ISSN: 2395-1680 (ONLINE)
DOI: 10.21917/ijme.2025.0339

# A NOBLE RECTILINEAR STEINER TREE WITH OBSTACLES USING PARALLEL DQN: NRST

**Chittaranjan Mohapatra and Nibedita Adhikari**
*Department of Computer Science and Applications, Utkal University, India*

***Abstract***

***Given a set of pins and obstacles in a Very-Large-Scale Integration (VLSI) chip layout, the goal is to develop an optimal routing path with minimal wire length. This work construct Obstacle Avoidance Rectilinear Steiner Minimal Tree (OARSMT) using a deep Q-learning approach, a type of reinforcement learning. It employs union-find data structure, parallel Deep Q-Network and Adam optimizer to train an agent to determine the optimal connection between pins. The DQN approximates Q-values, which reflect the likelihood of selecting an edge. Connections with higher Q-values are those that are obstacle-free, have lower weight values, and favors connections that share common paths. The DQN takes the help Kruskal's algorithm to construct a rectilinear steiner tree with the above connection constraints. The approach uses multi-threading during the training to handle large datasets. The proposed model returns shorter wire lengths with improvement of 5% for obstacle-based benchmark data. The model also achieves 9.8% less training time on an average due to the parallelization of the DQN. The proposed approach realizes an 85.3 % increase in reward gain than other approaches. The developed method achieved the objective and can attain superior performance not only in VLSI physical design but also in various obstacle based routing.***

***Keywords:***

***Deep Reinforcement Learning, Adam Optimizer, OARSMT, VLSI, Physical Design and Routing***

## 1. INTRODUCTION

A rectilinear steiner minimum tree construction problem of VLSI pins is a NP- hard problem [1]. The Obstacle Avoidance Rectilinear Minimum Steiner Tree (OARSMT) problem is a graph optimization problem. A set of pins and non-pins of a VLSI chips are considered as set of nodes of the graph. The non-pins are the corner points of the obstacles which can be used for the pins connections. The graph also contains a set of obstacles. The goal is to connect all the pins such that the connecting edge will not face any obstacles.

The machine learning field is trending toward finding a well-formed and accurate solution to most of the problems [2].This encourages us to develop obstacle avoidance routing using Deep Reinforcement Learning (DRL), where an agent is trained to carry out a certain task by learning from itself with rewards and punishment [3]. The agent selects some action from a set of actions for different states. The action is applied on an environment and the agent is updated as per the outcomes of the action. DRL is typically used in dynamic situations when it is impossible to predict what would work best ahead of time [3], [4]. For example a graph optimization problem requires a more dynamic and adaptable solution [5]. The Q-learning is a reinforcement learning technique to improve the quality of a solution by training [6].

This paper is going to use deep Q-learning, gradient descent, and the Adam optimizer to fine-tune the connections in the very well-known OARSMT problem. It takes a VLSI layout as an input with pins and obstacles data. The output is an OARSMT. It designs a feedforward neural network that selects a Q-value for a particular edge. The Q-value is the probability of an edge towards an optimal solution. The action is accomplished with the help of Kruskal's algorithm, where an MST is created by avoiding obstacles. The gradient descent and Adam optimizers are used to calculate the loss and update the Q-network. Sometimes a disconnected tree is obtained for heavily obstructed areas, where the Q-learning agent might be struggling to find a feasible solution. So an initial solution is created, and a new heuristic approach is designed to connect all nodes of the graph successfully. This refined solution is used for the training only to find the optimal OARSMT.

The structure of the paper is as follows: The Section 2 provides a summary of recent relevant publications. The Section 3 presents the problem statement. The Section 4 explains the proposed algorithm, while the Section 5 offers complexity analysis of the algorithms. The Section 6 covers the results and compares them to existing literature. Lastly, the concluding remarks for the current work are presented in the Section 7.

## 2. LITERATURE SURVEY

The OARSMT is solved using common optimization methods such as PB Sat, genetic algorithms and physarum-inspired optimization algorithms [7] - [9]. Some recent researches based on DRL technique to solve challenges akin to these are included in this section.

A DQN (Deep Q-network) is a combination of Q-learning and deep neural networks. Hasselt et al. [10] proposed Double DQN (DDQN) that uses the DQN algorithm to enhance Double Q-learning [11] Both DQN and Double DQN were compared and found that DQNs greedy policy performed better than DDQN. Liao et al. [12] addressed the routing problem by combining reinforcement learning and deep learning. A DQN router was proposed that established standards for solving pin decomposition. It modeled the circuit as a grid graph from which information was fed into the DQN router and outperformed the conventional $A^*$approach. It solved the unfair distribution of routing resources and optimized wire length while decentralizing routing resources. This optimization strategy produced high-quality global routing results without overflow.

Liu et al. [13] developed a reinforcement learning-based algorithm that is better in terms of quality and runtime for small to medium-sized networks. It may not handle problems with changing constraints or dynamic environments, which pose to be a substantial challenge. Yan et al. [14] designed a framework to find a steiner tree of minimum weight in a graph that connects a set of pins. The DRL based DQN and graph embedding methods were used in this work. It encrypted path-related data from a specified collection of steiner tree problem instances in order to retrieve solutions. It bears overhead of intensive computation with a heavy demand for computational resources. A space partition technique is employed by using attention-based policy parameter optimization and training schemes with different stopping criteria by Wang et al. [15]. It reduced computational and memory overheads compared to traditional algorithms for handling large instances. This problem faces a challenge due to the lack of publicly available problem set repositories with diverse sizes and constraints.

Lin et al. [16] developed DRL based edge embedding model and Multi-Source Dijkstra based steiner tree which has a slow convergence rate. The development of a better solution to handle obstacles is still in demand to improve the quality of convergence.

The proposed method targets to reduce the wire length with less computation overhead and offers an outperform reward growth rate using a Kruskal Based DRL method and Adam Optimizer. The suitability of this proposed method is discussed in the following sections.

## 3. PROBLEM STATEMENT

Let consider a graph $G(V,E)$ contains a set of vertices $V$ and a set of edges $E$. All vertex represent a VLSI circuit's pins, and edges represent the connection between two pins. There are three types of connections between two adjacent pins in a graph. These connections aim to discover shared paths among nearby nodes and get minimal steiner points. These three types of connections are shown in the Fig.1.

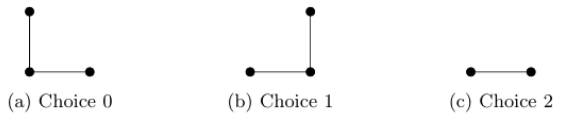


Fig.1. Types of Connection among Pins

If $u$ and $v$ represent two pins or nodes of a Graph $G$, the overall objective function aims to minimize the distance between these pins. It uses the standard Manhattan distance to compute the distance as shown in the Eq.(1). Here $x_1, y_1 \in u$ and $x_2, y_2 \in v$ are the coordinates of the pins respectively.

$$\min \text{Wirelength} = \sum_{(x_1,y_1),(x_2,y_2)} \left( |x_1 - x_2| + |y_1 - y_2| \right) \quad (1)$$

An edge $(u,v) \in G$ represent states where actions correspond to three types connections between two nodes. $R$ denotes a set of obstacles. A state-action pair combines a specific edge $(u,v)$ and an action $a$. A reward value is obtained based on criteria such as minimizing edge weight $w(u,v)$ and ensuring obstacle-free paths $((u,v) \cap R = \emptyset)$. The reward value can be a negative value if the criteria could not be passed. The Q-values are calculated using the current Q-value $Q[(u,v)][a]$ and the projected Q-value for the Next state-action pair $(u'',v'')$ and action $a''$ where $(u'',v'')$ represents the next state and $a''$ represents the next action. Thus, the objective function aims to maximize Q-values within the Deep Q-Network (DQN) for targeted state-action pairs as shown in the Eq.(2). The $\propto$ and $\gamma$ are the learning and discount factor respectively. The $\propto$ is used in the optimization process and the $\gamma$ represents difference in future and present rewards.

$$Q[(u,v)][a] \leftarrow Q[(u,v)][a] + \alpha \times \left( \text{Reward} + \gamma \times \max_{a'} Q[(u',v')][a'] - Q[(u,v)][a] \right) \quad (2)$$

## 4. PROPOSED METHOD

The Lin et al. [16] method applies DRL for bridge embedding, but in this work a steiner tree is constructed using a Kruskal based DRL approach. The Kruskal is applied to the environment to build the OARSMT. The complete process has three step, initial solution, training and testing. The testing method is similar to training process with one iteration. So, the initial solution, DQN training model and its working principles are discussed in this section.

First, an initial OARSMT is built to get the minimum number of pins and non-pins for connecting all pins. A Delaunay triangulation approach is used to create a graph of a given set of pins. The edges of the graph are sorted in descending order of their distance. The union and find data structures are used to create a minimum spanning tree of the created graph. The steiner point is found between two pins during the construction of the tree. There are three types of rectilinear paths. When one type faces any obstacle, it finds the alternative. If no path is possible the path is discarded. Additionally, the connection of two pins avoids cross connections and prefers sharing paths to reduce the number of steiner points. The procedure is mentioned in Algorithm 1.

**Algorithm 1: Obstacle Avoidance Rectilinear Steiner Tree**

**Input:** Set of Pins P and Obstacles O

**Output:** Obstacle Avoidance Rectilinear Steiner Tree

```
1. E = Delaunay Triangulation (P)
2. Create a Grid A=zeros(x_max,y_max)
3. Set A[i][j] = ∞ where i,j ∈ O
4. for v ∈ P do
       a. Makeset(v)
5. end for
6. Q=MinPriorityQueue(E)
7. Cost=0
8. while Q do
       a. e(u,v)=Delete(Q)
       b. if Find(u) != Find(v) then
              i. s=Find Steiner Point(u,v)
              ii. if s ∈ A where A ≠0 then
   continue
              iii. end if
              iv. if e(u,v) intersect any path and s∉ A
                  where A > 0 and A < ∞ then
   continue
              v. end if
              vi. if e(u,v) ∈ A where A == ∞ then
   continue
              vii. end if
              viii. Union(u,v)
```

```
                ix.  Cost = Cost + Cost_(e($u$,$v$))
            c.  end if
    9.  end while
    10. return Tree
```

The initial solution of the proposed work is different than others because it uses only pins. The algorithm fails to create a connected graph, or MST, if there are heavy obstacles between two nodes. So it forms a disconnected tree. A heuristic is developed to connect all forests of the disconnected tree in Algorithm 2. The objective is to detect and connect outliers. It finds the number of connected components using disjoint data structures. Next, it finds the closest forest and connects them with additional points. These additional points are the non-pins from the obstacle boundaries. Finally, these non-pins are added to the graph that is used to optimize.

**Algorithm 2: Connect Component Heuristic**

**Input:** OARSMT

**Output:** Set of nodes and edges

```
1. Find the component set  from the obtained OARSMT
2. List out all leaf nodes with among all component set of OARSMT
3. Determine the shortest distance between the leaf nodes of two adjacent components
4. Find the obstacles that lie between the leaf nodes of closest component
5. Construct a Graph using Delaunay Triangulation of the points of the obstacles and leaf
6. Find the shortest path between the leaf nodes in the Delaunay graph.
7. Include set of nodes and edges belonging to the shortest path into the original graph
```

## 4.1 PROPOSED NRST ARCHITECTURE

The proposed DRL method is called NRST (Noble Rectilinear Steiner Tree) which objective is to develop an OARSMT on an environment for a set of actions. A feed forward neural network with two fully connected layers is being built for approximating the Q-value function for the DRL algorithm. The is a Deep Q-Network (DQN) which assigned the task to different tensors of the tensor flow model. This DQN runs in parallel using multi-threading approach to handle massive pins and obstacles. A block diagram of the NRST agent training process is shown in the Fig.2 which shows the flow of data among the components. The agent has a central role in a reinforcement learning approach because all operations are carried out by the agent. So the agent is the hub of the architecture.

- **DQN:** The DQN architecture has four layers. A state is an edge between two pins. This is an input feature vector of the neural network, the coordinates of the two end points. The action is selected by using the policy gradient method which is a probability distribution over actions for a given state. When the input feature is fed to the Q-Network and action is sampled which results in a set of a Q-value. Two hidden dense layers use the Rectified Linear Unit (ReLU) activation function [17] and sample Q-values for each action and pass them to the output layer. The first fully connected (dense) layer of the neural network has 64 units. This layer performs matrix multiplication of the input with its weights, followed by applying the ReLU activation function.
- **Target DQN:** The target DQN is a copy of the DQN which is used to select the target state. A separate neural network is used to reduce the selection of the same state.

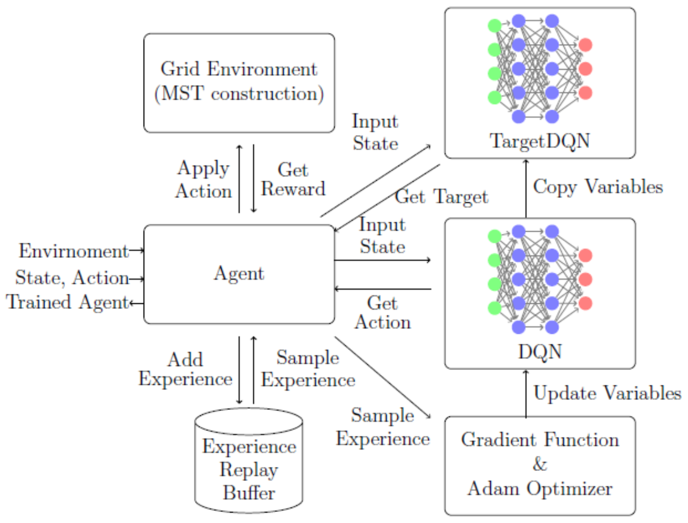


Fig.2. NRST Training Model

- **Grid Environment:** The grid is a layout of nodes and obstacles in the graph. An action of the agent is to construct OARSMT. An action is rewarded with a score of positive value corresponding to the type of edge used in the OARSMT otherwise the action is punished with a negative value. The punishment means it will have less probability of selection and vice versa.
- **Experience Replay Buffer:** A Q-Network stores data in an Experience Replay Buffer, unlike a Q-table. The buffer stores the information like state, action, reward, and target state in a tuple called experience. Its primary tasks are storing and sample experiences. It is implemented using a priority queue data structure. The agent samples a batch of experiences (mini-batches) randomly for optimizing the Q-network parameters during training. The random sampling breaks the temporal correlation between consecutive experiences and reduces the risk of over fitting to recent experiences.
- **Adam Optimizer:** The optimization of the input weight and other network parameters are performed by the Adam (Adaptive Moment) optimizer. The gradient of loss is calculated for a set of current and target sets. The loss is optimized to attain the minimum wire length by updating the Q-Network parameters.

## 4.2 NRST ALGORITHM

The working principle of the NRST Algorithm is mentioned in Algorithm 3. The inputs are graph and grid environments, and the output is a trained deep Q network. The objective is to optimize different types of connections in the steiner tree. The first step is to initialize DQN and Experience Replay Buffer. In every episode (epoch) a temporary graph and grid environment *T* are used to build an intermediate OARSMT with a connected set of edges. The size of $T$ is the maximum of $x$ and $y$ coordinate values from the pins or non-pins of the graph.

**Algorithm 3:**

**Input:** Graph $G = (V, E)$and Grid $T$

**Output:** Trained Deep Q- Network

```
Assumption: State = E , Actions A = [0, 1, 2]
    Initialize $DQN = FFN(s \in S, h_1, h_2, a \in A)$ , Replay Buffer D , $\varepsilon, \alpha, \gamma, \theta$
    for each episode do
        Initialize Grid $T\left[\max_{(x_i,y_i)\in V} x_i + 1\right]\left[\max_{(x_i,y_i)\in V} y_i + 1\right] = 0$ and Graph $G_t = G$
        Use Kruskal's algorithm to find the MST of $G_t$with edges $E_{MST} \subseteq E$
        for each edge $e \in E_{MST}$ do
            $s_t = e$ , Select action
            $a_t = \begin{cases} \sim random(s_t, a) \; if \; random\_value < \varepsilon \\ \operatorname{argmax}_{a\in A}(s_t, a) \quad Otherwise \end{cases}$
            Compute Reward
            $r = \begin{cases} 1, if \; a_t = Connection \; type \; of \; e \\ -1, \quad Otherwise \end{cases}$
            Compute Target state
            $s_{t+1} = g(s_t, a_t) = \max_{a'}(Q[e'][a'])$
            Create an Experience $P_t = (s_t, a_t, R, s_{t+1})$
            $D \leftarrow D \cup \{P_t\}$
            if $|D| \geq \min\_size$ then
                Sample Mini-batch
                    $\text{Mini} - \text{batch} = \{(s_i, a_i, r_i, s_i')\}_1^B \sim D$
                Compute Loss
                $L(\theta) = \frac{1}{|B|}\sum_{(s_i,a_i,r_i,s_i')\in B}\left(Q(s_i, a_i; \theta) - \left(r_i + \gamma \max_{a'} Q(s_i', a'; \theta^-)\right)\right)^2$
                Update Parameters $\theta \leftarrow \theta - \alpha\nabla_\theta L(\theta)$
            end if
        end for
    end for
```

- **Action Selection:** The $\varepsilon$-Greedy Technique is used for action selection by generating a random value. The $\varepsilon$ value is considered 0.1 to have a better exploration and exploitation. It results in positive convergence of the DQN. The action $a_t$ at t iteration is selected either randomly or the maximum probability distribution value from the Q-Network whose input is a state.
- **Reward Calculation:** The reward is collected from the grid environment by applying action to it. The action is to construct the OARSMT. The reward value r is awarded to (S_t,a_t ) which is set to 100 if the connection type matches the selected action otherwise -1.
- **Target state selection:** The target state is the edge followed by the current edge with maximum Q-value. Let the two end points of a state at the *t* iteration is $S_t$ with nodes $u$ and $v$ i.e. $S_t = (u,v)$. Let $(u'',v'')$ are the set of edges outgoing from v except the edge $(u,v)$. So the target state $S_{t+1}$ is another edge from $(u'',v'')$ having maximum action value. In the algorithm $(u,v)$ and $(u'',v'')$ are mentioned as $e$ and $e''$ respectfully. The $a''$ is a set actions belonging to $(u'',v'')$.
- **Q-Network Update:** An experience $P$ is a tuple of four values that is $P = (S_t,a_t,r_t,S_{t+1})$. This is created and added to replay buffer. A mini-batch of size B is randomly sampled from the Replay Buffer D if there are enough value of a specific size as per the Eq.(3). Here, ~ denotes random sampling from the replay buffer.

$$\text{Mini-batch} = \{(s_i, a_i, r_i, s_i')\}_{i=1}^{B} \sim D \tag{3}$$

A gradient descent function updates the network parameters $\theta$ with an objective to maximize the Q-value. The loss function is computed with respect to $\theta$ signifying the difference between the current and target state. The expectation of the Q-value at iteration $i$ is computed by the gradient descent function as per the Eq.(4). Here $\nabla_\theta L$ is the gradient value with respect to the loss of the Q-Network. This updates all the network parameters.

$$\nabla_\theta L(\theta) = \mathrm{E}\left[\sum_{i=0}^{B} \nabla_\theta Q_i \cdot r_i\right] \tag{4}$$

- **Parallelism and Synchronization:** This approach uses multiple threads to process different epochs of the learning method. These epochs are randomly assigned to these threads as parallel tensors. The parallel allocation of tensor may fall in deadlock. So synchronization of Tensor Flow operations is essential to controlling parallelism. However, Locks are maintained around the Tensor Flow model ensuring thread safety and avoiding race conditions.

## 5. RUN TIME ANALYSIS

In this section, efforts are made to determine the time complexity of the proposed algorithm based on its serial execution. The following section demonstrates that parallel execution is significantly faster in practice. Additionally, a proof is provided to compute the time taken for checking obstacles during the construction of an MST.

***Theorem 1*:** Let $G(V,E)$ be the graph and $T$ is the grid representing environment. If each obstacle in $T$ is represented with a two dimensional sub-array and vertex are represented in a single cell of the grid. The time complexity of OARSMT is O (E log*E* ) including the checking of obstacle in the path between two adjacent nodes.

***Proof:*** The OARSMT problem checks obstacles during construction of MST using Kruskal's algorithm whose time complexity is $E$ log$E$ [18]. The implementation process used the np.any() library function for a single connection between two coordinates. All the connections mentioned in the Fig.1 are rectilinear in nature. The search window is either the x-axis value or the y-axis value of the Grid Graph which is a liner search. Hence, the np.any() is linear with respect to the size of the window [19]. So, the checking time of obstacles during the construction of MST is $O(E)$. Hence, the overall time complexity of OARSMT is $(E\log E)$ .

***Theorem 2:*** For a graph with $E$ - number of edges, $V$ - number of vertices, $T$ - number of epochs, with $B$ - batch size used for updating the DQN, the time complexity of the Algorithm 3 can be expressed as $(T\times(E\log E+V\times B\times C))$.

***Proof:*** The proposed method's training is primarily driven by the computation of the minimum spanning tree using Kruskal's algorithm and DQN operation in each episode. As per the Theorem 1, the time complexity OARSMT is $O(E\log E) = O(V\log V)$. The OARSMT returns exactly $V$-1 edges for V vertices. The edges of OARSMT serve as input to DQN. So all operations take $O(V)$ time. The gradient function updates the parameters with a time complexity of $O(B\times C)$ where $C$ and $B$ represent the complexity of computing the gradient and the batch size respectively[20]. Combining all for T episodes, the overall time complexity of the Q-Network training function is $(T\times(E\log E+V\times B\times C))$

## 6. EXPERIMENTAL RESULTS

The experiment is conducted on a Google Colab Pro environment which is a Google Cloud service providing access to NVIDIA Tesla T4 or P100 GPUs, up to 50 GB of RAM, and a Linux-based operating system. The deep Q-Network is executed in parallel using Tensor Flow library and multi-threading. This development is carried out using Python language through a Jupiter Notebook interface. Next, fifteen obstacle based DIMACS Challenge Benchmark instances are used from [21] to test the wire length by avoiding obstacles. There are ten RC instances, designated RC1 through RC10 range from a minimum of 10 obstacles and 500 pins to a maximum of 1000 obstacles and 500 pins. The training is carried out for 200 epochs or episode for Lin et al. [16] and NRST. The wire length value, training time and cumulative reward are recorded at each epoch.

### 6.1 WIRE LENGTH COMPARISON

The proposed algorithm is also designed to manage obstacle-based instances. Its performance in terms of wire length is compared with results from previous studies on obstacle-based instances, specifically those by [9], [22], [23], [16] as shown in the Table.1.

Table.1. Wire Length Comparison

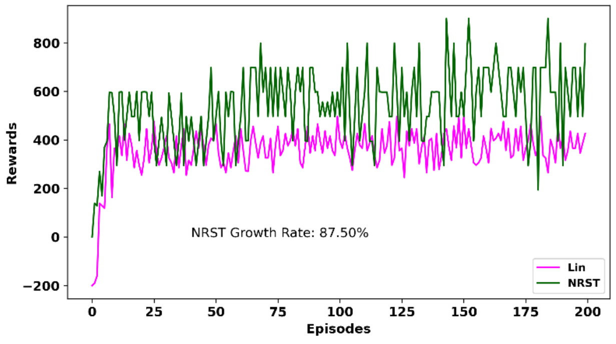
| **Bench-mark Instances** | **Wire Length** | | | | | **Improvement Rate** | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **[9]** | **[22]** | **[23]** | **[16]** | **NRST** | **[9]** | **[22]** | **[23]** | **[16]** |
| **RC01** | 26334 | 27630 | 27015 | 26040 | **26028** | 1.17 | 5.8 | 3.66 | 0.05 |
| **RC02** | 42462 | 43290 | 43882 | 41980 | **40436** | 4.78 | 6.6 | 7.86 | 3.68 |
| **RC03** | 54722 | 56940 | 54737 | 54560 | **53987** | 1.35 | 5.19 | 1.38 | 1.06 |
| **RC04** | 60925 | 61990 | 60800 | 59560 | **54020** | 11.34 | 12.86 | 11.16 | 9.31 |
| **RC05** | 75146 | 75685 | 75685 | 76640 | **74193** | 1.27 | 1.98 | 1.98 | 3.2 |
| **RC06** | 84030 | 84662 | 85808 | 82954 | **81967** | 2.46 | 3.19 | 4.48 | 1.19 |
| **RC07** | 113056 | 113598 | 113672 | 111961 | **110674** | 2.11 | 2.58 | 2.64 | 1.15 |
| **RC08** | 118277 | 119177 | 122057 | 119213 | **112825** | 4.61 | 5.33 | 7.57 | 5.36 |
| **RC09** | 117722 | 117074 | 117993 | 116295 | **112147** | 4.74 | 4.21 | 4.96 | 3.57 |
| **RC10** | 167781 | 167219 | 169443 | 169450 | **149305** | 11.02 | 10.72 | 11.89 | 11.89 |
| **Average** | | | | | | 4.48 | 5.84 | 5.75 | 4.05 |
| **Total average** | | | | | | | | | 5.03 |

The improvement rate I is calculated using the formula in the Eq.(5). The proposed method demonstrates an approximate 5% improvement over all the other methods. Thus, the proposed method NRST demonstrates a significant performance advantage across the instances compared to the existing alternatives.

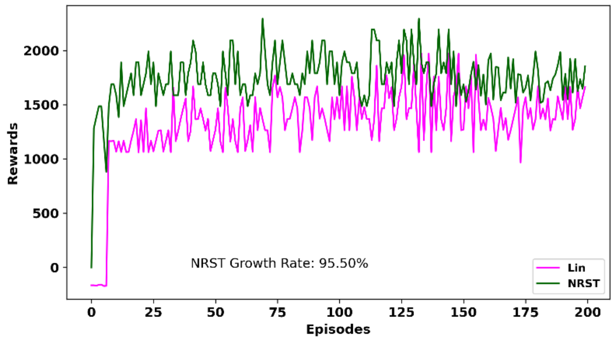
$$I=\left(\frac{\text{Wirelength}_{\text{Others}}-\text{Wirelength}_{\text{NRST}}}{\text{Wirelength}_{\text{Others}}}\right)\times 100 \tag{5}$$

### 6.2 COMPARISON OF CONVERGENCE TREND

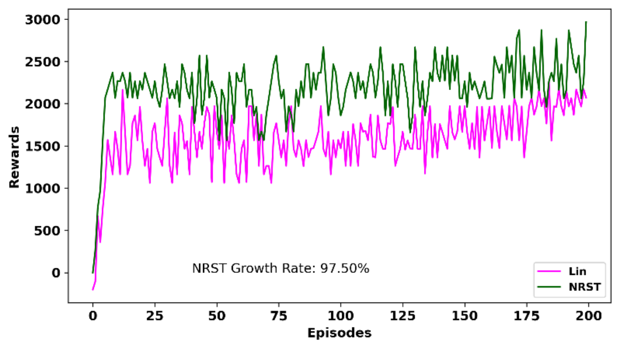
The convergence rate of DRL models defines the rate of change in the learning of the training parameter i.e. Q-values. The Fig.3 shows the reward curves for Lin et al. [16] and NRST approach. The reward value is the cumulative of the reward of each action in an episode. The Lin paper approach and NRST paper training model run same data set for 200 episode with same reward as 100 and punishment value to -1 for comparing the convergence rate. It also keeps the α, ε and γ value fixed as 0.001, 0.1 and 0.9 respectively for a meaningful comparison. The growth rate of NRST training model is compared with the Lin's [16] edge embedding model. The growth rate defines the sum of the number of episodes having reward more than or equal with the others [16]. These plots have the number of episodes on the x-axis and the corresponding episode reward on the y-axis. The dark green and magenta lines represent the NRST and Lin et al. [16] reward trends, respectively.



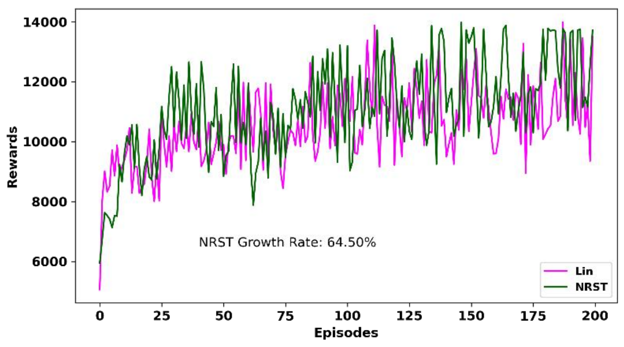
(a) RC1



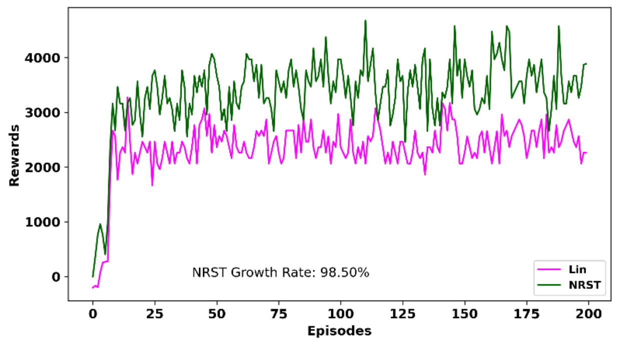
(b) RC2



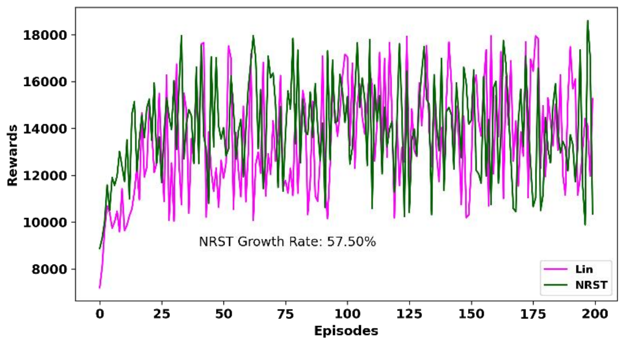
(c) RC3



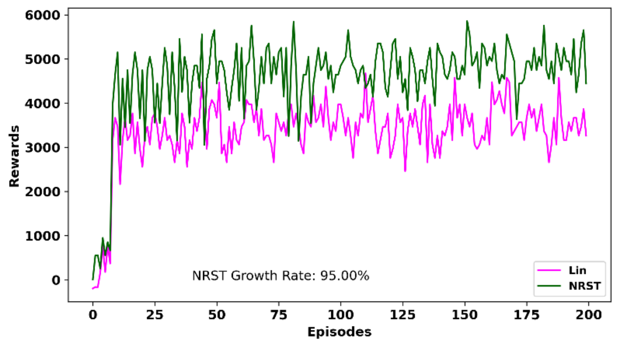
(g) RC7



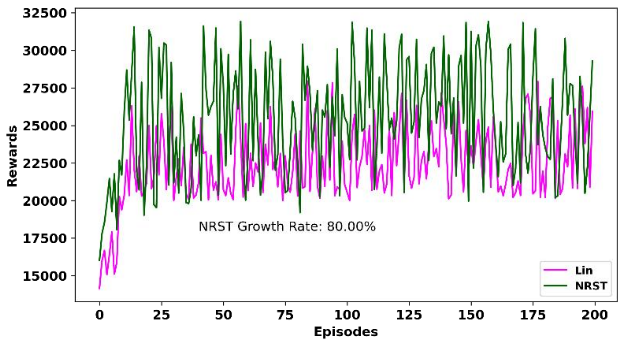
(d) RC4



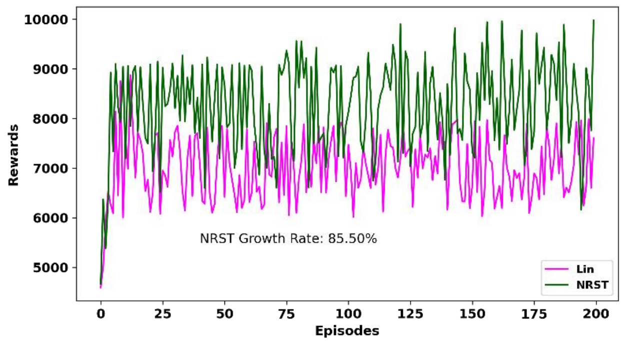
(h) RC8



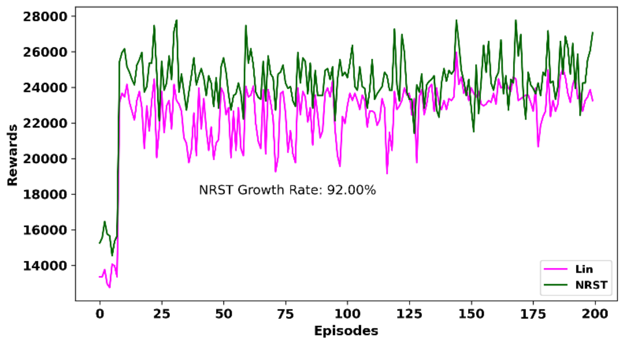
(e) RC5



(i) RC9



(f) RC6



(j) RC10

Fig.3. Training Convergence Comparison

For both the approaches, it is found that the reward value increases with increase in pins. The initial and final reward values display a hike if the number of obstacles is more than the number of pins as in the benchmark RC6, RC7, RC8 and RC9. The reward of the approaches is overlapping and competing more with each other lower gap. There is wider gap between the rewards in RC1, RC2, RC3, RC4 and RC10. In all the cases the NRST signal stays high achieving an average reward growth rate of 85.3% than the Lin et al. [16].

## 6.3 TRAINING TIME COMPARISON

The proposed model is trained using the DIMACS Challenge Benchmark Instances, RC1 through RC10, which differ in complexity and size. A comparison of training time is shown in the Fig.4 where the y-axis is the training time in second unit and x-axis are the bench mark instances. The Lin et al. [16] and NRST approaches are shown in blue and red color bar respectively. It is found the NRST has advantages of concurrency and trains faster than Lin. It can be noticed that RC10 takes less time because it has less number of Obstacles than RC6, RC7, RC8 and RC9. It can be considered that the use of parallelism makes a little advantage than the existing approach.

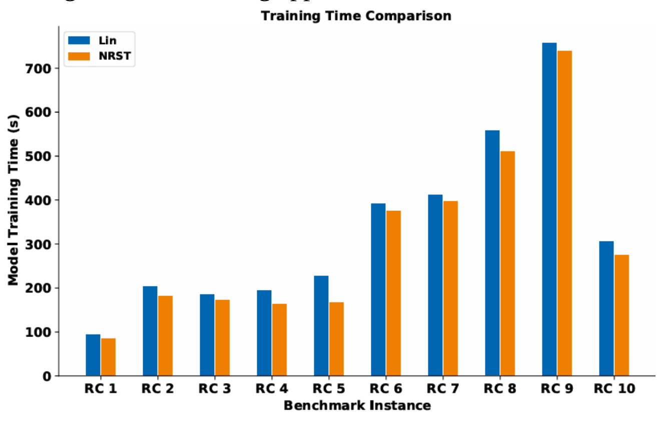


Fig.4. Training Time Comparison

## 7. CONCLUSION

The Steiner tree construction plays a critical role in VLSI design due to its significance in optimizing interconnects on a chip. Hence it always attracts sincere attention. A union-find data structure based deep Q-learning approach has been developed in this paper for construction of a rectilinear steiner tree. It considers the constraint to avoid obstacles while minimizing the wire length. It achieves lower wire length for obstacle based instances in DIMACS Implementation Challenge. The results obtained are quantified as below:

There is improvement of approximately 5% in wire length reduction. The convergence growth rate is 85.3% that indicates an accumulation of reward resulting in maximum return. The parallelism in the model leads to 9.8% reduction in training time across different complex benchmarks in diverse scenarios.

The current approach requires higher configuration-based architecture like memory and GPU. Therefore, attention can be directed toward developing alternative optimized solutions for Steiner tree construction by designing other evolutionary or nature-inspired algorithms to address the mentioned limitation.

## REFERENCES

[1] X. Chen, G. Liu, N. Xiong, Y. Su and G. Chen, “A Survey of Swarm Intelligence Techniques in VLSI Routing Problems, *IEEE Access* Vol. 8, pp. 26266-26292, 2020.

[2] C. Mohapatra, N. Adhikari, B. Ray, B. Nayak and A. Dash, “AMST: Accelerated Minimum Spanning Tree for Scalable Data”, *Indian Journal of Science and Technology*, Vol. 16, No. 37, pp. 3110-3120, 2023.

[3] R.S. Sutton and A.G. Barto, “Reinforcement Learning: An Introduction”, *IEEE Transactions on Neural Networks*, Vol. 9, No. 5, pp. 1-9, 2018.

[4] Z. Ding, Y. Huang, H. Yuan and H. Dong, “Introduction to Reinforcement Learning, Deep Reinforcement Learning: Fundamentals, Research and Applications”, Springer, pp. 47-123, 2020.

[5] B. Jang, M. Kim, G. Harerimana and J.W. Kim, “Q-learning Algorithms: A Comprehensive Classification and Applications”, *IEEE Access*, Vol. 7, pp. 133653-133667, 2019.

[6] B. Altuner and Z.H. Kilimci, “A Novel Deep Reinforcement Learning based Stock Price Prediction using Knowledge Graph and Community Aware Sentiments”, *Turkish Journal of Electrical Engineering and Computer Sciences*, Vol. 30, No. 4, pp. 1506-1524, 2022.

[7] S. Kundu, S. Roy and S. Mukherjee, “Rectilinear Steiner Tree Construction Techniques using PB-SAT-based Methodology”, *Journal of Circuits, Systems and Computers*, Vol. 29, No. 4, pp. 1-7, 2020.

[8] M. Rosenberg, T. French, M. Reynolds and L. While, “A Genetic Algorithm Approach for the Euclidean Steiner Tree Problem with Soft Obstacles”, *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 618-626, 2021.

[9] W. Guo and X. Huang, “Pora: A Physarum-Inspired Obstacle-Avoiding Routing Algorithm for Integrated Circuit Design”, *Applied Mathematical Modelling*, Vol. 78, pp. 268-286, 2020.

[10] H. Van Hasselt, A. Guez and D. Silver, “Deep Reinforcement Learning with Double Q Learning”, *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 30, No. 1, pp. 2094-2100, 2016.

[11] H. Hasselt, “Double Q-Learning”, *Advances in Neural Information Processing Systems*, Vol. 23, pp. 1-8, 2010.

[12] H. Liao, W. Zhang, X. Dong, B. Poczos, K. Shimada and L. Burak Kara, “A Deep Reinforcement Learning Approach for Global Routing”, *Journal of Mechanical Design*, Vol. 142, No. 6, pp. 1-17, 2020.

[13] J. Liu, G. Chen and E.F. Young, “Rest: Constructing Rectilinear Steiner Minimum Tree Via Reinforcement Learning”, *Proceedings of International Conference on Design Automation*, pp. 1135-1140, 2021.

[14] Z. Yan, H. Du, J. Zhang and G. Li, “Cherrypick: Solving the Steiner Tree Problem in Graphs using Deep Reinforcement Learning”, *Proceedings of International Conference on Industrial Electronics and Applications*, pp. 35-40, 2021.

[15] S. Wang, Y. Wang and G. Tong, “Deep-Steiner: Learning to Solve the Euclidean Steiner Tree Problem”, *Proceedings of International Conference on Wireless Internet*, pp. 228-242, 2022.

[16] Z. Lin, Y. Zhu, X. Huang, L. Yang and G. Liu, "Obstacle-Avoiding Rectilinear Steiner Minimal Tree Algorithm based on Deep Reinforcement Learning", *Proceedings of International Conference on Artificial Intelligence of Things and Systems*, pp. 149-156, 2023.

[17] A.K. Dubey and V. Jain, "Comparative Study of Convolution Neural Network's Relu and Leaky-Relu Activation Functions", *Applications of Computing, Automation and Wireless Systems in Electrical Engineering*, pp. 873-880, 2019.

[18] T. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein, "*Introduction to Algorithms*", 2009.

[19] R. Harris, K.J. Millman, S.J. Van Der Walt, R. Gommers, P. Virtanen, D. Cour napeau, E. Wieser, J. Taylor, S. Berg and N.J. Smith, "Array Programming with Numpy", *Nature*, Vol. 585, pp. 357-362, 2020.

[20] V. Mnih, K. Kavukcuoglu, D. Silver, A.A. Rusu, J. Veness, M.G. Bellemare, A. Graves, M. Riedmiller, A.K. Fidjel and G. Ostrovski, "Human-Level Control Through Deep Reinforcement Learning", *Nature*, Vol. 518, pp. 529-533 2015.

[21] Dimacs Implementation Challenge, Available at https://dimacs11.zib.de/downloads.html, Accessed in 2023.

[22] X. Huang, W. Guo, G. Liu and G. Chen, "Fh-oaos: A Fast Four-Step Heuristic for Obstacle-Avoiding Octilinear Steiner Tree Construction", *ACM Transactions on Design Automation of Electronic Systems*, Vol. 21, No. 3, pp. 1-31, 2016.

[23] X. Huang, G. Liu, W. Guo, Y. Niu and G. Chen, "Obstacle-Avoiding Algorithm in X-Architecture based on Discrete Particle Swarm Optimization for VLSI Design", *ACM Transactions on Design Automation of Electronic Systems*, Vol. 20, No. 2, pp. 1-28, 2015.