PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA

Curso: ICS2613-Inteligencia Artificial
Semestre: 2023-1
Profesor: Jorge Baier

Tarea 2

Fecha de entrega: Lunes 28 de Mayo 2023

Alumnos: Beatriz Errázuriz Camus nro.19638906

Parte 1

Actividad 3: Admisibilidad de heurísticas

Pregunta 1
Una función heurística h se dice admisible, si para todo

$$s : h(s) \leq h^*(s)$$

Por otro lado, una heurística se dice consistente si y solo si:

$$h(s) = 0 \text{ para } s \in G$$
$$h(s) \leq c(s, s') + h(s'), \text{ para todo vecino } s' \text{ de } s$$

A partir del teorema sabemos que si h es consistente por ende también es admisible.
Para esto agregamos el atributo $self.is\_admisible$ a la clase $AStarSolver$ y lo seteamos en $True$. Revisamos que la heurística sea admisible cada vez que creamos o actualizamos el valor de un nodo

```
if child_node.h > parent_node.h + neighbor[2]:
    self.is_admisible = False
```

Este valor lo retorna la función $search()$ por lo que después den finalizada su ejecución, imprimimos el valor de $self.is\_admisible$ para comprobar si la heurística es admisible.

Aplicando esto para ambas heurísticas obtenemos los siguientes resultados:

- **wagdy_heuristic** En este caso obtenemos que la heurística no es admisible. Para ahondar en los motivos mostraré un ejemplo.

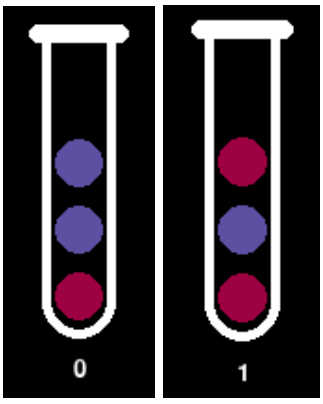

Figura 1: Ejemplo admisibilidad wadgy_heuristic a partir de dos tubos distintos que expresan un estado del juego

Las heurísticas del estado en este caso sería la suma de los pares de colores diferentes para cada tubo, que serían 3, por lo que $h(s') = 6$.
Consideremos el siguiente estado:

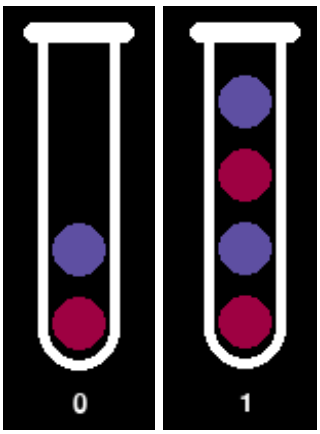

Figura 2: Ejemplo admisibilidad wadgy_heuristic a partir de dos tubos distintos que expresan un estado del juego

Al realizar el movimiento anterior vemos como la heurística del estado aumenta en 2, ya que el tubo 0 se mantuvo igual y el tubo 1 tiene un par más de colores distintos. Esta nueva heurística sería entonces $h(s) = 8$
Por la definición de consistencia mostrada anteriormente deberíamos demostrar que:

$$h(s) \leq c(s, s') + h(s')$$

donde el costo es 1 ya que el costo real de realizar un movimiento. Con esto tenemos que:

$$8 \leq 1 + 6$$
$$8 \leq 7$$

Esto demostraría que la heurística no es admisible ya que está sobre estimando el costo de llegar al estado objetivo.

- **repeated_color_heuristic:** En este caso la heurística es admisible. Esto puede simplemente comprobarse debido a que para cualquier caso la fórmula $h(s) \leq c(s, s') + h(s')$ se cumple. Esto implicaría que no se está sobre estimando el costo de llegar al estado objetivo.

Pregunta 2
Para demostrar que una heurística consistente tiene una secuencias de valores f monótonamente creciente simplemente verificamos que el valor f del padre no sea mayor que el del hijo.

```
if child_node.key < parent_node.key:
    self.f_states = False
```

En el caso de la heurística de **repeated_color_heuristic** (que ya demostramos es admisible) aplicando la fórmula al algoritmo obtemos que $self.f\_states = True$, es decir, que la secuencias de valores f es monótonamente creciente. Si imprimimos los valores f después de extraer el nodo del *open* obtenemos (como ejemplo):

```
F states: [3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5,
                                                                5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
```

Pregunta 3
Lazy A^* también encuentra soluciones óptimas y no es muy díficil de demostrar. La diferencia entre el algoritmo y A^* está en que *Lazy* A^* permite nodos duplicados en vez de actualizar los valores de un nodo si su

camino es mejor que el que estaba registrado antes.
Esto no es un problema para la optimalidad ya que de todas maneras al extraer un nodo del open este será el con el menor valor f por lo que realmente no es necesario actualizar los valores de un nodo con un peor rendimiento.
Un camino correspondiente a ese nodo duplicado con un mayor valor f no será elegido antes que el con un mejor (menor) valor f que si lleva al camino óptimo ya que siempre terminará antes de llegar al objetivo y para un **mismo** nodo duplicado siempre se preferirá el que correspondía al mejor camino (menor f).
Esto si podría ser un problema en términos de memoria y tiempo ya que el open contendrá más nodos que no son necesarios tener (ya que los nodos con un peor f nunca serán la solución al problema) y se expandirán una mayor cantidad de nodos (ya que puede pasar que se expanda nodos no óptimos)

Actividad 4: Comparación de heurísticas

En la siguiente tabla encontraremos los valores (promedio para 5 casos) de nodos expandidos y tiempo de ejecución para el caso del problema resuelto sin heurística y para los casos con las dos heurísticas. Al mismo tiempo se muestra la memoria utilizada en cada caso para comparar el algoritmo A^* *Star* y *Lazy* A^* *Star* :

	No heuristic			**Wadgy heuristic**			**Repeated color heuristic**		
	nodos expandidos	*tiempo de ejecución*	*memoria utilizada*	*nodos expandidos*	*tiempo de ejecución*	*memoria utilizada*	*nodos expandidos*	*tiempo de ejecución*	*memoria utilizada*
map_1	88	0.043235063552856445	48	8	0.0003366470336914062 5	15	36	0.03865504264831543	24
map_2	362278	39.20734429359436	457314	138	0.09641718864440918	791	2988	0.5230882167816162	12069
map_3	Programa tardó más de 2 minutos			480	0.2509589195251465	4399	Programa tardó más de 2 minutos		
map_4	Programa tardó más de 2 minutos			7940	6.252002000808716	136947	Programa tardó más de 2 minutos		
map_5	Programa tardó más de 2 minutos			4363	7.168849229812622	134341	Programa tardó más de 2 minutos		

Cuadro 1: Comparación empírica para algoritmo A^* *Star*

	No heuristic			**Wadgy heuristic**			**Repeated color heuristic**		
	nodos expandidos	*tiempo de ejecución*	*memoria utilizada*	*nodos expandidos*	*tiempo de ejecución*	*memoria utilizada*	*nodos expandidos*	*tiempo de ejecución*	*memoria utilizada*
map_1	1739	0.16568493843078613	5563	12	0.0002956390380859375	26	120	0.07563281059265137	323
map_2	Programa tardó más de 2 minutos			211	0.15310978889465332	2063	110342	28.879741430282593	1397941
map_3	Programa tardó más de 2 minutos			2836	0.7441332340240479	30432	Programa tardó más de 2 minutos		
map_4	Programa tardó más de 2 minutos			161219	56.21582579612732	2411554	Programa tardó más de 2 minutos		
map_5	Programa tardó más de 2 minutos			Programa tardó más de 2 minutos			Programa tardó más de 2 minutos		

Cuadro 2: Comparación empírica para algoritmo *Lazy* A^* *Star*

Analizando las distintas heurísticas, se observa una clara diferencia cuando no se utilizan heurísticas. En este caso, al no haber una estimación del costo hacia el nodo objetivo y simplemente considerar el costo hasta el nodo actual, es posible que el algoritmo priorice caminos que no se dirijan de manera óptima al objetivo. Esto implica que se expandan más nodos de los necesarios y que el algoritmo tarde más tiempo en ejecutarse.

Entre las otras dos heurísticas, puede suceder que Wadgy tarde menos tiempo en ejecutarse y expanda menos nodos. Sin embargo, como se demostró anteriormente, esta heurística no proporciona un camino óptimo debido a que no es admisible. Una posible razón de esto es que al sobreestimar el costo de llegar al estado objetivo, el algoritmo tarda menos en ejecutarse al çortarün mayor número de caminos. Es por esta razón que también necesita expandir menos nodos.

En términos generales, se observa que el algoritmo A^{Star} es notablemente superior a *Lazy* A^{Star}. Esto se debe a que este último puede explorar nodos duplicados con un valor f mayor que lidera a un camino no óptimo, lo que impide que se retorne la solución. Esto hace que el algoritmo tome más tiempo en ejecutarse y utilice más memoria al generar nodos duplicados innecesarios. Además, *Lazy* A^*Star utiliza más memoria en general, ya que no actualiza los valores de los nodos con mejores valores, sino que crea nodos nuevos.

Todas estas observaciones coinciden con la explicación dada anteriormente sobre la diferencia en los valores.

Pregunta 5
La diferencia entre A^* y *Greedy Best − First Search* es que este último solo basa su valor f en base a la heurística sin contar el costo hasta el nodo actual. Por lo que el algoritmo se vería de la siguiente manera.

```
def greedysearch(self):
```

```
        '''
            Performs the Greedy Best First Search for a solution.
        '''
        self.start_time = time.time()
        self.generated = {}

        # We create the initial node and add it to the open list
        initial_node = Node(self.initial_state)
        initial_node.h = self.heuristic(initial_node.state)
        initial_node.key = initial_node.h
        self.generated[repr(initial_node.state)] = initial_node
        self.open.insert(initial_node)

        # While there are still nodes to expand
        while not self.open.is_empty():
            # We get the node with the lowest f value
            parent_node = self.open.extract()
            self.game.current_state = parent_node.state

            # If the node is a goal node, we return the solution
            if parent_node.state.is_final():
                self.end_time = time.time()
                return parent_node.trace(), self.expansions, self.end_time - self.start_time

            # We expand the node
            self.expansions += 1
            neighbors = self.game.get_valid_moves()
            for neighbor in neighbors:
                in_open = True
                child_node = self.generated.get(repr(neighbor[0]))
                # If the node didn't exist it means it has never been in the open list
                in_open = False
                # If the node doesnt exist, we create it
                if child_node is None:
                    child_node = Node(neighbor[0])
                    self.generated[repr(child_node.state)] = child_node
                    child_node.h = self.heuristic(child_node.state)
                    child_node.parent = parent_node
                    child_node.action = neighbor[1]
                    child_node.key = child_node.h
                    # If node is not in open, we insert it
                    if not in_open:
                        self.open.insert(child_node)

        self.end_time = time.time()
        return None, self.expansions, self.end_time - self.start_time
```

Realizamos las mismas estadísticas que antes para compararlo con los resultados de A^*

	No heuristic			**Wadgy heuristic**			**Repeated color heuristic**		
	nodos expandidos	*tiempo de ejecución*	*memoria utilizada*	*nodos expandidos*	*tiempo de ejecución*	*memoria utilizada*	*nodos expandidos*	*tiempo de ejecución*	*memoria utilizada*
map_1	24	0.00294876098632812 5	19	6	0.0004394054412841797	12	7	0.0004353523254394531	10
map_2	188	0.2192411422729492 2	987	19	0.07439851760864258	131	35	0.1018834114074707	218
map_3	Programa tardó más de 2 minutos			49	0.28862953186035156	786	53	0.19287919998168945	767
map_4	Programa tardó más de 2 minutos			85	0.2543957233428955	1770	91	0.3385002613067627	2508
map_5	Programa tardó más de 2 minutos			107	0.3182852268218994	3960	304	1.1972403526306152	13359

Cuadro 3: Comparación empírica para algoritmo *Greedy Best – First Search*

Notamos principalmente una considerable reducción en los valores en comparación con el algoritmo $A*$. Esto se debe a que el algoritmo prioriza la heurística y no considera el costo hasta el nodo actual (g). Como resultado, el algoritmo logra llegar al objetivo rápidamente, ya que siempre busca acercarse a él lo más rápido posible. Sin embargo, esto implica sacrificar el camino óptimo para llegar al estado objetivo.