© JUN 2026 | IRE Journals | Volume 9 Issue 12 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I12-1718966

Survey of Metaheuristics for Steiner Trees, Wire-Length Driven Routing, and Constrained Spanning Tree Problems in VLSI*

JITENDRA SINGH[1], MANTRIPRAGADA SHABDA PYARI[2]
[1,2]*Dept. of Electrical Engineering Dayalbagh Educational Institute Agra, India*

Abstract- ***Routing in Very Large-Scale Integration (VLSI) is a challenging task that involves managing interconnect length, congestion, power consumption, and timing. With the rapid growth of semiconductor technology, traditional algorithms like Integer Linear Programming (ILP) and dynamic programming often become too slow and impractical for large-scale circuits. Overcome this, metaheuristic algorithms such as Genetic Algorithms (GA), Simulated Annealing (SA), Ant Colony Optimization (ACO), and Particle Swarm Optimization (PSO) offer effective solutions. These algorithms are particularly useful for NP-hard problems like Steiner tree construction and wire-length optimization, striking a balance between accuracy and computational effort. This paper provides a comprehensive overview of metaheuristic applications in VLSI routing, analyzing their strengths, limitations, and real-world performance. It also highlights emerging trends like AI-powered optimization and hybrid algorithms. By offering valuable insights, this survey serves as a helpful resource for researchers and engineers navigating the complexities of modern VLSI design.***

Key Words: ***VLSI Routing, Wire-Length Minimization, Steiner Tree Construction, Constrained Spanning Tree, Global Routing.***

I. INTRODUCTION

VLSI routing is an important step in designing circuits, as it directly affects performance, power use, and manufacturability. The main goal of routing is to connect different parts of a circuit while keeping wire length short, reducing power consumption, and avoiding congestion [1]. As technology advances and chip sizes shrink, the number of required connections grows significantly, making routing more complex. Traditional methods for routing, such as shortest-path algorithms, become slow and inefficient when dealing with large circuits [2]. This has led to the use of heuristic and metaheuristic techniques, which can find good solutions in a reasonable amount of time without needing to check every possibility [3].

The routing problem in VLSI can be divided into three main areas: Steiner tree construction, wire-length driven routing, and constrained spanning tree problems [4]. The Steiner tree problem focuses on finding the shortest network to connect a given set of terminals, sometimes adding extra nodes (Steiner points) to reduce total wire length. Wire-length driven routing aims to optimize wire length to improve signal delay and overall circuit performance. Constrained spanning tree problems involve routing under certain design limitations, such as power

limits, signal integrity requirements, or clock synchronization needs. Since these problems are difficult to solve exactly, researchers rely on metaheuristic approaches to find near-optimal solutions efficiently [5].

1.2 Metaheuristic Approaches in VLSI Routing

Many routing problems in VLSI design are NP-hard, which means finding the best possible solution can take an extremely long time, especially for large circuits [6]. Instead of using exact methods, which may be too slow, metaheuristic algorithms are used to quickly find good solutions. These algorithms do not guarantee the best solution, but they offer effective ways to explore many possible designs and choose one that is close to optimal. Inspired by natural processes such as evolution, swarm behavior, and thermodynamics, these techniques have been successfully applied to routing problems in VLSI [7].

Different metaheuristic methods have been used in routing, including Simulated Annealing, Tabu

Search, Ant Colony Optimization, Particle Swarm Optimization, and Genetic Algorithms [8]. Simulated Annealing follows the principle of heating and cooling in metals, allowing the algorithm to avoid getting stuck in poor solutions. Tabu Search remembers previously visited solutions and avoids them, helping to explore new possibilities. Ant Colony Optimization is based on how ants leave pheromone trails to guide others toward the shortest path. Particle Swarm Optimization simulates how birds move in flocks, balancing individual exploration with collective intelligence. Genetic Algorithms use selection, crossover, and mutation, similar to how evolution works, to gradually improve solutions over time [9]. Each of these methods has strengths and weaknesses, making them useful for different types of routing challenges. In particular, Genetic Algorithms have been widely used for optimizing wire length while handling other constraints like congestion and power consumption [10].

II. ROUTING PROBLEMS IN VLSI

Routing in Very Large-Scale Integration (VLSI) design is a critical challenge that affects how well circuits perform, how much power they use, and how reliable they are. Solving routing problems involves making choices to meet different goals while staying within limits [11]. Algorithms like Ant Colony Optimization (ACO), Particle Swarm Optimization (PSO), and Genetic Algorithms (GA) help by finding good solutions quickly. This section looks at key VLSI routing problems, including Steiner tree construction, wire-length driven routing, and constrained spanning tree problems.

2.2 Steiner Tree Construction

Steiner tree construction is important for connecting different points in a VLSI circuit using the shortest possible path. One common method is the Rectilinear Steiner Minimum Tree (RSMT), which only uses straight horizontal and vertical lines, making it suitable for traditional designs. Another method, the Hexagonal Steiner Tree (HST), uses 60-degree connections to reduce wire length in more complex layouts [12].

When there are obstacles, Obstacle-Aware Steiner Trees (OAST) adjust the paths to go around pre-existing wires or blocked areas. This ensures that the design remains practical while keeping wire length low. Algorithms like ACO, PSO, and GA are often used to optimize these trees. They refine the connections step by step to find a balance between short paths and fast results [13].

2.3 Wire-Length Driven Routing

Wire-length driven routing focused on reducing the length of connections to make signals travel faster, save power, and increase the reliability of circuits. In large circuits, global routing methods using ACO, Simulated Annealing (SA), and PSO spread wires efficiently across the chip, preventing congestion and making effective use of space [14].

Some models are designed to avoid obstacles. These algorithms, especially those using ACO and GA, ensure that the shortest possible paths are chosen while avoiding blocked areas. Multi-layer routing is another technique where fewer vertical connections (vias) are used to reduce resistance and improve performance [15].

2.4 Constrained Spanning Tree Problems

Constrained spanning tree problems add extra challenges to routing by setting specific goals like reducing signal delay, limiting power usage, or preventing congestion. Delay-aware minimum spanning trees (MSTs) are useful for minimizing the time it takes for signals to travel, which is important for large circuits [16].

When multiple constraints are involved, more advanced methods are needed. Hybrid algorithms, such as Genetic Algorithm-Tabu Search (GA-TS), combine the broad search ability of GAs with the precise adjustment of TS. This results in faster and better solutions [17].

Using these modern algorithms, VLSI designers can solve routing problems efficiently, ensuring circuits meet today's performance demands. This section provides insights into how these methods are applied in real-world scenarios.

© JUN 2026 | IRE Journals | Volume 9 Issue 12 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I12-1718966

III. METAHEURISTIC TECHNIQUES FOR VLSI OPTIMIZATION

TABLE 1. Overview of Metaheuristic-Based VLSI Routing Techniques.

Type of Metaheuristic	Proposal	Inspiration	Parameter Number / Parameters	Computational Complexity
Genetic Algorithm (GA)	Holland (1975)	Evolutionary natural selection	Population size, Selection rate, Crossover probability, Mutation rate	Medium
Simulated Annealing (SA)	Kirkpatrick et al. (1983)	Metal annealing process	Temperature schedule, Cooling rate, Cost function	High
Ant Colony Optimization (ACO)	Dorigo (1992)	Pheromone-based learning system mimicking ant foraging	Population size, Pheromone factor, Heuristic factor, Evaporation rate	High
Particle Swarm Optimization (PSO)	Kennedy and Eberhart (1995)	Social behavior of bird flocking	Population size, Inertia weight, Acceleration coefficients	Low
Tabu Search (TS)	Glover (1989)	Memory-based iterative search	Tabu list size, Iteration limit, Aspiration criterion	Medium
Hybrid Metaheuristic Approaches	Various researchers	Combination of multiple techniques	Adaptive tuning of parameters, Algorithm selection strategy	Variable

3.1 Genetic Algorithm (GA)

Genetic Algorithms (GA) are inspired by the principles of natural selection and evolution. They work by encoding potential solutions as chromosomes and applying genetic operators such as selection, crossover, and mutation to evolve better solutions over multiple generations. In the context of VLSI routing, GA is widely used in Steiner tree formation and spanning tree constraints, where it helps optimize routing paths by generating multiple candidate solutions and selecting the best based on a fitness function. The ability of GA to explore a vast solution space makes it effective in solving complex, large-scale routing problems.

Input: Terminal set T, connection graph
Output: A rectilinear Steiner tree T

1. Initialize a population of candidate solutions.
2. Evaluate the fitness of each individual based on wire length.
3. **while** termination condition not met **do**
4. Select parents using tournament or roulette wheel selection
5. Perform crossover to generate offspring.
6. Mutate offspring with a small probability.
7. Evaluate the fitness of offspring.
8. Select the best individuals to form the next generation.
9. **end while**
10. Return the best individual as the optimal Steiner tree.

Fig1: Algorithm For Construct Steiner Tree

Explanation

The process begins by creating a "population" of candidate solutions. Each candidate solution represents a potential Steiner tree configuration. We initialize these solutions randomly, creating a diverse starting point.

Next, we evaluate how "good" each candidate solution is. This evaluation, called "fitness," is based on the total wire length of the tree. Shorter wire lengths are better.

Then, we enter a loop that continues until we reach a stopping condition (e.g., a certain number of generations or a satisfactory solution). Inside this loop, we do the following:

1. Selection: We select pairs of "parents" from our population. The likelihood of a solution being selected as a parent is based on its fitness. Solutions with better fitness have a higher chance of being selected. This mimics natural selection, where the "fittest" individuals are more likely to reproduce. We can use methods like tournament selection or roulette wheel selection.
2. Crossover: We combine the "genetic material" of the selected parents to create new "offspring" solutions. This mimics biological crossover, where genetic material is exchanged between parents. The offspring inherit characteristics from both parents.

© JUN 2026 | IRE Journals | Volume 9 Issue 12 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I12-1718966

3. Mutation: We introduce random changes, or "mutations," to the offspring solutions with a small probability. This mimics genetic mutations, which introduce diversity into the population and can lead to new and potentially better solutions.
4. Evaluation: We evaluate the fitness of the new offspring solutions, just like we did for the initial population.
5. Selection (Next Generation): We select the best individuals from the combined pool of parents and offspring to form the next generation. This ensures that the population gradually improves over time.

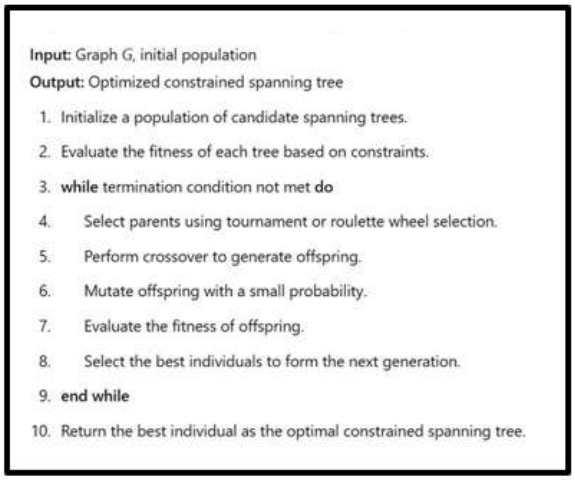

Fig2: Algorithm for Constrained Spanning Tree

Explanation

The process begins by creating a "population" of candidate spanning trees. Each tree represents a potential network design. We initialize these trees randomly, creating a diverse starting point for our search.

Next, we evaluate how "good" each tree is. This evaluation, called "fitness," is based on how well the tree meets our constraints. For example, we might evaluate the tree based on its total delay, power consumption, or congestion.

Then, we enter a loop that continues until we reach a stopping condition (e.g., a certain number of generations or a satisfactory solution). Inside this loop, we do the following:

1. Selection: We select pairs of "parent" trees from our population. The likelihood of a tree being selected as a parent is based on its fitness. Trees that better meet our constraints have a higher chance of being selected. This mimics natural selection, where the "fittest" individuals are more likely to reproduce. We can use methods like tournament selection or roulette wheel selection.
2. Crossover: We combine the "genetic material" of the selected parent trees to create new "offspring" trees. This mimics biological crossover, where genetic material is exchanged between parents. The offspring inherit characteristics from both parents.
3. Mutation: We introduce random changes, or "mutations," to the offspring trees with a small probability. This mimics genetic mutations, which introduce diversity into the population and can lead to new and potentially better trees.
4. Evaluation: We evaluate the fitness of the new offspring trees, just like we did for the initial population.
5. Selection (Next Generation): We select the best trees from the combined pool of parents and offspring to form the next generation. This ensures that the population gradually improves over time, focusing on trees that better meet our constraints.

3.2 Simulated Annealing (SA)

Simulated Annealing (SA) is a probabilistic optimization technique that mimics the annealing process in metallurgy. It allows for occasional acceptance of worse solutions to escape local minima, enabling a more thorough exploration of the solution space. In VLSI routing, SA is particularly effective for wire-length reduction and timing optimization, as it refines routing solutions by continuously adjusting net topologies and interconnections while maintaining design constraints. The slow cooling schedule ensures convergence to a near-optimal solution.

Accept new solution if,

$$P = e^{\frac{-\Delta E}{T}}$$

© JUN 2026 | IRE Journals | Volume 9 Issue 12 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I12-1718966

SA explores the solution space by iteratively modify a current solution, unlike greedy methods, greedy methods, SA allows occasional acceptance of worse solutions to escape local minima. The process involves:

1.Initialize temperature T

SA explores the solution space by iteratively modifying a current solution. Unlike greedy methods, SA **allows occasional acceptance of worse solutions** to escape local minima. The process involves:

1. **Initialize temperature** T_0 and generate an initial solution.
2. **Generate a neighboring solution** and compute the cost difference ΔC.
3. **Accept the new solution probabilistically:**
   $P = e^{-\frac{\Delta C}{T}}$
   If $\Delta C < 0$, accept the solution. If $\Delta C > 0$, accept with probability P.
4. **Gradually reduce temperature** using a cooling schedule.
5. **Repeat until temperature approaches zero.**

Input: Initial routing path P
Output: Optimized wire-length driven routing path

1. Initialize solution P randomly.
2. Set initial temperature T_0 and cooling rate α.
3. **while** termination condition not met **do**
4. Generate a neighboring solution P'.
5. Calculate the cost difference $\Delta E = E(P') - E(P)$.
6. **if** $\Delta E < 0$ **then**
7. Accept the new solution.
8. **else if** $e^{-\Delta E/T} > \text{random}(0, 1)$ **then**
9. Accept the new solution.
10. Reduce temperature: $T = \alpha T$.
11. **end while**
12. Return the best solution P as the optimized routing path.

Fig3: Algorithm for Wire Driven Routing
Explanation:

The process begins with an initial, randomly generated, layout of the wires. This initial layout serves as our starting point, acknowledging that it is not the best possible configuration. Guide the optimization, we establish a starting "temperature" and a "cooling rate." The temperature allows us to accept changes that might temporarily worsen the layout, giving us room to explore different possibilities. The cooling rate dictates how quickly we become more selective, gradually favouring improvements.

Now, the core of the algorithm kicks in. We enter a loop that continues until we have reached a satisfactory solution or a stopping point. Within this loop, we make small adjustments to the current wire layout, creating a slightly modified version. We then assess how these changes affect the total wire length. If the new layout results in shorter wires, we automatically adopt it. However, if the new layout increases the wire length, we do not automatically reject it. Instead, we use a probabilistic approach. We calculate the likelihood of accepting this change based on the current "temperature" and the increase in wire length. The higher the temperature, the more likely we are to accept a worse layout, allowing us to escape local optima.

As we progress through the loop, we gradually decrease the "temperature," making us less likely to accept changes that worsen the wire length. This simulates the cooling process in annealing, guiding us towards better and better layouts.

Finally, once we have completed the loop, we return the best wire layout we've discovered. This layout represents the optimized routing path, achieved by balancing exploration and refinement through the Simulated Annealing process.

3.3 Ant Colony Optimization (ACO)

Ant Colony Optimization (ACO) is a swarm intelligence algorithm that simulates the behavior of ants searching for the shortest path between food sources and their nest. In VLSI routing, ACO is extensively applied in Steiner tree construction and obstacle avoidance, where artificial ants traverse routing graphs and deposit pheromones to reinforce optimal paths. The self-reinforcing nature of ACO allows for dynamic adjustments to routing paths, making it highly effective in scenarios with congestion and variable constraints.

Ant Movement & Solution Construction: Ants traverse the graph, constructing a possible Steiner tree based on a probabilistic rule:

$$P(e) = \frac{(\tau(e))^{\alpha} \cdot (\eta(e))^{\beta}}{\sum (\tau(e))^{\alpha} \cdot (\eta(e))^{\beta}}$$

© JUN 2026 | IRE Journals | Volume 9 Issue 12 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I12-1718966

$\tau(e)$: Pheromone level on edge e.

$\eta(e) = 1/w(e)$: Heuristic information (inverse of wire length).

α, β: Parameters controlling the influence of pheromones and heuristic data.

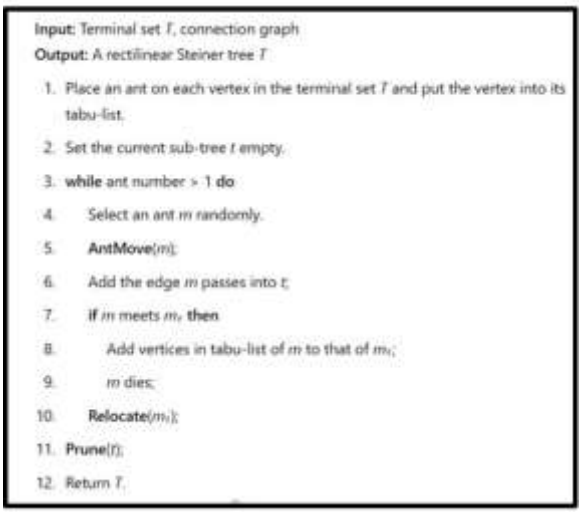

Fig4: Algorithm for Constructed Steiner Tree

Explanation:

The process begins by placing a virtual "ant" on each terminal point in our set of points (T). Each ant starts by marking its initial location as "visited" to prevent revisiting, using a "tabu-list" to keep track. We also initialize an empty sub-tree (t) where the ants will gradually build the Steiner tree.

Then, we enter a loop that continues as long as there are multiple ants still active. Inside this loop, we randomly select an ant (m) to move. The ant then performs a movement step, guided by the pheromone trails and heuristic information, aiming to connect to other terminals. The path this ant takes, the edges it traverses, are added to our growing sub-tree (t).

As the ants move, they might encounter each other. When an ant (m) meets another ant (m1), it means they have connected a portion of the tree. In this case, the ant (m) essentially "dies," and its "visited" locations (tabu-list) are added to the other ant's (m1) knowledge. The remaining ant (m1) then adjusts its position to integrate the newly connected path.

After all ants have either connected or "died," we perform a pruning step on the sub-tree (t). This step removes any unnecessary edges or branches, simplifying the tree and reducing its overall length. Finally, the algorithm returns the resulting rectilinear Steiner tree (T), which represents the optimized network connecting all terminal points.

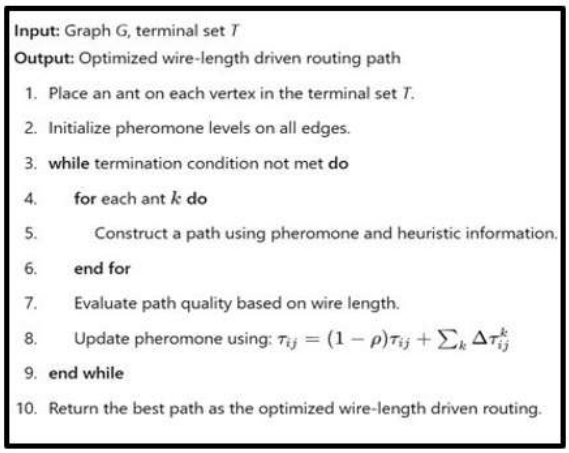

Fig5 : Algorithm for wire driven routing Explanation:

The process begins by placing a virtual "ant" on each terminal point within our graph (G) and terminal set (T). We then initialize the "pheromone levels" on all the edges of the graph. Think of pheromones as a guide; initially, all paths have equal attractiveness

.

We then enter a loop that continues until a termination condition is met (e.g., a certain number of iterations or a satisfactory solution). Inside this loop, each ant constructs a path. Ants don't just move randomly; they use a combination of pheromone information (more pheromone means a more attractive path) and heuristic information (like the distance to the next terminal) to decide their next step.

Once all the ants have constructed their paths, we evaluate the quality of these paths based on wire length. Shorter paths are considered better.

Then, we update the pheromone levels on the edges. This is a crucial step in ACO. We use the formula: τij = (1 - ρ)τij + Σk Δτijk

This formula essentially does two things:

- It reduces the pheromone level on all edges by a factor (1 - ρ), where ρ is the "evaporation rate." This prevents the algorithm from getting stuck in local optima.

© JUN 2026 | IRE Journals | Volume 9 Issue 12 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I12-1718966

- It increases the pheromone level on the edges used by the ants, with the increase (Δτijk) being higher for shorter paths. This reinforces the paths that lead to better solutions.

Finally, after the termination condition is met, the algorithm returns the best path found, which represents the optimized wire-length driven routing.

3.4 Particle Swarm Optimization (PSO)s

Particle Swarm Optimization (PSO) is inspired by the collective behavior of birds flocking or fish schooling. Each particle in the swarm represents a potential solution and updates its position based on both individual and global best-known positions. PSO is widely applied in timing-driven and congestion-aware routing, as it efficiently balances exploration and exploitation, enabling quick convergence to optimal routing solutions. Its adaptability makes it suitable for large-scale circuit designs.

Mathematical Formulation:

Velocity update:

$$v_i^{(t+1)} = wv_i^{(t)} + c_1 r_1 \left(pbest_i - x_i^{(t)}\right) + c_2 r_2 \left(gbest_i - x_i^{(t)}\right)$$

Position update: $x_i^{(t+1)} = x_i^{(t)} + v_i^{(t+1)}$

Where,

v_i is the velocity of particle i,

w is the inertia weight,

p_i is the personal best position,

g is the global best position.

Input: Graph G, initial particle positions and velocities
Output: Optimized routing paths

1. Initialize particles with random positions and velocities.
2. Evaluate the fitness of each particle based on congestion and wire length.
3. Set initial personal best p_i and global best g.
4. **while** termination condition not met **do**
5. **for** each particle i **do**
6. Update velocity using: $v_i = wv_i + c_1 r_1 (p_i - x_i) + c_2 r_2 (g - x_i)$
7. Update position using: $x_i = x_i + v_i$
8. Evaluate the new fitness.
9. Update personal and global bests.
10. **end for**
11. **end while**
12. Return global best g as the optimized routing path.

Fig 6: Algorithm Global Routing

Explanation:

The process begins by initializing a group of "particles," each representing a potential routing solution. We assign each particle a random position and velocity. Think of position as the current route, and velocity as the direction and speed of change for that route.

Next, we evaluate how "good" each particle's current route is. This evaluation, called "fitness," is based on factors like congestion (how crowded the routes are) and wire length (how long the routes are). We want to minimize both.

We then store the best position found by each particle so far (personal best, p_i) and the best position found by the entire group so far (global best, g).

Now, we enter a loop that continues until a stopping condition is met. Inside this loop, each particle adjusts its position and velocity based on the following:

- Inertia ($w*v_i$): The particle keeps moving in its current direction, but with some damping (w).
- Personal Best Influence ($c_1r_1(p_i - x_i)$): The particle is drawn towards its own best position found so far.
- Global Best Influence ($c_2r_2(g - x_i)$): The particle is drawn towards the best position found by the entire group.

© JUN 2026 | IRE Journals | Volume 9 Issue 12 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I12-1718966

Here, r_1 and r_2 are random numbers, and c_1 and c_2 are constants that control the influence of personal and global bests.

The particle's new position is then calculated by adding its updated velocity to its current position.

Fig 7: Algorithm on Construct Steiner Tree

Explanation:
The process begins by creating a "swarm" of particles, each representing a potential solution for the Steiner tree. We initialize each particle with a random position, essentially a random configuration of the Steiner tree. We also give each particle a random velocity, which represents how it will change its configuration.

Next, we evaluate how "good" each particle's current Steiner tree solution is. This evaluation, called "fitness," is based on the total wire length of the tree. Shorter wire lengths are considered better.

We then store the best Steiner tree solution found by each particle so far (personal best, p_i) and the best solution found by the entire swarm so far (global best, g).

Now, we enter a loop that continues until a stopping condition is met. Inside this loop, each particle adjusts its position and velocity based on the following:

- Inertia ($w*v_i$) : The particle maintains some of its current velocity, allowing it to continue exploring its current direction.
- Personal Best Influence ($c_1r_1(p_i - x_i)$): The particle is drawn towards its own best solution found so far.
- Global Best Influence ($c_2r_2(g - x_i)$): The particle is drawn towards the best solution found by the entire swarm.

Here, r_1 and r_2 are random numbers, and c_1 and c_2 are constants that control the influence of personal and global bests.

The particle's new position, representing its new Steiner tree configuration, is then calculated by adding its updated velocity to its current position.

3.5 Tabu Search (TS)
Tabu Search (TS) employs a memory-based approach to prevent revisiting previously explored solutions, helping the algorithm escape local minima. In VLSI routing, TS is highly effective in minimum spanning tree constraints, where it iteratively refines interconnect structures by adjusting spanning tree configurations while respecting design limitations. The use of a tabu list enhances the search efficiency by avoiding redundant calculations.

The probability of selecting a chromosome for reproduction is given by:

$$P_i = \frac{f_i}{\sum_j f_j}$$

where is the fitness of the -th chromosome.

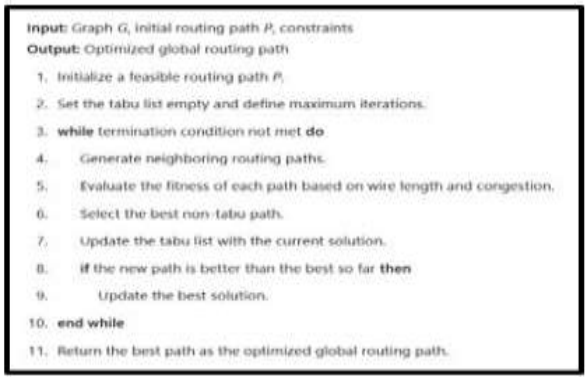

Fig 8: Algorithm on Global Routing

© JUN 2026 | IRE Journals | Volume 9 Issue 12 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I12-1718966

Explanation:
The process begins by creating an initial, working routing path (P) that meets the given constraints. We then set up a "tabu list" to remember the paths we have recently explored, and we define how many iterations (attempts) we will make to find the best solution.

Then, we enter a loop that continues until we reach our iteration limit or find a satisfactory solution. Inside this loop, we generate a set of slightly modified versions of our current routing path. We call these "neighbouring" paths.

Next, we evaluate how "good" each of these neighbouring paths is. This evaluation, called "fitness," is based on factors like wire length (how long the routes are) and congestion (how crowded the routes are). We want to find the path that minimizes both.

We then select the best of these neighbouring paths, but with a twist: we only consider paths that are not on our "tabu list." This prevents us from revisiting solutions we have recently explored, helping us escape local optima.

We then add the path we selected to our "tabu list," marking it as recently explored.

If the new path we selected is better than the best path we have found so far, we update our best path.

Input: Graph *G*, initial spanning tree *S*, constraints
Output: Optimized constrained spanning tree

1. Initialize solution *S* randomly.
2. Set tabu list empty and define maximum iterations.
3. **while** termination condition not met **do**
4. Generate neighboring solutions.
5. Evaluate the fitness of each solution based on constraints.
6. Select the best non-tabu solution.
7. Update the tabu list.
8. **if** the new solution is better than the best so far **then**
9. Update best solution.
10. **end while**
11. Return the best constrained spanning tree.

Fig9: Algorithm on Constrained Spanning Tree
Explanation:

The process begins by creating an initial spanning tree (S), often randomly generated. This serves as our starting point. We also set up a "tabu list" to remember the solutions we've recently explored, and we define how many iterations (attempts) we'll make to find the best tree.

Then, we enter a loop that continues until we reach our iteration limit or find a satisfactory solution. Inside this loop, we generate a set of slightly modified versions of our current spanning tree. We call these "neighboring" solutions.

Next, we evaluate how "good" each of these neighboring solutions is. This evaluation, called "fitness," is based on how well the tree meets our constraints. For example, we might evaluate the tree based on its total delay, power consumption, or congestion.

We then select the best of these neighboring solutions, but with a twist: we only consider

solutions that aren't on our "tabu list." This prevents us from revisiting solutions we've recently explored, helping us escape local optima and explore new possibilities.

We then add the solution we selected to our "tabu list," marking it as recently explored.

If the new solution we selected is better than the best solution we've found so far, we update our best solution.

3.6 Hybrid Metaheuristic Methods
Hybrid metaheuristic methods combine multiple optimization techniques to leverage their respective strengths. For instance, GA + SA integrates GA's exploration capability with SA's fine-tuning ability, while ACO + PSO merges swarm intelligence principles to enhance routing solutions. These hybrid approaches provide improved convergence rates and better-quality solutions, making them highly valuable in complex VLSI routing scenarios.

© JUN 2026 | IRE Journals | Volume 9 Issue 12 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I12-1718966

IV. CONCLUSION

In the realm of Very Large-Scale Integration (VLSI) design, efficient routing is paramount, and this study examined the effectiveness of several metaheuristic algorithms in tackling core routing challenges: constructing Steiner trees, minimizing wire length, and handling constrained spanning trees. Each method brings its own set of strengths and weaknesses to the table, influencing their suitability for different scenarios [18].

1. Steiner Tree lgorithms: A Comparative Look

Algorithm	Strengths	Weaknesses
GA	Excellent at exploring diverse routing options.	Can be slow to converge, especially with complex problems [19].
ACO	Highly effective for Rectilinear Steiner Minimum	Performance hinges on precise parameter tuning [20].
Algorithm	Strengths	Weaknesses
PSO	Quick to find solutions, good for dynamic environments.	Prone to getting stuck in suboptimal local solutions [21].

- In simpler terms: Genetic Algorithms (GA) are like casting a wide net, finding many solutions but taking time. Ant Colony Optimization (ACO) excels in specific scenarios like RSMT, but fine-tuning is crucial. Particle Swarm Optimization (PSO) is fast, but it might miss the best overall answer.

2. Wire-Length Optimization: Balancing Efficiency and Congestion

Algorithm	Wire-Length Reduction	Routing Congestion
SA	Moderate	Low [22]
PSO	High	Medium [23]
ACO	Very High	Low [24]

- In simpler terms: Simulated Annealing (SA) provides a balanced approach. Particle Swarm Optimization (PSO) reduces wire length significantly but can increase congestion. Ant Colony Optimization (ACO) shines in minimizing wire length without causing congestion issues.

3. Constrained Spanning Trees: Meeting Design Demands

Algorithm	Solution Quality (Constrained Spanning Trees)	Computational Complexity
GA-PSO	High	High [25]
TS	Medium-High	Medium [26]

- In simpler terms: Combining GA and PSO yields high-quality solutions, crucial when dealing with limitations like power consumption, but it's computationally intensive. Tabu Search (TS) offers a good balance between solution quality and computational effort.

Overall Reflection

The optimal algorithm choice is heavily dependent on the specific requirements of the VLSI routing task. ACO stands out for its effectiveness in wire-length reduction and Steiner tree construction, particularly in complex designs. Hybrid GA-PSO and TS are valuable when constraints are paramount. Looking ahead, integrating these algorithms with multi-objective optimization techniques and incorporating deep learning for heuristic tuning could further enhance their adaptability to the ever-evolving landscape of VLSI design [27].

REFERENCES

[1] M. A. Breuer and A. D. Friedman, Diagnosis and Reliable Design of Digital Systems, Computer Science Press, 1976.

[2] S. Hassoun and T. Sasao, Logic Synthesis and Verification, Springer, 2002.

[3] W. J. Dally and J. W. Poulton, Digital Systems Engineering, Cambridge University Press, 1998.

[4] L. Scheffer, L. Lavagno, and G. Martin, EDA for IC Implementation, Circuit Design, and Process Technology, CRC Press, 2006.

[5] G. Gielen and R. A. Rutenbar, "Computer-aided design of analog and mixed-signal integrated circuits," Proc. IEEE, vol. 88, no. 12, pp. 1825–1854, 2000.

[6] N. Sherwani, Algorithms for VLSI Physical Design Automation, Springer Science & Business Media, 2012.

[7] J. Cong and S. K. Lim, "Edge separability-based circuit clustering with application to multilevel circuit partitioning," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol. 23, no. 3, pp. 346–357, 2004.

[8] M. Sarrafzadeh and C. K. Wong, An Introduction to VLSI Physical Design, McGraw-Hill, 1996.

[9] T. Ohtsuki (Ed.), Advances in CAD for VLSI, North-Holland, 1986.

[10] R. H. J. M. Otten and R. K. Brayton, "Planning for performance," in Proc. 35th ACM/IEEE Design Automation Conf., 1998, pp. 122–127.

[11] H. Yao and M. D. F. Wong, "A Steiner tree based algorithm for simultaneous escape routing and layer assignment," in Proc. ACM/IEEE Int. Symp. Physical Design, 2006, pp. 189–196.

[12] [M. B. Tahir and S. M. Saeed, "Simulated annealing based VLSI global routing for interconnect length and congestion optimization," Int. J. Comp. Appl., vol. 107, no. 7, pp. 10–15, 2014.

[13] D. Z. Chen and H. Wang, "Oblivious routing for Steiner trees in VLSI design," J. Algorithms, vol. 55, no. 1, pp. 1–19, 2005.

[14] K. L. M. Leung and T. F. Gonzalez, "Fast algorithms for VLSI routing using Steiner trees," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol. 15, no. 9, pp. 1093–1100, 1996.

[15] G. A. Sigl, K. Doll, and F. M. Johannes, "Analytical placement: A linear or a quadratic objective function?," in Proc. 28th ACM/IEEE Design Automation Conf., 1991, pp. 427–432.

[16] C. Chu and Y. C. Wong, "FLUTE: Fast lookup table based wirelength estimation technique," in Proc. IEEE/ACM Int. Conf. Comput.-Aided Design, 2004, pp. 696–701.

[17] H. H. Yang and D. F. Wong, "Efficient Steiner tree construction based on a new exact algorithm," in Proc. ACM/IEEE Design Automation Conf., 2000, pp. 280–285.

[18] M. J. Hutton et al., "FPGA placement and routing challenges," IEEE Design & Test of Computers, vol. 18, no. 2, pp. 12–22, 2001.

[19] C. A. C. Coello, "Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey," Computer Methods in Applied Mechanics and Engineering, vol. 191, no. 11–12, pp. 1245–1287, 2002.

[20] M. Dorigo and T. Stützle, Ant Colony Optimization, Cambridge, MA: MIT Press, 2004.

[21] J. Kennedy and R. Eberhart, "Particle swarm optimization," in Proc. IEEE Int. Conf. Neural Networks, 1995, pp. 1942–1948.

[22] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," Science, vol. 220, no. 4598, pp. 671–680, 1983.

[23] Y. Shi and R. C. Eberhart, "Parameter selection in particle swarm optimization," in Proc. Int. Conf. Evolutionary Programming, 1998, pp. 591–600.

[24] C. Solnon, "Ant colony optimization for constraint solving," Artificial Intelligence, vol. 174, no. 8, pp. 643–671, 2010.

[25] F. Xhafa and A. Abraham, "Metaheuristics for scheduling in distributed computing environments," Studies in Computational Intelligence, vol. 146, pp. 1–43, 2008.

[26] F. Glover, "Tabu search—Part I," ORSA Journal on Computing, vol. 1, no. 3, pp. 190–206,1989.

[27] T. Chen et al., "A survey of deep learning for VLSI physical design automation," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 10, pp. 2107–2122, 2020.