Benchmarking Open-Endedness in Minimal Criterion Coevolution

To appear in: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2019). New York, NY: ACM

Jonathan C. Brant and Kenneth O. Stanley
Dept. of Computer Science, University of Central Florida, Orlando, FL
{jbrant,kstanley}@cs.ucf.edu

ABSTRACT

Minimal criterion coevolution (MCC) was recently introduced to show that a very simple criterion can lead to an open-ended expansion of two coevolving populations. Inspired by the simplicity of striving to survive and reproduce in nature, in MCC there are few of the usual mechanisms of quality diversity algorithms: no explicit novelty, no fitness function, and no local competition. While the idea that a simple minimal criterion could produce quality diversity on its own is provocative, its initial demonstration on mazes and maze solvers was limited because the size of the potential mazes was static, effectively capping the potential for complexity to increase. This paper overcomes this limitation to make two significant contributions to the field: (1) By introducing a completely novel maze encoding with higher-quality mazes that allow indefinite expansion in size and complexity, it offers for the first time a viable, computationally cheap domain for benchmarking open-ended algorithms, and (2) it leverages this new domain to show for the first time a succession of mazes that increase in size indefinitely while solutions continue to appear. With this initial result, a baseline is now established that can help researchers to begin to mark progress in the field systematically.

CCS CONCEPTS

• **Computing methodologies → Artificial life**; *Evolutionary robotics*; *Reinforcement learning*; *Neural networks.*

KEYWORDS

Open-ended evolution, coevolution, artificial life

ACM Reference Format:
Jonathan C. Brant and Kenneth O. Stanley. 2019. Benchmarking Open-Endedness in Minimal Criterion Coevolution: To appear in: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2019). New York, NY: ACM. In *Genetic and Evolutionary Computation Conference (GECCO '19), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3321707.3321756

Permission to make digital or hard copies of all or part 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 components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GECCO '19, July 13–17, 2019, Prague, Czech Republic
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6111-8/19/07…$15.00
https://doi.org/10.1145/3321707.3321756

1 INTRODUCTION

A deep philosophical question has long intrigued researchers in artificial life (alife): what are the key ingredients of natural evolution that facilitate boundless discovery of novel and increasingly complex forms, and how can these processes be reproduced in a computational simulation [21]? This characteristic of evolution, known as *open-endedness* [36, 37, 40], has been studied primarily in the context of alife worlds where simulated creatures interact in an attempt to continuously generate subjectively interesting behaviors that are increasingly novel and complex [5, 22, 27, 31, 35, 43].

Open-endedness also has important implications to researchers in evolutionary computation (EC), where evolutionary abstractions are often harnessed to evolve solutions to problems of practical relevance [7]. While most EC practitioners rely on an objective (fitness) function to guide search, the objective may become a liability when, through deception, following its gradient leads search farther away from its desired target [15]. In contrast, a new class of methods have recently emerged that take a different approach to search. Known as *quality diversity* (QD) algorithms, instead of converging to an objective region of the search space, they intentionally exhibit *divergence* by encouraging behavioral novelty while still rewarding objectively superior discoveries [29, 30], thereby harnessing an aspect of open-endedness in a framework that can be applied to solving practical problems.

Yet while QD algorithms take a step towards open-endedness, they require additional mechanisms that have no precedent in nature. In particular, most QD algorithms require a behavior characterization (BC) that describes an individual's position in behavior space, which is used to assess and reward novelty with respect to current and past discoveries [16, 24, 29]. For example, in a robot locomotion task, behavior might be characterized by distance traveled and average velocity, and individuals who exhibit novel combinations of both would be rewarded accordingly. However, as the space of possible distances and velocities is saturated, novelty pressure is reduced. In that way, the finite capacity of the BC limits scalability when the hope is open-ended discovery. QD algorithms also depend on fitness functions to measure quality, and forms of local competition to ensure that individuals compete within their local niches. None of these mechanisms are *explicitly* imposed in nature, yet nature still exhibits open-endedness.

Brant and Stanley [3] introduced an alternate approach to QD called *minimal criterion coevolution* (MCC) that discards many of these conventional techniques. MCC draws inspiration from natural evolution's ability to produce boundless diversity and complexity in an almost entirely undirected manner. Unlike QD algorithms that use novelty to promote divergence and local competition to ensure

quality, MCC adopts an unconventional hypothesis: divergence can also be induced by *drift* subject to a minimal criterion (MC). The MC is a constraint on reproduction, and while it has been explored in prior research [14, 20], harnessing the MC to produce an open-ended search process and determining how it should calibrate over time to avoid stagnation remains a challenge. MCC addresses this problem by using *coevolution* to interlock two populations as they each simultaneously drift through the space of possible elaborations while still satisfying their MC with respect to each other.

MCC was first demonstrated in a maze navigation domain, where a population of mazes are coevolved with maze navigators controlled by neural networks (NNs). Each navigator is required to solve at least one maze to satisfy its MC, while each maze has to be solved by at least one maze navigator. In Brant and Stanley [3], mazes were constrained to a fixed size with a complexity ceiling (maximum walls that fit) to maintain tractability and allow rapid experimentation; however, the result was that MCC rapidly exhausted domain complexity, thereby raising a compelling question: *what sort of maze and navigator pairings could MCC discover if this artificial constraint were removed and maze size was unbounded?*

The aim of this paper is to provide an initial analysis of what happens when *both populations* in MCC are allowed to evolve and complexify without bound. An interesting facet of this experiment is that there are few simple benchmarks that enable evaluating methods like MCC aimed at open-ended discovery, and accordingly also no baselines with which to compare any future such approach. By introducing an enhanced maze encoding that allows realistic mazes to increase in size and complexity indefinitely, and by quantifying and analyzing how MCC behaves in such an unbounded domain, this paper provides both a benchmark and a baseline that can serve as a foundation for inspiring and measuring future progress in the field. It also offers a window into one of the first experiments of its type, where open-endedness is pursued in an unbounded domain that is not an alife world. The results show MCC exploiting the unbounded domain to produce larger and more complex mazes throughout each run, while maintaining a breadth of maze sizes and path configurations, each with effective solutions. Our hope is that this contribution provokes the continued growth of research and consequent progress into open-ended coevolutionary systems.

2 BACKGROUND

This section reviews open-endedness in alife and the history behind the MC in EC, followed by a discussion of coevolution and concluding with a description of the MCC algorithm.

2.1 Open-endedness

The concept of open-endedness has its roots in alife–a field concerned with analyzing and understanding natural systems and the process by which individuals within those systems interact and evolve [2, 31, 39]. Natural evolution is often viewed as an unguided process, yet one that continually produces novel and increasingly complex artifacts, a phenomenon referred to as open-ended evolution [18, 33, 36]. A significant and lofty goal of alife research is to formalize and reproduce (in a simulation) the dynamics that lead to an open-ended evolutionary process.

While there is a lack of consensus regarding precise methodologies for defining and measuring the characteristics of an open-ended evolutionary system, it is generally agreed that such a system produces novel, functional and increasingly complex forms in perpetuity [1, 17]. Soros and Stanley [35] recently proposed a set of conditions hypothesized to be necessary for the generation of an open-ended evolutionary process and test their application in an alife world dubbed Chromaria. Central among these conditions is the enforcement of an MC that places a lower bound on individual complexity and prevents the population from degenerating into trivial behaviors. Moreover, individuals must interact to satisfy their MC, which allows its difficulty to vary organically over time. By constraining evolution in this way, Chromaria has been shown to produce functional and novel forms over lengthy evolutionary runs without stagnating. The goal of MCC is to harness a similar abstraction of open-endedness in a general-purpose algorithmic framework for application in practical domains.

2.2 The Minimal Criterion

The field of EC has historically been optimization-oriented, selecting individuals for reproduction based on an objective measure of performance and iteratively converging toward a fixed, a priori objective [7]. However, recent research has introduced alternative methods of search that jettison the narrow focus of the objective, opening up the search space by imposing a looser performance criterion. The concept of imposing a lower bound on solution quality was originally introduced by Mattiussi and Floreano [20] and later employed by Maesani et al. in an algorithm called *viability evolution* (ViE) [19]. ViE imposes a set of thresholds, or *viability boundaries*, on one or more constraints that individuals must meet to be eligible for reproduction. Initially, boundaries encompass the entire population, but are incrementally tightened until each constraint is satisfied, and those who remain are taken as solutions.

A convenient property of ViE is that it requires no explicit objective function, which is particularly advantageous for multiobjective problems where objective weighting significantly impacts solution characteristics. Moreover, the loose criterion of viability allows ViE to maintain a more diverse population and sample a broader range of the search space. However, ViE remains an algorithm that is driven toward convergence, and thus breaks with the notion of open-ended discovery.

Non-objective search methods, such as novelty search [13, 15] and QD algorithms [16, 24, 29, 30], also depart from traditional methods of objective-driven evolutionary search. Instead of converging toward a narrow objective, novelty search induces search space *divergence* by rewarding novelty alone. This approach is particularly effective for deceptive problems, where the gradient of the objective may lead search astray. A side of effect of rewarding only novelty, however, is the tendency to explore uninteresting areas of potentially vast behavior spaces [6]–an insight that led to the introduction of a variant called minimal criteria novelty search (MCNS) [14]. MCNS imposes behavior space boundaries which define the MC; individuals that traverse beyond those boundaries fail to meet the MC and are therefore ineligible for reproduction. This constraint avoids propagating a lineage of individuals that would explore areas of the search space orthogonal to those of interest.

In contrast to the aforementioned algorithms, MCC has no requirement for a behavior characterization, novelty archive or viability boundaries, relying on the MC *alone* as the sole driver of a broadly applicable, open-ended search process.

2.3 Coevolution

Most EAs adopt a static fitness function as a global, extrinsic measure of performance; however, in some domains this absolute performance metric may be computationally intractable or impossible to formalize. Coevolution in EC addresses these shortcomings by defining fitness as a relative measure that is based on interactions between individuals within a single population, or between two separate populations [4, 28]. Coevolutionary learning is typically framed in either a competitive or cooperative context. In competitive coevolution, self-interested individuals are pitted against each other and rewarded based on one individual's ability to outcompete the other [32]. Conversely, in cooperative coevolution, individuals work together to achieve an overarching objective [42].

An interesting property of coevolution is its theoretical ability to produce open-endedness by inducing an unending arms race [26]; however, this dynamic has proven difficult to sustain. Instead of producing creative behaviors and interactions in perpetuity, coevolutionary algorithms eventually converge to mediocre stable states, wherein individuals lack a fitness-based incentive to evolve beyond a limited set of sub-optimal strategies [9]. Recent work has suggested that replacing fitness with a novelty metric may help minimize coevolutionary stagnation in both a competitive and cooperative context by maintaining healthier levels of diversity within both populations [10, 11]. MCC also employs coevolution outside of a fitness-based paradigm, but frames interactions in a manner that is neither competitive nor cooperative. Instead, individuals are permitted to interact freely within the confines of their respective MC, thus promoting a divergent search dynamic that is intrinsically immune to evolutionary stasis.

2.4 Minimal Criterion Coevolution

Evolution on earth is perhaps the most dramatic example of a divergent process; it operates over vast periods and at a massive scale, producing a broad array of artifacts, each at varying levels of complexity and well-adapted to their ecological niche. There is no explicit drive toward a global performance criterion, and while novelty may exhibit a passive force on selection [18], there is no evidence of an explicit preference for novel behaviors. Instead, nature enforces an MC of reproductive viability, and organisms or species may discover a multitude of different ways in which to satisfy their MC. For example, the developmental path toward reproductive viability for bacteria tends to be far shorter and less complex than for mammals. Furthermore, the interactions between coevolving populations results in an on-going flux of each species' path toward their respective MC.

Inspired by this perspective, Brant and Stanley [3] proposed a dual-population, coevolutionary algorithm called minimal criterion coevolution (MCC). MCC does not evaluate and rank individuals by way of competition, nor does it grade solutions based on cooperative cohesion. Instead, individuals are evaluated and permitted to reproduce based solely on their satisfaction of an MC chosen by the experimenter (the MC for each population can be different).

Because this evaluation process imposes no ranking among individuals, the selection method is free of bias. The idea of such a coarse-grained selection (either you pass or not) within a coevolutionary system is to allow search to drift throughout the entire space of viable candidates, thereby covering as many possible stepping stones as possible without a priori bias towards which stepping stones are the right ones. In effect it is an attempt to remove artificially explicit selection pressure (e.g. towards novelty) while showing that QD and open-endedness are still possible.

Drawing inspiration from Chromaria's population structure and selection process [35], MCC stores both populations in a fixed-size queue. The queue retains individuals who satisfy the MC in the order of insertion, and a queue pointer identifies the next individual in line for reproduction. If the pointer reaches the end of the queue, it loops back to the beginning, thereby ensuring that every individual who satisfies the MC gets *at least one* chance to reproduce. If the insertion of a new individual would cause the queue to exceed its capacity, the oldest in the queue is removed to make room.

Most EAs start evolution with a randomly-generated population; however, it is unlikely that random individuals will possess sufficient complexity to satisfy a non-trivial MC from the start, rendering them inadmissible to either of the population queues. To mitigate this initialization problem, MCC uses a bootstrap process wherein the requisite number of genomes are pre-evolved to meet the MC and used to seed each population queue. In principle, any EA (objective or non-objective) could be used to evolve the seed genomes; however, novelty search was used for the bootstrap phase in the initial formulation of MCC based on its propensity for discovering a diverse set of solutions.

In nature, organisms are grouped into species, each of which occupy a distinct ecological niche [34]. Niches have a limited carrying capacity, which imposes a form of local regulation on the occupying species and encourages divergence by way of founding new niches. While optional, speciation was shown to aid in maintaining the reproductive viability of several lineages simultaneously. Individuals remain physically stored in population queues, but are logically clustered into species based on genetic similarity (a well-established method of diversity preservation in EAs [8]). Importantly, genetic similarity is *not* the same as novelty because it is entirely genetically-based and not aware of phenotypic behavior. At the beginning of evolution, seed genomes serve as species cluster centroids, and new additions are assigned to the closest species in genome space. Queue capacity is evenly distributed among species such that each species i has a maximum size equivalent to $\text{capacity}(i) = \frac{n}{s}$, where n is the number of individuals in the population, and s is the number of species. Similarly, selection draws proportionately from each species while respecting queue insertion order. If any species exceed its respective carrying capacity, the oldest from that species (rather than the global oldest) is removed to make room. Algorithm 1 formalizes the MCC selection, evaluation and removal process with speciation.

The selection process employed by MCC resembles that of a steady-state evolutionary algorithm; however, while steady state algorithms typically evolve the population serially, MCC's independent performance measure (the MC) permits much of the population to be evaluated in parallel. This decoupling in the evaluation

Algorithm 1 MCC Evaluation Process with Speciation

```
Require:
  batchSize - # of individuals to evaluate simultaneously
  numSpecies - # of species

  ▷ Evolve seed genomes that satisfy MC
  randPop ← GenerateRandomPopulation()
  viablePop ← EvolveSeedGenomes(randPop, numSpecies)
  ▷ Seed species with each viable individual as centroid
  Species ← SeedSpecies(viablePop)

  loop
    ▷ Produce offspring from each species & reinsert parents
    for all species ∈ Species do
      ▷ Produce offspring from current species
      parents ← species.Dequeue(batchSize)
      children ← Reproduce(parents)
      ▷ Reinsert parents into queue
      species.Enqueue(parents)

      for all child ∈ children do
        mcSatisfied ← EvaluateMC(child)
        if mcSatisfied then
          viablePop.Enqueue(child)
        end if
      end for
    end for

    ▷ Respeciate based on addition of viable children
    viablePop.Respeciate()

    for all species ∈ Species do
      ▷ Remove oldest from species if species capacity exceeded
      if species.Size > species.Capacity then
        numRemovals ← species.Size − species.Capacity
        RemoveOldest(species, numRemovals)
      end if
    end for
  end loop
```

stage facilitates a distributed execution paradigm wherein multiple evaluations can be executed simultaneously on separate nodes.

While the aim of introducing MCC was to demonstrate that even the most simple ingredients, namely genetic drift within a coevolving system, can support a divergent, open-ended process, a limitation of the initial experiments, which centered on mazes and maze-solvers, was that the mazes were static in size, and therefore bounded from increasing in complexity indefinitely. The work in this paper removes that bound through an entirely new maze encoding that is also offered as a lightweight benchmark for testing and comparing open-ended coevolutionary methods in the future.

3 APPROACH: UNBOUNDED MAZES

The first experiments with MCC introduced an evolving maze domain to demonstrate the algorithm's ability to coevolve principled navigation strategies in maze navigating agents along with maze environments. Maze generation is a particular instance of the broader challenge of procedural content and level generation, which is reviewed in Togelius et al. [41]. Mazes are a commonly-used benchmark domain in the non-objective search literature because they yield an interpretable sense of complexity through an arrangement of walls that complicate navigation, while also providing an explicit indication of domain deception through the presence of dead-ends and wandering paths [14, 15, 23, 25, 29]. These visual cues facilitate qualitative assessment of learned policies. While a QD algorithm can discover a diversity of solutions for a single maze, MCC discovers varying solutions to many different mazes within a single run and without the need for a BC or an archive. Maze navigating agents are simulated, wheeled robots that are controlled by evolved NNs. In Brant and Stanley [3] and in this paper, the Neuroevolution of Augmenting Topologies (NEAT) [38] algorithm is used to evolve the NN weights and topologies, though other neuroevolution algorithms are also feasible. NEAT incrementally adds structure to NNs, increasing the number of free parameters and facilitating representation of more complex strategies, thus making it ideal for a domain that is intended to demonstrate an open-ended process. The NN controller architecture in the ensuing experiments is identical to Brant and Stanley [3], with six rangefinder sensors (positioned at heading offsets of $-90°$, $-45°$, $0°$, $45°$, $90°$ and $180°$) that measure distance to line-of-sight obstructions, and four pie-slice radar sensors that cover the full circumference of the agent and activate when the target is within the sensors' arc. Two actuators apply forces that turn and propel the agent.

Evolved mazes are square grids with a start location in the upper-left corner and target location in the lower-right. Both points remain fixed throughout evolution. Agents begin a trial at the start location and are evaluated on their ability to reach the target location within a given simulation time; however, because mazes will be allowed to expand, the simulation time is dependent on the length and complexity of the solution path. The maze consists of internal walls that impede trivial, straight-line trajectories, as well as winding and deceptive traps through which an agent can wander and expend the allotted time. Both of these features are intended to encourage the development of complex and principled behaviors.

The original MCC experiments exploit both mazes and navigators to demonstrate how the MC-mediated interaction between the two populations leads to a potentially open-ended dynamic. However, the fixed-sized maze canvas size imposes a complexity ceiling by limiting the number of walls that can be added to the maze, which in turn reduces pressure on the agent population to evolve strategies more complex than those that can solve the maximum complexity mazes, thereby blunting the benefit of NEAT-style complexification in the NN population.

The major contribution of this paper is to introduce a fundamentally more powerful maze encoding to allow the full potential of algorithms like MCC to be tested. Furthermore, not only is the new encoding able to support unbounded increases in complexity, but it also produces aesthetically superior mazes, which are closer in appearance to conventional mazes, with the hope that these improvements will encourage more widespread adoption of the maze domain as a benchmark for open-ended coevolution. This advance is important not just for MCC, but for the field in general, because

to date there is no simple, easily-adopted benchmark within which to test and compare such algorithms.

With the new encoding, the MCC bootstrap process is unchanged: seed agent genomes are evolved to solve a set of randomly-generated mazes with fixed boundaries and uniform size. Upon completion of the bootstrap, the MCC reproduction process includes an *expand maze* mutation probability that controls the proportion of mutations that result in a maze expansion. When this mutation is applied, the outer boundaries of the affected maze are each lengthened by one unit, maintaining a square geometry. Expansion occurs down and to the right, thereby moving the target location further from the starting location and walls are lengthened proportionally to fill additional space.

After several iterations, the MCC maze population includes mazes that not only vary in structure, but also in size, with larger mazes permitting the addition of more interior walls and the formation of more complex solution paths. The hope is that this unending cycle of maze expansion and elaboration will encourage the development of increasingly complex navigation strategies in the agent population, promoting an unending discovery of increasingly complex and novel problems and solutions.

In the initial MCC experiments, walls were placed so as to permit only one path from start to finish. Yet crafting the solution path *after* wall placement has two shortcomings: (1) small changes in wall placement could result in substantial modifications to the path (potentially leading to a highly rugged search landscape), and (2) as mazes expand and more walls are added, the path is further subdivided into short directional segments, yielding a degenerate and unconvincing aesthetic to the mazes (figure 1, left column).

The main idea behind the new encoding is, rather than rely upon wall placement to mold the path, the *path itself is evolved* as a series of "waypoints," joined by a simple set of rules that derive the solution trajectory. The resulting path carves out sub-spaces in the maze, which open onto the trajectory and are themselves complexified by the positioning of wall genes, thereby introducing multiple opportunities for the agent to wander off the path and get stuck in long, winding corridors, which also gives an appearance more like conventional mazes (figure 1, middle and right columns).

Waypoint coordinates (called *path genes*) are directly encoded in the maze genome and augmented with an "intersection orientation," which specifies the direction of the path segment (i.e. vertical or horizontal) as it intersects the waypoint. A *mutate waypoint* probability shifts an existing waypoint by one unit in one of four directions (up, down, left or right) while an *add waypoint* probability controls the addition of new waypoints to the maze. The intersection orientation is randomly assigned and fixed. Waypoints are connected in the order that they are added to the maze genome, with respect to their intersection orientation. The leftmost maze in figure 2 (which shows how the encoding works) depicts a maze with one waypoint and a vertical intersection orientation at coordinate $(6, 4)$. The connecting path is laid down vertically from the start location, then forms a horizontal connection to the waypoint. Waypoint additions are constrained such that they are added either below (as in step 3 of figure 2) *or* to the right of all existing waypoints (recall that the maze expands down and to the right). Similarly, mutation cannot shift a waypoint both above and to the left of prior waypoints. These constraints prevent trajectory overlap while still permitting a solution path that forces movement in all cardinal directions.

When decoding a maze genome into its phenotype representation, the solution path is first determined, and then walls are positioned in sub-spaces of the maze induced by the path. Wall genes are encoded as in the original MCC experiments, with wall and passage positions represented as real numbers in the interval $[0, 1]$. Rather than dividing the entire maze, walls iteratively bisect their respective sub-space. For example, the first gene bisects the entire sub-space, creating two smaller sub-spaces with a passage through which the agent can traverse. If the bisection is vertical, the next gene bisects the left-most subspace, creating two additional sub-spaces. Similarly, if the bisection is horizontal, the next gene bisects the uppermost sub-space. This process repeats until each sub-space formed by the trajectory is filled with the maximum number of supported walls. If there are not enough wall genes to uniquely specify each possible bisection, genes are repeated from the beginning of the gene list and scaled to the dimensions of the applicable sub-space. Figure 2 depicts the trajectory complexification process and the genotype-phenotype mapping, while Supplemental Information (appendix A) contains detailed pseudocode for each stage of the maze generation process. As figure 1 shows, the new maze encoding produces non-trivial, winding paths while also allowing the maze environment to expand over time. Because the maze is built around the solution path, it looks more neatly filled. With this encoding, we can begin to observe the behavior of algorithms in which mazes can expand indefinitely.

4 EXPERIMENTAL SETUP

The bootstrap process is executed using the novelty search algorithm with parameters identical to those in Lehman and Stanley [15]. Agents are evaluated on ten simple, randomly-generated mazes, each with two waypoints and two wall genes. Execution halts when 20 distinct agents are evolved that can solve one or more mazes. Those 20 agents and 10 mazes seed their respective population queues and constitute the initial centroids of each species cluster. Each agent must solve at least one maze to satisfy their MC while each maze must be solved by at least one agent. Agents are evaluated on random mazes from the current population and vice versa, and evaluation halts when an individual satisfies their MC or when they have been evaluated on all members of the opposite population.

As with Brant and Stanley [3], experiments are executed for 20 runs and 2,000 batches per run, each batch evaluating 40 agent genomes and 10 maze genomes. The agent queue has a maximum capacity of 250 with a 0.6 probability of mutating connection weights, 0.1 probability of adding a connection, 0.01 probability of adding a neuron and a 0.005 probability of deleting a connection. The maze queue has a maximum capacity of 50 with a 0.05 probability of mutating a wall, passage or waypoint location, 0.1 probability of adding a wall, 0.005 probability of deleting a wall, 0.1 probability of adding a waypoint and a 0.1 probability of maze expansion. These values produced complex mazes of variable size and complexity during a parameter sweep.

Agents have a maximum velocity of 3 units per second and are allotted a simulation time equivalent to two times the length of the solution path. This limit allots agents simulation time proportional

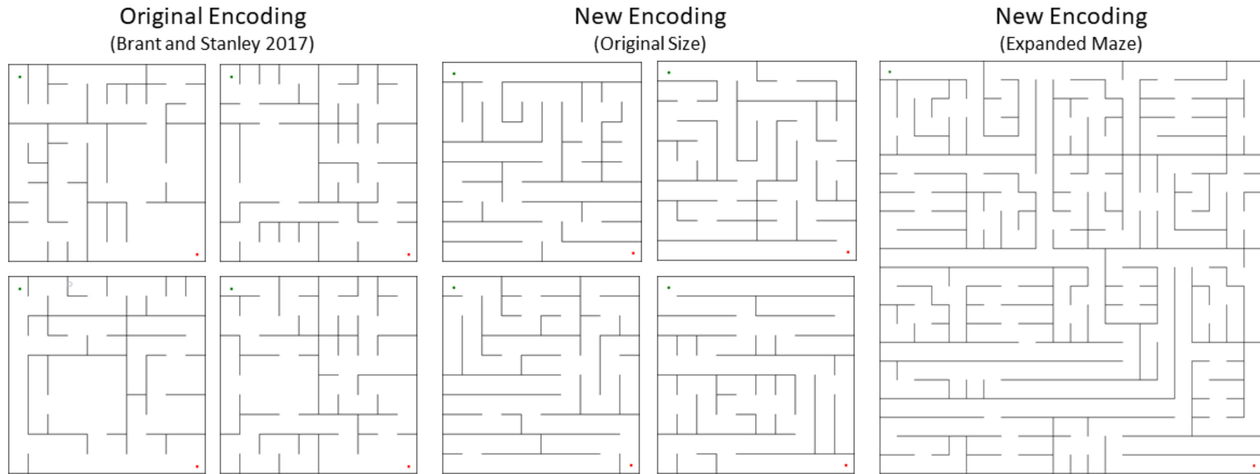

Figure 1: Comparing the original [3] and new maze encodings. Mazes generated in the original MCC experiments (left) are shown next to mazes generated by the new encoding, at different sizes (middle and right). While the original maze encoding prevents trivial strategies by obstructing straight line navigation, the solution path lacks a distinct shape, and wall placement is often unbalanced and overly segmented. In contrast, the new maze encoding produces non-trivial solution paths with balanced wall placement, and maintains these properties as the maze expands.

to the original MCC experiments with fixed-size mazes, while scaling with the size of the maze and the complexity of the solution path. SharpNEAT version 3.0 [12] is the neuroevolution platform, and was extended to implement the MCC and novelty search algorithms, as well as to support encoding and evolving maze genomes. Source code for the experiments is available at https://bit.ly/2uXGBpm. Note that a direct quantitative comparison of the present results with those presented in Brant and Stanley [3] would not be feasible because these experiments enable unbounded domain expansion – a characteristic that the original experiments lacked.

5 RESULTS

A primary goal of MCC is to induce an open-ended search process through coevolutionary interactions that result in increasingly complex and diverse discoveries. The results here probe MCC's ability to exploit the unbounded maze domain, evolving agents that are capable of solving larger, more complex mazes. The qualitative results showcase a diverse array of mazes that were evolved in a single run, each with varying size and complexity, and solved by an agent controller. Additionally, a quantitative analysis demonstrates a systematic increase in both maze and navigator complexity.

5.1 Qualitative Results

As mazes expand, additional waypoints can be added, which sometimes in turn further complicate the solution path and introduce more opportunities for deception through adjoining cul-de-sacs. Figure 3 depicts a sample of agent trajectories (from evolved NN controllers) through mazes evolved by a *single* run of MCC, each varying in both size and structure. These results show visually how mazes are expanding in size while continuing to be solved.

5.2 Quantitative Results

A quantitative analysis of domain size and complexity growth over evolution ascertains the open-ended characteristics of MCC, and contrasts these results with the original MCC experiments. All graphs in this section are shown with 95% confidence bounds on the means.

A key feature of the reconceived maze domain is the ability to dynamically expand, allowing additional space for longer, more convoluted trajectories, which can force agents to evolve increasingly complex navigation strategies to remain viable (i.e. to satisfy their MC). Figure 4 demonstrates that mazes indeed increase in size, linearly on average, without leveling off. The number of junctures (where turns are necessary in the main solution path) continues to increase at a similar rate (figure 5). As both results also show, MCC, on average, maintains a healthy diversity of maze sizes and configurations throughout each run. Regarding the agent NNs, figure 6 depicts a linear increase in maximum and mean number of connections throughout evolution, demonstrating, in part, the agents' evolutionary response to increasingly difficult mazes. However, the minimum complexity remains flat as less complex agents remain viable in smaller, less complex mazes. Moreover, the widening gap between minimum and maximum sizes suggests a continued search space divergence that shows no sign of leveling off.

A notable, yet intuitive distinction between the fixed maze domain used in the original MCC experiments [3] and the reconceived, expanding maze domain is the trend in population viability over the course of evolution. In particular, after batch 511, the percentage of maze offspring that are viable in a run of fixed mazes (averaged over 20 runs) is significantly higher than for expanding mazes ($p < 0.05$; Welch's t-test) because the fixed mazes have reached their complexity limit and therefore stop consistently posing new challenges to the agent population. On average, only between 10% and 15% of the agent offspring satisfy their MC on expanding mazes at any point in evolution. By allowing mazes to expand unboundedly, MCC is able to consistently generate new challenges for the agent population.

Figure 2: The maze evolution process. Maze genotypes (top), respective trajectories (middle), and respective phenotypes (bottom) are shown. The table above each maze enumerates the path (waypoint) and wall genes, while the visualization directly below depicts the resulting trajectory. In the maze phenotypes at bottom, dots are shown to depict the solution path. Each path gene specifies the coordinates of a waypoint, and each wall gene contains a wall position and passage position, both real numbers in the interval $[0, 1]$. The wall position (W) describes the relatives position of a wall within a sub-space induced by the solution path, while the passage position (P) describes the relative position of a passage (opening) within that wall. The initial genome (first column) consists of one path gene and one wall gene; the single wall gene is repeated to fill out all sub-spaces. The second column depicts an *add wall* mutation, while the third column demonstrates an *add waypoint* mutation. In the fourth column, a *mutate waypoint* operation is carried out, shifting the first waypoint one unit to the right. Finally, the fifth column depicts an *expand maze* mutation, extending the outer boundaries down and to the right, and resulting in a corresponding extension of the solution path.

6 DISCUSSION

The expandable maze encoding makes it possible to observe the MCC algorithm's ability to exploit a novel maze domain, relieved of complexity constraints and designed to grow unboundedly subject to the MC. Reciprocating increases in size are evident in both populations with no apparent stagnation. In all runs, agents evolved solutions to a breadth of non-trivial mazes, inducing a population-wide drift toward increasingly intricate mazes with multiple opportunities for deception. Both populations achieved an MC-constrained equilibrium with each other, maintaining a sustainable rate of complexification and producing a diverse array of mazes and associated solutions, relying on neither the guidance of fitness nor the divergent pressure of novelty.

Although these results are among the first to explore open-ended coevolutionary growth and complexification, there remain several opportunities for improvement. A close examination of learned navigation strategies hints that, in some cases, agents may be relying on simpler heuristics than their trajectories would suggest. For example, in figure 3 maze A, the agent is forced to make five decisions regarding the direction to turn. However, for all but one of those decisions, the agent only has one option that does not lead into a wall or backtrack on the solution path. This fact is a potential hint of some degree of collusion in the maze and agent populations: mazes discover the "path of least resistance" for adding complexity that is easily exploitable by agents. While not a desirable system characteristic, the extent to which such behavior can be observed at all is vastly improved in the reconceived maze encoding, offering hope that they can be addressed and overcome as MCC is extended and new coevolutionary open-ended systems are introduced and similarly benchmarked.

For example, one interesting future direction is to alter the MC in MCC and observe the consequent effects on unbounded mazes. We might, for example, require that ancestors cannot solve the same mazes as their progeny, thereby preventing trivial expansion from the perspective of the NNs. Additional diversity pressure could also be added, for example by limiting the number of agents that can use the same maze to satisfy their MC. The investigation of all such opportunities benefits now from the availability of the new unbounded maze benchmark, which can more clearly reveal their implications. Additionally, the maze domain provides a baseline

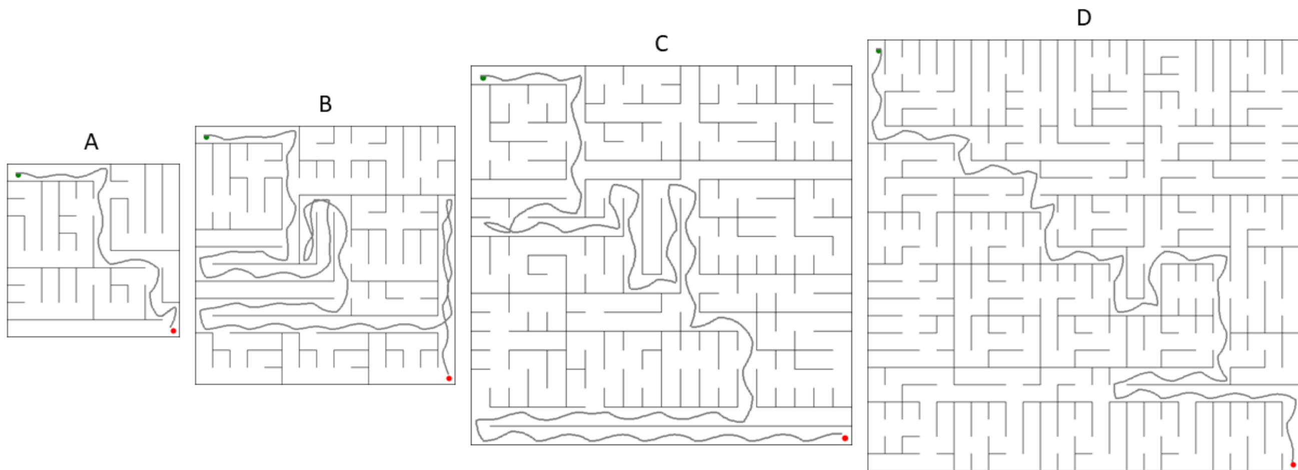

Figure 3: Agent Trajectories discovered within a single run of MCC. A sample of four solution trajectories through different mazes evolved within a single run of MCC are shown. Maze A is similar to the bootstrap mazes in size and complexity, with outer walls that are 10 units in length, 2 waypoints and 2 interior walls. Maze B is 15x15 with 4 waypoints and 3 walls, while maze C is 22x22 with 7 waypoints and 12 walls. The largest maze (D) is 25x25 with 8 waypoints and 6 walls.

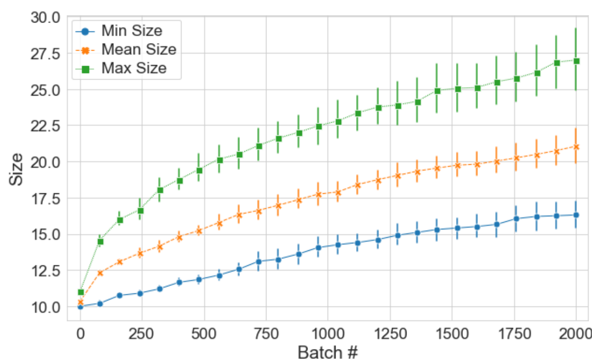

Figure 4: Maze Expansion Trend. The average minimum, mean and maximum maze dimensions over evolution are depicted. Mazes expand linearly throughout the course of a run with no evidence of tapering off, suggesting MCC's ability to induce a process of unbounded maze elaboration.

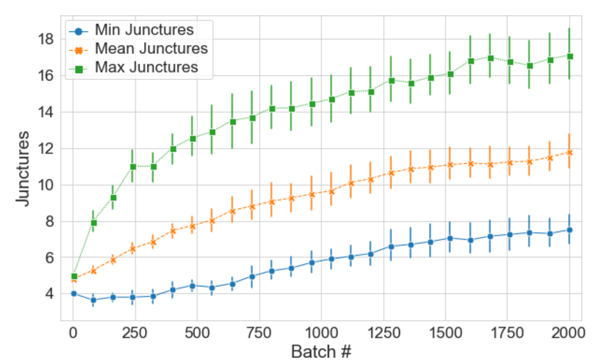

Figure 5: Solution Path Complexification Trend. The average minimum, mean and maximum junctures over evolution are shown. As waypoints are added, the solution path is convoluted by junctures–perpendicular intersections from waypoint connections.

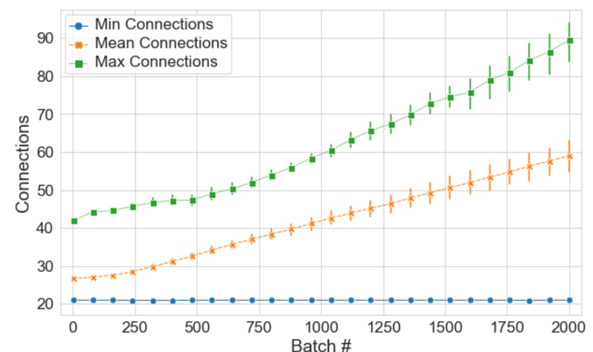

Figure 6: Agent NN Size Trend. The average minimum, mean and maximum agent NN connections over evolution are shown. Maximum and mean NN size increases as agents evolve to solve more difficult mazes, while some agents stay small.

validation for MCC and its variants as they are extended to support applications in other domains, such as evolutionary robotics, generative design, open-world games and others that have yet to be conceived.

7 CONCLUSIONS

This paper introduced a novel unbounded maze domain designed to further evaluate MCC's capacity for producing open-ended divergence in both the maze and agent populations. The results both help to establish a set of promising dynamics as mazes and solution paths expand, and also potential avenues for further investigation, such as into the extent of collusion to minimize the necessary complexity of solution strategies. At the same time, the reconceived maze domain provides a lightweight benchmark for directly assessing and comparing future MCC-inspired open-ended algorithms.

REFERENCES

- [1] Christoph Adami, Charles Ofria, and Travis C. Collier. 2000. Evolution of Biological Complexity. *Proceedings of the National Academy of Sciences* 97 (2000), 4463–4468.
- [2] Margaret A Boden. 1999. The philosophy of artificial life. *Minds Mach.* 9, 1 (Feb 1999), 139–143. https://doi.org/10.1023/A:1008373528631
- [3] Jonathan C Brant and Kenneth O Stanley. 2017. Minimal criterion coevolution: a new approach to open-ended search. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 67–74.
- [4] Larry Bull. 2001. On coevolutionary genetic algorithms. *Soft Computing* 5, 3 (2001), 201–207.
- [5] Alastair Daniel Channon and Robert I Damper. 2000. Towards the evolutionary emergence of increasingly complex advantageous behaviours. *International Journal of Systems Science* 31, 7 (2000), 843–860.
- [6] Giuseppe Cuccu and Faustino Gomez. 2011. When novelty is not enough. In *European Conference on the Applications of Evolutionary Computation*. Springer, 234–243.
- [7] Kenneth A De Jong. 2006. *Evolutionary computation: a unified approach.* MIT press.
- [8] Antonio Della Cioppa, Angelo Marcelli, and Prisco Napoli. 2011. Speciation in Evolutionary Algorithms: Adaptive Species Discovery. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation (GECCO ’11)*. ACM, New York, NY, USA, 1053–1060. https://doi.org/10.1145/2001576.2001719
- [9] Sevan G Ficici and Jordan B Pollack. 1998. Challenges in Coevolutionary Learning: Arms–Race Dynamics, Open–Endedness, and Mediocre Stable States. In *Proceedings of the Sixth International Conference on Artificial Life*, Adami et al (Ed.). MIT Press, Cambridge, MA, 238–247.
- [10] Jorge Gomes, Pedro Mariano, and Anders Lyhne Christensen. 2014. Avoiding convergence in cooperative coevolution with novelty search. In *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*. International Foundation for Autonomous Agents and Multiagent Systems, 1149–1156.
- [11] Jorge Gomes, Pedro Mariano, and Anders Lyhne Christensen. 2014. Novelty search in competitive coevolution. In *International Conference on Parallel Problem Solving from Nature*. Springer, 233–242.
- [12] Colin Green. 2016. SharpNEAT v3. *URL: http://sharpneat.sourceforge.net* (2016).
- [13] Joel Lehman and Kenneth O Stanley. 2008. Exploiting Open-Endedness to Solve Problems Through the Search for Novelty.. In *Proceedings of the Eleventh International Conference on Artificial Life (Alife XI)*. MIT Press, 329–336.
- [14] Joel Lehman and Kenneth O Stanley. 2010. Revising the evolutionary computation abstraction: minimal criteria novelty search. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2010)*. ACM, 103–110.
- [15] Joel Lehman and Kenneth O Stanley. 2011. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation* 19, 2 (2011), 189–223.
- [16] Joel Lehman and Kenneth O Stanley. 2011. Evolving a diversity of virtual creatures through novelty search and local competition. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. ACM, 211–218.
- [17] Richard E Lenski, Charles Ofria, Robert T Pennock, and Christoph Adami. 2003. The evolutionary origin of complex features. *Nature* 423, 6936 (2003), 139–144.
- [18] Michael Lynch. 2007. The frailty of adaptive hypotheses for the origins of organismal complexity. *Proceedings of the National Academy of Sciences* 104, suppl 1 (2007), 8597–8604.
- [19] Andrea Maesani, Pradeep Ruben Fernando, and Dario Floreano. 2014. Artificial evolution by viability rather than competition. *PloS one* 9, 1 (2014), e86831.
- [20] Claudio Mattiussi and Dario Floreano. 2003. *Viability Evolution: Elimination and Extinction in Evolutionary Computation.* Technical Report. EPFL.
- [21] T. Miconi. 2008. Evolution and complexity: The double-edged sword. *Artificial life* 14, 3 (2008), 325–344.
- [22] Thomas Miconi. 2008. Evosphere: evolutionary dynamics in a population of fighting virtual creatures. In *IEEE Congress on Evolutionary Computation*. IEEE, 3066–3073.
- [23] Jean-Baptiste Mouret. 2011. Novelty-based multiobjectivization. In *New Horizons in Evolutionary Robotics*. Springer, 139–154.
- [24] Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909* (2015).
- [25] Jean-Baptiste Mouret and Stéphane Doncieux. 2012. Encouraging behavioral diversity in evolutionary robotics: An empirical study. *Evolutionary computation* 20, 1 (2012), 91–133.
- [26] Stefano Nolfi and Dario Floreano. 1998. Coevolving predator and prey robots: do “arms races” arise in artificial evolution? *Artificial life* 4, 4 (1998), 311–335.
- [27] Charles Ofria and Claus O Wilke. 2004. Avida: A software platform for research in computational evolutionary biology. *Artificial life* 10, 2 (2004), 191–229.
- [28] Elena Popovici, Anthony Bucci, R. Paul Wiegand, and Edwin D. De Jong. 2012. *Coevolutionary Principles.* Springer Berlin Heidelberg, Berlin, Heidelberg, 987–1033. https://doi.org/10.1007/978-3-540-92910-9_31
- [29] Justin K Pugh, Lisa B Soros, and Kenneth O Stanley. 2016. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI* 3 (2016), 40.
- [30] Justin K Pugh, Lisa B Soros, Paul A Szerlip, and Kenneth O Stanley. 2015. Confronting the challenge of quality diversity. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2015)*. ACM, 967–974.
- [31] Thomas S. Ray. 1991. An Approach to the Synthesis of Life. *Artificial Life II* (1991), 371–408. https://doi.org/10.1016/S0040-4039(01)97473-8
- [32] Christopher D Rosin and Richard K Belew. 1997. New methods for competitive coevolution. *Evolutionary computation* 5, 1 (1997), 1–29.
- [33] Kepa Ruiz-Mirazo, Juli Peretó, and Alvaro Moreno. 2004. A universal definition of life: autonomy and open-ended evolution. *Origins of Life and Evolution of the Biosphere* 34, 3 (2004), 323–346.
- [34] Howard D Rundle and Patrik Nosil. 2005. Ecological speciation. *Ecology letters* 8, 3 (2005), 336–352.
- [35] L.B. Soros and Kenneth O Stanley. 2014. Identifying Necessary Conditions for Open-Ended Evolution through the Artificial Life World of Chromaria. In *ALIFE 14: The Fourteenth International Conference on the Synthesis and Simulation of Living Systems*. 793–800.
- [36] Russell K Standish. 2003. Open-ended artificial evolution. *International Journal of Computational Intelligence and Applications* 3, 02 (2003), 167–175.
- [37] Kenneth O. Stanley, Joel Lehman, and L. B. Soros. 2017. Open-endedness: The last grand challenge you’ve never heard of. https://www.oreilly.com/ideas/open-endedness-the-last-grand-challenge-youve-never-heard-of
- [38] Kenneth O Stanley and Risto Miikkulainen. 2002. Evolving neural networks through augmenting topologies. *Evolutionary computation* 10, 2 (2002), 99–127.
- [39] Charles Taylor and David Jefferson. 1993. Artificial life as a tool for biological inquiry. *Artificial Life* 1, 1_2 (1993), 1–13.
- [40] Tim Taylor, Mark Bedau, Alastair Channon, David Ackley, Wolfgang Banzhaf, Guillaume Beslon, Emily Dolson, Tom Froese, Simon Hickinbotham, Takashi Ikegami, Barry McMullin, Norman Packard, Steen Rasmussen, Nathaniel Virgo, Eran Agmon, Edward Clark, Simon McGregor, Charles Ofria, Glen Ropella, Lee Spector, Kenneth O. Stanley, Adam Stanton, Christopher Timperley, Anya Vostinar, and Michael Wiser. 2016. Open-Ended Evolution: Perspectives from the OEE Workshop in York. *Artificial Life* 22, 3 (2016), 408–423. https://doi.org/10.1162/ARTL_a_00210 PMID: 27472417.
- [41] Julian Togelius, Georgios Yannakakis, Kenneth O. Stanley, and Cameron Browne. 2010. Search-based Procedural Content Generation. In *Proceedings of the 2nd European event on Bio-inspired Algorithms in Games (EvoGAMES 2010)*. Springer, New York: NY.
- [42] R Paul Wiegand, William C Liles, and Kenneth A De Jong. 2001. An empirical analysis of collaboration methods in cooperative coevolutionary algorithms. In *Proceedings of the genetic and evolutionary computation conference (GECCO)*, Vol. 2611. 1235–1245.
- [43] Larry Yaeger. 1993. PolyWorld: Life in a new context. *Artificial Life III* (1993), 263–298.

A SUPPLEMENTAL INFORMATION

Algorithm 2 Maze Generation

```
1:  function GenerateMaze(mazeGenome)
2:      pathGenes ← mazeGenome.PathGenes          ▷ Genes in maze genome that code for waypoints and define path
3:      wallGenes ← mazeGenome.WallGenes          ▷ Genes in maze genome that code for internal walls
4:      height ← mazeGenome.Height
5:      width ← mazeGenome.Width
6:      startLocation ← (0, 0)                    ▷ Initialize start location to top left
7:      endLocation ← (height − 1, width − 1)     ▷ Initialize target location to bottom right
8:      grid ← InitializeGrid(height, width)      ▷ Creates an empty 2D grid to host waypoints and walls
9:      GeneratePath(grid, pathGenes, startLocation, endLocation)   ▷ Generates the solution path
10:     GenerateWalls(grid, wallGenes)            ▷ Generates the internal walls
11:     return Maze(grid)
12: end function
```

Algorithm 3 Maze Grid Initialization

```
1:  function InitializeGrid(height, width)
2:      grid ← Array[width, heigth]       ▷ Each cell in 2D grid stores path and wall orientation (if any)
3:      for y = 1 : height do
4:          for x = 1 : width do
5:              grid[x, y] ← None         ▷ Denotes an empty cell
6:          end for
7:      end for
8:      return grid
9:  end function
```

Algorithm 4 Maze Path Generation

```
1:  function GeneratePath(grid, pathGenes, startPoint, endPoint)
2:      pathIdx ← 0
3:      while pathIdx ≤ pathGenes.Count do            ▷ Iterates through waypoints and builds the solution path
4:          if pathIdx = 0 then
5:              curPoint ← startPoint                 ▷ First waypoint is accessed from maze start point
6:          else
7:              curPoint ← pathGenes[pathIdx − 1]     ▷ Previous waypoint is starting location
8:          end if
9:          if pathIdx = pathGenes.Count then
10:             tgtPoint ← endPoint                   ▷ Last waypoint connects to maze end point
11:         else
12:             tgtPoint ← pathGenes[pathIdx]         ▷ Current waypoint is the target destination
13:         end if
14:         orientation ← tgtPoint.Orientation        ▷ Horizontal or vertical orientation of incoming path
15:         grid[tgtPoint.X, tgtPoint.Y].IsWaypoint ← true   ▷ Marks current grid cell as waypoint
16:         if orientation = Horizontal then          ▷ Connect path by vertical followed by horizontal segment
17:             HorizontalPathReroute(grid, curPoint, tgtPoint)
18:             GenerateVerticalPathSegment(grid, curPoint, tgtPoint)
19:             GenerateHorizontalPathSegment(grid, curPoint, tgtPoint)
20:             if curPoint.X ≠ tgtPoint.X & curPoint.Y ≠ tgtPoint.Y then
21:                 grid[curPoint.X, tgtPoint.Y].IsJuncture ← true   ▷ Set intermediate juncture for vertical-horizontal transition
22:             end if
23:         else                                      ▷ Connect path by horizontal followed by vertical segment
24:             HandleVerticalOverlapCases(grid, curPoint, tgtPoint)
25:             GenerateHorizontalPathSegment(grid, curPoint, tgtPoint)
26:             GenerateVerticalPathSegment(grid, curPoint, tgtPoint)
27:             if curPoint.X ≠ tgtPoint.X & curPoint.Y ≠ tgtPoint.Y then
28:                 grid[tgtPoint.X, curPoint.Y].IsJuncture ← true   ▷ Set intermediate juncture for horizontal-vertical transition
29:             end if
30:         end if
31:     end while
32: end function
```

Algorithm 5 Vertical Path Segment Generation

```
1:  function GenerateVerticalPathSegment(grid, curPoint, endPoint)
2:      if curPoint.Y ≤ endPoint.Y then
3:          while curPoint.Y ≤ endPoint.Y do
4:              grid[curPoint.X, curPoint.Y].PathDirection ← South
5:              curPoint.Y ← curPoint.Y + 1
6:          end while
7:      else
8:          while curPoint.Y ≥ endPoint.Y do
9:              grid[curPoint.X, curPoint.Y].PathDirection ← North
10:             curPoint.Y ← curPoint.Y − 1
11:         end while
12:     end if
13: end function
```

Algorithm 6 Horizontal Path Segment Generation

```
1: function GenerateHorizontalPathSegment(grid, curPoint, endPoint)
2:     if curPoint.X ≤ endPoint.X then
3:         while curPoint.X ≤ endPoint.X do
4:             grid[curPoint.X, curPoint.Y].PathDirection ← East
5:             curPoint.X ← curPoint.X + 1
6:         end while
7:     else
8:         while curPoint.X ≥ endPoint.X do
9:             grid[curPoint.X, curPoint.Y].PathDirection ← West
10:            curPoint.X ← curPoint.X − 1
11:        end while
12:    end if
13: end function
```

Algorithm 7 Horizontal Path Reroute

```
1: function HorizontalPathReroute(grid, curPoint, endPoint)
2:     if endPoint.Y < curPoint.Y & endPoint.X > curPoint.X then      ▷ Tracing up and to the right will overlap existing path
3:         if grid[curPoint.X + 1, curPoint.Y].PathDirection ≠ None then      ▷ Descend one unit if waypoint immediately to the right
4:             grid[curPoint.X, curPoint.Y].PathDirection ← South
5:             grid[curPoint.X, curPoint.Y + 1].IsJuncture ← true
6:             curPoint.Y ← curPoint.Y + 1
7:         end if
8:         for curPoint.X : rightmostWaypoint.X do      ▷ Move past rightmost waypoint to avoid overlapping path on ascent
9:             grid[curPoint.X, curPoint.Y].PathDirection ← East
10:        end for
11:        grid[curPoint.X, curPoint.Y].IsJuncture ← true      ▷ Repositioned start point will be a juncture
12:    end if
13: end function
```

Algorithm 8 Vertical Path Reroute

```
1: function VerticalPathReroute(grid, curPoint, endPoint)
2:     if endPoint.X < curPoint.X & endPoint.Y > curPoint.Y then      ▷ Tracing left and down will overlap existing path
3:         if grid[curPoint.X, curPoint.Y + 1].PathDirection ≠ None then      ▷ Descend one unit if waypoint immediately to the right
4:             grid[curPoint.X, curPoint.Y].PathDirection ← East
5:             grid[curPoint.X + 1, curPoint.Y].IsJuncture ← true
6:             curPoint.X ← curPoint.X + 1
7:         end if
8:         for curPoint.Y : lowestWaypoint.X do      ▷ Move past rightmost waypoint to avoid overlapping path on ascent
9:             grid[curPoint.X, curPoint.Y].PathDirection ← South
10:        end for
11:        grid[curPoint.X, curPoint.Y].IsJuncture ← true      ▷ Repositioned start point will be a juncture
12:    end if
13: end function
```

Algorithm 9 Maze Wall Generation

```
1:  function GenerateWalls(grid, wallGenes)
2:      wallGeneIdx ← 0
3:      loopIteration ← 0
4:      EncloseAdjacentPathSegments(grid, mazeHeight, mazeWidth)                ▷ Generate walls that encapsulate path
5:      partitions ← SubdivideMaze(grid, mazeHeight, mazeWidth)                 ▷ Create subdivisions adjacent to path
6:      for all partition ∈ partitions do                                       ▷ Iteratively bisect each subdivision with wall genes
7:          partitionQueue ← Queue()                                            ▷ Initialize queue for further subdividing with internal walls
8:          if partition.SupportsWalls() = true then                            ▷ Walls supported if each dimension greater than 1 unit
9:              loopIteration ← loopIteration + 1
10:             wallGeneIdx ← loopIteration%wallGenes.Count                     ▷ Cycles wall genes, looping back to beginning of gene list
11:             MarkPartitionBoundaries(grid, partition)                        ▷ Encloses partition so that it is initially inaccessible
12:             InsertPartitionOpening(grid, partition, wallGenes[wallGeneIdx]) ▷ Creates opening into partition from path
13:             partitionQueue.Enqueue(partition)
14:             while partitionQueue.Count > 0 do
15:                 curPartition ← partitionQueue.Dequeue()
16:                 loopIteration ← loopIteration + 1
17:                 wallGeneIdx ← loopIteration%wallGenes.Count
18:                 partitions ← SubdividePartition(grid, partition, wallGenes[wallGeneIdx])
19:                 if partitions[0] ≠ None then                                ▷ Enqueue partition on top/left
20:                     partitionQueue.Enqueue(partitions[0])
21:                 end if
22:                 if partitions[1] ≠ None then                                ▷ Enqueue partition on bottom/right
23:                     partitionQueue.Enqueue(partitions[1])
24:                 end if
25:             end while
26:         else
27:             MarkBoundaries(grid, partition)                                 ▷ Enclose partition and open onto path
28:         end if
29:     end for
30: end function
```

Algorithm 10 Path Encapsulation

```
1:  function EncloseAdjacentPathSegments(grid, height, width)
2:      curCell ← grid[0, 0]                                          ▷ Begin path traversal at start point
3:      while curCell.X < width & curCell.Y < height do
4:          if curCell.PathDirection ≠ North & grid[curCell.X, curCell.Y − 1].PathDirection ≠ None then
5:              grid[curCell.X, curCell.Y − 1].SouthWall ← true           ▷ Block adjacent path above
6:          end if
7:          if curCell.PathDirection ≠ South & grid[curCell.X, curCell.Y + 1].PathDirection ≠ None then
8:              grid[curCell.X, curCell.Y].SouthWall ← true               ▷ Block adjacent path below
9:          end if
10:         if curCell.PathDirection ≠ West & grid[curCell.X − 1, curCell.Y].PathDirection ≠ None then
11:             grid[curCell.X − 1, curCell.Y].EastWall ← true            ▷ Block adjacent path to the left
12:         end if
13:         if curCell.PathDirection ≠ East & grid[curCell.X + 1, curCell.Y].PathDirection ≠ None then
14:             grid[curCell.X, curCell.Y].EastWall ← true                ▷ Block adjacent path to the right
15:         end if
16:         if curCell.PathOrientation = Horizontal then
17:             if curCell.PathDirection = West then
18:                 curCell ← grid[curCell.X − 1, curCell.Y]              ▷ Next path cell is to the left
19:             else
20:                 curCell ← grid[curCell.X + 1, curCell.Y]              ▷ Next path cell is to the right
21:             end if
22:         end if
23:         if curCell.PathOrientation = Vertical then
24:             if curCell.PathDirection = North then
25:                 curCell ← grid[curCell.X, curCell.Y − 1]              ▷ Next path cell is above
26:             else
27:                 curCell ← grid[curCell.X, curCell.Y + 1]              ▷ Next path cell is below
28:             end if
29:         end if
30:     end while
31: end function
```

Algorithm 11 Maze Subdivision

```
1:  function SubdivideMaze(grid, height, width)
2:      subdivisions ← List()                                        ▷ List of maze subdivisions
3:      for y=0:height do
4:          for x=0:width do
5:              if grid[x, y].PathDirection = None & IsCellInSubdivision(grid[x, y]) = false then
6:                  startPoint ← (x, y)
7:                  endPoint ← startPoint
8:                  wallFound = false                ▷ Stops and closes off subdivision when obstruction found
9:                  while endPoint.X < width & grid[endPoint.X, endPoint.Y].PathDirection = None do
10:                     endPoint.X ← endPoint.X + 1                          ▷ Locate right edge of subdivision
11:                 end while
12:                 while wallFound = false & endPoint.Y < height do  ▷ Scan down and to the right until subdivision edge is located
13:                     endPoint.Y ← endPoint.Y + 1
14:                     curX ← startPoint.X
15:                     while wallFound = false & curX < width do
16:                         if grid[curX, endPoint.Y].PathDirection ≠ None then
17:                             wallFound ← true
18:                             endPoint.Y ← endPoint.Y − 1      ▷ Back up one unit to maintain rectangular shape
19:                         end if
20:                         curX ← curX + 1
21:                     end while
22:                 end while
23:                 subdivisions.Add(startPoint, endPoint)
24:             end if
25:         end for
26:     end for
27:     return subdivisions
28: end function
```

Algorithm 12 Partition Encapsulation

```
1:  function MarkPartitionBoundaries(grid, partition)
2:      for x = partition.X : partition.Width do
3:          if partition.Y > 0 then
4:              grid[x, partition.Y − 1].SouthWall ← true                  ▷ Mark northern boundary
5:          end if
6:          grid[x, partition.Y + partition.Height − 1].SouthWall ← true   ▷ Mark southern boundary
7:      end for
8:      for y = partition.Y : partition.Height do
9:          if partition.X > 0 then
10:             grid[partition.X − 1, y].EastWall ← true                   ▷ Mark western boundary
11:         end if
12:         grid[partition.x + partition.Width − 1, y].EastWall ← true     ▷ Mark eastern boundary
13:     end for
14:     if partition.Height = 1|partition.Width = 1 then  ▷ Single unit partition cannot support walls, so create opening
15:         if partition.X = 0 then
16:             grid[partition.X + partition.Width − 1, partition.Y].EastWall ← false
17:         else
18:             grid[partition.X − 1, partition.Y].EastWall ← false
19:         end if
20:     end if
21: end function
```

Algorithm 13 Partition Opening Placement

```
1: function InsertPartitionOpening(grid, partition, wallGene)
2:     if wallGene.Orientation = Horizontal then
3:         wallLoc ← (partition.X, partition.Y + partition.Height ∗ wallGene.wallLoc)        ▷ Scale wall to height of partition
4:         if partition.X > 0 then
5:             grid[partition.X − 1, wallLoc.Y].EastWall ← false        ▷ Create opening on left side of partition
6:         else
7:             grid[partition.Width − 1, wallLoc.Y].EastWall ← false        ▷ Create opening on right side of partition
8:         end if
9:     else
10:        wallLoc ← (partition.X + partition.Width ∗ wallGene.wallLoc, partition.Y)        ▷ Wall start location
11:        passageLoc ← (0, wallLoc.Y + partition.Height ∗ wallGene.passageLoc)        ▷ Passage start location
12:        if partition.X > 0 then
13:            grid[partition.X − 1, passageLoc.Y].EastWall ← false        ▷ Create opening on left side level with wall passage
14:        else
15:            grid[partition.Width − 1, passageLoc.Y].EastWall ← false        ▷ Create opening on right side level with wall passage
16:        end if
17:    end if
18: end function
```

Algorithm 14 Partition Subdivision

```
1: function SubdividePartition(grid, partition, wallGene)
2:     if wallGene.Orientation = Horizontal then
3:         wallLoc ← (partition.X, partition.Y + partition.Height ∗ wallGene.wallLoc)        ▷ Wall start location
4:         passageLoc ← (wallLoc.X + partition.Width ∗ wallGene.passageLoc, wallLoc.Y)        ▷ Passage start location
5:         wallLength ← partition.Width        ▷ Wall spans full length of partition
6:         for position = 0 : wallLength do        ▷ Mark horizontal wall and passage in maze grid
7:             if position ≠ passageLoc.X then
8:                 grid[wallLoc.X + position, wallLoc.Y].SouthWall
9:             end if
10:        end for
11:        childPartition1.StartPoint ← (partition.X, partition.Y)        ▷ Top partition
12:        childPartition1.Width ← partition.Width
13:        childPartition1.Height ← wallLoc.Y − partition.Y
14:        childPartition2.StartPoint ← (partition.X, wallLoc.Y + 1)        ▷ Bottom partition
15:        childPartition2.Width ← partition.Width
16:        childPartition2.Height ← partition.Y + partition.Height − wallLoc.Y
17:    else
18:        wallLoc ← (partition.X + partition.Width ∗ wallGene.wallLoc, partition.Y)        ▷ Wall start location
19:        passageLoc ← (wallLoc.X, wallLoc.Y + partition.Height ∗ wallGene.passageLoc)        ▷ Passage start location
20:        wallLength ← partition.Height        ▷ Wall spans full height of partition
21:        for position = 0 : wallLength do        ▷ Mark vertical wall and passage in maze grid
22:            if position ≠ passageLoc.Y then
23:                grid[wallLoc.X, wallLoc.Y + position].EastWall
24:            end if
25:        end for
26:        childPartition1.StartPoint ← (partition.X, partition.Y)        ▷ Left partition
27:        childPartition1.Width ← wallLoc.X − partition.X
28:        childPartition1.Height ← partition.Y
29:        childPartition2.StartPoint ← (wallLoc.X, partition.Y)        ▷ Right partition
30:        childPartition2.Width ← partition.X + partition.Width − wallLoc.X
31:        childPartition2.Height ← partition.Height
32:    end if
33:    return < childPartition1, childPartition2 >        ▷ Returns a tuple of two partitions induced by bisecting wall
34: end function
```