UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Búsqueda bidireccional aplicada al *Another Solution Problem*

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Jose Luis Martín Navarro

Tutor: Eva Onaindía de la Rivaherrera

Curso 2018-2019

Resum

El *Shortest Path Problem* és un dels problemes de cerca de camins més estudiats en la literatura, raó per la qual altres problemes com l'*Another Solution Problem* (ASP) han guanyat interés. L'ASP s'enuncia com el problema en el qual, donada una solució òptima, l'objetiu és trobar la següent solució òptima. En aquest projecte es proposa resoldre el problema mitjançant l'aplicació de tècniques de cerca bidireccional de manera que el punt en el qual s'encontren els dos processos de cerca, començant un pel vèrtex inicial i l'altre pel vèrtex final, determina el segon camí òptim.

Paraules clau: algoritmes de cerca, grafs, PCC, ASP

Resumen

El *Shortest Path Problem* (SPP) es uno de los problemas de búsqueda más estudiados en la literatura, razón por la que han cobrado interés otros problemas relacionados como el *Another Solution Problem* (ASP). El ASP se enuncia como el problema en el que, dada una solución óptima, el objetivo es encontrar la siguiente solución óptima. En este proyecto se propone abordar el problema mediante la aplicación de técnicas de búsqueda bidireccional de modo que el punto en el que se encuentren los dos procesos de búsqueda, comenzando uno por el vértice inicial y otro por el vértice final, determinará el segundo camino óptimo.

Palabras clave: algoritmos de búsqueda, grafos, PCC, ASP

Abstract

The *Shortest Path Problem* (SPP) is a well-known and studied problem in the literature. Originated from this problem, other interesting and related problems have emerged like the *Another Solution Problem* (ASP). Given a problem and its optimal solution the ASP lies in finding the subsequent optimal solution. There exist different techniques to address the ASP but most of them present several limitations due to the exploration of unnecessary paths or an unsucessful search. In this project we propose to address the ASP by using techniques of bidirectional search, which consist in starting two search process, from the initial and final nodes, respectively, such that the node in which both searches find will determine the second best solution.

Key words: search algorithms, graph search, SPP, ASP

Índice general

Índice de figuras

Índice de tablas

Índice de algoritmos

CAPÍTULO 1
Introducción

Los algoritmos de búsqueda en grafos han recibido una gran atención durante toda la historia de la informática. Una de las primeras formulaciones la propuso Richard Bellman en 1958 [1] y la investigación continúa hasta nuestros días. La razón de este interés viene motivada tanto por los retos que plantea como por la multitud de sus aplicaciones. Problemas en campos tan diferentes como visión por computador [29, 27], procesamiento de lenguaje natural [34], control de tráfico [13] y logística [25] se han resuelto con algoritmos de búsqueda en grafos.

Uno de los problemas más estudiados en la teoría de grafos es la búsqueda de caminos en un grafo. La versatilidad que ofrece la modelización de un problema como búsqueda de caminos en grafos permite, en muchos problemas cotidianos, la abstracción sobre el dominio del problema, ayudando a que los procesos de búsqueda se puedan trasladar a multitud de áreas del conocimiento. Así por ejemplo, en el campo de la robótica, la planificación de los movimientos que debe realizar un robot se puede modelar como un problema de búsqueda de caminos en un grafo [7], definiendo los modos de configuración del robot como vértices y los movimientos entre modos como aristas.

Dentro de los problemas de búsqueda en grafos sin duda el que ha recibido más atención es el *Problema del Camino más Corto* (PCC) o *Shortest Path Problem* (SPP) [1] junto con numerosas variantes tales como la Búsqueda Multiagente[11, 10], la Búsqueda Múltiobjetivo[14] o la Búsqueda en entornos sensibles al tiempo [5], entre otras.

Sin embargo, menos esfuerzos han recibido otros problemas de búsqueda de caminos en grafos, como el *Longest Path Problem* (LPP) [37, 42], donde el objetivo es encontrar el camino más largo posible; el *Target-Value Search problem* (TVS) [16] donde el objetivo es encontrar un camino lo más próximo posible a un valor dado; el problema de los *K*-Caminos (*K-Paths*) [6, 45]) donde se pretende encontrar *K* soluciones; o el *Another Solution Problem* (ASP) aplicado al *Problema del Camino más Corto* (ASP-PCC) [1, 31, 44], donde dado un problema y una solución, el objetivo es encontrar otra solución que resuelva el problema. En este trabajo planteamos estudiar e investigar un esquema de resolución para el ASP-PCC.

1.1 Motivación

Como parte de mi estancia en la Universidad Carlos III de Madrid tuve la oportunidad de colaborar el pasado verano con Carlos Linares López, profesor e investigador del grupo de Planificación y Aprendizaje (PLG), que me puso al día de sus últimas investigaciones [20, 19] por si algún tema despertaba mi interés. De su investigación sobre problemas relacionados con el *Problema del Camino más Corto* llamo mi atención el *Another Solution Problem aplicado al Problema del Camino más Corto* (ASP-PCC), por no existir un algoritmo conocido que lo resolviera, únicamente resultados teóricos centrados en su aplicación a la unicidad de soluciones en puzles.

Al comienzo del curso contacté con mi actual tutora Eva Onaindía, que por su carrera en el área de Planificación Automática me parecía la persona idónea para tutorizar mi trabajo, y gracias al apoyo de la cual este trabajo ha salido adelante.

El estudio que se presenta en este Trabajo Fin de Grado versa sobre un problema de búsqueda de caminos en grafos poco estudiado, el *Another Solution Problem* aplicado al *Problema del Camino más Corto* (ASP-PCC). Como el trabajo realizado anteriormente sobre el ASP ha sido exclusivamente de carácter teórico, nos proponemos estudiar las características del problema y realizar una propuesta de solución que se pueda trasladar a un algoritmo de resolución de carácter general.

Aunque la definición de ASP generaliza sobre el tipo de problema al que se puede aplicar, en la actualidad únicamente se ha estudiado su aplicación a problemas de satisfacibilidad en puzles [1, 44, 31]. Por este motivo nos parece interesante ampliar su uso a un problema tan estudiado como es el *Problema del Camino más Corto* para tratar de aplicar los avances en búsqueda de caminos en grafos al ASP-PCC.

Además, es nuestra intención estudiar la aplicación del *Another Solution Problem* como estrategia de resolución para el problema de los *K*-Caminos y algunas de sus variantes: la versión del problema de los *K*-Caminos donde se exige que los caminos sean óptimos (K-OPT) o donde se requiere que las soluciones sean distintas entre sí hasta cierto grado (K-DISJ).

De especial interés es la versión *en vivo* del problema K-OPT, en la que se requiere que la resolución sea del estilo de los *anytime algorithms*, que se detallará en el siguiente capítulo, y que consiste en ir obteniendo soluciones hasta que el usuario cancele el algoritmo. El problema del ASP-PCC puede verse como una instancia de la segunda iteración del K-OPT, el cual es de utilidad en problemas como el propuesto por Han, Katoen y Damman [12], donde buscan contraejemplos para la comprobación de modelos probabilísticos, extensibles a los problemas de toma de decisión *Markov Decision Processes*.

1.2 Objetivos

Los objetivos de este Trabajo Fin de Grado son:

1. Proporcionar un algoritmo de carácter general para resolver el *Another Solution Problem* aplicado al *Problema del Camino más Corto* (ASP-PCC).
2. Realizar, por un lado, un estudio del estado del arte sobre todas las apariciones del ASP y los resultados teóricos obtenidos; y por otro lado, un estudio en materia de algoritmos de búsqueda en grafos, heurísticas y otras técnicas que puedan ser de utilidad a la hora de afrontar el problema.
3. Analizar el ASP-PCC tratando de extraer toda la información de utilidad, con especial atención a las problemáticas de las diferentes aproximaciones con vistas a su resolución.
4. Estudiar el algoritmo de Dijkstra como base sobre la que estructurar nuestra propuesta de resolución, ampliando dicha propuesta para incluir la técnica de búsqueda Bidireccional, aprovechando en todo momento el análisis del problema.
5. Materializar nuestra propuesta en un algoritmo que resuelva el ASP-PCC, presentando su aplicación en una serie de grafos que ayuden a visualizar el problema en diferentes casos de búsqueda de caminos.

1.3 Estructura de la memoria

El documento de la memoria se estructura como sigue. En el Capítulo 2 revisaremos el estado del arte, concretamente dos temas diferenciados; en la sección 2.1 la aparición del ASP en la literatura y en la sección 2.2 las técnicas de búsqueda en búsqueda de caminos.

En el Capítulo 3 especificaremos la notación básica a utilizar (Sección 3.1), definiremos y analizaremos el ASP y su aplicación al *Problema del Camino más Corto* (Secciones 3.2 y 3.3), explicaremos nuestra propuesta de resolución (Sección 3.4) y el algoritmo resultante (Sección 3.5).

En el Capítulo 4 mostraremos los resultados de los algoritmos, por un lado la visualización (Sección 4.1) y por otro la comparativa entre las dos propuestas y una modificación de Dijkstra (Sección 4.2).

Acabaremos la memoria con las conclusiones del trabajo realizado en el Capítulo 5.

CAPÍTULO 2
Revisión del estado del arte

En este capítulo se revisa la literatura sobre algoritmos de búsqueda en relación con el trabajo realizado en este proyecto. La revisión se ha dividido en dos grandes secciones; la sección 2.1 presenta el problema en el que se centra este TFG, el cual se conoce en inglés como *Another Solution Problem* (ASP); y la sección 2.2 resume los principales algoritmos de búsqueda que se han empleado en la búsqueda de caminos en un grafo.

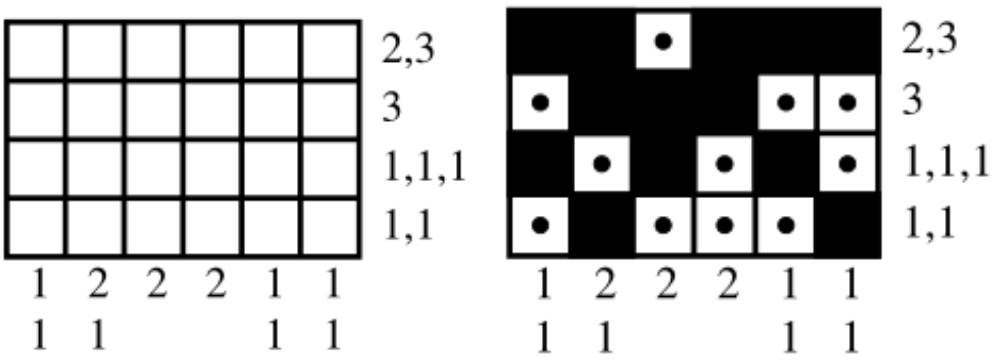
2.1 ¿Qué es el *Another Solution Problem*?

El *Another Solution Problem* (ASP) consiste en encontrar otra solución para un problema dado. Específicamente, dado un problema P y una solución X para P, el ASP es el problema de encontrar una segunda solución para P. Generalmente, la solución X de P satisface una serie de propiedades, por ejemplo que X es la solución óptima para P, de modo que el ASP consiste en encontrar la segunda solución que satisface las mismas propiedades.

El ASP fue introducido como una nueva característica computacional sobre problemas NP-completos por Ueda y Nagao [38], los cuales descubrieron que la definición de ASP permite una mayor caracterización de dichos problemas. Por ejemplo, el problema de satisfacibilidad lógica 3SAT [30] o el problema del coloración de grafos [15] son ambos problemas NP-completos, pero solo el 3SAT sigue siendo NP-completo en su versión ASP, siendo así trivial encontrar otra solución al problema de coloración mediante permutaciones de los colores de la solución.

La continuación de los trabajos de Ueda y Nagao que realizaron Seta [31], y posteriormente Seta con Yato [44] se enfocaron en el estudio de la complejidad de ASP aplicado a problemas NP-completos. Además, estos autores extendieron la definición de ASP al n-ASP, en el cual se pretende resolver el ASP para un problema dado y n de sus soluciones. Uno de los objetivos de estos autores era investigar si la complejidad de encontrar la siguiente solución variaba según el número de soluciones que tuviera el problema.

Las investigaciones realizadas permitieron además relacionar el ASP con el problema de diseño de puzzles, refiriéndose a estos como *pencil puzzles* en el trabajo publicado en [44] (Ueda y Nagao [38] estudian el juego *Nonograma* 2.1, Yato [44] estudia el juego *Sudoku*). Cuando se diseña uno de estos puzzles, se suele



(a) Instancia del problema **(b)** Solución de la instancia

Figura 2.1: Instancia del puzzle Nonograma y solución[38]

exigir que la solución sea única. Según explican los autores, el ASP podría aplicarse para garantizar que dado el diseño de un problema y la solución deseada, no existe otra solución para dicho problema, esto es, la solución proporcionada es única. De este modo, el ASP podría aplicarse a dicho problema para determinar que no existe otra solución. A tal efecto diseñaron un sistema iterativo para diseñar problemas con una solución única:

- Se diseña una solución (ver Figura 2.1b)
- Se deriva una instancia de problema a partir de la solución creada (en puzzles como el *Nonograma* este paso es sencillo – ver ejemplo de instancia de problema del *Nonograma* en la Figura 2.1a)
- Se resuelve el ASP para el problema y la solución diseñados. Si no existe otra solución el proceso acaba. En otro caso se vuelve al punto 1.

Otro enfoque que estudió Yato fue la relación entre el ASP, el n-ASP y el problema de enumeración de soluciones [44]. Gracias a su estudio se pudo determinar una manera de resolver en tiempo polinómico el n-ASP cuando dicho problema tiene un método de enumeración de soluciones en tiempo polinómico. De ello se deduce que la facilidad para enumerar soluciones de un problema afectará a la facilidad de encontrar las siguientes soluciones al problema.

Todos los estudios sobre el ASP son de caracter teórico, y en ellos se estudia la complejidad y completitud según el tipo de problema y sus propiedades teóricas.

Hasta donde nosotros conocemos no hay ningún trabajo previo que resuelva el *Another Solution Problem* aplicado al *Problema del Camino más Corto* (ASP-PCC); esto es, la aplicación de ASP para una entrada de datos que consiste en un problema y una solución que es el camino más corto para dicho problema.

Aunque no hay trabajos que aborden directamente la resolución del ASP-PCC, existen problemas relacionados en los que se han hecho más avances. Este es el caso del problema de los K-caminos [45], donde el objetivo es encontrar los K mejores caminos entre dos vértices, y para el cual se han adapatado algoritmos del *Problema del Camino más Corto* . Si tomamos $K = 2$ se podría encontrar una segunda solución al problema, aunque esto no garantiza que se aproveche la primera solución para ello.

La principal diferencia entre la definición de ASP con la que trabajaban Ueda y Nagao [38], Seta [31] y Seta y Yato [44], y la definición del ASP aplicado al *Problema del Camino más Corto* tiene que ver con la solución proporcionada. La definición de ASP con la que trabajaban los primeros investigadores no especifica si la solución proporcionada que resuelve el problema es óptima o no. La razón es que, en los problemas que estaban estudiando (de tipo puzzle), todas las soluciones tienen el mismo coste. Es por eso que al estudiar el ASP *aplicado al Problema del Camino más Corto* hay que especificar si la solución que se proporciona es óptima o si se considera el caso general, en el que la solución proporcionada no tiene porque ser óptima.

En este TFG hemos decidido trabajar partiendo de la solución óptima, de modo que el trabajo desarrollado se puede considerar como la primera propuesta que aborda el ASP-PCC. En la explicación del algoritmo propuesto comentaremos las diferencias que existen entre resolver el ASP para una solución óptima o para una solución cualquiera.

2.2 Técnicas de Búsqueda

El *Problema del Camino más Corto*, dentro de la familia de problemas de búsqueda de caminos en grafos, ha sido ampliamente estudiado en la literatura, dando lugar a famosos algoritmos como el algoritmo Bellman-Ford, Dijkstra, Primero en Profundidad (*DFS*), Primero en Amplitud (*BFS*) y A*, que se encuentran entre los mejores algoritmos de carácter general que pueden operar sobre un grafo sin necesidad de preprocesarlo antes de realizar la búsqueda.

Un campo de estudio muy activo es el de la aplicación de algoritmos de búsqueda a los videojuegos y la robótica, surgiendo problemas derivados del *Problema del Camino más Corto* como la búsqueda de caminos multiagente *Multi-agent Pathfinding* [10] o el *Problema del Camino más Corto* cuando los pesos de las aristas cambian dependiendo del tiempo, problema que solucionan la familia de algoritmos D* [33].

Los algoritmos de búsqueda y el diseño de heurísticas ha tenido también una gran repercusión en la búsqueda de espacio de estados que se emplea habitualmente en problemas de planificación independientes del dominio, conocido den inglés como *Heuristics and Search for Domain-Independent Planning* [25, 14], aunque muchos problemas y algoritmos son comunes en múltiples campos, como en el caso de la búsqueda de caminos multiagente [11].

A continuación nombramos algunas de las técnicas más utilizadas en el diseño de algoritmos, citando publicaciones novedosas que desarrollan dichas técnicas:

- *Prunning*: técnicas de poda del árbol de búsqueda que valoran cada nodo en función de lo cercanos que sus sucesores son al plan objetivo [22].
- *Backtracking & Backumping*: son técnicas utilizadas en algoritmos que crean de manera incremental el candidato a solución, utilizando *Backtracking* y *Backjumping* para abandonar dicho candidato, volviendo atrás en los estados tomados para explorar otros caminos. La diferencia entre *Baktracking*

y *Backjumping* tiene que ver con la distancia de salto. Mientras que con *Backtracking* es posible explorar todas las soluciones, *Backjumping* reduce el espacio de búsqueda descartando ramas vecinas, remontándose varios niveles en el árbol de búsqueda. Es común en la literatura encontrar los términos *Chronological Backtracking* o *Non-Chronological Baktracking* para referirse a *Backtracking y Backjumping* respectivamente, discutiendo la eficiencia de ambas estrategias dependiendo del problema [23]. Es posible encontrar aplicaciones del *Backtracking* en problemas de todo tipo, como es la atenuación de la señal por la lluvia en estaciones de gran altitud [39] o en la estimación de parámetros de paneles fotovoltaicos [46].

- *Branch and Bound*: algoritmos que intercalan dos fases en la búsqueda, una de exploración *Branch* con una de poda *Bound*, siendo el *Depth-First Branch-and-Bound* (DFBnB) de los algoritmos más novedosos que incorporan esta técnica. El DFBnB es un algoritmo de tipo *anytime* aplicado a problemas de planificación multiobjetivo [5]. Los algoritmos *anytime* son aquellos que devuelven una solución válida aunque se interrumpa el proceso de búsqueda, y son capaces de mejorar la solución encontrada hasta el momento cuando se les asigna más tiempo de procesamiento [47].

- *MonteCarlo Tree*: muestreo aleatorio del espacio de búsqueda para formar una heurística que ayude en la búsqueda [17].

- *Random Walk Planning* [24]: serie de técnicas que basan la búsqueda en el postprocesamiento de caminos aleatorios sobre el espacio de búsqueda [40].

- Técnicas de *Hill Climbing*: pertenecen a la familia de algoritmos búsqueda local, donde a partir de una solución arbitraria del problema intentan encontrar una mejor solución de manera iterativa (aunque solo encuentra el óptimo en problemas convexos). Muchos avances se han realizado en búsquedas locales estocásticas, como explican en esta reciente investigación Luo et al. [21], donde aplican estas técnicas al problema de satisfacibilidad máxima con pesos parciales (WPMS).

- *Contraction Hierarchies*: Gracias al preprocesado de los grafos es posible mejorar el rendimiento de la búsqueda, alterando el espacio de búsqueda con la creación de autovías (*highway-node routing* en inglés), transformando el proceso de búsqueda general en una búsqueda en varias fases, primero entre los nodos con una jerarquía más alta y posteriormente entre nodos que pertenecen a una jerarquía inferior [8].

- Búsqueda Bidireccional: técnica que permite realizar una búsqueda simultáneamente desde el nodo origen y el nodo destino con el objetivo de reducir el espacio de búsqueda que el algoritmo debe recorrer antes de garantizar que la solución es óptima [32, 13]. La búsqueda bidireccional suele utilizarse en combinación con heurísticas por ejemplo en la investigación de Rice y Tsotras [28] estudian la búsqueda bidireccional aplicada al A* añadiendo límites sobe el peor caso esperado, lo que mejora el rendimiento de las heurísticas de A* con una tasa de error baja en problemas prácticos.

De entre todas las técnicas estudiadas hemos elegido la búsqueda bidireccional para incorporarla al algoritmo de resolución general del ASP-PCC. Aunque hay otras técnicas muy interesantes que pueden ofrecer una considerable mejora del rendimiento, uno de los detalles fundamentales del ASP-PCC es garantizar que la siguiente solución encontrada sea la mejor de entre todas las posibles. Si no consideramos la solución proporcionada, esto es equivalente a encontrar la solución óptima, y muchas de las técnicas antes comentadas basan la mejora del rendimiento en sacrificar la optimalidad de la solución.

CAPÍTULO 3
Another Solution Problem

En este capítulo presentamos la definición del *Another Solution Problem* (ASP), las cuestiones que deben tenerse en cuenta en el planteamiento de una solución para el ASP y finalmente describimos la solución propuesta. Este capítulo se organiza en cuatro secciones. La sección 3.1 presenta los conceptos básicos necesarios para la descripción del problema y su solución. La sección 3.2 define formalmente el problema ASP y sus principales características. A continuación, la sección 3.3 introduce la definición del ASP aplicado al *Problema del Camino más Corto* y analiza las limitaciones de diferentes enfoques para resolver este problema. La sección 3.4 describe el esquema de solución propuesto en este TFG y la sección 3.5 presenta dos versiones de un algoritmo de búsqueda que resuelve dicho problema.

3.1 Notación básica

En esta sección definimos todos los elementos que vamos a necesitar para trabajar con el ASP.

Definición 1. (Grafo) — Un grafo G es una estructura compuesta por vértices V y aristas E, $G = \langle V, E \rangle$, donde una arista $e_{i,j} \in E$ está formada por la tupla $e_{i,j} = \langle v_i, v_j, c_{i,j} \rangle$, con $v_i, v_j \in V$ los dos vértices que conecta la arista, y el coste de la misma $c_{i,j} \in \mathbb{N}$ (en su defecto coste unitario).

Un grafo $G = \langle V, E \rangle$ es un grafo no dirigido si $\forall e_{i,j} \in E, \exists\, e_{j,i} \,/\, c_{i,j} = c_{j,i}$. En el caso que no se cumpla esta condición entonces decimos que G es un grafo dirigido. La Figura 3.1 muestra un ejemplo de grafo no dirigido con la notación que seguiremos en el documento.

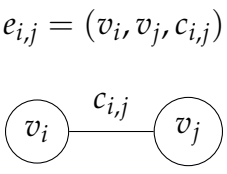


Figura 3.1: Ejemplo de grafo no dirigido

La definición 1 hace referencia a una estructura del grafo explícita. No obstante también es posible trabajar con una definición implícita de grafo, que vendría dada por un vértice inicial, un vértice objetivo y una serie de operadores de transición que generan el grafo.

Definición 2. (Camino) — Dado un grafo $G = \langle V, E \rangle$, un camino π se define como un conjunto de aristas $\pi = \langle e_{0,1}, e_{1,2}, \dots e_{n-1,n} \rangle$ tal que $\forall i$, $0 \leq i \leq n-1$, $e_{i,i+1} = \langle v_i, v_{i+1}, c_{i,i+1} \rangle$, $e_{i,i+1} \in E$ y $v_i, v_{i+1} \in V$.

Utilizaremos la notación dada en la definición 2 para hacer referencia a un camino entre dos vértices v_0 y v_n. El coste de un camino π se calcula como la suma del coste de sus aristas $C_\pi = \sum_{i=0}^{n-1} c_{i,i+1}$.

Dado que pueden existir distintos caminos posibles entre dos vértices cualesquiera, denotaremos Π_{v_0,v_n} al conjunto de todos los caminos existentes entre un vértice inicial v_0 y un vértice final v_n.

Definición 3. (*Camino Óptimo - Camino más Corto*) — Dado un grafo $G = \langle V, E \rangle$, un vértice inicial v_0 y un vértice final v_n, con $v_0, v_n \in V$, se define el camino óptimo o camino más corto entre dos nodos cómo el camino $\pi \in \Pi_{v_0,v_n}$ tal que $\forall \pi_i \in \Pi_{v_0,v_n}$ se cumple $C_\pi \leq C_{\pi_i}$.

En la teoría de grafos, el *Problema del Camino más Corto*, conocido en inglés como el *Shortest Path Problem*, es el problema que consiste en encontrar un camino entre dos vértices de manera que la suma de los pesos de las aristas del camino entre ambos vértices sea mínima. De acuerdo a la definición 3 y a la del problema del camino más corto, podemos equiparar los conceptos de camino óptimo y camino más corto. Únicamente en el caso particular de aristas con coste unitario coincidirá el camino óptimo con el camino de menor longitud.

Dado un camino $\pi = \langle e_{0,1}, e_{1,2}, \dots e_{n-1,n} \rangle$, $\pi \in \Pi_{v_0,v_n}$, definimos subcamino de π a todo camino $\pi_x = \langle e_{i,i+1}, \dots e_{k-1,k} \rangle$ tal que $e_{j,j+1} \in \pi$, $\forall j$, $i \leq j \leq k$.

Dado un camino entre dos vértices v_0 y v_n, $\pi \in \Pi_{v_0,v_n}$, definimos P_π como el conjunto de todos los subcaminos posibles de π.

Sea $\pi \in \Pi_{v_0,v_n}$ el camino óptimo entre los vértices v_0 y v_n; por definición, cualquier subcamino de π, $\pi_x = \langle e_{i,i+1}, \dots, e_{k-1,k} \rangle$, $\pi_x \in P_\pi$ es también un camino óptimo entre los vértices v_i y v_k.

De entre todas las variantes de grafos existentes, en este trabajo tomamos como punto de partida grafos no dirigidos, definidos explícitamente, con una única arista entre dos vértices y con coste de las aristas mayor que cero. Estas asunciones se adoptan en aras de una mayor simplicidad en la notación, no afectando al desarrollo de los algoritmos de este trabajo.

3.2 Definición de ASP

El *problema del camino más corto* (PCC) es un problema que se ha discutido ampliamente en la literatura, surgiendo muchos algoritmos y heurísticas para el mismo. Sin embargo, existen problemas relacionados con el PCC que no han sido estudiados tan exhaustivamente.

El objetivo de este TFG es estudiar el *Another Solution Problem* aplicado al PCC. A continuación se presenta la definición genérica del ASP (Definición 4), tal y como aparece en la publicación original de Ueda y Nagao [38]. Posteriormente, extendemos la definición de ASP a su aplicación concreta al *Problema del Camino más Corto* (Definición 5).

Definición 4. (*Another Solution Problem*) — *For a given NP problem X, ANOTHER SOLUTION PROBLEM for X (ASP for X) is to ask, for a given instance I for X and its solution, whether there is another solution for I* [38].

Definición 5. (*Another Solution Problem aplicado al Problema del Camino más Corto*) — Dada la tupla $\langle G, v_0, v_n, \pi \rangle$ donde G es un grafo, v_0 el vértice inicial, v_n un vértice final, y π el camino óptimo entre v_0 y v_n, $\pi \in \Pi_{v_0,v_n}$, *Another Solution Problem* aplicado al *Problema del Camino más Corto* es el problema que trata de encontrar un camino $\pi' \in \Pi_{v_0,v_n}$ tal que para cualquier otro camino $\pi_i \in \Pi_{v_0,v_n}$ se cumple $C_{\pi'} \leq C_{\pi_i}$ o $\pi' = \pi_i$

3.2.1. ASP aplicado al *Problema del Camino más Corto*

En esta sección analizamos las características y propiedades del ASP cuando se aplica al problema del camino más corto. En lo sucesivo nos referiremos a este nuevo problema como *Another Solution Problem* aplicado al *Problema del Camino más Corto* (ASP-PCC).

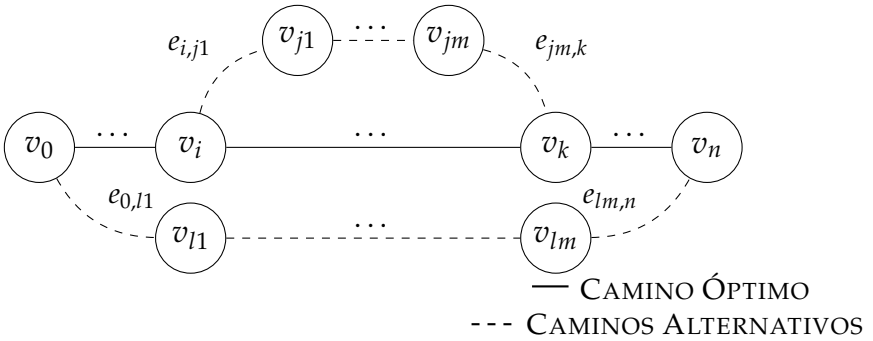


Figura 3.2: Camino óptimo y dos ejemplos de ASP a partir del camino óptimo

Dados dos vértices v_0 y v_n, denominaremos π al camino óptimo entre ambos vértices $\pi = \langle e_{0,\ldots}, \ldots, e_{\ldots,i}, \ldots, e_{k,\ldots}, \ldots, e_{\ldots,n} \rangle$ (véase línea continua en la Figura 3.2). El objetivo del ASP-PCC es encontrar la segunda solución óptima, que denominaremos π'; es decir, encontrar la siguiente solución mejor después de la óptima. A continuación analizamos con la ayuda de la Figura 3.2 los distintos tipos de solución que podemos encontrar:

1. Caminos que comparten aristas con el camino óptimo, bien al inicio, al final o en ambos casos: En el ejemplo de la Figura 3.2 este camino se identifica como $\pi' = \langle e_{0,1}, \ldots, e_{i-1,i}, e_{i,j_1}, \ldots, e_{j_m,k}, e_{k,k+1}, \ldots, e_{n-1,n} \rangle$, que se muestra asimismo en la Figura 3.3.

2. Caminos que no comparten aristas con el camino óptimo: $\pi' \in \Pi_{v_0,v_n}$ $\pi' = \langle e_{0,l_1}, \ldots, e_{l_m,v_n} \rangle$. En la Figura 3.2, es el camino que comienza con la arista e_{0,l_1}

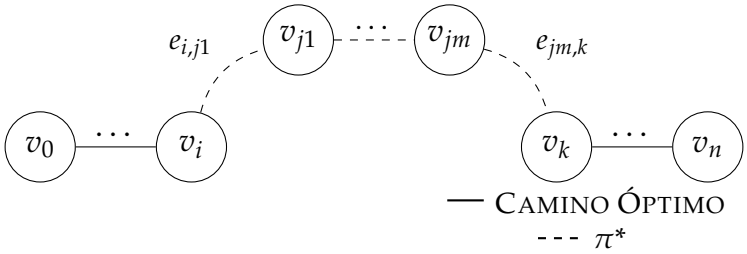


Figura 3.3: Solución óptima de tipo 1, comparte aristas del camino óptimo

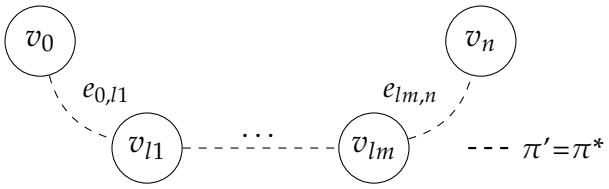


Figura 3.4: Solución óptima de tipo 2, no comparte aristas con el camino óptimo

y finaliza con e_{l_m,v_n}, por debajo del camino óptimo. El camino π' se muestra también en la Figura 3.4.

El caso en el que no se comparte ninguna arista es el peor caso de solución (caso de la Figura 3.4) para el ASP-PCC porque al no compartir ninguna arista con el camino óptimo no es posible utilizar la información de π. Con el fin de abordar el problema de una forma más global, vamos a generalizar los distintos tipos de camino, dando una definición común para ambos casos.

Definimos π' como un camino que empieza en v_0, comparte aristas en el inicio hasta llegar a un vértice v_i, y a partir de v_k vuelve a compartir aristas con el camino óptimo hasta llegar al vértice final v_n (Figura 3.1).

$$\pi' = \langle e_{0,\dots}, e_{i,\dots}, \dots, e_{\dots,k}, \dots, e_{\dots,n} \rangle, \pi' \in \Pi_{v_0,v_n}, 0 \leq i,k \leq n \tag{3.1}$$

Con esta definición de π' tenemos un mismo espacio de búsqueda para todos los tipos de camino, siendo los distintos tipos de solución casos particulares:

1. $v_i \neq v_0$, y/o $v_k \neq v_n$ para los caminos de tipo 1
2. $v_i = v_0, v_k = v_n$ en los caminos de tipo 2

La definición más general de los caminos en ASP-PCC (Ecuación 3.1) permite redefinir la búsqueda del segundo camino óptimo, de modo que en lugar de plantear el problema como la búsqueda de $\pi' \in \Pi_{v_0,v_n}$, lo plantearemos como la búsqueda de un camino $\pi^* \in \Pi_{i,k}$ tal que al completar π^* con los subcaminos de π óptimos entre $v_0 \rightarrow i$ y $k \rightarrow v_n$ hallamos π'. Fijandonos en la Figura 3.2, la definición de la solución del ASP-PCC sería como se muestra a continuación:

$$\pi' = \langle e_{0,\dots}, \pi*, e_{\dots,n} \rangle \tag{3.2}$$

Estos cambios nos permiten plantear la búsqueda del segundo camino óptimo π' como la búsqueda de un camino π^* al que se añaden los subcaminos de π.

3.3 Análisis del *Another Solution Problem* aplicado al *Problema del Camino más Corto*

En esta sección presentamos la propuesta de este TFG para resolver el problema ASP-PCC. Primeramente, se expone la formulación del problema ASP-PCC, y a continuación se exponen varias formas para simplificar el cálculo de las ecuaciones presentadas en la formalización y los problemas que surgen con las simplificaciones planteadas. Seguidamente en la sección 3.4 presentamos nuestra propuesta para resolver el problema ASP-PCC.

3.3.1. Formulación del ASP-PCC

Sea π el camino óptimo entre dos vértices v_0, v_n pertenecientes al grafo G. Resolver el ASP-PCC consiste en encontrar la siguiente solución óptima $\pi' \in \Pi_{v_0.v_n}$ en el mismo grafo G; es decir, encontrar la segunda solución de menor coste.

Utilizaremos la Ecuación 3.1 para denotar los caminos del espacio de búsqueda y, para simplificar la notación en la comparación de caminos, sustituiremos el desglose de caminos en aristas por el de subcaminos, quedando la definición del segundo camino óptimo como aparece a continuación:

$$\pi' = \langle \pi_{v_0,v_i}, \pi^*, \pi_{v_k,v_n} \rangle, 0 \leq i,k \leq n, \quad \pi_{v_0,v_i}, \pi_{v_k,v_n} \in P_\pi$$

De este modo, en lugar de tener que encontrar el camino π' en el espacio de búsqueda Π_{v_0,v_n} buscaremos el subcamino π* en el espacio de búsqueda $\Pi_{i,k} \cap P_\pi$ con $v_i, v_k \in \pi$.

Se puede formular el problema ASP-PCC como un problema de minimización tal que el siguiente camino óptimo sea solución de la siguiente expresión:

$$\min C_{\pi_h} - C_\pi \quad \forall \pi_h \in \Pi_{v_0,v_n} \tag{3.3}$$

Una forma de abordar el ASP-PCC consiste en simplificar alguna de las partes que tienen en común los caminos π y π', pues por la definición de π' que estamos empleando (ver ecuación 3.1) son fácilmente identificables las partes que π y π' tienen en común. En concreto sabemos que se trata de los vértices i y k, los vértices donde π* se separa de π y se vuelve a unir, respectivamente.

$$\pi = \langle e_{0,\dots}, \dots, e_{\dots,i}, e_{i,i+1}, \dots, e_{k-1,k}, e_{k,\dots}, \dots, e_{\dots,n} \rangle = \langle \pi_{v_0,v_i}, \pi_{v_i,v_k}, \pi_{v_k,v_n} \rangle, 0 \leq i,k \leq n \tag{3.4}$$

Utilizando el desglose en subcaminos definido en las ecuaciones de π (Ecuación 3.4) y de π' (Ecuación 3.1), los costes de ambos caminos quedarían como sigue:

$$C_\pi = \sum_{x=0}^{n-1} c_{x,x+1} = \sum_{x=0}^{i-1} c_{x,x+1} + \sum_{y=i}^{k-1} c_{y,y+1} + \sum_{z=k}^{n-1} c_{z,z+1} = C_{\pi_{v_0,v_i}} + C_{\pi_{v_i,v_k}} + C_{\pi_{v_k,v_n}} \tag{3.5}$$

$$C_{\pi'} = \sum_{x'=s}^{n-1} c_{x',x'+1} = \sum_{x=0}^{i-1} c_{x,x+1} + \sum_{y'=i}^{k-1} c_{y',y'+1} + \sum_{z=k}^{n-1} c_{z,z+1} = C_{\pi_{v_0,v_i}} + C_{\pi^*} + C_{\pi_{v_k,v_n}} \tag{3.6}$$

Por último, el problema de minimización de la ecuación 3.3, quedaría como aparece a continuación:

$$\min C_{\pi_h} - C_\pi = \min(C_{\pi_{v_0,v_i}} + C_{\pi_j} + C_{\pi_{v_k,v_n}}) - (C_{\pi_{v_0,v_i}} + C_{\pi_{v_i,v_k}} + C_{\pi_{v_k,v_n}}), \pi_j \in \Pi_{i,k} \tag{3.7}$$

3.3.2. Enfoques para la resolución del ASP-PCC

En esta sección analizaremos diferentes enfoques para simplificar el cálculo de la Ecuación 3.7 que se ha presentado en la sección anterior. La simplificación del cálculo de un tramo del segundo camino óptimo π' que se desea calcular se refiere a la reutilización de dicho tramo tal y como aparece en el camino óptimo π. Existen varias maneras de simplificar la fórmula vista en la Ecuación 3.7 (en todas ellas i,k dependen de cada camino π_h):

1. Simplificando el coste del primer tramo $v_0 \rightarrow v_i$:

$$\min C_{\pi_h} - C_\pi \rightarrow \min (C_{\pi_j} + C_{\pi_{vk,vn}}) - (C_{\pi_{vi,vk}} + C_{\pi_{vk,vn}}) \tag{3.8}$$

   Con esta simplificación se explorarán inicialmente las aristas de menor coste conectadas al camino óptimo, encontrando rápidamente caminos alternativos de bajo coste. En la Figura 3.5 se puede observar varios caminos alternativos: el camino que parte de $v1$ y continúa por v_a, v_b, v_c, v_n, el cual simplifica el tramo $v_0 \rightarrow v_1$; el camino que parte de $v2$ y continúa por v_b, v_c, v_n, el cual simplifica el tramo $v_0 \rightarrow v_2$ y así sucesivamente.

2. Simplificando el último tramo $v_k \rightarrow v_n$:

$$\min C_{\pi_h} - C_\pi \rightarrow \min (C_{\pi_{v_0,v_i}} + C_{\pi_j}) - (C_{\pi_{v_0,v_i}} + C_{\pi_{v_i,v_k}}) \tag{3.9}$$

   Esta simplificación es similar a la anterior pero reutilizando tramos del final del camino óptimo. Con esta simplificación se dejarán de explorar aquellos caminos que se reencuentren con el camino óptimo, sin tener que reexplorar el resto de vértices del camino óptimo hasta el vértice final.

3. Simplificando toda la expresión:

$$\min C_{\pi_h} - C_\pi \rightarrow \min C_{\pi_j} - C_{\pi_{v_i,v_k}} \tag{3.10}$$

   Este enfoque consiste en la combinación de los casos 1 y 2.

La idea principal de estas aproximaciones es evitar la reexpansión de los vértices del camino, ya sea cuando se utiliza el enfoque donde se comparte el tramo

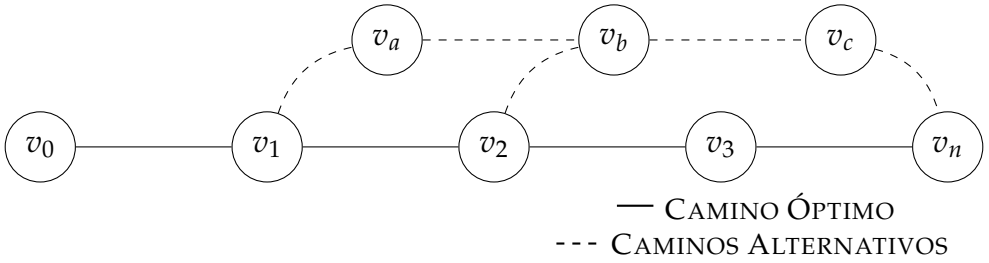


Figura 3.5: Grafo donde es aplicable la simplificación 3.8 ($v_0 \to v_1$ y $v_0 \to v_1 \to v_2$)

al inicio, el final o ambos. Sin embargo, para encontrar la siguiente solución óptima utilizando estás simplificaciones tenemos que explorar todos los subcaminos posibles.

Nuestro objetivo es reducir el número de nodos a expandir para encontrar la solución, asegurándose que al aplicar dicha reducción no se elimina parte del espacio de búsqueda que contiene la solución del problema. Este es uno de los riesgos que podrían suceder con la aplicación de ciertas reducciones de los enfoques expuestos anteriormente. Por ejemplo, en el caso de la simplificación 3.8, podríamos explorar los vértices teniendo en cuenta únicamente su sobrecoste respecto al camino principal. Si aplicáramos esto en el grafo de la figura 3.6, exploraríamos a partir de v_2 sin preocuparnos del tramo $v_0 \to v_1 \to v_2$, ya que sabemos que este tramo es óptimo. De esta manera se expandiría el camino alternativo $v_2 \to v_a \to v_n$. Sin embargo, a pesar de haber encontrado este camino alternativo, finalizar la búsqueda en este punto sin continuar la exploración ignoraría la solución alternativa $v_0 \to v_n$ de coste 4 que aparece con línea punteada en el grafo de la figura 3.6.

Para asegurar que se encuentra la segunda solución óptima debemos continuar la búsqueda del resto de subcaminos. Sin embargo, es necesario comprobar a su vez que el criterio de exploración no expande más caminos de los necesarios. En el mismo grafo (Figura 3.6), si continuamos con la exploración, con el criterio de la simplificación 3.8, después de haber encontrado $v_2 \to v_a \to v_n$, realizaríamos la expansión de v_b desde v_2, y encontraríamos la solución al problema $v_0 \to v_n$. Sin embargo, al hacer esto, hemos expandido v_b, cuyo camino tiene un coste total de cinco, mientras que la segunda solución óptima tiene un coste de cuatro.

Otro factor que puede ser de ayuda para finalizar antes la búsqueda sería conocer el vértice k en el que cada subcamino se reencuentra con el camino óptimo, pues sin conocer este punto no sabemos el coste total de cada camino alternativo y deberíamos continuar la exploración.

3.4 Propuesta de solución

Tras el análisis de los distintos enfoques de simplificación al problema de la búsqueda de la segunda solución óptima, concluimos que considerar tramos comunes con el camino óptimo obliga a explorar todos los posibles subcaminos. En el diseño de nuestra propuesta de solución aplicamos los resultados obtenidos

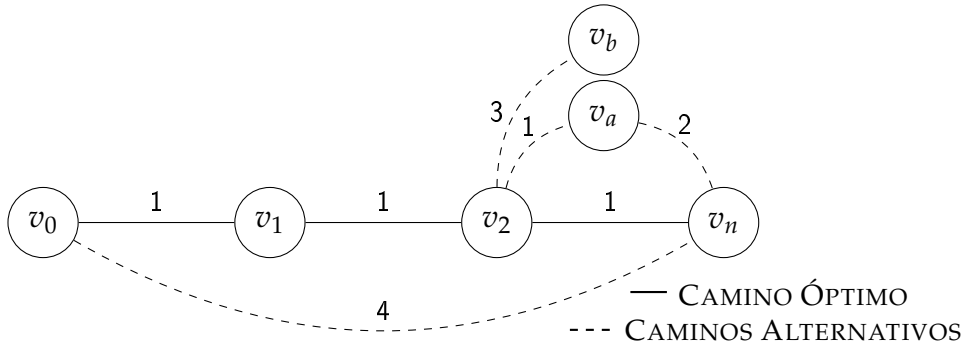


Figura 3.6: Peligros de explorar desde π si el segundo camino no comparte aristas.

del análisis sin dejar de considerar el coste total de cada camino, coincida o no con el camino óptimo.

A continuación presentamos dos propuestas de solución, un esquema de búsqueda unidireccional y un esquema de búsqueda bidireccional. El diseño de este último se construye sobre la base de la búsqueda unidireccional, y se definirán formalmente conceptos propios de la búsqueda bidireccional.

3.4.1. Propuesta Unidireccional

Para nuestra propuesta de solución unidireccional empleamos dos listas, la *lista abierta* y la *lista cerrada*. En la lista abierta se incorporan los nodos abiertos, es decir, aquellos nodos que han sido generados y son susceptibles de ser expandidos. En la lista cerrada incluimos los nodos que se extraen de la lista abierta para ser expandidos. Cuando un nodo se expande se comprueba si es solución del problema; en caso contrario se exploran sus nodos contiguos, los cuales se añaden a la lista abierta, y el nodo expandido se añade a la lista cerrada.

La propuesta de solución se estructura en dos puntos clave, que se explican a continuación:

1. **Recorrido inicial de los vértices de la solución óptima**. Para cada vértice del camino óptimo, $v_i \in \pi$, añadimos a la lista abierta aquellos vértices con los que v_i está conectado por una arista excepto si se trata del siguiente vértice del camino óptimo; esto es, añadimos a la lista abierta $v_j / \exists\, e_{v_i,v_j} \notin \pi$.

   Los vértices que se añaden a la lista abierta tendrán el coste del tramo inicial que comparten con π, C_{v_0,v_i}, y el sobrecoste será el coste de la arista e_{v_i,v_j} (Simplificación 3.8). Como resultado de esta operación tendremos la lista abierta con todos los vértices conectados a los vértices de π con sus costes actualizados según la simplificación 3.8.

   Como ejemplo de este primer paso, en el grafo de la Figura 3.6 tendríamos v_a con coste 2 y sobrecoste 1, v_b con coste 2 y sobrecoste 3, y v_n con coste 0 y sobrecoste 4. Es importante notar que aunque v_n pertenece a π hay que incluirlo si es producto de una arista que cumpla la condición $e_{v_0,v_n} \notin \pi$.

   Recorrer los vértices de π sin añadir estos a la lista abierta nos permitirá realizar la simplificación del tramo final, que se explicará en el siguiente punto.

2. **Expansión iterativa de la lista abierta**. El siguiente paso de nuestra propuesta consiste en expandir los vértices de la lista abierta. Al expandir un vértice añadiremos los vértices contiguos de nuevo a la lista abierta, y este proceso se efectuará de manera iterativa mientras el vértice expandido no cumpla la condición de parada, es decir, mientras el vértice expandido no sea solución del problema.

   Los aspectos claves de este paso son:

   - **Expansión por menor coste total**. Para evitar expandir más caminos de los necesarios, cada nodo expandido ha de ser el de menor coste total posible. De esta manera, el algoritmo finalizará encontrando la solución al problema la primera vez que se expanda el vértice final, sin necesidad de realizar más comprobaciones. Esto se logra ordenando la lista abierta por el coste total.
   - **Resumen del tramo final en común con π.** Cuando se expande un vértice v_k extraído de la lista abierta, en el caso que se cumpla $v_k \in \pi$, en lugar de explorar los nodos contiguos y añadirlos a a lista abierta, aplicaremos la simplificación 3.9, insertando en la lista abierta el nodo resultante de añadir a v_k el tramo hasta v_n, e insertando v_k en la lista cerrada.

     Como no se ha añadido ningún vértice de π a la lista abierta en el recorrido inicial, sabemos que cualquier vértice v_k de lista abierta que pertenezca a π es producto de un camino alternativo que se ha reencontrado con π, y por tanto podemos aplicar la simplificación 3.9.

     Mediante la combinación del resumen del tramo final con la expansión por menor coste total se logra el mismo resultado que con la utilización de un umbral. Cuando expandimos v_n significa que hemos encontrado un camino desde v_0 a v_n distinto de π, ya que dicho nodo se ha expandido desde la lista abierta. Como la lista abierta está ordenada por coste, no existe ningún otro camino con un coste menor entre v_0 y v_n, pues se habría expandido el vértice correspondiente antes. Esto quiere decir que no se expandirán vértices con un coste mayor al de la solución del problema.
   - **Detección de duplicados**. La lista cerrada se utiliza para evitar la reexpansión de un vértice. Como el objetivo es encontrar el segundo camino de menor coste, en la lista abierta solamente se mantendrá un nodo por cada vértice.

     Dado el ejemplo de la Figura 3.5, si el vértice v_b se encuentra en la lista abierta, asociado al coste correspondiente al camino $v_1 \to v_a \to v_b$, y queremos añadir el camino $v_2 \to v_b$, únicamente se mantendrá uno de los dos caminos, el de menor coste.

3.4.2. Propuesta Bidireccional

Nuestra propuesta de solución bidireccional se fundamenta en el mismo esquema de búsqueda unidireccional. A continuación se describen los cambios más importantes respecto a la anterior propuesta.

La búsqueda bidireccional consiste en una búsqueda *forward* partiendo desde v_0 y una búsqueda *reverse* desde v_n. La búsqueda *reverse* emplea los mismos vértices y aristas pero en sentido inverso. Dado un vértice $v_i \in G$, buscaremos los vértices a los que está conectado por las aristas e_{v_j,v_i}.

Utilizamos la notación que aparece en el artículo de Andrew Goldberg [9] y empleamos los términos *Forward* y *Reverse* para designar los dos tipos de búsqueda en el algoritmo bidireccional. Aunque los pioneros en la búsqueda Bidireccional, Ira Pohl[26] y Dennis de Champeaux[3] utilizaban *Forward* y *Backward*, la búsqueda *Backward* puede confundirse con una búsqueda direccional de atrás hacia adelante, por lo que *Reverse*, que se traduce como 'marcha atrás', explica mejor el proceso.

En nuestra propuesta de solución unidireccional el algoritmo finaliza cuando se expande un camino que alcanza v_n. Dado que la lista abierta está ordenada por coste, cuando se expande v_n significa que ya no es posible encontrar caminos de menor coste.

Con el objetivo de reducir la cantidad de nodos expandidos del espacio de búsqueda modificaremos la condición de parada en la búsqueda bidireccional. Para ello necesitamos mantener en memoria el mejor camino encontrado π_μ.

Definición 6. (Coste del mejor camino encontrado - μ) — El camino $\pi_\mu \in \Pi_{v_0,v_n}$, $\pi_\mu \neq \pi$ se define como el camino de menor coste entre todos los caminos expandidos en un instante dado tal que $C_{\pi_\mu} = \mu$. Inicialmente $\mu = \infty$.

El valor de μ se actualiza cuando se encuentra un camino de menor coste al expandir un nodo. Si se está explorando una arista e_{v_i,v_k}:

$$\mu = min(\mu, C^f_{v_0,v_i} + e_{v_i,v_k} + C^r_{v_k,v_n})$$

donde $C^f_{v_0,v_i}$ es el coste de la búsqueda *forward* del camino entre v_0 y v_i y $C^r_{v_k,v_n}$ es el coste de la búsqueda *reverse* del camino entre v_k y v_n.

Definición 7. (Condición de parada Bidireccional) — Sea min^f y min^r el mínimo coste entre todos los caminos equivalentes a los vértices de la lista abierta, en las búsquedas *forward* y *reverse* respectivamente, y μ el coste del mejor camino encontrado; el algoritmo finalizará cuando $min^f + min^r \geq \mu$

Demostración. Supongamos que existe un camino $\pi_2 \in \Pi_{v_0,v_n}$ con un coste menor que el de π_μ, es decir, $C_{\pi_2} < \mu$. Si lo relacionamos con min^f u min^r tenemos:

$$C_{\pi_2} < \mu \leq min^f + min^r$$

Si $\pi_2 \in \Pi_{v_0,v_n}$ entonces existe una arista (la que conecta las búsquedas *forward* y *reverse*) que cumple:

$$\exists e_{v_i,v_k} \in \pi_2 \text{tal que} \pi_2 := \{\pi_{v_0,v_i}, e_{v_i,v_k}, \pi_{v_k,v-n}\}$$

Por lo que su coste valdrá:

$$C_{\pi_2} := C_{\pi_{v_0,v_i}} + c_{v_i,v_k} + C_{\pi v_k,v_n}$$

Por otro lado, si se encontró la arista e_{v_i,v_j} significa que los dos vértices han sido previamente expandidos. Como min^f y min^r son el coste de los primeros vértices en la lista abierta *Forward* y *Reverse* respectivamente, si los dos vértices pertenecientes a e_{v_i,v_k} ya han sido expandidos, se cumple que su coste será menor que el de min^f y min^r, $C_{\pi_{v_0,v_i}} \leq min^f$ y $C_{\pi v_k,v_n} \leq min^r$

Y cuando se encuentra la arista se comprueba

$$\mu = min(\mu, C^f_{v_0,v_i} + e_{v_i,v_k} + C^r_{v_k,v_n}) = min(\mu, C_{\pi_2})$$

En este razonamiento hemos supuesto que $C_{\pi_2} < \mu$. Si lo aplicamos a la expresión anterior tenemos $\mu = min(\mu, C_{\pi_2}) = C_{\pi_2}$, llegando a la contradicción $C_{\pi_2} < \mu = C_{\pi_2}$

□

En cuanto a la reducción del espacio de búsqueda, si analizamos la complejidad de la búsqueda bidireccional, para un valor de ramificación r y una solución de longitud l el coste será $O(r^{l/2})$ para cada una de las búsquedas. Suponiendo el mismo valor de ramificación, el coste total sería $O(2r^{l/2})$, mientras que una búsqueda al uso tendría un coste de $O(r^l)$[41].

3.5 Algoritmo de búsqueda

En esta sección presentamos los dos algoritmos de búsqueda, unidireccional y bidireccional, para la resolución del problema ASP-PCC. Los parámetros de entrada para ambos algoritmos son un grafo G y un camino óptimo entre dos de sus vértices π.

3.5.1. Búsqueda Unidireccional

Se muestra el algoritmo de búsqueda unidireccional (Algoritmo 3.1), que es el que se explica a continuación por ser el más sencillo de seguir.

Las estructuras de datos empleadas en el diseño del algoritmo unidireccional 3.1 son las siguientes:

- **Lista Abierta**: lista de vértices pendientes de expansión, ordenados por el coste total de cada camino *coste* + *sobrecoste*.

  Esta lista cumple una doble función, pues cuando aprovechamos el resultado de la ecuación 3.9 en el algoritmo 3.1 (líneas 22:32), volvemos a guardar en Abierta los caminos entre v_0 y v_n a modo de umbral. De esta manera, una vez encontrado un camino entre v_0 y v_n seguiremos expandiendo los nodos de menor coste, buscando un camino de menor coste, pero en el momento en el que se expanda v_n tenemos la garantía de que es la solución al problema ASP-PCC.

Algoritmo 3.1: Resolución del ASP Unidireccional

Requiere: Un grafo G, un camino óptimo π solución del PCC sobre G
Garantiza: El siguiente camino óptimo π', solución del ASP

1: Abierta, Cerrada $\leftarrow \{\}$ # Reciben $\langle vértice, coste, sobrecoste, progenitor \rangle$
2: Abierta.Orden $\leftarrow$ **function**$\big(\langle v_a, c_{v_a}, sc_{v_a}, v_{aParent}\rangle, \langle v_b, c_{v_b}, sc_{v_b}, v_{bParent}\rangle\big)$ {
3: **return** $c_{v_a} + sc_{v_a} - (c_{v_b} + sc_{v_b})$ }
4: Óptima, AbiertaInit $\leftarrow \big\{\langle v_0, 0, 0, \emptyset\rangle\big\}$; AbiertaInit.Orden $\leftarrow$ Abierta.Orden
5: **while** AbiertaInit $\neq \{\}$ **do** # Recorrido inicial del camino óptimo
6: $\langle v_i, c_{v_i}, sc_{v_i}, v_{parent}\rangle \leftarrow$ AbiertaInit.popFirst()
7: **if** $v_i \neq v_n$ **then**
8: **for** v_j tal que $\exists e_{v_i,v_j}$ **do**
9: **if** $v_j \in \pi$ **and** $(\pi.next(v_i) = v_j)$ **then**
10: Óptima, AbiertaInit $\xleftarrow{push}$ $\langle v_j, c_{v_i} + e_{v_i,v_j}, 0, v_i\rangle$
11: **else**
12: Abierta $\xleftarrow{push}$ $\langle v_j, c_{v_i}, e_{v_i,v_j}, v_i\rangle$
13: **end if**
14: **end for**
15: **end if**
16: **end while**
17: **while** Abierta $\neq \{\}$ **do**
18: $\langle v_i, c_{v_i}, sc_{v_i}, v_{parent}\rangle \leftarrow$ Abierta.popFirst() # Estado Actual
19: **if** $v_i = v_n$ **then** # Condición de Parada
20: **return** BackTrack$\big(\langle v_i, c_{v_i}, sc_{v_i}, v_{parent}\rangle\big)$
21: **else if** $v_i \in \pi$ **then** # Ecuación 3.9
22: $v_{aux} \leftarrow v_i$; $v_{prev} \leftarrow v_{parent}$
23: **while** $v_{aux} \neq v_n$ **do**
24: Cerrada $\xleftarrow{push}$ $\langle v_{aux}, c_{v_{aux}}, sc_{v_{aux}}, v_{prev}\rangle$
25: $v_{prev} = v_{aux}$
26: $v_{aux} = \pi.next(v_{aux})$ # Siguiente vértice en π
27: $c_{v_{aux}} \leftarrow c_{v_{prev}} + e_{v_{prev},v_{aux}}$
28: $sc_{v_{aux}} \leftarrow sc_{v_{prev}}$
29: **end while**
30: Abierta $\xleftarrow{push}$ $\langle v_{aux}, c_{v_{aux}}, sc_{v_{aux}}, v_{prev}\rangle$
31: **if** $c_{v_{aux}}, sc_{v_{aux}} = C_\pi$ **then**
32: **return** BackTrack$\big(\langle v_i, c_{v_i}, sc_{v_i}, v_{parent}\rangle\big)$
33: **end if**
34: **else if** $v_i \notin$ Cerrada **then** # Evitamos re-expansión
35: Cerrada $\xleftarrow{push}$ $\langle v_i, c_{v_i}, sc_{v_i}, v_{parent}\rangle$
36: **for** v_j tal que $\exists e_{v_i,v_j}$ **do**
37: Abierta $\xleftarrow{push}$ $\langle v_j, c_{v_i}, sc_{v_i} + e_{v_i,v_j}, v_i\rangle$
38: **end for**
39: **end if**
40: **end while**
41: **return** $\{\emptyset\}$

- **Lista Cerrada**: lista de vértices expandidos que se utiliza para la detección de nodos duplicados. Cada nodo expandido contiene el coste necesario para alcanzarlo, por lo que de cada vértice de la lista cerrada se puede obtener el camino más corto entre v_0 y dicho vértice.

- **Lista Óptima**: lista de los vértices del camino óptimo. Se usa en el recorrido inicial del camino y su función principal es llevar un registro del coste de los subcaminos hasta cada vértice del camino principal.

  Inicialmente no se planteó la utilización de la lista óptima para guardar los estados correspondientes al camino óptimo, dado que estos estados se guardaban en la lista Cerrada. Aunque efectivamente son nodos que no se vuelven a re-expandir, almacenar dichos nodos en la lista cerrada provocaba que dicha lista no pudiera ejercer la función de detección de duplicados de los vértices del camino óptimo.

- **Lista AbiertaInit**: lista auxiliar de idéntico funcionamiento a la lista Abierta. Se utiliza durante el primer recorrido de los vértices del camino óptimo.

En el algoritmo trabajamos con estados, que contienen la información del camino que alcanza un vértice. Su formato es el siguiente:

$$\langle vértice, coste, sobrecoste, progenitor \rangle \tag{3.11}$$

- *vértice*: vértice actual.

- *coste*: coste del subcamino del camino proporcionado que se ha recorrido en la búsqueda.

- *sobrecoste*: sobrecoste del camino a partir de la separación con π.

- *progenitor*: vértice progenitor, indispensable para recuperar los vértices que forman el camino.

La primera fase del algoritmo realiza un recorrido del camino óptimo, diferenciando entre los vértices que se generan y los que se expanden.

Los vértices que pertenecen al camino principal (visitados desde el vértice del camino principal adecuado) se expanden, actualizando el campo *coste* con el coste del vértice progenitor más la arista que los une y añadiendo el estado resultante a la lista Óptima.

Para el resto de vértices, que son los que no pertenecen al camino óptimo y los que sí pertenecen pero han sido explorados usando aristas que no forman parte del camino óptimo, el coste de la arista que los une se añade en el campo de *sobrecoste*, manteniendo el campo de *coste* de su vértice progenitor y añadiéndolos a la lista Abierta. A este proceso nos referimos cuando decimos que han sido generados.

El último estado del camino óptimo (v_n) no se expandirá para evitar la generación de vértices innecesarios.

Como resultado, en la lista Óptima se guardarán los estados correspondientes a los vértices del camino óptimo, y en la lista Abierta los vecinos de los vértices del camino óptimo, listos para ser expandidos.

En la exploración principal (líneas 18:39), a diferencia del primer recorrido, los costes de las aristas se añaden siempre en el campo *sobrecoste*. Añadir el coste de la arista al sobrecoste nos permite identificar en cada camino el vértice del camino principal del que han partido, además de ser consistente con el cambio en el tipo de búsqueda, elaborada a partir de la última definición de la solución del ASP-PCC (Definición 3.2).

En la exploración principal se puede incluir el primer recorrido del camino óptimo, pues al estar la lista Abierta ordenada por el coste total se visitan los mismos vértices y en el mismo orden.No obstante, hemos preferido explicar el diseño separando ambos recorridos para que se aprecien las diferencias entre el primer recorrido del camino óptimo y cuando, al explorar, se expande uno de sus vértices. Esta expansión significa que un camino alternativo se ha reencontrado con el camino óptimo y también significa que podemos aplicar los resultados de la ecuación 3.9.

La exploración principal se ejecuta mientras sigan existiendo estados por expandir y no se haya satisfecho la condición de parada. Se estructura en tres bloques **if-else** principales:

(líneas 20-21) Condición de Parada: El algoritmo finalizará cuando el siguiente estado a expandir se corresponda con el vértice final del camino óptimo. Al estar la lista Abierta ordenada por el coste total, esta condición significa que no existe otro camino que pueda llegar al vértice objetivo con un coste menor.

(líneas 22:32) Resumen del camino óptimo: esta condición comprueba si el estado expandido se corresponde con un vértice del camino óptimo (distinto de v_n). Gracias al recorrido inicial del camino óptimo sabemos que a este estado se ha llegado desde un subcamino no óptimo. Como hemos encontrado π^* entre dos vértices del camino óptimo, podemos reconstruir el resto del camino desde el vértice del estado hasta v_n (Ecuación 3.9).

Aunque puede parecer condición suficiente para parar el algoritmo, gracias al estudio sobre los enfoques realizado en la sección 3.3.2 sabemos que el primer camino que se reencuentra con el camino óptimo no garantiza ser el siguiente camino óptimo. Esta garantía la conseguimos añadiendo el estado con el vértice final que hemos alcanzado completando el camino actual a lista Abierta, esperando que dicho estado sea expandido y se cumpla la condición de parada.

En el algoritmo 3.1(líneas 12:22) se aprecia como se recorre de nuevo el camino óptimo, sin re-expandir el mismo. Aunque en este punto es posible incluir directamente el estado del vértice v_n, añadiendo el coste del subcamino desde el vértice actual hasta el final, hemos decidido mostrar paso a paso la creación de cada uno de los estados intermedios para ser consistentes con la notación y que el proceso de *BackTracking* sobre la solución del ASP sea trivial.

(líneas 33:38) Exploración: En caso de expandir un estado cuyo vértice no pertenece al camino óptimo, se realiza la expansión de dicho estado, que consiste en añadir el estado a la lista Cerrada y generar todos los vértices con los que está conectado por alguna arista.

Cuando hablamos de exploración nos referimos a la creación del estado de cada vértice, con el sobrecoste correspondiente a la suma del sobrecoste de su progenitor y la arista que los une, y a la inclusión del nuevo estado en la lista Abierta.

La condición de Resumen del camino óptimo podría haberse incluido dentro del bucle de Exploración, detectando si el vértice explorado pertenece al camino óptimo. La decisión de hacer esta detección al expandir, y no al explorar, sacrifica memoria (en caso de que se permitan duplicados en la lista Abierta) a cambio de reducir el número de comprobaciones.

Aunque el algoritmo funcionaría en caso de que las listas Abierta y Cerrada permitieran duplicados, su rendimiento se vería mejorado si cada una de las listas detectara duplicados, de tal manera que, para cada vértice, únicamente se mantenga el estado con un coste menor.

Siendo fieles a la definición de ASP (Definición 4), dado un problema y una solución al mismo, nos preguntamos si existe otra solución al problema, por lo que, en caso de no encontrar ninguna, el algoritmo devuelve un conjunto vacío (línea 39).

De igual manera, aunque desde el análisis del problema se ha dado por supuesto que el camino proporcionado se trata del camino óptimo. Cambiando la condición de parada:

```
20: if v_i ∈ π then
21:     return BackTrack(⟨v_i, c_{v_i}, sc_{v_i}, v_parent⟩)
22: end if
```

Por:

```
20: if v_i ∈ π then
21:     if c_{v_i} + sc_{v_i} >= C_{π_{v_0,v_i}} then
22:         return BackTrack(⟨v_i, c_{v_i}, sc_{v_i}, v_parent⟩)
23:     end if
24: end if
```

Conseguimos que nuestro algoritmo resuelva el ASP y obtenga la siguiente solución aunque la solución proporcionada no sea la óptima del problema.

Sería también trivial modificar el algoritmo para guardar o admitir aquellos caminos con un coste menor que el óptimo con la siguiente modificación de la condición de parada:

```
20: if v_i ∈ π then
21:     if c_{v_i} + sc_{v_i} >= C_{π_{v_0,v_i}} then
22:         return BackTrack(⟨v_i, c_{v_i}, sc_{v_i}, v_parent⟩)
23:     else
24:         ...        # Guardamos los caminos con coste menor que el proporcionado
25:     end if
```

26: **end if**

3.5.2. Búsqueda Bidireccional

Se presenta el algoritmo que incorpora la búsqueda bidireccional (Algoritmo 3.2), realizado a partir del algoritmo unidireccional (Algoritmo 3.1). Por razones de espacio y legibilidad hemos separado en otra figura el primer recorrido del camino óptimo (Algoritmo 3.3).

A continuación destacaremos los principales aspectos del algoritmo bidireccional, procurando evitar aspectos del algoritmo unidireccional que no hayan sido alterados.

Los estados con los que trabaja el algoritmo respecto a los estados de la búsqueda unidireccional (3.11) son el resultado de duplicar los campos de *coste, sobrecoste y progenitor* para la búsqueda *forward* y *reverse*, y siguen el siguiente formato:

$$\langle vértice, coste^f, sobrecoste^f, coste^r, sobrecoste^r, progenitor^f, progenitor^r \rangle \quad (3.12)$$

- *vértice*: vértice actual
- $coste^f$: coste del subcamino del camino proporcionado que se ha recorrido en la búsqueda *forward*
- $sobrecoste^f$: sobrecoste del camino a partir de la separación del camino proporcionado en la búsqueda *forward*
- $coste^r$: coste del subcamino del camino proporcionado que se ha recorrido en la búsqueda *reverse*
- $sobrecoste^r$: sobrecoste del camino a partir de la separación del camino proporcionado en la búsqueda *reverse*
- $progenitor^f$: vértice progenitor en la búsqueda *forward*
- $progenitor^f$: vértice progenitor en la búsqueda *backward*

Las estructuras de datos empleadas en el diseño del algoritmo bidireccional 3.2 son:

- **Lista Abierta**: lista de vértices pendientes de expansión, ordenados por el coste total de cada camino $coste^f + sobrecoste^f + coste^r + sobrecoste^r$.

  La lista Abierta sigue formando parte del mecanismo de parada del algoritmo, pero no por almacenar el mejor camino encontrado, que por comodidad guardaremos en una variable, si no por mantener el valor del coste total del primer vértice de la búsqueda *forward* y *reverse*.

  Estos valores cumbre nos permitirán parar el algoritmo cuando la suma sea mayor que la del mejor camino encontrado, garantizando que la solución es óptima como se demostró en la subsección 3.4.2.

Algoritmo 3.2: Resolución del ASP Bidireccional

Requiere: Un grafo G, un camino óptimo π solución del PCC sobre G
Garantiza: El siguiente camino óptimo π', solución del ASP

Cerrada $\leftarrow \{\}$
Abierta, Óptima $\leftarrow$**Init( )** # Recorrido del camino óptimo(Algoritmo 3.3)
$\mu \leftarrow \infty, v_\mu \leftarrow \emptyset$
$min^f \leftarrow 0, min^r \leftarrow 0$
while Abierta $\neq \{\}$ **do**
  $\langle v_i, c^f_{v_i}, sc^f_{v_i}, c^r_{v_i}, sc^r_{v_i}, v^f_{parent}, v^r_{parent}\rangle \leftarrow$ Abierta.popFirst() # Estado Actual
  if $v_i \notin$ Cerrada **then** # Evitamos re-expansion
    Cerrada $\xleftarrow{push} \langle v_i, c^f_{v_i}, sc^f_{v_i}, c^r_{v_i}, sc^r_{v_i}, v^f_{parent}, v^r_{parent}\rangle$
    if $v^r parent = \emptyset$ **then** # Caso *Forward*
      for v_j tal que $\exists e_{v_i,v_j}$ **do**
        $c^f_{v_j} \leftarrow c^f_{v_i}$
        $sc^f_{v_j} \leftarrow sc^f_{v_i} + e_{v_i,v_j}$
        Abierta $\xleftarrow{push} \langle v_j, c^f_{v_j}, sc^f_{v_j}, c^r_{v_j}, sc^r_{v_j}, v_i, v^r_{parent}\rangle$
        if $c^r_{v_j} + sc^r_{v_j} > 0$ **and** $c^f_{v_j} + sc^f_{v_j} + c^r_{v_j} + sc^r_{v_j} < \mu$ **then**
          $\mu \leftarrow c^f_{v_j} + sc^f_{v_j} + c^r_{v_j} + sc^r_{v_j}, v_\mu \leftarrow v_j$
        end if
      end for
      $min^f \leftarrow c^f_{v_i} + sc^f_{v_i}$
    else # Caso *Reverse*
      for v_j tal que $\exists e_{v_j,v_i}$ **do**
        $c^r_{v_j} \leftarrow c^r_{v_i}$
        $sc^r_{v_j} \leftarrow sc^r_{v_i} + e_{v_j,v_i}$
        Abierta $\xleftarrow{push} \langle v_j, c^f_{v_j}, sc^f_{v_j}, c^r_{v_j}, sc^f_{v_j}, v^f_{parent}, v_i\rangle$
        if $c^f_{v_j} + sc^f_{v_j} > 0$ **and** $c^f_{v_j} + sc^f_{v_j} + c^r_{v_j} + sc^r_{v_j} < \mu$ **then**
          $\mu \leftarrow c^f_{v_j} + sc^f_{v_j} + c^r_{v_i} + sc^r_{v_i}, v_\mu \leftarrow v_j$
        end if
      end for
      $min^r \leftarrow c^r_{v_i} + sc^r_{v_i}$
    end if
  end if
  if $min^f + min^r \geq \mu$ **or** $\mu = C_\pi$ **then** # Condición de Parada
    return BackTrack(v_μ)
  end if
end while
return $\{\emptyset\}$

Algoritmo 3.3: Resolución del ASP Bidireccional - Recorrido del camino proporcionado

Requiere: Un grafo G, un camino óptimo π solución del SPP sobre G
Garantiza: La inicialización de las listas *Óptima* y *Abierta*

function Init() {
Abierta.Orden ← **function**$\big(\langle v_a, c^f_{v_a}, sc^f_{v_a}, c^r_{v_a}, sc^r_{v_a} v^f_{aParent}, v^r_{aParent}\rangle$,
$\langle v_b, c^f_{v_b}, sc^f_{v_b}, c^r_{v_b}, sc^r_{v_b}, v^f_{bParent}, v^r_{bParent}\rangle\big)$ {
  return $(c^f_{v_a} + sc^f_{v_a} + c^r_{v_a} + sc^r_{v_a}) - (c^f_{v_b} + sc^f_{v_b} + c^r_{v_b} + sc^r_{v_b})$
}
AbiertaInit.Orden ← Abierta.Orden
Óptima, AbiertaInit ← $\{\langle v_0, 0, 0, C_\pi, 0, \emptyset, \pi.next(v_0)\rangle\}$
for v_j tal que $\exists e_{v_0,v_j}$ **do**
  if $v_j = \pi.next(v_i)$ **then**
    Óptima, AbiertaInit $\xleftarrow{push}$ $\langle v_j, e_{v_0,v_j}, 0, C_\pi - e_{v_0,v_j}, 0, v_0, \pi.next(v_j)\rangle$
  else # En v_0 solo hay búsqueda *Forward*
    Abierta $\xleftarrow{push}$ $\langle v_j, 0, e_{v_0,v_j}, 0, 0, v_0, \emptyset\rangle$
  end if
end for
while AbiertaInit $\neq \{\}$ **do**
  $\langle v_i, c^f_{v_i}, sc^f_{v_i}, c^r_{v_i}, sc^r_{v_i}, v^f_{parent}, v^r_{parent}\rangle$ ←AbiertaInit.popFirst()
  if $v_i \neq v_n$ **then** # En v_n solo hay búsqueda *Reverse*
    for v_j tal que $\exists e_{v_i,v_j}$ **do** # *Forward*
      if $v_j \in \pi$ **then**
        $c^f_{v_j} \leftarrow c_{v_i} + e_{v_i,v_j}$
        $c^r_{v_j} \leftarrow c^r_{v_i} - e_{v_i,v_j}$
        Óptima, AbiertaInit $\xleftarrow{push}$ $\langle v_j, c^f_{v_j}, 0, c^r_{v_j}, 0, v_i, \pi.next(v_j)\rangle$
      else
        Abierta $\xleftarrow{push}$ $\langle v_j, c^f_{v_i}, e_{v_i,v_j}, 0, 0, v_i, \emptyset\rangle$
      end if
    end for
  end if
  for v_j tal que $\exists e_{v_j,v_i}$ **do** # *Reverse*
    if $v_j \notin \pi$ **then**
      Abierta $\xleftarrow{push}$ $\langle v_j, 0, 0, c^r_{v_i}, e_{v_j,v_i}, \emptyset, v_i\rangle$
    else
      Abierta $\xleftarrow{push}$ $\langle v_i, c^f_{v_j}, e_{v_j,v_i}, c^r_{v_i}, 0, v_j, v^r_{parent}\rangle$
    end if
  end for
end while
return Abierta, Óptima
}

- **Lista Cerrada**: lista de vértices expandidos que se utiliza para la detección de nodos duplicados. Cada nodo expandido contiene el coste necesario para alcanzarlo, por lo que de cada vértice de la lista cerrada se puede obtener el camino más corto entre dicho vértice y v_0 o v_n en el caso de la búsqueda *forward* o *reverse* respectivamente. Dependiendo de la implementación será más sencillo guardar el vértice visitado en la búsqueda *forward* y en la búsqueda *reverse* en la misma lista o en dos listas diferentes.

- **Lista Óptima**: lista de los vértices del camino óptimo. Se usa en el recorrido inicial del camino y su función principal es llevar un registro del coste de los subcaminos hasta cada vértice del camino principal.

- **Lista AbiertaInit**: lista auxiliar de idéntico funcionamiento a la lista Abierta. Se utiliza durante el primer recorrido de los vértices del camino óptimo (Algoritmo 3.3).

La primera fase del algoritmo, separada en el método **Init()** (Algoritmo 3.3), realiza un recorrido del camino óptimo, con algunas diferencias respecto al mismo recorrido en el algoritmo unidireccional (Algoritmo 3.1).

En el algoritmo unidireccional se expandían todos los vértices del camino óptimo visitados en el orden de π salvo en el caso del último estado. Este es el mismo caso que el de la búsqueda *forward*.

En el caso de la búsqueda *reverse*, siguiendo el mismo razonamiento, expandiremos todos los vértices salvo el vértice inicial, evitando incluir en Abierta estados innecesarios.

En los vértices del camino óptimo se actualiza el campo de coste tanto *forward* como *backward*. El resto de vértices se añadirán a la lista Abierta, con *coste* y *sobrecoste* dependiendo de si la búsqueda es *forward* o *reverse*, pero siempre guardando en el campo *coste* el valor del coste del vértice de π del que son originarios y en el campo *sobrecoste* el coste de la arista que los une..

El resultado será el mismo que en el algoritmo unidireccional, salvo que en Abierta habrán vértices visitados por la búsqueda *forward* y vértices visitados por la búsqueda *reverse*.

Los principales cambios en el cuerpo del algoritmo son los siguientes:

(líneas 31-33) Condición de Parada: El algoritmo finaliza cuando el coste de los siguientes elementos a expandir (min^f en la búsqueda *forward* y min^r en la búsqueda *reverse*)) en la lista abierta supere al de la mejor solución encontrada. La mejor solución encontrada será la de aquel vértice que haya sido explorado en ambos sentidos de la búsqueda (*forward* y *reverse*) con un coste menor. Al estar la lista Abierta ordenada por el coste total, esta condición significa que no existe otro camino que pueda llegar al vértice objetivo con un coste menor (Definición 7). También finalizará en el caso trivial de que encuentre una solución con el mismo coste que π.

Resumen del camino óptimo: ya no resulta necesario resumir la parte en común con el camino óptimo como se hacía en el algoritmo unidireccional. Esta función la desarrolla la búsqueda *reverse*, por lo que si la búsqueda *forward*

encuentra un vértice explorado por la búsqueda *reverse*, actualizará si procede el mejor camino encontrado π_μ y continuará la exploración.

(líneas 7:30) Exploración: Al expandir un vértice se actualizará el valor de min^f o min^r según el tipo de búsqueda, se añadirá dicho vértice a Cerrada y se generaran todos los vértices a los que está conectado comprobando que no ha sido anteriormente visitado por la búsqueda contraria.

Cuando hablamos de exploración nos referimos a la creación del estado de cada vértice, con el sobrecoste correspondiente a la suma del sobrecoste de su progenitor y la arista que los une, y a la inclusión del nuevo estado en la lista Abierta (la exploración actualizará $sobrecoste^f$ o $sobrecoste^r$ dependiendo del sobrecoste del progenitor).

Al igual que ocurre con el algoritmo unidireccional (Algoritmo 3.1), en caso de no encontrar solución el algoritmo devuelve el conjunto vacío.

El mismo tipo de modificación se requiere en el algoritmo bidireccional si queremos que obtenga la siguiente solución aunque la solución proporcionada no sea la óptima del problema. En este caso la comprobación se haría en la exploración, actualizando el mejor camino encontrado π_μ mientras su coste no mejora el de π, como se ve en el siguiente ejemplo:

14: **if** $c^r_{v_j} + sc^r_{v_j} > 0$ **and** $C_\pi \leq c^f_{v_j} + sc^f_{v_j} + c^r_{v_j} + sc^r_{v_j} < \mu$ **then**
15: $\quad \mu \leftarrow c^f_{v_j} + sc^f_{v_j} + c^r_{v_j} + sc^r_{v_j}, v_\mu \leftarrow v_j$
16: **end if**

Aunque no es motivo de estudio de este trabajo las mejoras del rendimiento en la implementación, los algoritmos bidireccionales son susceptibles de ser paralelizados, pues las dos búsquedas se pueden realizar en paralelo únicamente compartiendo los valores de min^f, min^r y π_μ.

La búsqueda *reverse* se puede aplicar en grafos direccionales pues se buscan las aristas inversas. De igual manera funcionaría con una definición implícita, requiriendo que los operadores tengan su correspondiente operador inverso. De hecho no es necesario que el operador sea invertible, basta con que, a partir de un vértice v_i se pueda obtener un conjunto de vértices tal que existan operadores válidos que los conecten a v_i.

CAPÍTULO 4
Resultados

En este capítulo presentamos los resultados que se han obtenido con los algoritmos propuestos para la resolución del ASP-PCC. Con el trabajo expuesto en este TFG hemos diseñado el primer algoritmo de carácter general que soluciona el problema del *Another Solution Problem* aplicado al *Problema del Camino más Corto*.

Dado que no existe otro algoritmo que resuelva el problema ASP-PCC con el que podamos comparar nuestras propuestas, hemos optamos por realizar una comparativa entre el esquema de búsqueda unidireccional y el algoritmo de Dijkstra, por ser Dijkstra un algoritmo de carácter general, y una comparativa entre la versión unidireccional y bidireccional del algoritmo. Adicionalmente, en la sección 4.1 explicamos la herramienta utilizada para la implementación de los dos algoritmos y mostramos varias capturas de la ejecución de la búsqueda unidireccional y bidireccional a través de la interfaz gráfica de la herramienta.

4.1 Representación Gráfica de la Búsqueda

Como se ha comentado anteriormente, según nuestro conocimiento no existen otras propuestas que resuelvan el ASP-PCC. Consecuentemente, y dado que no existen aproximaciones con las que comparar nuestros algoritmos, hemos optado por realizar la implementación en JavaScript aprovechando un proyecto de visualización de algoritmos llamado PathFinding.js [43].

PathFinding.js es un proyecto con licencia de software libre *MIT* [18]. Todo el proyecto es accesible desde GitHub, una popular plataforma desde la que es posible encontrar numerosos proyectos de código abierto [35]. A través de GitHub hemos buscado proyectos de visualización que se pudieran aprovechar para la realización de las pruebas experimentales.

El proyecto PathFinding.js [43] incluye la implementación de varios algoritmos de búsqueda de caminos, entre los cuales se encuentra el algoritmo Dijkstra [4], así como el código necesario para la visualización del proceso de búsqueda. Esta es una de las principales razones por las que se ha elegido este proyecto pues proporciona las herramientas necesarias para una rápida y fácil visualización gráfica de la ejecución de los algoritmos sin necesidad de realizar modificaciones adicionales, permitiendo así al desarrollador centrarse en la implementación del algoritmo.

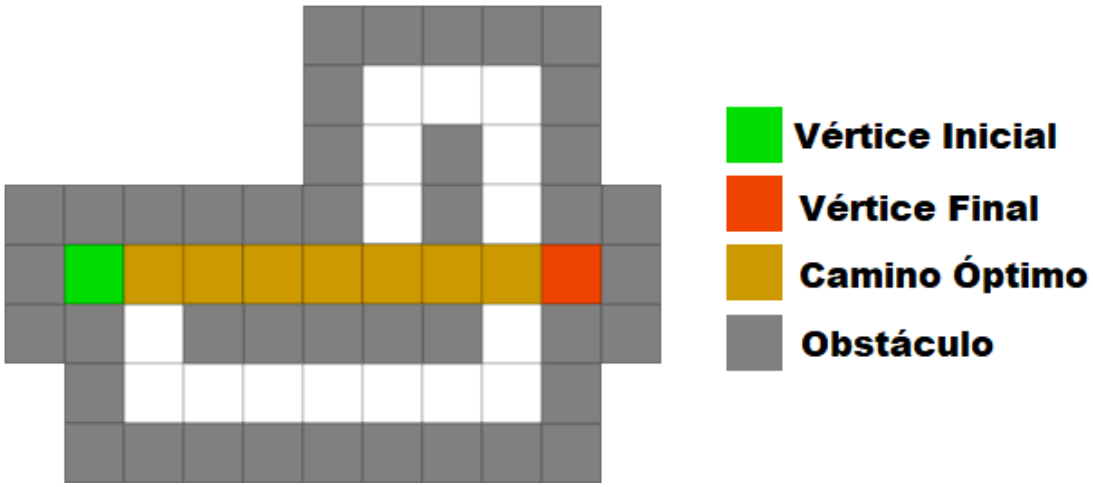


Figura 4.1: Elementos de la representación gráfica utilizando el proyecto PathFinding.js

Aunque se estudiaron otras opciones, no se encontró ningún proyecto de software libre [36] que ofreciera un código debidamente estructurado [2] y que facilitara la inclusión de nuevos algoritmos sin necesidad de tener que realizar modificaciones en la interfaz gráfica para la visualización de los algoritmos. La ventaja de PathFinding.js [43] es que ofrece una completa separación entre la parte lógica de resolución del problema y la representación gráfica de los algoritmos.

A continuación se detallan los elementos que utilizaremos para la representación visual de la búsqueda, los cuales se pueden observar gráficamente en las Figuras 4.1 y 4.2:

- **Bloque Verde Intenso:** v_0 (vértice inicial).
- **Bloque Rojo**: v_n (vértice final).
- **Bloque Gris Obstáculo:** nodo no visitable.
- **Bloque Marrón:** π, solución óptima al Problema del Camino más Corto, tal como se le ha proporcionado al algoritmo.
- **Bloque Verde Pálido:** lista Abierta (vértices generados).
- **Bloque Azul:** lista Cerrada (vértices expandidos).
- **Línea Amarilla:** π', solución encontrada (ver Figura 4.2).
- Texto en la esquina inferior izquierda - Longitud del camino encontrado, tiempo transcurrido y operaciones realizadas (generación y expansión)(esto se puede observar, por ejemplo, en las Figuras 4.3 o 4.5).

En la Figura 4.1 podemos observar el vértice inicial (en color verde), el vértice final (en color rojo) y los vértices del camino óptimo entre ambos (en color marrón). En esta figura existen dos sub-caminos alternativos, el camino que parte desde v_1 y llega hasta v_{n-1} (casillas de color blanco que aparecen debajo del camino óptimo) y el camino que parte desde v_{n-3} y llega hasta v_{n-1} (casillas de color blanco que se muestran por encima del camino óptimo π).

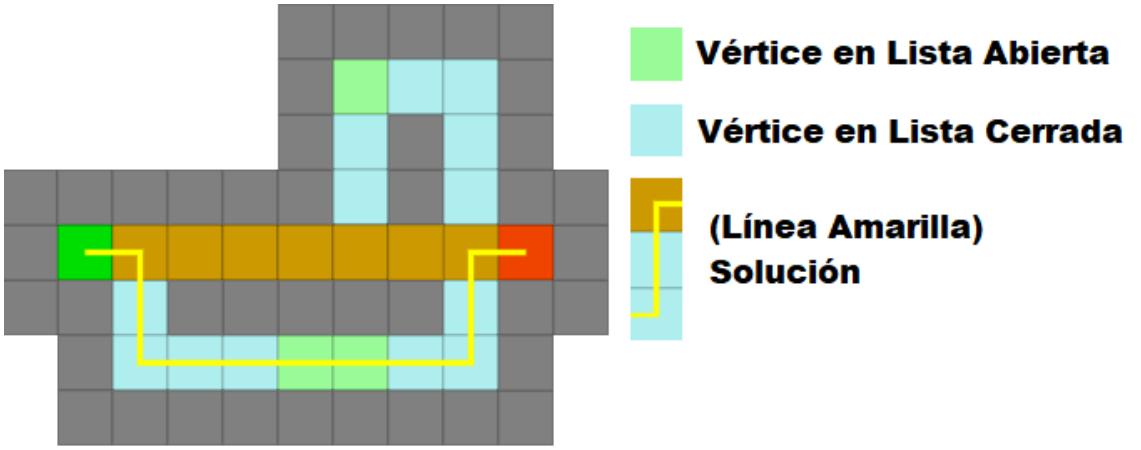


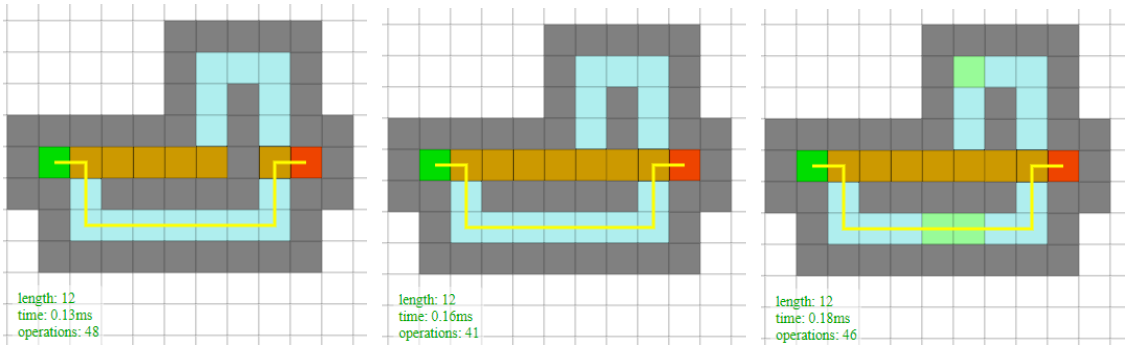
Figura 4.2: Elementos de la representación gráfica de la solución del algoritmo bidireccional

La Figura 4.2 muestra el resultado de aplicar el algoritmo bidireccional sobre el grafo de la Figura 4.1. En dicha figura podemos observar los elementos gráficos del proceso de búsqueda y la solución. Por un lado tenemos, en color azul, los vértices de la lista Cerrada, vértices que han sido expandidos durante la búsqueda de la solución al ASP-PCC. Y en color verde claro se muestran los vértices de la lista Abierta. La línea amarilla se corresponde con la solución encontrada por el algoritmo. En este caso vemos que la solución alternativa encontrada parte del nodo inicial, recorre un vértice de π y toma el camino alternativo inferior dado que su coste total es menor que utilizando el camino alternativo superior.

Un detalle de la implementación es que no se visualiza el último vértice expandido porque la solución se encuentra antes de que finalice la expansión. Por este motivo en la búsqueda bidireccional se pueden observar dos vértices en la lista Abierta que corresponden a la frontera de las búsquedas *forward* y *reverse* de la solución correspondiente.

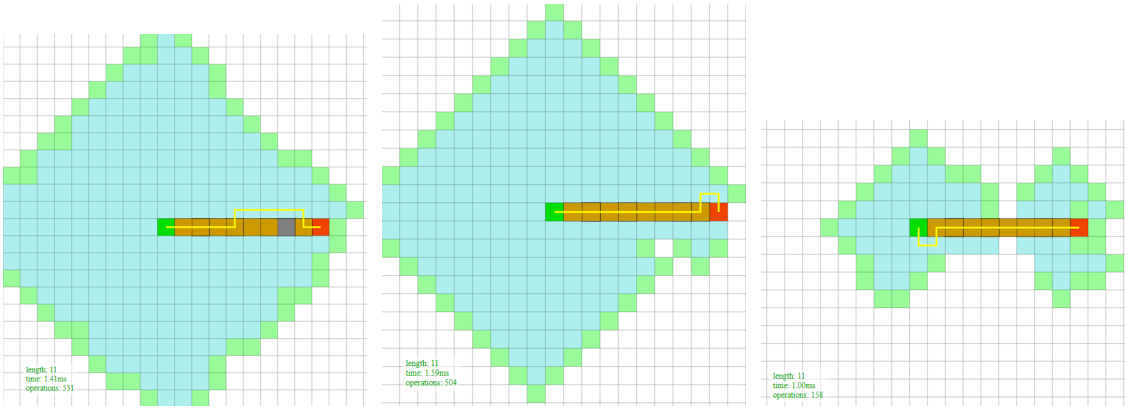
En el caso del algoritmo Dijkstra, dado que este algoritmo no resuelve exactamente el mismo tipo de problema, se ha obstaculizado un vértice del camino óptimo que no fuese crítico, es decir que no impidiese encontrar la siguiente solución. Por dicho motivo en las figuras correspondientes al algoritmo de Dijkstra se puede ver como el camino óptimo, π, está bloqueado en algún vértice. En la Figura 4.3a vemos que el vértice v_{n-2} está bloqueado mientras que en las figuras vecinas 4.3b y 4.3c este vértice forma parte de la solución óptima.

La Figura 4.3 muestra un grafo con dos caminos alternativos. Es un caso muy sencillo de ASP-PCC que sirve para comprobar que la solución obtenida es correcta. En el caso de las figuras 4.3 se puede observar el camino π proporcionado. Debido a los obstáculos solo hay dos caminos alternativos posibles. Uno que parte de π en v_1 y que se sitúa por debajo de π, y otro que parte desde v_{n-3} y se sitúa por encima de π. En este caso, de los dos sub-caminos alternativos formados por nodos no compartidos con π, el camino superior tiene menos coste que el inferior pero utilizando el sub-camino inferior se consigue una solución alternativa de menor coste total, la cual es la que devuelven los algoritmos que resuelven el ASP-PCC como segunda solución.



(a) Dijkstra **(b)** ASP-PCC unidireccional **(c)** ASP-PCC bidireccional

Figura 4.3: Grafo con dos caminos alternativos



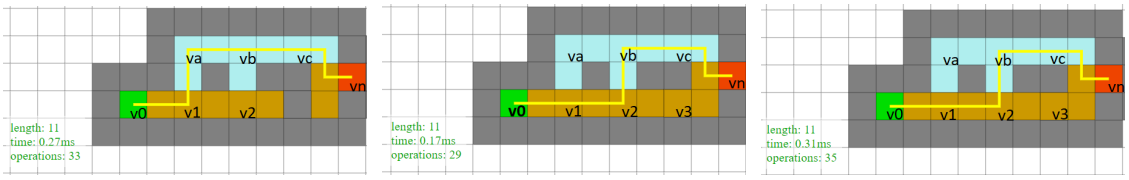
(a) Dijkstra **(b)** ASP-PCC unidireccional **(c)** ASP-PCC bidireccional

Figura 4.4: Comparativa del espacio de búsqueda explorado

La Figura 4.4 muestra la máxima ramificación que se puede producir en un espacio de búsqueda (sin obstáculos) generado con el proyecto PathFinding.js. Como PathFinding.js trabaja con problemas de búsqueda representados en una cuadrícula, el caso de la Figura 4.4 en el que no hay obstáculos representa el escenario donde se genera un mayor espacio de búsqueda. Se puede observar que apenas existen diferencias entre el resultado del algoritmo Dijkstra (Figura 4.4a) y la búsqueda unidireccional (Figura 4.4b), cumpliendo de este modo nuestras expectativas respecto al algoritmo unidireccional.

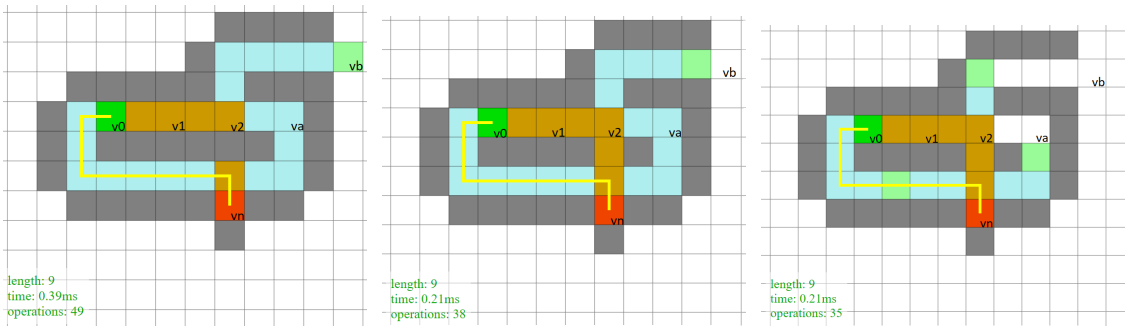
Si comparamos la búsqueda bidireccional (Figura 4.4c) con la unidireccional (Figura 4.4b) el espacio de búsqueda se reduce considerablemente, lo que es consistente con la estimación teórica al respecto de la búsqueda bidireccional en la sección 3.4.2.

La representación gráfica de las Figuras 4.5 y 4.6 reproduce lo más fielmente posible el proceso de búsqueda en los grafos de las Figuras 3.5 y 3.6, respectivamente. Para ello hemos dibujado los diferentes caminos posibles en ambas figuras, asignando dos vértices del mapa por cada camino de coste uno para respetar la vecindad. Por ejemplo, en la Figura 3.5 existe un vértice v_a conectado a v_1 por una arista de coste uno y un vértice v_b conectado a v_a y v_2 por aristas de coste uno. En la Figura 4.5 localizamos v_a en la esquina superior izquierda a dos vértices de distancia del camino óptimo (dos vértices = arista de coste uno), y lo mismo pasaría con v_b conectado por una arista a v_a y al camino óptimo.



(a) Dijkstra **(b)** ASP-PCC unidireccional **(c)** ASP-PCC bidireccional

Figura 4.5: Búsqueda en el grafo equivalente de la Figura 3.5



(a) Dijkstra **(b)** ASP-PCC unidireccional **(c)** ASP-PCC bidireccional

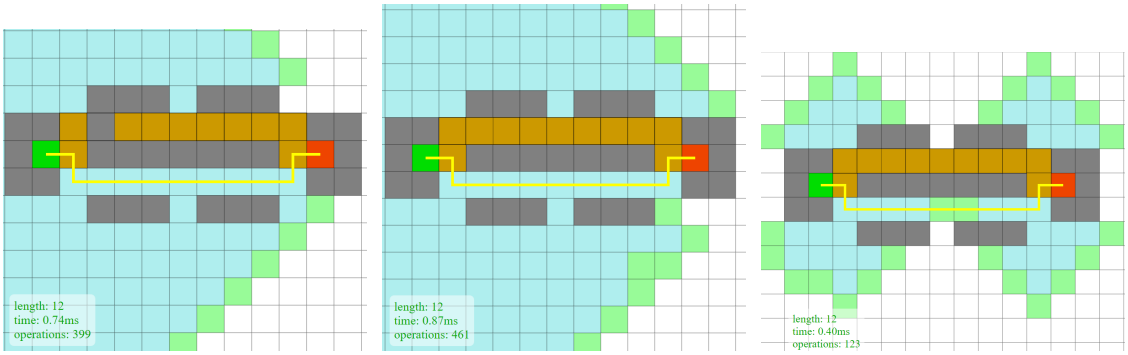
Figura 4.6: Búsqueda en el grafo equivalente de la Figura 3.6

En la Figura 4.5 apenas existe diferencia visual en el comportamiento de los algoritmos ya que el grafo muy sencillo. Recordemos que la búsqueda bidireccional es más efectiva cuanto mayor es el espacio de búsqueda.

En las Figuras 4.6 se puede observar una considerable mejora de la búsqueda bidireccional (Figura 4.6c) respecto al resto de algoritmos. Pese a ser un grafo muy sencillo, hay bastante diferencia respecto a la búsqueda de la Figura 4.5 ya que la búsqueda bidireccional de la Figura 4.6 muestra vértices que no se expanden mientras que en la Figura 4.5 no hay diferencia entre los vértices expandidos en ninguno de los algoritmos. La razón es que el algoritmo bidireccional solventa el inconveniente de la búsqueda unidireccional de desconocer el vértice $v_k \in \pi$ en el que cada subcamino que parte de π se reencontrará de nuevo con el camino óptimo. El algoritmo bidireccional **tendrá un mejor rendimiento** en aquellos grafos en los que la siguiente solución comparta pocos vértices con π o el coste del subcamino de la siguiente solución sea alto comparado con el resto de subcaminos, como es el caso de la Figura 4.6.

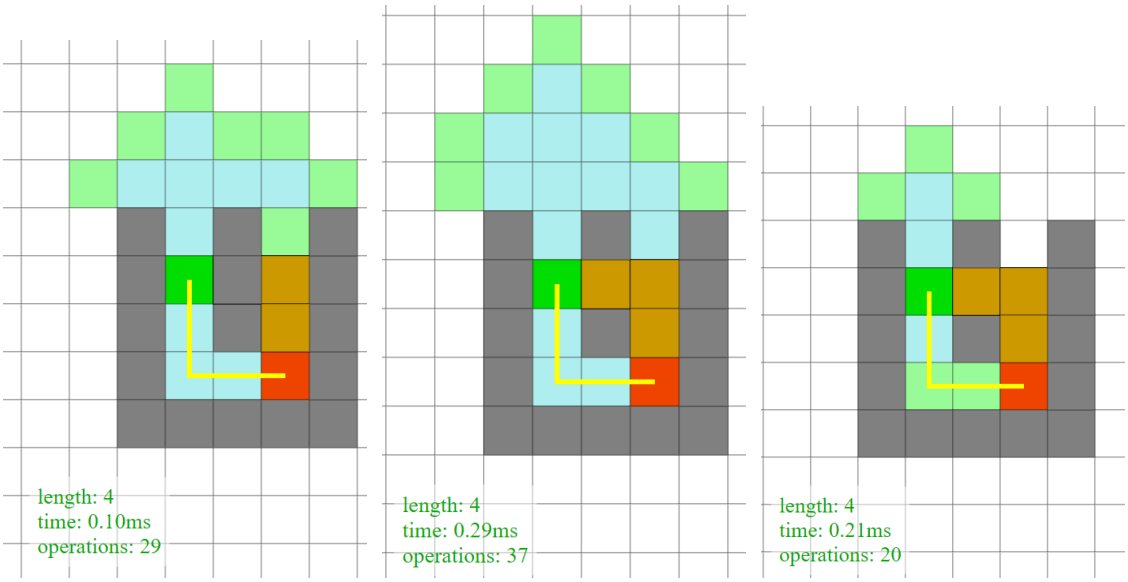
Se ha querido realizar la representación gráfica del caso del nodo v_b conectado al nodo v_2 de la Figura 3.6 con el camino inconcluso de la parte superior. En la Figura 4.6, el nodo v_2 ocupa la posición de la esquina del camino óptimo. Se puede comprobar que en la búsqueda unidireccional y la bidireccional no se expande ningún nodo con un coste mayor al de la segunda solución. Este es uno de los riesgos de un mal diseño, pues aunque un algoritmo encuentre la solución correcta podría explorar más caminos de los necesarios.

En la Figura 4.7 se muestra un caso en el que la solución del ASP-PCC tiene el mismo coste que el camino óptimo π. Las dos versiones del algoritmo ASP-PCC lo solucionan correctamente, la versión bidireccional con una exploración visiblemente menor (Figura 4.7c). Además en esta figura se puede observar el punto exacto en el que se encuentran las búsquedas *forward* y *reverse* en el algo-



(a) Dijkstra **(b)** ASP-PCC Unidireccional **(c)** ASP-PCC Bidireccional

Figura 4.7: Grafo con solución al ASP del mismo coste que π



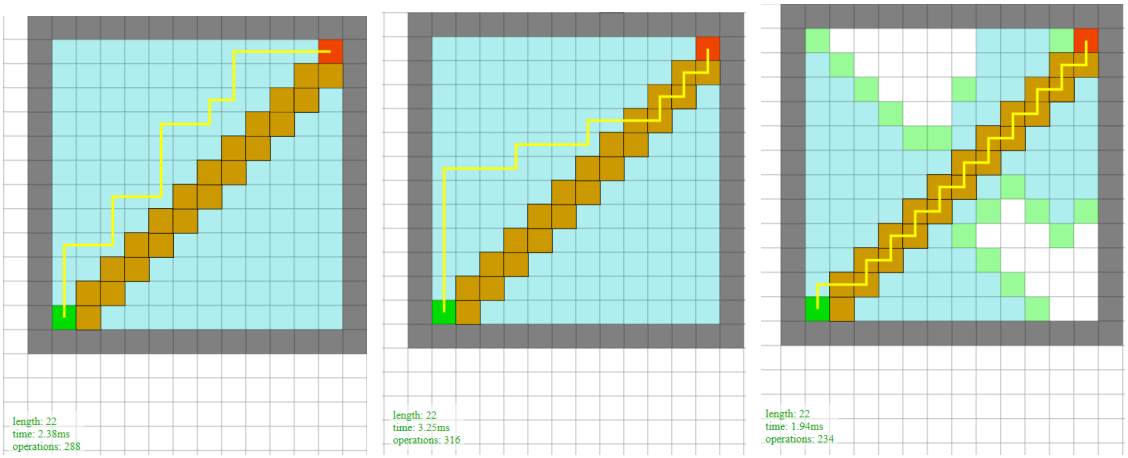
(a) Dijkstra **(b)** ASP-PCC unidireccional **(c)** ASP-PCC bidireccional

Figura 4.8: Grafo donde la solución al problema no comparte tramos con π

ritmo bidireccional de la figura 4.7c, justo a mitad de la solución encontrada. Esta es la representación de la nueva condición de parada del algoritmo bidireccional, explicada en la sección de propuesta bidireccional 3.4.2. El algoritmo finaliza cuando las dos búsquedas coinciden y la suma de los mejores caminos *reverse* y *forward* supera al mejor camino encontrado.

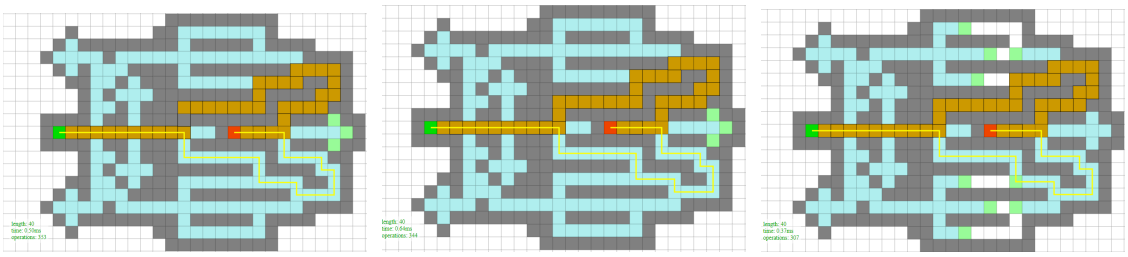
Otro caso *especial* a tener en cuenta es la de un grafo donde la solución del ASP-PCC no comparte vértices con la solución proporcionada, como se muestra en la Figura 4.8. Ambas versiones resuelven correctamente el problema.

En la Figura 4.9 hemos incluido una búsqueda en un mapa/cuadrícula. Este tipo de grafo es interesante porque todos los caminos posibles (con 2 operadores: arriba y derecha) tienen el mismo coste. Aunque nuestro algoritmo únicamente encuentra una solución, una posible ampliación del problema sería la de obtener todos los caminos con un coste equivalente. En el caso de nuestro diseño, sería tan sencillo como mantener todas las soluciones encontradas en una lista aparte sin detección de duplicados.



(a) Dijkstra **(b)** ASP-PCC unidireccional **(c)** ASP-PCC bidireccional

Figura 4.9: Grafo de tipo cuadricula



(a) Dijkstra **(b)** ASP-PCC unidireccional **(c)** ASP-PCC bidireccional

Figura 4.10: Grafo tipo mazmorra

Por último hemos incluido la Figura 4.10 (ampliado en la Figura 4.11) como ejemplo de problema de tipo *pathfinding*, para comprobar como responde el algoritmo visualmente.

4.2 Rendimiento Comparativo

En esta sección se presentan los resultados obtenidos en el rendimiento comparativo entre los tres algoritmos: Dijkstra, búsqueda unidireccional y búsqueda bidireccional.

Para poder aplicar Dijkstra en igualdad de condiciones, se ha bloqueado en cada caso un nodo de π para forzar al algoritmo de Dijkstra a encontrar otra solución. Dado que el problema que se resuelve con Dijkstra es ligeramente distinto que el que se resuelve con los dos algoritmos de búsqueda propuestos, nuestro objetivo no es demostrar que nuestra propuesta es más rápida sino comprobar que los resultados de la versión unidireccional son similares (del mismo orden de magnitud) que los resultados del algoritmo Dijkstra. También comprobaremos como cambia el rendimiento entre la versión unidireccional y bidireccional del algoritmo.

Como se comentó en la sección 2.1 del estado del arte, un problema similar al ASP-PCC es el de los K-Caminos [45], tomando $K = 2$, por lo que elegir el

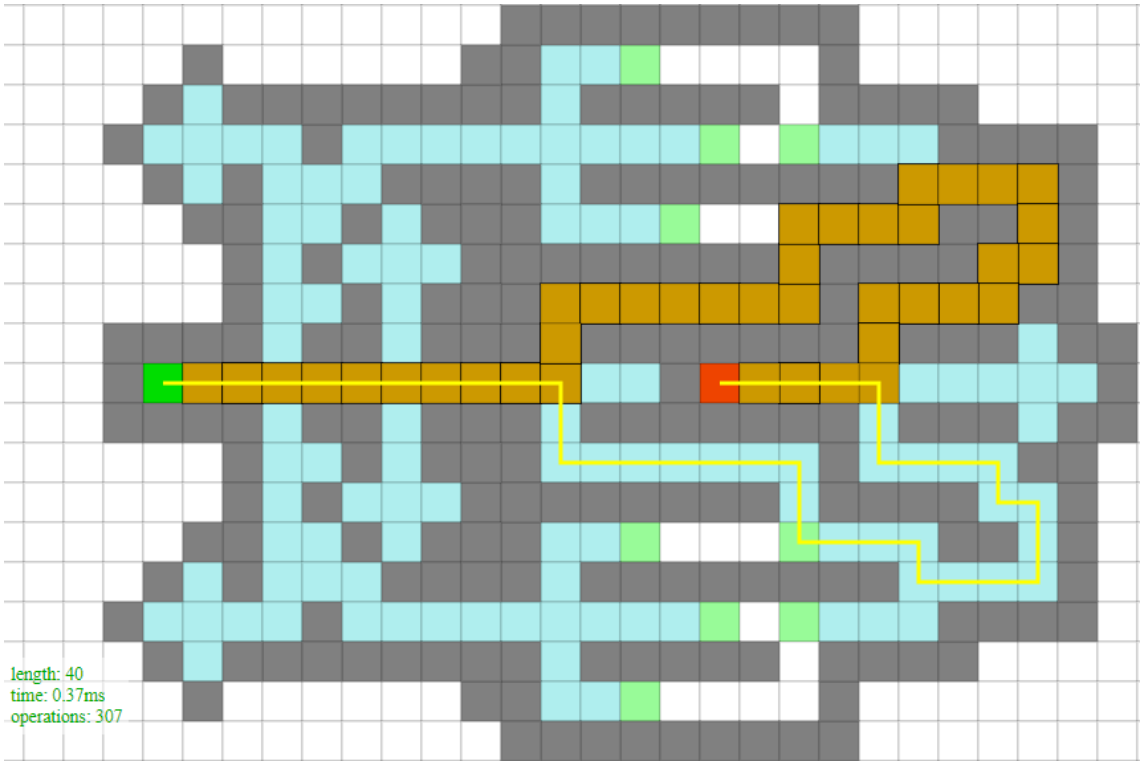


Figura 4.11: Grafo tipo mazmorra, ampliado

algoritmo de Dijkstra cobra más sentido, pues una adaptación de dicho algoritmo se utiliza para resolver el problema de los *K*-Caminos.

Para el rendimiento comparativo hemos utilizado dos tipos de mapas: un mapa sin obstáculos, con los vértices inicial y final separados en línea recta (Figura 4.12) y otro mapa de tipo cuadrícula, con los vértices inicial y final separados en diagonal (Figura 4.13).

Con los experimentos que se plantea en esta sección podemos ver como afecta el factor de ramificación a la búsqueda. Aunque con un mapa es complicado medir el factor de ramificación, en el caso de la cuadrícula el factor de ramificación es menor que en el caso del grafo sin obstáculos pues la búsqueda sin obstáculos se expande en las cuatro direcciones mientras que en la cuadrícula se expande solo en dos direcciones.

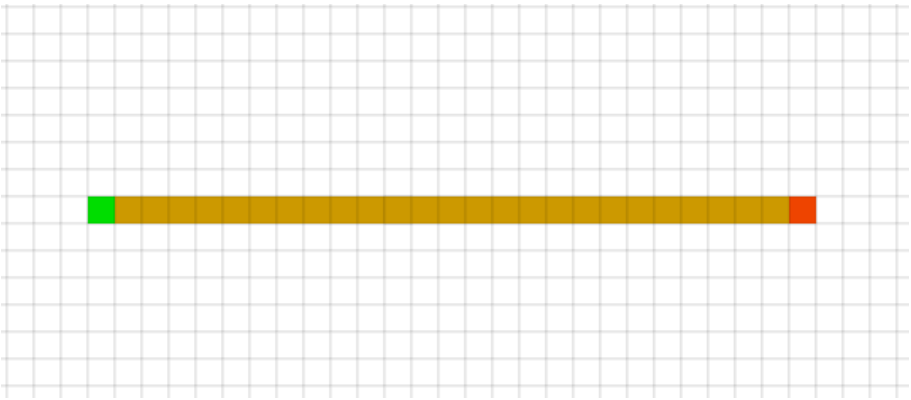


Figura 4.12: Grafo sin obstáculos

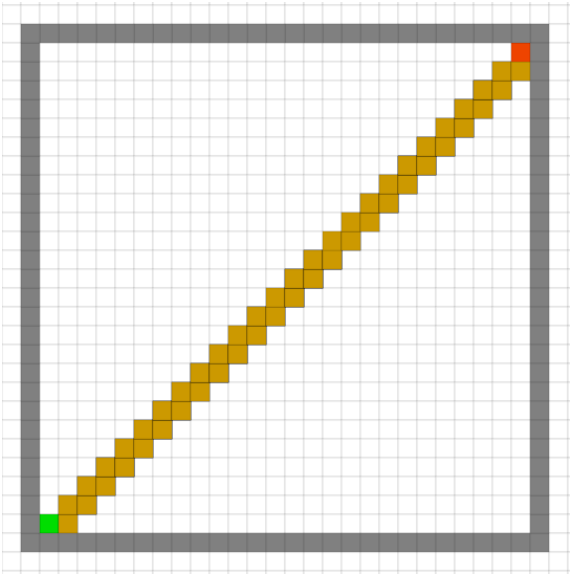


Figura 4.13: Grafo tipo cuadrícula

Incrementaremos la longitud del camino óptimo progresivamente y realizaremos tres mediciones para cada longitud y algoritmo, presentando la media de los resultados obtenidos. En cada prueba medimos el tiempo empleado, el número de nodos abiertos y el número de nodos cerrados en cada uno de los tres algoritmos: Dijkstra, la versión unidireccional de nuestro algoritmo (ASP Uni.) y la versión bidireccional (ASP Bi.).

Mapa sin Obstáculos				
Longitud	Algoritmo	N. Abiertos	N. Cerrados	Tiempo ($\overline{x}$ ms)
10	Dijkstra	353	298	100,54
	ASP Uni.	368	270	113,89
	ASP Bi.	121	98	63,161
25	Dijkstra	1.577	1452	125,52
	ASP Uni.	1.410	1321	105,63
	ASP Bi.	636	573	95,013
50	Dijkstra	5701	5471	198,48
	ASP Uni.	5.403	5.196	224,34
	ASP Bi.	2.650	2.513	211,47
100	Dijkstra	19.033	18.735	833,89
	ASP Uni.	18.963	18.963	992,51
	ASP Bi.	10.662	10.369	868,17

Tabla 4.1: Resultados obtenidos del mapa abierto. En la columna de tiempo mostramos la media de las tres mediciones realizadas en milisegundos.

Lo primero que se observa al ver la Tabla 4.1 es la similitud de los tiempos entre las resultados obtenidos con Dijkstra y con el algoritmo unidireccional. El algoritmo bidireccional también arroja tiempos similares con una longitud de camino alta, que sospechamos se debe a que la implementación que hemos realizado pueda mejorarse y hacerse más eficiente, pues a juzgar por el número de nodos abiertos y cerrados, debería de haber conseguido un rendimiento mejor.

Si comparamos la búsqueda de camino de longitud 10 con la búsqueda de camino 25 en los algoritmos de Dijkstra y Unidireccional de la Tabla 4.1 vemos que existe una gran diferencia entre el número de nodos abiertos y cerrados pero el tiempo se mantiene en un rango de valores similar. Creemos que, para espacios de búsqueda pequeños, tiene más peso el tiempo que tarda el programa en inicializar las variables y estructuras de datos que la propia búsqueda. Este efecto también se observa en los resultados de la Tabla 4.2, donde la diferencia de los tiempos apenas es significativa en los casos de longitud 10, 25 y 50.

En la comparación entre la versión unidireccional y bidireccional de la Tabla 4.1 se puede observar como la diferencia entre el número de nodos abiertos y cerrados entre ambos algoritmos se torna muy significativa conforme aumenta la talla del problema (Mapa Sin obstáculos - Longitud 100).

Si comparamos la respuesta de los algoritmos en función del factor de ramificación, en el caso del mapa abierto de la Figura 4.12 (Tabla 4.1), al tener un factor de ramificación mayor, el espacio de búsqueda aumenta, y es donde se ve mejor las ventajas de la propuesta bidireccional, que reduce el número de nodos abiertos y cerrados a la mitad respecto a la versión unidireccional.

En el mapa tipo cuadrícula (Tabla 4.2) no tendría tanto efecto las ventajas de usar el algoritmo bidireccional, lo cual es consistente con el análisis de complejidad de la propuesta de la solución (Sección 3.4.2).

Mapa Tipo Cuadrícula				
Longitud	Algoritmo	N. Abiertos	N. Cerrados	Tiempo ($\overline{x}$ ms)
10	Dijkstra	36	36	6,73
	ASP Uni.	36	36	10,57
	ASP Bi.	35	32	8,12
25	Dijkstra	182	182	19,87
	ASP Uni.	182	182	14,26
	ASP Bi.	128	118	25,22
50	Dijkstra	676	676	18,31
	ASP Uni.	676	676	36,57
	ASP Bi.	571	554	21,50
100	Dijkstra	2.600	2.600	38,46
	ASP Uni.	2.600	2.600	118,07
	ASP Bi.	2.284	2.196	60,21

Tabla 4.2: Resultados obtenidos del mapa tipo cuadrícula. En la columna de tiempo mostramos la media de las tres mediciones realizadas en milisegundos.

CAPÍTULO 5
Conclusiones

Con la propuesta realizada en este trabajo hemos logrado nuestros objetivos, en concreto el de diseñar un algoritmo de carácter general para resolver el *Another Solution Problem* aplicado al *Problema del Camino más Corto*.

Hemos mejorado además la primera propuesta de solución aplicando la técnica de búsqueda bidireccional, reduciendo el espacio de búsqueda de manera efectiva sin afectar a la calidad de la solución. Por otro lado, cabe mencionar que sería necesario optimizar la implementación para su uso en escenarios reales.

Como futuro trabajo proponemos el estudio de la variante del ASP cuando se proporcionan varias soluciones (n-ASP), el estudio de heurísticas para su aplicación al ASP-PCC y la aplicación del trabajo realizado sobre el ASP-PCC a la resolución del problema de los *K*-Caminos.

Bibliografía

[1] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[2] Kevin C. Cap. Visualizer for graph search algorithms. GitHub, https://github.com/kcappieg/graph-search-visualizer, 2018. [Online; accedido 2-Julio-2019].

[3] Dennis de Champeaux. An improved bidirectional heuristic search algorithm. *Journal of the ACM*, 1977.

[4] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[5] Stefan Edelkamp, Morteza Lahijanian, Daniele Magazzeni, and Erion Plaku. Integrating temporal reasoning and sampling-based motion planning for multigoal problems with dynamics and time windows. *IEEE Robotics and Automation Letters*, 3(4):3473–3480, 2018.

[6] Michael Günther et al. Symbolic calculation of k-shortest paths and related measures with the stochastic process algebra tool caspa. *Workshop on Dynamic Aspects in Dependability Models for Fault-Tolerant Systems (DYADEM-FTS)*, pages 13–18, 2010.

[7] David Ferguson, Maxim Likhachev, and Anthony (Tony) Stentz. A guide to heuristic-based path planning. In *Proceedings of the International Workshop on Planning under Uncertainty for Autonomous Systems, International Conference on Automated Planning and Scheduling (ICAPS)*, 2005.

[8] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms*, pages 319–333, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[9] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Efficient point-to-point shortest path algorithms. In *ALENEX*, 2006.

[10] Meir Goldenberg, Ariel Felner, Roni Stern, Guni Sharon, and Jonathan Schaeffer. A* variants for optimal multi-agent pathfinding. *Proceedings of the 5th Annual Symposium on Combinatorial Search, SoCS 2012*, pages 157–158, 2012.

[11] Florian Grenouilleau, Willem-Jan van Hoeve, and J. N. Hooker. A multi-label a* algorithm for multi-agent pathfinding. *The International Conference on Automated Planning and Scheduling (ICAPS)*, 2019.

[12] Tingting Han, Joost P. Katoen, and B. Damman. Counterexample generation in probabilistic model checking. *IEEE transactions on software engineering*, 35(2):241–257, 2009. 10.1109/TSE.2009.5.

[13] Hsu-Chieh Hu and Stephen Smith. Using bi-directional information exchange to improve decentralized schedule-driven traffic control. *The International Conference on Automated Planning and Scheduling (ICAPS)*, 2019.

[14] Michael Katz, Emil Keyder, Florian Pommerening, and Dominik Winterer. Oversubscription planning as classical planning with multiple cost functions. *The International Conference on Automated Planning and Scheduling (ICAPS)*, 2019.

[15] M Kubale. *Graph Colorings*. American Mathematical Society, 2004.

[16] Lukas Kuhn, Tim Schmidt, Bob Price, Johan Kleer, Rong Zhou, and Minh Do. Heuristic search for target-value path problem. *First International Symposium on Search Techniques in Artificial Intelligence and Robotics*, 2008.

[17] Robert Lieck and Vien Ngo Marc Toussaint. Exploiting variance information in monte-carlo tree search. *ICAPS 2017 - Heuristics and Search for Domain-independent Planning (HSDIP) Whorkshop*, 2017.

[18] Yi-Hsuan Lin, Tung-Mei Ko, Tyng-Ruey Chuang, and Kwei-Jay Lin. Open source licenses and the creative commons framework: License selection and comparison. *Journal of Information Science and Engineering*, 22(1):1–17, 2006.

[19] Carlos Linares López, Sergio Jiménez Celorrio, and Ángel García Olaya. The deterministic part of the seventh International Planning Competition. *Artificial Intelligence*, 223:82–119, 2015.

[20] Carlos Linares López and Abdallah Saffidine. A preliminary selection of problems in heuristic search. In *The eighth Annual Symposium on Combinatorial Search (SoCS 2015)*, 2015.

[21] Chuan Luo, Shaowei Cai, Kaile Su, and Wenxuan Huang. Ccehc: An efficient local search algorithm for weighted partial maximum satisfiability. *Artifical Intelligence*, 243:26–44, 2017.

[22] Álvaro Torralba. From qualitative to quantitative dominance pruning for optimal planning. *ICAPS 2017 - Heuristics and Search for Domain-independent Planning (HSDIP) Whorkshop*, 2017.

[23] Alexander Nadel and Vadim Ryvchin. Chronological backtracking. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 111–121. Springer, 2018.

[24] Hootan Nakhost. *Random Walk Planning: Theory, Practice, and Application*. PhD thesis, University of Alberta, 2013.

[25] Gerald Paul, Gabriele Röger, Thomas Keller, and Malte Helmert. Optimal solutions to large logistics planning domain problems. *ICAPS 2017 - Heuristics and Search for Domain-independent Planning (HSDIP) Whorkshop*, 2017.

[26] Ira Pohl. Bi-directional search. *PREPARED FOR THE U.S. ATOMIC ENERGY COMMISSION UNDER CONTRACT NO. AT(04-3)-515*, 1969.

[27] Xiaoqing Qu and Xiaobo Li. A 3d surface tracking algorithm. *Comput. Vis. Image Underst.*, 64(1):147–156, 1996.

[28] Michael N. Rice and Vassilis Tsotras. Bidirectional a* search with additive approximation bounds. *Proceedings of the 5th Annual Symposium on Combinatorial Search, SoCS 2012*, pages 80–87, 2012.

[29] Elena Sánchez-Nielsen and Mario Hernández-Tejera. Heuristic algorithms for fast and accurate tracking of moving objects in unrestricted environments. In Massimo De Gregorio, Vito Di Maio, Maria Frucci, and Carlo Musio, editors, *Brain, Vision, and Artificial Intelligence*, pages 507–516, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[30] Uwe Schöning. A probabilistic algorithm for k-sat and constraint satisfaction problems. In *Proceedings of the 40th Annual Symp. Foundations of Computer Science*, pages 410–414, 1999.

[31] Takahiro Seta. The complexities of puzzles, cross sum, and their another solution problems (asp). Master's thesis, Senior Thesis for the Degree of Bachelor Science, University of Tokyo, 2002.

[32] Eshed Shaham, Ariel Felner, Jingwei Chen, and Nathan R. Sturtevant. The minimal set of states that must be expanded in a front-to-end bidirectional. *SoCS 2017: The 10th Annual Symposium on Combinatorial Search*, 2017.

[33] Anthony Stentz and Is Carnegle Mellon. Optimal and efficient path planning for unknown and dynamic environments. *International Journal of Robotics and Automation*, 10:89–100, 1993.

[34] Asher Stern, Roni Stern, Ido Dagan, and Ariel Felner. Efficient search for transformation-based inference. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers-Volume 1*, pages 283–291. Association for Computational Linguistics, 2012.

[35] Yuri Takhteyev and Andrew Hilts. Investigating the geography of open source software through github. *Project Open Source | Open Access, KMDI*, 2010.

[36] Aristomenis Tressos and Nikos Sklavounos. A* and bfs searching visualization. GitHub, https://github.com/Tressos-Aristomenis/A-Star-and-BFS-Searching-Visualization, 2018. [Online; accedido 2-Julio-2019].

[37] I-Lun Tseng, Huan-Wen Chen, and Che-I Lee. Obstacle-aware longest-path routing with parallel milp solvers. In *Proceedings of the World Congress on Engineering and Computer Science 2010 Vol II WCECS 2010*, pages 827–831, 2010.

[38] Nobuhisa Ueda and Tadaaki Nagao. NP-completeness Results for NONOGRAM via Parsimonious Reductions. Technical report, Department of Computer Science, Tokyo Institute of Technology, 1996.

[39] T. Wang, Y. Lu, D. Zhang, G. Cui, and W. Wang. Backtracking algorithm based multi-relay selection in high altitude platforms system. In *2018 IEEE/CIC International Conference on Communications in China (ICCC)*, pages 877–881, 2018.

[40] Wei Wei, Chuang Li, Wei Liu, and Dantong Ouyang. Landmark-biased random walk for deterministic planning. In Zhi-Hua Zhou, Qiang Yang, Yang Gao, and Yu Zheng, editors, *Artificial Intelligence*, pages 155–165, Singapore, 2018. Springer Singapore.

[41] Jan Wiebe. Foundations of artifical intelligence, lectures. Technical report, University of Pittsburgh, 2003.

[42] Wan Yeung Wong, Tak Pang Lau, and Irwin King. Information retrieval in p2p networks using genetic algorithm. In *In Proceedings of the 14th International World Wide Web Conference*, pages 922–923, enero 2005.

[43] Xueqiao Xu. Pathfinding.js: A comprehensive path-finding library in javascript. GitHub, https://github.com/qiao/PathFinding.js, 2017. [Online; accedido 2-Julio-2019].

[44] Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 86(5):1052–1060, 2003.

[45] Jin Y Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.

[46] Kunjie Yu, JJ Liang, BY Qu, Zhiping Cheng, and Heshan Wang. Multiple learning backtracking search algorithm for estimating parameters of photovoltaic models. *Applied energy*, 226:408–422, 2018.

[47] Shlomo Zilberstein. Using anytime algorithmsin intelligent systems. *AI Magazine*, 17(3):73–83, 1996.