The 7-Module Design Problem

A Discrete Optimization Problem

Design Configuration $\longrightarrow$ Cost / Performance Evaluation $\longrightarrow$ Search

Insulating Modules Problem

Problem definition

- Given a set of insulating modules (filters) and a function that evaluates the insulation of any given ordering,
- determine the ordering of modules that **maximizes the overall insulation**.

Solution representation (for 7 modules)

- A solution is a permutation of the module indices:
  $x_0 = [2\ \ 5\ \ 7\ \ 3\ \ 4\ \ 6\ \ 1]; F(x_0) = 10$ units

Key observations

- All permutations are feasible solutions
- The objective value depends on the *entire sequence*
- Small re-orderings may cause large changes in performance

This is a permutation-based combinatorial optimization problem.

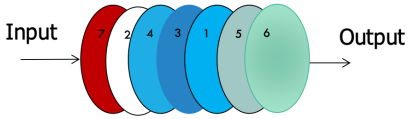


Figure: Filtering Sequence
Ordering of filters determines the overall insulating performance

Why metaheuristics for this problem?

- Exhaustive search is infeasible: $n!$ grows rapidly
- Greedy ordering easily gets trapped in poor local optima
- Cost surface is rugged and non-separable

Metaheuristic mapping

- Hill Climbing: fast, but easily trapped
- Simulated Annealing: probabilistic escape
- Tabu Search: deterministic escape using memory

Search Space and Neighborhood

Search space

- Number of possible solutions: $7! = 5040$
- Exhaustive search becomes infeasible as the number of modules increases

Typical neighborhood

- Swap two modules
- Insert one module at a different position
- Reverse a short subsequence

This structure makes the problem ideal for Tabu Search and Simulated Annealing.

Next Move

Neighborhood structure

- Define a move as swapping two modules
- This corresponds to a *pairwise exchange* neighborhood

Example solution

$$x_1 = [2 \;\; 6 \;\; 7 \;\; 3 \;\; 4 \;\; 5 \;\; 1]$$

Neighborhood size

- Number of possible swaps:

$$|N(x_1)| = \binom{7}{2} = 21$$

$$F(x_1) = 8$$

Each iteration evaluates only a tiny fraction of the full search space.

Why Neighborhood Size Matters

- Full search: $n!$ grows factorially (infeasible)
- Local search: $\mathcal{O}(n^2)$ neighbors per step
- Trade-off:
  - Larger neighborhoods: better moves, higher cost
  - Smaller neighborhoods: faster, but risk of poor local minima

Tabu Search balances neighborhood size with memory-based guidance.

Designing the Tabu List Data Structure

Key design challenge

How to implement the tabu list efficiently without excessive memory usage.

Matrix-based tabu representation

- Define the tabu list as an $n \times n$ matrix
$$\textbf{Tabu}(i, j)$$
- Entry (i, j) represents the swap move between modules i and j
- The value stored indicates how long this move remains tabu

Tabu update rule

- When a swap (i, j) is executed:
$$\textbf{Tabu}(i, j) \leftarrow T$$
where T is the tabu tenure
- After each iteration, all nonzero entries are decremented by 1

Interpretation

Designing the Tabu List Data Structure

Matrix-based tabu memory

- Tabu list represented as an $n \times n$ matrix

$$\textbf{Tabu}(i, j)$$

- Entry (i, j): swap between modules i and j
- Value stored = remaining tabu tenure

Short-term memory

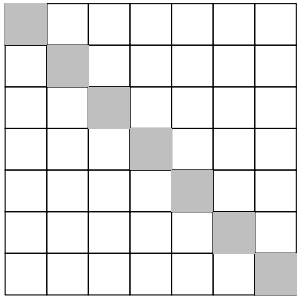
- Recently executed moves are marked tabu
- Prevents immediate reversal and cycling

Long-term memory (frequency)

- Tracks frequently used swaps or attributes

Tabu Matrix Illustration

Short-term memory



Long-term memory

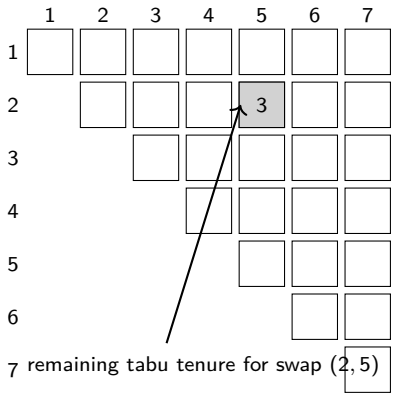
Example 1: Recency-Based Memory and Tabu Classification

- **Tabu attributes:** defined as the most recently executed swaps
- **Tabu tenure:** fixed to 3 iterations
- **Tabu rule:** any candidate solution that involves one of the three most recent swaps is classified as tabu
- **Aspiration criterion:** a tabu move is allowed if it produces a solution better than the best-so-far

Example 1: Initialization of Recency-Based Memory

- The **tabu list** is implemented as an upper-triangular matrix representing pairwise swap attributes
- Entry (i, j) stores the remaining tabu tenure for swapping modules i and j
- A nonzero value indicates that the corresponding swap is currently tabu
- In this example, the entry for module pair $(2, 5)$ indicates the remaining number of iterations for which this swap is forbidden

Note: Since swap (i, j) is identical to (j, i), only the upper triangular part of the matrix needs to be stored.



Example 1: Iteration 0

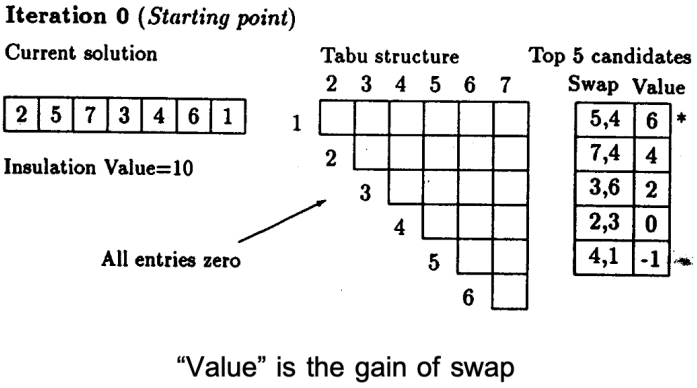


Figure: Iteration 0 of TABU on Filter Design

Example 1: Iteration 1

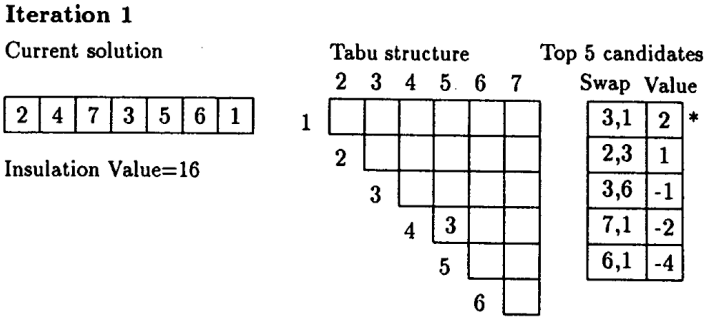


Figure: Iteration 1 of TABU on Filter Design

Example 1: Iteration 2

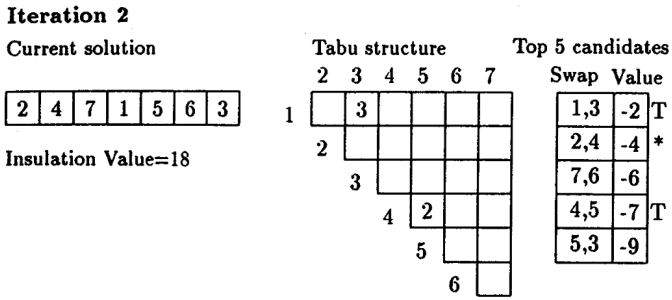


Figure: Iteration 2: Moves (1,3) and (4,5) have respective tabu tenures 3 and 2 No move with a positive gain, hence best (non-tabu) move will be non-improving

Example 1: Iteration 3

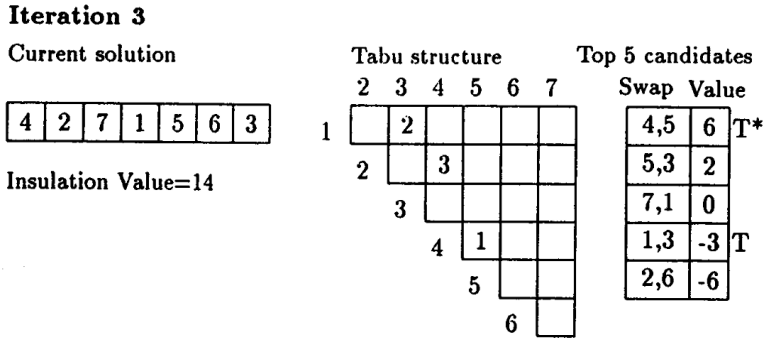


Figure: Move (4,5) has a tabu tenure of 1 iteration But this move results in the best solution so far Hence its tabu status is overridden

Example 1: Iteration 4

Iteration 4

Current solution

5	2	7	1	4	6	3

Insulation Value=20

Tabu structure

	2	3	4	5	6	7
1		1				
2			2			
3						
4				3		
5						
6						

Top 5 candidates

Swap	Value	
7,1	0	*
4,3	-3	
6,3	-5	
5,4	-6	T
2,6	-8	

Figure: Best move is (1,7)

Tabu Restrictions and Aspiration Criteria

Tabu Restrictions

A move is classified as **tabu** if it satisfies one of the following:

- Reverses a recently executed move (same exchange of positions)
- Involves any element (position or attribute) that participated in a recent tabu move
- Performs an inverse operation (e.g., add vs. delete instead of swap)

Goal: prevent cycling and enforce exploration.

Aspiration Criteria

A tabu move may be **overridden** if it satisfies one of the following:

- **Best-so-far rule:** produces a solution better than any seen so far
- **Objective threshold:** produces a solution better than an aspiration value
- **Search-direction rule:** does not reverse the current search trend

Goal: avoid over-restriction and allow strategic progress.

Intensification via Recency Memory

Key Observation

- Intensification is *not always necessary*
- In many cases, the search is already sufficiently thorough
- Over-intensifying too early may reduce exploration

When Intensification is Triggered

- Search shows prolonged stability
- Same components repeatedly appear in solutions
- Improvement slows or stagnates

Recency-Based Intensification

- Based on **recency memory**
- Memory records components that persist
- Counts consecutive iterations of uninterrupted presence

Intensification Phase

- Temporarily stop the normal search
- Fix persistent components
- Locally optimize the **best-known solution**

Intensification refines what the search has already identified as valuable