Proyecto Programación Orientada a Objetos

Convocatoria de enero de 2022
Curso 2021-2022

Última modificación: 14 de marzo de 2023

Índice

NOTA. Este fichero está sujeto a modificaciones continuas con el fin de que todo quede lo más claro posibles.

1. Introducción

En este documento se describe el trabajo a realizar para superar la segunda parte práctica de la asignatura Tecnología de la Programación de segundo curso del Grado en Matemáticas para la convocatoria de junio del curso 2022-2023, así como las normas de entrega.

Este documento incluye también las normas de realización, entrega y defensa, así como los criterios generales de evaluación que se utilizarán para corregir el trabajo entregado.

2. Requisitos para Aprobar

Para aprobar un ejercicio se deben de cumplir los siguientes requisitos mínimos. De no cumplirse alguno de ellos, el ejercicio se suspende automáticamente.

Requisito 1. Entregar un fichero .zip de acuerdo a las normas de entrega indicadas en el apartado 3.

Requisito 2. Entregar la documentación de acuerdo a las normas de entrega indicadas en el apartado 4.

Requisito 3. Cada programa principal se carga y lo programado funciona correctamente.

En particular, deberá de presentar como requisito un programa que cumpla con los "Requisitos Mínimos" que se indican en el punto 4 de la sección 5.2, aspecto que también deberá de estar documentado (como se indica en el apartado 4).

El profesor creará un nuevo proyecto en PyCharm y copiará todas las subcarpetas de la carpeta `Codigo` en la carpeta del proyecto de PyCharm. NO se copiará la carpeta `Codigo`, solo su contenido. El IDE no deberá mostrar errores (sí se admiten warnings) y se ejecutará los programas presentado sin generar errores con los objetos mínimos que permitan comprobar su funcionamiento correcto **de lo programado**.

Nunca, ningún programa, solicitará datos al usuario (salvo en los casos que se indicarán más adelante). Todo lo que fuera necesario deberá crearse en un `main()` del módulo principal por el procedimiento que se considere más adecuado.

Algunas observaciones:

- Si al abrir el proyecto, se generaran errores en el proyecto la práctica se dará por suspensa.
- Si durante la ejecución del proyecto entregado el programa no funcionara para los objetos indicados, la práctica se dará por suspensa sin importar la "gravedad" del error.

Requisito 4. Presentar un programa que responda a lo que se pide. No se invente el enunciado, léalo todo detenidamente y subraye lo que se le pide. En caso de dudas, pregunte antes de hacer interpretaciones propias.

Lo que se invente correrá por su cuenta y riesgo, y no será considerado en la evaluación final. Programe solo lo necesario.

3. Normas de entrega del proyecto

La práctica deberá de presentarse solo por el AV, vía `Tareas`, en las fechas establecidas y que será notificada por el AV.

Todo deberá entregarse en **un único fichero comprimido**. El fichero deberá tener cualquiera de las compresiones libres de patentes: 7z, zip, gzip, bzip, ... **ojo, .rar no está permitido**.

El fichero comprimido contendrá la **siguiente estructura de directorios** (**no uses acentos o espacios** en los nombres de los directorios/ficheros):

```
Proyecto
 |
 |-- Documentacion
 |      |-- Markdown
 |      |-- API
 |
 |-- Codigo
        |-- main.py
        |-- Paquete1
        |-- Paquete2
        |-- etc
```

El fichero comprimido contendrá dos subcarpetas.

- En una se tendrá el documento que contiene la documentación de la práctica. La documentación de la práctica consta de dos carpetas. En una se tendrán los ficheros Markdown y gráficos que se consideren necesarios y en otro se tendrá los ficheros .html sobe la API del proyecto.
- La otra carpeta contendrá el proyecto PyCharm completo: solo los fuentes **y no se incluirá, en ningún caso,** la carpeta **venv** del entorno virtual de Python.

4. Normas de entrega de la documentación

En la carpeta de la documentación, habrá dos subcarpetas:

1. Subcarpeta con los siguientes ficheros markdown y gráficos que contendrán lo siguiente:
   - `documentacion.md` contendrá los nombres de los autores y los siguientes apartados:
     - **Programas principales**, que serán un listado de los fichero Python .py que permiten ejecutar el programa o los distintos programas principales que haya construido.
       Todos los fichero .py principales estarán en la carpeta `Cogigo` Deberá existir al menos uno, llamado `main.py` que ejecute los requisitos mínimos.
     - **Clases abstractas**, con un listado de las clases abstractas indicando el propósito de cada una. El nombre de la clase y su propósito se indicará con una sola frase. A continuación indica el nombre de las clases hijas más representativas.
     - **Clases**, con un listado de las clases indicando el propósito de cada una. El nombre de la clase y su propósito se indicará con una sola frase. A continuación indica el nombre de las clases hijas más representativas. En su caso, destaca las que tengan herencia múltiple (entendiendo como tal las que heredan de dos o más clases, sin considerar la implementación de las interfaces que tuviesen).
     - **Interfaces**, con un listado de las interfaces indicando el propósito de cada una. El nombre de la interface y su propósito se indicará con una sola frase. A continuación indica el nombre de las clases más representativas que la implementen.
     - **Enumeraciones**, con un listado de las definidas. Para cada una, el nombre de la enumeración y su propósito se indicará con una sola frase. Indica para cada una qué clases la usa.
     - **Requisitos mínimos.** Indica, de acuerdo a los requisitos mínimos:
       - Módulos y clases donde se encuentran los métodos estáticos y de clase.
       - Módulos y clases donde se encuentra los atributos de clase.
       - Módulos donde están los tres pares de clases donde exista agregación/composición recíproca entre ellos e indica, al menos, un método delegado.
       - Cuáles son las de dos ramas que involucra al menos a 3 clases. Si hubiera alguna estructura de diamante indica cuál.
       - Cuáles son las tres clases abstractas y por qué lo son.
       - Cuáles son las cuatro interfaces abstractas.
       - En qué módulo principal se encuentra la creación de, al menos, 3 vehículos, uno de ellos con no menos de 3 ocupantes; y la creación de, al menos, 5 agentes persona fuera de cualquier vehículo.
     - **UML**, indicando los ficheros gráficos que se adjuntan y qué contiene cada uno. Sé lo más escueto posible.
     - **Paquetes**, apartado opcional. Indique brevemente cómo ha estructurado su programa con paquetes justificando la agrupación de las clases relacionadas.
     - **Implementaciones** contendrán el listado de los apartados implementados de acuerdo a los niveles realizados.
     - **Observaciones** Indique brevemente lo que quiera poner de relieve del ejercicio. Por ejemplo, problemas encontrados con la estructura del diamante, problemas encontrados, ...
       Indique el número de horas de dedicación al ejercicio de cada uno de los miembros del grupo por semana.
   - Los ficheros gráficos mencionados en el apartado UML de `documentacion.md` para que puedan ser cargados por el fichero .md.

   Todos los apartados anteriores los tienen en un fichero `documentacion.md` ya preparado en el AV. Guárdalo todo en la carpeta `Proyecto >Documentacion >Markdown`.
2. También habrá otra subcarpeta con la documentación de la API y que estará formada por la documentación generada por una de las librerías de Python: pydoctor, pdoc3 o Sphinx. Lea el primer capítulo de los apuntes sobre TDAs para saber cómo generarlos. La salida, que serán muchos ficheros .html, se presentarán en la carpeta `Proyecto >Documentacion >API`.
   Con caracter general:
   - Cada módulo comenzará con un docstring explicando la funcionalidad y justificación del mismo.
   - Cada clase contendrá un docstring explicando el propósito de dicha clase.
   - El método de inicialización tendrá un docstring indicando los atributos propios y, en su caso, los atributos

heredados más importante. Cada método (de instancia, de clase o estático) explicará en su docstring qué es lo que hace y, si fuera necesario, las particularidad y ejemplos de uso del mismo.
- Los métodos Getter y propiedades no requieren documentación.
- Los métodos Setter documentarán las restricciones de uso que tengan.

5. Evaluación

El plagio (total o parcial) en cualquiera de las partes presentadas para su evaluación conllevará una valoración de 0 puntos en la nota final de la convocatoria correspondiente para todos los alumnos implicados. El profesor usará diversas herramientas, incluidas algunas de carácter informático, para el seguimiento del trabajo y la detección de plagio sobre cualquier entrega realizada.

Muy importante:
- Hay partes del enunciado que se dejan libres y son ampliables: cabe esperar que no hayan dos proyectos con "las mismas ideas" (sería demasiada coincidencia). Si se detectarán "copias" de aspectos no incluidos explícitamente en los enunciados, se considerará que hay indicios de copia.
- No te explayes aumentando el enunciado abarcando aspectos que no se piden y sin cubrir antes los aspectos que se indican (y que son los que se exigen para aprobar). Antes de aumentar el enunciado pregunta a tu profesor de prácticas.

5.1. Criterios para la evaluación

Los aspectos que se tendrán en cuenta durante la evaluación y calificación del trabajo presentado son:
- Funcionamiento correcto de los programas presentados (no pueden tener errores en tiempo de ejecución)
- Presentación del trabajo/informe con todos los apartados mínimos solicitados.
- Usar correctamente las marcas/comandos de LaTeX para generar información. Documentación bien estructurada, ordenada y clara.
- En su caso, explicación, presentación y originalidad de la solución al problema planteado.
- Usar correctamente la metodología de la POO en todos los aspectos que han sido mostrados en las sesiones teorico prácticas, destacando:
  - diseñar e implementar correctamentes los TDAs,
  - definir correctamente las clases y sus relaciones de herencia o clientelismo, si las hubiera,
- En su caso, buena defensa oral de las soluciones presentadas en una entrevista presencial.

5.2. Calificación

La calificación de los ejercicios se hace de acuerdo a los niveles que a continuación se indican. Con carácter general, para calificar un nivel se tiene que haber superado el nivel anterior con, al menos, la mitad de su calificación. No desarrolle niveles posteriores si no ha desarrollado los anteriores, porque no serán evaluados. La calificación que se indica para cada nivel es su calificación máxima a la que puede aspirar.

1. **Documentación**
   Presentación completa de la documentación .md, gráficos y API en formato y forma de acuerdo al Requisito 2. En la documentación de la API no olvide desarrollar los docstrings de los métodos set() indicando claramente los controles de asignación.
2. **Gestor de Entidades**
   Implementación correcta del gestor de unidades, escenario, mapa y casillas. Deberán de presentar los atributos y métodos que se indican en el enunciado. Ver sección 6.1.
3. **Entidades**
   Implementación correcta de las entidades: físicas, estáticas y móviles. Mención especial tienen los caminos (que tienen métodos que se deben implementar) y los diferentes tipos de agentes (terrestres, marítimos, acuáticos) que definen distintos tipos de jerarquías de composición y herencia. Ver sección 6.2.
4. **Requisitos mínimos del programa**
   El programa funciona correctamente con todo lo exigido en los requisitos básicos.
   - Deberán de existir métodos estáticos y de clase.
   - Deberán de existir atributos de clase.
   - Exista agregación/composición recíproca entre, al menos, **tres pares** de clases. Indica qué clases usa y deja clara la delegación, si la hubiera.
   - Haya herencia. Con no menos de dos ramas con 3 clases involucradas en cada una. Justifica y, en su caso, explica si se genera una estructura de diamante y cómo se resuelve.
   - Existan no menos de tres clases abstractas.

- Debe contemplarse no menos de cuatro interfaces abstractas.
- Se crearán al menos 3 vehículos, uno de ellos con no menos de 3 ocupantes.
- Existirán, al menos, 5 agentes persona fuera de cualquier vehículo.

Se valorará la creación eficiente de los objetos que se piden.

5. **Paquetes, Excepciones y Ficheros.**
   Aquí se espera que construya una serie de paquetes coherentes para el proyecto junto con estas implementaciones:
   - Se disponen de clases propias para la gestión de errores.
   - Se usa la captura de errores en los lugares esperados.
   - El programa guarda la escena (al menos mapa y casillas). Usa `print()` si quieres hacer una versión sencilla.
   - El programa recupera la escena (al menos con mapa y casillas). Tarea un poco más complicada.
6. **Implementaciones. Camino más corto**
   Inplementa lo que se pide en la sección 6.3. Para pasar el nivel deberá construir correctamente el grafo que defina el escenario. Observa que aquí se espera al menos los TDAs Lista (mediante estructuras enlazadas) y Grafo (mediante listas enlazadas).
7. **Implementaciones. Steering Behaviour**
   Para este apartado se espera que se diseñe e implemente las distintas clases del apartado 6.4 así como todos los métodos **get_steering()**.
   Los algoritmos que se pueden implementar en este apartado son PathFollowing y Flocking (para este segundo se necesita obtener el grafo de partición del escenario).
   Si has implementado Dijkstra se recomienda centrarse en el PathFollowing. Si no lo has hecho, céntrate en el Flocking usando la partición del escenario. Con hacer uno de los dos, será más que suficiente para pasar el nivel.
8. **Implementaciones. Maquinas de estados**
   Para este apartado se espera que implemente el apartado 6.5. Algunos estados que se pueden definir son StateSeek, StatePathFollowing, StateFlocking. Para demostrar el funcionamiento correcto, cambie el estado del agente de vez en cuando.
9. **Interface Gráfica** - Tkinter.
   Se implementará como mínimo el código para las siguientes tareas:
   - Se dispone de un GUI que permite añadir y quitar agentes persona en un escenario.
   - En un canvas se muestra el comportamiento de los agentes. Basta con uno de los dos siguientes:
     - Se muestran 50 unidades haciendo flocking.
     - Hay 5 entidades haciendo el seguimiento de caminos aleatorios.

Las calificaciones de cada apartado son las siguientes:

Nivel realizado	Contenido exigido	Calificación Máxima	Acumulado
1	Documentación	.75	0.75
2	Gestor de entidades	2.5	3.25
3	Entidades	2.5	5.75
4	Requisitos mínimos del programa	.5	6.25
5	Paquetes, excepciones y ficheros	.75	7
6	Camino más corto	.75	7.75
7	Steering Behaviour	.75	8.5
8	Máquina de estados	.75	9.25
9	Interface Gráfica	.75	10

Observaciones:

- No entregar la documentación obligatoria descuenta 5 puntos sobre la calificación final.
- Se recomienda llegar como mínimo cubrir bien todos los apartados hasta el nivel 5 por los posibles fallos que se puedan cometer en los niveles anteriores.
- No realices el apartado 8 si antes no has hecho el 7.
- Puede hacer parte del apartado 9 si tienes hecho el 6 o el 7.
- Valora hasta qué nivel eres capaz de llegar. Recuerda que esto es un examen y como en todo examen está limitado a un tiempo. Sé **autocrítico** y determina hasta qué nivel eres capaz de llegar teniendo en cuenta las horas de dedicación a la asignatura, y trabaja por ello durante esas horas.

5.3. Entrevista

Recuerde que podrá ser citado para realizar una entrevista, que será grabada.

Como consideración general a la hora de presentarse a la entrevista de prácticas de la asignatura, se debe llevar muy bien repasado el trabajo realizado y presentado. En este sentido no se darán por válidas respuestas del estilo de *es que de esta parte ya no me acuerdo*, *es que de estoy muy nervioso/a*, ...

Durante la entrevista se preguntará cómo están hechas determinadas partes del proyecto de programación presen-

tado, al tiempo que normalmente se solicitará la modificación de alguna parte para que haga algo diferente. Algunas modificaciones pueden hacer referencia al diagrama UML, al diseño de algún algoritmo nuevo o a aspectos de programación.

El resultado de la entrevista puede cambiar por completo la calificación obtenida en el proyecto presentado.

6. Enunciado

Queremos generar un entorno de simulación basado en agentes. Todos los objetos del entorno están relacionados entre sí mediante relaciones de uso, asociación, agregación, composición y herencia. En el lenguaje natural se usan palabras como "usa", "tiene", "se compone de", "es un" pero el hecho de usarlas no significa que se correspondan necesariamente con las relaciones indicadas en términos de POO: puede que sí o puede que no. Interpreta todas las palabras indicadas como "está relacionado con" y determina, a partir del contexto y tu diseño, si la relación es de un tipo de otro. Del mismo modo, el hecho de que se diga que "hay" algún tipo de objeto no significa que se tenga que construir necesariamente clases nuevas. Analiza bien la situación. El entorno está inspirado en un entorno bélico ficticio.

6.1. Escenario y sus componentes

Un **gestor de entidades** administra las entidades del entorno de simulación. El gestor asigna identificadores consecutivos. A cada **entidad** que participa en la simulación, el gestor le asignará un identificador numérico único y la almacena en un diccionario. El gestor también informa de una entidad dado su identificador. En el caso de que se le suministre el nombre retornará todos los identificadores cuyas entidades tengan ese nombre.

Un **escenario** es un gestor de entidades que se caracteriza por tener los miembros de un gestor de entidades, un mapa, y distintas **listas** (de entidades) que son tratadas de forma independiente en el escenario.

El escenario, que es un rectángulo, se construye a partir de 4 valores: el alto y ancho del escenario y el número de filas y columnas que tiene su mapa.

Las listas de entidades y su diccionario se manipulan a partir de métodos específicos para dichas tareas contemplando, al menos, los métodos necesarios para añadir y quitar entidades en cada una de las listas. En una de tales listas estarán todas las entidades del escenario. También se puede tener una lista para las entidades que sean sensibles a los eventos del ordenador, otra puede representar a los agentes que pueden dibujarse en la pantalla, etc ....

El escenario dispone de un método que permite calcular un **árbol de partición** a partir de todas las entidades físicas y un parámetro que representa al número mínimos de entidades que son necesarias en una región para hacer una nueva partición en cuadrantes. Hay otro llamado **loop()** (invocado desde el módulo principal) que consta de un bucle que se ejecuta sin fin. En cada iteración recorre recorre cada una de sus listas e invoca, según la lista, a un método común a todas las entidades de esa lista. Es decir, sigue el patrón de diseño Observador.

Un **mapa** se codifica como un grid (matriz) formado por casillas. El mapa conoce a todas sus casillas y cada casilla conoce el mapa al que pertenece. También el mapa conoce al escenario en el que se encuentra. El constructor crea un mapa indicando cuántas casillas habrá por filas y cuántas habrá por columnas, junto con las dimensiones comunes de todas las casillas (el ancho y alto de las casilla).

Una **casilla** es un elemento de un mapa que se caracteriza por un coste, que representa el esfuerzo que tiene que hacer una entidad que transite por ella. Toda casilla también tiene una posición (i, j) dentro del mapa y tiene unas dimensiones (comunes a todas las casillas). Una casilla puede estar visible o no, transitable o no. Inicialmente todas son invisibles. Las casillas disponen de los siguiente métodos: el que establece una casilla en una posición (i, j) del mapa, los que cambian la visibilidad de una casilla (i, j) del mapa, el que permite consultar la casilla situada en un posición (i, j) del mapa, el que permite obtener la lista de casillas adyacentes dada una posición (i, j) del grid, la que determina la posición (x, y) del escenario que se corresponde con una casilla (i, j), la que determina cuál es la casilla (i, j) que le corresponde a una posición (x, y) del escenario, el que elimina todas las entidades físicas del escenario que están dentro de la casilla (i, j), el que nos indica si hay entidades físicas en la casilla (i, j), el que nos informa de cuáles son las entidades que se encuentra en esa casilla (i, j), el que nos indica si la casilla (i, j) es transitable y su coste, el que nos indica si la casilla es visible y los que cambian la visibilidad de la casilla.

Notar que algunos de los métodos se tengan que implementar en el escenario para tendrán que actuar por **delegación** sobre los métodos de los miembros que lo coomponenen. Por poner un ejemplo, puede ser que en tu diseño necesites que un agente del escenario necesite saber en qué casilla se encuentre, entonces el agente consultará al escenario, el escenario consultará al mapa y éste a la casilla. Estate pendiente de las delegaciones que se forman por asociación entre clases.

También permite obtener un grafo de su mapa (codificado como una lista de adyacencias).

6.2. Entidades del escenario

Al escenario de simulación habrá que añadirle las entidades que participarán en el simulación. Veamos cuántas hay.

Una **entidad** tiene un nombre, un identificador numérico que es asignado automáticamente por el gestor de entidades, conoce al escenario en le que se encuentra, y no se pueden instanciar. Entre las entidades se tienen las entidades **físicas** que tienen una posición, un radio interior y un radio exterior cuyo uso depende de las acciones que realice la entidad. Algunas no se pueden renderizar y otras sí (que se pueden mostrar gráficamente en la pantalla); por ejemplo, un waypoint no se renderiza pero un árbol sí.

Hay entidades **estáticas** renderizables como un árbol (que consta de un tronco y hojas) o un trozo de suelo. Un trozo puede representar al asfalto, al agua, etc .. Cada uno tienen asociado un coste y está asociado a la celda del mapa donde se ubica asignando al atributo coste de la celda el coste del trozo. Además, cuando se crea un trozo, éste redefine su posición de acuerdo a la posición central de la celda que le corresponde y redefine sus radios para que se corresponda al circulo inscrito a la celda. Algunas entidades estáticas no renderizables pueden agruparse para formar caminos. Las entidades que forman parte de un camino se le llama waypoints.

Un **camino** consta de una lista de waypoints y se le dota de métodos como calcular la longitud del camino (la suma de las distancias entre dos puntos/waypoints consecutivos), conocer el waypoint del camino más cercano a un ente dado así como la distancia entre ambos, y un método que retorna el siguiente waypoint. Todo waypoint se puede recorrer usando un iterador.

También hay entidades **móviles** que además tienen una masa y una velocidad. Se mueven de acuerdo a las ecuaciones de Newton y cada ente no puede superar una aceleración y velocidad máximas (y que son propias de cada ente móvil). Su movimiento se actualiza en cada iteración del **loop()** del escenariio.

Los **agentes** son entidades móviles que además disponen de cerebro. El cerebro estará modelado mediante una **máquina de estados**. Existen varios tipos de máquinas de estados: máquinas con estados locales, máquinas con estados locales y globales, máquinas de estados con pilas de estados, etc. El agente puede comunicarse con estas máquinas de estados mediante delegación tanto para actualizar el estado (local) del cerebro como para cambiar su estado. Ambos métodos son comunes a cualquier tipo de máquina de estados.

Un agente puede ser terrestre, aéreo o marítimo.

- Entre los terrestres tenemos a las personas y a los vehículos terrestres. Las personas pueden ser civiles o militares, distinguiendo estos últimos porque son portadoras de algún arma (pistola o metralleta, que se distinguen por su forma de disparar). De entre estos últimos están los soldados y los oficiales que se distinguen por ser líderes, distinguiendo su liderazgo entre cabo, teniente y coronel.
  Entre los vehículos terrestres están los de oruga y con ruedas. Vehículo de oruga son los tanques y los beowulf. Los tanques tienen un cañón en una torreta, una ametralladora y puede transportar hasta 5 personas. Los beowulf son articulados y sus dos cabinas tienen orugas, llevando en la segunda un equipamiento (un Bag) para todo tipo de misión. En los vehículos con ruedas se distinguen distintos tipos de vehículo entre 2 y 8 ruedas. Mencionamos solo las motos con sidecar (3 ruedas, hasta 2 personas, con metralleta en el sidecar) y el Jaguar (blindado de 6 ruedas con cañón en una torreta, para 8 personas). Mencionar que una metralleta y un cañón realizan acciones distintas.
- Entre los aéreos están los que tienen hélices y los que no. Entre los primeros están los helicópteros (2 hélices, una horizontal y otra vertical), los airbus (con 2 aspas, una en cada ala). Entre los que no tienen aspas están los bombardeos con capacidad nuclear y los cazas, que se distinguen por el arsenal que portan (establécelo tú).
- Entre los marítimos, que se caracterizan por el el nivel de profundidad que alcanzan, están los portaaviones (capaces de transportar aviones y helicópteros), cruceros (lanzan misiles) y submarinos (que transportan torpedos, lanzan misiles y tienen la capacidad de sumergirse o salir a flote).

Observa que existen distintas formas de desplazarse (ruedas, hélices, etc) y que también existen distintos tipos de armas (pistolas, metralletas, cañón, misil, etc) y cada una "dispara" de forma diferente. En este sentido, todas lanzan un proyectil.

Un proyectil es una entidad móvil que se desplaza en una dirección. Cada vez que se solicite su acción se mueve por el escenario en una dirección fija. Si el movimiento lo lleva a una posición fuera del escenario entonces desaparece; pero si entra en una casilla donde haya otras entidades no solo desaparece el proyectil sino que genera un daño a las unidades de la casilla donde explota provocando la desaparición de éstas.

Notar también que las personas pueden ser transportadas por vehículos. En estos casos actualizarán su posición al mismo valor de la posición del transporte. P.e. si una persona se encuentra en un Mastiff o un caza, la posición de la persona se actualiza con la posición del Mastiff o caza. La persona no realiza el movimiento, el vehículo lo hace por ella.

6.3. Camino de mínimo costo

También se puede calcular el mejor camino entre dos puntos, P y Q, del escenario (no del mapa) usando el algoritmo de **Dijkstra**.

El algoritmo de **Dijkstra** usará como grafo de entrada el que se construye a partir del mapa del escenario. El grafo se codificarán mediante **listas de adyacencias simplemente enlazadas**. Cada vértice del grafo se identificará con una celda transitable del mapa y tendrá como nodos adyacentes a los que representan a las celdas adyacentes

transitables. Se usará como coste el valor almacenado en la celda transitible y trabajará con una cola de prioridad implementada como un **Heap-Min**. El concepto de mejor vendrá en término de menor coste.

Dijkstra retornará un camino como una lista de nodos que tendrás que determinar pero que, en cualquier caso, vendrán en función de las casillas del mapa.

Tu método del mejor camino entre dos puntos, retornará un camino cuyos waypoints estarán situados en los puntos centrales de las casillas implicadas en la solución de Dijkstra y a los que añadirás un waypoint al inicio del camino correspondiente al punto P y se le añadirá al final el waypoint situado en el punto Q.

Si tienes dudas conceptuales sobre el algoritmo de Dijkstra y el heap-min consulta los apuntes del aula virtual.

6.4. Steering Behaviour

Un **SteeringBehaviour** representa a un comportamiento genérico en el movimiento del agente. Establece cómo se deberá de mover el agente en el escenario en función de cierto ente o entes objetivos (o targets) a través de un método que llamaremos **get_steering()**. Este método, que lo tienen todos los tipos de *SteeringBehaviour*, retorna un valor, llamado *steering*, que será interpretado como una fuerza. Es decir, cada SteeringBehaviour calcula una fuerza a partir de uno o varios objetivos (targets) usando cierto procedimiento dado por el método **get_steering()**.

Se distinguen varios tipos de steering que se clasifican por los objetivos (targets): Los básicos son lo que se marcan un objetivo y a partir de él calculan una fuerza, los delegados son los que se marcan un objetivo pero delegar en uno básico que puede usar otro objetivo y los grupales que son los que tiene varios objetivos simultáneamente para obtener una fuerza. Esta clasificación no definen clases de ningún tipo, cada steering define su propia clase. Cuando se combinan varios se habla de steerings compuestos.

Los steering behaviour *básicos* se caracterizan por tener una entidad física *target*. Un steerings básico es el **Seek**, que se usa para acercarse a un ente físico del escenario, el *target*, hasta que alcanza su radio interior. Otro steering básico es el **Flee**, que se usa para huir de un ente físico hasta que se llega a cierta distancia (que no es el radio exterior de ninguno de los entes, aunque puede venir en función de ellos). Las fuerzas que calculan son:

- Seek. $steering = normalizar(target - pos) \times max_v - vel$
  donde $target$ es la posición del ente objetivo, pos es la posición del agente, max_v es la velocidad máxima del agente, vel es la velocidad actual del agente
- Flee. $steering = normalizar(pos - target) \times max_v - vel$

El seek se usa para realizar un PathFollowing donde el agente sigue los waypoints de un camino. Lo primero que hará el agente es un Seek al waypoint que tenga más cercano del camino, cuando lo alcance se marca como objetivo el siguiente waypoint del camino y así sucesivamente. Se alcanza un waypoint cuando se entre en uno de sus radios.

El seek también se puede usar para realizar una persecución, donde el agente hace un seek a otro agente que está en movimiento.

También están los steering behaviour *delegados*, que aprovechan el **get_steering()** de un steering básico. El más conocido es el **Wander**, que es un Seek cuyo objetivo es un ente físico invisible en el escenario que existe delante de él y que cambia de posición de forma aleatoria pero siempre a la misma distancia del que realiza este steering. Otro es el seguimiento de caminos, **PathFollowing**, es un Seek donde el objetivo va cambiando y siempre es un waypoint del camino..

También están los steering behaviour *grupales*, donde un agente considera todos los agentes más cercanos a él para obtener un steering final. En estos casos se hace una partición del escenario y todos los agentes que están en el mismo nodo hoja del árbol de partición se consideran los más cercanos entre sí. Hay 3 steering behaviour grupales: **Cohesión**, **Separación** y **Alineación**. Vienen dados por las siguientes expresiones:

- Cohesión. Primer se calcula el centro de masas $Centro = \frac{1}{|V|-1}\sum_{e \in V}(e.pos - pos)$; donde V es la partición del plano donde se encuentra el agente, $|V|$ es el número de agentes que están en el cuadrante, e es cada uno de los agentes de V, $e.pos$ es la posición de cada e . Después se retorna el Seek marcando como objetivo a $Centro$.
- Separación. Para el agente a, se retorna el valor $steering = \sum_{e \in V-\{a\}} steer(a, e)$.
  A su vez $steer(a, e) = strength(a, e) \times \frac{e.pos - pos}{||e.pos - pos||}$ donde $strength(a, e) = \text{mín}\left\{\frac{decay}{dist^2}, max_a\right\}$ siendo max_a la máxima aceleración a la que puede llegar el agente, $dist^2$ es el cuadrado de la distancia entre los dos agentes y $decay$ es una constante de rechazo que tendrás que establecer.
- Alineación. Dada por $steering = \frac{1}{|V|-1}\sum_{e \in V-\{a\}} e.vel$; es decir, el promedio de las direcciones velocidad de los agentes. En ocasiones interesa considerar los vectores normalizados de $e.vel$ y multiplicar el resultado de $steering$ por alguna constante.

Cuando se aplican los tres se dice que se hace Flocking. El **flocking** es el steering resultante de la suma ponderada de los 3 steering grupales. El peso que se le dé a cada steering grupal lo establece el programador.

Por último están los steering behaviour *compuestos*, que son los que combinan los steering de forma similar a como lo hace un Flocking, con sumas ponderadas. Un ejemplo es el seguimiento al líder, donde varios agentes hacen simultáneamente un flee, seek y separación.

Observa que, en definitiva, un agente tendrá una lista de steering. Cada uno tendrá un target (o varios) para retornar una fuerza y todas son combinadas con un peso. Qué lista de SteeringBehaviour aplicará el agente dependerá del estado actual de la máquina de estados del agente. Si el agente tiene el estado de huir, puede usar un Flee; pero si tiene el estado 'crowd' podría usa un Flocking, etc.

6.5. Máquinas de estados

La implementación de agentes con cerebros de máquinas de estados consiste en implementar varios TDAs: el del agente, el de la máquina de estados y cada estado del agente.

StateXXXforAgent. Representan, a los estados de cualquier agente del tipo **Agent** (genérico). Habrá que definir una clase para cada estado; por ejemplo, StateSeek, StateFlocking, etc. Todos constan siempre de 3 operaciones. **No tienen constructor explícito**.

- `enter(agente: Agente, **kwargs)`. Las acciones que debe realizar el agente cuando entra en el estado XXX.
- `execute(agente: Agente, **kwargs)`. Las acciones que debe realizar el agente cuando se encuentra en el estado XXX. Se delegará a los métodos del agente para realizar las acciones que correspondan a este estado. Si fuera necesario, también se dará la orden de cambiar de estado.
- `exit(agente: Agente, **kwargs)`. Las acciones que debe realizar el agente cuando abandone el estado XXX.

Ejemplo: El método `.execute()` para el estado `StateLucharGuerrero` puede ser algo tan simple como `agente.luchar()` mientras que tenga energía, pero si tiene demasiadas heridas, se puede dar la orden de descansar cambiando de estado con `agente.chage_state(StateDescansarGuerrero())`. Observa bien que con esta invocación se crea una instancia de la clase `StateDescansarGuerrero`; pero no importa que se haga muchas veces, porque al ser singleton todas las instancias son el mismo objeto. El método `.exit()` del estado de luchar podría ser cambiar algunos atributos del agente (p.e. está invisible) y el método `.enter()` del descanso podría ser poner en marcha algún reloj de tiempo para contabilidad el tiempo de descanso. Observa también que las acciones `.luchar()` y `.change_state()` son del agente (**no se implementan en el estado**).

TDA FSM Representa a una máquina de estados.

- Creación: Se caracteriza porque conoce el `agente` sobre el que se va a aplicar la máquina, así como el `estado actual` y `estado previo` de dicho agente.
- `is_current(state: State): bool`. Indica si `state` coincide con el estado actual de la máquina para el agente.
- `update(**kwargs)`. Ejecuta el estado estado actual (e.d. invoca a su operador `execute()`) del agente.
- `change_state(new_state: State, **kwargs)`. Cambia el estado actual del agente por el estado dado. Previamente se guardará en el estado previo de la máquina y se habrá invocado a `exit()` del estado saliente. Posteriormente se invocará `enter()` del estado entrante.
- `revert_to_previous_state()`. Recupera como estado actual el que estuviera guardado como estado previo (si existe).

Puede que para la práctica no necesites todos los métodos que aquí se indican.

Agente Representa a un agente. Tiene como `cerebro` a una máquina de estados, con dos operaciones que actúan sobre él:

- Constructor. Se le asigna una máquina de estados (y a la máquina de estados se le asigna a él). A un agente también se le añadirán todos los atributos que lo definan y aquellos que les permita interactuar con el entorno.
- `update(**kwargs)`. Es invocado en cada ciclo de la simulación. Se limita a invocar al método `update()` de la máquina de estados en su versión más sencilla (que es en la que estamos).
- `change_state(new_state: State, **kwargs)`. Se limita a invocar al método `change_state()` de la máquina de estados. Es invocado desde los estados, método `execute()`, cuando hay que cambiar el estado actual del agente.
- Las operaciones/acciones necesarias que le sean requeridas por `execute()` u otros métodos de los estados.
- El resto de las operaciones que se contemple en este TDA dependerá del agente.

Resumen: En cada iteración de la simulación el agente se actualiza delegando en su cerebro, éste delega en el execute() de su estado actual y este método delega en un único método del agente. El método del agente se encargara de hacer toda las modificaciones de los atributos del agente y a invocar a los métodos que necesite para cumplir con la ejecución del estado. El método execute() del estado actual puede consultar atributos del agente para que, en función

de ellos, dar la orden al agente para que cambie de estado. Todo lo que hará el método execute() del estado es consultar estados para ver si tiene que cambiar de estado o indicarle al agente que haga una acción. Esto ya se indicaba en un ejercicio anterior (aunque no todos lo habéis hecho así en la primera parte).

7. Comentarios finales

Observación 1. En ingeniería de software, se llama singleton o instancia única a un patrón de diseño que permite restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. En palabras llanas: todos los objetos que se construyan de una clase singleton son el mismo. Conviene usar este patrón para crear los estados de un FSM. Si piensas, por ejemplo, en un videojuego de rol y consideras a todos los agentes/personajes de tipo guerrero verás que todos tienen los mismos estados: andar, correr, lazar lanzas, golpear con los puños, ... no es necesario que cada agente tenga un conjunto de estados que sea el mismo al de los demás guerreros. Es decir, los estados para un tipo de agente en un simulación siempre son los mismos.

En Python, los singleton se programan así:

1. Se crea la clase Singleton (cópiala tal cual en tu programa en el fichero `singlenton.py`)[1].

```
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]
```

2. En el fichero donde vayas a definir las clases/estados de tu agente, indica que son singleton. Por ejemplo:

```
from singleton import Singleton
class StateLuzOn(metaclass=Singleton):
    pass
class StateLuzOff(metaclass=Singleton):
    pass
```

   Cada uno de las clases/estados se construyen como se indican en el apartado siguiente.

Observación 2. Para ayudarte en la implementación, incluye en tu programa principal el siguiente código.

```
import logging
logging.basicConfig(filename='mylog.log',  # Fichero donde se guardarán los registros.
                    filemode='w',
                    level=logging.DEBUG,  #  Se registrará todo.
                    format='%(asctime)s | %(name)s | %(levelname)s | %(message)s')
```

En aquellos lugares donde quieras hacer un registro de lo que hace tu programa, no uses `print()`. Está prohibido el uso del `print()` salvo que lo invoque algún método por algo muy explícito (p.e. pedir datos al usuario, un menú, ...). En su lugar usa la instrucción `logging.info(texto)`. Pasa como argumento el `texto` que usarías en el `print()`, solo que ahora se guardará en el fichero `'mylog.log'`. Por ejemplo, puedes usar `logging.info()` en los métodos `.enter()` y `.exit()` de los estados para que muestre la posición actual del agente.

[1] Cópia esta clase y la guardas, no hay que cambiar nada.