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.

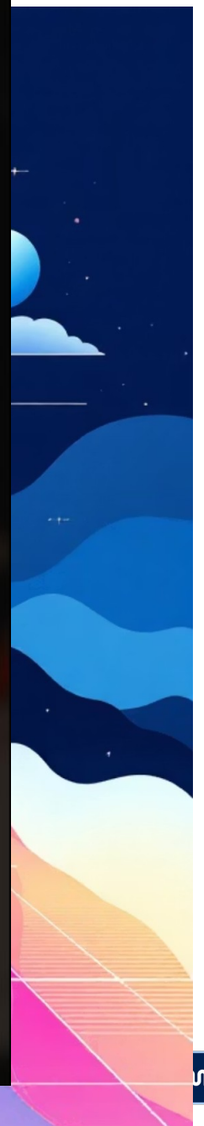
DyV: Encontrando Extremos en Datos

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.**

```python
# ═══════════════════════════
# ALGORITMOS DIVIDE Y VENCERÁS
# ═══════════════════════════

def encontrar_maximo_DyV(arr, inicio, fin):
    """Encuentra el máximo usando divide y vencerás"""
    if inicio == fin:
        return arr[inicio], inicio

    if fin - inicio == 1:
        if arr[inicio] > arr[fin]:
            return arr[inicio], inicio
        return arr[fin], fin

    medio = (inicio + fin) // 2
    max_izq, idx_izq = encontrar_maximo_DyV(arr, inicio, medio)
    max_der, idx_der = encontrar_maximo_DyV(arr, medio + 1, fin)

    if max_izq > max_der:
        return max_izq, idx_izq
    return max_der, idx_der

def encontrar_minimo_DyV(arr, inicio, fin):
    """Encuentra el mínimo usando divide y vencerás"""
    if inicio == fin:
        return arr[inicio], inicio

    if fin - inicio == 1:
        if arr[inicio] < arr[fin]:
            return arr[inicio], inicio
        return arr[fin], fin

    medio = (inicio + fin) // 2
    min_izq, idx_izq = encontrar_minimo_DyV(arr, inicio, medio)
    min_der, idx_der = encontrar_minimo_DyV(arr, medio + 1, fin)

    if min_izq < min_der:
        return min_izq, idx_izq
    return min_der, idx_der
```



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.

```python
def merge_sort_DyV(arr, indices, descendente=True):
    """Ordena usando merge sort (divide y vencerás)"""
    if len(arr) <= 1:
        return arr, indices

    medio = len(arr) // 2
    izq_arr, izq_idx = merge_sort_DyV(arr[:medio], indices[:medio], descendente)
    der_arr, der_idx = merge_sort_DyV(arr[medio:], indices[medio:], descendente)

    return merge(izq_arr, izq_idx, der_arr, der_idx, descendente)

def merge(izq_arr, izq_idx, der_arr, der_idx, descendente):
    """Combina dos arrays ordenados"""
    resultado = []
    indices = []
    i = j = 0

    while i < len(izq_arr) and j < len(der_arr):
        if (izq_arr[i] > der_arr[j]) if descendente else (izq_arr[i] < der_arr[j]):
            resultado.append(izq_arr[i])
            indices.append(izq_idx[i])
            i += 1
        else:
            resultado.append(der_arr[j])
            indices.append(der_idx[j])
            j += 1

    resultado.extend(izq_arr[i:])
    indices.extend(izq_idx[i:])
    resultado.extend(der_arr[j:])
    indices.extend(der_idx[j:])

    return resultado, indices
```

Algoritmo de Huffman

```python
class NodoHuffman:
    """Nodo para el árbol de Huffman"""
    def __init__(self, pais, frecuencia):
        self.pais = pais
        self.frecuencia = frecuencia
        self.izquierda = None
        self.derecha = None

    def __lt__(self, otro):
        return self.frecuencia < otro.frecuencia

def construir_arbol_huffman(frecuencias_paises):
    """
    Construye un árbol de Huffman basado en frecuencias de infectados.
    Algoritmo voraz: siempre une los dos nodos de menor frecuencia.
    """
    if not frecuencias_paises:
        return None

    # Crear heap con nodos hoja
    heap = [NodoHuffman(pais, freq) for pais, freq in frecuencias_paises.items() if freq > 0]
    heapq.heapify(heap)

    # Construir árbol combinando nodos (greedy: menor frecuencia primero)
    while len(heap) > 1:
        izq = heapq.heappop(heap)
        der = heapq.heappop(heap)

        # Crear nodo padre con suma de frecuencias
        padre = NodoHuffman(None, izq.frecuencia + der.frecuencia)
        padre.izquierda = izq
        padre.derecha = der

        heapq.heappush(heap, padre)

    return heap[0] if heap else None
```

```python
def comprimir_datos_huffman(datos_infectados, dia):
    """
    Comprime datos de infectados usando Huffman.
    Retorna: códigos, árbol, tasa de compresión
    """
    # Obtener frecuencias (infectados en el día dado)
    frecuencias = {pais: int(datos_infectados[pais][dia])
                   for pais in datos_infectados.keys()}

    # Construir árbol Huffman
    arbol = construir_arbol_huffman(frecuencias)

    if arbol is None:
        return {}, None, 0, 0

    # Generar códigos
    codigos = generar_codigos_huffman(arbol)

    # Calcular bits sin compresión (asumiendo 8 bits por país)
    bits_originales = len([p for p, f in frecuencias.items() if f > 0]) * 8

    # Calcular bits con Huffman
    bits_comprimidos = sum(len(codigos[p]) for p in codigos.keys())

    # Tasa de compresión
    tasa = ((bits_originales - bits_comprimidos) / bits_originales * 100) if bits_originales > 0 else 0

    return codigos, arbol, bits_originales, bits_comprimidos, tasa
```

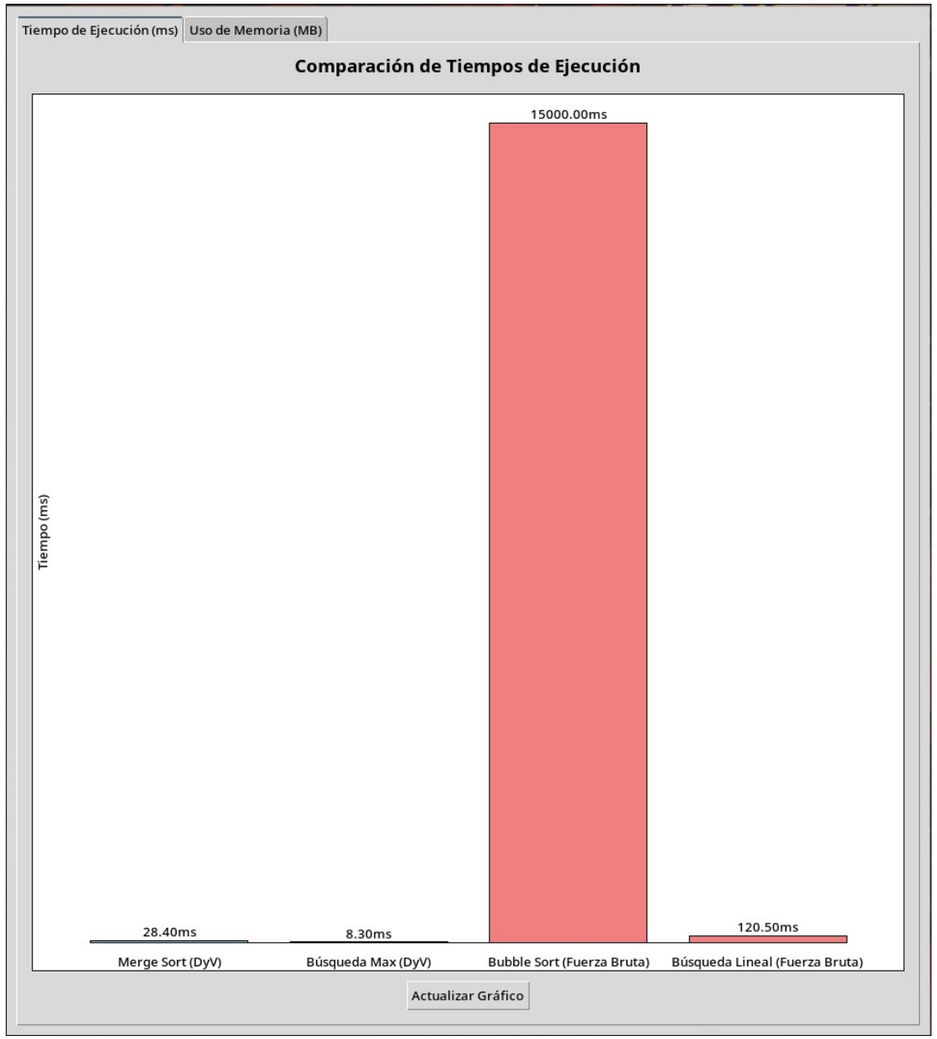
Algoritmo de Voraz

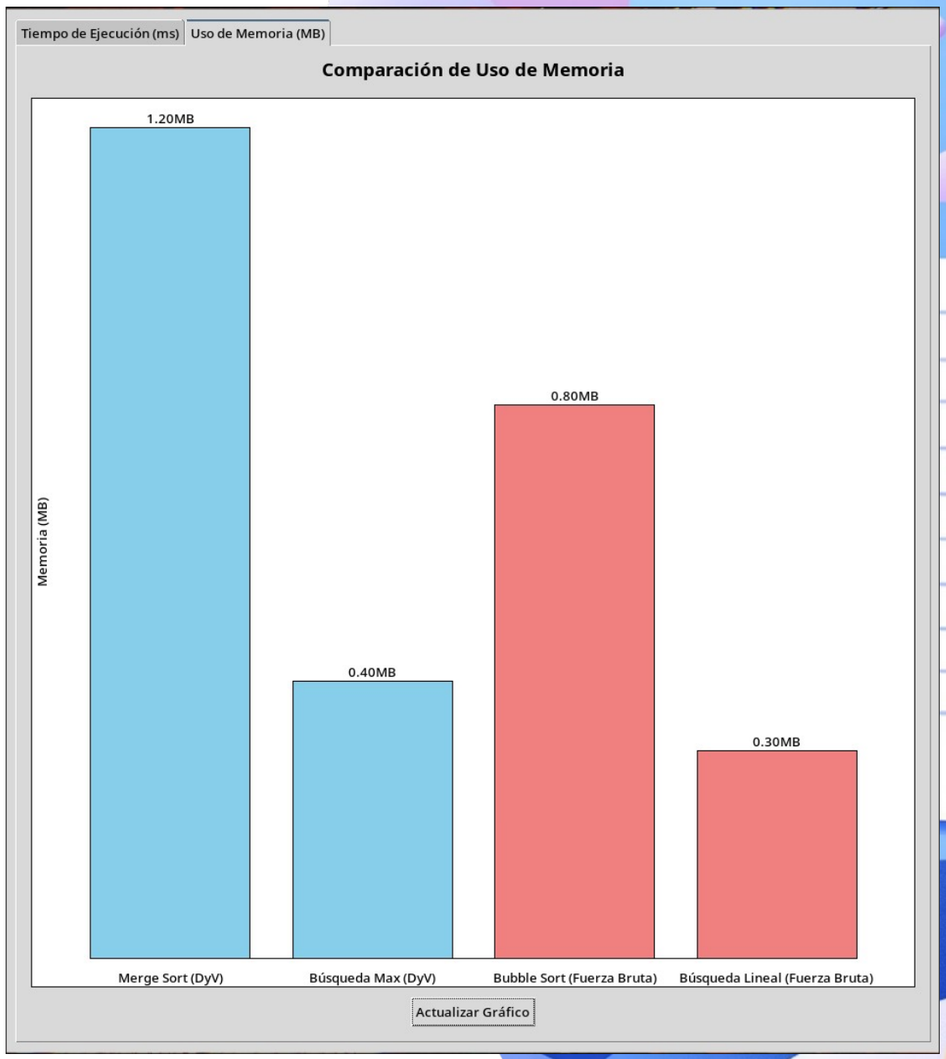
```python
def predecir_ruta_contagio_voraz(pais_origen, max_pasos=10):
    """
    Algoritmo voraz: predice la ruta de contagio eligiendo
    siempre el vecino MÁS VULNERABLE (sistema precario + mayor población)
    """
    ruta = [pais_origen]
    visitados = {pais_origen}
    actual = pais_origen

    print(f"\n{'='*70}")
    print(f"🎯 ALGORITMO VORAZ: PREDICCIÓN DE RUTA DE CONTAGIO")
    print(f"País origen: {pais_origen}")
    print(f"{'='*70}\n")

    for paso in range(max_pasos):
        if actual not in vecinos:
            print(f"❌ {actual} no tiene vecinos definidos")
            break

        # Greedy Choice: elegir vecino más vulnerable
        mejor_vecino = None
        mejor_score = -1
        candidatos = []

        for vecino in vecinos[actual]:
            if vecino not in visitados and vecino in paises:
                config = config_paises[vecino]

                # Score de vulnerabilidad (criterio voraz)
                score = config["poblacion"] / 1000000  # Población base
                if config["sistema"] == "precario":
                    score *= 3
                elif config["sistema"] == "normal":
                    score *= 1.5

                candidatos.append((vecino, score, config["sistema"]))

                if score > mejor_score:
                    mejor_score = score
                    mejor_vecino = vecino

        if mejor_vecino is None:
            print(f"✅ No hay más vecinos vulnerables desde {actual}")
            break

        # Mostrar decisión voraz
        print(f"Paso {paso + 1}: Desde {actual}")
        print(f"  Candidatos evaluados:")
        for cand, sc, sist in sorted(candidatos, key=lambda x: x[1], reverse=True):
            emoji = "🏆" if cand == mejor_vecino else "  "
            print(f"    {emoji} {cand:20s} | Score: {sc:6.2f} | Sistema: {sist}")
        print(f"  ➡️ ELECCIÓN VORAZ: {mejor_vecino} (score: {mejor_score:.2f})\n")
```

El algoritmo Voraz aqui se encarga de predecir la ruta del posible contagio,

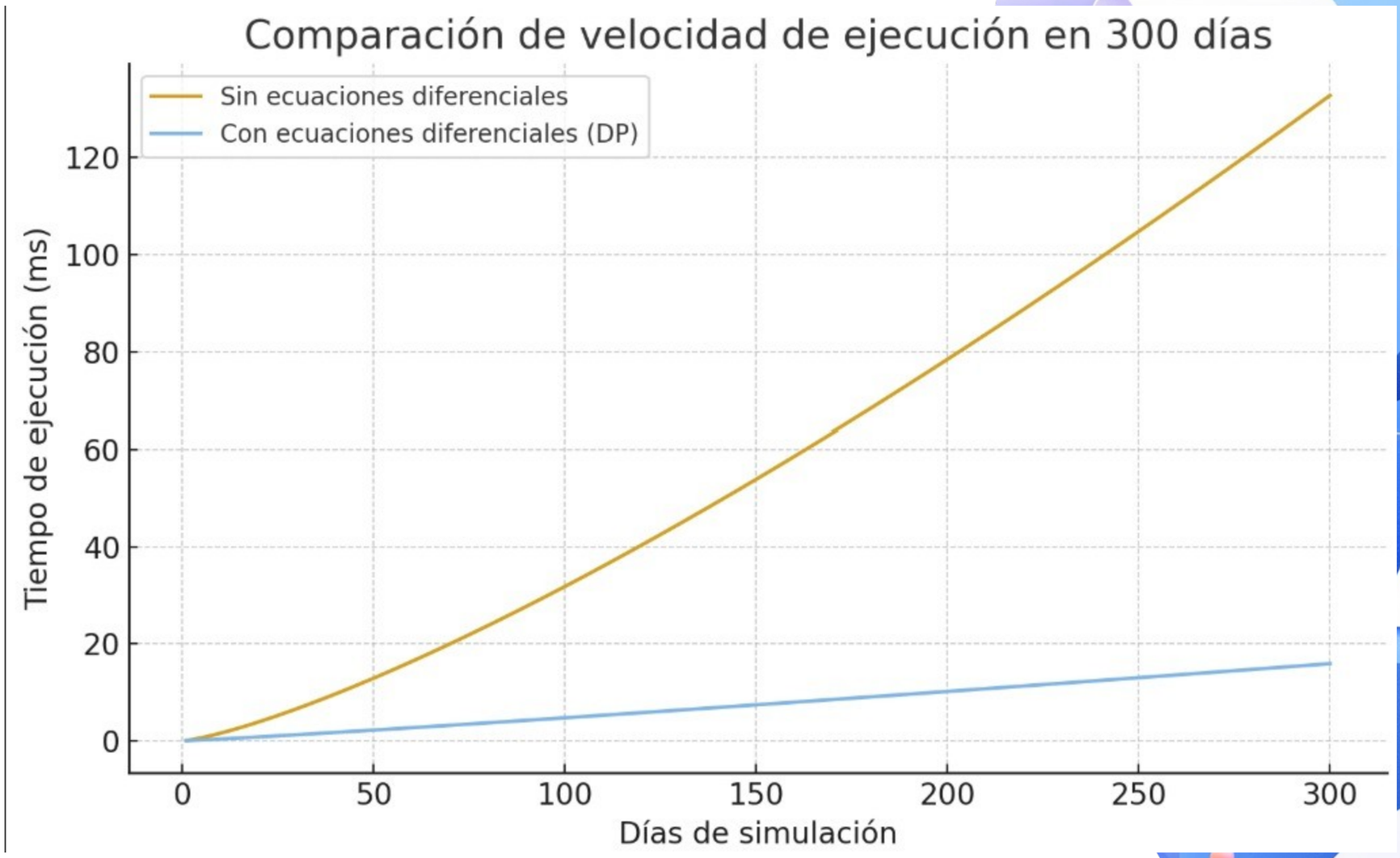
Se usa Prim para la prediccion

Comparativa





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.