UNIVERSIDAD NACIONAL DE INGENIERÍA

FACULTAD DE CIENCIAS

ESCUELA PROFESIONAL DE CIENCIA DE LA COMPUTACIÓN

Heurísticas y Metaheurísticas para resolver el Problema de Camino Mínimo en Carreteras

PROYECTO DE TESIS III

Autor: Franz Rony Ventocilla Tamara

Asesor: Jaime Osorio Ubaldo

4 de enero de 2023

Resumen

En este trabajo se comparará la heurística A* y el algoritmo metaheurístico Algoritmo Genético (AG) sobre grafos con distinta cantidad de nodos obtenidos de carreteras del mapa. Teniendo como referencia el algoritmo base Dijkstra. Utilizando las métricas: Tiempo y eficiencia.

Índice general

Índice de Acrónimos

ETC	Etcétera
P2P	Point to Point
A*	A Star, A estrella
OSM	OpenStreetMap
AG	Algoritmo Genético
VNS	Variable Neighborhood Search
MBF	Mean Best Fitness

Agradecimientos

Agradezco a mi familia por su apoyo constante en el transcurso de mi vida universitaria, y a mi asesor por los consejos brindados.

Capítulo 1

Introducción

En la actualidad se utilizan muy a menudo algoritmos para hallar la ruta más corta o el camino mínimo sobre carreteras, por ejemplo en las Apps de mapas como Google Maps o Waze, estos cálculos tienen que ser rápidos y dar la mejor respuesta posible. El problema del camino mínimo fue uno de los más estudiados según la literatura. Este consiste en que dado un punto inicial y un punto final, se tiene que hallar el camino óptimo. Dependiendo del contexto del problema, la palabra "óptimo"puede significar reducir costos, reducir tiempo, reducir distancias, etc. En esta tesis se realizará la comparación del algoritmo A* utilizando distintas funciones heurísticas y un algoritmo genético (AG) para hallar el camino óptimo en el problema de redes en carreteras o redes viales. Los grafos de entrada utilizados para la comparación serán obtenidos de localizaciones de mapa reales y las métricas utilizadas para la comparación serán: el tiempo, el número de vértices expandidos y la eficiencia con respecto al algoritmo exacto Dijkstra.

Capítulo 2

Planteamiento del Problema

2.1. Descripción del Problema

En el campo de la algoritmia, uno de los problemas más estudiados es el de hallar el camino mínimo en un grafo. Esto es debido a la cantidad de aplicaciones que existen, por lo que a través del tiempo también existieron muchas soluciones a este problema. Entre las más populares están los algoritmos exactos: Dijkstra, Bellman-Ford, etc. y sus variantes bidireccionales. También existen algoritmos que hacen uso de heurísticas para disminuir el tiempo de ejecución, la más popular para este problema es A*. Y también existen metaheurísticas que buscan ese mismo objetivo, para este trabajo se utilizará el Algoritmo Genético (AG). Dependiendo de la aplicación del problema es preferible escoger uno u otro. Por ello nace el deseo de conocer sus desempeños en distintos grafos aplicados al problema de redes de carreteras.

2.2. Formulación del Problema

¿Qué algoritmo de camino mínimo tiene mejor desempeño aplicado en carreteras?

2.3. Importancia y Justificación del estudio

En el problema de hallar el camino mínimo en un grafo aplicado en redes de carreteras o redes viales aún existen dudas, como por ejemplo en qué tipo

de grafo trabaja mejor cada algoritmo, o si existe algún algoritmo que siempre encuentre el camino mínimo en menos tiempo. Este trabajo comparará los algoritmos A*, AG y Dijkstra en redes de carreteras con la intención de aclarar ciertos puntos donde la literatura de trabajos anteriores carece.

2.4. Objetivos de la Investigación

Objetivo principal

- Comparar el desempeño entre los algoritmos A*, AG y Dijkstra P2P utilizando las métricas: tiempo y eficiencia.

Objetivos secundarios

- Generar grafos de gran cantidad de nodos mediante metadatos de mapas reales.
- Implementar A* y Dijkstra P2P y un AG para el problema del camino mínimo.
- Implementar el AG de forma que se obtenga buenas soluciones para el problema del camino mínimo en grafos con distintos números de nodos.

Capítulo 3

Estado del arte

- **A Note on Two Problems in Connexion with Graphs:** Un artículo publicado por E.W Dijkstra en 1959 [7], donde Dijkstra propone soluciones a dos problemas: El problema de construir un árbol de longitud mínima entre n nodos de un grafo y el problema de encontrar el camino mínimo entre dos nodos P y Q de un grafo. Su solución al segundo problema se hizo bastante popular y actualmente sigue siendo utilizada en distintas aplicaciones de caminos óptimos, a pesar de que existen métodos más eficientes para la expansión del camino en el grafo.

- **Problem-Solving Methods in Artificial Intelligence:** Es un libro escrito principalmente por Nils J. Nilsson [18] 1971, en el cual mencionan por primera vez el uso de heurísticas para resolver el problema de camino mínimo en un grafo, que posteriormente se hace conocido como el algoritmo A*, caracterizándose por ser eficiente en la expansión del camino, y por hallar el camino óptimo en poco tiempo.

- **Correction to a formal basis for the heuristic determination of minimum cost paths:** Es un artículo publicado en 1972 [14], donde Nils et al. corrigen el método presentado en el libro *Problem-Solving Methods in Artificial Intelligence*, y realizan nuevos corolarios, corrigiendo algunos ya existentes, entre una de las correcciones más importantes está la prueba de optimización, que no utilizaba el principio de consistencia que mencionaron en su libro.

- **Computing the shortest Path: A* Search Meets Graph Theory** Es un artículo escrito por Goldberg, Harrelson 2005 [13] donde se propone resolver el problema del camino mínimo P2P (Point to point) agregando límites basados en landmarks y la desigualdad triangular al algoritmo A*. Para las métricas usaron la eficiencia, que en el artículo es definida como el número de vértices del camino mínimo dividido entre el número de vértices escaneados por el algoritmo, esta eficiencia es hallada en forma de porcentaje. Otro parámetro usado fue el tiempo, pues se complementa bien con la eficiencia, ya que sirve para tener un mayor entendimiento del rendimiento del argoritmo. Una de las conclusiones más importantes de este artículo es que este algoritmo mejorado de A* es mejor en eficiencia que tanto Dijkstra y Dijkstra bidireccional(ejecución del algoritmo de Dijkstra desde dos direcciones, incio y final, hacia algún punto medio del grafo), pero el tiempo de ejecución es mayor.

- **Point to Point Shortest Path Algorithms with Preprocessing** Es un artículo escrito por Goldberg et al. 2007 [12], donde se muestra la idea de algoritmos para resolver el problema del camino mínimo P2P usando preprocesamiento. Los algoritmos basados en preprocesamiento tienen dos partes: Algoritmo de preprocesamiento y algoritmo de consulta. La primera parte es la que toma más tiempo de ejecución y no es práctico para usos en aplicaciones como redes viales. Entre los métodos para realizar el algoritmo de preprocesamiento se encuentran la selección de landmarks, y encontrar límites superiores para posteriormente intentar alcanzarlos. Como conclusión menciona que existen muchas preguntas abiertas para este tipo de algoritmos, entre ellas la justificación teórica del algoritmo y también si es posible realizar el algoritmo de preprocesamiento en un tiempo mejor que $O(n^2)$.

- **The Quest For Artificial Intelligence:** Es un libro escrito por Nils J. Nilsson 2009 [19], donde da una idea más general sobre el uso del algoritmo A* aplicado en robots móviles, y describe cómo fue que nació

la idea de crear el algoritmo A*, también plantea que debería ser usado para juegos de computadora, lo cual actualmente se cumple y es muy popular ver este algoritmo en distintos tipos de juegos, incluso grandes frameworks para crear videojuegos, o también conocidos como Game Engines lo implementan como un algoritmo standard para obtener el camino mínimo entre dos puntos.

- **A Review and Evaluations of Shortest Path Algorithms:** Es un artículo publicado en 2013 [16] donde se realiza la comparación entre los algoritmos Dijkstra, Floyd-Warshall y el algoritmo de Bellman-Ford. Evalúan la complejidad temporal de cada uno de ellos. También dan un marco bastante interesante sobre cómo podría ser resuelto el problema del camino mínimo mediante un algoritmo genético (AG). Sin embargo, este no entra en la comparación de complejidad temporal. La conclusión de este trabajo, por el lado de las complejidades temporales, es que los tres algoritmos resuelven el problema con un rendimiento aceptable. Por el lado de las soluciones se llegó a que tanto Dijkstra, Bellman-Ford como Floyd-Warshall producen una única óptima solución, a diferencia del enfoque evolutivo que produce diferente número de óptimas soluciones y la solución puede variar en cada ejecución del algoritmo.

- **A Hybrid Metaheuristic for Routing in Road Networks** Es un artículo publicado en el año 2015 [6] donde los autores resuelven el problema del camino mínimo utilizando metaheurísticas. Para este trabajo se combinó dos metauheurísticas: Algoritmo Genético (AG) como estructura principal del algoritmo, y Búsqueda de Vecindad Variable (VNS) para obtener buenas soluciones iniciales y también como función de mutación. Para los datos de entrada utilizaron el dataset DIMACS, que brinda varios grafos de mapas de USA. Finalmente compararon sus resultados con Dijkstra, y un enfoque utilizando programación entera (IP) en Cplex y Gurobi. En los resultados, el algoritmo utilizando metaheurísticas tuvo

mejores resultados que los otros dos. El peor algoritmo fue IP, superado por Dijkstra.

- **Explaining the Performance of Bidirectional Dijkstra and A* on Road Networks** Es un artículo publicado en 2017 [21] en el cual comparan los algoritmos Dijkstra Bidireccional con MM, Dijkstra bidireccional es la variante del algoritmo que realiza dos veces el algoritmo de Dijkstra, uno desde el vértice inicial a algún vértice intermedio, y otro desde el vértice final al vértice intermedio. Por otro lado MM es la variante de A* que realiza dos veces el algoritmo de A*, similar al de Dijkstra Bidireccional, uno desde el vértice inicial a algún vértice intermedio y otro desde el vértice final hasta el vértice intermedio. Para hallar el vértice intermedio en Dijkstra bidireccional se proporciona tres posibles variantes, mientras que en MM se realiza con el algoritmo *Meet on the Midle*, de ahí sus siglas. Como métricas a comparar usó el tiempo y los nodos que expande el camino. Como conclusión llegó a que un algoritmo no siempre es mejor que el otro.

- **Analysis of Dijkstra's Algorithm and A* Algorithm in Shortest Path Problem** Es un artículo escrito en 2019 [20] donde se re realizó la comparación entre los algoritmo Dijkstra y A* sobre un mapa de carreteras obtenido de Google Maps. Para este trabajo se realizó la comparación sobre un único grafo, desde una ubicación única, en este caso es la ubicación de la universidad a la que pertenece el autor, hacia dos destinos diferentes. La comparación se basó en el tiempo de ejecución de ambos algoritmos. Sus conclusiones fueron que en escalas de mapa pequeñas ambos algoritmos funcionan bien, pero en escalas de mapa grandes, A* provee una solución más rápida que Dijkstra.

Capítulo 4

Marco teórico

En esta sección se definirán algunos conceptos necesarios para el desarrollo de la tesis.

4.1. Grafo

En el libro [9] *cap 1* se define un grafo como una estructura $G(V, E)$ que consiste en un conjunto de vértices $V = \{v_1, v_2, v_3...\}$ y un conjunto de aristas $E = \{e_1, e_2, e_2, ...\}$, donde cada arista está delimitada por dos vértices. Tanto V y E son finitos, por lo tanto decimos que un grafo G es finito. Un ejemplo de grafo es la Figura 4.1 donde el conjunto de vértices es $V = \{v_1, v_2, v_3, v_4, v_5\}$ y el conjunto de aristas $E = \{e_1, e_2, e_3, e_4, e_5\}$.

Grado de un vértice v: El grado de un vértice que es denotado como $d(v)$, es el número de veces en que v es punto final de una o varias aristas. A un vértice con grado cero $d(v) = 0$ se le conoce como vértice aislado. Por ejemplo, en la Figura 4.1 v_3 es un vértice aislado porque $d(v) = 0$.

4.2. Camino

De acuerdo a Even [9, pags 2-4] y Sedgewick [22] un camino P es una secuencia de vértices y aristas sucesivas. De forma: P inicia con un vértice v_0 seguido de una arista e_1, seguido de un vértice v_1, y así sucesivamente. Para este trabajo se considerarán solo caminos finitos, un camino finito siempre termina en un vértice v. Suponiendo que el camino P inicia en v_0 y termina en v_1, al

número de aristas entre v_0 y v_1 se le conoce como longitud l del camino. Si $l = 0$ se dice que el camino P es vacío. Un camino simple es aquel donde todos los vértices y aristas son diferentes. Un ciclo es un camino simple con la excepción de que el vértice inicial y final son el mismo.

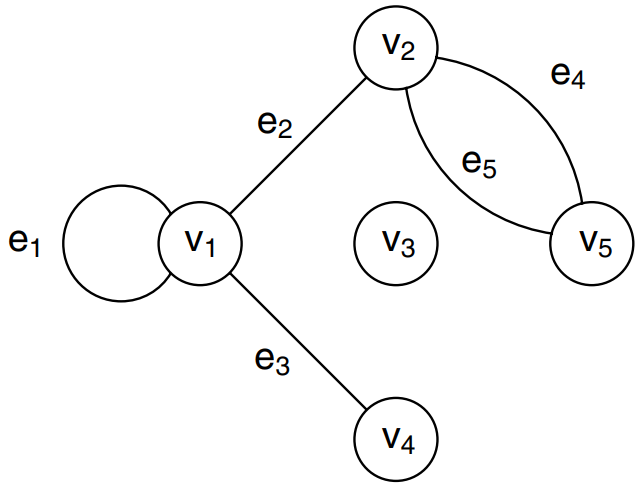


FIGURA 4.1: Ejemplo de grafo. *Fuente: [9]*

4.3. Grafo dirigido

Un grafo $G(V, E)$ es un grafo dirigido si cada par de vértices de las aristas están ordenados. Según [5] *pags 590-593* para representar los grafos dirigidos de forma computacional existen dos formas:

1. **Lista de adyacencia (Adj):** La representación mediante lista de adyacencia de un grafo $G = (V, E)$ consiste en un arreglo Adj de $|V|$ listas, cada vértice es un elemento de este arreglo. Para cada $u \in V$, la lista de adyacencia $Adj[u]$ contiene todos los vértices v que lo conectan mediante una arista, esto quiere decir $(u, v) \in E$. Un ejemplo de lista de adyacencia se encuentra en la Figura 4.2 (b), donde se puede observar un grafo dirigido G con $V = \{1, 2, 3, 4, 5, 6\}$ y $|E| = 8$, su representación en grafo está en (a).

2. **Matriz de adyacencia (A)** En la representación mediante matriz de adyacencia de un grafo $G = (V, E)$ se debe tener cada vértice numerado de 1 a $|V|$, la numeración puede ser arbitraria. Entonces la representación del grafo G consiste en una matriz $A = (a_{ij})$ de tamaño $|V| \times |V|$, de manera que:

   - $a_{ij} = 1$ si $(i, j) \in E$. Esto quiere decir que cuando los vértices i, j están conectados mediante una arista, la matriz será llenada con 1. En caso que las aristas tengan un peso asignado, entonces es posible llenar la matriz con dichos pesos.
   - En otro caso $a_{ij} = 0$.

   Un ejemplo de La representación de un grafo mediante matriz de adyacencia se encuentra en la Figura 4.2 (c).

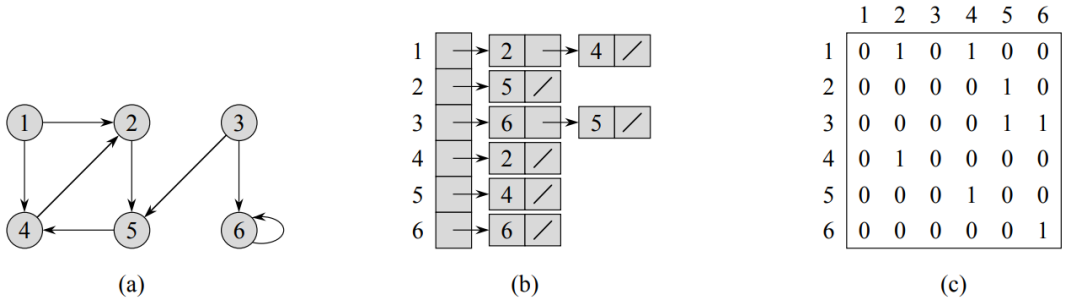


FIGURA 4.2: Ejemplos de grafos dirigidos. *Fuente: [5]*

4.4. Problema del camino más corto

En un grafo dirigido, todo camino entre dos puntos tiene asociado un peso del camino, el cual es la suma de los pesos de las aristas del grafo. Con esto se puede formular el problema: "¿Cuál es el camino mínimo entre dos vértices?". La solución más obvia a este problema podría ser calcular todos los caminos posibles entre estos dos vértices, guardarlos en una matriz de distancias, y hallar el mínimo valor en dicha matriz. Pero esta solución es demasiado costosa, por lo que actualmente existen varios algoritmos que resuelven este problema

de manera más eficiente, entre los más conocidos están: el algoritmo de Dijkstra, Bellman-Ford, A*, Floyd-Warshall, Johnson.

4.4.1. Camino mínimo

Según Sedgewick [22] *pags 279-281* El camino mínimo o más corto entre dos vertices s y t en una red, es un camino dirigido de s a t con la propiedad de que ningún otro camino de ese tipo tenga un peso menor. Teniendo esta definición, se debe tener en cuenta:

1. Si t no es alcanzable desde s, no existe camino, por lo tanto, no existe camino más corto.

2. Si existen varios caminos con el mismo peso de un vértice a otro, basta con escoger uno de ellos.

4.4.2. Variantes

Existen variantes al problema de hallar el camino mínimo:

1. **Camino mínimo entre dos vértices (P2P):** También conocido como *Point to point*. Teniendo un vértice inicial s y un vértice final t, se tiene que encontrar el camino mínimo de s a t

2. **Camino mínimo desde un vértice único:** Teniendo un vértice inicial s, se tiene que encontrar los caminos más cortos desde s a cada uno de los otros vértices del grafo.

3. **Camino mínimo de todos los pares de vértices:** Consiste en encontrar los caminos más cortos que conectan cada par de vértices en el grafo. Comúnmente se le suele denotar como el conjunto de caminos V^2.

4.5. Dijkstra

El algoritmo de Dijkstra resuelve el problema de camino mínimo desde un punto único en grafos con pesos no negativos. Este algoritmo también

puede resolver el problema de camino mínimo entre dos puntos deteniendo el algoritmo cuando alcance el vértice final t.
En el artículo *A note on two Problems in Connexion with Gaphs (1959)* [7], Dijkstra propone la solución a dos problemas:

1. Construir un árbol de longitud mínima entre los n nodos de un grafo.

2. Encontrar el camino mínimo entre dos nodos P y Q de un grafo.

De los cuales, su solución al segundo problema es la más popular. Esta se basa en el conocimiento de: *"Si existe un camino mínimo de P a Q, y R es un nodo de este camino, entonces el subcamino de P a R es el camino mínimo de P a R"*. Dijkstra considera a los vértices como nodos y a las aristas como ramas. Teniendo esto claro, divide a los nodos en 3 conjuntos:

- **A**: Los nodos para los que se conoce el camino mínimo desde P.

- **B**: Los nodos que están conectados a al menos uno de los nodos del conjunto A. De este conjunto se escongerá al siguiente nodo.

- **C**: El resto de nodos.

Las ramas también se subdividen en 3 conjuntos:

1. Las ramas que aparecen en los caminos mínimos desde P hasta los nodos del conjunto A.

2. Las ramas de las cuales la siguiente rama que se colocará en el conjunto 1 será escogida. Cada rama de este conjunto conducirá a cada nodo del conjunto B.

3. El resto de ramas.

El resto es un proceso iterativo con dos pasos:

- **Paso 1**: Se considera todas las ramas r que conectan el último nodo en A, con los nodos R en B o C. Si R pertenece al conjunto B, se investigará si el camino de P a R es mejor que el ya conocido, que usa las ramas del

conjunto 2. Si el camino no mejora, se rechaza r. Pero si el camino mejora, se actualizan las ramas en el conjunto 2. Si el nodo R pertenece a C, es agregado a B y la rama r es agregada al conjunto 2.

- **Paso 2**: Considerando grafos dirigidos, cada camino tendrá una distancia o peso asociado. El nodo con la mínima distancia desde P se agrega del conjunto B al conjunto A. Y su rama corresponiente pasa del conjunto 2 al 1. Se retorna al **Paso 1** y se repite el proceso hasta que el nodo final Q sea puesto en el conjunto A.

El algoritmo de Dijkstra se puede entender como el algoritmo Breadth-first search, pero con la variación de que los pesos no son todos 1, sino son los pesos respectivos a las aristas.

4.6. A*

A* o Algoritmo estrella es un algoritmo de búsqueda que se propuso en el libro [18] de Nils J. Nillson, que posteriormente fue corregido en el artículo [14] 1972. Este algoritmo resuelve el problema de encontrar un camino óptimo entre dos puntos, maximizando la eficiencia de la búsqueda, y al mismo tiempo garantizando un costo mínimo. Esto se debe a que utiliza una función heurística h(n), de forma:

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

Donde:

- **f(n)** Es una función cuyo valor en cualquier vértice n es el costo óptimo desde el vértice inicial al vértice n.
- **g(n)** Es un estimado del costo del camino mínimo del vértice inicial al vértice n. Usualmente se le considera como la suma de los pesos del camino mínimo hasta n.

- **h(n)** Es una estimación del costo desde n hasta el vértice final. Se entiende como una función heurística usada para aproximar la distancia del vértice inicial al vértice final.

Para garantizar que A* proporcione una solución correcta, h(x) debe ser admisible. Para que A* garantice que nunca expanda un nodo más de una vez, la función heurística h(x) debe ser consistente.

- **Admisibilidad de A*:** Según Nils J. Nilsson and Lee [18] *pags. 59-61*, un algoritmo de búsqueda es admisible si para cualquier grafo, el algoritmo siempre halla un camino óptimo si este camino existe. El algoritmo A* será admisible si h(n) es menor que el verdadero costo desde n hasta el vértice final.

$$h(n) \leq k(n,t)$$

  Donde t es el vértice final y k(n,t) es el costo real del camino de n a t.

- **Optimización de A*:** Si la heurística es admisible y consistente, entonces A* será óptimo. Es decir, no se expanderá más nodos que otro algoritmo admisible. Decimos que una heurística es consistente si cumple:

$$h(m) - h(n) \leq k(m,n)$$

  Donde m y n son cualquier dos vértices del grafo, y k(m,n) es el costo real del camino de m a n.

4.7. Metaheurísticas

Según Glover y Kochenberger pags. 1-2 (2006) [10] se entiende por Metaheurística a los métodos que son capaces de escapar de óptimos locales y realizar búsquedas robustas en un espacio de solución. También está relacionado con los algoritmos inspirados en la naturaleza, y que generalmente funcionan mejor que una heurística. Todos los algoritmos metaheurísticos se

apoyan en la aleatoriedad y búsqueda local para encontrar soluciones óptimas, y suelen ser de mucha utilidad para problemas de optimización difíciles. Las características más importantes de las metaheurísticas son:

- Tiempo de ejecución razonable.
- No hay garantía de alcanzar la solución óptima.

Entre los algoritmos metaheurísticos más populares tenemos: Recocido Simulado, Algoritmo Genético, Evolución Diferencial, Optimización de Colonias de Hormigas, Algoritmos de Abejas, Optimización de Emjambre de Partículas, Búsqueda Tabú, Búsqueda de Cuco, Búsqueda de vecindario variable, entre otros.
Para este trabajo se utilizará el algoritmo Genético (AG) y la búsqueda de vecindario variable (VNS).

4.8. Algoritmo Evolutivo

Según A.E. Eiben [8] pag.25 un algoritmo evolutivo es aquel que tiene como idea principal la de una población inicial de individuos sobre un entorno con recursos limitados, y la competencia por dichos recursos causa la **Selección Natural** (Sobreviven los individuos más aptos, teoría propuesta por Charles Darwin 1850). Esto conlleva a un aumento de aptitud de la población (En este trabajo se le llamará fitness).
Considerando el enfoque mencionado, un algoritmo evolutivo debe tener una función calidad a maximizar, luego se puede crear aleatoriamente una población inicial con soluciones candidatas, es decir, elementos del dominio de la función. Posteriormente se aplica la función calidad a estos, de forma que: Cuanto mayor sea el valor, tendrá mayor aptitud. En base de estos valores de aptitud, se eligen algunos mejores candidatos para pasar a una siguiente generación, esto se realiza mediante la recombinación y/o mutación. Estos son operadores que ayudan al algoritmo a tener variedad en las soluciones. La recombinación es un operador que es aplicado a dos o más individuos (padres),

obteniendo uno o más nuevos individuos (hijos). La mutación es un operador que es aplicado a un individuo, obteniendo otro nuevo individuo. A los nuevos individuos se le suele conocer como la descendencia o los descendientes, estos tienen una aptitud alta, y en función de su aptitud (y a veces edad), compiten por formar parte de una siguiente generación. Este proceso debe ser iterativo y debe parar cuando se encontró la solución deseada, o cuando se cumplió un límite de iteraciones.

4.9. Algoritmo genético

El algoritmo genético (AG) es el tipo de algoritmo evolutivo más conocido. John H. Holland fue el primero en dar esta idea mediante su libro Adaptation in Natural and Artificial Systems [15] en 1975. Según A.E. Eiben pag.99 [8]: Con ayuda del libro de Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning [11] y la tesis de De Jong [4], se logró lo que ahora se considera algoritmo genético o AG, o también llamado algoritmo canónico o algoritmo simple.
Un esquema general de un algoritmo genético es dado en Algoritmo 1.

Algoritmo 1: Algoritmo Genético

```
AlgoritmoGenetico
    Inicialización de Población Inicial con individuos aleatorios;
    Evaluar cada candidato con la función de aptitud;
    mientras Criterio de Parada hacer
        Seleccionar padres;
        Recombinar pares de padres;
        Mutar la descendencia obtenida;
        Evaluar los nuevos individuos con la función aptitud;
        Seleccionar los individuos para la siguiente generación;
    fin
```

4.10. Búsqueda de vecindad variable (VNS)

Es un algoritmo metaheurístico propuesto por Mladenovic & Hansen [17] 1977. La idea básica de este algoritmo es cambiar de vecindad cuando se encuentre un óptimo local. Es decir, una forma de salir de un óptimo local alcanzado al realizar una búsqueda, es cambiando el sistema de vecinos. Se comienza con una sucesión de sistemas de vecinos, y cada vez que se encuentre un óptimo local, se pasa al siguiente sistema de vecinos. Existen muchas variantes y extensiones de este algoritmo, un pseudocódigo que presenta la idea general de este se puede observar en Algoritmo 2.

Algoritmo 2: Búsqueda de vecindad variable

```
VNS
    Conjunto inicial de sistemas de vecinos V_k, k = 1, ..., k_max;
    Elegir una solución inicial x_0;
    mientras Se encuentre mejora hacer
        k = k + 1;
        mientras k es diferente de k_max hacer
            Encontrar solución óptima x en V_k;
            si x es mejor que x_0 entonces
                x_0 = x;
                k = 1;
            fin
            en otro caso
                k = k + 1
            fin
        fin
    fin
    devolver x_0
```

4.11. Ajuste de parámetros

El ajuste de parámetros se realiza para escoger qué parámetros son los adecuados de acuerdo a lo que buscamos en el algoritmo genético. Este se realizan antes de la ejecución del algoritmo, es decir para fijar parámetros con los que funcionará nuestro algoritmo. Existen varios enfoques, por ejemplo se puede buscar que el algoritmo sea rápido y genere soluciones deseables en muy

poco tiempo. O también se puede buscar presición en la solución, e intentar buscar buenas soluciones sin importar el tiempo. Para este trabajo se optará por buscar buenas soluciones para el problema del camino mínimo para grafos de gran cantidad de nodos.

4.11.1. Desempeño Online y Offline

Son tipos de desempeños con los cuales podemos medir el impacto de los parámetros de un AG. En la tesis de De Jong (1975) [4] se describen por primera vez, siendo estos:

- **Desempeño Online:** Se define como el promedio de fitness de todos los individuos de las últimas q generaciones.
- **Desempeño Offline:** Se define como el promedio de los fitness de todos los mejores individuos de las últimas q generaciones (de cada generación solo se considera el mejor).

En este trabajo se utilizará principalmente el desempeño online porque se quiere obtener una visión general de cómo evoluciona el algoritmo. Además esto servirá para obtener parámetros buenos en el caso se quiera escalar el algoritmo a un grafo con mayor número de nodos.

4.11.2. MBF

El MBF o Mean Best Fitness sirve para medir el desempeño o "performance"de cualquier problema en forma de AG. Para definirlo, en cada iteración del AG se guarda al individuo con mejor fitness. Entonces el MBF es el promedio de todos estos valores de todas las iteraciones. En este trabajo se utilizará para medir la función utilidad que será de ayuda para elegir un conjunto de parámetros adecuado.

Capítulo 5

Diseño e implementación

5.1. Pseudocódigo Dijkstra

Se utilizará el algoritmo de Dijkstra P2P (Point to point) como se muestra en el Algoritmo 3. Donde *Q* es el conjunto de vértices que fueron visitados, pero no expandidos, en la literatura se conoce a este conjunto como OPEN. El arreglo *dist* es el conjunto de pesos acumulados, o también se puede entender como distancias acumuladas. Y finalmente el arreglo *prev* es aquel donde se guardará el vértice previo en el camino mínimo para cada vértice, por ejemplo: prev[n] será el vértice previo de n en el camino mínimo.

5.2. Pseudocódigo A*

Se utilizará A* como se muestra en el Algoritmo 4. Se puede notar que el algoritmo es bastante similar al de Dijkstra, sin embargo, hay detalles a tener en cuenta como por ejemplo: El conjunto Q (conjunto de vértices que fueron visitados, pero no expandidos) no se llena al principio del algoritmo como lo hace Dijkstra, sino se va llenando en cada iteración. Esto quiere decir que Dijkstra visitará todos los vértices adyacentes al vértice inicial hasta llegar al vértice final, en cambio A* solo visitará algunos de esos vértices, esto dependerá de la heurística escogida.

Algoritmo 3: Algoritmo de Dijkstra P2P

Dijkstra $(Grafo, vInicio, vFin)$
Entradas: Grafo conformado por vértices y aristas, vértice inicial, vértice final
Salidas : Distancia mínima desde el vértice inicial al vértice final, camino recorrido
para cada *vertice* $v \in$ *Grafo.Vertices* **hacer**
dist[v] ← INFINITO;
prev[v] ← NO DENIFIDO;
agregar v a Q;
fin
$dist[vInicio] \leftarrow 0$;
mientras Q *no está vacío* **hacer**
$u \leftarrow$ vértice en Q con el menor valor dist[u];
si *u = vFin* **entonces**
camino ← arreglo vacio;
si *prev[u]* $\neq$ *NO DEFINIDO o u = vFin* **entonces**
mientras *prev[u]* $\neq$ *NO DEFINIDO* **hacer**
agregar u a camino;
u ← prev[u];
fin
devolver $dist[u], camino[]$;
fin
fin
Eliminar u de Q;
para cada *vértice vecino* v *de* $u \in Q$ **hacer**
$temp \leftarrow dist[u] + Grafo.Aristas(u, v)$;
si $temp < dist[v]$ *y* $dist[u] \neq INFINITO$ **entonces**
$dist[v] \leftarrow temp$;
$prev[v] \leftarrow u$;
fin
fin
fin
devolver *ERROR*;

Algoritmo 4: Algoritmo A*

AStar $(Grafo, vInicio, vFin, h)$
Entradas: Grafo conformado por vértices y aristas, vértice inicial, vértice final, función heurística
Salidas : Distancia mínima desde el vértice inicial al vértice final, camino recorrido
para cada *vertice* $v \in$ *Grafo.Vertices* **hacer**
distG[v] ← INFINITO;
distF[v] ← INFINITO;
fin
agregar $vInicio$ a Q;
prev ← arreglo vacío;
$distG[vInicio] \leftarrow 0$;
$distF[vInicio] \leftarrow h(vInicio)$;
mientras Q *no está vacío* **hacer**
$u \leftarrow$ vértice en Q con el menor valor distF[u];
si u = *vFin* **entonces**
camino ← arreglo vacio;
si *prev[u]* $\neq$ *NO DEFINIDO o* u = *vFin* **entonces**
mientras $u \neq$ *NO DEFINIDO* **hacer**
agregar u a camino;
u ← prev[u];
fin
devolver $distG[u], camino[]$;
fin
fin
Eliminar u de Q;
para cada *vértice vecino* v *de* $u \in Q$ **hacer**
$temporalG \leftarrow distG[u] + Grafo.Aristas(u, v)$;
si $temporalG < distG[v]$ *y* $distG[u] \neq INFINITO$ **entonces**
$prev[v] \leftarrow u$;
$distG[v] \leftarrow temporalG$;
$distF[v] \leftarrow temporalG + h(v)$;
si $v \notin Q$ **entonces**
agregar v a Q;
fin
fin
fin
fin
devolver *ERROR;*

5.3. Algoritmo evolutivo

Se implementará un algoritmo evolutivo, donde el objetivo es obtener el camino mínimo entre los vértices s y t de un grafo $G(V, E)$ donde $s, t \in V$. Para ello se describirán los pasos mencionados en el Algoritmo 1 del capítulo anterior detalladamente:

5.3.1. Individuo

Se considerará a un individuo como un conjunto ordenado de vértices o nodos, de forma que sea una solución factible que comience en el nodo s y termine en el nodo t sin formar ciclos, escogidos de forma aleatoria. En este trabajo el fenotipo y genotipo son lo mismo, es decir un conjunto de nodos de s a t. Por ejemplo en la Figura 5 se muestra un grafo G con 22 nodos, y un individuo desde el nodo $s = 1$ hasta el nodo $s = 7$ sería: $\{1, 5, 4, 7\}$.

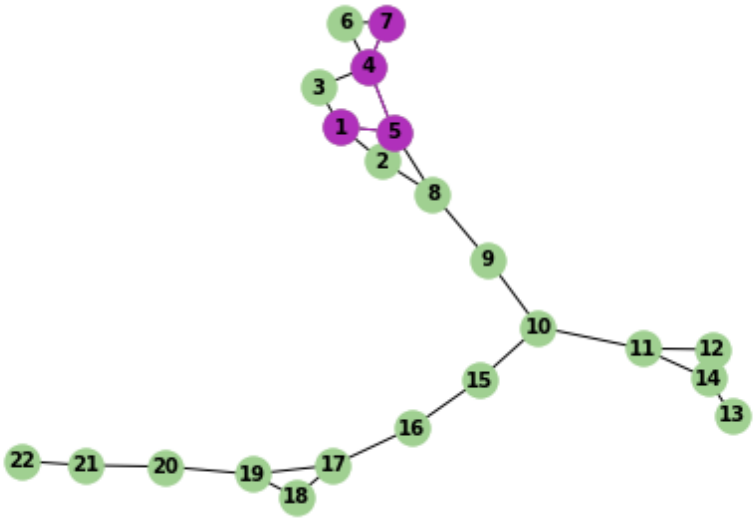


FIGURA 5.1: Ejemplo de individuo con $s = 1$ y $t = 7$. Elaboración propia

5.3.2. Población inicial

Será el conjunto de individuos con el que se empezará el algoritmo. Sus longitudes son variables, lo importante es que sean soluciones factibles, es

decir que exista dicho camino de s a t sin formar ciclos. Cada individuo se genera utilizando el Algoritmo 5, donde al inicio todos los nodos están como no visitados, y se realiza una búsqueda visitando vecinos de forma aleatoria evitando formar ciclos. Y en caso nos quedemos atrapados sin tener vecinos no visitados, volvemos al nodo anterior. Este proceso se realiza de forma iterativa hasta llegar al nodo final.

Algoritmo 5: Generar un individuo

```
Individuo (s, t, G)
    Entradas: Nodo inicial, nodo final, grafo G
    Salidas  : individuo
    camino[] <- s;
    actual = s;
    actual.visitado = True;
    mientras actual ≠ t hacer
        nodoSig = random(vecinos del nodo actual);
        si esBucle(nodoSig) y nodoSig ≠ t entonces
            nodoSig.visitado = True;
        fin
        si nodoSig.visitado = False entonces
            si actual no está en camino entonces
                camino[] <- actual;
            fin
            camino[] <- nodoSig ;
            nodoSig.visitado = True;
            actual = nodo;
        fin
        si esBucle(actual) y actual ≠ t entonces
            actual.visitado = True;
            actual = camino.pop()
        fin
    fin
    devolver camino
```

5.3.3. Función fitness

También llamada función aptitud, será definida como la suma de todas las aristas de un individuo. Será nuestra función a minimizar.

$$Minimizar \sum_{e \in U} \mathrm{e}$$

donde $U \subset G$ es un camino de s a t.

5.3.4. Función Crossover

La función Crossover o de Cruzamiento que se utilizará es una variante de la mencionada en el artículo [6], en la cual se explica que se comienza con dos individuos que serán los padres. Luego se buscará el primer nodo en común, que no sea el inicio ni el final, y este será el punto de corte (puede que dicho nodo esté en distintas posiciones en cada individuo), desde ese punto en adelante se realizará un intercambio entre los dos individuos, generando dos nuevos hijos. La modificación hecha al algoritmo del artículo es que para este trabajo el punto de corte se escoge de forma aleatoria del conjunto de puntos de cortes posibles, esto con la finalidad de evitar quedarnos en óptimos locales. Lo bueno de realizar el cruzamiento de esta manera es que siempre los nuevos hijos serán soluciones factibles, y ya no se necesitará gastar recursos en arreglar soluciones. Un ejemplo se puede observar en la Figura 5.2.

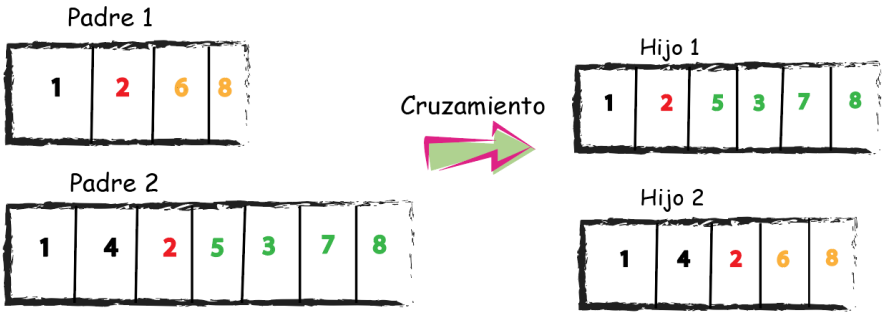


FIGURA 5.2: Ejemplo de la operación de cruzamiento. Elaboración propia

5.3.5. Función de Mutación

Para la función de mutación se utilizará la metaheurístisca VNS aplicada a un individuo, esto con el fin de mejorarlo generando un individuo con menor o igual fitness. La idea del uso de esta metaheurística para el problema del camino mínimo se presenta en el artículo [6] (2015). Para entenderlo se mostrará un ejemplo utilizando el grafo de la Figura 5.3 como referencia.

Para aplicar VNS se construye dos estructuras de vecinos de forma que

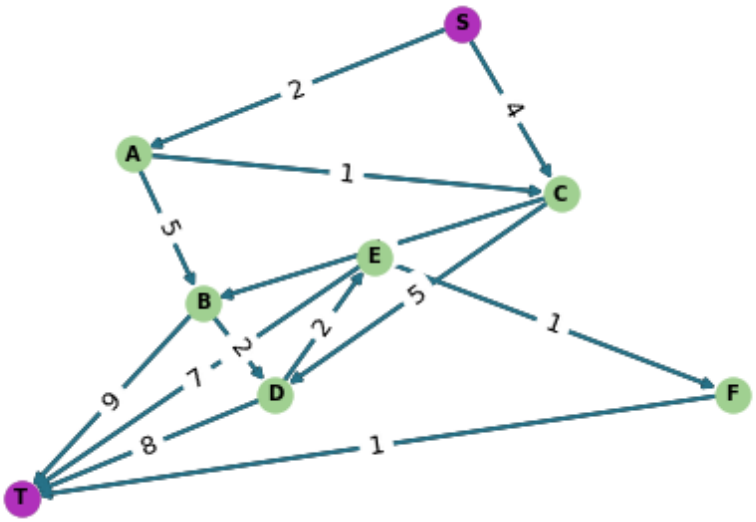


FIGURA 5.3: Grafo dirigido con pesos. Elaboración propia

la primera estructura reemplaza una arista por dos aristas, y la segunda reemplaza dos aristas por una arista. Los reemplazos tienen que ser mejor que el camino original, es decir menor que lo reemplazado. Utilizando el grafo de la Figura 5.3 se obtiene:

Camino	Reemplazos
[S,C]	[S,A,C]
[A,B]	[A,C,B]
[C,D]	[C,B,D]
[E,T]	[E,F,T]

CUADRO 5.1: Estructura1 VNS

Camino	Reemplazos
[B,D,T]	[B,T]
[D,E,T]	[D,T]

CUADRO 5.2: Estructura2 VNS

La creación de estas dos estructuras es un trabajo de preprocesamiento, es decir que se realiza cuando se genera el grafo, por lo que el tiempo demorado

no se incluye en la evaluación de los algoritmos.
Una vez obtenido estas dos estructuras, empezamos a utilizar el algoritmo VNS de la forma vista en Algoritmo 2. La idea es evaluar si existe algún reemplazo para el individuo en la estructura 1, hasta que ya no exista ningún reemplazo posible, luego realizar lo mismo en la estructura 2. Tomando como ejemplo al individuo [S,C,B,D,T], cuyo fitness es $4 + 1 + 2 + 8 = 15$. Primero se evalúa en la estructura 1 si existe algún reemplazo, y se encuentra que [S,C] se puede reemplazar por [S,A,C], dando como nuevo resultado [S,A,C,B,D,T], ahora con este nuevo individuo se vuelve a evaluar en la estructura 1, y se observa que ya no existe reemplazos. Entonces se empieza a evaluar con la estructura 2, y se encuentra que [B,D,T] se puede reemplazar por [B,T], dando el individuo [S,A,C,B,T], volviendo a evaluar sobre la estructura 2 notamos que ya no existen reemplazos disponibles, por lo que el resultado será [S,A,C,B,T] con un fitness de $2 + 1 + 1 + 9 = 13$.

5.3.6. Funciones de selección

Se utilizarán dos funciones de selección: Una para elegir los padres a realizar el cruzamiento y mutación, y la otra para pasar a la siguiente generación. Existen varias funciones de selección conocidas, entre ellas tenemos:

- **Selección proporcional al fitness:** Es una función de selección la cual asigna una probabilidad a cada individuo de acuerdo al fitness. En este trabajo se utilizará para seleccionar dos padres que posteriormente realizarán el cruzamiento y mutación. Cada individuo tiene la siguiente probabilidad de ser escogido:

$$\frac{f_i}{\sum_{i=1}^{n} f_i}$$

- **Selección elitista:** Esta función selecciona a cierta cantidad de los individuos con mejor fitness. Se suele utilizar este método para trasladar cierta cantidad de individuos a la siguiente generación. En este trabajo la

cantidad será el tamaño de la población inicial, es decir el tamaño de la pobación será siempre constante.

- **Selección Método de la Ruleta:** Es una función que selecciona cierta cantidad de individuos mediante una ruleta que está definida de acuerdo a los fitness de los individuos. Los de mayor fitness tendrán mayor probabilidad a ser seleccionados, sin embargo todos los individuos tienen oportunidad de ser escogidos, en la selección elitista esto no sucede.

5.4. Grafos de entrada

Herramientas utilizadas:

- **OpenStreetMap**: Para la recolección de datos se utilizó OpenStreetMap [23], la cual es una herramienta de datos abiertos que proporciona distintos metadatos de mapas para distintos propósitos.
- **Osm4routing**: Para exportar el mapa de interés se utilizó el parseador osm4routing [3], que proporciona el mapa en un formato apropiado para utilizarlo en algoritmos basados en grafos.

Utilizando estas herramientas se obtienen distintos atributos de las intersecciones y las calles como vértices y aristas respectívamente. Algo importante que mencionar es que al trabajar sobre carreteras o calles, la única condición que cumplirá el grafo es: $\forall e \in E, e > 0$, donde e es una arista del grafo y E es el conjunto de aristas. Para este trabajo nos interesan los siguientes atributos:

- **ID:** Entero de 11 cifras. Identificador único de un vértice.
- **Source:** ID tipo entero de 11 cifras. Identificador único que nos proporciona el vértice desde el cual comienza la arista.
- **Target:** ID tipo entero de 11 cifras. Identificador único que nos proporciona el vértice al cual llega la arista.

	Grafo1	Grafo2	Grafo3	Grafo4	Grafo5
N° vertices	722	2758	8814	80	10100
N° aristas	1052	2758	13291	109	16175

CUADRO 5.3: Descripción de los grafos de entrada

- **Length:** Número real. Proporciona la distancia de la arista en metros.
- **Lon:** Número real. Proporciona la coordenada de longitud en grados decimales.
- **Lat:** Número real. Proporciona la coordenada de latitud en grados decimales.

Para poder visualizar el grafo generado se utilizó la librería de Python NetworkX [2]. Un ejemplo de un grafo de entrada generado con OpenStreetMap, Osm4routing y NetworkX se puede observar en la figura 5.4. Se utilizaron cinco grafos donde existe una diferencia notable del número de

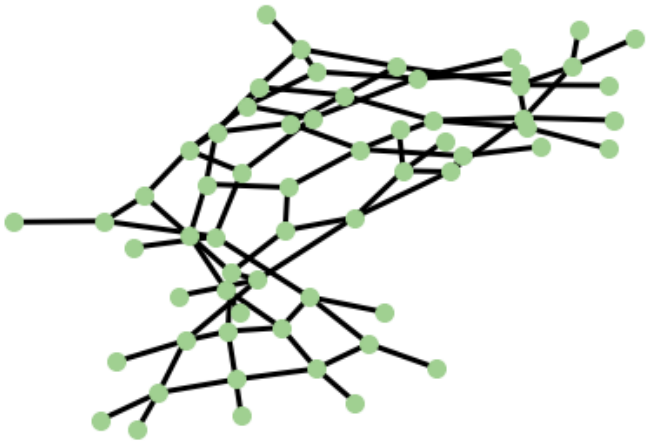


FIGURA 5.4: Ejemplo de grafo de entrada.

vértices y aristas como se puede observar en el Cuadro 5.3, de cada uno de ellos se escogerán cinco pares de vértices (inicial,final) de forma aleatoria y posteriormente se ejecutarán Dijkstra y A* sobre ellos. Todo esto con el objetivo de poder comparar estos algoritmos en grafos con distintos tamaños.

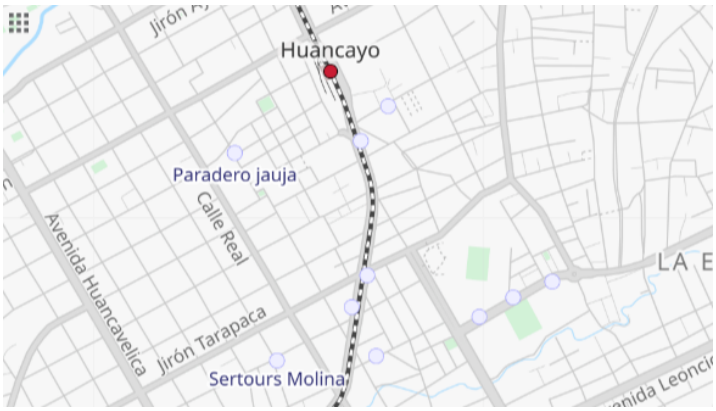


FIGURA 5.5: Mapa del grafo1.

Los mapas de donde se obtuvieron los grafos se pueden observar en las Figuras 5.5, 5.6, 5.7, 5.8 y 5.9.

5.5. Métricas

Teniendo en cuenta que se está trabajando sobre un problema de grafos y considerando [13], [12] y [21], para comparar los algoritmos se utilizarán las siguientes métricas:

- Tiempo: El tiempo de ejecución es importante porque complementa a otras métricas y ayuda tener un mejor entendimiento del rendimiento del algoritmo.
- Eficiencia: Para este trabajo se definirá como:

$$Eficiencia = \frac{Z}{Z_0} * 100\,\%$$

Donde:

Z : Suma de aristas del camino mínimo.

Z_0 : Suma de aristas del camino mínimo obtenido.

FIGURA 5.6: Mapa del grafo2.

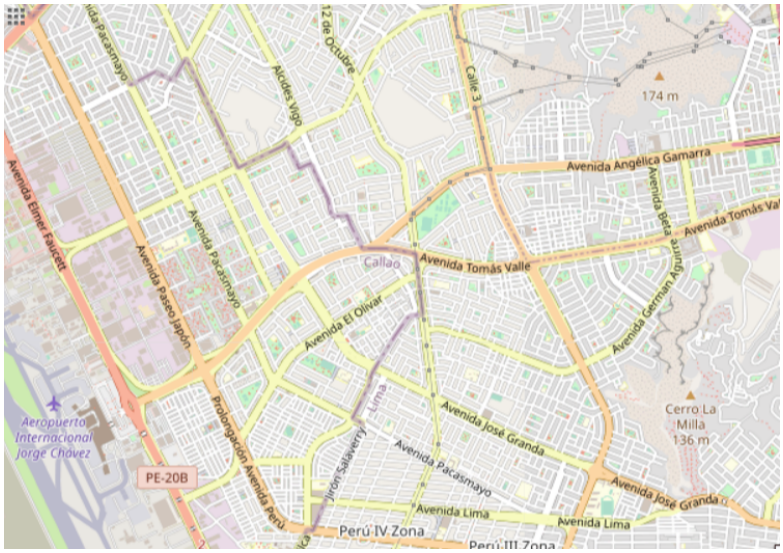


FIGURA 5.7: Mapa del grafo3.

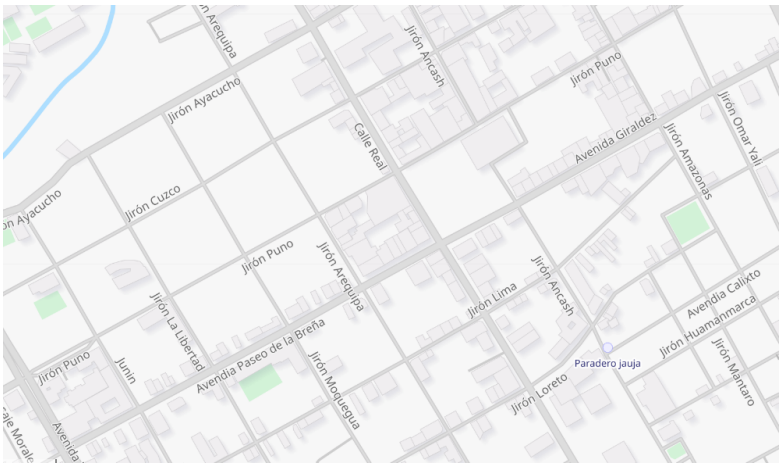


FIGURA 5.8: Mapa del grafo4.

FIGURA 5.9: Mapa del grafo5.

5.6. Implementación

La implementación de los Algoritmos de Dijkstra, A* y AG se realizó en Python. Para utilizar los grafos de entrada generados con osm4routing se utilizó dos dataframes de Pandas: Uno para los vértices, y otro para las aristas.

Teniendo en consideración los datos de entrada de 3 (Dijkstra), 4 (A*) y AG, se implementó una clase Grafo, de la cual, los métodos más importantes son:

- **Agregar vértice:** Agrega un vértice
- **Obtener vértice:** Se obtiene un vértice gracias a su ID correspondiente.
- **Agregar arista:** Se agrega la arista, y también el vecino del vértice donde inicia la arista.
- **VNS Estructuras:** Crea las estructuras para utilizar el algoritmo metaheurístico VNS. Solo se puede llamar si el grafo ya tiene todos los vértices y aristas puestos.

Cada vértice del grafo deberá tener también sus propios atributos y métodos, por lo que también se implementó una clase vértice, de la cual, los métodos más importantes son:

- **Obtener peso:** Se obtiene el peso del vértice actual con algún vértice adyacente.
- **Obtener distancia:** Se obtiene la distancia del camino desde el vértice inicial al vértice actual.
- **Obtener distanciaG:** Se obtiene la distancia G desde el vértice inicial al vértice actual. Este método es importante para el algoritmo A*.
- **Obtener distanciaF:** Se obtiene la distancia F, definida como la suma de la distancia G con la heurísitica evaluada en el vértice actual, del camino desde el vértice inicial al vértice actual. Este método es importante para el algoritmo A*.

- **Agregar previo:** Se guarda el vértice previo referente al vértice actual.
- **Obtener previo:** Se obtiene el vértice previo guardado.

Finalmente se implementó una clase Estructura, la cual será utilizada por el grafo en el método VNS Estructuras. Los métodos más importantes de esta clase son:

- **Obtener camino:** Devuelve el arreglo de caminos, es decir aquellos antes del reemplazo.
- **Obtener reemplazo:** Devuelve el arreglo de reemplazos.

5.6.1. Dijkstra

La implementación del algoritmo 3 (Dijkstra) se realizó utilizando colas de prioridad, ya que esto reduce la complejidad temporal. En Python un análogo a este tipo de estructura de datos es heapq [1], el cual es un montículo binario que provee métodos útiles como heappop y heappush:

- **heappop:** Retorna y elimina el menor elemento del montículo, tiene complejidad temporal $O(log(n))$.
- **heappush:** Añade un elemento al montículo. Tiene complejidad temporal $O(log(n))$

5.6.2. A*

La implementación del algoritmo 4 (A*) al igual que Dijkstra también se realizó utilizando un montículo binario. Algunos detalles importantes de la implementación son:

- **Heurística:** La heurística utilizada es la distancia euclídea en metros, aplicada a las coordenadas longitud y latitud. Se define como:

$$h(x_1, y_1) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} * radioTierra * (\pi/180) * 1000$$

Donde:

- **(x1,y1):** Variables que representan la longitud y la latitud del vértice actual respectivamente.
- **(x2,y2):** Constantes que representan la longitud y latitud del vértice final respectivamente.
- **radioTierra:** Constante que representa el radio de la tierra, en este trabajo se considerará como 6371.

- **Función g(n):** La función g utilizada es la distancia del camino desde el vértice inicial hasta el vértice actual n.
- **Función f(n):** La función f utilizada es la suma de g y h.

$$f(n) = g(x, y) + h(n)$$

Donde:

- (x, y) es la tupla (longitud, latitud) del vértice n.

5.6.3. AG

La implementación del Algoritmo Genético se realizó sobre la estructura Grafo. Una ventaja de trabajar sobre esta estructura es que se puede realizar operaciones de preprocesamiento al inicializar el grafo, esto mediante el método VNS Estructuras. Los parámetros que necesita este algoritmo son:

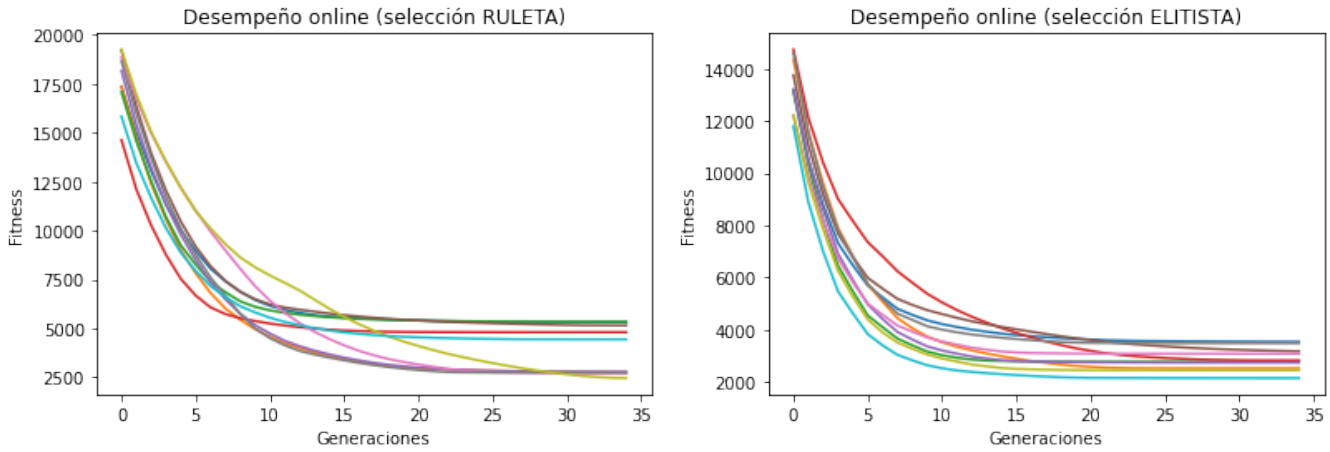
- **N** : Número de la población inicial.
- **numPadres** : número de padres que participarán en el cruzamiento, debe ser par, ya que se escogerán 2 padres por cada llamada a la función cruzamiento.
- **CROSSOVER RATE** : número de 0 a 1, que indica la probabilidad de que se produzca cruzamiento.

- **MUTATION RATE** : número de 0 a 1, que indica la probabilidad de que se produzca mutación.
- **GENERACIONES** : Número de iteraciones que tendrá el algoritmo genético.
- **s** : Vértice o nodo inicial.
- **t** : Vértice o nodo final.

5.6.4. Ajuste de parámetros del AG

Se realizó el ajuste de parámetros del AG sobre el mapa3 en dos etapas:

- Primero se escogió una función selección entre RULETA y ELITISTA. Para esto se realizaron dos algoritmos: El primero utilizando como función de selección a la siguiente generación el método de la ruleta, y el otro con el método elitista. El resto de parámetros son fijos y fueron escogidos a criterio propio. Se evaluó su desempeño online en cada cinco iteraciones, para 10 ejecuciones del algoritmo.



(A) Método de la ruleta (B) Método elitista

FIGURA 5.10: Algoritmos con distintos métodos de selección

Se elige el método de la ruleta porque es el que tiene cambios contínuos, aunque no llegue a una solución rápidamente como lo hace el elitista. La

gráfica del método elitista señala que se llega a una solución en muy pocas iteraciones, pero es muy probable que se quede atrapado en óptimos locales.

- Una vez escogida la función de selección, se procedió a escoger los mejores parámetros. Para esto se tuvo en cuenta la obtención de buenas soluciones y no tanto el tiempo de ejecución, con el objetivo de que el algoritmo funcione bien al escalarlo a grafos más grandes. Los parámetros fijos fueron: s, t y GENERACIONES. Los parámetros a ajustar fueron: MUTATION RATE, CROSSOVER RATE, N y numPadres. El rango para estos valores fueron:

  - **MUTATION RATE**: 0.3 - 0.9
  - **CROSSOVER RATE**: 0.5 - 0.9
  - **N**: 50 - 300
  - **numPadres**: del 20 % al 80 % de la población inicial

  Se escogió 6 conjuntos de parámetros en los rangos mecionados de forma aleatoria. Y cada conjunto de parámetros se ejecutó 8 veces. En la figura 5.11 se observa los desempeños online de las 8 ejecuciones por cada conjunto de parámetros. Posteriormente se realizó la función utilidad tomando como medida el promedio de MBF (Mean best fitness) de las 8 ejecuciones por cada conjunto de parámetros, obteniendo la Figura 5.12. De la cual se obtiene que el conjunto de parámetros que tiene el menor promedio MBF es el 4, además corroborando con la Figura 5.11 se nota que el conjunto 4 llega a fitness bastante bajos evitando quedarse en mínimos locales, lo cual indica que el algoritmo es bueno. Por estas razones se utilizará para este trabajo el conjunto de parámetros 4 de la tabla 5.4.

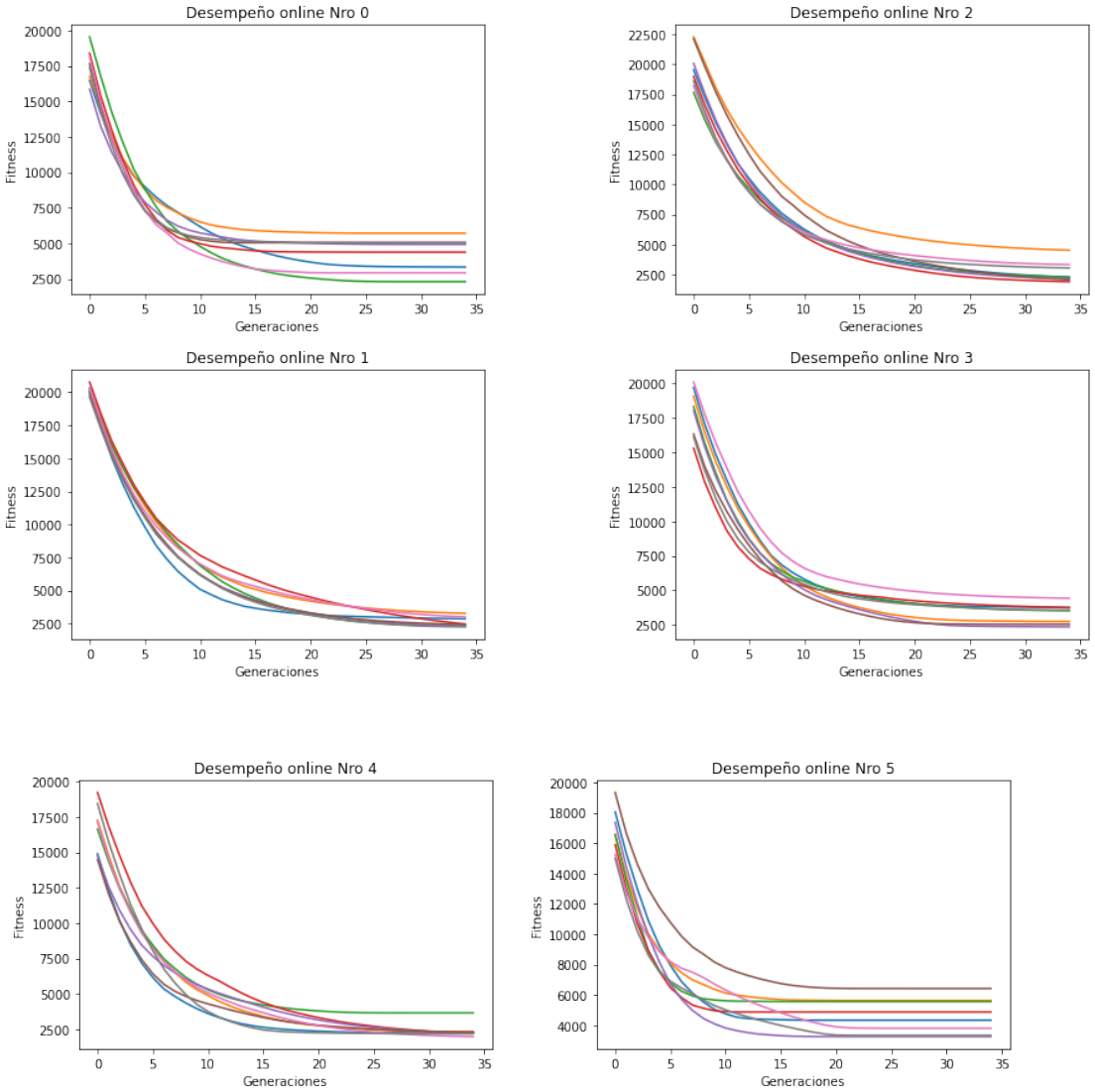


FIGURA 5.11: Desempeños online para cada conjunto de parámetros.

	Cjto 0	Cjto 1	Cjto 2	Cjto 3	Cjto 4	Cjto 5
MUTATION RATE	0.372217	0.74840	0.62607	0.55863	0.56888	0.32892
CROSSOVER RATE	0.52004	0.73470	0.70708	0.84853	0.56153	0.56562
N	94	220	260	141	144	61
numPadres	52	217	252	104	114	31

CUADRO 5.4: Conjuntos de parámetros para el Algoritmo Evolutivo

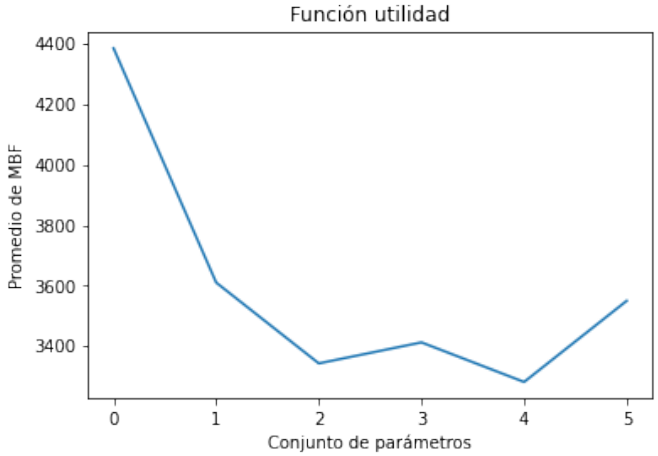


FIGURA 5.12: Función de utilidad.

Capítulo 6

Resultados

En esta sección se presentarán los resultados obtenidos de la implementación.
Se ejecutaron los algoritmos Dijkstra, A* y AG sobre los grafos descritos en la tabla 5.3, los resultados se ordenan de manera que cada grafo corresponde a dos tablas, una de iteraciones, donde se presentan las características que los algoritmos comparten: Vértice inicial, vértice final, longitud del camino mínimo y distancia mínima; y otra de resultados, donde se evalúan las métricas mencionadas en el capítulo 5: Tiempo y eficiencia para los algoritmos. Finalmente se presentan las gráficas de cómo se desenvuelven estos algoritmos en tiempo y eficiencia con distintas longitudes de camino mínimo.

6.1. Observaciones

- De las tablas presentadas y de la Figura 6.2 se observa que para caminos con menos de 20 aristas aproximadamente, el AG obtiene la respuesta correcta, es decir 100 % de eficiencia, luego comienzan a aparecer picos.
- En la Figura 6.1 se observa que para un camino mínimo donde la longitud llegó a 113 aristas, el AG tiene menos tiempo de ejecución que Dijkstra y A*, pero observando la Figura 6.2 se nota que la eficiencia está por debajo del 50 %. Este resultado se puede ver en la Tabla 6.10 en la fila N° 1.

6.2. Tablas

N°	V. inicial	V. final	Long camino	Distancia mínima
1	1209259836	1209249054	15	878.1235
2	5415433275	4150231329	50	3340.6989
3	5415363580	1052552702	37	2288.7804

CUADRO 6.1: Iteraciones Grafo1

	Tiempo (s.)			Eficiencia (%)	
N°	Dijkstra	A*	AG	A*	AG
1	0.0897	0.0289	1.3753	100.0	100.0
2	0.1545	0.0489	1.6864	100.0	99.5207
3	0.2014	0.0558	1.8526	100.0	84.3673

CUADRO 6.2: Resultados Grafo1

N°	V. inicial	V. final	Long camino	Distancia mínima
1	9851999130	9291573217	38	1688.3567
2	827102887	3978784924	24	1434.5994
3	6129593671	6996695354	48	2384.8048

CUADRO 6.3: Iteraciones Grafo2

	Tiempo (s.)			Eficiencia (%)	
N°	Dijkstra	A*	AG	A*	AG
1	2.2172	0.3291	2.5751	100.0	90.5038
2	1.4665	0.2314	2.8553	100.0	99.9885
3	2.9940	0.3470	2.8802	100.0	92.1028

CUADRO 6.4: Resultados Grafo2

N°	V. inicial	V. final	Long camino	Distancia mínima
1	4261133139	4436916496	46	1752.4968
2	7061771522	6381144732	58	2993.2620
3	847505374	846599893	32	1405.3569

CUADRO 6.5: Iteraciones Grafo3

	Tiempo (s.)			Eficiencia (%)	
N°	Dijkstra	A*	AG	A*	AG
1	11.2714	0.7759	11.4842	100.0	49.0469
2	17.2481	2.1293	9.6839	100.0	91.4694
3	4.5910	0.5724	9.0734	100.0	98.8651

CUADRO 6.6: Resultados Grafo3

N°	V. inicial	V. final	Long camino	Distancia mínima
1	1057089657	1209250686	11	1421.1030
2	1209258101	1041615481	14	875.3940
3	5411247482	1052552184	16	1159.4745

CUADRO 6.7: Iteraciones Grafo4

	Tiempo (s.)			Eficiencia (%)	
N°	Dijkstra	A*	AG	A*	AG
1	0.0039	0.0039	0.9494	100.0	100.0
2	0.0019	0.0039	0.8696	100.0	100.0
3	0.0019	0.0029	0.8417	100.0	100.0

CUADRO 6.8: Resultados Grafo4

N°	V. inicial	V. final	Long camino	Distancia mínima
1	1273895792	1404104167	113	5995.4888
2	3756612344	1657677575	25	659.7808
3	1273732538	6722093685	54	2117.1310

CUADRO 6.9: Iteraciones Grafo5

	Tiempo (s.)			Eficiencia (%)	
N°	Dijkstra	A*	AG	A*	AG
1	60.5464	44.6936	33.1752	99.9936	43.0524
2	4.1728	0.4647	12.8217	100.0	47.3910
3	21.8537	1.6017	15.2063	100.0	65.2497

CUADRO 6.10: Resultados Grafo5

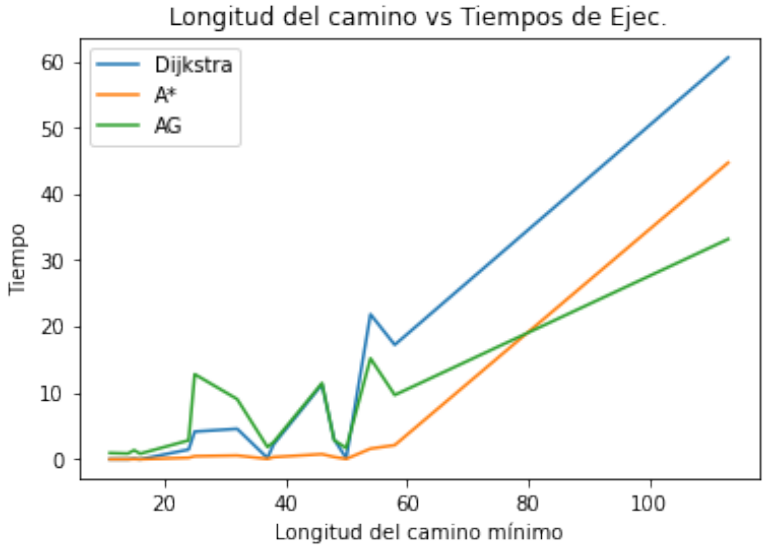


FIGURA 6.1: Longitud del camino mínimo vs Tiempos de ejecución.

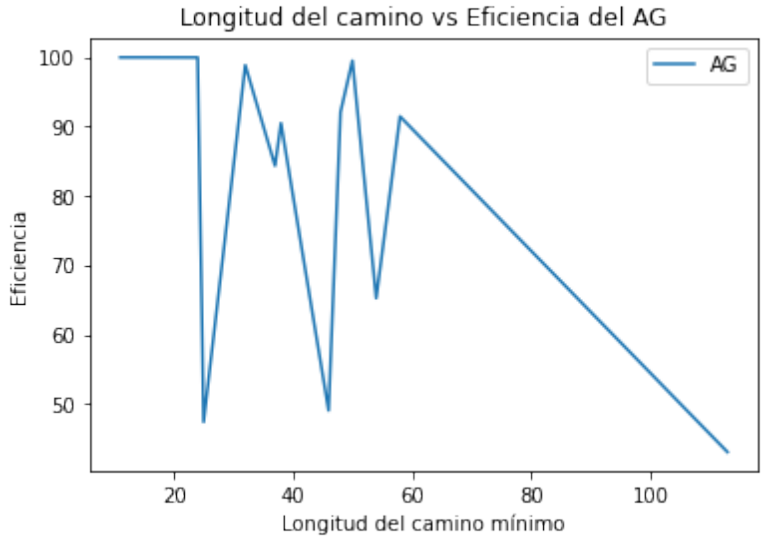


FIGURA 6.2: Longitud del camino mínimo vs Eficiencia del AG.

Capítulo 7

Conclusiones y Trabajo Futuro

En esta sección se presentarán las conclusiones de este trabajo y se describirá los posibles trabajos futuros.

7.1. Conclusiones

- Entre los tres métodos evaluados: Dijkstra, A* y AG. El mejor método es A*, ya que en la mayoría de casos obuvo mejor tiempo que el resto, y una eficiencia del 100 %, es decir la respuesta correcta. Pero en la fila N° 1 de la Tabla 6.10 se observa que el AG obtuvo el mejor tiempo, pero el problema es su eficiencia de 43 %, la cual es muy baja.

- Entre el AG y Dijkstra, este último obtiene mejores tiempos en longitudes de camino menores a 50 aproximandamente. Pero luego de eso, el AG empieza a obtener tiempos menores, aunque la eficiencia no siempre es la mejor.

- Si se fijan un par de vértices inicial y final sobre un grafo y sobre este se ejecutan A* y Dijkstra repetidas veces, la única métrica que cambia es el tiempo. La métrica eficiencia no cambia. En cambio en el AG en cada ejecución se obtienen distintos tiempos y distintas eficiencias (cuando el grafo es pequeño, aprox. menos de 100 vértices, AG siempre obtiene 100 % de eficiencia).

7.2. Trabajo Futuro

Se propone como trabajo futuro mejorar el algoritmo metaheurístico AG propuesto, encontrando una mejor forma de obtener la población inicial, sin realizar una búsqueda de forma aleatoria. También agregar a la comparación las complejidades temporales de los algoritmos, realizar la comparación con distintas heurísticas y asimismo comparar algoritmos de camino mínimo más actuales, como por ejemplo las variantes bidireccionales de A* y Dijkstra.

Bibliografía

[1] Heapq. https://docs.python.org/3/library/heapq.html.

[2] Networkx. https://networkx.org/.

[3] osm4routing. https://github.com/Tristramg/osm4routing.

[4] K. ALAN de JONG. Analysis of the behavior of a class of genetic adaptive systems. *Technical Report No. 185, Department of Computer and Communication Sciences, University of Michigan*, 1975.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2022.

[6] O. Dib, M.-A. Manier, and A. Caminada. A hybrid metaheuristic for routing in road networks. In *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, pages 765–770. IEEE, 2015.

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

[8] A. E. Eiben, J. E. Smith, et al. *Introduction to evolutionary computing*, volume 53. Springer, 2003.

[9] S. Even. *Graph algorithms*. Cambridge University Press, 2011.

[10] F. W. Glover and G. A. Kochenberger. *Handbook of metaheuristics*, volume 57. Springer Science & Business Media, 2006.

[11] D. E. Golberg. Genetic algorithms in search, optimization, and machine learning. *Addion wesley*, 1989(102):36, 1989.

[12] A. V. Goldberg. Point-to-point shortest path algorithms with preprocessing. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 88–102. Springer, 2007.

[13] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *SODA*, volume 5, pages 156–165. Citeseer, 2005.

[14] P. E. Hart, N. J. Nilsson, and B. Raphael. Correction to.[a] formal basis for the heuristic determination of minimum cost paths". *ACM SIGART Bulletin*, (37):28–29, 1972.

[15] J. H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[16] K. Magzhan and H. M. Jani. A review and evaluations of shortest path algorithms. *International journal of scientific & technology research*, 2(6): 99–104, 2013.

[17] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers & operations research*, 24(11):1097–1100, 1997.

[18] C. Nils J. Nilsson, Chang and R. Lee. Problem-solving methods in artificial intelligence (nils j. nilsson), 1972.

[19] N. J. Nilsson. *The quest for artificial intelligence*. Cambridge University Press, 2009.

[20] D. Rachmawati and L. Gustin. Analysis of dijkstra's algorithm and a* algorithm in shortest path problem. In *Journal of Physics: Conference Series*, volume 1566, page 012061. IOP Publishing, 2020.

[21] S. Sawlani. *Explaining the Performance of Bidirectional Dijkstra and A* on Road Networks*. PhD thesis, University of Denver, 2017.

[22] R. Sedgewick. Algorithms in c++–3rd edition, volume 2 (part 5), 2002.

[23] O. y colaboradores. Openstreetmap. https://www.openstreetmap.org/about.

Apéndice A

Repositorio con la implementación

Enlace al repositorio de github:
https://github.com/Franz04Rony/Proyecto-tesis3