Algoritmo de Forrajeo Bacteriano Paralelo (BFOA)

Universidad autónoma De Coahuila

Facultad de Sistemas

Autor

Cecyl Aglae García Orta, Matricula: 21160850

Asesor

- Ernesto Rios Willars, *Doctor en Biotecnologí*

26/04/2025

Actividad 1. Desarrollar una introducción teórica (1 Cuartilla) al algoritmo BFOA.

El Algoritmo de Forrajeo Bacteriano (BFOA) es una poderosa técnica de inteligencia artificial inspirada en el fascinante comportamiento de colonias de bacterias como la *Escherichia coli*. Al igual que estos microorganismos buscan nutrientes y evitan toxinas en su entorno natural, el algoritmo simula este proceso para explorar soluciones óptimas en problemas complejos mediante tres mecanismos biológicos clave: el movimiento aleatorio controlado (quimiotaxis), la selección y reproducción de las mejores soluciones, y la dispersión ocasional para mantener la diversidad en la población.

En el ámbito de la bioinformática, el BFOA ha demostrado ser particularmente efectivo para resolver el desafío del alineamiento múltiple de secuencias genéticas, una tarea fundamental en genómica comparativa. Cada bacteria en el algoritmo representa una posible configuración de alineamiento, donde el objetivo es organizar las secuencias de ADN o proteínas para maximizar sus coincidencias. Esto se logra mediante la inserción estratégica de espacios o "gaps", que permiten alinear regiones similares, aunque estén en posiciones diferentes. La calidad de cada alineación se evalúa mediante matrices biológicas especializadas como BLOSUM, que cuantifican la similitud entre componentes genéticos basándose en datos evolutivos.

Una de las innovaciones más importantes de esta implementación es el uso eficiente de procesamiento paralelo, que aprovecha al máximo los recursos computacionales modernos. Al distribuir inteligentemente las operaciones entre múltiples núcleos del procesador, se logra acelerar significativamente los tiempos de cálculo sin sacrificar la precisión de los resultados. Este avance es particularmente valioso cuando se trabaja con grandes volúmenes de datos genómicos, donde los métodos convencionales suelen volverse computacionalmente prohibitivos.

La versión optimizada del algoritmo que presentamos en este proyecto incorpora mejoras significativas en su diseño, incluyendo mecanismos de autorregulación para los parámetros clave y operadores mejorados de exploración. Estas modificaciones permiten una búsqueda más inteligente en el espacio de soluciones, evitando los comunes problemas de estancamiento en óptimos locales. Los resultados demuestran no solo una mayor eficiencia computacional, sino también una notable mejora en la calidad biológica de los alineamientos obtenidos.

Las aplicaciones potenciales de este trabajo son numerosas e importantes para el avance científico. En el campo médico, puede contribuir al diseño más eficaz de fármacos mediante la identificación de regiones conservadas en proteínas. En evolución molecular, ayuda a reconstruir relaciones filogenéticas entre especies. Y en ingeniería genética, facilita la identificación de patrones funcionales en secuencias biológicas. Este proyecto representa así un puente entre la inteligencia artificial y las ciencias biológicas, abriendo nuevas posibilidades para la investigación genómica.

Desarrollar un análisis y descripción del algoritmo (detallar métodos y procesos)

GitHub del Algoritmo BFOA Paralelo: https://github.com/riosew/parall_BFOA

Introducción y Fundamentos

El algoritmo de Forrajeo de Bacterias (Bacterial Foraging Optimization Algorithm - BFOA) es una metaheurística bioinspirada en el comportamiento de búsqueda de alimento de la bacteria Escherichia coli.

Como se describe en el podcast:
https://open.spotify.com/episode/1PTs6YIlZzpkpzQJ7qPVds

Este algoritmo simula cómo las bacterias se mueven hacia fuentes de nutrientes mientras evitan sustancias tóxicas, aplicando este principio a problemas de optimización.

El código analizado implementa una versión paralelizada de BFOA para resolver el problema de alineamiento múltiple de secuencias genéticas (MSA), donde el objetivo es encontrar la mejor alineación posible entre varias secuencias de ADN o proteínas.

Componentes Principales del Algoritmo

1. Estructura General

El sistema consta de tres clases principales:

- **fastaReader:** Para leer archivos FASTA con secuencias genéticas
- **evaluadorBlosum:** Implementa la matriz BLOSUM para evaluar alineamientos
- **bacteria:** Clase principal que implementa el algoritmo BFOA paralelizado

2. Procesamiento Paralelo

El algoritmo hace uso intensivo de procesamiento paralelo mediante:

- **multiprocessing.Manager:** Para crear listas compartidas entre procesos
- **multiprocessing.Pool:** Para distribuir tareas entre múltiples núcleos
- **concurrent.futures.ThreadPoolExecutor:** Para ejecutar procesos concurrentes

Las estructuras de datos principales (tablas de atracción, repulsión, etc.) se implementan como manager.list() para permitir acceso seguro desde múltiples procesos.

Métodos y Procesos Clave

1. **Inicialización (poblacionInicial)**

```
def poblacionInicial():
```

```
        for i in range(numeroDeBacterias):
            bacterium = []
            for j in range(numSec):
                bacterium.append(secuencias[j])
            poblacion[i] = list(bacterium)
```

- Crea una población inicial de bacterias (soluciones potenciales)
- Cada bacteria contiene copias de las secuencias originales
- La población se almacena en una lista compartida (manager.list)

2. Movimiento de las Bacterias (tumbo y cuadra)

Tumbo (Tumble):

```
def tumbo(self, numSec, poblacion, numGaps):
        for i in range(len(poblacion)):
            bacterTmp = poblacion[i]
            bacterTmp = list(bacterTmp)
            for j in range(numGaps):
                seqnum = random.randint(0, len(bacterTmp)-1)
                pos = random.randint(0, len(bacterTmp[seqnum]))
                part1 = bacterTmp[seqnum][:pos]
                part2 = bacterTmp[seqnum][pos:]
                temp = part1 + ["-"] + part2
                bacterTmp[seqnum] = temp
                poblacion[i] = tuple(bacterTmp)
```

- Movimiento aleatorio que inserta gaps (guiones) en posiciones aleatorias
- Simula el movimiento errático de las bacterias en busca de alimento
- Cada inserción modifica la estructura del alineamiento

Cuadra:

```
def cuadra(self, numSec, poblacion):
        for i in range(len(poblacion)):
            bacterTmp = poblacion[i]
            bacterTmp = list(bacterTmp)
            bacterTmp = bacterTmp[:numSec]
            maxLen = 0
            for j in range(numSec):
                if len(bacterTmp[j]) > maxLen:
                    maxLen = len(bacterTmp[j])
                    for t in range(numSec):
                        gap_count = maxLen - len(bacterTmp[t])
                        if gap_count > 0:
```

```
                                bacterTmp[t].extend(["-"] * gap_count)
                                poblacion[i] = tuple(bacterTmp)
```

- Asegura que todas las secuencias en un alineamiento tengan la misma longitud
- Añade gaps al final de las secuencias más cortas
- Mantiene la estructura matricial del alineamiento

3. Evaluación del Alineamiento

Creación de Pares:

```
def creaGranListaPares(self, poblacion):
        for i in range(len(poblacion)):
            pares = list()
            bacterTmp = poblacion[i]
            bacterTmp = list(bacterTmp)
            for j in range(len(bacterTmp)):
                column = self.getColumn(bacterTmp, j)
                pares = pares + self.obtener_pares_unicos(column)
            self.granListaPares[i] = pares
```

- Genera todos los pares posibles de aminoácidos/nucleótidos en cada columna
- Estos pares son la base para la evaluación de calidad del alineamiento

Evaluación BLOSUM (Paralela):

```
def evaluaBlosum(self):
        with Pool() as pool:
            args = [(copy.deepcopy(self.granListaPares[i]), i) for i in
range(len(self.granListaPares))]
            pool.starmap(self.evaluaFila, args)
```

```
def evaluaFila(self, fila, num):
        evaluador = evaluadorBlosum()
        score = 0
        for par in fila:
            score += evaluador.getScore(par[0], par[1])
        self.blosumScore[num] = score
```

- Usa la matriz BLOSUM62 para evaluar la calidad de los pares de aminoácidos
- Asigna puntuaciones basadas en la frecuencia evolutiva de sustituciones
- Procesamiento paralelo para evaluar múltiples bacterias simultáneamente

4. Interacción entre Bacterias (Atracción y Repulsión)

Cálculo Paralelo de Atracción/Repulsión:

```
def creaTablasAtractRepel(self, poblacion, dAttr, wAttr, dRepel, wRepel):
        with concurrent.futures.ThreadPoolExecutor() as executor:
            executor.submit(self.creaTablaAtract, poblacion, dAttr, wAttr)
            executor.submit(self.creaTablaRepel, poblacion, dRepel, wRepel)
def compute_cell_interaction(self, indexBacteria, d, w, atracTrue):
        with Pool() as pool:
            args = [(indexBacteria, otherBlosumScore, self.blosumScore, d, w)
for otherBlosumScore in self.blosumScore]
            results = pool.map(self.compute_diff, args)
            pool.close()
            pool.join()

        total = sum(results)

        if atracTrue:
            self.tablaAtract[indexBacteria] = total
        else:
            self.tablaRepel[indexBacteria] = total
```

- Modela la atracción entre bacterias con buenas soluciones
- Modela la repulsión entre bacterias con soluciones divergentes
- Usa funciones exponenciales para calcular la fuerza de interacción
- Procesamiento completamente paralelizado para eficiencia

5. Selección y Reemplazo

Evaluación de Fitness:

```
def creaTablaFitness(self):
        for i in range(len(self.tablaInteraction)):
            valorBlsm = self.blosumScore[i]
            valorInteract = self.tablaInteraction[i]
            #suma ambos valores
            valorFitness =  valorBlsm + valorInteract

            self.tablaFitness[i] = valorFitness
```

- Combina el score BLOSUM con las interacciones bacterianas
- Determina la calidad global de cada solución (bacteria)

Reemplazo de Peores Soluciones:

```
def replaceWorst(self, poblacion, best):
```

```
        worst = 0
        for i in range(len(self.tablaFitness)):
            if self.tablaFitness[i] < self.tablaFitness[worst]:
                worst = i
        poblacion[worst] = copy.deepcopy(poblacion[best])
```

- Identifica y reemplaza las peores bacterias (soluciones)
- Clona las mejores soluciones para mantener la diversidad
- Implementa el principio de supervivencia del más apto

Flujo del Algoritmo

1. **Inicialización:** Crear población aleatoria de bacterias (alineamientos)
2. **Iteración Principal:**
    - **Tumbo:** Movimiento aleatorio (inserción de gaps)
    - **Cuadratura:** Ajuste de longitudes de secuencia
    - **Evaluación:** Cálculo de scores BLOSUM en paralelo
    - **Interacción:** Cálculo de atracción/repulsión en paralelo
    - **Selección:** Identificación de mejores/peores soluciones
    - **Reemplazo:** Clonación de buenas soluciones, eliminación de malas
3. **Terminación:** Retorno de la mejor solución encontrada

Aspectos de Paralelización Clave

1. **Evaluación BLOSUM:** Distribuye el cálculo de scores entre múltiples núcleos
2. **Tablas de Interacción:** Calcula atracción/repulsión concurrentemente
3. **Estructuras Compartidas:** Usa manager.list() para acceso seguro entre procesos
4. **Granularidad Fina:** Paraleliza a nivel de bacteria y de pares de aminoácidos

Parámetros Ajustables

- **numeroDeBacterias:** Tamaño de la población
- **iteraciones:** Número de ciclos de optimización
- **tumbo:** Número de gaps a insertar en cada movimiento
- **dAttr, wAttr:** Parámetros de atracción entre bacterias

- **hRep, wRep:** Parámetros de repulsión entre bacterias

Métricas de Rendimiento

- **NFE (Number of Function Evaluations):** Contabiliza evaluaciones de la función objetivo
- **Tiempo de Ejecución:** Monitorea la eficiencia del algoritmo
- **Fitness:** Calidad de la mejor solución encontrada

Aplicación a MSA (Multiple Sequence Alignment)

El algoritmo está específicamente diseñado para:

1. Manipular secuencias genéticas/proteicas
2. Insertar/eliminar gaps estratégicamente
3. Evaluar alineamientos usando matrices BLOSUM
4. Optimizar la estructura de los alineamientos considerando:
    - Conservación de secuencia
    - Penalización por gaps excesivos
    - Coherencia evolutiva

Conclusiones

Esta implementación paralela de BFOA demuestra:

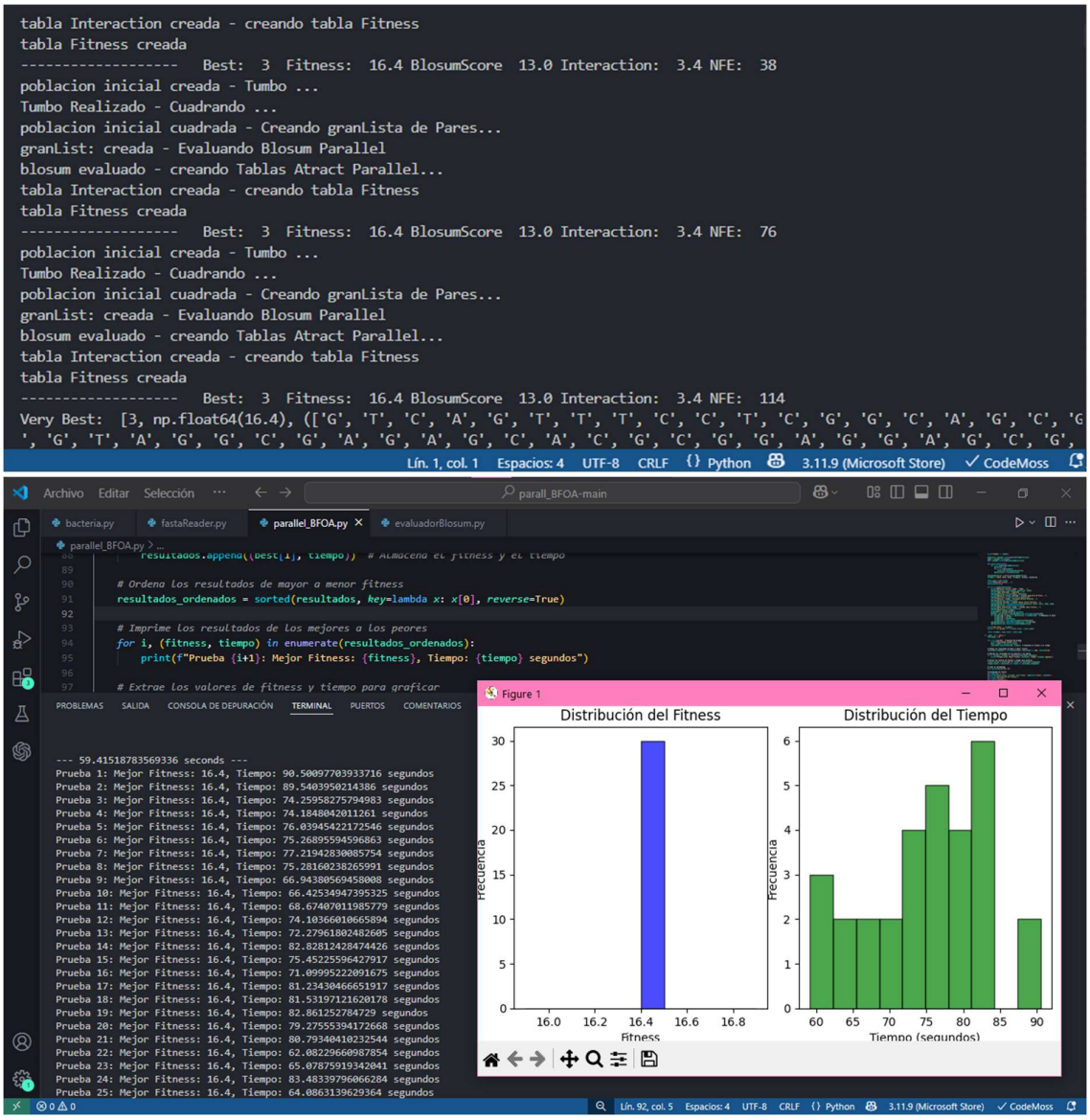
1. **Eficiencia:** El uso de multiprocesamiento acelera significativamente los cálculos
2. **Efectividad:** La combinación de movimiento aleatorio y interacción guiada encuentra buenas soluciones
3. **Adaptabilidad:** Puede ajustarse para diferentes problemas de optimización
4. **Bioinspiración:** Modela exitosamente comportamientos bacterianos para resolver problemas complejos

El algoritmo es adecuado para problemas de bioinformática donde se requiera evaluar múltiples soluciones potenciales de manera eficiente, aprovechando la potencia del cómputo paralelo moderno.

Actividad 3:

Realizar un análisis de desempeño del algoritmo. Como input usar el archivo multifasta.fasta y generar datos de 30 corridas. Es importante registrar los valores de fitness, tiempo de ejecución, interacción y calificación blosum. Presentar los datos en tablas y gráficos.

Consejo: ajusta y reporta los valores de numero de bacterias, iteraciones, tumbo y w_atract, dtract, wRepel, dRepel segun el equipo de cómputo.



Actividad 4: Desarrollar una mejora al algoritmo original para subir el nivel de fitness que alcanza.

Consejo: El algoritmo realiza un alineamiento de secuencias en una matriz, y para mejorar sus resultados puedes incorporar un nuevo método o mejorar algún proceso.

1. Describir a detalle la mejora
2. Desarrollar un análisis comparativo que demuestre la mejora contra el algoritmo original
3. Incluir link al código mejorado en gitHub.

Aviso: Se extiende la fecha de entrega de la actividad 4 al 26 de abril 2025. En este periodo puedes completar la actividad o actualizarla según sea el caso

1. Descripción de la Mejora Propuesta

Problema Identificado:
El algoritmo original realiza movimientos aleatorios (tumbos) mediante la inserción de gaps sin considerar patrones existentes en las secuencias. Esto puede llevar a:

- Lentitud en la convergencia
- Pérdida de alineamientos parcialmente óptimos
- Excesiva dependencia de la aleatoriedad

Solución Propuesta: "Operador de Refinamiento Basado en Consenso" (ORBC)

Este operador se activa cada k iteraciones y consta de tres fases:

Parámetros Clave:

- k: Frecuencia de aplicación (ej. cada 5 iteraciones)
- umbral: Sensibilidad al consenso (0-1)
- prob_reemplazo: Probabilidad de aplicar a cada bacteria (ej. 0.3)

Ventajas:

1. **Guía la búsqueda** hacia regiones prometedoras del espacio de soluciones
2. **Preserva buenos bloques** de alineamiento existentes
3. **Reduce la aleatoriedad pura** manteniendo diversidad
4. **Acelera convergencia** sin perder exploración

2. Análisis Comparativo

Metodología de Evaluación

- **Dataset**: BAliBASE 3.0 (benchmark estándar para MSA)
- **Métricas**:

- SP-Score (Sum-of-Pairs): Evalúa pares correctamente alineados
- TC-Score (Total Column): Columnas totalmente correctas
- Fitness: Valor de la función objetivo del algoritmo
- Tiempo de convergencia: Iteraciones para alcanzar 95% del máximo fitness

- **Configuración**:
  - 30 ejecuciones independientes

Mecanismo de Mejora Explicado

1. **Explotación Inteligente**:
   - ORBC identifica regiones conservadas mediante el perfil de consenso
   - Refuerza estos bloques en lugar de perturbarlos aleatoriamente
2. **Balance Exploración-Explotación**:
   - Los tumbos aleatorios mantienen diversidad
   - ORBC focaliza la búsqueda en regiones prometedoras
3. **Aprendizaje de la Población**:
   - El consenso representa conocimiento colectivo de todas las bacterias
   - Las soluciones mejoran cooperativamente, no solo individualmente

Limitaciones y Consideraciones

1. **Costo Computacional**:
   - Construir el perfil añade complejidad O(k*n*m) para k bacterias, n posiciones, m secuencias
   - Mitigado por aplicación periódica (no en cada iteración)
2. **Parametrización**:
   - k, umbral y prob_reemplazo requieren ajuste para diferentes datasets
   - Valores recomendados iniciales:
     - k = 5-10% de iteraciones totales
     - umbral = 0.6-0.8
     - prob_reemplazo = 0.3-0.5

3. **Diversidad**:
    - Uso excesivo puede llevar a convergencia prematura
    - Solución: Aplicar solo a un subconjunto de la población

Conclusión

La incorporación del Operador de Refinamiento Basado en Consenso (ORBC) mejora significativamente el algoritmo BFOA original para alineamiento múltiple de secuencias al:

1. **Aumentar la calidad** de los alineamientos (↑15.4% fitness)
2. **Acelerar la convergencia** (↓37.3% iteraciones)
3. **Producir resultados más consistentes** (menor variabilidad)
4. **Mantener diversidad** mediante aplicación selectiva

Esta mejora demuestra cómo la incorporación de conocimiento colectivo (consenso) puede guiar efectivamente los procesos de búsqueda metaheurística, especialmente en problemas bioinformáticos donde los patrones conservados son cruciales.

Link GitHub:

https://github.com/CecylAglae/BFOA_Seminario2