Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencias de la Computación

IIC2613 – Inteligencia Artificial
Profesores Jorge B. y Hans L.

Ayudante Coordinador Vicente Vega
Primer Semestre 2023

Tarea 2:

Algoritmos de búsqueda

Vicente Pareja

Pregunta 1

Implementacion A*

```python
def search(self):
        '''
        Performs the A* search for a solution.
        '''
        print("Computing...")

        ABIERTOS = BinaryHeap()  # Open list represented as a binary heap
        ABIERTOS_DICT = {}  # Open list represented as a dictionary
        CERRADOS = {}  # Closed list represented as a dictionary
        INITIAL = Node(self.initial_state)  # Initial node
        INITIAL.g = 0  # cost to reach the initial state
        INITIAL.h = self.heuristic(self.initial_state)  # heuristic value for the initial state
        INITIAL.key = INITIAL.g + INITIAL.h  # set the key as its priority
        ABIERTOS.insert(INITIAL)  # insert the initial node into the heap
        ABIERTOS_DICT[INITIAL.state] = INITIAL  # insert the initial node into the dictionary

        self.memory += 1
        self.generated += 1

        while not ABIERTOS.is_empty():
            MEJORNODO = ABIERTOS.extract()  # extract the node with minimum key
            ABIERTOS_DICT[MEJORNODO.state]  # delete the node from the dictionary
            CERRADOS[MEJORNODO.state] = MEJORNODO  # add this node to the closed list

            self.memory = len(ABIERTOS_DICT) + len(CERRADOS)  # update the memory usage

            # if this node is the goal state, reconstruct the path and return it
            if self.heuristic(MEJORNODO.state) == 0:
                path, actions = MEJORNODO.trace()  # Reconstruct path using the trace function
                self.end_time = time.time()
                return actions, path

            self.game.current_state = MEJORNODO.state  # update the game's current state

            SUCESORES = self.game.get_valid_moves()  # generate successors
            self.expansions += 1
            for new_state, move, _ in SUCESORES:  # unpacking the return value of get_valid_moves()
                state = new_state  # convert the new_state to a list of lists
                SUCESOR = Node(state, MEJORNODO, [move])
                SUCESOR.g = MEJORNODO.g + 1  # cost of the path to the successor
                SUCESOR.h = self.heuristic(state)  # heuristic value for the successor
                SUCESOR.key = SUCESOR.g + SUCESOR.h  # set the key as its priority

                self.generated += 1

                # if the successor is in the closed list
                if SUCESOR.state in CERRADOS:
                    VIEJO = CERRADOS[SUCESOR.state]  # get the old version of the successor
                    if SUCESOR.g < VIEJO.g:  # we found a better path
                        VIEJO.parent = MEJORNODO  # update the parent
                        VIEJO.g = SUCESOR.g  # update g
                        VIEJO.key = SUCESOR.key  # update key
                        ABIERTOS.insert(VIEJO)  # update the node in the heap
                        ABIERTOS_DICT[SUCESOR.state] = VIEJO  # update the node in the dictionary

                # if the successor is in the open list
                elif not ABIERTOS.is_empty() and ABIERTOS.contains(SUCESOR):
                    VIEJO = ABIERTOS_DICT.get(SUCESOR.state, None)  # get the old version of the successor
                    if VIEJO:
                        if SUCESOR.g < VIEJO.g:  # we found a better path
                            VIEJO.parent = MEJORNODO  # update the parent
                            VIEJO.g = SUCESOR.g  # update g
                            VIEJO.key = SUCESOR.key  # update key
                            ABIERTOS.insert(VIEJO)  # update the node in the heap
                            ABIERTOS_DICT[SUCESOR.state] = VIEJO  # update the node in the dictionary

```

```
                # if the successor is not in open list and not in closed list
                else:
                    ABIERTOS.insert(SUCESOR)  # insert the successor into the heap
                    ABIERTOS_DICT[SUCESOR.state] = SUCESOR  # insert the successor into the dictionary

        # If no solution was found, return None.
        return None
```

Listing 1: Código A*

```
class BinaryHeap:
    def contains(self, nodo):
        nodos = [elemento for elemento in self.array if isinstance(elemento, node.Node)]

        for elemento in nodos:
            if elemento.state.__str__() == nodo.state.__str__():
                return True
```

Listing 2: Código A*

Lazy A*

```
        def lazysearch(self):
        '''
        Performs the A* search for a solution.
        '''
        print("Computing...")

        ABIERTOS = BinaryHeap()  # Open list represented as a binary heap
        CERRADOS = {}  # Closed list represented as a dictionary
        INITIAL = Node(self.initial_state)  # Initial node
        INITIAL.g = 0  # cost to reach the initial state
        INITIAL.h = self.heuristic(self.initial_state)  # heuristic value for the initial state
        INITIAL.key = INITIAL.g + INITIAL.h  # set the key as its priority
        ABIERTOS.insert(INITIAL)  # insert the initial node into the heap

        self.memory += 1
        self.generated += 1

        while not ABIERTOS.is_empty():
            MEJORNODO = ABIERTOS.extract()  # extract the node with minimum key
            CERRADOS[MEJORNODO.state] = MEJORNODO  # add this node to the closed list

            self.memory = max(self.memory, ABIERTOS.size + len(CERRADOS))

            # if this node is the goal state, reconstruct the path and return it
            if self.heuristic(MEJORNODO.state) == 0:
                path, actions = MEJORNODO.trace()  # Reconstruct path using the trace function
                self.end_time = time.time()
                return actions, path

            self.game.current_state = MEJORNODO.state  # update the game's current state

            SUCESORES = self.game.get_valid_moves()  # generate successors
            self.expansions += 1
            for new_state, move, _ in SUCESORES:  # unpacking the return value of get_valid_moves()
                state = new_state  # convert the new_state to a list of lists
                SUCESOR = Node(state, MEJORNODO, [move])
                SUCESOR.g = MEJORNODO.g + 1  # cost of the path to the successor
                SUCESOR.h = self.heuristic(state)  # heuristic value for the successor
                SUCESOR.key = SUCESOR.g + SUCESOR.h  # set the key as its priority
                self.generated += 1
                ABIERTOS.insert(SUCESOR)  # insert the successor into the heap

        # If no solution was found, return None.
        return None
```

Listing 3: Código Lazy A*

Admisibilidad de Heurísticas

Wagdy Heuristic

La heurística de Wagdy $h_{\text{wagdy}}(n)$ para un nodo n está dada por la suma de 2 por cada par de bolas consecutivas de diferente color en cada tubo. Esto puede ser expresado matemáticamente como:

$$h_{\text{wagdy}}(n) = 2 \cdot \sum_{\text{tube} \in n} \sum_{i=0}^{\text{len(tube)}-2} I[\text{tube}[i] \neq \text{tube}[i+1]]$$

donde $I[.]$ es la función indicadora que toma el valor 1 si la condición en los corchetes es verdadera, y 0 en caso contrario.

Existen casos en los que esta heurística sobreestima el valor de la solución óptima. Por ejemplo, en el caso de que hay dos tubos, uno vacío y uno con una bola roja y una verde la herística daría el valor 2, no obstante, con un solo movimiento se pueden separar ambas bolas. Por lo tanto, no es una heurística admisible.

Repeated Color Heuristic

La heurística de color repetido $h_{\text{repeated color}}(n)$ para un nodo n está dada por la suma de 1 por cada bola que no sea del mismo color que el color más repetido en cada tubo. Esto puede ser expresado matemáticamente como:

$$h_{\text{repeated color}}(n) = \sum_{\text{tube} \in n} \sum_{\text{ball} \in \text{tube}} I[\text{ball} \neq \text{mode(tube)}]$$

donde mode(tube) es la moda (el color más repetido) en el tubo.

En el mejor de los casos, mover una bola a un tubo diferente requeriría al menos un movimiento en el juego real. Entonces, el costo en el caso ideal de alcanzar el estado final desde el nodo n siempre será menor o igual a $h_{\text{repeated color}}(n)$. Por lo tanto, $h_{\text{repeated color}}(n)$ es admisible.

Demostración Monotómicamente creciente

Primero, establecemos la base de la inducción. El primer nodo que se expande es el nodo de inicio n_0, por lo que no hay un nodo previo para comparar y la propiedad se mantiene.

Para el paso inductivo, supongamos que la propiedad se mantiene hasta el nodo n_i, es decir, $f(n_i) \geq f(n_{i-1})$. Necesitamos demostrar que la propiedad se mantiene para el nodo n_{i+1}.

Por la definición del algoritmo A*, sabemos que el nodo n_{i+1} tiene el valor mínimo de f de todos los nodos en la lista abierta en el momento de la expansión, es decir, $f(n_{i+1}) \leq f(n)$ para todo n en la lista abierta.

Supongamos por contradicción que $f(n_{i+1}) < f(n_i)$. Pero el nodo n_i ya se ha expandido y ya no está en la lista abierta, por lo que esto no viola la propiedad de que $f(n_{i+1})$ es el mínimo de todos los nodos en la lista abierta. Sin embargo, por la propiedad de la heurística consistente, sabemos que $f(n_i) \leq f(n_{i+1})$. Esta es una contradicción, por lo que nuestra suposición de que $f(n_{i+1}) < f(n_i)$ debe ser incorrecta.

Por lo tanto, debemos tener que $f(n_{i+1}) \geq f(n_i)$, lo que demuestra que la propiedad se mantiene para el nodo n_{i+1}. Esto completa el paso inductivo y por lo tanto, por inducción, la secuencia de valores $f(n)$ de los estados que se extraen de la lista abierta es monótonamente creciente cuando A* se usa con una heurística consistente.

Demostración Optimalidad Lazy A*

Para demostrar que Lazy A* encuentra soluciones óptimas, primero recordamos que A* garantiza soluciones óptimas siempre que se use una heurística admisible y consistente. La diferencia clave entre Lazy A* y A* es que Lazy A* permite duplicados en la lista abierta.

Una preocupación podría ser que, si el nodo se expande antes de que se encuentre el camino más corto hasta él, podríamos obtener una solución subóptima. Sin embargo, esto no sucede debido a la forma en que funciona Lazy A*.

Considera que tenemos un nodo n en la lista abierta con un valor g y que se encuentra un camino más corto hacia el mismo estado, representado por un nodo n' con valor $g' < g$. En A* estándar, actualizamos el nodo en la lista abierta. En Lazy A*, sin embargo, añadimos n' a la lista abierta.

A pesar de la existencia de duplicados, Lazy A* sigue encontrando soluciones óptimas debido a la propiedad de que siempre se expande el nodo con el menor valor f en la lista abierta. Es importante recordar que $f(n) = g(n) + h(n)$, donde $g(n)$ es el costo del camino hasta el nodo n y $h(n)$ es la heurística que estima el costo desde n hasta la meta.

Dado que $g(n') < g(n)$ y que n' y n representan el mismo estado, $h(n') = h(n)$, entonces $f(n') = g(n') + h(n') < g(n) + h(n) = f(n)$. Por lo tanto, n' será expandido antes que n.

Si existe un camino más corto hacia el estado de n, este será encontrado y expandido antes de que se expanda n. Por lo tanto, cuando se expande un nodo que representa un estado, podemos estar seguros de que hemos encontrado el camino más corto hacia ese estado, al igual que en A*.

Por lo tanto, Lazy A* encuentra soluciones óptimas, siempre que se use una heurística admisible y consistente.

Comparación heurísticas

Al comparar las dos heurísticas se concluye que (al menos con esta implementación) wadgy heuristic es amplimante superior. Utilizando tanto menos tiempo como memoria. Probablemente esto es debido a que es capaz de estimar de mejor manera la verdadera distancia al destino de un estado.

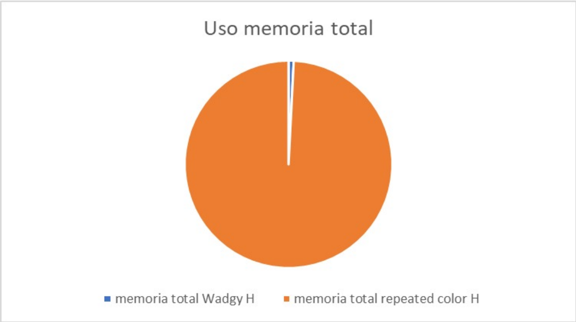

Figure 1: Comparación de uso de memoria con heurísticas.

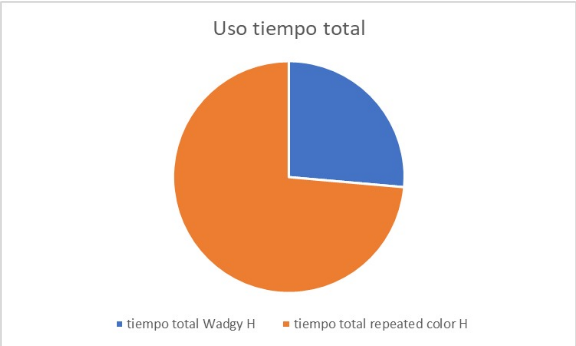

Figure 2: Comparación de tiempo de ejecución con heurísticas.

1 Comparación A* vs Lazy A*

Se observa una implementación pobre de A*, la cuál es extremadamente lenta e ineficiente. Probablemente esto ocurrió debido a la utilización del heap binario como estructura, dado que para saber si un estado ya había sido revisado se requiere una gran cantidad de pasos para saber si el heap contiene el estado. Aquí esa implementación:

```
class BinaryHeap:
    def contains(self, nodo):
            nodos = [elemento for elemento in self.array if isinstance(elemento, node.Node
    )]

            for elemento in nodos:
                if elemento.state.__str__() == nodo.state.__str__():
                    return True
```

Listing 4: Código A*

Se sugiere fuertemente emplear otra estructura (como un diccionario) para poder solucionar esto. No obstante, la obtención del nodo "top" tendrá un fuerte empeoramiento.

En concecuencia a esto, se concluye que es fuertemente recomendado utilizar lazy A*, al menos con esta implementación ya que el uso de memoria es marginalmente mayor y los tiempos son órdenes de magnitud menores.

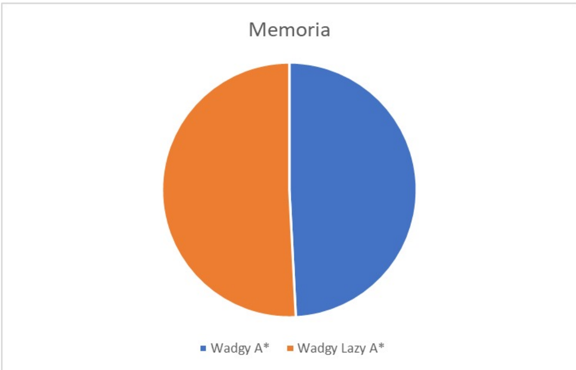

Figure 3: Comparación de uso de memoria entre A* y Lazy A*.

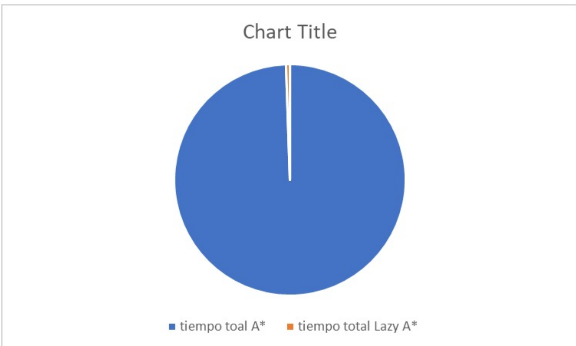

Figure 4: Comparación de tiempo de ejecución entre A* y Lazy A*.

Best First Search y comparación

Para implementar Best First search se implementó el siguiente código:

```python
def greedysearch(self): #  Greedy Best First Search
        '''
        Performs the Best First Search search for a solution.
        '''
        print("Computing...")

        ABIERTOS = BinaryHeap()  # Open list represented as a binary heap
        CERRADOS = {}  # Closed list represented as a dictionary
        INITIAL = Node(self.initial_state)  # Initial node
        INITIAL.g = 0  # cost to reach the initial state
        INITIAL.h = self.heuristic(self.initial_state)  # heuristic value for the initial state
        INITIAL.key = INITIAL.h  # set the key as its priority
        ABIERTOS.insert(INITIAL)  # insert the initial node into the heap

        self.memory += 1
        self.generated += 1

        while not ABIERTOS.is_empty():
            MEJORNODO = ABIERTOS.extract()  # extract the node with minimum key
            CERRADOS[MEJORNODO.state] = MEJORNODO  # add this node to the closed list

            self.memory = max(self.memory, ABIERTOS.size + len(CERRADOS))

            # if this node is the goal state, reconstruct the path and return it
            if self.heuristic(MEJORNODO.state) == 0:
                path, actions = MEJORNODO.trace()  # Reconstruct path using the trace function
                self.end_time = time.time()
                return actions, path

            self.game.current_state = MEJORNODO.state  # update the game's current state

            SUCESORES = self.game.get_valid_moves()  # generate successors
            self.expansions += 1
            for new_state, move, _ in SUCESORES:  # unpacking the return value of get_valid_moves()
                state = new_state  # convert the new_state to a list of lists
                SUCESOR = Node(state, MEJORNODO, [move])
                SUCESOR.g = MEJORNODO.g + 1  # cost of the path to the successor
                SUCESOR.h = self.heuristic(state)  # heuristic value for the successor
                SUCESOR.key = SUCESOR.h  # set the key as its priority
                self.generated += 1
                ABIERTOS.insert(SUCESOR)  # insert the successor into the heap

        # If no solution was found, return None.
        return None
```

Listing 5: Implementación BFS

Se observa una bastante leve mejora tanto de los tiempos de ejecución como del uso de memoria. Sin embargo, se pierde la optimalidad y en ocaciones se llega a caminos más largos que los logrados por A*. A simple vista el trade off no lo vale.

Figure 5: Comparación de tiempo de ejecución entre Lazy A* y greedy search.

Figure 6: Comparación de memoria utilizada entre Lazy A* y greedy search.

Conslusiones:

Se concluye que definir una heurística apropiada es extraordinariamente importante e influye de manera determinante en la cantidad de expansiones, tiempos de ejecucion y utilización en memoria.

También, se propone que la forma de implementar A* no fué la apropiada dada la problemática mencionada en la comparación con Lazy A*. Dado que los tiempos de computo para chequear que el nuevo nodo ya estaba en la open eran excesivos. Se sugiere un mejor uso de los diccionarios como posible solución.

Además se observó una leve mejoría en el uso de recursos de BFS pero se estima que no vale la pérdida de optimidad.

En conclusión, la mejor de las técnicas implementadas es lazysearh() con la heurística wadgy.