

BITÁCORA DE USO DE ASISTENTE DE IA

Proyecto: Courier Quest - Parte 2 (IA Competitiva)

Curso: EIF-207 Estructuras de Datos

Integrantes: Emmanuel Rodriguez, Justin Briones, Josué Vargas

PROMPT a CHAT:

Tengo que meter IA a mi juego de entregas Courier Quest. El PDF dice que debo hacer 3 niveles: fácil random, medio (greedy/minimax) difícil (A*/Dijkstra) por donde empiezo?

Respuesta de Chat:

¡Perfecto! Te explico de forma simple:

NIVEL FÁCIL (Random Walk):

- La IA elige pedidos al azar
- Se mueve en direcciones aleatorias hacia el objetivo
- Usa `random.choice()` para todo

NIVEL MEDIO (Greedy/Minimax):

- La IA "piensa" 2-3 movimientos adelante
- Evalúa cada opción con una función: `score = dinero - distancia`
- Elige el mejor movimiento

NIVEL DIFÍCIL (A*/Dijkstra):

- Calcula la RUTA ÓPTIMA completa
- Usa A* (más rápido que Dijkstra)
- Considera costos: clima, superficie, resistencia

PROMPT 2: Implementación Nivel Fácil - Estructura

Prompt a CHAT:

Ok empiezo con el nivel facil.

¿Como hago que el CPU se mueva aleatorio pero que al menos intente ir hacia los pedidos? Si solo uso random puro va a estar dando vueltas, o no?.

Respuesta de CHAT:

Buena observación! Necesitas un "random inteligente":

IDEA:

1. Elige un pedido objetivo al azar
2. El 70% del tiempo → muévete HACIA el objetivo
3. El 30% del tiempo → muévete aleatorio (para variación)
4. Cambia de objetivo cada 3-6 segundos

Te doy un CÓDIGO DE ARRANQUE básico:

```
```python
def _ia_facil(self, mapa, pedidos_activos, clima_mult, consumo_clima_extra):
 """IA nivel fácil: movimiento aleatorio inteligente."""

 # Elegir objetivo si no hay
 if self.objetivo_actual is None:
 if pedidos_activos:
 pedido_random = random.choice(pedidos_activos)
 self.objetivo_actual = pedido_random.pickup

 # Decidir movimiento: 70% hacia objetivo, 30% random
 if self.objetivo_actual and random.random() < 0.7:
 # TODO: implementar _mover_hacia_objetivo()
 pass
 else:
 # TODO: implementar _mover_aleatorio()
 pass
```

CON ESTO puedes empezar. Ahora TÚ debes:

1. Implementar \_mover\_hacia\_objetivo() calculando diff\_x y diff\_y
2. Implementar \_mover\_aleatorio() con las 4 direcciones
3. Agregar lógica para cambiar objetivo cada X segundos
4. Manejar recoger/entregar pedidos

## PROMPT 3:

Mi CPU nivel fácil se queda atrapado en esquinas dando vueltas en círculos. Se mueve entre las mismas 2-3 casillas y no avanza. ¿Cómo lo arreglo?

### Respuesta de Chat:

Clásico problema de bucles! Necesitas DETECTAR cuando está atrapado:

SOLUCIÓN SIMPLE:

1. Guarda las últimas 10 posiciones en una lista
2. Si la posición actual aparece 3+ veces en las últimas 6 → BUCLE!
3. Activa "modo escape": muévete 100% aleatorio por 2 segundos

Te doy el CÓDIGO INICIAL para detectar:

```
En __init__ agrega:
```

```
self.historial_posiciones = []
```

```
self.modos_escape = False
```

```
En _ia_facil, ANTES de decidir movimiento:
```

```
self.historial_posiciones.append((self.x, self.y))
```

```
Mantener solo últimas 10
```

```
if len(self.historial_posiciones) > 10:
```

```
 self.historial_posiciones.pop(0)
```

```
Detectar bucle
```

```
if len(self.historial_posiciones) >= 6:
```

```
 pos_actual = (self.x, self.y)
```

```
 repeticiones = self.historial_posiciones[-6:].count(pos_actual)
```

```
 if repeticiones >= 3:
```

```
 self.modos_escape = True
```

```
 print("¡CPU atrapado! Activando modo escape")
```

CON ESTO ya detectas bucles. Ahora TÚ debes:

1. Agregar variables de tiempo para controlar cuánto dura el escape
2. En modo escape, forzar movimiento 100% aleatorio
3. Desactivar modo escape después de 2 segundos
4. Limpiar historial después de salir

En modo escape, ignora el objetivo y solo muévete random.

PROMT 4

Prompt A CHAT:

El PDF dice "Greedy Mini-max Expectimax".

¿cual es más fácil para un juego de entregas?

Respuesta de CHAT:

Para tu caso, EXPECTIMAX es perfecto y más simple que Minimax:

MINIMAX: Asume que el oponente juega óptimamente contra ti

- Para juegos competitivos directos (ajedrez, damas)
- NO aplica bien aquí

EXPECTIMAX: Asume que hay elementos aleatorios/probabilísticos

- Perfecto para tu juego (clima cambia, humano no es adversario)
- Más fácil de implementar

ESTRUCTURA EXPECTIMAX:

MAX node (tu turno): elige MEJOR movimiento

CHANCE node (ambiente): promedio de todos los posibles siguientes estados

Te doy CÓDIGO DE ARRANQUE:

```
def _mover_expectimax(self, mapa, clima_mult, consumo_clima_extra,
profundidad=2):
```

```
 """Usa Expectimax para decidir mejor movimiento."""
```

```

direcciones = [(0, -1), (0, 1), (-1, 0), (1, 0)]
mejor_valor = float('-inf')
mejor_movimiento = None

for dx, dy in direcciones:
 nx, ny = self.x + dx, self.y + dy

 # Verificar que sea válido
 if not (0 <= nx < len(mapa[0]) and 0 <= ny < len(mapa)):
 continue
 if mapa[ny][nx] == "B":
 continue

 # Evaluar este movimiento
 valor = self._expectimax_valor(mapa, nx, ny, profundidad - 1, False)

 if valor > mejor_valor:
 mejor_valor = valor
 mejor_movimiento = (dx, dy)

Ejecutar mejor movimiento
if mejor_movimiento:
 dx, dy = mejor_movimiento
 self.mover(dx, dy, mapa, clima_mult, consumo_clima_extra)

```

CON ESTO tienes la estructura. Ahora TÚ debes:

1. Implementar `_expectimax_valor()` recursivo (siguiente conversación)
2. Manejar casos base (profundidad = 0)
3. Implementar nodos MAX y CHANCE
4. Integrar con `_ia_media()`

Usa profundidad = 2 (suficiente para ser inteligente, no muy lento)

## Prompt 6

Nivel difícil: el PDF dice "Dijkstra o A\*". ¿Cuál uso? ¿Cuál es la diferencia? Necesito el que encuentre rutas óptimas.

### Respuesta de Chat:

Ambos encuentran la RUTA ÓPTIMA, pero A\* es MEJOR para ti:

DIJKSTRA:

- Explora en TODAS direcciones por igual
- Más lento
- $O(E \log V)$  pero visita muchos nodos

A\* (A-Star):

- Explora primero hacia el DESTINO (usa heurística)
- Más rápido
- Misma complejidad pero MENOS nodos explorados

DIFERENCIA CLAVE:

Dijkstra:  $f(n) = g(n)$

- $g(n)$  = costo desde inicio

A\*:  $f(n) = g(n) + h(n)$

- $g(n)$  = costo desde inicio
- $h(n)$  = HEURÍSTICA (estimación al destino)

HEURÍSTICA para A\*: distancia Manhattan (mismo que usaste en expectimax)

CONSEJO: Usa A\*. Es Dijkstra mejorado con heurística.

Prompt 7

Ok voy con A\*. Como se implementa?

Respuesta del Chat:

A\* tiene una ESTRUCTURA CLARA. Te doy el esqueleto:

ESTRUCTURAS DE DATOS necesarias:

```
from heapq import heappush, heappop
```

```
frontera = [] # priority queue (heap)
g_score = {} # costo desde inicio
f_score = {} # g_score + heurística
vino_de = {} # para reconstruir camino
visitados = set()
```

Te doy CÓDIGO DE ARRANQUE con la estructura básica:

```
def _a_star(self, mapa, inicio, destino, clima_mult, consumo_clima_extra):
 """Encuentra ruta óptima usando A*."""
 from heapq import heappush, heappop
```

```
 # Verificar destino válido
 if destino[1] >= len(mapa) or destino[0] >= len(mapa[0]):
 return []
 if mapa[destino[1]][destino[0]] == "B":
 return []
```

```
 # Inicializar estructuras
 frontera = []
 contador = 0 # Para desempates
 g_score = {inicio: 0}
 f_score = {inicio: self._heuristica(inicio, destino)}
 vino_de = {}
 visitados = set()
```

```
 # Agregar nodo inicial
 heappush(frontera, (f_score[inicio], contador, inicio))
 contador += 1
```

```
 direcciones = [(0, -1), (0, 1), (-1, 0), (1, 0)]
```

```
 while frontera:
 _, _, actual = heappop(frontera)
```

```
 # ¿Llegamos?
 if actual == destino:
 return self._reconstruir_camino(vino_de, actual)
```

```
 # ¿Ya visitado?
 if actual in visitados:
```

```

 continue
 visitados.add(actual)

 # Explorar vecinos
 for dx, dy in direcciones:
 vecino = (actual[0] + dx, actual[1] + dy)

 # TODO: Verificar límites y edificios
 # TODO: Calcular costo del movimiento
 # TODO: Actualizar g_score, f_score, vino_de
 # TODO: Agregar a frontera si es mejor camino

 return [] # No hay ruta

def _heuristica(self, pos_actual, pos_destino):
 """Distancia Manhattan (admisible)."""
 return abs(pos_actual[0] - pos_destino[0]) + abs(pos_actual[1] - pos_destino[1])

```

CON ESTO tienes el esqueleto. Ahora TÚ debes:

1. Completar verificaciones de límites en el loop de vecinos
2. Calcular `tentativo_g = g_score[actual] + costo_movimiento`
3. Actualizar `vino_de`, `g_score`, `f_score` si encontramos mejor camino
4. Agregar vecino a frontera con su `f_score`
5. Implementar `_reconstruir_camino()` (siguiente conversación)

Empieza con `costo = 1.0` para todos los movimientos (lo haremos dinámico después).

## PROMPT 8

Mi A\* funciona pero todas las rutas cuestan lo mismo el PDF dice: "considerar clima y superficie", como hago que las aristas tienen diferentes costos?

Respuesta del CHAT:

Necesitas una función `CALCULAR_COSTO_ARISTA` que considere factores:

FACTORES DE COSTO:

1. SUPERFICIE (del tile destino):



```

surface_weights = {
 'C': 1.0, # Calle normal
 'P': 0.95, # Parque (más rápido)
 'B': 999.0 # Edificio (bloqueado)
}

```

## 2. CLIMA:

```

Peor clima = mayor costo
costo *= (2.0 - clima_mult)

```

## 3. RESISTENCIA (penalización si está bajo):

```

if resistencia < 30:
 costo *= 1.5 # Evitar rutas largas

```

Te doy CÓDIGO INICIAL básico:

```

def _calcular_costo_arista(self, mapa, desde, hacia, clima_mult,
consumo_clima_extra):
 """Calcula costo de moverse entre dos casillas."""

 # Costo base
 costo = 1.0

 # Factor por superficie del tile destino
 tile_destino = mapa[hacia[1]][hacia[0]]
 surface_weights = {
 'C': 1.0, # Calle
 'P': 0.95, # Parque (más rápido)
 'B': 999.0 # Edificio (bloqueado)
 }
 surface_weight = surface_weights.get(tile_destino, 1.0)

 # Invertir: menor peso = menor costo
 costo /= surface_weight

 # TODO: Agregar factor de clima
 # TODO: Agregar factor de resistencia
 # TODO: Agregar consumo_clima_extra

 return costo

```

CON ESTO tienes la base. Ahora TÚ debes:

1. Agregar ajuste por clima:  $\text{costo} *= (2.0 - \text{clima\_mult})$
2. Agregar penalización si resistencia baja
3. Integrar consumo\_clima\_extra
4. Probar diferentes valores para balancear

Usa esto en la línea de A\*: `tentativo_g = g_score[actual] + self._calcular_costo_arista(...)`

Ejemplo:  $\text{clima\_mult}=1.0$  (despejado)  $\rightarrow \text{costo} * 1.0$   $\text{clima\_mult}=0.75$  (tormenta)  $\rightarrow \text{costo} * 1.25$

Prompt 9:

Mi A calcula bien las rutas, pero como hago para que la ia decida cual pedido recoger? Hay varios disponibles

Respuesta del chat:

Necesitas una FUNCIÓN DE VALOR para comparar pedidos:

NO usar solo distancia, porque:

- Pedido cercano con \$50 < Pedido lejano con \$500

FUNCIÓN DE VALOR SIMPLE:

$\text{valor} = \text{payout} / (\text{distancia} + 1)$

Te doy CÓDIGO DE ARRANQUE para evaluación:

```
def _planificar_estrategia_entregas(self, mapa, pedidos_activos, clima_mult,
consumo_clima_extra):
```

```
 """Elige el mejor pedido usando función de valor."""
```

```
 # Prioridad 1: Si tengo pedidos, entregarlos primero
```

```

if self.inventario:
 mejor_pedido = max(self.inventario, key=lambda p: p.priority)
 destino = tuple(mejor_pedido.dropoff)
 self.ruta_planeada = self._a_star(mapa, (self.x, self.y), destino, clima_mult,
consumo_clima_extra)
 return

Prioridad 2: Elegir mejor pedido disponible
if not pedidos_activos:
 self.ruta_planeada = []
 return

mejor_valor = float('-inf')
mejor_ruta = []

Evaluar CADA pedido
for pedido in pedidos_activos:
 # Verificar capacidad
 if self.peso_total() + pedido.weight > self.capacidad:
 continue

 # Calcular ruta con A*
 ruta = self._a_star(mapa, (self.x, self.y), tuple(pedido.pickup), clima_mult,
consumo_clima_extra)

 if not ruta:
 continue

 # Calcular VALOR del pedido
 distancia = len(ruta)
 valor = pedido.payout / (distancia + 1)

 # TODO: Agregar bonus por prioridad
 # TODO: Agregar penalización por clima malo
 # TODO: Considerar resistencia actual

 if valor > mejor_valor:
 mejor_valor = valor
 mejor_ruta = ruta

Guardar mejor ruta encontrada
self.ruta_planeada = mejor_ruta

```

CON ESTO evalúas todos los pedidos. Ahora TÚ debes:

1. Agregar bonus: if pedido.priority  $\geq$  1: valor  $\ast$  1.5
2. Agregar penalización clima: if clima\_mult  $<$  0.85: valor  $\ast$  0.8
3. Considerar resistencia: si baja, evitar pedidos lejanos
4. Ajustar factores para balance del juego

Esto es TSP (Traveling Salesman Problem) aproximado - elige mejor pedido en cada momento.