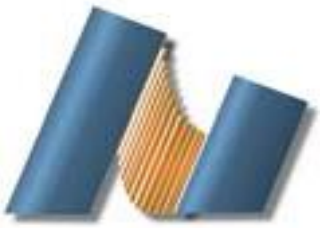
UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
CENTRO DE NANOCIENCIAS Y NANOTECNOLOGÍA



LICENCIATURA EN NANOTECNOLOGÍA
MICROELECTRÓNICA Y NANOFABRICACIÓN

ANÁLISIS DE ALGORITMOS DE BÚSQUEDA DE CAMINO EN REPRESENTACIONES DE CUADRÍCULA

TESIS
QUE PARA OPTAR POR EL TÍTULO DE:
LICENCIADO EN NANOTECNOLOGÍA

PRESENTA:
IVÁN JOSUÉ SAAVEDRA SOTO

DIRECTOR DE TESIS
MTRO. ARITZ BARRONDO CORRAL

ENSENADA, BAJA CALIFORNIA NOVIEMBRE 2020

Hago constar que el trabajo que presento es de mi autoría y que todas las ideas, citas textuales, datos, ilustraciones, gráficas, etc. sacados de cualquier obra o debidas al trabajo de terceros, han sido debidamente identificados y citados en el cuerpo del texto y en la bibliografía y acepto que en caso de no respetar lo anterior puedo ser sujeto a sanciones universitarias.

Afirmo que el material presentado no se encuentra protegido por derechos de autor y me hago responsable de cualquier reclamo relacionado con la violación de derechos de autor.

_____

IVÁN JOSUÉ SAAVEDRA SOTO

ÍNDICE

Parte I

INTRODUCCIÓN

¿QUÉ ES UN ALGORITMO?

Un algoritmo es una serie de pasos computacionales bien definidos que toman un valor o conjunto de valores como entrada y producen un valor o conjunto de valores como salida [1].

También se puede pensar en un algoritmo como la solución a un problema computacional donde el algoritmo establece la relación que existe entre una entrada y salidas dadas.

Un ejemplo son los algoritmos de búsqueda de caminos. Estos algoritmos toman como entrada un grafo, un vértice de partida y un vértice objetivo y devuelven como salida una lista de aristas que conforman el camino más corto. La serie de pasos para obtener dicha lista es la descripción del algoritmo.

Para que un algoritmo sea útil no sólo debe funcionar en un ejemplo particular o en una instancia de un problema, sino que deben abarcar todas las instancias. Volviendo al ejemplo anterior de la lista de números, si un algoritmo de ordenamiento sólo pudiera ordenar dicha lista, entonces no sería muy útil a comparación de un algoritmo que puede ordenar esa y cualquier otra lista conformada de números.

Se puede ser más específico en el dominio del problema que resuelve un algoritmo, por ejemplo, si los elementos de la lista que ordena deben ser números positivos enteros o si también se incluyen números negativos o fracciones, etc. Pero la finalidad de un algoritmo es que sea capaz de resolver cualquier instancia dentro del dominio del problema.

Los algoritmos de búsqueda de camino toman como entrada un grafo, un vértice de partida y un vértice objetivo; y devuelven como salida una lista de vértices o aristas (dependiendo de la implementación) a recorrer para llegar al vértice objetivo desde el vértice inicial.

Un grafo es una estructura matemática conformada por vértices (o nodos) y aristas que relacionan dichos vértices. Una representación visual de un grafo se ilustra en la *Figura 1*. Los círculos representan los vértices, mientras que las líneas representan las aristas que relacionan dichos vértices. Además, a través de las aristas es posible desplazarse de un vértice a otro.

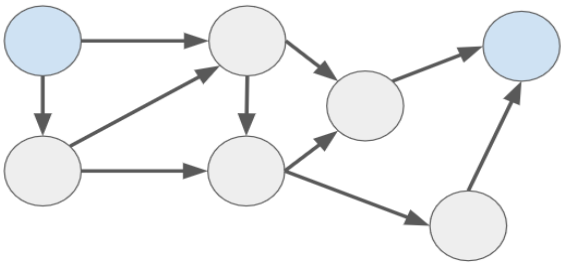


Figura 1. Representación visual de un grafo dirigido. Los vértices están representados con círculos, y las aristas con flechas. Los vértices marcados en azul indican un punto de partida y un objetivo, ¿cuál es el camino más corto para ir de un vértice azul a otro?

Sin embargo, observe que las aristas están representadas con flechas, indicando que sólo se puede transitar de un vértice a otro hacia una dirección.

Supóngase que se desea encontrar el camino más corto para ir desde el vértice en la esquina superior izquierda marcado en azul hacia el otro vértice marcado del mismo color. ¿Cuál es el camino óptimo? Se incita al lector a encontrar la solución.

La solución al problema se muestra a continuación en la *Figura 2* donde las flechas marcadas en rojo indican las aristas a transitar.

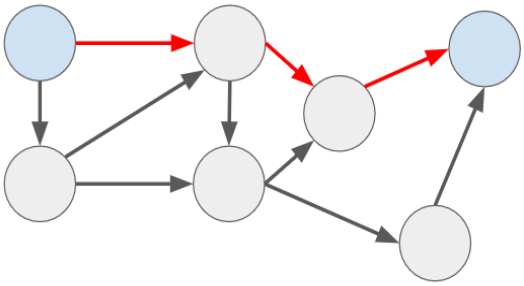


Figura 2. Solución al problema planteado en la *Figura 1*. Las flechas en rojo indican las aristas que describen el camino más corto entre ambos vértices marcados en azul.

Ahora considérese el grafo de la *Figura 3.* Observe que cada arista tiene asociado un número. Cada número es el *costo* que toma transitar de un vértice a otro. Una manera intuitiva de verlo es considerar el costo como la distancia que toma ir de un vértice a otro recorriendo la arista que los conecta.

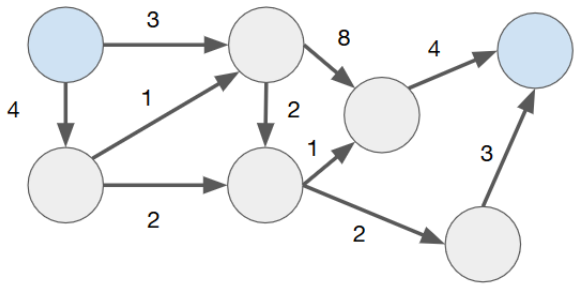


Figura 3. Representación visual de un grafo unidireccional ponderado. Los vértices marcados en azul indican el punto de partida y el objetivo y los números sobre las aristas su respectivo costo ¿cuál es el camino para ir de un vértice azul a otro con el costo mínimo?

De igual manera supóngase que se desea encontrar un camino entre los dos vértices marcados en azul de tal forma que la suma del costo de las aristas sea el mínimo. ¿Cuál es el camino óptimo? De nuevo se incita al lector a encontrar la solución. A continuación, se presenta la solución al problema en la *Figura 4*.

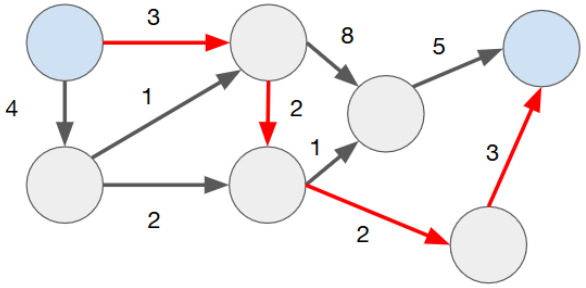


Figura 4. Solución al problema presentado en la Figura 3. Las aristas marcadas en rojo describen el camino con el menor costo para dirigirse de un vértice azul a otro.

Una estrategia común que utilizan las personas para solucionar el problema es primero encontrar un camino que conduzca de un vértice azul a otro y sumar el costo de las aristas. Después intentar hallar otros caminos y de igual manera calcular el costo. Si alguno de estos caminos tiene menor peso que el encontrado inicialmente, se elige este nuevo camino como la solución y se sigue buscando más rutas.

Dicha estrategia es lo que se suele hacer de manera intuitiva, pero sólo es efectiva para grafos pequeños. Por ejemplo, supongamos que se desea encontrar el camino más corto para ir de Ensenada a Ciudad de México. Se asigna a cada vértice un punto de interés y cada arista representa una carretera. Una ilustración se presenta en la *Figura 5*.

Figura 5. Mapa de México con algunos puntos de interés marcados con vértices y aristas representando carreteras. El camino más corto para ir de Ensenada a Cd. México es pasar por Tecate, Mexicali, Hermosillo, Ciudad Obregón, Culiacán, Mazatlán, Tepic, Guadalajara, Morelia y finalmente llegar a Cd. México.

Cuando el grafo es lo suficientemente grande, el problema toma demasiado tiempo para que lo resuelva una persona y entonces se recurre a utilizar una computadora.

Sin embargo, la estrategia de encontrar todos los caminos posibles y seleccionar el que tiene mayor peso no es eficiente. Por ejemplo, no tiene caso considerar los caminos que comienzan en Ensenada, pasan por Mérida, Yucatán y que finalmente llegan a Ciudad de México porque Mérida está al otro extremo del país.

El qué tan eficiente es la estrategia para resolver este tipo de problemas es de vital importancia para ahorrar tiempo de cómputo, sobre todo en grafos de mayor tamaño. Mientras más grande es el grafo, más tiempo toma computar la solución dependiendo de la estrategia utilizada.

Habiendo dejando en claro algunos conceptos, se define el problema principal que se aborda en este trabajo.

PROBLEMA DEL CAMINO MÁS CORTO

El ***problema del camino más corto*** consiste en encontrar en un grafo el camino entre dos vértices tal que la suma del costo de las aristas que constituyen dicho camino sea *mínima* [2].

Para resolver este problema se utilizan algoritmos denominados *algoritmos de búsqueda de camino*. Dichos algoritmos comúnmente requieren como entrada un grafo, un vértice de partida y un vértice objetivo para producir como salida el conjunto de aristas a recorrer para ir del vértice de partida hacia el objetivo.

El problema del camino más corto aparece en una gran variedad de campos de estudio. Una de las aplicaciones más intuitivas es el trazado de rutas en mapas recabados por satélites; servicio ofrecido por proveedores como Google Maps, Apple Maps y OpenStreetMap.

Otras aplicaciones son en sistemas de navegación automáticos en automóviles [3], ensamblaje automatizado de nanoestructuras utilizando microscopios electrónicos [4], control autónomo de microrobots en el sistema cardiovascular [5], identificación de nuevos genes asociados a cáncer [6], control de robots llamados rovers para exploración espacial [7], en telecomunicaciones cuando se transmiten paquetes de datos en varias localizaciones geográficas [8], etc.

En este trabajo se realiza un análisis de algoritmos de búsqueda de camino para entornos de cuadrícula 2D a través del estudio de la literatura estableciendo las ventajas y desventajas de cada uno. La finalidad es identificar cuál estrategia o algoritmo es el más eficiente para resolver el problema del camino más corto y bajo qué circunstancias. Se recopilan tiempos de ejecución de cada algoritmo en distintos mapas utilizando como medida el número de operaciones que se requieren para computar las soluciones.

OBJETIVOS DE ESTA TESIS

Objetivo general:

- Determinar cuáles algoritmos son más eficientes para resolver el problema del camino más corto dependiendo de la estructura del mapa en el que se

ejecutan.

Objetivos particulares:

- Describir varios algoritmos de búsqueda de camino y realizar un análisis en el que se determinen sus ventajas y desventajas.
- Implementar varios algoritmos de búsqueda de camino y ejecutarlos en distintos mapas.
- Recopilar tiempo de cómputo para cada algoritmo en cada mapa medido en número de operaciones.

ESTRUCTURA DE ESTA TESIS

La **sección II** de este escrito comienza describiendo algunas de las aplicaciones al resolver el problema del camino más corto, haciendo hincapié a aquellas relacionadas con las nanociencias.

En la **sección III** se explica toda la teoría respecto la implementación de cada algoritmo y la manera en la que se cuantifica el tiempo de ejecución. Primero se comienza definiendo lo que es un algoritmo y después se detalla lo que es la notación O grande y cómo puede utilizarse para medir la eficiencia de un algoritmo. Posteriormente se describen las distintas maneras de implementar un grafo en una computadora y cómo pueden utilizarse los grafos para representar mapas o mundos utilizando la representación de cuadrícula. Finalmente, la sección concluye presentando distintos algoritmos de búsqueda de camino, describiendo la implementación de cada uno, sus órdenes de crecimiento y sus ventajas y desventajas.

En la **sección IV** se detalla la realización de los experimentos para determinar la eficiencia de cada algoritmo. Se especifica cada uno de los mapas utilizados, se grafica el número de operaciones que toma para cada algoritmo computar la solución en un mapa y se elabora el análisis de resultados.

Finalmente, en la **sección V** y **sección VI** se presentan las conclusiones de este trabajo y se enlista la bibliografía utilizada para la realización del mismo.

Parte II

APLICACIONES

Los algoritmos de búsqueda de camino resuelven el problema de encontrar el camino óptimo en un grafo para ir de un vértice a otro. Y dado que los grafos son objetos matemáticos abstractos que pueden utilizarse para modelar problemáticas en las que existan una serie de elementos (vértices) que guardan relaciones entre sí (aristas), la resolución del problema del camino más corto a través de algoritmos da lugar a una amplia variedad de aplicaciones.

En esta sección se detallan algunas de las aplicaciones al emplear este tipo de algoritmos.

TRAZADO DE RUTAS EN MAPAS

Hoy en día gracias a las imágenes satelitales de la Tierra ha sido posible construir mapas de ciudades enteras y con el uso de teléfonos celulares acceder a dichos mapas. Varias plataformas ofrecen este servicio, entre ellas se encuentra Google Maps, Apple Maps, OpenStreetMap, MAPS.ME, etc.

Entre las funcionalidades de este tipo de servicio se encuentra el trazar una ruta para ir de un sitio a otro, ya sea a pie o utilizando un medio de transporte. Esta modalidad hace uso de algoritmos de búsqueda de camino para poder brindar el camino más corto. Varios puntos de interés como intersecciones o establecimientos son representados como los vértices de un grafo y la distancia que hay entre estos puntos de interés simbolizan las aristas.

EXPLORACIÓN ESPACIAL CON ROVERS

Un rover es un robot diseñado para la exploración de las superficies de otros planetas cuyas funciones principales suelen ser recabar información sobre el terreno y recolectar muestras de polvo, tierra, rocas e incluso líquidos. Dichos dispositivos pueden ser parcial o completamente autónomos.

Curiosity es un rover diseñado para explorar Marte que cuenta con dos métodos principales para explorar. El primero se denomina “exploración a ciegas”, donde el rover simplemente se dirige hacia un objetivo sin identificar terrenos peligrosos. El segundo es un sistema de navegación automático donde el rover identifica obstáculos

de manera autónoma, tales como rocas grandes y los evita para alcanzar su objetivo [9].

Ambos métodos tienen sus ventajas y desventajas. Para el caso de la “exploración a ciegas” el rover puede cubrir grandes distancias en cierto periodo de tiempo debido a que no tiene que procesar las imágenes del terreno. La desventaja de este enfoque está en que los ingenieros en la Tierra deben verificar que el camino entre el rover y el objetivo esté libre de obstáculos y terrenos peligrosos antes de ejecutar el comando. Por otro lado, el sistema de navegación automática es más lento, pero mantiene al rover seguro durante su trayecto, sin necesidad de que ingenieros lo supervisen. A menudo, los dos métodos se utilizan en conjunto. Primero se realiza una exploración a ciegas, tan lejos como sea seguro y después se utiliza el sistema de navegación automático para seguir explorando en terreno desconocido.

El sistema de navegación automático utiliza algoritmos de búsqueda de camino para trazar una ruta que evite los obstáculos siguiendo el camino óptimo y algunas implementaciones utilizan versiones tridimensionales de dichos algoritmos [7].

CONTROL DE MICROROBOTS DENTRO DEL SISTEMA CARDIOVASCULAR

Múltiples reportes existen sobre la posibilidad de utilizar algoritmos de búsqueda de camino para el control del movimiento de microrobots dentro del sistema cardiovascular con el objetivo de transportar fármacos a una región específica o acceder a zonas que por cirugía convencional son de alto riesgo o inaccesibles [10][11][12]. La necesidad de tener un sistema de navegación automático es debido a la complejidad del sistema cardiovascular y la gran cantidad de posibles caminos para ir de un lugar a otro. El control manual puede ser lento y tedioso y por ese motivo se opta por un sistema de navegación automático.

La estrategia principal es generar imágenes del sistema cardiovascular utilizando técnicas de imagen por resonancia magnética (MRI). A las imágenes generadas se les aplica filtros de contraste para poder discernir los obstáculos de las zonas por las cuales puede transitar el microrobot.

En su artículo *Robotic Path Planning Using A* Algorithm for Automatic Navigation in Magnetic Resonance Angiography*[13] Youchou utiliza el algoritmo de búsqueda de

camino A* para encontrar el camino más corto en imágenes obtenidas por MRI de vasos sanguíneos ubicados en tejido cerebral. En una interfaz de usuario selecciona dos píxeles que fungen como punto de partida y el punto objetivo.

Youchou utiliza dos conjuntos de imágenes de 454x318 y 270x170 píxeles respectivamente. En sus resultados la computación de la solución en el primer conjunto de imágenes ronda alrededor de los 5500 segundos, mientras que en el segundo toma alrededor de 1100 segundos. Mientras menor es la resolución de las imágenes, menor el tiempo de la computación, pero a cambio de que la imagen no sea una reproducción fiel a la realidad.

En su artículo *Use of 3D Potential Field and an Enhanced Breadth-first Search Algorithms for the Path Planning of Microdevices Propelled in the Cardiovascular System*[5] Sabra describe una metodología similar utilizando el algoritmo de búsqueda de camino Breadth First Search en tres dimensiones con imágenes de varios cortes de tejido cerebral. Con las imágenes recabadas construye una estructura en MATLAB que representa el tejido cerebral en tres dimensiones y resuelve el problema de encontrar el camino para ir de un punto a otro, aunque en su artículo no provee información del tiempo de computación en los experimentos realizados.

En la actualidad aún existen limitaciones tecnológicas para implementar un sistema de navegación de este tipo. Una de estas limitaciones recae en el recabado de las imágenes. Por ejemplo, la técnica MRI toma imágenes de manera secuencial y debido a que el cuerpo está en constante movimiento (ya sea de manera voluntaria o involuntaria) algunas de las imágenes obtenidas por la técnica aparecen discontinuas, no representando como luce en realidad el sistema cardiovascular del paciente.

Otro problema que complica la implementación de este sistema es que los vasos sanguíneos cambian su volumen debido a que se contraen. Esto causa que en las imágenes parezca que algunos vasos sanguíneos conducen a un camino sin salida, cuando en realidad no es así.

Otras cuestiones recaen en la física del movimiento del microdispositivo. Por ejemplo, la fuerza de propulsión del dispositivo debe ser lo suficientemente fuerte para contrarrestar la corriente del flujo sanguíneo, el cual suele ser de mayor magnitud en vasos sanguíneos de mayor diámetro. En el caso de que el microdispositivo sea muy

pequeño en comparación al diámetro del vaso sanguíneo, la fuerza de propulsión puede no ser suficiente para contrarrestar la corriente y se perdería control del dispositivo. Por lo tanto, el algoritmo debe tomar en cuenta también el evitar transitar en vasos sanguíneos con un diámetro grande.

De manera similar, si el tamaño del microdispositivo es grande en comparación al diámetro del vaso sanguíneo, el riesgo de que ocurra una obstrucción aumenta. En ambas situaciones se puede poner en riesgo la salud del paciente.

Adicionalmente la forma del microdispositivo influye en la fuerza de arrastre, la cual debe ser conocida y considerada por el software. Otros factores a tomar en cuenta son la flotabilidad, la viscosidad de la sangre y la amplitud de la contracción de los vasos sanguíneos para asegurar la fiabilidad del movimiento y la mayor seguridad posible.

Los resultados de los artículos muestran que es viable implementar software que permita automatizar el sistema de navegación de un microdispositivo. No obstante, es requerido que se realice trabajo adicional que considere la física del microdispositivo.

AUTOMATIZACIÓN DE ENSAMBLADO DE NANOESTRUCTURAS

La ingeniería y fabricación de nanorobots es un campo emergente de la nanotecnología. Prácticamente estos sistemas son dispositivos nanoelectromecánicos capaces de llevar a cabo tareas y funciones preprogramadas de manera precisa y fehaciente con energía provista por un nanomotor o una nanomáquina [14]. Varias de sus potenciales aplicaciones entran en el campo de la nanomedicina para el diagnóstico de enfermedades, transporte de fármacos, quimioterapia y operaciones quirúrgicas no invasivas [15]. Las dimensiones de estos dispositivos están por debajo o alrededor del rango de los micrómetros (0.01 μm a 10 μm); sin embargo, las partes que lo conforman son componentes a nanoescala como nanopartículas, nanovarillas, biomoléculas o polímeros.

Gracias a los avances tecnológicos, es posible manipular estos componentes de manera individual utilizando la punta de microscopios electrónicos tales como el microscopio de fuerza atómica (AFM) o el microscopio de sonda de barrido (SPM).

Sin embargo, construir patrones con un gran número de nanopartículas (potencialmente de cientos o miles) de manera interactiva es tedioso y se presta a error humano al posicionarlas en un lugar preciso. Por esta razón surge la necesidad de automatizar el proceso de planear el camino que va a seguir la punta del microscopio para posicionar varias nanopartículas en sitios específicos.

En su artículo *Automatic Planning of Nanoparticle Assembly Tasks* [16] Makaliwe describe una metodología para resolver este problema utilizando algoritmos de búsqueda de camino. El problema es abstraído planteando que la punta puede considerarse como un robot que opera en dos estados: el estado activo y el no activo. En el estado activo la punta es capaz de empujar un objeto movible cuyo centro se encuentre en el camino que sigue la punta. Mientras que en el estado no activo su movimiento no tiene ningún efecto en objetos movibles u obstáculos.

Además, se definen las nanopartículas como un conjunto de objetos movibles de forma circular cuya localización es conocida y se describen los obstáculos como objetos que la punta debe evitar entrar en contacto en el estado activo. Dichos obstáculos representan objetos que no pueden ser movidos o nanopartículas que ya han sido posicionadas en el lugar correcto.

El problema puede ser traducido a buscar el camino óptimo para ir de un punto a otro evitando una serie de obstáculos. El mapa del terreno puede obtenerse utilizando las imágenes recabadas por el propio microscopio y procesarlas en un equipo de cómputo conectado directamente. De esta manera Makaliwe logró mover alrededor de 50 nanopartículas de manera automatizada en el rango de 5-10 segundos.

Aunque el estudio de la manufactura automatizada está ampliamente estudiado a macroescala, poco se ha hecho al respecto a nanoescala. El paso de ir a macroescala a nanoescala no es trivial. Las investigaciones realizadas a nanoescala se limitan exclusivamente a utilizar nanopartículas porque sólo requieren operaciones de traslación. Para el caso de otras nanoestructuras como las nanovarillas, se pueden realizar movimientos de translación o rotación, aumentando la dificultad del problema [17].

En la actualidad el ensamblaje de nanorobots por medios artificiales como el uso de microscopios electrónicos es bastante limitado, pero es un concepto que se ha

explorado de manera persistente. Por ahora la manera práctica en la cual se han ensamblado nanorobots ha sido por medios biológicos utilizando proteínas o ADN. En el futuro es posible que se realice más investigación al respecto y pueda realizarse el ensamblaje de múltiples maneras.

IDENTIFICACIÓN DE GENES ASOCIADOS A CÁNCER

Inicialmente, las investigaciones en el área de la bioinformática estaban enfocadas al estudio individual de moléculas y componentes celulares. Sin embargo, los sistemas biológicos son complejos y algunos mecanismos no pueden ser explicados sólo estudiando sus componentes de manera aislada porque funcionan de forma diferente en distintos procesos [18].

Por esa razón, la bioinformática ha cambiado su enfoque hacia estudiar el cómo interactúan los componentes biológicos entre sí y el reto es analizar y comprender la estructura y el comportamiento de esta red de interacciones.

Para modelar esta red de interacciones entre moléculas biológicas se utilizan grafos, donde los vértices representan distintos componentes biológicos y las aristas describen la interacción entre dos de estos componentes.

Comúnmente se utilizan redes de interacción proteína-proteína debido a que las proteínas son los agentes principales de las funciones biológicas; controlan mecanismos a nivel molecular y celular y determinan si un organismo sufre alguna enfermedad o se encuentra en un estado saludable.

Las redes de interacción proteína-proteína brindan información para realizar predicción sobre la función de algunos genes si se supone que las proteínas que interactúan entre sí, tienen funciones similares o idénticas. En realidad, se ha mostrado que en una red de interacción proteína-proteína donde un par de proteínas que están conectadas a través de un camino corto suelen tener las mismas funciones [19].

En su artículo de investigación *Identification of Lung-Cancer-Related Genes with the Shortest Path Approach in a Protein-Protein Interaction Network* [20] Bi-Qing Li y su equipo obtienen una red de interacción proteína-proteína de una base de datos

llamada *Search Tool for the Retrieval of Interacting Genes* (STRING) [21]. Esta red de interacción incluye proteínas asociadas a genes que pueden causar cáncer de pulmón. Cada una de las aristas que unen una proteína con otra tienen asociado un número del 1 al 999 que cuantifica qué tan probable es que una proteína interactúe con la otra. Un número pequeño indica que es muy probable que interactúen, mientras que un número grande significa lo contrario. Puede pensarse en las aristas como las distancias que hay de una proteína a otra y entre menor la distancia, es más probable que las proteínas tengan funciones idénticas o similares.

Con dicha información se computó la distancia más corta entre todos los pares posibles de proteínas y de esta manera se identificaron nuevos genes que podrían estar asociados a cáncer de pulmón. Metodologías similares se han utilizado para identificar genes que pueden estar asociados a cáncer de mama [22][23], cáncer colorrectal[24], cáncer de próstata[25], cáncer de estómago[26] y cáncer de páncreas[27].

Parte III

MARCO TEÓRICO

ANÁLISIS DE ALGORITMOS

Aunque existen toda clase algoritmos para resolver un problema específico, no todos se utilizan debido a que algunos no son prácticos. Por ejemplo, una estrategia para resolver el problema del camino más corto es encontrar todos los caminos para ir de un vértice partida a un vértice objetivo, calcular el peso de las aristas para cada camino y seleccionar el que tenga el menor peso. El problema con esta estrategia es que para grafos grandes se requiere un gran número de pasos para computar la solución y el aumentar el número de vértices o aristas se presta a que el número de pasos aumente de manera súbita.

Los pasos u operaciones que realiza una computadora toman tiempo y con una cantidad de pasos lo suficientemente grande, se puede terminar realizando un cálculo que puede tomar miles de años en completarse o incluso más tiempo que la edad actual del universo. Por esta razón, se realiza el análisis de algoritmos para seleccionar aquellos algoritmos más eficientes.

El análisis de algoritmos es un proceso que consiste en predecir los recursos computacionales que requiere un algoritmo, ya sea memoria, hardware o tiempo.

En la mayoría de los casos es de interés saber el tiempo de ejecución. De esta manera, si se conoce cuánto tiempo tardan distintos algoritmos en producir sus respectivas salidas, se puede determinar cuál algoritmo es más eficiente.

El tiempo de ejecución de un algoritmo están en función del tamaño de la entrada que se le provee. Encontrar el camino más corto de un vértice a otro en un grafo relativamente pequeño que representa las calles de un vecindario es más rápido de computar que en un grafo que representa las calles de todo un país. Además, el tiempo puede variar para grafos del mismo tamaño según los vértices de origen y destino que se eligen.

Dependiendo del problema que se estudia, el tamaño de la entrada puede variar. En el caso de los algoritmos de ordenamiento, el tamaño de la entrada es el número de elementos de la lista a ordenar. Para el caso de algoritmos de búsqueda de camino, es conveniente representar el tamaño de la entrada con dos números: El número de vértices y el número de aristas en el grafo.

Para medir el tiempo de ejecución podrían utilizarse segundos, milisegundos o microsegundos, pero en ese caso los tiempos dependen de los componentes de la computadora donde se ejecutó el algoritmo o el lenguaje de programación utilizado. En cambio, si se utiliza como unidad de medida el número de pasos u operaciones que requiere un algoritmo para producir su salida, se tiene una manera de cuantificar la eficiencia de los algoritmos independientemente del hardware en el que se ejecuta.

Estas operaciones o pasos deben ser simples. Ejemplo de estas operaciones pueden ser suma, resta, multiplicación, división, operaciones lógicas como AND, OR y NOT o asignar un valor a una variable.

NOTACIÓN O

Usualmente para representar el número de pasos no se utiliza un número como 7, 42 o 1337. En realidad, se representa en función del tamaño de la entrada, que, de nuevo, para algoritmos de búsqueda de camino corresponde al número de vértices y de aristas del grafo.

La notación utilizada para representar la eficiencia de un algoritmo se denomina notación O. La notación O es un análisis simplificado de dicha eficiencia en términos del tamaño de la entrada y el número de pasos.

Formalmente la notación *O* está definida de la siguiente manera:

Para una función dada *g(n)*, *O(g(n))* denota el *conjunto de funciones*

O(g(n)) = {*f(n)* : dónde hay constantes positivas *c* y n_0 tal que $0 \leq f(n) \leq cg(n)$ para toda $n \geq n_0$}

Una función *f(n)* pertenece al conjunto *O(g(n))* si existe una constante positiva *c* tal que el valor de *f(n)* sea siempre inferior al valor de *cg(n)* para una *n* lo suficientemente grande (a partir de una *n* mayor o igual a una constante n_0).

cg(n) establece una *cota superior asintótica* para *f(n)*. Por ejemplo, $f(n) \in O(n^2)$ indica que *f(n)* pertenece a un conjunto de funciones, las cuales sin importar que tan grande sean los valores de *n* los valores de *f(n)* no sobrepasan los valores de la función cn^2.

Cuando se dice que un algoritmo es $O(n^2)$ se refiere a que cuando el tamaño de la entrada es muy grande, el número de pasos no supera la función n^2. Es decir, como mucho el orden de crecimiento del algoritmo es cuadrático.

En análisis de algoritmos la notación O indica que se está considerando el peor caso del algoritmo. El peor caso se refiere a la instancia donde el algoritmo tomará más tiempo en ejecutarse.

Lo común es emplear el peor caso para describir el orden de crecimiento de un algoritmo debido a que brinda la cota superior del tiempo de ejecución para cualquier entrada. Con esa información se tiene la garantía de que sea cual sea la entrada, el algoritmo no tardará más tiempo que el peor caso.

En algunos algoritmos el peor caso se da de manera frecuente, por ejemplo, para buscar un elemento en una lista puede darse la situación de que tenga que revisarse la lista completa debido a que el elemento no está en la lista.

En la *Figura 6* se presenta una visualización de distintos órdenes de crecimiento.

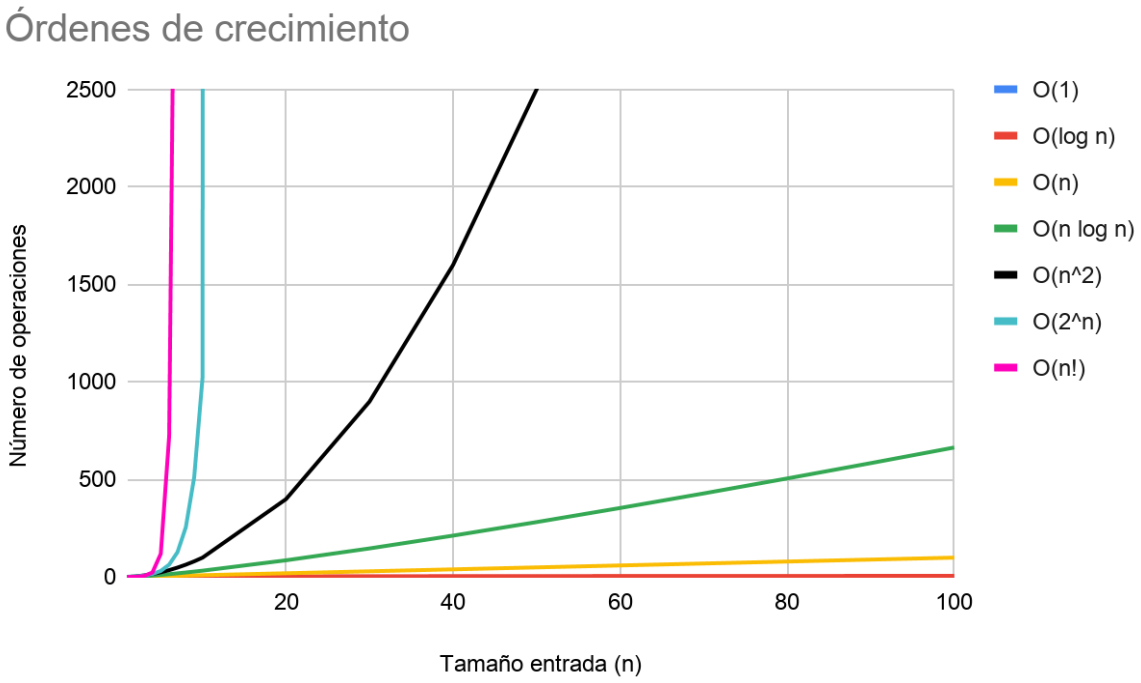


Figura 6. Visualización de distintos órdenes de crecimiento. Nótese que *O(1)* no se visualiza en el gráfico debido a que sus valores son muy pequeños. Un algoritmo con un orden de crecimiento grande, es un algoritmo lento, porque requiere mayor número de pasos para computar una solución.

En el gráfico se aprecia la relación que existe entre el número de operaciones y el tamaño de la entrada. Mayor número de operaciones denota que el algoritmo es más lento. Cuando se diseñan algoritmos se desea que tengan un orden de crecimiento pequeño.

O(1) significa que el orden de crecimiento es constante y es independiente del tamaño de la entrada. Eso quiere decir que no importa cuál es el tamaño de la entrada que se le asigna a estos algoritmos, siempre tardarán el mismo número de pasos en computar una solución. Este es el orden de crecimiento más pequeño, y en varios algoritmos, *O(1)* suele ser un sólo paso[28].

El orden de crecimiento *O(log n)* es el orden de crecimiento logarítmico. Cambiar la base de un logaritmo a otra, cambia el valor del logaritmo por un factor constante, por lo tanto, la base del logaritmo no es importante porque en notación O, los factores constantes se descartan[1]. Aunque los algoritmos con orden de crecimiento *O(log n)* son más lentos que *O(1)*, en la práctica la diferencia de tiempo es imperceptible, sobre todo en casos dónde el tamaño de la entrada es pequeño.

O(n) es crecimiento lineal y suele darse en casos dónde el código tiene un ciclo `for` o un ciclo `while`.

*O(nlog n)*es crecimiento linealítmico. Este tipo de crecimiento suele aparecer en algoritmos que utilizan una estrategia denominada “divide y vencerás” [1]. Es un poco más lento que el crecimiento lineal.

O(n^2) se refiere a crecimiento cuadrático. Se da en situaciones donde hay un ciclo dentro de un ciclo.

O(n^3) es crecimiento cúbico. Aparece en casos donde hay tres ciclos, uno dentro del otro.

O(2^n) y *O(n!)* corresponden a crecimiento exponencial y factorial respectivamente. Algoritmos de este tipo suelen ser bastante lentos, por lo que suelen evitarse si la entrada es grande. Se suele buscar alternativas más rápidas incluso si la solución que estas brindan es sólo una aproximación.

DEFINICIÓN DE GRAFO

Un grafo es una estructura conformada por un conjunto de vértices donde varios pares de estos vértices (o ninguno) están unidos por aristas. Típicamente se representan gráficamente. En la *Figura 7* se presenta un grafo con vértices *a, b, c, d, e* y *f* unidos por varias aristas.

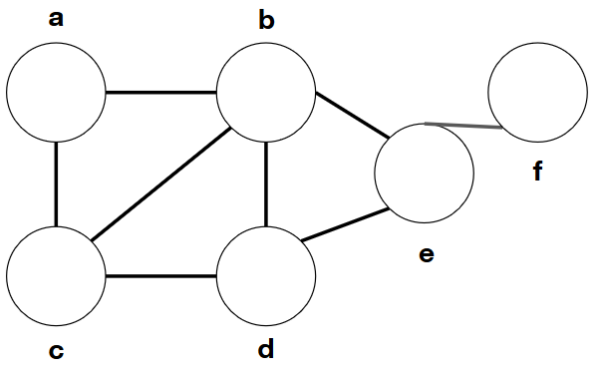


Figura 7. Grafo no dirigido con vértices *a, b, c, d, e* y *f*.

Algo a señalar es que un vértice puede tener *vértices vecinos*. Los vértices vecinos de un vértice dado son aquellos vértices que están conectados por una arista a dicho vértice. Por ejemplo, en la *Figura 7*, los vértices vecinos de *a*, corresponde a los *vértices b* y *c*. En cambio, los vértices vecinos de *e*, son *b*, *d* y *f*.

Una definición más formal desde un punto de vista matemático es la siguiente:

Un grafo es un objeto conformado por dos conjuntos, el *conjunto vértice* y el *conjunto arista*. El conjunto vértice es un conjunto finito no vacío. El conjunto arista puede estar vacío, y si no es así, entonces sus elementos son subconjuntos de dos elementos del conjunto vértice [29].

Ejemplos:

1. Los conjuntos {A, B, C, D, E} y {{A,B}, {A,C}, {A,D}, {B,D} {D,E}} constituyen un grafo.
2. Los conjuntos {U, V, W, X, Y, Z} y *ø* conforman un grafo. Este grafo en particular está constituido de sólo vértices sin unir.
3. Un grafo con el conjunto vértice {1, 2, 3, 4, 5} y el conjunto arista {{1,2}, {1,3}, {2,3}, {3,4}, {3,5}, {4,5}}.

4. El grafo de la *Figura 7* conformado por el conjunto vértice {a, b, c, d, e, f} y el conjunto arista {{a, b}, {a, c}, {b, c}, {b, d}, {b, e}, {c, d}, {d, e}, {e, f}}

Existen varios tipos de grafos. El grafo de la *Figura 7* se denomina ***grafo no dirigido*** debido a que las aristas que unen un par de vértices no tienen asociadas una dirección. En la *Figura 8a* se muestra un ***grafo dirigido***, que al contrario de un grafo no dirigido sus aristas tienen asociadas una dirección o sentido. En la *Figura 8b* se muestra un ***grafo dirigido ponderado***, que se distingue en que se asigna un valor o un peso a cada arista del grafo. Dicho costo podría interpretarse como la distancia para viajar de un vértice a otro.

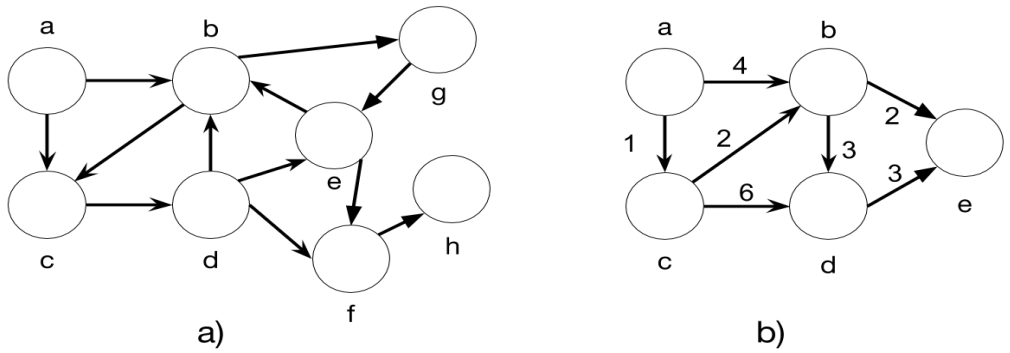


Figura 8. a) Grafo dirigido. b) Un grafo dirigido ponderado.

IMPLEMENTACIONES DE GRAFOS

Las dos maneras estándar de implementar grafos en una computadora es mediante ***listas de adyacencia*** y ***matriz adyacente*****.**

Un grafo implementado por ***listas de adyacencia*** es simplemente un arreglo de listas. Cada lista representa un vértice del grafo y sus elementos corresponden a los vértices a los que está unido un vértice en particular. Por ejemplo, en la *Figura 9b* la primera lista contiene los elementos *2* y *3*, es decir, el *vértice 1* está unido a los *vértices 2* y *3*. La segunda lista incluye los elementos *1, 3, 4, 5,* por consiguiente, el *vértice 2* está unido a los *vértices 1, 3, 4* y *5*. La cuarta lista tiene los elementos *3, 2* y *5*, indicando que el vértice 4 está unido a los *vértices 3, 2* y *5*.

La representación de listas de adyacencia puede adaptarse para grafos ponderados. Considere una arista *uv* que va del *vértice u* al *vértice v*, por lo tanto, se almacena el peso de dicha arista junto con el *vértice v* en la lista de adyacencia *u*.

Una potencial desventaja con la representación de listas de adyacencia es que es que para determinar si una arista dada *uv* está presente en el grafo, habría que buscar en la lista que corresponde al *vértice u* si alguno de sus elementos contiene el *vértice v*. Esto es algo que es remediado por la representación de matriz adyacente a costa de utilizar más memoria.

La representación por ***matriz adyacente*** consiste en utilizar matriz cuadrada *A* de $|V|$x$|V|$ donde $|V|$ representa el número de vértices en el grafo. Si el elemento A_{ij} es 1, el grafo contiene una arista que va del *vértice i* hacia el *vértice j*. Si A_{ij} es 0, no hay una arista que vaya del *vértice i* al *vértice j* (*Figura 9c*). Para grafos no dirigidos la matriz es simétrica, en otras palabras $A_{ij} = A_{ji}$.

Para grafos ponderados, el peso w_{ij} de la arista que va del vértice *i* al vértice *j* se representa asignando $A_{ij} = w_{ij}$.

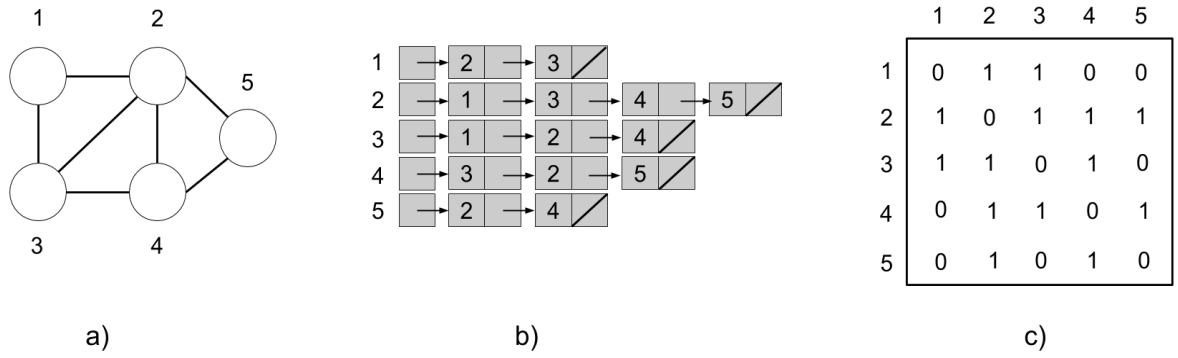


Figura 9. a) Representación de un grafo de manera gráfica. b) Representación del mismo grafo por listas de adyacencia. Nótese que el *vértice 2* está unido a todos los otros vértices y el *vértice 5* está unido sólo al *vértice 2* y *4*. c) Representación por matriz de adyacencia. Dado que el grafo es no dirigido, la matriz es simétrica (figura adaptada de *Introduction to Algorithms 3rd Edition*, Thomas H. Cormen p.590 [1]).

La implementación de matriz adyacente para un grafo requiere $O(|V|^2)$ de memoria dónde $|V|$ es el número de vértices del grafo; independientemente del número de aristas. Sin embargo, aunque la representación por listas de adyacencia es más compacta y asintóticamente requiere menos memoria, la matriz adyacente es más simple. Considerándose estas ventajas y desventajas, se pueden designar unas cuantas pautas generales:

Cuando se implementa un ***grafo disperso***, es decir, cuando el grafo tiene un número de aristas lejano a $|V|^2$, lo mejor es utilizar listas de adyacencia. Por otro lado, cuando el grafo tiene un número de aristas cercano a $|V|^2$, se dice que es un ***grafo denso*** y conviene utilizar la representación de matriz de adyacencia.

REPRESENTACIÓN POR CUADRÍCULA

Una forma de representar mapas o mundos utilizando grafos es a través de la ***representación por cuadrícula***. Dicha cuadrícula está conformada por un conjunto de celdas ordenadas de dos tipos: aquellas en las que se puede transitar (celdas de color claro en la *Figura 10*) y aquellas que son obstáculos (celdas oscuras). En la cuadrícula se permite moverse en 8 direcciones: 2 horizontales, 2 verticales y 4 diagonales.

En algunas implementaciones no se permite movimiento de manera diagonal, por lo que el movimiento está restringido a sólo 4 direcciones, 2 horizontales y 2 verticales, pero para este trabajo se utiliza la implementación que permite movimiento diagonal.

El costo de moverse de manera horizontal o vertical es 1, mientras que moverse de manera diagonal es $\sqrt{2}$.

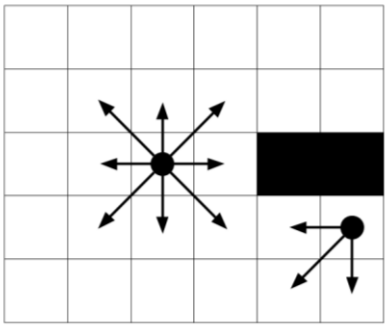


Figura 10. Representación de un mapa utilizando la abstracción de cuadrícula. En dicha representación es posible realizar movimiento en ocho direcciones (horizontal, vertical y diagonal). Las celdas oscuras representan obstáculos provocando que sólo pueda realizarse movimiento hacia sólo tres direcciones desde el punto marcado.

Esta representación de mapas puede ser implementada utilizando grafos designando cada celda transitable como un vértice en un grafo conectado por aristas a otros vértices que también simbolizan celdas transitables. Por ejemplo, en la *Figura 10* se muestra una celda desde la que se puede viajar a una de las ocho direcciones, lo cual

traducido a un grafo es un vértice conectado a otros 8 vértices mediante aristas. Cuatro de esas aristas tienen un peso de $\sqrt{2}$ representando movimiento diagonal, mientras que el resto tienen un peso de 1, representando el movimiento vertical y horizontal.

En la *Figura 10* también se denota una celda en la que sólo es posible movimiento hacia tres direcciones debido a los obstáculos. Trasladado a un grafo esto es un vértice conectado a otros tres vértices mediante aristas. Dos de estas aristas poseen un peso de 1 simbolizando el movimiento vertical hacia abajo y el movimiento horizontal hacia la izquierda, mientras que la arista restante posee un peso de $\sqrt{2}$ simbolizando movimiento diagonal.

Una limitante para este tipo de representación es la restricción de movimiento en 8 direcciones. En la realidad el movimiento no está restringido a sólo 8 direcciones, dando lugar a que las soluciones computadas brinden caminos no óptimos. Un ejemplo se muestra en la *Figura 11*.

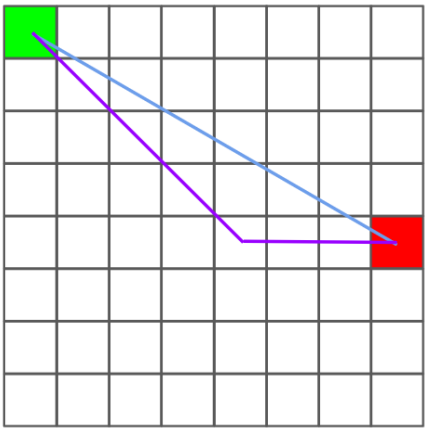


Figura 11. Representación de un mapa utilizando la abstracción de cuadrícula. La línea azul describe el camino óptimo para dirigirse de la celda verde hacia la celda roja, sin embargo, la representación por cuadrícula restringe la manera en la que se realiza el movimiento hacia sólo 8 direcciones. El camino óptimo considerando dicha restricción se describe con la línea morada.

La representación por cuadrícula también no es muy buena representando entornos con varios anchos. El problema recae en la resolución de las celdas; muy similar a como imágenes son representadas con píxeles. Si se desea representar un mapa de manera detallada sólo hay que utilizar mayor número de celdas de menor tamaño. Pero eso es a costa de aumentar el número de vértices del grafo que representa la

cuadrícula y por lo tanto incrementar el tiempo de cómputo para que el algoritmo provea una solución. Por consiguiente, para lograr que los algoritmos de búsqueda de camino funcionen lo más rápido posible, se debe aumentar el tamaño de cada celda sin perder detalles importantes del mapa.

COLA DE PRIORIDAD

Varios algoritmos de búsqueda de camino utilizan una estructura de datos denominada ***cola de prioridad***. Una *cola de prioridad* está constituida de elementos que tienen asociados una prioridad. Para los algoritmos de búsqueda de camino la *cola de prioridad* debe tener al menos tres operaciones:

- Añadir elemento a la cola
- Extraer elemento con mayor prioridad
- Cambiar valor de un elemento

Por ejemplo, considere la *cola de prioridad* constituida por los elementos {10, 17, 2, 8, 3, 9, 0, 7}, dónde los elementos con mayor prioridad son aquellos que tienen el valor más pequeño. Añadir el elemento “4” a la lista desordenada provocaría que dicha lista sea: {10, 17, 2, 8, 3, 9, 0, 7, 4}.

Extraer el elemento con mayor prioridad de esta última lista, dejaría la lista en el siguiente estado: {10, 17, 2, 8, 3, 9, 7, 4} debido a que 0 es el valor más pequeño.

Finalmente, si se desea cambiar el valor del segundo elemento por “1” la lista sería: {10, 1, 2, 8, 3, 9, 7, 4}

Cada una de estas operaciones tiene una complejidad temporal. Para el caso de una *cola de prioridad* implementada con una lista desordenada, las operaciones añadir, extraer y cambiar valor son *O(1)*, *O(n)* y *O(1)* respectivamente.

Existen otras implementaciones de *colas de prioridad* que utilizan estructuras de datos denominadas *montículos binarios* [30]. En la *Tabla 1* se enlista la complejidad temporal de las operaciones de una *cola de prioridad* utilizando distintas estructuras de datos.

	Lista desordenada	Montículo binario
Insertar	*O(1)*	*O(log n)*
Extraer	*O(n)*	*O(log n)*
Cambiar valor	*O(1)*	*O(log n)*

Tabla 1. Complejidad temporal para distintas operaciones de una *cola de prioridad* con *n* elementos utilizando una lista desordenada y un montículo binario.

Las *colas de prioridad* que utilizan montículos binarios parecen ser más rápidas que las listas desordenadas. Para verificar si es el caso de manera práctica, se realizaron unas cuantas pruebas iniciales. Se confirmó que utilizar montículos binarios es más rápido y por lo tanto para el resto de experimentos se utilizó esta implementación.

ALGORITMOS DE BÚSQUEDA DE CAMINO

ALGORITMO DE DIJKSTRA

El algoritmo de Dijkstra fue concebido en 1956 y publicado tres años después por Edsger Wybe Dijkstra en su artículo *A note on two problems in connexion with graphs*[31]. El algoritmo resuelve el problema del camino más corto entre dos vértices en un grafo. En el *Anexo* se incluye el pseudocódigo del algoritmo.

Dijkstra utiliza dos listas denominadas: ***lista abierta*** y ***lista cerrada*****,** dónde ésta primera está implementada con una *cola de prioridad*. El propósito de estas listas es llevar un registro de los ***valores G*** de cada vértice.

El *valor G* de un vértice es la distancia o el costo más pequeño para ir del mismo hacia el vértice de partida. Los *valores G* de los vértices dentro de la *lista abierta* son sólo estimados, mientras que los que están en la *lista cerrada* son correctos.

Cada vértice tiene asociado un *valor G*, pero para aquellos vértices que no están dentro de la *lista abierta* o *lista cerrada* sus valores se estiman como *indefinido*. En algunos se denota con el valor ∞ (infinito) o un número muy grande [1].

A continuación, se ilustra paso a paso una instancia de la ejecución del algoritmo de Dijkstra. En dicha instancia, se desea encontrar el camino más corto desde el *vértice a* hacia el *vértice e*.

Se comienza con un grafo donde el estimado de los *valores G* de todos los vértices es *indefinido* y las dos listas están vacías como se ilustra en la *Figura 12*.

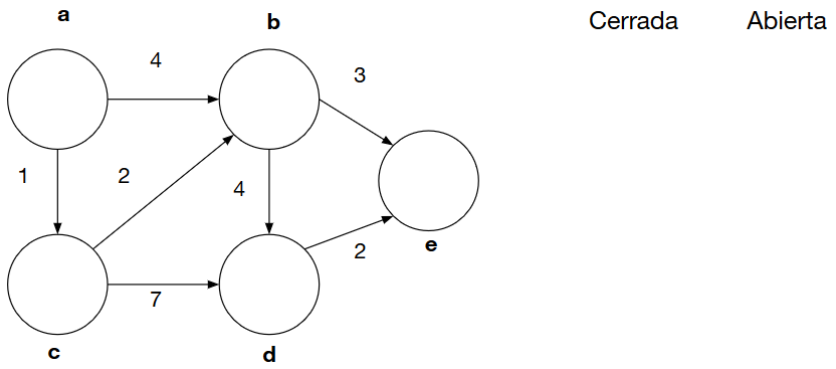


Figura 12. Visualización del algoritmo de Dijkstra. En su primera iteración la lista cerrada y abierta están vacías y el *valor G* asociado a cada vértice es indefinido.

El primer paso es colocar el vértice inicial (que para este caso es el *vértice a*) en la *lista cerrada* y designar su *valor G* como 0, debido a que es el vértice inicial. En la *Figura 13* el número dentro de los vértices representan su *valor G* asociado o un estimado. Los vértices coloreados de gris denotan que están en la lista cerrada.

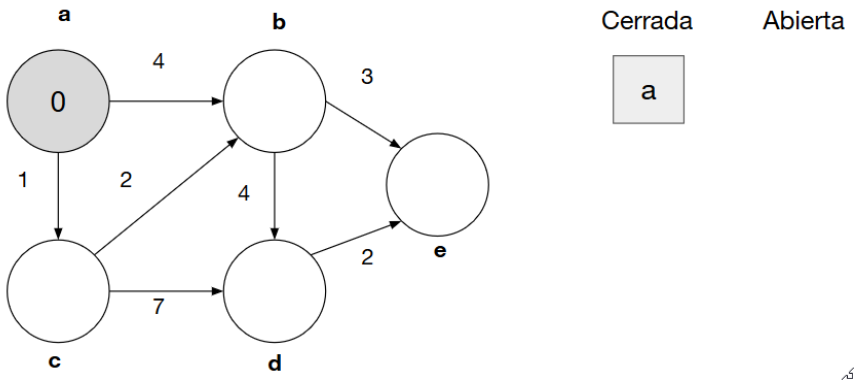


Figura 13. Visualización del algoritmo de Dijkstra. El siguiente paso es añadir el vértice inicial a la lista cerrada y asociarle un *valor G* de 0. El número dentro del vértice indicado su *valor G*. De color gris se denota que el vértice ha sido movido a lista cerrada. Los números en cada arista indican su costo y las letras el nombre del vértice.

A continuación, se añaden todos los vecinos del *vértice a* (*vértice b* y *vértice c*) a la *lista abierta* y se estiman sus *valores G*. En la *Figura 14* se muestran los vértices en la *lista abierta* coloreados de azul con sus respectivos *valores G* estimados.

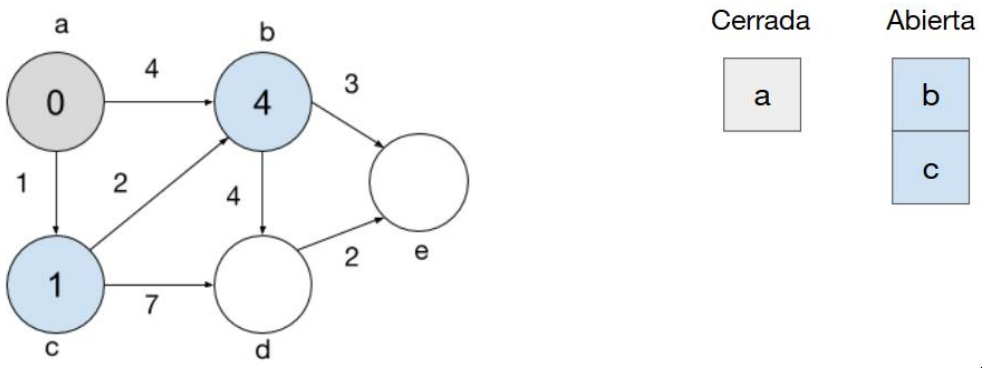


Figura 14. Visualización del algoritmo de Dijkstra, los vértices vecinos del último vértice en la lista cerrada se añaden a la lista abierta. Los números dentro de cada vértice denotan su *valor G* asociado. El color azul indica que el vértice se encuentra en la lista abierta y en gris que el vértice está en la lista cerrada. Los números en cada arista indican su costo y las letras el nombre del vértice.

El *valor G* estimado del *vértice b* es 4, porque el costo más pequeño para ir al *vértice a* desde *b* es 4, mientras que para *c* el *valor G* estimado es 1.

Algo a destacar es que la operación de añadir un vértice con su *valor G* estimado a la lista abierta (una cola de prioridad) toma *O(1)* si la lista es una lista desordenada. En cambio, si la *lista abierta* está representada con un *montículo binario*, la operación toma *O(log n)*.

Posteriormente se extrae el elemento con el *valor G* más pequeño de la *lista abierta* y se mueve a la *lista cerrada*. Esta operación de *extraer un elemento* con mayor prioridad toma *O(n)* en una lista desordenada mientras que para un montículo binario toma *O(log n)*.

Para este caso el elemento con el *valor G* más pequeño en la *lista abierta* es el *vértice c*. El vértice se coloca en la *lista cerrada*. Después se añaden todos los vecinos de *c* a la *lista abierta* (*vértice b* y *d*) y se estiman sus *valores G*.

Debido a que el *vértice b* ya se encuentra en la *lista abierta*, sólo es necesario recalcular su *valor G*. Esta vez el camino para ir hacia el vértice inicial debe pasar por el *vértice c*. De esta manera el nuevo valor estimado de *b* es 3. Si se compara con el

valor anterior de 4, el nuevo estimado es un valor más pequeño. Este cambio se le denomina ***relajación del valor G*.**

Relajar un valor G es equivalente a cambiar un valor en la lista abierta. La operación de cambiar un *valor G* es *O(1)* si la *lista abierta* es una lista desordenada y O(*log n*) si la lista es un montículo binario.

Por otro lado, el *vértice d* es añadido a la *lista abierta,* y se estima su *valor G*. Su *valor G* estimado es 8. En la *Figura 15* se ilustra el resultado de estos pasos.

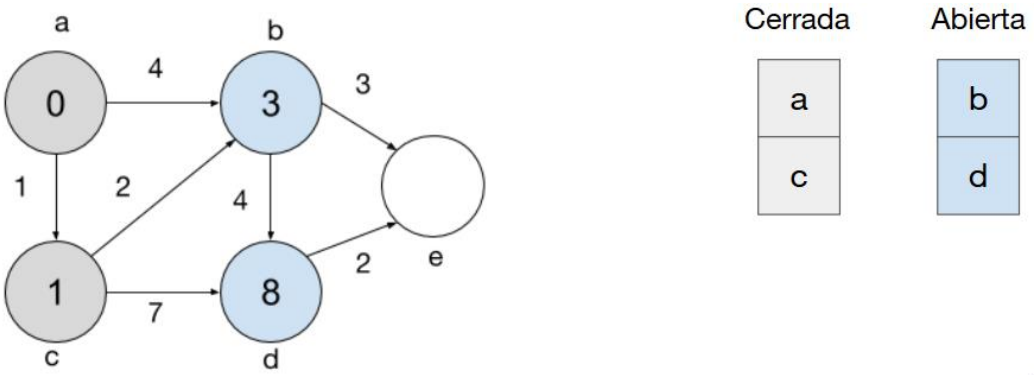


Figura 15. Visualización del algoritmo de Dijkstra. Los números dentro de cada vértice denotan su *valor G* asociado. El color azul indica que el vértice se encuentra en la lista abierta y en gris que el vértice está en la lista cerrada. Los números en cada arista indican su costo y las letras el nombre del vértice. Nótese que el *valor G* estimado del *vértice b* ha cambiado de 4 a 3. El *vértice d* es añadido a la *lista abierta*, y se estima que su *valor G* es 8.

Después, se extrae el vértice con el *valor G* más pequeño en la *lista abierta* y se mueve a la *lista cerrada*. En esta ocasión es el *vértice b*.

Los vecinos del *vértice b* (*vértices d* y *e*) se añaden a la *lista abierta* y se estima el *valor G* de cada uno. El *vértice d* ya estaba en la *lista abierta*, por lo tanto, se vuelve a estimar su *valor G* dónde el camino hacia el vértice inicial (*vértice a*) pase primero por el *vértice b* y luego *c*. Así, el nuevo *valor G* estimado de *d* es 7.

En el caso del *vértice e* su *valor G* estimado es 6. En la *Figura 16* se ilustra el resultado de seguir estos pasos.

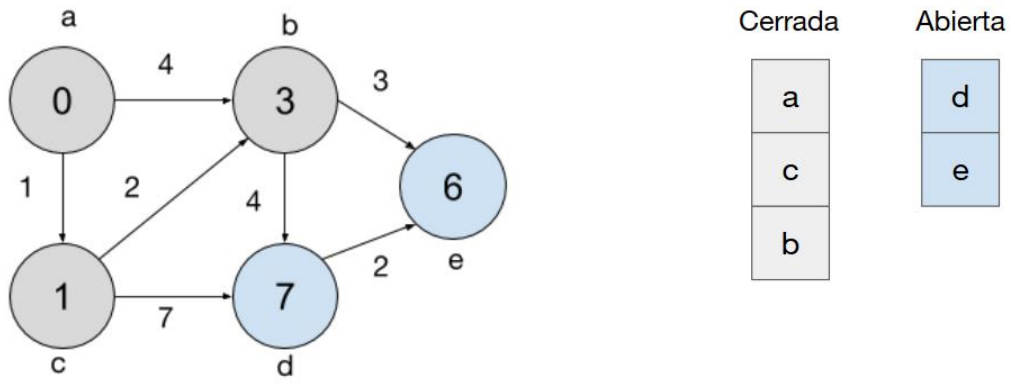


Figura 16. Visualización del algoritmo de Dijkstra. Los números dentro de cada vértice denotan su *valor G* asociado. El color azul indica que el vértice se encuentra en la lista abierta y en gris que el vértice está en la lista cerrada. Los números en cada arista indican su costo y las letras el nombre del vértice. El *vértice b* es añadido a la *lista cerrada* y el *vértice d* y *e* a la *lista abierta*, sus *valores G* estimados son respectivamente 7 y 6.

Finalmente se coloca el *vértice e* en la *lista cerrada* debido a que tiene el *valor G* estimado más pequeño en la *lista abierta*. Una vez que el vértice objetivo es añadido a la *lista cerrada*, se recorre el camino de regreso hacia el vértice inicial. El camino generado garantiza ser el camino más corto entre el vértice inicial y el objetivo. En la *Figura 17* se ilustra las aristas a recorrer marcadas en rojo, para transitar el camino más corto.

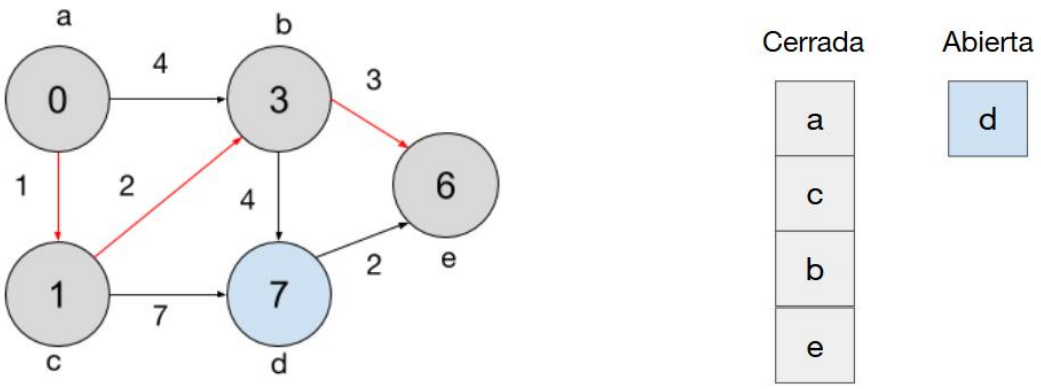


Figura 17. Visualización del algoritmo de Dijkstra. Los números dentro de cada vértice denotan su *valor G* asociado. El color azul indica que el vértice se encuentra en la lista abierta y en gris que el vértice está en la lista cerrada. Los números en cada arista indican su costo y las letras el nombre del vértice. El camino más corto entre el *vértice a* y *e* está conformado por las aristas marcadas en rojo.

Resumiendo, el funcionamiento del algoritmo; se utilizan dos listas llamadas *lista cerrada* y *lista abierta*. En la lista cerrada se almacenan los vértices cuyo *valor G* es conocido, mientras que en la lista abierta se añaden aquellos en los que sólo se tiene

un estimado. El *valor G* de un vértice es el costo de ir de dicho vértice hacia el vértice de inicio siguiendo el camino más corto. Inicialmente se añade el vértice inicial a la *lista cerrada* y se le asigna un *valor G* de 0. Después se añaden sus vecinos en la *lista abierta*, se estiman sus *valores G*, se extrae aquel vértice con el valor más pequeño y se coloca en la *lista cerrada*. A continuación, se añaden los vecinos de este vértice a la *lista abierta* y se repiten los pasos anteriores hasta añadir el vértice objetivo a la *lista cerrada*. Si alguno de los vecinos ya se encontraba en la *lista abierta* simplemente se vuelve a estimar su *valor G*. Cuando el vértice objetivo es añadido a la *lista cerrada* se recorre el camino de regreso hacia el vértice inicial. El camino recorrido está garantizado ser el óptimo.

ANÁLISIS DEL ALGORITMO DE DIJKSTRA

En el peor caso, el algoritmo de Dijkstra tendría que realizar la operación *extraer elemento* para todos los vértices del grafo y la operación *cambiar valor* para cada arista del grafo. Por lo tanto la complejidad temporal del algoritmo de Dijsktra es: $O(|V| * T_e + |E| * T_{cv})$.

Dónde $|V|$ es el número total de vértices en el grafo, $|E|$ el número de aristas en el grafo, T_e es la complejidad temporal de la operación *extraer elemento* y T_{cv} es la complejidad temporal de la operación *cambiar valor*.

Por lo tanto, si se utiliza la implementación de Dijkstra donde la *lista abierta* es una *cola de prioridad* representada con una lista desordenada, la complejidad temporal del algoritmo de Dijkstra es: $O(|V|^2 + |E|)$[1].

En cambio, para la implementación con montículo binario la complejidad temporal es: $O(|V|log|V| + |E|log|V|)$[1].

VENTAJAS Y DESVENTAJAS

La ventaja del algoritmo de Dijkstra es que establece la estrategia principal para encontrar el camino más corto entre dos vértices y garantiza encontrar el camino óptimo. Varios de los otros algoritmos que se mencionan en este texto han sido diseñados a partir del algoritmo de Dijkstra.

La desventaja principal, es que debido a que Dijkstra no realiza ningún tipo de suposición respecto a la estructura del grafo (a excepción de que se supone que las aristas tienen costo positivo), termina explorando vértices de más comparado con el resto de algoritmos. Los grafos generados por la representación de cuadrícula tienen un arreglo ordenado y predecible y esta información puede utilizarse para crear algoritmos más eficientes.

ALGORITMO A*

A* (pronunciado *A estrella*) es un algoritmo diseñado en 1968 por un grupo de investigadores en Stanford Research Institute[32]. Dicho algoritmo es una extensión del algoritmo de Dijkstra y su propósito es mejorar el tiempo de ejecución utilizando funciones heurísticas.

La estructura de este algoritmo es exactamente la misma con respecto a Dijkstra. Se coloca como entrada un vértice inicial, un vértice objetivo y un grafo, se utiliza una *lista cerrada* y una *lista abierta* y en cada iteración se mueve un vértice de la *lista abierta* hacia la *lista cerrada*. Sin embargo, a diferencia de Dijkstra que selecciona los vértices con los *valores G* más pequeños, A* elige los vértices con los *valores F* más pequeños.

El *valor F* de un vértice es su *valor G* sumado con su *valor H*. El *valor H* de un vértice es un *estimado* del costo para ir de dicho vértice hacia el vértice objetivo tomando el camino óptimo.

Para calcular el *valor H* de un vértice se utiliza una función heurística. Cabe recalcar que el *valor H* de un vértice no necesariamente debe ser *exactamente* la distancia entre dicho vértice y el objetivo. El *valor H* de un vértice es sólo un estimado.

En resumen:

- El *valor G* de un vértice es el costo para ir de dicho vértice hacia el vértice inicial tomando el camino más corto.
- El *valor H* de un vértice del costo para dirigirse desde dicho vértice hacia el vértice objetivo tomando el camino más corto.
- El *valor F* de un vértice es la suma del *valor G* y el *valor H*.

De nuevo, el algoritmo A* elige el vértice con el *valor F* más pequeño de la *lista abierta* y lo coloca en la *lista cerrada*. La eficiencia del algoritmo A* varía según la función heurística utilizada. Mientras más exacto sea el estimado del *valor H* para cada vértice, mejor será el tiempo de ejecución de A*.

DEFINICIÓN DE UNA HEURÍSTICA

Una heurística es una técnica para resolver problemas cuando el utilizar métodos convencionales es demasiado lento. También puede utilizarse para hallar una solución aproximada a un problema cuando no es posible encontrar una solución exacta [33].

El objetivo de una heurística es producir una solución en un tiempo razonable que sea lo suficientemente buena para resolver un problema dado. La solución dada puede no ser la mejor para solucionar el problema, o sólo una aproximación, pero en varios casos es preferible utilizarla porque no requiere una cantidad de tiempo irrazonable en computarse.

REQUERIMIENTOS DE LA FUNCIÓN HEURÍSTICA

Para que el algoritmo A* funcione de manera óptima, la función heurística debe tener dos propiedades: ***admisibilidad*** y ***consistencia*** [34].

Se dice que una función heurística es *admisible*, cuando dicha función nunca sobreestima el *valor H* de un vértice.

Considere una función heurística h y un vértice cualquiera v. Entonces $h(v)$ es un estimado heurístico del costo para dirigirse desde el vértice v hacia el vértice objetivo tomando el camino más corto. Considere también $c(v)$, que es el costo real para transitar desde el vértice v al vértice objetivo tomando el camino óptimo. Entonces se dice que h es admisible si y sólo si para todo vértice $v \in V$:

$$h(v) \leq c(v)$$

En otras palabras, la función h nunca sobreestima el costo real para ir del *vértice* v al vértice objetivo tomando el camino óptimo.

Si deliberadamente se diseña una función heurística que estime que el *valor H* es 0 para todos los vértices, entonces el *valor F* es igual al *valor G*:

$$F = G + H = G + 0 = G$$

y de esta el algoritmo A* se convierte en el algoritmo de Dijkstra.

Para el caso de que h siempre estima correctamente el *valor H* para cada vértice, el algoritmo siempre expande su búsqueda hacia los vértices que conforman el camino más corto.

Por otro lado, si la función h sobreestima el *valor H* de cada vértice, A* se ejecuta con mayor velocidad, pero no garantiza que el camino encontrado sea el más corto.

Para la propiedad de consistencia considere un vértice cualquiera v y su vértice vecino n. También considere una función d que calcula el costo entre un vértice y alguno de sus vértices vecinos. Entonces se dice que la función h es consistente si y sólo si:

1. $h(objetivo) = 0$ y
2. $h(v) \leq d(v,n) + h(n)$

Para todo vértice $v \in V$ y para todo $n \in vecinos(v)$. $vecinos(v)$ es el conjunto de todos los vértices aledaños al vértice v.

Otra forma de formular lo anterior es utilizando una representación visual. Observe la *Figura 18*.

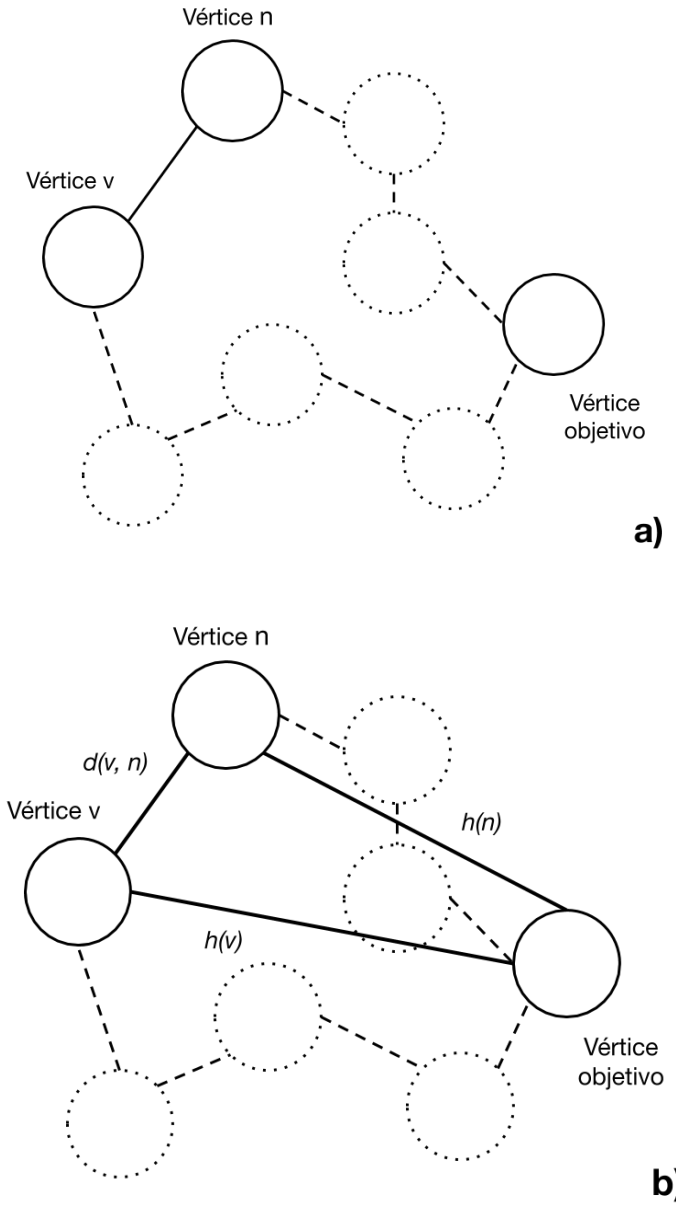


Figura 18. a) Un grafo con un vértice *v* y un vértice vecino *n*. Las aristas y vértices punteados representan el camino más corto que llevan al vértice objetivo. No se conoce el costo y la estructura de ese camino. b) La arista en negrita que une el vértice v y n tiene un costo de $d(v,n)$. La línea que va del vértice n hacia el vértice objetivo representa el costo estimado para llegar al objetivo. De igual manera para la línea que va del vértice v al vértice objetivo. Respectivamente los costos estimados son $h(n)$ y $h(v)$. El costo real es desconocido.

En la *Figura 18a* se presenta un grafo con un *vértice* v, su vértice vecino n y el vértice objetivo. Las aristas y los vértices punteados representan el camino con menor costo para llegar al vértice objetivo. El costo de esos trayectos y su forma es desconocido.

En la *Figura 18b*, la arista marcada en negrita que va del vértice v al vértice n tiene un costo de $d(v, n)$.

Como no es conocido el costo real para ir del vértice n al vértice objetivo, no queda de otra más que estimarlo. Por lo tanto, se traza una línea que va del *vértice* n al vértice objetivo y se le asigna un costo de $h(n)$ (el estimado de la función heurística h para el *vértice* n).

Del mismo modo para el *vértice* v, se traza una línea que va del mismo hacia al vértice objetivo y se le asigna un costo de $h(v)$. Nótese que las líneas en negrita forman un triángulo.

En matemáticas, la desigualdad del triángulo indica que para todo triángulo la suma de las longitudes de dos lados cualquiera debe ser mayor o igual a la longitud del lado restante. Si se toma la suma de las longitudes $d(v, n) + h(n)$, entonces para que la desigualdad del triángulo se cumpla $h(v) \leq d(v, n) + h(n)$.

Básicamente una de las condiciones de la propiedad de consistencia es que se cumpla la desigualdad del triángulo.

La otra condición para que la función heurística h sea consistente es que $h(objetivo) = 0$ debido a que se conoce con certeza que el costo para ir del vértice objetivo hacia a sí mismo es 0.

Para que el algoritmo A* garantice que su solución es el camino óptimo, únicamente es requerido que su función heurística satisfaga la condición de admisibilidad. Sin embargo, si se cumple también la propiedad de consistencia, su tiempo de ejecución mejora al explorar menos vértices [35].

Un detalle a señalar es que, si la función heurística es consistente, necesariamente es admisible. En cambio, si una función heurística es admisible no necesariamente es consistente.

FUNCIONES HEURÍSTICAS PARA A*

A* utiliza funciones heurísticas para estimar el costo de ir desde un vértice dado hacia un vértice objetivo utilizando el camino óptimo. Para estimar este costo, existen tres funciones heurísticas bastante comunes: *Distancia Euclidiana*, *distancia de Manhattan* y *distancia Octile*.

Distancia Euclidiana

La distancia Euclidiana es la longitud que hay de un punto a otro trazando una recta que pase por dichos puntos.

Distancia de Manhattan

A diferencia de la distancia Euclidiana que traza una línea en cualquier dirección, la distancia de Manhattan sólo permite movimientos en vertical y horizontal. La suma de la longitud de estas líneas es la distancia de Manhattan.

Distancia Octile

La distancia Octile permite movimientos en horizontal, vertical y diagonal. El costo de moverse de manera diagonal es $\sqrt{2}$, mientras que el costo de desplazarse de manera horizontal o vertical es 1. La distancia octile primero traza una línea recta en dirección al objetivo. Cuando la línea termina en la misma columna o fila que el objetivo de inmediato se traza otra línea vertical u horizontal hacia el objetivo.

En la *Figura 19.* se presenta una comparación entre la distancia Euclidiana, la distancia de Manhattan y la distancia Octile.

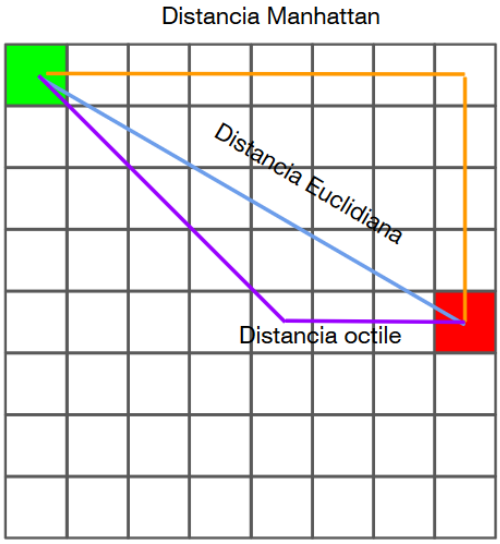


Figura 19. Comparación de la distancia Euclidiana (azul), Manhattan (naranja) y octile (morado) entre la celda verde y la celda roja. La distancia Euclidiana es una línea recta de una celda a otra, la distancia de Manhattan únicamente permite desplazamiento en horizontal y vertical y la distancia Octile inicialmente realiza movimiento diagonal hasta llegar a la misma fila o columna que el objetivo, después sólo se realiza movimiento en horizontal o vertical.

En general el uso de la de distancia de Manhattan como función heurística causa que el algoritmo A* sea más eficiente comparado con la distancia Euclidiana[36]. La distancia de Manhattan sólo es una función heurística admisible en representaciones de cuadrícula que sólo permite movimiento en horizontal y vertical. En implementaciones que incluyen movimiento diagonal la distancia de Manhattan puede sobreestimar el *valor H* de un vértice, no garantizándose el camino más corto.

Por esa razón es preferible utilizar la distancia octile en implementaciones que permiten movimiento diagonal y la distancia de Manhattan en aquellas donde sólo se permite movimiento vertical y horizontal.

VENTAJAS Y DESVENTAJAS

El algoritmo A* es una mejora respecto al algoritmo de Dijkstra. Gracias al uso de funciones heurísticas A* expande hacia las celdas que se encuentran más cercanas al objetivo. De esta manera se explora menor número de celdas, realizando menos operaciones en la *lista abierta* y disminuyendo el uso de memoria.

Sin embargo, en algunos mapas donde el camino óptimo tiene varios giros bruscos que se alejan del objetivo, la heurística se vuelve ineficiente y el algoritmo tiende a

comportarse como el algoritmo de Dijkstra. En realidad, A* no mejora su complejidad temporal con respecto a Dijkstra en el peor caso. Sin embargo, en términos prácticos A* suele ser más eficiente en la mayoría de los casos.

ALGORITMO JUMP POINT SEARCH (JPS)

El algoritmo Jump Point Search es un algoritmo diseñado recientemente en 2011 por Daniel Harabor y Al-ban Grastien en The Australian National University[37].

En su artículo Harabor y Grastien identifican que en las representaciones de cuadrícula se da el caso común de que existen varios caminos óptimos que conducen desde un punto de partida hacia el objetivo. Un ejemplo se muestra en la *Figura 20*.

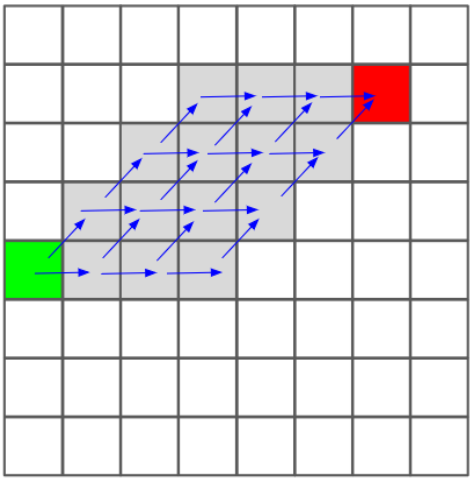


Figura 20. Varios caminos óptimos con la misma longitud que van de la celda verde hacia la celda roja en un mapa representado por la abstracción de cuadrícula. Dicho conjunto de caminos se les conoce como caminos simétricos. Figura adaptada de *zerowidth* [38].

En dicha figura se presentan varios caminos óptimos que van desde el punto de partida hacia el objetivo. Esencialmente todos los caminos son los mismos y tienen la misma longitud. La única diferencia es el orden en el que se da el desplazamiento vertical, horizontal y diagonal. En su artículo de investigación Harabor y Granstein se refieren a estos caminos como ***caminos simétricos***.

JPS utiliza una serie de reglas para que en situaciones donde existen caminos simétricos se identifique sólo uno y no todos ahorrando tiempo de cómputo.

La implementación JPS está basada a partir del algoritmo A*, el cual selecciona el vértice con el *valor F* más pequeño y expande su búsqueda añadiendo **todos** los vecinos a la *lista abierta*. Pero JPS va más allá expandiendo de manera inteligente [37]. A continuación, se describen las pautas que utiliza JPS para expandir su búsqueda.

Considere la *Figura 21*. La celda azul es la celda desde la que se desea expandir y la celda verde claro es la celda desde la cual se expandió anteriormente. A esta celda se le llama celda padre.

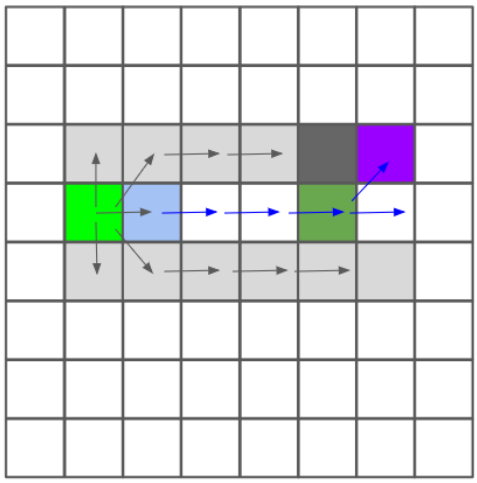


Figura 21. JPS hallando los sucesores de la celda azul. La celda verde llamada *celda padre*, es la celda desde la que se expandió anteriormente. Las celdas marcadas en gris son celdas que se pueden alcanzar de manera óptima desde la celda padre siguiendo el camino marcado por las flechas grises. Considerando el análisis de JPS, las únicas celdas a considerar como sucesores para la celda azul claro son aquellas en frente de ésta. Se inspecciona cada celda en frente hasta encontrar una celda (verde oscuro) que tenga un vecino forzado (celda morada). Figura adaptada de *zerowidth* [38].

Primero, para expandir la celda azul, se puede comenzar ignorando la celda padre, porque es una celda que ya ha sido explorada. También se pueden ignorar las celdas diagonales, detrás de la celda azul porque es más eficiente acceder a ellas desde la celda padre. Lo mismo se puede decir respecto las celdas que están arriba y abajo de la celda azul. Las celdas diagonales en frente de la celda azul, también pueden ignorarse porque se puede preferir tomar el camino que pasa a través de las celdas arriba y abajo de la celda azul.

De esta manera sólo hay un posible sucesor; la celda a la derecha de la celda azul. Pero si se aplican las mismas reglas a esta celda que está a la derecha, se puede

saltar hacia otra celda más a la derecha. Esto se repite de manera indefinida siempre y cuando el camino esté despejado. De esta manera se pueden saltar celdas sin tener que añadirlas oficialmente a la *lista abierta*.

Sin embargo, la celda verde oscuro tiene un obstáculo justo arriba. Normalmente no es necesario revisar las celdas diagonales al frente, pero debido al obstáculo se vuelve forzoso tener que revisar la celda diagonal que se muestra en la figura de color morado y también la celda a la derecha. A esta celda de color morado se le denomina *vecino forzado*.

Cuando se encuentra un *vecino forzado* se deja de saltar a la derecha y se añade la celda actual a la *lista abierta*. En este caso el sucesor de la celda azul es la celda verde oscuro.

Otro caso a considerar es cuando el obstáculo está directamente en frente del salto, como se ilustra en la *Figura 22*.

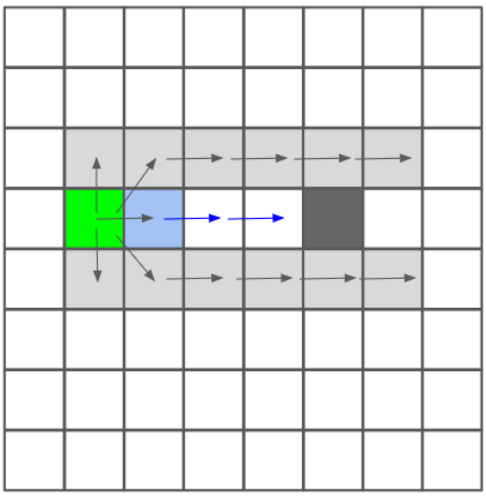


Figura 22. JPS intentando ubicar un punto de salto para la celda en azul, cuando hay un obstáculo en frente (celda oscura). La celda verde es la celda padre. Lo preferible es tomar los caminos que pasen por las celdas ubicadas arriba y abajo de la celda azul. Figura adaptada de *zerowidth* [38].

En esta situación el salto se descarta completamente debido a que se supone que es preferible tomar otro camino que pase por las celdas ubicadas arriba y abajo de la celda azul.

Estas reglas pueden extenderse para movimiento en las otras dos direcciones verticales y la otra dirección horizontal. Pero para el movimiento diagonal las pautas son un tanto distintas.

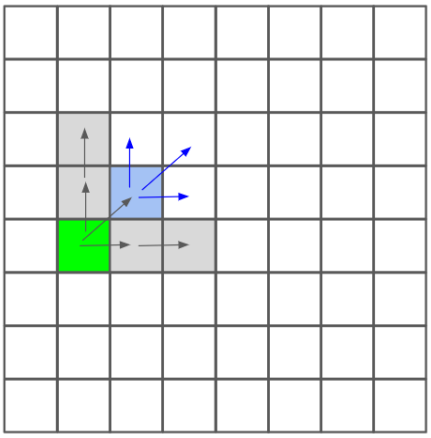


Figura 23. JPS buscando sucesores para la celda azul cuando la celda padre (marcada en verde) está detrás de manera diagonal. Las celdas a considerar están en dirección vertical, horizontal y diagonal indicadas por las flechas en azul. Figura adaptada de *zerowidth* [38].

En la *Figura 23*, se muestra una celda marcada de color azul, con su celda padre detrás de manera diagonal. Observe que 5 de sus vecinos pueden ser ignorados. Por otro lado, se puede llegar a las celdas que están arriba y a la derecha por otros caminos, pero se prefiere pasar a través de la celda azul.

Si hay obstáculos presentes, puede darse el caso de que se tengan que revisar más vecinos como se ilustra en la *Figura 24*.

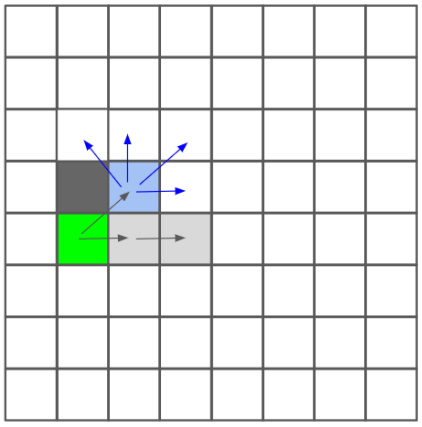


Figura 24. Cuando JPS expande celdas de manera diagonal y hay obstáculos aledaños a la celda a expandir (celda azul), es requerido considerar mayor número de celdas como posibles sucesora. Las flechas en azul indican las direcciones que hay que considerar para encontrar un sucesor para la celda azul. Figura adaptada de *zerowidth* [38].

Cuando hay movimiento en diagonal es requerido revisar tres vecinos o más, lo cual es un poco más difícil comparado con el movimiento vertical u horizontal. En la *Figura*

23, se observa que dos de estos vecinos requieren movimiento horizontal o vertical, por lo tanto, se intenta ubicar un vecino forzado en dichas direcciones siguiendo las reglas de movimiento vertical y horizontal. Si no se encuentra un *vecino forzado* se realiza un movimiento en diagonal y se vuelve a buscar de nuevo de manera horizontal y vertical. Esta situación se ilustra en la *Figura 25*. En este caso el sucesor de la celda verde es la celda azul, porque al explorar de manera horizontal y vertical se encuentra un *vecino forzado*.

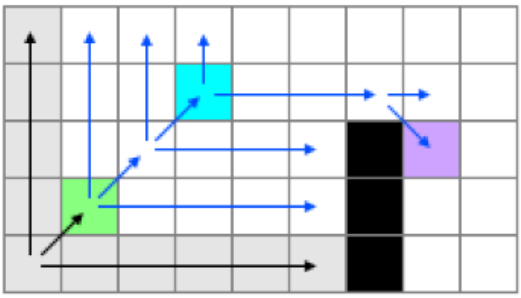


Figura 25. JPS expandiendo de manera diagonal. La celda azul claro representa la celda sucesora, la celda morada un vecino forzado, las celdas oscuras obstáculos. Las flechas en azul indican las celdas inspeccionadas por el algoritmo al intentar hallar un sucesor de la celda verde. Figura adaptada de *zerowidth* [38].

Resumiendo, JPS toma como base el algoritmo A* e intenta mejorar su función sucesora. A* añade todos los vecinos de un vértice a la *lista abierta* al intentar expandir su búsqueda, mientras que JPS va más allá y utiliza una serie de reglas para añadir la menor cantidad de vértices posible a la *lista abierta*.

VENTAJAS Y DESVENTAJAS

La ventaja de JPS es que reduce el número de vértices explorados de manera considerable. Sin embargo, aunque parezca que JPS requiere menos tiempo de cómputo porque explora menos vértices, lo cierto es que en algunos casos las reglas utilizadas requieren más pasos comparado con simplemente ejecutar A*.

Además, al ser un algoritmo basado en A*, en el peor de los casos el orden de crecimiento no mejora con respecto a Dijkstra.

El identificar las situaciones donde JPS es más rápido a comparación de A*, es un punto a analizar en los experimentos.

ALGORITMO BREADTH FIRST SEARCH (BFS)

Breadth First Search (BFS) es un algoritmo de búsqueda de camino inventado por Konrad Zuse en 1945[39]. Comparado con los algoritmos vistos ahora, BFS es un algoritmo relativamente simple. En el *Anexo* se incluye el pseudocódigo.

A diferencia de los otros, BFS utiliza una estructura de datos llamada *cola*. Las *colas* modelan el comportamiento que se ve en las filas de los supermercados -el primero en llegar es el primero en ser atendido-. Considere una cola con los números {5, 9, 2, 3, 1}. Si se realiza la operación *encolar*(4) se añade el número 4 al final de la cola y el resultado sería: {5, 9, 2, 3, 1, 4}. Por otro lado, si se ejecuta la operación *decolar*(), se elimina el primer número en la lista y el resultado es {9, 2, 3, 1, 4}.

A continuación, se detalla paso a paso el funcionamiento de BFS, utilizando un ejemplo.

Considere el grafo de la *Figura 26*. El objetivo es encontrar el camino más corto desde el *vértice a* hacia el *vértice i* y para llevar un registro de los vértices que han sido visitados se utiliza una *cola*. Dado que el *vértice a* es el inicio del trayecto, se añade a la cola.

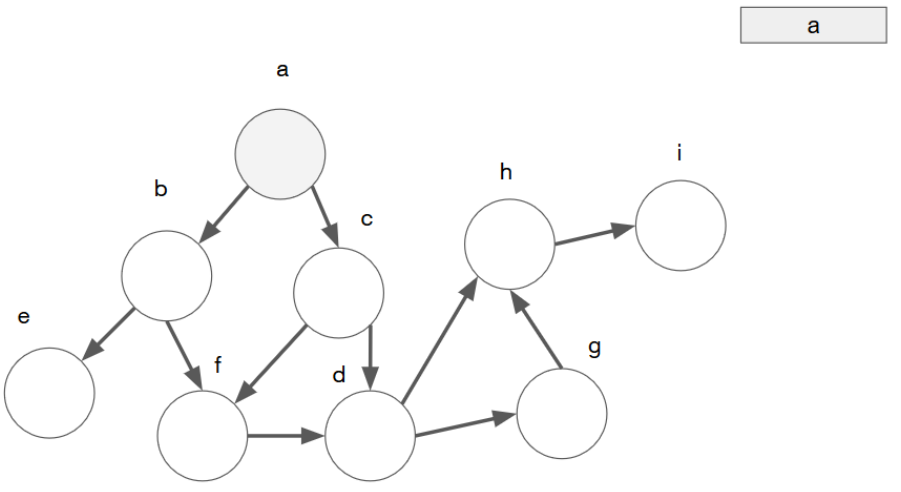


Figura 26. Ejecución del algoritmo Breadth First Search (BFS) en un grafo para encontrar el camino más corto entre el *vértice a* y *vértice i*. El vértice inicial *a* se añade a la *cola* (marcado en gris).

A continuación, se selecciona el *vértice a*, se añaden todos sus vértices vecinos a la cola y finalmente se decola el *vértice a*. En la *Figura 27.* se indica con color azul que el *vértice a* ha sido decolado y se han añadido sus vértices vecinos.

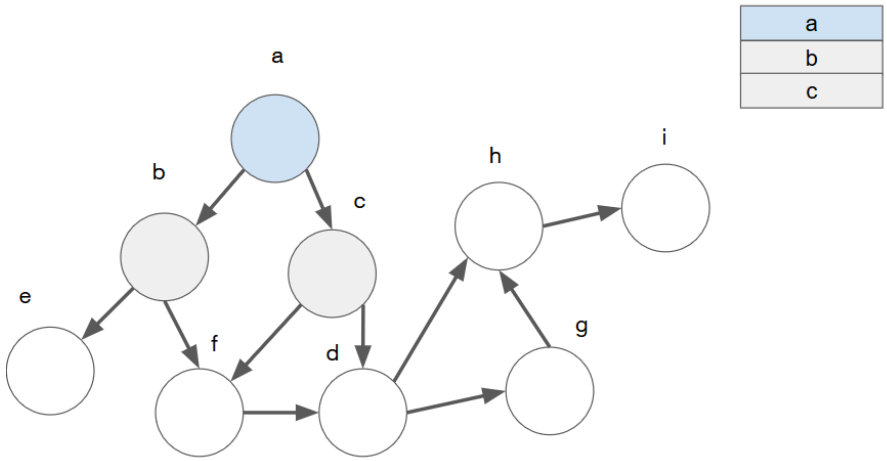


Figura 27. Ejecución del algoritmo BFS. En azul se marcan los vértices que ya han sido explorados y en gris los vértices que restan por explorar. El *vértice a* es un vértice explorado y que ha sido decolado, mientras que el *vértice b* y *c* han sido añadidos

El siguiente elemento en la cola es el *vértice b*. Al igual que el paso anterior, se añaden todos los vértices vecinos de *b* y finalmente se decola *b*. Una ilustración de este paso se muestra en la *Figura 28*.

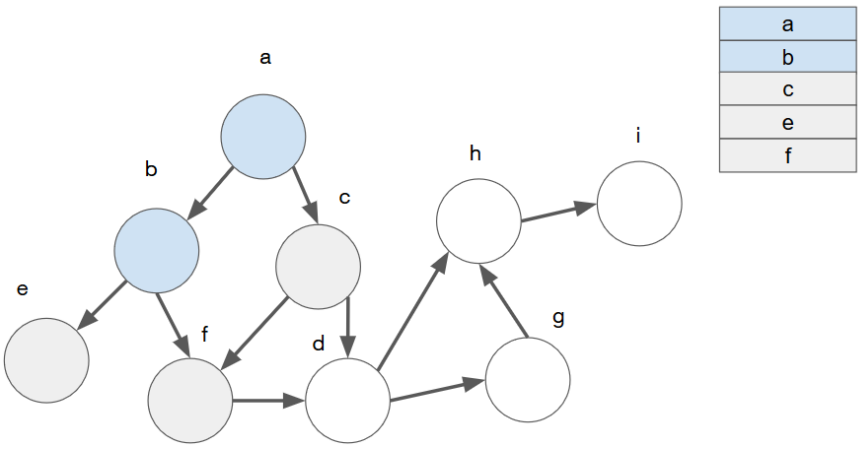


Figura 28. Expansión del *vértice b*. Dicho vértice se decola y se añaden sus vértices vecinos (*vértice e* y *f*) a la cola. En azul se marcan los vértices decolados, en gris los vértices que se encuentran en la cola y sin color aquellos vértices que no han sido agregados a la cola.

Posteriormente se selecciona el siguiente vértice en la cola, el cual vendría siendo el *vértice c*. Una vez más, se añaden todos sus vértices vecinos para después decolar *c*. En la *Figura 29* se ilustra este paso.

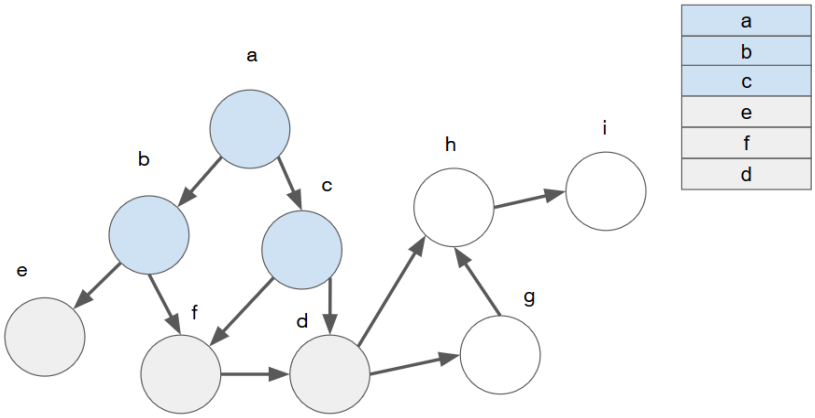


Figura 29. Expansión del *vértice c*. Dicho vértice se decola y se añaden sus vértices vecinos (*vértice d*) a la cola que no lo estuvieran anteriormente. En azul se marcan los vértices decolados, en gris los vértices que se encuentran en la cola y sin color aquellos vértices que no han sido agregados a la cola.

Después se selecciona el *vértice e*. Pero debido a que este vértice no cuenta con vértices vecinos, sólo se decola sin añadir nada nuevo a la cola (*Figura 30*).

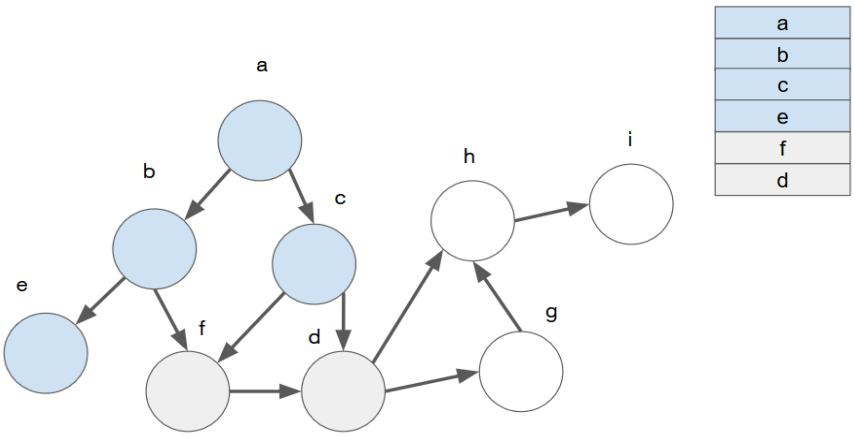


Figura 30. Expansión del *vértice e*. Dicho vértice se decola y se añaden sus vértices vecinos a la cola. Como no hay vecinos que añadir, únicamente el vértice se decola. En azul se marcan los vértices decolados, en gris los vértices que se encuentran en la cola y sin color aquellos vértices que no han sido agregados a la cola.

Enseguida se selecciona el *vértice f* cuyo único vecino es el *vértice d*. Sin embargo, debido a que el *vértice d* ya está presente en la cola, no se encola nada nuevo y simplemente se decola el *vértice f*.

Estos pasos se repiten una y otra vez hasta añadir el vértice objetivo y decolarlo.

Observe la siguiente figura.

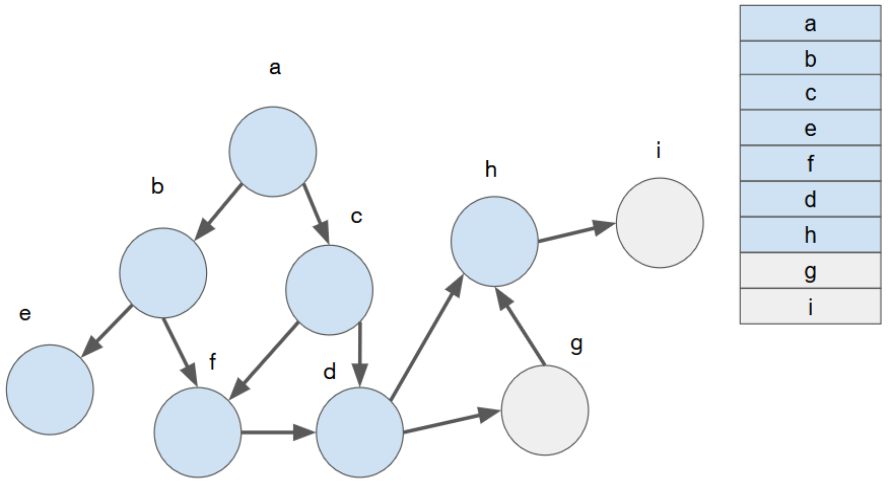


Figura 31. El *vértice i* ha sido añadido en la *cola* pero el algoritmo sigue ejecutándose hasta que dicho vértice haya sido decolado. En azul se marcan los vértices decolados y en gris los vértices que se encuentran en la cola.

El *vértice objetivo i* ha sido añadido en la cola, pero el algoritmo no deja de ejecutarse. La ejecución termina cuando el *vértice i* haya sido decolado. Por lo tanto, primero se selecciona el *vértice g* y se observa que no tiene vecinos que añadir a la cola.

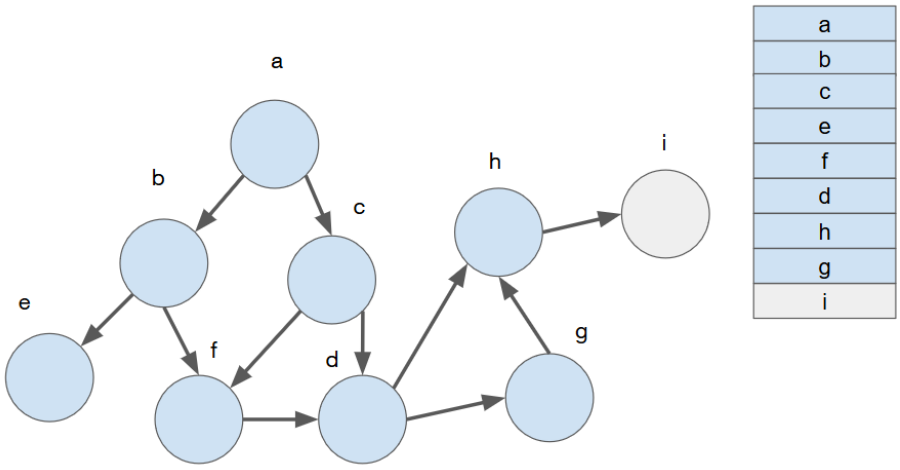


Figura 32. Expansión del *vértice g*. Dicho vértice se decola y se añaden sus vértices vecinos a la cola. Como no hay vecinos que añadir, únicamente el vértice se decola. En azul se marcan los vértices decolados y en gris los vértices que se encuentran en la cola.

Finalmente se selecciona el *vértice i* y se decola. A continuación, se sigue el camino de regreso para llegar al *vértice a* desde el *vértice i* y se devuelve la solución. El camino más corto para ir de *a* hacia *i* es pasar el *vértice c*, *d*, *h* y finalmente *i*.

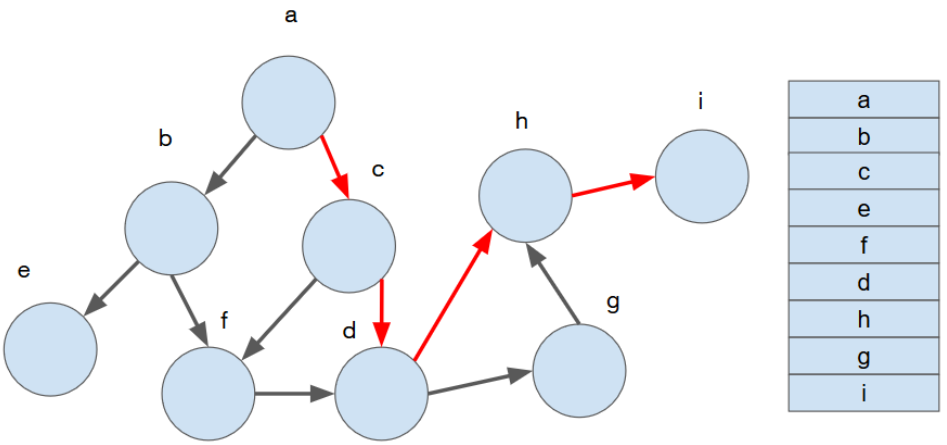


Figura 33. Expansión del *vértice i*. Finalmente, el algoritmo deja de ejecutarse debido a que el *vértice i* ha sido decolado. En azul se marcan los vértices decolados.

El orden de crecimiento de BFS es de $O(|V| + |E|)$ donde $|V|$ y $|E|$ representan el número total de vértices y aristas respectivamente. Esto es una mejora comparado con el orden de crecimiento de Dijkstra de $O(|E| \log |V| + V \log |V|)$ si se utiliza la implementación dónde la cola de prioridad se representa con un montículo binario.

La limitante con el algoritmo BFS es que sólo computa la solución correcta si las aristas del grafo no tienen peso. Considere el grafo en la *Figura 34*. BFS señalaría que el camino más corto para ir de A a B es tomar la arista AB directamente porque no se considera que el peso de la arista es 10, cuando en realidad el camino óptimo es transitar la arista AC y después CB.

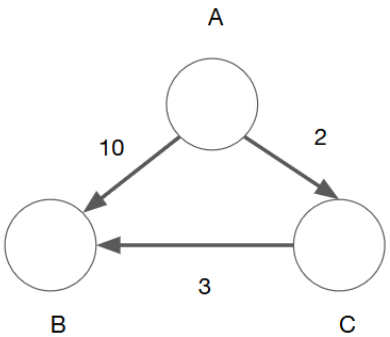


Figura 34. Un grafo ponderado con vértices A, B y C. Al ejecutar el algoritmo BFS para encontrar el camino más corto para ir de *A* y *B* en este grafo, se obtiene la respuesta equivocada A->B, cuando el camino más corto en realidad es A->B->C. BFS no da la respuesta correcta en grafos ponderados.

BFS computa la solución óptima, pero considera que dicha solución es aquella dónde se transita el menor número de aristas. A simple vista esto podría parecer un problema para las representaciones de cuadrícula dónde se permiten movimientos en diagonal con un peso de $\sqrt{2}$, pero transitar una arista en diagonal sigue siendo óptimo que transitar dos aristas en vertical y horizontal como se ilustra en la *Figura 35*.

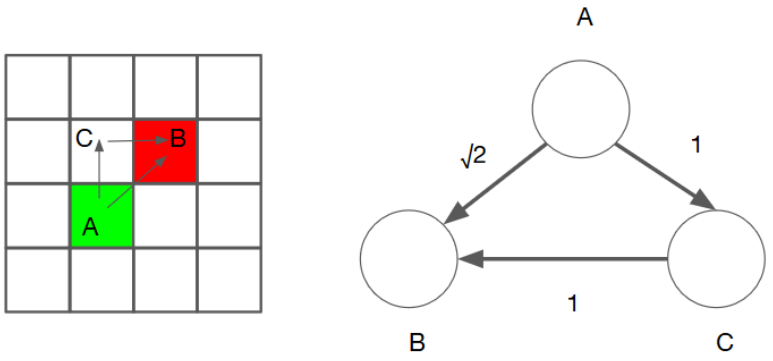


Figura 35. Grafo con vértices A, B y C representando tres celdas de una representación de cuadrícula. Ejecutar el algoritmo BFS en este grafo para ir de A hacia B brinda la respuesta correcta en este caso. En la representación de cuadrícula con movimiento diagonal con costo de $\sqrt{2}$, sigue funcionando correctamente.

Para estas representaciones de cuadrícula, el camino que transita menor número de aristas sigue siendo la solución correcta. BFS brindaría una solución errónea si los movimientos en diagonal fueran de un peso mayor a 2, lo cual no es el caso.

VENTAJAS Y DESVENTAJAS

La ventaja principal de BFS es que su orden de crecimiento es menor comparado con Dijkstra, convirtiéndolo en un algoritmo relativamente rápido. Sin embargo, al igual que Dijkstra, su enfoque es buscar a ciegas sin suponer nada del grafo; exceptuando que las aristas no tienen peso. Si el beneficio obtenido por el orden de crecimiento es mayor o no comparado con algoritmos como A* y JPS que utilizan heurísticas para acelerar la búsqueda, se verá en los experimentos.

Otra desventaja es que debido a que el algoritmo sólo funciona en grafos que no tienen aristas con peso, las circunstancias donde se puede aplicar BFS son limitadas.

ALGORITMO BEST FIRST SEARCH (CODICIOSO)

Los algoritmos tipo codicioso son una familia de algoritmos que utilizan heurísticas para computar soluciones óptimas de manera local en cada paso con la esperanza de producir una solución óptima de manera global. Aunque usualmente no suelen producir un resultado óptimo de manera global, son bastante prácticos porque son rápidos y la solución que computan brindan aproximaciones aceptables.

Para el caso de algoritmos de búsqueda de camino se traduce a seleccionar y expandir los vértices que estén más cerca del objetivo.

La implementación es similar al algoritmo A*, con la diferencia de que no se calculan los *valores G* y sólo se calculan los *valores H*. Por lo tanto, en la *lista abierta* el siguiente vértice a expandir es aquel con el *valor H* más pequeño, o en otras palabras, el vértice cuya distancia estimada que queda por transitar para llegar al objetivo sea la menor.

El algoritmo codicioso no garantiza encontrar el camino más corto, pero a cambio le toma menor número de pasos computar la solución.

VENTAJAS Y DESVENTAJAS

La ventaja del algoritmo codicioso es que es bastante rápido debido a que si su heurística lo lleva a buscar por el camino correcto, el número de vértices que tiene que expandir se reduce.

La desventaja está en que, si las circunstancias demandan estrictamente encontrar el camino más corto, entonces el algoritmo no es útil porque no garantiza computar la solución óptima. En algunas circunstancias realmente no es requerido obtener el camino más corto y simplemente encontrar un camino desde un vértice hacia el objetivo lo más rápido posible. En esa situación el algoritmo codicioso es el algoritmo ideal. Si las soluciones aproximadas que brinda el algoritmo codicioso son lo suficientemente aceptables, se analiza en los experimentos.

Parte IV

EXPERIMENTOS

METODOLOGÍA

En esta sección se describen y discuten los experimentos realizados. Los algoritmos que se analizaron en los experimentos son los siguientes:

- Dijkstra
- A* con función heurística octile
- JPS con función heurística octile
- Breadth First Search (BFS)
- Best First Search (codicioso) con función heurística octile

Para todos los algoritmos se utilizó la representación de cuadrícula que permite movimiento diagonal.

Se escribió un programa utilizando el lenguaje de programación JavaScript. Este programa permite colocar obstáculos en una malla cuadriculada, elegir una celda de inicio, una celda objetivo, un algoritmo y visualizar cómo el algoritmo encuentra el camino más corto desde la celda de inicio hacia la celda objetivo. El programa contabiliza el número de operaciones o pasos que toma el algoritmo para encontrar la solución.

Se diseñaron distintos mapas de diferentes tamaños y para cada uno se colocó una celda de inicio y una celda objetivo. Se ejecutó cada algoritmo para encontrar el camino más corto entre las celdas y conocer el número de pasos que le toma a cada uno encontrar la solución.

HARDWARE UTILIZADO

Los experimentos se realizaron en un equipo que cuenta con un procesador Intel Core i5 9300H de 2.4 GHz, 16 GB de RAM DDR4 de 2666 MHz en el sistema operativo Ubuntu 18.04.

CREACIÓN DE MAPAS

Algunos mapas como el *mapa de giros* y el *mapa de cuadros* fueron creados utilizando un editor de imágenes. En las imágenes cada píxel oscuro representa un obstáculo mientras que un pixel claro representa una celda transitable. El programa permite cargar las imágenes y visualizar las celdas que explora cada algoritmo.

En el caso de los *mapas de cruces*, se generaron automáticamente, colocando cruces de manera aleatoria. Para los *mapas laberinto* se generaron utilizando una herramienta online que permite generar imágenes de laberintos [40].

RESULTADOS Y DISCUSIÓN

MAPA DE CRUCES

El primer tipo de mapa utilizado en los experimentos contiene obstáculos en forma de cruces. Una ilustración del mapa de mayor tamaño, se muestra en la *Figura 36*. Con dicho mapa se pretende representar aquellos entornos con espacios abiertos y obstáculos pequeños distribuidos de manera uniforme. En este tipo de mapa se ejemplifica la situación donde los algoritmos de búsqueda de camino mantienen cierta dirección hacia el objetivo, realizando giros pequeños de vez en cuando para rodear los obstáculos. Es similar a caminar a pie en una vecindad y rodear edificios.

Se eligió la forma de cruz debido a que generan pequeños caminos sin salida y tienen más detalles a comparación de cuadrados o círculos.

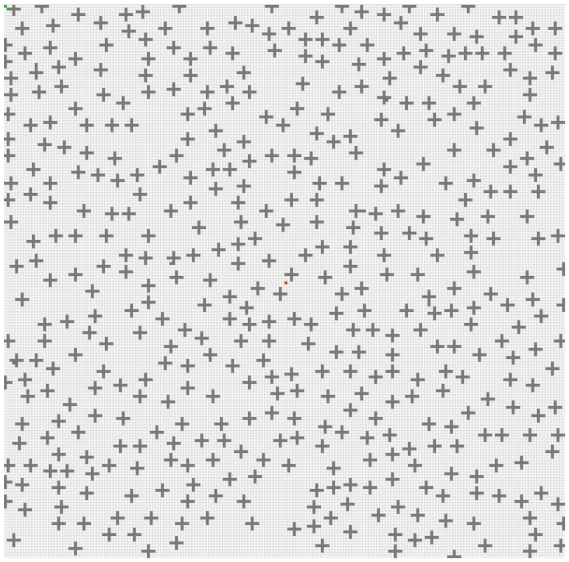


Figura 36. Mapa de 200x200 celdas de longitud con obstáculos en forma de cruces.

Por cada algoritmo se registró el número de operaciones para encontrar el camino más corto desde cada una de las cuatro esquinas hacia el centro del mapa. Se calculó el promedio de los cuatro datos.

Este procedimiento se repitió en mapas de distinto tamaño que fueron generados recortando el mapa más grande como se ilustra en la *Figura 37*.

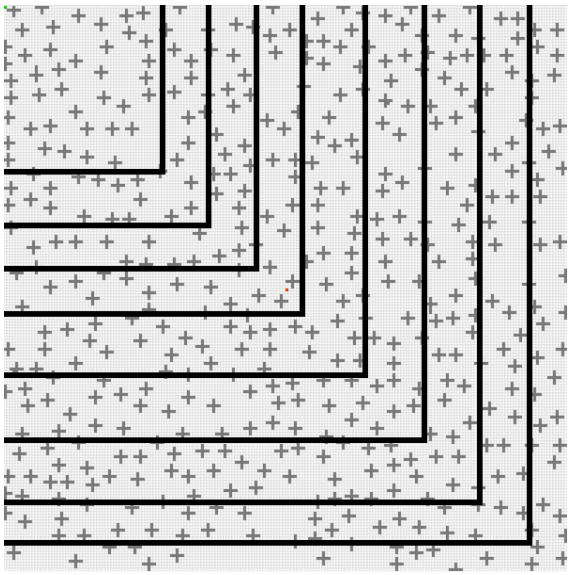


Figura 37. Ilustración de la manera en la que se recortó el mapa de cruces para obtener distintos mapas pequeños.

Los tamaños de cada mapa fueron de 20x20, 40x40, 60x60, 80x80, 100x100, 120x120, 140x140, 160x160, 180x180 y 200x200.

A continuación, en la *Figura 38* se presenta una gráfica del número de operaciones vs. la longitud del mapa para cada algoritmo ejecutado en los *mapas de cruces*.

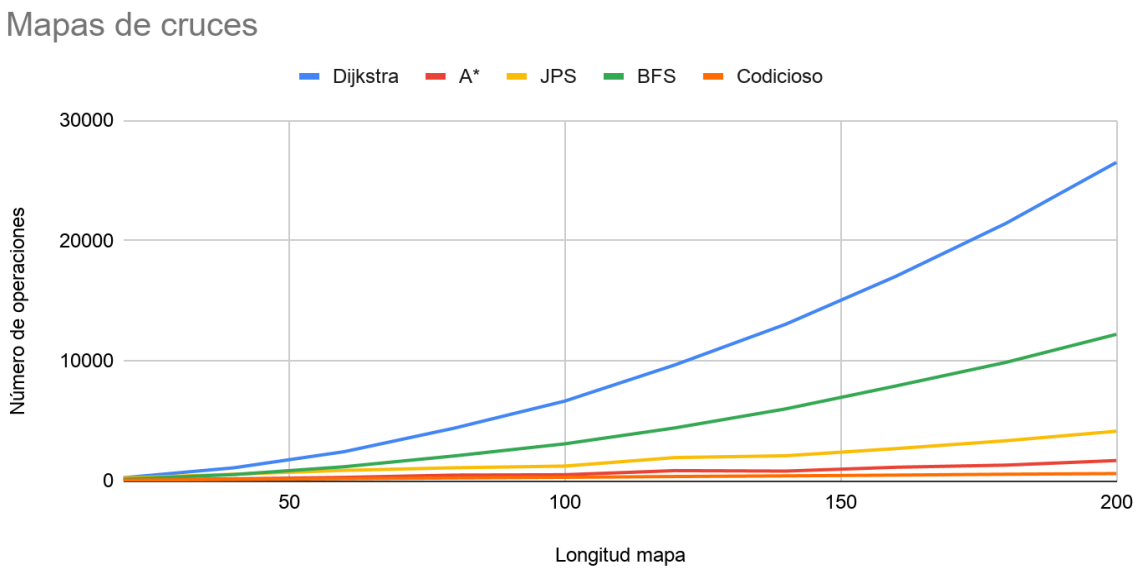


Figura 38. Resultados de búsqueda de camino más corto en el mapa de cruces. El algoritmo de Dijkstra es más lento (mayor número de operaciones) y le sigue BFS, JPS, A* y algoritmo codicioso.

En los resultados se observa que el algoritmo de Dijkstra es significativamente más lento (mayor número de operaciones) comparado con el resto de algoritmos. Esto es algo esperado debido a que Dijkstra al no utilizar ninguna función heurística, no tiene una “noción” de dónde se encuentra el objetivo y por lo tanto expande su búsqueda de manera homogénea hacia todas direcciones. Esto es contraste con A*, JPS y el algoritmo codicioso que son mucho más rápidos gracias a que sus heurísticas son útiles para hallar rápidamente el objetivo. Ejemplo de ello se ilustra en la *Figura 39*, donde se compara el número de celdas exploradas (celdas azules) por el algoritmo de Dijkstra y A*.

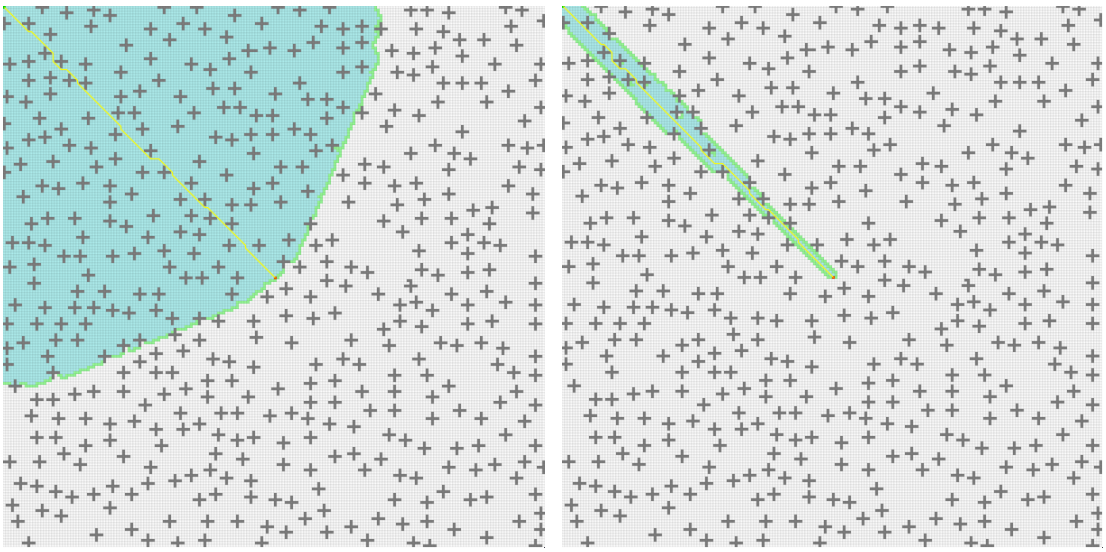


Figura 39. Comparación visual del número de celdas exploradas en el mapa de cruces con los algoritmos de Dijkstra (izquierda) y A* (derecha). Las celdas azules representan las celdas exploradas.

BFS tiene una complejidad temporal menor que Dijkstra y por consiguiente menor al resto de algoritmos. En los resultados se observa que le toma menor número de operaciones en computar una solución comparado con Dijkstra, pero no con el resto de algoritmos. BFS al igual que Dijkstra expande celdas de manera homogénea hacia todas direcciones, sin ninguna “noción” de dónde está el objetivo. La diferencia con el algoritmo de Dijkstra es que expandir le toma menor número de operaciones porque no toma en cuenta el costo de las aristas del grafo. Por lo tanto, BFS por lo menos es más rápido que el algoritmo de Dijkstra, en cualquier caso.

Comparando A* con JPS, en este caso la estrategia de JPS de controlar el cómo se expanden las celdas que se van a explorar, termina tomando más pasos que

simplemente utilizar A*. Por lo tanto, para los *mapas de cruces*, A* es el algoritmo más eficiente y que brinda una solución con el camino más corto.

Por otro lado, se observa que algoritmo tipo codicioso es más rápido que A*, pero la solución que brinda no es óptima. Una comparación entre la distancia del camino más corto y la distancia del camino encontrado por el algoritmo codicioso se muestra en la *Figura 40*.

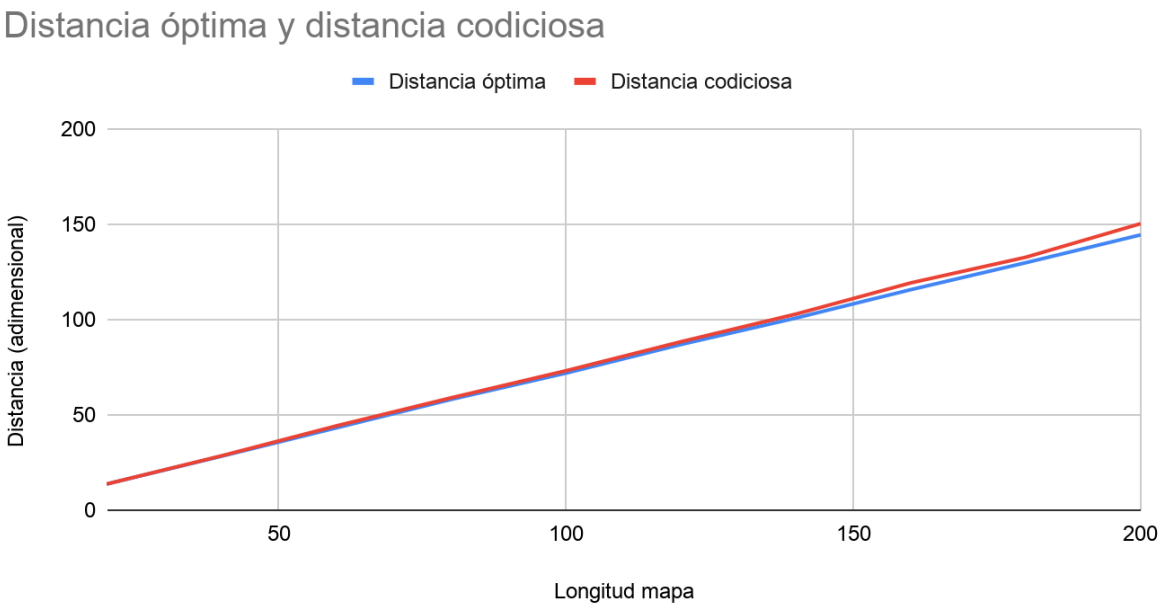


Figura 40. Comparación entre la distancia del camino más corto (azul) y la distancia del camino encontrado por el algoritmo codicioso (rojo) en mapas de distinto tamaño con obstáculos de cruces.

Nótese que la diferencia entre la distancia del camino más corto y la distancia del camino hallado por el algoritmo codicioso es casi negligible. Por ello, para mapas con obstáculos pequeños, con la misma forma y distribuidos de manera aleatoria, el algoritmo codicioso es un buen candidato si la prioridad es computar una solución lo más rápido posible sin importar mucho si el camino encontrado es exactamente el más corto, aunque la diferencia entre el camino óptimo no es mucha. En caso de que sea estrictamente necesario asegurar que el camino sea óptimo, lo mejor es utilizar A*.

MAPA DE CUADRADOS

El segundo tipo de mapa consiste en un arreglo de cuadrados similar a edificios que forman un sistema de calles. El diseño del mapa tiene como objetivo observar el comportamiento de los algoritmos en entornos donde hay pasillos largos y obstáculos distribuidos de manera ordenada, en contraste con el mapa de cruces en el cual los obstáculos están distribuidos de manera aleatoria. Una ilustración del mapa se muestra en la *Figura 41*.

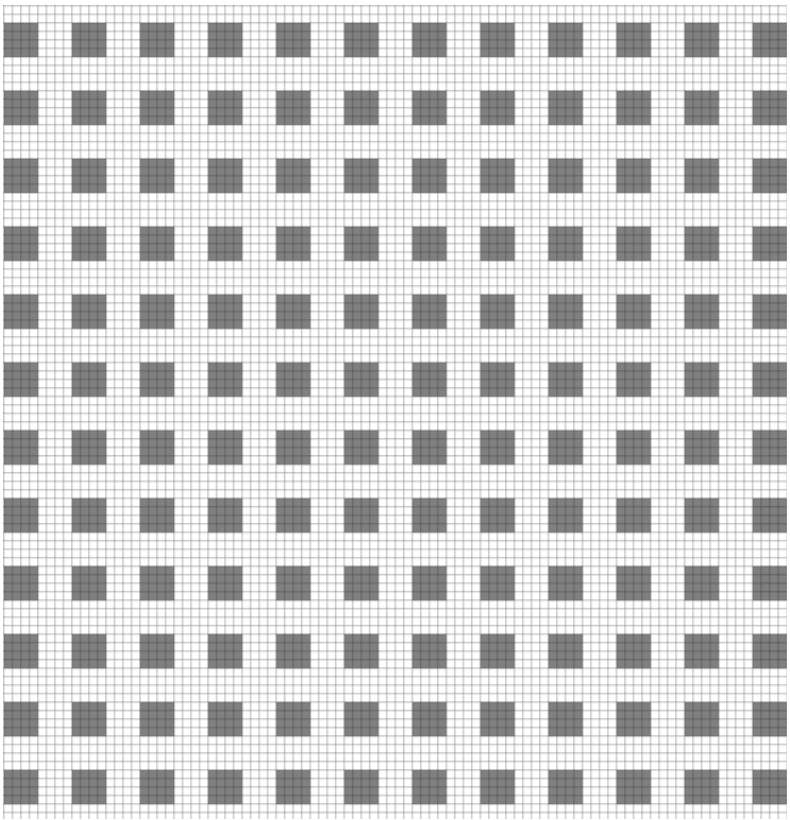


Figura 41. Mapa de 100x100 celdas de longitud con obstáculos en forma de cuadrados y pasillos. El punto de inicio es cada una de las esquinas, mientras que el objetivo es el centro del mapa

Al igual que el *mapa de cruces* se registró el número de operaciones que toma ir desde cada una de las cuatro esquinas hacia el centro del mapa y se calculó el promedio.

Se realizó el procedimiento para mapas de 20x20, 40x40, 60x60, 80x80, 100x100, 120x120, 140x140, 160x160, 180x180 y 200x200.

A continuación, en la *Figura 42*. se presentan los resultados del *mapa de cuadrados*.

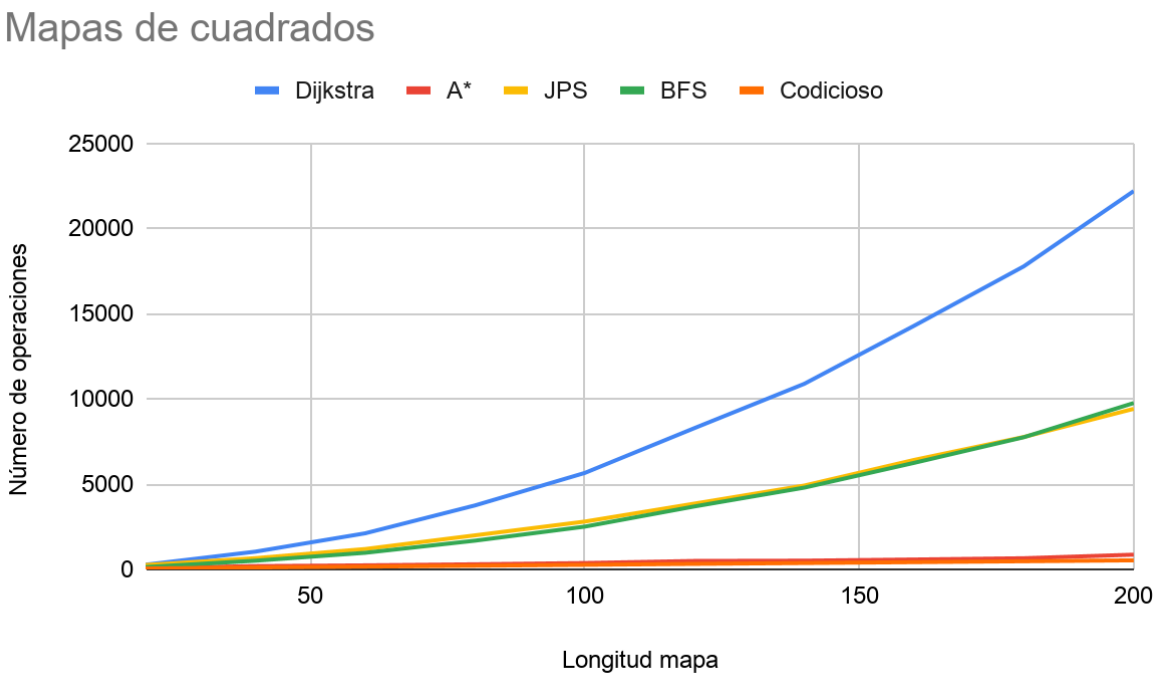


Figura 42. Resultados de búsqueda de camino más corto en los *mapas de cuadrados*. Obsérvese que Dijkstra es el algoritmo más lento, le sigue BFS, JPS, A* y algoritmo codicioso respectivamente.

Los resultados son similares al *mapa de cruces*. Dijkstra es el algoritmo que requiere mayor número de operaciones y luego le sigue BFS. Ambos algoritmos crecen de manera similar si se compara con el *mapa de cruces,* aunque en este caso requieren menor número de operaciones. Esto puede ser porque el *mapa de cuadros* tiene más obstáculos comparado con el *mapa de cruces* y por lo tanto hay menor cantidad de celdas a explorar.

Comparado con los resultados del *mapa de cruces*, JPS requiere significativamente mayor número de operaciones que A*. Esto es debido a que en este mapa hay pasillos sin ningún obstáculo. Al no haber ningún obstáculo, el algoritmo no encuentra un *vecino forzado*. Por lo tanto, antes de expandir y explorar la siguiente celda, JPS intenta buscar un *vecino forzado* de manera vertical u horizontal hasta llegar al borde del mapa. A diferencia del mapa de cruces, donde rápidamente el algoritmo encuentra un obstáculo y por lo tanto un vecino forzado.

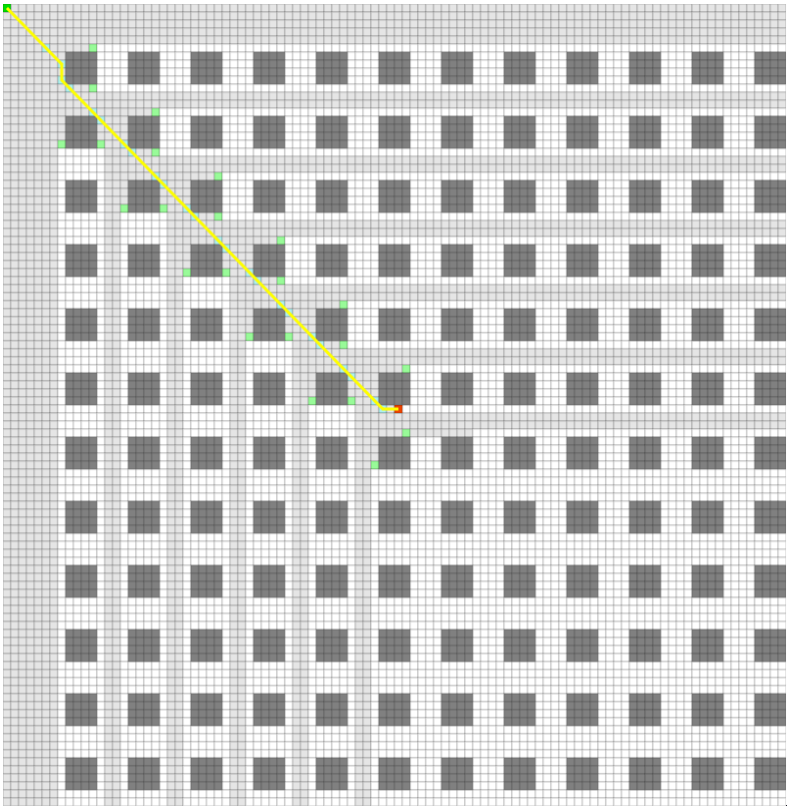


Figura 43. Visualización de celdas inspeccionadas por JPS para encontrar un *vecino forzado*. Las celdas en color gris claro son las celdas inspeccionadas para intentar encontrar celdas sucesoras. En verde se marcan posibles celdas sucesoras encontradas por el algoritmo.

Una visualización del número de celdas inspeccionadas por JPS para encontrar un *vecino forzado* se ilustra en la *Figura 43*. Las celdas grises representan las celdas inspeccionadas para encontrar un *vecino forzado*.

En este mapa el algoritmo codicioso es el algoritmo más rápido, aunque A* le sigue de cerca. El algoritmo codicioso no garantiza encontrar la solución óptima, pero en este caso la encontró para todos los mapas. Si se inspecciona el camino trazado por ambos algoritmos, se observa que la solución es casi una línea recta, como si jamás se hubieran topado con alguno de los obstáculos.

Por ello en situaciones donde no hay obstáculos o los obstáculos no provocan que el camino óptimo tenga que hacer giros o desviarse, el algoritmo codicioso tiende a encontrar la solución óptima.

MAPA DE GIROS

Con el tercer tipo de mapa se busca observar el comportamiento de los algoritmos en entornos donde el camino más corto tiene giros bruscos y momentáneamente se aleja del objetivo, en contraste con los dos mapas anteriores donde el camino más corto tiene giros pequeños. El mapa tiene un diseño laberíntico donde ciertos caminos conducen a pasillos sin salida. También a diferencia de los mapas laberínticos tradicionales donde hay pasillos estrechos, en este tipo de mapas hay espacios abiertos.

La finalidad de diseñar un mapa de este tipo es para observar cómo A* empieza a comportarse bastante similar al algoritmo de Dijkstra debido a que su función heurística no es muy útil en mapas de este tipo. Además, está la intención de observar si JPS ofrece una ventaja respecto a A* que no sea relacionado con funciones heurísticas.

En la *Figura 44* se ilustra el mapa de mayor tamaño junto con su camino más corto.

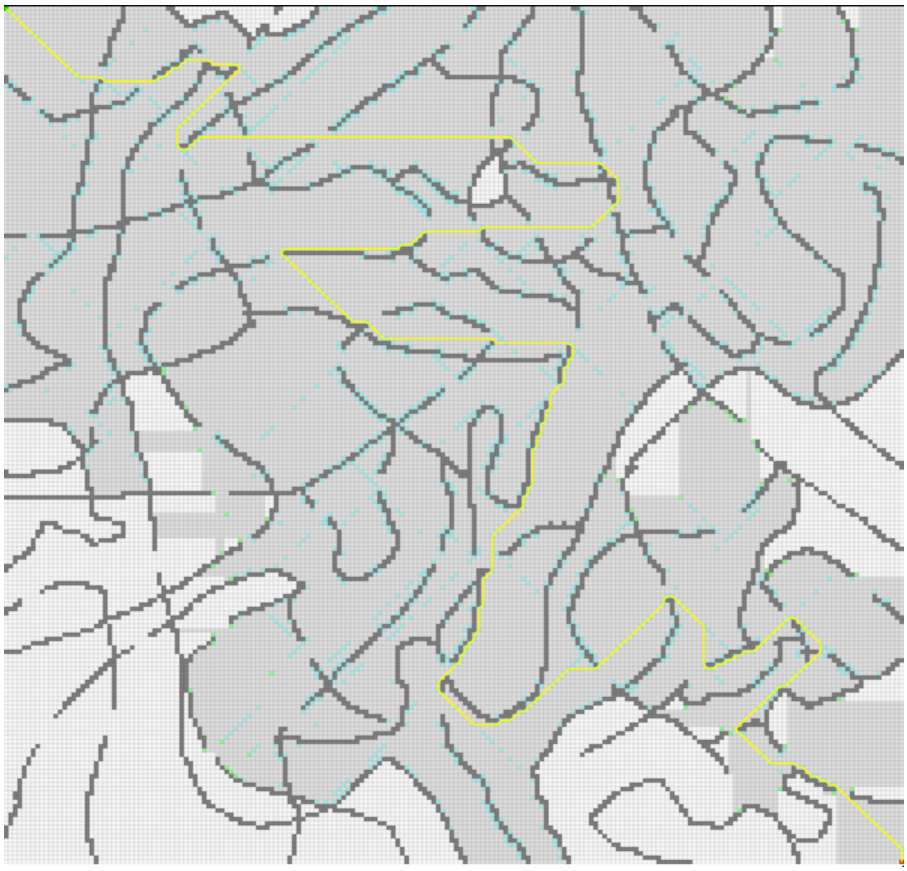


Figura 44. Mapa de 200x200 celdas de longitud, en el cual el camino más corto desde la esquina superior izquierda hacia la esquina inferior derecha requiere giros bruscos (línea amarilla).

De igual manera que el *mapa de cruces* se hicieron recortes del mapa de mayor tamaño para generar distintos tamaños.

Para cada mapa se buscaba que el camino más corto tuviera varios giros repentinos para evitar que los algoritmos se beneficien de sus funciones heurísticas. El punto de partida para este tipo de mapas se encuentra en la esquina superior izquierda y el objetivo en la esquina inferior derecha.

Para cada mapa se corrió cada algoritmo y se registró el número de pasos para computar el camino más corto. Los resultados se muestran en la gráfica de la *Figura 45*.

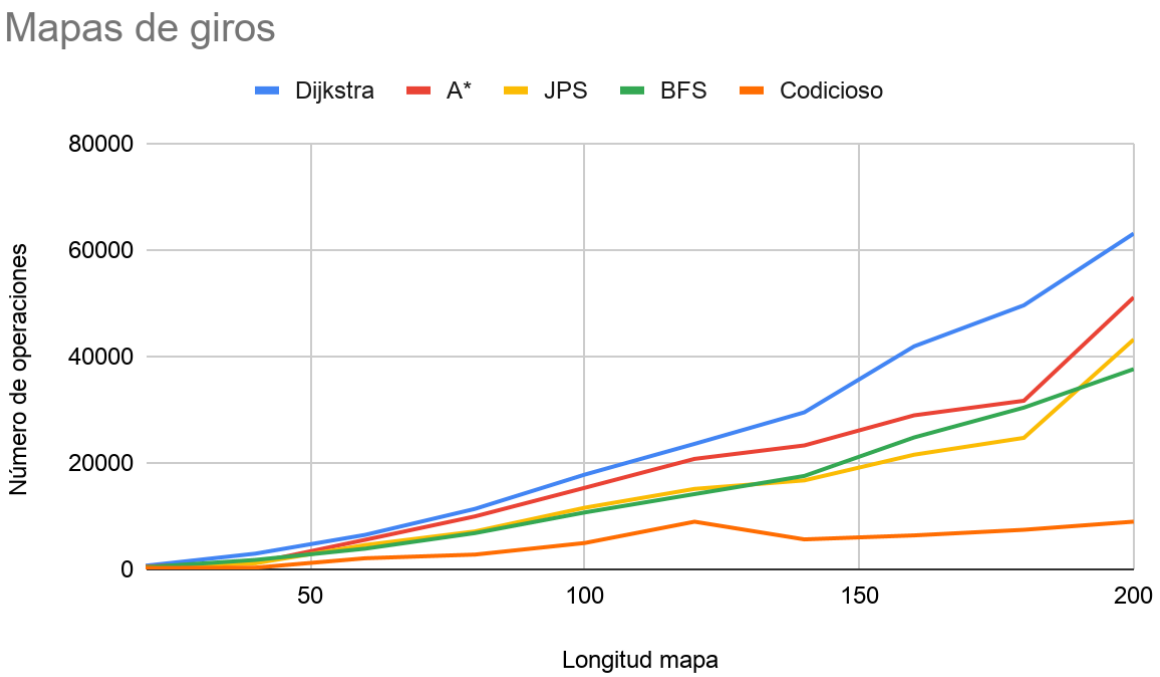


Figura 45. Resultados de búsqueda de camino más corto utilizando los *mapas de giros*. Nótese que el número de operaciones para A* tiende a ser similar al del algoritmo de Dijkstra. El algoritmo más rápido en este mapa es el algoritmo codicioso, seguido de BFS, JPS, A* y por último Dijkstra.

Se observa que A* crece más rápido a comparación de los mapas anteriores. Esto es debido a que la forma del camino más corto es un camino con giros bruscos y que en varias ocasiones se aleja de la celda objetivo. La heurística utilizada en A* causa que se dé prioridad a explorar las celdas más cercanas al objetivo. Sin embargo, esto provoca que se utilicen pasos de más cuando se encuentran callejones sin salida y se requiere buscar el camino por otra dirección.

Cuando la heurística de A* se vuelve ineficiente en encontrar el camino más corto hacia el objetivo, A* tiende a comportarse de manera similar al algoritmo de Dijkstra. Esto muestra que usar heurísticas no siempre provee un beneficio.

En estos mapas expandir de manera selectiva encontrando puntos de salto con el algoritmo JPS, es más eficiente que utilizar A*. A diferencia de los *mapas de cuadrados*, en este tipo de mapa hay varias paredes u obstáculos que permiten que JPS encuentre *vecinos forzados* y añada rápidamente una celda a la *lista abierta*.

BFS está a la par con JPS, lo que lo convierte en una muy buena alternativa cuando no hay mucho beneficio en utilizar funciones heurísticas.

El algoritmo codicioso es el algoritmo más rápido, pero a cambio de sólo encontrar una aproximación del camino óptimo. En la *Figura 46* se muestra una comparación de la distancia a recorrer entre el camino encontrado por el algoritmo codicioso y el camino óptimo.

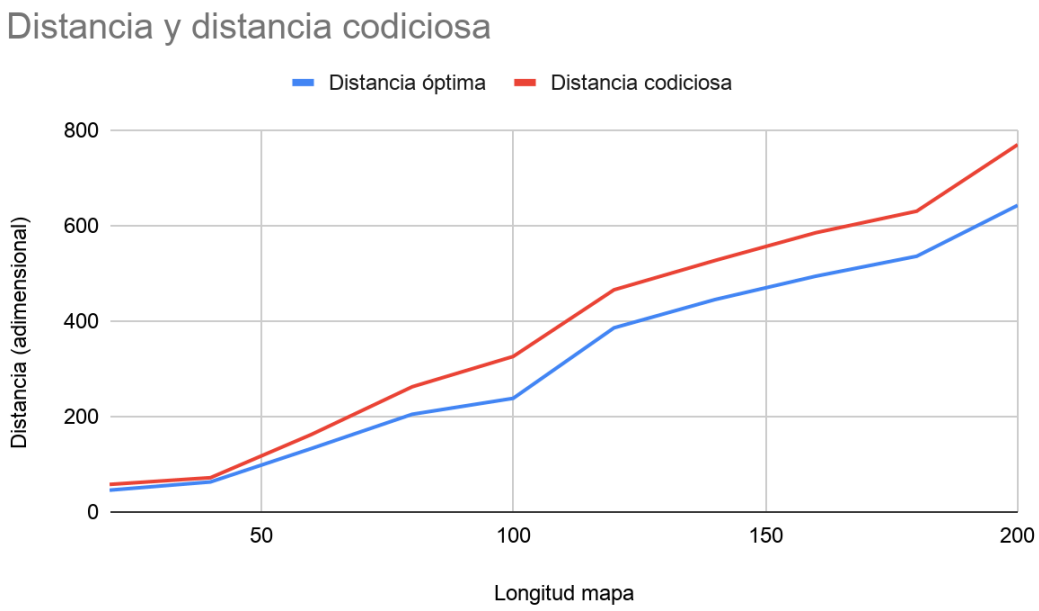


Figura 46. Comparación entre la distancia del camino más corto (azul) y la distancia del camino encontrado por el algoritmo codicioso (rojo) en los *mapas de giros*.

Para el mapa más grande la diferencia de distancia entre el camino más corto y el camino encontrado por el algoritmo codicioso es alrededor de 100 unidades. El algoritmo codicioso encuentra una solución distante de ser óptima cuando el camino

más corto tiene varios giros y se aleja del objetivo, en contraste con lo mencionado anteriormente en el *mapa de cuadrados*, dónde si el camino más corto no tiene desvíos el algoritmo codicioso tiende a encontrar una solución cercana a la óptima.

Al igual que A*, la heurística que utiliza el algoritmo codicioso se vuelve ineficiente, dado que dicha heurística considera que las celdas más prometedoras son aquellas que están más cercanas al objetivo.

Nótese que a comparación de los otros mapas el número de operaciones es mayor para todos los algoritmos. Esto es debido a que anteriormente la celda inicial y la celda objetivo eran una esquina y el centro del mapa respectivamente. Pero para estos mapas el inicio y el objetivo se encuentran en esquinas opuestas, provocando que exploren más celdas.

MAPA DE LABERINTO

Los siguientes mapas son laberintos tradicionales con pasillos estrechos y varios callejones sin salida. Son mapas similares a los *mapas de giros* con la diferencia de que no hay espacios abiertos.

Los mapas fueron creados utilizando un generador de mapas en línea [9]. Los pasillos sólo son de una celda de espesor y los tamaños de los mapas son 20x20, 40x40, 60x60, 80x80, 100x100, 120x120, 140x140, 160x160, 180x180 y 200x200 dónde cada uno es un laberinto diferente. Una ilustración del mapa de 100x100 se muestra en la *Figura 47*.

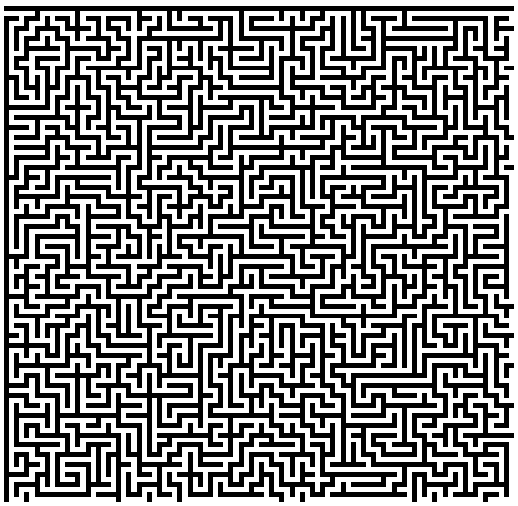


Figura 47. Mapa de 100x100 celdas de longitud, de un laberinto clásico. El punto de inicio se encuentra en la esquina superior izquierda y el objetivo en la esquina superior derecha.

Para cada mapa se ejecutó cada algoritmo para encontrar el camino más corto desde la esquina superior izquierda hacia la esquina inferior derecha y se registró el número de pasos.

En la *Figura 48* se muestran los resultados para los mapas laberínticos.

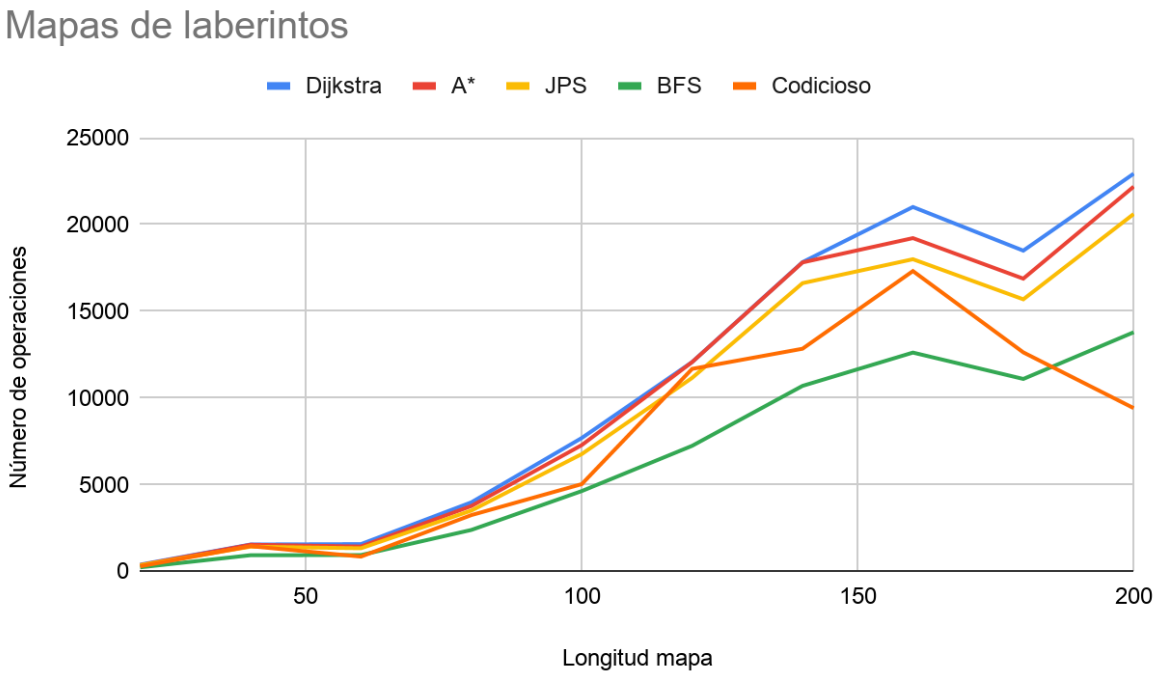


Figura 48. Resultados de búsqueda de camino más corto en *mapas de laberinto*. Nótese que Dijkstra, A* y JPS crecen de manera similar. El algoritmo más lento es Dijkstra, le sigue A*, JPS, BFS y finalmente el algoritmo codicioso.

Para este tipo de mapas JPS es más rápido que A*. Los mapas laberínticos con pasillos estrechos son ideales para JPS porque estos limitan las direcciones en las que hay que buscar puntos de salto. Además, como el mapa tiene varias paredes de obstáculos esto ayuda a que JPS encuentre *vecinos forzados* con frecuencia. Así JPS expande las celdas a explorar con rapidez haciéndolo más eficiente que A*.

Las heurísticas en este tipo de mapa no son muy útiles, a tal grado que incluso el algoritmo codicioso requiere un gran número de pasos en computar una solución. Esto ocurre porque hay varios callejones sin salida bastante profundos, es decir, caminos que requieren que se recorra un gran número de celdas de antemano. Por ello, se termina explorando casi todas las celdas transitables del mapa.

En estos mapas la solución que provee el algoritmo codicioso no es la óptima, pero la diferencia de unidades de distancia comparado con la solución óptima es de alrededor de 5 unidades para el mapa más grande. Esto ocurre porque no hay muchos caminos alternativos que se puedan encontrar, debido a que los pasillos son muy estrechos y sólo hay una forma de llegar hacia el objetivo. Los otros caminos alternativos conducen hacia callejones sin salida.

BFS es el algoritmo más rápido, porque si se da el caso de que se exploren todas las celdas del mapa a BFS le tomaría menor tiempo comparado con el algoritmo de Dijkstra debido a que su complejidad temporal es menor.

MAPA DE LABERINTO ESCALADO

Para el quinto tipo de mapa se generó un laberinto clásico y se escaló dicho mapa a distintos tamaños. Para cada mapa la estructura del laberinto es la misma, con la diferencia de que los pasillos son más anchos conforme se aumenta el tamaño (*Figura 49*).

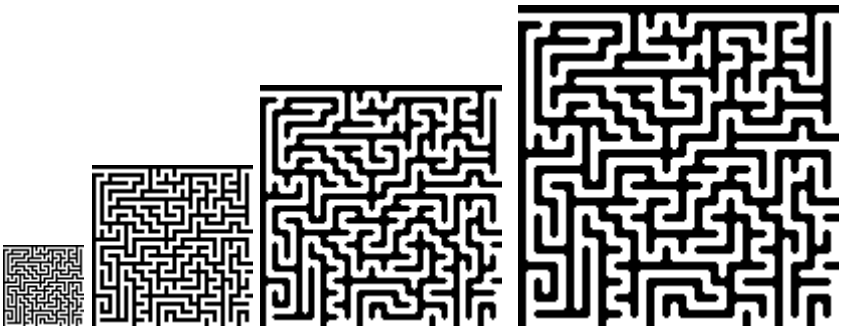


Figura 49. *Mapas de laberintos escalados* con distintos anchos de pasillo. De izquierda a derecha, mapa de 40x40 con pasillos de 1 pixel de ancho, mapa de 80x80 con pasillos de dos pixeles de ancho, mapa de 120x120 con pasillos de 3 píxeles de ancho, mapa de 160x160 con pasillos de 4 píxeles de ancho.

Para este tipo de mapas el objetivo es observar el comportamiento de cada algoritmo conforme se aumenta el ancho de los pasillos.

El punto de partida es la esquina superior izquierda y el objetivo la esquina inferior derecha. Para cada mapa se ejecutó cada algoritmo y se registró el número de pasos para computar el camino más corto.

A continuación, en la *Figura 50* se presentan los resultados del mapa del mismo laberinto, pero con pasillos de distinto ancho.

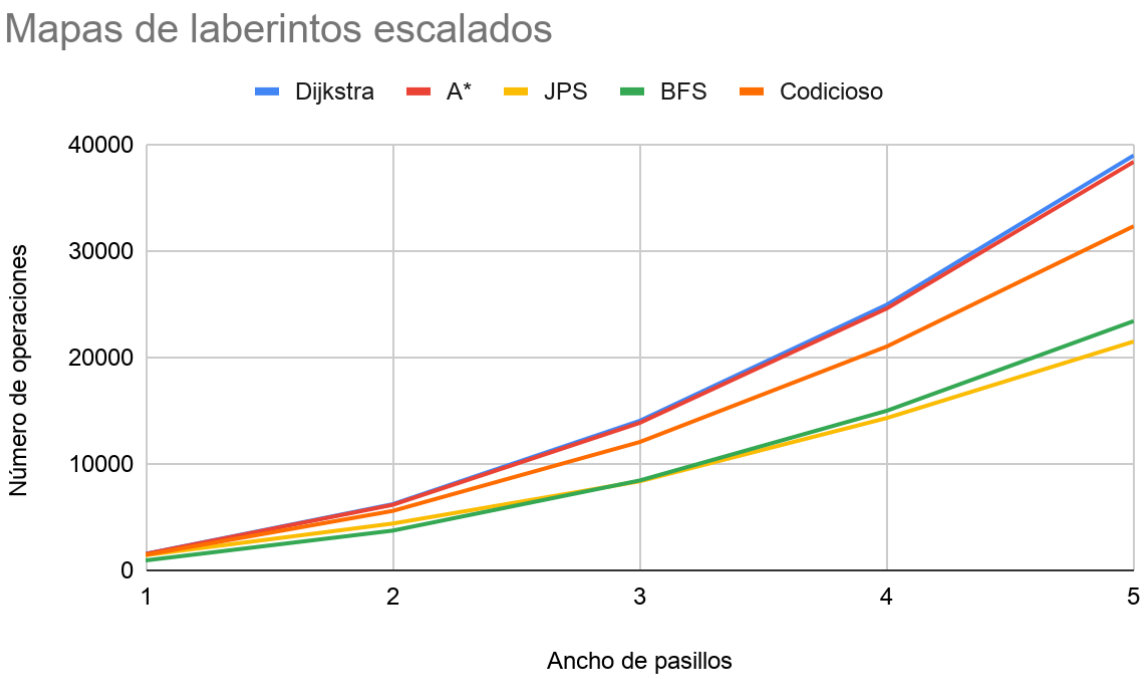


Figura 50. Resultados de búsqueda de camino más corto en *mapas de laberintos escalados*. El algoritmo más lento es Dijkstra, seguido de A*, el algoritmo codicioso, BFS y finalmente JPS.

En estos experimentos se utilizó el mapa de 40x40 del conjunto de *mapas de laberintos* y se escaló a varios tamaños. Observe en la *Figura 48* que para este mapa de 40x40 el algoritmo codicioso es menos eficiente que JPS. Por lo tanto, no es de extrañar que, si se utilizan versiones de distintos tamaños de este mapa, en los resultados el algoritmo codicioso se mantenga como menos eficiente que JPS.

El motivo por el cual el algoritmo codicioso se vuelve menos eficiente es que al igual que A* la función heurística no ayuda a acelerar la búsqueda del objetivo, porque en el mapa hay caminos que se acercan al objetivo, pero dichos caminos terminan en callejones sin salida. Cuando un algoritmo de búsqueda de camino termina buscando en un callejón sin salida, a continuación, empieza a explorar las celdas en otra sección del mapa. En el peor de los casos, el algoritmo explora todas las celdas del mapa, aumentando el tiempo de computación de la solución.

Para este mapa el algoritmo codicioso explora casi todas las celdas del mapa. Observe la *Figura 51*. Las celdas de color oscuro corresponden a obstáculos,

mientras que las celdas coloreadas de azul y verde son celdas que ha explorado el algoritmo para computar la solución. La línea en amarillo es el camino encontrado por el algoritmo codicioso.

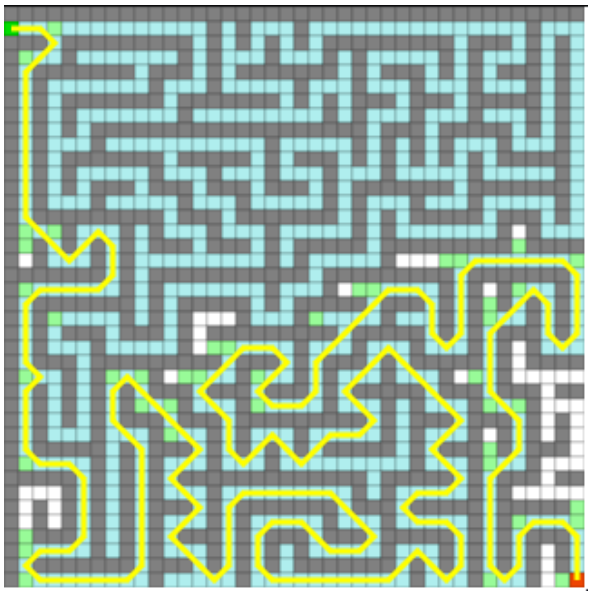


Figura 51. Visualización del algoritmo Best First Search (codicioso) encontrando el camino más corto desde la esquina superior izquierda hacia la esquina inferior derecha. Las celdas azules corresponden a las celdas exploradas. Nótese que se exploró casi todo el mapa.

JPS también busca en casi todo el mapa, pero termina siendo más eficiente debido a que es capaz de encontrar varios puntos de salto, ahorrando pasos.

Para todos los mapas de este tipo el algoritmo codicioso encontró el camino óptimo, pero esto es debido a que no hay caminos alternativos para llegar al objetivo.

Al igual que el algoritmo codicioso, la heurística utilizada en A* no es muy útil debido a que el mapa contiene varios callejones sin salida. Por lo tanto, A* busca en casi todo el mapa, comportándose de manera similar al algoritmo de Dijkstra.

BFS también busca en casi todo el mapa, pero su complejidad temporal le da el beneficio de que le tome menos tiempo explorar una celda estando a la par de JPS.

MAPA DE MAKALIWE

El siguiente tipo de mapa intenta replicar el mapa diseñado por Makaliwe en su artículo de investigación *Automatic planning of nanoparticle assembly tasks*. En dicho artículo Makaliwe utiliza algoritmos de búsqueda camino y un microscopio electrónico

de barrido para desplazar nanopartículas de manera automatizada hacia varios puntos designados tomando el camino más corto [9].

Una ilustración del diseño del mapa se muestra en la *Figura 52*. El punto de inicio está marcado en verde mientras que el objetivo está marcado en rojo.

Figura 52. Diseño del mapa de Makaliwe. La marca verde indica el punto de inicio, mientras que la marca roja el objetivo.

La intención con este mapa es observar lo que sucede cuando inicialmente el camino óptimo se aleja del objetivo.

Se utilizaron mapas de 20x20, 40x40, 60x60, 80x80, 120x120, 140x140, 160x160, 180x180 y 200x200 y para cada uno se ejecutó cada algoritmo con el mismo punto de inicio y objetivo.

En la *Figura 53* se presentan los resultados.

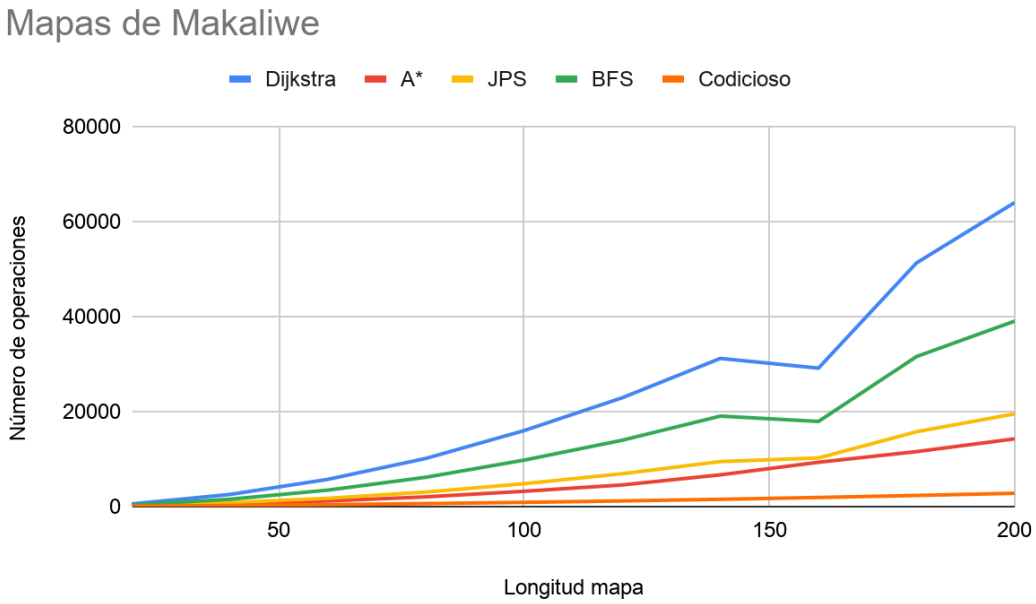


Figura 53. Resultados de búsqueda de camino más corto en *mapas de Makaliwe*. El algoritmo más lento es algoritmo de Dijkstra, seguido de BFS, JPS, A* y finalmente el algoritmo codicioso.

Se observa que Dijkstra es el algoritmo más lento, seguido por BFS. Ambos algoritmos no poseen una función heurística, por lo cual terminan realizando su búsqueda hacia todas direcciones. Aunque el número de celdas que exploran ambos es similar, BFS le toma menos operaciones debido a que tiene una complejidad temporal menor.

Por otro lado, el resto de algoritmos que utilizan funciones heurísticas son más rápidos por un gran margen, indicando que la heurística es útil en algún punto. Inicialmente cuando un algoritmo con heurística intenta salir de la estructura de obstáculos con forma de C, tiende a comportarse como el algoritmo de Dijkstra explorando varios nodos adicionales. Pero una vez fuera, el camino óptimo mantiene su dirección hacia el objetivo y de este modo el algoritmo se beneficia de su heurística.

En la *Figura 54* se muestra una comparación visual entre el número de celdas exploradas por el algoritmo de Dijkstra y A*.

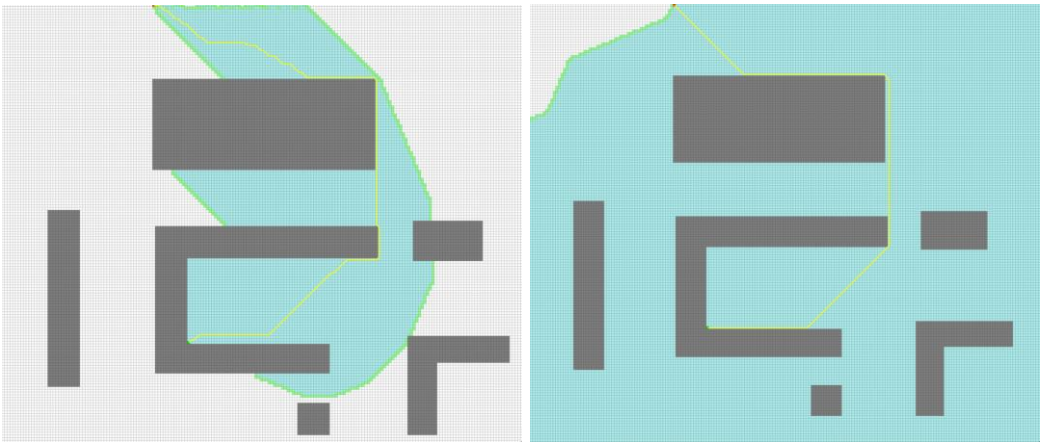


Figura 54. Comparación visual del número de nodos explorados por el algoritmo de Dijkstra y A* en un *mapa Makaliwe*.

En la ilustración se observa que en un principio A* expande hacia todos lados, pero una vez fuera de la estructura en forma de C, A* se beneficia de su heurística, mientras que el algoritmo de Dijkstra se expande hacia todas direcciones en todo momento explorando celdas adicionales.

A* y JPS están a la par, siendo JPS ligeramente más lento porque el espacio es abierto y no hay muchos obstáculos provocando que la búsqueda de vecinos forzados llegue varias veces al borde del mapa.

El algoritmo codicioso es el más rápido expandiendo el menor número de celdas. A diferencia de los otros algoritmos, el algoritmo codicioso toma un camino distinto. El camino no es el óptimo y la aproximación que brinda es diferente alrededor de 20 unidades de longitud en los mapas más grandes. Una comparación se muestra en la *Figura 55*.

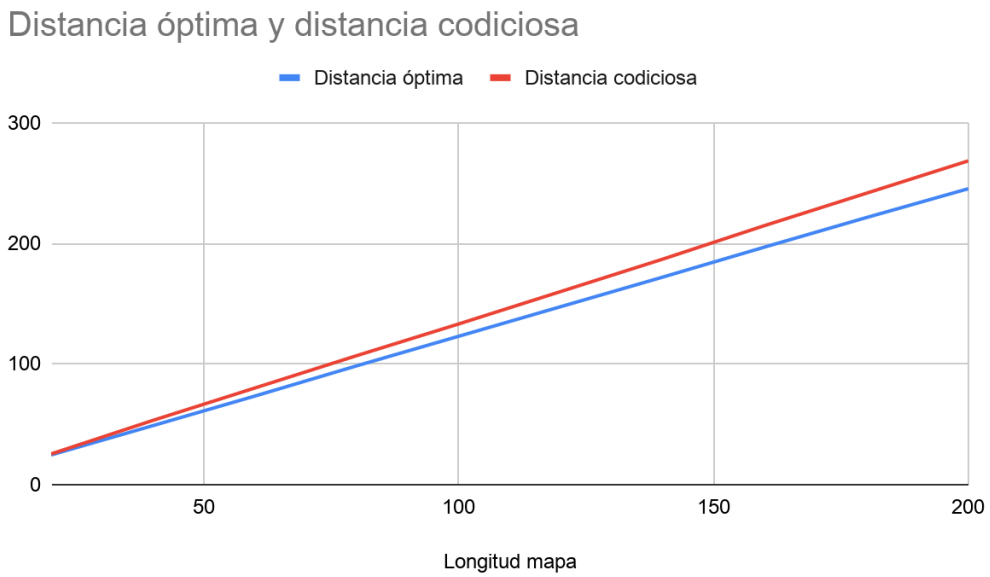


Figura 55. Comparación entre la distancia del camino más corto y la distancia del camino encontrado por el algoritmo codicioso en los *mapas de Makaliwe*.

Para espacios abiertos donde hay una variedad de caminos que conducen hacia el objetivo, el algoritmo codicioso suele ser más rápido. Debido a que inicialmente el camino óptimo requiere ir en dirección contraria del objetivo, esto ocasiona que el algoritmo empiece a encontrar un camino lejano de lo óptimo. Comparado con el *mapa de cruces* y *mapa de cuadrados* dónde el camino más corto era casi una línea recta y la diferencia entre la solución encontrada por el algoritmo codicioso y otros algoritmos que aseguran encontrar el camino óptimo es despreciable.

Parte V

CONCLUSIONES

En este trabajo se realizó una investigación sobre algoritmos de búsqueda de camino en entornos 2D con el objetivo de señalar qué algoritmo es preferible usar respecto a otro dependiendo del mapa. Dichos algoritmos tienen potenciales aplicaciones en nanotecnología en el ensamblaje automatizado de microrobots o en sistemas de navegación automatizados de nanomáquinas para el transporte de fármacos de manera autónoma.

Cuando se decide qué algoritmo utilizar, se debe esclarecer lo que se considera prioritario. Si es importante que la solución sea óptima lo mejor es utilizar JPS o BFS si el mapa tiene varios obstáculos, o A* si hay pocos obstáculos. En cambio, si no es importante que la solución sea el camino más corto, Best First Search (codicioso) es el algoritmo ideal.

Parte VI

EPÍLOGO

ANEXO

PSEUDOCÓDIGO DEL ALGORITMO DE DIJKSTRA

```
1. funcion DIJKSTRA(grafo, inicio, objetivo)
2.     para cada v de grafo.V hacer
3.       v <- ∞
4.     fin para
5.     lista_cerrada <- ∅
6.     lista_abierta <- ∅
7.     inicio.predecesor <- nulo
8.     inicio.G <- 0
9.     lista_abierta.adicionar(inicio)
10.    repetir:
11.      si lista_abierta = ∅ entonces regresar error
12.      v <- lista_abierta.extraer_minimo() //Extrae el
   vértice en la lista abierta con el valor G más pequeño
13.      lista_cerrada.adicionar(v)
14.      si v = objetivo entonces romper ciclo
15.      para cada sucesor s de v hacer
16.        si s está en la lista cerrada entonces continuar
17.        G <- v.G + costo(v, s) //Estimando valor G
18.        si s no está en la lista abierta entonces
19.           s.G <- G
20.           lista_abierta.adicionar(s)
21.           s.predecesor <- v
22.        si no si G < s.G entonces
23.           s.G <- G
24.           s.predecesor <- v
25.      fin para
26.    fin repetir
27.    camino <- ∅
28.    n <- objetivo
29.    mientras n no es nulo hacer
30.      camino.adicionar(n)
31.      n <- n.predecesor
32.    fin mientras
33.    camino.invertir()
34.    regresar camino
```

PSEUDOCÓDIGO DEL ALGORITMO A*

```
1. funcion A_ESTRELLA(grafo, inicio, objetivo)
2.     para cada v de grafo.V hacer
3.       v <- ∞
4.     fin para
5.     lista_cerrada <- ∅
6.     lista_abierta <- ∅
7.     inicio.predecesor <- nulo
8.     inicio.G <- 0
9.     inicio.H <- HEURISTICA_OCTILE(inicio, objetivo)
10.    inicio.F <- inicio.G + inicio.H
11.    lista_abierta.adicionar(inicio)
12.    repetir:
13.      si lista_abierta = ∅ entonces regresar error
14.      v <- lista_abierta.extraer_minimo() //Extrae el
   vértice en la lista abierta con el valor F más pequeño
15.      lista_cerrada.adicionar(v)
16.      si v = objetivo entonces romper ciclo
17.      para cada sucesor s de v hacer
18.        si s está en la lista cerrada entonces continuar
19.        G <- v.G + costo(v, s) //Estimando valor G
20.        H <- HEURISTICA_OCTILE(s, objetivo)
21.        F <- G + H
22.        si s no está en la lista abierta entonces
23.           s.G <- G
24.           s.H <- H
25.           s.F <- F
26.           lista_abierta.adicionar(s)
27.           s.predecesor <- v
28.        si no si G < s.G entonces
29.           s.G <- G
30.           s.H <- H
31.           s.F <- F
32.           s.predecesor <- v
33.      fin para
34.    fin repetir
35.    camino <- ∅
36.    n <- objetivo
37.    mientras n no es nulo hacer
38.      camino.adicionar(n)
39.      n <- n.predecesor
40.    fin mientras
41.    camino.invertir()
42.    regresar camino
```

```
1. funcion HEURISTICA_OCTILE(vertice, objetivo)
2.   x <- vertice.x //Coordenada x del vértice
3.   y <- vertice.y //Coordenada y del vértice
4.   x_objetivo <- objetivo.x //Coordenada x del objetivo
5.   y_objetivo <- objetivo.y //Coordenada y del objetivo
6.   dx <- abs(vertice.x - x_objetivo) //abs es el valor
   absoluto
7.   dy <- abs(vertice.y - y_objetivo)
8.   si (dx > dy) entonces
9.       H <- (dx - dy) + sqrt(2) * dy //Calculando valor H
10.  si no entonces
11.      H <- (dy - dx) + sqrt(2) * dx //Calculando valor H
12.  regresar H
```

PSEUDOCÓDIGO DE JUMP POINT SEARCH

```
1. funcion A_ESTRELLA(grafo, inicio, objetivo)
2.     para cada v de grafo.V hacer
3.       v <- ∞
4.     fin para
5.     lista_cerrada <- ∅
6.     lista_abierta <- ∅
7.     inicio.predecesor <- nulo
8.     inicio.G <- 0
9.     inicio.H <- HEURISTICA_OCTILE(inicio, objetivo)
10.    inicio.F <- inicio.G + inicio.H
11.    lista_abierta.adicionar(inicio)
12.    repetir:
13.      si lista_abierta = ∅ entonces regresar error
14.      v <- lista_abierta.extraer_minimo() //Extrae el
   vértice en la lista abierta con el valor F más pequeño
15.      lista_cerrada.adicionar(v)
16.      si v = objetivo entonces romper ciclo
17.      sucesores <- OBTENER_SUCESORES(v, inicio, objetivo)
18.      para cada elemento s de la lista de sucesores hacer
19.        si s está en la lista cerrada entonces continuar
20.        G <- v.G + costo(v, s) //Estimando valor G
21.        H <- HEURISTICA_OCTILE(s, objetivo)
22.        F <- G + H
23.        si s no está en la lista abierta entonces
24.           s.G <- G
25.           s.H <- H
26.           s.F <- F
27.           lista_abierta.adicionar(s)
28.           s.predecesor <- v
29.        si no si G < s.G entonces
30.           s.G <- G
31.           s.H <- H
32.           s.F <- F
33.           s.predecesor <- v
34.      fin para
35.    fin repetir
36.    camino <- ∅
37.    n <- objetivo
38.    mientras n no es nulo hacer
39.      camino.adicionar(n)
40.      n <- n.predecesor
41.    fin mientras
42.    camino.invertir()
43.    regresar camino
```

```
1. funcion OBTENER_SUCESORES(vertice, inicio, objetivo)
2.   sucesores <- ∅
3.   vecinos <- vertice.vecinos //vértices adyacentes al
   vértice
4.   para cada vecino en la lista vecinos hacer
5.     // Obteniendo direccion del vértice hacia el vecino
6.     si (vecino.x – vertice.x < 1) entonces
7.        dx = -1
8.     si no si (vecino.x - vertice.x > 1) entonces
9.        dx = 1
10.    si no entonces
11.       dx = vecino.x - vertice.x
12.    si (vecino.y – vertice.y < 1) entonces
13.       dy = -1
14.    si no si (vecino.y - vertice.y > 1) entonces
15.       dy = 1
16.    si no entonces
17.       dy = vecino.y - vertice.y
18.       //Buscando un vértice al cual saltar
19.    punto_salto <- SALTO(vertice.x, vertice.y, dx, dy,
   inicio, objetivo)
20.    //Si hay un punto de salto, añadirlo a la lista
21.    si punto_salto no es nulo entonces
22.       sucesores.adicionar(punto_salto)
23.  regresar sucesores
```

```
1. function SALTO(vx, vy, dx, dy, inicio, objetivo)
2.   // vx, vy - Coordenadas del vértice actual dx,dy - dirección
3.   //salto es un vértice en la cuadrícula como posible punto de salto
4.   salto.x <- vx + dx
5.   salto.y <- vy + dy
6.   si salto es un obstáculo en la cuadrícula entonces
7.      regresar NULO
8.   si salto.x = objetivo.x y salto.y = objetivo.y entonces
9.      regresar salto
10.  //Caso diagonal
11.  si dx ≠ 0 y dy ≠ 0
12.     si hay vecinos forzados para dirección en diagonal entonces
13.        regresar salto
14.  //Revisando vecinos forzados de manera horizontal y vertical
15.  //Este caso es especial para dirección diagonal
16.     si SALTO(salto.x, salto.y, dx, 0, inicio, objetivo) no es NULO o SALTO(salto.x, salto.y, 0, dy, inicio, objetivo) no es NULO entonces
17.        regresar salto
18.  si no
19.     //Caso horizontal
20.     si dx ≠ 0
21.        si hay vecinos forzados para dirección horizontal entonces
22.           regresar salto
23.     //Caso vertical
24.     si no
25.        si hay vecinos forzados para dirección vértical
26.           regresar salto
27.     //Si no se encontraron vecinos forzados probar con otro punto de salto
28.  regresar SALTO(salto.x, salto.y, dx, dy, inicio, objetivo)
```

PSEUDOCÓDIGO DEL ALGORITMO BREADTH FIRST SEARCH

```
1.  funcion BFS (grafo, inicio, objetivo)
2.      para cada v de grafo.V hacer
3.        v.distancia <- ∞
4.        v.explorado <- falso
5.      fin para
6.      inicio.explorado <- verdad
7.      inicio.predecesor <- nulo
8.      inicio.distancia <- 0
9.      Q <- Una cola vacía
10.     Q.encolar(inicio)
11.     mientras Q ≠ ∅ hacer
12.       v <- Q.decolar()
13.       si v = objetivo entonces romper ciclo
14.       para cada sucesor s de v hacer
15.            si s.explorado ≠ verdad entonces
16.                 s.explorado <- verdad
17.                 s.predecesor <- v
18.                 si s es diagonal a v entonces
19.                      s.distancia <- v.distancia + √2
20.                 si no entonces
21.                      s.distancia <- v.distancia + 1
22.                 Q.encolar(s)
23.       fin para
24.     fin mientras
25.     camino <- ∅
26.     n <- objetivo
27.     mientras n no es nulo hacer
28.       camino.adicionar(n)
29.       n <- n.predecesor
30.     fin mientras
31.     camino.invertir()
32.     regresar camino
```

PSEUDOCÓDIGO DEL ALGORITMO BEST FIRST SEARCH (CODICIOSO)

```
1. funcion CODICIOSO(grafo, inicio, objetivo)
2.     para cada v de grafo.V hacer
3.       v <- ∞
4.     fin para
5.     lista_cerrada <- Ø
6.     lista_abierta <- Ø
7.     inicio.predecesor <- nulo
8.     inicio.H <- HEURISTICA_OCTILE(inicio, objetivo)
9.     lista_abierta.adicionar(inicio)
10.    repetir:
11.      si lista_abierta = ∅ entonces regresar error
12.      v <- lista_abierta.extraer_minimo() //Extrae el
   vértice en la lista abierta con el valor H más pequeño
13.      lista_cerrada.adicionar(v)
14.      si v = objetivo entonces romper ciclo
15.      para cada sucesor s de v hacer
16.        si s está en la lista cerrada entonces continuar
17.        H <- HEURISTICA_OCTILE(s, objetivo)
18.        si s no está en la lista abierta entonces
19.           s.H <- H
20.           lista_abierta.adicionar(s)
21.           s.predecesor <- v
22.      fin para
23.    fin repetir
24.    camino <- Ø
25.    n <- objetivo
26.    mientras n no es nulo hacer
27.      camino.adicionar(n)
28.      n <- n.predecesor
29.    fin mientras
30.    camino.invertir()
31.    regresar camino
```

BIBLIOGRAFÍA

1. Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. *Introduction to Algorithms*, 3rd ed.; The MIT Press: Cambridge, Massachusetts, 2009; pp 5, 27, 57, 74, 589–592, 643, 661-662.
2. Bellman, R. On a Routing Problem. *Quarterly of Applied Mathematics* **1958**, *16* (1), 87–90. https://doi.org/10.1090/qam/102435.
3. Zhang, F.; Xia, R.; Chen, X. An Optimal Trajectory Planning Algorithm for Autonomous Trucks: Architecture, Algorithm, and Experiment. *International Journal of Advanced Robotic Systems* **2020**, *17* (2), 172988142090960. https://doi.org/10.1177/1729881420909603.
4. Heping Chen; Ning Xi; Guangyong Li; Jiangbo Zhang; Prokos, M. Planning and Control for Automated Nanorobotic Assembly. *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*. https://doi.org/10.1109/robot.2005.1570114.
5. Sabra, W.; Khouzam, M.; Chanu, A.; Martel, S. Use of 3D Potential Field and an Enhanced Breadth-First Search Algorithms for the Path Planning of Microdevices Propelled in the Cardiovascular System. *2005 IEEE Engineering in Medicine and Biology 27th Annual Conference* **2005**. https://doi.org/10.1109/iembs.2005.1615318.
6. Patil, A.; Patil, S.; Manickam, P. Identification of Lung Cancer Related Genes Using Enhanced Floyd Warshall Algorithm in a Protein to Protein Interaction Network. *International Journal of Intelligent Engineering and Systems* **2018**, *11* (3), 215–222. https://doi.org/10.22266/ijies2018.0630.23.
7. Fink, W.; Baker, V. R.; Brooks, A. J.-W.; Flammia, M.; Dohm, J. M.; Tarbell, M. A. Globally Optimal Rover Traverse Planning in 3D Using Dijkstra's Algorithm for Multi-Objective Deployment Scenarios. *Planetary and Space Science* **2019**, 104707. https://doi.org/10.1016/j.pss.2019.104707.
8. Mardana, H.; Maharani, S.; Hatta, H. R. Applications to Determine the Shortest Tower BTS Distance Using Dijkstra Algorithm. **2017**. https://doi.org/10.1063/1.4975967.
9. Carsten, J.; Rankin, A.; Ferguson, D.; Stentz, A. Global Path Planning on Board the Mars Exploration Rovers. *2007 IEEE Aerospace Conference* **2007**. https://doi.org/10.1109/aero.2007.352683.
10. Belharet, K.; Folio, D.; Ferreira, A. Three-Dimensional Controlled Motion of a Microrobot Using Magnetic Gradients. *Advanced Robotics* **2011**, *25* (8), 1069–1083. https://doi.org/10.1163/016918611x568657.

11. Meng, K.; Jia, Y.; Yang, H.; Niu, F.; Wang, Y.; Sun, D. Motion Planning and Robust Control for the Endovascular Navigation of a Microrobot. *IEEE Transactions on Industrial Informatics* **2020**, *16* (7), 4557–4566. https://doi.org/10.1109/tii.2019.2950052.
12. Belharet, K.; Folio, D.; Ferreira, A. MRI-Based Microrobotic System for the Propulsion and Navigation of Ferromagnetic Microcapsules. *Minimally Invasive Therapy & Allied Technologies* **2010**, *19* (3), 157–169. https://doi.org/10.3109/13645706.2010.481402.
13. Chang, Y.; Wang, X.; An, Z.; Wang, H. Robotic Path Planning Using A* Algorithm for Automatic Navigation in Magnetic Resonance Angiography. *2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)* **2018**. https://doi.org/10.1109/embc.2018.8512417.
14. Loukanov, A.; Gagov, H.; Nakabayashi, S. Artificial Nanomachines and Nanorobotics. *The Road from Nanomedicine to Precision Medicine* **2019**, 515–532. https://doi.org/10.1201/9780429295010-14.
15. Li, J.; Esteban-Fernández de Ávila, B.; Gao, W.; Zhang, L.; Wang, J. Micro/Nanorobots for Biomedicine: Delivery, Surgery, Sensing, and Detoxification. *Science Robotics* **2017**, *2* (4), eaam6431. https://doi.org/10.1126/scirobotics.aam6431.
16. Makaliwe, J. H.; Requicha, A. A. G. Automatic Planning of Nanoparticle Assembly Tasks. *Proceedings of the 2001 IEEE International Symposium on Assembly and Task Planning (ISATP2001). Assembly and Disassembly in the Twenty-first Century. (Cat. No.01TH8560)* **2001**. https://doi.org/10.1109/isatp.2001.929037.
17. Ning, X.; Guangyong, L. *Introduction to Nanorobotic Manipulation and Assembly*; Artech House Publishers: Norwood, 2011; pp 246–253.
18. Fionda, V. Networks in Biology. *Encyclopedia of Bioinformatics and Computational Biology* **2019**, 915–921. https://doi.org/10.1016/b978-0-12-809633-8.20420-2.
19. Sharan, R.; Ulitsky, I.; Shamir, R. Network-Based Prediction of Protein Function. *Molecular Systems Biology* **2007**, *3*. https://doi.org/10.1038/msb4100129.
20. Li, B.-Q.; You, J.; Chen, L.; Zhang, J.; Zhang, N.; Li, H.-P.; Huang, T.; Kong, X.-Y.; Cai, Y.-D. Identification of Lung-Cancer-Related Genes with the Shortest Path Approach in a Protein-Protein Interaction Network. *BioMed Research International* **2013**, *2013*, 1–8. https://doi.org/10.1155/2013/267375.
21. STRING: functional protein association networks https://string-db.org/.

22. Li, M.; Guo, Y.; Feng, Y.-M.; Zhang, N. Identification of Triple-Negative Breast Cancer Genes and a Novel High-Risk Breast Cancer Prediction Model Development Based on PPI Data and Support Vector Machines. *Frontiers in Genetics* **2019**, *10*. https://doi.org/10.3389/fgene.2019.00180.
23. Cai, Y.-D.; Zhang, Q.; Zhang, Y.-H.; Chen, L.; Huang, T. Identification of Genes Associated with Breast Cancer Metastasis to Bone on a Protein–Protein Interaction Network with a Shortest Path Algorithm. *Journal of Proteome Research* **2017**, *16* (2), 1027–1038. https://doi.org/10.1021/acs.jproteome.6b00950.
24. Li, B.-Q.; Huang, T.; Liu, L.; Cai, Y.-D.; Chou, K.-C. Identification of Colorectal Cancer Related Genes with MRMR and Shortest Path in Protein-Protein Interaction Network. *PLoS ONE* **2012**, *7* (4), e33393. https://doi.org/10.1371/journal.pone.0033393.
25. CHEN, C.; SHEN, H.; ZHANG, L.-G.; LIU, J.; CAO, X.-G.; YAO, A.-L.; KANG, S.-S.; GAO, W.-X.; HAN, H.; CAO, F.-H.; et al. Construction and Analysis of Protein-Protein Interaction Networks Based on Proteomics Data of Prostate Cancer. *International Journal of Molecular Medicine* **2016**, *37* (6), 1576–1586. https://doi.org/10.3892/ijmm.2016.2577.
26. Jiang, Y.; Shu, Y.; Shi, Y.; Li, L.-P.; Yuan, F.; Ren, H. Identifying Gastric Cancer Related Genes Using the Shortest Path Algorithm and Protein-Protein Interaction Network https://www.hindawi.com/journals/bmri/2014/371397/ (accessed Sep 19, 2020).
27. Shen, S.; Gui, T.; Ma, C. Identification of Molecular Biomarkers for Pancreatic Cancer with MRMR Shortest Path Method. *Oncotarget* **2017**, *8* (25), 41432–41439. https://doi.org/10.18632/oncotarget.18186.
28. Salem, A. An Easy-To-Use Guide to Big-O Time Complexity https://medium.com/@ariel.salem1989/an-easy-to-use-guide-to-big-o-time-complexity-5dcf4be8a444 (accessed Jun 4, 2020).
29. Trudeau, R. J. *Introduction to Graph Theory*, 2nd Revised ed.; Dover Publications, 1994; pp 27–29.
30. Garg, P. Heaps and Priority Queues - Prateek Garg https://www.hackerearth.com/practice/notes/heaps-and-priority-queues/.
31. Dijkstra, E. W. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik* **1959**, *1* (1), 269–271. https://doi.org/10.1007/bf01386390.

32. Hart, P.; Nilsson, N.; Raphael, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* **1968**, *4* (2), 100–107. https://doi.org/10.1109/tssc.1968.300136.
33. Romanycia, M. H. J.; Pelletier, F. J. What Is a Heuristic? *Computational Intelligence* **1985**, *1* (1), 47–58. https://doi.org/10.1111/j.1467-8640.1985.tb00058.x.
34. Patel, A. Heuristics http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html (accessed Sep 15, 2020).
35. Dechter, R.; Pearl, J. Generalized Best-First Search Strategies and the Optimality of A*. *Journal of the ACM* **1985**, *32* (3), 505–536. https://doi.org/10.1145/3828.3830.
36. Strand-Holm, A.; Strand-Holm, M. Pathfinding in Two-Dimensional LWorlds. Master's Thesis, Aarhus University, 2015.
37. Harabor, D.; Grastien, A. Online Graph Pruning for Pathfinding on Grid Maps. *AAAI'11: Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence* **2011**, 1114–1119.
38. Witmer, N. A Visual Explanation of Jump Point Search https://zerowidth.com/2013/a-visual-explanation-of-jump-point-search.html (accessed Aug 14, 2020).
39. K. Zuse.Der plankalkül **1972**, Konrad Zuse Internet Archive, 96–105. http://zuse.zib.de/item/gHI1cNsUuQweHB6.
40. Meijer, K. Maze generator https://keesiemeijer.github.io/maze-generator/ (accessed Oct 6, 2020).