

Universidad de los Andes

Departamento de Ingeniería de Sistemas y Computación



Taller #1

ISIS1611 – Inteligencia Artificial

Semestre 2026-1

Integrantes:

Mateo Rincon - 202221402

Silvana Echeverry - 202310470

Sofía Morato - 202321074

Juan Esteban Jiménez Benavides - 201922487

Tabla de contenido

| | |
|---|-----------|
| Punto 1: Respuesta a Baliza de emergencia..... | 3 |
| Complejidad en tiempo (Notación O) | 3 |
| Complejidad en espacio (Notación O) | 4 |
| ¿El algoritmo es completo? | 4 |
| ¿El algoritmo encuentra la solución óptima siempre o en cuáles condiciones? | 5 |
| Punto 2: Camino mas corto a un sobreviviente | 5 |
| Complejidad en tiempo (Notación O) | 5 |
| Complejidad en espacio (Notación O) | 6 |
| ¿El algoritmo es completo? | 7 |
| ¿El algoritmo encuentra la solución óptima siempre o en cuáles condiciones? | 7 |
| Punto 3: Navegación en terreno peligroso | 7 |
| Complejidad en Tiempo (Notación O)..... | 7 |
| Complejidad en espacio (Notación O) | 8 |
| ¿El algoritmo es completo? | 9 |
| ¿El algoritmo encuentra la solución óptima siempre o en cuáles condiciones? | 9 |
| Punto 4: Búsqueda Informada Eficiente..... | 9 |
| Complejidad en Tiempo (Notación O)..... | 9 |
| Complejidad en espacio (Notación O) | 10 |
| ¿El algoritmo es completo? | 11 |
| ¿El algoritmo encuentra la solución óptima siempre o en cuáles condiciones? | 11 |
| ¿Las heurísticas son consistentes? | 11 |
| Punto 5: Rescate de Múltiples Sobrevivientes | 12 |
| Complejidad en Tiempo (Notación O)..... | 12 |
| Complejidad en espacio (Notación O) | 13 |
| ¿El algoritmo es completo? | 13 |
| ¿El algoritmo encuentra la solución óptima siempre o en cuáles condiciones? | 14 |
| ¿Las heurísticas son consistentes? | 14 |
| Punto 6: Reflexión sobre la co-construcción de la solución final con apoyo de la IAG | 14 |

| | |
|--|----|
| Uso y Trazabilidad de la IA | 14 |
| Justificación de los cambios realizados con el apoyo de la IA..... | 15 |
| Reflexión sobre beneficios y limitaciones de la IA | 15 |

Punto 1: Respuesta a Baliza de emergencia

Para el análisis de complejidad de tiempo y espacial usaremos la siguiente tabla de variables:

| | |
|---|---|
| V | # de estados alcanzados (nodos del grafo) |
| E | # de transiciones (arcos) |
| B | Branching factor (sucesores por estado). El robot solo puede moverse a 4 posiciones es $B \leq 4$ |
| m | Profundidad Máxima |
| L | Longitud del camino (lista actual) |

Complejidad en tiempo (Notación O)

| | |
|---|--|
| <pre> stack = utils.Stack() visited = set() # Cada elemento del stack será: (estado, lista_de_acciones) start_state = problem.getStartState() stack.push((start_state, [])) </pre> | O(1) es una asignación |
| <pre> while not stack.isEmpty(): </pre> | O(V) en el peor caso el while se expande a todos los nodos alcanzables |
| <pre> state, actions = stack.pop() if problem.isGoalState(state): return actions if state not in visited: visited.add(state) </pre> | O(1) |

| | |
|--|---|
| <pre>for successor, action, cost in problem.getSuccessors(state):</pre> | $O(B \cdot V)$ se ejecuta V veces por cada nodo expandido y cada nodo tiene hasta B sucesores. Es aproximadamente E |
| <pre> new_actions = actions + [action] stack.push((successor, new_actions))</pre> | Copiar una lista es $O(L)$. en el peor caso es $L \leq m$. En el peor caso es $O(m)$. Entonces es: $O(E \cdot m)$ |

Como B es una constante pequeña, se puede simplificar la complejidad a esta ecuación:
 $O(Vm)$

Complejidad en espacio (Notación O)

Para este punto utilizamos 3 estructuras de datos que consumen memoria

| | |
|-----------------------|--|
| Stack | $O(V)$ |
| Lista de Visited | $O(V)$ |
| Lista de las acciones | Copiar una lista es $O(L)$. en el peor caso es $L \leq m$. En el peor caso es $O(m)$. Entonces es: $O(V \cdot m)$ |

Por lo tanto, la complejidad espacial se da por $O(Vm) \sim O(V)$.

¿El algoritmo es completo?

Un algoritmo es completo si garantiza encontrar una solución siempre que esta exista en el espacio de estados. En el caso de nuestra implementación de DFS, se utiliza un conjunto `visited` para registrar los estados ya explorados. Dado que el espacio de estados es finito (una grilla finita) y se evita revisitar estados, el algoritmo eventualmente explorará todos los estados alcanzables. Si no se utilizara `visited`, el algoritmo podría caer en ciclos y no garantizar la terminación, lo que afectaría su completitud. Sin embargo, en nuestra implementación sí se controla este aspecto. En conclusión, el algoritmo es completo, ya que en un espacio finito y evitando ciclos, siempre encontrará una solución si esta existe.

¿El algoritmo encuentra la solución óptima siempre o en cuáles condiciones?

Un algoritmo es óptimo si garantiza encontrar la solución de menor costo. DFS explora primero en profundidad, siguiendo un camino hasta el final antes de retroceder. Debido a esta estrategia, puede encontrar una solución que no sea la de menor costo ni la de menor número de pasos. Por ejemplo, puede avanzar por un camino muy largo o costoso antes de explorar alternativas más cortas o económicas. Por esta razón, DFS no garantiza optimalidad, independientemente de si los costos son uniformes o variables. En conclusión, El algoritmo no es óptimo, ya que no asegura encontrar la solución de menor costo, aunque sí es completo en este contexto.

Punto 2: Camino mas corto a un sobreviviente

Para el análisis de complejidad de tiempo y espacial usaremos la siguiente tabla de variables:

| | |
|---|---|
| V | # de estados alcanzados (nodos del grafo) |
| E | # de transiciones (arcos) |
| B | Branching factor (sucesores por estado). El robot solo puede moverse a 4 posiciones es $B \leq 4$ |

Complejidad en tiempo (Notación O)

| | |
|--|--|
| <pre>fila = utils.Queue() visited = set() start_state = problem.getStartState() fila.push((start_state, []))</pre> | O(1) |
| <pre>while not fila.isEmpty():</pre> | O(V) en el peor caso el while se expande a todos los nodos alcanzables |

| | |
|--|--|
| <pre> state, actions = fila.pop() if problem.isGoalState(state): return actions if state not in visited: visited.add(state) </pre> | O(1) |
| <pre> for successor, action, cost in problem.getSuccessors(state): ... </pre> | O($B \cdot V$) se ejecuta V veces por cada nodo expandido y cada nodo tiene hasta B sucesores. |
| <pre> new_actions = actions + [action] fila.push((successor, new_actions)) </pre> | Copiar una lista es $O(L)$. en el peor caso es $L \leq m$. En el peor caso es $O(m)$. Entonces es: $O(V \cdot m)$ |

Como B es una constante pequeña, se puede simplificar la complejidad a esta ecuación: $O(Vm)$

Complejidad en espacio (Notación O)

Para este punto utilizamos 3 estructuras de datos que consumen memoria

| | |
|-----------------------|--|
| Stack | $O(V)$ |
| Lista de Visited | $O(V)$ |
| Lista de las acciones | Copiar una lista es $O(L)$. en el peor caso es $L \leq m$. En el peor caso es $O(m)$. Entonces es: $O(V \cdot m)$ |

La complejidad espacial de BFS es $O(Vm) \sim O(V)$, ya que almacena los estados visitados y la frontera. Sin embargo, en términos de profundidad del árbol de búsqueda, puede

alcanzar $O(B^d)$, (d siendo la profundidad) lo cual es exponencial y generalmente mayor que la de DFS que es $O(bm)$

¿El algoritmo es completo?

Un algoritmo es completo si garantiza encontrar una solución siempre que esta exista en el espacio de estados. En el caso de nuestra implementación de DBFS, se utiliza un conjunto `visited` para registrar los estados ya explorados. Dado que el espacio de estados es finito (una grilla finita) y se evita revisitar estados, el algoritmo eventualmente explorará todos los estados alcanzables. Si no se utilizara `visited`, el algoritmo podría caer en ciclos y no garantizar la terminación, lo que afectaría su completitud. Sin embargo, en nuestra implementación sí se controla este aspecto. En conclusión, el algoritmo es completo, ya que en un espacio finito y evitando ciclos, siempre encontrará una solución si esta existe.

¿El algoritmo encuentra la solución óptima siempre o en cuáles condiciones?

BFS es óptimo únicamente cuando todas las acciones tienen el mismo costo, ya que garantiza encontrar el camino con menor número de pasos. Sin embargo, cuando los costos de transición son variables, como en este problema, BFS no garantiza minimizar el costo total, sino únicamente la profundidad del camino. Por lo tanto, en este contexto, el nos piden contar la cantidad de pasos y no tenemos en cuenta el costo (los costos serian iguales) el algoritmo si es óptimo.

Punto 3: Navegación en terreno peligroso

| | |
|---|---|
| V | # de estados alcanzados (nodos del grafo) |
| E | # de transiciones (arcos) |

Complejidad en Tiempo (Notación O)

| | |
|---|----------|
| <pre>fila = utils.PriorityQueue() visited = set()</pre> | O(1) |
| <pre>start_state = problem.getStartState() fila.push(start_state, 0, 0)</pre> | O(1) |
| <pre>fila.push((start_state, [], 0), 0)</pre> | O(log E) |
| <pre>best_cost = {start_state: 0}</pre> | O(1) |

| | |
|--|-------------|
| <code>while not fila.isEmpty():</code> | $O(E)$ |
| <code> state, actions, costo = fila.pop()</code> | $O(\log E)$ |
| <code> if state not in visited: visited.add(state) if problem.isGoalState(state): return actions</code> | $O(1)$ |
| <code> for successor, action, cost in problem.getSuccessors(state):</code> | $O(E)$ |
| <code> nuevo_costo=costo+cost if nuevo_costo < best_cost.get(successor, float("inf")): best_cost[successor] = nuevo_costo</code> | $O(1)$ |
| <code> new_actions = actions + [action]</code> | $O(V)$ |
| <code> fila.push((successor, new_actions, nuevo_costo),nuevo_costo)</code> | $O(\log E)$ |
| <code> return []</code> | $O(1)$ |

Por lo tanto, la complejidad temporal esta dada por: $O(EV+E\log E) \approx O(EV)$ donde V es el número de estados y E el número de transiciones. Aunque las operaciones de la cola de prioridad aportan un costo $O(E \log E)$, el término dominante proviene de la copia de la lista de acciones en cada mejora de costo. Como pueden existir hasta $O(E)$ push en la cola y cada copia puede costar hasta $O(V)$ en el peor caso, el tiempo total en el peor caso seria $O(EV)$.

Complejidad en espacio (Notación O)

| | |
|-----------------------|--|
| Priority Queue | $O(EV)$ |
| Lista de Visited | $O(V)$ |
| Best_cost | $O(V)$ |
| Lista de las acciones | Copiar una lista es $O(L)$. en el peor caso es $L \leq m$. En el peor caso es $O(m)$. Entonces es: $O(V \cdot m)$ |

La complejidad espacial de UCS es $O(EV)$ donde V es el numero de estados y E el numero de transiciones. Ya que aunque el `visited` y el `best_cost` requieren únicamente $O(V)$ espacio, la `PriorityQueue` en peor caso la cola puede contener $O(E)$ entradas (duplicadas) y cada entrada guarda una lista `actions` de tamaño hasta $O(V)$ y el espacio total queda dominado por esta cola por lo que su complejidad espacial es $O(EV)$.

¿El algoritmo es completo?

Un algoritmo es completo si garantiza encontrar una solución siempre que esta exista. UCS expande los nodos en orden creciente de costo acumulado y, dado que en este problema todos los costos son positivos, el costo siempre aumenta a medida que se profundiza en el árbol. Además, el espacio de estados es finito y se utiliza un conjunto `visited` para evitar ciclos. UCS es completo en este problema, ya que con costos positivos y espacio finito garantiza encontrar una solución si esta existe.

¿El algoritmo encuentra la solución óptima siempre o en cuáles condiciones?

UCS es óptimo cuando todos los costos de las acciones son no negativos (y especialmente positivos, como en este caso). Esto se debe a que siempre expande primero el nodo con menor costo acumulado, lo que asegura que la primera vez que alcanza el estado objetivo, lo hace con el menor costo posible. En este problema, como los costos de transición son positivos (1, 2, 3 y 5), se cumplen las condiciones necesarias para la optimalidad. UCS es óptimo en este contexto, ya que los costos son positivos y el algoritmo siempre selecciona el camino de menor costo acumulado.

Punto 4: Búsqueda Informada Eficiente

| | |
|-----|---|
| V | # de estados alcanzados (nodos del grafo) |
| E | # de transiciones (arcos) |
| H | Heuristica = $O(1)$ |

Complejidad en Tiempo (Notación O)

| | |
|---|--------|
| <pre>fila = utils.PriorityQueue() visited = set()</pre> | $O(1)$ |
|---|--------|

| | |
|--|----------|
| <code>start_state = problem.getStartState()</code> | O(1) |
| <code>fila.push((start_state, [], 0),0)</code> | O(log E) |
| <code>while not fila.isEmpty():</code> | O(E) |
| <code> state, actions, costo = fila.pop()</code> | O(log E) |
| <code> if problem.isGoalState(state):</code> <code> return actions</code> <code> if state not in visited:</code> <code> visited.add(state)</code> | O(1) |
| <code> for successor, action, cost in problem.getSuccessors(state):</code> | O(E) |
| <code> new_actions = actions + [action]</code> | O(V) |
| <code> CostoHeuristica=costo+heuristic(successor, problem)</code> | O(H) |
| <code> nuevo_costo=costo+cost</code> | O(1) |
| <code> fila.push((successor, new_actions, nuevo_costo),CostoHeuristica)</code> | O(log E) |
| <code> return []</code> | O(1) |

Por lo tanto, la complejidad temporal esta dada por: $O(EV+E\log E + O(H)) \approx O(EV)$ donde V es el número de estados y E el número de transiciones. Aunque las operaciones de la cola de prioridad aportan un costo $O(E \log E)$ y la heuristica un costo de $O(H)$ el término dominante proviene de la copia de la lista de acciones en cada iteración. Como pueden existir hasta $O(E)$ push en la cola y cada copia puede costar hasta $O(V)$ en el peor caso, el tiempo total en el peor caso seria $O(EV)$.

Complejidad en espacio (Notación O)

| | |
|-----------------------|-----------|
| Priority Queue | $O(EV)$ |
| Lista de Visited | $O(V)$ |
| Lista de las acciones | $O(EV)$. |

La complejidad espacial de A estrella es $O(EV)$ donde V es el numero de estados y E el numero de transiciones. Ya que aunque el visited requieren únicamente $O(V)$ espacio, la PriorityQueue en peor caso la cola puede contener $O(E)$ entradas (duplicadas) y cada entrada guarda una lista actions de tamaño hasta $O(V)$ y el espacio total queda dominado por esta cola por lo que su complejidad espacial es $O(EV)$

¿El algoritmo es completo?

Un algoritmo es completo si garantiza encontrar una solución siempre que esta exista en el espacio de estados. En el caso de nuestra implementación de A^* , se utiliza un conjunto visited para registrar los estados ya explorados y evitar ciclos. Dado que el espacio de estados es finito (una grilla finita) y los costos de las acciones son no negativos, el algoritmo eventualmente explorará todos los estados alcanzables que sean necesarios según la función de evaluación. Por lo tanto, mientras exista una solución alcanzable, el algoritmo terminará encontrándola. En conclusión, A^* es completo en este contexto, ya que el espacio es finito y se evita la repetición infinita de estados.

¿El algoritmo encuentra la solución óptima siempre o en cuáles condiciones?

Un algoritmo es óptimo si garantiza encontrar la solución de menor costo. A^* puede ser óptimo cuando la heurística utilizada es admisible (es decir, nunca sobreestima el costo real hasta la meta) y cuando se manejan correctamente los costos acumulados de cada estado. Sin embargo, en nuestra implementación se utiliza un conjunto visited que impide reabrir estados si posteriormente se encuentra un camino más barato hacia ellos. Debido a esto, el algoritmo podría descartar una mejor solución descubierta después, lo que afecta la garantía de optimalidad. En conclusión, en esta implementación A^* no garantiza siempre encontrar la solución óptima, aunque podría hacerlo bajo condiciones adecuadas de heurística y manejo de costos.

¿Las heurísticas son consistentes?

En el caso de la heurística Manhattan, esta calcula la distancia en términos de movimientos horizontales y verticales hasta la meta. En una grilla donde los movimientos son en cuatro direcciones y el costo mínimo por paso es mayor o igual a 1, la distancia Manhattan nunca disminuye más que el costo real de un movimiento. Por lo tanto, cumple la condición de consistencia siempre que los costos de las acciones sean no negativos.

De manera similar, la heurística Euclidiana calcula la distancia en línea recta hasta la meta. En una grilla con costos positivos, esta distancia tampoco sobreestima el costo real

del camino óptimo y respeta la desigualdad triangular, por lo que también resulta consistente bajo estas condiciones.

En conclusión, tanto la heurística Manhattan como la Euclídea son consistentes en este problema siempre que los costos de movimiento sean positivos, como ocurre en nuestro caso.

Punto 5: Rescate de Múltiples Sobrevivientes

Nota: se utiliza el A* del punto 4

| | |
|---|---|
| V | # de estados alcanzados (nodos del grafo) |
| E | # de transiciones (arcos) |
| H | Heurística = $O(1)$ |

Complejidad en Tiempo (Notación O)

| | |
|---|-------------|
| <code>fila = utils.PriorityQueue() visited = set()</code> | $O(1)$ |
| <code>start_state = problem.getStartState()</code> | $O(1)$ |
| <code>fila.push((start_state, [], 0), 0)</code> | $O(\log E)$ |
| <code>while not fila.isEmpty():</code> | $O(E)$ |
| <code> state, actions, costo = fila.pop()</code> | $O(\log E)$ |
| <code> if problem.isGoalState(state): return actions</code> | $O(1)$ |
| <code> if state not in visited: visited.add(state)</code> | |
| <code> for successor, action, cost in problem.getSuccessors(state):</code> | $O(E)$ |
| <code> new_actions = actions + [action]</code> | $O(V)$ |
| <code> CostoHeuristica=costo+heuristic(successor, problem)</code> | $O(H)$ |

| | |
|--|----------|
| <code>nuevo_costo=costo+cost</code> | O(1) |
| <code>fila.push(successor, new_actions, nuevo_costo),CostoHeuristica)</code> | O(log E) |
| <code>return []</code> | O(1) |

Por lo tanto, la complejidad temporal esta dada por: $O(EV+E\log E + O(H)) \approx O(EV)$ donde V es el número de estados y E el número de transiciones. Aunque las operaciones de la cola de prioridad aportan un costo $O(E \log E)$ y la heuristica un costo de $O(H)$ el término dominante proviene de la copia de la lista de acciones en cada iteración. Como pueden existir hasta $O(E)$ push en la cola y cada copia puede costar hasta $O(V)$ en el peor caso, el tiempo total en el peor caso seria $O(EV)$.

Complejidad en espacio (Notación O)

| | |
|-----------------------|-----------|
| Priority Queue | $O(EV)$ |
| Lista de Visited | $O(V)$ |
| Lista de las acciones | $O(EV)$. |

La complejidad espacial de A estrella es $O(EV)$ donde V es el numero de estados y E el numero de transiciones. Ya que aunque el visited requieren únicamente $O(V)$ espacio, la PriorityQueue en peor caso la cola puede contener $O(E)$ entradas (duplicadas) y cada entrada guarda una lista actions de tamaño hasta $O(V)$ y el espacio total queda dominado por esta cola por lo que su complejidad espacial es $O(EV)$

¿El algoritmo es completo?

Un algoritmo es completo si garantiza encontrar una solución siempre que esta exista en el espacio de estados. En el caso de nuestra implementación de A^* , se utiliza un conjunto visited para registrar los estados ya explorados y evitar ciclos. Dado que el espacio de estados es finito (una grilla finita) y los costos de las acciones son no negativos, el algoritmo eventualmente explorará todos los estados alcanzables que sean necesarios según la función de evaluación. Por lo tanto, mientras exista una solución alcanzable, el algoritmo terminará encontrándola. En conclusión, A^* es completo en este contexto, ya que el espacio es finito y se evita la repetición infinita de estados.

¿El algoritmo encuentra la solución óptima siempre o en cuáles condiciones?

Un algoritmo es óptimo si garantiza encontrar la solución de menor costo. A* puede ser óptimo cuando la heurística utilizada es admisible (es decir, nunca sobreestima el costo real hasta la meta) y cuando se manejan correctamente los costos acumulados de cada estado. Sin embargo, en nuestra implementación se utiliza un conjunto `visited` que impide reabrir estados si posteriormente se encuentra un camino más barato hacia ellos. Debido a esto, el algoritmo podría descartar una mejor solución descubierta después, lo que afecta la garantía de optimalidad. En conclusión, en esta implementación A* no garantiza siempre encontrar la solución óptima, aunque podría hacerlo bajo condiciones adecuadas de heurística y manejo de costos.

¿Las heurísticas son consistentes?

En el punto 5 se diseñó una heurística más informada para el problema de rescate múltiple. La función calcula, para cada estado, la distancia Manhattan desde la posición actual al sobreviviente más cercano y le suma el costo de un árbol de expansión mínima (MST) que conecta a todos los sobrevivientes restantes, usando también distancias Manhattan como pesos. La idea es aproximar el costo mínimo necesario para “visitar” a todos, similar a un problema tipo TSP en grilla. Esta heurística es admisible porque tanto la distancia al más cercano como el MST son subestimadores del costo real: el camino verdadero nunca puede ser menor que la distancia Manhattan entre puntos y cada movimiento tiene costo positivo. En cuanto a consistencia, aunque la parte basada en Manhattan se comporta de manera monótona, el término del MST puede variar cuando cambia el conjunto de sobrevivientes, por lo que no se puede garantizar formalmente consistencia estricta en todos los casos, aunque en la práctica funciona de manera estable y más informada que la versión inicial que solo consideraba el sobreviviente más cercano.

Punto 6: Reflexión sobre la co-construcción de la solución final con apoyo de la IAG

Uso y Trazabilidad de la IA

Durante el desarrollo del taller se utilizó la Inteligencia Artificial Generativa (IAG) como herramienta de apoyo principalmente para comprender mejor el enunciado del problema, las estructuras base proporcionadas y los conceptos teóricos asociados a los algoritmos de búsqueda (DFS, BFS, UCS y A*). La IA también fue empleada para analizar versiones

preliminares del código, sugerir mejoras estructurales, validar decisiones de diseño y revisar los análisis de complejidad temporal y espacial. En particular, fue útil para fortalecer la heurística del MultiSurvivorProblem, donde se propuso una versión más informada basada en distancia Manhattan y un árbol de expansión mínima (MST). En todo momento se mantuvo el control del código y de las decisiones finales, revisando manualmente cada sugerencia antes de incorporarla.

Justificación de los cambios realizados con el apoyo de la IA

La IA contribuyó principalmente en la optimización y mejora de soluciones iniciales y en análisis conceptual. Por ejemplo, ayudó a estructurar una heurística más robusta para el problema de múltiples sobrevivientes, combinando la distancia al sobreviviente más cercano con el costo de un MST como cota inferior. Sin embargo, no todas las sugerencias fueron correctas. En la versión propuesta del punto 3, la IA propuso una implementación donde un estado se insertaba en la cola de prioridad la primera vez que aparecía, en lugar de actualizarse cuando se encontraba un menor costo acumulado. Este enfoque habría comprometido la optimalidad del algoritmo UCS/A*, ya que no garantizaba expandir siempre el estado con menor costo. Esto evidenció la necesidad de revisar críticamente las propuestas generadas por la IA y validar su coherencia con los fundamentos teóricos vistos en la clase.

Reflexión sobre beneficios y limitaciones de la IA

La experiencia mostró que la IA es una herramienta poderosa para acelerar la comprensión de conceptos, estructurar ideas y mejorar la calidad técnica de las soluciones. Resultó especialmente útil para clarificar dudas sobre complejidad algorítmica y diseño de heurísticas. No obstante, también se evidenció que la IA puede cometer errores conceptuales o sugerir implementaciones que, aunque aparentemente correctas, no cumplen con propiedades fundamentales como la optimalidad o consistencia. Por ello, confiar ciegamente en sus respuestas puede llevar a soluciones incorrectas. En conclusión, la IA debe utilizarse como una herramienta de apoyo y co-construcción del conocimiento, pero siempre complementada con criterio propio, comprensión teórica y validación rigurosa. Su valor no reemplaza el razonamiento del estudiante, sino que potencia el proceso cuando se usa de manera crítica y responsable.