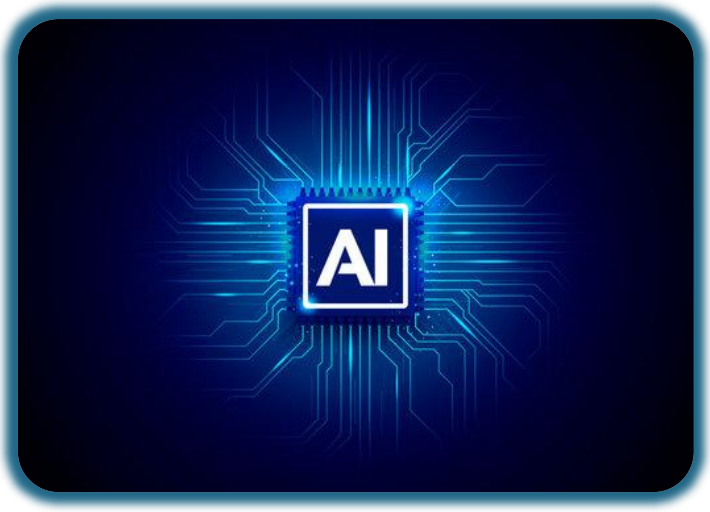


DYNAMIC THREAT-AWARE A*

Team 15

NAMES:

1. Mohamed Salem Mohamed Soliman
2. Mina Magdy Ezzat Ramzy
3. Moslem Sayed Shehata Ali
4. Abdulrahman Sobhy Hanafy
5. Hossam Eldin Omar Mohamed Sayed

INSTRUCTORS:

1. Dr. Shahira M.Habashy
2. Eng. Noor El-Deen Magdy

SUBMISSION: Saturday 17th May 2025

1. Overview

Dynamic Threat-Aware A*

Pac-Man game built using the **A*** search algorithm enhanced with dynamic heuristics. It aims to balance the following three core objectives:

- **Food Collection**: Efficiently locate and consume food pellets.
- **Ghost Avoidance**: Maintain a safe distance from dangerous ghosts.
- **Optimal Path Planning**: Minimize the overall travel distance while navigating the maze.

The agent is **highly configurable**, enabling it to adapt to different game scenarios and player preferences through adjustable parameters that control its behavior.

2. Code Approach

Step 1: Choose Pac-Man Version

We begin by selecting the appropriate version of Pac-Man. For this project, we use the Multiagent Pacman environment, which supports dynamic interactions with multiple entities like ghosts.

Step 2: Implement Custom Agent

A custom agent class is created: DynamicThreatAwareAStarAgent, where the core behavior is implemented.

Parameter	Default	Purpose
BETA	10	Weight for ghost threat avoidance
K Steps	3	Threshold distance to dynamically adjust beta
Food Weight	10	Weight for food importance in path planning

These parameters allow for significant customization:

- Setting BETA = 0 creates a food-focused agent that ignores ghosts entirely.
- Higher BETA values make the agent more cautious around ghosts.
- The K Steps parameter creates a dynamic response zone around ghosts.
- Higher Food Weight values prioritize food collection over other objectives.

3.Core Components:

- Uses **multi-source Breadth-First Search (BFS)** from ghost positions.
- Constructs a **threat map** that quantifies the danger level of each tile based on its proximity to ghosts.

2. Food-Aware Heuristic

- The heuristic balances:
  - Distance to goal (target food)
  - Inverse proximity to ghosts (more danger → higher cost)
  - Food reward (encourages reaching food-rich tiles)

3. Dynamic Beta Adjustment

- Adjusts the **BETA value** dynamically based on how close ghosts are to Pac-Man.
- This allows the agent to **increase caution** in high-risk zones (within K Steps) and **move more freely** elsewhere.

4. Intelligent Target Selection

- Selects the **optimal food pellet** to target, factoring in:
  - Distance from Pac-Man
  - Ghost threat near the food
- If no safe food is available, the agent retreats to **predetermined safe corners** on the map.

4. Implementation

4.1. Ghost Threat Mapping

Purpose:

The agent uses this method to **construct a threat map** that estimates how dangerous each tile in the maze is based on its proximity to ghosts. The threat map plays a central role in guiding Pacman's movement and decision-making, enabling the agent to **avoid areas with high ghost threat**.

```
def computeGhostThreatMap(self, gameState):
    """
    Perform multi-source BFS from all ghost positions to compute threat map.
    Returns a dictionary mapping positions to distance from closest ghost.
    """
    ghost_states = gameState.getGhostStates()
    ghost_positions = [ghost.getPosition() for ghost in ghost_states]

    threat_map = {}
    width, height = self.walls.width, self.walls.height

    fringe = Queue()
    for pos in ghost_positions:
        pos = (int(pos[0]), int(pos[1]))
        fringe.push((pos, 0))
        threat_map[pos] = 0

    while not fringe.isEmpty():
        (pos, dist) = fringe.pop()
        for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
            next_pos = (pos[0] + dx, pos[1] + dy)
            x, y = next_pos
           if 0 <= x < width and 0 <= y < height and not self.walls[x][y] and next_pos
not in threat_map:
                threat_map[next_pos] = dist + 1
                fringe.push((next_pos, dist + 1))

    return threat_map
```

the approach:

The algorithm performs a **multi-source BFS** from all ghost positions simultaneously. This ensures efficient computation of the **minimum distance** from each accessible tile to the **nearest ghost**. The result is a dictionary where each key is a position (x, y), and the value is its **threat level** (i.e., distance to the closest ghost).

4.2. Food-Aware Heuristic

Purpose:

The threat Heuristic function computes a cost estimate from the current position to a goal while factoring in:

- Distance to the goal
- Proximity to ghosts (threat)
- Presence and proximity of food (reward)

This allows Pacman to balance risk and reward, choosing paths that are safe, efficient, and nutritionally valuable.

```
def threatHeuristic(self, current_pos, goal_pos, gameState, threat_map, beta):
    """
    Compute the threat-aware heuristic with food consideration:
    h(n) = dist_to_goal(n) + beta / (1 + dist_to_ghost(n)) - food_weight * food_value(n)

    If beta is zero, ghosts are completely ignored for efficiency.
    """
    dist_to_goal = manhattanDistance(current_pos, goal_pos)

    food_grid = gameState.getFood()
    food_value = 0

     if current_pos[0] < food_grid.width and current_pos[1] < food_grid.height and
food_grid[current_pos[0]][current_pos[1]]:
        food_value += 1

    closest_food_dist = self.getClosestFoodDistance(current_pos, food_grid)
    if closest_food_dist < float('inf'):
        food_value += 1.0 / (1 + closest_food_dist)

    if beta == 0:
        return dist_to_goal - self.food_weight * food_value

    dist_to_ghost = self.getGhostDistance(current_pos, threat_map)

    if dist_to_ghost <= 1:
        return float('inf')

    threat_component = beta / (1 + dist_to_ghost)

    return dist_to_goal + threat_component - self.food_weight * food_value
```

Heuristic Formula:

The heuristic used is:

$h(n)=D(goal)+\beta/(1+D(ghost))-wfood*V(food)$

Where:

- D(goal): Manhattan distance to the goal
- D(ghost): Distance to the nearest ghost
- β: Threat penalty factor (higher = more avoidance)
- w_food: Food weight (encourages food collection)
- V(food): Food value (based on current or nearby food)

This creates a **threat-aware and food-seeking** behavior.

4.3. Dynamic Beta Adjustment

Purpose:

The adjustBeta function allows the agent to adaptively modify its ghost avoidance behavior based on how close it is to danger. Instead of using a fixed β (threat penalty), the agent increases its caution as ghosts approach, creating a “danger zone” effect.

This simulates a realistic survival instinct: Pacman becomes more evasive when ghosts are near and relax when they are farther away.

```
def adjustBeta(self, current_pos, threat_map):
    """
    Dynamically adjust beta based on ghost proximity.
    Increases beta when ghosts are within k_steps.
    """
    if self.beta == 0:
        return 0
    dist_to_ghost = self.getGhostDistance(current_pos, threat_map)

    if dist_to_ghost <= self.k_steps:
        self.last_beta = self.beta * (self.k_steps / max(1, dist_to_ghost))
        return self.last_beta
    else:
        self.last_beta = self.beta
        return self.beta
```

This creates a "danger zone" around ghosts where the agent becomes increasingly cautious as it gets closer, simulating a more realistic threat response.

4.4. Intelligent Target Selection

Purpose

This function allows the agent to dynamically select the **most strategic target**—either a food pellet or a **safe hiding corner**—based on a **balance of proximity and safety**.

Instead of blindly chasing the nearest food, the agent evaluates the **risk level** (ghost proximity) and chooses the **most rewarding and safest path**.

```
def findBestTarget(self, gameState, threat_map):
    """
    Find the best target considering food locations and safe corners.
    Prioritizes food locations but considers safety from ghosts.
    If beta is zero, simply targets the closest food.
    """
    pacman_pos = gameState.getPacmanPosition()
    food_grid = gameState.getFood()
    food_list = food_grid.asList()
```

```
    if not food_list:
        return self.findSafestCorner(gameState, threat_map)

    if self.beta == 0:
        closest_food = None
        min_distance = float('inf')

        for food_pos in food_list:
            dist = manhattanDistance(pacman_pos, food_pos)
            if dist < min_distance:
                min_distance = dist
                closest_food = food_pos

        return closest_food

    best_target = None
    best_score = float('-inf')

    for food_pos in food_list:
        dist_to_food = manhattanDistance(pacman_pos, food_pos)
        dist_to_ghost = self.getGhostDistance(food_pos, threat_map)

        score = -dist_to_food + 0.5 * dist_to_ghost

        if dist_to_ghost <= 2:
            score -= 100

        if score > best_score:
            best_score = score
            best_target = food_pos

    if best_target is None or best_score < -50:
        return self.findSafestCorner(gameState, threat_map)

    return best_target
```

The target selection logic has two modes:

1. When beta = 0: Simply select the closest food
2. When beta > 0: Score each food based on distance to Pacman and safety from ghosts

If all food pellets are too dangerous, the agent falls back to finding a safe corner using the findSafestCorner method.

4.5. A* Path Planning

Purpose

The agent uses the **A*** search algorithm to find an **optimal and safe path** from its current position to a chosen **goal** (food or safe corner).
It integrates a **custom heuristic** that considers:

- Distance to the goal
- Proximity to ghosts (danger)
- Incentives for nearby food

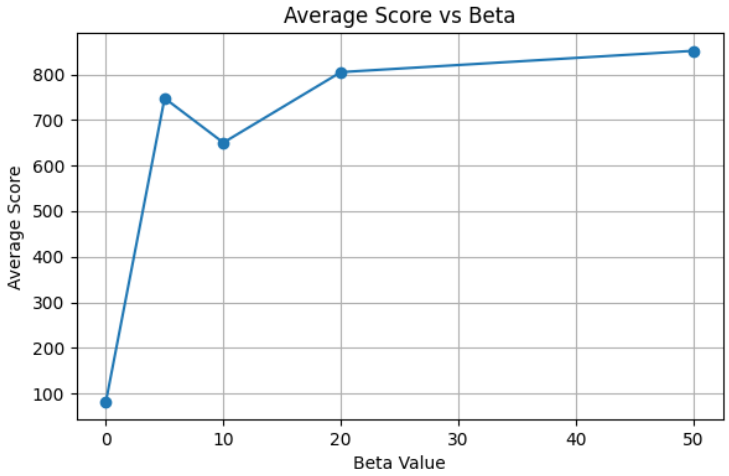
```
def aStarSearch(self, gameState, goal_pos):
    """
    A* search with the threat-aware and food-aware heuristic.
    Optimized to skip ghost calculations when beta is zero.
    """
    start_pos = gameState.getPacmanPosition()

    threat_map = {}
    if self.beta > 0:
        threat_map = self.computeGhostThreatMap(gameState)

    fringe = PriorityQueue()
    closed_set = set()

    fringe.push((start_pos, []), 0)
    g_scores = {start_pos: 0}

    while not fringe.isEmpty():
        (current_pos, path) = fringe.pop()

        if current_pos == goal_pos:
            return path

        if current_pos in closed_set:
            continue

        closed_set.add(current_pos)

        beta = 0
        if self.beta > 0:
            beta = self.adjustBeta(current_pos, threat_map)

        x, y = current_pos
        for dx, dy, action in [(0, 1, Directions.NORTH), (1, 0, Directions.EAST),
                               (0, -1, Directions.SOUTH), (-1, 0, Directions.WEST)]:
            next_pos = (x + dx, y + dy)
```

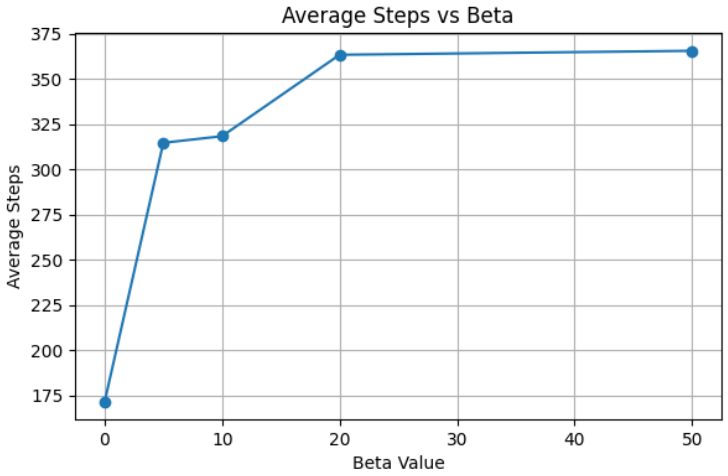
```
            if not gameState.hasWall(next_pos[0], next_pos[1]) and next_pos not in
closed_set:

                tentative_g = g_scores[current_pos] + 1

                if next_pos not in g_scores or tentative_g < g_scores[next_pos]:
                    g_scores[next_pos] = tentative_g

                     f_score = tentative_g + self.threatHeuristic(next_pos, goal_pos,
gameState, threat_map, beta)

                    new_path = path + [action]
                    fringe.push((next_pos, new_path), f_score)

    return []
```

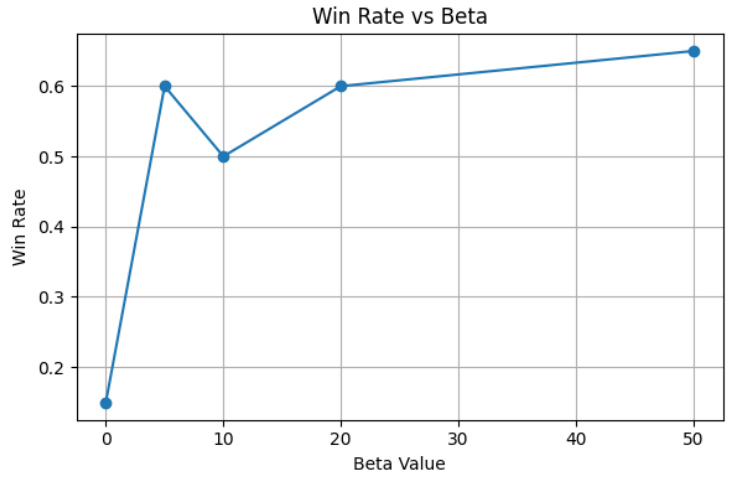
The A* implementation includes several optimizations:

1. Ghost-related calculations are skipped when beta = 0
2. A closed set prevents re-exploration of positions
3. The custom heuristic guides search toward food while avoiding ghosts

Output Comparison between Beta value and Average Score, Win Rate, and Steps:







Performance Optimizations

To enhance real-time decision-making and efficiency, several performance optimizations have been implemented in the agent:

1. Conditional Ghost Calculations
    - Ghost threat maps are computed only when necessary, reducing overhead during safe gameplay phases.
2. Path Caching
    - A* search returns full paths instead of recomputing from scratch each step, significantly improving runtime.
3. Corner Identification
    - Safe corners are detected during agent initialization, allowing fast retreat decisions in dangerous scenarios.
4. Heuristic Pruning
    - Any tile located too close to a ghost is assigned an infinite cost immediately, eliminating it from consideration early.
5. Beta = 0 Fast Path
    - When ghosts are ignored by the Pac-Man (BETA = 0), the agent switches to a simplified heuristic, providing optimal speed.

Behavioral Modes

The agent effectively has two distinct modes of operation:

Food-Focused Mode (beta = 0)

- Ignore ghosts completely.
- Directly targets the closest food.
- Uses simpler heuristic calculations.
- Optimized for performance.

Balanced Mode (beta > 0)

- Consider both food and ghosts.
- Dynamically adjusts ghost avoidance based on proximity.
- Uses sophisticated target scoring.
- Falls back to safe corners when food is too dangerous.

Fallback Mechanisms

The agent includes several fallback mechanisms to manage edge cases:

1. **No Food**: If no food is left, the agent navigates to safe corners.
2. **No Path**: If A* fails to find a path, the agent selects a random valid action.
3. **Dangerous Food**: If all food is too dangerous, the agent retreats to safe corners.
4. **No Good Targets**: If no good targets exist, the agent maximizes distance from ghosts.

Conclusion

The DynamicThreatAwareAStarAgent represents a sophisticated approach to Pac-Man gameplay that balances multiple objectives through intelligent path planning and dynamic threat assessment. By adjusting the parameters beta, k_steps, and food_weight, the agent can be tuned for different play styles from aggressive food collection to cautious ghost avoidance.

The implementation demonstrates several key AI concepts:

- Heuristic search with A*
- Multi-objective path planning
- Dynamic threat response
- Fallback strategies for edge cases

This makes the agent both effective at maximizing score and educational as an example of applied artificial intelligence techniques in game environments.