**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)

* **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)
* **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)

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. |

**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 |
|  | **True** | El método retorna true cuando el nodo ingresado pertenece al Grafo, y este es eliminado. retorna false, si el nodo no pertenece al Grafo, y por tanto no se puede eliminar. | removeVertex(Graph, Vertex) | **Graph** | SceneOne() |
|  | **False** |
|  | **True** | El método retorna true cuando el nodo ingresado no pertenece al grafo y es agregado. Retorna false en caso de pertenecer al grafo, y por tanto no se puede agregar de nuevo. | removeEdge(Graph,Edge) | **Graph** | SceneTwo() |
|  | **False** |
|  | **True** | El método retorna true cuando el nodo ingresado pertenece al grafo, retorna false cuando no pertenece. | contain(Graph,Vertex) | **Graph** | SceneThree() |
|  | **False** |
|  | **True** | 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() |
|  | **False** |
|  | **True** | Dado un grafo, el método retorna true si los dos vértices ingresados son adyacentes, retorta false si no lo son. | adjacent(Graph,Vertex,Vertex) | **Graph** | SceneFive() |
|  | **False** |
|  | **True** | El método retorna true cuando se ha agregado una arista del nodo ingresado, entre los dos nodos ingresados. Retorta false, en caso contrario. | addEdge(Graph,Vertex,Vertex) | **Graph** | SceneSix() |
|  | **False** |
|  | **True** | El método retorna true cuando se el nodo ingresado se ha agregado al grafo, retorta false en caso contrario. | addVertex(Graph,Vertex) | **Graph** | SceneSeven() |
|  | **False** |
|  | **True** | El método retorna true cuando se ha encontrado el recorrido con menor distancia, retorna false en caso contrario. | calculateLessDistance\_Dijkstra(Vertex,Vertex); |  | SceneEigth() |
|  | **False** |
|  | **True** | El método retorna true cuando se ha encontrado el punto de llegada, retorna false si el punto de llegada(nodo) no se encuentra en el grafo. | SearchNeighbordhood\_BFS(Graph, Vertex) |  | SceneNine() |
|  | **False** |