PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA INFORMÁTICA

MODELADO Y RESOLUCIÓN DEL "MARIO AI BENCHMARK" UTILIZANDO PROGRAMACIÓN CON RESTRICCIONES

RODRIGO IGNACIO MUÑOZ CORNEJO

TESIS DE GRADO
MAGÍSTER EN INGENIERÍA INFORMÁTICA

PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA INFORMÁTICA

MODELADO Y RESOLUCIÓN DEL "MARIO AI BENCHMARK" UTILIZANDO PROGRAMACIÓN CON RESTRICCIONES

RODRIGO IGNACIO MUÑOZ CORNEJO

Profesor Guía: **DR. RICARDO SOTO**

Programa: **Magíster en Ingeniería Informática**

Resumen

Super Mario Bros es un famoso videojuego producido por Nintendo, cuyo objetivo principal es conducir a Mario por distintas etapas sorteando una serie de obstáculos y enemigos con el objetivo final de rescatar a la princesa Peach. El propósito de este proyecto es simular el comportamiento de Mario de tal manera que supere la mayor cantidad de etapas dentro del juego. Esto se realiza por medio de la programación con restricciones donde cada movimiento de Mario se modela como un problema de satisfacción de restricciones. El sistema se implementó utilizando los solvers Choco y Gecode, una versión de código abierto de Super Mario Bros denominada Mario AI Benchmark y una interfaz que permite la comunicación entre ambos componentes.
Palabras Clave: *Programación con restricciones, Problemas de Satisfacción de restricciones, Video Juegos, Mario AI Benchmark.*

Abstract

Super Mario Bros is a famous video-game produced by Nintendo, whose goal is to drive Mario through different levels avoiding obstacles and enemies in order to rescue the pricess Peach. The purpose of this project is to simulate the behavior of Mario so as to pass as much levels as possible. This is done by using constraint programming where each Mario move is modeled as a constraint satisfaction problem. The system has been implemented by using the Choco and Gecode Solvers, an open source version of Super Mario Bros called Mario AI Benchmark, and an interface that allow the communication between both components.
Keywords: *Constraint Programming, Constraint Satisfaction Problems, Video Games, Mario AI Benchmark.*

Índice

Lista de Figuras

Lista de Tablas

1. Introducción

Super Mario Bros es un famoso videojuego producido por Nintendo, cuyo objetivo principal es conducir a Mario por distintas etapas sorteando una serie de obstáculos y enemigos con el objetivo final de rescatar a la princesa Peach. Dada la inherente complejidad de este juego, Super Mario Bros ha sido utilizado regularmente como plataforma para la experimentación y desarrollo de algoritmos en el área de la inteligencia artificial. En particular, dos problemáticas han sido comúnmente tratadas en artículos científicos: la creación automática de niveles del juego y el control de Mario por medio de un agente.

Este proyecto se centra en la segunda problemática, esencialmente en la creación de un agente que sea capaz de controlar a Mario en forma autónoma utilizando una técnica ampliamente utilizanda en el campo de la optimización denominada programación restricciones. Esto es posible mediante el modelado del problema bajo la forma de un problema de satisfacción de restricciones (CSP en inglés), el cual está compuesto por una secuencia de variables asociadas a un dominio y un conjunto de restricciones sobre esas variables. En la práctica, los posibles movimientos de Mario en una posición dada están representados por variables y las restricciones definen los lugares a los que puede acceder Mario evitando morir, como por ejemplo al caer por un acantilado o ser atacado por algún enemigo.

Asimismo, cada movimiento de Mario es modelado como un CSP, por lo cual la resolución de una etapa completa implica la resolución de tantos CSP como movimientos requiera Mario para alcanzar el final de la etapa. La resolución dinámica de esta secuencia de CSPs se controla por medio de una interfaz que es capaz de comunicar eficientemente el motor de resolución del CSP (comúnmente conocido como Solver) con el juego. Se implementaron dos agentes para Super Mario Bros, uno utilizando el solver Choco y otro por medio del solver Gecode. Ambos agentes utilizan una versión de código abierto de Super Mario Bros denominada Mario AI Benchmark y la interfaz

que permite la comunicación entre ambos componentes.

Este documento se organiza de la siguiente manera. En el capítulo 2 se plantean los objetivos del proyecto, en el capítulo 3 la metodología utilizada y en el capítulo 4 el estado del arte de la problemática. En el capítulo 5 se realiza una introducción a CP incluyendo los algoritmos clásicos de búsqueda. El capítulo 6 presenta la problemática a tratar y la implementación de la solución propuesta. El documento finaliza con las conclusiones y el trabajo futuro.

2. Definición de Objetivos

2.1. Objetivo General

- Modelar y resolver el Mario AI Benchmark utilizando programación con restricciones

2.2. Objetivos Específicos

- Modelar el Mario AI Benchmark como un CSP.
- Implementar el CSP en los solvers Gecode y Choco.
- Implementar una interfaz que comunique el solver con la interfaz gráfica de Mario AI Benchmark.
- Realizar experimentos con la solución propuesta y analizar sus resultados.

3. Metodología

La metodología de investigación utilizada en esta tesis cae dentro del marco de la investigación exploratoria, la cual se centra principalmente en examinar un tema poco estudiado o que no ha sido abordado con anterioridad. En particular la resolución de Mario AI benchmark ha sido abordada con diversas técnicas de búsqueda (lo cual se detalla en mayor grado en la sección estado del arte), sin embargo el problema no ha sido resuelto utilizando programación con restricciones, menos utilizando el enfoque de resolución dinámica consecutiva de CSPs.

De esta forma al ser una investigación exploratoria requiere de la recopilación, lectura y análisis de artículos científicos relacionados con el tema. Lo cual tiene como finalidad aumentar la familiaridad de un tema poco conocido, y como consecuencia tener una base conceptual con mayor solidez para futuras investigaciones.

4. Estado del Arte

En el marco de la Mario AI Competition [25], que se realiza anualmente desde el año 2009, se han presentado diversos agentes que buscan resolver el problema logrando distintos resultados. Sin embargo, hasta el momento no se ha presentado una solución que utilice programación con restricciones. Dentro de las competencias efectuadas, los agentes para Mario AI Benchmark propuestos han utilizado las siguientes técnicas de resolución: algoritmos de búsqueda A*, basados en reglas, algoritmos genéticos, redes neuronales, máquinas de estado, y heurísticas específicas construidas "a mano".

Cabe destacar que de las soluciones presentadas, las que mejores resultados obtuvieron fueron las basadas en A*, seguidas de las basadas en reglas. En este sentido, las que aplicaron el algoritmo A* presentaban el problema como un grafo en el cual buscaban el camino más corto para llevar a Mario desde su posición actual hasta el lado derecho de la etapa. Mientras que las basadas en reglas se basaban más bien en consultar por las condiciones dadas por el entorno para saber qué acción debía ser realizada por Mario.

A pesar de no existir publicaciones científicas para cada una de las técnicas presentadas para Mario AI, Karakovskiy y Togelius en [7, 25, 6] se recogen los principales caraterísticas de las competencias incluyendo las técnicas propuestas.

5. Programación con Restricciones y Técnicas de Optimización

La programación con restricciones, *Constraint Programming* (CP) en inglés, es un paradigma de la programación informática y una reciente tecnología de software, dedicada a la resolución de problemas de satisfacción de restricciones, en inglés *Constraint Satisfaction Problem* (CSP), y problemas de optimización con restricciones, *Constraint Satisfaction Optimization Problem* (CSOP).

Si un problema puede ser modelado desde un punto de vista matemático y éste involucra una serie de reglas, entonces puede aplicarse programación con restricciones para llegar a una solución. De este modo, este nuevo paradigma de programación se centra en cómo modelar la solución para obtener una solución y no en el cómo desarrollar una solución. Así, este paradigma puede aplicarse a una serie de problemas siempre y cuando estos puedan ser modelados como un CSP.

5.1. Definición Formal de un CSP

Un problema de Satisfacción de Restricciones $\mathcal{P}$ se define por una 3-tupla $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ donde:

- $\mathcal{X}$ es una n-tupla de variables $\mathcal{X} = \langle x_1, x_2, ..., x_n \rangle$,
- $\mathcal{D}$ son los correspondientes n-tuplas de dominios $\mathcal{D} = \langle D_1, D_2, ..., D_n \rangle$ tal que $x_i \in D_i$, y D_i es el conjunto de valores, para $i = 1, ..., n$.
- $\mathcal{C}$ es una t-tupla de restricciones $\mathcal{C} = \langle C_1, C_2, ..., C_t \rangle$.

Los CSP también pueden ser definidos mediante grafos, donde cada nodo representa una variable, etiquetada con su dominio, y las aristas representan las restricciones. El siguiente CSP

$\mathcal{P} = \langle\langle x, y, w, z\rangle, \langle D_x \in \{0,1\}, D_y \in \{1,2\}, D_w \in \{2,3\}, D_z \in \{3,4\}\rangle, \langle C_1 = \{x \leq y\}, C_2 = \{y \leq w\}, C_3 = \{w \leq z\}, C_4 = \{y + 2 \leq z\}\rangle\rangle$

es representado como un grafo en la Figura 1.

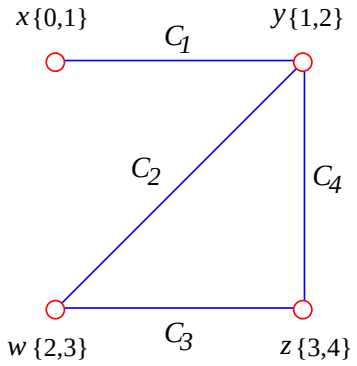

Figura 1: Un CSP representado por un grafo.

Los CSP clásicos apuntan a encontrar la primera solución que satisfaga todas las restricciones. Esta definición clásica puede ser extendida para representar problemas de optimización, los cuales se enfocan en encontrar la mejor solución o las mejores según sea el caso, según una condición dada. Bajo el esquema de satisfacción de restricciones, los problemas de optimización son llamados problemas de optimización con restricciones, *Constrain Optimization Problem* (COP), los cuales pueden ser definidos por una 4-tupla $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{O}\rangle$ donde O es la función objetivo correspondiente al criterio dado que debe ser maximizado o minimizado.

Un CSP en general puede también especializarse en otras categorías por ejemplo, dependiendo de los valores de los dominios. Como por ejemplo, un dominio finito de CSP apunta a un CSP que involucra sólo valores enteros, un CSP numérico sólo a problemas con números reales. Otros ejemplos son CSP Booleanos y Simbólicos, estos últimos consideran valores no numéricos. Para una revisión extendida de las categorías de CPS refiérase a [10, 23]

Figura 2: Solución del problema de las 4-reinas usando GT.

A continuación se ilustra el proceso GT bajo los términos del problema de las 4-reinas. La Figura 2 muestra un extracto del proceso realizado por el algoritmo GT para alcanzar un resultado para este problema. En lo que respecta a la figura es fácil ver que el procedimiento GT es muy ingenuo, las restricciones son probadas siempre con todas las variables instanciadas(al nivel más bajo del árbol), entonces si una solución parcial viola una restricción no puede ser detectada tan pronto como las variables involucradas en la solución parcial han sido instanciadas. Esto lleva a la generación de varias tareas incorrectas que podrían ser detectadas precozmente. Este enfoque es simple de implementar, sin embarg el costo de búsqueda es muy

alto, de hecho es proporcional al número de potenciales soluciones, que en muchos problemas tienden a crecer tan rápido como el tamaño del problema aumenta. Como consecuencia, el uso de GT es factible sólo para problemas de tamaño reducido.

5.3. Backtracking

Backtracking (BT) es otro enfoque para realizar búsquedas sistemáticas. En el método BT las soluciones potenciales son generadas de manera creciente mediante repetición eligiendo un valor por otra variable y en cuanto todas las variables relevantes a una restricción son instanciadas, la validez de la restricción es verificada. Entonces, si una solución parcial viola alguna de las restricciones, el algoritmo regresa a la variable instanciada más reciente que aún mantiene alternativas disponibles, por consecuencia eliminando el subespacio conflictivo.

La Figura 3 grafica el proceso de búsqueda realizado por el procedimiento BT en el problema de las 4-reinas. Aquí, podemos ver que BT permite detectar fallos tan pronto como 2 variables son instanciadas (al nivel medio del árbol), mucho antes que en el enfoque GT. A pesar de esto, el enfoque BT no permite detectar fallas antes de asignar los valores a todas las variables involucradas en una restricción conflictiva. Este problema puede ser manejado mediante la aplicación de consistencias locales.

5.4. Técnicas de Consistencia

Las Técnicas de Consistencia son mecanismos de filtro que apuntan a mejorar el rendimiento de los procesos de búsqueda intentando reducir el espacio de búsqueda. Esta reducción es posible por medio de un proceso denominado propagación de restricciones donde un nivel determinado de consistencia entre subgrupos de variables es forzado. Este nivel de consistencia es llamado de manera técnica “Consistencia Local” y existen varios tipos,

Figura 3: Resolviendo el problema de las 4-reinas utilizando BT.

5.4.1. Consistencia de Nodos

La Consistencia de Nodos es la más simple de las consistencias locales y trata con restricciones unarias. Un CSP es nodo-consistente, si para cada valor en el dominio actual de una variable x, cada restricción unaria en x es satisfecha. Si el dominio de una variable x contiene valores que no satisfacen la restricción unaria que involucra V, la consistencia de nodos puede ser eliminada simplemente removiendo los valores desde el dominio de x que no satisfaga la restricción unaria en V.

Por ejemplo el CSP compuesto por la restricción $x > 1$ sobre el dominio $D_x = [1.,4]$ de x no es nodo-consistente, debido a que el valor 1 no satisface la restricción.

5.4.2. Consistencia de Arcos

La Consistencia de Arcos trata con restricciones binarias. Una restricción binaria que involucra una variable x y una variable y es consistencia de arcos cada valor en el dominio de x e y participan en la solución. Un CSP es arco-consistente si todas sus restricciones binarias son arco-consistente.

Por ejemplo, el CSP mostrado en la Figura 4, que consiste en una restricción $x < y$ sobre los dominios $D_x = [3.,6]$ de x y $D_y = [4.,6]$ de y no es consistente de arco ya que el valor 6 en D_x no participa en ninguna solución.

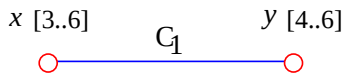

Figura 4: Un CSP no arco-consistente.

Se remarca que arco-consistencia no implica consistencia, así como consistencia no implica arco-consistencia. Tomando como ejemplo el CSP $\mathcal{P} = \langle\langle x, y\rangle, \langle D_x \in \{0,1\}, D_y \in \{0,1\}\rangle, \langle x = y, x \neq y\rangle\rangle$ graficado en la Figura 5, que es arco-consistente, pero inconsistente; y el CSP $\mathcal{P}\langle\langle x, y\rangle, \langle D_x \in \{0,1\}, D_y = \{0\}\rangle, \langle x = y\rangle\rangle$ ilustrado en la Figura 6, que es consistente, pero no es arco-consistente.

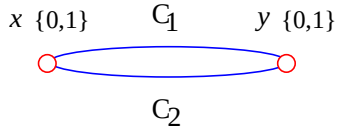

Figura 5: Un CSP arco-consistente, pero inconsistente.

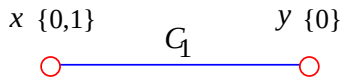

Figura 6: Un CSP consistente, pero no arco-consistente.

La propagación de restricciones puede hacer el problema arco-consistente mediante la eliminación de los valores para los cuales no hay valor correspondiente en el otro dominio de manera que las restricciones binarias son satisfechas. Este proceso podría considerar un par dado de variables más de una vez, ya que al remover valores del dominio de una variable podría causar que otras variables dejen de ser arco-consistente con ella. Estas preocupaciones han llevado al diseño de muchos algoritmos para forzar la consistencia de arcos. Por ejemplo AC-2 [28], AC-3 [14] y AC-4 [16]. Actualmente las consistencias de arco son las consistencias locales más usadas en sistemas de satisfacción de restricciones.

5.4.3. Consistencia de Hiper Arcos

La noción de la arco-consistencia puede ser generalizada en la hiper-arco-consistencia. En efecto, una restricción es hiper-arco-consistente si es que en cada dominio involucrado, cada elemento de éste participa en una solución.

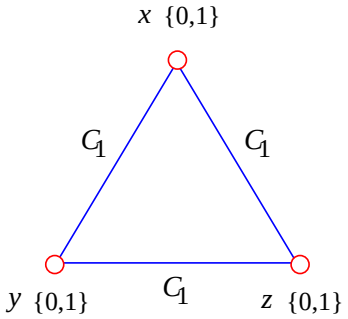

Figura 7: Un CSP hiper-arco-consistente.

Por ejemplo, el CSP Booleano $\mathcal{P} = \langle\langle x, y, z\rangle, \langle D_x \in \{0,1\}, D_y \in \{0,1\}, D_z \in \{0,1\}\rangle, \langle x \vee y = z\rangle\rangle$ mostrado en la Figura 7 es hiper-arco-consistente, ya que cada elemento de cada dominio participa en una solución. En contraste, el CSP $\mathcal{P} = \langle\langle x, y, z\rangle, \langle D_x \in \{1\}, D_y \in \{1\}, D_z \in \{0,1\}\rangle, \langle x \Rightarrow y = z\rangle\rangle$ mostrado en la Figura 8 no es hiper-arco-consistente ya que el valor 0 de D_z

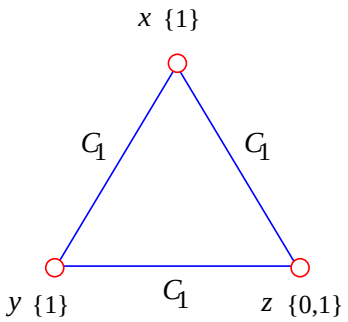

Figura 8: Un CSP no hiper-arco-consistente.

no participa en ninguna solución.

También existen maneras más fuertes de aplicar filtros, que podrían eliminar un número aún mayor de valores conflictivos en los dominios, pero a costos más altos en términos de cálculos. Algunos ejemplos son la consistencia de ruta y la k-consistencia. Para una presentación detallada de todas las técnicas de consistencia vea [10].

5.5. Algoritmos de Satisfacción de Restricciones

Los algoritmos de búsqueda sistemática y las técnicas de consistencia raramente son usados solos. El enfoque más común es combinar el algoritmo BT con una técnica de consistencia. De esta manera, el algoritmo intenta reducir el espacio de búsqueda aplicando una técnica de consistencia después que un valor sea asignado a una variable. Dependiendo del grado de la técnica de consistencia usada, existen varios algoritmos de satisfacción de restricciones.

El enfoque backtracking realiza en cierto sentido un tipo de técnica de consistencia debido a que comprueba la validez de las restricciones considerando las instanciaciones parciales, lo que puede ser visto como la aplicación de una fracción de una consistencia de arco entre variables ya instanciadas. En este enfoque la consistencia es aplicada a las variables instanciadas, por lo

que no es posible reducir el espacio de búsqueda antes de esta instanciación.

5.5.1. Forward Checking

El algoritmo Forward Checking (FC) hace frente a esta preocupación. Forward checking permite prevenir conflictos futuros forzando arco-consistencia a las variables aún no instanciadas. Esto es realizado mediante la eliminación temporal de valores de variables que causarán conflictos con la asignación actual de la variable. Por lo tanto, el algoritmo sabe inmediatamente que la actual solución parcial es inconsistente y consecuentemente el espacio de búsqueda puede ser podado antes de utilizar sólo backtracking.

La Figura 9 ilustra este proceso, donde los valores desde los dominios son eliminados desde el segundo nivel del árbol. Una vez que una reina es instanciada, sus futuros valores conflictivos son temporalmente removidos, de manera que las reinas instanciadas en la posición (1,1) elimina todos los valores correspondientes a la primera fila y a la diagonal NO-SE (indicados por las flechas azules). Entonces, en el subárbol izquierdo, la segunda reina es colocada en la posición (3,2) que es inmediatamente establecida como inconsistente ya que no deja espacio disponible para la tercera reina. La propagación sigue para cada reina instanciada en el tablero de ajedrez, permitiendo evitar la mayoría de declaraciones incorrectas realizadas mediante el enfoque BT.

5.5.2. Mantaining Arc Consistency

El proceso realizado mediante forward checking puede ser aún más ambicioso, por ejemplo si también se verifican los conflictos entre variables futuras (en adición a la prueba entre la variables actuales y futuras). Este enfoque se denomina Full Look Ahead o Maintaining Arc Consistency (MAC).

La Figura 10 ilustra este proceso, donde podemos ver que el algoritmo MAC permite reducir el espacio de búsqueda antes que el forward checking, pero realizando un trabajo mayor en cada asignación de variables. Por ejem-

Figura 9: Resolviendo el problema de las 4-reinas utilizando FC.

plo, cuando la primera reina es ubicada en la posición (1,1) los conflictos en la posición actual y las ubicaciones futuras son removidas (indicadas con las flechas azules). Después de eso, el algoritmo pasa a la siguiente celda valida(3,2), y realiza el mismo proceso. El algoritmo encuentra que la posición (3,2) es inconsistente ya que no deja un lugar disponible para la tercera reina (flechas naranjas), por lo tanto la posición (3,2) es eliminada. El algoritmo sigue con la celda (4,2), la próxima posición disponible en la segunda columna. Esta ubicación (flechas verdes) dejan la celda (2,3) como la única posición disponible en la tercera columna que es entonces establecida como una inconsistencia ya que no deja un lugar disponible para la cuarta reina (flechas rojas). Este proceso sigue hasta que el resultado es logrado en el subárbol de la derecha.

Figura 10: Resolviendo el problema de las 4-reinas utilizando MAC.

5.6. Heurísticas de Ordenación de Variables y Valores

En los ejemplos previos se ha comenzado el proceso de búsqueda seleccionando una variable (la primera reina) y entonces se le asignó un valor a su dominio (el primer valor). El orden en el cual esta elección es realizada son referidas tanto a la ordenación de la variable, así como del valor. Varios experimentos han demostrado que una decisión de ordenación correcta para las variables así como para los valores puede ser crucial para realizar un proceso de resolución efectivo [10].

La ordenación de los valores y variables puede ser estático o dinámico. En la ordenación estática el orden es especificado antes que comience la búsqueda, y en la ordenación dinámica la selección de la próxima variable depende de la información del estado actual de la búsqueda. La ordenación dinámica no puede ser aplicada a cualquier algoritmo de búsqueda, por ejemplo al usar simple backtracking no hay información extra durante la búsqueda que permita cambiar la ordenación inicial definida. Al contrario, utilizando forward checking, los dominios de otras variables podrían cambiar durante una instanciación dando información adicional que podría utilizarse para elegir la próxima variable.

5.6.1. Ordenación de Variables

Existen varias heurísticas para seleccionar la ordenación de variables. Por ejemplo, las más comunes son:

- Seleccionar la variable con el dominio más pequeño:
  Esta elección es motivada por la suposición que un éxito puede lograrse al intentar primero las variables que tengan una probabilidad mayor de fallar, en este caso, los valores con un número menor de alternativas disponibles.

- Seleccionar la variable más restringida:
  Esta elección puede ser justificada por el hecho de que la instanciación de una variable debería llevar a una reducción mayor del árbol a través de la propagación de restricción.

5.6.2. Ordenación de Valor

El orden en el cual los valores son seleccionados también pueden tener un impacto considerable. Por ejemplo, si el valor correcto es elegido al primer intento para cada variable, una solución puede ser encontrada sin realizar ningún backtrack. De todos modos, si el CSP es inconsistente o el grupo completo de soluciones es requerido, la ordenación de valor es irrelevante. La literatura presenta algunas maneras obvias de realizar esta selección que, dependiendo del problema, podrían derivar a una propagación de restricciones más eficiente [10]:

- Selecciona el valor más pequeño.

- Selecciona el valor medio.

- Selecciona el valor máximo.

También existen más heurísticas complejas de ordenación de valor que en general están basadas ya sea en la estimación del número de soluciones o estimando la probabilidad de una solución [23].

5.7. Optimización Global

Cuando se resuelven problemas bajo restricciones puede existir el interés en buscar la mejor solución, en vez de buscar la primera. En este contexto, se debe realizar mucho más procesamiento para conseguir el objetivo, ya que el proceso de búsqueda debe verificar varias soluciones. En los siguientes párrafos, se otorga un resumen de los enfoques principales para una optimización global.

5.8. Programación Matemática

La programación matemática es una rama de las matemáticas aplicadas y análisis numérico dedicada a hacer frente a los problemas de optimización. Las técnicas para afrontar con la programación matemática dependen tanto de la naturaleza de la función objetivo como del conjunto de restricciones. Algunos ejemplos son la programación entera, programación lineal, programación no-lineal [17, 4, 24].

5.8.1. Programación Lineal

La programación lineal estudia el caso en el cual la función objetivo es lineal y el grupo de restricciones es especificado sólo usando igualdades y desigualdades lineales. En términos geométricos, una región factible es definida por las restricciones lineales y la función objetiva en la forma de un politopo convexo donde lo óptimo, o óptimos, son ubicados en sus vértices o caras. Uno de los métodos para resolver estos problemas es llamado el algoritmo simplex, que se mueve a lo largo de las caras del politopo, de vertice en vertice, hasta alcanzar el óptimo, o bien, determina que no existe una solución óptima.

5.8.2. Programación Entera

Si se infiere adicionalmente que todas las variables sean enteras, entonces el problema se llama problema de programación entera, en inglés *Integer Programming* (IP) o programación entera lineal, *Integer Linear Programming* (ILP). Si sólo algunas de las variables se necesitan que sean enteras, entonces el problema se llama problema de programación entera mixta, *Mixed Integer Programming* (MIP). Algunos algoritmos para resolver programas ILP son el método Cutting-plane, el algoritmo Branch and Bound y el algoritmo Branch and Cut.

5.8.3. Método Cutting-plane

El método cutting-plane [18] es usado para encontrar soluciones enteras de un problema de programación lineal. La idea básica de este método es el siguiente: Primero, el problema ILP es relajado a un problema LP y entonces es resuelto por un solver LP. Si el óptimo encontrado es una solución no-entera, se agrega una nueva restricción que corta la solución no-entera pero no corta cualquier otro punto entero de la región factible. Esto es repetido hasta que una solución óptima entera es encontrada.

5.8.4. Branch and Bound

Branch and Bound (BB) es un algoritmo general para encontrar soluciones óptimas. Puede ser visto como una modificación de la búsqueda backtracking donde el valor de una función objetivo es tomado en cuenta. La búsqueda es realizada sistemáticamente como en backtracking pero incluyendo una variable global que mantiene el mejor valor actual de la función objetivo, entonces las instanciaciones son evaluadas en términos de la variable global. Si la instanciación actual es "mejor", la variable global es actualizada. De lo contrario, es descartada. Branch and bound también puede ser combinada con el método cutting-plane para generar el algoritmo branch and cut.

5.8.5. Programación No-lineal

La programación no-lineal estudia el caso general en el cual la función objetivo o las restricciones o ambas contienen partes no lineales. Si la función objetivo es cóncava, o convexa y el dominio es convexo, entonces el problema es llamado convexo y los métodos generales de la optimización convexa pueden ser utilizados, por ejemplo, el método del elipsoide [17], el método de subgradiente [22], los métodos Cutting-plane). Si el problema es no-convexo, puede ser resuelto utilizando técnicas branch and bound.

5.9. Metaheurísticas

Para ciertos problemas encontrar la mejor solución puede ser muy costoso en términos de tiempo de cálculo. Entonces, a menudo es preferible encontrar una buena solución en una cantidad de tiempo fijada en vez de encontrar la mejor. Las metaheurísticas están basadas en este principio y para muchos problemas son más efectivas que el llevar a cabo una búsqueda exhaustiva. Las metaheurísticas pueden ser vistas como un grupo de conceptos para definir métodos heurísticos que considerando un grupo limitado de modificaciones puede aplicarse para resolver diferentes problemas de optimización.

5.9.1. Simulated Annealing

Simulated Annealing (SA) [9] se basa en esta idea. El método considera que cada candidato del espacio de búsqueda es análogo a un estado de algún sistema físico, y la función a ser minimizada representa la energía interna del sistema en ese estado. El objetivo es traer el sistema de un estado inicial a otro estado con el mínimo de energía posible.

El algoritmo SA lleva a cabo transiciones entre un estado a otro usando una fórmula de probabilidad que depende de los valores de la función (las energías) de los estados correspondientes y en un parámetro global llamado la temperatura que gradualmente disminuye durante el proceso. Esta proba-

bilidad está normalmente escogida por lo que el sistema finalmente tiende a moverse a estados de energía menor. El algoritmo sigue el proceso repetidamente hasta que el sistema alcanza un estado que es suficientemente bueno para la aplicación.

5.9.2. Algoritmos Genéticos

Los algoritmos genéticos,en inglés *Genetic Algorithm (GA)* [5], corresponden a otra técnica para resolver problemas de optimización. GA pertenece a los llamados algoritmos evolutivos que usan técnicas inspiradas por biología evolutiva como la mutación, la selección y el crossover.

Los algoritmos genéticos consisten en simular la evolución de una población inicial (que representan soluciones candidatas) hacia una mejor población (mejores soluciones). Tradicionalmente, las soluciones son representadas en binario, pero otras codificaciones también son posibles. La evolución usualmente comienza desde una población compuesta de individuos generados de manera azarosa. En cada generación, la calidad (técnicamente llamada fitness) de cada individuo en la población es evaluada para seleccionar un subgrupo que es entonces modificado (recombinado o mutado) para crear una nueva población. Esta nueva población será usada en la próxima iteración del algoritmo. Comúnmente, el algoritmo termina ya sea cuando un número máximo de generaciones ha sido producido, o un nivel de fitness satisfactorio se alcanzó para la población. Si los algoritmos han terminado por un número máximo de generaciones, una solución satisfactoria puede haber sido alcanzada, o puede que no.

5.9.3. Optimización de Colonia de Hormigas

La optimización de colonia de hormigas [13] es una técnica probabilística para resolver problemas de optimización, inspirada en el comportamiento de las hormigas para encontrar comida. El comportamiento es interesante ya que las hormigas inicialmente comienzan a buscar al azar, pero una vez que

encuentran comida regresan a la colonia dejando un rastro de feromonas, entonces las próximas hormigas pueden seguir las feromonas para llegar a la comida rápidamente, y si ellas encuentran comida eventualmente, reforzarán la cantidad de feromonas aumentando el atractivo del camino. A medida que pase el tiempo, el rastro de feromonas comenzará a evaporarse, reduciendo su atractivo. Mientras más tiempo tome el viajar ida y vuelta en el camino, más tiempo tienen las feromonas para evaporarse. Por lo tanto, los caminos cortos son normalmente más atractivos.

Por lo tanto, cuando una hormiga encuentra un camino corto desde la colonia a una fuente de comida, es más probable que otras hormigas sigan ese camino, y el feedback positivo eventualmente llevará a todas las hormigas a seguir un solo camino. La idea del algoritmo ant colony es imitar aproximadamente este comportamiento. Más específicamente, una hormiga es un agente computacional simple, que iterativamente construye una solución para la instancia a resolver. Las soluciones de problemas parciales son vistas como estados y el algoritmo itera el mover hormigas de un estado a otro. Esta transición se lleva a cabo usando una distribución de probabilidad que depende en el atractivo de la movida (calculado por una heurística determinada) y en el nivel de feromonas del camino.

5.9.4. Optimización por Enjambre de Particulas

La optimización de partículas de enjambre, en inglés *Particle Swarm Optimization* (PSO) [8], es otra técnica de resolución de problemas que pertenece a la clase de algoritmos evolutivos. La técnica se inspira en el comportamiento social de las bandadas o cardúmenes que para encontrar comida aprenden del escenario actual.

En el enfoque PSO se inicia con una población de individuos definidos al azar respecto al problema. Estos individuos son las soluciones candidatas (llamadas partículas). Las soluciones propuestas son evaluadas en términos de una función de fitness. Una estructura de comunicación también es de-

finida, asignando vecinos para cada individuo para interactuar. Un proceso iterativo para mejorar estas soluciones candidatas es puesta en marcha. La aptitud de las partículas es iterativamente evaluada y la localización donde ellas han conseguido mejores éxitos es recordada. Cada partícula entrega esta información a sus vecinos, entonces las partículas pueden observar donde sus vecinos tuvieron éxito. Los movimientos a través del espacio de búsqueda son guiados por estos éxitos, con la población usualmente convergiendo, al final del juicio, a una solución del problema.

5.9.5. Búsqueda local & Hill Climbing

Búsqueda local [3] es otra metaheurística para solucionar problemas de optimización. Los algoritmos de búsqueda local comienzan desde una solución candidata y entonces iterativamente se mueven a una solución vecina. Típicamente, cada solución candidata tiene más de una solución vecina; la elección de hacia dónde moverse es tomada usando sólo información sobre las soluciones del vecindario actual, de ahí el nombre de búsqueda local. Cuando la elección de la solución vecina es hecha tomando una y maximizando el criterio, la metaheurística toma el nombre de hill climbing. La terminación de una búsqueda local puede ser definida por un límite de tiempo o cuando la mejor solución encontrada por el algoritmo no ha sido mejorada en un determinado número de pasos.

5.9.6. Búsqueda de Vecindario Variable

La idea básica de la búsqueda de vecindario variable [15] es cambiar sistemáticamente el vecindario de la solución actual que es buscada para explorar una región cada vez más grande del espacio de solución. El procedimiento puede ser descrito como sigue. En la etapa de inicialización, un grupo de diferentes estructuras de vecindarios es seleccionado, una solución inicial es determinada y un criterio de detención adecuado es escogido. En la siguiente etapa, comenzando con la primera estructura de vecindario se determina un

vecino de la solución inicial al azar, y una heurística de búsqueda local se aplica para encontrar una solución. Si esta solución representa una mejora a la solución inicial, el movimiento se acepta y el mismo proceso se repite comenzando con esta nueva solución. Si no, el óptimo local se abandona, y un vecindario diferente, usualmente mayor, se selecciona para comenzar este proceso otra vez.

5.10. Lenguajes y sistemas para CP y Optimización

Los lenguajes y sistemas para el modelado y resolución de un CSP han sido desarrollados bajo diferentes principios. El primer sistema data de los años 60, luego ha sido seguido por una larga lista en donde una gran cantidad de paradigmas se han involucrado. Por ejemplo, el uso de la programación lógica como el soporte para el paradigma Constraint Logic Programming (CLP) o el uso de objetos para la simulación de problemas sujetos a restricciones.

Desde el punto de vista de la implementación, diferentes métodos han sido propuestos, por ejemplo, la integración de librerías sobre un lenguaje de programación o la construcción de un nuevo lenguaje de programación con soporte para restricciones. Las últimas ideas que han surgido tienen que ver con la realización de un lenguaje de programación puro que sea entendible fácilmente para el usuario

Se pueden clasificar los lenguajes y sistemas para el manejo de restricciones en 4 grupos: sistemas CLP, librerías, lenguajes de modelado y lenguajes de programación como se muestra a continuación:

5.10.1. CLP

Constraint Logic Programming es un paradigma de la programación lógica. CLP tiene por objetivo combinar los aspectos declarativos de la programación lógica y la resolución de restricciones para la solución de un pro-

blema. CLP maneja restricciones sobre diferentes dominios que pueden ser declarados en un modelo uniforme y ser resueltas mediante técnicas diversas como inteligencia artificial o investigación de operaciones. CLP es utilizado principalmente en áreas de producción industrial, como por ejemplo, sistemas de gestión y programación de producción o diseño de circuitos digitales, los cuales poseen las características de ser compatibles con la programación con restricciones.

5.10.2. Librerías

Las librerías para CP proveen un lenguaje para la declaración de problemas sujeto a restricciones. Estas son normalmente implementadas mediante clases y métodos específicos en un determinado lenguaje de programación, por ejemplo, una determinada clase puede ser usada para declarar variables y métodos que definen la relación entre ellos. La gran particularidad de estos métodos es la no necesidad de implementación de un nuevo lenguaje para resolver un problema sujeto a restricciones.

Dentro de esta categoría nos encontramos con las librerías a utilizar en este proyecto:

- **Gecode y Gecode\J**: Gecode [1] es una librería escrita en C++. Fue diseñada especialmente para dar soporte a variables con dominios finitos. Gecode provee un solver que maneja diferentes clases de restricciones, como por ejemplo, enteros, booleanos o un conjunto de variables. Por otra parte, se ha implementado Gecode\J que es una interfaz basada en Java la cual extiende el uso de esta a este lenguaje.

- **Choco**: Choco [2] es un solver de programación con restricciones. Es una librería implementada en Java que busca principalmente dar soporte a problemas de optimización. Su principal característica es la de dividir entre el modelado (modeling) y la solución (solving) de un problema sujeto a restricciones. El modelado trata sobre los modelos

genéricos de las restricciones y variables instanciadas a través de un paradigma de CP. El segundo término trata acerca de las estructuras de datos sujeto a restricciones y variables, las cuales pasarán desde un modelo CP a un solver CP.

5.10.3. Lenguajes de Modelado

Los lenguajes de Modelado apuntan a simplificar la definición de los problemas sujetos a restricciones. Estos intentan desplazar a los usuarios fuera de la codificación complicada presente en las típicas librerías o lenguajes de programación. La esencia de este tipo de lenguajes es generalmente más comprensible debido a su sintaxis simple y una semántica amigable. Algunos ejemplos son OPL [27], Zinc [21] y MiniZinc [19].

6. Solución Propuesta

Existen diversas competencias en el mundo enforcadas en la implementación de agentes para resolver problemáticas relacionadas a un video juego determinado, tales como Ms PacMac [11], Cellz [12] y X-Pilot [20]. Como se mencionó anteriormente, Mario AI Competition es una de ellas, la cual comenzó en el año 2009 y actualmente continua realizándose cada año. El objetivo de esta competencia es implementar un agente que simule el comportamiento de Mario para superar la mayor cantidad de niveles posible. Para ello debe debe sortear una serie de obstáculos y enemigos tales como: criaturas, tuberías, acantilados y bloques, como se observa en la figura 11.

Figura 11: Juego Super Mario Bros

Mario AI Benchmark entrega toda la información necesaria para que

Mario pueda tomar decisiones, para lo cual provee una matriz en la que se detalla cada uno de los elementos del mapa. Además provee información del estado actual de Mario, tal como si se encuentra saltando, en el suelo, o deslizándose. En la solución propuesta se implementó un agente, el cual se encarga de tomar las decisiones por medio de la programación con restricciones. Este agente considera una grilla que abarca un espacio visible de 7x7 [26], con el fin de observar correctamente los cambios de escenario que puedan suceder, como se visualiza en la figura 12.

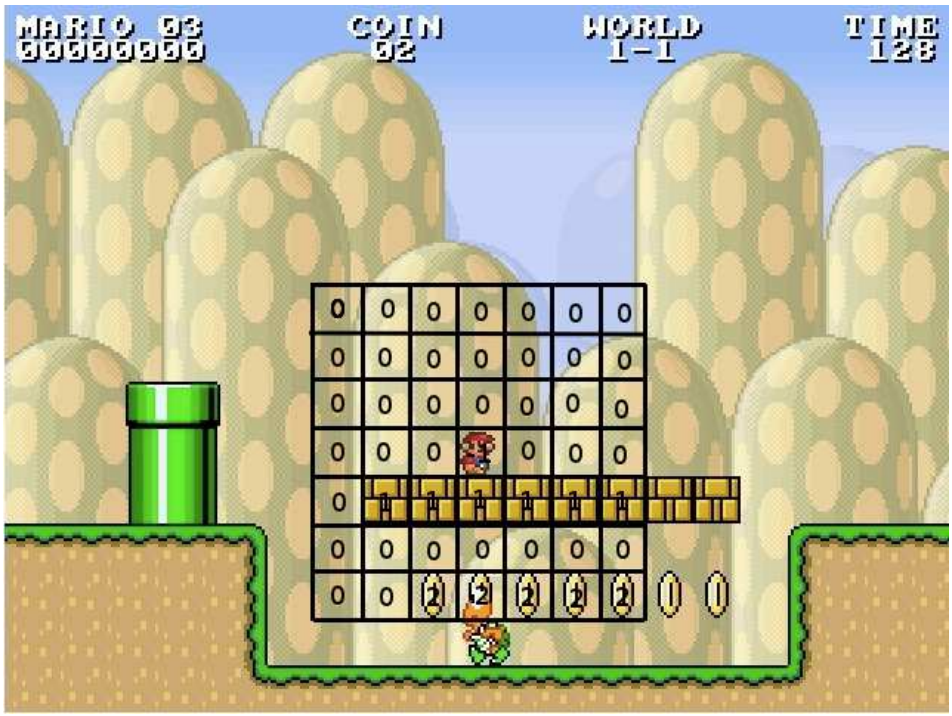

Figura 12: Matriz de observación

El agente propuesto también considera una serie de situaciones que podrían ocurrir. Ésto debido a que, dado que la generación de niveles es al azar, pueden suceder situaciones difíciles, o incluso imposibles de resolver,

un ejemplo claro es el de un acantilado que Mario no pueda sortear con su mayor salto. Además, pueden tomarse diversas decisiones tácticas que ayuden a Mario a completar el nivel, aunque afecten negativamente la solución inmediata, pero que ayuden más adelante, como por ejemplo, devolverse para tomar algún poder.

Todo lo mencionado debe situarse dentro de las características propias del juego, las cuáles son: distintos tipos de niveles, distintas dificultades y distintos eventos. Estas combinaciones de opciones proporcionan una cantidad innumerable de posibilidades. Sin embargo, toda decisión tomada por el agente tiene como plazo máximo, impuesto por la competencia, de 42 ms. En caso de superar este tiempo, el juego penalizará al agente y clasificará el nivel como no superado.

Antes de proceder al modelado del juego utilizando la programación con restricciones, se entregará una descripción necesaria del entorno del juego.

6.1. Entorno

El entorno de Mario entregado por el juego consiste en una matriz de 22x22, la cual contiene toda la información relevante del nivel. En ella se detallan todos los elementos que están en el mapa a través de un número que los identifica, a continuación se detallan los elementos que componen el mapa y sus número asociados:

- **Terreno**: Si la casilla es aire, se considera 0. Mientras que al suelo se le asigna el valor -60, otros elementos que se consideran en el terreno son las tuberías y los tipos de bloques (sólo ladrillo, oculto, con ítem, sin ítem). Para efectos de simplificar la resolución del juego se asignó un valor 1 a todo elemento que Mario no puede atravesar, en este caso suelo, ladrillos, tuberías; 0 cuando el valor es vacío y 2 cuando hay un elemento, pero este no representa un obstáculo, por ejemplo las mone-

das.

- **Criaturas**: A cada criatura se le asigna un valor distinto. De ese modo pueden tomarse decisiones acerca de qué forma actuar frente a ellos. Por ejemplo, si es una tortuga, puede pisarse y luego tomar la caparazón. En nuestro caso se tomaron estos valores y se les asignó el valor 1 para que al realizar los cálculos, Mario siempre las esquivará debido a que las consideraba un obstáculo.

Las acciones de Mario están dadas a partir de la posición en la que se encuentra con respecto al mapa, las posiciones de los enemigos también son distintas de los elementos del mapa ya que estos se desplazan a través de él. Es por ello que en estos casos se devuelven los valores de los ejes X e Y para saber la posición de Mario y los enemigos, para tomar las decisiones que correspondan y que se basan en el estado de Mario en ese momento, por ejemplo, si Mario acaba de ejecutar un salto ya no puede saltar de nuevo hasta que se haya vuelto a tocar el terreno.

- **Posición de Mario y los enemigos**: debido a que la matriz siempre se centra en Mario, siempre se sabrá en qué posición se encuentra y la posición relativa de los enemigos. Con esa información se podrán tomar las decisiones pertinentes.

- **Estado de Mario**: Las variables de decisión contendrán información sobre el estado de Mario. Estás permiten tomar ciertas decisiones. Por nombrar sólo algunas de ellas: puede saltar, está en el suelo, el poder actual, se está deslizando, etc.

La información del entorno es entregada al agente que tiene implementado el modelo. Este agente entonces tomará la decisión de qué acción llevar

a cabo. Las acciones de Mario tienen dos estados posibles, verdadero o falso. Estas indican si la acción se realiza o no. Las acciones que puede realizar Mario son:

- **Dirección**: son las 4 direcciones que pueden realizarse. Arriba, abajo, derecha e izquierda. De estas 4 sólo 3 tienen acciones asociadas en el juego (abajo, derecha e izquierda), y de estás 3 sólo dos tienen una influencia directa en la solución (derecha e izquierda).

- **Salto**: esta acción permite superar la gran mayoría de los obstáculos del juego, desde los relacionados al terreno (muros, tuberías, acantilados) hasta los obstáculos en movimiento (enemigos).

- **Acción**: tiene la particularidad de realizar dos cosas distintas. Una es que cuando se pulsa, realiza la acción especial que corresponde al estado de Mario. Por ejemplo si Mario está en el estado Fuego, entonces Mario disparará una bola de fuego. Y la otra acción posible, es que cuando se mantiene pulsada la tecla, Mario aumentará la velocidad de movimiento.

6.2. Análisis

Una vez estudiado el juego y la forma en que fue desarrollado y adaptado se procede a modelar el problema utilizando la programación con restricciones. La primera decisión corresponde a definir el objetivo, los alcances y los supuestos a utilizar. El objetivo perseguido a través de nuestro análisis, es que Mario sea capaz de superar el nivel. Las restricciones a tomar en cuenta dependerán del estado de Mario. Se considerarán dos estados diferentes: si Mario está en el aire, o Mario está en el suelo. Y para cada uno de estos estados se definirán restricciones distintas. Como el objetivo tomado

es terminar el nivel, se tomarán 4 variables que permiten que Mario pueda realizar el objetivo. Estas son: Izquierda, Derecha, Salto y Poder Especial.

Las decisiones serán tomadas en un entorno constante, el cual corresponde a un frame. Es por ello que todos los elementos del nivel, a excepción de las acciones de Mario, se considerarán constantes. Sin embargo, cabe recordar que estos son dinámicos en el tiempo. En conclusión, Mario saltará cuando se encuentre con algún obstáculo en el terreno, que no puede ser superado sólo caminando. Estos son: acantilados, muros, tuberías y bloques. Entonces, en la siguiente iteración ya no considerará el dominio en que Mario está en el suelo, si no que será considerado el dominio que corresponde a cuando Mario está en el aire.

6.3. Modelado del Mario AI Benchmark como un CSP

El modelado del problema como un CSP considera los siguientes elementos:

- **Constantes**

  `isOnGround, mayJump, nivel, PosX, PosY, Nivel[22][22]`

  Estas constantes corresponden al estado de Mario y del entorno en un momento dado. Los valores que pueden tomar isOnGround y mayJump son 1 y 0 para indicar si se cumple o no dicho estado. PosX y PosY entregan la posición de Mario dentro de la matriz Nivel. La matriz Nivel entrega el entorno de Mario, indicando en cada posición un número que identifica el elemento que se encuentra en dicha casilla.

- **Variables**

  $\texttt{salto, izquierda, derecha, accionEspecial} \in \texttt{[0,1]}$

Las acciones presentadas aquí son las que tienen un impacto directo en la solución, el dominio está dado por el hecho de que si deben realizar esa acción o no, siendo el valor 1 verdadero y 0 falso, ésto indica que mientras la acción sea 1 se mantendrá pulsada la tecla que corresponde a dicha acción.

- **Restricciones**

  - La primera restricción consiste en que Mario avanzará a la derecha si y solo si encuentra que en las siguientes dos posiciones delante de él no hay obstáculos.

    ```
    derecha == 1 <=>
    (nivel[PosX+1][PosY] == 0 AND
    (nivel[PosX+2][PosY] == 0 AND
    (nivel[PosX+3][PosY] == 0)
    ```

  - En la segunda restricción, salta si encuentra que delante no hay terreno sobre el cual caminar.

    ```
    salto == 1 <=>
    NOT(nivel[PosX+1][PosY+3]==1 OR nivel[PosX+1][PosY+2]==1
    OR nivel[PosX+1][PosY+1]==1)
    ```

  - La tercera restricción gatilla la acción especial de Mario si y solo si encuentra que delante de él no hay obstáculos y hay suelo.

    ```
    accionEspecial == 1 <=>
    ((nivel[PosX+1][PosY+1]==1 AND nivel[PosX+2][PosY+1]==1)
    AND (nivel[PosX+1][PosY] == 0))
    ```

- La cuarta restricción hace que Mario retroceda si encuentra que delante de él no hay un terreno seguro para andar y detrás de él hay terreno en el cual caminar.

```
izquierda == 1 <=>(derecha == 0) AND
(nivel[PosX-1][PosY+1] == 1) AND
(nivel[PosX-2][PosY+1] == 1))
```

- La quinta restricción realiza que Mario avance a la derecha si encuentra que la posición vecina de él a la derecha es segura, para esto, todas las posiciones deben valer 1.

```
derecha =
(nivel[PosX][PosY+1] *
nivel[PosX+1][PosY+1] * nivel[PosX+2][PosY+1])
```

- La sexta restricción provoca que Mario gatille la acción especial cuando se encuentra con un obstáculo delante que no puede ser superado solo andando.

```
accionEspecial =
(nivel[PosX][PosY+1] *
nivel[PosX+1][PosY+1] * nivel[PosX+2][PosY+1])
```

- La séptima restricción provoca que Mario salte bajo dos condiciones: se encuentra delante un acantilado por lo que debe saltar para superarlo (la única posición que vale 1 en este caso es la que se encuentra bajo él), ó delante hay un muro por lo que también debe saltar para superarlo (en este caso las posiciones alrededor de Mario valdrán 1).

```
salto == 1 <=>
((nivel[PosX][PosY+1] +
nivel[PosX+1][PosY+1] +
nivel[PosX+1][PosY]) == 1) OR
(nivel[PosX][PosY+1] *
nivel[PosX+1][PosY+1] * nivel[PosX+1][PosY])== 1))
```

El modelo obtenido y aplicado aquí es similar a uno basado en reglas, se eligió esta aproximación ya que es la que menos cambios introducía dentro de las librerías de Mario AI, sólo centrándose en las variables del entorno para resolver la siguiente acción a realizar.

6.4. Implementación

Como se mencionó anteriormente, cada movimiento de Mario se modela como un CSP, por lo tanto para resolver una etapa completa se necesitan tantos movimientos como Mario requiera para llegar al final. Esto se controla comunicando al juego con el agente como se visualiza en la figura 13. El juego retroalimenta al solver para que pueda decidir que movimiento realizar, una vez que el agente toma la decisión, esta es enviada al juego quien gatilla el movimiento. Esta interacción se realiza n veces hasta completar la etapa.

En la implementación, se decidió trabajar con la versión de Mario que está implementada en Java para facilitar la integración con las solvers a utilizar. Se eligió trabajar con dos solvers para comparar los resultados entre ellos, estos son Gecode\J [1] y Choco [2].

En Gecode\J específicamente se utilizará la versión 2.2 la cual tiene la característica de ser la última versión que presenta una interfaz Java. Choco también está implementada en Java lo que facilita la integración con el código de Mario AI, sin embargo en contraste con Gecode\J permite resolver problemas de optimización. Con esto se pretende dejar una ventana abierta

Figura 13: Interacción entre el Juego y el Agente

para futuros estudiantes que podrían modelar el Mario AI Benchmark como un problema optimización, como por ejemplo para maximizar la cantidad de monedas obtenidas.

Se implementaron dos agentes: el agente MarioGecode y MarioChoco. Para cada agente se implementó el CSP correspondiente al Mario AI Benchmark, presentado en la subsección anterior, por medio del lenguaje proveído por cada uno de los solvers. Un fragmento de código en Gecode\J se visualiza en la figura 14. Donde `IntVar` es un objeto que permite definir variables de decisión, en este caso la variable de decisión con nombre `s` posee el dominio $[0, 1]$. Luego con el metodo `post` se establece una restricción. Debido a que en Java no se pueden sobreescribir las operaciones matemáticas básicas, estas deben representarse por métodos. Asimismo, `BExpr` permite definir una expresión Booleana y `Expr` una expresión aritmética. El operador `IRT_EQ` corresponde al operador de igualdad y `p` corresponde al operador de suma.

6.5. Experimentos

En esta sección se describe una breve experimentación realizada con ambas implementaciones. Se realizaron 30 ejecuciones para ambos agentes. Aquellos que tuvieron éxito en alguno de ellos se detallan en Tabla 2 y Tabla 2. El nivel de dificultad aplicado fue 1, donde los acantilados son generados y los enemigos no se incluyeron. En estos resultados se incluye el número de seed con el que se generó el mapa y los puntajes obtenidos por Mario al finalizar las etapas, estos puntajes son obtenidos a partir de la cantidad de items y monedas que Mario recolectó durante el juego como también el tiempo sobrante.

Como se aprecia en la Tabla 1, no se visualiza gran variación entre Choco y Gecode\J. Sin embargo, al condiderar distintos seeds, Choco presentó mayores casos de éxito que Gecode\J como se presenta en la Tabla 2.

De estos resultados se aprecia que Choco obtiene mejores resultados. Sin embargo, debe considerarse que no se utiliza la última versión de Gecode

```
...
IntVar s  = new IntVar(this,"s",0,1);
vars.add(s);
IntVar v  = new IntVar(this,"v",0,1);
vars.add(v);
...

post(this,new BExpr(
        new Expr().p(s),
        IRT_EQ,
        new Expr().p(1))
        .and(
            new BExpr(
                new Expr().p(getScene(x+1,y) *
                getScene(x,y+1)*getScene(x+1,y+1)),
                IRT_EQ,
                new Expr().p(1))
            .or(
            new BExpr(
            new Expr().p(getScene(x+1,y) + getScene(x,y+1) +
                getScene(x+1,y+1)),
                IRT_EQ,
                new Expr().p(1))
                )
            )
        )
    );
...
```

Figura 14: Un fragmento del código implementado en Java

Seed	Puntaje Choco	Puntaje Gecode\J
6189642	8368.0	8480.0
2908345	8352.0	8568.0
291245	8368.0	8304.0
12457	8464.0	8424.0
213312	8488.0	8568.0
213312	8448.0	8232.0
47474	8400.0	8656.0

Tabla 1: Resultados con éxito en ambas librerías

Seed	Puntaje Choco	Puntaje Gecode\J	Exito
212256	5139.0	8136.0	Gecode\J
23131	3003.0	8736.0	Gecode\J
223523	8224.0	5417.84	Choco
235628	8440.0	2515.0	Choco
5643325	8312.0	3387.0	Choco
1234223	8760.0	8232.0	Choco
75433	8400.0	5830.75	Choco
1245	8648.0	5383.93	Choco
363965	8192.0	6878.69	Choco
4325	8472.0	1227.0	Choco

Tabla 2: Resultados con éxito en una librería

y que pueden haber diferencias debido a la implementación del modelo y también debido a la naturaleza dinámica del problema, lo que ocasionó que se produjera esa diferencia entre ambas librerías, ya que en los casos en que ambos tuvieron éxito los resultados eran bastante similares.

7. Conclusiones

En este trabajo se ha presentado el modelado y resolución del Mario AI Benchmark utilizando CP. Al utilizar CP y compararlo con otras técnicas de programación se encontraron varias diferencias, probablemente la más destacada es la capacidad de CP de centrarse en el modelado del problema, dejándole la tarea de buscar la solución al solver. De esta forma, se alivia la carga al desarrollador, el cual ya no debe preocuparse de implementar un algoritmo específico para resolver el problema. Sin embargo, CP sólo puede aplicarse a problemas que pueden llevarse a un modelado matemático compuesto principalmente por variables y restricciones. Además cabe destacar que CP hace una búsqueda completa, por lo cual en instancias de gran tamaño puede consumir tanto tiempo como memoria de orden exponenciales.

Al llevar a cabo este proyecto han surgido diversas ideas que han permitido ir extendiendo el alcance de éste, y que pueden tomarse como trabajo futuro:

- Mejorar las decisiones en forma evolutiva: al probar los modelos se han ido mejorando de tal forma que Mario logre cumplir con el objetivo de manera más eficiente.

- Agregar criaturas: esta será una de los puntos que más afectará el juego, debido a que dependiendo del enemigo pueden realizarse distintas acciones. Estas además provocarán que el mapa esté en permanente cambio, lo que producirá que una solución pueda quedar inválida a la próxima iteración, por ejemplo cuando un enemigo bloquee un camino.

- Ampliar la matriz para tomar decisiones: actualmente se consideró sólo un espacio cercano a Mario para tomar decisiones. Al ampliar este espacio se podrán tomar nuevas decisiones, lo que implica que nuevas restricciones saldrán al paso, las cuales tendrán incidencia en puntos como el tamaño del salto o pisar enemigos para saltar aún más,

etc. Sin embargo, debe evaluarse el costo de ampliar la matriz, ya que al ampliarla las restricciones se volverán más complejas.

- Optimización: Actualmente este problema se está tratando como un problema de satisfacción, sin embargo, puede ser también considerado como un problema de optimización en el cual existen una gran cantidad de funciones objetivo que pueden ser consideradas, como por ejemplo: Minimizar el tiempo de resolución del nivel, maximizar la cantidad de monedas conseguidas, maximizar la cantidad de enemigos pisados, etc.

Referencias

[1] *Gecode System*, Visited 8/2008. Available from http://www.gecode.org.

[2] *Choco Solver*, Visited 9/2008. http://www.emn.fr/x-info/choco-solver/doku.php.

[3] E. Aarts and J. K. Lenstra. *Local Search in Combinatorial Optimization*. John Wiley and Sons, NY, USA, 1997.

[4] G. Dantzig. *Linear Programming and Extensions,*. Princeton University Press, 1963.

[5] D. Goldberg, D. Kalyanmoy, and J.Horn. Genetic algorithms. In *in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[6] S. Karakovskiy, N. Shaker, J. Togelius, and G. N. Yannakakis. The mario ai championship. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–3, 2010.

[7] S. Karakovskiy and J. Togelius. The mario ai benchmark and competitions. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(1):55–67, 2012.

[8] J. Kennedy and R. Eberhart. Particle swarm optimization. In *IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

[9] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[10] K.R. Apt. *Principles of Constraint Programming*. Cambridge Press, 2003.

[11] Simon M. Lucas. Evolving a neural network location evaluator to play ms. pac-man. In *CIG*, 2005.

[12] S.M. Lucas. Cellz: a simple dynamic game for testing evolutionary algorithms. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 1, pages 1007–1014 Vol.1, 2004.

[13] M., V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, 26:29–41, 1996.

[14] A. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[15] N. Mladenovic and P. Hansen. Variable neighborhood search. *Computers & OR*, 24(11):1097–1100, 1997.

[16] R. Mohr and T. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28(2):225–233, 1986.

[17] S. Nash and A. Sofer. *Linear and Nonlinear Programming*. McGraw-Hill, 1996.

[18] G. Nemhauser and L.Wolsey. *Integer and Combinatorial Optimization*. John and Wiley and Sons, 1999.

[19] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. Minizinc: Towards a standard cp modelling language. In C. Bessière, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP 2007)*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007.

[20] M. Parker and G.B. Parker. The core: Evolving autonomous agent control. In *IEEE Symposium on Artificial Life*, pages 222 – 228, 2007.

[21] R. Rafeh, M. J. García de la Banda, K. Marriott, and M. Wallace. From zinc to design model. In *Proceedings of the 9th Symposium on Practical*

Aspects of Declarative Languages (PADL 2007), volume 4354 of *Lecture Notes in Computer Science*, pages 215–229, 2007.

[22] R. Rockafellar. *Convex analysis*. Princeton University Press, 1854.

[23] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

[24] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, 1998.

[25] J. Togelius, S. Karakovskiy, and R. Baumgarten. The 2009 mario ai competition. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2010.

[26] J. Togelius, S. Karakovskiy, J. Koutník, and J. Schmidhuber. Super mario evolution. In *Proceedings of the 2009 IEEE Symposium on Computational Intelligence and Games (CIG)*, pages 156–161, 2009.

[27] P. Van Hentenryck. *The OPL Language*. The MIT Press, 1999.

[28] D. Waltz. Understanding Line Drawings of Scenes with Shadows. In *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.