Optimizing QuickSort Pivot Selection Using Simulated Annealing

Vivek Shandilya
Department of Computer Science
Bowie State University
301-860-3963
vshandilya@bowiestate.edu

Bruce Metoyer
Department of Computer Science
Bowie State University
240-695-2661
metoyerb0612@students.bowiestate.edu

***Abstract*—Pivot selection significantly impacts QuickSort's efficiency, traditionally relying on heuristics such as selecting the first, last, median, or random element. This paper explores an alternative approach: leveraging Simulated Annealing (SA) to dynamically optimize pivot selection. By incorporating controlled randomness and gradual refinement, SA enables the algorithm to escape locally suboptimal pivot choices while converging toward more efficient selections. We propose a model in which SA perturbs pivot selection probabilistically, thereby balancing exploration and exploitation to enhance sorting performance. Our thorough and precise experiments will evaluate this approach against conventional pivot selection methods, measuring its impact on time complexity, sorting stability, and adaptability across diverse input distributions.**

Keywords—Pivot Selection; QuickSort; Heuristics; Simulated Annealing; Time Complexity

I. INTRODUCTION

QuickSort, a cornerstone of computer science, is widely known and efficiently used due to its **O(n log n)** average-case complexity. Its superiority over standard sorting algorithms such as MergeSort is evident in its minimal space requirement and in-place sorting capability. The worst case of QuickSort $\mathbf{O(n^2)}$ can be avoided using randomized quicksort, with a high possibility of choosing the correct pivot. QuickSort's good cache locality makes it faster than merge sort in many cases, like in a virtual memory environment. However, QuickSort's performance is heavily influenced by the selection of the pivot, which affects the efficiency of partitioning. The **first or last element** is often chosen as a pivot, but this approach can lead to the worst case when an array is already sorted. Selecting a **random element** as a pivot is often the preferred choice because it does not exhibit a pattern that leads to the worst-case scenario. Lastly, choosing the **median element** as a pivot is an ideal approach due to its time complexity, which proves that we can find the median in linear time. Additionally, the array will always be divided into two halves by the partition function. This last approach, however, takes more time on average as the median finding has high constraints.

The three most common partition algorithms are **the Naive, Lomuto, and Hoare Partition Algorithms**. All three algorithms have **O(n)** time complexity; however, Hoare's partition is the fastest. QuickSort applications exist today in sorting large datasets with **an O(n log n)** average-case time complexity, partitioning problems such as finding the kth smallest element, or dividing arrays using a pivot. QuickSort is essential for randomized algorithms by offering better performance than deterministic approaches; it is also utilized in cryptography for generating randomized permutations and unpredictable encryption keys, the partitioning step in the algorithm can be parallelized for improved performance in multi-core or distributed systems, and lastly, QuickSort is essential in theoretical computer science for analyzing average-case complexity and developing new techniques.

This paper introduces a **Simulated Annealing** approach for pivot selection in QuickSort. Simulated Annealing is based on metallurgical practices, in which a material is heated to a high temperature and then cooled, involving a process of heating, shaping, and cooling to create a product. In simulated annealing optimization problems, an **initial temperature** is generally set at 1, with a minimum temperature set at 10^{-4} (0.0001). The initial temperature is then multiplied by some fraction alpha and decreased until the minimum temperature is reached. A core optimization routine is run a fixed number of times for each distinct temperature value. This routine finds **neighboring solutions** and accepts them with probability $e^{(f(c) - f(n))}$, where c is the current solution and n is the neighboring solution. A slight perturbation is applied to the current solution to find the neighboring solution. This randomness helps escape the common mistakes of optimization heuristics when trapped in local minima, which are suboptimal solutions near the current solution. By possibly accepting a less optimal solution than we currently have and accepting it with a probability inverse to the increase in cost, the algorithm is more likely to converge near the global optimum. A neighbor function must be done on a case-by-case basis. Some ideas for finding neighbors in locational optimization problems include moving all points 0 or 1 units in a random direction, shifting input elements randomly, swapping random elements in an input sequence, permuting the input sequence, or partitioning the input sequence into random segments and permuting them. To determine where the algorithm begins, we provide the initial solution by either generating a random solution or using prior knowledge of the problem as a starting point.

By learning from previous sorting operations, **Simulated Annealing (SA)** can dynamically explore pivot selections by balancing randomization and optimization. SA enables controlled exploration of potential pivots, preventing the algorithm from getting stuck in locally suboptimal choices, such as choosing a pivot that leads to a highly unbalanced partition while converging toward an efficient solution. The algorithm strategically introduces entropy into the search space, periodically making non-optimal choices to escape poor local minima, and eventually stabilizes near an optimal pivot selection strategy. By incorporating **randomness and systematic refinement**, SA offers a novel perspective on sorting algorithm improvements, bridging the gap between classical heuristics and modern optimization techniques.

II. LITERATURE REVIEW

Bashath and Ismail [1] propose an improved Particle Swarm Optimization (PSO) using SA, underlining its adaptability to high-dimensional search spaces. Similarly, Aylaj and Nouh [5] analyze the performance of classical and degenerated simulated annealing (SA), demonstrating its robustness in optimization tasks. These studies suggest that SA's unique ability to escape local minima can be leveraged to optimize pivot selection in QuickSort. Katsuki et al. [2] apply SA to large-scale combinatorial problems using stochastic computing, highlighting its robustness in handling complex search spaces. Nakada et al. [3] introduce a

hierarchical structure in SA for combinatorial optimization, demonstrating its robustness with improved convergence rates. These findings suggest that SA can enhance decision-making in pivot selection by dynamically identifying optimal partitioning strategies, thereby reinforcing its robustness.

Marcellino et al. [4] comprehensively analyze sorting algorithms and compare QuickSort with techniques such as Heap Sort and Merge Sort, providing insight into their time and memory efficiency. Similarly, Faujdar and Ghrera [8] evaluate sorting algorithms on standard datasets, reinforcing QuickSort's advantages while highlighting its dependency on the choice of pivot. These studies justify the need for an adaptive pivot selection strategy to enhance QuickSort's efficiency, which can be achieved through simulated annealing (SA). CUDA-based implementations of QuickSort have been studied by Ćatić et al. [6], who explore pivot selection and branching avoidance to improve parallel performance. Taotiamton and Kittitornkun [7] investigate a parallel dual-pivot QuickSort using the Lomuto partitioning scheme, demonstrating significant performance improvements. Integrating SA into these methods can optimize pivot selection, potentially improving execution speed and efficiency in parallel computing environments, thereby enhancing system performance.

The versatility of Simulated Annealing is further demonstrated by its successful application to diverse optimization problems beyond sorting. Wulandari [10] utilizes SA for facility layout optimization, showcasing its ability to improve spatial arrangement in manufacturing settings. Cao et al. [11] apply a hybrid SA algorithm for multi-objective optimization in synchronous motors, demonstrating its effectiveness in balancing multiple constraints. These studies confirm SA's adaptability and effectiveness in complex decision-making processes, supporting its application in pivot selection.

Meng [12] and Wang et al. [13] examine SA's role in logistics and crew allocation, respectively, reinforcing its adaptability across domains. Yang et al. [14] extend the use of SA to tobacco sorting systems, combining it with genetic algorithms to enhance efficiency. Similarly, Jangra and Dubran [15] apply SA to cloud computing load balancing, demonstrating its ability to facilitate dynamic resource allocation. These applications demonstrate SA's potential to optimize sorting algorithms by dynamically adjusting pivot selection in response to system constraints.

Finally, Zheng and Zhang [16] introduce a variance-based SA algorithm for logistics cargo sorting, focusing on the optimization of inter-class variance. Their approach parallels the need for improved pivot selection in QuickSort, where variance-based strategies could optimize partitioning.

III. PROBLEM FORMULATION

Mathematically, we define the problem in terms of input, objective, and the method used to optimize QuickSort pivot selection. The key challenge in QuickSort lies in choosing an optimal pivot to balance partitioning. Poor pivot selection results in:

- Unbalanced partitions, increasing the worst-case complexity to $O(n^2)$.
- Excessive swaps and comparisons slow execution time.

We redefine pivot selection as an optimization problem that aims to find the pivot that minimizes sorting time and operations. Formally, given an unordered input array $A=[a_1, a_2, ..., a_n]$, we seek an optimal pivot selection function P(A) that minimizes:

1. **Sorting Time Complexity**: The number of recursive calls required to sort A fully
2. **Swap Operations**: The number of times elements are exchanged.
3. **Comparison Operations**: The number of pairwise comparisons performed.
4. **Cost**: Function for evaluating the efficiency of pivot selection

We aim to optimize P(A) using **Simulated Annealing (SA)**, a probabilistic algorithm designed to escape local minima efficiently and converge towards a near-optimal solution. The SA algorithm will consist of two main components:

1. Outer loop for temperature management
2. Inner loop for solution exploration

The initial temperature, T, will be set at a high level and will be executed in a series of nested iterations, progressively lowered. This will be the cooling schedule, and the iterations will not stop as long as the minimum temperature hasn't been reached. As T decreases, the process continues, and more iterations happen. Random candidate solutions are executed in the neighborhood of the current best solution during each cooling iteration in the inner loop. A new solution for the next iteration is found only when it offers a better objective function value. Otherwise, the solution won't change and may be accepted based on this function:

$$P\big(T, f(x'), f(x)\big) = \exp\left(-\frac{f(x') - f(x)}{T}\right)$$

Mathematical Proof Approach

1. **Pivot Selection as an Optimization Problem:**
   Define a pivot selection function P: $A \rightarrow a_p$, where a_p is the pivot element chosen based on an optimization criterion.
2. **QuickSort Recurrence Relation:**

   $T(n) = T(k) + T(n-k-1) + O(n)$, where k is the position of the pivot after partitioning.
3. **Simulated Annealing Formulation**:
   - Define the cost function J(P), which represents sorting inefficiencies, such as high time complexity or excessive swaps. The cost function reflects how well the chosen pivot helps in efficient sorting. The higher the cost function, the less efficient the method is. The function is evaluated by:
     - **Partition balance**: QuickSort partitions the array into two halves. The cost increases if the partition is highly unbalanced. Find the absolute difference between the differences of the partitions:
       - Balance cost = |(left partition size) - (right partition size)|
     - **Number of Comparisons**: The processing time will increase if there are more comparisons. This

means more comparisons, resulting in higher costs.

- **Number of Swaps**: The more swaps, the higher the costs and the more expensive the operations.

- **Cost Function Forumula**:
  - Cost = w_1(balance cost) + w_2(comparisons) + w_3(swaps), where w_1, w_2, and w_3 are weights that can be adjusted based on their importance. These weights are adjustable and can be modified according to priorities. If partition balance is one of the most essential factors in the cost function, then a higher weight will be assigned to w_1.
- Apply SA to iteratively adjust P(A) to minimize J(P), using probabilistic acceptance of suboptimal pivots to escape local minima.

By proving that SA improves QuickSort's expected-case efficiency, we validate its effectiveness in pivot selection.

IV. PROPOSED SOLUTION

The objective is to optimize QuickSort's pivot selection by implementing **Simulated Annealing (SA)**, enhancing efficiency compared to traditional pivot selection techniques.

1. Define the Problem and Setup

A. We begin by implementing a baseline QuickSort algorithm

- This algorithm uses traditional pivot selection methods (first-element, random-element, last-element, median-element)
- This algorithm also keeps track of the comparisons and swaps it made for each traditional method. This helps track the cost of each strategy to see which is the most efficient.
- This algorithm uses a sample array (unsorted and provided)
- This algorithm displays the pivot strategy, the sorted array, and the total number of swaps, comparisons, and selection costs for each pivot.

2. Optimization Framework

- Independent Variables (Inputs):
  - Array characteristics (size n, distribution type, variance).
    - Previous pivot choices.
  - Dependent Variable (Output):
    - The optimal pivot a_p selected by Simulated Annealing.
  - Controlled Variables:
    - Sorting algorithm: QuickSort.
    - Size & Distribution of data
    - Initial Temperature T and Cooling Schedule in SA.
      - Initial Temperature: 10
      - Minimum Temperature 10^{-4}

3. Implementation of Simulated Annealing

- **Initialization:** Randomly select an initial pivot. Initialize temperature and cooling rate. Define the number of iterations per temperature step.
- **Cost Evaluation:** Compute sorting inefficiency (e.g., imbalance in partition sizes, excessive swaps).
- **Probabilistic Acceptance:** Generating a new pivot candidate by exploring nearby values in the array. If the new pivot improves sorting efficiency, accept it; otherwise, accept it with probability $e^{-\Delta J/T}$, where T is the current temperature.
- **Cooling Schedule:** Reduce T over iterations to refine pivot selection. It will decrease until the stopping threshold.
- We will implement a Simulated Annealing algorithm using a Rastrigin Function. A Rastrigin function is a non-convex function used as a performance test problem for optimization problems. (Standard benchmark problem for optimization algorithms).

4. Performance Evaluation

- **Test Arrays**: Both the Baseline and SA-Optimized QuickSort will be experimented with by multiple different arrays:
  - Sorted array
    - Ex: [1,2,3,4,5,6]
  - Unsorted array
    - Ex: [3,1,69,38,4,20,0,44]
  - Inverse array
    - Ex: [6,5,4,3,2,1]
  - Nearly Sorted
    - Ex: [1,2,3,2,9,0,11,5]
  - Duplicate values
    - Ex: [1,1,1,1,11,1,1]
    - Ex: [1,2,1,2,1,2,1,2]
    - Ex: [1,1,1,1,2,2,2,2]
  - Large and small values
    - Small Values: [10, 2, 4, 8, 2]
    - Large Values: [1000, 2893, 47479, 448783, 7683, 9849]
- **Comparison against Traditional Methods:** Evaluate against median-of-three, random pivot, and first-element selection.
  - The SA approach is evaluated against a traditional QuickSort Pivot Selection approach.

- As stated earlier, the traditional QuickSort approach employs four different methods: choosing the last pivot, the first pivot, a random pivot, and the median pivot. The traditional approach will track the number of swaps, comparisons, and the selection cost needed to sort the array.

- **Final Observations and Results**: Display the data using a graph or table for readability and discuss the findings and data.
- **Performance Metrics:** Measure improvements in swaps, comparisons, and cost.
    - The number of swaps, comparisons, and pivot selection costs will be displayed using the QuickSort algorithm.
    - The simulated annealing function incorporated into QuickSort will also track the number of swaps, total comparisons, and cost.

We implement and compare three separate programs:

1. **Baseline QuickSort** using four traditional pivot strategies. Tracking, cost, swaps, and comparisons as well.
2. **Simulated Annealing Benchmark** applied to the Rastrigin function (used for testing SA implementation)
3. **SA-Optimized QuickSort**, where pivot selection is guided by a cost function and SA strategy

V. METHODOLOGY

5.1 QuickSort Pivot Selection

We implemented four traditional pivot selection strategies in QuickSort: first, last, median, and random. Each strategy was rigorously tested against multiple input scenarios, including:

- Random arrays
- Sorted arrays
- Partially sorted arrays
- Reversed arrays
- Arrays with duplicates
- Arrays with a mix of large and small values.

We designed a cost function to evaluate the performance of each pivot strategy. This function returns a float value that evaluates pivot selection efficiency by combining the number of swaps, comparisons, and a balance factor, which measures partition symmetry:

Cost = swaps + comparisons + (balance factor x array size)

The balance factor is calculated as the absolute difference between the sizes of the left and right partitions divided by the total array size:

$$\text{balance factor} = \frac{|\text{left\_parirtion\_size} - \text{right\_partition\_size}|}{\text{total\_size}}$$

This function captures both the performance (via swaps and comparisons) and the quality of the pivot (via partition balance). Each pivot strategy displays the following metrics:

- Final sorted array
- Number of swaps
- Total number of comparisons
- Pivot selection cost

5.2 Simulated Annealing Optimization

A standalone Simulated Annealing algorithm was implemented to explore optimization capabilities. The objective function used was the Rastrigin function, a non-convex, multimodal function commonly used for benchmarking global optimizers.

Key steps of the algorithm include:

1. Define the objective function (Rastrigin)
2. Generate a random initial solution within bounds
3. Use a neighborhood function to explore nearby solutions
4. Apply a probabilistic acceptance rule based on the current temperature
5. Decrease temperature over 1000 iterations to balance exploration and exploitation
6. Continuously track the best solution and score

In both physics and information theory, temperature represents a measure of disorder, commonly referred to as entropy. At higher temperatures, SA performs larger random jumps, increasing exploration. As the temperature decreases, the jumps will be lower, favoring smaller and more precise steps, which allows it to settle near a global optimum. The SA implementation is a foundation for integrating optimization into the QuickSort pivot selection process.

5.3 SA-Optimized QuickSort

We then integrated the SA into the QuickSort process to create SA-Optimized QuickSort. This hybrid approach aims to address the limitations of traditional pivot selection strategies by dynamically identifying near-optimal pivots during SA. To assess performance, the exact cost function from the conventional QuickSort tests was used, incorporating:

- Number of comparisons
- Number of swaps
- Partition balance (difference in size between left and right subarrays)

Key steps in the SA-QuickSort:

- At each recursive call, SA evaluates all possible pivot indices within the subarray and treats them as candidate solutions
- The temperature can either decrease by recursion depth or by a fixed iteration schedule
- Every 10 iterations, the algorithm logs:
    - Temperature
    - Current pivot index and cost
    - Best pivot index and cost

The best-performing pivot is selected to partition the array. This enables the algorithm to adapt its pivot selection based on the data structure of each subarray, thereby improving efficiency across diverse inputs. Final metrics reported for SA-QuickSort include:

- Selected pivot index, value, and cost for each subarray
- Sorted Array
- Total swaps and comparisons
- Total Pivot Selection Cost
- The initial pivot index, value, and cost
- Every 10th iteration starting from 0-100 displays:
    - Iteration
    - Temperature
    - Current pivot index, value, and cost
    - Best pivot index, value, and cost
    - Best cost

This comparative analysis is intended to determine whether this hybrid approach outperforms traditional pivot strategies in terms of efficiency and adaptability.

VI. Experimentation

6.1 QuickSort Pivot Selection Experimentation

The experiment implements QuickSort with four pivot selection strategies: random element, median element, last element, and first element. The algorithm was implemented in Python using Jupyter Notebook, with lists serving as the core data structure to store the arrays. The algorithm follows these main steps in the process:

1. **Choose the pivot**: Selecting the element from the array as the pivot. In this case, the pivot will be chosen for all strategies, including random, median, last, and first.
2. **Partitioning the array**: Rearrange the array around the pivot. Once partitioning has completed, all elements smaller than the pivot will be on its left while the greater elements will be on its right.
3. **Recursively Call**: QuickSort is called recursively on the left and right subarrays generated by partitioning
4. **Base Case**: When only one element is left in the sub-array, as a single element is already sorted, the recursion stops.

6.1.1 Data Structures and Function Details

The core data structures used are:

- **List(arr)**: The array is used to store and modify the elements during sorting
- **Counters(swap_count, comparison_count, pivot_cost_total)**: Global integers used to track the performance metrics across recursive calls

The main functions for QuickSort implementation were:

1. **Swap (arr, i, j)**: Kept track of the total number of swaps made for each pivot selection strategy.
    1. Parameters:
        1. **Arr**: the list to modify. Initially unsorted.
        2. **i, j:** indices of elements to swap
    2. Operation:
        1. if I does not equal j, exchanges arr[I] and arr[j]
        2. Increment swap_count by 1
2. **pivot_cost(arr, low, high, pivot_index)**: Calculates the cost of the pivot selection strategy. Based on the partition balance and the number of comparisons
    1. Parameters
        1. **Arr**: the original array
        2. **Low**: the starting index of the subarray
        3. **High**: the ending index of the subarray
        4. **pivot_index**: index of the chosen pivot
    2. Operation
        1. Compares all the elements to the pivot value
        2. Measures how balanced the partition would be (difference between left and right sides)
        3. Returns a composite cost combining several comparisons and the partition balance
3. **partition(arr, low, high, pivot_type)**: Partitions the array into two parts based on the pivot and places the pivot in its correct sorted position
    1. Parameters
        1. **Arr**: the original array
        2. **Low**: starting index
        3. **High**: ending index
        4. **pivot_type**: pivot selection strategy (first, last, random, or median
    2. Operation
        1. Selects a pivot index based on pivot_type
        2. Computes and accumulates the pivot selection cost using **pivot_cost()**
        3. Swaps the chosen pivot to the end (high)
        4. Traverse from low to high-1, moving elements smaller than the pivot to the left side using the swap() function
        5. Finally, swaps the pivot into its correct position and returns its new index

The implementation uses Lomuto's partition scheme, where the partition is placed in **O(n)** time. Lomuto's partition is a simple algorithm where the indices of the smaller elements are kept track of and are swapped

4. **quickSort(arr, low, high, pivot_type="last")**: Recursively sorts the array by partitioning it around the pivot and sorting each partition
    1. Parameters:
        1. **Arr**: the original array
        2. **Low**: starting index
        3. **High**: ending index
        4. **pivot_type**: pivot selection strategy
    2. Operation:
        1. If low < high, partitions the array and obtains the pivot index pi
        2. Recursively sorts the subarrays arr[low: pi-1] and arr[pi+1: high]

    Base case: Recursion stops when low is greater than or equal to high (zero or one element).

5. **run_quick_sort(arr, pivot_type)**: Resets all counters and runs QuickSort on a copy of the input array to avoid modifying the original
    1. Parameters
        1. **Arr**: the original array
        2. "**pivot_type**" pivot selection strategy
    2. Operation
        1. Resets global counters: swap_count, comparison_count, and pivot_cost_total
        2. Copies the array
        3. Calls **quickSort()** on the copy
        4. Returns the sorted array and recorded metrics

This experiment uses each pivot selection. For each pivot selection, **run_quick_sort()** is run, the array is sorted, and the number of swaps, comparisons, and pivot selection costs are recorded. The final data includes the performance data for comparing the strategies.

6.2 Simulated Annealing Experimentation

To assess SA in isolation before applying it to QuickSort, we tested it by optimizing the Rastrigin function, a standard benchmark for evaluating global optimization algorithms due to its many local minima. We implemented SA in Python using Jupyter notebook, using simple lists as the primary data structure to represent solutions (vectors of floating-point numbers). The SA algorithm operates in a probabilistic iterative manner:

6.2.1 Data Structures and Function Details

- **Solution Vector(best, current, candidate)**: A Python list of floats, representing the current point in the search space (2D in our case)
- **Score List (scores)**: A Python list that scores the evaluation scores (objective function values) over iterations, used for post-analysis of convergence behavior

The functions for SA included:

1. **objective_function(x):** The Rastigin function computes the "cost" or "energy" for a solution x. Lower values are better, with 0 being the global minimum.
    1. **Formula**: return 10 * len(x) + sum([xi**2 - 10 * math.cos(2 * math.pi * xi) for xi in x])
2. **get_neighbor(x, step_size=0.1)**: Generates a neighboring solution by randomly perturbing a single coordinate of the current solution vector within a small step. Mutating one variable keeps the search local and manageable
3. **Simulated_annealing(objective, bounds, n_iterations, step_size, temp):**
    1. **Bounds**: Limits for each dimension. Without proper domain bounds, SA could wander infinitely
    2. **n_iterations:** Total number of updates. Set to 1000
    3. **step_size**: Maximum amount by which a coordinate can change. The step size is set to 0.1. Too significant a step can skip over reasonable solutions; a small step size enables finer search
    4. **Temp**: the initial temperature. Set to 10

6.2.2 How the Algorithm Operates:

1. **Initialization**: A random solution is generated by sampling uniformly within the provided bounds. It is evaluated immediately to obtain an initial best score
2. **Temperature Schedule**: The temperature dynamically decreases according to t = temp /(i + 1), where i is the current iteration. The temperature gradually decreases in 1000 iterations. This ensures:
    1. High exploration early on (accept worse solutions freely)
    2. Focused exploitation later (accept only better solutions)
3. **Neighbor Generation**: A new candidate solution is created by slightly adjusting one random dimension (get_neighbor function). This local move keeps the search smooth and controlled
4. **Acceptance Criterion**: It is always accepted if the candidate offers a lower cost (a better solution). If it is worse, however, it may still be accepted with a probability: $p=e^{\Delta E/t}$, where ΔE is the difference in scores and t is the temperature. This allows the algorithm to escape local minima, which is crucial for reaching a near-global optimum
5. **Updating Best Solution**: Whenever a better solution than previously recorded is found, it becomes the new best, and its evaluation is added to the scores list
6. **Output**: After 1000 iterations:

1. Best Solution: The vector with the lowest cost
2. Best Score: The corresponding lowest evaluation value
3. Scores Over Time: Can be used to plot and observe convergence behavior

The formula to decrease temperature was used instead of a fixed amount for a gradual decrease, starting high and becoming smaller over time. This helps the algorithm explore widely at first and focus more narrowly later. The best solution means that, out of all vectors tested during the 1000 iterations, the vector with those two values yielded the lowest value when plugged into the Rastrigin function, achieving the best score. This vector is the best approximation to a global minimum found during the search. Lastly, the best score is the lowest value of the objective function (the Rastrigin function) found during optimization. The closer to 0, the better the score and the better the algorithm. These insights support future integration of Simulated Annealing into dynamic pivot selection for QuickSort. Here are the results of three trial runs of Simulated Annealing:

6.3 SA-Optimized QuickSort Experimentation

To evaluate the effectiveness of integrating Simulated Annealing (SA) into QuickSort pivot selection, we designed an enhanced QuickSort that dynamically chooses pivots based on minimizing a custom cost function. We compared the SA-optimized approach against traditional strategies and assessed the improvements in efficiency through swaps, comparisons, and partitioning balancing. This SA-optimized QuickSort approach evaluates multiple candidate pivots within each subarray, searching for a pivot that minimizes a multi-component cost function.

6.3.1 Pivot Cost Function

The cost function for each candidate pivot combines:

- Number of comparisons during partitioning
- Number of swaps caused by the pivot
- Partition balance factor (how evenly the pivot splits the subarray)

Formula:

cost = comparisons + (balance_factor x total_size)

Where

balance_factor = |left_size - right_size|/total_size

The same cost function is used for both SA and traditional pivot strategies to ensure fair compensation. By using the same cost function for both approaches, we ensure the differences in outcomes stem from the chosen pivot selection strategy and the efficiency of each approach. We will analyze comparisons and swaps, primarily focusing on cost.

6.3.2 SA Pivot Selection:

At each recursive QuickSort step:

- SA is initialized with a random pivot in the current subarray [low, high]
- The pivot's cost is calculated
- The algorithm probabilistically accepts better or slightly worse candidates, governed by the cooling schedule:
  - T = initial_temp / i + 1, where i is the current iteration
- Across 100 iterations, neighboring pivot candidates are generated and evaluated

Neighbor Strategy:

- 80% probability: neighbor = current index ± 1 (local search)
- 20% probability: neighbor = randomly selected pivot (global jump)

This exploration pattern balances local refinement and global exploration, which is critical for avoiding local minima in pivot selection

6.3.3 Data Structures and Functions

The core data structures used are:

1. Array arr: list of integers to be sorted
2. Counters:
   1. swap_count: Total number of swaps performed
   2. Comparison_count: Total number of comparisons made
   3. pivot_cost_total: Cumulative cost of all pivot selections
3. Subarrays during recursion: Defined by [low, high] indices

The functions in the algorithm are:

1. **swap(arr, i, j)**:
   1. Parameters
      1. Arr: The array containing elements
      2. i, j: indices of elements to swap
   2. Operation
      1. Swaps the elements at indices i and j in the array
      2. Increments the **swap_count** if a swap occurs
2. **partition(arr, low, high, pivot_index)**:
   1. Parameters
      1. Arr: Array to be partitioned
      2. Low: starting index of the subarray
      3. High: ending index of the subarray
      4. pivot_index: index of the pivot element
   2. Operation
      1. Moves through the pivot element to the end
      2. Partitions the subarray so that elements smaller than the pivot come before it and larger elements come after
      3. Increment **comparison_count** during element comparisons
      4. Returns the final index of the pivot after partitioning
3. **pivot_cost(arr, low, high, pivot_index)**:
   1. Parameters

1. Arr: the array being sorted
2. Low: starting index of the subarray
3. High: ending index of the subarray
4. pivot_index: Candidate pivot index for which the cost is being calculated

2. Operation
    1. Calculates the pivot cost by:
        1. Counting the number of comparisons needed
        2. Measuring how balanced the left and right partitions would be if this pivot were used
    2. Balance factor is computed as the normalized absolute difference between left and right partition sizes
    3. Returns a weighted cost that combines the number of comparisons and the balance factor

4. **simulated_annealing_pivot(arr, low, high, n_iterations=100, initial_temp=10)**:
    1. Parameters
        1. Arr: array to select the pivot from
        2. Low: Starting index of the subarray
        3. High: ending index of the subarray
        4. n_iterations: number of iterations for the SA search (100 iterations)
        5. initial_temp: Starting temperature for annealing. The initial temperature is 10
    2. Operation
        1. Starts with a random pivot candidate within [low, high]
        2. Iteratively explores neighboring pivot candidates, accepting better candidates or probabilistically accepting worse candidates depending on the current temperature
        3. Temperature gradually decreases over iterations (100)
        4. Tracks and prints:
            1. Iteration number
            2. Current temperature
            3. Current pivot index, value, and cost
            4. Best pivot found so far
        5. Returns the index of the selected pivot with the lowest observed cost
        6. Adds the pivot's cost to pivot_cost_total
5. **quickSort(arr, low, high)**:
    1. Parameters
        1. Arr: Array to be sorted
        2. Low: Starting index
        3. High: Ending index
    2. Operation
        1. Recursively sorts the array
        2. Uses **simulated_annealing_pivot** to select the pivot index at each step
        3. Partition the array using the partition
        4. Recursively sorts the left and right subarrays around the pivot
6. **run_quick_sort_with_sa(arr)**:
    1. Parameters
        1. Arr: Array to sort
    2. Operation
        1. Rests global counters (**swap_count, comparison_count, pivot_cost_total**)
        2. Creates a copy of arr
        3. Runs **quickSort** on the copied array
        4. Returns the sorted array along with the total swaps, comparisons, and cumulative pivot selection cost

6.3.4 Performance metrics collected

- Total swaps performed across the sorting process
- Total comparisons made while partitioning
- Total pivot selection cost accumulated across all SA pivot selections

6.3.5 Test Scenarios

- To fully assess the performance of the SA-Optimized QuickSort, we conduct experiments on:
    - Sorted arrays
    - Unsorted arrays
    - Reverse-sorted arrays
    - Nearly Sorted arrays
    - Arrays with duplicate values

VII. RESULTS

The simulated annealing-optimized QuickSort algorithm outperformed traditional QuickSort methods in terms of total pivot selection cost across various array types. While traditional strategies such as first, last, median, and random rely on fixed or randomized pivot choices, SA dynamically adapts pivot selection to minimize imbalance and comparison overhead. The table below summarizes the performance across different test cases.

Performance Table: Traditional vs SA-Optimized QuickSort

AT	LS	LCM	LCO	FS	FCM	FCO	RS	RCM	RCO	MS	MCM	MCO	SAS	SACM	SACO
SA	0	55	110.0	20	55	110.0	12	32	52.0	14	22	26.0	8	22	26.0
UA	12	23	30.0	18	26	40.0	14	25	38.0	20	23	32.0	18	22	26.0
RSA	5	45	90.0	13	45	90.0	15	26	42.0	15	19	24.0	13	19	24.0
NSA	7	20	32.0	11	17	24.0	11	20	34.0	9	18	28.0	13	16	20.0
DA1	30	465	930.0	60	465	930.0	56	465	930.0	60	465	930.0	58	465	930.0
DA2	41	134	234.0	70	135	238.0	61	134	234.0	64	134	234.0	58	134	234.0
DA3	14	41	72.0	24	36	62.0	22	41	72.0	23	41	72.0	20	36	62.0

Abbreviation Legend

Pivot Selection Strategy Metrics:

1. **LS** - Last Pivot selection strategy total swaps
2. **LCM** - Last Pivot selection strategy total comparisons
3. **LCO** - Last Pivot selection strategy total cost
4. **FS** - First Pivot Selection strategy total swaps
5. **FCM** - First Pivot selection strategy total comparisons
6. **FCO** - First Pivot selection strategy total cost
7. **RS** - Random Pivot selection strategy total swaps
8. **RCM** - Random Pivot selection strategy total comparisons
9. **RCO** - Random Pivot selection strategy total cost
10. **MS** - Median Pivot selection strategy total swaps
11. **MCM** - Median Pivot selection strategy total comparisons
12. **MCO** - Median Pivot selection strategy total cost
13. **SAS** - Simulated Annealing QuickSort Total Swaps
14. **SACM** - Simulated Annealing Total Comparisons
15. **SACO** - Simulated Annealing Final Total Cost

AT - Array Types:

16. **SA** - Sorted Array
17. **UA** - Unsorted Array
18. **RSA** - Reverse/Inverse Sorted Array
19. **NSA** - Nearly Sorted Array
20. **DA1, DA2, DA3** - Duplicate Arrays

Test Arrays:

1. **SA**: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2. **UA**: [20, 382, 78, 292, 1000, 889, 2, 10, 92, 3, 11]
3. **RSA**: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
4. **NSA**: [11, 12, 13, 14, 2, 1, 20, 30, 19]
5. **DA1**: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
6. **DA2**: [100, 49, 90, 100, 49, 90, 100, 49, 90, 100, 49, 90, 100, 49, 90, 100, 49, 90, 100, 49, 90, 100, 49, 90, 100, 49, 90, 100, 49, 90]
7. **DA3**: [10, 4, 10, 4, 10, 4, 10, 4, 10, 4, 10, 4]

Performance Table: Average metrics

	Last	First	Random	Median	SAQS
Swaps	15.5	30.8	27.28	29.28	26.28
Comparisons	111.85	111.28	106.14	103.14	102
Cost	214	213.42	200.28	192.28	188.85

Displays the average amount of swaps, comparisons, and cost of traditional pivot selection strategies and the simulated annealing selection strategy for QuickSort

We observed several trends across key performance metrics, including swaps, comparisons, and pivot selection costs, when analyzing the data. This helps contextualize the benefits and trade-offs of using simulated Annealing for pivot selection in QuickSort. Regarding the total number of swaps, we observe that the last pivot selection strategy yielded 0 swaps when tested on a sorted array. Naturally, this makes the most sense, though, as choosing the previous element in a pre-sorted array as the pivot leads to highly unbalanced partitions, but no rearrangement is needed. The array is already in order. However, this approach does not perform optimally when dealing with already sorted arrays.

Among the traditional strategies, Random Pivot Selection performed relatively well, achieving the lowest number of swaps (11) on the Nearly Sorted Array (NSA), an array that is mostly sorted but with a few elements out of place. On average, the Last Pivot strategy produced 15.5 swaps, and the Random Pivot averaged 27.28 swaps. By contrast, the Simulated Annealing (SA) Optimized QuickSort produced an average of 26.8 swaps, placing it second-best overall in terms of minimizing swaps. This performance can be attributed to SA's ability to probabilistically explore and refine pivot choices, often selecting pivots that divide the array more evenly, thus reducing the number of unnecessary element exchanges. While not consistently superior to the Random or Last strategies in every case, it offers more consistent performance across various array types. The SA approach showed no significant edge in minimizing swaps alone. Still, it remained competitive and more stable than strategies that perform well only under specific conditions, such as Last Pivot on sorted arrays.

When analyzing the number of total comparisons, the Median and Random pivot strategies produced the most efficient results among the traditional methods, with averages of 103.14 and 106.14, respectively. This is consistent with theoretical expectations: the median pivot provides more balanced partitions, while random pivot selection avoids worst-case scenarios associated with fixed-position strategies. Interestingly, all strategies yielded identical comparison counts when sorting Duplicate Arrays, which suggests that the pivot choice had a minimal effect on the partitioning structure due to the uniformity of the values. For example, all strategies required 465 comparisons for Duplicate Array 1, 134 for Duplicate Array 2, and 41 for Duplicate Array 3.

The SA QuickSort strategy achieved an average of 102 comparisons, the lowest among all methods. This reflects its ability to adaptively find pivots that approach the ideal of a median without requiring the complete computation of a median. Its comparative advantage was particularly evident in unsorted

and nearly sorted arrays, where unbalanced partitioning tends to inflate comparison counts. SA outperformed all other methods in minimizing comparisons, confirming its efficiency and adaptability in various scenarios.

The pivot selection cost provides a critical measure of the efficiency of each strategy's selection mechanism. Among the approaches, we see that the median achieved the best results with an average cost of 192.28, Last and First had the worst results, averaging 214 and 213.42, respectively, while Random averaged 200.28. We again observed consistent data when analyzing duplicate arrays, with average costs of 930 for duplicate 1, 234-238 for duplicate 2, and 62-72 for duplicate 3. This indicates that pivot cost is sensitive to input structure but not heavily influenced by pivot randomness when values are identical.

However, we saw a noticeable difference in the data compared to the SA QuickSort approach. The SA QuickSort approach had the lowest average cost of 188.85, outperforming even the Median Pivot. This outcome can be explained by SA's use of a probabilistic cost function, which balances exploration (trying suboptimal pivots) and exploitation (selecting better-performing pivots). As the algorithm progresses and the temperature decreases, SA tends to settle on pivot selections that reduce partitioning imbalance and improve selection efficiency. SA is the most cost-effective pivot strategy, offering savings over deterministic strategies without sacrificing performance.

Across all metrics, the Simulated Annealing-Optimized QuickSort approach demonstrated strong, balanced performance, being the most efficient in comparisons and pivot cost, most efficient in swaps, consistently strong across all input types, and robust performance for duplicate-heavy datasets. Its adaptability to various configurations, including nearly sorted or inversely sorted, provides a strong and adaptable alternative for large or complex datasets. However, SA QuickSort still has its disadvantages. It introduces overhead due to the added complexity of the SA algorithm itself, including parameter tuning such as cooling rate, initial temperature, and iteration limits, as well as a slightly higher runtime complexity resulting from repeatedly evaluating multiple pivot candidates. This overhead may not always translate into practical performance gains for small datasets or real-time systems, even though it is acceptable for theoretical or experimental comparisons. SA can provide a robust and adaptable alternative for large or complex datasets that are more challenging for traditional pivot strategies.

In summary, the simulated annealing quicksort approach offers advantages in terms of performance, particularly in comparison and selection costs, while maintaining competitive swap performance. It avoids the disadvantages of the traditional pivot selection strategies and adapts well to diverse array configurations due to its randomization and measure of disorder. While it may not always be optimal in every individual metric in this case, its overall efficiency and consistency make it a strong candidate for enhancing QuickSort performance in non-trivial applications.

VIII. ACKNOWLEDGEMENTS

I want to thank the Department of Computer Science at Bowie State University for facilitating this course. This course enabled me to develop research skills while also gaining knowledge about an interesting topic in simulated annealing. I would also like to thank the staff and peers who have assisted me throughout this journey.

IX. CONCLUSION

In conclusion, this research presents a novel approach to improving QuickSort's pivot selection using Simulated Annealing (SA), addressing key limitations of traditional strategies, including first, median, last, and random pivot selection. The SA-Optimized QuickSort demonstrated strong adaptability and consistent performance across various input types, including sorted, reverse-sorted, unsorted, partially sorted, and duplicate arrays. By incorporating a cost function and a probabilistic search strategy, SA outperformed other methods in terms of comparison count and pivot selection cost, while remaining competitive in swap efficiency.

Despite these advantages, the approach introduces several limitations. Its performance is susceptible to parameter tuning, including the number of iterations, initial temperature, and cooling schedule, which influence the balance between exploration and refinement. These parameters are data-dependent and require careful adjustment to achieve optimal performance. Additionally, the SA approach may converge to a local minimum due to its probabilistic nature, especially if the temperature decreases too quickly or the search space is irregular, resulting in variable outcomes across different runs. Like traditional strategies, SA-QuickSort also showed reduced effectiveness on datasets dominated by duplicate values.

Future work could further explore the integration of standard neural networks with Simulated Annealing or other heuristic techniques to enhance pivot selection. A hybrid model could allow neural networks to learn from previous sorting behavior and guide SA's search more effectively, while SA would retain its ability to escape local minima. Additional directions include designing specialized strategies for handling duplicate values and combining SA with other metaheuristics, such as Genetic Algorithms (GA) or Particle Swarm Optimization (PSO). These enhancements could yield more robust, adaptive, and efficient sorting algorithms capable of handling various data scenarios.

X. REFERENCES

[1] S. Bashath and A. R. Ismail, "Improved Particle Swarm Optimization By Fast Simulated Annealing Algorithm," 2019 International Conference of Artificial Intelligence and Information Technology (ICAIIT), Yogyakarta, Indonesia, 2019, pp. 297-301, doi: 10.1109/ICAIIT.2019.8834515.

[2] K. Katsuki, D. Shin, N. Onizawa, and T. Hanyu, "Fast Solving Complete 2000-Node Optimization Using Stochastic-Computing Simulated Annealing," 2022 29th IEEE International Conference on Electronics, Circuits and Systems (ICECS), Glasgow, United Kingdom, 2022, pp. 1-4, doi: 10.1109/ICECS202256217.2022.9971124.

[3] K. Nakada, D. Sekii, K. Tamura and K. Yasuda, "Combinatorial Optimization Method Based on Hierarchical Structure in Solution Space with Stochastic Neighborhood Selection," 2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC), Honolulu, Oahu, HI, USA, 2023, pp. 5058-5063, doi: 10.1109/SMC53992.2023.10394606.

[4] M. Marcellino, D. W. Pratama, S. S. Suntiarko and K. Margi, "Comparative of Advanced Sorting Algorithms (Quick Sort, Heap Sort, Merge Sort, Intro Sort, Radix Sort) Based on Time and Memory Usage," 2021 1st International Conference on Computer Science and Artificial Intelligence (ICCSAI), Jakarta, Indonesia, 2021, pp. 154-160, doi: 10.1109/ICCSAI53272.2021.9609715.

[5] B. Aylaj and S. Nouh, "Degeneration vs Classical of simulated annealing algorithm: performance analysis," 2022 5th International Conference on Advanced Communication

Technologies and Networking (CommNet), Marrakech, Morocco, 2022, pp. 1-5, doi: 10.1109/CommNet56067.2022.9993814.

[6] I. Ćatić, M. Mujić, N. Nosović and T. Hrnjić, "Enhancing Performance of CUDA Quicksort Through Pivot Selection and Branching Avoidance Methods," 2023 XXIX International Conference on Information, Communication and Automation Technologies (ICAT), Sarajevo, Bosnia and Herzegovina, 2023, pp. 1-5, doi: 10.1109/ICAT57854.2023.10171304.

[7] S. Taotiamton and S. Kittitornkun, "A Parallel Dual-Pivot QuickSort Algorithm with Lomuto Partition," 2017 21st International Computer Science and Engineering Conference (ICSEC), Bangkok, Thailand, 2017, pp. 1-5, doi: 10.1109/ICSEC.2017.8443883.

[8] N. Faujdar and S. P. Ghrera, "Analysis and Testing of Sorting Algorithms on a Standard Dataset," 2015 Fifth International Conference on Communication Systems and Network Technologies, Gwalior, India, 2015, pp. 962-967, doi: 10.1109/CSNT.2015.98.

[9] I. Ćatić, M. Mujić, N. Nosović and T. Hrnjić, "Enhancing Performance of CUDA Quicksort Through Pivot Selection and Branching Avoidance Methods," 2023 XXIX International Conference on Information, Communication and Automation Technologies (ICAT), Sarajevo, Bosnia and Herzegovina, 2023, pp. 1-5, doi: 10.1109/ICAT57854.2023.10171304.

[10] R. Wulandari, "Improving Facility Layout Using Genetic Algorithm and Simulated Annealing," 2024 International Conference on Intelligent Cybernetics Technology & Applications (ICICyTA), Bali, Indonesia, 2024, pp. 876-881, doi: 10.1109/ICICYTA64807.2024.10913089.

[11] X. Cao, G. Li, Q. Ye, R. Zhou, G. Ma and F. Zhou, "Multi-objective optimization of permanent magnet synchronous motor based on elite retention hybrid simulated annealing algorithm," 2017 12th IEEE Conference on Industrial Electronics and Applications (ICIEA), Siem Reap, Cambodia, 2017, pp. 535-540, doi: 10.1109/ICIEA.2017.8282902.

[12] T. Meng, "Research on Distribution Path Optimization of Warehousing Management System Based on Simulated Annealing Algorithm," 2024 4th International Signal Processing, Communications and Engineering Management Conference (ISPCEM), Montreal, QC, Canada, 2024, pp. 1068-1072, doi: 10.1109/ISPCEM64498.2024.00188.

[13] H. Wang, J. Xia and Y. Song, "A Study of Crew Allocation Based on a Novel Heuristic Algorithm and Simulated Annealing Algorithm," 2024 IEEE 2nd International Conference on Image Processing and Computer Applications (ICIPCA), Shenyang, China, 2024, pp. 1564-1568, doi: 10.1109/ICIPCA61593.2024.10709305.

[14] Y. Yang, J. Wang, Z. Wu, M. Luo, L. Wei and Z. Li, "Simulation Optimization of Tobacco Sorting System Based on Genetic Simulated Annealing Algorithm," 2024 IEEE 2nd International Conference on Electrical, Automation and Computer Engineering (ICEACE), Changchun, China, 2024, pp. 928-933, doi: 10.1109/ICEACE63551.2024.10898222.

[15] A. Jangra and H. Dubran, "Simulation Annealing Based Approach to Enhanced Load Balancing in Cloud Computing," 2021 9th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO), Noida, India, 2021, pp. 1-4, doi: 10.1109/ICRITO51393.2021.9596215.

[16] J. Zheng and P. Zhang, "A Simulated Annealing Algorithm based on Variance Consensus for Logistics Cargo Sorting," 2024 IEEE 6th Advanced Information Management, Communications, Electronic and Automation Control Conference (IMCEC), Chongqing, China, 2024, pp. 322-326, doi: 10.1109/IMCEC59810.2024.10575637.

[17] GeeksforGeeks, “Quick Sort Algorithm,” *GeeksforGeeks*, Jan. 07, 2014. https://www.geeksforgeeks.org/quick-sort-algorithm/

[18] GeeksforGeeks, “Implement Simulated Annealing in Python,” *GeeksforGeeks*, Jun. 2024. https://www.geeksforgeeks.org/implement-simulated-annealing-in-python/