Reflexión sobre el trabajo realizado – Taller #1

Nelson Felipe Celis – 202320636
María Alejandra Carrillo – 202321854
Isabella Bracho – 202423192
Alicia Robinson – 202321278

1. Análisis punto #1

```python
def depthFirstSearch(problem: SearchProblem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:

    print("Start:", problem.getStartState())
    print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
    print("Start's successors:", problem.getSuccessors(problem.getStartState()))
    """
    #Version inicial de DFS

    #Creamos visitados, la pila y una estructrua para almacenar los padres
    visited = set()
    stack = utils.Stack()
    parents = {}

    initial_state = problem.getStartState()
    stack.push(initial_state)

    while not stack.isEmpty():
        current = stack.pop()

        # Verificar objetivo ANTES de marcar visitado para no perder estados
        if problem.isGoalState(current):
            return _reconstruct_path(parents, initial_state, current)

        if current not in visited:
            visited.add(current)
            for node, action, cost in problem.getSuccessors(current):
                if node not in visited:
                    parents[node] = (current, action)
                    stack.push(node)

    return []
```

a. Complejidad en el espacio
Con respecto a la complejidad en espacio, encontramos en el código implementado tiene tres estructuras que ocupan la mayor cantidad de memoria: visited, parents y stack. Todas tienen una complejidad de O(V), ya que la pila puede crecer hasta V nodos, límite que nunca sobrepasa por la verificación de la condición `if node not in visited`.

b. Complejidad en tiempo
En el peor de los casos, este algoritmo recurre a visitar cada uno de los nodos una sola vez gracias a el set donde se guardan llamado `visited`. Después se examina cada arista después de generar los sucesores de cada uno de estos por medio de `getSuccessors`. Por esta razón, la complejidad de este algoritmo sería O(V+E), donde V tiene que ver con el número de vértices y E el número de aristas.

c. ¿El algoritmo es completo?
El algoritmo si es completo ya que garantiza que siempre que haya una solución, la encuentra. Usualmente, el DFS estándar no es completo, ya que en casos donde los grafos son infinitos o con ciclos no siempre encuentra una solución, sin embargo, esta

implementación de DFS que realizamos como grupo tiene ciertos cambios que permiten que este algoritmo sea completo. Entre estos cambios, encontramos que nunca revisita nodos, por lo que nos aseguramos de que no hay ciclos infinitos. Asimismo, el grafo de layouts es finito y sin ciclos cuando marcamos los nodos ya visitados. Por último, si existe un camino del nodo inicial al objetivo, este algoritmo siempre lo encuentra y si no lo hay, lo devuelve correctamente (`[]`). Todos estos factores hacen que este algoritmo sea completo.

d. ¿El algoritmo encuentra la solución óptima siempre o en cuáles condiciones?
DFS es un algoritmo que realiza la búsqueda por profundidad, este factor es el que se prioriza, por lo que no garantiza encontrar el camino más corto o con menor valor ya que ni el costo ni la distancia son priorizados. Este algoritmo puede encontrar antes un camino más largo que el más corto posible, ya que el orden de la exploración depende del orden de los sucesores y no realiza ninguna verificación de costo.
Para encontrar el camino óptimo, este algoritmo tendría que estar en un mundo donde todos los caminos tuvieran el mismo costo o que existiera una única manera de llegar desde el nodo inicial hasta el objetivo.

2. Análisis punto #2

```python
def breadthFirstSearch(problem: SearchProblem):
    """
    Search the shallowest nodes in the search tree first.
    """

    #Version inicial BFS

    visited = set()
    queue = utils.Queue()
    parents = {}

    initial_state = problem.getStartState()

    # Verificación temprana si el inicio ya es meta
    if problem.isGoalState(initial_state):
        return []

    visited.add(initial_state)
    queue.push(initial_state)

    while not queue.isEmpty():
        current = queue.pop()

        for node, action, cost in problem.getSuccessors(current):
            if node not in visited:
                parents[node] = (current, action)

                # Early exit: verificar al encolar evita expandir un nivel extra
                if problem.isGoalState(node):
                    return _reconstruct_path(parents, initial_state, node)

                visited.add(node)
                queue.push(node)

    return []
```

a. Complejidad en el espacio
Al igual que DFS, BFS tiene tres estructuras que ocupan la mayor cantidad de memoria: visited, parents y queue. La complejidad total es de O(V), ya que la cola puede crecer hasta

V nodos, límite que nunca sobrepasa por la verificación de la condición `if node not in visited`.

b. Complejidad temporal
En el peor de los casos, este algoritmo recurre a visitar cada uno de los nodos una sola vez gracias a el set donde se guardan llamado `visited`. Después se examina cada arista después de generar los sucesores de cada uno de estos por medio de `getSuccessors`. Por esta razón, la complejidad de este algoritmo sería O(V+E), donde V tiene que ver con el número de vértices y E el número de aristas.

c. ¿El algoritmo es completo?
BFS es completo siempre y cuando el grafo con el cual se implemente sea un grafo finito. Los nodos son marcados al ser encolados, lo que evita encolar el mismo nodo múltiples veces, cada nodo se visita una única vez, se encarga de los ciclos en caso tal de que hayan y la solución es encontrada en caso de que haya alguna, sino, retorna [].

d. ¿El algoritmo encuentra la solución óptima siempre o en cuáles condiciones?
BFS, a diferencia de DFS, siempre garantiza encontrar el camino con el menor número de pasos desde el estado inicial hasta el objetivo. Esto ocurre con este algoritmo ya que el grafo se explora por niveles, donde el primer nodo objetivo se encuentra en el nivel más bajo posible, y ningún camino más corto puede existir en niveles anteriores.
Sin embargo, con respecto al costo, como DFS, en este algoritmo el costo es de igual forma ignorado, y como en nuestro problema cada tipo de terreno tiene un costo diferente, BFS no encuentra el camino óptimo.

3. Análisis punto #3

```python
def uniformCostSearch(problem: SearchProblem):
    """
    Search the node of least total cost first.
    """
    #Version inicial UCS

    visited = set()
    queue = utils.PriorityQueue()
    parents = {}

    initial_state = problem.getStartState()
    costs = {initial_state: 0}
    queue.push(initial_state, 0)

    while not queue.isEmpty():
        current = queue.pop()

        # Verificar goal al desencolar garantiza optimalidad en UCS
        if problem.isGoalState(current):
            return _reconstruct_path(parents, initial_state, current)

        if current not in visited:
            visited.add(current)
            for node, action, cost in problem.getSuccessors(current):
                new_cost = costs[current] + cost
                if node not in visited and (node not in costs or costs[node] > new_cost):
                    costs[node] = new_cost
                    parents[node] = (current, action)
                    queue.update(node, new_cost)

    return []
```

a. Complejidad en el espacio

En este código, hay cuatro estructuras para guardar la información que ocupan la memoria: visited, parents, costs y queue. La totalidad de las complejidades de estas estructuras es O(V), incluso si la cola de prioridad puede tener varias entradas del mismo nodo antes de que este sea marcado como visitado. Cada una de estas estructuras permite verificar el costo de los caminos, lo cual es de suma importancia para este algoritmo.

b. Complejidad en el tiempo
Como dijimos anteriormente, cada nodo puede tener múltiples entradas en la cola de prioridad antes de ser marcado como visitado, donde cada operación push cuesta O(log V). En el peor de los casos, se procesan todas las aristas, por lo que la complejidad temporal de este algoritmo es de O(E log V), donde V son los nodos y E las aristas.

c. ¿El algoritmo es completo?
UCS es completo siempre y cuando los grafos no contengan valores negativos, lo cual no es un problema para este ejercicio, ya que todos los costos son mayores o iguales a 1. Lo que hace este algoritmo, es que comienza explorando los nodos en orden creciente de costo acumulado, para que cuando logre alcanzar todos los nodos posiblemente alcanzables, encontrar el mejor camino del nodo inicial al objetivo. La estructura visited, sirve para evitar ciclos infinitos.

d. ¿El algoritmo encuentra la solución óptima siempre o en cuáles condiciones?
Este algoritmo, a diferencia de los dos anteriores, garantiza encontrar el camino de menos costo acumulado, ya que tiene en consideración el costo de cada tipo de terreno en cada variable. La estructura que permite que se expanda siempre el nodo de menor costo es la cola de prioridad, por lo que cuando un nodo es extraído de la cola, su costo siempre es el mínimo. Ningún camino futuro puede tener menor costo y el objetivo se encuentra con el camino de menor costo posible, no únicamente con la menor cantidad de pasos como con BFS y DFS.

4. Análisis punto #4

```python
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
    """
    Search the node that has the lowest combined cost and heuristic first.
    """
    #Version inicial A*
    visited = set()
    queue = utils.PriorityQueue()
    parents = {}

    initial_state = problem.getStartState()
    costs = {initial_state: 0}
    queue.push(initial_state, heuristic(initial_state, problem))

    while not queue.isEmpty():
        current = queue.pop()

        # Verificar goal al desencolar garantiza optimalidad (heurística admisible)
        if problem.isGoalState(current):
            return _reconstruct_path(parents, initial_state, current)

        if current not in visited:
            visited.add(current)
            for node, action, cost in problem.getSuccessors(current):
                new_cost = costs[current] + cost
                if node not in visited and (node not in costs or costs[node] > new_cost):
                    costs[node] = new_cost
                    parents[node] = (current, action)
                    queue.update(node, new_cost + heuristic(node, problem))

    return []
```

Reflexión sobre el trabajo realizado – Taller #1

a. Complejidad en el tiempo
   Con el algoritmo de A* implementado, la heurística determina su rendimiento. La heurística perfecta en este caso sería que se expanden únicamente los nodos del camino óptimo, pero en cualquiera de los casos, sigue siendo más eficiente que UCS a pesar de expandir más nodos que el óptimo. En el peor de los casos, la complejidad es de O(b^d), donde b es el *branching factor* (es decir, cuantos sucesores tiene cada nodo en promedio), y d la profundidad de la solución óptima.

b. Complejidad en el espacio
   En el peor de los casos con el algoritmo implementado, el cual es A*, todos los nodos generados deben mantenerse en memoria. Las estructuras para almacenar datos que encontramos en este caso son visited, parents, costs y queue. El rendimiento depende de la calidad de la heurística, pero en el peor de los casos, la complejidad es de O(b^d), donde b es el *branching factor* (es decir, cuantos sucesores tiene cada nodo en promedio), y d la profundidad de la solución óptima.

c. ¿El algoritmo es completo?
   Si, este algoritmo es completo en cualquier caso donde los espacios sean finitos y los costos no sean negativos, lo cual es el caso del problema que nos encontramos resolviendo. Si existe un camino objetivo desde el nodo inicial hasta el objetivo, A* lo encontrará, y una buena heurística permite que no se descarten posibles caminos prontamente. Además, la estructura visited evita que hayan ciclos finitos. Tanto la distancia Manhattan como con la Euclidiana se garantiza completitud.

d. ¿El algoritmo encuentra la solución óptima siempre o en cuáles condiciones?
   Si, A* garantiza encontrar el camino de menor costo únicamente si la heurística es adecuada, el cual es el caso para nosotros. En este caso, tanto Manhattan como Euclidiana son admisibles, por lo que A* es óptimo con ambas.

e. ¿Las heurísticas son consistentes?
   Combinación de costo real y estimación heurística: g(n) = h(n) + f(n), donde g(n) es el costo real acumulado teniendo en cuenta el costo de pasar por cada terreno, h(n) siendo la estimación heurística del costo restante desde n hasta el objetivo y f(n) el costo estimado total del camino óptimo que pasa por n.

   Una heurística es consistente si el costo de ir de un nodo a otro mediante la acción más la suma de la heurística del siguiente nodo es mayor a la heurística del nodo actual. Esto implica que la heurística debe mejorar a medida que nos acercamos al objetivo.

   En un lado, en el caso de la distancia Manhattan, esta cumple la desigualdad triangular estricta. Cada paso reduce la distancia Manhattan en 1 si nos acercamos. Por lo tanto, Manhattan es consistente. Por otro lado, en el caso de la distancia Euclidiana, esta cumple con la desigualdad triangular de la geometría euclidiana. En cualquier triángulo, la longitud de un lado es menor o igual a la suma de los otros dos lados. Por lo tanto, Euclidiana es consistente.

5. Análisis punto #5
   a. Complejidad en el tiempo
      El estado en este problema no es solo la posición del robot, sino también cuáles sobrevivientes faltan por rescatar. Si el mapa tiene N celdas y hay k sobrevivientes, el total de estados posibles es N × 2^k, porque para cada posición el robot puede haber rescatado

cualquier combinación de sobrevivientes. En el peor caso A* tiene que revisar todos esos estados, y cada vez que mete o saca un estado de la cola de prioridad eso cuesta un logaritmo, así que la complejidad queda O(N × 2^k × log(N × 2^k)). Aparte, la primera vez que se calcula la distancia real entre dos puntos se corre un UCS interno, pero como eso se guarda en caché, solo se hace una vez por cada par de sobrevivientes.

b. Complejidad en el espacio
A* tiene que guardar en memoria todos los estados que ya generó pero todavía no revisó, más el diccionario de padres para reconstruir el camino al final. En el peor caso eso también es O(N × 2^k). Por eso el layout grande con 10 sobrevivientes se traba: 2^10 son 1024 combinaciones posibles multiplicadas por todas las celdas del mapa, lo que se vuelve gigantesco muy rápido.

c. ¿El algoritmo es completo?
Sí, siempre encuentra la solución si existe. El mapa es finito, los costos son positivos y el algoritmo nunca revisita el mismo estado gracias al conjunto visited, así que no puede quedarse dando vueltas infinitamente. Si hay un camino para rescatar a todos los sobrevivientes, A* lo va a encontrar.

d. ¿El algoritmo encuentra la solución óptima siempre o en cuáles condiciones?
Sí, siempre y cuando la heurística no sobreestime el costo real, que es justo lo que hace survivorHeuristic. La heurística suma la distancia real al sobreviviente más cercano más el costo del MST entre los sobrevivientes restantes, que es la estimación más optimista posible del trabajo que queda. Como nunca exagera el costo, A* garantiza encontrar el camino más barato para rescatar a todos.

e. ¿Las heurísticas son consistentes?
Sí. Una heurística es consistente si lo que estima desde un nodo nunca es mayor que el costo de dar un paso más lo que estima desde el nodo siguiente. Acá cuando el robot se mueve un paso, la distancia real al sobreviviente más cercano cambia máximo en el costo de ese paso, y el MST de los sobrevivientes restantes nunca puede aumentar porque son los mismos o menos. Entonces la estimación nunca "salta" más de lo que costó moverse, y la condición se cumple. Esto también significa que A* nunca necesita volver a revisar un estado que ya expandió.

6. Uso de la IA generativa:

Para este taller, utilizamos herramientas de IA generativa como Claude y ChatGPT como apoyo complementario en nuestro proceso de aprendizaje, pero siempre manteniendo el control sobre el desarrollo. Primero implementamos versiones iniciales de cada algoritmo (DFS, BFS, UCS y A*) de forma autónoma como grupo, basándonos en lo visto en clase y discutiendo entre nosotros la lógica de cada uno. Luego usamos la IA para verificar la correctitud de nuestro código, identificar casos que no habíamos considerado y profundizar en conceptos teóricos que nos generaban dudas, como por qué las heurísticas son consistentes; cada concepto y modificación sugerida la discutíamos como grupo y la probábamos en diferentes layouts antes de adoptarla.

Reflexión sobre el trabajo realizado – Taller #1

```python
def depthFirstSearch(problem: SearchProblem):
    """

    Version inicial de DFS implementación autonoma:


    #Creamos visitados, la pila y una estructrua para almacenar los padres
    visited=set()
    stack=utils.Stack()
    parents={}

    initial_state=problem.getStartState()
    stack.push(initial_state)
    #Se itera mientras no este vacia
    while not stack.isEmpty():
        current=stack.pop()
        #Si se encuentra el estado objetivo se reconstruye el camino, se revierte y se devuelve
        if problem.isGoalState(current):
            path=[]
            state=current
            while state != initial_state:
                parent, action = parents[state]
                path.append(action)
                state=parent
            path.reverse()
            return path
    #Si no ha sido visitado, se marca y se agregan los sucesores a la pila y se guarda en la estructura para almacenar la ruta padre-hijo
        if current not in visited:
            visited.add(current)
            for node,action,cost in problem.getSuccessors(current):
                if node not in visited:
                    parents[node]= (current,action)
                    stack.push(node)
    return []

    """
    #Version Optimizada con IA

    #Creamos visitados, la pila y una estructrua para almacenar los padres
    visited = set()
    stack = utils.Stack()
    parents = {}

    initial_state = problem.getStartState()
    stack.push(initial_state)

    while not stack.isEmpty():
        current = stack.pop()

        # Verificar objetivo ANTES de marcar visitado para no perder estados
        if problem.isGoalState(current):
            return _reconstruct_path(parents, initial_state, current)

        if current not in visited:
            visited.add(current)
            for node, action, cost in problem.getSuccessors(current):
                if node not in visited:
                    parents[node] = (current, action)
                    stack.push(node)

    return []
```

```python
def breadthFirstSearch(problem: SearchProblem):
    """
    Version inicial de BFS implementación autonoma:

    visited=set()
    queue=utils.Queue()
    parents={}

    initial_state=problem.getStartState()
    visited.add(initial_state)
    queue.push(initial_state)

    #Se itera mientras no este vacia
    while not queue.isEmpty():
        current=queue.pop()
        #Si se encuentra el estado objetivo se reconstruye el camino, se revierte y se devuelve
        if problem.isGoalState(current):
            path=[]
            state=current
            while state != initial_state:
                parent, action = parents[state]
                path.append(action)
                state=parent
            path.reverse()
            return path
    #Se marca y se agregan los sucesores a la fila y se guarda en la estructura para almacenar la ruta padre-hijo
        for node,action,cost in problem.getSuccessors(current):
            if node not in visited:
                visited.add(node)
                parents[node]= (current,action)
                queue.push(node)
    return []

    """

    #Version Optimizada con IA

    visited = set()
    queue = utils.Queue()
    parents = {}

    initial_state = problem.getStartState()

    # Verificación temprana si el inicio ya es meta
    if problem.isGoalState(initial_state):
        return []

    visited.add(initial_state)
    queue.push(initial_state)

    while not queue.isEmpty():
        current = queue.pop()

        for node, action, cost in problem.getSuccessors(current):
            if node not in visited:
                parents[node] = (current, action)

                # Early exit: verificar al encolar evita expandir un nivel extra
                if problem.isGoalState(node):
                    return _reconstruct_path(parents, initial_state, node)

                visited.add(node)
                queue.push(node)

    return []
```

```python
def uniformCostSearch(problem: SearchProblem):
    """

    Version inicial de UCS implementación autonoma:

    visited=set()
    queue=utils.PriorityQueue()
    parents={}

    initial_state=problem.getStartState()
    costs={initial_state:0} #Utilizamos una estructura de costos para saber cual es el menor costo que he encontrado
    queue.push(initial_state,0)

    while not queue.isEmpty():

        current= queue.pop()
        #Si se encuentra el estado objetivo despues de sacar de la cola de prioridad, se reconstruye el camino y se retorna
        if problem.isGoalState(current):
            path=[]
            state=current
            while state != initial_state:
                parent, action = parents[state]
                path.append(action)
                state=parent
            path.reverse()
            return path
        #Si el actual no esta visitado, lo marco y miro sus sucesores
        if current not in visited:
            visited.add(current)
            for node,action,cost in problem.getSuccessors(current):
                new_cost=costs[current] + cost
                if node not in visited and (node not in costs or costs[node]> new_cost): #Solo modifico los costos de los sucesores si no han sido visitados o si hay un costo menor al que ya habia encontrado
                    costs[node]=new_cost
                    parents[node]= (current,action)
                    queue.update(node,new_cost)
    return []


    """
    #Version inicial UCS

    visited = set()
    queue = utils.PriorityQueue()
    parents = {}

    initial_state = problem.getStartState()
    costs = {initial_state: 0}
    queue.push(initial_state, 0)

    while not queue.isEmpty():
        current = queue.pop()

        # Verificar goal al desencolar garantiza optimalidad en UCS
        if problem.isGoalState(current):
            return _reconstruct_path(parents, initial_state, current)

        if current not in visited:
            visited.add(current)
            for node, action, cost in problem.getSuccessors(current):
                new_cost = costs[current] + cost
                if node not in visited and (node not in costs or costs[node] > new_cost):
                    costs[node] = new_cost
                    parents[node] = (current, action)
                    queue.update(node, new_cost)

    return []
```

Reflexión sobre el trabajo realizado – Taller #1

```python
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
    """
    Version inicial de A* implementación autonoma:

     visited=set()
    queue=utils.PriorityQueue()
    parents={}

    initial_state=problem.getStartState()
    costs={initial_state:0}
    queue.push(initial_state,heuristic(initial_state,problem))

    while not queue.isEmpty():
        current= queue.pop()
        if problem.isGoalState(current): #Evaluar una vez sacamos de pila
            path=[]
            state=current
            while state != initial_state:
                parent, action = parents[state]
                path.append(action)
                state=parent
            path.reverse()
            return path

        if current not in visited:
            visited.add(current)
            for node,action,cost in problem.getSuccessors(current):
                new_cost=costs[current] + cost
                if node not in visited and (node not in costs or costs[node]> new_cost):
                    costs[node]=new_cost
                    parents[node]= (current,action)
                    queue.update(node,new_cost + heuristic(node,problem))
    return []

    """
    #Version optimizada con IA
    visited = set()
    queue = utils.PriorityQueue()
    parents = {}

    initial_state = problem.getStartState()
    costs = {initial_state: 0}
    queue.push(initial_state, heuristic(initial_state, problem))

    while not queue.isEmpty():
        current = queue.pop()

        # Verificar goal al desencolar garantiza optimalidad (heurística admisible)
        if problem.isGoalState(current):
            return _reconstruct_path(parents, initial_state, current)

        if current not in visited:
            visited.add(current)
            for node, action, cost in problem.getSuccessors(current):
                new_cost = costs[current] + cost
                if node not in visited and (node not in costs or costs[node] > new_cost):
                    costs[node] = new_cost
                    parents[node] = (current, action)
                    queue.update(node, new_cost + heuristic(node, problem))

    return []
```