FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

TA154 - Robótica Móvil

Práctica 3 -Localización EKF y Mapeo

1C2025

Dominguez Lautaro Tadeo 109645 ltdominguez@fi.uba.ar

Índice

1. FASTSLAM

El problema de SLAM (Simultaneous Localization and Mapping) consiste en que un robot móvil construya un mapa del entorno mientras se localiza dentro del mismo.

Una de las soluciones probabilísticas más eficientes a este problema es el algoritmo *FASTSLAM*, que combina el filtro de partículas con filtros de Kalman individuales para estimar las posiciones de los landmarks. En este enfoque, cada partícula representa una hipótesis de la pose del robot (posición y orientación) y contiene un conjunto de landmarks estimados mediante filtros de Kalman extendidos (EKF).

El algoritmo se divide en dos etapas principales:

- **Paso de predicción**, donde se actualiza la pose del robot según el modelo de movimiento.
- **Paso de corrección**, en el que se ajusta la estimación de los landmarks observados con los datos sensoriales, actualizando sus medias y covarianzas. Además, se recalcula el peso de cada partícula según la verosimilitud de las observaciones realizadas.

A continuación se muestra la implementación de este ultimo paso:

```
function particles = correction_step(particles, z)

% Weight the particles according to the current map of the particle
% and the landmark observations z.
% z: struct array containing the landmark observations.
% Each observation z(j) has an id z(j).id, a range z(j).range, and a bearing z(j).bearing
% The vector observedLandmarks indicates which landmarks have been observed
% at some point by the robot.

% Number of particles
numParticles = length(particles);

% Number of measurements in this time step
m = size(z, 2);

% TODO: Construct the sensor noise matrix Q_t (2 x 2)
Q_t = [0.1 0; 0 0.1];



% process each particle
for i = 1:numParticles
  robot = particles(i).pose;
  % process each measurement
  for j = 1:m
    % Get the id of the landmark corresponding to the j-th observation
    % particles(i).landmarks(l) is the EKF for this landmark
    l = z(j).id;

    % The (2x2) EKF of the landmark is given by
    % its mean particles(i).landmarks(l).mu
    % and by its covariance particles(i).landmarks(l).sigma

    % If the landmark is observed for the first time:
    if (particles(i).landmarks(l).observed == false)

      % TODO: Initialize its position based on the measurement and the current robot pose

       mux = robot(1) + z(j).range * cos( z(j).bearing + robot(3) );
       muy = robot(2) + z(j).range * sin( z(j).bearing + robot(3) );
       particles(i).landmarks(l).mu = [mux; muy];

      % get the Jacobian with respect to the landmark position
      [h, H] = measurement_model(particles(i), z(j));

      % TODO: initialize the covariance for this landmark

```

```
        Hinv = inv(H);

        particles(i).landmarks(l).sigma = Hinv * Q_t * Hinv';

      % Indicate that this landmark has been observed
      particles(i).landmarks(l).observed = true;

    else

      % get the expected measurement
      [expectedZ, H] = measurement_model(particles(i), z(j));

      % TODO: compute the measurement covariance
      sigma = particles(i).landmarks(l).sigma;
      Q =  H * sigma * H' + Q_t;

      % TODO: calculate the Kalman gain
      K = sigma * H' * inv(Q);

      % TODO: compute the error between the z and expectedZ (remember to normalize the angle
          using the function normalize_angle())
      diffZ =  [z(j).range - expectedZ(1) ; z(j).bearing - expectedZ(2)];

      % TODO: update the mean and covariance of the EKF for this landmark
      particles(i).landmarks(l).mu = particles(i).landmarks(l).mu + K *diffZ ;
      particles(i).landmarks(l).sigma =  (eye(rank(sigma))-K*H)*sigma ;

      % TODO: compute the likelihood of this observation, multiply with the former weight
      %       to account for observing several features in one time step
      w= (abs(1 / sqrt((2*pi)^2 * det(Q)))) * exp(-0.5 * diffZ' / Q * diffZ);
      particles(i).weight = particles(i).weight * w;

    end

  end % measurement loop
end % particle loop

end
```

Listing 1: Correction step FASTSLAM

Los resultados de SLAM obtenidos al aplicar el algoritmo FASTSLAM pueden visualizarse en el siguiente video, donde se observa la evolución de las partículas, la trayectoria estimada del robot y la ubicación de los landmarks detectados.

2. Planeamiento de caminos

En esta sección se implementan dos algoritmos de búsqueda para planeamiento de trayectorias en mapas de ocupación: Dijkstra y A^*. El objetivo es determinar una trayectoria desde una posición inicial hasta una meta, minimizando el costo y evitando colisiones con obstáculos representados en el mapa.

El mapa utilizado es una grilla 2D donde cada celda tiene una probabilidad de ocupación. Las celdas con alta probabilidad son tratadas como obstáculos y penalizadas en el cálculo de los costos.

2.1. Algoritmo de Dijkstra

El algoritmo de Dijkstra es una técnica clásica de búsqueda de caminos mínimos en grafos. A partir de un nodo inicial, explora los nodos vecinos con menor costo acumulado hasta alcanzar el nodo objetivo. Para su funcionamiento, se deben definir:

- Los vecinos de cada celda (nodos conectados).
- El costo de transición entre celdas, penalizando aquellas con alta ocupación.

La función **neighbors** define las celdas adyacentes (hasta 8 por celda, incluyendo diagonales) y se presenta en el siguiente código:

```
function n = neighbors(cell, map_dimensions)

  n = [];

  pos_y = cell(1);
  pos_x = cell(2);
  size_y = map_dimensions(1);
  size_x = map_dimensions(2);

  % Vecino N
  if pos_y - 1 > 0
      n = [n; pos_y - 1, pos_x];
  end
  % Vecino S
  if pos_y + 1 <= size_y
      n = [n; pos_y + 1, pos_x];
  end
  % Vecino O
  if pos_x - 1 > 0
      n = [n; pos_y, pos_x - 1];
  end
  % Vecino E
  if pos_x + 1 <= size_x
      n = [n; pos_y, pos_x + 1];
  end
  % Vecino NO
  if pos_y - 1 > 0 && pos_x - 1 > 0
      n = [n; pos_y - 1, pos_x - 1];
  end
  % Vecino NE
  if pos_y - 1 > 0 && pos_x + 1 <= size_x
      n = [n; pos_y - 1, pos_x + 1];
  end
  % Vecino SO
  if pos_y + 1 <= size_y && pos_x - 1 > 0
      n = [n; pos_y + 1, pos_x - 1];
  end
  % Vecino SE
  if pos_y + 1 <= size_y && pos_x + 1 <= size_x
      n = [n; pos_y + 1, pos_x + 1];
  end

end
```

Listing 2: neighbors function

Esta función se asegura de no considerar vecinos fuera de los límites del mapa, lo cual evita errores en la exploración.

Luego, la función *edge_cost* calcula el costo de moverse entre dos celdas adyacentes. Además de la distancia euclídea, incorpora una penalización en función de la probabilidad de ocupación:

```
function cost = edge_cost(parent, child, map)

    % Par metro: umbral de ocupaci n
    threshold = 0.3;

    % Obtener coordenadas
    x1 = parent(1);
    y1 = parent(2);
    x2 = child(1);
    y2 = child(2);

    % Calcular distancia eucl dea entre parent y child
    c_dist = sqrt((x1 - x2)^2 + (y1 - y2)^2);

    % Obtener probabilidad de ocupaci n de la celda destino (child)
```

```
    p_ocup = map(x2, y2);

    % Evaluar si es un obst culo o no
    if p_ocup > threshold
        c_ocup = inf;  % Penalizaci n grande para evitarlo
    else
        c_ocup = 0;
    end

    % Costo total = distancia + penalizaci n por ocupaci n
    cost = c_dist + c_ocup;
end
```

Listing 3: edge_cost function

En esta implementación se definió un umbral de ocupación de 0.3, ya que con valores mayores el robot atravesaba esquinas, generando trayectorias poco realistas. Si la celda destino tiene una probabilidad de ocupación superior a este umbral, se la considera un obstáculo y se le asigna un costo infinito, de modo que el algoritmo evita transitar por ella.

El resultado de aplicar el algoritmo de Dijkstra se muestra en la siguiente figura:

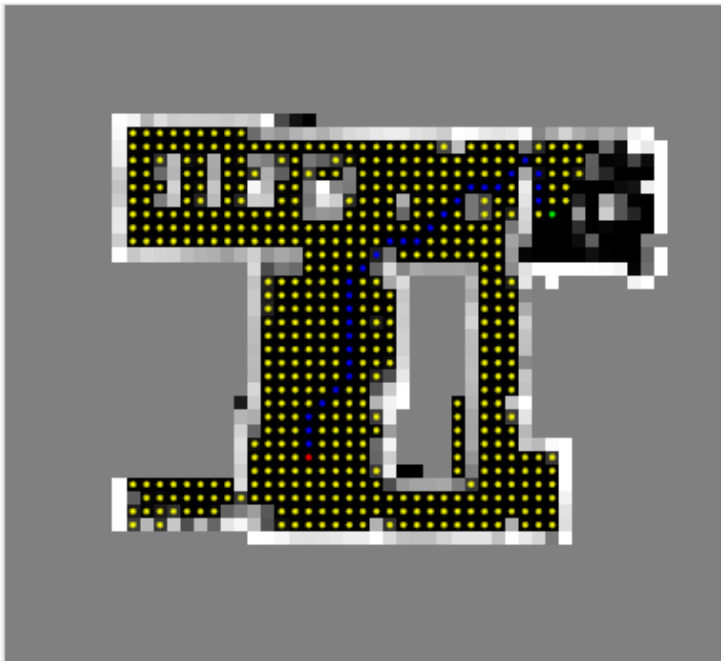

Figura 1: Trayectoria algoritmo de Dijkstra

Se puede observar que la trayectoria resultante evita correctamente las zonas ocupadas, siguiendo un camino seguro. Sin embargo, este recorrido no siempre es el más corto, ya que Dijkstra no utiliza información sobre la ubicación del objetivo para guiar su búsqueda. Como consecuencia, el algoritmo tiende a explorar una gran cantidad de nodos irrelevantes antes de encontrar la trayectoria, lo que evidencia su ineficiencia en entornos grandes o complejos.

2.2. Algoritmo A^*

El algoritmo A^* mejora a Dijkstra incorporando una heurística que estima el costo restante desde la celda actual hasta la meta. Esto permite priorizar celdas que parecen más prometedoras, reduciendo el número de exploraciones innecesarias.

Para garantizar que A^* encuentre el camino óptimo, la heurística utilizada debe ser:

- **Admisible**: La heurística nunca debe sobreestimar el costo real al objetivo, es decir:

$$h(n) \leq h^*(n)$$

  donde $h^*(n)$ es el costo real desde el nodo n hasta el objetivo.

- **Consistencia**: Los costos deben mantenerse coherentes a lo largo del grafo.

En este caso se utilizó como heurística la distancia euclídea entre la celda actual y el objetivo, implementada en la función **heuristic**:

```
function heur = heuristic(cell, goal)

  heur = 0;

    dx = abs(goal(1) - cell(1));
    dy = abs(goal(2) - cell(2));

    heur = sqrt(dx^2 + dy^2);

end
```

Listing 4: heuristic function

Este enfoque permite una exploración más dirigida que Dijkstra, acortando el tiempo de cálculo sin sacrificar la calidad del camino.

A continuación se presenta el resultado del planeamiento con A^*:

Comparando con el resultado de Dijkstra, se observa que A^* encuentra un camino más directo hacia el objetivo, demostrando su eficiencia.

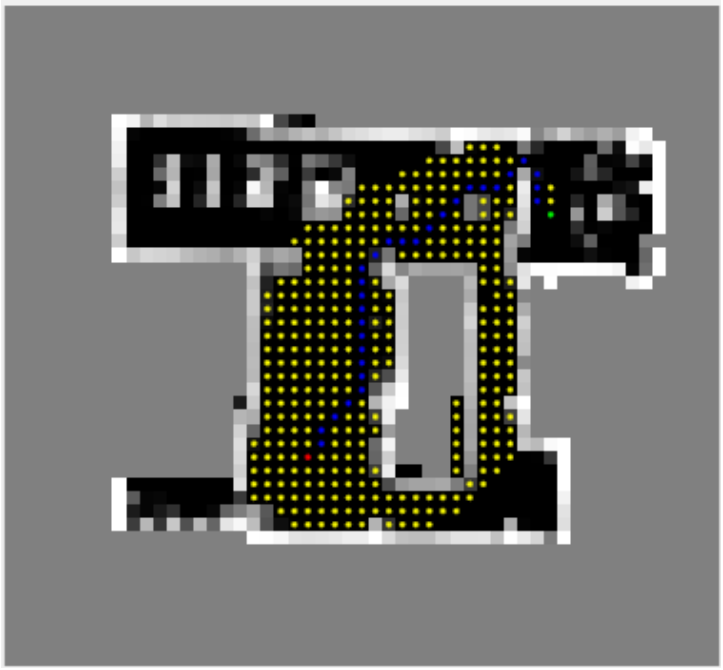

Figura 2: Trayectoria algoritmo A*

Comparando con el resultado de Dijkstra, se observa que A^* encuentra un camino más directo hacia el objetivo explorando una menor cantidad de nodos, demostrando su eficiencia.

A continuación se presentan los distintos casos obtenidos al modificar la heurística como un múltiplo de la original, utilizando los factores: $\{1, 2, 5, 10\} \cdot h$.

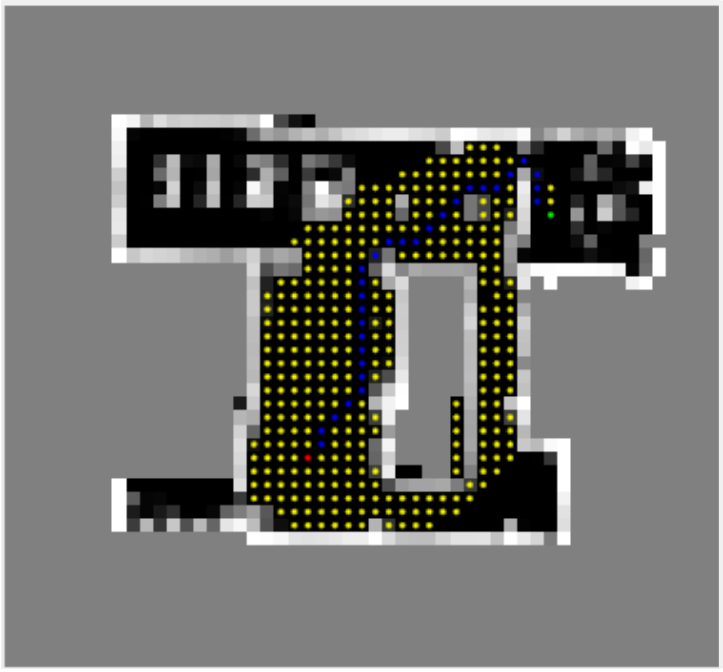

Figura 3: Trayectoria algoritmo A* con factor 1

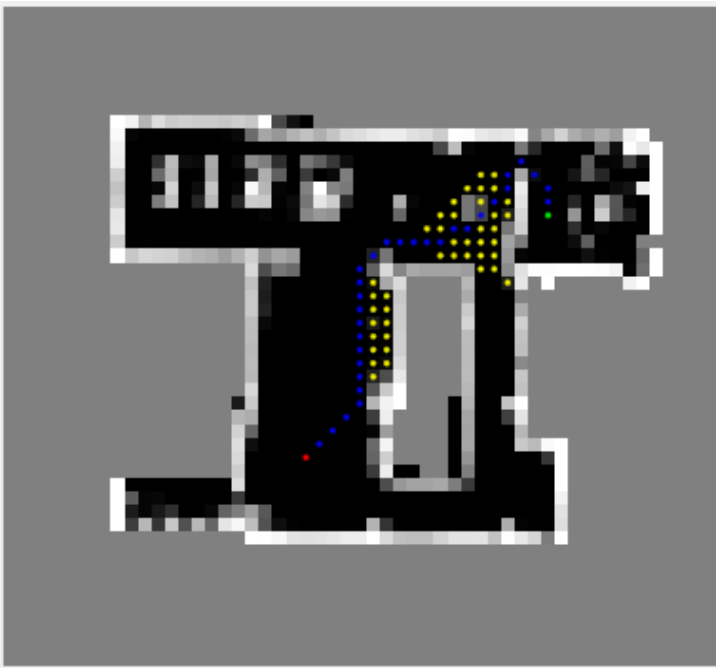

Figura 4: Trayectoria algoritmo A* con factor 2

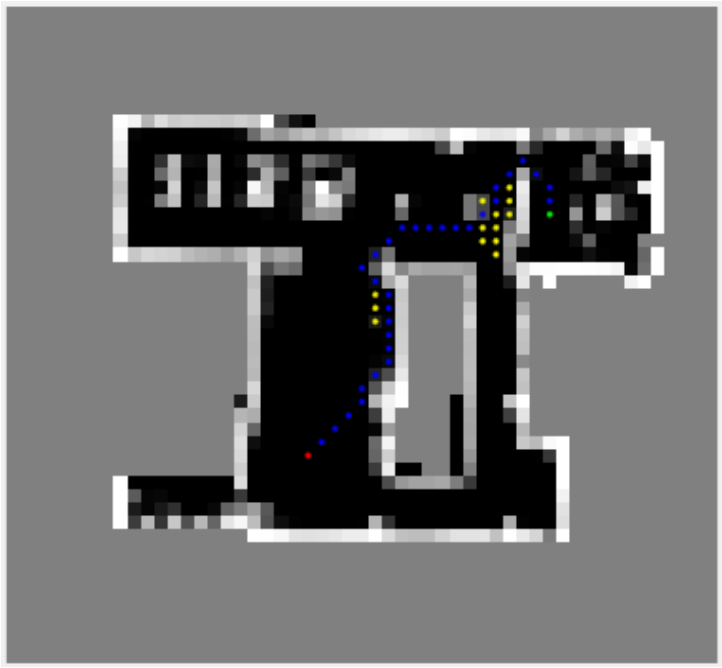

Figura 5: Trayectoria algoritmo A* con factor 5

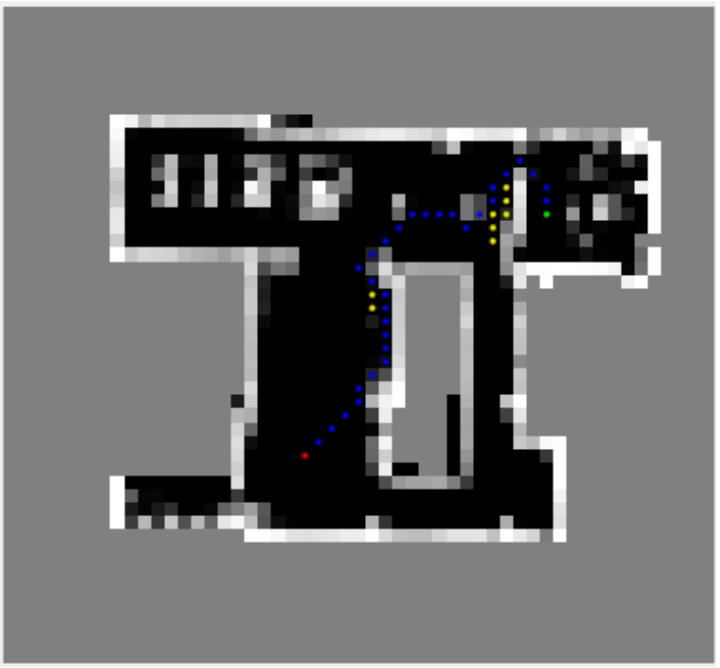

Figura 6: Trayectoria algoritmo A* con factor 10

Se puede observar que, a medida que se incrementa el factor de la heurística, el algoritmo A* prioriza cada vez más la cercanía al objetivo, dejando en segundo plano el costo real del trayecto. Esto genera caminos más directos, con mayor velocidad de convergencia, pero potencialmente menos óptimos.