Spring 2022

Finding Optimal Cayley Map Embeddings Using Genetic Algorithms

Jacob Buckelew
jbuckelew@rollins.edu

Finding Optimal Cayley Map Embeddings Using Genetic Algorithms

Jacob Buckelew

Faculty Sponsor: Dr. Mark Anderson

Department of Math and Computer Science
Rollins College
Winter Park, FL
May 2022

Finding Optimal Cayley Map Embeddings Using Genetic Algorithms

Jacob Buckelew

Abstract

Genetic algorithms are a commonly used metaheuristic search method aimed at solving complex optimization problems in a variety of fields. These types of algorithms lend themselves to problems that can incorporate stochastic elements, which allows for a wider search across a search space. However, the nature of the genetic algorithm can often cause challenges regarding time-consumption. Although the genetic algorithm may be widely applicable to various domains, it is not guaranteed that the algorithm will outperform other traditional search methods in solving problems specific to particular domains. In this paper, we test the feasibility of genetic algorithms in solving a common optimization problem in topological graph theory. In the study of Cayley maps, one problem that arises is how one can optimally embed a Cayley map of a complete graph onto an orientable surface with the least amount of holes on the surface as possible. One useful application of this optimization problem is in the design of circuit boards since such a process involves minimizing the number of layers that are required to build the circuit while still ensuring that none of the wires will cross. In this paper, we study complete graphs of the form K_{12m+7} for positive integers m and we work on mappings with the finite cyclic group Z_n. We develop several baseline search algorithms to first gain an understanding of the search space and its complexity. Then, we employ two different approaches to building the genetic algorithm and compare their performances in finding optimal Cayley map embeddings.

1 Introduction

For decades, mathematical optimization has been of significant interest not only to mathematicians but to computer scientists, economists, and engineers. Many optimization problems are classified as combinatorial optimization problems, which typically require one to find an optimal solution, such as a permutation or graph, in a particular search space that minimizes an objective function [1]. The most common combinatorial optimization problems such as the Traveling Salesman problem and Timetabling and scheduling problems are also considered $\mathcal{NP}$-hard indicating that these problems cannot be solved using polynomial time algorithms [1]. Although there are traditional search methods to find optimal solutions to problems, there has been growing interest in the use of metaheuristics to solve optimization problems. Using a metaheuristic algorithm, one can potentially discover near-optimal solutions to challenging problems that provide approximate solutions efficiently. Formally, a metaheuristic is defined as an "iterative search process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space, learning strategies are used to structure information in order to find efficiently near-optimal solutions" [2]. Common metaheuristic algorithms include Ant Colony Optimization, Genetic Algorithms, Evolutionary Programming, and Simulated Annealing [3].

The advent of genetic algorithms can be dated back to the second half of the 20th Century. Starting in the 1950s, the concept of "evolutionary computation" grew out of efforts to employ evolutionary systems as tools for mathematical optimization [4]. The birth of evolutionary computation spurred the growth of a wide host of metaheuristics including the genetic algorithm. Beginning in the 1960s and 1970s, John Holland and a group of researchers from the University of Michigan began establishing the theoretical framework for genetic algorithms [5]. Essentially, genetic algorithms were founded in natural processes such as natural selection and genetics. Holland's goals were to rigorously explain the adaptation process that occurs in natural systems as well as design robust artificial systems that could mirror such natural systems [6]. In other words, the goal of the genetic algorithm was to abstract the important adaptive processes found in biological evolution and build computer algorithms that could solve complex problems that demanded adaptation and robustness. The basic fundamental processes that increase variation in a population such as DNA crossover and mutation became building blocks for a new metaheuristic algorithm that was not deterministic in nature, but stochastic.

In the 1980s, David Goldberg began implementing Holland's framework with his Binary-coded Genetic Algorithm (BGA), which modeled chromosomes in a population of individuals as bit strings of 1's and 0's [6]. Using these bit strings an algorithm could then perform stochastic processes such as recombination and mutations to iteratively find a string that corresponds to an optimal solution. The algorithm became a driver of natural selection, learning the bit patterns that were most advantageous for an individual and then considering individuals with those patterns as having a higher degree of evolutionary fitness. Furthermore, BGAs were much simpler to implement, making it a very popular metaheuristic tool following its formulation [3]. For decades following Goldberg's work, researchers would continue pushing the limits of genetic algorithms and related algorithms. After nearly 30 years of development, the field of evolutionary computation would encompass a wide variety of "Evolution Programs" including Genetic Algorithms, Evolutionary Programming, Genetic Programming, Scatter Search, and Genetics-Based Machine Learning [7].

The rise of genetic algorithms spawned an extensive amount of literature regarding their viability in solving challenging combinatorial optimization problems. Beginning in 1985, researchers began using genetic algorithms to solve the Traveling Salesman Problem(TSP), a well-known $\mathcal{NP}$-Hard problem [8]. Many attempts were made at solving TSP with each approach using a different crossover technique as a way of integrating problem specific knowledge into the algorithm and thus better optimize solutions. More recently, in 2013 Nagata developed a genetic algorithm that could outperform other efficient heuristic algorithms in solving TSP instances with 200 or more cities [9]. Furthermore, genetic algorithms have achieved high accuracy rates in solving a wide variety of operation management problems such as scheduling, facility layout problems, network design, and forecasting [10]. In this paper, we will primarily be studying the application of genetic algorithms in solving a particular combinatorial optimization problem in topological graph theory that involves the embedding of Cayley maps onto orientable surfaces.

We will first provide a brief review of Cayley Maps in Section 1.1 to establish the necessary background for the optimization problem we attempt to solve. In Section 1.2, we will then review the various search methods that will be employed in our approach including brute-force search methods, hill climbing algorithms, and genetic algorithms. Section 2 provides a detailed look at the computational methods that will be used in our approach. Then, Section 3 will present our results followed by a discussion of these results in Section 4. Finally, Section 5 will be reserved for a conclusion of the paper followed by an Appendix that includes a link to a Github repository containing relevant Python code.

2 Background

In this section, we will provide a basic foundation for the concepts presented in the rest of the paper. In Section 2.1, we will provide a brief introduction to concepts that are necessary for understanding Cayley map embeddings onto orientable surfaces. We also describe an example of a small graph embedding using the complete graph K_5. Then, in Section 2.2, we will present an overview of the search methods that will be employed in this paper including brute force search, hill climbing, stochastic hill climbing, and genetic algorithms.

2.1 Cayley Maps

In order to understand the optimization problem that will be discussed in this paper, we will now review definitions that provide a starting point for understanding Cayley map embeddings. The definitions and several of the figures are credited to Miriam Scheinblum, another Rollins College student who conducted research on Cayley map embeddings for her Senior Honors Thesis [11].

Definition 1 *A graph G is defined as a 2-tuple such that $G = (V, E)$, where set V contains the vertices of G connected by a set E of edges.*

Although there are many classes of graphs in graph theory, we will focus primarily on complete graphs. Complete graphs contain properties that will allow us to simplify the algorithmic approach to solving the graph embedding problem discussed later in this section.

Definition 2 *A complete graph K_n has n vertices and an edge between every pair of vertices for a total of $\frac{n(n-1)}{2}$ edges.*

A common problem in graph embedding is the minimization of the number of edge crossings. Usually, smaller graphs can be drawn on a plane such that no edges cross. However, more complex graphs demand surfaces that allow for more potential ways for edges to connect to vertices. Orientable surfaces such as spheres and tori provide a solution when graph drawings become more complex for larger graphs. As the number of holes increase in the torus, this allows for more options in how one can draw a graph on the surface while preventing edges from crossing.

Embedding graphs onto orientable surfaces has been a critical part of the study of topology. In 1866, Jordan discovered that the set of closed, orientable surfaces includes the sphere, torus, double torus, etc.[12]. Embedding a graph onto an orientable surface such as a torus requires a mapping such that no edges cross.

Definition 3 *An embedding of a graph $G = (V, E)$ onto a surface S consists of*

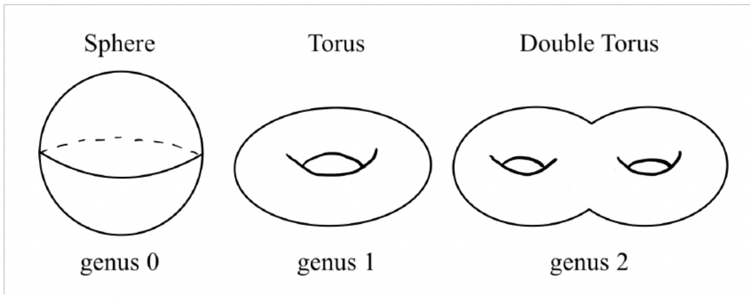

Figure 1: Orientable Surfaces

(1) a one-to-one function $f_V : V \to S$; and

(2) a continuous, one-to-one function $f_e : [0,1] \to S$ for each edge $e \in E$ such that e connects vertices v_0 and v_1, then $f_e(0) = v_0$ and $f_e(1) = v_1$ (or $f_e(0) = v_1$ and $f_e(1) = v_0$)

with the property that $f_{e_1}(x) = f_{e_2}(y)$ for any $x, y \in (0,1)$ implies $e_1 = e_2$ (and $x = y$).

A graph embedding divides the surface S into components, where each component is a face of the embedding [11]. When considering optimal graph embeddings, it is important to consider the amount of faces that are generated by the embedding. The number of faces that the embedding has will impact the Euler characteristic of the embedding, which is a number that "encodes" the number of holes required in the surface [13]. It is then very easy to calculate the genus of a surface which is essentially the number of holes in the surface.

Definition 4 *The Euler characteristic χ can be calculated using $\chi = |V| - |E| + |F|$ where $V, E,$and F are the number of vertices, edges, and faces, respectively.*

Definition 5 *The genus of an orientable surface g is determined by $\chi = 2 - 2g$.*

As the genus increases, the demand for a more complex surface also increases. Thus, in order to find an optimal embedding of a graph, we will want the genus of the surface to be as small as possible. In 1968, Ringel and Youngs discovered a formula that could calculate the genus of an optimal embedding of a complete graph K_n (which we denote $\gamma(K_n)$), therefore making it much easier to determine the simplest surface on which to embed a complete graph [14].

Definition 6 *The complete graph K_n has optimal genus*

$$\gamma(K_n) = \left\lceil \frac{(n-3)(n-4)}{12} \right\rceil$$

Now we introduce important definitions from group theory which will be prove to useful when performing arithmetic operations.

Definition 7 *A group G is a set with:*

(1) An associative binary operation $*$ (i.e., for all $a, b \in G$, $a * b \in G$);

(2) An identity element e (i.e., for all $a \in G, a * e = e * a = a$);

(3) And an inverse for each element (i.e., all $a \in G$ have an inverse $c \in G$ such that $ac = c*a = e$.

Definition 8 *A group G is abelian if the operation $*$ is also commutative (i.e., for all $a, b \in G, a*b = b*a$).*

For this paper, we will only be concerned with modular arithmetic, specifically within the finite cyclic group Z_n. Thus, addition will be the binary operation we use most frequently.

Definition 9 *Addition modulo n uses the integers $0, 1, ..., n-1$. Given an integer a, we define a mod $n = r$, where r is the remainder upon dividing a by n.*

Definition 10 *The finite cyclic group $Z_n = \{0, 1, ..., n-1\}$ is an abelian group under addition modulo n.*

In the abelian group Z_n, the identity element is $e = 0$ since $x + 0 = 0 + x = 0$. We define $-x$ to be the additive inverse of x and the order of a group element, $ord(x)$, to be the smallest positive integer m such that $mx = 0$. Note that $-x = n - x$ when working in addition modulo n.

We now introduce the fundamental definitions for understanding Cayley graphs and Cayley maps.

Definition 11 *Suppose H is a group with n elements and X is a subset of $H - \{e\}$ that is closed with respect to inverses. The Cayley graph $C_G(H, X)$ is a graph on n vertices, labeled by the n elements of H. The edges are determined by X : vertices g and h are adjacent if and only if there exists some $x \in X$ such that $g = h * x$.*

In Figure 2, we show that K_4 can be represented as the Cayley graph $C_G(Z_4, \{1, 2, 3\})$, where each of the edges are determined by the non-zero elements of Z_4. The blue edges represent the addition of a 1 in one direction and the addition of a 3 in the reverse direction. On the other hand, the red edges represent the addition of a 2 in both directions. Thus, all the vertices will be adjacent to one another since each vertex can reach every other vertex by the addition of another non-zero element, which ensures that the Cayley graph is a complete graph.

However, notice that there are edge crossings in the Cayley graph. When we want to embed Cayley graphs onto orientable surfaces without edge crossings, we instead opt to use Cayley maps.

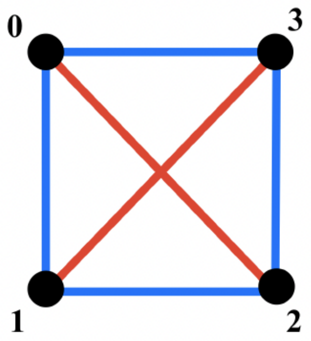

Figure 2: $C_G(Z_4, \{1, 2, 3\})$

Definition 12 *Cayley Map $C_M(H, \rho)$ embeds Cayley graph $C_G(H, X)$ onto a surface, where X is a closed subset of H and $\rho = (x_1, x_2, ..., x_k)$ is a cyclic permutation of X that gives the counterclockwise rotation of edges around each vertex.*

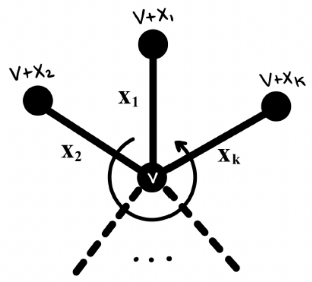

Figure 3: The counterclockwise rotation of edges around each vertex of a particular Cayley map

Figure 3 illustrates the counterclockwise rotation of edges around each vertex of a Cayley map with $\rho = (x_1, x_2, ..., x_k)$. It is possible to find optimal embeddings for many small complete graphs. For example, we will look at the optimal embedding of the Cayley map $C_M(Z_5, (1, 2, 4, 3))$ which optimally embeds on a torus.

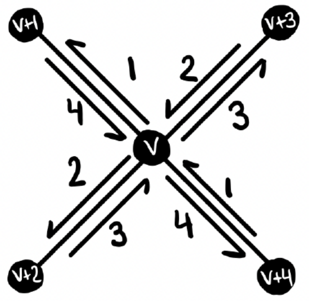

Figure 4: Counterclockwise rotation around each vertex in $C_M(Z_5, (1, 2, 4, 3))$

Taking a look at Figure 4, we can see that $\rho = (1, 2, 4, 3)$ determines the counterclockwise rotation around any particular vertex. By adding non-zero elements to the vertex in the middle, we can reach each of the other vertices in order to make the graph complete. Furthermore, like in the Cayley graph example shown earlier, each edge represents the addition of a non-zero number in the forward and reverse directions. If v represents a particular vertex in the Cayley map, then one must add 4 to v in order to form an edge with the vertex $v + 4$. Thus, in the reverse direction, one will need to add 1 to $v + 4$, which is also the additive inverse of 4, in order to return to v. This pattern holds true for any of the edges between the middle vertex v and the other surrounding vertices. Now that we have identified the general pattern around each vertex, we can start creating the full Cayley map for $C_M(Z_5, (1, 2, 4, 3))$.

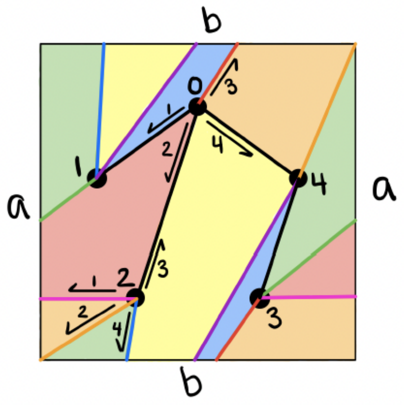

Figure 5: Counterclockwise rotation around each vertex in $C_M(Z_5,(1,2,4,3))$

In Figure 5, we illustrate the full Cayley map for $C_M(Z_5,(1,2,4,3))$. Note that since this Cayley map optimally embeds on a torus, edges can cross the boundaries of the rectangle in order to reach other vertices without crossing other edges. Each of the faces that are generated by the Cayley map are indicated in different colors. Thus, from this Cayley map we can see that $F = 5$, $E = 10$, and $V = 5$. Therefore the Euler characteristic $\chi = 0$, in which case the genus of the surface $g = 1$. We can verify that a one hole torus is the most optimal surface for the embedding since $\gamma(K_5) = 1 = g$. Also, notice that all of the faces generated are triangles.

Another way we can describe the faces of the Cayley map is by looking at λ, which is a permutation of disjoint cyclic permutations where each factor in λ describes an edge type. For example, Figure 5 has $\lambda = ((1,3,4,2))$, so there is only one single cyclic permutation which indicates that there will only be one type of face in the embedding. One can see that every face is generated by a cycle of the permutation $(1,3,4,2)$ so anytime there is an edge of type 1, that edge is followed by an edge of type 3, and so forth. The next definition is critical for understanding the relationship between ρ and λ.

Definition 13 *Suppose H is a group and X is a subset of H that is closed with respect to inverses. Then $\lambda(x) = \rho(x^{-1})$ (and therefore $\rho(x) = \lambda(x^{-1})$).*

Definition 3.10 gives us a way to move back and forth between λ and ρ in a way that will be very useful when we explain the search algorithms later in the paper. If λ_i is a cyclic permutation in λ, then $|\lambda_i|$ denotes the number of elements in λ_i. We also define the multiplicity of λ_i, $mult(\lambda_i)$, to be the order of $x_1 + x_2 + ... + x_n$ in Z_n. When defining λ, we will write $\lambda = \lambda_1 \cdot ... \cdot \lambda_m$ as a product of m disjoint cyclic permutations. Given a λ_i, $Face(\lambda_i) = |\lambda_i| \cdot mult(\lambda_i) = k$, where k will represent the type of polygon generated by λ_i. The number of k-gons can then be found using $\frac{p|\lambda_i|}{Face(\lambda_i)}$, where p refers to a prime n.

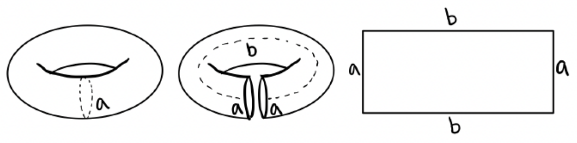

Figure 6: Folding a rectangle into a torus

Notice that the previous example produced an optimal embedding onto a torus. By starting with a rectangle, one can simply fold the long sides of the rectangle together, resulting in to openings which represent the short sides of the original rectangle. Then by connecting these two openings, one can create a torus out of a rectangle. Other surfaces can also be used for gluing together tori such as hexagons. In fact, the optimal embedding for K_7 requires folding a hexagon into a torus.

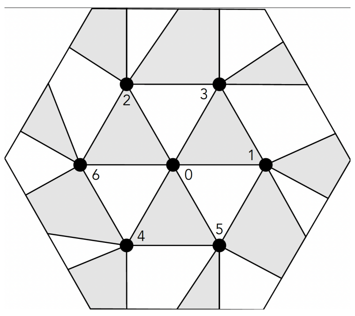

Figure 7: Optimal Embedding for K_7

As of now, it should be clear that the main optimization problem of constructing graph embedding involves minimizing the genus of the surface on which a graph is embedded. However, since the genus is directly calculated from the Euler characteristic, one must also consider how this value affects the genus. Recall that the Euler characteristic equation is $\chi = |V| - |E| + |F|$. From this equation, it should be clear that the number of vertices and edges will always remain constant for any Cayley map we make with a particular group Z_n. Also, $|V| - |E|$ will typically be a negative value, so thus the more faces we can generate in a graph embedding, the higher the Euler characteristic becomes (the closer it approaches 2, in which case the genus is 0). Since ρ and, therefore λ, have an effect on the number of faces generated on an embedding, one must try to maximize the number of faces by finding the most optimal ρ or λ that will result in the highest possible number of faces on the embedding. Typically, generating larger polygons will result in less faces, so reducing the size of the polygons will help to maximize the Euler characteristic. Then once the Euler characteristic is known, the equation $\chi = 2 - 2g$ tells us that an increase in the value of χ results in a lower genus.

In this paper, we will be working with the finite cyclic group Z_{12m+7} for $m = 0, 1, 2....$ Many of the these groups will be of the form Z_p, where p is prime. Scheinblum successfully proved that a Cayley map with the group Z_p can achieve an optimal embedding when all of its λ_i generate 3-gons [11]. While many of the algorithms we implement attack the the optimization problem by searching for ρ, Scheinblum's findings will be useful if we want to attack the problem by searching for optimal λ. Thus, we will base one of our genetic algorithms on the idea that we can form triplets in λ since there will be $12m + 6$ total elements in any particular group, which means that it is possible to group the elements into triplets. Then, we must confirm that the triplets are cyclic permutations that each have $mult(\lambda_i) = 1$. If such a λ exists for a Cayley map embedding with a group Z_{12m+7}, then that particular λ and its corresponding ρ will define an optimal graph embedding for K_{12m+7} onto an orientable surface.

2.2 Search Methods

One of the most common problems in computer science is the search problem. Usually, a search problem requires building a search algorithm that can find an optimal solution in a search space. A search space may be a structure or space of a specific problem domain. There are several classes of search algorithms that have developed in recent decades as the literature on search algorithms has expanded. Three of the most popular classes of search algorithms are enumerative, calculus-based, and random search methods [6]. An enumerative method can be thought of as an exhaustive search method that checks every possible candidate in a search space until it finds a solution. On the other hand, a calculus-based search method is inspired by the way in which local optima or global optima are found in calculus. This process usually involves working with the gradient of an objective function or using an iterative approach such as hill climbing to make progress toward a maximum or minimum solution. Finally, there are also random search methods which include genetic algorithms and other many other Evolutionary Programs mentioned in Section 1. This class of methods typically involves a probabilistic component but it is not necessarily limited to completely random techniques such as a random walk. Genetic algorithms use random choices as a way to effectively exploit a search space which still gives the random search a sense of direction [6].

In this section, we will provide background and the review of literature regarding relevant algorithms within these classes. For the purposes of this paper, we will only be concerned with brute force search methods (enumerative), hill-climbing methods (calculus-based), and genetic algorithms (random search). Furthermore, since the optimization problem in this paper is primarily a combinatorial optimization problem, we will focus on the feasibility of each search method in tackling such problems.

Brute Force Search

A common baseline algorithm for most search problems is a brute force search. A brute force search is an enumerative search method that exhausts all the potential solutions in a search space. Brute force algorithms are easy to implement as opposed to other search algorithms; however, their performance depends heavily on the size of a search space. This is especially the case when one must deal with problems in a specific domain. Thus, it is the job of a programmer to incorporate domain-specific knowledge into the brute force algorithm to improve its performance and perhaps cut down on the size of the search space. Basic pseudocode of a brute force search algorithm is layed out below in Algorithm 1. Note that the programmer has some discretion when it comes to choosing the stopping condition.

Algorithm 1 Brute force search

```
initialize the first candidate
while stopping condition is not met do
  if candidate meets condition then
    return candidate
  end if
  get next candidate
end while
```

For most problems where brute force search is used, one can think of the search space as a graph composed of initial states, goal states, functions defined on states or edges, and constraints [15].The most common brute force search techniques for graph traversals are depth-first search(DFS) and breadth-first search(BFS). DFS involves traversing a branch of a graph as far as possible until the algorithm must backtrack to previous states in order to find more branches to traverse. Thus, over time DFS will traverse each branch of the graph and cover the whole search space. BFS is an alternate search method that checks for solutions across each layer of a graph instead of directly traversing each branch one at a time. It is also not uncommon to combine DFS and BFS in a single brute force search algorithm so that the algorithm can benefit from the caution that BFS offers and the quick penetration of each branch that DFS offers.

Historically, brute force search has been a niche, only having significance in a small handful of domains such as number theory and games [15]. In number theory, mathematicians have used brute force search algorithms as way to provide evidence for conjectures. In the 1950s and 1960s, Derrick Lehmer utilized brute force search algorithms in order to prove theorems that demanded high amounts of computation [16, 17]. There have also been other cases in which brute force algorithms have assisted in making important discoveries including in the search for Mersenne primes and in proving the four color theorem [18]. Ultimately, the power of computation unleashed a whole new branch of mathematics: computational number theory.

There have also been extensive efforts to apply brute force search in solving games and puzzles. In the literature, chess has been the epitome of such applications since exhaustive search techniques can be useful in searching state spaces given a small amount of positional knowledge [15]. Other games where applications of exhaustive search have been successful include Qubic, Go-Moku, and Nine Men's Morris [19–21]. Many of the exhaustive search techniques used in games have also had applications in AI, where decision-making algorithms are crucial for determining the best moves a player can make in a game. There have also been efforts made in using exhaustive search to solve puzzles. One of the most classic puzzles is the Rubik's Cube. In 2010, mathematicians discovered that given any starting position on a Rubik's Cube, it would only take 22 or fewer moves to solve the Rubik's Cube. [22].

Outside of gaming and number theory, brute force search techniques have not been widely employed for optimization problems. As stated previously, such techniques fail greatly when the size of a search space increases. Without a heuristic or shortcut to help trim down the search space, a brute force search is almost guaranteed to be a memory intensive computational burden for problems involving large search spaces. Most combinatorial optimization problems will therefore require a more clever or heuristic approach in order to avoid the combinatorial explosion that arises from handling too many candidates in a search space.

Hill Climbing Algorithms

One of the most fundamental calculus-based search methods is the hill climbing algorithm. This algorithm is a local search method that analyzes only a portion of the total search space and converges to local optima quickly. The process of hill climbing can be attributed to the problem of finding maxima and minima of a function in calculus. Usually, an algorithm will start at a specific point in the search space and examine nearby neighbors. In order to compare the merit of each neighbor, an evaluation function will be used. If the algorithm is looking for a local minima, it will move to the neighboring state with the lowest value from the evaluation function, but if the algorithm is looking for a local maxima, it will move to the neighboring state with the highest value. The hill climbing process will continue iterating through the search space until it stops at a desired local optima. The algorithm knows that the state is a local optima if none of its neighbors have higher values(for local maxima) or lower values(for local minima) Hill climbing may be used in both discrete and continuous search spaces, but for the purposes of this paper we will focus on hill climbing in the context of discrete combinatorial optimization problems.

Algorithm 2 outlines a psuedocode example of a hill climbing algorithm in a discrete space. This form of hill climbing is also known as steepest ascent hill climbing since the algorithm analyzes every neighbor close to the current state instead of simply picking the first neighbor it finds that has a higher value.

Algorithm 2 Hill climbing algorithm

```
currentSolution = initialSolution
stopCondition = FALSE
while !stopCondition do
    neighbors = getNeighbors(currentSolution)
    nextValue = - infinity
    nextSolution = NULL
    for all members in neighbors do
        if value(member) > nextValue then
            nextSolution = member
            nextValue = value(member)
        end if
    end for
    if nextValue ≤ value(currentSolution then
        stopCondition = TRUE
    end if
    currentSolution = nextSolution
end while
```

Like the brute force search, hill climbing may not be suitable for problems in a wide variety of domains. If a search space contains various local optima, but very few or only a single global optima, then finding a best solution to a problem will be challenging. Furthermore, most of the functions that students work with in a calculus course are continuous and differentiable, making the process of finding local optima very easy. However, most optimization problems have some degree of noise or even discontinuity in the search space that prevents easy use of the hill climbing algorithm. Many search spaces can also contain structural problems for the hill climbing algorithm such as "plateaus" where values remain unchanged for many consecutive solutions, or "ridges" which force the hill climbing algorithm to zig-zag to a local optima. There also exists the danger of a search space having a very volatile structure where solutions often have extreme differences in their values. Thus, a global optima may only have neighbors with values at the opposite extreme, making it very difficult for there to be a gradual convergence to the global optima. For these reasons, along with the fact that the algorithm is limited to a local scope, hill climbing is considered to be less robust than other search techniques [6].

One modification that has often improved the performance of hill climbing algorithms is the addition of a stochastic component. When transitions are made at random, we consider the algorithm to be a stochastic hill climbing method. Given a set of neighbors, the algorithm will choose a neighbor at random chance, although some neighbors may hold higher weight over other neighbors if they are closer to the optimal solution. One must also consider what should happen if the neighbor that is chosen by the algorithm has a value that is worse than that of the current state. One approach is to retain the current state and repeat the process of choosing another neighbor in hopes that the neighbor will be a favorable state [23]. Another approach is to simply allow the algorithm to run for a definite number of iterations even if it makes moves that transition from a particular state to one that is worse than it. This approach also allows the algorithm to scan a broad range of the overall search space and one can simply keep track of the highest running value that the algorithm has found by the end of the simulation or simply stop the algorithm if it finds a global solution.

One commonly used analogy is to think of the pool of neighbors as a roulette wheel. The neighbors with values closer to the optimal solution should have a higher chance of being chosen, and thus one can think of each neighbor having its own chunk on a biased roulette wheel. However, this approach evaluates all neighbors of a current state though. If one wishes to cut down on computation, an alternate approach is to simply identify a random neighbor and accept the neighoring state with some probability that depends on how the neighboring state's value from the evaluation function differs from that of the current state's value [24]. With this approach, it is still possible to allow the algorithm to move freely to states even if their value is worse.

Stochastic hill climbing can be a vary useful algorithm that combines both calculus-based search techniques with random search. There have also been efforts made to compare the performance of stochastic hill climbing methods to other random search algorithms. Juels and Wattenberg recommended that stochastic hill climbing even be a baseline algorithm for genetic algorithms [23]. Many of the heuristics that are employed to implement stochastic hill climbing such as the roulette wheel can also provide the backbone for design choices in genetic algorithms.

Algorithm 2 presents a pseudocode example of the stochastic hill climbing algorithm. The transition probability is decided at the discretion of the programmer. As a starting point, Fogel and Michaelewicz provide the following function for calculating the transition probability [24]:

$$p = \frac{1}{1 + e^{\frac{eval(currentstate) - eval(neighbor)}{T}}}$$

Here, T acts as an additional parameter and its value typically depends on the nature of the problem. A higher value of T will make the search more random, but a lower value of T will make the algorithm more deterministic in nature.

Algorithm 3 Stochastic hill climbing algorithm

```
x = 0
select state at random
evaluate(state)
while t < maxIterations do
  select a neighbor at random
  transition to neighbor with probability p
  x = x + 1
end while
```

Although hill climbing algorithms are less robust than other search algorithms, there have been many examples of their success in solving combinatorial optimization problems. Traditional hill climbing algorithms have been used for producing one-factorizations of complete graphs and have outperformed genetic algorithms in solving graph coloring and bin packing problems [25, 26]. Stochastic hill climbing has also seen major success compared to other evolutionary programs in a variety of problems. Stochastic hill climbing has been a top-performer in automatic graph drawing problems, which have direct applications to software engineering and VSLI design [27]. There has also been a growing fascination in combining stochastic hill climbing with other Evolutionary Programs. One algorithm, titled "magnetic hill climbing", combines aspects of stochastic hill climbing with evolutionary strategies and has been used to solve the minimum cut graph partitioning problem [28]. Researchers have also combined evolutionary algorithms and stochastic hill climbing to solve the edge-biconnectivity augmentation problem [29].

In recent years, there have been efforts to evolve the basic hill climbing algorithm into a more robust and efficient method. One example is the development of Smart Hill Climbing (SHiC), which has made significant advances in optimizing application server configurations, agile dynamic mapping in many-core systems, and in finding better boolean functions for creating block and stream ciphers [30–32]. Improvements that make SHiC algorithms robust include their ability to cover global search spaces effectively, learning from past transitions, and making smart restarts in the search process [30]. Hill climbing has also been useful in optimizing machine learning and deep learning algorithms. In deep learning, hill climbing has outperformed other heuristic algorithms in training artificial neural networks for classification tasks [33]. Researchers have also developed multi-stage combinations of algorithms that combine hill climbing algorithms with learning algorithms such as K-means for optimizing sample allocation designs [34].

Genetic Algorithms

In this section, we will now introduce the primary metaheuristic search method that we implement in this paper, the genetic algorithm (GA). We'll begin by discussing the basic components of the GA and then review relevant literature that has provided key insights into the performance of GAs in solving combinatorial optimization problems. We'll also take a look at how the GA is related to other random search methods such as stochastic hill climbing.

The GA is a metaheuristic approach to solving optimization problems that takes inspiration from the biological processes that drive evolution. Such processes include genetic recombination and crossover, mutations, and reproduction. Arguably the most popular of all Evolutionary Programs, the GA stands out because it places heavy emphasis on the role of recombination over random mutations [35]. The reason is founded in the significance of genetic diversity, which is a result of recombination.

In a real-world population of animals, recombination causes genetic diversity to increase and over time individuals with the most favorable traits will be more likely to pass on their genes to future generations because of natural selection. Individuals who have favorable traits and can reproduce easily are deemed to be more "fit" than others. Similarly, we can think of encoded strings, usually composed of 0's and 1's, to represent the chromosomes of individuals in a computer simulation. By letting the individuals reproduce, we can allow the algorithm to practice natural selection and decide which patterns in these binary strings cause an individual to have a high fitness measure. Then, over a series of generations, the individuals with higher fitness will be more likely to keep passing on the patterns that made them more fit to begin with, thus causing every future generation to be slightly more fit than the previous generation. If the simulation continues for a large amount of generations, a convergence to an optimal individual should occur.

Compared to other heuristic search methods, the GA also stands out because of four significant features [6]:

- GAs work with encodings of parameters
- GAs search from a population of points
- GAs use objective functions
- GAs use probabilistic rules

The GA does not search for solutions in the way that a hill climbing algorithm would. While hill climbing may be dependent on continuity or differentiability, a GA simply works off of encodings of parameters. Usually, these encoded strings will be composed of 1's and 0's. These strings then form the chromosomes of a population. As a population-based approach, the GA handles a large number of chromosomes which maintains a great deal of diversity in a population. This approach allows the GA to let natural selection act on the whole population and future generations so that favorable patterns can be preserved and optimal chromosomes can be produced through the recombination process of solutions that are more fit.

The GA also incorporates the use of objective functions. We'll define the fitness function f to be an evaluation function that determines the merit that an individual has in a population. The fitness function ultimately decides how likely an individual is to reproduce with other individuals and pass on its traits to future offspring [6]. The fitness of an individual has a great impact on how one programs the reproduction operator, which we will discuss later in more detail.

Finally, the GA is classified as a random search method. Thus, GAs do not incorporate deterministic rules like the hill climbing algorithm or traditional brute force approaches. Because the process of evolution is partly driven by chance, so do GAs depend on randomness to determine which chromosomes are more optimal and will be able to pass on their traits to future offspring. Probability is also central to the mutation operator, which is a key step in the implementation of a GA.

At the heart of a genetic algorithm are three fundamental components [36]:

- A selection operator
- A crossover operator
- A mutation operator

We'll now review the details of each step in the GA in more detail while keeping our focus on the 3 major operators. Following this discussion, we will include a pseudocode example of a basic GA.

The Selection Process and Reproduction

After the objective function and the structure of the chromosomes are understood, the first task is to figure out how the selection process will work for reproduction. The reproduction (or selection) operator acts as an artificial version of natural selection since chromosomes with higher fitness values will be more likely to be selected for reproduction in the simulation, thus making it possible for offspring to carry on certain qualities that are favorable [6]. Once the programmer has initialized a population of chromosomes, the fitness of each chromosome will need to be assessed using the objective function. Although there are several selection methods, the most commonly used method is "fitness-proportionate selection" with a roulette wheel [4]. One way to partition the roulette wheel is to scale the fitness values of each chromosome in such a way that slightly favors those with higher fitness values, and thus when chromosomes are chosen for reproduction, it will be more likely for the algorithm to choose the more optimal chromosomes.

The selection process can have a very significant impact on the rate of convergence to an optimal solution. It is important to consider the "exploitation/exploration" balance since if the selection process is too biased toward optimal chromosomes, it is likely that future generations will lose genetic diversity necessary for further progress in the convergence process [4]. However, if the selection process is too reserved in its bias, convergence to a desirable solution will take much longer. It is also important to note that in a traditional GA, no chromosomes are carried over to subsequent generations after they undergo the reproductive process. One way to save high-performing chromosomes to be individuals of subsequent generations is to implement a technique called "elitism" [4]. Also, note that selection operator chooses chromosomes without replacement, which allows chromosomes to be chosen more than once [37].

As one can see, the nature of the reproduction operator is vital for the performance of the GA. It is up to the programmer to make the necessary adjustments so that the chromosomes in the algorithm can gradually converge to an optimal solution. For more on selection techniques, see Melanie Mitchell's *An Introduction to Genetic Algorithms* [4].

Crossover and Recombination

After selecting two chromosomes for the mating process, the crossover operator must be considered. Compared to other operators, the crossover operator is likely the main distinguishing feature of the GA since recombination, as a result of crossover, is very powerful for exploring a search space while helping the algorithm learn which traits of a chromosome should be preserved over others [4]. Typically, the crossover operator will perform some rearrangement of subsequences of the parent chromosomes so that the child chromosomes, or offspring, have traits from both parents. Techniques for crossover range from very simple methods like single-point crossover to more advanced methods which may be more specific to problems in certain domains.

The most traditional approach for crossover is the single-point crossover. Given two parent chromosomes, the operator will choose a random location on the bit strings to be a cutoff point. This process will produce two subsequences for each parent chromosome which will then be rearranged so that first half of the first parent chromosome will attach to the second half of the second parent chromosome to produce the first child chromosome. Likewise, the first half of the second parent chromosome will attach to the second half of the first parent chromosome to produce the second child chromosome.

Single-point crossover, although easily to implement, has its downsides. Two common problems that can arise are "positional bias" and the "endpoint effect" [4]. Positional bias is a concept that refers to the way in which schemas produced by parent chromosomes depend on the location of the bits in the chromosome [38]. The single-point crossover method may disrupt bits that interact with one another if the bits are located far away from each other. Thus, the method assumes that certain short schemas are what form the building blocks of a string, but usually one will not know how the ordering of bits affects the functional relationships between them [4]. Another common problem is the "endpoint effect", which happens when the endpoints of the parent chromosomes always show up in their subsequences. Therefore, it is likely that future generations will maintain these "vestigial" structures.

One alternative approach to single-point crossover is the double-point crossover method. Basically, this approach requires forming two cutoff points. It is then a common practice to take the middle chunk of one of the parent chromosomes and place it in between the first and third subsequences of the second parent chromosome to form the first child chromosome. Then, the middle chunk of the other parent's chromosome can be placed between the first and third subsequences of the other first parent chromosome to form the second child chromosome. Although double-point crossover can protect against positional bias and endpoint effects, it is still not an ideal approach for many problems. The literature on crossover methods is quite vast and includes more complex crossover methods such as "parameterized uniform crossover" and "partially mapped crossover" [39, 40].

The Mutation

Just like in the process of evolution, we can create mutations in our chromosomes in order to generate random variation in a population. The reasoning behind the addition of a mutation operator is that sometimes a program may converge slowly and require an extra push from another source of variation. The operator thus gives the GA a small chance to hop around the search space and possibly escape the traps of local optima [37]. The basic mutation operator involves randomly flipping bits in a chromosome with some small probability like 0.001 [36]. Once again, this operator can be tuned if the programmer believes that more exploration will help the program converge sooner. However, the probability of mutation should remain low. It is widely accepted that mutation is simply a secondary mechanism by which the GA can form adaptations [6].

Below, Algorithm 4 presents a pseudocode example of a basic GA. In Section 3 we will discuss more specifics regarding the implementation of specific operators and methods in the algorithm.

Algorithm 4 Genetic algorithm

```
k = 0
$P_k$ = Initial population of random individuals
evaluateFitness(individual) for each individual $\in P_k$
while k < maxIterations do
    Select individuals from $P_k$ for reproduction
    Recombine individuals
    Mutate individuals
    evaluateFitness(new individuals)
    $P_{k+1}$ = new individuals
    k = k + 1
end while
```

As stated in Section 1, the GA has been applied to a wide scale of problems. GAs have been used at General Electric for aircraft design, Los Alamos National Lab for work on satelite images, John Deere for assemby line optimization, and even in generating computer-animated horses in the hit movie *The Lord of the Rings: The Return of the King*, to name a few [41]. However, it is also worth taking a look at the literature regarding the GA's performance in solving combinatorial optimization problems to get an idea for when GAs are a preferred option over other search methods.

The most famous benchmark for the algorithm's feasibility and applicability is arguably its performance in solving the Traveling Salesman Problem(TSP), which is problem that requires one to find the shortest route on a graph of cities such that all cities are visited once but the route ends where it started. Once GAs become more commonly used search methods, the focus was set on developing optimal crossover operators for TSP. Following the work of Goldberg and Holland, many researchers developed complex crossover techniques that provided promising results such as heuristic crossover (1985), partially-mapped crossover (1985), order crossover and cycle crossover (1987), edge recombination crossover (1990), matrix crossover (1992), DPX crossover (1996), and edge assembly crossover (1997) [6, 42–47]. Since the crossover operator generates the most variation in a population compared to the other operators, most of the success of a GA depends on the efficiency of crossover and its ability to explore a broad area of the search space.

In recent literature, there has been growing interest in developing parallel genetic algorithms and as well as hybrid algorithms that can compete with more state-of-the-art optimization algorithms [48–50]. Essentially, a parallel algorithm can execute several tasks in a program simultaneously by taking advantage of multiple processing units, thus dividing the total amount of work into smaller, more manageable pieces. Once the parallel processing units finish, the algorithm will output a single result much faster than if there was only one processing unit available. Highly parallelized GAs have been successful at solving TSP for intelligent transportation systems, making it much easier for autonomous vehicles to make swift decisions in time-constrained problems [49].

Although GAs may perform well on their own in some cases, many researchers have also incorporated GAs into hybrid optimization algorithms to improve upon traditional GAs and compete with top-performing optimization algorithms. In one example, a hybrid genetic algorithm significantly out-performed a traditional genetic algorithm approach in solving TSP using Android Google Maps, especially as problem complexity increase [50]. Even more exciting for the state of GA research is the prospect of more hybrid optimization algorithms incorporating both GAs and machine learning algorithms. The hybridization of GAs and Multiagent Reinforcement Learning (MARL) has generated a powerful combination for solving TSP by letting the GA act as a tour improvement heuristic and MARL to act as a connection heuristic [48]. Such hybrid algorithms have been able to out-perform more state-of-the-art algorithms in solving TSP. Hybrid GAs have also been successful at solving other common combinatorial optimization problems that have had major implications for operations research including flow-shop scheduling, the three-index assignment problem, and the well-known quadratic assignment problem [51–53].

It is worth mentioning some cases in which GAs are not suited for certain optimization problems. For the most part, it is not recommended to use a GA if a search space is small, is smooth or unimodal, or has a structure that is well understood [4]. When search spaces are small, it is not worth implementing a complex algorithm such as a GA when a brute-force approach may be able to find a global optima much faster. Furthermore, GAs work very well in noisy environments, but they tend to fail compared to hill climbing techniques when dealing with smooth search spaces. Hill climbers can exploit the smoothness of such search spaces and thus approach a global optima much faster. When the structure of a search space is well understood, it is not efficient to use a GA since other domain-specific algorithms will better incorporate domain-specific knowledge, thus making those algorithms more efficient at finding global optima. However, these considerations are not necessarily the end all be all for deciding on the use of GAs for solving a particular optimization problem. The wide variety of research on GAs has proven that the method can incorporate very efficient design choices that enhance their performance and help them out-perform other modern algorithms, thus maintaining the trend of robustness that GAs have demonstrated since the days of Holland and Goldberg.

3 Computational Methods

In this section, we lay out the implementation of the computational methods used for solving the optimization problem we introduced at the end of Section 2.1. In Section 3.1, we will explain our baseline approach to solving the problem which involves the use of a brute force algorithm. Then, in Section 3.2, we will explain two hill climbing algorithms that we built in order to better understand the structure of the search space. Finally, in Section 3.3, we will end this section with the implementation of two GAs that tackle the optimization problem from different angles. For relevant code, see the Appendix section.

3.1 Brute Force Algorithm

Given a positive integer n, the brute force algorithm will search for an optimal Cayley map embedding for the graph K_{12m+7}. In order to determine if a solution is optimal, we base our search on the optimal genus for a complete graph K_n given by Ringel and Youngs that we mentioned in Section 2.1:

$$\gamma(K_n) = \left\lceil \frac{(n-3)(n-4)}{12} \right\rceil$$

.

For a given Cayley map $C_M(Z_n, \rho)$, recall that ρ represents a cyclic permutation of elements from 1 to $n-1$, and that ρ will determine the λ, which impacts the number of faces generated by the Cayley Map. Our brute force algorithm will search for a optimal ρ by checking permutations that will be stored in lists. The starting point for the search is the smallest permutation, and each subsequent permutation that is evaluated will be the next largest permutation lexographically. The algorithm will only need to check $(n-2)!$ permutations in the worst case as long as each permutation always begins with 1. As the algorithm runs, it keeps track of the most optimal ρ that it has found so far, and if it finds a ρ with genus equal to the optimal genus determined by the Ringel and Youngs definition, then the algorithm will stop and return the optimal solution for that particular Cayley map.

Most of the logic in the algorithm takes place in the *calculate_genus*() function which will be a fundamental function for other algorithms we present later in this section. This function will also act like an objective function for evaluating fitness. The basic idea behind the function is to compute the function $\lambda(x)$ for the value in ρ and from there we use the set of formulas we defined in Section 2.1 to calculate the number of faces that λ generates. Then, once we have the number of faces, edges, and vertices, we can calculate the Euler characteristic of the Cayley map and then the genus. Below, we list the key steps in calculating the genus given a ρ:

1. Build two dictionaries. One that will hold mappings of the function $\lambda(x)$ and one that will keep track of which elements of ρ have already been evaluated (set all values to 1).
2. Iterate through each value in ρ and save a mapping from each $x \in \rho$ to $\lambda(x)$, which we found to be equal to $\rho(x^{-1})$. For our approach we save the $\lambda(x^{-1})$ at each iteration since we know that $x^{-1} = n - x$ and thus we can map x^{-1} to $\rho(x)$, which is simply the value following x in ρ.
3. Calculate the number of edges using $\frac{n(n-1)}{2}$ and save the number of vertices, which is equal to n.
4. Find the number of faces generated. This requires iterating through the keys in the second dictionary that was created in step 1 and then calculating $Face(\lambda_i)$ for each $\lambda_i \in \lambda$. Then one can calculate the number of faces that each λ_i generates by doing $\frac{n|\lambda_i|}{Face(\lambda_i)}$
5. Calculate the Euler characteristic using $\chi = |V| - |E| + |F|$ and then obtain the genus using $g = \frac{\chi-2}{-2}$.

For example, suppose we want to find an optimal embedding for a Cayley map with the group Z_7, and suppose we set $\rho = (1, 2, 3, 6, 5, 4)$. First, we will want to figure out what λ looks like so that we can divide up our elements into their corresponding λ_i. Using the fact that $\lambda(x) = \rho(x^{-1})$, we have

$$\lambda(1) = \rho(6) = 5$$

$$\lambda(5) = \rho(2) = 3$$

$$\lambda(3) = \rho(4) = 1$$

.

Now we can obtain $\lambda_1 = (1, 5, 3)$. By performing a similar process on the remaining values in ρ we can find that $\lambda_2 = (2, 4, 6)$. Thus we have the product $\lambda = ((1, 5, 3), (2, 4, 6))$, so now we can calculate the number of faces that each factor generates. For λ_1 we can see that $1+5+3 = 2$ and since 2 and 7 are relatively prime, then $ord(2) = mult(\lambda_1) = 7$. Since $|\lambda_1| = 3$, then $Face(\lambda_1) = 3 \cdot 7 = 21$. Then, by calculating $\frac{n|\lambda_1|}{Face(\lambda_1)}$, we can see that the number of faces generated by λ_1 is 1. We can perform similar calculations to see that the number of faces generated by λ_2 is also 1. Thus, only 2 faces are generated by the Cayley map.

Since $E = \frac{7(6)}{2} = 21$ and $V = 7$, we have $\chi = 7 - 21 + 2 = -12$. Thus, $g = \frac{-12-2}{-2} = 7$. According to the Ringel and Youngs definition, this is not an optimal embedding since $\gamma(K_7) = 1$. As you can see, this approach does not say much about the structure of the search space as a whole. To better understand the search space, it is more appropriate to implement a calculus-based search such as hill climbing.

3.2 Hill Climbing Algorithms

In this section, we'll discuss two different approaches to building a hill climbing algorithm for our optimization problem. First, we will discus the implementation of the traditional hill climbing algorithm that we employ in this paper. Then, we will also consider a stochastic hill climbing algorithm since it may act as a good predictor for the performance of the GA that we will discuss in Section 3.3.

Traditional Approach

Instead of searching through possible permutations of ρ in lexographic order, the hill climbing algorithm instead picks a random permutation in the search space and begins a local search. Our algorithm will continue to use the same *calculate_genus*() function that was implemented in the brute force approach as a way of measuring the fitness of a ρ. In order for the algorithm to determine the next neighbor to transition to, the algorithm will need to figure out all the nearby neighbors, evaluate each of their fitness values, and then choose the neighbor that has a fitness value better than the current ρ. In our case, the algorithm will choose a ρ with a fitness value lower than the current ρ since a lower genus is more optimal. If the algorithm reaches a ρ and its neighbors all have genus values greater than that of ρ, then it will stop and return the current permutation and its genus.

In order to implement this approach, we must also define what it means to be a neighbor to a particular ρ. Suppose we are trying to find an optimal embedding for $C_M(Z_7, \rho)$ and that we begin our search with $\rho = (1, 4, 5, 2, 3, 6)$. A neighbor of ρ can be found by making a single swap between adjacent elements in ρ. One neighbor may have the first two elements swapped and another may have the first and last elements swapped. Thus, there will be 6 neighbors in total:

$$(4, 1, 5, 2, 3, 6)$$

$$(1, 5, 4, 2, 3, 6)$$

$$(1, 4, 2, 5, 3, 6)$$

$$(1, 4, 5, 3, 2, 6)$$

$$(1, 4, 5, 2, 6, 3)$$

$$(6, 4, 5, 2, 3, 1)$$

.

This approach is useful for understanding the structure of the search space. As long as permutations begin with the element 1, then there will be $(n-2)!$ possible solutions, so for much larger groups such as Z_{115}, it is important to exploit the structure of the search space as much as possible to improve the performance of the search. If the hill climbing algorithm finds that there are many local optima but few global optima, then it may not be helpful to depend on a hill climbing approach. Instead, a stochastic approach may prove to be more useful.

Stochastic Approach

The stochastic hill climbing algorithm can often be a useful baseline for determining the likely performance of a genetic algorithm. In our approach, neighbors are defined the same way as in the traditional hill climbing algorithm. However, now the search can transition randomly from permutation to permutation even if the new permutation has a higher genus than the previous one. The process works similar to a roulette wheel. When the algorithm begins, it will evaluate all of the genus values of the neighbors close to the starting point and then build a biased roulette wheel based on these values. Since many neighbors may share the same genus, we will cluster similar neighbors into their own intervals on the roulette wheel. Since the wheel is biased in favor of more optimal permutation, it is much more likely for a group of neighbors with genus 1 to be chosen randomly than a group of neighbors with genus 5. After a group is chosen randomly, a random permutation is then chosen from that group of candidates and the permutation that is chosen becomes the next permutation.

The algorithm will continue to make transitions even if the transitions aren't favorable in terms of reducing the value of the genus. Furthermore, it is up to the user to decide how many iterations the algorithm should run. This design choice allows the stochastic hill climbing algorithm to have freedom to scan the search space as much as possible. Other design choices of the stochastic hill algorithm will have a significant influence on the development of the GAs in the next section. As stated previously in the paper, stochastic hill climbing algorithms often provide the building blocks for more complex random search methods like GAs.

3.3 Genetic Algorithm

This section will review the implementation of two different GAs that we will use to solve our optimization problem. In the first approach, we continue to search across all possible permutations of ρ to find a optimal Cayley map embeddings. This method builds upon the previous building blocks that were established in our baseline algorithms. The second approach attacks the problem by searching for optimal λ permutations. In Sections 4 and 5, we will explain in more detail the the differences between searching for ρ and searching for λ and why one of the algorithms may have more powerful implications than the other.

The ρ Approach

Given a finite cyclic group of the form Z_{12m+7}, the first GA will find a ρ that produces an optimal Cayley map embedding according to Ringel and Youngs definition. Regarding the design of the algorithm, we will use object-oriented development. The Individual class will keep track of the fitness and permutation attributes of an individual as well include methods such as *mutate*() and *calculate_genus*(). There will also be a Population class that contains important attributes such as the list of individuals in the population, a list that keeps track of the best genus values in each generation, a list that keeps track of the average genus in each generation, and the most optimal permutation found so far in the search. In the Population class there are also several important methods that we list below:

- *metrics*(): After a generation of individuals is generated, this method will evaluate the fitness of each individual and keep track of the best genus, calculate the average genus, and update the best permutation if a more optimal permutation is found in the generation. This method will also update the optimal genus found so far in the search too.
- *select_individual*(): This method will serve as the selection operator for the algorithm. An individual will be randomly chosen to be reproduce using a roulette wheel selection.
- *save_individuals*(): This method will save the top 10% of individuals with regards to fitness so that these individuals can remain in the subsequent generation.
- *mate*(): The *mate*() method will control the reproduction process between two individuals. The crossover operator will be based on a double-point cutoff method In order to perform crossover, it will call on *crossover*().
- *crossover*(): Given substrings from the previous parents, this method will perform the crossover operation and return a child chromosome.

- *generate_population*(): This method replaces the old generation with the new one using the 3 core operators. All of the other methods except *metrics*() and *results*() will be called in order to complete the reproduction and replacement process.
- *results*(): This method will print out the most optimal rho and its fitness as well as the average and best rho for each generation. Also, this method will print out the individuals in the final generation.

There are three core operators in the algorithm: selection, crossover, and mutation. The selection operator will involve use of a biased roulette wheel selection process similar to the one used in the stochastic hill climbing algorithm. To add another degree of bias in the algorithm, the top 10% of individuals in each generation will carry over to the next generation to increase the rate of convergence to an optimal solution. In terms of crossover, we use a double-point crossover method that cuts at two points and then rearranges substrings to create the offspring chromosomes. Below, we outline the steps of the double-point crossover method:

1. Starting with two parent chromosomes, find two different indices and cut each chromosomes at those two indices.
2. Take the middle chunk from the chromosome of the first parent and replace the middle chunk from the chromosome of the second parent.
3. Make any necessary substitutions in case duplicate values are present in the new chromosome.
4. Repeat the process for the other child chromosome but instead replace the middle chunk from the first parent chromosome with that of the second parent chromosome.

Let's take a look at an example to examine the problem of duplicates that may occur with this method and how to solve it. Suppose we have two parent chromosomes, $(1, 4, 6, 3, 5, 2)$ and $(1, 5, 3, 6, 2, 4)$ and let one cutoff point be between the 3rd and 4th element and other cutoff point be between the 5th and 6th element. Thus we have the following substrings:

$$\text{Parent 1} : (1, 4, 6)(3, 5)(2)$$

$$\text{Parent 2} : (1, 5, 3)(6, 2)(4)$$

We'll then replace the middle substring of the second parent chromosome with the middle substring of the first parent chromosome:

$$\text{Child 1} : (1, 5, 3)(3, 5)(4)$$

Here, we can see that 3 and 5 are duplicate elements so the child chromosome is not a valid ρ. However, we can substitute the 3 in the first substring with a 6 since we know the 3 from the middle substring of the first parent chromosome replaced a 6 that was in the middle substring of the second parent chromosome. Similarly, we can substitute the 5 in the first substring with a 2. Thus, the first child chromosome will look like $(1, 2, 6, 3, 5, 4)$. The second child chromosome will look like $(1, 4, 3, 6, 2, 5)$.

The mutation operator will mutate a child chromosome at a probability of $p = 0.005$. If a chromosome undergoes a mutation, two random elements in ρ will swap positions. Both the crossover and mutation operators are likely to aid the algorithm in exploring the search space while the selection operator works on exploiting the search space. The double-point crossover has potential to make very different child chromosomes, and thus cover a very wide range of solutions. Similarly, the mutation operator can increase the amount of genetic diversity in a population of individuals and occasionally help the algorithm escape local optima.

The GA is not a complex metaheuristic search algorithm, and thus it lends itself to problems that may have search spaces that are not well understood. The implementation is quite simple and straightforward compared to other state-of-the-art algorithms or domain-specific search algorithms. However, with an alternate approach we may be able to simplify the optimization problem even further. Scheinblum's conclusion about finding optimal Cayley map embeddings using 3-gons in λ may provide enough domain-specific knowledge to streamline the search process for the genetic algorithm. However, if we're going to implement such an algorithm we will need to change our approach.

The λ approach

As mentioned earlier in Section 2.1, if we are working with a Cayley Map $C_M(Z_p, \rho)$ where p is prime, then we can find an optimal embedding as long as λ only generates 3-gons. Since our understanding of the structure and implications of λ is more comprehensive than that of ρ , we can use domain-specific knowledge to build an alternative approach in our search for optimal Cayley map embeddings. Thus, this approach has potential to cut down on the pool of possible solutions to something that is more manageable for the algorithm and less intensive.

The first problem that must be considered with this approach is the construction of chromosomes. One could build random permutations of λ where each λ_i contains 3 elements, but that would not guarantee that each λ generates a 3-gon. Scheinblum found that if the $mult(\lambda_i) = 1$, essentially meaning the elements in λ_i add up to zero, then the graph could always be optimally embedded for $C_M(Z_p, \rho)$ [11]. Recall that $Face(\lambda_i) = mult(\lambda_i) \cdot |\lambda_i|$. If we know that $mult(\lambda_i) = 1$ and $|\lambda_i| = 3$, then clearly $Face(\lambda_i) = 3$, and therefore that particular λ_i generates 3-gons. In order to construct chromosomes, we will instead take an alternative approach. We will incorporate a brute force search that looks for sets of triplets that add up to zero into our GA, thus making our algorithm a hybrid GA algorithm.

Instead of making our chromosomes the elements in λ, we will leave that to the brute force portion of the algorithm. Our brute force search is inspired by Scheinblum's brute force algorithm that generates the λ and ρ of optimal Cayley map embeddings for complete graphs of the form K_n where $n = 1 \bmod 3$ and $n > 7$ [11]. The number of elements in λ must be divisible by 3 in order to form triplets and since 0 is not an element that we need to consider, we require $n = 1 \bmod 3$. The algorithm is essentially a depth-first search algorithm that incorporates backtracking to find sets of triplets where each set is of the form (i, j, k) such that $i < j < k$ and $i + j + k \equiv 0 \bmod n$. Although these details are necessary, it is still possible to generate a ρ that is not one cyclic permutation containing all $n - 1$ elements. Thus, a λ that is found by the brute force search may not generate a valid Cayley map embedding.

Once a potential set of triplets is found by the brute force algorithm, the hyrbid algorithm will begin the GA portion of the search. Instead of our chromosomes representing λ, they will instead be an encoding of the orientations of each triplet in λ. For example, suppose we are working in Z_7 and $\lambda = ((4, 2, 1)(6, 5, 3))$. We can find ρ by using the fact that $\rho(x) = \lambda(x^{-1})$. Thus, $\rho = ((1, 5)(2, 3)(4, 6))$, which is not a valid ρ since it is does not contain all $n - 1$ elements in a single cyclic permutation. We'll define the orientation of a triplet to be the order of the elements, so we'll encode an unreversed triplet as a 0 and a reversed triplet as a 1. In our example we would encode the chromosome for $((4, 2, 1)(6, 5, 3))$ as $[0, 0]$. Then, if we want to randomize orientations of the triplets we could form a new λ using the chromosomes $[1, 0]$, which produces $\lambda = ((1, 2, 4)(6, 5, 3))$. We can then see that $\rho = (1, 5, 4, 6, 2, 3)$, which is a valid ρ. Therefore, this λ and ρ define an optimal Cayley map embedding for K_7.

The goal of the GA portion of the algorithm is to find optimal λ and ρ by discovering different combinations of orientations for each cycle in a given set of triplets, which is found by the brute force portion of the algorithm. If a λ does produce a valid ρ, then there is no reason to calculate the genus since we already know that the embedding should be optimal according to Ringel and Youngs definition. Thus, the evaluation function in the λ approach will not track the genus of each individual, but instead the number of cycles in ρ, which is generated by the set of triplets. Hence, individuals with the highest fitness will only have a single cycle in ρ. The algorithm will process a certain amount of generations, searching for the optimal orientation of triplets that make a valid ρ. After the GA finishes, the brute force search will then re-take control of the search and find the next set of triplets. The GA will then continue and the process will repeat until all of the possible sets of triplets have been tested in a GA for optimal orientations of their cycles.

The hybrid GA will maintain a similar object-oriented structure used in the first GA and contains two majors classes: the Population class and Individual class. Individuals will contain attributes that keep track of fitness, the set of triplets, the orientation of triplets, and a map that saves all of the $\rho(x)$ mappings for each value in the set of triplets. Like the previous algorithm, the Population class will save attributes such as the best fitness in each generation, optimal lambda found so far in the search, average fitness in each generation, and the overall optimal fitness found by the algorithm. Below, we list the significant functions in the Population class:

- *metrics*(): This method will save the average fitness, best fitness, and optimal lambda in each generation.
- *analyze*(): This method produces a convergence plot that plots the average fitness and best fitness for each generation.
- *select_individual*(): This method will employ a biased roulette wheel selection method to choose individuals for reproduction.
- *save_individuals*(): This method saves the top 10% of individuals in a generation to be carried over to the next generation.
- *mate*(): This method executes the crossover operator between two parent chromosomes using a single-point crossover method.
- *generate_population*(): This method replaces the old generation with the new one using the 3 core operators. All of the other methods except *metrics*() will be called in order to complete the reproduction and replacement process.

The major differences between this approach and the first GA are in the implementation of the crossover operator and mutation operator. Instead of using a double-point crossover, we will use a simple single-point crossover. A random index will be chosen and a cut will take place at that location in each of the parent chromosomes. Then, the first child chromosome will be composed of the first substring of the first parent chromosome along with the second substring of the second parent chromosome. Similarly, the second child chromosome will be composed of the first substring of the second parent chromosome with the second substring of the first parent chromosome. For example, suppose the first parent chromosomes is $(1, 0, 0, 1, 0, 0)$ and the second parent chromosome is $(0, 0, 1, 1, 1, 0)$. If we perform a single cut after index 3 then one of the child chromosomes will be $(1, 0, 0, 1, 1, 0)$ and the other one will be $(0, 0, 1, 1, 0, 0)$.

The mutation operator also works slightly different in this approach. Given a child chromosome, there will be a small probability (.001) that a bit will flip. Since we assume each mutation event to be independent, this probability will be the same for each bit. On the other hand, the selection operator essentially remains the same in both GAs with exception of the scaling operation for making the roulette wheel biased.

By making a hybrid GA, we can leverage a useful amount of domain-specific knowledge in our search for optimal Cayley map embeddings. Since the first GA was searching for permutations of ρ, any optimal solution would have to be found within a search space containing $(n-2)!$ possible solutions. Now, with the λ algorithm, most of the computational load will be placed on the brute force algorithm. After finding a set of triplets, the GA will only need to search $2^{\frac{n-1}{3}}$ total possibilities for optimal orientations of a set of triplets. Once a λ that produces a valid rho is found, we will have a guaranteed solution that optimally embeds a complete graph of the form K_{12m+7} onto an orientable surface.

4 Results

In this section, we present results of the proposed GAs as well as the hill climbing and brute force algorithms. We'll evaluate the effectiveness and efficiency of an algorithm based on its ability to find optimal solutions in a reasonable amount of time. We'll also take a look at how each algorithm performs for more complex Cayley maps.

In our results, we'll compare the brute force algorithm (BF), hill climbing algorithm (HC), stochastic hill climbing algorithm (SHC), genetic algorithm (GA), and hybrid GA algorithm (HGA). Since the performance of each algorithm is dependent on the complexity of each graph, we started by looking at how each algorithm performed on a small graph embedding. Table 1 presents the average time it took for each algorithm to find an optimal Cayley map embedding of the graph K_7. Note that this time may not be the full execution time of the program if the algorithm was able to find an optimal embedding before it finished (e.g. GAs will continue running until the last generation has been evaluated). In searching for optimal solutions of the graph K_7, all of the algorithms consistently found optimal graph embeddings except the HC algorithm. The overall average genus found for the HC algorithm was nearly 4, although in cases where it did find an optimal solution, the average time it took for the program to find a solution was 0.003 seconds.

Results for K_7	
Algorithm	Average Time to Find Solution (sec)
BF	0.016
HC	0.003
SHC	0.001
GA	0.001
HGA	0.005

Table 1: Average time spent finding solutions for optimal embeddings of K_7. The optimal genus $\gamma(K_7) = 1$.

The baseline embedding K_7 also gave us the opportunity to better understand the basic structure of its search space since the BF algorithm could efficiently search every possible permutation. In Figure 8, we graphed all of the permutations of ρ on the x-axis (every 4th tick is shown), and their genus values on the y-axis. The structure of the search space has a major effect on the performance of traditional hill climbing algorithms and brute force algorithms. As graphs became more complex, the structures of larger search spaces continued to have few global optima, causing the performance of these two algorithms to be much worse compared to other algorithms.

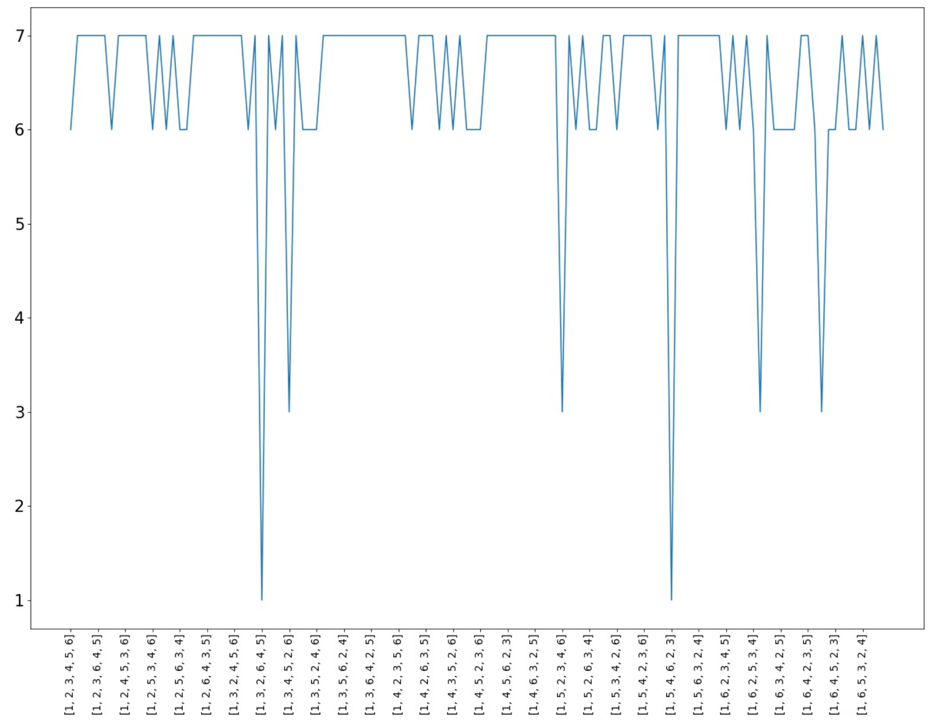

Figure 8: The search space for K_7.

We then gradually increased the complexity of the graphs and compared certain algorithms in their ability to find optimal embeddings of K_{19}. At this point, the BF algorithm became computationally intensive and the time it took to run the program quickly surpassed the reasonable capacity of standard computers. Thus, in Table 2, we compare all of the algorithms except the BF algorithm. In terms of metrics, we keep track of each algorithm's average genus found, best genus found, and average execution time of the program. By this point, not every algorithm was finding optimal solutions consistently, so it was apt to analyze these specific metrics and get a sense of the balance between execution time and ability of an algorithm to find optimal or near-optimal solutions.

Results for K_{19}			
Algorithm	Average Execution Time (sec)	Average Genus	Best Genus
HC	0.004	60	56
SHC	0.213	45	39
GA	0.260	40	39
HGA	0.408	N/A	20

Table 2: Comparing the performance of each algorithm in finding optimal Cayley map embeddings of K_{19}. The optimal genus $\gamma(K_{19}) = 20$.

While the HC algorithm did not run for a user-specified number of iterations, the rest of the algorithms were more customizable regarding the input parameters. To obtain the results in Table 2, we used 500 iterations for the SHC algorithm, 20 individuals and 20 generations for the HGA, and 50 individuals and 50 generations for the GA. Due to the nature of the HGA, this algorithm will tend to run longer than the other algorithms as it covers the search space of possible triplets in the brute force portion and the search space composed of all orientations of each triplet during the GA portion. The program finishes once it has covered all sets of valid triplets, so it is possible that the program finds many optimal embeddings along the way as opposed to the GA which may only find a single global optima in the search space of all permutations of ρ. Although its average execution time was 0.408 seconds, the HGA found at least 16 permutations of λ that formed optimal embeddings. Furthermore, in the implementation of the HGA, we were only focused on finding the number of cycles in ρ generated by each set of triplets, so finding the average genus was irrelevant.

Beyond K_{19}, HC and SHC suffered in performance and struggled to reach even local optima. The GA also struggled to provide near-optimal solutions as the complexity of graphs increased. For larger Cayley map embeddings we focused on finding solutions with the HGA, which continued to perform well and provide more than one optimal embedding for each graph we analyzed. Table 3 presents permutations of λ that produce optimal embeddings for graphs of the form K_{12m+7} using the HGA. For each simulation, we used 20 generations each with 20 individuals to cut down on computational burdens that are a result of processing larger graphs. We list one λ for each graph along with the number of optimal ways one could orient the triplets and still obtain an optimal embedding. Each λ is not the only set of triplets that can be found in the search space of possible triplets for each graph. The HGA program can find all possible sets of triplets using the brute force method as well as most of their optimal orientations using the GA portion of the algorithm, but for the sake of conciseness we only present one example. Note that not every graph is of the form K_p, where p is prime. Although a graph of the form K_p is guaranteed to have an optimal embedding with all 3-gons, we were still able to find optimal embeddings for graphs where $12m + 7$ was not prime.

Results for K_{12m+7}

K_{12m+7}	Permutations of λ	Optimal Orientations	Genus	$\gamma(K_{12m+7})$
K_7	(1,2,4)(3,5,6)	2	1	1
K_{19}	(1,2,16)(3,5,11)(4,7,8)(6,14,18)(9,12,17)(10,13,15)	24	20	20
K_{31}	(1,2,28)(3,4,24)(5,8,18)(6,10,5)(7,11,13)(9,23,30)(12,21,29)(14,22,26)(16,19,27)(17,20,25)	75	63	63
K_{43}	(1, 2, 40)(3, 4, 36)(5, 7, 31)(6, 12, 25)(8, 13, 22)(9, 14, 20)(10, 15, 18)(11, 33, 42)(16, 29, 41)(17, 30, 39)(19, 32, 35)(21, 27, 38)(23, 26, 37)(24, 28, 34)	39	130	130
K_{55}	(52, 2, 1)(48, 4, 3)(5, 6, 44)(38, 10, 7)(8, 12, 35)(9, 17, 29)(11, 18, 26)(23, 19, 13)(14, 20, 21)(15, 41, 54)(51, 43, 16)(22, 39, 49)(24, 33, 53)(25, 40, 45)(27, 36, 47)(50, 32, 28)(30, 34, 46)(42, 37, 31)	31	221	221

K_{67}	(1, 2, 64)(60, 4, 3)(56, 6, 5)(52, 8, 7)(48, 10, 9)(11, 12, 44)(35, 19, 13)(14, 20, 33)(31, 21, 15)(16, 23, 28)(17, 24, 26)(18, 50, 66)(22, 47, 65)(25, 46, 63)(62, 45, 27)(29, 51, 54)(61, 43, 30)(53, 49, 32)(34, 41, 59)(36, 40, 58)(55, 42, 37)(38, 39, 57)	25	336	336
K_{79}	(76, 2, 1)(72, 4, 3)(68, 6, 5)(7, 8, 64)(9, 10, 60)(11, 12, 56)(51, 15, 13)(14, 21, 44)(16, 22, 41)(17, 23, 39)(18, 24, 37)(19, 27, 33)(31, 28, 20)(25, 55, 78)(75, 57, 26)(77, 52, 29)(30, 54, 74)(32, 53, 73)(66, 58, 34)(62, 61, 35)(36, 59, 63)(38, 49, 71)(40, 48, 70)(69, 47, 42)(65, 50, 43)(67, 46, 45)	66	475	475
K_{91}	(1, 2, 88)(3, 4, 84)(80, 6, 5)(76, 8, 7)(9, 10, 72)(11, 12, 68)(64, 14, 13)(15, 16, 60)(17, 23, 51)(48, 25, 18)(19, 26, 46)(43, 28, 20)(41, 29, 21)(22, 30, 39)(35, 32, 24)(90, 65, 27)(89, 62, 31)(33, 63, 86)(87, 61, 34)(36, 67, 79)(37, 70, 75)(85, 59, 38)(40, 69, 73)(42, 57, 83)(44, 56, 82)(71, 66, 45)(47, 54, 81)(49, 55, 78)(50, 58, 74)(77, 53, 52)	47	638	638

Table 3: HGA results for graphs of the form K_{12m+7}.

When working with more complex graphs, we were also able to observe patterns of convergence in the HGA. In Figure 9, we provide an example of a convergence graph for a particular set of triplets that form an embedding of the graph K_{79}. As you can see in the convergence graph, we have plotted the generations on the x-axis and the number of cycles in ρ on the y-axis, and we focus on observing the change in average fitness and best fitness over the course of the genetic algorithm. Clearly, the particular set of triplets that were found by the brute force portion can be oriented in such a way that they form an optimal embedding since the the algorithm found orientations that generated only one cycle in ρ. Even with increasing complexity, the average fitness value within the population of chromosomes dropped very quickly once the algorithm discovered an optimal orientation of triplets soon after the fifth generation. By the 10th generation, most chromosomes in the population were already optimal. Convergence graphs like the one in Figure 9 can ultimately help one understand how fast the algorithm is converging to a particular solution and whether or not one should cut back on the number of generations if convergence is quite fast or perhaps increase the number of generations if convergence is slow.

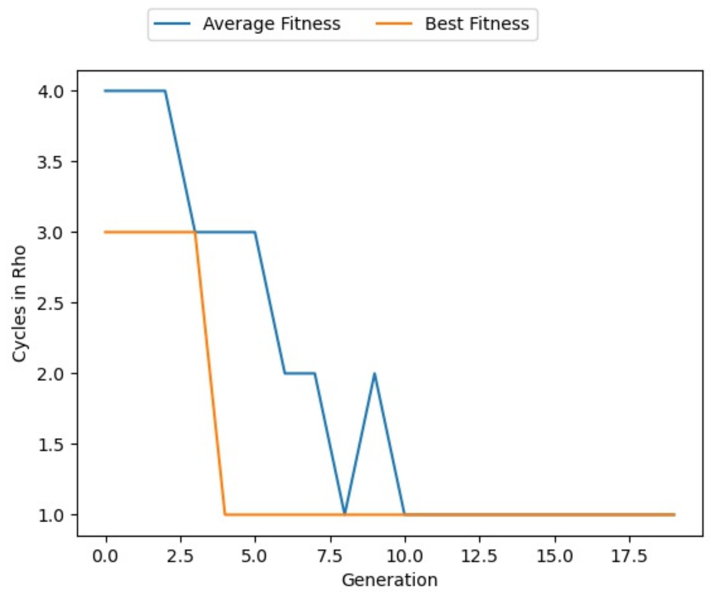

Figure 9: A convergence graph for a λ that produces an optimal embedding of K_{79}.

5 Discussion

For every graph that was simulated, the HGA outperformed every other algorithm in finding the most optimal solutions possible. This was especially the case for larger, more complex graphs beyond K_{19}. For smaller graphs such as K_7, the HGA took longer to find optimal solutions on average compared to other algorithms but these differences were essentially trivial considering how small the search space is for finding optimal solutions in K_7. For optimal embeddings of K_{19}, the HGA also had a longer average run-time compared to HC, SHC, and GA but the optimality of the solutions that it found heavily outweighed the loss of extra time. While the HGA could easily find an optimal solution for K_{19} with genus 20, the HC, SHC, and GA could only settle for near-optimal solutions with the next best possible embedding having a genus of 39. Beyond K_{19}, the HGA continued to perform effectively and without considerable computational burdens. Furthermore, our results provide empirical evidence that there exists several optimal embeddings for any graph of the form K_{12m+7}, further supporting Scheinblums conjecture that there exists an optimal embedding for any graph of that form [11].

A problem that hindered the other search algorithms was the structure of the search space for finding optimal permutations of ρ. The BF algorithm failed to find solutions for graphs beyond K_7, and was ultimately ruled out because of computational burdens. The lexographic search provided no help in handling the large search spaces of more complex graphs. The HC algorithm struggled to find optimal solutions consistently for embeddings of K_7, and that was likely due to the structural patterns that existed in the search space. Based on Figure 8, there were only 2 optimal permutations of ρ in the whole search space, and furthermore each global optima was isolated from other local optima. The structure of the search space did not lend itself well to gradual local search methods such as hill climbing since usually these methods follow gradual descents or ascents from sub-optimal solutions to optimal solutions. However, the local optima were essentially pits in the search space that trapped the hill climbing algorithm, preventing it from being able to transition to a better state. Thus, since there was a lack of paths for the HC to move gradually from a local optima to the global optima, the approach suffered in performance compared to the more advanced methods.

While the HC and BF approaches struggled to deal with higher complexity beyond the graph K_7, the SHC and GA scored similarly on most metrics until they became infeasible for graphs larger than K_{19}. In finding solutions for the embedding of K_{19}, the SHC and GA were in the middle of the pack in terms of performance. While these algorithms had shorter average execution times compared to HGA, they only found near-optimal solutions. For graphs larger than K_{19}, the SHC and GA suffered the same fate as the HC, and could not cover broad search spaces with such few global optima. In past literature, the genetic algorithm has usually out-performed stochastic hill climbing methods because of the crossover operator [54]. However, there may be some reasons as to why the GA did not quickly out-perform the SHC. One possibility is the existence of hitchhikers in the chromosomes of individuals. Hitchiking usually occurs when a large schema in the structure of a chromosome becomes so common in the population, and even though it may provide higher fitness for an individual, it is likely that other elements tag along near the schemas and become hard to replace [55]. This effect may have been a serious bottleneck for the GA in finding optimal solutions since a double-point crossover does not handle hitchikers as well as more advanced crossover methods.

While the simple GA may not have met expectations, it still provided further results that it can easily out-perform a traditional hill climbing method and brute force approaches. Genetic algorithms are not forced to travel down gradients and do not get confused by local optima, unlike traditional hill climbing methods [54]. The GA was able to use stochastic operations in its core operators to its advantage, thus making it much easier for it to cover a broad search space and piece together more optimal schemas in each chromosome. The HC, on the other hand, only depended on adjacent swaps between elements in each chromosome and could not gain an understanding of the underlying structural patterns in ρ that made one optimal. Furthermore, the GA was a population-based approach that could find desirable schemas within chromosomes in parallel with other desirable schemas, while the BF and HC were simply unable to exploit the search space and covered a much shorter range of possible solutions at a time.

However, the GA suffered in performance compared to the HGA, especially for larger graphs. The HGA benefited from all of the inherent strengths of GAs such as the use of coding, ability to search populations, blindness to auxiliary information, and operators that incorporate randomness [6]. All of these strengths clearly propelled the HGA past the BF, HC, and SHC approaches and ultimately contributed to the overall robustness of the algorithm. The addition of domain-specific knowledge and the ease in finding sets of triplets with the brute force component for most graphs also influenced the success of the HGA over the GA. A key distinction that makes the HGA stand out from the other algorithms is that it does not focus its search on finding ρ. When searching for optimal ρ, the GA could not benefit from any domain-specific findings to streamline the search process. On the other hand, our approach with finding λ in the HGA benefited from domain-specific knowledge regarding the formulation of triplets to create the maximum number of faces possible in an embedding of K_{12m+7}, and thus making it optimal. For now, we have inadequate understanding of the patterns in ρ that may increase the likelihood of a particular ρ producing an optimal embedding.

Applications of Graph Embeddings

While many readers may think that graph embeddings are only important for topological graph theory, it is worth taking a moment to consider a specific application of finding optimal Cayley map embeddings. One of the most important and obvious applications may be in the design of printed circuit boards, or PCBs [56]. If one thinks of the connection points of a PCB as the vertices of a graph and the wires as edges, then it is clear that the circuits can be modeled by graphical representations. A common risk that occurs in the construction of PCBs is the possibility of a short circuit, which can cause considerable damage to the circuit. As long as wires do not cross, short circuits can be more easily prevented. Thus, the Cayley map provides a reasonable solution to such a problem since it models the circuit board such that no two edges of the graph cross, and therefore there will be no wire crossings. Furthermore, finding optimal Cayley maps that require the least number of holes as possible on an orientable surface can be useful when cutting down on the number of layers in a circuit board. Since the layers in the circuit board are essentially the holes, optimal Cayley map embeddings can provide an advantage in the design and construction of PCBs.

6 Conclusion

In this paper, we explored new approaches to finding optimal Cayley map embeddings for complete graphs of the form K_{12m+7} where m is a positive integer. Previous attempts at finding optimal embeddings relied primarily on brute force approaches that found optimal permutations of λ, but we presented alternative methods including a brute force approach, hill climbing algorithm, stochastic hill climbing algorithm, and genetic algorithm. While all of these methods focused on finding optimal ρ, we also constructed a hybrid genetic algorithm that could finding optimal embeddings by searching for λ. We compared the algorithms in their ability to find optimal solutions quickly. We also compared how well the algorithms could handle larger graphs and if they could find not only local optima, but more importantly, global optima.

The traditional hill climbing algorithm and the brute force approach failed to exploit the structure of the search space in finding optimal ρ. The brute force algorithm could not handle vast search spaces while the hill climbing algorithm could not escape the confusion of local optima. On the other hand, the random search algorithms such as the stochastic hill climbing algorithm, genetic algorithm, and hybrid genetic algorithm made good use of probabilistic components in their implementations. The stochastic hill climbing algorithm and genetic algorithm both performed similarly, but due to the lack of domain-specific knowledge about what makes a particular ρ optimal, these algorithms could not surpass the performance of the hybrid genetic algorithm. The hybrid genetic algorithm made use of all of the benefits that come along with genetic algorithms as well as additional domain-specific knowledge about the nature of λ that made its approach more successful. Instead of going into the problem blind, the algorithm blended deterministic and probabilistic components in a way that contributed to its overall success.

This work has provided additional empirical evidence that optimal embeddings exist for graphs of the form K_{12m+7}. Furthermore, we have established another case in which genetic algorithms and evolutionary strategies can be useful in solving challenging combinatorial optimization problems.

7 Appendix

GitHub code repo: https://github.com/JacobBuckelew/HonorsThesis

References

[1] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Comput. Surv.*, vol. 35, pp. 268–308, 01 2001.

[2] I. Osman and G. Laporte, "Metaheuristics: A bibliography," *Annals of Operational Research*, vol. 63, pp. 513–628, 10 1996.

[3] S. Bandaru and K. Deb, "Metaheuristic techniques," 2016.

[4] M. Mitchell, *An Introduction to Genetic Algorithms.* Cambridge, MA, USA: MIT Press, 1998.

[5] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* Cambridge, MA, USA: MIT Press, 1992.

[6] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning.* USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed., 1989.

[7] D. Rocke, "Genetic algorithms + data structures = evolution programs by z. michalewicz," *Journal of the American Statistical Association*, vol. 95, pp. 347–348, 03 2000.

[8] J. Scholz, "Genetic algorithms and the traveling salesman problem a historical review," *CoRR*, vol. abs/1901.05737, 2019.

[9] Y. Nagata and S. Kobayashi, "A powerful genetic algorithm using edge assembly crossover for the traveling salesman problem," *INFORMS Journal on Computing*, vol. 25, no. 2, pp. 346–363, 2013.

[10] S. Katoch, S. Chauhan, and V. Kumar, "A review on genetic algorithm: past, present, and future," *Multimedia Tools and Applications*, vol. 80, pp. 8091–8126, 2021.

[11] M. Scheinblum, "Cayley map embeddings of complete graphs," *Honors Program Theses*, no. 151.

[12] C. Jordan, "Sur la déformation des surfaces," *Journ. Math. pures et appl.,* $2^e Série, vol.\ 11, pp.\ 105 - -109, 1866.$

[13] J. Gallier, "The classification theorem for compact surfaces and a detour on fractals," 2008.

[14] G. Ringel and J. W. T. Youngs, "Solution of the heawood map-coloring problem," *Proc. Nat. Acad. Sci. U.S.A.*, no. 60, pp. 438–445, 1968.

[15] J. Nievergelt, R. Gasser, F. Mäser, and C. Wirth, *All the needles in a haystack: Can exhaustive search overcome combinatorial chaos?*, pp. 254–274. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995.

[16] D. H. Lehmer, "The sieve problem for all-purpose computers," *Mathematics of Computation*, vol. 7, no. 41, pp. 6–14, 1953.

[17] E. F. Beckenbach and G. Pólya, *The Machine Tools of Combinatorics.* Wiley NY, 1964.

[18] K. Yan, "A review of the development and applications of number theory," *Journal of Physics: Conference Series*, vol. 1325, p. 012128, oct 2019.

[19] O. Patashnik, "Qubic: 4×4×4 tic-tac-toe," *Mathematics Magazine*, vol. 53, no. 4, pp. 202–216, 1980.

[20] L. Allis, *Searching for solutions in games and artificial intelligence.* PhD thesis, Maastricht University, Jan. 1994.

[21] R. Gasser, *Harnessing Computational Resources for Efficient Exhaustive Search.* PhD thesis, ETH Zurich, 1995.

[22] T. Rokicki, "Twenty-two moves suffice for rubik's cube®," *The Mathematical Intelligencer*, vol. 32, pp. 33–40, 03 2010.

[23] A. Juels and M. Wattenberg, "Stochastic hillclimbing as a baseline method for evaluating genetic algorithms," in *Advances in Neural Information Processing Systems* (D. Touretzky, M. Mozer, and M. Hasselmo, eds.), vol. 8, MIT Press, 1995.

[24] Z. Michalewicz and D. Fogel, *How to Solve It: Modern Heuristics.* 01 2004.

[25] J. H. Dinitz and D. R. Stinson, “A hill-climbing algorithm for the construction of one-factorizations and room squares,” *Siam Journal on Algebraic and Discrete Methods*, vol. 8, pp. 430–438, 1987.

[26] R. Lewis, “A general-purpose hill-climbing method for order independent minimum grouping problems: A case study in graph colouring and bin packing,” *Computers Operations Research*, vol. 36, pp. 2295–2310, 07 2009.

[27] A. Rosete-Suárez, A. Ochoa-rodrguez, and M. Sebag, “Automatic graph drawing and stochastic hill climbing,” vol. 2, 08 1999.

[28] D. H. Ackley, “A connectionist machine for genetic hillclimbing,” 1 1987.

[29] I. LjubiĆ and G. Raidl, “An evolutionary algorithm with stochastic hill-climbing for the edge-biconnectivity augmentation problem,” in *Applications of Evolutionary Computing* (E. J. W. Boers, ed.), (Berlin, Heidelberg), pp. 20–29, Springer Berlin Heidelberg, 2001.

[30] B. Xi, Z. Liu, M. Raghavachari, C. Xia, and L. Zhang, “A smart hill-climbing algorithm for application server configuration,” *Thirteenth International World Wide Web Conference Proceedings, WWW2004*, 04 2004.

[31] M. Fattah, M. Daneshtalab, P. Liljeberg, and J. Plosila, “Smart hill climbing for agile dynamic mapping in many-core systems,” in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2013.

[32] W. Millan and A. Clark, “Smart hill climbing finds better boolean functions,” *Workshop on Selected Areas in Cryptology 1997, Workshop Record*, 09 1997.

[33] S. Chalup and F. Maire, “A study on hill climbing algorithms for neural network training,” in *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, vol. 3, pp. 2014–2021 Vol. 3, 1999.

[34] M. O’Luing, S. Prestwich, and S. A. Tarim, “Combining k-means type algorithms with hill climbing for joint stratification and sample allocation designs,” 2021.

[35] N. Kokash, “An introduction to heuristic algorithms,” 2005.

[36] M. Mitchell, “Genetic algorithms: An overview,” *Complexity*, vol. 1, no. 1, pp. 31–39, 1995.

[37] J. A. Carr, “An introduction to genetic algorithms,” 2014.

[38] L. Eshelman, R. Caruana, and J. Schaffer, “Biases in the crossover landscape.,” pp. 10–19, 01 1989.

[39] W. Spears and K. De Jong, “On the virtues of parametrized uniform crossover,” 01 1991.

[40] D. E. Goldberg and R. Lingle, “Alleleslociand the traveling salesman problem,” in *Proceedings of the 1st International Conference on Genetic Algorithms*, (USA), p. 154–159, L. Erlbaum Associates Inc., 1985.

[41] M. Mitchell, *Complexity: A Guided Tour*. USA: Oxford University Press, Inc., 2009.

[42] J. J. Grefenstette, R. Gopal, B. J. Rosmaita, and D. V. Gucht, “Genetic algorithms for the traveling salesman problem,” in *Proceedings of the 1st International Conference on Genetic Algorithms*, (USA), p. 160–168, L. Erlbaum Associates Inc., 1985.

[43] I. Oliver, D. Smith, and J. R. Holland, “Study of permutation crossover operators on the traveling salesman problem,” in *Genetic algorithms and their applications: proceedings of the second International Conference on Genetic Algorithms: July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, MA*, Hillsdale, NJ: L. Erlhaum Associates, 1987., 1987.

[44] D. Whitley, T. Starkweather, and D. Shaner, “The traveling salesman and sequence scheduling: Quality solutions using genetic edge recombination,” in *In Handbook of Genetic Algorithms*, pp. 350–372, 1990.

[45] A. Homaifar, S. Guan, and G. E. Liepins, “Schema analysis of the traveling salesman problem using genetic algorithms,” *Complex Syst.*, vol. 6, 1992.

[46] B. Freisleben and P. Merz, “A genetic local search algorithm for solving symmetric and asymmetric traveling salesman problems,” in *Proceedings of IEEE International Conference on Evolutionary Computation*, pp. 616–621, 1996.

[47] Y. Nagata, S. Kobayashi, and T. Back, “Edge assembly crossover: A high-power genetic algorithm for the travelling salesman problem, international conference; 7th, genetic algorithms,” in *Proceedings of the International Conference on Genetic Algorithms, Genetic algorithms, International conference; 7th, Genetic algorithms*, (San Francisco, Calif.), pp. 450–457, M. Kaufmann;, 1997.

[48] M. M. Alipour, S. N. Razavi, M. R. Feizi Derakhshi, and M. Balafar, “A hybrid algorithm using a genetic algorithm and multiagent reinforcement learning heuristic to solve the traveling salesman problem,” *Neural Computing and Applications*, vol. 30, 11 2018.

[49] M. Abbasi, M. Rafiee, M. R. Khosravi, A. Jolfaei, V. G. Menon, and J. M. Koushyar, “An efficient parallel genetic algorithm solution for vehicle routing problem in cloud implementation of the intelligent transportation systems,” vol. 9, no. 1, 2020.

[50] S. Subiyanto and T. Narwadi, “An application of traveling salesman problem using the improved genetic algorithm on android google maps,” vol. 1818, 03 2017.

[51] J. Wu, K. Xu, Q. Kong, and W. Jiang, “Application of the hybrid genetic algorithm to combinatorial optimization problems in flow-shop scheduling,” in *2007 International Conference on Mechatronics and Automation*, pp. 1272–1277, 2007.

[52] G. Huang and A. Lim, “A hybrid genetic algorithm for three-index assignment problem,” in *The 2003 Congress on Evolutionary Computation, 2003. CEC ’03.*, vol. 4, pp. 2762–2768 Vol.4, 2003.

[53] A. Misevicius, “Testing of hybrid genetic algorithms for structured quadratic assignment problems,” *Informatica*, vol. 20, pp. 255–272, 2009.

[54] A. Prugel-Bennett, “When a genetic algorithm outperforms hill-climbing,” *Theoretical Computer Science*, vol. 320, pp. 135–153, 06 2004.

[55] M. Mitchell and J. Holland, “When will a genetic algorithm outperform hill-climbing?,” *Santa Fe Institute, Working Papers*, 01 1993.

[56] J. L. Gross and J. Yellen, *Graph Theory and Its Applications, Second Edition (Discrete Mathematics and Its Applications)*. Chapman Hall/CRC, 2005.