Sistemas Inteligentes - Explorador de Marte.

Integrantes:

Fernando Gómez Herrera A01020319
Rodolfo Andrés Ramírez Valenzuela A01169701

INTRODUCCIÓN

Un **agente** es cualquier objeto o persona el cual es capaz de percibir su **medio ambiente** con la ayuda de **sensores** y también de actuar en su medio utilizando **actuadores**. De esta forma, el agente podrá llevar a cabo dichas acciones para la realización de un **objetivo**, ya sea la recolección de algún material, la introducción de algún elemento al medio, la producción de algo, entre otros.

El siguiente documento explica qué tipo de agente fue utilizado para la resolución del problema, además de el medio ambiente donde el agente fue puesto, se especificarán las capas que fueron establecidas dentro del modelo de *Arquitectura de Brooks* tanto para el caso individual, como el cooperativo del explorador de Marte.

El explorador de marte:

El problema del explorador de marte consiste en explorar un planeta para recoger muestras de roca, sin embargo no se sabe dónde se encuentra la roca pero se conoce que se encuentra generalmente amontonada. Un conjunto de vehículos (agentes) deben de recorrer el planeta recolectando muestras, una vez obtenidas deben de regresar a la nave base. No existe un mapa del planeta pero se conoce que el terreno es accidentado e imposibilita la comunicación entre los vehículos.

Agente y Medio ambiente

1. Agente Reactivo:

Existen diferentes tipos de agentes para la resolución de un problema, reactivos, basados en modelos, objetivos, utilidades, con aprendizaje y deliberativos. Para este caso, se utilizará un agente reactivo, el cual es el tipo de agentes más simple, estos agentes se basan en percepciones actuales ignorando el resto de las obtenidas previamente. Dichos agentes tienen una arquitectura basada en reglas y en percepciones, de este modo, la forma en que tomarán decisiones vendrá limitada a lo que sienten en ese momento.

2. Medio ambiente

Como fue descrito en el problema del Explorador de Marte, el agente estará ubicado en un medio donde se imposibilita la comunicación, de este modo existirá un caso simple donde el agente estará moviéndose aleatoriamente en la busca de rocas, sin importar si existen otros agentes o no, al regresar la nave emite una señal de radio la cual ayuda a los agentes que fueron en búsqueda de rocas a volver a ella. Para el caso cooperativo, los agentes tienen una capacidad limitada, por lo mismo no pueden recoger toda la roca, al tener dicho problema los agentes podrán dejar migajas con la finalidad de regresar nuevamente a la muestra.

Solución del problema:

Arquitectura de Software:

Para la solución del problema se utilizará:

- Python 2.7.6 https://www.python.org/download/releases/2.7.6/
- Pygame 1.9.1 http://www.pygame.org/download.shtml
- Gameobjects 0.0.3 https://pypi.python.org/pypi/gameobjects/0.0.3

Arquitectura de Brooke

La solución del problema requiere seguir una serie de instrucciones con la finalidad de que el agente tenga un correcto comportamiento dentro del medio ambiente, de esta forma como material complementario para la arquitectura propuesta se exponen **diagramas de estados** en la siguiente sección, la arquitectura está definida de la siguiente manera:

Individual

Predicado	**Acción**
Nada	Moverse aleatoriamente
Detección de la muestra	Recoger muestra
Muestras y no en nave	Ir a la nave
Muestras y en nave	Soltar muestras
Evitar obstáculo	

Cooperativo

Predicado	Acción
Nada	Moverse aleatoriamente
Detección de moronas	Tomar 1 y seguir el camino
Detección de la muestra	Recoger muestra
Muestras y no en nave	Ir a la nave y soltar moronas
Muestras y en nave	Soltar muestras
Evitar obstáculo	

Diagramas de Estado

Un agente, compuesto de diferentes estados (e.g recolección, movimiento, introducción, etc) puede pasar de un estado a otro por medio de sus sensores, es decir, cuando un agente percibe cierto objeto en el medio pasa de un estado de “Percepción” a otro, realizando una acción dentro del medio, es así como se establecen diagramas de estado para los agentes.

Individual

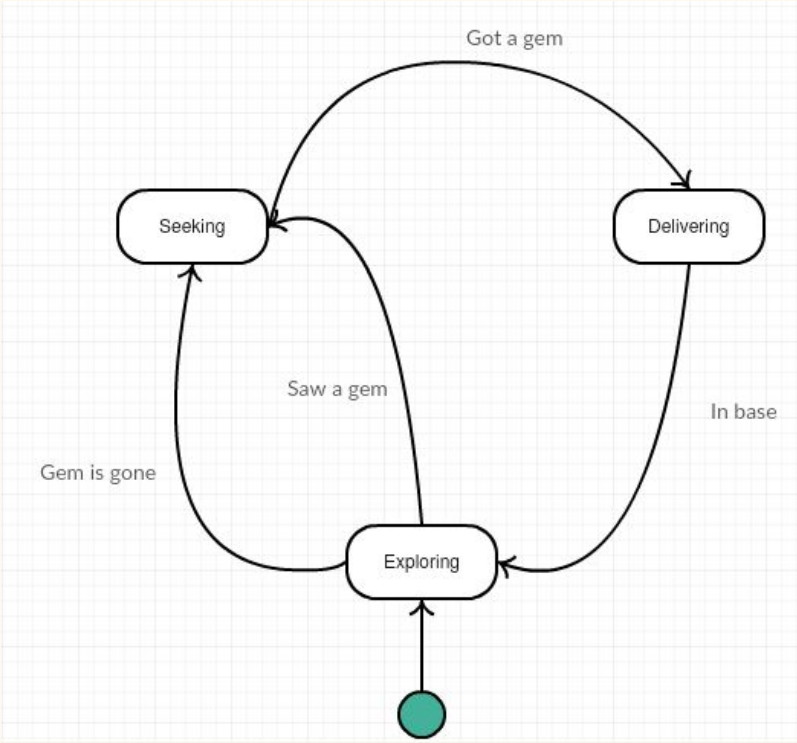

Figura 1. Diagrama de estado, comportamiento individual

La figura 1 muestra tres estados, los cuales serán utilizados por los agentes. El problema fue adaptado para la búsqueda de gemas con el uso de agentes como agentes, a continuación se describen los estados por los cuales pasará la agente.

Exploring: Estado inicial de la agente, en dicho estado la agente estará **evadiendo obstáculos** y moviéndose aleatoriamente dentro del medio ambiente en la búsqueda de gemas, al ver una gema, este pasará al estado **Seeking.**

Seeking: Estado subsecuente de **Exploring**, en dicho estado la agente ya ha detectado una gema, sin embargo esta puede regresar al estado inicial si una agente recogió la gema antes que ella. Si la agente llega exitosamente a la gema el estado cambiará a **Delivering**.

Delivering: Estado subsecuente de **Seeking** dicho estado está encargado en llevar la gema a la nave, una vez dentro, la agente dejará la gema y regresará a su estado inicial para la búsqueda de más gemas.

Cooperativo

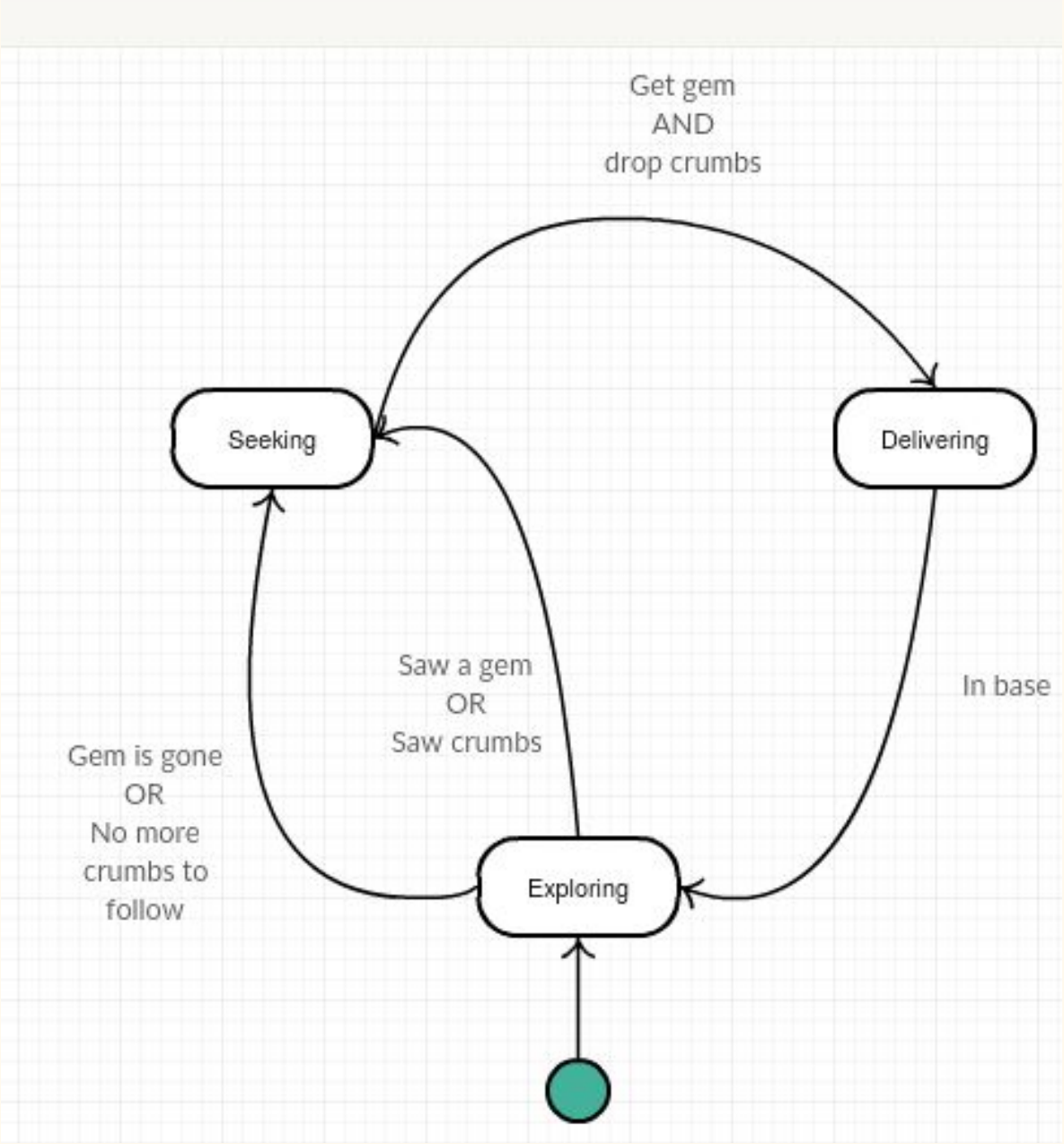

Figura 2. Diagrama de estado, comportamiento cooperativo.

La figura 2 muestra los tres estados con algunas modificaciones, esto con la finalidad de adoptar el comportamiento que debe de tener al hacerse cooperativo. Dado a que un agente tiene una capacidad limitada, los agentes dejarán una serie de moronas para que otros agentes lo sigan y puedan recolectar la gema que se dejó atrás.

Exploring: Estado inicial de la agente, en dicho estado la agente estará **evadiendo obstáculos** y moviéndose aleatoriamente dentro del medio ambiente en la búsqueda de gemas o **moronas** que fueron dejadas por otro agente, al ver una gema o morona, este pasará al estado **Seeking.**

Seeking: Estado subsecuente de **Exploring**, en dicho estado la agente ya ha detectado una gema u morona, en el caso de la gema este pasará directamente al estado **Delivering,** sin embargo, sí encontró una morona pasará nuevamente a exploring ya que buscará nuevamente moronas es preciso mencionar que este también puede regresar al estado inicial si una agente recogió la gema antes que ella.

Delivering: Estado subsecuente de **Seeking** dicho estado está encargado en llevar la gema a la nave, si la agente no pudo recoger toda la muestra, entonces dejará un camino de moronas para que otra agente pueda seguir el camino, en otro caso, si la agente ya ha llegado al sitio esta dejará la gema y regresará a su estado inicial.

Implementación

1. Archivos:

    a. Game.py: Dentro de este archivo se encuentra la implementación del explorador de marte en el caso individual.

    b. Cooperative.py: Dentro de este archivo se encuentra la implementación del explorador de marte en el caso grupal.

2. Clases: Dentro de ambos archivos se encuentran las siguientes clases:
    a. State: Clase base, la cual servirá para declarar las funciones base de todas las clases State. Dentro de esta clase se enceuntran las siguientes funciones:
        i. **__init__:** Inicializa el estado, con un nombre.
        ii. **do_actions**: función que se encarga de realizar las acciones como, moverse, recolectar, dejar muestra, etc.
        iii. **check_conditions**: Función que es utilizada para la transición de estados.
        iv. **entry_actions**: Función que es utilizada como entradas iniciales que recibe el estado, como por ejemplo la recolección de una muestra.
    b. StateMachine: State Machine servirá para simular la transición entre los estados de los agentes. Dentro de esta clase se encuentran las siguientes funciones:
        i. **__init__**: Inicializa la máquina de estados.
        ii. **add_state**: Se encarga de agregar un estado al diccionario de estados dentro de la máquina, es decir aquí se agregará la exploración, recolección, etc.
        iii. **think**: Esta función se encarga de simular el pensamiento que tiene el agente, si es correcto, entonces llama a la función auxiliar set_state para completar la transición.
        iv. **set_state**: Función auxiliar a think, el cual establece la transición del estado.
    c. World: Clase base, en la cual estarán guardados todos los objetos dentro de la simulación, los agentes, las muestras, el fondo de la pantalla entre otros. La clase World tiene las siguientes funciones:

i. **__init__:** Inicializa un nuevo mundo con un diccionario de entidades vacías y el fondo de la pantalla.
ii. **add_entity:** Función que agrega una entidad al diccionario, ya sea un agente, una roca, una gema.
iii. **remove_entity:** Función que remueve una entidad del diccionario.
iv. **get:** Función que devuelve la entidad determinada por su ID.
v. **process:** Función que realiza la entidad dentro del juego.
vi. **render:** Función que dibuja las entidades dentro del mundo.
vii. **get_close_entity:** Función que tendrá como parámetros la ubicación de un agente y de un objeto como es la roca o la gema, con la finalidad de calcular una colisión o la adquisición de una gema.
viii. **in_obstacle:** Función que devuelve si el agente puede pasar o no, dependiendo del obstáculo

d. **GameEntity:** Clase base que usarán todas las entidades del juego. La clase tiene las siguientes funciones:
   i. **__init__:** Tiene la función de inicializar la entidad dentro del juego, tendrá un mundo al que pertenece, un nombre, una imagen, una localización determinada por un vector de dos dimensiones, un destino, con la finalidad de calcular su siguiente posición, velocidad, **cerebro** (el cual es una máquina de estados) y un ID.
   ii. **render:** función con la cual se dibuja dentro del mundo.
   iii. **process:** función que determina las acciones que el agente tendrá.
e. **Gem:** Entidad que simula lo que los agentes tendrán que recolectar.
f. **Rock:** Entidad que simula un obstáculo dentro del mundo
g. **Agent:** Entidad base de la aplicación, dentro de ella se simula el agente reactivo, el cual tendrá la capacidad de cambiar de estados, percibir su alrededor y actuar dependiendo de lo que perciba, la clase contiene las siguientes funciones:
   i. **__init__** : Función que inicializa la instancia de Agent, la cual tendrá los diferentes estados como son, *explore, seek, deliver* estás se agregarán al cerebro y el agente podrá inicializar su ejecución dentro del mundo.
   ii. **carry:** Función que determina cuando el agente ha encontrado una gema, esta función hace un render del objeto dentro del agente.
   iii. **drop:** Función que determina cuando el agente ha llegado a la nave y tiene un objeto consigo, deposita el sprite dentro de la base.

iv. **drop_crumbs:** Función que se realiza cuando el agente se encuentra en el estado **AgentStateDroppingAndDelivering** el cual se describe en la siguiente sección.
v. **render:** Función que dibuja al agente dentro del mundo.

h. **AgentStateExploring:** Clase encargada de explorar el mundo, dentro de la misma se incluyen el movimiento aleatorio dentro del mundo, además de la función de no hacer colisión con las rocas, dicho estado, al encontrar una morona (**cooperativo)** o una gema cambiará su estado a **Seeking.**
i. **AgentStateSeeking:** Clase encargada de la recolección de una gema cuando ha sido encontrada por el estado anterior, una vez cerca cambiará a la clase **AgentStateDelivering.**
j. **AgentStateDelivering:** Clase encargada de la exportación de la gema a la base, cuando este ha llegado a la misma, regresa al estado inicial.
k. **GameOptions:** Clase encargada de determinar las opciones del juego.

El archivo cooperative.py contiene dos clases más de estado con la finalidad de poder simular las moronas.

l. **AgentStateDroppingAndDelivering:** Clase encargada de dejar en el camino moronas para que otros agentes las sigan, de igual forma tiene la misma funcionalidad de dejar la gema dentro de la nave.
m. **AgentStateSeekingAndPicking:** Clase encargada de la recolección de moronas, dentro de ella se buscarán moronas y gemas.

Posibles mejoras

Estamos muy conformes con el desempeño que tiene el programa, sin embargo siempre saldrán cosas que se puedan mejorar, una de ellas es la recolección de migajas, muchas veces los agentes al no tener alguna memoria, al seguir una morona, puede que la sigan para adelante o para atrás, por lo mismo se debe de establecer cierta relación con el radio de la nave. De igual forma podemos mejorar las colisiones que tiene el juego y posiblemente mejorar la interface del usuario.

Conclusiónes

Fernando:

Realizar esta actividad de programación presentó muchos retos para nosotros, retos tanto tecnológicos como intelectuales. En primera instancia no sabíamos cómo empezar ni qué tecnologías utilizar para desarrollar nuestra simulación. Fue difícil llegar a la conclusión de utilizar python y pygame para el proyecto, cosa que de ahí surgieron más problemas: nos enfrentamos a un framework totalmente nuevo (PyGame) el cual, al menos en mi caso, me costó mucho trabajo entender. Una vez solucionados los problemas tecnológicos surgieron aquellos relacionados con la teoría: ver en clase los modelos de agentes es muy diferente a tener que implementarlos de forma lógica en un programa. Crear nuestro modelo nos llevó bastante tiempo y si no fuera por mi compañero Rodolfo no habría podido entender el funcionamiento de los agentes.
La verdad es que fue una gran experiencia la cual nos retó bastante y me gustó el hecho de la libertad que se nos dio para desarrollar este proyecto.

Rodolfo:

Al estar tomando la clase y leer el libro individualmente me he dado cuenta que la Inteligencia Artificial, aunque muchas veces la vemos muy trivial y fácil de hacer tiene muchas complicaciones, en este proyecto desarrollamos una de las arquitecturas más simples, es decir un agente reactivo, en estos momentos me encuentro muy emocionado de poder desarrollar arquitecturas más complejas. Además de ello, este proyecto me ha ayudado a mejorar mis habilidades de programación con el lenguaje de programación Python y aprender el framework Pygame.

Honestamente me divertí mucho en esta actividad ya que fue un reto de investigación sobre cómo programar dicha arquitectura, además de haber podido compartir opiniones con mi equipo sobre cómo desarrollar la arquitectura y poder realizar la actividad exitosamente.

Agradecimientos:

Se agradece a *Daniel Cook* por el arte utilizado en este proyecto [1]. Tomamos las imágenes publicadas en su sitio web, las cuales tienen permiso de uso para proyectos personales. Todo el crédito a dicho autor.

Referencias:

[1] Cook D. (2007). Garden. *Danc's Miraculously Flexible Game Prototyping Tiles*. http://www.lostgarden.com/2007/05/dancs-miraculously-flexible-game.html

[2] Bendersky E. (2008). Eli Bendersky's website. *Writing a game in python with PyGame. Part II.* http://eli.thegreenplace.net/2008/12/20/writing-a-game-in-python-with-pygame-part-ii/

[3] Swelgart A. (2012). *Making Games with Python & Pygame*. https://inventwithpython.com/makinggames.pdf