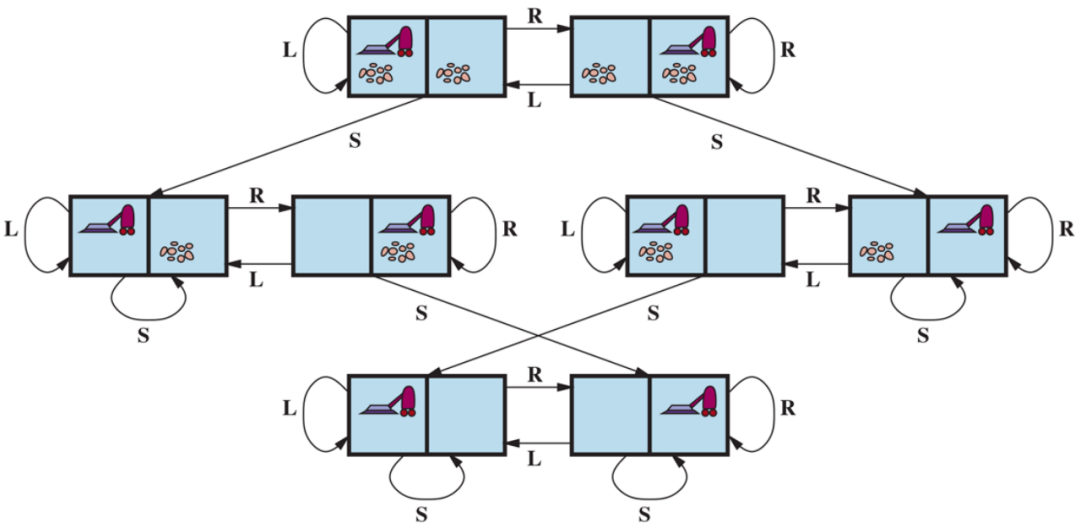
UNIVERSIDAD NACIONAL DE COLOMBIA

MINI PROYECTO 1

13 / 02 / 2023
Asignatura: Inteligencia artificial

Estudiantes	Correo
Andrés Camilo Peralta Fragozo	aperaltaf@unal.edu.co
Daniel Felipe Zuniga Hurtado	dfzunigah@unal.edu.co
Sergio Andres Guzman Carrascal	seguzmanc@unal.edu.co

PROGRAMA DE AGENTE DE SOLUCIÓN DE PROBLEMAS SENCILLOS



Descripción general del agente

Tenemos un agente capaz de realizar dos acciones, *aspira*r si detecta mugre y *moverse* si ya está limpio.

Este agente en particular tiene la capacidad de poder operar en un número indefinido de habitaciones siempre que esta sea de dimensiones $m \, x \, m$.

En cuanto al rango de movimiento, en un número de habitaciones mayor a dos, (2) el agente es capaz de desplazarse en cuatro direcciones, izquierda, derecha, arriba y abajo según corresponda.

Para implementar el algoritmo de búsqueda tomamos prestado las clases Nodo, Frotera, Pila y Cola:

```
class Nodo():
    def __init__(self, estado, padre, accion):
        self.estado = estado
        self.padre = padre
        self.accion = accion

class Frontera():
    def __init__(self):
        self.frontera =[]

    def empty(self):
        return (len(self.frontera) == 0)

    def add(self, nodo):
        self.frontera.append(nodo)

    def eliminar(self):
        # LIFO o FIFO
        pass

    def contiene_estado(self, estado):
        return any(nodo.estado == estado for nodo in
self.frontera)

class Pila(Frontera):
    def eliminar(self):
        # Termina la busqueda si la frontera esta vacia
        if self.empty():
            raise Exception("Frontera vacia")
        else:
            # Guardamos el ultimo item en la lista
            # (el cual es el nodo recientemente añadido)
            nodo = self.frontera[-1]
            # Guardamos todos los items excepto el
            # ultimo (eliminamos)
            self.frontera = self.frontera[:-1]
            return nodo

class Cola(Frontera):
    def eliminar(self):
        # Termina la busqueda si la frontera esta vacia
        if self.empty():
            raise Exception("Frontera vacia")
        else:
            # Guardamos el primer item en la lista
            # (el cual es el nodo añadido de primero)
            nodo = self.frontera[0]
            # Guardamos todos los items excepto el
            # primero (eliminamos)
            self.frontera = self.frontera[1:]
```

```
            return nodo
```

Clase principal:

```
class SimpleProblemSolvingAgentProgram:
    """
    [Figure 3.1]
    Abstract framework for a problem-solving agent.
    """

    def __init__(self, initial_state=None):
        """State is an abstract representation of the state
        of the world, and seq is the list of actions required
        to get to a particular state from the initial
state(root)."""
        self.state = initial_state
        self.seq = []

    def __call__(self, percept):
        """[Figure 3.1] Formulate a goal and problem, then
        search for a sequence of actions to solve it."""
        self.state = self.update_state(self.state, percept)
        if not self.seq:
            goal = self.formulate_goal(self.state)
            problem = self.formulate_problem(self.state, goal)
            self.seq = self.search(problem)
            if not self.seq:
                return None
        return self.seq
        # return self.seq.pop(0)


    def update_state(self, state, percept):
        raise NotImplementedError

    def formulate_goal(self, state):
        raise NotImplementedError

    def formulate_problem(self, state, goal):
        raise NotImplementedError

    def search(self, problem):
        raise NotImplementedError
```

Este es un programa de un agente de resolución de problemas simple en Python. Se utiliza una estructura abstracta para representar un agente que resuelve problemas y se define una clase llamada *SimpleProblemSolvingAgentProgram* que implementa esta estructura.

La clase tiene cuatro métodos que deben ser implementados para que el agente pueda funcionar correctamente:

- *update_state:* Este método actualiza el estado del agente basado en la percepción recibida.

- *formulate_goal:* Este método permite al agente formular una meta basada en su estado actual.

- *formulate_problem*: Este método permite al agente formular un problema a partir de su estado actual y su meta.

- *search:* Este método permite al agente buscar una secuencia de acciones para resolver el problema formulado.

Estos métodos se utilizan en el método *__call__* para formular un objetivo, formular un problema y buscar una secuencia de acciones para resolver el problema.

Considerando lo anterior se define la subclase vacuumAgent que define al agente que interactúa con el entorno y hereda e implementa los métodos faltantes de la clase base SimpleProblemSolvingAgentProgram.

```
state1 = [(1, 0),habitaciones]

class vacuumAgent(SimpleProblemSolvingAgentProgram):
        def update_state(self, state, percept):
            return percept

        def formulate_goal(self, state):
            goal = objetivo
            return goal

        def formulate_problem(self, state, goal):
            problem = state, goal
            return problem

        def search(self, problem):

            self.num_explorados = 0
            state, goal = problem
            # Inicializamos la frontera para empezar en la posicion inicial
            start = Nodo(estado=state, padre=None, accion=None)
            frontera = Pila()
            frontera.add(start)

            # Inicializamos en conjunto explorado vacio
            self.explorado = set()
            # Manetenmos el bucle hasta que encontremos la solución
            while True:
                # Si nada queda en la frontera, entonces no hay más camino
                if frontera.empty():

                    raise Exception("No hay Solución")

                # Escogemos un nodo de la frontera
                nodo = frontera.eliminar()
                self.num_explorados +=1
                if nodo.accion == "suck":
                  n,k = nodo.estado[0]
                  nodo.estado[1][n][k] = "Clean"
```

```
                # Si el nodo es el objetivo, entonces tenemos una solución
                if nodo.estado[1] == goal:
                    acciones = []
                    cel = []
                    # Rastreamos los nodos padre hasta la solución
(objetivo hasta estado inicial)
                    while nodo.padre is not None:
                        acciones.append(nodo.accion)
                        cel.append(nodo.estado)
                        nodo = nodo.padre
                    acciones.reverse()
                    cel.reverse()
                    self.solucion = (acciones, cel)
                    return acciones

                # Marcamos el nodo como exploado
                n,k = nodo.estado[0]
                if not nodo.estado[1][n][k] == "Dirty":
                  self.explorado.add(nodo.estado[0])
                # Agregamos vecinos a la frontera
                for accion, estado in self.acciones_valida(nodo.estado):
                    if not frontera.contiene_estado(estado[0]) and
estado[0] not in self.explorado:
                        hijo = Nodo(estado = estado, padre = nodo,
accion=accion)
                        frontera.add(hijo)

        def acciones_valida(self, state):
          fila, col = state[0]
          self.altura = len(habitaciones)
          self.ancho = len(habitaciones[0])
          candidatos = [
              ("up", (fila -1, col)),
              ("down", (fila +1, col)),
              ("left", (fila, col -1)),
              ("right", (fila, col +1)),
              ("suck", (fila, col)) ]

          resultados = []
          for accion, (f,c) in candidatos:
              if 0<=f < self.altura and 0<=c < self.ancho:
                if state[1][f][c] == "Dirty":
                  resultados.append((accion,[(f,c), state[1]]))
                if not state[1][f][c] == "Dirty" and not accion == "suck":
                  resultados.append((accion,[(f,c), state[1]]))

          return resultados
```

Ahora cada método de la clase *vacuumAgent* es una **sobreescritura** de un método de la clase padre que es necesario para que el programa funcione correctamente. Las actualizaciones de tales métodos se describe como sigue:

- *update_state:* este método actualiza el estado actual del agente con los percepciones recibidas del entorno. En este caso, simplemente devuelve el percepción sin realizar ninguna operación adicional.

- *formulate_goal*: este método forma una meta a partir del estado actual del agente. En este caso, la meta es una lista con dos estados posibles: state7 y state8.

- *formulate_problem:* este método forma un problema a partir del estado actual y la meta. En este caso, el problema es simplemente el estado actual del agente.

- *search*: este método implementa la búsqueda de la solución al problema. Aquí se hacen comparaciones con diferentes estados y se devuelve una secuencia de acciones que se deben ejecutar para llegar a un estado final objetivo. Cada comparación y secuencia es diferente dependiendo del estado actual del agente.

Respecto a las capacidades de movimiento del agente, por cada habitación se le debe formular un objetivo, teniendo en cuenta que el tamaño de las habitaciones debe ser cuadrado, por ejemplo:

El agente se encuentra en un entorno de 4 habitaciones:

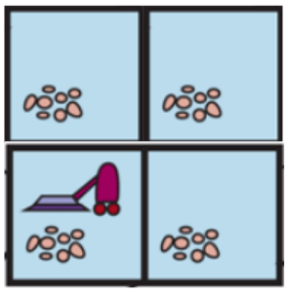
```
habitaciones = [["Dirty", "Dirty"],
                ["Dirty", "Dirty"]]

objetivo = [["Clean", "Clean"],
            ["Clean", "Clean"]]
```

considerando que,

```
state1 = [(1, 0),habitaciones]
```

corresponde a la posición inicial del agente, donde `state1[1]` seria la esquina inferior izquierda, así,

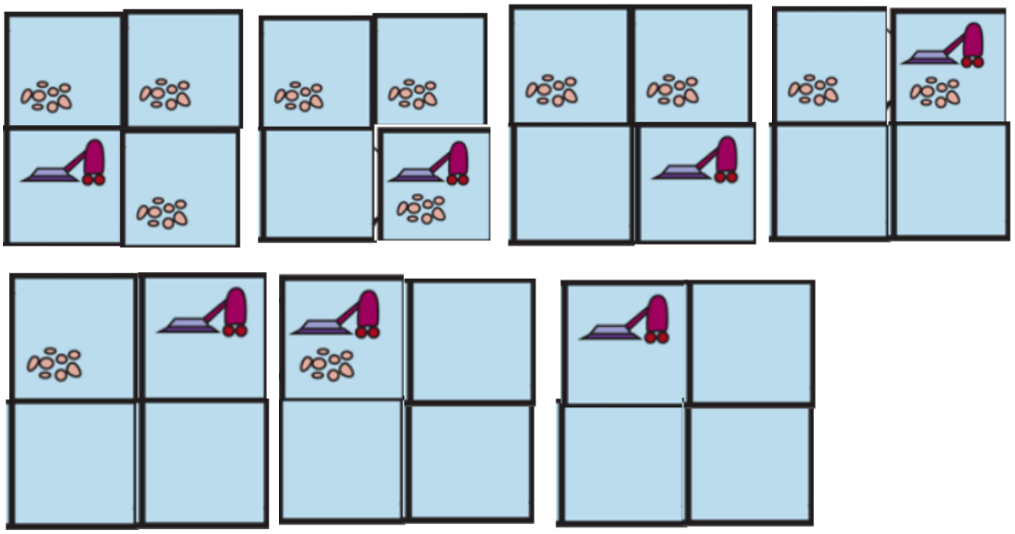


Al crear la instancia de la clase vacuumAgent la secuencia de acciones retornada por el agente sería como sigue.

```
state1 = [(1, 0),habitaciones]
a = vacuumAgent(state1)
print(a(state1))


['suck', 'right', 'suck', 'up', 'suck', 'left', 'suck']
```

Visto de manera grafica seria



En caso de que la cantidad de habitaciones sea mayor, la secuencia retornada será igual de grande.