Proyecto 1

Juan Pablo Ante 2140132

Nicolas Garcés 2180066

Alejandro Guerrero 2179652

Alejandro Zamorano 1941088

Universidad del Valle

Inteligencia Artificial

11 de Abril del 2024

Búsqueda no informada

Amplitud

El algoritmo de búsqueda por amplitud consiste en explorar todos los nodos adyacentes a un nodo dado antes de avanzar a los nodos adyacentes de los nodos ya explorados. Este enfoque se conoce como exploración en anchura o por niveles, y es característico de este algoritmo.

```python
def busqueda_amplitud(ambiente):
    inicio = time.time()
    nodo = Nodo(ambiente)
    queue = deque([nodo])  # Usar una cola para almacenar nodos a explorar
    explorados = set()  # Conjunto para almacenar estados explorados
    nodos_expandidos = []  # Lista para almacenar los nodos expandidos
    while queue:
        nodo_actual = queue.popleft()  # Obtener el nodo más antiguo de la cola

        if es_nodo_meta(nodo_actual):
            tiempo_total = time.time() - inicio
            return reconstruir_camino(nodo_actual), "Se encontró", nodos_expandidos, nodo_actual.profundidad, tiempo_total

        estado_actual = str(nodo_actual.estado.matriz)  # Convertir la matriz a una cadena para usarla como clave

        if estado_actual in explorados or evitar_ciclos(nodo_actual):
            continue  # Evitar nodos ya explorados y ciclos

        explorados.add(estado_actual)  # Agregar el estado actual al conjunto de explorados
        nodos_expandidos.append(nodo_actual)  # Agregar el nodo actual a la lista de nodos expandidos

        for accion in nodo_actual.estado.mando.get_movimientos_posibles(nodo_actual.estado.matriz):
            nuevo_estado = nodo_actual.estado.copy()
            nuevo_estado.transicion(accion)
            nuevo_nodo = Nodo(nuevo_estado, nodo_actual, accion, nodo_actual.profundidad + 1)
            queue.append(nuevo_nodo)  # Agregar el nuevo nodo a la cola

    return [], "No se encontró", nodos_expandidos
```

- Esta función toma como argumento ambiente, que representa el estado inicial del problema que se quiere resolver.
- Crea un bucle que se ejecutará mientras haya nodos en la cola queue para explorar.
- Obtiene y remueve el nodo más antiguo de la cola, utilizando popleft() para garantizar que se explore en orden de amplitud.
- Comprueba si el nodo actual es el nodo meta (es decir, si corresponde a una solución al problema).
- Si se encuentra el nodo meta, calcula el tiempo total de ejecución, reconstruye el camino desde el nodo inicial hasta el nodo meta y lo retorna junto con un mensaje indicando que se encontró la solución, la lista de nodos expandidos, la profundidad del nodo meta y el tiempo total de ejecución.
- Si el estado actual no ha sido explorado y no se está formando un ciclo, se marca como explorado y se expande. Se generan todos los estados alcanzables desde el estado actual y se agregan a la cola para su exploración posterior.

Profundidad

El algoritmo de búsqueda por profundidad es un enfoque para explorar un grafo o un árbol donde el proceso comienza en el nodo inicial y avanza lo más lejos posible a lo largo de cada rama antes de retroceder. Es decir, explora cada rama hasta que alcanza el nodo más profundo posible antes de retroceder al nodo anterior y continuar explorando otras ramas.

```python
def busqueda_profundidad(ambiente):
    inicio = time.time()
    nodo = Nodo(ambiente)
    stack = [nodo]  # Usar una pila para almacenar nodos a explorar
    explorados = set()  # Conjunto para almacenar estados explorados
    nodos_expandidos = []  # Lista para almacenar los nodos expandidos

    while stack:
        nodo_actual = stack.pop()  # Obtener el nodo más reciente de la pila

        if es_nodo_meta(nodo_actual):
            tiempo_total = time.time() - inicio
            return reconstruir_camino(nodo_actual), "Se encontró", nodos_expandidos, nodo_actual.profundidad, tiempo_total

        estado_actual = str(nodo_actual.estado.matriz)  # Convertir la matriz a una cadena para usarla como clave

        if estado_actual in explorados or evitar_ciclos(nodo_actual):
            continue  # Evitar nodos ya explorados y ciclos

        explorados.add(estado_actual)  # Agregar el estado actual al conjunto de explorados
        nodos_expandidos.append(nodo_actual)  # Agregar el nodo actual a la lista de nodos expandidos

        for accion in nodo_actual.estado.mando.get_movimientos_posibles(nodo_actual.estado.matriz):
            nuevo_estado = nodo_actual.estado.copy()
            nuevo_estado.transicion(accion)
            nuevo_nodo = Nodo(nuevo_estado, nodo_actual, accion, nodo_actual.profundidad + 1)
            stack.append(nuevo_nodo)  # Agregar el nuevo nodo a la pila

    return [], "No se encontró", nodos_expandidos
```

- Esta función toma como argumento ambiente, que representa el estado inicial del problema que se quiere resolver.
- Este bucle se ejecutará mientras haya nodos en la pila stack para explorar.
- Obtiene y elimina el nodo más reciente de la pila, utilizando pop() para garantizar que se explore en profundidad.
- Comprueba si el nodo actual es el nodo meta
- Si se encuentra el nodo meta, calcula el tiempo total de ejecución, reconstruye el camino desde el nodo inicial hasta el nodo meta y lo retorna junto con un mensaje indicando que

se encontró la solución, la lista de nodos expandidos, la profundidad del nodo meta y el tiempo total de ejecución.

- Si el estado actual no ha sido explorado y no se está formando un ciclo, se marca como explorado y se expande. Se generan todos los estados alcanzables desde el estado actual y se agregan a la pila para su exploración posterior.

Costo uniforme

El algoritmo de búsqueda de costo uniforme es una estrategia para encontrar la solución óptima en un grafo o un árbol ponderado. A diferencia de otros algoritmos, no solo considera la profundidad de un nodo, sino también el costo acumulado para llegar a ese nodo desde el inicio. Comienza explorando el nodo inicial y expande los nodos con el costo más bajo primero.

```python
def busqueda_costo_uniforme(ambiente):
    inicio = time.time()
    nodo = Nodo(ambiente)
    queue = PriorityQueue()  # Usar una cola de prioridad para almacenar nodos a explorar
    queue.put((nodo.costo, nodo))  # Insertar el nodo inicial en la cola de prioridad
    explorados = set()  # Conjunto para almacenar estados explorados
    nodos_expandidos = []  # Lista para almacenar los nodos expandidos

    while not queue.empty():
        _, nodo_actual = queue.get()  # Obtener el nodo con el menor costo acumulado de la cola de prioridad

        if es_nodo_meta(nodo_actual):
            tiempo_total = time.time() - inicio
            return reconstruir_camino(nodo_actual), "Se encontró", nodos_expandidos, nodo_actual.profundidad, tiempo_total, nodo_actual.costo

        estado_actual = str(nodo_actual.estado.matriz)  # Convertir la matriz a una cadena para usarla como clave

        if estado_actual in explorados or evitar_ciclos(nodo_actual):
            continue  # Evitar nodos ya explorados y ciclos

        explorados.add(estado_actual)  # Agregar el estado actual al conjunto de explorados
        nodos_expandidos.append(nodo_actual)  # Agregar el nodo actual a la lista de nodos expandidos

        for accion in nodo_actual.estado.mando.get_movimientos_posibles(nodo_actual.estado.matriz):
            nuevo_estado = nodo_actual.estado.copy()
            nuevo_estado.transicion(accion)
            if es_nave(nodo_actual) and not es_enemigo(nodo_actual):
                nuevo_nodo = Nodo(nuevo_estado, nodo_actual, accion, nodo_actual.profundidad + 1, nodo_actual.costo + 1/2, nodo_actual.contado
                if nuevo_nodo.contador_pasos >= 10:
                    nuevo_nodo.paso_por_nave = False
                    nuevo_nodo.contador_pasos = 0

            if es_nave(nodo_actual) and es_enemigo(nodo_actual):
                nuevo_nodo = Nodo(nuevo_estado, nodo_actual, accion, nodo_actual.profundidad + 1, nodo_actual.costo + 1/2, nodo_actual.contado
            elif es_enemigo(nodo_actual):
                nuevo_nodo = Nodo(nuevo_estado, nodo_actual, accion, nodo_actual.profundidad + 1, nodo_actual.costo + 5, nodo_actual.contador_
            if not es_enemigo(nodo_actual) and not es_nave(nodo_actual):
                nuevo_nodo = Nodo(nuevo_estado, nodo_actual, accion, nodo_actual.profundidad + 1, nodo_actual.costo + 1, nodo_actual.contador_
            queue.put((nuevo_nodo.costo, nuevo_nodo))  # Agregar el nuevo nodo a la cola de prioridad

    return [], "No se encontró", nodos_expandidos
```

- Se define la función busqueda_costo_uniforme, que toma como argumento ambiente, que representa el estado inicial del problema.
- Este bucle se ejecutará mientras la cola de prioridad queue no esté vacía, lo que significa que aún hay nodos por explorar.
- Se obtiene y elimina de la cola de prioridad el nodo con el menor costo acumulado. El guion bajo _ se utiliza para descartar el costo, ya que solo nos interesa el nodo en sí.

- Se verifica si el nodo actual es el nodo meta (es decir, si corresponde a una solución al problema).
- Si se encuentra el nodo meta, se calcula el tiempo total de ejecución, se reconstruye el camino desde el nodo inicial hasta el nodo meta y se retorna junto con un mensaje indicando que se encontró la solución, la profundidad del nodo meta, el tiempo total de ejecución y el costo acumulado del nodo meta.
- Se verifica si el estado actual ya ha sido explorado o si se está formando un ciclo. Si es así, se omite la exploración de este nodo.Se agrega el estado actual al conjunto de estados explorados y se añade el nodo actual a la lista de nodos expandidos.
- Se itera sobre las acciones posibles desde el nodo actual y se generan nuevos nodos basados en estas acciones.Cada nuevo nodo se agrega a la cola de prioridad, utilizando su
- costo acumulado como clave de ordenamiento.

Búsqueda informada

Avara

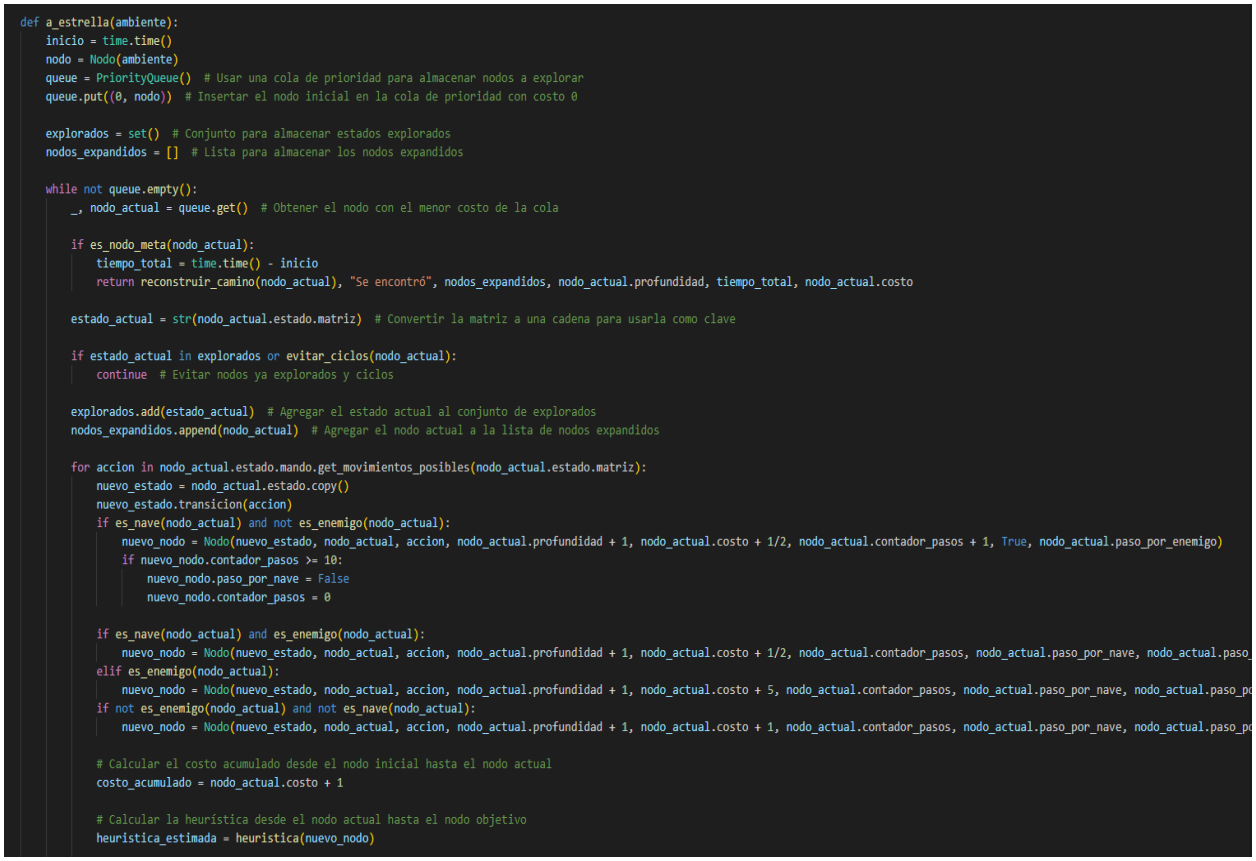
La búsqueda avara es un algoritmo de búsqueda informada que selecciona el nodo más prometedor en función de una función heurística. Aunque no garantiza la solución óptima, es eficiente en problemas bien definidos con una buena función heurística.

```python
def busqueda_avara(ambiente):
    inicio = time.time()
    nodo = Nodo(ambiente)
    queue = PriorityQueue()  # Usar una cola de prioridad para almacenar nodos a explorar
    queue.put((heuristica(nodo), nodo))  # Insertar el nodo inicial en la cola de prioridad

    explorados = set()  # Conjunto para almacenar estados explorados
    nodos_expandidos = []  # Lista para almacenar los nodos expandidos

    while not queue.empty():
        _, nodo_actual = queue.get()  # Obtener el nodo con la menor heurística de la cola

        if es_nodo_meta(nodo_actual):
            tiempo_total = time.time() - inicio
            return reconstruir_camino(nodo_actual), "Se encontró", nodos_expandidos, nodo_actual.profundidad, tiempo_total

        estado_actual = str(nodo_actual.estado.matriz)  # Convertir la matriz a una cadena para usarla como clave

        if estado_actual in explorados or evitar_ciclos(nodo_actual):
            continue  # Evitar nodos ya explorados y ciclos

        explorados.add(estado_actual)  # Agregar el estado actual al conjunto de explorados
        nodos_expandidos.append(nodo_actual)  # Agregar el nodo actual a la lista de nodos expandidos

        for accion in nodo_actual.estado.mando.get_movimientos_posibles(nodo_actual.estado.matriz):
            nuevo_estado = nodo_actual.estado.copy()
            nuevo_estado.transicion(accion)
            nuevo_nodo = Nodo(nuevo_estado, nodo_actual, accion, nodo_actual.profundidad + 1)
            queue.put((heuristica(nuevo_nodo), nuevo_nodo))  # Insertar el nuevo nodo en la cola de prioridad

    return [], "No se encontró", nodos_expandidos
```

- Se define la función busqueda_avara, que toma como argumento ambiente, que representa el estado inicial del problema.
- Se crea un bucle que se ejecutará mientras la cola de prioridad queue no esté vacía, lo que significa que aún hay nodos por explorar.
- Se obtiene y elimina de la cola de prioridad el nodo con la menor heurística. El guión bajo se utiliza para descartar la heurística, ya que solo nos interesa el nodo en sí.
- Se verifica si el nodo actual es el nodo meta (es decir, si corresponde a una solución al problema).
- Si se encuentra el nodo meta, se calcula el tiempo total de ejecución, se reconstruye el camino desde el nodo inicial hasta el nodo meta y se retorna junto con un mensaje indicando que se encontró la solución, la profundidad del nodo meta y el tiempo total
- Se verifica si el estado actual ya ha sido explorado o si se está formando un ciclo. Si es así, se omite la exploración de este nodo.Se agrega el estado actual al conjunto de estados explorados y se añade el nodo actual a la lista de nodos expandidos
- Se itera sobre las acciones posibles desde el nodo actual y se generan nuevos nodos basados en estas acciones.Cada nuevo nodo se agrega a la cola de prioridad, utilizando su heurística como clave de ordenamiento.

A*

A* Es un algoritmo de búsqueda informada que utiliza una función heurística para guiar la búsqueda hacia el objetivo de manera eficiente y garantiza encontrar la solución óptima si se cumplen ciertas condiciones sobre la heurística utilizada.



```
def a_estrella(ambiente):
    inicio = time.time()
    nodo = Nodo(ambiente)
    queue = PriorityQueue()  # Usar una cola de prioridad para almacenar nodos a explorar
    queue.put((0, nodo))  # Insertar el nodo inicial en la cola de prioridad con costo 0

    explorados = set()  # Conjunto para almacenar estados explorados
    nodos_expandidos = []  # Lista para almacenar los nodos expandidos

    while not queue.empty():
        _, nodo_actual = queue.get()  # Obtener el nodo con el menor costo de la cola

        if es_nodo_meta(nodo_actual):
            tiempo_total = time.time() - inicio
            return reconstruir_camino(nodo_actual), "Se encontró", nodos_expandidos, nodo_actual.profundidad, tiempo_total, nodo_actual.costo

        estado_actual = str(nodo_actual.estado.matriz)  # Convertir la matriz a una cadena para usarla como clave

        if estado_actual in explorados or evitar_ciclos(nodo_actual):
            continue  # Evitar nodos ya explorados y ciclos

        explorados.add(estado_actual)  # Agregar el estado actual al conjunto de explorados
        nodos_expandidos.append(nodo_actual)  # Agregar el nodo actual a la lista de nodos expandidos

        for accion in nodo_actual.estado.mando.get_movimientos_posibles(nodo_actual.estado.matriz):
            nuevo_estado = nodo_actual.estado.copy()
            nuevo_estado.transicion(accion)
            if es_nave(nodo_actual) and not es_enemigo(nodo_actual):
                nuevo_nodo = Nodo(nuevo_estado, nodo_actual, accion, nodo_actual.profundidad + 1, nodo_actual.costo + 1/2, nodo_actual.contador_pasos + 1, True, nodo_actual.paso_por_enemigo)
                if nuevo_nodo.contador_pasos >= 10:
                    nuevo_nodo.paso_por_nave = False
                    nuevo_nodo.contador_pasos = 0

            if es_nave(nodo_actual) and es_enemigo(nodo_actual):
                nuevo_nodo = Nodo(nuevo_estado, nodo_actual, accion, nodo_actual.profundidad + 1, nodo_actual.costo + 1/2, nodo_actual.contador_pasos, nodo_actual.paso_por_nave, nodo_actual.paso_
            elif es_enemigo(nodo_actual):
                nuevo_nodo = Nodo(nuevo_estado, nodo_actual, accion, nodo_actual.profundidad + 1, nodo_actual.costo + 5, nodo_actual.contador_pasos, nodo_actual.paso_por_nave, nodo_actual.paso_p
            if not es_enemigo(nodo_actual) and not es_nave(nodo_actual):
                nuevo_nodo = Nodo(nuevo_estado, nodo_actual, accion, nodo_actual.profundidad + 1, nodo_actual.costo + 1, nodo_actual.contador_pasos, nodo_actual.paso_por_nave, nodo_actual.paso_p

            # Calcular el costo acumulado desde el nodo inicial hasta el nodo actual
            costo_acumulado = nodo_actual.costo + 1

            # Calcular la heurística desde el nodo actual hasta el nodo objetivo
            heuristica_estimada = heuristica(nuevo_nodo)
```

- Se define la función a estrella, que toma como argumento ambiente, que representa el estado inicial del problema.
- Este bucle se ejecutará mientras la cola de prioridad queue no esté vacía, lo que significa que aún hay nodos por explorar.
- Se obtiene y elimina de la cola de prioridad el nodo con el menor costo total. El guión bajo se utiliza para descartar el costo, ya que solo nos interesa el nodo en sí.
- Se verifica si el nodo actual es el nodo meta (es decir, si corresponde a una solución al problema).
- Si se encuentra el nodo meta, se calcula el tiempo total de ejecución, se reconstruye el camino desde el nodo inicial hasta el nodo meta y se retorna junto con un mensaje

indicando que se encontró la solución, la profundidad del nodo meta, el tiempo total de ejecución y el costo acumulado del nodo meta.

- Se verifica si el estado actual ya ha sido explorado o si se está formando un ciclo. Si es así, se omite la exploración de este nodo.Se agrega el estado actual al conjunto de estados explorados y se añade el nodo actual a la lista de nodos expandidos.
- Se itera sobre las acciones posibles desde el nodo actual y se generan nuevos nodos basados en estas acciones.Se calcula el costo acumulado desde el nodo inicial hasta el nodo actual.Se calcula la heurística desde el nodo actual hasta el nodo objetivo.Se calcula el costo total del nuevo nodo como la suma del costo acumulado y la heurística.Cada nuevo nodo se agrega a la cola de prioridad, utilizando su costo total como clave de ordenamiento.

Explicación de la heurística

Distancia Manhattan entre el Mando y Grogu:

La primera parte de la heurística calcula la distancia Manhattan entre la posición del Mando y la posición de Grogu. La distancia Manhattan es la suma de las diferencias absolutas de las coordenadas en cada dimensión (fila y columna). Esta distancia proporciona una estimación de cuántos movimientos "horizontales" y "verticales" son necesarios para que el Mando alcance a Grogu.

Consideración de Naves y Enemigos:

Si hay naves o enemigos presentes en el entorno, la heurística tiene en cuenta la posición de estas entidades en relación con Grogu y el Mando.

Distancias desde Naves:

Para cada nave en el entorno, se calcula la distancia Manhattan desde la nave hasta Grogu y desde la nave hasta el Mando. Estas distancias se almacenan en listas separadas.

Distancia desde la Nave más Cercana:

Luego, se calcula la suma de la distancia más corta desde una nave hasta el Mando y la distancia más corta desde una nave hasta Grogu. Esta suma representa una estimación del costo restante desde la posición actual hasta Grogu, pasando por una nave.Si la distancia más corta desde Grogu hasta la nave es menor o igual a 5, se resta 5.

Penalización por la Presencia de Enemigos:

Si la posición actual del Mando coincide con la posición de un enemigo, se agrega un costo adicional de 10 a la distancia estimada. Esto penaliza la situación en la que el Mando está en una posición ocupada por un enemigo, lo que puede indicar un escenario más peligroso o difícil.

Retorno de la Heurística:

Finalmente, la función retorna la distancia desde la nave si hay naves o enemigos presentes, de lo contrario, retorna la distancia Manhattan entre el Mando y Grogu.En resumen, esta heurística combina la distancia directa entre el Mando y Grogu con consideraciones adicionales como la presencia de naves y enemigos, para proporcionar una estimación más precisa del costo restante para alcanzar el objetivo.

Admisibilidad de la Heurística

Subestimación:

La parte principal de la heurística es la distancia Manhattan entre el Mando y Grogu, que nunca sobreestima el costo real. La distancia Manhattan es la suma de las diferencias absolutas de las coordenadas en cada dimensión (fila y columna). Esta distancia proporciona una estimación del mínimo número de movimientos necesarios para llegar de un punto a otro en un grid.Además, cuando se consideran naves y enemigos, la heurística sigue siendo subestimada ya que se suma una distancia adicional desde una nave hasta Grogu y desde una nave hasta el Mando. Dado que el costo real para llegar de un punto a otro nunca será menor que la suma de las distancias absolutas en cada dimensión, esta parte de la heurística tampoco sobreestima el costo real.

Para demostrar la consistencia de la heurística en el contexto del algoritmo A*, necesitamos verificar que para cada nodo y cada sucesor del nodo, la estimación de la heurística desde el nodo inicial hasta el sucesor, más el costo de llegar al sucesor desde el nodo actual, es menor o igual que la estimación de la heurística desde el nodo inicial hasta el sucesor.

Para demostrar la consistencia, necesitamos demostrar que:

$$h(\text{nuevo\_nodo}) \leq (\text{nodo\_actual}, \text{nuevo\_nodo}) + h(\text{nodo\_actual})$$

Donde:

h(nuevo_nodo) es la estimación de la heurística desde el nodo inicial hasta el nuevo nodo.

c(nodo_actual,nuevo_nodo) es el costo real de llegar al nuevo nodo desde el nodo actual

h(nodo_actual) es la estimación de la heurística desde el nodo inicial hasta el nodo actual.

Dado que c(nodo_actual,nuevo_nodo) es siempre 1 en este caso (ya que cada movimiento tiene un costo de 1), podemos simplificar la desigualdad a:

$h(\text{nuevo\_nodo}) \leq 1 + h(\text{nodo\_actual})$

Lo que implica que la heurística es consistente si la estimación de la heurística desde el nodo inicial hasta el nuevo nodo es menor o igual que la estimación de la heurística desde el nodo inicial hasta el nodo actual más 1.

En la función de heurística proporcionada, la estimación se calcula principalmente en función de la distancia Manhattan entre el Mando y Grogu, y considera también la presencia de naves y enemigos. Dado que la distancia Manhattan es aditiva y la penalización por la presencia de enemigos es constante, podemos concluir que la heurística es consistente, ya que la distancia real desde el nodo actual hasta el nuevo nodo nunca será menor que la distancia Manhattan entre el Mando y Grogu más 1.

Por lo tanto, la heurística proporcionada es consistente, lo que garantiza la convergencia del algoritmo A*.