



UNIVERSIDAD SIMÓN BOLÍVAR

Ingeniería de Computación

Desarrollo de algoritmos de aprendizaje para un agente autónomo
aplicado a una variante del videojuego Super Mario Bros

Por

Daniel Barreto, Edgar Henríquez

Proyecto de Grado

Presentado ante la Ilustre Universidad Simón Bolívar
como Requerimiento Parcial para Optar el Título de
Ingeniero en Computación

Sartenejas, Octubre de 2010

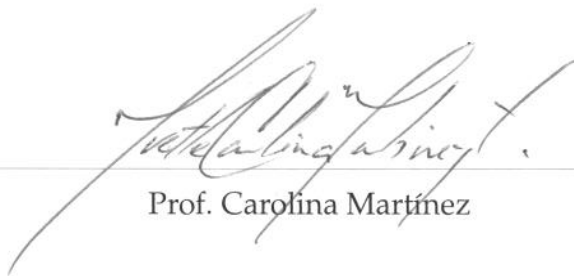
UNIVERSIDAD SIMÓN BOLÍVAR
DECANATO DE ESTUDIOS PROFESIONALES
COORDINACIÓN DE INGENIERÍA DE COMPUTACIÓN
ACTA FINAL DEL PROYECTO DE GRADO
DESARROLLO DE ALGORITMOS DE APRENDIZAJE PARA UN
AGENTE AUTÓNOMO APLICADO A UNA VARIANTE DEL
VIDEOJUEGO SUPER MARIO BROS

Presentado Por:
DANIEL BARRETO, EDGAR HENRÍQUEZ

Este proyecto de Grado ha sido aprobado por el siguiente jurado examinador:



Prof. Carolina Chang (Tutor Académico)



Prof. Carolina Martínez



Prof. Víctor Theoktisto

Desarrollo de algoritmos de aprendizaje para un agente autónomo aplicado a una variante del videojuego Super Mario Bros

Por

Daniel Barreto, Edgar Henríquez

RESUMEN

En el 2009 se realizó una competencia llamada “Mario AI Competition” cuyo propósito principal fue comparar y evaluar distintos enfoques de inteligencia artificial aplicados a un simple objetivo: crear un agente capaz de completar tantos niveles como sea posible de una versión especial de “Super Mario Bros”. Este trabajo comprende el desarrollo de un agente autónomo, llamado *Luigi*, capaz de desenvolverse de forma óptima en cualquier nivel del videojuego usado en la competencia mientras completa los objetivos de la misma. Se implementó un algoritmo de búsqueda informada en amplitud (A^*) como mecanismo de selección de las acciones a realizar por parte del agente. Para aplicar el algoritmo de búsqueda sobre el videojuego se desarrolló un *Simulador* y un *Reconstructor* de escenas. Ambas herramientas permitieron predecir estados futuros en el videojuego, y generar un grafo implícito donde realizar la búsqueda. Posteriormente se desarrollaron técnicas que usan el conocimiento de la física del juego para adaptar el algoritmo de búsqueda y optimizar la velocidad de exploración sobre el ambiente recorrido. Con las técnicas empleadas se consiguió mejorar considerablemente el rendimiento de *Luigi* hasta superar los resultados alcanzados por el agente ganador de la competencia, tanto en los criterios de evaluación considerados, como en el rendimiento algorítmico de la búsqueda: la velocidad de respuesta del agente y la cantidad de nodos visitados al buscar soluciones en un tiempo establecido. Finalmente, se desarrolló un planificador de alto nivel para que el agente considere otros objetivos durante el recorrido, y se realizó una aproximación inicial utilizando aprendizaje con *Neuroevolución*, logrando que *Luigi* completara objetivos adicionales a los propuestos para la competencia.

Índice general

Índice general	III
Índice de Cuadros	VII
Índice de Figuras	VIII
Glosario de Términos	IX
Capítulo 1. Introducción	1
Capítulo 2. Marco Teórico	4
2.1. El videojuego Super Mario Bros	4
2.2. La competencia “Mario AI Competition”	4
2.2.1. El simulador de juego	5
2.2.2. Interfaz del agente	6
2.2.3. Interfaz de observación	7
2.2.4. Reglas de la competencia	7
2.2.5. La evaluación de los agentes	8
2.3. Un agente artificialmente inteligente	8
2.4. Agentes presentados en la competencia	9
2.4.1. Algoritmos de búsqueda de caminos óptimos (A*)	10
2.4.2. Algoritmos genéticos	11
2.4.3. Algoritmos basados en redes neuronales	12
2.4.4. Algoritmos basados en reglas	12
2.5. Problemas de búsqueda y caminos óptimos	13
2.5.1. Algoritmo A*	14
2.6. Aprendizaje de máquinas	15
2.6.1. Redes Neuronales	15

2.6.2.	Algoritmos genéticos	16
2.6.3.	Neuroevolución	16
Capítulo 3. Diseño e Implementación del Agente		18
3.1.	Diseño General de Luigi	18
3.2.	Simulación	20
3.2.1.	Simulador de Física	20
3.2.2.	Escena Simulada	21
3.3.	Reconstrucción	22
3.4.	Algoritmo de Búsqueda (A*)	23
3.5.	El Espacio de Búsqueda	24
3.5.1.	Escenas Simuladas como nodos del problema	25
3.5.2.	Conexiones y la Función de Sucesores	26
3.6.	Consiguiendo la solución	27
3.6.1.	El Objetivo	27
3.6.2.	Costo de realizar acciones	28
3.6.3.	Función heurística	29
3.6.4.	Las Penalizaciones	34
3.6.5.	La Función de Selección	35
3.7.	Estrategias para cerrar nodos	35
3.8.	Tiempos de Respuesta	36
3.8.1.	Ajustar la búsqueda a un tiempo máximo	37
3.8.2.	Aumentar la velocidad de búsqueda	38
Capítulo 4. Diseño e Implementación del Aprendizaje		41
4.1.	Criterios de evaluación y Estrategias	41
4.2.	Aprendiendo a seleccionar estrategias	43
4.2.1.	Un seleccionador de estrategias	44
4.2.2.	La función de adaptación	45

4.2.3. Evolucionando al planificador	46
Capítulo 5. Experimentos y Resultados	48
5.1. Luigi para “Mario AI Competition”	48
5.1.1. Descripción general de los experimentos	49
5.1.2. Comparación contra el agente ganador de “Mario AI Competition” .	50
5.2. Aprendizaje	53
5.2.1. Descripción general de los experimentos	54
5.2.2. Entrenamiento para un ambiente aleatorio	55
5.2.3. Entrenamiento para ambientes de longitud variable	58
5.2.4. Entrenamiento para optimizar criterios de evaluación específicos . .	59
Capítulo 6. Conclusiones y Recomendaciones	63
6.1. Aportes realizados	65
6.2. Direcciones futuras	65
Bibliografía	67
Apéndice A. Interfaces de la competencia	69
A.1. Interfaz del agente	69
A.2. Interfaz de observación	70
Apéndice B. Mario AI Competition 2009	73
Apéndice C. Algoritmos implementados	74
C.1. A*	74
C.2. Reconstrucción de enemigos	75
C.3. A* con simulación retrasada	76
Apéndice D. Línea de tiempo de la Simulación	79
Apéndice E. Simulación interna del juego	80

Apéndice F. La Escena Simulada	82
F.1. Los atributos	82
F.2. Los métodos	82
Apéndice G. Reconstrucción	84
G.1. Reconstrucción de la geometría del nivel	84
G.2. Reconstrucción de los enemigos presentes	85
Apéndice H. Codificación de la geometría del nivel	87
Apéndice I. Las estructuras de datos	88
I.1. La lista de nodos abiertos	88
I.2. La lista de nodos cerrados	88
Apéndice J. Tablas de rendimiento (en uso de procesador)	90
Apéndice K. Experimentos y Resultados Auxiliares	91
K.1. Luigi para “Mario AI Competition”	91
K.1.1. Prueba preliminar sobre Luigi	91
K.1.2. <i>Luigi</i> vs. <i>Robin</i> con igual estrategia de selección de nodos cerrados	93
K.2. <i>Luigi</i> vs. <i>Luigi</i>	96
K.2.1. Variaciones del Factor de Parcialidad	96
K.2.2. Variaciones en la Restricción de Tiempo	99
K.3. Rendimiento de <i>Luigi</i> utilizando aprendizaje	100

Índice de cuadros

5.1. Configuración de Hardware para los experimentos	48
5.2. Resultados en variación de Fact. de Parcialidad entre <i>Luigi</i> y <i>Robin</i>	51
5.3. Comparación de objetivos entre Luigi y Robin	54
5.4. Rendimiento de <i>Luigi</i> con aprendizaje vs. <i>Luigi</i> de la Competencia	57
5.5. Validación del aprendizaje sobre niveles desconocidos	58
5.6. Validación del aprendizaje bajo niveles de longitud variable	60
5.7. Validación de aprendizaje selectivo	62
B.1. Puntuación de los participantes	73
H.1. Codificación de los elementos en la escena	87
J.1. <i>Profile</i> de rendimiento interno de <i>Luigi</i>	90
K.1. Resultados de pruebas preliminares	92
K.2. Comparación de objetivos en pruebas preliminares	94
K.3. Resultados en variaciones de <i>deltas</i> entre <i>Luigi</i> y <i>Robin</i>	95
K.4. Resultados en Fact. de Parcialidad en estrategias “Exc” e “Inc”	97
K.5. Resultados en Restricción de Tiempo para las estrategias “Exc” e “Inc”	99

Índice de figuras

3.1. Representación gráfica de <i>Luigi</i>	19
5.1. Nodos visitados en Fact. de Parcialidad entre <i>Luigi</i> y <i>Robin</i>	52
5.2. Metas no alcanzadas en Fact. de Parcialidad entre <i>Luigi</i> y <i>Robin</i>	52
5.3. Tiempo promedio de respuesta en Fact. de Parcialidad entre <i>Luigi</i> y <i>Robin</i> .	53
5.4. Entrenamiento del agente para un mundo aleatorio	56
5.5. Entrenamiento de <i>Luigi</i> sobre niveles de longitud variable	59
5.6. Entrenamiento de <i>Luigi</i> selectivo	61
A.1. Agente basado en objetivos	70
D.1. Comunicación de <i>Luigi</i> con el <i>engine</i> de la competencia	79
K.1. Nodos visitados en pruebas preliminares	92
K.2. Metas no alcanzadas en pruebas preliminares	93
K.3. Nodos visitados para las estrategias “Exc” e “Inc”	97
K.4. Metas no alcanzadas para las estrategias “Exc” e “Inc”	98
K.5. Tiempo promedio de respuesta para las estrategias “Exc” e “Inc”	98
K.6. Rendimiento de <i>Luigi</i> con aprendizaje vs. <i>Luigi</i> de la Competencia	100
K.7. Rendimiento de <i>Luigi</i> sobre ambientes aleatorios desconocidos	101
K.8. Rendimiento de <i>Luigi</i> con entrenamientos sobre niveles de longitud variable	101
K.9. Rendimiento de <i>Luigi</i> con entrenamientos sobre niveles de longitud variable	102
K.10. Rendimiento de <i>Luigi</i> con entrenamientos sobre niveles de longitud variable	102
K.11. Rendimiento de <i>Luigi</i> con entrenamientos sobre niveles de longitud variable	103
K.12. Rendimiento de <i>Luigi</i> utilizando aprendizaje selectivo	103

Glosario de Términos

- API *Application Programming Interface*. Conjunto de funciones y estructuras de datos para programar una aplicación.
- CIG *Computational Intelligence and Games*. Conferencia sobre investigación en inteligencia artificial aplicada a videojuegos.
- Fitness En modelos de poblaciones genéticas: describe la capacidad de cierto individuo o genotipo de reproducirse. Una función de fitness es un tipo de función objetivo que describe la optimalidad de una solución o hipótesis en un algoritmo genético.
- Heap *Montículo*. Estructura de datos de tipo árbol con información perteneciente a un conjunto ordenado, empleado para implementar colas de prioridad.
- ICE-GIC *International Consumer Electronics - Games Innovation Conference*. Conferencia sobre investigación en inteligencia artificial aplicada a videojuegos.
- Sprite Imagen o animación utilizada para conformar la visualización de una escena.
- TCP *Transmission Control Protocol*. Protocolo de comunicación y flujo de información entre equipos en una red.
- XML *Extensible Markup Language*. Metalenguaje extensible de etiquetas para definir gramáticas de lenguajes específicos.

Capítulo 1

Introducción

Actualmente los videojuegos han dejado de ser el pasatiempo ocasional de un pequeño grupo de personas, para convertirse en una actividad cotidiana para muchos. El desarrollo de videojuegos ha derivado en una disciplina que genera una importante cantidad de investigación y avances en áreas como la computación gráfica, la simulación de sistemas y, en nuestro interés particular, en la inteligencia artificial.

Los juegos, en general, han sido utilizados ampliamente en el área de la inteligencia artificial. En un principio, los juegos considerados como problemas o ejemplos clásicos fueron versiones de tablero, o juegos con una mecánica o sistema de reglas relativamente sencillo (ejem. damas [1], ajedrez [2, 3, 4, 5], backgammon [6]). Desde hace unos años los videojuegos han sido utilizados como herramienta en el estudio de comportamientos inteligentes y en el desarrollo de algoritmos de aprendizaje de máquina [7]. Esto se debe a que los videojuegos son ambientes ideales para probar algoritmos de aprendizaje y otros enfoques. Específicamente en los últimos años, los avances en el campo de los videojuegos han introducido nuevas características y condiciones en los mismos que han resultado útiles para la investigación en el área de la inteligencia artificial. El mismo hecho de que gente dedique parte de su tiempo a jugar videojuegos muestra que la habilidad y destreza para aprender a jugarlos está presente en el problema de dominarlos computacionalmente.

Simultáneamente con el nacimiento y evolución de los videojuegos, surgió la inquietud y necesidad de realizar agentes inteligentes que actúen en ellos igual o, incluso mejor, que los humanos. Ejemplos del desarrollo y aplicación de conceptos y algoritmos de inteligencia artificial los podemos encontrar en títulos arcade como PacMan [8], juegos de primera persona como Quake II [9, 25] y Unreal Tournament [10], juegos de estrategia como Age

of Empire [11], Creatures y Black & White [9, 8], simuladores como SimCity [10] y The Sims [9], variedad de juegos de carrera [12] y de acción [10]. Más allá de todo esto, una gran prueba del alto interés actual en el desarrollo de la inteligencia artificial sobre videojuegos es el constante crecimiento del número de conferencias y competencias dedicadas a éste tópico.

En el presente trabajo se toma a “Super Mario Bros.”, uno de los más populares juegos de plataforma como base para evaluar diferentes enfoques y técnicas de aprendizaje de máquinas y búsqueda informada de caminos óptimos. Durante el año 2009, Julian Togelius y Sergey Karakovskiy, pioneros en la investigación de agentes inteligentes usados para jugar “Super Mario Bros.” organizaron una competencia llamada “Mario AI Competition” [13], abierta a aceptar cualquier aproximación para crear nuevos agentes capaces de cumplir una única meta: Sobrevivir y completar la mayor cantidad posible de niveles del juego generados aleatoriamente. Los resultados de esta competencia demuestran claramente que los agentes más efectivos para resolver la tarea fueron aquellos basados en una búsqueda en amplitud informada (A^*) sobre el espacio de todas las posibles acciones del juego.

El objetivo general de este trabajo es diseñar, implementar y probar un agente autónomo, llamado *Luigi*, aplicado al ambiente aleatorio del clásico videojuego “Super Mario Bros.” usado en la competencia. Concretamente se plantea utilizar el algoritmo de búsqueda A^* para la selección de las acciones a ejecutar por el agente, y el enfoque de neuro-evolución para que el agente pueda seleccionar estrategias de búsqueda en función del objetivo a completar.

Para hacer que *Luigi* pueda desenvolverse en el juego de forma satisfactoria, se plantean los siguientes objetivos específicos:

1. Realizar un estudio sobre el problema que representa jugar “Super Mario Bros.” y la variante utilizada en la competencia.

2. Realizar un estudio general sobre los agentes presentados en “Mario AI Competition” y establecer un criterio sobre el rendimiento de las diferentes estrategias utilizadas por los participantes.
3. Superar el resultado alcanzado por el agente ganador de la competencia en el principal criterio de evaluación de la misma, utilizando el algoritmo de A^* como mecanismo de selección de las acciones a ejecutar por parte de *Luigi*.
4. Modificar el simulador del juego utilizado en la competencia para adaptarlo a reportar condiciones de evaluación genéricas.
5. Extender el mecanismo de decisión del agente implementado utilizando el enfoque de neuroevolución para optimizar distintos criterios de evaluación e identificar la estrategia de búsqueda más apropiada.
6. Evaluar el controlador de decisión implementado en el objetivo anterior con el simulador y los criterios de evaluación de la competencia.

El presente trabajo se encuentra dividido en 6 capítulos, incluida la introducción. El capítulo 2 introduce las definiciones necesarias para comprender los detalles más significativos de la implementación realizada, la cual está explicada en el capítulo 3. El capítulo 4 ofrece una explicación de los cambios realizados en el simulador de la competencia para contabilizar nuevos objetivos, y la implementación de la parte central del aprendizaje de *Luigi*. El capítulo 5 se dedica a explicar los experimentos que se realizaron para determinar la mejor configuración del agente y del algoritmo de aprendizaje, y los resultados obtenidos después de evaluarlo con los criterios de la competencia. Finalmente el capítulo 6 contiene las conclusiones y recomendaciones para trabajos futuros.

Capítulo 2

Marco Teórico

Este capítulo expone todos aquellos elementos teóricos de interés para desarrollar el argumento de la presente investigación. Se empieza por dar una explicación sobre el videojuego “Super Mario Bros.” y la competencia que lo utiliza como base de estudio para desarrollar agentes inteligentes. Posteriormente se explica que es un agente inteligente desarrollado para videojuegos, y las técnicas y algoritmos de búsqueda de caminos óptimos y aprendizaje de máquina utilizados en la investigación. Finalmente se describen los distintos tipos de agentes desarrollados para la competencia y sus resultados respectivos.

2.1. El videojuego Super Mario Bros

“Super Mario Bros.” es un videojuego de plataforma, originalmente diseñado por Shigeru Miyamoto, lanzado el 13 de septiembre de 1985 y producido por la compañía **Nintendo**, para la consola **Nintendo Entertainment System**. En el juego el personaje principal, *Mario*, busca rescatar a la *Princesa Peach* que fue capturada por *Bowser*, el rey de los *Koopas* y enemigo principal del protagonista. Los jugadores de “Super Mario Bros.” toman el papel de *Mario* (y a veces de su hermano menor *Luigi*, cuando es jugado simultáneamente por varias personas), y su objetivo principal es recorrer una serie de niveles que representan un reino ficticio llamado *Mushroom Kingdom*, tratando de evitar distintos tipos de enemigos, aliados del antagonista principal *Bowser*.

2.2. La competencia “Mario AI Competition”

“Mario AI Competition” es una competencia de inteligencia artificial iniciada en Abril del 2009 y culminada en Septiembre del mismo año, llevada a cabo por Julian Togelius y Sergey Karakovskiy, en asociación con la *IEEE Games Innovation Conference 09*. El objetivo principal de la competencia es “comparar diferentes enfoques en el desarrollo de agentes

artificiales, basados en paradigmas o técnicas como algoritmos de búsqueda de soluciones óptimas, aprendizaje, computación evolutiva, entre otros” [13].

Los agentes que participan en la competencia tienen como propósito completar la mayor cantidad de niveles (de dificultad creciente) en una simulación especial de “Super Mario Bros.”. El proceso de evaluación exacto para determinar la puntuación de los agentes es explicado en 2.2.5.

El simulador usado en la competencia, el cual se explica con detalle más adelante en la sección 2.2.1, implementa dos interfaces principales: *Agent* y *Environment*. Todos los agentes enviados a participar en la competencia deben implementar la primera interfaz, y obtendrán todos los datos para poder realizar su toma de decisiones desde la segunda. Cualquier agente enviado a la competencia que no cumpla con esas condiciones es descalificado.

2.2.1. El simulador de juego

El simulador utilizado para la competencia está basado en una versión bastante modificada del “Infinite Marios Bros” de Markus Persson [14], y es un tributo al videojuego “Super Mario Bros.” con el beneficio añadido de crear infinitos niveles aleatorios.

En el simulador original de Markus Persson, al momento de iniciar un nivel, se genera aleatoriamente el mismo usando una longitud preestablecida y se colocan elementos en él (como bloques, monedas, abismos y enemigos) de acuerdo a ciertas heurísticas. La generación de los niveles puede ser parametrizada estableciendo el nivel de dificultad deseado (entre un rango del 1 al 30, mientras más alto, más difícil). La dificultad elegida influye en el número y lugar de los abismos, así como en la cantidad y variedad de los enemigos y obstáculos presentes. Además, también es posible parametrizar la longitud y tipo de nivel a generar.

Las principales modificaciones realizadas por los organizadores sobre el simulador de la competencia incluyen: la capacidad de parametrizar la generación aleatoria de los niveles de juego, permitiendo evaluar distintos agentes sobre el mismo ambiente; la

eliminación de la dependencia de un output gráfico, y por último, la creación de un *API* para poder incluir cualquier controlador que cumpla con la interfaz *Agent*.

Otro punto importante a destacar del simulador usado en la competencia es que ofrece al agente la misma información que “vería” un ser humano al jugar el juego. Cuando una persona juega, ve únicamente una pequeña parte del nivel con *Mario* centrado en la pantalla, mientras el resto del mismo es totalmente desconocido. Análogamente, los agentes sólo pueden recibir en cualquier momento del juego los datos correspondientes a la parte del nivel visible, desconociendo el resto de la información del nivel.

Por otra parte, el simulador de la competencia permite un espacio de acciones discreto pero de amplitud considerable. Las posibles acciones representan los principales botones del control original de **Nintendo** usados en el juego: tres de las cuatro flechas direccionales (*Abajo*, *Izquierda* y *Derecha*) y dos botones (*A* y *B*). Podemos ver entonces que el agente tiene que devolver en cada tick de juego 5 posibles botones que pueden o no estar presionados, lo cual deja un espacio de acción de tamaño $2^5 = 32$, aunque varias de esas acciones no tienen sentido y son descartables, como por ejemplo presionar la dirección izquierda y derecha en la misma acción.

2.2.2. Interfaz del agente

Para la competencia “Mario AI Competition.” los agentes participantes deben ser programados en Java (comunicándose directamente con el simulador) o en cualquier otro lenguaje que pueda comunicarse a través de la interfaz TCP provista. Los agentes deben implementar la interfaz *Agent* del simulador, y sólo pueden usar la información disponible desde la interfaz *Environment*, y comunicarse con la simulación únicamente a través de los métodos definidos allí y en la interfaz del agente. Además, los agentes deben correr en tiempo real, lo que significa que en cada tick de juego no pueden consumir más de 40 ms de tiempo de procesador.

2.2.3. Interfaz de observación

El simulador en cada tick de juego provee al agente con información del entorno alrededor de él. Dicha información incluye:

- La posición exacta de Mario.
- Si Mario está tocando el suelo o si está saltando.
- Si Mario puede saltar.
- Si Mario está cargando un *Shell*.
- Si Mario puede disparar bolas de fuego.
- El estado de Mario: fuego, grande o pequeño.
- Un arreglo con la posición exacta y tipo de cada enemigo visible.
- Información discreta sobre el nivel visible (obstáculos, enemigos, abismos, monedas) en una matriz de 22x22.

La información que recibe el agente es provista por la interfaz *Environment* la cual, como se mencionó anteriormente, describe únicamente la parte del nivel que es visible durante cada tick de juego, mientras permanece oculto el resto del nivel. Para esto la interfaz *Environment* ofrece una matriz de enteros de 22x22, con Mario siempre en la fila 11 y columna 11. Cada casilla de la matriz representa un cuadro o espacio de la pantalla, y el número entero en cualquier casilla indica su contenido, que puede ser un espacio libre, un obstáculo indestructible, un bloque o ladrillo destructible, una moneda o algún enemigo.

2.2.4. Reglas de la competencia

La competencia enumera las siguientes reglas de participación:

1. El agente enviado debe implementar la Interfaz del Agente (descrita en el Apéndice A).

2. El agente debe usar únicamente la información provista a través de la Interfaz de Observación (descrita en el Apéndice A).
3. El agente no debe tomar más de 40 milisegundos en promedio por tick de juego.
4. Cualquier uso de `java.reflection` o métodos similares para tratar de influenciar directamente el funcionamiento o el estado interno del ambiente durante la corrida está prohibido.

2.2.5. La evaluación de los agentes

El objetivo principal de los agentes es llegar lo más lejos posible empleando la menor cantidad de tiempo. Para ello, cada agente en la competencia es sometido a la siguiente prueba: jugar 10 niveles bajo 4 dificultades distintas (0, 3, 5, 10) para un total de 40 niveles a completar. En cada nivel pueden variar el tipo del mismo (subterráneo, normal o castillo), la cantidad o variedad de los enemigos, abismos y obstáculos, y la longitud del nivel.

El puntaje final se calcula específicamente como la cantidad de pasos hacia adelante que consiguen avanzar los agentes en la prueba realizada. Sólo en caso de que dos o más agentes logren completar todos los niveles presentados, se procede a compararlos bajo los siguientes criterios:

- Tiempo restante total (en segundos del juego)
- Cantidad total de enemigos eliminados
- Estado de Mario al final de cada nivel completado

Para participar en la competencia es necesario desarrollar un agente inteligente capaz de resolver la prueba antes propuesta. En la siguiente sección se describe qué es un agente artificial, y cuáles son las condiciones necesarias para considerarlo “inteligente”.

2.3. Un agente artificialmente inteligente

En el campo de la inteligencia artificial se habla de *agente inteligente* o *agente artificialmente inteligente* como un modelo o entidad que percibe un ambiente por medio de

sensores y actúa sobre dicho ambiente a través de **actuadores** [15, 32–34]. Los agentes reciben información del ambiente, la procesan y ejecutan una acción o secuencia de acciones como respuesta, con el fin de lograr uno o más objetivos. En [15, 32–34], los sensores y actuadores de un agente son catalogados como la *arquitectura del agente*.

El proceso de razonar que acciones tomar se encuentra definido por una función que de forma genérica asigna cualquier posible estímulo recibido a alguna acción que pueda tomar el agente para modificar el ambiente en el que se encuentra. Esta función es llamada *programa del agente*, y debe estar ajustada a las características de su arquitectura.

El trabajo de la inteligencia artificial es diseñar el programa del agente que se encarga de realizar la asignación de cada percepción recibida a la acción adecuada. El programa del agente puede ser muy simple o tener alta complejidad, en [15, 44–54] se encuentran especificados 5 tipos básicos de programas de agente que van desde baja hasta alta complejidad: agente de reflejo simple, agente de reflejo basado en modelos, agente basado en objetivos, agente basado en utilidad y agentes de aprendizaje.

Además de los nombrados, se pueden realizar o ajustar distintos modelos para construir un programa de agente, y en el caso particular desarrollado en este trabajo el modelo escogido está basado en el presentado por *Ian Millington* en [9, 35–36] el cual divide la tarea de la inteligencia artificial en dos secciones: *estrategia* y *toma de decisiones*. Esto se explica con profundidad en los capítulos 3 y 4.

2.4. Agentes presentados en la competencia

Como se expuso anteriormente, la primera entrega de “Mario AI Competition” culminó en el 2009, y el resultado obtenido fue la presentación de varios agentes. Los mismos fueron implementados usando técnicas de inteligencia artificial diversas, como algoritmos de búsqueda de caminos óptimos, algoritmos evolutivos, redes neuronales, máquinas de estado y algoritmos sencillos basados en reglas.

Luego de haber estudiado el código fuente de todos los agentes presentados en la competencia, se presentan las características más relevantes de los mejores agentes agrupados

de acuerdo a la técnica o enfoque utilizado.

2.4.1. Algoritmos de búsqueda de caminos óptimos (A^*)

Los agentes que basan su toma de decisiones en algoritmos de búsqueda en amplitud como A^* buscan optimizar alguna heurística particular para cada estado en el que se encuentran.

Debido a que el ambiente que el controlador recibe es una matriz con posiciones de los objetos visibles en un momento dado y no un grafo, los agentes que utilizan ésta técnica deben poseer un simulador extra, el cual es llamado desde el agente y es utilizado para simular todas las posibles acciones que pueden devolverse en cada instante del juego.

Esta aproximación demostró ser la más efectiva para desarrollar agentes para la competencia. Los tres participantes que se exponen a continuación quedaron en primer, segundo y tercer lugar respectivamente, todos usando A^* .

■ Robin Baumgarten:

El agente presentado por Robin Baumgarten fue el ganador de la competencia. Utiliza A^* para conseguir el mejor salto hacia el borde derecho de la pantalla, tomando la distancia entre los nodos y dicho borde como heurística de su algoritmo.

Baumgarten analizó la física de *Mario* en el simulador de la competencia y la recreó en su propio simulador, para predecir los siguientes estados en el mundo.

La heurística se calcula asumiendo que *Mario* lleva la máxima velocidad posible en todo momento, y con dicha velocidad trata de calcular cuánto falta para llegar a la meta, siendo ésta el borde derecho de la pantalla.

En el cálculo de su heurística también toma en cuenta los posibles obstáculos que se puedan encontrar, sumando algún valor al costo del camino como penalización por ser herido o caer en abismos.

Su simulación resulta un poco lenta porque en cada análisis de toma de decisión sobre la acción a devolver se ejecuta una simulación extra de los posibles siguientes

estados.

- **Peter Lawford:**

El agente de Peter Lawford posee un controlador similar al desarrollado por Baumgarten, utilizando un enfoque prácticamente igual para calcular la mejor acción a tomar en cada tick de juego.

En su agente crea una simulación de la física de los personajes por medio de unas clases extras, llamadas “TheoricMario”, “TheoricEnemies” y “TheoricLevel”, que simulan parcialmente las posibles acciones siguientes, lo que le permite predecir cual es el mejor movimiento a ejecutar.

- **Andy Sloane:**

Similarmente a los agentes anteriores, el controlador de Andy Sloane también hace uso de una simulación “ad hoc” mediante unas clases llamadas “EnemyState”, “MarioState”, “WorldState”, entre otras, que le permiten predecir estados siguientes en la simulación. De esa forma, puede predecir la mejor acción a ejecutar en cada tick.

La única diferencia notable entre esta implementación y las dos anteriores es que el controlador de Andy Sloane no presenta propagación de penalizaciones por caminos que impliquen algún daño hacia *Mario*. Además, su rendimiento fue más bajo que el resto de los participantes que usaron A^* como técnica.

2.4.2. Algoritmos genéticos

- **Matthew Erickson:**

El agente de Matthew Erickson utiliza una combinación de reglas predefinidas junto con un algoritmo genético para evolucionar. La evolución del agente, que elige que botones presionar en cada iteración, se hizo con una población de 500 individuos, generando el 90 % de los descendientes con cruces de padres, 9 % mediante clonación y 1 % de mutación.

Las reglas predefinidas le permiten tomar decisiones específicas, por ejemplo cuando hay un enemigo o abismo cerca a la derecha.

2.4.3. Algoritmos basados en redes neuronales

- **Alexandru Paler:**

Este agente calcula que acción ejecutar mediante una red neuronal entrenada con datos de un humano jugando con el *HumanKeyboardAgent*. También utiliza un algoritmo de A* para buscar la ruta hacia el borde derecho de la pantalla, con la distancia *Manhattan* como heurística de la búsqueda. Luego calcula con la red los botones que deben usarse para mover a *Mario*.

2.4.4. Algoritmos basados en reglas

Los agentes basados en reglas operan de acuerdo a instrucciones de condición-acción, lo que significa que para cada condición recibida se ejecuta la acción correspondiente. Esto exige que el agente tenga predefinidas reglas del tipo *IF ... THEN* para cada observación diferente percibida del ambiente, lo que resulta complicado, poco flexible y raramente exitoso en problemas con ambientes de gran tamaño (que requieren muchas reglas).

- **Trond Ellingsen:**

Este agente tiene predefinida una serie de reglas que tratan de generalizar o abarcar todos los posibles patrones de percepciones recibidos desde el simulador, para los cuales se desencadena la acción correspondiente. De esta forma, la acción ejecutada por el agente en cada tick de juego siempre obedece a alguna condición que se cumple en las reglas establecidas en el controlador.

La mayoría de las reglas se basan en estimar peligros (de abismos o enemigos) y ejecutar movimientos preestablecidos para evitarlos. No se consideran monedas u obstáculos (como bloques) al momento de elegir la acción a ejecutar.

Los resultados finales de la competencia y la evaluación obtenida por cada agente puede verse en el Apéndice B. Como se mencionó anteriormente, los agentes implementados

exhibieron distintos enfoques y algoritmos para la resolución del problema, pudiendo agruparse en dos categorías: los primeros, basados en algoritmos de búsqueda de caminos, y los segundos en algoritmos de aprendizaje de máquina. Ambos enfoques serán tratados en las siguientes secciones.

2.5. Problemas de búsqueda y caminos óptimos

Existe una gran cantidad de problemas del mundo real donde es necesario encontrar una solución dentro de un amplio espacio de posibles estados o acciones. Los problemas de búsqueda son aquellos en los que se requiere encontrar una secuencia de acciones que conecte dos estados dados dentro del espacio antes mencionado.

Un problema de búsqueda se define formalmente como:

- Un espacio de estados S que constituye todos los posibles estados en el dominio del problema.
- Un estado inicial s_0 , a partir del cual se inicia la búsqueda.
- Un conjunto de estados objetivo $S_G \subseteq S$ que se quieren alcanzar, partiendo de s_0 .
- Un conjunto de operadores $A(s)$ para cada $s \in S$ que define las acciones que se pueden aplicar a cada estado posible.
- Un función de transición $f(a, s) \in S$ para todo $s \in S$ y $a \in A$. Esta función devuelve el estado que resulta de aplicar el operador (acción) a al estado s .
- Una función de costo $c(a, s)$ para cada $s \in S$ y $a \in A$. Esta función indica el costo de aplicar el operador a al estado s .

Cuando se habla de problemas de búsqueda, existen varios términos que deben ser definidos con claridad para evitar confusiones. A lo largo de este trabajo se utilizará la terminología definida en [16]. **Generar** un nodo significa crear en memoria principal la estructura de datos para representar al nodo; **expandir** un nodo se refiere a la acción de generar todos sus hijos haciendo uso de la función de transición definida anteriormente.

Cuando x' se genera como resultado de expandir x se dice que x es el **padre** de x' y que x' es **hijo** de x . Un **estado** es una configuración particular del problema y un **nodo** representa un estado alcanzado por un camino particular. Es posible que múltiples nodos representen al mismo estado, alcanzado por diferentes caminos, a los que se denominan **nodos duplicados**. Un **camino** es una secuencia de acciones aplicadas a un estado y la longitud de dicho camino define la **profundidad** del nodo.

En un algoritmo de búsqueda, la entrada es un problema y la respuesta es una solución que adopta la forma de una secuencia de acciones. Los algoritmos de búsqueda pueden clasificarse en dos categorías principales: algoritmos de búsqueda no informada o ciegos y algoritmos de búsqueda informada. Los primeros no conocen ningún tipo de información adicional de los estados más allá de la provista en la definición del problema, mientras que los segundos pueden saber cuando un estado no objetivo es “más prometedor” que otro estado [15, 73–81].

Existen varias consideraciones al evaluar un algoritmo de búsqueda: **Compleitud**, que significa que de existir solución, el algoritmo siempre la encuentra. **Optimalidad**, cuando el algoritmo siempre encuentra la mejor solución entre todas las posibles. Y por último, **Complejidad**, donde se valora tanto el tiempo que tarda en conseguir una solución como la cantidad de memoria que necesita para realizar la búsqueda.

Un caso particular de los problemas de búsqueda son aquellos donde no se requiere cualquier secuencia de acciones correcta, sino la solución óptima. Existen algoritmos específicos para esta clase de problemas, como el mencionado en la siguiente sección.

2.5.1. Algoritmo A*

Es un algoritmo de búsqueda en amplitud que encuentra el camino de menor costo en un grafo dado un nodo inicial y uno final. Para esto, el algoritmo usa una función basada en el costo acumulado desde el nodo inicial hasta el nodo actual más el costo estimado desde ese nodo hasta el nodo final. El costo “estimado” es lo que se conoce como función heurística $h(n)$, y es una parte fundamental para el funcionamiento del

algoritmo. El algoritmo de A* es completo y óptimo siempre que la función $h(n)$ escogida sea una heurística admisible, es decir, que $h(n)$ nunca sobrestime el costo de alcanzar el objetivo [15, 97–101].

A* es el algoritmo de búsqueda de caminos óptimos por excelencia en videojuegos debido a múltiples razones: es simple de implementar, es muy eficiente, y tiene muchas posibilidades de optimización [9, 223]. En el Apéndice C se presenta el pseudo-código del algoritmo.

2.6. Aprendizaje de máquinas

En [17, xv] se define aprendizaje de máquinas como el área de la inteligencia artificial que trata la pregunta de cómo construir programas de computadora que aprendan o mejoren automáticamente a través de la experiencia. En dicha área se estudian, formulan y describen distintas técnicas, algoritmos y paradigmas que buscan imitar y trasladar el proceso de aprendizaje y formación de conocimiento de los seres humanos a las computadoras. Los algoritmos basados en aprendizaje de máquinas más conocidos incluyen las redes neuronales y algoritmos basados en computación evolutiva, como los algoritmos genéticos. Ambos ejemplos serán explicados a continuación.

2.6.1. Redes Neuronales

Intuitivamente, las redes neuronales artificiales son un modelo computacional que trata de simular la estructura y/o funcionamiento de las redes neuronales biológicas. El procesamiento de la información se realiza siguiendo un enfoque conexionista, donde el sistema, como un todo, se adapta a los datos internos o externos que circulan a través de la red durante el proceso de aprendizaje. Las redes neuronales están compuestas de un conjunto de unidades o nodos de procesamiento unidos mediante conexiones. Cada conexión tiene asociado un peso numérico que determina la magnitud y signo de dicha conexión, y dichos pesos constituyen el principal recurso de memoria a largo plazo en las redes neuronales [15, 737].

Los métodos de aprendizaje basados en redes neuronales artificiales proveen un enfoque robusto a errores en los datos de entrenamiento y han sido aplicados satisfactoriamente a problemas como interpretación de escenas visuales, reconocimiento de voz y aprendizaje de estrategias de control para robots. Para ciertos tipos de problemas, como interpretación de complejos datos de sensores, las redes neuronales artificiales están entre los más efectivos métodos de aprendizaje disponibles hoy día [17, 81].

2.6.2. Algoritmos genéticos

Los algoritmos genéticos proveen un mecanismo de aprendizaje inspirado en el proceso de evolución biológica, con conceptos como herencia, mutación, selección y cruzamiento [17, 249]. El problema abordado por los algoritmos genéticos es buscar un espacio de hipótesis posibles para seleccionar la mejor hipótesis. La mejor hipótesis es definida como aquella que optimiza una medida numérica predefinida para el problema en cuestión, llamada *medida de adaptación* o *fitness* de la hipótesis. Los algoritmos genéticos comparten la siguiente estructura: el algoritmo actualiza iterativamente el espacio de hipótesis manejado, llamado *población*. En cada iteración, todos los miembros de la población son evaluados de acuerdo a una función de adaptación. Una población nueva es creada seleccionando probabilísticamente los individuos con mejor *adaptación* de la población actual. Algunos de los individuos seleccionados son pasados a la siguiente generación sin cambios. Otros son usados como base para crear nuevos individuos o hijos producto de la aplicación de operadores genéticos como *cruzamiento* o *mutación*.

2.6.3. Neuroevolución

La neuroevolución es un mecanismo de aprendizaje de máquinas que utiliza algoritmos evolutivos para entrenar redes neuronales artificiales [18]. El mecanismo consiste en modificar los pesos o topología de la red para que aprenda una tarea específica. Para ello, técnicas de computación evolutiva son empleadas para buscar parámetros de la red que maximicen una función de adaptación que mide el rendimiento en la tarea a aprender. Comparada a otros métodos de aprendizaje para redes neuronales, la neuroevolución es

altamente general, permitiendo el aprendizaje sin un proceso de supervisión estricto [19].

Debido a que la mayoría de los algoritmos de aprendizaje para redes neuronales están basados en un entrenamiento supervisado, y sin embargo muchos problemas del mundo real no son fácilmente representables bajo este enfoque, la principal motivación de la neuroevolución es poder entrenar redes neuronales en tareas con información de reforzamiento escasa o dispersa [18]. Para el entrenamiento de la red existen dos variaciones principales: la primera basada en la evolución de los pesos de redes con topologías previamente definidas, que es la elegida para este trabajo, y la segunda basada en la evolución de los pesos así como también la estructura de la red.

Este enfoque ha demostrado ser útil en una amplia variedad de problemas y especialmente prometedor en tareas donde el estado es parcialmente observable, como control de vehículos, alerta de colisiones y control de personajes en videojuegos [20, 21].

Capítulo 3

Diseño e Implementación del Agente

En este capítulo se describen todos los detalles relevantes a la implementación del agente inteligente realizado, el cual será llamado *Luigi*. Primero se presenta una descripción general del agente y la justificación del enfoque elegido. A continuación se explica la implementación del simulador de la física del juego, que es desarrollado para poder predecir estados o escenas de juego futuros. Luego se presenta la herramienta desarrollada para que el simulador pueda reconstruir escenas completas que vienen del *motor* de la competencia. Posteriormente se detalla el algoritmo de búsqueda utilizado para que *Luigi* pueda conseguir los objetivos planteados obteniendo la secuencia de acciones óptima para llegar a la meta, que involucra detalles de la heurística propuesta, función de sucesores, función de establecimientos de metas, penalizaciones y otros detalles de la búsqueda. Finalmente se explican las optimizaciones realizadas sobre el algoritmo de A* para lograr una búsqueda más amplia en menos tiempo y mejorar los resultados de *Luigi*.

3.1. Diseño General de Luigi

Para la implementación de *Luigi* se decidió construir un agente basado en objetivos [14] (Figura A.1), utilizando el algoritmo de A* como mecanismo de selección de la acción a ejecutar. Dicha decisión se fundamenta en dos razones principales: (1) los resultados de la competencia, mostrados en el Apéndice B, evidenciaron que los agentes más efectivos para resolver la tarea fueron aquellos basados en una búsqueda en amplitud informada (A*) sobre el espacio de todas las acciones posibles en el juego, y (2) debido a las características particulares del problema a resolver, como un juego con leyes físicas totalmente determinísticas y fácilmente replicables, es posible predecir estados futuros en el juego (dentro de las limitaciones dadas en la competencia), y aplicar un algoritmo como A* sobre el espacio de búsqueda generado.

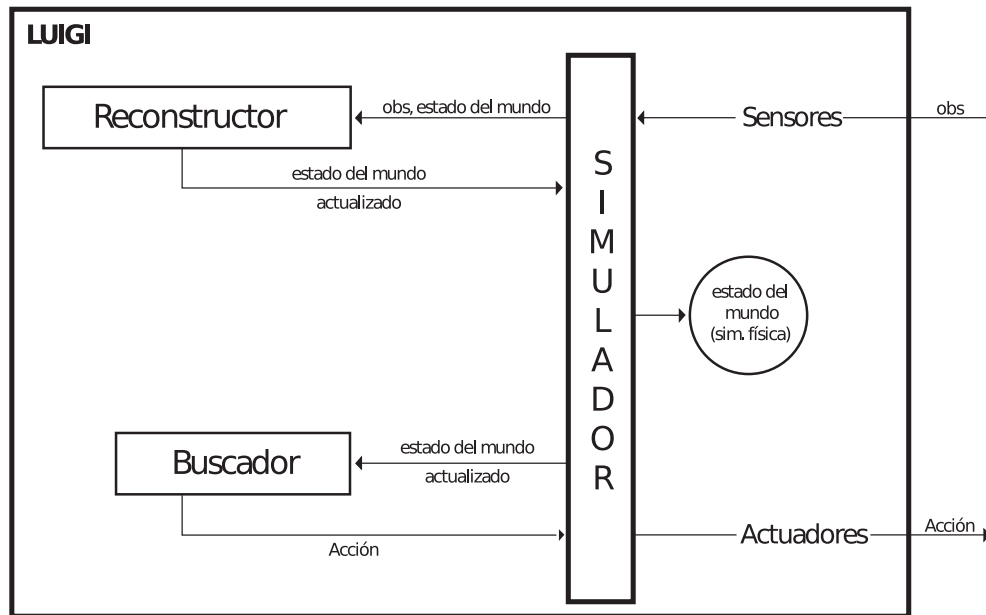


Figura 3.1: Representación gráfica de *Luigi*

Bajo la decisión de utilizar A^* , se realizó un agente inteligente que cumple con la interfaz señalada en el Apéndice A. El mismo cuenta con tres herramientas para “razonar” sus acciones dada cualquier escena (Figura 3.1), y estas son:

- Un *Simulador* que permite recrear los cambios producidos por una acción en una escena dada. Mediante esta herramienta *Luigi* es capaz de predecir los próximos estados o escenas a partir de la escena donde se encuentra, conociendo su posición exacta después de aplicar una acción cualquiera. De forma similar, el *Simulador* recrea el movimiento de los enemigos dentro de la escena, facilitándole a *Luigi* hacer un plan de acciones para lograr su objetivo sin ser lastimado.
- Un *Reconstructor* de escenas que recibe la información de la escena que está viendo *Luigi*, y que es provista por el *motor* de la competencia (detallada en la Sección 2.2.3). Dicha información incluye la configuración y geometría del nivel (obstáculos, abismos y ladrillos), y los enemigos que se encuentran en él. Posteriormente el *Reconstructor* analiza la información recibida para integrarla con la simulación interna del agente, y mantener el *Simulador* sincronizado con el nivel jugado hasta el momento.

- Finalmente, un *Buscador* que ejecuta un algoritmo de búsqueda informada en amplitud (A^*) para conseguir la secuencia óptima de acciones que *Luigi* debe ejecutar para lograr su objetivo. El algoritmo utiliza el espacio de búsqueda generado por todas las posibles acciones que se pueden realizar en una escena dada, empleando para ello las predicciones obtenidas a través del *Simulador*.

El funcionamiento de *Luigi*, desde el momento en que recibe la observación de la escena actual y hasta que devuelve la acción a ejecutar, comprende las siguientes etapas:

- **Preparación:** Recibir la información de la observación y sincronizarla con la representación interna del nivel actual.
- **Simulación:** Simular la última acción devuelta al ambiente (*Environment*) para quedar a la par con la escena manejada por el simulador de la competencia.
- **Chequeo:** Verificar que ambas simulaciones, tanto la de *Luigi* como la que es llevada por el *motor* de la competencia, hayan quedado iguales. De no ser así, tratar de arreglar la simulación interna del agente.
- **Búsqueda:** Buscar el camino óptimo (esto es, la secuencia de acciones más conveniente) para lograr el objetivo, y devolver la primera acción de dicha solución.

A continuación se explican detalladamente dichas etapas o componentes.

3.2. Simulación

La simulación en el agente está conformada por dos partes esenciales: el *Simulador* de la física del mundo y las *Escenas Simuladas*. Ambas se describen a continuación.

3.2.1. Simulador de Física

El simulador de física, implementado en `Simulator.java`, posee una escena donde mantiene la simulación del juego, la cual es explicada más adelante en la Sección 3.2.2. Las tareas realizadas por el simulador comprenden, en orden de ejecución:

1. Verificar si el juego ha terminado. De ser así, finalizar la ejecución.
2. Utilizar los datos recibidos en el tick anterior por parte del *motor* de la competencia y sincronizar el estado de *Luigi* en la escena simulada con ellos. De forma similar, usando los datos del tick anterior, sincronizar el estado de la escena simulada utilizando el *Reconstructor* de escenas. Este proceso es detallado más adelante en la Sección 3.3.
3. Guardar en memoria los datos recibidos por el *motor* de la competencia en el tick actual, para sincronizarlos en el siguiente tick.
4. Simular la acción devuelta al *motor* de la competencia en el tick anterior, diciéndole a la escena simulada que avance un tick usando dicha acción.
5. Chequear que la simulación actual sea igual a la observación recibida del *motor* de la competencia. Este proceso únicamente se encarga de verificar que la posición de *Luigi*, luego de haber simulado la última acción, sea igual a la posición actual de *Mario* en el *motor* de la competencia.

Como se puede apreciar en el orden de las acciones ejecutadas por el simulador, la simulación interna del agente siempre va un paso por detrás con respecto a la simulación del juego en el *motor* de la competencia (Figura D.1). La justificación de esta decisión de diseño se encuentra detallada en el Apéndice D.

3.2.2. Escena Simulada

La *Escena Simulada* contiene toda la información de la simulación interna mantenida por el agente mediante dos clases principales: `LevelScene.java` y `SimulatedScene.java`. La primera es una versión modificada (y reducida) de la provista por el *motor* de la competencia, mientras la segunda fue implementada especialmente para proveer a *Luigi* con toda la información requerida para realizar la búsqueda del camino óptimo.

En `LevelScene.java` se mantiene parte de la información para la representación del nivel, que incluye:

- La lista de *Sprites* de los enemigos vivos en la escena.
- Una referencia al objeto o instancia que representa a *Luigi*.
- Una referencia a la matriz con la información de la geometría del nivel.

Además, también contiene los métodos necesarios para calcular las interacciones entre los distintos elementos de la escena. De `LevelScene.java` fueron eliminados todos los métodos originales del *motor* que eran inútiles para la simulación física, como aquellos encargados de la carga de imágenes y artes, el dibujado de las mismas en la pantalla de juego, entre otros. En el Apéndice E se encuentra la lista de las clases respectivas, así como una breve explicación de su función.

Por su parte, la clase `SimulatedScene.java`, que hereda de la clase anterior, incluye atributos y métodos adicionales que permiten la comunicación de la información del nivel con el *Simulador*, la actualización de la simulación interna con respecto a los cambios en la simulación del *motor* de la competencia, información adicional para la búsqueda, entre otros. Se incluye además la implementación de las interfaces `Clonable` y `Comparable` de *Java™*; la primera para duplicar la escena en el momento en que se esté realizando la búsqueda y se quieran simular las siguientes escenas producto de las posibles acciones ejecutadas por *Luigi*; y la segunda para comparar dos escenas según su **Función de Selección** (Sección 3.6).

Cada *Escena Simulada* guarda varios atributos importantes para conocer el estado de la simulación. En el Apéndice F se puede encontrar una descripción detallada de cada uno de ellos.

3.3. Reconstrucción

A partir de la observación recibida del *motor* de la competencia (explicada con detalle en la Sección 2.2.3), se reconstruye la escena en dos pasos: **La reconstrucción de la geometría**, y **la reconstrucción de los enemigos presentes**. Cada paso es realizado por un método estático de la clase `Rebuilder.java`.

1. Reconstrucción de la geometría del nivel:

Este paso es llevado a cabo por el método estático `setLevelScene`, que recibe dos parámetros: la *Escena Simulada* a reconstruir, y la matriz de datos de tamaño 22x22 enviada en la observación provista por el *motor* de la competencia. Básicamente, en este paso se recibe la matriz y se procesa para integrarla con la información del nivel mantenida en la *Escena Simulada* del *Simulador*. Durante el proceso se revisan las últimas 3 columnas de la representación del mismo en la matriz de observación, para ver si existe algún bloque en ellas que represente el suelo. Si no se encuentra ningún bloque que represente el suelo en una columna, se asume que hay un abismo allí y se establece, a priori, que el abismo tiene 3 casillas o celdas de ancho (por ser el máximo ancho de un abismo en el juego).

2. Reconstrucción de los enemigos presentes:

Este paso es llevado a cabo por el método estático `setEnemies`, que recibe dos parámetros: la *Escena Simulada* donde se van a reconstruir los enemigos, y el arreglo de enemigos proveniente de la observación recibida del *motor* de la competencia. La función de dicho método es recibir el arreglo con la información de los enemigos en el juego, e integrarlo a la lista de enemigos que es mantenida en la *Escena Simulada* del *Simulador*.

El proceso de reconstrucción conlleva ciertas tareas, condiciones y estrategias que son detalladas a profundidad en el Apéndice G.

3.4. Algoritmo de Búsqueda (A*)

Como se mencionó en las Secciones 2.5 y 2.5.1, A^* es un algoritmo de búsqueda informada en amplitud, por lo que saca ventaja de las características del problema dado para poder estimar lo “prometedor” que puede ser un nodo o estado referente a las probabilidades de que la solución óptima del problema lo incluya.

En el caso de los videojuegos, como el tratado en este trabajo, los algoritmos de

búsqueda se utilizan normalmente para conseguir caminos entre dos posiciones o puntos de referencia dentro del mapa o ambiente de juego. Sin embargo, para que un agente se mueva a través de cierto escenario o ambiente de forma inteligente y sin tener una ruta pre-establecida, es necesario simplificar el espacio de búsqueda en un grupo de regiones o nodos, sobre los cuales el agente puede moverse. De esta forma, en vez de moverse entre dos puntos cualesquiera del ambiente, el agente está limitado a puntos o zonas discretas del mismo. Dichos puntos o zonas representan el grafo o espacio de búsqueda donde se aplica el algoritmo para conseguir caminos.

A continuación se explica cómo el problema que tiene *Luigi* para llegar a la meta de un nivel que desconoce se puede representar como un problema de búsqueda de caminos que puede ser resuelto eficientemente con el algoritmo A^* .

3.5. El Espacio de Búsqueda

Tal como se explicó en la Sección 2.5, lo primero es definir los aspectos principales del problema: ¿Cuál será el espacio sobre el que *Luigi* se moverá en el nivel?, ¿Qué representa cada una de las regiones o nodos del grafo?, ¿Cómo están conectados estos nodos?.

La primera pregunta es la más complicada de explicar, pues en el videojuego propuesto en la Competencia (Sección 2.2), los niveles de juego son generados aleatoriamente y son totalmente desconocidos para los agentes. Esto implica que no se puede generar un grafo del nivel antes de jugarlo y por lo tanto su conjunto de nodos y conexiones (incluyendo los costos) se desconocen. En este caso, el grafo se va construyendo a medida que se busca la solución del problema.

Para construir un grafo simplemente se necesita saber cómo son realizadas las conexiones entre sus nodos, ésto es: para cada nodo es necesario saber cuáles son sus nodos adyacentes. Este cálculo será realizado en una función que recibirá como parámetro un nodo cualquiera (en el caso de *Luigi*, una posición cualquiera) y devolverá todos los posibles nodos adyacentes. Llamaremos a esta función la **Función de Sucesores**, y será explicada

más adelante.

Se puede ver que con la **Función de Sucesores** es posible construir un grafo utilizando únicamente el nodo inicial del problema, aplicándole la función a él, a sus sucesores, a los sucesores de sus sucesores, y así hasta tener todos los nodos del grafo. En el caso de estar realizando una búsqueda informada, se puede construir únicamente la parte del grafo que interesa para poder llegar a la solución, aplicándole la función de sucesores sólo a los nodos que parezcan “prometedores” para encontrar la meta.

Con todo lo explicado se puede pensar intuitivamente que los nodos del problema son posiciones de *Luigi* en el nivel, pero no se puede saber de forma sencilla dónde se encuentra *Luigi* sin tomar en cuenta las posibles interacciones que puede tener con su ambiente, es decir, con la geometría del nivel y con los enemigos que lo acompañan. Esto hace ver que es necesaria más información que simplemente la posición del agente.

3.5.1. Escenas Simuladas como nodos del problema

Se explicó en la Sección 3.2.2 lo que es una *Escena Simulada*. Ahora se explicará cómo usar un conjunto de ellas como nodos del grafo del problema a resolver.

Como se mencionó anteriormente, utilizar únicamente la posición de *Luigi* como nodos del grafo no es una opción viable, pues *Luigi* se mueve en un mundo con una geometría y unos enemigos definidos, pero sobre la cual no se ha establecido un grafo de movimiento. Por lo tanto es necesario saber que posiciones puede ocupar el agente mientras se va buscando la solución.

Se puede ver entonces que se requiere de una representación del mundo que sea interna al agente, es decir, una *simulación* del mundo en el que se juega, sobre la cual el agente pueda estudiar las posiciones que son válidas e inválidas, y los costos de llegar a ellas. Es por esto que se utilizan *Escenas Simuladas* como nodos del grafo, pues éstas guardan toda la información necesaria para saber todas las posiciones en las que *Luigi* puede estar en el nivel que se está jugando. En la *simulación* no se podrá imaginar que el agente se encuentra atravesando algún bloque que no pueda ser traspasado, o cayendo en algún

punto más bajo que el suelo del nivel.

Las últimas preguntas que faltan por responder son: ¿Cómo están conectadas estas *Escenas Simuladas*?, y ¿Qué se necesita para poder estimar las próximas posiciones de *Luigi* dada una *Escena Simulada* cualquiera?

3.5.2. Conexiones y la Función de Sucesores

En el juego original **Super Mario Bros.**, el personaje principal se mueve a través del nivel gracias a las acciones que el jugador le indica usando un control remoto. Con dicho control, el jugador puede dar 5 tipos de órdenes a *Mario* y realizar los movimientos deseados. Usando este mismo concepto se puede ver que la relación entre una posición de *Mario* con respecto a su posición anterior está establecida en la última acción que se tomó para llegar hasta allí.

Desde la simulación implementada se pueden enviar las mismas órdenes en formas de acciones de control remoto al agente. De esta forma se puede generar para cada *Escena Simulada* todas las posibles escenas siguientes que resultarían de aplicar cada una de las posibles acciones de control a *Luigi*.

En la implementación realizada del algoritmo A^* , cada *Escena Simulada* guarda los detalles de la conexión con la cual está relacionada al resto de los nodos del grafo de búsqueda, esto quiere decir que, además de guardar una simulación del mundo, también mantiene en memoria la acción que se ejecutó para llegar hasta ese punto y el costo acumulado que se ha gestado en la búsqueda.

Ahora podemos ver que el único cálculo que necesita hacer la **Función de Sucesores** es saber que acciones se pueden realizar en un ambiente dado. Por supuesto, en la vida real se pueden presionar todos los botones del control en cualquier momento, pero en ciertas circunstancias algunas acciones no realizan ningún movimiento válido del juego y no son tomadas en cuenta, por ejemplo cuando *Luigi* está cayendo en un abismo, intentar saltar es inválido pues para saltar es necesario estar primero tocando suelo.

A nivel del algoritmo de búsqueda, es importante que la **Función de Sucesores** sepa eliminar cualquier acción que no vaya a producir un movimiento válido, para así no perder tiempo simulando una escena que no va a variar la posición del agente, y no agregar nodos inútiles que harían más lenta la búsqueda de caminos. Se puede ver que mientras más acciones posibles reporte la **Función de Sucesores**, más amplio será el espacio de búsqueda. Un espacio de búsqueda muy amplio puede significar más tiempo utilizado en buscar los caminos, mientras que uno muy reducido puede estar obviando caminos importantes para conseguir la solución.

3.6. Consiguiendo la solución

Sabiendo cómo se crea el espacio de búsqueda sobre el cual se mueve *Luigi*, se necesita saber cuál será el procedimiento para buscar la solución. Existen varias preguntas a resolver: ¿Cuál es el costo de realizar una acción?, ¿Qué determina que una *Escena Simulada* sea más “prometedora” que otra?, y lo más importante ¿Cuál es el objetivo al que *Luigi* quiere llegar?. Se empezará la explicación por ésta última pregunta y luego se responderán las demás.

3.6.1. El Objetivo

Debido a que los niveles de juego son generados aleatoriamente y son invisibles para el agente hasta recorrerlos, el objetivo de *Luigi* no puede ser llegar a la meta del mundo, pues es desconocida. El agente sólo tiene conocimiento del espacio que podría ver un ser humano y por lo tanto tiene que tener una meta más sencilla que la de recorrer todo el nivel hasta el final sin equivocarse.

Al pensar cuál es el objetivo de un ser humano al jugar un nivel que desconoce, se establece que la prioridad es avanzar “hacia el lado derecho de la pantalla”, aunque en realidad *Mario* esté siempre en el centro de la cámara. Utilizando el concepto anterior se puede plantear un objetivo simplificado para el agente: Lograr llegar al lado derecho de la pantalla en la simulación, porque después de ese límite se desconoce lo que puede venir.

Esto muestra que el espacio de búsqueda tiene más de un solo nodo objetivo, y que para que un nodo sea meta en la búsqueda, sólo hace falta cumplir la condición de estar a una distancia fija, y ya precalculada, de la posición origen de la búsqueda.

3.6.2. Costo de realizar acciones

Una vez establecido hasta donde se quiere llegar, sólo falta establecer el costo de las acciones para poder conseguir el camino “óptimo” o menos costoso.

Una primera aproximación podría ser tomar la distancia recorrida en una acción, y para estimar la promesa de cierto nodo se calcula la distancia en el eje X hacia el límite derecho de la pantalla (el objetivo de la búsqueda), ignorando cualquier tipo de obstáculos o enemigos que puedan ser encontrados. En esta aproximación, una escena que tenga una distancia X estimada hacia el borde derecho de la pantalla, será preferida sobre cualquier otra cuya distancia estimada sea mayor.

Tras conocer la física de los movimientos de *Luigi*, este enfoque se vuelve una forma inocente de aproximar las soluciones, debido al fallo de no tomar en cuenta la velocidad que lleva el agente en la *Escena Simulada*, y las aceleraciones o desaceleraciones que sean necesarias para llegar al objetivo.

Para ilustrar una situación en la que el enfoque de utilizar “distancias” fallaría, se propone el siguiente escenario donde existen dos *Escenas Simuladas* que van a ser comparadas por el algoritmo de búsqueda A^* . Las características de ambas escenas son:

- **Escena 1:** Se encuentra a una distancia de 50px del borde derecho de la pantalla. *Luigi* corre a máxima velocidad en sentido contrario, es decir, hacia la izquierda.
- **Escena 2:** Se encuentra a una distancia de 55px del borde derecho de la pantalla. *Luigi* corre en la dirección correcta (hacia la derecha).

Utilizando la aproximación propuesta, se elegiría a la **Escena 1** para ser evaluada, sin tomar en cuenta la velocidad que esta llevando el agente en ese momento, el tiempo y la distancia que le va a costar desacelerar y cambiar de dirección. Se puede ver que en este caso, la opción correcta sería la **Escena 2**.

Tomando en cuenta estas fallas se ideó una aproximación más precisa: usar como costo la cantidad de ticks de juego que han transcurrido desde el origen de la búsqueda hasta la *Escena Simulada* a evaluar. Se tiene entonces que cada acción costaría un tick de juego. La función heurística tiene que ser redefinida en estos términos.

3.6.3. Función heurística

Como se mencionó anteriormente, la heurística estudiada se precisó en función de los ticks de juego, siendo, en un principio, la cantidad de ticks estimada en la que *Luigi* alcanza el borde derecho de la pantalla corriendo a máxima aceleración hacia la derecha y llevando la velocidad actual de la *Escena Simulada*, suponiendo de forma optimista que corre hacia la derecha sin obstáculos y sin enemigos.

Se puede ver que esta aproximación toma en cuenta la velocidad que lleva *Luigi* en la *Escena Simulada*, lo cual resuelve el problema planteado anteriormente: La **Escena 1** sería descartada en favor de la **Escena 2** debido a que la heurística tomará en cuenta los ticks de juego que costará desacelerar la velocidad llevada hacia la izquierda para luego poder empezar a correr hacia la derecha.

No obstante, tras un estudio realizado, el valor semántico de la heurística pasó de ser el mencionado anteriormente para transformarse en: **La cantidad de ticks necesarios para llegar a una distancia fija de 100.000 pixeles desde el origen del nivel, suponiendo que en 1000 ticks se llegó a una distancia x_{1000} dada, y se corre a velocidad constante y máxima aceleración sin encontrar ningún obstáculo en el camino.** La fórmula cerrada de la heurística desarrollada es:

$$h = \frac{100000 - (8,090909090909092 * V + 9630,535537190084)}{10,909084} \quad (3.1)$$

Donde V representa la velocidad en el eje X que lleva *Luigi* en la *Escena Simulada*.

Para realizar el cálculo de esta heurística fue requerido revisar la física del agente, y estudiar cuáles son las aceleraciones en el eje X realizadas por las distintas acciones posibles. También es necesario ver la fricción que ejerce el suelo y el aire sobre *Luigi*.

Los movimientos horizontales están dados por los botones de izquierda y derecha, ambos movimientos imprimen la misma aceleración al agente pero en sentidos opuestos. Sin embargo, cuando el botón de “velocidad” (botón **B** en el control clásico de **Nintendo™**) es apretado al mismo tiempo que cualquier movimiento horizontal, la aceleración dada es mayor. Por otra parte, la fricción siempre es igual tanto en el aire como en el suelo.

A continuación se presenta el pseudo-código de una primera aproximación iterativa para calcular la heurística estudiada para una *Escena Simulada* dada.

Pseudo-código de cálculo iterativo de heurística

```

1  def iter_heuristica(escena):
2      ticks = 0
3      x = 0
4      vx = velocidad_luigi(escena)
5
6      while x < DISTANCIA_BORDE_DERECHO:
7          vx = vx + MAX_ACELERACION
8          vx = vx * FRICCION
9          x = x + vx
10         ticks = ticks + 1
11
12     return ticks

```

El problema de esta aproximación es su carácter iterativo, para las primeras escenas exploradas en la búsqueda se realizan aproximadamente 25-30 iteraciones, lo cual representa un cálculo considerable a realizar en muchas de las *Escenas Simuladas* que estudiaría el algoritmo de búsqueda. En aras de conseguir una versión que requiera menos cómputo, se buscó una fórmula cerrada para representar la sumatoria efectuada en la iteración. Sin embargo, conseguir la fórmula cerrada tuvo como consecuencia modificar semánticamente el significado de la heurística, más no la idea de trasfondo.

Se mostrará paso a paso la construcción de una nueva forma de la función heurística a partir del Algoritmo descrito anteriormente. Debido a que el valor de x funciona como la condición de parada de la iteración, queremos calcular cuánto valdrá x en la iteración n . Además podemos ver que x funciona como un acumulador de la suma de los valores

actualizados de vx , que es la única variable extraída de la *Escena Simulada*. Desglosando la iteración del algoritmo tenemos:

Sea $A = \text{Máxima Aceleración}$; $F = \text{Fricción}$; $\mathbb{W} = \text{Velocidad inicial}$

$$tick_0 : vx_0 \leftarrow \mathbb{W}$$

$$tick_1 : vx_1 \leftarrow (vx_0 + A) * F = (\mathbb{W} + A) * F$$

$$tick_2 : vx_2 \leftarrow (vx_1 + A) * F = ((\mathbb{W} + A) * F + A) * F$$

$$vx_2 \leftarrow \mathbb{W} * F^2 + A * F^2 + A * F$$

$$tick_3 : vx_3 \leftarrow (vx_2 + A) * F = ((\mathbb{W} * F^2 + A * F^2 + A * F) + A) * F$$

$$vx_3 \leftarrow \mathbb{W} * F^3 + A * F^3 + A * F^2 + A * F$$

⋮

$$tick_n : vx_n \leftarrow \mathbb{W} * F^n + A * F^n + \dots + A * F$$

$$vx_n \leftarrow \mathbb{W} * F^n + A \sum_{k=1}^n F^k$$

Por otra parte, se tiene que x es una acumulación de todos los valores de vx , es decir:

$$x_n = \sum_{k=1}^n vx_k \quad (3.2)$$

$$x_n = \sum_{k=1}^n (\mathbb{W} * F^k + A \sum_{j=1}^k F^j) \quad (3.3)$$

$$x_n = \sum_{k=1}^n \mathbb{W} * F^k + A \sum_{k=1}^n \sum_{j=1}^k F^j \quad (3.4)$$

Tras una serie de pasos se puede demostrar que a partir de la Ecuación 3.4 se llega a:

$$x_n = \left(\frac{F^{n+1} - 1}{F - 1} - 1 \right) \mathbb{W} + \frac{A * F^{n+2} - A * F}{(F - 1)^2} - \frac{A * F}{F - 1} - \frac{A * n}{F - 1} - A * n \quad (3.5)$$

El procedimiento que se esperaría realizar sería igualar x_n a la distancia entre la posición de la *Escena Simulada* y el borde derecho de la pantalla, y luego despejar n (la cantidad

de ticks) en la ecuación. Sin embargo hay que notar que la variable n se encuentra siendo usada como exponente y como factor, para despejar este tipo de ecuaciones se necesita usar técnicas de solución de ecuaciones diferenciales, y el proceso resolutivo sería una iteración, al igual que lo que se está tratando de evitar.

Tras realizar un estudio sobre la velocidad de *Luigi* se pudo observar que crece exponencialmente, alcanzando una asíntota de 10.909084 siempre en menos de 100 ticks si corre con máxima aceleración hacia la derecha sin conseguir obstáculos. Si igualamos $n = 1000$ obtendremos en x_n la distancia recorrida en 1000 ticks de juego, y podemos asumir que ya transcurridos 1000 ticks de juego, asumiendo no haber conseguido obstáculos en el camino, la velocidad de *Luigi* está al máximo y es **constante**.

Luego, si se propone como una meta única alcanzar un punto en el eje X a una distancia establecida de 100.000 pixeles, la cual es mayor al tamaño de cualquier nivel de juego, se puede utilizar la ecuación clásica de la velocidad (Ecuación 3.6) para calcular el tiempo que toma llegar a dicha distancia desde el punto x_{1000} previamente calculado.

$$V = \frac{D}{T} \quad (3.6)$$

La nueva heurística se calcula entonces como: **La cantidad de ticks necesarios para llegar a una distancia fija de 100.000 pixeles desde el origen del nivel, asumiendo que en 1000 ticks se llego a una distancia x_{1000} dada, y se corre a velocidad constante y máxima aceleración sin encontrar ningún obstáculo en el camino.** Se utiliza la siguiente fórmula:

$$h = \frac{100000 - x_{1000}}{10,909084} \quad (3.7)$$

Esta heurística trae la ventaja de, al establecer n en un valor fijo, poder observar en la Ecuación 3.5 que lo único variable es la velocidad inicial V de *Luigi* en la *Escena Simulada*, y todo lo demás es un valor constante que puede ser precalculado, dejando la ecuación de esta forma:

$$x_n = C1 * V + C2 \quad (3.8)$$

En particular, para $n = 1000$:

$$C1 = 8,090909090909092$$

$$C2 = 9630,535537190084$$

Finalmente, substituyendo la Ecuación 3.8 en la Ecuación 3.7, se tiene que el valor de la heurística para cualquier *Escena Simulada* sólo depende de la velocidad inicial V de *Luigi*, y es la siguiente:

$$h = \frac{100000 - (C1 * V + C2)}{10,909084} \quad (3.9)$$

Por último, es necesario ver que la heurística propuesta es **admisibile**, es decir, que nunca sobreestima el costo verdadero de alcanzar el objetivo [15, 110–115]. Para esto sólo se requiere un análisis de lo que la heurística representa: ¿Cuántos ticks de juego hacen falta para llegar a una meta ficticia puesta a 100.000 píxeles de distancia desde el origen del nivel?. Esta distancia se escoge bajo el conocimiento de que 100.000 píxeles es un número excesivamente mayor que el tamaño de cualquier nivel de juego, y además que no puede ser alcanzada en sólo 1000 ticks¹.

En este escenario, una heurística que no “sobreestima” el costo verdadero de alcanzar el objetivo es aquella que no estima más ticks de los necesarios para llegar desde un nodo dado hasta la meta ficticia. Se puede ver que la heurística propuesta trabaja sobre la hipótesis **optimista** de que *Luigi* va a correr a máxima aceleración y máxima velocidad hacia la meta, y nunca va a desacelerar por algún tipo de obstáculo en su camino. Ésto quiere decir que la heurística propuesta va a dar el menor costo posible para poder llegar a la meta, partiendo con una velocidad desde un punto dado. Se puede ver entonces que

¹La máxima posible distancia a alcanzar en sólo 1000 ticks es: 9707.39, calculada usando la Ecuación 3.8

la heurística propuesta cumple con el requerimiento de ser una heurística **admisible**.

Con todo esto se puede ver que está definida una búsqueda de caminos sobre un nivel ideal, en el que se pueden simular todas las posibles posiciones del agente y se puede evaluar el valor de la **Función de Selección** de cualquier *Escena Simulada*, en función del tiempo que le toma alcanzar la meta en vez de la distancia a la cual se encuentra.

Todavía falta un problema más por resolver: el choque con los enemigos. Ahora se revisará como hace el algoritmo para evitar a los enemigos del nivel.

3.6.4. Las Penalizaciones

Para que *Luigi* pueda conseguir caminos que eviten ser herido por los enemigos, o caer en los abismos que se encuentran en la escena, se utilizan penalizaciones sobre las *Escenas Simuladas* que incurran en dichos errores.

Estas penalizaciones son dadas por una función que recibe una *Escena Simulada* y devuelve el costo extra de la misma según las fallas cometidas. Existen dos tipos de penalizaciones tomadas en cuenta por dicha función:

1. **Por estar cayendo en un abismo:** Cuando en una *Escena Simulada* el agente está cayendo en un abismo, la función devuelve una penalización de magnitud 150.
2. **Por haber sido herido por un enemigo:** Cuando el agente es golpeado por un enemigo y dicho enemigo le causa daño, la función devuelve una penalización que depende del estado de *Luigi* en ese momento: Si estaba en estado de *Fuego* y pasó a ser *Grande*, el costo adicional será 150; Si estaba en estado *Grande* y pasó a ser *Pequeño*, el costo será 500; Por último, si estaba en estado *Pequeño* el costo será 1000.

Las penalizaciones son añadidas al costo acumulado de llegar hasta la *Escena Simulada* con fallos, lo que quiere decir que cualquier escena tendrá como costo la suma del número de acciones para llegar desde el origen hasta ella, más las penalizaciones de cualquiera de los nodos que se encuentren en el camino recorrido.

3.6.5. La Función de Selección

La función de selección (f) es establecida como la sumatoria de los últimos tres valores explicados: La cantidad de ticks usados para llegar al nodo a evaluar desde el origen (g); La cantidad estimada de ticks para llegar a la meta establecida en la **Función Heurística** (h); y las penalizaciones obtenidas (p).

Un último elemento es agregado para facilitar la tendencia a elegir nodos que se encuentren lejos del origen. Debido a que el costo de realizar cualquier acción es sólo un tick, se utiliza un número real entre 0 y 1 como Factor de Parcialidad en el costo de llegar a un nodo (b). Mientras b sea más pequeño, habrá mayor parcialidad por escoger nodos que tengan un costo g alto:

$$f = (g * b) + h + p \quad (3.10)$$

3.7. Estrategias para cerrar nodos

Para un algoritmo de búsqueda como el presentado en este proyecto, poder saber cuáles nodos han sido previamente evaluados es una parte muy importante del desarrollo pues permite descartar aquellas posiciones o nodos que lleguen a los mismos valores por distintos caminos. De esta forma el algoritmo puede sólo tomar en cuenta un único camino (el mejor) para llegar a cada posible configuración. Cuando el algoritmo consigue un nodo que ya ha sido previamente evaluado, se dice que éste ya ha sido “visitado” o “cerrado” [15, 110–115].

En el caso del algoritmo de búsqueda de *Luigi*, realizar esta tarea no es sencilla debido a que cada escena puede diferir de todas las otras en detalles muy pequeños, por ejemplo un sólo enemigo en una posición diferente. Para facilitar esta tarea se ideó una primera simplificación del problema en donde se declarará a una *Escena Simulada* como “cerrada” si y sólo si la posición de *Luigi* es igual a la posición de alguna otra escena que ya haya sido evaluada por el algoritmo, y cuya diferencia de ticks, o puesto de otra forma, la diferencia de profundidad en el árbol de búsqueda, sea menor a cierto número determinado.

No obstante, pedir como condición la igualdad de posiciones de *Luigi* para determinar que una escena está “repetida” no trae un beneficio real sobre no utilizar la lista de nodos cerrados, debido a que los valores de las posibles posiciones del agente son números reales y pueden variar por una diferencia muy pequeña. Por este motivo se decidió utilizar un *delta* sobre la diferencia entre las posiciones en el eje X y uno sobre la diferencia de posiciones en el eje Y entre dos escenas que estén siendo comparadas. De esta forma se establece que una *Escena Simulada* ya se encuentra “cerrada” si existe alguna otra escena en la lista de nodos cerrados que tenga una posición de *Luigi* de la cual difiera en menos de los *deltas* establecidos.

Cuando se encuentra una *Escena Simulada* previamente cerrada, se puede tomar la decisión de descartarla y continuar con el resto, o se puede reinsertar en la búsqueda pero colocándole alguna penalidad. La estrategia de descartar las escenas que se catalogan como “cerradas” trae beneficios en la velocidad de respuesta del agente, al tener que expandir menos nodos en la búsqueda, pero también trae posibles problemas al momento de encontrar la solución: Si el *delta* de espacio utilizado para considerar a una *Escena Simulada* como “cerrada” es muy amplio, se corre el riesgo de dejar de evaluar posiciones importantes de *Luigi*, y un error más grave puede ser dejar al algoritmo sin nodos para evaluar.

Por otra parte, mientras más pequeño se establezcan los *delta* utilizados, más nodos tendrá que evaluar el algoritmo de búsqueda. Se puede ver entonces que para llevar a cabo esta estrategia de descartar los nodos cerrados, es necesario que el algoritmo haga la búsqueda y evalúe los nodos rápidamente. En el Capítulo 5 se muestra con detalle las ventajas obtenidas en el juego al utilizar la estrategia expuesta.

3.8. Tiempos de Respuesta

El mayor problema que puede tener un agente que basa su toma de decisiones en la búsqueda de caminos óptimos es el tiempo que ésta búsqueda pueda durar. En el caso de un juego como “Super Mario Bros”, el espacio de búsqueda es bastante amplio, y se espera

que las reacciones del juego sean lo suficientemente rápidas como para que el mismo se vea fluido.

En vista de estas condiciones, la competencia “Mario AI Competition” exige entre sus reglas (detalladas en la Sección 2.2.4) que cualquier agente participante no puede tomar, en promedio, más de 40 milisegundos por tick de juego para devolver la acción a ejecutar al *motor* de la competencia.

A continuación se presenta la estrategia implementada en *Luigi* para cumplir esta restricción, y posteriormente se hablará de la técnica utilizada para aumentar la velocidad de la búsqueda y poder explorar un mayor espacio de posibilidades en el tiempo establecido.

3.8.1. Ajustar la búsqueda a un tiempo máximo

Para poder asegurar que *Luigi* no tome más de 40 milisegundos en promedio para responder al *motor* de la competencia con una acción a realizar, el algoritmo de búsqueda implementado debe estar preparado para devolver una acción aún cuando no haya podido conseguir la meta de la búsqueda para una *Escena Simulada* dada.

El algoritmo de búsqueda realiza esta tarea contando los milisegundos que han transcurrido desde que se llamó al método `getAction(Environment observation)` (detallado en el Apéndice A) en cada una de sus iteraciones y llevando la cuenta de cuantos milisegundos han sido utilizados, en promedio, en todas las acciones realizadas. Ésto se realiza llevando un **Acumulador de Tiempo Disponible** en el que se acumula en cada tick la diferencia entre el máximo tiempo de restricción dado y el tiempo tomado en devolver una acción al *motor* de la competencia.

Por otra parte, el algoritmo utiliza un apuntador que señala al nodo más lejano del origen encontrado en la búsqueda, y lo actualiza cada vez que se visite un nuevo nodo que se encuentre más lejos. Cada nodo guarda información sobre si el camino que llega hasta él ha sido penalizado, realizando ésto se garantiza el correcto funcionamiento del algoritmo de búsqueda en los casos en los que las restricciones de tiempo no le permitan conseguir una solución a las condiciones de una *Escena Simulada* dada, y se garantiza el

beneficio a aquellos movimientos que alejen por completo a *Luigi* del peligro.

3.8.2. Aumentar la velocidad de búsqueda

Se puede ver que el mayor problema presentado por la restricción de tiempo impuesta por la competencia es que mientras más lenta es la búsqueda, habrá más ticks de juego en los que el agente devolverá al *motor* de la competencia una acción que guía hacia un objetivo sub-óptimo, y por lo tanto no podrá lograr su mejor desempeño en los niveles que juegue. Es por esto que es de vital importancia lograr aumentar lo más posible la velocidad de la búsqueda para tratar de reducir los escenarios en los cuales el algoritmo no logre encontrar una solución óptima.

Para tratar de conseguir el objetivo de aumentar la velocidad de búsqueda, se utilizó una herramienta de *profiling* para realizar un análisis dinámico del agente mientras se ejecutaba, y así conseguir cuáles métodos o procedimientos demoraban más tiempo en ejecución al momento de buscar las soluciones. Los resultados, que se pueden ver con detalle en el Apéndice J, mostraron que el tiempo de búsqueda era gastado principalmente en los métodos `tick()` y `getBlock()`, ambos pertenecientes al proceso de *Simulación* (Sección 3.2), donde cada método es llamado en la **Función de Sucesores**, cuando se crean las nuevas *Escenas Simuladas*.

Conociendo esta información se ideó realizar una evaluación retrasada de las *Escenas Simuladas*, eliminando la simulación de la acción correspondiente a una escena al ser insertada en la lista de nodos abiertos (explicación de las estructuras de datos en el Apéndice I), y postergándola al momento en que el algoritmo de búsqueda la saque de la lista de nodos abiertos e intente generar sus escenas sucesoras. Con esto se logra reducir la cantidad de simulaciones desde el número de nodos expandidos al número de nodos visitados por el algoritmo.

El problema de este enfoque está en que el cálculo de la heurística de una *Escena Simulada* viene dada por la velocidad de *Luigi* en dicha escena. Ese valor es desconocido antes de realizar la simulación y por lo tanto no se puede calcular la heurística de una

escena sin antes haberla simulado. Al no tener el valor de las heurísticas de las escenas generadas, el algoritmo es incapaz de decidir cuál escena elegir para expandir en el espacio de búsqueda.

Este último problema es resuelto creando un método de “estimación” de velocidad y posición para las *Escenas Simuladas*, usando la misma filosofía utilizada para el cálculo de la heurística de una escena. Como el movimiento físico de *Luigi* es determinístico y podemos calcularlo sin necesidad de simular el resto de los elementos de la escena, se puede establecer, para cualquier escena, cuál será la posición y velocidad estimada de *Luigi* para la acción correspondiente.

Usando cálculos similares a los encontrados en el Algoritmo para el cálculo de la heurística, se obtiene la velocidad estimada después de realizar la acción correspondiente, y se puede calcular la heurística para dicha escena. Sin embargo hay que tener cuidado con aquellas escenas en donde la posición y velocidad estimadas no sean iguales a las obtenidas luego de realizar la simulación, debido a haber colisionado horizontalmente con algún bloque u objeto.

Además hay que tomar en cuenta que, habiendo postergado la simulación, también es necesario postergar el cálculo de penalizaciones de la *Escena Simulada*, pues esta depende de la posición de los enemigos luego de la simulación, o de la posición en el eje Y de *Luigi*. Sin la penalización en el costo de la *Escena Simulada*, la función de selección utilizada por el algoritmo de búsqueda funcionaría incorrectamente.

Esto hace ver que aún estimando correctamente la posición del agente, dada cualquier acción, se necesita un paso extra para revisar que la estimación haya acertado la posición simulada posteriormente, y que luego de haber simulado no se hayan añadido nuevos costos por penalizaciones.

Para tomar en cuenta todos los detalles mencionados, el algoritmo de búsqueda actúa de la siguiente forma: Al retirar una escena de la lista de nodos abiertos, revisa si ya ha sido simulada; de serlo, continúa con la evaluación normal del algoritmo A^* , pero si por el contrario todavía no ha sido simulada, realiza los siguientes pasos en orden:

1. Simula la escena.
2. Revisa si *Luigi* muere en esa escena, de ser así, la descarta y continúa con alguna otra.
3. Revisa si *Luigi* recibió algún daño o está dentro de algún abismo en la escena. Si es así, establece como *verdadero* una variable booleana a la que llamaremos “repensar” y que será usada mas adelante.
4. Revisa si la diferencia entre la distancia en el eje X de *Luigi* y la estimación realizada previamente es mayor a un delta específico. De ser así, establece “repensar” en *verdadero*, y vuelve a calcular la heurística para la *Escena Simulada*.
5. Si “repensar” es *verdadero*, vuelve a agregar la escena a la lista de nodos abiertos y empieza el ciclo de nuevo, de lo contrario, continúa con la evaluación normal del algoritmo A^* .

En el Apéndice C se encuentra una versión simplificada en pseudo-código del algoritmo implementado.

Capítulo 4

Diseño e Implementación del Aprendizaje

Este capítulo describe el uso de algoritmos de aprendizaje de máquina para lograr que la implementación del agente expuesta en el capítulo anterior aprenda a satisfacer múltiples criterios de evaluación. Se explicará cuáles criterios de evaluación serán tomados en cuenta y cómo se redefinirá la búsqueda en *Luigi* para alcanzar las nuevas metas propuestas. Posteriormente se hablará de la implementación de un entrenamiento al cual será sometido *Luigi* para que, utilizando los algoritmos de aprendizaje de máquinas realizados, logre aprender qué estrategia o búsqueda aplicar en distintos escenarios y así optimizar la evaluación bajo la que está siendo entrenado.

4.1. Criterios de evaluación y Estrategias

De acuerdo a lo explicado previamente en el Capítulo 3, puede observarse que desarrollar un controlador basado en búsqueda de caminos óptimos permite resolver el problema de la competencia, y en los experimentos realizados en el Capítulo 5 se verá que se logra de forma satisfactoria y eficiente. Sin embargo, el enfoque descrito sólo toma en cuenta tratar de llegar lo más lejos posible de la manera más rápida, por como fué construida la **Función Heurística** para los nodos en la búsqueda. Cualquier otro objetivo que pueda existir en el juego es ignorado por dicha heurística. Bajo esta condición, si se quisiera conseguir optimizar algún otro objetivo en el juego, se necesitaría utilizar una *estrategia* diferente para la búsqueda de caminos de manera que *Luigi* intente conseguir la meta requerida.

Como ya ha sido explicado previamente, los posibles objetivos del juego son los siguientes: (1) Avanzar la mayor cantidad de pasos posibles hacia la derecha para tratar de

llegar a la meta del mundo, (2) ser herido la menor cantidad de veces, (3) llegar a la meta lo más rápido posible, (4) agarrar la mayor cantidad de monedas, (5) destruir la mayor cantidad de enemigos.

Los primeros 3 objetivos son cumplidos por la *estrategia* de búsqueda detallada en la Sección 3.4. Para conseguir nuevas formas de interacción con el juego, se realizaron las estrategias que optimizan los dos objetivos restantes. Una estrategia no es más que la composición de todos los puntos tratados en la Sección 3.6: un **Objetivo**, un **Costo de realizar acciones**, una **Función Heurística**, unas **Penalizaciones** y una **Función de Selección**.

Tanto para la *estrategia* de agarrar monedas, como para la de destruir enemigos, las penalizaciones y la función de selección son las mismas que las explicadas en el capítulo anterior. Sin embargo, el resto de los componentes varían: Debido a que ambas estrategias apuntan a conseguir un objetivo que se encuentra en pantalla (sea una moneda o un enemigo), sus funciones heurísticas fueron definidas bajo el concepto de distancia estimada hacia el objetivo, y los costos de realizar acciones fueron definidos en función a la distancia recorrida por cada acción. Por otra parte, los objetivos de las dos nuevas estrategias se reducen a conseguir la moneda más cercana, y destruir al enemigo más cercano. A continuación se muestra brevemente las características de las dos estrategias nuevas:

■ **Agarrar monedas:**

- *Objetivo:* Obtener la moneda más cercana al agente.
- *Costo de realizar una acción:* Distancia recorrida tras haber realizado dicha acción.
- *Función Heurística:* Estimación de la distancia euclideana entre la posición del agente y la posición de la moneda objetivo.

■ **Destruir enemigos:**

- *Objetivo:* Destruir al enemigo “aplastable” más cercano al agente.

- *Costo de realizar una acción*: Distancia recorrida tras haber realizado dicha acción.
- *Función Heurística*: Estimación de la distancia euclideana entre la posición del agente y un punto arbitrario fijado sobre el tope del enemigo objetivo, de manera que la búsqueda oriente al agente a tratar de llegarle por arriba para poder destruirlo.

Se puede observar que, a nivel de la implementación del agente, para las nuevas estrategias el **Espacio de Búsqueda** y las **Estrategias para cerrar nodos** sirven sin tener que ser modificados, y las *estrategias* terminan siendo sólo una abstracción de cómo conseguir la solución. Por ello, para aplicar diversas estrategias, el algoritmo de búsqueda puede trabajar exactamente como fue implementado y descrito en el capítulo anterior y lo único que habría que informarle es: ¿qué objetivo está persiguiendo? y ¿cómo evaluar a las escenas simuladas?. Para realizar esto se creó una clase abstracta llamada `Movement.java`, que representa la abstracción mencionada. El algoritmo de búsqueda recibe ahora una *estrategia* o *Movement* como parámetro, y orienta con ella la búsqueda. De ésta clase padre extienden las tres clases que representan las *estrategias* planteadas.

Para definir cuál *estrategia* va a ser asignada al algoritmo de búsqueda en un tick dado, se implementó un planificador de alto nivel que recibe la *simulación* llevada por *Luigi* y la observación dada por el *motor* de la competencia, y decide, según la información del ambiente, cuál es la mejor *estrategia* a utilizar.

4.2. Aprendiendo a seleccionar estrategias

Intuitivamente se puede ver que la selección de la estrategia a aplicar en cierto tick de juego dado es la parte más importante de toda la planificación y desempeño del agente. Si el agente decide utilizar la *estrategia* de agarrar monedas en un tick donde no se encuentran monedas en la escena, no va a tener ningún objetivo que alcanzar y no se moverá de su sitio. Igualmente sucede con los enemigos. Escoger de forma errada la estrategia a aplicar se verá reflejado de forma negativa en la evaluación que tenga el agente, al no conseguir las metas que se están buscando.

A continuación se explicará qué concepto es utilizado como *seleccionador de estrategias* dentro del agente, para ser el encargado de tomar las decisiones de las estrategias convenientes a aplicar en cualquier momento. Posteriormente se mostrará cómo se realiza la evaluación de un *seleccionador de estrategias* en un mundo determinado. Por último, se explicará cómo es usado el concepto de evolución y de aprendizaje por reforzamiento para lograr que el agente mejore su proceso de planificación a través de cierto entrenamiento.

4.2.1. Un seleccionador de estrategias

Como su nombre lo indica, un *seleccionador de estrategias* se encarga de examinar el ambiente en el que se encuentra el agente, y decidir que estrategia de movimiento le conviene utilizar, sea avanzar lo más rápido posible hacia la derecha, agarrar las monedas que encuentre en la escena que está viendo, o destruir a los enemigos que tenga alrededor. Puede verse que, simplificada, la tarea del seleccionador es recibir una entrada de valores referentes a lo que se está viendo en el ambiente que se juega, realizar un análisis sobre dicha información, y devolver una información adecuada.

Para realizar esta tarea y representar un *seleccionador de estrategias*, el trabajo fue basado en la *Neuroevolución* (Sección 2.6.3). Un seleccionador es una **Red Neuronal** que tiene una capa de receptores encargada de recibir la información del ambiente, y dos capas de perceptrones encargadas de hacer el análisis y devolver una evaluación, respectivamente. La capa de entrada (o capa de receptores) se encarga de recibir sólo una simplificación de la información del ambiente, debido a que los datos que representa *Escena Simulada* son complejos y no todos ellos son relevantes para la toma de decisiones de estrategias y el aprendizaje. La simplificación utilizada contempla únicamente los siguientes datos del ambiente:

- La posición en el eje X de *Luigi*, lo cual permite saber la distancia que ha sido recorrida hacia la derecha en el mundo.
- La velocidad que lleva *Luigi* en la escena.
- El tiempo restante que queda en el mundo.

- La distancia entre el agente y la moneda más cercana en escena. Si en la escena que se evalúa no existe ninguna moneda, esta distancia es establecida en el máximo número real existente en la máquina virtual de *Java*TM.
- El total de monedas en escena.
- La distancia entre el agente y el enemigo más cercano en escena. Análogamente a lo anterior, si no hay enemigos en escena, este valor pasa a ser el mayor número real.
- El total de enemigos en escena.
- El estado en el que se encuentra *Luigi*, siendo un número entre 0 y 2, donde 0 representa estar *Pequeño*, 1 representa estar *Grande* y 2 representa estar en *Fuego*.

La configuración utilizada para la red neuronal fue tomada del trabajo de Simon M. Lucas [22]: Se utiliza una neurona receptor para cada dato de entrada mencionado, lo que da un total de 8 neuronas. En la capa intermedia se colocan 20 perceptrones, y para finalizar, en la capa de salida se utilizan 3, una para cada estrategia posible. Al finalizar el análisis de la red se evalúan los valores de las tres neuronas de salida y se devuelve la estrategia que corresponda a la neurona que obtuvo el mayor valor. Aunque los experimentos y resultados de su trabajo no son comparables con el realizado, se trató de extrapolar la estrategia utilizada al aprendizaje de *Luigi*. Los resultados obtenidos son mostrados en el Capítulo 5.

4.2.2. La función de adaptación

Para evaluar el desempeño de un *seleccionador de estrategias* de *Luigi*, se necesita un valor numérico que sirva de puntuación o calificación de dicho seleccionador. La función que realiza el cálculo de dicha puntuación es llamada la **función de adaptación**, y recibe todos los datos de la evaluación del mundo una vez finalizado para luego combinarlos linealmente y producir una puntuación. Los datos de evaluación recibidos por parte del *motor* de la competencia son: Los enemigos destruidos (*ED*), las monedas agarradas (*MA*), el tiempo de duración del agente en el mundo (*TD*), la distancia recorrida (*DR*),

los golpes recibidos (GR) y por último el estado de finalización del mundo, que dice si *Luigi* logró completar el nivel (EL). La **función de adaptación** le da valores positivos a los objetivos logrados y valores negativos a los aspectos que perjudiquen al agente, como recibir golpes y no completar el mundo. Además de esto, las variables mencionadas son multiplicadas por ciertos pesos para llevar la puntuación a reforzar algunos objetivos más que otros. La Ecuación 4.1 muestra la **función de adaptación** utilizada, donde los P_i representan los pesos mencionados.

$$fda(S) = ED \times P_1 + MA \times P_2 + \frac{DR}{TD} \times P_3 + EL \times P_4 - GR \times P_5 \quad (4.1)$$

Se puede ver que los pesos P_i rigen la valoración dada por la **función de adaptación** al desempeño de un “seleccionador de estrategias”. Si se quisiera obviar la capacidad de los seleccionadores para agarrar monedas, se podría establecer $P_2 = 0$ y la evaluación no tomaría en cuenta el factor MA de monedas agarradas. Como se verá más adelante, la **función de adaptación** guiará el proceso de aprendizaje al valorar, tras la evaluación obtenida, qué tan buen desempeño tienen los seleccionadores aprendidos. Si se ajustan los pesos de la función para no tomar en cuenta ciertos objetivos del juego, se puede decidir qué es lo que se quiere que el agente aprenda a obtener. Experimentos que validan ésta suposición se encontrarán en el Capítulo 5.

4.2.3. Evolucionando al planificador

Una vez definido lo que es un *seleccionador de estrategias*, y siguiendo con lo explicado en la Sección 2.6.3, falta explicar el mecanismo utilizado para dirigir la búsqueda de seleccionadores que cumplan los objetivos planteados.

Se utilizó un algoritmo genético que utiliza como individuos o hipótesis los vectores de pesos entre las conexiones de las neuronas de la red neuronal previamente explicada. Las hipótesis son evolucionadas usando como operador genético la mutación, escogiendo las nuevas poblaciones basándose en su **función de adaptación**. Se puede ver que, al utilizar únicamente los pesos de las conexiones entre las neuronas, lo que se evoluciona

no es la topología de la red, sino la configuración en las conexiones para una red neuronal pre-establecida. El entrenamiento realizado con el algoritmo genético busca conseguir la mejor configuración de pesos para un sólo nivel de juego previamente establecido. Para evaluar la adaptación de un individuo de la población, se establece el vector de pesos que representa dicho individuo en el *seleccionador de estrategias* de *Luigi*, y se evalúa que tan buen resultado obtiene el agente, en promedio, jugando tres veces el nivel determinado. Idealmente se espera que los vectores de pesos de redes neuronales que van pasando de generación en generación sean cada vez mejores tomando decisiones, una vez sean establecidos en la red neuronal del seleccionador. A continuación se explica con más detalle los elementos claves del entrenamiento.

4.2.3.1. Generación de poblaciones

Para los miembros de la población inicial, los valores que conforman los vectores de pesos utilizados como individuos son tomados de una distribución Gaussiana que tiene media cero y desviación estándar igual a la raíz cuadrada de la cantidad de conexiones que llegan al nodo correspondiente. Para generar las generaciones siguientes a partir de una población dada, el algoritmo implementado selecciona al mejor 20 % de individuos de dicha población y los pasa a la siguiente población sin ser modificados. A partir de allí, el restante 80 % de la nueva población es generado utilizando el operador genético de mutación (que será explicado más adelante) sobre los individuos que ya formen parte de la nueva población.

4.2.3.2. Mutación de individuos

La mutación puede darse de tres formas distintas sobre el vector de pesos, cada forma de la siguiente manera: (1) mutar todos los pesos del vector (probabilidad 10 %), (2) mutar todos los pesos de conexiones que salen de una capa de la red seleccionada aleatoriamente (probabilidad 54 %), y (3) mutar únicamente un peso del vector (probabilidad 36 %).

Cada mutación de peso se realiza utilizando un número aleatorio tomado de la misma distribución empleada para generar la población inicial.

Capítulo 5

Experimentos y Resultados

Este capítulo se encuentra dividido en dos secciones principales: los experimentos y resultados obtenidos con el agente implementado para la “Mario AI Competition” (detallado en el Capítulo 3), y los experimentos y resultados del desarrollo realizado sobre dicho agente como aproximación para el aprendizaje de estrategias de búsqueda (detallado en el Capítulo 4).

Todos los experimentos fueron realizados en computadoras que tienen las especificaciones de hardware y software mostradas en el Cuadro 5.1

Componente	Descripción
Procesador	Intel(R) Pentium(R) 4 CPU 2.80GHz
Memoria Ram	1GB DDR2
Sistema Operativo	GNU/Linux Kernel 2.6.26-2-686
Java VM	Java(TM) SE Runtime Environment 1.6.0_20

Cuadro 5.1: Descripción de la plataforma sobre la que se desarrollaron los experimentos

5.1. Luigi para “Mario AI Competition”

Los objetivos principales del conjunto de pruebas realizados en esta sección son: Medir el rendimiento del agente implementado según los patrones de evaluación de “Mario AI Competition”, variando algunos parámetros de interés explicados más adelante, y comparar a *Luigi* contra el agente ganador de la competencia, el cuál, como se explicó previamente, utiliza la misma técnica de búsqueda de caminos óptimos. La comparación es realizada inicialmente utilizando los patrones de evaluación de la competencia, y posteriormente se muestran otras mediciones para evaluar el rendimiento del algoritmo de búsqueda y de la simulación obtenida.

5.1.1. Descripción general de los experimentos

El proceso para realizar los experimentos se diseñó de forma similar a la evaluación utilizada en “Mario AI Competition”, explicada en la Sección 2.2.5. Sin embargo, para este trabajo se expandió la cantidad de niveles de dificultad a jugar, quedando como sigue **0, 3, 5, 8, 10, 15, 20**. En cada uno de los niveles, son jugados 10 escenarios distintos. Ésto da un total de 70 niveles a completar por cada agente. A continuación se muestran las variables de interés estudiadas y los parámetros modificados en cada experimento realizado.

5.1.1.1. Variables de interés

Como se mencionó anteriormente, los criterios de evaluación de “Mario AI Competition” (explicados con detalle en la Sección 2.2.5) son utilizados para medir el rendimiento de *Luigi*. Dichos criterios estarán representados en las siguientes variables: **Distancia Recorrida, Tiempo Restante, Enemigos Eliminados, Golpes Recibidos**, y son obtenidos a partir de la simulación de los 70 niveles jugados. También son consideradas las variables **Niveles Completados y Monedas Agarradas**.

Además, se evalúa el comportamiento del algoritmo de búsqueda midiendo la cantidad de **Nodos Visitados**, el **Tiempo Promedio de Respuesta**, y las **Metas No Alcanzadas**, que representan todas aquellas *Escenas Simuladas* en las que el algoritmo no logró conseguir una solución en la restricción de tiempo establecida y devolvió una acción utilizando lo explicado en la Sección 3.8.

5.1.1.2. Parámetros estudiados

Para seleccionar los parámetros a variar del algoritmo de búsqueda se estudiaron aquellos elementos mencionados en el Capítulo 3 que podían generar cambios importantes en el proceso de conseguir los posibles caminos o secuencias de acciones que genera el agente. Entre éstos parámetros, los estudiados serán variaciones de: el Factor de Parcialidad (explicado al final de la Sección 3.6), la Restricción de Tiempo (Sección 3.8) y los *Delta* de Selección de Nodos Cerrados (Sección 3.7).

Se realizaron unas pruebas preliminares sobre *Luigi* para elegir la mejor combinación *delta*s de espacio y tiempo para seleccionar nodos cerrados, y en base a los resultados obtenidos se procedió a compararlo con el agente ganador de la competencia. Una descripción más detallada de estas pruebas, junto con los resultados obtenidos, se encuentran en el Apéndice K, Sección K.1.1.

5.1.2. Comparación contra el agente ganador de “Mario AI Competition”

El ganador de “Mario AI Competition” es *Robin Baumgarten* (los resultados se encuentran en el Apéndice B), quien realizó un agente basado en búsqueda de caminos óptimos al igual que el agente implementado en este trabajo.

Las pruebas realizadas para comparar a *Luigi* contra el agente realizado por *Robin Baumgarten*, el cual será llamado *Robin*, se basaron en variar los parámetros de **Factor de Parcialidad** y **Restricción del Tiempo de Búsqueda**, utilizando la mejor configuración obtenida en los resultados de las pruebas preliminares.

Luigi utiliza una estrategia similar a la de *Robin* para cerrar nodos, pero sus *delta* de espacio y tiempo son 2 y 5, respectivamente. Los resultados son extraídos de la evaluación de los mundos con enemigos, los mundos pausados son ignorados pues no traen diferencias significativas con respecto a los primeros resultados. Todos los resultados en los Cuadros presentados son obtenidos usando el **Acumulador de Tiempo Disponible**. A continuación se muestran los resultados obtenidos en ambos conjuntos de pruebas.

5.1.2.1. Variaciones del Factor de Parcialidad

Los valores probados en las variaciones son: **0.7, 0.8, 0.9 y 1.0**. Los resultados expuestos en el Cuadro 5.2 muestran que, con excepción del Factor de Parcialidad 0.8, *Luigi* gana en todos los experimentos; En el factor 1.0 usando el primer criterio de comparación de la competencia, y el segundo criterio en los factores 0.7 y 0.9. También se puede ver que el mejor resultado obtenido, tanto para *Luigi* como para *Robin*, fue conseguido con el Factor de Parcialidad 0.9.

<i>Luigi</i>			
	Niveles Completados (70)	Distancia Recorrida (8135.84)	Tiempo Restante
0.7	70	8135.84	8629
0.8	68	8094.62	8680
0.9	70	8135.84	8695
1.0	70	8135.84	8610
<i>Robin</i>			
	Niveles Completados (70)	Distancia Recorrida (8135.84)	Tiempo Restante
0.7	70	8135.84	8503
0.8	70	8135.84	8562
0.9	70	8135.84	8607
1.0	68	8022.88	8665

Cuadro 5.2: Resultados de simulación para variaciones de Factor de Parcialidad entre *Luigi* y *Robin*.

Usando el Factor de Parcialidad 0.9 como referencia, se muestran los resultados de simulación y rendimiento algorítmico para este experimento.

En la Figura 5.1 se puede apreciar la cantidad de nodos visitados por cada agente. La media indica que *Luigi* visita más nodos por tick de juego.

En la Figura 5.2 se muestra que *Luigi* también alcanza una mayor cantidad de metas que *Robin*. Ésta apreciación, unida a la de la Figura 5.1, da un resultado a destacar: Debido a que *Luigi* utiliza unos delta de selección de nodos cerrados más pequeños que los de *Robin*, tiene un espacio de búsqueda más amplio a explorar, por lo tanto necesita visitar más nodos para conseguir la meta. Lo importante es que, gracias a la velocidad de búsqueda del algoritmo implementado, *Luigi* puede explorar todo el espacio de búsqueda que supone su selección de nodos cerrados angosta y puede conseguir alcanzar más metas en la simulación.

Otro resultado importante obtenido es el tiempo promedio de respuesta de *Luigi* frente al de *Robin*. En la Figura 5.3 se puede apreciar cómo *Luigi* responde, en promedio, a casi el doble de la velocidad de su contrincante, aún teniendo que explorar más espacio para obtener las soluciones.

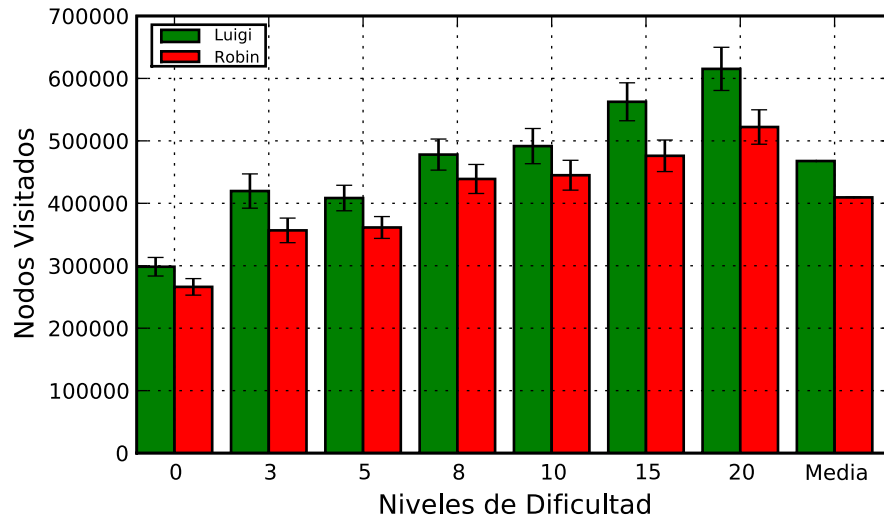


Figura 5.1: Nodos visitados por nivel de dificultad jugados en la variación 0.9 de las pruebas realizadas para *Luigi* y *Robin*

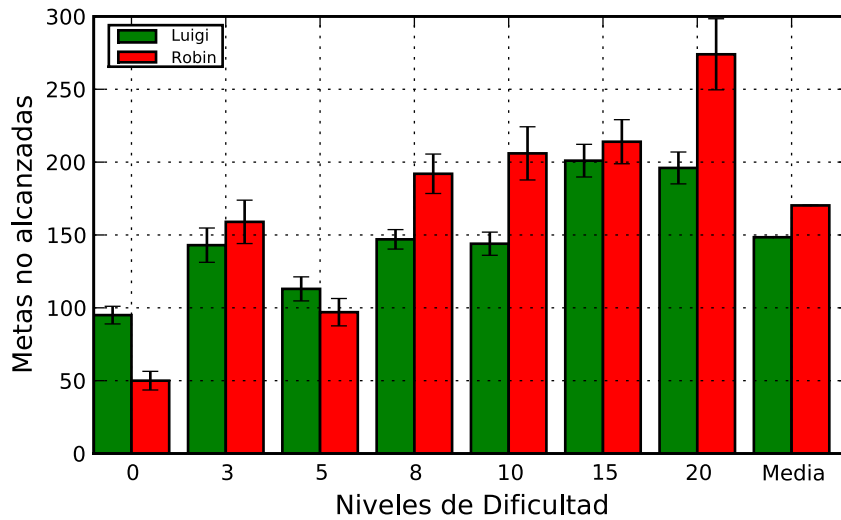


Figura 5.2: Metas no alcanzadas por nivel de dificultad jugados en la variación 0.9 de las pruebas realizadas para *Luigi* y *Robin*

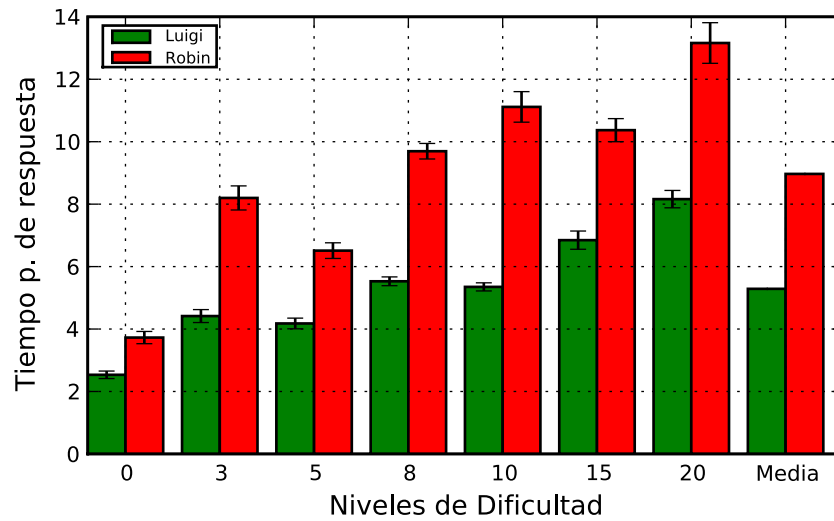


Figura 5.3: Tiempo promedio de respuesta por nivel de dificultad jugados en la variación 0.9 de las pruebas realizadas para *Luigi* y *Robin*

Por último, en el Cuadro 5.3, se pueden ver los resultados del resto de las variables de simulación tomadas en cuenta en los experimentos. *Luigi* destruye menos enemigos y agarra menos monedas que *Robin*, pero también fue herido una menor cantidad de veces.

5.2. Aprendizaje

En esta sección se presentan los experimentos realizados para evaluar el aprendizaje obtenido por el agente bajo el enfoque expuesto en el Capítulo 4. Los objetivos planteados son los siguientes: Evaluar el aprendizaje de *Luigi* bajo condiciones y escenarios aleatorios, comprobar si el aprendizaje es efectivo para optimizar múltiples criterios de evaluación, comparar contra el agente realizado para ganar la competencia (descrito en el Capítulo 3), determinar características útiles de los escenarios de entrenamiento para optimizar el aprendizaje, y finalmente comprobar si el aprendizaje puede ser orientado a satisfacer objetivos específicos.

<i>Luigi</i>			
	Enemigos Destruídos	Golpes Recibidos	Monedas Agarradas
0.7	703	3	3406
0.8	669	3	3389
0.9	650	3	3404
1.0	558	8	2801
<i>Robin</i>			
	Enemigos Destruídos	Golpes Recibidos	Monedas Agarradas
0.7	826	2	3627
0.8	779	5	3526
0.9	725	5	3383
1.0	705	6	3360

Cuadro 5.3: Comparación de objetivos de simulación entre *Luigi* y *Robin*. para las variaciones de Factor de Parcialidad

5.2.1. Descripción general de los experimentos

Se realizaron 5 entrenamientos: El primero sobre un nivel de baja dificultad y longitud media. El segundo y tercero sobre un nivel de corta y larga longitud respectivamente, ambos con baja dificultad. Los últimos dos entrenamientos fueron realizados sobre un nivel con longitud y dificultad media. La configuración utilizada para el entrenamiento de los pesos de la red neuronal es la siguiente: Población de 20 individuos y 200 generaciones por evolución. Como fue mencionado previamente en la Sección 4.2.3.1, para generar una nueva población se utiliza 20% de selección elitista de individuos y el resto se genera como una mutación aleatoria de los primeros individuos.

Cada entrenamiento se realizó utilizando una configuración de pesos para la **función de adaptación** específica al objetivo planteado. En cada entrenamiento se seleccionó al azar la semilla de generación del nivel sobre la cual se haría el aprendizaje, y dicha configuración fue mantenida durante las 200 generaciones.

También se realizó un conjunto de pruebas sobre los entrenamientos descritos, midiendo los siguientes valores (en un promedio de 20 corridas):

1. Tiempo restante de juego.

2. Cantidad de enemigos destruidos.
3. Cantidad de monedas agarradas.
4. Distancia recorrida.
5. Estado de finalización del nivel (0 en caso de haber perdido, 1 en caso contrario).
6. Estado final de Mario (0 en caso de finalizar *Pequeño*, 1 en caso de terminar *Grande* y 2 en caso de *Fuego*).
7. Medida de adaptación final de la partida

Por otra parte, en algunas pruebas que lo requieran, también será calculado la cantidad de veces que el agente seleccione distintas estrategias de búsqueda de caminos para comprobar que el reforzamiento dado por la **función de adaptación** influye en la toma de decisiones del *seleccionador de estrategias*. El conteo será realizado únicamente sobre las elecciones de estrategias que tengan sentido, es decir, solo será válida la estrategia de Agarrar Monedas, cuando haya monedas que agarrar en la escena. Análogamente para los enemigos.

Adicionalmente, en el Apéndice K, Sección K.3 se encuentran las Figuras con el rendimiento alcanzado por *Luigi* en las pruebas realizadas sobre los entrenamientos, para los principales criterios de evaluación medidos (enemigos destruidos y monedas agarradas).

5.2.2. Entrenamiento para un ambiente aleatorio

Este primer experimento sirve para comprobar el aprendizaje alcanzado por el agente en un mundo aleatorio. El vector de pesos utilizado en la **función de adaptación** fue $\langle 12, 25, 0, 25, 200, 50 \rangle$ (para enemigos destruidos, monedas agarradas, velocidad al recorrer, haberlo completado y estado del personaje respectivamente). El nivel empleado para el entrenamiento posee las siguientes características: Semilla de generación 9999121, Dificultad 3 y Longitud 500. La Figura 5.4 muestra la evolución de la adaptación a lo largo del entrenamiento, alcanzando el máximo de 1688 en la generación 26. Es importante destacar que aún cuando el proceso de evolución es errático, el agente logra buenos

resultados como se verá en el experimento siguiente.

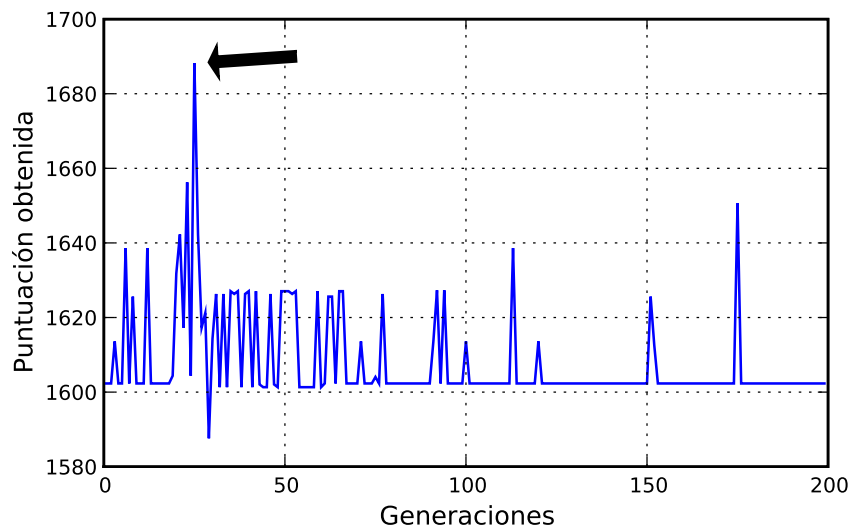


Figura 5.4: Entrenamiento del agente para un mundo aleatorio

5.2.2.1. Rendimiento de *Luigi* con aprendizaje vs. *Luigi* con A*

Luego del entrenamiento es necesario comprobar si efectivamente el agente aprendió a completar varios objetivos a lo largo del juego. Además, en el caso de que el aprendizaje sea exitoso, resulta interesante comparar el rendimiento del agente con aprendizaje contra el agente realizado para ganar la competencia, y ver las diferencias (si las hay) en los objetivos completados. Para llevar a cabo esta prueba, ambos agentes fueron puestos a jugar sobre el nivel utilizado como entrenamiento.

Como se puede observar en el cuadro 5.4, después del entrenamiento el agente logró seleccionar diferentes estrategias de movimiento según la observación recibida del ambiente. Debido a que la **función de adaptación** utilizada da una recompensa por otros objetivos además de completar el nivel rápidamente, *Luigi* avanzó a lo largo del recorrido completando objetivos secundarios (como matar enemigos o agarrar monedas). En el caso del *Luigi* realizado para ganar la competencia, se puede observar que la cantidad de objetivos completados es inferior al logrado con aprendizaje, debido a que la búsqueda en este caso está orientada a un único objetivo.

Criterio de evaluación	Aprendizaje	Competencia
Tiempo restante de juego	126.25	153.95
Enemigos destruidos	23.65	12.05
Monedas agarradas	35.40	30.0
Distancia recorrida	7149.25	7264.0
Estado de finalización del nivel	0.90	1.0
Estado final de Mario	1.3	1.95
Medida de Adaptación	1538.81	1330.69
Estrategias seleccionadas	N° veces	
Destruir Enemigos	343.80	NA
Agarrar Monedas	125.55	NA
Velocidad	628.0	685.75

Cuadro 5.4: Rendimiento de *Luigi* con aprendizaje vs. *Luigi* de la Competencia

5.2.2.2. Generalización del aprendizaje

En el experimento anterior se comprobó que el aprendizaje es efectivo sobre el nivel de entrenamiento. Ahora se cuestionará si el resultado obtenido es bueno debido a una “memorización” de las características del nivel, o el aprendizaje también logra un buen rendimiento sobre ambientes y niveles diferentes. Para verificar si el aprendizaje sobre condiciones específicas es generalizable, se utilizó el agente entrenado sobre dos escenarios generados al azar, con el mismo nivel de dificultad usado en el entrenamiento y la misma longitud. Los resultados, mostrados en el Cuadro 5.5, evidencian que el agente bajo ambientes desconocidos todavía lograr seleccionar estrategias diferentes para completar varios objetivos, lo cual da la idea de que el agente evalúa condiciones similares en estos mundos.

En el Cuadro 5.5 también puede observarse una diferencia en la cantidad de objetivos completados (monedas/enemigos) entre ambos niveles. Esto se debe a que los mundos son generados aleatoriamente con una cantidad de enemigos y monedas variable de acuerdo a la semilla utilizada, por lo que la cantidad de objetivos alcanzables por el agente en cada nivel depende del número de monedas y enemigos que se generaron en el mismo.

Criterio de evaluación	Semilla 256	Semilla 118740
Tiempo restante de juego	122.3	142.25
Enemigos destruidos	7.40	13.05
Monedas agarradas	60.29	75.55
Distancia recorrida	5603.39	6169.61
Estado de finalización del nivel	0.15	0.60
Estado final de Mario	0.15	0.95
Medida de Adaptación	1756.00	2339.60
Estrategias seleccionadas	N° veces	
Destruir Enemigos	344.60	215.80
Agarrar Monedas	436.20	241.5
Velocidad	376.15	399.30

Cuadro 5.5: Validación del aprendizaje sobre niveles desconocidos

5.2.3. Entrenamiento para ambientes de longitud variable

Los niveles utilizados en la competencia para la evaluación de los agentes variaban, además de la semilla, la longitud y dificultad de los mismos. Mientras la dificultad sólo determina el número de enemigos y obstáculos en el nivel, una mayor longitud permite previsiblemente un conjunto más amplio de escenas que observar. Debido a esto, el agente fue sometido a entrenamiento en un nivel generado con la semilla 3143, dificultad 5, y dos longitudes distintas: 320 y 1000. El vector de pesos para la **función de adaptación** es el mismo utilizado anteriormente para el primer experimento.

En las Figura 5.5 se puede apreciar la evolución del *fitness* a lo largo de los entrenamientos. La diferencia notable en el rango exhibido entre ambos entrenamientos se debe a que el nivel de mayor longitud tiene más objetivos que alcanzar (monedas y enemigos) sumado a una extensión mayor, lo que hace que el agente entrenado en ese nivel obtenga una mayor puntuación que entrenando en un nivel de menor longitud.

Para determinar si la longitud del nivel influye en el rendimiento del aprendizaje obtenido, se compararon los agentes entrenados en este experimento sobre dos niveles escogidos aleatoriamente, uno corto de 320 píxeles de longitud, y otro largo de 1000. En los resultados presentados en el Cuadro 5.6 puede observarse que el agente entrenado en

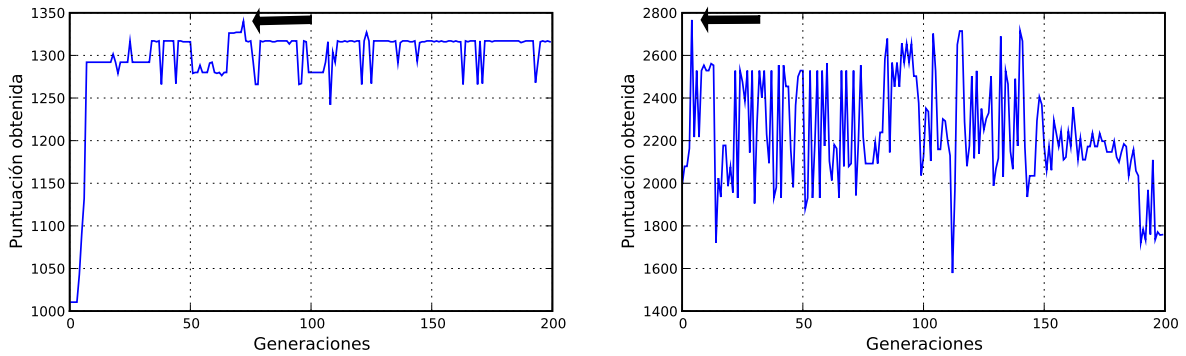


Figura 5.5: Entrenamiento del agente sobre un nivel de corta y larga longitud respectivamente

un nivel de longitud menor logra completar más objetivos (agarrar monedas o destruir enemigos) que el agente entrenado en un nivel de mayor longitud. De este resultado podemos obtener que ambientes más largos conducen a una pobre generalización del aprendizaje, quizás debido a estímulos u observaciones repetitivas a lo largo del nivel que son “memorizadas” por la red a lo largo del entrenamiento.

5.2.4. Entrenamiento para optimizar criterios de evaluación específicos

Con los experimentos ya realizados se comprobó la efectividad del aprendizaje como mecanismo de decisión para optimizar múltiples objetivos. No obstante, es posible querer orientar el aprendizaje hacia la optimización de un objetivo concreto. Para esto, se realizó dos entrenamientos distintos bajo el mismo nivel con las características siguientes: Semilla 500, Dificultad 4 y Longitud 500. La diferencia entre ambos entrenamientos radica en el único objetivo que se desea optimizar: destruir enemigos en el primero y agarrar monedas en el segundo. En función de los dos objetivos mencionados, los vectores de pesos utilizados en cada **función de adaptación** para ambos entrenamientos fueron $\langle 25, 0, 0, 25, 200, 50 \rangle$ y $\langle 0, 25, 0, 25, 200, 50 \rangle$ respectivamente. Es importante notar que, para el primer entrenamiento (destruir enemigos) el peso de recompensa a obtener monedas (P_2) es igualado a 0. Análogamente, para el segundo entrenamiento, el peso de recompensa a destruir enemigos (P_1) también es igualado a 0.

<i>Agente entrenado en mundo de 320 pixeles</i>		
Criterio de evaluación	Semilla 2280921 Longitud 320	Semilla 7534 Longitud 1000
Tiempo restante de juego	156.90	115.05
Enemigos destruidos	7.90	25.25
Monedas agarradas	24.19	66.95
Distancia recorrida	4061.10	12765.61
Estado de finalización del nivel	0.90	0.84
Estado final de Mario	1.15	1.8
Medida de Adaptación	1061.89	2372.56
<i>Agente entrenado en mundo de 1000 pixeles</i>		
Criterio de evaluación	Semilla 2280921 Longitud 320	Semilla 7534 Longitud 1000
Tiempo restante de juego	156.35	116.90
Enemigos destruidos	4.45	22.19
Monedas agarradas	24.15	43.30
Distancia recorrida	4162.19	7369.96
Estado de finalización del nivel	0.90	0.14
Estado final de Mario	0.80	0.15
Medida de Adaptación	1001.02	1508.62

Cuadro 5.6: Comparativa entre los agentes entrenados bajo niveles de longitud variable

En la Figura 5.6 se puede apreciar la evolución de la adaptación a lo largo de los entrenamientos. Aún cuando ambos entrenamientos son realizados sobre el mismo nivel y la recompensa por matar enemigos y agarrar monedas es igual, puede verse una diferencia considerable en el *fitness* alcanzado. Esto se debe a dos motivos: primero, en promedio suele haber más monedas por nivel que enemigos, por lo que hay una recompensa potencial mayor para el agente que busca monedas que para el agente que caza enemigos, y segundo, la heurística que guía hacia los enemigos no es tan precisa como la que guía hacia las monedas (principalmente porque los enemigos son objetivos móviles), por lo que suceden más fallas en el primer entrenamiento que en el segundo.

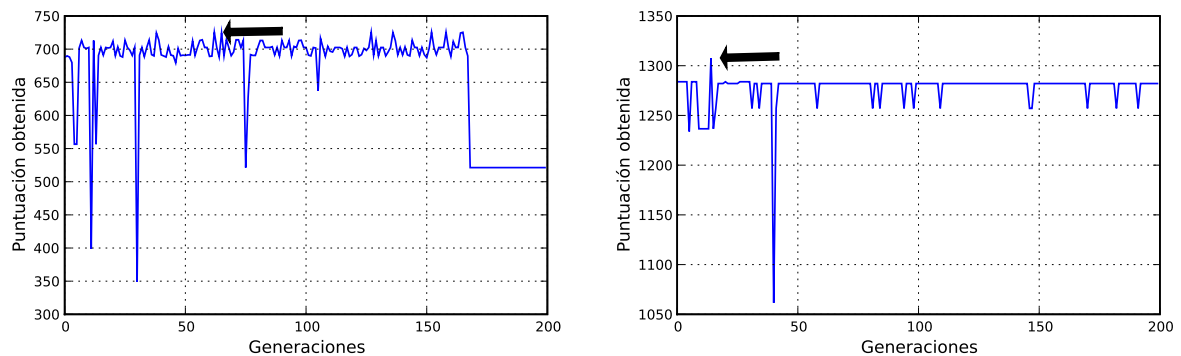


Figura 5.6: Entrenamiento selectivo para destruir enemigos y recoger monedas respectivamente

Para comprobar que el aprendizaje podía ser guiado hacia un objetivo específico, se procedió a comparar a los agentes entrenados bajo un mismo nivel, seleccionado aleatoriamente. Los resultados presentados en el Cuadro 5.7 muestran claramente que para ambos casos, el agente entrenado aprende a optimizar únicamente los criterios premiados por la **función de adaptación**, mientras ignora el resto de los objetivos.

En este experimento también es importante destacar varios resultados: En promedio, el agente entrenado para optimizar las monedas agarradas obtiene un tiempo restante que duplica al que optimiza la destrucción de enemigos, esto es coherente con el hecho de que las monedas son objetivos estáticos, mientras que los enemigos se mueven. Para conseguir una moneda, la cantidad de caminos a estudiar no varía, mientras que para un

Criterio de evaluación	Favoreciendo Monedas	Favoreciendo Enemigos
Tiempo restante de juego	146.65	68.40
Enemigos destruidos	6.55	14.15
Monedas agarradas	37.0	12.4
Distancia recorrida	7216.0	3663.43
Estado de finalización del nivel	1.0	0.05
Estado final de Mario	1.05	0.5
Estrategias seleccionadas	N° veces	
Destruir Enemigos	0.0	1652.7
Agarrar Monedas	113.25	0.0
Velocidad	677.85	316.30

Cuadro 5.7: Comparativa de aprendizaje selectivo sobre nivel desconocido, cuya semilla de generación es 92060

enemigo, cada vez que éste cambia de posición, hay que planificar de nuevo el camino para destruirlo, esto hace que matar enemigos sea mucho más difícil.

En ambos agentes se destruyen enemigos y se agarran monedas, y esto se debe a que, aunque el agente no busque monedas, si se encuentra con ellas en su paso, las agarra. Al igual pasa con los enemigos, donde a veces los utiliza para impulsarse y llegar más lejos, o los mata con bolas de fuego. Sin embargo, al estudiar la cantidad de veces que fueron seleccionadas las estrategias correspondientes, se puede ver que el agente que optimiza la destrucción de enemigos, nunca seleccionó la estrategia de agarrar monedas. Lo mismo sucedió en el caso contrario. Se puede ver también que el promedio de finalización del nivel para el agente que optimiza monedas fue **1.0**, lo que quiere decir que completó todos los niveles. El entrenamiento para la cacería de enemigos, por el contrario, logró completar el nivel sólo la mitad de las veces. Esto se debe a que la heurística para perseguir enemigos trae problemas al acercar a *Luigi* a los elementos del mundo que le hacen daño.

Capítulo 6

Conclusiones y Recomendaciones

En el presente trabajo se realizó un agente inteligente capaz de jugar satisfactoriamente cualquier nivel del videojuego usado en la competencia “Mario AI Competition”, y superar notablemente el resultado alcanzado por todos los participantes de la misma. Posteriormente se realizó una aproximación inicial, combinando dicha implementación con enfoques de aprendizaje de máquinas, para conseguir que el agente se adapte a criterios de evaluación distintos a los propuestos en la competencia.

Para la implementación realizada se efectuó un estudio de todos los elementos que componen el ambiente de juego y se aprovechó el carácter determinístico de la física del mismo para crear un controlador que busque caminos óptimos dentro de los niveles jugados. El controlador implementado está basado en el algoritmo de búsqueda en amplitud A^* , que es utilizado cada vez que el agente devuelve una acción a ejecutar sobre el personaje principal. Además del motor de búsqueda realizado para conseguir soluciones, se desarrollaron varias herramientas que le permiten al agente utilizar la información provista en la competencia para construir una simulación interna del nivel que está jugando. También se tomó ventaja de las propiedades físicas conocidas del juego para implementar optimizaciones sobre el algoritmo A^* clásico, que mostraron ser efectivas en aumentar la velocidad de la búsqueda de soluciones, y consecuentemente el desempeño del agente en la simulación de los niveles aleatorios generados. Después de experimentar con varios ajustes y configuraciones diferentes para el algoritmo de búsqueda, se realizó una comparación contra el agente ganador de la competencia (que también utiliza el enfoque de búsqueda de caminos). Bajo las pruebas realizadas, se obtuvieron resultados superiores con el agente implementado, tanto en la simulación de los mundos jugados, como en el rendimiento del algoritmo de búsqueda.

La adecuación del agente implementado para poder optimizar distintos criterios de

evaluación se realizó como una abstracción de los objetivos a ser buscados por el A^* implementado. Luego fue implementado un planificador de alto nivel encargado de recibir la información del mundo, analizarla con un seleccionador de estrategias, e indicarle al algoritmo de búsqueda cuál objetivo es conveniente buscar. Para que dicho componente del agente tomara decisiones correctas, se realizó una aproximación inicial con *Neuroevolución* [18] como mecanismo de aprendizaje.

Las conclusiones específicas obtenidas tras la realización de este trabajo son las siguientes:

1. Para un ambiente de juego con física determinística como el de la competencia, las aproximaciones de búsqueda de caminos ofrecen el mejor desempeño. En el caso particular mostrado en “Mario AI Competition”, este enfoque obtuvo los mejores resultados.
2. Realizar una versión particular del algoritmo A^* que aproveche el conocimiento sobre la física determinística del ambiente para la creación de los nodos sucesores, hizo posible obtener unas mejoras significativas en la velocidad de búsqueda, casi doblando el valor original, y permitiendo superar ampliamente el rendimiento mostrado por el agente ganador de la competencia, que utilizaba el algoritmo A^* clásico.
3. Utilizando el enfoque de búsqueda de caminos óptimos, es posible realizar agentes que completen los niveles optimizando distintos criterios. Sin embargo, esto aumenta considerablemente el costo de la evaluación de las heurísticas apropiadas a cada objetivo.
4. Utilizando una planificación de alto nivel, es posible aprovechar y mantener el rendimiento de la búsqueda mientras se completan múltiples objetivos, y a su vez, separar el proceso de búsqueda de la toma de decisiones. Esto último permite aplicar distintos enfoques para seleccionar la estrategia a completar.

5. Aunque es posible alcanzar el aprendizaje, la complejidad del juego, el tiempo de evaluación y la necesidad de generar una gran cantidad de poblaciones para poder apreciar resultados, hace que las pruebas y entrenamientos sean costosos de realizar en tiempo. Con una mayor cantidad de tiempo y procesamiento se pueden conseguir resultados más exactos.

6.1. Aportes realizados

1. Se realizó una aproximación de búsqueda de caminos que toma en cuenta características particulares del problema a resolver para apoyar no sólo a la heurística de un nodo, sino también a la evaluación de todos los nodos sucesores de cualquier nodo que esté siendo expandido. Utilizando la técnica de simulación retrasada realizada en este proyecto, se disminuye significativamente este costo y se obtiene una velocidad de búsqueda mayor.
2. Teniendo una mayor velocidad de búsqueda se pudo realizar un agente que inspecciona con mayor profundidad los espacios del ambiente de juego, logrando conseguir caminos más refinados. La diferencia lograda al conseguir estos caminos se tradujo en mejores resultados en la simulación del juego, y en un mejor comportamiento con respecto a los criterios de evaluación utilizados en la competencia.
3. Se implementó una abstracción que une los enfoques de búsqueda de caminos y aprendizaje de máquinas para un agente autónomo que se desenvuelve en el ambiente de "Super Mario Bros.". El aporte de éste enfoque implica un primer paso en la exploración de técnicas de aprendizaje para conseguir agentes de dicho juego que se adapten a nuevas circunstancias.

6.2. Direcciones futuras

1. Se recomienda realizar un estudio a mayor profundidad de las clases de *Java*TM que conforman la simulación de las escenas, debido a que si se logran simplificar más y

se logran integrar todas en menos abstracciones, se podría eliminar cierto *overhead* de instancias en la máquina virtual que se encarga de correr las evaluaciones de la competencia, originado por todas los nodos que son clonados durante la búsqueda.

2. Se aconseja profundizar las heurísticas planteadas para los criterios de evaluación de destruir enemigos y agarrar monedas. Se cree que realizando mejoras sobre estas heurísticas se conseguirán resultados mucho mejores en el aprendizaje.
3. Considerando los posibles objetivos del videojuego se recomienda crear nuevas heurísticas que los maximicen, para mejorar el comportamiento y puntuación del agente.
4. Es recomendable probar con distintos enfoques de aprendizaje de máquina para realizar la selección de estrategias y la planificación, como por ejemplo Clasificadores Genéticos [23]. También se recomienda utilizar *Neuroevolución* [18] tratando de evolucionar las topologías de las redes neuronales utilizadas como hipótesis.

Bibliografía

- [1] A. L. Samuel, "Some Studies in Machine Learning Using The Game of Checkers," *In Computation and Intelligence: Collected Readings*, 1995.
- [2] S. Akl, *Checkers-Playing Programs*. In *Encyclopedia of Artificial Intelligence*, vol. 1, pp. 88–93. Ed. Shapiro New York: John Wiley and Sons, 1987.
- [3] C. E. Shannon, "Programming a Computer for Playing Chess," *Philosophical Magazine*, vol. 41, 1950.
- [4] M. Newborn, *Computer Chess*. Academic Press, NY, 1975.
- [5] H. A. Simon and J. Schaeffer, "The Game of Chess," *Handbook of Game Theory with Economic Applications*, vol. 1, pp. 1–17, 1992.
- [6] G. Tesauro, "Temporal Difference Learning and TD-Gammon," *Communications of the ACM*, vol. 38, pp. 58–68, 1995.
- [7] M. Mateas, "Expressive AI: Games and Artificial Intelligence," Level Up: Digital Games Research Conference, 2003.
- [8] P. Burrow and S. Lucas, "Evolution versus Temporal Difference Learning for Learning to Play Ms. Pac-Man," *IEEE Symposium on Computational Intelligence and Games*, 2009.
- [9] I. Millington, *Artificial Intelligence for Games*. Elsevier, 2006.
- [10] J. E. Laird and M. van Lent, "Human-level AI's Killer Application: Interactive Computer Games,"
- [11] M. Buro and T. Furtak, "RTS Games as Test-Bed for Real-Time AI Research,"
- [12] S. Rabin, *AI Game Programming Wisdom 2*, pp. 485–498. 2003.
- [13] J. Togelius. Disponible en: <http://julian.togelius.com/mariocompetition2009>, consultado el 23 de septiembre de 2010.
- [14] M. Persson. Disponible en: <http://www.mojang.com/notch/mario>, consultado el 23 de septiembre de 2010.
- [15] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2 ed., 2003.
- [16] I. T. R. Korf, W. Zhang and H. Hohwald, *Frontier Search*, vol. 52. 2005.
- [17] T. Mitchell, *Machine Learning*. McGraw-Hill, New York, NY, USA, 1 ed.

- [18] R. Miikkulainen, "Neuroevolution," *Encyclopedia of Machine Learning*.
- [19] K. Chellapilla and B. Fogel, "Evolution, neural networks, games, and intelligence," *Proceedings of the IEEE*, vol. 87, pp. 1471–1496, 1999.
- [20] S. J. M. R. Gomez, F., "Efficient non-linear control through neuroevolution," *Proceedings of the European Conference on Machine Learning (ECML-06, Berlin)*.
- [21] M. R. Stanley, K. O., "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10 (2), 2002.
- [22] S. M. Lucas, "Evolving a Neural Network Location Evaluator to Play Ms. Pac-Man,"
- [23] R. J. Urbanowicz and J. H. Moore, "Learning classifier systems: A complete introduction, review, and roadmap," *Journal of Artificial Evolution and Applications*, 2009.

Apéndice A

Interfaces de la competencia

Éste apéndice describe con profundidad las dos interfaces utilizadas por la competencia para realizar la comunicación entre los agentes implementados y la simulación de juego.

A.1. Interfaz del agente

La interfaz del agente, descrita en la Sección 2.2.2, presenta los siguientes métodos.

- `public void reset()`

Método invocado antes de iniciar la ejecución del agente, para crear y preparar las estructuras necesarias para devolver la acción a ejecutar.

- `public boolean[] getAction(Environment observation)`

Es el método principal de la interfaz, y la parte central en la implementación del agente. Aquí se debe calcular la acción a ser ejecutada por Mario. La acción es un arreglo de *booleanos* que corresponden a los movimientos posibles [*Izquierda, Derecha, Abajo, Saltar, Correr*] que se pueden ejecutar en un determinado momento. En cada tick de juego, el agente calcula y devuelve dicha acción al simulador a través del valor de retorno de este método, para que la misma sea ejecutada en la simulación.

- `public AGENT_TYPE getType()`

Devuelve el tipo del agente, que varía entre *AI, HUMAN* o *TCP_SERVER*.

- `public String getName()`

Devuelve el nombre del agente.

- `public void setName(String name)`

Asigna *name* como nombre del agente.

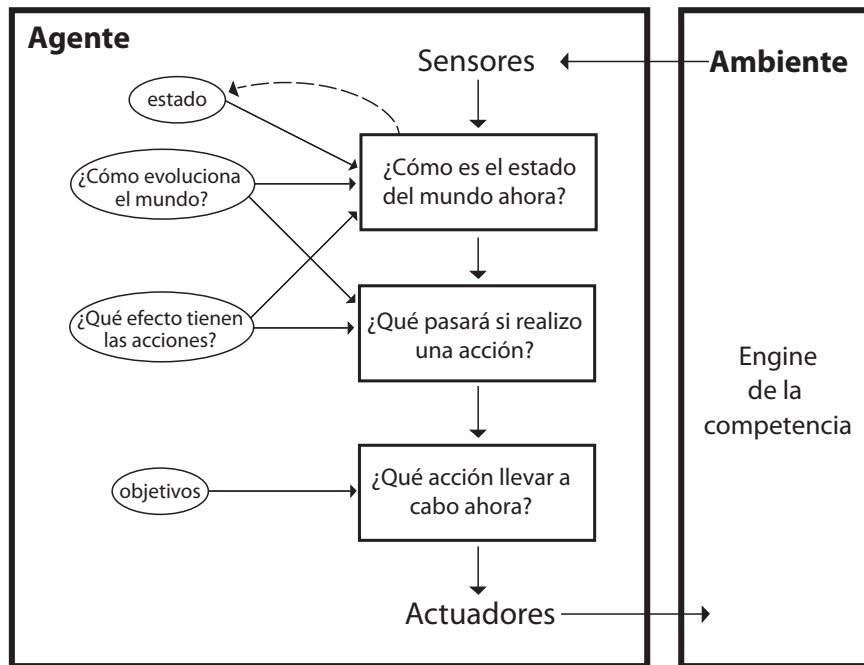


Figura A.1: Un agente basado en objetivos mantiene un seguimiento del estado del ambiente donde se desenvuelve y de las metas que quiere conseguir, eligiendo las acciones que lo llevarán a obtenerlas.

A.2. Interfaz de observación

La interfaz provista por el motor de simulación de la competencia ofrece los siguientes métodos para acceder a la información del mundo:

- `public byte[][] getCompleteObservation()`
 Devuelve una matriz de 22x22 con la representación en *bytes* de la geometría de la escena y todos los enemigos en ella.
- `public byte[][] getEnemiesObservation()`
 Devuelve una matriz de 22x22 con la representación en *bytes* de los enemigos en la escena.
- `public byte[][] getLevelSceneObservation()`
 Devuelve una matriz de 22x22 con la representación en *bytes* de la geometría de la escena.

- `public float[] getMarioFloatPos()`
Devuelve la posición de Mario en los ejes X e Y.
- `public int getMarioMode()`
Devuelve 2 si Mario está en estado *Fuego*, 1 si está *Grande* y 0 si está *Pequeño*.
- `public float[] getEnemiesFloatPos()`
Devuelve un arreglo con el tipo y posición en los ejes X e Y de cada enemigo en la escena.
- `public boolean isMarioOnGround()`
Devuelve true si Mario está en el suelo.
- `public boolean mayMarioJump()`
Devuelve true si Mario puede saltar.
- `public boolean isMarioCarrying()`
Devuelve true si Mario está cargando un *Shell*.
- `public int getKillsTotal()`
Devuelve la cantidad total de enemigos eliminados.
- `public int getKillsByFire()`
Devuelve la cantidad de enemigos eliminados con bolas de fuego.
- `public int getKillsByStomp()`
Devuelve la cantidad de enemigos aplastados.
- `public int getKillsByShell()`
Devuelve la cantidad de enemigos eliminados con *Shells*.
- `public boolean canShoot()`
Devuelve true si Mario puede disparar bolas de fuego.

- `public String getBitmapEnemiesObservation()`

Devuelve un string con la representación de los enemigos en la escena (para uso de la interfaz *TCP*).

- `public String getBitmapLevelObservation()`

Devuelve un string con la representación de la geometría de la escena (para uso de la interfaz *TCP*).

Apéndice B

Mario AI Competition 2009

Participante	Algoritmo	Puntuación	Niveles	Tiempo restante	Total enemigos muertos	Modo
Robin Baumgarten	A*	46564.8	40	4878	373	76
Peter Lawford	A*	46564.8	40	4841	421	69
Andy Sloane	A*	44735.5	38	4822	294	67
Trond Ellingsen	RB	20599.2	11	5510	201	22
Sergio Lopez	RB	18240.3	11	5119	83	17
Spencer Schumann	RB, H	17010.5	8	6493	99	24
Matthew Erickson	Ev, GP	12676.3	7	6017	80	37
Douglas Hawkins	Ev, GP	12407.0	8	6190	90	32
Sergey Polikarpov	CN	12203.3	3	6303	67	38
Mario Pérez	SM	12060.2	4	4497	170	23
Alexandru Paler	NN, A*	7358.9	3	4401	69	43
Michael Tulacek	SM	6571.8	3	5965	52	14
Rafael Oliveira	RB, H	6314.2	1	6692	36	9
Glenn Hartmann	RB, H	1060	0	1134	8	71
Erek Speed	GA		Consumió toda la memoria			

Cuadro B.1: Puntuación de los participantes

Enfoques o métodos utilizados			
CN	Ciberneuronas	GP	Programación genética
Ev	Algoritmo evolutivo	NN	Redes neuronales
H	Algoritmo heurístico	SM	Máquina de estados
GA	Algoritmo genético	RB	Basado en reglas

Apéndice C

Algoritmos implementados

C.1. A*

A continuación se muestra el algoritmo clásico de A*, en forma de pseudo-código. las funciones `estimar_costo`, `reconstruir_camino`, `sucesores` y `costo` son auxiliares, y realizan el trabajo comentado previamente en el libro.

La variable `abiertos` representa una cola de prioridad y la variable `cerrados` representa una lista enlazada.

```
_____ Pseudo-código del algoritmo A* clásico _____
1 def a_star(nodo_inicial, nodo_objetivo):
2     abiertos = [nodo_inicial]
3     cerrados = []
4     nodo_inicial.g = 0
5     nodo_inicial.h = estimar_costo(nodo_inicial, nodo_objetivo)
6     nodo_inicial.f = nodo_inicial.h
7
8     while abiertos != []:
9         nodo = abiertos.tope()
10        if nodo == nodo_objetivo:
11            return reconstruir_camino(nodo)
12
13        cerrados.agregar(nodo)
14        foreach hijo in sucesores(nodo):
15            if hijo in cerrados:
16                continue
17
18            puntuacion_g_tentativa = nodo.g + costo(nodo, hijo)
19            if hijo not in abiertos:
20                abiertos.agregar(hijo)
21                estimado_es_mejor = True
22            else if puntuacion_g_tentativa < hijo.g:
23                estimado_es_mejor = True
24            else:
25                estimado_es_mejor = False
26
```

```

27         if estimado_es_mejor:
28             hijo.g = puntuacion_g_tentativa
29             hijo.h = estimar_costo(hijo, nodo_objetivo)
30             hijo.f = hijo.g + hijo.h
31     return ERROR

```

C.2. Reconstrucción de enemigos

A continuación se presenta, en forma de pseudo-código, el algoritmo implementado para reconstruir los enemigos en la simulación, a partir de los enemigos que llegan en la observación dada por el *motor* de la competencia.

Ambos parámetros son listas de sprites. El primero es un parámetro de entrada y salida, el segundo sólo de entrada.

Pseudo-código de la reconstrucción de enemigos

```

1     def setEnemies(sprites_en_simulacion, enemigos_en_obs):
2         enemigos_tmp = []
3         nuevos_sprites = []
4         foreach sprite in sprites_en_simulacion:
5             if sprite is Enemy:
6                 enemigos_tmp.agregar(sprite)
7             else:
8                 nuevos_sprites.agregar(sprite)
9
10        foreach enemigo in enemigos_en_obs:
11            enemigo_encontrado = False
12
13            foreach enemigo_tmp in enemigos_tmp:
14                if enemigo == enemigo_tmp:
15                    nuevos_sprites.agregar(enemigo)
16                    enemigos_tmp.eliminar(enemigo)
17                    enemigo_encontrado = True
18                    break
19            if enemigo_encontrado:
20                continue
21
22            foreach enemigo_tmp in enemigos_tmp:
23                if fixup_definite_match(enemigo, enemigo_tmp):
24                    nuevos_sprites.agregar(enemigo)
25                    enemigos_tmp.eliminar(enemigo)

```

```

26         enemigo_encontrado = True
27         break
28     if enemigo_encontrado:
29         continue
30
31     foreach enemigo_tmp in enemigos_tmp:
32         if fixup_probably_match(enemigo, enemigo_tmp):
33             nuevos_sprites.agregar(enemigo)
34             enemigos_tmp.eliminar(enemigo)
35             enemigo_encontrado = True
36             break
37     if enemigo_encontrado:
38         continue
39
40     # Nuevo enemigo
41     nuevo_enemigo_simulado = crear_enemigo_simulado(enemigo)
42     nuevos_sprites.agregar(nuevo_enemigo_simulado)
43
44     sprites_en_simulacion = nuevos_sprites

```

C.3. A* con simulación retrasada

El siguiente algoritmo es una modificación del algoritmo clásico A* realizada en este trabajo para mejorar el rendimiento de la búsqueda de caminos en el caso particular del agente artificial implementado para la competencia.

```

_____ Pseudo-código de la reconstrucción de enemigos _____
1     def luigi_a_star(escena):
2         tiempo_inicial = TIEMPO_ACTUAL
3         escena.g = 0
4         escena.ya_simulada = True
5
6         abiertas = [escena]
7         cerradas = []
8
9         escena_mas_lejana = escena
10        meta_encontrada = False
11
12        while abiertos != [] &&
13            (TIEMPO_ACTUAL - tiempo_inicial) < max_tiempo:
14            escena_actual = abiertos.tope()

```

```
15         if not escena_actual.ya_simulada:
16             escena_actual.simular()
17             escena_actual.ya_simulada = True
18             repensar = False
19             penalizacion = calcular_penalizacion(escena_actual)
20
21             if penalizacion > 0:
22                 escena_actual.p = penalizacion
23                 escena_actual.ha_sido_herida = True
24                 repensar = True
25                 if abs(escena_actual.x - escena_actual.xa) < 0.01:
26                     escena_actual.h = estimar_costo(escena_actual.v)
27                     repensar = True
28                 if repensar:
29                     abiertos.agregar(escena_actual)
30                     continue
31
32         if escena_actual.es_meta():
33             meta_encontrada = True
34             break
35
36         # Chequeo de nodos cerrados
37         if escena_actual.ya_estaba_cerrada():
38             escena_actual.quitar_penalizacion_por_cierre()
39         else:
40             if escena_actual in cerradas:
41                 escena_cerrada = cerradas.obtener_serializacion(escena_actual)
42                 if escena_cerrada[g] <= escena_actual.g:
43                     escena_actual.dar_penalizacion_por_cierre()
44                     abiertas.agregar(escena_actual)
45                     continue
46                 else:
47                     escena_cerrada[g] = escena_actual.g
48             else:
49                 cerradas.agregar(escena_actual)
50
51         if not escena_actual.ha_sido_herida &&
52             escena_actual.x > escena_mas_lejana.x:
53             escena_mas_lejana = escena_actual
54
55         foreach accion in escena_actual.posibles_acciones():
56             escena_vecina = copia(escena_actual)
57             escena_vecina.padre = escena_actual
58             escena_vecina.accion = accion
59             escena_vecina.g = escena_actual.g + 1
```

```
60         escena_vecina.h = estimar(escena_actual)
61         abiertos.agregar(escena_vecina)
62
63     if meta_encontrada:
64         return reconstruir_camino(escena_actual)
65     else:
66         return reconstruir_camino(escena_mas_lejana)
```

Apéndice D

Línea de tiempo de la Simulación

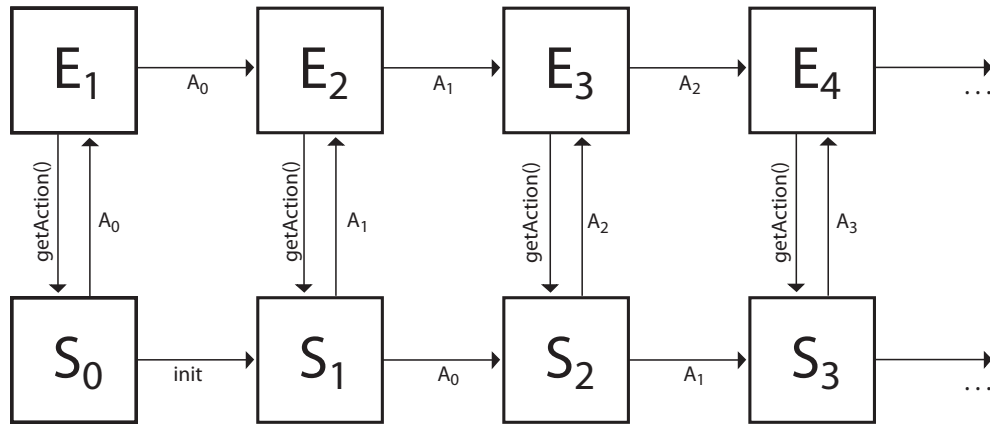


Figura D.1: Comunicación de *Luigi* con el *engine* de la competencia

Como se puede ver en la Figura D.1, la simulación del mundo realizada por el agente implementado se encuentra siempre un paso por detrás de la realizada por el motor de simulación de la competencia.

Esta decisión de diseño fue tomada para optimizar el tiempo de respuesta del agente: si el *Simulador* estuviese un paso por delante del motor de simulación de la competencia, en cada tick se tendría que preguntar: ¿Es el primer tick de juego? Si es así entonces se debe iniciar el simulador con información de nivel cargada por defecto, y realizar una simulación de la acción a devolver sobre ese estado del nivel; si no, se debe simular a partir del último estado de nivel que se realizó en la simulación previa. Al postergar la simulación para que se realice únicamente bajo la información del nivel dada por el *engine* de la competencia, se elimina la necesidad de realizar ésta verificación previa antes de cada tick de juego.

Apéndice E

Simulación interna del juego

Como se mencionó en 3.2, para la representación y simulación del estado interno manejado por *Luigi* se reutilizaron varias clases originales del motor de simulación de la competencia. Esto permitió simular con precisión las acciones a ejecutar por *Luigi* y su efecto en el ambiente que lo rodea, así como también predecir el comportamiento de los enemigos en la escena. A continuación presentamos una breve descripción de las clases utilizadas para ello:

- `Level.java`

Contiene métodos para la creación del nivel y asignación de la información geométrica del mismo (obstáculos, abismos, entre otros objetos.)

- `Sprite.java`

Contiene la rutinas para actualizar el estado y posición de los sprites del juego.

- `Mario.java`

Contiene la inicialización de Mario, métodos para acceder a su estado, fórmulas de movimiento, rutinas de comprobación de colisiones contra obstáculos y enemigos, rutinas para detectar si es herido por un enemigo, para actualizar su estado cuando toma un hongo o una flor.

- `Enemy.java`

Contiene la inicialización, comprobación de colisiones, fórmulas de movimiento para Goombas, Spiky, Red y Green Koopas.

- `CoinAnim.java`

Contiene un método para la animación de las monedas cuando son tomadas por Mario.

- **BulletBill.java**

Contiene la inicialización, comprobación de colisiones y fórmulas de movimiento para las balas.

- **FlowerEnemy.java**

Contiene la inicialización y fórmula de movimiento para la flor piraña.

- **Shell.java**

Contiene la inicialización, comprobación de colisiones, fórmulas de movimiento para los caparzones.

- **FireFlower.java**

Contiene la inicialización y fórmula de movimientos de las flores.

- **Mushroom.java**

Contiene la inicialización y fórmula de movimientos de los hongos.

- **Fireball.java**

Contiene la inicialización y fórmula de movimientos de las bolas de fuego.

Apéndice F

La Escena Simulada

A continuación se muestra con detalle los elementos que componen una *Escena Simulada*:

F.1. Los atributos

- La acción que *Luigi* tiene establecida en la escena.
- La *Escena Simulada* padre, de la cual proviene la escena.
- Valores para calcular su **Función de Selección**, que son explicados en la Sección 3.6.
- Valores para establecer si *Luigi* alcanzó la meta, explicado en la Sección 3.5.
- Un booleano que indica si la escena fue simulada antes o no.
- Un booleano que indica si el nodo actual, o algún nodo antecesor a él, ha sido penalizado (Sección 3.6). El uso de este atributo se explica con detalle en la Sección 3.8.1.
- Valores para establecer estimaciones de la posición de *Luigi* sin haber realizado la simulación (Sección 3.8.2)

F.2. Los métodos

- `public void simulate()`: método que se encarga de simular una acción. Lo hace estableciendo en *Luigi* la acción propia de la escena y luego llamando al método `tick()` que ha sido heredado de `LevelScene.java`.
- `public ArrayList<boolean[]>possibleActions()`: método que se encarga de estudiar el estado actual de *Luigi* en la escena y ver cuales podrían ser las acciones a realizar en ese momento. Este método representa la función de sucesores en el algoritmo de búsqueda, que es detallada en la Sección 3.5.2.

- `public boolean hasTerminalDamage()`: método que indica si *Luigi* está siendo herido de forma mortal en la escena.
- `public boolean isInsideAGap()`: método que estudia la geometría de la escena y la posición de *Luigi* para poder indicar si *Luigi* se encuentra dentro de un abismo.

Apéndice G

Reconstrucción

A continuación se explica con detalle los métodos nombrados en la Sección 3.3.

G.1. Reconstrucción de la geometría del nivel

Este paso es llevado a cabo por el método estático `setLevelScene`, que recibe dos parámetros: la *Escena Simulada* a reconstruir, y la matriz de datos de tamaño 22x22 enviada en la observación provista por el *motor* de la competencia. Básicamente, en este paso se recibe la matriz y se procesa para integrarla con la información del nivel mantenida en la *Escena Simulada* del *Simulador*.

No obstante, el proceso de recibir la matriz e integrarla con la información del nivel mantenido en la simulación no es directo, debido a que dicha información se mantiene en una representación de bajo nivel igual a la empleada en el *motor* de la competencia. Para lograr esto, se analizó el procedimiento ejecutado por el *motor* para generar la matriz a partir de la representación interna que maneja del nivel. En éste procedimiento el *motor* utiliza la posición actual de *Mario* para ubicar en cuál celda o casilla se encuentra de la matriz que representa el nivel, luego crea una sub-matriz de *bytes* de 22x22 a partir de la matriz original, con un *byte* representando a *Mario* en la casilla (11,12) y otros *bytes* en el resto de las casillas, representando los elementos de la geometría del nivel (en el Apéndice H se detalla la codificación de *bytes* usada en la competencia). Sabiendo como se realiza la construcción de la matriz pasada en la observación, es sencillo implementar el algoritmo que describe el procedimiento inverso, y así utilizar dicha matriz para generar la representación interna del nivel que será integrada en la *Escena Simulada*.

Adicionalmente, durante la reconstrucción de la geometría del nivel, se revisan las últimas 3 columnas de la representación del mismo en la matriz de observación, para

ver si existe algún bloque en ellas que represente el suelo. Si no se encuentra ningún bloque que represente el suelo en una columna, se asume que hay un abismo allí y se establece, a priori, que el abismo tiene 3 casillas o celdas de ancho (por ser el máximo ancho de un abismo en el juego). También se guarda a qué altura en el eje Y se encuentra el suelo alrededor de dicho abismo, para saber si *Luigi* se encuentra debajo de esa altura. Esta información es almacenada en dos arreglos que tienen como tamaño la cantidad de columnas en la matriz de representación del nivel en la *Escena Simulada*. El primero es un arreglo de booleanos donde cada casilla representa una columna del nivel en juego, y su contenido es *verdadero* si en esa columna se encuentra un abismo y *falso* en caso contrario. El segundo es un arreglo de enteros que almacena la altura a la que se encuentra el suelo en cada abismo, o 0 si la columna no representa uno.

Para saber si *Luigi* se encuentra cayendo en un abismo, basta chequear ambos arreglos y comparar con la posición en el eje Y.

G.2. Reconstrucción de los enemigos presentes

Este paso es llevado a cabo por el método estático `setEnemies`, que recibe dos parámetros: la *Escena Simulada* donde se van a reconstruir los enemigos, y el arreglo de enemigos proveniente de la observación recibida del *motor* de la competencia. La función de dicho método es recibir el arreglo con la información de los enemigos en el juego, e integrarlo a la lista de enemigos que es mantenida en la *Escena Simulada* del *Simulador*.

Para la reconstrucción de los enemigos dentro de una escena, hay que considerar varios aspectos importantes: (1) si la *Escena Simulada* actual no tiene enemigos, sólo es necesario tomar la información proveniente del *motor* (posiciones y tipo de cada enemigo) y crear nuevos *Sprites* para su representación y adición a la lista de enemigos de la *Escena Simulada*, la cual debe estar vacía para el momento; (2) si la *Escena Simulada* ya tiene enemigos en su representación interna del nivel, entonces es necesario realizar los siguientes pasos adicionales:

1. Identificar cuáles enemigos en la lista de la *Escena Simulada* están en el arreglo

recibido desde la observación.

2. Identificar cuáles enemigos del arreglo de la observación son nuevos, no se encuentran en la lista de la *Escena Simulada*, y deben ser creados en la simulación.
3. Identificar cuáles enemigos de la lista de la *Escena Simulada* no están en el arreglo recibido desde la observación, lo que quiere decir que son enemigos que ya no están en la escena del *motor* de la competencia y deben ser removidos.

Además, también se debe considerar que la simulación física de los enemigos en la *Escena Simulada* puede fallar, lo que hace todavía más complicado el proceso de reconstrucción. Esto significa que en caso de que la simulación falle y se obtenga una posición errónea para algún enemigo, en la reconstrucción se debe evitar que se duplique el mismo dentro de la lista mantenida en la *Escena Simulada*.

Atendiendo a todas las consideraciones expuestas anteriormente, se diseñó el algoritmo de reconstrucción de enemigos que se presenta en forma de *pseudo-código* en el Apéndice C. El funcionamiento del algoritmo es como sigue: revisa cada enemigo del arreglo proveniente desde la observación y lo compara, uno a uno, con los enemigos en la lista de la *Escena Simulada*. Para realizar la comparación, primero busca los enemigos en la lista que tengan el mismo tipo y estén en la misma posición. De no encontrar ninguno, hace otra verificación utilizando una función auxiliar que tiene los errores predecibles para cada tipo de enemigo existente. Si aún no consigue ningún enemigo que corresponda, se realiza una tercera verificación con una función que calcula errores comunes para cualquier tipo de enemigos. Si falla en este último intento, se asume que el enemigo del arreglo que se está comparando es nuevo y se crea un *Sprite* que lo represente, luego lo añade a la lista de enemigos de la *Escena Simulada*. En caso contrario, si encuentra al enemigo entonces lo declara como un enemigo existente cuya simulación fue exitosa, y procede a realizar el mismo proceso de comparación con el siguiente enemigo del arreglo.

Apéndice H

Codificación de la geometría del nivel

La información de los elementos dentro del nivel es codificada en *bytes* por el motor de simulación de la competencia antes de ser provista a los agentes mediante los métodos de la interfaz *Environment*. La codificación es como sigue:

Byte	Significado
0	Espacio vacío
1	Mario
2	Goomba
3	Goomba con alas
4	Red Koopa
5	Red Koopa volador
6	Green Koopa
7	Green Koopa volador
8	BulletBill
9	Spiky
10	Spiky volador
12	Enemy Flower
13	Shell
14	Mushroom
15	Fire Flower
16	Ladrillo
21	Bloque con bonus
34	Moneda

Cuadro H.1: Codificación de los elementos en la escena

Apéndice I

Las estructuras de datos

Como en cualquier algoritmo de búsqueda A^* , el algoritmo desarrollado cuenta con dos estructuras de datos críticas en su desempeño: La **lista de nodos abiertos** y la **lista de nodos cerrados**.

I.1. La lista de nodos abiertos

Esta lista fue implementada como una cola de prioridad basada en un *heap*, lo cual la hace ideal para cumplir su tarea principal con eficiencia: conseguir los elementos “más pequeños” de la lista de forma rápida.

En esta lista de nodos abiertos se ordenan todos sus elementos basados en su “orden natural”, el cual, para las *Escenas Simuladas*, fue implementado como la **Función de Selección** f explicada anteriormente. Cuando se le pide a la lista obtener el nodo “más pequeño”, devuelve la *Escena Simulada* que tenga el valor de f de menor magnitud.

I.2. La lista de nodos cerrados

Esta lista fue implementada como una lista enlazada. Cada elemento de dicho lista es una serialización de una *Escena Simulada*, esto es un arreglo de números reales de tamaño 4. Los valores de una serialización son los siguientes:

[posición en X de Luigi, posición en Y de Luigi, tick, costo acumulado]

La lista de nodos cerrados tiene como función tratar de señalarle al algoritmo de búsqueda cuáles *Escenas Simuladas* han sido previamente evaluadas, para no ser evaluadas de nuevo.

El último elemento de la serialización de una *Escena Simulada* es el costo acumulado de dicha escena. Este valor es utilizado al momento de comparar una *Escena Simulada* que este siendo evaluada por el algoritmo de búsqueda con una escena que se encuentre en la

lista de nodos cerrados. Si la escena que está siendo evaluada tiene un costo acumulado menor a la que ya se encuentra en la lista, la segunda escena es removida de la lista y se ingresan los valores de la primera. De lo contrario, la primera escena es descartada, se saca de la lista de abiertos y se sigue con las iteraciones del algoritmo de búsqueda.

Apéndice J

Tablas de rendimiento (en uso de procesador)

Mediante el uso de una herramienta de *profiling* llamada HProf ⁵ se consiguió identificar los métodos que consumían más tiempo de procesador, que se listan a continuación:

Agente sin simulación retrasada	
Uso de CPU (%)	Método o rutina
10.01 %	luigi.engine.LevelScene.tick
7.01 %	luigi.engine.Level.getBlock
0.50 %	luigi.engine.LevelScene.clone
0.43 %	luigi.simulation.SimulatedScene.f
Agente con simulación retrasada	
Uso de CPU (%)	Método o rutina
5.84 %	luigi.engine.LevelScene.tick
4.02 %	luigi.engine.Level.getBlock
1.26 %	luigi.simulation.SimulatedScene.f
1.20 %	luigi.engine.LevelScene.clone

Cuadro J.1: Métodos que consumen más tiempo de cómputo en el agente implementado

⁵<http://java.sun.com/developer/technicalArticles/Programming/HPROF.html>

Apéndice K

Experimentos y Resultados Auxiliares

Descripción de algunos experimentos y resultados de apoyo para realizar las pruebas mencionadas en el Capítulo 5.

K.1. Luigi para “Mario AI Competition”

K.1.1. Prueba preliminar sobre Luigi

En esta prueba se compara el rendimiento del agente implementado usando distintos valores para los *delta* de espacio y tiempo utilizados al momento de declarar nodos como cerrados. Como fue explicado en la Sección 3.7, se utilizan 3 *delta*: dos para las posiciones en el eje *X* y en el eje *Y* de *Luigi* respectivamente, y el último para la diferencia de ticks entre dos escenas.

Para las pruebas realizadas, el valor utilizado como delta de espacio para la posición del agente en algún eje fue el mismo utilizado para el otro eje, de esta forma se tomaron en cuenta combinaciones de variaciones del *delta* de espacio y el *delta* de tiempo.

Los experimentos demostraron que variar el *delta* de espacio a tamaños mayores a 1 no trae beneficios en el desempeño del agente, por lo ya explicado en la Sección 3.7. Sin embargo se incluye en los resultados a continuación los valores de una prueba del agente utilizando el valor 2 como *delta* de espacio.

De ahora en adelante se utilizará la siguiente notación para hacer referencia a configuraciones o variaciones de valores de los *delta* de Espacio(*e*) y Tiempo(*t*): e_i-t_j , donde *i* representa el tamaño en píxeles del *delta* de espacio, y *j* representa el tamaño en ticks de juego del *delta* de tiempo.

Las variaciones de los valores probados para *delta* de Espacio(*e*) y de Tiempo(*t*) fueron las siguientes: e1-t1, e1-t3, e1-t5, e1-t7, e2-t5.

Los resultados interesantes fueron obtenidos en los niveles que presentan enemigos.

	Niveles Completados (70)	Distancia Recorrida	Tiempo Restante
e1-t1	66	7921.7	8140
e1-t3	69	8116.91	8695
e1-t5	69	8116.91	8700
e1-t7	69	8116.91	8697
e2-t5	48	6451.06	6772

Cuadro K.1: Mediciones principales de pruebas preliminares para *Luigi* en niveles con enemigos

Los niveles pausados no mostraron diferencias significativas de desempeño. En el Cuadro K.1 se puede apreciar que la configuración con e1-t5 es la ganadora utilizando el segundo criterio de evaluación **Tiempo Restante**, debido a empatar con e1-t3 y e1-t7 en la **Distancia Recorrida**. Es importante notar que, como se especificó antes, el rendimiento de la configuración e2-t5 logra los peores resultados. A continuación se muestran los gráficos del desempeño de cada configuración.

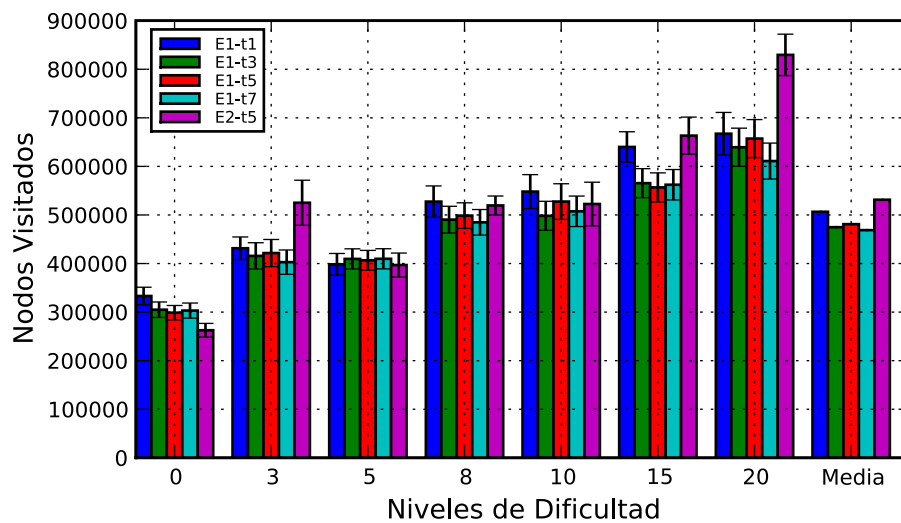


Figura K.1: Nodos visitados por niveles de dificultad jugados en las pruebas preliminares sobre *Luigi*

En las Figuras K.1 y K.2 se puede ver que, en promedio, la configuración que tiene menos metas no alcanzadas es la ganadora, lo cual indica que logra conseguir más caminos

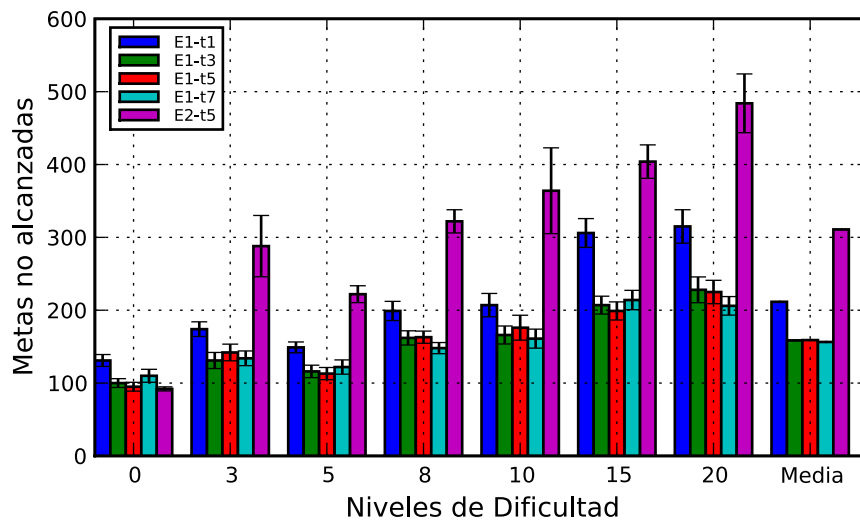


Figura K.2: Metas no alcanzadas por niveles de dificultad jugados en las pruebas preliminares sobre *Luigi*

óptimos para los ticks jugados.

También se puede apreciar que todas las configuraciones que utilizan un *delta* de espacio igual a 1pixel se comportan relativamente igual. El *delta* de tiempo parece no afectar de gran manera los resultados.

Por último, en el Cuadro K.2 se muestra el resto de los elementos de simulación que fueron medidos, donde una vez más la peor configuración resulta ser e2-t5. Con todos estos datos se pudo concluir que, con excepción de los primeros casos extremos e1-t1 y e2-t5, todas las configuraciones mostraron un rendimiento similar y por esto se usa únicamente la configuración e1-t5 (ganadora en **Tiempo Restante**) para realizar el resto de las pruebas.

K.1.2. *Luigi* vs. *Robin* con igual estrategia de selección de nodos cerrados

En la sección anterior se mostró el mérito del algoritmo de búsqueda implementado para *Luigi* en cuanto a la velocidad de búsqueda de caminos óptimos para la simulación.

	Enemigos Destruídos	Golpes Recibidos	Monedas Agarradas
e1-t1	574	4	3258
e1-t3	583	1	3242
e1-t5	575	3	3264
e1-t7	573	4	3244
e2-t5	411	42	281

Cuadro K.2: Comparación de objetivos de simulación en pruebas preliminares.

Este mérito viene dado por la diferencia de *deltas* de espacio y tiempo utilizados por ambos agentes para seleccionar nodos cerrados. Como fue explicado en la Sección 3.7, se utilizan 3 *delta*: dos para las posiciones en el eje *X* y en el eje *Y* de *Luigi* respectivamente, y el último para la diferencia de ticks entre dos escenas. En el Apéndice K (Sección K.1.1) se estableció que *Luigi* utiliza 1 *pixel* como *delta* de espacio y 5 *ticks* como *delta* de tiempo, mientras que *Robin* utiliza el doble de *delta* de espacio. Esto hace ver que *Luigi* explora, en promedio, el doble del espacio de juego de la simulación, comparado con su contrincante. Aún así, consigue alcanzar más metas.

En esta sección se estudia cuál sería el resultado de la comparación de los agentes si ambos buscan soluciones en un espacio de tamaño similar. A primera vista, un problema encontrado es el descrito en la Sección K.1.1, donde se puede ver que *Luigi* tiene un mal rendimiento utilizando la configuración de 2 *píxeles* para el *delta* de espacio, y 5 *ticks* para el de tiempo. Sin embargo, esto se debe a lo explicado previamente en la Sección 3.7: *Luigi* descarta cualquier nodo que haya encontrado como cerrado, por el contrario *Robin* no los descarta si no que los reingresa a la cola de abiertos añadiéndoles cierta penalidad. Para poder realizar una comparación justa entre ambos agentes, se varió en *Luigi* la estrategia de descartar los nodos cerrados, y se utilizó la misma que fue implementada por *Robin Baumgarten* en su agente.

Para las pruebas realizadas, el valor utilizado como *delta* de espacio para la posición del agente en algún eje fue el mismo utilizado para el otro eje, de esta forma se tomaron

en cuenta combinaciones de variaciones del *delta* de espacio y el *delta* de tiempo. De ahora en adelante se utilizará la siguiente notación para hacer referencia a configuraciones o variaciones de valores de los *delta* de Espacio(e) y Tiempo(t): $ei-tj$, donde i representa el tamaño en píxeles del *delta* de espacio, y j representa el tamaño en ticks de juego del *delta* de tiempo.

Se realizaron pruebas utilizando las siguientes configuraciones: e1-t1, e1-t3, e1-t5, e2-t3, e2-t5 y e4-t5. Los resultados son mostrados en el Cuadro K.3, se puede apreciar que *Luigi* gana en todos los experimentos excepto en el primero. Sin embargo se obtiene un resultado relevante en cuanto a su rendimiento: Al igual que los experimentos realizados en la Sección K.1.1, usando la estrategia de reinsertar los nodos cerrados a la lista de nodos abiertos, se ve que la configuración que mejor beneficia al agente es e1-t5, pues obtiene el mejor **Tiempo Restante**.

<i>Luigi</i>			
	Niveles Completados (70)	Distancia Recorrida (8135.84)	Tiempo Restante
e1-t1	66	7851.42	8284
e1-t3	70	8135.84	8680
e1-t5	70	8135.84	8684
e2-t3	70	8135.84	8536
e2-t5	70	8135.84	8540
e4-t5	44	5810.38	7290
<i>Robin</i>			
	Niveles Completados (70)	Distancia Recorrida (8135.84)	Tiempo Restante
e1-t1	70	8135.84	8609
e1-t3	69	8133.43	8629
e1-t5	68	8048.65	8647
e2-t3	67	7970.93	8696
e2-t5	69	8113.76	8600
e4-t5	39	5751.56	9741

Cuadro K.3: Resultados de simulación para variaciones de *deltas* de selección de nodos cerrados entre *Luigi* y *Robin*.

K.2. *Luigi vs. Luigi*

Por último se decidió realizar pruebas sobre las dos estrategias utilizadas para tratar a los nodos cerrados en el agente implementado. A continuación se muestran los resultados de haber variado el Factor de Parcialidad y la Restricción de Tiempo Disponible, midiendo el rendimiento de la estrategia de excluir los nodos ya cerrados, la cual será llamada “Exc” y la estrategia de volver a incluirlos dentro de la lista de nodos abiertos colocándoles cierta penalización, la cual será llamada “Inc”. Ambas estrategias fueron configuradas con *delta* de espacio = 1 pixel y *delta* de tiempo = 5, por los resultados obtenidos en las Secciones ?? y K.1.2.

K.2.1. Variaciones del Factor de Parcialidad

Los valores probados en estas pruebas son los mismos de la Sección 5.1.2.1. Los resultados mostrados en el Cuadro K.4 ofrecen una evidencia similar a las obtenidas en las pruebas anteriores. La estrategia de excluir los nodos cerrados resultó perdedora en las variaciones **0.7** y **0.8**, pero ganadora en las restantes.

Una vez más se puede ver que la mejor variación de Factor de Parcialidad para ambas estrategias es **0.9**, y el mejor **Tiempo Restante** se obtuvo con la estrategia “Exc”.

En las Figuras K.3, K.4 y K.5 se puede apreciar que la diferencia de resultados entre ambas estrategias varía muy poco.

En la Figura K.4 se ve que la estrategia “Exc” obtuvo mejores resultados, al tener menos cantidad de **Metas no Alcanzadas**. Ésto es consistente con los resultados mostrados en la Figura K.3, debido a que al explorar un espacio del juego idéntico (ambos utilizan la misma configuración para establecer nodos cerrados), la estrategia que consigue alcanzar correctamente más metas debe ser aquella que exploró de mejor forma el espacio de búsqueda y obtuvo las soluciones más rápido, teniendo que visitar menos nodos.

La Figura K.5 muestra que la estrategia “Exc” también resultó ser, por poco, más efectiva en el **Tiempo Promedio de Respuesta**. No obstante es importante señalar el hecho

<i>“Exc”</i>			
	Niveles Completados (70)	Distancia Recorrida (8135.84)	Tiempo Restante
0.7	69	8116.91	8655
0.8	68	8094.62	8680
0.9	70	8135.84	8695
1.0	70	8135.84	8610

<i>“Inc”</i>			
	Niveles Completados (70)	Distancia Recorrida (8135.84)	Tiempo Restante
0.7	70	8135.84	8616
0.8	70	8135.84	8659
0.9	70	8135.84	8682
1.0	70	8135.84	8638

Cuadro K.4: Resultados de simulación para variaciones de Factor de Parcialidad entre las estrategias “Exc” e “Inc”.

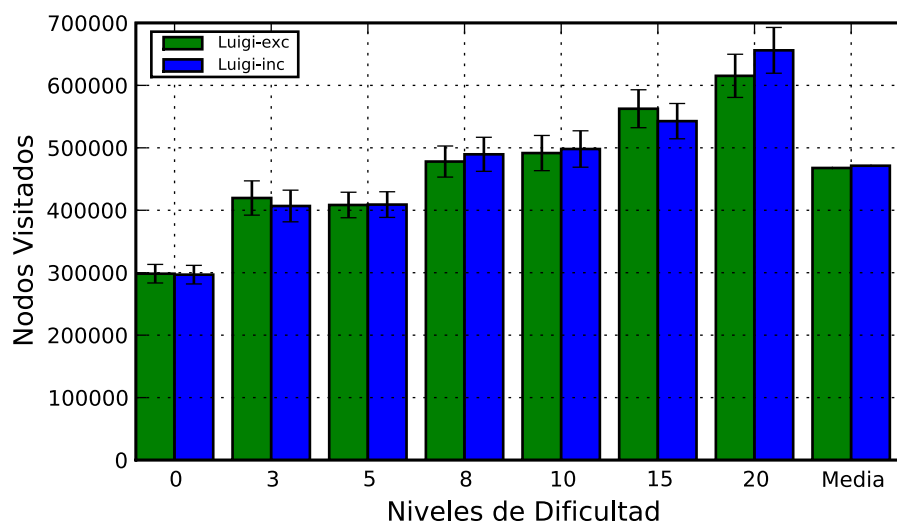


Figura K.3: Nodos Visitados por nivel de dificultad jugados en la variación 0.9 de las pruebas realizadas para las estrategias “Exc” e “Inc”

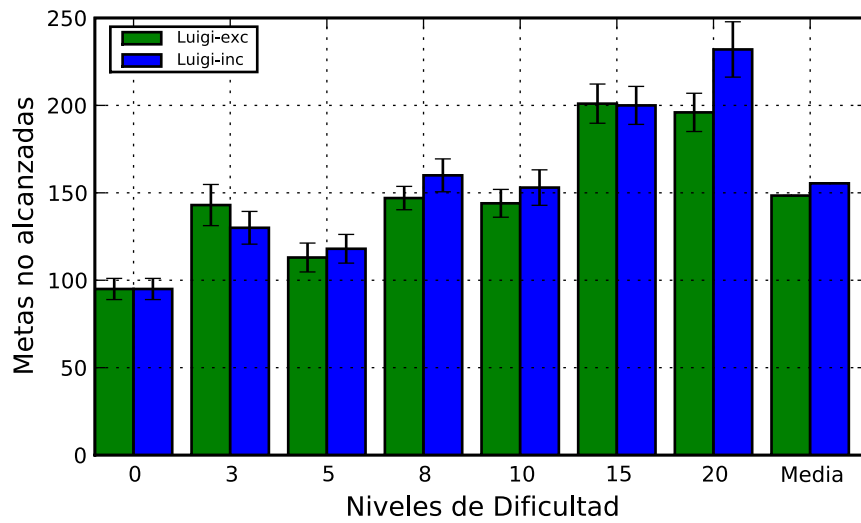


Figura K.4: Metas no alcanzadas por nivel de dificultad jugados en la variación 0.9 de las pruebas realizadas para las estrategias "Exc" e "Inc"

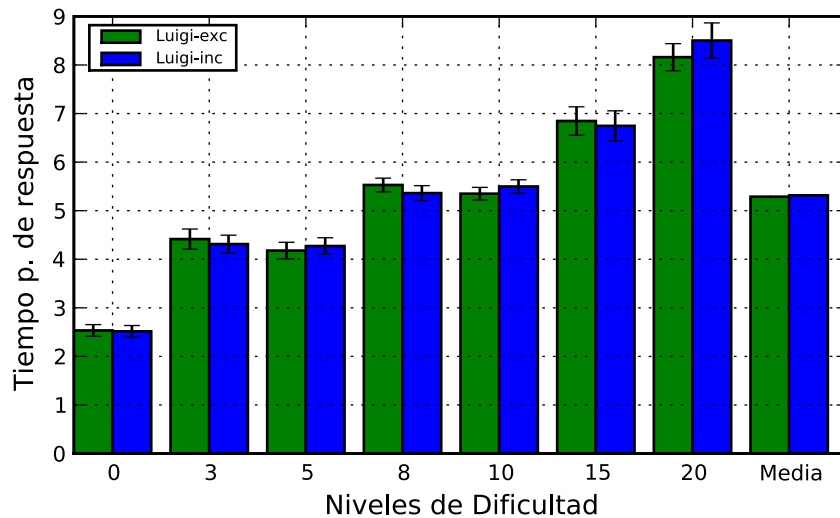


Figura K.5: Tiempo promedio de respuesta por nivel de dificultad jugados en la variación 0.9 de las pruebas realizadas para las estrategias "Exc" e "Inc"

de que en la mayor dificultad jugada, la estrategia “Inc” tuvo un mejor comportamiento, lo cual también es reflejado anteriormente en la Figura K.4.

K.2.2. Variaciones en la Restricción de Tiempo

Una vez más, las variaciones son las mismas establecidas en la Sección ???. Los resultados mostrados en el Cuadro K.5 son, como se esperaba, similares entre sí. La estrategia “Exc” presenta mejores resultados en el **Tiempo Restante** de simulación en todos las variaciones realizadas menos la de **20ms** donde no consigue completar uno de los niveles.

<i>Exc</i>			
	Niveles Completados (70)	Distancia Recorrida (8135.84)	Tiempo Restante
20ms	69	8116.91	8694
40ms	69	8116.91	8692
80ms	70	8135.84	8690
160ms	70	8135.84	8698
320ms	70	8135.84	8681
<i>Inc</i>			
	Niveles Completados (70)	Distancia Recorrida (8135.84)	Tiempo Restante
20ms	70	8135.34	8676
40ms	70	8135.84	8684
80ms	70	8135.84	8684
160ms	70	8135.84	8688
320ms	70	8135.84	8572

Cuadro K.5: Resultados de simulación para variaciones de Restricción de Tiempo entre las estrategias “Exc” e “Inc”.

El agente implementado usando la estrategia “Inc” obtiene los mejores resultados en la variación de **160ms**, al igual que la estrategia “Exc”. Los resultados de **Nodos Visitados**, **Metas no Alcanzadas** y **Tiempo Promedio de Respuesta** son muy parecidos a los exhibidos anteriormente para las variaciones de Factor de Parcialidad (Sección K.2.1).

Tras realizar ambos experimentos se pudo concluir que la principal diferencia entre el rendimiento de ambas estrategias fue que al incluir de nuevo los nodos que son cerrados, añadiéndoles cierta penalidad, el agente completó todos los niveles que jugó. Por el contrario, la estrategia “Exc” sólo completa todos los niveles en algunas variaciones.

Sin embargo es importante destacar que “Exc” presenta un mejor rendimiento en cuanto al **Tiempo Restante**, lo cual es una ventaja en el segundo criterio de comparación de la competencia.

K.3. Rendimiento de *Luigi* utilizando aprendizaje

A continuación se muestran gráficas con los resultados obtenidos en los criterios de evaluación de agarrar monedas y destruir enemigos. Los resultados son obtenidos de un promedio de 20 corridas sobre las configuraciones de mundo descritas.

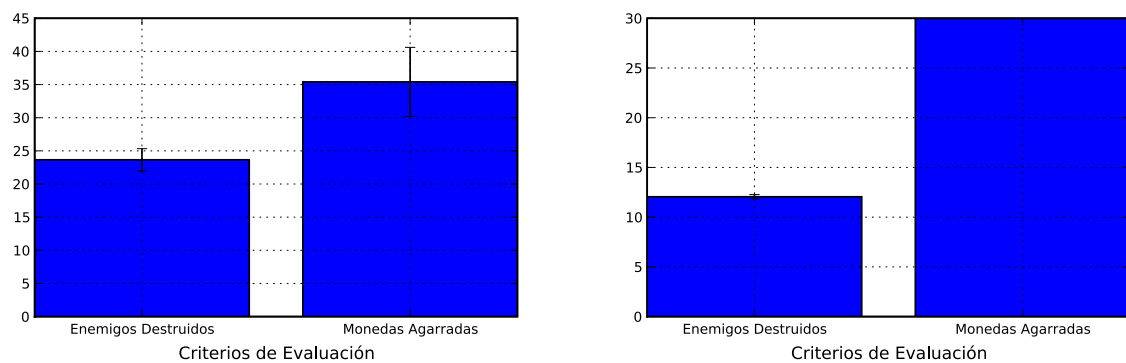


Figura K.6: Rendimiento de *Luigi* con aprendizaje vs. *Luigi* de la Competencia

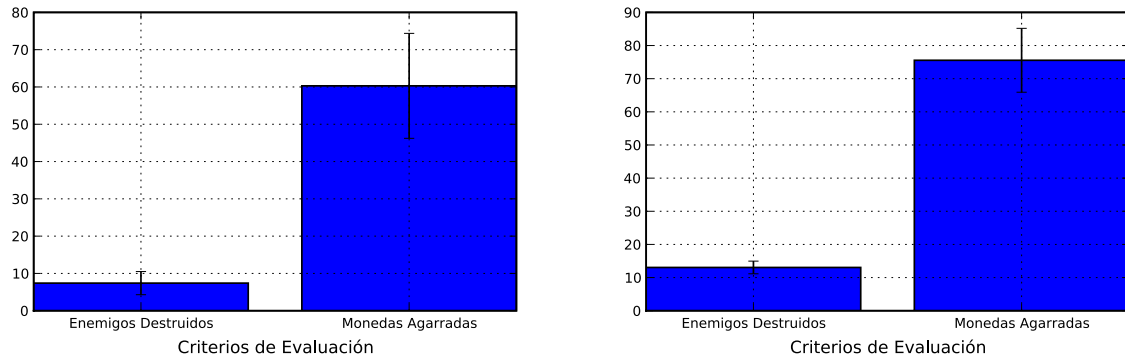


Figura K.7: Rendimiento de *Luigi* sobre ambientes aleatorios desconocidos

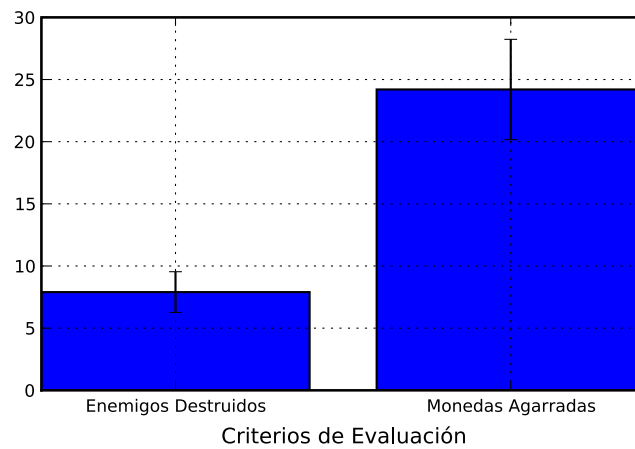


Figura K.8: Rendimiento de *Luigi* al jugar un nivel aleatorio corto después de ser entrenado en un nivel corto

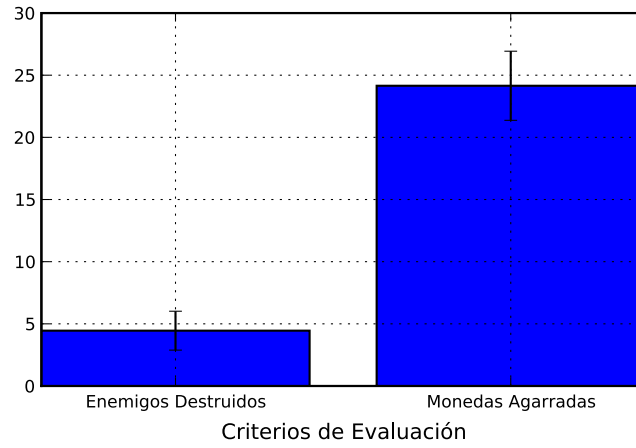


Figura K.9: Rendimiento de *Luigi* al jugar un nivel aleatorio largo después de ser entrenado en un nivel corto

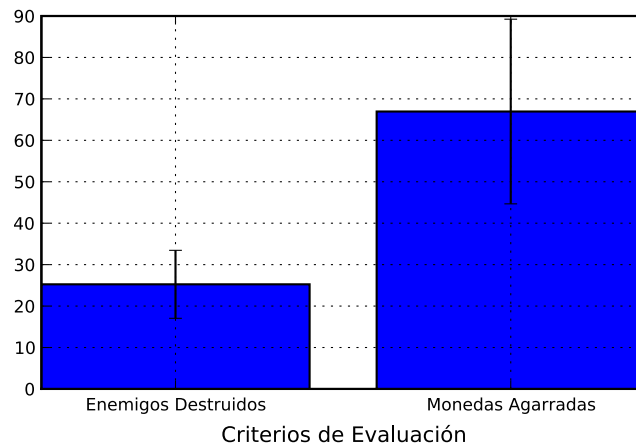


Figura K.10: Rendimiento de *Luigi* al jugar un nivel aleatorio corto después de ser entrenado en un nivel largo

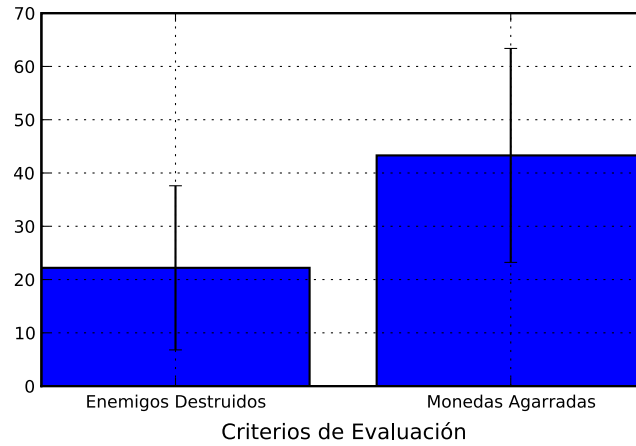


Figura K.11: Rendimiento de *Luigi* al jugar un nivel aleatorio largo después de ser entrenado en un nivel largo

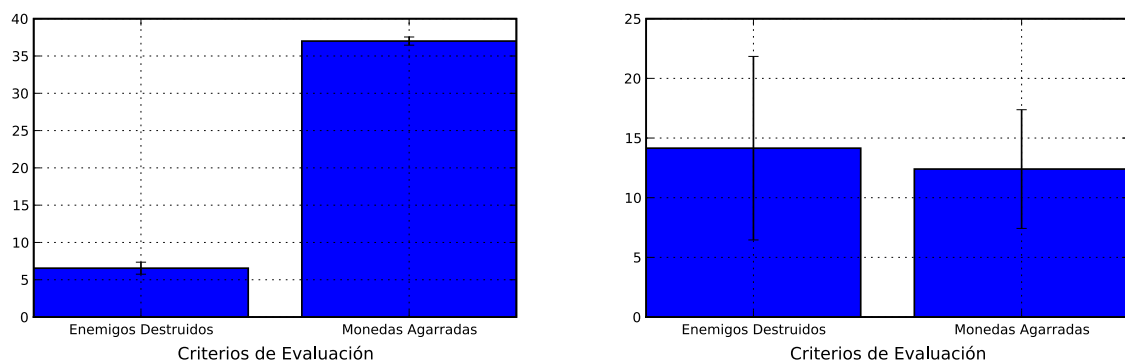


Figura K.12: Rendimiento de *Luigi* utilizando aprendizaje selectivo (favoreciendo monedas y enemigos respectivamente) sobre un nivel aleatorio