Robótica - Taller 3
Planeación de rutas y percepción en Robots móviles

John Alejandro Duarte Carrasco - ja.duarte10@uniandes.edu.co - 2016140888
Jonathan Steven Roncancio Pinzon - js.roncancio@uniandes.edu.co - 201617312
Miguel Angel Mozo Reyes - ma.mozo@uniandes.edu.co - 201615973
Santiago Devia Valderrama - s.devia10@uniandes.edu.co - 201313766

15 de abril de 2019

1. Punto 1

1.1.

(0.65) La implementación de la solución se puede encontrar en el vídeo adjunto al archivo en la carpeta de respuestas

1.2.

(0.35) Para la solución del problema se implemento en Python un código que implementará el algoritmo de *Bread First Search* para encontrar la ruta más corta del Pacman a una galleta, y una vez encontrada dicha ruta, se publicaría en el tópico de control de acciones de pacman para que este siguiese dicha ruta.

Para la implementación del algoritmo se importaron, además de todas las librerías relacionadas a ROS-pacman, 2 librerías de python particularmente importantes: la primera la librería *Queue* y la segunda la librería *Time*.
La librería de filas (queue) permite crear un arreglo con las características de una fila tipo FIFO, es decir, añadir al fondo de la fila un objeto y extraer de la fila el objeto que se haya añadido con mayor anterioridad, gracias a esto se implementa el algoritmo de la siguiente forma:

- Se crea una fila vacía
- Se extrae el primer elemento de la fila
- Al elemento extraído se le añaden los 4 posibles movimientos de pacman (generando así 4 elementos adicionales para la primera iteración).
- Se evalúa la factibilidad de esos movimientos adicionales desde la posición del pacman (que no generen choques con los obstáculos del mapa).
- Se evalúa la posición a la que llevaría esa combinación de movimientos.
- Si la combinación de movimientos es factible y lleva a un posición que no ha sido visitada, se añade dicha combinación a la fila y se marca como visitada. De lo contrarío se continua evaluando las demás combinaciones para agregarlas a la fila. Sí la combinación de movimientos no lleva a una posición con galleta se repite el algoritmo desde el paso 2.
- Si la combinación de movimientos genera una posición en la que hay una galleta, se ejecuta el código de movimiento de pacman y se reinicia la variable que guarda si una posición ha sido visitada o no (un matriz).
- Sí el número de galletas en el mapa es 0 el algoritmo termina, de lo contrario se repite desde el paso 1.

El proceso anterior se puede ver representado en la siguiente figura (1).

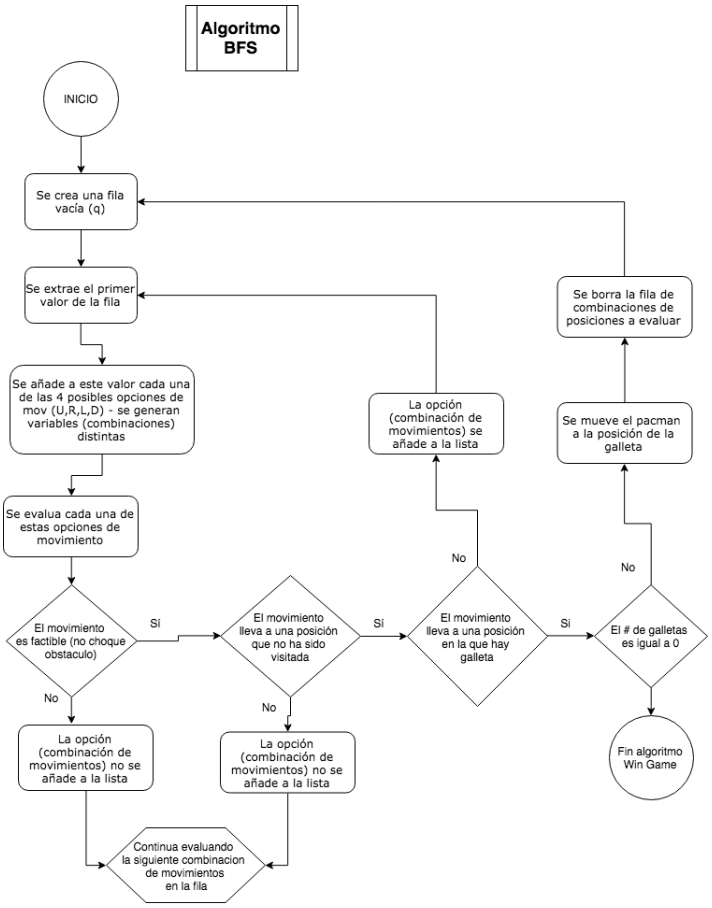


Figura 1: Flujograma algoritmo BFS

Ahora bien, para realizar evaluar el camino óptimo a recorrer y ejecutar las acciones de pacman, se hace uso de distintas funciones creadas en python junto con la información arrojada por el nodo de ROS. Finalmente es importante mencionar que para ejecutar las acciones del pacman se hace uso de la librería *time* de python la cual permite poner a 'dormir' el *Script* dejando de ejecutar el código por un tiempo definido. Para que pacman realice las acciones de forma adecuada (una vez se ha encontrado la ruta óptima), se ejecuta publica un mensaje en el tópico *PacmanController0* con el valor de la acción a tomar (0,1,2,3,4) se poner a 'dormir' el código durante 150 ms y se detiene momentáneamente la acción del pacman (enviando un 4) y luego se envia el siguiente mensaje con la acción correspondiente y se repite el proceso. Se escoge esta tasa de suspensión del código de 150 ms debido a que este es el tiempo en el que el nodo de pacman refresca los tópicos y mensajes y la información relacionada con estos, debido a esto, también se define la frecuencia del nodo como $rate = \frac{1}{0,15} Hz$.

En la siguiente figura se muestra la definición de las funciones utilizadas en python con su respectiva descripción junto con la información extraída de los tópicos de pacman y las variables utilizadas para la realización y ejecución del algoritmo descrito anteriormente.

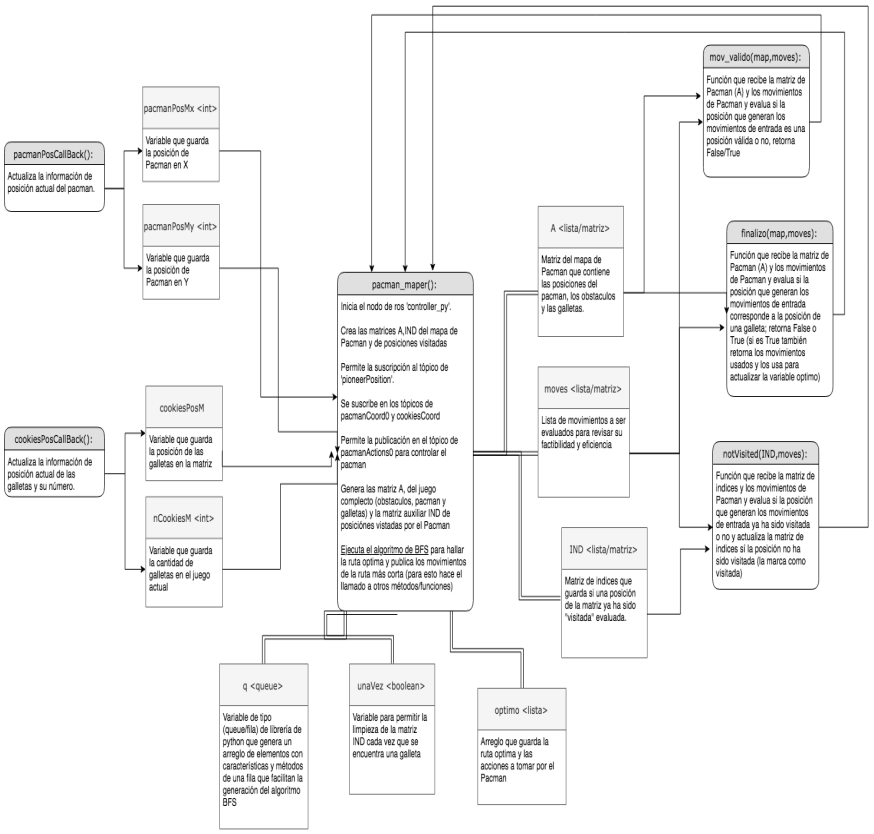


Figura 2: Diagrama funcionamiento punto 1

Adicionalmente, se muestra a continuación el diagrama de conexión de tópicos generado por ROS (*rqt_graph*)

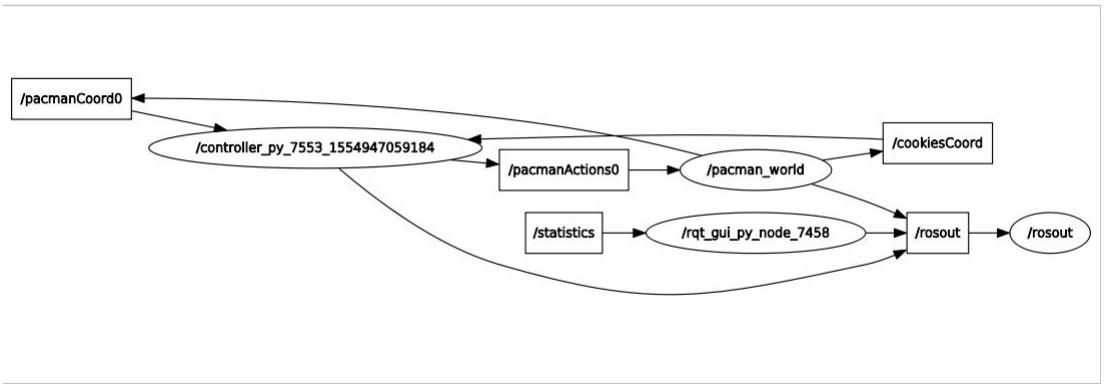


Figura 3: Grafo de nodo ROS

1.3.

(0.25)

En la carpeta de resultados se puede encontrar el los vídeos de la solución generada por el algoritmo para 3 niveles (distintos a mediumCorners) de distinta dificultad en términos de su tamaño y cantidad de galletas en el mapa. Los 3 niveles resueltos fueron: *bigMaze*,*oddSearch* y *greedySearch*. Para resolverlos el algoritmo, de acuerdo a la implementación del primer literal, escoge la galleta que se encuentre más cercana y envía al pacman a dicha dirección, una vez encontrada la galleta y movido el pacman, se recalcula la ruta para llevarlo a la siguiente galleta más cercana de acuerdo a la nueva posición. En este caso el criterio de selección de galleta se basa en el mismo algoritmo BFS en el cual se busca el camino más corto al objetivo y se recalcula de acuerdo a la nueva posición existente del pacman.

1.4.

(0.25)

Luego de realizar la simulación del movimiento de pacman 20 veces consecutivas se obtuvieron los siguientes resultados:

PACMAN MEDIUM CORNERS									
PRUEBA	t [s] en encontrar galleta:				t [s] en llegar a galleta:				t [s] total
	Galleta 1	Galleta 2	Galleta 3	Galleta 4	Galleta 1	Galleta 2	Galleta 3	Galleta 4	
1	0,0147	0,0157	0,0762	0,0023	2,718	2,874	8,156	2,26	16,008
2	0,0099	0,0205	0,0562	0,0021	2,724	2,873	8,18	2,268	16,045
3	0,0041	0,0183	0,0635	0,0044	2,729	2,868	8,155	2,262	16,014
4	0,0145	0,0148	0,089	0,0066	2,713	2,869	8,147	2,271	16
5	0,0057	0,031	0,067	0,0021	2,731	2,867	8,159	2,267	16,024
6	0,0047	0,0213	0,0552	0,0061	2,725	2,869	8,156	2,265	16,015
7	0,0044	0,0421	0,0589	0,002	2,7152	2,872	8,171	2,563	16,3212
8	0,0099	0,0159	0,066	0,0033	2,718	2,871	8,148	2,262	15,999
9	0,0086	0,0059	0,0751	0,0027	2,729	2,8904	8,167	2,822	16,6084
10	0,0086	0,0151	0,0543	0,0062	2,7175	2,873	8,184	2,272	16,0465
11	0,0123	0,0221	0,0604	0,01	2,727	2,871	8,176	2,564	16,338
12	0,0096	0,0058	0,0707	0,0032	2,724	2,874	8,15	2,271	16,019
13	0,0091	0,0101	0,0563	0,0019	2,717	2,874	8,141	2,2624	15,9944
14	0,011	0,0067	0,0672	0,0021	2,721	2,869	8,151	2,281	16,022
15	0,0087	0,0311	0,0861	0,0079	2,73	2,811	8,19	2,276	16,007
16	0,0043	0,0212	0,0538	0,0064	2,782	2,861	8,165	2,27	16,078
17	0,0051	0,032	0,071	0,0023	2,715	2,831	8,131	2,233	15,91
18	0,0079	0,0185	0,0525	0,0036	2,73	2,866	8,178	2,262	16,036
19	0,0094	0,0313	0,0477	0,0094	2,734	2,874	8,16	2,259	16,027
20	0,009	0,0148	0,0451	0,0066	2,718	2,868	8,171	2,263	16,02
PROMEDIO	0,0086	0,0197	0,0636	0,0046	2,7259	2,8663	8,1618	2,3227	**16,0766**
DESV	0,0032	0,0097	0,0119	0,0026	0,0146	0,0168	0,0153	0,1493	**0,1611**

Figura 4: Datos simulaciones

De acuerdo a estos datos se puede ver que el tiempo que le toma al pacman recorrer el mapa completo en todos los casos es casi idéntico, con una un valor medio de 16 segundos y una desviación estándar de apenas una décima de segundo, esto concuerda con los datos esperados pues, una vez el código ha realizado el trabajo de calcular la ruta, su trabajo se limita a publicar los movimientos a seguir con una tasa de espera de 150 ms (mucho más lento que la velocidad a la que se ejecuta el código de python) de modo que la ejecución las diferencias de tiempo entre iteración e iteración es mínima.

Por otro lado, para el caso de los tiempos en los que el algoritmo tarda en calcular la ruta más corta a cada galleta, se puede notar una gran diferencia respecto a los tiempos de movimiento del pacman, nuevamente debido a que la tasa a la que se ejecuta el código en python en mucho mayor a la tasa de refresco del nodo de ROS, el tiempo en el que se encuentra la ruta óptima es mucho menor, en general, para cada par de puntos a galleta se tienen los siguientes valores de tiempo promedio y desviación estandar:

- Galleta 1: $(\mu,\sigma) = (0.0086;0.0032)$
- Galleta 2: $(\mu,\sigma) = (0.0197;0.0097)$
- Galleta 3: $(\mu,\sigma) = (0.0636;0.0119)$
- Galleta 4: $(\mu,\sigma) = (0.0046,0.0026)$

Como se puede ver, el tiempo más corto para encontrar una galleta es el de la galleta 4 pues para este caso el pacman inicia desde una posición bastante cercana a dicha galleta, caso contrario al de encontrar la galleta 3. Adicionalmente se tiene una desviación estándar relativamente grande en comparación con la media de los datos pues una tasa de refresco tan alta como la del código en python genera mayores distorsiones a la hora de comparar dicha desviación con la media pues ambos están en ordenes de magnitud similares (contrario a lo que ocurre en los tiempos de movimiento del pacman).

2. Punto 2

a)

La escena creada como solución de este punto se encuentra en la carpeta resources y lleva el nombre de *Escena.ttt*. Dicho archivo contiene una escena de ROS con 5 obstáculos cilíndricos de diferente color y radio, los cuales se pueden ver en la figura 5.

Una vez se corre la escena se crea el tópico *obstaclesPosition* que publica el numero de obstáculos, su posición y diámetro. El mensaje se construye de la siguiente forma:

- La primera posición del vector corresponde al numero de obstáculos presentes en el mapa

$$data[0] = \#ObstáculosEnElmapa$$

- Tomando el caso donde el numero de obstáculos en el mapa es igual a 5 (caso del taller), de la posición 1 a la 5 se encuentran las posiciones en x de cada cilindro, siendo x_i la coordenada x del cilindro i se tiene que:

$$data[1:5] = [x_1, x_2, x_3, x_4, x_5]$$

- Siendo con el caso del taller, de la posición 6 a la 10 se envían las posiciones en y de cada cilindro, siendo y_i la coordenada y del cilindro i:

$$data[6:10] = [y_1, y_2, y_3, y_4, y_5]$$

- Por ultimo desde la posición 7 hasta el final del vector se tienen los diámetros de cada cilindro, con d_i siendo el diámetro del cilindro i:

$$data[7:end] = [d_1, d_2, d_3, d_4, d_5]$$

Siendo la variable *data* un vector que contiene la información del mensaje. Además, es importante tener en cuenta que la coordenada x_i, y_i de cualquier obstáculo corresponde al centro del mismo.

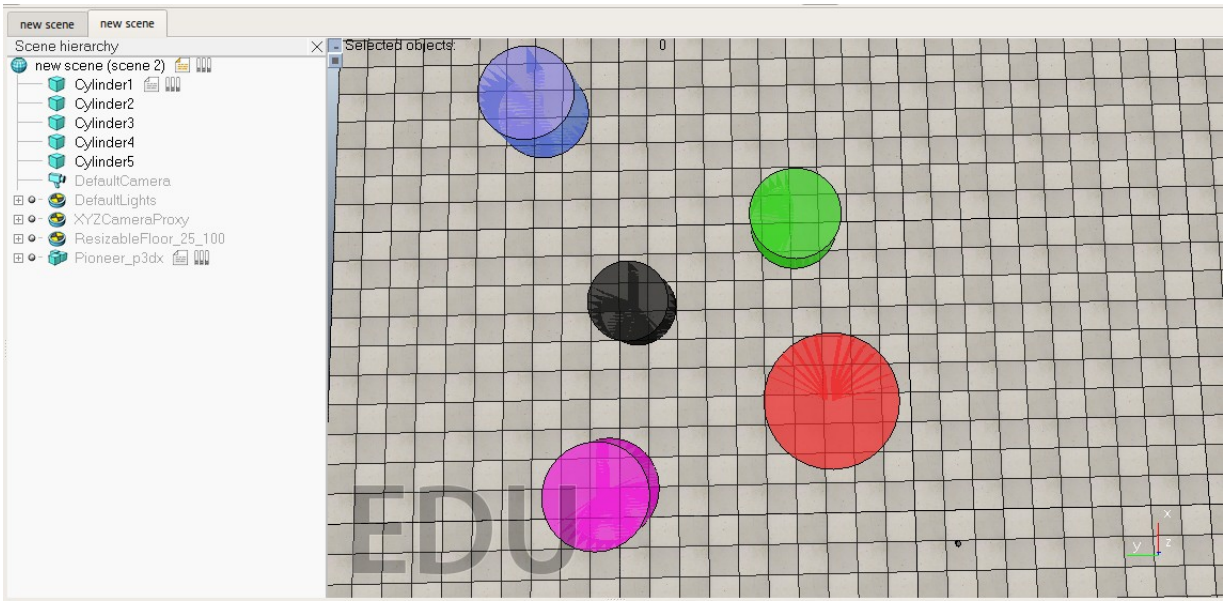


Figura 5: Obstáculos presentes en el mapa

b)

Para este punto se tenia que diseñar un código que creara la grilla del mapa y el grafo de esta. Para esto, el código inicia su ejecución creando el nodo ante ROS, suscribiéndose al tópico de *obstaclesPosition*. Una vez se actualiza por primera vez la información por este tópico, se inicia los métodos para crear la grilla y el grafo. Para la creación de la grilla, se crea una matriz de ceros que significan que no hay un obstáculo en dicha posición. Luego, se realiza un barrido por cada grilla posición de la matriz, representado cada grilla en el mapa, determinando si dicha posición corresponde a un obstáculo o no. Si corresponde a un obstaculo, se dispone un 1 en dicha posición de la matriz. Dicha matriz tiene un tamaño de $tamanoMapa/tamanoGrilla$, en este caso de 100x100 casillas.

Para la creación del grafo, se utilizo la librería NetworkX. Esta permite la creación de grafos de forma sencilla, y permite realizar operaciones matemáticas en ellos de forma sencilla y versátil. Para este caso, se creo un grafo del número de casillas en la matriz, disponiendo un nodo por cada casilla, y un arco a todos los nodos vecinos. Dichos arcos son 4 (arriba, abajo, izquierda y derecha) y 4 más a los nodos dispuestos en posiciones diagonales. Al final, se gráfica la grilla y el grafo en figuras diferentes:

A continuación se puede evidenciar dichas gráficas para un tamaño de mapa de 50m, y un tamaño de grilla de 0.5m:

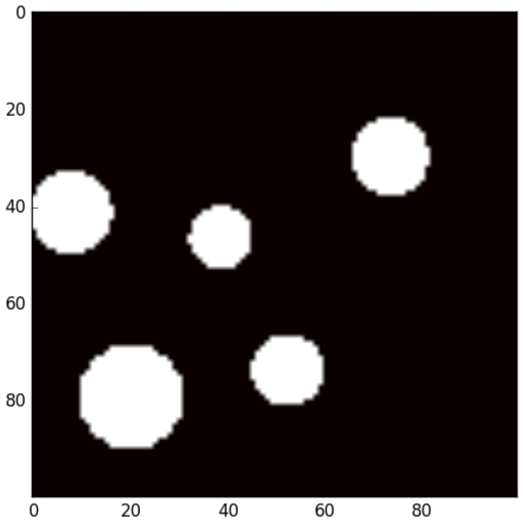


Figura 6: Gráfica de la grilla creada por el programa para un tamaño de mapa de 50m y un tamaño de grilla de 0.5m.

Figura 7: Gráfica del grafo creada por el programa para un tamaño de mapa de 50m y un tamaño de grilla de 0.5m.

A continuación se ve un zoom de dicho grafo:

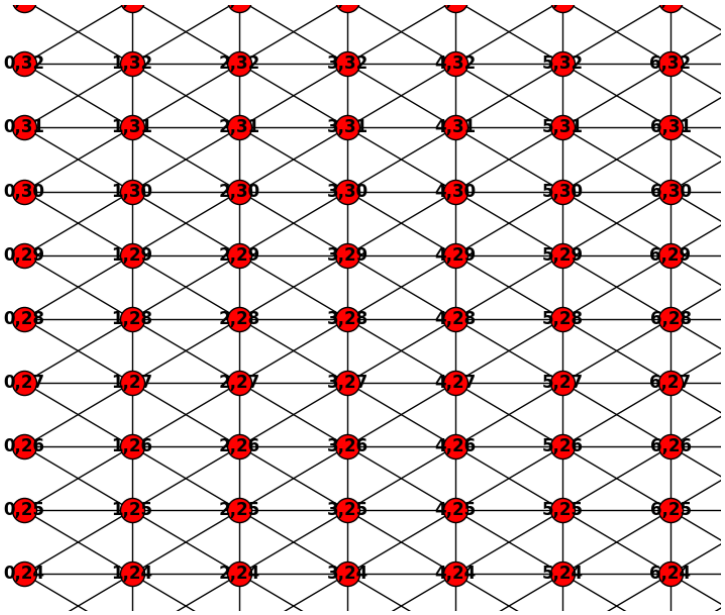


Figura 8: Zoom del grafo creada por el programa para un tamaño de mapa de 50m y un tamaño de grilla de 0.5m.

c-d)

En primer lugar, al inicializar el nodo se realizan las suscripciones con los tópicos *obstaclesPosition*,*simulationTime* y *pioneerPosition* con el fin de conocer en todo momento la posición actual del robot, la distribución de los obstáculos en el mapa y el tiempo que lleva en ejecución la simulación. Así mismo, se realiza la relación de publicación con el tópico *motorsVel* teniendo en mente utilizar la ley de control vista en clase para manejar las velocidades del robot según se necesite. Las relaciones enunciadas anteriormente se ven reflejadas en la figura 9.Además, se estableció la frecuencia de muestreo del sistema en 20Hz.

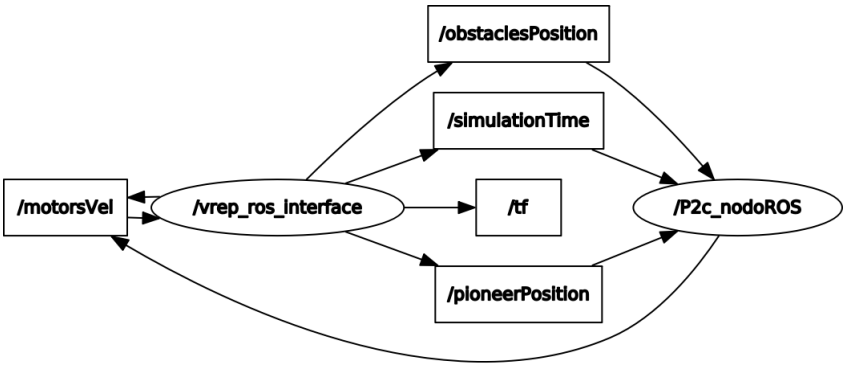


Figura 9: Diagrama de nodos de ROS para el punto 2.c

Una vez se crean las relaciones con los demás tópicos de ROS el programa utiliza la grilla del mapa y el grafo creados en el literal 2.b para calcular la ruta mas corta entre el punto inicial del robot y el punto final ingresado por parámetro. Dicha ruta se calcula haciendo uso de el algoritmo A* implementado en la librería *networkx*, este solicita como parámetros de entrada una función que calcule la heurística entre dos puntos,el grafo del mapa y la pareja de puntos [Inicial,Final].

Para la función heurística se utilizo la distancia Manhattan como representación del costo estimado entre dos puntos con un peso D=3.Dicho peso se escogió después de someter el algoritmo a un barrido del parámetro D entre 1 y 10; en este barrido se halló que para pesos menores a 3 A* es mas çonservadorz se demora más en calcular la ruta óptima, mientras que para valores de D mayores a 3 el algoritmo calcula la ruta de manera rápida, pero en algunos casos termina siendo muy larga o con muchas curvas por lo que el robot se demora más tiempo en alcanzar el objetivo.

Después de calcular la ruta hasta el objetivo, se utiliza la ley de control aplicada en el punto 4 del taller pasado con algunas variaciones. La variación mas importante consiste en pasar por parámetro a la ley de control la posición

a la que se quiere desplazar el robot en lugar de asumir que esta posición siempre es la posición final. Esto se hizo con el fin de recorrer la ruta óptima cambiando cada cierto tiempo la variable *posicionDeseada* que ingresa a la ley de control, hasta que el robot llegue a la posición final. Para lograrlo, se estableció un limite de .erroro "visión"que corresponde a la mínima distancia que puede haber entre el robot y la posición deseada actual, de forma tal que cuando este limite es superado la variable *posicionDeseada* pasa a ser la siguiente posición en la ruta calculada. Es decir, una vez el robot se encuentra lo suficientemente cerca de la posición deseada actual(posición de la ruta a la que debo ir) el sistema cambia el punto deseado por el siguiente punto en la ruta para que el robot se siga moviendo hacia el objetivo. Esto se hace para cada posición calculada en la ruta óptima exceptuando la posición final para la cual no hay posición siguiente en la ruta y por ende no se puede cambiar la variable *posicionDeseada*.

Para el punto final se utiliza el mismo error que se utilizo en el punto 4 del taller anterior, es decir, un error de "precisión"que indica cual es el máximo cambio que puede haber entre la posición final del robot y la ingresada por parámetro al arrancar el nodo.

En especifico, el error de "precisión"del robot es de 0.05m con 0,02° y el error de "visión.es de 2*$tamanoGrilla$. Donde la variable $tamanoGrilla$ modela el tamaño de una grilla en metros(ancho,largo). Por ejemplo, si es de 1 significa que por cada $metro^2$ en el mapa se tiene una casilla en la grilla, mientras que si es de 0.5 significa que por cada medio metro al cuadrado $(0{,}5 * metro)^2$ del mapa se tiene 1 casilla en la grilla, o lo que es lo mismo, por cada metro cuadrado de mapa se tienen 4 casillas. Se utiliza una diferencia de 2 casillas con el fin de que el robot tenga el tiempo suficiente de cambiar su dirección y velocidad para dirigirse hacia el nuevo punto, el escalado de 2 se halló después de probar con diferentes valores y encontrar que era el mejor para el caso tratado.

Por otro lado, la ley de control, o mas específicamente sus constantes, fueron cambiadas para ser inversamente proporcionales al tamaño de la grilla del mapa. Se dejaron así porque al hacer las casillas del mapa muy pequeñas,o lo que es igual, hacer la grilla del mapa muy precisa, el error entre punto(nodo) y punto(nodo) era muy pequeño y la ley de control hacia que el robot se moviese muy lento hacia el objetivo; pero al dejarlas inversamente proporcionales se logra que sin importar la precisión de la grilla el robot pueda avanzar a una velocidad media y constante que le permita alcanzar el objetivo en un tiempo promedio suficientemente bueno.

Es importante tener en cuenta que si la variable $tamanoGrilla$ es menor a 0.5 el robot no çabría.en una casilla de la matriz del mapa, por lo que el entender hacia donde moverse se volvería mas complicado y puede llevar a problemas; en especifico, se noto que para valores del tamaño de la grilla menores o iguales a 0.3 el programa puede o no funcionar dependiendo de que tan rápido arranque el robot.

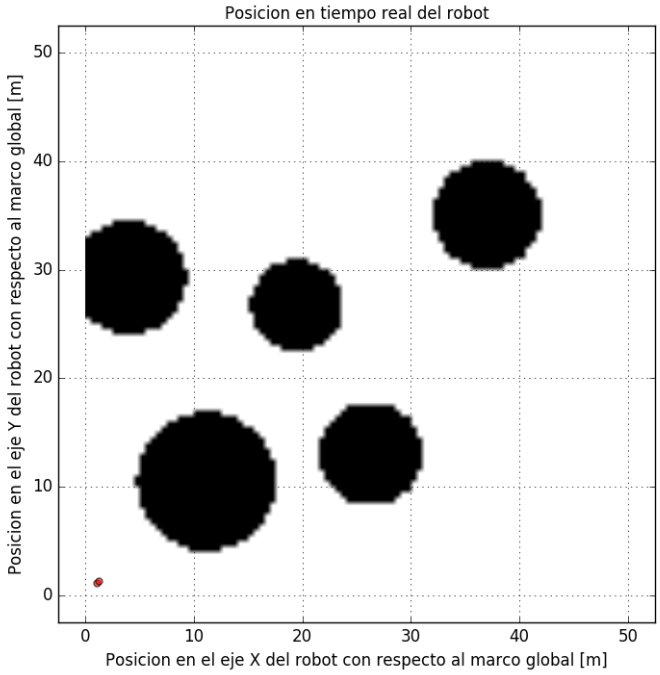


Figura 10: Mapa de la grilla junto con el robot

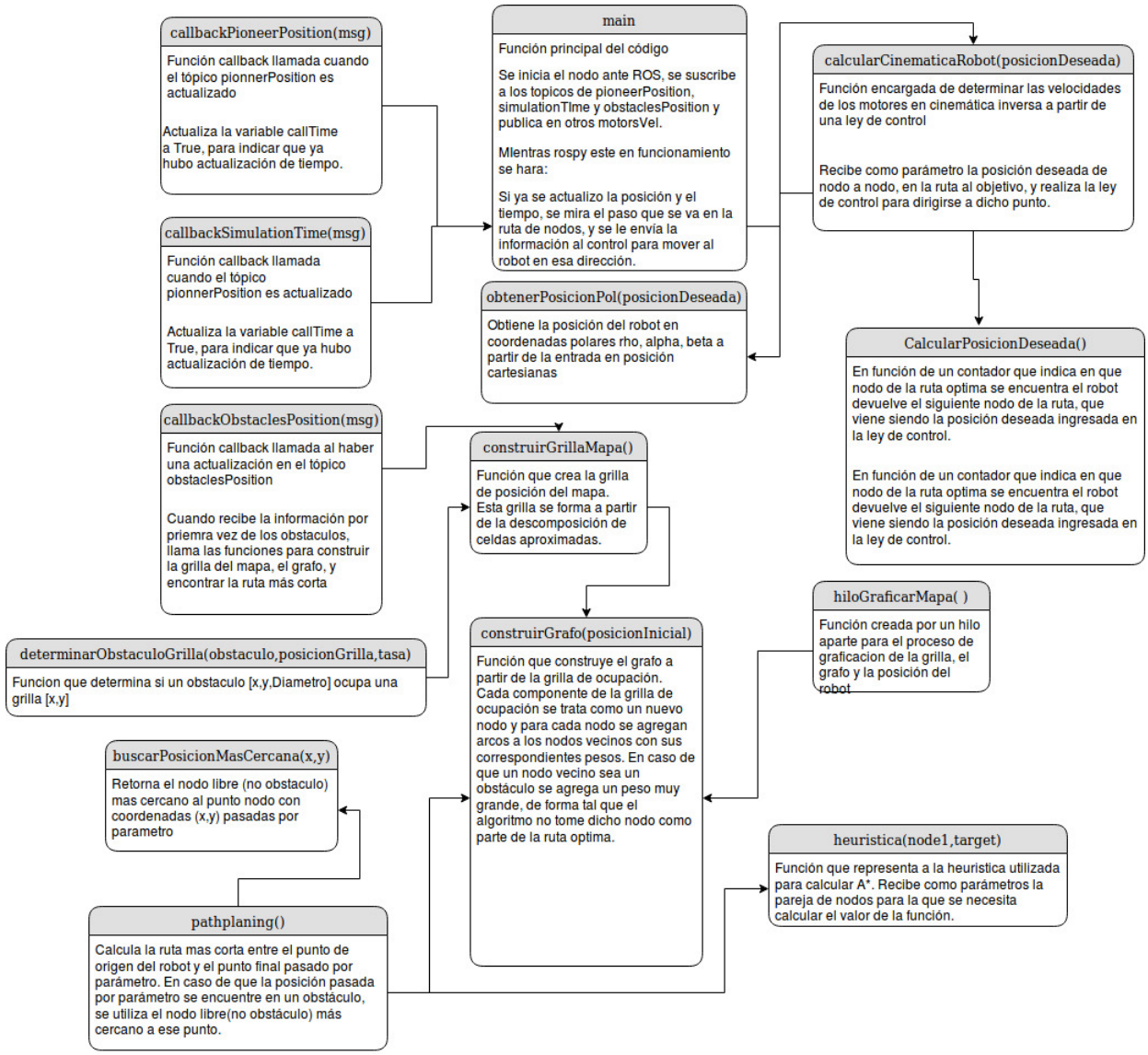


Figura 11: Diagrama de bloques del punto 2.c

e-f)

Para este punto se tenia que diseñar un código que creara la grilla del mapa y un árbol por RRT. Una vez se ha creado el árbol, se busca la ruta más corta en la serie de nodos, y luego se ejecuta el sistema de control para llevar al sistema al punto final.

El código en su inicio, inicia el nodo ante ROS como *P2e_RRT*. Luego de esto se suscribe a los nodos *pioneerPosition*, *simulationTime* y *obstaclesPosition*, asociando a cada suscripción una función callback que se ejecutan cuando ocurre una actualización de la información en estos tópicos. Al mismo tiempo, se realiza la publicación en el nodo *motorsVel*, que actualiza la información de la velocidad de los motores de V-REP. A continuación se muestra el diagrama de nodos de ROS:

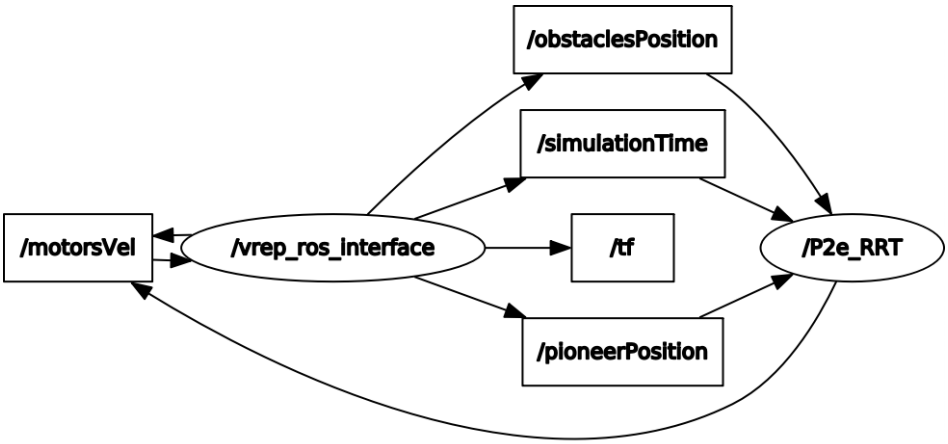


Figura 12: Diagrama de nodos de ROS del punto 2e.

Una vez se ha realizado el inicio del código, este se mantiene ejecutando con una tasa de $20Hz$. En este, se pregunta si se tiene una ruta ya calculada, y si ya se ha actualizado la información de tiempo y posición del robot, con objeto de verificar el movimiento del robot en V-REP. Si esto se cumple, el código mira paso el actual en la serie de pasos de nodos al objetivo final, y manda dicha posición al calculador de control de la cinemática del robot.

La primera vez que llega la información de los obstáculos se construye la grilla del mapa, el grafo del mapa y se calcula la ruta más corta del grafo. Para la creación de la grilla de mapa, se crea una matriz de tamaño: $tamanoMapa/tamanoGrilla$. Para este caso dicha matriz es de 50x50 celdas. El código realiza un recorrido de cada grilla del mapa, y determina si su posición coordinaría con un obstáculo. Así, se termina creando una matriz de 0 y 1, donde 0 corresponde a una grilla sin obstáculo, y 1 una con obstáculo.

Luego de esto se ejecuta el código para crear el árbol por RRT. El árbol inicia uniendo un nodo de la posición inicial del robot, supuesta en (0,0). Luego de esto, se crean puntos aleatorios con distribución uniforme de 0 a 50, tanto para x como para y. Para este punto aleatorio, primero, se busca el nodo más cercano dentro del grafo. Luego, se proyecta una linea de este punto más cercano, y se empieza a acercar un punto desde esa posición al punto aleatorio. Si en dicho recorrido se encuentra con un obstáculo, se detiene el recorrido y se crea un arco del punto cercano a este punto. Si no se encuentra obstáculo, se crea un arco del punto cercano al punto aleatorio originalmente creado. Este procedimiento se realiza para 300 iteraciones, cubriendo de una manera aproximada el mapa. Al final, se busca el nodo más cercano al punto final pasado por parámetro, y se crea un arco entre dichos nodos. Así, se tiene por completo el arco completo.

Una vez se ha culminado el proceso de creación del mapa y arbol, se encuentra la ruta más dentro del grafo del punto inicial al punto final, haciendo uso de la librería networkX, método *short_path*. Este método retorna un vector con la serie de los nombres de los nodos que se tienen que recorrer en dicha ruta. Dicho vector se guarda de forma global, y es utilizado en la serie de control. Al mismo tiempo que se lanza esta función, se lanza un hilo que permite la graficación de la grilla, el grafo y el punto del robot. A continuación se puede evidenciar una captura de dicha gráfica:

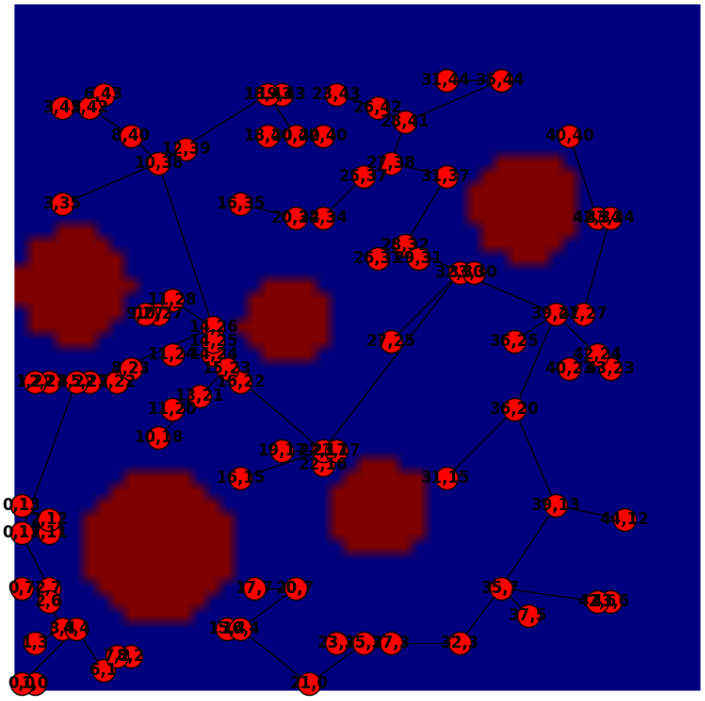


Figura 13: Gráfica de la grilla y del grafo creados por el código.

Por último, se tiene la ley de control. Esta recibe por parámetro la *Posición final*, que corresponde en realidad a un nodo en la serie de pasos de la ruta más corta calculada anteriormente. Así, una vez el robot llega a un nodo, se aumenta el *paso*, y se le envía a la ley de control un nuevo objetivo. Esto se realiza iterativamente en la serie de pasos, hasta llegar al punto final, donde la ley de control cambia dependiendo si la magnitud al punto final ya supero un error. Aquí se ajusta la ley de control para que el control haga rotar el robot al anguilo dado. Así, el control hace llegar al robot a la posición final. A continuación se puede evidenciar el diagrama de bloques total del código:

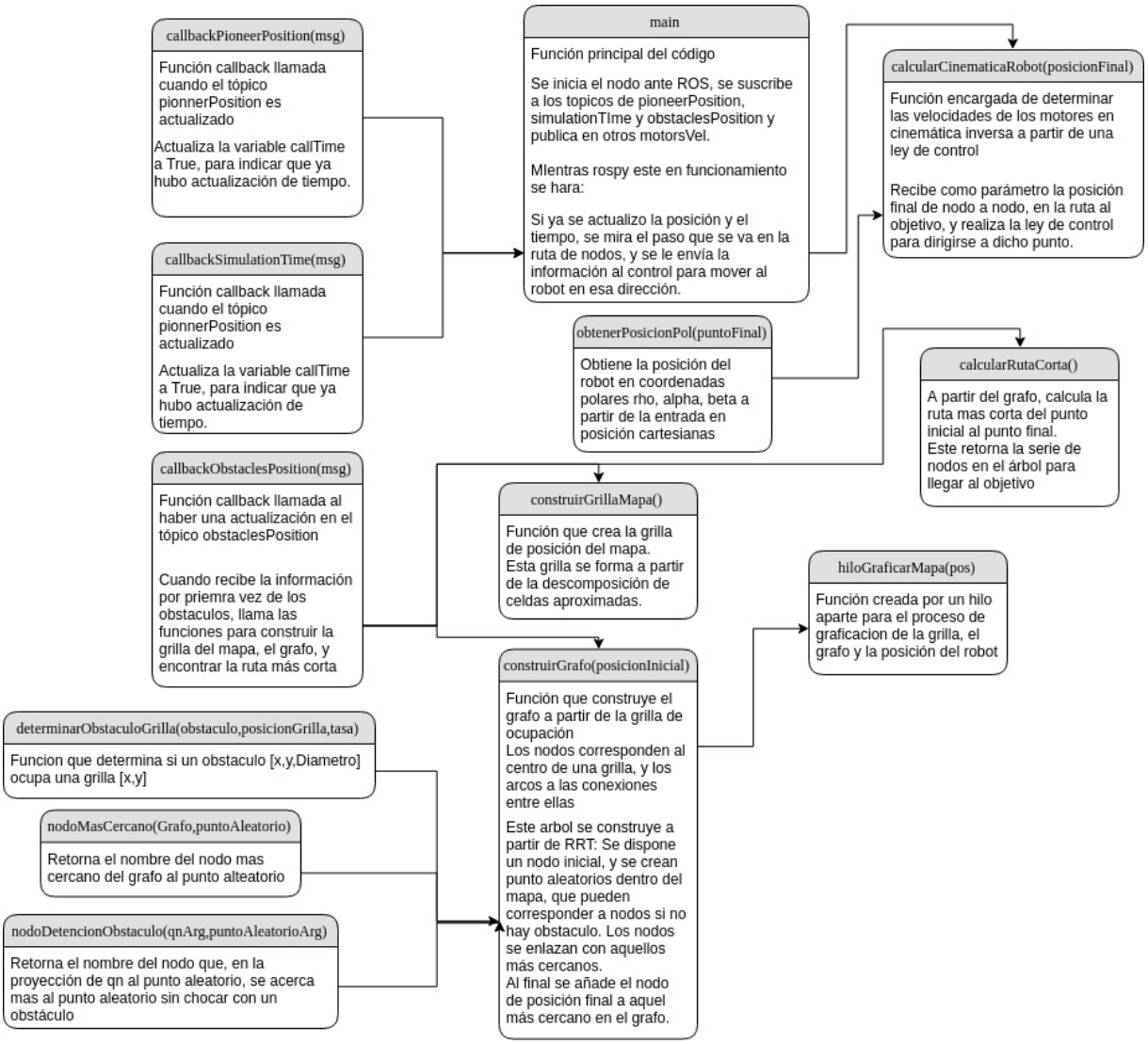


Figura 14: Diagrama de bloques del punto 2e.

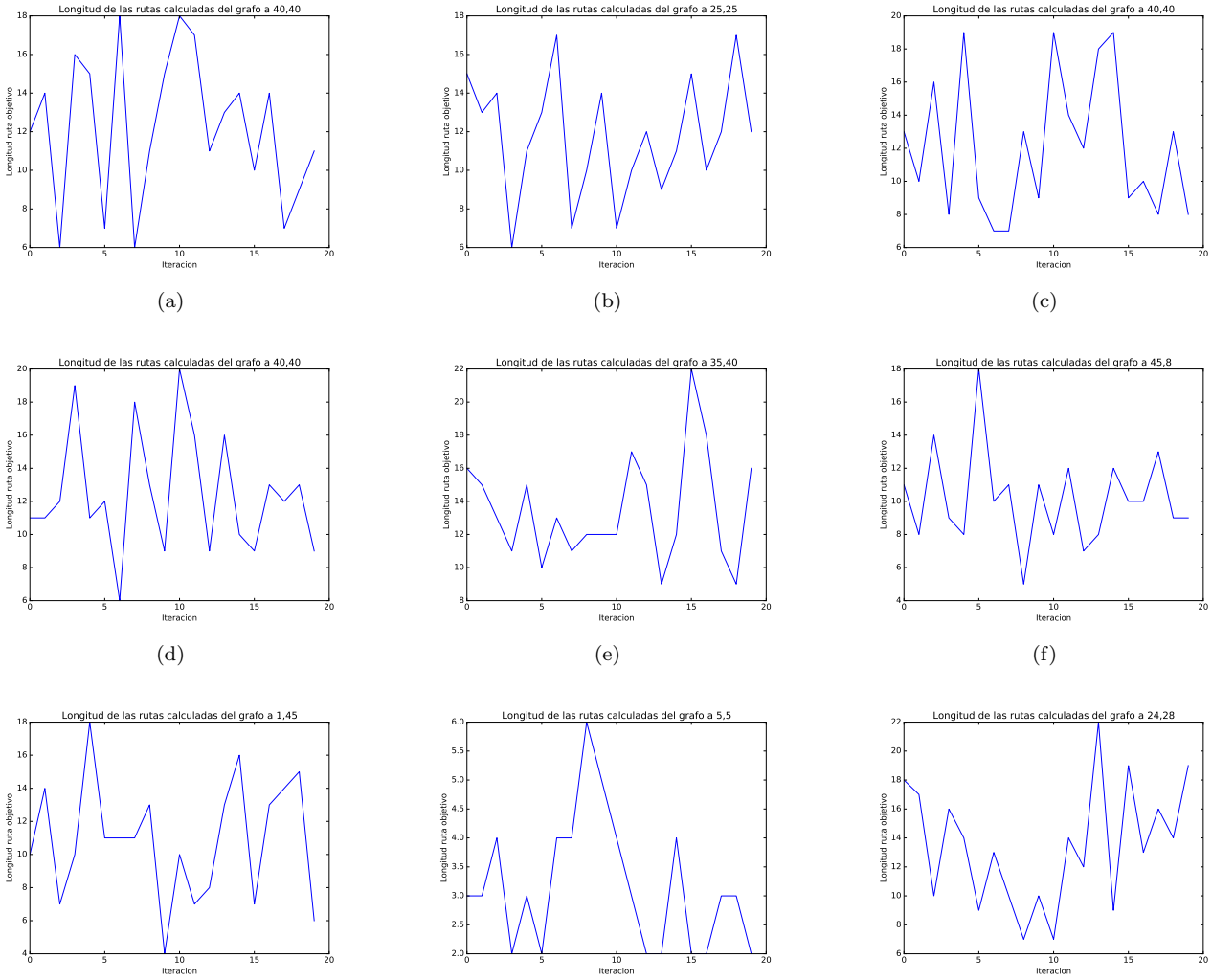
g)

Para este punto, se tenia que ejecutar la creación del grafo por RRT del anterior punto, 20 para cada iteración en la ruta al destino. En este caso, debido a que el grafo se calcula una sola vez al inicio del recorrido, se realizo el calculo del grafo para 9 pares orígenes - destino. El origen se mantuvo fijo en (10,10), y el final se vario. Por cada par, se realizo el calculo 20 veces. Por cada par, se realizo una gráfica de la magnitud de la distancia de la ruta más corta, y el tiempo de ejecución para cada creación de grafo. Así mismo, se obtuvo una tabla de la media y las desviaciones estándar de dichos valores. Esta se puede ver a continuación:

Punto Final	Magnitud ruta – Media	Magnitud Ruta – Desviación	Tiempo Calculo- Media	Magnitud Ruta – Desviación
[10,10]	12,2	37,496,666,519	18,295,595,288	7,145,476,214
[25,25]	11,75	30,475,399,915	369,173,733	10,115,986,982
[40,40]	12,05	41,167,341,425	5,172,298,801	10,189,022,249
[25,10]	12,45	35,982,634,701	70,792,604,804	10,585,648,791
[35,40]	13,45	32,011,716,605	91,139,627,576	12,450,805,484
[45,8]	10,15	27,617,928,959	104,912,484,169	14,383,689,197
[1,45]	10,9	35,623,026,261	119,804,852,366	26,295,945,118
[5,5]	3,15	11,079,259,903	13,998,567,009	226,867,252
[24,28]	13,45	41,409,539,964	156,321,096,897	15,446,533,517

Figura 15: Tabla de valores para los 9 pares de puntos, ejecutados 20 cada uno.

A continuación se puede evidenciar las gráficas de las magnitudes y de los tiempos de calculo por cada iteración para los nueve diferentes pares de puntos:

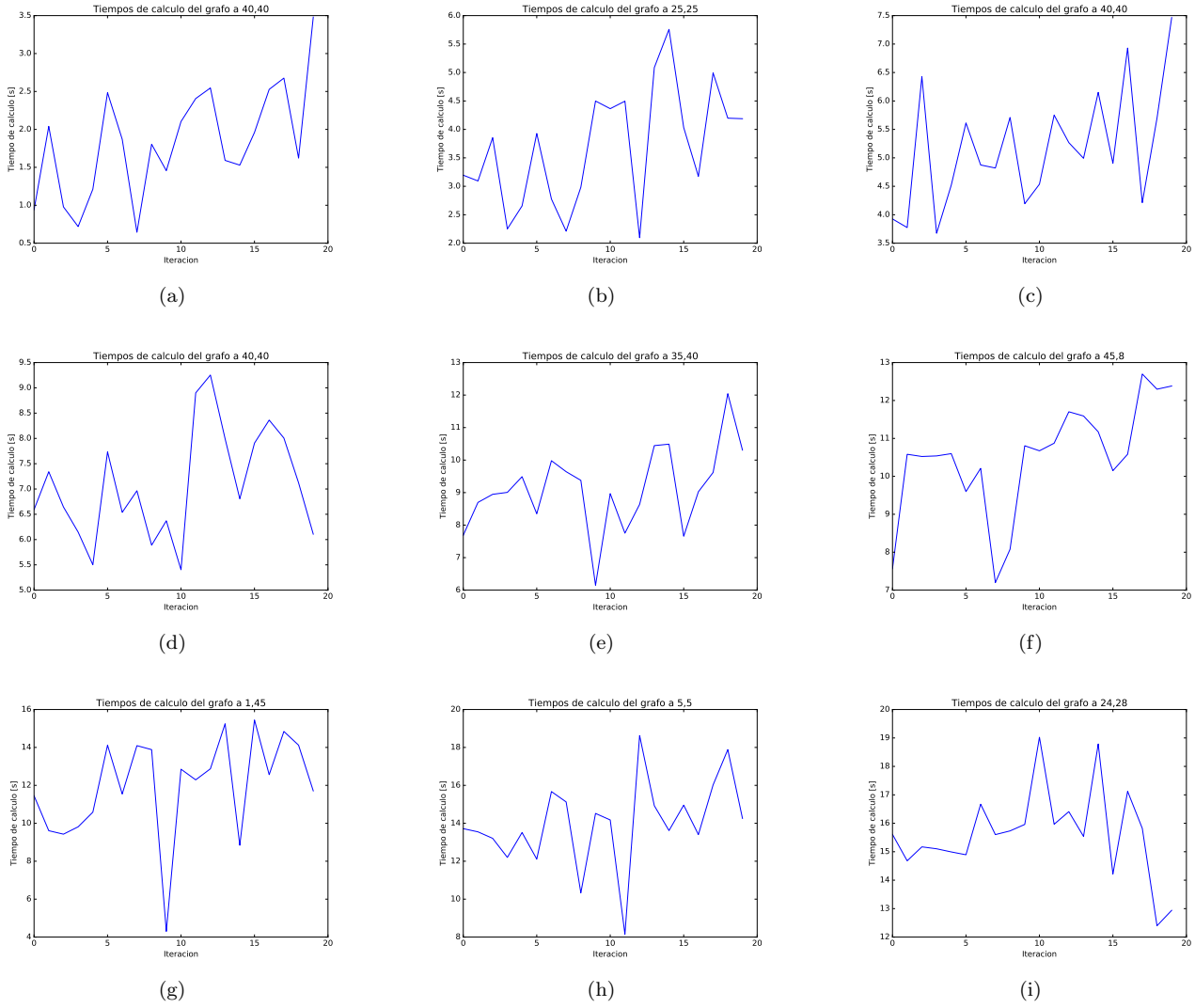


(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

Figura 16: Gráficas de tiempos de calculo para los 9 diferentes sets de puntos, graficados para las 20 diferentes iteraciones por cada uno.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

Figura 17: Gráficas de magnitud para los 9 diferentes sets de puntos, graficados para las 20 diferentes iteraciones por cada uno.

3. Punto 3

3.1.

(0.5)

3.2.

(0.35)

3.3.

(0.9)

3.4.

(0.35)