Monte Carlo Tree Search for Coarse Hex Blocking Generation

Paul Bourmaud [*] Rao Garimella [†] Navamita Ray [‡] Cristina C. Garcia [§]
Jean-Christophe Janodet [¶] Franck Ledoux [*]

Abstract

Decomposition of CAD models into shapes amenable for automatic meshing such as blocks is a crucial step in hexahedral mesh generation. In this paper, we propose a Monte Carlo Tree Search (MCTS) based approach to obtain possible decomposition solutions for a given CAD model which can be subsequently used for mesh generation. This prototype study shows promises in automating the decomposition of complex shapes into blocks and could be combined in the future with machine learning methods to enhance the method.

1 Introduction

Many numerical methods compute approximate solutions over a mesh of topologically simpler elements (tetrahedra, hexahedra) representing the computational domain. In highly non-linear problems, hexahedra are preferred, or even required, over tetrahedra because of their superior accuracy and directional control of the solution [21]. However, in spite of 30+ years of research, there are no reliable algorithms that can automatically generate hexahedral meshes for general CAD models [14, 15]).

The process of generating high-quality hexahedral meshes for complex shapes is at best a semi-automatic process. Analysts spend hours to weeks segmenting or decomposing complex CAD models into simpler shapes, like six-sided 3D blocks, that can be meshed using automatic algorithms. This process is a combination of experience and intuition, and automating it has been the holy grail of hexahedral mesh generation.

With the impressive success of machine learning in a wide variety of fields, it is worth investigating whether some learning-based techniques can be applied to the problem of block decomposition. Recently, DiPrete et al. [7] have explored the use of reinforcement learning to learn the sequence of steps to subdivide 2D shapes into general rectangles. The paper describes using a powerful Soft-Actor-Critic type Reinforcement Learning (RL) method to decompose polygonal shapes with all axis-aligned edges into rectangles. The decomposition is performed by cutting the model recursively along the model edges. The justification for using a powerful RL method to solve this simple problem is the promise of using continuous actions to decompose more complex shapes.

Some early papers have explored the use of knowledge-based tree search methods to decompose geometric models [13, 19]. Of particular note is the paper by Phillips et al. [13] in which they proposed a semantic-based intelligent tree search that is closely related to more modern methods explored here. In their work they avoid exploring the all the numerous possible paths by pruning those that produce undesirable intermediate stages. The work of Takata et al. [19] is similar but less general.

In this paper, we revisit the spirit of earlier knowledge-based attempts by applying newer intelligent search techniques to decompose a model into meshable blocks. In particular, we explore if the Monte Carlo Tree Search (MCTS) method [5] can be applied to the problem of creating a block decomposition of the shape by recursive cuts. The MCTS method has been used successfully in recent years in playing games such as Go [18] and chess. We show that the very simple MCTS-based method implemented here can generate viable decompositions for many of the 2D planar shapes tested in the study. This holds the promise that an improved algorithm combining MCTS with neural networks [18], utilizing more sophisticated rewards and a wider range of actions can be used effectively to decompose new shapes into meshable subdomains without human intervention.

We start by providing a brief background on the MCTS method in Section 2. In section 3, we describe the MCTS algorithm modified for constructing search

[*]CEA, DAM, DIF, F-91297 Arpajon, France

[†]Applied Computational Physics Division, XCP-4, Los Alamos National Laboratory, Los Alamos 87544, New Mexico, USA

[‡]Computer, Computational and Statistical Sciences Division, CCS-7, Los Alamos National Laboratory, Los Alamos 87544, New Mexico, USA

[§]Computer, Computational and Statistical Sciences Division, CCS-3, Los Alamos National Laboratory, Los Alamos 87544, New Mexico, USA

[¶]IBISC, University of Evry, Paris-Saclay University, 91025, Evry, France

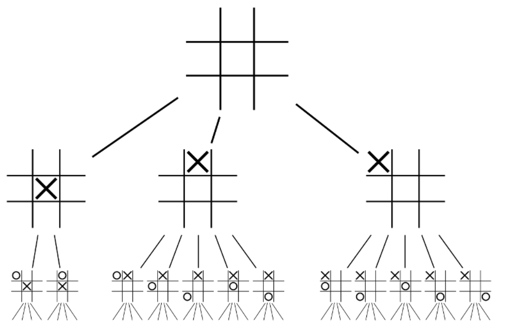


Figure 1: Example game tree of Tic Tac Toe game [8].

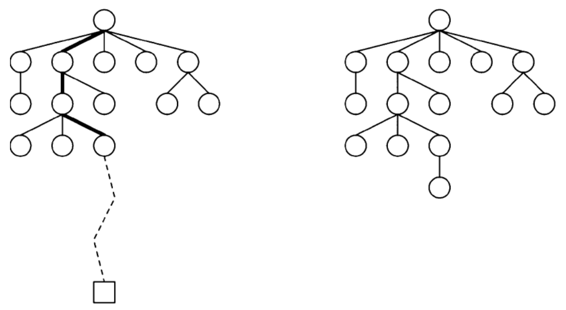


Figure 2: The basic MCTS process [1]

trees over CAD models. Finally, results and conclusions are presented in Sections 4 and 5.

2 Background

In this section, we describe the basic Monte Carlo Tree Search method and its key components.

2.1 Monte Carlo Tree Search Monte Carlo Tree Search (MCTS) is a widely used approach for solving finite size and finite horizon Markov Decision Process (MDP) based decision problems. It is a heuristic search algorithm that is used to obtain optimal or quasi-optimal solutions in a search space. MCTS achieves this by randomly sampling the decision space and by constructing a search tree based on the outcomes. In recent years, it has significantly influenced the field of artificial intelligence (AI) problems which can be described using a sequential decision tree, such as various games and planning scenarios.

MCTS method was first introduced by Coulom ([5]) and improved by Kocsis and Szepesvári ([9]). The algorithm is easier to understand by using the example of a combinatorial game. A game can be represented as a game tree (see Fig.1), where each tree node corresponds to a *state* of the game. The child nodes represent the possible next states for all possible *actions* that the player can take. Based on the number of states and the number of actions, the combinatorial space of possible states can be gigantic. For example, Phillips et al.[13] state that for a model with N vertices and m possible actions at each vertex, the total number of paths in the tree is equal to $N!m^N$. For $N = 6$ and $m = 3$, this means 524,880 possible paths resulting in 729 unique end states. In such scenarios, the goal is to conduct a more intelligent search by performing random sampling (called *simulations*) instead of brute force search, and capturing the effect of actions to make better choices in subsequent iterations.

The basic process is conceptually straightforward and is illustrated in Figure 2 (from [1]). It involves incrementally constructing the game tree in an asymmetric manner. During each iteration, a *tree policy* identifies the most critical node in the current tree. This policy aims to balance between *exploration* (venturing into less explored areas) and *exploitation* (focusing on promising areas). From this node, a simulation is executed, updating the search tree based on the outcome. A simulation is defined as a sequence of actions, either randomly selected or statistically biased, that is continuously applied to the current state until a terminal condition is met. The update process includes adding a child node corresponding to the action taken and adjusting the statistics of its ancestors. During the simulation stage, moves are determined by a *default policy*, often using uniformly random choices as a simple example. The algorithm progressively constructs a search tree until a predefined computational limit, usually defined by a bound on time execution, on memory footprint, or a number of iterations, is reached. Once this predefined limit is reached, the search concludes, and the optimal root action is returned.

The MCTS algorithm performs series of four steps [3] to iteratively construct the search tree as shown in Figure 3:

1. *Selection:* Beginning from the root node, a policy for selecting child nodes is applied recursively to traverse down the current tree until the most critical *expandable* node is reached. A node is considered expandable if it represents a state that is not terminal and has children that have not yet been visited or expanded;

2. *Expansion:* Involves adding one or more child nodes to grow the tree based on available actions,

3. *Simulation:* Entails simulating the newly added node(s) according to a default policy to determine an outcome;

4. *Backpropagation:* Refers to updating the statistics

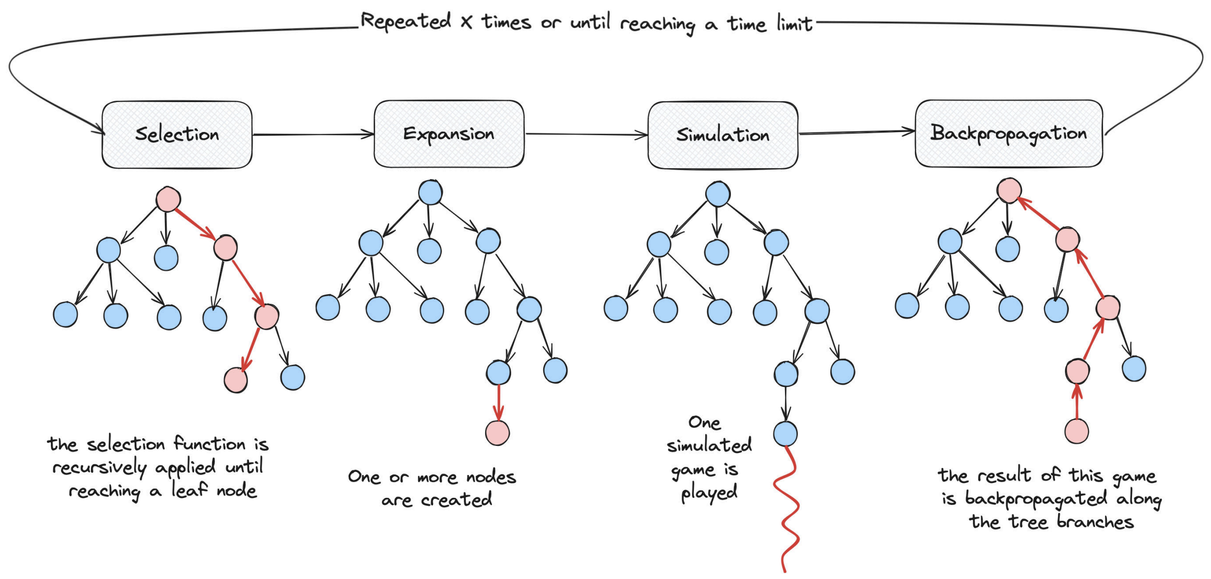

Repeated X times or until reaching a time limit
Selection
Expansion
Simulation
Backpropagation
the selection function is recursively applied until reaching a leaf node
One or more nodes are created
One simulated game is played
the result of this game is backpropagated along the tree branches

Figure 3: One iteration of the general MCTS approach [10]

of selected nodes by propagating the result of the simulation backward through the tree.

In contrast to depth-limited minimax search [17], MCTS does not require evaluating intermediate state values, and thus significantly reducing the needed domain knowledge. Only the final state's value at the end of each simulation is essential for decision-making.

Tree Policy The tree policy is used during the selection and expansion steps. The policy is essentially a strategy to reach a next new state by selecting or creating a new leaf node from an existing node within the search tree.

For example, Kocsis and Szepesvár[9], recommend the use of the following formula in order to choose the most promising node:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}. \tag{2.1}$$

In this formula:

- w_i represents the number of wins for the node under consideration following the i-th move;
- n_i represents the number of simulations conducted for the node under consideration after the i-th move;
- N_i represents the total number of simulations conducted up to the i-th move by the parent node of the node under consideration;
- c is the exploration parameter, ideally set to $\sqrt{2}$ in theory, but typically chosen empirically in practice.

The first part of this equation is used like the exploitation parameter, the fraction is large for the successors which have been so successful up to now. The second part corresponds to exploration; it plays a big part in choosing successors who have only been involved in a few simulations. This tree policy is chosen at the beginning of the search and remains fixed during the process. During the backpropagation phase, the values w_i and n_i of the formula 2.1 are updated for each selected node. We will use a modified version of this selection formula described in Section 3.2.1

Default Policy The default policy is used during the simulation step and dictates how the game is played when starting from a non-terminal state. As a result of the policy, the simulation step performs a sequence of actions that are either randomly selected or statistically biased according to some distribution is applied until a terminal state is reached. The end goal of a simulation step is generating an estimate of the value of a non-terminal state. A high-value state indicates that the chances of reaching a winning state are high.

During the backpropagation step, no policy is applied directly. Instead, node statistics are updated to guide future decisions made by the tree policy during selection. As mentioned above, the information carried by the nodes is updated. Algorithm 1 provides the pseudocode of the above steps.

Algorithm 1 Monte Carlo Tree Search (MCTS)

```
1: Function MCTS(state)
2: Create root node $v_0$ with state $s_0$
3: while within computational budget do
4:     $v_l \leftarrow$ TREEPOLICY($v_0$)
5:     reward $\leftarrow$ DEFAULTPOLICY($v_l$)
6:     BACKUP($v_l$, reward)
7: end whilereturn BESTCHILD(children of $v_0$)
8: End Function

1: Function TREEPOLICY(v)
2: while v is non-terminal and unexplored do
3:     if not all children of v are expanded thenchoisir
   return EXPAND(v) //Create a new child node
4:     else
5:         $v \leftarrow$ BESTCHILD($v$) //Using the selection
   policy to choose
6:     end if
7: end whilereturn v
8: End Function

1: Function DEFAULTPOLICY(v)
2: Randomly simulate to terminal state from v return
   reward from terminal state
3: End Function

1: Function BACKUP(v, reward)
2: while v is not null do
3:     Update v's visit count and total value with
   reward
4:     $v \leftarrow$ parent of $v$
5: end while
6: End Function
```

3 MCTS for CAD Decomposition

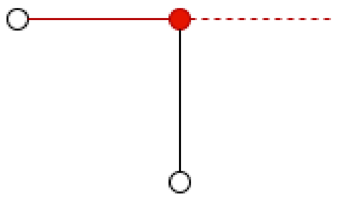
The block decomposition of a planar two-dimensional CAD model can be viewed as a single-player game where the game play involves applying a discrete set of actions that modify the geometry sequentially to reach a terminal state where either the model is decomposed or not resulting in a winning or a losing game state. We first begin by describing the representation of the decomposition game.

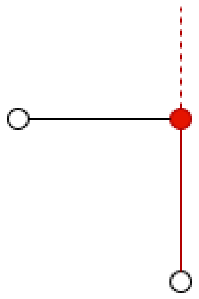
3.1 Game Representation

State Our approach considers the entire CAD model, incorporating all the entities that make up the model in the state evaluation. This means that every aspect of the model, from geometric shapes to intricate details, is taken into account during the decomposition and optimization process. By adopting this comprehensive approach, we capture the interactions between different parts of the model and ensure that each entity is properly addressed in the decomposition process. Taking the whole CAD model into account allows us to achieve a more accurate and consistent decomposition that faithfully reflects the model's overall structure and constraints.

The CAD model is represented using a full-featured 3D geometric modeler called OpenCascade [11] and is queried and modified through an in-house simplified Python [20] API based on PythonOCC [12].

Actions Our action set contains three different actions that can be applied at a model vertex to realize the CAD decomposition (illustrated in Figure 4). The three actions correspond to cutting along model edges at a model vertex and a bisector.


(a) Cut along red edge 1.


(b) Cut along red edge 2.

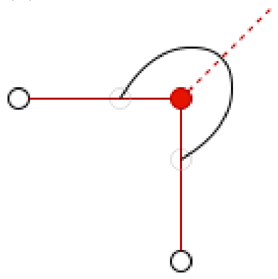
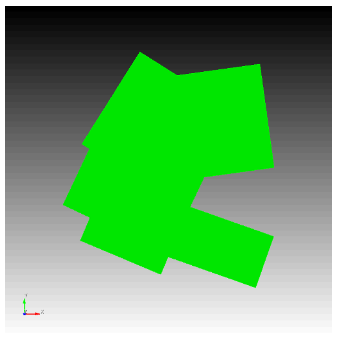

(c) Cut along angle bisector.

Figure 4: Possible actions from a model vertex.


(a) Root CAD Model

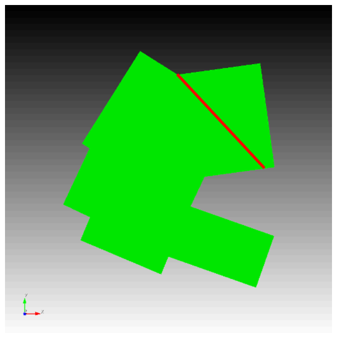

(b) State obtained after a cut dissecting the model

Figure 5: Example of a CAD Model as a root and a child node after an action is applied.

In our approach, the action list provided by a state is derived by exploring all the boundary vertices of the model. For each boundary vertex, we consider the possibility of executing one of three available actions. This method ensures that all potential actions are evaluated and included in the action list, allowing for a comprehensive exploration of the possible transitions and manipulations of the model. By examining each boundary vertex and permitting any of the three actions, we ensure that the action list reflects a diverse set of options, thereby enhancing the flexibility and effectiveness of the decomposition process.

Root and Child Nodes We start from the CAD model as the root node. A child node is a modified model after an action has been applied. Figures 5a and 5b illustrate the root and child nodes for an example CAD model.

Terminal state A terminal state is reached when all the model faces are quadrilateral shapes (see Figure 7) or when we generate a triangular face (see Figure 6). We consider shapes that logically represent a quadrilateral such as shown in Figure 8 as quadrilaterals as well. We name it a general quadrilateral. The detection of such faces relies on a tolerance parameter on the incident angle at a vertex.

At the end of the algorithm, after n iterations, we obtain a partially constructed tree, which may include one or more final states that are a solution to the given problem. This may be due to the fact that it is possible to take several paths (sequence of actions) leading to the same state, but on different leaves. At the same time, several different states meeting the termination criteria can be considered as victory since the solution is not necessarily unique.

Quality function For evaluating the effect of actions, we use the state of the CAD model after a simulation to build the following quality function:

$$\mathcal{R} = \gamma N_Q + \beta N_{GQ} - \alpha N_T - \theta N_P, \tag{3.2}$$

where N_Q, N_{GQ}, N_T, N_P are respectively the number of pure quadrilaterals, general quadrilaterals, triangles, and polygons, and γ, β, α, θ their respective weights. As mentioned above, the quality function allows us to evaluate a state obtained during the simulation

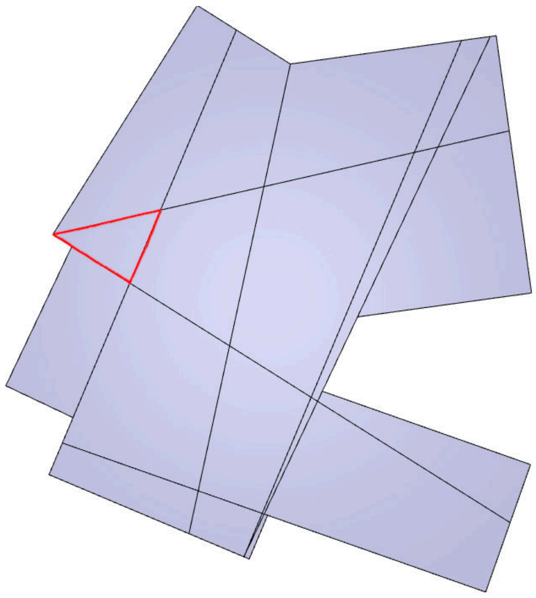


Figure 6: Terminal state where a triangle is generated.

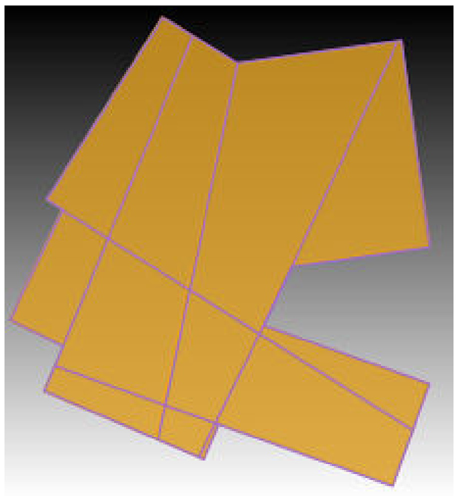


Figure 7: A terminal state where the initial CAD shape is fully decomposed into a set of quadrilateral faces.

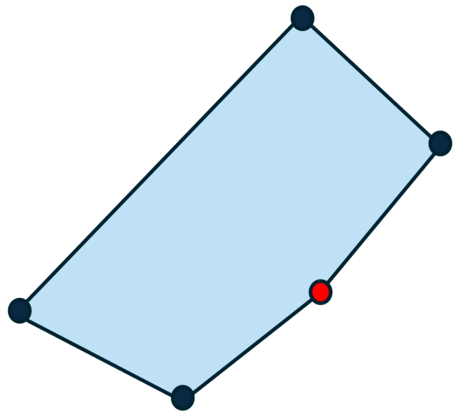


Figure 8: General Quadrilateral with five edges. The black node denotes the corners, and the red node denotes a side vertex.

phase. Note that we have not chosen to incorporate quadrilateral quality into the quality function in this initial study.

3.2 MCTS Decomposition Algorithm

We now explain the four key components of the MCTS algorithm when applied to the decomposition game.

3.2.1 Selection

As mentioned in the background section, the selection step of the MCTS algorithm is a strategic process aimed at choosing one of the children of a given node while managing the trade-off between *exploitation* and *exploration*.

Exploitation focuses on actions that have previously yielded the best results, while exploration involves considering less promising moves that might still offer potential, given their uncertain evaluations.

Various algorithms [2] have been developed to handle this selection process. One such tree policy for a single-player game is an alternative version of the Upper Confidence Bound for Trees (UCT) described in Schadd et all [16] :

$$\frac{w_i}{n_i} + C.\sqrt{\frac{\ln N_i}{n_i}} + \sqrt{\frac{\sum_{k=0}^{n_i} (w_k)^2 - n_i.(\frac{w_i}{n_i})^2 + D}{n_i}}. \tag{3.3}$$

The first two terms in the original formula 2.1 use the number of times a node n has been visited, denoted n_i, and the number of times a parent node N has been visited, denoted N_i, to compute an upper confidence bound for the average game value $\frac{w_i}{n_i}$. The term w_i is the sum of all previously obtained results. We use the formula 3.2 to evaluate at the end of a simulation (see Section 3.2.3).

For a single-player game, Schadd et al. [16] modified this formula by adding a third term that accounts for the deviation of the child node's results from the expected value (original formula 2.1). This term incorporates the sum of the squared outcomes obtained thus far $(\sum_{k=0}^{n_i}(w_k)^2)$ in the child node, adjusted by the expected results $n_i(\frac{w_i}{n_i})^2$.

In single-player games, unlike in two-player games where outcomes are generally constrained to win, draw, or lose (corresponding to values in $[-1, 1]$), scores can significantly exceed these bounds. To address this issue, the constants C and D in the UCT formula need to be adjusted to fit the score range of the game. Alternatively, scores can be scaled to fit within a normalized range, such as $[-1, 1]$, using theoretical upper bounds. For our approach, we set C to $\sqrt{2}$, as it is the theoretically proven value, and D to $10,000$. This choice is motivated by the significant variation in the value of our score depending on the geometries, so using a high value minimizes potential risks. According to the literature [16], it is recommended to set the parameter D equal to the maximum value of the evaluation criterion. Since the value of our score changes for each new CAD provided, it is challenging to determine a consistent value or predict the expected outcome when obtaining a valid result. Based on this, we chose to set D to $10,000$ to ensure it would not negatively affect our selection process during the selection phase.

Another important difference is that single-player games do not involve an opponent whose moves introduce uncertainty. Thus, the optimization focuses solely on improving the game score without considering adversarial actions.

3.2.2 Expansion

When a leaf node is reached, the expansion strategy determines which nodes are added to the tree under the selected node. We use the approach proposed in Coulom [5] for expanding one child node per simulation, where the expanded node is the first encountered new position that is not already in the tree.

3.2.3 Simulation

For the simulation step, we start from a leaf node and make random moves until reaching the end of the game (win or lose). The game ends if we generate a terminal state, that is, a solution with only quadrilaterals or at least one triangle. We can, therefore, say that our default policy consists of a random choice of available actions for a given state. This allows for a faster simulation phase than would be the case if we were to choose actions using heuristics. We perform 1000 simulations before selecting a new root node. With this process, we reduce the complexity of the tree because we cut a part of the tree with each new root selection. We repeat this process until we obtain a solution with a terminal state (cf. Section 3.1). After the end of the simulation, we go to the back-propagation phase.

3.2.4 Back-Propagation

During the back-propagation phase, the simulation result at the leaf node is transmitted backward up to the root. Various back-propagation strategies have been explored in the literature [4][2][5]. Our approach uses the simple average of the simulations by updating w_i and n_i. To do this, we increment by 1 all the values n_i for each node leading to the node where the simulation occurs. For w_i, we add the quality of the state (using formula 3.2) obtained at the end of the simulation. Consequently, we update (1) the average score of a node ($\frac{w_i}{n_i}$), (2) the sum of the squared results $((w_i)^2)$, which is necessary for the third term in the selection strategy (see Formula 3.3), and (3) the UCT value.

The four phases are repeated until the game time expires. When time runs out, a final move selection determines which move should be played.

4 Results

For our experiments, we generated a set of general planar polygonal shapes by merging a small number (2-10) of randomly rotated planar rectangles. We used the CUBIT package [6] for creating such shapes. Figure 9 shows some of the generated shapes based on this strategy. The MCTS algorithm is then tested on this set of shapes. The algorithm constructs a search tree for each individual shape, and at the end of the search provides either a search tree with leaf nodes corresponding to a success state, with a full quadrilateral decomposition, or a failure state, where at least one single triangle occurs.

Figure 10 showcases successful outcomes of the algorithm on a set of shapes where a full quadrilateral decomposition was obtained. For the shapes shown in each subfigure in Figure 10, the algorithm terminated after finding a search path to a full quadrilateral decomposition. In particular, the algorithm for the shape in subfigure 10b correctly found that an edge-aligned cut rather than a bisection cut would lead to a quadrilateral decomposition when the shape is axis-aligned. We should reiterate that since the reward function did not include any measure of quality of the blocks, this is a success in a purely topological sense. Overall, these results demon-

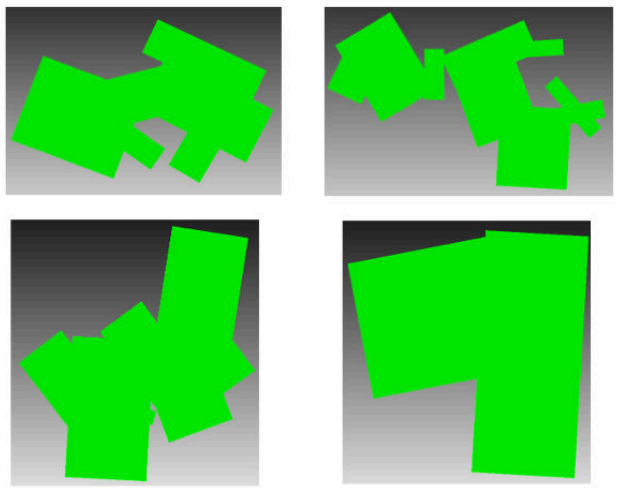


Figure 9: Planar polygonal shapes generated from rotated rectangles.

strate the effectiveness of our approach in managing and optimizing the decomposition of CAD models. They illustrate how even a simple MCTS framework can lead to appropriate and suitable solutions.

However, it is possible that the MCTS algorithm does not achieve a terminal state with only quadrilaterals. For example, Figure 11 illustrates the partial decompositions due to presence of triangles at leaf nodes. These results highlight that, in some cases, the decomposition process is incomplete, leaving residual polygons in the model. This issue arises because the algorithm terminates when all children of the selected root node are triangles, which may prevent the discovery of a final, fully optimized solution.

Despite incomplete decompositions, the algorithm can be made useful by utilizing intermediate child nodes as a starting guess to complete the decomposition process. This property can be of significant help for complex shapes, particularly in 3D where such guesses can be reasonably useful guide for further decomposition.

On Figures 10 and 11, we provide the computation time of our MCTS approach for each illustrated model. This computation time can be explained by the fact that, during the construction of the tree, each decomposed model is saved for every node in our search tree. This decision was made due to technical constraints, as the code did not ensure consistent naming of decomposition elements based on their IDs. As a result, applying an action to a node with ID i does not guarantee the same outcome across different runs. Therefore, we were forced to proceed as described, which led to an increase in the complexity of our code.

5 Conclusions

In this paper, we have demonstrated the application of the MCTS algorithm for the problem of CAD decomposition in 2D. The results show that such an algorithm can be effective in producing a fully quadrilateral decomposition for two-dimensional planar polygons.

On the other hand, due to the inherent stochastic nature of the algorithm, the search tree can result in partial decompositions. The current approach is limited by several factors that affect its overall performance and versatility. One notable constraint is the action list's restriction to only cutting type operations faces, which hampers the method's ability to handle more complex geometries.

6 Future Work

This initial approach has already shown promising and encouraging results, which bodes well for the future. The first 2D results demonstrate that the Monte Carlo Tree Search (MCTS) algorithm is a promising method for exploring the solution space in 2D mesh problems. However, expanding to 3D will require the integration of additional actions. In the future, we plan to add more types of action, such as one-sided cuts instead of through cuts, and cuts along more directions than just the bisector, to increase the probability of finding a good decomposition. We will also explore the opportunity to incorporate geometric terms into the reward function to generate decompositions with better aspect ratios and angles. Furthermore, we aim to couple the MCTS algorithm with a neural network that can predict the best action to take in each state. Performance improvements can be achieved by parallelizing the implementation and removing redundant or unnecessary actions.

Finally, we plan to expand our test bench to include a wider variety of shapes and extend the method to handle simple three-dimensional models. The promising results from our 2D approach show that MCTS is a valuable tool for exploring solution spaces, but moving to 3D will require adapting the approach and introducing new actions to account for the added complexity.

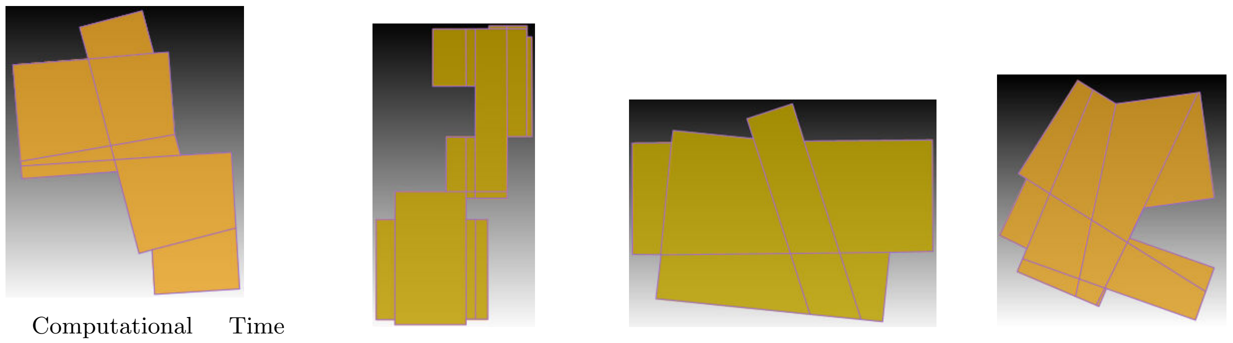
Acknowledgements

Redacted for anonymity.

This publication has been reviewed and approved for unlimited release (LA-UR-24-29786).

References

[1] H. Baier and P. D. Drake, *The power of forgetting: Improving the last-good-reply policy in monte carlo go*, IEEE Transactions on Computational Intelligence and AI in Games, 2 (2010), pp. 303–309.



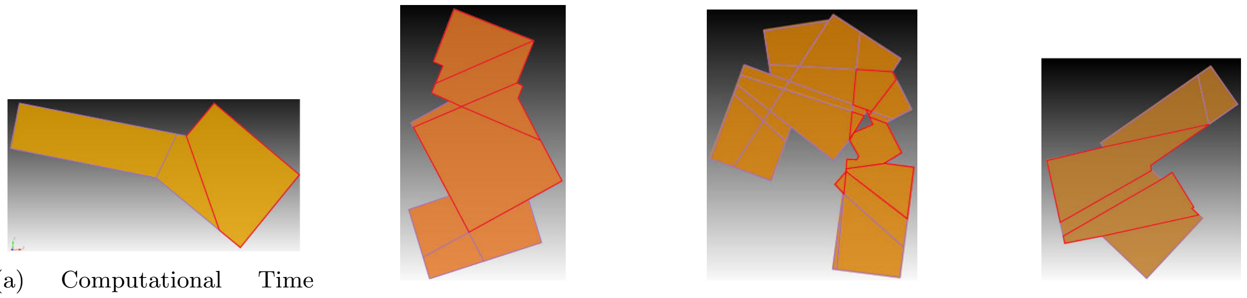
(a) Computational Time (CT): 1,946 seconds

(b) CT: Not recorded

(c) CT: 1,740 seconds

(d) CT: 3,359 seconds

Figure 10: Examples of successful decompositions using the MCTS algorithm



(a) Computational Time (CT): 174 seconds

(b) CT: 3,273 seconds

(c) CT: Not recorded

(d) CT: 1863 seconds

Figure 11: Examples of decompositions that did not meet the requirements. In red, the unexpected parts.

[2] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, *A survey of monte carlo tree search methods*, IEEE Transactions on Computational Intelligence and AI in Games, 4 (2012), pp. 1–43.

[3] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, *Monte-carlo tree search: A new framework for game ai.*, 01 2008.

[4] G. Chaslot, J. Saito, B. Bouzy, J. Uiterwijk, and H. van den Herik, *Monte-carlo strategies for computer go*, in BNAIC'06: Proceedings of the 18th Belgium-Netherlands Conference on Artificial Intelligence, P.-Y. Schobbens, W. Vanhoof, and G. Schwanen, eds., University of Namur, Jan. 2006, pp. 83–91. BNAIC'06: Proceedings of the 18th Belgium-Netherlands Conference on Artificial Intelligence University of Namur Namen ; BNAIC'06 ; Conference date: 05-10-2006 Through 06-10-2006.

[5] R. Coulom, *Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search*, in 5th International Conference on Computer and Games, P. Ciancarini and H. J. van den Herik, eds., Turin, Italy, May 2006.

[6] CUBIT, *The Cubit Geometry and Mesh Generation Toolkit*, Sandia National Laboratories, Albuquerque, NM, USA, 2022.

[7] B. DiPrete, R. Garimella, N. Ray, and C. Cardona, *Reinforcement learning for block decomposition of planar cad models*, Engineering with Computers.

[8] Inconnu, *Tic-tac-toe game tree*, 2021. Fichier SVG sur Wikipedia.

[9] L. Kocsis and C. Szepesvári, *Bandit based monte-carlo planning*, vol. 2006, 09 2006, pp. 282–293.

[10] F. Ledoux, *Spam yields for **Single PlAyer Monte carlo tree search**. it is an implementation of monte carlo tree search that focuses on single-player games.*, 2024. Accessed: 2024-12-11.

[11] OpenCascade, *OpenCascade Technology, 7.7.0dev*, OpenCascade.com, Moulineaux, France, 2022.

[12] T. Paviot, *pythonocc – python interface to opencascade*, 2024. https://github.com/tpavio/pythonocc.

[13] L. R. Phillips, J. L. Mitchiner, T. D. Blacker, and Y. T. Lin, *A knowledge system for automatic finite element mesh generation: Ameks*, in Proceedings of the 1st International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems - Volume 2, IEA/AIE '88, New York, NY, USA, 1988, Association for Computing Machinery, p. 668–678.

[14] N. Pietroni, M. Campen, A. Sheffer, G. Cherchi, D. Bommes, X. Gao, R. Scateni, F. Ledoux, J.-F. Remacle, and M. Livesu, *Hex-mesh generation and processing: a survey*, ACM Transactions on Graphics, 42 (2022), pp. 1–44.

[15] J. Sarrate, E. Ruiz-Gironés, and X. Roca, *Unstructured and semi-structured hexahedral mesh generation methods*, Computational Technology Reviews, 10 (2017).

[16] M. P. D. Schadd, M. H. M. Winands, H. J. van den Herik, G. M. J. B. Chaslot, and J. W. H. M. Uiterwijk, *Single-Player Monte-Carlo Tree Search*, Springer Berlin Heidelberg, 2008, p. 1–12.

[17] S. Shevtekar, M. Malpe, and M. Bhaila, *Analysis of game tree search algorithms using minimax algorithm and alpha-beta pruning*, International Journal of Scientific Research in Computer Science, Engineering and Information Technology, (2022), pp. 328–333.

[18] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, J. van den Driessche, G.; Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, Nature, 529 (2016), pp. 484–489.

[19] O. Takata and et.al., *A knowledge-based mesh generation system for forging simulation*, Applied Intelligence, 11 (1999), pp. 149–168.

[20] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*, CreateSpace, Scotts Valley, CA, 2009.

[21] E. Wang, T. Nelson, and R. Rauch, *Back to elements - tetrahedra vs hexahedra*, in Proceedings of the 2004 International Ansys Conference, 2004.