Documentación Básica del Sistema de Benchmarking

Configuracion.java

Propósito: Almacena y valida todos los parámetros de configuración para la ejecución del benchmark.

Funcionalidad principal:

- Mantiene los parámetros del experimento con valores por defecto razonables
- Valida que todos los parámetros estén dentro de rangos válidos
- Proporciona acceso controlado a la configuración a través de getters y setters

Parámetros de configuración:

- **n:** Número de candidatos a generar (por defecto: 1000)
- **m:** Valor máximo para los atributos (por defecto: 100000)
- **k:** Número de repeticiones por experimento (por defecto: 5)
- **semilla:** Semilla para el generador de números aleatorios
- **distribuciones:** Tipos de distribuciones a probar (todas por defecto)
- **atributosSeleccionados:** Índices de atributos a evaluar (todos por defecto)
- **formatoExportacion:** Formato de salida CSV o JSON (CSV por defecto)
- **gigasHeap:** Tamaño del heap de la JVM en GB (0 para usar por defecto)

Validaciones implementadas:

- n debe ser mayor que 0
- m debe ser mayor que 0 y mayor o igual que n
- k debe ser mayor que 0
- Debe haber al menos una distribución seleccionada
- El tamaño de heap no puede ser negativo

TerminalUI.java

Propósito: Proporciona una interfaz de usuario por consola para interactuar con el sistema de benchmark.

Funcionalidad principal:

- Presenta un menú interactivo para navegar por las opciones del sistema
- Permite configurar parámetros de forma guiada
- Coordina la ejecución del benchmark y muestra el progreso
- Facilita la exportación de resultados en diferentes formatos
- Proporciona información del entorno de ejecución

Menú principal:

1. **Configurar parámetros:** Solicita al usuario todos los parámetros necesarios
2. **Ejecutar benchmark:** Ejecuta el benchmark completo con la configuración actual
3. **Exportar resultados:** Guarda los resultados en formato CSV o JSON
4. **Mostrar información del entorno:** Muestra detalles de la JVM y sistema operativo
5. **Salir:** Termina la aplicación

Proceso de configuración:

- Solicita número de candidatos (n)
- Solicita valor máximo (m)
- Solicita número de repeticiones (k)
- Solicita semilla (0 para usar timestamp actual)
- Solicita tamaño de heap en GB (0 para usar por defecto)
- Valida automáticamente todos los parámetros ingresados

Proceso de ejecución:

- Verifica que los parámetros sean válidos antes de comenzar
- Crea una instancia de Benchmark con la configuración especificada
- Muestra progreso durante la ejecución
- Maneja excepciones de memoria y otros errores
- Presenta los resultados en formato tabular legible

Formato de resultados mostrados:

- Tabla con columnas: Algoritmo, Distribución, Atributo, Tamaño, Comparaciones Mediana, Intercambios Mediana, Tiempo(ms)
- Los valores se muestran formateados para facilitar la lectura

Información del entorno mostrada:

- Versión de JDK utilizada
- Sistema operativo y versión
- Memoria máxima disponible para la JVM
- Memoria libre actual

Manejo de errores:

- Captura y muestra errores de memoria de forma clara
- Maneja excepciones durante la ejecución del benchmark
- Valida entrada del usuario para evitar errores de formato
- Proporciona mensajes de error descriptivos

Características de usabilidad:

- Interfaz de menú cíclico que permite múltiples operaciones
- Mensajes de confirmación para operaciones exitosas
- Indicadores de progreso durante ejecuciones largas
- Manejo robusto de entrada inválida del usuario

Validador.java

Propósito: Verifica la correctitud de los algoritmos de ordenamiento y la consistencia de las métricas recolectadas.

Funcionalidad principal:

- Valida que los arreglos estén correctamente ordenados después de aplicar un algoritmo
- Verifica que las métricas recolectadas sean consistentes con el comportamiento esperado de cada algoritmo
- Confirma que el resultado sea una permutación válida del arreglo original
- Proporciona validación completa que combina todas las verificaciones

Validaciones de ordenamiento:

- Recorre el arreglo verificando que cada elemento sea menor o igual al siguiente
- Identifica la posición exacta donde se rompe el orden si hay errores
- Maneja casos especiales como arreglos vacíos o de un solo elemento

Validaciones de métricas por algoritmo:

- **Bubble Sort:** Verifica que intercambios ≤ comparaciones (no puede intercambiar sin comparar)
- **Insertion Sort:** Verifica que intercambios ≤ comparaciones (cada intercambio requiere comparación)
- **Selection Sort:** Verifica que se realicen comparaciones (debe buscar el mínimo)
- **Merge Sort:** Verifica que intercambios = 0 (no hace intercambios directos, solo fusiones)
- **Quick Sort:** Verifica que se realicen comparaciones (necesarias para particionamiento)

Validación de permutación:

- Crea mapas de frecuencia para cada valor del atributo ordenado
- Compara las frecuencias del arreglo original vs. el ordenado
- Asegura que no se pierdan ni agreguen elementos durante el ordenamiento

ManejoMemoria.java

Propósito: Gestiona y monitorea el uso de memoria para prevenir errores por datasets grandes.

Funcionalidad principal:

- Calcula la memoria requerida antes de ejecutar experimentos
- Verifica si hay suficiente memoria disponible para completar las operaciones
- Proporciona sugerencias para optimizar el uso de memoria
- Monitorea el estado de la memoria durante la ejecución

Cálculos de memoria:

- **Por Candidato:** 4 bytes (id) + 20 bytes (atributos) + 16 bytes (overhead) = ~40 bytes base
- **Por Arreglo:** n × 8 bytes (referencias) + 24 bytes (overhead del arreglo)
- **Operaciones:** Hasta 3 copias simultáneas del dataset para algoritmos recursivos
- **Factor de seguridad:** 1.5× para cubrir memoria adicional no calculada

Verificaciones de factibilidad:

- Compara memoria requerida vs. memoria disponible en la JVM
- Proporciona mensajes claros cuando la memoria es insuficiente
- Sugiere alternativas como reducir n o aumentar el heap

Monitoreo y diagnóstico:

- Reportes detallados del uso actual de memoria
- Detección de situaciones críticas (< 10% de memoria disponible)
- Funciones de limpieza que ejecutan garbage collection
- Sugerencias automáticas del tamaño máximo factible para n

Configuración de heap:

- Información sobre el heap máximo configurado vs. solicitado
- Instrucciones para configurar más memoria con parámetros JVM
- Advertencias cuando el heap disponible es menor al requerido

Exportador.java

Propósito: Exporta los resultados del benchmark en diferentes formatos para análisis posterior.

Funcionalidad principal:

- Genera archivos CSV con estructura tabular para análisis estadístico
- Crea archivos JSON estructurados para procesamiento programático
- Produce reportes de texto completos para revisión humana
- Maneja errores de E/O de forma elegante

Formato CSV:

- Encabezados: algoritmo, característica, distribución, tamaño, métricas de mediana y RIC
- Una fila por experimento individual (algoritmo × atributo × distribución)
- Compatible con herramientas de análisis como Excel, R, Python pandas
- Valores numéricos formateados para precisión apropiada

Formato JSON:

- Estructura jerárquica que preserva la relación entre experimentos
- Incluye metadatos completos de cada experimento
- Anidación de estadísticas por tipo (comparaciones, intercambios, tiempo)
- Ideal para procesamiento automatizado y APIs

Reporte completo de texto:

- Organización por algoritmo y luego por distribución
- Formato legible para revisión manual rápida
- Resumen de métricas clave con formato numérico amigable
- Separadores visuales para facilitar la lectura

Manejo de errores:

- Captura y reporta errores de escritura de archivos
- Mensajes descriptivos cuando ocurren problemas de E/O
- Confirmación exitosa cuando los archivos se generan correctamente

Utilidad de búsqueda:

- Función auxiliar para localizar resultados específicos por criterios
- Soporte para búsqueda por algoritmo, distribución y atributo
- Manejo de casos donde no se encuentra un resultado esperado

Candidato.java

Propósito: Representa un candidato con un ID y 5 atributos numéricos.

Funcionalidad principal:

- Almacena un identificador único y un arreglo de 5 atributos enteros
- Permite obtener y establecer valores de atributos específicos por índice
- Incluye validación de límites para evitar accesos fuera del rango del arreglo
- Implementa equals y hashCode basados en el ID para comparaciones correctas

Métodos importantes:

- `getAtributo(int index)`: Obtiene el valor de un atributo específico
- `setAtributo(int index, int value)`: Establece el valor de un atributo específico
- `getAtributos()`: Devuelve una copia del arreglo completo de atributos

Metricas.java

Propósito: Registra las métricas de rendimiento durante la ejecución de algoritmos de ordenamiento.

Funcionalidad principal:

- Cuenta el número de comparaciones realizadas entre elementos
- Cuenta el número de intercambios de elementos
- Mide el tiempo de ejecución en nanosegundos usando System.nanoTime()
- Proporciona conversiones a milisegundos y microsegundos para facilitar la lectura

Métodos importantes:

- `incrementarComparaciones()`: Suma uno al contador de comparaciones
- `incrementarIntercambios()`: Suma uno al contador de intercambios
- `iniciarTiempo()`: Marca el inicio del cronometraje
- `finalizarTiempo()`: Calcula el tiempo transcurrido desde el inicio

ResultadoEstadisticas.java

Propósito: Calcula estadísticas robustas (mediana y rango intercuartílico) a partir de múltiples ejecuciones.

Funcionalidad principal:

- Recibe una lista de objetos Metricas de múltiples repeticiones del mismo experimento
- Calcula la mediana para cada métrica (comparaciones, intercambios, tiempo)
- Calcula el rango intercuartílico (RIC) como medida de dispersión robusta
- Usa interpolación lineal para cálculos precisos de cuartiles cuando es necesario

Métodos importantes:

- `calcularMediana(List<Long> valores)`: Calcula la mediana de una lista ordenada
- `calcularRangoIntercuartilico(List<Long> valores)`: Calcula Q3 - Q1
- `calcularCuartil(List<Long> sortedValues, int quartile)`: Calcula Q1 o Q3 con interpolación

ResultadoExperimento.java

Propósito: Encapsula los resultados completos de un experimento específico.

Funcionalidad principal:

- Almacena la configuración del experimento (algoritmo, atributo, distribución, tamaño)
- Contiene las estadísticas calculadas del experimento
- Proporciona múltiples formatos de salida (CSV, JSON, tabla, string)
- Incluye mapeo de índices de atributos a nombres descriptivos

Métodos importantes:

- `toCSV()`: Genera una línea CSV con todos los datos del experimento
- `toJSON()`: Genera representación JSON estructurada
- `getNombreAtributo()`: Convierte índice numérico a nombre descriptivo del atributo

Nombres de atributos:

- 0: Distancia_Marchas
- 1: Horas_Perdidas
- 2: Prebendas_Sindicales
- 3: Sobornos_Politicos
- 4: Actos_Corrupcion

GeneradorData.java

Propósito: Genera conjuntos de datos de prueba con diferentes distribuciones.

Funcionalidad principal:

- Crea arreglos de candidatos con valores controlados según la distribución solicitada
- Usa semillas para garantizar reproducibilidad de los experimentos
- Implementa tres tipos de distribuciones de datos para pruebas comprehensivas

Tipos de distribución:

- **Aleatoria:** Valores completamente aleatorios entre 1 y m
- **CasiOrdenada:** Valores mayormente ordenados con pequeñas perturbaciones (5% de intercambios aleatorios)
- **OrdenInverso:** Valores en orden descendente perfecto

Métodos importantes:

- `generarDatosAleatorios(int n, int m)`: Crea datos completamente aleatorios
- `generarDatosCasiOrdenados(int n, int m)`: Crea datos casi ordenados con variación del 10%
- `generarDatosOrdenInverso(int n, int m)`: Crea datos en orden descendente

AlgoritmosOrdenamiento.java

Propósito: Implementa cinco algoritmos de ordenamiento con medición de métricas.

Funcionalidad principal:

- Implementa algoritmos clásicos de ordenamiento
- Cuenta comparaciones e intercambios durante la ejecución
- Mide tiempo de ejecución para cada algoritmo
- Valida que el resultado esté correctamente ordenado

Algoritmos implementados:

- **Bubble Sort:** Intercambia elementos adyacentes, $O(n^2)$ en el peor caso
- **Insertion Sort:** Inserta cada elemento en su posición correcta, $O(n^2)$ en el peor caso
- **Selection Sort:** Selecciona repetidamente el mínimo, $O(n^2)$ siempre
- **Merge Sort:** Divide y conquista, $O(n \log n)$ siempre
- **Quick Sort:** Particionamiento recursivo, $O(n \log n)$ promedio, $O(n^2)$ peor caso

Detalles de implementación:

- Bubble Sort incluye optimización de parada temprana cuando no hay intercambios
- Merge Sort usa arreglos temporales para la fusión
- Quick Sort usa el último elemento como pivote
- Todos los algoritmos modifican el arreglo original

Benchmark.java

Propósito: Coordina la ejecución completa de experimentos de benchmarking.

Funcionalidad principal:

- Ejecuta experimentos sistemáticos con múltiples repeticiones
- Combina todas las configuraciones posibles (algoritmos × atributos × distribuciones)
- Valida que hay memoria suficiente antes de ejecutar
- Verifica que los resultados estén correctamente ordenados

Proceso de ejecución:

1. Verifica factibilidad de memoria para los parámetros dados
2. Para cada combinación de distribución, atributo y algoritmo:
    - Ejecuta k repeticiones del experimento
    - Genera datos frescos para cada repetición
    - Ejecuta el algoritmo y recolecta métricas
    - Valida que el resultado esté ordenado correctamente
3. Calcula estadísticas robustas de todas las repeticiones
4. Crea objeto ResultadoExperimento con los resultados finales

Configuraciones probadas:

- 3 distribuciones × 5 atributos × 5 algoritmos = 75 experimentos por configuración de tamaño
- Cada experimento se repite k veces para obtener estadísticas robustas