Bitácora de Prompts
Proyecto #2
Estructuras de Datos
CRN: 51219
Estudiantes: Ximena Ortiz Llanes
Luis Diego Alvarado Salazar
Profesor: José Pablo Calvo Suárez

Prompt: ¿Cómo puedo utilizar Greedy best-first para eliminar todo del espectimax? Siguiendo esto: Objetivo Introducir la evaluación de estados y una anticipación limitada Comportamiento
Mantiene un horizonte de anticipación pequeño (2-3 acciones por delante)
Evalúa movimientos potenciales con una función de puntuación simple, por ejemplo: score = α*(expected payout) – β*(distance cost) – γ*(weather penalty)
Selecciona el movimiento con la puntuación máxima
Opciones de implementación "Greedy best-first" (más simple que el "minimax" completo, pero con la misma
filosofía)
Respuesta: Brief fix: Replace expectimax with a greedy best-first one-ply evaluator. For each legal move, simulate the new position, call `evaluate_state`, pick the max score. Removed all expectimax methods and updated medium difficulty to use `greedy_move`. Horizon is implicitly 1 (you can raise by chaining lookahead, but requirement is small and simple).

Prompt: ¿Cómo puedo implementar el algoritmo Dijkstra con NetworkX en la clase AIController.py?
Respuesta: Here is a compact, robust Dijkstra implementation with NetworkX for AIController.py. It:
- Builds a weighted graph of only walkable tiles.
- Rebuilds the graph if weather changes.
- If a target is blocked, it routes to the nearest walkable adjacent tile.
- Fixes job priorities to prefer deliveries over pickups.

```
# Add this in __init__ (near other attributes)
self.last_weather_mult = None  # Track last weather multiplier to invalidate graph
self.graph_needs_update = True

# Add this helper anywhere in the class
def invalidate_graph(self):
    self.graph_needs_update = True

def _neighbors4(self, x, y):
    return [(x, y - 1), (x, y + 1), (x - 1, y), (x + 1, y)]
```

```
def _goal_or_adjacent_accessible(self, goal):
    """
    If goal is blocked, return nearest accessible adjacent tile.
    If no adjacent accessible tile exists, return None.
    """
    mapa = self.game.mapa
    gx, gy = goal
    if 0 <= gx < mapa.width and 0 <= gy < mapa.height and not mapa.is_blocked(gx,
gy):
        return goal

    candidates = []
    for nx_, ny_ in self._neighbors4(gx, gy):
        if 0 <= nx_ < mapa.width and 0 <= ny_ < mapa.height and not
mapa.is_blocked(nx_, ny_):
            candidates.append((nx_, ny_))
    return candidates[0] if candidates else None  # Pick first; Dijkstra will compute the
best

def get_tile_cost(self, x, y, weather):
    """
    Weighted edge cost to enter tile (x, y):
    - base surface weight (defaults to 1.0 if not available)
    - + weather modifier if bad weather
    """
    mapa = self.game.mapa
    base_cost = 1.0
    if hasattr(mapa, 'get_surface_weight'):
        base_cost = float(mapa.get_surface_weight(x, y)) or 1.0

    weather_modifier = 0.0
    if weather and hasattr(weather, 'current_multiplier'):
        weather_mult = weather.current_multiplier
        if weather_mult < 1.0:
            weather_modifier = (1.0 - weather_mult) * 2.0

    return base_cost + weather_modifier

def build_city_graph(self, weather):
    """
    Build an undirected weighted graph of the city:
    - Nodes: walkable tiles only
    - Edges: 4-direction neighbors with weight = cost to enter destination tile
    """
    mapa = self.game.mapa
    G = nx.Graph()

    # Nodes: walkable only
    for y in range(mapa.height):
```

```
        for x in range(mapa.width):
            if not mapa.is_blocked(x, y):
                G.add_node((x, y))

    # Edges: 4-neighbors
    for y in range(mapa.height):
        for x in range(mapa.width):
            if (x, y) not in G:
                continue
            for nx_, ny_ in self._neighbors4(x, y):
                if (0 <= nx_ < mapa.width and 0 <= ny_ < mapa.height and (nx_, ny_) in
G):
                    # Cost to enter destination tile
                    w = self.get_tile_cost(nx_, ny_, weather)
                    G.add_edge((x, y), (nx_, ny_), weight=w)

    self.city_graph = G

def collect_job_targets(self, inventory):
    """
    Returns: list of dict {position, type, payout, priority, job}
    Priorities: delivery (3) > pickup (2)
    """
    targets = []
    if not inventory:
        return targets

    # Deliveries first (picked jobs)
    picked_jobs = inventory.get_picked_jobs()
    for job in picked_jobs:
        targets.append({
            'position': job.dropoff,
            'type': 'delivery',
            'payout': job.payout,
            'priority': 3,
            'job': job
        })

    # Accepted but not picked up
    all_jobs = inventory.get_jobs()
    for job in all_jobs:
        if not job.is_picked_up():
            targets.append({
                'position': job.pickup,
                'type': 'pickup',
                'payout': job.payout,
                'priority': 2,
                'job': job
            })
```

```
    return targets

def choose_best_job(self, character, job_targets, weather):
    """
    score = (payout * priority) / (path_cost + 1)
    Uses Dijkstra path length on the cached graph.
    If goal is blocked, routes to nearest accessible adjacent tile.
    """
    if not job_targets or not self.city_graph:
        return None

    start = (character.tile_x, character.tile_y)
    best_target, best_score = None, float('-inf')

    for target in job_targets:
        raw_goal = target['position']
        goal = self._goal_or_adjacent_accessible(raw_goal)
        if not goal:
            continue

        try:
            path_cost = nx.dijkstra_path_length(self.city_graph, start, goal,
weight='weight')
            payout = float(target['payout'])
            priority = float(target['priority'])
            score = (payout * priority) / (path_cost + 1.0)
            if score > best_score:
                best_score, best_target = score, {**target, 'position': goal}
        except (nx.NetworkXNoPath, nx.NodeNotFound):
            continue

    return best_target

def path_to_moves(self, path):
    """
    Converts [(x1,y1), (x2,y2), ...] -> [(dx,dy), ...]
    """
    moves = []
    for i in range(len(path) - 1):
        x1, y1 = path[i]
        x2, y2 = path[i + 1]
        moves.append((x2 - x1, y2 - y1))
    return moves

def dijkstra_move(self, character, weather=None, inventory=None):
    """
    Hard difficulty: Dijkstra with NetworkX and weighted edges.
    - Rebuilds the graph if weather changed
    - Follows cached path; replans if blocked
    """
```

```
    if character.resistencia_exhausto:
        return (0, 0)

    char_inventory = inventory if inventory is not None else getattr(character,
'inventory', None)

    # Rebuild graph if needed or weather changed
    current_mult = getattr(weather, 'current_multiplier', None) if weather else None
    if self.city_graph is None or self.graph_needs_update or current_mult !=
self.last_weather_mult:
        self.build_city_graph(weather)
        self.last_weather_mult = current_mult
        self.graph_needs_update = False
        self.current_path = []
        self.current_target = None

    # Follow current path if still valid
    if self.current_path and self.current_target:
        current_pos = (character.tile_x, character.tile_y)
        if current_pos == self.current_target:
            self.current_path, self.current_target = [], None
        elif self.current_path:
            next_move = self.current_path[0]
            if self.is_valid_move(character, next_move):
                self.current_path.pop(0)
                return next_move
            else:
                # Path invalid now, replan
                self.current_path, self.current_target = [], None

    # Choose best target using current graph
    job_targets = self.collect_job_targets(char_inventory)
    if not job_targets:
        return (0, 0)

    best_target = self.choose_best_job(character, job_targets, weather)
    if not best_target:
        return (0, 0)

    start = (character.tile_x, character.tile_y)
    goal = best_target['position']

    try:
        path = nx.dijkstra_path(self.city_graph, start, goal, weight='weight')
        if len(path) < 2:
            return (0, 0)

        self.current_target = goal
        self.current_path = self.path_to_moves(path)
```

```
        if self.current_path:
            next_move = self.current_path.pop(0)
            if self.is_valid_move(character, next_move):
                return next_move
            else:
                # Shouldn't happen with clean graph; force replan next tick
                self.current_path, self.current_target = [], None
                return (0, 0)

    except (nx.NetworkXNoPath, nx.NodeNotFound):
        return (0, 0)

    return (0, 0)
```

Resultado: Se implementó el código generado por la IA y, finalmente, se le hicieron modificaciones para contemplar los trabajos adyacentes.