# **USER:**

#### System Prompt:

You are an expert in the domain of optimization heuristics. Your task is to design heuristics that can effectively solve optimization problems.

Your response outputs Python code and nothing else. Format your code as a Python code string: "```python ... ```".

#### User Prompt:

Write a crossover function for a crossover function to solving the port selection optimization problem using a genetic algorithm. The problem requires selecting a subset of ports from a given set of ports to serve users, with the goal of minimizing the transmission power. The encoding uses binary encoding, where the number of binary bits equals the number of ports, and a bit value of 1 represents the selection of that port. The solution generated after the crossover must be valid.

The `crossover` function takes as input a 2D NumPy array parents and an integer n\_pop. The function performs a genetic crossover operation on parents to generate n\_pop offspring. Use vectorized implementation if possible.

def crossover\_v1(parents: np.ndarray, n\_pop: int) -> np.ndarray:

```
n_parents, n_decap = parents.shape
# Split genomes into two halves
left_halves = parents[:, :n_decap // 2]
right_halves = parents[:, n_decap // 2:]
# Create parent pairs
parents_idx = np.stack([np.random.choice(range(n_parents), 2, replace=False)
for _ in range(n_pop)])
```

parents\_left = left\_halves[parents\_idx[:, 0]]

parents\_right = right\_halves[parents\_idx[:, 1]]

# Create offspring

offspring = np.concatenate([parents\_left, parents\_right], axis=1)

return offspring

Refer to the format of a trivial design above. Be very creative and give `crossover\_v2`. Output code only and enclose your code with Python code block: ```python ... ```

# LLM Response:

```python

import numpy as np

```
def crossover_v2(parents: np.ndarray, n_pop: int) -> np.ndarray:
```

n\_parents, n\_decap = parents.shape

# Random parent pair indices for each offspring

parent\_pairs = np.random.randint(0, n\_parents, size=(n\_pop, 2))

# Create mask with 50% chance for each parent's gene

mask = np.random.random(size=(n\_pop, n\_decap)) < 0.5

# Vectorized selection using mask

offspring = np.where(mask, parents[parent\_pairs[:, 0]], parents[parent\_pairs[:,

#### 1]])

return offspring

•••

**The comment:** Since the API is accessed by multiple processes, only the output returned by the LLM in one of the processes is shown here.

### **USER:**

#### System Prompt:

You are an expert in the domain of optimization heuristics. Your task is to give hints to design better heuristics.

#### User Prompt:

Below are two crossover functions for Write a crossover function for solving the port selection optimization problem using a genetic algorithm. The problem requires selecting a subset of ports from a given set of ports to serve users, with the goal of minimizing the transmission power. The encoding uses binary encoding, where the number of binary bits equals the number of ports, and a bit value of 1 represents the selection of that port. The solution generated after the crossover must be valid.

The `crossover` function takes as input a 2D NumPy array parents and an integer n\_pop. The function performs a genetic crossover operation on parents to generate n\_pop offspring. Use vectorized implementation if possible.

You are provided with two code versions below, where the second version performs better than the first one.

[Worse code]

n\_parents, n\_decap = parents.shape

# Create parent pairs with distinct indices using vectorized approach

first\_parent = np.random.randint(0, n\_parents, n\_pop)

second\_parent = (first\_parent + np.random.randint(1, n\_parents, n\_pop)) %

n\_parents

parent\_pairs = np.column\_stack((first\_parent, second\_parent))

# Extract parent genomes using advanced indexing

parents\_a = parents[parent\_pairs[:, 0]]

parents\_b = parents[parent\_pairs[:, 1]]

# Create random crossover mask with probability weights

crossover\_probs = np.random.uniform(0.4, 0.6, (n\_pop, n\_decap)) # Biased uniform crossover

mask = np.random.rand(n\_pop, n\_decap) < crossover\_probs

# Perform element-wise crossover using the probabilistic mask

offspring = np.where(mask, parents\_a, parents\_b)

# Ensure at least one port is selected (problem-specific validity)

all\_zero = (offspring.sum(axis=1) == 0)

offspring[all\_zero] = parents[np.random.choice(n\_parents, np.sum(all\_zero))] return offspring

#### [Better code]

n\_parents, n\_decap = parents.shape

# Random parent pair indices for each offspring

parent\_pairs = np.random.randint(0, n\_parents, size=(n\_pop, 2))

# Create mask with 50% chance for each parent's gene

mask = np.random.random(size=(n\_pop, n\_decap)) < 0.5

# Vectorized selection using mask

offspring = np.where(mask, parents[parent\_pairs[:, 0]], parents[parent\_pairs[:,

1]])

return offspring

You respond with some hints for designing better heuristics, based on the two code versions and using less than 20 words.

**The comment:** [Better code] and [Worse code] are both results returned by the LLM during the initialization phase.

### LLM Response:

Uniform crossover (50% per gene) and unrestricted parent selection enhance diversity and efficiency.

**The comment:** Since the API is accessed by multiple processes, only the output returned by the LLM in one of the processes is shown here.

### **USER:**

System Prompt:

You are an expert in the domain of optimization heuristics. Your task is to design heuristics that can effectively solve optimization problems.

Your response outputs Python code and nothing else. Format your code as a Python code string: "```python ... ```".

#### User Prompt:

Write a crossover function for Write a crossover function for solving the port selection optimization problem using a genetic algorithm. The problem requires selecting a subset of ports from a given set of ports to serve users, with the goal of minimizing the transmission power. The encoding uses binary encoding, where the number of binary bits equals the number of ports, and a bit value of 1 represents the selection of that port. The solution generated after the crossover must be valid.

The `crossover` function takes as input a 2D NumPy array parents and an integer n\_pop. The function performs a genetic crossover operation on parents to generate n\_pop offspring. Use vectorized implementation if possible.

[Worse code]

def crossover\_v0(parents: np.ndarray, n\_pop: int) -> np.ndarray:

```
n_parents, n_decap = parents.shape
```

# Create parent pairs with distinct indices using vectorized approach

first\_parent = np.random.randint(0, n\_parents, n\_pop)

```
second_parent = (first_parent + np.random.randint(1, n_parents, n_pop)) %
n_parents
```

```
parent_pairs = np.column_stack((first_parent, second_parent))
```

# Extract parent genomes using advanced indexing

parents\_a = parents[parent\_pairs[:, 0]]

parents\_b = parents[parent\_pairs[:, 1]]

# Create random crossover mask with probability weights

crossover\_probs = np.random.uniform(0.4, 0.6, (n\_pop, n\_decap)) # Biased uniform crossover

mask = np.random.rand(n\_pop, n\_decap) < crossover\_probs</pre>

# Perform element-wise crossover using the probabilistic mask

offspring = np.where(mask, parents\_a, parents\_b)

# Ensure at least one port is selected (problem-specific validity)

```
all_zero = (offspring.sum(axis=1) == 0)
```

offspring[all\_zero] = parents[np.random.choice(n\_parents, np.sum(all\_zero))]

return offspring

[Better code]

def crossover\_v1(parents: np.ndarray, n\_pop: int) -> np.ndarray:

```
n_parents, n_decap = parents.shape
# Random parent pair indices for each offspring
parent_pairs = np.random.randint(0, n_parents, size=(n_pop, 2))
# Create mask with 50% chance for each parent's gene
mask = np.random.random(size=(n_pop, n_decap)) < 0.5
# Vectorized selection using mask
offspring = np.where(mask, parents[parent_pairs[:, 0]], parents[parent_pairs[:, 1]])
```

return offspring

#### [Reflection]

Uniform crossover (50% per gene) and unrestricted parent selection enhance diversity and efficiency.

#### [Improved code]

Please write an improved function `crossover\_v2`, according to the reflection. Output code only and enclose your code with Python code block: ```python ... ```.

### LLM Response:

```
def crossover v1(parents: np.ndarray, n pop: int) -> np.ndarray:
    n_parents, n_ports = parents.shape
    indices = np.arange(n ports)
    # Select distinct parent pairs using vectorized operations
    parent a idx = np.random.randint(n parents, size=n pop)
    parent b idx = np.random.randint(n parents - 1, size=n pop)
    parent b idx += (parent b idx >= parent a idx)
    # Gather parent chromosomes
    parent_a = parents[parent_a_idx]
    parent b = parents[parent b idx]
    # Generate two-point crossover masks
    cross_points = np.random.randint(0, n_ports + 1, size=(n_pop, 2))
    cross points.sort(axis=1)
    mask = (indices >= cross points[:, 0, None]) & (indices < cross points[:, 1,
None])
    # Combine parent genetic material
```

offspring = np.where(mask, parent b, parent a)

return offspring.astype(np.int32)

### USER:

#### System Prompt:

You are an expert in the domain of optimization heuristics. Your task is to give hints to design better heuristics.

#### User Prompt:

Below is your prior long-term reflection on designing heuristics for Write a crossover function for solving the port selection optimization problem using a genetic algorithm. The problem requires selecting a subset of ports from a given set of ports to serve users, with the goal of minimizing the transmission power. The encoding uses binary encoding, where the number of binary bits equals the number of ports, and a bit value of 1 represents the selection of that port. The solution generated after the crossover must be valid.

Below are some newly gained insights.

Uniform crossover (50% per gene) and unrestricted parent selection enhance diversity and efficiency.

- 1. Ensure offspring validity via repair steps.
- 2. Check and fix empty port selections post-crossover.
- 3. Allow parent pairing with replacement for diversity.
- 4. Use mask-based crossover with validity guarantees.

Hints: Ensure diverse parent pairs, avoid correlated selections, and maintain genetic diversity.

Hints: Prefer uniform crossover over single-point for diversity. Avoid disruptive postcrossover mutations. Simplify gene selection logic.

- 1. Prefer structured crossovers (e.g., two-point) over uniform for validity.
- 2. Ensure distinct parent pairs to diversify offspring.

3. Use vectorized masks for efficiency.

4. Inherit validity via crossover design, not fixes.

Handle validity via mutation, not crossover. Avoid repair bias. Use uniform crossover for diversity.

Hints:

- 1. Simplify crossover masks.
- 2. Avoid redundant validity checks.
- 3. Use uniform parent selection.
- 4. Prefer full vectorization.

Use distinct parent pairs, two-point crossover, and avoid excessive random swapping.

Use structured crossover points, maintain genetic blocks, ensure parent diversity.

- 1. Prefer uniform crossover over single-point for diversity
- 2. Ensure validity constraints explicitly
- 3. Avoid destructive random modifications
- 4. Use problem-specific repair mechanisms

Write constructive hints for designing better heuristics, based on prior reflections and new insights and using less than 50 words.

The comment: This section contains reflective outputs from multiple LLMs..

# LLM Response:

\*\*Hints:\*\* Blend uniform crossover (diversity) with two-point validity checks. Use vectorized masks, ensure distinct parents via uniform selection. Design crossover to inherit valid gene blocks, avoiding empty solutions without repair. Prioritize structured diversity over random swaps.

### **USER:**

#### system Prompt:

You are an expert in the domain of optimization heuristics. Your task is to design heuristics that can effectively solve optimization problems.

Your response outputs Python code and nothing else. Format your code as a Python code string: "```python ... ```".

#### User Prompt:

Write a crossover function for Write a crossover function for solving the port selection optimization problem using a genetic algorithm. The problem requires selecting a subset of ports from a given set of ports to serve users, with the goal of minimizing the transmission power. The encoding uses binary encoding, where the number of binary bits equals the number of ports, and a bit value of 1 represents the selection of that port. The solution generated after the crossover must be valid.

The `crossover` function takes as input a 2D NumPy array parents and an integer n\_pop. The function performs a genetic crossover operation on parents to generate n\_pop offspring. Use vectorized implementation if possible.

#### [Prior reflection]

\*\*Hints:\*\* Blend uniform crossover (diversity) with two-point validity checks. Use vectorized masks, ensure distinct parents via uniform selection. Design crossover to inherit valid gene blocks, avoiding empty solutions without repair. Prioritize structured diversity over random swaps.

#### [Code]

```
def crossover_v1(parents: np.ndarray, n_pop: int) -> np.ndarray:
```

```
n_parents, n_ports = parents.shape
indices = np.arange(n_ports)
# Select distinct parent pairs using vectorized operations
parent_a_idx = np.random.randint(n_parents, size=n_pop)
parent_b_idx = np.random.randint(n_parents - 1, size=n_pop)
```

```
parent_b_idx += (parent_b_idx >= parent_a_idx)
```

# Gather parent chromosomes

```
parent_a = parents[parent_a_idx]
```

parent\_b = parents[parent\_b\_idx]

# Generate two-point crossover masks

```
cross_points = np.random.randint(0, n_ports + 1, size=(n_pop, 2))
```

cross\_points.sort(axis=1)

```
mask = (indices >= cross_points[:, 0, None]) & (indices < cross_points[:, 1,
```

None])

# Combine parent genetic material

offspring = np.where(mask, parent\_b, parent\_a)

return offspring.astype(np.int32)

[Improved code]

Please write a mutated function `crossover\_v2`, according to the reflection. Output code only and enclose your code with Python code block: ```python ... ```.

# LLM Response:

```python

```
import numpy as np
```

```
def crossover_v2(parents: np.ndarray, n_pop: int) -> np.ndarray:
```

```
n_parents, n_ports = parents.shape
```

indices = np.arange(n\_ports)

# Select distinct parent pairs using vectorized operations

parent\_a\_idx = np.random.randint(n\_parents, size=n\_pop)

parent\_b\_idx = np.random.randint(n\_parents - 1, size=n\_pop)

parent\_b\_idx += (parent\_b\_idx >= parent\_a\_idx)

```
# Gather parent chromosomes
```

parent\_a = parents[parent\_a\_idx]

parent\_b = parents[parent\_b\_idx]

# Generate two-point crossover masks with sorted points

cross\_points = np.random.randint(0, n\_ports + 1, size=(n\_pop, 2))

cross\_points.sort(axis=1)

```
mask = (indices >= cross_points[:, 0, None]) & (indices < cross_points[:, 1,
None])
```

# Combine parent genetic material

offspring = np.where(mask, parent\_b, parent\_a).astype(np.int32)

# Validity repair: Ensure at least one port is selected

row\_sums = offspring.sum(axis=1)

invalid\_mask = (row\_sums == 0)

if np.any(invalid\_mask):

invalid\_indices = np.where(invalid\_mask)[0]

# Get union of parent genes for invalid offspring

repair\_candidates = parent\_a[invalid\_indices] | parent\_b[invalid\_indices]

# Vectorized repair using argmax on random mask

rand\_mask = np.random.rand(len(invalid\_indices), n\_ports)

rand\_mask = rand\_mask \* repair\_candidates # zero out invalid positions

repair\_choices = rand\_mask.argmax(axis=1)

# Apply repairs using advanced indexing

offspring[invalid\_indices, repair\_choices] = 1

return offspring

•••