# Simulated Annealing

## Simulated Annealing: General Idea

Imagine you're trying to **find the lowest point in a hilly landscape** at night.

- You can't see far ahead, so you take small steps downhill.
- If you only move downhill (like **greedy hill climbing**), you might get stuck in a small valley (local minimum) and never find the deepest valley (global minimum).

Now, here's where **simulated annealing** comes in:

Think of yourself as **a hiker with a backpack full of energy (heat)**.

- At the start, you're very energetic (high temperature).
  - You don't mind sometimes walking *uphill* (taking worse steps).
  - Why? Because maybe going uphill a bit lets you escape a small valley and reach a deeper one later.
- As time goes on, your energy slowly cools down (temperature decreases).
  - You become less and less willing to go uphill.
  - Eventually, when you're almost out of energy (temperature near zero), you only move downhill into the best valley you can find.

## Real-Life Analogy: Metal Cooling

The name comes from **metallurgy** (the science of making metals).

- When blacksmiths heat metal until it glows, the atoms are very energetic and moving around randomly.
- As the metal slowly cools (annealing), atoms settle into a stable, strong structure.
- If cooled too quickly, the structure is weak (like getting stuck in a local minimum).
- If cooled slowly, the structure becomes optimal and stable (global minimum).

## Core Components of Simulated Annealing

**Energy Function (Cost/Objective Function)**

The **Energy Function** (often called the **Cost Function** or **Objective Function**) represents the quality or fitness of a solution.

- **Definition:**
  - The energy function assigns a value (or "cost") to each possible solution in the search space.
  - Lower energy values represent better solutions, and higher energy values represent worse solutions.
- **Role in SA:**
  - SA uses the energy function to evaluate how good or bad a solution is. The algorithm attempts to minimize this function, just as a physical material seeks to minimize its energy state during annealing.
- **Example:**
  - In the Travelling Salesman Problem (TSP), the energy function could be the total distance traveled. The goal would be to minimize this distance.

## Temperature (T)

The **Temperature** (T) controls the algorithm’s willingness to accept worse solutions.

- **Initial High Temperature:**
  - At the start of the algorithm, the temperature is high, which means that the system can explore freely by accepting both better and worse solutions.
  - The probability of accepting a worse solution is determined by the equation:

$$P(\Delta E) = \exp\left(\frac{-\Delta E}{T}\right)$$

where ΔE is the difference between the current solution and the new solution.

  - If ΔE is positive (i.e., the new solution is worse), there is still a chance that the solution will be accepted, especially when T is high.
- **Decreasing Temperature:**
  - As the temperature decreases, the algorithm becomes less likely to accept worse solutions. Eventually, when the temperature is near zero, the system only accepts better solutions.
- **Intuition:**
  - At high temperatures, the algorithm explores more of the solution space (including suboptimal solutions).
  - At low temperatures, the algorithm focuses on refining the best solution found so far.

## Cooling Schedule

The **Cooling Schedule** determines how the temperature is decreased over time.

- **Cooling Function:**
  - The cooling schedule specifies the rate at which the temperature decreases as the algorithm progresses.
  - Common cooling schedules include:
    - **Linear Cooling:** $T = T_0 - \alpha k$, where $k$ is the iteration number.
    - **Exponential Cooling:** $T = T_0 \cdot \alpha^k$, where $\alpha < 1$ is a constant.
    - **Logarithmic Cooling:** $T = \frac{T_0}{\log(1+k)}$, where $k$ is the iteration number.
- **Choosing the Right Cooling Schedule:**
  - A **slow cooling schedule** provides more time for the algorithm to explore, increasing the chances of finding a global optimum but at the cost of longer runtime.
  - A **fast cooling schedule** may lead to quicker convergence, but the system might miss the global optimum and get stuck in a local minimum.

# The Simulated Annealing Algorithm

**1. Initialization Phase**

- **Initial Solution:**
  - The algorithm begins with a randomly selected initial solution from the solution space. This solution is often generated arbitrarily, but it should belong to the set of all possible solutions for the given optimization problem.
  - Example: In the Travelling Salesman Problem (TSP), an initial solution might be a random order of visiting cities.
- **Initial Temperature ($T_0$):**
  - The algorithm sets an initial temperature ($T_0$), typically high, to allow for broad exploration of the solution space. The high temperature enables the algorithm to potentially accept worse solutions early on.
  - The initial temperature is a key hyperparameter that influences the initial exploration phase.

**2. Generate a New Solution (Neighbor Selection)**

- **Neighboring Solution:**
  - From the current solution, a new neighboring solution is generated by making a small, random perturbation or modification to the current solution.
  - The neighboring solution should be similar to the current one, ensuring that the search progresses gradually rather than making large jumps in the solution space.
  - Example: In the TSP, a neighboring solution could be generated by swapping the order of two cities in the current route.

## 3. Evaluate the New Solution

- **Energy Function (Objective Function):**
  - The quality of both the current and new solutions is evaluated using the **energy function** (also known as the **cost** or **objective function**).
  - The energy function represents the quality of the solution—lower values represent better solutions.
- **Energy Difference (ΔE):**
  - Compute the energy difference between the current solution and the new (neighboring) solution:

$$\Delta E = E_{\text{new}} - E_{\text{current}}$$

- **Interpretation of ΔE:**
  - If $\Delta E < 0$, the new solution is better (lower cost) than the current solution.
  - If $\Delta E > 0$, the new solution is worse (higher cost) than the current solution.

## 4. Acceptance Criteria

- **Better Solutions (ΔE < 0):**
  - If the new solution has a lower cost (i.e., ΔE < 0), it is immediately accepted as the current solution.
  - This ensures that the algorithm always moves toward improving solutions when they are found.
- **Worse Solutions (ΔE > 0):**
  - If the new solution is worse than the current solution (i.e., ΔE > 0), it can still be accepted based on a probability:

$$P(\Delta E) = \exp\left(\frac{-\Delta E}{T}\right)$$

- **Explanation:**
  - The probability of accepting a worse solution depends on both the temperature (T) and the energy difference (ΔE).
  - At high temperatures, this probability is higher, allowing the algorithm to explore the solution space freely. At low temperatures, the probability of accepting worse solutions decreases, making the algorithm more selective.
- This mechanism allows SA to escape local optima by occasionally accepting worse solutions, especially in the early stages when the temperature is high.

## 5. Update Temperature

- **Cooling Schedule:**
  - After evaluating and accepting (or rejecting) the new solution, the temperature is reduced according to a predefined **cooling schedule**. The cooling schedule dictates how quickly or slowly the temperature decreases over time.

**Types of Cooling Schedules:**

1. **Linear Cooling:**

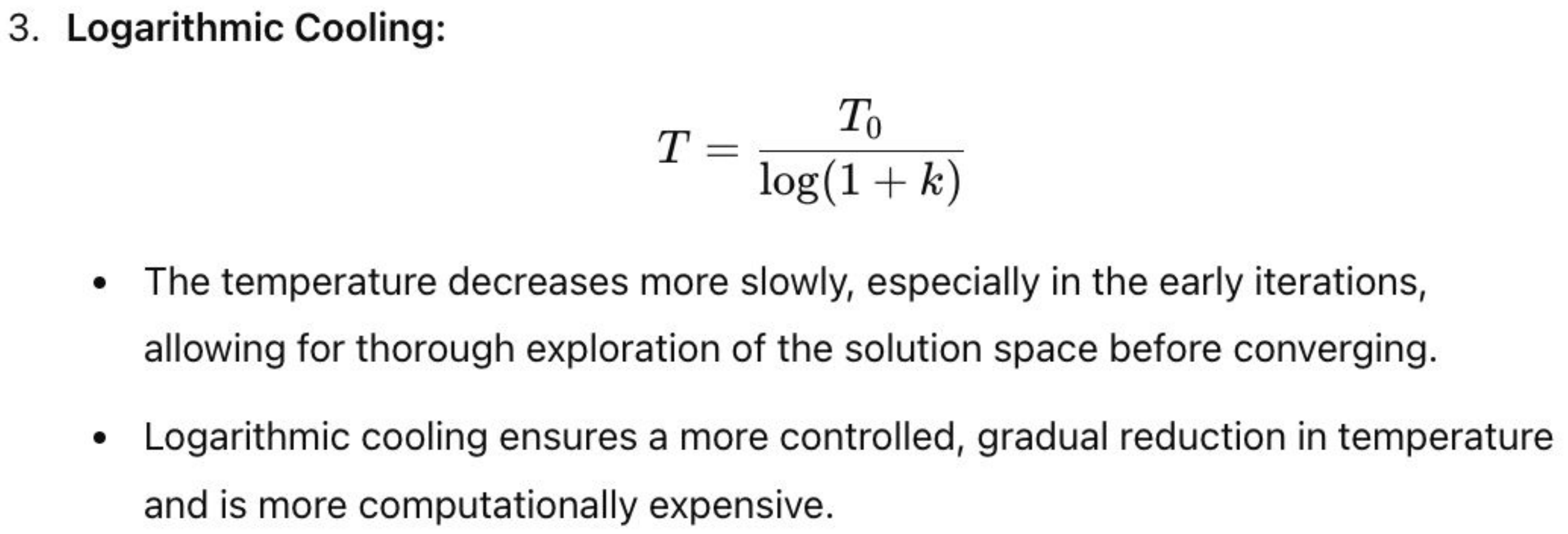
$$T = T_0 - \alpha \cdot k$$

   - $\alpha$ is a constant cooling rate.
   - The temperature decreases linearly with each iteration.
   - Suitable for problems where a steady decrease in temperature is desired.

2. **Exponential Cooling:**

$$T = T_0 \cdot \alpha^k$$

- $\alpha$ is a constant between 0 and 1.
- The temperature decreases exponentially with each iteration.
- This cooling schedule allows for faster cooling in the later stages of the algorithm.



3. **Logarithmic Cooling:**

$$T = \frac{T_0}{\log(1+k)}$$

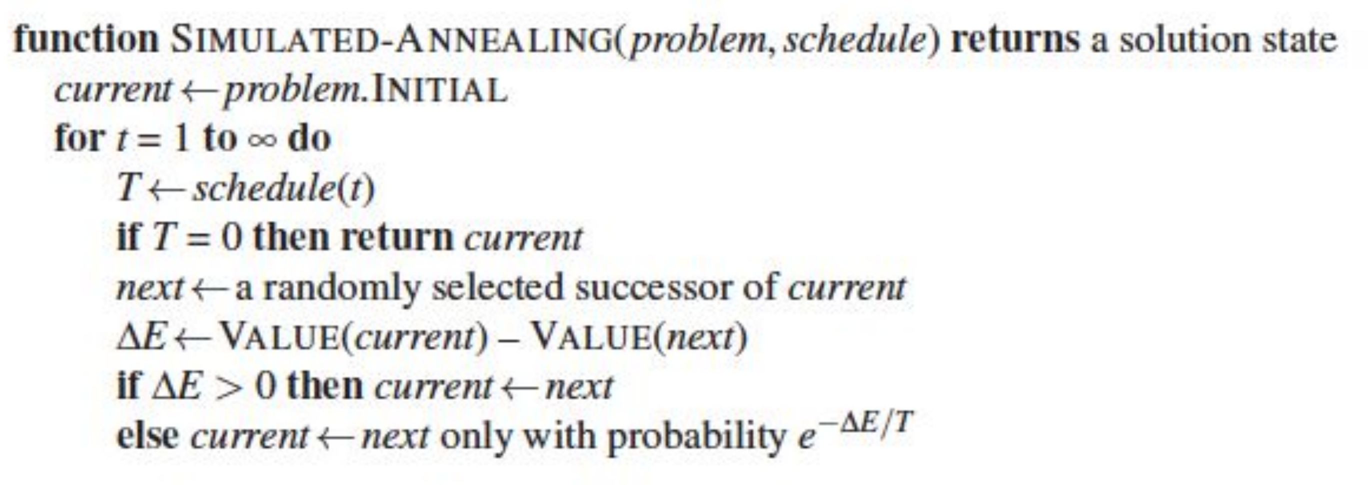
- The temperature decreases more slowly, especially in the early iterations, allowing for thorough exploration of the solution space before converging.
- Logarithmic cooling ensures a more controlled, gradual reduction in temperature and is more computationally expensive.

- **Selection of Cooling Schedule:**
  - The choice of cooling schedule depends on the nature of the problem. Faster cooling schedules (e.g., exponential) might be used when quick convergence is desired, while slower cooling schedules (e.g., logarithmic) might be preferred for more thorough exploration.

## 6. Termination Criteria

- **Freezing the System:**
  - The process continues until the system is "frozen," which typically means the temperature approaches zero (or becomes very low), effectively stopping the acceptance of worse solutions. At this point, the algorithm is highly selective and only accepts better solutions.
  - Another common termination criterion is to stop the process when no significant improvement is observed over a number of iterations.
- **Convergence:**
  - The algorithm is said to converge when no further improvements can be made, or the temperature has decreased to a point where the solution has stabilized.
  - The solution at this point is either the global optimum or a near-optimal solution.
- **Stopping Conditions:**
  - The algorithm can also terminate based on:
    - A maximum number of iterations.
    - A predefined time limit.
    - The difference between consecutive solutions becoming negligible (indicating convergence).

## The Simulated Annealing algorithm



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  current ← problem.INITIAL
  for $t = 1$ to $\infty$ do
      $T$ ← schedule($t$)
      if $T = 0$ then return current
      next ← a randomly selected successor of current
      $\Delta E$ ← VALUE(current) – VALUE(next)
      if $\Delta E > 0$ then current ← next
      else current ← next only with probability $e^{-\Delta E/T}$
```

## Applications of SA

- In addition to the well-known applications like the **Travelling Salesman Problem (TSP)**, **job scheduling**, **neural network training**, and **VLSI design**, Simulated Annealing (SA) has a wide variety of other applications in artificial intelligence (AI) and related fields. In addition to the well-known applications like the **Travelling Salesman Problem (TSP)**, **job scheduling**, **neural network training**, and **VLSI design**, Simulated Annealing (SA) has a wide variety of other applications in artificial intelligence (AI) and related fields. Below are additional applications where SA has been effectively used:
- **Feature Selection in Machine Learning:**.
- **Hyperparameter Tuning in Machine Learning**
- **Reinforcement Learning Policy Optimization**
- **AI for Game Playing (Game Theory)**
- **Clustering in Data Mining**
- **Path Planning in Robotics**
- **Genetic Algorithm Hybridization**
- **Computer Vision: Image Segmentation**
- **Protein Folding in Bioinformatics**
- **Autonomous Vehicle Navigation**
- **AI for Financial Portfolio Optimization**
- **Spatial Layout Design (Architectural Optimization)**
- **Circuit Routing in Electronic Design Automation (EDA)**
- **3D Object Recognition**

## Why Simulated Annealing Works in These Applications

**Flexibility in Handling Non-Convex and Complex Objective Functions:**

- Many real-world problems, especially those in AI, have non-convex objective functions with multiple local optima. SA is well-suited for such problems because it doesn't just move to better solutions but also explores worse solutions, helping to avoid local optima.

**Works Well in Large, Combinatorial Search Spaces:**

- In problems like TSP, job scheduling, and VLSI design, the solution space is vast and highly combinatorial. SA's ability to explore large portions of the search space by accepting suboptimal moves early in the process helps it find global or near-global optima efficiently.

# Limitations and Challenges of Simulated Annealing

1. **Slow Convergence**

2. **Sensitivity to Cooling Schedule**

3. **Not Ideal for Small-Scale Problems**

4. **Difficulty in Parameter Tuning**

## Potential Solutions to These Challenges

- **Hybrid Algorithms:**
    - Combine SA with other optimization techniques (e.g., **genetic algorithms** or **tabu search**) to take advantage of SA's exploration capabilities while using faster algorithms for local exploitation.
- **Adaptive Cooling Schedules:**
    - Implement **adaptive cooling schedules** that dynamically adjust the temperature based on the performance of the algorithm. For example, if improvements are happening rapidly, the schedule can slow down the cooling rate.
- **Parallelization:**
    - One way to speed up SA and improve exploration is to use **parallelization**. Multiple instances of the algorithm can explore different parts of the solution space simultaneously, helping to identify promising areas more quickly.

# N Queens: Simulated Annealing

**1. Generating Neighboring States:** A neighboring state is generated by moving any one queen to a different row in the same column.

**2. Move Acceptance Based on Temperature:**

- **High Temperature**: At the start, the algorithm makes more random moves, even if the new state has more conflicts. The algorithm is willing to explore the search space more broadly and escape local minima.
  - **Action**: Move a randomly selected queen to any other row in its column, even if it results in more conflicts (higher cost).
- **Low Temperature**: As the temperature decreases, the algorithm becomes less likely to accept bad moves. It behaves more like steepest-ascent hill climbing, focusing on improving moves (those that reduce conflicts).
  - **Action**: Select a queen and move it to a position that either reduces conflicts or keeps the number of conflicts the same. However, if a small cost increase happens, it will still be accepted with some probability.

**3. Cooling Schedule and Behavior:**

- **Cooling Schedule**: The temperature decreases gradually, controlling how likely the algorithm is to accept worse moves. At high temperatures, the algorithm is more exploratory, while at low temperatures, it becomes more exploitative, fine-tuning the solution.
- **Actions based on Temperature**:
  - **High Temperature**:
    - Explore random neighboring states, accepting worse states frequently.
    - This prevents the algorithm from getting stuck in a local minimum early on.
  - **Low Temperature**:
    - The algorithm becomes more selective, accepting only moves that improve the current solution or slightly worsen it.

## Example Walkthrough of Simulated Annealing for N-Queens:

**Step 1: Initialize**

- Start with a random configuration:
    - Example: $N = 8$, queens placed randomly in each column:
    - State: $[4, 6, 3, 7, 1, 8, 5, 2]$ (where the numbers represent the row of the queen in each column).
    - Initial cost: 5 conflicts (e.g., some queens share the same diagonal).

**Step 2: Choose a Neighboring State**

- Pick a random queen, say the queen in column 2 (currently in row 6), and move it to a different row, say row 3. This creates a new state:
    - New state: $[4, 3, 3, 7, 1, 8, 5, 2]$.
    - New cost: 6 conflicts (more conflicts than before).

### Step 3: **Acceptance Decision**

- At **high temperature**: The algorithm may accept this worse move with a certain probability. Let's assume $T = 100$, and the cost increase $\Delta E = 1$:
  - Acceptance probability: $P(\Delta E) = e^{-\frac{1}{100}} \approx 0.99$ (so it's very likely to accept the worse move).
- If accepted, the new state becomes the current state.

### Step 4: **Reduce Temperature**

- Update the temperature according to the cooling schedule:
  - $T_{\text{new}} = 0.95 \times T_{\text{old}}$.

### Step 5: **Repeat**

- Continue selecting random moves and making acceptance decisions based on the current temperature until the temperature is sufficiently low, at which point the algorithm will focus mostly on reducing the number of conflicts.

| Algorithm | Optimality for Small $N$ | Optimality for Large $N$ | Steps for Small $N$ | Steps for Large $N$ |
|---|---|---|---|---|
| **Simple Hill Climbing** | 70-80% | < 5% | 200-300 | 10,000+ (fails often) |
| **Steepest-Ascent Hill Climbing** | 85-90% | 15-20% | 150-200 | 8,000-12,000 |
| **First-Choice Hill Climbing** | 75-85% | 10-15% | 250-350 | 9,000-11,000 |
| **Stochastic Hill Climbing** | 80-85% | 25-30% | 300-400 | 8,000-10,000 |
| **Random-Restart Hill Climbing** | 95-99% | 40-50% | 500-600 (due to restarts) | 15,000-20,000 (due to restarts) |
| **Simulated Annealing** | 90-95% | 70-80% | 300-500 | 10,000-15,000 |

# Prompt-Based Editing for Text Style Transfer

## 3.2 Search Objective

We apply an edit-based search for unsupervised style transfer. This follows the recent development

In our work, we use the SAHC algorithm: in a search step $t$, SAHC enumerates every editing position and performs every editing operation (namely, word deletion, replacement, and insertion)[5]. Then it selects the highest-scored candidate sentence $\mathbf{y}^*$ if the score $f(\mathbf{y}^*, \mathbf{x})$ is higher than $f(\mathbf{y}^{(t-1)}, \mathbf{x})$ before it reaches the maximum edit steps. Otherwise, SAHC terminates and takes the candidate $\mathbf{y}^{(t-1)}$ as the style-transferred output. In this way, our SAHC greedily finds the best edit for every search step and is more powerful than SA and FCHC in our

## 3.3 Discrete Search Algorithm

We perform style-transfer generation by discrete local search using editing operations, such as word insertion, deletion, and replacement, following pre-

| Dataset | Algorithm | ACC% | BLEU | GM | HM |
|---|---|---|---|---|---|
| YELP | SAHC | **73.0** | **40.1** | **54.1** | **51.7** |
| | FCHC | 67.2 | 31.8 | 46.2 | 43.1 |
| | SA | 66.0 | 28.7 | 43.5 | 40.0 |
| AMAZON | SAHC | **72.7** | **28.6** | **45.6** | **41.0** |
| | FCHC | 64.1 | 24.8 | 39.8 | 35.7 |
| | SA | 63.2 | 23.7 | 38.7 | 34.4 |

Table 5: Results of different search algorithms on the sentiment transfer datasets.

# Unsupervised Paraphrasing by Simulated Annealing

We propose UPSA, a novel approach that accomplishes Unsupervised Paraphrasing by Simulated Annealing. We model paraphrase generation as an optimization problem and propose a sophisticated objective function, involving semantic similarity, expression diversity, and language fluency of paraphrases. UPSA searches the sentence space towards this objective by performing a sequence of local edits.

| Line # | UPSA Variant | iBLEU | BLEU | Rouge1 | Rouge2 |
|---|---|---|---|---|---|
| 1 | UPSA | **12.41** | 18.48 | 57.06 | 31.39 |
| 2 | w/o $f_{\text{sim,key}}$ | 10.28 | 15.34 | 50.85 | 26.42 |
| 3 | w/o $f_{\text{sim,sen}}$ | 11.78 | 17.95 | 57.04 | 30.80 |
| 4 | w/o $f_{\text{exp}}$ | 11.93 | 21.17 | 59.75 | 34.91 |
| 5 | w/o copy | 11.42 | 17.25 | 56.09 | 29.73 |
| 6 | w/o annealing | 10.56 | 16.52 | 56.02 | 29.25 |

Let $\mathcal{X}$ be a (huge) search space of sentences, and $f(\mathrm{x})$ be an objective function. The goal is to search for a sentence x that **maximizes** $f(\mathrm{x})$. At a searching step $t$, SA keeps a current sentence $\mathrm{x}_t$, and proposes a new candidate $\mathrm{x}_*$ by local editing. If the new candidate is better scored by $f$, i.e., $f(\mathrm{x}_*) > f(\mathrm{x}_t)$, then SA accepts the proposal. Otherwise, SA tends to reject the proposal $x_*$, but may still accept it with a small probability $e^{\frac{f(\mathrm{x}_*)-f(\mathrm{x}_t)}{T}}$, controlled by an annealing temperature $T$. In other words, the probability of accepting the proposal is

$$p(\text{accept}|\mathrm{x}_*, \mathrm{x}_t, T) = \min\left(1, e^{\frac{f(\mathrm{x}_*)-f(\mathrm{x}_t)}{T}}\right). \quad (1)$$

# Simulated Annealing for Emotional Dialogue Systems

ions. In this study, we consider the task of expressing a specific emotion for dialogue generation. Previous approaches take the emotion as an input signal, which may be ignored during inference. We instead propose a search-based emotional dialogue system by simulated annealing (SA). Specifically, we first define a scoring function that combines contextual coherence and emotional correctness. Then, SA iteratively edits a general response and searches for a sentence with a higher score, enforcing the presence of the desired emotion. We evaluate our system on the NLPCC2017 dataset.

## 2.3 Search by Simulated Annealing (SA)

We use the simulated annealing (SA) algorithm to search for a desired utterance. SA starts from a general dialogue response $\mathbf{y}^{(0)} = \operatorname{argmax} P_{\text{Seq2Seq}}(\mathbf{y}|\mathbf{x})$, obtained by standard beam search (BS) or diverse beam search (DBS) on the trained Seq2Seq model. Then, SA maximizes the scoring function (5) by iteratively editing the candidate response.

| | Models | BLEU Scores | | Diversity | | Embedding-Based Metrics | | | | Emotion |
|---|---|---|---|---|---|---|---|---|---|---|
| | | BLEU-1 | BLEU-2 | Dist-1 | Dist-2 | Average | Greedy | Extreme | Coherence | accuracy |
| Previous | Seq2Seq | 4.24 | 0.73 | 0.035 | 0.119 | 0.497 | 0.328 | 0.352 | 0.582 | 0.244 |
| | EmoEmb | 7.22 | 1.64 | 0.040 | 0.133 | 0.532 | 0.356 | 0.381 | 0.594 | 0.693 |
| | EmoDS | 9.76 | 2.82 | 0.050 | 0.174 | 0.623 | 0.403 | 0.427 | 0.603 | 0.746 |
| | ECM | 10.23 | 3.32 | 0.052 | 0.177 | 0.625 | 0.405 | 0.433 | 0.607 | 0.753 |
| | CDL | 12.54 | 3.70 | **0.065** | 0.221 | 0.642 | 0.438 | 0.457 | 0.635 | 0.823 |
| Ours | Seq2Seq BS | 10.78 | 3.11 | 0.058 | 0.215 | 0.765 | 0.543 | 0.594 | 0.690 | 0.253 |
| | Seq2Seq BS + SA | 13.90 | 4.03 | 0.051 | **0.276** | 0.782 | **0.569** | 0.610 | 0.701 | 0.928 |
| | Seq2Seq DBS | 12.14 | 3.89 | 0.061 | 0.209 | 0.768 | 0.545 | 0.601 | 0.699 | 0.264 |
| | Seq2Seq DBS + SA | **14.26** | **4.12** | 0.053 | 0.239 | **0.786** | 0.556 | **0.611** | **0.703** | **0.942** |

## Local Beam Search

**Local Beam Search** is a type of heuristic search algorithm that explores the state space in search of a solution by maintaining multiple states (or solutions) at each step rather than just one. Unlike algorithms like hill climbing or gradient descent that work with a single state and progress iteratively, local beam search keeps track of **k** candidate states at each step, aiming to improve efficiency by exploring multiple parts of the search space simultaneously.

This method is effective in avoiding the problem of getting stuck in local optima, which is common in many single-state local search methods. It does so by allowing parallel exploration of the search space.

## Core Concepts of Local Beam Search

1. **Multiple States**:
   - Instead of one state, local beam search operates with **k** states at a time. This means that at each iteration, there is a parallel exploration happening in k different regions of the state space.
2. **Successor Generation**:
   - For each of the k states, their successors (neighboring states) are generated by applying actions or transformations. These are essentially new states that are one step away from the current states.
3. **Selection of Best States**:
   - From all the successors generated across all k states, the best k states are selected based on some evaluation criteria (e.g., the value of an objective function). These k states then become the current states for the next iteration.
4. **Termination**:
   - The process continues until either a goal state (a solution) is found, or a termination condition is met (such as a maximum number of iterations, time limit, or convergence).

## Algorithm Steps

1. **Initialization**:
   - Start with **k** randomly chosen initial states (or heuristically selected states if prior information is available). These states form the first beam.
2. **Generate Successors**:
   - For each state in the current beam, generate all possible successors. Successors are neighboring states that can be reached by applying some action or transformation to the current state.
3. **Evaluate Successors**:
   - Evaluate the successors using an evaluation function or objective function that measures the quality of the state (e.g., distance to goal, cost, or fitness score).
4. **Select the Best k States**:
   - From the pool of all successors generated across the k states, select the top k states based on the evaluation function. These selected states will form the next beam.
5. **Repeat**:
   - Repeat the process of generating successors and selecting the best states until a goal state is found or a stopping criterion is met (such as a limit on the number of iterations).
6. **Termination**:
   - The algorithm terminates if a solution (goal state) is found or if it reaches a stopping criterion such as a time limit, or if there are no improvements in the best states after several iterations.

# Example: Traveling Salesman Problem (TSP)

Consider the **Traveling Salesman Problem (TSP)**, where the objective is to find the shortest possible route that visits a set of cities exactly once and returns to the starting city. Using local beam search:

1. **Initialization**:
    - Start with k = 3 random routes (states), each representing a different ordering of cities.
2. **Successor Generation**:
    - For each route (state), generate successors by making small changes, such as swapping two cities or reversing a part of the route.
3. **Evaluation**:
    - Calculate the total distance of each route (state) as the evaluation metric. The goal is to minimize this distance.
4. **Selection**:
    - Choose the 3 shortest routes from the pool of generated successors.
5. **Iteration**:
    - Repeat the process, generating new routes from the best 3 and selecting the shortest ones, until a satisfactory route is found or a stopping criterion is met.

## Key Properties

1. **Parallel Search**:
    - Unlike single-state search methods (such as hill climbing or gradient descent), local beam search performs parallel exploration of the search space. By maintaining multiple states, it can avoid local optima more effectively.
2. **Focuses on the Best**:
    - Local beam search selectively keeps the best k states in each iteration. This means the search is directed toward regions of the state space that seem most promising, which makes the search more focused and efficient.

## Challenges and Limitations

1. **Loss of Diversity**:
   - A major challenge in local beam search is the potential **loss of diversity**. If all k states converge to a similar region of the search space, the algorithm may prematurely focus on a suboptimal region, missing better solutions elsewhere.
   - **Solution**: **Stochastic Beam Search**, a variation of local beam search, helps mitigate this by probabilistically selecting the next k states based on their evaluation scores rather than deterministically choosing the top k. This keeps diversity in the search process.
2. **Computational Overhead**:
   - Maintaining and evaluating k states in parallel can lead to higher computational costs compared to single-state search methods, especially if k is large or if generating successors is computationally expensive.
3. **Beam Width**:
   - Choosing the correct number of beams (the value of k) is crucial. If k is too small, the algorithm may not explore enough of the search space. If k is too large, it could become computationally expensive without adding much benefit.

## Example of Local Beam Search in N-Queens (With $k$ =3)

**Initial Step: Randomly Initialize 3 Board Configurations**

- **State 1**: $[2, 4, 6, 8, 1, 3, 5, 7]$
  (Cost: 6 conflicts)
- **State 2**: $[3, 1, 4, 7, 2, 5, 8, 6]$
  (Cost: 4 conflicts)
- **State 3**: $[1, 3, 5, 8, 6, 4, 2, 7]$
  (Cost: 5 conflicts)

**Step 2: Generate Successors for Each State**

For each state, generate new configurations by moving one queen to a different row in its column.

Let's generate a few successors for each state.

- **State 1**: $[2, 4, 6, 8, 1, 3, 5, 7]$
  Possible moves for queens:
  - Move queen in column 1 (currently row 2) to rows 1, 3, ..., 8.
  - Move queen in column 2 (currently row 4) to rows 1, 2, ..., 8.
  - Continue generating successors by changing positions of other queens.
  - **Sample successor**: $[1, 4, 6, 8, 1, 3, 5, 7]$ (Cost: 5 conflicts)

- **State 2**: $[3, 1, 4, 7, 2, 5, 8, 6]$

  Possible moves for queens:

  - Move queen in column 1 (currently row 3) to rows 1, 2, ..., 8.
  - Move queen in column 2 (currently row 1) to rows 2, 3, ..., 8.
  - Continue generating successors for other queens.
  - **Sample successor**: $[3, 2, 4, 7, 2, 5, 8, 6]$ (Cost: 3 conflicts)

- **State 3**: $[1, 3, 5, 8, 6, 4, 2, 7]$

  Possible moves for queens:

  - Move queen in column 1 (currently row 1) to rows 2, 3, ..., 8.
  - Move queen in column 2 (currently row 3) to rows 1, 2, ..., 8.
  - Continue generating successors for other queens.
  - **Sample successor**: $[1, 2, 5, 8, 6, 4, 2, 7]$ (Cost: 4 conflicts)

**Step 3: Evaluate Successors and Select the Best $k$**

After generating a large number of successors from each state, evaluate their costs (conflicts).

Here are some sample successors with their costs:

- **From State 1**:
  - $[1, 4, 6, 8, 1, 3, 5, 7]$: 5 conflicts
  - $[3, 4, 6, 8, 1, 3, 5, 7]$: 4 conflicts
  - $[2, 1, 6, 8, 1, 3, 5, 7]$: 5 conflicts
- **From State 2**:
  - $[3, 2, 4, 7, 2, 5, 8, 6]$: 3 conflicts
  - $[3, 1, 4, 7, 3, 5, 8, 6]$: 4 conflicts
- **From State 3**:
  - $[1, 2, 5, 8, 6, 4, 2, 7]$: 4 conflicts
  - $[2, 3, 5, 8, 6, 4, 2, 7]$: 4 conflicts

Now, choose the **best 3 states** (with the lowest number of conflicts) to continue the search. For example:

- $[3, 2, 4, 7, 2, 5, 8, 6]$ (Cost: 3 conflicts)
- $[3, 4, 6, 8, 1, 3, 5, 7]$ (Cost: 4 conflicts)
- $[1, 2, 5, 8, 6, 4, 2, 7]$ (Cost: 4 conflicts)

**Step 4: Repeat the Process**

Repeat the process by generating successors for these 3 selected states, evaluating them, and selecting the best 3 configurations to continue.

**Termination:**

- If one of the states reaches **0 conflicts** (no queens attacking each other), the algorithm terminates with a solution.
- If after several iterations no improvement is made, the algorithm may stop (based on a predefined stopping criterion).