Tarea 5 - Programación Genética, Neutralidad y Bloating

October 18, 2025

1 Tarea 5: Programación Genética, Neutralidad y Bloating.

Estudiante: Rodrigo S. Cortez Madrigal

La Teoría Neutral de la Evolución Molecular propuesta por Motoo Kimura, sostiene que muchas mutaciones en el ADN son neutras, es decir, no tiene un efecto significativo que afecten la aptitud del organismo.

En Programación Genética, esto se traduce en la idea de que ciertos cambios en la estructura del árbol no cambian el resultado observable del programa. Es decir, hay individuos (programas) con diferente estructura genética (genotipo) que, sin embargo, producen el mismo valor de aptitud (es decir, tienen el mismo desempeño en la tarea a resolver). Esto implica que ciertas modificaciones genéticas (como mutaciones o cruces) no alteran el comportamiento funcional del programa, y por tanto no afectan su evaluación.

En biología, la neutralidad puede manifestarse de varias maneras, y en PG podemos observar fenómenos similares. Algunas de las manifestaciones más comunes incluyen los intrones.

Son partes del programa que nunca se ejecutan (inactivo) o que no influyen en la salida.

Por ejemplo:

- (if (2 = 1) … x) → la rama del if nunca se ejecuta.
- (+ x (- x x)) → el término (- x x) siempre es cero y no afecta el resultado.

Estos fragmentos pueden crecer sin afectar la aptitud, contribuyendo al fenómeno conocido como hinchamiento (bloat).

1.1 Objetivo

Analizar empíricamente el fenómeno de bloat y neutralidad en algoritmos de Programación Genética, bajo dos configuraciones.

Lineamientos metodológicos:

- Ejecución completa: correr siempre hasta maxGen (no emplear paro temprano por convergencia).
- Diversidad sin restricciones: no utilizar elitismo ni penalizaciones que limiten la diversidad en la generación de individuos.
- Repeticiones: realizar 10 corridas independientes por experimento (use semillas distintas).
- Evaluación: defina explícitamente la métrica de aptitud (por ejemplo, error absoluto f(x)−10). Control de tamaño: respete nodosMax como restricción.

Resultados y visualizaciones requeridas:

- Dinámica de terminales: graficar, por generación, el conteo de 0, 1 y 4 en la población (promedio sobre las 10 corridas).
- Bloat: graficar el tamaño promedio de los árboles por generación para evidenciar crecimiento estructural.
- Comparación entre configuraciones: presentar en una figura o tabla la comparación de métricas clave (tamaño promedio, mejor aptitud final, tasa de soluciones exactas) entre {+} y {+, *}.

Entregables:

- Planteamiento y definiciones operativas.
- Descripción de parámetros y operadores (incluya semillas).
- Resultados con las gráficas solicitadas y discusión crítica.
- Conclusiones y limitaciones.

[ ]:
```
import deap
import math
import operator
import random
import functools
import numpy as np
import pandas as pd
from deap import gp, base, creator, tools, algorithms
import plotly.graph_objs as go
from plotly import express as px
import matplotlib.colors as mcolors

# Plotly render svgs
import plotly.io as pio
pio.renderers.default = "svg"
```

[2]:
```
# Fitness function

def target_function():
    """Valor objetivo constante: 10"""
    return 10
```

1.2 Experimento 1

- Funciones: {+}
- Terminales: {0, 1, 4}
- Prob. de cruza: 0.9
- Prob. de mutación nodo: 0.01
- Tamaño de población: 100
- Generaciones máximas: 1000
- Tamaño máximo de árbol: 120
- Objetivo: Inducir una expresión que sume 10.

[ ]:
```
POP_SIZE = 100 # tamaño de la población
N_GEN = 1000 # número de generaciones
CXPB = 0.9   # probabilidad de cruzamiento
MUTPB = 0.01  # probabilidad de mutación
MAX_TREE_SIZE = 120 # tamaño máximo del árbol
RANDOM_SEEDS = [42, 56, 78, 91, 13, 27, 34, 65, 89, 100]
```

[ ]:
```
# Primitivas y terminales

# Definición del conjunto de primitivas y terminales
pset = gp.PrimitiveSet('EXP1', 0)  # 0 argumentos de entrada

# Funciones
pset.addPrimitive(operator.add, 2)  # Suma

# Terminales constantes
pset.addTerminal(0)
pset.addTerminal(1)
pset.addTerminal(4)

# Fitness
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))  # Minimizar el␣
 ↪error
creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMin)

# Toolbox
toolbox = base.Toolbox()
toolbox.register("expr", gp.genFull, pset=pset, min_=1, max_=3) # Generar␣
 ↪árboles completos
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.
 ↪expr) # Crear individuos
toolbox.register("population", tools.initRepeat, list, toolbox.individual) #␣
 ↪Crear población
toolbox.register("compile", gp.compile, pset=pset) # Compilar árboles a␣
 ↪funciones

# Evaluación


def fitness_function(individual, pset):
    """Compila el individuo (árbol GP) y calcula la aptitud como una tupla
    con el error absoluto |f(x) - 10|.
    """
    # Compila el árbol en una función ejecutable usando el primitive set
    func = gp.compile(expr=individual, pset=pset)
    if callable(func):
        try:
```

```
            value = func()
        except Exception:
            return (float('inf'),)
    else:
        value = func

    # Asegurar que el resultado sea numérico
    try:
        err = abs(float(value) - target_function())
    except Exception:
        return (float('inf'),)

    # DEAP espera una tupla como valor de fitness
    return (err,)

toolbox.register("evaluate", fitness_function, pset=pset)

# Operadores genéticos

toolbox.register("mate", gp.cxOnePoint)  # Cruzamiento de un punto
toolbox.register("expr_mut", gp.genFull, min_=0, max_=2) # Mutación
toolbox.register("mutate", gp.mutUniform, expr=toolbox.expr_mut, pset=pset) #␣
 ↪Mutación uniforme
toolbox.register("select", tools.selTournament, tournsize=3)  # Selección por␣
 ↪torneo

# Control de tamaño
toolbox.decorate("mate", gp.staticLimit(key=operator.attrgetter("height"),␣
 ↪max_value=MAX_TREE_SIZE))
toolbox.decorate("mutate", gp.staticLimit(key=operator.attrgetter("height"),␣
 ↪max_value=MAX_TREE_SIZE))

# Estadísticas y registro del mejor
stats = tools.Statistics(lambda ind: ind.fitness.values[0])
stats.register("min", min)
stats.register("avg", np.mean)
stats.register("std", np.std)
hof = tools.HallOfFame(1)
```

[ ]:
```
# Veamos algunos individuos iniciales

pop = toolbox.population(n=POP_SIZE) # Población inicial

for i, ind in enumerate(pop[:5]):  # Muestra los primeros 5 árboles
    print(f"Árbol {i}:")
    print("Expresión:", str(ind))
    val = toolbox.compile(expr=ind)
```

```
    print("Valor evaluado:", val)
```

```
Árbol 0:
Expresión: add(0, 4)
Valor evaluado: 4
----------------------------------------
Árbol 1:
Expresión: add(add(add(0, 4), add(1, 4)), add(add(0, 4), add(4, 4)))
Valor evaluado: 21
----------------------------------------
Árbol 2:
Expresión: add(add(0, 0), add(0, 0))
Valor evaluado: 0
----------------------------------------
Árbol 3:
Expresión: add(0, 4)
Valor evaluado: 4
----------------------------------------
Árbol 4:
Expresión: add(4, 1)
Valor evaluado: 5
----------------------------------------
```

[ ]:
```
# Tomado de deap.algorithms.eaSimple, pero modificado para retornar todas las␣
 ↪poblaciones

def custom_eaSimple(population, toolbox, cxpb, mutpb, ngen, stats=None,␣
 ↪halloffame=None, verbose=True):
    logbook = tools.Logbook()
    logbook.header = ['gen', 'nevals'] + (stats.fields if stats else [])
    populations = []

    # Evaluate the individuals with an invalid fitness
    invalid_ind = [ind for ind in population if not ind.fitness.valid]
    fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit

    if halloffame is not None:
        halloffame.update(population)

    record = stats.compile(population) if stats else {}
    logbook.record(gen=0, nevals=len(invalid_ind), **record)
    if verbose:
        print(logbook.stream)

    # Begin the generational process
```

```
    for gen in range(1, ngen + 1):
        # Select the next generation individuals
        offspring = toolbox.select(population, len(population))

        # Vary the pool of individuals
        offspring = algorithms.varAnd(offspring, toolbox, cxpb, mutpb)

        # Evaluate the individuals with an invalid fitness
        invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
        fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit

        # Update the hall of fame with the generated individuals
        if halloffame is not None:
            halloffame.update(offspring)

        # Replace the current population by the offspring
        population[:] = offspring

        # Append the current generation statistics to the logbook
        record = stats.compile(population) if stats else {}
        logbook.record(gen=gen, nevals=len(invalid_ind), **record)
        if verbose:
            print(logbook.stream)
        populations.append([ind.copy() for ind in population])

    return populations, logbook
```

1.2.1 Corrida del experimento 1

[7]:
```
# Corremos múltiples semillas y guardamos las poblaciones

logs = []
all_pops = {}  # all_pops[seed][gen]

for seed in RANDOM_SEEDS:
    random.seed(seed)
    pop = toolbox.population(n=POP_SIZE)

    pops, log = custom_eaSimple(
        pop, toolbox, cxpb=CXPB, mutpb=MUTPB, ngen=N_GEN,
        stats=stats, halloffame=hof, verbose=False,
    )
    log = pd.DataFrame(log)
    logs.append(log)
    all_pops[seed] = pops
```

[51]:
```
# Graficar, por generación, el fitness mínimo promedio y su desviación estándar␣
 ↪por corrida

logs_df = pd.concat(logs, keys=RANDOM_SEEDS, names=['seed']).reset_index().
 ↪drop(columns=['level_1'])
```

[52]:
```
logs_df
```

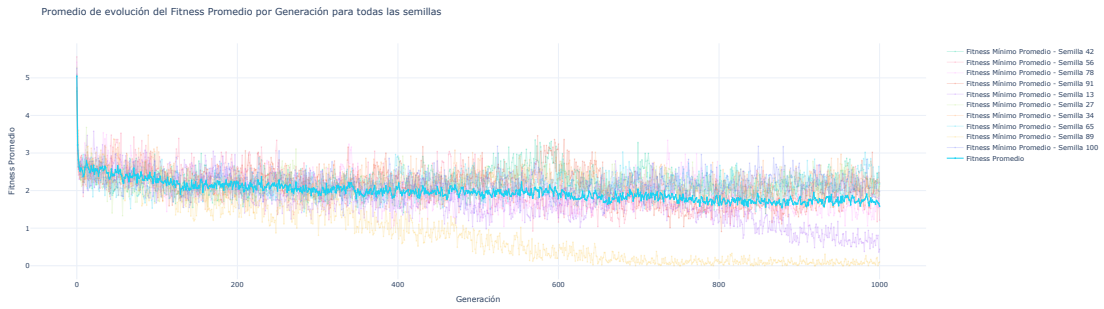
[52]:

	seed	gen	nevals	min	avg	std
0	42	0	100	0.0	4.64	2.819645
1	42	1	92	0.0	3.52	2.256014
2	42	2	92	0.0	2.72	1.855155
3	42	3	86	0.0	2.57	1.893436
4	42	4	96	0.0	2.79	2.169309
…	…	…	…	…	…	…
10005	100	996	93	0.0	2.10	1.682260
10006	100	997	94	0.0	2.21	1.940593
10007	100	998	90	0.0	2.00	1.860108
10008	100	999	88	0.0	1.61	1.586789
10009	100	1000	98	0.0	1.98	1.843800

[10010 rows x 6 columns]

[79]:
```
# Graficar, por generación, el fitness mínimo promedio y su desviación estándar␣
 ↪por corrida


mean_df = logs_df.groupby('gen').agg({'avg': 'mean', 'std': 'std'}).
 ↪reset_index()

fig = go.Figure()

for seed in RANDOM_SEEDS:
    seed_df = logs_df[logs_df['seed'] == seed]
    hex_color = px.colors.qualitative.Plotly[seed % 10]
    rgba = mcolors.to_rgba(hex_color, alpha=0.2)
    rgba_str = f'rgba({int(rgba[0]*255)}, {int(rgba[1]*255)},␣
 ↪{int(rgba[2]*255)}, {rgba[3]})'

    fig.add_trace(go.Scatter(
        x=seed_df['gen'],
        y=seed_df['avg'],
        mode='lines+markers',
        name=f'Fitness Mínimo Promedio - Semilla {seed}',
        line=dict(color=rgba_str)
    ))
```

```
fig.add_trace(go.Scatter(
    x=mean_df['gen'],
    y=mean_df['avg'],
    error_x=dict(type='data', array=mean_df['std'], visible=False),
    mode='lines+markers',
    name='Fitness Promedio',
    line=dict(color=px.colors.qualitative.Plotly[5], width=1)
))

fig.update_layout(
    title='Promedio de evolución del Fitness Promedio por Generación para todas␣
 ↪las semillas',
    xaxis_title='Generación',
    yaxis_title='Fitness Promedio',
    template='plotly_white'
)
fig.update_traces(marker=dict(size=3))
fig.update_traces(line=dict(width=1))
fig.update_layout(width=2000, height=600)
fig.show()
```



[8]:
```
len(all_pops[42]) # Confirma que hay 1000 generaciones guardadas
```

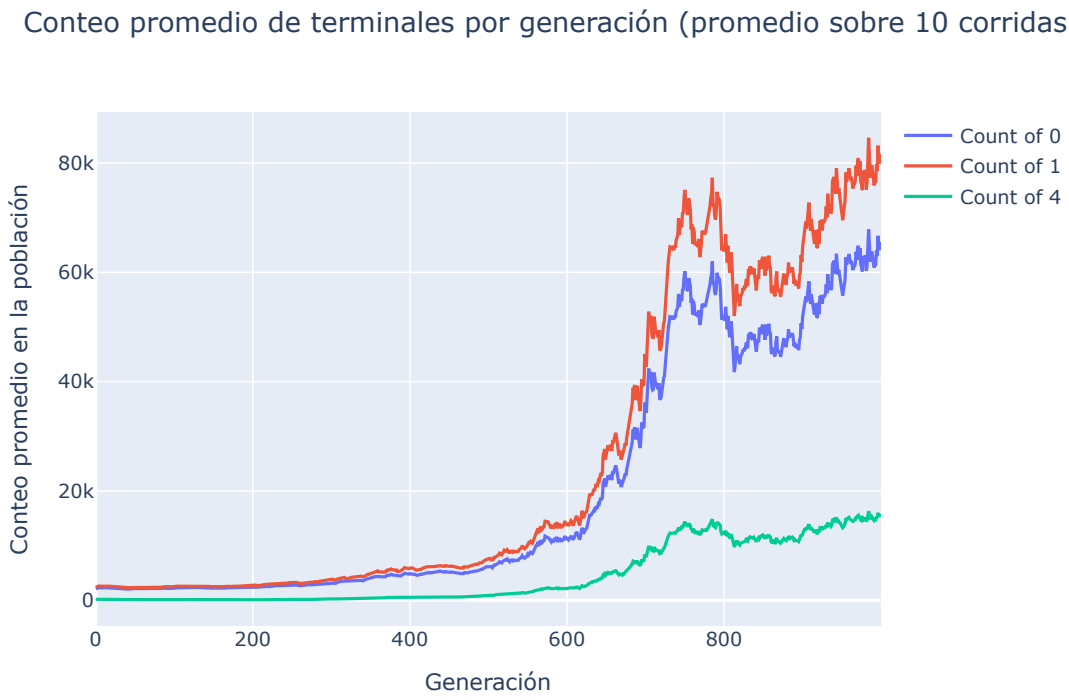
[8]: 1000

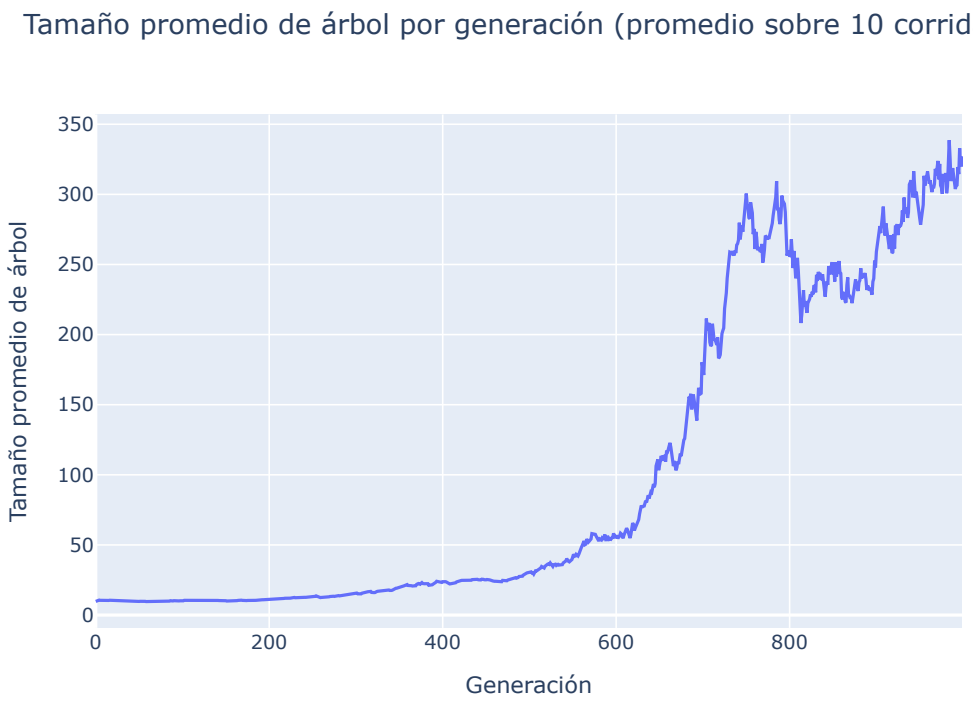
[9]:
```
# Graficar, por generación, el conteo de 0, 1 y 4 en la población (promedio␣
 ↪sobre las 10 corridas)
# Calcular el tamaño promedio de los árboles por generación y su promedio sobre␣
 ↪las corridas

counts = [] # Para guardar el conteo por generación de 0, 1 y 4 para cada
avg_tree_sizes = []  # Para guardar el tamaño promedio de árboles por␣
 ↪generación para cada corrida
```

```
for seed in RANDOM_SEEDS:
    gen_counts = {'0': [], '1': [], '4': []}
    gen_tree_sizes = []
    for gen in range(N_GEN):
        pop = all_pops[seed][gen]
        count_0 = sum(str(ind).count('0') for ind in pop)
        count_1 = sum(str(ind).count('1') for ind in pop)
        count_4 = sum(str(ind).count('4') for ind in pop)
        gen_counts['0'].append(count_0)
        gen_counts['1'].append(count_1)
        gen_counts['4'].append(count_4)
        # Tamaño promedio de árboles en esta generación
        avg_size = np.mean([len(ind) for ind in pop])
        gen_tree_sizes.append(avg_size)
    counts.append(gen_counts)
    avg_tree_sizes.append(gen_tree_sizes)

# Ahora counts tiene, para cada semilla, el conteo por generación de 0, 1 y 4
# avg_tree_sizes tiene, para cada semilla, el tamaño promedio de árboles por␣
 ↪generación

avg_counts = {'0': [], '1': [], '4': []}
avg_tree_size_per_gen = []
for gen in range(N_GEN):
    # Promedio del conteo de terminales en esta generación sobre todas las␣
 ↪corridas
    avg_0 = np.mean([counts[seed_idx]['0'][gen] for seed_idx in␣
 ↪range(len(RANDOM_SEEDS))])
    avg_1 = np.mean([counts[seed_idx]['1'][gen] for seed_idx in␣
 ↪range(len(RANDOM_SEEDS))])
    avg_4 = np.mean([counts[seed_idx]['4'][gen] for seed_idx in␣
 ↪range(len(RANDOM_SEEDS))])
    avg_counts['0'].append(avg_0)
    avg_counts['1'].append(avg_1)
    avg_counts['4'].append(avg_4)
    # Promedio del tamaño de árbol en esta generación sobre todas las corridas
    avg_tree_size = np.mean([avg_tree_sizes[seed_idx][gen] for seed_idx in␣
 ↪range(len(RANDOM_SEEDS))])
    avg_tree_size_per_gen.append(avg_tree_size)
```

[10]:
```
# Graficar el conteo promedio de terminales por generación para todas las␣
 ↪corridas
fig = go.Figure()
fig.add_trace(go.Scatter(x=list(range(N_GEN)), y=avg_counts['0'], mode='lines',␣
 ↪name='Count of 0'))
```

```
fig.add_trace(go.Scatter(x=list(range(N_GEN)), y=avg_counts['1'], mode='lines',␣
 ↪name='Count of 1'))
fig.add_trace(go.Scatter(x=list(range(N_GEN)), y=avg_counts['4'], mode='lines',␣
 ↪name='Count of 4'))
fig.update_layout(title='Conteo promedio de terminales por generación (promedio␣
 ↪sobre 10 corridas)',
                  xaxis_title='Generación',
                  yaxis_title='Conteo promedio en la población')
fig.show()

# Promedio del tamaño del árbol por generación para todas las corridas
fig2 = go.Figure()
fig2.add_trace(go.Scatter(x=list(range(N_GEN)), y=avg_tree_size_per_gen,␣
 ↪mode='lines', name='Tamaño promedio de árbol'))
fig2.update_layout(title='Tamaño promedio de árbol por generación (promedio␣
 ↪sobre 10 corridas)',
                   xaxis_title='Generación',
                   yaxis_title='Tamaño promedio de árbol')
fig2.show()
```





Observamos que el tamaño promedio de árbol por generación para las 10 corridas del experimento 1 tiende a incrementarse con el tiempo, lo que indica la presencia del fenómeno de bloat. A pesar de que la aptitud mejora en las primeras generaciones, el tamaño de los árboles sigue creciendo, lo que sugiere que los programas están acumulando intrones o fragmentos neutrales que no contribuyen a mejorar la solución.

1.3 Experimento 2

Mismo objetivo y parámetros que el Experimento 1, modificando: - Funciones: {+, *} - Prob. de mutación de rama : 0.1

[ ]:
```
POP_SIZE = 100
N_GEN = 1000
CXPB = 0.9   # probabilidad de cruzamiento
MUTPB = 0.1  # probabilidad de mutación
MAX_TREE_SIZE = 120
RANDOM_SEEDS = [42, 56, 78, 91, 13, 27, 34, 65, 89, 100] # mismas semillas que␣
 ↪antes
```

[ ]:
```
# Primitivas y terminales

# Definición del conjunto de primitivas y terminales
```

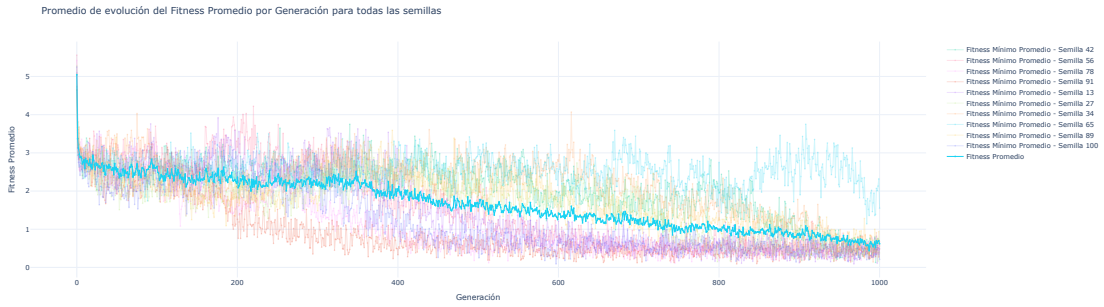
```
pset2 = gp.PrimitiveSet('EXP1', 0)  # 0 argumentos de entrada

# Funciones
pset2.addPrimitive(operator.add, 2)  # Suma
pset2.addPrimitive(operator.mul, 2)  # Multiplicación

# Terminales constantes
pset2.addTerminal(0)
pset2.addTerminal(1)
pset2.addTerminal(4)

# Fitness
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))  # Minimizar el␣
 ↪error
creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMin)

# Toolbox
toolbox2 = base.Toolbox()
toolbox2.register("expr", gp.genFull, pset=pset2, min_=1, max_=3) # Generar␣
 ↪árboles completos
toolbox2.register("individual", tools.initIterate, creator.Individual, toolbox2.
 ↪expr) # Crear individuos
toolbox2.register("population", tools.initRepeat, list, toolbox2.individual) #␣
 ↪Crear población
toolbox2.register("compile", gp.compile, pset=pset2) # Compilar árboles a␣
 ↪funciones

# Evaluación
toolbox2.register("evaluate", fitness_function, pset=pset2)

# Operadores genéticos

toolbox2.register("mate", gp.cxOnePoint)  # Cruzamiento de un punto
toolbox2.register("expr_mut", gp.genFull, min_=0, max_=2) # Mutación
toolbox2.register("mutate", gp.mutUniform, expr=toolbox2.expr_mut, pset=pset2)␣
 ↪# Mutación uniforme
toolbox2.register("select", tools.selTournament, tournsize=3)  # Selección por␣
 ↪torneo de 3 individuos

# Control de tamaño
toolbox2.decorate("mate", gp.staticLimit(key=operator.attrgetter("height"),␣
 ↪max_value=MAX_TREE_SIZE))
toolbox2.decorate("mutate", gp.staticLimit(key=operator.attrgetter("height"),␣
 ↪max_value=MAX_TREE_SIZE))

# Estadísticas y registro del mejor
```

```
stats2 = tools.Statistics(lambda ind: ind.fitness.values[0])
stats2.register("min", min)
stats2.register("avg", np.mean)
stats2.register("std", np.std)
hof2 = tools.HallOfFame(1)
```

[13]:
```
# Corremos múltiples semillas y guardamos las poblaciones

logs2 = []
all_pops2 = {}  # all_pops[seed][gen]

for seed in RANDOM_SEEDS:
    random.seed(seed)
    pop = toolbox.population(n=POP_SIZE)

    pops, log = custom_eaSimple(
        pop, toolbox, cxpb=CXPB, mutpb=MUTPB, ngen=N_GEN,
        stats=stats, halloffame=hof, verbose=False,
    )
    log = pd.DataFrame(log)
    logs2.append(log)
    all_pops2[seed] = pops
```

[78]:
```
# Graficar, por generación, el fitness mínimo promedio y su desviación estándar␣
 ↪por corrida

logs_df2 = pd.concat(logs2, keys=RANDOM_SEEDS, names=['seed']).reset_index().
 ↪drop(columns=['level_1'])
# Ahora el promedio de todas las corridas
mean_df2 = logs_df2.groupby('gen').agg({'avg': 'mean', 'std': 'std'}).
 ↪reset_index()

fig = go.Figure()

for seed in RANDOM_SEEDS:
    seed_df = logs_df2[logs_df2['seed'] == seed]
    hex_color = px.colors.qualitative.Plotly[seed % 10]
    rgba = mcolors.to_rgba(hex_color, alpha=0.2)
    rgba_str = f'rgba({int(rgba[0]*255)}, {int(rgba[1]*255)},␣
 ↪{int(rgba[2]*255)}, {rgba[3]})'

    fig.add_trace(go.Scatter(
        x=seed_df['gen'],
        y=seed_df['avg'],
        mode='lines+markers',
        name=f'Fitness Mínimo Promedio - Semilla {seed}',
        line=dict(color=rgba_str)
```

```
    ))

fig.add_trace(go.Scatter(
    x=mean_df2['gen'],
    y=mean_df2['avg'],
    error_x=dict(type='data', array=mean_df2['std'], visible=False),
    mode='lines+markers',
    name='Fitness Promedio',
    line=dict(color=px.colors.qualitative.Plotly[5], width=1)
))

fig.update_layout(
    title='Promedio de evolución del Fitness Promedio por Generación para todas␣
 ↪las semillas',
    xaxis_title='Generación',
    yaxis_title='Fitness Promedio',
    template='plotly_white'
)
fig.update_traces(marker=dict(size=3))
fig.update_traces(line=dict(width=1))
fig.update_layout(width=2000, height=600)
fig.show()
```

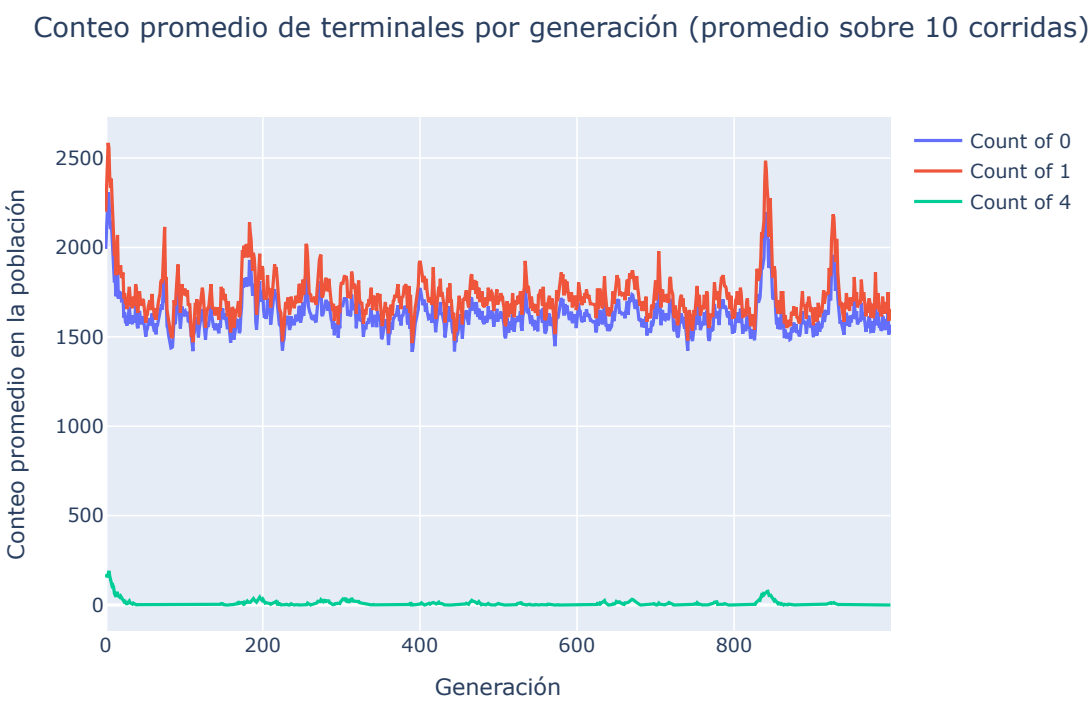


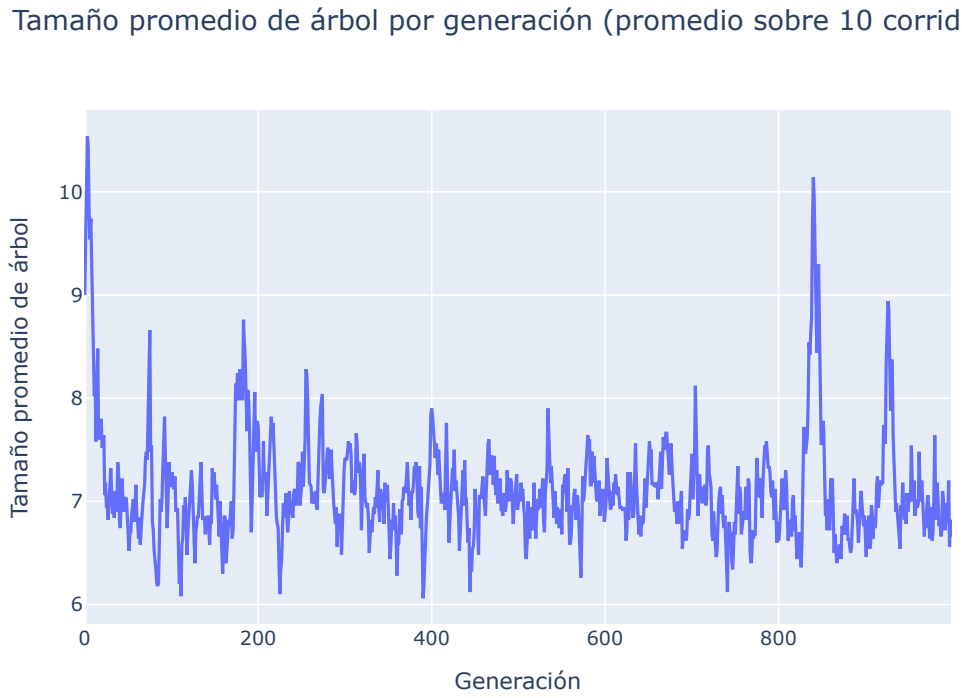
[14]:
```
# Graficar, por generación, el conteo de 0, 1 y 4 en la población (promedio␣
 ↪sobre las 10 corridas)
# Calcular el tamaño promedio de los árboles por generación y su promedio sobre␣
 ↪las corridas

counts2 = [] # Para guardar el conteo por generación de 0, 1 y 4 para cada
avg_tree_sizes2 = []  # Para guardar el tamaño promedio de árboles por␣
 ↪generación para cada corrida

for seed in RANDOM_SEEDS:
    gen_counts2 = {'0': [], '1': [], '4': []}
```

```
    gen_tree_sizes2 = []
    for gen in range(N_GEN):
        pop = all_pops2[seed][gen]
        count_0 = sum(str(ind).count('0') for ind in pop)
        count_1 = sum(str(ind).count('1') for ind in pop)
        count_4 = sum(str(ind).count('4') for ind in pop)
        gen_counts2['0'].append(count_0)
        gen_counts2['1'].append(count_1)
        gen_counts2['4'].append(count_4)
        # Tamaño promedio de árboles en esta generación
        avg_size2 = np.mean([len(ind) for ind in pop])
        gen_tree_sizes2.append(avg_size2)
    counts2.append(gen_counts)
    avg_tree_sizes2.append(gen_tree_sizes)

# Ahora counts tiene, para cada semilla, el conteo por generación de 0, 1 y 4
# avg_tree_sizes tiene, para cada semilla, el tamaño promedio de árboles por␣
 ↪generación

avg_counts2 = {'0': [], '1': [], '4': []}
avg_tree_size_per_gen2 = []
for gen in range(N_GEN):
    # Promedio del conteo de terminales en esta generación sobre todas las␣
 ↪corridas
    avg_0 = np.mean([counts2[seed_idx]['0'][gen] for seed_idx in␣
 ↪range(len(RANDOM_SEEDS))])
    avg_1 = np.mean([counts2[seed_idx]['1'][gen] for seed_idx in␣
 ↪range(len(RANDOM_SEEDS))])
    avg_4 = np.mean([counts2[seed_idx]['4'][gen] for seed_idx in␣
 ↪range(len(RANDOM_SEEDS))])
    avg_counts2['0'].append(avg_0)
    avg_counts2['1'].append(avg_1)
    avg_counts2['4'].append(avg_4)
    # Promedio del tamaño de árbol en esta generación sobre todas las corridas
    avg_tree_size2 = np.mean([avg_tree_sizes2[seed_idx][gen] for seed_idx in␣
 ↪range(len(RANDOM_SEEDS))])
    avg_tree_size_per_gen2.append(avg_tree_size2)
```

[15]:
```
# Graficar el conteo promedio de terminales por generación para todas las␣
 ↪corridas
fig = go.Figure()
fig.add_trace(go.Scatter(x=list(range(N_GEN)), y=avg_counts2['0'],␣
 ↪mode='lines', name='Count of 0'))
fig.add_trace(go.Scatter(x=list(range(N_GEN)), y=avg_counts2['1'],␣
 ↪mode='lines', name='Count of 1'))
```

```
fig.add_trace(go.Scatter(x=list(range(N_GEN)), y=avg_counts2['4'],␣
 ↪mode='lines', name='Count of 4'))
fig.update_layout(title='Conteo promedio de terminales por generación (promedio␣
 ↪sobre 10 corridas)',
                  xaxis_title='Generación',
                  yaxis_title='Conteo promedio en la población')
fig.show()

# Promedio del tamaño del árbol por generación para todas las corridas
fig2 = go.Figure()
fig2.add_trace(go.Scatter(x=list(range(N_GEN)), y=avg_tree_size_per_gen2,␣
 ↪mode='lines', name='Tamaño promedio de árbol'))
fig2.update_layout(title='Tamaño promedio de árbol por generación (promedio␣
 ↪sobre 10 corridas)',
                   xaxis_title='Generación',
                   yaxis_title='Tamaño promedio de árbol')
fig2.show()
```





Al introducir la función de multiplicación, observamos cambios significativos en el comportamiento del algoritmo. El promedio terminales por generación es mas estable al igual que el tamaño promedio de los árboles, que ademas es considerablemente menor que en el Experimento 1.

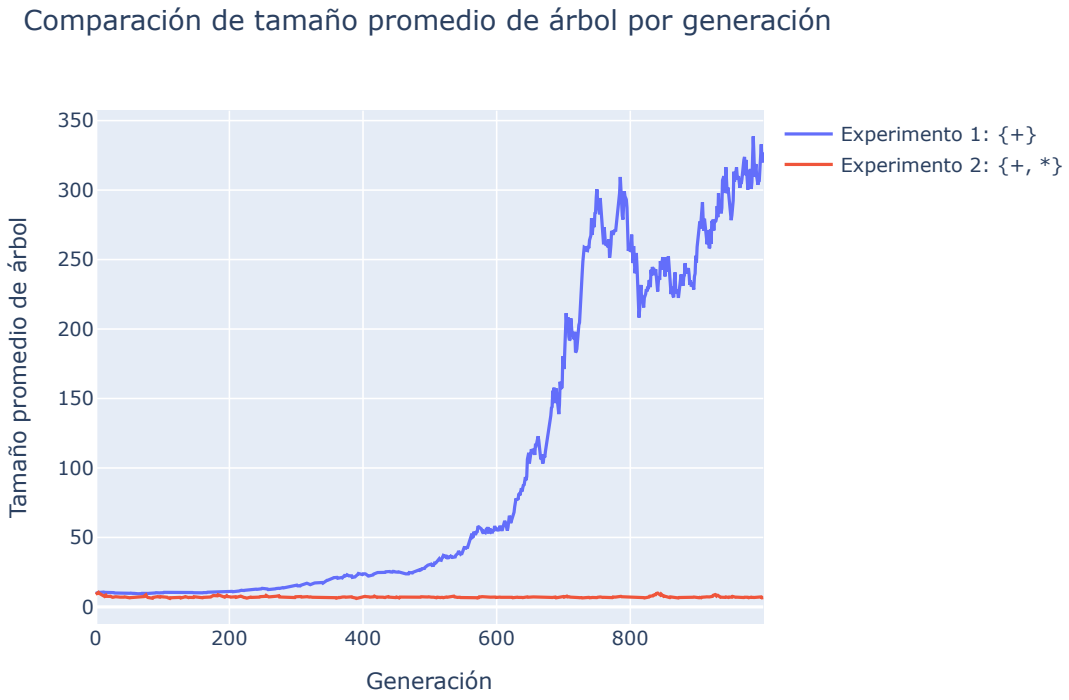
Los terminales 0 y 1 son usados con mayor frecuencia, mientras que el terminal 4 es menos frecuente, lo que sugiere que la función de multiplicación permite construir soluciones más compactas y eficientes.

1.4 Comparación de resultados entre Experimento 1 y Experimento 2

Comparación entre configuraciones: presentar en una figura o tabla la comparación de métricas clave (tamaño promedio, mejor aptitud final, tasa de soluciones exactas) entre {+} y {+, *}.

[16]:
```
# Comparacion de tamaños de árboles entre Experimento 1 y Experimento 2

# Para eso vamos a reutilizar avg_tree_size_per_gen y avg_tree_size_per_gen2

fig = go.Figure()
fig.add_trace(go.Scatter(x=list(range(N_GEN)), y=avg_tree_size_per_gen,␣
 ↪mode='lines', name='Experimento 1: {+}'))
fig.add_trace(go.Scatter(x=list(range(N_GEN)), y=avg_tree_size_per_gen2,␣
 ↪mode='lines', name='Experimento 2: {+, *}'))
```

```
fig.update_layout(title='Comparación de tamaño promedio de árbol por␣
 ↪generación',
                  xaxis_title='Generación',
                  yaxis_title='Tamaño promedio de árbol')
fig.show()
```



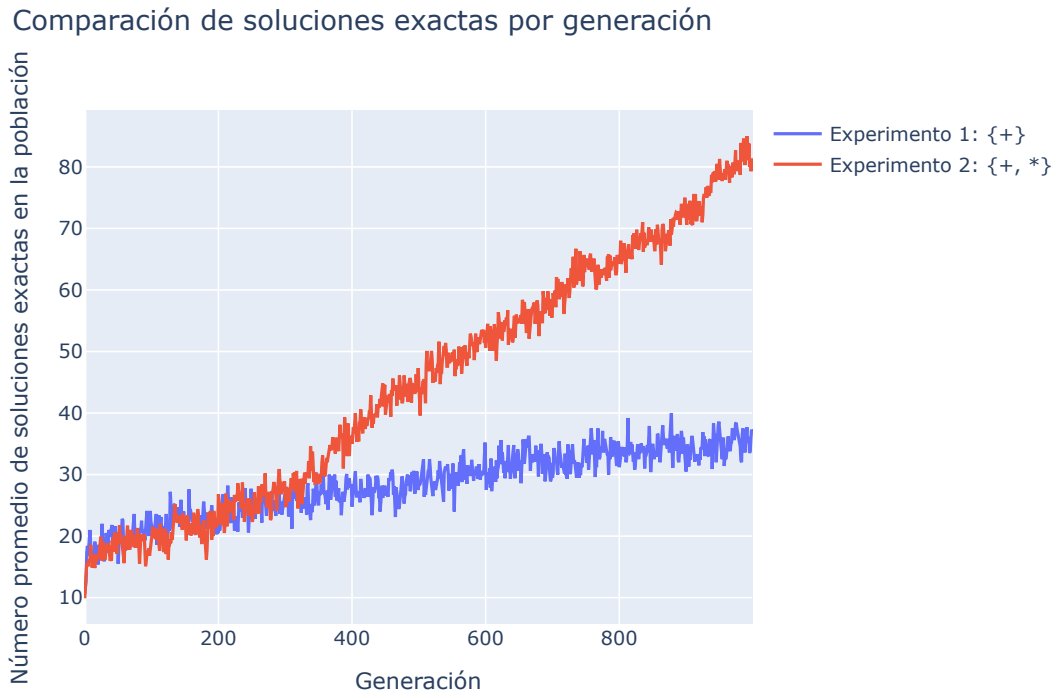
Es interesante que en el experimento 2, el tamaño promedio de los árboles es muchisimo menor que en el experimento 1. Es posible que la inclusión de la función de multiplicación permita construir soluciones más compactas y eficientes, reduciendo la necesidad de árboles grandes para alcanzar el mismo objetivo.

Es interesante

[20]:
```
# Comparemos ahora la taza de soluciones exactas (fitness 0) entre los dos␣
 ↪experimentos
# Para eso contamos, por generación y por corrida, cuántos individuos tienen␣
 ↪fitness 0

def count_exact_solutions(all_pops, toolbox, ngen, seeds):
    exact_counts = []  # exact_counts[run][gen]
    for seed in seeds:
        run_counts = []
```

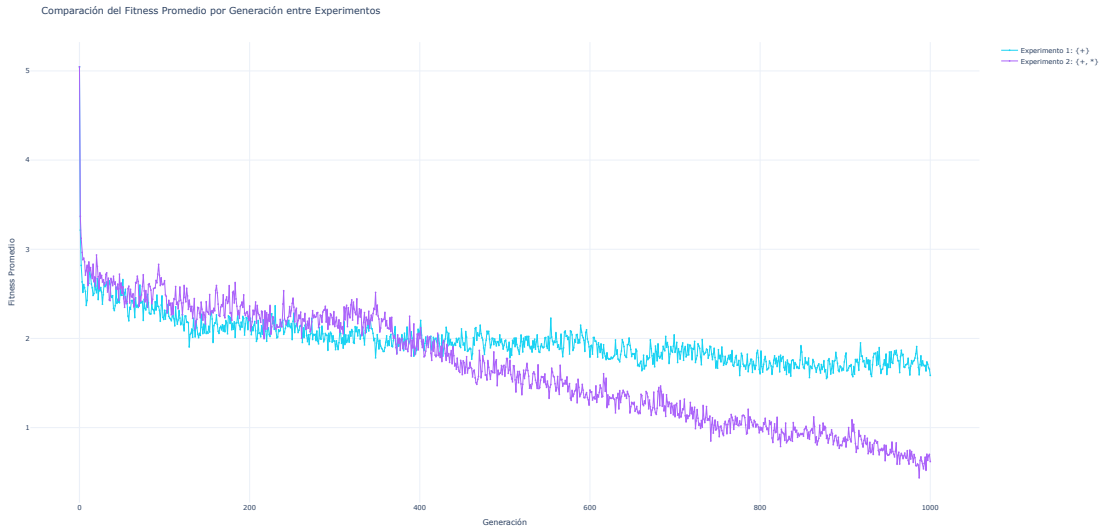
```
        for gen in range(ngen):
            pop = all_pops[seed][gen]
            count_exact = sum(
                1 for ind in pop
                if toolbox.evaluate(gp.PrimitiveTree(ind))[0] == 0
            )
            run_counts.append(count_exact)
        exact_counts.append(run_counts)
    return exact_counts

# Experimento 1
exact_counts1 = count_exact_solutions(all_pops, toolbox, N_GEN, RANDOM_SEEDS)
# Experimento 2
exact_counts2 = count_exact_solutions(all_pops2, toolbox2, N_GEN, RANDOM_SEEDS)
```

[21]:
```
# Graficar la taza de soluciones exactas por generación para ambos experimentos
avg_exact1 = np.mean(exact_counts1, axis=0)
avg_exact2 = np.mean(exact_counts2, axis=0)

fig = go.Figure()
fig.add_trace(go.Scatter(x=list(range(N_GEN)), y=avg_exact1, mode='lines',␣
 ↪name='Experimento 1: {+}'))
fig.add_trace(go.Scatter(x=list(range(N_GEN)), y=avg_exact2, mode='lines',␣
 ↪name='Experimento 2: {+, *}'))
fig.update_layout(title='Comparación de soluciones exactas por generación',
                  xaxis_title='Generación',
                  yaxis_title='Número promedio de soluciones exactas en la␣
 ↪población')
fig.show()
```



[83]:
```
# Comparación del fitness promedio por generación entre los dos experimentos

fig = go.Figure()
fig.add_trace(go.Scatter(
    x=mean_df['gen'],
    y=mean_df['avg'],
    error_x=dict(type='data', array=mean_df['std'], visible=False),
    mode='lines+markers',
    name='Experimento 1: {+}',
    line=dict(color=px.colors.qualitative.Plotly[5], width=1)
))
fig.add_trace(go.Scatter(
    x=mean_df2['gen'],
    y=mean_df2['avg'],
    error_x=dict(type='data', array=mean_df2['std'], visible=False),
    mode='lines+markers',
    name='Experimento 2: {+, *}',
    line=dict(color=px.colors.qualitative.Plotly[3], width=1)
))
fig.update_layout(
    title='Comparación del Fitness Promedio por Generación entre Experimentos',
```

```
    xaxis_title='Generación',
    yaxis_title='Fitness Promedio',
    template='plotly_white'
)
fig.update_traces(marker=dict(size=3))
fig.update_traces(line=dict(width=1))
fig.update_layout(width=2000, height=1000)
fig.show()
```



Podemos observar que la inclusión de la función de multiplicación en el Experimento 2 hace que el número promedio de soluciones exactas sea mayor en comparación con el Experimento 1, que solo utiliza la suma.

Esto implica que la diversidad funcional adicional facilita la exploración del espacio de soluciones y aumentar la probabilidad de encontrar soluciones óptimas.

1.5 Conclusiones y limitaciones

La neutralidad en Programación Genética permite la existencia de múltiples soluciones genéticas que producen el mismo resultado funcional, lo que puede contribuir a mantener la diversidad genética en la población. Este fenómeno puede proteger contra operadores destructivos y facilitar la exploración del espacio de búsqueda.

Sin embargo, la presencia de intrones y fragmentos neutrales también puede llevar al hinchamiento (bloat), donde los programas crecen en tamaño sin una mejora correspondiente en la aptitud. Esto puede resultar en una mayor complejidad computacional y dificultades para encontrar soluciones óptimas.

No obstante, la introducción de neutralidad controlada (explícita) puede tener beneficios en ciertos contextos.

- Podría mantener diversidad genética sin afectar el desempeño.
- Podría proteger contra operadores destructivos (como cruces que rompen bloques útiles).
- Podría facilitar la exploración del espacio de búsqueda, permitiendo que nuevas soluciones emerjan desde regiones neutrales.
- Y por lo tanto podría mejorar la probabilidad de éxito en problemas difíciles.

La programación genética a pesar de de ser una herramienta asombrosa para comprender muchos aspectos de la evolución y la adaptación, tiene sus limitaciones y desafíos. Mas allá de la neutralidad y el bloat, existen otros factores como la selección de funciones y terminales, la representación de los individuos, y la configuración de los parámetros del algoritmo que pueden influir significativamente en su desempeño.

Consideremos además que cuando tenemos un espacio de busqueda grande, mucchas generaciones, poblaciones grandes, el costo computacional puede ser muy alto, lo que limita la aplicabilidad práctica de estos métodos en ciertos escenarios. Ademas de ser muy sensible a la configuración de sus parámetros, lo que puede requerir una afinación cuidadosa para obtener resultados óptimos en diferentes problemas.

La interpretabilidad de los programas generados también puede ser un desafío, especialmente cuando los árboles crecen en tamaño y complejidad debido a los intrones y el bloat. Aquí no se si ya exista, pero estaría interesante utilizar otra tecnica de optimización para los arboles generados (como poda o simplificación) para mejorar la interpretabilidad mientras son evolucionados.

En general, lo veo poco práctico para problemas del mundo real, pero muy útil para entender conceptos evolutivos y explorar ideas en un entorno controlado, porque la idea esta de evolucionar programas esta súper bonita.