Inteligencia Artificial

1er Cuatrimestre 2024

Trabajo Práctico:

Búsquedas en Python

Facultad de Ciencias de la Administración

Fecha límite de entrega: Viernes 26/04/2024 a las 23:59.

Condiciones de entrega: el trabajo práctico deberá ser realizado en forma **individual**. Se deberá subir en la sección del Campus Virtual correspondiente un **único archivo** comprimido con formato `zip, rar, tar.gz` u otro. El mismo contendrá los **archivos `.py` separados por cada punto/consigna**. En cada archivo `.py` se debe presentar el código solución a la consigna y algunas líneas adicionales de código que sirvan para testear la solución presentada (No es necesario presentar interfaz gráfica). Además, pueden incluir un `.pdf` que presente las respuestas, suposiciones y aclaraciones pertinentes de cada punto.

1. Analice el siguiente código **Python** teniendo en cuenta las búsquedas de primero en profundidad y primero en anchura vistas en clases:

```
def search(start,stop):
    """
    :param start: estado inicial
    :param stop: test objetivo
    trajectory() nos devuelve el recorrido al nodo
    """
    frontera = deque()
    explored = set()
    if(stop(start)):
        return trajectory(start)
    frontera.append(start)
    while(frontera):
        nodo = frontera.popleft()
        explored.add(nodo)
        for child in nodo.expand():
            if stop(child):
                return trajectory(child)
            elif not child in explored:
                frontera.append(child)
    return None
```

   a) Con el uso de comentarios explicar que realiza el código en cada línea.

   b) Identificar a que tipo de búsqueda corresponde el código.

   c) Especificar las modificaciones necesarias al código para implementar el otro tipo de búsqueda nombrado en el enunciado.

2. A partir del siguiente código *Python* donde se presenta una solución para el problema de las n-reinas.

```
"""Puzzle n-reinas."""
class NQueens:
    """Genere todas las soluciones válidas para el puzzle de n reinas"""
    def __init__(self, size):
        # Almacene el tamaño del puzzle (problema) y la cantidad de soluciones válidas
        self.size = size
        self.solutions = 0
        self.solve()

    def solve(self):
        """Resuelve el puzzle de las n reinas e imprime el número de soluciones"""
        positions = [-1] * self.size
        self.put_queen(positions, 0)
        print("Found", self.solutions, "solutions.")

    def put_queen(self, positions, target_row):
        """
```

```
            Intente colocar una reina en target_row marcando todos los N casos posibles.
            Si se encuentra un lugar válido, la función se llama a sí misma tratando de
                colocar
            una reina en la siguiente fila hasta que todas las N reinas se colocan en el
                tablero NxN.
            """
            # Caso base (corte): todas las N filas están ocupadas
            if target_row == self.size:
                self.show_full_board(positions)
                # self.show_short_board(positions)
                self.solutions += 1
            else:
                # Para todas las posiciones de N columnas, intente colocar una reina
                for column in range(self.size):
                    # Rechazar todas las posiciones inválidas
                    if self.check_place(positions, target_row, column):
                        positions[target_row] = column
                        self.put_queen(positions, target_row + 1)


        def check_place(self, positions, ocuppied_rows, column):
            """
            Compruebe si una posición determinada está siendo atacada
            por alguna de las reinas colocadas anteriormente
            (verifique las posiciones de la columna y la diagonal)
            """
            for i in range(ocuppied_rows):
                if positions[i] == column or \
                    positions[i] - i == column - ocuppied_rows or \
                    positions[i] + i == column + ocuppied_rows:

                    return False
            return True

        def show_full_board(self, positions):
            """Mostrar el tablero completo de NxN"""
            for row in range(self.size):
                line = ""
                for column in range(self.size):
                    if positions[row] == column:
                        line += "Q␣"
                    else:
                        line += ".␣"
                print(line)
            print("\n")

        def show_short_board(self, positions):
            """
            Muestre las posiciones de las reinas en el tablero en forma comprimida,
            cada número representa la posición de la columna ocupada en la fila
                correspondiente.
            """
            line = ""
            for i in range(self.size):
                line += str(positions[i]) + "␣"
            print(line)

    def main():
        """Inicializa y resuelve el rompecabezas de n reinas"""
        NQueens(4)

    if __name__ == "__main__":
        # execute only if run as a script
        main()
```

Adecuar la implementación anterior para:

a) n-torres

3. **Puzzle**. Problema que consiste en alcanzar una ubicación objetivo, pudiendo mover únicamente las piezas adyacentes a un lugar determinado. **"z"** para este caso particular. En el campus encuentran disponible el .py para modificar.

a) Indique que tipo de algoritmo de los dados en clase está implementado en el código (Resolvente de problemas, búsqueda local, problema de satisfacción de restricciones, búsqueda adversaria o sus subtipos).

b) Completar con el código correspondiente en los lugares indicados

```
#
# Completar con el código correspondiente
#
```

c) Explicar que realiza el código en los lugares indicados

```
#
# Explicar el objetivo de (parte del código)
#
```

```
from simpleai.search import astar, SearchProblem

# Clase que contiene métodos para resolver el puzzle.
class PuzzleSolver(SearchProblem):
    #
    # Explicar el objetivo de "def actions(self, cur_state)"
    #
    def actions(self, cur_state):
        rows = string_to_list(cur_state)
        row_empty, col_empty = get_location(rows, 'e')

        actions = []
        if row_empty > 0:
            actions.append(rows[row_empty - 1][col_empty])
        if row_empty < 2:
            actions.append(rows[row_empty + 1][col_empty])
        if col_empty > 0:
            #
            # Completar con el código correspondiente
            #
        if col_empty < 2:
            #
            # Completar con el código correspondiente
            #
        return actions

    # Devuelve el estado resultante después de mover una pieza al espacio vacío
    def result(self, state, action):
        rows = string_to_list(state)
        row_empty, col_empty = get_location(rows, 'e')
        row_new, col_new = get_location(rows, action)
```

```
            rows[row_empty][col_empty], rows[row_new][col_new] = \
                    rows[row_new][col_new], rows[row_empty][col_empty]

            return list_to_string(rows)

        # Retorna verdadero si el estado es estado objetivo
        def is_goal(self, state):
            return # Completar con el código correspondiente

        #
        # Explicar el objetivo de "def heuristic(self, state)"
        # Indicar que heurística se utiliza
        #
        def heuristic(self, state):
            rows = string_to_list(state)

            distance = 0

            for number in '12345678z':
                row_new, col_new = get_location(rows, number)
                row_new_goal, col_new_goal = goal_positions[number]

                distance += abs(row_new - row_new_goal) + abs(col_new - col_new_goal)

            return distance

    # Convierte lista en string
    def list_to_string(input_list):
        return '\n'.join(['-'.join(x) for x in input_list])

    # Convierte string en lista
    def string_to_list(input_string):
        return [x.split('-') for x in input_string.split('\n')]

    # Encuentra la ubicación 2D del elemento de entrada
    def get_location(rows, input_element):
        for i, row in enumerate(rows):
            for j, item in enumerate(row):
                if item == input_element:
                    return i, j

    # Final result that we want to achieve
    GOAL = '''1-2-3
    4-5-6
    7-8-z'''

    # Starting point
    INITIAL =
    ##
    ## Completar con el código correspondiente
    ##

    #Crea un caché para la posición de la meta de cada pieza.
    goal_positions = {}
    rows_goal = string_to_list(GOAL)
    for number in '12345678z':
        goal_positions[number] = get_location(rows_goal, number)

    #
    # Explicar el objetivo de "result"
    #
    result = astar(PuzzleSolver(INITIAL))

    # Muestra en pantalla los resultados
    for i, (action, state) in enumerate(result.path()):
```

```
print()
if action == None:
    print('Configuración inicial')
elif i == len(result.path()) - 1:
    print('Después de mover', action, 'al espacio vacío. Objetivo alcanzado!')
else:
    print('Después de mover', action, 'al espacio vacío')

print(state)
```

4. **Coloreo de mapas.** El problema consistente en colorear un mapa con determinados colores de forma que no haya vecinos que tengan el mismo color.
   Para este práctico les pedimos que asignen los colores rojo, verde, azul y amarillo a cada país de américa del sur, de forma tal que se cumpla la restricción de que no existan vecinos que tengan el mismo color.

   a) ¿Qué método o técnica se podría utilizar para determinar la menor cantidad de colores necesarios para resolver el problema? Explicar.
   b) Resolver utilizando la libreria ***simpleai***. `https://simpleai.readthedocs.io/en/latest/constraint_satisfaction_problems.html`. Junto al práctico encuentran un ejemplo.

5. Misioneros y Caníbales Tres misioneros y tres caníbales quieren cruzar un río. Solo hay una canoa que puede ser usada por una o dos personas, ya sean misioneros o caníbales. Hay que tener cuidado en que en ningún momento el numero de caníbales supere al de misioneros en ninguna de las dos orillas, o se los comerán. En el campus encontrarán el archivo **mc.py** que van a utilizar para resolver el ejercicio.

   1. Completar con el código correspondiente en los lugares indicados

```
#
# Completar con el código correspondiente
#
```

   2. Explicar que realiza el código en los lugares indicados

```
        #
        # Explicar el objetivo de (parte del código)
        #


class Estado(object):
        def __init__(self, estado, canoa):
                self.estado = estado
                self.canoa = canoa

                if not self.es_valido():
                        raise ValueError("estado no valido")

        def es_valido(self):
                #
                # Explicar el objetivo de "es_valido"
                #

                for gente in self.estado:
                        for persona in gente:
                                if persona > 3 or persona < 0:
                                        return False

                for mis, can in self.estado:
                        if mis and can > mis:
                                return False

                return True

        def viaja(self, gente):
                #
                # Explicar el objetivo de "viaja"
                #

                estado = copy.deepcopy(self.estado)
                canoa = self.canoa

                estado[canoa][0] -= gente[0]
                estado[canoa][1] -= gente[1]
                canoa = 0 if canoa else 1
                estado[canoa][0] += gente[0]
                estado[canoa][1] += gente[1]

                return Estado(estado, canoa)

        def __str__(self):
                """Muestra el estado como una representacion en texto."""
                return "M: %d C: %d | %s\\___/ %s | M: %d C: %d" % (
                            self.estado[0][0], self.estado[0][1],
                            '~' * 8 * self.canoa, '~' * (8 - 8 * self.canoa),
                            self.estado[1][0], self.estado[1][1]
                            )

        def __eq__(self, other):
                return self.estado == other.estado and self.canoa == other.canoa

        def __ne__(self, other):
                return not self.__eq__(other)

def main():
        # donde empezamos
        #
        # Completar con el código correspondiente
        #
        inicio =
        # a donde queremos llegar
```

```
        #
        # Completar con el código correspondiente
        #
        final =

        # los viaje posibles ( legales )
        #
        # Completar con el código correspondiente
        #
        viajes =

        # los viajes que probamos desde cada estado
        viajes_posibles = list(viajes)

        # guardamos el recorrido y las opciones que no hemos usado
        # para poder 'volver atras' si hay problemas (backtracking)
        recorrido = []
        viajes_restantes = []

        while inicio != final and viajes_posibles:
                while viajes_posibles:
                        # probamos un viaje cualquiera
                        viaje = viajes_posibles.pop()

                        try:
                                nuevo = inicio.viaja(viaje)

                                # si no hemos estado nunca alla
                                if nuevo not in recorrido:
                                        # guarda el estado y las opciones restantes
                                        recorrido.append(inicio)
                                        viajes_restantes.append(viajes_posibles)

                                        # continuamos desde la nueva posicion
                                        inicio = nuevo
                                        viajes_posibles = list(viajes)
                        except ValueError:
                                # no es valido, probamos el siguiente
                                pass

                # si no hemos encontrado nada, backtracking
                if not viajes_posibles and recorrido:
                        inicio = recorrido.pop()
                        viajes_posibles = viajes_restantes.pop()

        if inicio == final:
                print ("Tenemos un resultado!")
                for estado in recorrido:
                        print (estado)
                print (inicio)
        else:
                # no va a pasar ;)
                print ("No hemos encontrado un resultado :(")

if __name__ == "__main__":
        main()
```

6. Resolución de problema criptoaritmético En este ejercicio, se trabajará en la resolución del problema criptoaritmético "SEND + MORE = MONEY". En el campus encontrarán el archivo **cripto.py** que van a utilizar para resolver el ejercicio.

   - Completar el código faltante en `evaluar_solución`, en los lugares indicados con `completar código`. Esta función evalúa la validez de una solución propuesta para el problema criptoaritmético.
   - Identificar a qué tipo de búsqueda corresponde el algoritmo implementado. Colocar la respuesta

como comentario en las primeras líneas de código del algoritmo.

- Analizar y describir qué realiza la función `generar`.
- ¿Qué problemas se presentan con este algoritmo de búsqueda? Modifiquelo para solucionar el problema.

Referencias

[1] S. Russell, P. Norvig., *Artificial Intelligence: A Modern Approach.* 3rd Edition. Cap13. y Cap14. 2010. Prentice Hall.

[2] D. Poole, A. Mackworth., *Artificial Intelligence. Foundations of Computational Agents.* Cap6. 2010. Cambridge University Press.

[3] G. F. Luger, *Artificial intelligence : structures and strategies for complex problem solving.* 6th Edition. Cap9. Secc 9.3. 2009. Pearson Education

[4] *https://wiki.python.org/moin/BeginnersGuide*