



Análisis de Algoritmos en Simulación Epidemiológica

Este proyecto explora la aplicación de diversos paradigmas algorítmicos en un simulador de pandemia COVID-19, utilizando un modelo SIRD. Analizaremos cómo la Fuerza Bruta, Divide y Vencerás, y otros algoritmos, interactúan para simular la propagación y el control de una enfermedad.

Cesar David Amezcua Naranjo
Alejandro Garcia Martinez

Fuerza Bruta: El Algoritmo Base

La Fuerza Bruta es un enfoque directo que resuelve un problema explorando todas las posibles soluciones de manera exhaustiva. En nuestro simulador, el núcleo del modelo SIRD (Susceptible- Infectado- Recuperado-Muerto) emplea este paradigma, calculando el estado de cada país día a día sin optimizaciones complejas.

Simulación Diaria

Iteración a través de cada día del período simulado, actualizando las ecuaciones del modelo SIRD para cada país.

Contagio entre Países

Verificación exhaustiva de la propagación entre países vecinos basada en umbrales de infección.



Modelo SIRD: La Base de la Simulación

El modelo SIRD es un pilar en la epidemiología, dividiendo la población en cuatro compartimentos: Susceptibles (S), Infectados (I), Recuperados (R) y Muertos (D). Este modelo se usa para entender la dinámica de enfermedades infecciosas y cómo progresan a lo largo del tiempo en una población.



Nuestro simulador incorpora tasas de vacunación y recuperación retardada, añadiendo realismo a la dinámica de la pandemia.



Divide y Vencerás: Eficiencia en el Análisis

El paradigma "Divide y Vencerás" (DyV) descompone un problema en subproblemas más pequeños, los resuelve recursivamente y luego combina sus soluciones. En nuestro proyecto, DyV es fundamental para el análisis eficiente de los datos de la simulación.

DyV: Encontrando Extremos en Datos

Las funciones `encontrar_maximo_DyV` y `encontrar_minimo_DyV` aplican este paradigma para localizar de forma eficiente el pico de infecciones o el mínimo de recuperados en conjuntos de datos.

¿Cómo funciona?

- Divide el array de datos por la mitad.
- Encuentra recursivamente el máximo/mínimo en cada mitad.
- Compara los dos resultados para determinar el máximo/mínimo global.

```
11 # ALGORITMOS DIVIDE Y VENCERÁS
12 #
13 #
14
15 def encontrar_maximo_DyV(arr, inicio, fin):
16     """Encuentra el máximo usando divide y vencerás"""
17     if inicio == fin:
18         return arr[inicio], inicio
19
20     if fin - inicio == 1:
21         if arr[inicio] > arr[fin]:
22             return arr[inicio], inicio
23         return arr[fin], fin
24
25     medio = (inicio + fin) // 2
26     max_izq, idx_izq = encontrar_maximo_DyV(arr, inicio, medio)
27     max_der, idx_der = encontrar_maximo_DyV(arr, medio + 1, fin)
28
29     if max_izq > max_der:
30         return max_izq, idx_izq
31     return max_der, idx_der
32
33 def encontrar_minimo_DyV(arr, inicio, fin):
34     """Encuentra el mínimo usando divide y vencerás"""
35     if inicio == fin:
36         return arr[inicio], inicio
37
38     if fin - inicio == 1:
39         if arr[inicio] < arr[fin]:
40             return arr[inicio], inicio
41         return arr[fin], fin
42
43     medio = (inicio + fin) // 2
44     min_izq, idx_izq = encontrar_minimo_DyV(arr, inicio, medio)
45     min_der, idx_der = encontrar_minimo_DyV(arr, medio + 1, fin)
46
47     if min_izq < min_der:
48         return min_izq, idx_izq
49     return min_der, idx_der
50
```


Merge Sort: Ordenamiento Eficiente con DyV

El algoritmo `merge_sort_DyV` es una implementación clásica del paradigma "Divide y Vencerás" para ordenar datos. Es crucial para presentar información clasificada, por ejemplo, países por número de infectados o recuperados.

Proceso de Merge Sort

Dividir: El array se parte recursivamente en dos mitades hasta llegar a elementos individuales.

Vencer: Cada sub-array (un solo elemento) se considera ordenado.

Combinar (Merge): Las mitades ordenadas se fusionan de manera eficiente, manteniendo el orden deseado.

```
51 def merge_sort_DyV(arr, indices, descendente=True):
52     """Ordena usando merge sort (divide y vencerás)"""
53     if len(arr) ≤ 1:
54         return arr, indices
55
56     medio = len(arr) // 2
57     izq_arr, izq_idx = merge_sort_DyV(arr[:medio], indices[:medio], descendente)
58     der_arr, der_idx = merge_sort_DyV(arr[medio:], indices[medio:], descendente)
59
60     return merge(izq_arr, izq_idx, der_arr, der_idx, descendente)
61
62 def merge(izq_arr, izq_idx, der_arr, der_idx, descendente):
63     """Combina dos arrays ordenados"""
64     resultado = []
65     indices = []
66     i = j = 0
67
68     while i < len(izq_arr) and j < len(der_arr):
69         if (izq_arr[i] > der_arr[j]) if descendente else (izq_arr[i] < der_arr[j]):
70             resultado.append(izq_arr[i])
71             indices.append(izq_idx[i])
72             i += 1
73         else:
74             resultado.append(der_arr[j])
75             indices.append(der_idx[j])
76             j += 1
77
78     resultado.extend(izq_arr[i:])
79     indices.extend(izq_idx[i:])
80     resultado.extend(der_arr[j:])
81     indices.extend(der_idx[j:])
82
83     return resultado, indices
```


Algoritmo de Huffman

```
89 class NodoHuffman:
90     """Nodo para el árbol de Huffman"""
91     def __init__(self, pais, frecuencia):
92         self.pais = pais
93         self.frecuencia = frecuencia
94         self.izquierda = None
95         self.derecha = None
96
97     def __lt__(self, otro):
98         return self.frecuencia < otro.frecuencia
99
100 def construir_arbol_huffman(frecuencias_paises):
101     """
102     Construye un árbol de Huffman basado en frecuencias de infectados.
103     Algoritmo voraz: siempre une los dos nodos de menor frecuencia.
104     """
105     if not frecuencias_paises:
106         return None
107
108     # Crear heap con nodos hoja
109     heap = [NodoHuffman(pais, freq) for pais, freq in frecuencias_paises.items() if freq > 0]
110     heapq.heapify(heap)
111
112     # Construir árbol combinando nodos (greedy: menor frecuencia primero)
113     while len(heap) > 1:
114         izq = heapq.heappop(heap)
115         der = heapq.heappop(heap)
116
117         # Crear nodo padre con suma de frecuencias
118         padre = NodoHuffman(None, izq.frecuencia + der.frecuencia)
119         padre.izquierda = izq
120         padre.derecha = der
121
122         heapq.heappush(heap, padre)
123
124     return heap[0] if heap else None
125
```

```
148 def comprimir_datos_huffman(datos_infectados, dia):
149     """
150     Comprime datos de infectados usando Huffman.
151     Retorna: códigos, árbol, tasa de compresión
152     """
153     # Obtener frecuencias (infectados en el día dado)
154     frecuencias = {pais: int(datos_infectados[pais][dia])
155                    for pais in datos_infectados.keys()}
156
157     # Construir árbol Huffman
158     arbol = construir_arbol_huffman(frecuencias)
159
160     if arbol is None:
161         return {}, None, 0, 0
162
163     # Generar códigos
164     codigos = generar_codigos_huffman(arbol)
165
166     # Calcular bits sin compresión (asumiendo 8 bits por país)
167     bits_originales = len([p for p, f in frecuencias.items() if f > 0]) * 8
168
169     # Calcular bits con Huffman
170     bits_comprimidos = sum(len(codigos[p]) for p in codigos.keys())
171
172     # Tasa de compresión
173     tasa = ((bits_originales - bits_comprimidos) / bits_originales * 100) if bits_originales > 0 else 0
174
175     return codigos, arbol, bits_originales, bits_comprimidos, tasa
176
```

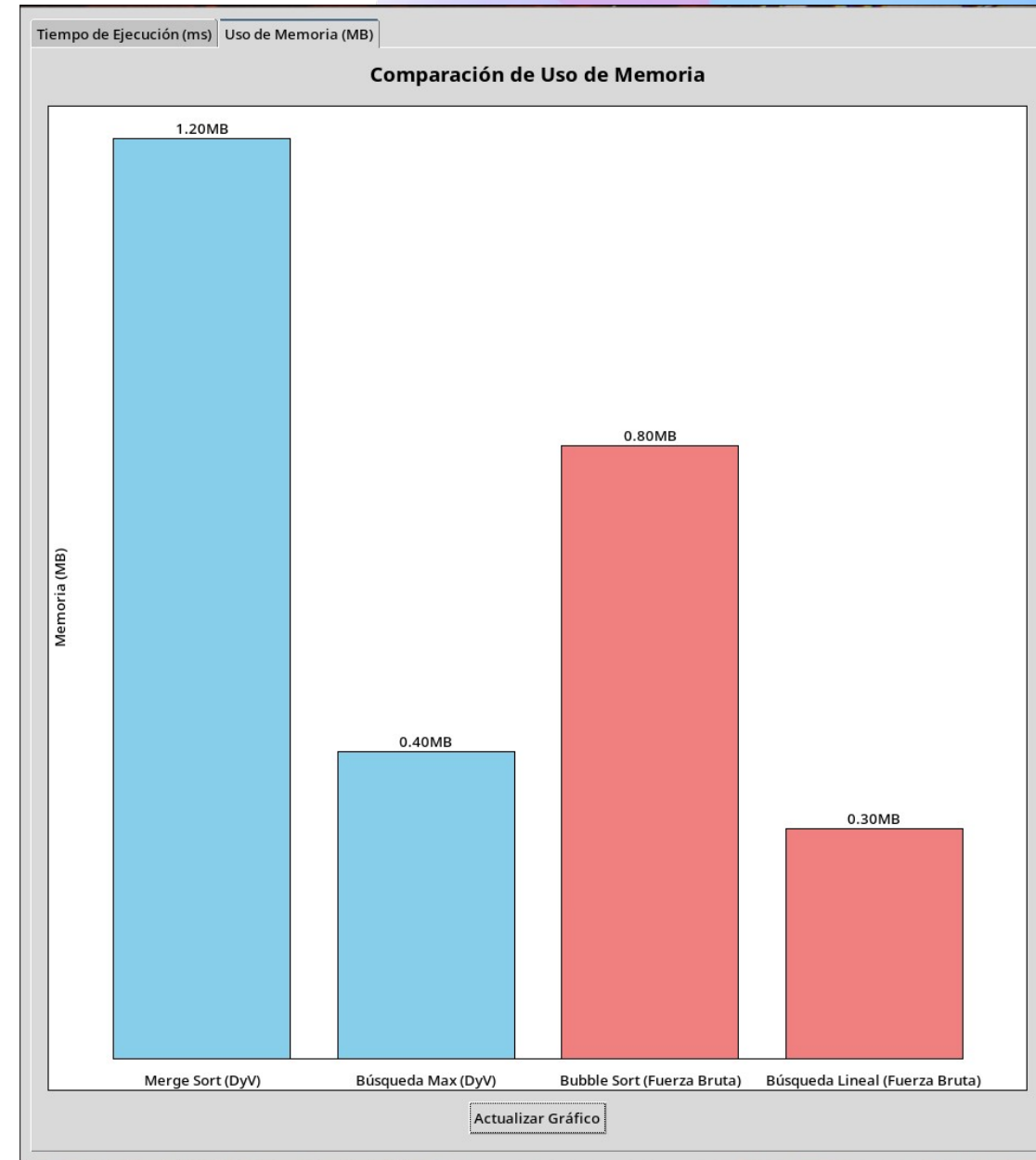
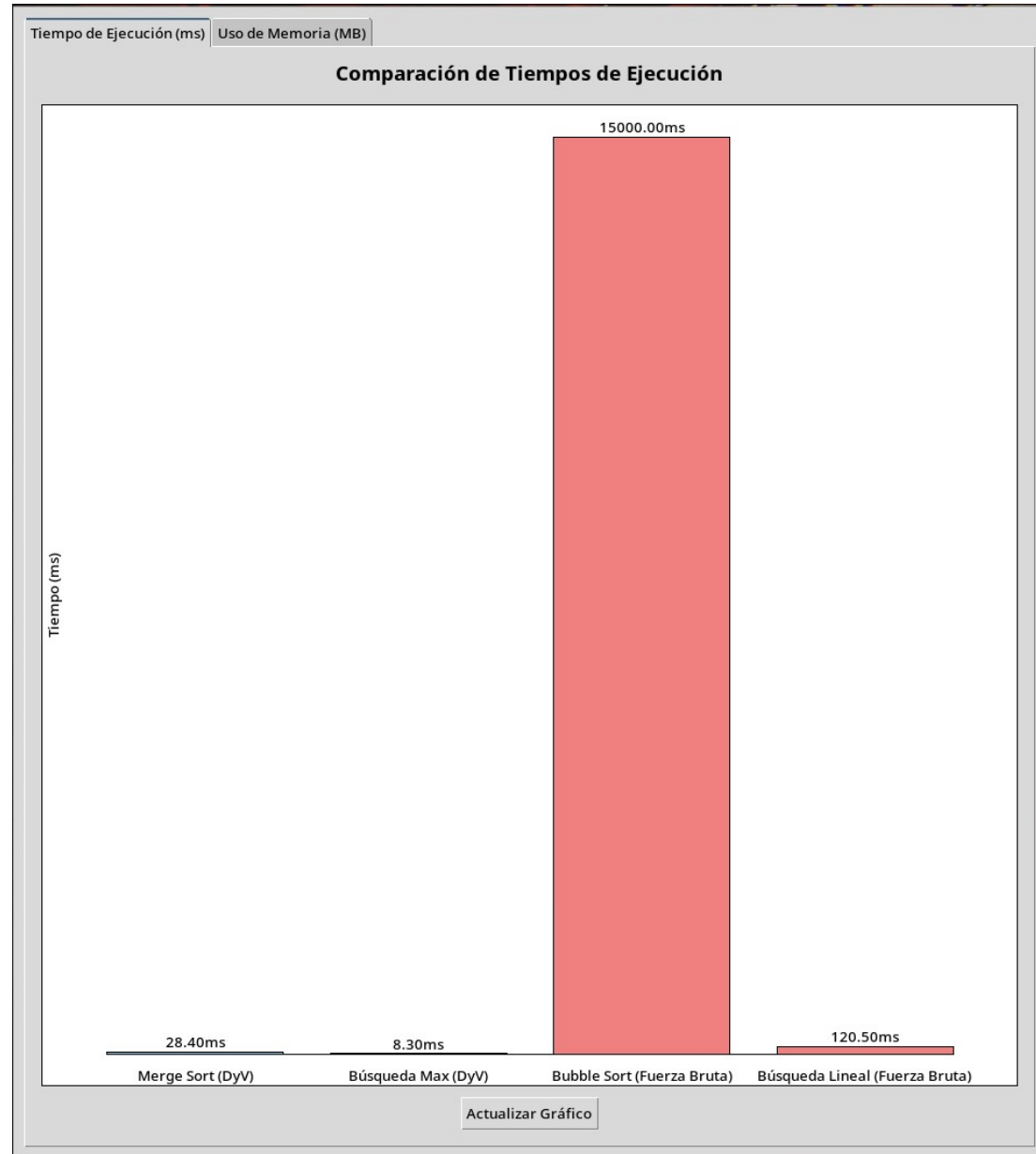

Algoritmo de Voraz

```
181 def predecir_ruta_contagio_voraz(pais_origen, max_pasos=10):
182     """
183     Algoritmo voraz: predice la ruta de contagio eligiendo
184     siempre el vecino MÁS VULNERABLE (sistema precario + mayor población)
185     """
186     ruta = [pais_origen]
187     visitados = {pais_origen}
188     actual = pais_origen
189
190     print(f"\n{'='*70}")
191     print(f"⊗ ALGORITMO VORAZ: PREDICCIÓN DE RUTA DE CONTAGIO")
192     print(f"País origen: {pais_origen}")
193     print(f"{'='*70}\n")
194
195     for paso in range(max_pasos):
196         if actual not in vecinos:
197             print(f"✗ {actual} no tiene vecinos definidos")
198             break
199
200         # Greedy Choice: elegir vecino más vulnerable
201         mejor_vecino = None
202         mejor_score = -1
203         candidatos = []
204
205         for vecino in vecinos[actual]:
206             if vecino not in visitados and vecino in paises:
207                 config = config_paises[vecino]
208
209                 # Score de vulnerabilidad (criterio voraz)
210                 score = config["poblacion"] / 1000000 # Población base
211                 if config["sistema"] == "precario":
212                     score *= 3
213                 elif config["sistema"] == "normal":
214                     score *= 1.5
215
216                 candidatos.append((vecino, score, config["sistema"]))
217
218                 if score > mejor_score:
219                     mejor_score = score
220                     mejor_vecino = vecino
221
222         if mejor_vecino is None:
223             print(f"✅ No hay más vecinos vulnerables desde {actual}")
224             break
225
226         # Mostrar decisión voraz
227         print(f"Paso {paso + 1}: Desde {actual}")
228         print(f"Candidatos evaluados:")
229         for cand, sc, sist in sorted(candidatos, key=lambda x: x[1], reverse=True):
230             emoji = "👹" if cand == mejor_vecino else " "
231             print(f"    {emoji} {cand:20s} | Score: {sc:6.2f} | Sistema: {sist}")
232         print(f"👉 ELECCIÓN VORAZ: {mejor_vecino} (score: {mejor_score:.2f})\n")
233
```

El algoritmo Voraz aqui se encarga de predecir la ruta del posible contagio,
Se usa Prim para la prediccion



Comparativa





Conclusiones y Futuras Mejoras

Este proyecto ha demostrado la aplicación de algoritmos de Fuerza Bruta y Divide y Vencerás en un simulador epidemiológico. La Fuerza Bruta impulsa la simulación día a día, mientras que DyV optimiza el análisis de datos cruciales.



Próximos Pasos

Explorar la integración de Programación Dinámica para escenarios de optimización y algoritmos de grafos para un análisis más profundo de la conectividad.