



PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC2233 Programación Avanzada (2020-2)

Tarea 03

Entrega

- **Avance de tarea**

- **Fecha y hora:** miércoles 25 de noviembre de 2020, 20:00
- **Lugar:** Repositorio personal de GitHub — Carpeta: Tareas/T03/

- **Tarea**

- **Fecha y hora:** sábado 5 de diciembre de 2020, 20:00
- **Lugar:** Repositorio personal de GitHub — Carpeta: Tareas/T03/

- **README.md**

- **Fecha y hora:** lunes 7 de diciembre de 2020, 20:00
- **Lugar:** Repositorio personal de GitHub — Carpeta: Tareas/T03/

Objetivos

- 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. <i>DCColonos</i> | 4 |
| 2. Flujo del programa | 4 |
| 3. Mapa | 5 |
| 3.1. Nodos | 5 |
| 3.2. Caminos | 5 |
| 3.3. Hexágonos | 5 |
| 4. Reglas de <i>DCColonos</i> | 6 |
| 4.1. <i>Ítems</i> del juego | 6 |
| 4.1.1. Tipos de cartas | 6 |
| 4.1.2. Carreteras | 7 |
| 4.1.3. Chozas | 7 |
| 4.1.4. Banco | 7 |
| 4.2. Desarrollo del juego | 7 |
| 4.2.1. Preparar el mapa | 7 |
| 4.2.2. Comienza la partida | 8 |
| 4.2.3. Lanzamiento de dados | 8 |
| 4.2.4. Acciones posibles durante el turno del jugador | 8 |
| 4.2.5. Fin del turno | 9 |
| 5. Networking | 9 |
| 5.1. Arquitectura cliente-servidor | 9 |
| 5.1.1. Separación funcional | 9 |
| 5.1.2. Conexión | 10 |
| 5.1.3. Envío de información | 11 |
| 5.1.4. Ejemplo de codificación | 11 |
| 5.1.5. <i>Logs</i> del servidor | 12 |
| 5.1.6. Desconexión repentina | 13 |
| 5.2. Roles | 13 |
| 5.2.1. Servidor | 13 |
| 5.2.2. Cliente | 13 |
| 6. Interfaz | 14 |
| 6.1. Sala de espera | 14 |
| 6.2. Sala de juego | 14 |
| 6.3. Fin de partida | 15 |
| 7. Archivos | 16 |
| 7.1. Archivos a crear | 16 |
| 7.1.1. parametros.json | 16 |
| 7.2. Archivos entregados | 17 |
| 7.2.1. generador_de_cartas.py | 17 |
| 7.2.2. Sprites | 17 |
| 7.3. grafo.json | 19 |
| 7.4. generador_grilla.py | 19 |
| 8. Bonus | 20 |
| 8.1. Ladrón (4 décimas) | 20 |
| 8.2. Ciudades (2 décimas) | 20 |
| 8.3. Chat (4 décimas) | 21 |
| 9. Avance de tarea | 21 |
| 10..gitignore | 21 |

| | |
|----------------------------------------|----|
| 11. Entregas atrasadas | 22 |
| 12. Importante: Corrección de la tarea | 22 |
| 13. Restricciones y alcances | 23 |

1. *DCColonos*

Apagas la televisión. Lo que acabas de oír te decepciona: los lugares de entretenimiento no abrirán hasta marzo del 2021. ¿Cómo podrás demostrar tus destrezas de conquistador a aquella persona que deseas enamorar? Por suerte, gracias a Programación Avanzada, recuerdas que con tus habilidades de **networking** y **serialización**, puedes crear un juego de mesa **100 online**, lo suficientemente elaborado como para conquistar el corazón de esa persona que tanto añoras. El nombre del juego es... **DCColonos**.



Figura 1: Logo de *DCColonos*

DCColonos es un juego de estrategia donde el jugador que logre ser el mejor colono será el ganador. Deberás usar tus habilidades de negociación para intercambiar los materiales necesarios y así poder construir y poblar hasta vencer.

2. Flujo del programa

DCColonos corresponde a un **juego de mesa multijugador por turnos**, en donde los jugadores compiten para conquistar y expandir sus territorios sobre un mapa.

Al ser la interacción entre jugadores vital para el juego, es necesario que el programa cuente con un **servidor** que pueda transmitir datos entre jugadores y manejar el flujo del juego en sí, por lo que siempre se debe comenzar ejecutando este servidor previo a la ejecución del programa por parte de los jugadores.

A continuación, cada jugador debe correr una instancia de **cliente**, programa que iniciará una interfaz gráfica, la cual será su único medio de interacción con el juego. Luego de ejecutarla, si el servidor tiene espacio suficiente como para aceptar a este nuevo jugador, se abrirá una **Sala de espera** donde se pueden ver al resto de jugadores conectados, con un nombre único asignado por el servidor. En caso contrario, se abrirá una ventana con un mensaje indicando que ya empezó una partida, dando la posibilidad de cerrar el programa.

Una vez dentro de la **Sala de espera**, los jugadores deberán esperar a que ingrese el número necesario de jugadores para iniciar la partida, valor definido por el parámetro **CANTIDAD_JUGADORES_PARTIDA**. Cuando este valor se alcanza, el servidor redirigirá inmediatamente a todos los jugadores a la **Sala de juego**, donde podrán dar inicio a una nueva partida de *DCColonos*.

Una partida de *DCColonos* se divide en dos etapas principales. En la primera, el servidor deberá cargar el **Mapa** del juego a partir de un **grafo no dirigido**, luego asignar de manera aleatoria diversos atributos a cada celda de este mapa, repartir las construcciones iniciales de cada jugador sobre este, para finalmente entregarle a cada jugador sus cartas iniciales y definir el orden de los turnos. En la segunda etapa y siguiendo el orden asignado, cada jugador deberá lanzar un par de dados para intentar **recibir recursos** y tendrá la posibilidad de realizar diversas acciones, como **comprar construcciones**, **cartas**

especiales o realizar intercambios con otro jugador, logrando acumular puntos de victoria. Esta última etapa se repetirá hasta que algún jugador logre alcanzar los **puntos de victoria** necesarios para ganar, coronándose como ganador de *DCColonos*.

3. Mapa

Para jugar una partida de *DCColonos*, necesitarás primero armar un **tablero** que simulará el **mapa** sobre el cuál implementarás tus poderosas estrategias de conquista. Deberás representar este mapa mediante un **grafo no dirigido**, compuesto de nodos conectados entre sí de tal forma que generan una **grilla hexagonal** con la cual los jugadores podrán interactuar para situar construcciones de diversa índole.

A continuación se describen las partes más relevantes de este mapa:

3.1. Nodos

Los nodos corresponden a las unidades fundamentales del mapa, representando terrenos aislados sobre los cuales cada jugador puede construir chozas y así apoderarse de dichos terrenos.

Cada nodo debe almacenar como mínimo el número de **id** que le identifica, su **estado actual** (libre u ocupado por una choza) y de estar actualmente ocupado, el **nombre de usuario** del jugador que se ha apoderado de este. Fuera de estos elementos anteriores, siéntete libre de almacenar cualquier otro tipo de información que creas puede ser necesaria para un correcto desarrollo de tu programa.

3.2. Caminos

Los caminos corresponden a las aristas que conectan los nodos del mapa, representando caminos geográficos que permiten viajar entre sectores de terreno aislados. A medida que el juego avanza, cada jugador puede invertir sus recursos para transformar un camino en una carretera y así expandir su alcance en el mapa, aumentando su poder de conquista y acercándolos a la victoria.

3.3. Hexágonos

Corresponden a agrupaciones de exactamente 6 nodos cuyas conexiones forman un hexágono en el mapa. Es importante notar que al ser dicho mapa una **grilla**, puede ocurrir que un **nodo** pertenezca a más de un hexágono.

Cada hexágono tiene asociado un **id** que le identifica, seis **nodos** que lo conforman en el mapa, un **número de ficha** que, al salir en el lanzamiento de los dados, definirá en qué ocasiones el hexágono entregará recursos a sus chozas aledañas durante la partida, según el **tipo de materia prima** que tenga asignado dicho hexágono (madera, arcilla, trigo). Esta materia prima será asignada de forma aleatoria, mientras que el número de ficha también se deberá asignar de manera aleatoria, siendo un número del **2** al **12**, sin considerar el **7**. Por último, el mapa con las asignaciones finales de tanto materias primas como números de fichas deberán seguir las restricciones indicadas en [Preparar el mapa](#).

Para instanciar el mapa, se te entrega el archivo **grafo.json**, que contiene la cantidad de filas y columnas que posee el mapa, la lista de adyacencia de este y los nodos que componen cada hexágono. Puedes asumir que el mapa de *DCColonos* siempre será el mismo, siendo su representación visual la indicada en la Figura 2. También podrás encontrar un ejemplo de su implementación en la interfaz en la sección [Sala de juego](#).

Finalmente, podrás hacer uso del módulo **generador_grilla.py** entregado para facilitar la visualización del mapa en tu interfaz.

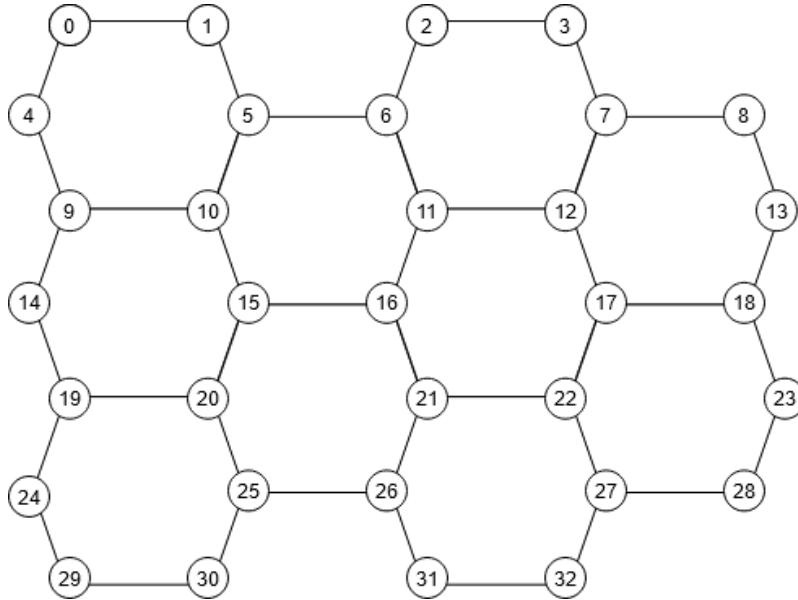


Figura 2: Representación visual del mapa completo

4. Reglas de *DCColonos*

DCColonos posee un tablero que representa un mapa, cartas muy entretenidas, construcciones varias y un par de reglas a seguir durante la partida. El objetivo del juego es ser la primera persona que logre alcanzar un total de **PUNTOS PARA VICTORIA** puntos, conocidos como **puntos de victoria**.

El funcionamiento del juego se presenta a continuación:

4.1. Ítems del juego

Los ítems son elementos con los cuales los jugadores podrán interactuar o colecciónar durante la partida. Estos pueden ser **cartas**, **construcciones** como las chozas y carreteras, o entidades como el **banco**.

4.1.1. Tipos de cartas

En *DCColonos* existen dos tipos de cartas: **materias primas** y **eventos especiales** que beneficiarán al jugador. En específico, estas son:

- **Cartas de materia prima:** cada una de estas cartas representa una unidad de materia prima que puede ser acumulada por los jugadores y sirve para construir chozas o comprar cartas de desarrollo. Hay tres tipos de materia prima: **Arcilla**, **Madera** y **Trigo**.
- **Cartas de desarrollo:** estas cartas poseen efectos especiales para el jugador y al momento de comprarlas, no sabremos de qué tipo saldrá. Su efecto se aplica de forma inmediata una vez adquirida la carta y para comprar una se necesita de una cantidad de **CANTIDAD TRIGO CARTA DESARROLLO** cartas de **trigo**, **CANTIDAD MADERA CARTA DESARROLLO** cartas de **madera** y **CANTIDAD ARCILLA CARTA DESARROLLO** cartas de **arcilla**. Los tipos de cartas de desarrollo son:
 - **Punto de victoria:** esta carta otorga **+1 punto** de victoria para el jugador.
 - **Monopolio:** el jugador que saque esta carta deberá elegir un tipo de materia prima. Los demás jugadores deberán automáticamente darle todas las cartas que tengan de esta materia prima.

4.1.2. Carreteras

Como se indicó anteriormente, un **camino** corresponde a las aristas entre dos o más nodos del mapa. A lo largo de la partida, un jugador puede construir una **carretera** sobre lo que antes era un camino, expandiendo sus territorios en el mapa y aumentando sus chances de victoria.

Cada carretera debe ubicarse sobre un camino y, para poder ser construida, debe ubicarse junto a una choza o junto a otra carretera ya existente del mismo jugador, es decir, no se puede construir de manera aislada. Para construir una carretera, se necesitarán **CANTIDAD_ARCILLA_CARRETERA** y **CANTIDAD_MADERA_CARRETERA** unidades de arcilla y madera, respectivamente. Cada carretera construida aportará **+1 punto** de victoria al jugador que la haya construido.

Adicionalmente, se bonificará al jugador que actualmente tenga la carretera más larga construida, lo que se refleja en la **mayor cantidad de carreteras consecutivas construidas**. Las carreteras consecutivas se definen como aquellas que no estén siendo interrumpidas por caminos sin construir, carreteras o chozas de otro jugador. El jugador que cumpla con la carretera más larga tendrá **+2 puntos** de victoria adicionales, perdiéndolos en el caso de que otro jugador le quite el logro.

4.1.3. Chozas

Las chozas son las construcciones más sencillas de *DCColonos* y la base para comenzar tu conquista épica. Cada jugador comienza una partida con **2** chozas ubicadas aleatoriamente sobre el mapa, siendo posible adquirir más de estas construcciones a lo largo del juego a cambio de materias primas.

Al momento de ubicar una choza en el tablero, esta no se puede colocar en un nodo adyacente a otro con una choza ya construida. En otras palabras, debe haber como mínimo **dos caminos** de distancia entre cada choza.

Para construir una choza se necesitan de **CANTIDAD_ARCILLA_CHOZA** unidades de arcilla, **CANTIDAD_MADERA_CHOZA** unidades de madera, y **CANTIDAD_TRIGO_CHOZA** unidades de trigo. Cada choza ubicada en el mapa aportará **+1 punto** de victoria para su dueño.

4.1.4. Banco

El banco es una entidad del juego que no es manejada por ningún jugador, sino que por el servidor. Su objetivo es apoyar a los jugadores en el desarrollo de la partida, mediante la repartición de cartas a cada jugador al comienzo del juego, repartir las cartas de materia prima a los jugadores que tengan chozas ubicadas en los terrenos con el número igual al obtenido por los dados, vender chozas, carreteras o cartas de desarrollo al jugador en turno, entre otras cosas.

4.2. Desarrollo del juego

A continuación se presentan las etapas que componen el flujo de una partida de *DCColonos*, ordenadas cronológicamente:

4.2.1. Preparar el mapa

Para comenzar una partida de *DCColonos*, el servidor deberá primero preparar el mapa sobre el cual se desarrollará el juego. Para esto, se debe instanciar el grafo con los nodos y aristas correspondientes. Además, a cada hexágono se le deberá asignar un **número de ficha** aleatorio en el rango $[2, 6] \cup [8, 12]$ (enteros entre el 2 y el 12, excluyendo el número 7) y una **materia prima** asociada, siguiendo estas restricciones:

- No pueden haber dos hexágonos con el mismo número de ficha.

- Cada tipo de materia prima debe estar, como mínimo, en tres hexágonos.

4.2.2. Comienza la partida

Una vez listo el mapa, el servidor primero deberá elegir de forma aleatoria el orden que tendrán los turnos de los jugadores. Luego, por cada jugador el servidor deberá ubicar de forma aleatoria en el mapa [2](#) chozas y [2](#) carreteras, pero cumpliendo con las restricciones detalladas anteriormente para chozas y carreteras.

Una vez acomodadas las construcciones de cada jugador, el banco deberá repartir las cartas de materia prima correspondientes a cada jugador. Esto quiere decir que por cada nodo en donde haya asentada una choza, cada jugador recibirá una cantidad [GANANCIA_MATERIA_PRIMA](#) de la materia prima de los hexágonos que tengan asociado ese nodo. Si una choza se ubica en un nodo que pertenece a más de un hexágono, el jugador obtendrá materia prima por cada hexágono al que pertenezca el nodo.

Una vez finalizada la distribución de materia prima, comenzarán los turnos individuales de cada jugador, nuevamente siguiendo el orden definido anteriormente.

4.2.3. Lanzamiento de dados

Cada turno individual comienza cuando un jugador lanza dos dados de seis caras cada uno, siendo el valor total del lanzamiento el resultado de la suma de los valores obtenidos en cada dado. Antes del lanzamiento, el jugador no debe poder realizar ninguna otra acción.

Según el número obtenido, pueden ocurrir los siguientes eventos:

- Si el número obtenido es distinto de 7 y existe una choza en alguno de los nodos del hexágono cuyo **número de ficha** sea igual al número obtenido por los dados, el banco debe repartir una cantidad [GANANCIA_MATERIA_PRIMA](#) de cartas de la materia prima correspondiente a ese hexágono al dueño de la choza. Si un jugador posee más de una choza en ese mismo hexágono, el monto a repartir se deberá multiplicar por la cantidad de chozas que tenga.
- Si el número obtenido corresponde a 7, el o los jugadores con 8 o más cartas deberán devolver la mitad de ellas de manera aleatoria al banco. Es decir, no podrán elegir qué cartas devolver.

4.2.4. Acciones posibles durante el turno del jugador

Luego de lanzar los dados y aplicarse su efecto inicial, el jugador al que le corresponda el turno podrá realizar **una** acción antes de dar finalizado su turno. Las acciones disponibles son:

- **Comprar una carta de desarrollo:** se debe entregar la cantidad de cartas indicada en la sección [Tipos de cartas](#) al banco, y este utilizará la función [generador_de_cartas.py](#) para entregar de vuelta una carta de desarrollo (monopolio o puntos de victoria). Los efectos de esta carta tendrán efecto de forma inmediata y la carta procederá a desaparecer del mazo del jugador.
- **Intercambiar con jugadores:** deberá existir una opción que te permita intercambiar cartas con los demás jugadores. Al momento de seleccionarla, se deberá abrir una nueva ventana o *pop-up*¹ donde podrás seleccionar la materia prima que ofreces y su cantidad, junto con la materia prima que esperas a cambio, su cantidad y el jugador con el cual deseas intercambiar. Luego de enviar la solicitud, se le abrirá una ventana al jugador seleccionado con los datos del intercambio, donde deberá aceptar o rechazar dicha solicitud. **Ten en cuenta que solo podrás realizar tanto ofrecer como pedir un tipo de materia prima a la vez** Además, deberás verificar que tanto tú como el jugador con el cual deseas intercambiar tengan los recursos y cantidades necesarias para llevar a cabo el intercambio. De lo contrario, se deberá notificar que el intercambio es inválido.

¹Para implementarlo te recomendamos investigar sobre la clase [QDialog](#) de la librería PyQt5

- **Construir carretera:** se podrá seleccionar un camino en el mapa sobre el cual ubicar una carretera, siempre y cuando se tengan los recursos necesarios y se cumplan las condiciones especificadas en la sección [Carreteras](#). En caso de no cumplirse alguna condición, se deberá notificar al jugador que la acción no fue posible y dar la oportunidad de intentarlo de nuevo.
- **Construir choza:** se podrá seleccionar un nodo en el mapa sobre el cual ubicar una nueva choza, siempre y cuando se tengan los recursos necesarios y se cumplan las condiciones especificadas en la sección [Chozas](#). En caso de no cumplirse alguna condición, se deberá notificar al jugador que la acción no fue posible y dar la oportunidad de intentarlo de nuevo.

4.2.5. Fin del turno

Al finalizar el turno un jugador, se deberán contar y actualizar los puntos de victoria de todos los jugadores. También se deberá verificar si algún jugador ha logrado obtener a lo menos [PUNTOS_VICTORIA_FINALS](#) puntos de victoria y así declararse como ganador de la partida.

Dependiendo de sus puntos de victoria actuales, el programa realizar diferentes acciones dependiendo de:

- Si **algún jugador logra obtener los puntos necesarios**, se termina el juego y se deberá mostrar un *pop-up* o ventana adicional a cada jugador, indicando el jugador ganador y dando la posibilidad de volver a jugar otra partida.
- Si **ningún jugador tiene los puntos necesarios** para ganar, se deberá avisar a todos los jugadores que el turno del jugador actual ha acabado y el siguiente jugador deberá comenzar su turno.

5. Networking

Para lograr la comunicación entre los distintos jugadores de *DCColonos*, deberás implementar una arquitectura cliente-servidor. Para esto, tendrás que hacer uso del *stack* de protocolos de red **TCP/IP** a través del módulo [socket](#). Esta librería te proporcionará las herramientas necesarias para administrar la conexión entre dos computadores conectados a una misma red local².

Por lo tanto, deberás implementar dos programas: el programa del **servidor** y el programa del **cliente**, en donde el servidor deberá ser el **primero** en ejecutarse. Luego de empezar, el servidor quedará a la espera de que uno o más clientes se conecten a él. Es de suma importancia tener presente que mediante este modelo, la comunicación ocurre exclusivamente entre cada cliente con un único servidor, es decir, **nunca directamente entre dos clientes**.

5.1. Arquitectura cliente-servidor

En esta sección se presentarán consideraciones que deberás aplicar en la elaboración de tu tarea. **Todo lo que se indique deberás seguirlo al pie de la letra**, mientras que todo lo que no se encuentre especificado podrás implementarlo de la forma en que más te acomode, siempre y cuando cumpla con lo mínimo solicitado (siéntete libre de [preguntar](#) por este mínimo si no está especificado o si no queda completamente claro).

5.1.1. Separación funcional

El cliente y el servidor deberán estar separados, lo que implica que deben estar contenidos en directorios diferentes, uno llamado **client**, y el otro llamado **server**. Cada uno de ellos debe tener un archivo llamado

²Una red local es una infraestructura que conecta uno o más dispositivos dentro un mismo espacio físico. Puedes leer más en [este artículo en Wikipedia \(Red de área local\)](#).

`main.py` (archivo principal a ejecutar) y debe contar con todos los módulos y demás archivos necesarios para su correcta ejecución. El siguiente diagrama ilustra la estructura mencionada.

```

T03
├── client
│   ├── main.py
│   ├── parametros.json
│   ├── archivo(s)_solo_del_client.dcc
│   ├── sprites
│   ...
├── server
│   ├── main.py
│   ├── parametros.json
│   ├── archivo(s)_solo_del_server.dcc
│   ...
├── .gitignore
└── README.md
└── archivo(s)_que_no_son_del_client_ni_del_server.owo
└── ...

```

Aunque el cliente y el servidor comparten el mismo directorio padre (T03), esto es así solo para efectos de la entrega de la tarea y la ejecución de uno no podrá depender en absoluto de ningún archivo del otro, es decir, el cliente y el servidor deben estar implementados como si estuvieran en computadores diferentes. En la siguiente figura se muestra un diagrama que explica la separación esperada entre los distintos componentes de tus programas:

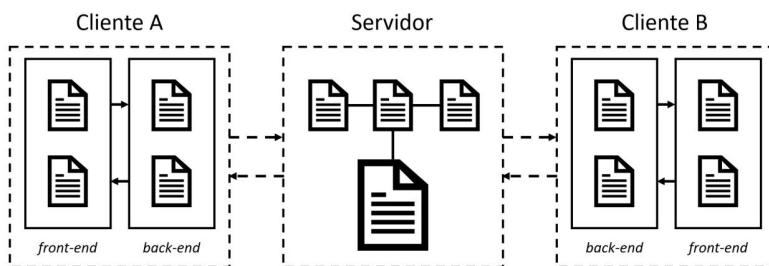


Figura 3: Separación cliente-servidor y *front-end-back-end*.

La comunicación entre el *back-end* y el *front-end* se debe realizar mediante señales, mientras que la comunicación entre el cliente y el servidor debe realizarse mediante *sockets*. Además, los clientes serán los únicos que presentarán una interfaz gráfica.

5.1.2. Conexión

El servidor abrirá un *socket* haciendo uso de los datos encontrados en el archivo `server/parametros.json`. Tal como sugiere su extensión, éste será de tipo JSON y seguirá el formato mostrado a continuación.

```
{
  "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 `client/parametros.json`.

5.1.3. Envío de información

Una vez establecida la conexión, el servidor y el cliente deberán intercambiar información en varias situaciones (descritas en más detalle en [Roles](#)). Por ejemplo, el cliente debe comunicarle al servidor cuando un jugador quiera entrar a la partida y este debe responderle al cliente con una confirmación o rechazo. Los mensajes que se envíen entre el cliente y el servidor **deberán** codificarse mediante la siguiente estructura:

- Los primeros 4 *bytes* deben indicar el largo del contenido del mensaje, los que tendrán que estar en formato *big endian*.³
- Luego, el contenido del mensaje debe ser dividido en tantos bloques de 60 *bytes* como sea necesario y, para cada uno, se deberá anteponer 4 *bytes* que indiquen el número del bloque, los que estarán codificados en *little endian*. Si el último bloque resulta ser de largo menor a 60 *bytes*, se deberá llenar con *bytes* 0 (`b'\x 00'`) hasta alcanzar ese largo.
- Si deseas transformar *strings* a *bytes* deberá utilizar UTF-8, así no tendrás problemas con las tildes ni caracteres especiales de ningún tipo.

5.1.4. Ejemplo de codificación

Supongamos que queremos enviar un mensaje entre el cliente y servidor, tal que al codificarlo como *bytes* utilizando UTF-8 resulta en una cadena de 160 *bytes*.

Según lo especificado, lo primero que se deberá enviar será un mensaje de 4 *bytes* en formato *big endian* que contiene el largo de todo el contenido que va a ser enviado (160 *bytes*).

Luego, debemos separar los 160 *bytes* del contenido en bloques de 60 *bytes*, lo cual equivale a 3 bloques, donde los primeros 2 se envían completos ($60 \times 2 = 120$) pero para el tercero solo es necesario utilizar 40 *bytes*, sobrando 20. Estos últimos *bytes* del tercer bloque deberán ser llenados con ceros (`b'\x 00'`).

Para cada uno de estos bloques, se deberá enviar el número del bloque correspondiente en un mensaje de 4 *bytes* en formato *little endian*. Posterior a ello, se debe enviar el bloque de 60 *bytes* que contiene parte del mensaje. Una vez enviado todos los bloques, el proceso de envío habrá finalizado.

En la siguiente figura se muestra gráficamente como está compuesta la codificación de ejemplo. En azul se muestran las porciones de bloques usadas por el contenido original del mensaje, mientras que en púrpura se muestra el relleno con ceros para cumplir con el espacio de 60 *bytes* por bloque.

³El *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 quieras usar para representarlo. Para más información puedes revisar este [enlace](#).

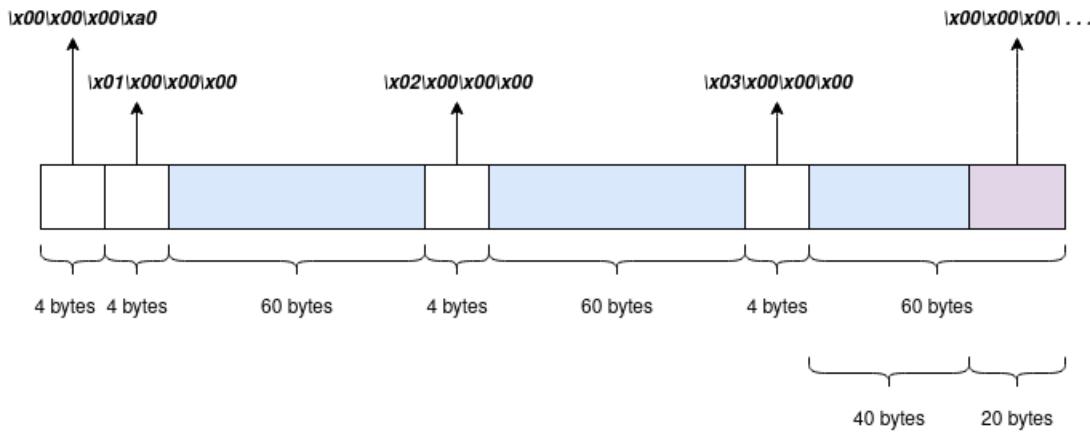


Figura 4: Ilustración del *bytearray*.

5.1.5. Logs del servidor

Como se indicó con anterioridad, el servidor **NO** cuenta con una interfaz gráfica. En su lugar, debe constantemente dejar registros de lo que ocurre en él mediante la consola. Estos mensajes se conocen como mensajes de *log*, lo que para efectos prácticos se traducen en llamar a la función `print` cada vez que ocurra un evento importante en el servidor. Específicamente, se debe imprimir un *log* al menos cada vez que:

- Un cliente se conecte al o desconecte del servidor, identificando el cliente. Para identificarlo, se le debe asignar un nombre o *id* al cliente al conectarse.
- Un cliente intente ingresar a la sala de espera.
- El programa detecte que hay suficientes clientes para comenzar la partida y da inicio a esta.
- Comience el turno de un cliente. Se deberá indicar el nombre del cliente.
- El cliente lance los dados. Se deberá indicar el nombre del cliente, la acción y el resultado del lanzamiento.
- Un cliente reciba los recursos correspondientes de algun acción. Se deberá indicar el nombre del cliente y la acción en cuestión.
- El cliente realice alguna compra. Se deberá indicar el nombre del cliente, la compra y el lugar donde ubicó la construcción.
- El cliente solicite un intercambio con otro cliente. Se deberá indicar el nombre del cliente que solicita el intercambio, el cliente al que se le está solicitando intercambio, y los elementos a intercambiar.
- Un cliente conteste una solicitud de intercambio. Se deberá indicar el nombre del cliente que solicita el intercambio, el cliente al que se le está solicitando intercambio, y los elementos a intercambiar.
- Se detecte que un nuevo cliente ha obtenido la carretera más larga. Se debe indicar el nombre del cliente, y el largo de la carretera.
- El juego detecte un ganador. Se debe indicar el nombre del cliente ganador.
- El servidor mande o reciba algún otro tipo de mensaje a algún cliente. Debe indicar a qué cliente, el tamaño del mensaje y su contenido.

El formato para hacer los *logs* queda a tu criterio mientras exponga toda la información de forma ordenada, pero se recomienda un formato similar a una tabla:

| Cliente | Evento | Detalles |
|------------|------------------|------------------|
| gatochico | Conectarse | - |
| lily416 | Lanzar dados | 4 |
| drpinto | Recibir recursos | 2 maderas |
| bimartinez | Comprar | DCChoza, Nodo: 5 |

La implementación de estos *logs* puede ser de mucha ayuda, ya que te permitirá comprobar el correcto funcionamiento del servidor y potencialmente te puede ayudar a encontrar *bugs* de funcionamiento.

5.1.6. Desconexión repentina

En caso de que algún ente se desconecte ya sea por un error o a la fuerza, tu programa debe reaccionar para resolverlo:

- Si es el **servidor** quien se desconecta, cada cliente conectado debe mostrar un mensaje explicando la situación antes de cerrar el programa.
- Si es un **cliente** quien se desconecta, se descarta su conexión. Si se desconecta mientras está en sala de espera, entonces deja de ser considerado para el juego y el cupo queda disponible. Si se desconecta mientras hay una partida en curso, este automáticamente desaparece de los participantes, de forma que el resto pueda continuar con la partida. Sin embargo, sus construcciones como chozas o carreteras deberán permanecer en el mapa sin dueño.

5.2. Roles

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

5.2.1. Servidor

- **Asignar** de forma automática un nombre de usuario a cada cliente que se conecte. La forma en que generas los nombres queda a tu criterio: puedes tener una lista de nombres predeterminados o utilizar alguna librería externa como [Faker](#), por ejemplo.
- **Procesar y validar** las acciones realizadas por todos los participantes. Por ejemplo, si se intenta colocar una choza en un lugar no permitido, el servidor deberá avisarle al usuario que la acción es inválida.
- **Distribuir los cambios** en tiempo real a los jugadores conectados correspondientes, con el fin de mantener sus interfaces actualizadas y consistentes entre sí. Por ejemplo: cuando un usuario tire los dados, el servidor se encargará de distribuir los recursos a cada cliente que tenga chozas en ese número.
- **Almacenar y actualizar** toda la información necesaria de los usuarios y sus recursos. Además, se deberá actualizar las construcciones realizadas en el mapa representado en la sección [Mapa](#).

5.2.2. Cliente

Cada cliente contará con una interfaz gráfica que le permitirá realizar acciones, y a su vez, comunicarse con el servidor. Las acciones que maneja el cliente son las siguientes:

- **Envía todas las acciones** realizadas por el usuario hacia el servidor.
- **Recibe e interpreta** las respuestas y actualizaciones que envía el servidor.

- **Actualiza** la interfaz gráfica acorde a las respuestas enviadas por el servidor.

6. Interfaz

Para mejorar la experiencia de juego, deberás implementar una interfaz gráfica que guíe al usuario a lo largo del transcurso del programa, vale decir, desde que decida ingresar a la sala de espera hasta el cierre de la aplicación.

Cabe destacar que todo lo descrito en esta sección hace referencia a una interfaz gráfica implementada **solo para el cliente**, haciendo uso de la librería [PyQt5](#).

Recuerda que los ejemplos expuestos en esta sección solo son 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:

6.1. Sala de espera

Es la primera ventana que se muestra al ejecutar el programa. Tiene por objetivo reunir a todos los jugadores en sesión mientras esperan a que la partida comience. Esta sala debe mostrar en todo momento la **lista de los jugadores conectados en la sala**.



Figura 5: Ejemplo de la Sala de espera de 3 jugadores.

En caso de que una partida ya haya comenzado y se conecte un nuevo cliente, esta ventana deberá mostrar un mensaje que avise de lo anterior, dando la opción de **cerrar la ventana** y finalizar la ejecución del programa para dicho cliente.

6.2. Sala de juego

Una vez que ingresen todos los jugadores necesarios para iniciar una partida, se deberá desplegar la **Sala de juego**. En esta ventana se desarrollará la gran parte del programa. En particular, se deben visualizar como mínimo los siguientes elementos:

- El mapa de juego con los hexágonos y construcciones actuales de los jugadores, junto a una forma de interactuar con este mapa para ubicar nuevas construcciones.
- Los recursos y puntos de victoria del propio usuario.

- La cantidad de recursos y puntos de victoria de los demás jugadores
- Dados, representando el último valor obtenido tras un lanzamiento.
- Nombre del jugador en el turno actual.
- Botón para lanzar los dados.
- Botón para terminar el turno.
- Tienda con las posibles construcciones que se pueden realizar.
- Opción para cerrar la ventana⁴.



Figura 6: Ejemplo de la Sala de juego.

6.3. Fin de partida

Esta ventana debe mostrarse una vez la partida haya finalizado. A cada jugador deberá mostrarse una *label* indicando si perdió o ganó la partida, junto a un listado de los puntajes finales de cada jugador que participó, ordenados de mayor a menor. Además, deberá contar con un botón que redirija al usuario a la [Sala de espera](#).

⁴Quedará a tu criterio implementar un botón que cierre la ventana o utilizar la opción “Cerrar” de la ventana.

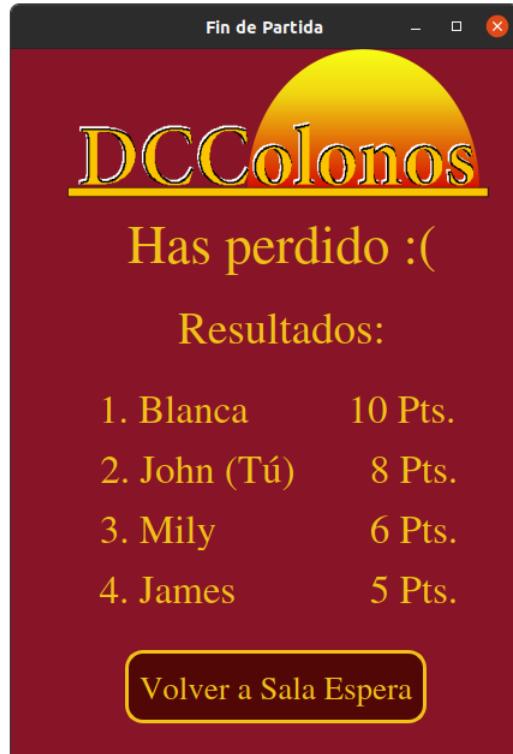


Figura 7: Ejemplo de la ventana de Fin.

7. Archivos

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

7.1. Archivos a crear

7.1.1. `parametros.json`

A lo largo del enunciado se han ido presentando 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/` contendrá 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 [Networking](#).

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

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

7.2. Archivos entregados

7.2.1. generador_de_cartas.py

Este archivo contiene la función `sacar_cartas(cantidad_cartas)`, que como dice su nombre, permite realizar la acción de sacar cartas de desarrollo de un mazo. Para llamar esta función, deberías entregarle un número entero positivo, el cual indicaría la cantidad de cartas que te retorna la función. El formato en el que retorna el mazo es una lista de la forma:

```
[("victoria", "2"), ("monopolio", "1"), ("victoria", "1"), ("monopolio", "1"), ...]
```

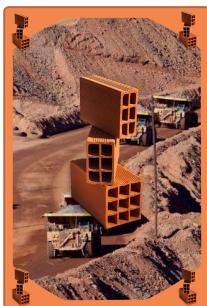
Donde cada tupla es una carta (te darás cuenta que los datos de la tupla te servirán para llamar a los *sprites* de las cartas de desarrollo). Por último, el largo de esta lista dependerá del numero que entregas como input a la función.

Es importante notar que **no debes modificar este archivo, sino solamente utilizarlo.**

7.2.2. Sprites

La carpeta `sprites` contiene varias subcarpetas con *sprites* para utilizar en tu interfaz.

- **Materias_primas:** esta carpeta contiene los sprites relacionados con las materias primas. Estarán en el siguiente formato:
 - `carta_<tipo>`, en caso de ser una carta de materia prima y la carta de reverso.



(a) carta_arcilla



(b) carta_madera



(c) carta_trigo

Figura 8: Cartas recursos *DCColonos*

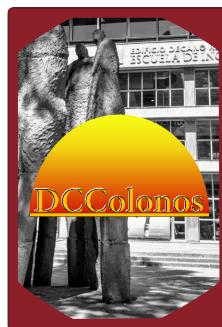


Figura 9: carta_reverso

- `hexagono_<tipo>`, en caso de ser un hexágono y su recurso asociado.



(a) hexagono_arcilla



(b) hexagono_madera



(c) carta_trigo

Figura 10: Hexágonos recursos *DCColonos*

- `ficha_numero`, para la ficha en donde se coloca el número de ficha de cada hexágono.

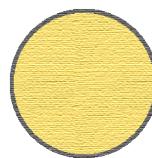
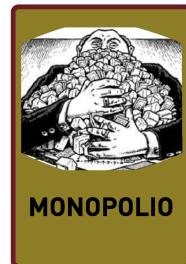


Figura 11: `ficha_numero`

- **Desarrollo:** esta carpeta contiene los dos tipos de cartas de desarrollo del programa y la carta de reverso.



(a) Punto de victoria



(b) Monopolio

Figura 12: Cartas desarrollo *DCColonos*

- **Construcciones:** esta carpeta contiene los *sprites* de las construcciones del juego.

- `camino_<color>_<angulo>`, en caso de ser un camino, su color y ángulo (el uso del ángulo es opcional, si lo deseas puedes rotar los *sprites*).
- `<tipo>_<color>`, en caso de ser una choza o ciudad, y su color asignado.

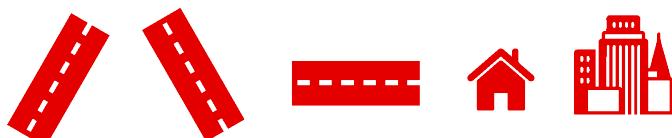


Figura 13: Ejemplo fichas rojas

- **Datos:** esta carpeta contiene los *sprites* de los dados del juego.
 - *dado_<numero>*, en caso de ser un dado acompañado de su número correspondiente.

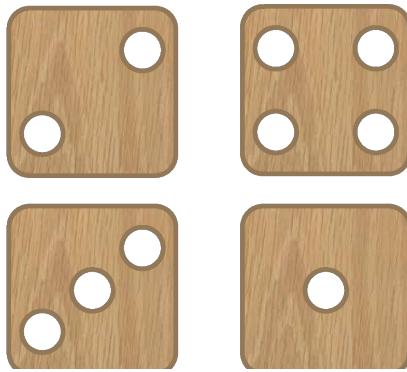


Figura 14: Ejemplo dados

7.3. grafo.json

Se te entregará un archivo en formato json que contendrá las **dimensiones** (cantidad de filas y columnas) que tiene el mapa, la **lista de adyacencia** del grafo, y finalmente, los nodos que componen cada hexágono. Para la lista de adyacencia, cada nodo tendrá su id correspondiente y una lista de los nodos adyacentes a él, mientras que para los hexágonos, cada uno tendrá su id correspondiente y una lista de los nodos que lo conforman.

A continuación se muestra un ejemplo del formato en que viene el contenido del archivo:

```
{
  "dimensiones_mapa" : [5,2],
  "nodos" : {
    "0" : ["1", "4"],
    "1" : ["0", "5"],
    "2" : ["3", "6"],
    "3" : ["2", "7"],
    "4" : ["0", "9"],
    "5" : ["1", "6", "10"],
    ...
  },
  "hexagonos" : {
    "0" : ["0", "1", "4", "5", "9", "10"],
    "1" : ["2", "3", "6", "7", "11", "12"],
    ...
  }
}
```

7.4. generador_grilla.py

Para facilitar la visualización del mapa en tu interfaz, puedes hacer uso de la clase **GeneradorGrilla(largo_arista)** contenida en este archivo.

Esta clase recibe un **int** que representará el **largo en píxeles** que tendrá cada arista de la grilla. Una vez instanciada, puedes llamar a su método **generar_grilla(dimensiones_mapa, traslacion_x,**

`traslacion_y`), que recibe dos `int` que trasladarán la grilla resultante, de forma que puedes situarla en la posición que deseas de tu interfaz. Esta función retornará un `dict` de la forma:

```
{  
    "0": (145, 100),  
    "1": (235, 100),  
    "2": (415, 100),  
    ...  
}
```

Donde las llaves serán *strings* que representan los **id** de cada nodos y sus valores correspondientes serán tuplas con la posición en la interfaz donde deberá ir dicho nodo.

Esta función se entrega como ayuda para situar los nodos en la interfaz, sin embargo puedes crear tu propia función o implementar la visualización del mapa de otra manera si así lo deseas.

8. *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 4.0⁵**.
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.

8.1. Ladrón (4 décimas)

Luego de jugar innumerables partidas, te das cuenta que siempre existe un jugador que planea monopolizar un recurso, sin dejar que otros puedan intercambiar materias primas con él. Es por esto que decides inventar una nueva regla, que te permitirá castigarle por su codicia.

Cada vez que un jugador lance los dados al comienzo del turno y salga el número **7**, deberá seleccionar uno de los hexágonos para **situar** al ladrón sobre este. Luego, deberá elegir a alguno de los jugadores que posean construcciones en los nodos del mismo hexágono para robarle una materia prima. El recurso robado será elegido de manera aleatoria.

En caso de que al lanzar los dados en un turno posterior salga el número de ficha de un hexágono ocupado por un ladrón, ninguno de los jugadores que posean construcciones en sus nodos recibirán el recurso de ese hexágono. Finalmente, al **ladrón** le gusta ir cambiando de lugar, por lo que cuando se deba bloquear un nuevo hexágono con un ladrón, habrá que asegurar que la nueva ubicación sea distinta a la anterior.

La forma en que decides implementar la elección del robo puede ser mediante un *pop-up* o una ventana adicional. Además, quedará a tu criterio la forma de representar el hexágono bloqueado, siempre y cuando sea reconocible y no oculte de forma completa su número de ficha. Puedes asumir que al comienzo de la partida el **ladrón** no se ubicará en ningún hexágono.

8.2. Ciudades (2 décimas)

En una de tus partidas, te das cuenta que los demás jugadores te han dejado acorralado, sin darte la posibilidad de expandir tus territorios y ganar puntos de victoria. Es por eso que decides inventar una

⁵Esta nota es sin considerar posibles descuentos.

nueva regla, la cual consiste en construir ciudades.

El proceso de construcción de una ciudad es similar al de las demás construcciones, con la diferencia en que un jugador solo podrá ubicar una ciudad en el mismo nodo que el de una de sus chozas previamente construida. Para construirla, se necesitarán **CANTIDAD_ARCILLA_CIUDAD** arcilla, **CANTIDAD_MADERA_CIUDAD** madera, **CANTIDAD_TRIGO_CIUDAD** trigo. Las ciudades aportan los siguientes beneficios:

- Cada ciudad aportará **+2** puntos de victoria. Debes tener en cuenta que la choza previamente construida desaparece, y por ende ya no aportará con su respectivo puntaje.
- Cada vez que un jugador lance los dados y arroje un número en la ubicación de una ciudad, recibirá el **doble** de recursos de lo que haría con una choza.

8.3. *Chat* (4 décimas)

La comunicación es esencial para cualquier juego, por lo que decides agregarle un *chat* a *DCColonos* para poder hablar libremente con el resto de los jugadores. El *chat* debe mostrarse tanto en la Sala de Espera como en la Sala de Juego, y debe incluir como participantes solo a quienes estén en la sala de espera o jugando una partida.

El diseño del *chat* queda a tu libre criterio (siendo, por ejemplo, una ventana aparte que esté siempre abierta cuando se esté en las salas de espera y juego, o incorporado dentro de las mismas ventanas anteriores), siempre y cuando este sea intuitivo de usar. Cada mensaje de *chat* debe seguir el siguiente formato:

Usuario : Mensaje

Debes asegurarte de que los mensajes se muestren siguiendo el orden cronológico en que fueron enviados.

9. Avance de tarea

Para esta tarea, el avance consistirá en implementar el protocolo de envío de información, junto con la interacción inicial de *DCColonos*, tanto en el cliente como en el servidor.

En particular, se pide implementar un servidor que sea capaz de conectarse con **múltiples** clientes, asignándoles un **nombre único** a cada uno de ellos y enviando de vuelta a cada cliente su nombre asignado. Además, deberás implementar el inicio de los **logs**, por lo que tendrás que imprimir qué cliente se conectó con el servidor siguiendo el formato indicado anteriormente.

Por último, deberás implementar el protocolo de **envío de mensajes**. Para esto, tendrás que aplicar los **mismos** pasos realizados en [Envío de información](#) tanto desde el cliente hacia el servidor, como para el servidor hacia el cliente.

Para este avance **no** será necesario realizar la ventana de la [Sala de espera](#) inicializada desde el cliente, ni una actualización en vivo de los usuarios dentro de ella. Quedará a tu criterio si decides implementarlo o no para este avance.

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.

10. `.gitignore`

Para esta tarea **deberás utilizar un `.gitignore`** para ignorar los archivos indicados, este deberá estar dentro de tu carpeta `Tareas/T03/`. Puedes encontrar un ejemplo de `.gitignore` en el siguiente [link](#).

En esta ocasión deberás ignorar los siguientes archivos:

- Enunciado.
- Carpeta de *sprites*.
- `generador_de_cartas.py`
- `grilla.py`
- `grafo.json`

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.

11. 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 llenar el *form* será de 24 horas. En caso de que no lo contestes en dicho plazo, se procederá a corregir el *commit* recolectado.

12. 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.

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

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, ero implicará que **no obtengas puntaje en los ítems relacionados**. Si lo llegases a implementar de esta forma, recuerda indicarlo en tu `README.md`.

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 [correo](mailto:correo(bienestar.iic2233@ing.puc.cl)) (`bienestar.iic2233@ing.puc.cl`).

13. 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 o .ui.
- 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 48 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 24 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).