

Proyecto - RISK



Juan Diego Echeverry Plazas

Nicolas Rincón Ballesteros

Santiago Yañez Barajas

PRESENTADO A:

Alejandro Castro Martinez

Decanatura Facultad de ingeniería

Carrera de Ingeniería de Sistemas

ESTRUCTURAS DE DATOS

PONTIFICIA UNIVERSIDAD JAVERIANA

BOGOTÁ D.C

2023

Tabla de Contenido

Primera Entrega	3
Descripción de entradas, salidas y condiciones	3
Procedimiento Principal	3
Operaciones Auxiliares (Comandos)	9
Diseño de TAD'S	10
TAD JUGADOR	10
TAD CARTA	11
TAD PAÍS	11
TAD CONTINENTE	11
TAD PARTIDA	12
Diagrama TAD'S	13
Planes de Prueba	13
Inicializar	13
Segunda Entrega	16
Acta de evaluación	16
Documento de diseño	17
Descripción de entradas, salidas y condiciones:	17
Procedimiento Principal	17
Guardar Texto Plano	17
Guardar Comprimido	18
Inicializar partida de archivo	19
Diseño de TAD's	20
TAD ArbolH	20
TAD NodoH	21
TAD Menú	21
Diagrama TAD's Árbol Huffman de Codificación	22
Diagrama TAD's actualizado	22
Plan de pruebas	23
guardar_comprimido	23
Tercera Entrega:	24
Acta de Evaluación	24
Descripción de entradas, salidas y condiciones:	26
Procedimiento principal	26
Costo conquista	26
Conquista más barata	27
Diseño de TAD's	27
TAD Grafo	27
Diagrama TAD's actualizado	28
Plan de pruebas	28
Costo conquista	28

Primera Entrega

Descripción de entradas, salidas y condiciones

Procedimiento Principal

Antes que nada, es importante entender la forma en la que guardamos los datos en la aplicación, para esto, se decidió crear el siguiente diagrama:

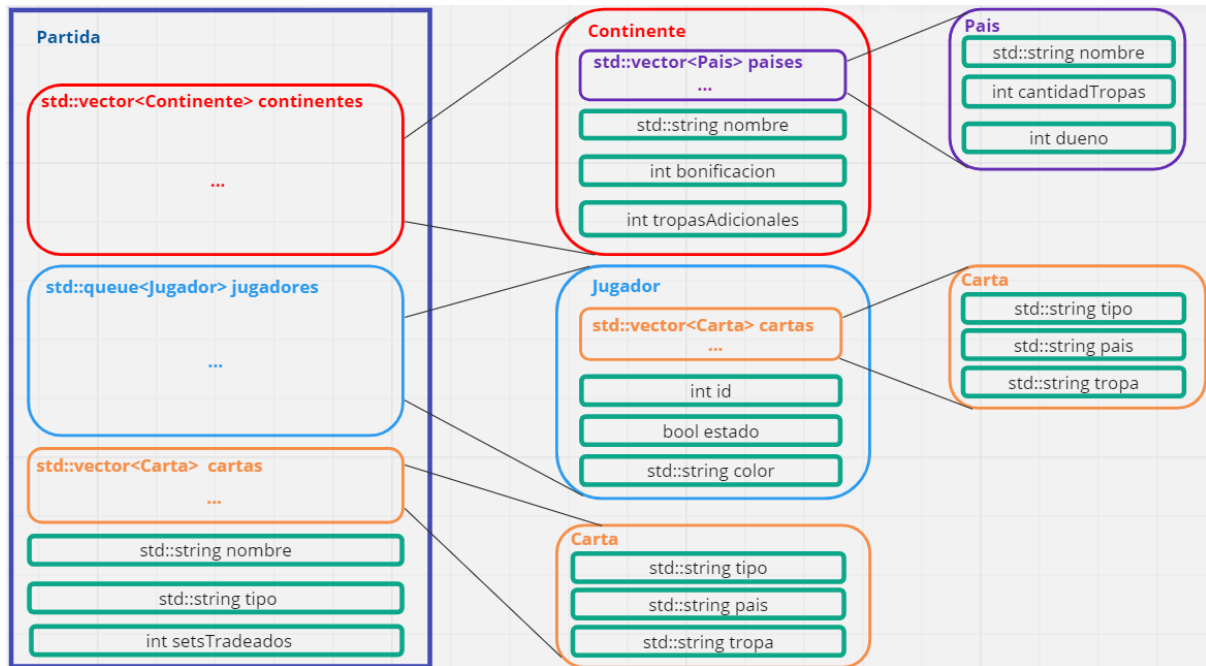


Figura 1. Estructura de datos. Fuente: Elaboración propia.

Como se puede ver, se tiene una estructura llamada *Partida*, de tipo *Partida* la cual tiene como atributos, nombre, tipo, setsTradeados, un vector de tipo *Continente*, una queue de tipo *Jugador*, un vector de tipo *Carta* y un vector de tipo *Dado*. Cabe recalcar que cada partida es un juego de RISK diferente. La siguiente es la explicación de cada uno de los atributos de *Partida*.

- **Nombre:** Este es el nombre con el cual se guardará la partida, este es ingresado por el usuario al momento de iniciar la partida.
- **Tipo:** Este es el tipo de la partida, este es ingresado por el usuario al momento de iniciar la partida. Tiene dos posibles tipos: Normal o MisionSecreta.
- **SetsTradeados:** Al momento de crear la partida, este valor inicia en 0 por defecto. Este representa el número de sets tradeados en la partida para así saber cuántas cartas serán asignadas cuando un jugador complete ciertas cartas.
- **Vector de tipo Continente:** Al momento de crear la partida, se cargaran todos los continentes con su respectivo nombre, vector de tipo Pais, con su respectivo nombre y cantidad de tropas que se encuentran en ese país en específico y finalmente, por cada Continente, un número entero tropasAdicionales que representa el número de tropas otorgadas si un jugador es dueño de todo el continente.

- Queue de tipo Jugador: Al momento de crear la partida, se le pregunta al usuario la cantidad de jugadores, a cada uno de estos se le asigna un id representando su turno, booleano de estado indicando si sigue en la partida o ya fue eliminado, un color que se le pregunta al usuario, un vector de strings con el nombre de países que actualmente domina y un vector de cartas de tipo Carta con toda su información necesaria.
- Vector de tipo Carta: Al momento de crear la partida, se crean todas las cartas con su respectivo tipo, país y tropa para posteriormente ser asignado a los jugadores.

Tras entender la estructura de cada Partida, se muestra el funcionamiento de los siguientes comandos de manera gráfica:

Comando inicializar:

Para entender mejor cómo es el funcionamiento del comando inicializar, se decidió realizar el siguiente diagrama de flujo:

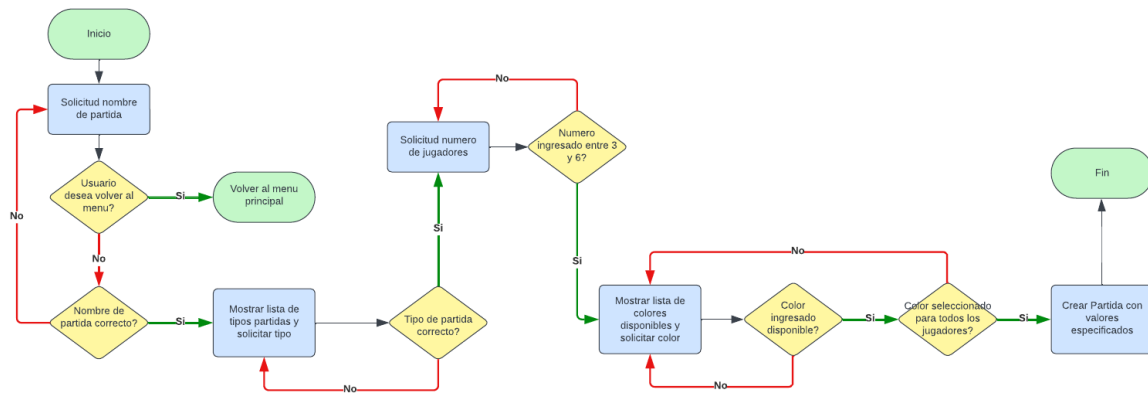


Figura 2. Diagrama de flujo inicializar. Fuente: Elaboración propia.

Cabe recalcar que el usuario puede escribir salir en cualquier momento del flujo de interacción para volver al menú principal si cometió algún error, descartando los cambios realizados.

Tras entrar al comando inicializar, el sistema hace una solicitud para que el usuario ingrese el nombre de la partida a crear, pueden surgir las siguientes cosas:

- Nombre de Partida Vacío: El sistema rechaza el nombre porque envió un espacio y se imprime el siguiente mensaje de error: *“El nombre de la partida no puede contener solo espacios en blanco”*. Se vuelve a pedir el nombre de la partida.
- Nombre de Partida con Espacios: El sistema rechaza el nombre porque envía dos argumentos y se imprime el siguiente mensaje de error: *“El nombre de la partida no puede estar separada por espacios”*. Se vuelve a pedir el nombre de la partida.
- Usuario presiona ‘Enter’: El sistema vuelve a pedir el nombre de la partida.

Una vez aceptado el nombre para crear la partida, se muestra una lista de los tipos de partida para que el usuario seleccione. El sistema hace la solicitud para que el usuario ingrese el nombre del tipo de la partida, pueden surgir las siguientes cosas:

- Tipo con espacios: El sistema rechaza lo ingresado por el usuario y se imprime el siguiente mensaje de error: *“Ingrese solamente el tipo de la partida”*. El sistema vuelve a mostrar la lista de los tipos de partida y vuelve a pedir el tipo de la partida.
- Tipo inexistente: El sistema rechaza lo ingresado por el usuario y se imprime el siguiente mensaje de error: *“Tipo partida: ‘ ‘ no encontrado. Por favor ingrese un tipo de partida valido”*. El sistema vuelve a mostrar la lista de los tipos de partida y vuelve a pedir el tipo de la partida.
- Usuario presiona ‘Enter’: El sistema vuelve a mostrar la lista de los tipos de partida y vuelve a pedir el tipo de la partida.

Una vez aceptada el tipo de partida, el sistema hace la solicitud para que el usuario ingrese el número de jugadores para la partida, pueden surgir las siguientes cosas:

- Número con espacios: El sistema rechaza lo ingresado por el usuario porque ingresó dos argumentos e imprime el siguiente mensaje de error: *“Ingrese solamente la cantidad de jugadores”*. El sistema vuelve a pedir el número de jugadores.
- Número incorrecto: El sistema rechaza lo ingresado por el usuario porque el número de jugadores debe estar entre 3 y 6. Se imprime el siguiente mensaje de error: *“Error. Debe ingresar entre 3 a 6 jugadores”*. El sistema vuelve a pedir el número de jugadores.
- No ingresa número: El sistema rechaza lo ingresado por el usuario porque no ingresa un dígito y se imprime el siguiente mensaje de error: *“Error. Debe ingresar el número de jugadores para la partida”*. El sistema vuelve a pedir el número de jugadores.
- Usuario presiona ‘Enter’: El sistema vuelve a pedir el número de jugadores.

Una vez aceptados el número de jugadores, el sistema muestra una lista de los colores disponibles hasta que se seleccionen para todos los jugadores creados previamente, pueden surgir las siguientes cosas:

- Ingresa color inexistente: El sistema rechaza lo ingresado por el usuario y se imprime el siguiente mensaje *“Color ‘ ‘ no encontrado o ya fue seleccionado. Por favor ingrese un color válido”*. El sistema vuelve a mostrar la lista de colores posibles para que el usuario seleccione.
- Ingresa color previamente seleccionado: El sistema rechaza lo ingresado por el usuario y se imprime el siguiente mensaje *“Color ‘ ‘ no encontrado o ya fue seleccionado. Por favor ingrese un color válido”*. El sistema vuelve a mostrar la lista de colores posibles para que el usuario seleccione.
- Usuario presiona ‘Enter’: El sistema vuelve a mostrar la lista de colores posibles para que el usuario seleccione.

Una vez que se ha solicitado la información necesaria, que comprende el nombre de la partida, el tipo de partida, la cantidad de jugadores y el color asignado a cada uno, el sistema procederá a asignar los países a cada jugador. Este proceso se llevará a cabo mediante un funcionamiento automatizado que implica la distribución aleatoria de los países y tropas de manera equitativa por parte del sistema.

Para lograr la distribución aleatoria de los países el sistema primero genera un número aleatorio entre la cantidad de jugadores que hay en la partida, por ejemplo, si hay 4 jugadores, es un número entre el 1 y el 4. Posteriormente, la lógica que se utiliza es tener un vector auxiliar de enteros llamado “vectorJugadoresIguales” el cual tendrá el número de posiciones como tantos jugadores haya en la partida. En este, se va a ir guardando el número aleatorio generado, sin embargo, este solo puede aparecer una vez porque se va a ir asignando de a 4 países a la vez, de forma que el primer número (representado el id de cada jugador) que se genere es quien va a recibir el primer país, el segundo número que salga será quien recibirá el segundo país y de esta forma hasta completar el número de jugadores en la partida.

Para representar esto mejor, se crea el siguiente diagrama el cual es el vector “vectorJugadoresIguales”:



Figura 3. Vector vectorJugadoresIguales. Fuente: Elaboración propia.

En este caso, ya salieron los siguientes números aleatorios: 3,1 y 4. Cabe mencionar que cada que se encuentre un número y se agregue a este vector, se creará el respectivo país con nombre y continentes ya guardados en otro vector.

Si vuelve a salir el 3, este se rechaza porque este vector se utiliza para asegurar que todos los jugadores tengan igual cantidad de países (en los casos que sea posible). Una vez se genere el valor aleatorio 2, este se agrega al vector.

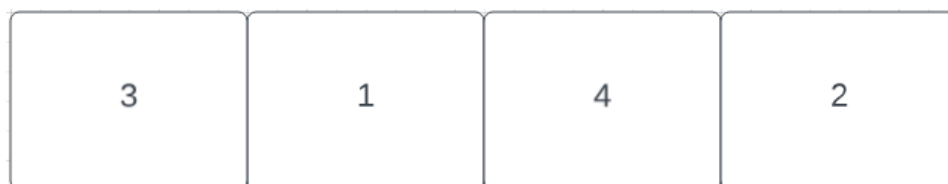


Figura 4. Vector tras encontrar el último valor restante. Fuente: Elaboración propia.

Una vez se encuentre que el tamaño del vector sea igual al tamaño de jugadores, a este vector se le hará .clear() para volver a iniciar el proceso de selección.

Comando turno:

Este comando recibe un id de turno para el cual se va a efectuar el turno. Para entender el funcionamiento de cómo se tratan los turnos en la aplicación, se muestra el siguiente gráfico que representa un ejemplo de una cola con 5 jugadores:

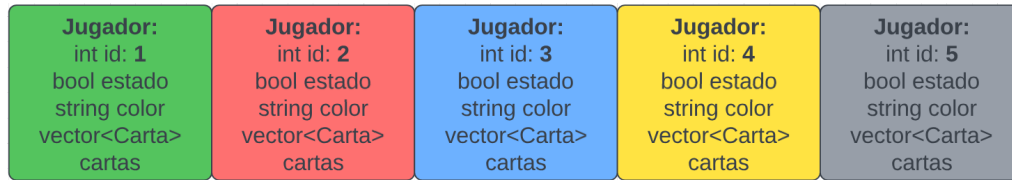


Figura 5. Cola de jugadores. Fuente: Elaboración propia.

Primero, se crea una variable tipo Jugador auxiliar para guardar el frente de la cola y utilizar el método ObtenerId() para compararlo con el que llegó como parámetro, si este no coincide, quiere decir que no es turno de ese jugador y se rechaza la petición.

Si el turno de jugador es correcto, se ejecutan las tres fases correspondientes del turno, se hace .pop() de la cola de Jugadores y se hace .push() para agregarlo al final de la cola. La siguiente es la representación visual de esto:

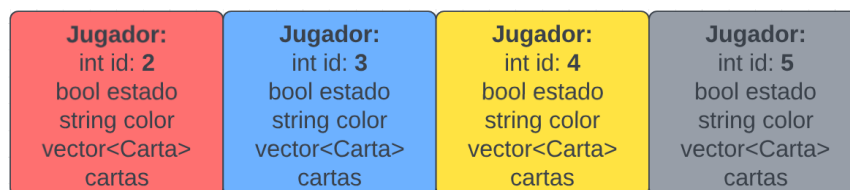


Figura 6. pop() de la cola de Jugadores. Fuente: Elaboración propia.

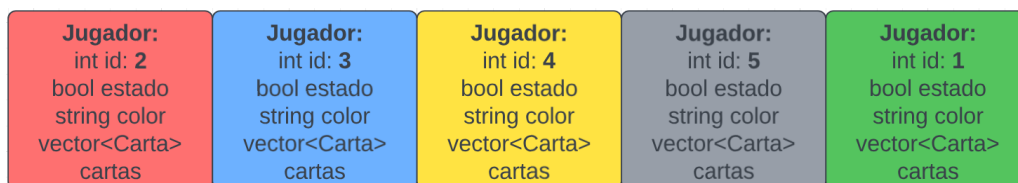


Figura 7. push() de la cola de Jugadores. Fuente: Elaboración propia.

Cada que un jugador sea eliminado de la partida, este se elimina de la cola de jugadores y el único jugador que quede en la cola será el ganador de la partida.

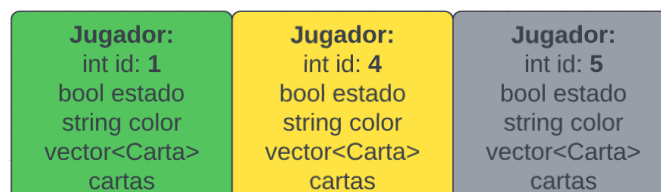


Figura 8. Jugadores 2 y 3 eliminados. Fuente: Elaboración propia.

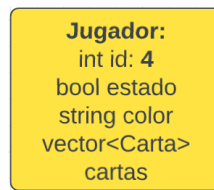


Figura 9. Jugador 4 ganador. Fuente: Elaboración propia.

Tras entender cómo funciona el sistema de turnos mediante colas en la partida, para entender mejor el funcionamiento de cada unas de las fases de turno, se decidió realizar el siguiente diagrama de flujo:

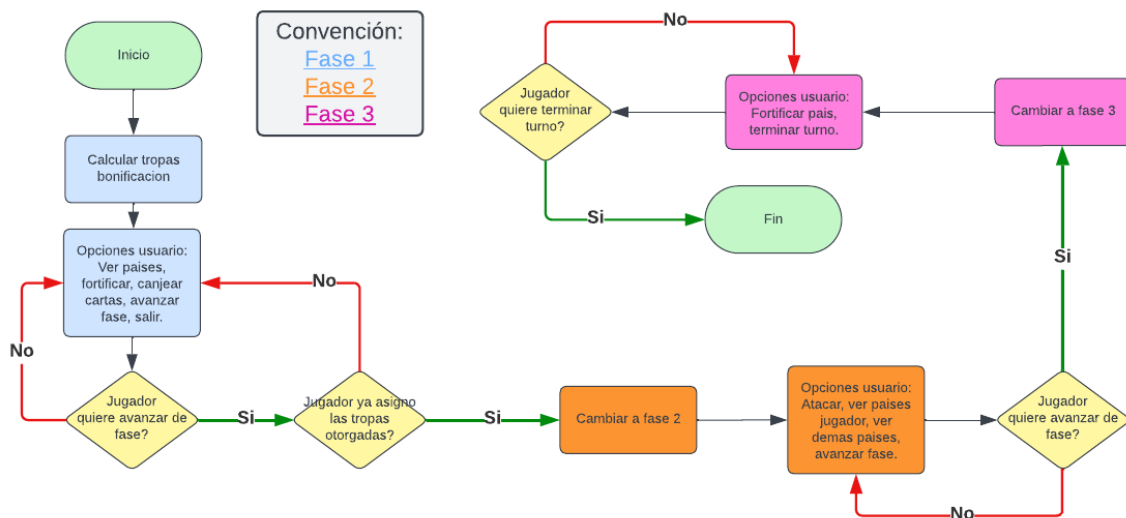


Figura 10. Flujo sistema de turnos. Fuente: Elaboración propia.

- **Fase 1 (Obtener nueva unidades):** Durante esta fase se le informará al usuario cuantas tropas adicionales puede reclamar, para luego preguntar en cuál(es) de los territorios de su propiedad las quiere asignar y en qué cantidad. Ahora, el sistema tendrá la tarea de calcular la bonificación de las tropas, que este variará dependiendo de la cantidad de territorios que tenga el usuario. Posteriormente se le mostrará un menú al usuario donde podrá ver todos los países, ver sus países, fortificar el país con las tropas de bonificación, canjear cartas, dirigirse a la siguiente fase o salir, es importante resaltar que el usuario no podrá avanzar a la siguiente fase si no ha distribuido las tropas de bonificación anteriormente.
- **Fase 2 (Atacar):** Durante el transcurso de esta etapa, el usuario dispondrá de la oportunidad de llevar a cabo la configuración de su estrategia ofensiva. En este proceso, podrá seleccionar tanto el punto de origen del ataque como el destino al cual desea dirigirse, teniendo en cuenta que el usuario solo podrá seleccionar este territorio si tiene más de una tropa en este, ya que no está permitido atacar desde un territorio en el que solo exista una tropa. Además, el sistema proporciona un menú de opciones para cada jugador en esta fase, en el cual podrá ejecutar opciones como llevar a cabo

ataques, ver sus propios territorios, mostrar los territorios de los otros jugadores y progresar a la siguiente fase del juego. Cabe recalcar que es posible no atacar y simplemente ir a la siguiente fase.

- **Fase 3 (Fortificar):** En la fase final del turno de comando, el usuario tendrá la oportunidad de fortalecer uno de sus territorios, mediante la selección de la cantidad de tropas que desee movilizar desde un territorio propio hacia el territorio elegido previamente, sin embargo, sólo podrá movilizar tropas si el territorio en cuestión tiene más de una tropa, ya que, no es posible dejar un territorio sin tropas.

Operaciones Auxiliares (Comandos)

- inicializar:
 - Comando: inicializar <nombre_archivo>
 - Descripción: Se inicializa una nueva partida si no se especifica el nombre del archivo de la partida guardada. De especificarse uno se reanuda la partida.
 - Condiciones: El usuario no ingrese un espacio o un vacío.
- turno:
 - Comando: turno <id_jugador>
 - Descripción: Informa al jugador la cantidad de unidades que puede reclamar, para luego asignarlas y en qué cantidad. Después, se encarga de la configuración del ataque desde que territorio y hacia cual, luego informa los valores obtenidos con los dados y la cantidad de unidades que se ganan o se pierden, este proceso se repite hasta que alguno de los territorios se quede sin unidades o hasta que el atacante decida detenerse. Por último, pregunta al jugador los territorios vecinos que desea seleccionar para la fortificación y también la cantidad de unidades que se trasladan de uno al otro.
 - Condiciones: El usuario debe indicar el id del jugador del cual quiere usar el turno.
- guardar:
 - Comando: guardar nombre_archivo
 - Descripción: El estado actual del juego es guardado en un archivo de texto plano, se guarda la cantidad de jugadores, nombre de cada jugador, color de cada jugador, países que ocupa, etc.
 - Condiciones: Debe existir una partida previamente creada.
- guardar comprimido:
 - Comando: guardar nombre_archivo
 - Descripción: el estado del juego actual es guardado en un archivo binario. Se guarda la cantidad de jugadores, nombre de cada jugador, color de cada jugador, países que ocupa, etc.
 - Condiciones: Debe existir una partida previamente creada.
- costo_conquista:
 - Comando: costo_conquista <territorio>

- Descripción: Se calcula el costo y secuencia de territorios a ser conquistados para lograr controlarlo. El territorio donde debe atacar debe ser aquel que el jugador tenga controlado más cerca al dado por el usuario.
- Condiciones: Debe existir una partida previamente creada y se debe ingresar un territorio válido.
- conquista_mas_barata:
 - Comando: conquista_mas_barata
 - Descripción: Calcula la conquista más barata para el jugador actual de todos los territorios posibles. Es decir, aquel territorio que pueda implicar un menor número de unidades perdidas.
 - Condiciones: Debe existir una partida previamente creada.
- clear:
 - Comando: clear
 - Descripción: Limpia la consola.
 - Condiciones: Ninguna.
- salir:
 - Comando: salir
 - Descripción: Termina la ejecución del programa. Toda partida que no se haya guardado se perderá.
 - Condiciones: Ninguna.

Diseño de TAD'S

TAD JUGADOR

Datos mínimos:

- id, entero, representa el identificador único de cada jugador (puede ser o un número o un nombre).
- estado, booleano, representa si el jugador ha sido eliminado o no.
- color, cadena de caracteres, representa el color del jugador.
- cartas, lista de Cartas del jugador, representa las cartas que posee el jugador.

Operaciones:

- ObtenerId(): Retorna el id del jugador
- ObtenerEstado(): Retorna el estado del jugador
- ObtenerColor(): Retorna el color elegido por el jugador
- ObtenerCartas(): Retorna la baraja de cartas del jugador
- FijarId(nid): Fija el id del jugador a nid.
- FijarEstado(nbool): Fija el estado de juego del jugador a nbool (eliminado, activo).
- FijarColor(ncolor): Fija el color del jugador a ncolor.
- FijarCartas(ncartas): Fija la baraja de cartas que posee el jugador a ncartas.

TAD CARTA

Datos mínimos:

- tipo, cadena de caracteres, representa el tipo de carta (normal, comodín o misión secreta).
- pais, cadena de caracteres, representa el país de la tarjeta.
- tropa, cadena de caracteres, representa la tropa de la tarjeta.

Operaciones:

- ObtenerTipo(): Retorna el tipo de carta
- ObtenerPais(): Retorna el nombre del país de la tarjeta
- ObtenerTropa(): Representa la tropa de la tarjeta
- FijarTipo(ncarta): Cambia el tipo de carta a ncarta
- FijarPais(ntarjeta): Cambia el país de la tarjeta a ntarjeta
- FijarTropa(ntropa): Cambia la tropa a ntropa

TAD PAÍS

Datos mínimos:

- nombre, cadena de caracteres, representa el nombre del país.
- idOcupanteActual, cadena de caracteres, representa el identificador del dueño del país.
- tropasPresentes, entero, representa las tropas presentes en el país.
- paisesVecinos, vector, representa los países vecinos de cada país.

Operaciones:

- ObtenerNombre(): Obtiene el nombre del país.
- ObtenerIdOcupanteActual(): Obtiene el identificador del dueño del país.
- ObtieneTropasPresentes(): Obtiene las tropas presentes en el país.
- ObtienepaisesVecino(): Obtiene los países vecinos de cada país.
- FijarNombre(nnombre): Obtiene el nombre del país a nnombre.
- FijarIdOcupanteActual(nocupanteactual): Obtiene el identificador del dueño del país a nocupanteactual.
- FijarTropasPresentes(ntropaspresentes): Obtiene las tropas presentes en el país a ntropaspresentes.
- Fijar paisesVecino(npaisesvecinos): Obtiene los países vecinos de cada país a npaisesvecinos.

TAD CONTINENTE

Datos mínimos:

- nombre, cadena de caracteres, representa el nombre del continente.

- tropasAdicionales, entero, representa la bonificación de tropas adicionales del continente cuando un jugador domina todos los países de este.
- paises, lista de Pais, representa los países que se encuentran en ese continente.w
- bonificación, entero, representa cuando el jugador completa alguno de los continentes recibiría tropas extra

Operaciones:

- ObtenerNombre(): Obtiene el nombre del continente.
- ObtenerTropasAdicionales(): Obtiene el número de tropas adicionales.
- ObtenerPaises(): Obtiene la lista de países que componen al continente.
- ObtenerBonificacion(): Obtiene la bonificación por completar continente
- FijarNombre(nnombre): Fija el nombre del continente a nnombre.
- FijarTropasadicionales(nTropasAdicionales): Fija el número de tropas adicionales.
- FijarPaises(npaises): Fija la lista de países que componen al continente a npaises
- FijarBonificacion(nbonificacion): Fija la bonificación por completar continente a nbonificacon.

TAD PARTIDA

Datos mínimos:

- nombre, cadena de caracteres, representa el nombre del juego actual.
- tipo, cadena de caracteres, representa la variante del juego actual.
- setsTradeados, entero, representa el número de sets tradeados en la partida.
- continentes, lista de Continente, representa los continentes que se encuentran en la partida.
- jugadores, cola de Jugador, representa los jugadores que se encuentran en la partida en su respectivo orden.
- cartas, lista de Carta, representa las cartas que se encuentran en la partida.

Operaciones:

- ObtenerNombre(): Obtiene el nombre de la partida
- ObtenerTipoPartida(): Obtiene el modo de juego de la partida
- ObtenerSetsTradeados(): Obtiene la cantidad de tradeos de la partida
- ObtenerContinentes(): Obtiene los continentes de la partida
- ObtenerJugadores(): Obtiene los jugadores de la partida
- ObtenerCartas(): Obtiene las cartas de la partida
- FijarNombre(nnombre): Fija el nombre de la partida a nnombre
- FijarTipoPartida(ntipo): Fija el modo de juego de la partida a ntipo
- FijarSetsTradeados(ntradeos): Fija la cantidad de tradeos de la partida a ntradeos
- FijarContinentes(ncontinentes): Fija los continentes de la partida a ncontinentes
- FijarJugadores(njugadores): Fija los jugadores de la partida njugadores
- FijarCartas(ncartas): Fija las cartas de la partida a ncartas

Diagrama TAD'S

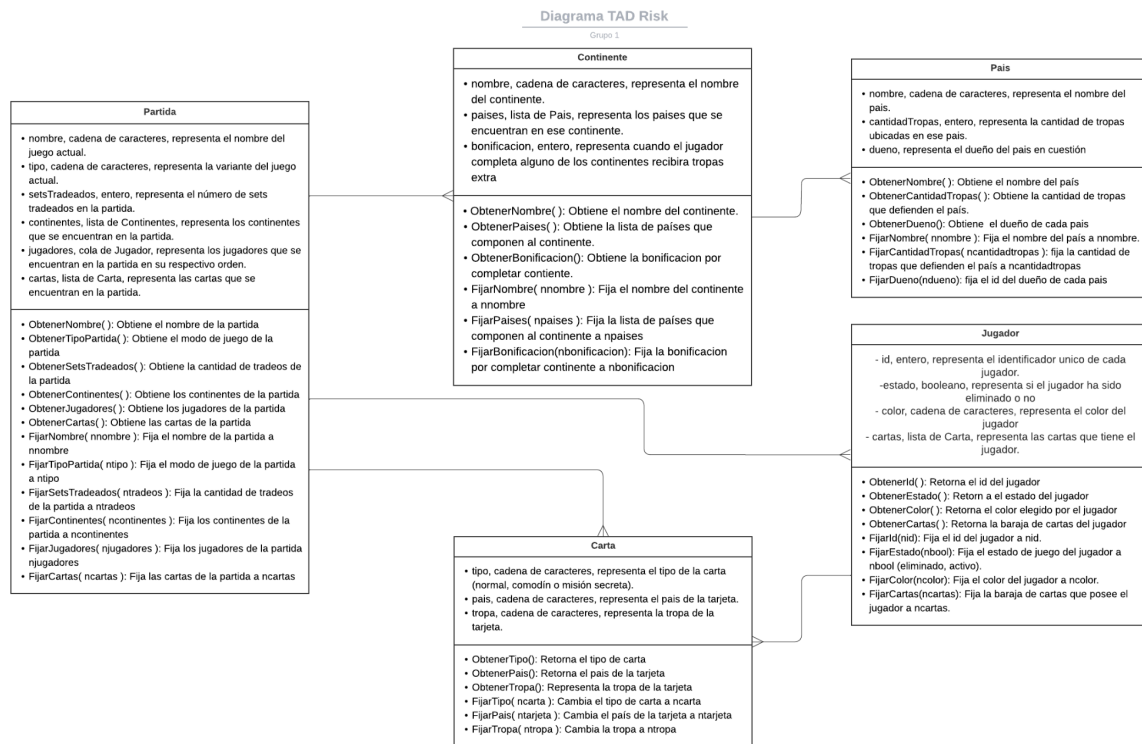


Figura 11. Diagrama de TADS. Fuente: Elaboración propia.

Planes de Prueba

Inicializar

Plan de pruebas: Nombre de partida				
Descripción del caso	Valores de entrada	Resultado esperado	Resultado obtenido	
Se ingresa solo letras	Cualquier letra del abecedario	Se crea la partida correctamente	Continúa con el flujo normal	
Se ingresa solo números	Cualquier número	Se crea la partida correctamente	Continúa con el flujo normal	
Se ingresa solo caracteres especiales	Cualquier carácter especial	Se crea la partida correctamente	Continúa con el flujo normal	
Se ingresa solo un espacio	Solo un espacio	No se crea la partida	“El nombre de la partida no puede contener solo espacios en blanco.”	
Se ingresan dos	Mi primera partida	No se crea la partida	“El nombre de la	

cosas separadas por espacio(s)	Hola mundo 2 2 ? ?		partida no puede estar separada por espacios.”
--------------------------------	--------------------------	--	--

Plan de pruebas: Tipo de partida			
Descripción del caso	Valores de entrada	Resultado esperado	Resultado obtenido
Se selecciona el tipo de partida “Normal”	Normal	Se selecciona a partida para inicializar como partida “Normal”	Continúa con el flujo normal para inicializar como partida “Normal”
Se selecciona el tipo de partida “Misión Secreta”	Misión Secreta	Se selecciona a partida para inicializar como partida “Misión Secreta”	Continúa con el flujo normal para inicializar como partida “Misión Secreta”
El usuario ingresa una opción inexistente o incorrecto	Cualquier entrada diferente a “Normal”, “Misión secreta”, “Salir”	Opción inválida	“Opción inválida, ingrese un modo de juego”
El usuario ingresa una cadena vacía		Espera por input del usuario	“Elija el modo de juego”
El usuario selecciona la opción “Salir”	“Salir”	Sale al menú principal	Va al flujo normal del menú principal

Plan de pruebas: Cantidad de jugadores			
Descripción del caso	Valores de entrada	Resultado esperado	Resultado obtenido
Inicializa con cantidad de jugadores negativos	Números negativos	Cantidad de usuarios no válida	“Error. Debe ingresar entre 3 y 6 jugadores”
Inicializa con menos de dos jugadores	1	Cantidad de usuarios no válida	“Error. Debe ingresar entre 3 y 6 jugadores”
Inicializa entre 3 y 6 jugadores	Números entre 3 y 6	Cantidad de usuarios válida	Continúa con el flujo normal
Inicializa con más	Números mayores a	Cantidad de usuarios	“Error. Debe

de 6 jugadores	6	no válida	ingresar entre 3 y 6 jugadores”
Inicializa con números decimales	Ingresa un número no entero	Cantidad de usuarios no válida	“Error. Debe ingresar entre 3 y 6 jugadores”
Inicializa con un carácter no numérico	Ingresa algo diferente a un número	Cantidad de usuarios no válida	“Error. Debe ingresar entre 3 y 6 jugadores”
Inicializa con una entrada vacía	No ingresa nada	Vuelve a preguntar	“Cuántos jugadores tendrá la partida ‘nombre_partida’ (min 2, max 6):”

Plan de pruebas: Elección de color del jugador			
Descripción del caso	Valores de entrada	Resultado esperado	Resultado obtenido
El jugador selecciona un color de la lista	Cualquier color de la lista de colores disponibles	Conecta al usuario con ese color	Continúa con el flujo normal
El jugador selecciona un color que ya ha sido escogido	Color elegido por un jugador anteriormente	Vuelve a pedir que ingrese el color	“Color: 'ingreso' no encontrado o ya fue seleccionado. Por favor ingrese un color válido
El jugador ingresa una opción inválida o inexistente	Color inexistente	Vuelve a pedir que se ingrese color	“Color: 'ingreso' no encontrado o ya fue seleccionado. Por favor ingrese un color válido
El jugador ingresa una entrada vacía		Vuelve a preguntar	Vuelve a pedir que se ingrese el color

Segunda Entrega

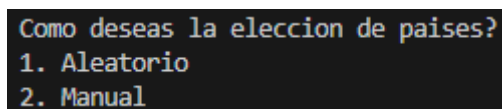
Acta de evaluación

Comentarios profesor:

1. Permitir que la asignación de tropas inicial se pueda hacer de forma manual

Como fue corregido:

Al momento de terminar de elegir los colores para los jugadores en la partida en el comando “inicializar” como se puede ver en la figura 2, se decidió agregar un menú en el cual se le pregunta al usuario si desea elegir los países de manera manual o manera automática.



```
Como deseas la eleccion de paises?  
1. Aleatorio  
2. Manual
```

Figura 12. Elección de países en el programa. Fuente: Elaboración propia.

Como se puede ver en la imagen adjunta, en caso de que el usuario elija la opción automática, el proceso se desarrollará de acuerdo con la configuración previamente establecida. Sin embargo, en dado caso que se desee llevar a cabo la elección de los países de manera manual, se procederá a solicitar el nombre del país a cada jugador de manera secuencial. Este procedimiento se repetirá hasta que se hayan distribuido la totalidad de los 42 países que conforman el juego.

Es por lo cual, el diagrama de flujo del comando inicializar fue modificado a lo siguiente:

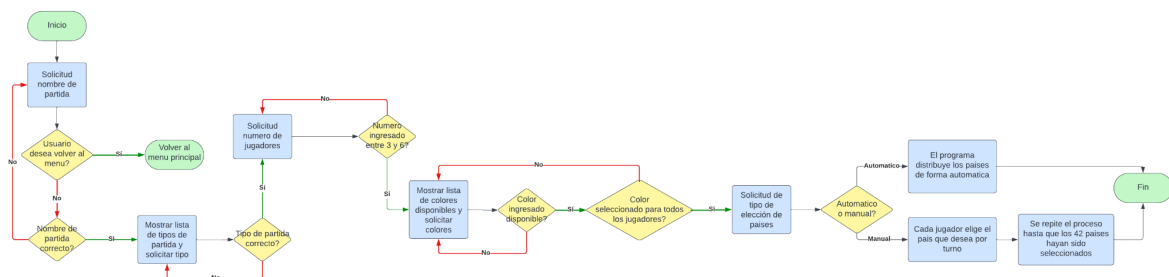


Figura 13. Diagrama de secuencia inicializar partida. Fuente: Elaboración propia.

2. Modificar la figura 1 del documento

Se realizó la modificación correspondiente con las recomendaciones dadas por el profesor durante la sustentación.

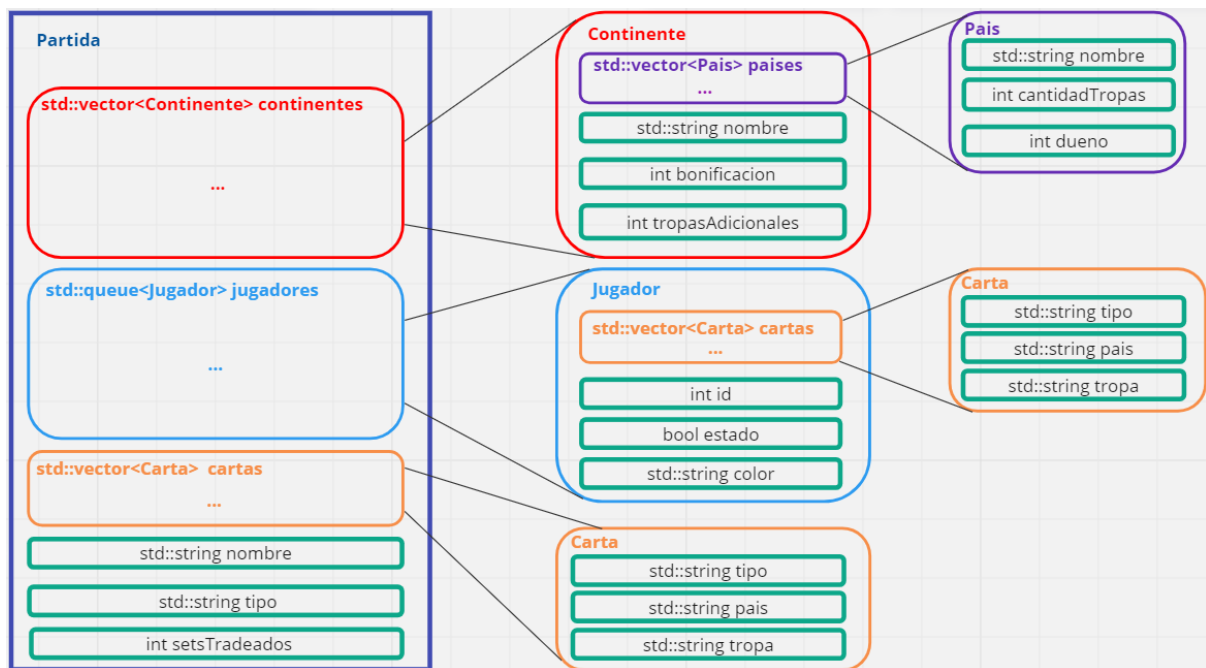


Figura 14. Estructura de datos. Fuente: Elaboración propia.

La explicación de nuestro diagrama de estructura de datos se puede encontrar al inicio del documento donde estaba ubicada la imagen anterior.

Documento de diseño

Descripción de entradas, salidas y condiciones:

Procedimiento Principal

Guardar Texto Plano

Para guardar una partida, es necesario que primero haya una partida activa en el programa, de lo contrario no se procederá a guardar ningún archivo.

Para generar el archivo, lo primero que se realiza es almacenar toda la información de la partida contenida en una cadena de caracteres en formato json para que pueda ser leído en un futuro por el programa cuando el usuario lo desee, para realizar este proceso, se hace un llamado a la función “partida_a_JSON()” que retorna un string guardada en **partidaJSON**.

Para finalizar, se genera un archivo con el nombre que el usuario haya querido darle a este, y con extensión .json, el contenido dentro del archivo es la cadena de caracteres en formato json que se ha creado previamente. Una vez se crea el archivo, se le indica al usuario que este ha sido creado con éxito, de no ser así, se le informa del problema.

La siguiente figura ilustra el proceso de guardar en texto plano en RISK:

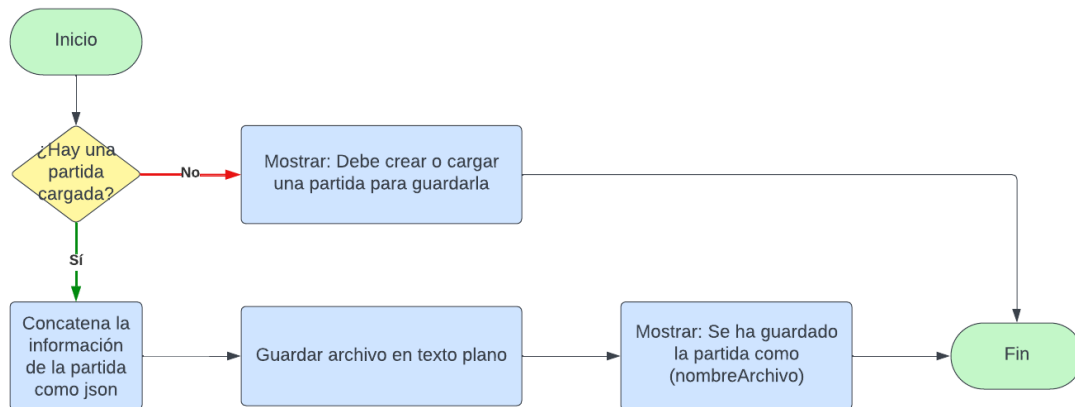


Figura 15. Diagrama de flujo “guardar”. Fuente: Elaboración propia.

Guardar Comprimido

Igual que guardar en texto plano, para guardar una partida, es necesario que primero haya una partida activa en el programa, de lo contrario no se procederá a guardar ningún archivo.

Teniendo en cuenta la lógica de guardar una partida a un string JSON, lo primero que se debe hacer es llamar la función “partida_a_JSON()” la cual devuelve un string con toda la partida en un solo string que se guarda en la variable **partidaJSON** que se utilizará más adelante.

Cabe recalcar que nosotros estamos utilizando el **Árbol de Huffman** para codificar la partida y que sea posible guardarla en un archivo **.bin**. Para poder generar este árbol con el cual se hará la codificación, se deben calcular las frecuencias en las cuales aparecen cada símbolo (carácter) de toda la partida.

Esto se hace con un simple for el cual va agregando a un **map<char, int>** la frecuencia en la cual aparece cada carácter de la siguiente forma:

```

std::map<char, int> frequencies;

for (char c : partidaJSON) {
    frequencies[c]++;
}
  
```

Figura 16. Generación tabla de frecuencias. Fuente: Elaboración propia.

Tras la generación de esta tabla de frecuencias en el map, se procede a crear el Árbol de Huffman simplemente usando el constructor creado que recibe un map de frecuencias. Después de esto, se utiliza la función codificar del Árbol de Huffman que devuelve el string “cifrado” y posteriormente se utiliza la función para guardar en archivo binario.

Lo que hace la función “codificar” del Árbol de Huffman es lo siguiente:

Teniendo en cuenta que el árbol ya fue creado con las frecuencias calculadas previamente, la lógica que se sigue es que por cada carácter en el string de la partida completa se busca el código asociado con una función recursiva que hace un recorrido por profundidad que va buscando cada carácter en el árbol y va construyendo el código.

Cada vez que baje a la izquierda, le agrega 0 al código de longitud variable y cada que baje a la derecha le agrega 1 al código de longitud variable hasta que encuentre el carácter deseado.

Esta cadena de 1 's y 0' s se va construyendo poco a poco hasta que llega al final del JSON.

Tras tener este string codificado, se llama a la función “guardarEnArchivoBinario” que recibe el string codificado, el nombre del archivo en donde se va a guardar y las frecuencias asociadas. En pocas palabras, lo que hace esta función es simplemente escribir en un archivo binario teniendo en cuenta la estructura de un archivo binario que fue proporcionada en el enunciado del proyecto.

La siguiente figura ilustra el proceso de guardar comprimido en Risk:

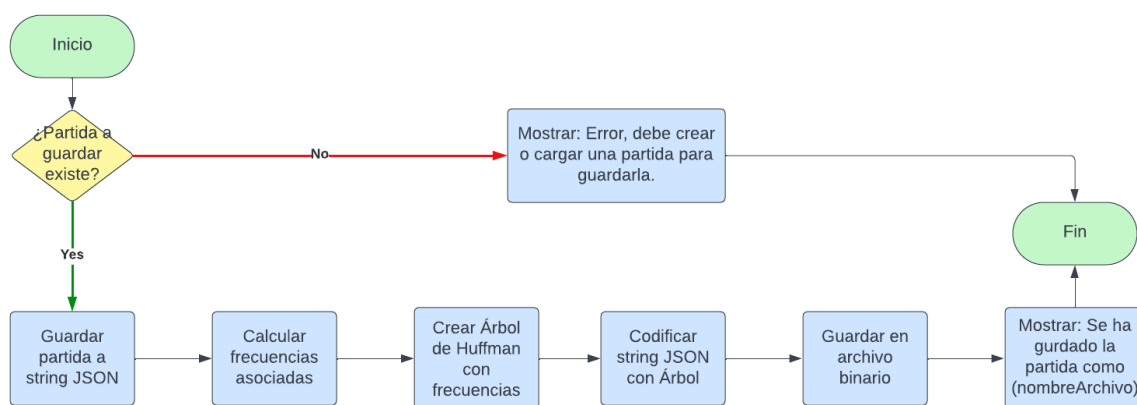


Figura 17. Diagrama de flujo “guardar_comprimido”. Fuente: Elaboración propia.

Inicializar partida de archivo

Cuando se quiere cargar una partida, el usuario en la opción de “inicializar” pone el nombre del archivo para poder utilizarla. Lo primero que se hace es comparar la extensión del archivo y revisar si esta tiene la partida como texto plano, o como un archivo comprimido, para hacer la verificación se toman los últimos caracteres del nombre del archivo y busca si hay coincidencia con alguno de los dos formatos, de no ser así, le indica que el archivo es incompatible con el programa.

Si el usuario indica un archivo con extensión .json, a partir de los valores de cada una de las llaves del string se construye la partida.

Si el archivo indicado tiene la extensión .bin, el programa llamará a la función “leerArchivoBinario()” del árbol de Huffman y se halla primeramente la tabla de frecuencia

de caracteres de la partida para posteriormente decodificarlo y retornarlo como cadena de caracteres. Una vez que se hace la conversión de binario a string con formato json, a partir de los valores de cada una de las llaves se construye la partida.

La siguiente figura ilustra el proceso de cargar una partida a Risk:

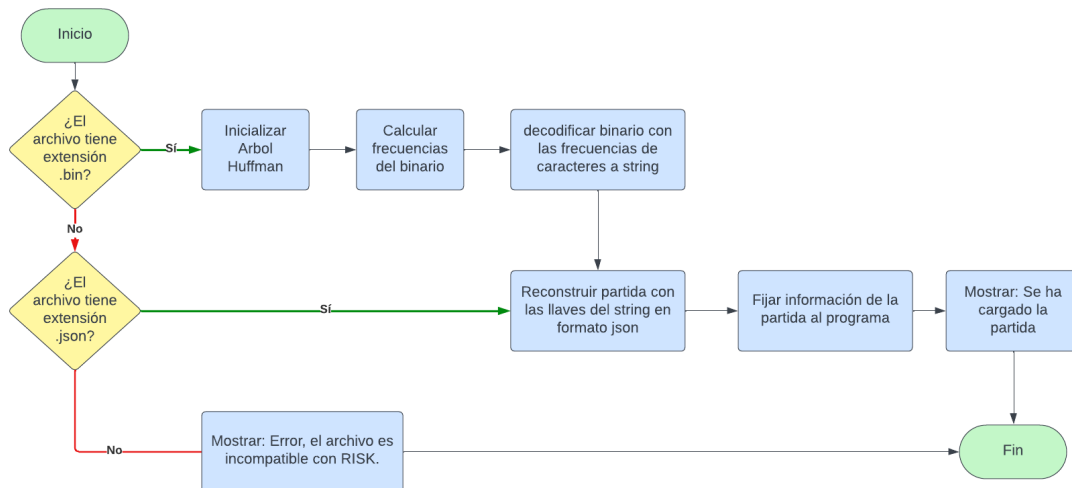


Figura 18. Diagrama de flujo “inicializar <partida>”. Fuente: Elaboración propia.

Diseño de TAD's

Lo siguiente es lo modificado para la segunda entrega.

TAD ArbolH

Datos mínimos:

- raíz, apuntador a NodoH, representa el nodo raíz del árbol.

Operaciones:

- construirArbolH(frecuencias): construye el árbol de huffman a partir de las frecuencias que llegan como parámetro.
- construirCodigoHuffman(nodo, codigo, codigosHuffman): construye el código huffman a partir del nodo, codigo y lo guarda en codigosHuffman.
- obtenerCodigoHuffman(): obtiene el código Huffman a partir del árbol.
- codificar(texto): codifica el texto que llega como parámetro teniendo en cuenta el árbol de Huffman asociado.
- decodificar(texto): decodifica el texto que llega como parámetro teniendo en cuenta el árbol de Huffman asociado.
- guardarEnArchivoBinario(texto, nombreArchivo, frecuencias): guarda el texto que llega como parámetro en un archivo binario que tiene el nombre nombreArchivo que llega como parámetro, teniendo en cuenta la frecuencia que llega como parámetro.

- leerArchivoBinario(nombreArchivo): lee el archivo binario nombreArchivo que llega como parámetro.

TAD NodoH

Datos mínimos:

- data, tipo char, representa el dato almacenado en este nodo.
- frecuencia, tipo entero, representa la frecuencia almacenada en ese nodo.
- izq, apuntador a NodoH, representa el hijo izquierdo del nodo.
- der, apuntador a NodoH, representa el hijo derecho del nodo.

Operaciones:

- NodoH(data, frecuencia): constructor del nodoH que recibe una data y una frecuencia asociada.

TAD Menú

Datos mínimos:

-

Operaciones:

- comando_inicializar_nueva_partida(): inicializa una nueva partida de Risk.
- comando_inicializar_existente(comando): carga una partida existente de Risk, dependiendo de comando para saber si carga desde un archivo binario o un archivo de texto plano.
- comando_turno(id): realiza el turno del jugador con id que llega como parámetro.
- comando_guardar(nombreArchivo): guarda la partida en un archivo de texto plano con el nombre nombreArchivo que llega como parámetro.
- comando_guardar_comprimido(nombreArchivo): guarda la partida en un archivo binario con el nombre nombreArchivo que llega como parámetro.
- comando_costo_conquista(territorio): calcula el costo de conquista del territorio que llega como parámetro.
- comando_conquista_mas_barata(): calcula la conquista más barata del jugador actual.
- comando_ayuda(): muestra por pantalla cuales son los comandos disponibles en el programa.
- comando_ayuda_comandos(comando): dependiendo del comando que llega por parámetro, muestra una ayuda de ese comando en específico.
- limpiar_consola(): limpia la pantalla de la consola.
- interaccion_usuario(): pregunta al usuario el comando a ejecutar y redirige a este.
- imagen_risk(): imprime la imagen banner del juego Risk.
- contieneSoloEspacios(str): revisa si la cadena de texto str que llega por parámetro contiene solo espacios o no.

- partida_a_JSON(): guarda toda la información de la partida a un solo string JSON.

Diagrama TAD's Árbol Huffman de Codificación

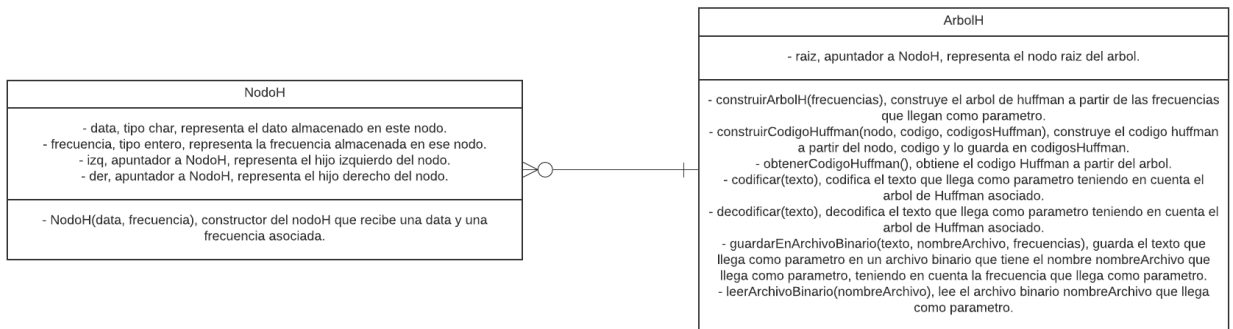


Figura 19. Diagrama TAD 's Árbol de Huffman. Fuente: Elaboración propia.

Diagrama TAD's actualizado

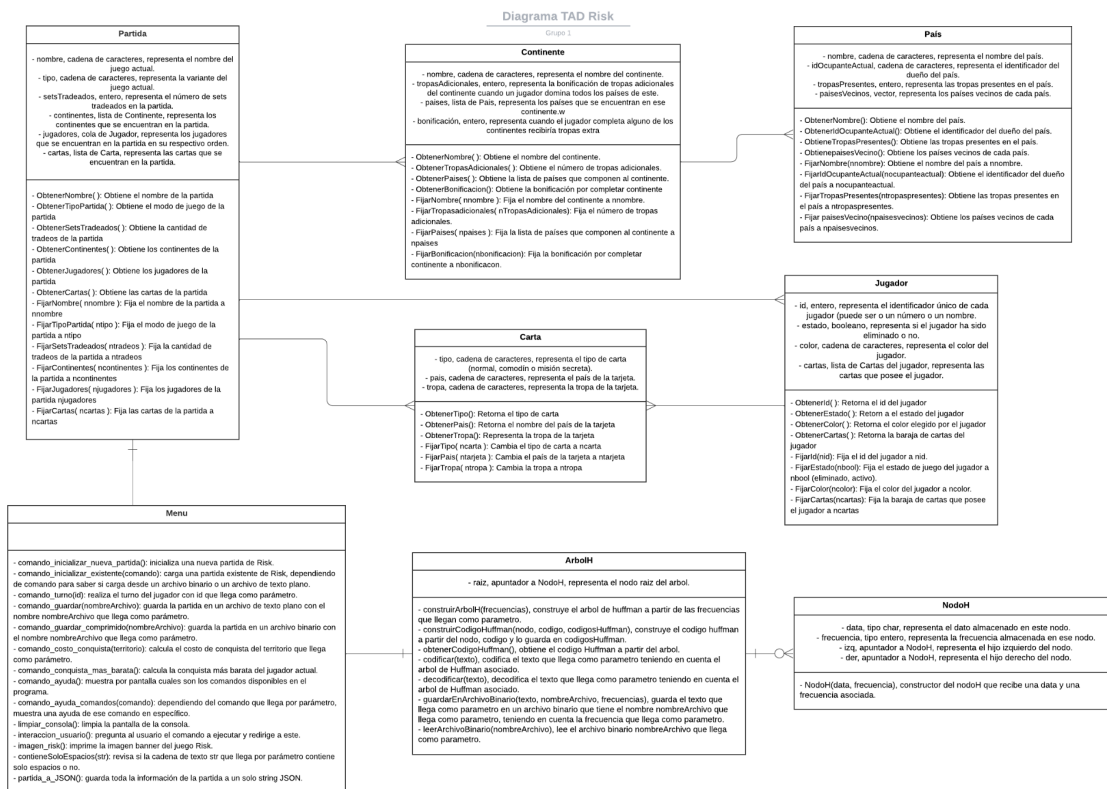


Figura 20. Diagrama TAD 's actualizado. Fuente: Elaboración propia.

Plan de pruebas

guardar_comprimido

Plan de pruebas: Nombre archivo a guardar			
Descripción del caso	Valores de entrada	Resultado esperado	Resultado obtenido
Se intenta guardar sin nombre	“guardar_comprimido”	Fallo en guardar	Error: Debe ingresar el nombre del archivo para guardar la partida comprimida.
Se intenta guardar con un espacio	“guardar_comprimido “	Fallo en guardar	Error: Debe ingresar el nombre del archivo para guardar la partida comprimida.
Se intenta guardar con letras	“guardar_comprimido nombre“	Guardado correctamente	Se ha guardado la partida como nombre.bin
Se intenta guardar con números	“guardar_comprimido 777“	Guardado correctamente	Se ha guardado la partida como 777.bin
Se intenta guardar con números y letras	“guardar_comprimido partida777“	Guardado correctamente	Se ha guardado la partida como nombre777.bin
Se intenta guardar con caracteres especiales	“guardar_comprimido funciona?“#”	Guardado correctamente	Se ha guardado la partida como funciona?“#.bin
Se intenta guardar con dos palabras	“Se ha guardado la partida como dos.bin”	Guardado correctamente pero únicamente con la primera palabra	Se ha guardado la partida como dos.bin

Tercera Entrega:

Acta de Evaluación

Comentarios profesor:

1. Arreglar los pesos porque el archivo comprimido está pesando más que el de texto plano y debería ser al revés.

Como fue corregido:

El principal error que teníamos es que nosotros estábamos guardando cada uno de los caracteres codificados como una cadena de texto, es por lo cual, el archivo “comprimido” pesaba más que el de texto plano, porque todo lo guardábamos en cadena de texto.

Después de hablar con el profesor, él nos recomendó que deberíamos pasar ese número binario a decimal y ahí si escribirlo en el archivo comprimido para reducir su peso.

Lo que se hizo fue lo siguiente:

Simplemente cambia la forma en la que se guardan los datos en el bin, primero se debe convertir cada 8 bits (1 byte) el string codificado a decimal y ahí si se escribe en el archivo.

Es por lo cual se cambia la lógica para escribir en binario entonces en el archivo **arbol.hxx** se modifica la siguiente función:

void ArbolH::guardarEnArchivoBinario(std::string mensaje, std::string nombreArchivo, std::map<char,int> frecuencias)

```
for (size_t i = 0; i < binarioCompleto.size(); i += 8) {  
    unsigned char byteDecimal = 0;  
    for (int j = 0; j < 8; j++) {  
        byteDecimal = (byteDecimal << 1) | (binarioCompleto[i + j] - '0');  
    }  
    archivo.write(reinterpret_cast<char*>(&byteDecimal), sizeof(byteDecimal));  
}
```

Figura 21. Cambio funcion guardarEnArchivoBinario. Fuente: Elaboración propia.

Como se puede ver, primero se encuentra si el binario está completo, si no lo está, lo completa y finalmente lo convierte a decimal para poderlo escribir en el archivo.

Ahora, teniendo en cuenta que se cambia la estructura del archivo al guardarlo, es necesario cambiar cómo se lee de este, es por lo cual realizamos lo siguiente:

Se cambió la forma en la que se lee del archivo binario porque se debe tener en cuenta que ahora están guardados en decimal. Cuando se lee, se convierte en binario de nuevo para poder decodificar con el árbol de Huffman:

Es por lo cual se cambia la lógica para leer del archivo binario, más específicamente en el archivo **arbol.hxx** se modifica la siguiente función:

std::string ArbolH::leerArchivoBinario(std::string nombreArchivo)

```
while (archivo.read(&buffer, sizeof(buffer))) {  
    for (int i = 7; i >= 0; --i) {  
        secuenciaBinaria += ((buffer >> i) & 1) ? "1" : "0";  
    }  
}
```

Figura 22. Cambio funcion leerArchivoBinario. Fuente: Elaboración propia.

Aquí, mientras se siguen leyendo decimales binarios, este se convierte de decimal a binario para después poder decodificar con el árbol de Huffman.

Tras haber realizado estos cambios, se vuelven a generar los archivos con la misma partida, uno en texto plano y otro comprimido lo cual nos da el siguiente resultado:

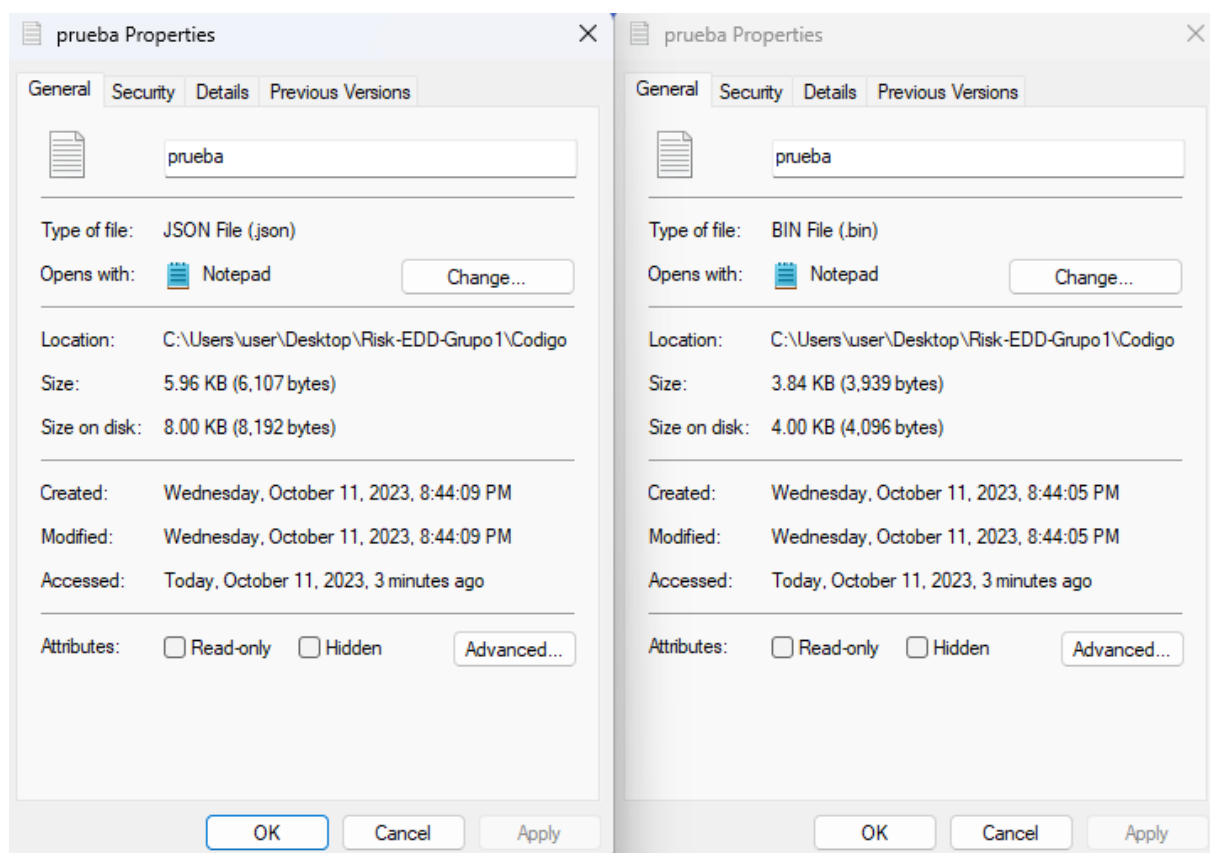


Figura 23. Comparación pesos archivos. Fuente: Elaboración propia.

Se genera la siguiente tabla de pesos comparativos:

Tipo de Archivo	Texto Plano (.json)	Comprimido (.bin)
Tamaño	5.96 KB	3.84 KB
Tamaño en Disco	8.00 KB	4.00 KB

Como se puede evidenciar, el archivo comprimido es la mitad del peso que el de texto plano en el tamaño en disco entonces queda corregido los comentarios para poder dar inicio a la tercera entrega.

Descripción de entradas, salidas y condiciones:

Procedimiento principal

Costo conquista

Para determinar el costo de conquista de un país que se pretende atacar, es importante seguir un procedimiento específico dentro del sistema. Al acceder a la opción de turno, el jugador debe completar obligatoriamente la fase 1 para avanzar a la fase 2, donde se ubica la funcionalidad destinada a calcular el costo de conquista de un país en particular.

Una vez dentro de la sección de costo de conquista, se presentará al jugador una tabla que enumera todos los países disponibles para un ataque potencial. En este punto, se solicitará al usuario ingresar el ID correspondiente al país sobre el cual desea conocer el costo de ataque. Una vez que el usuario haya seleccionado un ID válido de país, se iniciará el algoritmo de Dijkstra para determinar el camino más corto desde cada país del jugador. Finalmente, el programa mostrará el costo de la conquista hacia el país que se desea atacar.

La siguiente figura ilustra el proceso del comando costo conquista:

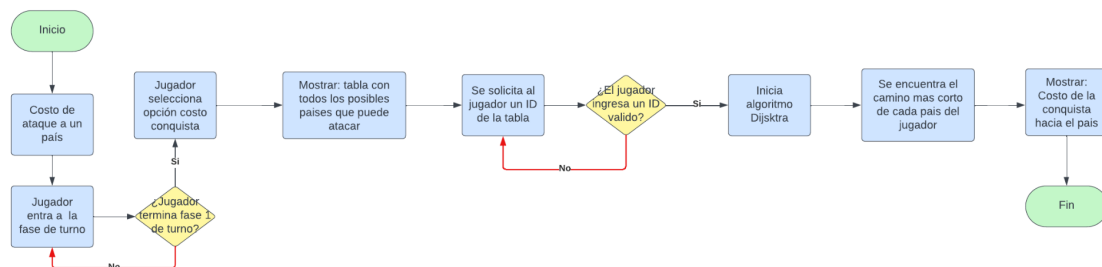


Figura 24. Flujo comando costo_conquista. Fuente: Elaboración propia.

Conquista más barata

Para determinar el costo de la conquista más económica disponible para el jugador, se requiere seguir un procedimiento específico. En primer lugar, es fundamental que el usuario se sitúe en la opción de turno correspondiente, específicamente en la segunda fase.

Una vez dentro de la segunda fase del turno, el usuario deberá seleccionar la opción numerada como cinco, lo que permitirá al programa mostrar la conquista que tenga el costo más bajo. El programa obtendrá el ID de todos los países pertenecientes al jugador y los IDs de los países de los demás jugadores para realizar una comparación de tropas entre el jugador y sus vecinos. Finalmente, el programa mostrará la conquista de menor costo disponible.

La siguiente figura ilustra el proceso del comando costo conquista:

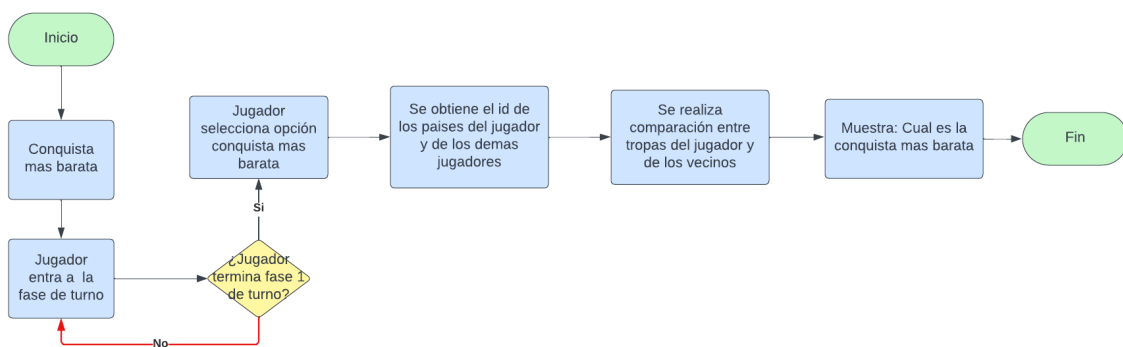


Figura 24. Flujo comando conquista_mas_barata. Fuente: Elaboración propia.

Diseño de TAD's

TAD Grafo

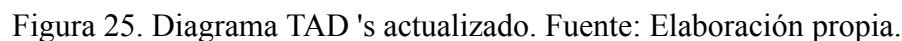
Datos mínimos:

- matriz_adyacencia, lista de listas de tipo entero, representa la matriz de adyacencia del juego.
- paises, lista de cadenas de texto, representa los paises del juego.

Operaciones:

- setMatriz_Adyacencia(m): Cambia la matriz_adyacencia del grafo a m que llega como parámetro.
- obtenerMatriz(): Retorna la matriz de adyacencia del grafo.
- mostrarMatrizAdyacencia(): Imprime la matriz de adyacencia del grafo.
- setPaises(p): Cambia los paises del grafo a p que llega como parámetro.

- ### Diagrama TAD's actualizado



Plan de pruebas

Costo conquista

Plan de pruebas: Costo conquista			
Descripción del caso	Valores de entrada	Resultado esperado	Resultado obtenido
Ingreso a opción	Opción 4 en Fase 2 de turno	Tabla de todos los países con sus tropas y dueño	Tabla con la información y solicitud del número

			del país del que se desea atacar
Ingreso de número de país inexistente	Id de pais inexistente	Mensaje informando que el país elegido no existe	Id del país enemigo no existe, ingrese de nuevo el numero del país
Ingreso de letras	“a” “b” “c”	Mensaje informando que solo se reciben números	Solo debe ingresar el numero del país
Ingreso de caracteres especiales	“?” “*” “)” “!”	Mensaje informando que solo se reciben números	Solo debe ingresar el numero del país
Ingreso de número de pais valido	Id del pais valido	Mensaje informando del costo de conquista	Para conquistar el territorio <territorio>, debe atacar desde <territorio_1>, pasando por los territorios <territorio_2>, <territorio_3>,..., <territorio_m> Debe conquistar <n> unidades de ejército.