# Pen2Graph: Synthesising Node-Edge Programs From Hand-Drawn Sketches

Žiga Kovačič[1]

[1]Cornell University

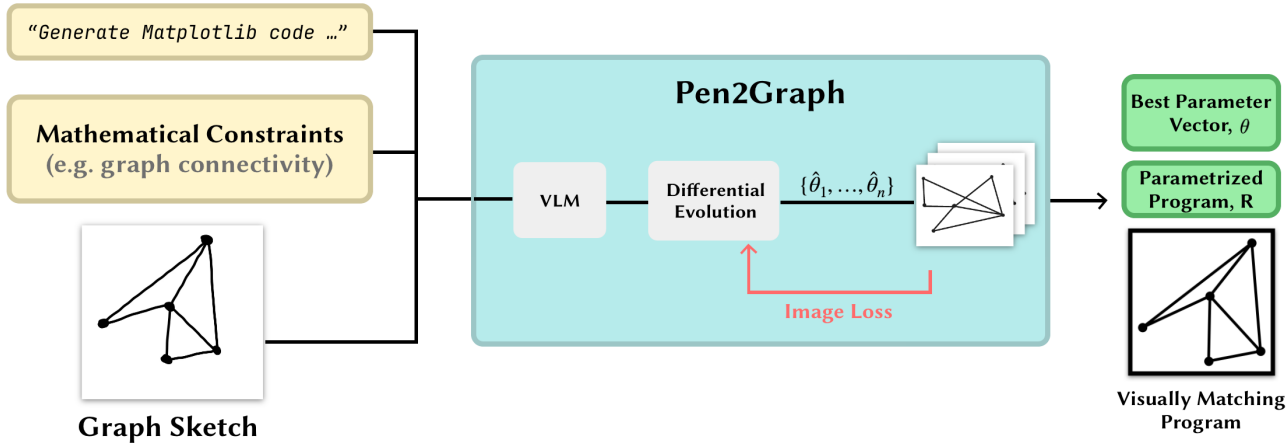


Figure 1: Pen2Graph pipeline takes as an input an image $I$ of a target sketch of a node-edge graph, a graph-connectivty for the sketched graph, and a text prompt asking for a parametrized program that generates a program $R$ and parameter set $\theta$. Connectivy of the graph is added to the prompt so that the VLM knows the connectivy of the sketched structure when generating the program. The program $R$ and parameter set $\theta$ are then used in the Differential Equation algorithm to search for the parameter set that when rendered with $R(\theta)$ produces a graph most similar to $S$. Guidance for the evolution is provided by loss between the rendered and the input sketch images. Pen2Graph at the end returns the parameter set, the parametrized program, and a rendering that best visually matches $S$.

**Abstract.** Creating appealing and reusable diagrams for research papers or any technical presentation is known to be a daunting process. The process pertains ideating, sketching the diagram, and finally transforming into a digital form. Digitalization can be a tedious multi step process, either employing 3D rendering tools, vector art software, or vector art DSLs. In this work we take a step towards automating this process using image guidance in evolutionary algorithms. Specifically, we focus on simple node-edge graphs with the goal of transforming an image of a diagram combined with mathematical properties of the graph into a program that visually matches the sketched graph. We show that the added mathematical constraint on the structure of the object increases performance of vision-language models in predicting a parametrized program that correctly represents the sketched object. Additionally we show that image guidance is effective in exploring the node-edge graph parameter space effectively.

## 1 Introduction

We commonly use sketches to communicate complex ideas, concepts, and information. However, trying to digitalize our sketches, i.e. produce high-quality, reusable versions of our sketches is a long and tedious task with a high learning curve; the task commonly consists of drawing tools such as Adobe Illustrator, 3D rendering software such as Blender, or even graphical languages like TiKz and Penrose. We show that using the sketch in combination with mathematical properties of the object drawn as a prior, we can recover a graphical program replicating our sketch.

Most of the "sketch-to-X" works use an image of the sketch and process it into some representation (vector language, SVG, etc.) describing that sketch, usually with a step that "processes" the image, finding the primitve elements, and a "synthesis" part where the primites are combined into a program describing the desired shape [Ellis et al.(2018)]. However, as we make our DSL more expressive to represent more complex shapes, the synthesis

quickly becomes underconstrained.

An insight we had is that in the process of creating diagrams *we usually know more about the diagram than just having its visual representation (sketch).* Indeed, we usually know extra information about the elements in the sketch and relations between them; we might know about the topology of the shapes drawn (distingushing a planar graph from a 3D polygon), the connectiveness of elements (nodes connected with edges), etc. This becomes increasingly important when we suprvise optimization of 3D shapes using 2D images, since we are inherently dealing only with their 2D projections—a non-injective function from 3D to 2D—and constraining that optimization space using mathematical properties of the object's geometry could speedup that process.

In this work we create a synthesis pipeline that takes as an input a *hand-drawn sketch* and a *specification* describing the sketched object and uses an *evolutionary algorithms* to find a program describing that sketch. Given that a lot of geometric objects can be represented as a set of nodes and edges we focus on generating node-edge graph sketches.

## 2 Related Work

While drawing geometric diagrams is a very general task, surprisingly there has not been much work on transferring sketches to interpretable and modular representations such as programs.

**Sketch-to-DSL.** Arguably the most relevant past work to this project is [Ellis et al.(2018)] where the focus is on hand-drawn-to-TikZ conversion and the learning of high-level patterns in the diagrams. While this does actually succeed in generating TikZ program that visually match the input sketch, it only supports simple constructs like circles, lines, and boxes. Additionally, the method requires a trained CNN for part recognition in the input image as well as a policy that provides guidance to the parameter search, and training specialized networks that work on more general drawings might be prohibitevly expensive.

Other relevant works focus on using program synthesis for generating CAD programs using parametric sketch models [Seff et al.(2021)] or focusing on engineering sketch generation for CAD using CurveGen and TurtleGraphics [Willis et al.(2021)]. There is a more general line of work focusing on finding arbitrary representations (called Templates programs) [Jones et al.(2024)] that find shared structures for different types of visual inputs (3D objects, 2D turtle graphics, etc.), but that too requires a lot of data and training. In our project we want to create a data-free method.

**LLM-to-plot.** Another relevant line of work is using LLMs or VLMs to generate plotting code given a diagram sketch [Belouadi et al.(2024)], which while fundamentally differnet from our approach because of its data-driven nature, it's still worth discussing here. Most methods in this direction prompt language-vision models with an image of the diagram and text prompt asking the model to generate some code that generates that graph. While this method could potentially generalize well with enough with slight modifications to the method, we do not have the kind of data needed to train this models; in this vein a very relevant work that could be generalized to this applicatio is [Li and Ellis(2024)].

In comparison, we will give the VL models both the visual input as well as a more mathematical constraint on the object which constraints the probability space more. This will allow us to use off-the-shelf, pretrained VLMs.

## 3 Method

Our method uses a sketch of the graph, $I$, and a specification, $S$, describing the graphs structure as guidance for generating visually matching node-edge programs. We first extract the parametrized program from the user friendly specification—in our case the *connectivity of the graph*—using a VLM. Then we use the sketch to guide the exploration of the space of possible parameters, using an evolutinary algorithm called Differential evolution, to find a prameter set that when rendered visually matches the sketch.

### 3.1 Overview

**Input Image and Specification.** As an input to our pipeline we take an image of the sketch of a node-edge graph, $I$, and the corresponding specification, $S$, describing connectivity of the graph. The specificaition takes a simple form where we define the `Nodes` and `Edges` connecting those nodes, which completely characterises the connectivity of any node-edge structure 2.

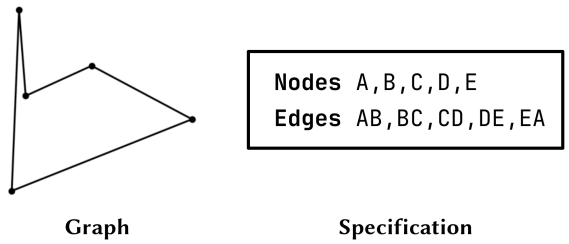


Figure 2: Example of a node-edge graph and the corresponding specification describing its connectivity.

Note that we do not have to label the nodes in our sketch, because the specified connectivity is invariant under any node-labeling. This greatly simplifies the workload on the user.

**Parametrized Program Generation.** Given the specification $S$ we want to generate a parametrized program $R$ describing and outputing an image of our graph. We frame this as a synthesis task where we prompt a VLM to synthesize a parametrized program that fits $S$. The program is parametrized with a vector $\hat{\theta}$ of `Node` $(x, y)$ positions. As one of the goals of our project is having a *reusable* and *editable* digital representation while allowing for fast iteration time, we decided to use `MatPlotLib` as our language. While languages like TiKz and Penrose might be more suitable to express geometric relation in the long run, they are either too complex or have a too slow rendering times, inhibiting any form of parameter search at this point in time.

**Guiding Parameter Search.** Once we obtain a parametrized program, its renderng with random parameters does not necessarily visually match our sketch yet. Given the parmaeter space $\Omega^n$ to explore we want to find $\hat{\theta}^\star \in \Omega^n$ s.t.:

$$\hat{\theta}^\star = \arg\min_{\hat{\theta} \in \Omega^n} \mathcal{L}(I, R(\hat{\theta})), \tag{1}$$

where $I$ is the input sketch, $R(\hat{\theta})$ is the rendered output of the program with parameters $\hat{\theta}$ and $\mathcal{L}$ is our loss function. Since our image based loss function is not differentiable, we need a non-gradient based approach to perform the optimization. We search the space $\Omega^n$ to find a solution to the optimization problem in (1) using Differential evolution algorithm where an individual's—in our case a vector $\hat{\theta}$—fitness function which measures how similar $R(\hat{\theta})$ is to $I$.

## 3.2 Differential Evolution

The objective function in (1) does not have computable gradients, so we search the parameter space using differential evolution algorithm (DE). DE like any evolutionary algorithm starts with some initial population. In our case we decided on a standard normal distribution of the population $\hat{\theta}_i \in \mathbb{R}^n$, which we will call the 0-th generation and denote it as $P_0 = \left\{ \hat{\theta}_i \right\}_{i=1}^{m}$, where $|P_0| = m$ is the number of individuals in the population at each generation (i.e. step of the evolution). Then the algorithm updates the current individuals in the population and proposes new individuals to be added to this generation. This is done through the processes of mutation and crossover. In DE, the initial (random) population of candidates in generation $t$, $P_t$, is updated by computing new candidate vectors as simple expressions of the most fit candidates; this is expressed as $\theta_{\text{new}} = U(\phi)$ for some $\phi \subseteq P_t$ and a simple operation $U$ (e.g. vector difference, summation, etc.). $\theta_{\text{new}}$ is then either accepter or rejected based on whether its fitness score is better or worse than the individual being replaced.

**Mutation.** In our case we set the mutation function to:

$$U : (\vec{a}, \vec{b}, \vec{c}) \to \vec{a} + \lambda(\vec{b} - \vec{c}),$$

i.e. adding a multiplicative factor of a vector distance between two indviduals to another individual. If we take steps by subtracting the vector with a higher loss from a vector with a lower loss, we get a crude approximation of the descent direction.

**Self-Mutation.** We equip DE algorithm with a so call "self-mutation" step, which can be summarized as a slight pertubation of the current indvidualss' parameters. The intuition for this is that once some of individuals $\hat{\theta} \in P_t$ are close to the optimal $\theta^\star$, we do not want to use other, potentially less fit individuals as anchors for our mutation; rather, we want to simply perturb $\hat{\theta}$ by a small amount. In the current version we perturb the $\hat{\theta}_i$ by sampling from a $|\hat{\theta}_i|$-dimensional normal distribution:

$$\hat{\theta}_{i,\text{self}} = \hat{\theta}_i + \mathcal{N}(I, \Sigma),$$

where the covariance coeficcients of $\Sigma$ are set to a scaled version of the mutation magnitude, $\lambda$. Since we employ a decreasing $\lambda$ schedule—discussed later—, this essentially enforces that as we get closer to our solution, the self-mutation will decrease as well.

In the future self-mutation could be improved by sampling multiple perturbuations at each step, determining the more and the less fit perturbed individuals, and using vector distance between them to approximate the direction of fitness ascent.

**Simulated Annealing.** Proposed individuals $\theta_{\text{new}}$ at each step are either accepted— and added to $P_t$—or rejected. We consider a more sophisticated acceptance criteria which still accepts worse individuals with a certain probability. Concretely, for some $\theta_{\text{new}}$ even if $\forall \theta_i \in P_t, \mathcal{L}(R(\theta_{\text{new}}), I) > \mathcal{L}(R(\theta_i), I)$, the probability of $\theta_{\text{new}}$ being added to $P_t$ follows an exponentially decreasing schedule, $\mathbb{P} = e^{-kt}$, where $t$ is the current generation. This allows the algorithm to jump out of local optima earlier in the evolution process.

**Parameter Schedules.** DE is known to be sensitive to mutation and crossover rate parameters, so we adaptively change the parameters as we converge closer to the optimal solution. The intuition is that the initial stages of exploration $\hat{\theta} \in P_t$ could be far from the optimal solution so having larger mutation and crossover rates can help us jump out of local minima. We do this by using an exponential schedule $\alpha : t \to e^{-kt}$ on the mutation and crossover probability and mutatation magnitude that decreases from 1.0 to 0, where $k$ is the decay-rate hyperparameter and $t$ is the current generation. Once the loss values drop closer to 0, we explicitly decrease the mutation and crossover magnitudes so we don't accidentaly jump out of global maximum.

### 3.3 Image Guidance and Fitness Function

We hypothesise that images can be extremely hepful in finding the correct program parameters. For effective guidance, having a proper loss function is cruicial. Naive $L_2$ or $L_1$ losses will not converge, because they significantly penalize simple transformations such as shifts and rotations, pushing the network to diverge or result in a blank render. Instead we emply a combination of the Hausdorff distance $d_H$, mathching Fourier descriptors $l_F$, SSIM, and Blob-Overlap-Measure (called BOM and explained later):

$$\mathcal{L}(I, R(\hat{\theta})) = \left( d_H(I, R(\hat{\theta})) + l_F(I, R(\hat{\theta}) \right) (1 - \mathrm{SSIM}) \, (1 - \mathrm{BOM}). \tag{2}$$

$d_H$ measures the maximum distance between two finite sets $(A, B)$—in our case binarized pixels of the rendered image—and as such penalizes rotations and translations approximately by their magnitude. This in turn gives relatively good signals as to which renderings $R(\hat{\theta})$ are actually further away structurally from the target image in which are just an affine transformation away from the target.

Fourier descriptors are a compact set of Fourier coefficients that capture the essential frequency features of a shape. As such they are not too affected by seminatically non-significant transformations and also capture the structural similarity of rendered graphs.

Finally, we add a made up measure called "**Blob-Overlap-Measure**", which measures the overlap of nodes in two images. Specifically, we detect the $|V|$ major blobs in the images and measure the normalized overlap of the pixels. This measures how close the positions of nodes are between two images, while not being affected by any edges in the graph.

## 4 Experiments

For experiments we generate parametrized programs using LLamma-3.2 and Qwen models as described in 3. We initialize the population of size $m = 50$ and let it evolve for 100 generations. If the loss reaches the lower loss threshold earlier, we stop the evolution and save the best parameter vector. We test the performance both on synthetically generated node-edge graphs as well as hand-drawn sketches.

**LLM inference.** To obtain a parametrized program and a parameter set $\theta$ we prompt a LLM with a text prompt asking for a parametrized program in `MatPlotLib`, a text specification of the edge connectivity in the graph 2 and an image of the target sketch. The exact prompt is attached in the Appendix.

**Generation size $m$.** We explored different $m$ values, and we found that $m = 50$ was the best mix between quick generation completion time and exploration of new individuals. A more comprehensive search for $m$ could be performed, but this was not the focus of the project. We hypothesise that $m$ should increase with the number of nodes $|V|$ and edges $|E|$ in the graph drawn, but further study needs to be performed to assert this with full confidence.

**Synthetic Sketches.** We first test our method on synthetically generated sketches of node-edge graphs using MatPlotLib renderings that we noise to achieve a "sketch" look. We generate graphs where $|V|$ ranges from 3 to 5 and a large example of $|V| = 10$. For simplicty we set $|E| = |V|$. We then run Differential Evolution algorithm to optimize the parameters of the parametrized program.

**Real World Sketches.** We also test our method on real hand-drawn sketches of graphs. The sketches are drawn on a pure white background with a black pen. Sketches are then converted to binarized image representation before being used for image guidance. Before performing the optimization we also find the approximate line thickness that best matches the sketch.

## 5 Results

### 5.1 Synthetic Sketches

Image guidance on synthetically generated sketches proves to be very effective and the optimization converged on all the attempted test cases, some are displayed in Fig. 3. For more test cases refer to the Appendix.

### 5.2 Hand-drawn Sketches

We find that when hand-drawn sketches become increasingly more noisy (e.g. wiggly edges), the optimization takes a longer time to converge. This is because the loss

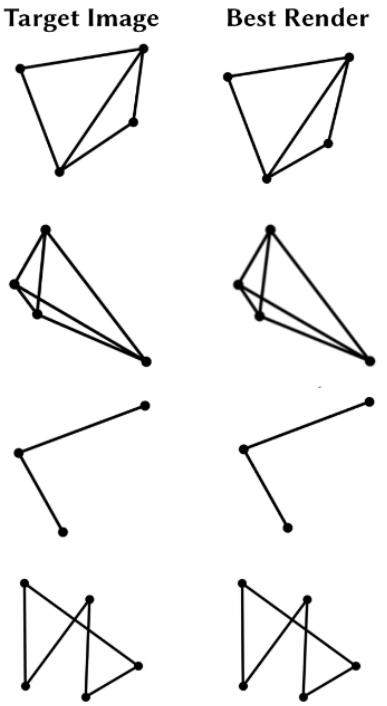


Figure 3: Results on synthetically generated sketches. Target sketch on the left and best generated parameter set on the right.

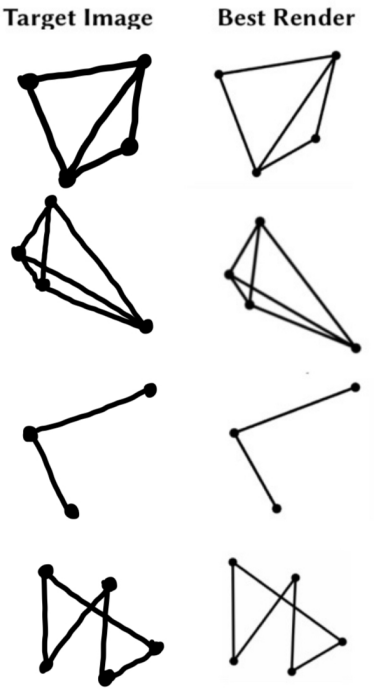


Figure 4: Results of optimization on synthetically generated sketches. Target sketch on the left and best generated parameter set on the right.

function compares a noisy sketch to a straight-lined render from MatPlotLib. We attempt to mitigate this issue by blurring both images, but the improvement does not appear to statistically significant. Sample results of the optimization are in Fig. 4.

## 5.3 Runtime

We report the average time to find a close to optimal parameter set in our synthetic sketch test cases. We set the loss lowerbound at $\delta = 5.0$ and for each $n$ (number of nodes) we determine the time by generating $\geq 3$ examples and averaging.

| # Nodes | Average Runtime |
|---|---|
| 3 | 2.9’ |
| 3 | 3.5’ |
| 4 | 4.2’ |
| 5 | 4.8’ |
| 10 | 6.6’ |

Table 1: Average optimization time for graphs with different number of nodes and $|V| = |E|$.

$\delta$ value was set empirically by looking at what loss value do most renderings look indistinguishable from the input

Note that the results were obtained by running to code on a CPU without any GPU optimization, which could speedup our computations a lot. We can see that with the size of $n$ the time does increase, but the complexity is not too prohibiting. That said, we did see an increase in the time when increasing the ratio of $|E|/|V|$. We hypothesis this is the case becase—besides the obvious fact that there are more parameters to be optimize—the increased ratio means more of the pixels come from edges which means there are more degrees of freedom in which two different renderings can have similar loss functions. We plot an example of the decreasing loss through generations in 5.

## 5.4 Ablation Studies

**Specification.** We experiment with using and not using the specification (graph connectivy) as the input to the VLM. We observe that for simple example graphs (triangle, square, pentagon etc.) that are very similar to the things VLMs have saw already, the output program is indeed correct. But as soon as the graph comes out of distribution of things the VLM has seen before, the outputs are incorrect. In this setting we query the model 50x and

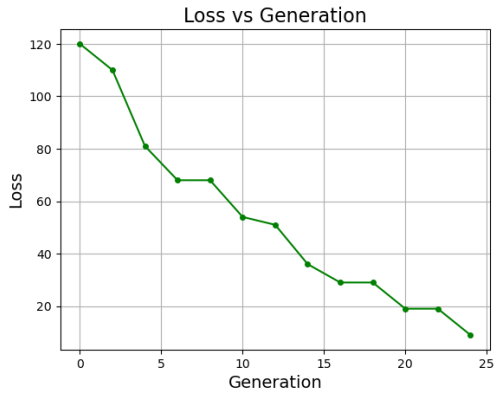


Figure 5: Progress of fitness through the generations of evolutionary algorithm

in not so complicated graphs that might not be too common, all of the 50 outputs do not represent the correct graph structure. If the specification is passed to the VLM

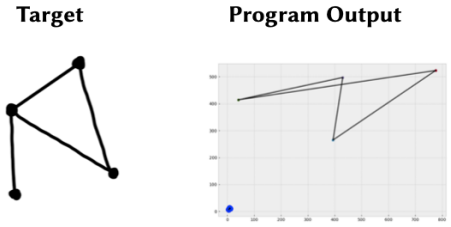


Figure 6: Input image and randomly initialized rendering of the returned program from VLM with no specificaition provided.

in addition to the prompt and the sketch image, then the resulting parametrized programs return correct programs almost 100% of the time in our experiments.

**Differential Evolution.** Our implementatin of differential evolutin has a few additions on top of a vanilla version and we evaluate the need for adding those here.

Firstly, we observe that if we remove "self-mutation", then the time to convergence is about 3x longer and sometime the algorithm doesn't even converge in the first 100 generations. We hypothesise this is the case because even if we land on $\hat{\theta} \approx \theta^{\star}$, the next promposed candidate is not guaranteed to be on average close to $\hat{\theta}$, since we compute it using a vector difference between two other individuals in the population. If the multiplicative factor $\lambda$ of the vector difference being added to $\hat{\theta}$ has not decayed enough, then the proposed candidate will potentially be too far away from the optimal to be accepted. As such once the rates decay enough, the optimization usually converges.

Secondly, removing the mutation and crossover rate schedule causes prolonged convergnence times for quite similar reason. Once some of our $\theta \in P_t$ come close to $\theta^{\star}$, we want to consider the candidates close to that candidate, besides considering more dissimilar candidates received from crossover. If the magnitudes and rates of mutation are too high, the new candidates will likely be far away from the optimal. At the same time, in the early stages of the evolution, we want to explore the space of solutions more aggresivelly, justfiying the need for larger rates.

# 6 Conclusion

In this work we study the potential of mathematical in addition to visual information as guidence for transforming a sketch of a node-edge graph to a parametrized program representation with parameter set that produces a visually matching graph. We show that using mathematical constraints on the structure of the sketched object is helpful in genearting the parametrzied program and that image guidence is effictive at providing the right signals to an evolutionary algorithm to perform optimization over the space of solutions.

**Limitations and Further Work.** Main limitations of our work are that we currently do not determine the directions of mutations based on the ascent of the fitness function, which could speedup the exploration process. Additionally, our method only works on a small subset of diagrams, namely node-edge graphs in a 2D plane.

We think that combining more matheamtical constraints with visual constraints on the program generation process for diagram generation tasks is a promising direction. In the future we could explore a 3D setting of node-edge graphs, where supervision would be performed using multiple views. Additionally, we could attempt to combine our approach with a preexisting DSL language like Penrose and ideally train a model to take in a sketch of a diagram and output the program describing that sketch in the DSL, motivated by [Li and Ellis(2024)].

# References


[Belouadi et al.(2024)] Jonas Belouadi, Simone Paolo Ponzetto, and Steffen Eger. 2024. DeTikZify: Synthesizing Graphics Programs for Scientific Figures and Sketches with TikZ. *arXiv preprint arXiv:2405.15306* (2024).

[Ellis et al.(2018)] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. 2018. Learning to infer graphics programs from hand-drawn images. *Advances in neural information processing systems* 31 (2018).

[Jones et al.(2024)] R. Kenny Jones, Siddhartha Chaudhuri, and Daniel Ritchie. 2024. Learning to Infer Generative Template Programs for Visual Concepts. In *International Conference on Machine Learning (ICML)*.

[Li and Ellis(2024)] Wen-Ding Li and Kevin Ellis. 2024. Is Programming by Example solved by LLMs? *arXiv preprint arXiv:2406.08316* (2024).

[Seff et al.(2021)] Ari Seff, Wenda Zhou, Nick Richardson, and Ryan P Adams. 2021. Vitruvion: A generative model of parametric cad sketches. *arXiv preprint arXiv:2109.14124* (2021).

[Willis et al.(2021)] Karl DD Willis, Pradeep Kumar Jayaraman, Joseph G Lambourne, Hang Chu, and Yewen Pu. 2021. Engineering sketch generation for computer-aided design. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2105–2114.


# Appendix

## 6.1 Prompt

Here we provide the exact prompt used for the expriment where we didn't provide the graph connectivity to the VLM.

**Prompt without connectivty**

Describe the connectivity and the nodes of the graph in the image. I will tell you there are $|V|$ nodes and $|E|$ edges. Then, generate a function that takes in a list of node positions and runs a matplotlib code to recreate the graph and also saves the plotted graph as a random string filename. There should only be one "'python"' block in the response and no other text.

Now we provide the exact prompt used for the expriment where we did provide the graph connectivity to the VLM.

**Prompt without connectivty**

Describe the connectivity and the nodes of the graph in the image. I will tell you there are $|V|$ nodes and $|E|$ edges. The graph has the following edges: <list_of_edges>. Then, generate a function that takes in a list of node positions and runs a matplotlib code to recreate the graph and also saves the plotted graph as a random string filename. There should only be one "'python"' block in the response and no other text.

## 6.2 Results

Additional results:

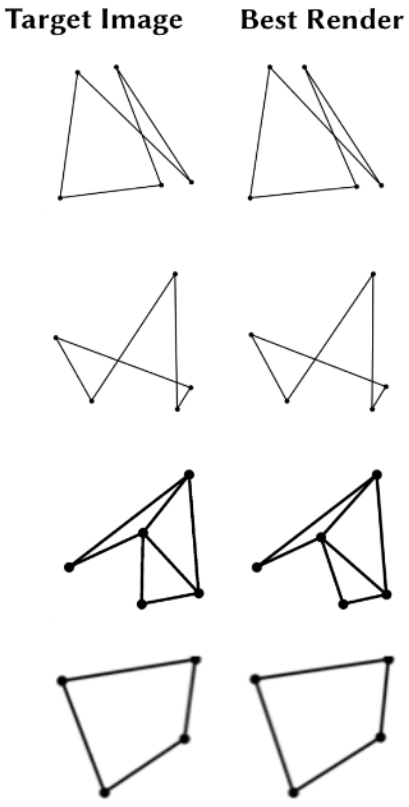


Figure 7: Optimization results using syntetically generated target sketches.

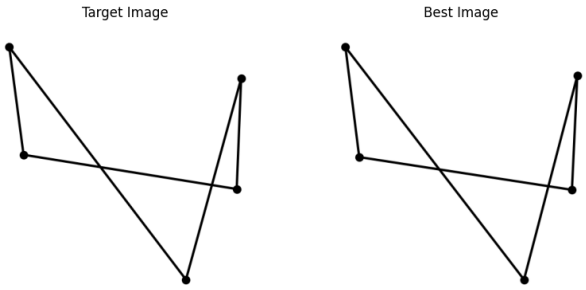


Figure 8: Example of using the pipeline to create a digital version of the sketch on a synthetic node-graph image