上海交通大学学位论文

基于离散贝叶斯推断的自动化 PCB 布线方法

姓　　名：罗正，周小米，甘润，陈颀轩

学　　号：521370910152，521370910065，521370910103，521370910128

导　　师：邹桉

学　　院：密西根学院

专业名称：电子与计算机工程

申请学位层次：学士

2025 年 8 月

A Dissertation Submitted to

Shanghai Jiao Tong University for the Degree of Bachelor

AUTOMATIC PCB ROUTING USING DISCRETE BAYESIAN INFERENCE

Author: Zheng Luo, Xiaomi Zhou, Run Gan, Qixuan Chen

Supervisor: An Zou

UM-SJTU Joint Institute

Shanghai Jiao Tong University

Shanghai, P.R. China

August 6^{th}, 2025

上海交通大学

学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，已在文中以适当方式予以致谢。若在论文撰写过程中使用了人工智能工具，本人已遵循《上海交通大学关于在教育教学中使用 AI 的规范》，确保人工智能生成内容的应用场景、引用范围及标注方式均符合规定，并杜绝学术不端行为。本人完全知晓本声明的法律后果由本人承担。

学位论文作者签名：罗正 陈倾轩

日期：2025 年 8 月 11 日

上海交通大学

学位论文使用授权书

本人同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。

本学位论文属于：

☑**公开论文**

☐**内部论文**，保密☐1 年/☐2 年/☐3 年，过保密期后适用本授权书。

☐**秘密论文**，保密____ 年（不超过 10 年），过保密期后适用本授权书。

☐**机密论文**，保密____ 年（不超过 20 年），过保密期后适用本授权书。

（请在以上方框内选择打“✓”）

学位论文作者签名：罗正 陈倾轩　　指导教师签名：An Zou

日期：2025 年 8 月 11 日　　日期：2025 年 8 月 11 日

摘 要

本文研究一种以离散贝叶斯推断作为核心思想的印刷电路板自动布线算法，在朴素的基于规则的自动布线算法基础上提升运行效率和布线质量。我们提出的贝叶斯推断算法通过对印刷电路板中走线的并行采样，迭代计算各个走线之间的干扰情况，得到它们的相对评分，从而给基于规则的回溯算法提供更好的启发式规则，以降低回溯算法探索分支数量的期望。我们以桌面应用程序的形式实现了该算法，并与流行的开源自动布线程序 FreeRouting 进行性能和效果的比较，结果表明我们的算法在大部分简单测试用例中快于 FreeRouting，且布线质量相当。我们由此得出结论：离散贝叶斯推断作为一种轻量级的算法运用于印刷电路板自动布线任务，在平衡布线效率、布线质量和运算开销方面有着较大的潜力。

关键词：离散贝叶斯推断，PCB 自动布线，多目标优化

ABSTRACT

This paper presents an automated printed circuit board (PCB) routing algorithm based on discrete Bayesian inference, aiming to improve runtime efficiency and routing quality compared to naive rule-based routing approaches. The proposed Bayesian inference algorithm employs parallel sampling of PCB traces and iteratively evaluates interference between traces to compute relative scores. These scores provide enhanced heuristic rules for the rule-based backtracking algorithm, reducing the expected number of explored branches. We implemented the algorithm as a desktop application and compared its performance with FreeRouting, a popular open-source autorouting program. Experimental results demonstrate that our method outperforms FreeRouting in speed for most simple test cases while maintaining comparable routing quality. Our findings suggest that discrete Bayesian inference, as a lightweight algorithm for PCB autorouting, holds significant potential for balancing routing efficiency, quality, and computational overhead.

Key words: discrete Bayesian inference, automatic PCB routing, multi-objective optimization

Contents

Chapter 1 Introduction

1.1 Background

Printed circuit board (PCB) autorouting is a critical step in electronic design automation (EDA), where the objective is to efficiently connect components while adhering to design constraints such as signal integrity, spacing rules, and layer limitations. Traditional rule-based routing algorithms, which rely on predefined heuristics (e.g., A* search or maze routing), offer low computational overhead but often produce suboptimal routing quality due to their inability to adapt to complex interference patterns between traces. Conversely, modern deep learning-based approaches leverage neural networks to learn high-quality routing policies from large datasets, significantly improving routing success rates and design efficiency. However, these methods demand substantial computational resources, making them impractical for real-time or resource-constrained applications.

This trade-off between routing quality and computational cost has motivated research into lightweight yet adaptive algorithms. In this work, we propose a discrete Bayesian inference-based autorouting algorithm that bridges the gap between efficiency and effectiveness. By probabilistically modeling trace interactions through parallel sampling and iterative scoring, our method generates dynamic heuristics for rule-based backtracking, reducing the search space while maintaining competitive routing quality.

1.2 Related Work

We have reviewed several papers regarding automatic PCB routing algorithms.

1.2.1 Unet-Astar: Fast Unified PCB Routing via CNN and A* Search

Yin et al.[1] propose a hybrid method combining a CNN-based model (Deeper-Unet[2]) and a traditional A* pathfinding algorithm[3] to achieve **unified PCB routing**, covering both escape and area routing. The network predicts recommended regions to guide pathfinding, thereby reducing routing time and improving overall quality.

Key Contributions:

- Reformulates routing as a segmentation-like task.
- Introduces Deeper-Unet for region recommendation.
- Uses predicted regions to accelerate A* search.
- Achieves up to 70% speedup and reduces via count by 22%.

Experimental Setup: The authors implemented the algorithm using Python and tested it on a desktop system with an **Intel Core i7-7700K CPU**, **NVIDIA RTX 2080 Ti GPU**, and **64GB RAM**, running Windows 10.

Quantitative Results: In their benchmark of 200 test cases:

- **Runtime:** Reduced from 302s (baseline A*) to **78s**, achieving approximately **4× speedup**.
- **Via Count:** Decreased from 331.11 to **257.30** (about **22.3%** fewer vias).
- **Wirelength:** Slightly reduced from 8822 to **8797** units.

These results indicate that Unet-Astar not only improves efficiency but also leads to more compact and manufacturable routing layouts. However, it is worth noting that the system requires relatively high computational resources, especially the presence of a dedicated GPU for inference.

1.2.2 Chip Placement with Diffusion Models: Zero-Shot Generative Modeling

Lee et al.[4] reframe chip placement as a **conditional generation problem**, solved using denoising diffusion probabilistic models (DDPMs). Their approach bypasses reinforcement learning (RL), which is typically slow and struggles to generalize. Instead, the authors use synthetic data to pretrain a diffusion model capable of placing new circuits in a zero-shot manner using guided sampling.

Key Contributions:

- Creates synthetic netlist/placement datasets (v1/v2) for scalable pretraining.
- Proposes a GNN-attention hybrid model[5] that balances efficiency and expressiveness.
- Introduces guided sampling to jointly optimize legality and wirelength.
- Demonstrates zero-shot transfer to the IBM (ICCAD04) benchmark dataset.

Experimental Setup: Their models were trained on machines with **Intel Xeon Gold 6326 CPUs** and a single **NVIDIA A5000 GPU**, using the Adam optimizer for 3 million steps and fine-tuned for 250k steps.

Quantitative Results: On the IBM benchmark:

- **Legality:** Improved from 0.8835 to **0.9970** with guided sampling.
- **HPWL (wirelength):** Reduced from 3.203 to **2.976** ($\times 10^7$), outperforming **DREAMPlace**'s 3.724.
- **Mixed-size placement:** Achieved lowest average HPWL of **22.7** ($\times 10^6$) across 18 IBM circuits, compared to DREAMPlace's 23.6 and ChiPFormer's 28.9.

Computational Considerations: While their method avoids costly online RL training and supports zero-shot inference, training the diffusion model still requires high-end GPU resources and millions of synthetic samples. Inference for new circuits, however, can be completed in minutes using guided sampling, offering a favorable trade-off between quality and latency.

1.2.3 GPCB Routing: Token-Level Transformer Routing from Human Demonstrations

Chen et al.[6] reformulates PCB routing as a **sequence modeling task**, using tokenized representations of routing flows and applying a transformer architecture inspired by generative pre-trained transformer (GPT). Unlike traditional model-based or heuristic routing, GPCB leverages human-routed datasets and mimics human routing behavior through token prediction.

Key Contributions:

- Introduces a flow-based network encoding to transform routing into a token prediction problem.
- Utilizes a sliding window and local memory architecture with cross-attention for long-range flow modeling.
- Incorporates multi-information fusion, including capacity and start/end points, to enhance routing accuracy.
- Implements post-routing and masking mechanisms to ensure design rule compliance and 100% routability.

Experimental Setup: The model was trained on a high-performance machine with a **16-core CPU** and a **GPU with 9728 cores and 16GB VRAM**. The training involved up to

1.5 million iterations with 3.6 million tokens, using the nano-GPT framework and PyTorch backend.

Quantitative Results:

- **Routability:** Achieved 100% routability after post-routing (PR), compared to **87.90%** for method[7] and <90% for FreeRouting.
- **Runtime:** Averaged **4.99 seconds** per case; up to **6.67× faster** than FreeRouting.
- **Wirelength:** Normalized average wirelength was 1.00, outperforming method[7] (1.13) and method[8] (1.12).
- **Case Study:** Routed **262 nets and 1526 pins** in under **20 seconds** with 100% success rate.

Computational Considerations: GPCB requires substantial compute during both training and inference. While the transformer-based architecture enables high parallelism, the use of large memory blocks, attention layers, and post-processing steps such as A* rerouting introduces latency and GPU dependency. This makes it challenging to deploy on low-resource devices. Nonetheless, GPCB demonstrates a powerful paradigm shift by leveraging human routing knowledge, achieving state-of-the-art performance in routability and wirelength with a token-level generative model. It proves the feasibility of adopting large-scale pretrained models in EDA applications.

1.3 Research Subject

Although papers mentioned above made substantial contributions to high-quality automatic PCB routing, they typically have high computational cost and are not suitable for deploying on a regular personal computer. Our goal is to explore the possibility of using Bayesian inference to balance the speed and optimality in PCB routing. This idea is based on the following observations:

1. Due to the complex problem set-up of modern PCB routing, including variable pad and trace shapes and complex design rules, routing a PCB is generally considered not optimally solvable using trivial or low time-complexity algorithms. Therefore, we typically use greedy rule-based algorithms to obtain a near-optimal solution or use machine learning to learn from small-scale problems and hope the model generalizes to large-scale ones.

2. The difficulty regarding pure machine learning approaches is that a function that maps a PCB routing problem to its optimal solution is non-differentiable (see figures 1–1, 1–2). This makes PCB routing intractable to solve end-to-end using probabilistic models like variational autoencoders and diffusion models, which requires the output to be differentiable with respect to its input. Another problem with pure machine learning approaches is that they are likely to give a seemingly correct solution but actually violates the design rules (like hallucination in LLMs). Therefore, when considering probabilistic approaches, we typically use a hybrid approach that includes a rule-based backbone to ensure the correctness of the overall routing.

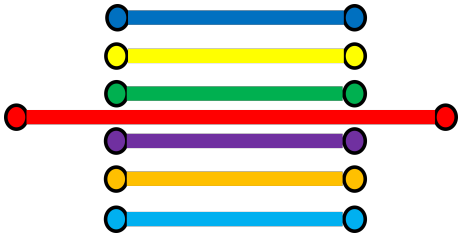


Illustration 1–1 An example of a drastic change in output (non-differentiable) in response to a small change in input

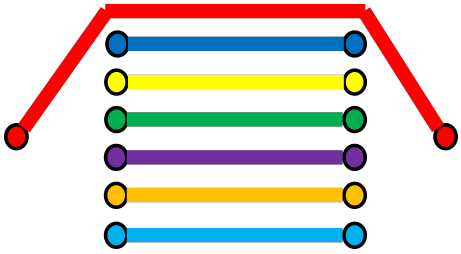


Illustration 1–2 The middle trace seems to "teleport" to a random place when it can no longer squeeze through a gap

3. The difficulty regarding rule-based approaches is the trade-offs between trace candidates. A fundamental design rule of PCBs is that traces from different nets cannot overlap each other in the same layer. Therefore, different routing orders typically produce different solutions that vary in correctness and optimality. However, the best routing order cannot be trivially determined, and typically involves exhaustive searching strategies like backtracking, or heuristic strategies like simulated annealing. Therefore, the core problem in rule-based approaches is to determine the best routing order.

4. Although it is intractable to solve the PCB routing problem end-to-end using probabilistic approaches, we observe strong correlations between the acceptance of a trace candidate and certain characteristic scores of the trace, including its length, via counts, etc. If we obtain these characteristic scores and use them to guide the rule-based searching algorithm, we can focus on more promising directions and reduce the average time for finding a satisfactory solution. Most of the characteristics can be easily quantified using manually specified algorithms.

However, there is one important characteristic that cannot be trivially calculated, which is described as "how valuable are the trace candidates that must be given up giving we choose the current trace". We call this the "opportunity cost" of a trace candidate, meaning that if a trace candidate gets in the way of too many highly-prospective trace candidates, we are less willing to pay the cost to adopt it. This is where "Bayesian inference" comes in: we propose an algorithm based on Bayesian inference to calculate the "opportunity cost" of each trace candidate.

In conclusion, the main idea of our PCB routing algorithm is to guide a rule-based searching algorithm with the characteristic scores of each trace candidate to speed up the searching significantly while maintaining the correctness and quality of the solution. And for the critical "opportunity cost" characteristic, we propose a Bayesian inference algorithm to infer it from the scores of other traces.

Chapter 2 Research Methodology and Approach

2.1 Discrete Bayesian Inference as the Core Algorithm

Bayesian Inference[9] is the core idea behind many state-of-the-art generative models like variational autoencoder (VAE)[10] and denoising diffusion probabilistic model (DDPM)[11].We choose discrete Bayesian inference as the core algorithm for PCB routing based on its ability to integrate prior knowledge with observed evidence.

The intuitive idea is that a trace candidate that is clear of any potential other traces in the way is more promising to be selected. If a trace candidate has a tendency to collide with other traces, it will be penalized by the number of traces in the way and how promising those traces are to be selected. Therefore, we can construct a probabilistic model where all trace candidates are probabilistic and can influence each other through an iterative updating policy.

The following algorithm is a simple formulation of this idea.

$$p(x) \coloneqq \prod_i (1 - p(\overline{x}_i)) \tag{2–1}$$

where x denotes the event of selecting a specific trace candidate, and $\overline{x}_i$ represents events where conflicting trace candidates are selected.

This formula states that the probability of adopting a trace candidate equals to the probability that none of the relevant trace candidates is in its way. However, in this formulation, the equilibrium is unstable, since at equilibrium, if one trace has a small increase in probability, the conflicting one will have a probability decrease in response, which will further increase the first trace's probability. This would cause the probabilistic model to collapse to a deterministic state, providing little help to the backtrack algorithm.

Therefore, we came up with the following regularization mechanism to stop the collapsing behavior.

We assign each trace candidate a manually specified prior probability (that hopefully reflects the average probability of the trace to appear in the final solution, given little knowledge about its context). Then, we make the "interference" to act as a "force" to push the probability away from the anchor, or the prior probability. Then, we need to set up a "counter force" that tends to push the probability back, which grows larger when the probability deviates fur-

ther, just like a spring. In this way, the probabilistic model will converge and reach a stable equilibrium.

To create the "counter force" effect, we design the algorithm as follows:

$$\text{raw penalty} = \sum_i p(\text{conflicting trace } i) \tag{2–2}$$

$$\text{penalty coefficient} = \exp\left(-\frac{\ln(2) \cdot \text{raw penalty}}{K}\right) \tag{2–3}$$

Formula 2–3 maps the "raw penalty" range of $[0, +\infty)$ to "penalty coefficient" range of $(0, 1]$. The hyperparameter K is the "half probability raw penalty", which describes the amount of raw penalty increase needed to cause the penalty coefficient to be halved. The input-output relationship is shown in figure 2–1.

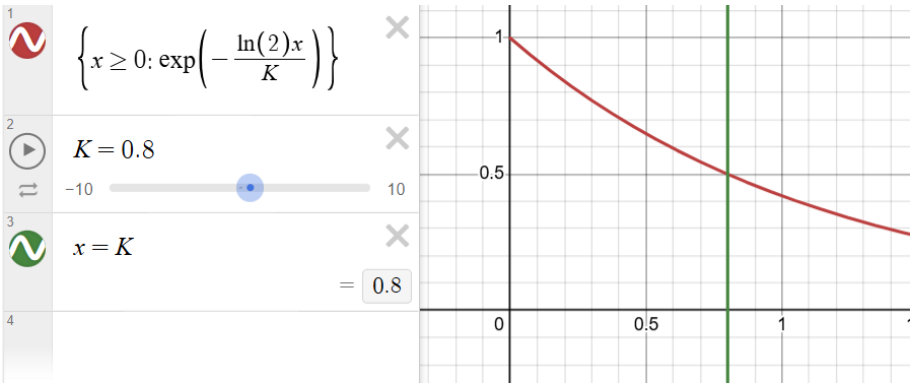


Illustration 2–1 Penalty coefficient with respect to raw penalty, illustrated using Desmos

Apart from the interference between trace candidates, another important metric for trace's quality is its length and via count, which we denote as "score" for simplicity. Similarly, we map the raw score of a trace to a "score coefficient" between 0 and 1 using the exponential function with negative exponential term:

$$\text{raw score} = \text{trace length} + \text{num vias} \cdot \text{via cost} \tag{2–4}$$

$$\text{score coefficient} = \exp\left(-\frac{\ln(2) \cdot \text{raw score}}{K'}\right) \tag{2–5}$$

where K' is the "half probability raw score", the amount of increase in score needed for the score coefficient to be halved.

Finally, we have the update rule as:

$$p(\text{trace}|\text{other conflicting traces}) \longleftarrow p(\text{trace}) \cdot \text{penalty coefficient} \cdot \text{score coefficient} \quad (2\text{–}6)$$

Now you can see why the two hyperparameters are named with prefix "half probability". The coefficients apply directly to the trace's probability and reduce it proportionally.

2.2 Trace Sampling Mechanism

Now that we have the trace probability update rule, we need to generate the traces in the first place for the rule to apply.

2.2.1 First Iteration Traces

Unlike the naive backtrack algorithm that generates traces one by one, each is aware of the previous ones, our idea is to generate the traces "at the same time". This means that we run the path finding algorithm without considering traces of the same iteration as an obstacle. This will result in interleaving traces that could not be selected at the same time. See figure 2–2.

After that, we can apply the update rule on each trace simultaneously until convergence. The result will give us a good heuristic regarding which trace to select first.

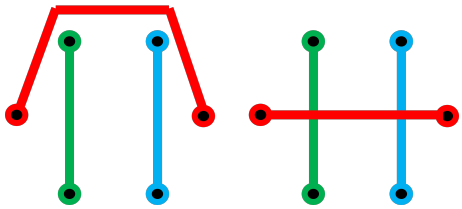


Illustration 2–2 In a one-layer PCB, backtrack trace generation (left), our trace generation (right)

2.2.2 Second Iteration Traces

Based on the first iteration probabilistic model, we can further "foresee the future" by generating the second iteration traces that are aware of the first iteration ones. Since the first iteration traces are probabilistic, the second iteration traces may cross them or avoid them.

Since it is computationally heavy to consider each case where an existing trace is either turned on or off and then sum up the marginal probabilities, we use the Monte Carlo approach to sample new traces. See figure 2–3.

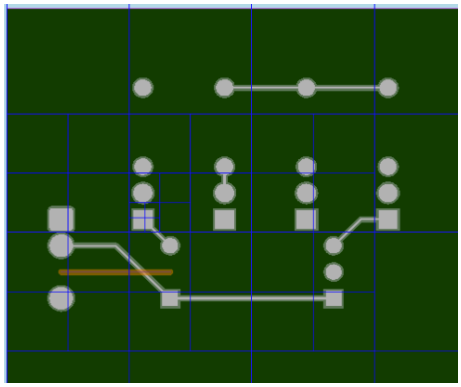


Illustration 2–3 Sample the existing trace obstacles according to their probabilities before generating a new trace

The significance of the second iteration traces is that they provide additional hints on the alternatives a pad pair connection can choose. If a second iteration trace is free of competition, it is likely to replace the first iteration version and thus yield the way to its conflicting trace. See figure 2–4 for the comparison between first and second iteration traces.

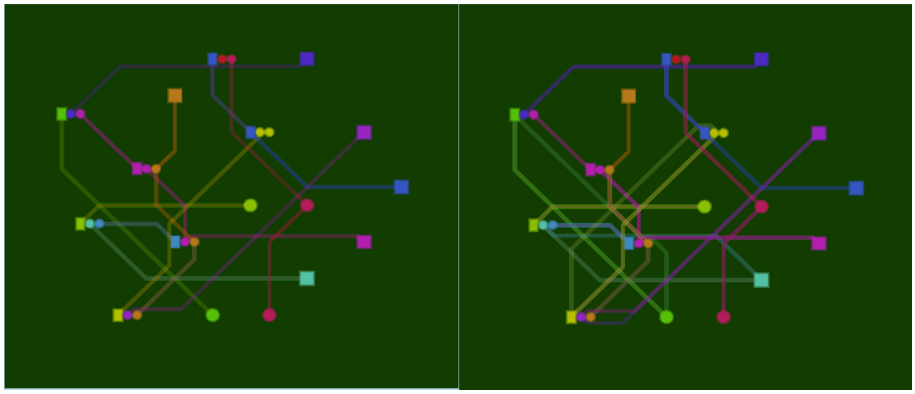


Illustration 2–4 First iteration traces (left), and both first and second iteration traces (right)

Theoretically, we can add third iteration traces to the probabilistic model, but they are costly to compute and do not contribute much to the result (since a third iteration trace is rare in a routing solution). Therefore, we decide to stop at the second iteration.

2.3 Trace Fixing and Progressing

After we have obtained a converged probabilistic model, we can determine the first few traces to route by selecting the ones with the hightest probability.

However, as we progress, the probabilistic model will begin to get outdated, and the trace candidates will become fewer and fewer for us to consider. Therefore, we need to sample more traces and update the probabilistic model periodically as we fix traces.

Now, there is a trade-off regarding how frequently we update the model and resample traces. A less frequent update will cause the model to outdate, providing less helpful heuristics; an exceedingly frequent update, while providing an up-to-date model, can be computationally expensive. We choose the update frequency to be once per two trace fixes for balancing model quality and computation efficiency.

2.4 Dead-end Handling

Our progressing policy does not involve making a recall on the determined traces when meeting a dead-end like the naive backtrack algorithm does, since it is hard to keep track of the trace combinations that have been visited. Therefore, our proposed algorithm does not guarantee to find a solution given enough time if there is one.

To solve this problem, we record the order of the traces determined till the dead-end, and pass the information as the heuristic to the naive backtrack algorithm and run it, which will eventually give us a solution if there is one.

Chapter 3 Implementation

3.1 Implementation Overview

3.1.1 Programming Language Selection

We choose the Rust programming language to implement and visualize our algorithm, because it has comparable runtime performance to C++, provides a rich toolchain for highly customizable visualization solution, and is highly maintainable for implementation-heavy projects.

3.1.2 Project Structure

We divided the whole project into four separate sub-projects (or "crates" in Rust):
1. A shared library that includes the PCB routing problem and solution formulations, utility data structures and functions, etc., that are shared across the whole project. 2. The parser sub-project. This includes all the parsing functions and intermediate representations that enables the parsing of the external PCB file formats from and to our router's format. 3. The router sub-project. This includes all the autorouting algorithm logic, which is the core to our project. 4. The app sub-project. This is a wrapper that combines the functionalities of the parser and the router, and provides an interactive interface for users to visualize the PCB autorouting process and understand the principle of our algorithm. We used Tauri as the web-view-based desktop application framework and Leptos as the frontend framework. For the autorouting process visualization, we used the WGPU graphics API.

The dependency graph is shown in figure 3–1.

The code is available at https://github.com/LuoZheng2002/bayesian_router_pro_max.

3.2 Implementing the Core Algorithm

The core algorithm is composed of three parts:

1. Sample trace.
2. Update probability.
3. Fix trace.

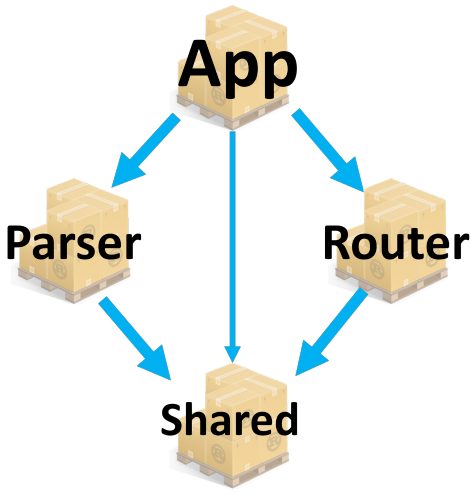


Illustration 3–1 The dependency graph of the project

These three steps form the backbone of our method, working together to iteratively refine the solution. The algorithm begins by generating candidate solutions through trace sampling, then adjusts the underlying probabilistic model based on observed outcomes, and finally selects the traces to ensure consistency with constraints. In the following sections, we will explain each part in detail.

3.2.1 Sample Trace

The trace sampling procedure is composed of two iterations. For the first iteration, we directly connect all the pad pairs without considering the interference from other traces. For the second iteration, we first sample the trace obstacles based on the first iteration traces' probability, and then generate the second iteration traces by avoiding these obstacles.

In the actual implementation, we have the following parameters to consider:

3.2.1.1 Max Number of Generation Attempts

We enforce a limit on how many trials of trace generation can be conducted in one iteration of sampling. The algorithm will keep trying to generate traces until a pad pair connection's trace count reaches the target number for the current iteration, or the number of generation attempts reaches this boundary.

3.2.1.2 Prior Probabilities

The prior probability describes how likely a trace is accepted in the final solution given very little information about it and its context. The actual value needs to be obtained through collecting a large number of PCB samples and count the number of straight and detoured traces, which involves a lot of work and may not be worthwhile. For simplicity in this project, we estimate the prior probabilities ourselves.

We manually assign each trace a prior probability according to their iteration number. We set the default values to be 0.8 for the first iteration and 0.6 for the second iteration. The absolute value does not matter since we eventually select traces based on their relative posterior probability. Therefore, a pair of $(0.8, 0.6)$ is the same as $(0.4, 0.3)$ for the first and second iteration's prior probability.

However, what do matter is the probability that is used for the Monte Carlo obstacle sampling, which was mentioned before with figure 2–3. In this case, if we assign the manual prior probabilities to be too high or too low, the obstacle sampling result will not represent the practical scenario. To address this problem, we normalize all the traces' posteriors by mapping them from $[\min \text{posterior}, \max \text{posterior}]$ to $[0.2, 0.8]$. In this way, we obtain a reasonable range of probabilities for sampling, and still retain traces' relative probability information.

3.2.1.3 Number of Second Iteration Traces

For the first iteration, a pad pair connection can only have one optimal trace path since it does not consider interferences from other traces and no random sampling is involved. However, for the second iteration, we may obtain more than one trace based on the different set of obstacles the connection has to avoid.

We encourage generating second iteration traces as many as possible, since in this way we have more alternatives to choose from and thus ease the tension of the conflicting first iteration traces. However, doing so is computationally expensive, so we set a max limit on the number of second iteration traces per connection.

3.2.2 Update Probability

We maintain an collision adjacency matrix that describes which trace conflicts with which, and update a trace's posterior probability based on only traces that conflicts with it.

We used formula 2–2, 2–3, 2–4, 2–5, 2–6 for the update policy as mentioned in the methodology section.

To make sure the trace probabilities update simultaneously, for each iteration we update a trace's posterior based on the posteriors of other traces in the last iteration.

3.2.3 Fix Trace

After we have the initial probabilistic model ready, we begin to determine and fix the first few traces based on their probability rankings. To make the probabilistic model up to date, we need to resample the traces and update their posteriors periodically. We use the hyperparameter "update model skip stride" to describe the update frequency. It means how many trace fixes we can do before having to update the probabilistic model.

It is computationally heavy to resample the traces each time we update the probabilistic model, so we use a "trace cache" to store the traces that are ever generated through the whole problem, and consider them first before attempting to generate a new one using path finding.

3.2.4 Fallback Mechanism

As mentioned in the methodology section, our Bayesian inference algorithm alone does not guarantee to produce a valid solution when there is one, since the probabilistic model only provides a coarse indication regarding which trace is more promising, which could mislead the progress into a dead-end. Also, our algorithm does not implement the recall mechanism like what the naive backtrack algorithm does, so it will just halt at the dead-end.

To make the full algorithm always succeed given there is a solution, in the case where our algorithm reaches a dead-end, we collect the ordering of the trace fixes till the dead-end and send it to the naive backtrack algorithm. Then it will make full routing attempts starting from the orderings most adhered to the heuristic to the orderings least adhered to the heuristic. In the unlikely case where the heuristic points to a direction totally opposite to the actual solution, the naive backtrack fallback will still produce the solution given sufficient amount of time.

3.2.5 Handling Nets with Multiple Pads

Usually in a PCB there are nets that have more than two pads, like the power nets and the ground net. To simplify the problem, we used the Prim's algorithm to find the minimum spanning tree of the net based on the octile distance of the pad pairs. Therefore, whether there is a connection between two pads is predetermined and will not change as the probabilistic model updates.

3.3 Hybrid Grid-Shape Based A* Path Finding

The path-finding algorithm is a critical building block in both the naive backtrack algorithm and our Bayesian inference algorithm. It is responsible for connecting a pad pair by avoiding any given obstacles like pads and traces from other nets.

We choose the A* algorithm as the baseline algorithm for path-finding for simplicity and optimality. However, a considerable amount of modification is needed for it to perform well in the PCB path-finding task.

3.3.1 Problem Formulation

Modern PCB designs often involve different pad shapes and sizes, trace widths, via diameters, and clearances. A PCB may both include the standard 2.54mm-stride through-hole component and chips with tiny little round-rectangle-shaped surface mount pads. Considering this, the standard grid-based A* algorithm can hardly maintain a balance between granularity and performance. Therefore, we propose a modified A* algorithm that is shape-aware on the basis of grid search.

We formulate the problem as follows:

3.3.1.1 Pads

The most common shape of a pad includes circle, rectangle and rounded-rectangle. In practice, there are also capsule-shaped and oval-shaped pads, but for parsing and implementation simplicity, we discard these options and use a circle as a fallback representation.

We provide three kinds of primitive shapes for representing all the elements in a PCB problem: circle, polygon and line.

A round pad can be represented by a single circle.

A rectangle pad can be represented by a polygon with four vertices.

A rounded-rectangle pad can be represented by four corner circles and two overlapping rectangles (see figure 3–2).

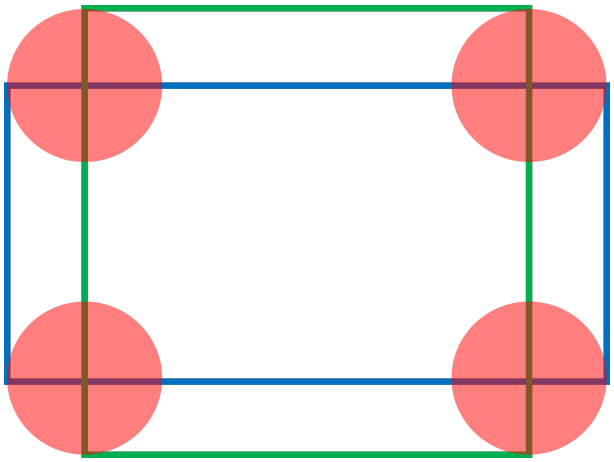


Illustration 3–2 A rounded rect pad is represented with 4 circles and 2 rectangles

3.3.1.2 Traces

In KiCad, a trace segment is of a long capsule shape, which can be represented by two circles and a rectangle. See figure 3–3. We use the same representation.

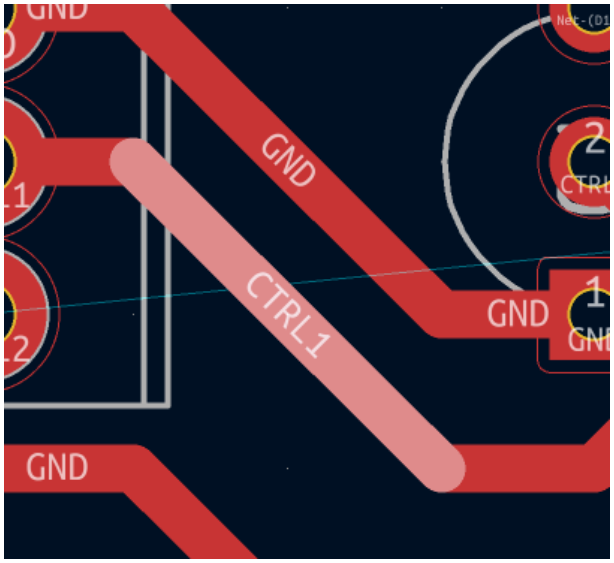


Illustration 3–3 In KiCad, a trace is a capsule shape.

A trace may contain vias if it spans different layers. A via is typically in a circle shape.

We only support round vias in our project.

3.3.2 Borders

We use the smallest rectangle that encloses every edge-cut element in the PCB problem as the border of the PCB. Due to time limit, we can only support a rectangular border in our project.

3.3.3 Collision Detection

Collision detection is vital in a shape-based context. In a purely grid-based A* algorithm, we go around an obstacle by visiting the set of obstacles accessed by grid coordinates, but in a shape-based context we have to call the collision detection algorithm each time we want to verify if a point on the PCB is accessible.

For a correct routing, a trace cannot overlap with pads, traces and vias of other nets in the same layer, but it can overlap with pads, traces and vias in the same net. Therefore, the obstacle is composed of pads, traces and vias of other nets. The PCB border is also an obstacle to be considered.

Apart from a direct overlap, a trace typically cannot get too close to elements of other nets due to the clearance requirements. For two elements to obey the clearance rules, they have to be separated by the max value between the two clearances of the elements. Therefore, we typically need to do the collision detection twice for a pair of elements, see figure 3–4.

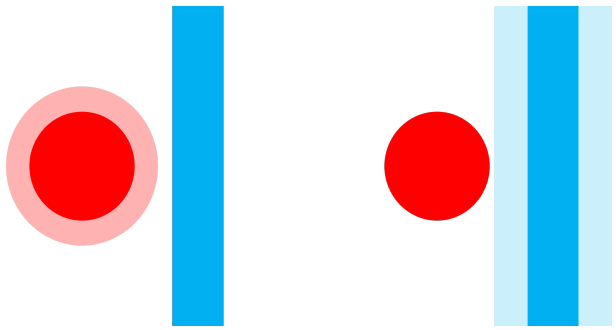


Illustration 3–4 First detection: red clearance vs. blue; second detection: red vs. blue clearance

Any collision detection between pads, traces and vias can be decomposed into a series of collision detection of primitive shapes. The collision detection between primitive shapes can then be calculated using the Separating Axis Theorem.

3.3.3.1 Optimized Collision Detection Using Quad Tree

We used the Quad Tree Algorithm to split the obstacles into different groups according to their spatial information, so that we no longer need to test the collision of two objects that are far away from each other. The region separation is demonstrated by figure 3–5.

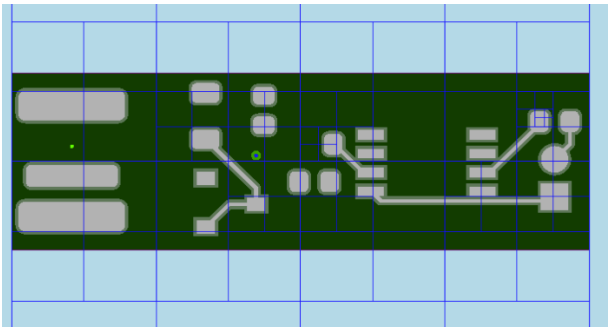


Illustration 3–5 The blue lines represent the boundary of each quad tree node's region. A large region will split into 4 sub-regions if the number of obstacles in it exceeds certain threshold.

3.3.4 Search Strategy

The search process employs several innovative techniques to improve efficiency in the PCB routing domain:

3.3.4.1 Eight-Direction Expansion

To balance computational efficiency and routing performance, the algorithm expands each frontier node in eight predefined directions: horizontal (0°, 180°), vertical (90°, 270°), and diagonal (45°, 135°, 225°, 315°). This structured approach ensures systematic exploration while optimizing resource usage.

3.3.4.2 Direct-to-Target Expansion

If a frontier node is aligned with the goal in one of the eight directions, we check for the collision in the way, and then push the end position to the frontier if there is no collision.

3.3.4.3 Grid-Point Prioritized Expansion

We prefer grid-point positions for the frontier nodes because they are more likely to be merged from different directions. In comparison, a dangling point (or a non-grid-point) is unlikely to expand and reach a new position that is already visited, so the memoization property

of the A* algorithm cannot be utilized to speed up searching.

If for any reason, we try to expand a frontier node with a dangling position, we will only expand it to a position that is less "dangling", and eventually to a grid point.

To formulate this idea, we use fixed-point numbers as coordinates of a position, which retain the ability to represent values with decimals, and also allow us to handle edge case precisely.

We can classify the 2D points into three categories:

1. Grid points. They have x and y coordinates to be a multiple of the expansion stride.

2. Partially dangling points. They are not grid points but are on one of the four kinds of edges in the grid system.

3. Fully dangling points, which is neither on a grid point nor on any of the edges. See figure 3–6.

For a partially dangling point, it is trivial to determine the direction and distance to go to the nearest grid point. However, for a fully dangling point to go to a partially dangling position, there is a caveat. Consider the scenario displayed by figure 3–7:

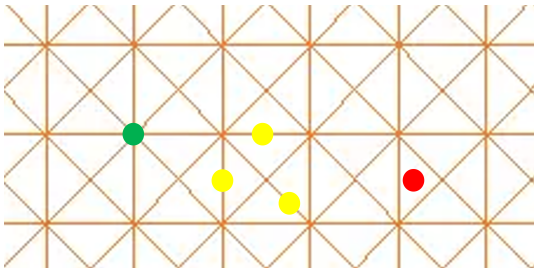


Illustration 3–6 Green: grid point; yellow: partially dangling; red: fully dangling

When we try to expand a fully dangling point to a diagonal edge, the intersection position coordinates may lie in between two consecutive fixed point values and cannot be represented in our formulation. This is caused by an odd expansion stride. To fix this, we enforce the expansion stride to be even, and any nodes in the frontier to have an (even, even) coordinate.

3.3.4.4 Obstacle-Adaptive Radial Expansion

When an expansion meets an obstacle in the way, a binary search will be conducted to find the longest collision-free distance, and the resulting point will be pushed to the frontier.

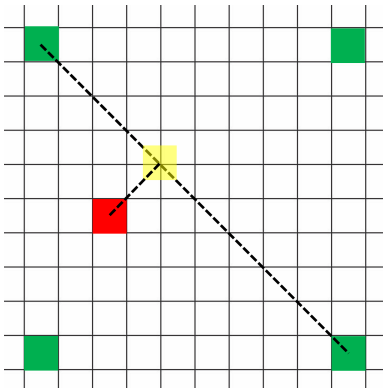


Illustration 3–7 The problem with diagonal intersection

We call it a “leaning off-grid point”.

When the “leaning off-grid point” is popped from frontier and ready for expansion, it will first check whether it is leaning against an obstacle. If so, it is restricted to only expand on the radial direction of the obstacle. This will help reduce the number of off-grid points and guide them to merge with grid points later. See figure 3–8.

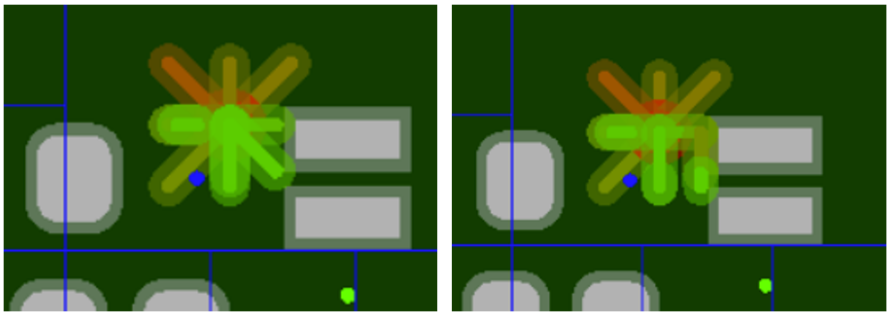


Illustration 3–8 Left: expansions restricted by the obstacles; right: “leaning nodes” only expand radially with respect to obstacles

3.3.4.5 Via Placement

Since via diameters are often much larger than trace widths, it is not valid to drill directly at the position of a “leaning point”. To make things simpler, we only attempt to drill vias on grid points.

3.3.5 A* Cost Function Design

The algorithm uses a sophisticated cost function that accounts for multiple physical factors:

$$f(n) = g(n) + h(n) \tag{3–1}$$

Where:

- $g(n)$ represents the actual path cost including cumulative trace length and via transition penalties
- $h(n)$ uses octile distance scaled by an estimation coefficient:

$$h(n) = \text{octile\_distance}(n, \text{goal}) \times \text{ESTIMATE\_COEFFICIENT} \tag{3–2}$$

For stability, ESTIMATE_COEFFICIENT is often set to 1, but for a more aggressive directed search, it can be set to more than 1.

The cost function ensures admissible heuristic while accounting for the true costs of PCB trace routing.

3.3.6 A* Visualization

Figure 3–9 visualizes the frontier nodes of the A* algorithm in the middle of path finding.

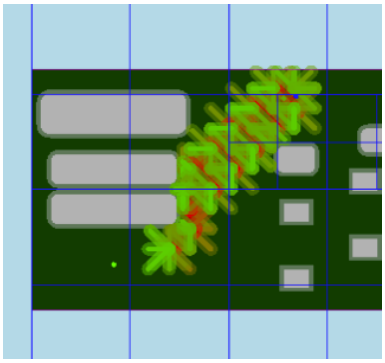


Illustration 3–9 The frontier nodes in the A* algorithm. The green segments represents nodes with low cost, while the red segments represents nodes with high cost.

3.3.7 A* Full Pseudo algorithm

The following is the full pseudo algorithm for the hybrid grid-shape based A* search.

Algorithm 3–1 Enhanced A* Routing Algorithm

```
function A*_ROUTING(start_node, end_node)
    frontier ← new BinaryHeap()                                  ▷ Min-heap priority queue
    frontier.push(start_node)
    visited ← new HashSet()
    while !frontier.empty() do
        current_node ← frontier.pop()
        if current_node.position == end then
            trace_path ← current_node.to_trace_path()
            trace_path ← optimize_path(trace_path)                          ▷ Post-processing
            return trace_path
        end if
        if visited.contains(current_node) then
            continue                                          ▷ Skip already explored nodes
        end if
        visited.insert(current_node)
        if is_aligned_with_end(current_node) then
            if !check_collision(current_node, end_node) then
                frontier.push(end_node)                                  ▷ Approach to end
            end if
        end if
        if is_grid_point(current_node) then
            vias = get_valid_vias(current_node)
            frontier.push(via ∈ vias)                          ▷ Multi-layer via exploration
        end if
                                                               ▷ Hybrid expansion strategy
        for node ∈ to_grids(current_node) ∪ along_obstacles(current_node) do
            if check_collision(current_node, node) then
                node ← binary_node_search(current_node, node)
                                                               ▷ Find furthest safe point
            end if
            frontier.push(node)
            node ← get_intersection_with_end_alignments(current_node, node)
            frontier.push(node)                          ▷ Add intermediate alignment points
        end for
        if no valid movement found then
            Try all directions until finding a valid movement
        end if
    end while
    return none                                          ▷ No path exists under constraints
end function
```

3.3.8 Algorithmic Components

- **is_grid_point(p)**: Verifies whether point p is on the grid point.
- **check_collision(a,b)**: Detects the collisions between obstacles and a segment with a as the start node and b as the end node.
- **is_aligned_with_end(node)**: Verifies whether the current node is aligned with the goal in one of the eight directions.
- **get_valid_vias(node)**: Computes all legal layer transitions considering maximum via stack height constraints, layer-specific design rules, and obstacle clearance requirements.
- **to_grid(node)**: Computes feasible grid points within routing constraints the node can get to.
- **along_obstacles(node)**: Computes accessible positions adhering to obstacle contours while maintaining minimum clearance.
- **binary_node_search(start,end)**: Implements adaptive step sizing by starting with full segment length, performing binary search for collision boundaries when detected, and returning the furthest valid sub-segment.
- **optimize_path(path)**: Performs geometric optimization on routing paths to enhance smoothness through segment merging, minimize inter-path spacing while maintaining design rules, and eliminate acute angles to improve manufacturability and signal integrity.

3.3.9 Path Post-Processing Optimization

The post-processing stage refines the initial A*-generated PCB trace paths through an iterative optimization pipeline (Algorithm 3–2). The algorithm applies four carefully designed adjustments that work together to enhance path quality while meeting all design requirements. Based on geometric simplification principles, each modification targets particular weaknesses in the paths through precise, localized changes. The pipeline's iterative nature allows continuous improvements, as each round of optimization may create new opportunities for further refinements.

Key optimizations include:

1. collinear segment merging

Algorithm 3–2 Trace Path Refinement

```
1: function OPTIMIZE_PATH(path)
2:     opt_path ← path                          ▷ Initialize with input path
3:     merge_colinear_segments(opt_path)
4:     repeat
5:         parallel_shift(opt_path)
6:         optimize_corners(opt_path)
7:         split_right_and_acute_angles(opt_path)
8:     until no further improvements
9:     merge_colinear_segments(opt_path)
10:    return opt_path
11: end function
```

2. parallel trace adjustment
3. convex corner refinement
4. right-angle and acute-angle mitigation

Example implementations are illustrated in figure 3–10. The optimization loop terminates when no further transformations can be applied.

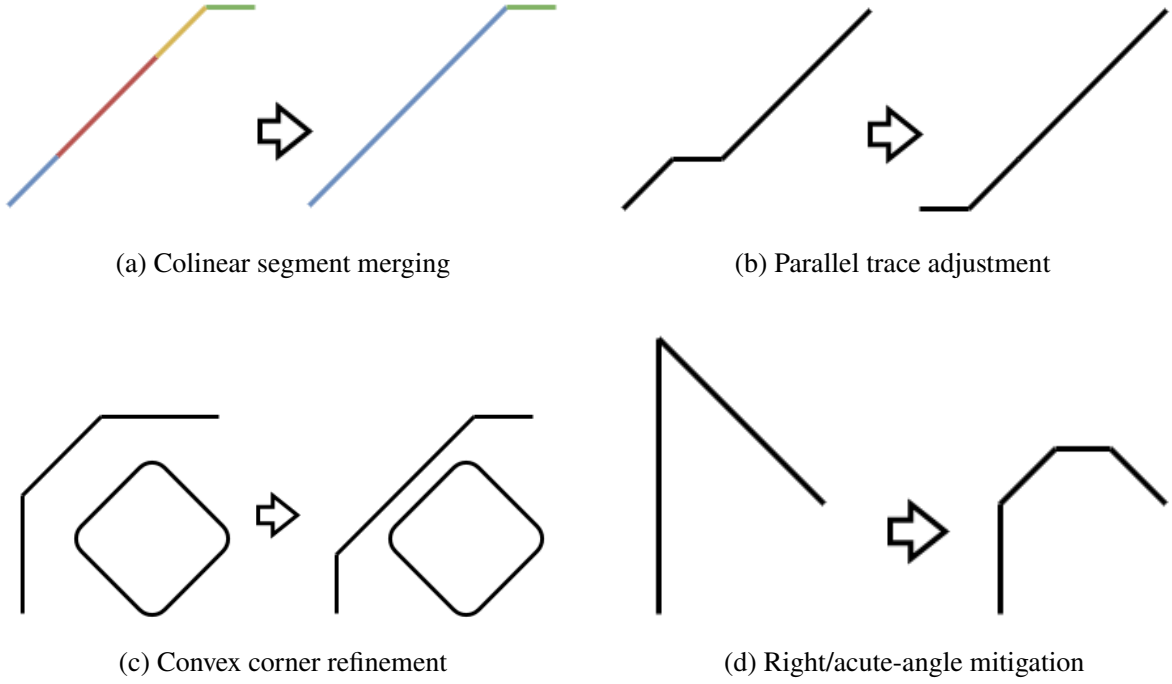
Collinear segment merging consolidates all contiguous collinear segments into a single segment connecting the first segment's start node and the last segment's end node, effectively simplifying the path geometry.

Parallel trace adjustment identifies all parallel segment pairs separated by a single differently-oriented segment. Each segment in the pair attempts to shift toward its counterpart. When obstacles prevent complete alignment, the system implements a partial shift that maintains maximum clearance for subsequent routing operations.

Convex corner refinement activates when three consecutive segments form a right or left turn. The middle segment extends toward the vertex until either contacting an obstacle or reaching the adjacent segments' endpoints, thereby smoothing the corner geometry.

Angle mitigation eliminates all right and acute angles due to their adverse effects on PCB trace connectivity. The process converts right angles into two obtuse angles and acute angles into three obtuse angles.

The complete pipeline achieves significant improvements in trace quality metrics including total path length reduction, elimination of sharp angles, and enhanced manufacturability - all critical factors for high-performance PCB designs.



(a) Colinear segment merging

(b) Parallel trace adjustment

(c) Convex corner refinement

(d) Right/acute-angle mitigation

Illustration 3–10 Four Key Path Optimizations

3.4 The Fallback Naive Backtrack Algorithm

Backtrack algorithm is one of the simplest ideas for solving constraint satisfaction problems. The backtracking paradigm, originally conceptualized in mathematical logic by D.H. Lehmer[12], was later formalized for algorithmic use by Walker and Golomb[13].

The PCB routing problem can be formulated as a constraint satisfaction problem where the constraints can only be determined through exploration.

The idea of adapting backtrack algorithm to PCB routing is simple: we fix one trace at a time, and if we encounter a connection with no possible solution, we recall the last fix and try the alternatives. Essentially, the backtrack algorithm iterates through all the permutations of the connection fixing order and stops at the first one that makes it to the end.

3.4.1 Heuristic for the Backtrack Algorithm

To avoid adopting less promising routing order at the beginning that results in a lot of failed branches, we can assign the algorithm an initial heuristic. We use the octile distance of each connection as the heuristic and let the algorithm adhere to the heuristic first and then explore its neighboring combinations.

3.4.2 Trace Caching for the Backtrack Algorithm

During execution, the algorithm first attempts to reuse cached valid paths for each connection. On cache misses, it dynamically invokes A* path-finding with incremental obstacle management –accounting for previously fixed traces, other nets' pads, and board boundaries.

When encountering unroutable connections, the system backtracks by discarding the most recent trace and exploring alternatives. The search terminates when either all connections are successfully routed, or all candidate combinations are exhausted without finding a valid configuration.

3.4.3 The Pseudo Algorithm

Algorithm 3–3 Naive Backtrack PCB Routing

```
1: function BACKTRACK(trace_cache, alternative_connections)
2:     Initialize backtrack_stack
3:     root_node←init_node(alternative_connections)
4:     backtrack_stack.push(root_node)
5:     while backtrack_stack not empty do
6:         top_node←backtrack_stack.top()
7:         if top_node.alternative_ connections is empty then
8:             if top_node.failed_connections is empty then
9:                 solution← construct_solution(top_node)
10:                return solution
11:            else
12:                return error
13:            end if
14:        end if
15:        connection←get_current_connection(top_node.alternative_connections)
16:        trace_path←get_trace_from_cache(trace_cache,connection)
17:        if trace_path is none then
18:            Run A* to find trace_path
19:            if no trace_path found then
20:                backtrack_stack.pop()
21:                continue
22:            end if
23:            trace_cache.push(trace_path)
24:        end if
25:        new_node←update_node(top_node,connection,trace_path)
26:        backtrack_stack.push(new_node)
27:    end while
28:    return none
29: end function
```

3.5 Implementing the Parser

Specctra Design (DSN) files and Specctra Session (SES) files serve complementary roles in the PCB design workflow[14]. The conversion between these formats serves a key purpose: The DSN file contains all the necessary routing information that our system needs to process. After generating the routing solution, we save it as an SES file. This SES file can then be

easily imported back into PCB design software like KiCad, allowing designers to view and work with the final routed board layout. The complete workflow of our parser is shown in figure 3–11.

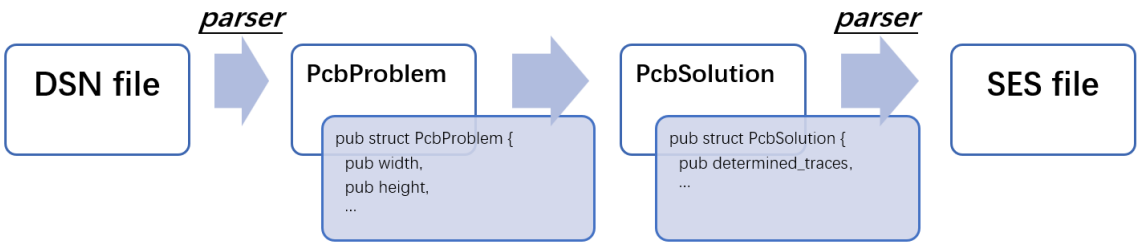


Illustration 3–11 Parser workflow

3.5.1 Parsing Pcb Problem from DSN files

The DSN file records the complete board specification including component information, padstack placements, network connections, and design constraints. The process of parsing PcbProblem struct from DSN file is shown in Algorithm 3–4.

The conversion pipeline transforms Spectra DSN files into routable PCBProblem instances through three coordinated phases. First, the parse_s_expr_to_struct decomposes the DSN structure into a typed DsnStruct. The parser identifies key blocks such as network through pattern matching. At this stage, DsnStruct merely stores the raw data, such as the relative position of the padstack to the component, and the boundaries are recorded in the form of a vector of FloatVec2.

The dsnToDisplay then maps DsnStruct to the unified DisplayFormat representation. In this function, data are transformed to a solvable format. The width, height, and center fields are computed from DsnStruct.boundary. The absolute position of padstacks are computed by rotation matrix and displacement matrix. The resulting global pad coordinates are stored in the DisplayNetInfo structure alongside inherited netclass properties, forming a complete routing target specification.

Finally, the Converter allows users to make custom settings from ExtraInfo. User settings have more privilege over all other data. For nets with designated sources in ExtraInfo, star topologies extend from the forced source pad. When source pads are unspecified, a minimum spanning tree (MST) connects all pads within each net, minimizing total wirelength. Because

Algorithm 3–4 DSN to PCBProblem Conversion

Require: DSN file content as string
Ensure: PcbProblem structure or error message

```
function PARSE_END_TO_END(dsnFileContent)
    sExpr ← parse_dsn_to_s_expr(dsnFileContent)
    if sExpr is Err then
        return Err("Failed to parse DSN: " + sExpr.err)
    end if
    dsnStruct ← parse_s_expr_to_struct(sExpr)
    if dsnStruct is Err then
        return Err(dsnStruct.err)
    end if
    displayFormat ← dsn_to_display(dsnStruct)
    if displayFormat is Err then
        return Err(displayFormat.err)
    end if
    extraInfo ← ExtraInfo{net_name_to_source_pad : HashMap::new()}
    pcbProblem ← Converter::convert(displayFormat, extraInfo)
    if pcbProblem is Err then
        return Err(pcbProblem.err)
    end if
    return Ok(pcbProblem)
end function
```

of the time limit, these features are implemented in the parser but not yet implemented in the routing procedure.

The converted PcbProblem serves as the key fundamental support for subsequent routing optimization and design verification processes.

3.5.2 Parsing Pcb Solution to SES files

The SES file records the finalized routing solutions and simulation parameters generated by external auto-routing tools. It can be imported into design tools to visualize our wiring results, facilitating Design Rule Check and allowing users to make their own modifications to the wiring. The process of SES file generation is shown as Algorithm 3–5.

First, the components and placements are copied from DsnStruct. These parameters rewrite to SES file by generate_placement to generate PCB board.

Algorithm 3–5 SES File Generation

```
1: function GENERATE_PLACEMENT(dsn)
2:     Write placement resolution
3:     for each component in dsn.placement do
4:         for each instance in component do
5:             Write instance: reference, position, layer, rotation
6:         end for
7:     end for
8:     return placement section
9: end function
10: function GENERATE_NETWORK(dsn, solution, layers, vias)
11:     Group traces by net name
12:     for each net in traces do
13:         via_name ← via name for current net
14:         for each trace in net do
15:             for each via in trace.vias do
16:                 Write via: name, position
17:             end for
18:             for each segment in trace do
19:                 Write wire: layer, width, start, end
20:             end for
21:         end for
22:     end for
23:     return network section
24: end function
25: function WRITE_SES(dsn, solution, output)
26:     layers ← get layer names from dsn
27:     vias ← extract via info from dsn.library
28:     Write session header
29:     Write GENERATE_PLACEMENT(dsn)
30:     Write via library: GENERATE_VIA_ENTRIES(vias, layers)
31:     Write GENERATE_NETWORK(dsn, solution, layers, vias)
32:     Write file footer
33: end function
```

For routing data, the solution organizes determined_traces by net and reconstructs their connectivity. For via generation, the system cross-references net classes in the original DSN to select appropriate padstacks. For each segment in the trace, their start and end points, that is, the turning points of the trace, are scaled and recorded under the corresponding net. , The scale_down_factor derived from DSN's resolution unit and value is applied to convert floating-point coordinates back to integer-based system.

The generated SES file provides a standardized interface that can be directly imported into PCB design tools like KiCad or Cadence Allegro for visualization, verification, and manufacturing preparation.

Chapter 4 Outcome

4.1 Deliverables

We integrated our algorithm and visualization implementations as a cross-platform desktop application. The source code is in appendix 1.

4.1.1 User Interface

4.1.1.1 Home Page

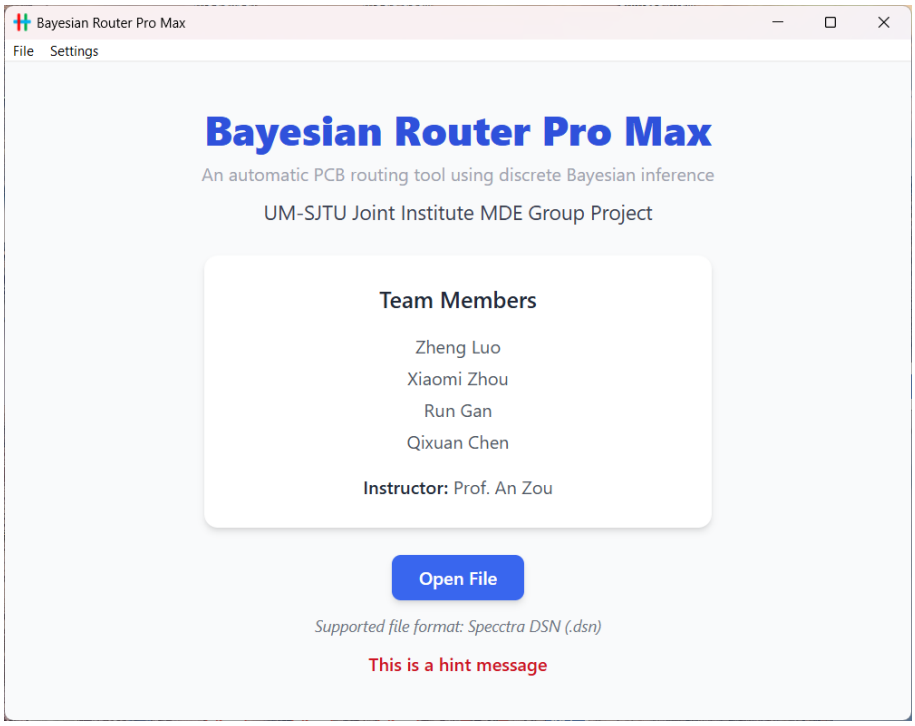


Illustration 4–1 The home page of our desktop application

4.1.1.2 PCB Page

On the left, there is a canvas that displays the PCB problem and the routing process.

On the right, there are buttons that controls the routing procedure, including the start / pause button, step in, step over and step out button that controls the granularity of the routing procedure. There is also a "view stats" button and a "save result" button that navigates to the

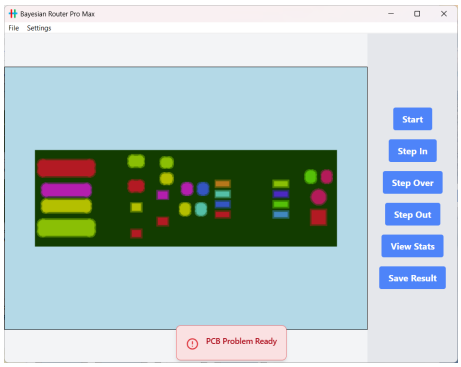


Illustration 4–2 The PCB page of our desktop application

statistics page and save the resulting Specctra Session file to the local storage respectively.

4.1.1.3 Settings Page

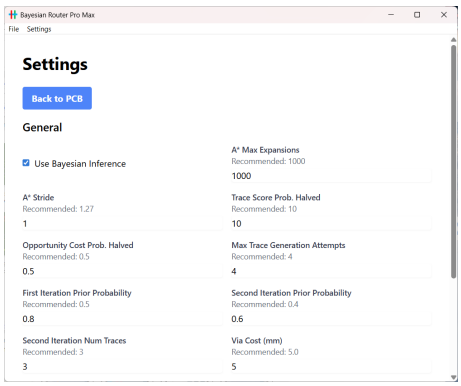


Illustration 4–3 The settings page - general settings

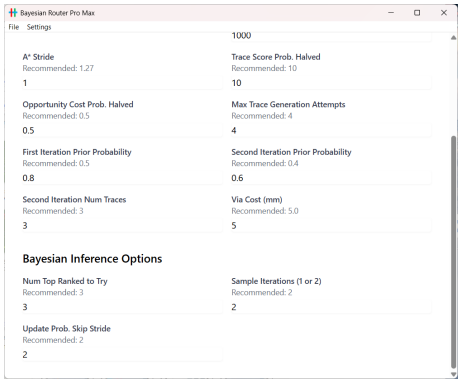


Illustration 4–4 The settings page - Bayesian inference options

The settings page includes all the tunable hyperparameters. Table 4–1 briefly describes

their meanings.

Hyperparameter Name	Description
Use Bayesian Inference	Use our proposed Bayesian inference algorithm or just the naive backtrack algorithm
A* Max Expansions	The maximum number of expansions allowed for one A* path-finding attempt.
A* Stride	The distance between the expanded point and the original point (virtual grid width).
Trace Score Prob. Halved	The amount of increase in "trace score" for a trace's posterior probability to be halved.
Opportunity Cost Prob. Halved	The amount of increase in "opportunity cost" for a trace's posterior probability to be halved.
Max Trace Generation Attempts	The maximum number of attempts allowed to generate traces before a connection reaches its maximum capacity.
First Iteration Prior Probability	The prior probability assigned to all traces that are generated in the first sampling iteration.
Second Iteration Prior Probability	The prior probability assigned to all traces that are generated in the second sampling iteration.
Second Iteration Num Traces	The maximum number of traces that can be generated in the second sampling iteration.
Via Cost (mm)	A length in mm such that the cost of a via is equivalent to the cost of a trace with this length.
Num Top Ranked to Try	In the Bayesian inference progressing procedure, the number of top ranked traces to try before declaring a dead-end.
Sample Iterations (1 or 2)	Number of iterations to sample probabilistic traces in the model updating phase.
Update Prob. Skip Stride	Number of trace fixes that can be made before having to update the probabilistic model.

Table 4–1 The descriptions of hyperparameters in the settings.

4.1.1.4 Stats Page

In the statistics page, statistics like routing time, total trace length and via count are displayed for the last routing task.

It also displays the number of path-finding (A*) calls used by the Bayesian inference al-

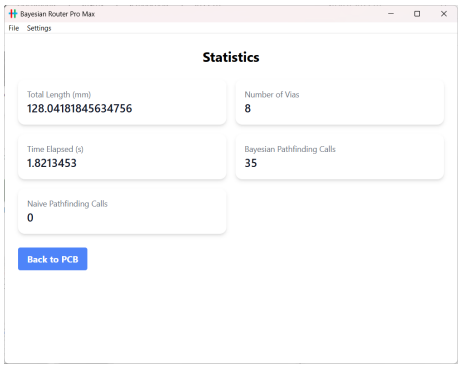


Illustration 4–5 The statistics page

gorithm and the naive backtrack algorithm that helps us visualize the effectiveness of the Bayesian inference algorithm in reducing the number of path-finding calls (thus reduce computation time) compared with the naive backtrack algorithm.

4.1.2 Workflow

4.1.2.1 Open File

We can open a PCB file from either the home page or the top menu. Only Specctra DSN file format is supported.

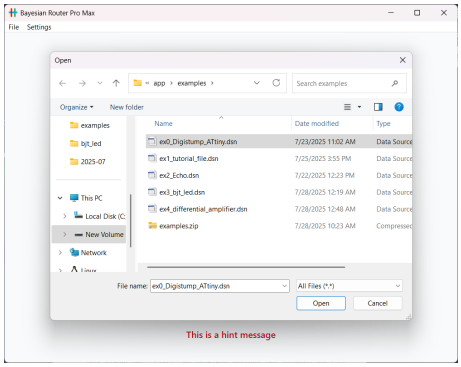


Illustration 4–6 Open a PCB file

4.1.2.2 Start Routing

The routing control system is initialized with the finest granularity in solving visualization. Click “Start”, “Step Over” or “Step Out” to start the routing procedure.

Clicking “Pause” will reset the solving visualization to the finest granularity.

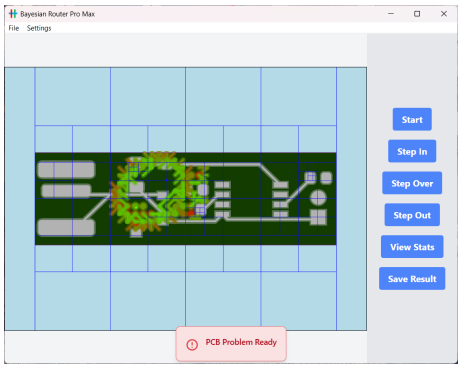


Illustration 4–7 Routing in progress

4.1.2.3 Finish Routing and Save File

When the autorouter finishes running, it will display a hint message and display the final routed PCB. Then you can choose to view statistics and save the result as a Specctra Session file.

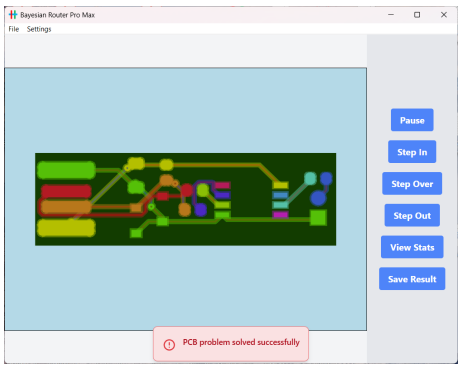


Illustration 4–8 Autorouting complete

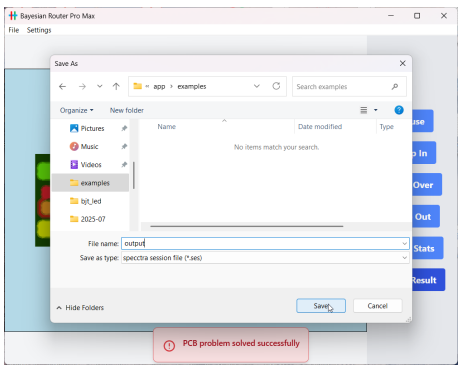


Illustration 4–9 Save the Specctra Session file to the local storage

4.2 Performance Measure

4.2.1 Choose *FreeRouting* as the Benchmark Target

FreeRouting is a widely used, free, and open-source PCB autorouting tool that has become a de facto benchmark in academic and practical evaluations of routing algorithms. Its popularity stems from its robust performance, support for various design formats, and compatibility with major EDA tools, making it a reliable and accessible reference point for comparison. As an open-source project, *FreeRouting* offers transparency in algorithm design and output behavior, allowing for fair and reproducible benchmarking. These characteristics make it an ideal standard against which to evaluate the effectiveness and efficiency of new PCB autorouting algorithms.

4.2.2 Experimental Setup

4.2.2.1 Instance Selection

We collected a variety of PCB design instances online. Due to the suboptimal performance of our customized A* algorithm on complex boards, we mainly focus on small to medium sized examples. Most of the test cases were selected from projects available on the official *KiCad* website: https://www.kicad.org/made-with-kicad/.

4.2.2.2 Execution Environment

All experiments were conducted on a laptop running Windows 10, equipped with an Intel i7-7500U processor, 16 GB RAM, and an NVIDIA 940MX GPU.

4.2.2.3 Dataset

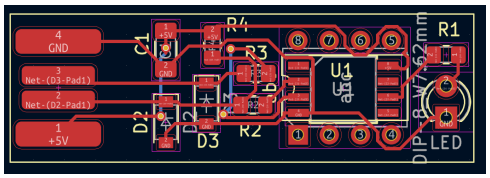
We use the 5 PCB files in figure 4–10 for testing. All of them are two layers.

4.2.2.4 Evaluation Metrics

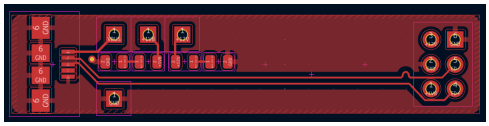
We evaluate the performance of an autorouting algorithm based on the following metrics:

1. Whether managed to conduct a complete routing within tolerable amount of time. For small to medium sized PCB problem, we set the threshold to 1 minute.

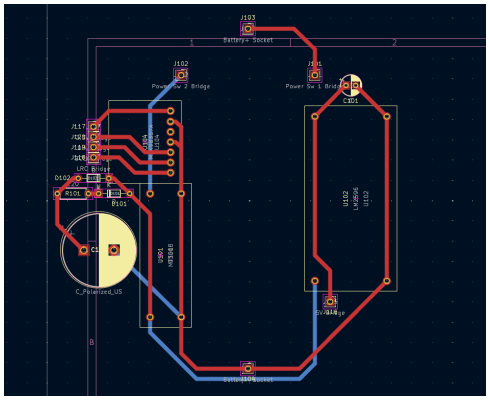
2. Execution time. Our desktop application is designed to output the execution time as one of its outputs, but for the *FreeRouting* software, we failed to find a command line interface option for it, so we used a stopwatch to measure the runtime. Although this approach is not



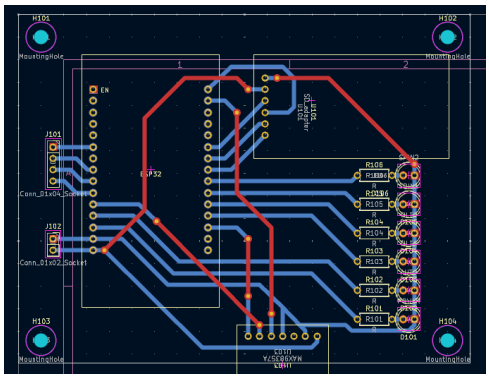
(a) Digistump ATtiny
Source: https://github.com/Hasona57/Digistump-ATtiny



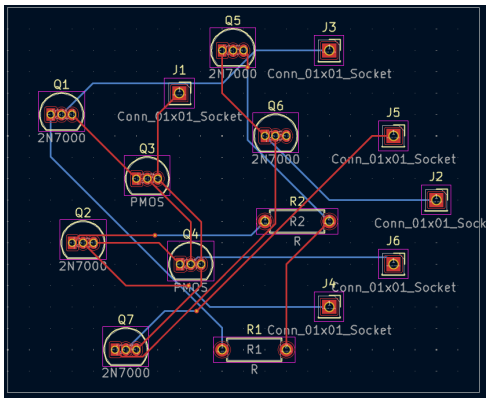
(b) Echo Debug
Source: https://gitlab.com/phodina/echo-debug-gen3



(c) Music Box
Source: https://github.com/LuoZheng2002/board_down



(d) Ping Pong
Source: https://github.com/LuoZheng2002/ping_pong_esp32



(e) Differential Amplifier
Made by our team (source in project repository in appendix 1)

Illustration 4–10 PCB designs for testing

perfectly rigorous and may introduce slight errors, we believe it does not significantly affect the evaluation.

3. Total trace width and via count. The number of vias and whether the routing was successfully completed can be directly obtained from *KiCad*' s built-in statistics. As for total wire length, *KiCad* does not explicitly provide this metric. However, it offers a Python API, and we wrote a simple script to extract the total wire length from the board files.

4.2.2.5 Testing Procedure

We ran our algorithm and *FreeRouting* 10 times each on five chosen boards, record the raw data, and calculate statistics like mean execution time and mean total trace length. For gathering the total trace length information for *FreeRouting*, we wrote a Python script, as shown in figure 4–11.

The raw experimental results are attached in Appendix 2, where each table corresponds to a specific instance.

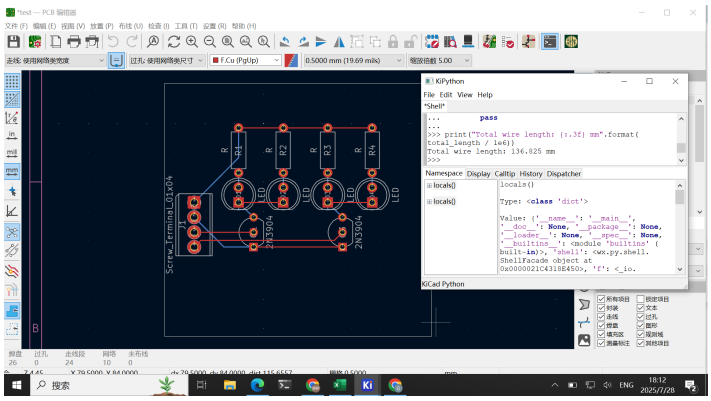


Illustration 4–11 The Python script output for gathering total trace width information

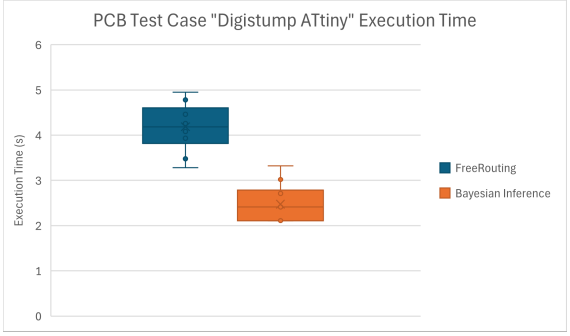
4.3 Results

4.3.1 Execution Time

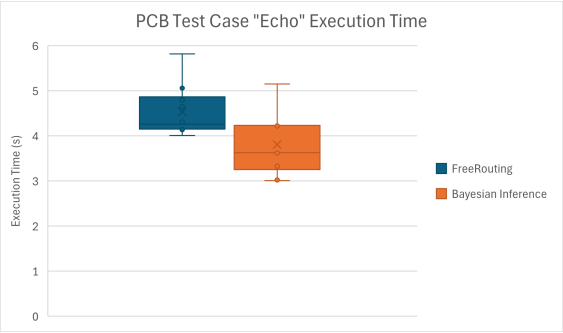
Figure 4–12 are the execution time box plots of both *FreeRouting* and our algorithm on each of the five test cases.

For most cases, our algorithm consistently achieves shorter execution times compared to *FreeRouting*. More significantly, the observed reduction in box height in our results indicates greater stability across multiple runs, suggesting more predictable performance behavior.

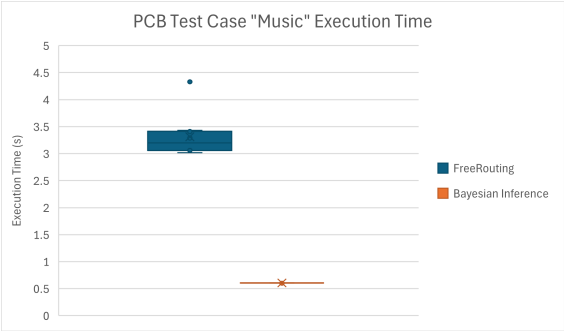
While case 5 presents an exception where our algorithm takes longer to run than *FreeRouting*, this demonstrates that our algorithm can be influenced by case-specific complexities. Statistical analysis confirms that our algorithm maintains superior overall performance metrics when considering the complete test suite.



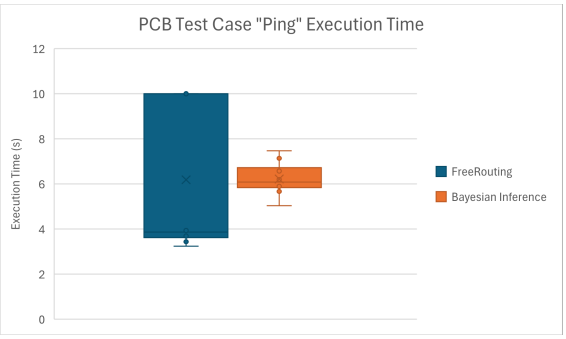
(a) Case 1



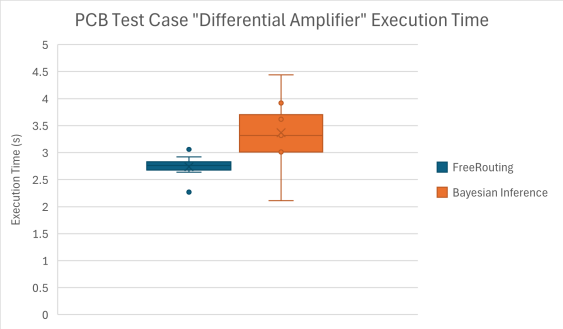
(b) Case 2



(c) Case 3



(d) Case 4



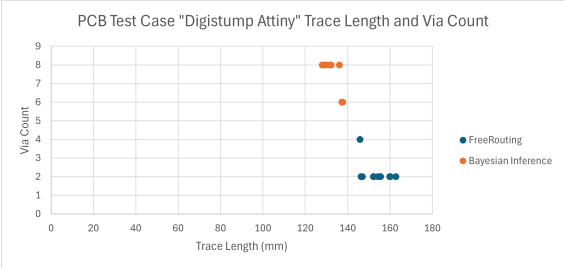
(e) Case 5

Illustration 4–12 Execution time comparison across five test cases

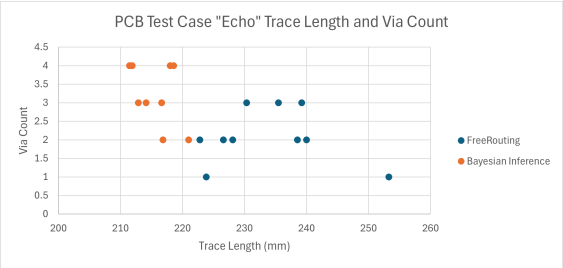
4.3.2 Total Trace Length and Via Count

Since it can be controversial about how much a via costs with respect to trace length, we display the comparison result using 2D scatter plots in figure 4–13.

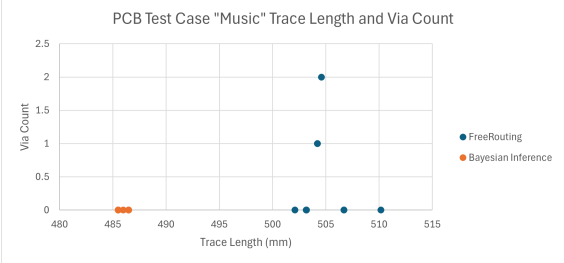
In most cases, our algorithm produces shorter trace length, but at the same time, it generates more vias. Generally, the trace lengths of our algorithm distribute more closely compared to *FreeRouting*, which indicates that it generates traces with more stable lengths. How-



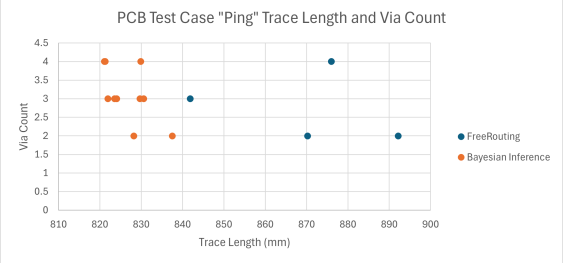
(a) Case 1



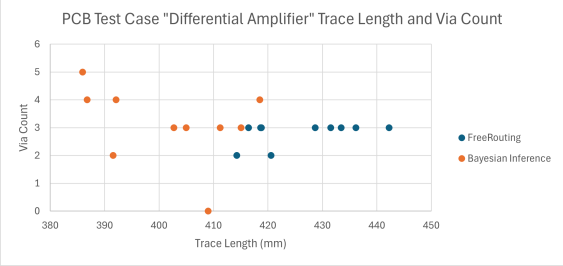
(b) Case 2



(c) Case 3



(d) Case 4



(e) Case 5

Illustration 4–13 Trace length and via count comparison across five test cases

ever, our algorithm usually leads to higher via count. This occurs because our algorithm prioritizes minimal trace length.

In some cases like figure 4–13d, we manage to achieve shorter trace lengths while maintaining the same number of via count compared with *FreeRouting*.

4.3.3 Number of Path Finding Algorithm Calls

Apart from the direct comparison of execution time, the number of path finding algorithm calls is also a vital metric indicating how good our inference algorithm is regardless of the absolute performance of the path finding algorithm. We obtained the following statistics using both the Bayesian inference algorithm and the naive backtrack algorithm from our

desktop application.

Table 4–2 illustrates the comparison of the number of path-finding calls for the Bayesian inference algorithm and the naive backtrack algorithm.

PCB Test Case	Bayesian Inference Average Num Path-Finding Calls	Naive Backtrack Average Num Path-Finding Calls
“Digistump ATtiny”	41.4	24.9
“Echo”	40.3	23.5
“Music”	33.6	26.1
“Ping”	87.2	45.1
“Differential Amplifier”	71.0	28.0

Table 4–2 Comparing the number of path-finding algorithm calls for the Bayesian inference algorithm and the naive backtrack algorithm

We can see from the table that in general, the Bayesian inference algorithm has a larger number of path-finding algorithm calls than the naive backtrack algorithm. This is not surprising since we are mainly testing on simple multi-layer PCBs, which can be solved easily with the naive backtrack algorithm without involving much trial and error. In comparison, the overhead of the Bayesian inference algorithm is larger since it involves sampling traces for gathering information, rather than directly fixing it as a part of the solution.

Although the result may not look promising at the first glance, the good news is that the number of path finding calls for Bayesian inference is only a couple of times larger than the naive backtrack algorithm to start with. When the problem gets more complex, the expected number of path finding calls will surge for the naive backtrack algorithm since its worst case time complexity is O(n!) with respect to the number of connections to route. In comparison, our Bayesian inference algorithm can invest a linear amount of overhead for a decent heuristic, which is likely to save much more time in trials and errors in the wrong direction, compared with the overhead invested. Therefore, we think that our Bayesian inference idea is still promising in solving more complex and highly constrained problem, given we have a more efficient path-finding algorithm compared with the current customized A* algorithm.

4.4 Analysis

We evaluated our self-developed auto-router on the selected five PCB board instances. The results indicate that our tool exhibits potential advantages in several aspects.

First, in terms of execution time, our auto-router achieved a shorter average runtime. Moreover, the corresponding boxplots indicate smaller interquartile ranges, implying a lower standard deviation and hence more stable performance.

Second, regarding trace length and vias, the two tools exhibited different tendencies. Ideally, the goal is to achieve both shorter trace length and fewer vias simultaneously. However, the statistics show that our auto-router tends to produce solutions with shorter trace lengths but slightly more vias, whereas *FreeRouting* tends to find solutions with fewer vias but longer trace lengths.

Finally, we examined the number of path-finding function calls. The result indicates that our Bayesian inference algorithm generally requires more path-finding calls than the naive backtrack algorithm. This demonstrates a severe overhead of our Bayesian inference algorithm on simple cases. However, as the routing problem gets more complex, the marginal computation cost of the naive backtrack algorithm may become significant larger than that of our Bayesian inference algorithm, making our algorithm more promising in dealing with more complex problems.

Chapter 5 Conclusions

5.1 Main Conclusions

This thesis has made contributions to automated PCB routing by developing a novel system that combines discrete Bayesian inference with a grid-shape hybrid path-finding algorithm. The research has successfully demonstrated that probabilistic methods can effectively guide routing decisions while maintaining computational efficiency.

At the core of this work lies the discrete Bayesian inference engine, which provides an intelligent probability update mechanism for trace routing through parallel sampling and iterative scoring. This probabilistic framework is complemented by an advanced path-finding solution that integrates grid-shape adapted A* algorithm with optimized border handling and collision detection capabilities. The complete system implementation has been realized through a cross-platform desktop application that bridges theoretical research with practical application, featuring both step-wise debugging and continuous production modes alongside comprehensive visualization tools.

Experimental validation against the *FreeRouting* benchmark has confirmed the system's advantages, showing faster routing times and higher stability for simple boards while maintaining comparable solution quality. The results collectively establish that lightweight Bayesian methods can serve as a viable alternative to traditional rule-based approaches in PCB routing applications.

5.2 Research Outlook

This approach opens several promising directions for future work. Algorithmic enhancements could incorporate machine learning to improve probability estimation. Practical next steps include developing plugins for commercial tools and creating cloud-based solutions capable of handling large designs. Future research might investigate learning from historical routing data and adapting to designer feedback. Overall, this work lays a strong foundation for smarter routing tools that balance performance and quality. Moving forward, the priority should be to ensure the method performs well on complex, real-world designs and integrates

seamlessly into existing design workflows.

References

[1] YIN S, JIN M, CHEN G, et al. Unet-Astar: A Deep Learning-Based Fast Routing Algorithm for Unified PCB Routing[J]. IEEE Access, 2023, 11: 113712-113725.

[2] RONNEBERGER O, FISCHER P, BROX T. U-net: Convolutional networks for biomedical image segmentation[C]//Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18. 2015: 234-241.

[3] HART P E, NILSSON N J, RAPHAEL B. A formal basis for the heuristic determination of minimum cost paths[J]. IEEE transactions on Systems Science and Cybernetics, 1968, 4(2): 100-107.

[4] LEE V, NGUYEN M, ELZEINY L, et al. Chip Placement with Diffusion Models[EB/OL]. [2025-05-21]. https://arxiv.org/abs/2407.12282.

[5] SCARSELLI F, GORI M, TSOI A C, et al. The graph neural network model[J]. IEEE transactions on neural networks, 2008, 20(1): 61-80.

[6] CHEN J, TU J, WANG H, et al. GPCB Routing: Generative Pretrained Transformers-Based Printed Circuit Board Routing Method[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2025, 44(4): 1420-1433.

[7] LIN T C, MERRILL D, WU Y Y, et al. A Unified Printed Circuit Board Routing Algorithm With Complicated Constraints and Differential Pairs[C]//2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC). 2021: 170-175.

[8] OZDAL M, WONG M. Simultaneous escape routing and layer assignment for dense PCBs[C]//IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004. 2004: 822-829.

[9] BAYES T. An essay towards solving a problem in the doctrine of chances[J]. Biometrika, 1958, 45(3-4): 296-315.

[10] KINGMA D P, WELLING M. Auto-Encoding Variational Bayes[EB/OL]. [2025-07-01]. https://arxiv.org/abs/1312.6114.

[11] HO J, JAIN A, ABBEEL P. Denoising diffusion probabilistic models[J]. Advances in neural information processing systems, 2020, 33: 6840-6851.

[12] LEHMER D H. Inverse combinatorial problems[J]. Bulletin of the American Mathematical Society, 1954, 60(6): 596-598.

[13] GOLOMB S W, BAUMERT L D. Backtrack Programming[J]. J. ACM, 1965, 12(4): 516-524.

[14] Cadence Design Systems, Inc. SPECCTRA® Design Language Reference[A]. 10.0. Document No. SPECCTRA-DLR-10.0. 555 River Oaks Parkway, San Jose, CA 95134, USA: Cadence Design Systems, Inc., 2000.

Source Code Appendix 1

https://github.com/LuoZheng2002/bayesian_router_pro_max

Experimental Results Appendix 2

2.1 FreeRouting Statistics

Finished	Exec. time	Wire length	Vias
TRUE	4.55 s	152.34 mm	2
TRUE	4.78 s	154.38 mm	2
TRUE	4.26 s	159.94 mm	2
TRUE	4.08 s	146.41 mm	2
TRUE	4.95 s	155.60 mm	2
TRUE	3.93 s	162.83 mm	2
TRUE	4.46 s	152.23 mm	2
TRUE	3.48 s	145.83 mm	4
TRUE	4.11 s	146.96 mm	2
TRUE	3.28 s	160.20 mm	2

Table 2–1 PCB Test Case 1 “Digistump ATtiny” using FreeRouting

Finished	Exec. time	Wire length	Vias
TRUE	4.65 s	230.34 mm	3
TRUE	4.80 s	240.01 mm	2
TRUE	4.19 s	239.23 mm	3
TRUE	4.16 s	228.10 mm	2
TRUE	4.01 s	222.80 mm	2
TRUE	4.21 s	223.84 mm	1
TRUE	5.82 s	235.48 mm	3
TRUE	4.31 s	226.57 mm	2
TRUE	4.14 s	253.27 mm	1
TRUE	5.06 s	238.54 mm	2

Table 2–2 PCB Test Case 2 “Echo” using FreeRouting

2.2 Bayesian Inference Statistics

2.3 Bayesian Inference and Naive Backtrack Path-Finding Calls

Finished	Exec. time	Wire length	Vias
TRUE	3.02 s	503.17 mm	0
TRUE	3.06 s	503.17 mm	0
TRUE	3.11 s	506.70 mm	0
TRUE	3.31 s	502.10 mm	0
TRUE	3.41 s	503.16 mm	0
TRUE	3.06 s	504.20 mm	1
TRUE	3.43 s	506.70 mm	0
TRUE	3.09 s	504.20 mm	1
TRUE	4.33 s	504.60 mm	2
TRUE	3.29 s	510.16 mm	0

Table 2–3 PCB Test Case 3 “Music Box” using FreeRouting

Finished	Exec. time	Wire length	Vias
TRUE	3.94 s	876.02 mm	4
FALSE	/	/	/
FALSE	/	/	/
FALSE	/	/	/
TRUE	3.79 s	892.18 mm	2
TRUE	3.78 s	841.85 mm	3
TRUE	3.44 s	870.21 mm	2
TRUE	3.24 s	846.20 mm	3
TRUE	3.69 s	919.66 mm	4
FALSE	/	/	/

Table 2–4 PCB Test Case 4 “Ping Pong” using FreeRouting

Finished	Exec. time	Wire length	Vias
TRUE	2.75 s	418.63 m	3
TRUE	2.69 s	442.25 m	3
TRUE	2.92 s	428.60 m	3
TRUE	3.06 s	431.54 m	3
TRUE	2.27 s	418.81 m	3
TRUE	2.69 s	433.41 m	3
TRUE	2.77 s	420.59 m	3
TRUE	2.80 s	416.45 m	3
TRUE	2.64 s	414.26 m	2
TRUE	2.77 s	436.14 m	3

Table 2–5 PCB Test Case 5 “Differential Amplifier” using FreeRouting

Finished	Exec. time	Wire length	Vias
TRUE	2.11 s	137.67 mm	6
TRUE	2.41 s	137.63 mm	6
TRUE	2.11 s	137.26 mm	6
TRUE	3.32 s	131.62 mm	8
TRUE	3.02 s	137.38 mm	6
TRUE	2.11 s	132.14 mm	8
TRUE	2.41 s	128.07 mm	8
TRUE	2.11 s	129.75 mm	8
TRUE	2.41 s	129.57 mm	8
TRUE	2.71 s	136.08 mm	8

Table 2–6 PCB Test Case 1 “Digistump ATtiny” using Bayesian Inference

Finished	Exec. time	Wire length	Vias
TRUE	3.62 s	216.62 mm	3
TRUE	4.22 s	214.14 mm	3
TRUE	5.15 s	218.00 mm	4
TRUE	3.02 s	211.46 mm	4
TRUE	3.33 s	218.56 mm	4
TRUE	4.24 s	221.01 mm	2
TRUE	3.01 s	216.84 mm	2
TRUE	3.62 s	211.46 mm	4
TRUE	4.23 s	212.88 mm	3
TRUE	3.64 s	211.90 mm	4

Table 2–7 PCB Test Case 2 “Echo” using Bayesian Inference

Finished	Exec. time	Wire length	Vias
TRUE	0.61 s	485.51 mm	0
TRUE	0.61 s	485.51 mm	0
TRUE	0.60 s	485.49 mm	0
TRUE	0.60 s	485.49 mm	0
TRUE	0.61 s	485.51 mm	0
TRUE	0.61 s	485.51 mm	0
TRUE	0.61 s	485.51 mm	0
TRUE	0.61 s	485.97 mm	0
TRUE	0.60 s	485.51 mm	0
TRUE	0.60 s	485.48 mm	0

Table 2–8 PCB Test Case 3 “Music Box” using Bayesian Inference

Finished	Exec. time	Wire length	Vias
TRUE	7.47 s	829.90 mm	4
TRUE	6.20 s	829.72 mm	3
TRUE	5.67 s	823.58 mm	3
TRUE	5.96 s	821.08 mm	4
TRUE	5.04 s	828.23 mm	2
TRUE	5.98 s	821.31 mm	4
TRUE	6.58 s	830.63 mm	3
TRUE	7.14 s	824.09 mm	3
TRUE	6.30 s	821.97 mm	3
TRUE	5.90 s	837.51 mm	2

Table 2–9 PCB Test Case 4 “Ping Pong” using Bayesian Inference

Finished	Exec. time	Wire length	Vias
TRUE	3.01 s	392.12 mm	4
TRUE	3.92 s	411.25 mm	3
TRUE	3.62 s	386.80 mm	4
TRUE	4.44 s	391.57 mm	2
TRUE	3.32 s	405.00 mm	3
TRUE	3.32 s	402.70 mm	3
TRUE	3.32 s	415.05 mm	3
TRUE	2.11 s	385.93 mm	5
TRUE	3.63 s	408.99 mm	0
TRUE	3.01 s	418.48 mm	4

Table 2–10 PCB Test Case 5 “Differential Amplifier” using Bayesian Inference

Bayesian Inference Num Path-Finding Calls	Naive Backtrack Num Path-Finding Calls
35	22
46	23
38	23
46	23
46	42
42	23
38	23
49	25
37	23
37	22

Table 2–11 PCB Test Case 1 “Digistump ATtiny” using Bayesian Inference

Bayesian Inference Num Path-Finding Calls	Naive Backtrack Num Path-Finding Calls
39	23
43	23
38	25
40	28
35	23
42	24
43	22
40	22
43	23
40	22

Table 2–12 PCB Test Case 2 “Echo” using Bayesian Inference

Bayesian Inference Num Path-Finding Calls	Naive Backtrack Num Path-Finding Calls
31	26
33	27
36	26
34	26
33	26
33	25
33	25
36	27
33	26
34	27

Table 2–13 PCB Test Case 3 “Music Box” using Bayesian Inference

Bayesian Inference Num Path-Finding Calls	Naive Backtrack Num Path-Finding Calls
92	46
87	46
80	45
86	46
82	44
84	44
87	44
95	43
88	46
91	47

Table 2–14 PCB Test Case 4 “Ping Pong” using Bayesian Inference

Bayesian Inference Num Path-Finding Calls	Naive Backtrack Num Path-Finding Calls
69	29
76	29
76	27
56	27
58	28
76	28
72	30
67	28
75	28
75	26

Table 2–15 PCB Test Case 5 “Differential Amplifier” using Bayesian Inference

Research Projects and Publications during Undergraduate Period

N/A

Acknowledgements

We received valuable guidance throughout this project and would like to express our sincere gratitude to the Joint Institute and our instructor, Dr. An Zou. We are especially thankful to Dr. Zou for his dedicated support and insightful guidance, from helping us with topic selection and research direction to providing feedback during the analysis of our results. We also deeply appreciate the collaborative efforts of every team member—this project would not have been possible without each person's contribution.

AUTOMATIC PCB ROUTING USING DISCRETE BAYESIAN INFERENCE

This thesis proposed an automatic printed circuit board (PCB) routing system based on discrete Bayesian inference, aiming to achieve better trade-offs between runtime performance and routing quality than traditional rule-based approaches. Starting from a motivation rooted in the inefficiencies of existing routing tools, we introduced a probabilistic framework that guides routing decisions through parallel trace sampling and iterative score updates.

The research methodology centers on a Discrete Bayesian Inference algorithm as the core probability update rule for traces. It is supported by an iterative trace sampling mechanism which generates traces for the rule to apply. A trace fixing and progressing mechanism can then select traces according to the probabilistic model generated by them and determine the frequency of updating the model periodically. A dead-end handling strategy is also implemented to deal with situations where our probabilistic model fails to find a solution.

For path-finding, the solution employs a hybrid grid-shape based A* algorithm enhanced with optimized border handling, collision detection, and an adaptive search strategy. The cost function is carefully designed to balance efficiency and solution quality, while post-processing optimizations refine the generated paths. The framework also includes parsers for both input (DSN) and output (SES) file formats, enabling seamless integration with existing PCB design workflows. Together, these components form a cohesive system that addresses complex routing problems through a combination of probabilistic inference and algorithmic path optimization.

The algorithm is then integrated with visualization implementations as a cross-platform desktop application. The application provides a comprehensive workflow supporting: customizable file import, adjustable parameter settings for algorithm tuning, multiple routing modes including both step-wise execution for debugging and continuous operation for production, detailed statistics review for performance analysis, and configurable file export options. This implementation bridges the theoretical framework with practical usability, offering both research-oriented inspection tools and production-ready automation capabilities through an intuitive graphical interface.

Experimental comparisons with the open-source tool FreeRouting show that our approach achieves faster routing times, higher stability, while maintaining comparable routing quality. These results suggest that discrete Bayesian inference, though lightweight, can effectively guide routing decisions with minimal computational overhead.

Overall, this work demonstrates the feasibility of combining probabilistic modeling with classical search techniques in PCB routing, opening up new directions for efficient, scalable, and intelligent design automation. Future work may focus on integrating machine learning, creating cloud-based solutions for large designs, and learning from past routing data.