RoboGrammar: Graph Grammar for Terrain-Optimized Robot Design

ALLAN ZHAO, JIE XU, MINA KONAKOVIĆ-LUKOVIĆ, JOSEPHINE HUGHES, ANDREW SPIELBERG, DANIELA RUS, and WOJCIECH MATUSIK, Massachusetts Institute of Technology

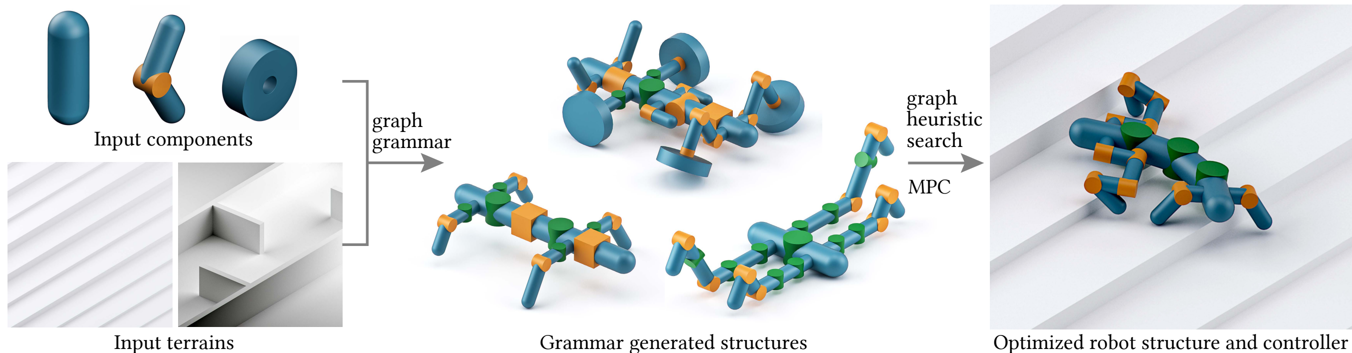

Fig. 1. The input to our system is a set of base robot components, such as links, joints, and end structures, and at least one terrain, such as stepped terrain or terrain with wall obstacles. *RoboGrammar* provides a recursive graph grammar to efficiently generate hundreds of thousands of robot structures built with the given components. We then use Graph Heuristic Search coupled with model predictive control (MPC) to facilitate exploration of the large design space, and identify high performing examples for a given terrain. Our approach enables co-optimization of both robot structures and controllers.

We present *RoboGrammar*, a fully automated approach for generating optimized robot structures to traverse given terrains. In this framework, we represent each robot design as a graph, and use a graph grammar to express possible arrangements of physical robot assemblies. Each robot design can then be expressed as a sequence of grammar rules. Using only a small set of rules our grammar can describe hundreds of thousands of possible robot designs. The construction of the grammar limits the design space to designs that can be fabricated. For a given input terrain, the design space is searched to find the top performing robots and their corresponding controllers. We introduce Graph Heuristic Search – a novel method for efficient search of combinatorial design spaces. In Graph Heuristic Search, we explore the design space while simultaneously learning a function that maps incomplete designs (e.g., nodes in the combinatorial search tree) to the best performance values that can be achieved by expanding these incomplete designs. Graph Heuristic Search prioritizes exploration of the most promising branches of the design space. To test our method we optimize robots for a number of challenging and varied terrains. We demonstrate that *RoboGrammar* can successfully generate nontrivial robots that are optimized for a single terrain or a combination of terrains.

CCS Concepts: • **Computing methodologies → Procedural animation**; **Evolutionary robotics**; **Search methodologies**.

Additional Key Words and Phrases: graph grammars, graph neural networks

Authors' address: Allan Zhao; Jie Xu; Mina Konaković-Luković; Josephine Hughes; Andrew Spielberg; Daniela Rus; Wojciech Matusik, Massachusetts Institute of Technology.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2020 Copyright held by the owner/author(s).
0730-0301/2020/12-ART188
https://doi.org/10.1145/3414685.3417831

ACM Reference Format:
Allan Zhao, Jie Xu, Mina Konaković-Luković, Josephine Hughes, Andrew Spielberg, Daniela Rus, and Wojciech Matusik. 2020. RoboGrammar: Graph Grammar for Terrain-Optimized Robot Design. *ACM Trans. Graph.* 39, 6, Article 188 (December 2020), 16 pages. https://doi.org/10.1145/3414685.3417831

1 INTRODUCTION

The automation and understanding of robot design, and the interplay between the structure and controller of a robot has long been a key research question [Hiller and Lipson 2011]. This is a particularly challenging research problem as the design space is vast and intractable and there are limited tools for automatically and efficiently exploring it. To enable large scale search and optimization of robots we need new approaches for structuring this design space, tools for searching it efficiently, and simulations for exploring and evaluating many thousands of designs [Pugh and Martinoli 2007]. Importantly, the tools developed must allow for the emergence of imaginative, or even inventive solutions, which diverge from those that manifest from more traditional design approaches [Pollack et al. 2003].

To address this challenge, we introduce a simulation-based system for simultaneously optimizing the physical structures and controllers of robots. The goal of the system is to take a set of user-specified primitive components and generate an optimal robot structure and controller for traversing a given terrain. The primitive components include different joint types, links, and wheels, each with user prescribed attributes such as rotational angles and axes, sizes, and weights. The user can specify the makeup of the primitives to match what physical components they have available. In the *RoboGrammar* framework each robot is represented by a graph. To efficiently search the design space of robot graphs, we introduce a recursive graph grammar that emphasizes mobility and fabricability.

Our grammar is expressive and maps realistic components to fabricable configurations, and can adapt to changing component primitives. Furthermore, we couple the grammar with physical simulation and controller synthesis, allowing us to rapidly create, optimize, and test robot designs. We use model predictive control (MPC) to provide a stochastic approach to controller learning. In order to make the search of a large design space efficient, we introduce a novel Graph Heuristic Search algorithm which generalizes the knowledge of explored designs to predict performance of unexplored branches of the search space. Specifically, our search algorithm takes a learning-based approach inspired by reinforcement learning, iteratively exploring a large space of robot designs for a given task, and learning a heuristic function to gradually steer that search toward optimal designs. Our learning model takes a neural-based approach, exploiting a graph neural network architecture to provide a fast method for approximating the performance metrics of best designs.

In summary, this paper presents the following key contributions:

- A recursive graph grammar that enables the generation of a wide range of inherently fabricable robot forms, which can be built in their simulated configurations.
- A Graph Heuristic Search method for efficiently searching the design space described with the grammar. This is benchmarked against alternatives including Monte Carlo tree search and random search.
- A demonstration of terrain-driven optimization using MPC based stochastic evaluation of each proposed design. We show the variety of innovative robot designs that are obtained across six different terrains. In addition, our approach identifies a number of high-performing robots for a single terrain or combination of terrains.

The following section of this paper reviews related work presenting current state-of-the-art research. In Section 3 we provide an overview of the different systems and algorithms used to enable the grammar-driven exploration and optimization of robots. Section 4 focuses on the graph grammar which is developed to express a wide range of different robots. We detail the components of the grammar, and the rules enabling the creation of innovative robot structures. In order to evaluate the performance of robots and find controllers we use model predictive control (MPC) with integral control and low-pass filtering (Section 5). In Section 6 we present the optimization process, providing details of the Graph Heuristic Search and the Monte Carlo tree search implemented for sake of comparison. The results are presented in Section 9, showing best-performing designs generated using *RoboGrammar* for six different terrains, as well as combinations of terrains. We conclude with a discussion of the limitations of our approach and identify avenues for future work.

2 RELATED WORK

In this section we review techniques for generative design of robots, and existing state of the art work relating to generative design, graph grammars, and controller optimization.

2.1 Generative design for robotics

The development of effective generative design tools for robotics, including search and evolutionary design algorithms, is a key research challenge [McCormack et al. 2004]. We review several approaches, detailing the various advantages they offer and the applications for which they are best suited.

Many recent approaches use *deep learning* which provides powerful tools for generative design. [Pathak et al. 2019] develop a modular co-evolution strategy in which a collection of primitive agents learn to dynamically self-assemble into a composite body while also learning to coordinate their behavior to control the body. However, their method results in only simple robots with a few components, and assumes physically implausible reconfigurability. Reinforcement learning has also been applied in order to continually improve agent design [Ha 2018]. In their design framework, both the environment and robot are altered to enable the agent to learn more effectively. Deep learning methods are particularly suited to co-optimization of the robot body and controller [Schaff et al. 2019], and have the potential to offer improved performance. Within the domain of soft robots, there has also been an example of end-to-end controller and structure design using deep latent representations [Spielberg et al. 2019].

Formal Design Methods are an alternative approach which use a model based approach to design robots and their controllers. Jing et al. [2018] demonstrate the formal approach using a large design library and bespoke tools to configure modular robots. The arrangements of modular robots are particularly suited for these methods [Bi and Zhang 2001; Chen and Burdick 1995]. Such methods can require significant hand crafting and do not scale or generalize well.

To search a parameterized design space, in our case a graph grammar, efficient search algorithms are required. *Genetic algorithms* (GAs) iteratively improve designs through a process inspired by natural evolution [Bongard 2013]. Applied to real-world systems [Brodbeck et al. 2015], GAs transfer positive 'phenotypes' between generations of robots, and allow the development of large populations of robot designs [Watson et al. 2002]. Sims [1994] evolves creatures optimized for movement in a three-dimensional environment using a GA. Creatures are represented as directed graphs, which may include multi-edges. Each creature's kinematic tree is determined by tracing unique paths through its directed graph. This representation encodes symmetry and repetition in the graph itself, leading to more biologically plausible forms. It is not clear how the creatures translate into robotic components, however. The effect of terrain on optimal morphology is also not addressed, as all creatures are simulated on a flat surface or in fluid. GAs have been widely used within the robotics community with demonstrable success [Koza 1995], however, in some cases they do not scale well and can have a high sensitivity to input parameters [Sivanandam and Deepa 2008]. They can also be sensitive to parameters such as population size and rate of mutations or crossovers, and there is little evidence demonstrating convergence to global or even local minima.

There are also a number of *hybrid approaches* which combine multiple methods. Wang et al. [2019b] propose Neural Graph Evolution, an evolutionary search in graph space, which iteratively

evolves graph structures using simple mutation primitives. Wang et al. [2019a] pairs the generation of environments with the optimization of agents. Their method simultaneously explores many different paths through the space of possible problems and solutions, utilizing neural networks and optimization techniques. However, without a judicious grammar to help constrain the search, their approach has only been shown to scale to robot designs with a few joint and link components. Further, their reliance on a genetic algorithm to search over the design space has the same pitfalls as discussed previously.

In this paper we explore two candidates for grammar-based generative robot co-optimization. First, *Monte Carlo tree search* (MCTS) is a stochastic search algorithm widely used across a variety of problem domains [Browne et al. 2012]. It has been applied to game based decision making, notably for playing Go [Gelly et al. 2012], as well as robot optimization and planning [Munos et al. 2014; Nguyen et al. 2017]. The algorithm identifies the most promising moves, expanding the search tree based on random sampling of the search space. Schadd et al. [2008] have shown the ability to extend MCTS to a single-player scenario, which is the case when searching over a grammar. Despite its wide applicability, MCTS suffers from low searching efficiency on complex robot design problems. We therefore use it as a baseline. Our core algorithmic contribution is an novel search optimization strategy, referred to as *Graph Heuristic Search* (GHS). GHS generalizes the knowledge from explored designs to untested designs to improve searching efficiency. By learning an estimator of performance, a heuristic can guide the search, helping to find optimal solutions faster. We apply deep learning, specifically graph neural networks (GNNs), to learn this heuristic. GNNs are particularly suited for learning on topological or structured inputs, such as robot designs [Lederman et al. 2018]. A variety of GNN models exist, but the differentiable-pooling model [Ying et al. 2018] is of particular interest for grammar based exploration.

2.2 Graph grammars

In this work, we examine *recursive graph grammars* as a method for generating expressive search spaces that are restricted to feasible designs with user-defined components. Formal grammars, sets of production rules for creating valid strings from a language's alphabet, are widely used for linguistics and natural language processing [Chomsky 1956]. This concept can be extended to graph grammars, which define sets of valid graphs rather than linear sequences. For the purposes of generative design, the graphs usually describe spatial configurations of mechanical components. Early examples include graph grammars for epicyclic gear train systems [Schmidt et al. 1999] and Meccano®-based machines [Schmidt and Cagan 1997]. In robotics, graph grammars have been applied to modeling self-assembly of robotic systems [Klavins et al. 2004] as well as the structure of robot interactions and control laws [Smith et al. 2009]. More recently, a graph grammar has been developed to describe the physical structure and also the fabrication process for furniture [Lau et al. 2011]. This grammar allows complex 3D furniture models to be expressed as manufacturable parts and connectors. Scalability and applicability to physical systems are key advantages of graph grammars [Hornby and Pollack 2001]. They provide a way of parameterizing the search space to prevent intractability, while allowing many different structures to emerge.

Most analogous to our work is Stöckli and Shea [2015], which describes passive dynamic brachiating robots with a graph grammar and evaluates them using dynamic simulation. Their work showed the power of graph grammars in automated robot design, with a wide range of robots emerging from a relatively limited grammar. Their work did not examine the expensive control synthesis aspect (as well as the resulting co-optimization problem) and focused solely on 2D systems with swinging locomotion. In our scenario, simulation and robot evaluation is far more complex, and requires a more scalable search algorithm.

2.3 Grammar-based procedural modeling

Shape grammars use graphical primitives to generate complex geometric shapes. Initially introduced by Stiny and Gips [1971], shape grammars have been used in architectural design [Downing and Flemming 1981; Duarte 2005; Stiny and Mitchell 1978]. A simplified version of shape grammars which is more popular in computer graphics applications is the set grammar [Stiny 1982]. Grammars for modeling streets and buildings have been proposed in [Jesus et al. 2016; Krecklau et al. 2010; Müller et al. 2006; Parish and Müller 2001; Wonka et al. 2003]. Van Diepen and Shea [2019] apply shape grammars to soft robot design, using predetermined actuation patterns instead of full control synthesis.

A majority of high-performing grammars are still manually designed by experts. Several works propose automating grammar creation based on a set of example designs [Bokeloh et al. 2010; Št'ava et al. 2010; Wu et al. 2014]. Lipp et al. [2008] develop a framework for interactive grammar editing. Dang et al. [2015] expand shape grammars with probability density functions defined through an interactive design exploration tool to obtain designs with higher preference score. A probabilistic grammar is also used in [Liu et al. 2014] to parse unseen scenes and assign segmentation, labels, and object hierarchies.

2.4 Control methods and approaches

In addition to generating the structure of the robot, some method of generating appropriate controls is necessary. To allow exploring many different robot structures across different terrains the control approach must be highly efficient and robust.

Model predictive control (MPC) is a family of control algorithms used widely for robotics and process control [Garcia et al. 1989]. A model of the system is used to make predictions about future behaviour, with online optimization used to find optimal controls to meet a desired output. Within robotics it is widely used to develop trajectory tracking controllers [Klančar and Škrjanc 2007; Kuhne et al. 2004]. There are many variants of MPC suitable for different settings. When no derivatives of the dynamics are available, for example in a non-differentiable simulator, sampling based MPC methods such as model predictive path integral control (MPPI) [Williams et al. 2016] are typically used. Low-pass filtering has been shown to be helpful in other simulation-based applications of MPC [Lowrey et al. 2018].

Another increasingly popular control approach is the use of reinforcement learning. Stochastic policy gradient reinforcement learning can enable a 3D biped to walk over rugged terrain [Tedrake et al. 2004]. Peng et al. [2017] demonstrate how hierarchical deep reinforcement learning can learn dynamic locomotion skills on challenging terrains with a limited amount of prior knowledge. Reinforcement learning is characterized by the need for large amounts of data, which can hamper efforts to wrap control synthesis in an outer design search loop.

3 SYSTEM OVERVIEW

RoboGrammar consists of three main components listed below. Figure 1 provides a graphical overview.

First, a recursive graph grammar forms the core of our approach to optimizing the structure of robots (Section 4). We use a graph representation for robot structure and define the set of components and grammar rules which can be used to assemble robots. Our grammar encodes simple and intuitive rules in order to efficiently generate interesting, feasible robot designs during the search process.

Secondly, each design generated by the grammar is evaluated using the model predictive control (MPC) algorithm described in Section 5. MPC aims to rapidly find a stable, periodic gait. The objective function optimized by MPC rewards high forward speed and maintaining the initial orientation.

The third component of our system is the novel Graph Heuristic Search (GHS) algorithm described in Section 6. GHS searches over the design space defined by the grammar to efficiently identify optimal robots and controllers. The algorithm exploits a graph neural network-based heuristic function, whose architecture is analogous to the graph-like structure of rigid robots. The heuristic function is learned as the search progresses using ground-truth data from the MPC-based evaluations.

We describe how users specify problems in our framework in Section 7, and provide implementation details in Section 8. Different terrains are used to embody objective functions for which the robots are optimized. We demonstrate our system on single and multi-objective optimization problems, and showcase our results in Section 9.

4 ROBOT GRAMMAR

Starting with a given set of robot components, our goal is to efficiently explore over the space of robots that can be formed from these components. However, this space can be combinatorially large and primarily composed of nonsensical designs. In order to make the search more tractable, we constrain it with a bio-inspired graph grammar. The structural rules are inspired by arthropods, which make up a majority of known animal species. Note that we only use bio-inspiration as a starting point. We choose to include wheels, which enable even more forms of locomotion at the cost of only one additional rule. Our grammar includes many familiar forms, while allowing novel designs to emerge.

4.1 Robot representation

In order to enable robot design with graph grammars, we represent robots in the form of directed acyclic graphs. Each node of the graph represents a physically realizable component. We consider robot structures to consist of body segments and limbs, with optional head and tail. The structures are composed of rigid links and rigid or articulated joints. Each body segment can have at most one pair of legs attached to it.

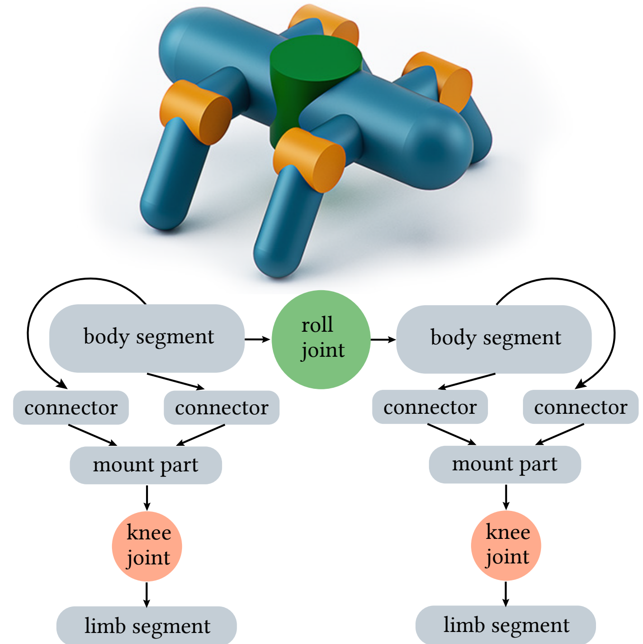

Fig. 2. An example of a kinematic tree (top) with the corresponding robot graph (bottom). To enforce symmetry in leg pairs, after adding nodes for connectors on both sides of the body, both legs of one pair are defined in one branch of the graph.

As our grammar is based on arthropods, it only describes symmetrical robots. Legs are always added in pairs and each pair has identical leg structure on both sides. Exceptions to this rule are the optional head and tail limbs, which are individually formed and are not necessarily symmetrical. We encode symmetry and repetition by representing each pair of legs with a single branch in the robot graph (see Figure 2). This scheme helps simplify rule definitions in the grammar.

After deriving a full robot graph, we convert it to a kinematic tree for efficient simulation. Note that all robots in our grammar, as well as the vast majority of animals and existing robots have kinematics that can be described by a tree. A new robot component is added per graph node, for each unique path connecting the tree root and that node. Finding the root node involves selecting an arbitrary node and following edges backwards until a node with no incoming edges is reached. Each graph node may produce multiple components in the kinematic tree due to our symmetry-enforcing scheme.

4.2 Grammar definitions

We define a recursive graph grammar $\mathcal{G}$ as a tuple

$$\mathcal{G} = (\mathbf{N}, \mathbf{T}, \mathbf{A}, \mathbf{R}, S)$$

Grammar structural rules

Body structure

r_1: $S \Rightarrow H \leftarrow B \leftarrow T$

r_2: $T \Rightarrow Y \leftarrow B \leftarrow T$

Adding appendage to body

r_3: $B \Rightarrow U$; $U \rightarrow C$, $U \rightarrow C$; $C \rightarrow M$, $C \rightarrow M$; $M \rightarrow E$

r_4: $B \Rightarrow U$

Appendages

r_5: $E \Rightarrow J \rightarrow L \rightarrow E$

r_6: $T \Rightarrow C \rightarrow M \rightarrow E$

r_7: $H \Rightarrow E \leftarrow M \leftarrow C$

Legend

S	start symbol
H	head part
Y	body joint
B	body part
T	tail part
U	body link
C	connector
M	mount part
E	limb end
J	limb joint
L	limb link

Fig. 3. A list of structural rules of our robot grammar. Here $S, H, Y, B, T, U, E, J, L \in \mathbf{N}$ are non-terminal symbols. Rule r_1 initializes the body structure, while r_2 can be used to extend the body. Note that each body segment U can have at most one pair of limbs attached to it. Rule r_3 enforces symmetry of the limb pairs, and rule r_4 allows body segments without limbs. Rule r_5 serves for extending the limbs, and r_6 and r_7 for adding back and front limbs.

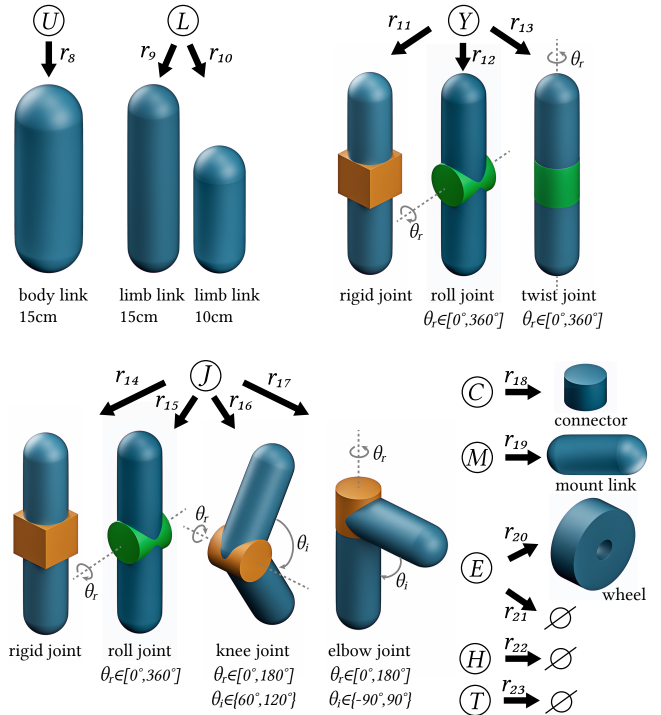

Fig. 4. Component-based rules of the robot grammar. Initial-pose angle θ_i and rotational range angle θ_r are attributes of joints. Adding, removing, or editing any link or joint component is straightforward. Note that we do not use a physical representation for the head, tail, hands, feet, and other types of end components for simplicity. However, the grammar can easily incorporate any new component representing an end structure.

where $\mathbf{N}$ and $\mathbf{T}$ are sets of non-terminal and terminal symbols respectively, $\mathbf{A}$ is a set of attributes for certain terminal symbols, $\mathbf{R}$ are production rules and $S \in \mathbf{N}$ is a starting symbol. Non-terminal symbols are temporary graph nodes that help us construct different body and leg parts. Terminal symbols are final graph nodes which represent physical robot components (e.g., links, joints, wheels, etc.). We refer to graphs with only terminal symbols as "complete" robot designs, and all other graphs as "partial" robot designs. In addition, we assign attributes to several terminal symbols. The attributes define the initial state of the robot by determining initial lengths and angles between robot parts. Each production rule from $\mathbf{R}$ has the form:

$$Q \Rightarrow W$$

where $Q \in \mathbf{N}$ is a non-terminal symbol, and W is a graph that has at least one node, no matter if it is a non-terminal or a terminal symbol (see Subsection 4.3 for more details). A production rule is applied to a current robot graph by detecting an occurrence of Q to replace with W. Since our grammar is recursive, a non-terminal symbol Q can appear again on the right hand-side of the rule as part of W. A recursive grammar allows us to concisely represent a wide variety of structures, including complex body shapes with numerous limbs such as a centipede.

4.3 Production rules and robot components

Here we present the production rules used to create robot structures. We divide our robot grammar rules into two main categories: *structural rules* and *component-based rules*. Structural rules serve to construct a physically realistic topology for the robot and define the number of body and limb segments. Every structural rule has at least one non-terminal symbol on the right-hand side. The list of these rules is given in Figure 3.

Component-based rules replace non-terminal symbols with terminal symbols representing robot components. Graph nodes with terminal symbols cannot be replaced. Several terminal symbols have assigned attributes. Attribute $\theta_r \in \mathbf{A}$ limits the rotational range of a joint. Attribute $\theta_i \in \mathbf{A}$ determines the initial position of the links connected with the corresponding joint and it can be chosen from a given set of possible values. All the angle values θ_i are defined relative to the position of the previous robot component. A list of the component-based rules in our framework is presented in Figure 4. Note that thanks to our grammar-based approach and the set of rules we propose, the list of components used for robot construction can be easily adapted. There is no requirement that structural rules be applied before component-based rules or vice versa.

Our recursive grammar is designed in a way that allows for a potentially infinite number of legs and body segments. In order to

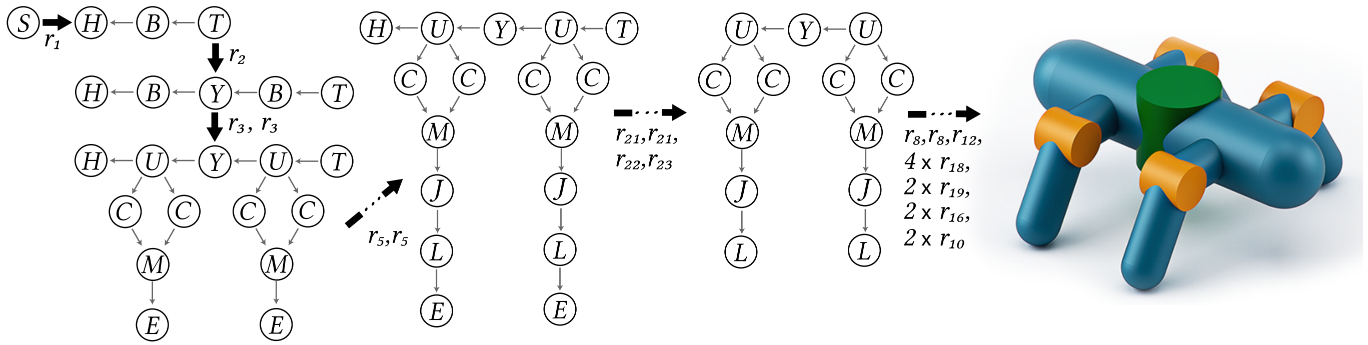

Fig. 5. A derivation sequence for a Simple Walker robot generated with our grammar. Derivation begins with the start symbol S, then creates the body and extends it with rules r_1,r_2 respectively. Legs are added on both body segments with rule r_3 applied twice. Both pairs of legs are extended, adding additional sets of joints and links, with r_5. For the Simple Walker, end structures are not used, hence we remove them with r_{21},r_{22}, and r_{23}. Finally, terminal components are added for each segment of the robot, following the rules from Figure 4.

limit the design space, our implementation uses a recursion counter. The recursion counter counts the total number of derivation steps. We set the maximum as 40 in our experiments. Increasing this parameter would allow for creation of more complex designs, while also increasing algorithm run time.

An example robot derived from grammar $\mathcal{G}$, along with the sequence of production rules applied, is shown in Figure 5. More interesting examples generated with our grammar are presented in Figure 6.

5 SIMULATION & CONTROL

Here, we describe our method for simulating and controlling generated designs. This is necessary both for co-optimizing designs and controllers, as well as evaluating the quality of generated designs.

5.1 Simulation

Robot performance on a given task is evaluated using rigid body dynamics simulation. The robots are modeled as articulated rigid bodies using an efficient recursive formulation based on Featherstone's algorithm [Featherstone 1983]. This recursive formulation reduces the complexity of the equations of motion, providing scalability and numerical stability. Collisions between the robot and the environment as well as between different rigid components of the robot (self-collision) are fully modeled. Graphs describing robots that self-collide in the initial configuration are discarded. Wheels are velocity-controlled, while all other joints are position-controlled. Our simulation is implemented using the Bullet Physics library [Coumans 2015], which is widely used in robotics.

5.2 Model Predictive Control

We generate optimized control inputs for each design using model predictive control, specifically the MPPI algorithm [Lowrey et al. 2018]. MPPI was chosen because it does not rely on derivatives and is relatively simple to implement. Our MPC implementation maintains a sliding window U of control inputs H time steps long, representing the "best" control inputs found so far. Each iteration begins by sampling K perturbed sequences of control inputs based on U, and applying each sequence of control inputs to a separate instance of the simulation. A new sequence U is then computed as a weighted sum of the perturbed sequences, with each sequence's weight depending on the discounted sum of rewards for that simulation. The first control input of U is appended to the optimized control sequence, which is the output of the algorithm. U is shifted forward one time step to prepare for the next iteration, filling with zeroes as necessary. The algorithm repeats until an optimized control sequence of length T (the episode length) is obtained. To take advantage of multiple CPU cores, we run the simulation instances in parallel using a thread pooling library.

Note that each MPC time step corresponds to multiple time steps in simulation, with the ratio being the *control interval.* Control inputs are repeated for the control interval, reducing the number of MPC time steps necessary. This reduces computation time significantly without noticeably degrading robot performance.

An outline of our MPPI implementation is given in Algorithm 1.

Algorithm 1 Model Predictive Control based on MPPI

Inputs: Episode length T, number of samples K, horizon H, initial state s_0, initial input sequence $U = [u_0, u_1, \ldots u_{H-1}]$, simulation dynamics f.
Output: The optimal control input sequence $[a_0, a_1, \ldots a_T]$.
for $t \leftarrow 0$ **to** $T-1$ **do**
  for $k \leftarrow 1$ **to** K **do**
    Sample input sequence U_k based on U.
    Apply inputs U_k starting at s_t, yielding return r_k.
  end for
  for $k \leftarrow 1$ **to** K **do**
    $w_k \leftarrow e^{\kappa(r_k - \max_l r_l)}$
  end for
  $U \leftarrow \sum_{k=1}^{K} U_k \frac{w_k}{\sum_{l=1}^{K} w_l}$
  $a_t \leftarrow u_0$
  $U \leftarrow [u_1, \ldots u_{H-1}, \mathbf{0}]$
  $s_{t+1} \leftarrow f(s_t, a_t)$
end for

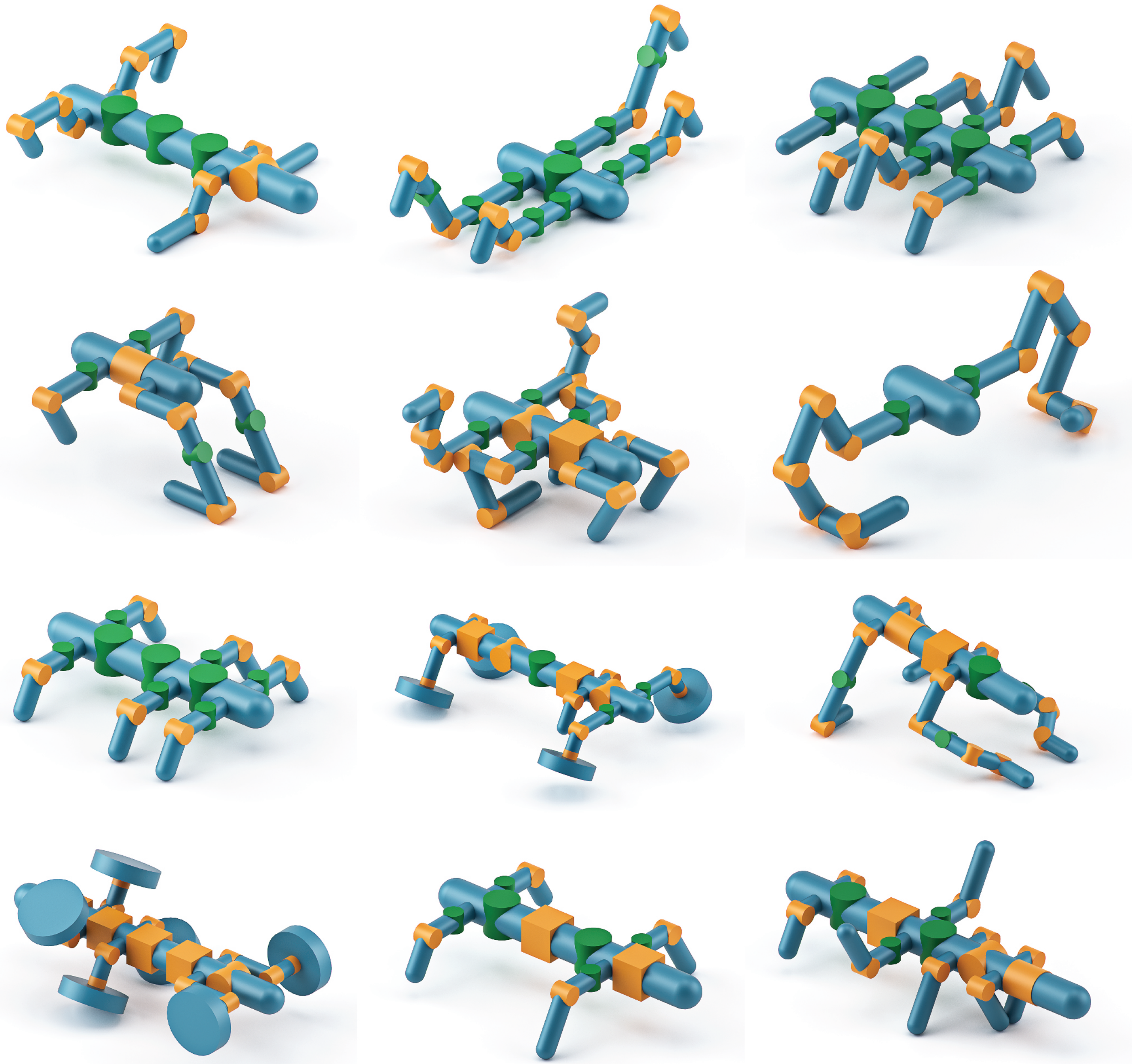

Fig. 6. Designs generated with *RoboGrammar* that appear during the search process for various terrains. Further optimization is necessary to identify the best performing ones for a given task.

5.2.1 Sampling. Careful design of the sampling distribution is key to achieving high MPC sample efficiency. Input sequences U_k are matrices where each entry is sampled from an independent normal distribution. The parameters of the distribution vary depending on the type of sample: *warm-start* or *history*.

Half of the samples are *warm-start* samples, whose distribution is centered on the input sequence U from the previous iteration shifted forwards by one step. Their standard deviation is σ_1.

As we expect many high-performing gaits to be periodic, the remaining samples are *history* samples. They are sampled from a normal distribution whose mean is the last M control inputs repeated

until the MPC horizon. M varies per sample, and ranges from $H/2$ to H. Their standard deviation is σ_2.

Combining the two different types of samples enables MPC to generate lifelike, periodic gaits while reacting to obstacles and changing terrain conditions.

6 SEARCH & OPTIMIZATION

Our grammar forms the front end of the *RoboGrammar* system, constraining the search space of all robotic structures to a tractable and meaningful subset. Next, we describe the search and optimization algorithms that efficiently search for high-performing designs and controllers, exploiting the reduced search space this grammar provides. Here, we present two search algorithms: our novel Graph Heuristic Search, and our variant of Monte Carlo tree search, which serves as a baseline.

6.1 Graph Heuristic Search

Our heuristic search algorithm is learning-based, using a learned heuristic function to inform and accelerate the search of the design space. This heuristic takes the form $V(g) : (\mathcal{V}, \mathcal{E}) \to \mathbb{R}$. The input to the function is a graph representing a partial robot design, where some nodes correspond to non-terminal symbols. The partial design may be expanded into one of many complete designs which have only terminal symbols. The function $V(g)$ aims to outputs the highest achievable performance across all of these complete designs. Our search algorithm is agnostic to the model used for the heuristic, and thus we describe it in general terms. In practice, we take a deep-learning-based approach and use graph neural networks to create our learnable heuristic.

6.1.1 Search Algorithm. Our Graph Heuristic Search algorithm works by interleaving a design phase (in which a candidate robot is sampled, guided by our heuristic function), an evaluation phase (in which the candidate robot is evaluated in simulation), and a learning phase (in which the heuristic function is improved based on the simulated data). These three phases are repeated over N episodes, or until they converge on an optimal design. The algorithm is described in Alg. 2.

Design Phase. During the design phase, K possible candidate robot designs are generated and one of them is selected for evaluation. Each design is generated by the following procedure. Starting from a partial robot design s_0, composed solely of the initial start symbol ($s_0 := S$), production rules of the grammar are iteratively applied to the partial robot design until it contains only terminal symbols. The selection of production rules is inspired by Q-learning [Norvig and Russell 2002] and follows an ϵ-greedy approach. Given a partial robot design s_l after l production rules have been applied, the $l+1^{\text{th}}$ rule a_{l+1} is selected as follows. With probability ϵ, a random rule is applied from the set of possible rules. Otherwise, with probability $(1-\epsilon)$, the rule that leads to the design with the highest heuristic score is applied, *i.e.* $a_{l+1} \leftarrow \arg\max_a V_\theta(P(s_l, a))$, where $P(\mathcal{S}, \mathcal{A})$ is a function which applies production rule $\mathcal{A}$ to partial design $\mathcal{S}$. Once a candidate design with only terminal symbols is produced, it is added to the list of possible candidate designs. From the final list of K candidates, a random robot is selected with probability ϵ; with probability $1-\epsilon$, the design with the highest heuristic score is chosen as the candidate to continue to the evaluation phase.

Algorithm 2 Graph Heuristic Search

```
Inputs: Number of iterations N, number of candidate designs M, Adam
optimization steps opt_iter and batch size M.
Output: The best design s*.
Initialize the look up table V̂ ← {}.
Initialize the graph neural network V_θ(s) with random parameters θ.
Initialize the best design s* ← None, r* ← 0.
for episode j ← 1 to N do
  ▷ Design Phase: Generate a candidate design
  𝒫 ← {}                          ▷ Initialize possible design candidates
  ▷ Sample K designs by ε-greedy approach
  for k ← 1 to K do
    s ← initial design graph
    while s has non-terminals do
      With probability ε select a random rule a from available rules
      otherwise select a = argmax_a V_θ(P(s, a)).
      s ← P(s, a)
    end while
    Add possible candidate s to 𝒫.
  end for
  ▷ Choose one to be the candidate
  With probability ε select a random sampled design from 𝒫 as the can-
  didate design d, otherwise select d = argmax_{d∈𝒫} V_θ(d) by heuristic
  function V_θ.
  ▷ Evaluation Phase: Compute the average reward for the design
  Run MPC to evaluate d and get average reward r.
  ▷ Update the best design and V̂
  if r > r* then
    s* ← d
    r* ← r
  end if
  for Each partial ancestor design d_p of d do
    Update V̂(d_p) ← max(V̂(d_p), r).
  end for
  ▷ Learning Phase: train heuristic value function V_θ
  for i ← 1 to opt_iter do
    Sample a minibatch S of seen designs (partial or complete) of size
    M.
    Update V_θ(s) one step by Adam with the loss:
        Σ_{s∈S} ||V_θ(s) − V̂(s)||²
  end for
end for
```

Two ϵ-greedy selection steps are applied during candidate robot generation. This strategy is necessary to ensure that the space of possible robot designs is sufficiently explored; if one begins with a pure greedy strategy, the algorithm quickly converges to a suboptimal design. This is because we are taking a learning-based approach; our heuristic function is inaccurate at the beginning (and not strictly admissible) and improves in accuracy as the algorithm progresses. Until the heuristic function converges to an accurate estimator mapping robot design to performance, it is necessary to generate a diverse collection of robot designs from which to learn from (beginning with $\epsilon = 1$). The first ϵ-greedy exploration rule, within a given design generation, helps guarantee a diverse collection of possible

candidate designs (with the variance of that set parameterized by ϵ). Since K applications of this process (with large K and small ϵ) makes it likely that at least one robot resembling the pure greedy-strategy will be generated, the second ϵ-greedy exploration guarantees that the same best candidate is not chosen each time. ϵ is decreased with each episode toward 0 as the heuristic's accuracy increases, according to an exponential decay schedule (as in Q-learning); this is made possible by a fast, accurate learned heuristic function, and shifts the algorithm from exploration to exploitation.

Evaluation Phase. After a candidate robot has been decided on, its performance must be evaluated. We simulate the candidate robot with actuation inputs generated by the MPC algorithm described in Section 5.2. It is possible for the same design candidate to be proposed multiple times. Because our MPC algorithm is sampling-based and stochastic, different average rewards $\hat{V}$ may be seen for the same design between episodes. We consider the $\hat{V}$ of a design to be the best average reward over all evaluations of the design. If this average reward is the best seen so far, the candidate is stored as the current best design, along with $\hat{V}$. Regardless, the candidate robot design and its corresponding $\hat{V}$ are stored (or updated) in a lookup table, and the $\hat{v}$ label of all of that design's partial design ancestors are updated to be the maximum of their current value and the candidate robot's average reward. This is important for the upcoming learning phase, which must learn a heuristic function that is valid for both complete and partial designs.

The number of candidate designs evaluated in each iteration is an algorithm design trade-off. Evaluating more candidates will collect more data, helping to train a more accurate prediction function. It will also significantly increase the computation time, however, since evaluation is the time bottleneck in our algorithm. We therefore choose to evaluate only one design per iteration.

Learning Phase. The heuristic is trained using the data stored in the lookup table. For `opt_iter` epochs, minibatches of $(s_i, \hat{V}_i)$ pairs are sampled, and the loss $L = \frac{1}{2}\|V_\theta(s_i) - \hat{V}\|_2^2$ is minimized using Adam [Kingma and Ba 2015].

6.1.2 Heuristic Function Model. To implement our heuristic function, we choose to leverage the expressive nature of graph neural networks. Graph neural networks (GNNs) are neural network architectures which aim to extend the benefits of deep learning to a graphical setting. Unlike other neural network types such as CNNs, which operate on images and data with fixed grid-like topologies, graph neural networks aim to be flexible and operate on structures with arbitrary topologies. The input to a GNN consists of a graph topology (*e.g.* an adjacency matrix), and values associated with nodes (*e.g.* feature vectors). While many GNN models have been proposed in recent years, our architecture is based on the differentiable-pooling model. This model was designed for inference tasks involving graphs with a *hierarchical nature*, by iteratively reducing the graph to a "lower resolution" graph in a manner similar to hierarchical clustering. Please see [Ying et al. 2018] for more details. This model is well-suited for our scenario, where each robot is itself created through a hierarchical substitution of grammar rules.

The differentiable-pooling GNN extends the GraphSage framework from [Hamilton et al. 2017], which in turn is based on graph convolutional networks (GCN) [Kipf and Welling 2016]. Analogously to CNNs, GCNs apply a generalized convolution operator that operates on graphs rather than grids. We adopt a similar model as [Ying et al. 2018]. In this model, two "mean" GraphSage+BatchNormalization layers are applied, followed by the hierarchical clustering DiffPool layer, followed by three layers of graph convolutions. This process is repeated one additional time, followed by a final GraphSage layer, a mean pooling layer, and a final ReLU. The output is a positive real-valued scalar representing predicted robot performance. Each DiffPool layer reduces the node set's cardinality by 75%.

An important property of this GNN model is that it is isomorphism-invariant, meaning any two isomorphic graphs will have the same value and gradient without the need for explicit transposition tables. This greatly simplifies the bookkeeping in Graph Heuristic Search.

We convert robot design graphs into inputs for the GNN model as follows. The graph is first converted to a kinematic tree, so that each link has a unique joint connecting it to its parent link. Each link and its parent joint is considered a node in this new graph. Note that this representation differs from the one described in Section 4. Next, an m-dimensional feature vector is extracted from each node. If the link associated with the node is a terminal symbol, the feature vector encodes the link's initial position, orientation, and geometric description. The parent joint's rotation and servo parameters are included similarly, if present. If either the link or joint are non-terminal symbols, the feature vector one-hot encodes the non-terminal type.

6.2 Monte Carlo Tree Search

For comparison, we also implement Monte Carlo tree search (MCTS), specifically the UCT algorithm [Kocsis and Szepesvári 2006]. An outline of our MCTS implementation is given in Algorithm 3. The state of the algorithm is represented by a directed acyclic graph of nodes representing partial designs and associated statistics. Directed edges between the nodes represent actions, or rule applications. MCTS implementations tend to differ in which statistics they store. Because our problem setting is single-player and nonadversarial, we choose to store the visit count and maximum reward. Like Graph Heuristic Search, each iteration of our MCTS implementation incorporates design, evaluation, and learning phases.

Our design phase combines the selection and expansion steps of the standard MCTS iteration. The selection step begins at the root of the tree, representing the start symbol of the grammar. Edges are followed repeatedly until a node with no children (a leaf node) is reached. When multiple edges are available, the one with the highest UCT score is chosen:

$$\arg\max_{a \in A(s)} \left(Q_{s,a}(t) + \sqrt{\frac{2 \ln N_s(t)}{N_{s,a}(t)}} \right) \quad (1)$$

where $Q_{s,a}(t)$ is the maximum result from all iterations where rule a was applied to graph s, $N_s(t)$ is the visit count of s, and $N_{s,a}(t)$ is the number of iterations where a was applied to s. If the last node reached represents a partial design, an expansion step is performed: a randomly selected rule is applied and the node for the resulting design added to the tree. Without further modifying the tree, rules

Algorithm 3 Monte Carlo Tree Search

Inputs: Number of iterations N, maximum number of simulation attempts M.
for $i \leftarrow 1$ **to** N **do**
  Select node s using UCT formula.
  Expand node s.
  $j \leftarrow 0$ ▷ Number of attempted playouts
  repeat
    if $j = M$ **then**
      Block node s. ▷ Do not allow s to be selected again
      Select new node s using UCT formula.
      Expand new node s.
      $j \leftarrow 0$
    end if
    Starting with graph in s, apply randomly sampled rules until only terminal symbols are left.
    $j \leftarrow j + 1$
  until graph is simulable
  Run MPC (Algorithm 1) to obtain result (average reward).
  Update all nodes from s to the root with the result.
end for

are then randomly selected and applied until a complete design is obtained. The complete design is evaluated in simulation using MPC, in a way similar to Graph Heuristic Search.

In the learning phase, the statistics of nodes in the search tree that were visited during the design phase are updated. Each visited node's visit count is incremented, and its maximum reward is replaced with the evaluation result if it is higher. These updated statistics will guide node selection in the next iteration.

In addition to the basic UCT algorithm described above, we implement several common enhancements which promote faster convergence. To address the possibility of designs that are not simulable due to self-collision, we apply rejection sampling. These modifications are described in the following paragraphs.

Transpositions. Some designs can be reached through multiple different rule sequences (transpositions). To improve efficiency, only one node is created per unique design. Statistics from all iterations that visited a node, regardless of the exact sequence of rules, are aggregated together. Childs et al. [2008] show that this scheme can improve efficiency compared to the basic UCT algorithm for games with many transpositions. An approximate hashing-based scheme is used to detect isomorphic designs with a low probability of false positives.

UCT-RAVE. Certain rules may tend to improve performance no matter when they are applied. The UCT-RAVE algorithm [Gelly and Silver 2011] takes this insight into account through the all-moves-as-first (AMAF) heuristic [Helmbold and Parker-Wood 2009]. AMAF estimates the value of an action at a node based on iterations that visited the node, like vanilla UCT. However, AMAF also uses iterations where the action is taken anytime in the future, instead of immediately at the node. UCT-RAVE blends between the vanilla Monte Carlo value of an action and the AMAF value of an action according to a schedule. As the visit count of a node increases, the AMAF value is weighted less heavily.

Rejection Sampling. A simulation may fail to produce a valid result for a number of reasons, including self-collision in the starting configuration. In that case, the search algorithm tries another random playout from the same leaf node. Note that simply returning a fixed, low result unnecessarily biases the search towards simpler designs which are less likely to self-collide.

There may be partial designs that no sequence of rule applications can convert into simulable designs. All playouts starting from the corresponding nodes will fail to produce a valid result. We address this potential issue by limiting the number of playout attempts for a single node. Upon reaching the limit (set to 100), the node is *blocked* and can no longer be selected. The iteration restarts at the selection step and a different leaf node is expanded.

7 PROBLEM SPECIFICATION

Here, we describe the remaining (user-specified) components needed to define a co-optimization problem with RoboGrammar.

7.1 Reward Function

A single reward function (Equation 2) is used to evaluate designs on every terrain, and is computed at every time step of the simulation. For the purposes of design optimization we use the average reward across all time steps.

$$r(t) = \vec{w_x} \cdot \vec{d_x}(t) + \vec{w_y} \cdot \vec{d_y}(t) + \vec{w_v} \cdot \vec{v}(t) \tag{2}$$

We consider the base link of the robot, or the forwardmost wide body segment, to be representative of the robot's motion. $\vec{d_x}$ and $\vec{d_y}$ are unit vectors pointing forward and upward in the base link's reference frame, respectively, and $\vec{v}$ is the base link's velocity. All quantities are expressed in world coordinates.

$\vec{w_x} = [-2, 0, 0]^T$, $\vec{w_y} = [0, 2, 0]^T$, and $\vec{w_v} = [2, 0, 0]^T$ are weighting vectors which set the relative importance of each term. They also scale the reward function's magnitude to the range expected by the design search algorithm.

The first two terms encourage maintaining the initial orientation, and the last term rewards forward progress. The robot starts with its local x-axis pointing in the negative x direction in world coordinates.

7.2 Terrains

Terrains, in conjunction with the reward function, define tasks to optimize robot structures for. Each terrain is intended to result in a different set of optimal designs.

Flat terrain. A featureless surface with a friction coefficient of 0.9, the flat terrain accommodates the greatest variety of locomotion styles.

Frozen lake terrain. A flat surface with a low friction coefficient of 0.05, the frozen lake terrain encourages designs which maximize traction or use the low friction to their advantage.

Ridged terrain. Ridges or hurdles spaced an average of one meter apart span the entire width of the ridged terrain, requiring designs to jump or crawl in order to make progress.

Table 1. Graph Heuristic Search hyperparameter values

Hyperparameter	Value
Number of iterations (N)	2000
Initial ϵ (ϵ_0)	1.0
Final ϵ (ϵ_1)	0.1
ϵ exponential decaying factor (ϵ_{decay})	0.3
Number of possible candidate robots (K)	16
Optimization steps (opt_iter)	25
Optimization batch size (M)	32
Adam learning rate	1×10^{-4}

Table 2. Simulation and MPC hyperparameter values

Hyperparameter	Value
Simulation time step (dt)	1/240 s
Default coefficient of friction	0.9
Joint torque limit	1 Nm
Control interval	16 time steps
Episode length (T)	128 control intervals
Number of samples (K)	64
MPC horizon, default (H)	16 control intervals
MPC horizon, wall terrain (H)	32 control intervals
Sample standard deviation, *warm-start* (σ_1)	0.25
Sample standard deviation, *history* (σ_2)	0.05

Wall terrain. Walls which are too high to traverse directly are placed in a slalom-like arrangement. Designs must move around the walls, requiring them to change their direction of motion rapidly.

Gap terrain. A series of platforms separated by gaps require designs to tread carefully. As the gaps become progressively wider, designs with the ability to take larger steps are favored.

Upward stepped terrain. A series of steps resembling a flight of stairs test the ability of designs to climb. The steps are of varying height, producing different gait variations over time.

8 IMPLEMENTATION

Before presenting our results, we briefly describe important details regarding the implementation of *RoboGrammar*, including hyperparameter choices and experimental setup.

8.1 Hyperparameters

We run all experiments with the same hyperparameters unless otherwise specified. The exploration parameter ϵ in GHS follows an exponential decay schedule: $\epsilon(i) = \epsilon_1 + (\epsilon_0 - \epsilon_1)\exp(-\frac{i/N}{\epsilon_{decay}})$, where i is the current iteration, N is the total number of iterations, $\epsilon_0 = 1$, $\epsilon_1 = 0.1$ and $\epsilon_{decay} = 0.3$. We run GHS for 2,000 iterations ($N = 2000$). In the design phase of each iteration, GHS uses the two ϵ-greedy steps to select one design to be tested by MPC from 16 sampled possible candidate robots. In the learning phase, the Adam optimizer runs for 25 steps with batch size 32 and learning rate 1×10^{-4}.

Hyperparameters of the simulation and MPC are either chosen based on our problem setting or are manually tuned. We use the default time step of 1/240 s for Bullet Physics, which is the recommended value for robotics applications. The default friction coefficient of 0.9 approximates that of rubber on pavement. Joint torques are limited to one Newton-meter, which is within the capability of affordable hobby servos. The MPC control interval, sample count, and horizon are tuned to produce consistently high rewards while limiting computation time. Computation time increases with control interval, sample count, and horizon. Sample standard deviations are tuned to balance progress on terrains with smooth motion. Higher standard deviations result in faster but more erratic motion. A summary of hyperparameters is provided in Table 1 and Table 2.

8.2 Experiment Setup & Computational Time

We implemented our Graph Heuristic Search algorithm in Python, and the simulation and MPC in C++. Experiments were run on VM instances with either 32 or 64 Intel Cascade Lake vCPUs on Google Cloud Platform. Each iteration of GHS spends less than 1 second per possible candidate robot in the design phase, 40-60 seconds in the MPC evaluation phase, and 6-8 seconds in the learning phase. Since each possible candidate robot is sampled independently, the design phase can be fully parallelized for further speedup. The time bottleneck of the MPC evaluation phase shows the necessity of our Graph Heuristic Search algorithm, which is able to find the best-performing robots while evaluating a significantly fewer number of robot designs. Section 9.4 describes this efficiency in detail.

Our two baseline search algorithms, MCTS and random search, are also implemented in Python. Both baselines also spend the vast majority of their computation time on MPC, with their design and learning phases (if applicable) taking a negligible amount of time.

9 RESULTS

Here we demonstrate *RoboGrammar* on a collection of different problems, examining how terrain affects which designs and controllers are optimal. Furthermore, we examine how designs translate over multiple terrains, analyzing the Pareto set of synthesized controllers over terrain pairs. We qualitatively demonstrate the efficacy of MPC for our problems and quantitatively analyze the efficiency of our Graph Heuristic Search algorithm compared to the baselines.

9.1 Terrain Driven Optimization

As an end-to-end demonstration of the *RoboGrammar* pipeline, we run Graph Heuristic Search on several different terrains. Each search run consists of 2,000 iterations. A selection of best-performing designs is shown in Figure 7.

Optimal designs for the ridged terrain are characterized by long limbs which are able to swing upwards and clear obstacles. Although the set of optimal designs consists mainly of quadrupeds, a few tripedal designs emerge. These designs use their body as a third point of contact with the ground.

The flat terrain, despite having no obstacles, still produces specialized designs. One successful strategy is to place short limbs spaced far apart on the body, giving them a full range of motion. Although

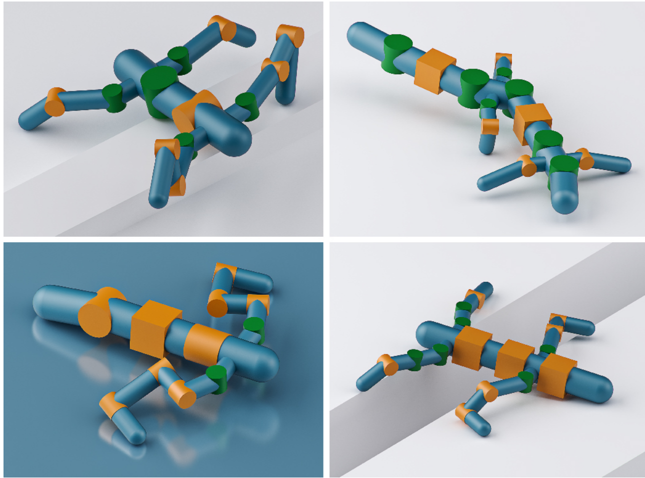

Fig. 7. Selection of best-performing designs generated with the grammar, and optimized with Graph Heuristic Search and MPC for ridged, flat, frozen lake, and gapped terrain respectively.

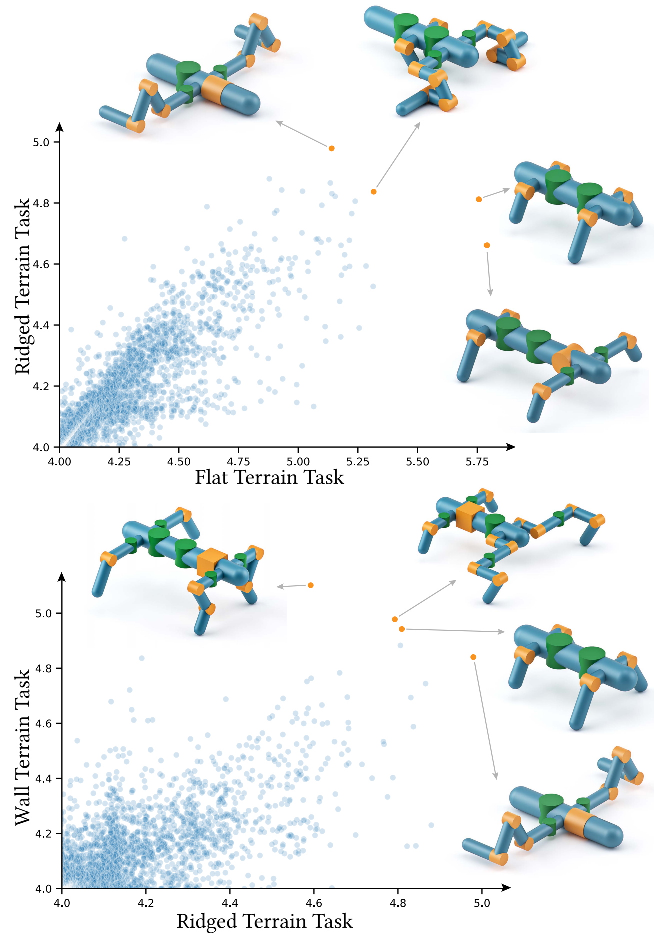

Fig. 8. Pareto optimal designs for pairs of terrains. Designs offering a variety of trade-offs are discovered using only 20,000 iterations of random search, demonstrating the versatile yet compact search space of our grammar.

short limbs would be unable to clear obstacles on most of the other terrains, their low inertia enables quick movement.

The frozen lake is superficially similar to the flat terrain, but its low friction coefficient requires a different strategy. Successful designs can both overcome the low friction and use it to their advantage. The bottom-left design in Figure 7 serves as an example. Highly articulated yet compact arms maintain contact with the ground during the stance phase, while the rear body segment slides freely.

The gap terrain tends to produce designs with long limbs, much like the ridged terrain. However, designs for gap terrain tend to have limbs that are optimized for forward reaching instead of climbing. Green joints, which enable limbs to move horizontally, are more prevalent than orange joints, which enable vertical motion.

9.2 Pareto Analysis

In addition to single terrain optimization, we also show how our grammar can help in discovering high-performing robots for multiple terrains simultaneously. We choose to use random search for this experiment to avoid biasing the search towards a particular objective. Each random design is evaluated on flat, ridged, and wall terrain. The average reward of designs on two pairs of tasks is plotted in Figure 8, with the Pareto optimal designs highlighted. The Pareto sets include a variety of morphologies, each of which offers an optimal trade-off, showing that our grammar is highly expressive. Note that these results were obtained after evaluating only 20,000 random designs. This evaluation count is comparable to evolutionary robot design methods operating on a single objective, showing that our grammar also has a compact search space.

9.3 Gait Patterns

In Figure 9, we demonstrate that our MPC implementation can generate plausible gaits for challenging terrains. Time-lapses of high performing designs traversing the upward stepped terrain and wall terrain are shown. The robot on the upward stepped terrain initially adopts a cyclic trotting gait. Most steps are low, requiring only one of the front legs and one of the back legs to make contact. Some of the steps are higher, causing MPC to switch to an ad-hoc gait as necessary. The wall terrain-optimized robot also employs a trotting gait, but with exaggerated body movements. Multiple joints in the body allow the robot to curve sharply and change direction rapidly. This can be seen in the last frame of the time-lapse, where the robot turns sharply to the left in order to avoid a wall.

9.4 Efficiency of Graph Heuristic Search

To show the efficiency of the proposed Graph Heuristic Search algorithm, we compare our algorithm with two baselines on four different terrain tasks (flat terrain, frozen lake terrain, ridged terrain, and wall terrain) described in Section 7.2. Specifically, the first baseline is an adapted Monte Carlo tree search (MCTS) algorithm described in Section 6.2. The second baseline algorithm is a random search algorithm. In each iteration, one candidate design is selected

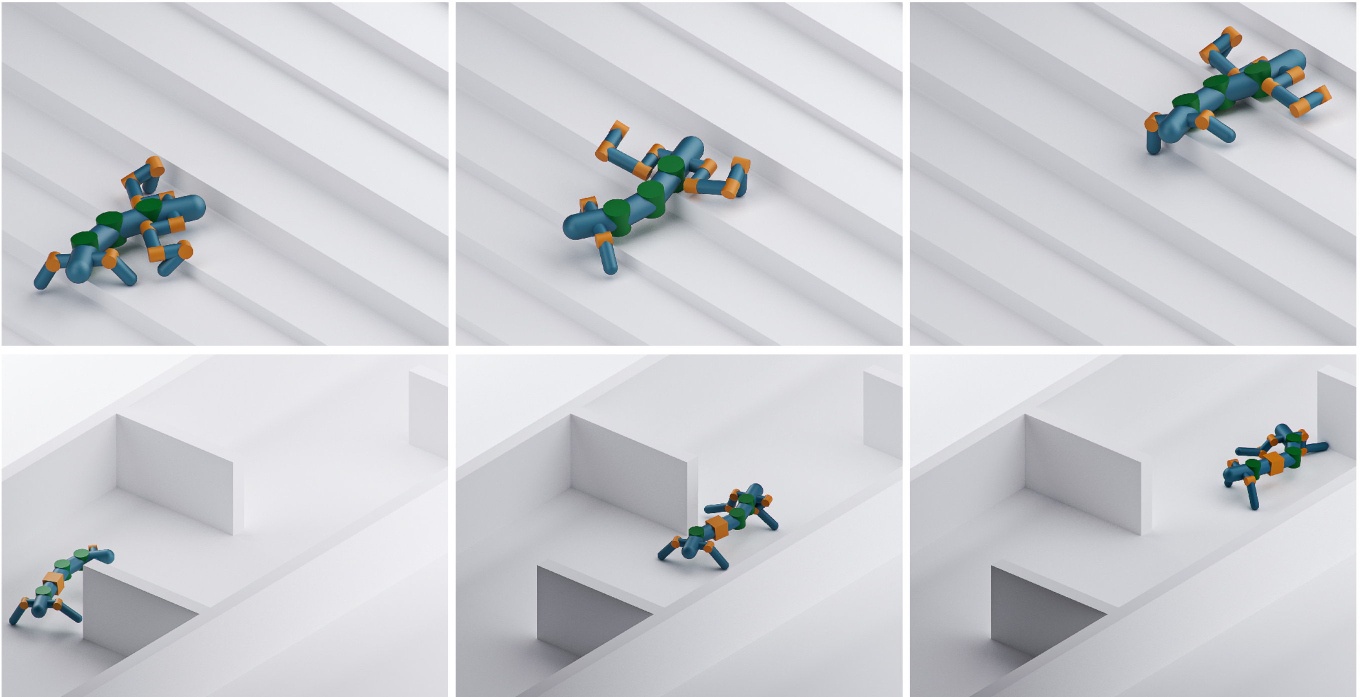

Fig. 9. Time-lapse showing movement of the high performing robots on upward stepped terrain and terrain with wall obstacles. Each front leg of the first robot consists of 3 elbow joints and a final knee joint forming a hook-like ending to facilitate climbing high steps. The second robot has a long body with many rotational joints and front legs designed for turning stability.

by applying random rules. Due to the stochasticity of the search algorithms, we run each algorithm on each task at least three times with different random seeds. Note that we run our Graph Heuristic Search algorithm for only 2,000 iterations, as opposed to 5,000 iterations for each baseline algorithm. The results in Figure 10 show that Graph Heuristic Search consistently finds better designs (achieves greater reward) than the baseline algorithms in much fewer iterations. Due to the expensive MPC-based evaluation of designs and the combinatorial nature of our design space, this sample efficiency is key to finding high-performing designs in a reasonable amount of time. The efficiency comes from the ability of the learned heuristic function to generalize knowledge from explored designs to predict the performance of untested designs and effectively prune the search space.

Running 2,000 iterations of Graph Heuristic Search requires approximately 31 hours on a 32-core Google Cloud machine (instance type `n2-highcpu-32`). This computational intensity is comparable to other state-of-the-art robot design methods. For example, Neural Graph Evolution [Wang et al. 2019b] is evaluated using 12 hours on a 64-core machine. Note that we simulate designs with greater numbers of joints on more complex terrains, resulting in more expensive evaluations. Evaluations account for approximately 20 hours out of our total run time.

9.5 Convergence of Graph Heuristic Search

To demonstrate that Graph Heuristic Search is agnostic to the specific grammar described (the *standard* grammar), we optimize robots for flat terrain using two modified grammars. The *simple* grammar removes the rules for long limb links and elbow joints (r_9 and r_{17} in Figure 4 respectively), whose functionality is provided by other rules. The *asymmetric* grammar increases complexity by allowing opposite limbs to develop independently. Figure 11 shows that Graph Heuristic Search consistently converges in training loss, prediction error, and maximum cumulative reward. More complex grammars require a greater number of iterations to achieve the same reward.

9.6 Design Space Bias

MCTS and GHS both rely on pruning the search space, so some degree of bias can be expected in the resulting designs. In each given state, MCTS explores every available action once before exploiting a promising branch. We therefore expect the average number of derivation steps in designs explored by MCTS to be low. The statistics in Table 3 reflect this hypothesis, with designs found by MCTS having fewer derivation steps than GHS on average. When considering all designs explored, random search has the lowest average derivation length. This reflects the fact that random search does not focus its exploration and will visit the same (simple) designs repeatedly.

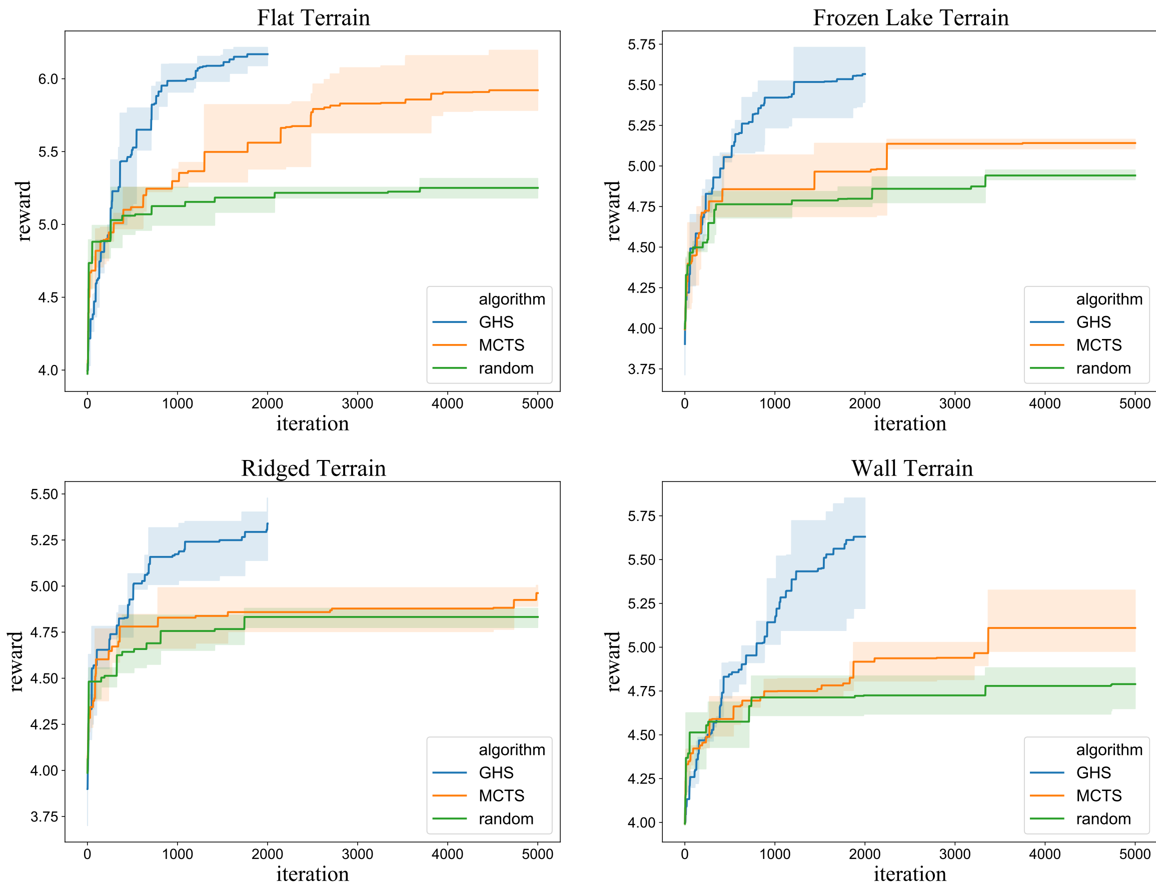

Fig. 10. Cumulative maximum reward versus iteration for Graph Heuristic Search, Monte Carlo tree search, and random search on four different terrains. Each solid line is the mean of three different seeds, with the error band representing the range. Graph Heuristic Search consistently outperforms the baselines.

Table 3. Average number of derivation steps in designs evaluated by each algorithm. Results are for flat terrain.

Algorithm	All designs	Best 100 designs
GHS	24.4	25.0
MCTS	24.0	20.9
random	11.4	23.4

10 LIMITATIONS AND FUTURE WORK

The example grammar presented in this work only considers bilateral symmetry and joint types that occur in natural systems. To explore designs outside this scope — for example, sea creatures — the grammar from Section 4 could be extended. The grammar has been designed to be easily expandable, and we leave this exploration as an interesting direction for future work.

Model predictive control is able to produce stable gaits across a wide variety of terrains and robot topologies. Dynamic gaits may also exist, but they are unlikely to be found. This is due to the bias introduced by our sampling scheme, as well as the limited torque and high damping of the simulated motors.

Our framework ensures that simulated designs are at least fabricable, or able to be built in their simulated configurations. With the appropriate rules, robot grammars enforce the use of a limited set of components and ensure that the components are connected in a feasible manner. Providing an accurate match between simulated and physical motions is not explicitly addressed in this work, however, and it remains an interesting direction for future research.

Our grammar does not include continuous parameters, because we are focused on optimizing discrete topology. Our framework could easily be extended to handle continuous parameters using attribute grammars, or in a post-processing step after the discrete robot structure is identified.

11 CONCLUSION

Intelligent and efficient generative robot design methods will drive the future of design processes for robotics. In this paper we present a novel approach which leverages several different computational tools to provide a new pipeline for efficient and large-scale exploration of robot designs. By developing a graph grammar that allows for a wide variety of robots to be generated, we show that creative

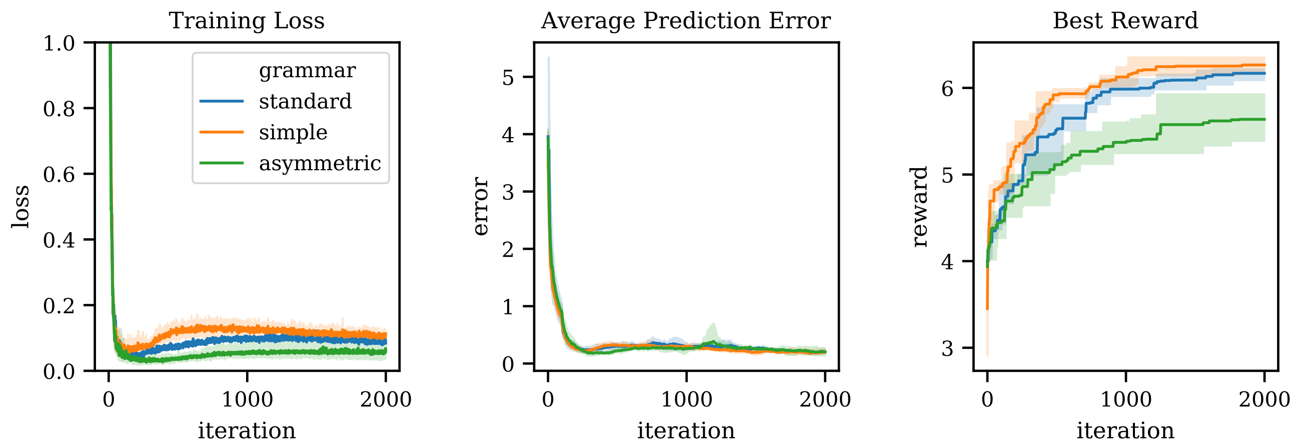

Fig. 11. Training loss, prediction error, and cumulative maximum reward versus iteration for Graph Heuristic Search on multiple grammars. Robots are optimized for flat terrain. Prediction error is the absolute difference between predicted reward and evaluated reward, and is averaged over 100 iterations. Each solid line is the mean of at least three different seeds, with the error band representing the range. Graph Heuristic Search consistently converges in all three criteria.

solutions can emerge in response to different terrains. We introduce a Graph Heuristic Search algorithm to search the combinatorial search space, and couple it with MPC for evaluation. Unlike many alternative approaches to generative robot design, this allows us to structure and limit the design space by applying a graph grammar, whilst allowing creative solutions to emerge. Importantly, the emergent designs are observably physically fabricable and there is significant potential for the designs to be translated to real-world scenarios and environments.

ACKNOWLEDGMENTS

We are grateful to anonymous reviewers for their valuable feedback. This work was supported by IARPA grant no. 2019-19020100001 and NSF grant no. 1644558. M.K.L. would like to acknowledge support from the Schmidt Science Fellowship.

REFERENCES

ZM Bi and Wen-Jun Zhang. 2001. Concurrent optimal design of modular robotic configuration. *Journal of Robotic systems* 18, 2 (2001), 77–87.
Martin Bokeloh, Michael Wand, and Hans-Peter Seidel. 2010. A Connection between Partial Symmetry and Inverse Procedural Modeling. *ACM Trans. Graph.* 29, 4, Article Article 104 (July 2010), 10 pages. https://doi.org/10.1145/1778765.1778841
Josh C Bongard. 2013. Evolutionary robotics. *Commun. ACM* 56, 8 (2013), 74–83.
Luzius Brodbeck, Simon Hauser, and Fumiya Iida. 2015. Morphological evolution of physical robots through model-free phenotype development. *PloS one* 10, 6 (2015), e0128444.
Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4, 1 (2012), 1–43.
I-Ming Chen and Joel W Burdick. 1995. Determining task optimal modular robot assembly configurations. In *proceedings of 1995 IEEE International Conference on Robotics and Automation*, Vol. 1. IEEE, 132–137.
Benjamin E Childs, James H Brodeur, and Levente Kocsis. 2008. Transpositions and move groups in Monte Carlo tree search. In *2008 IEEE Symposium On Computational Intelligence and Games*. IEEE, 389–395.
Noam Chomsky. 1956. Three models for the description of language. *IRE Transactions on Information Theory* 2, 3 (Sep. 1956), 113–124. https://doi.org/10.1109/TIT.1956.1056813
Erwin Coumans. 2015. Bullet physics simulation. In *ACM SIGGRAPH 2015 Courses*. ACM, 7.
Minh Dang, Stefan Lienhard, Duygu Ceylan, Boris Neubert, Peter Wonka, and Mark Pauly. 2015. Interactive Design of Probability Density Functions for Shape Grammars. *ACM Trans. Graph.* 34, 6, Article Article 206 (Oct. 2015), 13 pages. https://doi.org/10.1145/2816795.2818069
Frances Downing and Ulrich Flemming. 1981. The bungalows of Buffalo. *Environment and Planning B: Planning and Design* 8, 3 (1981), 269–293.
José Pinto Duarte. 2005. Towards the Mass Customization of Housing: The Grammar of Siza's Houses at Malagueira. *Environment and Planning B: Planning and Design* 32, 3 (2005), 347–380. https://doi.org/10.1068/b31124
Roy Featherstone. 1983. The calculation of robot dynamics using articulated-body inertias. *The International Journal of Robotics Research* 2, 1 (1983), 13–30.
Carlos E Garcia, David M Prett, and Manfred Morari. 1989. Model predictive control: theory and practice—a survey. *Automatica* 25, 3 (1989), 335–348.
Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michele Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. 2012. The grand challenge of computer Go: Monte Carlo tree search and extensions. *Commun. ACM* 55, 3 (2012), 106–113.
Sylvain Gelly and David Silver. 2011. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence* 175, 11 (2011), 1856–1875.
David Ha. 2018. Reinforcement learning for improving agent design. *arXiv preprint arXiv:1810.03779* (2018).
Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Advances in neural information processing systems*. 1024–1034.
David P Helmbold and Aleatha Parker-Wood. 2009. All-Moves-As-First Heuristics in Monte-Carlo Go.. In *IC-AI*. 605–610.
Jonathan Hiller and Hod Lipson. 2011. Automatic design and manufacture of soft robots. *IEEE Transactions on Robotics* 28, 2 (2011), 457–466.
Gregory S Hornby and Jordan B Pollack. 2001. The advantages of generative grammatical encodings for physical design. In *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No. 01TH8546)*, Vol. 1. IEEE, 600–607.
Diego Jesus, António Coelho, and António Augusto Sousa. 2016. Layered Shape Grammars for Procedural Modelling of Buildings. *Vis. Comput.* 32, 6–8 (June 2016), 933–943. https://doi.org/10.1007/s00371-016-1254-8
Gangyuan Jing, Tarik Tosun, Mark Yim, and Hadas Kress-Gazit. 2018. Accomplishing high-level tasks with modular robots. *Autonomous Robots* 42, 7 (2018), 1337–1354.
Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6980

Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

Gregor Klančar and Igor Škrjanc. 2007. Tracking-error model-based predictive control for mobile robots in real time. *Robotics and autonomous systems* 55, 6 (2007), 460–469.

Eric Klavins, Robert Ghrist, and David Lipsky. 2004. Graph grammars for self assembling robotic systems. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*, Vol. 5. IEEE, 5293–5300.

Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *European conference on machine learning*. Springer, 282–293.

John R Koza. 1995. Survey of genetic algorithms and genetic programming. In *Wescon conference record*. Western Periodicals Company, 589–594.

Lars Krecklau, Darko Pavic, and Leif Kobbelt. 2010. Generalized Use of Non-Terminal Symbols for Procedural Modeling. *Computer Graphics Forum* 29, 8 (2010), 2291–2303. https://doi.org/10.1111/j.1467-8659.2010.01714.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2010.01714.x

Felipe Kuhne, Walter Fetter Lages, and J Gomes da Silva Jr. 2004. Model predictive control of a mobile robot using linearization. In *Proceedings of mechatronics and robotics*. Citeseer, 525–530.

Manfred Lau, Akira Ohgawara, Jun Mitani, and Takeo Igarashi. 2011. Converting 3D furniture models to fabricatable parts and connectors. In *ACM Transactions on Graphics (TOG)*, Vol. 30. ACM, 85.

Gil Lederman, Markus N Rabe, Edward A Lee, and Sanjit A Seshia. 2018. Learning heuristics for automated reasoning through deep reinforcement learning. *arXiv preprint arXiv:1807.08058* (2018).

Markus Lipp, Peter Wonka, and Michael Wimmer. 2008. Interactive Visual Editing of Grammars for Procedural Architecture. In *ACM SIGGRAPH 2008 Papers (SIGGRAPH '08)*. Association for Computing Machinery, New York, NY, USA, Article Article 102, 10 pages. https://doi.org/10.1145/1399504.1360701

Tianqiang Liu, Siddhartha Chaudhuri, Vladimir G. Kim, Qixing Huang, Niloy J. Mitra, and Thomas Funkhouser. 2014. Creating Consistent Scene Graphs Using a Probabilistic Grammar. *ACM Trans. Graph.* 33, 6, Article Article 211 (Nov. 2014), 12 pages. https://doi.org/10.1145/2661229.2661243

Kendall Lowrey, Aravind Rajeswaran, Sham Kakade, Emanuel Todorov, and Igor Mordatch. 2018. Plan online, learn offline: Efficient learning and exploration via model-based control. *arXiv preprint arXiv:1811.01848* (2018).

Jon McCormack, Alan Dorin, Troy Innocent, et al. 2004. Generative design: a paradigm for design research. *Proceedings of Futureground, Design Research Society, Melbourne* (2004).

Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. 2006. Procedural Modeling of Buildings. *ACM Trans. Graph.* 25, 3 (July 2006), 614–623. https://doi.org/10.1145/1141911.1141931

Rémi Munos et al. 2014. From bandits to Monte-Carlo Tree Search: The optimistic principle applied to optimization and planning. *Foundations and Trends® in Machine Learning* 7, 1 (2014), 1–129.

Quan V Nguyen, Francis Colas, Emmanuel Vincent, and François Charpillet. 2017. Long-term robot motion planning for active sound source localization with Monte Carlo tree search. In *2017 Hands-free Speech Communications and Microphone Arrays (HSCMA)*. IEEE, 61–65.

Peter Norvig and Stuart Russell. 2002. *Artificial Intelligence: A modern approach* (3rd ed.). Prentice Hall.

Yoav I. H. Parish and Pascal Müller. 2001. Procedural Modeling of Cities. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01)*. Association for Computing Machinery, New York, NY, USA, 301–308. https://doi.org/10.1145/383259.383292

Deepak Pathak, Chris Lu, Trevor Darrell, Phillip Isola, and Alexei A Efros. 2019. Learning to control self-assembling morphologies: a study of generalization via modularity. *arXiv preprint arXiv:1902.05546* (2019).

Xue Bin Peng, Glen Berseth, KangKang Yin, and Michiel Van De Panne. 2017. Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Transactions on Graphics (TOG)* 36, 4 (2017), 1–13.

Jordan B Pollack, Gregory S Hornby, Hod Lipson, and Pablo Funes. 2003. Computer creativity in the automatic design of robots. *Leonardo* 36, 2 (2003), 115–121.

Jim Pugh and Alcherio Martinoli. 2007. Inspiring and modeling multi-robot search with particle swarm optimization. In *2007 IEEE Swarm Intelligence Symposium*. IEEE, 332–339.

Maarten PD Schadd, Mark HM Winands, H Jaap Van Den Herik, Guillaume MJ-B Chaslot, and Jos WHM Uiterwijk. 2008. Single-player monte-carlo tree search. In *International Conference on Computers and Games*. Springer, 1–12.

Charles Schaff, David Yunis, Ayan Chakrabarti, and Matthew R Walter. 2019. Jointly learning to construct and control agents using deep reinforcement learning. In *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 9798–9805.

Linda C Schmidt and Jonathan Cagan. 1997. GGREADA: a graph grammar-based machine design algorithm. *Research in Engineering Design* 9, 4 (1997), 195–213.

Linda C Schmidt, Harshawardhan Shetty, and Scott C Chase. 1999. A graph grammar approach for structure synthesis of mechanisms. *J. Mech. Des.* 122, 4 (1999), 371–376.

Karl Sims. 1994. Evolving virtual creatures. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. ACM, 15–22.

SN Sivanandam and SN Deepa. 2008. Genetic algorithms. In *Introduction to genetic algorithms*. Springer, 15–37.

Brian Smith, Ayanna Howard, John-Michael McNew, Jiuguang Wang, and Magnus Egerstedt. 2009. Multi-robot deployment and coordination with embedded graph grammars. *Autonomous Robots* 26, 1 (2009), 79–98.

Andrew Spielberg, Allan Zhao, Yuanming Hu, Tao Du, Wojciech Matusik, and Daniela Rus. 2019. Learning-In-The-Loop Optimization: End-To-End Control And Co-Design of Soft Robots Through Learned Deep Latent Representations. In *Advances in Neural Information Processing Systems*. 8282–8292.

Ondrej Šťava, Bedrich Beneš, Radomir Měch, Daniel G Aliaga, and Peter Krištof. 2010. Inverse procedural modeling by automatic generation of L-systems. In *Computer Graphics Forum*, Vol. 29. Wiley Online Library, 665–674.

George Stiny. 1982. Spatial Relations and Grammars. *Environment and Planning B: Planning and Design* 9, 1 (1982), 113–114. https://doi.org/10.1068/b090113

George Stiny and James Gips. 1971. 'Shape Grammars and the Generative Specification of Painting and Sculpture'. *IFIP Congress* 71, 1460–1465.

George Stiny and William J. Mitchell. 1978. The Palladian Grammar.

Fritz R Stöckli and Kristina Shea. 2015. A simulation-driven graph grammar method for the automated synthesis of passive dynamic brachiating robots. In *ASME 2015 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. American Society of Mechanical Engineers Digital Collection.

Russ Tedrake, Teresa Weirui Zhang, and H Sebastian Seung. 2004. Stochastic policy gradient reinforcement learning on a simple 3D biped. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, Vol. 3. IEEE, 2849–2854.

Merel Van Diepen and Kristina Shea. 2019. A spatial grammar method for the computational design synthesis of virtual soft locomotion robots. *Journal of Mechanical Design* 141, 10 (2019).

Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O Stanley. 2019a. Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions. *arXiv preprint arXiv:1901.01753* (2019).

Tingwu Wang, Yuhao Zhou, Sanja Fidler, and Jimmy Ba. 2019b. Neural Graph Evolution: Towards Efficient Automatic Robot Design. *arXiv preprint arXiv:1906.05370* (2019).

Richard A Watson, Sevan G Ficici, and Jordan B Pollack. 2002. Embodied evolution: Distributing an evolutionary algorithm in a population of robots. *Robotics and Autonomous Systems* 39, 1 (2002), 1–18.

Grady Williams, Paul Drews, Brian Goldfain, James M Rehg, and Evangelos A Theodorou. 2016. Aggressive driving with model predictive path integral control. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 1433–1440.

Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. 2003. Instant Architecture. *ACM Trans. Graph.* 22, 3 (July 2003), 669–677. https://doi.org/10.1145/882262.882324

Fuzhang Wu, Dong-Ming Yan, Weiming Dong, Xiaopeng Zhang, and Peter Wonka. 2014. Inverse Procedural Modeling of Facade Layouts. *ACM Transactions on Graphics* 33 (08 2014). https://doi.org/10.1145/2601097.2601162

Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. 2018. Hierarchical graph representation learning with differentiable pooling. In *Advances in neural information processing systems*. 4800–4810.