**PROYECTO FINAL AED**

**JSDomicilios**

**Contexto problemático:**

Wakanda era un sitio tranquilo y armónico, con una población que amaba la comida. Debido a que había un mal que asechaba, más conocido como Thanos, su gente se encontraba entrenada para la guerra en cualquier momento. Desafortunadamente el día de la guerra llegó y a pesar de su entrenamiento Thanos fue más fuerte, destruyendo varios edificios y vías de la ciudad.

JSDomicilios es una empresa que maneja su propio menú de domicilios, este nace de la necesidad de llevar domicilios más rápida y eficientemente alrededor de la ciudad de Wakanda, ya que luego de la invasión de Thanos los niveles de tráfico aumentaron considerablemente debido a la destrucción de algunas vías, esto llevo a la reestructuración de las mismas lo cual hizo que las distancias se modificaran. La misión de JSDomicilios es llevar los pedidos de comida al lugar de residencia del usuario lo más rápido posible, considerando las condiciones de tráfico y distancias.

**Identificación y definición concreta del problema:**

La población de Wakanda quiere obtener su comida con mayor rapidez.

**Recopilación de Información:**

* **Grafo:** En matemáticas y ciencias de la computación, un grafo (del griego grafos: dibujo, imagen) es un conjunto de objetos llamados vértices o nodos unidos por enlaces llamados aristas o arcos, que permiten representar relaciones binarias entre elementos de un conjunto.​ Son objeto de estudio de la teoría de grafos. (Wikipedia, 2018)
* **Domicilio:** Es la circunscripción territorial donde se asienta una persona.

A partir de esta definición, también se entiendo como domicilio o a domicilio, a todo tipo de servicio que es prestado en el lugar de residencia de una persona. (Wikipedia, 2018)

* **Wakanda:** Es una nación ficticia del universo de Marvel Comics localizada en África. Es el más prominente de varias naciones nativas de África y hogar del superhéroe Pantera Negra.
* **Thanos:** Es un supervillano ficticio que aparece en los cómics norteamericanos publicados por Marvel Comics.

1. **Especificación de Requerimientos Funcionales**

|  |  |
| --- | --- |
| **Nombre** | **Req. 001 – Hacer pedido** |
| **Resumen** | Dada una lista de productos, permite al usuario escoger los productos para su pedido. |
| **Entrada** | * Productos a escoger |
| **Salida** | El pedido se ha realizado. |

|  |  |
| --- | --- |
| **Nombre** | **Req. 002 – Mostrar estado del pedido** |
| **Resumen** | Permite visualizar gráficamente el camino transcurrido del domiciliario que lleva el pedido a la residencia del usuario. |
| **Entrada** | <No requiere> |
| **Salida** | Se ha mostrado el estado del pedido. |

|  |  |
| --- | --- |
| **Nombre** | **Req. 003 – Entregar pedido rápidamente** |
| **Resumen** | Permite que el domiciliario encuentre el camino más corto para entregar el pedido lo más rápido posible. Tiene en cuenta la distancia (sumatoria del largo de las calles), y los niveles de tráfico. |
| **Entrada** | <No requiere> |
| **Salida** | Se ha entregado el pedido al usuario. |

|  |  |
| --- | --- |
| **Nombre** | **Req. 004 – Mostrar tráfico** |
| **Resumen** | Durante el recorrido del pedido, este requerimiento permite visualizar es estado del tráfico de las calles representado por colores. Rojo: cuando el nivel de tráfico es alto, Amarillo: cuando el nivel de tráfico es medio, Verde: cuando el nivel de tráfico es bajo. |
| **Entrada** | **<**No requiere**>** |
| **Salida** | Se ha mostrado el estado del tráfico. |

**Búsqueda de Soluciones creativas:**

*Alternativa 1: BFS*

Es un algoritmo de búsqueda no informada utilizado para recorrer o buscar elementos en un grafo. Intuitivamente, se comienza en la raíz 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. (Wikipedia, 2017)

*Alternativa 2: DFS*

Es un algoritmo de búsqueda no informada utilizado para recorrer todos los nodos de un grafo de manera ordenada, pero no uniforme. 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. (Wikipedia, 2017)

*Alternativa 3: Algoritmo de Dijkstra*

Es un algoritmo para la determinación del camino más corto, dado un vértice origen, hacia el resto de los vértices en un grafo que tiene pesos en cada arista. (Wikipedia, 2018)

*Alternativa 4: Algoritmo de Floyd-Warshall*

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. (Wikipedia, 2018)

*Alternativa 5: Algoritmo de Bellman-Ford*

Al igual que el algoritmo de Dijkstra, este genera el camino más corto en un grafo dirigido ponderado. Con la diferencia de que el peso de alguna de las aristas puede ser negativo. (Wikipedia, 2017)

*Alternativa 6: Algoritmo de Prim*

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. (Wikipedia, 2017)

*Alternativa 7: 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 de la suma 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). (wikipedia, 2018)

Las anteriores estructuras de datos serán utilizadas en este caso para:

* Recorrer las ciudades de Wakanda teniendo en cuenta las conexiones entre ellas (caminos).
* Encontrar el camino con menor distancia para llevar el domicilio desde el edificio de JSDomicilios a alguna de las ciudades de Wakanda.
* Encontrar el subconjunto de caminos que conecten todas las ciudades con la mínima distancia posible.

**Transición de la formulación de ideas a los diseños preliminares:**

Alternativa 1:

* Explora los vértices de G para “descubrir” todos los vértices alcanzables desde s.
* Calcula la distancia (menor número de vértices) desde s a todos los vértices alcanzables.
* expande y examina todos los nodos de un árbol sistemáticamente para buscar una solución.
* Realiza el recorrido por niveles.
* Implementa una cola en su estructura.
* Se puede utilizar cuando se necesita buscar el camino más corto en grafos no ponderados.

Alternativa 2:

* Puede utilizarse de forma recursiva, o en caso contrario, implementar una pila en su estructura.
* Se puede utilizar para detectar ciclos en un grafo, determinar si un grafo es conexo o no y cuántas componentes conexas tiene, determinar puntos de articulación y biconexión de grafos.
* Recorre el grafo desde la raíz hasta el extremo de una de sus ramas (Búsqueda en profundidad).

Alternativa 3:

* Es posible resolver grafos con muchos nodos.
* Encuentra las rutas más cortas entre un origen y todos los destinos en una red.
* Determina la longitud del camino más corto entre dos vértices de un grafo ponderado simple, conexo y no dirigido con n vértices.

Alternativa 4:

* Encuentra el camino entre todos los pares de vértices, en donde cada arista en el grafo tiene un peso, el cual puede ser positivo o negativo.
* Se puede usar para detectar ciclos negativos

Alternativa 5:

* Permite encontrar el camino con menor distancia entre 2 nodos del grafo.
* Se recomienda usar cuando hay aristas con peso negativo.
* Si el grafo contiene un ciclo de coste negativo, el algoritmo lo detectará, pero no encontrará el camino más corto que no repite ningún vértice.

Alternativa 6:

* Los vértices se pueden añadir como una única Lista enlazada o un vector de vértices, o como una estructura de datos organizada con una cola de prioridades, más compleja.
* El algoritmo incrementa continuamente el tamaño de un árbol.
* Está completamente construido cuando no quedan más vértices por agregar.
* El peso total de todas las aristas en el árbol es el mínimo posible.

Alternativa 7:

* Si el grafo no es conexo, entonces busca un bosque expandido mínimo.
* Al acabar el algoritmo, el bosque tiene un solo componente, el cual forma un árbol de expansión mínimo del grafo.

**Evaluación y selección de la mejor solución:**

Criterios para evaluación:

*Criterio A:* Complejidad (peor caso).

* [8] complejidad constante.
* [7] complejidad logarítmica.
* [6] complejidad radical.
* [5] complejidad lineal.
* [4] complejidad n(log n).
* [3] complejidad polinómica.
* [2] complejidad exponencial.
* [1] complejidad factorial.

Criterio B: Eficiencia en manejo de datos

* [2] Eficiente para muchos datos.
* [1] Eficiente para pocos datos.

Criterio C: Dificultad de implementación.

* [3]Facilidad de implementación.
* [2]Normalidad en la implementación.
* [1]Dificultad para la implementación.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Criterio A** | **Criterio B** | **Criterio C** | **Total**  Mientras más alto sea el número, mejor es en complejidad. |
| **BFS** | Es siendo V el número de nodos que visita y E el número de aristas.  **8** | Es eficiente para muchos datos.  **2** | Es fácil de implementar.  **3** | **13** |
| **DFS** | Es siendo b el factor de ramificación (número promedio de ramificaciones por nodo) y m la máxima profundidad del espacio de los estados.  **3** | Es eficiente para muchos datos.  **2** | Es fácil de implementar. **3** | **8** |
| **Dijkstra** | Es utilizando cola de prioridad.  **7** | Es eficiente para muchos datos.  **2** | Es fácil de implementar  **3** | **12** |
| **Floyd-Warshall** | Es siendo V el número de vértices en un grafo.  **3** | Es eficiente para muchos datos  **2** | Tiene una implementación Normal.  **2** | **7** |
| **Bellman - Ford** | Es siendo V el número de vértices en un grafo y E el número de aristas. Es en caso que .  **3** | Es eficiente para muchos datos.  **2** | Tiene una implementación Normal.  **2** | **7** |
| **Prim** | Es siendo V el número de vértices en un grafo, y E el número de aristas. **7** | Es eficiente para muchos datos.  **2** | Tiene una implementación difícil  **1** | **10** |
| **Kruskal** | Es siendo V el número de vértices en un grafo y E el número de aristas.  **7** | Es eficiente para muchos datos.  **2** | Tiene una implementación difícil.  **1** | **10** |

**Selección de la mejor solución:**

Analizando cada uno de los casos, llegamos a la conclusión de que implementaremos las alternativas 1 y 3 por su eficiencia algorítmica y facilidad de implementar; también las alternativas 2 y 4 por su facilidad de implementación.

**Diseño de tipos abstractos de datos TAD**

|  |  |  |
| --- | --- | --- |
| **TAD** | **GRAFO** | |
| https://upload.wikimedia.org/wikipedia/commons/thumb/5/5b/6n-graf.svg/200px-6n-graf.svg.png  Grafo de 6 vértices y 7 aristas | | |
| **Invariantes**  El grafo se define como **G = (V, E)** siendo **V** el conjunto de vértices y **E** un conjunto de aristas. | | |
| Grafo () 🡪 Constructor  crearGrafo () 🡪 Grafo  agregarArista (grafo,nodo1,nodo2) 🡪 Grafo 🡪 Arista  agregarNodo (grafo, nodo) 🡪 Grafo 🡪 Nodo  eliminarNodo (grafo, nodo) 🡪 Grafo 🡪 Grafo  eliminarArista (grafo, nodo, nodo) 🡪 Grafo 🡪 Grafo  isEmpty () 🡪 Grafo 🡪 boolean  contain () 🡪 Grafo 🡪 boolean  adyacentes(Grafo, nodo, nodo)🡪 Grafo 🡪 boolean  predecesor(Grafo, nodo, nodo) 🡪 Grafo🡪 boolean  sucesor (Grafo, nodo, nodo) 🡪 Grafo 🡪 boolean | | Constructor  Generador  Generador Generador  Constructor Constructor  Consultor  Consultor  Consultor  Consultor  Consultor |
| * **Pre:** Se ha creado una instancia de un grafo. * **Post:** Se ha creado un nuevo grafo vacío. * **Def:** Crea un grafo vacío.   ***public Grafo crearGrafo ()***   * **Pre:** El grafo(grafo) ha sido creado. Los nodos (nodo1, nodo2) han sido añadidos al grafo(grafo). * **Post:** Se ha creado una relación entre los dos nodos (nodo1, nodo2). * **Def:** Dado un grafo(grafo) añade una relación entre dos nodos (nodo1 y nodo2) del mismo grafo.   ***public void agregarArista (grafo, nodo1, nodo2)***   * **Pre:** El grafo(grafo) ha sido creado. * **Post:** Se ha añadido un nuevo nodo en el grafo(grafo). * **Def:** Dado un grafo(grafo) incluye un nodo( nuevoNodo) en él.   **public void agregarNodo( grafo,nuevoNodo)**   * **Pre:** El grafo(grafo) ha sido creado. El nodo(nodo) ha sido añadido al grafo(grafo). * **Post:** El nodo(nodo) ha sido eliminado del grafo. * **Def:** Dado un grafo(grafo) elimina un nodo(nodo) que está incluido en él.   **public Grafo eliminarNodo(grafo, nodo)**   * **Pre:** El grafo(grafo) ha sido creado. Los nodos (nodo1, nodo2) han sido añadidos al grafo(grafo). * **Post:** La arista existente entre el nodo1 y nodo2 del grafo, ha sido eliminada. * **Def:** Dado un grafo elimina la relación entre dos nodos pertenecientes a este.   **public Grafo eliminarArista(grafo,nodo1,nodo2)**   * **Pre:** El grafo ha sido inicializado. * **Post:** Se ha retornado un booleano que determina si el grafo está vacio. * **Def:** Verifica si un grafo está vacío (no contiene nodos).   **public boolean isEmpty(grafo)**   * **Pre:** El grafo ha sido inicializado. * **Post:** Se ha retornado un booleano que determina si el nodo(nodo) pertenece al grafo(grafo) * **Def:** Verifica si un nodo pertenece a un grafo.   **Public boolean contain(grafo, nodo)**   * **Pre:** El grafo ha sido inicializado. Los nodos (nodo1, nodo2) han sido añadidos al grafo (grafo). * **Post:** Se ha verificado si los nodos ( nodo1, nodo2) tienen una arista en común. * **Def:** Comprueba si dos nodos tienen una arista que los relaciones.   **public boolean adyacentes(grafo,nodo1,nodo2)**   * **Pre:** El grafo ha sido inicializado. Los nodos (nodo1, nodo2) han sido añadidos al grafo (grafo). * **Post:** Se ha verificado si el nodo2 es predecesor del nodo1. * **Def:** Dado un grafo, comprueba si un nodo es predecesor de otro. (para un grafo dirigido)   **public boolean predecesor(grafo,nodo1,nodo2)**   * **Pre:** El grafo ha sido inicializado. Los nodos (nodo1, nodo2) han sido añadidos al grafo (grafo). * **Post:** Se ha verificado si el nodo2 es sucesor del nodo1. * **Def:** Dado un grafo, comprueba si un nodo es sucesor de otro. (para un grafo dirigido)   **public boolean sucesor(grafo,nodo1,nodo2)** | | |

**Diseño de pruebas unitarias**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entrada | | Salida | Descripción | Nombre del Método | Clase | Escenario |
| <B> | | **True** | El método disminuye la cantidad de vértices cuando el nodo ingresado pertenece al Grafo, y este es eliminado. El método no hace nada si el nodo no pertenece al Grafo, y por tanto no se puede eliminar. | removeVertex(Verte) | **Graph** | SceneOne() |
| <H> | | **False** |
| (<B>,<E>) | | **True** | El método disminuye la cantidad aristas cuando la arista ingresada pertenece al grafo y es eliminada. No hace nada en caso de no pertenecer al grafo, y por tanto no se puede eliminar. | removeEdge(Edge) | **Graph** | SceneTwo() |
| (<B>,<A>) | | **False** |
| Dado el grafo :  Vertices:  <A> <B> <C> <D> <E> <F>  Aristas:  (<A>, <B>)  (<B>, <A>)  (<B>, <C>)  (<C>, <B>)  (<C>, <D>)  (<D>, <C>)  (<E>, <D>)  (<D>, <E>)  (<F>, <E>)  (<E>, <F>) | | | El método retorna true cuando el nodo ingresado pertenece al grafo, retorna false cuando no pertenece. | contain(Vertex) | **Graph** | SceneThree() |
| <A> | | **True** |
| <H> | | **False** |
| Dado el grafo1:  Vertices:  Null  Aristas:  Null | **Dado el grafo 2:**  **Vertices:**  <A> <B> <C> <D> <E> <F>  Aristas:  (<A>, <B>)  (<B>, <A>)  (<B>, <C>)  (<C>, <B>)  (<C>, <D>)  (<D>, <C>)  (<E>, <D>)  (<D>, <E>)  (<F>, <E>)  (<E>, <F>) | | El método retorna true cuando el grafo está vacío (no contiene ningún nodo), retorna false si posee por lo menos un nodo. | isEmpty(Graph) | **Graph** | SceneFour() |
| Grafo1 | | **True** |
| Grafo2 | | **False** |
| Dado el grafo:  Vertices:  <A> <B> <C> <D> <E> <F>  Aristas:  (<A>, <B>)  (<B>, <A>)  (<B>, <C>)  (<C>, <B>)  (<C>, <D>)  (<D>, <C>)  (<E>, <D>)  (<D>, <E>)  (<F>, <E>)  (<E>, <F>)  Crear una arista entre los nodos = A, F | | | El método aumenta la cantidad de aristas cuando se ha agregado una arista del nodo ingresado, entre los dos nodos ingresados. No hace nada , en caso contrario. | addEdge(Vertex,Vertex) | **Graph** | SceneFive() |
| (<A>, <B>)  (<F>, <A>) | | **True** |
| (<A>, <B>)  (<F>, <F>) | | **False** |
| Dado los vértices del grafo  <A> <B> <C> <D> <E> <F>  Nodo a agregar= <G> | | | El aumenta la cantidad de vértices cuando el nodo ingresado se ha agregado al grafo, No hace nada en caso contrario. | addVertex(Graph,Vertex) | **Graph** | SceneSix () |
| <A> <B> <C> <D> <E> <F> | | **True** |
| <A> <B> <C> <D> <E> <F> <G> | | **False** |
| Dado el grafo:  Vertices:  <A> <B> <C> <D> <E> <F>  Aristas:  (<A>, <B>)  (<B>, <A>)  (<B>, <C>)  (<C>, <B>)  (<C>, <D>)  (<D>, <C>)  (<E>, <D>)  (<D>, <E>)  (<F>, <E>)  (<E>, <F>) | | | El método retorna los vértices cuando se ha realizado la búsqueda utilizando el modo de recorrido BFS. Retorna false si no se realizó correctamente el tipo de recorrido. | BFS() | **Graph** | SceneSeven() |
| <A> <C> <F> <E> <B> <D> | | **True** |
| <A> <D> <E> <F> <B> <C> | | **False** |
| Dado el grafo:  Vertices:  <A> <B> <C> <D> <E> <F>  Aristas:  (<A>, <B>)  (<B>, <A>)  (<B>, <C>)  (<C>, <B>)  (<C>, <D>)  (<D>, <C>)  (<E>, <D>)  (<D>, <E>)  (<F>, <E>)  (<E>, <F>) | | | El método retorna los vértices cuando se ha realizado la búsqueda utilizando el modo de recorrido DFS. Retorna false si no se realizó correctamente el tipo de recorrido. | DFS() | **Graph** | SceneEigh () |
| <A> <C> <F> <E> <B> <D> | | **True** |
| <A> <D> <F> <C> <B> <E> | | **False** |
| Dado el grafo:  Vertices:  <A> <B> <C> <D> <E> <F>  Aristas:  (<A>, <B>)  (<B>, <A>)  (<B>, <C>)  (<C>, <B>)  (<C>, <D>)  (<D>, <C>)  (<E>, <D>)  (<D>, <E>)  (<F>, <E>)  (<E>, <F>) | |  | El método retorna el arreglo de aristas las cuales se recorren siendo el camino más corto del grafo | dijkstra(Vertex) | **Graph** | SceneNine() |
| (<A>, <B>)(<B>, <C>)(<C>, <D>) | | **True** |
| (<A>, <C>)(<B>, <A>)(<C>, <D>) | | **False** |

# **Referencias**

Wikipedia. (27 de 09 de 2017). *Wikipedia*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/B%C3%BAsqueda\_en\_anchura

Wikipedia. (12 de 09 de 2017). *Wikipedia*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/B%C3%BAsqueda\_en\_profundidad

Wikipedia. (19 de 07 de 2017). *Wikipedia*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/Algoritmo\_de\_Bellman-Ford

Wikipedia. (10 de 10 de 2017). *Wikipedia*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/Algoritmo\_de\_Prim

wikipedia. (26 de 10 de 2018). *wikipedia*. Obtenido de wikipedia: https://es.wikipedia.org/wiki/Algoritmo\_de\_Kruskal

Wikipedia. (4 de 9 de 2018). *Wikipedia*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/Domicilio

Wikipedia. (11 de 10 de 2018). *Wikipedia*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/Algoritmo\_de\_Dijkstra

Wikipedia. (9 de 10 de 2018). *Wikipedia*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/Algoritmo\_de\_Floyd-Warshall#El\_algoritmo\_de\_Floyd

Wikipedia. (18 de 10 de 2018). *Wikipedia*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/Grafo