Manual Técnico
Analizador de Algoritmos
Sistema de Análisis de Complejidad Algorítmica

Universidad de Caldas
Análisis de Algoritmos

5 de diciembre de 2025

Índice

1. Introducción

1.1. Propósito del Sistema

Analizador de Algoritmos es un sistema inteligente basado en agentes que utiliza modelos de lenguaje (LLM) para analizar la complejidad algorítmica de código Python. El sistema emplea arquitectura de grafos con LangGraph y el modelo Gemini 2.5 Flash de Google para determinar automáticamente la complejidad temporal y espacial de algoritmos iterativos y recursivos.

1.2. Alcance

Este manual técnico documenta la arquitectura, componentes, configuración y funcionamiento interno del sistema Analizador de Algoritmos versión 0.1.0.

1.3. Audiencia

Este documento está dirigido a desarrolladores, ingenieros de software y personal técnico que necesite comprender, mantener o extender el sistema.

2. Arquitectura del Sistema

2.1. Visión General

El sistema está construido con una arquitectura de agentes basada en grafos, donde cada nodo representa una tarea específica en el flujo de análisis. La arquitectura utiliza:

- **LangGraph**: Framework para orquestar flujos de agentes
- **Google Gemini 2.5 Flash**: Modelo de lenguaje para procesamiento
- **FastAPI**: Backend para exponer servicios REST
- **Python 3.12**: Lenguaje base del sistema

2.2. Componentes Principales

2.2.1. Módulo de Agentes (`app/agents`)

Contiene toda la lógica de procesamiento mediante agentes:

- **graph.py**: Define el grafo de ejecución de agentes
- **state.py**: Maneja el estado compartido entre nodos
- **nodes/**: Nodos individuales del grafo
- **llms/**: Configuración de modelos de lenguaje
- **tools/**: Herramientas para análisis iterativo y recursivo
- **prompts/**: Templates de prompts para el LLM
- **utils/**: Utilidades para análisis sintáctico

3. Requisitos del Sistema

3.1. Requisitos de Software

Componente	Versión/Requisito
Python	3.12
google-genai	Última versión
python-dotenv	Última versión
sympy	Última versión
lark	Última versión
langchain	Última versión
langchain-google-genai	Última versión
langgraph	Última versión
langsmith	Última versión
fastapi[standard]	Última versión
matplotlib	¿= 3.10.7
ipython	¿= 9.7.0

Cuadro 1: Dependencias del proyecto

3.2. Requisitos de Hardware

- Procesador: Mínimo I5-8va generación (O similares)
- RAM: Mínimo 8 GB
- Almacenamiento: 500 MB libres
- Conexión a Internet: Requerida para API de Google Gemini

4. Configuración del Sistema

4.1. Variables de Entorno

El sistema requiere configuración mediante archivo `.env`:

```
GOOGLE_API_KEY=<api-key>
GEMINI_MODEL=gemini-2.5-flash
```

Listing 1: Configuración .env

Parámetros:

- `GOOGLE_API_KEY`: Clave de API de Google Cloud para acceso a Gemini
- `GEMINI_MODEL`: Modelo específico de Gemini a utilizar

4.2. Configuración del Proyecto

El archivo `pyproject.toml` define los metadatos y dependencias:

```
[project]
name = "agent-algorithms"
version = "0.1.0"
description = "Sistema de analisis de complejidad algoritmica"
readme = "README.md"
requires-python = ">=3.12"
```

Listing 2: pyproject.toml

5. Componentes Detallados

5.1. Nodos del Grafo de Agentes

5.1.1. initial_decision.py

Función: Determina si el algoritmo es iterativo o recursivo.
Entrada: Código Python
Salida: Tipo de algoritmo detectado

5.1.2. parse_nl_code.py

Función: Convierte lenguaje natural a código ejecutable.
Entrada: Descripción en lenguaje natural
Salida: Código Python válido

5.1.3. ast_node.py

Función: Genera el árbol sintáctico abstracto (AST).
Entrada: Código Python
Salida: Representación AST

5.1.4. code_description.py

Función: Genera descripción detallada del código.
Entrada: Código Python
Salida: Descripción textual del funcionamiento

5.1.5. validate.py

Función: Valida la sintaxis y estructura del código.
Entrada: Código Python
Salida: Estado de validación

5.1.6. Nodos de Análisis Iterativo

- `iterativo_temporal.py`: Analiza complejidad temporal O(n)
- `iterativo_espacial.py`: Analiza complejidad espacial O(n)

5.1.7. Nodos de Análisis Recursivo

- `recursivo_temporal.py`: Analiza complejidad temporal recursiva
- `recursivo_espacial.py`: Analiza complejidad espacial recursiva
- `recursivo_recurrence.py`: Resuelve relaciones de recurrencia

5.1.8. result.py

Función: Consolida y presenta resultados finales.
Salida: Informe completo de complejidad

5.2. Módulo LLMs

5.2.1. gemini.py

Configuración base del modelo Gemini sin herramientas adicionales.

5.2.2. geminiWithTools.py

Configuración del modelo Gemini con capacidad de usar herramientas (function calling).

5.3. Sistema de Prompts

El sistema utiliza prompts estructurados en Markdown organizados por:

- `NL_TO_CODE.md`: Conversión de lenguaje natural a código
- `SINTAXE.md`: Validación de sintaxis
- `GENERAR_RESULT.md`: Generación de resultados
- `iterativos/`: Prompts para análisis iterativo
  - Temporal: Mejor caso, caso promedio, peor caso
  - Espacial: Mejor caso, caso promedio, peor caso

5.4. Herramientas (Tools)

5.4.1. tools_iterativas.py

Proporciona funciones especializadas para:

- Contar operaciones en bucles
- Analizar estructuras iterativas anidadas
- Calcular complejidad temporal y espacial iterativa

5.4.2. tools_recursivas.py

Proporciona funciones para:

- Identificar casos base
- Detectar llamadas recursivas
- Formular relaciones de recurrencia
- Aplicar Teorema Maestro

5.5. Utilidades

5.5.1. generate_ast.py

Genera el árbol sintáctico abstracto usando el parser de Python.

5.5.2. generate_sum.py

Genera expresiones matemáticas para sumatorias de complejidad.

5.5.3. costo_lineas.py

Calcula el costo computacional de líneas individuales de código.

5.5.4. lark.txt

Gramática Lark para parsing personalizado de estructuras algorítmicas.

6. Flujo de Ejecución

6.1. Flujo General

1. **Entrada**: Usuario proporciona código o descripción
2. **Decisión Inicial**: Sistema determina tipo de algoritmo
3. **Parsing**: Conversión a código válido (si es necesario)
4. **Validación**: Verificación de sintaxis y estructura
5. **Generación AST**: Construcción del árbol sintáctico
6. **Análisis**: Rama iterativa o recursiva según tipo
7. **Cálculo**: Determinación de complejidades
8. **Resultado**: Generación de informe final

6.2. Diagrama de Estados

El grafo de agentes implementa un autómata finito donde cada nodo representa un estado de procesamiento y las transiciones están determinadas por el resultado del nodo anterior.

7. API y Endpoints

El sistema expone servicios REST mediante FastAPI:

```
POST /analyze
Content-Type: application/json

{
    "code": "def fibonacci(n): ...",
    "type": "recursive"
}
```

Listing 3: Ejemplo de endpoint

8. Consideraciones de Seguridad

8.1. Protección de API Keys

- Las claves API deben almacenarse en `.env`
- Nunca commitear el archivo `.env` a repositorios
- Rotar claves periódicamente

8.2. Validación de Entrada

El sistema valida todo código antes de ejecutar para prevenir:

- Inyección de código malicioso
- Bucles infinitos
- Consumo excesivo de recursos

9. Mantenimiento y Troubleshooting

9.1. Logs del Sistema

Los logs se generan automáticamente y deben revisarse ante errores.

9.2. Problemas Comunes

9.2.1. Error de API Key

Síntoma: Error de autenticación con Gemini
Solución: Verificar `GOOGLE_API_KEY` en `.env`

9.2.2. Timeout del LLM

Síntoma: Timeout en respuestas de Gemini
Solución: Aumentar timeout o simplificar entrada

9.2.3. Error de Parsing

Síntoma: Fallo al generar AST
Solución: Validar sintaxis del código de entrada

10. Extensión del Sistema

10.1. Agregar Nuevos Nodos

1. Crear archivo en `app/agents/nodes/`
2. Implementar función con firma estándar
3. Registrar en `graph.py`
4. Actualizar `state.py` si es necesario

10.2. Agregar Nuevas Herramientas

1. Definir función en `tools/`
2. Documentar con docstring
3. Registrar en configuración del LLM

11. Referencias

- LangChain Documentation: `https://python.langchain.com/`
- LangGraph Documentation: `https://langchain-ai.github.io/langgraph/`
- Google Gemini API: `https://ai.google.dev/`
- FastAPI Documentation: `https://fastapi.tiangolo.com/`

12. Apéndices

12.1. Apéndice A: Estructura de Directorios Completa

```
Agent_Algorithms/
|-- app/
|   |-- agents/
|       |-- __init__.py
|       |-- graph.py
|       |-- state.py
|       |-- llms/
|       |   |-- gemini.py
|       |   |-- geminiWithTools.py
|       |-- nodes/
|       |   |-- __init__.py
|       |   |-- ast_node.py
|       |   |-- code_description.py
```

```
|       |   |-- initial_decision.py
|       |   |-- iterativo_espacial.py
|       |   |-- iterativo_temporal.py
|       |   |-- parse_nl_code.py
|       |   |-- recursivo_espacial.py
|       |   |-- recursivo_recurrence.py
|       |   |-- recursivo_temporal.py
|       |   |-- result.py
|       |   |-- validate.py
|       |-- prompts/
|       |   |-- __init__.py
|       |   |-- GENERAR_RESULT.md
|       |   |-- NL_TO_CODE.md
|       |   |-- SINTAXE.md
|       |   |-- iterativos/
|       |       |-- espacial/
|       |       |   |-- CASO_PROMEDIO.md
|       |       |   |-- MEJOR_CASO.md
|       |       |   |-- PEOR_CASO.md
|       |       |-- temporal/
|       |           |-- CASO_PROMEDIO.md
|       |           |-- MEJOR_CASO.md
|       |           |-- PEOR_CASO.md
|       |-- tools/
|       |   |-- tools_iterativas.py
|       |   |-- tools_recursivas.py
|       |-- utils/
|           |-- costo_lineas.py
|           |-- generate_ast.py
|           |-- generate_sum.py
|           |-- lark.txt
|-- .env
|-- .python-version
|-- pyproject.toml
```

Listing 4: Estructura del proyecto

12.2. Apéndice B: Glosario

AST Abstract Syntax Tree - Árbol sintáctico abstracto

LLM Large Language Model - Modelo de lenguaje grande

API Application Programming Interface

REST Representational State Transfer