 655–661. https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-116
[4] Steven Borowiec. 2016. AlphaGo seals 4-1 victory over Go grandmaster Lee Sedol. *The Guardian* 15 (2016).
[5] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4, 1 (March 2012), 1–43. https://doi.org/10.1109/TCIAIG.2012.2186810
[6] Tristan Cazenave. 2009. Monte-carlo kakuro. In *Advances in Computer Games.* Springer, 45–54.
[7] Tristan Cazenave. 2009. Nested monte-carlo search. In *Twenty-First International Joint Conference on Artificial Intelligence.*
[8] Bernard Chazelle. 1983. The bottomn-left bin-packing heuristic: An efficient implementation. *IEEE Trans. Comput.* 8 (1983), 697–707.
[9] Kathryn A Dowsland. 1993. Some experiments with simulated annealing techniques for packing problems. *European Journal of Operational Research* 68, 3 (1993), 389–399.
[10] Lu Duan, Haoyuan Hu, Yu Qian, Yu Gong, Xiaodong Zhang, Jiangwen Wei, and Yinghui Xu. 2019. A Multi-task Selected Learning Approach for Solving 3D Flexible Bin Packing Problem. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems.* International Foundation for

Autonomous Agents and Multiagent Systems, 1386–1394.
[11] Stefan Edelkamp, Max Gath, and Moritz Rohde. 2014. Monte-Carlo Tree Search for 3D Packing with Object Orientation. In *KI 2014: Advances in Artificial Intelligence*, Carsten Lutz and Michael Thielscher (Eds.). Springer International Publishing, Cham, 285–296.
[12] Thomas Fischer, Thomas Stützle, Holger Hoos, and Peter Merz. 2005. An analysis of the hardness of TSP instances for two high performance algorithms. In *Proceedings of the Sixth Metaheuristics International Conference*. 361–367.
[13] Julia Funke, Stefan Hougardy, and Jan Schneider. 2016. An exact algorithm for wirelength optimal placements in VLSI design. *Integration* 52 (2016), 355–366.
[14] Michael R Garey and David S Johnson. 1979. *Computers and intractability*. Vol. 174. freeman San Francisco.
[15] Fréjus AR Gbaguidi, Selma Boumerdassi, and Eugène C Ezin. 2017. Adapted Bin Packing Algorithm For Virtuals Machines Placement Into Datacenters. *Computer Science & Information Technology* (2017), 69.
[16] Carla Gomes and Bart Selman. 1999. On the fine structure of large search spaces. In *Proceedings 11th International Conference on Tools with Artificial Intelligence*. IEEE, 197–201.
[17] Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. 2014. Deep learning for real-time Atari game play using offline Monte-Carlo tree search planning. In *Advances in neural information processing systems*. 3338–3346.
[18] Stefan Hougardy. 2012. A scale invariant algorithm for packing rectangles perfectly. In *Proceedings of the fourth International Workshop on Bin Packing and Placement Constraints, Nantes, France*.
[19] Stefan Hougardy. 2012. *A scale invariant exact algorithm for dense rectangle packing problems*. Forschungsinstitut für Diskrete Mathematik, Rheinische Friedrich-Wilhelms-Universität Bonn.
[20] E. Huang and R. E. Korf. 2013. Optimal Rectangle Packing: An Absolute Placement Approach. *Journal of Artificial Intelligence Research* 46 (Jan 2013), 47–87. https://doi.org/10.1613/jair.3735
[21] Jiayi Huang, Md. Mostofa Ali Patwary, and Gregory F. Diamos. 2019. Coloring Big Graphs with AlphaGoZero. *CoRR* abs/1902.10162 (2019). arXiv:1902.10162 http://arxiv.org/abs/1902.10162
[22] Stefan Jakobs. 1996. On genetic algorithms for the packing of polygons. *European journal of operational research* 88, 1 (1996), 165–181.
[23] Claire Kenyon and Eric Rémila. 2000. A near-optimal solution to a two-dimensional cutting stock problem. *Mathematics of Operations Research* 25, 4 (2000), 645–656.
[24] Richard E. Korf. 2003. Optimal Rectangle Packing: Initial Results. In *Proceedings of the Thirteenth International Conference on International Conference on Automated Planning and Scheduling (ICAPS'03)*. AAAI Press, 287–295.
[25] Richard E Korf. 2004. Optimal Rectangle Packing: New Results.. In *ICAPS*. 142–149.
[26] N. Lesh, J. Marks, A. McMahon, and M. Mitzenmacher. 2004. Exhaustive approaches to 2D rectangular perfect packings. *Inform. Process. Lett.* 90, 1 (2004), 7 – 14. https://doi.org/10.1016/j.ipl.2004.01.006
[27] Hailiang Li, Yan Wang, DanPeng Ma, Yang Fang, and Zhibin Lei. 2018. Quasi-Monte-Carlo Tree Search for 3D Bin Packing. In *Pattern Recognition and Computer Vision*, Jian-Huang Lai, Cheng-Lin Liu, Xilin Chen, Jie Zhou, Tieniu Tan, Nanning Zheng, and Hongbin Zha (Eds.). Springer International Publishing, Cham, 384–396.
[28] Dequan Liu and Hongfei Teng. 1999. An improved BL-algorithm for genetic algorithm of the orthogonal packing of rectangles. *European Journal of Operational Research* 112, 2 (1999), 413–420.
[29] Igor Pejic. 2020. Intances from the G-set and the B-set used for solving PRPP. https://github.com/igorpejic/visapi/tree/master/problems. (2020).
[30] Maarten P. D. Schadd, Mark H. M. Winands, H. Jaap Van Den Herik, and Huib Aldewereld. 2008. Addressing NP-complete puzzles with Monte-Carlo methods. In *In proceedings of the AISB 2008 symposium on logic and the simulation of interaction and reasoning, volume 9*.
[31] Yuji Shigehiro, Seiji Koshiyama, and Tatsuya Masuda. 2001. Stochastic tabu search for rectangle packing. In *2001 IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace (Cat. No. 01CH37236)*, Vol. 4. IEEE, 2753–2758.
[32] 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. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484.
[33] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *Nature* 550, 7676 (2017), 354–359.
[34] Daan van den Berg, Florian Braam, Mark Moes, Emiel Suilen, and Sandjai Bhulai. 2016. Almost Squares in Almost Squares: Solving the Final Instance. *DATA ANALYTICS 2016* (2016), 81.
[35] Gijs van Horn, Richard Olij, Joeri Sleegers, and Daan van den Berg. 2018. A Predictive Data Analytic for the Hardness of Hamiltonian Cycle Problem Instances. *DATA ANALYTICS 2018* (2018), 101.
[36] Basil Vandegriend and Joseph Culberson. 1998. The G(n,m) phase transition is not hard for the Hamiltonian Cycle problem. *Journal of Artificial Intelligence Research* 9 (1998), 219–245.
[37] Weixiong Zhang and Richard E Korf. 1996. A study of complexity transitions on the asymmetric traveling salesman problem. *Artificial Intelligence* 81, 1-2 (1996), 223–239.