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

IIC2233 Programación Avanzada (II/2017)

Tarea 2

1. Objetivos

- Ser capaz de crear estructuras de datos propias.
- Aplicar conceptos y nociones de estructuras de datos para el correcto modelamiento de un problema.
- Crear y usar algoritmos eficientes para recorrer las estructuras creadas.
- Comprender el uso y aplicar el archivo `.gitignore`.
- Familiarizarse con `PyQt5`.

2. Introducción

¡El mercado de monedas ha caído! La gente pierde su dinero y ante tal desastre comienzan a destruirlo todo. Debido a esto, se produce un caos en DCCesteros, región donde antiguamente la reina Barrios y Lucas ~~el doglover~~ dominaban los Siete Reinos Informáticos junto a sus fieles mascotas: Copi-Copi, Elemento, Adjetivo, y Tepo-tepo . Distintas facciones tratarán de tomar el poder del trono, pero para evitar numerosas muertes, se te pide a ti, ~~plebeyo~~ noble consejero, programar un juego para que la reina y el rey puedan competir de forma justa.

3. Prograsonne

El juego consiste en dos jugadores que ingresan piezas a un tablero de forma alternada. Estas piezas poseen partes de ciudades, caminos, ríos y pasto. El objetivo es obtener un mapa coherente. En otras palabras, adueñarse de ciudades al juntar distintas piezas para crearlas con tamaño variable y caminos con los cuales se unen. Además, existe un sistema de puntaje que sirve para determinar el ganador según las piezas que ingresó y las ciudades de las que se adueñó. Por otro lado, los jugadores pueden agregar piezas al tablero, deshacer la jugada y volver a un estado anterior específico la partida. Finalmente, existen tres formas de terminar el juego que se explicarán más adelante. Una de formas es que uno de los jugadores consiga la *snitch dorada*[1]. Esto determina automáticamente el ganador.

3.1. Entidades - Tipos de segmentos

Cada pieza del tablero puede contener distintos tipos de entidades o segmentos. Éstos sirven para mejorar el estado de las ciudades ya sea por llevar agua o establecer una ruta comercial mediante un camino. Si el borde de una pieza contiene un tipo de segmento, entonces sólo puede unirse por ese borde con otra pieza que contenga **ese mismo** tipo de segmento.

[1] Al igual que en la ~~famosa~~ saga de *Harry Potter*.

- **Pasto**: Este tipo de segmento no tiene una función específica.
- **Ciudad**: Esta entidad se puede formar con una o más piezas. Una ciudad se dice **completa** o **cerrada** si las piezas exteriores que la conforman están rodeadas con murallas.
- **Camino**: Los caminos son uniones que sirven para poder conectar ciudades entre sí, bonificando económicamente a los pueblos que se unen.
- **Río**: Un río es un tipo de unión que sirve para llevar agua a las ciudades, lo que hace más próspera a estas últimas. A continuación, se muestra la manera correcta de unir estos segmentos.

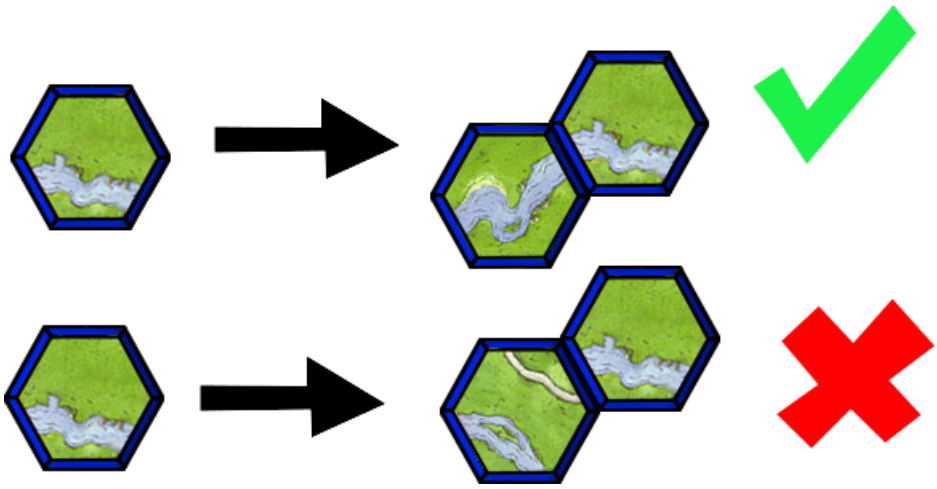


Figura 1: Ejemplo unión ríos

3.2. Tipos de piezas

Cada pieza tiene seis bordes enumerados del 1 al 6, como se indica en la Figura 2.

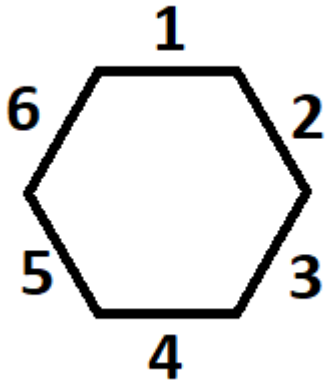


Figura 2: Enumeración de las piezas

Cada uno de éstos tiene un **tipo de borde** específico. Existen 4 diferentes tipos de bordes y cada uno tiene una letra que lo representa. Los tipos de bordes y sus respectivas letras son:

- C: City
- G: Grass
- R: River
- P: Path

Cada pieza tiene un **tipo de pieza**. Éste está definido por una secuencia ordenada de largo 6 que indica cuál es el tipo de cada uno de sus bordes (según su enumeración).

A continuación se muestra un ejemplo de dos piezas distintas:

El juego tendrá 24 tipos de piezas distintas que están definidos en el archivo `pieces.csv`. Las imágenes que representarán a estas piezas se encuentran en las carpetas `Tareas/T02/gui/assets/red/` y `Tareas/T02/gui/assets/blue/` para los jugadores 1 y 2, respectivamente.

3.3. Reglas [12 %]

El juego tendrá un tablero de dimensiones 8×8 como el siguiente:

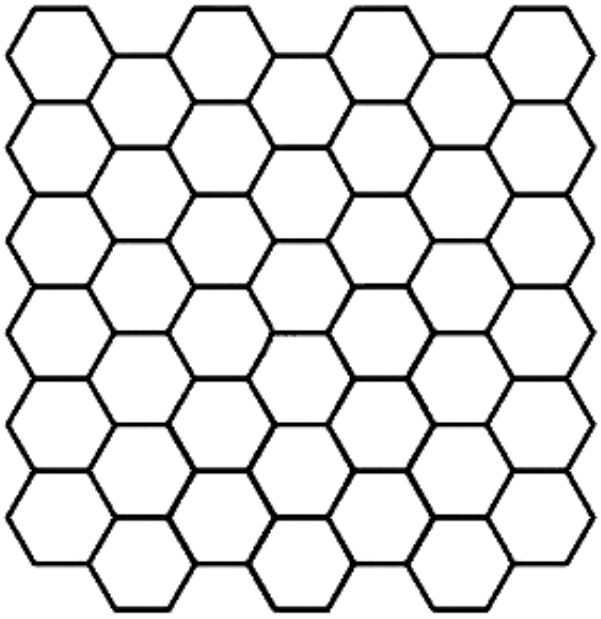


Figura 3: Tablero Prograsonne

1. Al inicio del juego se colocará 1 pieza en el mapa de forma aleatoria (en cualquier posición) y ésta **no** puede ser retirada de la partida.
2. En cada turno los jugadores pueden elegir entre colocar una pieza nueva o deshacer la última jugada que realizaron: Al inicio del turno, el programa deberá entregarle una pieza cualquiera para que el jugador decida:
   - Deshacer la última jugada realizada por él mismo. Si se elige esta opción, la pieza debe desaparecer del tablero.
   - En otro caso, tener la opción de rotar la pieza y colocarla en el tablero. La utilidad de rotar una pieza consiste en poder calzarla con algún lado y formar uniones con entidades adyacentes.
3. El turno finaliza cuando la acción escogida por el jugador haya terminado.
4. Al momento de colocar una pieza, si un lado está en contacto con otra pieza, ambas deben poseer el mismo tipo de segmento. En caso contrario, no se debe permitir poner la pieza en esa orientación.
5. Sólo se pueden poner piezas que estén en contacto con otras. Es decir, no se puede insertar una pieza en un lugar que sin vecinos.
6. El juego termina cuando un jugador no puede poner su pieza, cuando no quedan más, o bien, cuando se logra completar la *snitch dorada*. Al terminar el juego, se deberá calcular el puntaje acumulado de cada jugador. En el caso de no completar la *snitch dorada*, el jugador que posea mayor puntaje será el ganador.
7. La ~~malvada~~ hechicera Chau dijo: "¿Para qué hacer la tarea fácil?", e hizo su famoso hechizo **Plot twist**. Éste consiste en que el dueño de una ciudad es el jugador que la complete, es decir, **la ciudad le pertenece al jugador que ponga la última pieza**. Este cambio **debe** verse reflejado en la interfaz.
8. En el caso de los caminos y ríos, ocurre algo parecido a lo descrito en el punto anterior. El jugador que ponga la última pieza para unir tanto un camino como un río, se convierte en su dueño.

3.4. Ctrl-Z [12 %]

El programa debe ser capaz de guardar **todos** los movimientos de cada jugador. Se debe dar la opción de retroceder **una vez** por turno para deshacer una jugada anterior propia. Al deshacer cada jugada, la ficha correspondiente desaparece y se acaba el turno del jugador actual.

3.5. Historial [18 %]

El programa debe ser capaz de **guardar el estado actual** del tablero completo a modo de historial utilizando los métodos que se explican más adelante. Luego, en cada turno, el jugador actual debe tener la opción guardar el estado del tablero, o bien, volver a estados **anteriores** —es decir, que hayan sido guardados anteriormente— cuantas veces quiera.

Para guardar el estado actual, en la interfaz hay que hacer *click* en el botón `Guardar` y el estado se representa con un círculo enumerado ubicado abajo a la derecha de la ventana. Luego, para volver a los estados anteriores, basta con apretar alguno de los círculos enumerados. El manejo de cómo guardar la información queda a su criterio.

Puesto de otra forma gráfica, el objetivo de esta sección es que sean capaces de guardar estados, más bien, un historial tal como lo hacen distintas plataformas como *GitHub* y *Overleaf*. En las siguientes imágenes se explicará de forma gráfica cómo funcionan los historiales en *GitHub*:

En la **figura 4** se encuentra "guardado" un estado[2].

Figura 4: Un estado sin ramificaciones

En la **figura 5** se encuentran guardadas varias versiones. Como pueden ver, cada una de estas apunta a su nodo padre.

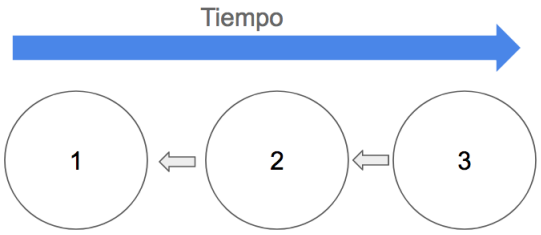


Figura 5: Tres estados sin ramificaciones

En la **figura 6** veremos lo que pasa cuando hacemos *click* en un estado anterior.

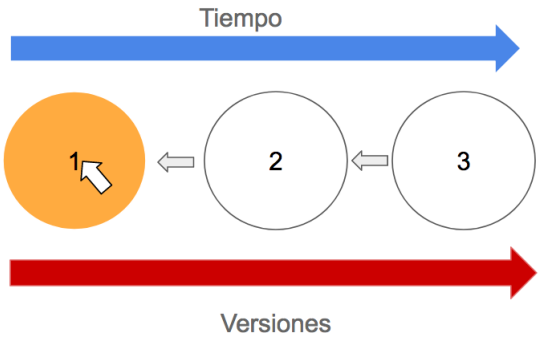


Figura 6: *Click* en el estado 1

Al hacer *click* en el estado 1, se forma una nueva *rama* tal como se ve en la **figura 7** y todos los estados hacia la **derecha** del estado 1 se eliminan.

[2]Los estados son los círculos enumerados.

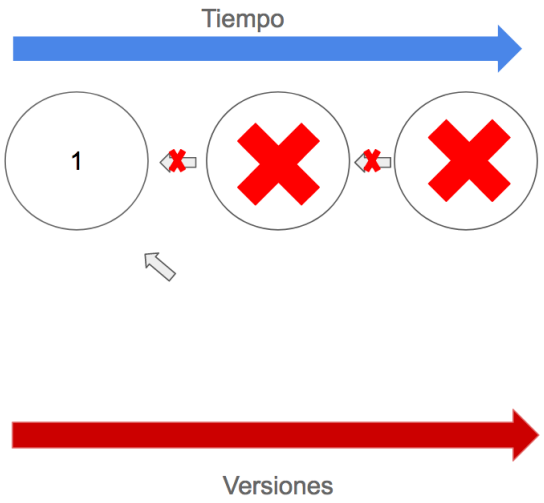


Figura 7: Nueva rama resultante

Finalmente, en la **figura 8**, se muestra cómo se agregan estados luego de la ramificación.

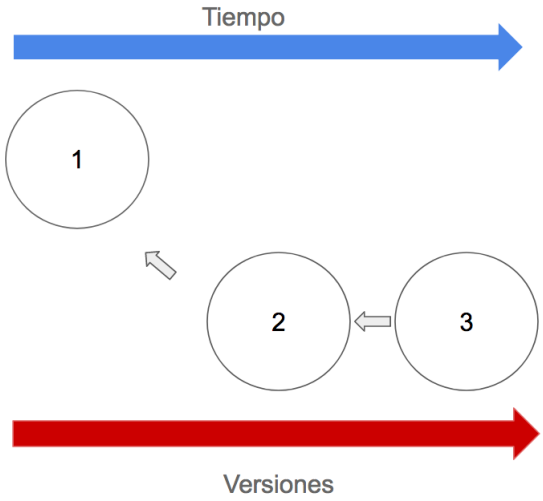


Figura 8: Tres estados con 1 ramificación

Ahora, les explicaremos cómo se debería ver en la interfaz:

En las **figuras 9 y 10**, se simula el *click* del jugador azul en el botón `Guardar` y la aparición de la representación del estado actual mediante un círculo enumerado.

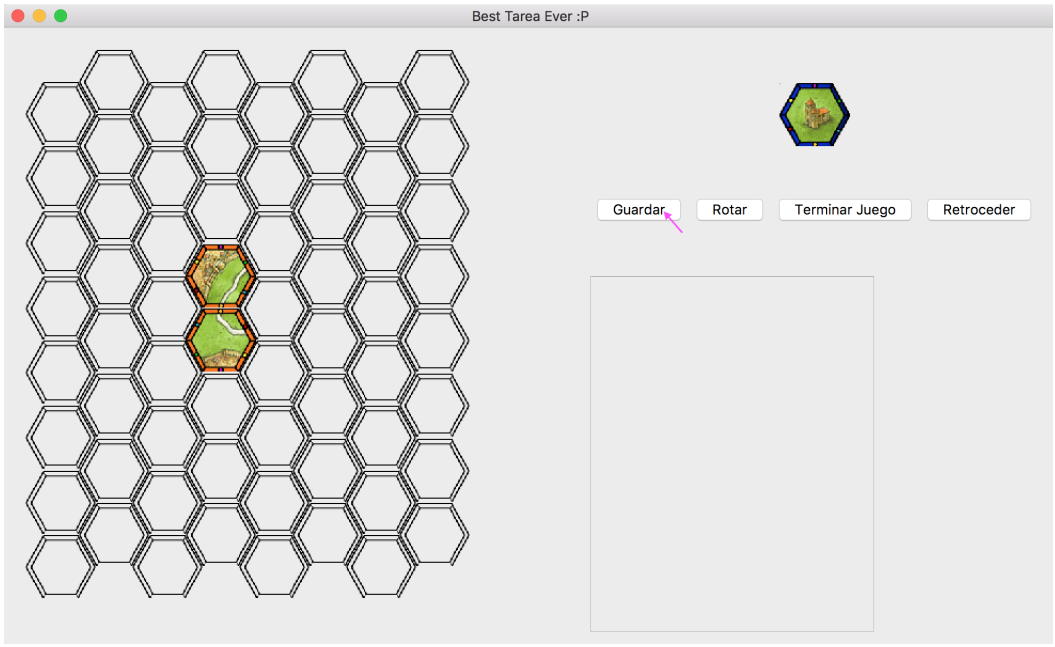


Figura 9: Imagen del jugador azul haciendo *click* en el botón `Guardar`

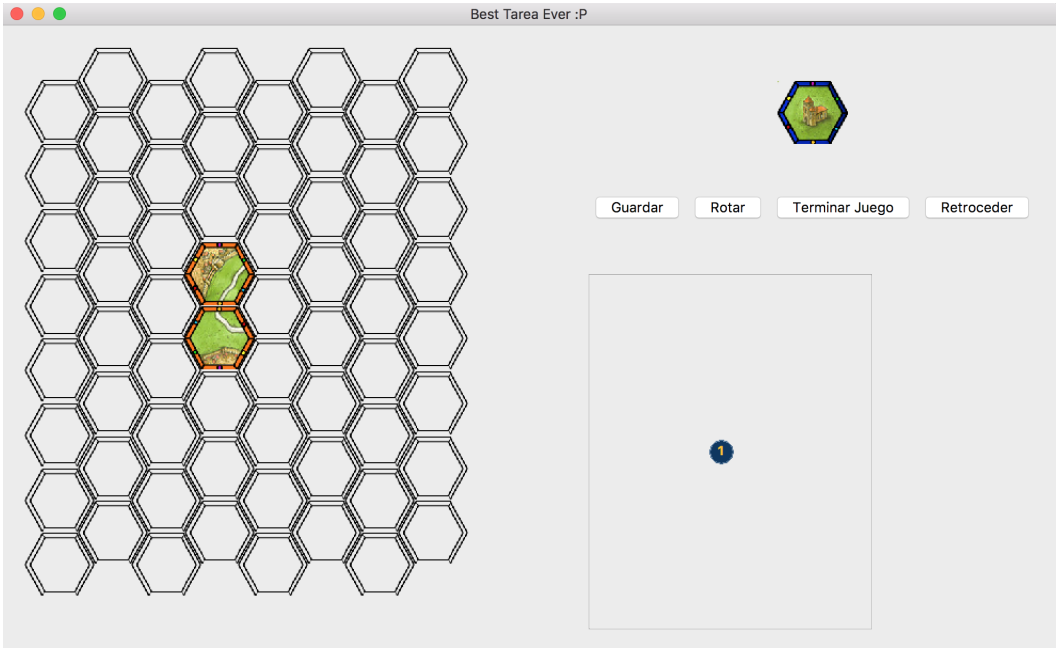


Figura 10: Representación del estado actual del tablero en el círculo #1

Luego, en la **figura 11** se muestra un avance en el tablero y que el jugador naranjo/rojo guardó el estado del tablero actual.

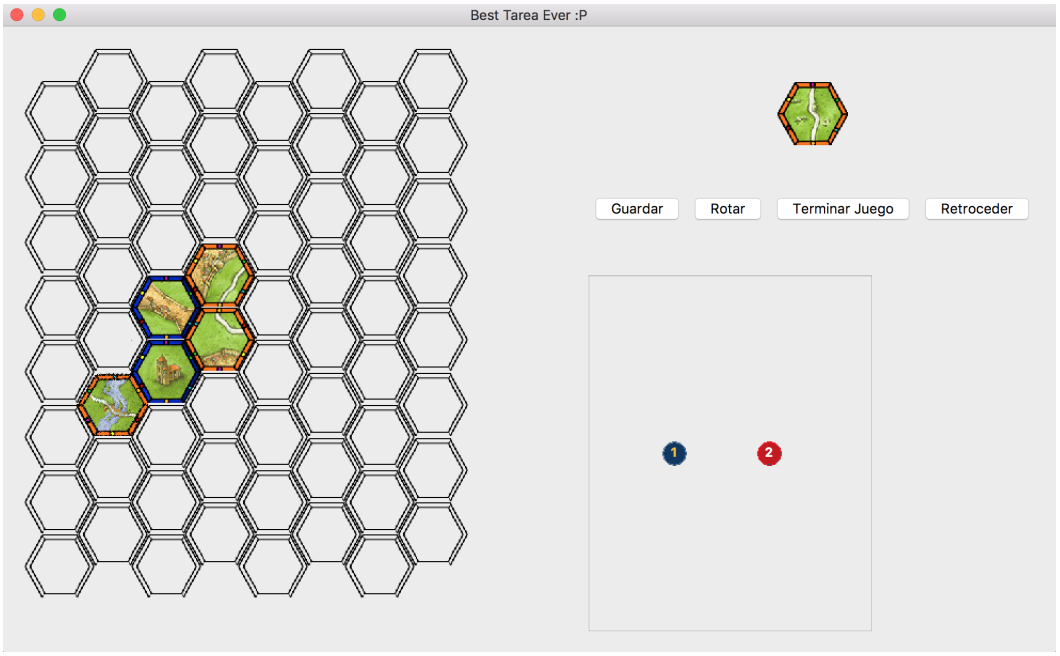


Figura 11: Aparición de otro círculo: #2

Finalmente, en la **figuras 12** se muestra como el jugador naranjo/rojo hace *click* sobre uno de los estados. En la **figura 13**, se muestra cómo queda el tablero al volver a como estaba en el estado #1.

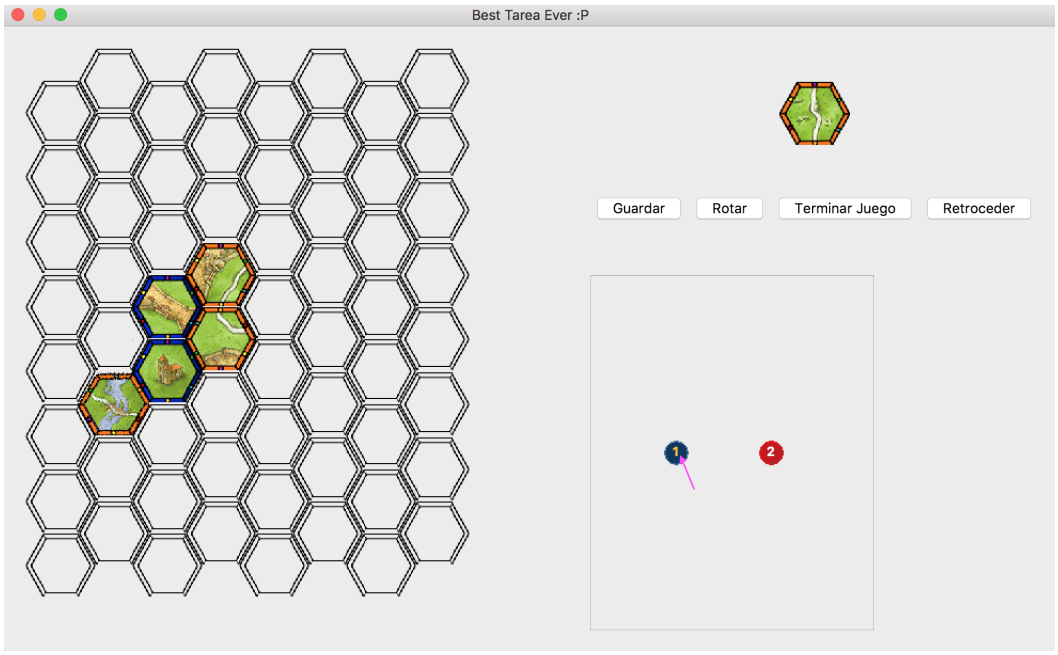


Figura 12: Imagen del jugador naranjo/rojo haciendo *click* sobre uno de los estados.

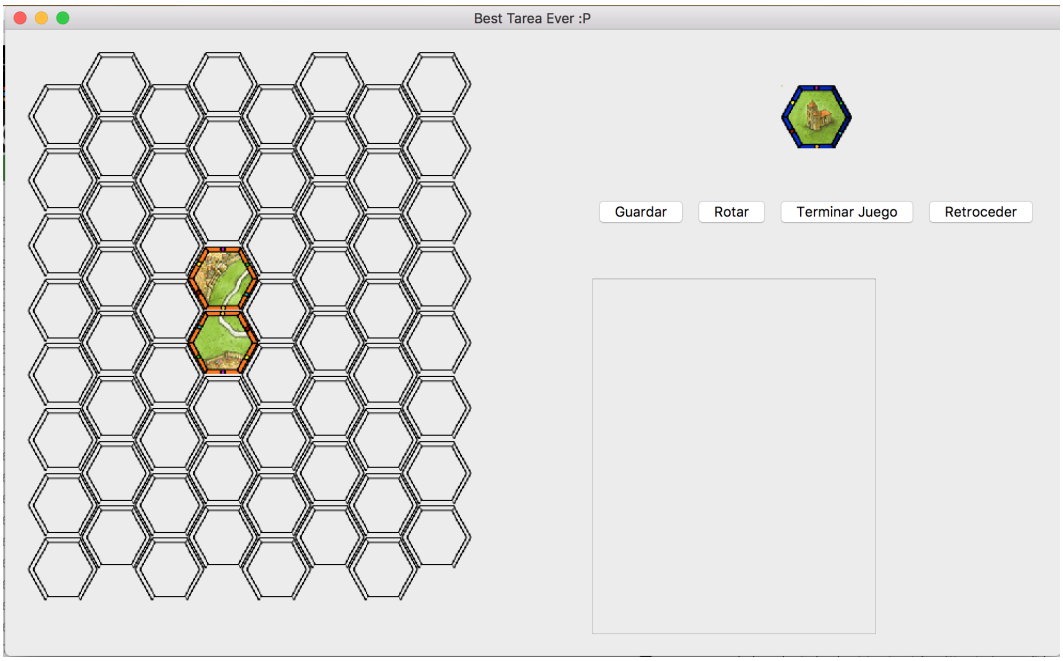


Figura 13: Como estaba el tablero en el estado #1

3.6. Puntaje [27 %]

El cálculo del puntaje se realizará si es que no se pueden realizar más jugadas, ya sea porque el tablero no tiene espacios disponibles o porque no existen jugadas válidas para realizar. También se terminará si se presiona el botón terminar juego. Cada jugador sumará puntos según las siguientes reglas:

- **Ciudad completada**: Una ciudad se considera completada si está totalmente rodeada por murallas. Se entregarán 30 puntos por ficha involucrada en la creación de la ciudad y se otorgarán 40 puntos adicionales sólo por completar la ciudad.
- **Ciudad incompleta**: Este tipo de ciudad no tiene las defensas como para proteger a los mercaderes. Cada pieza que no este rodeada por murallas entregará 10 puntos.
- **Caminos**: Los caminos tendrán dos formas de puntajes: 20 puntos base si el camino conecta una ciudad con el borde de la grilla y 50 puntos base si el camino conecta dos ciudades que no estén completas (pueden estar incompletas ambas o una sola). Sumado a esto existen bonificaciones según el largo del camino, en el primer caso, cuando el camino conecta la ciudad con el borde de la grilla, se entregarán 10 puntos por cada cada ficha que componga el camino. En cambio, en el otro caso, se entregarán 30 puntos por cada ficha involucrada; En el caso que los caminos conecten más de dos ciudades, se entregarán 40*n puntos adicionales, donde n es el número de ciudades relacionadas.
- **Ríos**: Los ríos también tendrán dos formas de puntajes: 25 puntos si el río conecta una ciudad con el borde de la grilla y 55 puntos si el río conecta dos ciudades (una de ellas **no completas**). En el primer caso, cuando el río conecta la ciudad con el borde de la grilla, se entregarán 15 puntos por cada cada ficha que componga el río. En cambio, en el otro caso, se entregarán 35 puntos por cada ficha involucrada; En el caso que los ríos conecten más de dos ciudades (con al menos una no completa), se entregarán 45*n puntos adicionales, donde n es el número de ciudades relacionadas.

3.7. Hints [15 %]

Para hacer más entretenido el juego, existe la opción de entregar *hints*. Éste consiste en mostrarle en la interfaz, representado por un hexágono con bordes amarillos, al jugador del turno actual la mejor jugada posible de tal forma de otorgarle el mayor puntaje. Algunos ejemplos posibles: terminar una ciudad, colocar camino/río, cerrar un camino/río.

3.8. Snitch Dorada

Existe una condición que permite a los jugadores ganar instantáneamente sin tener que cumplir necesariamente las condiciones mencionadas anteriormente. Esto consiste en tener **dos ciudades cerradas conectadas por medio de un río**. El río debe estar compuesto de **al menos tres** piezas. El jugador que ingrese la última pieza necesaria para completar esta unión, independiente del dueño de las otras piezas, ganará la partida.

Figura 14: Imagen de la *snitch dorada*

3.9. GUI

Para desarrollar esta tarea interactuarás con una interfaz grafica hecha con elementos de `PyQt5`. Deberás **sobreescribir** los métodos del archivo `demo.py` para que tu programa haga lo adecuado cuando se opriman los botones de la *GUI* o se *clickee* en algún sitio. A continuación están los métodos que deberás sobreescribir:

3.9.1. Métodos de GameInterfaz

Los siguientes métodos corresponden a la clase GameInterfaz de la cual **debes heredar**, tal como se muestra en ejemplo del archivo `demo.py`. Estos métodos son llamados cuando se interactúa con la interfaz, por lo que **debes sobreescribirlos** para obtener su comportamiento deseado. Los métodos son:

- `rotar_pieza(self, sentido)`: Cuando se presiona el botón **ROTAR**, se ejecuta rotar_pieza. Este método recibe el sentido de rotación 'Izquerda' o 'Derecha' y no retorna nada.
- `colocar_pieza(self, i, j)`: cada vez que se hace *click* en una posición del tablero, este método se ejecuta recibiendo las coordenadas (i, j) de la pieza o espacio donde se hizo *click*. Retorna `True` si es que la pieza puede ir en esa posición y False en caso contrario. Si retorna `True`, cambia el turno del jugador.
- `terminar_juego(self)`: Se ejecuta cuando se presiona el botón **TERMINAR JUEGO**. Se encarga de calcular el puntaje final por jugador e imprime en consola al ganador junto al puntaje de cada jugador.
- `retroceder(self)`: Se ejecuta cuando se presiona el botón **RETROCEDER**, el cual debe retornar la posición (i,j) de la pieza a eliminar. En caso de no poder eliminar una pieza debe retornar (-1,-1).

- `hint_asked(self)`: Se ejecuta cuando se presiona la tecla *h*, debe agregar un hint de pieza a jugar al usuario.
- `click_number(self, number)`: Se ejecuta cuando clickeas un estado de la sección de estados, recibe el numero de la pieza que se clickeo (el número que visualmente se ve en ella).
- `guardar_juego(self)`: Se ejecuta cuando se presiona el botón **GUARDAR**, debes manejar en ella guardar el estado actual del tablero.
- `agregar_puntaje(self, num_jugador, cantidad)`: Se ejecuta cuando el juego finaliza y se calcula el puntaje de cada jugador.

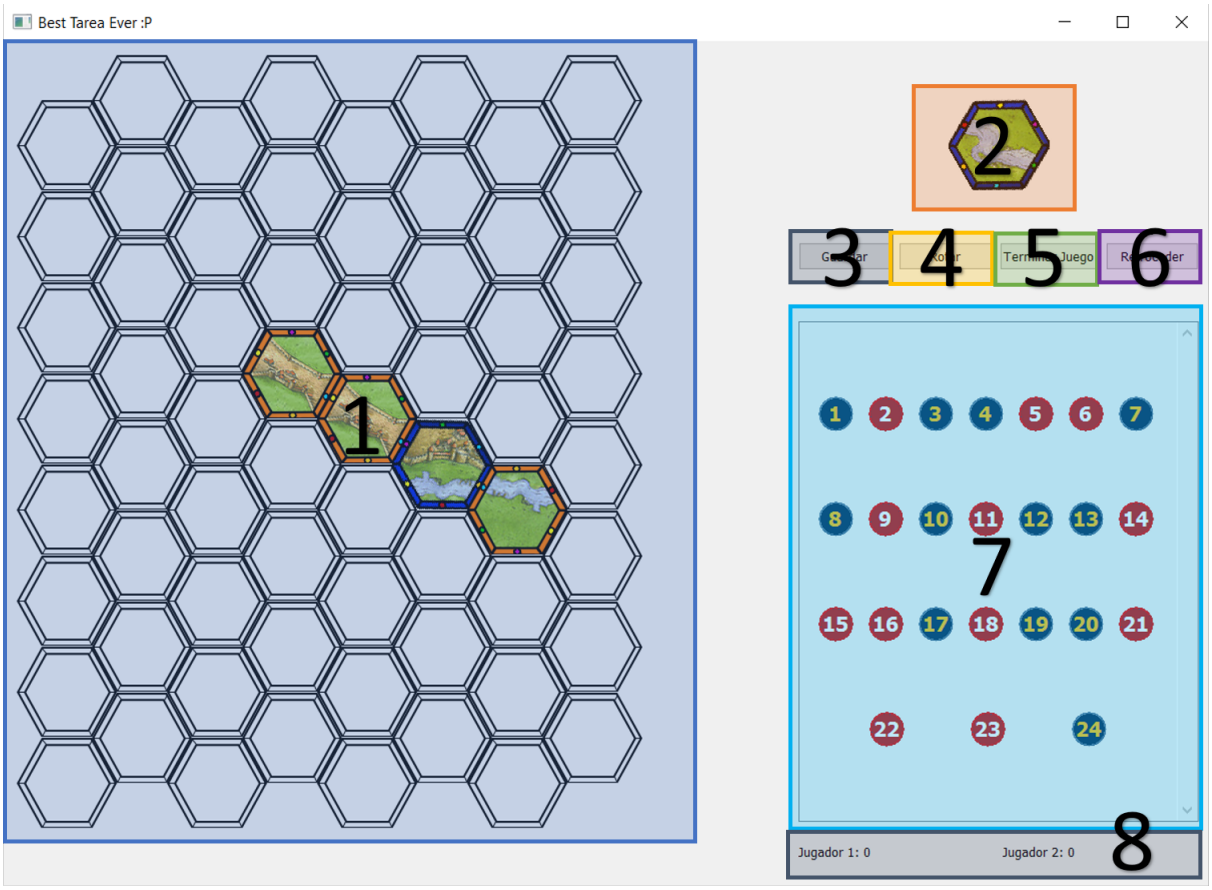
3.9.2. Métodos de la GUI

Para lograr que la interfaz muestre lo que tu quieres, tienes las siguientes funciones que interactúan con la GUI. Para usarlas, basta con escribir `gui.funcion(x, y, z)`.

- `nueva_pieza(color, tipo_pieza)`: Este método reemplaza la pieza actual que se está jugando, recibe el color del jugador ("red" o "blue") y tipo_pieza que es el tipo de pieza que quieres poner. Éste corresponde a un string donde cada letra representa un lado. Por ejemplo: CCCCCC representa una pieza con ciudad por todos los bordes.
- `add_piece(i, j)`: Este método inserta la pieza actual en la posición (i, j) del tablero. En caso de que ya esté ocupado arroja una *Excepción*.
- `pop_piece(i, j)`: Este método elimina del tablero la pieza que está en la posición (i, j).
- `pop_number()`: Este método elimina la elimina el último de los estados guardados.
- `add_number(number, color)`: Este método inserta un estado nuevo guardado en la sección de los estados guardados con el numero que recibe y el color del jugador (puede ser "red" o "blue")
- `add_hint(i,j)`: Este método destaca la pieza vacía en la posición (i, j) del tablero.
- `set_points(player_number, points)`: Este método agrega recibe el número del jugador y los puntos que tiene. Se actualiza el puntaje en la interfaz.

3.9.3. Ejemplo

A continuación se muestra un ejemplo de la interfaz gráfica:



1. Tablero: En este lugar se ubican las piezas del juego. Cuando haces *click* con el mouse en este lugar se llama al método colocar_pieza recibiendo las coordenadas en las que se clickeo. El método `gui.add_piece` agrega las piezas al tablero (tal como se ve en el ejemplo).

2. Pieza a jugar: En este lugar puedes observar la siguiente pieza que se jugará cuando se haga *click* en el tablero, el método `gui.nueva_pieza` actualiza esta imagen.

3. Botón Guardar: Al hacer *click* en este botón se llama al método `guardar_juego`.

4. Botón Rotar: Al hacer *click* en este botón se llama al método `rotar_pieza`.

5. Botón Terminar Juego: Al hacer *click* en este botón se llama al método `terminar_juego`.

6. Botón Retroceder: Al hacer *click* en este botón se llama al método `retroceder`.

7. Estados: En este lugar irán apareciendo los estados guardados. Debes preocuparte de que el estado que se guarde tenga el color del jugador que lo guardó. Para agregar estados, puedes utilizar el método `add_number`. Luego, cuando al hacer *click* sobre alguno de los estados, se llama al método `click_number`. Esto último será de utilidad para eliminarlos de la interfaz.

8. Puntajes: En este lugar se muestran los puntajes de ambos jugadores. Al finalizar el juego, se deberá calcular el puntaje de ambos jugadores y agregarlos en la interfaz llamando al método `agregar_puntaje`.

4. Archivos [1 %]

Les entregaremos un archivo generador `csv_generator.py` que creará el archivo `pieces.csv`. Éste tiene la información del nombre de las piezas y la cantidad. En la siguiente tabla se muestra el formato del archivo donde x representa un número (la cantidad de piezas no necesariamente es la misma):

Cuadro 1: `pieces.csv`

Pieza:string	**Cantidad:int**
CCCCCC	x
CCCGCC	x
CCCGCC	x
CCCPCC	x
CGGCGG	x
CGGGCC	x
CGGPPG	x
CPGPCC	x
CRCCRC	x
GCCRPG	x
GGCGGC	x
GGGGGC	x
GGGGGG	x
GGGGGR	x
GGGPPG	x
GGPGPP	x
GGRGGG	x
GGRGRG	x
GPRGPR	x
GRGGRG	x
GRPGRG	x
PGGPGG	x
PPGPPG	x
PPGRRG	x

5. .gitignore

Para no saturar los repositorios de `GitHub`, **deberás** agregar en tu carpeta `Tareas/T02/` un archivo llamado `.gititgnore` de tal forma de **no** subir los archivos de la carpeta `gui`. La siguiente página les será muy útil para crear el contenido del archivo: `www.gitignore.io`.

6. Master [15 %]

Para demostrar tu capacidad como programador, decides demostrarle al mundo que eres capaz de unificar a las ciudades **SIN USAR LAS ESTRUCTURAS DADAS POR PYTHON** *(listas, diccionarios, sets, etc..)* Debes detallar en el README las especificaciones de cómo implementaste cada una de las estructuras utilizadas.

Tus implementaciones deben replicar la funcionalidad básica de sus contrapartes, esto es, aquellas que son iterables e iteradores (`__iter__`, `__next__`) deben serlo. Si son estructuras ordenadas deben poder ordenarse usando el método `sort`. Además deben responder bien a la función `len` (`__len__`), `append` y a la

indexación (`__getitem__`, `__setitem__`, `__delitem__`) . Para esto, deben hacer uso de los métodos mágicos correspondientes.

7. Últimas Consideraciones

- Los puntos presentes en los bordes de las piezas no representan nada, son sólo de decoración.
- Al momento de colocar una pieza puede que visualmente no coincida los lados (un camino esté mas arriba que el otro o una ciudad es mas grande que la ciudad de la otra pieza). Solo importa que el tipo de borde coincida dentro de la lógica del juego.
- Se entiende por *jugada* cuando un jugador pone una pieza en el tablero.
- El archivo `demo.py` es solo un archivo de ejemplo.

8. Documento Entregable .md

Deben subir un documento en formato `Markdown` donde se identifique cada estructura que se utilizará dentro del programa. Además, se deben indicar los métotods que serán necesarios para el correcto modelamiento del problema.

9. Restricciones y alcances

- Tu programa debe ser desarrollado en Python v3.6.
- Esta tarea es estrictamente individual, y está regida por el Código de Honor de la Escuela: Clickear para Leer.
- Tu código debe seguir la guía de estilos descrita en el `PEP8`.
- Si no se encuentra especificado en el enunciado, asume que el uso de cualquier librería Python está prohibida. Pregunta en el foro si es que es posible utilizar alguna librería en particular.
- Si no subes el entregable, tendrás **5 décimas menos** en la nota final de la tarea.
- El ayudante puede castigar el puntaje[3] de tu tarea, si le parece adecuado. Se recomienda ordenar el código y ser lo más claro y eficiente posible en la creación algoritmos.
- Debes adjuntar un archivo `README.md` donde comentes sus alcances y el funcionamiento del sistema (*i.e.* manual de usuario) de forma *concisa* y *clara*. **Tendrás hasta 24 horas después de la fecha de entrega** de la tarea para subir el `README.md` a tu repositorio.
- El no cumplimiento de los puntos 5 y 6 vendrá asociado con un fuerte descuento.
- Crea un módulo para cada conjunto de clases. Divídelas por las relaciones y los tipos que poseen en común. **Se descontará hasta un punto si se entrega la tarea en un solo módulo[4].**
- **No se corregirán tareas que no interactúen con la gui.**
- Cualquier aspecto no especificado queda a tu criterio, siempre que no pase por sobre otro.

[3] Hasta −5 décimas.
[4] No agarres tu código de un solo módulo para dividirlo en dos; separa su código de forma lógica

10. Entrega

- `Entregable`
  - **Fecha/hora**: 6 de septiembre del 2017, 23:59 horas.
  - **Lugar:** Cuestionario en el Siding
- `Tarea`
  - **Fecha/hora:** 14 de septiembre del 2017, 23:59 horas.
  - **Lugar:** GitHub – Carpeta: Tareas/T02/
- `README.md`
  - **Fecha/hora:** 15 de septiembre del 2017, 23:59 horas.
  - **Lugar:** GitHub – Carpeta: Tareas/T02/

Tareas que no cumplan con las restricciones señaladas en este enunciado tendrán la calificación mínima (1.0).