# PECL3: Integrando PP y PDDL

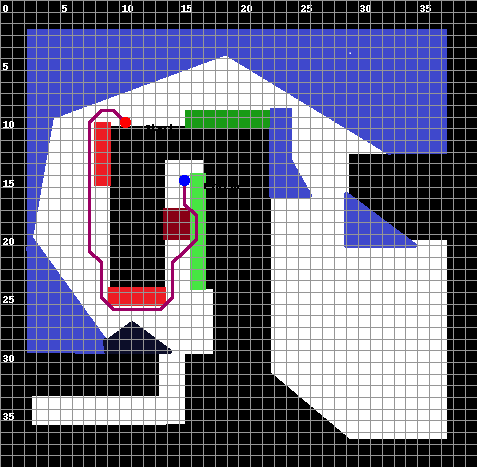
## Parte 1: Path-Planning

### Ejecución de Dijkstra

Comenzamos con la parte de path-planning ejecutando el algoritmo de Dijkstra sobre el mapa test\_2.png. El punto de inicio es el 10,10 y la meta es el 15,15. Ejecutamos el siguiente comando:

*python3 run\_path\_planning.py --scenario ../res/test\_2.png --start "(10,10)" --finish "(15,15)" --grid\_size 40 --algorithm Dijkstra --heuristic naive*

El resultado obtenido es el siguiente:



En la imagen se puede ver el recorrido realizado, siendo el punto rojo el punto de inicio y el punto azul el punto final. Es también interesante observar que cuando ejecutamos el comando por consola nos aparece una información muy relevante para poder comparar posteriormente los algoritmos y heurísticas:

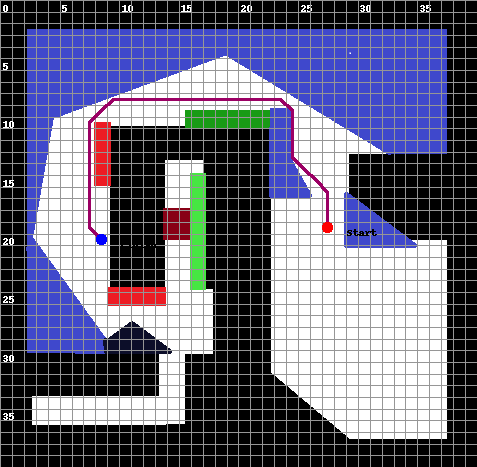


Esta imagen nos indica que el coste del camino conseguido es de 49, siendo su longitud de 35. Para poder lograr el camino, ha sido necesario expandir 682 nodos. Esta última métrica nos será muy interesante para comparar las heurísticas ya que según el número de nodos expandidos podremos saber cómo de buena es una heurística.

### Ejecución de Dijkstra en otro punto

Ejecutamos el mismo comando que en el anterior punto, pero cambiamos las coordenadas de inicio y fin. El punto de inicio es el (27,19) y el punto meta es el (8,20). Ejecutamos el siguiente comando:

*python3 run\_path\_planning.py --scenario ../res/test\_2.png --start "(27,19)" --finish "(8,20)" --grid\_size 40 --algorithm Dijkstra --heuristic naive*

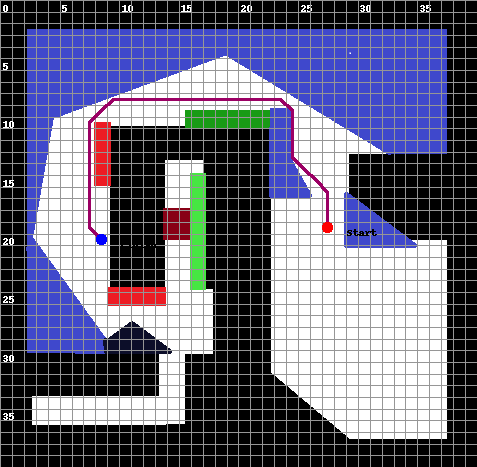


Se puede ver cómo se traza el camino desde el inicio hasta el final con el algoritmo de Dijkstra evitando obstáculos.

### Ejecución de A\*

En este punto, dado que tenemos el algoritmo A\* implementado también. Vamos a hacer una pequeña comparación entre el desempeño de A\* y Dijkstra en el mismo escenario que teníamos en el punto 2. Para ejecutar A\* con la heurística naive lo hacemos con el siguiente comando:

*python3 run\_path\_planning.py --scenario ../res/test\_2.png --start "(27,19)" --finish "(8,20)" --grid\_size 40 --algorithm A\* --heuristic naive*



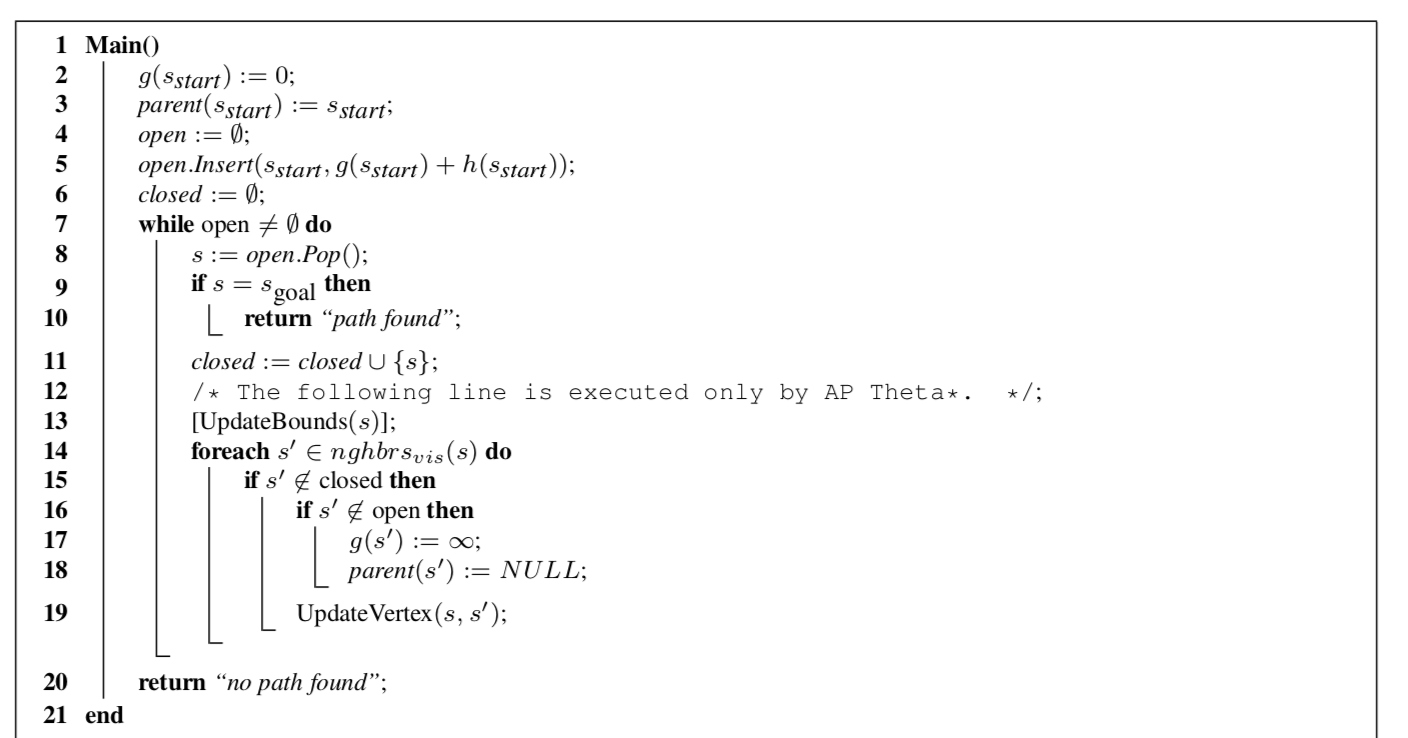
Como se puede apreciar, el camino logrado por A\* es prácticamente el mismo que Dijkstra. De hecho, si vamos más allá y nos fijamos en las métricas presentadas por consola, nos percatamos tanto que Dijkstra como A\* tienen el mismo rendimiento, es decir, mismo camino de misma longitud y coste; y mismo número de nodos expandidos. Es importante recalcar que A\* puede hacer uso de las heurísticas para reducir considerablemente el número de nodos expandidos; pero dado que usamos la heurística naive (que es constante y no proporciona información) es como si estuviéramos haciendo una búsqueda a ciegas.

### Implementación de A\*

En este punto vamos a realizar la implementación de el algoritmo Theta\*. Este algoritmo funciona prácticamente igual que A\* pero radica en él una pequeña diferencia que lo hace diferente: el cálculo de la línea de visibilidad.

Si nos fijamos en los caminos que logra conseguir A\* vemos que los ángulos de giro de la trayectoria están restringidos en términos de 45º. Con Theta\* y el cálculo de la línea de visibilidad eliminamos esa restricción y permitimos que los caminos tengan una trayectoria en con cualquier grado de giro.

La implementación del algoritmo Theta\* se ha hecho siguiendo el pseudocódigo proporcionado en el enunciado:



Es importante recalcar que para la implementación de este pseudocódigo ha habido una gran influencia de la implementación de A\* para que todo funcione bien:

def thethaStar(start, goal, grid, heur='naive'):

    openset = set()

    closedset = set()

    start.G = 0

    start.parent = start

    current = start

    openset.add(current)

    while openset:

        current = min(openset, key=lambda o:o.G + o.H)

        pp.expanded\_nodes += 1

        if current == goal:

            path = []

            while current.G != 0:

                path.append(current)

                current = current.parent

            path.append(current)

            return path[::-1]

        openset.remove(current)

        closedset.add(current)

        for node in children(current,grid):

            if node not in closedset:

                if node not in openset:

                    #Inicializamos las variables para el vecino

                    node.G = m.inf

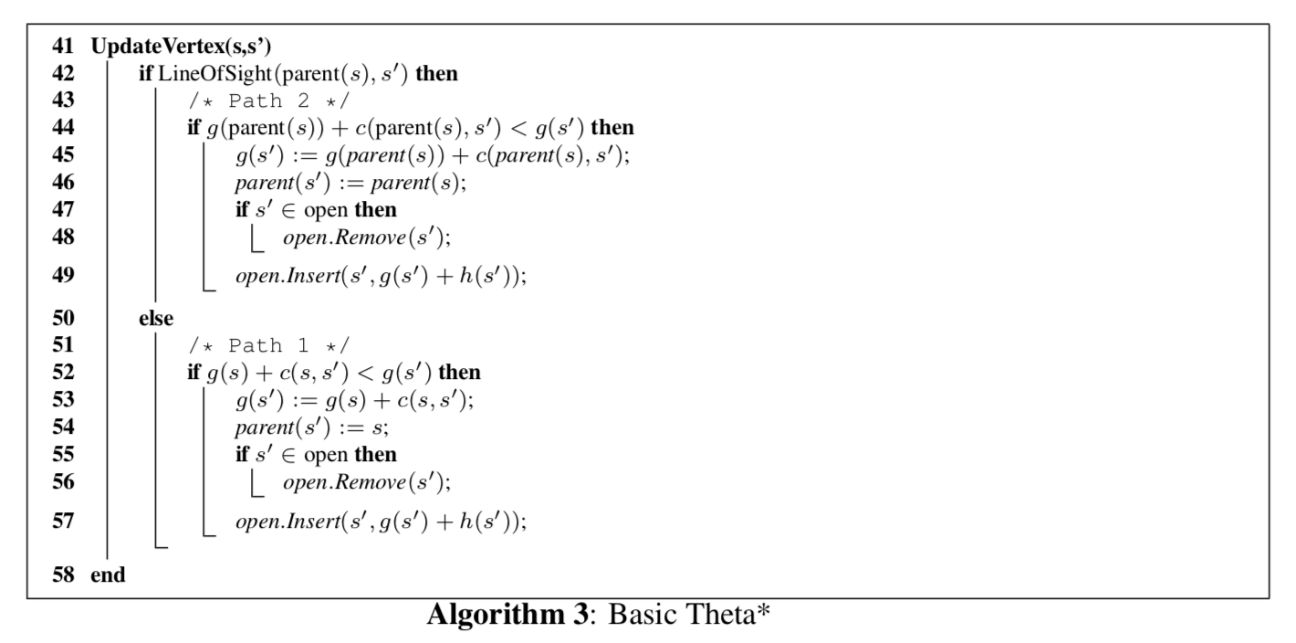
                    node.parent = None

                update\_Vertex(current, node, grid, openset, closedset, goal, heur)

    raise ValueError('No Path Found')

Una pequeña diferencia que quizás podamos encontrar con la implementación de A\* es la reconstrucción del camino conseguido una vez hemos llegado a la meta. En el algoritmo A\* ejecutábamos el bucle mientras el nodo actual tuviese algún padre. En el algoritmo Theta\* esto ha tenido que cambiar ya que establecemos que el padre del nodo inicial es él mismo (tal y como aparece en el pseudocódigo) y eso hace que caigamos en un bucle infinito. Para solucionar esto simplemente ponemos como condición de parada que el costo para llegar al nodo actual (G) sea 0 (eso nos indica que hemos llegado al nodo inicial).

La segunda diferencia, tal y como dijimos anteriormente, es el procedimiento update\_Vertex, donde entra en juego la línea de visión. En este procedimiento podemos llevar a cabo dos caminos distintos: el primero sería el que diferenciaría Theta\* de A\* y que nos permitiría realizar giros con cualquier ángulo y el segundo sería exactamente el mismo camino que seguiría A\*:



def update\_Vertex(current, children, grid, openset, closedset, goal, heur):

    """

        Procedimiento que actualiza la forma de llegar a un nodo a través de

        su nodo padre o abuelo en función de la linea de visión

        Input:

            current: nodo actual

            children: nodo hijo (vecino)

            grid: grid en el que se trabaja

    """

    #Camino 1

    if lineaDeVision(current.parent,children,grid):

        new\_g = current.parent.G + current.parent.move\_cost(children)

        if children.G > new\_g:

            children.G = new\_g

            children.parent = current.parent

            #Si ya está en abiertos, lo eliminamos para volverlo a meter actualizado

            if children in openset:

                openset.remove(children)

            children.H = pp.heuristic[heur](children, goal)

            openset.add(children)

    #Camino 2

    else:

        new\_g = current.G + current.move\_cost(children)

        if children.G > new\_g:

            children.G = new\_g

            children.parent = current

            #Si ya está en abiertos, lo eliminamos para volverlo a meter actualizado

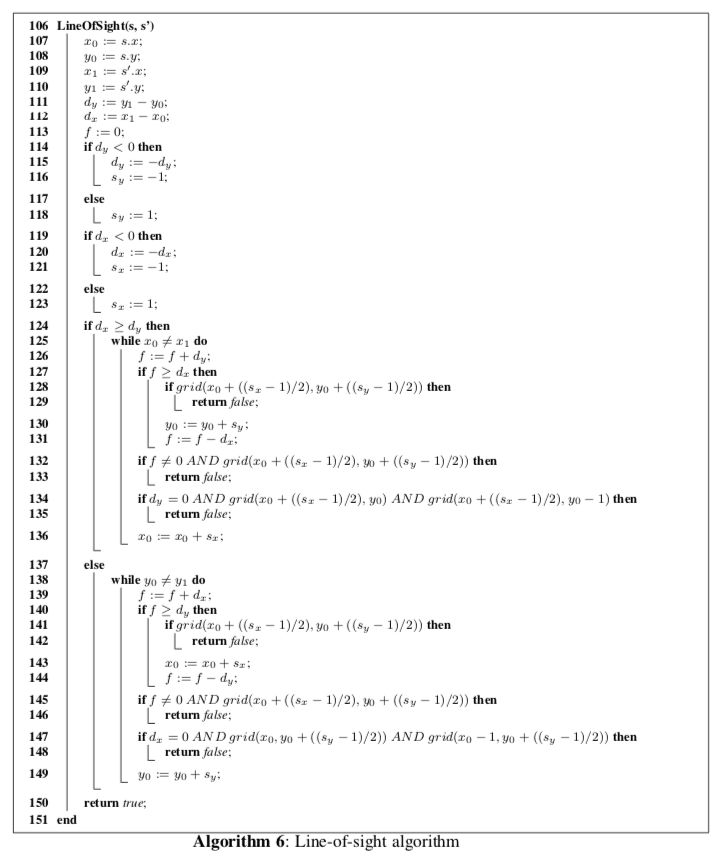
            if children in openset:

                openset.remove(children)

            children.H = pp.heuristic[heur](children, goal)

            openset.add(children)

Una vez hecho esto falta implementar el método de la línea de visión. Con este método podremos saber si podemos hacer caminos en cualquier ángulo, es decir, si podemos ir de un nodo a otro más lejano directamente sin tener que pasar por nodos intermedios ya que por medio no hay ningún obstáculo. El pseudocódigo de este método y su implementación son prácticamente iguales:



def lineaDeVision(current, children, grid):

    """

        Calcula la línea de visión del algoritmo Theta\*

        Input:

            current: nodo actual

            children: nodo vecino/hijo

            grid: grid en el que se trabaja

        Output:

            True si se puede ir del hijo al current, false en caso contrario

    """

    x0, y0 = current.grid\_point

    x1, y1 = children.grid\_point

    dy = y1 - y0

    dx = x1 - x0

    f = 0

    if dy < 0:

        dy = -dy

        sy = -1

    else:

        sy = 1

    if dx < 0:

        dx = -dx

        sx = -1

    else:

        sx = 1

    if dx >= dy:

        while x0 != x1:

            f += dy

            if f >= dx:

                if intransitable(x0+((sx-1)//2),y0+((sy-1)//2),grid):

                    return False

                y0 = y0 + sy

                f = f - dx

            if f != 0 and intransitable(x0+((sx-1)//2),y0+((sy-1)//2),grid):

                return False

            if (dy == 0 and intransitable(x0+((sx-1)//2),y0,grid) and

                intransitable(x0+((sx-1)//2),y0-1,grid)):

                return False

            x0 = x0 + sx

    else:

        while y0 != y1:

            f += dx

            if f >= dy:

                if intransitable(x0+((sx-1)//2),y0+((sy-1)//2),grid):

                    return False

                x0 = x0 + sx

                f = f - dy

            if f != 0 and intransitable(x0+((sx-1)//2),y0+((sy-1)//2),grid):

                return False

            if (dx == 0 and intransitable(x0,y0+((sy-1)//2),grid) and

                intransitable(x0-1,y0+((sy-1)//2),grid)):

                return False

            y0 = y0 + sy

    return True

En este método tenemos un método llamado *intransitable()* que es equivalente al método *grid()* del pseudocódigo. Este método recibe un punto en un grid concreto y concreta si ese punto es o no transitable:

def intransitable(x, y, grid):

    """

        Calcula si un punto es transitable

        Input:

            node: nodo que representa al punto

            grid: grid que se esta usando

        Output:

            -true en caso de no ser transitable, false en caso contrario

    """

    isInside = False

    isTransitable = True

    if x > 0 and x < len(grid) - 1:

        if y > 0 and y < len(grid[0]) - 1:

            isInside = True

            #El punto es un obstaculo

            if grid[x][y].value != 1:

                isTransitable = False

    return (not isTransitable) or (not isInside)

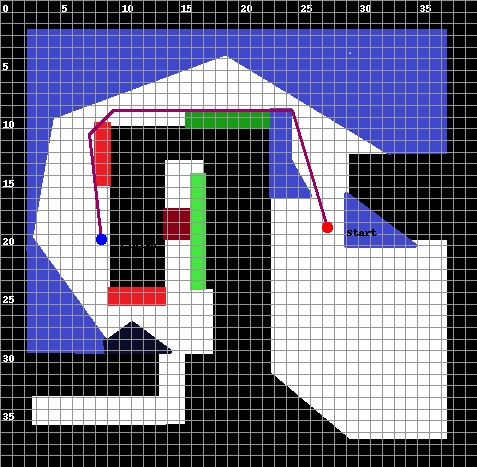
En el método *intransitable()* se devuelve true cuando ocurre lo siguiente:

* Si el punto está fuera del grid
* Si el punto es un obstáculo

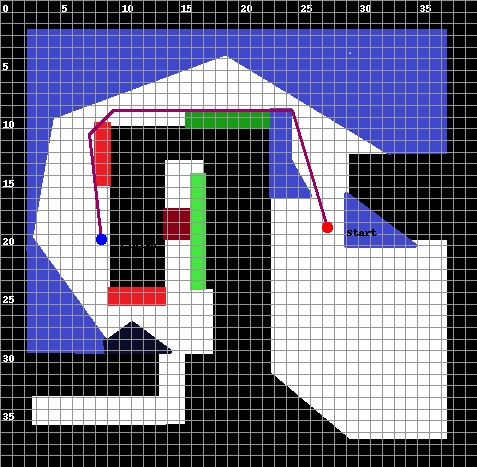
Lo primero que hace el método es verificar que el punto se encuentra dentro del grid ya que sino no tiene sentido saber si es o no un obstáculo. En el grid los puntos tienen valores que van del 1 al 9, siendo el 1 el espacio blanco y el 9 el espacio en negro (los números intermedios son los diferentes colores que representan zonas que no son obstáculos en sí pero que tienen más coste por pasar por ellas). Dado que queremos que evita absolutamente todo menos los espacios en blancos consideramos que el punto es un obstáculo si su valor es distinto de 1 (distinto de blanco).

Una vez hecho todo esto consideramos ya por terminada la implementación de Theta\*. Pasa observar cómo es la diferencia de los caminos que logra con respecto a A\* vamos a ejecutar el mismo escenario del punto anterior pero usando Theta\*:

*python3 run\_path\_planning.py --scenario ../res/test\_2.png --start "(27,19)" --finish "(8,20)" --grid\_size 40 --algorithm Theta\* --heuristic naive*

**

Vamos a disponer las dos imágenes una al lado de otra para verlo mejor:

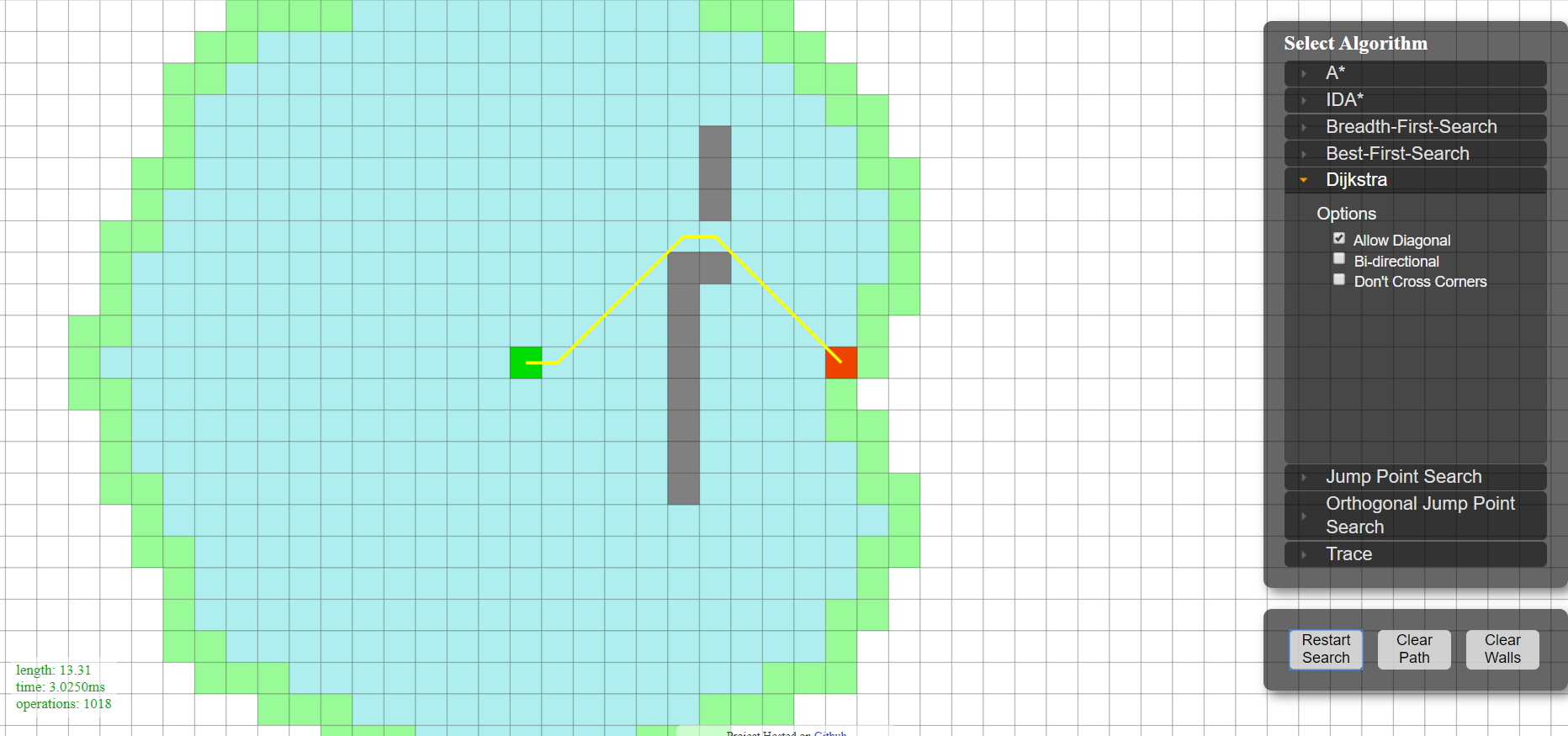
A*
 **

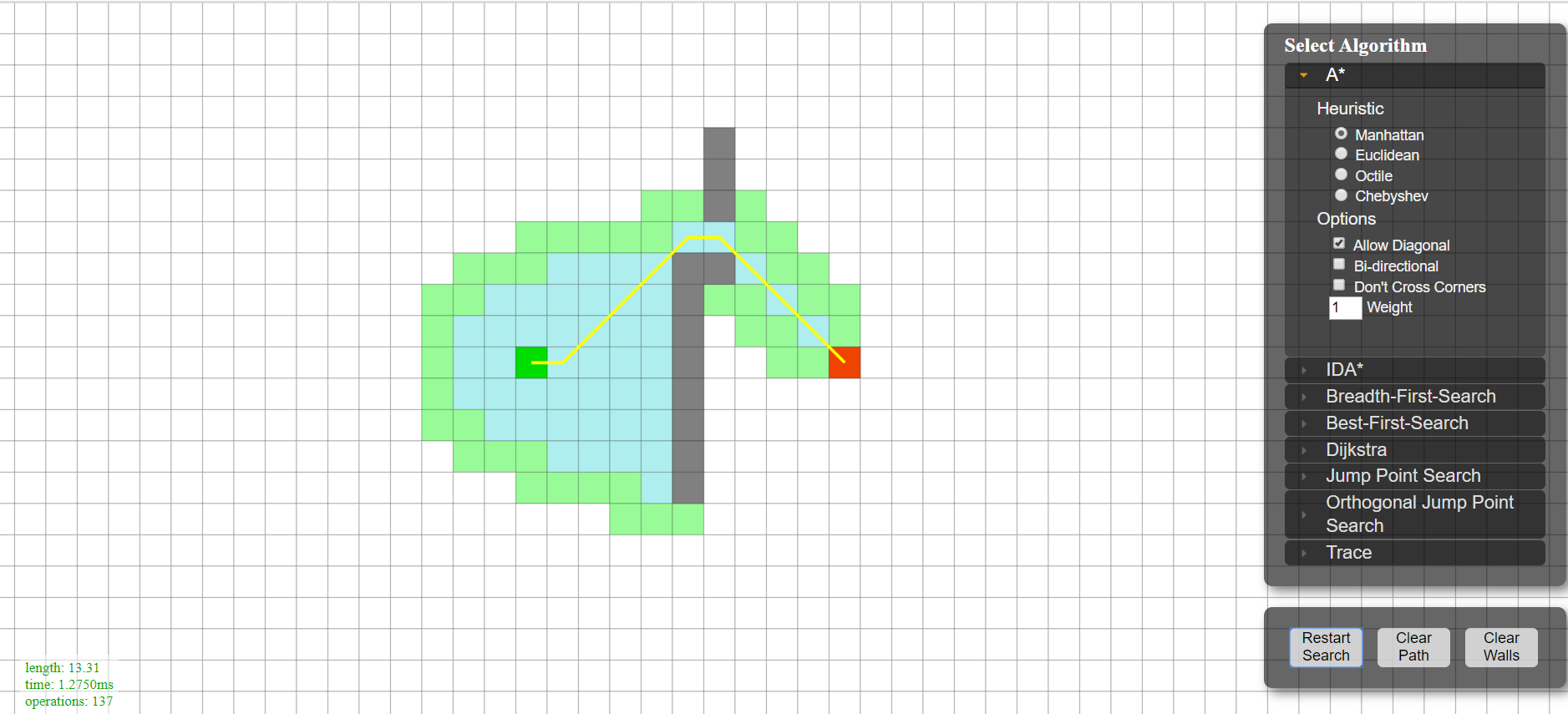
Camino con A\* Camino con Theta\*

Se aprecia perfectamente la diferencia de los caminos entre los dos algoritmos: mientras que A\* está restringido a giros de 45º, Theta\* no tiene esa restricción. También podemos observar la diferencia entre los algoritmos según el coste y longitud del camino: Theta\* consigue un camino de coste y longitud 5 mientras que A\* consigue un camino de coste y longitud 38.

### Expansión de los nodos

Antes de comenzar con las heurísticas es interesante observar cómo se expanden los nodos en la búsqueda. En concreto, creo muy interesante la comparación entre Dijkstra y A\* con heurísticas. Como vimos anteriormente ambos algoritmos nos proporcionaban caminos y número de nodos expandidos prácticamente iguales. Esto sucedía principalmente porque A\* usaba la heurística naive, que al ser constante no nos proporciona información y es como si estuviéramos haciendo una búsqueda a ciegas. Haciendo uso de [este](https://qiao.github.io/PathFinding.js/visual/) simulador podemos comparar ambos algoritmos en términos de nodos expandidos. Realizamos un pequeño escenario y observamos los resultados:





El punto verde oscuro representa la salida y el punto rojo representa la meta. En cuanto a nodos expandidos podemos ver que en azul se representan los nodos ya expandidos y metidos en la lista de cerrados y en verde claro los nodos en la lista de abiertos. Como se puede ver, A\* con una heurística como la distancia Manhattan deja de ser una búsqueda a ciegas como puede ser Dijkstra y pasa a ser una búsqueda informada. La diferencia entre los dos algoritmos es evidente: con A\* usando la distancia Manhattan el número de nodos expandidos y por tanto el tiempo necesario para calcular el camino es significativamente mejor que con Dijkstra.

Por el contrario, se puede ver como estos dos algoritmos consiguen encontrar el mismo camino. Por tanto, su diferencia radica, grosso modo, en el número de nodos expandidos.

### Implementación de heurísticas

En este apartado vamos a implementar las heurísticas para los algoritmos A\* y Theta\*. En concreto vamos a implementar la distancia Manhattan, la distancia Euclidiana, y la distancia Octal:

#### Distancia Manhattan

La distancia Manhattan se calcula mediante la suma de las diferencias absolutas de las coordenadas de dos puntos. Es decir, dado un punto P1 y P2 su distancia Manhattan se calcula como sigue:

A continuación, se presenta el código de la implementación de dicha distancia:

def manhattan(point,point2):

    """

        Function that performs Manhattan heuristic.

    """

    x1, y1 = point.grid\_point

    x2, y2 = point2.grid\_point

    dx = abs(x1-x2)

    dy = abs(y1-y2)

    return dx+dy

#### Distancia Euclidiana

La distancia Euclidiana entre dos puntos se calcula aplicando el teorema de Pitágoras. Es decir, dado un punto P1 y P2 su distancia Euclidiana se calcula como sigue:

def euclidean(point, point2):

    """

        Function that performs euclidean heuristic.

    """

    x1, y1 = point.grid\_point

    x2, y2 = point2.grid\_point

    dx = (x2-x1)\*\*2

    dy = (y2-y1)\*\*2

    return m.sqrt(dx+dy)

#### Distancia Octal

Dado un punto P1 y P2 su distancia Octal se calcula como sigue:

, donde y

def octile(point, point2):

    """

        Function that performs octile heuristic.

    """

    x1, y1 = point.grid\_point

    x2, y2 = point2.grid\_point

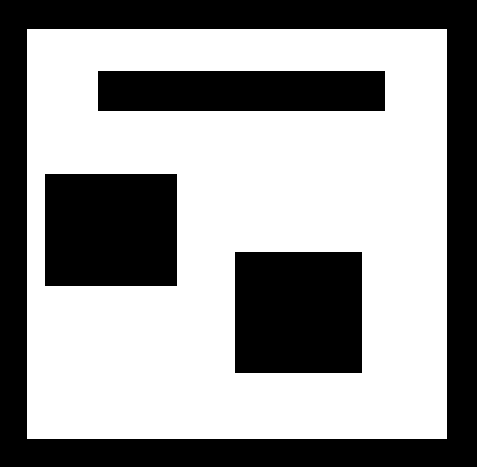
    dx = abs(x1-x2)

    dy = abs(y1-y2)

    return m.sqrt(2)\*min(dx,dy)+abs(dx-dy)

### Comparación de heurísticas y algoritmos en un escenario creado desde 0

En este apartado vamos a comparar los tres algoritmos con sus heurísticas. Para ello creamos un mapa personalizado inspirado en la diapositiva 23 de las diapositivas de path planning:



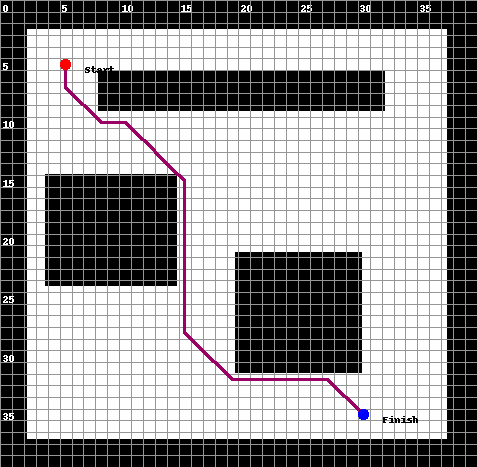
En todas nuestras pruebas el punto de inicio será la esquina superior izquierda y el punto final será la esquina inferior derecha. Primero vamos a capturar las imágenes de los recorridos conseguidos por los algoritmos.

* Dijkstra:

El comando usado será el siguiente:

*python3 run\_path\_planning.py --scenario ../res/test\_3.png --start "(5,5)" --finish "(30,35)" --grid\_size 40 --algorithm Dijkstra --heuristic naive*

Al no usar heurística, este algoritmo siempre nos dará el mismo resultado elijamos la heurística que elijamos. A continuación, se muestra el resultado de la ejecución:



Como se ve, el algoritmo consigue encontrar un camino que llega a la meta indicada. El número de nodos expandidos así como la longitud del camino y su coste se guardará para posteriormente hacer una comparativa general en una tabla.

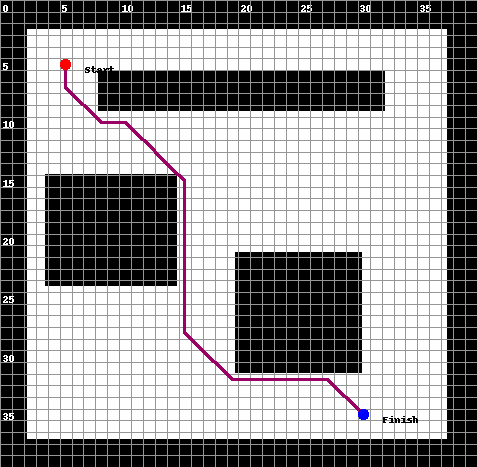
* A\*:

El comando usado será como este, cambiando los nombres de las heurísticas en la última parte del comando:

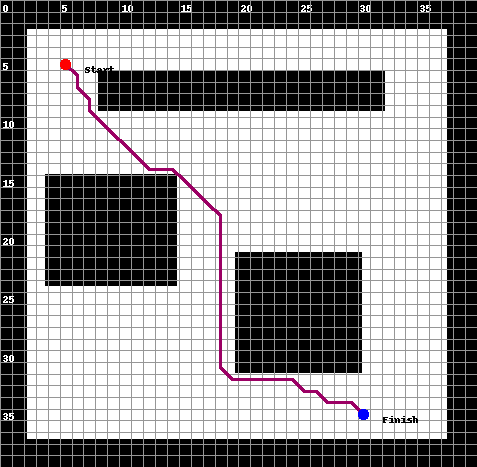
*python3 run\_path\_planning.py --scenario ../res/test\_3.png --start "(5,5)" --finish "(30,35)" --grid\_size 40 --algorithm A\* --heuristic “nombre heurística”*

Donde *“nombre heurística”* puede tomar los siguientes valores: *naive, euclidean, octile, manhattan*.

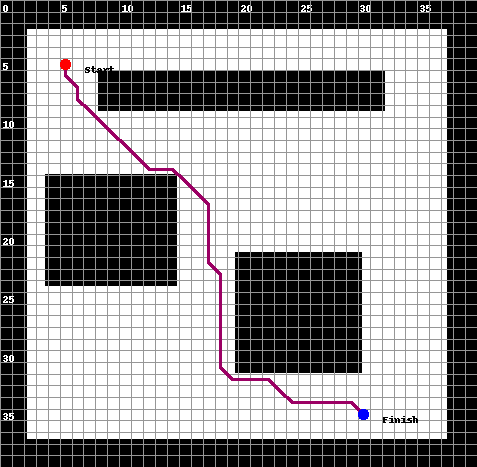
* + Naive:



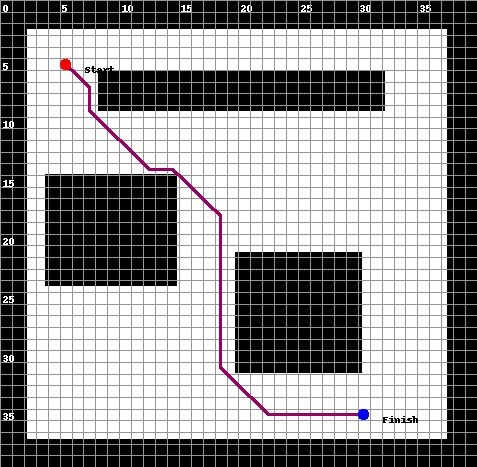
* + Euclidean:



* + Octile:



* + Manhattan:

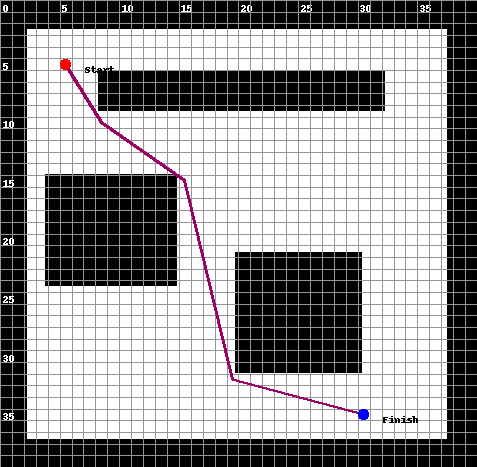


* Theta\*:

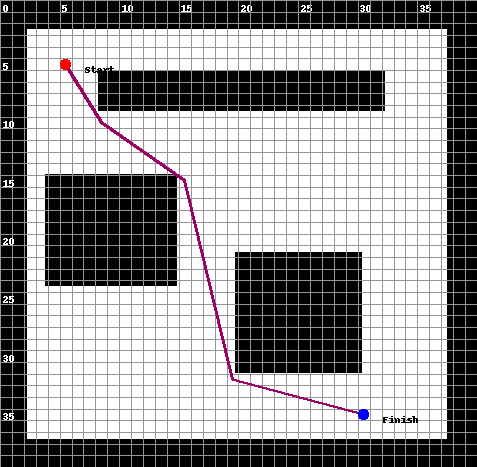
Al igual que en A\* el comando usado será de este estilo:

*python3 run\_path\_planning.py --scenario ../res/test\_3.png --start "(5,5)" --finish "(30,35)" --grid\_size 40 --algorithm Theta\* --heuristic “nombre heurística”*

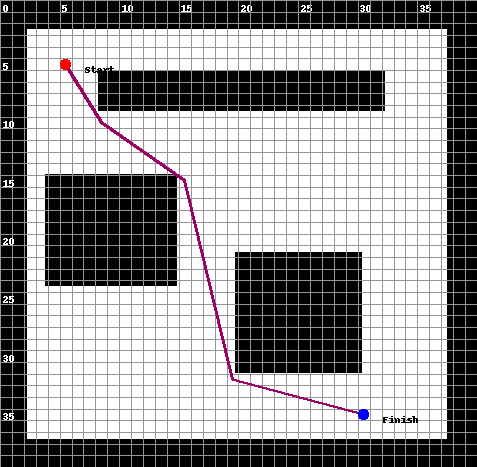
* + Naive:



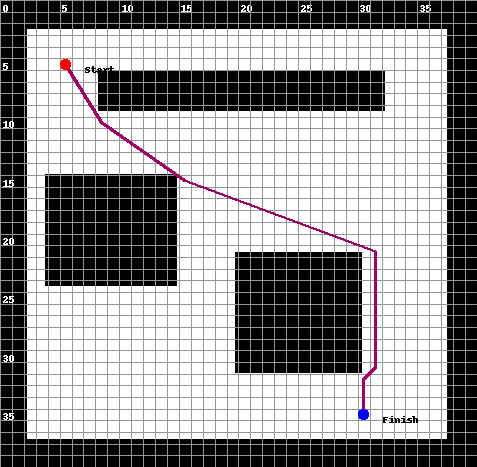
* + Euclidean:



* + Octile:



* + Manhattan:



Tal y como se puede apreciar los algoritmos Dijkstra y A\* con sus diferentes heurísticas consiguen encontrar caminos muy parecidos entre sí. Por otro lado, se puede ver claramente la diferencia que indicábamos que había entre A\* y Theta\*, donde se puede ver que en Theta\* los caminos consiguen librarse de la restricción de giro de 45º y permiten ir más directos al objetivo, por lo que su coste y longitud de nodos recorridos será menor.

Para ver todos estos algoritmos y heurísticas conjuntamente vamos a disponer de una pequeña tabla comparativa donde veamos las tres métricas que nos indican por consola al ejecutar el algoritmo: costo y longitud del camino conseguido, y el número de nodos explorados:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algoritmo | Heurística | Longitud del camino | Coste del camino | Nodos explorados |
| Dijkstra | - | 41 | 41 | 778 |
| A\* | Naive | 41 | 41 | 778 |
| Euclidean | 41 | 41 | 322 |
| Octile | 41 | 41 | 265 |
| Manhattan | 41 | 41 | 77 |
| Theta\* | Naive | 5 | 5 | 778 |
| Euclidean | 5 | 5 | 251 |
| Octile | 5 | 5 | 230 |
| Manhattan | 7 | 7 | 52 |

Tal y como se ve, tanto en la tabla como en las imágenes vemos que Dijkstra y A\* consiguen encontrar caminos de igual longitud y coste. La diferencia entre estos dos algoritmos radica en el uso de las heurísticas: con A\* usando una heurística distinta a Naive conseguimos reducir mucho el tiempo de búsqueda con una menor cantidad de nodos explorados. Véase que A\* con Naive trabaja como Dijkstra, ya que Naive no proporciona ningún tipo de información.

Por otro lado, tenemos que Theta\* consigue encontrar más rápidamente el camino y de una forma mejor con menor coste y longitud. Theta\* reduce al máximo el número de nodos a visitar para encontrar el camino. Véase también el efecto de las heurísticas, que funcionan mejor con Theta\* que con A\*.

Si comparamos las heurísticas sabemos que Naive es la peor de todas ya que es la que no proporciona ningún tipo de información, las otras tres al proporcionar información sobre la distancia al objetivo permiten que el algoritmo (en mayor proporción Theta\* que A\*) explore menos nodos y vaya más directo a la solución. La mejor de todas las heurísticas es sin duda la distancia Manhattan que reduce al máximo el número de nodos explorados.

Las diferencias de los caminos conseguidos con Theta\* se deben principalmente al efecto de la heurística, ya que es esta la que nos guía según su criterio hacia la meta. Es por ello por lo que con la distancia Manhattan conseguimos un camino distinto al del resto de heurísticas, pero es muy interesante ver que el número de nodos explorados es muy bajo.

## Parte 2: Integración

En esta parte vamos a realizar una pequeña integración entre Path Planning y PDDL. Para ello reusaremos el dominio y problema usados en el primer ejercicio de la práctica 2, el del robot explorador de Marte.

### Adaptación de Dominio/Problema

En este apartado voy a explicar qué cambios he introducido en el dominio y problema para realizar la integración. Lo primero que tenía que hacer era cambiar ligeramente el dominio para eliminar las dos acciones que teníamos para movernos (lentos o rápidos) y hacer algo similar con el ejercicio de la cooperación con los modos de navegación. Dado que esto introduciría la necesidad de un nuevo parámetro de entrada a la acción (modo rápido o lento) al final tuve que deshacerme de esta utilidad y permitir solamente un modo. Todo lo demás en el dominio (excavar muestras, hacer fotos, permitir hacer varias tareas en el mismo sitio, consumo de batería, etc.…) se ha conservado.

En cuanto al dominio se ha mantenido la idea de tener 6 puntos pero sus nombres se han sustituido conforme a sus coordenadas en el mapa *test\_2.png* que vamos a usar en esta integración. Todo lo demás se mantiene (con el sumo cuidado de sustituir el nombre de los puntos): estado inicial, objetivos del plan, etc. Al final en el anexo se incluye el código del dominio y problema descritos para esta integración, además del plan conseguido.

### Implementación de run\_integration.py

La implementación de este código se ha hecho a partir del siguiente [enlace](https://gist.github.com/R012/0fe8ec73b58c0a95844370c6510454d9). Para adaptar el código a nuestra situación establecemos el punto de inicio como el mismo punto de inicio del plan y especificamos las rutas de donde se encuentra tanto el plan como el mapa:

    position = (3, 35) # change this to the starting position for your PDDL problem

task\_plan = generate\_task\_list('../res/planRobot.txt') # Must be a path to a text file

visualize\_paths\_from\_pddl(task\_plan, '../res/test\_2.png') # path\_to\_map\_file should be a path to an image

### Ejecución de la integración

Una vez hecho todo esto ya podemos ejecutar la integración. Para ver distintos resultados vamos a ejecutar la simulación de 3 formas distintas:

1. Usando el algoritmo de Dijkstra
2. Usando el algoritmo A\* con heurística la distancia Manhattan
3. Usando el algoritmo Theta\* con heurística la distancia Manhattan

#### Dijkstra

#### A\* y distancia Manhattan

#### Theta\* y distancia Manhattan

## Anexo

### Dominio del problema del robot de Marte

### Problema del problema del robot de Marte

### Plan del problema del robot de Marte