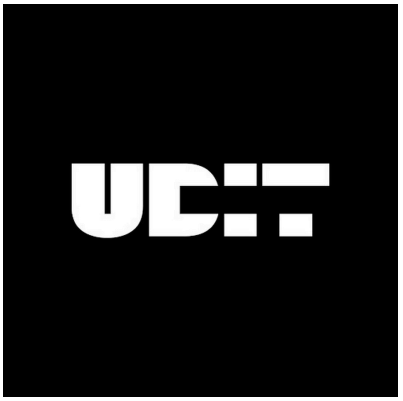
DISEÑO Y DESARROLLO DE VIDEOJUEGOS

Q-Learning y SARSA

Hecho por Iñaki Cezón Ortega y
Ignacio Tapia Marfil

Promoción: 2024/2025

Fecha de entrega: 15/05/2025



Índice

Resumen Ejecutivo

El entregable es un programa que implementa un agente que intentará salir de un laberinto por el mejor camino posible. Se han implementado los métodos de Q-Learning y SARSA como algoritmos de aprendizaje por refuerzo.

Diseño

El programa es un ejecutable de consola del sistema que imprime un mapa por pantalla importado de un archivo de texto. En el mapa aparecerán el personaje, muros, enemigos, tesoros y las salidas como posibles elementos.

La estructura del código se divide en “Controller”, “Enemy”, “Entity”, “FileReader”, “main”, “Map”, “MovementComponent”, “Player”, “Q-Learning”, “RandomController”, “Render”, “Sarsa” y “Treasure”.

El sistema de mapas funciona tal que: Hay que introducir en la carpeta Resources un archivo .txt con el formato (Map_xx) siendo “xx” un número, dentro de este escribir el laberinto con esta nomenclatura.

0 = Suelo.
1 = Paredes.
2 = Salida.
3 = Inicio.
4 = Enemigo.
5 = Tesoro.

Los números pueden ir separados entre sí con espacios, comas u otros caracteres. Cualquier espacio que se deje en las filas será rellenado con 1.

Las Q-tables se crearán en esa misma carpeta con el nombre del Mapa_xx + _qtable + nombre del algoritmo.

Para reentrenar los algoritmos desde 0 borrar el archivo del algoritmo y mapa correspondiente.

Explicación de clases

A continuación desglosamos las partes implementadas en el código:

- Controller: Clase padre que contiene funciones propias de los dos algoritmos Q-Learning y Sarsa. Las funciones comunes que define son el “LoadQTable” y el “SaveQTable” y el “DoAction”.

- Q-Learning: Clase hija de “Controller” donde se implementan las funciones mencionadas en Controller.
- Sarsa: Clase hija de “Controller” donde se implementan las funciones mencionadas en Controller.
- RandomController: Clase hija de “Controller” que implementa la función “DoAction” para que ejecute acciones aleatorias.
- Entity: Clase padre que define las entidades posibles del mapa (Player, Enemy, Treasure). Define la función “Update” y gestiona la posición x, y además de si la entidad está activa o inactiva y el símbolo asociado a cada entidad.
- Enemy: Clase hija de “Entity” que inicializa el símbolo (&) de la entidad cuando se crea.
- Player: Clase hija de “Entity” que inicializa el símbolo (@) de la entidad cuando se crea. Implementa la función “Update” que gestiona las llamadas al controller y el encuentro del jugador con tesoros o enemigos, además también implementa una función Reset que devuelve todos los valores a su estado original para simular una nueva generación. Si encuentra un enemigo hay un 50% de probabilidad de matarlo y que sume puntos o de morir y que ese individuo muera y pase al siguiente.
- Treasure: Clase hija de “Entity” que inicializa el símbolo ($) de la entidad cuando se crea.
- FileReader: Clase que lee el mapa deseado desde el archivo .txt y lo almacena en un vector de dos dimensiones. También incluye otra función que devuelve un vector Entities para luego generar estos en el laberinto.
- Render: Clase que contiene una función “RenderScene” que lee las casillas del mapa y las posiciones de las entidades y copia todos los chars correspondientes en un ostringstream para imprimir eso por pantalla.
- Map: Clase que contiene una estructura sobre el tipo de tile (Floor, Wall, Goal, Start) y que gestiona un vector de dos dimensiones para simular los tiles del laberinto.
- MovementController: Clase que se añade a una Entity para darle la habilidad de moverse por el entorno.
- main: Aquí se encuentra toda la estructura central del proyecto, en las variables globales encontramos todas las variables que van a ser editables desde consola para cambiar el comportamiento de los controllers, learning rate, discount rate, rewards etc.. Para hacer la selección de estos datos así como elegir el algoritmo a usar y el mapa tenemos la función Menú. Aquí dentro se llama al “SelectMap” que nos expondrá todos los archivos .txt con formato (Map_xx), después la selección de el controller y una función lambda a la que vamos a llamar para cada una de las variables a editar. Luego tenemos una función “Reset” que reinicia las entidades del mapa así como el controller, también tenemos la función que crea el array de entidades que habrá en el mapa y después tenemos la función principal del programa que es la función “Game”, aquí se llaman a las funciones que crean el mapa, las entidades, se crea el player con el controller seleccionado y se inicia el bucle general de todo el proyecto.

Análisis de Resultados

Para el análisis de resultados vamos a estudiar 2 mapas cargados desde un fichero de texto. El número de generaciones máximas está definido en 300 y el número de pasos posibles es de 5000. Posteriormente en el apartado de conclusiones se analizaran los resultados obtenidos del experimento.

Esta es la tabla que representa los valores que hemos utilizado para el entrenamiento de todas las pruebas posteriores.

Learning Rate	Discount Rate	Epsilon	Goal Reward	Movement Reward	Collision Reward	Kill Reward	Treasure Reward	Die Reward	GoBack Reward
0.7	0.6	0.6	1000	-1	-10	100	200	-1000	-5

Estas son las pruebas que hemos realizado sobre el primer mapa con los dos algoritmos.

Sarsa Map 1	Total	Individuo Prueba 1	Individuo Prueba 2	Individuo Prueba 3	Individuo Prueba 4
Generations	300	41	71	120	291
Maze Completed	291	35	63	111	282
Reaches Goals		No	Yes	Yes	Yes
Steps		100	15	12	12
Enemies Killed	11	0	1	0	0
Treasures Caught	594	2	2	2	2

Q-Learning Map 1	Total	Individuo Prueba 1	Individuo Prueba 2	Individuo Prueba 3	Individuo Prueba 4
Generations	299	39	106	165	256
Maze Completed	295	37	103	161	252
Reaches Goals		No	Yes	Yes	Yes
Steps		18	50	14	14
Enemies Killed	5	1	0	0	0
Treasures Caught	253	1	0	2	2

Estas son las pruebas realizadas para el segundo mapa con los dos algoritmos

Sarsa Map 2	Learning Phase 1	Individuo Prueba 1	Individuo Prueba 2	Individuo Prueba 3	Individuo Prueba 4
Generations	299	45	100	156	230
Maze Completed	42	4	9	40	42
Reaches Goal		Yes	No	No	No
Steps		4766	1062	3301	4656
Enemies Killed	47	1	0	0	0
Treasures Caught	162	0	0	0	0

Sarsa Map 2	Total	Individuo Prueba 1	Individuo Prueba 2	Individuo Prueba 3	Individuo Prueba 4
Generations	299	45	100	156	230
Maze Completed	65	11	24	65	65
Reaches Goals		No	No	No	No
Steps		3606	1943	2831	2336
Enemies Killed	37	0	0	0	0
Treasures Caught	112	0	0	0	0

Q-Learning Map 2	Total	Individuo Prueba 1	Individuo Prueba 2	Individuo Prueba 3	Individuo Prueba 4
Generations	299	45	100	200	260
Maze Completed	141	12	29	48	103
Reaches Goal		No	No	Yes	Yes
Steps		224	105	120	106
Enemies Killed	83	0	0	0	0
Treasures Caught	271	0	0	1	1

Q-Learning Map 2	Total	Individuo Prueba 1	Individuo Prueba 2	Individuo Prueba 3	Individuo Prueba 4
Generations	299	45	100	151	248
Maze Completed	211	16	39	65	161
Reaches Goals		No	Yes	Yes	Yes
Steps		1101	706	114	96
Enemies Killed	42	0	0	0	0
Treasures Caught	72	0	0	0	0

Conclusiones

A la hora del desarrollo de los algoritmos, para optimizar y afinar un poco más los algoritmos, hemos considerado implementar penalizaciones si chocase con un muro y si deshace un movimiento (retrocede una casilla), además de la penalización común en algoritmos de este tipo de una penalización continua de movimiento.

Respecto a ambos algoritmos hemos comprobado que ambos, incluso en nuestro “Map_02” que tiene una complejidad grande, en algunas de las pruebas consiguen acabar encontrando la salida, al principio solo con la penalización por choque y movimiento, le costaba muchas más generaciones encontrarla. Pero al introducir en los dos algoritmos la penalización por repetir el anterior movimiento, la cantidad de generaciones necesaria para que el algoritmo encontrará un camino óptimo a la salida se redujo muchísimo.

A la hora de añadir los enemigos y los tesoros nos encontramos con un problema, estos algoritmos sufren mucho en entornos cambiantes, cuando con cualquiera de los dos algoritmos el Player recoge por ejemplo un tesoro (Reward positivo) el algoritmo entiende que el movimiento que acaba de hacer es muy bueno, por lo tanto trata de replicarlo.
Lo que pasa es que cuando replica el movimiento, ese tesoro ya no está ahí, por lo que el Q de ese estado acción baja mucho y al final el algoritmo no se “acuerda” de que ahí hay un tesoro en las siguientes generaciones. Esto he visto que se puede solucionar de ciertas

formas pero esto implica cambiar un poco la naturaleza del algoritmo. Debido a estos problemas los algoritmos acaban encontrando que los caminos óptimos son simplemente los más directos a la meta.

Comparando los algoritmos en el segundo mapa, se puede apreciar claramente que al algoritmo Sarsa le cuesta mucho más encontrar un camino hasta la meta y a veces se queda “atascado” después del primer entrenamiento. En el caso de Q-Learning, en el primer entrenamiento logra encontrar un camino hasta la salida que además recoge un tesoro. Al ejecutar el algoritmo en este momento sin aprendizaje, completa el algoritmo a la perfección en 106 pasos recogiendo 1 tesoro y sin enfrentar enemigos. Tras esto se realiza un segundo entrenamiento a partir del anterior y se logra que el algoritmo realice el laberinto en 95 pasos sin recoger tesoros ni enfrentar enemigos. Esto demuestra que al volver a entrenarlo una segunda vez, ha preferido realizar el camino en menos pasos a sacrificar 11 pasos por coger un tesoro.

Con estos datos podemos observar que el algoritmo Q-Learning es más veloz para aprender que el Sarsa, debido a su naturaleza Off-Policy. También observamos que los algoritmos tal y como están realizados si consiguen encontrar las salidas al cabo de cierto número de generaciones pero tienden siempre a ignorar tanto enemigos como tesoros.