Castle Defense

Se presenta un juego sencillo en el cual se necesita defender un castillo de oleadas de ataques. Se modeló el juego de manera tal que se puede saber las acciones a realizar para ganar y se creó un simulador del juego para poder jugarlo.

Uso del programa

1. Correr setup.sh. Esto instalará los paquetes necesarios de Python, por lo que solo es necesario correrlo la primera vez.
2. Asegurarse de tener la terminal en la misma carpeta que el archivo `main.py`.
3. En la terminal correr `python3 main.py` o `python main.py`.

Enunciado de Castle Defense

Estás en un castillo que será asediado por una fuerza que te supera, por suerte tu salvación se encuentra a unos N días de espera. Tu misión es resistir hasta que lleguen los refuerzos. En los almacenes del castillo se tienes unos recursos X los cuales te ayudarán.

Entre tus filas cuentas con una fuerza de A artesanos para confeccionar las armas necesarias y recolectar recursos y G guerreros que puedes asignar a las armas construídas para defenderte. Las armas se pueden demorar varios días en construirse y necesitan ser utilizadas por uno o varios guerreros para que sean efectivas durante el combate.

El enemigo tardará unos D días en llegar y luego atacará en oleadas cada vez más fuertes, aprovecha el tiempo que tienes para irte preparando para la dura batalla, recolecta recursos, construye armas que puedan contener las arremetidas furiosas de los malvados que quieren tomar las vidas de tus súbditos. Si todo sale bien seguro saldrás victorioso.

Suerte, esperemos que no queden solo ruinas para los aliados.

Aplicación

La aplicación consiste en una colección de niveles del juego anterior. Estos niveles se organizan por dificultad. La aplicación provee de dos modos. Una muestra una manera de jugar óptima de manera tal que se pueda completar el nivel seleccionado, mientras que otra el usuario tiene que interactuar con la aplicación hasta que pierda o gane el nivel.

Creación de niveles

La aplicación permite la adición de nuevos niveles de manera amigable al usuario común. Para eso es necesario crear las dependencias necesarias de **Nivel**, que son principalmente la **EstrategiaEnemiga** y el **Castillo**. Una vez se tengan estas la instancia de **Nivel** creada se puede añadir a cualquiera de las listas en `castle_defense_discrete/levels.py` en correspondencia al criterio de dificultad. Para ejemplos observar este mismo archivo.

API

Se implementó una API capaz de representar este tipo de problemas con el objetivo de poder usar la misma representación para diferentes objetivos, como por ejemplo usar otro solucionador o hacer un simulador del juego en el que el usuario interactúe y sea el que tome la decisiones. Esta API consiste en un conjunto de clases cada una con una responsabilidad

diferente.

- **Artesano**:
  - Representa la `cantidad` de habitantes Artesanos para determinada acción.
- **Guerrero**:
  - Representa la `cantidad` de habitantes Guerreros para determinada acción.
- **Arma**:
  - Representa el `ataque` de un arma. Describe la dependencia de construcción de dicha arma, la cantidad de artesanos y querreros necesarios para hacerla funcionar y los recursos necesarios para hacerla.
- **Recurso**:
  - Representa la `cantidad` de recursos para determinada acción.
  - Representa la `cantidad` de recursos disponibles y su capacidad de recolección por turno.
- **Castillo**:
  - Representa el castillo a defender. Contiene toda la información inicial necesaria, recursos, armas, artesanos, guerreros, armas iniciales.
- **AtaqueEnemigo**:
  - Representa un solo ataque enemigo con su `poder`.
- **EstrategiaEnemiga**:
  - Representa la estrategia completa del enemigo durante todo el juego.
- **Nivel**:
  - Representa una estructura que agrupa metadatos sobre el juego a realizar: dificultad, nombre, castillo, estrategia enemiga.

Modelo

Se modeló el problema como un problema de optimización lineal discreto. Para más información ver `CastleDefense.ipynb`. Este archivo contiene las restricciones y variables utilizadas en el modelado de este así como una descripción más detallada del proceso.

El problema se implementó usando **GEKKO** como solucionador de problemas de optimización. **GEKKO** permite una representación más simple de las variables y función objetivo del problema haciendo que sea más fácil de leer e implementar. La implementación se encuentra en `castle_defense_discrete/castle_gekko.py`.

GEKKO

Para trabajar con **GEKKO** primero hace falta crear un modelo de **GEKKO** y configurarlo para que pueda resolver problemas de optimización lineal en enteros:

```
from gekko import GEKKO

modelo = GEKKO(remote=False)
        modelo.options.SOLVER = 1  # APOPT is an MINLP solver
        modelo.options.LINEAR = 1 # Is a MILP
```

Para crear variables en GEKKO se usa el modelo previo para definirlas, en el constructor de la variable se le ponen distintos datos como por el ejemplo el nombre, los límites inferiores y superiores, si es entera la variable. Estas variables luego de que el problema sea resuelto contienen el valor óptimo encontrado para ellas.

```
# Variable entera del modelo con límite inferior 0 y nombre "x"
variable_entera = modelo.Var(lb=0, integer=True, name="x")
# Variable no entera del modelo con límite inferior 0 y nombre "y"
variable_no_entera = modelo.Var(lb=0, integer=False, name="y")
```

Para agregarle restricciones al modelo se usan ecuaciones que pueden operar con variables del modelo y también con constantes.

```
# x < y + 10
modelo.Equation(variable_entera < variable_no_entera + 10)
# x + y > 30
modelo.Equation(modelo.sum([variable_entera, variable_no_entera]) > 30)
```

La función objetivo se define operando las variables del problema con constantes en caso de ser necesario:

```
# Minimiza x+y
modelo.Minimize(variable_entera + variable_no_entera)
# Maximiza -(x + 10) + 2 * y
modelo.Maximize(-(variable_entera + 10) + 2 * variable_no_entera)
```

Resolver el modelo una vez definidas todas las restricciones variables

```

# Si lanza una excepción es porque el problema no tiene solución.
# En las variables creadas se encuentran los valores que conducen al óptimo
modelo.solve()
```

Simulación

Para que el usuario pueda jugar se implementó un modelo del juego que interactúa con el ambiente y este le brinda las acciones a realizar para ir avanzando en el progreso del juego. Se crearon diferentes clases para este modelo, las cuales se encuentran en `castle_defense_discrete/castle_simulation.py`:

- Accion: representan las posibles maneras de interactuar con el estado del juego.
    - **AsignarArtesanoArma**
    - **DesasignarArtesanoArma**
    - **AsignarArtesanoRecurso**
    - **DesasignarArtesanoRecurso**
    - **AsignarGuerreroArma**
    - **DesasignarGuerreroArma**
    - **PedirHint**
    - **PasarTurno**
- **EstadoDeJuego**: Contiene el estado del juego y reacciona a las diferentes acciones devolviendo un nuevo **EstadoDeJuego** con retroalimentación sobre la acción realizada

Asistencia durante el juego

Entre las acciones a realizar en el juego se encuentra la de **PedirHint**. Esta acción le dice al usuario dado el **EstadoDeJuego** actual cómo debe jugar para poder ganar el juego. Para extraer esta información se usó el modelo planteado en **GEKKO** y usando su solución se le da respuesta al usuario.