The Journal of Supercomputing
https://doi.org/10.1007/s11227-024-06338-9

Subgraph matching-based reference placement for printed circuit board designs

Ziran Zhu[1] · Yilin Li[1] · Miaodi Su[2] · Shu Zhang[2] · Haiyuan Su[2] · Yifeng Xiao[3] · Huan He[4] · Jianli Chen[5] · Yao-Wen Chang[6,7]

Accepted: 1 July 2024

Abstract

Reference placement is promising to handle the increasing complexity in printed circuit board (PCB) designs, which aims to find the isomorphism of the placed template in component combination to reuse the placement. In this paper, we convert the netlist information into a graph and then model the reference placement as a subgraph matching problem. Since the state-of-the-art subgraph matching methods usually recursively search the solutions and suffer from high time and memory consumption in large-scale designs, we develop a novel subgraph matching algorithm D2BS with diversity tolerance and improved backtracking to guarantee matching quality and efficiency. The D2BS algorithm is founded on a data structure called the candidate space (CS) structure. We build and filter the candidate set for each query node according to our designed features to construct the CS structure. During the CS optimization process, a graph diversity tolerance strategy is adopted to achieve efficient inexact matching. Then, hierarchical matching is developed to search the template embeddings in the CS structure guided by branch backtracking and matched-node snatching strategies. Based on the industrial PCB designs, experimental results show that D2BS outperforms the state-of-the-art subgraph matching method in matching accuracy and running time.

Keywords Printed circuit board · Placement · Reference placement · Subgraph matching

Preliminary version of this paper was presented at the 2022 ACM/IEEE Design Automation Conference (DAC'22), San Francisco, CA, USA, July 2022 [1].

Extended author information available on the last page of the article

Published online: 06 July 2024

1 Introduction

The printed circuit board (PCB) is a type of board made of insulating material with conductive pathways inscribed on its surface. These pathways allow electrical signals to be routed between various components that are placed onto the board. PCBs play a vital role in the design and manufacturing of modern electronics and have influenced the electronics industry by making it possible to create compact, reliable, and cost-effective devices [2]. Nowadays, PCBs are widely used in electronics, including computers, smartphones, televisions, and many other devices [3]. The design of a PCB is crucial to its function and performance. PCB designers should consider the placement of components, the routing of conductive pathways, and various electrical and thermal characteristics in order to create a functional and reliable board.

PCB placement refers to the process of determining the physical location of components on a PCB. This process is one of the most important stages in the PCB design flow, as the placement of components strongly influences the efficiency, performance, and reliability of PCB production. Several previous works [4–8] have developed automated methods for the PCB placement. The work [4] presents a self-organizing genetic algorithm (SOGA) method for solving the multi-objective placement optimization problem in PCB. The SOGA can be viewed as a cascade of two GAs which consists of two steps fitness evaluation process to ensure that the fitness of selected chromosomes for each iteration process is optimally selected. The work [5] proposes a method for optimizing the thermal placement of heat and non-heat-generating electronic components on a PCB. It uses a genetic algorithm to optimize the maximum temperature of the PCB and the total wirelength between components. The work [6] proposes a new system of optimization design of PCB that combines genetic algorithm and routing Lee algorithm. The genetic algorithm allocates components automatically, and the routing Lee algorithm creates routes between components. The work [7] proposes a heuristic algorithm for isolating the connected fragments of the graph model as building blocks. The developed integrated algorithm based on genetic, evolutionary, and ant search strategies is organized hierarchically. Moreover, the work [8] first proposes a gradient descent-based algorithm to optimize wirelength, component density, and routability. Then, a mixed-integer linear program is developed for local legalization.

The above methods [4–8] utilize various meta-heuristics or analytical algorithms to generate layouts without overlapping while considering various metrics, such as power and thermal characteristics of components, timing, and tidiness [8]. However, these methods incur some drawbacks, such as being computationally expensive, difficulty in handling multi-objective optimization or inability to rotate components, etc., making it hard to meet the efficiency and quality requirements of PCB designs [8, 9]. Therefore, modern PCB placement still relies on manual optimization and thus consumes significant design time, especially for large-scale designs.

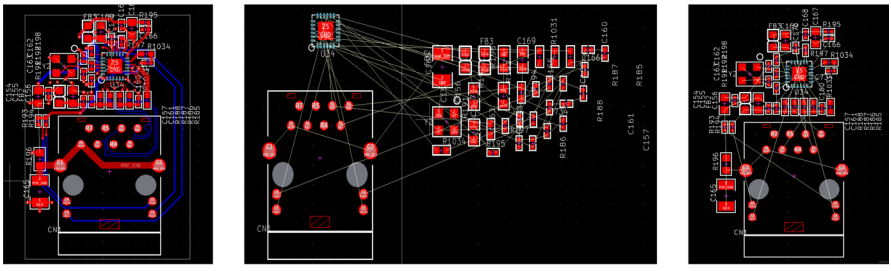
Industrial PCB designs often have the same or similar components that need special placement. As a result, reference placement has emerged as key research

to improve the efficiency and quality of PCB placement and design. Reference placement aims to take the placed components selected by the users as a template and find out the components with the same pattern to be placed for placement reuse. Figure 1 displays the function of reference placement. In Fig. 1a, there is a high-quality PCB placement optimized by manual, which is selected as the template. Now another circuit including some same or similar components needs to execute placement as shown in Fig. 1b. Reference placement identifies the same patterns with the template and places these components. Finally, it generates the same or similar placement with the template, as shown in Fig. 1c.

Recently, graph matching has been applied in integrated circuits, particularly in analog circuit placement. Kunal et al. [10, 11] proposed a GNN-based method to identify the hierarchical functional blocks in analog circuits. The authors convert a circuit into a bipartite multigraph, which consists of element nodes and net nodes. The edges are labeled with pin information, and then, the graph is used to train the GNN to identify functional blocks. The purpose is to approximately match the structure and connections with the graph. The same spirit of graph matching in these works and reference placement is finding similar structures between circuits. However, the graph in [10, 11] is extracted from a transistor-level circuit, while the graph in PCB reference placement is extracted from an IC-level circuit. Generally, each transistor has only a few pins, while a component in PCB design may have hundreds of pins. This results in a substantial difference in the degree of nodes, making PCB subgraph matching potentially more challenging.

To achieve high-quality reference placement, in this paper, we convert the netlist information into a graph and then model the reference placement as a subgraph matching problem. In detail, the template and the components to be placed are defined as a query graph and a data graph, respectively. Then, we search for similar patterns from the data graph and query graph, which can be achieved by using the subgraph matching algorithms in graph theory.

In recent years, researchers have made great progress in subgraph matching. Existing subgraph matching algorithms can be generally divided into three categories: graph index-based, constraint programming-based, and tree search-based approaches. Representative graph index-based methods include QuickSI [12], Spath



(a) (b) (c)

Fig. 1 An example of reference placement. **a** A circuit is selected as the template. **b** A circuit with some same of similar patterns of the template needs placement. **c** The reference placement result of (**b**)

[13], and TurboISO [14]. The basic idea is to retrieve a graph with a given pattern from the graph database according to the graph index, where the index is a vector or a feature tree representing the structure and semantic information of the graph. The index is used to search for all subgraphs in the target graph with the same index and then refines the solution by checking for isomorphic constraints and removing any incongruent matches. Existing constraint programming-based methods [15–17] map a subgraph matching problem into a constraint satisfaction problem to effectively find matched subgraphs. Given a set of variables and a set of constraints between them, the approach finds all the variable assignments to satisfy all the constraints. By conveying constraints, nodes not included in the solution can be iteratively filtered out until a few candidate matches are left. Tree search-based methods are considered to be the most popular kind of subgraph matching approach. This approach finds all subgraphs that meet the matching requirements in the search space tree. The search space tree is a tree structure. Each node in the tree represents a matching pair, and the path from the root node to the current node represents a partial match. The subgraph matching algorithms generally use heuristic rules to reduce the search space to achieve the purpose of fast matching. The search process generally adopts the depth-first search of backtracking to gradually determine the complete matching results. Representative tree search-based algorithms include Ullmann [18], series of VF [19–21], BM1 [22], L2G [23], RI algorithm [24], and DAF [25].

Efficiency has always been a significant challenge for applying subgraph matching algorithms in industrial chip design, especially for large-scale designs. Some algorithms have been proposed to solve the case of graphs with thousands of nodes and high edge density, such as the state-of-the-art VF3 [21] and DAF [25]. To improve the matching efficiency, VF3 and DAF both prune infeasible branches (failure sets) in the search tree in advance. Although these algorithms enjoy some success to some degree, they still exhibit some intrinsic limitations. First, these algorithms include redundant calculations because the matching process is carried out by the backtracking method. Second, the execution times of each method fluctuate greatly for different PCB instances, often too time-consuming for large-scale components placement. Third, these algorithms cannot handle the pattern differences between the query and data graphs well, leading to inflexible placement reuse. To handle these challenges, we propose an efficient subgraph matching algorithm called D2BS with diversity tolerance and improved backtracking to facilitate PCB reference placement. Our major contributions are as follows:

- By abstracting the components and nets as nodes, we convert the PCB netlist information into a graph and then model the reference placement as a subgraph matching problem.
- The features of components and nets on a PCB are used for the candidate set construction and filtering based on a proposed similarity score. As a result, we can prune unnecessary search paths to reduce the graph search space significantly.
- We propose a graph diversity tolerance strategy to handle inexact graph matching. A query graph is split to obtain the same pattern as a data graph to match as many nodes as possible.

- We apply a hierarchical layer-wise subgraph matching method and develop a novel backtracking method with branch backtracking and matched-node snatching to facilitate the matching process.
- Experimental results based on the industrial PCB designs show that our proposed D2BS algorithm is suitable for PCB reference placement, which not only increases matching accuracy but also significantly minimizes the computation time for large-scale designs.

The rest of this paper is organized as follows: Sect. 2 gives the graph construction, terminologies, and problem statement. Section 3 details our proposed algorithm. Section 4 shows the experimental results, followed by the conclusion in Sect. 5.

2 Preliminary

In this section, we first convert the PCB netlist into a graph and then give some terminologies and the problem statement.

2.1 Graph construction

In the previous placement problems, the netlist information is usually formulated as a hypergraph, where each component is regarded as a node, and each net is regarded as a hyperedge [26, 27]. However, such a graph construction method is not suitable well for solving reference placement by subgraph matching. In the subgraph matching problem, the connection between nodes is a judgment condition to find matching pairs. For a multi-pin net in the netlist, each component of the net is connected to the other components of that net. Take Fig. 2 as an example. In Fig. 2a, there is a multi-pin net connecting four components. The graph construction method in [26, 27] formulates these components and the net into a hypergraph, as shown in Fig. 2b. For each component node, it, respectively, connects to the other three component nodes through the net. We can transform the multi-pin net into two-pin nets, resulting in the equivalence graph indicated in Fig. 2c. According to 2c, we can clearly store the

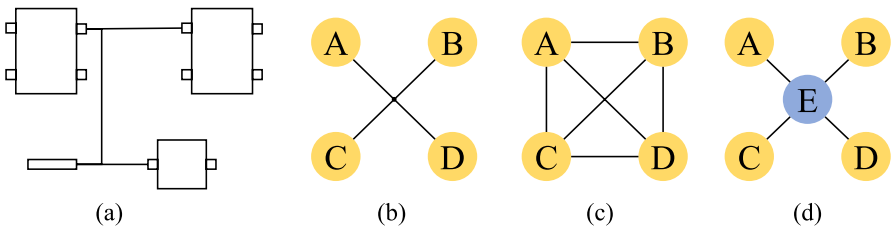


Fig. 2 Two methods of formulating a graph. The yellow circle represents component nodes, and the blue circle represents net nodes. **a** A multi-pin net and its components. **b** Hypergraph formulated by the method in [26, 27]. A hyperedge connects four nodes. **c** Equivalence graph of (**b**), in the form of two-node edge. **d** Bipartite graph formulated by our method, where a net node connects four component nodes

connections. For example, the connections of node A are $\{B, C, D\}$, the connections of node B are $\{A, C, D\}$, and so on. Thus, the total number of connections comes to 4×3. This generates so many connections in the search space, leading to significant computational time overhead.

In contrast, this paper proposes a different graph construction method that is more suitable to facilitate the storage and utilization of netlist information. Specifically, we treat the components and nets as two types of nodes, namely component nodes and net nodes. There exists an edge between a component node and a net node if the component is connected by the net. As shown in Fig. 2d, each component node connects to the net node instead of the other component nodes, so it stores only just the net node E in its connection array. The total number of connections is 4×1, which is less than that is shown in Fig. 2c. When the multi-pin net has large amounts of components, the reduction in connections in the graph will be extremely substantial. It evidently indicates the advantage of our graph construction method. Therefore, in this graph construction, component nodes can only connect to net nodes and vice versa, so the constructed graph is bipartite. Figure 3 gives an example of graph construction, where Fig. 3a shows a circuit diagram and Fig. 3b shows the corresponding bipartite graph.

2.2 Terminologies

- *Template* a set of placed electronic components and routed nets selected by users for placement reuse.
- *Query graph* the graph exacted from the template, represented as $Q = (V_q, E_q)$, where V_q is the node set and E_q is the edge set.
- *Unplaced module* a set of components and nets not yet placed and routed.
- *Data graph* the graph exacted from the unplaced module, represented as $D = (V_d, E_d)$, where V_d is the node set and E_d is the edge set.
- *Directed acyclic graph (DAG)* a directed acyclic graph is a connected directed graph without cycles. In our subsequent algorithm, we will transform the query graph Q into a DAG Q_{D}.

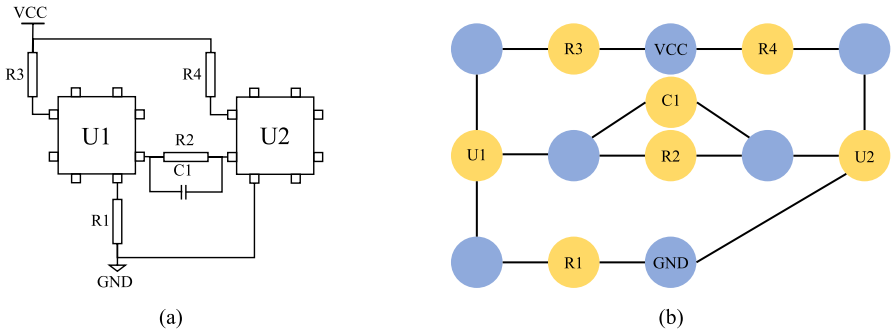


Fig. 3 An example of graph construction. **a** A circuit diagram. **b** Corresponding bipartite graph, where the yellow circles represent the component nodes, the blue circles represent the net nodes, VCC represents the power net, and GND represents the ground net

- *Candidate set* the set of data nodes that can be matched to a query node u, represented as $C(u)$.
- *Embedding* a map between Q and D, which denotes the matched pairs and is represented as $M = \{(u, v)|u \in V_q, v \in V_d\}$.
- *Matching accuracy (Acc)* the ratio of the matched query node number (or the matched data node number) in total query nodes (or total data nodes), given by:

$$\text{Acc} = \begin{cases} \frac{|V_{\text{matched}}|}{|V_q|} & \text{for query nodes,} \\ \frac{|V_{\text{matched}}|}{|V_d|} & \text{for data nodes.} \end{cases} \tag{1}$$

  where V_{matched} denotes the matched query nodes (or matched data nodes). It is an industrial-standard evaluation metric for PCB reference placement and is suitable for practical designs. In our experiments, Sect. 4.1.1 uses Acc for query nodes, and Sect. 4.1.2 uses Acc for data nodes.

Frequently used notations are listed in Table 1.

2.3 Problem statement

Various electronic components on a PCB are connected through nets, and the combination of components and nets can be represented as a bipartite graph, as described in Sect. 2.1. Further, the template placed is selected as a query graph, and the components and nets to be placed are regarded as a data graph. As a result, reference placement can be formulated as a subgraph matching problem. We formally define the addressed problem as follows:

- *Subgraph matching-based reference placement problem* Given a data graph $D = (V_d, E_d)$ and query graph $Q = (V_q, E_q)$, the reference placement problem aims to find all embeddings of Q in D. For the parts in Q that have different patterns from those in D, the *Acc* defined in Eq. (1) should be as high as possible.

Table 1 Frequently used notations

Notation	Description
Q and D	Query graph and data graph
V and E	Vertex set and edge set
DAG	Directed acyclic graph
Acc	The ratio of successfully matched nodes
u and v	A query node and a data node
$C(u)$	The candidate set of a query node u
NEC	Neighborhood equivalence class
Q_{D}	Query DAG

- *PCB Placement generation by the result of subgraph matching* According to the matched pairs, modify the locations of unplaced components. Suppose a matched pair is (u, v), where u is a query node and v is a data node. If the corresponding component of u is located at (x, y), then the corresponding component of v will be placed at $(x + a, y + b)$, where a and b are set by users.

After solving the subgraph matching problem, the data nodes matching the template can reuse the placement of the template to reduce manual design and computation times, especially for large-scale designs.

3 Our algorithm

To further improve the efficiency and quality of the subgraph matching and make it suitable for industrial PCB designs, we propose an algorithm named D2BS. The overall framework of our algorithm is shown in Fig. 4. D2BS algorithm generates matching pairs between the query graph and the data graph according to the PCB netlist information, which is decomposed into three phases: (1) graph preprocessing;

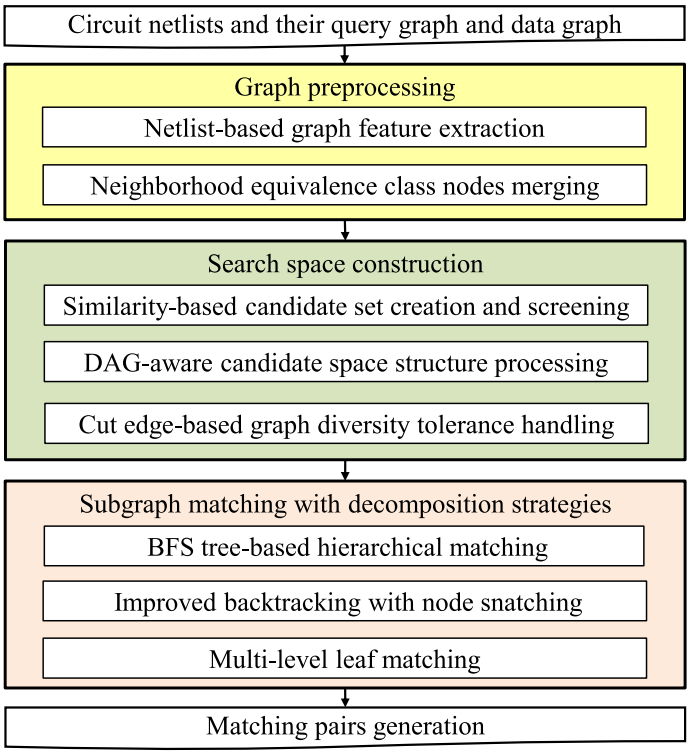


Fig. 4 Overall framework of the proposed D2BS

(2) search space construction; and (3) subgraph matching with decomposition strategies.

In the graph preprocessing phase, we first extract the effective features of components and nets based on PCB netlist information (Sect. 3.1). The features will be used for the subsequent candidate set construction and filtering. Moreover, the nodes of the NEC are combined into one hypernode to simplify the template architecture (Sect. 3.2). In the search space construction phase, to shorten the search path, the candidate set $C(u)$ for each node u in query graph Q is constructed and filtered based on the proposed similarity score (Sect. 3.3). Additionally, a candidate space structure similar to that proposed in [25] is constructed and optimized based on a DAG (Sect. 3.4). Besides, we develop the graph diversity tolerance strategy to achieve inexact matching, so that the nodes of Q that have different patterns than that of D can be matched as much as possible (Sect. 3.5). In the subgraph matching phase, we develop a novel hierarchical matching (Sect. 3.6) with the branch backtracking and matched-node snatching techniques to facilitate the matching process (Sect. 3.7). Finally, the multi-level leaf nodes are graded according to the distance from the outermost layer, making fast matching (Sect. 3.8). We shall detail these major parts in the following sections.

3.1 Netlist-based graph feature extraction

In the graph, each node should store attributes and connection information for the corresponding component or net. The features of a node are composed of a type value and a connection array extracted from netlists. The type value represents the type of node, which could be divided into three categories: power or ground net, general net, and component. The connection array stores different kinds of attributes and connection information according to the node types. The details are as follows:

- *Power or ground net* The necessary information of the node is just its name, extracted from the netlist. It is worth mentioning that the name could contain voltage as an identification. Several power supplies in the netlist may have different voltages, so we mark the names of the power nets with their voltages such as "VCC5v." Consequently, we can distinguish different power net nodes.
- *General net* This node records the pins in the net, including the component to which the pin belongs. Specifically, we adopt type–pin pairs to represent the node type and the pin index of the connected components.
- *Component* This node records the name of the power or ground net connected to this component and the adjacent components connected by different pins. Note that components connected to power or ground are not considered adjacent, so they cannot be added to the connection array of each other.

3.2 Neighborhood equivalence class (NEC) nodes merging

To reduce the number of query nodes and shorten the traversal time, we construct the NEC hypernode [14] by merging the nodes with the same features, degree, and adjacent nodes, as displayed in Fig. 5. The NEC of node u in Q is a set of query nodes that have an equivalence relation with u. The nodes with the same attributes and connections have the same candidate sets, so these nodes can be combined to share one candidate set. Note that only the same-type nodes could be merged. For those different types of nodes, it may not be beneficial to merge them because it will not bring a smaller search space. Besides, all the NEC nodes in query graph Q can be found through breadth-first search (BFS) [14]. Firstly, we select the node with the highest degree as the root node. Then we traverse every node in Q through BFS to check the condition, including type value, feature array, degree, and adjacent nodes. The node that has the same feature as some other nodes is an NEC node. Finally, we merge each NEC into a hypernode. Note that Q denotes the query graph after NEC merging in the following sections.

3.3 Similarity-based candidate set creation and screening

In query graph Q, we create a candidate set for each query node to store similar data nodes that can be matched in data graph D. The candidate set significantly influences the quality and efficiency of subgraph matching. With smaller candidate sets, the search space is reduced. Therefore, we propose a candidate set creation and screening method to construct and narrow the search space. The data nodes are selected as the initial candidates for a query node based on their features. The construction of the initial candidate set varies concerning the type of query nodes:

- For a power or ground net node u_1, the data nodes with the same name are stored in the initial candidate set.

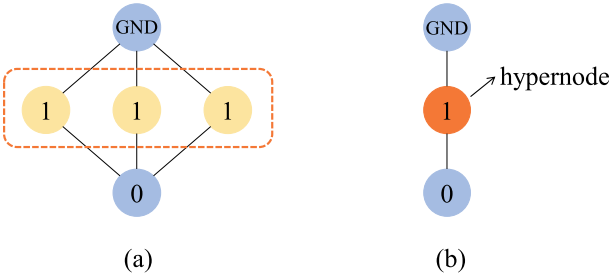


Fig. 5 An example of NEC nodes merging, where GND represents the ground node and the numbers denotes the node type. **a** Three nodes framed by dashed lines have the same attributes and can be merged into an NEC hypernode. **b** Graph after NEC nodes merging

- For a general net node u_2, if there are the data nodes whose connected component types and pin indices are covered in the connection array of u_2, these data nodes are saved as the initial candidate set.
- For a component node u_3, the data nodes with the same type are saved as the initial candidate set.

Although the data nodes with similarity are stored in the corresponding initial candidate sets, there are still redundant nodes with a low similarity that need to be further filtered. Next, we design similarity scores between a query node and a candidate for different types of nodes and then screen the initial candidate sets by reserving only the data nodes with relatively high similarity scores. Component nodes and net nodes have different similarity score calculation methods, which are detailed as follows.

- Suppose a general net node u in Q, whose connection array contains the pairs F_u of the connected component type and pin index. One of the candidate nodes of u is v, whose connection array is F_v. Then, the similarity score is defined as:

$$g_v = |F_u \cap F_v|. \tag{2}$$

  That is, g_v is the number of the same elements in F_u and F_v. It represents the number of the same component connections between the data node v and the query node u.
- Suppose a component node u in Q, and one of the candidate nodes of u is v. The similarity score between u and v is defined as:

$$g_v = \mathrm{Cnt}(P_v \geq P_u), \tag{3}$$

  where P_v represents the number of components connected to a pin in v, P_u represents the number of components connected to a pin in u, and Cnt is the pin count of u that meets the condition in parentheses. The formula gives a condition that the degree of a pin of v should not be less than that of the corresponding pin of u. We record the count of pins that satisfy this condition; the higher the count, the more similar v and u are. Note that the screening rule of the component candidate believes that only when the pin degree of a candidate is not less than that of the corresponding component node, the candidate is the node with a large similarity score in the candidate set. Therefore, this rule is only applicable to the case that the connection of query nodes is not more than that of data nodes.
- As for power or ground nodes, the initial candidate sets are not screened to avoid excessive deletion.

Figure 6 gives an example of the similarity calculation, where Fig. 6a shows a general net query node u_{n1} (corresponding to net1) with the three connected components. The numbers in the figure represent the index of pins. Obviously,

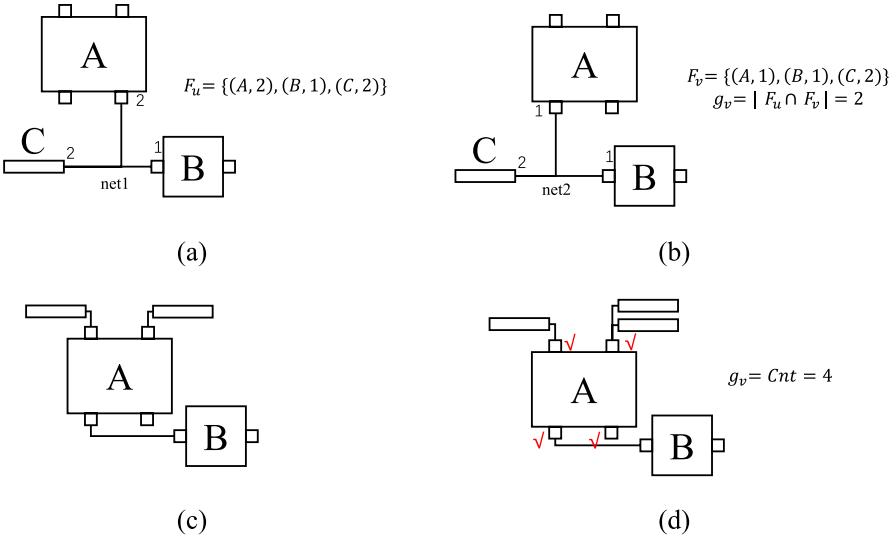


Fig. 6 An example of similarity calculation. **a** A query net node u_{n1}. **b** A candidate data node v_{n2} of u_{n1}. **c** A query component node u_{cA}. **d** A candidate data node v_{cA} of u_{cA}

$F_u = \{(A,2),(B,1),(C,2)\}$. Figure 6b shows a candidate nodes v_{n2} (corresponding to net1) of u_{n1}, and the F_v of v_{n2} is $\{(A,1),(B,1),(C,2)\}$. The common part of F_u and F_v is $\{(B,1),(C,2)\}$. Hence, according to Eq. (2), the g_v of v_{n2} equals 2. Similarly, Fig. 6c shows a component query node u_{cA}, and Fig. 6d gives a candidate node v_{cA} of u_{cA}. We check whether each pin degree of v_{cA} is no less than the corresponding pin degree of u_{cA}. The result is the four pins of v_{cA} satisfy the condition, so g_v equals 4.

The similarity-based candidate set creation and screening is the initial phase in filtering the candidate nodes. To prevent the excessive elimination of potential candidates, we establish a similarity threshold at a moderate level.

3.4 DAG-aware candidate space structure processing

For two adjacent nodes with a connection in the query graph, there should also be a connection between the corresponding candidate nodes in the data graph to meet the subgraph matching. However, since the similarity-based candidate set creation and screening stage does not consider the relationship between adjacent nodes, there still may be no connection between the corresponding candidate data nodes of adjacent query nodes, resulting in unnecessary search paths in the subgraph matching. Therefore, similar to the work [25], we adopt a DAG-aware candidate space structure processing method to refine the search space for the subsequent subgraph matching.

Candidate space is an auxiliary data structure that stores the candidate set $C(u)$ of each node $u \in Q$ and the edges between candidates, which satisfies the following conditions [25]:

- For each u in Q, its candidate set $C(u)$ is a subset of $C_f(u)$, which denotes the filtered candidate set.
- There is an edge between $v \in C(u)$ and $v' \in C(u')$ if and only if $(u, u') \in E_q$ and $(v, v') \in E_d$.

According to [25], we also have the following theorem.

Theorem 1 *Since all edges in Q are used in the candidate space structure, candidate space is a complete search space. In other words, finding the embeddings of Q in D is equivalent to finding the embeddings of Q in candidate space.*

3.4.1 DAG construction

DAG plays an important role in optimizing candidate space structure. These are two steps to construct DAG.

- Select a root node. Because the root node of the DAG is the first node to match during the subgraph matching process, it is desirable that the root node can have fewer the number of candidate data nodes and a larger degree for better pruning. The fewer the number of candidate nodes, the greater the possibility of correct matching of the root node. Besides, the larger the degree, the more branches under the root node, which is conducive to the backtracking process in the subgraph matching. Hence, we select the node as the root node with the smallest ratio of the number of candidate data nodes to the degree.
- Determine the parent–child relationship between adjacent nodes. After determining the root node of the DAG, we traverse the query graph through BFS. Nodes at the same level are grouped by type, and these groups are then sorted so that less common types appear earlier in the data graph. The nodes in each group are sorted in descending order of node degree. At this point, a BFS traversal order of the query graph is obtained, which can be utilized in the subsequent construction of candidate space structures. According to this sequence, the direction of adjacent nodes is known, and the parent–child relationship between two adjacent nodes can be determined.

3.4.2 Candidate space structure processing

The initial candidate space structure is redundant because it contains all filtered candidate sets and the edges between them. Consequently, it needs to be optimized based on DAG, that is, the candidate space is compressed according to the connection relationship between adjacent nodes. Figure 7a displays an instance of Q_{D}. The parent–child relationship in Q_{D} provides assistance for optimizing. The Q_{D}^{-1} shown in

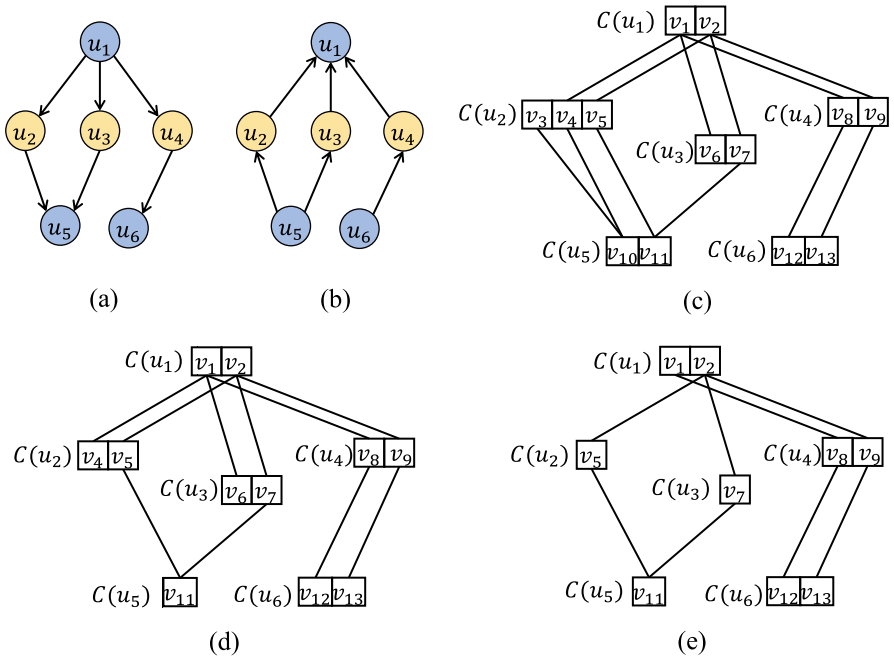


Fig. 7 An example of the candidate space structure processing. u represents a query node, $C(u)$ is the candidate set of u, and v represents a data node in $C(u)$. **a** A DAG Q_D. **b** Q_D^{-1}. **c** Initial candidate space structure. **d** Candidate space after the first step refinement. **e** Final candidate space after the second step refinement

(b) is also needed, which is in the reverse order with Q_D. The optimization steps of the candidate space structure are as follows:

- Step 1. According to the sequence of Q_D^{-1}, traverse each node u in Q, and check whether each candidate node of u has an edge with the candidate nodes of u' child node. If there is no edge connection, the candidate node of u is deleted from the candidate space structure and the candidate set $C(u)$.

  For example, for the initial candidate space structure shown in Fig. 7c, the candidate node v_{10} of u_5 has no connection with any candidate of u_3, so it will be removed from $C(u_5)$. Similarly, v_3 is removed from $C(u_2)$. Figure 7d shows the candidate structure after this step refinement.
- Step 2. Repeat step 1 by traversing the nodes in Q in the order of Q_D. Take Fig. 7 again, for the candidate space structure shown in Fig. 7d, since v_4 and v_6 are not connected to the candidate of u_5, they are removed. After that, we finished the second step of refinement, as shown in Fig. 7e.

Note that we will repeat the above two-step refinement until a compact enough candidate space structure is achieved.

The DAG-aware candidate space processing is a refined optimization method for candidate space management. For exact matching scenarios, where the query graph pattern is fully encompassed within the data graph, this processing ensures no over-filtering occurs. However, in the case of inexact matching (introduced in Sect. 3.5), it may result in the absence of candidate nodes for certain query nodes. When some query nodes share the same candidate set, but the number of candidate nodes is insufficient to cover all these query nodes, we flag these query nodes. In such a situation, it becomes evident that there will be query nodes that cannot be matched. Additionally, due to the DAG-aware optimization, some candidate nodes of these query nodes' parents or children may be deleted, further exacerbating the issue of unmatched query nodes. To address this challenge, we have devised a strategy. Specifically, for the parents or children of the flagged query nodes, the DAG-aware processing will refrain from deleting candidate nodes that do not have any connections to the candidate nodes of the flagged query nodes. This approach effectively avoids over-filtering and ensures a more accurate candidate space for matching.

3.5 Cut edge-based graph diversity tolerance handling

In the actual PCB instances, there may be slight differences between the template and the part of the data graph that is intended to be matched. It is expected that these differences can be tolerated to enable the designer to reuse placement without being

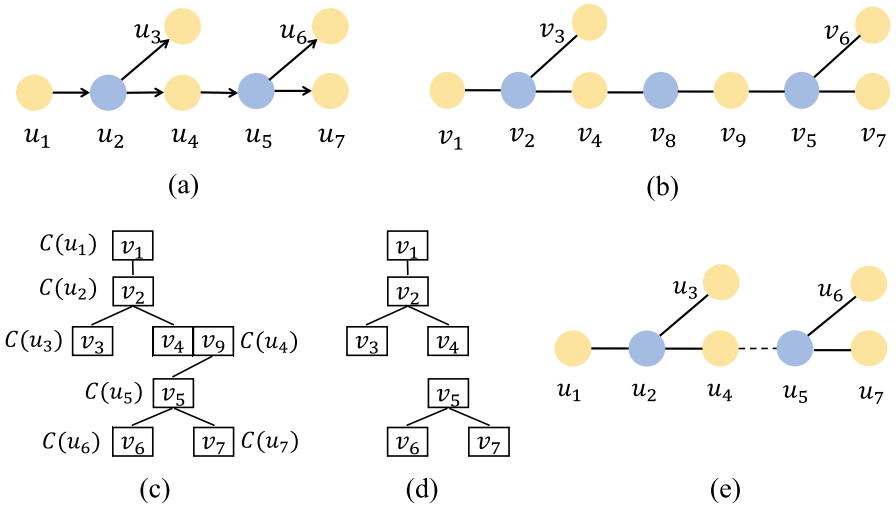


Fig. 8 An example of the graph diversity tolerance strategy. **a** DAG of a query graph Q_D. **b** A data graph D. **c** Initial candidate space structure for Q_D. **d** Candidate space structure after v_9 being deleted from the candidate set $C(u_4)$. **e** Query graph after removing the cut edge between u_4 and u_5

constrained to the exact matching conditions. When exact matching conditions are strictly enforced, even slight differences can result in a significant reduction in Acc. Conversely, by disregarding these minor differences, it becomes possible for a greater number of query nodes to correctly match the corresponding data nodes.

Take Fig. 8 as an example, where Fig. 8a shows the DAG of a query graph Q_D, and Fig. 8b shows a data graph D. As can be seen from the figure, most parts of Q_D and D are similar, but they are not completely the same. Suppose we will search for all the embeddings of Q_D in D, and the traversal order of the DAG is $u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow u_4 \rightarrow u_5 \rightarrow u_6 \rightarrow u_7$. The candidates of u_4 are v_4 and v_9, and all candidates of other nodes are nodes with the same subscript in D, as shown in Fig. 8c. When optimizing the candidate space structure in Fig. 8c, the connection relationship between adjacent nodes in the data graph D and the query graph Q_D should be consistent. Therefore, for the adjacent nodes u_2 and u_4 in Q_D, because candidates v_2 and v_9 in D are not connected, v_9 is deleted from the candidate set of u_4, as shown in Fig. 8d. After that, if we still continue to optimize the candidate space structure, for the adjacent nodes u_4 and u_5, v_5 is deleted from the candidate set for the same reason, so there is no element in the candidate set of u_5. Consequently, u_5, u_6 and u_7 cannot match the corresponding data nodes.

To avoid excessive candidates being filtered and improve the matching accuracy, we propose a graph diversity tolerance strategy in the process of candidate optimizing. This strategy decreases the dissimilarity between the query graph and the data graph by deleting cut edges. We first define the cut edge as follows.

Definition 1 (cut edge) If an edge e in query graph Q connects the nodes u_1 and u_2, and there is no edge between the candidate sets $C(u_1)$ and $C(u_2)$, then the edge e is called the cut edge.

Then, we find and remove all cut edges in Q when optimizing the candidate space structure to prevent excessive candidates from being filtered, so that the query pattern can be flexibly transformed into the same patterns as that in the data graph for matching. For example, the cut edge (the edge between u_4 and u_5) is removed as shown in Fig. 8e, because there is no edge between any candidate nodes of u_4 and u_5

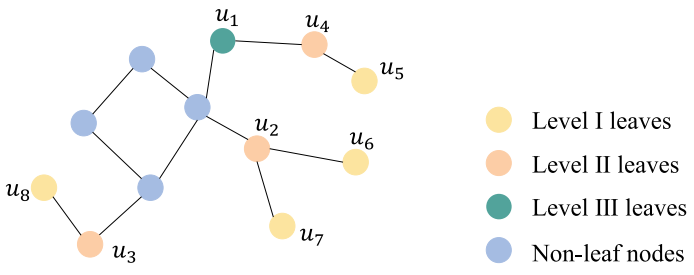


Fig. 9 Multi-level leaf nodes marked graph. The level of a leaf node is the round where it is removed

as shown in Fig. 8d. At this point, Q_D is divided into two parts, and the patterns are both included in D, so all nodes in Q_D can find matched nodes in D.

3.6 BFS tree-based hierarchical matching

Before the subgraph matching, we further adopt the leaf decomposition strategy [28] to preprocess the query graph to accelerate the matching process. Specifically, we iteratively remove the degree-one nodes in Q until there is no degree-one node, and then, the query graph Q is decomposed into the set of degree-one nodes and the set of remaining nodes, as shown in Fig. 9. We set the level information for each decomposed node according to the round where it is removed. Those degree-one nodes (e.g., u_5, u_6, u_7, and u_8) removed in the first round are marked as Level I leaves. Then, the new degree-one nodes (e.g., u_2, u_3, and u_4) appear, and they are removed in the second round and marked as Level II leaves. In this way, we separate and mark the multi-level leaf nodes. After successively removing all leaf nodes, the smallest connected subgraph left is the core structure. Obviously, the graph is divided into two parts, the non-leaf nodes (core structure) and the multi-level leaf nodes.

We first find embeddings of the core structure in this subsection by BFS tree-based hierarchical matching and then find the embeddings of the multi-level leaf nodes by the leaf-matching algorithm presented in subsection 3.8.

3.6.1 BFS tree construction

For the core structure subgraph (denoted as Q_s), we construct a BFS tree to obtain the matching order. In fact, circles composed of directed edges do not occur in the DAG constructed via BFS. However, when all edges are regarded as undirected edges, circles may occur and confuse the matching process in PCB instances. Hence, we delete one of the edges that can be closed to guarantee no undirected circles exist, so as to avoid matching errors. We first traverse the query nodes by depth-first search, and when a circle is found, we remove the last edge that formed the circle. After those edges are deleted, the BFS is conducted to construct the DAG. In this situation, the DAG is in the form of a tree, called the BFS tree.

Theorem 2 *In the BFS tree of Q_s, node types of each layer are the same. That is, they are all component nodes or net nodes. Component and net nodes appear alternately in the layers of the BFS tree.*

Proof Since we delete one of the edges that can form an undirected circle, each node has at most one parent node. Besides, we choose the node with few candidates and a high degree as the root. When constructing DAG, we traverse the bipartite graph Q with breadth-first search, so the types of nodes in the same layer of the DAG are the same, and the types of nodes in adjacent layers are different. □

3.6.2 Subgraph hierarchical matching

Considering Q_s may include several connected graphs, Q_s is divided into multiple subgraphs according to connectivity. The divided subgraphs are denoted as Q_i ($i = 1, 2, ...$). Noted that the graph is bipartite and no circle exists, we hierarchically search for embeddings of Q_i in the candidate space structure according to the BFS tree. In this case, parent nodes of one layer can only exist in its former layer in the BFS tree, so we can achieve efficient layer-by-layer hypernode matching of Q_i, instead of node by node.

Algorithm 1 Subgraph hierarchical matching

Require: The BFS tree $BTree$ with root node u_r.
Ensure: The matching result $matchPair$.
1: $matchPair \leftarrow \emptyset$;
2: $matchPair$.append($[u_r, v_r]$);
3: $lIdx = 1$;
4: **while** $lIdx <$ Length($BTree$) **do**
5: $B_l \leftarrow$ Sort($BTree[lIdx]$);
6: $UB_l \leftarrow$ The nodes that are not visited in B_l;
7: **for** each u in UB_l **do**
8: $isBack, lIdx \leftarrow$ NodeCheck($u, matchPair, lIdx$);
9: **if** $isBack == 1$ **then**
10: Break;
11: **end if**
12: **end for**
13: $lIdx \leftarrow lIdx + 1$;
14: **end while**
15: **return** $matchPair$.

Algorithm 1 summarizes the subgraph hierarchical matching for Q_i. The input is the BFS tree *BTree* with root node u_r. Generally, the ground node GND is chosen to be the root node, but if there is no GND in the template, the root selection method of DAG will be utilized. Besides, the output is the matching result *matchPair*, which contains the matched pairs between the nodes in the query graph Q_i and their corresponding matched data nodes. Line 1 initializes the *matchPair* and then starts to match. In Line 2, the root node u_r is matched at first. For easier presentation, we assume that the selected root u_r has only one candidate date node v_r. In Line 3, the variable *lIdx* denotes the index of the current matching layer of the BFS tree, which is initialized as 1. Then, the matching process is performed layer by layer of the BFS tree, as shown in Lines 4–14. The matching order of nodes in the same layer is dynamically sorted according to the length of candidate sets in Line 5, and B_l stores the sorted nodes in the layer. Line 6 further collects the unvisited nodes in B_l and puts them in UB_l. For each unvisited node u, the function `NodeCheck`(u, *matchPair*, *lIdx*) in Line 8 checks whether query node u can be

matched, and finally returns the variable *isBack* and *lIdx*. We shall detail `NodeCheck` function in Algorithm 2. Then, if a layer can be matched successfully, enter the next layer; otherwise, return to the previous layer to modify the matched pairs. The same operations will be applied if the root has multiple candidates.

Algorithm 2 details the function `NodeCheck`, which adopts different operations for different match cases. The input parameters are the current query node *u* to be matched, the matched pairs *matchPair*, and the index of layer *lIdx*. Finally, the function returns the variables *isBack* and *lIdx*. In Lines 1–2, the candidate data nodes of *u* are collected into *Cdd*, and the matched data nodes in *matchPair* are collected into *matchedD*. To prevent multiple query nodes from matching to the same data node, a candidate will not be matched again if it has been matched in embedding. Hence, those unoccupied candidate data nodes in *Cdd* form the *vList* in Line 3, and the matched data node of *u* will be searched from *vList*. If *vList* is not empty, it means that the query node *u* can be matched, and we set *isBack* to 0 in Line 5. Besides, we update the matched pairs *matchPair* in Line 6. Specifically, if *u* is not an NEC node, assign one candidate data node to it; otherwise, several candidate nodes (the same quantity as NEC) are assigned to *u* so that every node in NEC can be matched. In the case of Lines 8–13, *vList* is empty, which means that there is no candidate data node that can be matched. We set *isBack* to 1 and then revise the matching pairs by backtracking. If the current layer is not the first layer, then we conduct branch backtracking `Backtrack` in Line 10 (to be elaborated in Sect. 3.7.1). If backtracking to the root node is still ineffective, the matched-node snatching function `NodeSnatch` will be performed in Line 12 (to be elaborated in Sect. 3.7.2). Finally, these two functions will return the layer of the parent of the backtracking node or snatching node.

Algorithm 2 NodeCheck (*u*, *matchPair*, *lIdx*)

Require: $u,\ matchPair,\ lIdx.$
Ensure: $isBack,\ lIdx.$
1: $Cdd \leftarrow$ The candidate set of u;
2: $matchedD \leftarrow$ The matched data nodes in $matchPair$;
3: $vList \leftarrow Cdd - matchedD$;
4: **if** $vList$!= $\emptyset$ **then**
5: $\quad isBack = 0$;
6: $\quad matchPair \leftarrow matchPair + \text{Match}(u, vList)$;
7: **else**
8: $\quad isBack = 1$;
9: $\quad$ **if** $lIdx$!= 1 **then**
10: $\quad\quad lIdx \leftarrow \text{Backtrack}(u)$;
11: $\quad$ **else**
12: $\quad\quad lIdx \leftarrow \text{NodeSnatch}(u)$;
13: $\quad$ **end if**
14: **end if**
15: **return** $isBack, lIdx$;

3.7 Improved backtracking with node snatching

In hierarchical matching, it is possible that some query nodes cannot continue to match because none of their candidates meet the matching conditions, which results from the wrong-matched pairs in embeddings. Consequently, the wrong-matched pairs are expected to be modified. Assuming that the ancestor set of a query node that cannot continue to match is A, there are two types of matching errors. The first type is that one of the nodes in A does not choose the correct candidate. We adopt the branch backtracking method to address it. The second type is for the candidate occupation by another query node. We propose a matched-node snatching rule to snatch the candidate forcibly.

3.7.1 Branch backtracking

Let the parent node of u be p. The *parentPair* is the matched pairs of p and all the matched pairs of the children of p. The core of branch backtracking is to return to the previous layer of the unmatched u to rematch the parent node p and its children.

Specifically, when a query node u cannot be matched, we first remove the *parentPair* from the *matchPair* and backtrack to the previous layer of the BFS tree, which is the layer where the parent node p is located. Then, we try to rematch the parent node, but exclude the previously matched nodes from the candidate set. This ensures that those nodes will not be matched again in future iterations. If the parent node p can be rematched, the matching process will continue; otherwise, the backtracking process continues. However, if the correct match cannot be found even after backtracking to the root node, it is considered that some correct-matched nodes are occupied. In this case, the matched-node snatching rule is adopted to solve the problem.

3.7.2 Node snatching

The purpose of node snatching is to rectify the matching errors that cannot be resolved through branch backtracking. The core idea of matched-node snatching is to forcibly snatch the ideal data nodes for the query nodes that cannot be matched by the branch backtracking, even though these ideal data nodes have been matched by other query nodes. The node snatching is based on a back tree. In the following, we first introduce the concept of the back tree.

Definition 2 (back tree) The back tree is a tree that is constructed to determine the order of node snatching. It has a virtual root node, and the other nodes, called "snatchable nodes," represent the BFS tree nodes that can snatch the matched nodes occupied by other BFS tree nodes. The order of node snatching can be obtained by depth-first search on the back tree.

For each unvisited problem node that cannot be matched even after backtracking to the root node, a back tree is constructed, and the back tree will be deleted after the problem node is solved. The construction of the back tree is dynamic and will be

gradually extended as the snatch layer increases. Firstly, in the process of the branch backtracking from a problem node to the root node, all nodes that meet the snatchable node conditions will be saved as a layer in the back tree. The snatchable node conditions include: (1) the candidate set of the node is not empty, and (2) at least one candidate has been matched by the embedded nodes. Then, when a single-node snatching fails to resolve the matching issue of the problem node and its snatching node, the snatchable nodes saved in the new round of backtracking to the root will become the child nodes of the previous snatching node.

It is obvious that a back tree may consist of multiple layers. Since the snatched data node may not necessarily be the correct one, snatching may result in an excessively long search path, causing the back tree to deepen significantly. To mitigate this issue, we establish a threshold for the maximum number of layers in the back tree. We expect the correct match to be found by branch backtracking and node snatching before all back tree nodes are traversed. Otherwise, the problem query node is considered unmatched and skipped.

In the following, we explain the back tree and the matched-node snatching, using Fig. 10 as an illustrative example. Figure 10a–c depicts three distinct stages, respectively, where the figure on the left of each stage is a BFS tree, and the figure on the right of each stage shows the matched pairs and the back tree. In Fig. 10, we use different colors to indicate the status of the nodes and assume that every query node should match the corresponding data node with the same subscript.

In Fig. 10a, suppose that u_6 has matched v_7, u_7 has matched v_8, and the remaining blue query nodes have also matched the data nodes with corresponding subscripts. Now, we are looking for a matching data node for u_8. Assuming that u_8

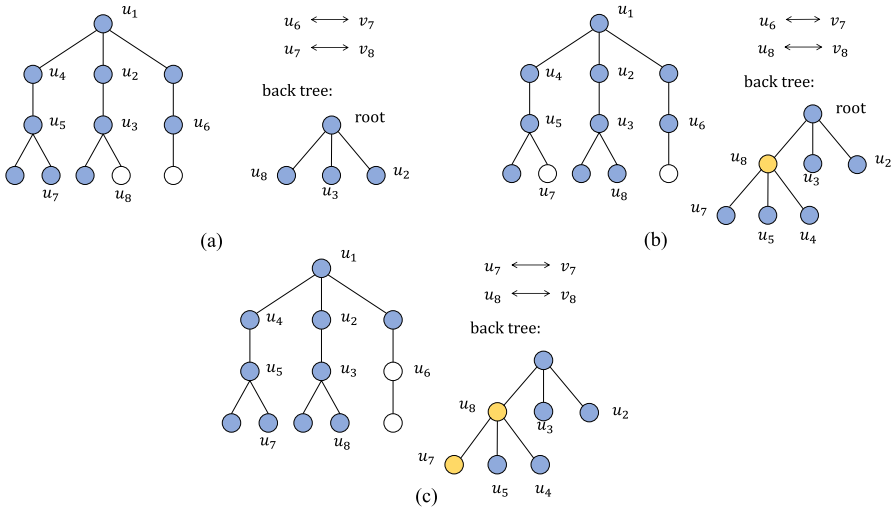


Fig. 10 An example of node snatching. The blue nodes indicate the matched state, the white nodes indicate the need for being matched, and the yellow nodes in the back tree represent nodes that perform the node snatching. **a** u_8 cannot be matched and needs to snatch a matched node. **b** u_8 has snatched v_8, but u_7 cannot be matched and needs to snatch a matched node. **c** u_7 has snatched v_7, both u_7 and u_8 match the correct nodes

has no available candidates, then the branch backtracking method is applied to modify the embedding, in other words, the ancestors of u_8, namely u_3, u_2, and u_1, whose matched nodes are replaced to try the better matching results. However, since v_8 is occupied by u_7, and u_7 has not replaced the matched node in branch backtracking, we still cannot find a match for u_8 even after backtracking to the root node. Therefore, a back tree is constructed, and node snatching is used to force the unmatched u_8 to snatch v_8. Assuming that all nodes in this example satisfy the snatchable nodes conditions, so u_8 and its ancestor nodes u_3 and u_2 are saved in the first layer of the back tree. Then, the node snatching begins in the order of a depth-first traversal, such as [u_8,u_3,u_2].

In Fig. 10b, u_8 has snatched and matched v_8, so u_8 in the back tree is marked as yellow. Suppose that u_7 cannot match after v_8 was snatched, the branch backtracking is applied again. If backtracking to the root does not work, the second layer of the back tree is added as the children of u_8. It consists of u_7, u_5, and u_4. Then the snatching begins from u_7, and now the order is [u_8,u_7,u_5,u_4,u_3,u_2]. Finally, as shown in Fig. 10c, u_7 performs node snatching according to the depth-first traversal order, and it matches the v_7. At this point, both u_7 and u_8 match the correct nodes, and u_6 will be rematched. Because the correct match is found before all nodes of the back tree are traversed, no nodes are skipped.

In summary, the node snatching technique relies on the status of visited nodes to make decisions. After node snatching, if all visited nodes can be matched, the current result is retained, indicating an improved search status. Conversely, a poor status arises when all nodes in the back tree have been reattempted for matching, yet a mismatch persists. In such cases, the problem node is disregarded, and the previously matched pairs are reinstated.

3.8 Multi-level leaf matching

In the leaf-matching algorithm [28], multi-level leaf nodes firstly are separated from query graph Q for subsequent matching individually, as shown in Fig. 9. Then the leaf-matching algorithm performs fast matching of multi-level leaf nodes from the high level to the low level. However, since the separated multi-level leaf nodes are scattered, it is necessary to construct a multi-level leaf node tree, as shown in Fig. 11. The root of the tree is virtual, and the nodes are multi-level leaf nodes. We adopt an in-outside order to construct the tree. For example, the nodes u_1, u_2, and u_3 are adjacent to the core structure in Fig. 9, so they are

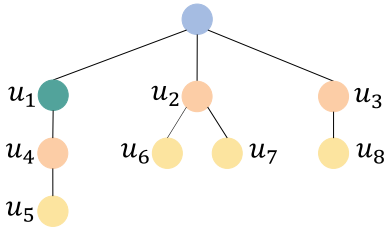


Fig. 11 A multi-level leaf node matching tree. Different colors represent the levels of leaf nodes

saved in the first layer, though they are not the same level. Then, according to the connection relationship, the rest leaf nodes are saved. The matching method is still hierarchical matching with branch backtracking and node snatching. For multi-leaf matching, there is a likelihood that embeddings may contain incorrectly matched pairs. It is expected that these errors will be rectified, and branch backtracking stands as a proven approach to revise incorrect pairs. Consequently, improved branch backtracking is crucial for optimizing the entire matching process.

4 Experimental results

This section presents the experimental results to show the effectiveness and efficiency of our D2BS algorithm. We tested our algorithm on two sets of benchmarks. These benchmarks are extracted from real industrial PCB designs by a company and are suitable to verify the effect of the subgraph matching algorithm for PCB placement. In the first set of benchmarks, each benchmark meets the structure requirement of the exact matching, whose query graph patterns are completely contained in the data graph. That is, the query graph is the subgraph of the data graph. In the second set of benchmarks, there are some differences in the structure pattern between the query graph and data graph, which is an inexact matching situation. The characteristics of the two sets of benchmarks are given in Tables 2 and 3, respectively. In the tables, "#Comp_Q" and "#Net_Q" denote the number of components and the number of nets in query graph Q, respectively. "#Node_Q" is the sum of "#Comp_Q" and "#Net_Q." Similarly, "#Comp_D" and "#Net_D" denote the number of components and the number of nets in data graph D, respectively. "#Node_D" is the sum of "#Comp_D" and "#Net_D." We compared with the state-of-the-art subgraph matching algorithm DAF [25] under the conditions of exact matching and inexact matching. All experiments were conducted on the ubuntu Linux system with an Intel Xeon(R) 2.30 GHZ CPU and 8GB RAM. Due to slight variations in machine state, there are very minor differences in runtime for each run (almost negligible). Nevertheless, we ran each test case 5 times and collected the runtime data by averaging. Besides, the results of each run are consistent, meaning there is no variance in accuracy.

4.1 Comparison with DAF

The first experiment is the comparison between the DAF algorithm [25] and our D2BS algorithm in the metrics of Acc and running time, under the situations of exact matching and inexact matching.

Table 2 Details of the cases and the comparison in matching accuracy and running time (exact matching)

Cases	Statistics						Matching accuracy		Running time	
	#Comp_Q	#Net_Q	#Node_Q	#Comp_D	#Net_D	#Node_D	DAF (%)	D2BS (%)	DAF (s)	D2BS (s)
Case1	20	8	28	21	12	33	100	100	0.011	0.010
Case2	48	15	63	48	15	63	100	100	0.032	0.031
Case3	79	65	144	79	65	144	100	100	0.158	0.126
Case4	54	19	73	112	37	149	100	100	0.050	0.039
Case5	23	10	33	41	18	59	100	100	0.017	0.013
Case6	48	16	64	48	18	66	100	100	0.044	0.037
Case7	558	254	812	558	254	812	–	100	> 3600	5.173
Case8	790	641	1431	790	641	1431	100	100	25.203	12.203
Case9	1080	361	1431	2240	721	2961	100	100	32.350	17.474
Case10	1040	601	1641	1580	1281	2861	100	100	65.537	52.117
Case11	960	301	1261	960	341	1301	–	100	> 3600	61.116
Case12	1118	513	1631	1118	513	1631	–	100	> 3600	18.621

Table 3 Details of the cases and the comparison in Acc and running time (inexact matching)

Cases	Statistics						*Acc*		Running time	
	#Comp_Q	#Net_Q	#Node_Q	#Comp_D	#Net_D	#Node_D	DAF (%)	D2BS (%)	DAF (s)	D2BS (s)
Case13	48	15	63	46	13	59	96.61	100	0.049	0.062
Case14	54	19	73	53	18	71	70.42	100	0.048	0.049
Case15	52	31	83	52	30	82	89.02	100	0.073	0.064
Case16	48	18	66	48	16	64	76.56	100	0.048	0.273
Case17	558	255	813	558	254	812	–	99.02	> 3600	5.342
Case18	559	257	816	558	254	812	–	97.92	> 3600	6.462

4.1.1 Exact matching situation

Table 2 shows the experimental results of twelve benchmarks with the exact matching situation. Our D2BS algorithm achieves 100% Acc for all the benchmarks, while DAF does not. Specifically, for cases 1–6 where the number of nodes in the query graph and data graph is less than 200, both D2BS and DAF can quickly accomplish subgraph matching and achieve 100% Acc . The running time of the two algorithms is generally similar, although our D2BS is slightly quicker than the DAF. For the large-scale cases 7–12, our D2BS still achieves 100% Acc . In contrast, although DAF can also achieve 100% Acc for cases 8–10, the running time of DAF is longer than that of D2BS. Besides, for other cases (i.e., case7, case11, and case12), DAF fails to obtain matching results even after running for more than one hour. Overall, the experimental results demonstrate that our D2BS outperforms DAF under the exact matching situation.

4.1.2 Inexact matching situation

In the PCB reference placement, there may be some differences in the structure pattern between the query graph and data graph, which is an inexact matching situation. Table 3 shows the experimental results of six benchmarks with the inexact matching situation. In these benchmarks, the number of query nodes is more than the number of data nodes; then, Q must contain some patterns that are not in D. As can be seen from the table, our algorithm achieves 100% Acc in cases 13–16 and more than 97% Acc in cases 17–18. In contrast, there is a clear gap in *Acc* between DAF and our algorithm in cases 13–16. Moreover, DAF fails to obtain matching results in cases 17–18, perhaps because DAF does not apply to some graph structures generated from PCB netlist information. As for the running time, our algorithm still obtains matching results within several seconds, which reflects good stability.

4.2 Comparison with an automatic PCB placement tool

To further demonstrate the application and effectiveness of our reference placement method in PCB design, we compared the proposed method with an in-house commercial PCB placement tool. This tool employs a simulated annealing-based algorithm for PCB placement. Due to commercial confidentiality, the specific layout information of the tested cases used in Sect. 4.1 cannot be publicly disclosed; therefore, a different case is used for the experiment in this subsection. This case contains 52 components, and the corresponding result is shown in Fig. 12.

In Fig. 12a, since our algorithm achieves the 100% matching rate, the placement result obtained by our algorithm is consistent with the manual placement template. For this placement result, the wirelength is 581, the number of net crossings is 91, and the runtime is 0.32 s. In contrast, for the results generated by the automatic PCB placement tool, as shown in Fig. 12b, the wirelength is 1000, the number of net crossings is 302, and the runtime is 3.98 s. The experiments demonstrate the effectiveness of our reference placement method in PCB design.

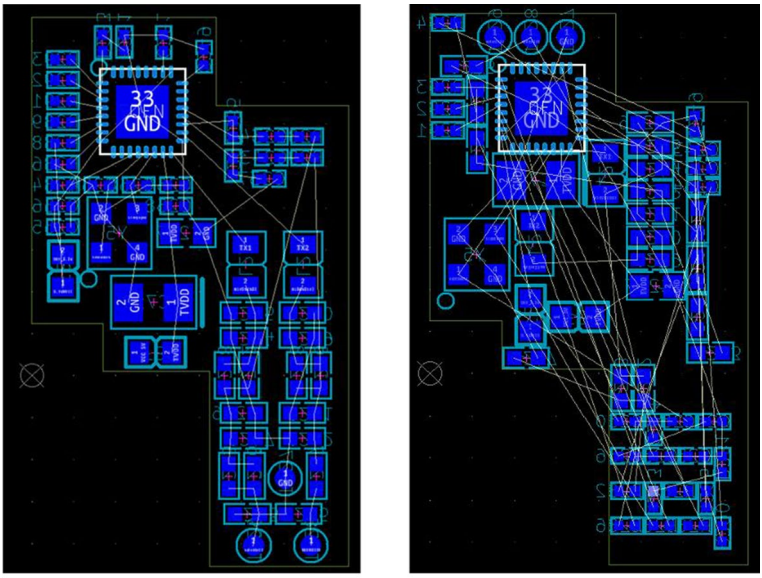


Fig. 12 Placements obtained by our method and an in-house tool. (a) Placement obtained by our method. (b) Placement obtained by an in-house tool

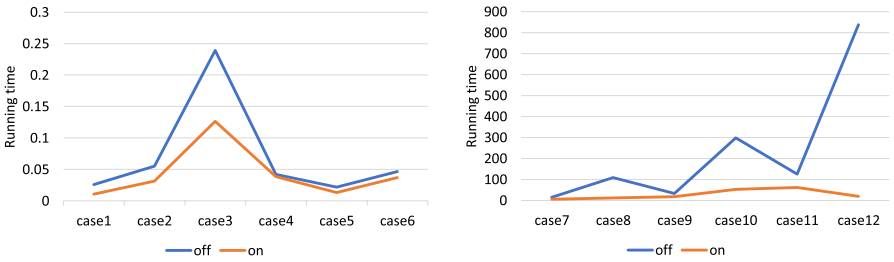


Fig. 13 Comparison in running time for the candidate set screening being turned on and off

4.3 Effectiveness of individual techniques

In this section, we analyze the effects of individual techniques (including the similarity-based candidate set creation and screening, the cut edge-based graph diversity tolerance handling, and the improved backtracking with node snatching) applied in our algorithm.

4.3.1 Effectiveness of similarity-based candidate set creation and screening

The screening of the candidate set aims to accelerate the process of candidate space optimization and matching, as introduced in Sect. 3.3. In this experiment, we validate the effectiveness of this technique by comparing the running time before and after turning off the screening. We use cases 1–12 to conduct the experiment, and the results are shown in Fig. 13. As can be seen from the figure, when the screening is off, the running time increases, which is more obvious in large-scale cases. In the small-scale cases 1–6, there is a steady improvement for each case except case 3, which has a more obvious improvement. The reason for the more noticeable improvement in case 3 is that its netlist contains more same-type components, resulting in a higher number of redundant nodes in the initial candidates. Without the screening technique, it would take more time to search through these candidates. Similarly, for the large-scale cases 8, 10, and 12, they also exhibit more outstanding runtime reduction due to the reason mentioned above. These netlists originally contain too many components and nets with similar features, so the screening technique is essential for candidate space refinement. The screening of candidate set effectively shortens the running time and strengthens the robustness of the proposed algorithm.

4.3.2 Effectiveness of cut edge-based graph diversity tolerance handling

In Sect. 3.5, we propose a cut edge-based graph diversity tolerance handling technique to improve the Acc under the inexact matching situation of the PCB instances. To verify the effectiveness of the technique, we conducted another experiment by turning off the graph diversity tolerance handling in our algorithm. Since the technique is mainly applied in the inexact matching situation, we compare the Acc of cases 13–18. Besides, because case 13 and case 14 generate no cut edge, we exhibit the *Acc* for cases 15–18 in Fig. 14. It can be seen that there appears to be an obvious decrease in Acc for case 15 and case 18 after turning off the graph diversity tolerance handling technique, which demonstrates the effectiveness of the proposed technique. Comparatively, case 16 and case 17 are slightly improved. The reason is that the differences are so tiny between the template and data graph in case 16 and case 17, with few cut edges. We can see that case 16 and case 17 already have high Acc even if without this technique, also indicating they are close to the exact matching

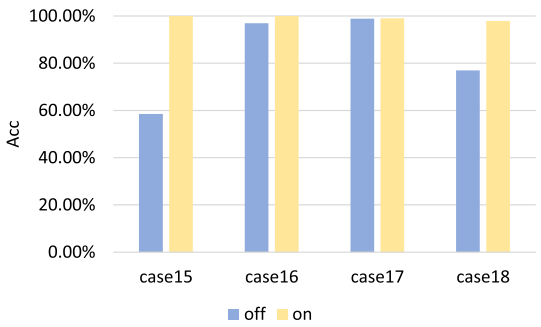


Fig. 14 Comparison in Acc Acc for the cut edge-based graph diversity tolerance handling being turned on and off

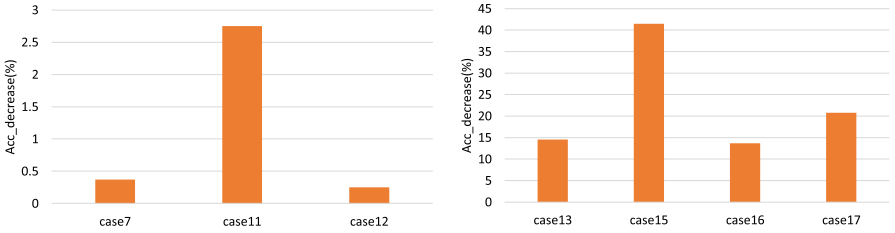


Fig. 15 Decrease in Acc after turning off the node snatching

situation. In summary, with more differences caused by cut edges, the improvement may be more obvious.

4.3.3 Effectiveness of node snatching

In Sect. 3.7.2, we develop the node snatching in the process of the improved backtracking to modify the occupied ideal matching nodes and improve the *Acc*. In this experiment, we verify the effectiveness of node snatching by comparing the improved backtracking without node snatching. As shown in Table 2, cases 1–12 reach 100% *Acc* when the node snatching is turned on. In contrast, when the node snatching is turned off, case 7, case 11, and case 12 experience a decrease in *Acc*, shown in Fig. 15. For the inexact matching cases 13–18, the decrease of *Acc* is more evident. Except for case14, other cases have a sharp decrease in *Acc*, especially case18, which fails to generate any result. Overall, the decrease in *Acc* after turning off the node snatching demonstrates the effectiveness of the node snatching technique.

Table 4 Total number of candidate nodes in different stages

Cases	Initial num.	After screening		After DAG optimization	
		Num.	*R*(%)	Num.	*R*(%)
Case1	133	59	57.9	33	44.1
Case2	1277	802	37.2	265	70
Case3	4498	1511	66.4	148	90.2
Case4	1067	579	45.7	291	49.7
Case5	358	179	50	58	67.6
Case6	1227	763	37.8	261	65.8
Case7	103,371	40,435	60.9	1504	96.3
Case8	449,701	151,001	66.4	14,701	90.3
Case9	426,401	231,201	45.8	116,001	49.8
Case10	931,201	367,801	60.5	69,801	81
Case11	475,201	301,761	36.5	100,961	66.5
Case12	424,075	164,931	61.1	5659	96.6

4.4 Contributions of individual techniques to efficiency

In this section, we analyze the contributions of individual techniques (including the similarity-based candidate set creation and screening, the DAG-aware candidate space optimization, and the multi-level leaf node decomposition) applied in our algorithm to the algorithm's efficiency.

4.4.1 Contributions of candidate nodes filtering

To effectively reduce the search space, we apply the similarity-based candidate set screening presented in Sect. 3.3 and the DAG-aware candidate space optimization presented in Sect. 3.4. Table 4 lists the total number of candidate nodes in different stages, where Num. represents the total number of current candidate nodes and $R(\%)$ represents the percentage of node reduction before and after executing this stage. As can be seen from the table, the two techniques effectively reduce the total number of candidate nodes in all cases. In most cases, the DAG-aware candidate space processing technique filters out a significantly higher proportion of candidate nodes. Conversely, the similarity-based screening, which lacks consideration for the interconnectivity between nodes, serves as a more rudimentary process. Therefore, it leaves behind a certain degree of redundant candidates. However, during the DAG-aware processing stage, owing to the stringent connection constraints, a significantly larger percentage of candidate nodes are eliminated.

4.4.2 Ablation study on the efficiency improvement techniques

Our algorithm incorporates several techniques to reduce runtime, mainly including candidate set screening, DAG-aware candidate space optimization, and multi-level

Table 5 Comparison of runtime reduction by various techniques

Cases	Initial time	w/o screening		w/o optimization		w/o decomposition	
		Time	*R*	Time	*R*	Time	*R*
Case1	0.010	0.032	3.20	0.042	4.20	0.028	2.80
Case2	0.031	0.051	1.65	0.271	8.74	0.043	1.39
Case3	0.126	0.242	1.92	0.215	1.71	0.144	1.14
Case4	0.039	0.041	1.05	0.082	2.10	0.047	1.21
Case5	0.013	0.021	1.62	0.068	5.23	0.021	1.62
Case6	0.037	0.048	1.30	0.612	16.54	0.051	1.38
Case7	5.173	5.582	1.08	5.799	1.12	7.412	1.43
Case8	12.203	106.511	8.73	30.086	2.47	16.234	1.33
Case9	17.474	35.763	2.05	69.316	3.97	28.777	1.65
Case10	52.117	298.632	5.73	508.457	9.76	137.120	2.63
Case11	61.116	127.468	2.09	160.833	2.63	77.369	1.27
Case12	18.621	837.585	44.98	21.914	1.18	28.751	1.54

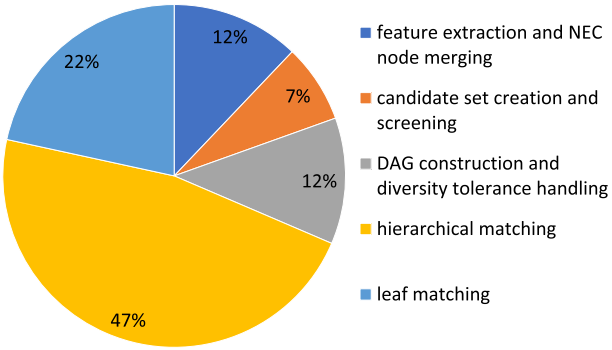


Fig. 16 Running time breakdown of the case11

leaf node decomposition. In this section, we delve into the impact of each of these techniques on runtime reduction. Table 5 presents a comparative analysis, outlining the original runtime, the runtime without a specific technique, and the respective multiples of the original runtime. Here, time represents the runtime in seconds, and R denotes the multiple of the original runtime. As can be seen from the table, all techniques contribute significantly to improving runtime. Broadly speaking, the filtering techniques (candidate set screening and candidate space optimization) are more instrumental in reducing runtime compared to leaf node decomposition. When comparing the two filtering techniques, we observe that DAG-aware candidate space optimization plays a pivotal role in most scenarios. However, in certain cases (such as case 8 and case 12), candidate set screening offers a more substantial time savings. This can be attributed to the significant influence candidate set screening has on candidate space optimization, where unfiltered candidate sets can hamper the performance of the latter.

4.4.3 Running time breakdown

To provide additional insights into time consumption of our algorithm, Fig. 16 shows the running time breakdown of the case11, which is the most time-consuming case for our algorithm. We divide the algorithm into five steps: (1) feature extraction and NEC node merging, (2) candidate set creation and screening, (3) DAG construction and diversity tolerance handling, (4) hierarchical matching, and (5) leaf matching. As can be seen from the figure, nearly 70% of the total running time is spent on the hierarchical matching and leaf matching. Besides, the candidate set screening and the diversity tolerance handling are effective as validated in Sects. 4.3.1 and 4.3.2, but from Fig. 16 we can see that they are not time-consuming.

5 Conclusions

In this paper, we have first converted the PCB netlist information into a graph, where the template that has been placed is regarded as a query graph, and the components and nets to be placed are regarded as a data graph. Then, we modeled the PCB reference placement as a subgraph matching problem. Particularly, we have developed a novel subgraph matching algorithm D2BS with diversity tolerance and improved backtracking to guarantee matching quality and efficiency. The D2BS algorithm is founded on a data structure called the candidate space structure. We built and filtered the candidate set for each query node according to our designed features to construct the candidate space structure. During the candidate space optimization process, a graph diversity tolerance strategy is adopted to achieve efficient inexact matching. Then, hierarchical matching is developed to search the template embeddings in the candidate space structure guided by branch backtracking and matched-node snatching strategies. Based on the industrial PCB designs, experimental results have shown that D2BS outperforms the state-of-the-art subgraph matching method in Acc and running time.

Acknowledgements This work was partially supported by the National Natural Science Foundation of China under Grants 62104037 and 92373207 and the MOST of Taiwan under Grant MOST 110-2221-E-002-177-MY3.

Author contributions ZZ, YL, and MS gave the idea and wrote the main manuscript text. YL, MS, HS, and YX did the experiments. All authors reviewed the manuscript.

Data availability No datasets were generated or analyzed during the current study.

Declarations

Conflict of interest The authors declare no competing interests.

References

1. Su M, Xiao Y, Zhang S, Su H, Xu J, He H, Zhu Z, Chen J, Chang Y-W (2022) Late breaking results: Subgraph matching based reference placement for pcb designs. In: Proceedings of the 59th ACM/IEEE Design Automation Conference, pp 1400–1401
2. Khandpur RS (2006) Printed circuit boards design, fabrication, and assembly. The McGraw-Hill Companies, New York
3. Zhang D, Ren Q, Su D (2021) A novel authentication methodology to detect counterfeit pcb using pcb trace-based ring oscillator. IEEE Access 9:28525–28539. https://doi.org/10.1109/ACCESS.2021.3059100
4. Ismail FS, Yusof R, Khalid M (2012) Optimization of electronics component placement design on pcb using self organizing genetic algorithm (soga). J Intell Manuf 23:883–895
5. Satomi Y, Hachiya K, Kanamoto T, Watanabe R, Kurokawa A (2020) Thermal placement on pcb of components including 3d ics. IEICE Electron Express 17(3):20190737–20190737
6. Badriyah T, Setyorini F, Yuliawan N (2016) The implementation of genetic algorithm and routing lee for pcb design optimization. In: International Conference on Informatics and Computing (ICIC)
7. Kureichik V, Kuliev E (2020) Integrated algorithm for elements placement on the printed circuit board. In: IOP Conference Series: Materials Science and Engineering, vol 734, p 012146

8. Cheng C-K, Ho C-T, Holtz C (2022) Net separation-oriented printed circuit board placement via margin maximization. In: 27th Asia and South Pacific Design Automation Conference (ASP-DAC), pp 288–293
9. Zhang C, Jin H, Chen J, Zhu J, Luo J (2020) A hierarchy mcts algorithm for the automated pcb routing. In: 16th International Conference on Control & Automation (ICCA), pp 1366–1371
10. Kunal K, Dhar T, Madhusudan M, Poojary J, Sharma AK, Xu W, Burns SM, Hu J, Harjani R, Sapatnekar SS (2023) Gnn-based hierarchical annotation for analog circuits. IEEE Trans Comput-Aided Des Integr Circuit Syst 42(9):2801–2814. https://doi.org/10.1109/TCAD.2023.3236269
11. Kunal K, Dhar T, Madhusudan M, Poojary J, Sharma A, Xu W, Burns SM, Hu J, Harjani R, Sapatnekar SS (2020) Gana: graph convolutional network based automated netlist annotation for analog circuits. In: 2020 design, automation & test in Europe conference & exhibition (DATE), pp 55–60. https://doi.org/10.23919/DATE48585.2020.9116329
12. Shang H, Zhang Y, Lin X, Yu JX (2008) Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. Proc LDB Endow 1(1):364–375
13. Zhao P, Han J (2010) On graph query optimization in large networks. Proc VLDB Endow 3(1–2):340–351
14. Han W-S, Lee J, Lee J-H (2013) Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In: Proceedings of the 2013 ACM SIGMOD international conference on management of data, pp. 337–348
15. Zampelli S, Deville Y, Solnon C (2010) Solving subgraph isomorphism problems with constraint programming. Constraints 15(3):327–353
16. Solnon C (2010) All different-based filtering for subgraph isomorphism. Artif Intell 174(12):850–864
17. Kotthoff L, McCreesh C, Solnon C (2016) Portfolios of subgraph isomorphism algorithms. In: International Conference on Learning and Intelligent Optimization, Springer. pp 107–122
18. Ullmann JR (1976) An algorithm for subgraph isomorphism. J ACM (JACM) 23(1):31–42
19. Cordella LP, Foggia P, Sansone C, Vento M (2001) An improved algorithm for matching large graphs. In: 3rd IAPR-TC15 Workshop on Graph-Based Representations in Pattern Recognition, pp 149–159. Citeseer
20. Cordella LP, Foggia P, Sansone C, Vento M (2004) A (sub) graph isomorphism algorithm for matching large graphs. IEEE Trans Pattern Anal Mach Intell 26(10):1367–1372
21. Carletti V, Foggia P, Saggese A, Vento M (2017) Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with vf3. IEEE Trans Pattern Anal Mach Intell 40(4):804–818
22. Battiti R, Mascia F (2007) An algorithm portfolio for the sub-graph isomorphism problem. In: International Workshop on Wngineering Stochastic Local Search Algorithms, Springer. pp 106–120
23. Almasri I, Gao X, Fedoroff N (2014) Quick mining of isomorphic exact large patterns from large graphs. In: 2014 IEEE International Conference on Data Mining Workshop, pp 517–524. IEEE
24. Bonnici V, Giugno R (2016) On the variable ordering in subgraph isomorphism algorithms. IEEE/ACM Trans Comput Biol Bioinform 14(1):193–203
25. Han M, Kim H, Gu G, Park K, Han W-S (2019) Efficient subgraph matching: harmonizing dynamic programming, adaptive matching order, and failing set together. In: Proceedings of the 2019 International Conference on Management of Data, pp 1429–1446
26. Zhu Z, Mei Y, Li Z, Lin J, Chen J, Yang J, Chang Y-W (2022) High-performance placement for large-scale heterogeneous fpgas with clock constraints. In: Proceedings of the 59th ACM/IEEE Design Automation Conference, pp 643–648
27. Zhu Z, Chen J, Peng Z, Zhu W, Chang Y-W (2018) Generalized augmented lagrangian and its applications to vlsi global placement. In: Proceedings of ACM/ESDA/IEEE Design Automation Conference, pp 1–6
28. Bi F, Chang L, Lin X, Qin L, Zhang W (2016) Efficient subgraph matching by postponing cartesian products. In: Proceedings of the 2016 International Conference on Management of Data, pp 1199–1214

Authors and Affiliations

Ziran Zhu[1] · Yilin Li[1] · Miaodi Su[2] · Shu Zhang[2] · Haiyuan Su[2] · Yifeng Xiao[3] · Huan He[4] · Jianli Chen[5] · Yao-Wen Chang[6,7]

✉ Ziran Zhu
zrzhu@seu.edu.cn

Yilin Li
220226192@seu.edu.cn

Miaodi Su
2003200222@fzu.edu.cn

Shu Zhang
zhangshu@fzu.edu.cn

Haiyuan Su
221900122@fzu.edu.cn

Yifeng Xiao
yifengx@usc.edu

Huan He
huan.he@huawei.com

Jianli Chen
chenjianli@fudan.edu.cn

Yao-Wen Chang
ywchang@ntu.edu.tw

[1] National ASIC System Engineering Center, Southeast University, Nanjing 210096, China

[2] College of Mathematics and Computer Science, Fuzhou University, Fuzhou 350108, China

[3] Department of Electrical Engineering, University of Southern California, Los Angeles, CA, USA

[4] Hangzhou Huawei Enterprises Telecommunication Technologies Co., Ltd, Hangzhou 310000, China

[5] State Key Lab of ASIC and System, Fudan University, Shanghai 200433, China

[6] Graduate Institute of Electronics Engineering, National Taiwan University, Taipei 10617, Taiwan

[7] Department of Electrical Engineering, National Taiwan University, Taipei 10617, Taiwan