Informe del Proyecto: "Hormiga busca el Hongo"

Estudiantes

Juan David García Arroyave 2359450

Juan José Hincapié Tascón 2359493

Sebastián Zacipa Martínez 2359695

Docente

TRIANA MADRID JOSHUA DAVID

Universidad del Valle-Sede Tuluá

2025

ÍNDICE

1. Descripción General

El proyecto “Hormiga busca el Hongo” es una aplicación visual e interactiva que representa un entorno cuadrado donde una hormiga (A) debe llegar al hongo (H), evitando obstáculos (venenos marcados con “X”).

La aplicación utiliza **Tkinter** para la interfaz gráfica, donde se visualiza la cuadrícula y se permite ejecutar dos algoritmos de búsqueda informada:

- **Beam Search (Búsqueda en haz)**
- **Dynamic Weighted A*** (A* con peso dinámico)

Ambos algoritmos pueden compararse visualmente gracias a la animación paso a paso del recorrido de la hormiga.

2. Módulos del Sistema

a. Módulo environment.py

Este módulo define la clase **Environment**, que modela el tablero donde se realiza la búsqueda.

Funciones principales:

- __init__(self, size=10): inicializa el entorno.
- generate(self): genera obstáculos aleatorios, colocando la hormiga en (0,0) y el hongo en (n-1, n-1).
- get_neighbors(self, node): devuelve las celdas vecinas accesibles con sus costos.

Fragmento de código:

```python
class Environment:
    def __init__(self, size=10):
        self.size = size
        self.grid = [['.' for _ in range(size)] for _ in range(size)]
        self.start = (0, 0)
        self.goal = (size-1, size-1)
        self.generate()

    def get_neighbors(self, node):
        x, y = node
        moves = [(1,0), (-1,0), (0,1), (0,-1)]
        neighbors = []
        for dx, dy in moves:
            nx, ny = x + dx, y + dy
            if 0 <= nx < self.size and 0 <= ny < self.size:
                cell = self.grid[nx][ny]
                cost = 5 if cell == 'X' else 1
                neighbors.append(((nx, ny), cost))
        return neighbors
```

Estructura del entorno:

- “A” → Hormiga (inicio)
- “H” → Hongo (meta)
- “X” → Obstáculo (veneno)
- “.” → Celda libre

b. Módulo algorithms.py

Este módulo implementa los algoritmos de búsqueda y funciones auxiliares.

Funciones principales:

- beam_search(env, beta)
- dynamic_weighted_a_star(env, epsilon)
- heuristic(a, b)
- path_cost(path, env)

Fragmento: función heurística y reconstrucción de camino

```python
def heuristic(a, b):
    return abs(a[0]-b[0]) + abs(a[1]-b[1])

def reconstruct_path(parent, node):
    path = []
    while node is not None:
        path.append(node)
        node = parent[node]
    return list(reversed(path))
```

Fragmento: cálculo de costo total

```
def path_cost(path, env):
    total = 0
    for i in range(len(path)-1):
        curr, nxt = path[i], path[i+1]
        for (nb, cost) in env.get_neighbors(curr):
            if nb == nxt:
                total += cost
                break
    return total
```

---

c. Módulo main.py

Contiene la clase **App**, que gestiona la interfaz y la interacción con el usuario.

Funciones principales:

- draw_grid(): dibuja el tablero con colores.
- run_beam() y run_dynamic(): ejecutan los algoritmos.
- start_animation(), animate_step(): animan el recorrido.
- reset() y change_size(): regeneran o cambian el entorno.

Fragmento de código:

```
def run_beam(self):
    beta = self.beta_var.get()
    path = beam_search(self.env, beta=beta)
    if path:
        total_cost = path_cost(path, self.env)
        self.current_path = path
        self.current_step = 0
        self.draw_grid()
        self.show_path_info(path, f"Beam Search (β={beta}) – Costo: {total_cost}")
```

La interfaz utiliza botones para aplicar el tamaño, ejecutar los algoritmos y controlar la animación.

3. Algoritmos Implementados

a. Beam Search

Descripción:
Beam Search es una búsqueda informada que explora los nodos más prometedores, pero limita la cantidad de nodos expandidos en cada nivel a un parámetro β.
Es una variante de la búsqueda en anchura, optimizada en memoria y rendimiento.

Fragmento de código:

```
def beam_search(env, beta):
    start, goal = env.start, env.goal
    g = {start: 0}
    parent = {start: None}
    frontier = [start]
    visited = set([start])

    while frontier:
        successors = []
        for node in frontier:
            if node == goal:
                return reconstruct_path(parent, goal)
            for (nb, cost) in env.get_neighbors(node):
                if nb not in visited:
                    visited.add(nb)
                    g_new = g[node] + cost
                    if g_new < g.get(nb, float('inf')):
                        g[nb] = g_new
                        parent[nb] = node
                    successors.append(nb)
        successors = list(set(successors))
        successors.sort(key=lambda n: g[n] + heuristic(n, goal))
        frontier = successors[:beta]
    return None
```

Ventajas:

- Reduce el espacio de búsqueda.
- Rápido en entornos grandes.

Desventajas:

- Puede omitir la ruta óptima si β es pequeño.

b. Dynamic Weighted A*

Descripción:
Este algoritmo es una extensión de A* en la que el peso de la heurística varía dinámicamente.
El peso inicial es mayor, lo que favorece la velocidad, pero se reduce con la profundidad para mejorar la precisión.

Fórmula del peso:
w = 1 + ε * (1 - (depth / N))

Fragmento de código:

```
def dynamic_weighted_a_star(env, epsilon=1.5):
    start = env.start
    goal  = env.goal
    N = env.size * env.size

    g = {start: 0}
    parent = {start: None}
    heap = []
    counter = 0
    h0 = heuristic(start, goal)
    weight0 = 1 + epsilon * (1 - (0 / N))
    heapq.heappush(heap, (g[start] + weight0 * h0, counter, start, 0))
    closed = set()

    while heap:
        f, _, node, depth = heapq.heappop(heap)
        if node in closed:
            continue
        if node == goal:
            return reconstruct_path(parent, goal)
        closed.add(node)
        for (nb, cost) in env.get_neighbors(node):
            tentative_g = g[node] + cost
            if tentative_g < g.get(nb, float('inf')):
                g[nb] = tentative_g
                parent[nb] = node
                depth_nb = depth + 1
                h = heuristic(nb, goal)
                weight = 1 + epsilon * (1 - (depth_nb / N))
                f_nb = tentative_g + weight * h
                counter += 1
                heapq.heappush(heap, (f_nb, counter, nb, depth_nb))
    return None
```

Ventajas:

- Balance entre rapidez y exactitud.
- Se adapta dinámicamente al progreso.

Desventajas:

- Puede ser más lento que A* en entornos simples.

---

4. Código de Colores

Color y significado:

- Azul: posición inicial (hormiga).
- Verde: meta (hongo).
- Rojo: obstáculo o veneno.
- Amarillo: camino recorrido.
- Blanco: celda libre.

Ejemplo de asignación en draw_grid():

```
if self.env.grid[i][j] == 'X':
    color = "red"
elif (i, j) == self.env.start:
    color = "blue"
elif (i, j) == self.env.goal:
    color = "green"
elif (i, j) in self.current_path[:self.current_step]:
    color = "yellow"
elif (self.current_step > 0 and (i, j) == self.current_path[self.current_step - 1]):
    color = "orange"
else:
    color = "white"
```

5. Funcionamiento del Algoritmo

1. Se genera un entorno aleatorio con obstáculos.
2. El usuario selecciona un algoritmo (Beam Search o Dynamic A*).
3. El sistema calcula el camino más prometedor desde A (inicio) hasta H (meta).
4. Se muestra una animación paso a paso con cambio de colores.

5. Se muestra el costo total y la longitud del recorrido.

6. Guía de Uso

1. Ejecutar el programa:
2. python main.py
3. Configurar los parámetros:
    - Tamaño del entorno (3–30)
    - β para Beam Search
    - ε para Dynamic Weighted A*
4. Ejecutar un algoritmo desde los botones:
    - “Beam Search”
    - “Dynamic A*”
5. Controlar la animación:
    - Iniciar, pausar o avanzar paso a paso.
6. Regenerar el entorno con el botón “Regenerar”.

7. Conclusiones

- La simulación permite comparar visualmente dos algoritmos heurísticos.
- Beam Search es rápido pero puede no hallar la ruta óptima.
- La interfaz con colores facilita la comprensión del proceso de búsqueda.
- La estructura modular del proyecto facilita la extensión y mantenimiento.