

# LAB 4

Sorting networks - the experiment of Hillis

## ABSTRACT

The task is to reproduce Hillis' experiment and locate the optimal grid for sorting  $K$  numbers.

**Kareem Ghattas - 207478728**

**Hakeem Abu Shqara - 207691312**

This project represented an important exploration into the Genetic Algorithms with co-evolution for evolving sorting networks, a complex and challenging computational problem. We successfully applied our GA implementation to create efficient sorting networks for smaller values of N, such as N=6.

However, as we extended our approach to N=16, the complexity of finding the optimal number of swaps became a significant barrier. Despite our best efforts, we were unable to attain the optimal solution for this specific case within our computational constraints.

The project was a rich learning experience, enriching our understanding of coevolutionary genetic algorithms, their optimization techniques, and the intricate challenges in scaling these complex algorithms.

## *1. Representation of individuals in the population and fitness functions:*

- The individuals are represented as networks (lists of comparator pairs), where each comparator is a two-element list that represents a swap operation.

```
def create_network():  
    return [sorted(random.sample(range(N), 2)) for _ in range(random.randint(N, SWAP_MAX))]
```

- The fitness of each individual is calculated based on the total number of binary sequences it correctly sorts and the number of swaps. A penalty is applied to the fitness score if the network uses a higher number of swaps.

```
def fitness_network(network):  
    # we need to convert the network into a hashable type to use it as a key in the dictionary  
    network_tuple = tuple(tuple(x) for x in network)  
  
    if network_tuple in fitness_cache:  
        return fitness_cache[network_tuple]  
    else:  
        score = sum(1 for sequence in binary_sequences if apply_network(network, sequence) == sorted(sequence))  
        result = score - len(network) / SWAP_MAX  
        fitness_cache[network_tuple] = result  
        return result
```

## 2. Construction and improvement heuristics:

- Our initial population is randomly generated.
- Each network in the population undergoes selection, crossover, and mutation for improvement.
- The fittest individuals are selected for reproduction using tournament selection. During crossover, two parent networks combine to create two children networks, with swapping occurring at a random point in the sequence. Three types of mutation are possible: addition of a random swap, removal of a swap, or swapping two comparators in the network.

```
def crossover_network(network1, network2):
    if len(network1) < len(network2):
        network1, network2 = network2, network1
    idx = random.randint(1, len(network1) - 2)
    child1 = network1[:idx] + network2[idx:]
    child2 = network2[:idx] + network1[idx:]

    # Check if the child networks have less than MIN_SWAPS,
    # and if so, add swaps until they reach MIN_SWAPS.
    while len(child1) < MIN_SWAPS:
        swap = sorted(random.sample(range(N), 2))
        if swap not in child1:
            child1.append(swap)
    while len(child2) < MIN_SWAPS:
        swap = sorted(random.sample(range(N), 2))
        if swap not in child2:
            child2.append(swap)

    return child1, child2
```

```

def mutate_network(network):
    if random.random() < MUTATION_RATE:
        operation = random.choice(['add', 'remove', 'swap'])

        if operation == 'remove' and len(network) > MIN_SWAPS:
            # Remove a random comparator
            network.pop(random.randint(0, len(network) - 1))
        elif operation == 'add':
            # Add a random comparator
            swap = sorted(random.sample(range(N), 2))
            if swap not in network:
                network.append(swap)
        elif operation == 'swap' and len(network) > 1:
            # Swap two existing comparators
            idx1, idx2 = random.sample(range(len(network)), 2)
            network[idx1], network[idx2] = network[idx2], network[idx1]

    return network

```

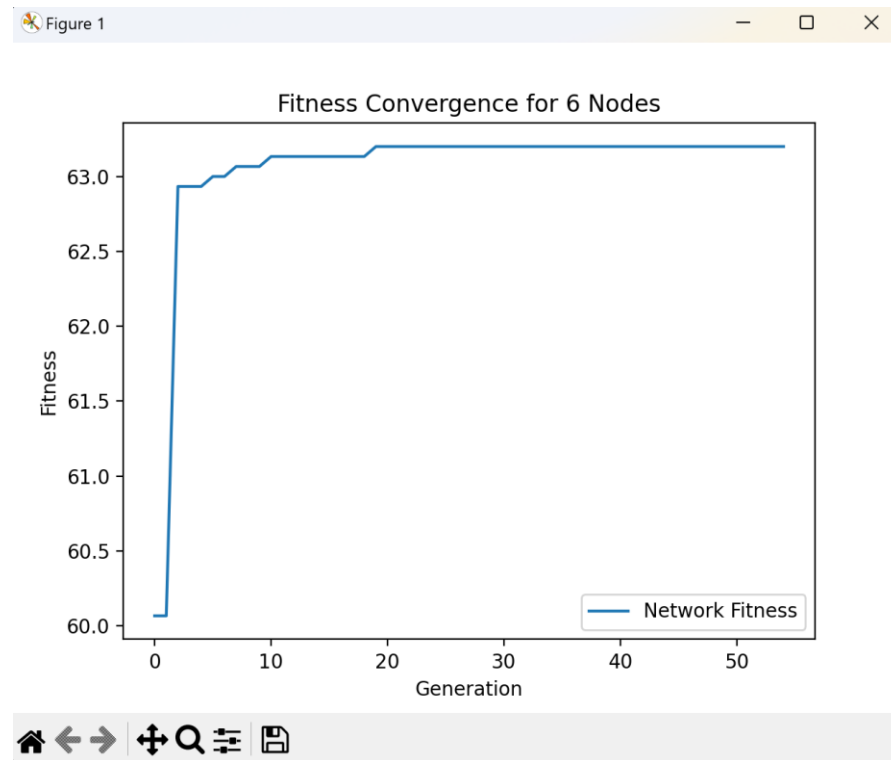
```

def tournament_selection(population, fitnesses):
    tournament = [random.choice(list(range(len(population)))) for _ in range(TOURNAMENT_SIZE)]
    return population[max(tournament, key=lambda x: fitnesses[x])]

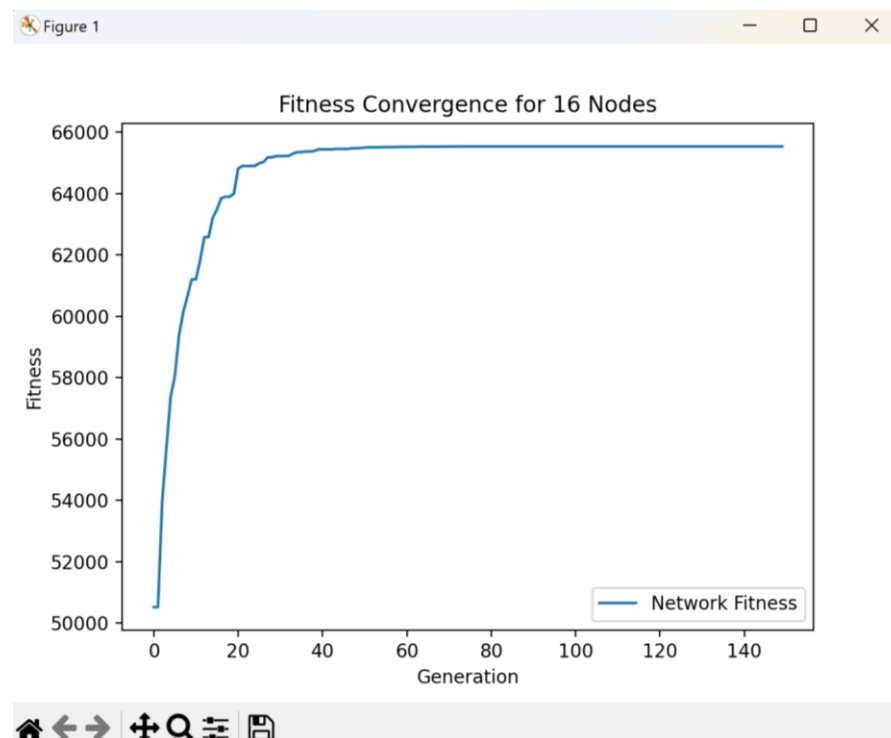
```

### 3. Convergence graph of fitness:

For  $N=6$ :



For  $N=16$ :

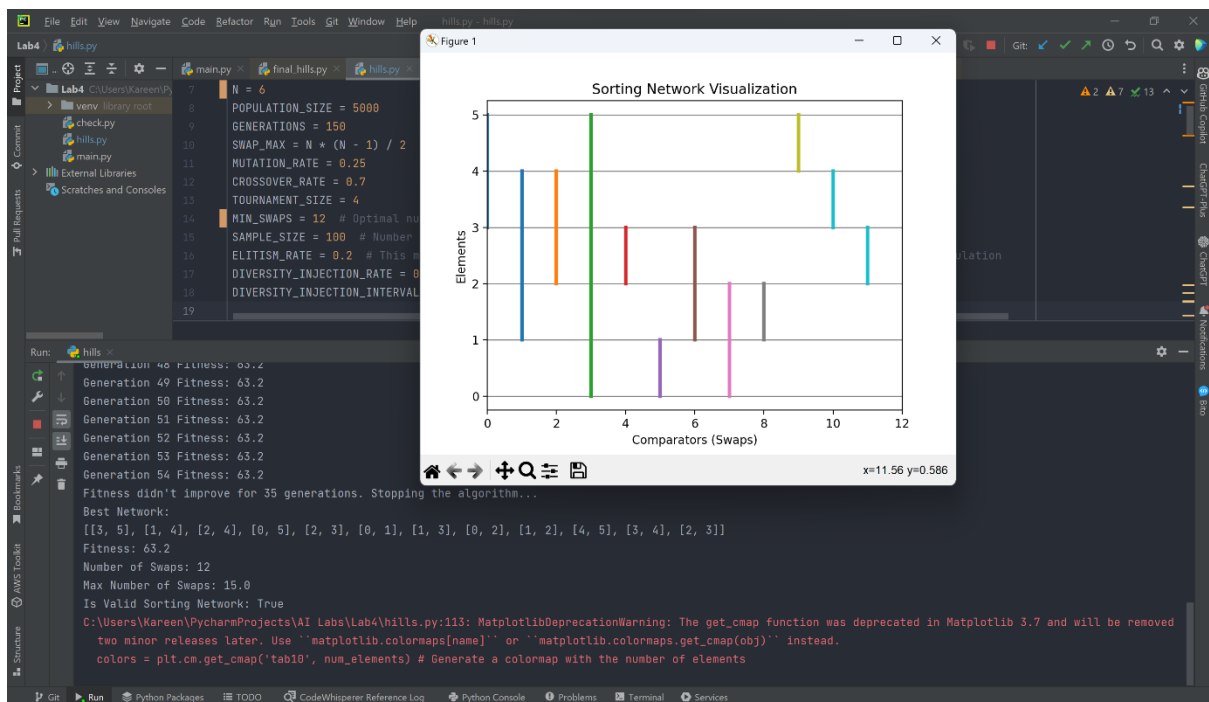
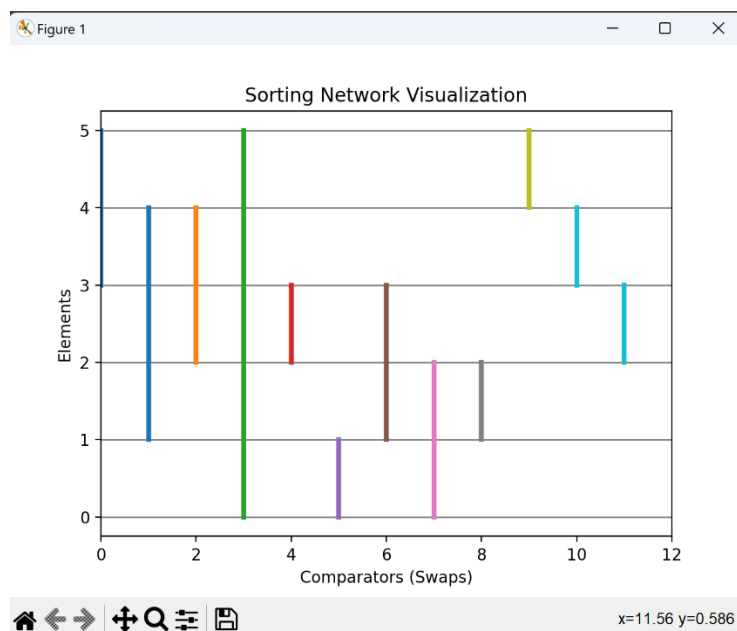


## 4. Best genes and networks:

For  $N=6$ , we were able to find the optimal sorting network that utilizes the minimum number of swaps, which is 12.

Our best sorting network is:

$[[3, 5], [1, 4], [2, 4], [0, 5], [2, 3], [0, 1], [1, 3], [0, 2], [1, 2], [4, 5], [3, 4], [2, 3]]$

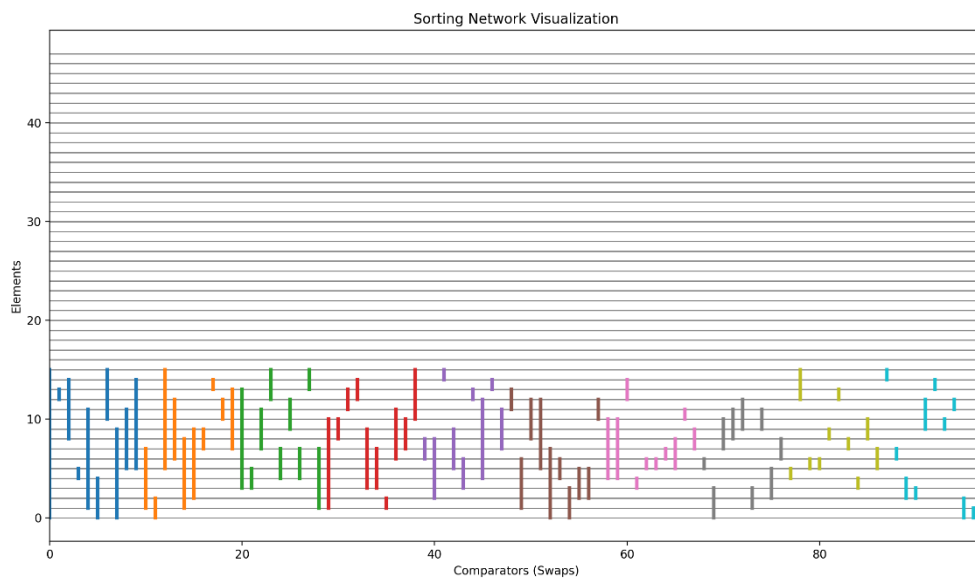


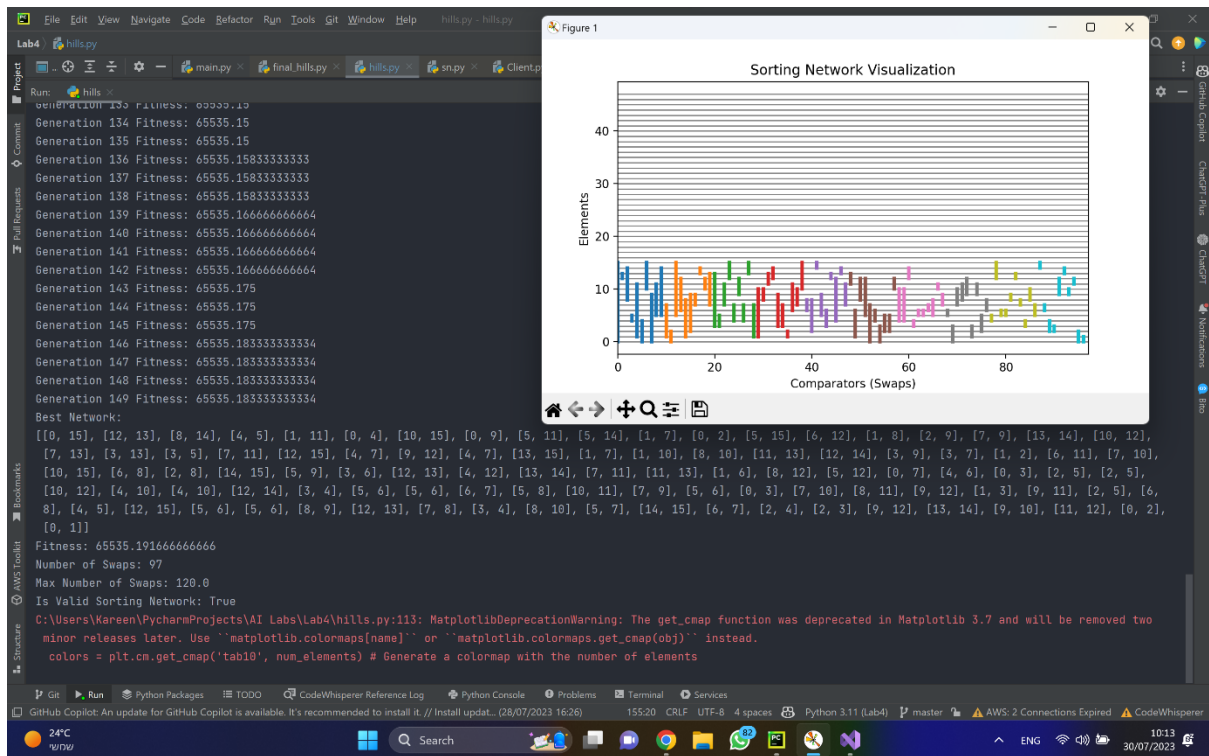
For  $N=16$ , despite extensive experimentation, the best result we achieved was a sorting network utilizing 97 swaps. Although far from the optimal 60 swaps, given our computational resources and time constraints, this was the most efficient network we were able to generate.

Our best sorting network is:

```
[[0, 15], [12, 13], [8, 14], [4, 5], [1, 11], [0, 4], [10, 15], [0, 9], [5, 11], [5, 14], [1, 7],
[0, 2], [5, 15], [6, 12], [1, 8], [2, 9], [7, 9], [13, 14], [10, 12], [7, 13], [3, 13], [3, 5],
[7, 11], [12, 15], [4, 7], [9, 12], [4, 7], [13, 15], [1, 7], [1, 10], [8, 10], [11, 13], [12,
14], [3, 9], [3, 7], [1, 2], [6, 11], [7, 10], [10, 15], [6, 8], [2, 8], [14, 15], [5, 9], [3, 6],
[12, 13], [4, 12], [13, 14], [7, 11], [11, 13], [1, 6], [8, 12], [5, 12], [0, 7], [4, 6], [0,
3], [2, 5], [2, 5], [10, 12], [4, 10], [4, 10], [12, 14], [3, 4], [5, 6], [5, 6], [6, 7], [5, 8],
[10, 11], [7, 9], [5, 6], [0, 3], [7, 10], [8, 11], [9, 12], [1, 3], [9, 11], [2, 5], [6, 8], [4,
5], [12, 15], [5, 6], [5, 6], [8, 9], [12, 13], [7, 8], [3, 4], [8, 10], [5, 7], [14, 15], [6, 7],
[2, 4], [2, 3], [9, 12], [13, 14], [9, 10], [11, 12], [0, 2], [0, 1]]
```

Figure 1



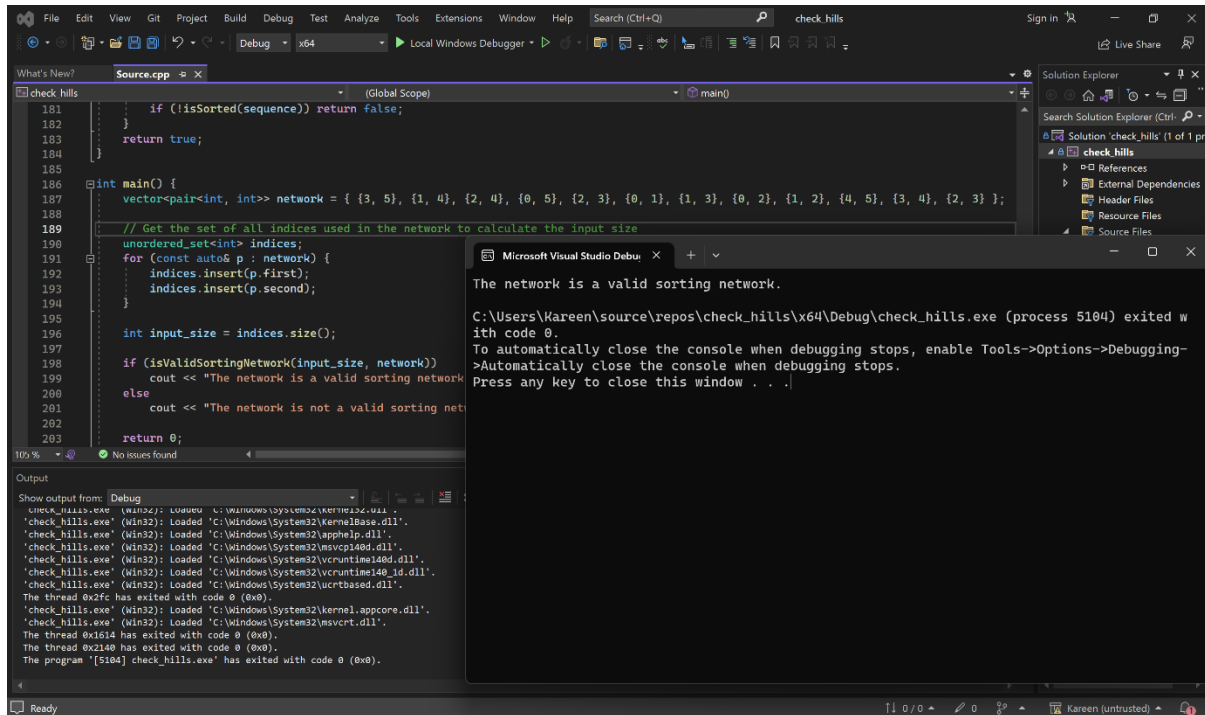




## Validation:

To validate our results and to ensure that our sorting networks are functioning correctly, we developed a C++ validation script. This script generates all binary sequences of a given length, sorts them using the sorting network, and checks whether the resulting sequence is sorted.

*N=6:*

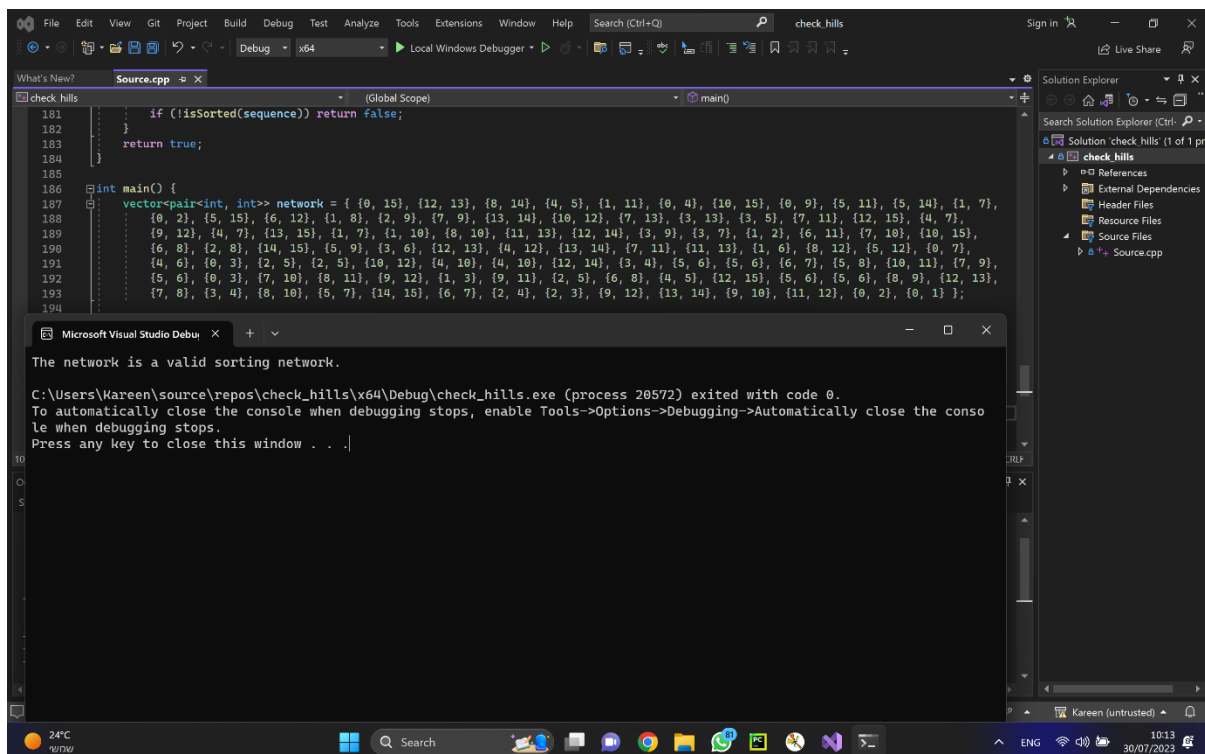


```
181     if (!isSorted(sequence)) return false;
182 }
183 return true;
184 }
185
186 int main() {
187     vector<pair<int, int>> network = { {3, 5}, {1, 4}, {2, 4}, {0, 5}, {2, 3}, {0, 1}, {1, 3}, {0, 2}, {1, 2}, {4, 5}, {3, 4}, {2, 3} };
188
189     // Get the set of all indices used in the network to calculate the input size
190     unordered_set<int> indices;
191     for (const auto& p : network) {
192         indices.insert(p.first);
193         indices.insert(p.second);
194     }
195
196     int input_size = indices.size();
197
198     if (isValidSortingNetwork(input_size, network))
199         cout << "The network is a valid sorting network\n";
200     else
201         cout << "The network is not a valid sorting network\n";
202
203     return 0;
204 }
```

Microsoft Visual Studio Debug Console Output:

```
The network is a valid sorting network.
C:\Users\Kareem\source\repos\check_hills\x64\Debug\check_hills.exe (process 5104) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

*N=16:*



```
181     if (!isSorted(sequence)) return false;
182 }
183 return true;
184 }
185
186 int main() {
187     vector<pair<int, int>> network = { {0, 15}, {12, 13}, {8, 14}, {4, 5}, {1, 11}, {0, 4}, {10, 15}, {0, 9}, {5, 11}, {5, 14}, {1, 7},
188     {0, 2}, {5, 15}, {6, 12}, {1, 8}, {2, 9}, {13, 14}, {10, 12}, {7, 13}, {3, 13}, {3, 5}, {7, 11}, {12, 15}, {4, 7},
189     {9, 12}, {4, 7}, {13, 15}, {1, 7}, {1, 10}, {8, 10}, {11, 13}, {12, 14}, {3, 9}, {3, 7}, {1, 2}, {6, 11}, {7, 10}, {10, 15},
190     {6, 8}, {2, 8}, {14, 15}, {5, 9}, {2, 6}, {12, 13}, {4, 12}, {13, 14}, {7, 11}, {11, 13}, {1, 6}, {8, 12}, {5, 12}, {0, 7},
191     {4, 6}, {0, 3}, {2, 5}, {2, 5}, {10, 12}, {4, 10}, {4, 10}, {12, 14}, {3, 4}, {5, 6}, {5, 6}, {6, 7}, {5, 8}, {10, 11}, {7, 9},
192     {5, 6}, {0, 3}, {7, 10}, {8, 11}, {9, 12}, {1, 3}, {9, 11}, {2, 5}, {6, 8}, {4, 5}, {12, 15}, {5, 6}, {5, 6}, {8, 9}, {12, 13},
193     {7, 8}, {3, 4}, {8, 10}, {5, 7}, {14, 15}, {6, 7}, {2, 4}, {2, 3}, {9, 12}, {13, 14}, {9, 10}, {11, 12}, {0, 2}, {0, 1} };
194 }
```

Microsoft Visual Studio Debug Console Output:

```
The network is a valid sorting network.
C:\Users\Kareem\source\repos\check_hills\x64\Debug\check_hills.exe (process 20572) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

## 5. Choosing the Populations and Parasites:

In the developed co-evolutionary algorithm, two populations were selected: Sorting Networks and Test Sequences. The Sorting Networks are the primary population, while the Test Sequences act as the parasites, challenging the primary population. Here is a detailed explanation of how these entities are utilized in the evolutionary algorithm:

### Initialization:

The algorithm begins by initializing two distinct populations: the Sorting Networks (**network\_population**) and the Test Sequences (**test\_seq\_population**). Each Sorting Network is an individual solution for the sorting problem, while each Test Sequence is a unique problem instance or challenge for the Sorting Networks.

```
network_population = [create_network() for _ in range(POPULATION_SIZE)]
test_seq_population = [random.sample(range(N), N) for _ in range(POPULATION_SIZE)]
```

### Fitness Evaluation:

The fitness evaluation for the Sorting Networks is determined based on how effectively they sort the Test Sequences. Each network's performance is assessed using the **fitness\_network** function, which tests its ability to sort the provided test sequences. In contrast, the **fitness\_test\_sequences** function is used to evaluate the fitness of each Test Sequence, determined by the average performance of the networks against that particular sequence.

```
def fitness_network(network):
    # we need to convert the network into a hashable type to use it as a key in the dictionary
    network_tuple = tuple(tuple(x) for x in network)

    if network_tuple in fitness_cache:
        return fitness_cache[network_tuple]
    else:
        score = sum(1 for sequence in binary_sequences if apply_network(network, sequence) == sorted(sequence))
        result = score - len(network) / SWAP_MAX
        fitness_cache[network_tuple] = result
        return result
```

```
def fitness_test_sequences(test_seq, network_population):
    """Calculate the fitness of a test sequence by using Monte Carlo sampling."""
    sampled_networks = random.sample(network_population, min(SAMPLE_SIZE, len(network_population)))
    return sum(1 for network in sampled_networks if apply_network(network, test_seq) == sorted(test_seq))
```

### Selection Process:

The **tournament\_selection** function is utilized for choosing parents from both populations for the generation of offspring. The selection mechanism is such that it rewards the fittest individuals: the best Sorting Networks and Test Sequences from a randomly chosen sample (the tournament) are selected and become parents for the next generation.

### Crossover and Mutation:

In the evolutionary algorithm, the generation of new offspring for both populations is achieved by copying and altering the parents. The **crossover\_network** and **mutate\_network** functions perform

these operations for the Sorting Network population. For the Test Sequences, fresh random sequences are generated, introducing new problem instances for the networks to tackle.

#### *Elitism:*

A predefined constant **ELITISM\_RATE** has been set to ensure the survival of a certain percentage of the best-performing individuals from both populations. By carrying the elites forward to the next generation, the algorithm ensures that the overall performance does not decrease over time.

#### *Diversity Injection:*

Finally, to maintain diversity within the Sorting Network population and avoid premature convergence to suboptimal solutions, an **inject\_diversity** function is employed. This function infuses new, randomly generated networks into the population at regular intervals.

```
def inject_diversity(network_population, generation):
    if generation % DIVERSITY_INJECTION_INTERVAL == 0:
        # Introduce new random individuals
        num_new_individuals = int(DIVERSITY_INJECTION_RATE * len(network_population))
        network_population[-num_new_individuals:] = [create_network() for _ in range(num_new_individuals)]
    return network_population
```

Through the above-mentioned processes, the Sorting Networks and Test Sequences are co-evolved. While the Sorting Networks are evolving to better sort the Test Sequences, the Test Sequences are evolving to pose increasingly challenging sorting problems for the networks. This dynamic creates a robust and effective co-evolutionary system.

## *6. Usage of Bitonic Networks:*

We initially used Bitonic Networks, however, after extensive testing, we found our algorithm performed better without the constraints of Bitonic networks. Despite their deterministic structure, they didn't enhance our problem-solving capabilities and limited diversity in our solutions.

## 7. Adaptation of the genetic engine:

The genetic algorithm was tailored for the problem with specific selection, survival, and genetic operators:

- **Selection:** We used a tournament selection method.
- **Survival:** Elitism was used to ensure the survival of the fittest individuals.
- **Genetic operators:** Crossover and mutation operators were defined specifically for our representation of sorting networks.
- **Parameters of evolution:** The parameters of the genetic algorithm, such as population size, mutation rate, crossover rate, and number of generations, were set based on preliminary testing and tuning.

```
N = 6
POPULATION_SIZE = 5000
GENERATIONS = 150
SWAP_MAX = N * (N - 1) / 2 # Maximum possible swaps in a network
MUTATION_RATE = 0.25
CROSSOVER_RATE = 0.7
TOURNAMENT_SIZE = 4 # The number of individuals participating in each tournament selection.
MIN_SWAPS = 12 # Optimal number of swaps
SAMPLE_SIZE = 100 # Number of sequences to sample
ELITISM_RATE = 0.2 # The percentage of the population that automatically passes to the next generation.
# Parameters that control the introduction of new random individuals to maintain diversity in the population.
DIVERSITY_INJECTION_RATE = 0.2 # Introduce 20% new random individuals every few generations
DIVERSITY_INJECTION_INTERVAL = 15 # Introduce new random individuals every 15 generations
```

## 8. Convergence Problems and Mitigation Strategies

### a. Disconnection Effect

The disconnection effect refers to the scenario where a population splits into several subpopulations, each converging to a different solution. This can often result in sub-optimal solutions as some subpopulations may converge to local optima.

In our implementation, we addressed this by maintaining a large population size and ensuring enough mixing of genes through crossover and mutation. Also, the tournament selection method helped prevent premature convergence by preserving a diverse set of individuals for crossover.

### b. Circularity and Return of Trivial Solutions

The return of trivial solutions, where the algorithm repeatedly evolves and discards the same sub-optimal solutions.

To avoid this, we implemented a technique known as "diversity injection." Every **DIVERSITY\_INJECTION\_INTERVAL** generations, we introduced new random individuals into the population at a rate defined by **DIVERSITY\_INJECTION\_RATE**. This helped maintain genetic diversity, discouraging the algorithm from getting stuck in cycles of trivial solutions.

### c. System's Forgetfulness Effect and Over-fitness

The system's forgetfulness effect occurs when beneficial traits from early generations are lost over time.

To tackle these issues, we implemented an elitism mechanism. A certain percentage (**ELITISM\_RATE**) of the fittest individuals from each generation automatically survive to the next one. This prevents the loss of beneficial traits, ensuring they remain in the gene pool.

Over-fitness, on the other hand, happens when the algorithm overly optimizes for a specific dataset, reducing its general performance.

To avoid over-fitness, we ensured that our fitness function was well-suited to the problem at hand and not overly specific to any particular instance. This, along with maintaining a diverse population and using a good genetic operators, helped us produce sorting networks that are effective across various inputs.

