Universidad Nacional
de La Matanza

Reconquistando la Tierra de Fantasía

Programación Avanzada

Estudiantes

- DNI: 43.664.669 - Di Nicco Luis Demetrio
- DNI: 43.630.151 - Antonioli Iván Oscar

Docentes

- Dra. Verónica Aubín
- Ing. Federico Gasior
- Ing. Hernán Lanzillotta
- Ing. Lucas Videla

Comisión

- Jueves - Turno tarde

14 de noviembre de 2024

Tabla de Contenidos

Consigna

En esta aventura, acompañarás a una raza de guerreros en su misión por reconquistar la tierra que le ha sido arrebatada en anteriores batallas. Estos guerreros han recobrado fuerzas y disponen de un ejército que consideran suficientemente poderoso para emprender esta epopeya.

De camino a la tierra deseada se encontrarán con diversos poblados. Algunos de los cuales son hostiles, a los que deberán vencer para poder continuar el camino. Otros poblados son aliados, y permitirán que las tropas descansen y la mitad de su población se sumará a tu ejército.

Es importantísimo no desperdiciar recursos y, aún a riesgo de no encontrar poblados aliados, recorrer el camino más rápido entre su ubicación actual y la tierra destino: el tiempo apremia. Cada batalla o descanso, dura un día.

Se te buscó, hechicero del código, para que prepares unos conjuros algorítmicos que te permitan:

1. predecir si esta misión es factible, y
2. en caso de serlo:
    a. cuántos guerreros llegarían hasta el final del camino, y
    b. en cuánto tiempo.

Datos para resolver el problema

Razas

Un Wrives tiene una salud inicial de 108. Utiliza magia, y su rango de ataque es de 14 a 28 metros. Ocasiona un daño básico de 113 puntos. Cuando ataca, lo hace con 2 veces su daño, cada 2 ataques. Al recibir un ataque recibe 2 veces el daño, ya que no lleva armadura. Cuando descansa, medita, y como considera la violencia como algo malo, se rehúsa a atacar hasta que lo ataquen. Gracias a esto, aumenta su salud y su salud máxima en 50.

Una Reralopes tiene una salud inicial de 53. Utiliza una catapulta, y su rango de ataque es de 5 a 46 metros. Ocasiona un daño básico de 27 puntos. Cuando ataca, erra 2 de cada 4 ataques. Al recibir un ataque se desconcentra y sus ataques vuelven al valor normal inicial. Cuando descansa, se concentra y sus próximos 3 ataques (de esa unidad) dañan el doble del valor correspondiente.

Una Radaiteran tiene una salud inicial de 36. Utiliza shurikens, y su rango de ataque es de 17 a 41 metros. Ocasiona un daño básico de 56 puntos. Cuando ataca, lo hace cada vez con más fuerza (3 de daño extra x la cantidad de ataques dados). Al recibir un ataque lo hace normalmente. Cuando descansa, no le sucede nada.

Un Nortaichian tiene una salud inicial de 66. Utiliza un arco, y su rango de ataque es de 16 a 22 metros. Ocasiona un daño básico de 18 puntos. Cuando ataca, se cura un 4 por ciento de su salud. Al recibir un ataque se enfurece y sus ataques multiplican por 2 su daño (dura 2 turnos propios). Cuando descansa, recupera toda su salud, pero se vuelve de piedra por 2 turnos (contiguos), lo que hace que no pueda atacar, pero reduce el daño entrante en 1/2.

Batallas

- Las batallas en la Tierra de Fantasía se realizan de una manera muy ordenada:
- Se forman ambos ejércitos en línea. Nuestro ejército formará primero a las unidades aliadas, luego a las propias.
- La unidad que haya quedado herida de la batalla anterior siempre será la última en recibir ataques.
- Siempre comienza a atacar nuestro bando.
- Se turnan ambos ejércitos para atacarse.
- Al quedarse con la salud en cero, la unidad se desmaya y queda fuera de combate y no continúa la misión.
- Termina el combate cuando un ejército se queda sin contendientes de pie.

Poblados

Se suministrará un archivo con la información de los caminos que interconectan a los poblados, y los datos de dicho poblado. Por ejemplo:

```
4
1 100 Wrives propio
2 30 Reralopes aliado
3 40 Nortaichian enemigo
4 60 Nortaichian enemigo
1 -> 4
1 2 10
1 3 20
2 3 5
3 4 7
```

En ese archivo figura toda la información necesaria para la predicción:

- Una línea con la cantidad de pueblos (n, 4 en el ejemplo).
- N líneas auto numeradas, que representan cada pueblo, con el total de habitantes, la raza, y si es propio / aliado / enemigo. “Propio” será un único pueblo.
- Una línea que indica el pueblo inicial, y el pueblo final (1 -> 4).
- x líneas que indican la distancia entre cada par de pueblos, siendo estos datos pueblo de origen, pueblo destino, distancia en kilómetros.
- Dato de vital importancia para la trama: Una tropa avanza 10 kilómetros por día.

Condiciones

- Las razas deberán programarse utilizando pruebas. Se verificará una cobertura mayor al 92%.
- El camino más corto deberá calcularse utilizando un algoritmo de grafos apropiado. El mismo deberá programarse en términos de un grafo y no de los terrenos, para poder ser reutilizado en futuros usos.
- El mapa es único para todo el problema, y debe poder accederse a la misma instancia desde cualquier clase que lo requiera. Utilizar un patrón de diseño adecuado para este comportamiento.
- Tanto el ejército como la unidad individual deben poder tratarse de manera uniforme para el ataque, la recepción de golpes y el descanso. Utilizar un patrón de diseño adecuado para este comportamiento.

Resolución

Introducción

En la consigna se plantea proponer una solución para que una raza de guerreros lleguen al destino indicado, pasando por diferentes pueblos, los cuales pueden ser aliados o enemigos.

El trabajo consiste en indicar cómo sería la secuencia de batallas y el camino tomado por el pueblo inicial para llegar a destino. Se debe informar si el camino tomado llevó a la victoria o derrota; en caso de la victoria se indica cuántas tropas llegaron y cuanto tiempo les tomó. En el caso de la derrota se propone de ser posible otro camino que tenga el menor costo posible y que pueda conducir al ejército del poblado a la victoria.

Desarrollo

Diagrama de Clases

A continuación, presentamos el diagrama de clases para mostrar cómo está estructurado la solución propuesta.

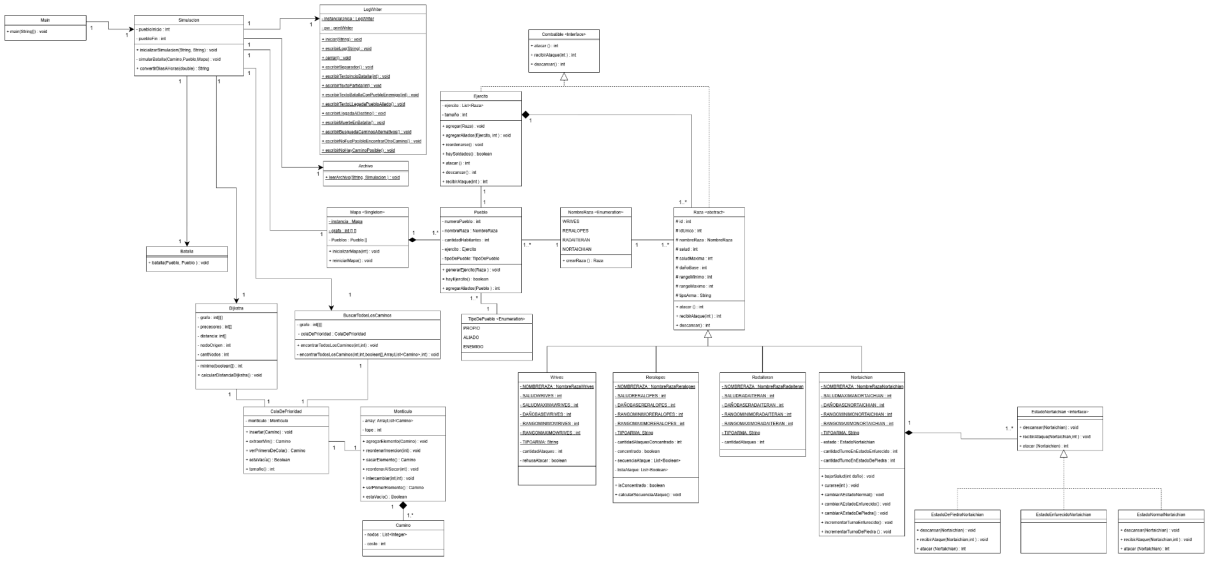

Imagen 1: Diagrama de clases

Decisiones de Diseño

Aspectos relevantes del problema

Los aspectos relevantes del problema que detectamos fueron los siguientes:

- Se pide encontrar el camino de menor costo (distancia) entre dos pueblos.
- Para el cálculo del tiempo total de la aventura, se debe considerar que las tropas del ejército recorren 10 kilómetros por día.
- Cada pueblo puede ser aliado o enemigo, lo que añade dificultad a la búsqueda del mejor camino posible, ya que el camino más óptimo puede no ser el más corto.
- Cada pueblo posee un ejército y estos se forman en fila.
- En caso de llegar a un pueblo aliado, nuestro ejército puede descansar y la mitad de las tropas del ejército de ese poblado se suman a nuestro ejército.
- En caso de llegar a un pueblo enemigo, nuestro ejército debe batallar contra el ejército del poblado para poder seguir con su camino.
- En las batallas, solo pelean un miembro de cada ejército simultáneamente. En caso de que al finalizar la batalla haya quedado algún soldado herido, este pasa al final de la fila.

Los aspectos del problema que decidimos ignorar fueron los siguientes:

- Los tipos de armas que utiliza cada tipo de raza.
- El rango mínimo y máximo de ataque de cada tipo de raza.

Opciones de Implementación y Alternativas Elegidas

Para llevar a cabo la implementación del algoritmo decidimos:

Respecto a la representación del mapa:

- Utilizar los conocimientos adquiridos sobre grafos y considerar cada ciudad como si fuera un Nodo o Vértice y cada camino como si fuera una arista o arco no dirigido ya que se puede transitar hacia ambos lados.
- Para garantizar la integridad de la información del mapa en todo momento de la ejecución del programa, decidimos utilizar el patrón singleton para conseguir un punto de acceso global. De esta manera, unificamos la información asociada a los nodos y aristas, evitando desincronizaciones y desactualizaciones.

Respecto al cálculo de la ruta mínima:

- Elegimos utilizar el algoritmo de dijkstra para encontrar el camino de coste mínimo entre el pueblo inicio y el pueblo de destino. Decidimos no implementarlo utilizando una cola de prioridad debido a que la estructura de los grafos puede ser variable (es decir, se puede recibir por archivo grafos ralos, con pocas aristas, o grafos densos, con muchas aristas). Consideramos que lo mejor era tomar el peor escenario, en donde todos los grafos que se reciban sean grafos densos. Por lo tanto, en base a esa suposición, lo mejor es implementar el algoritmo de Dijkstra en su implementación básica, sin utilizar la cola de prioridad. Como resultado, la complejidad computacional de nuestra implementación del algoritmo de Dijkstra es (V^2), siendo V la cantidad de nodos o pueblos.

Respecto a la herencia de las razas y la aplicación del polimorfismo:

- Considerando que las 4 diferentes razas del enunciado tiene atributos en común y comportamientos comunes, como puntos de salud, ataque, rango de ataque decidimos que era conveniente evitar repetir métodos o comportamientos en todas las clases y utilizar una clase padre Raza.
- En relación al polimorfismo lo aplicamos cuando las subclases (las razas Wrives, Reralopes, Radaiteran y Nortaichian) heredan de la clase Raza y sobreescriben los métodos de ataque, descansar y recibir ataque según en base a su comportamiento.

Respecto a los posibles estados de la raza Nortaichan:

- Decidimos utilizar el patrón de diseño State, el cual es un patrón de comportamiento que nos permite modificar el funcionamiento del objeto según el estado en el que se encuentra. En el caso de la raza Nortaichian identificamos 3 posibles estados:
    - Estado Normal:

        - ataque: al atacar ocasiona un daño de 18 puntos y se cura un 4% de su salud máxima.
        - recibir ataque: al recibir daño pasa a estado enfurecido.
        - descansar: recupera toda su salud pero pasa a estado de piedra.
    - Estado de Piedra:
        - ataque: no puede atacar debido a que se encuentra en un estado de piedra. Luego de dos turnos contiguos pasa a estado normal.
        - recibir ataque: el daño recibido es de la mitad del valor.
        - descansar: no puede recuperar su salud.
    - Estado Enfurecido:
        - ataque: al atacar ocasiona un daño del doble del valor base y se cura un 4% de su salud máxima. Luego de dos turnos propios pasa a estado normal.
        - recibir ataque: recibe el daño ocasionado por el enemigo.
        - descansar: recupera toda su salud pero pasa a estado de piedra.

    Como podemos ver el comportamiento de la raza Nortaichian varía o depende según el estado en el que se encuentra y por eso decidimos que era una buena opción implementar el patrón state.

Respecto al tratamiento del ejército:

- Debido a que se pide que tanto el ejército como la unidad individual se puedan tratar de manera uniforme para el ataque, la recepción de golpes y el descanso decidimos utilizar el patrón Composite. Este patrón estructural, propone definir una interfaz común tanto para los objetos contenedores (el ejército) como para los objetos contenidos (unidades individuales). De esta manera se simplifican las simulaciones de batallas, reduciendo la complejidad de manejar a cada unidad individual por separado.

Respecto al seguimiento del funcionamiento del programa:

- Originalmente, consideramos mostrar por consola todas las acciones y eventos sucedidos en la aventura (el pueblo de inicio, la ruta a seguir, los pueblos visitados, los descansos realizados, las batallas ocurridas, los caminos alternativos en caso de haber sido necesario y el resultado final aventura). Sin embargo, debido a la naturaleza de los combates y a la posibilidad de manejar ejércitos con decenas o centenas de unidades, quedaba una salida por consola demasiado extensa y difícil de entender. Por ejemplo, tomando el archivo proporcionado por la consigna, la salida por consola arrojaba más de mil líneas.
- Para simplificar la salida por consola y hacerla más simple y fácil de entender, decidimos generar un archivo log, que funcione como un recopilador de eventos ocurridos durante la aventura (pueblos visitados, batallas, descansos y resultado final de la aventura). De esta manera, en la salida por consola solo se incluyen los eventos principales como llegada a poblados, inicio de batallas o descansos

y los resultados finales. En caso de querer entrar en detalle en una batalla particular, en el archivo log se detallan cada una de las acciones llevadas a cabo por las unidades de los ejércitos.

Respecto a la búsqueda de caminos alternativos:

- Inicialmente, consideramos diferentes algoritmos para obtener rutas alternativas entre el pueblo de inicio y el pueblo de destino, como por ejemplo el algoritmo de búsqueda en profundidad (DFS) y el algoritmo de búsqueda en anchura (BFS). Si bien estos algoritmos cumplían en resolver el problema, decidimos investigar y buscar otros algoritmos con el fin de mejorar la complejidad computacional.

  Intentamos utilizar otros algoritmos vistos en la cursada tales como Dijkstra o Floyd, adaptándolos para el contexto de este problema pero no llegamos a un resultado positivo.

  También intentamos aplicar programación dinámica a la búsqueda en profundidad pero, debido a la naturaleza de los ejércitos, el estado en el que llegan a un pueblo (tamaño del ejército, razas que lo componen, y vida de cada una de las unidades) depende de la ruta seguida para llegar allí, lo que imposibilitaba almacenar resultados de batallas previas.

  Investigamos otros algoritmos más avanzados que no vimos en la cursada como el algoritmo A* o el algoritmo de KCaminos, sin embargo decidimos no utilizarlos debido a la dificultad que tuvimos para entenderlos y poder adaptarlos para la resolución de este problema.

- Finalmente, en caso de que el camino de mínimo coste termine en una derrota desarrollamos un algoritmo que calcula todos los posibles caminos desde el pueblo inicio hasta el pueblo de destino mediante la aplicación del algoritmo DFS, el cual modificamos aplicando backtracking para que por cada camino posible se busquen todos los nodos alcanzables hasta el pueblo destino y se tenga registro en cada uno de los nodos visitados.
    - Este algoritmo calcula todos los posibles caminos entre el nodo inicio y el destino, y los agrega a una cola de prioridad que luego utilizamos para realizar la simulación de la batalla.

Respecto a la construcción de los objetos de cada una de las razas:

- En un principio, debido a la herencia que implementamos en la clase Raza para generalizar la información y comportamiento comunes en cada una de las razas, utilizamos un bloque switch para instanciar cada tipo de raza concreta.
- Para mejorar la mantenibilidad y modularidad del código, decidimos delegar la construcción de los objetos dentro de una clase enum, lo que nos permite abstraernos de la complejidad de construir cada una de las razas concretas.

Análisis de la Complejidad Computacional

Para analizar la complejidad computacional de nuestra solución, se deben tener en cuenta 3 aspectos principales:

El primero, es el cálculo del algoritmo de Dijkstra para obtener la ruta principal. Este algoritmo, como vimos en clase, debido a que no utilizamos una cola de prioridad posee una complejidad computacional de $O(V^2)$, siendo V la cantidad de pueblos.

El segundo, es la simulación de la batalla. La complejidad computacional es O(N*T*A) siendo N la cantidad de pueblos que posee el camino (en el peor de los casos puede ser la cantidad total de pueblos, es decir, N = V), luego T son la cantidad de unidades del ejército aliado que participaron de la batalla (en el peor de los casos, T es igual al tamaño del ejército aliado) y A son las acciones que hace cada una de las unidades durante la batalla.

Por último, en términos de complejidad, la más significativa sería la del algoritmo para encontrar un camino alternativo que resulte en victoria se analizará en base a:

$E = Cantidad\ de\ aristas$

$V = Cantidad\ de\ nodos$

$C = Cantidad\ de\ caminos\ posibles\ entre\ el\ nodo\ inicio\ y\ el\ nodo\ destino$

El algoritmo realiza una búsqueda en profundidad para cada camino lo cual nos da una complejidad computacional de:

$$O(C * (E + V))$$

Siendo E+V el costo de realizar la búsqueda en profundidad de un solo camino, multiplicado por C que es la cantidad de caminos posibles entre el nodo inicio y el nodo destino.

En el peor de los casos, si se tiene un grafo completamente conexo, los posibles caminos entre dos nodos pueden calcularse como la combinatoria entre (V-2 nodos)!, restando 2 debido a que no tenemos en cuenta el nodo inicio y fin. De esta forma C = V-2! y por lo tanto:

$$O((V - 2)! * (E + V))$$

Sacando las constantes, la complejidad queda:

$$O((V)! * (E + V))$$

A su vez en un grafo completamente conexo la cantidad de aristas puede calcularse como $E = (V * (V - 1))/2$ y finalmente la complejidad computacional en el peor caso podemos expresarla como:

$$O(V! * (V^2 + V))$$

Complejidad computacional resultante del algoritmo para encontrar el camino que resulte en éxito de menor costo:

$$O(V! * V^2)$$

Por regla de la suma, podemos concluir que la complejidad computacional de nuestro algoritmo es $O(V! * V^2)$. Si se desea disminuir la complejidad computacional, se podría no realizar la búsqueda de los caminos mínimos alternativos, lo que dejaría a nuestro programa con una complejidad computacional de O(N*T*A)

Limitaciones:

Como podemos ver en la siguiente foto, este algoritmo no es el más óptimo debido a su complejidad, pero luego de analizar varias formas de implementarlo quisimos agregarlo para mostrar que se puede encontrar la solución a pesar de que no logramos optimizarla o desarrollarla con una complejidad aceptable.

En la foto podemos ver en un ejemplo de 13 pueblos conectados todos entre sí como no se logra obtener los caminos debido a exceder la memoria asignada durante la ejecución del algoritmo. El límite para una ejecución exitosa es de 12 pueblos totalmente conectados.

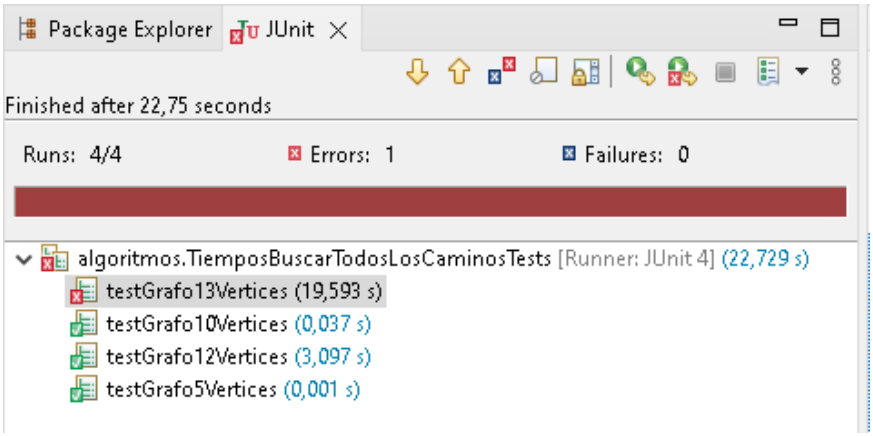

Imagen 2: Análisis de Tiempos para encontrar una solución alternativa.

Organización y distribución del trabajo

El trabajo lo fuimos haciendo en conjunto a lo largo del cuatrimestre. La mayor parte del desarrollo fue hecho en forma sincrónica, trabajando en conjunto para avanzar a la par, consultarnos dudas y definir criterios comunes. En menor medida, utilizamos un repositorio de github para compartir el avance que cada uno fue haciendo por separado.

Los pasos que fuimos realizando para desarrollar el trabajo fueron:

1. Pensamos que algoritmo se adecuaba mejor para encontrar el mínimo camino.
2. Comenzamos a desarrollar el código dividiendo las responsabilidades identificadas en distintas clases.
3. Realizamos la simulación de las clases utilizando el algoritmo de dijkstra para ver si funcionaba correctamente.
4. Revisamos el comportamiento y la estructura del código así como lo pedido en el enunciado para aplicar los patrones de diseño explicados anteriormente.
5. Evaluamos distintas alternativas para encontrar el camino más óptimo que resulte en éxito. Implementamos el algoritmo con menor complejidad computacional.
6. Realizamos test de las distintas clases del proyecto para asegurarnos que su funcionamiento era correcto.
7. Realizamos una revisión final de la estructura del código para cumplir los principios de programación orientada a objetos y modularidad.
8. Por último, validamos el código con las rúbricas para la entrega del código del trabajo práctico subidas en dojo.

Seguimiento de un ejemplo

A continuación presentamos un breve seguimiento para el ejemplo propuesto en la consigna de como es el funcionamiento de nuestra solución.

Algoritmo de camino mínimo

Primero, realizamos una lectura del archivo de entrada para representar en un grafo los poblados y los caminos.

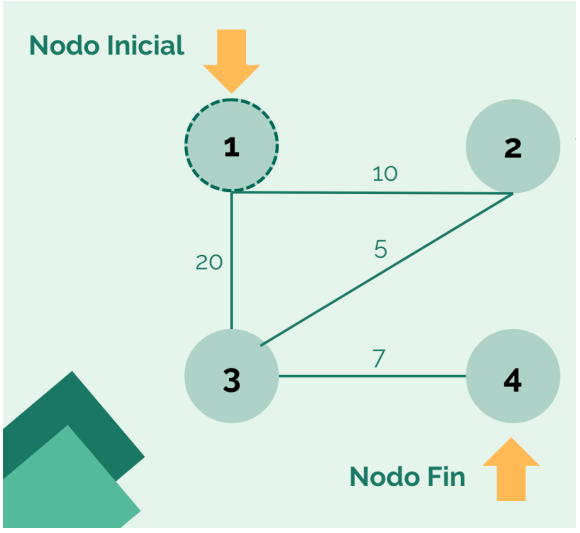

Imagen 3: Mapa inicial recibido por archivo, representado en forma de grafo

Una vez obtenido el grafo, aplicamos el algoritmo de Dijkstra para obtener el camino de menor distancia entre el pueblo inicio y el pueblo fin.

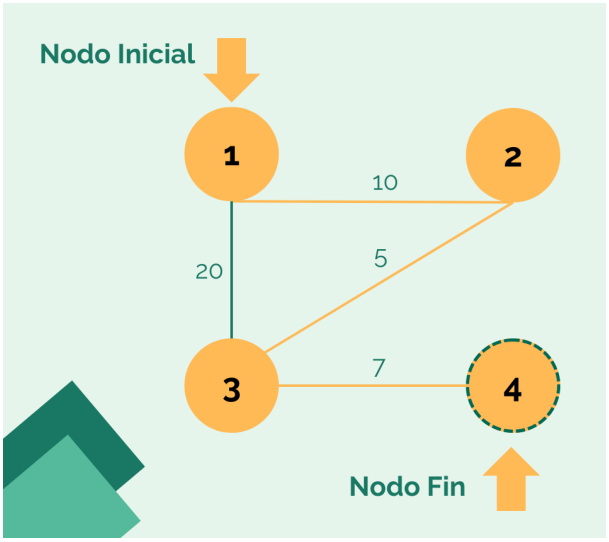

Imagen 4: Camino de distancia mínima, entre el nodo inicio y el nodo fin, obtenido con el algoritmo de Dijkstra

```
------------------------------------------------------------------------
                        Inicio de la Aventura
------------------------------------------------------------------------

Inicia una nueva aventura para el pueblo: 1...
Su objetivo? ~Reconquistar la tierra de fantasia~
Lograrán nuestros héroes llegar al pueblo: 4?

La ruta de menor costo a seguir es:
[1, 2, 3, 4]

Comienza la aventura!
Partimos desde el pueblo: 1

------------------------------------------------------------------------
```

Imagen 5: Salida por consola de la ruta mínima entre el pueblo inicio y el pueblo fin.

Luego, seguimos la ruta minima, visitando cada uno de los pueblos y simulando los descansos o batallas.

```
------------------------------------------------------------------------
Partiendo hacia el pueblo: 2...
Resultó ser un pueblo aliado, podremos descansar!
La mitad del ejercito del poblado se sumará a nuestro ejercito!
Se sumará la mitad de su ejercito
    -Reralopes [350]  se ha concentrado.
    -Reralopes [349]  se ha concentrado.
    -Reralopes [348]  se ha concentrado.
    -Reralopes [347]  se ha concentrado.
    -Reralopes [346]  se ha concentrado.
    -Reralopes [345]  se ha concentrado.
    -Reralopes [344]  se ha concentrado.
```

Imagen 6: Log de Simulación al visitar un pueblo aliado.

```
Partiendo hacia el pueblo: 3...
Resultó ser un pueblo enemigo, debemos vencerlos para poder avanzar!
Comienza la batalla en el pueblo: 3
                    ----------------------------------
                                Turno: 1
                    ----------------------------------
Ataque de nuestro ejercito:
    -Reralopes [350]  ataca con 54 puntos de daño!
        --Nortaichian [367] recibe 54 puntos de daño. Salud restante: 12
Ataque del ejercito enemigo:
    -Nortaichian [367] atacó haciendo 36 de daño.
        --Nortaichian [367] se cura 2 puntos. Salud actual: 14
        --Reralopes [350]  recibe 36 puntos de daño. Salud restante: 17
                    ----------------------------------
                                Turno: 2
                    ----------------------------------
Ataque de nuestro ejercito:
    -Reralopes [350]  fallo su ataque, no hizo daño!
        --Nortaichian [367] recibe 0 puntos de daño. Salud restante: 14
Ataque del ejercito enemigo:
```

Imagen 7: Log de Simulación al visitar un pueblo enemigo y tener una batalla..

Finalmente, en caso de que el ejército haya llegado con vida al pueblo destino, se muestra un resumen de la cantidad de unidades que sobrevivieron, el tiempo que les tomó llegar hasta allí, y la distancia recorrida en kilómetros.

```
----------------------  Final del camino Orignal  ----------------------

El ejército del pueblo 1 llegó a destino.
Sobrevivieron 71 soldados!
Luego de 5 días y 5 horas  llegamos al destino
Tuvieron que recorrer 22 kilometros.

------------------------------------------------------------------------
                        Fin de la aventura
------------------------------------------------------------------------
```

Imagen 8: Salida por consola del resultado final de la aventura.

En caso de que el camino conduzca al pueblo a una derrota se evaluarán si existen posibles rutas alternativas y se verificará si al elegirlas se puede alcanzar la victoria.

Solución alternativa

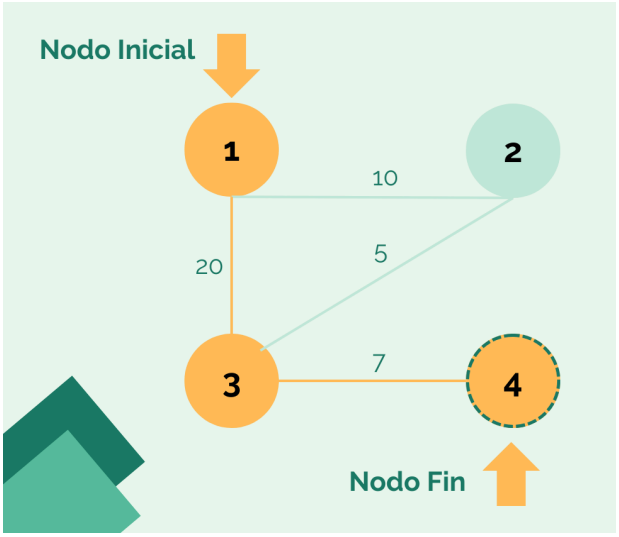

Imagen 9: Camino de distancia mínima, entre el nodo inicio y el nodo fin, obtenido con el algoritmo de Dijkstra

En caso de que el camino mínimo termine en una derrota, se recalcularán los caminos alternativos, siguiendo con el ejemplo anterior el segundo camino mínimo sería 1 - 3 - 4, y en ese caso se evaluaría si siguiéndolo se llega a la victoria o derrota.

Conclusiones

En conclusión, a lo largo de este trabajo práctico, pudimos aplicar los conocimientos adquiridos en las materias de programación avanzada y paradigmas de programación. Este trabajo integró conceptos clave como estructuras de datos, lectura y escritura de archivos, algoritmos de grafos, programación orientada a objetos, herencia, polimorfismo y complejidad computacional.

Gracias a la consigna planteada, pudimos analizar en mayor profundidad los algoritmos de grafos vistos durante la cursada, evaluando sus ventajas y desventajas, para saber cuáles eran las mejores alternativas que podíamos utilizar.

Respecto al desarrollo de la solución principal, pudimos entender mejor el funcionamiento del algoritmo de Dijkstra. También, al tener que implementar patrones de diseño pudimos terminar de entender su funcionamiento y las ventajas que estos ofrecen, ya que de no haberlos aplicados la complejidad del código hubiese aumentado mucho.

Respecto a la búsqueda de la solución de los caminos alternativos, esa problemática nos permitió investigar sobre distintos algoritmos sobre recorridos de grafos. La solución alternativa que planteamos, nos hizo entender la importancia de tener en cuenta la complejidad computacional a la hora de evaluar un algoritmo, ya que no alcanza solo que funcione bien, sino que las soluciones deben ser eficientes en el uso de recursos.

Finalmente, pudimos poner en práctica conceptos vistos de manera teórica en otras materias, como por ejemplo la importancia de la modularidad y mantenibilidad del código. Esto combinado con lo aprendido durante la cursada de la materia creemos que nos ayudó a asentar los conocimientos sobre cómo evaluar y determinar posibles soluciones a los problemas de manera eficiente.

Bibliografía

Aho, A. V., Hopcroft, J. E., y Ullman, J. D. (1998). *Estructuras de datos y algoritmos*. Addison Wesley Longman.

Aubin, V. I., y Videla, L. (2023a). *Análisis de Algoritmos* [Apuntes de cátedra]. Cátedra de Programación Avanzada, Universidad Nacional de La Matanza.

Aubin, V. I., y Videla, L. (2023b). *Grafos - Dijkstra* [Apuntes de cátedra]. Cátedra de Programación Avanzada, Universidad Nacional de La Matanza.

Aubin, V. I., y Videla, L. (2023c). *Grafos - Generalidades* [Apuntes de cátedra]. Cátedra de Programación Avanzada, Universidad Nacional de La Matanza.

Refactoring Guru. (s. f.-a). *Composite* [Patrón de diseño]. Recuperado el 12 de noviembre de 2024, de https://refactoring.guru/design-patterns/composite

Refactoring Guru. (s. f.-b). *State* [Patrón de diseño]. Recuperado el 12 de noviembre de 2024, de https://refactoring.guru/design-patterns/state

Weiss, M. A. (1995). *Estructuras de datos y algoritmos*. Addison Wesley Iberoamericana