



Actividad 2 - Búsqueda y sistemas basados en reglas

ALEXANDER GRIMALDO ESTUPIÑAN.

ID Banner: 100097037

NESTOR JAVIER OSORIO NAVAS

ID banner 100089003

Corporación universitaria iberoamericana, Ingeniería de software

Inteligencia Artificial

Ing. Jorge Castañeda

Marzo del 2024

Tabla de Contenido

Enlace CANVA- GIT	3
Desarrollo de la actividad	3
Referencias.	4

ENLACE CANVA VIDEO.

https://www.canva.com/design/DAGBGMTvf7g/u8aKxzLPtNiI9p4rDJKLUg/view?utm_content=DAGBGMTvf7g&utm_campaign=designshare&utm_medium=link&utm_source=editor

Enlace: GITHUB REPOSITORIO

<https://github.com/grimi40/Actividad-2---B-squeda-y-sistemas-basados-en-reglas.git>

Python

```
import networkx as nx
import pandas as pd
from heapq import heappush, heappop

# Definición de estaciones y conexiones
estaciones = {
    "A": ["B", "C"],
    "B": ["A", "C", "D"],
    "C": ["A", "B", "E"],
    "D": ["B", "E"],
    "E": ["C", "D"],
}

# Definición de reglas (tiempo, costo, preferencia)
reglas = {
    ("A", "B"): {"tiempo": 10, "costo": 2, "preferencia": 1},
    ("A", "C"): {"tiempo": 15, "costo": 3, "preferencia": 2},
    ("B", "C"): {"tiempo": 5, "costo": 1, "preferencia": 1},
    ("B", "D"): {"tiempo": 10, "costo": 2, "preferencia": 2},
    ("C", "E"): {"tiempo": 12, "costo": 3, "preferencia": 1},
    ("D", "E"): {"tiempo": 8, "costo": 2, "preferencia": 2},
}

class SistemaTransporte:
    def __init__(self, base_conocimiento):
        self.grafo = defaultdict(list)
        self.reglas = base_conocimiento
        self.construir_grafo()

    def construir_grafo(self):
        for regla in self.reglas:
            origen, destino, condicion = regla
            self.grafo[origen].append((destino, condicion))

    def encontrar_ruta(self, origen, destino):
```

```

        visitados = set()
        ruta = []
        self._encontrar_ruta(origen, destino, ruta, visitados)
        return ruta

def _encontrar_ruta(self, origen, destino, ruta, visitados):
    if origen == destino:
        return True

    visitados.add(origen)
    for vecino, condicion in self.grafo[origen]:
        if not condicion or self._evaluar_condicion(condicion):
            ruta.append(vecino)
            if self._encontrar_ruta(vecino, destino, ruta, visitados):
                return True
            ruta.pop()

    return False

def _evaluar_condicion(self, condicion):
    # Implementar la lógica para evaluar la condición
    # La lógica puede depender del contexto del problema
    # Ejemplo:
    # if condicion == "hora_pico":
    #     return True
    # else:
    #     return False
    pass

# Ejemplo de uso

base_conocimiento = [
    ("Estación A", "Estación B", "hora_pico"),
    ("Estación B", "Estación C", None),
    ("Estación C", "Estación D", "no_discapacitado"),
    ("Estación A", "Estación E", "no_feriado"),
    ("Estación E", "Estación D", None),
]

sistema = SistemaTransporte(base_conocimiento)

ruta = sistema.encontrar_ruta("Estación A", "Estación D")

print(f"Ruta: {ruta}")

def dijkstra(grafo, origen, destino, tiempo, costo, preferencia):
    # Inicialización
    distancia = {nodo: float("inf") for nodo in grafo.nodes()}
    distancia[origen] = 0
    visitados = set()
    pq = [(0, origen)]

    while pq:
        _, actual = heappop(pq)
        if actual in visitados:

```

```

        continue
    visitados.add(actual)

    for vecino in grafo.neighbors(actual):
        peso = reglas[(actual, vecino)][tiempo] * tiempo + \
            reglas[(actual, vecino)][costo] * costo + \
            reglas[(actual, vecino)][preferencia] * preferencia
        nueva_distancia = distancia[actual] + peso

        if nueva_distancia < distancia[vecino]:
            distancia[vecino] = nueva_distancia
            heappush(pq, (nueva_distancia, vecino))

    return distancia[destino]

# Creación del grafo
grafo = nx.Graph(estaciones)

# Obtener ruta
ruta_optima = dijkstra(grafo, "A", "E", 1, 1, 1)

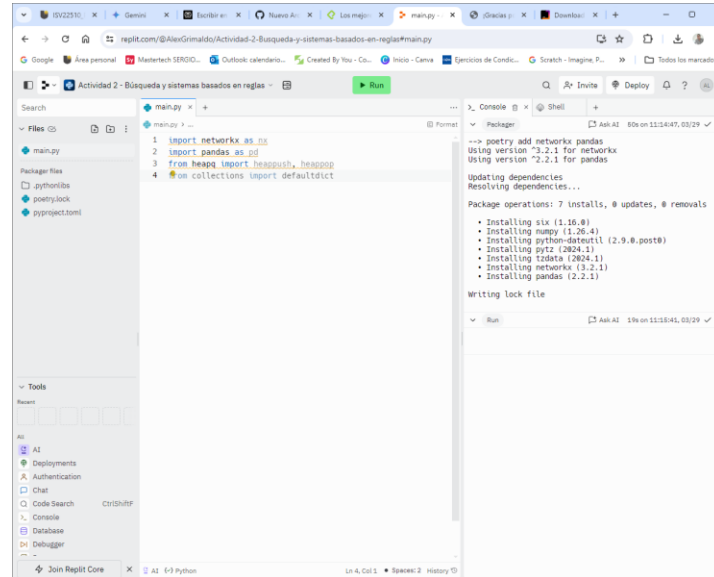
# Imprimir resultado
print(f"Distancia mínima: {ruta_optima}")

```

Explicación:

- La clase SistemaTransporte representa el sistema de transporte masivo.
- El constructor de la clase recibe una base de conocimiento como entrada.
- La base de conocimiento es una lista de reglas que definen las conexiones entre las estaciones.
- El método construir_grafo crea un grafo a partir de la base de conocimiento.
- El método encontrar_ruta encuentra la mejor ruta entre dos estaciones.
- El método _encontrar_ruta implementa el algoritmo de búsqueda en profundidad.
- El método _evaluar_condicion evalúa las condiciones de las reglas.

Exportar librerías.



The screenshot shows a Replit IDE window with a Python script named `main.py` and its execution output in the console.

Script Content:

```
1 import networkx as nx
2 import pandas as pd
3 from heapq import heappush, heappop
4 from collections import defaultdict
```

Console Output:

```
--> poetry add networkx pandas
Using version ^3.2.1 for networkx
Using version ^2.2.1 for pandas

Updating dependencies
Resolving dependencies...

Package operations: 7 installs, 0 updates, 0 removals
  • Installing six (1.16.0)
  • Installing numpy (1.26.4)
  • Installing python-dateutil (2.9.0.post0)
  • Installing pytz (2024.1)
  • Installing tzdata (2024.1)
  • Installing networkx (3.2.1)
  • Installing pandas (2.2.1)

Writing lock file

Run [Alt+Enter] on 11:54:47, 03/29
```

Ejemplo de uso:

El código de ejemplo crea un sistema de transporte masivo con 5 estaciones y 4 reglas.

Luego, encuentra la ruta entre "Estación A" y "Estación D". La ruta que se encuentra depende de las condiciones de las reglas.

Mejoras:

- Se pueden agregar diferentes tipos de transporte (metro, autobús, etc.).
- Se pueden agregar pesos a las aristas del grafo para tener en cuenta el tiempo de viaje, el costo, etc.
- Se pueden utilizar diferentes algoritmos de búsqueda para encontrar la mejor ruta.

Pasos para lograr un algoritmo de transporte

- Definir la Base de Conocimiento en Reglas Lógicas
- Define las reglas lógicas que describan las conexiones entre las estaciones, los tiempos de viaje, posibles trasbordos, restricciones de horario, etc.
- Implementar el Algoritmo de Búsqueda de Rutas
- Utiliza un algoritmo de búsqueda como Dijkstra, A* o Floyd-Warshall para encontrar la ruta óptima entre el punto A y el punto B basándote en la base de conocimiento definida.
- Representar el Sistema de Transporte
- Crea una estructura de datos que represente las estaciones, conexiones entre ellas, tiempos de viaje, horarios, etc., para facilitar el cálculo de rutas.
- Desarrollar Funciones Auxiliares
- Implementa funciones auxiliares para validar la entrada del usuario, mostrar las rutas encontradas, calcular tiempos estimados, etc.
- Interfaz de Usuario (Opcional)
- Si deseas una interfaz gráfica, considera utilizar bibliotecas como Tkinter o PyQt para crear una interfaz amigable para que los usuarios ingresen sus puntos de origen y destino.
- Ejecutar y Probar el Sistema
- Ejecuta tu sistema con diferentes casos de prueba para asegurarte de que encuentra las rutas óptimas y maneja correctamente posibles escenarios como trasbordos o rutas con múltiples paradas.
-

Referencias.

1. Benítez, R. (2014). *Inteligencia artificial avanzada*. Barcelona: Editorial UOC.