Cristian Camilo Rivadeneira A00354996
Andrés Orozco Nuñez A00355202

CONTEXTO DEL PROBLEMA

Halo es un videojuego creado por la empresa Bungie en el año 2001 en este juego la humanidad debe enfrentarse a una alianza de alienígenas que buscan su extinción sin embargo la esta encuentra un artefacto conocido como aló dando así fin a la guerra entre la humanidad y esta facción alienígena sin embargo durante la batalla del Halo también conocido como instalación 04. Antes de descubrir la instalación 04 se desarrolló la batalla de Reach en el cual hubieron muchos contingentes Spartan, estos soldados están genéticamente modificados y poseen una armadura protectora.

Usted tiene el trabajo de ayudar a un integrante del equipo Noble con la programación de su sistema de interfaz. Como podrá inferir, Cortana es una IA que usa algoritmos para su funcionamiento y en este caso sus operaciones están definidas en base a la teoría de grafos. La misión principal es obtener la información necesaria del territorio de Valhalla y luego escapar por la ruta más corta en un recorrido lleno de aliados del Covenant.

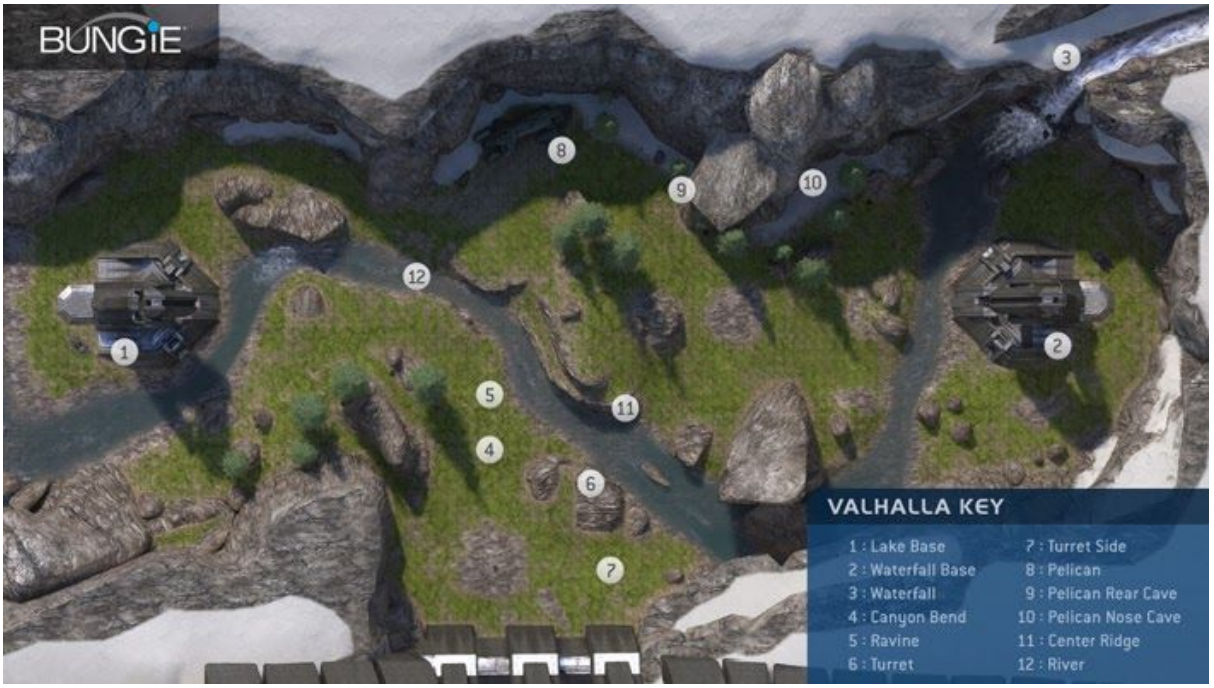


Figura 1. Mapa del Valhalla

IDENTIFICACIÓN DEL PROBLEMA

Necesidades:

1. Identificar los vértices y aristas
2. Identificar si el tipo de grafo
3. Identificar si tiene peso o no

Una vez identificado el tipo de grafo y si tiene peso o no se procede a identificar cuál de los siguientes algoritmos satisfacen la solución

Búsqueda en anchura y profundidad	DFS
	BFS
Camino Mínimo	Dijkstra
	Floyd Warshall
Árbol recubridor	Kruskal
	Prim

RECOPILACIÓN DE INFORMACIÓN

Teoría de Grafos:La **teoría de grafos**, también llamada **teoría de gráficas**, es una rama de las matemáticas y las ciencias de la computación que estudia las propiedades de los grafos. [1]

Grafo: Un grafo es una estructura de datos no lineal que consta de nodos y aristas. Los nodos a veces también se conocen como vértices y los edge son líneas o arcos que conectan dos nodos en el gráfico. Más formalmente, un grafo se puede definir como: “Un gráfico consiste en un conjunto finito de vértices (o nodos) y un conjunto de bordes que conectan un par de nodos.”[2]

Vértices:En teoría de grafos, un **vértice** o **nodo** es la unidad fundamental de la que están formados los grafos. Un grafo no dirigido está formado por un conjunto de vértices y un conjunto de aristas (pares no ordenados de vértices), mientras que un grafo dirigido está compuesto por un conjunto de vértices y un conjunto de **arcos** (pares ordenados de vértices). En este contexto, los vértices son tratados como objetos indivisibles y sin

[1] Tomado de https://es.wikipedia.org/wiki/Teor%C3%ADa_de_grafos
[2] https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/

propiedades, aunque puedan tener una estructura adicional dependiendo de la aplicación por la cual se usa el grafo; por ejemplo, una red semántica es un grafo en donde los vértices representan conceptos o clases de objetos.

Los dos vértices que conforman una arista se llaman **puntos finales** ("endpoints", en inglés), y esa arista se dice que es **incidente** a los vértices. Un vértice *w* es **adyacente** a otro vértice *v* si el grafo contiene una arista (*v*,*w*) que los une. La vecindad de un vértice *v* es un grafo inducido del grafo, formado por todos los vértices adyacentes a *v*.[3]

Aristas:Una **arista** corresponde a una relación entre dos vértices de un grafo. [4]

Caminos: Se llama **camino** a una secuencia de vértices dentro de un grafo tal que exista una arista entre cada vértice y el siguiente. Se dice que dos vértices están conectados si existe un camino que vaya de uno a otro, de lo contrario estarán desconectados. Dos vértices pueden estar conectados por varios caminos. El número de aristas dentro de un camino es su longitud. Así, los vértices adyacentes están conectados por un camino de longitud 1, y los segundos vecinos por un camino de longitud 2.Si un camino empieza y termina en el mismo vértice se le llama **ciclo**.[5]

La Programación de un grafo con interfaz Gráfica.

Para empezar con una base sólida se necesitan tomar ejemplos de diversas maneras de programar grafos (simples y complejos).

Un ejemplo con ciudades se encuentra aquí:
https://devs4j.com/2017/11/24/implementa-un-grafo-de-ciudades-en-java/

En caso de algoritmos más complejos, como por ejemplo: Kruskal se tuvo en cuenta sus consideraciones teóricas.

El algoritmo de kruskal, es un algoritmo voraz utilizado en la teoría de grafos, con el fin de encontrar un árbol recubridor mínimo de un grafo conexo y ponderado.

El algoritmo de kruskal consiste en:

- Paso 0: Iniciar el árbol T con n nodos y sin arcos

```
T=({1, 2, …n},∅)
```

- Paso 1: Con los arcos de G crear una lista L de arcos, en orden ascendente de peso. Los arcos con el mismo peso son ordenados arbitrariamente.

[3] https://es.wikipedia.org/wiki/V%C3%A9rtice_(teor%C3%ADa_de_grafos)

[4]

[5] https://es.wikibooks.org/wiki/Programaci%C3%B3n_en_Java/Ap%C3%A9ndices/Implementaci%C3%B3n_del_Algoritmo_de_Kruskal_en_Java

- Paso 2. Seleccionar el arco (i,j) que esté al comienzo de L. Si éste forma un circuito en T no se transfiere a T y se borra de L y se repite el paso 2. Si no forma circuito en T se transfiere a T y se borra de L.
- Paso 3. Si L es no vacío, volver al paso 2, de lo contrario PARAR.[6]

Dado que en la construcción del proyecto no es exacta a los códigos tradicionales es una buena idea explorar proyectos un poco más “vulgares” alojados en los repositorios de GitHub, en los cuales existen bastantes implementaciones que podrían ser de ayuda.

El funcionamiento en interfaz también es importante por lo que el API de java FX también será una referencia importante. Sobre todo en sus componentes de dibujo como canvas o primitivos como shapes[7]

BÚSQUEDA DE SOLUCIONES CREATIVAS

1. La primera propuesta que surge de una lluvia de ideas consiste en la búsqueda de un diseño implementado con distintos algoritmos sobre grafos para poder adaptar dicho código en uno que nos ayude en la solución de nuestro problema.

2. Lluvia de Ideas: Generar una interfaz donde el usuario pueda dibujar si grafo. Un modelo similar a “arrastrar” circulos y lineas para formar grafos.

3. Lluvia de Ideas: Generar una interfaz donde el grafo sea predeterminado, basado en la idea de que un mapa siempre tendrá puntos fijos para recorrer e inspirados en el problema de Dijkstra.

4. Lluvia de Ideas: Generar un grafo aleatorio y dar opciones sobre qué tipo de algoritmo se pueda aplicar.

5. Lluvia de Ideas: Crear un modelo simple pero funcional que muestra un recorrido basado en la premisa del problema y su integración con la temática de Halo.

6. Por una relación forzada: muerte-grafos, el punto consiste en la búsqueda de caminos seguros dado cierto tipo de grafo.

7. Por relación forzada: Crear una mini-consola para generar resultados.

8. Combinar la creación de un grafo con la interfaz (introducir lógica del negocio en la interfaz)

[6] https://es.wikipedia.org/wiki/Camino_(teor%C3%ADa_de_grafos)
[7] https://docs.oracle.com/javase/8/javafx/api/toc.htm

TRANSICIÓN A DISEÑOS PRELIMINARES

¿Que descartamos?

a) Generar una interfaz donde el usuario pueda dibujar si grafo. Un modelo similar a “arrastrar” circulos y lineas para formar grafos.

Para nuestro diseño, esto es inviable puesto que las librerías de java FX son bastantes complicadas de acoplar a una interfaz de dibujo-arrastre. Los modelos que construimos como prueba resultaban ser horriblemente codificados y con un montón de código espagueti. Consideramos también usar Swing, pero el diseño de cómo queríamos el programa no se veían bien estéticamente y nos parecieron anticuados.

b) Lluvia de Ideas: Generar un grafo aleatorio y dar opciones sobre qué tipo de algoritmo se pueda aplicar.

Si de por sí ya es complicado correr un algoritmo sobre un grafo indicado, lo es aún más tratar de correr varios algoritmos sobre el mismo. Una idea compleja y que no nos animamos a ejecutar. Además, la generación de grafos aleatorios con la interfaz sería una tarea aún más difícil.

c) Por una relación forzada: muerte-grafos, el punto consiste en la búsqueda de caminos seguros dado cierto tipo de grafo.

Descartada puesto que no logramos que un tipo de grafo se adaptara a esta idea.

d) Por relación forzada: Crear una mini-consola para generar resultados.

Las idea de imprimir la información de manera forzada en la interfaz gráfica surgió porque no sabíamos cómo mostrar los resultados sobre un grafo dibujado. Sin embargo la reformamos para ciertos datos que regresaban nuestros métodos.

e) Combinar la creación de un grafo con la interfaz (introducir lógica del negocio en la interfaz)

Según los métodos de mi compañero esto es posible. Pero una muy mala práctica de programación.

¿Que NO descartamos?

a) La primera propuesta que surge de una lluvia de ideas consiste en la búsqueda de un diseño implementado con distintos algoritmos sobre grafos para poder adaptar dicho código en uno que nos ayude en la solución de nuestro problema.

No descartamos esto debido a que nuestras implementaciones tenían bastantes limitaciones, fue entonces cuando decidimos buscar por código ya implementado y tratar de usarlo de acuerdo a nuestros intereses de diseño. Esto no significa que copiamos directamente el código (Aunque algunas partes si) tratamos de mantener nuestra implementación lo más fiel a nuestro conocimiento.

b) Lluvia de Ideas: Generar una interfaz donde el grafo sea predeterminado, basado en la idea de que un mapa siempre tendrá puntos fijos para recorrer e inspirados en el problema de Dijkstra.
c) Lluvia de Ideas: Crear un modelo simple pero funcional que muestra un recorrido basado en la premisa del problema y su integración con la temática de Halo.

Con la idea de evitar problemas con la interfaz gráfica, decidimos usar un grafo fijo, con la intención de probar los métodos de una manera eficiente y amigable con el usuario. Además, esto se adapta de manera bastante increíble a nuestro problema propuesto y nos pareció una forma interesante de codificar nuestro proyecto.

EVALUACIÓN Y SELECCIÓN DE LA MEJOR SOLUCIÓN

Criterios

- *Funcionalidad:* Los algoritmos funcionan como deberían.
- *Fácil implementación gráfica:* Los algoritmos se pueden implementar gráficamente.
- *Calidad de presentación:* Los algoritmos generan un programa agradable.
- *Mejores prácticas de programación:* Los algoritmos llevan buenas practicas de programacion.
- *Adaptabilidad*: Los algoritmos se pueden acoplar a otras funcionalidades como un Thread.

Evaluación.

a) búsqueda de un diseño implementado con distintos algoritmos sobre grafos para poder adaptar dicho código en uno que nos ayude en la solución de nuestro problema.

- *Funcionalidad:* **Excelente.**
- *Fácil implementación gráfica:* **Excelente.**
- *Calidad de presentación:* **Excelente.**
- *Mejores prácticas de programación:* **Regular.**
- *Adaptabilidad*: **Excelente.**

b) Generar una interfaz donde el grafo sea predeterminado, basado en la idea de que un mapa siempre tendrá puntos fijos para recorrer e inspirados en el problema de Dijkstra.

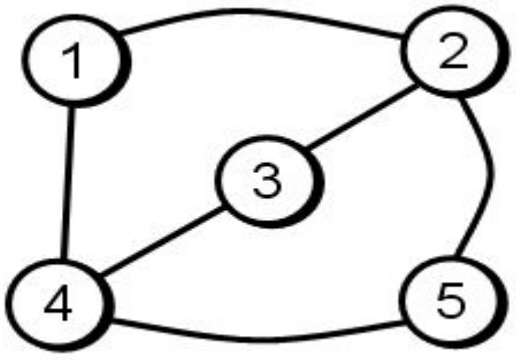
- *Funcionalidad:* **Regular.**
- *Fácil implementación gráfica:* **Excelente.**
- *Calidad de presentación:* **Excelente.**
- *Mejores prácticas de programación:* **Regular.**
- *Adaptabilidad*: **Regular.**

c) Crear un modelo simple pero funcional que muestra un recorrido basado en la premisa del problema y su integración con la temática de Halo

- *Funcionalidad:* **Excelente.**
- *Fácil implementación gráfica:* **Excelente.**
- *Calidad de presentación:* **Excelente.**
- *Mejores prácticas de programación:* **Excelente.**
- *Adaptabilidad*: **Regular.**

En este caso, decidimos que utilizaremos los apartes más importantes de cada idea para nuestra implementación, puesto que no podemos elegir una idea que solucione todos nuestros requerimientos de manera completa, así que gracias a nuestros criterios podemos elegir entre las opciones y combinarlas para obtener el mejor resultado posible.

TAD GRAFO



Inv: G = [(V , A) | G ≠ {}]

Operaciones Primitivas:

CrearGrafo		Grafo
AñadirVertice	Grafo x Vertice →	Grafo
AñadirArista	Grafo x Arista →	Grafo
TienePeso	Grafo →	Booleano
EsDirigido	Grafo →	Booleano
EliminarVertice	Grafo x Vertice →	Grafo
EliminarArista	Grafo x Arista →	Grafo

Descripción de operaciones

CrearGrafo()
“Se crea un nuevo grafo con al menos un vértice”
pre = {TRUE}
post = { Se crea un nuevo grafo con al menos un vértice}

AñadirArista()
“Dado un grafo, añade una relación entre dos nodos de dicho grafo.”
pre = {El grafo ha sido creado previamente}
post = {Se añade una arista al grafo existente}

AñadirVértice()
“Dado un grafo, incluye un nuevo vértice en él, en caso en el que no exista previamente”
pre = {El grafo ha sido creado previamente}
post = {Se añade un nuevo vértice al grafo}

TienePeso()
“Dado un grafo, verifica si dicho grafo tiene un peso.”
pre = {El grafo ha sido creado previamente}
post = {Verdadero en caso de que el grafo tenga peso, falso de lo contrario}

EsDirigido()
“Dado un grafo, verifica si dicho grafo es dirigido”
pre = {El grafo ha sido creado previamente}
post = {Verdadero en caso de que el grafo sea dirigido, falso de lo contrario}

EliminarArista()
“Dado un grafo, elimina una relación entre dos vértices de dicho grafo.”
pre = { La relación a eliminar existe}
post = {Se elimina la arista del grafo existente}

-

EliminarVertice()
“Dado un grafo, elimina un vértice en él”
pre = { El vértice a eliminar existe y el grafo tiene más de un vértice}
post = {Se elimina el vértice del grafo existente}

Especificación de requerimientos

RF #1	**Mostrar el recorrido de un grafo con BFS.**
Entrada	El vértice origen, el vértice destino.
Resumen	Permite mostrar el recorrido de un grafo cuando se aplica sobre este un algoritmo de búsqueda BFS.
Salida	Los vértices recorridos de la búsqueda en anchura.

RF #2	**Mostrar el camino más corto entre los vértices de un grafo.**
Entrada	El vértice origen, el vértice destino.
Resumen	Permite mostrar el camino más corto entre dos vértices de un grafo ponderado usando el algoritmo de Dijkstra.
Salida	El peso del camino más corto entre dichos vértices.

RNF #1	**Mostrar gráficamente el recorrido BFS**
Resumen	Permite que el usuario pueda visualizar de manera gráfica las operaciones que se llevan sobre el grafo al que se le aplique el BFS.

RNF #2	**Mostrar gráficamente el recorrido Dijkstra**
Resumen	Permite que el usuario pueda visualizar de manera gráfica las operaciones que se llevan sobre el grafo al que se le aplique el Dijkstra.