PROYECTO: BÚSQUEDA DE RUTAS EN UN PROBLEMA DE LOGÍSTICA - IA 2018/19

Álvaro Galisteo y Javier Fernández

I. Introducción

Con este proyecto, buscamos desarrollar un programa, o inteligencia artificial que resuelva el siguiente problema:

Un repartidor de una cadena de pizzerías, debe entregar pedidos a distintos clientes, comenzando y regresando a un mismo punto inicial.

En la parte básica de este proyecto, el objetivo es entregar a un único cliente. Por otro lado, en la parte avanzada, además de entregar a más de un cliente desde más de una pizzería, hemos decidido añadir características adicionales, como distintos niveles o alturas para formar cuestas, una motocicleta eléctrica, coste de llevar pizzas y distintos terrenos con costes variados, así como sus respectivos nuevos escenarios. Las nuevas imágenes de esta parte avanzada se encuentran en la carpeta `images` dentro de la carpeta `student-advanced`.

II. Descripción de la tarea realizada

Parte básica

Representación de estados

Los estados en nuestro código están definidos con la siguiente tupla:

$$(posicion, pizzas, pedidos)$$

. Con:

- $posicion = (r_x, r_y)$
  - r_x: posición x del repartidor
  - r_y: posición y del repartidor
- $pizzas$: número de pizzas cargadas
- $pedidos : (c_x, c_y, c_p)$
  - c_x: posición x del cliente
  - c_y: posición y del cliente
  - c_p: cantidad pedida por el cliente

Definimos el estado inicial, en `initial_state`, con todos los valores anteriormente mencionados y el estado final, en `final_state`, con los mismos valores del estado inicial, a excepción de c_p que será 0.

Acciones y restricciones

A partir de este estado, definimos las siguientes acciones:

- **North**: Acción que define que el repartidor se puede mover hacia el norte. Este movimiento es equivalente a reducir r_y en una unidad
- **South**: Acción que define que el repartidor se puede mover hacia el sur. Este movimiento es equivalente a aumentar r_y en una unidad
- **East**: Acción que define que el repartidor se puede mover hacia el este. Este movimiento es equivalente a aumentar r_x en una unidad
- **West**: Acción que define que el repartidor se puede mover hacia el oeste. Este movimiento es equivalente a reducir r_x en una unidad
- **Unload**: Acción que define que el repartidor puede descargar la cantidad de pizzas requeridas por el cliente.
- **Load**: Acción que define que el repartidor puede cargar una cantidad de pizzas desde una pizzería.

Por otro lado, estas acciones tienen aplicadas las siguientes restricciones:

El repartidor sólo se puede desplazar hacia cualquiera de las direcciones si la siguiente posición no se encuentra fuera de los límites del mapa y si la siguiente posición no es una casilla bloqueada, como una casa sin pedidos. Además, el repartidor sólo puede descargar si se encuentra en una casa que ha realizado un pedido, en cuyo caso descargará todas las pizzas de una sola vez. Finalmente, el repartidor podrá cargar pizzas de una en una (para simular que una pizza se está haciendo) y si se encuentra en una pizzería.

Métodos implementados

En la parte básica podemos encontrar hemos implementado las funciones provistas por los profesores. Cada una de ellas hace lo siguiente:

- `actions(state)`: Devuelve la lista completa de acciones que se pueden ejecutar en el estado actual. En ella se encuentran las posibles acciones del repartidor de pizzas que han sido descritas anteriormente junto a sus restricciones.
- `result(state, action)`: Devuelve el estado resultado que se obtiene a partir del estado actual y la acción. Aquí se implementa la actualización del estado a partir de la acción provista. Como hemos comentado antes, en la acción *Unload*, se descargan todas las pizzas de una única vez, mientras que en la acción `Load`, se cargan las pizzas de una en una.
- `is_goal(state)`: Devuelve True si se ha alcanzado el estado final. Para ello comprueba que no haya más entregas por realizar que el repartidor no lleve ninguna pizza encima y que el repartidor se encuentre en la posición inicial.
- `cost(state, action, state2)`: Devuelve el coste de aplicar la acción desde un estado a otro. Como esta es la parte básica, el coste de avanzar es de una unidad y el coste de descarga y carga es de una unidad por pizza.
- `heuristic(state)`: Devuelve el valor de la función heurística aplicada al estado. La heurística se describe a continuación en la siguiente sección.
- `setup()`: Función que inicializa el problema para su resolución. Establece los estados iniciales y finales, asi como el algoritmo a usar y la ubicación de los clientes y de la posición final.
- `printState(state)`: Devuelve una cadena de caracteres para mostrar el estado de forma "bonita" en la ventana del problema.
- `getPendingRequests(state)`: Devuelve la cantidad de pizzas pendientes para la posición en la que se encuentra el repartidor. Si la posición en la que se encuentra no es un cliente, devuelve `None`.

Heurística

La heurística que utilizamos es la siguiente:

- d_c: Distancia en línea recta desde la posición del repartidor a la posición del cliente
- d_0: Distancia en línea recta al punto de origen
- p_r: Número de pizzas restantes por entregar

$$h = \begin{cases} d_c + c_p - pizzas & \text{si } p_r > 0 \\ d_0 & \text{si } p_r = 0 \end{cases}$$

En nuestra heurística podemos diferenciar dos casos: cuando no se han entregado todas las pizzas y cuando se han entregado todas. En el primer caso, la heurística que utiliza es h_i que devuelve un valor basado en la distancia hacia el cliente, la cantidad del pedido y la cantidad de pizzas que lleva. Por otro lado, en el segundo caso, la heurística que utiliza es h_f, basada en la distancia en línea recta hasta el punto final.

Además, se comprueba fácilmente que la heurística es admisible.

Parte avanzada ### Representación de estados Los estados en nuestro código están definidos con la siguiente tupla:

$$(posicion, pizzas, pedidos, electrico, carga, nivel)$$

. Con:

- $posicion = (r_x, r_y)$
  - r_x: posición x del repartidor
  - r_y: posición y del repartidor
- $pizzas$: número de pizzas cargadas
- $pedidos = ((c_x, c_y, c_p), ...)$: Tupla con las posiciones y las cantidades pedidas por cada cliente.
  - c_x: posición x del cliente
  - c_y: posición y del cliente
  - c_p: cantidad pedida por el cliente
- $electrico = True \mid False$: Variable booleana que define si el repartidor lleva una bicicleta o una motocicleta eléctrica.
- $carga$: Nivel de carga de la motocicleta eléctrica. Si no es eléctrica, la carga es siempre 100%
- $nivel$: Nivel o altura del terreno en el que se encuentra actualmente

Definimos el estado inicial, en `initial_state`, con todos los valores anteriormente mencionados y el estado final, en `final_state`, con los mismos valores del estado inicial, a excepción de todos los c_p que serán 0.

Acciones y restricciones

A partir de este estado, definimos las siguientes acciones:

- **North**: Acción que define que el repartidor se puede mover hacia el norte. Este movimiento es equivalente a reducir r_y en una unidad
- **South**: Acción que define que el repartidor se puede mover hacia el sur. Este movimiento es equivalente a aumentar r_y en una unidad
- **East**: Acción que define que el repartidor se puede mover hacia el este. Este movimiento es equivalente a aumentar r_x en una unidad
- **West**: Acción que define que el repartidor se puede mover hacia el oeste. Este movimiento es equivalente a reducir r_x en una unidad
- **Unload**: Acción que define que el repartidor puede descargar la cantidad de pizzas requeridas por el cliente.
- **Load**: Acción que define que el repartidor puede cargar una cantidad de pizzas desde una pizzería.
- **Recharge**: Acción que define que el repartidor puede recargar la batería de la motocicleta eléctrica.

Por otro lado, estas acciones tienen aplicadas las siguientes restricciones:

El repartidor sólo se puede desplazar hacia cualquiera de las direcciones si la siguiente posición no se encuentra fuera de los límites del mapa, ni es una casilla bloqueada, como una casa sin pedidos, ni es una casa cuyo pedido ya ha sido entregado, ni puede desplazarse de una pizzería, cargador o casa con pedido a una casa adyacente. En este último caso, debe salir primero a la calle.

Además, el repartidor sólo puede descargar si se encuentra en una casa que ha realizado un pedido, en cuyo caso descargará todas las pizzas de una sola vez. El repartidor podrá también cargar pizzas de una en una (para simular que una pizza se está haciendo) y si se encuentra en una pizzería. Este podrá, además, cargar la batería si se encuentra en una pizzería o en un cargador, y su nivel de batería es inferior a 100%. Esta batería recargará un 20% cada vez para simular la carga lenta de ésta.

Por otro lado, el repartidor no podrá saltar a un nivel cuya diferencia entre este y el actual sea mayor a uno y no podrá avanzar si se queda sin batería. Por último, la batería se descarga un 10% por cada casilla que avanza, obteniendo un rango máximo de 10 casillas. Si está subiendo una cuesta, la batería se descarga el doble y si está bajándola, la batería se recarga un poco.

Métodos implementados

En la parte avanzada podemos encontrar las siguientes funciones, necesarias para el correcto funcionamiento de la búsqueda. Entre ellas se encuentran:

- `getRemaining(customers)`: Esta función devuelve el número de pizzas que quedan por entregar. Esta función es necesaria debido a que la parte avanzada contiene varios clientes dispersos por el mapa.

- `canPass(pos, state)`: Esta función devuelve un valor booleano. Este valor depende de las restricciones anteriormente mencionadas.
- `getDischarge(pos, electric)`: En el caso de que la bicicleta sea eléctrica, esta función devuelve el porcentaje de batería que se descarga.
- `actions(state)`: Devuelve la lista completa de acciones que se pueden ejecutar en el estado actual. En ella se encuentran las posibles acciones del repartidor de pizzas que han sido descritas anteriormente junto a sus restricciones.
- `result(state, action)`: Devuelve el estado resultado que se obtiene a partir del estado actual y la acción. Aquí se implementa la actualización del estado a partir de la acción provista. Como hemos comentado antes, en la acción *Unload*, se descargan todas las pizzas de una única vez, mientras que en la acción *Load*, se cargan las pizzas de una en una. Por otro lado *Recharge* carga la batería en un 20% hasta llegar al 100% y además actualiza el nivel en el que se encuentra. Si la casilla siguiente no tiene el atributo nivel, el nivel se mantiene.
- `is_goal(state)`: Devuelve True si se ha alcanzado el estado final. Para ello comprueba que no haya más entregas por realizar que el repartidor no lleve ninguna pizza encima y que el repartidor se encuentre en la posición inicial.
- `cost(state, action, state2)`: Devuelve el coste de aplicar la acción desde un estado a otro. El coste de avanzar depende de la casilla a la que avance, además de las pizzas que lleve. Subir de nivel cuesta también, 10 veces el peso de una pizza, pero bajar de nivel, cuesta un 25% del coste total de avanzar. Por último, si es una motocicleta eléctrica, cuanta menos carga tenga, más cuesta avanzar. Recargar no tiene coste y es gratuito.
- `distanceToNearestCharger(customer)`: Devuelve la distancia entre el cliente la estación de carga más cercana. Esta función se utiliza para las heurísticas descritas más adelante.
- `distanceToNearestShop(customer)`: Devuelve la distancia entre el cliente y la pizzería más cercana, además de la posición de esta pizzería. Utilizado para las heurísticas.
- `heuristic(state)`: Devuelve el valor de la función heurística aplicada al estado. Las heurísticas se describen a continuación en la siguiente sección.
- `setup()`: Función que inicializa el problema para su resolución. Establece los estados iniciales y finales, asi como el algoritmo a usar y la ubicación de los clientes y de la posición final. Establece además si el repartidor lleva una motocicleta electrica, dependiendo de si existe algún cargador en el mapa o no.
- `printState(state)`: Devuelve una cadena de caracteres para mostrar el estado de forma "bonita" en la ventana del problema.
- `getPendingRequests(state)`: Devuelve la cantidad de pizzas pendientes para la posición en la que se encuentra el repartidor. Si la posición en la que se encuentra no es un cliente, devuelve `None`.

Además, comentar que como el algoritmo de búsqueda sólo puede hacer uso de tuplas, en la función `result(...)`, debemos convertir previamente la tupla *pedidos*, a una lista para poder modificar cuando se entregan pizzas a un cliente. De la misma forma, debemos revertir el proceso cuando la función va a devolver el nuevo estado. El código es el siguiente:

```
# Convierte de tupla de tuplas a lista de listas
customers = list(customers)
for i in xrange(len(customers)):
    customers[i] = list(customers[i])

...

# Revierte el proceso
for i in xrange(len(customers)):
    customers[i] = tuple(customers[i])

... tuple(customers) ...
```

Heurísticas

Para implementar distintas heurísticas realizamos la siguiente pregunta a distintas personas:

¿Si fueses un repartidor de pizza, a que casas repartirías primero?

A partir de estas respuestas llegamos a la implementación de 11 heurísticas distintas, cuya definición esperamos que se cumpla.

Distancia en línea recta hasta la posición inicial.

Ésta heurística sólo se usa cuando no quedan pizzas por entregar. Es una heurística trivial basada en la distancia Manhattan. Es común a todas las otras heurísticas.

- d_0: Distancia en línea recta al punto de origen

$$h_{final} = d_0$$

1) Casas más lejanas primero.

- p: Cantidad de pizzas que lleva el repartidor
- l: Nivel en el que se encuentra el repartidor actual
- d_m: Distancia máxima entre la posición actual del repartidor y la casilla más lejana
- c_e: Carga de la motocicleta eléctrica. Si no es eléctrica, el nivel es siempre 100%
- c_c: coste de la casilla actual

$$h_1 = \begin{cases} (p \cdot l + \sum_i (d_m + 1 - d_i)) \cdot (1 - c_e) \cdot \frac{c_c \cdot (l+1)}{2} & \text{si } p_r > 0 \\ h_{final} & \text{si } p_r = 0 \end{cases}$$

2) Casas que más cercanas primero.

$$h_2 = \begin{cases} (p \cdot l + \sum_i (d_i)) \cdot (1 - c_e) \cdot \frac{c_c \cdot (l+1)}{2} & \text{si } p_r > 0 \\ h_{final} & \text{si } p_r = 0 \end{cases}$$

3) Casas más lejanas primero y con mayor pedido.

- m_p: Pedido máximo. Por defecto, 3
- p_i: Pedido del cliente i

$$h_3 = \begin{cases} (p \cdot l + \sum_i ((d_m + 1 - d_i) + (m_p + 1 - p_i)) \cdot (1 - c_e) \cdot \frac{c_c \cdot (l+1)}{2} & \text{si } p_r > 0 \\ h_{final} & \text{si } p_r = 0 \end{cases}$$

4) Casas más lejanas primero y con menor pedido.

$$h_4 = \begin{cases} (p \cdot l + \sum_i ((d_m + 1 - d_i) + p_i) \cdot (1 - c_e) \cdot \frac{c_c \cdot (l+1)}{2} & \text{si } p_r > 0 \\ h_{final} & \text{si } p_r = 0 \end{cases}$$

5) Casas con cargador más cercano y menor pedido primero. Si la motocicleta no es eléctrica, se cambia a la heurística 3.

- d_{ic}: Distancia entre el cliente i y el cargador más cercano

$$h_5 = \begin{cases} (p \cdot l + \sum_i ((m_p + 1 - p_i) + d_{ic}) \cdot (1 - c_e) \cdot \frac{c_c \cdot (l+1)}{2} & \text{si } p_r > 0 \\ h_{final} & \text{si } p_r = 0 \end{cases}$$

6) Agrupar casas por zonas alrededor de una pizzería. Intenta ir primero a las casas cuya distancia entre "*repartidor - pizzería - casa*" sea menor.

- $f_d(i)$: Funcion que devuelve la distancia entre el cliente i y la pizzería más cercana
- d_p: Distancia desde la posición actual del repartidor y la pizzería p
- $f_p(i)$: Funcion que devuelve la posición de la pizzería más cercana al cliente i

$$h_6 = \begin{cases} (p \cdot l + \sum_i (f_d(i) + d_{f_d(i)}) \cdot (1 - c_e) \cdot \frac{c_c \cdot (l+1)}{2} & \text{si } p_r > 0 \\ h_{final} & \text{si } p_r = 0 \end{cases}$$

$$f_d(i) = min(d_{ip}) \ / \ \forall i \in clientes, \forall p \in pizzerias$$

7) Variación de la heurística 6, priorizando también las casas con mayor pedido.

$$h_7 = \begin{cases} (p \cdot l + \sum_i (f_d(i) + d_{f_d(i)} + (m_p + 1 - p_i)) \cdot (1 - c_e) \cdot \frac{c_c \cdot (l+1)}{2} & \text{si } p_r > 0 \\ h_{final} & \text{si } p_r = 0 \end{cases}$$

8) Variación de la heurística 6, priorizando también las casas con menor pedido.

$$h_8 = \begin{cases} (p \cdot l + \sum_i (f_d(i) + d_{f_d(i)} + p_i) \cdot (1 - c_e) \cdot \frac{c_c \cdot (l+1)}{2} & \text{si } p_r > 0 \\ h_{final} & \text{si } p_r = 0 \end{cases}$$

9) Variación de la heurística 6, priorizando también las casas con cargador más cercano. Si la motocicleta no es eléctrica, se cambia a la huerística 6.

$$h_9 = \begin{cases} (p \cdot l + \sum_i (f_d(i) + d_{f_d(i)} + d_{ic}) \cdot (1 - c_e) \cdot \frac{c_c \cdot (l+1)}{2} & \text{si } p_r > 0 \\ h_{final} & \text{si } p_r = 0 \end{cases}$$

10) Llenar la bolsa y entregar pequeñas cantidades

- m_b: Máximo de pizzas que puede llevar

$$h_{10} = \begin{cases} (p \cdot l + \sum_i (d_i + p_i + (m_b + 1 - p)) \cdot (1 - c_e) \cdot \frac{c_c \cdot (l+1)}{2} & \text{si } p_r > 0 \\ h_{final} & \text{si } p_r = 0 \end{cases}$$

11) Llenar la bolsa y entregar los pedidos más grandes primero

$$h_{11} = \begin{cases} (p \cdot l + \sum_i (d_i + (m_p + 1 - p_i) + (m_b + 1 - p)) \cdot (1 - c_e) \cdot \frac{c_c \cdot (l+1)}{2} & \text{si } p_r > 0 \\ h_{final} & \text{si } p_r = 0 \end{cases}$$

Como podemos observar, todas las heurísticas tienen elementos en común, como la cantidad de pizzas que lleva por el nivel en el que se encuentra $(p \cdot l)$, el desaconsejado de llevar una carga de batería muy baja $(1 - c_e)$ y la aproximación en coste a partir del nivel actual y la casilla actual $(\frac{c_c \cdot (l+1)}{2})$, por lo que, en el código, hemos extraído estos parámetros a partes comunes:

```
# Pizzas por nivel actual
h = pizzas * level

# Heurística -1
if self.getRemaining(customers) == 0:
    distX = math.fabs(self.GOAL[0][0] - pos[0])
    distY = math.fabs(self.GOAL[0][1] - pos[1])
    h = math.sqrt(math.pow(distX, 2) + math.pow(distY, 2))
```

```
#...

# Desaconsejación de carga de batería baja
if electric:
    h *= 1-(charge/100)

# Aproximación de coste
return math.ceil(h * (self.getAttribute(pos, 'cost')/2 * (level+1)))
```

Por otro lado, la heurística a usar se puede cambiar con la variable `heuristic`.

Finalmente, al tener en cuenta distancias en línea recta en todas las heurísticas, junto con la aproximación del coste y del nivel y de la carga de pizzas y de batería, podemos observar fácilmente que las heurísticas son admisibles y no sobre-estiman el coste real.

III. Experimentación

Todas las pruebas de experimentación han sido realizadas en un sistema con las siguientes características:

- Procesador Intel Core i5 a 1,6 GHz,
- RAM de 4GB
- Sistema operativo basado en GNU/Linux con 3,31GB de espacio libre inicial en RAM

Parte básica

¿Que diferencia hay entre heurística y sin heurística?

Con este experimento queremos comprobar como se comporta nuestra heurística frente a ninguna heurística con algoritmos que hacen uso de ella. Para ello hemos hemos creado distintos mapas, separados en dos bloques. El primer bloque (map_test1.txt a map_test10.txt) consiste en mapas cuyo ancho va creciendo en una unidad a medida que avanzamos al siguiente mapa, mientras que el segundo bloque (map_test11.txt a map_test17.txt) consiste en mapas cuya posición del cliente se va alejando a medida que avanzamos de nuevo al siguiente mapa.

Una vez ejecutados los test, hemos obtenido los siguientes resultados. Como son muchos mapas y no queremos ocupar mucho espacio en este docuemento, hemos decidido mostrar sólo las **medias aritméticas** de los dos algoritmos. Los resultados completos se pueden encontrar en el *siguiente archivo de Excel*.

Además, como el número de iteraciones y de nodos expandidos es el mismo, vamos a juntarlos en la misma columna.

Tabla 1a: Sin heurística. Bloque de mapas 1

Algoritmo	Longitud	Coste	Iteraciones/Nodos	Lista Abierta
greedy	17	14	80	11
astar	17	14	83	12

Tabla 1b: Sin heurística. Bloque de mapas 2

Algoritmo	Longitud	Coste	Iteraciones/Nodos	Lista Abierta
greedy	40	35	369	27
astar	46	41	296	29

Podemos observar que A* obtiene un peor resultado en el segundo bloque de mapas, comparado con Greedy en ambos bloques y con A* en el primer bloque, obteniendo un coste y longitud mayores que Greedy. Además A* hace ligeramente un mayor uso de memoria.

Tabla 2a: Con heurística. Bloque de mapas 1

Algoritmo	Longitud	Coste	Iteraciones/Nodos	Lista Abierta
greedy	17	14	28	13
astar	17	14	61	12

Tabla 2b: Con heurística. Bloque de mapas 2

Algoritmo	Longitud	Coste	Iteraciones/Nodos	Lista abierta
greedy	40	35	286	35
astar	40	35	372	37

Sin embargo, podemos observar que el uso de una heurística favorece a A* obteniendo unos resultados en longitud y coste iguales a Greedy. Además, el uso de la heurística mejora a ambos algoritmos, reduciendo el

tiempo de búsqueda (número de iteraciones) a cambio de sacrificar ligeramente el tamaño máximo de la lista abierta en memoria.

A partir de estos datos, podemos llegar a la conclusión de que la implementación de una heurística mejora la búsqueda para los algoritmos que hacen uso de ella, sobre todo con el algoritmo A*, pero estas mejoras son sólo posibles sacrificando espacio en memoria.

¿Que diferencias encontramos entre los distintos algoritmos?

A continuación deseamos comparar todos los algoritmos, haciendo uso de los mapas usados en la experimentación anterior. Como Greedy y A* hacen uso de heurísticas, vamos a comparar estos dos cuando la heurística está disponible y cuando no, y los vamos a comparar con todos los algoritmos.

Una vez ejecutados los test, hemos obtenido los siguientes resultados. Como son muchos mapas hemos decidido mostrar sólo las **medias aritméticas** de todos los algoritmos. De nuevo, los resultados completos se pueden encontrar en el *siguiente archivo de Excel*.

Tabla 3a: Sin heurística. Bloque de mapas 1

Algoritmo	Longitud	Coste	Iteraciones/Nodos	Lista Abierta
greedy	17	14	80	11
astar	17	14	83	12
breadth_first	17	14	85	9
depth_first	23	20	103	20

Tabla 3b: Sin heurística. Bloque de mapas 2

Algoritmo	Longitud	Coste	Iteraciones/Nodos	Lista Abierta
greedy	40	35	369	27
astar	46	41	296	29
breadth_first	40	35	372	21
depth_first	65	60	230	32

Tabla 4a: Con heurística. Bloque de mapas 1

Algoritmo	Longitud	Coste	Iteraciones/Nodos	Lista Abierta
greedy	17	14	28	13
astar	17	14	61	12
breadth_first	17	14	85	9
depth_first	23	20	103	20

Tabla 4b: Con heurística. Bloque de mapas 2

Algoritmo	Longitud	Coste	Iteraciones/Nodos	Lista abierta
greedy	40	35	286	35
astar	40	35	372	37
breadth_first	40	35	372	21
depth_first	65	60	230	32

Figura 1: Sin heurística. Iteraciones respecto a mapa (Bloque de mapas 1 y 2)

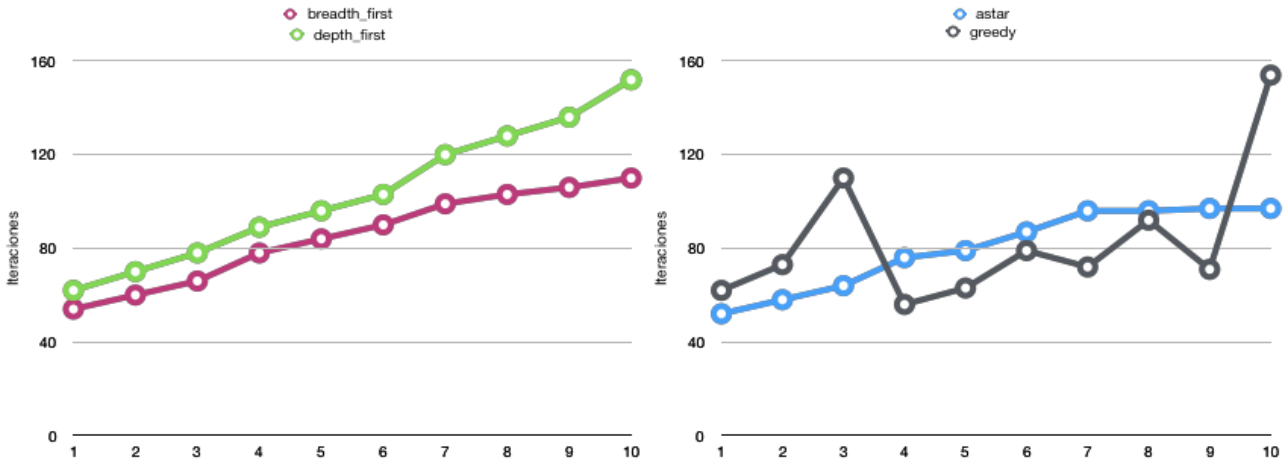

Figure 1:

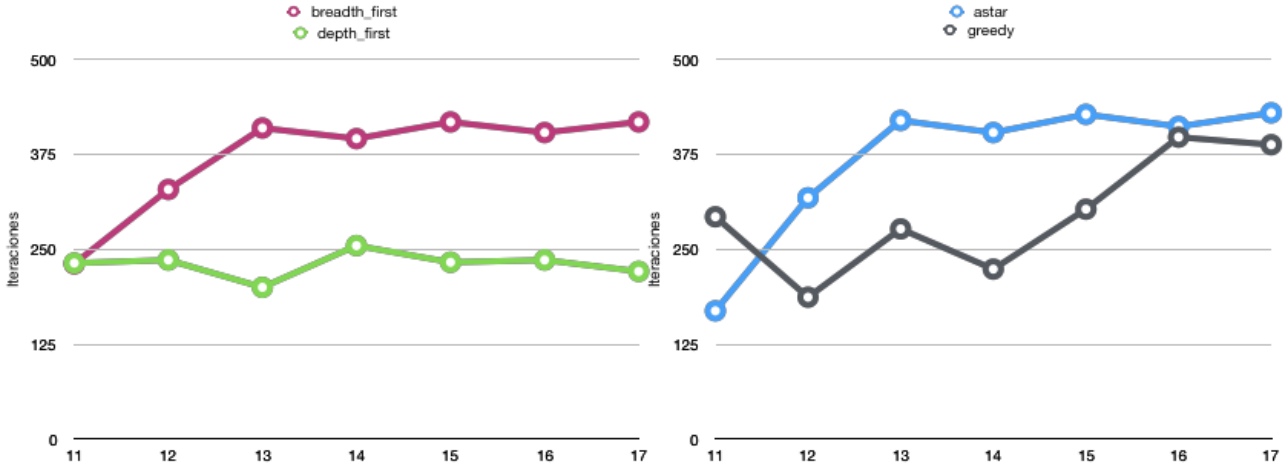

Figure 2:

Figura 2: Sin heurística. Coste de memoria respecto a mapa (Bloque de mapas 1 y 2)

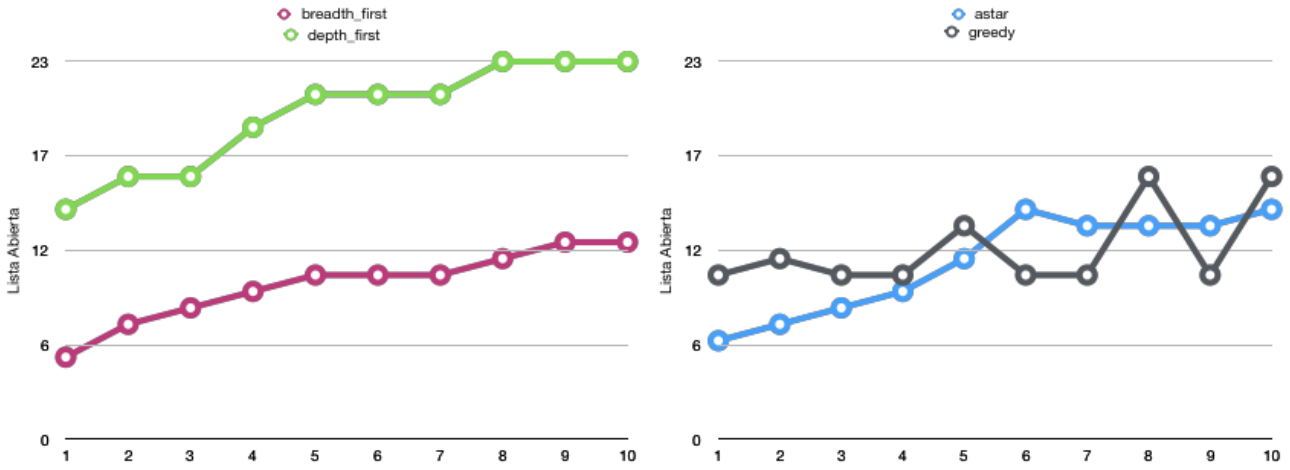

Figure 3:

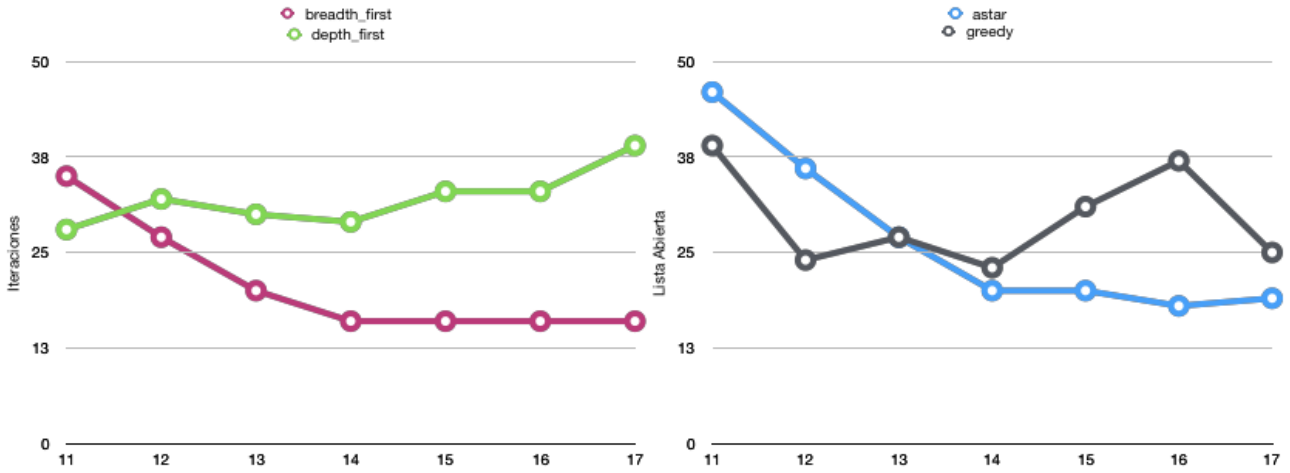

Figure 4:

Figura 3: Con heurística. Iteraciones respecto a mapa (Bloque de mapas 2)

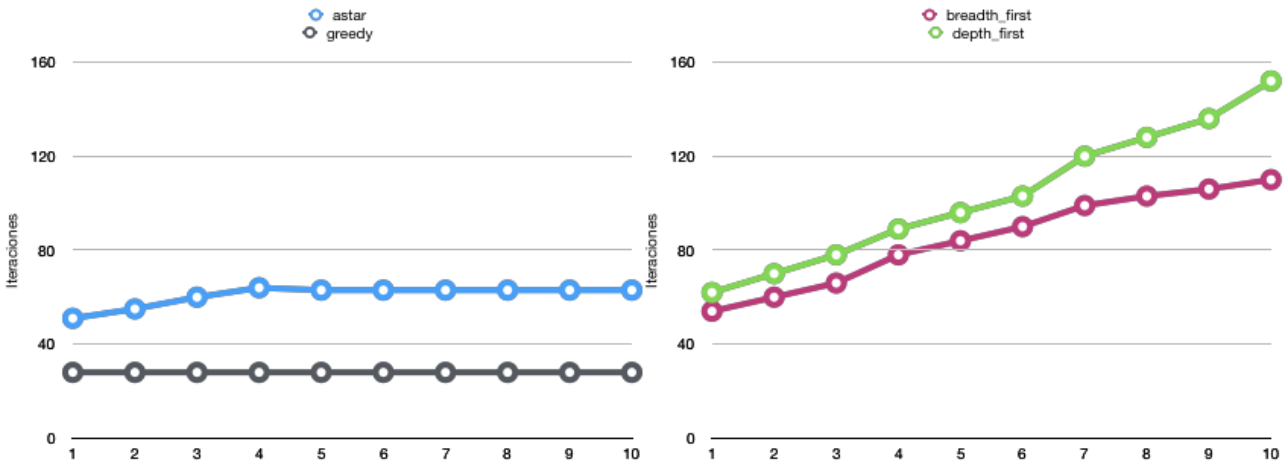

Figure 5:

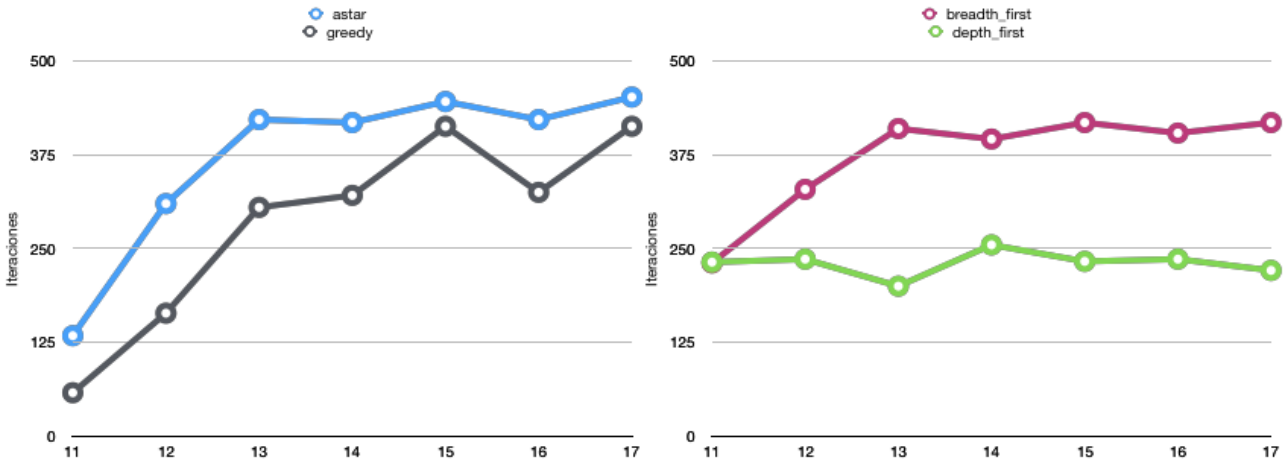

Figure 6:

Figura 4: Con heurística. Coste de memoria respecto a mapa (Bloque de mapas 2)

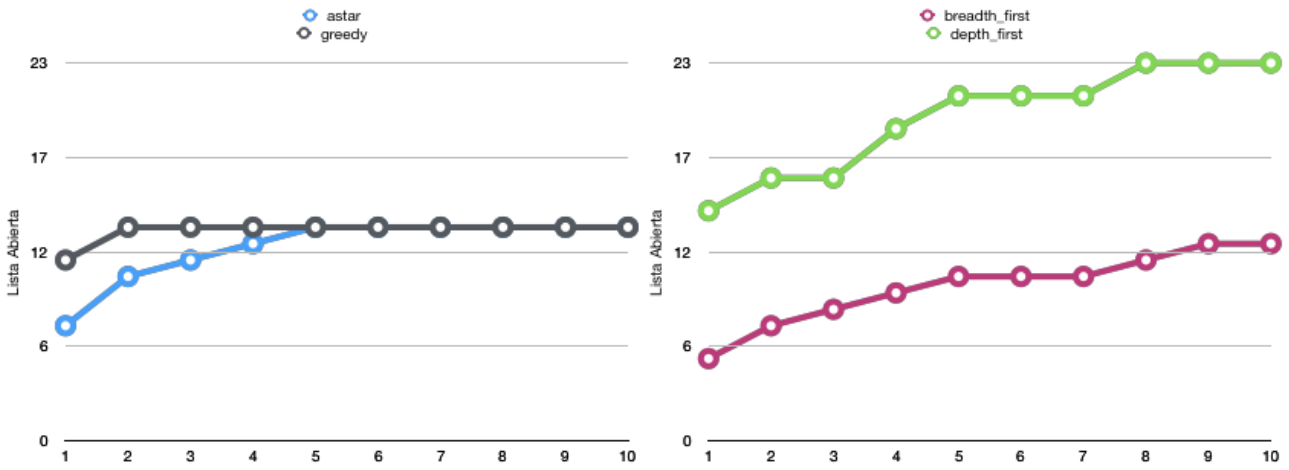

Figure 7:

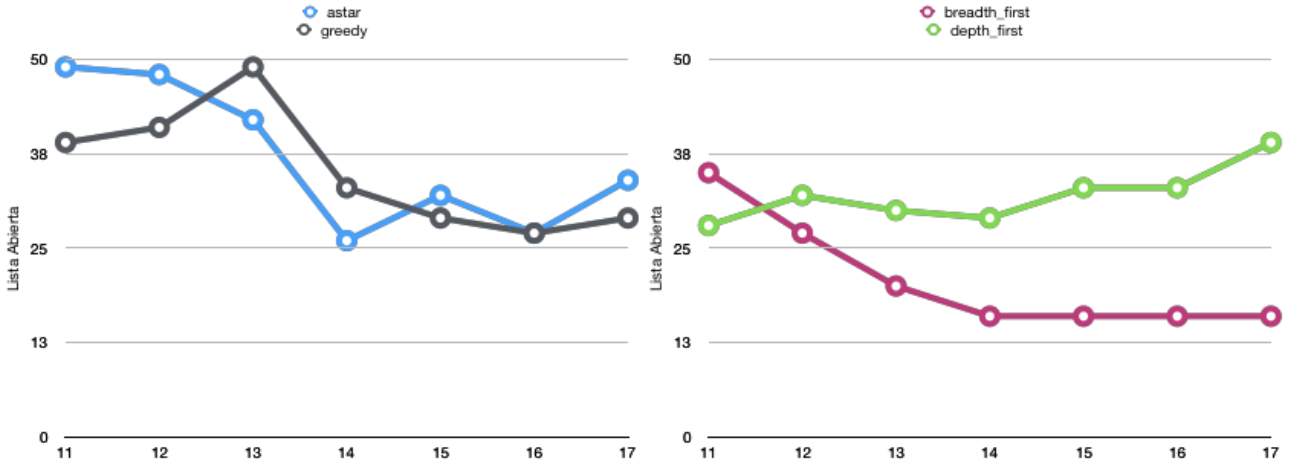

Figure 8:

En el caso de las pruebas sin heurística, el primer resultado que destaca es el de la búsqueda en profundidad, obteniendo una longitud y coste mayor que el resto de algoritmos. Además A*, Greedy y Búsqueda en amplitud tienen un coste de tiempo y de memoria similar, con una diferencia máxima entre los tres de 5 iteraciones y 3 nodos en la lista abierta (Tabla 3a); y 76 iteraciones 8 nodos en la lista abierta (Tabla 3b). A* es el más "perjudicado" con el segundo bloque de mapas quedándose detrás de Greedy y Búsqueda en amplitud en coste y longitud, sin embargo, realiza la búsqueda en menos iteraciones que estos dos. Se observa además que Greedy fluctua mucho respecto a la memoria e iteraciones, mientras que A*, profundidad y amplitud crecen de una forma lineal

En el caso de las pruebas con heurística, el primer resultado que de nuevo destaca es el de la búsqueda en profundidad, obteniendo una longitud y coste mayor que el resto de algoritmos y sin diferencias con las tablas 3a y 3b. Otro resultado que destaca es la reducción en las iteraciones (Tabla 4a) en A* y Greedy con respecto a los resultados de la prueba sin heurística. Además, podemos observar que, de nuevo, la existencia de una heurística beneficia al algoritmo A*, igualando a Greedy y Búsqueda en amplitud en coste y en longitud (Tabla 4b). Sin embargo, estas mejoras ocurren a costa de un aumento de tamaño máximo de la lista abierta. Se observa además que Greedy ya no fluctua tanto como en el caso de "Sin heurística" y se une a A*, profundidad y amplitud con su crecimiento, de nuevo, lineal.

A partir de estas gráficas

Por lo tanto, podemos obtener las siguientes conclusiones:

- El algoritmo de búsqueda en profundidad no recorre el árbol de nodos entero para buscar la mejor solución, por lo que realiza rutas demasiado lejanas, como hemos observado.

- El algoritmo de búsqueda en amplitud es el que menor coste de memoria tiene, a cambio de sacrificar tiempo de ejecución.
- El algoritmo Greedy es el que menor tiempo de ejecución tiene, a cambio de sacrificar memoria. Además, el uso de una heurística le favorece.
- El algoritmo A* se encuentra entre Greedy y búsqueda en amplitud por lo que podemos decir que es un algoritmo "híbrido" que obtiene un equilibrio entre tiempo y memoria.

¿Y un laberinto?

Este experimento no es muy riguroso, pero decidimos incluirlo por pura curiosidad. Con el queremos comprobar como se comportan los algoritmos frente a un laberinto sencillo, con una casa en el final del laberinto que pide 3 pizzas. Para generar este mapa hemos creado un *programa sencillo* en Python que convierte una imagen en blanco y negro a un mapa.

Nota: Si se va a ejecutar este mapa, recomendamos reducir `tile_size` en `config.py` a 15, debido al gran tamaño del mapa provisto.

Tabla 5: Con heurística. Resultados de map_test18.txt

Algoritmo	Longitud	Coste	Iteraciones/Nodos	Lista Abierta
greedy	1907	1907	12631	23
astar	1907	1907	19103	37
breadth_first	1907	1907	18889	35
depth_first	1907	1907	13328	50

Nuestro pensamiento inicial era que el programa tardaría mucho tiempo en realizar la búsqueda y se quedaría antes sin memoria, pero la velocidad de resolución y los resultados nos sorprendieron. Como podemos observar, la cantidad de memoria usada es relativamente pequeña.

Parte avanzada

¿Cuales son las mejores heurísticas para cada parámetro?

Con esta prueba, nuestra intención es comprobar el rendimiento de todas las heurísticas que hemos implementado, para obtener las "mejores" respecto a todos los parámetros; para posteriormente comparar su escalamiento en una siguiente experimentación. Queremos observar, además, si éstas cumplen la descripción que habíamos escrito para ellas. Vamos a utilizar el algoritmo de búsqueda A*.

Para realizar la prueba hemos diseñado distintos mapas que comprueban el rendimiento de cada una de las heurísticas. Además, cada mapa intenta probar distintas características:

- **Mapa 1**: Mapa sencillo. Usado para observar el patrón que sigue la heurística para realizar entregas.
- **Mapa 2**: Usado para comprobar el comportamiento de la heurística con entregas lejanas y cercanas.
- **Mapa 3**: Usado para comprobar las heurísticas con respecto a distintos terrenos con distinto coste y en zonas separadas por "ciudad" y "campo". Los carriles bici cuestan menos que las calle normales y cruzar el puente de madera cuesta menos que cruzar el bache antes del puente de asfalto (cuyo coste es inferior), pero este puente de madera está más lejos.
- **Mapa 4**: Similar al mapa 3. Usado para comprobar las heurísticas con respecto a distintos terrenos y distintos costes y con entregas a la misma distancia de la pizzería.
- **Mapa 5**: Usado para comprobar las heurísticas en cuestas, por ejemplo, si decide subir una pizza cada vez o subirlas de golpe.
- **Mapa 6**: Similar al mapa 5. Usado para comprobar las heurísticas en cuestas, esta vez con distintos pedidos en distintas casillas.
- **Mapa 7**: Usado para comprobar las heurísticas con motocicleta eléctrica y con una única entrega
- **Mapa 8**: Similar al mapa 7. Usado para comprobar las heurísticas con motocicleta eléctrica, distintos costes y con más de una entrega.
- **Mapa 9**: Usado para comprobar las heurísticas cuando los clientes están organizados alrededor de una "zona" cubierta por una pizzería.

- **Mapa 10**: Usado para comprobar las heurísticas cuando los clientes están muy dispersos a lo largo del mapa sin "zonas" concretas.
- **Mapa 11**: Similar al mapa 7. Usado para comprobar las heurísticas con motocicleta eléctrica.

Desgraciadamente no hemos podido realizar pruebas con mapas que mezclen cuestas y motocicleta eléctrica debido a que la memoria disponible para la ejecución era insuficiente.

Creemos que la heurística 6 va a dar un buen resultado, ya que esta forma de repartir, por zonas alrededor de una tienda, es la más usada en la vida real.

Como son demasiados mapas de prueba y heurísticas, hemos creado un pequeño script en bash para automatizar el cambio de heurísticas, de mapas y de guardado de los resultados de las pruebas. El tiempo está obtenido mediante la utilidad `time` en GNU/Linux. Los scripts de esta prueba y la siguiente están disponibles en el *siguiente enlace*.

Al ser muchos datos, vamos a presentar solamente las medias de cada una de las heurísticas. Los resultados completos se pueden encontrar en el *siguiente archivo de Excel*.

Tabla 6a: Media aritmética y veces que han obtenido la longitud y coste mínimos por mapa.

- V_l: Veces que obtiene la longitud mínima (respecto a las otras heurísticas)
- V_c: Veces que obtiene el coste mínimo
- V_i: Veces que obtiene el mínimo de iteraciones
- V_{la}: Veces que obtiene el mínimo tamaño de lista abierta
- V_t: Veces que obtiene el tiempo real de ejecución mínimo

Heurística	Longitud	V_l	Coste	V_c	Iter./Nodos	V_i	Lista Abierta	V_{la}	Tiempo	V_t
1	66	8	184,98	7	7715	0	429	8	31s 903ms	3
2	67	4	186,57	3	7614	0	455	2	32s 763ms	0
3	67	6	188,94	5	7111	0	499	3	34s 700ms	2
4	67	5	187,32	5	7435	0	470	0	33s 474ms	1
5	67	6	189,04	4	7187	0	491	2	34s 703ms	0
6	68	4	190,32	4	7438	0	496	0	36s 170ms	1
7	70	3	193,00	3	6556	7	567	0	32s 335ms	0
8	69	4	190,50	3	7283	1	530	0	36s 121ms	0
9	68	4	190,32	4	7412	0	505	0	35s 201ms	1
10	67	4	188,14	4	7195	0	504	1	32s 872ms	1
11	69	5	193,20	4	6622	3	545	0	30s 629ms	4

Tabla 6b: Mejores 3 por veces que han obtenido el mínimo en longitud

Heurística	Longitud	V_l	Coste	Iteraciones/Nodos	Lista Abierta	Tiempo
1	66	8	184,98	7715	429	31s 903ms
3	67	6	188,94	7111	499	34s 700ms
5	67	6	189,04	7187	491	34s 703ms

Tabla 6c: Mejores 3 por veces que han obtenido el mínimo en coste

Heurística	Longitud	Coste	V_c	Iteraciones/Nodos	Lista Abierta	Tiempo
1	66	184,98	7	7715	429	31s 903ms
4	67	187,32	5	7435	470	33s 474ms
3	67	188,94	5	7111	499	34s 700ms

Tabla 6d: Mejores 3 por veces que han obtenido el mínimo en iteraciones

Heurística	Longitud	Coste	Iteraciones/Nodos	V_i	Lista Abierta	Tiempo
7	70	193,00	6556	7	567	32s 335ms
11	69	193,20	6622	3	545	30s 629ms
8	69	190,50	7283	1	530	36s 121ms

Tabla 6e: Mejores 3 por veces que han obtenido el mínimo en tamaño de lista abierta

Heurística	Longitud	Coste	Iteraciones/Nodos	Lista Abierta	V_{la}	Tiempo
1	66	184,98	7715	429	8	31s 903ms
3	67	188,94	7111	499	3	34s 700ms
2	67	186,57	7614	455	2	32s 763ms

Tabla 6f: Mejores 3 por veces que han obtenido el mínimo en tiempo real de ejecución

Heurística	Longitud	Coste	Iteraciones/Nodos	Lista Abierta	Tiempo	V_t
11	69	193,20	6622	545	30s 629ms	4
1	66	184,98	7715	429	31s 903ms	3
3	67	188,94	7111	499	34s 700ms	2

A lo largo de las pruebas hemos observado que todas las heurísticas cumplen, la descripción que habíamos realizado sobre ellas, aproximadamente. En el caso de los mapas con motocicleta eléctrica, todas las heurísticas intentan recargar la batería al máximo posible antes de continuar, y en mapas con cuestas, intentan realizar caminos lo menor elevados posible, para reducir el coste. También hemos observado que las heurísticas que consiguen un menor coste, en el caso de las cuestas, intentan realizar más de una subida para reducir este valor.

Podemos observar que las heurísticas 1 y 3 se encuentran en cuatro de las cinco tablas (Tablas 6b, 6c, 6e y 6f), por lo que deducimos que estas encuentran un equilibrio entre longitud, coste, memoria y tiempo real de ejecución. Desgraciadamente, la heurística 6 no se encuentra en ninguna de las tablas, pero si dos variantes suyas, las heurísticas 7 y 8 en la tabla 6d que obtienen un mejor resultado en el número de iteraciones realizadas.

Sin embargo, encontramos una diferencia notable entre tiempo real de ejecución y de iteraciones. Esto es debido a que dependiendo de la heurística, unas realizan menores cálculos en procesador porque no dependen de otras funciones (como `distanceToNearestCharger(customer)` o `distanceToNearestShop(customer)`) mientras que otras si lo hacen, aumentando ligeramente el tiempo real de ejecución.

Finalmente obtenemos la conclusión de que, respecto al coste, la longitud, el coste de memoria y el tiempo real de ejecución de la solución, es mejor realizar primero las entregas más lejanas que ordenar por zonas o entregar las más cercanas o mantener siempre la bolsa llena y repartir. Además, nuestra hipótesis sobre la entrega por "zonas" cercanas a una tienda no obtiene buenos resultados y no se puede aplicar bien a este tipo de problema, en el que las entregas no dependen del tiempo y pueden entregarse en cualquier orden.

Como las heurísticas 1, 3 y 11 son las que más aparecen en estas cinco tablas, serán las que usaremos en la siguiente experimentación.

¿Como escalan las mejores heurísticas cuando el tamaño del problema aumenta?

Con esta prueba queremos observar como escala nuestra implementación cuando la dificultad del problema aumenta. Para ello hemos generado 47 mapas aleatorios distintos, comenzando por un mapa de 3x3 con 1 pizzería y 1 cliente y aumentando en cada iteración el tamaño del mapa en uno, los clientes en 2 y las pizzerías en 1 cada dos iteraciones:

Tabla 7: Características de los mapas

Mapa	Tamaño	Casas	Entregas	Pizzerías
3	3x3	1	3	1
4	4x4	2	2	1
5	5x5	3	6	2
6	6x6	5	10	2
7	7x7	7	13	3

Al igual que la experimentación anterior, como son demasiados mapas de prueba y heurísticas, hemos creado un pequeño script en bash para automatizar la recolección de resultados y los cambios de mapas y heurísticas. El tiempo está obtenido mediante la utilidad `time` en GNU/Linux. Todas las pruebas han sido realizadas, de nuevo, con el mismo algoritmo, A*.

Nota: Debido a las características del sistema en el que se han realizado, a partir del mapa 7, la memoria disponible era insuficiente.

Tabla 8a: Resultados de la heurística 1

Mapa	Longitud	Coste	Iteraciones/Nodos	Lista Abierta	Tiempo
3	21	41,00	72	11	430ms
4	18	37,00	80	16	230ms
5	54	127,00	1132	48	580ms
6	71	146,00	23158	1503	3m 40s 170ms
7					

Tabla 8b: Resultados de la heurística 3

Mapa	Longitud	Coste	Iteraciones/Nodos	Lista Abierta	Tiempo
3	21	41,00	72	10	260ms
4	18	37,00	65	20	240ms
5	54	127,00	1073	54	540ms
6	71	146,00	21828	1769	4m 19s 480ms
7					

Tabla 8c: Resultados de la heurística 11

Mapa	Longitud	Coste	Iteraciones/Nodos	Lista Abierta	Tiempo
3	21	41,00	71	10	210ms
4	18	37,00	56	22	230ms
5	54	127,00	1082	57	550ms
6	71	146,00	21099	1832	4m 22s 750ms
7					

Con estos resultados podemos observar que a partir del mapa 6, la diferencia en tiempo de ejecución y en uso de memoría es muy grande comparado con los mapas anteriores a éste. Por otro lado podemos ver que todas las heurísticas tienen un uso de memoria e iteraciones similar, sin diferencias significativas.

Figura 5: Iteraciones

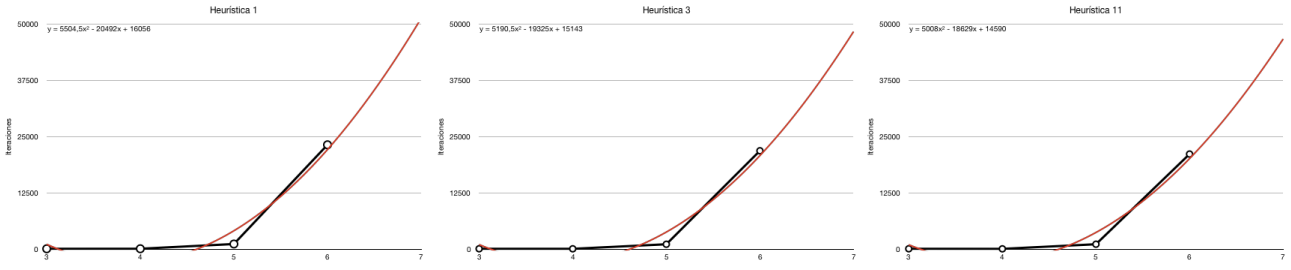

Figure 9:

Figura 6: Tamaño máximo de lista abierta

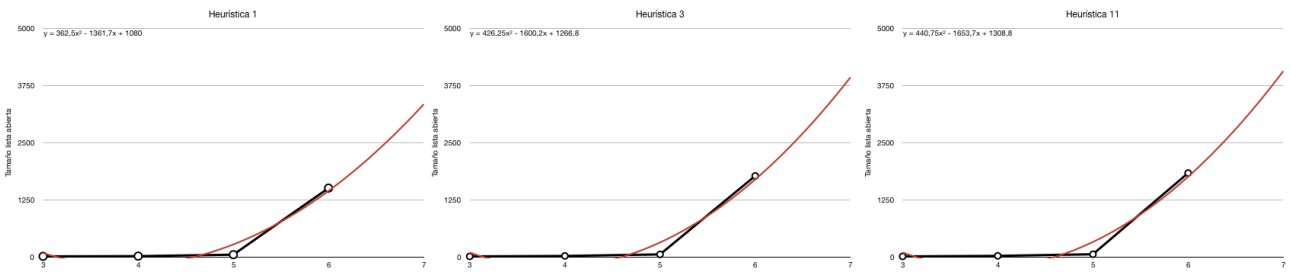

Figure 10:

Figura 7: Tiempos de ejecución

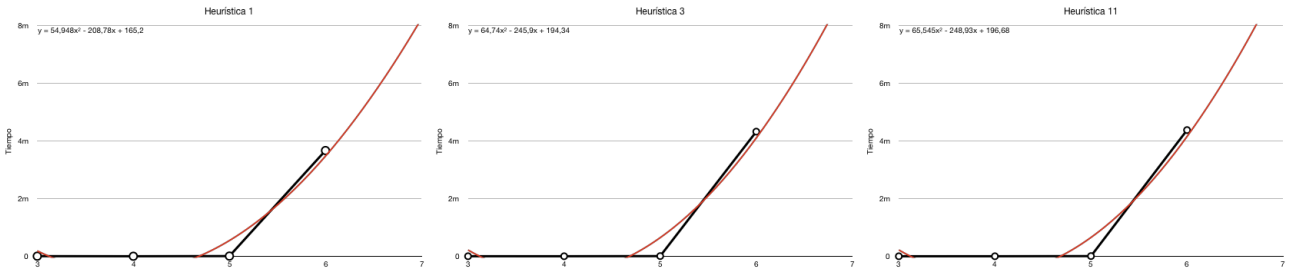

Figure 11:

Finalmente, y a pesar que debido a las limitaciones del sistema hemos obtenido datos limitados, si graficamos el coste de tiempo y de memoria, observamos que todas las heurísticas crecen de una forma polinómica, a partir de una función de segundo grado.

Con estos resultados podemos concluir que a medida que el problema crece en complejidad, el tiempo y el uso de memoria crecen de forma muy rápida. Así observamos que, en efecto, el problema de búsqueda es un problema de tipo NP.

Laberinto: Parte básica vs Parte avanzada

De nuevo, esta prueba está realizada por curiosidad. Como hemos observado en las experimentaciones anteriores y comparando con ambas partes, cuantos más elementos se añaden al problema, más compleja es la búsqueda de la solución. Por ello, la parte básica, al tener únicamente un cliente, la búsqueda de la solución es más rápida que, por ejemplo, en el mapa 7 de la parte avanzada donde intervienen terrenos de distinto coste y más de un cliente. Igualmente, a continuación queremos comparar ambas partes en los mismos mapas, en este caso, distintos laberintos. Los cinco laberintos tienen en su final, una casa con 3 pedidos. Para la parte avanzada vamos a usar la heurística 2 que es la más aproximada a la parte básica.

Nota: Si se van a ejecutar estos mapas, recomendamos reducir `tile_size` en `config.py` a 15, debido al gran tamaño del mapa provisto.

Tabla 9a: Resultados de la parte básica

Mapa	Longitud	Coste	Iteraciones/Nodos	Lista Abierta
10x10	485	485	1527	21
20x20	1319	1319	5354	68
30x30	1853	1853	11903	46
40x40	1843	1843	16864	65
50x50	2893	2893	26059	84

Tabla 9b: Resultados de la parte avanzada

Mapa	Longitud	Coste	Iteraciones/Nodos	Lista Abierta
10x10	485	1317	2108	11
20x20	1091	2986	7953	21
30x30	1853	5079	17475	28
40x40	1843	5054	27752	37
50x50	2893	7939	46282	34

Fácilmente observamos que la parte básica, por lo general, es más rápida al buscar la solución y necesita de menos iteraciones para llegar a ella, sin embargo el tamaño máximo de la lista abierta es mayor, haciendo un mayor uso de memoria, además de que la solución para el mapa de 20x20 es más larga que la solución de la parte avanzada que hace uso de menor memoria. Los costes no son comparables debido a que las definiciones de costes son distintas para ambas partes.

De este modo obtenemos la conclusión de que, la parte básica, al ser más simple, encuentra una solución en menos iteraciones, sin embargo, la parte avanzada, al ser más compleja, encuentra una solución mucho más óptima, aunque tomando más tiempo.

Conclusiones

Finalmente concluimos que a mayor complejidad de los distintos elementos de la búsqueda, la solución es mucho más óptima a cambio de un mayor tiempo de ejecución, además de que priorizar las entregas a la casas más lejanas primero es mucho mejor que otras prioridades.

Respecto al aspecto técnico, esta práctica ha sido muy entretenida, es muy asequible y es fácil de realizar con las herramientas y el código inicial ya provistos, además de ser muy fácil la adición de nuevos elementos al problema sin mucha más complejidad. Sin embargo, han surgido varios problemas técnicos con PyGame en Python 2.7 en las últimas versiones del sistema operativo macOS y recomendaríamos actualizar el lenguaje a Python 3.