GPTur

Claudia Hernández Pérez, Joel Aparicio Tamayo, and Kendry Javier Del Pino Barbosa

Facultad de Matemática y Computación, La Habana, Cuba

Resumen. GPTur es un sistema diseñado con una arquitectura de Generación Aumentada por Recuperación (RAG) con agentes especializados para cada funcionalidad y una base de datos vectorial para la recuperación de la información. Posee, además, varios agentes encargados de tópicos específicos como gastronomía, vida nocturna, alojamiento y lugares históricos, que comparten datos en una pizarra central. Estos agentes están dirigidos por un guía y un planificador. El primero se encarga de responder consultas mientras el segundo utiliza algoritmos para planear itinerarios ajustados a presupuestos. Este agente resuelve un problema de satisfacción de restricciones utilizando la técnica de precocido simulado. También, el programa preprocesa la consulta para obtener ciertas entidades, palabras claves y sentimientos que puedan influir en los resultados esperados. Todo eso, permite conformar una respuesta acertada, acorde a las necesidades del usuario, validadas experimentalmente.

Palabras clave: RAG, agente, problema de satisfacción de restricciones, precocido simulado

1. Introducción

La industria turística enfrenta desafíos únicos en contextos con limitaciones en la infraestructura digital y actualización constante de servicios. Cuba, como destino turístico emblemático, requiere soluciones innovadoras que superen estas barreras mediante inteligencia artificial. *GPTur*, un sistema de agentes inteligentes que funciona como guía turístico virtual, está diseñado específicamente para ser una alternativa viable en ese escenario.

La aplicación consiste en un chatbot capaz de responder consultas sobre destinos turísticos cubanos y sus peculiaridades, así como de generar itinerarios para *tours* ajustándose a un presupuesto. El sistema está programado en *Python*, aprovechando la comodidad del lenguaje y la diversidad de bibliotecas útiles que ofrece. La interfaz está diseñada con *streamlit* (framework de *Python* para *frontend*), pues ofrece un estilo sencillo, moderno y fácil de programar para entornos web. El código fuente de la aplicación se puede encontrar en la plataforma GitHub: https://github.com/ClaudiaHdezPerez/GPTur.

2. Detalles de Implementación

En esta sección, se abordarán los distintos temas asociados a la implementación de la solución propuesta, que incluyen la arquitecura RAG y multiagente, funcionamiento del

crawler automatizado y recopilación del corpus, base de datos vectorial y representación semántica del conocimiento mediante *embeddings*, procesamiento de lenguaje natural y algoritmos basados en metaheurísticas.

2.1. Arquitectura RAG (*Generación Aumentada por Recuperación*)

Un sistema de Generación Aumentada por Recuperación combina generación de texto con recuperación de información para mejorar la calidad, precisión y actualidad de las respuestas obtenidas de los modelos de lenguaje. Está compuesto por una entrada, un recuperador, un generador y una salida. La implementación propuesta es bastante sencilla, ajustada a la definición:

1. Entrada: constituida por la consulta del usuario en lenguaje natural.
2. Recuperador: agente que recibe la consulta y le pide a la base de datos vectorial que recupere la información relacionada.
3. Generador: agente que utiliza la consulta y la información recuperada para decidir si la respuesta debe ser generada por el guía o el planificador. Para tomar esa decisión le pide a un modelo de lenguaje que detecte la intención del usuario.
4. Salida: respuesta generada por el agente inteligente encargado de elaborarla.

2.2. Arquitectura Multiagente [1]

Un sistema multiagente está constituido por un conjunto de agentes, cada uno con determinadas capacidades y recursos para la solución de problemas inherentemente distribuidos. Los agentes interactúan entre ellos, determinan y coordinan tareas a realizar según sus capacidades y recursos.
GPTur implementa una arquitectura multiagente con especialización por dominios turísticos:

- *GuideAgent*: Agente coordinador principal
- *GastronomyAgent*: Especializado en restaurantes y gastronomía
- *HistoricAgent*: Expertos en sitios históricos y culturales
- *LodgingAgent*: Encargado de alojamientos y hospedajes
- *NightlifeAgent*: Especialista en vida nocturna y entretenimiento
- *PlannerAgent*: Generador de itinerarios personalizados

Modelo *Creencias-Deseos-Intenciones*: Cada agente extiende la clase base *BDIAgent* que implementa el ciclo BDI (Beliefs-Desires-Intentions):

```
class BDIAgent:
    def __init__(self, name, vector_db=None):
        self.beliefs = {}      # Estado interno y conocimiento
        self.desires = []      # Objetivos del agente
        self.intentions = []   # Planes seleccionados
        self.plans = {}        # Planes disponibles
```

El ciclo de ejecución BDI sigue el algoritmo clásico:

1. *action(percept)*: Punto de entrada principal
2. *brf(percept)*: Actualización de creencias
3. *generate_options()*: Generación de opciones
4. *filter(options)*: Selección de intenciones
5. *execute()*: Ejecución de acciones

Cada agente especializado implementa planes y acciones específicos, funciones de relevancia y verificación de precondiciones.

Comunicación entre Agentes: El sistema implementa un patrón de pizarra compartida para colaboración:

```
class Blackboard:
    _instance = None # existe una sola pizarra en el programa
    _lock = Lock() # variable de bloqueo

    def __new__(cls):
        if cls._instance is None:
            # crea la instancia
        return cls._instance

    def __init__(self):
        self._shared_space = {}
        self._current_problem = None

    def write(self, agent_name, contribution):
        # Escribe en el espacio compartido

    def read(self, problem_id):
        # Lee contribuciones de otros agentes
```

El GuideAgent utiliza un sistema de *Thread Pool* para ejecutar a los agentes especializados en paralelo. Cada agente escribe sus contribuciones en la pizarra si no interfiere con otro agente. En caso que sí, se queda esperando hasta poder escribir. El GuideAgent lee todas las contribuciones de la pizarra y las fusiona para generar una respuesta. Para eso se utiliza un modelo de lenguaje. Entre los beneficios de esta arquitectura se pueden mencionar:

- *Modularidad*: Cada agente encapsula conocimiento especializado
- *Escalabilidad*: Fácil añadir nuevos dominios (playas, deportes, etc.)
- *Robustez*: Fallos en un agente no colapsan el sistema

Esta arquitectura permite construir un sistema de recomendaciones turísticas complejo, adaptable y altamente especializado.

2.3. Crawler Automatizado en Dos Fases. Recopilación del *Corpus*

Se conoce como *crawler* a un software que realiza un proceso automatizado de navegar por la web de manera sistemática para indexar y recopilar información de diferentes sitios web. El sistema implementa un *crawler* en dos fases.

Fase Inicial: En esta fase el *crawler* se configura con los parámetros necesarios, como las urls objetivo (disponibles en *src /data /sources.json*), criterios de búsqueda y profundidad de rastreo. Utiliza técnicas de *scraping* (extracción de la web) para navegar por sitios web turísticos, identificando y extrayendo información relevante como descripciones de lugares, eventos, alojamientos y actividades. Los datos extraídos son normalizados y almacenados en un formato *.json*, facilitando su posterior análisis y procesamiento.

Fase Dinámica: Esta fase actúa cuando el programa detecta que la información con que cuenta no es suficiente para responder concretamente la consulta del usuario. Esta detección es realizada por un modelo de lenguaje al cual se le envía un *prompt* con la consulta y la respuesta generada por el sistema, esperando una indicación de *OK* o *Actualizar*. En caso de la segunda, en un nuevo *prompt* se le pide *urls* que permitan enriquecer el corpus y responder la consulta. El *crawler* accede a ellas y se toma dicha información para fusionarla con la respuesta inicial. A su vez es añadida a la base de datos vectorial.

Recopilación y Construcción del Corpus: Este proceso consta de cuatro tareas principales:

- *Selección de Fuentes:* Se identificaron y seleccionaron fuentes web relevantes para el turismo, como portales oficiales, blogs y sitios de reseñas.
- *Rastreo y Descarga:* El *crawler* recorre las páginas seleccionadas, descargando el contenido textual y metadatos asociados.
- *Filtrado y Normalización:* Se aplican filtros para eliminar información irrelevante o duplicada y se normalizan los datos para mantener la coherencia en el corpus.
- *Almacenamiento:* El corpus final se almacena en formato *.json* (disponible en *src /data /processed /normalized_data.json*).

Este proceso automatizado permite mantener un corpus representativo del dominio turístico, esencial para el tema que aborda el sistema.

2.4. Base de Datos Vectorial y Modelo de Representación del Conocimiento

El sistema utiliza *ChromaDB* como base de datos vectorial. *ChromaDB* es una solución especializada para el almacenamiento y consulta de vectores de alta dimensión, optimizada para tareas de inteligencia artificial y procesamiento de lenguaje natural.

- *Integración con el Sistema:* La integración se realiza a través del módulo *src /vector_db /chroma_storage.py*, que proporciona funciones para insertar, actualizar y consultar vectores en la base de datos.
- *Almacenamiento Eficiente: ChromaDB* permite almacenar grandes volúmenes de *embeddings* generados a partir del corpus, manteniendo tiempos de respuesta bajos incluso en consultas complejas.
- *Búsqueda por Similitud:* Utilizando algoritmos de búsqueda aproximada de vecinos más cercanos (ANN), *ChromaDB* facilita la recuperación de fragmentos de información relevantes a partir de consultas vectorizadas, mejorando la precisión y relevancia de las respuestas del sistema.
- *Escalabilidad y Actualización: ChromaDB* soporta la actualización dinámica de la base de datos, permitiendo añadir o eliminar vectores conforme el corpus evoluciona, sin afectar el rendimiento general.
- *Persistencia y Seguridad:* Los datos almacenados en *ChromaDB* pueden persistirse en disco, asegurando la integridad y disponibilidad de la información incluso ante reinicios del sistema.

El uso de *ChromaDB* proporciona una infraestructura robusta y escalable para la gestión del conocimiento en forma vectorial, facilitando la integración de capacidades avanzadas de recuperación semántica y razonamiento en el sistema.

Modelo de Representación del Conocimiento: El sistema emplea un modelo de representación del conocimiento basado en *embeddings* semánticos, lo que permite capturar relaciones y similitudes entre conceptos turísticos. Las principales características son:

- *Representación Distribuida:* Cada entidad o fragmento de información se representa como un vector en un espacio de alta dimensión, facilitando la comparación y agrupación de conceptos similares.
- *Recuperación Semántica:* Las consultas del usuario se transforman en vectores y se comparan con los almacenados en la base de datos, recuperando la información más relevante según la similitud semántica. Todo eso gracias al método *similarity_search* de *Chroma*.

Este enfoque permite una gestión eficiente y flexible del conocimiento, adaptándose a la naturaleza dinámica y heterogénea de la información turística.

2.5. Procesamiento de Lenguaje Natural (NLP)

El sistema GPTur incorpora un módulo de procesamiento de lenguaje natural (NLP) para analizar y comprender las consultas de los usuarios, así como para procesar y organizar la información turística recopilada. Las principales tareas de NLP implementadas son:

- *Preprocesamiento de texto:* Se realiza limpieza, normalización y lematización de las consultas y documentos, eliminando palabras vacías y signos de puntuación, utilizando el modelo *es_core_news_md* de *spacy*.

- *Extracción de entidades:* Se identifican entidades nombradas relevantes (como ciudades, atracciones y organizaciones) en las consultas y documentos, facilitando la recuperación precisa de información.
- *Análisis de sentimiento:* Se implementa un análisis de sentimiento simple para detectar la polaridad de las consultas, lo que puede influir en la generación de respuestas más adecuadas.
- *Extracción de palabras clave:* Se extraen los términos más relevantes de cada consulta o documento, mejorando la indexación y recuperación.
- *Fragmentación de texto:* Los textos largos se dividen en fragmentos coherentes para optimizar el procesamiento y la generación de embeddings.

Estas tareas permiten que el sistema interprete correctamente las necesidades del usuario y recupere la información más relevante, integrando el procesamiento de lenguaje natural en la lógica central de los agentes inteligentes.

2.6. Metaheurística y Problema de Satisfacción de Restricciones [2]

En el desarrollo de sistemas de planificación de viajes inteligentes, surge el problema de optimizar itinerarios turísticos considerando múltiples restricciones y objetivos.
El sistema debe generar un itinerario diario considerando:

- *Actividades:* Desayuno, almuerzo, cena, actividad nocturna y alojamiento.
- *Restricciones:* Presupuesto diario máximo.
- *Objetivo:* Maximizar la proporción calidad-precio de los lugares seleccionados.

Matemáticamente, el problema se formula como:

$$\text{máx} \sum_{d=1}^{D} \frac{\sum_{a \in A} r_{d,a}}{\sum_{a \in A} \overline{X_a}}$$

$$\text{sujeto a: } \sum_{a \in A} c_{d,a} \leq B_d \quad \forall d \in \{1, \ldots, D\}$$

Donde:

- D: Número de días del viaje
- $A = \{\text{desayuno}, \text{almuerzo}, \text{cena}, \text{noche}, \text{alojamiento}\}$
- $r_{d,a}$: Valoración del lugar asignado al día d para actividad a
- $c_{d,a}$: Costo del lugar asignado al día d para actividad a
- $\overline{X_a}$: Precio medio de la actividad a
- B_d: Presupuesto máximo para el día d

El problema está entonces sujeto a un proceso estocástico, pues para calcular $\overline{X_a}$ se realizan 30 simulaciones sobre la solución actual, calculando la media de precios para cada actividad. Los precios están representados por una variable aleatoria que distribuye normal con parámetros $\mu = \text{precio\_base}$ y $\sigma = 20\,\% * \text{precio\_base}$.

Algoritmo de Recocido Simulado: El recocido simulado es una metaheurística inspirada en el proceso térmico de recocido en metalurgia. El problema que se presenta, debe ser resuelto utilizando metaheurística pues es representado por un proceso estocástico. Específicamente se eligió este algoritmo porque era sencillo de implementar A continuación se describe la implementación del algoritmo.

Representación de la Solución: Cada solución se representa como una matriz donde las filas corresponden a días y las columnas a tipos de actividad:

Día	Desayuno	Almuerzo	Cena	Noche	Alojamiento
1	L_{11}	L_{12}	L_{13}	L_{14}	L_{15}
2	L_{21}	L_{22}	L_{23}	L_{24}	L_{25}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Donde L_{ij} es un lugar seleccionado del conjunto correspondiente a la categoría.

Función de Evaluación: La calidad de una solución se calcula como la suma de las valoraciones de todos los lugares seleccionados:

$$f(S) = \sum_{d=1}^{D} \frac{\sum_{a \in A} \text{rating}(L_{d,a})}{\sum_{a \in A} \overline{X_a}}$$

Generación de Vecinos: El operador de vecindad modifica aleatoriamente una actividad en un día. Ver Alg. 1.

Esquema de Enfriamiento: Se utiliza un enfriamiento geométrico:

$$T_{k+1} = \alpha \cdot T_k \quad \text{con} \quad \alpha = 0{,}99$$

Parámetros iniciales:

- Temperatura inicial (T_0): 100.0
- Temperatura mínima ($T_{\text{mín}}$): 0.1
- Iteraciones por temperatura: 1000

Manejo de Restricciones: Las soluciones se validan verificando que el costo diario total no exceda el presupuesto. Ver Alg. 2.

Implementación: La implementación completa se puede apreciar en el Alg. 3.

3. Validación y Experimentación

Esta sección fue un elemento esencial en la etapa de desarrollo y, en muchos casos, influyó en la toma de decisiones. Se realizaron varios experimentos, que incluyen valoraciones de la respuesta del sistema y evaluación de los algoritmos de metaheurísticas.

3.1. Respuesta del Sistema

Durante las distintas etapas de desarrollo, fue importante evaluar el comportamiento de la aplicación. Para estos experimentos se utilizó un modelo de lenguaje para generar 30 preguntas relacionadas con el corpus y obtener las respuestas de GPTur automáticamente. Luego, se creó una cosulta para un modelo de lenguaje pidiendo una evaluación entre 0 y 5 puntos de la respuesta del sistema en cuanto a:

- Precisión de la información
- Coherencia y relevancia
- Cobertura del tema
- Calidad de las recomendaciones
- Ajuste a la pregunta realizada

Después de obtener los resultados de la muestra original, se llegó a la conclusión de que se debían analizar muchas más muestras, es decir, ejecutar la experimentación varias veces, pero la capacidad de cómputo y tiempo necesarios para generar 30 preguntas y evaluar las respuestas un número *n* de veces, con *n* relativamente grande, hacían inviable esa solución. Por tanto, se recurrió a una alternativa no menos eficaz: simular la evaluación de nuevas muestras. Es entonces cuando se decidió aplicar la técnica conocida como *bootstrapping*.

Procedimiento de Bootstrapping y Número de Iteraciones: En el procedimiento de bootstrapping se generan, con reemplazamiento, B muestras de tamaño n a partir de la muestra original, calculándose de cada una el estimador $\bar{X}^*$. El error estándar asociado a la distribución de estos estimadores se expresa como:

$$\mathrm{SE}_B = \frac{\sigma}{\sqrt{B}}. \tag{1}$$

Para los experimentos realizados utilizando esta técnica, inicialmente se había planteado un umbral del 5 % de σ, para que SE_B fuese insignificante comparado con la variabilidad natural de los datos, lo que implicaba:

$$\mathrm{SE}_B \leq 0{,}05\,\sigma. \tag{2}$$

Despejando en la inecuación se obtiene

$$\frac{\sigma}{\sqrt{B}} \leq 0{,}05\,\sigma \quad \Longrightarrow \quad \sqrt{B} \geq 20 \quad \Longrightarrow \quad B \geq 400.$$

Entonces, para mantener el error estándar por debajo del umbral propuesto, en cada experimento se debió realizar al menos 400 simulaciones. Sin embargo, se tomó la decisión de hacer 1000 en total, para hacer más robusto el experimento y disminuir aún más el error estándar:

$$\frac{1}{\sqrt{1000}} \approx 0{,}0316,$$

lo que implica que $\mathrm{SE}_B \approx 0{,}0316\,\sigma$, cumpliéndose de forma estricta la condición.

Cuadro 1. Resultados del análisis de bootstrapping (IC 95 %)

Métrica	Original	Bootstrapped
Puntuación Media	3.7	3.6982...
Desviación Estándar	0.4582...	0.0839...

Sistema con Agentes Guía y Planificador Solamente: La Tabla Tab. 1 resume las métricas clave obtenidas:

Remark 1. Se obtuvo un intervalo de confianza de $[3{,}5333..., 3{,}8666...]$, lo que indica unos resultados aceptables. La Figura Fig. 1 muestra la distribución bootstrap de la puntuación media y la figura Fig. 2 muestra un mapa de calor con las puntuaciones otorgadas a las respuestas del sistema a las 30 preguntas iniciales.

Sistema con Agentes Especializados: La Tabla Tab. 2 resume las métricas clave obtenidas:

Cuadro 2. Resultados del análisis de bootstrapping (IC 95 %)

Métrica	Original	Bootstrapped
Puntuación Media	4.1333...	4.1358...
Desviación Estándar	0.6182...	0.1166...

Remark 2. Se obtuvo un intervalo de confianza de $[3{,}90, 4{,}3666...]$, lo que indica unos resultados bastante buenos. La Figura Fig. 3 muestra la distribución bootstrap de la puntuación media y la figura Fig. 4 muestra un mapa de calor con las puntuaciones otorgadas a las respuestas del sistema a las 30 preguntas iniciales.

En resumen, se puede observar que al añadir agentes especializados al sistema, mejoró la respuesta del modelo, lo cual demostró que la decisión tomada estuvo acertada.

3.2. Evaluación de la Metaheurística

Este estudio evalúa el algoritmo de recocido simulado implementado, con enfoque en su convergencia y sensibilidad a los parámetros T y α.
Se evaluaron combinaciones sistemáticas de:

- *Temperatura inicial (T):* 100, 200
- *Factor de enfriamiento (α):* 0.95, 0.99

Para el experimento se tuvo en cuenta las siguiente configuraciones:

- *Ejecuciones:* 30 por combinación (total 120 ejecuciones)
- *Duración viaje:* 3 días
- *Presupuesto diario:* 150 dólares

- *Lugares disponibles:*
  - Restaurantes: 15
  - Locales nocturnos: 10
  - Alojamientos: 8
- *Tiempo máximo:* 180 segundos por ejecución

Para evaluar cada combinación se tuvo en cuenta:

1. Resultado (evaluación de la solución dada)
2. Iteración de convergencia (iteración a partir de la cual la evaluación de la solución se mantuvo sobre el 99 % de la solución final)

Resultados: Luego de realizar el experimento, se llegó a los siguientes resultados por combinación (Tab. 3):

Cuadro 3. Resultados promedio de convergencia por combinación de parámetros

T	α	Puntuación	Mejora	Convergencia
100	0.95	1.15	0.23	5120
100	0.99	**1.16**	**0.24**	**4239**
200	0.95	1.14	0.22	5870
200	0.99	1.15	0.23	5015

Remark 3. Las curvas de convergencia promedio se muestran en la Fig. 5

Conclusión: Para la planificación de itinerarios mediante recocido simulado, se recomienda:

- Temperatura inicial: $T = 100$
- Factor de enfriamiento: $\alpha = 0{,}99$

Justificación:

- Maximiza la calidad de la solución (puntuación promedio: 1.16)
- Converge más rápido (promedio de iteraciones: 4239)

4. Conclusiones

GPTur constituye un sistema robusto, ideal como proyecto de carácter educativo para estudiantes de Ciencias de la Computación, pues fomentó la investigación, recolección y almacenamiento de datos, ingeniería de software, trabajo en equipo y la experimentación, que son las habilidades que la carrera busca impregnar en cada estudiante. No obstante, su

interfaz amigable, sus respuestas actualizadas y personalizadas, así como su arquitectura escalable, abren las puertas a futuras aplicaciones que mejoren la experiencia de los turistas en la Isla.
La eficacia del sistema no solo se basa en resultados subjetivos, sino que está respaldada por un conjunto de pruebas y experimentos que han demostrado cómo ha ido evolucionando GPTur, incrementando la precisión de sus respuestas en cada fase de desarrollo, lo que justifica las decisiones tomadas para la implementación propuesta.
La aplicación es una muestra del esfuerzo de todo un semestre y de los conocimientos adquiridos en los temas de *Simulación*, *Inteligencia Artificial* y *Sistemas de Recuperación de Información*, pues en tiempo récord se logró implementar un RAG, una arquitectura multiagente, metaheurística, entre otros muchos aspectos, que debieron ser quirúrjicamente enlazados para el funcionamiento completo del programa.
Por supuesto, como todo sistema, tiene limitaciones que no pudieron resolverse en el tiempo especificado:

- No considerar las distancias entre los sitios a la hora de planear viajes utilizando la metaheurística.
- No considerar el transporte como elemento fundamental en los *tours*.
- No utilizar ontologías para realizar inferencias sobre los sentimientos e intenciones del usuario.

No obstante, se proponen algunas soluciones para lograrlas:

- Integrar el sistema a una API con datos sobre la ubicación de sitios turísticos y que permita calcular distancias.
- Añadir nuevos agentes especializados para manejar otros problemas, como por ejemplo el del transporte.
- Integrar una ontología o un sistema basado en reglas para captar mejor las intenciones del usuario.

Por todo lo expuesto y más, GPTur ha llegado a ser una experiencia magnífica para cada uno de sus desarrolladores.

Referencias

[1] Luciano García Garrido, Luis Martí Orosa y Luis Pérez Sánchez. *Temas de Simulación*. La Habana, Cuba: Facultad de Matemática y Computación, Universidad de la Habana.

[2] El-Ghazali Talbi. *Metaheuristics: From Design to Implementation*. Wiley Series on Parallel and Distributed Computing. Hoboken, NJ: Wiley, 2009. ISBN: 978-0-470-27858-1.

Apéndices

A continuación se muestran los apéndices del informe.

A. Estudios Estadísticos

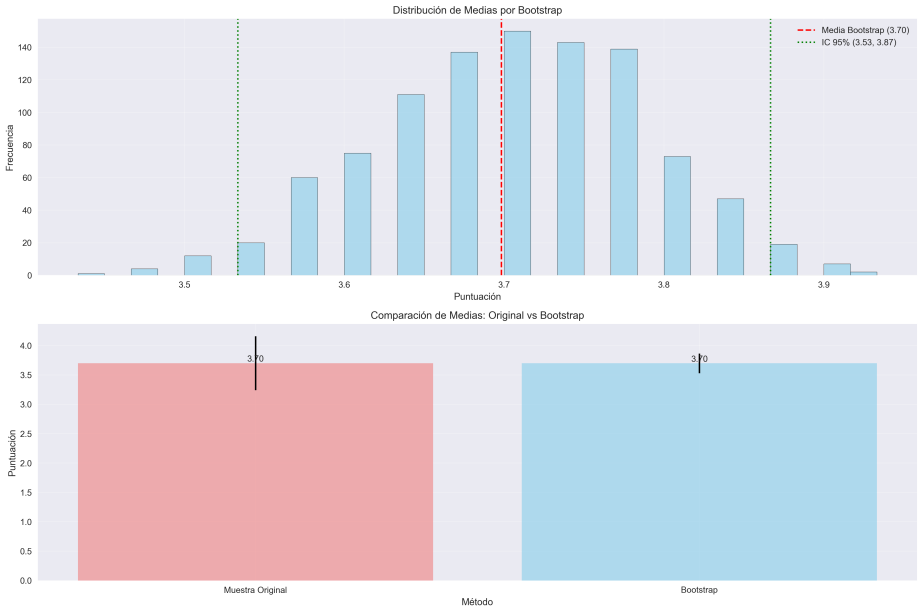

Figura 1. Distribución bootstrap de la puntuación media ($N = 1000$). La línea discontinua roja indica la media bootstrap (3.70)

Figura 2. Evaluaciones obtenidas para la respuesta a cada pregunta en la muestra inicial (tamaño 30)

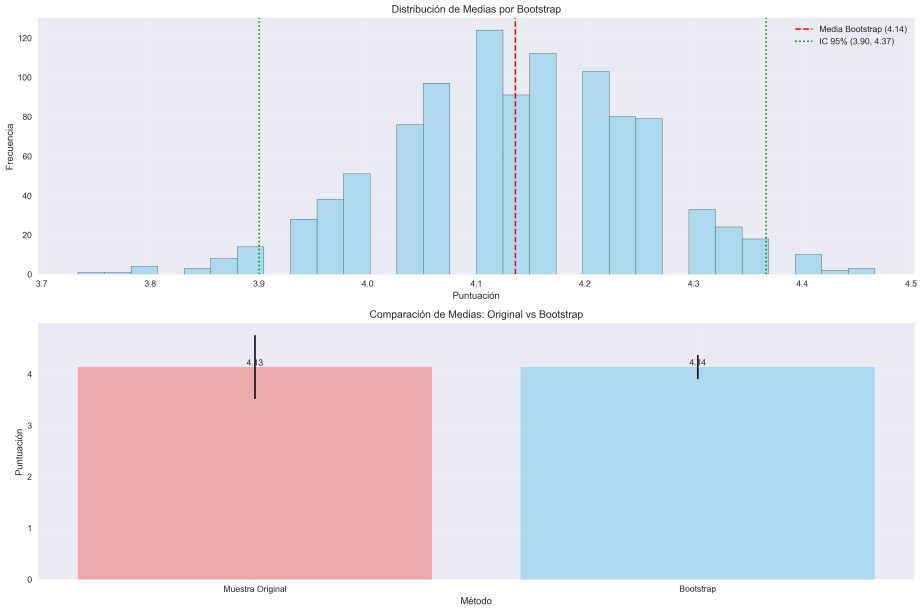

Figura 3. Distribución bootstrap de la puntuación media ($N = 1000$). La línea discontinua roja indica la media bootstrap (4.14)

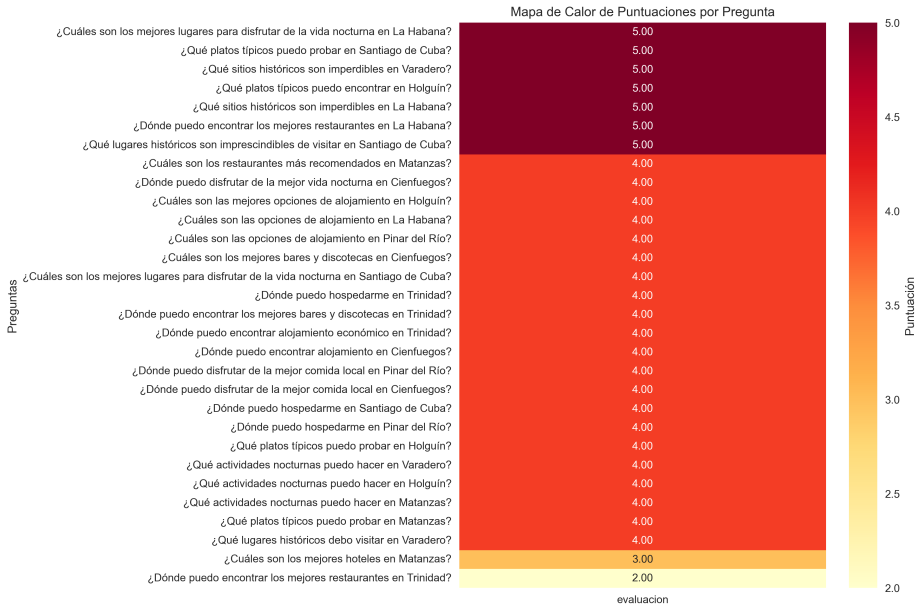

Figura 4. Evaluaciones obtenidas para la respuesta a cada pregunta en la muestra inicial (tamaño 30)

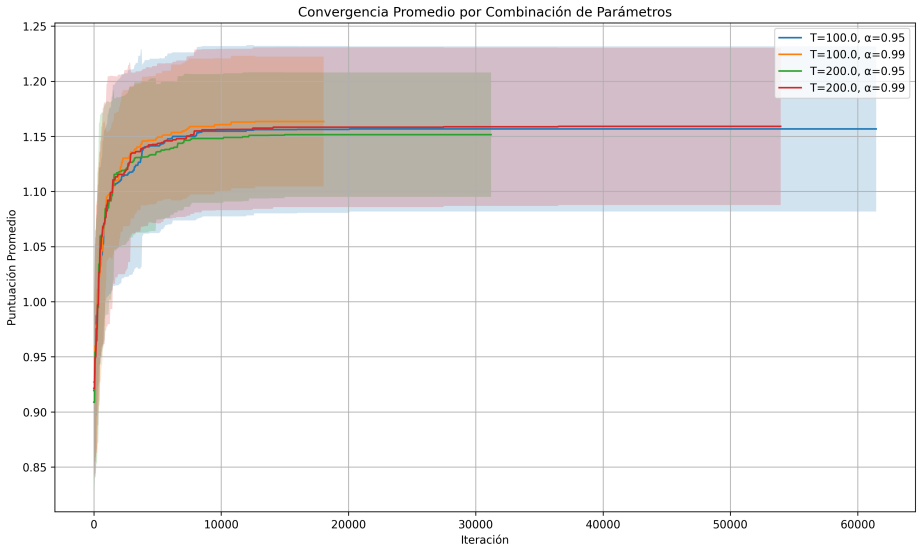

Figura 5. Convergencia promedio por combinación de parámetros

B. Algoritmos

Algorithm 1 Generación de Vecino

```
procedure GENERATENEIGHBOR(sol)
    day ← random(0, len(sol) − 1)
    activity ← random.choice(actividades)
    if activity ∈ {desayuno, almuerzo, cena} then
        new_place ← random.choice(lugares_gastronomicos)
    else if activity = noche then
        new_place ← random.choice(lugares_nocturnos)
    else
        new_place ← random.choice(alojamientos)
    end if
    sol[day][activity] ← new_place
    return sol
end procedure
```

Algorithm 2 Validación de Solución

```
procedure ISVALIDSOLUTION(sol)
    for day in sol do
        total_cost ← 0
        for activity in day do
            total_cost ← total_cost + day[activity].cost
        end for
        if total_cost > budget_per_day then
            return False
        end if
    end for
    return True
end procedure
```

Algorithm 3 Algoritmo de Recocido Simulado para Planificación de Itinerarios

Require: $days \geq 1$, $places \neq \emptyset$, $budget\_per\_day > 0$, $max\_iter > 0$, $max\_time > 0$
Ensure: Solución óptima de itinerario válido

function SIMULATEDANNEALINGCSP($days$, $places$, $budget\_per\_day$, $destination$, max_iter, max_time)
    $start\_time \leftarrow$ time.now()
    $current\_sol \leftarrow$ generate_initial_solution() ▷ Solución aleatoria inicial
    $best\_sol \leftarrow$ copy($current\_sol$)
    $best\_rating \leftarrow$ calculate_total_rating($current\_sol$)
    $T \leftarrow 100{,}0$ ▷ Temperatura inicial
    $\alpha \leftarrow 0{,}99$ ▷ Factor de enfriamiento
    $T_{min} \leftarrow 0{,}1$ ▷ Temperatura final
    while $T > T_{min}$ **and** time.now() $- start\_time < max\_time$ **do**
        for $i = 1$ **to** max_iter **do**
            if time.now() $- start\_time \geq max\_time$ **then**
                return $best\_sol$ ▷ Tiempo límite alcanzado
            end if
            $neighbor\_sol \leftarrow$ generate_neighbor($current\_sol$)
            if not is_valid_solution($neighbor\_sol$) **then**
                continue ▷ Descarta soluciones inválidas
            end if
            $\Delta \leftarrow$ calculate_total_rating($neighbor\_sol$) $-$ calculate_total_rating($current\_sol$)
            if $\Delta > 0$ **or** random() $< \exp(\Delta/T)$ **then**
                $current\_sol \leftarrow neighbor\_sol$
                if calculate_total_rating($current\_sol$) $> best\_rating$ **then**
                    $best\_sol \leftarrow$ copy($current\_sol$)
                    $best\_rating \leftarrow$ calculate_total_rating($current\_sol$)
                end if
            end if
        end for
        $T \leftarrow \alpha \times T$ ▷ Enfriamiento
    end while
    return $best\_sol$
end function