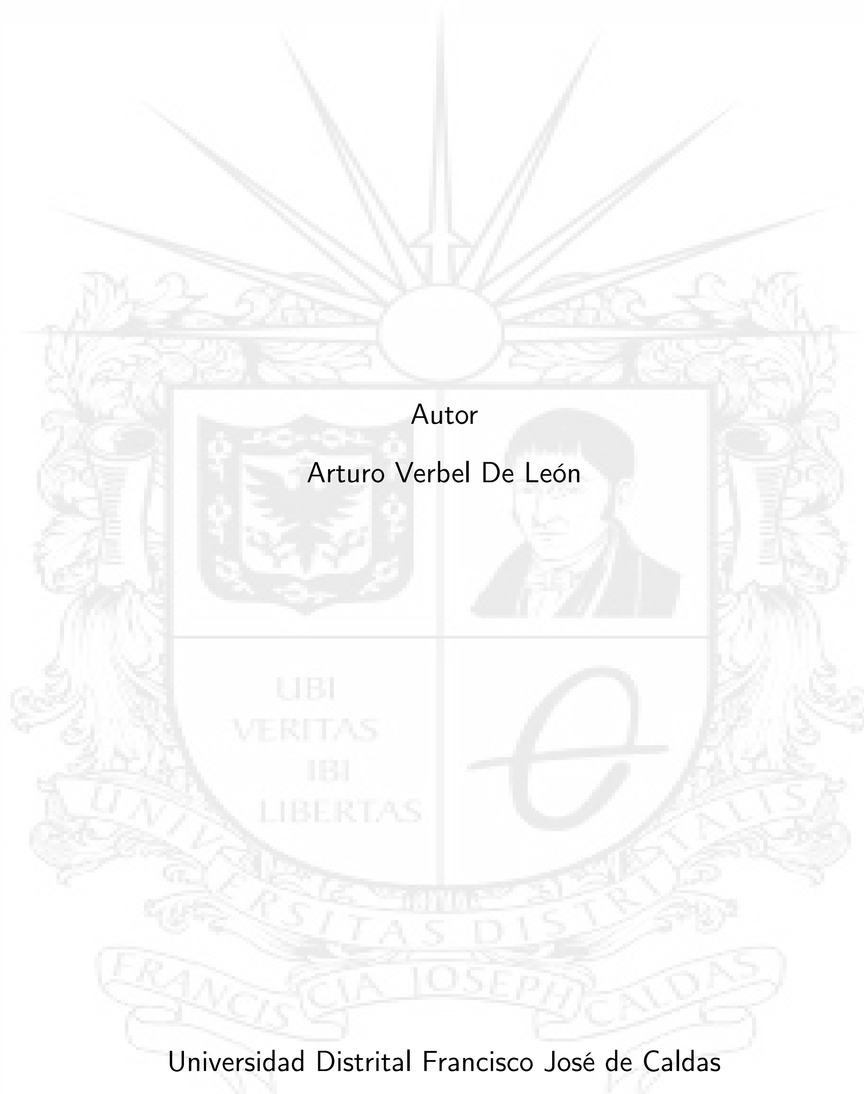
Actualización de Todos los Caminos más Cortos en Grafos Dinámicos con Inserción de Arcos



Autor

Arturo Verbel De León

Universidad Distrital Francisco José de Caldas

Facultad de Ingeniería

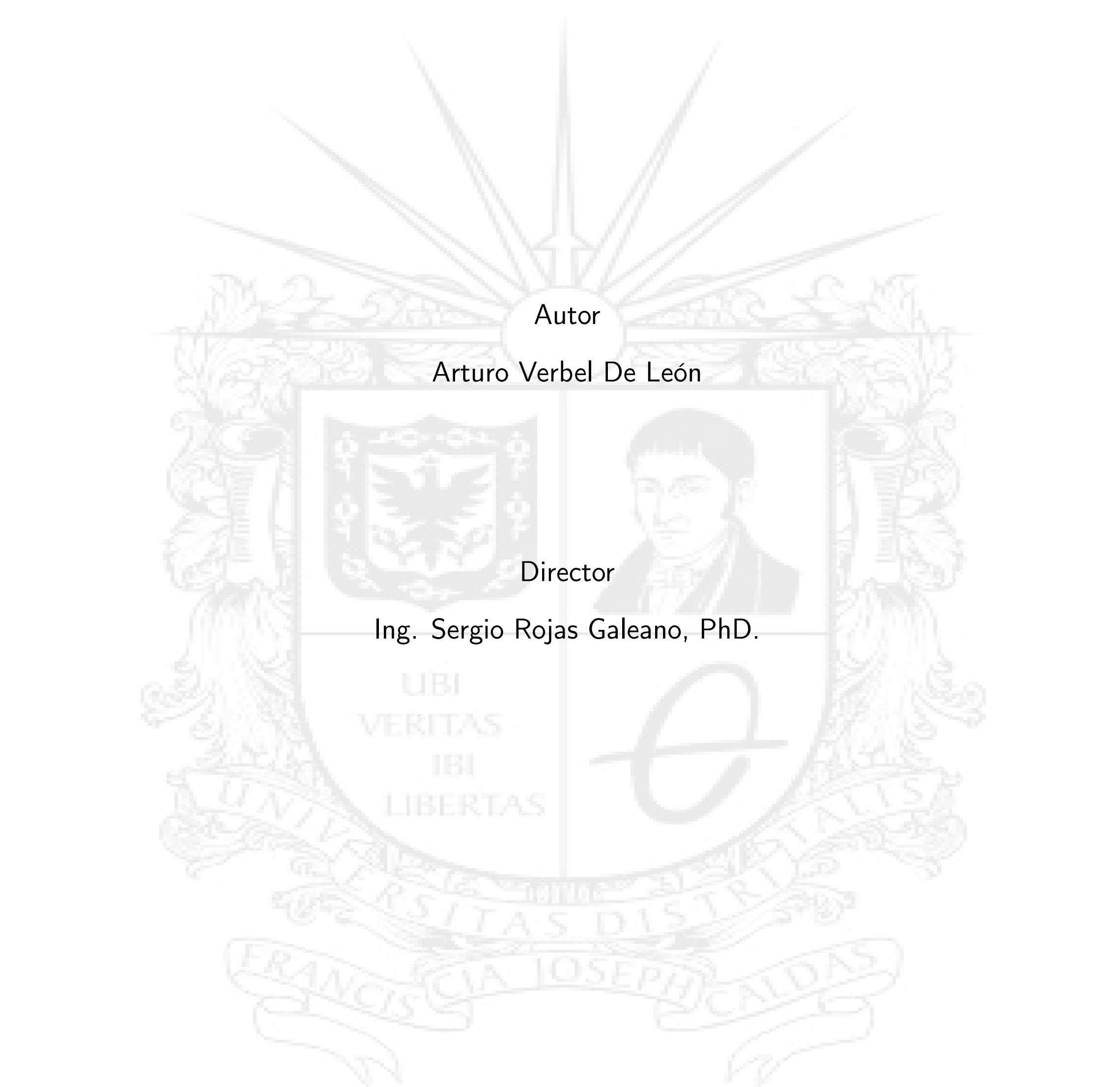
Maestría en Ciencias de la Información y las Comunicaciones

Énfasis en Teleinformática

Bogotá, Colombia

Octubre de 2021

Actualización de Todos los Caminos más Cortos en Grafos Dinámicos con Inserción de Arcos



Autor

Arturo Verbel De León

Director

Ing. Sergio Rojas Galeano, PhD.

Universidad Distrital Francisco José de Caldas

Maestría en Ciencias de la Información y las Comunicaciones

Énfasis en Teleinformática

Bogotá, Colombia

Octubre de 2021

TABLA DE CONTENIDO

1. RESUMEN . . . 7
2. AGRADECIMIENTOS . . . 8
3. INTRODUCCIÓN . . . 9
   3.1. Problema de Investigación y motivación . . . 10
   3.2. Objetivos . . . 11
      3.2.1. Objetivo general . . . 11
      3.2.2. Ojetivos específicos . . . 11
   3.3. Revisión de literatura . . . 12
      3.3.1. Introducción a Teoría de Grafos . . . 12
      3.3.2. Problema de caminos más cortos . . . 13
      3.3.3. Grafos Dinámicos . . . 14
      3.3.4. APSP para grafos dinámicos . . . 15
      3.3.5. Even & Gazit . . . 16
      3.3.6. Ramalingam & Reps . . . 16
      3.3.7. QUINCA . . . 17
      3.3.8. Muteb & Abdelmounaam . . . 17
   3.4. Publicaciones . . . 18
4. ALGORITMO ABM . . . 19
   4.1. Motivación . . . 19
   4.2. Diseño de algoritmo para cálculo de distancias . . . 20
   4.3. Ejemplo de funcionamiento para cálculo de distancias . . . 22
   4.4. Análisis computacional . . . 25
   4.5. Diseño de algoritmo para cálculo de caminos . . . 25
   4.6. Ejemplo de funcionamiento para cálculo de caminos . . . 26
   4.7. Implementación . . . 29
      4.7.1. Métodos de actualización incremental . . . 29
      4.7.2. Algoritmos implementados . . . 31
5. EXPERIMENTOS . . . 34
   5.1. Experimentos con Grafos sintéticos . . . 34
      5.1.1. Escenario de laboratorio . . . 34
      5.1.2. Discusión . . . 37
   5.2. Experimentos con Datasets reales . . . 42
      5.2.1. Escenario de laboratorio . . . 42
      5.2.2. Discusión . . . 43
6. CONCLUSIONES Y TRABAJOS FUTUROS . . . 44
7. REFERENCIAS . . . 45

8. ANEXOS 46
8.1. Algoritmo ABM Update Predecesores 46
8.2. Algoritmo Even Gazit 47
8.3. Algoritmos de Ramalingam & Reps 48
8.4. Algoritmo QUINCA 51
8.5. Algoritmo de Muteb Abdelmounaam 52
8.6. Artículo WEA 54

LISTA DE FIGURAS

1. Grafo dirigido y su matriz de distancia más corta 22
2. Grafo dirigido después de una inserción por arco 22
3. Matriz de distancia con actualización de nodos destinos 23
4. Matriz de distancia con identificación de arcos fuente y matriz resultante 23
5. Grafo dirigido después de una segunda inserción por arco 24
6. Grafo dirigido después de una segunda inserción por arco 24
7. Grafo dirigido y su matriz de predecesores 26
8. Grafo dirigido después de una inserción por arco 27
9. Matriz de predecesores con actualización de nodos destinos 27
10. Matriz de distancia con identificación de arcos fuente y matriz resultante 27
11. Grafo dirigido con matriz de predecesores después de una segunda inserción por arco 28
12. Grafo dirigido después de una segunda inserción por arco 28
13. Resultados para $n = 100$ y $p \in \{0.01, \ldots, 0.5\}$ para inserción de arco aleatorio. Baja densidad (a), alta densidad (b). 37
14. Resultados para $n = 100$ y $p \in \{0.01, \ldots, 0.5\}$. para disminución de arco aleatorio. Baja densidad (a), alta densidad (b). 38
15. Resultados para $n = 100$ y $p \in \{0.01, \ldots, 0.5\}$ para inserción de arco en el peor escenario. Baja densidad (a), alta densidad (b). 38
16. Resultados para $n = 100$ y $p \in \{0.01, \ldots, 0.5\}$ para disminución de arco en el peor escenario. Disminución de arco peor escenario: Baja densidad (a), alta densidad (b). 39
17. Resultados para $n = 1000$ y $p \in \{0.01, \ldots, 0.5\}$. para inserción de arco aleatorio: Baja densidad (a), alta densidad (b). 39
18. Resultados para $n = 1000$ y $p \in \{0.01, \ldots, 0.5\}$. para disminución de arco. Baja densidad (a), alta densidad (b). 40
19. Resultados para $n = 1000$ y $p \in \{0.01, \ldots, 0.5\}$. para inserción de arco en el peor escenario: Baja densidad (a), alta densidad (b). 40
20. Resultados para $n = 1000$ y $p \in \{0.01, \ldots, 0.5\}$. para disminución de arco en el peor escenario: Baja densidad (a), alta densidad (b). 41

LISTA DE ALGORITMOS

1. ABM 20
2. Implementación en Python de ABM para cálculo de distancias 21
3. ABM para cálculo de caminos 26
4. Crear un grafo y realizar actualización aleatoria incremental 30
5. Crear un grafo aleatorio, realizar una inserción aleatoria y calcular con ABM 33
6. Implementación en Python de ABM para cálculo de caminos 46
7. Implementación en Python de Even & Gazit 47
8. Implementación en Python de Ramalingam & Reps (Dijkstra Truncado) 48
9. Implementación en Python de Ramalingam & Reps (BFS Truncado) 49
10. Implementación en Python de Ramalingam & Reps (BFS Truncado con fuentes) . . 50
11. Implementación en Python de QUINCA 51
12. Implementación en Python de Muteb & Abdelmounaam 53

LISTA DE TABLAS

1. Lista de algoritmo de APSP en grafos dinámicos 16
2. Ejemplo de resultados del laboratorio para grafos sintéticos 36
3. Resultados de conjunto de datos reales para inserción de arco aleatorio. La columna **Estático (s)** indica el tiempo en segundos de un algoritmo estático. Las últimas 5 columnas se muestra el tiempo en segundos de cada algoritmo. 43
4. Resultados de conjunto de datos reales para disminución de peso en un arco aleatorio. Las últimas 5 columnas se muestran los tiempos en segundos de cada algoritmo. . . . 43

1 RESUMEN

En la actualidad existe gran cantidad de sistemas compuestos por actores o agentes que interactúan entre sí, tales como redes sociales, redes de transporte, redes de publicaciones académicas, redes de computadoras físicas, entre otras. El formalismo matemático de estos sistemas se conocen como grafos. Las aplicaciones modernas que utilizan estos sistemas manejan grandes cantidades de datos y muchas actualizaciones en un corto lapso de tiempo, lo que ha hecho resurgir un gran interés en problemas de la teoría de grafos pero con un enfoque dinámico. Uno de ellos en particular, es el problema de la actualización de los caminos más cortos entre todos los nodos (APSP, sigla en inglés de All-Pairs Shortest Paths) ante la inserción repetida de uno o algunos nuevos arcos en un grafo establecido, el cual resultaría muy costoso computacionalmente partiendo desde cero, por lo que justifica estudiar alternativas que permitan actualizar los APSP con base en cálculos obtenidos con previas configuraciones de la estructura del grafo.

El presente trabajo de investigación propone el diseño e implementación de un nuevo algoritmo para actualización de APSP en grafos dinámicos con inserción de arcos en grandes volúmenes de datos, con base a ideas preliminares desarrolladas en el grupo LAMIC de la Universidad Distrital. Los algoritmos clásicos tales como Dijsktra o Floyd-Warshall, requieren mucho tiempo en calcular el APSP cada vez que un grafo dinámico se actualiza. El algoritmo propuesto, actualiza las distancias en base a cálculos obtenidos con previas configuraciones de la estructura del grafo. Así mismo se propone una nueva variante del algoritmo que no solo compute la distancia, sino el camino como tal (secuencia de nodos que une cada nodo con los demás).

Dicho algoritmo es más simple en estructura con el objetivo de ser mejor aplicado en la práctica y que cubra los mismos rendimientos que las anteriores técnicas de cálculo de APSP después de una actualización incremental.

PALABRAS CLAVE

Grafos, APSP, inteligencia artificial, estructura de datos, complejidad algorítmica.

2 AGRADECIMIENTOS

A mi familia por ser el motor principal para seguir adelante. A mis padres por darme el apoyo más grande que un hijo pueda tener; por darme la educación, orientación y formarmación para ser la persona integral que soy hoy por hoy.

A mis compañeros de clase y amigos, quienes siempre han estado a mi lado demostrando su apoyo incondicional, lo cual me ayudó a superar muchas dificultades gracias a sus ánimos y buenas energías.

A nuestros docentes por todas aquellas enseñanzas que nos inculcaron para ser mejores personas y mejores profesionales cada día.

Quiero expresar la inmensa gratitud que le tengo a mi tutor de tesis, docente, compañero de trabajo y amigo, el doctor Sergio Rojas por toda la ayuda que me brindó durante este proceso, su paciencia y confianza fueron un apoyo fundamental para la realización de este trabajo. Siempre estuvo atento con el proceso de mi formación profesional.

A todos ustedes mis más sinceros agradecimientos.

3 INTRODUCCIÓN

La teoría de grafos es una de las ramas de la ciencia de la computación acerca del estudio de estructuras caracterizadas por nodos (personas, ciudades, computadores, palabras, fotos, etc) enlazados por algún tipo de afinidad (amistad, carretera, conexión, sinonimia, autor, etc). Existen una gran cantidad de aplicaciones en diversas áreas, como por ejemplo redes sociales, sistemas de movilidad, sistemas de transmisión de datos, redes neuronales de inteligencia artificial, entre otras.

Se define un grafo como un par de (V, E) donde $V = \{v_1, v_2, \ldots, v_n\}$ es un conjunto finito de vértices o nodos y $E = \{(v_i, v_j) : v_i, v_j \in V\}$ es un conjunto de arcos que relacionan estos nodos (Cormen et al., 2001). Se define: $w : E \rightarrow \mathbb{R}_{\geq 0}$ como la función del peso del arco. Se define $n = |V|$ como el número de nodos que tiene el grafo, y $m = |E|$ como el número de arcos que posee. Por ejemplo: $G = (\{s, t, u, v\}, \{(s, t), (t, u), (v, t), (u, v)\})$, es un grafo con cuatro nodos y 4 arcos, es decir, $n = 4$, $m = 4$. Cuando estas entidades se mantienen con el tiempo, se define como grafo estático. Un claro ejemplo son las direcciones de una ciudad, cada avenida o calle serían los arcos y las intersecciones de los nodos [1].

Entre los problemas en la teoría de grafos encontramos el problema del camino más corto de todos los pares de nodos dado un nodo origen, conocido por su abreviatura en inglés SSSP (Single Source Shortest Path), y el cálculo de las rutas más cortas entre pares de nodos, conocido por su abreviatura en inglés APSP; son los problemas más frecuentados en la práctica. Existen varias aplicaciones para las distancias cortas como análisis de redes sociales, redes de transporte, sistemas de bases de datos y otras.

Uno de los algoritmos más recientes que resuelve este problema de gragos es el de Slobbe (Slobbe et al., 2016). Este estudio proponen un método eficiente en el escenario de peor caso con una complejidad algorítmica de $O(n^2)$ basado en el algoritmo de Dijkstra. Según sus experimentos en redes reales muestra que supera los métodos existentes, sobretodo en inserción de nodo.

En este trabajo se porpone un algoritmo del cálculo de APSP con una complejidad igual al peor de los casos, pero con mayor simplicidad algorítmica que pueda hacerlo fácil de implementar con estructuras de simples. Lo que pretende este estudio es validar su viabilidad en la práctica frente a las demás propuestas.

[1] Debido a las variaciones de tráfico, no se consideran grafos estáticos, más bien grafos dinámicos por arcos.

3.1 Problema de Investigación y motivación

Encontrar caminos con distancias más cortas es un problema fundamental en el análisis de grafos y se ha estudiado ampliamente desde los inicios de la informática. En el caso de grafos con estructuras estáticas (es decir, conjuntos fijos de nodos y arcos), los algoritmos establecidos como Dijkstra y Floyd-Warshall son capaces de encontrar efectivamente pares de fuente única y todos rutas más cortas (conocidas como SSSP y APSP respectivamente). Estos algoritmos son el núcleo de las tecnologías de rutas ampliamente utilizadas en las redes de comunicación y de transporte. El tamaño creciente de estas redes en los últimos años ha motivado el estudio de estrategias más eficientes destinadas a hacer frente a las limitaciones de tiempo asociadas a dichos volúmenes, en particular, resolviendo las rutas más cortas punto a punto. Por lo general, en esos escenarios la estructura de la red es conocida y fija, por lo que los algoritmos antes mencionados se pueden usar para calcular de manera eficiente las distancias desde cero, con un costo $O(n(m + n \log n))$, donde n es el número de nodos y m es el número de arcos en el grafo.

Un problema diferente en el que la estructura del grafo cambia dinámicamente con el tiempo (sea agregando, eliminando o actualizando arcos o nodos), ha atraído nueva atención por parte de los investigadores. Muchas actividades de la vida moderna pueden enmarcarse dentro de ese entorno. Tomemos, por ejemplo, las redes de datos inalámbricas ad-hoc, en las que el enrutamiento cambia continuamente según la disponibilidad de los nodos conectados. Del mismo modo, los grafos de las interacciones que ocurren en las redes sociales digitales y las llamadas de teléfonos móviles cambian rápidamente su estructura y, por lo tanto, los caminos más cortos u otras propiedades interesantes del grafo, como la accesibilidad, la centralidad o los diámetros, deben mantenerse dinámicamente. Los sistemas de navegación deben lidiar con las condiciones del tráfico en tiempo real para proporcionar recomendaciones de rutas más cortas. La pandemia actual de COVID-19 está proporcionando evidencia de que rastrear las redes de transmisión de contacto para estimar la exposición al virus en función de la distancia a los casos confirmados, es una de las políticas de salud pública más efectivas para contener la propagación de la enfermedad.

Dada la pertinencia del cómputo APSP para grafos dinámicos con inserción de arcos y su efectiva aplicación en una gran variedad de tecnologías informáticas y de telecomunicación en la actualidad, se expone a continuación el problema de investigación y motivación de este trabajo. El enfoque de este trabajo de investigación se enmarca en la implementación y proposición de un algoritmo de APSP para grafos con inserción de arcos.

3.2 Objetivos

3.2.1 Objetivo general

Proponer un algoritmo que calcule el APSP en grafos dinámicos con inserción de arcos que posea una complejidad algorítmica por inserción de arco, al menos igual a los trabajos propuestos ya existentes, pero que utilice estructuras computacionales más simples.

3.2.2 Ojetivos específicos

- Identificar los elementos y las características del algoritmo propuesto, para implementarlo y verificar su validez de manera empírica.
- Implementar el algoritmo para calcular no solo la distancia más corta, sino también las rutas más cortas, entendiendo por ruta el camino o secuencia de nodos y arcos que unen un punto de partida con uno de llegada.
- Realizar un estudio comparativo de eficiencia, así como una validación empírica y teórica de su complejidad algorítmica, contra otros algoritmos propuestos recientemente aplicados con datos en grandes cantidades de datos.
- Presentar las ventajas y debilidades del algoritmo presentado, y analizar posibilidades de mejora y escenarios de aplicación.

3.3 Revisión de literatura

3.3.1 Introducción a Teoría de Grafos

Los grafos son estructuras matemáticas aplicadas para modelar relaciones por pares entre objetos; estos objetos son conocidos con los nombres de vértices o nodos unidos. Las relaciones entre estos pares de objetos se definen como aristas o arcos, las cuales permiten representar relaciones binarias entre elementos de un conjunto. Típicamente, un grafo se representa de manera gráfica como un conjunto de puntos (nodos) unidos por líneas (arcos) (Trudeau, 1993).

Puntualmente un grafo se define como un par de (V, E) donde $V = \{v_1, v_2, \ldots, v_n\}$ es un conjunto finito de vértices o nodos y $E = \{(v_i, v_j) : v_i, v_j \in V\}$ es un conjunto de arcos que relacionan estos nodos (Cormen et al., 2001). Se define: $w : E \to \mathbb{R}_{\geq 0}$ como la función del peso del arco. Se define $m = |V|$ como el número de nodos que tiene el grafo, y $n = |E|$ como el número de arcos que posee. Por ejemplo: $G = (\{s, t, u, v\}, \{(s, t), (t, u), (v, t), (u, v)\})$, es un grafo con cuatro nodos y 4 arcos.

Los grafos son utilizados para redes de comunicación, organización de datos, dispositivos computacionales, flujos de computación, entre otros campos. También pueden ser empleados en otros campos además de las ciencias computacionales, tales como redes sociales (Grandjean, 2016), viajes, biología, diseño de chips de computadora y un sinfín de aplicaciones. El desarrollo de algoritmos para manejar y estudiar los grafos es de gran interés para la informática.

La **Teoría de Grafos** es una rama de las matemáticas y las ciencias de la computación sobre el estudio de los grafos. Debido a la gran cantidad de aplicaciones en la optimización de recorridos, procesos, flujos, algoritmos de búsquedas, entre otros; se presentan a continuación algunos de los distintos problemas que estudia la teoría de grafos:

- **Subgrafos**: Un problema común es encontrar un grafo fijo en un grafo dado. El interés de este tipo de problema es que muchas propiedades de grafos son heredadas de subgrafos. En este estudio encontramos el problema de la clique y el problema del conjunto independiente.
- **Ciclos y caminos hamiltonianos**: Un ciclo es una sucesión de arcos adyacentes, donde no se recorre el mismo arco dos veces y donde se regresa al punto inicial. Un camino hamiltoniano tiene además que recorrer todos los nodos exactamente una vez (excepto el nodo del que parte y el cual llega). Este tipo de problema es el mismo que planteó Euler en 1736.

- **Coloración de nodos y arcos**: El problema consiste en colorear un grafo de modo que no haya dos nodos adyacentes que tengan mismo color o con otras restricciones. El problema del mapamundi en cuatro colores propuesto en 1852 es un ejemplo de este tipo de estudio.
- **Problemas de rutas**: Encontrar una ruta con alguna característica o restricción. Este estudio posee varios problemas, entre ellos se mencionan:
  - **Problema del camino hamiltoniano**: Consiste en encontrar el camino hamiltoniano en un grafo dirigido o no dirigido (Garey and Johnson, 1979).
  - **Árbol de expansión mínimo**: o árbol de expansión de peso mínimo consiste en encontrar el camino que conecte todos los nodos y dicho camino sea el peso mínimo de todas las rutas que conecten al grafo. Los grafos son sin orientación.
  - **Problema del camino más corto**: Es encontrar la ruta más corta entre dos nodos en un grafo de manera que se minimice la suma de los pesos de sus arcos.

3.3.2 Problema de caminos más cortos

Es uno de los problemas más comunes que se presentan en la teoría de grafos. Tienen distintas variaciones, las cuales son:

- **SSSP**: (por sus siglas en inglés: *single-source shortest path problem*). El problema es encontrar las rutas más cortas desde un nodo fuente v a todos los demás nodos en el grafo (Cormen et al., 2001).
- **SDSP**: (por sus siglas en inglés: *single-destination shortest path problem*). El problema es encontrar las rutas más cortas desde todos los nodos en el grafo dirigido a un nodo de destino v (Cormen et al., 2001).
- **APSP**: (por sus siglas en inglés: *all-pairs shortest path problem*). El problema es encontrar las rutas más cortas entre cada par de nodos v, v' en el grafo (Cormen et al., 2001).

El problema de APSP está definido para grafos no dirigidos, dirigidos o mixtos. Dos nodos son adyacentes cuando ambos comparten un arco en común. Un camino en un grafo no dirigido es una secuencia de arcos $P = (e_1, e_2, ..., e_n) \in E \times E \times ... \times E$ tal que e_i es adyacente a e_{i+1} para $1 \leq i < n$. Tal ruta P se llama una ruta de longitud $n - 1$ desde e_1 a e_n. (Las e_i son variables, su numeración se relaciona con su posición en la secuencia y no necesita relacionarse con ninguna etiqueta canónica de los arcos) (Cormen et al., 2001).

Sea $e_{i,j}$ el arco incidente tanto para e_i como para e_j. Dado una función de peso real $f : E \to \mathbb{R}$ y un grafo no dirigido simple G. La ruta más corta desde e a e' es el camino $P = (e_1, e_2, ..., e_n)$, donde $e = e_1$ y $e' = e_n$, que todos los posibles n minimize la suma:

$$f(P(u,v)) = \sum_{i=1}^{n-1} f(e_i)$$

Para cada par de arcos (u, v) tal que v es adyacente de u, se requiere encontrar una ruta $P^*(u, v)$ que tenga el peso total mínimo posible.

3.3.3 Grafos Dinámicos

Cuando en un grafo sus estructuras cambian con el tiempo, se conocen como grafos dinámicos (Ramalingam and Reps, 1996). Están clasificados como:

- **Grafos dinámicos por nodos:** El conjunto de nodos varía con el tiempo. Se conocen como crecientes si se agregan nodos al grafo, en cuyo caso los nuevos nodos también se agregan con arcos relacionados; decrecientes si se eliminan nodos del grafo, también se eliminan los arcos incidentes con ellos y completamente dinámicos cuando se agregan y/o se eliminan nodos. Un ejemplo claro son las redes sociales de artículos de revistas científicas, como ResearchGate, es una aplicación de esta categoría debido a los nuevos artículos publicados que se agregan diariamente.
- **Grafos dinámicos por arcos:** El conjunto de arcos varía con el tiempo. Los arcos pueden agregarse o eliminarse del grafo pero los nodos se mantienen. Casos como Facebook y Twitter son aplicables en esta categoría debido a que sus conexiones se agregan o se eliminan con el tiempo. La adición y eliminación de rutas en los sistemas de transporte masivo, como el Transmilenio de Bogotá, es otro ejemplo de esta categoría que conectan y desconectan estaciones o paradas de los buses del sistema.
- **Grafos dinámico ponderado:** Los pesos de cada arco del grafo varían con el tiempo. Es aplicable en cálculos de rutas más cortas como en rutas de transporte como el Transmilenio en Bogotá. Las estaciones, que vendrían siendo los nodos, y las conexiones entre ellas, se mantienen; pero la duración del viaje, que sería su peso ponderado, varía con el tiempo dependiendo de la demanda de los pasajeros.

3.3.4 APSP para grafos dinámicos

Existen muchos algoritmos para detectar el camino más corto de todos los pares en grafos estáticos. Para grafos dinámicos, la solución es más difícil, debido al costo computacional que conlleva calcular desde cero el APSP cada vez que el grafo dinámico actualiza su estructura. El problema se agrava cuando se trata de grandes cantidades de datos que se actualizan con gran velocidad; casos como los análisis de redes sociales en tiempo real, inteligencia artificial que toma decisiones en árboles o estructuras de datos de gran volumen, aplicaciones celulares para movilidad, descongestionamiento de tráfico, entre otras.

Se han propuesto varios algoritmos para calcular el APSP de un grafo dinámico desde hace cinco décadas. Algunos de ellos son:

- El primer algoritmo se propuso en 1967 por Peter S. Loubal en su libro *A Network Evaluation Procedure* (Loubal and Commission, 1967). Tiene una complejidad de $O(n^2)$ en el peor de los casos para las inserciones de arcos. La idea de este algoritmo es: Teniendo un nuevo arco (u, v) en el grafo, con peso $w(u, v)$, para cada par de nodos (x, y). Se asigna un nuevo camino más corto si cumple que la suma $d(x, u)+w(u, v)+d(v, y)$ (Donde $d(x, y)$ representa la distancia entre el node x y u) sea menor a $d(x, y)$

- El algoritmo de Ramalingam y Reps (Ramalingam and Reps, 1996) propuesto en 1996 es uno de los más conocidos, otros enfoques propuestos recientemente se han derivado de este. Proponen algoritmos con pesos reales arbitrarios tanto para el SSSP como para el APSP. Sin embargo los tiempos de ejecución en el peor de los casos son los mismos como si calculara de nuevo. Posee una complejidad algorítmica de $O(mn)$. Más información de este algoritmo en la sección de Identificación del Problema.

- En 2004 se proponen los primeros algoritmos asintóticamente más rápidos para grafos completamente dinámicos con pesos de arcos de números reales. Fueron propuestos por Demetrescu e Italiano (Demetrescu and Italiano, 2004). Su algoritmo requiere $O(n^2 log^3 n)$ por actualización. Este algoritmo fue mejorado por Thorup (Thorup, 2004), cuya complejidad algorítmica alcanza $O(n^2(logn + log^2((m + n)/n)))$.

- Uno de los algoritmos más recientes (Slobbe et al., 2016) proponen un método más eficiente que el algoritmo anterior en el escenario de peor caso con una complejidad algorítmica de $O(n^2)$ basado en el algoritmo de Dijkstra. Según sus experimentos en redes reales muestra que supera los métodos existentes, sobretodo en inserción de nodo.

Algoritmo	Nombre	Herramientas	Complejidad
Even & Gazit	EV	Matriz de distancia	$O(n^2)$
Ramlingam & Reps	RR	BFS y colas de prioridad	$O(mn)$
Slobbe	QUINCA	BFS	$O(n^2)$
Muteb & Abdelmounaam	MA	Colas de prioridad y Árboles	$\Theta(n^2)$
Propuesto	ABM	Matriz de distancia	$O(n^2)$

Tabla 1: Lista de detalles de cada algoritmo para los problemas de APSP en grafos dinámicos.

En la Tabla 1 se presenta un resumen de los algoritmos de estudio, la columna de herramientas hace referencia a las estructuras de datos que se manejan en el algoritmo y/o los algoritmos en los que se basan, a su vez también se coloca en evidencia la complejidad algorítmica para cada uno.

En las secciones siguientes se puntualiza con más detalle sobre el desarrollo funcional de los algoritmos mencionados.

3.3.5 Even & Gazit

Even y Gazit (Even and Gazit, 1985) fueron los primeros en proponer un algoritmo para problemas APSP para grafos dinámicos con una complejidad de $O(n^2)$ en el peor de los casos para inserción de arcos. La idea consiste en nombrar (u, v), el arco recién insertado, para cada par de nodos (x, y), para ello se toma el mínimo entre la antigua distancia $d(x, y)$ y la suma $d(x, u) + (u, v) + d(v, y)$, donde (u, v) corresponde al peso del arco (u, v). En el Algoritmo 7 de la sección de anexos se presenta la implementación en Python de este algoritmo.

3.3.6 Ramalingam & Reps

Geetha Ramalingam y Thomas William Reps (Ramalingam and Reps, 1996) propusieron algoritmos dinámicos de camino más corto para grafos con pesos reales arbitrarios, tanto para el problema SSSP como para el APSP. En esta entrada hablaremos sólo de su algoritmo incremental para grafos con actualización de arco. Para encontrar el APSP, RR propone un algoritmo con el fin de optimizar el tiempo de respuesta. Según las pruebas, el tiempo es menor que correr un algoritmo como Floyd-Warshall. Ramalingam y Reps nos muestra una manera mejor para resolverlo, si sabemos los nodos destino que posee nuestro nodo fuente por la actualización, entonces estos nodos destino los agregamos como fuente en otro cola. Para esto se utiliza un código BFS (Breadth- First Search, algoritmo para búsqueda en árboles) en Dijkstra.

Tenemos un grafo G el cual sufre una actualización en uno de sus arcos, lo denotaremos como $(u, v, w'(u, v))$ donde u y v son nodos del grafo y $w'(u, v)$ es el nuevo valor ponderado de este arco. Tenemos un nodo s al cual queremos encontrar todas las distancias de este nodo fuente a todos los demás nodos, denotamos esta distancia como $d'(s, \cdot)$. Se definen todos los nodos $t \in V$, tal que $d'(s, t) < d(s, t)$ como los nodos destino afectados. Las actualizaciones se hacen entonces de la siguiente manera: $d'(s, t) = d(s, u) + w'(u, v) + d(v, t)$. Es así como lo único que se necesita para actualizar es la información en $d'(s, \cdot)$ para encontrar $d'(s, t)$ para los nodos afectados por $d(v, t)$. La complejidad algoritmica de RR también recorre los arcos de salida de cada objetivo afectado varias veces, quiere decir que para un escenario en el peor de los casos, RR lleva a un tiempo de ejecución de $O(mn)$. Sus estudios sobre grafos dinámicos han sido un pilar fundamental para muchos algoritmos propuestos hoy día. En el Algoritmo 8 de la sección de anexos se presenta la implementación en Python de este algoritmo.

3.3.7 QUINCA

El estudio realizado pro Slobbe (Slobbe et al., 2016) propone un algoritmo incremental con una complejidad óptima en el peor de los casos de $O(n^2)$ con el objetivo de evitar el recálculo, siendo más rápido en la práctica que todos los métodos existentes. Demuestra que tras la inserción de una arco o un nodo, los nuevos caminos más cortos creados tienen ciertas propiedades que permiten reducir significativamente la cantidad de trabajo necesaria para volver a calcularlos, es decir, en lugar de iniciar un BFS desde v para cada una de las fuentes afectadas, se ejecuta el BFS sólo una vez para la fuente afectada u, mientras que también se actualiza al mismo tiempo las distancias de las otras fuentes afectadas, la cual se basa en una propiedad de los nodos afectados, llamando $S(y)$ al conjunto de fuentes afectadas del nodo y, es decir, $S(y) := x \in V : d0(x, y) < d(x, y)$. En el Algoritmo 11 de la sección de anexos se presenta la implementación en Python de este algoritmo.

3.3.8 Muteb & Abdelmounaam

El algoritmo presentado por estos autores (Alshammari and Rezgui, 2020) no describe el APSP para el algoritmo incremental, la complejidad descrita viene dado por un nodo de un árbol. El algoritmo dinámico incremental mantiene las rutas y distancias más cortas desde un nodo de origen, que es el nodo raíz del SPT t (shortest path tree) a todos los demás nodos. El algoritmo admite tanto la inserción de un arco como la disminución del peso del arco. El algoritmo reconstruye la sub-street (y) después de una operación de arco usando una adaptación del algoritmo de Dijkstra. El algoritmo comienza fijando el primer nodo afectado y (es decir, el punto final del arco modificado (x, y)). Luego, el algoritmo corrige los otros nodos afectados caminando

por la sub-street (y). Cabe tener en cuenta que $t\ (y) \subseteq t\ (s)$ donde es la fuente del árbol actual t.

El algoritmo se divide en tres fases: la primera fase consiste en encontrar el efecto del arco (x, y) en el SPT t, en la segunda fase, el algoritmo actualiza la distancia y el padre de y, ya que el arco (x, y) puede mejorar la distancia de y en t. Luego, el algoritmo inserta y en H. La tercera fase es esencial, ya que el algoritmo actualiza los nodos afectados para mantener SPT t. Esta fase es una adaptación del algoritmo de Dijkstra. Para ello se utiliza un bucle sobre H. Inicialmente, H contiene el punto final del borde actualizado (es decir, y). Los nodos afectados (que no sean, si los hay) se insertan en H and extraído secuencialmente durante las actualizaciones de los nodos afectados. En cada iteración, un nodo afectado (digamos u) se extrae de H y sus sucesores (es decir, los vecinos in out (u)) se inspeccionan para posibles mejoras. Si un sucesor de u se ve afectado (mejora de la distancia), entonces este sucesor se actualiza (su distancia y su puntero principal) y se inserta en H. Finalmente, el algoritmo termina solo cuando H está vacío y cuando las rutas y distancias más cortas en t se actualizan correctamente. En el Algoritmo 12 de la sección de anexos se presenta la implementación en Python de este algoritmo.

3.4 Publicaciones

En el año 2020 se realizó la presentación de los resultados en experimentos con grafos generados sintéticamente. Fueron publicados en la revista Applied Computer Sciences in Engineering (Verbel et al., 2020). Durante la presentación en el Workshop on Engineering Applications (WEA) fue premiado como mejor artículo del workshop de ingeniería 2020. El artículo se encuentra publicado en la URL: `https://link.springer.com/chapter/10.1007%2F978-3-030-61834-6_19` y que también se incluye en el Anexo Artículo WEA para el lector interesado.

4 ALGORITMO ABM

4.1 Motivación

De las siglas en inglés de matriz de bloques afectada (Affected Block Matrix). El nombre se debe al funcionamiento del algoritmo que va actualizando por secciones de bloques dentro de la matriz de distancia. En términos generales, cuando se inserta un nueva arco con costo w' (o se disminuye a un costo menor, si ya existe), el algoritmo verifica si es menor que la distancia actual d_{uv} entre estos dos nodos, si es así, actualiza esta nueva distancia para el arco uv y luego verifica qué subcaminos están afectados, desde los que comienzan en cualquier otro nodo de origen y terminan en v, y desde los que comienzan en u y termina en cualquier otro nodo de destino en el grafo. Esto se hace con un ciclo de tiempo lineal que atraviesa cada nodo del grafo k, registrando aquellos que cumplen con la condición de relajación $(d_{ku} + w'_{uv}) < d_{kv}$ en una lista de fuentes afectadas A_s, y aquellas que cumplen con la condición de relajación $(w'_{uv} + d_{vk}) < d_{uk}$ en una lista de objetivos afectados A_t.

En el Algoritmo 1 se puede encontrar el pseudocódigo del algoritmo propuesto. Los valores de entrada del algoritmo son la matriz de distancia d y el arco que se acaba de actualizar, u y v como los nodos y w' como el peso.

La linea 1 nos confirma que la actualización efectivamente es mejor al peso anterior. Se inicializan las listas A y D que representan el conjunto de nodos ascendentes (A) y el conjunto de nodos descendientes (B).

La siguiente parte que abarca desde la linea 4 hasta la linea 8 nos permite identificar cuáles nodos hacen parte de los nodos sucesores y predecesores a partir de la actualización. Los sucesores (o ascendentes) los agregamos a la lista A si la distancia es más corta; los predecesores (o descendientes) los agregamos a la lista B si la distancia es más corta después de la actualización.

Luego de tener todos los nodos que se ven afectatos por la actualización, tanto los descendientes y los ascendentes, se pasa al último paso del algoritmo en donde se actualiza la matriz de distancia por cada uno de los nodos de cada lista.

La segunda parte del algoritmo tiene una complejidad lineal y la tercera parte es una complejidad cuadrática claramente que depende de la primera. En un escenario del peor de los casos tiene $O(n^2)$. Un escenario donde todos los nodos del grafo se ven afectados por la actualización.

Algoritmo 1 ABM

Input: matriz de distancias d, arco actualizado u, v, w'_{uv}
Output: matriz de distancias d
1: **if** $w'_{uv} < d(u, v)$ **then**
2: $\quad A = [\ \ ]$
3: $\quad D = [\ \ ]$
4: $\quad$ **for** $n \in N$ **do**
5: $\quad\quad$ **if** $w'_{uv} + d(v, n) < d(u, n)$ **then**
6: $\quad\quad\quad D = D \cup n$
7: $\quad\quad$ **if** $d(n, u) + w'_{uv} < d(n, v)$ **then**
8: $\quad\quad\quad A = A \cup n$
9: $\quad$ **for** $j \in D$ **do**
10: $\quad\quad$ **for** $i \in A$ **do**
11: $\quad\quad\quad d(i, j) = d(i, u) + w'_{uv} + d(v, j)$
12: **return** d

La idea fundamental de la implementación de este algoritmo es manejar estructuras de datos más sencillas, más cortas y que presenten un desempeño tan bueno como los demás algoritmos de APSP dinámicos, esto lo cual lo hace mucho más ideal para ser implementados en la práctica.

4.2 Diseño de algoritmo para cálculo de distancias

El Algoritmo 2 es la implementación en Python del algoritmo propuesto para este proyecto. En la linea 1 se importa la librería `deque` de `collections` (Python-Software-Foundation, 2021) porque este es mucho más rápido para procesos de insertar datos que en una lista.

La linea 5 se utiliza una propiedad de la clase de Graph que devuelve la última actualización por arco del grafo, la primera y segunda variable corresponden a los nodos del arco actualizado y la última variable al peso.

Las lineas 7 y 8 realiza una validación para confirmar que efectivamente la actualización es incremental, comparando el peso de la última actualización (`c_uv`) con el peso que se encuentra en la matriz de distancias (`dist[u, v]`). Si es mayor o igual entonces no es una actualización incremental y termina el algoritmo.

Algoritmo 2 Implementación en Python de ABM para cálculo de distancias

```
from collections import deque

# Affected Block Matriz = ABM Update
def ABM_Update(graph, dist):
    u, v, c_uv = graph.last_edge_updated

    if c_uv >= dist[u, v]:
        return dist

    affected_sources = deque()
    affected_targets = deque()

    for k in graph.nodes:
        if (c_uv + dist[v, k]) < dist[u, k]:
            affected_targets.append(k)
        if (dist[k, u] + c_uv) < dist[k, v]:
            affected_sources.append(k)

    for j in affected_targets:
        for i in affected_sources:
            sum = dist[i, u] + c_uv + dist[v, j]
            if sum < dist[i, j]:
                dist[i, j] = sum

    return dist
```

Las lineas 10 y 11 se inician las variables para guardar los nodos afectados descendientes (`affected_sources`) y los nodos ascendientes (`affected_targets`) tras la actualización. Se usa la estructura de datos `deque` para tener una inserción más rápida.

Las lineas 13 y 17 es la sección del algoritmo para buscar los nodos que se afectan tras la actualización. La linea 14 y 15 verifica que el peso anterior sea menor luego de la actualzación. Si es menor entonces es un nodo para actualizar y se agrega en la varable `affected_targets` que corresponde a los nodos ascendientes. Los mismo ocurre para la linea 16 y 17 para los nodos descendientes. Esta sección tiene un costo computacional de $O(n)$ porque recorre todos los nodos del grafo. El número de iteraciones siempre será n (número de nodos).

La segunda sección, de la linea 19 a la linea 23, realizamos una combinación entre los nodos ascendentes y descendientes para actualizar su peso en la matriz de distancia. En la linea 21 se realiza el nuevo peso de la actualización. Si es menor que el peso anterior en la matriz de

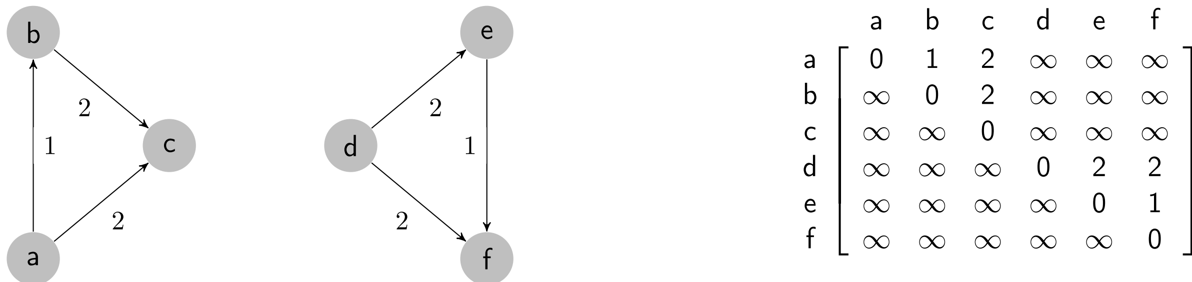
distancia entonces se realiza la actualización para esos dos nodos. El costo computacional es de $O(n^2)$ en el peor de los casos, que se da cuando `affected_targets` y `affected_sources` tengan todos los nodos insertados.

La suma de los dos costos computaciones $O(n) + O(n^2)$ es $O(n^2)$ en el peor de los escenarios.

4.3 Ejemplo de funcionamiento para cálculo de distancias

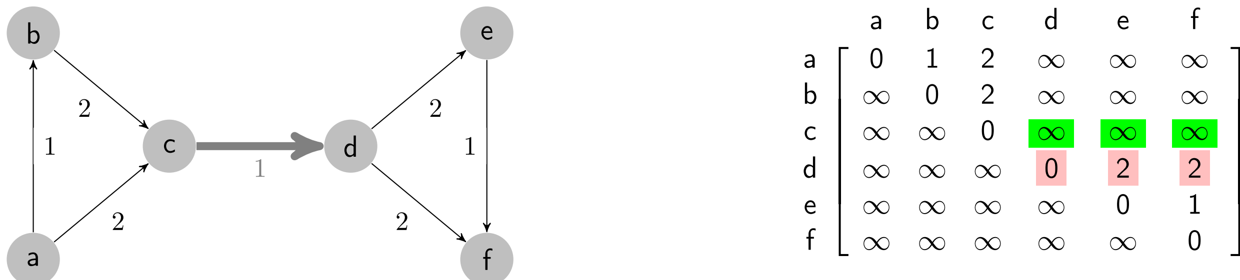
El funcionamiento del algoritmo se basa en que, cuando se incrementa el número de arcos del grafo, se identifican los nodos afectados,los nodos que le preceden a estos, y a la vez se identifican los nodos que ascienden de los nodos afectados.

Se cuenta con un grafo dirigido $G = (V, E)$ donde $V = \{a, b, c, d, e, f\}$ son los nodos y los arcos del grafo definidos como $E = \{(a, b), (a, c), (b, c), (d, e), (d, f), (e, f)\}$, se puede ver representado en la Figura 1 junto con su matriz de distancia.



	a	b	c	d	e	f
a	0	1	2	∞	∞	∞
b	∞	0	2	∞	∞	∞
c	∞	∞	0	∞	∞	∞
d	∞	∞	∞	0	2	2
e	∞	∞	∞	∞	0	1
f	∞	∞	∞	∞	∞	0

Figura 1: Grafo dirigido y su matriz de distancia más corta. Los valores marcados con ∞ indican que no es posible llegar desde el nodo origen (fila) al nodo destino (columna).



	a	b	c	d	e	f
a	0	1	2	∞	∞	∞
b	∞	0	2	∞	∞	∞
c	∞	∞	0	∞	∞	∞
d	∞	∞	∞	0	2	2
e	∞	∞	∞	∞	0	1
f	∞	∞	∞	∞	∞	0

Figura 2: Grafo dirigido después de un aumento por arco. Identificación los arcos que se actualizan de color verde y los nodos destino desde d de color rosado.

$$
\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array}
\begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\
\begin{bmatrix}
0 & 1 & 2 & \infty & \infty & \infty \\
\infty & 0 & 2 & \infty & \infty & \infty \\
\infty & \infty & 0 & 1 & 3 & 3 \\
\infty & \infty & \infty & 0 & 2 & 2 \\
\infty & \infty & \infty & \infty & 0 & 1 \\
\infty & \infty & \infty & \infty & \infty & 0
\end{bmatrix} \end{array}
$$

Figura 3: Matriz de distancia con actualización de los nodos destinos del arco insertado $(c \to d)$.

En la Figura 2 hay una actualización en la matriz con el peso del nuevo arco. Se identifica en la matriz todos aquellos nodos que vienen del nodo d, estos son aquellos nodos que en la fila de d tienen algún número, los cuales se almacenan como un arreglo de nodos $t = [d, e, f]$. Ahora se recorre cada elemento y se suma el valor del peso del arco insertado, este valor se asigna en la correspondiente fila c. El resultado se puede ver en la Figura 3. Hasta aquí el algoritmo posee una complejidad algorítmica de $O(n)$ en el peor de los casos, es decir, cuando el nuevo arco afecta todos los caminos más cortos desde el origen a todos los demás nodos del grafo.

$$
\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array}
\begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\
\begin{bmatrix}
0 & 1 & 2 & \infty & \infty & \infty \\
\infty & 0 & 2 & \infty & \infty & \infty \\
\infty & \infty & 0 & 1 & 3 & 3 \\
\infty & \infty & \infty & 0 & 2 & 2 \\
\infty & \infty & \infty & \infty & 0 & 1 \\
\infty & \infty & \infty & \infty & \infty & 0
\end{bmatrix} \end{array}
\qquad
\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array}
\begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\
\begin{bmatrix}
0 & 1 & 2 & 3 & 5 & 5 \\
\infty & 0 & 2 & 3 & 5 & 5 \\
\infty & \infty & 0 & 1 & 3 & 3 \\
\infty & \infty & \infty & 0 & 2 & 2 \\
\infty & \infty & \infty & \infty & 0 & 1 \\
\infty & \infty & \infty & \infty & \infty & 0
\end{bmatrix} \end{array}
$$

Figura 4: Grafo dirigido después de un aumento por arco. En la matriz de la izquierda, se identifican los arcos que se actualizan de color verde y los nodos destino desde d de color rosado. En la matriz de la derecha se actualizan los valores en la casilla verde sumando los valores de la columna c con la fila c.

En la matriz izquierda de la Figura 4 se identifican todos los nodos que llegan a c para ser actualizados a todos los nodos destinos que identificamos en el arreglo t. Estos nodos se almacenan con el arreglo $s = [a, b]$. A partir de aquí se realiza un recorrido del arreglo t por cada nodo del arreglo s y se actualiza cada valor de cada pareja con los valores en c de cada uno. Por ejemplo, para los valores de a y e el resultado sería $(a \to e) = (a \to c) + (c \to e)$. La matriz resultante queda expuesta en la matriz de la derecha en la Figura 4.

En la Figura 5 se aprecia otra actualización del grafo G, una que afecta a toda la matriz de distancia, esto debido a que todos los nodos pueden alcanzar a todos los demás nodos.

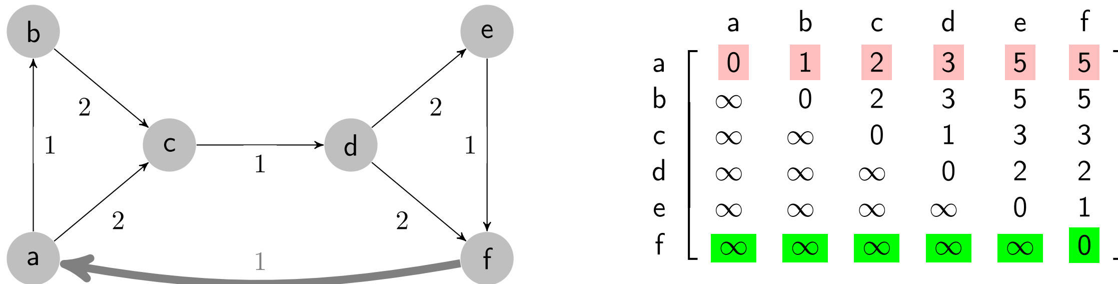


Figura 5: Grafo dirigido después de un segundo aumento por arco. Identificación los arcos que se actualizan de color verde y los nodos destino desde a de color rosado.

Como se hizo en la Figura 2, en la Figura 5 se identifican los nodos objetivos luego de la actualización con color rosado, en el algoritmo ABM se almacenan en una variable t con los valores $t = [a, b, c, d, e, f]$. Luego se identifican todos los nodos que llegan a f, que serían los nodos fuentes de nuestro algoritmos, en este caso sería tendría los mismos valores de t, quedaría s y $s = [a, b, c, d, e, f]$. Como se muestra en la Figura 6

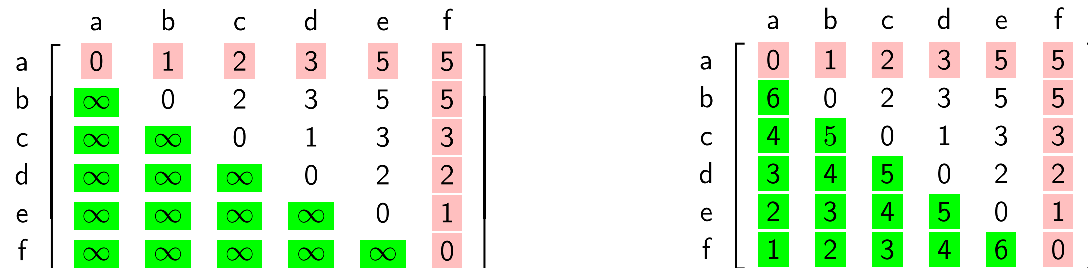


Figura 6: Grafo dirigido después de un segundo aumento por arco. Se identifican los arcos que se actualizan de color verde. Los nodos destino desde a de color rosado en la primera fila y los nodos fuentes de f también de color rosado pero en la última columna.

Luego de identificar los nodos fuentes y los nodos objetivos luego de la actualización, se procede a actualizar el cruce de estos valores. Como los nodos fuentes y destinos abarcan todos los nodos, estamos en el escenario de peor caso de una actualización, del cruze de estos valores, los que logran actualizarse son todos aquellos que tienen como ∞ en la matriz de distancias, como se ve en la Figura 6 del lado derecho.

4.4 Análisis computacional

Para calcular la complejidad algorítmica de ABM podemos dividir el algoritmo en partes. La primera parte identifica todos los nodos afectados por la actualización. En la linea 4 del Algoritmo 1 se aprecia un recorrido de cada nodo del grafo, esto lo hace una complejidad algorítmica de $O(n)$. La segunda parte depende del resultado de la identificación de estos nodos, el escenario de peor caso sería que todos los nodos del grafo fueran afectados por la actualización. Las variables A y D tendrían longitud de n, es decir todos los nodos. Su recorrido anidado en las lineas 9 y 10 tienen una complejidad algorítmica de $O(n^2)$. La suma de estas dos partes del algoritmo en el escenario de peor caso sería $O(n) + O(n^2) = O(n^2)$

El algoritmo propuesto en este trabajo tiene la misma complejidad computacional del algoritmo propuesto por QUINCA, cuyos estudios presentan mejor rendimiento que los estudios previos. Además este algoritmo posee una estructura más simple, debido a que solo utiliza como estructura de datos la matriz de distancias, dando un mayor alcance en la práctica.

4.5 Diseño de algoritmo para cálculo de caminos

Para el cálculo de caminos de un grafo utilizaremos un modelo de descendientes en una matriz para determinar el camino que tiene un nodo para llegar a otro, también es conocido como matriz de predecesores. Esta matriz ayuda a encontrar cuál es la ruta (lista de nodos) para llegar de un nodo a otro. En la figura 7 se observa la matriz de predecesores para el grafo de la izquierda.

El algoritmo implementado para el cálculo de caminos es muy parecido al algoritmo del cálculo de distancias. En la linea 2 se agrega la actualización del nuevo predecesor por la actualización. Las lineas 13-16 del pseudocódigo que se presenta en el Algoritmo 3 se actualizan los nodos precedesores. Se tiene en cuenta la condición del nodo que estamos preguntando hace parte del arco insertado, entonces se actualiza con el nodo de la actualización, caso contrario atrapa los predecesores de la matriz.

Este algoritmo se encuentra implementado en Python en la sección de Anexos, Algoritmo 2

Algoritmo 3 ABM para cálculo de caminos

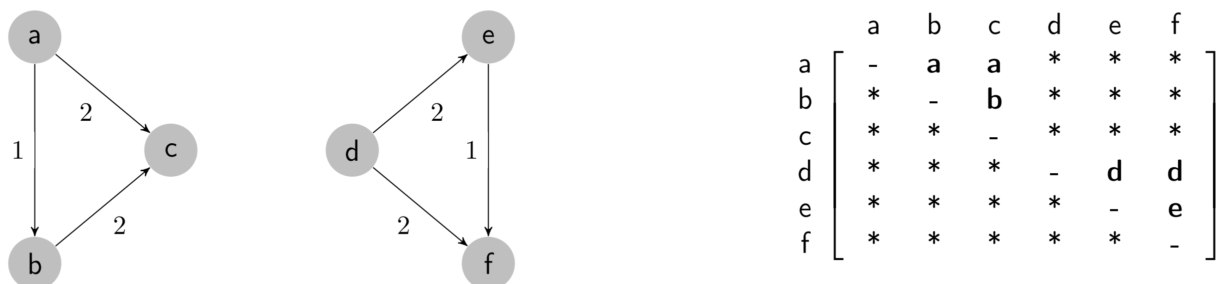
Input: matriz de distancias d, matriz de predecesores p, arco actualizado u, v, w'_{uv}
Output: matriz de predecesores p

1: **if** $w'_{uv} < d(u, v)$ **then**
2: $\quad p(u, v) = u$
3: $\quad A = [\quad]$
4: $\quad D = [\quad]$
5: $\quad$ **for** $n \in N$ **do**
6: $\quad\quad$ **if** $w'_{uv} + d(v, n) < d(u, n)$ **then**
7: $\quad\quad\quad D = D \cup n$
8: $\quad\quad$ **if** $d(n, u) + w'_{uv} < d(n, v)$ **then**
9: $\quad\quad\quad A = A \cup n$
10: $\quad$ **for** $j \in D$ **do**
11: $\quad\quad$ **for** $i \in A$ **do**
12: $\quad\quad\quad d(i, j) = d(i, u) + w'_{uv} + d(v, j)$
13: $\quad\quad\quad$ **if** $j == v$ **then**
14: $\quad\quad\quad\quad p(i, j) = u$
15: $\quad\quad\quad$ **else**
16: $\quad\quad\quad\quad p(i, j) = p(v, j)$
17: **return** p

4.6 Ejemplo de funcionamiento para cálculo de caminos

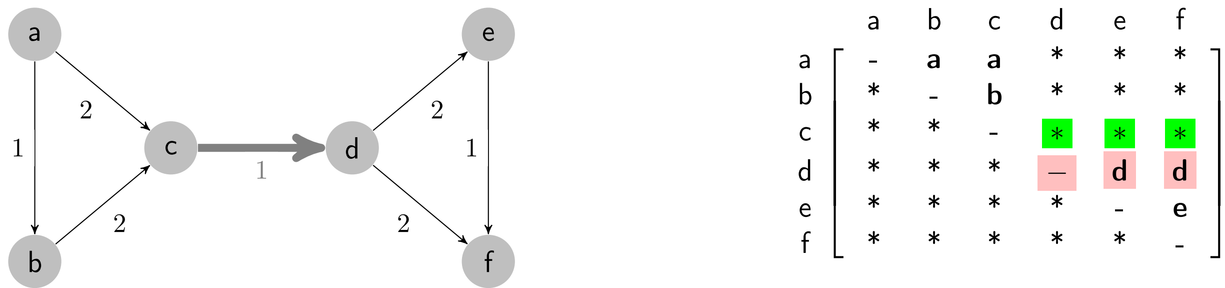
Cuando se inserta un nuevo arco al grafo, se identifican los nodos afectados, los nodos que le preceden a estos, y a la vez se identifican los nodos que ascienden de los nodos afectados.

Se cuenta con un grafo dirigido $G = (V, E)$ donde $V = \{a, b, c, d, e, f\}$ son los nodos y los arcos del grafo definidos como $E = \{(a, b), (a, c), (b, c), (d, e), (d, f), (e, f)\}$, se puede ver representado en la Figura 7 junto con su matriz de predecesores.



	a	b	c	d	e	f
a	-	**a**	**a**	*	*	*
b	*	-	**b**	*	*	*
c	*	*	-	*	*	*
d	*	*	*	-	**d**	**d**
e	*	*	*	*	-	**e**
f	*	*	*	*	*	-

Figura 7: Grafo dirigido y su matriz de predecesores. Los valores marcados con $*$ indican que no es posible llegar desde el nodo origen (fila) al nodo destino (columna).



	a	b	c	d	e	f
a	-	a	a	*	*	*
b	*	-	b	*	*	*
c	*	*	-	*	*	*
d	*	*	*	–	d	d
e	*	*	*	*	-	e
f	*	*	*	*	*	-

Figura 8: Grafo dirigido después de un aumento por arco. Identificación los arcos que se actualizan de color verde y los nodos destino desde d de color rosado.

	a	b	c	d	e	f
a	-	a	a	*	*	*
b	*	-	b	*	*	*
c	*	*	-	c	d	d
d	*	*	*	-	d	d
e	*	*	*	*	-	e
f	*	*	*	*	*	-

Figura 9: Matriz de predecesores con actualización de los nodos destinos del arco insertado $(c \to d)$.

En la Figura 8, igual que en el algoritmo para calcular la matriz de distancias, se identifica en la matriz todos aquellos nodos que vienen del nodo d, se recorre cada elemento y se actualiza el nodo predecesor como el nodo que acaba de ser insertado, este valor se asigna en la correspondiente fila c. El resultado se puede ver en la Figura 9.

	a	b	c	d	e	f
a	-	a	a	*	*	*
b	*	-	b	*	*	*
c	*	*	-	c	d	d
d	*	*	*	-	d	d
e	*	*	*	*	-	e
f	*	*	*	*	*	-

	a	b	c	d	e	f
a	-	a	a	c	d	d
b	*	-	b	c	d	d
c	*	*	-	c	d	d
d	*	*	*	-	d	d
e	*	*	*	*	-	e
f	*	*	*	*	*	-

Figura 10: Grafo dirigido después de un aumento por arco. En la matriz de la izquierda, se identifican los arcos predecesores que se actualizan de color verde y los nodos destino desde d de color rosado. En la matriz de la derecha se actualizan los valores en la casilla verde consiguiendo el precesor.

En la matriz izquierda de la Figura 10 se identifican todos los nodos que llegan a c para ser actualizados a todos los nodos destinos que identificamos en el arreglo t. Estos nodos se almacenan con el arreglo $s = [a, b]$. A partir de aquí se realiza un recorrido del arreglo t por cada nodo del arreglo s y se actualiza cada valor de cada pareja con los valores del predecesor de cada

uno. Por ejemplo para los valores de a y e el predecesor sería d y para a y d el valor sería el mismo nodo del arco actualizado. La matriz resultante queda expuesta en la matriz de la derecha en la Figura 10.

Así como en la sección anterior. En la Figura 11 se realiza una actualización de arco que logra afectar a todos los nodos.

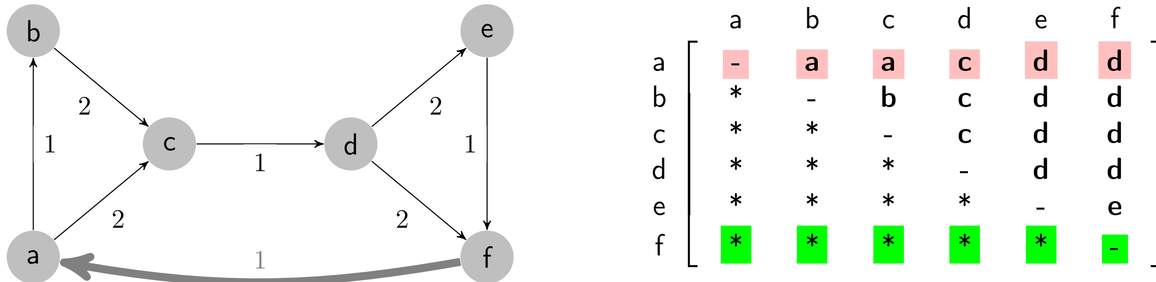


Figura 11: Grafo dirigido después de un segundo aumento por arco. Identificación los arcos que se actualizan de color verde y los nodos destino desde a de color rosado.

En la Figura 11 se identifican los nodos objetivos luego de la actualización con color rosado, en el algoritmo ABM se almacenan en una variable t con los valores $t = [a, b, c, d, e, f]$. Luego se identifican todos los nodos que llegan a f, que serían los nodos fuentes de nuestro algoritmos, en este caso sería tendría los mismos valores de t, quedaría s y $s = [a, b, c, d, e, f]$. Como se muestra en la Figura 12.

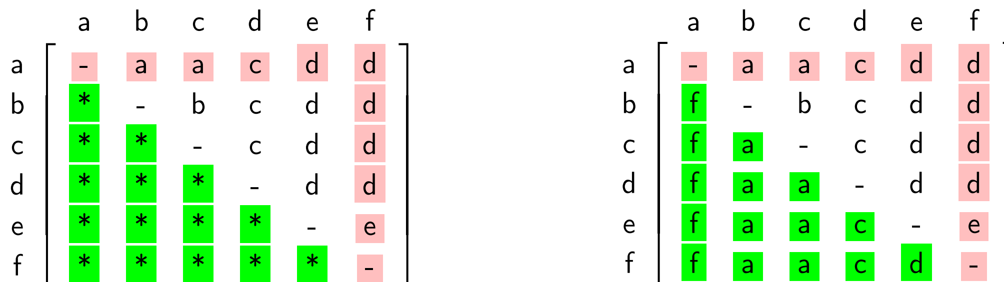


Figura 12: Grafo dirigido después de un segundo aumento por arco. Se identifican los arcos que se actualizan de color verde. Los nodos destino desde a de color rosado en la primera fila y los nodos fuentes de f también de color rosado pero en la última columna.

El proceso es igual al de la matriz de distancias, luego de identificar los nodos fuentes y destinos, se procede a actualizar el cruze entre estos. Los que logran actualizarse son todos aquellos que tienen como ∞ en la matriz de distancias, como se ve en la Figura 6 del lado derecho. Tener en cuenta la condición de la linea 13 en el algoritmo 3 debido a que el algoritmo de predecesores tiene en cuenta los nodos dentro de la matriz, toca recalcar cuando hay un nodo nuevo, en este caso f.

4.7 Implementación

Para trabajar con grafos dinámicos se creó una herramienta que permite trabajar con grafos y sus actualizaciones incrementales. Este ambiente de trabajo se encuentra desarrollado en Python3 y puede encontrarse en el repositorio público del proyecto, en el módulo de graph `https://github.com/apspgraphincremental/graph/tree/master/graph`. Las funcionalidades de esta herramientas las podemos mencionar en las siguientes categorías:

4.7.1 Métodos de actualización incremental

- `insert_edge`: Insertar un nuevo arco dado dos nodos y un peso.
- `insert_random_edge`: Insertar un nuevo arco entre dos nodos aleatorios del grafo.
- `insert_worst_edge`: Insertar un nuevo arco cuyos nodos tienen la mayor cantidad de nodos ascedentes y descendientes. Esto para crear el peor escenario de una actualización de arco.
- `decrease_edge`: Decrecer el peso de un arco dado dos nodos y un peso.
- `decrease_random_edge`: Decrecer el peso de un arco elegido aleatoriamente.
- `decrease_worst_edge`: Decrecer el peso de un arco cuyos nodos tienen la mayor cantidad de nodos ascedentes y descendientes.
- `action_incremental`: Recibe un parámetro para realizar una actualización incremental y llama alguna de las funciones anteriores.
- `reverse_action_incremental`: Devuelve los cambios de la actualización (Usado para realizar pruebas unitarias).
- `print_r`: Muestra en consola de forma amigable para el usario, los arcos del grafo.

En el algoritmo 4 se puede ver el uso de la librería de grafos. En la linea 7 se crea un grafo pasando como parametros lista de los nodos fuentes, nodos destinos y su respectivo peso. De esa

Algoritmo 4 Crear un grafo y realizar actualización aleatoria incremental

```
from Graph import Graph

sources = [1, 2, 3, 0, 2, 3]
targets = [0, 0, 0, 4, 1, 1]
weights = [3, 2, 4, 1, 3, 2]

graph = Graph(sources, targets, weights)

new_edge = graph.insert_random_edge()

print(new_edge)
graph.print_r()

"""
{'last_edge_updated': [3, 2, 7], 'last_edge_action': 'ADD'}
Source:  [0 1 2 2 3 3 3]
Target:  [4 0 0 1 0 1 2]
Weight:  [1 3 2 3 4 2 7]
Vertex:  [0 1 2 3 4]
"""
```

manera se establecen los arcos del grafo.

En la linea 9 se realiza la inserción de arco aleatorio y en las siguientes lineas se imprime, tanto el nuevo arco como el grafo generado. La salida puede verse en el comentario de la linea 14 a la linea 20. El arco insertado es `[3, 2, 7]` que es un arco que parte del nodo 3 al nodo 2 con un peso de 7.

La herramienta tiene además funcionalidades básicas de los grafos, como obtener el peso entre dos nodos, seleccionar los nodos ascendentes o desdentes dado un nodo del grafo, devolver la densidad, entre otras. También se encuentran implementados todos los algoritmos y una herramienta que permite calcular el tiempo de ejecución de cada una de ellos en un grafo actualizado. Cada uno de estos grafos tiene unos script de verificación para comprobar que el algoritmo efectivamente funciona y calcula la matriz de distancias correctamente.

4.7.2 Algoritmos implementados

Las siguientes implementaciones en python, son los algoritmos trabajados en este proyecto para resolver problemas de SSSP, APSP y APSP incremental. Se encuentran en el enlace `https://github.com/apspgraphincremental/graph/tree/master/algorithms`.

- `dijkstra.Dijkstra`: Algoritmo de Dijkstra, necesita un nodo fuente y el grafo para retornar la la lista de distancia de dicho nodo fuente.
- `dijkstra.DijkstraQQ`: Algoritmo de Dijkstra usando cola de prioridad, necesita un nodo fuente y el grafo para retornar la la lista de distancia de dicho nodo fuente.
- `dijkstra.Dijkstra_apsp`: Algoritmo de Dijkstra para el problema de APSP de grafos. Recibe un grafo y devuelve toda la matriz de distancias.
- `floyd_warshall.Floyd_Warshall`: Algoritmo de Floyd Warshall para APSP. Recibe un grafo y devuelve la matriz de distancias.

Los siguientes algoritmos resuelven el problema del APSP incremental con base a una matriz de distancias.

- `eg.Even_Gazit`: Algoritmo de Even Gazit para APSP incremental. Recibe un grafo con la actualización y una matriz de distancias. Devuelve la nueva matriz de distancias.
- `rr.Dijkstra_Truncated`: Algoritmo de Ramalingam and Reps para SSSP incremental. Recibe un grafo con la actualización y una lista de distancias del nodo fuente. Devuelve la nueva lista de distancias.
- `rr.Bfs_Truncated`: Algoritmo de Ramalingam and Reps para APSP incremental. Recibe un grafo, un nodo fuente y una matriz de distancias. Devuelve la nueva matriz de distancias. Solo actualiza los nodos ascendentes de la actualización, no los descendientes.
- `rr.Find_Affected_Sources`: Algoritmo de Ramalingam and Reps que devuelve los nodos afectados (descendientes) después de una actualización.
- `rr.Bfs_Truncated_With_Sources`: Algoritmo de Ramalingam and Reps para APSP incremental. Recibe un grafo con la actualización y la matriz de distancias antes de la actualización. Utiliza el `Bfs_Truncated` por cada nodo resultante de `Find_Affected_Sources`
- `quinca.Quinca`: Algoritmo de Slobbe, Arie and Bergamini. Recibe un grafo con la actualización y la matriz de distancias antes de la actualización. Devuelve la nueva matriz de distancias.

- `forest.Forest`: Algoritmo de Alshammari y Muteb. Recibe un grafo y una lista de distancias del nodo que tuvo la actualización. Devuelve una nueva lista con las distancias a los nodos.
- `forest.Forest_apsp`: Algoritmo de Alshammari y Muteb para el problema de APSP incremental. Hace un recorrido para cada nodo usando el algoritmo `Forest`
- `abm.ABM_Update`: Algoritmo de ABM, recibe un grafo y la matriz de distancias para resolver. Devuelve otra matriz de distancias

El algoritmo 5 es un ejemplo común del uso de los algoritmos incrementales. En las lineas 8, 9 y 10 se definen los nodos fuentes, nodos destino y los pesos del grafos para obtener los arcos (Ejemplo: El arco `(1, 0)` y con peso 3 se encuentra en el primer item de los arrays de `sources`, `targets` y `weights`), estos pasan a ser importados en la linea 12 para generar el grafo en el constructor de la clase.

En la linea 13 calculamos la matriz de distancias usando el algoritmo para grafos estádicos Floyd Warshall, almacenamos el resultado en la variable. En la linea 15 insertamos un arco en el grafo, desde aquí nuestro grafo será un grafo dinámico incremental. En la linea 16 usamos el algoritmo ABM para calcular la nueva matriz de distancia basado en la matriz de distancia antes de la actualización, se utiliza el comando `matriz_distance.copy()` para generar una copia de la matriz y poder ser usado por los algoritmos QUINCA y RR en las lineas 21 y 22 respectivamente.

Las linas 24 y 25 se utiliza los comandos de testing de `numpy` para validar las matrices resultantes de cada algoritmo, devolvera una excepción si las matrices son diferentes, en caso de que sean iguales devolverá un `None`. Esta sección de código no se utiliza en las experimentaciones, se utiliza para validar que la matriz resultante de cada algoritmo es efectivamente la correcta. Para probarlo se corre el comando `pytest scripts/algoritmo_5_doc.py` para ejecutarlo. El resultado se aprecia en el comentario del algoritmo entre las lineas 28 y 44.

Algoritmo 5 Crear un grafo aleatorio, realizar una inserción aleatoria y calcular con ABM

```
import numpy as np
from graph.Graph import Graph
from algorithms.abm import ABM_Update
from algorithms.floyd_warshall import Floyd_Warshall
from algorithms.quinca import Quinca
from algorithms.rr import Bfs_Truncated_With_Sources

sources = [1, 2, 3, 0, 2, 3]
targets = [0, 0, 0, 4, 1, 1]
weights = [3, 2, 4, 1, 3, 2]

graph = Graph(sources, targets, weights)
matriz_original = np.array(Floyd_Warshall(graph))

graph.insert_worst_edge()
matriz_actualizada_abm = ABM_Update(graph, matriz_original.copy())

print("Matriz original: \n", matriz_original)
print("\nMatriz actualizada ABM: \n", matriz_actualizada_abm)

matriz_quinca = Quinca(graph, matriz_original.copy())
matriz_rr = Bfs_Truncated_With_Sources(graph, matriz_original.copy())

result_1 = np.testing.assert_array_equal(matriz_actualizada_abm, matriz_quinca)
result_2 = np.testing.assert_array_equal(matriz_actualizada_abm, matriz_rr)
print("\nErrores en las comparaciones: ", result_1, result_2)

"""
Matriz original:
[[ 0. inf inf inf  1.]
 [ 3.  0. inf inf  4.]
 [ 2.  3.  0. inf  3.]
 [ 4.  2. inf  0.  5.]
 [inf inf inf inf  0.]]

Matriz actualizada ABM:
[[ 0.  5.  2. inf  1.]
 [ 3.  0.  5. inf  4.]
 [ 2.  3.  0. inf  3.]
 [ 4.  2.  6.  0.  5.]
 [ 3.  4.  1. inf  0.]]

Errores en las comparaciones:  None None
"""
```

5 EXPERIMENTOS

5.1 Experimentos con Grafos sintéticos

5.1.1 Escenario de laboratorio

Se implementó un laboratorio específico para estos experimentos, el cual consiste en generar varios grafos de manera aleatoria con ciertas especificaciones y siguiendo un orden.

Parámetros

- `type_incremental`: Tipo de actualización incremental (Ej: Inserción aleatoria de un arco).
- `num_nodes`: Número de nodos que tendrá cada grafo.
- `density`: Densidad del grafo (por medio de una probabilidad de existir un arco entre dos nodos).
- `num_graphs`: El número de grafos que se van a generar.
- `epochs`: El número de intentos que un algoritmo va a ejecutar su solución en ese grafo.

Pasos

1. Definir todos los parámetros del laboratorio.
2. Crear el grafo con los parámetros `num_nodes` y `density`
3. Calcular la matriz de distancias (antes de la actualización)
4. Realizar la actualización incremental dado por el parámetro `type_incremental`
5. Exportar el grafo
6. Correr cada algoritmo el número de veces dato por `epochs` y almacenarlos en memoria
7. Repetir el paso 1 hasta tener el número de grafos dado por `num_graphs`
8. Exportar resultados de los tiempos

Para el laboratorio empírico se generaron un conjunto de grafos sintéticos. Cada grafo $G(n, p)$, consta de n nodos y arcos agregados al azar entre todos los posibles $n \times n$ pares con probabilidad $0 < p \leq 1$. El peso de los arcos también se selecciona aleatoriamente entre $2 \leq w \leq 10$. Probamos $n \in \{100, 1000\}$ para tener en cuenta los grafos de tamaño pequeño y mediano, con densidades $p \in \{0.01, \ldots 0.05, 0.1, \ldots 0.5\}$; los grafos son dirigidos. Para cada combinación (n, p), se generaron 30 grafos aleatorios diferentes; se probaron los algoritmos y se promediaron los resultados en estas instancias para evitar sesgos que surjan en un solo grafo aleatorio.

En resumen, se utilizaron 600 grafos en los experimentos empíricos. Esta colección de conjuntos de datos de grafos está disponible en `https://github.com/apspgraphincremental/graph-generated`.

El escenario **inserción de arco** describe dos nodos alteatorios no relacionados a los cuales se le añade un arco, esto genera que las distancias de algunos nodos se disminuyan y al realizar el recorrido sea más rápido, por ende se trata de una actualización incremental; por su parte el escenario **disminución de arco** elige un arco aleatorio del grafo y disminuye su peso, este a su vez también hace referencia a una actualización incremental.

Realizamos experimentos centrados en los tiempos de ejecución de una actualización de un solo arco en la colección de grafos sintéticos generados. Comparamos el algoritmo ABM descrito anteriormente con los siguientes algoritmos: EG (Even and Gazit, 1985), QUINCA (Algoritmo propuesto), RR (Ramalingam and Reps, 1996) y MA (Alshammari and Rezgui, 2020), que fueron elegidos de acuerdo con nuestra revisión de literatura de la Sección 3.3, como los más relevantes para el problema dinámico incremental de distancias más cortas de todos los pares.

Definimos dos conjuntos de experimentos: **inserción de arco**, donde se inserta un nuevo arco entre dos nodos elegidos al azar (no conectados), y **disminución de arco**, donde el peso de un arco (existente) elegido al azar se reduce a $w = 1$. Los experimentos se ejecutaron en una CPU Intel Core i5-8250U, 1,60 GHz con 1 núcleo y 16 GB de RAM, en un servidor Ubuntu 20.04 LTS de 64 bits.

Teniendo en cuenta lo presentado en el desarrollo del escenario de laboratorio, se eligieron los tres algoritmos más rápidos de cada experimento para comparar el rendimiento con el algoritmo postulado (ABM).

El resultado de este laborario será un dataframe con los tiempos de cada ejecución de un algoritmo sobre el grafo. Como se muestra en la tabla 2.

Algoritmo	Tiempo (segundos)	Nodos	Arcos	Densidad	Tipo
abm	1.0218	100	992	0.1	decrease_edge
abm	0.9765	100	992	0.1	decrease_edge
abm	0.9424	100	992	0.1	decrease_edge
abm	0.9574	100	992	0.1	decrease_edge
abm	0.9469	100	992	0.1	decrease_edge
abm	0.9462	100	992	0.1	decrease_edge
abm	0.9469	100	992	0.1	decrease_edge
abm	0.9431	100	992	0.1	decrease_edge
abm	0.9462	100	992	0.1	decrease_edge
abm	0.9424	100	992	0.1	decrease_edge
rr-bfs-truncated	1.8661	100	992	0.1	decrease_edge
rr-bfs-truncated	1.9028	100	992	0.1	decrease_edge
rr-bfs-truncated	1.8520	100	992	0.1	decrease_edge
rr-bfs-truncated	1.8711	100	992	0.1	decrease_edge
rr-bfs-truncated	1.8491	100	992	0.1	decrease_edge
rr-bfs-truncated	1.8711	100	992	0.1	decrease_edge
...	...	...	...	...	...

Tabla 2: Ejemplo de resultados del laboratorio para grafos sintéticos creados aleatoriamente.

5.1.2 Discusión

En el experimento de **inserción de arco aleatorio** en grafos con nodos más pequeños (Figura 13), notamos que para grafos de muy baja densidad, ABM presenta un rendimiento comparable, muy similar con QUINCA y mucho mejor que RR y MA, mientras que en grafos de mayor densidad ABM presenta una diferencia notoria al ejecutarse más rápido que los otros tres algoritmos. Una observación interesante es que para los grafos de menor densidad ABM mantiene aproximadamente el mismo rendimiento promedio de alrededor de 1.3ms pero luego reduce sus tiempos de ejecución a casi un tercio en el grafo de mayor densidad (particularmente en $p = 0.4, 0.5$). Por el contrario, QUINCA mantiene aproximadamente un rendimiento similar dentro de todas las densidades del grafo. Los resultados donde la densidad es 0.01 no coinciden del todo bien con el resto de densidades (el tiempo es mucho menor), sobre todo comparado con los comportamientos de los experimentos con 1000 nodos. Esto es debido a que el grafo solo posee un solo edge en los grafos de 100 nodos.

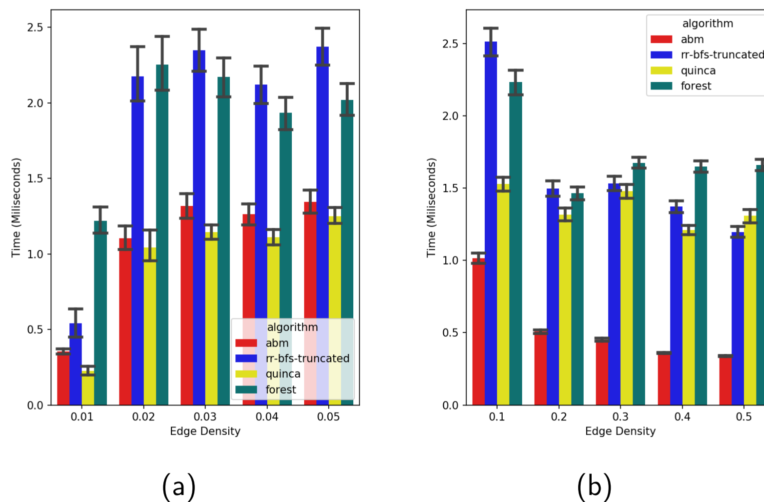


Figura 13: Resultados para $n = 100$ y $p \in \{0.01, \ldots, 0.5\}$ para inserción de arco aleatorio. Baja densidad (a), alta densidad (b).

Por otro lado, en el experimento de **disminución de arco alateatorio** (Figura 14), se observó que para los grafos de baja densidad, nuevamente ABM funciona comparable con QUINCA y mejor que RR y MA. En los grafos de mayor densidad, ABM también obtuvo un desempeño notorio al registrar tiempos más pequeños que los otros algoritmos. Los resultados donde la densidad es 0.01 es mucho menor que las demás densidades puede ser debido a que el grafo solo posee un solo edge en los grafos de 100 nodos.

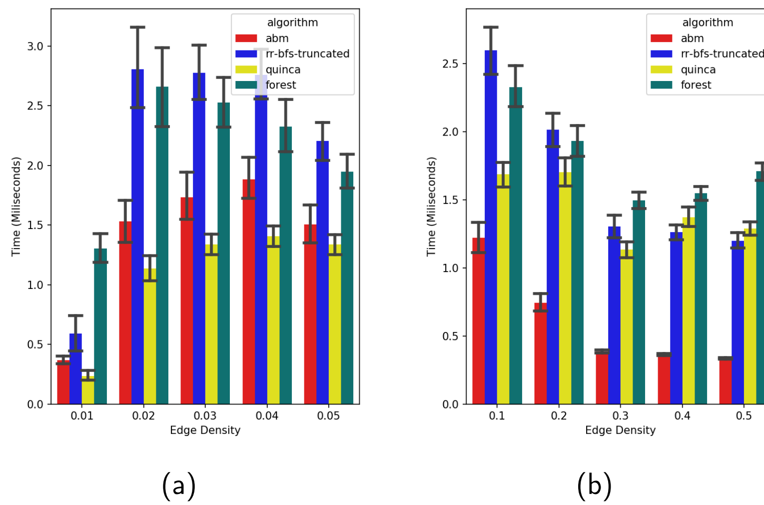


Figura 14: Resultados para $n = 100$ y $p \in \{0.01, \ldots, 0.5\}$. para disminución de arco aleatorio. Baja densidad (a), alta densidad (b).

En los casos de peor escenario para grafos (véase Figura 15) se presenta un comportamiento bastante similar al escenario aleatorio, sin embargo en el experimento de **inserción de arco** para grafos de baja densidad, ABM presenta un rendimiento deficiente con relación a QUINCA y MA, en cambio funciona comparable con RR; para grafos de mayor densidad, ABM tiene un tiempo de ejecución más rápido que los otros algoritmos presentados. Igual que en los escenarios anteriores, los resultados con densidad de 0.01 es menor que las otras densidades.

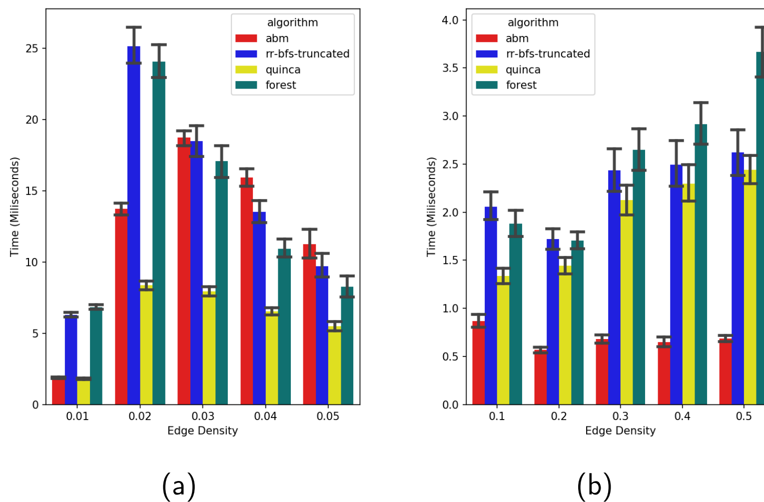
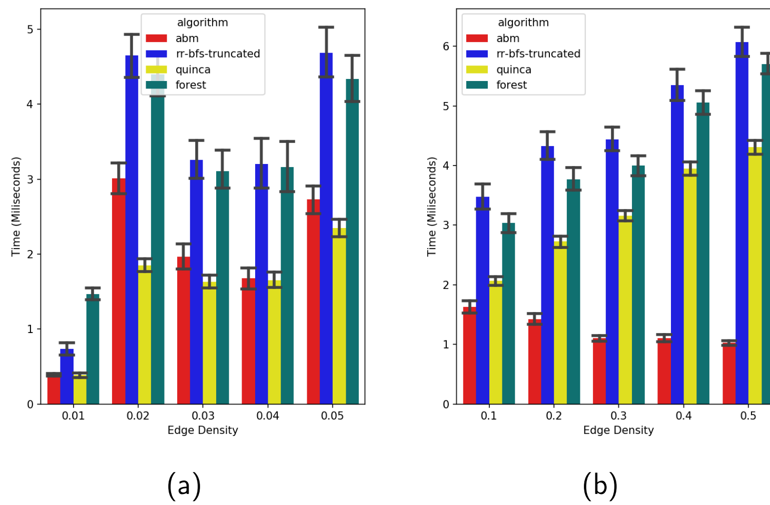


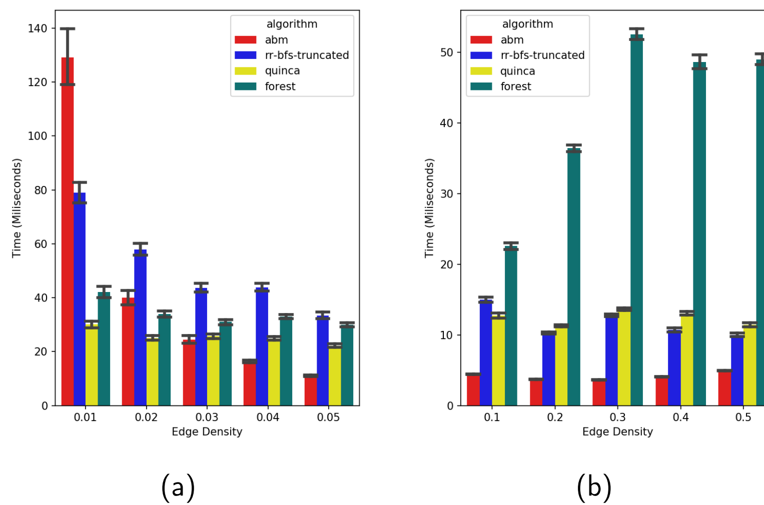
Figura 15: Resultados para $n = 100$ y $p \in \{0.01, \ldots, 0.5\}$ para inserción de arco en el peor escenario. Baja densidad (a), alta densidad (b).



(a) (b)

Figura 16: Resultados para $n = 100$ y $p \in \{0.01, \ldots, 0.5\}$ para disminución de arco en el peor escenario. Disminución de arco peor escenario: Baja densidad (a), alta densidad (b).

En los experimentos de **disminución de arco** (véase Figura 16), se observa que para los grafos de baja densidad, ABM funciona comparable con QUINCA y presenta mejor rendimiento que RR y MA. En los grafos de alta densidad, ABM también obtuvo tiempos de ejecución más pequeños que los otros algoritmos. Se repite el mismo resultado para las densidades de 0.01.



(a) (b)

Figura 17: Resultados para $n = 1000$ y $p \in \{0.01, \ldots, 0.5\}$. para inserción de arco aleatorio: Baja densidad (a), alta densidad (b).

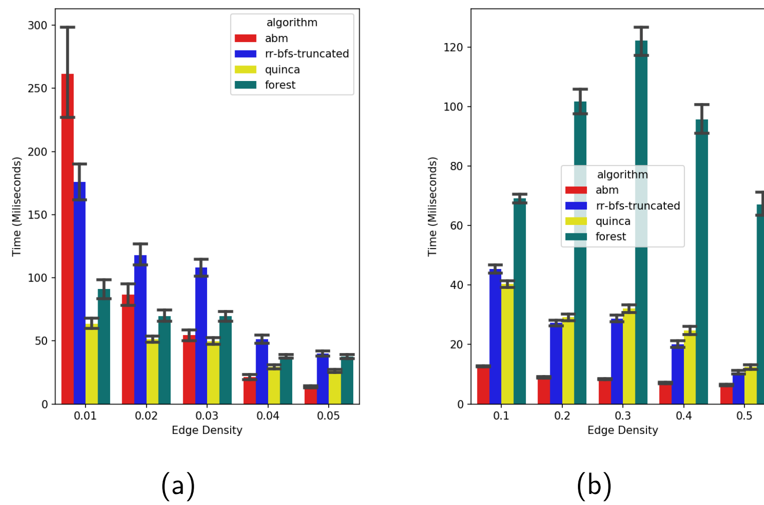


Figura 18: Resultados para $n = 1000$ y $p \in \{0.01, \ldots, 0.5\}$. para disminución de arco. Baja densidad (a), alta densidad (b).

Por último, se presentan los resultados obtenidos para grafos más grandes como se observa en la Figura 17. De acuerdo al experimento de **inserción de arco aleatorio** para densidades bajas, ABM se comporta de manera comparable a QUINCA, MA y de forma similar a RR, excepto en $p = 0.01$ donde toma más tiempo. Para densidades mayores, ABM resulta ser más rápido que los otros algoritmos, incluyendo a QUINCA. El caso del experimento **diminución de arco** (Figura 18) se sigue un patrón similar al del escenario **inserción de arco**.

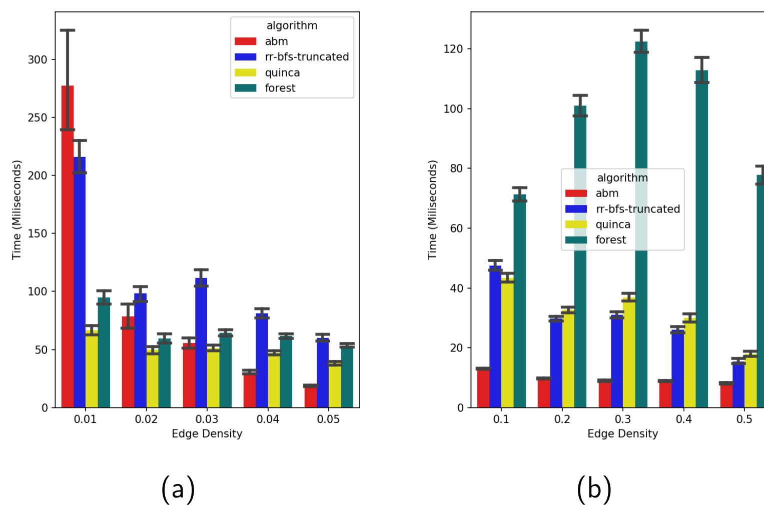


Figura 19: Resultados para $n = 1000$ y $p \in \{0.01, \ldots, 0.5\}$. para inserción de arco en el peor escenario: Baja densidad (a), alta densidad (b).

Finalizamos con los experimentos presentados en la Figura 19 en casos de peor escenario para grafos con nodos grandes. Para el experimento **incersión de arco** podemos apreciar un comportamiento funcional bastante similar al descrito por el escenario aleatorio de $n = 1000$ nodos; no siendo el mismo caso para los experimentos de **disminución de arco** (Figura 20), ya que se puede observar que para grafos de baja densidad, ABM tiene un tiempo de ejecución comparativamente más alto que los demás algoritmos, aún así para grafos de alta densidad, ABM presentó tiempos de ejecución más pequeños que los mismos.

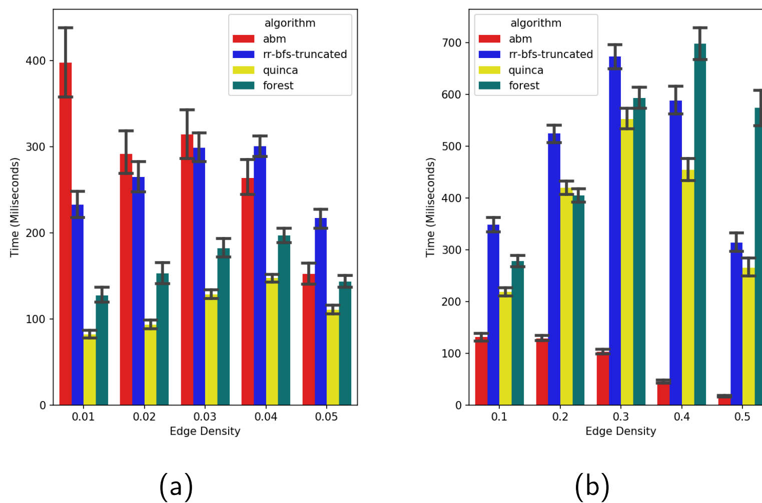


Figura 20: Resultados para $n = 1000$ y $p \in \{0.01, \ldots, 0.5\}$. para disminución de arco en el peor escenario: Baja densidad (a), alta densidad (b).

5.2 Experimentos con Datasets reales

5.2.1 Escenario de laboratorio

Se implementó un laboratorio particular para estos tipos de experimentos. El laboratorio consiste en los siguientes pasos:

1. Cargar los datos del dataset
2. Se realiza la actualización de inserción de arco y decreción de arco
3. Crear la matriz de distancias y exportarla. Se realiza por medio de Dijistra APSP y el tiempo de ejecución también se exporta a los resultados.
4. Se corren los algoritmos (usando el mismo módulo del laboratorio anterior)
5. Exportar resultados de los tiempos

Para nuestros experimentos se consideraron un conjunto de grafos del mundo real, estos fueron extraídos de SNAP (Leskovec and Krevl, 2014) y NETWORK REPOSITORY (Rossi and Ahmed, 2015). En SNAP podemos encontrar una compilación de información para varios grafos, mientras que NETWORK REPOSITORY provee mayor amplitud de datos específicos aunque sólo para algunos grafos.

Los resultados fueron realizados para experimentar con actualizaciones incrementales de inserciones y disminución de arco en datasets reales, dentro de ellos comparamos el algoritmo propuesto ABM con QUINCA, RR, MA y EG, utilizando las aceleraciones con respecto al tiempo estático.

La información contenida en las tablas que se presentan a continuación especifican los nodos del grafo, el arco entre cada nodo. La densidad del grafo, la cual está definida como la proporción de aristas que posee el grafo y está expresado matemáticamente como: $\frac{|A|}{|N*N|}$, donde $|A|$ es el número de arcos y $|N|$ es el número de nodos.

La implementación del laboratorio también se encuentra desponible en un repositorio de código público de github en el módulo `lab_real` `https://github.com/apspgraphincremental/graph/tree/master/lab_real`.

5.2.2 Discusión

En esta sección nos concentraremos particularmente en los resultados obtenidos para ABM y QUINCA.

De acuerdo a los datos reflejados por la Tabla 3 para los resultados de **inserción de arco aleatorio**, se puede notar que los tiempos en segundos para ABM y QUINCA son muy similares; en este caso QUINCA presenta mejores tiempos para grafos pequeños a densidades bajas, mientras que ABM presenta mejores tiempos para grafos grandes (muchos nodos) a densidades muy pequeñas.

Los resultados para los experimentos de **disminución de arco aleatorio** reflejados en la Tabla 4 evidencian un comportamiento similar al escenario anterior, donde los tiempos en ABM y QUINCA son comparables; QUINCA presenta un mejor desempeño para grafos pequeños a bajas densidades, por su parte ABM presenta mejores resultados en tiempo de ejecución para grafos grandes a densidades muy pequeñas.

Datasets	Nodos	Arcos	Estático (s)	ABM	QUINCA	RR	MA	EG
facebook_combined	4.039	88.234	11 948 515	114 $\pm$ 1.2	102 $\pm$ 0.7	110 $\pm$ 1.9	145 $\pm$ 0.8	14 515 $\pm$ 4.6
polblogs	1.490	19.025	665 242	10 $\pm$ 2.1	25 $\pm$ 1.1	133 $\pm$ 1.7	25 $\pm$ 1.2	2 035 $\pm$ 5.0
subelj_jung_j	6.120	128.706	35 706 819	249 $\pm$ 0.9	240 $\pm$ 2.8	233 $\pm$ 1.3	338 $\pm$ 2.3	33 904 $\pm$ 36.6
wiki_vote	7.115	103.689	74 618 372	447 $\pm$ 2.8	421 $\pm$ 2.6	418 $\pm$ 2.7	524 $\pm$ 3.1	61 523 $\pm$ 36.2
elec	7.118	103.675	46 119 663	332 $\pm$ 3.0	319 $\pm$ 2.0	312 $\pm$ 2.3	405 $\pm$ 2.9	45 143 $\pm$ 24.8
p2p_gnutella09	8.114	26.013	69 108 428	433 $\pm$ 3.2	440 $\pm$ 3.7	1 170 $\pm$ 3.9	498 $\pm$ 3.4	58 741 $\pm$ 22.7
p2p_gnutella04	10.876	39.994	165 625 503	768 $\pm$ 5.2	818 $\pm$ 5.8	2 687 $\pm$ 26.5	858 $\pm$ 5.5	106 572 $\pm$ 65.6
dpbl_cite	12.591	49.743	273 230 848	1 018 $\pm$ 6.9	975 $\pm$ 7.0	969 $\pm$ 4.3	1 141 $\pm$ 5.3	142 670 $\pm$ 213.4

Tabla 3: Resultados de conjunto de datos reales para inserción de arco aleatorio. La columna **Estático (s)** indica el tiempo en segundos de un algoritmo estático. Las últimas 5 columnas se muestra el tiempo en segundos de cada algoritmo.

Datasets	Nodos	Arcos	Estático (s)	ABM	QUINCA	RR	MA	EG
facebook_combined	4.039	88.234	12 472 753	114 $\pm$ 0.9	100 $\pm$ 0.7	101 $\pm$ 0.9	149 $\pm$ 2.0	14 519 $\pm$ 48.57
polblogs	1.490	19.025	691 526	9.33 $\pm$ 1.2	9.27 $\pm$ 1.1	26.5 $\pm$ 1.2	25.7 $\pm$ 1.1	2 045 $\pm$ 7.2
subelj_jung_j	6.120	128.706	36 126 861	249.7 $\pm$ 2.1	229.7 $\pm$ 2.1	229.3 $\pm$ 2.2	330.9 $\pm$ 2.5	33 893 $\pm$ 55.8
wiki_vote	7.115	103.689	74 522 247	622.9 $\pm$ 3.1	491.3 $\pm$ 57.8	507.8 $\pm$ 2.8	805.9 $\pm$ 4.8	61 543 $\pm$ 51.9
elec	7.118	103.675	46 089 479	614.0 $\pm$ 2.9	420.0 $\pm$ 8.7	569.5 $\pm$ 3.3	1 152.3 $\pm$ 6.2	45 154 $\pm$ 19.4
p2p_gnutella09	8.114	26.013	68 742 560	437.2 $\pm$ 3.4	470.5 $\pm$ 3.6	1 856 $\pm$ 5.8	504.3 $\pm$ 3.4	58 711 $\pm$ 59.9
p2p_gnutella04	10.876	39.994	165 618 290	767.3 $\pm$ 5.1	871.8 $\pm$ 6.9	3 814 $\pm$ 9.0	858.5 $\pm$ 4.0	105 544.1 $\pm$ 50.9
dpbl_cite	12.591	49.743	272 159 999	1 023.6 $\pm$ 4.7	968.2 $\pm$ 5.4	972.9 $\pm$ 4.2	1 119.6 $\pm$ 5.9	142 753 $\pm$ 227.3

Tabla 4: Resultados de conjunto de datos reales para disminución de peso en un arco aleatorio. Las últimas 5 columnas se muestran los tiempos en segundos de cada algoritmo.

6 CONCLUSIONES Y TRABAJOS FUTUROS

En este trabajo se ha propuesto y evaluado un método simple pero efectivo para distancias más cortas incrementales de todos los pares que realiza una visita previa a todos los pares de nodos, cuya longitud de ruta se ve afectada cuando un nuevo atajo acorta la distancia entre dos nodos arbitrarios. El nuevo algoritmo funciona en el espacio $O(n^2)$ generando un tiempo de consulta óptimo. Su costo de actualización en el peor de los casos también es de $O(n^2)$, aunque el rendimiento empírico promedio indica que es comparable con los métodos de última generación aplicados a grafos con conectividad de baja densidad, se ejecuta más rápido que esos otros algoritmos para grafos de mayor densidad. Por esta razón, llegamos a la conclusión de que podría ser de interés práctico para muchas de las aplicaciones dinámicas de distancia más corta de la actualidad.

Aunque los resultados fueron satisfactorio para la tesis propuesta, se tuvieron varias limitaciones como los recursos de cómputo. Debido a correr grandes cantidades de grafos, varias veces cada uno y con diferentes algoritmos, los recursos de tiempo fueron prolongandose. Por tal motiva se consideran como trabajo futuros el desarrollo de algoritmos que trabajen con paralelismo para mejorar estos tiempos considerablemente, este tipo de procesamiento abre la posibilidad de abarcar otros problemas de la teoría de grafos, como medidades de centralidad, cercanía, búsqueda, etc. Durante el proyecto se trabajó con una plataforma para trabajar con grafos, implementada por los autores, pero se podría trabajar con otras herramientas como `NetworKit`[2] y también utilizar una arquitectura web para implementar un laboratorio escalable para correr las pruebas de grafos estáticos y datasets reales.

Con respecto a la proyección de trabajos futuros, se considera además extender el estudio a grafos de gran tamaño (10,000 o más nodos), incluidos conjuntos de datos del mundo real y abordar el problema de actualizar las rutas más cortas reales. Se pueden incluir además otras operaciones dinámicas relacionadas como la actualización decreciente, tales como eliminar arcos del grafo y eliminar nodos.

Recientemente nuevos estudios han comenzado a aplicar ideas similares a la propuesta del algoritmo ABM en grafos estáticos como son los resultados de mejora del algoritmo de Floy-Warshall (Toroslu, 2021). Este tipo de investigación abre la posibilidad de obtener un algoritmo que trabaje en diferentes escenarios, como calcular la matriz de distancias y predecesores en grafos dinámicos completos (fully dinamic) y en grafos estáticos. También se podrían abarcar otros problemas de la teoría de grafos como medidas de centralidad entre otros.

[2] `NetworKit` Es un framework de código abierto para análisis de redes a gran escala de alto rendimiento.

7 REFERENCIAS

Alshammari, M. and Rezgui, A. (2020). An all pairs shortest path algorithm for dynamic graphs. *Computer Science*, 15(1):347–365.

Cormen, T. H., Leiserson, C. E., Stein, R. L. R., and Clifford (2001). *Introduction to Algorithms*. Number 2. MIT Press, 3 edition.

Demetrescu, C. and Italiano, G. F. (2004). A new approach to dynamic all pairs shortest paths. *Proceedings of the thirty-fifth ACM symposium on Theory of computing - STOC '03*, (January):159.

Even, S. and Gazit, H. (1985). Updating distances in dynamic graphs. *Methods of Operations Research*, 49:371—-387.

Garey, M. R. and Johnson, D. S. (1979). *Computers and intractability : a guide to the theory of NP-completeness*. A Series of Books in the Mathematical Sciences. W.H. Freeman, San Francisco, Calif.

Grandjean, M. (2016). A social network analysis of Twitter: Mapping the digital humanities community. *Cogent Arts & Humanities*, 3(1):1171458.

Leskovec, J. and Krevl, A. (2014). SNAP Datasets : Stanford large network dataset collection.

Loubal, P. and Commission, B. A. T. S. (1967). *A Network Evaluation Procedure*. Bay Area Transportation Study Commission.

Python-Software-Foundation (2021). collections — container datatypes.

Ramalingam, G. and Reps, T. (1996). On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158(1-2):233–277.

Rossi, R. A. and Ahmed, N. K. (2015). The network data repository with interactive graph analytics and visualization. In *AAAI*.

Slobbe, A., Bergamini, E., and Meyerhenke, H. (2016). Faster Incremental All-pairs Shortest Paths. *Forschungsbericht*.

Thorup, M. (2004). Fully-Dynamic All-Pairs Shortest Paths: Faster and Allowing Negative Cycles. *Scandinavian Workshop on Algorithm Theory*, 3111:384–396.

Toroslu, I. H. (2021). Improving The Floyd-Warshall All Pairs Shortest Paths Algorithm. *Dept. of Computer Eng., METU, Ankara, Turkey*.

Trudeau, R. J. (1993). *Introduction to graph theory*. Dover Pub.

Verbel, A., Rodriguez, N., and Rojas–Galeano, S. (2020). A Simple Yet Effective Algorithm to Compute Incremental All-Pairs Shortest Distances. *Communications in Computer and Information Science*, 1274 CCIS:222–229.

8 ANEXOS

8.1 Algoritmo ABM Update Predecesores

Algoritmo 6 Implementación en Python de ABM para cálculo de caminos

```
import numpy as np
from collections import deque

# Affected Block Matriz = ABM Update
def ABM_Update_PRED(graph, dist, pred):
    u, v, c_uv = graph.last_edge_updated

    if c_uv >= dist[u, v]:
        return dist, pred

    pred[u, v] = u
    affected_sources = deque()
    affected_targets = deque()

    for k in graph.nodes:
        if (c_uv + dist[v, k]) < dist[u, k]:
            affected_targets.append(k)
        if (dist[k, u] + c_uv) < dist[k, v]:
            affected_sources.append(k)

    for j in affected_targets:
        for i in affected_sources:
            sum = dist[i, u] + c_uv + dist[v, j]
            if sum < dist[i, j]:
                dist[i, j] = sum
                pred[i, j] = u if j == v else pred[v, j]

    return dist, pred
```

El Algoritmo 6 es la implementación en Python del algoritmo propuesto para este proyecto para para la matriz de predecesores (Explicada en la sección de Introducción). Es muy parecido al algoritmo anterior, usa las mismas estructuras de datos y las mismas secciones, igual que el mismo costo computacional. La diferencia es que ahora se va a actualizar la matriz de predecesores.

El algoritmo, además de recibir el grafo y la matriz de distancias, también recibe la matriz de predecesores. En la linea 11 se actualiza el predecesor debido a la inserción del nuevo arco.

En la linea 25 se realiza la actualización en la matriz de distancias (importante para próximas evaluaciones) y en la linea 26 se realiza la actualización de la matriz de predecesores. Como el algoritmo no cambió del todo se mantiene el mismo costo computacional de $O(n^2)$

8.2 Algoritmo Even Gazit

El algoritmo 7 es el algoritmo más sencillo de todos. Las lineas 2 a la linea 7 realiza lo mismo que el algoritmo de ABM, recibe los nodos y el peso del arco insertado, verifica que la matriz de distancia sea menor al nuevo peso del arco insertado y si no es así el algoritmo continua y actualiza la matriz de distancias.

La lineas 9 a la 13 realiza una combinación por cada nodo. Un nodo por todos los demás nodos. Verifica que efectivamente el peso de la actualización es menor en la linea 11 y 12 y actualiza en la linea 13. Para este algoritmo el costo computacional es $O(n^2)$ en todos los escenarios.

Algoritmo 7 Implementación en Python de Even & Gazit

```
1  def Even_Gazit(graph, dist):
2      u, v, w_uv = graph.last_edge_updated
3  
4      if dist[u, v] <= w_uv:
5          return dist
6  
7      dist[u, v] = w_uv
8  
9      for x in graph.nodes:
10         for y in graph.nodes:
11             sum = dist[x, u] + w_uv + dist[v, y]
12             if sum < dist[x, y]:
13                 dist[x, y] = sum
14 
15     return dist
```

8.3 Algoritmos de Ramalingam & Reps

Los estudios de Geetha Ramalingam y Thomas William Reps sobre grafos dinámicos en 1996 han servido de bases para muchos algoritmos de hoy. Los algoritmos 8, 9 y 10 son la implementación en Python.

Algoritmo 8 Implementación en Python de Ramalingam & Reps (Dijkstra Truncado)

```
1  import numpy as np
2  from collections import defaultdict
3  from collections import deque
4
5  def Dijkstra_Truncated(graph, dist_source):
6      u, v, w_uv = graph.last_edge_updated
7
8      if dist_source[v] <= dist_source[u] + w_uv:
9          return
10
11     dist_source[v] = dist_source[u] + w_uv
12
13     PQ = defaultdict(int)
14     S[v] = Find_Affected_Sources(graph, dist_source[v])
15
16     while len(PQ) > 0:
17         (y, weight) = min(PQ.items(), key=lambda x: x[1])
18         PQ.pop(y)
19
20         u_targets, u_weights = graph.get_targets_from_source(u, return_weight=True)
21
22         for z, weigth_uv in zip(u_targets, u_weights):
23             if dist_source[z] <= weight + weigth_uv:
24                 continue
25
26             new_weight = weight + weigth_uv
27             dist_source[z] = new_weight
28             PQ[z] = new_weight
29
30     return dist_source
```

Para este algoritmo se tiene un grafo G la cual sufre una actualización en uno de sus arcos, lo denotaremos como $(u, v, w'(u, v))$ donde u y v son nodos del grafo y $w'(u, v)$ es el nuevo valor ponderado de este arco. Tenemos un nodo s el cual queremos encontrar todas las distancias de este nodo fuente a todos los demás nodos, denotamos esta distancia como $d'(s, \cdot)$. Se definen todos los notos $t \in V$ tal que $d'(s, t) < d(s, t)$ como los nodos destino afectados. Las actualizaciones se hacen entonces: $d'(s, t) = d(s, u) + w'(u, v) + d(v, t)$. Así que lo único que se necesita para actualizar es la información en $d'(s, \cdot)$ para encontrar $d'(s, t)$ para los nodos afectados por $d(v, t)$. La implementación de este algoritmo en el proyecto se llama `Dijkstra_Truncated` escrito en el algortimo 8.

El algoritmo basicamente es como Dijkstra rotando en v, la diferencia es que solo los nodos afectados entran en la cola de prioridad. En el algoritmo, la cola de prioridad es la variable PQ. Podemos probar la velocidad de este algoritmo si, después de una actualización de arco, ejecutamos el dijkstra y el algoritmo SSSP de RR. El algoritmo fue editado de manera que no recibe cualquier nodo fuente, más bien el nodo fuente es aquel que se actualiza.

Para encontrar el APSP basta con correr el algoritmo anterior por cada nodo para obtener una solución rápida, sin embargo RR proponen un nuevo algoritmo para optimizar el tiempo de respuesta. Si se conocen los nodos destino que posee el nuevo nodo fuente por la actualización, entonces estos nodos destino se agregan como fuente en otro cola. Para esto se utiliza un código BFS (Breadth- First Search, algoritmo para búsqueda en árboles) en Dijkstra. Se puede ver en el algoritmo 9 en la función llamada `Bfs_Truncated`.

Algoritmo 9 Implementación en Python de Ramalingam & Reps (BFS Truncado)

```
1  def Bfs_Truncated(graph, source, dist):
2      u, v, w_uv = graph.last_edge_updated
3      if dist[source, v] <= dist[source, u] + w_uv:
4          return dist
5
6      vis = [False for i in graph.nodes]
7
8      dist[source, v] = dist[source, v] + w_uv
9
10     Q = deque([v])
11     vis[v] = True
12
13     while len(Q) > 0:
14     y = Q.popleft()
15     dist[source, y] = dist[source, u] + w_uv + dist[v, y]
16
17     for z in graph.get_targets_from_source(y):
18         if not vis[z] and dist[source, z] > dist[source, u] + w_uv + dist[v, z]:
19             vis[z] = True
20             Q.append(z)
21
22     return dist
```

Algoritmo 10 Implementación en Python de Ramalingam & Reps (BFS Truncado con fuentes)

```
import numpy as np
from collections import defaultdict
from collections import deque

def Find_Affected_Sources(graph, dist):
    u, v, w_uv = graph.last_edge_updated
    sources_affected = deque([])

    if dist[u, v] <= w_uv:
        return sources_affected

    vis = [False for i in graph.nodes]

    Q = deque([v])
    vis[v] = True

    while len(Q) > 0:
        x = Q.popleft()

        # TODO it can be better?
        for z in graph.source[graph.target == x]:
            if not vis[z] and dist[z, v] > dist[z, u] + w_uv:
                vis[z] = True
                Q.append(z)
                sources_affected.append(z)

    return sources_affected

def Bfs_Truncated_With_Sources(graph, dist):
    for source in Find_Affected_Sources(graph, dist):
        dist = Bfs_Truncated(graph, source, dist)

    return dist
```

Para identificar ahora los nodos fuentes, RR implementa otro algoritmo para detectar que nodos son afectados. este algoritmo complementa el BFS truncado de manera que identifica que nodos son afectados, que llegan al nodo fuente de la actualización. La idea de estos algoritmos es completar el APSP de actualización de arco corriendo primero el algoritmo para detectar los nodos afectados y luego el BFS. Se encuentra implementado en el algoritmo 10.

8.4 Algoritmo QUINCA

Algoritmo 11 Implementación en Python de QUINCA

```
import numpy as np
from algorithms.rr import *
from collections import defaultdict
from collections import deque

def Quinca(graph, dist):
    u, v, w_uv = graph.last_edge_updated

    if dist[u, v] <= w_uv:
        return dist

    S = defaultdict(list)
    S[v] = Find_Affected_Sources(graph, dist)

    dist[u, v] = w_uv

    Q = deque([v])
    P = {v: v}
    vis = [False for i in graph.nodes]
    vis[v] = True

    while len(Q) > 0:
        y = Q.popleft()
        # update distances for source nodes
        for x in S[P[y]]:
            if dist[x, y] > dist[x, u] + w_uv + dist[v, y]:
                dist[x, y] = dist[x, u] + w_uv + dist[v, y]
                if y != v:
                    S[y].append(x)

        y_targets, y_weights = graph.get_targets_from_source(y, return_weight=True)

        for z, w_yz in zip(y_targets, y_weights):
            if not vis[z] and dist[u, z] > w_uv + dist[v, z] and \
                    dist[v, z] == dist[v, y] + w_yz:
                Q.append(z)
                vis[z] = True
                P[z] = y

    return dist
```

La idea del algoritmo de QUINCA es evitar el recálculo hecho por RR. En lugar de iniciar un BFS desde v para cada una de las fuentes afectadas, ejecutamos el BFS solo una vez para la fuente afectada u, actualizando al mismo tiempo también las distancias de las otras fuentes afectadas. El algoritmo 11 muestra la implementación en python para la actualización de APSP. Primero, identifican todas los nodos fuentes afectados del nodo v (línea 13).

Luego, básicamente ejecutan el algoritmo de BFS truncado (algoritmo 9) con el nodo fuente $s = u$, con algunas diferencias. Primero, al insertar un objetivo afectado w en Q, también se hace un seguimiento del predecesor y en (una de) las rutas más cortas desde v (línea 38). Entonces, no solo actualizamos la distancia de u a y, sino también las de cada nodo en $S(y)$ a y. Dado que el conjunto de objetivos afectados de y está contenido en el de z, podemos simplemente pasar por $S(z)$ (Línea 25) para averiguar cuáles de las $x \in S(z)$ también son fuentes afectadas para y $d(x, y) > d'(x, y)$. En este caso, se suma x a $S(y)$ (que será utilizado por los sucesores de y) y se establece $d(x, y)$ a $d'(x, y)$ (líneas 27-29). Luego, se recorren los arcos salientes de y para encontrar nuevos objetivos afectados para u (líneas 33 a 38).

8.5 Algoritmo de Muteb Abdelmounaam

La implementación en Python del algoritmo de Muteb y Abdelmounaam se encuentra descrito en el algoritmo 12. Se crea una función llamada `ma_apsp` para que el algoritmo pueda resolver problemas de APSP y no de SSSP como originalmente fue planteado. Continene 3 fases. La primera se encuentra entre las lineas 8 y 9 la cual valida si efectivamente la actualización es un cambio de tipo incremental. En la fase 2 (lineas 12 - 15) se establecen las variables a utilizar, entre ellas la cola de prioridad usando la librería `heapq`. La tercera face es muy parecido al Dikstra truncado de RR visto en el algortimo 8.

Algoritmo 12 Implementación en Python de Muteb & Abdelmounaam

```python
import numpy as np
import heapq

def MA(source, graph, t_dist):
    x, y, w_xy = graph.last_edge_updated

    # Phase 1
    if t_dist[y] < t_dist[x] + w_xy:
        return t_dist

    # Phase 2
    t_dist[y] = t_dist[x] + w_xy

    H = []
    heapq.heappush(H, (t_dist[y], y)) # H is a min-heap

    # Phase 3
    while len(H) > 0:
        (weight, u) = heapq.heappop(H)

        u_targets, u_weights = graph.get_targets_from_source(u, return_weight=True)

        for index, v in enumerate(u_targets):
            if t_dist[u] + u_weights[index] < t_dist[v]:
                t_dist[v] = t_dist[u] + u_weights[index]
                heapq.heappush(H, (t_dist[v], v))

    return t_dist


def ma_apsp(graph, dist):
    for source in graph.nodes:
        dist[source] = MA(source, graph, dist[source])

    return dist
```

8.6 Artículo WEA

A simple yet effective algorithm to compute incremental All-pairs Shortest Distances

Arturo Verbel[1], Nestor Rodriguez[2] and Sergio Rojas–Galeano[1]

[1] Universidad Distrital Francisco José de Caldas, Bogotá, Colombia.
averbel@correo.udistrital.edu.co
srojas@udistrital.edu.co
[2] Formerly at Universidad Distrital Francisco José de Caldas, Bogotá, Colombia.
nesterran@gmail.com

Abstract. Many activities of modern day living operate on the basis of graph structures that change over time (e.g. social networks, city traffic navigation, disease transmission paths). Hence, the problem of dynamically maintaining properties of such structures after modifying one of its edges (or links), specially for large-scale graphs, has received a great amount of attention in recent years. We address the particular case of updating all-pairs shortest distances upon *incremental* changes, i.e. recomputing shortest distances among all the nodes of a graph when a new or smaller shortcut between two nodes arises. We build upon the naive algorithm that visits all pairs of nodes comparing if the new shortcut shortens the distances, and propose a simple variation that instead chooses pairs of source and target nodes, only from the *affected* shortest paths. The new algorithm works with an optimal data structure, constant query time and worst-case $O(n^2)$ update cost, although our results on synthetic datasets hints at its practicality when compared with state-of-the-art approaches.

Key words: all-pairs shortest distances, dynamic incremental graphs

1 Introduction

Finding paths with shortest distances is a foundational problem in the analysis of graphs and has been extensively studied since the beginnings of computer science. In the case of graphs with static structure (i.e. fixed sets of nodes and edges) established algorithms such as Dijkstra's [8] and Floyd-Warshall's [11] are able to find effectively single–source and all–pairs shortest paths (known as SSSP and APSP respectively). Such algorithms are at the core of route planning technologies amply used in communication [17] and transportation networks [9, 6]. The growing size of these networks in recent years has motivated the proposal of more efficient strategies intended to cope with time limitations associated with such volumes, in particular, solving point-to-point shortest paths [15, 12, 5]. Usually in those scenarios the network structure is known and fixed, hence the aforementioned algorithms can be used to compute efficiently the distances from

scratch, with a cost $O(n(m + n \log n))$, where n is the number of nodes and m is the number of edges in the graph (or $O(nm)$ in unweighted graphs).

A different problem where the graph structure changes dynamically over time, either by adding, removing or updating edges or nodes, has brought in new attention from researchers. Many activities of modern day living can be casted within such setting. Take for example wireless ad–hoc data networks, where routing paths are continuously changing depending on the availability of connected nodes (users or vehicles) [16]. Likewise, graphs of interactions happening in digital social networks [14] and mobile telephone calls [19] rapidly change their structure and therefore shortest paths or other interesting graph properties such as reachability, centrality or diameters, need to be maintained dynamically [4]. Navigation systems must deal with real-time traffic conditions so as to provide shortest routes recommendations [2]. The current COVID-19 pandemic is providing evidence that tracing contact transmission networks to estimate the expose to the virus as a function of distance to confirmed cases, is one of the more effective public health policies to contain the disease spread [13].

In this paper we focus on shortest–path distances over all–pairs of nodes of a directed graph upon *incremental* changes, i.e. on re-computing shortest distances among all the nodes of a graph when a new or a smaller shortcut between two nodes is inserted. We revisit the naive approach that examines all pairs of nodes to verify if the new shortcut shortens the distances [10], a strategy that takes $\Theta(n^2)$ time. We propose a simple variation that instead of traversing all combinations of target/source pairs, firstly filters only those nodes involved in the shortest paths that resulted *affected* with the new insertion. Although the worst-case update cost remains as $O(n^2)$, empirical evidence hints at a better average performance compared to the naive version and also comparable to recently proposed algorithms that likewise use the idea of restricting updates to the *affected* paths [20].

Notation. Let $G = (V, E, \omega)$ be a graph consisting of a set V of vertices or nodes, a set E of connecting edges, and a function $\omega : E \rightarrow \mathbb{R}^+$ assigning costs to each edge; $|V| = n$ and $|E| = m$. Let $D \in \mathbb{R}^{n \times n}$ be the all-pairs shortest distances matrix, with entries $d(s, t) < \infty$ being the distance of the shortest path connecting source node $s \in V$ to target node $t \in V$; if such a path does not exist then $d(s, t) = \infty$. We consider only updates consisting of insertions of a new or a smaller edge in terms of ω, that is, $\omega'(u, v) < \omega(u, v)$ for any $u, v \in V$. When this happens we denote by $\mathcal{A}_S$ the set of source nodes from the affected shortest paths whose target node is v; likewise we denote by $\mathcal{A}_T$ the set of target nodes from the affected shortest paths whose source node is u.

Problem statement. Given $G = (V, E, \omega)$, $D = \{d(i, j)\}_{i,j=1}^n$ and $\omega'(u, v)$, find the resulting all–pairs shortest distances matrix $D' = \{d'(i, j)\}_{i,j=1}^n$ corresponding to $G' = (V, E', \omega')$.

2 Previous work

The naive procedure comparing distances pairwise among all nodes of the graph is an algorithm that runs in $\Theta(n^2)$, originally proposed by Even and Gazit in 1985 [10]. The problem has attracted much attention since then. An algorithm that constraints the function ω to a an integer positive domain bounded by a constant C was proposed by Ausiello et al. [3]; its amortised running time is $O(n \log n)$. Aftterwards, Ramalingam and Reps [18] proposed algorithms for incremental SSSP and APSP based on the idea of maintaining only the shortest paths *affected* by the edge update; basically they use a truncated version of Dijkstra's algorithm that traverses only the affected nodes, which in the worst-case is as costly as computing distances from scratch. Nonetheless, empirical studies have shown that their algorithms are well–performing in practice for real–life graphs [7].

More recently Slobbe, Bergamini and Meyerhenke [20] proposed novel SSSP and APSP algorithms for incremental edge also node insertions, both running in $O(n^2)$ worst–case time; their edge–update algorithms are improvements to Ramalingam and Reps algorithms, by simultaneously updating the distances from the affected source nodes, therefore avoiding repeating unnecessary truncated Dijkstra's executions. Despite their worst–case complexity, the authors show than empirically their algorithms achieve better performance than the baseline existing methods. Our study resembles this same approach, in that we also constraint the updating condition to the sets of affected sources and targets, although we focus on computing distances (not paths) and we use simpler algorithms and basic data structures. Our worst–case and empirical performance are nonetheless comparable with theirs. Lastly, a recent work reported by Alshammari et al.[1] also considers fully dynamic algorithms; their incremental algorithm maintains a forest of shortest path trees rooted in every node of the graph, that upon edge insertion updates distances in every tree using an adaptation of Dijkstra's algorithm; in the empirical study we conducted, our proposed algorithm also performs better in artificially generated graphs.

3 An incremental APSD algorithm

Broadly speaking, when a new edge with cost (length) $\omega'(u, v)$ is inserted (or replaced with a smaller length, if already exists), the algorithm checks if it is shorter than the current distance estimate $d(u, v)$ between these two nodes. If so, it updates this new distance for the edge $\overline{uv}$ and then checks which sub-paths are affected, from those starting in any other source node and ending at v, and from those starting at u and ending at any other target node in the graph. This is done with a linear-time loop that traverses every graph node k, recording those complying with the relaxing condition $(d(k, u) + \omega'(u, v)) < d(k, v)$ in a list of affected sources $\mathcal{A}_S$, and those complying with the relaxing condition $(\omega'(u, v) + d(v, k)) < d(u, k)$ in a list of affected targets $\mathcal{A}_T$.

Algorithm 1: `abm` $(D, \omega'(u,v))$

```
Input: All–pairs distance matrix D = {d(i,j)}_{i,j=1}^n, new decreased edge
       ω'(u,v)
Output: Updated matrix D
if ω'(u,v) < d(u,v) then
    A_S = A_T = ∅                    // Lists of affected sources and targets
    for k = 1...n do
        if (d(k,u) + ω'(u,v)) < d(k,v) then A_S = A_S ∪ k
        if (ω'(u,v) + d(v,k)) < d(u,k) then A_T = A_T ∪ k
    foreach s ∈ A_S do
        foreach t ∈ A_T do
            d(s,t) = min(d(s,t), d(s,u) + ω'(u,v) + d(v,t))
```

The described step can be regarded as a filter that determines the actual affected pairs of distances that need to be updated. Since we keep distances in a square matrix D, indexed by the nodes of the graph in no particular order, the lists $\mathcal{A}_S$, $\mathcal{A}_T$ define in fact a subset of index nodes corresponding to a *block* of the matrix, namely the block of entries of *affected* distances. Thus, we call this algorithm the affected block matrix update (`abm`). The algorithm finishes traversing all-pairs in the block, updating only entries with shorter distances.

The time complexity of the algorithm is dominated by the double nested loop that traverses the lists of affected sources and targets. The size of these lists depend on how many elements are inserted during execution of the preceding loop, which in turn traverses all n nodes, that is, $|\mathcal{A}_S| \leq n$ and $|\mathcal{A}_T| \leq n$; since the actual sizes resulting because of an arbitrary decrease update are not known in advanced, hence we estimate a worst-case scenario for time complexity of $O(n^2)$. In practice, however, this cost may be usually smaller, as our empirical experiments suggest (see next section). This is because the size of the resulting block that needs to be updated is commonly much smaller than n^2. Besides, since the only data structure the algorithm uses to operate is the all-pairs distance matrix, its space complexity is also $O(n^2)$ with an optimal $O(1)$ query time.

4 Empirical study

4.1 Synthetic graphs

We created synthetic random graphs $G(n,p)$, consisting of n nodes and edges added at random between all possible $n \times n$ pairs with probability $0 < p < 1$. The weight of the edges is also randomly selected between $2 \leq \omega(u,v) \leq 10$. We tried $n \in \{100, 1000\}$ so as to account for small- and medium-size graphs, with densities $p \in \{0.01, \ldots 0.05, 0.1, \ldots 0.5\}$; edges are directed and no self-connections were allowed. For each combination (n,p), 30 different random graphs were generated; the algorithms were tested and results averaged over these instances in order to avoid biases arising in a single random graph.

In summary, 600 graphs were used in the empirical experiments. This collection of graph datasets is available at `https://github.com/apspgraphincremental/graph-generated`.

4.2 Experiment design

We performed experiments focusing on running times of a single edge update on the collection of the generated synthetic graphs. We compared the `abm` algorithm described above with the following algorithms: `even-gazit` [10], `quinca` [20], `rr-bfs-truncated` [20], and `forest` [1], which were chosen according to our literature review of Section 2, as the most relevant for the dynamic incremental All-pairs Shortest Distances problem. We defined two sets of experiments: **insert**, where a new edge is inserted between two randomly chosen (unconnected) nodes, and **decrease**, where the weight of a randomly chosen (existing) edge is decreased to $\omega'(u, v) = 1$.

The experiments were run on a Intel Core i5-8250U CPU, 1.60 GHz using 1 core and 16 GB of RAM, on an Ubuntu 20.04 LTS 64bit Server.

4.3 Results

In this section we report plot bars of average runtimes with standard deviations as error intervals, for the experiments described earlier. The three faster algorithms in each experiment were chosen in order to compare their performance.

Figure 1 shows the results for graphs with $n = 100$ nodes. In the **insert** experiment, we notice that for very low density graphs ($0.01 < p \leq 0.05$, (a)), `abm` performs comparable with `quinca` and better than `rr-bfs-truncated` and `forest`, whereas in higher density graphs ($0.1 < p \leq 0.5$, (b)), `abm` runs faster than the other three algorithms. An interesting observation is that for the lower density graphs ($p = 0.02, \ldots, 0.05$), `abm` maintains roughly the same average performance of around 1.3ms but then it reduces its runtimes to almost a third in the higher density graphs (particularly in $p = 0.4, 0.5$). On the contrary, `quinca` maintains roughly a similar performance within all the graph densities.

On the other hand, in the **decrease** experiment with $n = 100$, we notice that for the low density graphs (c), again `abm` performs comparable with `quinca` and better than `rr-bfs-truncated` and `forest`. In the high density graphs (d), `abm` also obtained smaller times than the other algorithms.

Lastly, let us comment on the results shown in Figure 2 regarding larger graphs of 1000 nodes. Here, in the **insert** experiment with low densities (a), `abm` behaves comparable to `quinca` and `forest` except on $p = 0.01$ where it takes longer in average, similarly to `rr-bfs-truncated`. For larger densities (b), `abm` happens to be faster than the other algorithms, including `quinca`. The case of the **decrease** experiment (c-d) follows a similar pattern than the **insert** scenario.

5 Conclusion

We described a simple yet effective method for incremental all-pairs shortest distances that pre-visits all pairs of nodes whose path length is affected when

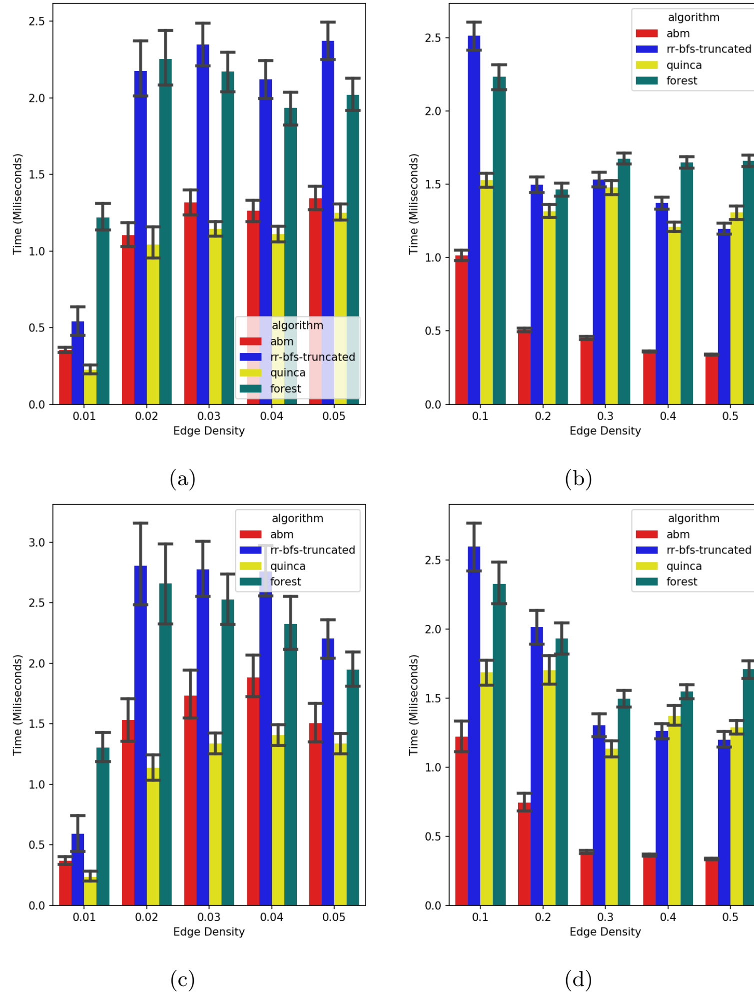


Fig. 1. Results for $n = 100$ and $p \in \{0.01, \ldots, 0.5\}$. **insert**: (a)-(b), **decrease**: (c)-(d).

a new shortcut shortens the distance between any arbitrary two nodes. The new algorithm works in $O(n^2)$ space yielding optimal query time. Its worst-case update cost is also $O(n^2)$, although average empirical performance indicates it is comparable with state-of-the-art methods applied to graphs with low-density connectivity, whilst running faster than those other algorithms for higher-density graphs. For this reason, we conclude it could be of practical interest for many of nowadays dynamic shortest distance applications.

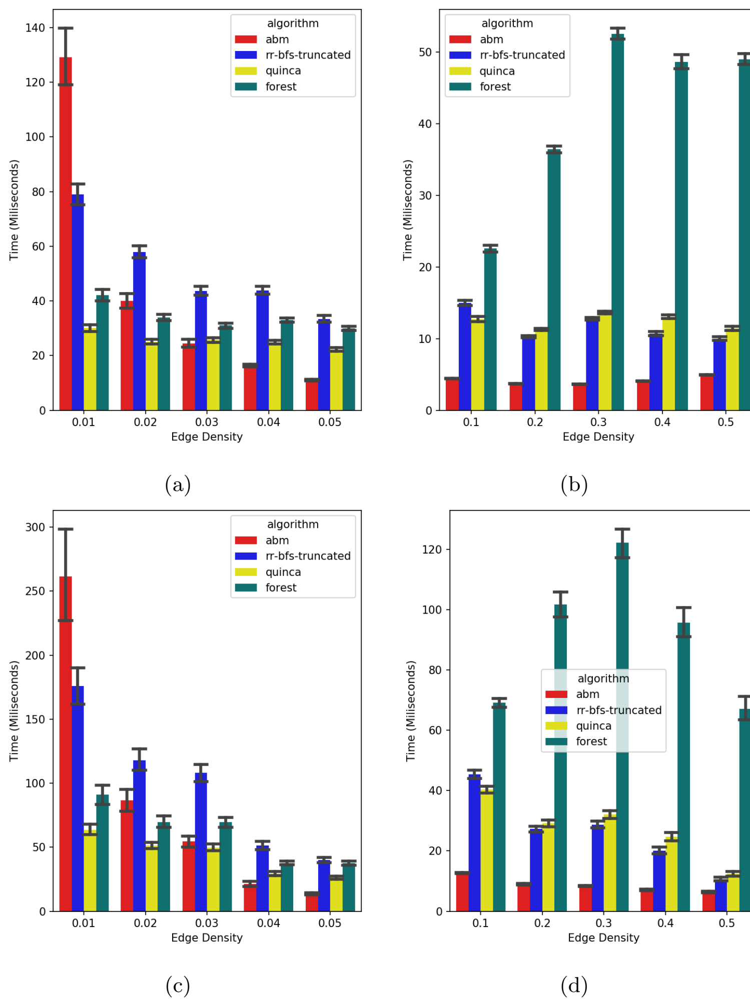


Fig. 2. Results for $n = 1000$ and $p \in \{0.01, \ldots, 0.5\}$. **insert**: (a)-(b), **decrease**: (c)-(d).

Regarding ideas for future work, we plan to extend our study to large-size graphs (10,000 or more nodes), including real-world data sets and to address the problem of updating the actual shortest paths, as well as other related dynamic operations, such as removing edges and adding or removing nodes.

References

1. Muteb Alshammari and Abdelmounaam Rezgui. An all pairs shortest path algorithm for dynamic graphs. *Computer Science*, 15(1):347–365, 2020.

2. Mostafa K Ardakani and Madjid Tavana. A decremental approach with the A* algorithm for speeding-up the optimization process in dynamic shortest path problems. *Measurement*, 60:299–307, 2015.
3. Giorgio Ausiello, Giuseppe F Italiano, Alberto Marchetti Spaccamela, and Umberto Nanni. Incremental algorithms for minimal length paths. *Journal of Algorithms*, 12(4):615–638, 1991.
4. Elisabetta Bergamini, Henning Meyerhenke, and Christian Staudt. Approximating betweenness centrality in large evolving networks. In *ALENEX*, pages 133–146. SIAM, 2015.
5. Daniel Delling, Andrew V Goldberg, Thomas Pajor, and Renato F Werneck. Customizable route planning in road networks. *Transportation Science*, 2015.
6. Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering route planning algorithms. In Jurgen et al., editor, *Algorithmics of Large and Complex Networks*, volume 5515 of *LNCS*. Springer, 2009.
7. Camil Demetrescu and Giuseppe F Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. *ACM Transactions on Algorithms (TALG)*, 2(4):578–601, 2006.
8. Edsger W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.
9. Christine E. Dunn and David Newton. Optimal routes in GIS and emergency planning applications. *Area*, 24(3):259–267, 1992.
10. Shimon Even and Hillel Gazit. Updating distances in dynamic graphs. *Methods of Operations Research*, 49:371—-387, 1985.
11. Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345+, June 1962.
12. Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Decremental single-source shortest paths on undirected graphs in near-linear total update time. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on*, pages 146–155. IEEE, 2014.
13. Matt J Keeling, T Deirdre Hollingsworth, and Jonathan M Read. The efficacy of contact tracing for the containment of the 2019 novel coronavirus (covid-19). *medRxiv*, 2020.
14. Sushant S Khopkar, Rakesh Nagi, Alexander G Nikolaev, and Vaibhav Bhembre. Efficient algorithms for incremental all pairs shortest paths, closeness and betweenness in social network analysis. *Social Network Analysis and Mining*, 4(1):1–20, 2014.
15. Andrei Lissovoi and Carsten Witt. Runtime analysis of ant colony optimization on dynamic shortest path problems. *Theoretical Computer Science*, 561:73–85, 2015.
16. Jianqi Liu, Jiafu Wan, Qinruo Wang, Pan Deng, Keliang Zhou, and Yupeng Qiao. A survey on position-based routing for vehicular ad hoc networks. *Telecommunication Systems*, 62:15–30, 2016.
17. John T. Moy. *OSPF: Anatomy of an Internet Routing Protocol*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.
18. Ganesan Ramalingam and Thomas Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21(2), 1996.
19. Rui Sarmento, Márcia Oliveira, Mário Cordeiro, Shazia Tabassum, and João Gama. Social network analysis in streaming call graphs. In *Big Data Analysis: New Algorithms for a New Society*, pages 239–261. Springer, 2016.
20. Arie Slobbe, Elisabetta Bergamini, and Henning Meyerhenke. Faster Incremental All-pairs Shortest Paths. Technical report, Karlsruhe Institute of Technology, Faculty of Informatics, 2016.