UADE

INTELIGENCIA ARTIFICIAL

TRABAJO PRÁCTICO OBLIGATORIO GRUPAL N°2
“Machine Learning (ML) - Buseuda”

Turno: Noche		
Integrantes:		
1.	Di Fresco, Lucas	LU:
2.	Gugliemone, Lucas	LU: 1110022
3.	Lacuesta, Gastón Axel	LU: 1117695
4.	Rondan, Ignacio	LU: 1068437
5.	Orozo, Patricio	LU: 1058382
6.	Vera, Samira	LU: 1101684
Profesor: Parkinson, Christian		
Fecha: 28/11/2022	Cuatrimestre: 2C 2022	

Depth-first:

Para esta función de búsqueda inicializamos el primer nodo a partir del estado inicial que obtenemos del problema.

Confirmamos que el estado inicial no es el estado objetivo y de no ser así, creamos una pila de la librería **util,** y un set para almacenar los nodos ya visitados.

Mientras la pila no se encuentre vacía, visitaremos el último nodo agregado a la pila, si este no es el estado objetivo continuaremos consultando el primer nodo sucesor de este hasta encontrar el objetivo. Una vez encontrado se devolverá una lista de acciones que se fue completando a partir de cada nodo sucesor para poder llegar desde el estado inicial hasta el objetivo.

```
def depthFirstSearch(problem):
    startingNode = problem.getStartState()
    if problem.isGoalState(startingNode):
        return []

    frontier = util.Stack()
    frontier.push((startingNode,[]))
    explored = set()

    while not frontier.isEmpty():
        currentNode, actions = frontier.pop()
        if currentNode not in explored:
            explored.add(currentNode)
            if problem.isGoalState(currentNode):
                return actions
            for nextNode, action, cost in problem.getSuccessors(currentNode):
                nextAction = actions + [action]
                frontier.push((nextNode,nextAction))
```

Breadth-first:

En este caso en lugar de crear una pila, utilizaremos una cola, por lo que primero se visitan todos los nodos sucesores del nodo evaluado antes de los nodos hijos de estos. De esta manera cambia el tipo de búsqueda a bfs.

```
def breadthFirstSearch(problem):
    node = {'state':problem.getStartState(), 'cost': 0}
    if problem.isGoalState(node["state"]):
        return []

    frontier = util.Queue()

    frontier.push(node)
    explored = set()

    while not frontier.isEmpty():

        node = frontier.pop()
        explored.add(node['state'])
        successors = problem.getSuccessors(node['state'])

        for successor in successors:
            child = { 'state': successor[0], 'action':successor[1], 'cost': successor[2],
            if child['state'] not in explored:
                if problem.isGoalState(child['state']):
                    actions = []
                    node = child
                    while 'parent' in node:
                        actions.append(node['action'])
                        node = node['parent']
                    actions.reverse()
                    return actions
                frontier.push(child)
```

Uniform cost:

En uniform cost, en lugar de solo almacenar la frontera en una cola, como hacíamos en bfs, la cola será una cola de prioridades, por lo que siempre se visitará el vecino más cercano con menor costo primero. Para evitar loops infinitos, se revisa siempre que el nodo evaluado no se haya visitado previamente.

```
def uniformCostSearch(problem):
    startingNode = problem.getStartState()
    if problem.isGoalState(startingNode):
        return []

    frontier = util.PriorityQueue()
    frontier.push((startingNode,[],0),0)
    explored = set()

    while not frontier.isEmpty():
        currentNode, actions, oldCost = frontier.pop()
        if currentNode not in explored:
            explored.add(currentNode)
            if problem.isGoalState(currentNode):
                return actions
            for nexNode, action, cost in problem.getSuccessors(currentNode):
                nextAction = actions + [action]
                priority = oldCost + cost
                frontier.push((nexNode, nextAction, priority), priority)
```

A Star:

En esta función, para la selección del próximo nodo se utiliza de nuevo una cola de prioridades, donde la prioridades es la suma del costo estimado (heurístico) hasta el estado objetivo y el costo de alcanzar ese nodo. La selección de nodos será dada por la suma de estos costos, generando el camino menos costoso desde el estado inicial al objetivo.

```
def aStarSearch(problem, heuristic=nullHeuristic):
    startingNode = problem.getStartState()
    if problem.isGoalState(startingNode):
        return []

    explored = set()

    frontier = util.PriorityQueue()
    frontier.push((startingNode, [], 0),0)

    while not frontier.isEmpty():
        currentNode, actions, oldCost = frontier.pop()
        if currentNode not in explored:
            explored.add(currentNode)
            if problem.isGoalState(currentNode):
                return actions
            for nextNode, action, cost in problem.getSuccessors(currentNode):
                nextAction = actions + [action]
                newCostToNode = oldCost + cost
                heuristicCost = newCostToNode + heuristic(nextNode, problem)
                frontier.push((nextNode, nextAction, newCostToNode), heuristicCost)
```