



Búsqueda A*

Axentes Intelixentes
Grao en Robótica



Jaime Rodríguez Rodríguez
Martín Sánchez López

Índice

Índice	2
1. Introducción	3
1.1 ¿Cómo funciona A*?	3
1.2 Características de A*	3
2. Búsqueda para un pallet	3
2.1 Representación de los estados	4
2.2 Operadores	5
2.3 Algoritmo de A*	7
2.4 Función heurística	7
2.4.1 Algoritmo de Dijkstra	7
2.4.2 Distancia pallets	8
2.4.3 Distancia de Hamming	8
2.5 Mejoras para un pallet...	8
2.6 Lanzar búsqueda y simulación	9
2.7 Análisis del algoritmo	9
3. Búsqueda para varios pallets	11
3.1 Propuesta de solución	11
3.2 Clase Estado	11
3.3 Operadores	11
3.4 Función heurística	12
3.4 Análisis del algoritmo	12
3.5 Lanzar búsqueda y simulación	13
4. Conclusiones	14

1.Introducción

El Algoritmo A* es una de las estrategias más conocidas, se clasifica dentro de los algoritmos de búsqueda con información y es mayormente utilizado para búsquedas sobre grafos. El algoritmo encuentra siempre y el camino de menor costo entre un nodo origen y uno objetivo siempre y cuando exista la misma.

En esta práctica usaremos este algoritmo para poder encontrar la solución de transportar pallets de un punto a otro de la manera más óptima.

1.1 ¿Cómo funciona A*?

El algoritmo va a construir diferentes rutas para trasladar el pallet de la pose de inicio (nodo inicial) a la pose final indicada (nodo meta o final). Pero el algoritmo solo va a construir aquellas que están mejor posicionadas para encontrar la solución, y para poder determinar esto va a usar la siguiente fórmula (búsqueda heurística):

$$F(n) = g(n) + h(n)$$

Donde:

- $F(n)$: coste total.
- $g(n)$: coste hasta el nodo actual.
- $h(n)$: coste heurístico que estima el algoritmo en llegar al nodo final.

El algoritmo va a expandir el primer nodo que se encuentra en la lista abierta (nodo inicial, en la primera expansión) y compara si es el nodo meta; si no es así calcula $F(n)$ para todos los nodos hijos, e introduce en orden (en función de $F(n)$ en abierta). El nodo expandido pasa a la lista cerrada. Y esto se repite hasta encontrar una solución.

1.2 Características de A*

- **Completo:** si existe una solución el algoritmo la encuentra.
- **Complejidad temporal:** depende de la función heurística implementada. Con una heurística pésima será exponencial, mientras que en el mejor caso será lineal.
- **Complejidad en memoria:** depende también de la función heurística. Es uno de sus mayores problemas, ya que tiene que guardar todos los posibles nodos de cada estado.
- **Óptimo:** es el algoritmo más eficiente. No hay otro algoritmo que permita encontrar una solución expandiendo menos nodos.

2.Búsqueda para un pallet

Para poder plantear una solución óptima al problema planteado empezamos resolviendo almacenes donde solo manejamos un pallet. Lo primero que tenemos que hacer es plantear: cómo vamos a representar el espacio de estados, operadores (con sus

precondiciones y efectos), costes de la aplicación de los diferentes operadores y función heurística.

2.1 Representación de los estados

Para poder representar los diferentes nodos creamos una clase llamada *Estado* en que cada instancia va a ser un estado diferente de la búsqueda. Dentro de la propia clase tenemos diferentes métodos que van a describir las diferentes características de cada estado (Fig. 1).

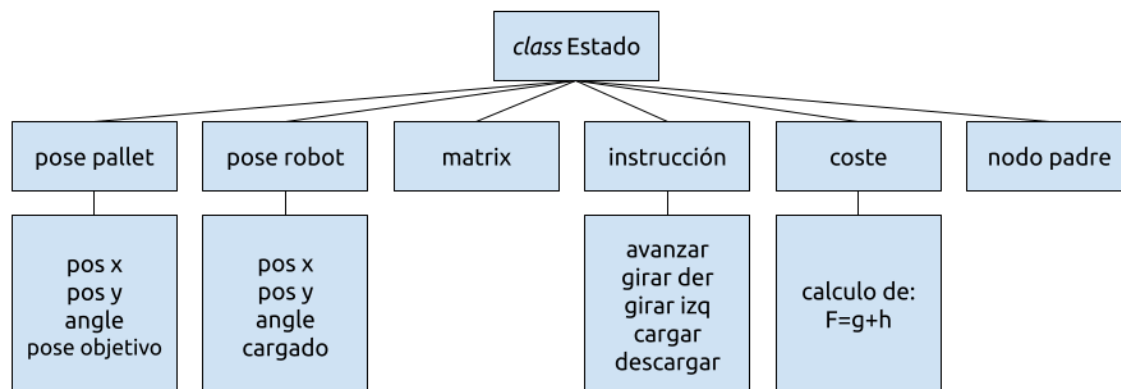


Fig. 1

- **Matrix:** matrix que representa el almacén. Donde tenemos los siguiente objetos representados: 9 (paredes), 3 (cubos, obstáculos), 0 (posiciones donde el robot puede avanzar) y 1 (robot). La matriz la iremos actualizando según la posición del robot.
- **Pose robot:** va a describir tanto la posición x e y como la orientación del robot. Vamos a calcular en función de la posición del uno en la matriz. La orientación es un argumento para crear el nodo. También tenemos una variable booleana que indica si está cargado o no.
- **Pose pallet:** en un primer inicio el pallet también estaba representado en la matriz con un 2. Pero esto dificulta bastante las acciones de carga y descarga (1 y 3 tienen que estar en la posición de la matriz). Por lo que sacamos esta información de la matriz y la añadimos como un elemento de la clase (*pose_objetivo_pallet*). La posición del pallet si está cargado es la misma que la del robot, pero si no está cargado la hereda del nodo padre. La orientación (V o H) es un argumento cuando se crea la instancia de la clase. Existe también un método que define la pose objetivo del pallet, en el nodo inicio inicial la iniciamos a través del paso de los valores como argumento de la instancia, en los siguientes nodos lo hereda del nodo padre.
- **Instrucción:** es una lista con un str dentro, que contiene la acción que tiene que llevar a cabo el robot para poder transitar desde el padre nodo hasta sí mismo. Una vez que tengamos el camino de solución encontrado, iremos recorriendo desde el

nodo meta hasta el nodo inicial y guardando las instrucciones que tiene que ir ejecutando.

- **Coste:** en este método calcularemos el valor de $F(n)$ para cada nodo expandido. El valor de $g(n)$ lo va a heredar del nodo padre y le sumaremos el coste para transitar al estado actual. A mayores $h(n)$ será calculado a través de la invocación de la función heurística.
- **Nodo padre:** lugar donde vamos a almacenar la referencia al nodo padre del estado actual. Una vez el algoritmo encuentre la solución iremos recorriendo hacia atrás la ruta de los nodos padres hasta el nodo inicial.

2.2 Operadores

Podemos definir los operadores de búsqueda como el conjunto de acciones que puede llevar a cabo el robot en el almacén. Durante el proceso de programación de nuestro algoritmo este fue uno de los puntos que más cambios tuvo. En un primer momento decidimos concatenar varias acciones en un solo operador, es decir, todos los operadores obligaban al robot a moverse de lugar. Estos fueron los operadores en las primeras versiones:

```
ins0=["self.nav.move(1)"]
ins1=["self.nav.rotateRight()", "self.nav.move(1)"]
ins2=["self.nav.rotateRight()", "self.nav.rotateRight()", "self.nav.move(1)"]
ins3=["self.nav.rotateLeft()", "self.nav.move(1)"]
ins4=["self.nav.downLift()"]
ins5=["self.nav.upLift()"]
```

Después de darnos cuenta de este error definimos de nuevo los operadores. En esta segunda versión no concatenamos acciones y las hacíamos de forma aislada cada acción (Fig.2).

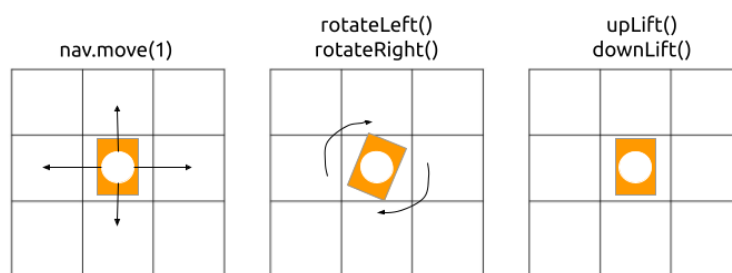


Fig. 2

- **move(1):** Esta será la única acción que permitirá al robot cambiar de posición. Tendremos que tener en cuenta la orientación del robot, para saber a qué posición podemos avanzar (Fig. 3). (No se pueden realizar movimientos diagonales, cruces de la Fig. 3).






	FILA -1 COL 0 ORI 0°	
FILA 0 COL -1 ORI 270°		FILA 0 COL +1 ORI 90°
	FILA +1 COL 0 ORI 180°	

Fig. 3

En la siguiente figura (Fig.3) podemos ver la posibles orientaciones del robot representadas en un mapa, así como a que posición de la matriz se va a mover después de producirse el avance, suponiendo que la posición actual es [FILA,COL].

	Orientación robot	Precondición	Efecto	Coste
move(1)	0°	[FILA-1,COL]==0 and ORI==0	[FILA-1,COL]==1 [FILA,COL]==0	1 (no carga) 2 (cargado)
	90°	[FILA,COL+1]==0 and ORI==90	[FILA,COL+1]==1 [FILA,COL]==0	1 (no carga) 2 (cargado)
	180°	[FILA+1,COL]==0 and ORI==180	[FILA+1,COL]==1 [FILA,COL]==0	1 (no carga) 2 (cargado)
	270°	[FILA,COL-1]==0 and ORI==270	[FILA,COL-1]==1 [FILA,COL]==0	1 (no carga) 2 (cargado)

En el nuevo estado si el pallet la variable booleana que indica si está cargado la hereda del nodo padre, así como la orientación del pallet. En caso de que esté cargado el robot la nueva posición del pallet se actualiza con la pose del robot (1 en la matriz). La orientación del robot se mantiene como la del nodo padre. En el apartado [2.5 Mejoras para un pallet](#) introduciremos nuevas precondiciones para cuando esté cargado.

- **rotateRight() rotateLeft():** acciones de girar el robot sin moverse de la posición donde está. No va a tener precondiciones siempre se va a poder realizar. En el apartado [2.5 Mejoras para un pallet](#) introduciremos nuevas precondiciones para cuando esté cargado.

	Nueva orientación robot	Nueva orientación pallet (Si está cargado)	Coste
rotateLeft()	(actual-90)%360	Si actual==H, update=V. Si actual==V, update=H	2 (no carga) 3 (cargado)
rotateRight()	(actual+90)%360	Si actual==H, update=V. Si actual==V, update=H	2 (no carga) 3(cargado)

No hay cambios en la posición del robot ni del pallet. En caso de que haya que girar 180°, el algoritmo encadenará dos acciones de giro consecutivas.

- **upLift() downLift();** acciones para cargar y descargar el pallet. Lo único que cambiará será de valor la variable booleana *cargado*. En un principio el robot sólo le dábamos al robot la opción de descargar en la *pose_finall_pallet*, en versiones posteriores nos dimos cuenta que el robot necesitaba hacer descarga, colocarse bien (girar) y volver a cargar para encontrar una solución óptima.

	Precondición	Efecto	Coste
downLift()	Si cargado==True	cargado=False	3
upLift()	pose_robot tiene que ser la misma que pose_pallet, y el pallet no puede estar en la pose objetivo.	cargado=True	3

2.3 Algoritmo de A*

Implementamos los dos pseudocódigos dados en el enunciado de la práctica. Uno de ellos consume más espacio de memoria ya que añade todos los nodos expandidos en abierta (tipo 1), pero menos tiempo ya que no realiza tantas comprobaciones. El otro (tipo 2) consume más tiempo debido a que realiza muchas más comprobaciones antes de añadir en abierta o en cerrada. En el punto [2.7 Análisis del algoritmo](#) evaluaremos cada una de las implementaciones.

2.4 Función heurística

Una función heurística es usada en un algoritmo de búsqueda informada para estimar el coste de alcanzar un determinado estado (nodo meta). La función heurística es el componente más importante en una búsqueda A*, y lo más importante es que sea admisible:

Una función heurística será admisible siempre y cuando el coste estimado sea menor o igual al coste real de alcanzar el estado objetivo.

A lo largo de la práctica probamos diferentes funciones heurísticas: distancia euclídea, distancia de Manhattan, ... En el punto [2.7 Análisis del algoritmo](#) evaluaremos cuál funciona mejor.

2.4.1 Algoritmo de Dijkstra

Es un caso especial de la búsqueda A*, en el cuál el valor de la función heurística siempre es 0. La idea es explorar todos los caminos más cortos desde el nodo origen hasta el nodo meta.

2.4.2 Distancia pallets

Para el algoritmo de un pallet vamos a dividir el cálculo de la estimación del coste en varias etapas, es decir, vamos a colocar en el espacio un conjunto de puntos y calcularemos la distancia entre ellos. En caso de que una etapa ya esté realizada la saltaremos; por ejemplo si el robot ya tiene cargado el pallet la distancia entre la *pose_robot* y *pose_pallet* (inicio) no hará falta calcularla.

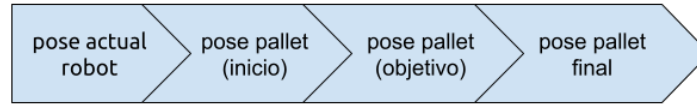


Fig. 4

En cuanto a la forma de calcular la distancia tendremos dos formas:

- **Distancia euclidiana:** distancia mínima entre dos puntos del espacio, a través del segmento que los une.

$$d_E(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- **Distancia de Manhattan:** es la suma de las diferencias absolutas de sus coordenadas. Es decir, es la suma de las longitudes de los dos catetos del triángulo rectángulo.

$$d_1(\mathbf{p}, \mathbf{q}) = \|\mathbf{p} - \mathbf{q}\|_1 = \sum_{i=1}^n |p_i - q_i|,$$

Podemos ver un ejemplo de la diferencia entre la distancia euclidiana (azul) y la distancia de Manhattan (roja). En un problema como al que nos enfrentamos la distancia de Manhattan será mucho más exacta (Fig. 5).

Además para ambas distancias, la distancia que el robot tiene que hacer cargado la multiplicaremos por dos y sumaremos seis (mínimo una carga y una descarga). Y por último si el robot no está en la misma columna o en la misma fila que donde tiene que dejar el pallet sumaremos dos (mínimo un giro).



Fig. 5

2.4.3 Distancia de Hamming

Diferencia entre pallets bien colocados y los que aún no fueron colocados. Aunque esta heurística tiene más sentido para cuando tratemos varios pallets aquí también la evaluaremos.

2.5 Mejoras para un pallet...

Después de comprobar que la implementación del algoritmo funcionaba correctamente, solamente nos faltaba una última mejora con un pallet: el pallet ocupa tres casillas y teníamos que tenerlo en cuenta a la hora de avanzar y de girar. La clave del problema no era que pudiese chocar con las paredes sino con los cubos. Esto solo lo tendremos que tener en cuenta cuando el robot esté cargado. Aquí introduciremos un nuevo elemento en la matriz que va a representar a los cubos, el 3. También tendremos que incorporar nuevas

precondiciones en los operadores ya que cuando el robot está cargado ocupa más de una casilla de la matriz. Por ejemplo para la Fig. 6:

- **move(1):** a mayores de mirar solo la que está delante del robot, tendremos que mirar que la de los lados a donde va a avanzar el robot también estén libres, si el pallet está en horizontal. Si el pallet estuviese en vertical tendríamos que mirar la casilla a donde va a avanzar el robot y una más.
- **rotateLeft() rotateRight():** como podemos ver en el dibujo, al girar tenemos que mirar varias casillas durante el proceso de girar, como también en la posición final.

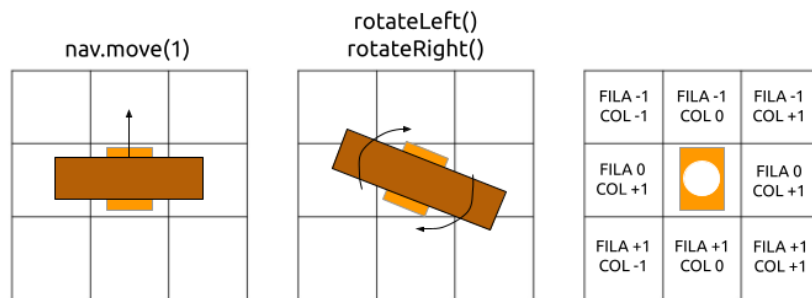


Fig. 6

2.6 Lanzar búsqueda y simulación

Lanzar script Python donde se lleva a cabo búsqueda:

```
>>> python un_pallet.py 5,2 V 1,5 H
```

Los argumentos que acompañan al archivo script son: 5,2 (posición pallet inicio); V (orientación inicial pallet); 1,5 (pose final pallet); H (orientación final pallet). Estos se pueden cambiar tal como quiera el usuario.

Una vez encontrada la solución el script genera tres archivos txt:

- **matriz.txt:** archivo donde podremos observar cómo evoluciona la matriz desde el nodo inicial hasta el nodo final, así como las instrucciones que lleva a cabo para transitar.
- **plan.txt:** archivo donde se guardan las instrucciones que realiza el robot.
- **statistics.txt:** archivo donde se muestran aspectos estadísticos como: nodos expandidos, tiempo de búsqueda, tiempo de ejecución en Gazebo...

Si se quiere cambiar la pose donde empieza y donde va a acabar el robot en el algoritmo de búsqueda, solamente hay que cambiar el uno en la matriz, en la función `estado_inicial_final()` de la clase `Practica1`.

2.7 Análisis del algoritmo

Vamos a analizar las diferentes funciones heurísticas planteadas para observar cuál funciona mejor (eficiencia). Para poder compararlas tendremos en cuenta los siguientes valores:

- **Coste:** coste que tendría para el robot transitar del estado inicial al estado final. El coste debería de ser igual independientemente de la heurística usada (solución óptima).
- **Tiempo que tarda en encontrar solución:** tiempo que tarda el algoritmo A* en encontrar una solución óptima.
- **Nodos expandidos:** nodos que A* le hace falta para poder encontrar solución.
- **Longitud del plan:** número de nodos por los que tiene que transitar el robot para llegar al estado meta desde el estado inicial.

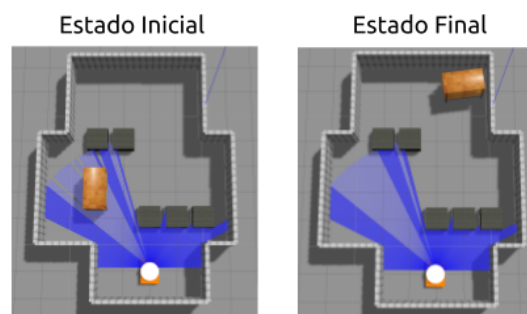


Fig. 7

Para la evaluación establecemos el nodo inicial y final de la Fig.7.

	Heurística	Coste	Tiempo búsqueda A*	Nodos expandidos	Longitud del plan
Algoritmo A* TIPO 1	Dijkstra	68	2.22 min	6326	38
	Distancia euclidea	68	1.78 min	5605	38
	Distancia manhattan	68	1.55 min	5470	38
	Distancia Hamming	68	2,19 min	6287	38
Algoritmo A* TIPO 2	Dijkstra	68	3.26 min	6325	38
	Distancia euclidea	68	2.59 min	5602	38
	Distancia manhattan	68	2.33 min	5466	38
	Distancia Hamming	68	3.13 min	6284	38

En todas las pruebas que hicimos y que realmente existía una solución el algoritmo siempre fue capaz de encontrarla.

Si estamos ante un almacén que estamos seguros que el robot no hace falta que descargue por el medio del trayecto para esquivar obstáculos, podemos añadir una

precondición más al operador de descarga, y es que descargue solamente si se encuentra en la pose final del pallet. Esto reduce el tiempo de búsqueda a un par de segundos.

3. Búsqueda para varios pallets

Para poder solucionar el almacén pero con más de un pallet partiremos del código para un pallet (*un_pallet.py*). La base de la solución va a ser muy parecida, tendremos que añadir para que pueda manejar más de un pallet a la vez.

3.1 Propuesta de solución

La gran diferencia entre estar en un almacén con un pallet o varios pallets sería la forma de tratar a estos. En nuestra solución planteamos dos listas de listas donde cada lista representa un pallet diferente:

1. **pactual:** lista de listas donde guardamos la posición actual de cada pallet. En el nodo inicial en esta lista en donde vamos a poner donde se encuentran los pallets.
2. **pallet_objetivo:** lista de listas donde cada lista representa la posición final donde hay que dejar cada pallet.

Cada pallet tiene la posición en las listas, es decir, el pallet de *pactual* que ocupa la posición 2, tiene que colocarse en la pose que indica la lista de la posición 2 de *pallet_objetivo*.

3.2 Clase Estado

Los cambios que tenemos que hacer en la clase Estado para que funcione con lista de listas.

- Ahora en vez de tener una variable *booleana* que indique si el robot está cargado o no, tendremos una variable que tendrá guardada la posición del pallet que ocupa en las listas y en caso de no estar cargado será *None*. Para comprobar si está cargado o no tuvimos que hacerlo a través de *type()*, ya que si estaba cargado con el pallet de la posición 0, y hacíamos un *if pcargado*: esto nos daba que *pcargado* es *False*, ya que 0 es *False*.
- La pose del robot se calculará de la misma forma, y la de pallet también; excepto por que solo se actualizará aquel pallet que esté cargado.

3.3 Operadores

Los operadores seguirán siendo los mismos, pero ahora tenemos un nuevo obstáculo en el almacén: otros pallets.

- **move(1):** si el robot está cargado tiene que comprobar que a donde va a avanzar no haya un pallet. Si no está cargado nos da igual por que pasa por debajo.
- **rotateRight() rotateLeft():** no hay cambios. En caso de estar cargado solo tenemos que modificar de orientación el pallet que tenemos cargado.
- **downLift():** no se introducen modificaciones.
- **upLift():** hay que identificar qué pallet se está levantando.

Un error que tuvimos al actualizar los operadores fue que no creamos una copia de las listas de los pallets, y estábamos siempre actualizando la misma dirección de memoria. Para solucionar esto usamos la librería *copy*, que nos permitió crear una variable nueva independiente de la que copiamos.

3.4 Función heurística

Igual que para cuando solo trabajamos con un solo pallet, aquí también implementamos varias funciones heurísticas; así como los dos tipos de búsqueda A* descritas en el enunciado.

- **Algoritmo de Dijkstra:** función heurística siempre con valor 0.
- **Distancia pallets:** sumamos la distancia que tenemos que mover los pallets desde su posición inicial a la posición final. La distancia la vamos a medir usando la distancia euclídea como la distancia de Manhattan.
- **Distancia Hamming:** Diferencia entre pallets bien colocados y los que aún no fueron colocados.

3.4 Análisis del algoritmo

En almacenes con varios pallets se usaron las mismas funciones heurísticas que para un pallet.

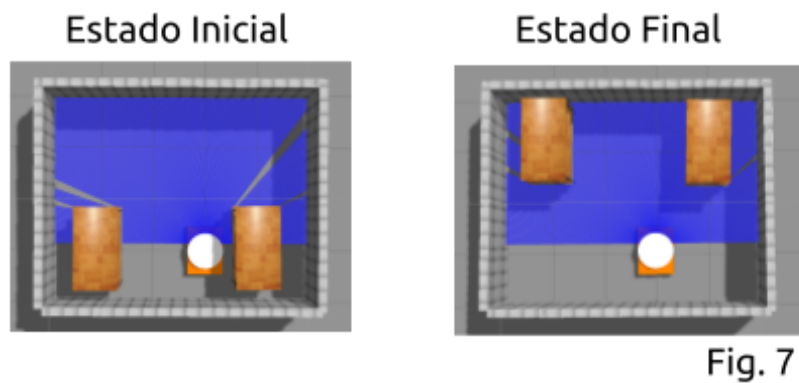


Fig. 7

	Heurística	Coste	Tiempo búsqueda A*	Nodos expandidos	Longitud del plan
Algoritmo A* TIPO 1	Dijkstra	50	5.45 min	23463	28
	Distancia euclídea	50	2.93 min	17117	28
	Distancia manhattan	50	1.45 min	15922	28
	Distancia Hamming	50	3.55 min	22104	28
Algoritmo A* TIPO 2	Dijkstra	50	7.21 min	23096	28
	Distancia euclídea	50	4.90 min	19965	28
	Distancia manhattan	50	4.55 min	19243	28
	Distancia Hamming	50	4.51 min	21752	28

Si estamos ante un almacén que estamos seguros que el robot no hace falta que descargue por el medio del trayecto para esquivar obstáculos, podemos añadir una precondition más al operador de descarga, y es que descargue solamente si se encuentra en la pose final del pallet. Esto reduce el tiempo de búsqueda a un par de segundos, pero expande menos nodos.

El algoritmo para varios pallets, fue también probado y funciona para un solo pallet.

Hay que tener en cuenta que cuando más grande sea el almacén más estados y más nodos expandirá y esto aumentará de forma exponencial.

3.5 Lanzar búsqueda y simulación

Lanzar script Python donde se lleva a cabo búsqueda:

```
>>> python varios_pallets.py 3,1 V 1,1 V 3,4 V 1,4 V
```

Los argumentos que acompañan al archivo script son: 3,1 (posición primer pallet inicio); V (orientación inicial del primer pallet); 1,1 (pose final primer pallet); H (orientación final primer pallet); los siguientes datos son los mismos para el segundo pallet. Estos se pueden cambiar tal como quiera el usuario.

Una vez encontrada la solución el script genera tres archivos txt:

- **matriz.txt:** archivo donde podremos observar cómo evoluciona la matriz desde el nodo inicial hasta el nodo final, así como las instrucciones que lleva a cabo para transitar.
- **plan.txt:** archivo donde se guardan las instrucciones que realiza el robot.
- **statistics.txt:** archivo donde se muestran aspectos estadísticos como: nodos expandidos, tiempo de búsqueda, tiempo de ejecución en Gazebo...

Si se quiere cambiar la pose donde empiece y donde va a acabar el robot solamente hay que cambiar el uno en la matriz, en la función *estado_inicial_final()* de la clase *Practica1*.

4. Conclusiones

La eficiencia de la búsqueda A* está muy ligada a la eficiencia de la heurística usada. Por lo que hemos visto a lo largo de esta memoria es que hay diferentes grados de eficiencia en la función heurística. La clave para encontrar una buena heurística es encontrar el equilibrio entre la sencillez y lo informada que esté.

Después del análisis hecho en los puntos anteriores podemos decir que el mejor algoritmo de A* es el tipo 1 donde se realizan muchas menos comprobaciones a la hora de añadir en abierta pero consume más memoria. También podemos concluir que la mejor función heurística es la explicada en el [punto 2.4.2](#) usando la distancia de Manhattan, ya que calcula los costes de forma mucho más real (no movimientos diagonales).

También podemos concluir que el código implementado puede ser escalable a almacenes más grandes, y que funciona razonablemente con el algoritmo de Dijkstra para tener una complejidad exponencial.