Reinforcement Learning Based Simulated Annealing

Nathan Qiu
Stony Brook University
Stony Brook, United States of America
nathanqiu07@gmail.com

Daniel Liang
Stony Brook University
Stony Brook, United States of America
daniel.liang.2@stonybrook.edu

ABSTRACT

Simulated Annealing (SA) is a stochastic optimization algorithm widely employed to approximate the global optimum of an energy function in both discrete and continuous problem domains. As an extension of conventional gradient descent methods, SA probabilistically accepts worse solutions to escape local optima, thereby enhancing the exploration of the solution space. SA's performance is highly contingent upon specific components, notably the neighbor proposal distribution and the temperature annealing schedule. Recent advancements such as Neural SA have improved upon traditional SA by adopting a reinforcement learning perspective, interpreting the neighbor proposal distribution as a learnable policy. Neural SA outperforms vanilla SA algorithms across various combinatorial optimization benchmarks and exhibits scalability and computational efficiency for larger problems. However, its performance remains inferior to standard commercial solvers, and it is not very generalizable across continuous problems. In this work, we introduce Reinforcement Learning Based Simulated Annealing (RL Based SA), a significant enhancement over Neural SA in terms of performance and generalizability. RL Based SA modifies the state parameters to include the change in energy from SA. It also replaces the multilayer perceptron neural networks trained using proximal policy optimization (PPO) with long short-term memory (LSTM) neural networks. This substitution enables the processing of time-series inputs of variable lengths, allowing the utilization of the entire SA rollout as input. We demonstrate that RL Based SA achieves superior results over Neural SA, vanilla SA, and adaptive SA, while attaining performance comparable to standard solvers in terms of solution quality and runtime across a spectrum of discrete and continuous problems. The benchmarks evaluated include the Knapsack, Bin Packing, and Traveling Salesperson problems, as well as continuous optimization functions such as Rosenbrock, Ackley, and Eggholder functions, and we presented training and convergence time comparisons on each function to highlight the computational trade-offs of our approach. Additionally, we show that RL Based SA is generalizable across different continuous problems, robustly scalable with respect to problem size, and computationally efficient.

KEYWORDS

Reinforcement Learning, Simulated Annealing, Combinatorial Optimization

Proc. of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025), Y. Vorobeychik, S. Das, A. Nowé (eds.), May 19 – 23, 2025, Detroit, Michigan, USA. © 2025 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org).

ACM Reference Format:
Nathan Qiu and Daniel Liang. 2025. Reinforcement Learning Based Simulated Annealing. In *Proc. of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025), Detroit, Michigan, USA, May 19 – 23, 2025*, IFAAMAS, 9 pages.

1 INTRODUCTION

Combinatorial optimization (CO) problems are widespread in various real-world applications, including logistics, resource allocation, scheduling, and network design. Efficient solutions are crucial, as suboptimal ones can cause financial losses, higher costs, and adverse environmental impacts [33]. However, the inherent computational complexity of CO problems, characterized by NP-hardness, makes exact algorithms impractical for large instances. This necessitates the use of metaheuristics, which are general problem solving frameworks that can provide near-optimal solutions within reasonable timeframes [20].

Simulated Annealing (SA) is a metaheuristic inspired by the annealing process in metallurgy. It approximates the global optimum by probabilistically accepting both improvements and occasional worse solutions, enhancing exploration and helping escape local minima [7, 30]. Despite its simplicity and broad applicability, SA's performance relies on careful tuning of components such as the neighbor proposal distribution and temperature schedule, which limits scalability and adaptability across different problems [4].

Recent advances in Simulated Annealing integrate machine learning, particularly reinforcement learning (RL), to enhance performance. For example, [2, 31] use RL to generate and refine candidate solutions. However, these methods focus on initial solutions or partial policy parameters rather than learning the full proposal distribution. Neural SA [5] models the neighbor proposal distribution as a learnable policy, improving scalability and efficiency on combinatorial benchmarks. However, it still underperforms commercial solvers and lacks generalizability for continuous problems.

In this work, we introduce Reinforcement Learning Based Simulated Annealing (RL Based SA), enhancing both performance and generalizability. By framing SA within an RL approach, we optimize neighbor generation, automate parameter tuning, and enable dynamic adaptation for diverse problems. RL Based SA preserves SA's guaranteed convergence while improving solution quality across various benchmarks. We evaluate RL Based SA on three discrete problems: the Knapsack, Bin Packing, and Traveling Salesperson problems, and three continuous functions: the Rosenbrock, Ackley, and Eggholder functions.

2 BACKGROUND AND RELATED WORK

2.1 Simulated Annealing Algorithm

Simulated Annealing (SA) is a probabilistic optimization technique designed to find an approximate global optimum of a function in

a large search space, especially when the search space contains numerous local optima. Inspired by the physical process of annealing in metallurgy, SA emulates the cooling of a material to reach a state of minimum energy. The algorithm starts with an initial solution and a high "temperature," which allows it to accept not only improvements but also occasional worsening in the objective function. This characteristic enables the algorithm to escape local minima [7, 30].

First, the algorithm starts with an initial solution x_0 and an initial temperature T_0. The initial solution can be chosen randomly or based on some heuristic. For each iteration k, a new candidate solution x' is generated from the current solution x_k using a neighbor proposal distribution $N(x_k)$. This distribution defines how the algorithm explores the neighboring solutions. Common strategies include small random perturbations or moves along specific dimensions. Then, the change in the objective function, also known as the energy function, is calculated $\Delta E = E(x') - E(x_k)$ [19].

To decide whether to accept the new solution x', the Metropolis-Hastings criterion is applied. Specifically, the acceptance probability is defined as [3]:

$$P_{\text{accept}}(x') = \begin{cases} 1, & \text{if } \Delta E \leq 0, \\ \exp\left(-\dfrac{\Delta E}{T_k}\right), & \text{if } \Delta E > 0. \end{cases}$$

This criterion ensures that any solution that improves the objective function ($\Delta E \leq 0$) is accepted, while solutions that worsen it ($\Delta E > 0$) are accepted with a probability that decreases exponentially with ΔE and increases with temperature T_k. This mechanism allows the algorithm to escape local minima by accepting worse solutions with a certain probability.

Finally, if the new solution x' is accepted, set $x_{k+1} = x'$; otherwise, retain the current solution ($x_{k+1} = x_k$). As the SA algorithm progresses, the temperature T_k is updated according to a cooling schedule. A commonly used schedule is the exponential cooling schedule [17], $T_{k+1} = \alpha T_k$, where $\alpha \in (0, 1)$ is the cooling rate. The algorithm continues iterating until a stopping criterion is met, such as reaching a minimum temperature T_{min}, a maximum number of iterations, or a convergence threshold for the objective function.

2.2 Markov Decision Process

In the context of a Markov Decision Process (MDP) [29], defined by $\mathcal{M} = (\mathcal{S}, \mathcal{A}, R, P, \gamma)$, the primary objective is to maximize the expected return, denoted as $\mathbb{E}_{\tau \sim P(\tau|\pi)}[R(\tau)]$. This expected return is calculated by summing the discounted rewards received over a trajectory, $R(\tau) = \sum_{k=0}^{K-1} \gamma^k r_k$, where $r_k = R(s_k, a_k, s_{k+1})$ represents the immediate reward for transitioning from state s_k to state s_{k+1} via action a_k. Each trajectory $\tau = (s_0, a_0, s_1, a_1, \ldots, s_K)$ is generated according to a stochastic policy π, which dictates the probability distribution over actions given the current state, and a transition kernel P, which governs the probability distribution over the next states. The trajectory's initial state s_0 is sampled from the start-state distribution ρ_0. The goal within this framework is to find a policy π that results in the highest expected return, effectively solving the MDP by optimizing the policy to maximize the cumulative rewards across the trajectories determined by $\mathcal{S}, \mathcal{A}, R, P$, and γ.

2.3 Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning where an agent learns to make decisions by interacting with an environment to maximize cumulative rewards. Unlike supervised learning, which relies on labeled data, RL involves learning through trial and error from the consequences of actions without explicit instruction [16].

In RL, the agent interacts with the environment over discrete time steps. At each time t, the agent observes the current state s_t, selects an action a_t based on a policy $\pi(a_t \mid s_t)$, and receives a reward r_t. The environment then transitions to a new state s_{t+1} according to the transition probability $P(s_{t+1} \mid s_t, a_t)$ [16]. Reinforcement learning can be formally modeled as a Markov Decision Process (MDP), where the dynamics of the environment are governed by the transition probability $P(s_{t+1} \mid s_t, a_t)$, which is unknown. The reward r_t is typically available at time $t + 1$, providing feedback for updating the policy.

The agent's objective is to find an optimal policy π^* that maximizes the expected cumulative discounted reward, defined as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1},$$

where $\gamma \in [0, 1)$ is the discount factor that balances the importance of immediate and future rewards [27].

The value function $V^\pi(s)$ represents the expected return when starting from state s and following policy π, and is defined as $V^\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s]$. Additionally, the optimal value function $V^*(s)$ is the maximum value achievable from state s under any policy, defined as $V^*(s) = \max_\pi V^\pi(s)$ [27].

2.4 Application of RL on SA and Advantages over other SA methods

We formulate simulated annealing (SA) within a reinforcement learning (RL) framework by interpreting neighbor generation and temperature scheduling as policies, allowing RL methods to learn optimal proposal distributions and temperature schedules. Unlike previous works [2, 31] that use RL to generate initial solutions for SA, we integrate RL directly into SA by making key components—specifically the proposal distribution—learnable policies.

Adaptive Simulated Annealing (ASA) improves the proposal distribution by dynamically adjusting parameters during optimization based on criteria like neighbor proposal distribution or cooling scheduling [14, 17]. However, ASA relies on heuristic rules that may not generalize well across different problems. Applying RL to SA enables learning adaptive proposal distributions and temperature schedules through experience, automatically adjusting behavior based on specific problem characteristics.

This approach allows the proposal distribution to be adapted based on the current state and past experience, enabling more efficient exploration of the solution space compared to fixed or heuristically adjusted proposal distributions in vanilla SA and ASA. Moreover, the temperature schedule can also be learned as part of the policy, providing further adaptability.

3 PROBLEM SETTING AND METHOD

In this section, we outline the problem setting, and define the MDP formulation for SA. We then demonstrate how a proposal distribution can be learned, and describe the architectures of the various models we employ. Finally, we justify how in RL Based SA, convergence is still guaranteed just like in vanilla SA.

3.1 Problem Setting

The objective is to find the optimal solution $\mathbf{X}^*$ that minimizes (or maximizes) the energy function $E(\mathbf{X})$ over the domain $\mathbb{D}$:

$$\mathbf{X}^* = \arg\min_{\mathbf{X}\in\mathbb{D}} E(\mathbf{X})$$

This optimization employs the Simulated Annealing (SA) algorithm, a stochastic technique that mimics the cooling process in metallurgy to escape local minima and approach a global optimum.

Algorithm 1 Reinforcement Learning Based Simulated Annealing

Input: Initial state $\mathbf{s}_0 = (\mathbf{x}_0, \psi, \Delta E, T_0)$, proposal distribution π, temperature schedule $T_1, T_2, T_3, \ldots$, energy function $E(\bullet; \psi)$
Output: Approximate solution $\mathbf{x}^*$
for $i = 1 : K$ **do**
  Generate action $\mathbf{a}$ using policy $\pi_\theta(\mathbf{s}_i)$
  Propose $\mathbf{x}' \leftarrow \mathbf{x}$ from action $\mathbf{a}$
  Calculate $\Delta E \leftarrow E(\mathbf{x}'; \psi) - E(\mathbf{x}; \psi)$
  Calculate acceptance probability $p \leftarrow \exp\left(-\frac{\Delta E}{T}\right)$
  Generate a random number $u \sim \text{Uniform}(0, 1)$
  if $u < p$ **then**
    $\mathbf{s}_{k+1} \leftarrow (\mathbf{x}', \psi, \Delta E, T_{k+1})$
  else
    $\mathbf{s}_{k+1} \leftarrow (\mathbf{x}_k, \psi, \Delta E, T_{k+1})$
  end if
  Update T using temperature schedule $T_1, T_2, T_3, \ldots$
end for
return X

The Simulated Annealing (SA) algorithm systematically explores the solution space to find an approximate global minimum of the energy function $E(\mathbf{X})$. Starting from an initial solution $\mathbf{X}_0$ and temperature T_0, the algorithm iteratively generates new candidate solutions by applying a perturbation defined by the policy $\pi_{\Delta\mathbf{X}}$. Each new solution undergoes an acceptance test based on the Boltzmann distribution, which favors solutions with lower energy but allows occasional acceptance of higher energy solutions to escape local minima. The temperature is methodically reduced according to a predefined schedule π_T, decreasing the likelihood of accepting worse solutions as the process continues. This gradual cooling mirrors the physical process of annealing, aiming to balance exploration and exploitation until the termination condition is met, typically after a fixed number of iterations. The final solution $\mathbf{X}$ approximates the global minimum, effectively utilizing the dual strategy of adaptive search and controlled temperature descent.

3.2 MDP formulation of SA

Simulated Annealing (SA) can be effectively described within the Markov Decision Process (MDP) [29] framework where SA acts as an agent interacting with an environment defined by the optimization problem, providing a structured method to analyze its optimization capabilities. In this formalization, the components of the MDP are tailored to encapsulate the mechanics of SA, enhancing the clarity of its stochastic and adaptive processes.

The state space $\mathcal{S}$ in our MDP is defined by each state $\mathbf{s} = (\mathbf{x}, \psi, \Delta E, T)$, where $\mathbf{x}$ is the current solution within the problem domain, ψ is a parametric description of the problem instance, ΔE is change in energy from the previous state, and T represents the instantaneous temperature, crucial for the adaptive acceptance of new solutions.

Actions $\mathbf{a} \in \mathcal{A}$ correspond to transitions in the solution space, mapping $(\mathbf{x}, \psi, \Delta E, T) \mapsto (\mathbf{x}', \psi, \Delta E, T)$, where $\mathbf{x}' \in \mathcal{N}(\mathbf{x})$ signifies a candidate solution within a defined neighborhood of $\mathbf{x}$. This neighborhood is intentionally restricted to limit the energy variation between states, a heuristic that discards extreme moves and generally results in faster convergence.

The transition dynamics are governed by the Metropolis-Hastings (MH) algorithm [23], serving as a stochastic transition kernel dependent on the current temperature:

$$\mathbf{x}_{k+1} = \begin{cases} \mathbf{x}', & \text{with probability } p_{\text{accept}} \\ \mathbf{x}_k, & \text{with probability } 1 - p_{\text{accept}} \end{cases}$$

The reward structure can be either the negative change in energy, which is the immediate energy gain:

$$\text{Immediate gain} = -\Delta E_k = E(\mathbf{x}_k) - E(\mathbf{x}_{k+1}),$$

or a terminal reward capturing the lowest energy achieved during a complete trajectory:

$$\text{Terminal reward} = -\min_{\mathbf{x}\in\mathbf{x}_{1:k}} E(\mathbf{x}).$$

This MDP formulation provides a robust framework for analyzing and enhancing the SA algorithm, utilizing both contemporary reinforcement learning strategies and traditional optimization techniques.

3.3 Neural SA Agent Architecture

In Neural SA [5], the SA state is defined as (x, ψ, T), where x is the current solution, ψ is a parametric description of the problem instance, and T is the instantaneous temperature. They trained the SA chain using Proximal Policy Optimization (PPO) [26] and Evolution Strategies (ES) [24]. The immediate gain reward works best with PPO, providing feedback at each iteration on whether the previous action improved the solution. ES works best with the primal reward, which is non-local and provides the minimum value over the entire trajectory τ at the end of the rollout.

In their policy network architecture, the state is mapped into a set of features for each problem. Each feature is fed into a Multilayer Perceptron (MLP), which embeds it into a logit space; a softmax activation function is then applied to generate probabilities. Since the computations and MLPs are embarrassingly parallel, the complexity increases linearly with N, crucial for high-dimensional problems. This architecture is permutation equivariant [34], meaning the network output changes consistently when the input order is rearranged—an important requirement for the considered combinatorial optimization problems [25].

3.4 Neural SA Modified State Formulation

We reformulate the Neural SA state to include the change in energy between each step, effectively incorporating the immediate reward into the state. The SA state becomes $(x, \psi, \Delta E, T)$. Including ΔE enables the agent to learn which actions decrease energy and improve solutions. Providing the MLP [22] with explicit information about the immediate effect of transitions helps the model understand the local energy landscape and the impact of solution modifications. The immediate energy change captures the gradient of energy transitions, offering the network direct insight into the quality of each step and enhancing its ability to balance exploration and exploitation. This richer representation leads to more informed decision-making, allowing for more efficient exploration of the solution space and potentially higher-quality solutions as the network better learns how changes propagate across the energy landscape.

3.5 LSTM Agent Architecture

We replace the MLPs in both the actor and critic of PPO with Long Short-Term Memory Networks (LSTMs) [13]. Instead of using only the current state, we utilize the entire sequence of states for an SA rollout. Each element in the time series is expressed as $S_i = (x, \psi, E, \Delta E, T)$ for discrete optimization problems and $S_i = (x, E, \Delta E, T)$ for continuous optimization problems. Discrete problems include problem parameters ψ in the state to facilitate learning, while continuous problems omit them to demonstrate the model's generalizability and ease of use.

We demonstrate that an RL Based SA LSTM agent trained on one continuous problem performs well when evaluated on different continuous problems. Incorporating LSTMs into PPO for time series data offers significant advantages over MLPs. Since SA involves sequential decision-making where each step depends on past solutions and temperatures, LSTMs more effectively capture temporal trends across the SA chain. By retaining information from prior time steps through a hidden state, LSTMs help the PPO agent to predict the impact of future actions, improving neighbor sampling decisions. This enhances solution space exploration and optimization performance, especially in complex energy landscapes with strong temporal correlations.

In order to reduce computational costs, we feed time series states individually into the LSTM during SA rollout. However, this method is not applicable during the backpropagation and optimization phase of the model training process since training the model for the entire time series data requires inputting the entire SA rollout up to a given time step into the LSTM architecture for each parameter update. Thus, while LSTMs can offer significantly better modeling of temporal dependencies, they are more computationally expensive than MLPs. Future work could explore methods like truncated backpropagation through time (TBPTT) [28] to limit the inputted SA rollout and reduce computational cost.

3.6 Convergence Guarantee

Note that despite these modifications, RL Based SA preserves Neural SA's theoretical convergence, as the Metropolis-Hastings acceptance step remains intact. Therefore, the learned policy proposes neighbor generation moves but does not override Neural SA's temperature annealing schedule, ensuring global convergence.

4 EXPERIMENT

We evaluate our RL Based SA architecture on two different types of problems: discrete and continuous optimization problems. By using the same model architecture and hyperparameters for all problems, we show broad applicability and user convenience. For discrete problems, we test RL Based SA on various problem sizes, but only train on the smallest problem size for each corresponding problem setting. This allows us to demonstrate effective scalability of a light model with lower training time. Similarly, we consider different SA rollout lengths but train only on short rollouts, reducing training time and demonstrating generalization. Using these methods, we demonstrate great performance and generalizability with a lightweight, equivariant architecture.

In all experiments, we start from random solutions and use an exponential cooling schedule: $T_k = \alpha^k T_0$, where α is determined by fixing T_0 and T_K and computing it from the total steps K. This method permits us to vary the rollout length while maintaining the same range of temperatures for every run. We use the Adam optimizer [8] for both the actor and critic networks in PPO.

We first incorporate the modified state formulation, which we refer to as "Added ΔE to State (Ours)" in tables, and then add in LSTMs in combination as well, which we refer to as "LSTMs (Ours)." We generate all datasets of problems using consistent random seeds to allow for reproducibility.

4.1 Discrete Problems

For discrete optimization problems, we train and evaluate on the Knapsack, Bin Packing, and Traveling Salesperson problems. We demonstrate superior performance for both our architectures, modified state formulation and LSTMs, compared to vanilla SA, adaptive SA, and superior or comparable performance to the original Neural SA. For the different methods trained within each discrete problem, we keep the hyperparameters such as number of problems, number of epochs, reward type, batch size, and learning rate constant to demonstrate generalizability. All models trained and evaluated on discrete problems were implemented using Pytorch 2.5 and trained and evaluated with 2 V100 GPUs.

4.1.1 *Knapsack Problem.*

The Knapsack problem is a fundamental combinatorial optimization challenge in resource allocation. Given N items with values $v_i > 0$ and weights $w_i > 0$, the objective is to select a subset of these items that maximizes the total value without exceeding a weight capacity W. Each item is either fully included or excluded. The problem is weakly NP-complete with an exponential search space of size 2^N.

The formal mathematical formulation is as follows:

$$\begin{aligned} \text{Maximize} \quad & E(x;\psi) = \sum_{i=1}^{N} v_i x_i, \\ \text{Subject to} \quad & \sum_{i=1}^{N} w_i x_i \le W, \\ & x_i \in \{0,1\}, \quad \forall i = 1, \ldots, N, \end{aligned}$$

Table 1: Average cost of Knapsack Problem solutions across five random seeds. In parentheses is the optimality gap to the best solution. Bigger is better. *Values reported in [1]

Problem Dimension	*Bello RL*	*Bello AS*	*Vanilla SA*	*ASA*	*Neural SA PPO*	*Added ΔE to State (Ours)*	*LSTMs (Ours)*	*OR-Tools*
Knap50	19.86*	20.07*	18.37 (8.61%)	18.52 (7.86%)	19.58 (2.59%)	19.47 (3.13%)	19.99 (0.55%)	**20.10** (0.00%)
Knap100	40.27*	40.50*	36.69 (9.21%)	36.79 (8.96%)	39.20 (3.00%)	39.02 (3.44%)	40.10 (0.77%)	**40.41** (0.00%)
Knap200	57.10*	57.45*	50.85 (11.66%)	50.00 (13.13%)	51.31 (10.86%)	53.15 (7.66%)	54.96 (4.52%)	**57.56** (0.00%)
Knap500	-	-	126.97 (11.28%)	123.30 (14.28%)	130.08 (9.57%)	135.22 (5.99%)	126.94 (11.75%)	**143.84** (0.00%)
Knap1000	-	-	254.59 (11.77%)	245.85 (14.80%)	263.99 (8.51%)	273.34 (5.27%)	253.84 (12.03%)	**288.56** (0.00%)
Knap2000	-	-	508.71 (11.89%)	489.47 (15.22%)	533.04 (7.68%)	549.50 (4.83%)	508.97 (11.85%)	**577.37** (0.00%)

where the solution $x = (x_1, x_2, \ldots, x_N)$ is a binary vector indicating the inclusion ($x_i = 1$) or exclusion ($x_i = 0$) of item i, and ψ encapsulates the problem parameters v_i and w_i.

In Simulated Annealing (SA), potential solutions are represented by x. The solution space is explored by defining a neighborhood around x that includes all feasible solutions reachable by flipping a single bit—changing the inclusion status of one item—while ensuring the total weight constraint is not violated.

A new solution x' is proposed by flipping the bit for item i, ensuring that adding or removing the item keeps the total weight within capacity W. If flipping x_i from 0 to 1 (adding the item) exceeds W, the move is disallowed.

To efficiently navigate the solution space, we implement a policy $\pi_\theta(i \mid s)$ parameterized by θ, where s represents the current state. This policy is modeled using a neural network that predicts the probability of selecting each item for a flip based on the item's features and the current system state.

The mapping from state to action probabilities is defined as:

$$\pi_\theta(i \mid s) = \text{softmax}(z)_i, \quad \text{where} \quad z_i = f_\theta\left([x_i, w_i, v_i, W, T]\right),$$

and the proposed new solution is: $x' = x + \text{onehot}(i) \pmod 2$ with f_θ being the neural network function with parameters θ which we define using our model architecture.

In the original Neural SA, the neural network f_θ is a simple two-layer network $5 \to 16 \to 1$. It consists of an input layer of size 5, corresponding to the five features, a hidden layer with 16 neurons using ReLU activations, and an output layer producing a single logit z_i. This architecture is lightweight, with 112 learnable parameters.

In our modified state formulation, f_θ becomes a two-layer network $6 \to 16 \to 1$. The input layer now has size 6, adding a parameter to denote the change in energy. The hidden and output layers remain the same, resulting in a lightweight model with 129 learnable parameters.

Finally, when incorporating LSTMs instead of MLPs in PPO, f_θ is a three-layer network $7 \to 16 \to 16 \to 1$. The input layer is of size 7, adding two parameters for the current energy and immediate gain. There are two hidden layers with 16 neurons each using ReLU activations, and an output layer producing a single logit. This architecture is more complex but still relatively lightweight, with 417 learnable parameters.

We implement the setup as outlined in [1], focusing on the self generated datasets Knap50, Knap100, and Knap200. For each dataset, Knap$N$ contains N items with weights and values uniformly randomly generated from $(0, 1]$. Additionally, the knapsack capacities C_N are defined such that $C_{50} = 12.5$, $C_{100} = C_{200} = 25$, and $C_N = \frac{N}{8}$ for $N > 200$. We use OR-Tools [21] to compare against as a standard CO solver. We train for 1000 epochs on 128 problems on the Knap50 dataset.

The results in Table 1 indicate that RL Based SA improves significantly over vanilla SA and adaptive SA, up to a 9% optimality gap. The addition of ΔE to the state results in comparable performance to Neural SA for Knap50, Knap100, and Knap200, and consistently superior performance for Knap500, Knap1000, and Knap2000. However, it slightly falls behind the two methods used in [1], referred to as "Bello RL" and "Bello AS" in Table 1, which use a much larger neural network with orders of magnitude more parameters and $10,000$ training steps. It also falls slightly behind OR-Tools, with an average optimality gap of around 5%.

Our LSTM implementation achieves highly competitive results for Knap50 and Knap100, matching the performance of [1] and OR-Tools, despite a lightweight architecture with far fewer parameters. It significantly outperforms SA, ASA, and Neural SA for these problem sizes. On Knap200, LSTMs fall slightly behind [1] and OR-Tools. LSTMs perform worse on larger problem sizes, exhibiting results similar to vanilla SA. Thus, this suggests that our LSTM method excels on problem sizes similar to the model's training data, but lack scalability for larger Knapsack problems. Therefore, our LSTM method might require more specialized training or more layers to scale more efficiently.

Overall, incorporating ΔE into the state consistently outperforms vanilla SA, ASA, and Neural SA, only slightly trailing optimizers with significantly larger architectures and many more parameters. LSTMs exhibit extremely strong performance on problem sizes similar to their training set, being comparable to larger state-of-the-art optimizers. In general, RL Based SA demonstrates strong performance with a lightweight generalizable architecture not specifically tailored for this problem.

4.1.2 *Bin Packing Problem.*

The Bin Packing problem, related to the Knapsack problem, is a fundamental combinatorial optimization problem that aims to pack N items into the fewest bins of fixed capacity W without exceeding the weight W in any bin. Every item is either fully included or excluded. Each item $i \in \{1, 2, \ldots, N\}$ has weight $w_i > 0$ and must fit in some bin, assuming $W \geq \max_i w_i$.

Table 2: Average cost of Bin Packing Problem solutions across five random seeds. In parentheses is the optimality gap to the best solution. Lower is better. *Indicates that only the trivial solution was found in the set time

Problem Dimension	*Vanilla SA*	*ASA*	*Neural SA PPO*	*Added ΔE to State (Ours)*	*OR-Tools (SCIP)*	*FFD*
Bin50	30.29 (13.40%)	29.87 (11.83%)	27.25 (2.02%)	27.20 (1.83%)	**26.71** (0.00%)	27.10 (1.46%)
Bin100	60.49 (14.33%)	59.36 (12.19%)	53.35 (0.83%)	53.24 (0.62%)	53.91 (1.89%)	**52.91** (0.00%)
Bin200	121.18 (16.24%)	118.77 (13.93%)	105.77 (1.46%)	105.57 (1.27%)	109.19 (4.74%)	**104.25** (0.00%)
Bin500	302.73 (17.78%)	296.35 (15.30%)	261.19 (1.62%)	260.76 (1.46%)	267.63 (4.13%)	**257.02** (0.00%)
Bin1000	604.79 (18.71%)	592.43 (16.29%)	519.60 (1.99%)	518.83 (1.84%)	1000*	**509.46** (0.00%)
Bin2000	1209.37 (17.57%)	1184.41 (15.14%)	1035.56 (0.67%)	1034.04 (0.52%)	2000*	**1028.67** (0.00%)

The problem can be formulated as an integer linear program using binary variables x_{ij} and y_j. Here, $x_{ij} = 1$ if item i is placed in bin j and $x_{ij} = 0$ otherwise. Similarly, $y_j = 1$ if bin j is used and $y_j = 0$ otherwise. The mathematical formulation of the Bin Packing problem is as follows:

$$\text{Minimize } E(x;\psi) = \sum_{j=1}^{M} y_j,$$

$$\text{Subject to } \sum_{i=1}^{N} w_i x_{ij} \le W, \ \forall j = 1,\dots,M, \ \text{(bin capacity constraint)}$$

$$\sum_{j=1}^{N} x_{ij} = 1, \quad \forall i = 1,\dots,N, \ \text{(1 bin per item)}$$

$$y_j \ge x_{ij}, \quad \forall i = 1,\dots,N, \quad \forall j = 1,\dots,N,$$

$$x_{ij} \in \{0,1\}, \ y_j \in \{0,1\}, \ \text{(bin occupancy indicators)},$$

The Bin Packing problem is NP-hard with a combinatorial search space of size given by the N-th Bell number, representing ways to partition N items into bins.

In applying Simulated Annealing (SA) to Bin Packing, a policy guides item-to-bin assignments. Each action selects an item i for reassignment and a bin j for placement to improve packing efficiency.

This policy can be represented as a joint probability distribution:

$$\pi_{\theta,\phi}(a = (i,j) \mid s) = \pi_\theta(i \mid s) \cdot \pi_\phi(j \mid s, i)$$

In this equation, the policy probabilities are computed using neural networks as follows:

$$\pi_\theta(i \mid s) = \text{softmax}(z_i^{\text{item}}), \ \text{where } z_i^{\text{item}} = f_\theta\left(x_i, w_i, c_{b(i)}, T\right),$$

$$\pi_\phi(j \mid s, i) = \text{softmax}(z_j^{\text{bin}}), \ \text{where } z_j^{\text{bin}} = f_\phi\left(x_i, w_i, c_j, T\right),$$

where: $b(i)$ is the index of the bin that item i is currently assigned to, and $c_j = W - \sum_{i=1}^{N} w_i x_{ij}$ is the remaining capacity of bin j. f_θ and f_ϕ are both neural networks that output logits that are fed into softmax activation functions.

In the original Neural SA, the neural networks f_θ and f_ϕ are two-layer neural networks $4 \to 16 \to 1$. The input layer of size 4 corresponds to the features w_i, $b(i)$ or c_j, and T. This is followed by a hidden layer with 16 neurons using ReLU activations and an output layer producing a single logit z_i, resulting in 97 learnable parameters.

In our modified state formulation, f_θ and f_ϕ are two-layer neural networks $5 \to 16 \to 1$. The input layer now has size 5, adding a parameter for the change in energy. The hidden and output layers remain the same, yielding 113 learnable parameters.

For our incorporation of LSTMs, f_θ and f_ϕ are three-layer neural networks $6 \to 16 \to 16 \to 1$. The input layer has size 6, with two extra parameters representing current energy and immediate gain. Both networks have two hidden layers with 16 neurons each (ReLU activations), and the output layer produces a single logit. This architecture has 401 learnable parameters.

We did not train or evaluate our LSTM implementation for the Bin Packing problem because we could not do so for the same problem size as the other models due to computing limitations. Maintaining a consistent problem size across all models is essential for a fair comparison, and ensuring constant problem size across different problems is important for demonstrating generalizability.

The results in Table 2 demonstrate that RL Based SA with ΔE added to the state consistently performs better than Neural SA, and significantly outperforms vanilla SA and ASA. It is also able to consistently achieve solutions with energy values of around only 1% higher than those found by FFD [15], a highly effective heuristic for the Bin Packing problem. We also note that our model with ΔE very often outperformed the SCIP [10] optimizer in OR-Tools. RL Based SA is able to achieve superior or comparable results to specialized optimizers with a lightweight and generalizable architecture not specifically designed for the Bin Packing Problem.

4.1.3 *Traveling Salesperson Problem.*

The Traveling Salesperson Problem (TSP) is a combinatorial optimization challenge that seeks the shortest route visiting N cities exactly once before returning to the start. Despite its simple formulation, TSP is NP-hard, with a factorial search space ($N!$), making exact solutions impractical for large N and necessitating heuristics. It has key applications in logistics, route planning, and circuit design.

Formally, given N cities $\{c_1, c_2, \dots, c_N\}$ with coordinates $c_i \in \mathbb{R}^2$, the goal is to find a permutation $x = (x_1, x_2, \dots, x_N)$ that minimizes the total tour length. The problem is mathematically formulated as:

$$\text{Minimize} \quad E(x;\psi) = \sum_{i=1}^{N} \|c_{x_{i+1}} - c_{x_i}\|_2,$$

$$\text{Subject to} \quad x_i \ne x_j, \quad \forall i \ne j,$$

$$x_i \in \{1,2,\dots,N\}, \quad \forall i = 1,\dots,N,$$

Table 3: Average cost of TSP solutions across five random seeds. In parentheses is the optimality gap to the best solution. Lower is better. *Values reported by [11], [12], [21], [32], [6], and [9] respectively

	TSP20			TSP50			TSP100			TSP200		
	Cost	Gap	Time	Cost	Gap	Time	Cost	Gap	Time	Cost	Gap	Time
CONCORDE*	3.836	0.00%	48s	5.696	0.00%	2m	7.764	0.00%	7m	10.70	0.00%	38m
LKH-3*	3.836	0.00%	1m	5.696	0.00%	14m	7.764	0.00%	1h	10.70	0.00%	21m
Vanilla SA	3.881	1.17%	6s	5.944	4.35%	38s	8.342	7.44%	5m	11.97	11.87%	9m
Neural SA PPO	3.838	0.05%	11s	5.734	0.67%	2m	7.874	1.42%	10m	11.00	2.80%	16m
Added ΔE To State (Ours)	3.836	0.00%	17s	5.712	0.28%	2m	7.826	0.80%	12m	10.88	1.68%	28m
LSTMs (Ours)	3.839	0.08%	2m	5.742	0.81%	23m	-	-	-	-	-	-
OR-Tools*	3.86	0.85%	1m	5.85	2.87%	5m	8.06	3.86%	23m	-	-	-
GAT-T {1000}*	3.84	0.03%	12m	5.75	0.83%	16m	8.01	3.24%	25m	-	-	-
Costa {500}*	3.84	0.01%	5m	5.72	0.36%	7m	7.91	1.84%	10m	-	-	-
Fu et al.*	3.84	0.00%	1m	5.70	0.01%	8m	7.76	0.04%	15m	-	-	-

where $x_{N+1} = x_1$ ensures the tour is closed by returning to the starting city, $\|\cdot\|_2$ denotes the Euclidean distance between two points, and ψ encapsulates the problem instance, specifically the coordinates c_i of the cities.

In the context of Simulated Annealing (SA), solutions are represented by permutations x of the city indices. To effectively explore the solution space, we employ the 2-opt move, a well-established local search operator for the TSP. A 2-opt move involves selecting two positions i and j (with $i < j$) in the tour and reversing the subsequence between them, potentially reducing the tour length by eliminating crossings. The new solution x' resulting from a 2-opt move is given by:

$$x' = \left(x_1, x_2, \ldots, x_{i-1}, x_j, x_{j-1}, \ldots, x_i, x_{j+1}, \ldots, x_N\right).$$

To navigate the solution space efficiently, we implement a policy $\pi_{\theta,\phi}(a \mid s)$ parameterized by θ and ϕ, where s represents the current state, including the current tour x and the temperature T. The action $a = (i, j)$ specifies the indices for the start and end of the segment to reverse. The policy operates in two stages. First, the start index i is selected based on a probability distribution:

$$\pi_\theta(i \mid s) = \text{softmax}(z_i^{\text{start}}), \quad z_i^{\text{start}} = f_\theta\left([c_{x_{i-1}}, c_{x_i}, c_{x_{i+1}}, T]\right)$$

where f_θ represents the first neural network. Second, the end index j is selected using:

$$\pi_\phi(j \mid s, i) = \text{softmax}(z_j^{\text{end}}),$$
$$z_j^{\text{end}} = f_\phi\left([c_{x_{i-1}}, c_{x_i}, c_{x_{i+1}}, c_{x_{j-1}}, c_{x_j}, c_{x_{j+1}}, T]\right).$$

where f_ϕ represents the second neural network. By applying the softmax function to the logits, we obtain probability distributions over possible start and end indices, from which actions are sampled.

In the original Neural SA [5], simple MLP neural networks are used, with f_θ and f_ϕ being two-layer networks of dimensions $7 \to 16 \to 1$ and $13 \to 16 \to 1$, respectively. f_θ has an input layer of size 7 due to the state $(c_{x_{i-1}}, c_{x_i}, c_{x_{i+1}}, T)$. The hidden layer has 16 neurons with ReLU activations, and the output layer produces a single logit z_i, resulting in lightweight models with 145 parameters for f_θ and 241 parameters for f_ϕ.

In our modified state formulation, f_θ and f_ϕ remain two-layer neural networks but now have dimensions $8 \to 16 \to 1$ and $14 \to 16 \to 1$, respectively. The input layers gained an additional parameter representing the change in energy. This simple and effective architecture produces lightweight models with 161 parameters for f_θ and 257 parameters for f_ϕ.

With the substitution of LSTMs for MLPs, both f_θ and f_ϕ become three-layer networks with dimensions $9 \to 16 \to 16 \to 1$ and $15 \to 16 \to 16 \to 1$ respectively. The input layers have two additional parameters corresponding to the current energy and immediate gain. Both networks feature two hidden layers with 16 neurons each using ReLU activation functions, and the output layer produces a single logit. While more complex, this architecture remains relatively lightweight, with 449 parameters for f_θ and 545 parameters for f_ϕ.

We evaluate our models on the publicly available TSP 20 / 50 / 100 / 200 [18], with 10,000 problems each. The results in Table 3 indicate that RL Based SA with ΔE in the state consistently achieves superior results to Neural SA, and significantly outperforms vanilla SA. It also consistently attains better results than OR-Tools [21], and slightly outperforms other neural improvement heuristic methods in [32] and [6]. RL Based SA with ΔE in the state demonstrates comparable results to the optimal solutions achieved by CONCORDE [11] and LKH-3 [12], and also similar performance to [9]. Considering that RL Based SA has a lightweight generalizable architecture not custom designed for TSP while the competing methods do, we consider these results to be quite decent.

4.2 Continuous Problems

For continuous optimization, we train and evaluate on three 2D continuous optimization functions: the Rosenbrock, Ackley, and Eggholder functions, known for their challenging landscapes with narrow valleys and numerous local minima. We demonstrate superior performance and generalizability by training on one problem and evaluating on another, with our RL Based SA architecture consistently arriving at satisfactory solutions. To ensure fair evaluation and generalizability, hyperparameters such as problem count, epochs, reward type, batch size, and learning rate remain constant

Table 4: Average cost of solutions for various 2D continuous optimization functions across five random seeds for constant SA rollout length (640 steps). In parentheses is the optimality gap to the best solution. Lower is better.

Function	*Vanilla SA*	*ASA*	*Neural SA PPO Rosenbrock Model*	*Neural SA PPO Ackley Model*	*Neural SA PPO Eggholder Model*
Rosenbrock	0.0791 (1,482%)	0.0793 (1,486%)	0.0183 (266%)	0.0255 (410%)	0.0280 (460%)
Ackley	0.850 (1,015)	0.668 (777%)	3.450 (4,428%)	0.105 (37.80%)	0.584 (666%)
Eggholder	-519.76 (9.10%)	-519.51 (9.14%)	-296.24 (48.19%)	-455.52 (20.33%)	-441.93 (22.71%)

Function	*Added ΔE to State Rosenbrock Model*	*Added ΔE to State Ackley Model*	*Added ΔE to State Eggholder Model*	*LSTMs Rosenbrock Model*	*LSTMs Ackley Model*	*LSTMs Eggholder Model*
Rosenbrock	0.0232 (364%)	0.0232 (364%)	0.0440 (780%)	**0.00500** (0.00%)	0.0192 (284%)	0.0727 (1,354%)
Ackley	2.721 (3,471%)	0.0869 (14.04%)	1.077 (1,313%)	4.622 (5,966%)	**0.0762** (0.00%)	0.578 (659%)
Eggholder	-290.39 (49.21%)	-441.26 (22.83%)	-420.13 (26.52%)	-416.24 (27.20%)	**-571.78** (0.00%)	-521.65 (8.77%)

across all problems. All models were implemented in PyTorch 2.5 and trained on an Apple M3 Pro 12-core CPU.

The Rosenbrock function is characterized by a narrow, parabolic-shaped valley where the global minima resides. Its two-dimensional formulation is defined by:

$$f(x, y) = (a - x)^2 + b(y - x^2)^2,$$

where $a \in (0, 1]$ and $b \in (0, 100]$. The function has a global minimum at $(x, y) = (a, a^2)$, where $f_{\min}(x, y) = 0$. Despite simple gradient descent methods being more effective for this function, we will use it as an example of how Neural SA and RL Based SA perform on simple continuous functions.

The Ackley function is characterized by its large number of local minima, making it difficult for optimization algorithms to converge to the global minimum. It is defined as:

$$f(x, y) = -a \exp\left(-b\sqrt{\frac{1}{2}(x^2 + y^2)}\right) - \exp\left(\frac{1}{2}[\cos(cx) + \cos(cy)]\right) + a + \exp(1),$$

where $a \in [0, 20)$, $b \in [0, 0.2)$, and $c \in (0, 2\pi]$. The global minimum is located at $(x, y) = (0, 0)$, where $f_{\min}(x, y) = 0$. Due to the function's numerous local minima, pure gradient descent methods are much less effective. The Ackley function's numerous local minima test an algorithm's ability to escape suboptimal solutions and effectively navigate complex landscapes.

The Eggholder function is another challenging optimization problem due to its complex surface with many local minima and maxima. It is defined as:

$$f(x, y) = -(y + 47)\sin\left(\sqrt{\left|\frac{x}{2} + y + 47\right|}\right) - x\sin\left(\sqrt{|x - (y + 47)|}\right).$$

The global minimum of the Eggholder function is approximately at $(x, y) = (512, 404.2319)$, where $f_{\min}(x, y) \approx -959.6407$. The function's intricate topology makes it suitable for assessing an algorithm's capability to explore complex multi-modal spaces.

The results in Table 4 indicate extremely strong performance of our RL Based SA LSTM implementation on continuous problems. For the same rollout length, when trained and evaluated on the same problem, our LSTM implementation significantly outperforms vanilla SA, ASA, and Neural SA on all three continuous problems, achieving considerably lower energies than all other methods. Even when trained on one problem and evaluated on another, our LSTM implementation matches Neural SA's performance when trained and evaluated on that same problem. Moreover, our LSTM implementation shows better results than Neural SA in cross-problem training and evaluation, demonstrating improved generalizability across continuous problems.

5 CONCLUSION

We introduced Reinforcement Learning Based Simulated Annealing (RL Based SA), enhancing the traditional SA algorithm by incorporating the change in energy ΔE into the state representation and replacing MLPs in PPO with LSTMs. Our approach improves the agent's understanding of the energy landscape, effect of actions, and temporal dependencies across the SA chain. We conducted experiments across various combinatorial optimization benchmarks, including the Knapsack, Bin Packing, and Traveling Salesperson problems, and two-dimensional continuous optimization functions such as the Rosenbrock, Ackley, and Eggholder functions. Our results demonstrate that RL Based SA consistently outperforms vanilla SA, adaptive SA, and Neural SA, achieving solution quality comparable to state-of-the-art or problem specific solvers, despite utilizing a significantly more lightweight architecture.

The key advantages of RL Based SA are its generalizability, ease of use, and flexibility. Adding ΔE to the state improves adaptability across different problem instances and sizes, ensuring robust scalability. The use of LSTMs enables the processing of entire SA rollouts, capturing long-term dependencies without significantly increasing computational complexity. With its simple and lightweight architecture, RL Based SA is easy to implement and applicable to diverse optimization tasks without extensive tuning, making it a practical and efficient tool for both discrete and continuous domains.

ACKNOWLEDGMENTS

We wish to acknowledge our PhD student mentor, Ruichen Xu, and our professor, Prof. Yuefan Deng, with guiding us throughout this project and providing crucial insight.

REFERENCES

[1] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. 2016. Neural combinatorial optimization with reinforcement learning. arXiv preprint arXiv:1611.09940 (2016).

[2] Qingpeng Cai, Will Hang, Azalia Mirhoseini, George Tucker, Jingtao Wang, and Wei Wei. 2019. Reinforcement learning driven heuristic optimization. arXiv preprint arXiv:1906.06639 (2019).

[3] Siddhartha Chib and Edward Greenberg. 1995. Understanding the metropolis-hastings algorithm. The american statistician 49, 4 (1995), 327–335.

[4] Vincent A Cicirello. 2021. Self-Tuning Lam Annealing: Learning Hyperparameters While Problem Solving. Applied Sciences 11, 21 (2021), 9828.

[5] Alvaro HC Correia, Daniel E Worrall, and Roberto Bondesan. 2023. Neural simulated annealing. In International Conference on Artificial Intelligence and Statistics. PMLR, 4946–4962.

[6] Paulo R d O Costa, Jason Rhuggenaath, Yingqian Zhang, and Alp Akcay. 2020. Learning 2-opt heuristics for the traveling salesman problem via deep reinforcement learning. In Asian conference on machine learning. PMLR, 465–480.

[7] Daniel Delahaye, Supatcha Chaimatanan, and Marcel Mongeau. 2019. Simulated annealing: From basics to applications. Handbook of metaheuristics (2019), 1–35.

[8] P Kingma Diederik. 2014. Adam: A method for stochastic optimization. (No Title) (2014).

[9] Zhang-Hua Fu, Kai-Bin Qiu, and Hongyuan Zha. 2021. Generalize a small pre-trained model to arbitrarily large tsp instances. In Proceedings of the AAAI conference on artificial intelligence, Vol. 35. 7474–7482.

[10] Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, Wei-Kun Chen, Leon Eifler, Maxime Gasse, Patrick Gemander, Ambros Gleixner, Leona Gottwald, Katrin Halbig, et al. 2020. The SCIP optimization suite 7.0. (2020).

[11] Michael Hahsler and Kurt Hornik. 2007. TSP-Infrastructure for the traveling salesperson problem. Journal of Statistical Software 23, 2 (2007), 1–21.

[12] Keld Helsgaun. 2017. An extension of the Lin-Kernighan-Helsgaun TSP solver for constrained traveling salesman and vehicle routing problems. Roskilde: Roskilde University 12 (2017), 966–980.

[13] S Hochreiter. 1997. Long Short-term Memory. Neural Computation MIT-Press (1997).

[14] Lester Ingber. 2000. Adaptive simulated annealing (ASA): Lessons learned. arXiv preprint cs/0001018 (2000).

[15] David S Johnson. 1973. Near-optimal bin packing algorithms. Ph.D. Dissertation. Massachusetts Institute of Technology.

[16] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. Journal of artificial intelligence research 4 (1996), 237–285.

[17] Mariia Karabin and Steven J Stuart. 2020. Simulated annealing with adaptive cooling rates. The Journal of Chemical Physics 153, 11 (2020).

[18] Wouter Kool, Herke Van Hoof, and Max Welling. 2018. Attention, learn to solve routing problems! arXiv preprint arXiv:1803.08475 (2018).

[19] Alexander G Nikolaev and Sheldon H Jacobson. 2010. Simulated annealing. Handbook of metaheuristics (2010), 1–39.

[20] Fernando Peres and Mauro Castelli. 2021. Combinatorial optimization problems and metaheuristics: Review, challenges, design, and development. Applied Sciences 11, 14 (2021), 6449.

[21] Laurent Perron and Vincent Furnon. [n.d.]. OR-Tools. Google. https://developers.google.com/optimization/

[22] Marius-Constantin Popescu, Valentina E Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. 2009. Multilayer perceptron and neural networks. WSEAS Transactions on Circuits and Systems 8, 7 (2009), 579–588.

[23] Christian P Robert, George Casella, Christian P Robert, and George Casella. 2004. The metropolis—hastings algorithm. Monte Carlo statistical methods (2004), 267–320.

[24] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. 2017. Evolution strategies as a scalable alternative to reinforcement learning. arXiv preprint arXiv:1703.03864 (2017).

[25] Harvey M Salkin and Cornelis A De Kluyver. 1975. The knapsack problem: a survey. Naval Research Logistics Quarterly 22, 1 (1975), 127–144.

[26] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs.LG] https://arxiv.org/abs/1707.06347

[27] Ashish Kumar Shakya, Gopinatha Pillai, and Sohom Chakrabarty. 2023. Reinforcement learning algorithms: A brief survey. Expert Systems with Applications 231 (2023), 120495.

[28] Ilya Sutskever. 2013. Training Recurrent Neural Networks. Ph.D. Dissertation. University of Toronto.

[29] William Uther. 2010. Markov Decision Processes. Springer US, Boston, MA, 642–646. https://doi.org/10.1007/978-0-387-30164-8_512

[30] Peter JM Van Laarhoven, Emile HL Aarts, Peter JM van Laarhoven, and Emile HL Aarts. 1987. Simulated annealing. Springer.

[31] Dhruv Vashisht, Harshit Rampal, Haiguang Liao, Yang Lu, Devika Shanbhag, Elias Fallon, and Levent Burak Kara. 2020. Placement in integrated circuits using cyclic reinforcement learning and simulated annealing. arXiv preprint arXiv:2011.07577 (2020).

[32] Yaoxin Wu, Wen Song, Zhiguang Cao, Jie Zhang, and Andrew Lim. 2021. Learning improvement heuristics for solving routing problems. IEEE transactions on neural networks and learning systems 33, 9 (2021), 5057–5069.

[33] Xiaofeng Xu, Jing Liu, and Jue Wang. 2018. AN–N optimization model for logistic resources allocation with multiple logistic tasks under demand uncertainty. Soft Computing 22, 21 (2018), 7073–7086.

[34] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. 2017. Deep sets. Advances in neural information processing systems 30 (2017).