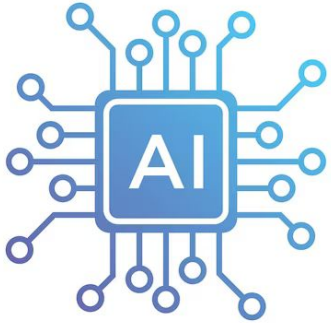
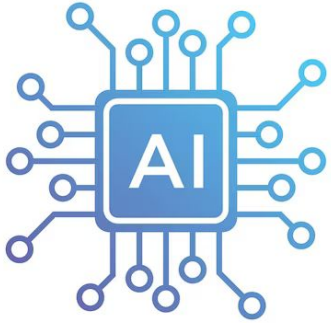


Inteligencia Artificial



Profesor: Ing. Jeremías Baez Carballo
Alumno: Juan Cruz Favaro

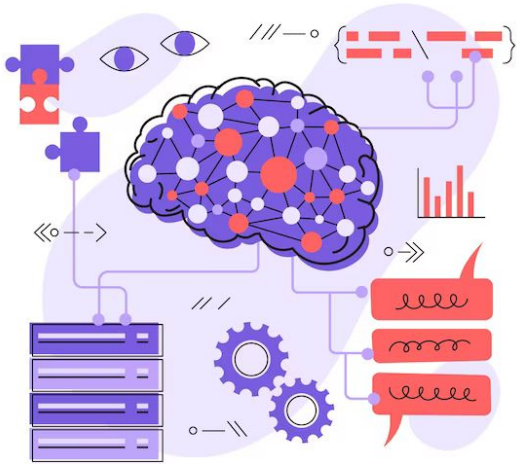
Grados de Separación

- Encontrar el camino más corto entre dos actores/actrices.
- 2 estados (persona1, persona2)
- Acciones: películas, nos llevan de un actor/actriz a otro/a

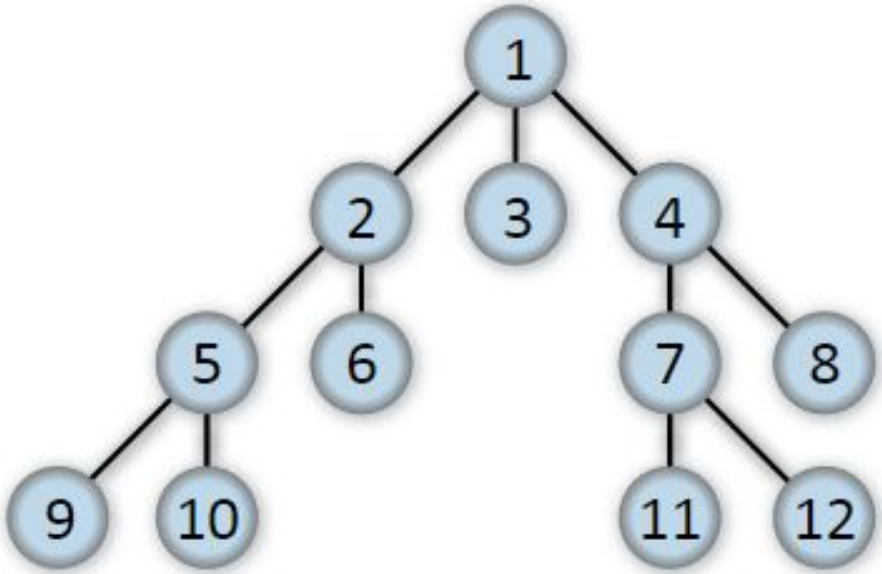
Algoritmo - Búsqueda en Amplitud **BFS**

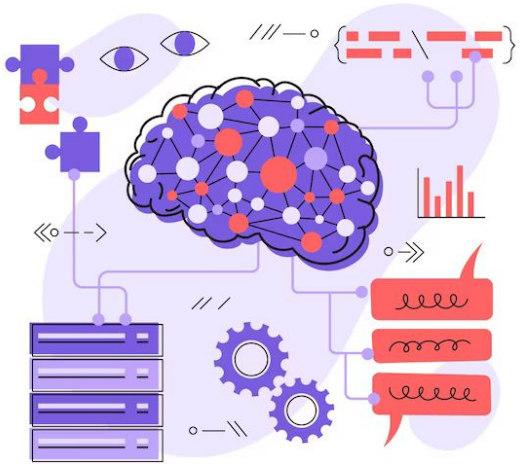
- Exploración nivelada

- Búsqueda sistemática



Algoritmo - Búsqueda en Amplitud **BFS**



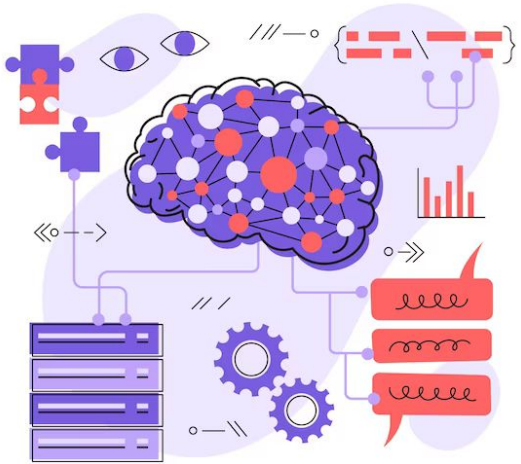


Algoritmo - Búsqueda en Amplitud **BFS**

```
# Inicializamos la frontera con el nodo fuente
start = Node(state=source, parent=None, action=None)
frontier = QueueFrontier()
frontier.add(start)

visitedNodes = set()


while not frontier.empty():
    node = frontier.remove()

    if node.state == target:
        path = []
        while node.parent is not None:
            path.append((node.action, node.state))
            node = node.parent
        path.reverse()
        return path

    visitedNodes.add(node.state)

    neighbors = neighbors_for_person(node.state)

    for action,state in neighbors:
        if state not in visitedNodes:
            child = Node(state=state, parent=node, action=action)
            frontier.add(child)


return None
```

Algoritmo A*

- Grafos ponderados
- Ruta más eficiente en términos de costos (mapas)
- Cola de prioridades
  - Nodo prometedor en términos de llegar al objetivo de manera más eficiente.
  - Ordenación eficiente a la hora de seleccionar el nodo con mayor prioridad.
- Heurística
  - Función estimación del costo para llegar al objetivo



Algoritmo A*

```python
class PriorityQueueFrontier():
    def __init__(self):
        self.frontier = []

    def add(self, node, priority):
        self.frontier.append((node, priority))
        self.frontier.sort(key=lambda x: x[1])

    def contains_state(self, state):
        return any(node.state == state for node, _ in self.frontier)

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

    def remove(self):
        if self.empty():
            raise Exception("empty frontier")
        else:
            node, _ = self.frontier[0]
            self.frontier = self.frontier[1:]
            return node
```

Algoritmo A*

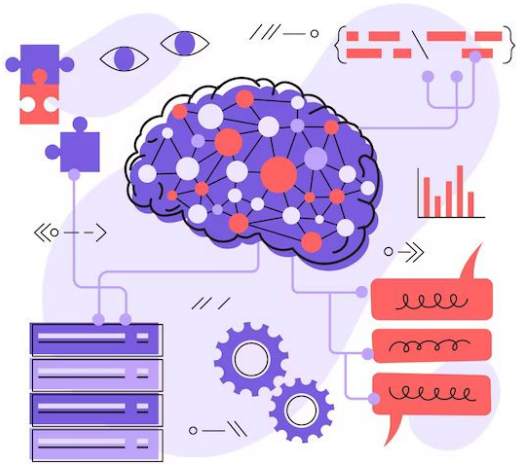
```
start = Node(state=source, parent=None, action=None)
frontier = PriorityQueueFrontier()
frontier.add(start, 0)  # 0 es la prioridad inicial

visitedNodes = set()

while not frontier.empty():
    node = frontier.remove() # Removemos nodo con mayor prioridad

    if node.state == target:
        path = []
        while node.parent is not None:
            path.append((node.action, node.state))
            node = node.parent
        path.reverse()
        return path

    visitedNodes.add(node.state)

    neighbors = neighbors_for_person(node.state)

    # Por cada vecino, verificamos que el vecino no ha sido visitado, si es asi creamos nodo child
    for action, state in neighbors:
        if state not in visitedNodes:
            child = Node(state=state, parent=node, action=action)
            priority = heuristic(state, target)
            frontier.add(child, priority)

return None
```

Algoritmo A*

```python
def heuristic(state, target):
    """
    Función de heurística que estima la distancia
    entre la persona actual (state) y la persona objetivo (target).

    F. heuristica: ofrece una medida conceptual de la dist entre un estado dado y el estado objetivo
    """
    # El - es para priorizar nodos con menos peliculas en comun (seria mas rapido)
    commonFilms = -len(people[state]["movies"].intersection(people[target]["movies"]))
    return commonFilms
```

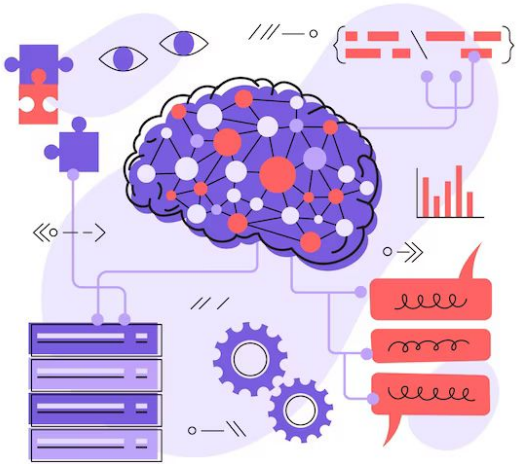
Aplicaciones similares

- Redes sociales
- Navegación y GPS
- Análisis de trayectorias en Data Science



Posibles mejoras

- Caché de búsquedas
- Paralelización




THANK YOU FOR YOUR
ATTENTION