Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

Método de la ingeniería

Fase 1: Identificación del problema.

- **Contexto del problema:** La universidad Icesi, es una universidad privada sin ánimo de lucro ubicada en el suroccidente de Colombia en la ciudad de Cali, departamento del Valle del Cauca. Esta universidad cuenta con un área de aproximada de164 mil metros cuadrados que consta de instalaciones deportivas, edificios, zonas verdes, restaurantes y parqueaderos.
  Actualmente, las personas que frecuentan en la universidad son estudiantes, profesores y colaboradores de esta, algunos de estos estudiantes son microempresarios que venden comida a toda la comunidad universitaria. Para todos los mencionados anteriormente es necesario desplazarse por todo el campus universitario de manera eficiente ya que el aprendizaje activo (el cual es el modelo de enseñanza de la universidad Icesi) les quita mucho tiempo y necesitan movilizarse de un lugar del campus a otro de manera que no les quite tanto tiempo y puedan cumplir sus labores cada uno.

- **Problema:** La universidad universitaria quiere implementar un programa el cual permita a sus usuarios ver la ruta más cercana que hay de un lugar a otro en el campus, lo cual disminuiría el tiempo para llegar a su destino, además el programa debe tener una funcionalidad que permita al usuario saber cuál es el camino más corto que conecta a todas las áreas de la universidad ya que esto le permitirá al estudiante general otros ingresos por dar un tour a los estudiantes de primer semestre.

- **Requerimientos Funcionales**

  Requerimiento funcional 1: Encontrar el camino más corto desde un edificio a otro.
  Entradas: Edificio de llegada y edificio de salida.
  Salidas: Camino más corto entre los dos edificios.

  Requerimiento funcional 2: Conocer cuál es el camino más corto, el cual conecta todos los edificios de la universidad.
  Entradas: Edificio inicial.
  Salidas: Camino más corto que conecta todos los edificios de la universidad.

  Requerimiento funcional 3: Visualizar el camino más corto entre dos edificios.
  Entradas: Ninguna.
  Salidas: Visualización del camino más corto entre dos edificios.

  Requerimiento funcional 4: Visualizar el camino más corto que conecta a todos los edificios.
  Entradas: Ninguna.
  Salidas: Visualización del camino más corto entre dos edificios.

Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

Fase 2: Recopilación de Información.

A continuación, recopilamos los conceptos más relevantes y de interés de los componentes que influyen en el proyecto.

Referencias

Cormen. (02 de 04 de 2019). *Wikipedia*. Obtenido de https://es.wikipedia.org/wiki/B%C3%BAsqueda_en_anchura

Ecured. (24 de 04 de 2018). *Ecured*. Obtenido de Ecured: https://www.ecured.cu/Algoritmo_de_Prim

Ecured. (15 de 04 de 2019). *Ecured*. Obtenido de https://www.ecured.cu/Algoritmo_de_Kruskal

Gregorio. (17 de 05 de 2017). *dma*. Obtenido de dma: http://www.dma.fi.upm.es/personal/gregorio/grafos/web/dijkstra/

IAGraph. (29 de 07 de 2018). *IAGraph*. Obtenido de http://www.dma.fi.upm.es/personal/gregorio/grafos/web/iagraph/busqueda.html

IES, M. (12 de 02 de 2019). *matmaticasies*. Obtenido de https://matematicasies.com/Matriz-de-adyacencia-de-un-grafo

Merino, M. (12 de 05 de 2012). *Definicion.de*. Obtenido de Definicion.de: https://definicion.de/vertice/

Porto, J. P. (21 de 04 de 2012). *Definicion.de*. Obtenido de Definicion.de: https://definicion.de/grafos/

Reinhad, D. (07 de 03 de 1997). *Wikipedia*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/Arista_(teor%C3%ADa_de_grafos)

Wikipedia. (21 de Mayo de 2017). *Wikipedia*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/Algoritmo_de_Floyd-Warshall

Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

Grafo:

Un grafo es una representación gráfica de diversos puntos que se conocen como nodos o vértices, los cuales se encuentran unidos a través de líneas que reciben el nombre de aristas. Al analizar los grafos, los expertos logran conocer cómo se desarrollan las relaciones recíprocas entre aquellas unidades que mantienen algún tipo de interacción.

Los grafos pueden ser clasificarse de diversas maneras según sus características. Los grafos simples, en este sentido, son aquellos que surgen cuando una única arista logra unir dos vértices. Los grafos complejos, en cambio, presentan más de una arista en unión con los vértices.

Por otra parte, un grafo es conexo si dispone de dos vértices conectados a través de un camino. ¿Qué quiere decir esto? Que, para el par de vértices (p, r), tiene que existir algún camino que permita llegar desde p hasta r.

En cambio, un grafo es fuertemente conexo si el par de vértices tiene conexión a través de, como mínimo, dos caminos diferentes.

Un grafo simple, además, puede ser completo si las aristas están en condiciones de unir todos los pares de vértices, mientras que un grafo es bipartito si sus vértices surgen por la unión de un par de conjuntos de vértices y si se cumple una serie de condiciones. (Porto, 2012)

Vértice:

En la teoría de grafos, cada vértice está considerado como la unidad fundamental que compone a los grafos. Los grafos no dirigidos están compuestos por vértices y aristas (es decir, pares desordenados de vértices), mientras que los grafos dirigidos abarcan vértices y arcos (pares ordenados de vértices).
(Merino, 2012)

Arista:

En el terreno de la teoría de grafos, la arista surge por el vínculo que mantienen dos vértices de un mismo grafo. Cuando dos vértices se encuentran conectados a través de una artista, son adyacentes. En este marco, se dice que los vértices son incidentes a la arista en cuestión.
Para caracterizar un grafo G son suficientes únicamente el conjunto de todas sus aristas, comúnmente denotado con la letra E (del término en inglés edge), junto con el conjunto de sus vértices, denotado por V. Así, dicho grafo se puede representar como G(V,E), o bien G = (V,E).

En un grafo, dos vértices son adyacentes si están conectados por una arista. En tal caso, cada uno de estos vértices es incidente a dicha arista. (Reinhad, 1997)

Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

Algoritmo de Floyd Warshall:

El algoritmo de Floyd-Warshall, descrito en 1959 por Bernard Roy, es un algoritmo de análisis sobre grafos para encontrar el camino mínimo en grafos dirigidos ponderados. El algoritmo encuentra el camino entre todos los pares de vértices en una única ejecución. El algoritmo de Floyd-Warshall es un ejemplo de programación dinámica.

Muchos problemas de la vida cotidiana se pueden expresar e incluso resolver en forma de grafo. Existen algoritmos que encuentran distintos tipos de soluciones, tanto booleanas como de eficiencia. El grafo se representa en una tabla (matriz) que se conoce como “matriz de adyacencia” y representa si existe una unión entre dos nodos (boolean).

El algoritmo de Floyd-Warshall compara todos los posibles caminos a través del grafo entre cada par de vértices. (Wikipedia, 2017)

Algoritmo de Dijkstra:

El algoritmo de Dijkstra, es uno de los algoritmos de caminos mínimos, se usa para la determinación del camino más corto dado un vértice origen al resto de vértices en un grafo dirigido y con pesos en cada arista.

La idea subyacente en este algoritmo consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices; cuando se obtiene el camino más corto desde el vértice origen, al resto de vértices que componen el grafo, el algoritmo se detiene. El algoritmo es una especialización de la búsqueda de costo uniforme, y como tal, no funciona en grafos con aristas de costo negativo (al elegir siempre el nodo con distancia menor, pueden quedar excluidos de la búsqueda nodos que en próximas iteraciones bajarían el costo general del camino al pasar por una arista con costo negativo). (Gregorio, 2017)

Algoritmo de Prim:

Algoritmo de Prim es un algoritmo perteneciente a la teoría de los grafos para encontrar un árbol recubridor mínimo en un grafo conexo, no dirigido y cuyas aristas están etiquetadas.
En otras palabras, el algoritmo encuentra un subconjunto de aristas que forman un árbol con todos los vértices, donde el peso total de todas las aristas en el árbol es el mínimo posible. Si el grafo no es conexo, entonces el algoritmo encontrará el árbol recubridor mínimo para uno de los componentes conexos que forman dicho grafo no conexo.

El algoritmo incrementa continuamente el tamaño de un árbol, comenzando por un vértice inicial al que se le van agregando sucesivamente vértices cuya distancia a los anteriores es mínima. Esto significa que, en cada paso, las aristas a considerar son aquellas que inciden en vértices que ya pertenecen al árbol.
El árbol recubridor mínimo está completamente construido cuando no quedan más vértices por agregar. (Ecured, 2018)

Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

Algoritmo de kruskal:

Es un algoritmo de la teoría de grafos para encontrar un árbol recubridor mínimo en un grafo conexo y ponderado. Es decir, busca un subconjunto de aristas que, formando un árbol, incluyen todos los vértices y donde el valor total de todas las aristas del árbol es el mínimo. Si el grafo no es conexo, entonces busca un bosque expandido mínimo (un árbol expandido mínimo para cada componente conexa).

El algoritmo de Kruskal es un ejemplo de algoritmo voraz.
Un ejemplo de árbol expandido mínimo. Cada punto representa un vértice, el cual puede ser un árbol por sí mismo. Se usa el Algoritmo para buscar las distancias más cortas (árbol expandido) que conectan todos los puntos o vértices. (Ecured, Ecured, 2019)

Algoritmo BFS:

Es un algoritmo de búsqueda no informada utilizado para recorrer o buscar elementos en un grafo (usado frecuentemente sobre árboles). Intuitivamente, se comienza en la raíz (eligiendo algún nodo como elemento raíz en el caso de un grafo) y se exploran todos los vecinos de este nodo. A continuación, para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el árbol.

Formalmente, BFS es un algoritmo de búsqueda sin información, que expande y examina todos los nodos de un árbol sistemáticamente para buscar una solución. El algoritmo no usa ninguna estrategia heurística. (Cormen, 2019)

Algoritmo DFS:

El algoritmo DFS (Depth-First Search) es una forma sistemática de encontrar todos los vértices alcanzables de un grafo desde un vértice de origen. Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa, de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado.

Si G es un grafo un grafo conexo, el algoritmo de búsqueda en profundidad obtiene un árbol recubridor de G. Se trata de un grafo en el que aparecen todos los vértices de G, pero no todas sus aristas. El árbol recubridor no es único, depende del vértice de partida. (IAGraph, 2018).

Matriz de adyacencia:

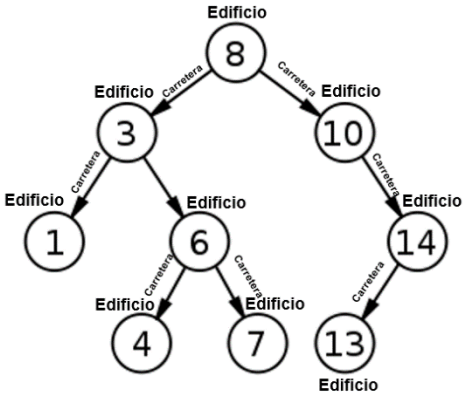
Todo grafo simple puede ser representado por una matriz, que llamamos matriz de adyacencia.
Se trata de una matriz cuadrada de N filas x N columnas (siendo N el número de vértices del grafo).
Para construir la matriz de adyacencia, cada elemento a_{i,j} vale {{1}} cuando haya una arista que una los vértices i y j. En caso contrario el elemento a{i,j} vale 0. La matriz de adyacencia, por tanto, estará formada por ceros y unos. (IES, 2019).

Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

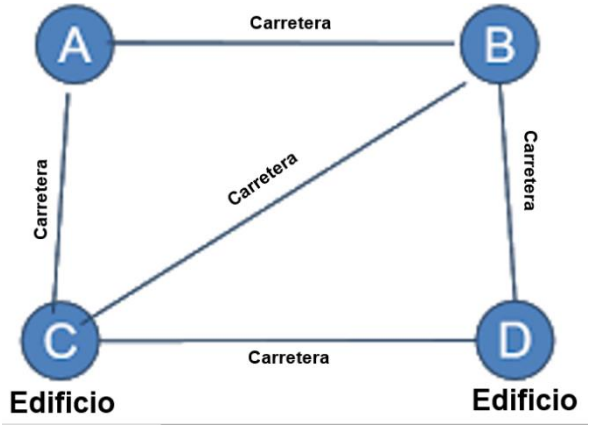
Fase 3: Búsqueda de soluciones creativas.

Para la solución de este problema necesitamos enfocarnos en que estructura de datos representa de la manera más precisa el contexto del problema, para esto vamos a generar las ideas usando conocimientos aprendidos en clase de Algoritmos y Estructuras de Datos.

Alternativa 1: Árbol binario de búsqueda.
Para esta alternativa vamos a representar y guardar los edificios y carreteras de la universidad en un árbol binario donde los nodos serían los edificios y las carreteras serian la relación del padre con el hijo (arista).



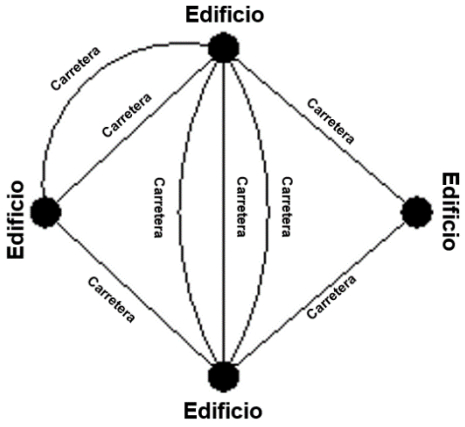
Alternativa 2: Grafo simple.
En esta alternativa se va a representar el problema por medio de un grafo simple, el cual sus aristas son no dirigidas, no se aceptan aristas múltiples y no se admiten bucles. En este grafo los vértices representaran los edificios y las aristas las carreteras.



Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

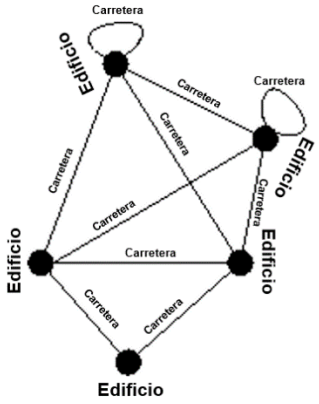
Alternativa 3: Multígrafo.

En esta alternativa se va a representar el problema por medio de un multígrafo, el cual sus aristas son no dirigidas, se aceptan aristas múltiples de un vértice a otro y no se admiten bucles. En este grafo los vértices representaran los edificios y las aristas las carreteras.



Alternativa 4: Pseudografo.

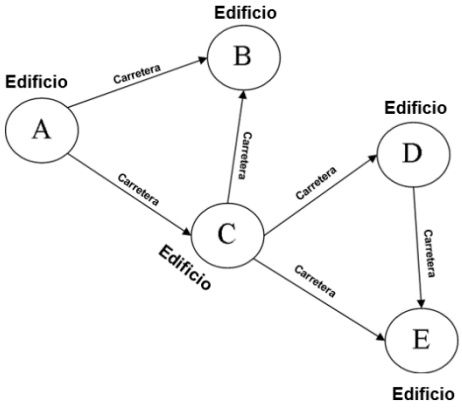
En esta alternativa se va a representar el problema por medio de un pseudografo, el cual sus aristas son no dirigidas, se aceptan aristas múltiples de un vértice a otro y se admiten bucles (arista que va de un nodo a él mismo). En este grafo los vértices representaran los edificios y las aristas las carreteras.



Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

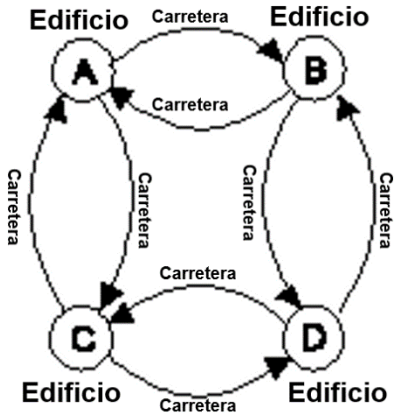
Alternativa 5: Grafo dirigido.

En esta alternativa se va a representar el problema por medio de un grafo dirigido, el cual sus aristas son dirigidas, no se aceptan aristas múltiples de un vértice a otro y se admiten bucles (arista que va de un nodo a él mismo). En este grafo los vértices representaran los edificios y las aristas las carreteras.



Alternativa 6: Multígrafo dirigido.

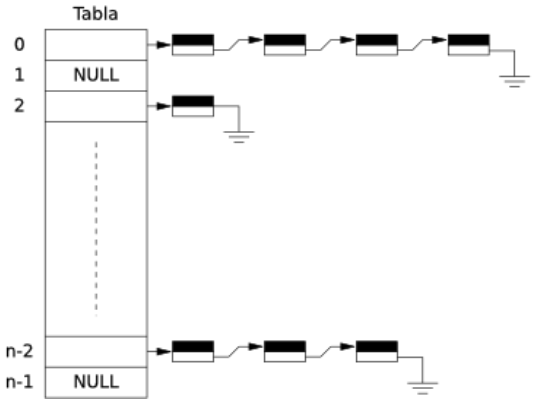
En esta alternativa se va a representar el problema por medio de un multígrafo dirigido, el cual sus aristas son dirigidas, se aceptan aristas múltiples de un vértice a otro y se admiten bucles (arista que va de un nodo a él mismo). En este grafo los vértices representaran los edificios y las aristas las carreteras.



Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

Alternativa 7: Doble tabla hash

Esta alternativa se basa en tener una tabla Hash que contenga dentro de ella otra tabla hash. Cada slot de la primera tabla será el vértice y la segunda tabla hash contendrá las aristas que contiene dicho vértice.



Fase 4: Transición de formulación de ideas a diseños preliminares.

- **Descarte de ideas no factibles**

Se descartaron las siguientes opciones de la búsqueda de soluciones creativas debido a:

Alternativa 1	Esta alternativa se descartó debido a que representar los caminos de la universidad entre cada edificio con un árbol de búsqueda binaria no es para nada representativo, debido a que en un árbol no podemos tener circuitos ni tampoco más de dos caminos hacia otro edificio, por estas razones esta idea no es factible.
Alternativa 3	Esta alternativa se descartó debido a que en el contexto del problema un edificio no puede tener más de dos carreteras hacia otro edificio.
Alternativa 5	Esta alternativa se descartó debido a que los caminos de la universidad hacia otro edificio se pueden transitar en los dos sentidos fácilmente, cosa que esta representación no tiene en cuenta.
Alternativa 6	Esta alternativa se descartó debido a que los caminos de la universidad hacia otro edificio se pueden transitar en los dos sentidos fácilmente, también porque resultaría un poco ilógico según el contexto del problema que tuviera bucles hacia el mismo edificio.
Alternativa 7	Esta alternativa se descartó debido a que es muy complicado y confuso representar los edificios y carreteras con dos tablas Hash, además esto resultaría bastante complicado de implementar en código.

Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

Fase 5: Evaluación y selección de la mejor solución.

Actualmente tenemos dos buenas propuestas para representar de manera eficiente el contexto de nuestro problema, pero sólo podemos escoger una, debido a esto vamos a evaluar las dos alternativas con una serie de criterios que nos dirán cual es la mejor solución.

Criterios:

- **Criterio A:** Representación del problema.
  Este criterio se basa en que tan representativa es la alternativa al contexto del problema.
  - ✓ Exacta: 3 puntos.
  - ✓ Media: 2 puntos.
  - ✓ Inexacta: 1 punto.

- **Criterio B:** Facilidad en la implementación.
  Este criterio se basa en que tan fácil es implementar la alternativa en código.
  - ✓ Fácil: 3 puntos.
  - ✓ Media: 2 puntos.
  - ✓ Difícil: 1 punto.

- **Criterio C:** Complejidad Espacial.
  Esta alternativa se basa en que tanto la alternativa memoria la alternativa necesita para almacenar los datos.
  - ✓ O(1): 3 puntos.
  - ✓ O(n^2): 2 puntos.
  - ✓ O(n!): 1 punto.

	Criterio A	Criterio B	Criterio C	Total
Alternativa 2	3	3	2	8
Alternativa 4	2	3	2	7

Con base en los resultados obtenidos vamos a descartar la alternativa 4 y vamos a implementar la alternativa 2.

Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

Fase 6: Preparación de informes y especificaciones.

Objetivo: Probar que el método instert(T objeto) funcione correctamente para diferentes casos de prueba				
Clase: OurGraph		**Método: Insert**		
Caso #	**Descripción:**	**Escenario**	**Valores de entrada**	**Resultado**
1 (Caso estandar)	Se crea un grafo en el cual sus vértices contienen los enteros {1,2,3,4,5} respectivamente, y sus aristas con pesos enteros {3,3,2,6,4}	stageOne()	Se entregan dos vértices, uno con valor 4 y el otro con valor 6. El peso de la arista que va a ser 2, y el dato que contiene la arista que será 60	El algoritmo insert() debe de devolver el valor 6, indicando que se insertó todo correctamente
2 (Caso interesante)	Se crea un grafo en el cual sus vértices contienen los strings {A,B,C,D,E} respectivamente, y sus aristas con pesos enteros {3,3,2,6,4}	stageTwo()	Se entregan dos vértices, uno con valor “D” y el otro con valor “F”. El peso de la arista que va a ser 2, y el dato que contiene la arista que será “DF”	El algoritmo insert() debe de devolver el valor “F”, indicando que se insertó todo correctamente
3 (Caso limite)	Se crea un grafo universidad con vértices que representan los edificios {A,B,C,D,E}, y sus aristas son caminos representados como {AB,BC,BD, DC,CE}	stageThree()	Se entregan dos vértices uno que representan un objeto tipo edificio con valor “F” y el otro representa el vértice con el cual será conectado con valor “D”. Y un camino	El algoritmo insert() debe de devolver el valor “F”, indicando que se insertó todo correctamente

Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

			con valor “DF” con peso 2	

Objetivo: Probar que el método Search(T objeto) funcione correctamente para diferentes casos de prueba				
Clase: OurGraph		**Método: Search**		
Caso #	**Descripción:**	**Escenario**	**Valores de entrada**	**Resultado**
1 (Caso estandar)	Se crea un grafo en el cual sus vértices contienen los enteros {1,2,3,4,5} respectivamente, y sus aristas con pesos enteros {3,3,2,6,4}	stageOne()	Se entregan 3 vértices de tipo entero a buscar con valores 1,3,5 respectivamente	El algoritmo Search() debe de devolver los valores 1, 3,5 respectivamente indicando que se encontraron los vértices correctamente
2 (Caso interesante)	Se crea un grafo en el cual sus vértices contienen los strings {A,B,C,D,E} respectivamente, y sus aristas con pesos enteros {3,3,2,6,4}	stageTwo()	Se entregan 3 vértices de tipo String a buscar con valores “B”,”C”, ”D” respectivamente	El algoritmo Search() debe de devolver los valores “B”,”C”,”D” respectivamente indicando que se encontraron los vértices correctamente
3 (Caso limite)		stageThree()	Se entregan dos vertices vértice a buscar de tipo edificio con valores “A” y “E” respectivamente	El algoritmo Search() debe de devolver los valores “A” Y “E” indicando que se encontraron los vértices correctamente

Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

Objetivo: Probar el correcto funcionamiento del método Dijkstra				
Clase: OurGraph		**Método: Dijkstra (Vértice)**		
Caso #	**Descripción:**	**Escenario**	**Valores de entrada**	**Resultado**
3	Se crea un grafo universidad con vértices que representan los edificios {A,B,C,D,E}, y sus aristas son caminos representados como {AB,BC,BD,DC,CE}	stageThree()	El método recibe dos vértices “A” y “C” que corresponden a edificios	Se espera que el método devuelva una pila de Strings con las distancias más cortas del vértice pasado como parámetro hacia todos los demás

Objetivo: Probar el correcto funcionamiento del método que encuentra el árbol de mínima expansión Prim				
Clase: OurGraph		**Método: Prim (Vértice)**		
Caso #	**Descripción:**	**Escenario**	**Valores de entrada**	**Resultado**
3	Se crea un grafo universidad con vértices que representan los edificios {A,B,C,D,E}, y sus aristas son caminos representados como {AB,BC,BD,DC,CE}	stageThree()	El método recibe una pila con los vértices a buscar el camino mas corto	Se espera que el resultado sea una pila con los pesos de las aristas que corresponden al camino más corto entre dos vértices dados

Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

Objetivo: Probar el correcto funcionamiento del método de búsqueda de amplitud BFS				
Clase: OurGraph		**Método: BFS(Vertex)**		
Caso #	**Descripción:**	**Escenario**	**Valores de entrada**	**Resultado**
1 (Caso estandar)	Se crea un grafo en el cual sus vértices contienen los enteros {1,2,3,4,5} respectivamente, y sus aristas con pesos enteros {3,3,2,6,4}	stageOne()	Se el vértice con valor entero 1 a buscar el camino por amplitud	Devuelve un arreglo de enteros con los valores {1,2,3,4,5} correspondientes al orden de la búsqueda por profundidad
2 (Caso interesante)	Se crea un grafo en el cual sus vértices contienen los strings {A,B,C,D,E} respectivamente, y sus aristas con pesos enteros {3,3,2,6,4}	stageTwo()	Se el vértice de tipo String con valor “A” a buscar el camino por amplitud	Devuelve un arreglo de Strings con los valores {“A”,”B”,”D”,”C”,”E”} correspondientes al orden de la búsqueda por profundidad
3 (Caso limite)	Se crea un grafo universidad con vértices que representan los edificios {A,B,C,D,E}, y sus aristas son caminos representados como {AB,BC,BD,DC,CE}	stageThree()	Se el vértice de tipo Edificio con valor “A” a buscar el camino por amplitud	Devuelve un arreglo de Strings con los valores {“A”,”B”,”C”,”D”,”E”} correspondientes al orden de la búsqueda por profundidad

Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

Objetivo: Probar el correcto funcionamiento del método de búsqueda por profundidad BFS				
Clase: OurGraph		**Método: DFS(Vertex)**		
Caso #	**Descripción:**	**Escenario**	**Valores de entrada**	**Resultado**
1 (Caso estandar)	Se crea un grafo en el cual sus vértices contienen los enteros {1,2,3,4,5} respectivamente, y sus aristas con pesos enteros {3,3,2,6,4}	stageOne()	Se el vértice con valor entero 1 a buscar el camino por amplitud	Devuelve un arreglo de enteros con los valores {1,2,4,3,5} correspondientes al orden de la búsqueda por profundidad
2 (Caso interesante)	Se crea un grafo en el cual sus vértices contienen los strings {A,B,C,D,E} respectivamente, y sus aristas con pesos enteros {3,3,2,6,4}	stageTwo()	Se el vértice de tipo String con valor “A” a buscar el camino por amplitud	Devuelve un arreglo de Strings con los valores {“A”,”B”,”D”,”C”,”E”} correspondientes al orden de la búsqueda por profundidad
3 (Caso limite)	Se crea un grafo universidad con vértices que representan los edificios {A,B,C,D,E}, y sus aristas son caminos representados como {AB,BC,BD, DC,CE}	stageThree()	Se el vértice de tipo Edificio con valor “A” a buscar el camino por amplitud	Devuelve un arreglo de Strings con los valores {“A”,”B”,”D”,”C”,”E”} correspondientes al orden de la búsqueda por profundidad

Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

Objetivo: Probar el correcto funcionamiento del método Floyd-Warshall				
Clase: OurGraph		**Método: Floyd-Warshall (Matriz de adyacencia)**		
Caso #	**Descripción:**	**Escenario**	**Valores de entrada**	**Resultado**
1	Se crea un grafo, el cual sus vértices contendrán edificios y sus aristas contendrán caminos	scenarioThree()	El método recibirá una matriz de adyacencia, la cual representa todos los caminos que hay de la universidad para llegar a cualquier edificio	Se espera que el método devuelva una matriz de adyacencia con los caminos más cortos que hay de un vértice a los demás.

Objetivo: Probar el correcto funcionamiento del método que encuentra el árbol de mínima expansión Kruskal				
Clase: OurGraph		**Método: Kruskal (Vértice)**		
Caso #	**Descripción:**	**Escenario**	**Valores de entrada**	**Resultado**
1	Se crea un grafo, el cual sus vértices contendrán edificios y sus aristas contendrán caminos	scenarioThree()	El método recibirá un vértice como parámetro, el cual será la raíz del árbol	Se espera que el resultado sea un árbol cuya raíz sea el vértice pasado como parámetro y que contenga todos los vértices

Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

Diseños preliminares:

Para los diseños preliminares hemos decidido hacer el algoritmo más importante de cada estructura de datos, esto nos servirá para tener una idea de cómo implementar estos algoritmos y aprovecharemos para sacar la complejidad espacial y temporal de cada algoritmo.

Metodo 1:

#	dijkstra(V initialVertex)	C.E	C.T
1	`Vertex<V> newVertex = new Vertex<V>(initialVertex)`	1	1
2			
3	`int[][] distances = new int[2][adjMatrix.length]`	N	1
4	`PriorityQueue<Node> myQueue = new PriorityQueue<Node>(SIZE_PQ)`	N	1
5	`Set<Integer> visited = new HashSet<Integer>()`	N	1
6	`for(int i = 0; i < adjMatrix.length; i++)`	0	N + 1
7	`distances[0][i] = Integer.MAX_VALUE`	1	N
8	`distances[1][i] = -1`	1	N
9			
10	`myQueue.insert(new Node(representationV.get(newVertex.getDateV()), 0))`	0	1
11	`distances[0][representationV.get(newVertex.getDateV())] = 0`	1	1
12	`distances[1][representationV.get(newVertex.getDateV())] = representationV.get(newVertex.getDateV())`	1	1
13			
14	`while(!myQueue.isEmpty())`	0	N+1
15	`int evaluationNode = myQueue.extractMin().node`	1	N
16	`visited.add(evaluationNode)`	0	N
17			
18	`int edgeDistance = -1`	1	N
19	`int newDistance = -1`	1	N
20			
21	`int algo = representationV.get(newVertex.getDateV())`	1	N
22	`for(int i = 0; i < adjMatrix.length; i++)`	0	N^2 + N
23	`if(adjMatrix[evaluationNode][i] != null)`	0	N^2
24	`if(!visited.contains(i))`	0	N^2
25	`if(adjMatrix[evaluationNode][i].getWeight() != Integer.MAX_VALUE)`	0	N^2
26	`edgeDistance = adjMatrix[evaluationNode][i].getWeight()`	1	N^2
27	`newDistance = distances[0][evaluationNode] + edgeDistance`	1	N^2
28	`if(newDistance < distances[0][i])`	0	N^2
29	`distances[0][i] = newDistance`	1	N^2
30	`distances[1][i] = algo`	1	N^2
31			

Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

32	myQueue.insert(new Node(i,distances[0][i]))	0	N^2
33			
34	algo = evaluationNode	1	N^2
35			
36	return distances	0	1
37			
38			

Metodo 2:

#	prim(V from)	C.E	C.T
1	boolean[] visited = new boolean[adjMatrix.length]	N	1
2	PriorityQueue<Edge<V,A>> theQueue = new PriorityQueue<Edge<V,A>>(*SIZE_PQ*)	N	1
3	int auxVisited = 0	1	1
4	Stack<Edge<V,A>> theReturn = new Stack<Edge<V,A>>()	N	
5			1
6	visited[representationV.get(from)] = true	1	1
7	auxVisited++	0	1
8			
9	for(int i = 0 ; i < adjMatrix[0].length; i++)	0	N+1
10	if(adjMatrix[representationV.get(from)][i] != null)	0	N
11	theQueue.insert(adjMatrix[representationV.get(from)][i])	0	N
12			
13	while(auxVisited < visited.length)	0	N+1
14	Edge<V, A> auxEdge = theQueue.extractMin()	1	N
15			N
16	if((visited[representationV.get(auxEdge.getVertexOne())] == false) || (visited[representationV.get(auxEdge.getVertexTwo())] == false))	0	N
17			
18	theReturn.push(auxEdge)	0	N
19			
20	if(!visited[representationV.get(auxEdge.getVertexOne())])	0	N
21			
22	visited[representationV.get(auxEdge.getVertexOne())] = true	1	N
23	auxVisited++	0	N
24			
25	for(int i = 0 ; i < adjMatrix[0].length; i++)	0	N^2 + n
26	if(adjMatrix[representationV.get(auxEdge.getVertexOne())][i] != null)	0	N^2
27	theQueue.insert(adjMatrix[representationV.get(auxEdge.getVertexOne())][i])	0	N^2
28			
29	if(!visited[representationV.get(auxEdge.getVertexTwo())])	0	N
30	visited[representationV.get(auxEdge.getVertexTwo())] = true	1	N
31	auxVisited++	0	N

Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

32			
33	`for(int i = 0 ; i < adjMatrix[0].length; i++)`	0	N^2
34	`if(adjMatrix[representationV.get(auxEdge.getVertexTwo())][i] != null)`	0	N
35	`theQueue.insert(adjMatrix[representationV.get(auxEdge.getVertexTwo())][i])`	0	N
36			
37	`return theReturn`	1	1
38			

Metodo 3:

#	bfs(V from)	C.E	C.T
1	`boolean[] visited = new boolean[adjMatrix.length]`	N	1
2	`Queue<V> theQueue = new Queue<V>()`	N	1
3	`ArrayList<V> theReturn = new ArrayList<V>()`	N	1
4			
5	`visited[representationV.get(from)] = true`	0	1
6	`theQueue.enqueue(from)`	0	1
7			
8	`while(!theQueue.isEmpty())`	0	N+1
9			
10	`V aux = theQueue.dequeue()`	1	N
11	`theReturn.add(aux)`	0	N
12			
13	`for(int i = 0; i < adjMatrix.length; i++)`	0	N^2
14	`if(adjMatrix[representationV.get(aux)][i] != null)`	0	N
15	`V anotherAux = adjMatrix[representationV.get(aux)][i].getVertexTwo()`	1	N
16	`if(!visited[representationV.get(anotherAux)])`	0	N
17	`theQueue.enqueue(anotherAux)`	0	N
18	`visited[representationV.get(anotherAux)] = true`	1	N
19			
20	`return theReturn;`	0	1

Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

Análisis de complejidad temporal

- ✓ **Método 1:**
  $\boldsymbol{T(n) = An^2 + Bn + C}$

Por medio del T(n) anterior podemos concluir que la notación asintótica para la complejidad temporal del método 1 es: $O(n^2)$.

- ✓ **Método 2:**
  $\boldsymbol{T(n) = An^2 + Bn + C}$

Por medio del T(n) anterior podemos concluir que la notación asintótica para la complejidad temporal del método 2 es: $O(n^2)$.

- ✓ **Método 3:**
  $\boldsymbol{T(n) = An^2 + Bn + C}$

Por medio del T(n) anterior podemos concluir que la notación asintótica para la complejidad temporal del método 3 es: $O(n^2)$.

Análisis de complejidad espacial

- ✓ **Método 1:**
  $\boldsymbol{T(n) = An + C}$

Por medio del T(n) anterior podemos concluir que la notación asintótica para la complejidad espacial del método 1 es: $O(n)$.

- ✓ **Método 2:**
  $\boldsymbol{T(n) = An + C}$

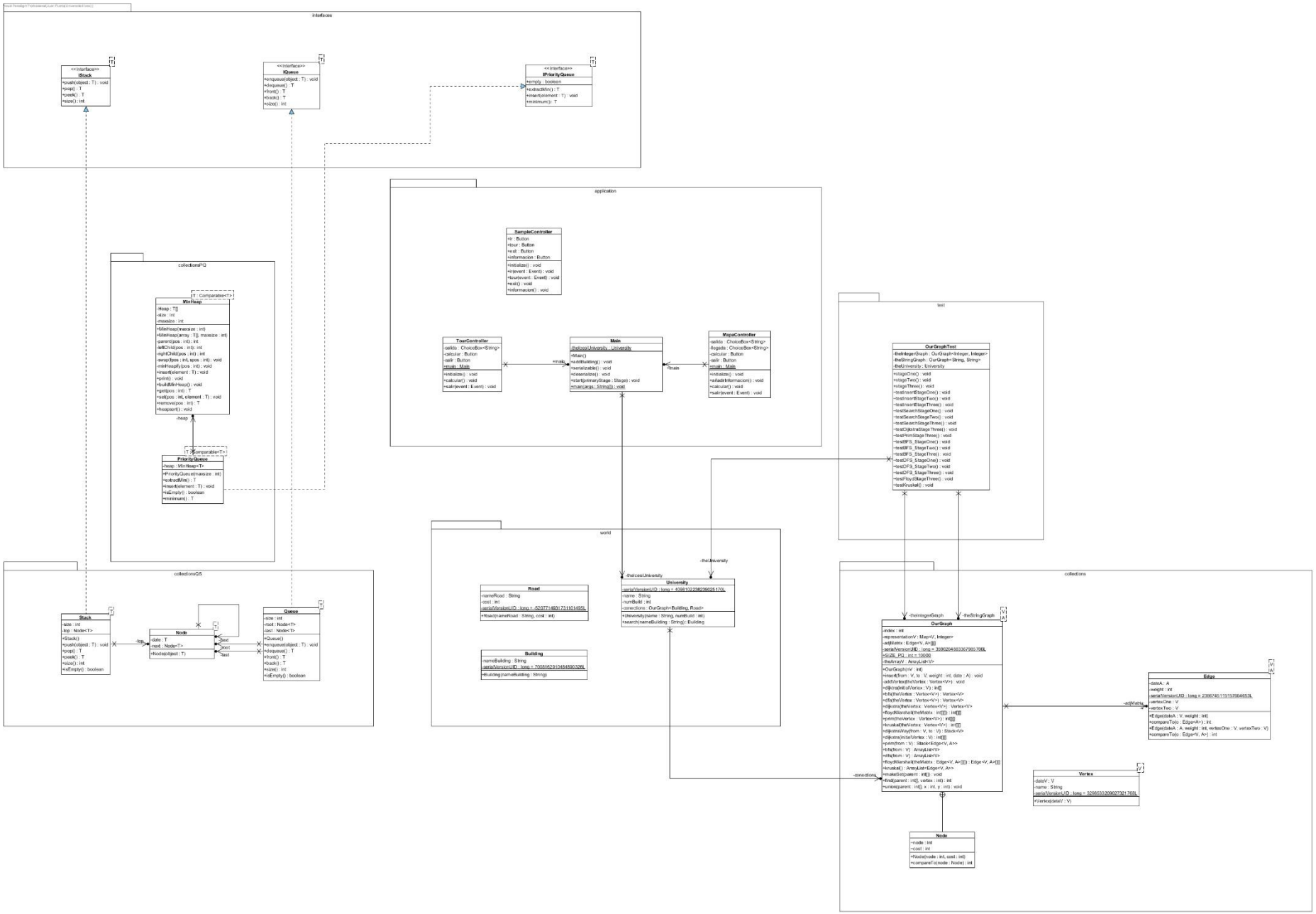
Por medio del T(n) anterior podemos concluir que la notación asintótica para la complejidad espacial del método 2 es: $O(n)$.

- ✓ **Método 3:**
  $\boldsymbol{T(n) = An + C}$

Por medio del T(n) anterior podemos concluir que la notación asintótica para la complejidad espacial del método 3 es: $O(n)$.

Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

Diagrama de clases



Fabián David Portilla
Sebastián Puerta
Juan Camilo Castillo

Fase 7: Implementación.

La implementación de la solución se encuentra en el siguiente repositorio de github:

https://github.com/Juan-Puerta/ProyectoGrafo