

EVOLVING GRAPH NEURAL NETWORK ARCHITECTURES WITH CARTESIAN GENETIC PROGRAMMING FOR THE CAPACITATED VEHICLE ROUTING PROBLEM

Piotr Wierciński, Mariusz Kaleta

Warsaw University of Technology
Institute of Control and Computational Engineering
Nowowiejska 15/19, 00-665 Warsaw, Poland

Deep reinforcement learning combined with Graph Neural Networks (GNNs) has emerged as a promising paradigm for solving the Capacitated Vehicle Routing Problem (CVRP). However, existing approaches predominantly rely on hand-crafted, standard transformer architectures. This study presents an automated evolutionary Neural Architecture Search (NAS) framework utilizing Cartesian Genetic Programming (CGP) to evolve specialized GNN encoder topologies for end-to-end CVRP solvers. The proposed methodology employs a graph-based 2D grid representation, leveraging neutral drift, a $(1 + \lambda)$ evolutionary strategy, partial weight inheritance, and an accelerated proxy-evaluation scheme to efficiently explore the search space. Controlled experiments demonstrate that CGP-NAS significantly outperforms random search under identical proxy budgets. Furthermore, when initialized with standard transformer baselines, CGP-NAS discovers novel non-sequential architectures that match or surpass standard single-layer transformer models across synthetic CVRP benchmarks (CVRP10–100) while achieving up to a 66% parameter reduction. Moreover, the evolved architectures remain competitive with or even outperform the baseline three-layer transformer model. These findings confirm that CGP-NAS effectively navigates GNN search spaces and uncovers scalable, parameter-efficient encoder structures for vehicle routing.

1 Introduction

The Capacitated Vehicle Routing Problem (CVRP) is a foundational combinatorial optimization problem with significant practical applications in logistics, supply chain management, and transportation systems (Dantzig and Ramser, 1959; Vidal et al., 2020). As an NP-hard problem (Lenstra and Rinnooy Kan, 1981), classical exact algorithms become computationally intractable for large-scale instances, necessitating reliance on hand-designed heuristics, metaheuristics, or hybrid heuristics such as Large Neighborhood Search (LNS) (Hottung and Tierney, 2022), Genetic Algorithms (Vidal, 2022), or Lin-Kernighan-Helsgaun (LKH-3) (Helsgaun, 2023). While effective, these hand-crafted heuristics require substantial domain expertise and extensive parameter tuning for specific problem distributions.

In recent years, deep reinforcement learning (RL) combined with Graph Neural Networks (GNNs) has emerged as a promising paradigm for learning heuristic routing solvers. Two main RL-based paradigms have emerged in this domain:

non-end-to-end (or ‘learning-to-improve’) methods, in which the RL agent augments or hybridizes existing algorithms rather than solving the problem directly (e.g., Kaleta and Śliwiński (2025)), and end-to-end approaches, which generate complete solutions in a single pipeline (e.g., Kool et al. (2019)). While exact methods and heuristic approaches solve individual problem instances, Machine Learning (ML) models attempt to learn a policy that generalizes across a distribution of instances. Once trained, such models can produce solutions with relatively low inference cost and execution time. In this framework, instances are modeled as complete graphs, where node feature encoders map spatial location coordinates and demand attributes into rich latent embeddings. An autoregressive decoder then utilizes these embeddings to construct valid vehicle tours sequentially. Models like the attention-based architecture proposed by Kool et al. (2019) have demonstrated strong performance and fast inference times.

Despite these advances, the vast majority of deep learning approaches for vehicle routing rely on off-the-shelf, standard architectures—such as standard transformer layers or multi-layer perceptrons—without adapting the internal structural topology of the encoder to the specific structural properties of routing graphs. Manually designing effective neural architectures remains a time-consuming, empirical process based largely on trial and error. Neural Architecture Search (NAS) offers an automated alternative to systematically explore candidate model topologies (Elsken et al., 2019). However, applying NAS to combinatorial routing domains introduces significant computational challenges due to the high cost of evaluating candidate architectures via RL training loops.

To address these limitations, this paper introduces an automated evolutionary search framework based on Cartesian Genetic Programming (CGP) (Miller and Thomson, 2000) to design tailored GNN encoder computational graphs for CVRP solvers. CGP employs a graph-based representation in the form of a two-dimensional computational grid, making it uniquely suited for representing GNNs as explicit directed computational graphs. This representation allows a direct mapping between neural network structures and evolutionary individuals, enabling flexible exploration of complex architectural designs. The proposed approach uses CGP to evolve the encoder architecture of the attention-based model proposed by Kool et al. (2019). This involves defining the set of available neural network operators and building blocks, designing the CGP representation and search space, and selecting the parameters controlling the evolutionary process. The architecture search is limited to the encoder of the model, which isolates the encoder architecture as the main subject of the search and allows the impact of architectural modifications to be evaluated independently of changes to other components of the model. We incorporate a $(1 + \lambda)$ evolutionary strategy, partial weight inheritance, and an accelerated proxy evaluation scheme to efficiently navigate the architecture space within bounded computational constraints.

The main contributions of this work are summarized as follows:

- We formulate a CGP-based NAS (CGP-NAS) framework for discovering specialized GNN encoder topologies tailored for neural vehicle routing solvers.

- We demonstrate that CGP-NAS navigates the graph search space significantly better than random search given an identical proxy evaluation budget.
- We evaluate evolved architectures across synthetic CVRP benchmarks of varying scales (CVRP10, CVRP20, CVRP50, and CVRP100). The discovered topologies achieve superior solution quality or achieve up to a 66% parameter reduction while matching standard baselines.

The remainder of this paper is structured as follows. Section 2 reviews related work in neural combinatorial optimization, classical VRP heuristics, and hybrid reinforcement learning approaches. Section 3 formalizes the Capacitated Vehicle Routing Problem and details the mathematical formulation of our proposed RL architecture. Section 4 describes the experimental setup, training methodology, and benchmark datasets used for evaluation. Section 5 presents the empirical results, comparing our model’s performance against the baselines in terms of solution quality and computation time. Finally, Section 6 concludes the paper and discusses directions for future research.

2 Related Work

2.1 Deep Learning and Reinforcement Learning for Vehicle Routing

The application of neural networks to routing problems dates back to Hopfield and Tank (1985), who formulated the Traveling Salesperson Problem (TSP) as an energy minimization problem using a recurrent neural network (Hopfield network). Although progress remained limited for decades, the past decade has witnessed accelerated advancements driven by Graph Neural Networks and reinforcement learning. Early foundational work by Vinyals et al. (2015) introduced Pointer Networks to sequence routing outputs using attention mechanisms. Subsequent architectures transitioned toward attention-based encoder-decoder paradigms and introduced reinforcement learning in place of supervised learning (Bello et al., 2016). Nazari et al. (2018) replaced the LSTM encoder used in the Pointer Network with a feed-forward projection, allowing each node to be embedded independently rather than sequentially, which enabled dynamic state information. Kool et al. proposed a transformer-based architecture, employing a multi-head transformer encoder paired with an autoregressive decoder, trained end-to-end via REINFORCE using a deterministic greedy rollout baseline (Kool et al., 2019). This mechanism can be interpreted as message passing over a fully connected graph, where each node aggregates information from all other nodes, weighted by the corresponding attention scores. While effective, these paradigms typically use fixed, hand-crafted encoder architectures. Research focusing on structural optimization of the underlying GNN computational graph for routing remains relatively limited.

2.2 Neural Architecture Search and Evolutionary Algorithms

Neural Architecture Search, popularized by Zoph and Le (2016), encompasses automated techniques designed to replace manual, intuition-driven architecture design with algorithmic optimization. Their approach uses a controller recurrent neural network to generate architectural hyperparameters, from which the target network is synthesized. Over the years, numerous NAS approaches have been proposed, varying primarily in the way the search is performed and candidate architectures are evaluated.

Baker et al. (2016) proposed a reinforcement learning approach based on Q-learning to automatically design convolutional neural networks. Real et al. (2017) demonstrated that a mutation-based evolutionary algorithm, in which the genotype encodes specifics of architecture structure and hyperparameters, can outperform manually designed networks. Zoph et al. (2018) observed that modern neural networks are often composed of repeated computational structures and consequently, they proposed an algorithm that searches only architectures composed of so called cells, showing that architectures found for smaller image recognition problems, such as CIFAR-10, can be transferred to larger-scale image recognition datasets, such as ImageNet. Liu et al. (2019) proposed the DARTS architecture, which relaxes the discrete search space into a continuous one, allowing architectures to be optimized directly using gradient descent. This differentiable formulation significantly reduced the computational cost of neural architecture search while achieving state-of-the-art performance.

Most of the above discussed approaches were designed for NAS in Convolutional Neural Networks (CNNs), which were the primary focus of research at the time. NAS was extended to GNNs by Gao et al. (2019) with GraphNAS, that employs a recurrent controller trained with the REINFORCE algorithm to sequentially generate candidate architectures, which are then evaluated on a downstream graph learning task. Unlike CNN-oriented NAS methods, the search space is specifically designed for graph neural networks and includes graph-specific architectural choices such as aggregation functions, attention mechanisms, hidden dimensionality, and the number of message-passing layers. GraphNAS demonstrated that NAS can be successfully applied to GNNs.

2.3 Cartesian Genetic Programming in Architectural Search

Cartesian Genetic Programming is a variant of genetic programming introduced by Miller and Thomson (2000). It represents a computational structure as a directed acyclic graph arranged on a two-dimensional grid, hence the name Cartesian. CGP is a general-purpose representation that can be applied to a wide range of problems, including image filtering, digital circuit design, game-playing agents, and neural architecture search.

Early CGP-based NAS implementations primarily relied on discrete encodings and mutation-only strategies, employing a $(1 + \lambda)$ evolutionary strategy on directed acyclic graphs to design Convolutional Neural Networks for im-

age classification benchmarks like CIFAR-10 (Suganuma et al., 2017). Subsequent enhancements expanded this scope by incorporating broader functional operation sets, initializing populations with seeded topologies such as modified ResNets and DenseNets, and implementing early candidate termination to substantially reduce evaluation costs across CIFAR-10 and CIFAR-100 benchmarks (Suganuma et al., 2020). Demonstrating its domain-agnostic adaptability, CGP-based NAS was later successfully applied to Natural Language Processing tasks such as sentence classification (Wu et al., 2021).

More recent developments have modernized CGP-based NAS by overcoming traditional limitations; for instance, introducing Multiple Sequence Alignment (MSA)-based crossover operators allows offspring to retain beneficial structural motifs without destroying phenotype integrity (Torabi et al., 2022), while Continuous Cartesian Genetic Programming (CCGP) transitions to continuous encodings to enable multi-objective optimization that simultaneously balances classification performance alongside real-world efficiency constraints such as parameter size, inference latency, and power consumption (Garcia-Garcia et al., 2023).

2.4 Research gaps and questions

Neural Architecture Search has been successfully applied to GNNs, while CGP has also been investigated as a representation and search mechanism for neural architecture optimization. At the same time, learning-based approaches have become increasingly prominent for solving VRP. However, these research directions have largely developed independently. To the best of our knowledge, CGP has not yet been explored as a NAS framework to evolve the encoder architectures for attention-based routing models, such as proposed by Kool et al. (2019). Our work addresses this gap by applying CGP-based architecture search to the model’s encoder and evaluating whether automatically discovered architectures can improve routing performance over the manually designed baseline.

The central hypothesis of this work is that CGP-based NAS can effectively guide the exploration of the architectural search space, outperform random search, and discover effective modifications of the original transformer-based encoder. We formulate the following two specific research questions:

- RQ1 Can CGP-based NAS consistently discover better encoder architectures than random search under the same search space and computational budget?
- RQ2 Can CGP-based NAS evolve the original transformer-based encoder to improve its routing performance?

While RQ1 evaluates whether the evolutionary search process can effectively exploit information gathered during evolution rather than behaving similarly to random exploration of the architecture space, RQ2 evaluates whether CGP-NAS can discover effective modifications of an existing, manually designed architecture. The original transformer-based encoder serves as the starting point of

the evolutionary search, and the resulting architectures are evaluated across different CVRP problem sizes.

3 Proposed Methodology

3.1 Underlying Problem Statement

The underlying Capacitated Vehicle Routing Problem is defined on a complete directed graph $G = (V, E)$, where $C = \{1, \dots, n\}$ represents the set of customers and $V = \{0\} \cup C$ represents the set of all vertices with 0 designating the central depot. Each ordered pair $(i, j) \in V \times V$ with $i \neq j$ has an associated travel cost $c_{ij} > 0$, and each customer $i \in C$ has a known demand $d_i > 0$ with $d_0 = 0$. A fleet of vehicles $K = \{1, \dots, |K|\}$, each with maximum capacity Q satisfying $Q \geq \max_{i \in C} d_i$, is stationed at the depot to serve customer demands.

The model utilizes binary decision variables $x_{ijk} \in \{0, 1\}$ for all $i, j \in V$ with $i \neq j$ and $k \in K$, where $x_{ijk} = 1$ if vehicle k travels directly from vertex i to vertex j , and 0 otherwise. Additionally, continuous auxiliary variables $u_i \geq 0$ are introduced for each customer $i \in C$ to enforce capacity constraints sequentially and eliminate subtours.

The mathematical formulation of the CVRP is stated as follows:

$$\text{minimize} \quad \sum_{k \in K} \sum_{i \in V} \sum_{j \in V, j \neq i} c_{ij} x_{ijk} \quad (1)$$

$$\text{subject to} \quad \sum_{j \in C} x_{0jk} \leq 1 \quad \forall k \in K \quad (2)$$

$$\sum_{k \in K} \sum_{j \in V, j \neq i} x_{ijk} = 1 \quad \forall i \in C \quad (3)$$

$$\sum_{j \in V, j \neq i} x_{ijk} - \sum_{j \in V, j \neq i} x_{jik} = 0 \quad \forall i \in V, \forall k \in K \quad (4)$$

$$\sum_{i \in C} d_i \sum_{j \in V} x_{ijk} \leq Q \quad \forall k \in K \quad (5)$$

$$u_j - u_i \geq d_j - Q(1 - x_{ijk}) \quad \forall i, j \in C, i \neq j, \forall k \in K \quad (6)$$

$$d_i \leq u_i \leq Q \quad \forall i \in C, \forall k \in K \quad (7)$$

$$x_{ijk} \in \{0, 1\} \quad \forall i, j \in V, i \neq j, \forall k \in K \quad (8)$$

$$u_i \geq 0 \quad \forall i \in C \quad (9)$$

Objective function (1) minimizes the total routing costs across all vehicles. Constraints (2) restrict each vehicle to at most one departure from the depot. Constraints (3) ensure that each customer is visited exactly once by one vehicle. Constraints (4) guarantee flow conservation for each vehicle at every vertex. Maximum vehicle capacity is not exceeded due to constraints (5). Constraints (6)-(7) enforce Miller-Tucker-Zemlin subtour elimination. Finally, binary requirements on arc traversals per vehicle and non-negativity are specified in (8)-(9).

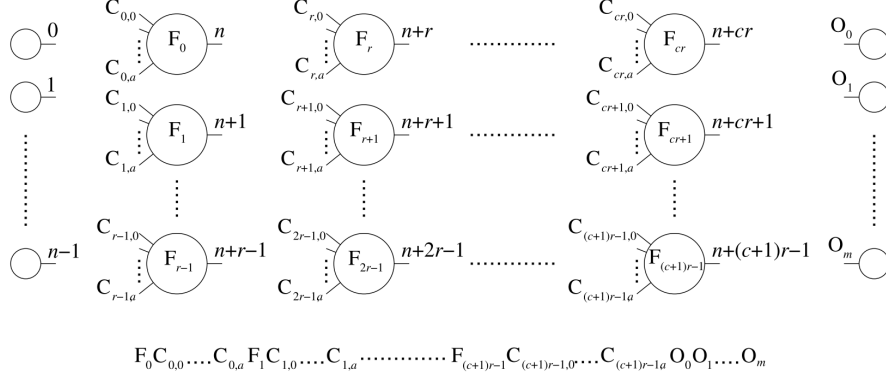


Figure 1: General form of CGP; Nodes $0, \dots, n-1$ represent inputs while nodes $O_0, \dots, O_m$ represent outputs (source: Miller (2011)).

3.2 Cartesian Genetic Programming

Cartesian Genetic Programming represents program structures and computational graphs as directed acyclic graphs indexed within a 2D grid (Miller and Thomson, 2000; Miller, 2011). The genotype has a fixed length determined by the number of rows and columns in the grid. Each node in the grid represents a primitive computational unit, such as a mathematical function, a logical operator, or a neural network component.

Each node corresponds to a gene defined by its function type and the indices of its input connections. The position of a gene in the genotype determines the identifier of the corresponding node in the graph. Connections are restricted according to the position of nodes in the grid. A node can receive inputs only from nodes located in preceding columns. The gene for a node i is represented as a tuple

$$(F_i, C_{0,0}, \dots, C_{0,\alpha}) \quad (10)$$

in which F_i denotes the operation performed by the node, and $C_{0,0}, \dots, C_{0,\alpha}$ is a list of indices of the node's inputs. The genotype architecture is illustrated in Figure 1.

A defining feature of CGP is the decoupling of the genotype from the phenotype. Because of the graph structure, only a subset of nodes takes part in the computation. This creates room for structural redundancy and neutral drift. Neutral drift is a key component of CGP and occurs when a mutation affects an inactive gene, resulting in the offspring having the same fitness as its parent. In this case, the offspring is always preferred, which allows the search process to better explore the search space.

3.3 Overall Framework Architecture

Our architecture search framework builds upon the attention-based encoder-decoder model introduced by Kool et al. (2019). To isolate the effect of architecture search, the evolutionary process is restricted to the encoder. The decoder, reinforcement learning framework, and other components of the baseline model remain unchanged. As a result, differences in routing performance can be attributed primarily to changes in the encoder architecture rather than to modifications of the overall solution framework.

The overall CGP-based architecture search procedure is summarized in Algorithm 1. The evolutionary loop begins by initializing and evaluating a single parent architecture. Until the evaluation budget is depleted, offspring are produced via mutation; if an offspring’s expressed phenotype matches its parent, it instantly inherits the parent’s fitness score without consuming budget, leveraging CGP’s neutral drift to explore the search space efficiently. Otherwise, the child is trained and evaluated at the cost of one budget unit. Any offspring matching or improving upon the parent’s score becomes the new parent for subsequent generations.

Algorithm 1 General concept of proposed CGP-NAS method

```

1: parent  $\leftarrow$  INITIALIZE_PARENT()
2: parentScore  $\leftarrow$  EVALUATE(parent)
3: while budgetRemaining > 0 do
4:   children  $\leftarrow$  PRODUCE_OFFSPRING(parent)
5:   for each child in children do
6:     if PHENOTYPES_MATCH(child, parent) then
7:       childScore  $\leftarrow$  parentScore
8:     else
9:       childScore  $\leftarrow$  EVALUATE(child)
10:      budgetRemaining  $\leftarrow$  budgetRemaining − 1
11:    end if
12:    if childScore  $\leq$  parentScore then
13:      parent  $\leftarrow$  child
14:      parentScore  $\leftarrow$  childScore
15:    end if
16:  end for
17: end while
18: return parent

```

3.4 CGP Representation for GNN Encoders

A genome representing the encoder architecture consists of a sequence of genes that encode a directed acyclic computational graph whose nodes are arranged on a two-dimensional rectangular grid. Each gene describes a single computational block and therefore corresponds to one node of the computational graph. A

gene is represented as a tuple of the following form:

$$\text{GENE} := (\text{TYPE}, [\text{INPUTS}], \text{ARGUMENT?}) \quad (11)$$

where **TYPE** is the type of the computational block, **INPUTS** is the list of identifiers specifying the inputs to the block, and **ARGUMENT** is an operation-specific parameter if required by the selected block type.

The functional block primitives were selected to represent standard neural operations while providing the evolutionary process sufficient flexibility to modify feature transformations, attention, nonlinearities, and the connectivity between different computational paths. The computational block types utilized in the architecture are summarized in Table 1.

Table 1: Overview of Computational Block Types

ID	Block Type	Description
1	Identity	Returns the input unchanged.
2	Normalization	Applies batch normalization to the input embeddings.
3	Linear Scaling	Applies a learnable linear transformation whose output dimension is determined by the scaling argument.
4	Multi-Head Attention	Performs multi-head self-attention over the input representations.
5	Add	Projects all input tensors to a common embedding dimension if needed, and returns their element-wise sum. This is the only block that can accept more than one input.
6	GELU	Applies the Gaussian Error Linear Unit activation function.
7	ReLU	Applies the Rectified Linear Unit activation function.

The **Linear Scaling** block is the only computational block that accepts an explicit argument. Depending on the value of the argument, the embedding dimension can be increased by a factor of four (up to a maximum of 4096), reduced by a factor of four (down to a minimum of 8), or left unchanged.

The encoder computational graph is encoded with a grid consisting of N rows and M columns of functional nodes indexed sequentially. The genome contains $N \times M + 1$ genes, since the additional gene represents the output of the entire network and is always of type **Add**, allowing it to aggregate one or more outputs produced by the last column of computational blocks. To prevent feedback loops and limit the connectivity of the computational graph, every node may receive inputs only from nodes in the immediately preceding column.

The input embedding initially has a fixed dimensionality. However, the **Linear Scaling** block may increase or decrease the embedding dimension along

individual branches of the computational graph. Consequently, tensors arriving at an **Add** block may have different dimensionalities. To ensure compatibility, the **Add** block first projects each input back to the target embedding dimension of the model using a learnable linear transformation whenever necessary, and only then computes their element-wise sum.

The genotype contains all genes defined by the grid, however only a subset of them may contribute to the network output. During decoding, traversal starts from the output gene and recursively follows all referenced input connections. Only visited genes are considered active and are instantiated as computational blocks. Genes that are not reachable from the output gene are considered inactive and do not contribute to the resulting neural network. As a result, different genotypes may represent the same phenotype if their differences occur only in inactive genes. This property plays an important role in the evolutionary search, as mutations affecting inactive genes do not modify the expressed neural network and therefore constitute neutral mutations.

Let us consider two illustrative examples of the genome representation and related phenotype. First, consider the following genome of length 5, representing a 2×2 grid:

$$(2, [0]), (7, [1]), (5, [0]), (4, [3]), (5, [2, 4]). \quad (12)$$

Figure 2a illustrates the corresponding computational graph, where node 0 represents the network input. For instance, $(2, [0])$ means that block of type 2 (**Normalization**) is the first block in the first row and is connected to the input. It is followed by $(7, [1])$, that is, **ReLU** directly connected to the node 1.

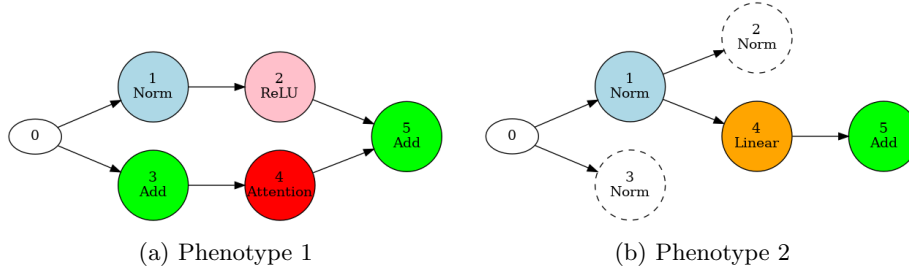


Figure 2: Examples of two phenotype graph structures.

The second genome also represents a 2×2 grid:

$$(2, [0]), (2, [1]), (2, [0]), (3, [1, 1]), (5, [4]). \quad (13)$$

The decoded computational graph is presented in Figure 2b. In this case, the **Normalization** nodes 2 and 3 become inactive nodes and are omitted in the resulting architecture.

3.5 Evolutionary Search and Partial Weight Inheritance

Once candidate architectures are represented as CGP genomes, the resulting search space is explored through an evolutionary process. Starting from a sin-

gle parent architecture, new candidates are generated through mutation and evaluated according to their fitness. Based on these evaluations, the search progressively explores different regions of the architecture space while retaining promising solutions.

We employ a $(1 + \lambda)$ evolutionary strategy ($\lambda = 4$) commonly used in CGP, including in the original formulation by Miller and Thomson (2000). The evolutionary search starts from a single initial parent genome, which can be either generated randomly or constructed from a predefined neural network architecture.

In each generation, λ offspring are generated from a single parent through point mutations. Each offspring is generated as a copy of the current parent and subsequently modified through a sequence of mutations. The number of mutations is determined by the proportion of the remaining computational budget to the total budget and follows an exponential decay schedule. As the search progresses and the remaining budget decreases, the number of mutations applied to each offspring also decreases. Specifically, the number of mutated genes is computed as:

$$m = \max \left\{ 1, \lfloor 0.5 \cdot (|G| - 1) \cdot e^{-k(1-r)} \rfloor \right\} \quad (14)$$

where $|G|$ denotes the genome length, k is the decay coefficient, and

$$r = B_{remaining} / B_{total} \quad (15)$$

is the fraction of remaining computational budget.

Each selected gene undergoes either a type mutation or an input mutation. The probability of selecting an input mutation depends on the number of rows R in the grid and is given by:

$$P(\text{input mutation}) = \frac{R}{R + 7}. \quad (16)$$

The probability of selecting a type mutation is defined as follows:

$$P(\text{type mutation}) = \frac{7}{R + 7}. \quad (17)$$

If the newly selected block requires operation-specific parameters, such as the scaling factor of the **Linear Scaling** block, these parameters are initialized randomly.

When mutating input connections, one input is always selected, while each additional input (in case of **Add** block) is included with probability 0.5. Input connections are sampled without replacement from the set of valid predecessors.

After all offspring have been evaluated, the best candidate is compared with the current parent. If its fitness is better than or equal to that of the parent, it replaces the parent in the next iteration. Otherwise, the parent is retained.

The evolutionary search requires a fitness score to compare candidate architectures and select the parent for subsequent generations. Following Kool et al. (2019), the architecture score is calculated as the mean tour length over

the corresponding evaluation dataset, with lower values indicating better performance. Greedy decoding is used to construct the routes during both proxy and full evaluation. Evaluating individual candidates via full RL training from scratch is computationally costly. To reduce this overhead, mutations restricted to inactive genes are identified prior to training. Since such mutations do not alter the expressed neural network, the offspring has the same phenotype as its parent. Therefore, it inherits the parent’s fitness score without requiring training or evaluation. As a result, the computational budget is consumed only when an offspring requires an actual neural network training run.

To further reduce the computational cost of training offspring architectures, partial weight inheritance is used, which is a well-known technique in NAS (Liu et al., 2021). Before training an offspring, parameters are transferred from the parent for computational blocks that occupy the same position in the genome and have the same neural network module type. As a result, parameters that remain compatible between the parent and offspring can be reused, while newly introduced or dimensionally incompatible parameters are trained from scratch.

Finally, following common approaches in NAS, a proxy evaluation procedure is used to estimate the performance of candidate architectures (Elsken et al., 2019). The computational cost is reduced primarily by limiting the training of each candidate to fewer epochs and fewer training instances per epoch. It is worth noting that the term evaluation refers here to training a network for a specified number of epochs and calculating its score after training. Accordingly, two types of evaluation are used in this paper: 1) Proxy evaluation used during the architecture search to quickly estimate relative performance with only 10 epochs, and epoch size equal to 12,800, and 2) Full evaluation used for the final evaluation of selected architectures with 100 epochs, each of size 1,280,000.

4 Experimental Setup

4.1 CGP-NAS hyperparameters

The size of the CGP grid is selected separately for each experiment and is therefore specified in the corresponding experiment sections.

The decay coefficient k in the mutation schedule defined in equation (14) is fixed to $k = 3$ across all experiments. This value was chosen to provide a balance between larger mutations during the early stages of the search, encouraging exploration, and smaller mutations towards the end of the search, promoting local exploitation.

The offspring population size is set to $\lambda = 4$, yielding a $(1 + 4)$ evolutionary strategy wherein four offspring are generated per parent in each generation. The $(1 + 4)$ strategy is commonly used in CGP and follows the configuration used by Miller and Thomson (2000).

4.2 Reinforcement Learning Hyperparameters

All candidate architectures are trained using the REINFORCE algorithm with a deterministic greedy rollout baseline, following the hyperparameter choices established by Kool et al. (2019). Models are optimized using the Adam optimizer with a fixed learning rate of $\eta = 10^{-4}$. Training batches consist of 512 synthetic CVRP instances generated online per step. The common model and training parameters used across all experiments are summarized in Table 2:

Table 2: Common training and model configuration.

Parameter	Value
Batch size	512
Embedding dimension	128
Hidden dimension	128
Attention heads	8
Tanh clipping	10
Gradient clipping	1.0
Baseline method	Rollout
Optimizer	Adam
Decoding strategy	Greedy
Learning rate	10^{-4}

4.3 Problem Benchmarks

The benchmark instances for the Capacitated Vehicle Routing Problem are generated synthetically using the procedure proposed by Nazari et al. (2018), and later adopted by Kool et al. (2019). Experiments span four problem scales denoted with **CVRP** N , where N is equal to 10, 20, 50, or 100 customer nodes, and vehicle capacity Q is 20, 30, 40, or 50, respectively. The depot location and customer coordinates are sampled independently from a uniform distribution over the unit square:

$$(x_i, y_i) \sim U(0, 1) \quad (18)$$

Customer demands are sampled uniformly from the integers 1 to 9 and normalized by the vehicle capacity:

$$d_i \sim U(1, \dots, 9) \quad (19)$$

$$\hat{d}_i = d_i / C, \quad C \in \{20, 30, 40, 50\} \quad (20)$$

where C denotes the vehicle capacity and is set to 20, 30, 40, and 50 for CVRP10, CVRP20, CVRP50, and CVRP100, respectively.

Three distinct dataset types are utilized: (i) Training datasets, generated synthetically on the fly per epoch; (ii) Validation datasets, consisting of 10,000 offline-generated instances used for monitoring at epoch boundaries; (iii) Test

datasets, comprising 10,000 fixed offline instances used exclusively for final evaluation.

4.4 Evolutionary Search Experiments

To address the research questions, two primary experimental protocols are established:

1. **Experiment I (CGP-NAS vs. Random Search):** Designed to answer RQ1, evaluating whether CGP systematically outperforms unbiased random search. Both methods are executed across 10 independent search runs on CVRP10 instances with a 15×5 CGP grid layout, each starting with randomly generated initial parent. Search budget is capped at 200 proxy evaluations per run. Each proxy evaluation trains the candidate encoder for 10 epochs on 10,000 synthetic CVRP10 instances. The top-performing candidate topologies discovered by each search strategy are subsequently fully retrained from scratch for 100 epochs.
2. **Experiment II (Evolution of Transformer Encoders):** Designed to answer RQ2, investigating whether CGP can enhance an established single-layer transformer encoder baseline. The CGP grid is sized to 8×5 , and the initial parent genotype is explicitly seeded to represent a standard single-layer transformer encoder (see Figure 3). Search is conducted over 10 independent evolutionary runs under the $(1 + \lambda)$ strategy. High-performing candidate graphs are extracted and fully trained across CVRP10, CVRP20, CVRP50, and CVRP100 instances to evaluate performance and parameter efficiency against standard transformer baselines.

The CGP-based NAS method was implemented in Python using PyTorch and executed on CUDA-enabled NVIDIA GPUs. The implementation of the attention-based routing model is based on the implementation of Kool et al. (2019), and was extended with the CGP representation, evolutionary search procedure, genotype-to-phenotype decoding, mutation operators, and partial weight inheritance described in the previous Section. The complete implementation, including experiment configurations as well as validation and test datasets, is available in the source code repository <https://github.com/laminowany/cgp-nas-cvrp>.

5 Results and Discussion

5.1 Comparison of CGP-NAS vs. Random Search

While in CGP-based NAS, new candidates are generated by mutating the current parent according to the evolutionary procedure, in random search, each candidate architecture is generated independently by randomly sampling a new genome from the search space. Therefore, for a given computational budget of

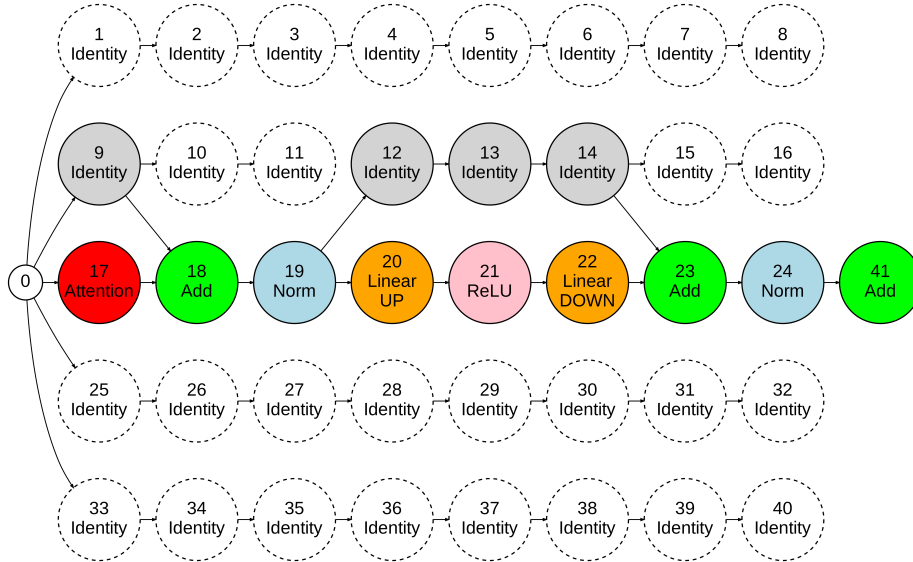


Figure 3: Single-layer transformer in CGP representation.

N evaluations, N architectures are randomly generated and the best-performing one is selected.

To evaluate whether CGP efficiently navigates the space of GNN computational graphs compared to unbiased random sampling (RQ1), we analyze 10 independent runs for each setting under an identical evaluation budget of 200 proxy evaluations per run. That yields ten evolved architectures: CGP-1 to CGP-10, and ten randomly search architectures: RND-1 to RND-10.

As shown in Table 3, CGP-NAS consistently converges to architectures achieving significantly lower average tour lengths than random search. The best score achieved by CGP-based NAS (4.8492) also clearly outperforms the best score found by random search (4.9352). Regarding performance stability, the proposed CGP-NAS method exhibits lower variability ($\sigma = 0.009$) compared to random search ($\sigma = 0.015$).

Table 3: Performance Comparison of CGP-NAS and Random Search (CVRP10) with proxy evaluation

Search Strategy	Proxy Tour Length		
	Mean	Best	Worst
Random Search	4.9536 ± 0.015	4.9352	4.9825
CGP-NAS (Ours)	4.8695 ± 0.009	4.8492	4.8810

Figure 4a visualizes the distribution of architectures that clearly shows no overlap between them. Figure 4b shows the best-so-far proxy score for each

strategy as the evaluation budget increases. Random search shows relatively little improvement after approximately 60 evaluations, while CGP-based NAS continues to improve throughout most of the remaining evaluation budget, which indicates the ability of CGP-NAS approach to search the space more effectively.

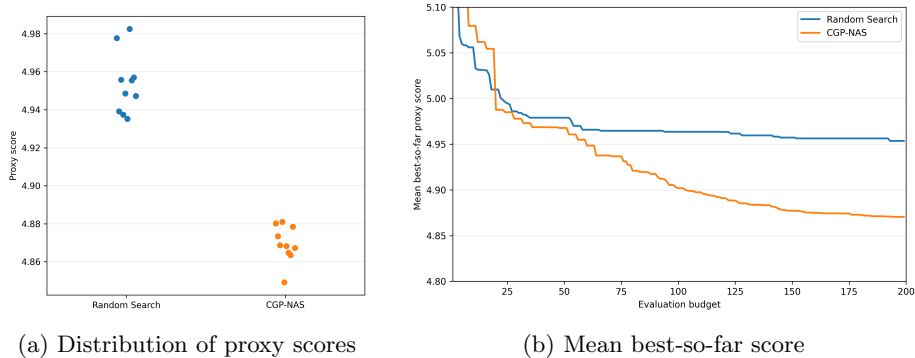


Figure 4: Random search vs CGP-NAS at proxy evaluation.

The three best-performing architectures discovered by CGP (CGP-5, CGP-7, and CGP-8) and the three best-performing architectures found via random search (RND-2, RND-4, RND-6) were selected for full evaluation on the CVRP10 and CVRP20 datasets. Table 4 presents the performance of these architectures when trained from scratch.

Table 4: Experiment I: Full evaluation performance comparison of CGP and random search (RND) architectures.

Architecture	# parameters	CVRP10	CVRP20
RND-2	11,305,376	10.4263	20.8635
RND-4	2,839,808	4.6629	6.4382
RND-6	816,288	4.7136	6.4890
CGP-5	215,072	4.6725	6.4359
CGP-7	156,448	4.6670	6.4340
CGP-8	437,024	4.6858	6.4533

The CGP-based architectures demonstrate consistent performance with low variance across runs (mean score: 4.6751 ± 0.008), whereas random search exhibits significant instability. Surprisingly, the best individual solution for CVRP10 was achieved by RND-4. In contrast, RND-2 performed the worst, displaying severe convergence issues during full training. RND-2 features an excessively large parameter count and an overly complex topology (see Fig. 5), which likely hindered optimization during full training. Notably, all CGP-discovered architectures are significantly smaller in parameter size; among them, the largest model (CGP-8) achieved the worst score within the CGP group.

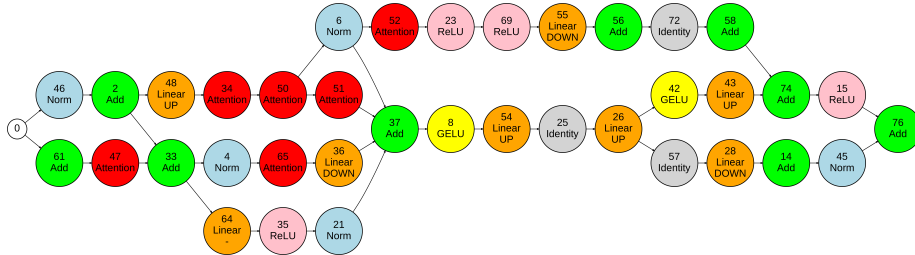


Figure 5: Architecture topology of RND-2.

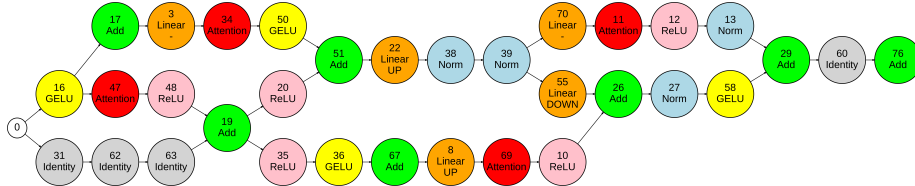


Figure 6: The best performing randomly searched architecture (RND-4).

These results confirm that our evolutionary approach using proxy evaluations outperforms the random search under full evaluation, with performance transferring effectively to larger CVRP20 instances. The surprisingly strong performance of RND-4 on CVRP10 highlights that a 10-epoch proxy evaluation may not always be a perfectly stable or precise predictor of full training performance. Detailed analysis of the evaluation trajectories reveals that while most architectures follow similar convergence paths, RND-4 progresses relatively slowly during early epochs before accelerating around epoch 40 to match and ultimately surpass the best CGP-based architectures. While extended proxy budgets could improve fidelity, determining the optimal proxy duration a priori remains non-trivial. Thus, the 10-epoch proxy serves as a practical trade-off between computational budget and predictive accuracy, providing a reliable guidance signal rather than an exact performance guarantee. As shown in the next section, this 10-epoch proxy scheme is effective enough to improve upon the transformer-based baselines.

Figures 6 and 7 illustrate the best-performing architectures discovered by random search and CGP evolution, respectively. It is notable that RND-4 (29 active computational nodes, excluding **Identity** nodes), is substantially more complex than CGP-5 (18 active nodes). This suggests that the strong performance of RND-4 stems primarily from higher representational capacity via parametr over-parametrization rather than structural efficiency. Although RND-4 did not show an advantage during the proxy evaluation stage, its capacity enabled high performance under extended training. However, random generation remains computationally expensive and unreliable on average, rendering RND-4 an outlier rather than a systematic success.

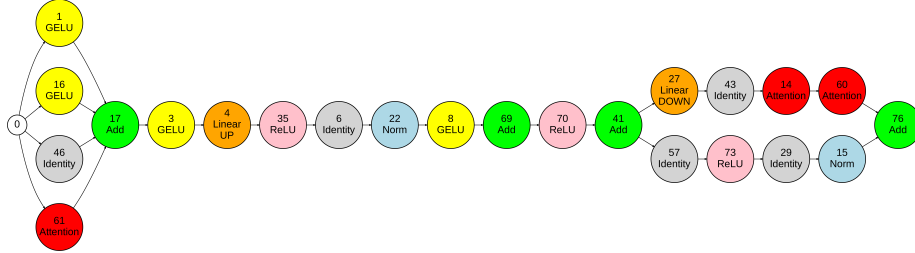


Figure 7: The best performing CGP-based architecture (CGP-5).

5.2 Evolution of Transformer Encoders

In this experiment the CGP was initialized with a single-layer transformer encoder genotype to examine whether structural mutations could discover superior topologies (RQ2). In each evolutionary run, the initial parent has the same architecture of a transformer, but may differ in trained parameters and therefore it has different initial proxy score. Table 5 presents results of proxy evaluations of 10 runs. The proxy scores of the initial transformer vary between 4.9589 and 4.9949.

The CGP-based NAS method manages to improve the initial transformer in all 10 runs. The best final proxy score is achieved by EVO-6 with 4.8609, followed by EVO-3 with 4.8679 and EVO-4 with 4.8693. The top-performing initial transformer baseline achieved a proxy score inferior to even the lowest-performing evolved architecture. Following this initial results, three architectures: EVO-6, EVO-3, and EVO-4 were selected for the full evaluation.

Table 5: Proxy evaluation results of CGP-based NAS runs initialized with a single-layer transformer.

	Proxy Tour Length		
	Mean	Best	Worst
Initial transformer	4.9785 ± 0.011	4.9589	4.9949
CGP-NAS	4.8771 ± 0.009	4.8609	4.8896

The three selected architectures have been further evolved with a full evaluations on CVRP10 instances, and then, the resulting architectures have been tested against larger datasets. The results of full evaluation are collected in Table 6. They show that the evolved architectures remain competitive with the single-layer transformer across the larger problem sizes. EVO-3 achieves the best score on CVRP20 and CVRP50, with scores of 6.4628 and 11.0324, respectively. On CVRP100, the best score is achieved by EVO-6 with 16.7204, closely followed by EVO-3 with 16.7213. EVO-3 is particularly interesting, as its encoder contains 197,504 trainable parameters, almost the same number as the original transformer with 197,760 parameters, while achieving a better score

on all problem sizes.

The evolved architectures also differ considerably in their parameter counts. EVO-4 contains only 67,072 encoder parameters, approximately 66% fewer than the transformer, while remaining relatively competitive across all evaluated problem sizes. In contrast, EVO-6 is the largest architecture with 280,064 parameters and achieves the best result on CVRP100. These results show that the evolutionary process discovers architectures with various levels of complexity rather than simply increasing the size of the initial transformer.

Overall, the results suggest that the architectural modifications discovered during the search on CVRP10 can transfer to larger problem sizes. In particular, EVO-3 consistently outperforms the transformer on CVRP20, CVRP50, and CVRP100 without increasing the number of encoder parameters. However, the differences in routing scores remain relatively small, and the evaluation is based on a single full training run for each architecture and problem size. Therefore, the results should not be interpreted as conclusive evidence that the evolved architectures generally outperform the transformer.

Table 6: Full evaluation of the three best evolved architectures and the single-layer transformer on CVRP10, CVRP20, CVRP50, CVRP100.

Architecture ID	Encoder Params	CVRP10 Score	CVRP20 Score	CVRP50 Score	CVRP100 Score
Transformer	197,760	4.6894	6.4633	11.0679	16.7780
EVO-3	197,504	4.6824	6.4628	11.0324	16.7213
EVO-4	67,072	4.6985	6.4762	11.1093	16.8082
EVO-6	280,064	4.6874	6.4670	11.0613	16.7204

We also compared our evolved architectures on publicly available datasets used by Kool et al. (2019). We compared them against single-layer transformer, but also against the original implementation of the three-layered transformer proposed by Kool et al. and our reproduced implementation. Result are presented in Table 7.

On CVRP20 and CVRP50, the original three-layer model achieves the best results, both in the values reported by Kool et al. (2019) and in our reproduction. However, on CVRP100, EVO-3 and EVO-6 achieve the best score of 16.77, outperforming both the reported score of 16.80 and the reproduced score of 16.91. It is also worth noting that EVO-3 contains less than one third of the encoder parameters of the original three-layer transformer, while achieving a slightly better score on CVRP100. However, the difference on CVRP100 is very small and should be interpreted with caution. The reported result of Kool et al. (2019) and the results obtained in this work come from separately trained models, and the training process is stochastic. Therefore, these results do not provide sufficient evidence to conclude that EVO-3 or EVO-6 generally outperform the original model. Nevertheless, achieving comparable or slightly better performance on CVRP100 is particularly interesting given that the evolved architec-

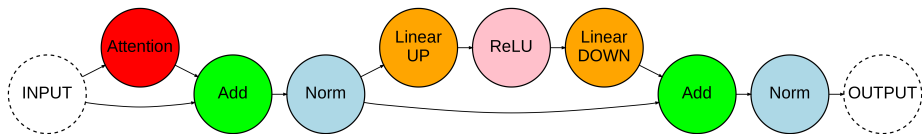


Figure 8: Architecture of single-layer transformer.

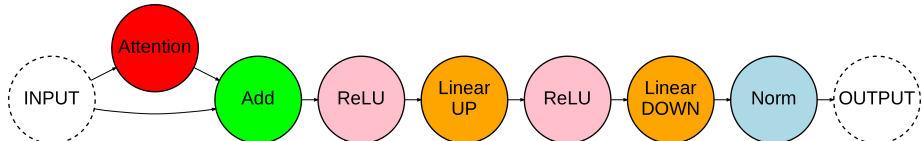


Figure 9: Architecture of EVO-3.

tures originate from a single-layer transformer and were discovered through an evolutionary search conducted only on CVRP10.

Table 7: Comparison of the selected architectures with the reported and reproduced results of Kool et al. on their test datasets.

	CVRP20 Score	CVRP50 Score	CVRP100 Score
Kool et al. (reported)	6.40	10.98	16.80
Kool et al. (reproduced)	6.41	11.00	16.91
Single-layer transformer	6.47	11.09	16.82
EVO-3	6.46	11.05	16.77
EVO-4	6.48	11.13	16.85
EVO-6	6.48	11.08	16.77

5.3 Architectural Analysis of Evolved Architectures

Beyond numerical performance, analyzing the structural modifications introduced by the evolutionary process yields valuable architectural insights. Simplified phenotypes (inactive and **Identity** nodes excluded) of the single-layer transformer and three architectures selected for full evaluation, EVO-3, EVO-4, and EVO-6 are presented in Figures 8-11.

The single-layer transformer consists of two main blocks. The first is a multi-head attention block with a residual connection. The second is a feed-forward block consisting of a **Linear** transformation that scales the embedding dimension up, followed by a **ReLU** activation and another linear transformation that scales the dimension back down. This block also includes a residual connection. Both blocks are followed by **Normalization**. This structure follows the standard transformer architecture commonly used in the literature.

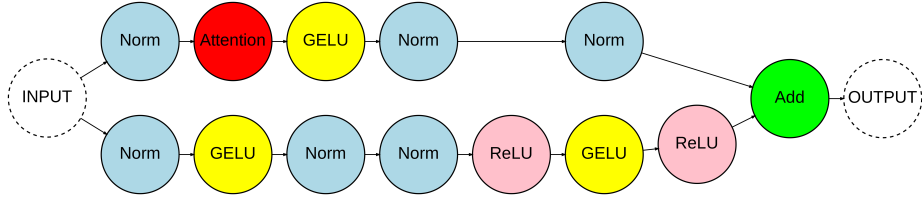


Figure 10: Architecture of EVO-4.

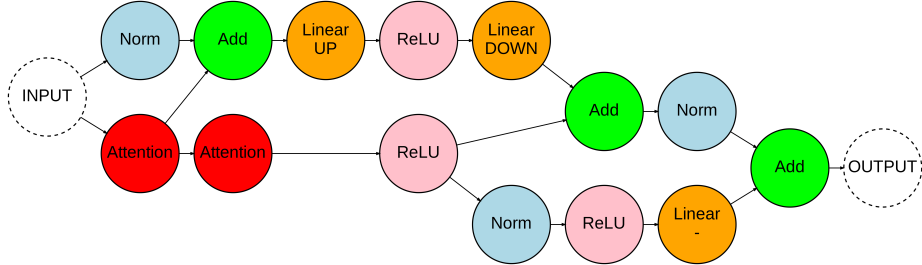


Figure 11: Architecture of EVO-6.

While EVO-3 preserves the main structure of the transformer, it simplifies its architecture. It removes the residual connection around the feed-forward block and replaces one of the normalization operations with a **ReLU** activation. The number of trainable parameters is almost identical (197,504 vs. 197,760), which shows that the improvement in performance cannot be explained by an increase in the parameter count alone.

EVO-4 differs substantially from the initial transformer architecture. The input is split into two parallel branches, which are later combined using an **Add** operator. While one branch preserves the **Attention** operation, the standard transformer residual and feed-forward structures are no longer present. Instead, the architecture relies heavily on normalization and activation operations. EVO-4 is also considerably smaller, containing only 67,072 trainable parameters, approximately 66% fewer than the single-layer transformer.

EVO-6 is the most complex of the three selected architectures and differs substantially from the initial transformer. It contains two consecutive **Attention** operators and multiple parallel branches. This increased structural complexity is also reflected in the parameter count, with EVO-6 containing 280,064 trainable parameters compared with 197,760 in the single-layer transformer. EVO-6 performs particularly well on CVRP100, but its larger size makes it difficult to separate the effect of the architectural changes from the increased model capacity.

Overall, the three evolved architectures differ substantially from each other, both in their structure and parameter count. This demonstrates the diversity of solutions explored by the CGP-based search, ranging from the relatively simple EVO-4 to the more complex EVO-6. Importantly, the search does not con-

verge towards a single type of modification of the transformer, but discovers several structurally different architectures that remain competitive. This suggests that multiple different encoder structures can provide effective solutions for the CVRP within the investigated search space.

6 Conclusion

In this work, we presented an automated Neural Architecture Search framework based on Cartesian Genetic Programming to design specialized Graph Neural Network encoder architectures for the Capacitated Vehicle Routing Problem. By isolating structural evolution to the node feature encoder while keeping the autoregressive decoder and REINFORCE training procedure fixed, our methodology directly evaluates feature representation capabilities. Combining CGP’s flexible grid encoding, neutral drift, partial weight inheritance, and fast proxy evaluation enables efficient exploration of high-dimensional operation spaces.

Empirical evaluations across CVRP benchmarks demonstrate two primary conclusions:

1. CGP-NAS significantly outperforms random search under identical proxy evaluation budgets, confirming the sample efficiency of guided evolutionary operators.
2. When seeded with standard transformer baselines, CGP-NAS discovers novel non-sequential architectures (such as EVO-3) that match or exceed single-layer transformer performance while achieving up to a 66% reduction in parameters. Furthermore, architectures evolved exclusively on small instances (CVRP10) generalize effectively to larger routing instances (CVRP100). The discovered architectures remain competitive with standard three-layer transformers – even outperforming them on larger instances (CVRP100) while requiring significantly fewer parameters. Overall, despite the inherent approximations of proxy evaluations, CGP-NAS provides a robust, sample-efficient alternative to random exploration for discovering tailored GNN encoders.

The resulting architectures are diverse and represent interesting competitive solutions to the initial transformer architecture. Despite limitations of the proxy evaluation, CGP-NAS is able to guide the architecture search more effectively than random search and can discover effective modifications to an existing transformer encoder.

7 References

- B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

- I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural combinatorial optimization with reinforcement learning. *CoRR*, abs/1611.09940, 2016. URL <http://arxiv.org/abs/1611.09940>.
- G. B. Dantzig and J. H. Ramser. The truck dispatching problem. *Management Science*, 6(1):80–91, 1959. ISSN 00251909, 15265501.
- T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.
- Y. Gao, H. Yang, P. Zhang, C. Zhou, and Y. Hu. GraphNAS: Graph neural architecture search with reinforcement learning. *arXiv preprint arXiv:1904.09981*, 2019.
- C. Garcia-Garcia, A. Morales-Reyes, and H. J. Escalante. Continuous cartesian genetic programming based representation for multi-objective neural architecture search. *Applied Soft Computing*, 147:110788, 2023.
- K. Helsgaun. Lkh-3: A vehicle routing and traveling salesman problem solver. <http://webhotel4.ruc.dk/keld/research/LKH-3/>, 2023. Accessed: 2026-06-27.
- J. Hopfield and D. Tank. Neural computation of decisions in optimization problems. *Biological cybernetics*, 52:141–52, 02 1985. doi: 10.1007/BF00339943.
- A. Hottung and K. Tierney. Neural large neighborhood search for routing problems. *Artificial Intelligence*, 313:103786, 2022. ISSN 0004-3702. doi: <https://doi.org/10.1016/j.artint.2022.103786>.
- M. Kaleta and T. Śliwiński. Learning insertion heuristics for the traveling salesman problem via neural networks and black-box optimization. In *2025 11th International Conference on Control, Decision and Information Technologies (CoDIT)*, volume 1, pages 2190–2195. IEEE, 2025.
- W. Kool, H. van Hoof, and M. Welling. Attention, learn to solve routing problems, 2019. URL <https://arxiv.org/abs/1803.08475>.
- J. K. Lenstra and A. Rinnooy Kan. Complexity of vehicle routing and scheduling problems. *Networks*, 11:221–227, Jan. 1981.
- H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search, 2019. URL <https://arxiv.org/abs/1806.09055>.
- H. Liu, B. Nicolae, S. Di, F. Cappello, and A. Jog. Accelerating dnn architecture search at scale using selective weight transfer. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 82–93, 2021. doi: 10.1109/Cluster48925.2021.00051.
- J. F. Miller, editor. *Cartesian Genetic Programming*. Natural Computing Series. Springer, 2011. doi: 10.1007/978-3-642-17310-3.

- J. F. Miller and P. Thomson. Cartesian genetic programming. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 121–132, Edinburgh, 2000. Springer-Verlag. ISBN 3-540-67339-3. doi: 10.1007/978-3-540-46239-2_9.
- M. Nazari, A. Oroojlooy, L. Snyder, and M. Takác. Reinforcement learning for solving the vehicle routing problem. *Advances in neural information processing systems*, 31, 2018.
- E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin. Large-scale evolution of image classifiers. In *International conference on machine learning*, pages 2902–2911. PMLR, 2017.
- M. Suganuma, S. Shirakawa, and T. Nagao. A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the genetic and evolutionary computation conference*, pages 497–504, 2017.
- M. Suganuma, M. Kobayashi, S. Shirakawa, and T. Nagao. Evolution of deep convolutional neural networks using cartesian genetic programming. *Evolutionary Computation*, 28(1):141–163, 03 2020. ISSN 1063-6560. doi: 10.1162/evco-a.00253.
- A. Torabi, A. Sharifi, and M. Teshnehlab. Using cartesian genetic programming approach with new crossover technique to design convolutional neural networks. *Neural Process. Lett.*, 55(5):5451–5471, Dec. 2022. ISSN 1370-4621. doi: 10.1007/s11063-022-11093-0.
- T. Vidal. Hybrid genetic search for the cvrp: Open-source implementation and swap* neighborhood. *Computers & Operations Research*, 140:105643, 2022.
- T. Vidal, G. Laporte, and P. Matl. A concise guide to existing and emerging vehicle routing problem variants. *European Journal of Operational Research*, 286(2):401–416, 2020. ISSN 0377-2217. doi: <https://doi.org/10.1016/j.ejor.2019.10.010>.
- O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. *Advances in neural information processing systems*, 28, 2015.
- X. Wu, L. Jia, X. Zhang, L. Chen, Y. Liang, Y. Zhou, and C. Wu. Neural architecture search based on cartesian genetic programming coding method. *arXiv preprint arXiv:2103.07173*, 2021.
- B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.
- B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *2018 IEEE/CVF conference on computer vision and pattern recognition*, pages 8697–8710. IEEE, 2018.

Author contributions Conceptualization, P.W. and M.K.; methodology, P.W.; software, P.W.; investigation, P.W.; writing, P.W. and M.K.; visualization, P.W.; supervision, M.K.;