**PROYECTO FINAL AED**

**Planet Express**

**Contexto problemático:**

El mundo es un sitio armónico, con una población que ama viajar conocer y descubrir. Debido a que había grandes problemas con aerolíneas para el transporte entre ciudades, las personas se encuentran con algunos problemas a la hora de viajar por demoras en trasportes y acceso a las ciudades. Desafortunadamente existen pocas soluciones para mejorar la rapidez y mejoras de los transportes en el mundo.

Planet express es una empresa que maneja una extensa variedad de destinos en el mundo, que nacen de la necesidad de prestar un servicio de transporte de manera más rápida y eficiente alrededor del mundo, ya que luego del gran aumento de la población, los niveles de tráfico aumentaron considerablemente debido al masivo transporte aéreo. Esto lleva a la implementación de estrategias que mejoren este sistema de trayectos y distancias para transporte en el mundo. La misión de planet express es mejorar el transporte entre ciudades y mejorar el tiempo de cada vuelo para los usuarios y hacer posible un viaje más rápido, considerando las condiciones de tráfico y distancias.

**Identificación y definición concreta del problema:**

Los vuelos en el mundo requieren obtener una estrategia que mejore la manera de transportar usuarios de un lugar a otra con rapidez y recorriendo menor trayecto.

**Recopilación de Información**

1. **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.
2. **Multígrafo:** Un multígrafo o pseudografo es un grafo que está facultado para tener aristas múltiples; es decir, aristas que relacionan los mismos nodos. De esta forma, dos nodos pueden estar conectados por más de una arista.
3. **Grafo dirigido:** Un grafo dirigido o dígrafo es un tipo de grafo en el cual las aristas tienen un sentido definido, ​ a diferencia del grafo no dirigido, en el cual las aristas son relaciones simétricas y no apuntan en ningún sentido.
4. **Vuelo:** Es la acción de volar con cualquier movimiento o efecto a través del aire. También recibe el nombre de vuelo el desplazamiento de las naves espaciales más allá de la atmósfera terrestre. En el contexto es el trayecto que recorre un avión entre su punto de despegue y el destino final se denomina vuelo.
5. **Aerolínea:** Empresa o compañía dedicada al transporte aéreo.

**Especificación de requerimientos**

|  |  |
| --- | --- |
| **Nombre** | R1. Realizar la solicitud de viaje |
| **Resumen** | Permite al usuario seleccionar su ciudad de destino y la ciudad a la que desea llegar. |
| **Entradas** | lugar de salida y destino deseado. |
| **Salida** | Solicitud realizada con éxito. |

|  |  |
| --- | --- |
| **Nombre** | R2. Consultar disponibilidad de ciudades para viajar. |
| **Resumen** | El usuario puede saber a qué ciudades puede llegar desde la ciudad de origen. |
| **Entradas** | Ciudad de origen. |
| **Salida** | Lugares accesibles desde la ciudad seleccionada. |

|  |  |
| --- | --- |
| **Nombre** | R3. Mostrar ruta más corta. |
| **Resumen** | Muestra al usuario la ruta más óptima desde su salida hasta su destino. |
| **Entradas** | Ciudad de salida. |
| **Salida** | Ruta eficiente para el viaje. |

|  |  |
| --- | --- |
| **Nombre** | R4. Mostrar tráfico aéreo. |
| **Resumen** | Permite visualizar el tráfico aéreo en las rutas disponibles de la aérolinea en el mapa con su respectiva distancia. |
| **Entradas** | Ninguna. |
| **Salida** | Rutas de la aerolínea con su distancia y disponibilidad para volar. |

|  |  |
| --- | --- |
| **Nombre** | R5. Mostrar recorrido en anchura por el grafo. |
| **Resumen** | Muestra las ciudades del grafo en su respectivo orden (anchura). |
| **Entradas** | Ninguna. |
| **Salida** | Ciudades del recorrido. |

|  |  |
| --- | --- |
| **Nombre** | R6. Mostrar recorrido en profundidad por el grafo. |
| **Resumen** | Muestra las ciudades del grafo en su respectivo orden (profundidad). |
| **Entradas** | Ninguna. |
| **Salida** | Ciudades del recorrido. |

|  |  |
| --- | --- |
| **Nombre** | R7. Cargar un mapa. |
| **Resumen** | Permite cargar un mapa predeterminado con sus respectivas ciudades y rutas entre ellas. |
| **Entradas** | Archivo del mapa. |
| **Salida** | Ciudades con sus respectivas rutas en el mapa. (grafo). |

**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.

*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.

*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.

*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.

*Alternativa 5: 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.

*Alternativa 6: 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).

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

**Alternativa 1:**

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

**Alternativa 2:**

1. Puede utilizarse de forma recursiva, o en caso contrario, implementar una pila en su estructura.
2. 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.
3. Recorre el grafo desde la raíz hasta el extremo de una de sus ramas (Búsqueda en profundidad).

**Alternativa 3:**

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

**Alternativa 4:**

1. 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.
2. Se puede usar para detectar ciclos negativos.

**Alternativa 5:**

1. Permite encontrar el camino con menor distancia entre 2 nodos del grafo.
2. Se recomienda usar cuando hay aristas con peso negativo.
3. 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:**

1. 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.
2. El algoritmo incrementa continuamente el tamaño de un árbol.
3. Está completamente construido cuando no quedan más vértices por agregar.
4. El peso total de todas las aristas en el árbol es el mínimo posible.

**Alternativa 7:**

1. Si el grafo no es conexo, entonces busca un bosque expandido mínimo.
2. 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.

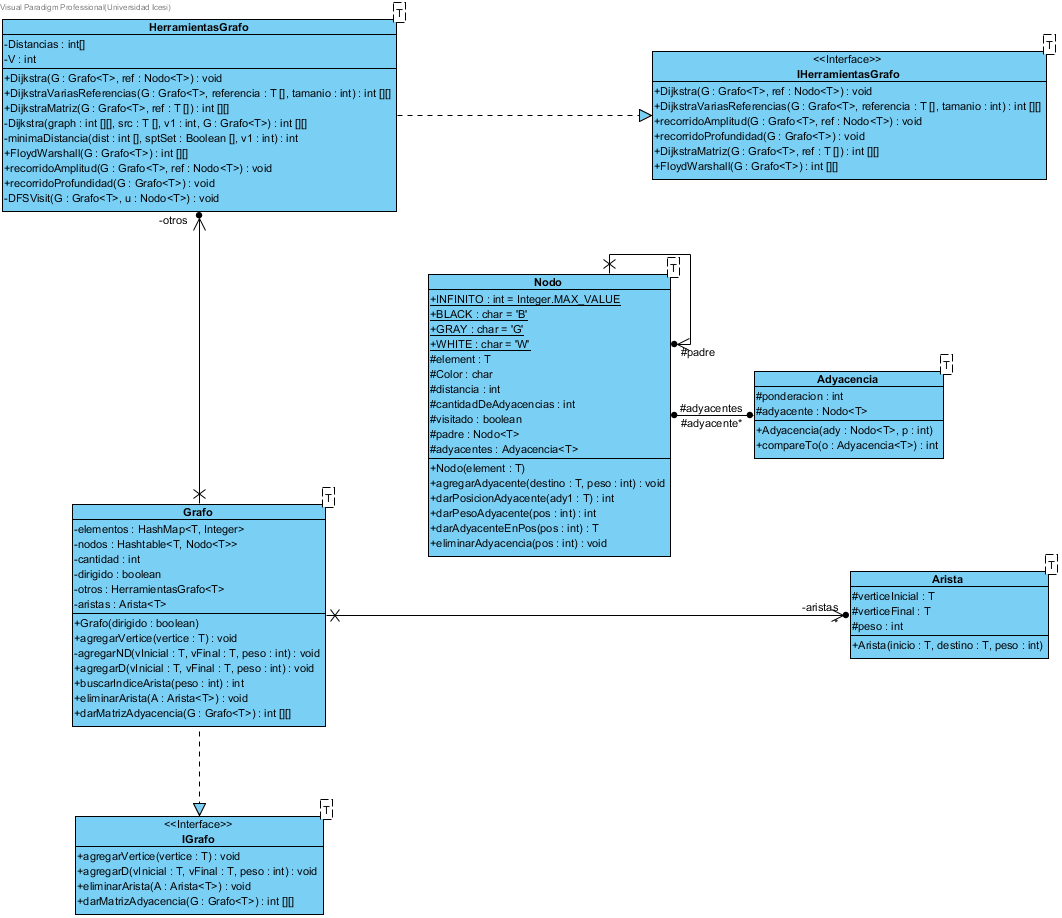
• [2]Normalidad en la implementación.

• [1]Dificultad para la implementación.

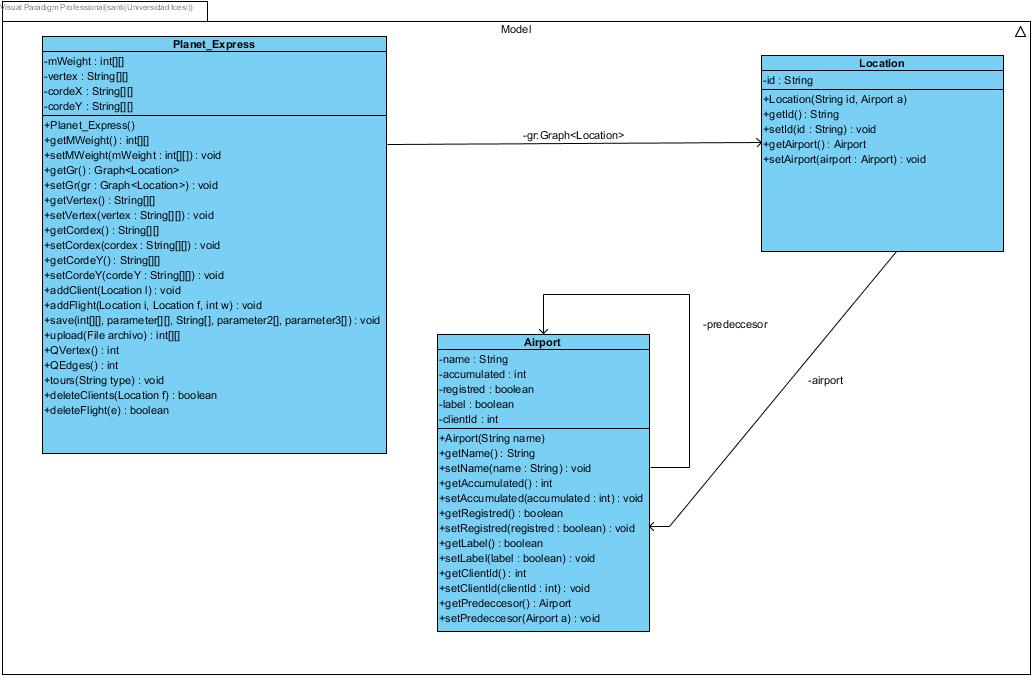
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algoritmos** | **Criterio A** | **Criterio B** | **Criterio C** | **Total** |
| **BFS** | Es siendo V el número de nodos que visita y E el número de aristas.  **8** | Es eficiente para muchos datos.  **2** | Es fácil de implementar.  **3** | **13** |
| **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** |
| **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.

**Diagrama grafo**

**Diagrama Modelo**

****

|  |  |
| --- | --- |
| **TAD** | **GRAFO** |
| C:\Users\fsociety\Pictures\grafo.png | |
| **Invariantes**  El grafo se define como **G = (V, E)** siendo **V** el conjunto de vértices y **E** un conjunto de aristas. | |
| Graph () 🡪 Constructor  createGraph() 🡪 Graph  addEdge(graph,node,node) 🡪 Graph 🡪 Edge  addNode (graph, node) 🡪 Graph 🡪 Node  deleteNode (graph, node) 🡪 Graph 🡪 Graph  deleteEdge (graph, node, node) 🡪 Graph 🡪 Graph  isEmpty () 🡪 Graph 🡪 boolean  contain () 🡪 Graph 🡪 boolean  adjacents(Graph, node, node)🡪 Graph 🡪 boolean  predecessor(Graph, node, node) 🡪 Graph🡪 boolean  successor (Graph, node, node) 🡪 Graph🡪 boolean | |
| ***public Graph createGraph ()***   * **Pre:** Se ha creado una instancia de un grafo. * **Post:** Se ha creado un nuevo grafo vacío. * **Def:** Crea un grafo vacío.   ***public void addEdge (graph, node, node)***   * **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 nodo1) del mismo grafo.   ***public void addNode (graph, node)***   * **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(node) en él.   ***public Graph deleteNode (graph, node)***   * **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 Graph deleteEdge (graph, node, node)***   * **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 boolean isEmpty(graph)***   * **Pre:** El grafo ha sido inicializado. * **Post:** Se ha retornado un booleano que determina si el grafo está vacío. * **Def:** Verifica si un grafo está vacío (no contiene nodos).   ***Public boolean contain (graph, node)***   * **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 adjacents (graph, node, node)***   * **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 predecessor (graph, node, node)***   * **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 successor (graph, node, node)***   * **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). | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Entrada** | **Salida** | **Descripción** | **Nombre del Método** | **Clase** | **Escenario** |
| <Cali> | **True** | El método disminuye la cantidad de vértices cuando la localización ingresado pertenece al Grafo, y este es eliminado. El método retorna falso si el nodo no pertenece al Grafo, y por tanto no se puede eliminar. | deleteVertex(Location) | **Graph** | SceneOne() |
| <Argentina > | **False** |
| <Cali>, <China> | **True** | El método disminuye la cantidad aristas cuando la arista ingresada pertenece al grafo y es eliminada. Retorna falso en caso de no pertenecer al grafo, y por tanto no se puede eliminar. | deleteEdge(Edge) | **Graph** | SceneTwo() |
| <Reino\_Unido>, <China> | **False** |
| **Dado el grafo:**  **Vértices:**  <Paris > <China> <Cali> | | El método retorna true cuando el nodo ingresado pertenece al grafo, retorna false cuando no pertenece. | contain(Vertex) | **Graph** | SceneThree() |
| <Paris> | **True** |
| <Argentina> | **False** |
| **Dado el grafo:**  **Vertices:**  **Aristas:** | | El método retorna los caminos mínimos para recorrer el grafo y dar solución al problema. | optimizeFlight (Graph,Location) | **Planet\_Express** | SceneFour() |
|  | |
| **Dado el grafo:**  **Vertices:** <Chile> <Francia> <Congo> <Dinamarca> <Inglaterra> <China>  **Aristas:**  **(<**China**>,** **<**Francia**>)**  **(<**Chile**>, <**Inglaterra**>)**  **(<**Dinamarca**>, <**Francia**>)**  **Crear una arista entre los vertices =** Congo**,** Inglaterra | | 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(Vertex1,vertex2, peso) | **Graph** | SceneFive() |
| **(<**Congo**>, <**Inglaterra**>)** | **True** |
| **(<**China**>, <**Congo**>)**  **(<**Inglaterra**>, <**Peru**>)** | **False** |
| **Dado los vértices del grafo  <**Argentina**> <**Venecia**> <**China**> <**Mexico**> <**Canada**> <**Egipto**>**  **Nodo a agregar= <**Guatemala**>** | | El aumenta la cantidad de vértices cuando el nodo ingresado se ha agregado al grafo, No hace nada en caso contrario. | addVertex(Vertex) | **Graph** | SceneSix () |
| **<**Argentina**> <**Venecia**> <**China**>,<**Mexico**> <**Canada**> ,null** | **False** |
| **<**Argentina**> <**Venecia**> <**China**>,<**Mexico**> <**Canada**>** | **True** |

**Implementacion**

<https://github.com/Jhusseth/Planet_Express_Proyecto.git>