ESTADO DEL ESTADO DEL ARTE DE LOS ALGORITMOS DE NAVEGACIÓN EN DOS DIMENSIONES

SERGIO DAVID LEÓN BELTRÁN

FUNDACIÓN UNIVERSITARIA LOS LIBERTADORES
FACULTAD DE INGENIERÍA
INGENIERÍA ELECTRÓNICA
BOGOTÁ DC
2017

ESTADO DEL ESTADO DEL ARTE DE LOS ALGORITMOS DE NAVEGACIÓN EN DOS DIMENSIONES

SERGIO DAVID LEÓN BELTRÁN

Trabajo de grado para optar por el título de Ingeniero Electrónico

Director
Iván Darío Ladino Vega
Ingeniero Electrónico

FUNDACIÓN UNIVERSITARIA LOS LIBERTADORES
FACULTAD DE INGENIERÍA
INGENIERÍA ELECTRÓNICA
BOGOTÁ DC
2017

Nota de aceptación

_____

_____
_____
_____

_____
Firma
Nombre:
Presidente del jurado

_____
Firma
Nombre:
Jurado

_____
Firma
Nombre:
Jurado

Bogotá DC, 18 de septiembre de 2017

A mis padres por su apoyo incondicional y demás personas que siempre amablemente ayudaron a mi formación.

AGRADECIMIENTOS

Agradezco al docente Ingeniero Iván Darío Ladino su confianza, apoyo y ayuda para llevar a cabo este trabajo. También al docente Nelson Lozano y al docente Ingeniero John Anzola por su amable colaboración.

ÍNDICE

ÍNDICE DE FIGURAS

Pág

ÍNDICE DE TABLAS

Pág

ÍNDICE DE GRÁFICAS

Pág

RESUMEN

Se consultó sobre los algoritmos fundamentales para la navegación de robots móviles en un espacio de configuración bidimensional (BCS) que operan con completa o incompleta información, sobre pistas de navegación establecidas y con presencia de obstáculos fijos o dinámicos. Para este documento se incluyeron los algoritmos de la familia Bug que garantizan una terminación en tiempo finito los cuales son el Bug1, Bug2, Alg1, Alg2, DistBug y TangentBug. Algunos algoritmos no pertenecientes a la familia Bug también se incluyeron como el Dijkstra, A* (A estrella), D*, Com, Class1, Rev1, Rev2, Curv1, Curv2, Curv3, OneBug, MultiBug y LeaveBug, considerando lo que ofrece cada uno en la navegación autónoma, su principio de funcionamiento y su pseudocódigo. Se presentaron los resultados de la comparación entre los algoritmos Bug para dos BCS diferentes, con los que pudo observarse la respuesta que estos algoritmos tuvieron en cuanto a la longitud de sus rutas de navegación, teniendo en cuenta los resultados de las simulaciones del PhD James Ng. También se incluyeron algunos métodos de finalización de los algoritmos Bug, así como un método de navegación segura y determinación de una ruta corta de aproximación al objetivo por medio de la parametrización de curvas.

Se concluyó sobre las ventajas que tienen ciertos algoritmos con respecto a otros por medio de una comparación entre el Bug1, Bug2, Alg1, Alg2, DistBug y TangentBug, D*, Class1, Rev1, Rev2 y la importancia de comprender las condiciones en las que un algoritmo de navegación va a operar. Se analizaron los resultados de las simulaciones del PhD en Ingeniería James Ng de los algoritmos de navegación en dos espacios de configuración. Para rutas de navegación cortas se consideraron los algoritmos TangentBug y D* en BCS abiertos como los mejores a elegir, pero sus costos en cuanto a requerimiento en memoria y complejidad son considerables. Para soluciones económicas, simples e incluso para tareas de navegación en las que no sea muy exigente la longitud de ruta, los algoritmos Bug1 y Bug2 no se quedaron atrás en cuanto a su competitividad por ser una gran opción. El algoritmo Bug2 puede ser empleado para BCS cerrados, y para los abiertos el algoritmo Bug1 podría ser elegido. Al considerar que el robot operará en presencia de mínimos locales, el algoritmo TangentBug debería ser elegido. Para un BCS cerrado en el que la tarea de navegación tolere ciertos errores, el algoritmo Alg2 puede operar. De lo contrario, el DistBug sería el adecuado. Para una tarea de navegación en la que se tengan pistas de navegación preestablecidas como por ejemplo a manera de cuadrícula o una hoja de ruta, el algoritmo Curv3 operaría para esta tarea incluso en presencia de obstáculos dinámicos.

INTRODUCCIÓN

Los algoritmos de navegación le permiten a un robot tener cierto grado de autonomía en una tarea que requiere desplazamiento a un punto específico. Considere por un momento como para algunas personas el desplazarse caminando puede ser fácil, pero el desplazarse con autonomía para un robot depende de la elaboración de un procedimiento de navegación.

El reemplazar al humano en algunas tareas de navegación puede llegar a considerarse en el futuro [1]. Tareas como el transporte de elementos pesados, navegación en lugares geográficos expuestos a elementos nocivos, desplazamiento turístico de personas, entre muchas más, fomentan el desarrollo de algoritmos de navegación cada vez más eficientes y confiables.

En este documento se presentan los algoritmos de navegación básicos para espacios de configuración bidimensionales (BCS) que se consideraron fundamentales para iniciar el diseño de un algoritmo en la tarea de navegación autónoma.

JUSTIFICACIÓN

Tener conocimiento de la existencia de algoritmos básicos para la navegación en un BCS es fundamental para metodológicamente emprender la navegación autónoma bidimensional. Para quienes desean iniciarse o ya han hecho parte de implementaciones de navegación autónoma bidimensional, este documento contiene información que bien puede ser empleada como base de un proceso de aprendizaje, sustentación de técnicas de navegación bidimensional para informes escritos, elemento de consulta para la selección adecuada de un algoritmo de navegación para un BCS, e incluso con fines informativos. Saber que existen técnicas en la navegación autónoma bidimensional que han sido simuladas, a las que se les ha comparado, de las que se han generado conclusiones sobres sus características, es algo que se ofrece al lector de este documento, teniendo como referencias bibliográficas publicaciones de la IEEE y universitarias, así como textos especializados en ingeniería.

OBJETIVOS

OBJETIVO GENERAL

Ofrecer al lector un mapa de características en el cual se valore la eficiencia entre los algoritmos de navegación bidimensional Bug1, Bug2, Alg1, Alg2, DistBug, TangentBug, Dijkstra, A*, D*, Com, Class1, Rev1, Rev2, Curv1, Curv2, Curv3, OneBug, MultiBug y LeaveBug, a partir de la información consultada en las fuentes bibliográficas confiables que se tuvieron en cuenta para este documento, que incluyen resultados de investigación y estudios de los algoritmos de navegación bidimensional por parte del PhD en Ingeniería James Ng.

OBJETIVOS ESPECÍFICOS

- Consultar información concerniente al principio de funcionamiento y pseudocódigo de los algoritmos de navegación bidimensional Bug1, Bug2, Alg1, Alg2, DistBug, TangentBug, Dijkstra, A*, D*, Com, Class1, Rev1, Rev2, Curv1, Curv2, Curv3, OneBug, MultiBug y LeaveBug, tanto de publicaciones en la IEEE como universitarias, y también en libros de texto especializados en ingeniería.
- Generar conclusiones sobre los algoritmos consultados a partir del análisis de resultados de simulaciones, comparaciones y opiniones realizadas el PhD en Ingeniería James Ng.
- Sustentar la valoración de eficiencia aplicada al mapa de características de los algoritmos de navegación consultados, de tal manera que el lector encuentre conceptos básicos y referencias para una mejor comprensión y acceso a información más detallada del algoritmo de su interés.

1. INTRODUCCIÓN A LOS ALGORITMOS DE NAVEGACIÓN

Un espacio de configuración bidimensional (BCS) para un robot es lo que comúnmente se le considera al espacio geográfico en el que va a navegar. Para la navegación autónoma, este BCS contiene un punto de inicio o de partida y un punto final u objetivo. Se establece que puede existir una cantidad finita de obstáculos de área finita, y la finalidad del robot es encontrar una ruta continua libre de estos obstáculos.

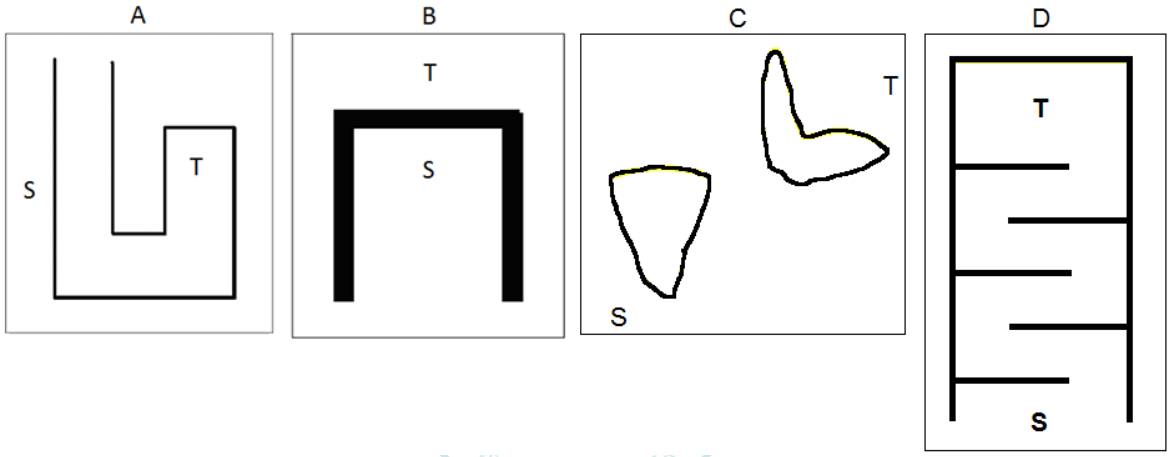


Figura 1.1: BCS empleados en la simulación de algoritmos de navegación [11, 12].

El objetivo principal de los algoritmos de navegación es el de guiar al robot desde el punto de inicio S (start) al punto de destino T (target), y para esta consideración general se asume que el robot se encuentra en un BCS desconocido. El robot debe lograr esta tarea con la menor información global posible. Esto significa que el robot puede memorizar puntos de interés, más no una tarea de mapeo. De no encontrarse una ruta posible, el algoritmo finaliza y reporta que llegar al punto de destino es imposible. A esto se le considera terminación o finalización [1].

1.1. EL MODELO BUG

Los algoritmos Bug pueden ser programados en cualquier tipo de robot con sensores táctiles o de proximidad (alcance), y emplear métodos de localización como lo pueden ser odómetros, reconocimiento de terreno o GPS. Con esto se desea que el robot encuentre de manera autónoma una ruta que lo lleve al punto final deseado.

Considerando que el BCS pueda cambiar continuamente, es decir, que obstáculos dinámicos se encuentren en este, la variación en la información sobre el BCS puede ser conocida en cualquier instante de tiempo [13]. La estrategia de navegación debe lograr que el robot logre su tarea con la menor cantidad de información posible, preferiblemente solo la información de posición actual y el punto de destino*.

Para el modelo Bug se asumen tres condiciones simples sobre el robot [1]:

1. El robot es un es un punto objeto. Esto significa que este no tiene dimensiones que interfieran con el BCS y se pueda mover entre espacios arbitrarios, sin problema a quedar atrapado.
2. El robot posee una excelente capacidad de orientación. Gracias a esto el robot sabe su posición y orientación relativa al origen en cualquier momento. Se determina con precisión la distancia y la ubicación con respecto al punto de destino, para garantizar si cumplir la tarea es posible.
3. El robot cuenta con sensores de excelente calidad. Algunos algoritmos de navegación dependen de la información obtenida con estos sensores, como la distancia para la navegación. Los sensores defectuosos afectarán el rendimiento del robot debido a fallos en el algoritmo.

Claramente las tres condiciones anteriores son ideales e irreales para un robot que efectuará una tarea en un BCS físico. Es lógico pensar que debido a estas condiciones los algoritmos Bug no pueden ser empleados para la navegación en un robot real, pero pueden ser empleados como un componente de supervisión de alto nivel en un sistema que considere estas tres condiciones [11].

* La autonomía en la tarea de navegación se afecta negativamente si el robot está en comunicación continua con un elemento, como lo puede ser una central remota.

2. LOS ALGORITMOS BUG

Los algoritmos Bug son considerados una familia* y estos manejan una notación típica para su empleo [11]:

- H_i = este es el iésimo punto (hit) en la ruta hacia el destino, cuando el robot pasa de dirigirse al objetivo a seguir la frontera del obstáculo.
- L_i = este es el iésimo punto (leave) en la trayectoria hacia el destino, cuando el robot pasa de seguir la frontera del obstáculo a dirigirse al objetivo.
- *S* = punto de inicio (start).
- *T* = punto de destino (target).
- *x* = Posición actual del robot.
- *d(a, b)* = Distancia Euclidiana entre los puntos arbitrarios a y b.
- $d_{ruta}(a, b)$ = Distancia de la ruta del robot entre los puntos arbitrarios a y b.
- *r* = Rango máximo del dispositivo sensible a la posición PSD (Position Sensitive Device).
- *r(ɵ)* = Rango libre en una dirección ɵ. La distancia entre en robot y el primer obstáculo en la dirección ɵ.
- *F* = Espacio libre en la dirección T (target). Donde F = r(ɵ), donde ɵ es la dirección hacia T.

2.1. FAMILIA DE ALGORITMOS BUG

2.1.1. Algoritmo Bug1

Fue el primer algoritmo creado en esta familia por V. Lumelsky y A. Stephanov [1], y su principio de funcionamiento se basa en buscar cada obstáculo en el punto más cercano al objetivo *T*. Una vez ese punto más cercano es establecido, el robot evalúa si puede dirigirse directamente al objetivo o no. De no ser posible, el robot determina que el objetivo es inalcanzable [14]. De lo contrario, el robot dejará ese punto como muestra de que dirigirse directamente hacia el objetivo es posible, marcando este punto y almacenándolo en sus datos para no volver a él en su navegación [13]. Su modo principal de operación ante un obstáculo es el *boundary-following* o seguidor de frontera, que es seguir el contorno del obstáculo. La representación del funcionamiento de este algoritmo se observa en la figura 2.1. A continuación se muestra el pseudocódigo para este algoritmo de navegación [11]:

* Considerados familia especialmente porque garantizan una finalización en tiempo finito del algoritmo y sus dos modos de operación *motion-to-goal* y *boundary-following* [1, 13].

0. Iniciar la variable *i* en *0 (i = 0)*.
1. Incrementar i y moverse hacia el objetivo T hasta que ocurra una de las siguientes condiciones:
    - El robot llegue a T = Finalizar satisfactoriamente.
    - El robot encuentra un obstáculo = marcar este punto como H_i y proceder al paso 2.
2. Mantener el obstáculo a la derecha del robot y moverse bordeándolo. Mientras esto se realiza, se almacenan los puntos $d_{ruta}(H_i, x)$ donde $d(x, T)$ es mínimo y el robot pueda ir de x a T directamente desde aquí. Uno de esos puntos mínimos se almacena como L_i. Una vez el robot vuelva a pasar por H_i, se analiza si el robot puede ir de L_i hasta T. Si el robot no puede dirigirse a T, entonces termina el algoritmo notificando que el destino es inalcanzable. Pero si el robot puede dirigirse hasta T, se elige una dirección donde el robot estará paralelo al obstáculo (en este caso pared) y esta dirección es la menor en $d_{ruta}(H_i, x)$, para dirigirse hacia T. En L_i se regresa al paso 1.

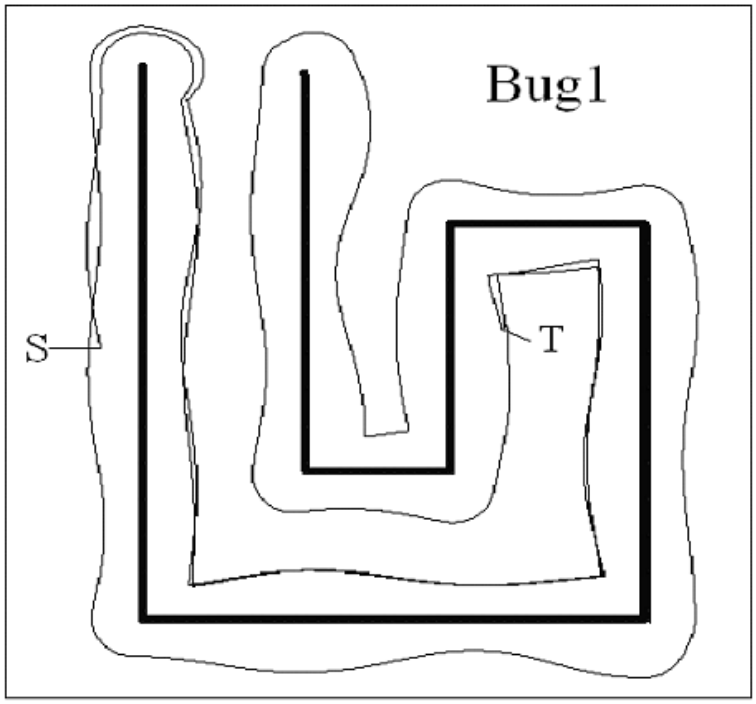


Figura 2.1: Representación del algoritmo Bug1 en un BCS A [12].

2.1.2. Algoritmo Bug2

Este algoritmo de navegación también fue creado por V. Lumelsky y A. Stephanov [1]. Para el algoritmo Bug2, se tendrá una línea imaginaria *M* directa desde *S* hasta *T*. Este algoritmo es menos estricto que el Bug1, ya que el robot puede partir del obstáculo (*leave*) hacia el destino debido a *M* [13]. Este algoritmo también emplea el modo seguidor de frontera y al partir el robot hacia *T*, se le considera en modo *move-to-goal* o moverse al objetivo. Se representa este algoritmo en la figura 3.1 y a continuación se muestra el pseudocódigo para este algoritmo de navegación [11]:

0. Se dibuja una línea imaginaria *M*, directamente desde *S* hasta *T*. Se establece a $i = 0$.
1. Incrementar *i* y desplazarse sobre *M* hacia *T* hasta que:
   - El robot logre llegar a *T* = Finalizar satisfactoriamente.
   - El robot encuentra un obstáculo = Marcar este punto como H_i. Ir al paso 2.
2. Manteniendo el obstáculo a la derecha del robot, seguir el borde de este. Realizar lo anterior hasta que ocurra una de las siguientes condiciones:
   - El robot llegue a *T* = Finalizar satisfactoriamente.
   - Se tiene un punto a lo largo de *M* de manera que $d(x, T) < d(H_i, T)$. Si el robot es capaz de moverse hasta *T*, marque este punto como L_i e ir al paso 1. De lo contrario, actualizar $d(H_i, T)$ *con* $d(x, T)$.
   - Cuando el robot llegue a H_i se considera *T* inalcanzable. Terminar el algoritmo.

Con respecto al cuando el robot puede partir sobre *M* han existido varias interpretaciones, debido a que no son explicitas las condiciones para que se realice la tarea "*leave*" o partir hacia *T* [1, 12]. Se aclara que la línea imaginaria *M* se alcanza a una distancia *d* desde *T* cuando $d < d(H,T)$. Se define el punto de partida "*leave*" como L_j. Se establece $j = j + 1$ y finalmente se inicia el paso 1. Si se lee con cuidado lo anterior se estaría definiendo un punto sin que el robot se mueva. No tiene sentido que a un robot se le permita dejar su posición inicial si no empieza a moverse hacia el objetivo inmediatamente al dejar su posición inicial (*leave*). Por lo tanto, solo se le permitiría moverse si es directamente hacia *T*.

También si al robot no se le permite dejar su posición al no poder moverse hacia *T*, entonces se debería actualizar $d(H_i, T)$ con $d(x\ T)$*. Obviamente si al robot no se le permite dejar su posición debido a que no puede moverse hacia *T*, entonces debe existir un punto en el mismo obstáculo y sobre la línea imaginaria *M* que sea más cercano a *T*.

---

* Una versión mejorada del Bug2 es el Bug2+, cuya única diferencia es que el último actualiza $d(H_i, T)$ cuando ejecuta el paso 1 [12].

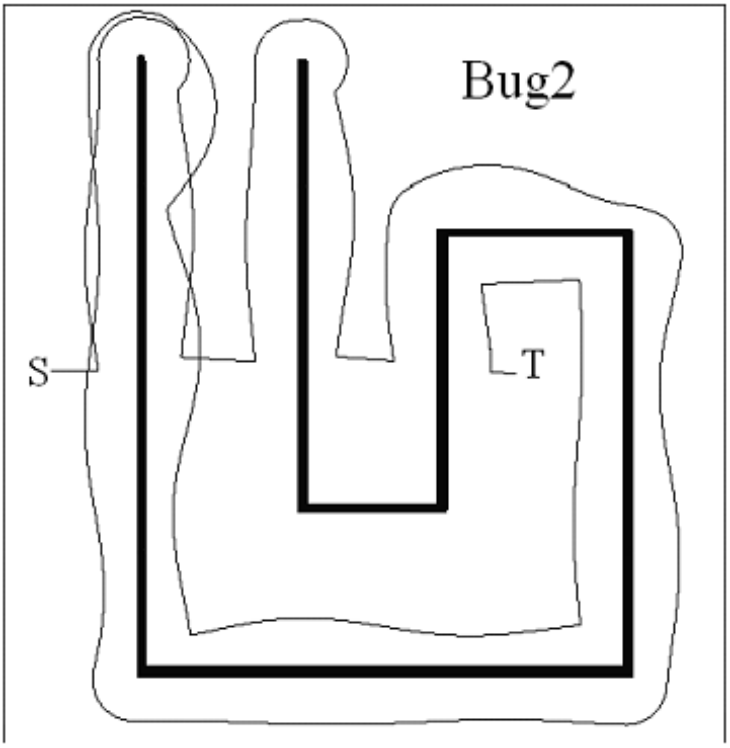


Figura 3.1: Representación del algoritmo Bug2 en un BCS A [12].

2.1.3. Algoritmo Alg1

Este algoritmo fue creado por A. Sankaranarayanan y M. Vidyasagar [2]. El algoritmo Alg1 es una extensión del Bug2. El Bug2 es vulnerable a trazar dos veces la misma ruta y crear rutas largas. Para corregir esto el Alg1 recuerda los puntos previos "*hit*" H_k y "*leave*" L_k, empleándolos para crear rutas cortas. Se muestra la operación de este algoritmo en la figura 4.1 y a continuación se muestra el pseudocódigo para este algoritmo de navegación [11]:

0. Inicialmente se dibuja una línea imaginaria *M* directa desde *S* hasta *T* y se establece $i = 0$.
1. Se incrementa *i* y se sigue la línea M hasta que:
   - El robot llegue hasta *T* = Finalizar satisfactoriamente.
   - El robot encuentre un obstáculo = Se define este punto como H_i y se ejecuta el paso 2.
2. Manteniendo el obstáculo a la derecha del robot, moverse por la frontera del obstáculo. Hacer esto hasta que ocurra una de las siguientes condiciones:
   - El robot llega hasta *T* = Finalizar satisfactoriamente.
   - Un punto y es marcado cuando:
     - Está sobre *M*.
     - $d(y,T) < d(x,T)$ para toda x realizada por el robot a lo largo de *M*.

- El robot puede moverse desde y hacia *T*.

Definir este punto como L_i e ir al paso 1.

- Un punto previo definido como H_i o L_i es encontrado cuando $j < i$. Girar el robot de vuelta y regresar a H_i. Una vez se regrese a H_i, bordear el obstáculo pero esta vez manteniéndolo a la izquierda del robot. Esta condición no puede ser aplicada nuevamente hasta que se defina L_i.
- Cuando el robot retorne a H_i, se considera *T* inalcanzable. Terminar el algoritmo.

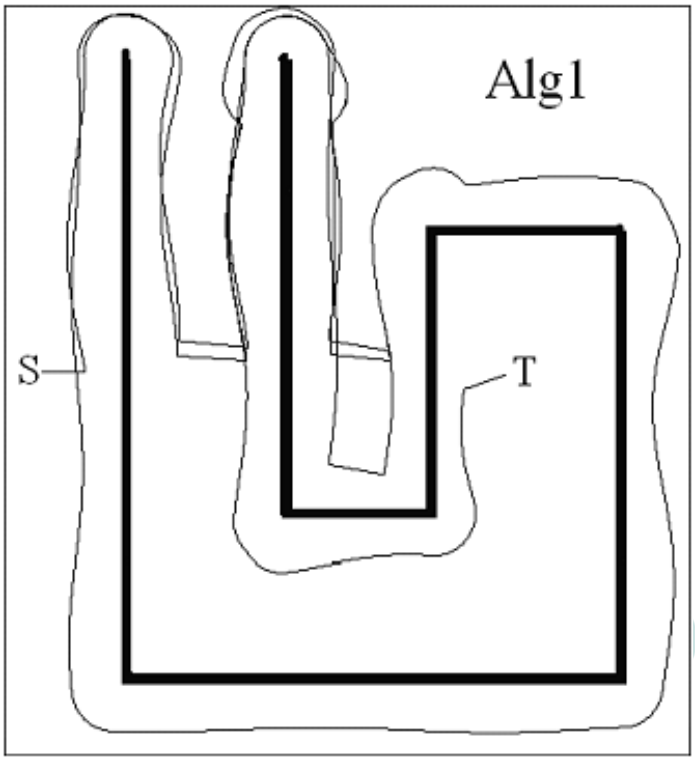


Figura 4.1: Representación del algoritmo Alg1 en un BCS A [12].

2.1.4. Algoritmo Alg2

Creado también por A. Sankaranarayanan y M. Vidyasagar [3], es una mejora del Alg1. El robot abandona el concepto de la línea imaginaria *M*, y una nueva condición de partida "*leave*" se incorpora. La operación del algoritmo en un BCS se muestra en la figura 5.1, y a continuación se muestra el pseudocódigo para este algoritmo de navegación [11]:

0. Iniciar $Q = d(S, T)$ e $i = 0$.
1. Incrementar i y proceder en la dirección T, actualizando continuamente Q con $d(x, T)$ si $Q < d(x, T)$. Q ahora debería representar el punto en el que el robot ha estado más cerca de T. Hacer esto hasta que ocurra lo siguiente:
   - El robot llegue a T = Finalizar satisfactoriamente.
   - Un obstáculo sea encontrado = Marque este punto como H_i y proceder al paso 2.
2. Manteniendo el obstáculo a la derecha, mover el robot cerca al borde del obstáculo y actualizando continuamente Q con $d(x, T)$ si $Q < d(x, T)$, hasta que ocurra una de las siguientes condiciones:
   - El robot llegue a T = Finalizar satisfactoriamente.
   - Un punto y es determinado cuando:
     - $d(y, T) < d(Q, T)$
     - El robot pueda moverse desde el punto y hacia T.

   Definir este punto como L_i y proceder al paso 1.
   - Un punto previo definido como H_i o L_i es encontrado cuando $j < i$. Girar el robot de vuelta y regresar a H_i. Una vez se regrese a H_i, bordear el obstáculo pero esta vez manteniéndolo a la izquierda del robot. Esta condición no puede ser aplicada nuevamente hasta que se defina L_i.
   - Cuando el robot retorne a H_i, se considera T inalcanzable. Terminar el algoritmo.

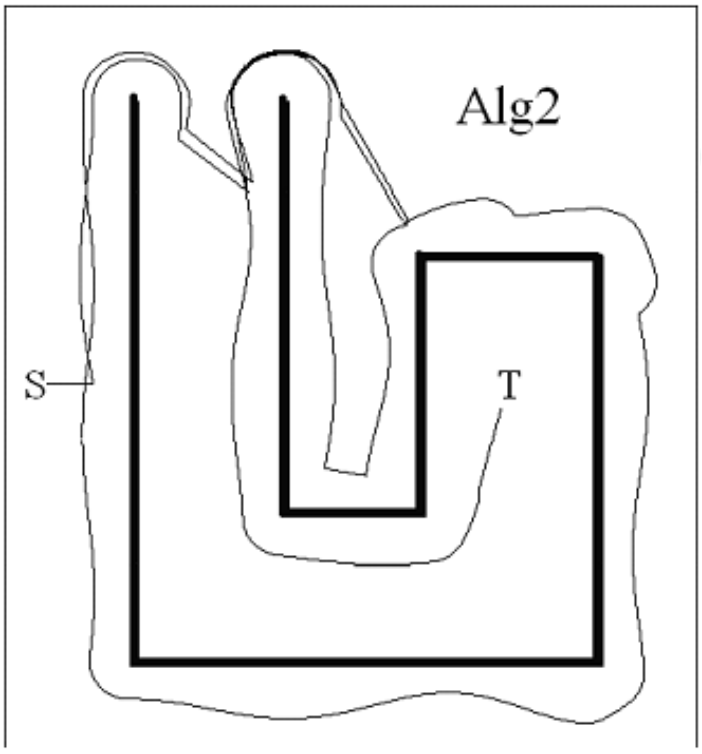


Figura 5.1: Representación del algoritmo Alg2 en un BCS A [12].

La condición "leave" en el Alg2 ha sido una gran mejora porque con ella el robot no necesita estar sobre *M* para dejar el obstáculo, lo cual resulta computacionalmente eficiente. Sin embargo, estas mejoras requieren un método para prevenir el inconveniente del algoritmo no oficial Class1 que se trata en la sección 2.2.5, y en resumen trata sobre el robot al no alejarse adecuadamente de un obstáculo cercano a *T* [12].

2.1.5. Algoritmo DistBug

El algoritmo DistBug usa un sensor de distancia para detectar *F* (espacio libre en la dirección θ hacia *T*) y la emplea para alejarse del obstáculo. Fue creado por I. Kamon y E. Rivlin [4]. La operación del algoritmo se observa en la figura 6.1 y a continuación se muestra el pseudocódigo para este algoritmo de navegación [11]:

0. Iniciar $i = 0$. Existe una condición STEP que especifica el ancho de los obstáculos fijos en el BCS. Con lo anterior, el robot debe moverse cerca a estos obstáculos teniendo en cuenta que la condición STEP es determinada por el usuario, y afecta como el robot se dirigirá al objetivo.
1. Incrementar *i* y proceder a moverse hacia el objetivo, hasta que una de las siguientes situaciones ocurra:
   - El robot llegue al objetivo *T* = Finalizar satisfactoriamente.
   - El robot se encuentre con un obstáculo = Marcar este punto como H_i y proceder al paso 2.
2. Gire a la izquierda y sigua la frontera del obstáculo mientras continuamente se actualiza el mínimo valor de *d(x, T)*, determinándolo como $d_{min}(T)$. Hacer esto hasta que una de las siguientes situaciones ocurra:
   - El objetivo es visible = $[d(x, T) - F] \leq 0$. Denominar este punto como L_i y ejecutar el paso 1.
   - El rango base de la condición "leave" mantiene = $[d(x, T) - F] \leq [d_{min}(T) - STEP]$. Denominar este punto como L_i y ejecutar el paso 1.
   - El robot realizó un bucle y regresó a H_i. Se considera que *T* es inalcanzable. Terminar el algoritmo.

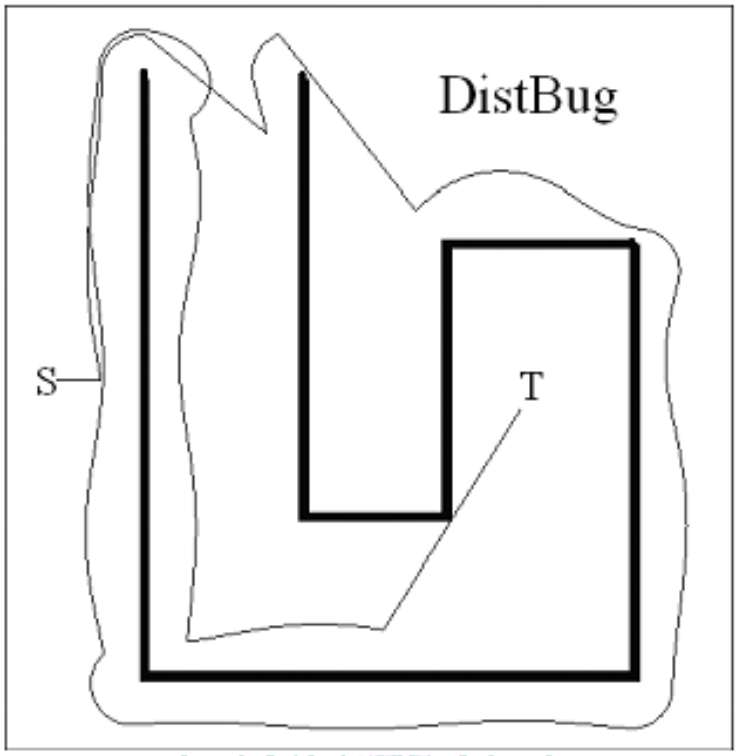


Figura 6.1. Representación del algoritmo DistBug en un BCS A *[12]*.

El algoritmo DistBug puede emplearse para mejorar el rendimiento en la longitud de una ruta a seguir, debido a que cada vez analiza la condición "*leave*" [11] basándose en dos parámetros [12, 15]:

1. El robot puede usar su sensor de distancia para detectar un punto cercano a T jamás alcanzado o visitado.
2. El criterio STEP es empleado para evitar que el robot no actúe eficientemente ante un obstáculo y no alejarse apropiadamente de él.

La sección 4.1.4 de en este documento trata el criterio STEP, pero puede resumirse como la partida del robot únicamente si está en una distancia predefinida, dentro de una región determinada.

2.1.6. Algoritmo TangentBug

Este algoritmo fue creado por I. Kamon, E. Rivlin y E. Rimon [5]. El algoritmo TangentBug hace uso de sensores para construir una gráfica de los alrededores del robot, para minimizar la longitud de la ruta a seguir [13]. Para comprender mejor el

funcionamiento de este algoritmo, es necesario tener en cuenta varios conceptos asociados a su funcionamiento [12]:

2.1.6.1. La gráfica tangencial global (LTG)

La figura 7.1 muestra un BCS (superior izquierda) en el que los obstáculos tienen vértices y estos se observan marcados con círculos naranja a manera de nodos (superior derecha). Cuando se tienen las posibles rutas (denominada la ruta tangencial global, inferior izquierda) que unan estos vértices para llegar al objetivo, siempre se elegirá la más optima [8, 13, 14] (inferior derecha) desde el inicio (cuadrado verde) al final (cuadrado rojo).

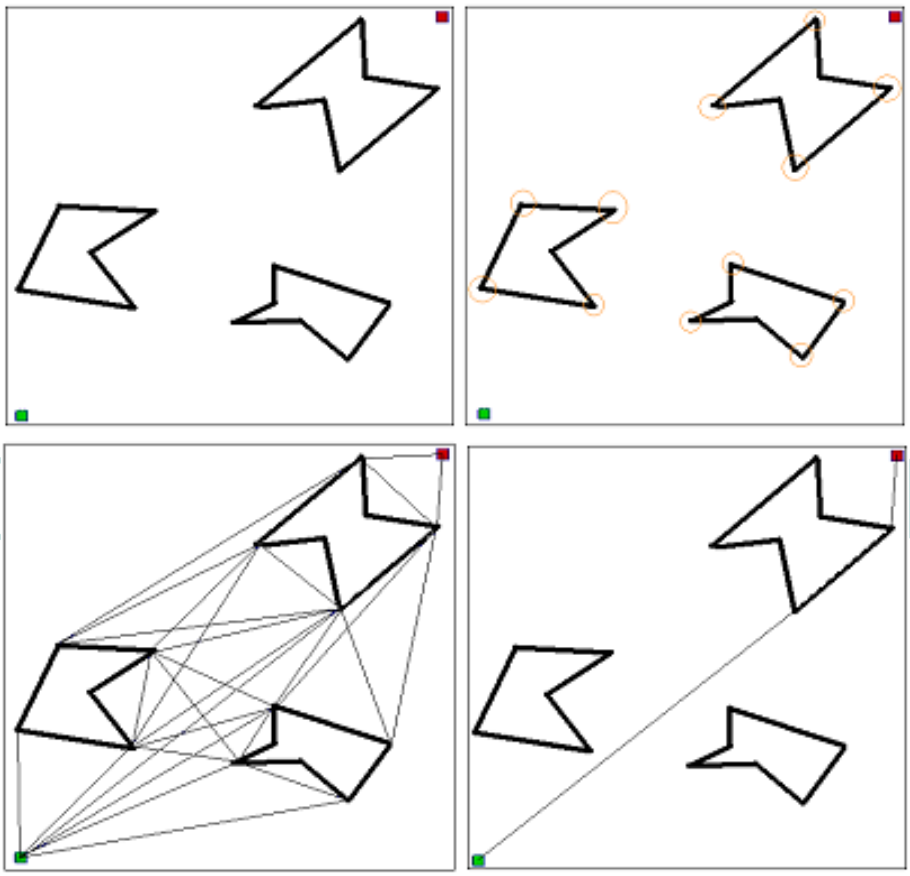


Figura 7.1: BCS de obstáculos con vértices y representación de la ruta tangencial global [12].

La figura 7.2 muestra la gráfica local tangencial, que depende del rango de alcance del sensor en el robot.

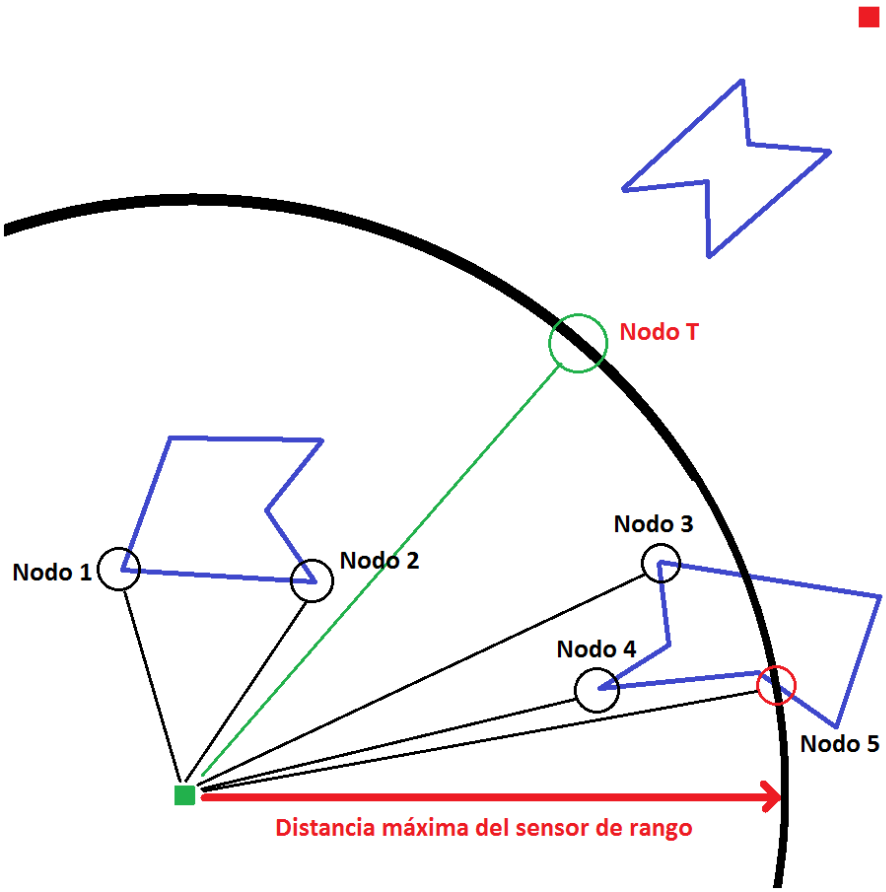


Figura 7.2: La gráfica local tangencial LTG.

Para la figura 7.2, la LTG se genera recopilando datos de la función $r(\theta)$ y F. $r(\theta)$ retorna la distancia del primer obstáculo visible en la dirección θ. Luego de esto, $r(\theta)$ es procesada según las siguientes reglas [12]:

- Si $d(x, T) - F \leq 0$, el objetivo T es visible. Crear un nodo llamado "*Nodo T*" en el objetivo.
- Si $F \geq r$, entonces no hay obstáculos visibles en la dirección de T.
- Analizar la función $r(\theta)$ en caso de discontinuidades. Si una discontinuidad es encontrada, crear un nodo en la dirección θ.
- Si $r(\theta) = r$ (el máximo rango del dispositivo sensible a la posición), y $r(\theta)$ subsecuentemente decrece, crear un nodo en la dirección θ. De igual manera para $r(\theta) \neq r$ y $r(\theta)$ subsecuentemente incrementa de tal manera que $r(\theta) = r$, se crea un nodo en la dirección θ.

El procedimiento para determinar la óptima dirección y distancia después de identificar todos los nodos es el siguiente:

- Para cada nodo, evaluar la distancia *d(Ni, T)*, donde *Ni* es el iésimo nodo.
- El nodo con el menor valor de *d(Ni, T)* es marcado como el nodo óptimo *N**.

El robot debe dirigirse hacia *N** mientras continuamente actualiza la gráfica local tangencial, procediendo al *N** más reciente. Para la figura 7.2, *N** es el nodo "*Nodo T*" debido a que es el más cercano al objetivo.

2.1.6.2. Los mínimos locales

En ciertas ocasiones el robot debe alejarse del objetivo para evadir obstáculos y llegar a él. Esto está definido como un mínimo local [5, 11, 13]. Cuando esto sucede, el algoritmo TangentBug pasa al modo seguidor de frontera. Esto involucra elegir una dirección para seguir la pared usando la gráfica local tangencial LTG. Mientras el robot sigue la pared, continuamente actualizan dos variables [12]:

- $d_{seguida}(T)$: Esta variable registra la distancia mínima hacia el objetivo, desde un obstáculo que se encuentre más cerca al objetivo.
- $d_{alcanzada}(T)$: En cada paso, el algoritmo TangentBug analiza el BCS dentro del rango de alcance de sus sensores y para un punto *P* en el cual *d(P, T)* es mínima. A $d_{seguida}(T)$ se le asigna el valor *d(P, T)*.

El modo seguidor de frontera sigue en ejecución hasta que una de las siguientes situaciones ocurra:

- $d_{alcanzada}(T) < d_{seguida}(T)$.
- El robot haya rodeado completamente el obstáculo más cercano al objetivo. Se considera el objetivo inalcanzable.

La operación del algoritmo TangentBug se observa en la figura 7.4 y el pseudocódigo del algoritmo se observa a continuación [13]:

1. Moverse continuamente a un punto $n \in \{T, O_i\}$ que minimice la distancia heurística $d(x,n) + d(n, q_{objetivo})$. Hacer esto hasta que ocurra lo siguiente:
    - El robot llegue a $q_{objetivo}$ = Finalizar satisfactoriamente ó
    - La dirección θ que minimiza a $d(x,n) + d(n, q_{objetivo})$ comience a incrementar a $d(n, q_{objetivo})$, como por ejemplo en un Mínimo Local $d(\cdot, q_{objetivo})$.

    Elegir una dirección para el modo *boundary-following* que continúe con la misma dirección como la del modo *move-to-goal* más reciente.
2. Actualizar continuamente $d_{alcanzada}, d_{seguida}$ y O_i.
3. Moverse continuamente hacia $n \in \{O_i\}$ que esté en la dirección elegida del modo *boundary-following*, hasta que ocurra lo siguiente:
    - El robot llegue a $q_{objetivo}$ = Finalizar satisfactoriamente ó
    - El robot circunnavegue W = El objetivo es inalcanzable.
    - $d_{alcanzada} < d_{seguida}$

Del pseudocódigo se identifica lo siguiente:

- T: punto donde un círculo centrado en la posición actual x de radio r interseca el segmento (x, $q_{objetivo}$).
- O_i: vértice del máximo rango del sensor de rango con la frontera del obstáculo W, que más reduce la distancia heurística hacia $q_{objetivo}$.
- Modos *boundary-following* y *move-to-goal*: modos seguidor de límite y dirigirse al objetivo "leave" respectivamente.

Se puede observar un robot en presencia de un mínimo local en la figura 7.3.

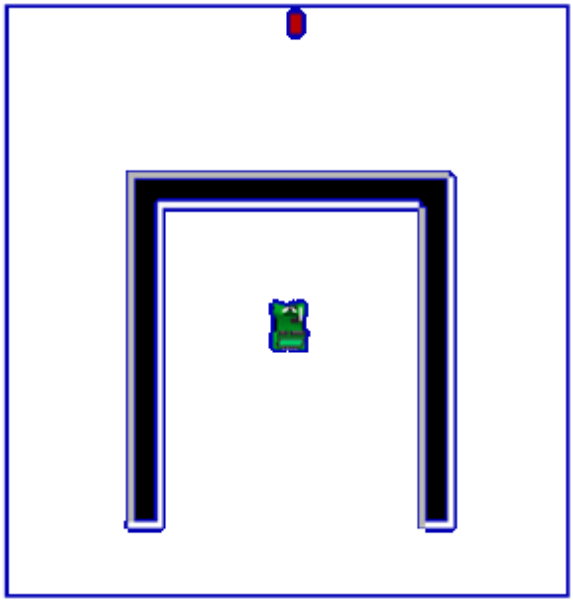


Figura 7.3: Un robot ante un mínimo local [12].

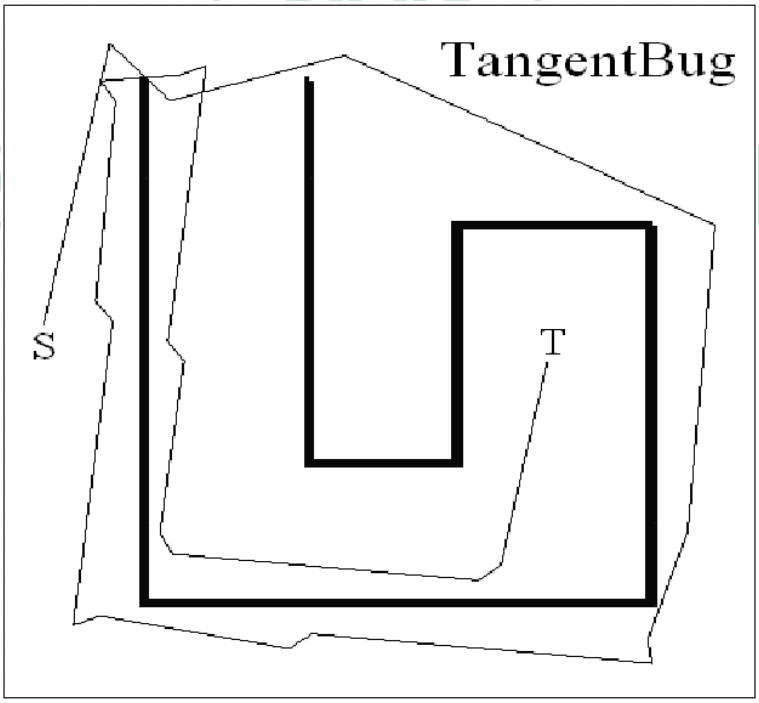


Figura 7.4: Representación del algoritmo TangentBug en un BCS A [12].

Si el robot llega a encontrar un obstáculo plano como una pared, el modo seguidor de ese límite o contorno indicaría al robot a moverse paralelo al obstáculo. Por lo general los obstáculos no suelen ser así y debido a esto se incrementa la eficiencia del algoritmo por medio de la representación geométrica [16] de elementos del BCS, que se aborda en la sección 6 de este documento.

2.2. ALGORITMOS NO PERTENECIENTES A LA FAMILIA BUG

2.2.1. El algoritmo Dijkstra

Creado por Edsger W. Dijkstra en 1956, es el algoritmo más famoso en el proceso de obtener una ruta corta entre dos nodos en un plano, gráfica o sistema [7]. Proviene de la investigación para lograr mover piezas de ajedrez en un tablero y mover fragmentos en un mapa de rompecabezas, y su operación se efectuó con el robot *Shakey* navegando en una habitación con obstáculos fijos. El algoritmo Dijkstra evalúa el costo de movimiento desde un nodo a otro y establece la ruta más corta respecto al costo para conectarlos. Con su aplicación se computan todas las rutas cortas desde un nodo de partida establecido en una gráfica completamente interconectada. La complejidad temporal cuantifica que tanto tiempo es necesario para que en la implementación el algoritmo cumpla su función con respecto a los datos de entrada. Para este algoritmo se calcula como $O(e + v \bullet \log v)$ [15]. Donde *e* (*edge*) representa los bordes y *v* los nodos. La distancia entre los nodos vecinos está dada por *e(n, m)*.

Al tener un BCS como un tablero de ajedrez, cada cuadro representa una celda y el algoritmo debe calcular la ruta más corta desde una celda de inicio *S* a una celda objetivo *T*. El algoritmo Dijkstra es una buena referencia para soluciones deterministas debido a la robusta matemática al momento de calcular la ruta más corta. Sin embargo, presenta dos principales problemas:

- El tiempo de ejecución aumenta al calcular todas las celdas del BCS, lo que significa una baja velocidad en el rendimiento.
- El uso de memoria presenta un crecimiento cuadrático, exigiendo una gran costo en memoria RAM para BCS masivos.

El explorar partes innecesarias es considerado como *OP* del inglés "*Overhead Problem*", algo que se ha mejorado con el desarrollo del algoritmo Dijkstra. Estas evoluciones del algoritmo son consideradas como la familia A* ó A estrella (A star), empleadas en muchas aplicaciones de planificación de rutas. Para el problema *OP* se emplea la heurística como método de eliminación de cálculo de celdas innecesarias [15].

El algoritmo almacena los costos de desplazamiento en una matriz y opera nodo a nodo para trazar la ruta más corta, teniendo en cuenta los menores costos nodo a nodo de toda la operación. A continuación se muestra el pseudocódigo para este algoritmo [15]:

1. Se comprende a *dist[s]* como la distancia a la fuente y se establece *dist[s] = 0*. Todas las demás distancias se establecen como infinitas *dist[v]* = ∞. Ejecutar paso 2.
2. Se comprende a *S* como el conjunto de vértices visitados y se establece como vacío, *S = Ø*. *Q* es la cola que contiene los demás vértices. Ejecutar paso 3.
3. Mientras *Q* no sea vacío, *Q ≠Ø*:
   - Seleccionar el elemento de *Q* con la menor distancia y asignarlo a *u*: $u = d_{mínima}(Q, d)$.
   - Agregar a *{u}* en una lista de vértices visitados, S. Ejecutar paso 4.
4. Para todo *v* que pertenecen a las celdas vecinas de *u*, *n[u]*:
   - Se tiene a *w(u,v)* como los pesos de *(u,v)*. Si la nueva ruta más corta fue encontrada, *dist[v] > dist[u] + w(u,v)*, entonces se establece el nuevo valor de la ruta más corta, *d[v] = d[u] + w(u,v)*. Ejecutar el paso 1.

*2.2.2. El algoritmo A**

Se consideran a P. Hart, N. Nilsson y B. Raphael como los creadores del algoritmo A* en 1968 [8], al mejorar el algoritmo Dijkstra para el robot *Shakey*, que debía navegar en una habitación que contenía obstáculos fijos. El objetivo principal del algoritmo A* es la eficiencia en la planificación de rutas y lo hace combinando las ventajas del algoritmo Dijkstra y el *Best First Search** (BFS). El BFS es una mejora del Dijkstra al estimar la distancia entre el punto de inicio y el objetivo, eligiendo si el próximo paso es más cercano hacia el objetivo.

El A* requiere una enorme cantidad de memoria para mantener el uso de datos relacionados con los nodos a evaluar [11]. Durante cada paso que se ejecuta en el algoritmo, busca con cada nuevo movimiento tomar la menor ruta e intentar determinar si esa ruta llevará al robot hacia el objetivo. Sin embargo, las rutas que no llevan al robot hacia T también son almacenadas

A diferencia del algoritmo Dijkstra el algoritmo A* no debe pasar por todos los nodos de la ruta más corta en el proceso de navegación, porque existe un costo estimado entre el inicio *S* y el objetivo *T*. Se emplea el método heurístico (prueba y error) donde se compara el árbol de navegación con la ruta ideal, determinando si la ruta que se elegirá es óptima y sin necesidad de pasar por un nodo adyacente [13].

Al hacer empleo del algoritmo BFS, se busca la ruta con menor costo desde un punto de inicio *S* hacia un punto objetivo *T*. Se usa una función heurística *distancia-más-costo*, usualmente denotada como $f(n)$ para determinar el orden en el cual se van a visitar los nodos de un árbol de búsqueda de nodos. Este árbol de nodos, rutas y costos se observa en la figura 8.1. La distancia-más-costo está expresada como la suma de dos funciones:

1. La función *ruta-costo* que representa el costo desde *S* hasta el nodo de posicionamiento actual, representada como $g(n)$.
2. Un estimado heurístico desde el nodo de posicionamiento actual, representado como $h(n)$.

La ecuación 1 expresa la función de costo para cada nodo *n* en el árbol de búsqueda de nodos actual, implementando el algoritmo A*.

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

Ecuación 1. Función distancia-más-costo del algoritmo A [8, 13].*

Para comprender el pseudocódigo del algoritmo es necesario tener presente la siguiente notación [13]:

- $Inicio_{(n)}$: Representa el conjunto de nodos que son adyacentes a *n*.
- $c(n_1, n_2)$: Es longitud del borde conectando n_1 y n_2.
- $g_{(n)}$: Es la longitud total de la ruta del punto de referencia desde *n* hasta q_{ruta}.
- $h_{(n)}$: Es la función de costo heurístico que determina el costo de la ruta más corta desde *n* hasta q_{ruta}.
- $f_{(n)} = g_{(n)} + h_{(n)}$: Es el costo estimado de la ruta más corta desde q_{inicio} hasta $q_{objetivo}$ vía *n*.

Se incluyen también dos estructuras de datos:

- *O*: Es el conjunto abierto de la cola de prioridad.
- *C*: Contiene todos los nodos procesados.

* El BFS es un algoritmo de búsqueda que se expande hacia el nodo más óptimo. A este tipo de búsqueda se le considera "*codiciosa*" (*greedy*) [13].

Se considera al algoritmo A* como uno que genera un árbol de búsqueda, que por definición no ejecuta ciclos. Hay un número finito de rutas acíclicas que son consideradas por el algoritmo, lo cual significa que el proceso de buscar todas las rutas acíclicas también es finito [13]. La figura 8.1 muestra los nodos representados con letras y los costos heurísticos son los números encerrados dentro de los círculos de nodo. Los costos de borde, que son las líneas o rutas que unen los nodos, se encuentras adyacentes a estas rutas. El nodo de inicio está representado con un cero "*0*", lo que indica la importancia jerárquica de máxima prioridad en la cola de búsqueda.

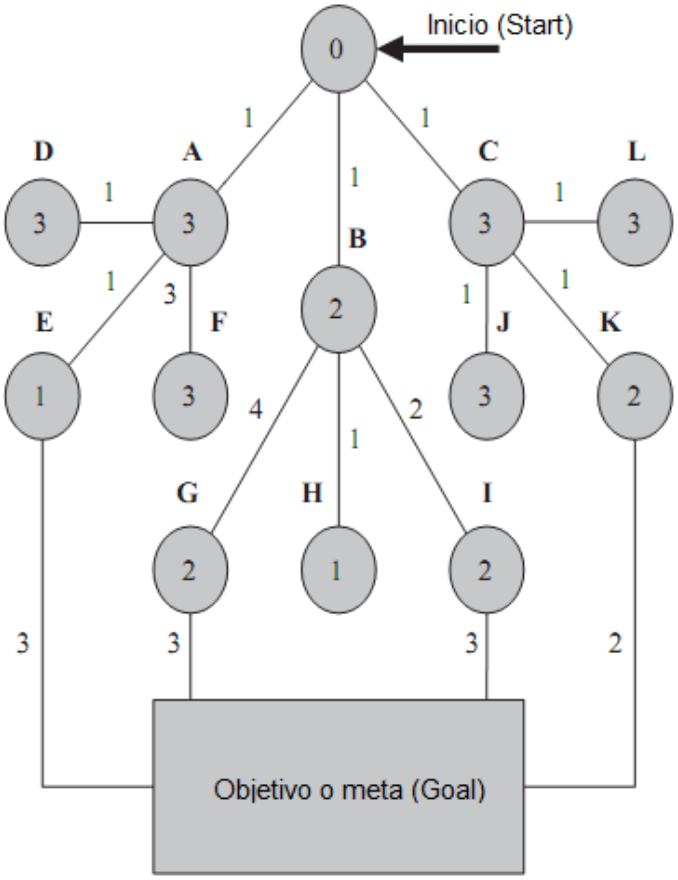


Figura 8.1: Representación del árbol de búsqueda del algoritmo A* [13].

Los nodos con alto costo son descartados por el algoritmo y la ruta que va desde el inicio al objetivo, es una serie puntos de referencia. Estos puntos de referencia representan el historial inmediato del proceso de expansión.

Por ejemplo, se considera el proceso de ir de *0* al objetivo. Si se toma la ruta de puntos *0*, *A*, *E*, los puntos *A*, *B* y *C* tienen a *0* como punto de referencia, porque ahí empieza un proceso de expansión. De igual manera sucede para *D*, *E* y *F*, que tienen al punto *A* como punto de referencia.

El pseudocódigo del algoritmo A* por pasos se presenta a continuación [13]:

1. Seleccionar la mejor vía $n_{óptima}$ desde *O*, de tal manera que $f(n_{óptima}) \leq f_{(n)}$, para todo $n \in O$.
2. Remover a $n_{óptima}$ de *O* y agregarla en *C*.
3. Si $n_{óptima} = q_{objetivo}$, finalizar.
4. Expandir $n_{óptima}$: para todo $x \in Inicio(n_{óptima})$ que no están en *C*.
5. Si $x \notin O$ entonces:
   - Agregar a *x* en *O*.
   - De lo contrario, si $g(n_{óptima}) + c(n_{óptima}, x) < g(x)$ entonces:
     - Actualizar el punto de referencia de *x* a $n_{óptima}$.
6. Repetir si *O* es vacío.

Cuando se hace referencia a algoritmos que discretizan el BCS, se considera la *Distancia Manhattan* como un complemento de la métrica Euclidiana para determinar la distancia entre dos puntos. En la figura 9.1 se observa como la *Distancia Manhattan* tiene una medida de distancia rectilínea, a comparación de la tradicional Euclidiana directa.

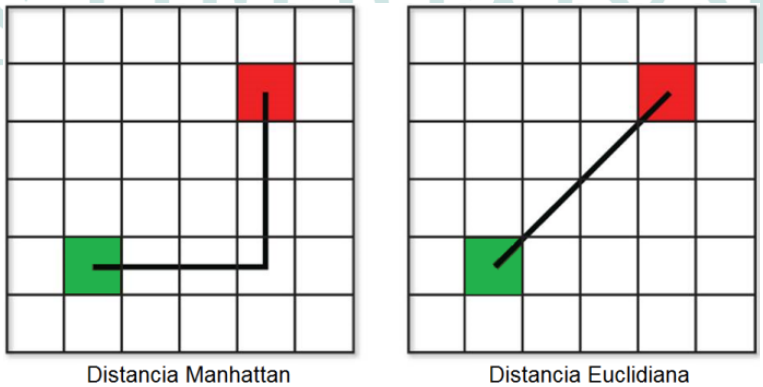


Figura 9.1: Representación de la Distancia Manhattan y la Distancia Euclidiana.

La eficiencia de un algoritmo para determinar la ruta más corta entre un punto de inicio *S* y un punto de objetivo *T*, depende de cómo el robot pueda operar óptimamente los obstáculos que encuentre en su navegación. La distancia Euclidiana se emplea para reducir el costo de

ruta, pero al calcular las raíces cuadradas de muchas búsquedas de ruta puede generar una gran necesidad de recursos computacionales [13].

*2.2.3. Algoritmo D**

El algoritmo D* ("D" estrella) creado por A. Stentz [9] es muy diferente de la familia de algoritmos Bug debido a que usa "*mapeo*" (en inglés *mapping*), que es el proceso de elaborar una representación métrica o topológica de su BCS. Con esta representación el robot se localizará a sí mismo en un mapa. El *mapping* está prohibido en la familia de algoritmos Bug, pero este proceso tiene una interesante metodología de operación. El D* es un algoritmo de "*fuerza bruta*" con propiedades únicas [11, 13], para segmentar un mapa en áreas discretas denominadas celdas.

El D* se diferencia del algoritmo A* en que puede operar más eficientemente ante obstáculos dinámicos, gracias a una modificación en el algoritmo Dijkstra. Con esta modificación, se considera un proceso de "reparación local" de la ruta de navegación, permitiendo una actualización eficiente en la búsqueda de una ruta óptima en BCS con obstáculos dinámicos. El algoritmo A* vuelve a ejecutarse por completo cuando aparece un obstáculo dinámico en su ruta de navegación, lo que representa un gran consumo de recursos computacionales al determinar el árbol de rutas en celdas innecesariamente [13].

Cada celda posee un punto base que representa la dirección óptima de navegación en el área propia de la celda, así como el "*costo*" de desplazamiento a una celda próxima o vecina. Para generarse una ruta adecuada se establece la segmentación o discretización del BCS en celdas, como se observa en la figura 10.1. Para este caso se discretiza en una cuadrícula en la que cada desplazamiento entre celdas se considera como un "costo". Por medio de un ejemplo se muestra el funcionamiento del algoritmo D*, en el que los desplazamientos verticales y horizontales entre celdas tendrán un costo de 1, mientras que el desplazamiento diagonal tiene un costo de $\sqrt{2}$. Se hace uso de la Distancia Manhattan y de la Distancia Euclidiana de la figura 9.1 para buscar una ruta óptima [13].

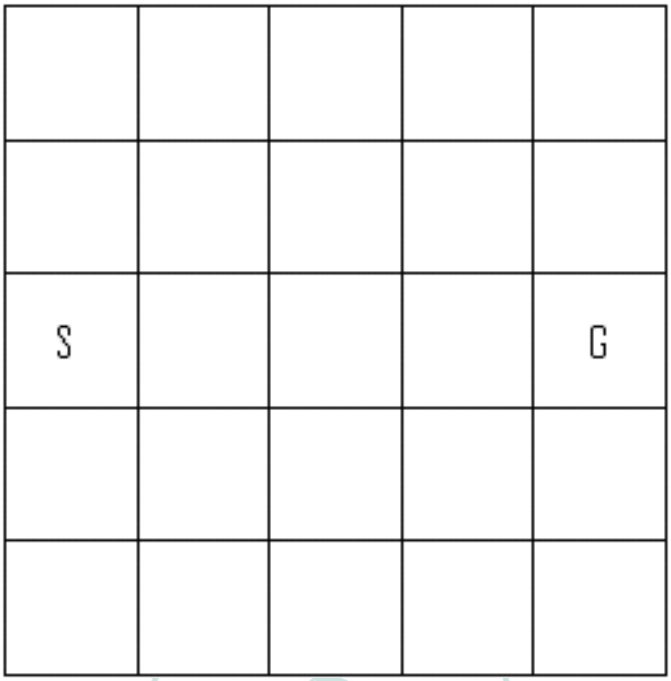


Figura 10.1: BCS segmentado o discretizado para el algoritmo D* [11].

El BCS tendrá coordenadas *(X, Y)* y cada cuadro se denomina celda. Para el algoritmo D* se tendrá una tabla con las coordenadas y los costos de desplazamiento entre celdas, todo para llegar al objetivo *G* según la notación que se observa en la figura 10.2. Las flechas cerca a *G* son los puntos base mencionados anteriormente, que ubican las celdas más cercanas al objetivo *G* y muestran el tipo de desplazamiento óptimo vertical u horizontal para alcanzar a *G*.

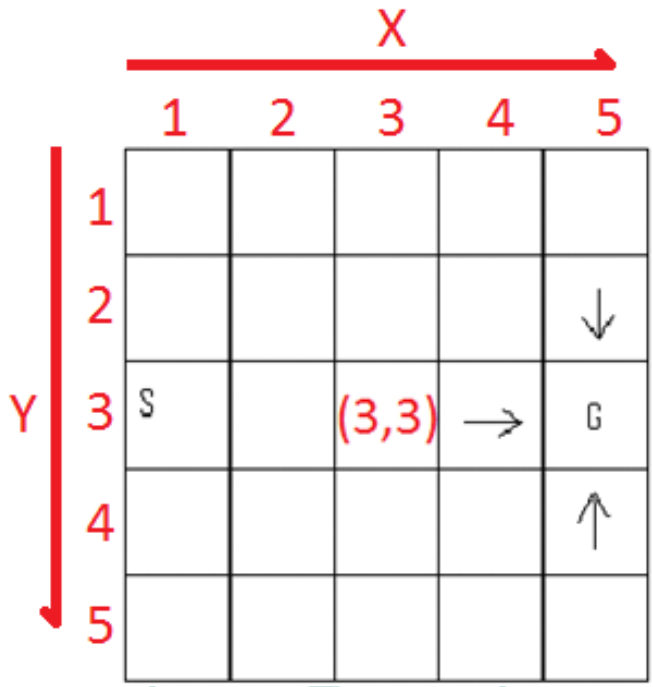


Figura 10.2: BCS del D* con celdas de aproximación a G [11].

Una vez se tiene claro el significado de "*costo*", su magnitud y tipo de desplazamiento, se procede a tener tablas de estos datos.

Tabla 1: Primera tabla generada por el algoritmo D* [11].

Posición (1)	**Celda vecina con señalador u objetivo (2)**	**Costo desde (1) a (2)**	**Costo desde (2) a G**	**Costo total**
(5,4)	T	1	0	1
(5,2)	T	1	0	1
(4,3)	T	1	0	1
(4,2)	T	1.414	0	1.414
(4,4)	T	1.414	0	1.414

La tabla 1 muestra que las celdas (5,4), (5,2) y (4,3) poseen el menor costo. Para saber interpretar lo que hace el algoritmo D* con una tabla de datos, los títulos de la tabla 1 contienen (1) y (2). Para la columna "Costo de (1) a (2)", se observa el tipo de desplazamiento desde la celda (5,4) hasta la celda *G* que es (5,3), determinando que es un desplazamiento vertical con costo 1. Estas celdas establecen los puntos base de

aproximación hacia el objetivo. La figura 10.3 muestra los datos de la tabla 1. Con lo anterior, las celdas vecinas de *G*, (5,4), (5,2) y (4,3) son considerados por el costo mínimo total de la tabla 2.

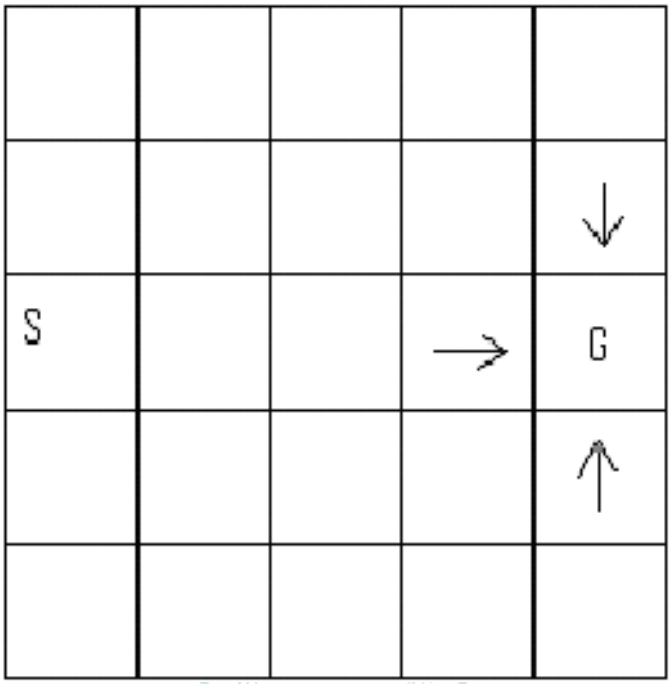


Figura 10.3: BCS del D* con puntos base hacia G [11].

Se tiene una segunda tabla con los datos de las celdas restantes que generará el algoritmo D*.

Tabla 2: Segunda tabla generada por el algoritmo D* [11].

Posición (1)	**Celda vecina con señalador u objetivo (2)**	**Costo desde (1) a (2)**	**Costo desde (2) a G**	**Costo total**
(4,4)	T	1.414	0	1.414
(4,2)	T	1.414	0	1.414
(3,3)	(4,3)	1	1	2
(5,1)	(5,2)	1	1	2
(5,5)	(5,4)	1	1	2
(3,2)	(4,3)	1.414	1	2.414
(4,5)	(5,4)	1.414	1	2.414
(4,1)	(5,2)	1.414	1	2.414
(3,4)	(4,3)	1.414	1	2.414

La tabla 2 muestra que las celdas (4,4) y (4,2) tienen el menor costo. Estas celdas establecen los puntos base para aproximarse hacia el objetivo, y la figura 10.4 muestra los datos de la tabla 2.

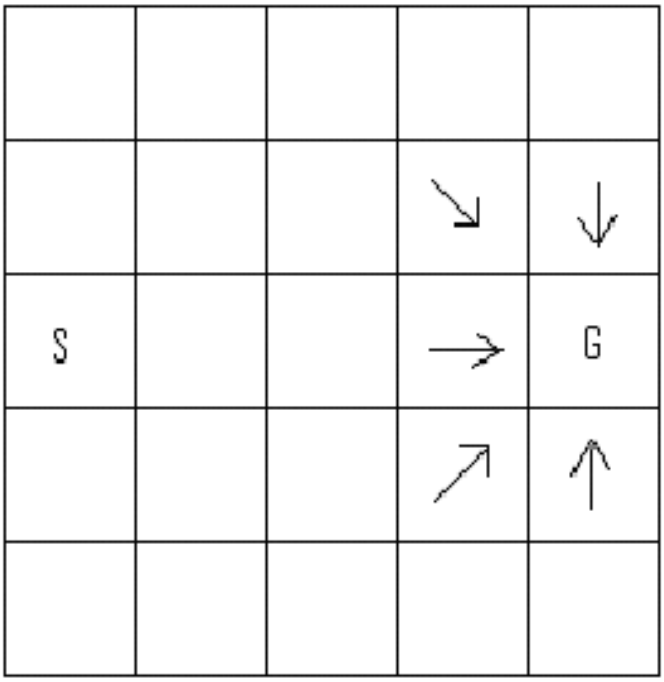


Figura 10.4: BCS del D* con puntos base previos y actuales [11].

Este proceso se repite hasta que la posición en la que se encuentra el robot contiene un punto base o todo el BCS es llenado con datos. Si una celda tiene un punto base, esta celda representa el menor costo para moverse hacia el objetivo. Este es el proceso del cómo se generan las rutas óptimas con el algoritmo D*. La figura 10.5 muestra finalmente el BCS de 5x5 con *G* y los puntos base que llevaran al robot hacia G.

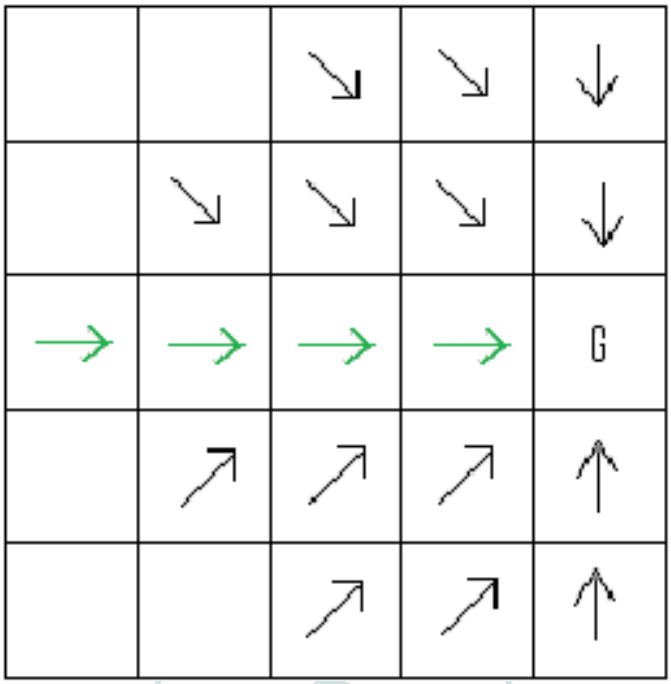


Figura 10.5: BCS del D* con puntos bases y la ruta óptima de aproximación a G [11].

Para el algoritmo D*, ¿qué sucede cuando el robot encuentra un obstáculo? Se puede dar respuesta a este interrogante con otro ejemplo, que involucre un BCS de 5x5 celdas y un obstáculo en (3,3) como se observa en la figura 10.6. El algoritmo D* representa obstáculos incrementando el costo de desplazamiento, pero no desde las celdas obstáculo. Es decir, si un obstáculo se encuentra en una celda *O*, el valor de costo de desplazamiento asociado desde las celdas vecinas alrededor de *O* hacia *O* aumenta.

Una vez que los costos de desplazamiento son modificados, el algoritmo D* vuelve a procesar los puntos base para asegurarse que sean óptimos y esto se hace considerando los puntos base hacia la celda (3,3). La tabla 3 muestra este proceso.

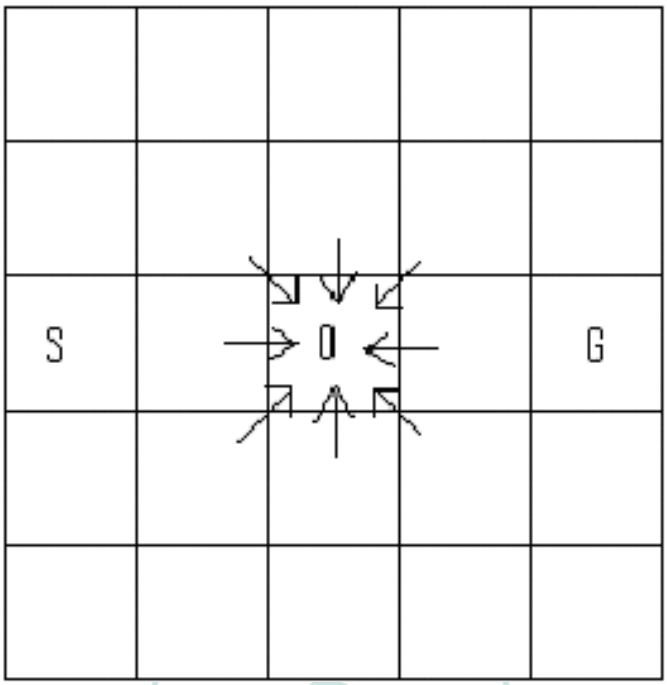


Figura 10.6: BCS del D* con obstáculo en la celda (3,3) [11].

Tabla 3: Primera tabla generada cuando se detecta un obstáculo con el algoritmo D* [11].

Posición (1)	**Celda próxima con señalador u objetivo (2)**	**Costo desde (1) a (2)**	**Costo desde (2) a G**	**Costo total**
(2,2)	(3,2)	1	2.414	3.414
(2,4)	(3,4)	1	2.414	3.414
(2,3)	(3,4)	1.414	2.414	3.828

La tabla 3 muestra que las celdas (2,2) y (2,4) tienen un nuevo costo de desplazamiento mínimo, y que sus puntos base cambian a las celdas especificadas en la columna 2 de la tabla 3. El BCS actualizado se muestra en la figura 10.7 y al repetirse este proceso nuevamente, el algoritmo D* genera la tabla 4.

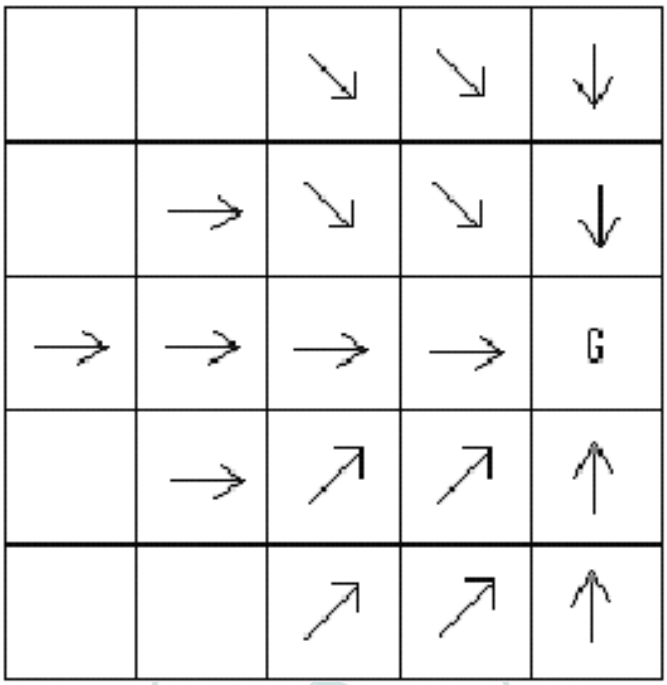


Figura 10.7: BCS de D* actualizado después de procesar la tabla 3 [11].

Tabla 4: Segunda tabla generada cuando se detecta un obstáculo con el algoritmo D* [11].

Posición (1)	**Celda próxima con señalador u objetivo (2)**	**Costo desde (1) a (2)**	**Costo desde (2) a G**	**Costo total**
(2,3)	(3,4)	1.414	2.414	3.828
(2,1)	(3,2)	1.414	2.414	3.828
(2,5)	(3,4)	1.414	2.414	3.828
(1,4)	(2,4)	1	3.414	4.414
(1,2)	(2,2)	1	3.414	4.414
(1,3) (S)	(2,2)	1.414	3.414	4.828
(1,5)	(2,4)	1.414	3.414	4.828
(1,1)	(2,2)	1.414	3.414	4.828

La tabla 4 muestra que las celdas (2,3), (2,1) y (2,5) cambian sus puntos base de tal manera que su costo de desplazamiento es minimizado. La actualización del BCS una vez se procesa la tabla 4 se muestra en la figura 10.8.

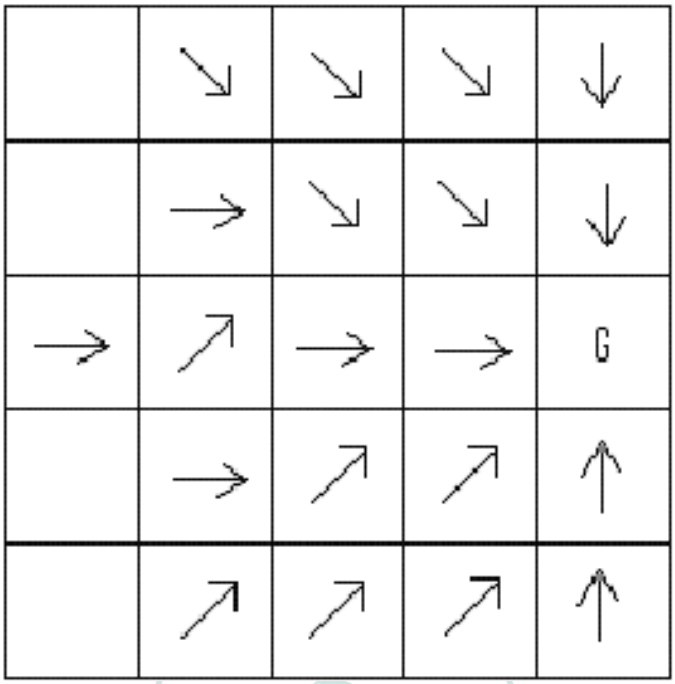


Figura 10.8: BCS del D* actualizado al procesar la tabla 4 [11].

Nuevamente el algoritmo D* repite este proceso hasta que el costo mínimo total en la nueva tabla generada sea mayor o igual al costo de desplazamiento en la ruta de puntos base que el robot está siguiendo. El costo mínimo es mayor porque está en presencia de un obstáculo, pero será el menor de las futuras tablas generadas por el algoritmo. Entonces se trata de un costo ligeramente mayor o igual a la ruta de puntos base que el robot se encuentra siguiendo.

Una vez ocurrido lo anterior, será claro que los futuros procesamientos computacionales no generarán una ruta con menor costo de desplazamiento que el actual. La tabla 5 muestra la computación de la ruta para este ejemplo.

Tabla 5: Tercera tabla ante la detección de un obstáculo en la celda (3,3) [11].

Posición (1)	**Celda próxima con señalador u objetivo (2)**	**Costo desde (1) a (2)**	**Costo desde (2) a G**	**Costo total**
(1,2)	(2,2)	1	3.414	4.414
(1,4)	(2,4)	1	3.414	4.414
(1,3) (S)	(2,3)	1	3.828	4.828
(1,5)	(2,4)	1.414	3.414	4.828
(1,1)	(2,2)	1.414	3.414	4.828

Algo que se ha tenido muy presente por la importancia en los algoritmos de navegación es su finalización. La condición de finalización del algoritmo D* en presencia de un obstáculo se muestra en la tabla 6. La figura 10.9 muestra el BCS actualizado al generarse la tabla 6.

Tabla 6: Cuarta tabla generada con la condición de terminación en presencia de un obstáculo para el Algoritmo D* [11].

Posición (1)	**Celda próxima con señalador u objetivo (2)**	**Costo desde (1) a (2)**	**Costo desde (2) a G**	**Costo total**
(1,3) (S)	(2,3)	1	3.828	4.828
(1,5)	(2,4)	1.414	3.414	4.828
(1,1)	(2,2)	1.414	3.414	4.828

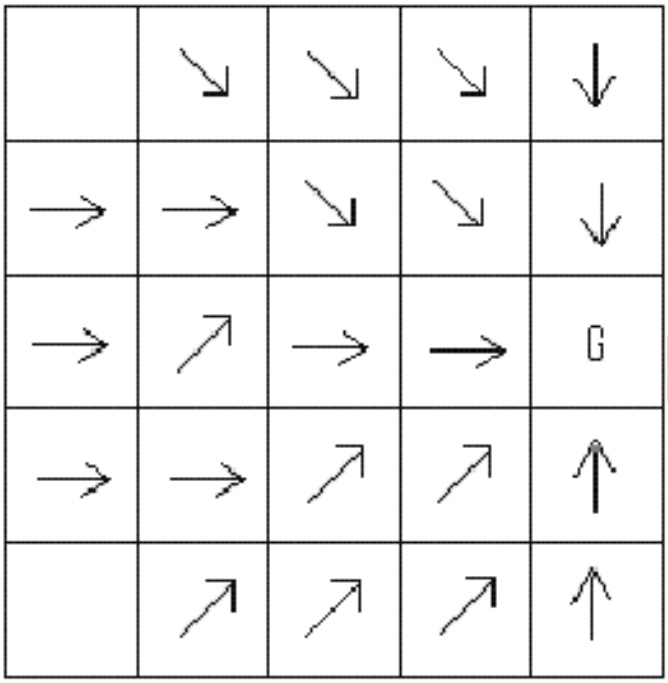


Figura 10.9: BCS del D* actualizado al generar la tabla 6 [11].

De la figura 10.9 debe notarse que la dirección en la celda (2,3) no apunta hacia la celda anterior o el inicio en (1,3). El algoritmo D* en los mínimos locales se mantiene de manera óptima con respecto a otras técnicas similares [11]. Sin embargo, esto implica otro costo en tiempo de computación. Se recuerda que al tratarse de mínimos locales, el robot en otras técnicas se aleja del objetivo para sobreponerse a un obstáculo.

Para el algoritmo D*, el costo de desplazamiento puede hacerse en cualquier instante. Esto permite al algoritmo adaptarse dinámicamente ante obstáculos previamente no detectados y generar rutas óptimas. El mecanismo de costo del algoritmo D* se permite en BCS indeseados en cuanto a zonas en las que podría hacerse más difícil desplazarse que en otras, como por ejemplo la no uniformidad del terreno. Abrían costos más altos en esas zonas comparadas con el terreno uniforme, más no necesariamente esas zonas indeseables se considerarían un obstáculo [11].

2.2.3.1. Determinación de acceso del robot al objetivo T

Cada obstáculo posee dimensiones y para un entorno de configuración bidimensional el robot detectará una dimensión del obstáculo (podría depender de la forma y posición del obstáculo). Una dimensión del obstáculo puede estar asociada a un valor umbral y este valor umbral también es determinante para alcanzar o no el objetivo.

Considerar si el robot puede acceder al objetivo o no, está determinado por una comparación entre el costo de la ruta de puntos base con el valor umbral grande de los obstáculos. Si el costo de la ruta de puntos base es superior al valor umbral grande del obstáculo, esto implica que la ruta óptima está expuesta a la dimensión siempre superior de un obstáculo y por lo tanto el objetivo es inalcanzable.

El valor umbral grande debe ser seleccionado por el robot para que el costo de cualquier secuencia de puntos base que no se topa con un obstáculo, nunca exceda el valor umbral grande. La figura 11.1 muestra la operación del algoritmo D* en un BCS.

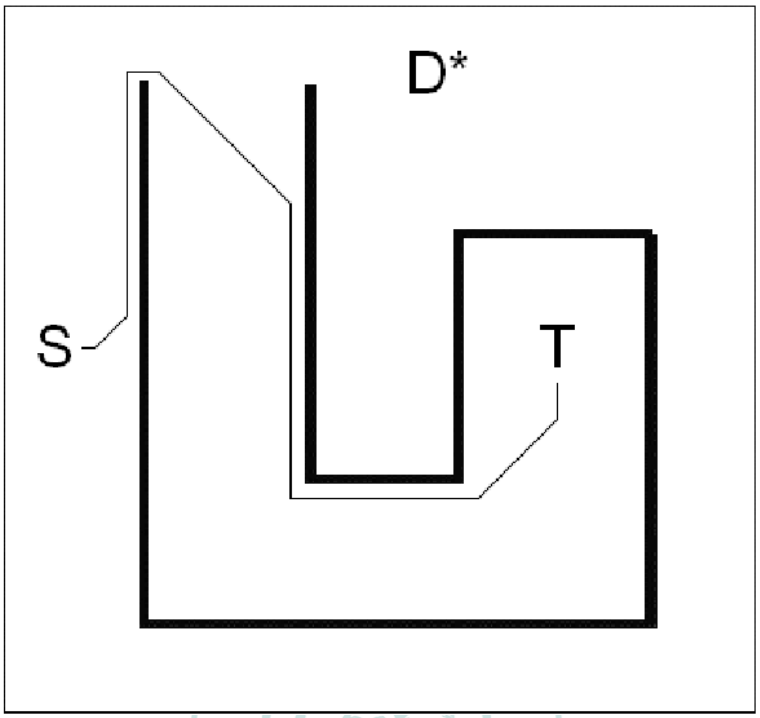


Figura 11.1: Representación del algoritmo D* en un BCS A [12].

2.2.4. Algoritmo Com

Este algoritmo no es un algoritmo Bug oficial y no garantiza una finalización [12]. Sin embargo, es comúnmente empleado para mostrar lo que sucede cuando el robot se dirige al objetivo en cualquier instante que le sea posible. El algoritmo Com se emplea para mejorar los algoritmos Bug y justificar porque se necesitan reglas especiales para que un robot se aleje del obstáculo, lo que se ha denominado "leave". La operación de este algoritmo se observa en la figura 12.1 y a continuación se muestra el pseudocódigo para este algoritmo de navegación [12]:

1. Moverse hacia el objetivo hasta que una de las siguientes situaciones ocurra:
    - El robot alcance el objetivo *T* = Finalizar satisfactoriamente.
    - El robot encuentre un obstáculo = Moverse por la frontera del obstáculo. Ejecutar el paso 2.
2. Alejarse del obstáculo si el robot puede desplazarse hacia *T*. Ejecutar el paso 1.

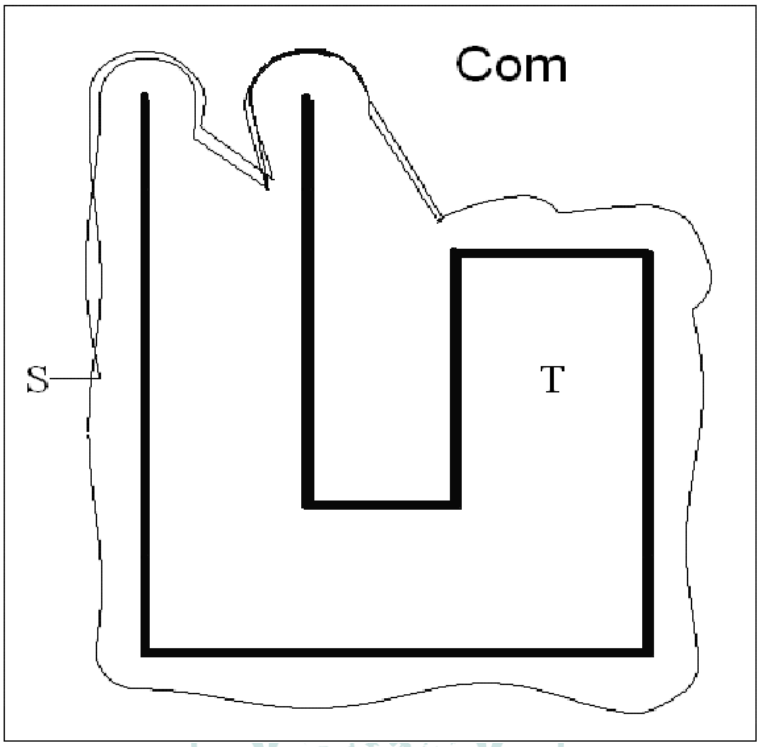


Figura 12.1: Representación del algoritmo Com en un BCS A [12].

Se observa en la figura 12.1 que el robot nunca alcanzará a *T* y lo va a circunnavegar indefinidamente.

2.2.5. Algoritmo Class1

No es un algoritmo oficial en la familia Bug y no garantiza una finalización finita [12]. Es comúnmente empleado para ilustrar lo que sucede cuando el robot se dirige al objetivo en el instante que le sea posible en cualquier punto visitado con anterioridad, cuando está cerca del objetivo *T*. De igual manera que el algoritmo Com, El Class1 sirve para mejorar los algoritmos Bug y justificar porque las reglas especiales "leave" deben ser aplicadas. La figura 13.1 representa la operación del algoritmo Class1 y a continuación se muestra el pseudocódigo para este algoritmo de navegación [12]:

1. Moverse hacia el objetivo hasta que una de las siguientes situaciones ocurra:
   - El robot alcanza el objetivo *T* = Finalizar satisfactoriamente.
   - El robot encuentra un obstáculo = Moverse por la frontera del obstáculo e ir al paso 2.
2. Partir hacia el objetivo si el robot puede hacerlo y si se encuentra cerca a él en cualquier punto visitado previamente. Ejecutar el paso 1.

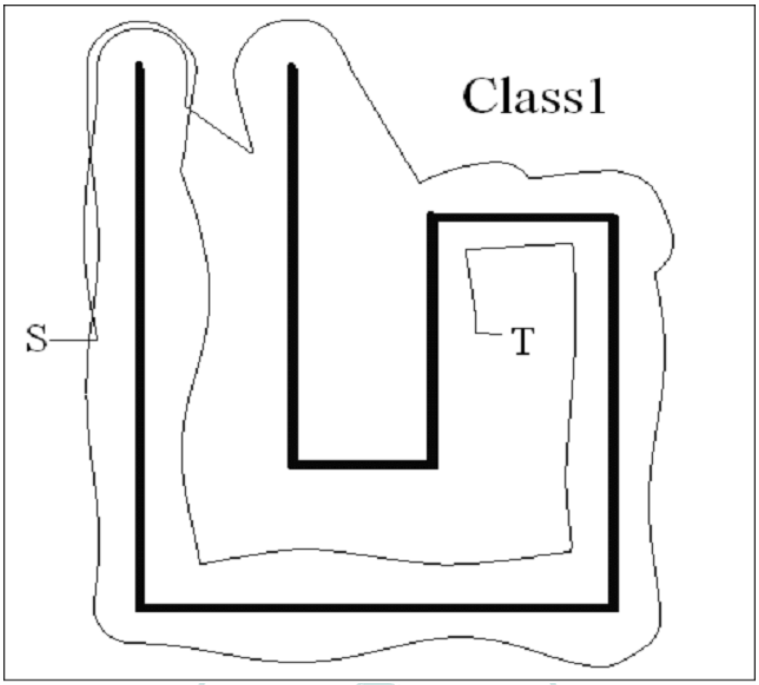


Figura 13.1: Representación del algoritmo Class1 en un BCS A [12].

2.2.6. Algoritmo Rev1

El algoritmo Rev1 fue creado por Y. Horiuchi y H. Noborio [6]. Es muy similar al Alg1, excepto que al seguir una pared el robot cambia la dirección de desplazamiento al encontrar un obstáculo. También tiene unas listas H y L como mecanismo de almacenamiento de datos. La operación de este algoritmo se observa en la figura 14.1 y a continuación se muestra el pseudocódigo para este algoritmo de navegación [12]:

1. Moverse hacia el objetivo T hasta que una de las siguientes situaciones ocurra:
   - El robot alcanza el objetivo T = Finalizar satisfactoriamente.
   - El robot se encuentra con un obstáculo = Marcar el punto como H_i y registrar los detalles en las listas H y L.
2. La dirección de desplazamiento es revisada en H_i y la lista H. Si ambas direcciones ya fueron revisadas en H_i, se elimina de la lista H. El robot traza un obstáculo en la dirección D hasta que una de las siguientes situaciones ocurra:
   - El robot alcanza el objetivo T = Finalizar satisfactoriamente.
   - Si la distancia hacia T es corta respecto a la de cualquier otro punto visitado anteriormente (condición métrica), el robot se encuentra sobre la línea M (condición de segmento) y el robot puede dirigirse directamente hacia T (condición física). Almacenar este punto como L_i en la lista L, cambiar la dirección D y ejecutar el paso 1.

- Si el robot regresa al último punto H_i, finalice con un fallo. Para este caso, T está completamente encerado por el límite de un obstáculo.
- Si el robot regresa a un punto H previo, el punto H_k es almacenado así como el último punto Q_i ($Q = d(S, T)$) en la lista H. El robot regresa a H_i empleando la ruta más corta de la lista H y L. una vez el robot llegue a H_i, este sigue la pared en dirección opuesta a como lo hacía previamente.
- Si el robot regresa a un punto L_k, este es almacenado como el último punto en la lista H. Luego el robot regresa al último punto H_i empleando la ruta más corta de la lista H y L. Una vez el robot regrese a H_i, debe seguir moviéndose por la pared en la dirección opuesta que tenía anteriormente.

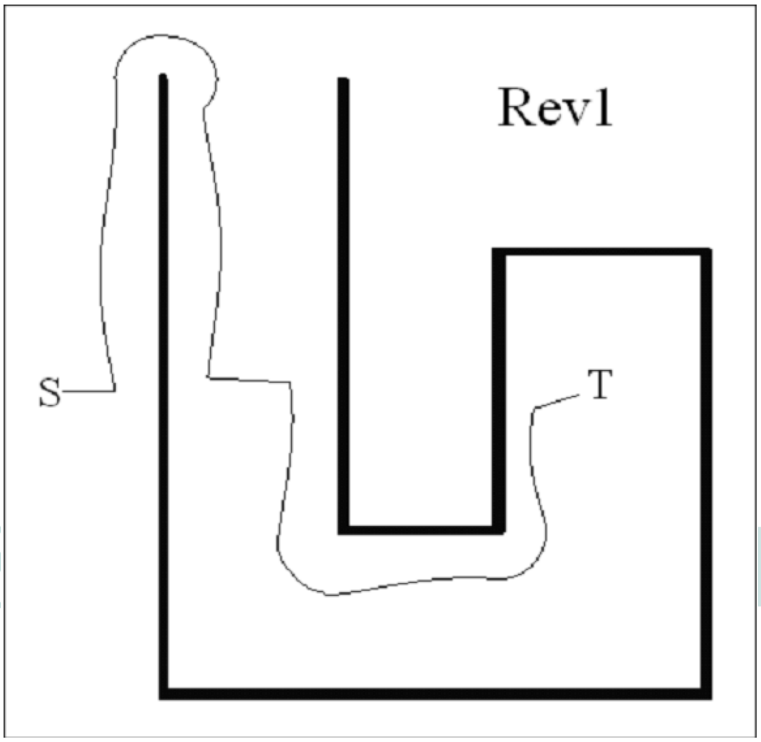


Figura 14.1: Representación del algoritmo Rev1 en un BCS A [12].

2.2.7. Algoritmo Rev2

El algoritmo Rev2 es muy similar al Alg2, excepto que al seguir una pared el robot cambia la dirección de desplazamiento. También fue creado por Y. Horiuchi y H. Noborio [6]. Además la única diferencia entre los algoritmos Rev2 y Rev1 es la ausencia de la condición de segmento en el paso 2. La operación de este algoritmo se observa en la figura 15.1 y a continuación se muestra el pseudocódigo para este algoritmo de navegación [12]:

1. Moverse hacia el objetivo T hasta que una de las siguientes situaciones ocurra:
    - El robot alcanza el objetivo T = Finalizar satisfactoriamente.
    - El robot se encuentra con un obstáculo = Marcar el punto como H_i y registrar los detalles en las listas H y L.
2. La dirección de desplazamiento es revisada en H_i y la lista H. Si ambas direcciones ya fueron revisadas en H_i, se elimina de la lista H. El robot traza un obstáculo en la dirección D hasta que una de las siguientes situaciones ocurra:
    - El robot alcanza el objetivo T = Finalizar satisfactoriamente.
    - Si la distancia hacia T es corta respecto a la de cualquier otro punto visitado anteriormente (condición métrica) y el robot puede dirigirse directamente hacia T (condición física). Almacenar este punto como L_i en la lista L, cambiar la dirección D y ejecutar el paso 1.
    - Si el robot regresa al último punto H_i, finalice con un fallo. Para este caso, T está completamente encerado por el límite de un obstáculo.
    - Si el robot regresa a un punto H previo, el punto H_k es almacenado así como el último punto Q_i ($Q = d(S, T)$) en la lista H. El robot regresa a H_i empleando la ruta más corta de la lista H y L. una vez el robot llegue a H_i, este sigue la pared en dirección opuesta a como lo hacía previamente.
    - Si el robot regresa a un punto L_k, este es almacenado como el último punto en la lista H. Luego el robot regresa al último punto H_i empleando la ruta más corta de la lista H y L. Una vez el robot regrese a H_i, debe seguir moviéndose por la pared en la dirección opuesta que tenía anteriormente.

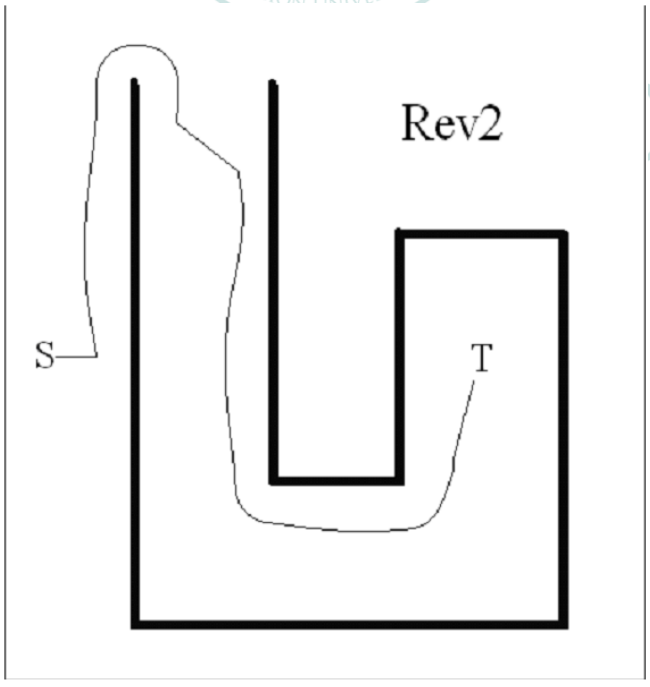


Figura 15.1: Representación del algoritmo Rev2 en un BCS A [12].

3. ALGORITMOS DE NAVEGACIÓN PARA BCS CON PISTAS GUÍA DE NAVEGACIÓN

Se considera que existe una pista guía entre el inicio S y el objetivo T. Un nuevo algoritmo de navegación Curv [10] se desarrolla y se puede expandir de cuatro maneras:

1. El algoritmo Curv1 fue desarrollado para llevar un robot desde S hasta T en un BCS desconocido, con una simple pista guía de navegación sin intersección propia. Cuando se tiene una pista de navegación con intersección propia, un nuevo algoritmo se desarrolla como Curv2 para garantizar la terminación.
2. Existe la duda de si el algoritmo Curv1 es el único capaz de conducir o no al robot al lugar que se desea alcanzar, es decir, al objetivo.
3. Los obstáculos dinámicos son considerados. Estos obstáculos pueden ir y venir durante la tarea de navegación.
4. Múltiples parejas de inicio/objetivo y múltiples pistas guía de navegación son consideradas. Con lo anterior, un nuevo algoritmo Curv3 es desarrollado para emparejar de manera única inicio/objetivos.

Se plantea el problema de planeación de rutas de navegación para un robot en un BCS bidimensional desconocido. Se requiere que el robot se desplace desde un punto de inicio hasta un objetivo, y existe una pista de navegación desde el inicio hasta el objetivo. También existe un número finito de obstáculos, cada uno con un perímetro finito. Estos obstáculos pueden o no estar sobre la pista guía de navegación. El robot es capaz de determinar la dirección de la pista guía que debe seguir para llegar al objetivo.

La ventaja de estas pistas es que permiten una cierta autonomía de navegación, donde no existe localización ni técnicas de compensación de error como la localización probabilística. La desventaja es que las pistas guía de navegación deben ser creadas antes de que la navegación del robot se realice.

3.1. ALGORITMO CURV1

Con este algoritmo se asume que el objetivo siempre se puede alcanzar. El robot siempre debe almacenar por lo menos un punto para detectar circunnavegación. Si la circunnavegación es detectada, el robot usualmente debe concluir que el objetivo es inalcanzable. Como el algoritmo Curv1 no asume la habilidad de localización, no hay una posibilidad de detectar que la circunnavegación ha ocurrido y por lo tanto el objetivo es inalcanzable [10]. La figura 16.1 representa la operación del algoritmo y a continuación se muestra el pseudocódigo para este algoritmo de navegación [12]:

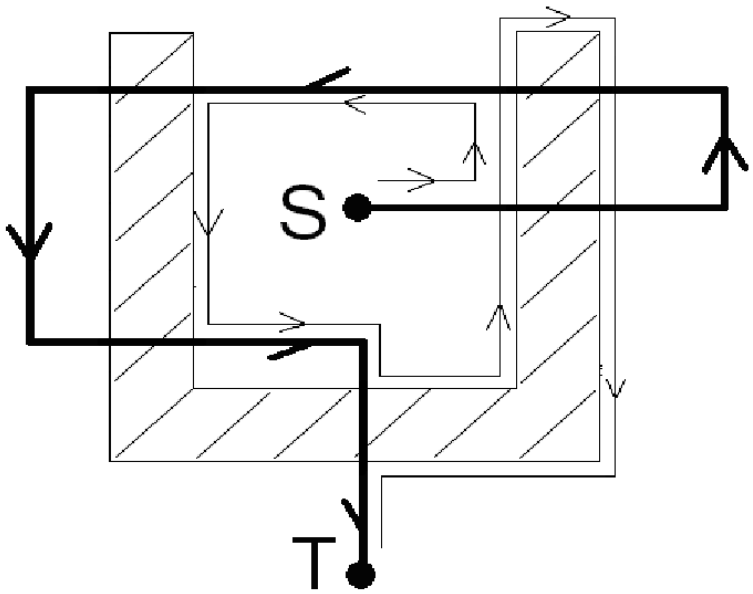


Figura 16.1: Representación del algoritmo Curv1 [12].

1. Establecer el contador C en cero. Iniciar desde el punto S y moverse hacia T. Ejecutar el paso 2.
2. Moverse sobre la curva ST hasta que una de las siguientes situaciones ocurra:
   - El robot alcanza el objetivo T = Finalizar satisfactoriamente.
   - El robot encuentra un obstáculo = Siga el límite del obstáculo en la dirección local izquierda. Ejecutar el paso 3.
3. Siga el límite o borde del obstáculo hasta que una de las siguientes situaciones ocurra:
   - El robot alcanza el objetivo T = Finalizar satisfactoriamente.
   - La curva ST es alcanzada en un punto P. Uno de los siguientes pasos es ejecutado:
     - El contador C lee cero. En el punto P, MA se puede mover a lo largo de la curva ST hacia T. Dejar el obstáculo y seguir la curva ST. Ejecutar el paso 2.
     - El contador C no es cero. En el punto P, MA se puede mover a lo largo de la curva ST hacia T. Decrementar el contador C en 1. Continuar moviéndose a lo largo de la frontera del obstáculo. Ejecutar el paso 3.
     - En el punto P, MA no se puede mover a lo largo de la curva ST hacia T. Incrementar el contador C en 1. Continuar moviéndose a lo largo de la frontera del obstáculo. Ejecutar el paso 3.

3.2. ALGORITMO CURV2

Cuando el robot encuentre intersecciones en una pista guía, se puede considerar y tratar como un obstáculo [10], excepto que no necesita realizar la acción de seguimiento de pared. Existe una notación para las auto intersecciones y se dividen en dos. Los segmentos que entran en la intersección son considerados como Afluentes (*inflows*). Estos afluentes se pueden denotar como I_j, donde j representa el "j" ésimo afluente. Los segmentos que salen de la intersección son considerados como salida o fuga (*outflows*). Estas salidas se pueden denotar como O_j, donde j representa la "j" ésima salida. Los afluentes y salidas pueden ser marcadas arbitrariamente. Se considera A como el ángulo medido en el sentido horario con respecto a I donde el robot entra a la intersección, I_n. I_n puede cambiar si el robot vuelve a visitar la misma intersección. La figura 17.1 muestra la notación.

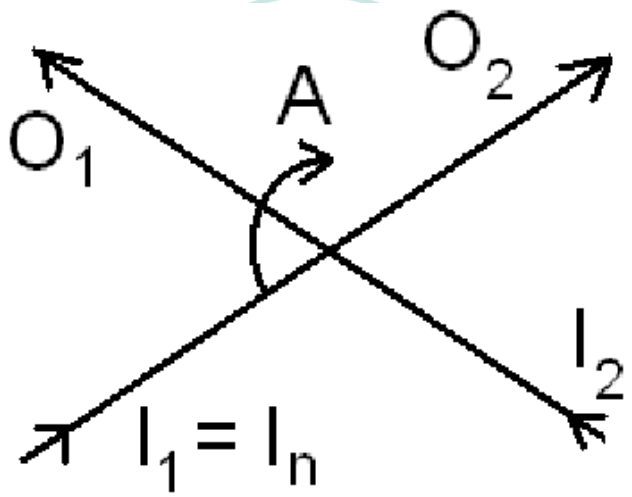


Figura 17.1: Representación de *Inflows* y *Outflows* junto con el ángulo A medido en sentido horario con respecto a I_n [12].

La representación del algoritmo Curv2 se observa en la figura 18.1 y a continuación se muestra el pseudocódigo para este algoritmo de navegación [12]:

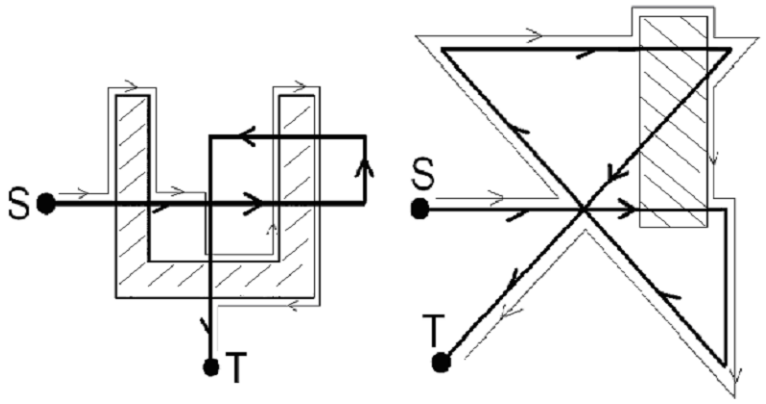


Figura 18.1: Representación de dos BCS del algoritmo Curv2 [12].

1. Establecer el contador C en cero. Iniciar desde el punto S. Ejecutar el paso 2.
2. Moverse sobre la curva ST hasta que una de las siguientes situaciones ocurra:
   - El robot alcanza el objetivo T = Finalizar satisfactoriamente.
   - El robot encuentra un obstáculo = Siga el límite del obstáculo. Ejecutar el paso 3.
   - Una intersección es encontrada = Iniciar A = 0. Ejecutar el paso 4.
3. Siga el límite o borde del obstáculo hasta que una de las siguientes situaciones ocurra:
   - El robot alcanza el objetivo T = Finalizar satisfactoriamente.
   - La curva ST es alcanzada en un punto P. Uno de los siguientes pasos es ejecutado:
     - El contador C lee cero. En el punto P, MA se puede mover a lo largo de la curva ST hacia T. Dejar el obstáculo y seguir la curva ST. Ejecutar el paso 2.
     - El contador C no es cero. En el punto P, MA se puede mover a lo largo de la curva ST hacia T. Decrementar el contador C en 1. Continuar moviéndose a lo largo de la frontera del obstáculo. Ejecutar el paso 3.
     - En el punto P, MA no se puede mover a lo largo de la curva ST hacia T. Incrementar el contador C en 1. Continuar moviéndose a lo largo de la frontera del obstáculo. Ejecutar el paso 3.
4. Incrementar A hasta que una de las siguientes situaciones ocurra:
   - Una salida O_j es encontrada y el contador C lee cero. Dejar la intersección en sentido O_j. Ejecutar el paso 2.
   - Una salida O_j es encontrada y el contador C no es cero. Decrementar el contador C en 1. Ejecutar el paso 4.
   - Un afluente I_j es encontrado. Incrementar el contador C en 1. Ejecutar el paso 4.

3.3. ALGORITMO CURV3

Con este algoritmo se pueden tratar obstáculos dinámicos que entran y salen del BCS, mientras el robot se encuentra en su tarea de navegación. Se asumen varias condiciones como que cada obstáculo puede estar en dos estados "On" u "Off". Si el robot se encuentra moviéndose por el borde de un obstáculo, este obstáculo no puede pasar a estado "Off". Un obstáculo puede ser removido del BCS, como una transición "On"/"Off". Un obstáculo no puede cambiar a estado "On" si el robot se encuentra en el espacio ocupado por el perímetro externo del obstáculo. Esto incluye cavidades que puedan estar dentro den obstáculo, y mucho menos el robot puede quedar atrapado dentro de la cavidad de un obstáculo, previniendo que el robot alcance el objetivo [12]. Asumir lo anterior es una realidad en la industria.

Los obstáculos no pueden ser dinámicos para el algoritmo Curv2, para que este garantice una terminación. Para el algoritmo Curv2 existe un concepto de pistas guías de navegación múltiples, el cual tiene un principio de emparejamiento de Afluente/Salidas. Los emparejamientos complejos son posibles por medio de un entero no negativo finito Z, con cada afluente para una intersección. La figura 20.1 muestra una asociación por medio de Z. Para la figura 19.1, pueden existir emparejamientos como S_1T_1, S_2T_2, S_1T_2 y S_2T_1, de tal manera que un punto S puede tener dos posibles rutas para dirigirse a un objetivo T.

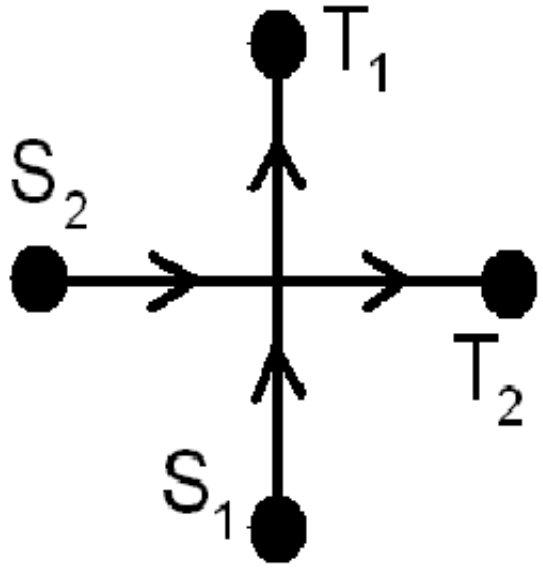


Figura 19.1: Emparejamiento S_1/T_1 y S_2/T_2 para un BCS con múltiples pistas de navegación [12].

Z representa el número de afluentes que el robot debe contar antes de ser habilitado para dejar una intersección. Aquí es donde el algoritmo Curv3 es desarrollado para acomodar Z. El funcionamiento de Z para el algoritmo Curv3 se muestra en la figura 20.1 y a continuación se muestra el pseudocódigo para este algoritmo de navegación [12]:

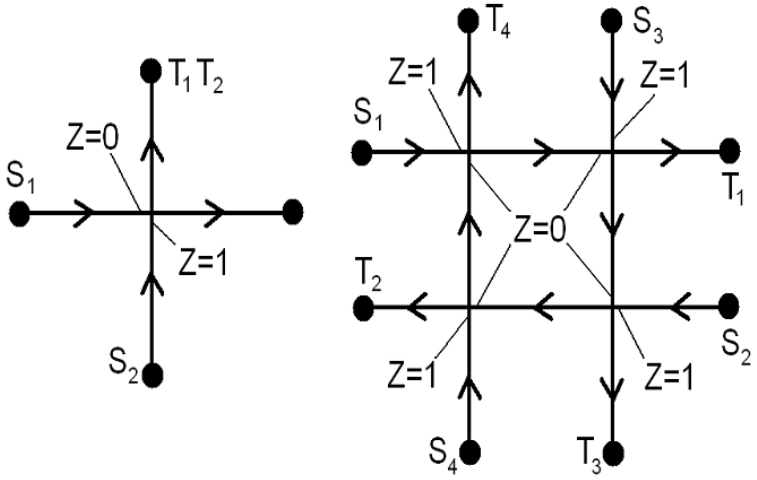


Figura 20.1: Principio de funcionamiento de Z con el algoritmo Curv3 [12].

1. Establecer el contador C en cero. Iniciar desde el punto S. Ejecutar el paso 2.
2. Moverse sobre la curva ST hasta que una de las siguientes situaciones ocurra:
   - El robot alcanza el objetivo T = Finalizar satisfactoriamente.
   - El robot encuentra un obstáculo = Siga el límite del obstáculo en la dirección local izquierda. Ejecutar el paso 3.
   - Una intersección es encontrada = Iniciar A = 0 y Z al valor asociado con el afluente. Si Z = 0 ejecutar el paso 5. De lo contrario ejecutar el paso 4.
3. Siga el límite o borde del obstáculo hasta que una de las siguientes situaciones ocurra:
   - El robot alcanza el objetivo T = Finalizar satisfactoriamente.
   - La curva ST es alcanzada en un punto P. Uno de los siguientes pasos es ejecutado:
     - El contador C lee cero. En el punto P, MA se puede mover a lo largo de la curva ST hacia T. Dejar el obstáculo y seguir la curva ST. Ejecutar el paso 2.
     - El contador C no es cero. En el punto P, MA se puede mover a lo largo de la curva ST hacia T. Decrementar el contador C en 1. Continuar moviéndose a lo largo de la frontera del obstáculo. Ejecutar el paso 3.
     - En el punto P, MA no se puede mover a lo largo de la curva ST hacia T. Incrementar el contador C en 1. Continuar moviéndose a lo largo de la frontera del obstáculo. Ejecutar el paso 3.
4. Incrementar A hasta que un afluente I_n sea encontrado. Decrementar Z. Si Z = 0 ejecutar el paso 5. De lo contrario repetir el paso 4.

5. Incrementar A hasta que una de las siguientes situaciones ocurra:
    - Una salida O_n es encontrada y el contador C lee cero. Dejar la intersección en sentido O_n. Ejecutar el paso 2.
    - Una salida O_n es encontrada y el contador C no es cero. Decrementar el contador C en 1. Ejecutar el paso 4.
    - Un afluente I_n es encontrado. Incrementar el contador C en 1. Ejecutar el paso 5.

4. BREVE ANÁLISIS DE LOS ALGORITMOS BUG

Los algoritmos Bug garantizan una finalización para la navegación autónoma de robots en entornos conocidos y desconocidos. Esta finalización puede ser satisfactoria o insatisfactoria, entendiendo que la finalización satisfactoria es en la que el robot llega hasta el punto deseado u objetivo. Fundamentalmente tienen similitudes, pero difieren en como son modificados para garantizar una finalización. El consumo de recursos puede ser una característica importante para la selección de alguno de estos métodos [13].

Otros algoritmos que no pertenecen a la familia Bug no garantizan que exista una finalización, debido a que el algoritmo puede entrar en un bucle infinito o "loop", e incluso tomar un tiempo de respuesta indeterminado para encontrar una solución. El robot rodeará el objetivo intentando determinar una ruta óptima indefinidamente debido a este bucle. Aunque no garantice una finalización finita con la implementación de un algoritmo no perteneciente a los Bug, esto ha servido para la determinación de las condiciones que llevan a estos bucles y el desarrollo de métodos que los eviten [11, 12]. Algunos de estos métodos se mostrarán a continuación ya que son de importancia para mejorar la eficiencia de un algoritmo de navegación.

Todos los algoritmos Bug usan por lo menos dos modos de operación [1, 12, 13]:

1. *Move-to-goal*: Moverse al objetivo.
2. *Boundary-Following*: Seguimiento del contorno o límite de un obstáculo.

Las reglas para dejar de seguir la frontera del obstáculo y moverse hacia el objetivo (leaving) siempre comparan el punto potencial para alejarse "leave" (que puede ser diferente a la posición actual del robot debido al rango de alcance de los sensores) con el punto cercano visitado o localizado por sensores. Estas reglas establecen si el robot debe partir o no hacia el objetivo en puntos donde lo había hecho anteriormente [12, 13]. Realmente debe considerarse si estas reglas son fundamentales y si son propiedades esenciales de cualquier algoritmo, o si existe alguna otra manera de resolver el problema.

El modo 1 es único curso de acción lógico en donde el robot se dirige directamente al objetivo sin saber del BCS. La longitud de estas rutas debe ser finita ya que la distancia desde cualquier punto hacia el objetivo en el plano bidimensional es finita. Sin embargo, existen obstáculos mientras el robot se mueve hacia el objetivo y por lo tanto la necesidad del modo 2 [1, 12, 13].

La máxima longitud de la ruta de cualquier segmento del modo 2 es el perímetro del obstáculo que se está siguiendo. Una vez el robot haya recorrido la longitud total del perímetro del obstáculo y no se haya establecido un punto de partida para alejarse de él, se concluye que el objetivo es inalcanzable y se finaliza el algoritmo para no almacenar más datos por circunnavegación [1]. Conociendo que las longitudes de los dos modos son finitas, lo único que puede hacer una distancia infinita es la condición de partida o "leave", y por esto su importancia en el estudio de los algoritmos Bug [11].

La importancia de la condición de partida se realizó en el desarrollo del algoritmo Alg2, cuya conclusión breve fue que en el trayecto del inicio al objetivo el robot encontrará un número finito de obstáculos, y no permitir que nuevos obstáculos sean adicionados cuando el robot se encamine al objetivo. Es necesario establecer que el número de obstáculos en un perímetro dentro del BCS sea finito y sin posibilidad de adicionarse más, o de lo contrario el robot podría encontrarse en una situación en donde considere un número infinito de obstáculos, y su tarea de navegación jamás finalice [12]. La figura 21.1 representa un ejemplo de esta situación.

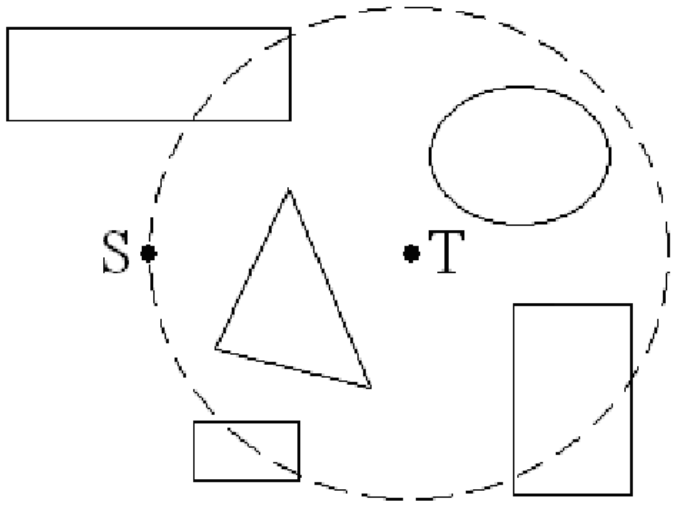


Figura 21.1: Representación de un número infinito de obstáculos dentro del *Shrinking-Disc* (circunferencia punteada) [12].

Para la figura 21.1 se asume la circunferencia punteada denominada "disco en contracción" S-D (del inglés *Shrinking-Disc*) como un perímetro de trabajo del robot dentro del BCS, S el punto de inicio y T el objetivo. A medida que el robot ejecuta el modo 1 (moviéndose hacia el objetivo) el radio del S-D se va reduciendo y ningún obstáculo nuevo puede aparecer. Cuando el robot pasa al modo 2 (siguiendo la frontera del obstáculo) debe asegurarse que no aparezcan nuevos obstáculos cuando el robot parta hacia T. Por lo tanto, la necesidad de partir solo cuando el robot está más cerca al objetivo con respecto a cualquier otro punto L visitado o detectado por medio de sensores con anterioridad.

Para satisfacer este requerimiento, el único método es comparar $d(x,T)$ con $d(L,T)$. Si $d(x,T) < d(L,T)$, entonces el robot puede partir y el BCS no admitirá nuevos obstáculos dentro del disco. Esto verifica efectivamente si el robot se encuentra dentro del S-D centrado en T con radio $d(L,T)$. Claramente no hay otra limitante que satisfaga este requerimiento por los siguientes argumentos [12]:

- Una limitante de gran tamaño podría admitir nuevos obstáculos dentro del perímetro.
- Una limitante de tamaño muy pequeño podría ocasionar que el robot empiece una circunnavegación y se determine erróneamente que el objetivo es inalcanzable.

Los algoritmos Class1 y Com que no pertenecen a la familia Bug pero son empleados para el desarrollo de algoritmos Bug eficientes, son un medio para ilustrar los argumentos anteriores.

4.1. MÉTODOS DE FINALIZACIÓN DE LOS ALGORITMOS BUG Y GENERACIÓN DE NUEVOS ALGORITMOS ONEBUG, MULTIBUG Y LEAVEBUG

En la figura 21.1 se muestra un perímetro de trabajo del robot o S-D y se establece que al reducirse el S-D es posible que ocurra una circunnavegación. Para sobreponerse al problema S-D se necesita implementar otra regla que garantice que el número de puntos de partida "*leave*" L_k sea finito. Se han considerado cinco métodos para la solución de este problema [12].

4.1.1. Método de Punto más Cercano

Cada obstáculo tiene una distancia mínima con el objetivo y todos los puntos que satisfacen esto se encuentran en C. El robot solo puede partir desde los puntos en C.

Este método funciona indicando que si el robot parte hacia el objetivo desde un punto en C, el robot no encontrará ese obstáculo nuevamente a lo largo de su navegación. Por lo tanto, debe existir al menos un punto de partida asociado con cada número de obstáculos finitos dentro del conjunto de obstáculos intersecando el S-D.

El gran inconveniente para identificar C para un obstáculo en particular O, es que el robot debe circunnavegar completamente O y esto lleva a rutas innecesariamente largas. Bug1 es el único algoritmo capaz de emplear el método del Punto más Cercano [12].

4.1.2. Método de la Línea-M

Se crea una línea imaginaria M desde el punto d partida S hasta el objetivo T. El robot realiza la tarea "leave" sobre la línea M.

Con este método se crea una línea en el interior de S-D ya que todos los puntos de M también son los puntos con S-D. La intersección de M con un obstáculo crea un punto único y distinto de otros. Asumiendo que el número de intersecciones con la línea M es finito, existe también un número finito de puntos "leave" L_k en toda la tarea de navegación del robot.

El inconveniente con este método es que se pueden crear muchos ciclos si hay demasiadas intersecciones con la línea M. Los algoritmos que usan este método son Bug2, Alg1, Rev1 [12].

4.1.3. Método de segmentos inhabilitados

Mantiene el número de puntos H_i finitos que ocurren a lo largo de los segmentos inhabilitados.

Para cualquier obstáculo encontrado existe un número finito y diferente de cero de segmentos inhabilitados, y de manera similar existe un número finito y posiblemente cero de segmentos habilitados. Un segmento inhabilitado existe cuando el robot no puede dirigirse hacia el objetivo por todos los puntos del segmento. Un segmentos habilitado ocurre existe el robot puede dirigirse hacia el objetivo por todos los puntos de un segmento.

El robot solo puede encontrar un obstáculo en un segmento deshabilitado y si esos encuentros pueden ser restringidos de acuerdo al siguiente teorema [12]:

- Un robot nunca puede registrar un punto H_i a lo largo de un segmento que ya haya visitado en el contorno de un obstáculo. Durante el modo 1 (moviéndose al objetivo) la distancia decrece monótonamente. Si se encuentra un obstáculo entonces el punto H_i será más cercano al objetivo que cualquier otro punto visitado anteriormente y por lo tanto, no puede estar a lo largo de un segmento del contorno de un obstáculo anteriormente visitado.

Con esto, el número de puntos H_i se mantiene finito. La figura 22.1 ilustra este método.

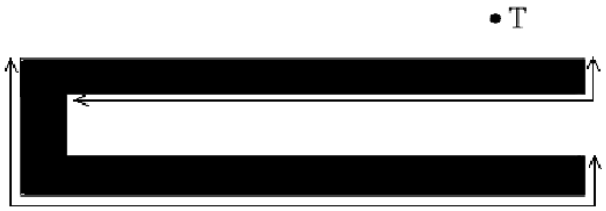


Figura 22.1: Representación del método de segmentos inhabilitados [12].

Existen varias maneras de implementar este método. Por ejemplo, el algoritmo Alg2 lo implementa cuando por fuerza hace que el robot regrese al anterior punto H_i establecido y explore en sentido antihorario el seguimiento de pared restante. Similarmente lo hace el Rev2 forzando al robot a retornar al punto más cercano de la lista H y explorar la dirección alterna. Estas reglas aseguran que existirá un número finito de puntos H_i por segmento deshabilitado.

Obviamente hay muchas maneras de implementar este método. Se considera un nuevo algoritmo OneBug en el cual el robot debe explorar el segmento deshabilitado antes de explorar un segmento habilitado vecino. OneBug puede ser descrito como un algoritmo que funciona como el Alg2 sin los puntos almacenados, y Class1 sin las rutas arbitrariamente largas [12]. La operación del algoritmo OneBug se muestra en la figura 23.1.

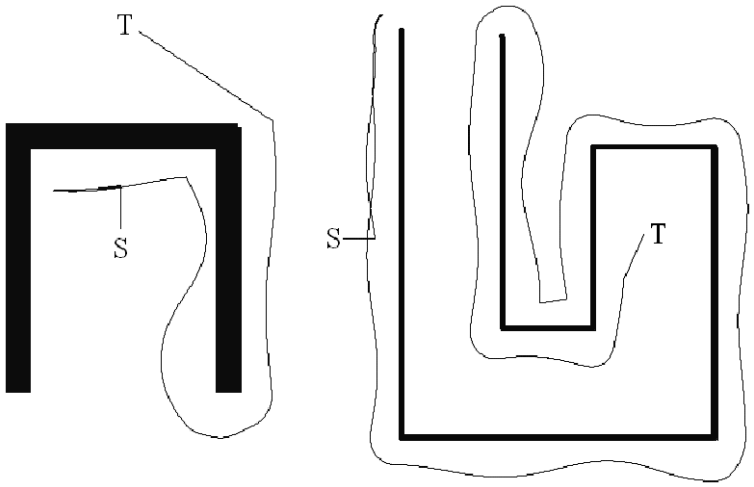


Figura 23.1: Representación del algoritmo OneBug en dos diferentes BCS A y B [12].

Los puntos almacenados en Alg2 también pueden ser empleados para reducir la longitud de ruta. Aunque esto se puede contradecir al comparar las figuras 5.1 y 6.1 y la razón de esto es porque Alg2 no permite continuar la búsqueda desde el sentido direccional de la ruta cerca a regiones inexploradas, pero elige el último punto H_i definido.

Con esto, ahora se define un nuevo algoritmo que es el MultiBug. A continuación se muestra el pseudocódigo para este algoritmo de navegación [12]:

1. Moverse directamente hacia el objetivo T hasta que una de las siguientes situaciones ocurra:
   - El robot alcanza el objetivo = Finalizar satisfactoriamente.
   - Un obstáculo es encontrado = Almacenar el punto como H_i. Ejecutar el paso 2.
2. Realizar una circunnavegación horaria hasta que una de las siguientes situaciones ocurra:
   - El robot está en un punto más cercano al objetivo que cualquier otro visitado anteriormente y puede dirigirse directamente hacia el objetivo. Ejecutar el paso 1.
   - El robot encuentra un punto H_k previamente almacenado. Ejecute el paso 3.
   - El robot encuentra H_i. Finalizar como un fallo en la navegación.
3. Decidir cual dirección para seguir la pared tomará el robot hacia el punto H_k más cercano que solo haya registrado una exploración horaria. Regresar a ese punto H_k y sobre él, realizar un seguimiento de pared antihorario hasta que una de las siguientes situaciones ocurra:
   - El robot está en un punto más cercano al objetivo de los que se han visitado anteriormente y puede dirigirse directamente hacia el objetivo. Ejecutar el paso 1.
   - El robot regresa a el punto H_k. Finalizar como un fallo en la navegación.

Este algoritmo permite que el robot elija la circunnavegación horaria o antihoraria para explorar las regiones deshabilitadas cercanas. El número de puntos H_i por segmento deshabilitado permanece finito debido a que al regresar a un punto H_k almacenado previamente en el paso 3 del pseudocódigo MultiBug, el robot debe elegir seguir la pared en una dirección antihoraria. Esto significa que el segmento deshabilitado asociado con H_k debe ser completamente explorado usando máximo dos encuentros [12]. La figura 24.1 muestra el funcionamiento del algoritmo Multibug en dos BCS diferentes.

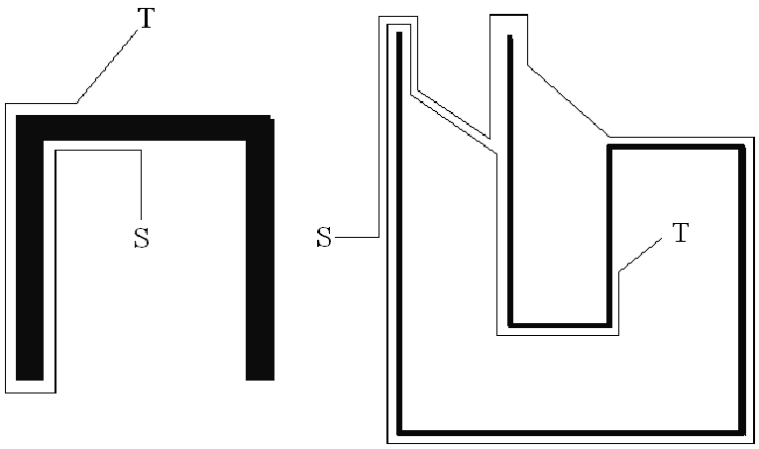


Figura 24.1: Representación del algoritmo MultiBug en dos BCS diferentes A y B [12].

4.1.4. Método STEP

Partir hacia el objetivo solo cuando el robot se encuentre a una distancia predefinida, STEP, dentro del S-D.

Empleando este método, el robot puede partir como máximo $\left[{}^{d(S,T)}/_{STEP} \right]$ veces en su navegación total. Desafortunadamente este método requiere conocimiento de los alrededores ya que STEP debe ser seleccionado de tal manera que durante la circunnavegación, un punto en cada obstáculo encontrado estará en un segmento habilitado que se encontrará a una distancia STEP cercana al objetivo. Si esto no se mantiene, entonces es posible que el robot circunnavegue un obstáculo y determine erróneamente que el objetivo es inalcanzable. El método STEP es empleado en el algoritmo DistBug tanto para los sensores táctiles como los de proximidad (alcance) [12].

4.1.5. Método del mínimo local

Restringir puntos H_k a los mínimos locales que ocurran en los obstáculos encontrados.

Usando este método el robot puede definir únicamente los puntos H_k, si este detecta que está en un mínimo local en el obstáculo encontrado con respecto al objetivo. El número de mínimos locales de cualquier obstáculo es finito. El algoritmo TangentBug emplea el método de mínimo local [12]. La figura 25.1 muestra en donde se ubican los mínimos locales entre un obstáculo y el objetivo para dos BCS diferentes.

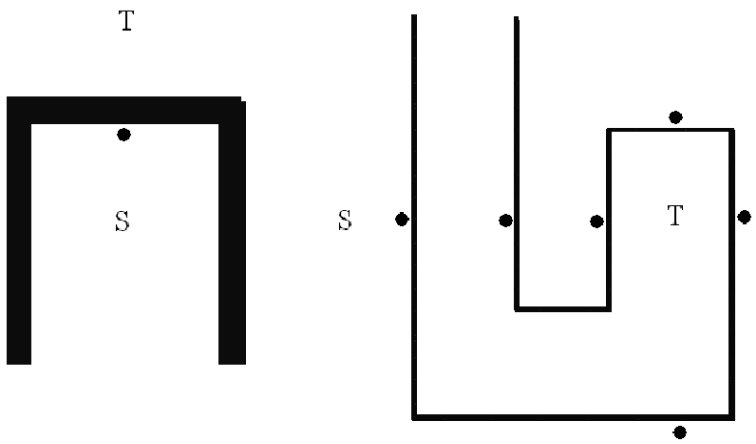


Figura 25.1: Ubicación de los mínimos locales cercanos al objetivo T en dos BCS diferentes A y B [12].

4.1.6. Método de segmento habilitado

Mantener el número de puntos H_k como un valor finito que puedan ocurrir a lo largo de los segmentos habilitados.

El método de segmentos habilitados emplea la certeza que existe únicamente un número finito de segmentos de partida "leave" para un obstáculo dado. Se considera un nuevo algoritmo LeaveBug que implementa el método de segmentos habilitados. La figura 26.1 muestra la operación del algoritmo y a continuación se muestra el pseudocódigo para este algoritmo de navegación [12]:

1. Moverse directamente hacia el objetivo T hasta que una de las siguientes situaciones ocurra:
   - El robot alcanza el objetivo = Finalizar satisfactoriamente.
   - Un obstáculo es encontrado = Ejecutar el paso 2.
2. Realizar una circunnavegación horaria hasta que una de las siguientes situaciones ocurra:
   - El robot alcanza el objetivo = Finalizar satisfactoriamente.
   - El robot puede dirigirse directamente al objetivo. Ejecute el paso 3.
   - El robot completa la circunnavegación total del obstáculo. Se determina que el objetivo es inalcanzable y finaliza la navegación.
3. Realizar una circunnavegación horaria mientras se actualiza un punto de L hasta x si el robot se encuentra más cerca del objetivo con respecto a algún otro punto anteriormente visitado. Hacer esto hasta que una de las siguientes situaciones ocurra:

- El robot alcanza el objetivo = Finalizar satisfactoriamente.
- El robot es incapaz de moverse hacia el objetivo. Si L registra una posición, ejecutar el paso 4. De lo contrario ejecutar el paso 2.
- El robot completa la circunnavegación total del obstáculo. Se determina que el objetivo es inalcanzable y finaliza la navegación.

4. Si $d_{ruta}(x, L)$ es cero, resetear L como nulo y proceder al paso 2. De lo contrario, realizar una circunnavegación antihoraria hasta que el robot regrese a L. Una vez en L, resetear L a un valor nulo y proceder con el paso 1.

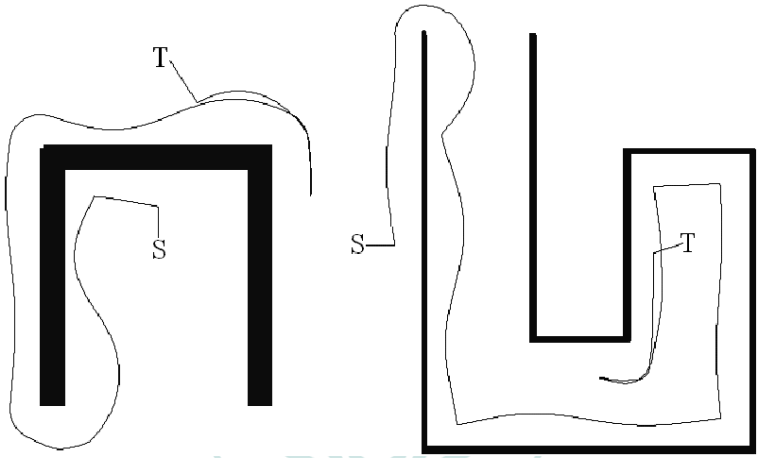


Figura 26.1: Representación del algoritmo LeaveBug en dos BCS diferentes A y B [12].

4.1.7. El método Q

Existe un grupo único de puntos Q que es finito para cualquier obstáculo poligonal O. Q es el grupo de puntos en el cual el robot debe detenerse y obtener datos para continuar circunnavegando a O, tanto en sentido horario como antihorario. Esta circunnavegación horaria y antihoraria debe realizarse desde un vértice de O con el propósito de determinar los miembros de O [12].

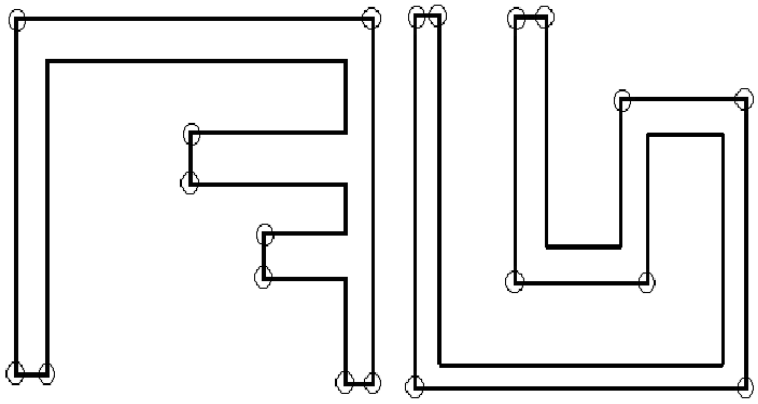


Figura 27.1: Puntos Q del obstáculo O, encerrados en círculos [12].

En la figura 27.1 se observan los puntos circulares que representan los miembros de Q, asumiendo que el robot posee sensores con rango infinito. El método Q se empleó para el desarrollo del algoritmo SensorBug, pero algunos de los inconvenientes es el cómo implementarlo. Pueden plantearse tres interrogantes para este tipo de método [12]:

1. ¿Cómo se puede garantizar que el robot siempre partirá del O hacia T desde un punto Q?
2. ¿Qué sucedería en el caso donde Q esté vacío, por ejemplo si el robot inicia dentro de una caja?
3. ¿Cómo garantizar que el robot siempre dejará un obstáculo el cual no encierre al objetivo?

El método Q posee un gran inconveniente el cual requiere un obstáculo poligonal para que así el número de vértices sea finito. De lo contrario, Q podría ser infinito en una sección curva de O. Además el robot debe tener sensores de proximidad, porque de existir un sensor táctil Q será infinito.

Dos principales ventajas de este método son [12]:

- Puede ser fácilmente implementado en robots que usen sensores de proximidad comparados con TangentBug.
- El robot no tiene que estar dependiendo continuamente de decisiones relacionadas con el BCS mientras navega entre miembros de Q. En lugar de eso, puede libremente dedicar recursos computacionales para otras tareas.

5. ANÁLISIS DE RESULTADOS Y COMPARACIÓN DE RENDIMIENTO

Para los algoritmos Bug, el robot debe finalizar su navegación si este llega al objetivo o determina que el objetivo es inalcanzable, pero sin realizar un mapeo del BCS. Un algoritmo de navegación puede presentar estadísticamente un mejor rendimiento computacional que otro, pero no necesariamente eso significa un mejor desempeño al enfrentarse a un BCS.

Desde que cada algoritmo de la familia Bug debe tener la propiedad de finalizar, se han desarrollado algoritmos Bug que traten de mejorar el rendimiento del algoritmo [1, 12, 13]. Por ejemplo, mejorar la condición leaving del robot ante un obstáculo o considerar un cambio en la dirección de navegación cada vez que se encuentre un obstáculo. Con la variedad de algoritmos y las necesidades a satisfacer, algo apropiado es identificar que técnicas de navegación pueden ofrecer mejor rendimiento y trabajar mejor, con resultados que evalúen diferentes criterios de funcionamiento en diferentes BCS. Se pueden mencionar dos algoritmos como lo son el LeaveBug y OneBug.

El algoritmo LeaveBug posee similitudes con el Bug1, excepto que en lugar de circunnavegar el obstáculo completamente antes de evaluar una línea de segmento $[Q_m, T]$, el robot evalúa esta condición después de completar cada segmento de la ruta que no evita al robot dirigirse directamente al objetivo.

El algoritmo OneBug posee similitudes con Alg2, excepto que los puntos almacenados no son empleados. En lugar de ello, el robot explora completamente un segmento a lo largo del obstáculo que está evitando el desplazamiento directo del robot al objetivo.

Respecto a la frecuencia de actualización, ciertos algoritmos Bug continuamente actualizan datos de su posición actual, y detectan automáticamente que cualquiera de las condiciones de navegación se hayan cumplido. Por ejemplo, para el algoritmo Bug2 tan pronto que el robot se posiciona sobre la línea M, el robot detecta esto y actúa de acuerdo a las instrucciones.

En la práctica, esto requiere que el robot actualice los datos de su posición actual, y revise las condiciones de navegación. La posición del robot está basada en "cuenta muerta" y por cada actualización, los codificadores de tracción deben ser leídos y se deben realizar cálculos. Claramente se necesitan recursos computacionales y las actualizaciones no pueden suceder con tanta frecuencia para cumplir con un criterio de autonomía.

Inicialmente la actualización de posición del robot y chequeo de condiciones eran realizadas como una tarea de segundo plano. Sin embargo estos procesos tienden a ser impredecibles, especialmente cuando el robot está en movimiento a alta velocidad o incluso en presencia de obstáculos y rutas irregulares. Asociar estos inconvenientes a una rutina principal requiere de mucho esfuerzo de programación.

A través de simulaciones y pruebas en BCS reales por algunos autores, una distancia de 40mm entre actualizaciones reflejó un balance óptimo [11, 12], entre actualizaciones con más frecuencia y menor frecuencia empleando el simulador EyeSim. Esto implica que en esos 40mm una posición significante incluye un margen de error.

Los algoritmos Bug en teoría usan pequeños puntos infinitesimales para representar el inicio S, el objetivo T, último punto H_k y otras posiciones relevantes, en Alg1, Alg2, Rev1 y Rev2. Pero en la práctica esto no es posible debido a que la frecuencia de actualización está sometida a límites, y desviaciones durante el modo seguidor de frontera. Por lo tanto, en la implementación cada posición significante es representada por cuadrados de lado de 50mm. El cuadrado es elegido por ser computacionalmente eficiente al analizar si el robot se encuentra dentro de esta área [12]. El lado del cuadrado se elige al ser ligeramente mayor a la distancia que recorre el robot en un ciclo de actualización, pero no tan grande como para poder generar falsos positivos frecuentes.

Se incluyen en esta sección tablas de comparación entre algoritmos Bug1, Bug2, Alg1, Alg2, DistBug, TangentBug, OneBug, LeaveBug, Rev1, Rev2 y Class1* para los BCS A y B. Los resultados son sustentados por medio de simulación de los algoritmos con el paquete de software EyeSim*.

Algunos algoritmos requieren solo de sensores táctiles como en el caso de Bug1 y Bug2. En estos algoritmos los sensores de proximidad son empleados solo como una asistencia para los modos de detección y seguidor de frontera.

En todos los algoritmos Bug la habilidad de revisar si el robot puede moverse directamente hacia el objetivo es esencial. Bug1 requiere revisar $[Q_m, T]$ en su prueba de alcance de objetivo. De igual manera los algoritmos Bug2, Alg1, Alg2, Rev1 y Rev2 requieren de esta revisión para futuros puntos de partida o "leave" L_k.

* Las simulaciones se realizaron en espacios de configuración abiertos y cerrados, produciendo resultados en los que la longitud de ruta fue un atributo importante para determinar el algoritmo más eficiente.

La figura 28.1 y 29.1 muestran la operación de los algoritmos Bug contenidos en este documento para los dos BCS A y B. Se considera para estos algoritmos la longitud de las rutas de navegación desde el inicio S hacia el objetivo T, un factor determinante para una simple comparación de eficiencia en la tarea de navegación.

La figura 28.1 muestra la operación simulada de los algoritmos de la familia Bug y los algoritmos no pertenecientes a la familia Bug OneBug, LeaveBug, Rev1, Rev2, Rev2 y Class1 contenidos en este documento para un BCS A. La gráfica 1 contiene información sobre la distancia de la ruta de navegación necesaria por cada algoritmo, para completar su tarea de navegación en el BCS A.

La figura 29.1 muestra la operación simulada de los algoritmos Bug para un BCS B. La gráfica 2 contiene información sobre la distancia de la ruta de navegación necesaria por cada algoritmo, para completar su tarea de navegación en el BCS B.

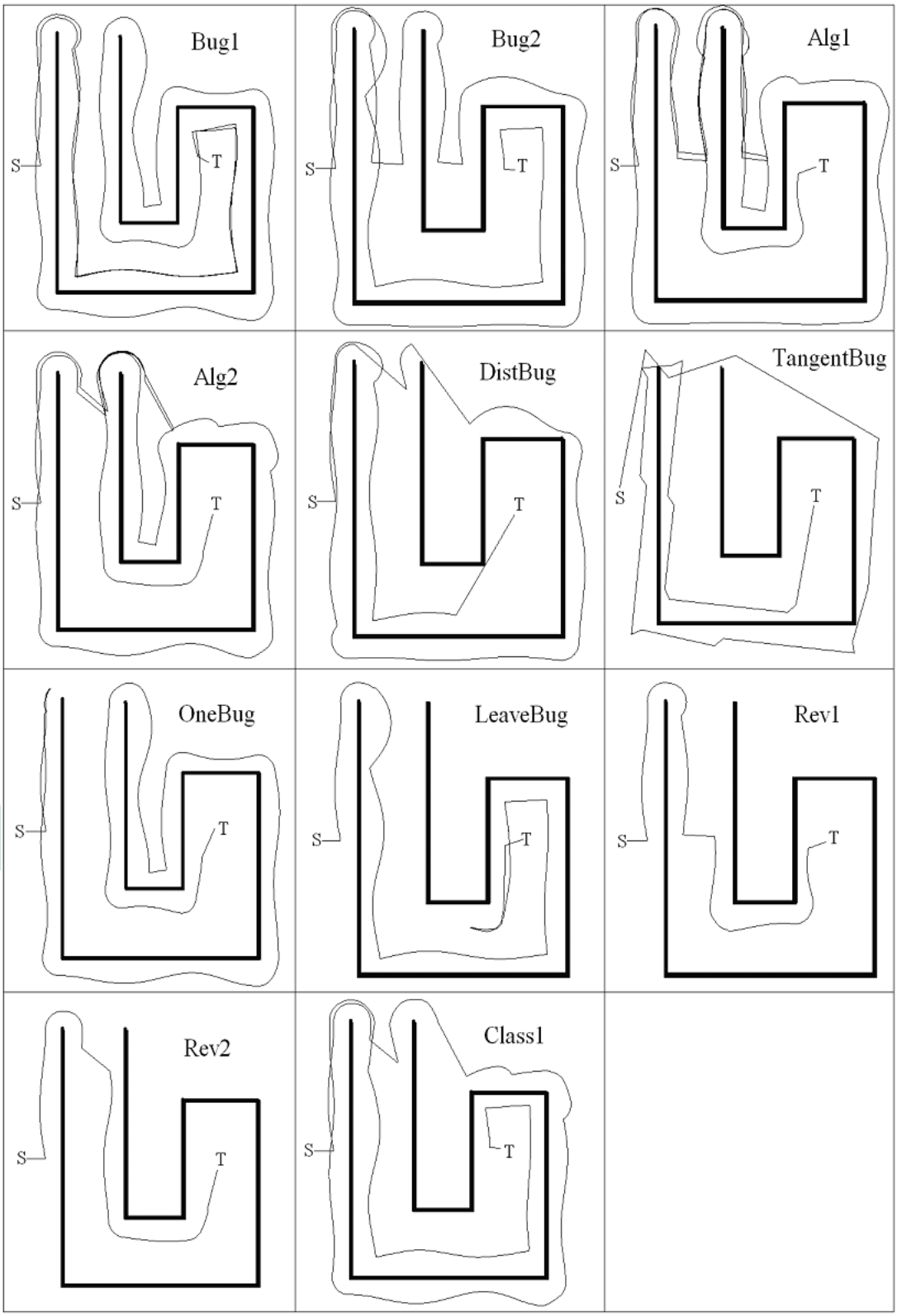
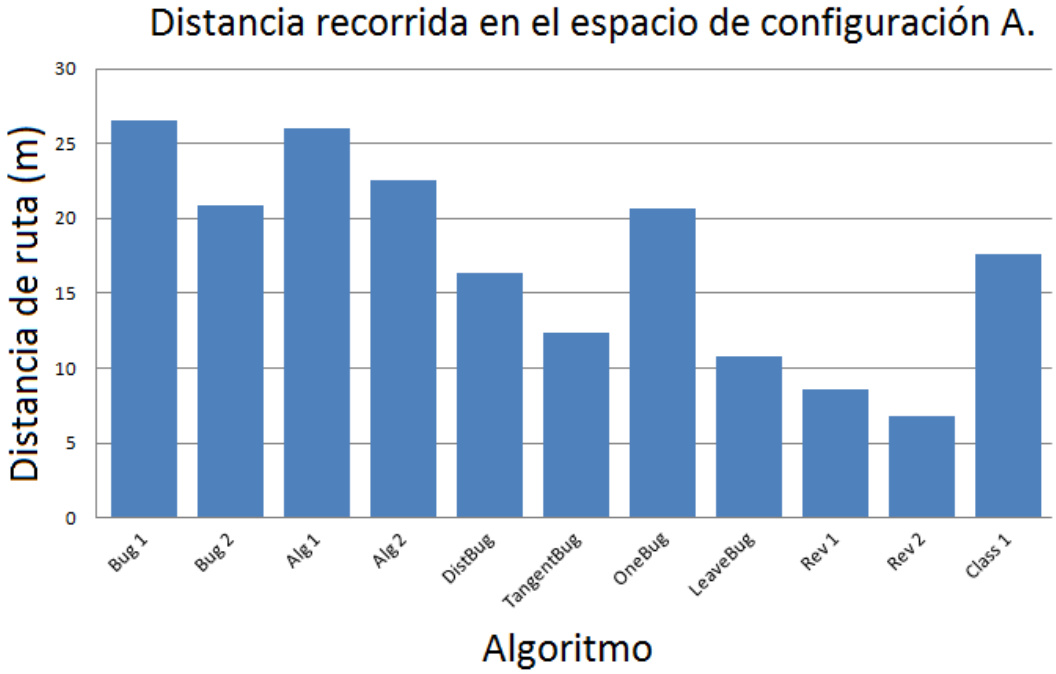


Figura 28.1: Representación de la simulación de los algoritmos en el BCS A [12].

Gráfica 1. Longitud de ruta de navegación para los algoritmos Bug en el BCS A [12].



En la figura 28.1 se observa en el trazado de la ruta hacia el objetivo T para los algoritmos simulados. Todos excepto el TangentBug mostraron curvas en su trayectoria, debido a errores rotacionales, métodos de aproximación en el cambio de modo seguidor de frontera y condiciones de hardware. Estos errores se producen cuando el robot no posee un elemento que le permita mantener una dirección en la que θ no varíe arbitrariamente. También los componentes del robot involucrados en el desplazamiento como lo son los motores, no poseen una similitud en cuanto a su relación de giro. Al ser una simulación, estos factores pueden ser ajustados para que los resultados se acerquen más a lo que se puede experimentar con un montaje real.

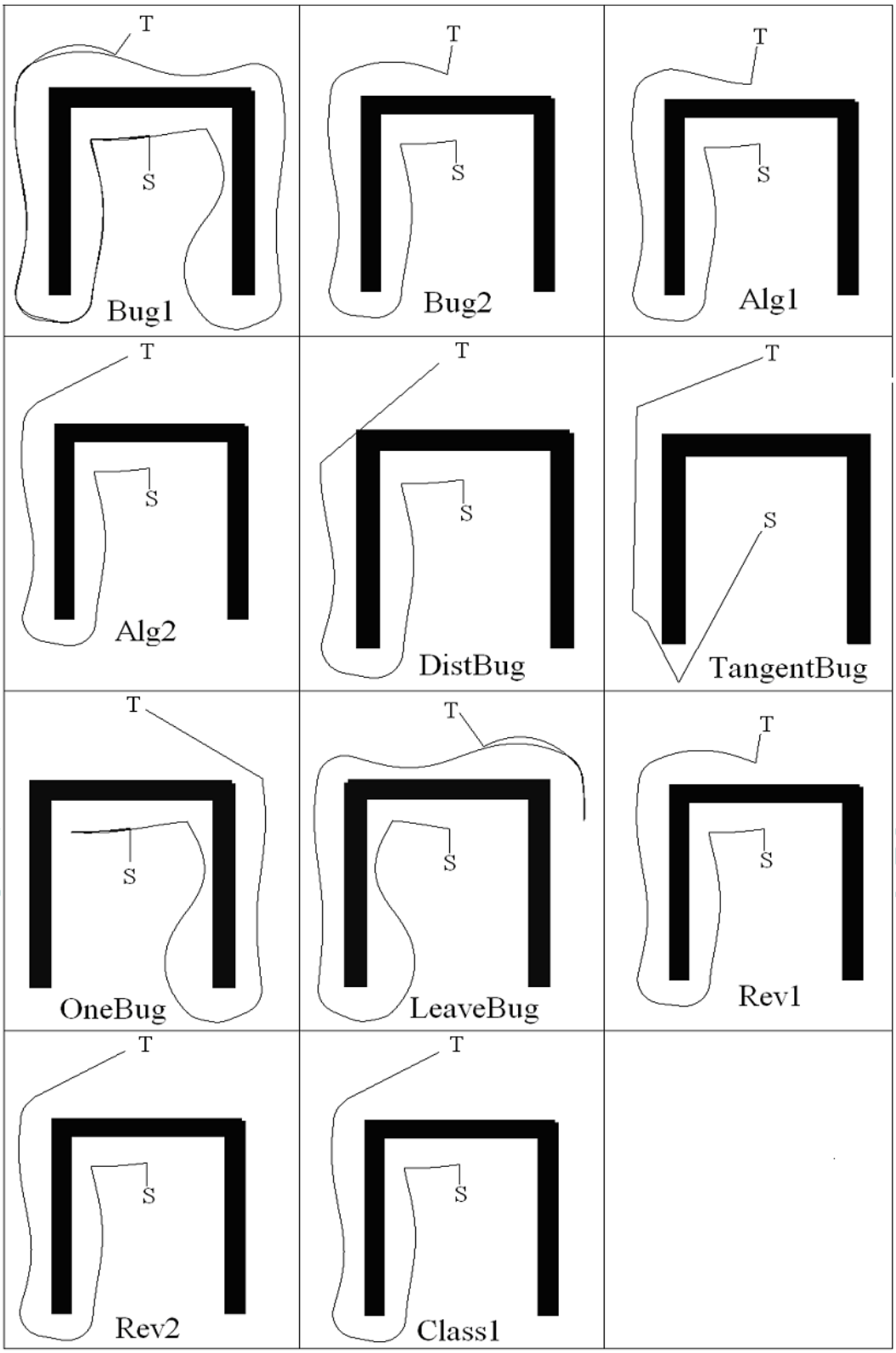
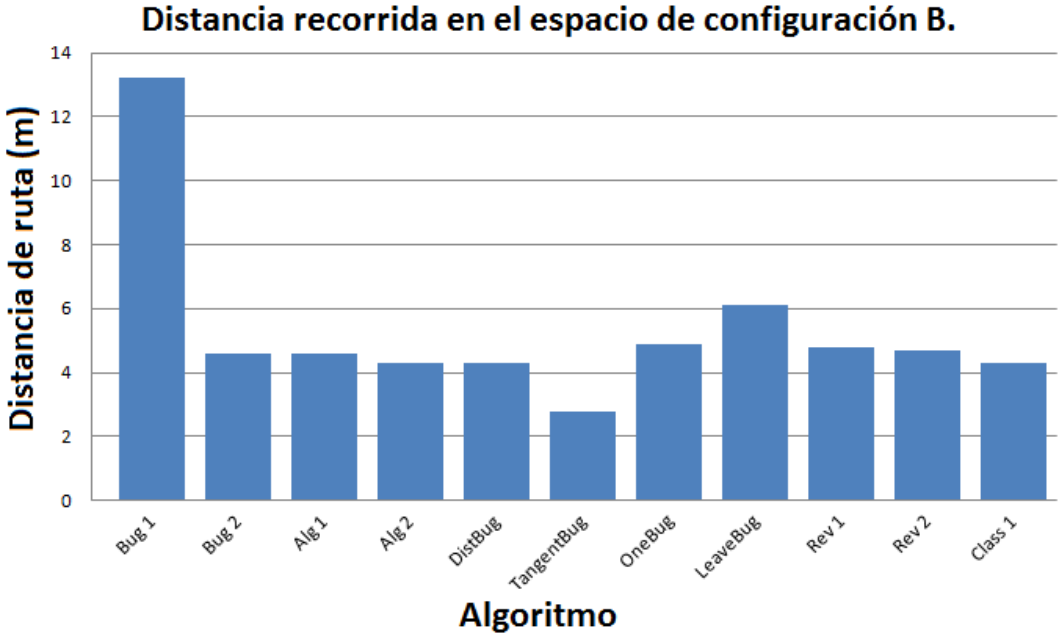


Figura 29.1: Representación de la simulación de los algoritmos para el BCS B [12].

Gráfica 2. Longitud de ruta de navegación para los algoritmos Bug en el BCS B [12].



La tabla 7 contiene información de los algoritmos comparados en esta sección del documento. PD significa que se usa un método proporcional derivativo, y para el tipo de sensor IR significa infrarrojo. La tabla muestra información básica de cada algoritmo según varios criterios, a manera de sumario para finalizar esta breve comparación entre los algoritmos Bug por parte del autor.

Tabla 7: Comparación de características de los algoritmos simulados [12].

Algoritmo	Longitud de ruta (m)	Algoritmo seguidor de frontera	Sensor	Revisión Leave	Gráfica Tangencial Local (LTG)	Detección de M
Bug1	107	PD	Táctil	Si	No	No
Bug2	74.3	PD	Táctil	Si	No	Si
Alg1	83.3	PD	Táctil	Si	No	Si
Alg2	76	PD	Táctil	Si	No	No
DistBug	64.5	PD	IR	No	No	No
TangentBug	46.5	No	IR	No	Si	No
OneBug	68.5	PD	Táctil	No	No	No
LeaveBug	65	PD	Táctil	No	No	No
Rev1	64.8	PD	Táctil	Si	No	Si
Rev2	62	PD	Táctil	Si	No	No
Class1	67.5	PD	Táctil	No	No	No

Las simulaciones realizadas contienen resultados para considerar el algoritmo de la familia Bug que mejor se desempeñó para los BCS A y B. Estos resultados se incluyen en la tabla 7 y muestran al algoritmo TangentBug como el más eficiente en cuanto a la distancia de ruta más corta, seguido del algoritmo DistBug. No obstante, el desempeño de los algoritmos Rev1 Rev2 son significantes en espacios cerrados, y pueden emplearse para tareas en las que sea necesario que el robot cambie la dirección 180 grados de su anterior desplazamiento cuando encuentre un obstáculo.

El mismo autor en su tesis del año 2005 comparó las simulaciones de los algoritmos de la familia Bug e incluyó el algoritmo D*, empleando los BCS B, C y D. Para esta tesis se simularon los algoritmos Bug1, Bug2, Alg1, Alg2, DistBug, TangentBug y D*. Los atributos por algoritmo tenidos en cuenta para estas simulaciones fueron longitud de ruta, rotación, rotación inherente, tiempo total de computación, computación por metro, llamado de funciones, error PSD, error de desplazamiento rotacional, simplicidad y requerimiento en memoria. A continuación se presentan los resultados de las simulaciones.

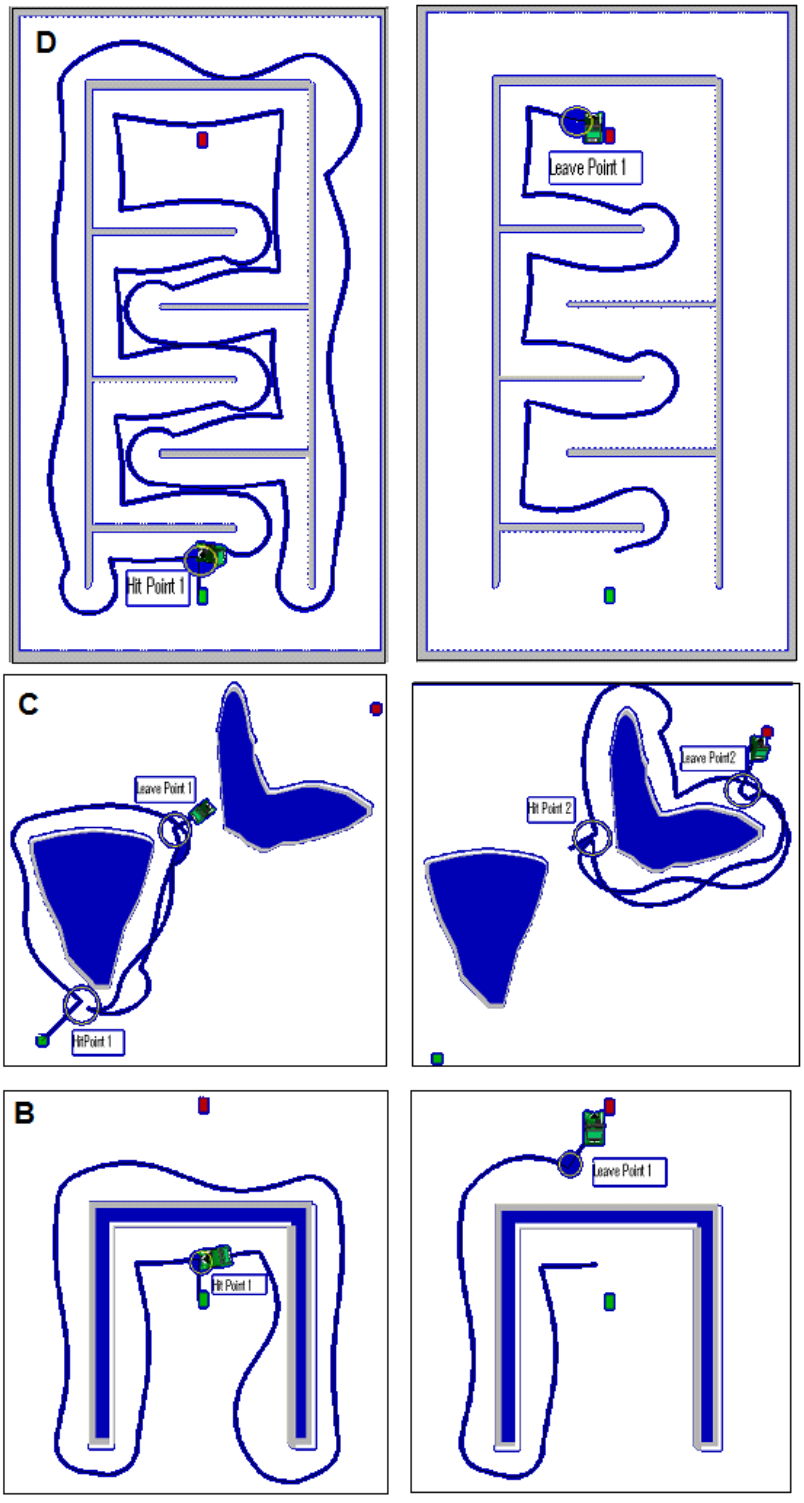


Figura 30.1: Respuesta de simulación del algoritmo Bug1 en BCS D, C y B [11].

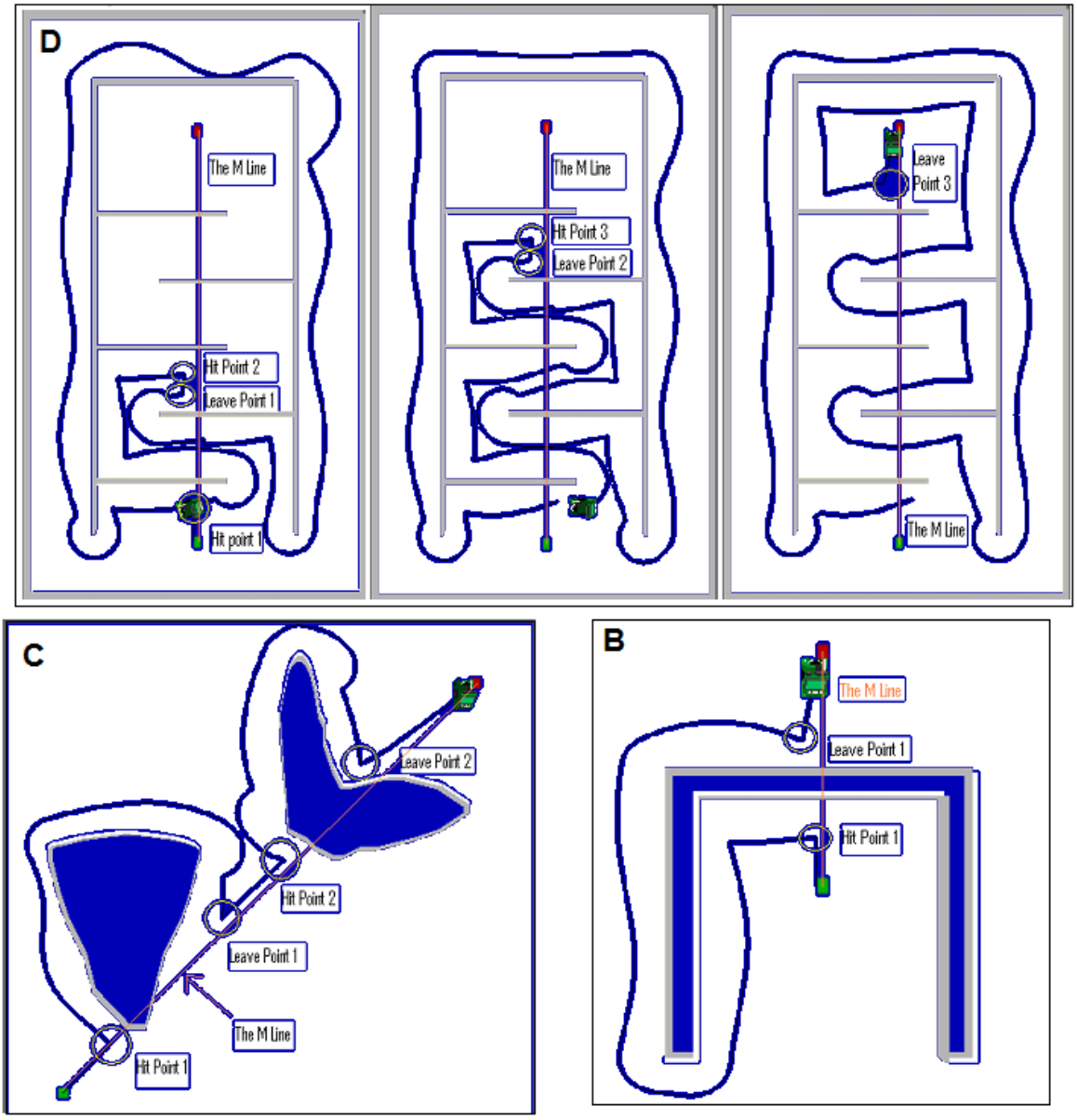


Figura 31.1: Respuesta de simulación del algoritmo Bug2 en BCS D, C y B [11].

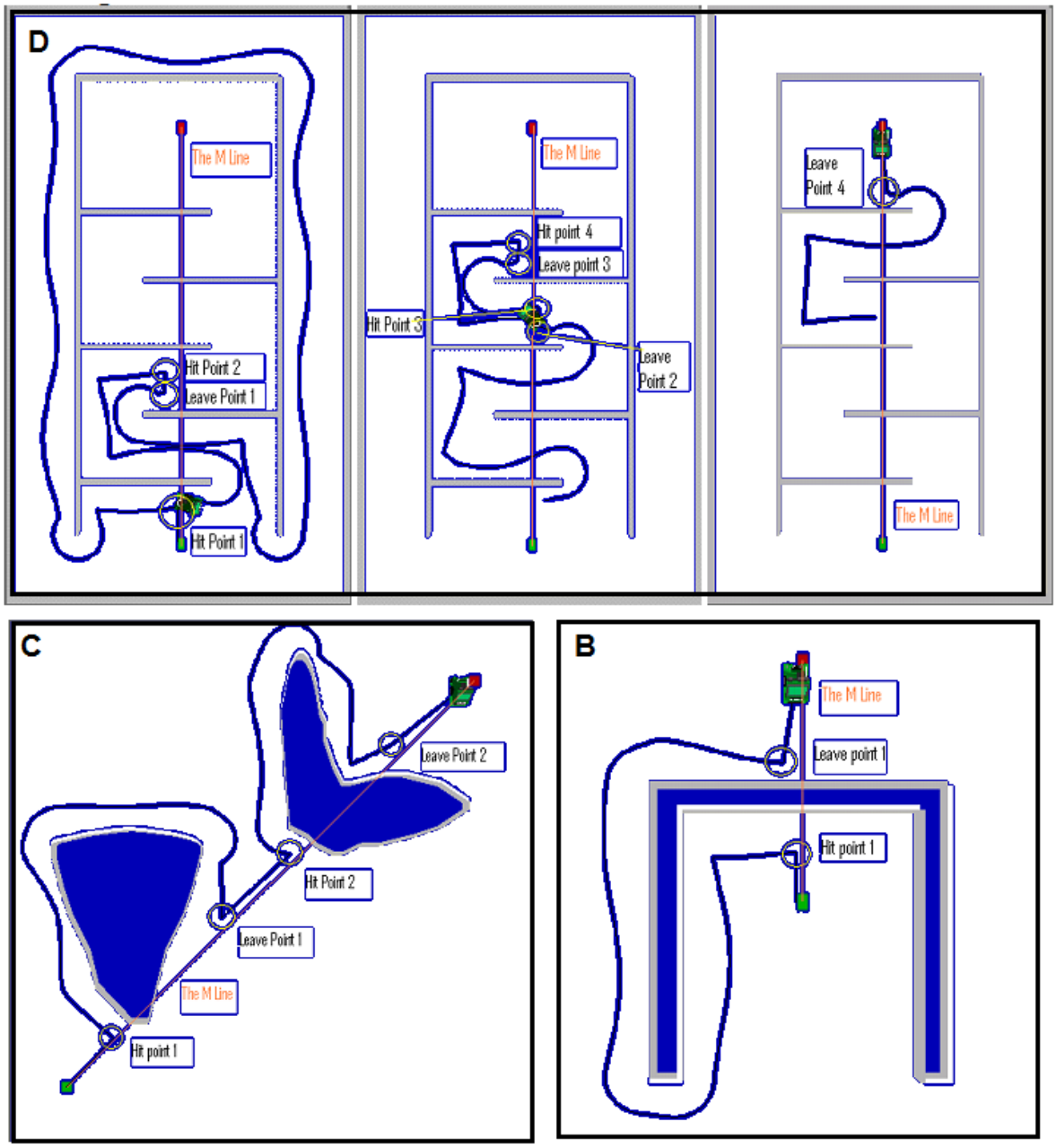


Figura 32.1: Respuesta de simulación del algoritmo Alg1 en BCS D, C y B [11].

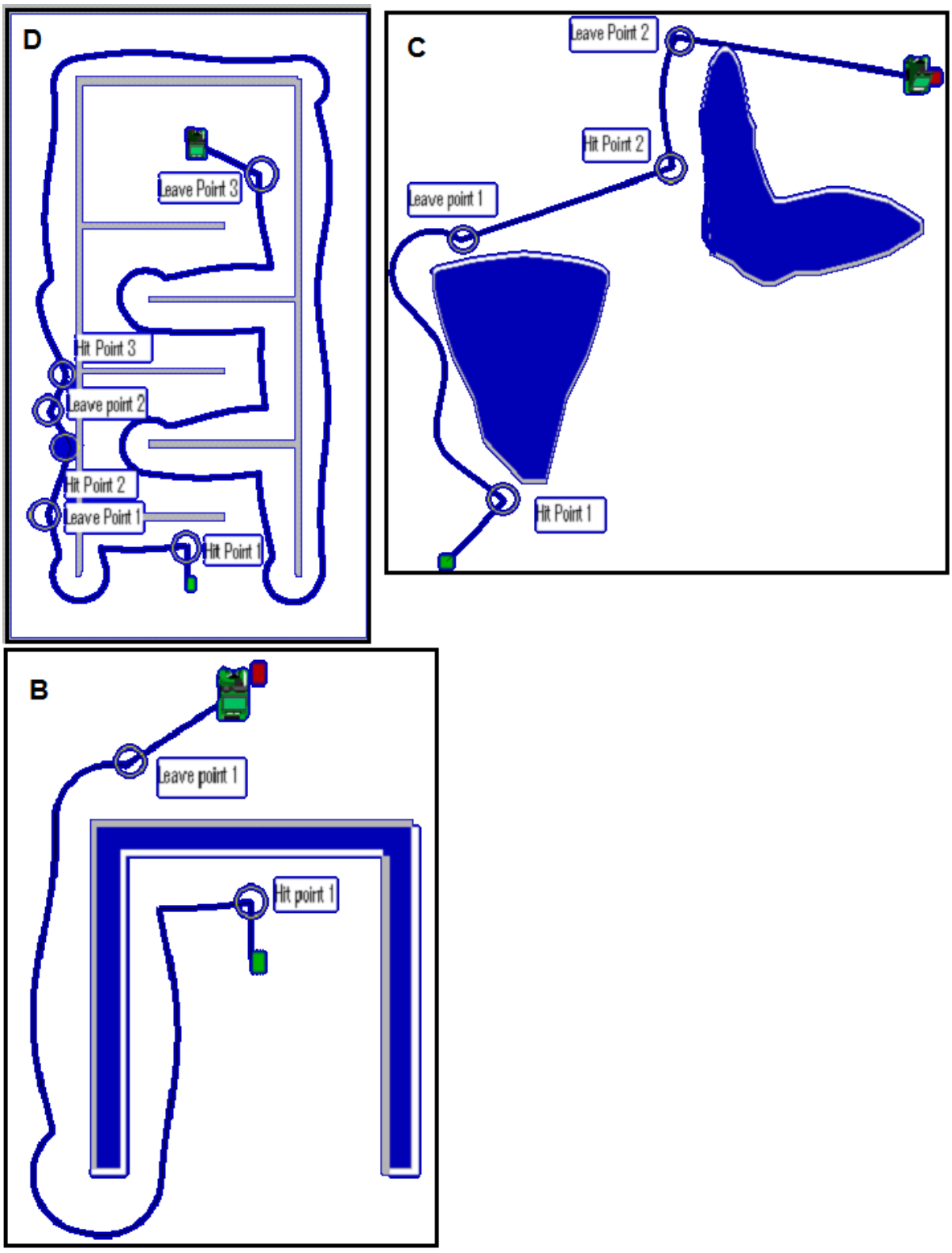


Figura 33.1: Respuesta de simulación del algoritmo Alg2 en BCS D, C y B [11].

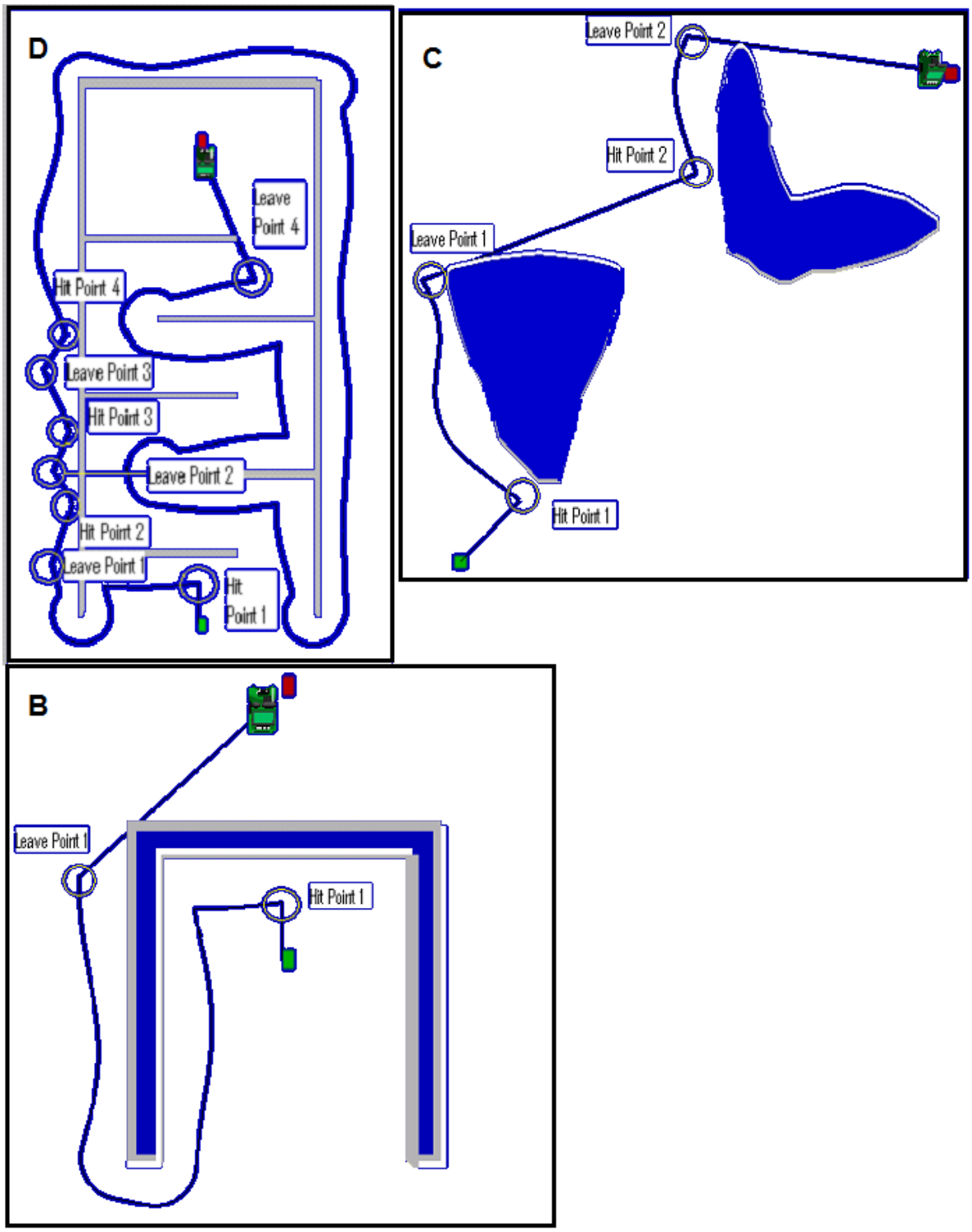


Figura 34.1: Respuesta de simulación del algoritmo DistBug en BCS D, C y B [11].

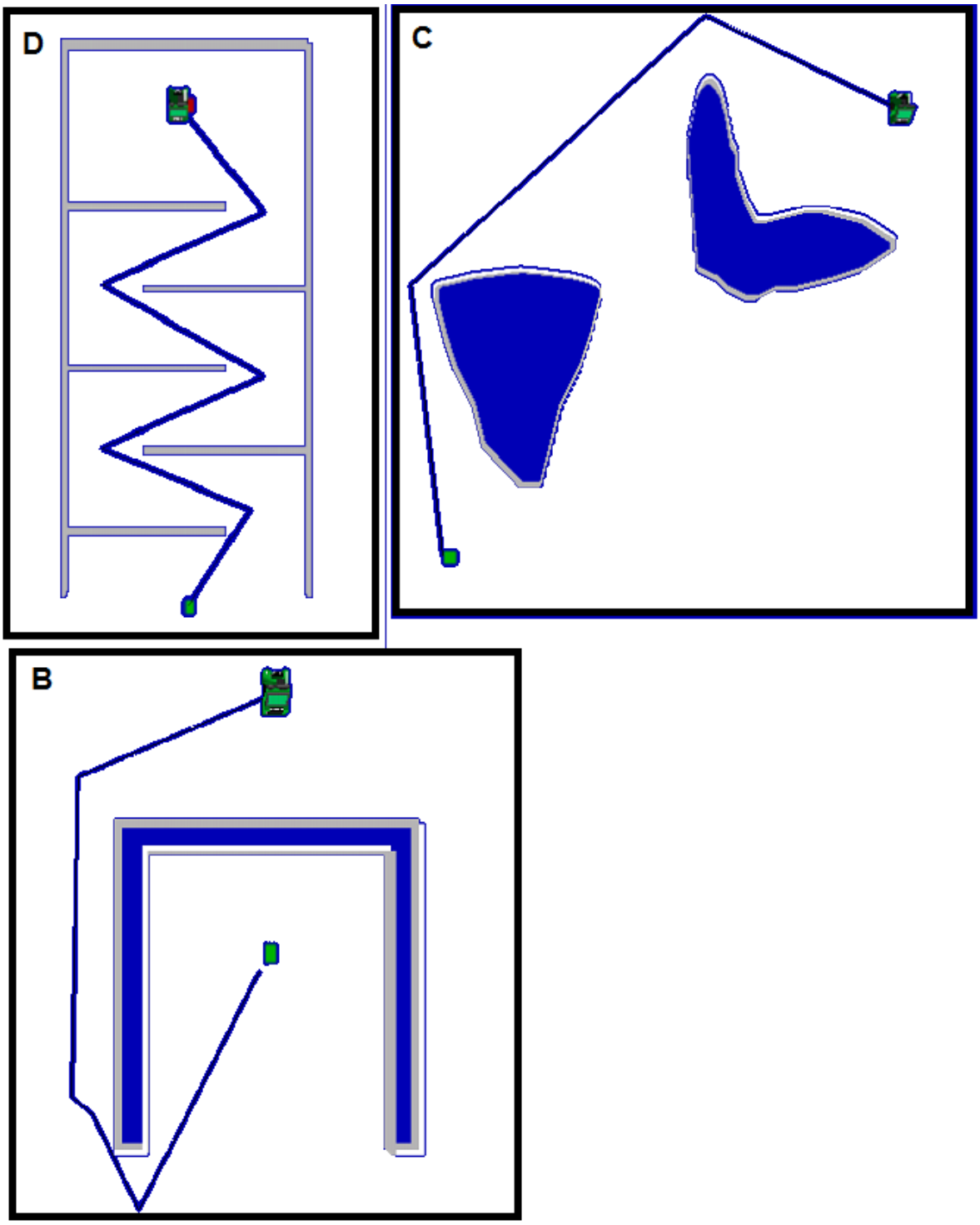


Figura 35.1: Respuesta de simulación del algoritmo TangentBug en BCS D, C y B [11].

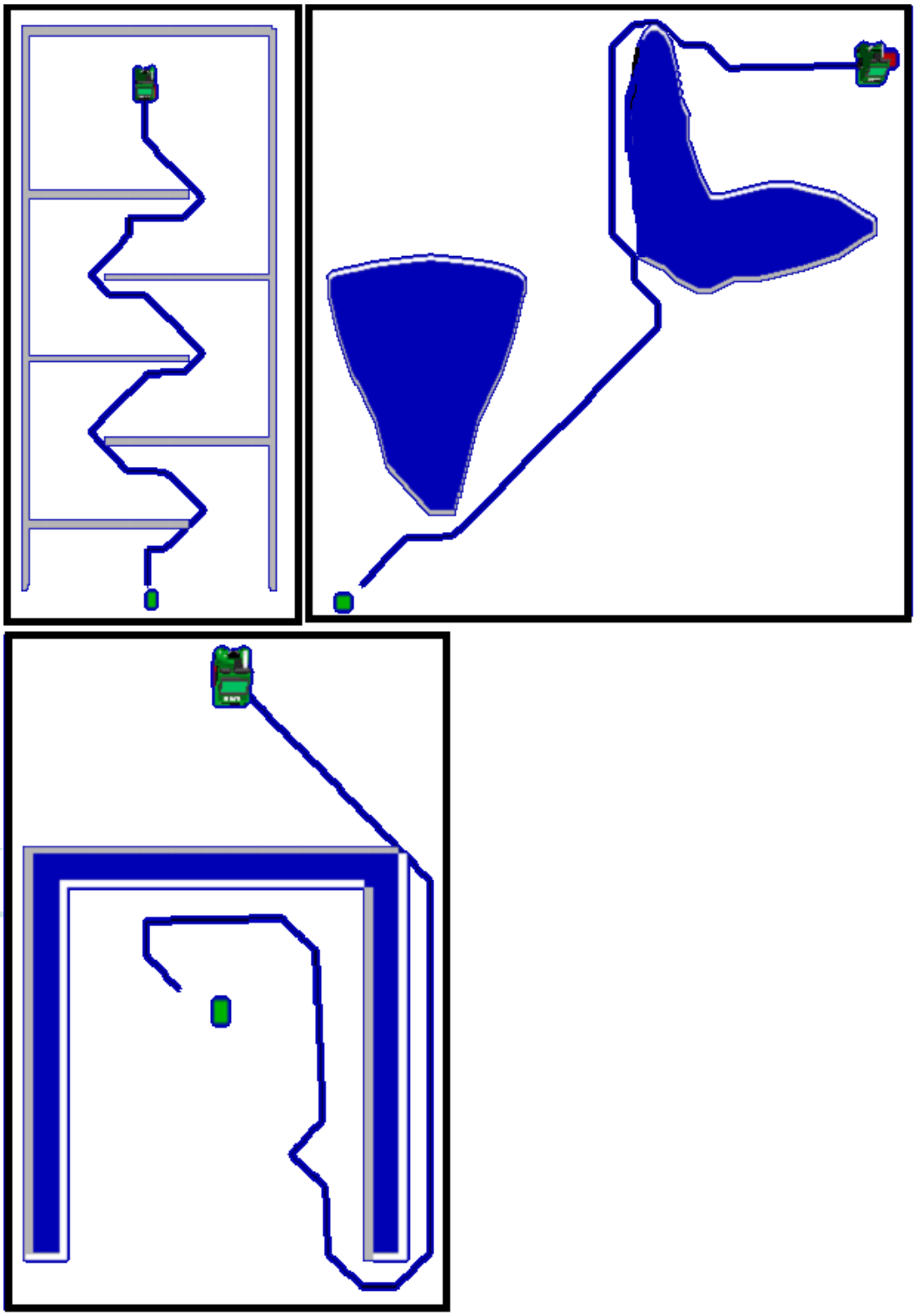


Figura 36.1: Respuesta de simulación del algoritmo D* en BCS D, C y B [11].

La figura 30.1 muestra la respuesta que tuvo el algoritmo Bug1 en los BCS D, C y B. En la parte izquierda de D se observa que el robot se acerca a una pared, marca el punto H_1 e inicia la circunnavegación a la estructura manteniéndose a la izquierda. En la parte derecha de D, una vez regresa cerca de H_1 el robot ahora se mantiene a la derecha de la pared e ingresa al espacio cerrado. Cuando pasa cerca al objetivo T marca un punto L_1 y sigue hasta llegar cerca a H_1. Regresa a L_1 para finalmente dirigirse a T.

En la sección del BCS B en la parte izquierda, se observa como se marca un punto H_1 y el robot empieza a circunnavegar el obstáculo cerca a S manteniéndolo a la derecha. Marca un punto L_1 cuando se encuentra más cerca a T y regresa a H_1, en donde regresa a L_1 nuevamente para continuar en dirección de T y encontrarse con un segundo obstáculo. Para la figura de la parte izquierda, cambia a modo move-to-goal y se marca un punto H_2 en cuanto encuentra el segundo obstáculo, lo mantiene a la derecha y cambia al modo boundary-following. El robot circunnavega el segundo obstáculo y marca un punto L_2 cuando se encuentra más cerca a T, sigue su circunnavegación hacia H_2 y regresa al punto L_2 de donde parte finalmente a T.

La simulación del BCS B presenta el mismo comportamiento que en la figura 29.1. Se marca un punto H_1, se circunnavega todo el obstáculo manteniéndolo a la derecha, marca un punto L_1 cuando se encuentra más cerca de T, regresa a H_1 y se devuelve a H_1 para dirigirse a T.

Claramente se comprende que el algoritmo genera una ruta de navegación larga porque el robot circunnavega todos los obstáculos para estar seguro que puede dirigirse hacia T.

La figura 31.1 muestra la simulación del algoritmo Bug2. En el BCS D para se tiene una línea M desde S a T, y cada vez que encuentra un obstáculo circunnavega como el Bug1 pero marcando los puntos H y L sobre M. Marca un punto L cuando está más cerca de T y regresa a un punto H previamente visitado. una vez se tenga un punto L_i más cercano a T, sobre la línea M se dirige a T. Este es el inconveniente que presenta este algoritmo, que puede hacer que el robot pase sobre una ruta varias veces.

Una gran mejora se observa cuando el Bug2 se encuentra en un BCS abierto como lo es el C. No existe una circunnavegación completa del obstáculo porque la línea M previene que el robot explore lo que está a la derecha de esta. Para este BCS C el robot no necesita regresar a un punto H_k previamente visitado porque no existe una pared que lo obligue a regresar, sino que puede continuar siguiendo la frontera del obstáculo en sentido horario. Lo que se observa en C es que los puntos H_1 y H_1 podrían hacer que el robot pase dos veces por la misma ruta, tal como sucedió para el BCS D.

Para el BCS B la mejora es muy positiva. El robot no circunnavega el obstáculo que genera el mínimo local gracias a M. Aunque depende del modo seguidor de frontera.

La respuesta del algoritmo Alg1 se observa en la figura 32.1. Para el BCS D el algoritmo Alg1 mejora al Bug2 porque recuerda los puntos H_k previos. Cuando se encuentra el robot en el punto H_3 no regresa al punto H_1 sino que lo hace al H_2, lo cual es una gran mejora. Para los BCS C y B al Alg1 presenta el mismo comportamiento que el Bug2 de la figura 31.1.

La figura 33.1 muestra la respuesta de la simulación del algoritmo Alg2. Para el BCS D el algoritmo compara una variable Q (distancia entre S y T) con la posición actual del robot. Un punto *y* es establecido si la distancia entre "*y*" y T es menor a la distancia entre Q y T. El robot marcó puntos H en su acercamiento a la pared, y en los puntos L se dirigió hacia T. Es decir, que encuentra un obstáculo y se aleja, lo intenta así por tres ocasiones y regresa a H_1. Debe circunnavegar porque no puede girar 180 grados. Cuando se acerca a H_1 el robot ahora mantiene el obstáculo a la derecha y procede hacia T con un modo seguidor de pared, donde finalmente marca a L_1, que es la condición para que el robot parta hacia T. Aumenta la eficiencia con respecto al Bug 1, pero contra el Bug2 no mejora la longitud de ruta.

Para el BCS C el Alg2 claramente supera a sus antecesores porque del punto L_1 a H_2 hay un segmento rectilíneo y no un seguimiento de frontera del obstáculo cercano a S. De igual manera se presenta este segmento rectilíneo desde el punto L_2 a T.

En el BCS B también existe una mejora con respecto a sus antecesores porque no necesita estar sobre una línea M para dirigirse a T, sino que lo hace casi una vez supera el mínimo local.

La figura 34.1 muestra el resultado de la simulación del algoritmo DistBug. para el BCS D genera un punto H_4 lo que es un punto L_3 adicional con respecto al Alg2. El robot cambia a modo seguidor de frontera hasta encontrarse cerca de H_1. mantiene el obstáculo ahora a la derecha y se dirige a T de igual manera que el Alg2, pero con la diferencia que una vez el robot tenga a la vista a T no marca un punto L sino que se dirige directamente a T.

El comportamiento de dirigirse directamente a T por parte del algoritmo DistBug se observa en los BCS C y B, donde el robot no debe recorrer parcialmente un segmento de un obstáculo para generar un punto L, sino que inmediatamente al no tener un obstáculo entre él y T se dirige al objetivo.

El algoritmo TangentBug muestra su respuesta a la simulación en la figura 35.1. Es el mejor algoritmo en responder a las condiciones de los tres BCS D, C y B. El robot se dirige directamente a los puntos donde existe discontinuidad de los segmentos de los obstáculos en trayectoria recta para los BCS D y B. Para el BCS C, aunque sobrepasa completamente a sus antecesores, se dirige a un punto n cercano al objetivo pero no al más óptimo. En la sección 7.6 de las conclusiones se aclara este concepto de punto eficiente para el TangentBug.

Para la figura 36.1 se muestra al D* con su respuesta a los tres BCS. Tiende a comportarse como el TangentBug en el BCD D pero se observa la influencia de la división del BCD en celdas. Para el C el DistBug si opera eficientemente ante el obstáculo cerca a S, pero el que está cerca a T no es eficientemente operado porque el robot se dirige al punto más distante entre el obstáculo y T. Para el B este algoritmo opera peor que el DistBug y el Alg2. Una razón más para considerar que la discretización del BCS debe ser un factor importante al emplear este algoritmo.

Se considera de mucha importancia cuando el robot dejará un obstáculo para dirigirse al objetivo (condición leaving). Por ejemplo, mientras el Bug1 puede ofrecer mejor rendimiento en ciertos espacios cerrados debido a que reúne todos los datos antes de decidir el punto de partida. Al contrario, el Bug2 es menos conservador y partirá hacia el objetivo una vez esté sobre la línea M. Esto hace que el Bug2 se desempeñe mejor en espacios abiertos [11].

La tabla 8 contiene resultados de las simulaciones por algoritmo, realizadas por el autor*. Los puntajes son el valor promedio de la posición obtenida en cada prueba (al ser siete algoritmos, habrán siete posiciones), y a menor valor se tendrá un mejor rendimiento. Las celdas en color verde representan el mejor algoritmo (vertical) de esa categoría (horizontal) a evaluar. Las celdas en color naranja representan la segunda mejor puntuación, las celdas en rojo la peor puntuación evaluada.

* En estas simulaciones consideró el atributo de rotación como un directo implicado en la longitud de una ruta de aproximación [11].

Tabla 8: Clasificación de los algoritmos según valores de atributos [11].

Atributos	Bug 1	Bug 2	Alg 1	Alg 2	DistBug	TangentBug	D*
Longitud de ruta	6.67	6.33	5	3.67	2.67	1.67	2
Rotación	6	3.67	4	6	4.67	2.33	1.33
Rotación inherente	6.67	6.63	4	4.66	2	2	2.67
Tiempo total de computación	4	3	5.33	5	4.33	3.33	3
Computación por metro	2	2.33	5	5.67	5.33	4.67	3
Llamado de funciones matemáticas	4.33	3.67	5.33	4.67	2	2.67	N/A
Error PSD	6	4	4	2	4.33	4	3.67
Error de desplazamiento lineal	6	4.83	4.83	3.33	4.33	3.33	1.33
Error de desplazamiento rotacional	6	5	2.83	4	5	2.83	2.33
Sencillez	2	1	4	5	3	6	7
Requerimiento en memoria	2	2	4	5	2	6	7

En la tabla 8 se observa la dominante fuerza de varios algoritmos. Para una ruta corta TangentBug y D* son los mejores. Considerando una combinación de categorías, para la categoría de tiempo de computación Bug1 y Bug2 pueden ser una buena opción si además se observa que en la sencillez del algoritmo son los mejores. Siendo el algoritmo D* totalmente diferente en cuanto al funcionamiento de los algoritmos de la familia Bug, notoriamente se ha visto bien favorecido en varias categorías evaluadas, pero en cuanto a su requerimiento en memoria puede ser una opción que influye en el presupuesto de un proyecto e incluso su rendimiento se vería afectado si no se tiene esta exigencia en hardware. El autor realizó 10 simulaciones de cada algoritmo, determinando la posición de rendimiento obteniendo el promedio de la posición en tres BCS [11].

Por lo tanto, el rendimiento de los algoritmos de navegación depende también de donde va a operar, y bajo qué condiciones hará la tarea. El mismo BCS junto con las características

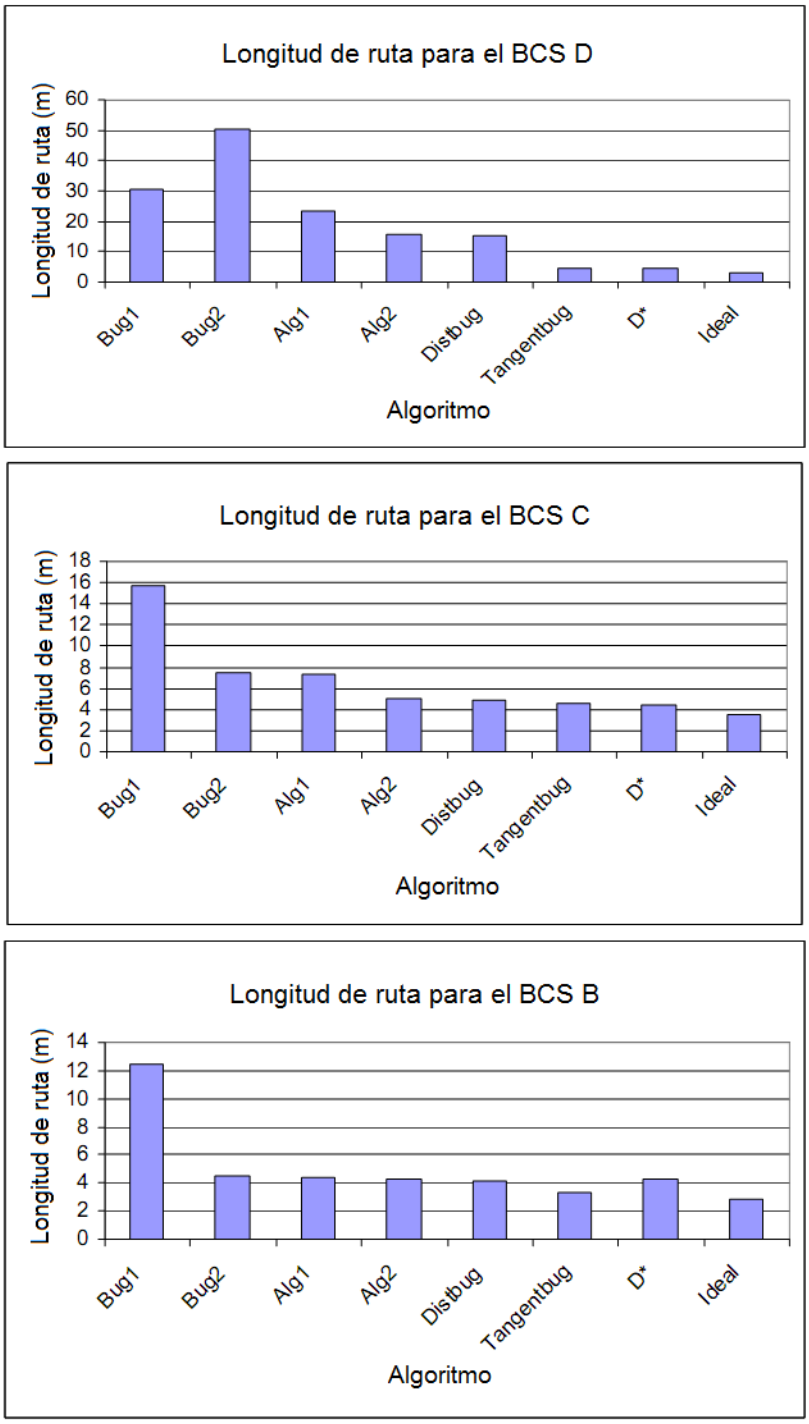
que éste tiene, serán los causantes de las modificaciones del algoritmo para obtener mejoras en la navegación.

Las gráficas incluidas en esta sección son complemento de la tabla 8 porque se observan los valores obtenidos de cada algoritmo en cada BCS. Con estos valores el autor asignó la posición en cuanto a la eficiencia de cada atributo en cada BCS simulado.

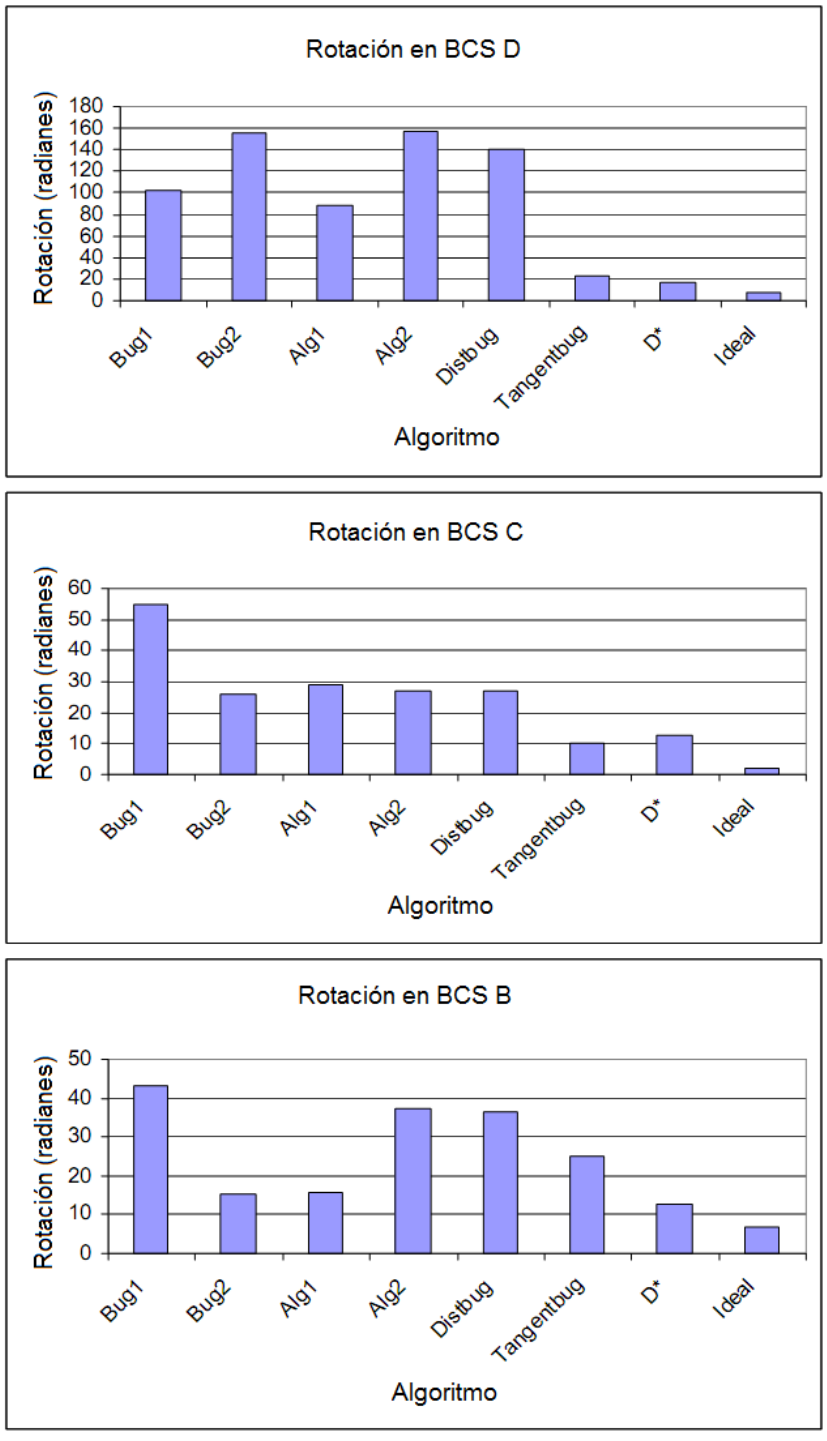
Para la gráfica 3 el algoritmo D* aunque es competitivo, mostró una excepción a la tendencia de la rotación, y es debido a que el BCS se segmenta en celdas. El robot solo puede moverse en giros discretos de 45 grados. El TangentBug puede moverse sin rotación en presencia de vértices o protrusiones, mientras que el D* debido a la discretización en celdas del espacio de don configuración, en cada vértice puede existir una segmentación navegue entre celdas vecinas, rotando frecuentemente.

En la gráfica 3 se observa que para espacios abiertos, el algoritmo TangentBug y D* por su comportamiento proactivo, reflejado en su capacidad de respuesta incluso ante obstáculos dinámicos. Las desventaja para el TangentBug es su falla al no detectar discontinuidades cerca de paredes o vértices iguales o menores a 90 grados. La falla para el D* está en que la segmentación o discretización, es decir la generación de celdas, no se realice en todo el BCS.

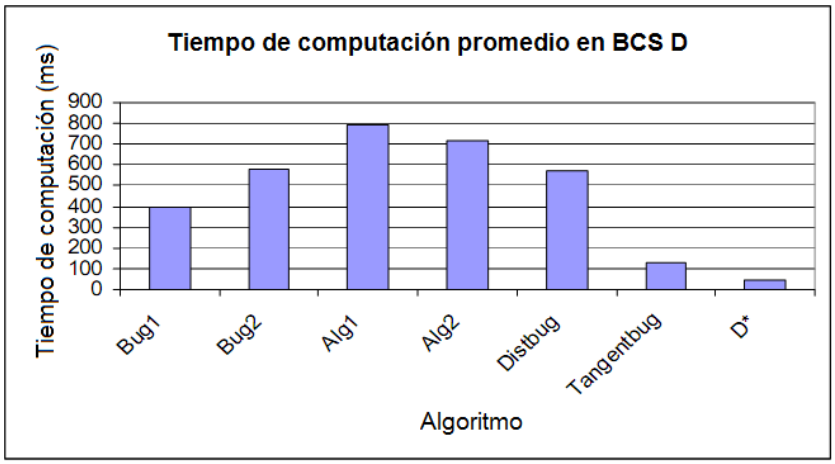
Gráfica 3: Longitud de ruta de los algoritmos en D, C y B [11].

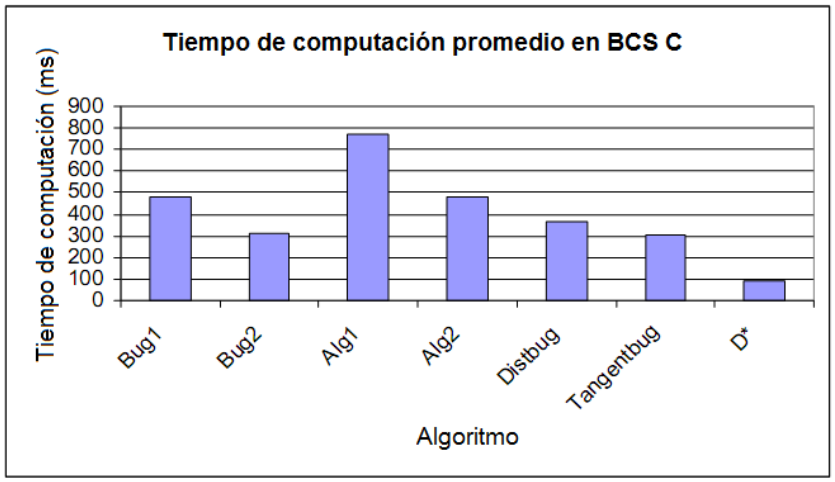


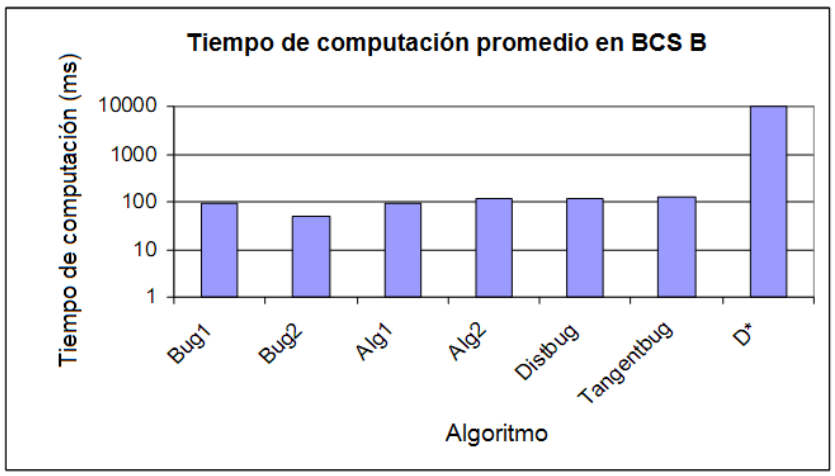
Grafica 4: Rotación de los algoritmos en D, C y B [11].



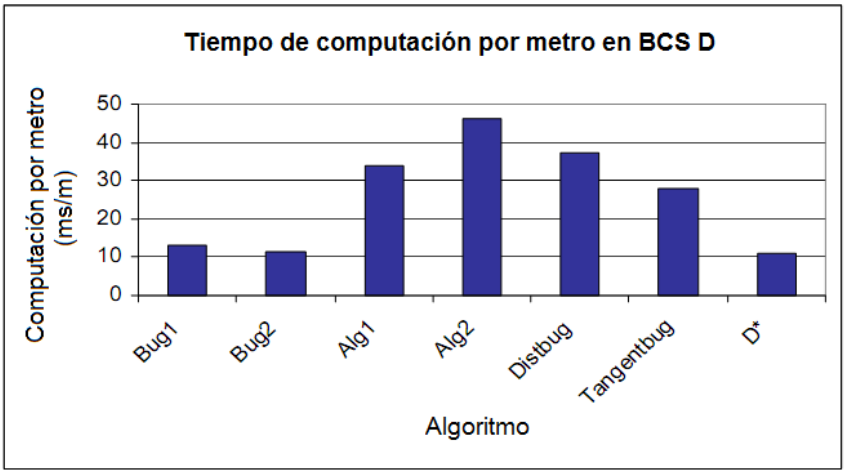
Grafica 5: Tiempo de computación de los algoritmos en D, C y B [11].

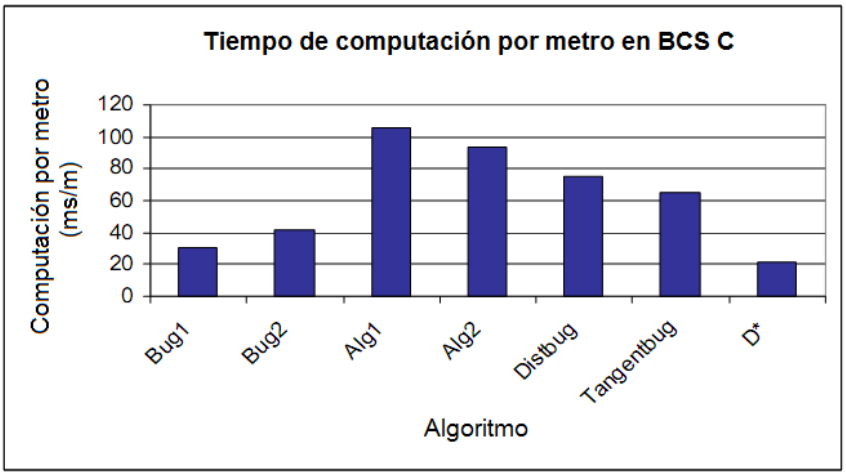


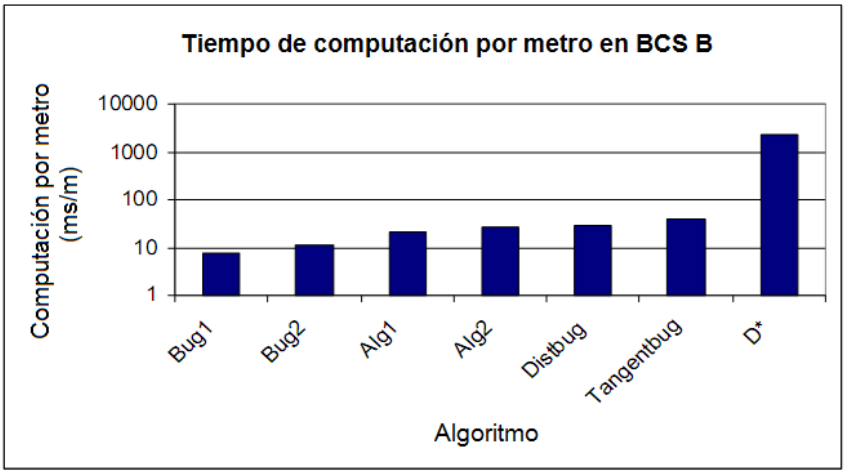




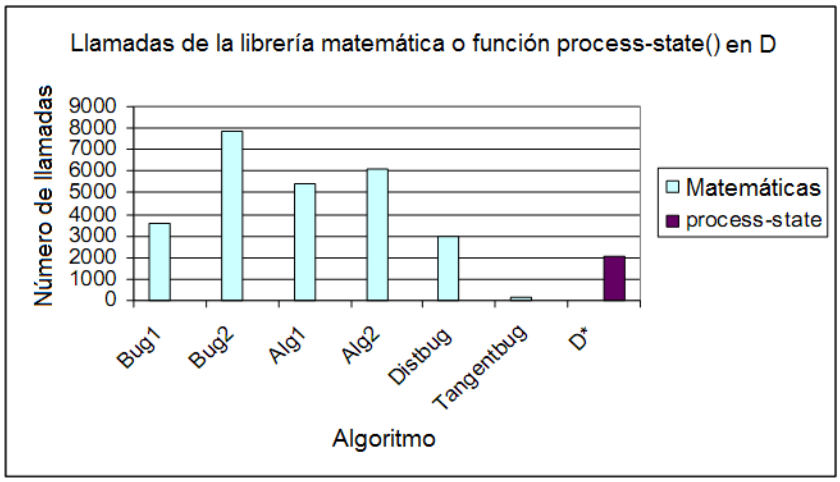
Grafica 6: Tiempo de computación por metro de los algoritmos en D, C y B [11].

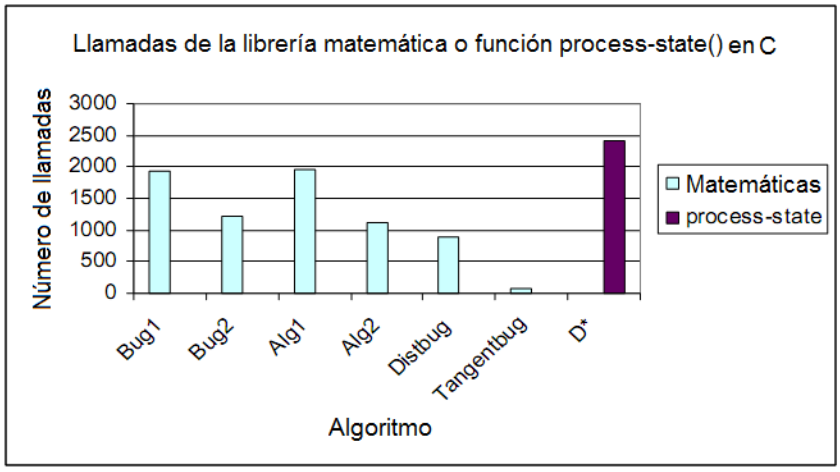


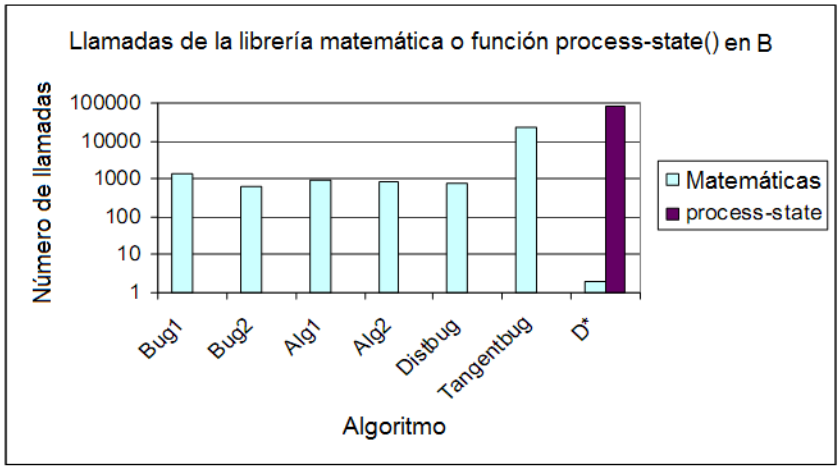




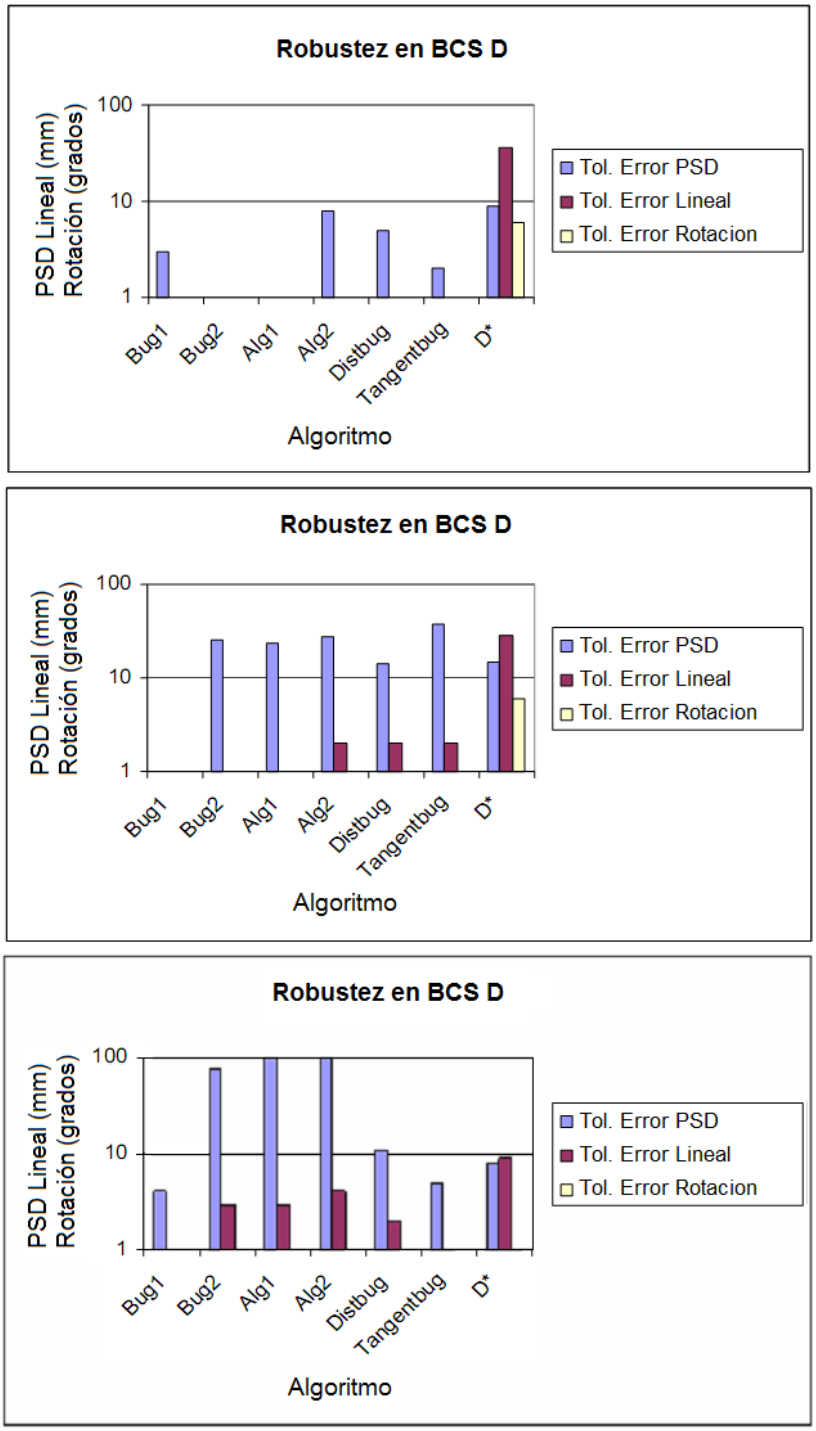
Grafica 7: Llamada de funciones de los algoritmos en D, C y B [11].



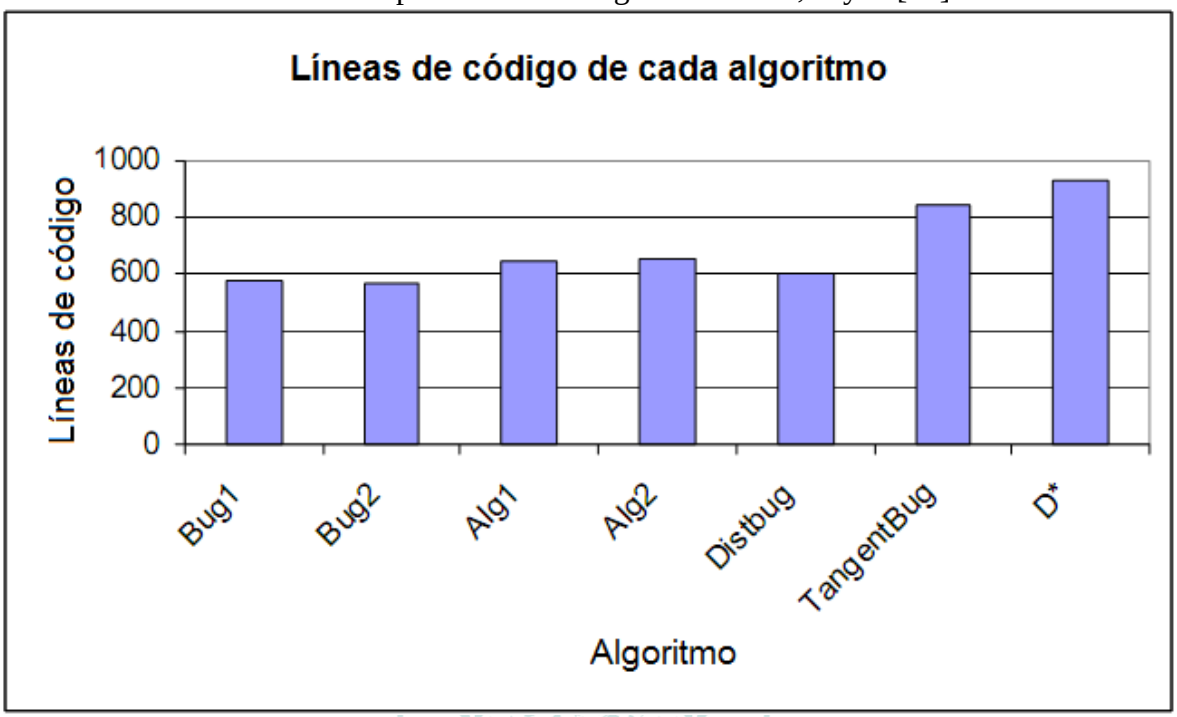




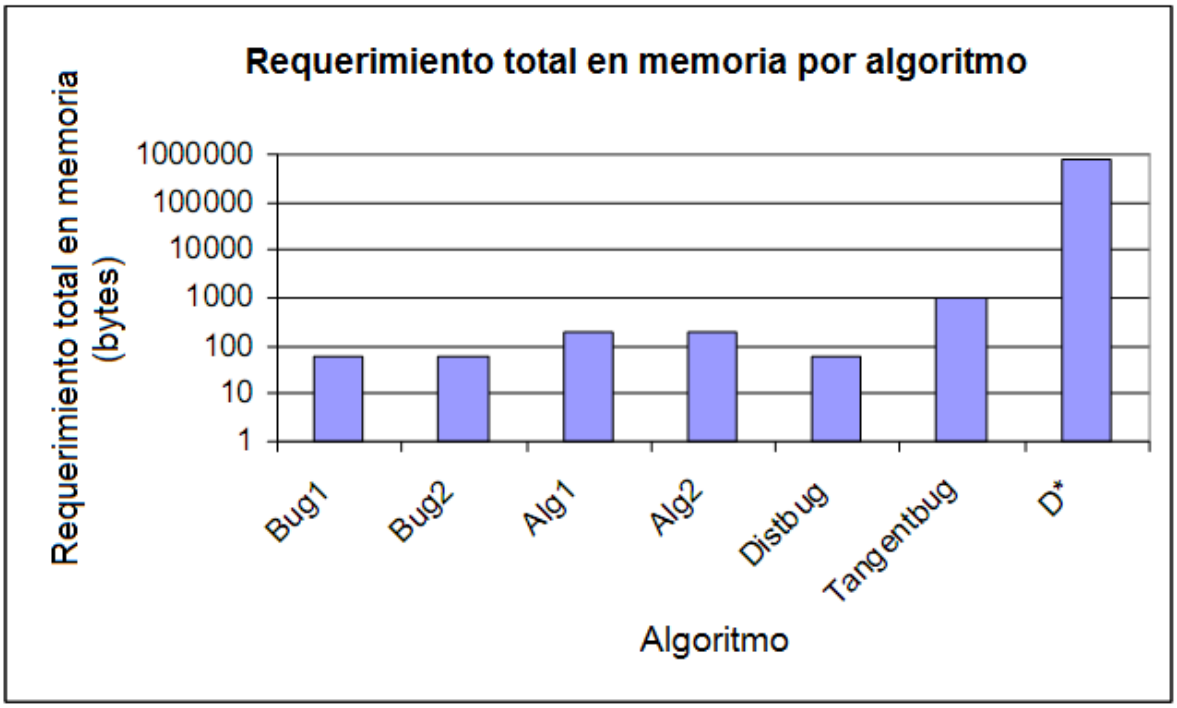
Grafica 8: Robustez de los algoritmos en D, C y B [11].



Grafica 9: Simplicidad de los algoritmos en D, C y B [11].



Gráfica 10: Requerimiento en memoria de los algoritmos en D, C y B [11].



La gráfica 4 de rotación muestra una relación con la longitud de ruta. Entre mayor sea la longitud de ruta de un algoritmo, mayor será también la rotación realizada [11].

En el Alg2 y DistBug aumentó la rotación posiblemente debido a la revisión del espacio libre F, que se ejecuta durante el modo seguidor de frontera. Se realizó la revisión de F cada 40mm en un rango de rotación de 0 a 45 grados. La rotación podría reducirse de implementarse un servomotor o un mejor complemento del PSD. La rotación no tuvo gran efecto en el D* ya que la discretización de los BCS por celdas reduce el movimiento del robot a 45 grados [11].

El tiempo de computación es una medida objetiva en el uso de la unidad central de procesamiento CPU en la navegación. El tiempo de desplazamiento del robot no es tan importante por depender de la velocidad de éste y mientras se desplaza la navegación puede reducir su uso de CPU [11]. De la gráfica 5 se puede concluir que un algoritmo menos sencillo puede tener a duras penas el mismo o menor tiempo de computación [11]. Posiblemente debido a que a menor desplazamiento se pueden tener rutas cortas que se acercan a lo ideal. El D* mostró un pésimo comportamiento en el BCS B debido a su vulnerabilidad a los mínimos locales.

Alg1 y Alg2 en los BCS D y C obtuvieron mal rendimiento debido a que revisan los puntos de la lista H y L visitados previamente. Les sigue el DistBug que es similar al Alg2. Sin embargo, para el BCS B los algoritmos Bug1 y Bug2 son mejores que el Alg1 y Alg2. El D* para B requiere de un tiempo excesivo para volver a computar la ruta óptima, algo que viene de su antecesor. El TangentBug también requiere de bastante tiempo para B para procesar las zonas visibles para determinar $d_{alcanzada}(T)$.

Para la gráfica 6 D* requiere el menor tiempo de computación por metro incluso mejor que Bug1 y Bug2. Probablemente se deba a la segmentación del BCS que permite al algoritmo no depender de las librerías matemáticas, además que no hay operaciones intensivas de multiplicación o división con punto flotante [11].

El TangentBug resultó ser competitivo porque aunque realizar la gráfica tangencial local LTG, emplea poco tiempo computacional mientras se desplaza. También llama el menor número de funciones matemáticas para D y C en la gráfica 7, lo que puede explicar su computación por metro. Este algoritmo posee la ventaja del procesamiento paralelo [11] por recolectar todos los datos en un punto. Otros algoritmos requieren que el robot se mueva para poder efectuar medidas.

El algoritmo D* no puede emplear paralelismo debido a que secuencia de puntos base óptimos debe ser computada secuencialmente desde el punto de partida. Esto lo hace en un algoritmo de procesamiento secuencial.

El algoritmo D* consiste principalmente de dos funciones [7]:

- process-state: empleada para computar el costo óptimo para llegar a T.
- modify-cost: usada para cambiar la función de costo direccional e ingresar a los estados afectados en la lista OPEN.

Ningún algoritmo fue diseñado por el autor para las simulaciones, aunque es esencial para cualquier aplicación práctica de un robot. Sin embargo, los resultados de la gráfica 8 muestran que algoritmos poseen una robustez inherente. Robustez se comprende como la capacidad del algoritmo para tolerar los errores PSD, lineal y rotacional. Debe ser tan alta como se pueda.

Los algoritmos no son robustos al ruido aunque se comportaron bien ante el error PSD que a los relacionados con los de desplazamiento. Contra el ruido de desplazamiento lineal varios se comportaron ineficientemente, excepto por el D*. La robustez depende también del BCS por lo observado en D. Para C y B se presentó más tolerancia a los errores del direccionamiento. Por lo tanto, los algoritmos presentaron mayor robustez en BCS abiertos que cerrados, probablemente por la necesidad de mayor precisión al navegar en espacios cerrados. En un espacio abierto hay mucho más espacio como para que un error de precisión ocurra.

Los algoritmos que recordaron los puntos previos tuvieron un mal desempeño. Para el Bug1 fue necesario saber cuando el robot circunnavegó el objetivo completamente. TangentBug presentó falsa localización debido a su vulnerabilidad de dirigirse a segmentos discontinuos sin conocer realmente que hay más adelante de su decisión.

El D* presentó mejor robustez en todos los BCS, posiblemente debido a que la discretización en celdas del espacio de configuración hace que el robot se desplace en pequeños pasos, que incluso limitan los grados de rotación que el robot posee, haciéndolo también robusto a errores rotacionales.

La sencillez del algoritmo es algo deseable aunque es una parte subjetiva, dependiendo de quien lo implemente. La cantidad de líneas de código del Bug1 y Bug2 los posicionan como algoritmos sencillos, mientras que el TangentBug y el D* tienden a ser

completamente un reto para quienes no posean buen conocimiento en la implementación de algoritmos de navegación.

Los requerimientos en memoria deben mantenerse tan bajos como sea posible. Se observa en la gráfica 10 que los algoritmos sencillos requieren menos memoria. El TangentBug requiere memoria para almacenar a r(ɵ) y D* requiere almacenar una grilla llena de celdas.

6. NAVEGACIÓN SEGURA ENTRE OBSTÁCULOS FIJOS Y DETERMINACIÓN DE UNA RUTA CORTA DE APROXIMACIÓN AL OBJETIVO

La eficiencia en la navegación de un robot móvil en un BCS bidimensional depende de todos los elementos involucrados en esta acción. Como se observa en la sección 5, no todos los algoritmos responden adecuadamente al BCS, y ante obstáculos los algoritmos presentan sus diferencias de operación.

Para algunos autores la mejora de los algoritmos solo se logra al aplicarlos a diversas condiciones, y mejorar la manipulación de la información obtenida por los sensores. Es el caso para esta sección el mostrar brevemente cómo al representar un BCS de manera geométrica, un algoritmo debe actuar de acuerdo a teoremas y parámetros matemáticos, más no en rutinas intuitivas que no muestran una mayor exigencia en cuanto a un proceso de navegación.

Lo que se desea en esta sección es comprender cómo se realiza un sistema que ejecutará la tarea de navegación. Este sistema debe ser modelado de manera que considere parámetros propios del funcionamiento del robot para una navegación segura y eficiente.

6.1. DESCRIPCIÓN DEL SISTEMA

A continuación se describirá un robot móvil que navegará a una velocidad constante, y con una velocidad angular controlada. Se hace uso de la figura 31.1 para la descripción del sistema.

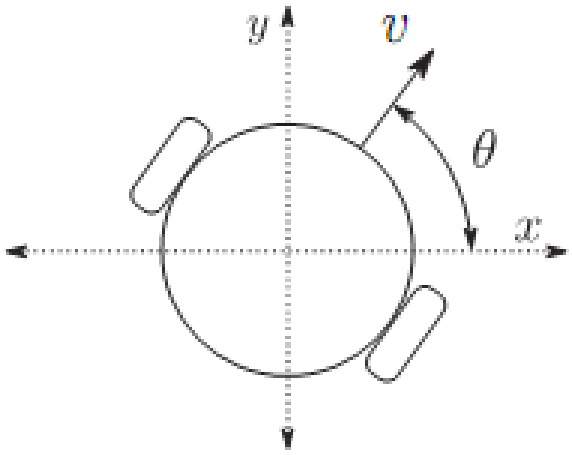


Figura 36.1: Modelo de un robot móvil [16].

En la figura 36.1 se observa un robot para el cual su acción de giro depende de la relación de dos ruedas. Para el robot se tiene un plano cartesiano que relaciona posición, velocidad y ángulo de orientación.

$$vx = v\cos\theta$$
$$vx = {dx}/{dt} = \dot{x} = v\cos\theta$$
$$vy = {dy}/{dt} = \dot{y} = v\,sen\theta$$

$$\dot{\theta} = \omega$$
$$\dot{\theta} = u \in [-\bar{u}, \bar{u}]$$

$$x(0) = x_0$$
$$y(0) = y_0$$
$$\theta(0) = \theta_0$$

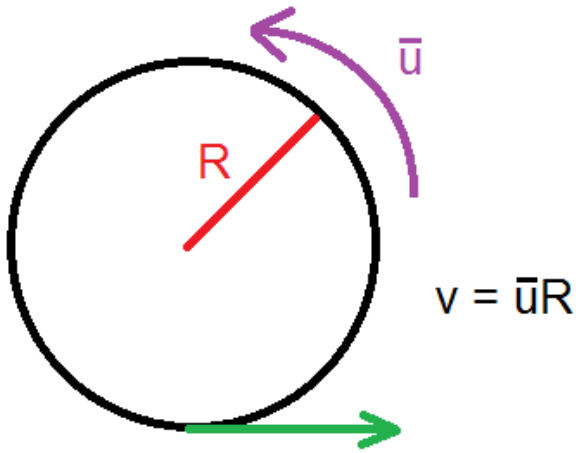


Figura 37.1: Definición de la máxima velocidad angular.

Se tiene a *[x y]* como el vector de las coordenadas cartesianas del robot y ɵ da la orientación del robot. *v* es la velocidad del robot y *u* es la velocidad angular con la que él gira. El radio mínimo de giro del robot se interpreta de la gráfica 37.1 y se establece como:

$$R_{min} = \frac{v}{\bar{u}}$$

El desplazamiento seguro y eficiente del robot depende de la capacidad que tiene para mantenerse sobre la ruta corta de navegación, controlando la velocidad angular con la cual se ubicará sobre dicha ruta en presencia de obstáculos. La figura 38.1 muestra una ruta curva P(s) y se realiza una parametrización. Un segmento curvo de P(s) puede ser reemplazado por una línea recta que une a P'(s1) y P'(s2), para hacer el segmento P'(s1,s2) mucho más corto [16].

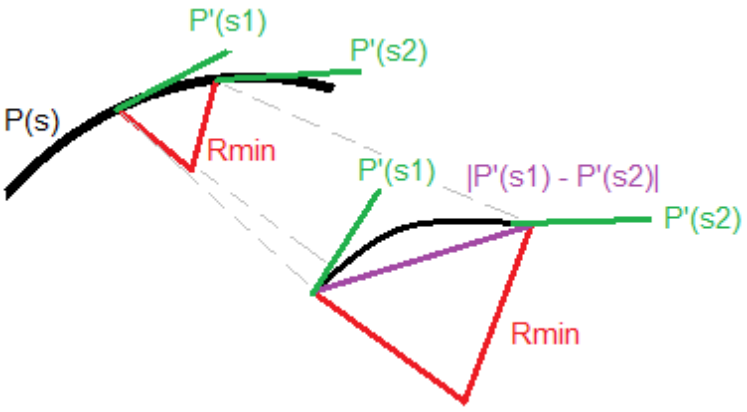


Figura 38.1: Parametrización de la curva P(s).

De la figura 38.1 se deduce la restricción denominada curvatura promedio, dada por la ecuación 2:

$$\|P'(s_1) - P'(s_2)\| \leq \frac{|s_1 - s_2|}{R_{min}}, s_1 \neq s_2$$

Ecuación 2. Ecuación de curvatura promedio [16].

6.2. DEFINICIÓN DE DISTANCIAS DE SEGURIDAD

Se considera un margen de seguridad d_0 mayor a cero. Este d_0 determina que tan cerca estará el robot de un obstáculo, es el radio de una circunferencia y puede ser representado como un conjunto D. La distancia entre una ruta P y un obstáculo se denomina $dist[P, D_k]$.

Para la figura 39.1, todo obstáculo está rodeado de múltiples círculos iniciales de radio d_0. Es decir, acotados por un conjunto denominado $\mathcal{N}[D_i, d_0]$, que es una sucesión de líneas rectas. Se considera una *ruta de alcance al objetivo con evitación de obstáculo* (TROA) a una la ruta P que alcanza al objetivo T evitando los obstáculos, sí $r(t_f) = \mathrm{T}$ (t_f es el tiempo final cuando se alcanza a T), y $r(t) \notin \mathcal{N}[D_i, d_0]$ [16].

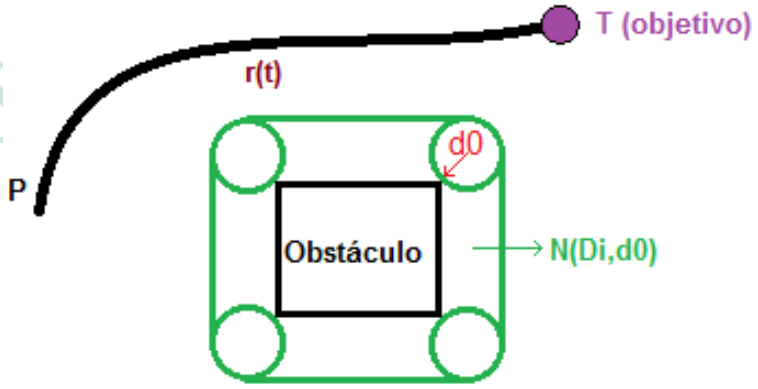


Figura 39.1: Identificación de las distancias de seguridad y conjuntos.

La curvatura para los bordes de un obstáculo en cualquier punto P para la vecindad $\mathcal{N}[D_i, d_0]$, no excede $\frac{1}{R_{min}}$.
Para cualquier *i*, se cumple $\|\mathrm{T} - r(0)\| \geq 8R_{min}$ y $dist[r(0); \mathcal{N}[D_i, d_0]] \geq 8R_{min}$ [16].

En la figura 40.1 se observa el caso en el que el robot no puede alcanzar al T debido a que los espacios entre los obstáculos que lo rodean son menores que el margen de seguridad d_0.

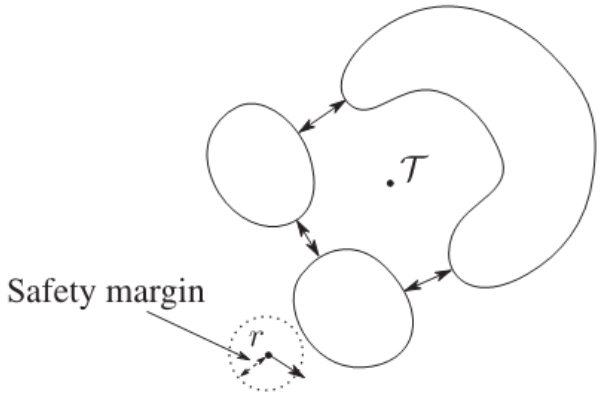


Figura 40.1: Representación para el caso que el objetivo T no puede ser alcanzado por el robot [16].

6.3. PLANEACIÓN DE LA RUTA MÁS CORTA DE ALCANCE AL OBJETIVO

Se describe como se puede determinar la ruta más corta con evitación de obstáculos. Para esto, se define una línea recta L que representa la línea tangente si esta línea satisface por lo menos una de las siguientes condiciones [16]:

1. L es tangente a dos límites $\partial Di(d_0)$ y $\partial Dj(d_0)$ simultáneamente, siendo $i \neq j$.
2. L es tangente a un límite $\partial Di(d_0)$ en dos puntos diferentes.
3. L es tangente a un límite $\partial Di(d_0)$ y a un círculo inicial simultáneamente.
4. L es tangente a un límite $\partial Di(d_0)$ y cruza el objetivo T.
5. L es tangente a un círculo inicial y cruza al objetivo T.

Se define un círculo inicial como aquel círculo de radio d_0 (margen de seguridad) ubicado en la posición inicial del robot r(0), con el que r(0) es tangente. Los puntos donde una línea tangente toca ya sea al conjunto de distancias $\partial Di(d_0)$ o un círculo inicial, son denominados puntos tangentes. Se observa lo anterior en la figura 41.1.

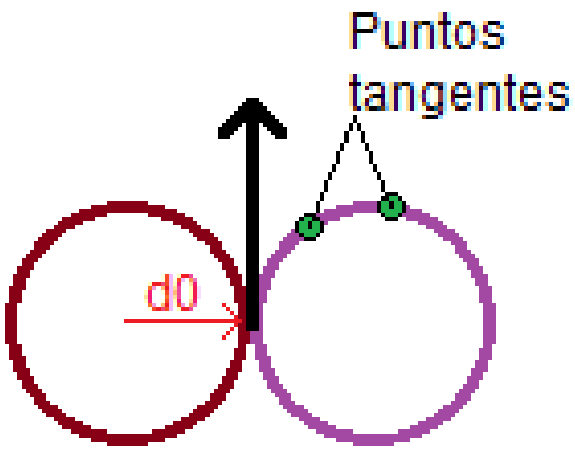


Figura 41.1: Dos círculos iniciales a cada extremo de la posición del robot y dos puntos tangentes.

También existen segmentos que surgen de aquellas condiciones para que una línea L exista. Esos segmentos son los siguientes:

- Segmento (OO): Si es del tipo 1 o 2.
- Segmento (CO): Si es del tipo 3.
- Segmento (OT): Si es del tipo 4.
- Segmento (CT): Si es del tipo 5.

Una ruta corta de alcance al objetivo consiste de $n \geq 2$ segmentos $s_1, s_2 \ldots s_n$, de tal manera que si:

- $n = 2$: s_1 es un segmento tipo (C) y s_2 tipo (CT).
- $n \geq 3$: s_1 es un segmento tipo (C), s_2 es tipo (CO) y s_n tipo (OT).
- $n > 3$ y $3 \leq k \leq (n-1)$: cualquier segmento s_k es tipo (OO) ó tipo (B).

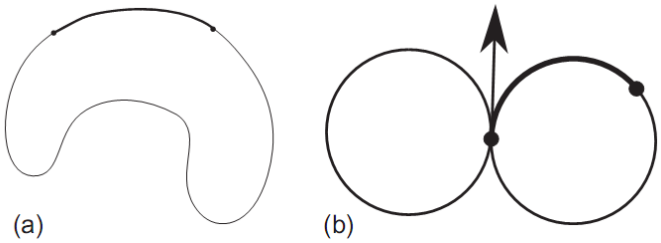


Figura 42.1: Segmentos tipo (B) y (C) [19].

El segmento tipo (B) de la figura 42.1 parte (a), es aquel arco entre dos puntos tangenciales en el límite del conjunto $\partial Di(d_0)$ sí la curvatura es no negativa. Un segmento tipo (C) como el que se observa en la figura 42.1 parte (b), es aquel arco de un círculo inicial entre la posición inicial del robot r(0) y un punto tangente. Para la existencia de un segmento corto de alcance al objetivo, el primer paso es saber que P no va a pasar dentro de alguno de los círculos iniciales. Para esto se considera ahora un círculo C de radio $6R_{min}$ centrado en r(0).

Se asume que el objetivo y los obstáculos están fuera de (C). Con lo anterior, se tiene a p_1 como un punto de (C) en el que la ruta P sale de (C) por última vez, y θ_1 es la trayectoria hacia p_1. Así P consiste en dos segmentos $(r(0), p_1)$ y (p_1, T), donde (p_1, T) está fuera de (C). Entre todas las rutas que conectan r(0) y p_1, que tienen direcciones θ_0 y θ_1 en estos puntos y satisfacen la condición de curvatura promedio de la ecuación 2, se toma una ruta P_1 de mínima longitud. Se considera que esta ruta exista y pertenezca al disco de radio $8R_{min}$ centrado en r(0).

P_1 no interseca algún $\partial Di(d_0)$. Siendo P_1 una longitud mínima, su longitud L_1 no es mayor que la longitud de L de $(r(0), p_1)$. En el caso que $L_1 < L$, la ruta compuesta de P_1 y (p_1, T) sería una ruta corta de alcance al objetivo. Para $L_1 = L$ y $(r(0), p_1)$, también es una ruta corta que satisface la ecuación 2 de curvatura promedio.

Se establece que P_2 está constituida por piezas de los límites de $\mathcal{N}[D_i, d_0]$ con curvaturas no negativas, partes de los círculos iniciales y segmentos lineales rectos tangencialmente conectando puntos de estas curvas [16]. Una figura que muestra la representación de este planteamiento es la figura 43.1.

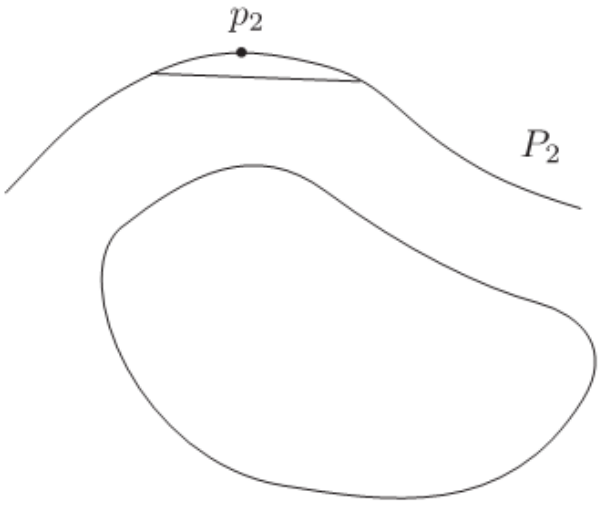


Figura 43.1: Reducción de una longitud de ruta por medio de un reemplazo de segmento curvo por uno recto [16].

Se establece que la ruta P_2 contiene un punto p_2 que se atribuye a ninguna de las curvas. p_2 está por fuera del conjunto $N[D_i, d_0]$ y fuera también de los círculos iniciales, tampoco es un punto interno de un segmento de línea recta de P_2. Por lo tanto, en una vecindad de p_2 un segmento lo suficientemente pequeño de P_2 puede ser reemplazado por un segmento de línea recta que no interseca ni el conjunto $N[D_i, d_0]$ como los círculos iniciales.

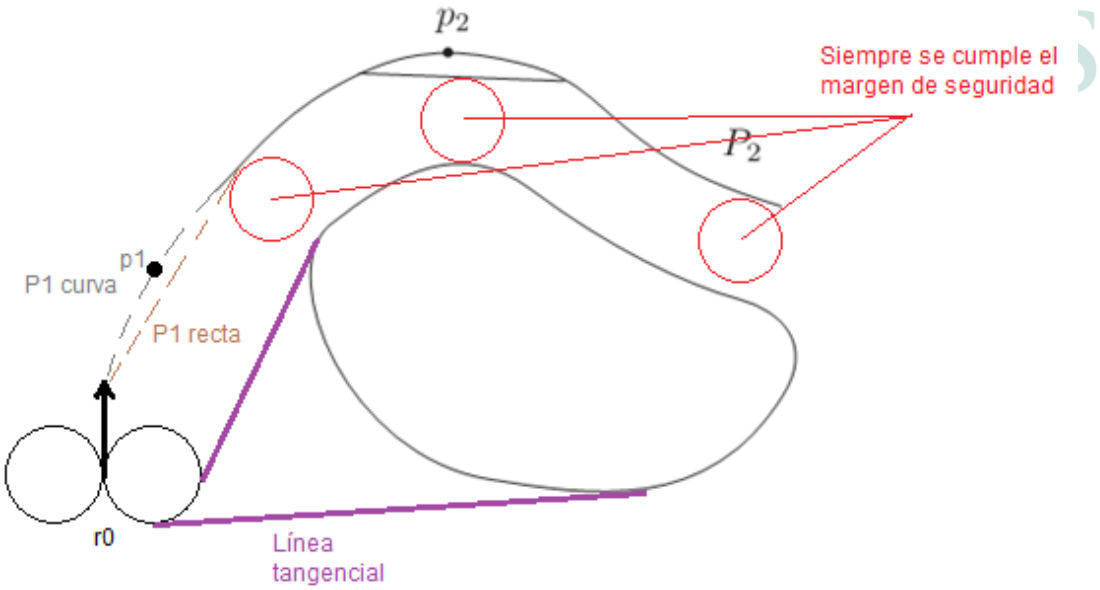


Figura 43.2: Modificación de la figura 42.1 añadiendo r0, margen de seguridad, líneas tangenciales de campo de visión, P1 recta y curva.

La figura 43.2 contiene varios elementos a manera de interpretación de cómo podría un robot en la posición r(0) con sus círculos iniciales, detectaría un obstáculo a distancia (suponiendo que su campo de visión sea muy amplio) y se generaría una gráfica tangencial. Para esta figura se tiene un segmento inicial como P_1 hasta el punto tangencial p_1, el cual se reemplazaría por un segmento recto. Se cumple que la parte P_1 es un arco de segmento de un círculo inicial menor o igual a $\frac{1}{R_{min}}$, porque a esto se debe incluir el margen de seguridad que conforma el grupo $\mathrm{N}[D_i, d_0]$ que acota al obstáculo, una circunferencia de radio d_0. Se observa que siempre se cumple con el margen de seguridad, incluso en el punto p_2 donde se reemplazó el segmento curvo con uno recto.

Con lo documentado en esta sección se reduce el problema de que el robot encuentre una ruta corta de alcance al objetivo, porque busca entre las posibles rutas en una gráfica geométrica especial, como lo es la gráfica tangencial de la sección 2.1.6, la más óptima y ejecutando una navegación segura. La figura 44.1 muestra un ejemplo de una gráfica tangencial para (a), en la cual se incluye al objetivo T como un punto. La parte (b) muestra cómo opera un robot r y su sensor de visión capaz de detectar obstáculos mientras estos se encuentren en su campo de visión.

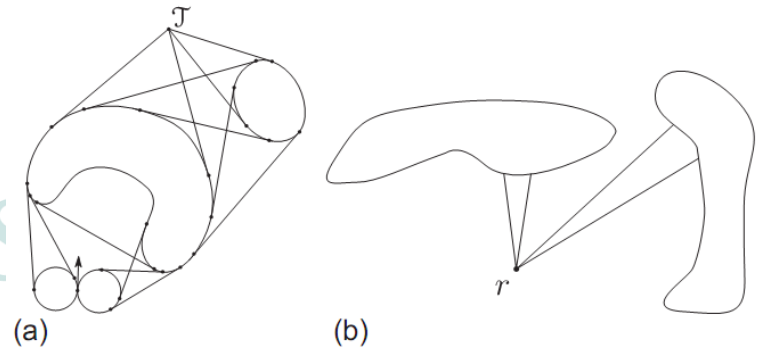


Figura 44.1: Gráfica tangencial con el punto objetivo T y campo de visión [16].

7. CONCLUSIONES

El rendimiento de los algoritmos de la familia Bug varía considerablemente dependiendo del BCS. Con los resultados de la gráfica 2 y la tabla 7 de la sección 5, el TangentBug produjo la ruta más corta en un BCS abierto. La capacidad de los sensores de rango le permiten al robot moverse hacia áreas en las que otros algoritmos acceden dependiendo del modo seguidor de frontera.

Los métodos de finalización permiten a los algoritmos de navegación determinar si para el robot es posible o no llegar al punto objetivo T y terminar, evitando que éste se ejecute indefinidamente al haber realizado todas los procedimientos necesarios para la aproximación. Para el caso del TangentBug debe tenerse en cuenta la clase de técnicas que se implementarán al él, pero siempre teniendo muy en cuenta el BCS a navegar. La generación de una gráfica tangencial global es mucho más conveniente computacionalmente que el mapeo. Con el mapeo se pueden obtener rutas de navegación muy eficientes en cuanto a su corta longitud de ruta, pero un BCS dinámico, es decir con muchos obstáculos moviéndose, puede empezar a causar los problemas típicos de un proceso dependiente de los recursos computacionales, como tiempos largos en procesamiento y uso excesivo de memoria.

Al estudiar los primeros algoritmos de la familia Bug y continuar con aquellos no pertenecientes, los resultados por fuera de la familia han sido importantes para la mejora de los Bug. Los autores del libro Safe Robot Navigation Among Moving and Steady Obstables expresan que la navegación segura y libre de colisiones en BCS con presencia de obstáculos tanto fijos como dinámicos es un principal problema en la robótica de navegación [16]. Cualquier algoritmo puede mejorarse por medio de la interpretación geométrica y parametrización de un BCS, junto con los elementos en él. Las rutas de navegación curvas representan más distancia recorrida por el robot, y pueden mejorarse al realizar una parametrización de curvas para cambiar segmentos curvos por rectos.

La estructura de un algoritmo de navegación debe conservarse para evitar problemas en su funcionamiento. Lo que permite una amplia cantidad de algoritmos es que puede modificarse su método de adquisición y manipulación de datos de los sensores, la técnica empleada para el acercamiento y alejamiento de obstáculos, la respuesta del robot para un BCS específico y muchas más. Dependiendo la complejidad de la navegación y los recursos físicos con los que se cuente, la modificación de los algoritmos permite una gran personalización para obtener el mejor resultado. Las adaptaciones de los algoritmos también deben considerarse si se tendrá una tarea de navegación en un espacio cerrado como es el A y el D ó abierto como el B y el C. Cada modificación puede inferir en la capacidad de respuesta positiva y complejidad del algoritmo, y dependiendo del tipo de

algoritmo puede disminuir considerablemente su rendimiento por la manera en que se ejecutará.

Se concluye que cuando el algoritmo genera una longitud de ruta más larga, una mayor cantidad de rotación es realizada [11]. Algoritmos como Alg2 y DistBug se presentaron como atípicos, concluyendo que el incremento en su rotación fue posiblemente debido a su revisión del espacio libre F en su modo seguidor de frontera o límite. Esta revisión se realizó cada 40mm y la rotación por cada acción tenía un rango entre los 0 a 45 grados. Esta rotación podría disminuir si en una implementación real se hace uso de un servomotor que apunte a la dirección del objetivo T, o incluso más detectores sensibles a la posición PSD*. En la simulación, al emplear servos en lugar de más PSD, se observó que los algoritmos Alg2 y DistBug obtuvieron mejoras significativas en espacios abiertos [12].

7.1. SEGMENTACIÓN DE ALGORITMOS

Los algoritmos Bug no cuentan con un límite en la libertad de rotación, y tampoco poseen una discretización del BCS. Como se explicó en la sección 5, al existir mayor rotación por parte del robot también mayor será la longitud de ruta. Se concluye que sería muy interesante observar el rendimiento de los algoritmos Bug si se tiene una división por celdas del BCS como en el algoritmo D*. Claro está que esto podría aumentar lo robusto del algoritmo para rutas largas, y aumentar los requerimientos en memoria [11].

7.2. LOCALIZACIÓN

La tarea de "reconocimiento muerto", es decir, que el robot se dirija a zonas sin relevancia para la navegación, causa una localización errónea que produce errores de desplazamiento a los que el algoritmo no corregirá. Este comportamiento errático podría solucionarse con múltiples muestreos, y con esto la mejor solución sería una realimentación (en inglés feedback). Un posible método de realimentación es el reconocimiento del BCS por medio de imágenes provenientes de una cámara. De ser implementada la realimentación por imágenes de cámara, sería necesario escribir un módulo de procesamiento de imagen. Varios métodos existen para estas mejoras [17].

---

* Interpretación fundamentada en los comentarios realizados por el PhD James Ng de con los resultados obtenidos de las simulaciones [11].

También podrían implementarse ciertos puntos de acceso con posibilidad de interacción con el robot, y estos funcionarían como postes que por medio de tecnologías de radio podrían medir únicamente la distancia del robot con estos elementos. Estos postes podrían, por ejemplo, encontrarse en un nivel de un centro comercial referenciando un baño, ascensores, salidas de emergencia, etcétera. Con esta implementación se perdería la autonomía de navegación del robot pero un argumento que así como los humanos pueden emplear señales para navegar en entornos desconocidos, bien podría hacerlo un robot [11].

7.3. MEJORAS PARA EL ALGORITMO D*

El tamaño de las celdas de discretización del BCS también es importante para el impacto de recursos computacionales y tiempos de finalización. La estructura de la celda podría alterarse y someterse a pruebas, siendo un cambio principal el emplear celdas con forma de triángulo en lugar de un cuadrado. También debe considerarse el problema de que las celdas no cubran completamente en BCS, y una opción sería la ubicación de celdas dinámicas.

7.4. APRENDIZAJE DEL ROBOT

Otro aspecto a tener en cuenta es el aprendizaje propio del robot. Esto se lograría guardando cierta información sobre áreas libres que previamente el robot haya atravesado con anterioridad. Esto podría ser de mucha utilidad para los algoritmos Alg1 y Alg2 para recorridos con posibilidad de circunnavegaciones subsecuentes.

7.5. TOLERANCIA AL ERROR

Las simulaciones fueron realizadas sin toleración al error, y en algunas ocasiones el robot no podía realizar ciertas correcciones que en un humano podrían considerarse como improvisación. Cierta tolerancia al error podría generar una garantía sobre las decisiones que se toman en la navegación, y esto se observaría en el PSD (dispositivo sensible a la posición) para realizar un promedio total de error [11]. Un compás, un giroscopio, un GPS, al someter al robot al exterior, pueden ser empleados para disminuir el error de rotación.

7.6. MEJORAS PARA EL TANGENTBUG

Una mejora para el algoritmo TangentBug podría surgir al observar la figura 45.1. En simulaciones el robot decide tomar la Ruta A producto del Nodo 1 y no la Ruta B producida por el Nodo 2.

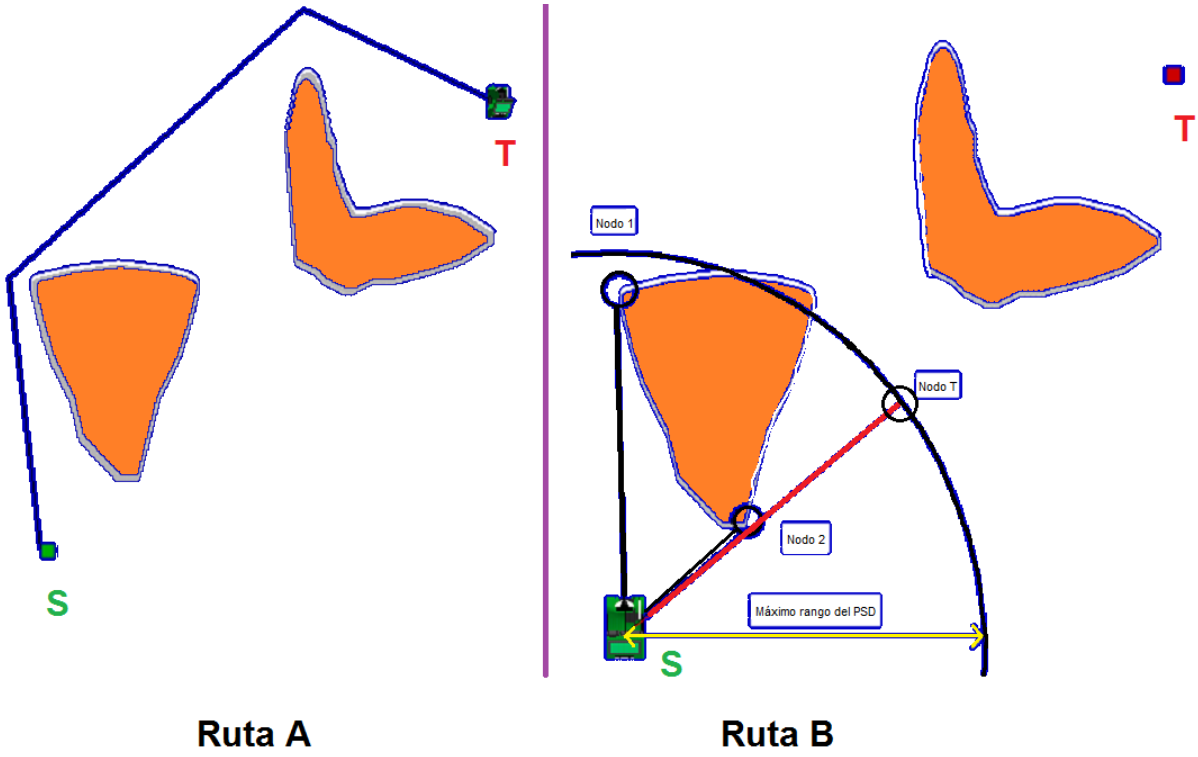


Figura 45.1: Simulación del algoritmo TangentBug en el BCS con el que V. Lumelsky prueba los algoritmos Bug1 y Bug2 [1, 11, 12].

La ruta B es más corta que la ruta A pero el robot decide dirigirse al Nodo 1 porque está más cerca del objetivo T. La Ruta B tendría en cuenta al Nodo T que es aquel que en dirección al objetivo T, está al límite del rango de operación del dispositivo sensible a la posición PSD. Si se lograra una mejora para que el robot decida que el Nodo 2 es más óptimo, se tendría que el algoritmo TangentBug tuviese una ruta ideal de S hacia T. Podría pensarse en lo que implementa el algoritmo D* en la sección 2.2.3 respecto a los costos de desplazamiento. Sin embargo, sería muy complejo implementarlo al TangentBug porque este algoritmo no tiene concepto alguno sobre costos de navegación en su elaboración [11].

7.7. MAPA DE CARACTERÍSTICAS

Para valorar las características de los algoritmos se consideran los resultados de la simulación de los algoritmos Bug1, Bug2, Alg1, Alg2, DistBug, TangentBug, D*, Com, Class1, Rev1, Rev2, OneBug y LeaveBug [12]. Para los algoritmos Dijkstra, A*, Curv1, Curv2, Curv3 y MultiBug se interpreta lo consultado en las fuentes bibliográficas y comparaciones no contenidas en tablas.

7.7.1. Sustentación de las características

7.1.1.1. Algoritmos Bug1 y Bug2

Longitud de ruta: El algoritmo Bug1 circunnavegó en modo *boundary-following*, hasta encontrar nuevamente un único punto H previamente visitado. Por emplear sensores táctiles genera rutas largas porque debe "tocar" un obstáculo, y si se registra un punto muy cercano a la frontera el robot debe alejarse. Presentó errores de navegación lineal y rotacional porque no se empleó un PSD para corregir la dirección de ruta. Emplea el método de finalización del punto más cercano [12, 13].

El algoritmo Bug2 partió hacia T solo si está sobre M. No es tan estricto con el modo seguidor de frontera del Bug 1, pero al emplear un sensor táctil debe ejecutarlo y con esto es vulnerable a trazar dos veces la misma ruta. Los errores de navegación aumentan la longitud de ruta e incluso sin importar que use a M. Emplea el método de M [12]

Operación ante obstáculos: El Bug1 circunnavega el obstáculo para estar seguro que lo ha rodeado. Se observa en los BCS A, B, C y D. No opera ante obstáculos dinámicos porque depende del modo boundary-following. Podría circunnavegar cada obstáculo nuevo que encuentre [12].

El Bug2 no circunnavega la totalidad del obstáculo como se observa en el BCS C pero sigue siendo de baja eficiencia para operar con obstáculos por su modo boundary-following [11, 13].

Seguridad contra colisión: el Bug1 y el Bug2 necesitan establecer una circunferencia con centro en el robot. El Bug2 no depende estrictamente del modo seguidor de frontera [11].

Uso de memoria: Son algoritmos sencillos que requieren módulos comunes [11, 12].

Finalización en tiempo finito: es garantizada porque al regresar a un punto H previo después de la circunnavegación sin haber establecido un punto L finaliza el algoritmo. Los Bug1 y Bug2 realizan un proceso exhaustivo para establecer un punto L. Si la línea desde L a T interseca el obstáculo entonces se establece que T no es alcanzable. [11, 12, 13].

Sencillez en la implementación: Son sencillos por sus bajo requerimiento en líneas de código [11, 12]

Correcciones rápidas ante obstáculos: No presentan correcciones ante obstáculos dinámicos. El modo seguidor de frontera solo presenta variación en la distancia del círculo centrado en el robot el contorno del obstáculo [11].

Eficiencia ante mínimos locales: El Bug1 tiende a circunnavegar el obstáculo hasta determinar el punto L como el caso del BCS B. Teniendo en cuenta la eficiencia no solo de operar ante el obstáculo sino de la longitud de ruta, la eficiencia es baja. El Bug2 se desempeña mejor que el bug1 por poseer una línea M que no le permite circunnavegar.

Giro de 180 grados ante obstáculo: No poseen esta opción, no es propio de su pseudocódigo [11, 12].

Navegación sobre rutas predeterminadas: No.

Uso de línea M: Solo el Bug 2.

Uso de técnicas heurísticas: No.

7.1.1.2. Algoritmos Alg1 y Alg2

Longitud de ruta: El algoritmo Alg1 se comportó como el Bug 2 pero corrigió el repetir la ruta recordando los puntos H y L. Usa la línea M y ejecuta el modo seguidor de frontera de ser necesario. Presentó errores de navegación considerados de rotación. Emplea el método M [12].

El algoritmo Alg2 mejora al Alg 1 por no emplear M. Hace uso de "*y*" que es una posición actual más cercana a T que una posición anterior cercana a T (reasignación). Puede partir inadecuadamente hacia T al emplear el método de finalización de segmentos inhabilitados, por retornar a un punto H más cercano a T [12].

La probabilidad de rotación incrementó debido a la revisaba el estado de F mientras se encontraba en modo seguidor de frontera. Esta revisión la realizaba cada 40mm en rangos de 0 a 45 grados [11].

Operación ante obstáculos: Presentan el mismo comportamiento que el Bug2 como se observa en los BCS C y B.

Seguridad contra colisión: La misma para sus antecesores por parametrizar el robot como un elemento del BCS.

Uso de memoria: Superior a los algoritmos Bug1 y Bug2 [11].

Finalización en tiempo finito: De igual manera que sus antecesores.

Sencillez en la implementación: El Alg1 emplea módulos necesarios para la línea M y la revisión de espacio libre. No es significativa la diferencia entre el número de líneas para ambos, pero si es superior a sus antecesores. El Alg2 puede ser implementado reusando módulos así como el Bug1 [11].

Correcciones rápidas ante obstáculos: No por depender de sensores táctiles. No revisan el espacio libre [11].

Eficiencia ante mínimos locales: El Alg2 mejora a sus antecesores por no depender de la línea M ni de segmentos extensos de los obstáculos [11].

Giro de 180 grados ante obstáculo: No.

Navegación sobre rutas predeterminadas: No.

Uso de línea M: Solo Alg1.

Uso de técnicas heurísticas: No.

7.1.1.3. Algoritmos DistBug y TangentBug

Longitud de ruta: El algoritmo DistBug actualiza continuamente F en dirección de T. Aplica el criterio del método STEP para evitar partir en un punto L cualquiera hacia T. Se dirige a T cuando le sea posible y en caso contrario emplea el modo boundary-following. Debido al ruido que los sensores pueden experimentar, además de no poseer un direccionamiento a T constante, tiende a presentar errores de navegación. Emplea el método STEP [12, 15].

De la misma manera que el Alg2, cada 40mm realizaba una revisión de F. Lo que incrementó la probabilidad de errores de rotación [11].

Longitud de ruta: El algoritmo TangentBug en un espacio de configuración abierto con presencia de obstáculos realmente encontró una ruta corta hacia T al localizar un nodo T. Solo en presencia de un mínimo local su modo cambia a seguidor de frontera. Para el espacio de configuración cerrado, aún en modo seguidor de frontera sigue buscando segmentos discontinuos y generando un nodo T. Por su problema de limitación en la detección de discontinuidades puede dirigirse a un nodo que se encuentre cerca a T, pero que no necesariamente lleve a una ruta más corta. Emplea el método de mínimos locales [12, 13].

Operación ante obstáculos: Ambos presentan sensores de rango y pueden revisar F en caso de obstáculos dinámicos, logrando una LTG. Se desempeñan de manera considerable por no depender del modo seguidor de frontera. El TangentBug por revisar los segmentos en los vértices de los obstáculos y su sensor de rango, actúa de manera evasiva, y emplea los mínimos locales. El DistBug emplea más el modo seguidor de frontera pero aplica el criterio STEP para dirigirse a F. [11, 12, 13].

Seguridad contra colisión: El DistBug podría presentar problemas debido a que en cuanto no hay obstáculo entre el robot y T, puede ignorar lo cerca que puede pasar de obstáculos. Depende de una parametrización del robot como objeto en el BCS. Al TangentBug se le puede implementar la parametrización de curvas y la navegación segura [11, 12, 16]

Uso de memoria: Ambos algoritmos tienen la capacidad del procesamiento en paralelo. El DistBug puede revisar el espacio libre mientras está en modo seguidor de frontera [11]. TangentBug supera a sus antecesores pero es por lo menos la mitad de líneas de código menor que el D* [11].

Finalización en tiempo finito: El DistBug en caso de circunnavegación determina la finalización porque emplea sus sensores para localizar un punto L más cercano a T. TangentBug garantiza aunque requiera circunnavegar el obstáculo. Si los parámetros del método mínimo local no se cumplen porque el robot no puede localizar un espacio para llegar a T (espacio que depende de la parametrización de seguridad y STEP) [12, 13].

Sencillez en la implementación: El DistBug es más sencillo que el Alg2 porque no emplea una estructura de datos. TangentBug emplea varios módulos para su implementación, y por esto no es sencillo de implementar. James Ng realizó una nueva escritura del TangentBug con programación empleada a objetos [11].

Correcciones rápidas ante obstáculos: Ambos cuentan con la opción de revisar sus alrededores gracias a sus sensores de rango, para las simulaciones sensores infrarrojos.

Eficiencia ante mínimos locales: El DistBug tiende a dirigirse a T una vez se no se encuentre obstáculo entre el robot y T. El TangentBug presenta la mejor eficiencia de las simulaciones.

Giro de 180 grados ante obstáculo: No.

Navegación sobre rutas predeterminadas: No.

Uso de línea M: No.

Uso de técnicas heurísticas: Solo el TangentBug.

7.1.1.4. Algoritmos OneBug, MultiBug y LeaveBug

Longitud de ruta: El algoritmo OneBug es similar al Alg 2, excepto que no hace uso de los puntos H y L almacenados. Busca segmentos a lo largo del obstáculo que bloquea su ruta hacia T, y hace uso del método de segmentos inhabilitados. Con este método debe seguir la frontera del obstáculo [12].

Longitud de ruta: El algoritmo MultiBug es similar al Alg 2, excepto que permite la circunnavegación CW ó CCW para explorar la región de segmentos deshabilitados. Depende del modo boundary-following [12].

Longitud de ruta: El algoritmo LeaveBug es similar al Bug 1, excepto que en lugar de circunnavegar completamente el obstáculo antes de evaluar el segmento de línea, completa los segmentos que no le permiten desplazarse hacia T antes de considerar una circunnavegación. Tiende a regresar a un punto L previo para compararlo con el último alcanzado. Emplea el método de segmentos habilitados [12].

Operación ante obstáculos: el OneBug opera como el Alg2 pero con un punto H por segmento deshabilitado. Si T está al alcance se dirige a él. Explora todo un segmento deshabilitado. El LeaveBug explora todo el segmento habilitado y regresa a un punto cercano de T. El MultiBug explora los segmentos deshabilitados. Como a manera de una línea imaginaria el MultiBug evita cruzar la zona habilitada y circunnavega. Puede girar 180 grados [12].

Seguridad contra colisión: El MultiBug podría quedar atrapado dependiendo del espacio entre dos paredes. Si no puede ingresar a un espacio cerrado no explora de no cumplir las condiciones STEP. La distancia STEP debe ser conocida previamente. Determinada como en el Alg1 y Alg2 [12].

Uso de memoria: el OneBug no usa más de un punto H. El MultiBug emplea puntos H y método STEP. Alg1 y Alg2 [12].

Finalización en tiempo finito: No garantizan [12].

Sencillez en la implementación: LeaveBug actualiza y recuerda el punto P. OneBug solo emplea un punto H y circunnavega. El MultiBug explora segmentos deshabilitados, gira 180 grados en exploración más no en obstáculo. En lugar de ingresar a un espacio cerrado,

lo circunnavega. Si desea ingresar se debe cumplir un valor STEP con anterioridad, es decir, conocimiento previo del BCS [12].

Correcciones rápidas ante obstáculos: No. Dependen de sensores táctiles, circunnavegan y no poseen uso de lista completa de puntos visitados [12].

Eficiencia ante mínimos locales: Como los algoritmos Alg1 y Alg2, pasan a modo seguidor de pared. El OneBug y el MultiBug parten a T en cuanto tienen campo de visión. El LeaveBug se aleja de T para revisar el segmento habilitado desconocido [12].

Giro de 180 grados ante obstáculo: No.

Navegación sobre rutas predeterminadas: No.

Uso de línea M: No.

Uso de técnicas heurísticas: No.

7.1.1.5. Algoritmos Rev1 y Rev2

Longitud de ruta: El algoritmo Rev 1 es muy similar al Alg 1, excepto que el robot alterna la dirección (180 grados) cada vez que encuentra un obstáculo con error de navegación. También hace uso de las listas H y L además de la línea M, esta última restringe la libertad de partir inmediatamente a T [12].

Longitud de ruta: El algoritmo Rev 2 es similar al Alg 2, excepto en el cambio de dirección (180 grados) en el modo seguidor de frontera. La diferencia entre el Rev 1 y el Rev 2 es que la sección de segmento es removida (no M). Debido a la estrategia alterna en el modo seguidor de frontera, este algoritmo genera una buena ruta de navegación en BCS cerrados al minimizar la dependencia de este modo [12].

Operación ante obstáculos: Rev1 emplea la línea M pero tiende a realizar desplazamientos para alejarse y acercarse a un obstáculo. Al contar con sensores táctiles la acción de "toque"

del obstáculo permite un cambio de giro si se ha registrado un punto H. Este registro le permitirá rodear el obstáculo y marcar otro punto L más cercano a T [6, 12].

Seguridad contra colisión: el Bug1 y el Bug2 necesitan establecer una circunferencia con centro en el robot. El Bug2 no depende estrictamente del modo seguidor de frontera.

Uso de memoria: El Rev 1 como Alg1 con listas H y L, además de la línea M. El Rev2 solo cambia por no tener que estar sobre la línea M para dirigirse a T [12].

Finalización en tiempo finito: es garantizada porque al regresar a un punto H previo después de la circunnavegación sin haber establecido un punto L finaliza el algoritmo. Los Bug1 y Bug2 realizan un proceso exhaustivo para establecer un punto L. Si la línea desde L a T interseca el obstáculo entonces se establece que T no es alcanzable. [11, 12, 13].

Sencillez en la implementación: Son sencillos por sus bajo requerimiento en líneas de código [11, 12]

Correcciones rápidas ante obstáculos: No presentan correcciones ante obstáculos dinámicos [12].

Eficiencia ante mínimos locales: Presentan el comportamiento de los algoritmos Alg1 y Alg2.

Giro de 180 grados ante obstáculo: Si poseen esta opción en cuanto un punto H registre solo una dirección de exploración. Al registrar un punto H con la exploración horaria y antihoraria, se elimina de la lista [12].

Navegación sobre rutas predeterminadas: No.

Uso de línea M: Solo el Rev1.

Uso de técnicas heurísticas: No.

7.1.1.6. Algoritmo Class1

Longitud de ruta: El algoritmo Class1 no garantiza terminación y se emplea con frecuencia para demostrar lo que sucede cuando el M.A decide partir hacia el objetivo en cualquier otro punto jamás visitado antes. Solo se aplica para mejorar los algoritmos Bug y comprobar que los métodos de finalización deben ser aplicados, como el STEP para el DistBug. Emplea el método segmentos deshabilitados [12].

Operación ante obstáculos: Puede circunnavegar sin encontrar una solución. Por dirigirse a T en cualquier momento que le sea posible puede trazar la misma ruta varias veces [12]

Seguridad contra colisión: Presenta para la simulación un círculo centrado en el robot que le permite estar a una distancia r de la frontera del obstáculo.

Uso de memoria: La más baja [12].

Finalización en tiempo finito: No se garantiza [12].

Sencillez en la implementación: El más sencillo [12]

Correcciones rápidas ante obstáculos: No.

Eficiencia ante mínimos locales: En la simulación muestra un resultado como el Rev2 e incluso casi al punto del DistBug.

Giro de 180 grados ante obstáculo:

Navegación sobre rutas predeterminadas:

Uso de línea M:

Uso de técnicas heurísticas: No.

7.1.1.7. Algoritmos Dijkstra, A* y D*

Longitud de ruta: El algoritmo Dijkstra explora todo el espacio de configuración para buscar T. Las rutas cortas dependen de los pesos o costos que se les asignen a las celdas, producto de la discretización del espacio de configuración. Dependiendo de la distancia rectilínea o euclidiana para la navegación entre celdas, la ruta puede ser corta o no dependiendo de la asignación de los costos en los nodos [13, 15].

Longitud de ruta: El algoritmo A* mejora del Dijkstra al implementar métodos de aproximación heurísticos. Gracias a esto no debe explorar todo el espacio de configuración y sus distancias entre celdas son euclidianas. La longitud de ruta puede variar dependiendo de la geometría de las celdas [13, 14, 15].

Longitud de ruta: El algoritmo D* actúa con eficiencia ante obstáculos dinámicos a diferencia del A*. También ofrece rutas cortas y de igual manera dependen de la geometría de las celdas. Si las celdas no son adecuadas se puede presentar un error de rotación, que está ligado a la longitud de ruta. Puede operar ineficientemente en presencia de mínimos locales pero ofrece competencia con respecto al TangentBug [9, 11, 12, 13].

Operación ante obstáculos: El Dijkstra puede llegar a tomar tiempos de actualización considerables debido a que hace una expansión de su búsqueda para la ruta más corta, dependiendo de la segmentación de un BCS. Fue probado para la búsqueda de rutas cortas o con poca densidad de tráfico por los autores con el propósito de expansión sobre toda una hoja de ruta. Sin embargo la presencia de ruido puede ser importante para un sistema que tiende a no enfocar su búsqueda hacia un punto T, sino en todo el espacio de configuración [7, 15]

El A* depende de la discretización del BCS como su antecesor pero por medio de técnicas heurísticas enfoca la búsqueda de una ruta corta hacia el objetivo. Esto permite que explore y determine costos de desplazamiento solo en objetivos que están entre el robot y T [13, 14].

Seguridad contra colisión: Depende de la discretización del BCS.

Uso de memoria: Son algoritmos que hacen uso considerable de memoria. El Dijkstra al hacer una búsqueda expansiva en todo un rango de visión puede generar un número considerable de iteraciones. El algoritmo A* es pésimo ante la aparición de obstáculos dinámicos. Se reinicia completamente para determinar los costos de desplazamiento

llegando a ocasionar un caso de Overhead Problem. Claro está que este caso tiene más probabilidad en BCS altamente dinámicos y extensos [13, 14].

Finalización en tiempo finito: No se garantiza dependiendo del tamaño del BCS y la cantidad de obstáculos que entran y salen del campo de visión del A*.

Sencillez en la implementación: No es sencillo implementar el A* pero su predecesor puede establecerse como de simplicidad alta Un algoritmo Dijkstra básico puede tener menos de 9 líneas de código, e incluso puede emplearse una rutina "for" para su implementación [13].

Correcciones rápidas ante obstáculos: No.

Eficiencia ante mínimos locales:

Giro de 180 grados ante obstáculo: No.

Navegación sobre rutas predeterminadas: No.

Uso de línea M: No.

Uso de técnicas heurísticas: Si para el A*.

7.1.1.8. Algoritmo Curv

Longitud de ruta: Para el algoritmo Curv que se compone de los algoritmos Curv1, Curv2 y Curv3, la relación entre la ruta corta está relacionada con la hoja de ruta de navegación, especialmente para el Curv 3. Pueden presentar correcciones ante un obstáculo y muchas veces son insignificantes. Para el caso del Curv 3 la longitud de ruta depende de la disponibilidad de ciertos segmentos en la hoja de ruta. El Curv1 presenta una operación de circunnavegación, por lo tanto genera rutas largas en BCS con pistas de navegación extensas [12].

Operación ante obstáculos: Para el Curv1 esta operación depende de la condición de no auto intersecar su ruta de navegación preestablecida, pero no opera ante obstáculos dinámicos. Puede ser posible que el Curv1 realice correcciones ante un obstáculo fijo pero un factor como la tolerancia de la ruta preestablecida es importante. El Curv2 no opera ante obstáculos dinámicos pero puede sobrepasar auto intersecciones, ya que toma estas como obstáculos. El Curv3 opera ante obstáculos de todo tipo, dependiendo de la disponibilidad de la hoja de ruta para ello. De no ser posible encontrar una solución favorable, el algoritmo determina una finalización [12].

Seguridad contra colisión: Para el Curv3 depende de las rutas preestablecidas. Comúnmente este tipo de problema tiene solución en un algoritmo de nivel Curv3, pero sus antecesores no podrían hacerlo ante obstáculos dinámicos [12].

Uso de memoria: Uso alto para el Curv3 y básico para Curv1 y Curv2 por no almacenar listas de puntos H o L.

Finalización en tiempo finito: Para el Curv1 se tiene la condición de que el objetivo se puede alcanzar. El Curv2 fue desarrollado para garantizar finalización y el Curv3 sigue este desarrollo.

Sencillez en la implementación: El Curv1 emplea contadores y un modo seguidor de pista de navegación. El Curv2 emplea afluentes y salidas que son controladas por medio de un ángulo de giro horario (dependiendo del implementador) A, que permite la navegación 1 a 1, es decir una navegación de S a T. El Curv3 opera con obstáculos dinámicos y emplea el principio de emparejamiento una entrada varias salidas SIMO [12].

Correcciones rápidas ante obstáculos: Solo para Curv3.

Eficiencia ante mínimos locales: para Curv3 esta característica depende de la cantidad de obstáculos que estén cerca a T, y la condición de paridad.

Giro de 180 grados ante obstáculo: No.

Navegación sobre rutas predeterminadas: Si.

Uso de línea M: No.

Uso de técnicas heurísticas: No.

8. REFERENCIAS Y BIBLIOGRAFÍA

[1] V. Lumelsky y A. Stephanov, *Path-Planning Strategies for a Point Mobile Automaton Moving Amidst Unknown Obstacles of Arbitrary Shapes*, Algorithmica, vol. 2, pp. 403-430, 1987.

[2] A. Sankaranarayanan y M. Vidyasagar, *A new path planning algorithm for moving a point object amidst unknown obstacles in a plane*, IEEE Conference on Robotics and Automation, pág. 1930-1936, 1990.

[3]A. Sankaranarayanan and M. Vidyasagar, Path Planning for Moving a Point Object Amidst Unknown Obstacles in a Plane: A New Algorithm and a General Theory for Algorithm Development, Proceedings of the 29th IEEE Conference on Decision and Control, pp. 1111-1119, 1991.

[4] I. Kamon y E. Rivlin, *Sensory-based motion planning with global proofs*, IEEE Transaction on Robotics and Automation, vol. 13, pág 814-822, 1997.

[5] I. Kamon, E. Rivlin y E. Rimon, *TangentBug: A range-sensor based navigation algorithm*, Journal of Robotics Research, vol. 17, No 9, pág 934-953, 1996.

[6] Y. Horiuchi y H. Noborio, *Evaluation of path length made in sensor-based path-planning with the alternative following*, Proc. of the 2001 IEEE International Conference on Robotics and Automation, pág 1728-1735, 2001.

[7] P.W. Eklund, S. Kirkby, S. Pollitt, *A dynamic multi-source Dijkstra's algorithm for vehicle routing*, IEEE Conference on Intelligent Information Systems, pp. 329-333, 1996.

[8] Peter E. Hart, Nils J. Nilsson, Bertram Raphael, *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*, IEEE Transactions on Systems Science and Cybernetics, vol. 4, pág 100-107, 1968.

[9] A. Stentz, *Optimal and efficient path planning for partially-known environments*, Proc. of IEEE Conference on Robotic Automation, pág 1058-1068, 1994.

[10] A. Sankaranarayanan é I. Masuda, *A new algorithm for robot curve-following amidst unknown obstacles, and a generalization of maze-searching*, IEEE International Conference on Robotics and Automation, pág 2487-2494, 1992.

[11] Ng. James, *A Practical Comparison of Robot Path Planning Algorithms given only Local Information*, University Of Western Australia, Centre for Intelligent Information Processing Systems, pág 10-26, 56-86, 2005.

[12] Ng. James, *An analysis of mobile robot navigation algorithms in unknown environments*, University Of Western Australia, School of Electrical, Electronic and Computer Engineering, pág 17-44, 52-58, 60-71, 72-86, 107-126, 2010.

[13] H. Choset, K. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. Kavraki y S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementation*, pág 17-38, 107-110, 161-169, 521-546, 2005.

[14] J. Latombe, *Robot Motion Planning*, pág 43-48, 105-122, 153-167, 603-608, 1991.

[15] T. Bräunl, *Embedded Robotics: Mobile Robot Design and Applications with Embedded Systems*, pág 83-90, 197-216, 2006.

[16] A. Matveev, A. Savkin, M. Hoy, C. Wang, *Safe Robot Navigation Among Moving and Steady Obstacles*, pág 51-63, 2016.

[17] J.B. Hayet, F. Lerasle y M. Devy, *A visual landmark framework for indoor mobile robot navigation*, IEEE Conference on Robotics and Automation, pp. 3942-3947, 2002.

9. ANEXOS

9.1. MAPA DE CARACTERÍSTICAS

9.1.1. Mapa de características a partir de los resultados de las simulaciones

Características	Bug1	Bug 2	Alg 1	Alg 2	DistBug	TangentBug	OneBug	LeaveBug	Rev 1	Rev 2	Class 1	D*
Longitud de ruta	*BAJA*	*BAJA*	*BAJA*	MEDIA	ALTA	ALTA	*BAJA*	*BAJA*	MEDIA	MEDIA	*BAJA*	ALTA
Operación ante obstáculos	*BAJA*	*BAJA*	MEDIA	MEDIA	MEDIA	ALTA	*BAJA*	MEDIA	MEDIA	MEDIA	*BAJA*	ALTA
Seguridad contra colisión	*BAJA*	MEDIA	MEDIA	MEDIA	*BAJA*	ALTA	*BAJA*	ALTA	ALTA	ALTA	*BAJA*	ALTA
Uso de memoria	ALTA	ALTA	MEDIA	*BAJA*	ALTA	*BAJA*	ALTA	MEDIA	MEDIA	MEDIA	ALTA	*BAJA*
Garantiza finalización en tiempo finito	SI	SI	SI	SI	SI	SI	*NO*	*NO*	*NO*	*NO*	*NO*	*NO*
Complejidad	ALTA	ALTA	*BAJA*	MEDIA	MEDIA	*BAJA*	MEDIA	MEDIA	*BAJA*	MEDIA	ALTA	*BAJA*
Correcciones rápidas ante obstáculos	*NO*	*NO*	*NO*	*NO*	MEDIA	MEDIA	*NO*	*NO*	*NO*	*NO*	*NO*	ALTA
Eficiencia ante mínimos locales	*BAJA*	MEDIA	MEDIA	MEDIA	MEDIA	ALTA	*BAJA*	*BAJA*	MEDIA	MEDIA	MEDIA	MEDIA
Giro de 180 grados ante obstáculo	*NO*	*NO*	*NO*	*NO*	*NO*	*NO*	*NO*	*NO*	SI	SI	*NO*	*NO*
Navegación sobre rutas predeterminadas	*NO*	*NO*	*NO*	*NO*	*NO*	*NO*	*NO*	*NO*	*NO*	*NO*	*NO*	*NO*
Uso de línea M	*NO*	SI	SI	*NO*	*NO*	*NO*	*NO*	*NO*	SI	*NO*	*NO*	*NO*
Uso de técnicas heurísticas	*NO*	*NO*	*NO*	*NO*	*NO*	*NO*	*NO*	*NO*	*NO*	*NO*	*NO*	SI

9.1.2. Mapa de características a partir de la interpretación bibliográfica

Características	MultiBug	Dijkstra	A*	Curv 1	Curv 2	Curv 3
Longitud de ruta	*BAJA*	MEDIA	**ALTA**	MEDIA	MEDIA	MEDIA
Operación ante obstáculos	MEDIA	*BAJA*	**ALTA**	*BAJA*	MEDIA	**ALTA**
Seguridad contra colisión	MEDIA	*BAJA*	MEDIA	*NO*	*BAJA*	MEDIA
Uso de memoria	**ALTA**	*BAJA*	*BAJA*	**ALTA**	MEDIA	MEDIA
Garantiza finalización en tiempo finito	*NO*	*NO*	*NO*	*NO*	*NO*	*NO*
Complejidad	MEDIA	MEDIA	*BAJA*	MEDIA	*BAJA*	*BAJA*
Correcciones rápidas ante obstáculos	*NO*	*BAJA*	*BAJA*	*NO*	*BAJA*	MEDIA
Eficiencia ante minimos locales	MEDIA	*BAJA*	MEDIA	*BAJA*	MEDIA	MEDIA
Giro de 180 grados ante obstáculo	*NO*	*NO*	*NO*	*NO*	*NO*	*NO*
Navegación sobre rutas predeterminadas	*NO*	*NO*	*NO*	**SI**	**SI**	**SI**
Uso de línea M	**SI**	*NO*	*NO*	*NO*	*NO*	*NO*
Uso de técnicas heurísticas	*NO*	*NO*	**SI**	*NO*	*NO*	*NO*