AI-in-the-Box: Generating Snake-in-the-Box Constructions with AI

A Major Qualifying Project (MQP) Report
Submitted to the Faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements
for the Degree of Bachelor of Science in

Computer Science,
Mathematical Sciences

By:

Warren Carstensen

Project Advisors:

Daniel Reichman
Gábor N. Sárközy

Date: March 27, 2026

Abstract

The snake-in-the-box (SIB) problem asks for the longest induced path in an n-dimensional hypercube graph. Recent advances in AI have motivated mathematicians to explore machine learning as a research tool. This work provides the first application of neural network-based methods to SIB, using the Deep Cross-Entropy Method (DCEM) and PatternBoost. PatternBoost recovered optimal snakes for $n \leq 7$ and found a snake of length 95 in dimension 8, within 3 edges of the known optimum. However, results for $n \geq 9$ reveal a widening gap between achieved lengths and current lower bounds, suggesting that general-purpose neural methods struggle with local optima and computational bottlenecks. Advancing the state of the art likely requires alternative AI approaches or significant domain-specific engineering.

Acknowledgments

This research was performed using computational resources supported by the Academic & Research Computing group at Worcester Polytechnic Institute.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recent advances in Artificial Intelligence (AI) and Machine Learning (ML) have significantly improved the ability of AI systems to complete complex tasks. This has led to an increased interest in the use of AI as a tool for mathematical discovery. Mathematics poses a unique challenge for AI tools given its exactness compared to natural language [19], multi-step reasoning, and the often large search spaces the problems occupy.

Computers have been assisting mathematicians for nearly as long as they have existed. For example, in 1956 Newell, Simon and Shaw used their "logic theorist" program to prove a number of theorems from *Principia Mathematica* [12]. This is widely regarded as one of the first attempts to perform automated reasoning and as one of the first AI programs. Since then, a variety of computational tools have been developed to assist mathematicians such as SMT solvers and proof assistants like Lean. More recently neural networks, transformers, and reinforcement learning have been applied to a number of mathematical tasks such as automated theorem proving and combinatorial optimization [14, 23]. The use of AI for mathematics research has garnered significant mainstream attention in the past few years. Methods such as *FunSearch* have achieved impressive results and have even assisted in solving a variety of open problems in mathematics [17]. Most recently, Terence Tao claimed that a language model had "more or less autonomously" solved an open problem of Erdös [22].

This work focuses on using AI to tackle a famous NP-hard discrete optimization problem called the "snake in the box" (SIB) problem, or the task of finding the length of the longest induced path in a given n-dimensional hypercube graph. There is also a related problem, called the "coil in the box" problem, which considers the length of the longest induced chordless cycle in the hypercube. The SIB problem was first introduced in 1958 by William H. Kautz, when studying error-detecting codes for analog-to-digital conversion [9]. He combined Hamming's Single-Error-Detection (SED) property and Gray codes to create a new type of error-detecting code, essentially a SED Gray code. These codes have the properties that moving from one value of the code to the next requires exactly one bit flip (Gray code property), and all non-successive values differ by more than one bit flip (SED property) [6, 8]. These types of codes have a variety of applications in cryptography, electrical engineering, and network engineering [9, 25]. Because longer codes represent more states within the same n-bit memory constraint, the maximum length code for a given n represents the optimal

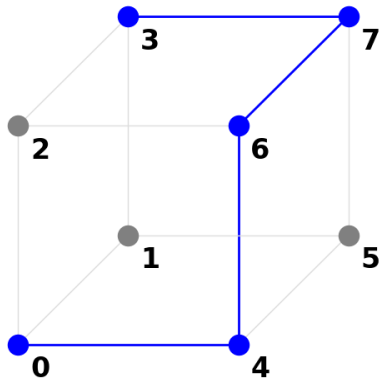

Figure 1.1: Graph of a solution to the SIB problem in dimension 3.

information density for a SED Gray code. The longest possible SED Gray code for a given bit length n corresponds to the vertices of a solution to the SIB in the n-cube. An example of the solution to the snake in the box problem can be seen in Figure 1.1.

Shortly after the problem was proposed, work focused on providing bounds for a general solution to the snake and coil in the box problems [16, 11] and constructing paths using computational tools. Initial computational approaches used traditional depth first search (DFS) algorithms augmented with complex pruning rules to increase search efficiency in the large search space [5]. Davies quickly enumerated all snakes in $d \leq 6$, but it took until 1994 for the problem to be solved in dimension 7 and until 2015 for dimension 8 using sophisticated DFS algorithms [15, 13]. During this time, improvements on the lower bounds to the SIB problem represented a significant shift in the nature of the techniques used to find long snakes [2, 1]. Evolutionary algorithms and heuristic searches had become the state-of-the-art for constructing snakes as traditional DFS remained computationally infeasible for greater dimensions.

The problem has remained open for dimensions $n \geq 9$. Lower bounds in dimensions 9, 10, 11, 12 and 13 have recently seen improvement, with much of the recent literature making use of new searches and metaheuristic algorithms [24, 4].

In this work, we present the **first attempts at using recent neural machine learning methods to generate new constructions for the SIB problem**, including frameworks like the Deep Cross Entropy Method (DCEM) and PatternBoost [23, 3]. The DCEM is our "standard" reinforcement learning approach where for thousands of iterations a neural network generates many constructions and then is trained on its own best constructions. PatternBoost is a two phase transformer-based framework where a transformer is trained on a dataset of constructions and then generates many new constructions autoregressively which serve as a starting point for a simpler local search such as a greedy or heuristic search. We also present, to our knowledge, the first formulation of PatternBoost which makes use of Tabu Search and Tabu-Assisted Simulated Annealing for the local search phase of the

Table 1.1: Previously Known and Our Achieved Longest Snake Lengths by Dimension

Dimension	**Previous Best**	**Achieved (This Work)**	
		Length	% of Best
3	4	**4**	**100%**
4	7	**7**	**100%**
5	13	**13**	**100%**
6	26	**26**	**100%**
7	50	**50**	**100%**
8	98	95	96.9%
9	190	173	91.1%
10	373	296	79.4%
11	721	443	61.4%

algorithm [7].

We demonstrate that these neural methods can recover known optimal snakes in dimensions $n \leq 7$ and get within 3 edges (or 3.06%) of finding a known optimal snake in dimension 8 with little domain-specific engineering. Additional experiments in dimensions $9 \leq n \leq 11$ show that neural methods struggle as dimension grows, suggesting a need for alternative AI approaches to this task. A summary of our results can be found in Table 1.1.

Chapter 2

Background

2.1 Graphs

Definition 2.1. An undirected graph is a pair $G = (V, E)$ where V is a set of vertices, and E is a set of edges, or unordered pairs of vertices $\{v_1, v_2\}$ with $v_1 \neq v_2$.

For an edge $\{v_1, v_2\}$, the vertices v_1 and v_2 are the *endpoints* of that edge and they are said to be *incident* to that edge. Additionally, v_1 and v_2 are considered *adjacent*. The *degree* of a vertex is equal to the number of edges the vertex is incident to. A graph is *connected* if for every pair of vertices u, v there exists a sequence of vertices $v_1, v_2, ..., v_n$ such that $v_1 = u$, $v_n = v$ and $\{v_i, v_{i+1}\} \in E$ for each $i = 1, ..., n - 1$.

2.1.1 Paths

Given a formal definition of a graph and related terminology, another necessary preliminary is the idea of a path.

Definition 2.2. A graph is a *path* if the graph is connected and there are exactly two vertices with degree equal to one, and all other vertices have a degree of two. Equivalently, a graph is a path if the vertices can be ordered such that

$$\{v_i, v_{i+1} \mid i = 1, 2, \ldots, |V| - 1\} = E$$

A path with order n has size $n - 1$, and the length of a path is equal to its size. The path graph on n vertices can be denoted as P_n. Figure 2.1 shows P_5 as an example of a path graph.

2.1.2 Subgraphs

Along with paths, subgraphs are another necessary preliminary.

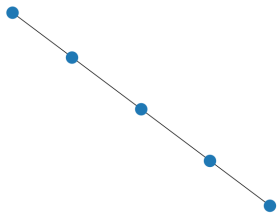

Figure 2.1: P_5

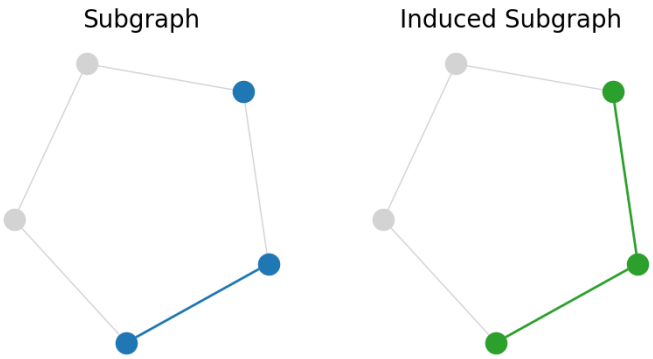

Figure 2.2: A subgraph and an induced subgraph of the cycle on 5 vertices.

Definition 2.3. A *subgraph* is a graph $G_S = (V_S, E_S)$ such that $V_S \subset V$ and $E_S \subset E$ where (V, E) is also a graph. Additionally, it is required that for each $e \in E_S$ both endpoints of e are in V_S.

A subgraph is said to be *induced* if $E_S = \{\{u, v\} \mid u, v \in V_S\} \cap E$. In simpler terms, a subgraph is induced if the edge set of the subgraph contains all edges from the original graph for which both endpoints are in the subgraph. Figure 2.2 shows both a non-induced and an induced subgraph.

2.1.3 Graph Isomorphisms

Next we introduce graph isomorphisms, which we later use to drastically decrease our search space.

Definition 2.4. Two graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ are *isomorphic* if there exists a bijection $f : V_G \to V_H$ such that $\{u, v\} \in E_G$ if and only if $\{f(u), f(v)\} \in E_H$.

Essentially, an isomorphism is a bijection which preserves adjacency in both graphs. A graph isomorphism is an equivalence relation (meaning the operation is reflexive, symmetric, and transitive) and therefore partitions all graphs into distinct equivalence classes.

Definition 2.5. A *graph automorphism* is an isomorphism from a graph to itself, meaning it is a permutation of the vertices of a graph which preserves adjacency.

Similarly, a graph automorphism represents an equivalence relation which partitions the set of all subgraphs into disjoint equivalence classes.

2.2 The Hypercube

Definition 2.6. A n-dimensional hypercube graph, denoted Q_n, is defined as (V_n, E_n) where

$$V_n = \{0,1\}^n$$

and

$$E_n = \{(u,v) \mid u,v \in V_n,\ d_H(u,v) = 1\}$$

where d_H is the Hamming distance, or the number of positions where the bit strings u and v differ.

Since the vertex set V_n is the collection of all bit strings of length n, we can equivalently represent each vertex by its decimal value, identifying V_n with the set $\{0,1,\ldots,2^n-1\}$.

Clearly the hypercube has order 2^n. Since the each vertex has n bits, then each vertex also has n neighbors, one for each bit. Therefore, Q_n has size $n2^n/2 = n2^{n-1}$, by the Handshaking Lemma ($\Sigma deg(v) = 2|E|$).

2.2.1 Automorphisms of the Hypercube

Theorem 2.1. *Permutations of bit indices (dimensions of the hypercube) and translations of vertices are both isometric (Hamming distance preserving) automorphisms of the hypercube.*

Proof. **Translations:** Let $f_v : V_n \to V_n$ be a translation in the hypercube defined by $f_v(x) = x \oplus v$ for some vertex v and all vertices x. Here $\oplus$ is the binary XOR operation. Notice that $f_v(f_v(x)) = (x \oplus v) \oplus v = x$ for all vertices x. Therefore, f is an involution and also a bijection. Additionally, given two vertices u, v, then clearly $d_H(u_1, u_2) = d_H(u_1 \oplus v, u_2 \oplus v)$, as the XOR operation flips the same bits in both u_1 and u_2. This means that the operation is also an isometry. If u and v are adjacent, meaning $d_H(u_1, u_2) = 1$ then so are $f_v(u_1)$ and $f_v(u_2)$ because $d_H(u_1 \oplus v, u_2 \oplus v) = 1$, and vice versa. Thus f_v is an adjacency preserving bijection and an automorphism of Q_n.

Permutations: Let $\sigma : \{0,\ldots,n-1\} \to \{0,\ldots,n-1\}$ be a permutation of the dimensions of the n-cube. Since σ is a permutation it must also be a bijection as it is invertible, let the inverse of σ be σ^{-1}. Let the binary representation of any vertex a be $b_0^{(a)}, b_1^{(a)}, \ldots b_{n-1}^{(a)}$. Define $g_\sigma : V_n \to V_n$ as $g_\sigma(x) = y$ such that $b_k^{(x)} = b_{\sigma(k)}^{(y)}$ for $k = 0, 1, \ldots, n-1$. Notice that if we define $g_{\sigma^{-1}}$ in the same way, then $g_{\sigma^{-1}}(g_\sigma(x)) = x$ and $g_\sigma(g_{\sigma^{-1}}(x)) = x$ for all vertices x. This shows that g_σ is a bijection. Once again we can also show that this operation is also an isometry. If we permute the bit indices of two vertices u, v with g, then we can see $d_H(u_1, u_2) = d_H(g(u_1), g(u_2))$, as the permutation operation simply changes the order of the bits in both u_1 and u_2 and not their value. Here, given two adjacent vertices u_1, u_2

then the binary representations of u_1 and u_2 must differ by a single bit in the $i = \log_2(u_1 \oplus u_2)$ position. $g_\sigma(u_1)$ and $g_\sigma(u_2)$ also differ by a single bit, this time in the $\sigma(i)$ position. The same procedure can show that for non-adjacent vertices, they remain non-adjacent after g_σ. Therefore, g_σ is an adjacency preserving bijection and an automorphism of Q_n. □

Any composition of these two operations, translation and permutation, is also an automorphism of the hypercube. This forms the entire automorphism group, as translations and permutations account for all symmetries of the hypercube. Since there are 2^n possible translations, and $n!$ possible permutations, the automorphism group has order $2^n \cdot n!$. Consequently, any two subgraphs that can be transformed into one another through these operations are considered structurally identical, and the set of all such related subgraphs forms an equivalence class.

From these operations, we can easily prove two useful properties of Q_n. First, the hypercube is vertex-transitive, meaning that given any two vertices u, v there exists an automorphism f such that $f(u) = v$.

Proof. Let two nodes x, y be given. Find a vertex v such that $v = x \oplus y$ and consider $f_v : V_n \to V_n$ as defined above. We know that f_v is an automorphism of Q_n. Here, notice that $f_v(x) = x \oplus v = x \oplus (x \oplus y) = y$. Since the automorphism f_v maps an arbitrary vertex x to an arbitrary vertex y then Q_n is vertex-transitive. □

Additionally, we can show that Q_n is also edge-transitive, meaning that given any two edges $e = \{u_1, v_1\}, g = \{u_2, v_2\}$ there exists an automorphism f such that $f(u_1) = u_2$ and $f(v_1) = v_2$.

Proof. Let $e_1 = \{u_1, v_1\}, e_2 = \{u_2, v_2\}$ be given and $\sigma : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ be a permutation of the dimensions of the n-cube such that if $i = \log_2(u_1 \oplus v_1)$ and $j = \log_2(u_2 \oplus v_2)$ then $\sigma(i) = j$. This permutation maps the position of the bit flipped when traversing e_1 to the position of the bit flipped when traversing e_2. Consider the automorphism g_σ as defined above. From the previous proof, find two other automorphisms f_{u_1} and f_{u_2}. Consider $h = f_{u_2} \circ (g_\sigma \circ f_{u_1})$. The function h is the composition of automorphisms and therefore an automorphism itself. Further, notice that $f_{u_1}(u_1) = 0$ and $f_{u_1}(u_1), f_{u_1}(v_1)$ are adjacent. They differ by a bit in the position i, meaning $f_{u_1}(v_1) = 2^i$. Then by applying the permutation we get, $g_\sigma(0) = 0$ and $g_\sigma(2^i) = 2^{\sigma(i)} = 2^j$. Lastly, by applying our final translation we find $f_{u_2}(0) = u_2$ and $f_{u_2}(2^j) = u_2 \oplus 2^j = u_2 \oplus (2^{\log_2(u_2 \oplus v_2)}) = u_2 \oplus (u_2 \oplus v_2) = v_2$. So h is an automorphism which maps u_1 to u_2 and v_1 to v_2. □

These two properties, vertex- and edge-transitive, become important later when discussing "normalizing" snake representations.

2.2.2 Snakes, Node Characterizations, and Snake Equivalence

Now we can formally define a snake as it relates to the SIB problem.

Definition 2.7. A *snake* is an induced subgraph of Q_n such that the subgraph is a path, or simply an "induced path".

A vertex in a snake with a degree equal to one is called either the head node or the tail node, and a vertex with degree two is called a body node. Each vertex in Q_n that is not in the snake and is adjacent to a vertex in the snake is called a skin node. Additionally, nodes which are adjacent to the head or tail node but are not adjacent to any other nodes in the snake are often called a *valid extension* of the snake. If a node is not characterized as a path node or skin node, it is called *available*. Two snakes S_1 and S_2 are said to be *equivalent* if there exists an automorphism of Q_n, f, such that $f(S_1) = S_2$, because these two snakes must belong to the same equivalence class.

2.2.3 Snake Representations

There are three main ways to represent a snake used in this work. Consider a snake $S = (V_S, E_S)$.

Node Sequences: A *node sequence* (NS) is an ordered list of vertices $[n_0, n_1, \ldots, n_{m-1}]$ such that $V_S = \{n_0, n_1, \ldots, n_{m-1}\}$ and $E_S = \{\{n_i, n_{i+1}\} \mid i = 0, 1, \ldots, m-2\}$. This representation is just a list of nodes the snake visits in the order they appear in the path starting at the tail. A node sequence contains all information necessary to reconstruct the entire snake. As an example, the NS of the path shown in Figure 1.1 would be [0,4,6,7,3], as the snake starts at node 0 and ends at node 3. The length of a snake defined by a NS with m elements is $m - 1$.

A NS can start from any node, and because Q_n is vertex-transitive, then a node sequence can be "normalized" so the tail node is 0.

Transition Sequences: Given a NS $[n_0, n_1, \ldots, n_{m-1}]$, a *transition sequence* (TS) is an ordered list defined as $[\log_2(n_0 \oplus n_1), \log_2(n_1 \oplus n_2), \ldots, \log_2(n_{m-2} \oplus n_{m-1})]$. Notice that every value in a TS is in $\{0, 1, \ldots, n-1\}$. In simple terms a TS is a sequence of dimensions the snake travels. Since this snake representation method does not capture any information about the position of the snake in the hypercube, only the dimensions it travels, the path cannot be fully reconstructed from a TS alone; a starting node is also needed. Once again, because Q_n is vertex-transitive this is not an issue, and if needed the starting node can simply be set to 0. The transition sequence for the path shown in Figure 1.1 would be [2,1,0,2].

Similarly to how an NS can be easily normalized to start at 0, since every permutation of the dimensions of Q_n is an automorphism of the cube, then a TS can also be manipulated such that it begins by traversing dimension 0, and we can also control in what order each other dimension is traversed.

Canonical Form: A TS $[d_0, d_1, \ldots, d_{m-1}]$ is in *canonical form* if given two dimensions i and j with $i \leq j$ such that d_k is the first occurrence of i in the TS and d_l is the first occurrence of j then $k \leq l$. In simpler terms, a TS is in canonical form if the first occurrences of each dimension occur in order (e.g. 0 appears before 1, which appears before 2 and so on). Thus, the canonical form *of* a TS is an equivalent TS which is in canonical form.

Algorithm 1 Finding Canonical Form

Require: $TS \leftarrow$ transition sequence of a path with n dimensions and length m
Ensure: $CF \leftarrow$ the canonical form of TS
  $M \leftarrow$ empty list of size n
  $I \leftarrow 0$
  for $i = 0, 1, \ldots, m-1$ **do**
    if $M[TS[i]]$ is empty **then**
      $M[TS[i]] \leftarrow I$
      $I \leftarrow I + 1$
    end if
  end for
  $CF \leftarrow$ list of size m with $CF[i] = M[TS[i]]$ for all $i = 0, 1, \ldots, m-1$

We present an algorithm for finding a canonical form of a snake in Algorithm 1. This algorithm constructs a permutation of the dimensions that gives us the canonical form property. Given that permuting dimensions of the cube is an automorphism of Q_n, we know that the sequence CF is equivalent to the original snake TS.

Clearly, reversing the order in which the dimensions are listed does not change the structure of the underlying path, but it does provide us with a different representation which has a different canonical form. To account for this, we also define a *complete canonical form* of a TS as the lexicographic minimum between the canonical form of the TS and the canonical form of the reversal of the TS.

Theorem 2.2. *If two snakes have the same complete canonical form, then they are equivalent.*

Proof. Let two snakes S_1 and S_2 be given with TS representations $c = [c_0, c_1, \ldots, c_{m-1}]$ and $d = [d_0, d_1, \ldots, d_{m-1}]$ respectively. Using Algorithm 1, we can construct permutations $\sigma_c : \{0, 1, \ldots, n-1\} \to \{0, 1, \ldots, n-1\}$ and $\sigma_d : \{0, 1, \ldots, n-1\} \to \{0, 1, \ldots, n-1\}$ such that

$$c_{CF} = [\sigma_c(c_0), \sigma_c(c_1), \ldots, \sigma_c(c_{m-1})]$$

and

$$d_{CF} = [\sigma_d(d_0), \sigma_d(d_1), \ldots, \sigma_d(d_{m-1})]$$

are in canonical form. Apply the same process to $[c_{m-1}, c_{m-2} \ldots, c_0]$ and $[d_{m-1}, d_{m-2} \ldots, d_0]$ to find σ_{c^R} and σ_{d^R} such that

$$c^R_{CF} = [\sigma_{c^R}(c_{m-1}), \sigma_{c^R}(c_{m-2}) \ldots, \sigma_{c^R}(c_0)]$$

and

$$d^R_{CF} = [\sigma_{d^R}(d_{m-1}), \sigma_{d^R}(d_{m-2}) \ldots, \sigma_{d^R}(d_0)]$$

are in canonical form. Find the lexicographic minimum of both sequences for c and d, or their complete canonical forms. These complete canonical forms are identical. We have already shown that permutations of the dimensions of the cube are automorphisms, so clearly the

original snakes S_1 and S_2 are equivalent to their complete canonical form.
Case 1 if $c_{CF} = d_{CF}$, then the permutation $\sigma_d^{-1} \circ \sigma_c$ maps c to d and therefore S_1 and S_2 are equivalent.
Case 2 if $c_{CF} = d_{CF}^R$ then $\sigma_{d^R}^{-1} \circ \sigma_c$ maps c to d^R and once again S_1 and S_2 are equivalent. □

2.2.4 Current Bounds for the SIB Problem

Table 2.1: Known and Proven Lower Bounds by Dimension

Dimension	Previous Best	Found By
3	4	
4	7	
5	13	
6	26	Davies, 1965 [5]
7	50	Potter et al., 1994 [15]
8	98	Ostergard et al., 2015 [13]
9	190	Wynn, 2012 [24]
10	373	Dang et al., 2023 [4]
11	721	Dang et al., 2023 [4]
12	1383	Dang et al., 2023 [4]

The bounds provided in Table 2.1 represent the length of the longest known snake construction in a given dimension. Likewise, this work is an attempt to improve these lower bounds by providing longer snake constructions. Conversely, several works establish theoretical upper bounds on the coil-in-the-box problem [20, 26]. This bound is given by

$$|S| \leq 1 + 2^{n-1} \frac{6n}{6n + \frac{1}{6\sqrt{6}} n^{1/2} - 7} \leq 2^{n-1} \left(1 - \frac{1}{89n^{1/2}} + O\left(\frac{1}{n}\right)\right).$$

where S is a coil.

Chapter 3

Results

In this chapter, we summarize the main findings of this work. We first present high-level takeaways, then report results for heuristic searches and learning algorithms separately, and conclude with a discussion of computational cost. Detailed experimental configurations and per-experiment analysis can be found in Chapter 6. A summary of our best results by dimension is shown in Table 3.1.

3.1 Overview

To our knowledge, this work represents **the first application of neural network-based methods to the SIB problem**. Specifically, we apply the Deep Cross-Entropy Method (DCEM) and the PatternBoost framework to the task of constructing long induced paths in hypercube graphs. We also present what we believe to be the first formulation of PatternBoost using Tabu Search and Tabu-Assisted Simulated Annealing as local search algorithms, as opposed to simpler greedy searches.

Despite these novel contributions, none of our methods produced new record-length snakes in any open dimension. Our heuristic searches recovered known optimal snakes in dimensions $n \leq 7$, and **we focused a vast majority of compute attempting to replicate the only remaining solved case**, dimension 8. Our best result in dimension 8, a snake of length 95, found by PatternBoost, falls 3 edges short of the known optimum of 98 (a gap of 3.06%). In dimensions 9-11, the gap between our results and the known best grows substantially, from roughly 10% in dimension 9 to nearly 39% in dimension 11. For each dimension $n \geq 8$, only a single non-equivalent longest snake was found.

These results suggest that general-purpose neural methods, while effective for other combinatorial optimization problems, struggle significantly on the SIB problem without considerable domain-specific engineering. We invested substantial computational resources, running experiments near-continuously from October 2025 through early January 2026 across CPU and GPU nodes mostly in dimension 8, and could not replicate even known constructions in this dimension. This points to a fundamental flaw with these methods applied to this task,

Dimension	Known Best	This Work	% of Best	Best Method
3	4	4	100%	Heuristic Searches
4	7	7	100%	Heuristic Searches
5	13	13	100%	Heuristic Searches
6	26	26	100%	Heuristic Searches
7	50	50	100%	Heuristic Searches
8	98	95	96.9%	PatternBoost
9	190	173	91.1%	PatternBoost
10	373	296	79.4%	TA Simulated Annealing
11	721	443	61.4%	PatternBoost

Table 3.1: Best snake lengths achieved in this work by dimension, compared to known best lengths. The rightmost column indicates the method which produced the longest snake in each dimension.

rather than a shortage of compute.

3.2 Heuristic Algorithms

Dimension	Known Best	Tabu Search	Sim. Annealing	TA Sim. Annealing
7	50	50	43	39
8	98	90	80	74
9	190	158	156	168
10	373	297	276	296
11	721	526	479	581

Table 3.2: Longest snake found by each standalone heuristic search algorithm per dimension.

A summary of results of our heuristic algorithms can be found in Table 3.2.

3.2.1 Simple vs. Restricted Search

Across all algorithms and dimensions tested, restricted searches uniformly and significantly outperformed simple searches. Simple searches frequently produced heavily chorded (invalid) paths. Even with strong penalization for chords, the proportion of valid snakes generated was extremely low. In particular, DCEM trained from random weights under the simple search variant failed to generate any valid snakes at all. Based on these findings, all subsequent experiments used the restricted search variant exclusively. A detailed comparison is presented in Subsection 6.2.1.

3.2.2 Scoring Function

We evaluated four scoring functions (length, tightness, mobility, and a linear interpolation between tightness and mobility) across all heuristic algorithms in dimension 8. No single scoring function was uniformly superior. The best-performing scoring function varied by algorithm: tightness produced the longest snake for Tabu Search, while length was best for Simulated Annealing and Tabu-Assisted Simulated Annealing. Detailed experiments are presented in Section 6.2.2.

3.2.3 Exploration Behavior

Tabu Search and Simulated Annealing both converged quickly to local optima and spent the vast majority of their iterations revisiting structurally equivalent states. Tabu-Assisted Simulated Annealing explored far more non-equivalent states, though its maximum snake lengths were oftentimes shorter than those found by the other two algorithms. This demonstrates a key tradeoff between exploration and exploitation in any intelligent search algorithm. It is difficult to take advantage of any strong characteristics of the current construction while still making sure we are not wasting our time on a suboptimal path. See Section 6.2.3 for a full experimental analysis on the exploration behavior of our heuristic search algorithms.

3.3 Learning Algorithms

3.3.1 DCEM

DCEM experiments focused on dimension 8 using the LSTM architecture with restricted search. The method exhibited rapid early improvement but plateaued quickly, reaching a maximum snake length of 92 in dimension 8. This falls 3 edges short of our best PatternBoost result and 5 edges short of the known optimum. DCEM was not run in dimensions $n \leq 7$, as heuristic searches had already recovered optimal snakes in those dimensions at substantially lower computational cost. DCEM under the simple search variant, trained from random weights, failed to produce any valid snakes.

3.3.2 PatternBoost

PatternBoost was our most successful method overall, producing the longest snakes in dimensions 8, 9, and 11. Table 3.3 summarizes the best results by dimension and local search algorithm.

In dimension 8, both local search variants (Tabu Search and Tabu-Assisted Simulated Annealing) independently found snakes of length 95. The majority of our PatternBoost compute was allocated to dimension 8, where tens of thousands of iterations were completed over

Dimension	Known Best	Tabu Search	TA Sim. Annealing
8	98	95	95
9	190	—	173
10	373	—	276
11	721	—	443

Table 3.3: Longest snake found by PatternBoost per dimension and local search algorithm. Dashes indicate experiments that were not run.

at most ∼1.5 months of wall-clock time. Far less compute was spent on higher dimensions. In dimensions 9 and 10, several hundred to a few thousand iterations were completed; in dimension 11, only hundreds of iterations were feasible within the time budget.

3.3.2.1 Snake Representation

We tested PatternBoost with both node sequence and complete canonical form (CCF) transition sequence representations. The transformer trained on node sequences began generating valid snakes much earlier in training compared to the CCF representation, which required substantially more iterations before producing non-trivial valid constructions. As a result, all best PatternBoost results reported in this work used node sequence representations.

3.3.2.2 Local Search Bottleneck

The computational bottleneck in PatternBoost was the local search phase, not transformer inference. Generating ∼100,000 candidate snakes via the transformer took at most an hour per iteration using batching on a single A100 GPU. However, running the local heuristic search on each valid candidate dominated per-iteration cost, often requiring many hours even with heavy parallelization across available CPU cores. This bottleneck worsened as training progressed, and the transformer produced a higher proportion of valid candidates, each of which required local search processing.

3.4 Compute Requirements

All experiments were conducted on WPI's Turing high-performance computing cluster. GPU experiments used a single NVIDIA A100 GPU per run; CPU experiments requested many cores for parallelized heuristic search. Experiments ran near-continuously from October 2025 through early January 2026, spanning approximately 3.5 months of active computation.

The bulk of compute was devoted to dimension 8, where PatternBoost alone ran for approximately 1.5 months of wall-clock time. Individual runs in higher dimensions were shorter, on the order of 1-2 weeks per configuration, but still represented significant resource

usage. Heuristic search experiments were comparatively inexpensive per iteration but were executed in large quantities (e.g., often over 10,000 instances of Tabu-Assisted Simulated Annealing for 40,000 iterations each, or 400,000,000 total iterations, per single PatternBoost iteration).

Despite this substantial investment, our methods did not produce new record-length snakes in any open dimension. This shows that the SIB problem presents a qualitatively different challenge from the combinatorial optimization tasks where these neural methods have previously succeeded, and suggests that significant domain-specific engineering or different algorithmic approaches may be necessary to advance the state of the art.

Chapter 4

Methodology

4.1 Search Space

Each algorithm we consider constructs snakes *autoregressively*, meaning the path is built one node at a time. Starting from some initial path (maybe just the origin $\mathbf{0}$, or an existing long path in a lower dimension), the algorithm repeatedly selects a new node to append to the current path.

Most searches were conducted using node sequence representation of snakes. At each step, the set of *candidate nodes* consists at the very least of all skin nodes adjacent to the head or the tail of the snake. Depending on the search variant (see Subsection 4.1.1), additional constraints may be imposed on this candidate set. Additional searches were conducted using transition sequences and those in canonical form. In this case, the candidate sets contained dimensions corresponding to current candidate nodes.

All the search algorithms described in this chapter, except simulated annealing, are depth-first, meaning the search progresses by extending the current path as far as possible before reaching a state where the set of candidate nodes is empty. At this point, the behavior of our algorithms differ, with some having backtracking implemented and some moving on to other constructions after a dead end is reached.

4.1.1 Simple and Restricted Searches

We define two variants of the search space, simple and restricted, which were implemented separately for each algorithm.

- **Simple Search**: The candidate set at each step includes all skin nodes adjacent to the head or tail that are not already in the path. This means the algorithm may construct paths that contain chords, i.e., paths that are not valid snakes. The resulting path must be checked for validity after construction, or the longest chord-free prefix must be extracted. In these approaches, invalid snakes are heavily penalized which provides strong pressure for each algorithm to generate valid snakes.

- **Restricted Search**: The candidate set at each step includes all skin nodes whose only neighbor in the current path is the head or the tail. This guarantees that every path produced is a valid snake by construction. Here, the tradeoff is that the search space is very strongly constrained, but we have the added guarantee that all snakes are valid.

We use CANDIDATES(P, n) to represent the candidate set for a given path P in dimension n. This set of course varies based on search variant.

4.1.2 Search Initialization

Each search is initialized with a known long snake from dimension $n-1$. This provides the search with a strong initial construction, reducing the time spent exploring short, uncompetitive paths, and preventing the search from immediately running into a dead end. For dimensions where no prior construction was available, searches were initialized with a single node (just the origin $\mathbf{0}$).

4.1.3 Scoring Function

We define a *scoring function* $f(P, v)$ that evaluates the desirability of appending candidate node v to the current path P. The specific form of f varies by algorithm, but common choices include:

- **Length**, simply $f(P, v) = |P| + 1$.
  - There is also a variant of this for simple searches where we subtract the number of chords in a path, augmented by some weight. Formally, $f(P, v) = |P| + 1 - \alpha|C|$ where C is the set of all chords in $P \cup \{v\}$ and $\alpha \in \mathbb{R}$ is some weight. Intuitively, the penalty for chords should be greater than the reward for adding valid edges, so we often chose $1 \leq \alpha \leq 3$.
- **Tightness**, which measures how tightly the snake is packed into the hypercube, or the percentage of nodes that would remain available after appending v to the path. More specifically, in an n-dimensional hypercube if P' is the resulting snake after appending v to the path and S is the set of skin nodes for P', then the tightness is

$$f(P, v) = 1 - \frac{|P'| + |S|}{2^n}$$

  Given two valid snakes with the same length, this score will be higher for the snake which occupies less of the hypercube (higher percentage of available nodes) and lower for the snake which occupies more space in the hypercube (lower percentage of available nodes).

- **Mobility**, how free the snake is in the hypercube. Formally, we can define

$$f(P, v) = |\textsc{Candidates}(P \cup \{v\}, n)|$$

  This essentially provides the search a one step look-ahead by scoring candidate nodes based on the number neighboring states the resulting snake would have.

4.2 Heuristic Algorithms

The heuristic algorithms described in this section use relatively simple, efficiently computable metrics to determine which candidate node to append at each step. These metrics serve as proxies for the long-term quality of the resulting snake, guiding the search toward longer constructions without requiring complex learned models. Because these algorithms rely only on simple bitwise computations and set/list operations at each step, the computational runtime of searches are CPU-bound and can execute rapidly, making them well-suited for generating large numbers of constructions.

4.2.1 Tabu Search

Tabu search is a metaheuristic that maintains a *tabu list* or a fixed-length list of recently visited nodes or recently taken actions that are temporarily forbidden. The purpose of the tabu list is to prevent the search from cycling back to recently explored regions of the search space, encouraging exploration and helping the algorithm escape local optima [7].

At each step, the algorithm scores all non-tabu candidate nodes using the scoring function f and selects the highest-scoring candidate. When the path reaches a dead end (no valid, non-tabu candidates remain), the algorithm backtracks by removing the most recently added node and adding it to the tabu list. The tabu list operates as a fixed-size queue: when a new entry is added and the list is full, the oldest entry is removed.

The key hyperparameters are the tabu tenure T (the size of the tabu list) and the scoring function f.

4.2.2 Simulated Annealing

Our implementation of simulated annealing adapts the classical framework to an autoregressive snake construction setting [10]. At each step, the algorithm selects a candidate node uniformly at random from the candidate set. If appending this node results in a path at least as long as the current best, the move is accepted. If the move does not improve the current path (e.g., after a backtrack), it is accepted with a probability that depends on the current temperature parameter τ, which decreases over time according to a cooling schedule.

This probabilistic acceptance of non-improving moves allows the algorithm to escape local optima early in the search (when τ is high) while converging toward greedy behavior as the search progresses (when τ is low).

Algorithm 2 Tabu Search

Require: $n \leftarrow$ number of dimensions
Require: $T \leftarrow$ tabu tenure (size of tabu list)
Require: $f(P, v) \leftarrow$ scoring function
Require: $t_{\max} \leftarrow$ maximum number of steps
Require: $S \leftarrow$ some starting path ([**0**] or long snake in dimension $n-1$)
Ensure: $P^* \leftarrow$ longest snake found

```
P ← S
P* ← P
T ← ∅                                              ▷ Initialize tabu list
for t = 1, 2, ..., t_max do
    C ← CANDIDATES(P, n) \ T                       ▷ Non-tabu candidates
    if C ≠ ∅ then
        v* ← arg max_{v∈C} f(P, v)                 ▷ Randomly select one for tiebreaker
        Append v* to P          ▷ Appended to head or tail, whichever v* is adjacent to
        if |P| > |P*| then
            P* ← P
        end if
    else
        v_removed ← most recently added node of P
        Remove v_removed from P
        Add v_removed to T
        if |T| > T then
            Remove oldest entry from T
        end if
    end if
end for
```

Specifically, at each step we can take one of two actions: extend the current path by appending a candidate node, or backtrack by removing the last node. One of these actions is selected uniformly at random. If the action produces a longer path, it is always accepted. Otherwise, it is accepted with probability $e^{-1/\tau}$. The temperature is updated as $\tau \leftarrow \tau \cdot \alpha$ after each step, where $\alpha \in (0, 1)$ is the cooling rate.

Algorithm 3 Simulated Annealing

Require: $n \leftarrow$ number of dimensions
Require: $\tau_0 \leftarrow$ initial temperature
Require: $\alpha \leftarrow$ cooling rate, $0 < \alpha < 1$
Require: $t_{\max} \leftarrow$ maximum number of steps
Require: $f(P, v) \leftarrow$ scoring function
Require: $S \leftarrow$ some starting path ([**0**] or long snake in dimension $n - 1$)
Ensure: $P^* \leftarrow$ longest snake found

```
P ← S
P* ← P
τ ← τ_0
for t = 1, 2, ..., t_max do
    C ← CANDIDATES(P, n)
    a ← uniformly random action from {EXTEND, BACKTRACK}
    if a = EXTEND and C ≠ ∅ then
        v ← arg max_{v∈C} f(P, v)
        P' ← P with v appended
    else if a = BACKTRACK and |P| > 1 then
        P' ← P with most recently added node removed
    else
        continue                ▷ Dead end or snake with length 1, skip this iteration
    end if
    Δ ← |P'| − |P|              ▷ Either +1 or -1
    if Δ ≥ 0 then
        P ← P'                  ▷ Accept extending move
    else if random(0, 1) < e^{−1/τ} then
        P ← P'                  ▷ Accept non-extending move probabilistically
    end if
    if |P| > |P*| then
        P* ← P
    end if
    τ ← max(τ · α, τ_min)
end for
```

4.2.3 Tabu-Assisted Simulated Annealing

Tabu-Assisted Simulated Annealing combines simulated annealing with a tabu list. At each step, the algorithm randomly chooses to either extend the snake by appending a can-

didate node, or backtrack by removing a node from the head or tail. Extension moves are always accepted if the candidate is not tabu; if the candidate is tabu, it is accepted with probability proportional to how close it is to aging off the tabu list. Backtracking is governed by the standard simulated annealing acceptance criterion: non-improving backtracks are accepted with probability $e^{-1/\tau}$, where τ decreases over time according to an exponential cooling schedule. If the snake is stuck entirely, a node is removed from the path and added to the tabu list. The key hyperparameters are tabu tenure T, initial temperature τ_0, and cooling rate α.

Formally, let age(v) denote the number of additions since v was added to the tabu list. A tabu candidate v is accepted with probability proportional to age(v)/T, where T is the tabu tenure. Non-tabu candidates are scored and selected as in standard tabu search.

4.3 Learning Algorithms

The learning algorithms described in this section train neural networks to learn a policy for constructing snakes autoregressively. Rather than relying on hand-designed scoring functions, these methods learn from thousands of iterations of trial and error to learn how to construct long snakes.

The general structure mostly shared by these approaches is an iterative generate-and-filter loop:

1. Use the current model to generate a batch of snakes.
2. Score each construction by some metric (typically path length).
3. Select the top-performing constructions (e.g., the top p% by score).
4. Train the model on the selected constructions.
5. Repeat.

Over successive iterations, the goal is that the model's distribution of constructions shifts toward higher-quality (longer) snakes.

4.3.1 Deep Cross-Entropy Method

The Deep Cross-Entropy Method (DCEM) adapts the classical cross-entropy method for combinatorial optimization by parameterizing the sampling distribution with a neural network. At each iteration, the network generates a large batch of snake constructions. These constructions are scored by length, and the top p% are selected as the *elite set*. The weights of the network are updated to maximize the likelihood of the elite constructions. Over many iterations, the network learns to assign high probability to node sequences that produce long snakes.

Algorithm 4 Tabu-Assisted Simulated Annealing

Require: $n \leftarrow$ number of dimensions
Require: $T \leftarrow$ tabu tenure
Require: $\tau_0 \leftarrow$ initial temperature
Require: $\alpha \leftarrow$ cooling rate, $0 < \alpha < 1$
Require: $t_{\max} \leftarrow$ maximum number of steps
Require: $f(P, v) \leftarrow$ scoring function
Require: $S \leftarrow$ some starting path ($[\mathbf{0}]$ or long snake in dimension $n-1$)
Ensure: $P^* \leftarrow$ longest snake found

```
P ← S
P* ← P
𝒯 ← ∅
τ ← τ_0
for t = 1, 2, ..., t_max do
    C ← CANDIDATES(P, n)
    a ← uniformly random action from {EXTEND, BACKTRACK}
    if a = EXTEND and C ≠ ∅ then
        v ← arg max_{v∈C} f(P, v)
        if v ∈ 𝒯 then
            if random(0,1) < age(v)/(2T) then    ▷ Accept tabu move with probability ∝ age(v)
                Append v to P
            end if
        else
            Append v to P
            Add v to 𝒯
        end if
    else if |P| > 2 then
        v_rem ← head or tail of P (random)
        if v_rem ∈ 𝒯 then
            if random(0,1) < age(v)/(2T) and random(0,1) < e^{-1/τ} then
                Remove v from P
            end if
        else
            if random(0,1) < e^{-1/τ} then
                Remove v from P
                Add v_rem to 𝒯
            end if
        end if
    end if
    if |𝒯| > T then
        Remove oldest entry from 𝒯
    end if
    if |P| > |P*| then
        P* ← P
    end if
    τ ← max(τ · α, τ_min)
end for
```

4.3.1.1 Model Architecture

We initially considered two architectures for the policy network, a multilayer perceptron (MLP) and a long short-term memory (LSTM) network After some initial testing with the MLP architecture, we focused singularly on the using an LSTM policy network for our experimentation. Our LSTM processes the sequence of nodes/transitions taken so far and outputs a distribution over the next dimension to traverse. The LSTM maintains a hidden state that holds a compact representation of the sequence of actions that led to the current path state.

At each step during generation, the model outputs logits with dimension n ($l \in \mathbb{R}^n$). Based on the search variant and current state, a subset of these values may be masked to zero, then $\pi = \textsc{Softmax}(l)$ is our probability distribution across n dimensions. The selected dimension is sampled from this distribution, $d \sim \pi$, and the search traverses this dimension from the head of the snake. For example if the network predicts dimension d and the current head is u, the next node is $u \oplus 2^d$ (u with the d-th bit flipped). This value is appended to the path and given as input to the model again until some stop condition is met. In the simple search, we score candidates using the length minus chords scoring function introduced in Section 4.2, and path length is used to score constructions in the restricted search.

4.3.2 PatternBoost

PatternBoost is a hybrid method that combines a learned generative model with a lightweight heuristic search [3]. The core idea is to use a transformer model to propose candidate constructions, which are then refined by a fast heuristic algorithm. The entire PatternBoost framework is roughly structured as follows.

1. **Global Search:** Train a transformer model on the current dataset of snake constructions. Use the trained transformer to sample a batch of new candidate snakes.
2. **Local Search:** Pass each generated candidate to a fast heuristic search (e.g., tabu search) that attempts to extend or improve the construction.
3. **Iterate Dataset:** Add any improved constructions to the dataset, and cull shorter ones if the dataset has reached its maximum size.

Over successive rounds, the transformer learns patterns from increasingly high-quality snakes, and can hopefully generate many high quality initial constructions for the local heuristic search, which can then further refine the transformer's output. The distribution of snake lengths in the training dataset is one of the key metrics used to evaluate the performance of PatternBoost on this task. We tested PatternBoost using each of our heuristic searches for the local search step.

Algorithm 5 Deep Cross-Entropy Method (node sequences)

Require: $n \leftarrow$ number of dimensions
Require: $\theta \leftarrow$ initial network parameters
Require: $C \leftarrow$ number of constructions per iteration
Require: $p \leftarrow$ elite percentile (e.g., 0.05)
Require: $s \leftarrow$ generation percentage (e.g. 0.8)
Require: $I \leftarrow$ number of iterations
Require: $E \leftarrow$ training epochs per iteration
Ensure: $P^* \leftarrow$ longest snake found

```
P* ← ∅
for i = 1, 2, ..., I do
    D ← ∅
    for c = 1, 2, ..., C do
        P ← [0]
        while CANDIDATES(P, n) ≠ ∅ do
            h ← P[|P| − 1]
            l ← network output given current state, parameterized by θ        ▷ l ∈ ℝ^n
            for j = 0, 1, ..., n − 1 do
                if h ⊕ 2^j ∉ CANDIDATES(P, n) then
                    l_j ← 0
                end if
            end for
            π ← SOFTMAX(l)
            Sample dimension d ∼ π
            v ← h ⊕ 2^d
            Append v to P
        end while
        D ← D ∪ {P}
        if |P| > |P*| then
            P* ← P
        end if
    end for
    E ← top p fraction of D by path length
    for e = 1, 2, ..., E do
        Update θ by minimizing cross-entropy loss on E
    end for
    D ← top s fraction of D by path length
end for
```

4.3.2.1 Tokenization

For our implementation of PatternBoost, we use a very simple tokenization scheme. If we are training our transformer on node sequence representations, then we we construct a fixed vocabulary containing one token per hypercube node (represented by its integer index in $\{0, 1, \ldots, 2^n - 1\}$) for searches utilizing a node sequence representation. Likewise, for transition sequence representations we can assign one token per dimension (represented similarly). We encode each snake directly as a sequence of these tokens in visitation order, and include standard special tokens BOS, EOS, and PAD to mark the beginning of a sequence, the end of a sequence, and padding for batching, respectively.

For example the node sequence $[n_1, n_2, \ldots, n_m]$ would be encoded as

$$[\text{BOS}, n_1, n_2, \ldots, n_m, \text{EOS}]$$

utilizing the decimal representation of nodes.

We also experimented with byte-pair encoding (BPE) tokenization over node sequences, but observed only minimal improvements in our experiments. As a result, we use the fixed node/dimension-level tokenization described above for all PatternBoost findings reported in this work.

Algorithm 6 PatternBoost

Require: $n \leftarrow$ number of dimensions
Require: $\mathcal{D}_0 \leftarrow$ initial dataset of snake constructions
Require: $S \leftarrow$ maximum dataset size
Require: $C \leftarrow$ number of candidates generated per round
Require: $I \leftarrow$ number of iterations
Require: LOCAL $\leftarrow$ a fast heuristic search algorithm
Ensure: $P^* \leftarrow$ longest snake found

```
D ← D_0
P* ← longest construction in D
for r = 1, 2, ..., I do
    Train transformer M on D
    for b = 1, 2, ..., C do
        P ← recursively sample from M until END OF SEQUENCE token
        if P is a valid snake then
            P' ← LOCAL(P, n)
        else
            continue                ▷ if the snake is invalid, do not pass it off to LOCAL
        end if
        if |P'| > min_{P∈D} |P| or |D| < S then
            Add P' to D
        end if
        if |P'| > |P*| then
            P* ← P'
        end if
    end for
    if |D| > S then
        Remove the |D| − S shortest constructions from D
    end if
end for
```

Chapter 5

Implementation

All algorithms were implemented in Python, and this chapter outlines important details regarding our search implementations. Experiments were conducted on WPI's Turing high-performance computing cluster. Heuristic search experiments were run on CPU nodes, while learning algorithm experiments utilized GPU nodes with NVIDIA A100 GPUs.

5.1 Algorithmic Faithfulness

A central implementation principle of this work was to remain as faithful as possible to the published specifications of each algorithm we applied. We made no significant algorithmic modifications intended to improve performance on the SIB task specifically. The DCEM and PatternBoost frameworks were implemented according to their respective original formulations [23, 3], and our heuristic searches follow standard descriptions of Tabu Search and Simulated Annealing from the metaheuristics literature [7, 10].

This decision was deliberate. Our primary goal was to evaluate how well these general methods perform on the SIB problem, rather than to engineer a specialized solver. Introducing domain-specific modifications, such as SIB-aware pruning rules, custom reward shaping, or problem-specific neural architectures, would obscure the degree to which the underlying frameworks generalize to this class of combinatorial optimization problems. By keeping the algorithms as close to their original forms as possible, the results reported in Chapter 3 reflect an honest assessment of the strengths and limitations of these methods when applied to the SIB problem.

The only adaptations made were those strictly necessary to apply the algorithms to the SIB domain: defining the construction vocabulary (hypercube nodes or transition dimensions), implementing the snake validity constraint (restricted search), and specifying the scoring function used to rank constructions. These changes are problem representation choices rather than algorithmic modifications, and are described in detail in Chapter 2 and Chapter 4.

5.2 Heuristic Search Implementation

The three heuristic search algorithms (Tabu Search, Simulated Annealing, and Tabu-Assisted Simulated Annealing) were each implemented as self-contained Python scripts with no dependencies. All snake state was represented using standard Python lists and sets, with bitwise operations used for neighbor computation in the hypercube.

To maximize the number of constructions explored within the available compute budget, all heuristic searches were parallelized using Python's `multiprocessing` module. This allowed multiple independent search instances to run simultaneously across CPU cores, with each process maintaining its own instance of the search. Results were aggregated across processes at the end of each batch of runs.

5.3 Learning Algorithm Implementation

The DCEM policy networks and the PatternBoost transformer were implemented in PyTorch. Both models were trained using the Adam optimizer.

The local search phase of PatternBoost was again parallelized across CPU cores using `multiprocessing`, with each worker receiving a single transformer-generated candidate snake and running a full heuristic search from that initialization. Given the large number of valid candidates produced in later iterations, this phase constituted the dominant computational bottleneck of the PatternBoost pipeline (see 3.3.2.2). Despite this, transformer snake generation was still batched to maximize GPU use.

5.4 Compute Environment

All experiments were conducted on WPI's Turing high-performance computing cluster. Heuristic search experiments were run on CPU nodes, while learning algorithm experiments utilized GPU nodes equipped with NVIDIA A100 GPUs. Individual experiment runs ranged from several hours to multiple weeks depending on the dimension and algorithm. For the largest experiments in dimensions 8-11, individual runs were allocated up to 14 days of wall-clock time. Longer PatternBoost runs were checkpointed and restarted across multiple jobs, extending effective wall-clock time to approximately one and a half months for select configurations.

Chapter 6

Testing and Evaluation

In this chapter, we present experimental results for our approaches to the SIB problem. Unless otherwise stated, all experiments were conducted on WPI's Turing high-performance computing cluster. Heuristic searches were run on CPU nodes, and learning algorithm experiments utilized NVIDIA A100 GPUs. All results reported beyond Section 6.2 use the restricted search variant, unless otherwise noted.

6.1 Experimental Design

6.1.1 Heuristic Searches

Each search algorithm's hyperparameters were frequently tweaked based on the given experiment, but many remained constant throughtout our testing.

- **Tabu Search:** The lone hyperparameter for our traditional tabu search is the tenure. For a search in dimension n, this value was selected uniformly at random from $T = n, \ldots, 2n$.
- **Simulated Annealing:** For simulated annealing, our initial temperature was initialized to $\tau = 0.5$ and we followed an exponential cooling schedule with $\alpha = 0.999999$.
- **Tabu-Augmented Simulated Annealing:** Tabu-augmented simulated annealing combines tabu search and simulated annealing, so naturally we had to set tenure, initial temperature, and cooling rate for this search. We initialized $\tau = 0.5$ and $\alpha = 0.999999$ again, but this time for a search in dimension n, tenure was chosen uniformly at random from $T = 3n, \ldots, 5n$. Empirically, we found that a longer tenure helped this algorithm explore more.

6.1.2 DCEM Architecture

Initial experiments with an MLP architecture were conducted but the MLP was quickly abandoned in favor of an LSTM network. All DCEM results reported in this work use the LSTM architecture.

Our LSTM network contained 2 hidden layers with hidden size 256. At each step, the network processed a bit vector representation of the current hypercube node and output a distribution over the next dimension to traverse. The network was trained with a learning rate of 10^{-3}, an elite percentile of 10%, and 2,500 constructions per iteration, using the Adam optimizer. Experiments were run for as many iterations as possible within approximately one week of wall-clock time.

6.1.3 PatternBoost Setup

The PatternBoost transformer was trained from randomly initialized weights with 6 layers, 8 attention heads, and embedding dimension 128. The model was trained using the Adam optimizer with a learning rate of 10^{-3} and a batch size of 16.

Each PatternBoost iteration generated approximately 100,000 candidate snakes via autoregressive sampling from the transformer. Valid candidates were passed to a local search algorithm, either Tabu Search or Tabu-Assisted Simulated Annealing. Experiments were run for as many iterations as possible within approximately one week of wall-clock time; the number of completed iterations varied considerably by dimension and local search algorithm due to the computational cost of the local search phase (see Subsubsection 6.3.2.1). Some longer searches were checkpointed at the end of the 1 week maximum job length, and restarted from the checkpoint, increasing the effective runtime to up to 1 month.

6.1.3.1 Initial Dataset

The initial training dataset for PatternBoost was generated by running over 240,000 instances of Tabu-Assisted Simulated Annealing, each for approximately 5,000 iterations. To increase the size and diversity of the training data, we applied hypercube automorphisms (translations and dimension permutations) to a subset of the generated snakes, producing additional equivalent constructions. This served as a regularization technique, exposing the transformer to multiple representations of structurally identical snakes during training.

6.2 Heuristic Searches

First, we report on two design decisions that apply across multiple methods, those being the choice between simple and restricted search, and the choice of scoring function.

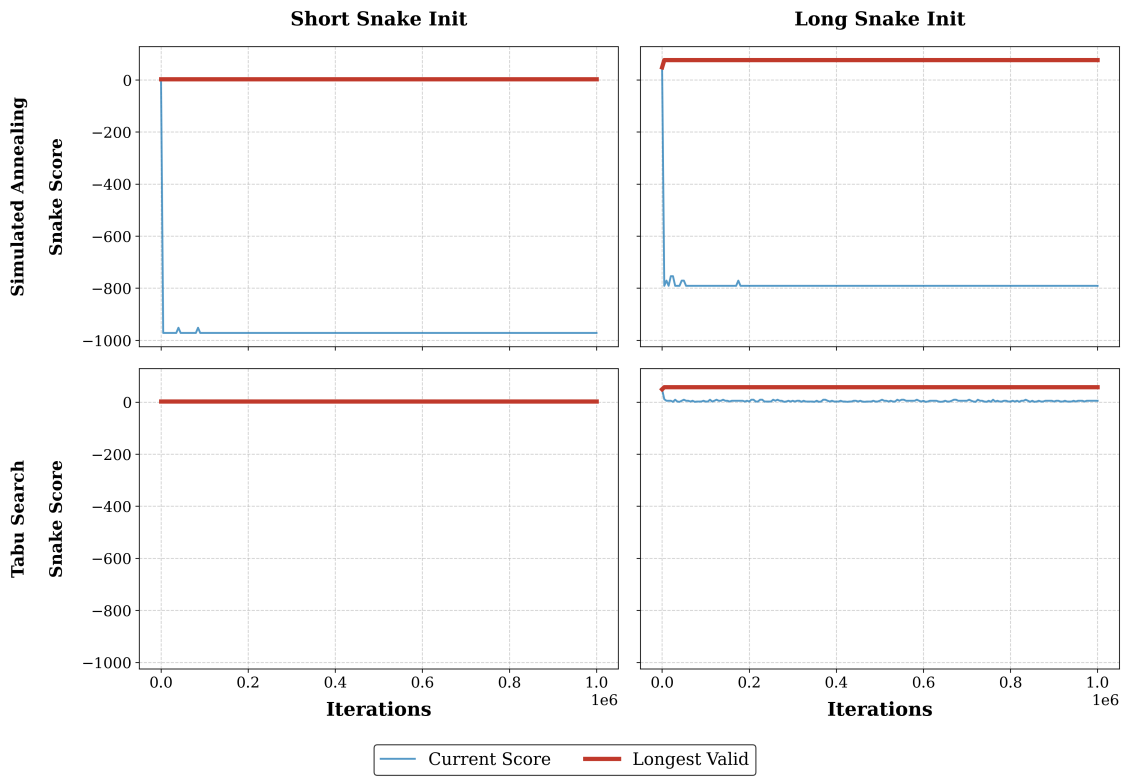

Figure 6.1: Comparison of different search strategies using the simple search variant and multiple initialization strategies.

6.2.1 Simple vs. Restricted Search

We compared simple and restricted search variants for Tabu Search and Simulated Annealing. Across all algorithms and dimensions tested, restricted searches initialized with long snakes in dimension $n-1$ produced longer snakes and converged faster. Simple searches frequently generated invalid paths containing many chords, and even with heavy penalization for invalid constructions, the proportion of valid snakes produced was incredibly low or zero. To show this, we ran instances of both variants of tabu search and simulated annealing in dimension 8 for 1,000,000 iterations.

The results for the simple search comparison are shown in Figure 6.1. Simulated annealing struggled greatly with this search variant. For both initializations strategies, simulated annealing performed incredibly poorly, greedily adding edges until the search was stuck at a highly negative score. On the other hand, tabu search also performed poorly. Despite some initial scoring gains, the current path length quickly collapses to near zero and remains there, indicating the algorithm is repeatedly constructing heavily chorded paths. Unlike simulated annealing, the tabu mechanism does keep the search exploring (the current path fluctuates rather than flatlines), but the exploration is unproductive in the simple search space.

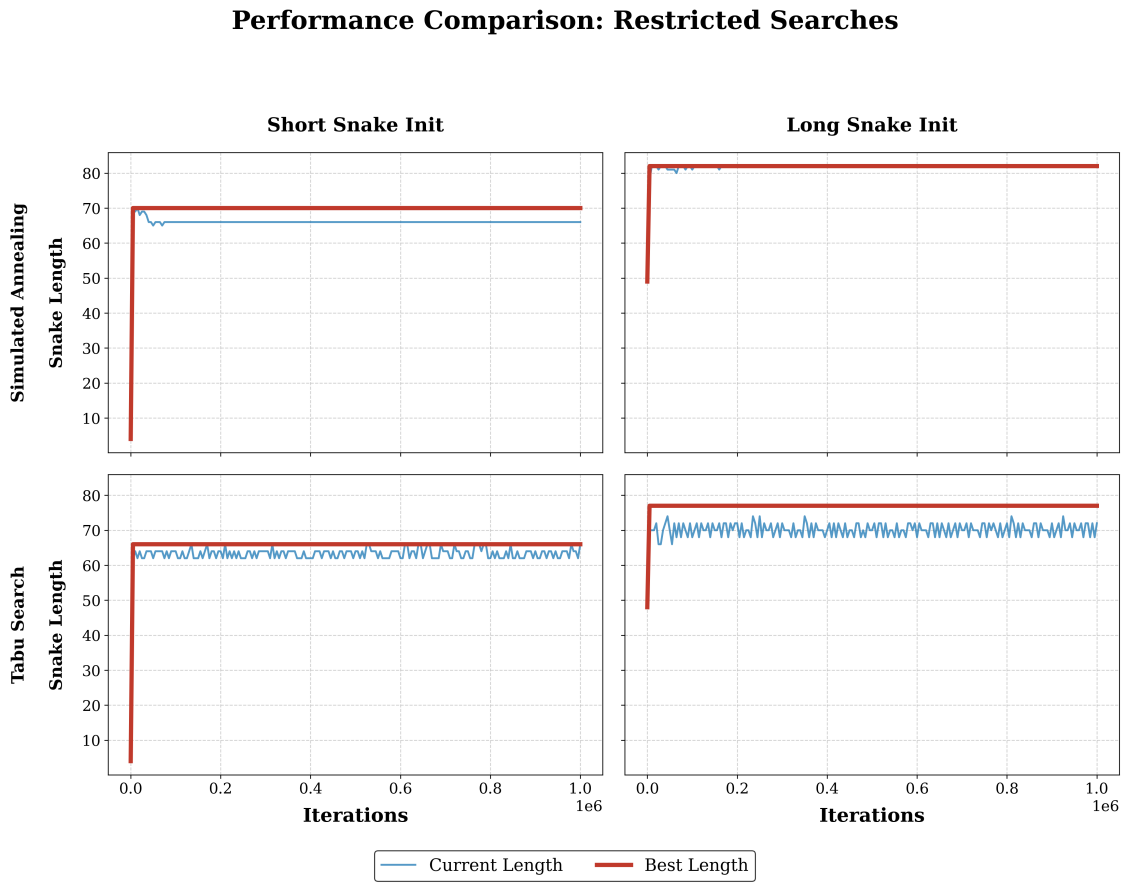

Figure 6.2: Comparison of different search strategies using the restricted search variant and multiple initialization strategies.

The results for the restricted search comparison are shown in Figure 6.2. For both searches, the restricted variant shows markedly better behavior. The long snake initialization tabu search results show the current path length oscillates between ∼67 and ∼73 throughout the search, with a best snake of length 77. These results are promising, but it is unable to push significantly beyond the local optimum it finds early on. The simulated annealing search with a long snake initialization finds a snake of length 82 within the first few thousand iterations but then plateaus entirely, with no improvement over the remaining iterations. Once the temperature begins to cool, simulated annealing behaves nearly greedily and lacks any mechanism to systematically escape the local optimum. These results illustrate two important limitations with simulated annealing, those being that the simple search is unsuited for simulated annealing on this problem, and even with restricted search, simulated annealing's reliance on random perturbations and a monotonically decreasing acceptance probability makes it poorly suited for this kind of search space where escaping local optima requires structured exploration.

In all of our initial testing, the restricted search outperformed the simple search. Consequently, all subsequent experiments use the restricted search variant exclusively.

6.2.2 Scoring Function Comparison

We evaluated four scoring functions for the heuristic search algorithms: length, tightness, mobility, and a linear interpolation between tightness and mobility that transitions from tightness-dominant to mobility-dominant as the snake length increases. Each scoring function was tested across all three heuristic algorithms in dimension 8. Again, each search ran for 1,000,000 iterations.

In Figure 6.3, no single scoring function stood out as clearly better suited for this task than others. The longest snake generated by tabu search was found using tightness, for simulated annealing and tabu-augmented simulated annealing it used length. Likewise, we did not uniformly apply one single scoring function throughout the rest of our experimentation. Each scoring function had different performance based on the underlying search algorithm. Some of the same behavior from the previous experiments can be seen across search strategies, such as tabu search exploring but not exceeding an early local optimum, and simulated annealing getting completely stuck at the first local optimum found.

For the learning algorithms, scoring was done exclusively by snake length. In PatternBoost, the local search phase occasionally used alternative scoring functions, but the dataset curation was always based on length.

6.2.3 Exploration Behavior

The experiments previously presented showed that tabu search and simulated annealing in particular did not efficiently explore the search space. These searches quickly converged to local optimum and were unable to escape for the remainder of the search.

For each configuration, we measured the average number of non-equivalent states searched

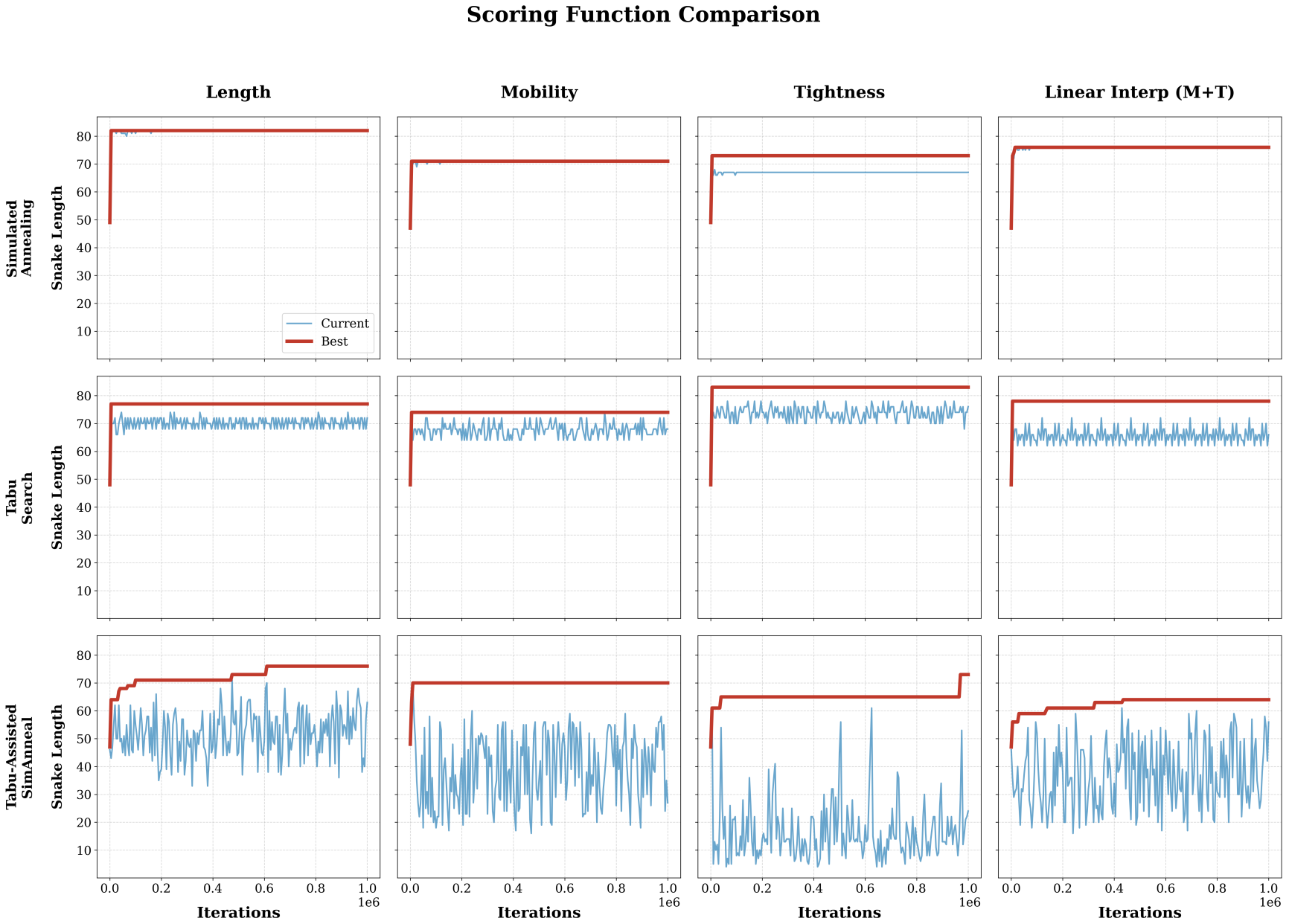

Figure 6.3: Comparison of scoring functions across search strategies.

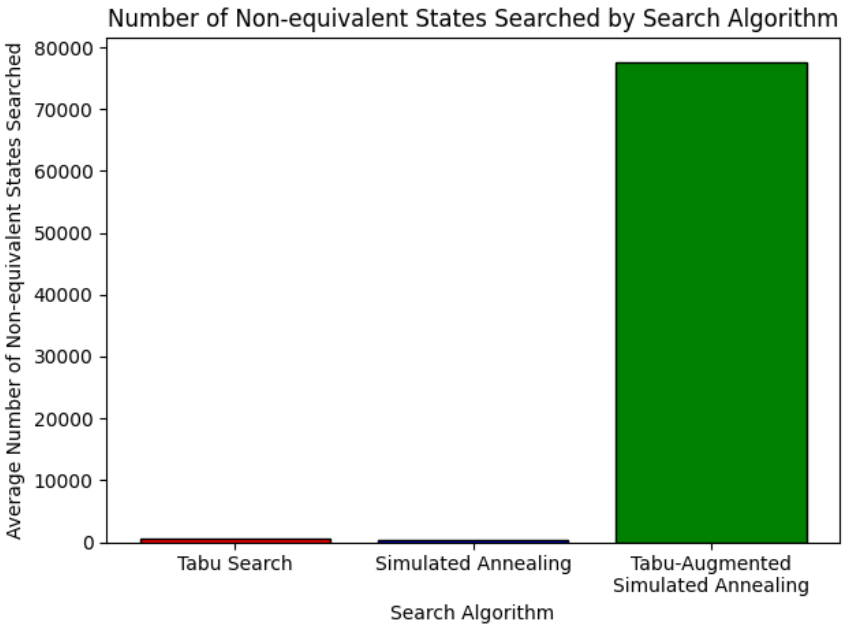

Figure 6.4: Number of non-equivalent states explored across search algorithm.

across multiple independent runs. Non-equivalent states were identified by computing the complete canonical form of each snake encountered during the search. This metric captures how effectively each algorithm explores structurally distinct regions of the search space, as opposed to revisiting equivalent constructions. Intuitively a higher number here is better, because it means that the search is exploring more unique states, but it also cannot be considered in a vacuum. It is possible for a search to explore many non-equivalent states, but if all of those states are short paths then all that exploration might not actually lead to better constructions.

By this metric alone, tabu-augmented simulated annealing is far and away the dominant search algorithm, as shown in Figure 6.4. This paints a clearer picture of what is occurring with tabu search and simulated annealing. They find strong initial constructions, but then spend hundreds of thousands of iterations looping through different but structurally equivalent node sequences. That being said, for each scoring function, the maximum length snakes found by one or both of tabu search and simulated annealing were longer than compared to tabu-augmented simulated annealing. This demonstrates the tradeoff here between high exploration and maximum path length.

We also evaluated the average number of non-equivalent states searched across multiple independent runs categorized by the scoring function used for the search. These results are shown in Figure 6.5. Once again, there is no clear "winner" here. The tightness scoring function performed significantly worse than others, but among length, mobility, and the linear interpolation metric, each scoring function explored a large number of nonequivalent states.

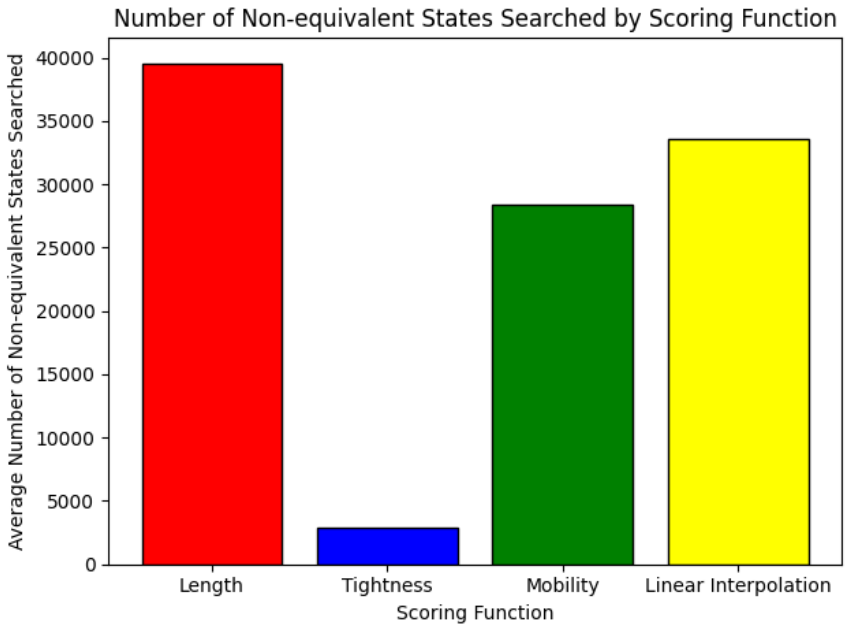

Figure 6.5: Number of non-equivalent states explored across scoring functions.

6.3 Learning Algorithms

6.3.1 DCEM

The DCEM experiments focused primarily on dimension 8 using the restricted search variant. While the model demonstrated rapid early improvement, it quickly plateaued, reaching a maximum snake length of 92 edges. This result fell 3 edges short of the best results achieved by PatternBoost and 5 edges short of the known optimum for that dimension. One such search is outlined in Figure 6.6.

A significant limitation observed was the model's tendency to converge toward a narrow region of the construction space. Once a local optimum was identified, the policy network lacked sufficient selection pressure to generate snakes exceeding the lengths already observed in the elite set. Consequently, the LSTM-based policy often learned to merely replicate a single known construction rather than exploring for novel, longer paths. Subsequent iterations after the initial growth phase produced no improvements in snake length.

The choice of search variant was also a decisive factor in performance. Figure 6.7 outlines a simple DCEM search. DCEM instances trained from random weights under the simple search variant failed to generate any valid snakes at all. This failure stemmed from the lack of strong constructions in early iterations of the search. Without strong examples to learn from early on, the network learns essentially random paths and is unable to generate any useful structures in later iterations. Furthermore, DCEM was not utilized for dimensions $n \leq 7$ because standard heuristic searches were able to recover optimal snakes for those dimensions at a significantly lower computational cost.

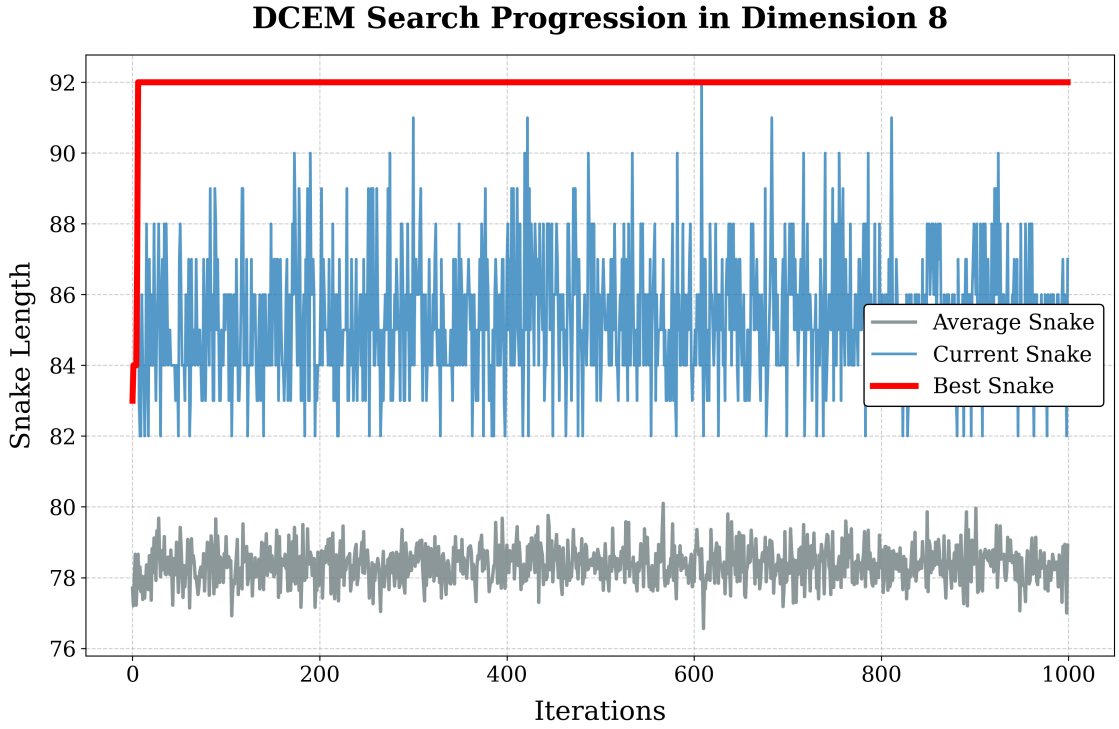

Figure 6.6: Search progress over iterations for DCEM in dimension 8.

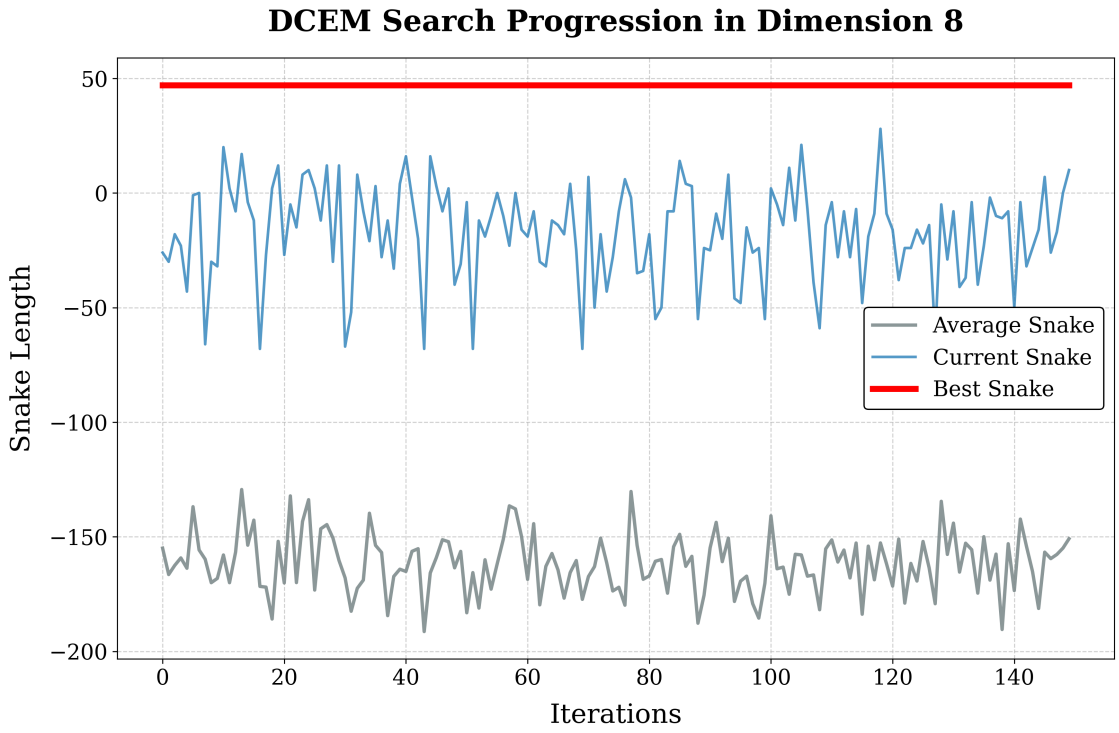

Figure 6.7: Search progress over iterations for DCEM (simple variant) in dimension 8.

Dimension	Known Best	Tabu Search	TA Simulated Annealing
8	98	95	95
9	190	–	173
10	373	–	276
11	721	–	443

Table 6.1: Longest snake found by PatternBoost per dimension and local search algorithm.

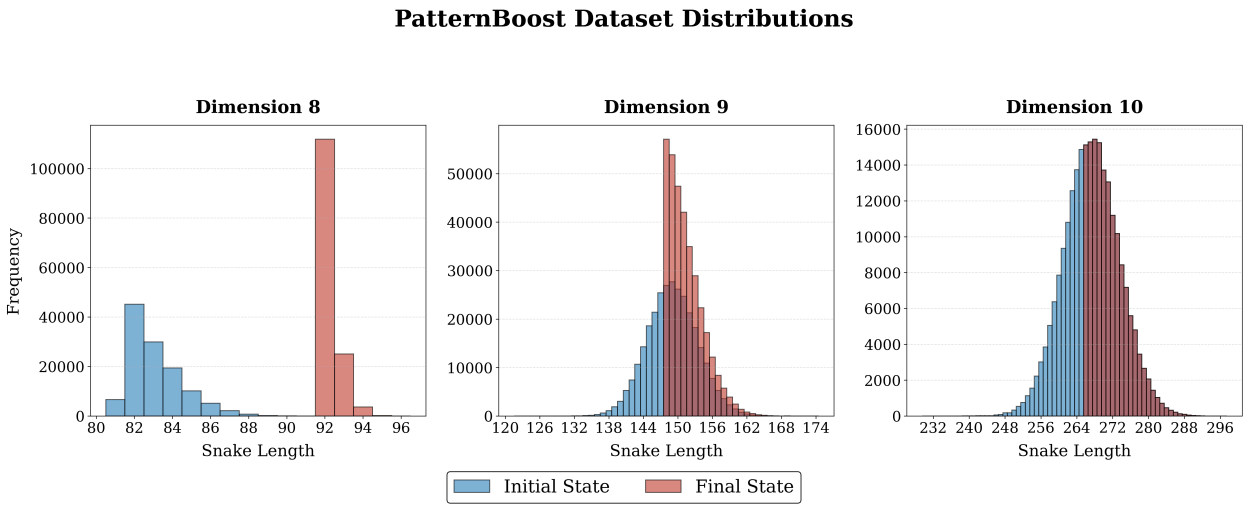

Figure 6.8: Distribution of snake lengths in the PatternBoost training dataset (initial set and final set) in dimensions 8-10.

6.3.2 PatternBoost

PatternBoost produced the longest snakes among our learning-based methods in dimensions 8, 9, and 11. Table 6.1 summarizes the best snake lengths achieved by PatternBoost with each local search algorithm. PatternBoost searches were run for at least one week of wall clock time, and the number of iterations completed within the time budget varied considerably by dimension and local search algorithm.

Additionally, the vast majority of the experimentation with PatternBoost focused on dimension 8. Early on we identified PatternBoost as our most promising algorithm, and ran thousands of hours of testing in dimension 8 to attempt to reconstruct a known-optimal snake.

During a run of PatternBoost, a dataset of constructions is maintained which the transformer is trained on. For maximization problems, one goal of PatternBoost is to gradually shift the distribution of scores in this dataset higher and higher. Figure 6.8 shows how these distributions changed over the course of our searches in dimensions 8-10. Clearly, a significant shift in the dataset occurred in dimension 8, but a similar change was not observed in dimensions 9 or 10. In dimension 9, the search began replicating some of the longer snakes in the dataset, but the transformer had not yet learned to replicate some of the longest constructions and the local search was unable to adapt the constructions the global search was

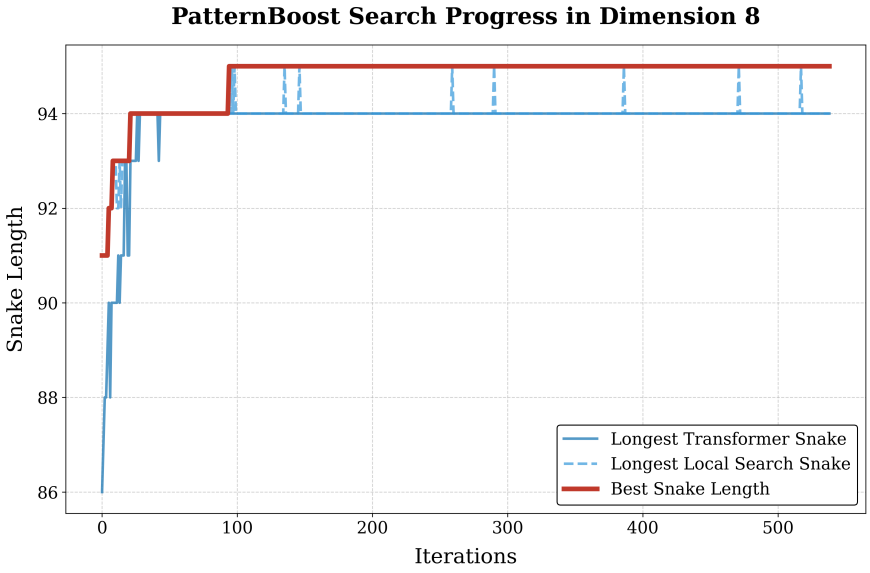

Figure 6.9: Search progress over iterations for PatternBoost in dimension 8.

able to generate into longer paths. In dimension 10 the dataset remained largely unchanged with the exception of some of the shortest snakes being culled, demonstrating the difficulty of the task as the dimension grew.

Looking more at the evolution of an individual runs of PatternBoost, we can a variety of behavior, some of which is similar to that of our other searches. Figure 6.9 visualizes the entirety of a run of PatternBoost in dimension 8. The first 100 iterations of the search are promising, as gradual improvement in the best snake construction can be observed. The final 400+ iterations of the search (which represents a vast majority of compute) shows the transformer completely converging to a local optimum and the local search occasionally replicating the top construction, but never exceeding it.

In dimensions 9 and 10, shown in Figure 6.10, we see different behavior. A reasonable first observation is that these searches completed significantly more PatternBoost iterations than were completed in dimension 8, and this will be covered later in Subsubsection 6.3.2.1. The search in dimension 9 shows great promise, and it is possible (maybe even likely) that with more compute and experimentation, we could exceed the 173 length construction found during our searches. Dimension 10 is a different story. Once again, we can observe initial gains in the beginning of the search, followed by a dramatic convergence to a local optimum. It is unlikely that more compute would improve this search at all. In fact, we can see in Figure 6.8 that the longest snake generated by PatternBoost in dimension 10 (length 276) is shorter than the longest snake in the initial dimension 10 dataset (length 296). The hypercube structure in dimension 10 must be quite difficult for a transformer of this size to learn, considering it could not even replicate the training data.

These PatternBoost experiments demonstrate that while this transformer-based approach can find long snakes in select dimensions, it faces a clear convergence plateau as complexity increases. The results highlight a recurring pattern where initial optimization

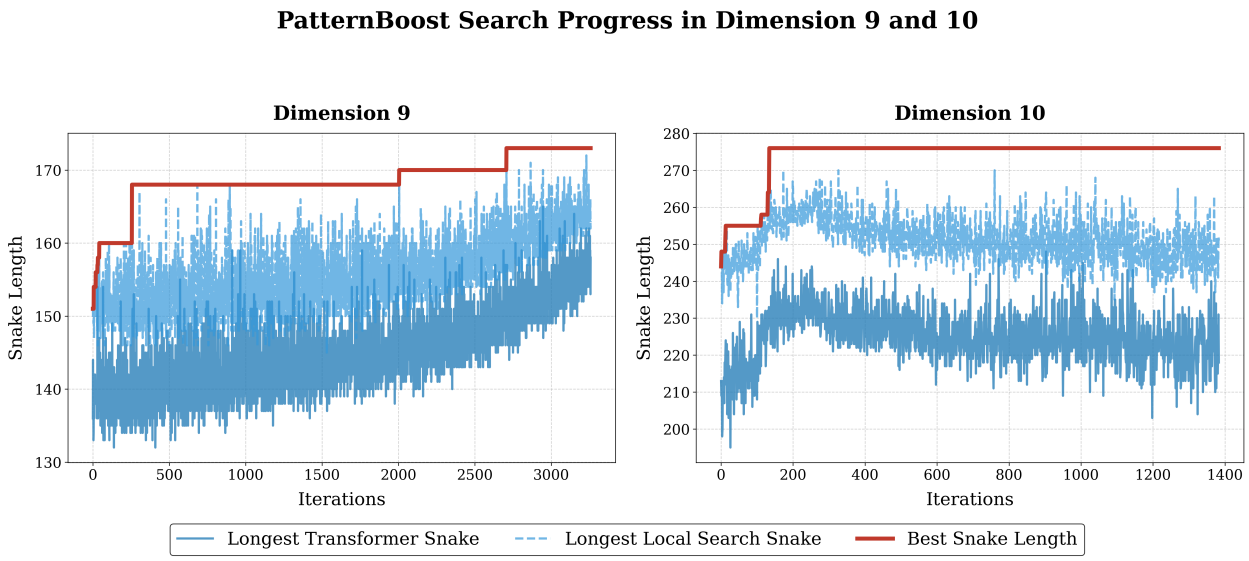

Figure 6.10: Search progress over iterations for PatternBoost in dimensions 9 and 10.

gains are followed by a total convergence to local optima, and additional compute fails to yield further improvements. Furthermore, the difficulty the model faced in even replicating the best snakes in the training data for the highest dimensions suggests that the complexity of the hypercube structure eventually outpaces the current transformer's learning capacity.

6.3.2.1 Local Search Bottleneck

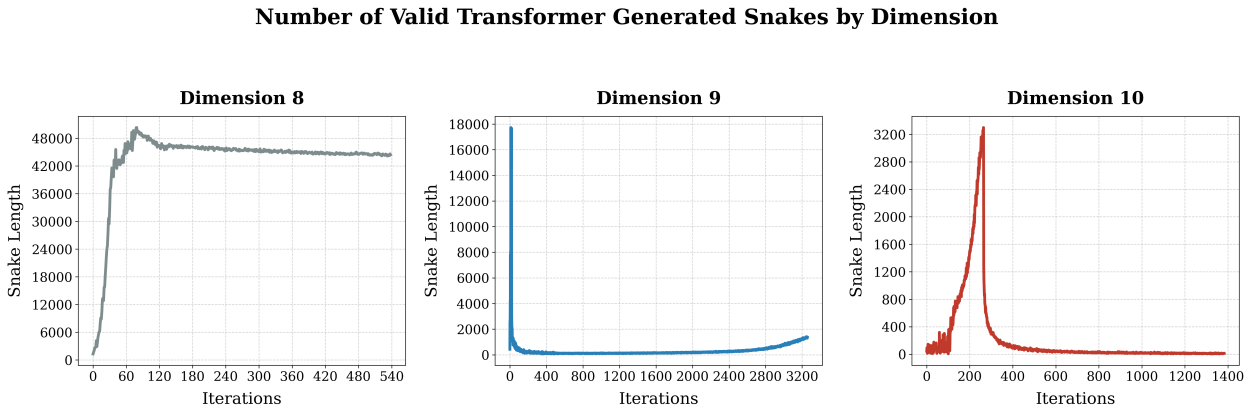

Figure 6.11: Number of valid snake constructions over iterations for dimensions 8-10.

The computational bottleneck in PatternBoost was the local search phase. Transformer inference on A100 GPUs was fast, generating ∼100,000 candidate snakes in at most an hour per iteration. However, the local search step required running a heuristic algorithm on each valid candidate. Even with the large amount of CPU resources and memory available on WPI's Turing cluster, processing the full batch of candidates took many hours and, dominated the per-iteration cost when the transformer successfully generated many valid

constructions. This bottleneck was more pronounced in dimension 8 compared to dimensions 9 and 10.

In Figure 6.11, we can see that the transformer generated many hundred times more snakes in dimension 8 than in 9 and 10, with the exception of some phases of replication early in training. Each of these valid snakes was passed off to the local search, and with tens of thousands of valid snake constructions this ate up a large majority of compute each iteration in dimension 8. Fewer valid transformer constructions each iteration in dimensions 9 and 10 meant that many more iterations of PatternBoost can be completed in the same amount of time.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This project provided a rigorous evaluation of modern neural machine learning frameworks applied to the SIB problem. By implementing the Deep Cross-Entropy Method (DCEM) and the PatternBoost framework, we established that general-purpose ML combinatorial optimization can effectively learn the structural properties of hypercube graphs to generate long induced paths. A critical technical finding of this work was the uniform superiority of restricted search variants, which guarantee validity by construction, over simple searches that utilize chord penalties. Our results demonstrate that restricted search is a necessary condition for neural methods to navigate the massive search space of the n-dimensional hypercube effectively.

The most successful implementation in this study, PatternBoost, successfully recovered known optimal snakes for dimensions $n \leq 7$ and reached a maximum length of 95 in dimension 8, falling within 3.06% of the known record. Despite this success, the widening gap between our achieved results and known lower bounds for dimensions $n \geq 9$ indicates that these methods struggle to scale without significant modification or a massive amount of compute. We identified that the primary computational bottleneck resides in the local search phase of the construction pipeline rather than in transformer inference, but the transformer still struggled to generate diverse candidates.

While this work did not produce new record-length snakes in open dimensions, it provides an essential baseline for the application of AI to combinatorial optimization in discrete geometry. The rapid convergence to local optima observed across our models suggests that imitation-style training alone lacks the optimization pressure required to discover novel constructions in higher dimensions. These findings point toward the necessity of diversity-aware objectives, true reinforcement learning frameworks, or other heuristic algorithmic improvements to push beyond the local optima identified in this study.

7.2 Future Work

7.2.1 Addressing Convergence to Local Optimum

The most significant limitation shared by nearly all methods is convergence to local optima. Neither of our learning algorithms provide direct optimization pressure to generate snakes longer than those already observed. Potential remedies include:

- **Diversity-aware training objectives**: Incorporating novelty or diversity bonuses into the elite selection (DCEM) or dataset curation (PatternBoost) to penalize redundant constructions and encourage exploration of underrepresented regions of the search space.
- **Curriculum strategies**: Gradually increasing the minimum snake length required for inclusion in the training set, forcing the model to continually target longer constructions.
- **True reinforcement learning**: Replacing the imitation-style training in both frameworks with a reinforcement learning objective that directly rewards length improvement.

7.2.2 Automated Algorithmic Discovery

A promising direction not fully explored in this work is **automated algorithmic discovery** using large language models (LLMs). In preliminary experiments, we implemented a framework called CPro1 that prompts an LLM to propose, implement, and optimize heuristic search algorithms for combinatorial designs [18]. CPro1 automates the algorithmic design loop: the LLM proposes a high-level heuristic strategy, plans an implementation, writes the code, and the system automatically tunes hyperparameters via grid search. The LLM is then prompted to optimize the algorithm based on its performance on solved instances (dimensions $n \leq 8$), and the best-performing algorithm is selected for extended runs on open dimensions.

This approach is conceptually related to FunSearch [17], which uses LLMs to search the space of programs rather than the space of solutions directly. The advantage of this paradigm is that the LLM can leverage broad programming knowledge and mathematical intuition to propose algorithmic strategies that would be difficult to design by hand, while the evaluation loop ensures that only effective algorithms are retained. We believe this direction warrants significant further investigation, particularly as LLM capabilities continue to improve. These methods have already provided impressive results, such as in the problem of slicing the hypercube [21].

7.2.3 Scaling and Engineering

Given the ineffectiveness of our current algorithms in high dimensions ($d \geq 9$), we identify several practical improvements could extend the reach of our methods.

- **Faster local search**: Implementing the heuristic search algorithms in a compiled language (e.g., C or Rust) would reduce Python overhead, which contributes to PatternBoost's local search bottleneck.
- **Selective candidate filtering**: Rather than running local search on all valid constructions out of the $\sim$100,000 transformer-generated candidates, pre-filtering based on lightweight heuristics (e.g., initial snake length or other scoring function) could focus compute on the most promising candidates.

Bibliography

[1] Benjamin P. Carlson and Dean F. Hougen. Phenotype feedback genetic algorithm operators for heuristic encoding of snakes within hypercubes. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, pages 791–798, New York, NY, USA, July 2010. Association for Computing Machinery.

[2] D.A. Casella and W.D. Potter. Using evolutionary techniques to hunt for snakes and coils. In *2005 IEEE Congress on Evolutionary Computation*, volume 3, pages 2499–2505 Vol. 3, September 2005.

[3] François Charton, Jordan S. Ellenberg, Adam Zsolt Wagner, and Geordie Williamson. PatternBoost: Constructions in Mathematics with a Little Help from AI, November 2024.

[4] Chen Dang, Cristina Bazgan, Tristan Cazenave, Morgan Chopin, and Pierre-Henri Wuillemin. Warm-Starting Nested Rollout Policy Adaptation with Optimal Stopping. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(10):12381–12389, June 2023.

[5] D. W. Davies. Longest "Separated" Paths and Loops in an N Cube. *IEEE Transactions on Electronic Computers*, EC-14(2):261–261, April 1965.

[6] Gray Frank. Pulse code communication, March 1953.

[7] Fred Glover. Tabu Search—Part I. *ORSA Journal on Computing*, 1(3):190–206, August 1989.

[8] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, April 1950.

[9] W. H. Kautz. Unit-Distance Error-Checking Codes. *IRE Transactions on Electronic Computers*, EC-7(2):179–180, June 1958.

[10] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

[11] Victor Klee. What is the Maximum Length of a d-Dimensional Snake? *The American Mathematical Monthly*, January 1970.

[12] Allen Newell, J. C. Shaw, and Herbert A. Simon. Elements of a theory of human problem solving. *Psychological Review*, 65(3):151–166, 1958.

[13] Patric R. J. Östergård and Ville H. Pettersson. Exhaustive Search for Snake-in-the-Box Codes. *Graphs and Combinatorics*, 31(4):1019–1028, July 2015.

[14] Stanislas Polu and Ilya Sutskever. Generative Language Modeling for Automated Theorem Proving, September 2020.

[15] W. D. Potter, R. W. Robinson, J. A. Miller, K. J. Kochut, and D. Z. Redys. Using the genetic algorithm to find snake-in-the-box codes. In *Proceedings of the 7th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, IEA/AIE '94, pages 421–426, USA, May 1994. Gordon and Breach Science Publishers, Inc.

[16] C Ramanujacharyulu and VV Menon. A note on the snake-in-the box problem. In *Annales de l'ISUP*, volume 13, pages 131–135, 1964.

[17] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, January 2024.

[18] Christopher D. Rosin. Using Code Generation to Solve Open Instances of Combinatorial Design Problems, January 2025.

[19] Jia Tracy Shen, Michiharu Yamashita, Ethan Prihar, Neil Heffernan, Xintao Wu, Ben Graff, and Dongwon Lee. MathBERT: A Pre-trained Language Model for General NLP Tasks in Mathematics Education, August 2023.

[20] Hunter S. Snevily. The snake-in-the-box problem: A new upper bound. *Discrete Mathematics*, 133(1):307–314, October 1994.

[21] Duncan Soiffer, Nathaniel Itty, Christopher D. Rosin, Blake Bruell, Mason DiCicco, Gábor N. Sárközy, Ryan Offstein, and Daniel Reichman. Improved Upper Bounds for Slicing the Hypercube, February 2026.

[22] Terence Tao. Mastadon post. https://mathstodon.xyz/@tao/115855840223258103, January 2026.

[23] Adam Zsolt Wagner. Constructions in combinatorics via neural networks, April 2021.

[24] Ed Wynn. Constructing circuit codes by permuting initial sequences, January 2012.

[25] Yonatan Yehezkeally and Moshe Schwartz. Snake-in-the-Box Codes for Rank Modulation. *IEEE Transactions on Information Theory*, 58(8):5471–5483, August 2012.

[26] Gilles Zémor. An upper bound on the size of the snake-in-the-box. *Combinatorica*, 17(2):287–298, June 1997.

Appendices

Appendix A

Longest Snakes Found

A.1 Dimension 8

0 1 2 0 3 4 0 5 3 2 0 4 3 1 6 3 0 1 2 0 3 4 1 7 4 3 0 4 1 0 2 4
3 0 1 5 0 3 4 0 1 4 2 0 3 4 0 1 7 4 3 0 4 1 2 4 3 0 4 5 1 6 5
0 7 2 4 3 0 4 1 0 2 4 3 0 4 5 0 2 4 3 0 2 1 4 0 2 7 4 0 3 7 4

A.2 Dimension 9

0 1 2 0 3 4 0 1 5 6 1 0 4 3 2 0 1 2 6 0 3 2 1 0 4 3 1 0 7 3 0 4
3 1 0 2 3 4 0 3 1 6 2 5 3 2 1 0 3 5 2 3 4 0 2 1 0 3 2 5 1 0 2
1 8 3 1 2 0 4 3 0 2 6 3 0 4 3 2 1 3 0 4 3 5 1 0 3 4 0 2 3 1 0 3
4 0 6 1 0 4 3 2 0 1 3 0 4 3 2 5 6 8 4 3 0 1 2 0 4 3 7 6 1 8 6
3 1 4 0 2 1 3 4 5 1 3 0 4 3 2 1 0 3 4 6 3 1 0 3 4 0 2 3 5 0 1 5
3 4 5 1 0 2 8 4 2 1 0 2 8 0 3

A.3 Dimension 10

0 1 2 3 1 4 3 5 6 4 2 7 3 2 6 1 3 4 8 0 2 3 7 2 8 0 2 3 8 6 0 2
4 0 7 1 4 5 1 7 8 3 2 4 8 7 2 8 3 2 4 5 3 0 4 7 1 8 0 2 9 3 2
7 0 2 1 6 0 2 7 0 3 8 4 3 2 1 6 3 7 0 2 8 7 0 6 7 3 8 0 7 8 2 7
3 4 2 8 1 4 8 7 2 4 8 1 3 7 0 3 1 4 5 2 8 7 2 4 3 8 2 7 0 1 6
4 8 7 2 8 3 2 4 8 7 2 3 1 2 7 8 2 3 8 4 2 7 8 2 3 6 8 7 2 8 3 4
8 7 2 8 1 4 2 8 7 2 4 3 2 0 6 8 4 6 3 2 4 6 8 4 7 2 8 4 6 8 2
3 4 8 6 4 1 3 8 2 7 8 4 2 3 8 7 2 5 0 2 7 8 3 2 4 8 7 2 8 3 2 0
7 2 3 8 4 7 9 4 6 5 8 2 0 4 8 3 4 0 6 2 1 7 4 6 5 2 8 6 7 2 1

5 3 1 0 2 8 7 0 1 2 4 8 3 2 4 6 7 1 8 3 2 5 8 2 0 8 3 4 0 8 1 7
2 1 3 8 5 2 0 7 2 3 8 0

A.4 Dimension 11

0 1 2 3 4 5 0 6 7 5 4 0 3 2 8 0 2 7 5 9 8 0 6 3 2 4 8 7 9 0 10
1 9 4 6 7 2 9 6 7 8 2 7 5 2 10 5 9 7 4 1 5 10 8 9 10 0 9 1 3 9
7 3 5 2 0 9 8 6 4 5 10 4 1 8 5 10 9 0 10 6 2 9 7 4 1 7 3 0 2 1
0 10 5 3 9 6 0 1 6 9 10 2 7 10 9 3 5 7 0 2 10 4 2 0 6 5 3 2 6 0
4 7 2 4 5 6 2 4 7 10 9 7 5 0 3 8 10 4 0 2 6 3 4 6 2 5 4 7 8 3
4 7 2 1 8 9 6 5 2 0 10 4 0 8 7 0 4 10 0 2 3 0 6 9 10 0 7 3 6 5
4 0 2 10 5 8 7 4 6 5 7 10 3 2 10 9 8 3 2 6 0 4 2 5 9 7 0 10 8 0
6 8 9 10 3 9 7 8 5 10 3 0 5 8 2 0 5 10 8 6 3 0 4 7 2 9 0 6 5 3
0 6 4 3 2 0 7 2 5 0 3 6 7 9 6 0 9 2 0 4 10 2 7 9 10 8 5 6 1 7
10 0 5 4 6 8 2 10 0 7 4 10 9 4 0 3 9 5 6 0 2 8 7 5 9 7 3 5 2 8
0 2 3 5 4 0 2 8 6 0 2 1 0 7 2 10 6 8 10 0 3 10 1 5 0 4 5 3 7 5
2 3 4 5 9 7 0 4 9 3 5 9 10 4 6 2 10 1 7 10 9 1 2 3 5 9 6 2 9 8
2 5 10 4 2 7 10 9 3 1 8 9 0 1 2 3 4 5 2 8 4 9 7 10 2 3 8 7 6 5
3 4 0 9 4 2 7 0 8 4 9 0 7 4 0 10 8 6 1 5 9 7 0 3 10 4 5 0 9 8 6
10 4 1 9 0 6 1 0 10 3 8 0 7 9 8 4 3 1 2 9 10