

# Crossover Prompt:

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
    # Randomly select distinct parent pairs for each offspring
    parent_indices = np.array([np.random.choice(n_parents, size=2, replace=False)
    for _ in range(n_pop)])
    parent1 = parents[parent_indices[:, 0]]
    parent2 = parents[parent_indices[:, 1]]

    # Generate two random crossover points for each offspring
    cross_points = np.random.randint(0, n_decap, size=(n_pop, 2))
    cross_points.sort(axis=1)

    # Create mask for parent2 segments using broadcasting
    idx = np.arange(n_decap)
    mask = (idx >= cross_points[:, 0, None]) & (idx < cross_points[:, 1, None])

    # Combine genetic material using mask
    offspring = np.where(mask, parent2, parent1)
    return offspring
```

[Better code]

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

```

n_parents, n_decap = parents.shape

# Handle insufficient parents by duplicating if necessary
if n_parents == 0:
    return np.zeros((n_pop, n_decap), dtype=parents.dtype)
elif n_parents == 1:
    parents = np.tile(parents, (2, 1))
    n_parents = 2

# Generate parent indices for each offspring (two distinct parents per offspring)
parent_indices = np.array([np.random.choice(n_parents, 2, replace=False) for _
in range(n_pop)])
parent1_indices = parent_indices[:, 0]
parent2_indices = parent_indices[:, 1]

# Generate two random crossover points for each offspring and sort them
points = np.random.randint(0, n_decap + 1, size=(n_pop, 2))
points.sort(axis=1)

# Create mask for the middle segment between crossover points
col_indices = np.arange(n_decap)
mask = (col_indices >= points[:, 0, None]) & (col_indices < points[:, 1, None])

# Construct offspring: start with parent1 and replace middle segment with
parent2
offspring = parents[parent1_indices, :].copy()
offspring[mask] = parents[parent2_indices, :][mask]

return offspring

```

[Reflection]

Here are concise design hints based on the comparison:

1. **\*\*Handle edge cases\*\*** (0-1 parents)
2. **\*\*Use direct array copying\*\*** instead of ``np.where``
3. **\*\*Ensure crossover boundaries\*\*** include end indices (+1)
4. **\*\*Avoid broadcasting\*\*** where direct masking suffices
5. **\*\*Explicitly manage memory\*\*** with ``.copy()``

\*(18 words total)\*

[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:

```
```python

import numpy as np

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

    n_parents, n_decap = parents.shape

    # Handle edge cases: no parents or only one parent
    if n_parents == 0:

        return np.zeros((n_pop, n_decap), dtype=parents.dtype)

    if n_parents == 1:

        parents = np.vstack([parents, parents])

        n_parents = 2

    # Select distinct parent pairs for each offspring

    parent_indices = np.random.choice(n_parents, size=(n_pop, 2), replace=False)

    parent1_idx = parent_indices[:, 0]

    parent2_idx = parent_indices[:, 1]

    # Generate and sort crossover points

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

    cross_points.sort(axis=1)

    # Create mask for swapping segments
```

```

col_grid = np.arange(n_decap)

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

# Construct offspring using segments from both parents

offspring = parents[parent1_idx].copy()

offspring[mask] = parents[parent2_idx][mask]

return offspring
'''

```

## Mutation Prompt:

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]

Based on prior reflections and new insights, here are concise heuristic design hints (42 words):

**\*\*Prefer two-point crossover\*\*** to preserve contiguous port blocks and minimize structural disruption.

- \*\*Enforce distinct parent pairs\*\*** maintaining diversity.
- \*\*Handle edge cases\*\*** (0-1 parents) explicitly.
- \*\*Use vectorized segment copying\*\*** instead of per-bit operations.
- \*\*Simplify with direct array assignment\*\*** avoiding complex masking.

Key focus: Structural preservation > randomization for port selection optimization.

[Code]

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

    n_parents, n_decap = parents.shape
    if n_parents == 0:
        return np.empty((n_pop, n_decap))

    # Generate parent indices for each offspring
    idx1 = np.random.randint(0, n_parents, size=n_pop)
    if n_parents == 1:
        idx2 = np.zeros(n_pop, dtype=int)
    else:
        idx2 = np.random.randint(0, n_parents, size=n_pop)
        same = idx1 == idx2
        if np.any(same):
            offset = np.random.randint(1, n_parents, size=same.sum())
            idx2[same] = (idx2[same] + offset) % n_parents

    # Generate two crossover points for each offspring and sort them
    if n_decap > 0:
        cross_points = np.random.randint(0, n_decap, size=(n_pop, 2))
        cross_points.sort(axis=1)
        # Create mask for the segment between crossover points
        col_indices = np.arange(n_decap)
        mask = (col_indices >= cross_points[:, 0, None]) & (col_indices <
cross_points[:, 1, None])
        # Construct offspring using segments from both parents
        offspring = np.where(mask, parents[idx2], parents[idx1])
    else:
        offspring = parents[idx1]

    return offspring
```

## LLM Response:

```
```python
```

```
import numpy as np
```

```

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

    n_parents, n_decap = parents.shape

    if n_parents == 0:

        return np.empty((n_pop, n_decap))

    # Generate distinct parent pairs for each offspring

    idx1 = np.random.randint(0, n_parents, size=n_pop)

    if n_parents == 1:

        idx2 = np.zeros(n_pop, dtype=int)

    else:

        idx2 = np.random.randint(0, n_parents, size=n_pop)

        same = idx1 == idx2

        if np.any(same):

            offset = np.random.randint(1, n_parents, size=same.sum())

            idx2[same] = (idx2[same] + offset) % n_parents

    # Handle crossover when ports exist

    if n_decap > 0:

        # Generate crossover points in [0, n_decap+1] to include full segment
possibility

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

        cross_points.sort(axis=1)

    # Vectorized mask creation for segment between points

    col_indices = np.arange(n_decap)

    mask = (col_indices >= cross_points[:, 0, None]) & (col_indices <

```

```
cross_points[:, 1, None])
```

```
    # Construct offspring by combining parent segments
```

```
    offspring = np.where(mask, parents[idx2], parents[idx1])
```

```
else:
```

```
    offspring = parents[idx1]
```

```
return offspring
```

```
'''
```