Tarea 4.

Cómputo Evolutivo.

Luis Andrés Eguiarte Morett.

3 de mayo de 2021

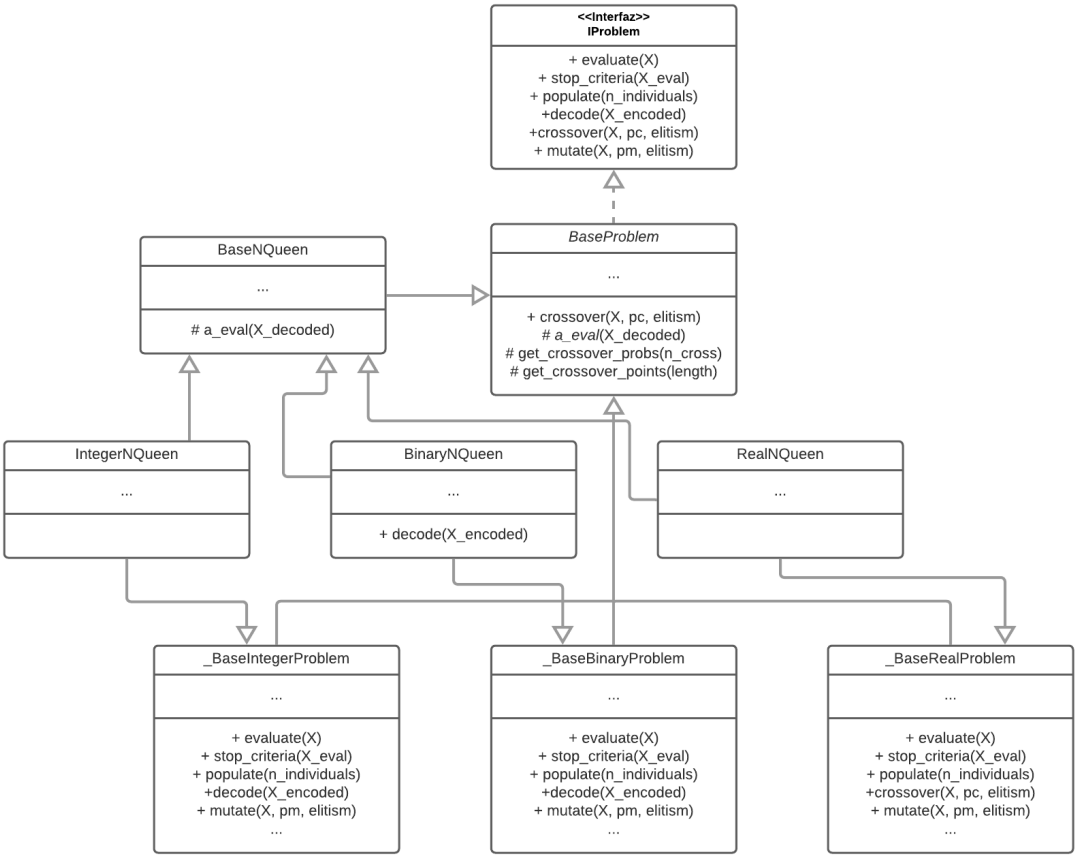
1. Desarrollo del programa.

1.1. Problemas.

Se trabajó sobre el programa de la tarea anterior. Para poder utilizarlo, se abstrayeron por completo los operadores de cruza y mutación a la definción de los problemas, puesto que estos dos operadores son completamente dependientes de la codificación elegida para el problema a resolver, e inclusive si el problema lo amerita, puede requerir sus propios operadores especializados (i.e. TSP). También se resolvieron algunos errores evidentes (a partir de los gráficos) de implementación en la tarea anterior, uno de ellos en el operador de cruza binaria y el otro en selección proporcional por ruleta. La interfaz IProblem para un problema se define ahora como:

1. evaluate: Evalua las soluciones potenciales del problema.
2. stop_criteria: Regresa si la población ha llegado al criterio de paro.
3. populate: Crea una poblacion inicial de posibles soluciones.
4. decode: Pasa a la población del genotipo al fenotipo.
5. crossover: Efectúa la cruza con los elementos de la población.
6. mutate: Efectúa la mutación con los elementos de la población.

Con ello se logra un desacoplamiento total entre el problema a resolver y su codificación y el algoritmo genético con cualquiera de los métodos de selección ya implementados, siempre y cuando se cuide que la función de aptitud para la selección proporcional sea positiva siempre.
Se crea una jerarquía de clases para abstraer problemas en general, en un siguiente nivel de abstracción, se define un problema con una cruza de un punto implementada por defecto pero extendible o modificable por alguna clase hija, y después se definen las tres codificaciones (entera, real o binaria), donde para la codificación entera y binaria se reutiliza el operador de cruza de la clase padre, y en la codificación real se implementa el operador de cruza propio. Dicha estructura se ilustra en el siguiente diagrama de clases:



En cuanto a las clases *IntegerNQueen*, *RealNQueen* y *BinaryNQueen*, cada una es una instancia del problema de las N-Reinas, que heredan de la clase *BaseNQueen* el método de evaluación de población $a\_eval(X\_decoded)$, el cuál contiene toda la lógica para evaluar fenotipos del problema de las N-Reinas representados por vectores de números enteros al contar el número de conflictos existentes con la configuración descrita por dicho vector, cada entrada i-ésima representa el renglón que ocupa una reina en la i-ésima columna del tablero, esta representación elimina automáticamente todas aquellas soluciones que tendrían conflictos debido a que hubiesen reinas en una misma columna. Las clases *IntegerNQueen*, *RealNQueen* y *BinaryNQueen*, cada una heredan de sus correspondientes clases más generales de problemas correspondiente a cada codificación, *_BaseIntegerProblem*, *_BaseRealProblem* y *_BaseBinaryProblem*.

1.2. Codificaciones.

Cada una de las codificaciones están implementadas de manera general para cualquier problema, mientras que los problemas específicos son libres de implementar, modificar o extender cualquiera de los métodos de las clases más generales correspondientes a la codificación que usen.

1.2.1. Codificación entera.

Para este caso lo único que solo se requirió implementar el operador de mutación, esto porque la cruza de un punto tan solo requiere de un punto de cruza aleatorio y con base en este se da el intercambio elemento a elemento para producir a los dos hijos, mientras que para la mutación, se tiene que aplicar la probabilidad de mutación a cada elemento de la población para decidir si se muta o no y luego intercambiar dos genes de manera aleatoria, lo cual es fundamentalmente distinto a la mutación para el caso binario, donde se generaba una matriz de números aleatorios y con base en ella y la probabilidad de mutación p_m, se decidía elemento a elemento cual mutar al invertir el bit, es decir, la mutación es sobre los genes de toda la población.

```
class _BaseIntegerProblem(BaseProblem):
    def __init__(self, thresh, n_dim = 2):
        self.n_dim = n_dim
        self.thresh = thresh

    def evaluate(self, X):
        ...

    def stop_criteria(self, X_eval):
        ...

    def populate(self, n_individuals):
        ...

    def decode(self, X_encoded):
        ...

    def get_mutation(self, shape):
        ...

    def mutate(self, X, pm, elitism):
        mutate_m = self.get_mutation((X.shape[0], 1))
        mutate_m = mutate_m <= pm
        if not elitism:
            for i, m  in enumerate(mutate_m):
                if m:
                    indices = np.random.permutation(X.shape[1])[0 : 2]
                    X[i,indices[0]], X[i, indices[1]] =
                    X[i, indices[1]], X[i, indices[0]]
        else:
            elitism_num = math.floor(elitism * X.shape[0])
            for i in range(elitism_num, X.shape[0]):
                if mutate_m[i]:
```

```
                indices = np.random.permutation(X.shape[1])[0 : 2]
                X[i,indices[0]], X[i, indices[1]] =
                X[i, indices[1]], X[i, indices[0]]
        return X
```

Para el caso del problema de las N-reinas, el genotipo es igual al fenotipo con esta codificación.

1.2.2. Codificación real.

Para la codificación real, para el caso del problema de las N-reinas, se utilizó un rango fijo de valores reales entre 0 y 5 para el genotipo. Para la decodificación al fenotipo, se utilizó el esquema de ordenamiento del genotipo del individuo, tomar como quedaron los índices originales después de ser ordenado, y esto es el fenotipo, esto tiene el efecto de que cada individuo sea una permutación de los renglones ocupados por las n reinas, y a su vez esto provoca que se eliminen todas las soluciones con conflictos debidos a estar en los mismos renglones, es decir se reduce el espacio de búsqueda.
La cruza utilizada fue intermedia, descrita por:

$$z_i = x_i + \alpha_i(y_i - x_i)$$

La mutación se aplicó de la siguiente manera:

$$p_m \in [0, 1/n]$$

Donde n es el número de variables (genes).
Uno de los valores entre $[-rango_i, rango_i]$ se suma a la variable seleccionada, donde $rango_i$ es el rango de mutación de la variable i-ésima.

$$rango_i = 0.1(intervalodebusqueda)$$

Donde el intervalo de búsqueda es el rango de valores que puede tomar la variable.
De esta forma la mutación de un gen i-ésimo es:

$$z_i = x_i \pm rango_i\delta$$

Se elige $\pm$ con probabilidad 0.5.

$$\delta = \sum_{i=0}^{n-1} \alpha_i 2^{-i}, \alpha_i \in 0, 1$$

Cada α_i se inicializa en 0, posteriormente, se muta con probabilidad $p_\delta = 1/n$, a $\alpha_i = 1$.
Lo anterior queda expresado de la siguiente forma en el código:

```
class _BaseRealProblem(BaseProblem):
    def __init__(self, thresh, n_dim = 2):
        self.n_dim = n_dim
        self.thresh = thresh
```

```
def evaluate(self, X):
    decoded_rep = self.decode(X)
    #print(decoded_rep)
    #X_eval = np.array(list(zip(1./(1. + 10.*self.n_dim + np.sum(decoded_rep**2 - 10
    X_eval = self.a_eval(decoded_rep)
    return X_eval


def stop_criteria(self, X_eval):
    return list(np.where(X_eval >= self.thresh)[0])

def populate(self, n_individuals):
    return np.random.uniform(0, 5.1, size = (n_individuals, self.n_dim))

def decode(self, X_encoded):
    X_decoded = np.zeros(X_encoded.shape, dtype=np.int64)
    for i, x in enumerate(X_encoded):
        indexed = np.array(list(zip(x, list(range(X_decoded.shape[1])))),
        dtype = [('real_rep', float),('index', int)])
        indexed = np.sort(indexed, order=["real_rep"])
        X_decoded[i, :] = indexed["index"]
    return X_decoded

def get_crossover_points(self, length):
    return np.random.uniform(low = -.25 , high = 1.25, size = length)

def crossover(self, X, pc, elitism):
    if not elitism:
        n_cross = X.shape[0] // 2
        elitism_num = 0
    else:
        elitism_num = math.floor(elitism * X.shape[0])
        n_cross = (X.shape[0] - elitism_num) // 2
    prob_cross = self.get_crossover_probs(n_cross)
    for i, p in enumerate(prob_cross):
        if p <= pc:
            alphas = self.get_crossover_points(X.shape[1])
            X[2*i + elitism_num,:] += alphas * (X[2*i + 1 + elitism_num, :]
            - X[2*i + elitism_num,:])
            X[2*i + 1 + elitism_num,:] += alphas * (X[2*i + elitism_num,:]
            - X[2*i + 1 + elitism_num, :])
    return X

def get_mutation(self, shape):
```

```
        return np.random.uniform(size = shape)

    def mutate(self, X, pm, elitism):
        if not elitism:
            elitism = 0

        rang = 5.*.1
        mutate_m = self.get_mutation((X.shape[0], X.shape[1]))

        mutate_plus_minus = self.get_mutation((X.shape[0], X.shape[1]))

        mutate_m[mutate_m <= pm] = 1.
        mutate_m[mutate_m < 1.] = 0.
        mutate_plus_minus[mutate_plus_minus <= .5] = 1.0
        mutate_plus_minus[mutate_plus_minus > .5] = -1.0

        elitism_num = math.floor(elitism * X.shape[0])
        for i in range(elitism_num, X.shape[0]):
            mutate_delta = self.get_mutation((X.shape[1], X.shape[1]))
            mutate_delta[mutate_delta <= 1./self.n_dim] = 1.
            mutate_delta[mutate_delta < 1.] = 0.
            deltas = (mutate_delta @
            (2**-np.arange(self.n_dim, dtype = np.float64)[:, np.newaxis])).T
            X[i, :] = X[i, :] + mutate_m[i, :] *
            mutate_plus_minus[i, :] * rang * deltas

        return X
```

1.2.3. Codificación binaria.

En cuanto a la codificación binaria, se utiliza tal cual se definió para la tarea anterior, la cruza es tal como se describió para la codificación real y la mutación es por gen, obteniendo una matriz de valores aleatorios con dimensión igual a la matriz de población y mutando (invirtiendo el bit) todo aquel elemento de la matriz cuyo valor sea menor o igual que la probabilidad de mutación.
Para poder decodificar los valores representados en el genotipo se utiliza la misma decodificación que se utilizó para la tarea pasada, pero en este caso se redondean dichos valores para así obtener los fenotipos enteros representativos de las soluciones del problema.

```
class BinaryNQueen(_BaseBinaryProblem, BaseNQueen):
    def __init__(self, n_dim = 2, n_prec = 4):
        super().__init__(0, (0.01, n_dim), n_dim = n_dim, n_prec=n_prec)

    def decode(self, X_encoded):
```

```
        return np.ceil(super().decode(X_encoded)).astype(int) - 1
```

2. Resultados.

Para todos los siguientes experimentos se siguieron los siguientes parámetros de cruza, mutación y número de individuos y se efectuaron 5 corridas por cada instancia de las n-reinas, donde $n \in [12, 16, 20, 24, 28, 32]$, a lo largo de esas 5 corridas, para cada una de las codificaciones, se guardaron las aptitudes promedio por generación y la aptitud del mejor individuo por generación, para así promediar esos datos y graficar hasta el número de generaciones promedio, esto de nuevo, por cada una de las codificaciones, de tal suerte que se tienen 3 gráficas por cada instancia del problema de las n-reinas.

2.1. Parámetros

$p_c = 0.9$
$n\_individuals = 500$
$elitism = 0.1$
$selection = tournament$

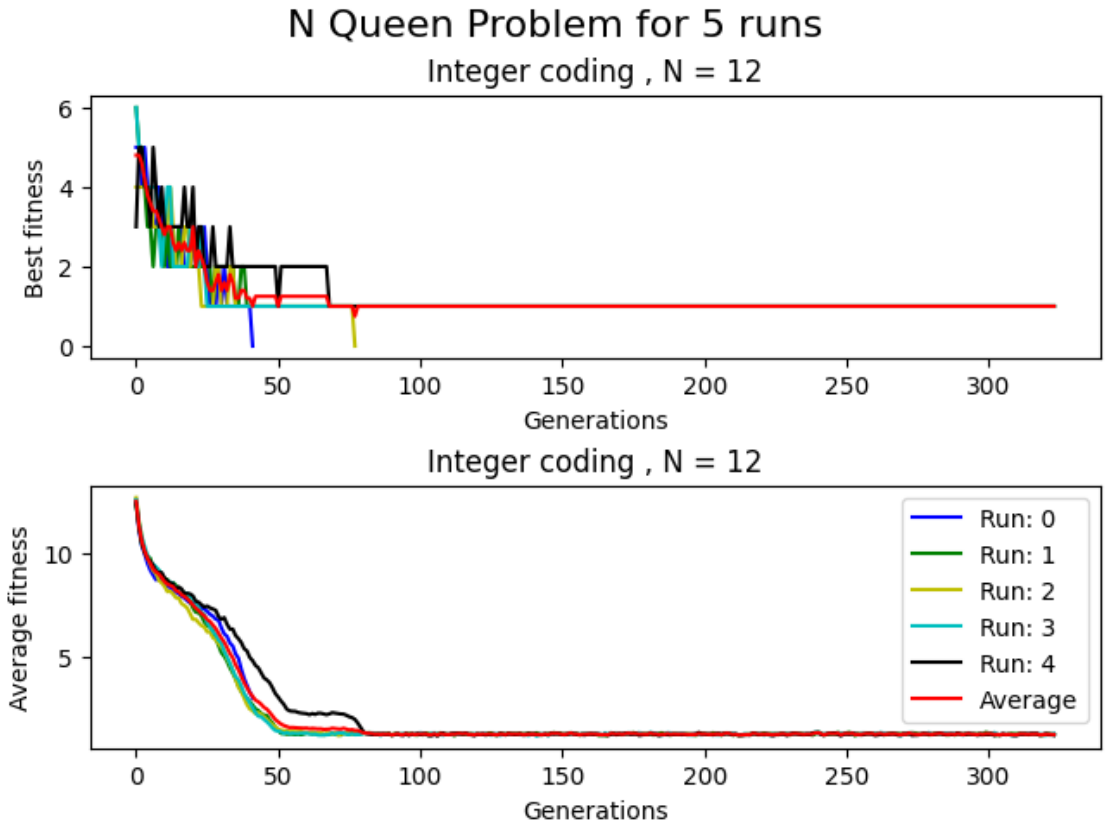
Codificación entera.
$max\_iter = 500$
$p_m = 1/n$

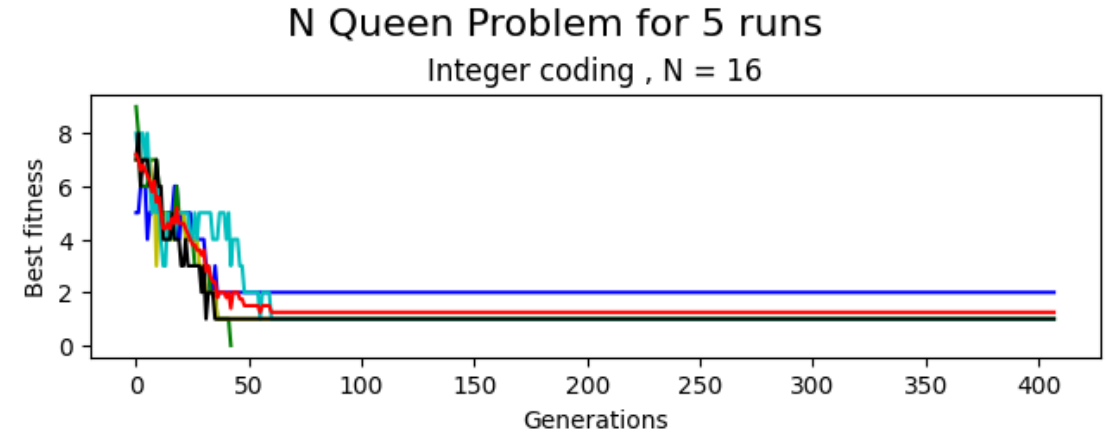
Codificación real.
$max\_iter = 500$
$p_m = 1/n$
Donde n corresponde al número de reinas.

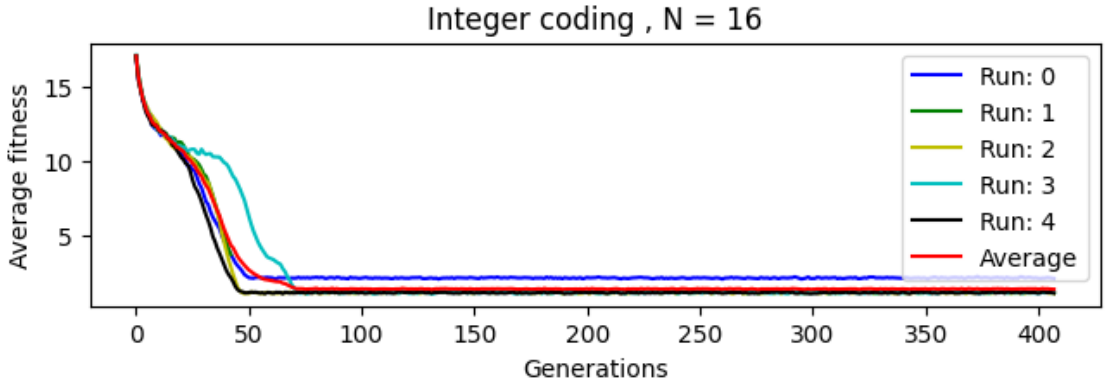
Codificación binaria.
$max\_iter = 1000$
$p_m = 1/l$
$n\_prec = 4$
Donde $l = n(\lceil \log_2(n - 0.01) \cdot 10^{n\_prec} \rceil)$
Y n_prec es el número de decimales de precisión.

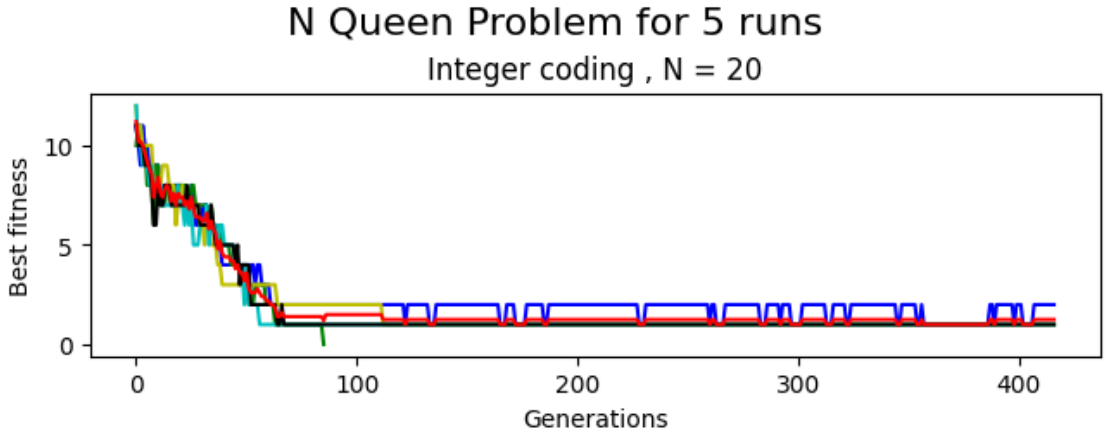
2.2. Aptitudes de cada codificación.

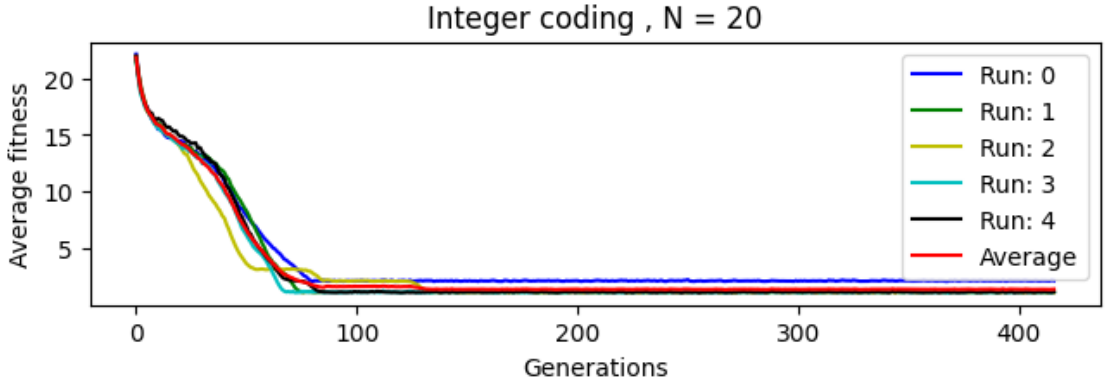
2.2.1. Entera

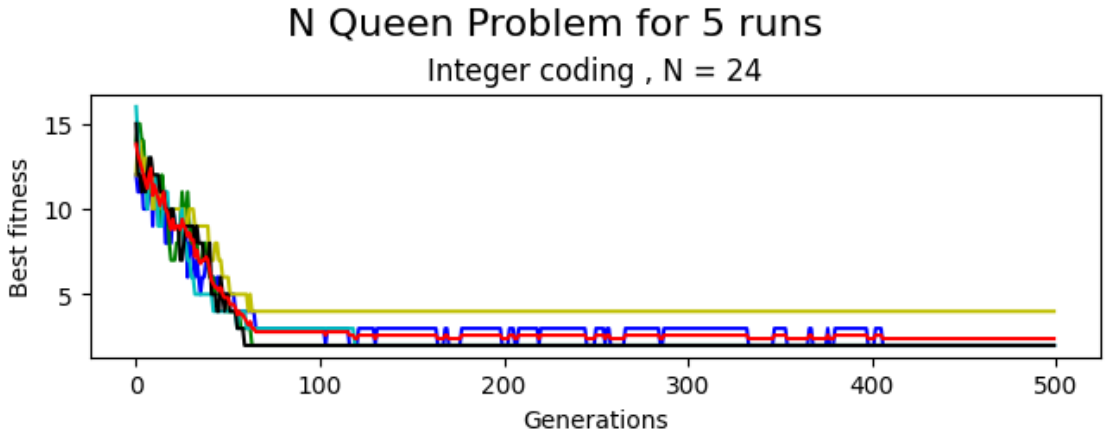


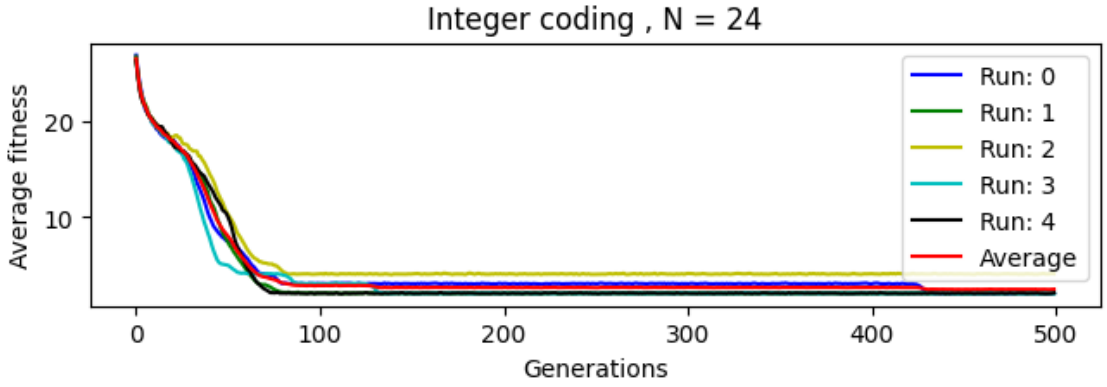

N Queen Problem for 5 runs
Integer coding , N = 16
Best fitness
Generations

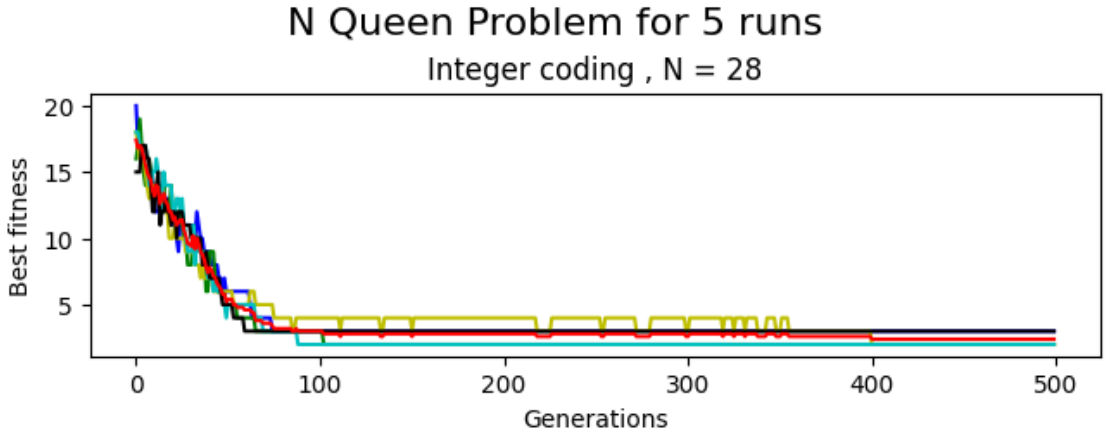

Integer coding , N = 16
Average fitness
Generations
Run: 0
Run: 1
Run: 2
Run: 3
Run: 4
Average

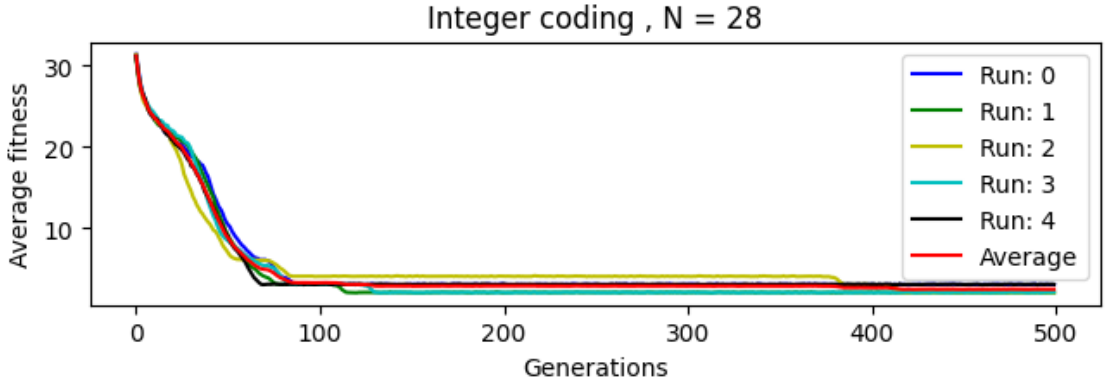

N Queen Problem for 5 runs
Integer coding , N = 20
Best fitness
Generations

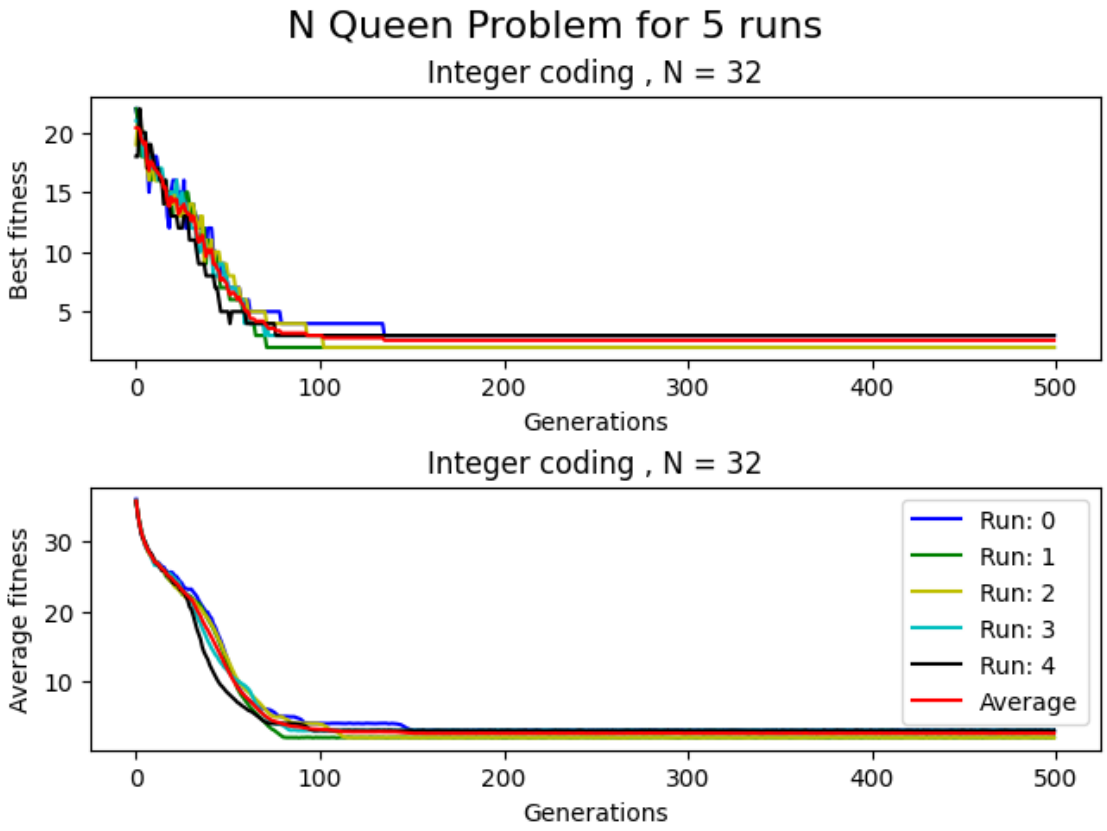

Integer coding , N = 20
Average fitness
Generations
Run: 0
Run: 1
Run: 2
Run: 3
Run: 4
Average


N Queen Problem for 5 runs
Integer coding , N = 24
Best fitness
Generations

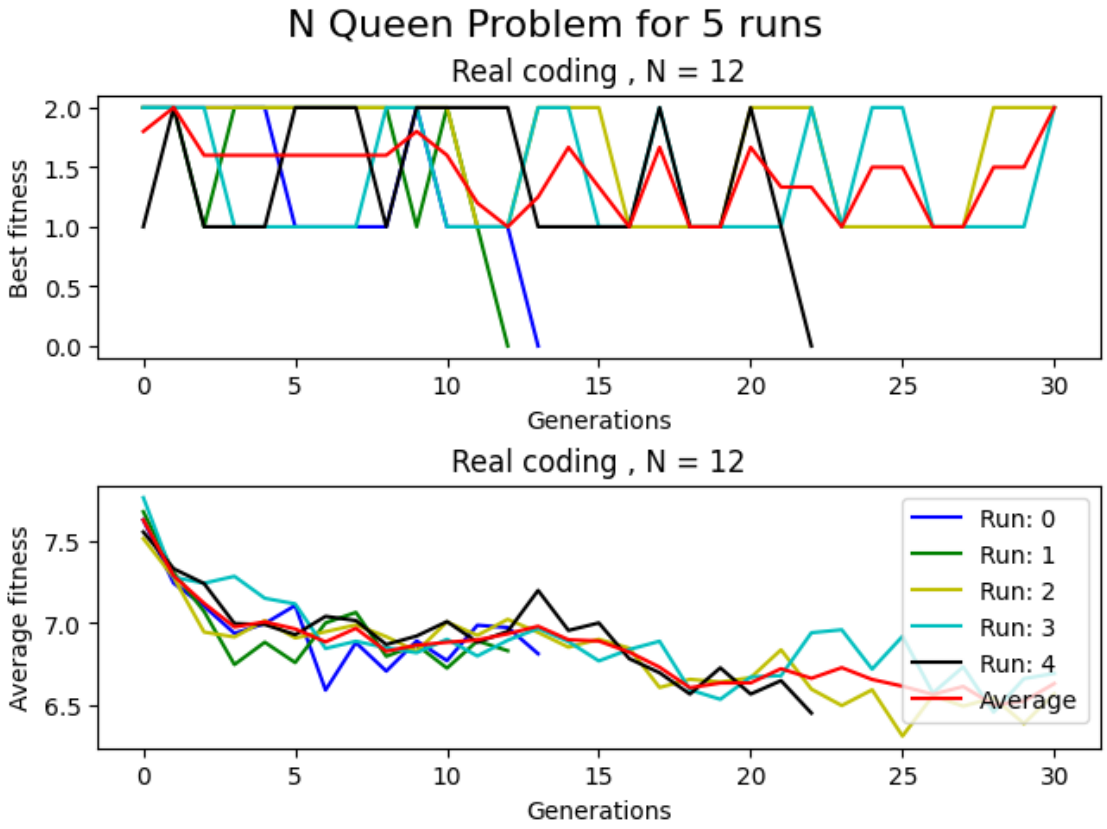

Integer coding , N = 24
Average fitness
Generations
Run: 0
Run: 1
Run: 2
Run: 3
Run: 4
Average

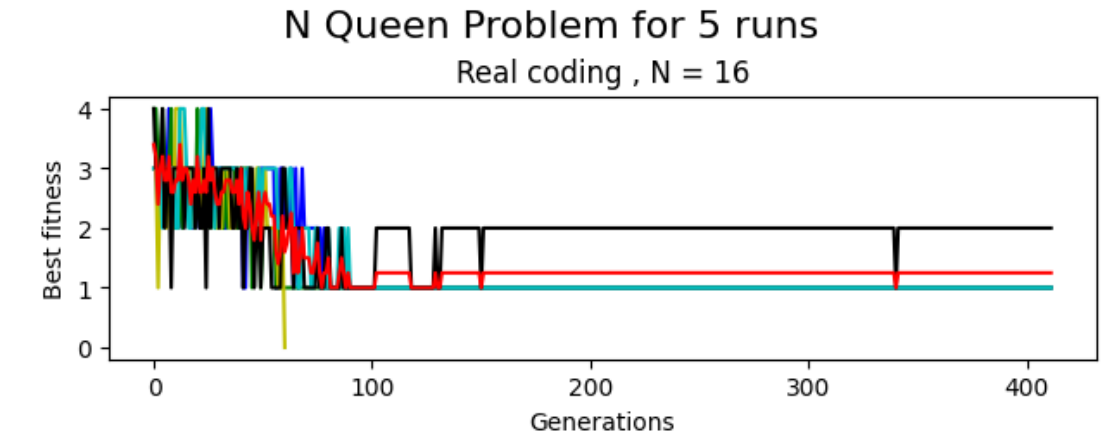

N Queen Problem for 5 runs
Integer coding , N = 28
Best fitness
Generations

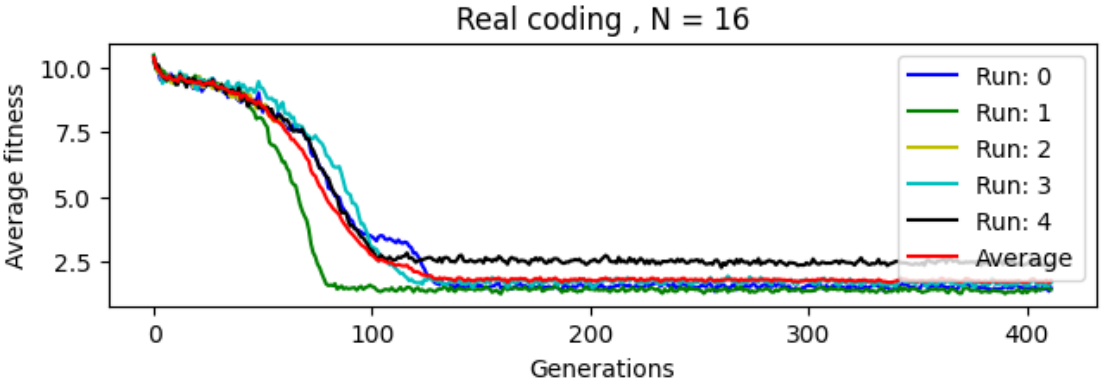

Integer coding , N = 28
Average fitness
Generations
Run: 0
Run: 1
Run: 2
Run: 3
Run: 4
Average

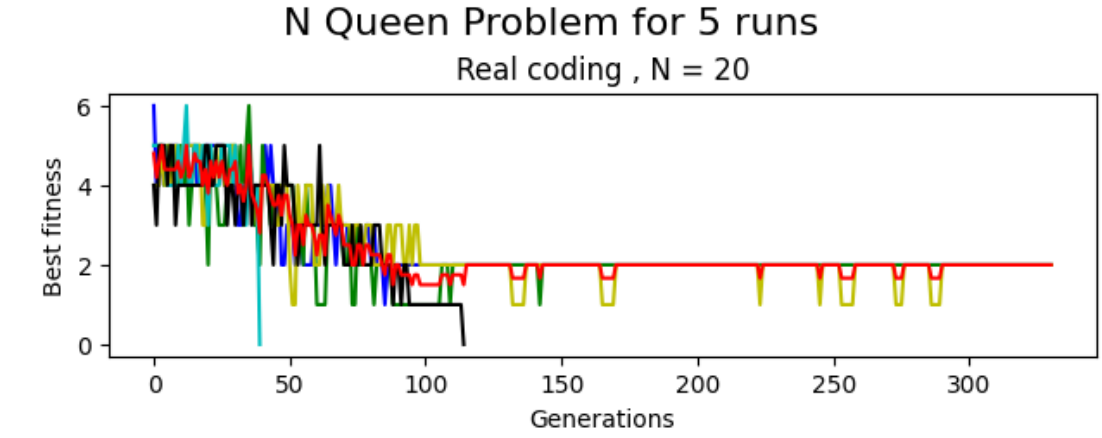

N Queen Problem for 5 runs
Integer coding , N = 32
Best fitness
Generations
Integer coding , N = 32
Average fitness
Generations
Run: 0
Run: 1
Run: 2
Run: 3
Run: 4
Average

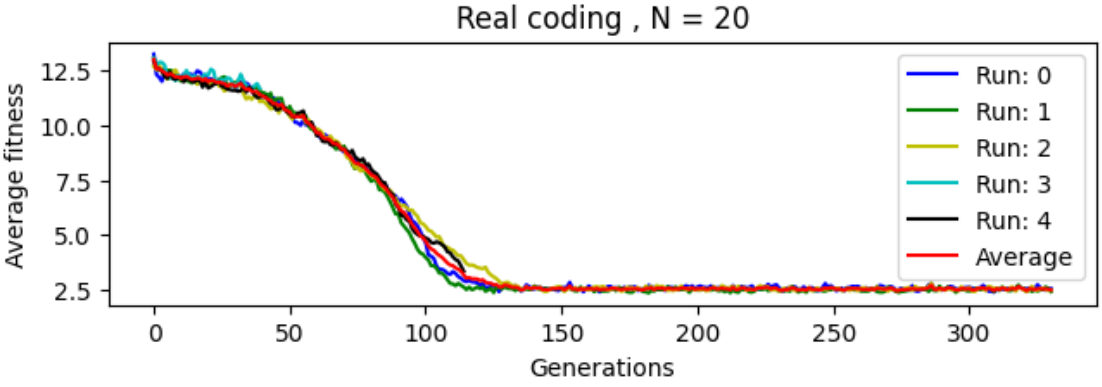
2.2.2. Real

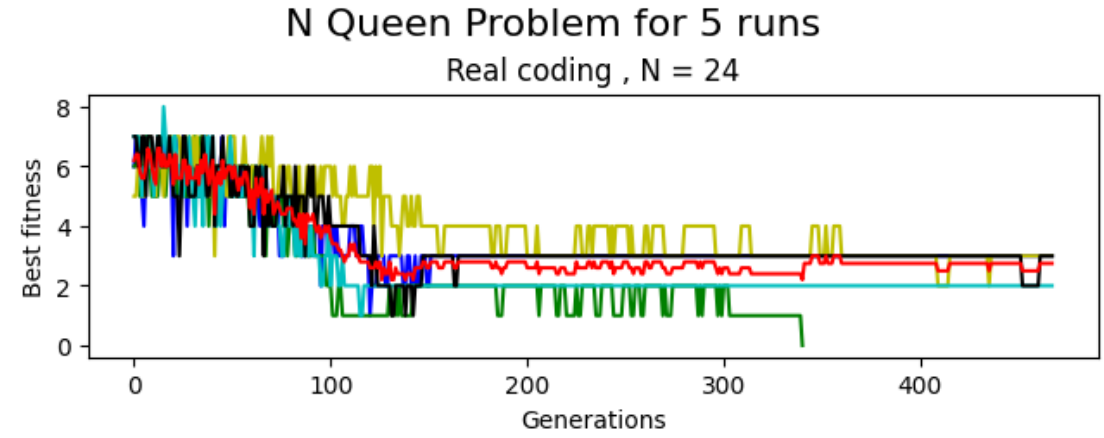

N Queen Problem for 5 runs
Real coding , N = 12
Best fitness
Generations
Real coding , N = 12
Average fitness
Generations
Run: 0
Run: 1
Run: 2
Run: 3
Run: 4
Average

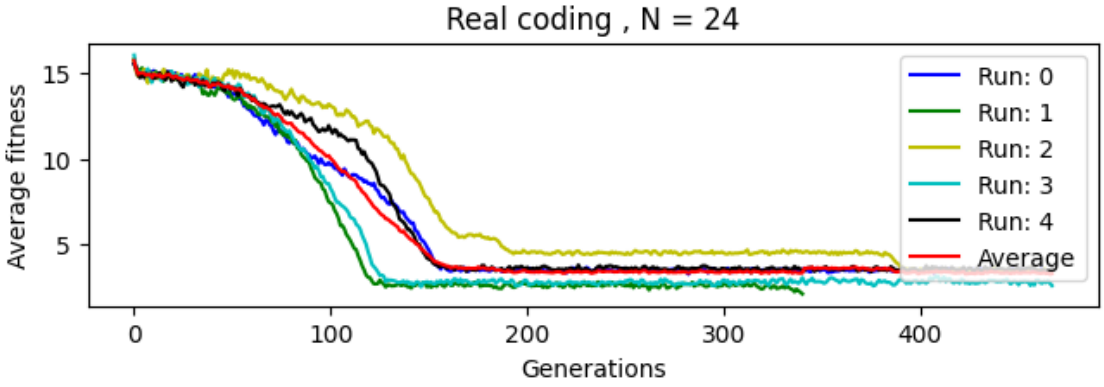

N Queen Problem for 5 runs
Real coding , N = 16
Best fitness
Generations

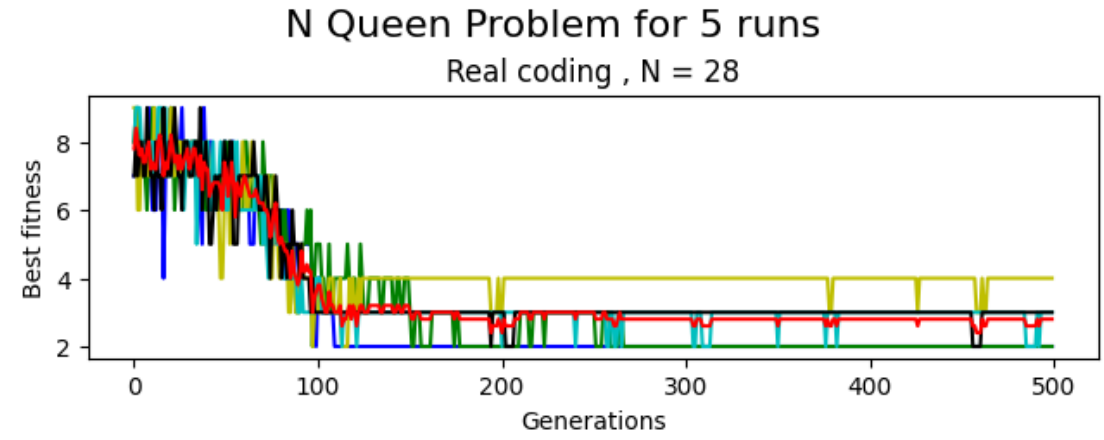

Real coding , N = 16
Average fitness
Generations
Run: 0
Run: 1
Run: 2
Run: 3
Run: 4
Average

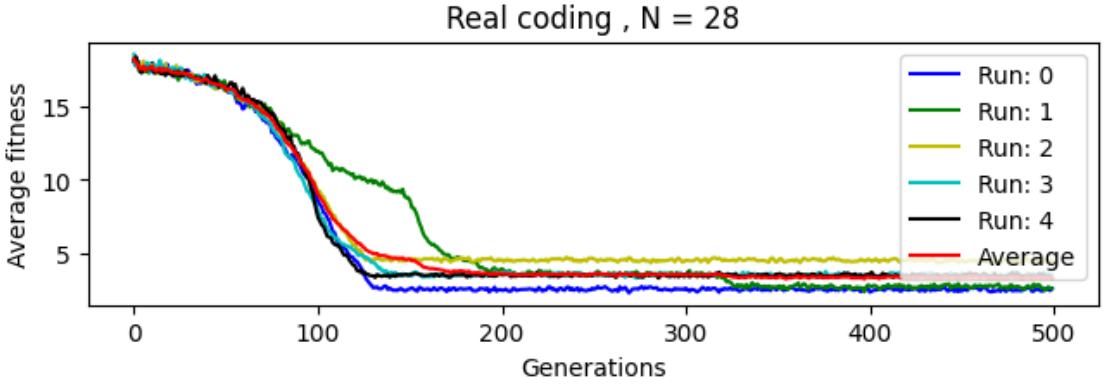

N Queen Problem for 5 runs
Real coding , N = 20
Best fitness
Generations

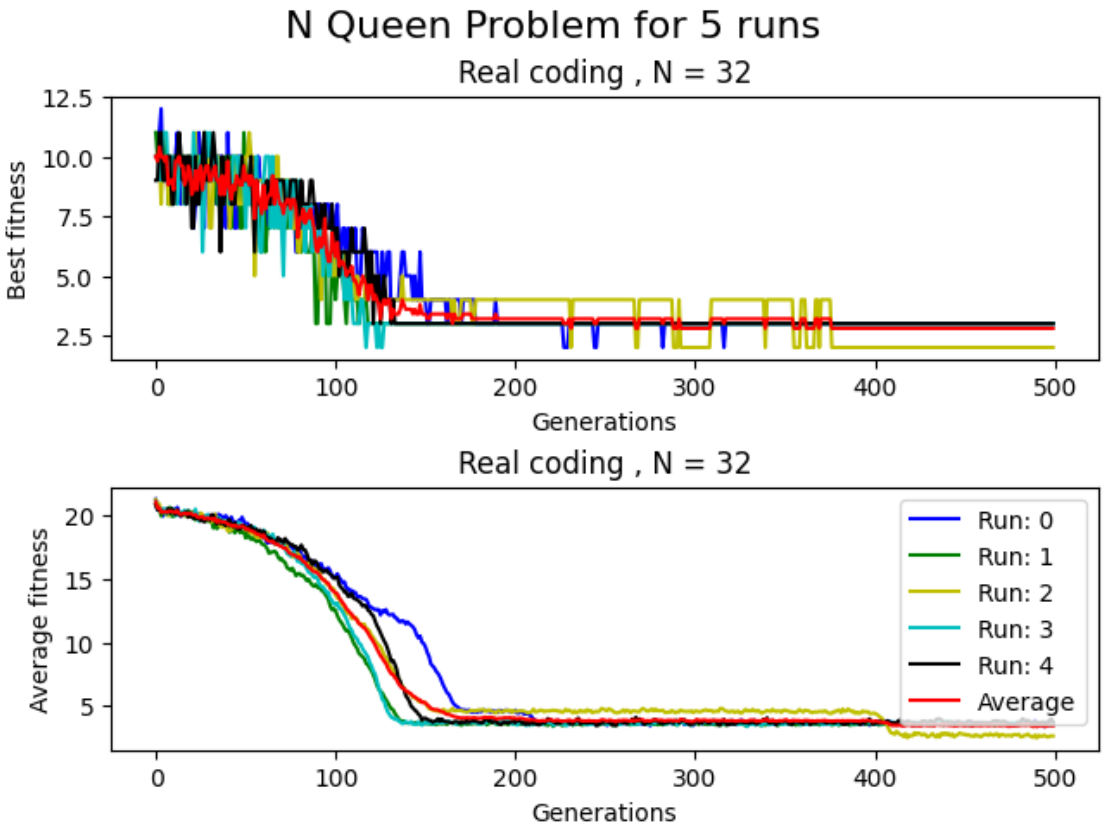

Real coding , N = 20
Average fitness
Generations
Run: 0
Run: 1
Run: 2
Run: 3
Run: 4
Average


N Queen Problem for 5 runs
Real coding , N = 24
Best fitness
Generations

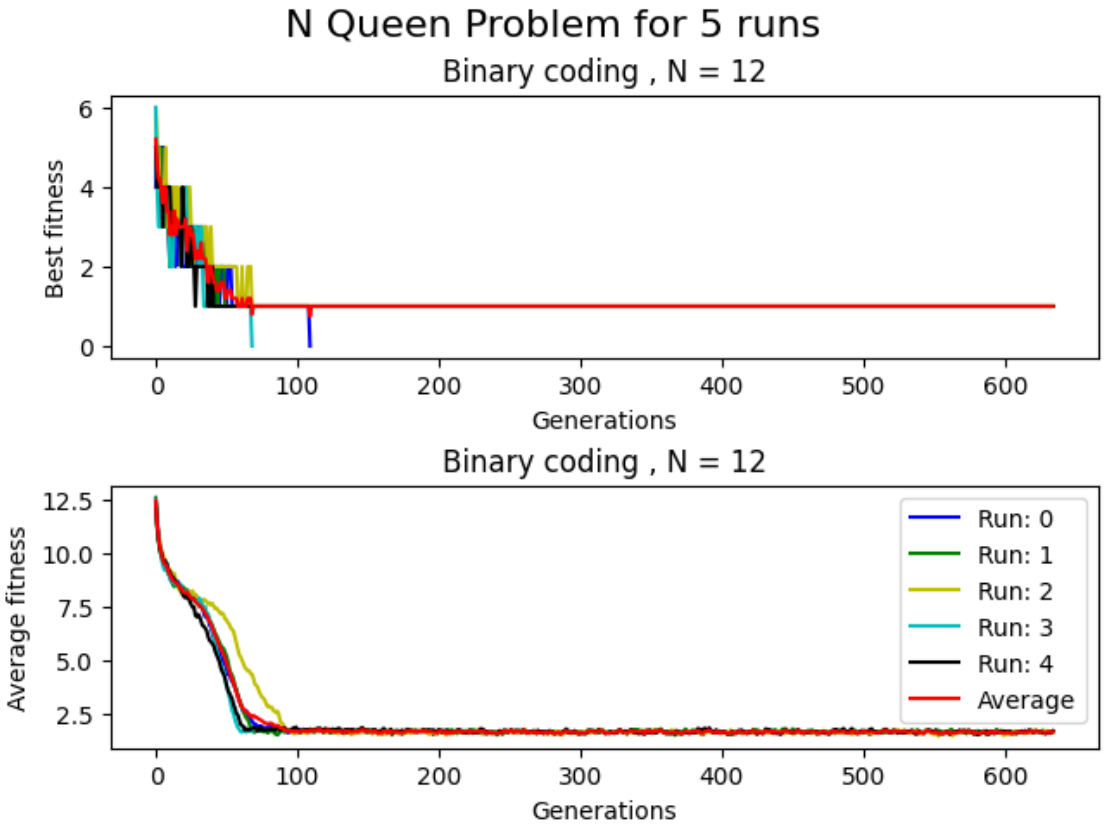

Real coding , N = 24
Average fitness
Generations
Run: 0
Run: 1
Run: 2
Run: 3
Run: 4
Average

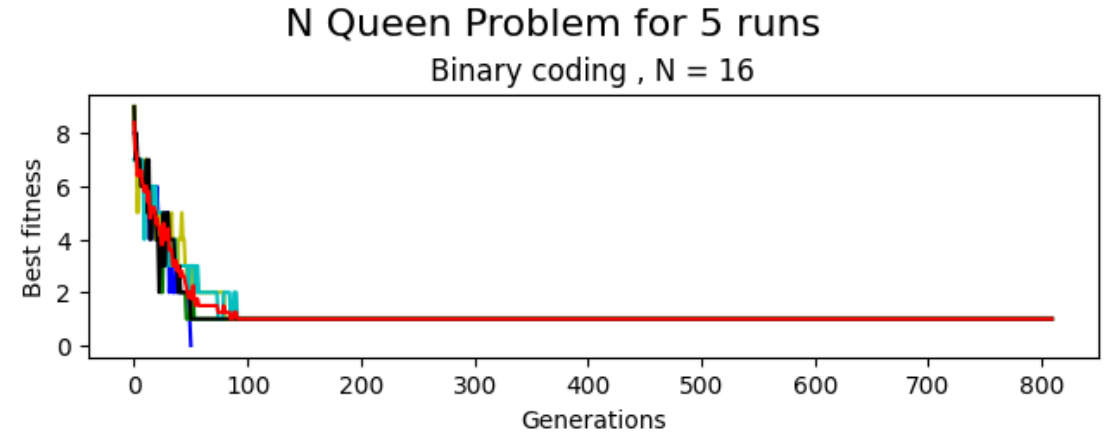

N Queen Problem for 5 runs
Real coding , N = 28
Best fitness
Generations

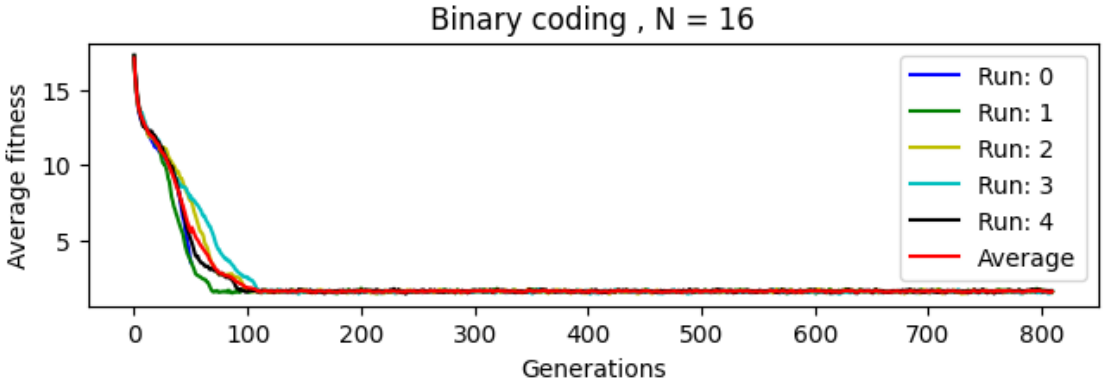

Real coding , N = 28
Average fitness
Generations
Run: 0
Run: 1
Run: 2
Run: 3
Run: 4
Average

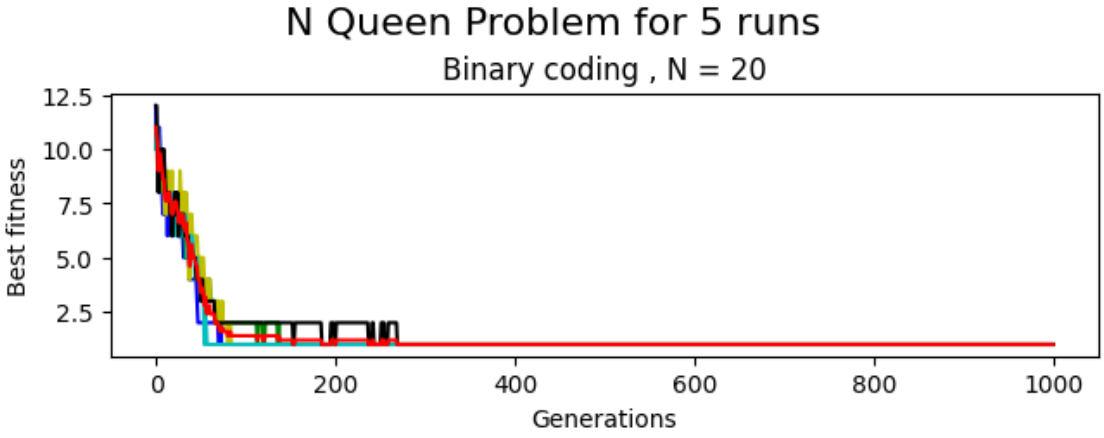

N Queen Problem for 5 runs
Real coding , N = 32
Best fitness
12.5
10.0
7.5
5.0
2.5
0
100
200
300
400
500
Generations
Real coding , N = 32
Average fitness
20
15
10
5
Run: 0
Run: 1
Run: 2
Run: 3
Run: 4
Average
0
100
200
300
400
500
Generations

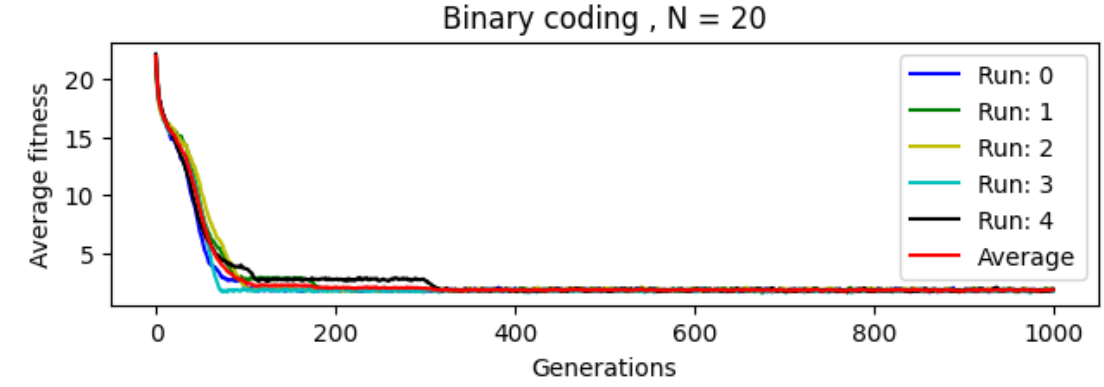
2.2.3. Binaria

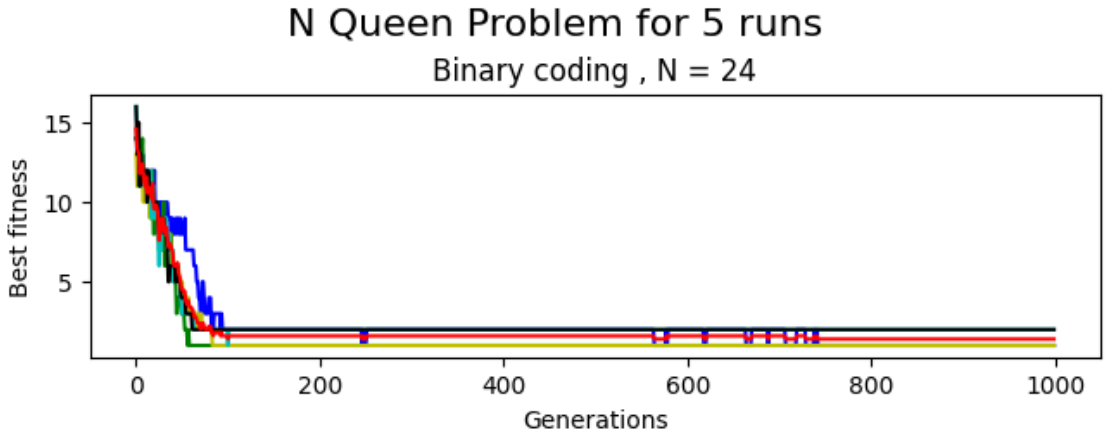

N Queen Problem for 5 runs
Binary coding , N = 12
Best fitness
Generations
Binary coding , N = 12
Average fitness
Generations
Run: 0
Run: 1
Run: 2
Run: 3
Run: 4
Average

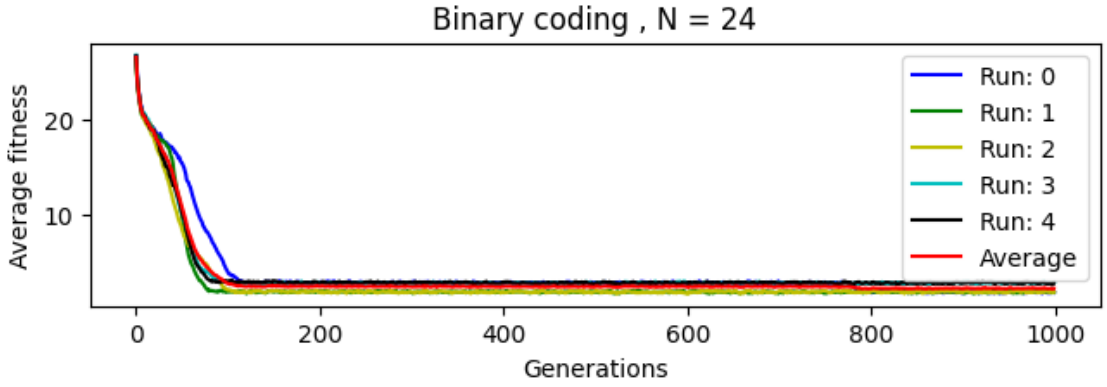

N Queen Problem for 5 runs
Binary coding , N = 16
Best fitness
Generations

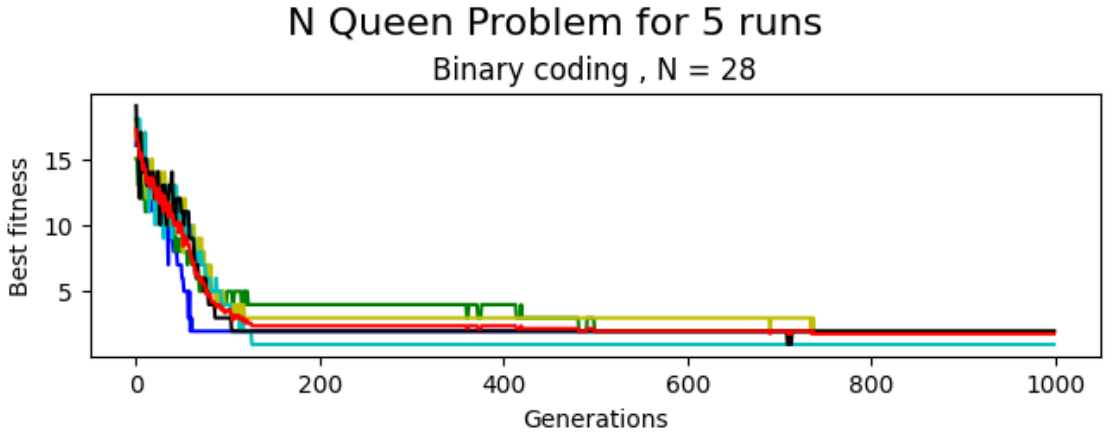

Binary coding , N = 16
Average fitness
Generations
Run: 0
Run: 1
Run: 2
Run: 3
Run: 4
Average

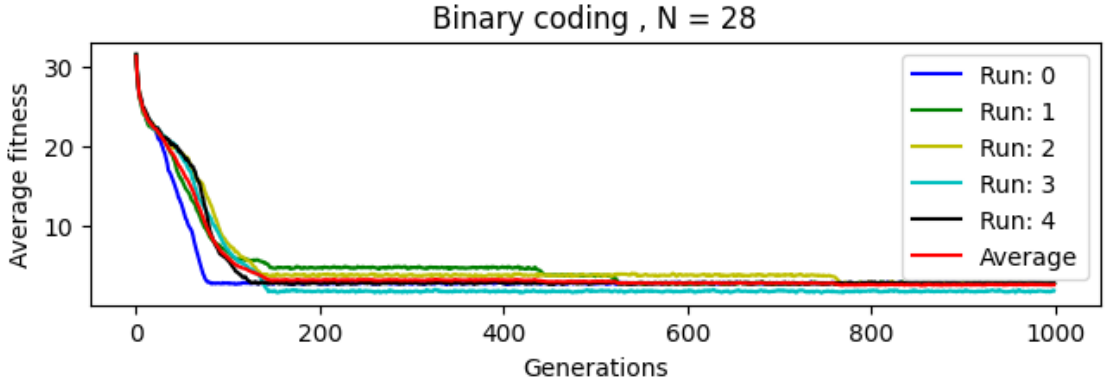

N Queen Problem for 5 runs
Binary coding , N = 20
Best fitness
Generations

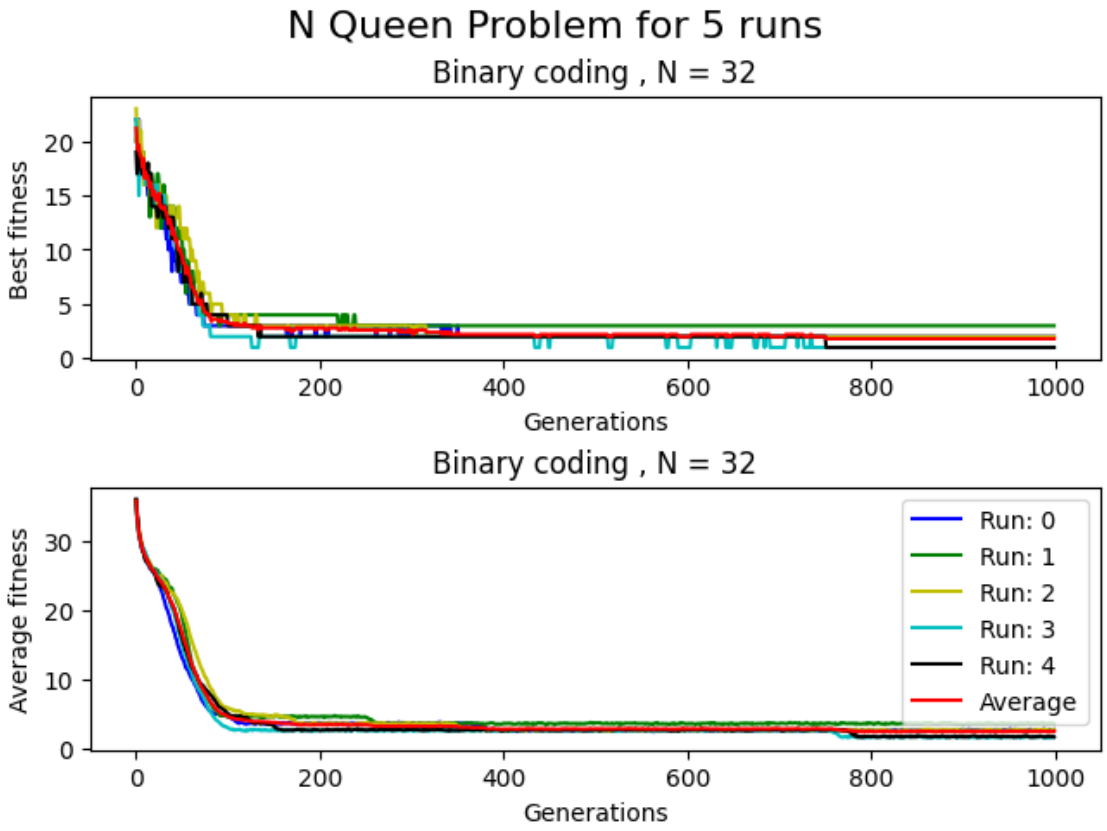

Binary coding , N = 20
Average fitness
Generations
Run: 0
Run: 1
Run: 2
Run: 3
Run: 4
Average


N Queen Problem for 5 runs
Binary coding , N = 24
Best fitness
15
10
5
0
200
400
600
800
1000
Generations

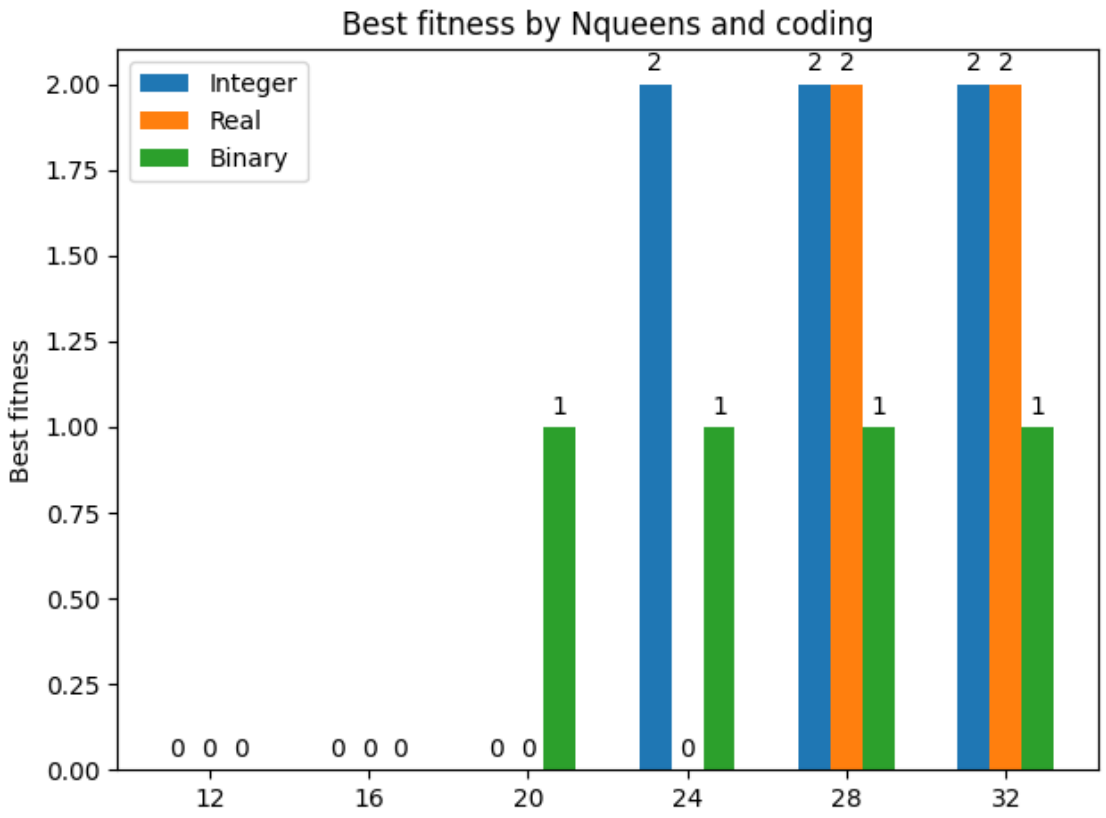

Binary coding , N = 24
Average fitness
20
10
0
200
400
600
800
1000
Generations
Run: 0
Run: 1
Run: 2
Run: 3
Run: 4
Average

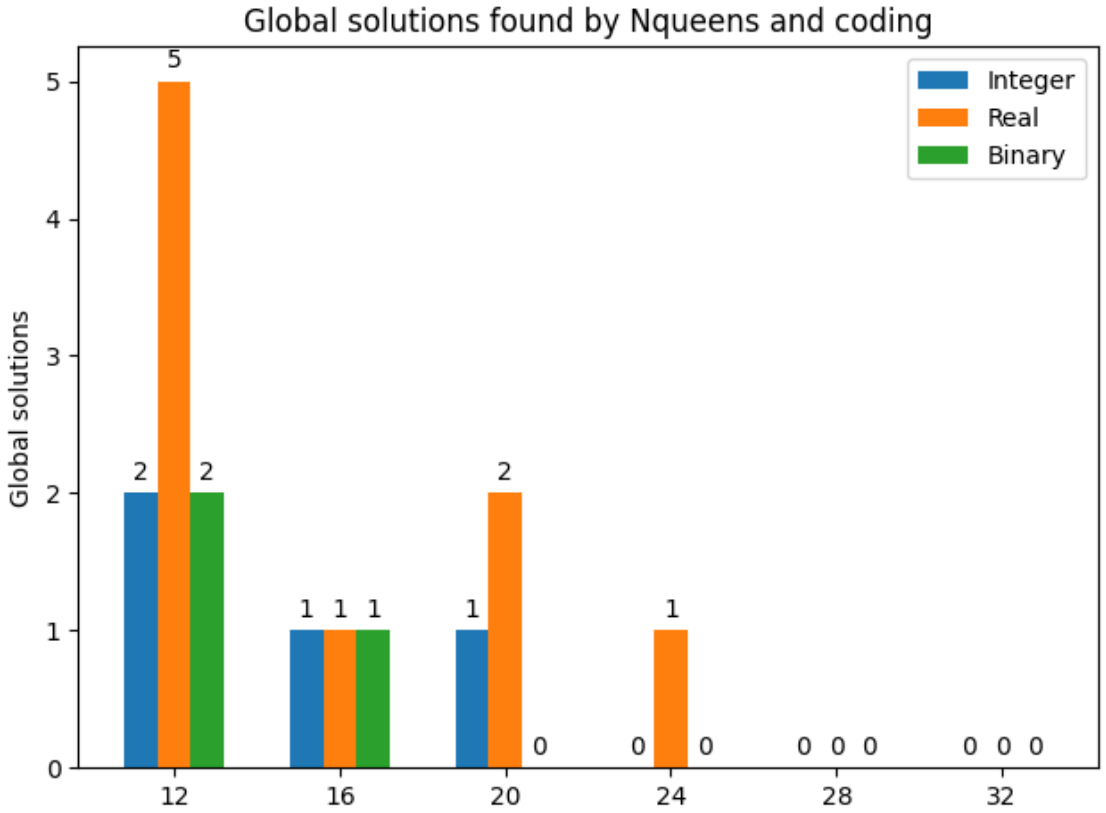

N Queen Problem for 5 runs
Binary coding , N = 28
Best fitness
15
10
5
0
200
400
600
800
1000
Generations


Binary coding , N = 28
Average fitness
30
20
10
0
200
400
600
800
1000
Generations
Run: 0
Run: 1
Run: 2
Run: 3
Run: 4
Average



Además de lo anterior, para poder cuantificar de mejor manera los rendimientos alcanzados con cada codificación en cada instancia del problema, se guardaron las aptitudes de los mejores individuos de cada codificación e instancia en las 5 corridas y la cantidad de veces que cada combinación de codificación-instancia alcanzó un mínimo global (ningún conflicto, es decir, que logró resolver el problema de las n-reinas).


Best fitness by Nqueens and coding
Integer
Real
Binary
Best fitness
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0 0 0
0 0 0
0 0
1
2
0
1
2 2
1
2 2
1
12
16
20
24
28
32


Global solutions found by Nqueens and coding
Integer
Real
Binary
Global solutions
5
4
3
2
1
0
2
5
2
1 1 1
1
2
0
0
1
0
0 0 0
0 0 0
12
16
20
24
28
32

3. Discusión.

Analizando los gráficos de fitness, junto con los de mejor individuo y solución global es posible comentar lo siguiente.

3.1. Entera

Respecto a los resultados con esta codificación, se puede ver que en la instancia con $n = 12, 16, 20$ se llegó a un mínimo global o solución al problema de las n reinas, el cual se encontró en un promedio de aproximadamente 60 generaciones para $n = 12$, este minimo solo se alcanzó en dos de las cinco corridas. Para $n = 16$, se alcanzó el mínimo en 40 generaciones con únicamente una sola corrida, mientras que para $n = 20$, se logró llegar al mínimo con 90 generaciones. Con las demás instancias para $n = 24, 28, 32$, lo más que se logró fue que el algoritmo se estancara en mínimos locales con 2 conflictos. Para $n = 24$, se tuvo el peor resultados en cuanto a los mejores individuos, donde se tuvieron la peor de la mejor solución. Para $n = 12$, se tuvo un empate con la codificación binaria en cuanto a la menor cantidad de soluciones globales encontradas y no solo eso, sino en promedio el algoritmo corrió por 320 generaciones, bastante más que las 30 de la codificación real.

3.2. Real

Con esta codificación se logró encontrar mínimos globales para $n = 12, 16, 20, 24$, y no solo eso, sino que se encontraron la mayor cantidad de mínimos globales para $n = 12, 20, 24$, para $n = 12$ en todas las corridas se encontraron mínimos globales. Además, la cantidad de generaciones en promedio que le tomó al algoritmo converger al mínimo global fue la menor para cada una de esas instancias del problema.

3.3. Binaria

Con esta codificación se encontraron mínimos globales para $n = 12, 16$, cabe recalcar que para esta codificación se utilizó un máximo de 1000 generaciones, a comparación con las 500 generaciones de las configuraciones anteriores, esto porque el espacio de búsqueda pareciera ser más grande por la longitud de las cadenas de bits representativas de los individuos y porque la operación de cruza para esta representación resulta en mayor variabilidad de los hijos y mayor pérdida de información, por lo tanto se le debía dar más tiempo al algoritmo para explorar. Con el ajuste mencionado que se deduce de manera empírica, aún así, el desempeño del algoritmo en cuanto a cantidad de soluciones encontradas por instancia del problema fue el peor, si además tomamos en cuenta que fue la codificación que corrió un mayor número de generaciones, en este rubro también fue la peor.

3.4. Conclusión

Los resultados tanto en rendimiento en términos de la cantidad de soluciones encontradas como en tiempo de ejecución parecen favorecer a la codificación real, pareciera ser que la

eliminación del espacio de soluciones de todas aquellas que sitúen a reinas en un mismo renglón mejora de manera sustancial la convergencia a mínimos globales, es decir, se está tratando con un espacio de búsqueda más pequeño.