Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2233 Programación Avanzada (2021-1)

Tarea 3

Entrega

- **Avance de tarea**
  - **Fecha y hora:** viernes 2 de julio de 2021, 20:00
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T3/`
- **Tarea**
  - **Fecha y hora:** martes 13 de julio de 2021, 20:00
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T3/`
- `README.md`
  - **Fecha y hora:** martes 13 de julio de 2021, 22:00
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T3/`

Objetivos

- Tomar decisiones de diseño y modelación en base a un documento de requisitos.
- Aplicar conocimientos de creación de redes para comunicación efectiva entre computadores.
- Diseñar e implementar una arquitectura cliente-servidor.
- Diseñar e implementar estructuras de datos propias basadas en nodos, para modelar y solucionar un problema en concreto.
- Aplicar conocimientos de manejo de *bytes* para seguir un protocolo de comunicación preestablecido.

Índice

1. *DCCópteros*

Ya han pasado 474 días desde que inició el encierro: te has escuchado todas las canciones de Chayanne, has dormido más clases de las que deberías (y por cosas como esas es que has descubierto el poder de ver clases en 2x), y extrañas el campus y la Universidad presencial más que a nada.

Un día mientras jugabas DCCimpsons, recibes un *mail* del decano diciendo que el campus San Joaquín comenzará su retorno a clases presenciales 🤩. Ves pasar mil imágenes distintas por tu cabeza: las papitas de agro 🍟, los sillones de la biblioteca 😴, los helados del casino de las K 🍦, las sopaipillas a la salida de la U 😋. Son tantas las opciones que no se te ocurre por cuál partirás apenas llegue el día. Sin embargo, resulta que para que todo esto sea posible, el decano fue estricto con respecto a evitar cualquier tipo de aglomeración de estudiantes y, por ende, les estudiantes solo se podrán mover entre las salas asignadas (nada de pasear por todo el campus 😠, así evitar posibles contagios). Para facilitar esto, se le solicitó a los estudiantes de Programación Avanzada 😎, con sus amplios conocimientos de ***Networking*, Serialización y Estructuras nodales**, que realizaran el proyecto *DCCópteros* en conjunto con estudiantes del major de robótica para hacer una flota de drones a los cuales se les pedirán encargos. Como incentivo, dijo que el equipo con el mejor proyecto se titularía de forma automática.[1].

Tu misión será encargarse del *software* detrás de este proyecto. Para esto, harás en primera instancia una simulación de cómo se vería esto una vez ya todo esté funcional: La idea es que los estudiantes podrán solicitar el arriendo de caminos que podrán ser recorridos por sus drones, en donde el costo de cada camino será equivalente a la cantidad de baterías que necesite su dron para recorrerlo. Todo se explicará en más detalle en las secciones siguientes.

Figura 1: Logo de *DCCópteros*

2. Flujo del programa

DCCópteros es un juego de simulación multijugador por turnos, en el que se competirá por ser el jugador que logre acumular más puntos al final de la partida. Un jugador ganará puntos ya sea por adquirir un camino que conecta 2 lugares o bien por completar un objetivo secreto que se le asigna al inicio del juego.

DCCópteros debe contar con un **servidor** que pueda transmitir datos entre jugadores y manejar el flujo del juego. El servidor debe ser lo primero que se ejecuta, antes que los programas de los jugadores.

[1]Todo lo que respecta a notas o titulación es meramente a modo de broma, por favor no tomarse nada de esto de forma seria.

Cada jugador debe ejecutar una instancia de cliente, programa que iniciará una interfaz gráfica la cual será su único medio de interacción con el juego. Inicialmente, el jugador comenzará en la Ventana de Inicio, donde tendrá que escribir su nombre de usuario. En caso de que este sea válido, y si el servidor tiene espacio suficiente para aceptar a este nuevo jugador en la partida se procederá a la Sala de Espera donde se puede ver al resto de jugadores conectados, junto con una votación para elegir uno de los 2 mapas que hay disponibles para la simulación. En caso contrario, la interfaz deberá mantenerse en la Ventana de Inicio notificando al jugador que no fue posible entrar al juego dando la posibilidad de intentarlo nuevamente o de salir del programa.

Una vez dentro de la Sala de Espera, los jugadores deberán votar por su mapa preferido y esperar a que ingrese el número necesario de jugadores para iniciar la partida, valor definido por el parámetro `CANTIDAD_JUGADORES_PARTIDA` (que puede ser de 2 a 4 jugadores). Cuando este valor se cumpla y todos los usuarios hayan votado por el mapa, el jugador definido como *Host* de la partida podrá seleccionar la opción de iniciar la partida y redirigir a todos los jugadores a la Sala de Juegos, para comenzar a jugar.

Al iniciar la partida, se asigna de forma aleatoria el orden de turnos en que los jugadores participarán y un color que represente a cada jugador. Además, a cada uno se le entregará de forma aleatoria un objetivo que deberá completar.

Al iniciar el turno de un jugador, este tiene 2 posibilidades: sacar una carta que representa la cantidad de baterías que obtendrá o comprar caminos que conectarán 2 puntos del mapa.

Finalmente, una vez que todos los caminos han sido comprados, la partida se dará por acabada. A cada jugador se le mostrará el podio de los jugadores con sus puntajes respectivos, ordenados de forma descendente, y mediante un botón se le dará la opción de regresar a la Sala de Espera para que pueda unirse a una nueva partida.

3. Reglas de *DCCópteros*

DCCópteros posee un tablero que representa el mapa de San Joaquín, en el cual se ubicarán cada una de las facultades y caminos por donde se desplazarán los drones de cada uno de los jugadores.

3.1. Mapa

El mapa de *DCCópteros* presenta un conjunto de facultades conectadas a través de caminos **no dirigidos**. Cada una de las facultades representa un nodo de un grafo, mientras que los caminos son las aristas del grafo. El jugador debe comprar estos caminos para hacer uso de los drones y poder usar esa ruta bajo su dominio. Cada camino adquirido otorgará **puntos**.

A continuación se presentan las componentes principales del mapa:

3.1.1. Facultades

Las facultades son los nodos ubicados en el tablero. Cada facultad puede tener uno o más caminos hacia otras facultades. Cada camino estará representado como una arista no dirigida entre cada uno de los nodos. Esto quiere decir que si una facultad **A** tiene un camino con la facultad **B**, entonces ese mismo camino conecta la facultad **B** con la facultad **A**.

3.1.2. Caminos

Al comienzo del juego, ninguno de los caminos presentes en el tablero poseen dueño.Cada camino posee un **costo asociado** equivalente a las baterías necesarias para que un dron pueda recorrer ese camino. Los jugadores pueden comprar cada uno de los caminos, pagando el **costo** de este. Una vez que se

compra un camino, este se debe pintar según el color del jugador, y no puede ser comprado ni transferido posteriormente a otro jugador. Cada camino otorga una cierta cantidad de puntos **puntos** al jugador que lo haya comprado de acuerdo a la siguiente tabla:

Costo del camino	Puntos
1	1
2	2
3	4
4	7
5	10
6	15

Cuadro 1: Relación del costo del camino y los puntos que otorga.

La Figura 2 muestra el grafo de un mapa, junto a sus nodos, aristas y costos:

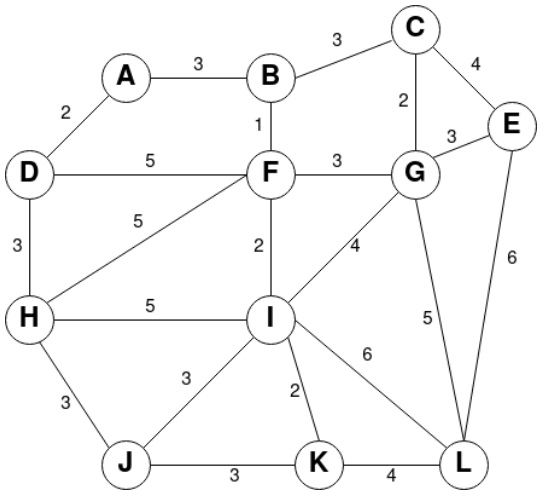


Figura 2: Representación visual de un mapa de juego.

3.2. Objetivos

A cada jugador se le asigna un objetivo al inicio de la partida. El objetivo siempre consistirá en lograr una ruta de **caminos** del color del jugador, que logre conectar una facultad con otra. Al final del juego, si logras cumplir con tu objetivo, recibirás una cantidad de `PUNTOS_OBJETIVO` puntos.

El objetivo asignado estará dado de forma **aleatoria** para cada jugador al inicio de cada partida. El objetivo será construir una ruta entre 2 nodos **distintos** donde **no exista** una arista directa entre ellos. Por ejemplo, tomando el grafo de la figura 2:

- Unir **D** con **F** no será un objetivo válido, ya que existe una arista entre el nodo **D** con el nodo **F**.
- Unir **H** con **C** sí será un objetivo válido, ya que no existe una arista entre esos dos nodos.

Al ser los objetivos aleatorios, dos jugadores podrían tener el mismo objetivo.

3.3. Desarrollo del juego

Al comienzo de la partida, se deberá mostrar el tablero junto a las facultades y caminos disponibles. El jugador inicial estará dado de manera **aleatoria**, seguido de los siguientes jugadores, manteniendo el

orden respectivo a lo largo del juego. Cada jugador solamente puede realizar **1** de las siguientes acciones por turno:

1. **Sacar carta**: El jugador puede sacar una carta que representa la cantidad de **baterías** que se le sumarán a su total de baterías. La cantidad de baterías que obtendrá el jugador será un número **aleatorio** entre `BATERIAS_MIN` y `BATERIAS_MAX`, **incluyendo** los extremos.
2. **Comprar camino**: Cada jugador puede comprar un camino en cualquier arista del grafo, siempre y cuando posea la cantidad de baterías correspondiente. Al comprar un camino, se le debe sumar inmediatamente a su **puntuación total** los puntos que se obtienen al comprar ese camino, además de disminuir las baterías del jugador producto de la compra. Una vez comprado un camino, ningún otro jugador podrá comprarlo. En caso de no poseer las baterías necesarias, la compra no se llevará a cabo y se deberá notificar en los *Logs* del servidor.

Una vez realizada alguna de las acciones anteriores, se acabará su turno y le corresponderá al siguiente jugador hacer su respectiva jugada.

3.4. Fin del juego

Una vez que todos los caminos del tablero hayan sido comprados, **inmediatamente** el juego terminará y se deberán desplegar los resultados de la partida. Antes de mostrar los puntajes de cada uno de los jugadores, se deberán sumar o restar los siguientes puntajes al total de cada jugador:

1. **Puntos de objetivo**: Cada jugador que haya cumplido con su objetivo recibirá una bonificacion de `PUNTOS_OBJETIVO` puntos. En caso de que no lo haya logrado, se deberá **restar** `PUNTOS_OBJETIVO` a su puntuación total.
2. **Ruta más larga**: El jugador que posea la ruta más larga recibirá una bonificación de `PUNTOS_RUTA_LARGA` puntos. La ruta más larga se define como la suma de los caminos consecutivos multiplicado por el costo de cada uno. A modo de ejemplo, si un jugador posee los siguientes caminos de acuerdo con la figura 2:
   - **A**-**B** de costo 3
   - **H**-**I** de costo 5
   - **G**-**I** de costo 4
   - **E**-**G** de costo 3.

   Entonces la ruta más larga de ese jugador sería **E**-**H** con un valor de 3 (E-G) + 4 (G-I) + 5 (H-I) = 12.

   Finalmente, la ruta mas larga de la partida se calculará comparando la ruta más larga de cada jugador. En caso de que uno o más jugadores empaten en la ruta más larga, se les otorgará la bonificación a todos estos jugadores.

Luego de sumar o restar estos últimos puntajes, se deberán mostrar en orden descendente el nombre del jugador junto a su puntaje, indicando el o los ganadores (en caso de empate), y dar la oportunidad de volver a jugar una nueva partida.

4. Networking

Para lograr el funcionamiento correcto de tu tarea deberás implementar una arquitectura cliente - servidor usando los protocolos **TCP/IP** a través del módulo `socket`. Este módulo permite administrar la conexión entre dos computadores conectados a una misma red local.

Deberás implementar el programa del **servidor** y el programa del **cliente**. El servidor es quien debe ejecutarse primero y recibir solicitudes de uno o más clientes. La comunicación siempre ocurrirá entre el cliente y el servidor, y **nunca directamente entre dos clientes**

4.1. Arquitectura cliente-servidor

Las siguientes consideraciones **deben ser cumplidas al pie de la letra**. Lo que no esté especificado aquí puedes implementarlo según tu criterio, siempre y cuando cumpla con lo solicitado y no contradiga nada de lo indicado (puedes preguntar si algo no está especificado o si no queda completamente claro).

4.1.1. Separación funcional

El cliente y el servidor deben estar separados, esto implica que deben estar en directorios diferentes, uno llamado `cliente` y otro llamado `servidor`. Cada directorio debe contar con los archivos y módulos necesarios para su correcta ejecución, asociados a los recursos que les correspondan, además de un archivo principal `main.py`, el cual inicializa cada una de estas entidades funcionales. La estructura que debes seguir se indica en el siguiente diagrama:

```
T3
├── cliente
│   ├── main.py
│   ├── parametros.json
│   ├── archivo(s)_solo_del_cliente.dcc
│   ├── sprites
│   └── ...
├── servidor
│   ├── main.py
│   ├── parametros.json
│   ├── archivo(s)_solo_del_servidor.dcc
│   └── ...
├── .gitignore
├── README.md
├── archivo(s)_que_no_son_del_cliente_ni_del_servidor.owo
└── ...
```

Si bien, las carpetas asociadas al cliente y al servidor se ubican en el mismo directorio (`T3`), la ejecución del cliente **no debe depender de archivos en la carpeta `servidor`**, y la ejecución del servidor **no debe depender de archivos y/o recursos en la carpeta `cliente`**. Esto implica que debes tratarlos como si estuvieran ejecutándose en computadores diferentes. La figura 3 muestra una representación esperada para los distintos componentes del programa:

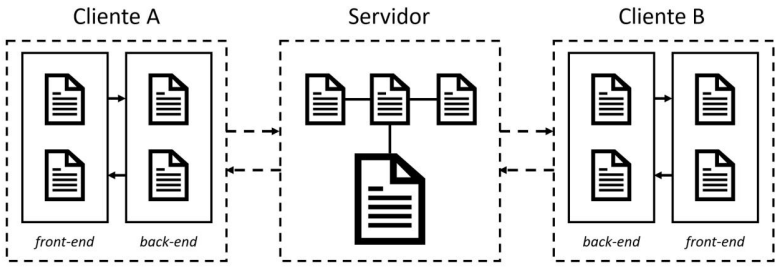


Figura 3: Separación cliente-servidor y *front-end-back-end*.

Solo el cliente contará con una interfaz gráfica. Por tanto, un cliente debe contar con una separación entre *back-end* y *front-end*. Mientras que la comunicación entre el cliente y servidor debe realizarse mediante *sockets*.

4.1.2. Conexión

El servidor contará con un archivo de formato JSON, ubicado en el directorio `servidor`, el cual contiene datos necesarios para instanciar un *socket*. El archivo es del siguiente formato:

```
{
    "host": <dirección_ip>,
    "port": <puerto>,
    ...
}
```

Por otra parte, el cliente deberá conectarse al *socket* abierto por el servidor haciendo uso de los datos encontrados en el archivo JSON del directorio `cliente`.

4.1.3. Envío de información

Cuando se establece la conexión entre el cliente y el servidor, deberán encargarse de intercambiar información entre sí constantemente. Por ejemplo, el cliente deberá comunicarle al servidor que desea sacar una carta, ante lo cual el servidor deberá responderle entregando la cantidad de baterías que obtendrá o indicando que no puede realizar dicha acción. Los mensajes codificados entre cliente y servidor **deben** seguir la siguiente estructura:

- Los primeros 4 *bytes* deben indicar el largo del contenido del mensaje, los que tendrán que estar en formato *big endian*.[2] Por contenido se entiende al mensaje inicial que debe ser enviado.
- A continuación le sigue un bloque de 4 *bytes* que contiene un número entero que indica el tipo de mensaje: `1` si corresponde a una imagen de perfil enviada por el servidor al cliente o `2` en caso de que sea un **diccionario** que contenga cualquier otro tipo de mensaje. Este bloque debe estar en formato *little endian*. El envío de imágenes por parte del servidor se explica con más detalle en la subsección Roles.
- Solo en caso de ser un mensaje correspondiente a una imagen de perfil, después del identificador del mensaje, deberás incluir un bloque de 4 *bytes* que contiene un número entero codificado en *big endian* que representa el color del avatar, según la siguiente notación:
  - 1: Azul
  - 2: Rojo
  - 3: Verde
  - 4: Amarillo
- Luego de eso, debes enviar el contenido del mensaje enviado. Este debe separarse en bloques de 100 *bytes* si corresponde a una imagen de perfil y en bloques de 60 *bytes* para un diccionario con cualquier otro tipo de mensaje. En ambos casos los bloques deben estar precedidos de 4 *bytes* que indiquen el número de bloque, comenzando a contar desde **0**, codificados en *little endian*. Si el último bloque es menor a 100 *bytes* en caso de una imagen, o 60 *bytes* en caso de un diccionario con un mensaje, deberás rellenar al final con *bytes* 0 (`b'\x 00'`), hasta completar esa cantidad.

[2] El *endianness* es el orden en el que se guardan los bytes en un respectivo espacio de memoria. Esto es relevante porque cuando intentes usar los métodos `int.from_bytes` e `int.to_bytes` deberás proporcionar el *endianness* que quieras usar, además de la cantidad de bytes que quieres usar para representarlo. Para más información puedes revisar este **enlace**.

- Si deseas transformar *strings* a *bytes* deberá utilizar UTF-8, así no tendrás problemas con las tildes ni caracteres especiales de ningún tipo.

4.1.4. Ejemplo de codificación

En la siguiente figura se muestra una representación de la estructura que debes enviar. En este caso se muestra cómo se manejaría una **imagen de perfil** de color **Rojo**.

Supongamos que al codificar la imagen a bytes, el resultado es una cadena de 250 bytes. Siguiendo el protocolo especificado, lo primero que se deberá enviar es un bloque de 4 *bytes* en *big endian* que indica el largo del contenido que se está enviando (250 bytes).

Luego se envía otro bloque de 4 *bytes*, codificado en *little endian*, que indicará que corresponde a un mensaje del tipo **imagen**, por lo que su contenido será el número `1`. Como es una imagen, se procederá a enviar un bloque extra de 4 *bytes* en *big endian* que contiene el color del avatar. En este caso, al ser de color rojo, el contenido del bloque será el número `2`.

A continuación, se separan los 250 *bytes* del contenido en bloques de 100 *bytes*, resultando en 3 bloques: los primeros dos bloques se envían de forma completa (100 x 2 = 200), pero como el tercero solo necesita enviar los 50 *bytes* restantes, se debe rellenar con 50 *bytes* 0 (`b'\x 00'`) para que el bloque alcance el largo 100 pedido.

Después de esto y para cada uno de los bloques resultantes, se deberá enviar primero el número del bloque correspondiente en un mensaje de 4 *bytes* en formato *little endian*, comenzando a contar desde el **0**. Posterior a este bloque, se debe enviar el bloque de 100 *bytes* que contiene parte de la imagen. Una vez se hayan enviado todos los bloques, el proceso de envío habrá finalizado correctamente.

En la figura adjunta se muestra de forma gráfica la composición del *bytearray* que se debe obtener como resultado a partir de este proceso de codificación:

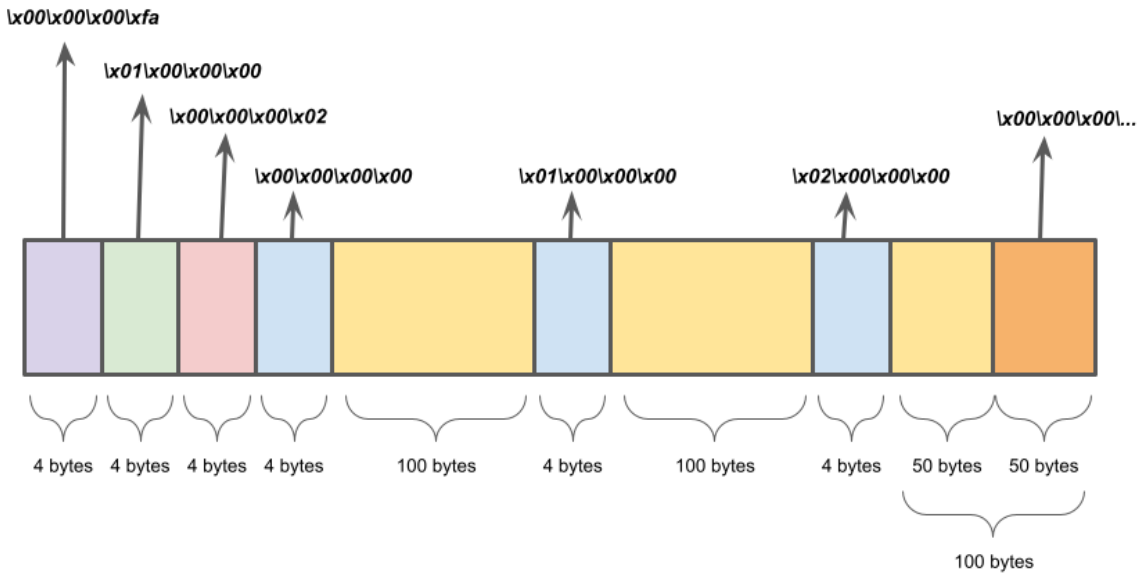


Figura 4: Ejemplo estructura del *bytearray* para una imagen de perfil

.

4.2. Módulos a completar

Con el fin de que tengas una tarea más estructurada, te entregaremos un módulo que debes completar para implementar el protocolo de codificación y decodificación de información mencionado en la sección anterior. Debes crear dos copias: una para el servidor y otra para el cliente. Estos módulos tendrán como nombre `codificacion.py`, y contienen los siguientes métodos, en donde los primeros dos se utilizan para trabajar con el envío de diccionarios con mensajes y los últimos dos para trabajar con imágenes:

- `codificar_mensaje(mensaje: dict) -> bytearray`: Esta función recibe un diccionario, el cual deberás codificar según lo especificado en Envío de información. Ante lo cual, la función deberá retornar un *bytearray* que cumpla la estructura correspondiente para un mensaje que no es una imagen de perfil.
- `decodificar_mensaje(mensaje: bytearray) -> dict`: Ese método recibe como argumento un *bytearray* que corresponda a un mensaje codificado que recibió el cliente o el servidor según corresponda, el cual **no es una imagen de perfil**. En este caso deberás encargarte de decodificar el mensaje según su estructura y retornar la estructura de datos que corresponda al mensaje decodificado original.
- `codificar_imagen(ruta: str) -> bytearray`: Este método recibe como argumento la ruta de uno de los avatares que aloja el servidor, retornará un *bytearray* que incluye la imagen codificada según el protocolo de Envío de información.
- `decodificar_imagen(mensaje: bytearray) -> list`: Este método recibe como argumento un *bytearray* que corresponde a una **imagen de perfil** codificada. Deberás decodificarlo según la estructura indicada en el protocolo y **retornar una lista con dos elementos**, donde el primero corresponde a un `bytearray` con la **imagen decodificada en bytes** y el segundo corresponde a un `int` que represente el color del avatar.

Debes implementar estas funciones al pie de la letra, respetando el formato indicado. Es decir, deberás implementarlas en el módulo entregado y cumplir con los argumentos de entrada y salida señalados. Puedes apoyarte de funciones que puedes llamar en estos métodos, pero las funcionalidades asociadas a codificación y decodificación **solo dependerán de estos métodos**. Si no cumples alguna de estas indicaciones, la sección Envío de información no será corregida y no recibirás el puntaje correspondiente.

4.2.1. *Logs* del servidor

Como el servidor no cuenta con interfaz gráfica, debe indicar constantemente lo que ocurre en él mediante la consola. Estos mensajes se conocen como mensajes de *log*, es decir, deberás llamar a la función `print` cuando ocurra un evento importante en el servidor. Estos mensajes se deben mostrar para cada uno de los siguientes eventos:

- Se conecta un cliente al servidor, debes indicar además un identificador para este.
- Un jugador indica que está listo para iniciar la partida, debes indicar el nombre del jugador.
- El programa detecta que todos los clientes están listos para comenzar la partida.
- Se da inicio a una nueva partida.
- Un cliente ingresa un nombre de usuario, debes indicar el nombre y si es válido o no.
- Se asigna un avatar a algún cliente, debes indicar el *color* de la imagen asignada y el nombre del jugador correspondiente.
- Comienza el turno de un cliente. Se debe indicar su nombre y número de turno.

- Un jugador saca una carta. Se debe indicar el nombre del jugador y el monto de baterías obtenido.
- Se compra un camino. En caso de resultar una compra exitosa se debe indicar el nombre del jugador, la arista del grafo y las baterías del jugador luego de la transacción. Si no puede realizarse la compra se debe indicar un mensaje de error, el nombre del jugador y su cantidad de baterías actual.
- Si se termina la partida, además de indicar esto debes señalar el nombre del jugador ganador.

Es de **libre elección** la forma en que representes los *logs* en la consola, un ejemplo de esto es el siguiente formato:

```
     Cliente     |        Evento          |      Detalles
-----------------|------------------------|------------------------
     pizza       |      Conectarse        |          -
-----------------|------------------------|------------------------
    aljara97     | Está listo para partir |          -
-------------------------------------------------------------------
        -        |  Término de partida    | Ganador: gatochico
-------------------------------------------------------------------
```

4.3. Roles

A continuación se detallan las funcionalidades que deben ser manejadas por el servidor y las que deben ser manejadas por el cliente.

4.3.1. Servidor

- **Asignar** una foto de perfil a cada jugador. Si bien, el servidor no cuenta con una interfaz gráfica, almacena un conjunto de imágenes que representan el avatar que identificará a cada uno de los clientes en el juego. Para esto, debe encargarse de enviar **una única imagen en bytes** al azar a cada cliente al momento de **conectarse al servidor**. Luego cada cliente deberá representar esos bytes en un `QPixmap`[3] en la **Sala de Juegos**, podrás encontrar más detalles de su representación en la sección Interfaz gráfica.
- **Procesar y validar** las acciones realizadas por los clientes. Por ejemplo, si deseamos comprobar que el jugador ingresa un nombre correcto, el servidor es quien debe verificarlo y enviarle una respuesta al cliente según corresponda.
- **Distribuir y actualizar** en tiempo real los cambios correspondientes a cada uno de los participantes del juego, con el fin de mantener sus interfaces actualizadas y que puedan reaccionar de manera adecuada. Por ejemplo, si un jugador indica que está listo para empezar a jugar, el servidor deberá notificar de esto a todos los clientes conectados y se deberá reflejar en la ventana **Sala de Espera** de cada uno de ellos.
- **Almacenar y actualizar** la información de cada cliente y sus recursos. Por ejemplo, el servidor manejará la información de los caminos ocupados por cada uno de los clientes conectados y actualizará el grafo ante una acción de un cliente.

4.3.2. Cliente

Todos los clientes cuentan con una interfaz gráfica que le permitirá interactuar con el programa y enviar mensajes al servidor. Maneja las siguientes acciones:

[3] El método `loadFromData` podría serte útil.

- **Enviar todas las acciones** realizadas por el usuario hacia el servidor.
- **Recibir e interpretar** las respuestas y actualizaciones que envía el servidor.
- **Actualizar** la interfaz gráfica de acuerdo a las respuesta que recibe del servidor.

5. Interfaz gráfica

El cliente deberá poseer una interfaz gráfica implementada con **PyQt5**.

Los ejemplos expuestos en esta sección son solo para que tengas una idea de cómo podría verse cada ventana. Eres libre de modificar o diseñar las ventanas a tu gusto, siempre y cuando cumplan con los requerimientos mínimos.

Las ventanas a implementar son:

5.1. Ventana de Inicio

Se despliega en el inicio del programa. Debe permitir que se introduzca el nombre del jugador, que será con el que se identificará en el servidor y que se mostrará a los demás jugadores de la partida. Un nombre de usuario debe tener una extensión menor a **15 carácteres** y **no estar siendo utilizado por algún otro jugador**. Si el nombre es válido, se prosigue con el flujo del juego a la Sala de espera, mientras que en caso contrario se debe avisar al usuario y permitir ingresar otro nombre.

Figura 5: Ventana de Inicio del Juego.

Finalmente, en caso de que un usuario se conecte y la partida ya haya comenzado o que la sala ya esté llena, se le debe notificar y mantener en la ventana de inicio.

5.2. Sala de Espera

Aparece luego de introducir el nombre en la ventana de inicio. En esta sala se despliegan los nombres de todos los jugadores que se van uniendo a la partida actual. Debe mostrar la información de los dos mapas disponibles a jugar con su nombre, y dar la opción de votar por cuál es el que el jugador desea jugar. La

interfaz debe actualizarse para reflejar la cantidad de votos que lleva cada mapa, y de quien ya ha votado y quien no. En caso de que exista un empate de votos en los mapas, el servidor elegirá de forma aleatoria cuál se jugará.

Para poder hacer inicio de la partida deben estar todos los jugadores en **LISTO**. La forma de obtener este estado dependerá si el jugador es **Host** (fue primero en ingresar a la sala de espera) o **normal**:

1. **Jugador normal:** Para que el jugador cambie su estado de **PENDIENTE** a **LISTO** debe haber votado por uno de los mapas disponibles. Mientras no vote su estado permanecerá como pendiente.

Figura 6: Ejemplo de Sala de Espera normal

2. **Jugador Host:** También debe votar para obtener el estado de **LISTO**. Adicionalmente, su interfaz debe tener un botón **iniciar** que se habilita cuando todos los jugadores de la partida estén listos.

Figura 7: Ejemplo de Sala de Espera host

Luego de iniciar partida, todos los jugadores avanzan a la sala de Juegos.

5.3. Sala de Juegos

En esta sala es donde se desarrollará toda la partida. Debe tener:

- Una sección para hacer compras, donde se podrá seleccionar el camino que se desea comprar. Puedes implementarlo de la forma que desees.
- Una sección con el nombre del mapa y el nombre del jugador que está jugando en ese turno.
- Un resumen de cada uno de los jugadores con el turno que se le fue asignado aleatoriamente al iniciar la partida, su avatar asignado de forma aleatoria que determinará su color en el tablero, su nombre ingresado en la ventana de inicio y la cantidad de baterías que posee cada uno.
- Una sección con la información propia, donde se despliega el objetivo a cumplir dentro de la partida, que dependiendo si se cumplió o no se mostrará en verde o amarillo, respectivamente. También se muestra la cantidad de baterías disponibles, el turno asignado al jugador, puntaje actual y un botón de **SACAR CARTA**, el cual entregará una cantidad de baterías **aleatoria** entre `BATERIAS_MIN` y `BATERIAS_MAX`, incluyendo extremos.
- Un área con el mapa donde se deben marcar los caminos[4] que obtengan cada uno de los jugadores, con el color de su avatar correspondiente, y actualizarse para todos cada vez que un jugador adquiera un camino nuevo.

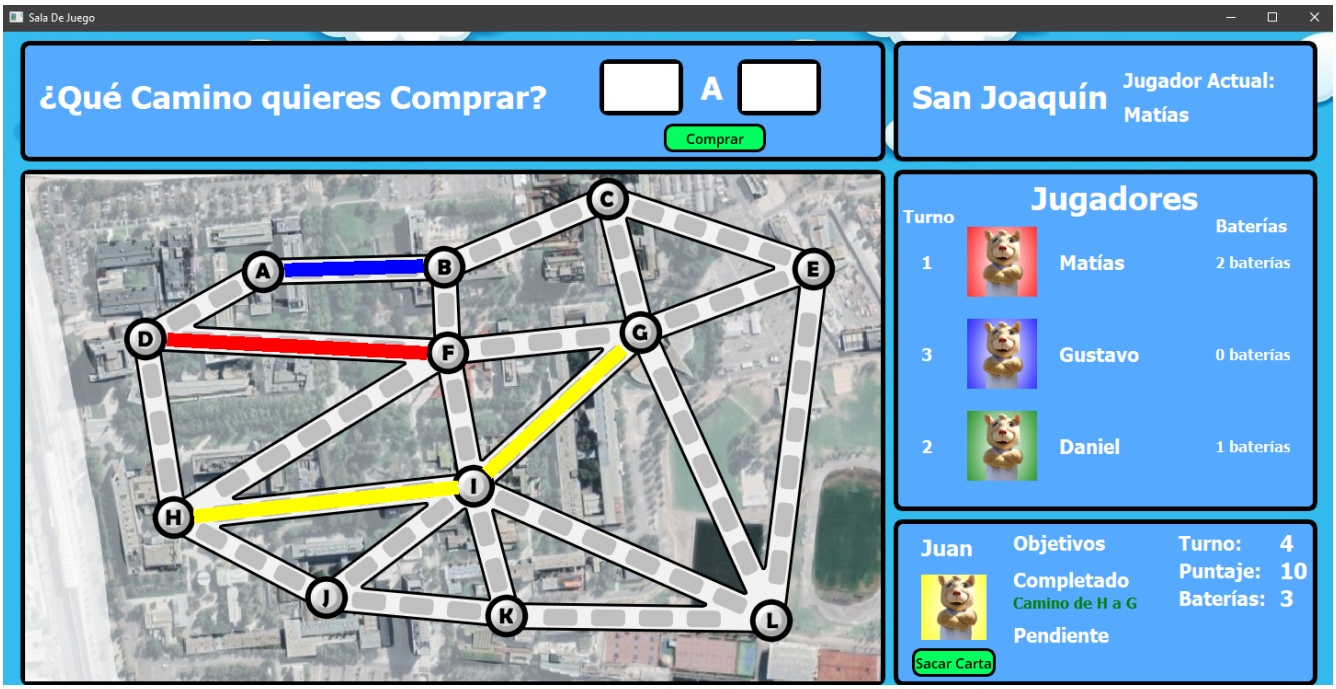


Figura 8: Ejemplo de Sala de Juego

5.4. Fin de Partida

Esta ventana muestra al jugador ganador, junto con el orden de cada uno de los lugares de los demás participantes y su puntaje correspondiente. Además, posee un botón **JUGAR OTRA VEZ** que redirigirá a la sala de espera para poder iniciar otra partida y definir quién será el mejor en un nuevo encuentro.

[4] La función **drawLine** de Qpainter te podría ser útil para dibujar caminos.

Figura 9: Ventana Fin de Partida

6. Archivos

Para desarrollar tu programa de manera correcta, deberás crear y utilizar los siguientes archivos:

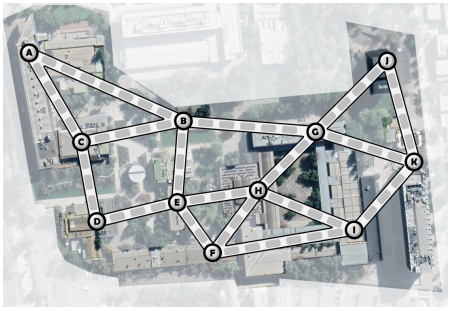
6.1. Archivos entregados

6.1.1. Sprites

Carpeta que contiene todos los elementos visuales que necesitas para llevar a cabo tu tarea, entre ellos encontrarás las subcarpetas:

- **Avatares**: Contiene los distintos avatares que deberás asignar a cada jugador de la partida, su formato es de la forma `avatar-x`, siendo `x` el número del jugador correspondiente. Los avatares deben ser almacenados por el servidor, y deben ser enviados al cliente mediante el protocolo definido en Envío de información.
- **Mapas**: Contiene los dos mapas de juego que serán utilizados siguiendo el formato `Mapa <ABREVIACIÓN>`. Estos mapas ya contienen nodos y caminos posicionados a modo de ayudarte a visualizar el grafo de cada mapa.
- **Logo**: Contiene el logo del juego en formato transparente y con fondo blanco, por si deseas utilizarlo.

(a) *Sprite* para avatar de juego.



(b) *Sprite* para mapa de juego

Figura 10: Ejemplos de *sprites* para cada carpeta.

6.1.2. `mapa.json`

Corresponde a un archivo en formato `JSON` que contendrá toda la información relevante sobre cada uno de los mapas de juego. Este seguirá la siguiente estructura:

```
{
    "mapa": {
        "<nombre_mapa>": {
            "numero_nodos" : int,
            "posiciones": {
                "A": {"x": <pos-x>, "y": <pos-y>},
                ...
            },
            "caminos": {
                "A": [["<nodo_conectado_1>", <cantidad_baterias>], ...],
                ...
            }
        }
    }
}
```

Donde:

- La llave `'<nombre_mapa>'` tomará los valores de `'san_joaquin'` o `'ingenieria'`, ambos *strings*.
- Dentro de cada mapa, hay tres llaves que permitirán acceder a distintas informaciones sobre este:
  - La llave `'numero_nodos'` permite acceder al número de nodos que hay en el mapa de juego.
  - La llave `'posiciones'` nos permite acceder a las distintas posiciones de cada nodo en el mapa, llamados con letras desde la A a la Z en orden (Por ejemplo, si San Joaquín tiene 12 nodos, estarían llamados desde la A a la L). Estas posiciones podrán ser accedidas por las llaves "x" e "y".

    Las posiciones están entregadas de forma porcentual relativa a la imagen del mapa entregado. De este modo, si un nodo tiene posiciones `{'x': 0.5, 'y': 0.25}`, quiere decir que se encontrará a la mitad del ancho del mapa y a un cuarto del alto del mapa.
  - Finalmente, la llave `'caminos'` nos permite acceder a un diccionario donde el nombre de cada nodo es una llave para una lista de listas, que indican los nodos con que estos se conectan y la cantidad de baterías necesarias para comunicar uno con otro. Por ejemplo, en el caso del nodo "A" del mapa "San Joaquín" se tendría:

    `'A': [['B', 3], ['D', 2]]`

    Lo cual significa que el nodo "A" se conecta tanto al nodo "B" como al nodo "D" y que se necesitan 3 baterías para comprar el camino "A-B" o bien 2 baterías para comprar "A-D".

6.1.3. `codificacion.py`

Este módulo contiene la declaración de las funciones especificadas en Módulos a completar. Deberás completar el archivo con tu implementación para cada función que viene dentro del módulo, procurando seguir el formato pedido de *input* y *output* para cada una. Se recuerda que puedes usar funciones auxiliares adicionales siempre y cuando el proceso de codificación y decodificación sea realizado por las funciones ya definidas.

6.2. Archivos a crear

6.2.1. `parametros.json`

A lo largo del enunciado se han presentado distintos números y palabras en `ESTE_FORMATO`, estos son conocidos como **parámetros** del programa y son valores que permanecerán constantes a lo largo de toda la ejecución de tu código.

Dentro de tu tarea deben existir dos archivos de parámetros independientes entre sí: uno para el servidor, y otro para el cliente. Cada archivo de parámetros debe contener solamente los parámetros que corresponden a su respectiva parte, es decir, `parametros.json` en la carpeta `cliente/` deberá contener solamente parámetros útiles para el cliente mientras que `parametros.json` de `servidor/` contendrá solamente parámetros útiles para el servidor, tal como se explica en detalle en la Sección 4: Networking.

Los archivos deben encontrarse en formato `JSON`, como se ilustra a continuación:

```
{
    "host": <dirección_ip>,
    "port": <puerto>,
    ...
}
```

7. *Bonus*

En esta tarea habrá una serie de *bonus* que podrás obtener. Cabe recalcar que necesitas cumplir los siguientes requerimientos para poder obtener *bonus*:

1. La nota en tu tarea (sin bonus) debe ser **igual o superior a 4.0**[5].
2. El bonus debe estar implementado **en su totalidad**, es decir, **no se dará puntaje intermedio**.

Finalmente, la cantidad máxima de décimas de *bonus* que se podrá obtener serán 8 décimas. Deberás indicar en tu `README` si implementaste alguno de los bonus, y cuáles fueron implementados.

7.1. GIF de celebración (2 décimas)

Después de tanto jugar a *DCCópteros* te comienzas a aburrir de las imágenes estáticas de Pangui[6] que hay en todo el juego. Así no es como lo recordabas cuando estaba en el campus, por lo que te decides a darle un poco más de vida con una celebración al final de una partida.

Para implementar este bonus se pide mostrar el gif[7] incluido en la carpeta `sprites/Bonus` en la ventana del fin de partida en conjunto a toda la información previamente pedida en su implementación.

[5] Esta nota es sin considerar posibles descuentos.
[6] Alias, la mascota de la universidad.
[7] O jif, llif, como desees llamarle.

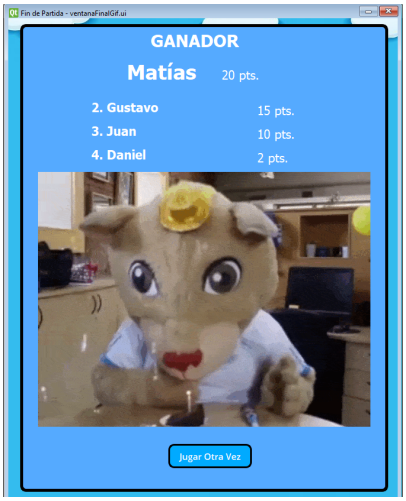


Figura 11: Ventana Fin de Partida con Gif

7.2. Turnos cronometrados (5 décimas)

Buscando otorgarle mayor dificultad al juego, decides presionar a los jugadores otorgándoles un tiempo máximo para poder desempeñar su turno. Este tiempo debe estar indicado en el parámetro `TIEMPO_TURNO` y el turno deberá automáticamente terminar una vez terminado el contador, sin importar si el jugador realizó una acción o no. En caso de que el jugador realice una acción antes de que acabe este tiempo máximo, el turno también debe acabarse inmediatamente, sin importar la cantidad de tiempo por turno que quede.

La cantidad restante de tiempo deberá ser mostrada en la interfaz de todos los jugadores y actualizarse en tiempo real. Finalmente, si deseas adornar un poco más la interfaz se incluye un *sprite* de un reloj en la carpeta `sprites/Bonus`.

8. Avance de tarea

Para esta tarea, el avance corresponde a implementar el **inicio de la interacción con** *DCCópteros*, con tanto un servidor como cliente implementado.

En específico, para este avance deberás entregar un servidor capaz de conectarse a múltiples clientes, que sea capaz de recibir nombres para ingresar a la sala de espera y les permita ingresar solo si el nombre cumple las condiciones pedidas (no estar siendo utilizado por otro usuario y tener un largo máximo de 15 carácteres). También deberás entregar un cliente que tenga implementadas la Ventana de Inicio y la Sala de Espera, y pueda interactuar con el servidor para poder ingresar y mostrar la Sala de Espera.

Para fines de este avance, no es necesario implementar la Selección de Mapas en la Sala de Espera, ni una actualización en vivo de los usuarios dentro de ella. Tampoco es necesario que la información que sea enviada entre cliente y servidor siga el protocolo de envío de mensajes.

A partir de los avances entregados, se les brindará un *feedback* general de lo que implementaron en sus programas y además, les permitirá optar por **hasta 2 décimas** adicionales en la nota final de su tarea.

9. `.gitignore`

Para esta tarea **deberás utilizar un** `.gitignore` para ignorar los archivos indicados, este deberá estar dentro de tu carpeta `Tareas/T3/`. Puedes encontrar un ejemplo de `.gitignore` en el siguiente ***link***.

Para esta tarea, deberás ignorar en específico los siguientes archivos:

- Enunciados
- `sprites`
- `mapa.json`

Recuerda que no debes ignorar el archivo `codificacion.py` ni los archivos `parametros.json`, ya que de lo contrario no se podrá corregir tu tarea adecuadamente.

Se espera que no se suban archivos autogenerados por las interfaces de desarrollo o los entornos virtuales de Python, como por ejemplo: la carpeta `__pycache__`.

Para este punto es importante que hagan un correcto uso del archivo `.gitignore`, es decir, los archivos no **deben** subirse al repositorio debido al archivo `.gitignore` y no debido a otros medios.

10. Entregas atrasadas

Posterior a la fecha de entrega de la tarea se abrirá un formulario de Google Form, en caso de que desees que se corrija un *commit* posterior al recolectado, deberás señalar el nuevo *commit* en el *form*.

El plazo para rellenar el *form* será de 16 horas, en caso de que no lo contestes en dicho plazo, se procederá a corregir el *commit* recolectado.

11. Importante: Corrección de la tarea

Para esta tarea, el carácter funcional del programa será el pilar de la corrección, es decir, **sólo se corrigen tareas que se puedan ejecutar**. Por lo tanto, se recomienda hacer periódicamente pruebas de ejecución de su tarea y *push* en sus repositorios.

Cuando se publique la distribución de puntajes, se señalará con color:

- **Amarillo**: cada ítem que será evaluado a nivel de código, todo aquel que no esté pintado de amarillo significa que será evaluado si y sólo si se puede probar con la ejecución de su tarea.
- **Morado**: cada ítem que será evaluado mediante corrección automática, es decir, se evaluará mediante la implementación correcta del archivo `codificacion.py`

En tu archivo `README.md` deberás señalar el archivo y la línea donde se encuentran definidas las funciones o clases relacionados a los ítems en amarillo.

Se recomienda el uso de Logs del Servidor para ver los estados del sistema para apoyar la corrección. De todas formas, ningún ítem de corrección se corrige puramente por consola, ya que esto no valida que un resultado sea correcto necesariamente. Para el cliente, todo debe verse reflejado en la interfaz, pero el uso de *logs* también es recomendado.

Finalmente, en caso de que no logres completar el Envío de información, puedes enviar los mensajes mediante codificación directa para que así no afecte otras funcionalidades de la tarea, pero implicará que **no obtengas puntaje en los ítems relacionados**.

Finalmente, si durante la realización de tu tarea se te presenta algún problema o situación que pueda afectar tu rendimiento, no dudes en contactar al ayudante jefe de Bienestar a su ***mail*** (`bienestar.iic2233@ing.puc.cl`).

12. Restricciones y alcances

- Esta tarea es **estrictamente individual**, y está regida por el **Código de honor de Ingeniería**.
- Tu programa debe ser desarrollado en Python 3.7.
- Tu programa debe estar compuesto por uno o más archivos de extensión `.py`.
- Si no se encuentra especificado en el enunciado, supón que el uso de cualquier librería Python está prohibido. Pregunta en la *issue* especial del **foro** si es que es posible utilizar alguna librería en particular.
- Debes adjuntar un archivo `README.md` **conciso y claro**, donde describas los alcances de tu programa, cómo correrlo, las librerías usadas, los supuestos hechos, y las referencias a código externo. **Tendrás hasta 2 horas después del plazo de entrega** de la tarea para subir el `README` a tu repositorio.
- Tu tarea podría sufrir los descuentos descritos en la **guía de descuentos**.
- Entregas con atraso de más de 16 horas tendrán calificación mínima (1,0).
- Cualquier aspecto no especificado queda a tu criterio, siempre que no pase por sobre otro.

Las tareas que no cumplan con las restricciones del enunciado obtendrán la calificación mínima (1,0).