Parallel Simulated Annealing Optimization In Heterogeneous Environments

Suraj Anand, Mason Lee
Brown University
`suraj_anand@brown.edu, mason_lee@brown.edu`

1 Introduction

Optimization in complex problem domains, characterized by high-dimensional, nonconvex, and multimodal landscapes, poses significant challenges to traditional gradient-based methods. The primary limitation of gradient descent and its variants arises in their dependence on the differentiability of the objective function and their vulnerability to becoming ensnared in local optima in nonconvex settings. This is particularly problematic in scenarios where the solution space exhibits a high degree of ruggedness or when the objective function is nondifferentiable, discontinuous, or computationally burdensome to evaluate.

Simulated Annealing (SA), a probabilistic heuristic inspired by the annealing process in metallurgy, offers an alternative optimization strategy in these intricate domains. SA's core mechanism, which probabilistically accepts worse solutions, permits the algorithm to surmount local optima, a capability that gradient-based approaches inherently lack. This feature of SA is vital in exploring nonconvex, multimodal landscapes, making it a robust choice for complex, high-dimensional problems [2].

The evolution of SA methodologies has led to the development of Multi-Start Simulated Annealing (MSA), which involves running multiple independent SA processes. While this enhances the exploration of the solution space, it does not inherently facilitate interaction or information sharing among the individual processes [1].

Building upon this concept, we focus on Coupled Simulated Annealing (CSA), an advanced variant that amalgamates multiple SA processes into a cooperative framework. CSA leverages the collective behavior of these processes, enabling them to share information and thus guide each other out of local optima, enhancing the overall convergence to global optima [7].

The paper's primary focus is the design and comparison of parallelized MSA and CSA for non-convex optimization in heterogeneous computing environments combining GPU and CPU resources. This exploration is crucial in the current computational landscape, where leveraging the synergistic capabilities of diverse computing architectures can significantly amplify the efficiency and scalability of optimization algorithms.

In essence, this paper addresses a pivotal challenge in the optimization of intricate problem spaces by presenting CSA as a potent, scalable solution. The exploration of CSA's parallel implementation on heterogeneous computing platforms marks a significant stride in the practical application of advanced optimization techniques in various domains, from scientific computations to machine learning.

1.1 Simulated Annealing

Temperature Schedules and Acceptance Probability

The variables T_k and T_{ac}^k are the generation and acceptance temperature parameters at time instant k, respectively. The generation distribution $g(\varepsilon, T_k)$ and the acceptance probability function $A(x \to y)$ are crucial in the SA process. They are defined as follows:

- **Generation Distribution ($g(\varepsilon, T_k)$):** This function generates a probing solution by adding a perturbation to the current solution. The nature of the perturbation depends on the current temperature, influencing the exploration of the solution space.

Algorithm 1 Naive Simulated Annealing

```
1: Initialization: Assign a random initial solution to $x$. Assess its cost $E(x)$. Set the initial temperatures $T_k = T_0$ and $T_{ac}^k = T_{ac}^0$. Set the time index $k = 0$.
2: Generate a probing solution: $y = x + \varepsilon$, where $\varepsilon$ is a random variable sampled from a given distribution $g(\varepsilon, T_k)$. Assess the cost for the new probing solution $E(y)$.
3: Accept solution: Accept solution $y$ with probability 1 if $E(y) \leq E(x)$; otherwise, with probability $A(x \to y)$, i.e., make $x := y$ only if $A > r$, where $r$ is a random variable sampled from a uniform distribution $[0, 1]$. Go to Step 2 for $N$ inner iterations (equilibrium criterion).
4: Decrease temperatures: According to schedules $U(T_k, k)$ and $V(T_{ac}^k, k)$. Increment $k$.
5: Stopping Criterion: Stop if stopping criterion is met; otherwise, go to Step 2.
```

- **Acceptance Probability ($A(x \to y)$):** Given the current and probing solutions x and y, the acceptance probability is determined by the Metropolis criterion:

$$A(x \to y) = \min\left(1, \exp\left(\frac{E(x) - E(y)}{T_{ac}^k}\right)\right) \tag{1}$$

  This equation allows the algorithm to probabilistically accept worse solutions, aiding in escaping local minima and facilitating a more global search.

The acceptance and generation temperature schedules, $U(T_k, k)$ and $V(T_{ac}^k, k)$, respectively, are functions that govern how the temperature parameters are decreased over time, controlling the balance between exploration and exploitation in the search process.

1.2 Multi-Start Simulated Annealing (MSA)

Multi-Start Simulated Annealing (MSA) is an extension of the basic SA algorithm where multiple SA processes are run independently. Each process starts from a different initial solution, increasing the probability of finding a global optimum.

Algorithm 2: Multi-Start Simulated Annealing

Algorithm 2 Multi-Start Simulated Annealing (MSA) [6]

```
1: Initialization: Initialize $N$ independent SA processes. For each process $i$, assign a random initial solution $x_i$, assess its cost $E(x_i)$. Set initial temperatures $T_{k_i} = T_0$ and $T_{ac}^{k_i} = T_{ac}^0$. Set time index $k_i = 0$.
2: Parallel SA Execution: Each process executes the steps of the Naive Simulated Annealing algorithm independently.
3: Collecting Results: Gather the best solutions $\{x_i^*\}$ from each SA process.
4: Select Best: Choose the best solution $x^* = \arg\min_{x_i^*} E(x_i^*)$ from all processes.
```

MSA increases the exploration capability by diversifying the search across different regions of the solution space, thus reducing the risk of converging to local optima [6].

1.3 Coupled Simulated Annealing (CSA)

Coupled Simulated Annealing (CSA) introduces a cooperative mechanism among multiple SA processes. In CSA, processes are not only independent but also interact with each other by exchanging information, which guides the search process.

CSA's main advantage lies in its collaborative search strategy, allowing processes to learn from each other, thus enhancing the exploration and exploitation balance. This collaborative approach is particularly effective in complex optimization problems where the solution landscape is rugged and multi-modal [7].

Comparison of MSA and CSA:

- **Independence vs. Cooperation:** While MSA relies on independent runs of SA, CSA fosters cooperation among parallel processes.
- **Exploration:** MSA enhances exploration by diversifying initial conditions. CSA, in contrast, achieves exploration through interaction among processes, potentially leading to a more effective search.

Algorithm 3 Coupled Simulated Annealing (CSA) [7]

```
1: Initialization: Initialize $N$ SA processes with random solutions $\{x_i\}$. Set initial temperatures $T_{k_i} = T_0$ and $T_{ac}^{k_i} = T_{ac}^0$ for each process. Set time indices $\{k_i = 0\}$.
2: Parallel Execution with Interaction: Each process executes the steps of the Naive Simulated Annealing algorithm. Periodically, processes interact as follows:
    1. Exchange Mechanism: At specific intervals, each process shares its current best solution $x_i^*$ with others or incorporates information from others to influence its search.
3: Synchronization and Update: Processes synchronize at certain steps to update their states based on the exchanged information.
4: Finalization: Collect final solutions $\{x_i^*\}$ from each process and select the best one $x^* = \arg\min_{x_i^*} E(x_i^*)$.
```

- **Convergence:** CSA's cooperative approach can lead to faster convergence and a higher likelihood of finding the global optimum compared to MSA.

Both MSA and CSA are valuable extensions of the SA algorithm, each with its strengths in tackling different types of optimization problems.

2 Problem Statement

2.1 The Schwefel Function

The Schwefel function is a well-known test function used in the field of optimization, particularly for evaluating the performance of algorithms like Simulated Annealing. It is defined as:

$$f(\mathbf{x}) = 418.9829d - \sum_{i=1}^{d} x_i \sin(\sqrt{|x_i|}), \quad \mathbf{x} \in [-500, 500]^d \tag{2}$$

where d is the dimensionality of the problem and $\mathbf{x} = (x_1, x_2, ..., x_d)$ is a vector in the domain of the function.

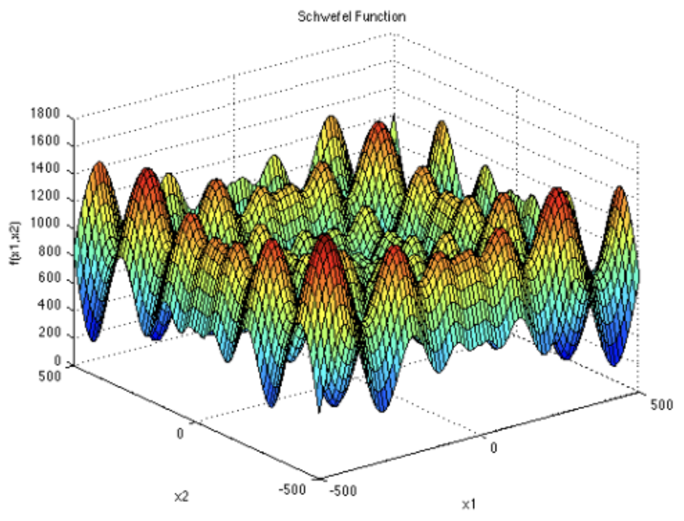

Figure 1: Schwefel Function (3-Dimensional) Optimization Landscape [5]

Ideal Use Case

The Schwefel function is particularly useful for testing optimization algorithms because:

- It has a large search space with many local minima, making it challenging for optimization algorithms.

- The global minimum is located away from the center of the search space, which tests an algorithm's ability to avoid local minima and explore effectively.

Linear Separability

The Schwefel function is linearly separable, which means it can be expressed as a sum of functions that each depend on a single variable x_i. This property is evident from its definition as a sum over dimensions. Each term in the summation depends only on x_i, and thus the optimization of each dimension can be considered independently. This property makes the Schwefel function a useful benchmark for algorithms in high-dimensional optimization problems [5].

Other Properties

Other notable properties of the Schwefel function include:

- **Non-convexity**: The function is non-convex, which means it has multiple local minima. This property is essential for testing the global optimization capabilities of algorithms.
- **Continuity and Differentiability**: The function is continuous and differentiable, although this is not necessary for simulated annealing.
- **Global Minimum**: The global minimum of the Schwefel function is located at $\mathbf{x}^* = (420.9687, ..., 420.9687)$ with $f(\mathbf{x}^*) \approx 0$, which provides a clear target for optimization algorithms to aim for [5].

3 Experiments

In the following experiments, we aim to evaluate and enhance the performance of global optimization algorithms, specifically focusing on Multi-Start Simulated Annealing (MSA) and Coupled Simulated Annealing (CSA). Our primary objectives are to assess the efficiency of various optimization strategies in different computational environments and to investigate the impact of parallel processing techniques on the overall computational time and accuracy of these algorithms. These experiments are designed to provide insights into the effectiveness of function inlining, OpenMP-based thread parallelization, and CUDA kernel implementation in optimizing the simulated annealing process for the Schwefel function. We chose these experiments by evaluating which parts of the minimization process took the most time. Through these investigations, we hope to establish a set of best practices for implementing and optimizing MSA and CSA in both single-threaded and parallel computing contexts. We tested out our approaches on a 10 and 100 dimensional function optimization landscape.

3.1 Naive Annealing

To establish a foundation for performance comparisons, we implemented basic versions of MSA and CSA using a single-threaded, sequential approach. This naive implementation served as a baseline, allowing us to quantify improvements from subsequent optimizations. Each of the n simulated annealing processes was performed in sequence, ensuring a clear understanding of the algorithm's base efficiency.

3.2 Function Inlining

Function inlining was our first optimization strategy. The concept behind inlining is to reduce the overhead associated with function calls. Specifically, we experimented with inlining the

1. Minimize Function - minimizes the objective function, called once at beginning
2. Step Function - takes a step every iteration (per SA process)
3. Objective Function f - evaluates the cost of a particular n-dimensional point

3.3 OpenMP Thread Parallelization

Leveraging OpenMP, a shared-memory parallel programming model, we introduced parallelism in our algorithms for execution on multi-core CPUs. In the case of MSA, each thread operated independently, converging only upon reaching the maximum number of iterations to return the optimal SA process. Conversely, in CSA, threads synchronized after every iteration, facilitating communication for shared parameter updates (mainly to reduce variance of acceptance probabilities). To prevent race conditions while updating these shared parameters, we employed a mutual exclusion lock provided by OpenMP.

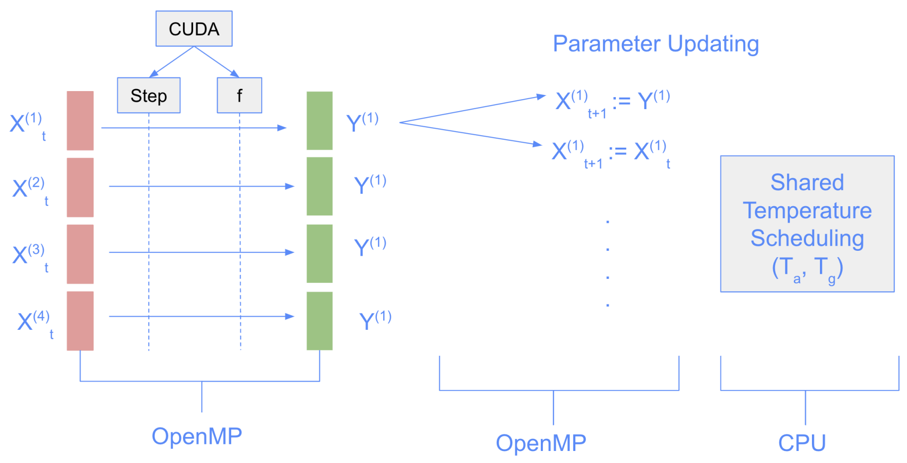
CUDA
Step
f
$X^{(1)}_t$
$X^{(2)}_t$
$X^{(3)}_t$
$X^{(4)}_t$
$Y^{(1)}$
Parameter Updating
$X^{(1)}_{t+1} := Y^{(1)}$
$X^{(1)}_{t+1} := X^{(1)}_t$
Shared Temperature Scheduling (T_a, T_g)
OpenMP
OpenMP
CPU

Figure 2: Parallelization Models Used in Coupled Simulated Annealing (CSA) Algorithm (MSA Model is similar but uses OpenMP for Individual Temperature Scheduling)

3.4 CUDA Kernels

The experiments investigate CUDA for parallel processing for the "step" function and objective function "f." For the "step" function, each thread computes a step in a different dimension. For the objective function, each thread computes a step in a different dimension i for a particular simulated annealing process j in $X_t^{(i,j)}$. We are able to parallelize this due to the linear separability of dimensions for the Schwefel function. For f, we also test using warps to sum across dimension quickly using `shfl_down`. Note that when we employed CUDA kernels in addition to OpenMP, each OpenMP thread would run a CUDA kernel for taking a step and evaluating the objective function. We illustrate this in Figure 2.

4 Results

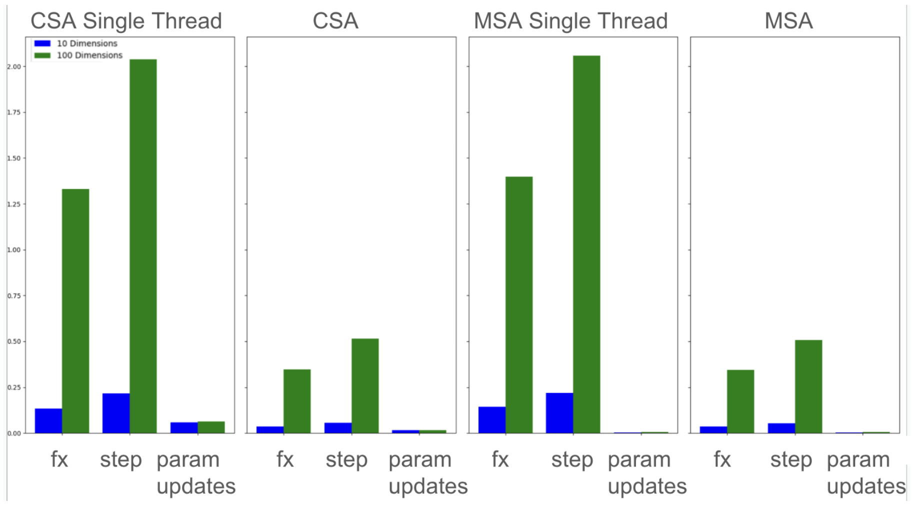

Figure 3: Total Algorithm Timing of f, step, parameter evaluation during Minimization

We compared execution time for algorithmic parts of CSA and MSA with Single and Multiple Threads (using OpenMP) in Figure 3. Note that the shared parameter updates for CSA take longer than MSA; however, the difference in parameter update times shrinks when using Multiple Threads even though CSA requires locking for parameter sharing (as this is only for a few computations). In addition, our results show that thread parallelization provides almost a linear speedup - we use four threads rather than one (one for each simulated annealing processes) and reach almost four times a speedup. This is apparent in both 10 and 100 dimensional optimization landscapes. Our results also show that while parameter updates take the roughly the same amount of time for 10 and 100 dimensional setups, the objective function and step scale quasi-linearly.

We found that inlining marginally sped up operations by about 2000 microseconds for the single thread MST run. The inlining of f and $step$ was were most of this speedup occured as they are small functions that were called once per iteration.

Time (Microseconds)	**MSA Single Thread**	**MSA**	**CSA Single Thread**	**CSA**
CPU	3,493,635	876,549	3,884,980	995,846
GPU + CPU	1,358,433	344,000	1,563,125	440,082
GPU (Warping) + CPU	1,208,433	321,389	1,413,214	400,981

Table 1: Simulated Quantum Annealing Schwefel Function Optimization (1000000 Iterations)

In Table 1, we show results across CPU and heterogeneous CPU + GPU experiments for a 10-dimensional setup. From these experiments, it is evident that GPU one-thread per dimension usage reduces minimization time to a third of the original value. When we use warping, this provides further gain. We would expect this gap between CPU and GPU timing to increase further for higher dimensional setups. This timing also shows us that even with additional memory copying, there exists GPU speed up even for not extremely high dimensional setups (this was suprising to us).

Kernel	**FLOPs (per iter)**	**Memory Ops (per iter)**	**Arithmetic Intensity**
Function Kernel	25	2	12.5
Warped Function Kernel	25	2 + 1 (shared mem write)	8.33
Step Kernel	23	2	11.5

Table 2: Kernel operations

We calculated arithmetic intensity for each operation in Table 2. While the Warped Function Kernel has a slightly smaller arithmetic intensity, the shared memory operation is faster than summing up on the CPU in a sequential method as evident from Table 1. We estimated the FLOPs of $\sin .$ and $\sqrt{.}$ by estimation provided in [3].

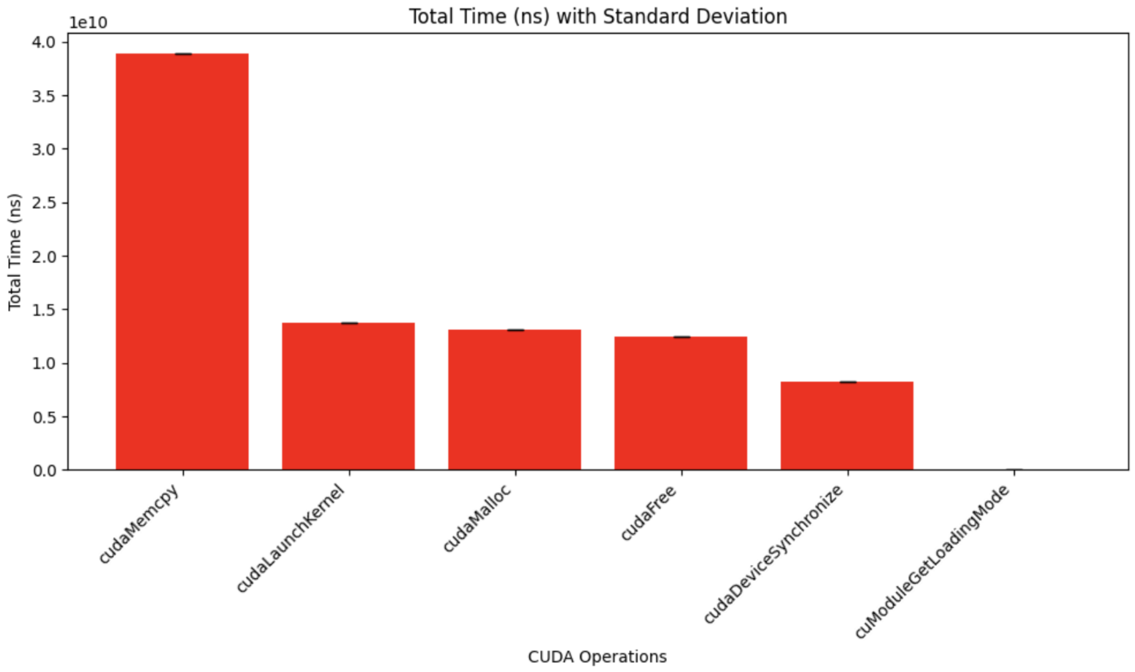

Figure 4: CUDA Total Time Profiling

Finally, in Figure 4 we can see CUDA time usage by function. Around 42% of CUDA time is Memory Copying, 16% is Kernel Lanching, 16% is CUDA Memory Allocating, and 15% is CUDA Freeing. Further experiments should be conducted to examine Unified Memory model usage to reduce copying expenses. [3]

5 Discussion

We have presented comparative experiments for the design of several classes of global optimization methods, including variants of Multi-Start Simulated Annealing and Coupled Simulated Annealing (Single Threaded, Multi Threaded, Multi Threaded with CUDA Kernals). We show the speed tradeoff of information sharing to control acceptance probability variance in Coupled Simulated Annealing. Additionally, we demonstrate that heterogeneous programming significantly accelerates the processing of linearly separable functions, particularly in high-dimensional spaces.

Future research should focus on evaluating performance in even higher-dimensional spaces to further understand the capabilities and limitations of this approach. In addition, future work should explore more sophisticated memory copying techniques, which could optimize the efficiency and speed of processing in these complex computational environments.

References

[1] Ge-Fei Hao, Wei-Fang Xu, Sheng-Gang Yang, et al. "Multiple Simulated Annealing-Molecular Dynamics (MSA-MD) for Conformational Space Search of Peptide and Miniprotein". In: *Scientific Reports* 5 (2015), p. 15568. DOI: 10.1038/srep15568. URL: https://doi.org/10.1038/srep15568.

[2] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. "Optimization by Simulated Annealing". In: *Science* 220.4598 (1983), pp. 671–680. DOI: 10.1126/science.220.4598.671. eprint: https://www.science.org/doi/pdf/10.1126/science.220.4598.671. URL: https://www.science.org/doi/abs/10.1126/science.220.4598.671.

[3] Latkin. *A Simple Benchmark of Various Math Operations*. Accessed: [2023]. Nov. 2014. URL: https://latkin.org/blog/2014/11/09/a-simple-benchmark-of-various-math-operations/.

[4] Florian Neukart et al. *Traffic flow optimization using a quantum annealer*. 2017. arXiv: 1708.01625 [quant-ph].

[5] Simon Fraser University. *Schwefel's Function - Optimization Test Problems*. https://www.sfu.ca/~ssurjano/schwef.html. Accessed: [insert date here].

[6] Emrullah Sonuc, Baha Sen, and Safak Bayir. "A cooperative GPU-based Parallel Multistart Simulated Annealing algorithm for Quadratic Assignment Problem". In: *Engineering Science and Technology, an International Journal* 21.5 (2018), pp. 843–849. ISSN: 2215-0986. DOI: https://doi.org/10.1016/j.jestch.2018.08.002. URL: https://www.sciencedirect.com/science/article/pii/S2215098618308103.

[7] Samuel Xavier-de-Souza et al. "Coupled Simulated Annealing". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 40.2 (2010), pp. 320–335. DOI: 10.1109/TSMCB.2009.2020435.

6 Appendix

6.1 Schwefel Optimization Results

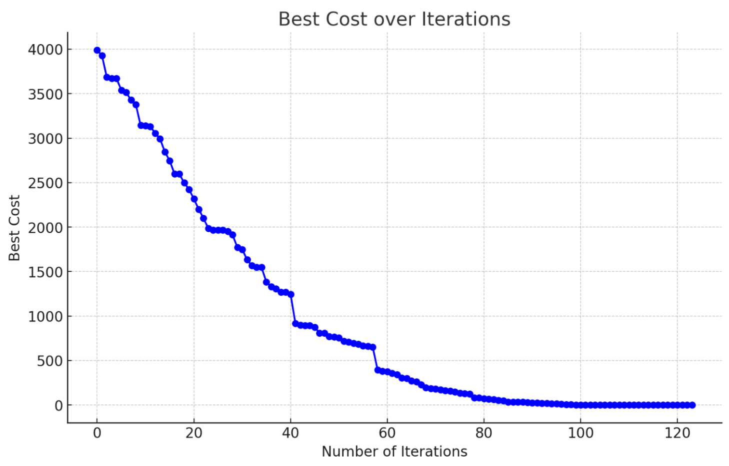

Figure 5: Optimization of Schwefel Function varying by Iterations for CSA

This objective function curve shows that optimization proceeds as expected and eventually converges upon a global minimum for CSA. We saw similar such curves for MSA (although convergence took a little longer).

6.2 Traffic Decongestion

We spent a significant amount of time formulating traffic optimization before pivoting to the Schwefel function. Here is our problem statement for this task, based off of [4].

Graph and Traffic Model

The traffic optimization model is built upon a graph structure, comprising points representing intersections and edges signifying roads. Each `Point` is defined by its coordinates (x, y), and each `Edge` connects two points, carrying a distance metric. The `Car` structure encapsulates individual vehicles, each with a source and destination point and a set of possible routes.

Initialization with Minimum Spanning Tree

The graph initialization employs Prim's algorithm to construct a Minimum Spanning Tree (MST), ensuring connectivity with minimal total edge weight. This tree forms the backbone of the traffic network, to which additional edges are randomly added, enhancing the network's complexity and realism.

Quadratic Unconstrained Binary Optimization (QUBO)

The traffic optimization problem is formulated as a QUBO problem, where the objective is to minimize a quadratic binary function representing the total travel distance and congestion levels [4]. The QUBO matrix is constructed based on the routes and their interrelations. CSA can be employed to solve this QUBO problem by exploring the solution space in a parallelized and cooperative manner, potentially leading to superior solutions compared to traditional methods.

Data Structures and Equations

The traffic model is underpinned by several data structures:

- `Point`: A structure representing graph nodes.
- `Edge`: A structure representing graph edges.
- `Car`: Vehicles navigating the graph, each with potential routes.
- `Route`: A sequence of edges denoting a path from source to destination.

Mathematical Formulation of QUBO

The Quadratic Unconstrained Binary Optimization (QUBO) problem for traffic optimization can be mathematically formulated as follows:

$$\min_{\mathbf{x}} \sum_{i=1}^{N} \sum_{j=1}^{N} Q_{ij} x_i x_j \tag{3}$$

Here, $\mathbf{x} = (x_1, x_2, \ldots, x_N)$ is a binary vector where each $x_i \in \{0, 1\}$ represents the selection status of a route for a car (1 if selected, 0 otherwise). Q_{ij} is an element of the QUBO matrix that encodes the cost or penalty associated with selecting routes i and j simultaneously. N is the total number of possible routes across all cars.

The objective is to minimize the total cost, which could include factors such as distance, time, and congestion.

Objective Function Calculation

The objective function value for a given configuration of routes is calculated as shown in the `calculateObjective` function in the provided C++ code. This function takes the QUBO matrix `Q` and the configuration vector, and computes the sum of the product of corresponding elements:

$$f(\mathbf{x}) = \sum_{i=1}^{N} \sum_{j=1}^{N} Q_{ij} x_i x_j \tag{4}$$

Simulated Annealing for QUBO Solution

The Simulated Annealing method is used to solve the QUBO problem, as implemented in `monteCarloQUBOSolver` in the C++ code. This method iteratively samples different configurations and selects the one with the lowest objective value:

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x}) \tag{5}$$

Each iteration involves randomly flipping a section of the binary vector (representing a particular set of routes for all cars), calculating the objective value, and keeping track of the best solution found so far.

Integration with Coupled Simulated Annealing

Coupled Simulated Annealing can be integrated into this framework to enhance the exploration of the solution space. In CSA, multiple SA processes are run in parallel, with periodic information exchange to guide the search towards more promising regions of the solution space. This approach can be particularly beneficial in solving the QUBO problem, where the landscape is complex and multi-modal. Although our current implementation does not fully integrate Coupled Simulated Annealing, the structure of the traffic model and graph is conducive to its application. CSA, with its ability to escape local optima through probabilistic hill-climbing and cooperative information exchange among annealing processes, can optimize the routing problem efficiently. The algorithm would iteratively adjust routes to minimize overall travel distance and congestion, adhering to the constraints and dynamics of the traffic network. All the data structures and spanning tree algorithms were implemented by scratch from our team, but due to time constraints, we were unable to fully implement the CSA algorithm here.