Simulated Annealing Individual Project
Bemacs 25

Ludovico Amedeo Panariello - 3192212

6 December 2023

Abstract

This paper presents the systematic steps undertaken to develop a computational code aimed at solving a specific optimization problem. The document outlines the process of code development, starting from problem formulation and data generation, and leading to the implementation of a Simulated Annealing algorithm discussing its limitations and possible improvements. The code is designed to find optimal solutions through iterative optimization, and a key focus of this research is the adaptation of the annealing process to handle situations with high differences in solution quality. Through this code development journey, I offered insights into problem-solving techniques, algorithm design, and strategies for dynamically adjusting optimization parameters based on feedback from the problem space. This research seeks to provide a comprehensive understanding of the code's construction and its ability to adapt to complex optimization challenges.
Please read first the '/README' file inside the folder and enjoy this shared coding journey!

Contents

1 Introduction

The problem at hand involves optimizing a target function represented as a discrete grid in a two-dimensional space. The goal is to find the optimal solution through a simulated annealing algorithm. The problem space is defined on a grid with dimensions [n x n], where each entry in the grid corresponds to the value of the target function for specific input values x and y. The function is defined on the grid $\{0, 1, 2, ..., n - 1\} \times \{0, 1, 2, ..., n - 1\}$, and the objective is to optimize this function using simulated annealing. The size of the problem, n, is a crucial factor in understanding the algorithm's performance and behavior. The algorithm's performance is also contingent on the size of the annealing steps as well as MCMC iterations employed during its execution

To achieve optimization through simulated annealing, a move proposal mechanism is implemented. This proposal selects a random move from the current point to a proposed point using a specific random choice technique:

$$x_{prop}|x^t = i = random.choice([i-1]_n, [i+1]_n),\ y_{prop}|y^t = j = random.choice([j-1]_n, [j+1]_n).$$

The move proposal allows for adjustments at the borders of the grid domain, ensuring that the algorithm can handle boundary cases. The acceptance or rejection of a move is determined by comparing the values of the function at the current point and the proposed point. The probability of accepting a move is calculated based on the difference in function values, and this mechanism is crucial for the simulated annealing process.

$$p((x^t, y^t) \to (x_{prop}, y_{prop})) = min(1, exp^{-(C[x_{prop}, y_{prop}] - C[x^t, y^t])}).$$

2 Task to Complete

2.1 Class Implementation

In this section, we present the formal description and implementation of the `IndividualProject` class, an integral component of our optimization framework. This class encapsulates the operational details of a simulated annealing algorithm designed for the resolution of discrete optimization problems within a grid-based problem space.

2.1.1 Class Definition: `IndividualProject`

The `IndividualProject` class is defined to represent an instantiation of an individual optimization endeavor. It embodies a comprehensive suite of components and mechanisms crucial to the successful execution of the simulated annealing process. The class is endowed with the following attributes and methods:

Initialization

The class constructor `__init__` governs the initialization process, accepting the following parameters:

- `n (int)`: Signifying the dimensions of the grid that models the problem space.
- `seed (int)`: Prescribing the seed used for the generation of grid data.

Additionally, the class maintains the ensuing pivotal attributes:

- `data`: A two-dimensional matrix storing the optimization landscape, denoting the grid's configuration.
- `position`: Signifying the current coordinates within the grid, subject to an initial random selection.
- `visited_frequency`: A matrix meticulously recording the frequency of visits to grid locations, instrumental for heatmap visualization.
- `path`: An array, purposefully engineered to chronicle the path traversed by the optimization algorithm, facilitating subsequent analytical endeavors.
- `potential_moves`: A matrix defining four admissible cardinal moves, thus enabling a degree of randomness within the simulated annealing process.
- `total_distance_traveled`, `last_position`, and `last_cost`: These variables serve as critical metrices, pivotal for the establishment of halting criteria and the evaluation of optimization progress.
- `colorbar_initialized`: A flag thoughtfully employed to manage the initialization of colorbars, a crucial facet of data visualization.
- `true_min_pos` and `true_min`: Repositories of intelligence concerning the global minimum's position and value within the problem space.

Move Proposals

The `propose_moves` method is the locus of stochasticity within the simulated annealing process. This method, guided by precomputed probabilities, selectively elects the subsequent move from the set of potential moves. The stochastic nature of this selection engenders exploration, thereby precluding entrapment within local optima.

Acceptance of Moves

The `accept_move` method assumes the responsibility of updating the algorithm's state upon the acceptance of a proposed move. This update encompasses the modification of the current position, the aggregation of the total distance traveled, and the tabulation of visits to grid points. These updates are of paramount significance, underpinning the tracking of the exploration pattern and the appraisal of the simulated annealing's efficacy.

Halting Criteria

Two distinct methods have been endowed within the class to ascertain halting criteria:

1. `has_traversed_enough`: This method scrutinizes whether the algorithm has traversed a sufficiently extensive distance, thereby constituting a termination criterion predicated upon the cumulative distance traveled. Its functionality assumes centrality in the endeavor to curtail the algorithm's execution subsequent to the accomplishment of a prescribed level of exploration.

2. `is_stagnant`: The `is_stagnant` method discerns the algorithm's stagnation status by scrutinizing infinitesimal changes in both position and cost. This scrutiny finds its import in the judicious cessation of the algorithm, thwarting unproductive iterations.

Visualization

The `display` method has been thoughtfully devised to expedite the visualization of the optimization landscape and the trajectory charted by the algorithm. Its offerings encompass the following visual constituents:

- The presentation of the optimization landscape through the utilization of a colormap.
- The demarcation of the final position attained and the identification of the global minimum within the landscape.
- The heatmap representation denoting the frequency of visits to grid points.
- The management of colorbar instantiation to enhance visualization fidelity.

2.2 Modulo Operation

The modulo operation used to update the proposed move helps manage grid boundaries when suggesting a movement. It essentially allows random movements to seamlessly wrap from one edge of the grid to the opposite edge, mimicking a toroidal (periodic) boundary condition. This technique is valuable in specific simulations as it ensures that the random walk can fully explore the entire grid without any edge-related problems. This approach offers both mathematical and computational benefits. If we choose not to allow it, then we would need to carefully manage boundaries through explicit checks with few changes inside the code:

- **Mathematical Implications**
  The use of modulo operations in Markov Chain Monte Carlo (MCMC) algorithms has significant mathematical implications for ensuring symmetric transition probabilities, which are essential for maintaining detailed balance and thus the convergence of the Markov chain to its equilibrium distribution. In MCMC, the proposal mechanism is central to suggesting new states, and symmetry here implies that the probability of transitioning from an old state to a new state should be the same as the probability of transitioning back. This symmetry is a cornerstone for

maintaining the detailed balance condition.

Incorporating modulo operations into the algorithm is particularly beneficial when handling transitions near the borders of a grid. Without modulo operations, moves near the borders are constrained, limiting the options available for transitions. For instance, in a grid-based system where each position typically has four neighbors, a position near the border without modulo wrapping would have fewer than four options for movement. This creates an asymmetry in transition probabilities, as border positions would inherently have different transition probabilities compared to inner positions.

Conversely, by utilizing modulo operations, this constraint is removed. Positions near the borders can 'jump' to the opposite side of the grid, maintaining the uniformity of having four potential moves from any position. This ensures that the transition probabilities remain symmetric, irrespective of the position on the grid. Each potential move, therefore, has an equal probability of being selected (e.g., 0.25 in a grid with four possible moves).

If modulo operations are avoided, an asymmetry in transition probabilities arises, as some moves are effectively 'blocked' at the borders, leading to invalid or disallowed states. This asymmetry can disrupt the detailed balance condition, potentially preventing the Markov Chain from converging to the desired equilibrium distribution.

To address this issue in scenarios where modulo operations are not used, one might employ the Metropolis-Hastings algorithm. This algorithm adjusts for the asymmetry in the proposal distribution, allowing for the acceptance of non-symmetric proposals through a specific acceptance rule. This rule compensates for the differing probabilities of forward and backward moves, thereby enabling the Markov Chain to satisfy the detailed balance condition and converge to the equilibrium distribution even in the absence of symmetric proposals through the following formula:

$$\min\left(1, \frac{C_{ij}\rho_i}{C_{ji}\rho_j}\right)$$

- **Code Implementation Implications**
  In this advanced examination of the simulated annealing algorithm, we consider the integral role of border jumps in exploring the state space, denoted as S. This space is formally defined as a set of discrete states, $s \in S$, and our objective is to identify an optimal state s^* that minimizes a cost function $f : S \to R$. The characteristics of S, such as its finiteness and dimensionality, along with the properties of f, like its non-linearity or potential multi-modality, are central to understanding the algorithm's efficacy.

  For enhanced exploration, we introduce a border jump mechanism formulated as $s' = (s + \delta) \mod n$. Here, s' is the proposed new state, s the current state, $\delta \in \Delta$ a predefined step in the state space, and n the cardinality or dimensionality of S. The modulo operation here is pivotal, enforcing a toroidal traversal of the state space, thus avoiding premature convergence to local minima and promoting a global search. The set Δ of possible moves is chosen to ensure adequate coverage of S.

  The practical implementation of this concept is shown in the following code, where the propose_moves function algorithmically determines the next state based on a uniform probability distribution over Δ:

```
def propose_moves(self):
    # Proposes the next move based on precomputed move probabilities.
    move_idx = np.random.choice(4, p=self.move_probabilities)
    self.move_idx = move_idx
    return (self.position + self.potential_moves[move_idx]) % self.n
```

  In contrast, eliminating the modulo operation introduces fixed boundaries to the state space. This approach changes the algorithm's interaction with the state space edges, potentially restricting exploration to non-boundary states. The second code snippet, from 'Class_no_modulo.py', depicts this scenario. The modified propose_moves function now incorporates boundary conditions,

adjusting the move probabilities based on the current state's location relative to the grid edges. Mathematically, this is represented as a conditional probability distribution for selecting move indices. If the current state is at a corner, a deterministic move ($p = 1$) is made diagonally. If on a border, a binary distribution ($p = \frac{1}{2}$ for each diagonal move) is used. These probabilities are mathematically justified to ensure a uniform exploration of the boundary-adjacent states, compensating for the absence of the modulo operation and its induced toroidal state space traversal.

```
    def propose_moves(self):
        """
        Proposes the next move based on position and boundary conditions.
        Adjusts move probabilities when on borders.
        """
        x, y = self.position

        # Define border and corner conditions
        is_on_border = x in [0, self.n - 1] or y in [0, self.n - 1]
        is_on_corner = x in [0, self.n - 1] and y in [0, self.n - 1]

        if is_on_corner:
            # If on a corner, reverse direction
            move_idx = (self.move_idx + 2) % 4
        elif is_on_border:
            # If on a border, choose to go diagonally left or right with
            # equal probability
            left_or_right = np.random.choice([1, 3])
            move_idx = (self.move_idx + left_or_right) % 4
        else:
            # Elsewhere, select any diagonal move
            move_idx = np.random.choice(4)

        # Calculate new position
        new_position = np.array(self.position) + self.potential_moves[move_idx]

        # Boundary checks
        if not (0 <= new_position[0] < self.n and 0 <= new_position[1] < self.n):
            # If proposed move goes out of bounds, choose a different move
            return self.propose_alternative_move(move_idx)

        return new_position
```

Post-move, the algorithm includes a stringent boundary check to validate the new state's location within the confines of S. This procedural check is crucial for maintaining the mathematical integrity of the state space and ensuring algorithmic validity. Such checks, coupled with the strategic adaptation of move probabilities, balance the exploration and exploitation trade-off inherent in the simulated annealing algorithm. This balance is vital for the algorithm's ability to navigate complex state spaces and to converge towards a global minimum of the cost function f. I implemented the check in this way:

```
        def propose_alternative_move(self, invalid_move_idx):
        """
        Proposes an alternative move if the initial move goes out of bounds.
        """
        valid_moves = [i for i in range(4) if i != invalid_move_idx]
        for move_idx in valid_moves:
            new_position = np.array(self.position) + self.potential_moves[move_idx]
            if 0 <= new_position[0] < self.n and 0 <= new_position[1] < self.n:
                return new_position

        raise ValueError("No valid alternative moves available from current position.")
```

2.3 Algorithm Performance as Temperature Increase

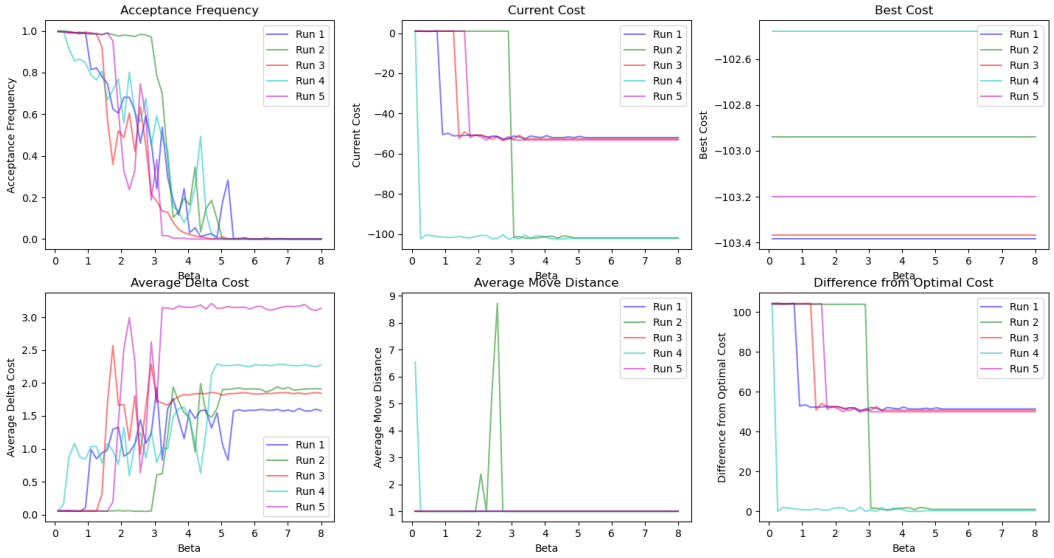


Figure 1: Results from different run on same parameters

I ran 5 times the algorithm on parameters:

- n = 200
- beta0 = 0.1
- beta1 = 8
- mcmc_steps = 1000
- anneal_steps = 50
- stopping_criteria = position 100 and cost 10

The observed data from multiple iterations of the simulated annealing algorithm reveal distinct convergence behaviors. Specifically, in 40% of the trials, the algorithm successfully approximates the global minima, while in the remaining 60%, it prematurely converges to local minima. This bifurcation in outcomes underscores the stochastic nature of the algorithm's initial conditions, which can lead either to the global minimum or to sub-optimal local minima within the defined landscape.

The acceptance frequency graph elucidates the adequacy of the beta values chosen for the annealing schedule. Notably, the acceptance rate diminishes to zero across all runs when beta approaches 5, signifying a state of equilibrium where the algorithm no longer accepts inferior cost positions in its immediate vicinity, given the predetermined move set. This plateau suggests that the algorithm has likely settled into a minimum, whether global or local.

Moreover, by a beta average of 1.5, the algorithm consistently reaches a zone proximal to the minima, dedicating the remainder of the annealing process to refining its position within this low-cost basin. This is evidenced by the negligible cost fluctuations observed beyond this beta threshold, implying that higher betas contribute minimally to escaping these basins once entered.

It is pertinent to note that the optimal solution is not attained in approximately half of the algorithmic runs. This is an inherent limitation that can be attributed to the algorithm's sensitivity to its initial state, coupled with the computational impracticality of executing an exhaustive number of MCMC steps to guarantee convergence to the global minimum.

In light of the landscape's characteristics—wherein two specular low-cost basins are identified amidst a terrain of marginal cost variability—the algorithm employed is very much influenced by the initial position, which is subsequently mitigated by multiple algorithmic executions. This approach strikes a balance between computational efficiency and solution quality, with the early stopping criteria facilitating this by halting the algorithm upon encountering prolonged periods of minimal cost variation and after traversing a sufficient path length. This decision reflects a pragmatic trade-off, prioritizing expedience and resource conservation over exhaustive search completeness.

2.4 Analyzing Algorithm Performance as Problem Dimension Increases

To tackle the inquiry at hand, I devised an algorithm named '/study_performance,' which resides within the '/running files' directory. Within this codebase, I conducted a comprehensive series of simulations, (approximately 1000) These simulations encompassed a wide array of parameters, capturing a broad spectrum of scenarios and configurations.

Following these simulations, I systematically collated the resulting data, organizing it into a structured CSV file. This file became a valuable resource for not only visualizing the variations in computational time but also for training a RandomForest regressor. The regressor was employed to delve deep into the intricate relationship between hyperparameters and the ultimate cost incurred by the algorithm.

By taking this meticulous approach, I aimed to gain a deeper understanding of how algorithm performance evolves as problem n increases, crucial for making informed decisions and optimizations in addressing higher-dimensional problem spaces.

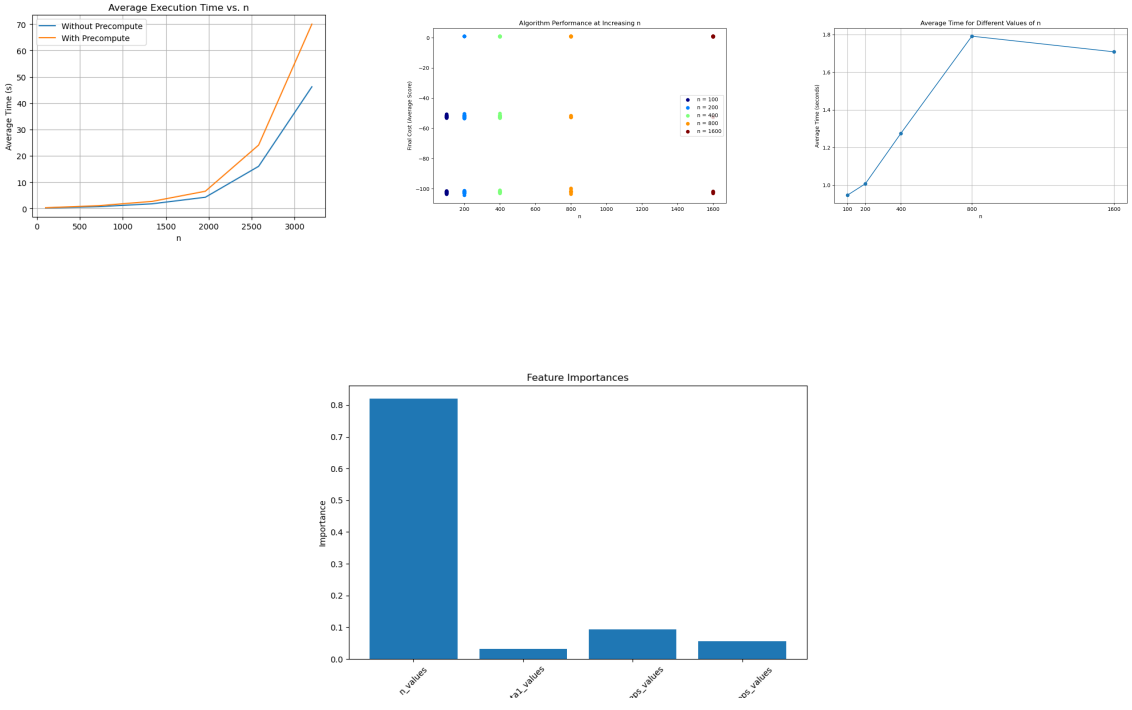


Figure 2: Enter Caption

Observations from the simulation data indicate a notable scaling in the algorithm's execution time beginning at approximately $n = 2000$. This trend, discernible from the 'simulated_annealing_performance.csv' file, suggests that the algorithm's time complexity continues to increase with larger values of n. However, the rate of increase appears to plateau after $n = 800$, which may be attributed to the

implementation of early stopping criteria. These criteria seem to effectively reduce computational overhead beyond this point, resulting in a diminished incremental time cost.

At smaller values of n, the algorithm demonstrates a reduced likelihood of becoming ensnared in a performance plateau. This suggests that it is easier for the algorithm to avoid local minima and continue searching for the global minimum in lower-dimensional spaces.

Further analysis of parameter influences reveals that n is the predominant factor in determining the efficiency of the algorithm. The parameter n dictates the extent of the search space that the 'random mover' must navigate to locate the global minimum. As such, it is a critical determinant of the algorithm's performance, underscoring its importance in the overall search strategy.

2.5 Acceptance Rate Probabilities

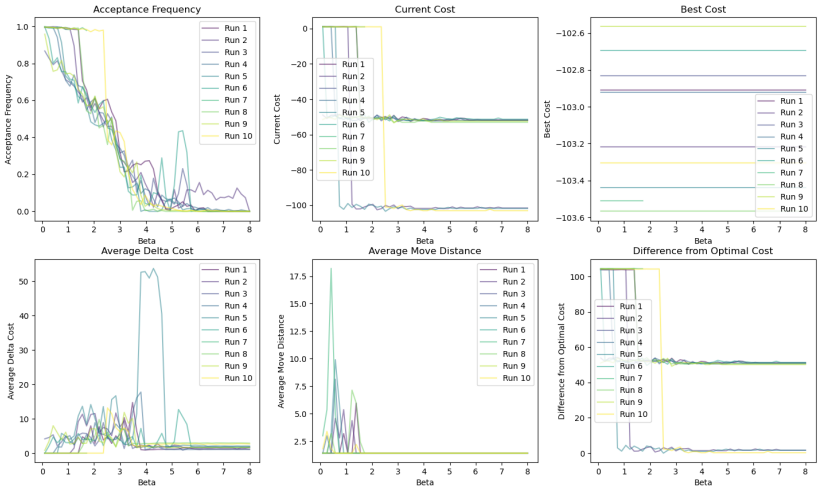


Figure 3: Results from different run on same parameters

I ran 10 times the algorithm on parameters:

- n = 500
- beta0 = 0.1
- beta1 = 8
- mcmc_steps = 1000
- anneal_steps = 80
- stopping_criteria = position 100 and cost 10

The acceptance probability in simulated annealing algorithms is pivotal in dictating the stochastic search dynamics. This probability is governed by the Metropolis criterion:

$$P(\text{accept}) = \min\left(1, \exp\left(-\frac{\Delta C}{kT}\right)\right) \tag{1}$$

where ΔC represents the change in cost resulting from a proposed move, T denotes the system's temperature, and k is the Boltzmann constant, which is often normalized to 1 in computational simulations. The temperature parameter is inversely related to β (where $\beta = \frac{1}{kT}$), with higher values of β corresponding to lower system temperatures.

The provided Acceptance Frequency plot exhibits a monotonically decreasing relationship with β. At the algorithm's initiation ($\beta \approx 0$, indicative of high temperature), the acceptance probabilities

approach unity, reflecting a globally exploratory search that permits transitions to higher-cost states, thus enabling the escape from local optima.

As β increases, mirroring the system's cooling, there is a marked decrement in acceptance probability. This decline is congruent with the annealing schedule's intent to incrementally contract the search radius as the global optimum is neared. The mathematical encapsulation of this phenomenon resides in the exponential term of the Metropolis criterion; an elevated β increases the acceptance probability's sensitivity to positive ΔC. Consequently, as the system's temperature descends, only moves yielding a decrease in cost (negative ΔC) have a significant probability of acceptance, thus steering the algorithm towards convergence.

Variations in acceptance probability across iterations, particularly pronounced at lower β values, may signal an energy landscape's heterogeneity. Intermittent spikes are indicative of transient interactions with regions characterized by steep gradients. As β extends to more substantial values, the alignment of acceptance probabilities across iterations denotes a homogenization of search behaviors, suggestive of the algorithm's settlement near an optimum.

Furthermore, the trajectory of acceptance probability as a function of increasing β can provide inference on the landscape's ruggedness. A smooth transition from high to low acceptance probabilities would imply a landscape with gradual slopes, whereas a steep transition may indicate a landscape with sharp gradients.

In conclusion, the observed behavior of acceptance probabilities elucidates the energy landscape's topology and the dynamics of the simulated annealing search. It underscores the algorithm's capability to navigate the intricate balance between exploration and exploitation, as governed by the annealing schedule, to efficaciously optimize the stipulated cost function.

2.6 Optimization Landscape

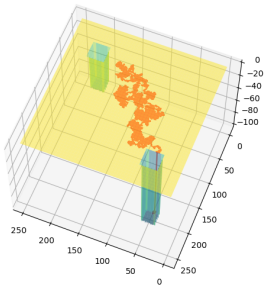


Figure 4: Optimization Landscape - 3D

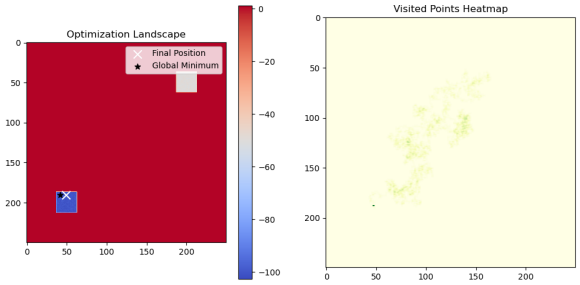


Figure 5: Optimization Landscape - Heatmap

The optimization landscape, as observed, consists of a plateau with marginal variations in cost, into which two distinct depressions are embedded. These depressions, or 'holes', descend to depths of approximately -50 and -100, with the latter housing the global minimum. The landscape's configuration presents a significant challenge: once the simulated annealing algorithm descends into the sub-optimal hole (the one reaching -50), it becomes problematic for the algorithm to ascend back to the plateau and continue searching for the global minimum.

Simulated annealing is designed to address such challenges by allowing a certain degree of uphill movement, particularly at higher temperatures (lower beta values), which facilitates escape from local minima. However, as the temperature lowers and beta increases, the algorithm's acceptance probability for uphill moves diminishes, and it becomes more likely to converge within a hole. To mitigate the risk of being trapped in the suboptimal hole, the algorithm is executed multiple times. This repetition helps to average out the initial condition biases, providing a more robust search for the global minimum.

The trajectory plotted in red within the landscape visualization represents the path of the random mover throughout the simulated annealing process. This path provides insights into the algorithm's behavior over time. Initially, when the acceptance probability for all moves is high, the path is likely to cover a greater area of the landscape, indicating an extensive search. As the temperature decreases, the path converges, indicating the algorithm is settling into a basin.

The ability to revisit the plateau and escape the suboptimal hole is crucial for the algorithm to find the global minimum effectively. The multiple runs of the algorithm, each starting from a different random position, help ensure that it does not consistently converge to the local minimum due to a poor initial starting point. Instead, it has a greater chance of sampling the entire landscape, including the deeper hole where the global minimum lies. This approach is a testament to the adaptive nature of simulated annealing, where the exploration-exploitation trade-off is finely balanced by the temperature parameter, allowing the algorithm to overcome the complex topology of the optimization landscape.

2.7 Final Considerations on Simulated Annealing

Simulated Annealing (SimAnn) is particularly suited for complex optimization landscapes with multiple minima, like the one under consideration. The primary advantage of SimAnn over greedy algorithms is its intrinsic ability to escape local minima, a vital feature in landscapes replete with suboptimal solutions.

2.7.1 Comparison with Greedy Algorithms

Greedy algorithms, which iteratively make locally optimal choices, are likely to underperform in such scenarios. These algorithms lack a mechanism to retreat from local minima, making them susceptible to convergence at suboptimal points. In landscapes characterized by plateaus (since the entire space is not regular) or deceptive minima (e.g., a suboptimal hole at -50), a greedy approach may prematurely converge to these local minima without exploring the global minimum (e.g., at around -100).

Mechanism of Simulated Annealing In contrast, SimAnn incorporates a probabilistic element that allows occasional uphill moves, particularly at higher temperatures (analogous to lower β values). This probabilistic approach provides a mechanism for the algorithm to escape suboptimal regions. As the temperature decreases (or as β increases), SimAnn gradually becomes more selective, progressively reducing the acceptance of deleterious moves. This controlled cooling process aids in fine-tuning the algorithm's convergence towards the global minimum. The trajectory of SimAnn, (e.g., marked in red), illustrates its initial widespread exploration, followed by a focused search in promising regions as the temperature decreases.

Limitations and Computational Considerations Despite its effectiveness, SimAnn is not without limitations. It can be computationally demanding, particularly for large state spaces (large n values). Its performance heavily depends on the chosen parameters, such as the cooling schedule, initial temperature, and stopping criteria. Furthermore, the necessity of multiple runs to ensure robust search results adds to the computational burden.

2.7.2 Enhanced Stopping Criteria

To augment the efficiency of the Simulated Annealing algorithm, I introduced two mathematical stopping criteria already from the standard algorithm:

Traversed Distance Criterion: The first criterion, `has_traversed_enough`, employs a distance threshold to decide when the algorithm has sufficiently explored the state space. This is mathematically represented as a condition where the total distance traveled by the algorithm exceeds a predefined threshold:

```
def has_traversed_enough(self, distance_threshold):
        """
        Determines if sufficient distance has been traversed, serving as a stopping crite

        Based on the total distance traveled.
        Helps terminate the algorithm after adequate exploration.
        """
        return self.total_distance_traveled >= distance_threshold
```

Stagnancy Assessment: The second criterion, `is_stagnant`, uses mathematical thresholds for position and cost changes to determine if the algorithm has become stagnant. It assesses whether the Euclidean distance in position and the absolute difference in cost between consecutive iterations fall below specified thresholds:

```
def is_stagnant(self, position_threshold, cost_threshold):
        """
        Assesses whether the algorithm has become stagnant.

        Checks for minimal changes in position and cost.
        Important for stopping the algorithm to avoid unproductive iterations.
        """
        position_change = np.linalg.norm(self.position - self.last_position)
        cost_change = abs(self.data[tuple(self.position)] - self.last_cost)
        self.last_position = np.copy(self.position)
        self.last_cost = self.data[tuple(self.position)]
        return position_change < position_threshold and cost_change < cost_threshold
```

These criteria mathematically formalize the stopping conditions, ensuring that the algorithm terminates when further exploration is unlikely to be productive.

2.7.3 Enhanced Move Selection in Simulated Annealing

To address computational challenges in the simulated annealing algorithm, an enhancement focusing on precomputation and caching of move costs has been proposed. This involves adapting the probability distribution of moves to favor those with lower delta costs. The cost difference, $\Delta f(s, s') = f(s') - f(s)$, defines the change in cost for a move from the current state s to a potential state s'. The probability of making such a move is then expressed as:

$$p(s \to s') = \frac{e^{-\Delta f(s,s')/T}}{\sum_{\forall s''} e^{-\Delta f(s,s'')/T}}, \tag{2}$$

where each term in the formula is defined as follows:

- s represents the current state of the system.
- s' denotes a potential new state that the system can transition to from the current state.
- $\Delta f(s, s')$ is the change in the cost function, calculated as $f(s') - f(s)$, when moving from state s to state s'.
- T is the temperature parameter in simulated annealing, a control parameter that affects the probability of accepting states with higher costs. It is typically set high initially and decreased gradually.
- The exponential term $e^{-\Delta f(s,s')/T}$ determines the likelihood of accepting the transition from s to s'. This likelihood is higher for transitions that lead to a lower cost.

– The denominator $\sum_{\forall s''} e^{-\Delta f(s,s'')/T}$ is a normalization factor summing over all possible new states s'' that can be reached from the current state s. This ensures that the probabilities of transitioning to all possible new states sum to 1, forming a valid probability distribution.

In practical terms, this formula calculates the probability of accepting a move to a new state in the simulated annealing algorithm, balancing the exploration of new states with the exploitation of known good states based on the cost function and the temperature parameter.

Precomputation and Caching of Move Costs This enhancement introduces a systematic approach for precomputing move costs, thereby optimizing the move selection process. The method is encapsulated in the `precompute_move_costs` function, which calculates the delta costs $\Delta f(s, s') = f(s') - f(s)$ for each possible move from the current state s. These costs are then used to adapt the probability distribution for move selection, favoring states with lower costs. The algorithmic implementation can be found in folder '/precomputation' with 'SimAnn_precompute.py' and 'Class_precompute.py'. I coded it as:

```
def precompute_move_costs(self):
        """
        Precomputes and caches the move costs from the current position.
        Optimizes move selection by storing probabilities for all possible moves.
        Positions with lower costs are assigned higher probabilities, influencing the mov
        Utilizes caching to avoid redundant computation for revisited positions.
        """
        if tuple(self.position) in self.cost_cache:
            move_probabilities = self.move_prob_cache[tuple(self.position)]
            delta_costs= self.cost_cache[tuple(self.position)]
        else:
            delta_costs = np.zeros(len(self.potential_moves))
            for i, move in enumerate(self.potential_moves):
                moved_position = tuple((self.position + move) % self.n)
                new_cost = self.data[moved_position]
                delta_costs[i] = new_cost - self.data[tuple(self.position)]
            move_probabilities = np.exp(-delta_costs)
            move_probabilities /= np.sum(move_probabilities)
            self.move_prob_cache[tuple(self.position)] = move_probabilities
            self.cost_cache[tuple(self.position)] = delta_costs

        self.move_probabilities = move_probabilities
```

This function reduces computational overhead by caching move costs and their corresponding probabilities, thus avoiding redundant recalculations.

Propose move The class method `propose_moves` is designed to propose the next move based on these precomputed move probabilities, introducing stochasticity into the simulated annealing process and enabling exploration while avoiding local minima:

```
    def propose_moves(self):
        """
        Proposes the next move based on precomputed move probabilities.

        Introduces stochasticity in the simulated annealing process.
        Randomly selects the next move, allowing for exploration and avoiding local minim
        """
        move_idx = np.random.choice(4, p=self.move_probabilities)
        self.move_idx = move_idx
        return (self.position + self.potential_moves[move_idx]) % self.n
```

2.7.4 Computational Complexity in Enhanced Simulated Annealing

The proposed enhancement to SimAnn, while mathematically robust in adapting the move probability distribution, introduces significant computational considerations. These considerations arise primarily from the exponential increase in computations (no longer $O(1)$) with respect to the dimensionality and size of the state space.

Computational Complexity with Dimensionality Increase: In a d-dimensional space, where each dimension contributes two possible moves, the total number of moves becomes 2^d. Consequently, the complexity of computing move costs for a single state is $O(2^d)$, exhibiting exponential growth with respect to the dimensionality d. This exponential increase poses a considerable challenge in high-dimensional optimization problems, where the computational resources required for precomputing move costs escalate rapidly with increasing d.

Complexity Scaling with State Space Size: Additionally, as the size of the grid (denoted by n) increases, the algorithm must compute and evaluate a greater number of moves. Considering a fixed number of computations per move (e.g., 4 computations per move), the complexity for traversing the entire space, without considering the holes, scales linearly with the grid size, approximately $O(4n)$. However, this linear scaling becomes more complex in a d-dimensional space, where the overall complexity for computing all moves across the state space is $O(2^d \cdot n)$. While linear in terms of n, the complexity remains exponentially dependent on d.

Storage Complexity: The enhancement also necessitates storing the arrays of precomputed probabilities for each state, adding to the computational burden. The space complexity for storing these probabilities is $O(n)$, linear with respect to the number of states in the grid. However, this does not account for the additional complexity introduced by the dimensionality of the state space which will change just the dimension of the array (i.e. dimension of the memory).

2.7.5 Empirical Study on Enhanced vs. Standard Simulated Annealing

In order to quantitatively assess the efficacy and computational intensity of the enhanced Simulated Annealing (SimAnn) algorithm compared to its standard counterpart, an empirical study was conducted. This study focused on measuring both the success rate in reaching the global minimum and the computational time taken by each algorithm.

Methodology The empirical analysis was based on averaging results from approximately 50 runs for each algorithm, ensuring diverse initial conditions by varying the initialization seeds (while maintaining the same landscape with seed 3192212). A key variable in this comparison was the β parameter (inverse temperature), which was adjusted to be $\frac{1}{3}$ higher for the standard algorithm, in line with the differing nature of their move proposal distributions.

Observations and Results The range of grid sizes considered spanned from $n = 100$ to $n = 3500$. For each size n, the time taken to find the minimum (truncated according to uniform stopping criteria applied to both algorithms) was recorded.

Figure 6:

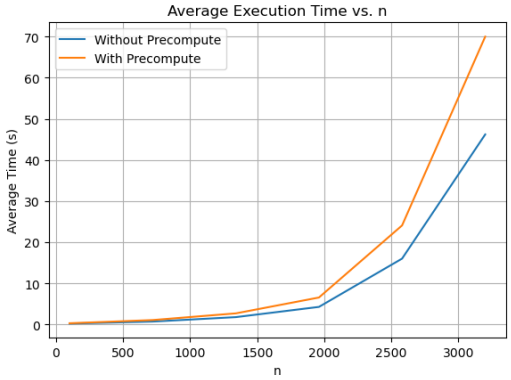
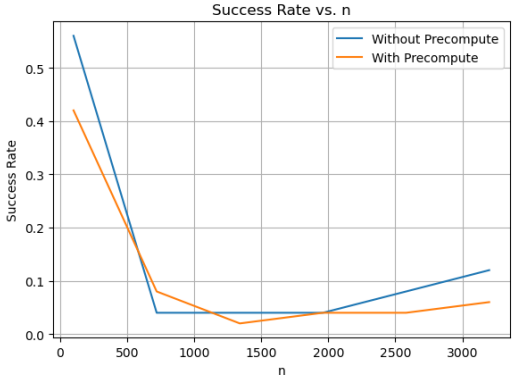


Figure 7:



Success Rate: Observations indicated that for larger grid sizes, the standard Simulated Annealing (SimAnn) algorithm exhibited a higher frequency of successfully locating the global minimum compared to the standard algorithm. However, in the specific range from $n = 600$ to $n = 1200$, this trend did not hold. This deviation could be attributed to a point in the problem's complexity where the generated grid presents optimal minima that are particularly challenging to reach. In such scenarios, the enhanced algorithm's higher propensity to accept lower cost moves may effectively facilitate reaching these difficult-to-access minima.

Computational Time: A notable increase in average computational time was observed for the enhanced algorithm, particularly as the grid size n increased. This trend suggests an exponential scaling of computational time with respect to n, indicating a trade-off between the precision of the enhanced algorithm (specifically in the range $n = 600$ - $n = 1200$ and its computational demands.

Concluding Insights and Introduction of a Mixed Algorithm Approach The empirical data corroborate the theoretical analysis that the enhanced SimAnn, while more effective in certain scenarios, incurs a significant computational overhead. This is primarily due to the exponential complexity introduced by the precomputation and caching of move costs, especially in higher-dimensional spaces and larger grids. There is a crucial need to strike a balance between precision and computational efficiency in the application of the enhanced Simulated Annealing (SimAnn) algorithm.

To address these computational challenges while retaining the benefits of the enhanced approach, I implemented a 'mixed' algorithm. This algorithm dynamically selects between the standard and the enhanced SimAnn methodologies based on the dimensionality of the problem. The decision criterion for this selection process is based on a threshold that balances computational efficiency with the precision gains of the enhanced method.

The implementation of this mixed approach is available in the '/mixed' directory, with the algorithm defined in 'SimAnn_mixed.py' and the associated class in 'Class_mixed.py'. The mixed algorithm aims to leverage the strengths of both the standard and the enhanced SimAnn methods, adapting to the problem's complexity to optimize performance.

2.8 Conclusion

Moreover, in conclusion, this project not only advances our comprehension of the Simulated Annealing (SimAnn) algorithm within grid-based optimization landscapes but also makes a valuable

contribution to the ongoing discourse on algorithmic efficiency and adaptability in the field of optimization. The introduction of a hybrid algorithm, seamlessly merging traditional and enhanced SimAnn strategies, represents a thoughtful and strategic response to computational constraints while remaining steadfast in the pursuit of optimal solutions within intricate optimization domains.

This research significantly augments our understanding of Simulated Annealing as a robust optimization tool and highlights its practical applicability in addressing real-world optimization challenges. Despite the computational demands associated with Simulated Annealing, its remarkable ability to navigate complex landscapes and discover near-optimal solutions positions it as a compelling choice for a diverse range of complex optimization tasks, underscoring its relevance and importance in academic and practical contexts alike.