Design and Analysis of Algorithm CS-213

Topic: Harmony Search Algorithm

Submitted by:

Wania Baig 4724-FOC/BSCS/F22 B

Areeba Manzoor 4693-FOC/BSCS/F22 B

Mariyam Tariq 4691-FOC/BSCS/F22 B

Zujaja Rasheed 4668-FOC/BSCS/F22 B

Submitted to:

Ma'am Maria Ashraf

Content

1.1 Metaheuristic algorithms

Metaheuristic algorithms are problem-solving techniques inspired by natural phenomena. They work by iteratively searching for the best solution within a large solution space. They start with an initial solution and then use a set of rules to explore the solution space, gradually improving the solution. These algorithms are often used to solve complex optimization problems that are difficult to solve using traditional methods.

1.1.1 Background and History

Metaheuristic algorithms have their roots in the 1950s and 1960s when early optimization techniques like genetic algorithms and simulated annealing were inspired by natural and physical processes. Over time, researchers developed other metaheuristics, such as particle swarm optimization (inspired by bird flocking) and ant colony optimization (based on ant foraging). These algorithms were created to solve complex optimization problems by mimicking real-world behaviors and processes, making them flexible and powerful tools across various fields like engineering, economics, and artificial intelligence.

1.2 Types of meta-heuristic algorithms

There are many types of heuristic algorithms some of them are:

1. Firefly Algorithm
2. Simulated Annealing
3. Genetic Algorithms
4. Ant Colony Optimization
5. Harmony search and many more

1.3 General Applications of Meta-Heuristics

- **Optimization Problems**
  Solving complex optimization tasks such as scheduling, route planning, and resource allocation, where exact methods are computationally expensive or infeasible.
- **Engineering Design**
  Used in structural optimization, circuit design, and control system tuning to find near-optimal solutions for complex engineering systems.
- **Artificial Intelligence and Machine Learning**
  Assists in hyperparameter tuning, feature selection, and neural network training.
- **Scientific and Industrial Applications**
  Applied in resource allocation, water distribution systems, power grid optimization, and chemical process optimization.
- **Operations Research**
  Used in large-scale combinatorial problems like transportation planning, supply chain optimization, and workforce scheduling.

1.4 Harmony

Harmony, in music, is the pleasing combination of different sounds played simultaneously. It's about how different musical notes work together to create a beautiful and satisfying sound.

1.5 Harmony Search Algorithm (HS)

1.5.1 Definition

The **Harmony Search Algorithm (HS)** is a metaheuristic optimization technique inspired by the process of musical improvisation. It mimics the way musicians in a band or orchestra create harmonious music by adjusting their instruments to play in tune with each other. Similarly, in the context of optimization, HS combines different solutions to find a near-optimal or optimal outcome.

1.5.2 Background and History

The Harmony Search algorithm was invented by **Zong Woo Geem** in 2001. Geem, a researcher from South Korea, was inspired by the concept of musical improvisation, where musicians adjust their instruments and play together to create a harmonious performance. He applied this idea to optimization problems, where different solutions (similar to musical notes) are adjusted and combined to find the best possible solution.

1.6 Characteristics

Parameter Simplicity: Harmony Search requires fewer parameters compared to other metaheuristic algorithms, making it easier to implement and adjust. This simplicity reduces the computational burden and makes it user-friendly for a wide range of problems.

Flexibility: The algorithm is highly adaptable and works effectively for various optimization problems. Its ability to handle both discrete and continuous variables makes it versatile for applications in different domains like engineering and operations research.

Exploration-Exploitation Balance: Harmony Search ensures a good balance between exploring the solution space and exploiting the best solutions. This balance helps avoid getting stuck in local optima while still converging on high-quality solutions.

Global Optimization: The algorithm is designed to find optimal or near-optimal solutions efficiently. Its mechanism mimics the creative process of musical improvisation, which allows it to explore innovative and effective solutions.

1.7 How harmony search works?

Harmony Search works by simulating the improvisation process of musicians to find optimal solutions. It begins with the **initialization** phase, where a set of random solutions, called the Harmony Memory (HM), is generated. During the **improvisation** phase, new solutions are created by either selecting or combining existing ones from the HM, slightly adjusting them to explore nearby regions, or generating completely random solutions to maintain diversity. Each newly generated solution is then **evaluated** based on its quality, and if it outperforms weaker solutions in the HM, it replaces them. This process is repeated iteratively until a predefined stopping criterion, such as a maximum number of iterations or a desired solution quality, is met.

1.8 HS Operators

1. **Random Playing**: Adds random elements to explore new ideas or possibilities.
2. **Memory Considering**: Uses previous good solutions to guide the current decisions.
3. **Pitch Adjusting**: Makes small changes to improve the solution, similar to fine-tuning a musical note.
4. **Ensemble Considering**: Ensures all parts work well together, like a group of musicians playing in harmony.
5. **Dissonance Considering**: Sometimes introduces unexpected or unusual choices to bring variety and avoid getting stuck in one idea.

1.9 Three rules

1.9.1 Use from HM

Harmony Search maintains a collection of solutions known as the Harmony Memory (HM), which stores good solutions found during the search. When generating a new solution, the algorithm picks values (like notes in music) from the existing solutions in the HM. This ensures that the new solution is based on previous good solutions, guiding the search toward better results.

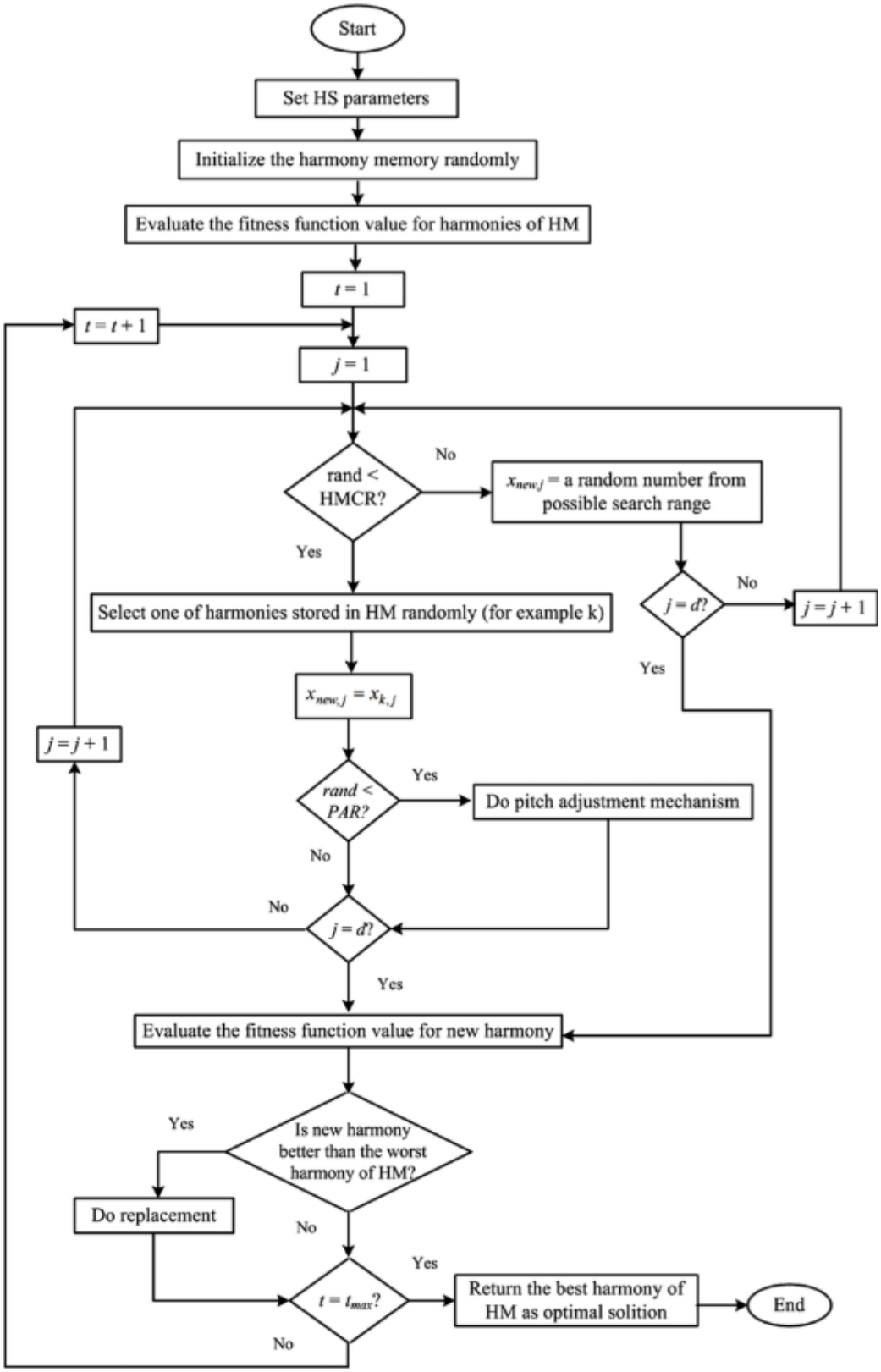
1.9.2 Slight adjustment in pitch

After selecting values from the Harmony Memory, the algorithm makes small adjustments (called **pitch adjustment**) to the solution's components. This is like tuning a musical note. The adjustment helps improve the solution, ensuring it's not too similar to existing solutions and increasing the chances of finding an optimal or better solution.

1.9.3 Random note generation

Occasionally, the algorithm generates a completely **random solution** rather than relying only on the existing solutions from the Harmony Memory. This introduces diversity and prevents the algorithm from getting stuck in a local optimum. By exploring new possibilities, it helps the algorithm discover potentially better solutions that were not part of the known solutions.

1.10 Flow chart



1.11 Algorithm

```
function HarmonySearch(HMS, HMCR, PAR, bw, max_iterations)
```

Initialize HM with HMS random solutions
Evaluate fitness of each solution in HM

for t = 1 to max_iterations
for each solution in HM
for each parameter j
rand = random number
if rand < HMCR
k = random index in HM
copy parameter j from solution k to current solution
if rand < PAR
adjust parameter j by random value within bw
else
generate random value for parameter j
end if
end for
Evaluate fitness of the new solution
if new solution is better than worst solution in HM
replace worst solution with new solution
end if
end for
end for

return best solution from HM
end function

Notes:

- HMCR: Harmony Memory Consideration Rate.
- PAR: Pitch Adjustment Rate.
- bw: Bandwidth for adjusting parameters.
- max_iterations: Number of iterations for optimization.

1.12 Time Complexity Analysis

1. Initialization

Initializing **HMS** random solutions: **O(HMS * n)** for generating each solution with **n** parameters.

2. Outer Loop

for t = 1 to max_iterations

The outer loop runs **max_iterations** times, so it contributes **O(max_iterations)**.

3. Inner Loop (Loop Over Solutions in Harmony Memory)

The inner loop runs for **HMS** solutions, so this contributes **O(HMS)**.

4. Parameter Loop

for each parameter j

This loop runs for **n** parameters of each solution, so this contributes **O(n)**.

5. Random Selection and Memory Consideration

It takes **O(1)**(Steps like random number generation, copying a parameter, and adjusting it are constant time operations.)

7. Fitness Evaluation

Fitness evaluation is assumed to be **O(1)** per solution (though it can vary based on the problem).So this contributes **O(1)** time for each solution.

8. Checking and Updating Harmony Memory

Finding the worst solution in **HM** requires scanning through all **HMS** solutions, so it is **O(HMS)**.Replacing the worst solution with the new one is **O(1)**.So the time complexity for this step is **O(HMS)**.

9. Final Return

Finding the best solution involves scanning through **HMS** solutions, which is **O(HMS)**.

Overall Time Complexity

Now, combining all the time complexities from each step.So, the overall time complexity is **O(HMS×n×max_iterations).**

Time Complexity Summary

Aspects	Description	Time complexity
Best case	- Few iterations required for convergence. - Minimal adjustments needed.	O(HMS×n×max_iterations)
Average case	- Moderate number of iterations and adjustments. - Pitch Adjustment Rate (PAR) triggered occasionally.	O(HMS×n×max_iterations)
Worst case	- Large Harmony Memory Size (HMS). - Numerous iterations before convergence. - Frequent parameter adjustments and slow fitness evaluation. - Algorithm struggles with local minima.	O(HMS×n×max_iterations)

1.13 Space Complexity

The space complexity is primarily driven by the storage of the harmony memory and the number of solutions. Thus, the space complexity is **O(HMS×n).`**

1.14 How to avoid worst case scenario?

To avoid the worst-case scenario in Harmony Search, there are a few strategies you can use. First, adjust the algorithm's settings during the process to help it better balance exploring new solutions and improving existing ones. Starting with diverse solutions is also important, as it prevents the algorithm from getting stuck in a bad starting point. You can also set dynamic stopping rules to end the algorithm if it's not making progress, saving time. Combining Harmony Search with other methods, like Genetic Algorithms, can help prevent it from getting stuck in local problems. Lastly, periodically introducing new random solutions keeps the search process fresh and helps the algorithm explore more possibilities.

1.15 Code

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

// Objective function to minimize (e.g., f(x) = x^2)
double objectiveFunction(double x) {
    return x * x;  // You can change this to any function you want to minimize
}

int main() {
    srand(time(0));  // Initialize random number generator

    // Define the Harmony Memory Size (how many solutions will be stored in the
memory)
    int HM_size = 10;  // Example: Use 10 solutions in memory

    // Create an array to store harmony memory solutions
    double* harmony_memory = new double[HM_size];

    // Step 1: Initialize harmony memory with random values
    for (int i = 0; i < HM_size; i++) {
        harmony_memory[i] = (rand() / (double)RAND_MAX) * 2 - 1;  // Random
values between -1 and 1
        cout << "Initial Harmony Memory[" << i << "]: " << harmony_memory[i] <<
endl;
    }

    // Step 2: Define the Harmony Memory Considering Parameters
    double HMCR = 0.9;   // Harmony Memory Considering Rate (Probability of
selecting from harmony memory)
    double PAR = 0.3;    // Pitch Adjustment Rate (Probability of adjusting
selected value)
    double BW = 0.1;     // Bandwidth for the adjustment

    // Step 3: Create a variable to store the best solution found
    double best_solution = harmony_memory[0];
    double best_fitness = objectiveFunction(best_solution);

    // Step 4: Start the search process (this could be a number of iterations)
    int num_iterations = 1000;
    for (int iter = 0; iter < num_iterations; iter++) {
        // Step 5: Generate a new harmony solution
        double new_solution = 0.0;

        // Decide whether to select a harmony from the memory (HMCR) or
generate randomly
        if (rand() / double(RAND_MAX) < HMCR) {
            // Select a random harmony from memory
            int index = rand() % HM_size;
            new_solution = harmony_memory[index];
```

```
            // Apply pitch adjustment with probability PAR
            if (rand() / double(RAND_MAX) < PAR) {
                new_solution += (rand() / double(RAND_MAX) - 0.5) * 2 * BW;  //
Adjustment in the range of -BW to +BW
            }
        }
        else {
            // Generate a random solution
            new_solution = (rand() / (double)RAND_MAX) * 2 - 1;  // Random
between -1 and 1
        }

        // Step 6: Evaluate the fitness of the new solution
        double new_fitness = objectiveFunction(new_solution);

        // Step 7: If the new solution is better, update harmony memory
        if (new_fitness < best_fitness) {
            best_solution = new_solution;
            best_fitness = new_fitness;

            // Update the harmony memory: find and replace the worst solution
            int worst_index = 0;
            double worst_fitness = objectiveFunction(harmony_memory[0]);
            for (int i = 1; i < HM_size; i++) {
                double fitness = objectiveFunction(harmony_memory[i]);
                if (fitness > worst_fitness) {
                    worst_fitness = fitness;
                    worst_index = i;
                }
            }

            // Replace the worst harmony solution with the new better solution
            harmony_memory[worst_index] = best_solution;
        }
    }

    // Step 8: Output the best solution found
    cout << "Best solution found: " << best_solution << endl;
    cout << "Best fitness (objective function value): " << best_fitness <<
endl;

    return 0;
}
```

Output:

```
Initial Harmony Memory[0]: -0.436934
Initial Harmony Memory[1]: -0.806146
Initial Harmony Memory[2]: 0.0679037
Initial Harmony Memory[3]: -0.448347
Initial Harmony Memory[4]: -0.432966
Initial Harmony Memory[5]: 0.768792
Initial Harmony Memory[6]: 0.477645
Initial Harmony Memory[7]: -0.763848
Initial Harmony Memory[8]: 0.823542
Initial Harmony Memory[9]: 0.916746
Best solution found: -0.000155644
Best fitness : 2.42252e-08
```

1.16 Example

1.16.1 Real-Life Application Example

Creating a Playlist with Harmony Search Algorithm

Problem:

You want to create a playlist for a party with the following criteria:

1. **Energy Level**: Select upbeat songs.
2. **Genre**: Prefer pop songs.

Available Songs:

- **Song 1**: Low Energy, Pop
- **Song 2**: High Energy, Pop
- **Song 3**: Low Energy, Jazz
- **Song 4**: High Energy, Rock
- **Song 5**: High Energy, Pop

Steps with Harmony Search Algorithm:

1. **Define the Objective**:
    - Maximize **energy level** and select **pop songs** for the playlist.
2. **Initialize Harmony Memory (HM)**:
    - Randomly select three initial playlists as the starting point.
        - **Playlist 1**: Song 1, Song 3, Song 5
        - **Playlist 2**: Song 2, Song 4, Song 5
        - **Playlist 3**: Song 1, Song 2, Song 5
3. **Evaluate Fitness**:
    - Assess each playlist based on the criteria (energy level and genre).
4. **Improve Playlists Using HS Operators**:
    - **Memory Consideration**: Use good playlists from the memory to guide the selection.
    - **Pitch Adjustment**: Make small changes, like replacing a song or adjusting criteria.
    - **Random Playing**: Add random songs to explore new possibilities.
5. **Repeat Until Convergence**:
    - Continue improving playlists in each iteration until no significant improvement is possible.
6. **Select the Best Playlist**:
    - After iterations, choose the playlist that best meets the criteria.

This structured process helps create a playlist with an optimal mix of upbeat and pop songs for the party.

1.16.2 Numerical Problem

We want to find the value of x that minimizes the function:

f(x)=(x−3)^2

The goal is to find the value of x that makes f(x) as small as possible (the smallest value occurs at x=3).

Given Parameters:

- **HMS (Harmony Memory Size)**: 4 (Number of solutions in memory)
- **HMCR (Harmony Memory Considering Rate)**: 0.9 (90% chance to pick from memory)
- **PAR (Pitch Adjustment Rate)**: 0.7 (70% chance to adjust the solution)
- **bw (Bandwidth)**: 0.1 (Bandwidth for pitch adjustment)
- **max_iterations**: 5 (Number of iterations)

Step 1: Initialize Harmony Memory (HM)

We start by randomly choosing 4 solutions (values of xxx):

- Solution 1: x1=1.5
- Solution 2: x2=4.0
- Solution 3: x3=2.5
- Solution 4: x4=5.0

Step 2: Calculate Fitness (How good each solution is)

The fitness of each solution is calculated using the formula f(x)=(x−3)^2

- Fitness of x1=1.5: f(1.5)=(1.5−3)^2=2.25
- Fitness of x2=4.0: f(4.0)=(4.0−3)^2=1.0
- Fitness of x3=2.5: f(2.5)=(2.5−3)^2=0.25
- Fitness of x4=5.0: f(5.0)=(5.0−3)^2=4.0

Step 3: Apply Harmony Search Algorithm (1st Iteration)

Now we start the iterations, applying the HS algorithm for each solution.

Solution 1 (x = 1.5):

- **Memory Consideration**: 90% chance of picking a solution from memory. Let's say we pick solution x3=2.5.
- **Pitch Adjustment**: 70% chance of adjusting the solution. Let's say the value of x1x_1x1 gets adjusted by 0.1, so x1′=2.6.
- **New fitness**: f(2.6)=(2.6−3)^2 =0.16.

Solution 2 (x = 4.0):

- **Memory Consideration**: We pick solution x3=2.5 again.
- **No pitch adjustment** (because the random number was greater than 0.7).
- **New fitness**: f(2.5)=(2.5−3)^2= 0.25.

Solution 3 (x = 2.5):

- **Memory Consideration**: We pick solution x2=4.0.
- **Pitch Adjustment**: We adjust the value by 0.1, so x3′=4.1.
- **New fitness**: f(4.1)=(4.1−3)^2=1.21.

Solution 4 (x = 5.0):

- **Memory Consideration**: We pick solution x1=1.5.
- **Pitch Adjustment**: We adjust the value by 0.1, so x4′=1.6.
- **New fitness**: f(1.6)=(1.6−3)^2=1.96.

Step 4: Update Harmony Memory (after 1st iteration)

Now, we update the Harmony Memory by replacing the worst solution (the one with the highest fitness value):

The new Harmony Memory will look like this:

Solution	X value	fitness
1	2.6	0.16
2	2.5	0.25
3	4.1	1.21
4	1.6	1.96

Step 5: Repeat for More Iterations

The algorithm repeats this process for a total of 5 iterations, adjusting the solutions each time. Each iteration will bring the solutions closer to the optimal value (which is x=3).

Final Result:

After 5 iterations, the Harmony Search algorithm will return the best solution found from the Harmony Memory, which will be close to x=3.

1.17 Advantages of Harmony Search

1. **Easy to Use**: Harmony Search is simple to implement and doesn't need complex coding, making it user-friendly for beginners.
2. **Flexible**: It can solve different types of problems, whether they involve numbers, schedules, or even multiple goals at once, like trying to reduce both cost and time in a project..
3. **Balanced Search**: It does a good job of exploring new solutions while also refining the ones it already has, reducing the chance of getting stuck in bad solutions.
4. **No Need for Detailed Information**: Unlike some algorithms, Harmony Search doesn't need detailed information about the problem (like gradients or derivatives), so it works well for complex or messy problems.
5. **Works Well with Noisy Data**: It handles problems where there's uncertainty or noise in the data, which is common in real-world situations.
6. **Good for Large Problems**: It can handle big problems with lots of variables or parameters.

1.18 Disadvantages of Harmony Search

1. **Slow for Big Problems**: If the problem is too large, the algorithm can take a long time to find a good solution.
2. **Depends on Right Settings**: Harmony Search relies on specific settings (like how many solutions to store, how much to adjust a solution). If these are not set correctly, it may not work well.
3. **Can Get Stuck**: Sometimes, it may find a good solution but not the best one, and then get stuck there instead of finding a better one.
4. **Doesn't Guarantee the Best**: Harmony Search usually finds a good solution, but it doesn't always find the absolute best one.
5. **Takes Time to Find a Solution**: It might need to run many times before finding a good solution, which can take a lot of time, especially for tough problems.
6. **Can Be Inefficient**: It doesn't always take advantage of patterns or special features in the problem, so it might explore areas that aren't helpful.

1.19 Which type of Problem does HS solves?

Harmony Search (HS) is an optimization algorithm used to find the best solution for various problems. It is effective for both **minimization** problems, where the goal is to reduce values such as cost or error, and **maximization** problems, where the aim is to increase values like profit or efficiency. HS is particularly useful for **complex problems** with large and difficult-to-explore solution spaces, such as those found in engineering design, scheduling, or machine learning. It can also handle **nonlinear** problems, where the relationship between variables is intricate and not easily solved by traditional methods.

1.20 HS vs. Other Metaheuristic Algorithms

Aspect	Harmony Search (HS	Genetic Algorithm (GA)	Particle Swarm Optimization (PSO)
Inspiration	Musical improvisation	Musical improvisation	Musical improvisation
Solution Generation	Combines memory and random pitch adjustments	Combines memory and random pitch adjustments	Combines memory and random pitch adjustments
Strengths	Simplicity, adaptability	Simplicity, adaptability	Simplicity, adaptability
Weaknesses	Struggles with highly complex problems	Struggles with highly complex problems	Struggles with highly complex problems
Complexity	Simple, fewer parameters	Simple, fewer parameters	Simple, fewer parameters

1.21 Real world applications

1.21.1 Course Timetabling

You need to assign classes to rooms, teachers, and time slots, avoiding conflicts. Harmony Search can help by trying different combinations of assignments, evaluating their quality, and gradually improving the timetable.

1.21.2 Water Distribution Network Design

Designing a water distribution network involves determining the optimal pipe sizes and layout to efficiently supply water to homes and businesses. Harmony Search can explore various network configurations, considering factors like water pressure, flow rate, and construction costs.

1.21.3 Resource Allocation

Allocating resources like workers, machines, or budgets to different tasks is a common optimization problem. Harmony Search can help by trying different allocation strategies, assessing their performance and refining the allocations over time.

1.21.4 Job Shop Scheduling

In a manufacturing plant, jobs need to be assigned to machines and scheduled in a specific order to minimize production time and costs. Harmony Search can generate various job sequences and machine assignments, evaluating their efficiency and identifying the optimal schedule.

1.22 Challenges in harmony search

Harmony Search (HS) algorithm, while effective, faces several challenges:

1. **Premature Convergence:**
   Sometimes, the algorithm finds a good solution too soon and stops looking for better ones, missing the best possible solution.
2. **Parameter Sensitivity**:
   The success of the algorithm depends a lot on choosing the right settings, like the size of the memory (HMS) and how much to adjust solutions (PAR).
3. **Balancing Search and Improvement:**
   It can be tricky to strike the right balance between trying out new ideas (exploration) and improving current solutions (exploitation).
4. **Hard for Complex Problems:**
   When there are too many things to consider or too many restrictions, the algorithm might need a lot of time and resources to work well.
5. **Less Creativity Over Time (Lack of Diversity):**
   As the algorithm goes on, it may start reusing the same ideas and stop exploring new possibilities, which can limit its effectiveness.

1.23 Conclusion

This report examined the **Harmony Search (HS)** algorithm, highlighting its **simplicity**, **flexibility**, and effectiveness in solving **complex optimization problems**. The **numerical example** demonstrated how HS can iteratively refine solutions, making it suitable for tasks like **resource allocation** and **scheduling**.

Despite its advantages, HS faces challenges like **parameter sensitivity** and **premature convergence**. These can be mitigated by tuning parameters and combining HS with other methods. Overall, **HS** is a valuable tool for **real-world optimization problems**, offering **near-optimal solutions** even in complex scenarios, with future work focusing on improving its **robustness** and handling **high-dimensional problems**.

1.24 References

- **Geem, Z. W., Kim, J. H., & Loganathan, G. V.** (2001). A New Heuristic Optimization Algorithm: Harmony Search. *Simulation*, 76(2), 60-68.

  This is the foundational paper on Harmony Search and its basic principles.

- **Geem, Z. W., & Rhee, J. K.** (2011). Harmony Search Algorithm: Theory and Applications. *Springer Science & Business Media*.

This book covers the theory and various applications of Harmony Search, providing more in-depth insights and case studies.

- **Wang, L., & Zhang, J.** (2011). Harmony Search and its Applications. *Springer-Verlag*.

  Another important resource on the applications and variations of Harmony Search.

- **Kennedy, J., & Eberhart, R. C.** (1995). Particle Swarm Optimization. *Proceedings of IEEE International Conference on Neural Networks*, 1942-1948.

  A core reference for Particle Swarm Optimization, a method often compared to Harmony Search.

- **Goldberg, D. E.** (1989). Genetic Algorithms in Search, Optimization, and Machine Learning. *Addison-Wesley Longman Publishing Co.*

  A key book on Genetic Algorithms (GA), another optimization algorithm frequently compared with HS.

- **Pardalos, P. M., & Resende, M. G. C.** (2002). Handbook of Applied Optimization. *Oxford University Press*.

  A comprehensive resource on various optimization techniques, including Harmony Search.