UNIVERSIDAD DE TALCA
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA EN DESARROLLO DE
VIDEOJUEGOS Y REALIDAD VIRTUAL

Generación Procedural de Niveles Interesantes de Sokoban

HANS SCHAA LEPE

Profesor Guía: NICOLAS BARRIGA

Memoria para optar al título de
Ingeniero en Desarrollo de Videojuegos y Realidad Virtual

Talca – Chile
Julio, 2021

CONSTANCIA

La Dirección del Sistema de Bibliotecas a través de su unidad de procesos técnicos certifica que el autor del siguiente trabajo de titulación ha firmado su autorización para la reproducción en forma total o parcial e ilimitada del mismo.

Talca, 2021

AGRADECIMIENTOS

Agradecimientos a mis padres, abuelas, tías y hermanos por el apoyo en esta aventura facilitándome en lo posible la vida para concentrarme en mis estudios. También a mi hermano Piero que sin su existencia no habría hecho el mismo esfuerzo en cada actividad académica.

A mis amigos, amigas y conocidos por ser compañeros de este largo viaje y disfrutar juntos la vida.

Gracias a mi profesor guía que sin su ayuda este documento no hubiera sido posible. Por la paciencia, su compromiso con el proyecto y por generar en mí el interés para seguir estudiando y llegar a ser algún día docente. A todas las personas que conforman la Escuela de Videojuegos que aportaron a mi desarrollo como persona y profesional.

A mis compañeros de Ingeniería en computación por esas relajantes tardes de Wurlitzer en el difunto Sabor del Puerto, mis amigos Patricio y Víctor.

Por último, a Dios por permitirme seguir vivo y no dejar que la desesperanza consuma mi humanidad.

TABLA DE CONTENIDOS

página

ÍNDICE DE FIGURAS

página

ÍNDICE DE TABLAS

página

RESUMEN

La evolución de los métodos para crear videojuegos ha ocasionado una avalancha de nuevas formas para satisfacer distintas demandas relacionadas al contenido de estos proyectos interactivos.

Un buen método de generación procedural de contenido (PCG) nos proporciona una vía para generar distintos tipos de elementos de manera diversa, rápida, controlable, fiable y creativa de tal forma que permita la entrega de nuevo contenido a los usuarios y un ahorro en los costos de producción al no necesitar personas que nos ayuden creando las imágenes, vídeos, ambientes, sonidos y niveles de videojuegos solicitados.

En este estudio implementamos un algoritmo PCG constructivo de abajo hacia arriba para generar tableros de Sokoban. El propósito es investigar si este proceso nos permite crear niveles entretenidos, diversos y difíciles. Realizamos una encuesta por medio de una aplicación web donde los usuarios deben jugar y calificar los niveles creados frente a estas tres variables antes mencionadas. Mostramos los resultados obtenidos y cómo este proceso puede ser mejorado implementando nuevas formas de exploración del espacio de búsqueda para guiarla hacia nuevas instancias de tableros difíciles de resolver.

Concluimos frente a los datos obtenidos de los usuarios y promovemos otras líneas de investigación para crear nuevo contenido basado en algoritmos de búsqueda estocástica.

1. Introducción

La alta demanda de contenido dio lugar a nuevas formas de enfrentar este problema. Existen dos maneras principales de crear nuevo contenido para videojuegos: una es de forma manual con ayuda de diseñadores, artistas, guionistas, etc.; la otra manera es por medio de algoritmos que generan el contenido de forma procedural. Es en ese último enfoque en el cual nos focalizamos para llevar a cabo esta investigación. Con la llegada de los *smartphones* hubo una tendencia exponencial de personas que se divierten con videojuegos casuales. Juegos como Sokoban entran en esta categoría y por ende un enfoque PCG para la generación de entretenidos niveles de juego es una buena idea. Un buen método PCG nos permite mantener una constante entrega de nuevo contenido a nuestros clientes, contenido diverso y de calidad, el cual es generado de forma automática permitiéndonos también ahorrar en recursos humanos.

PCG ha sido un área ampliamente estudiada con distintos propósitos. Sus primeros usos comúnmente citados datan de 1980 en el videojuego Rogue o Elite [2], sin embargo, sus aplicaciones abarcan desde el cine [3], literatura [4], educación [5] hasta los videojuegos [6].

Los puzles casuales como Sokoban son diariamente jugados por millones de usuarios. Las plataformas varían desde celulares hasta consolas de alta gama, juegos *Sokoban-based* como "Baba is you" han recaudado alrededor de 2.8 millones de dólares en copias vendidas [7] y al igual que Sokoban también ha sido fuente de estudio y experimentos para dar con el método que nos permita hacer PCG para generación de niveles [8].

1.1. Problema

La tarea de diseñar niveles desafiantes para videojuegos del género puzle como Sokoban puede llegar a ser muy difícil de abordar. Factores como balance de dificultad y la creatividad del diseñador pueden marcar la diferencia entre un videojuego que retenga al usuario por unos pocos segundos a varios minutos. La tarea de mantener en control la dificultad de los niveles que un diseñador crea puede ser muy complicado [9, 10], anticipar el impacto que puede tener agregar o quitar obstáculos particularmente en rompecabezas, puede reflejar una cantidad no menor de nuevos movimientos para resolver el desafío propuesto al usuario.

Otro factor es la alta demanda de nuevo contenido sin dejar de lado la expectativa que tienen los usuarios sobre este. Debido a esto hemos decidido estudiar la implementación de técnicas PCG para la creación de niveles de Sokoban. Otras motivaciones para enfrentar este problema es la activa comunidad que está interesada en generar niveles de Sokoban y los subproblemas que están inmersos en él como por ejemplo garantizar jugabilidad en cada nivel generado. También hay una gran banco de niveles [11] que han sido agrupados por la comunidad científica y no científica proporcionando por ejemplo *training data* para métodos basados en redes neuronales. Este banco de información nos proporciona un punto de comparación para niveles generados por métodos automáticos versus los creados por humanos.

Muchas personas juegan videojuegos diariamente y para casi todos los usuarios el contenido (historias, niveles, personajes, mecánicas, música, sonidos, etc.) entregado en cada proyecto representa un papel importante para su diversión. En la mayoría de los videojuegos el trabajo manual del equipo de desarrollo da garantías de que la calidad y cantidad del contenido del videojuego cumpla con las demandas constantes de la comunidad de jugadores. Sin embargo, existen proyectos donde la demanda supera las capacidades productivas del equipo de trabajo.

Resolver niveles de Sokoban es NP-Hard[1], característica que ha despertado interés en una gran cantidad de investigadores [12, 13, 14]. El alto factor de ramificación de sus estados de juego expone un gran desafío. Para solucionar este problema proponemos un procedimiento de 2 fases: la creación de los tableros bases y la combinación de la información base por medio de computación evolutiva para la generación de

[1]Categoría de problemas para los cuales no se garantiza que la solución se encuentre en tiempo polinomial

niveles entretenidos, desafiantes y diversos. Los objetivos son generar tableros para el puzle Sokoban de forma *offline* que sean interesantes de jugar para el usuario, además que éstos presenten una clara diferencia entre sí, ya sea por medio de las formas que el usuario tiene para recorrer el mapa y llevar las cajas hacia los objetivos (ver anexo B) o por la cantidad de metas dispuestas en el tablero. Tales métricas estarán acompañadas de estudios de usuarios para contar con la retroalimentación suficiente para dar origen a la clasificación dentro del banco de niveles generados. El resultado esperado es un método para gefnerar diversos niveles de Sokoban que sean desafiantes de resolver pero a la vez entretenidos de jugar.

Para solucionar el problema de la alta demanda de contenido y ayudar con su creación existen enfoques que pueden facilitar la generación de niveles y con ello cumplir las expectativas de los usuarios. Existen estrategias que plantean construir el tablero a partir del estado solución hacia atrás o desde el vacío hacia el estado solución. Algunas técnicas usadas son el uso de plantillas [15], *Backtracking* [16], *Monte-Carlo Tree Search* (MCTS) [17], algoritmos evolutivos [18], *Novelty Search* (NS) [19], *Machine Learning* (ML) [20] y el uso del algoritmo *Wave Function Collapse* (WFC) [21].

El objetivo de nuestro estudio es generar niveles (estados iniciales) para el puzle Sokoban de forma *offline* que sean entretenidos y desafiantes para el usuario, además que estos presenten una clara diferencia entre sí, la cual puede estar dada por las formas que el agente tiene para recorrer el mapa, cantidad de metas y cajas dispuestas en el tablero, disposición de las murallas, dimensión del nivel, etc. Tales métricas estarán acompañadas de estudios de usuarios para contar con la retroalimentación suficiente que respalde las eventuales conclusiones. En lo que resta del documento llamaremos a los estados iniciales de Sokoban que sean difíciles (desafiantes), que sean distintos de sus pares (diversos) y atractivos (entretenidos) como "Niveles interesantes".

En la presente tesis se busca dar sustento a la siguiente hipótesis general: El uso de algoritmos evolutivos para generar niveles interesantes permite obtener niveles entretenidos, desafiantes y diversos. Vamos a separar esta hipótesis general en tres sub-hipótesis:

1. Es posible generar tableros de Sokoban entretenidos por medio de algoritmos evolutivos.

2. Es posible generar tableros diversos por medio de algoritmos evolutivos.

3. Es posible generar tableros difíciles por medio de algoritmos evolutivos.

A continuación, describiremos las métricas a medir al momento de estudiar las hipótesis mencionadas en la numeración anterior:

Entretención Medir entretención es una tarea difícil de llevar a cabo. La fuente de entretención de una persona puede ser la forma de aburrirse de otra. Por medio de *user-study* evaluaremos el nivel de entretención que los Niveles Interesantes causan en las personas. Creemos que una curva de dificultad moderada puede desencadenar un flujo de juego que respete la habilidad de un usuario medianamente experimentado.

Diversidad Para responder a ésta métrica nos tenemos que hacer la pregunta ¿Qué artefactos o características de los niveles los hacen diferentes? Identificamos 4 elementos principales que dan sentido a las mecánicas de juego, estos son las murallas u obstáculos estáticos, las cajas u objetos movibles, las metas y el jugador. Estos componentes creemos que son piezas claves para que el usuario mida la diversidad de los niveles jugados.

Dificultad Nos referimos al esfuerzo implicado por el usuario para resolver los niveles generados. Para ello también necesitaremos datos como el nivel de experiencia del usuario en Sokoban para concluir de manera más acertada frente a esta métrica. Nos proponemos generar tableros de juego difíciles pero que conserven la característica de que sean entretenidos de jugar, además de que sean un desafío para personas con una experticia moderada en la resolución de niveles de Sokoban.

El alcance de este proyecto es la creación de una estrategia para generar niveles automáticamente para el puzle Sokoban, por ahora nos conformamos con que el generador nos entregue buenos candidatos para que posteriormente un humano los pueda filtrar de forma visual, con esto nos referimos a seleccionar los niveles mas diversos del conjunto de niveles generados. Estos niveles deben tener cualidades como una alta dificultad, diferentes entre sí y que sean entretenidos de resolver. Los métodos que examinaremos para la eventual estrategia PCG serán basados en búsqueda y optimización.

La sección 2 aborda el estado del arte en PCG para niveles de Sokoban y otras soluciones basadas en PCG para juegos que resultan relevantes en la investigación. Adicionalmente se presenta el marco teórico y los conceptos que forman parte de las soluciones actuales al problema de generar estados de juegos iniciales para Sokoban. En la sección 4 mostraremos las técnicas propuestas y cómo estas convergen hacia los resultados, también se mencionarán y explicarán los experimentos realizados y la metodología de desarrollo aplicada para las herramientas que nos permitieron realizar el estudio. Luego en la sección 5 se dan a conocer los resultados de los experimentos junto a su respectivo análisis y conclusiones. Por último adjuntamos en la sección 6 información sobre el puzle Sokoban, JSoko y algunos detalles sobre la herramienta de *User testing* Sokostigation.

Finalmente se espera contribuir con nuevas alternativas para llevar a cabo PCG en juegos *Sokoban-based*, en primera instancia de forma *offline* pero con la intención de que los métodos expuestos se puedan llevar ejecutar en un contexto *online*, donde los tiempos de espera para la generación de nuevo contenido procedural no sean extensos y se puedan incluir en el flujo de juego sin dañar la experiencia de usuario.

2. Revisión Bibliográfica

En esta sección se presentarán los métodos existentes para implementar generación de contenido automático enfocado en la construcción de los niveles de videojuegos. Se darán a conocer las metodologías usadas y el impacto que han generado en el estado del arte. Entre ellas podemos destacar el uso de algoritmos evolutivos, MCTS y *Machine Learning*, técnicas que han sido pulidas a los largo de los años evolucionando en nuevas variantes que amplían las alternativas para dar solución a problemas relacionados con la generación de contenido procedural.

2.1. Generación de estados iniciales para Sokoban

Resolver tableros de Sokoban es un problema NP-Hard, esta característica de jugabilidad es un subproblema dentro de la generación de niveles de Sokoban, esto y muchos otros factores lo convierten en un interesante videojuego a estudiar [22]. Existen varios enfoques relacionados con PCG para niveles de videojuegos ejecutados en contextos *offline*, por ejemplo, Taylor y Parberry [15] generaron niveles de Sokoban que están garantizados de tener solución, sobre la base de buscar hacia atrás a partir de las metas. Aseguran que su técnica produce niveles interesantes, pero solo es adecuada para la generación *offline* porque se ejecuta en un tiempo exponencial y no puede manejar niveles con más de seis cajas. Cuatro años más tarde plantean un enfoque constructivo [23] dividiendo la generación del nivel en tres etapas principales:

1. Ubicación de las paredes: Por medio de *templates* y algunas restricciones se arma la base del nivel, los *templates* son rotados y probados al finalizar para garantizar el buen funcionamiento del algoritmo en las etapas posteriores.

2. Posicionamiento de las metas: Se analizan todas las combinaciones posibles para las metas por medio de *Backtracking* y se colocan en la estructura del nivel.

3. Ubicación del jugador.

Al igual que Taylor y Parberry [15], Murase [24] propone una solución basada en la división del proceso de generación en una serie de etapas. Estas etapas fueron nombradas generación, verificación y evaluación. Primero se hace un diseño de nivel a través de una combinación aleatoria de *templates* de nivel y la ubicación aleatoria de los elementos del juego, luego se utiliza un *solver* de búsqueda en amplitud (BFS) para verificar una solución. Uno de los principales problemas con este enfoque de generación es la restricción de longitud de la solución, ya que BFS sólo logra resolver soluciones de secuencias cortas.

Un método reciente basado en algoritmos evolutivos y *Pattern Databases* (PDB) [25] permite generar estados iniciales de Sokoban complicados de solucionar [19]. Proponen métricas basadas en las PDB para calificar la dificultad de los estados iniciales creados. Además hacen uso del algoritmo *Novelty Search* [26] para garantizar diversidad en cada generación de individuos. El sistema usado para generar los estados iniciales lo llaman β, herramienta que permitió generar niveles más difíciles de resolver que los creados por humanos.

Unas variantes de algoritmos *Novelty Search* fueron investigadas por Liapis et al. [27] y se han explorado aún más en generación de niveles jugables de videojuegos [28], donde se presentan dos métodos inspirados en FI-2pop [29]. Ambos algoritmos evolucionan dos poblaciones distintas, una con individuos factibles y la otra no, adicionalmente cada una maneja su propio método de selección. Sus resultados dan cuenta de que los métodos de novedad restringidos de dos poblaciones pueden generar con la ayuda de ciertas condiciones, grupos más grandes y diferentes de niveles jugables que los métodos actuales de *Novelty Search*.

Monte-Carlo Tree Search es uno de los algoritmos más populares en el campo de la inteligencia artificial por su uso en investigación para juegos de mesa y de cartas. Se ha usado en la generación de niveles de Sokoban de diversos tamaños y dificultades [30]. Es aplicado definiendo la generación de rompecabezas como un problema de optimización. Este enfoque ha sido exitoso para otros problemas con altos factores de ramificación, la estructura de árbol de búsqueda garantiza la capacidad de solución.

Machine Learning forma parte de las herramientas usadas para enfrentar PCG en distintas áreas. Por ejemplo haciendo uso de UnityVGDL[1], sumado a las herramientas de *ML* que contiene el motor de videojuegos Unity, se logran entrenar y ejecutar agentes en juegos descritos por Videogame Description Language (VGDL) [31] como Sokoban [32].

Suleman et al. [20] proponen una alternativa basada en aprendizaje reforzado. Los autores entrenan un modelo de red neuronal con el que encuentran estados de solución de Sokoban. Ellos destacan que han podido verificar la validez de un rompecabezas con una precisión del 99 %. Adicionalmente entrenaron su red neuronal para generar el siguiente estado posible de un estado cualquiera con una exactitud del 99 %. El enfoque propuesto es superior en tiempo a los basados en A*, pero el máximo posible de tableros resueltos es 8 x 8. Argumentan que no han podido probar con tableros de 9 x 9 o mayores porque la mayoría de los *solvers* actuales se atascan con estos problemas. Pero esto podría cambiar en los siguientes años ya que ha sido creado un nuevo *solver* llamado Festival [33], el cual podría brindar nueva información de entrenamiento para alimentar estos algoritmos de RL, ya que es el único capaz de resolver los 90 niveles del conjunto xSokoban [34] en 900 minutos. Festival usa un algoritmo llamado *Feature Space Search algorithm* (FESS) el cual guía su búsqueda entre dos estados por medio de transiciones en el espacio de características del dominio de nuestro problema [33].

En la literatura encontramos variados modelos que implementan ML para la generación y resolución de distintos puzles. Hald et al. [35] emplean *Generative Adversarial Networks (GANs)* para generar niveles del puzle *Lily's Garden*. También encontramos aplicaciones para juegos como Doom [36] o Mario Bros [37] usando redes generativas antagónicas (GANs) para la creación de nuevos niveles de juego. Si bien hay muchas implementaciones de algoritmos PCGML para una amplia variedad de juegos, para puzles *Sokoban-Type* hay muy pocas, lo que permite vislumbrar nuevas oportunidades para enfrentar PCG en este tipo de juegos con la ayuda de algoritmos ML.

Una técnica que ha emergido con aplicaciones interesantes es el algoritmo *Wave Function Collapse* (WFC). Como se aprecia en la figura 2.1 [38] su uso puede dar lugar a diversos resultados muy interesantes. Se ha usado para PCG de mapas tridi-

[1]Implementación del lenguaje de descripción de videojuegos VGDL en el motor de juegos de Unity

mensionales [39], bidimensionales [40], para producir planes de ruta para NPCs[2] [41], entre otros. WFC es un algoritmo *non-backtracking* y *Greedy* que es conocido generalmente por su capacidad para tomar una imagen como entrada y generar otras similares [42].

En el cuadro 2.1 se presenta un resumen de los métodos implementados para la generación procedural de niveles en Sokoban.

2.2. Dificultad

Un área de alto interés en la generación procedural de niveles de juego es cómo clasificarlos en base a su dificultad. Mantere y Koljonen [43] usan algoritmos evolutivos para resolver, crear y calificar niveles de Sudoku. Utilizan un *solver* basado en GA para obtener la dificultad del nivel, el supuesto es que los niveles que tardan más en ser resueltos serán percibidos como niveles desafiantes al ser jugados por seres humanos. En el año 2010 [44] se vuelve a implementar algoritmos evolutivos para resolver niveles de Sokoban que eran generados de forma aleatoria. El tiempo promedio empleado por el algoritmo genético junto a su probabilidad de fallo fue utilizado para evaluar la dificultad del tablero. Ese mismo año se describen dos métodos para calificar dificultad [10]. El primero es una métrica de descomposición de problemas tomando como una unidad del problema a una caja o un conjunto del total de cajas del nivel y la otra es un modelo computacional que simula el recorrido humano en un espacio de búsqueda dado. En el año 2015 [45] se investigaron puzles como Flow, Lazors y Move para elaborar un método para cálculo de dificultad basado en el tamaño del nivel, el número de obstáculos, número de curvas en una ruta de solución, para combinarlos y dar origen en una función que evaluara la dificultad de los tableros generados.

Para medir que tan interesante o desafiante era un nivel generado [16] se plantearon la hipótesis "No hay diferencia entre los grados de atención de los jugadores al jugar los niveles generados por software versus sus grados de atención al jugar los niveles hechos a mano", para ello presentaron los niveles de Sokoban creados por software y por humanos a estudiantes, los cuales por medio de un *Stroop test*[3],

[2]Personaje no jugable controlado por el videojuego.

[3]El test de Stroop es un test psicológico vinculado especialmente a la parapsicología que permite medir el nivel de interferencia generada por los automatismos en la realización de una tarea

Specific Method	**Constructive vs Generative-and-test**	**Evaluation (Direct vs Simulation- based)**	**Degree and dimension of control**	**Quality Consideration**
Random template combination + BFS test [53]	Generate-and-test	Simulation-based: automatic solver	Level templates	Uninteresting candidates removed
Random template combination + heuristic-guided search [70]	Search-based	Metrics from backward play	Level templates	Reject easy / similar levels
MCTS using gameplay moves [32]	Search-based, whit simulated gameplay	Direct: level layout metrics	Eval. function	Quality improves incrementally
Data-driven MCTS [33]	Search-based, whit simulated gameplay	Direct: features correlated whit difficulty	Eval. function	Addresses difficulty
Large-scale BFS whit endgame data [66]	Search-based	Direct: using end game data base	No much control	Addresses difficulty
Novelty search whit PDB heuristic [4]	Search-based	Simulation-based: automatic solver	No much control	Addresses difficulty
Random template combination + Genetic Algorithm	Search-based	Simulation-based: automatic solver	No much control	Addresses difficulty

Cuadro 2.1: Resumen de métodos PCG para la generación de niveles en puzles *Sokoban-type* [1]

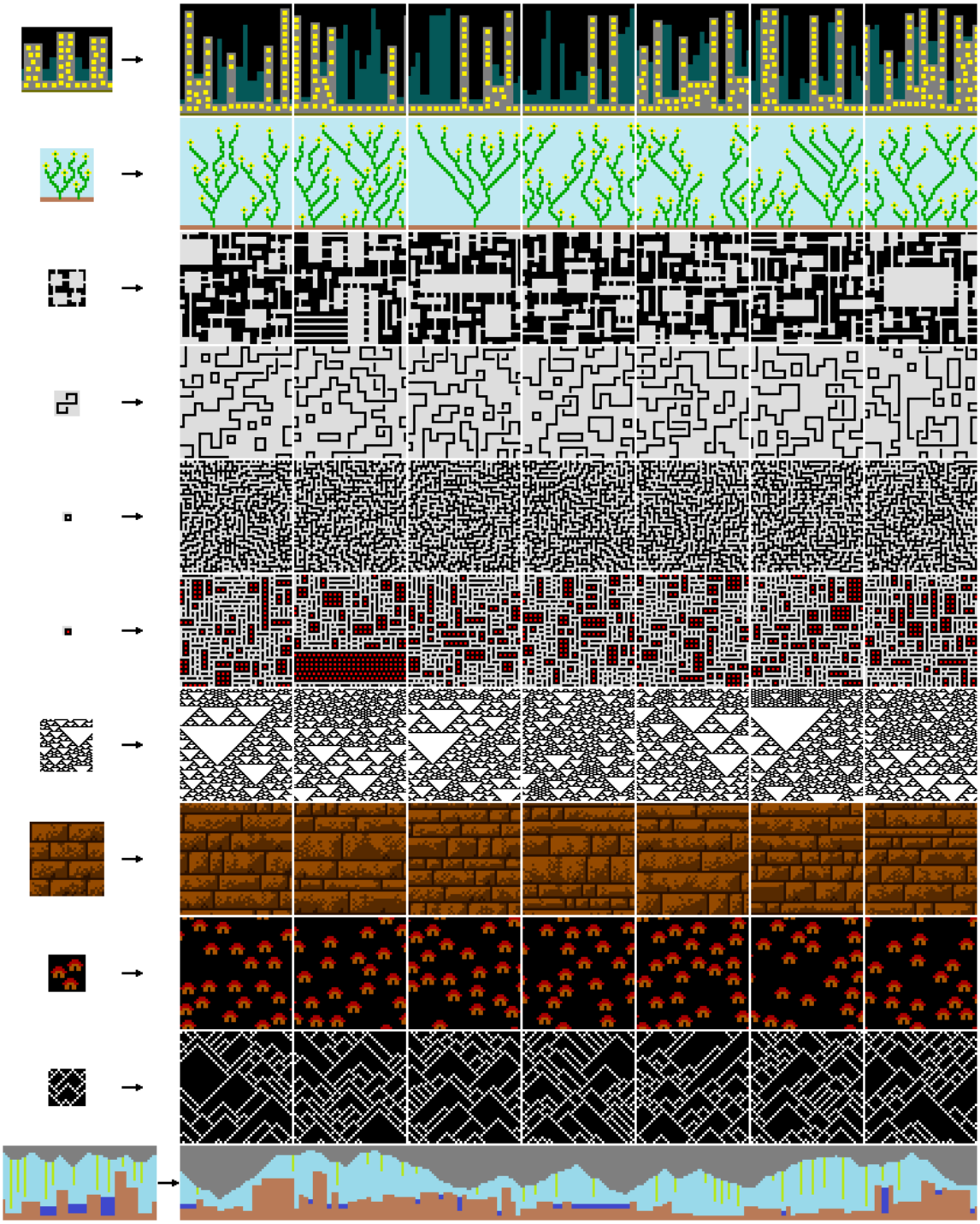


Figura 2.1: Ejemplos utilizando el algoritmo Wave Function Collapse [38] para el cual se toma como entrada una imagen que describa un patrón

midieron la reacción de del usuario a la información en conflicto percibida.

2.2.1. Resolución de niveles de Sokoban

Crippa y Marrochi [46] evaluaron el rendimiento de MCTS aplicado a Sokoban además de presentar una comparación con otro algoritmo llamado *Iterative Deepening A** (IDA*) [47]. Concluyen que a pesar de las optimizaciones hechas a MCTS, este método no iguala a IDA* en cuanto al número de niveles resueltos.

La mayoría de las técnicas mostradas en esta sección hacen búsquedas en un árbol de estados o resuelven el problema a través de métodos de optimización. Bhaumik et al. [48] hacen una comparación entre estos dos tipos de enfoques, examinan el rendimiento de métodos basados en búsquedas en árbol como *Breadth First Search* (BFS), *Depth First Search* (DFS), *Greedy Best First Search* (GBFS), *Monte-Carlo Tree Search* y algoritmos de optimización como *Hill Climbing* (HC), *Simulated Annealing* (SA), *Evolution Strategy* (ES) y algoritmos evolutivos. El estudio fue llevado a cabo en los juegos Sokoban, Legend of Zelda y Binary. Los experimentos consisten en ejecutar los 8 algoritmos antes mencionados en tableros con tamaños similares y analizar el comportamiento junto al resultado final. Introducen las representaciones *Turtle*, *Narrow* y *Wide* útiles para combatir el problema del alto factor de ramificación que puede afectar el rendimiento de la búsqueda de árboles [48]. Sus conclusiones son:

1. MCTS no logró un buen rendimiento en los juegos como Binary y Zelda, pero superó las expectativas en Sokoban. El mal rendimiento puede ser debido al atascamiento en un óptimo local. Sin embargo, se cree que las capacidades de muestreo aleatorio podrían prevenir estas situaciones en MCTS.

2. Los niveles generados por cada algoritmo son similares, pero tienen diferencias sutiles, lo que revela que cada uno tiene un estilo distinto para encontrar una solución. Si bien los algoritmos de optimización a menudo llegan a resultados cuantitativamente mejores, los algoritmos de búsqueda de árbol mezclados con la representación correcta pueden resolver el mismo problema de una forma distinta.

3. Marco Teórico

En esta sección se presenta de forma general los conceptos y técnicas más utilizadas en la generación de contenido procedural tomando como base lo expuesto en la sección anterior, entre los métodos expuestos se encuentran algoritmos evolutivos, Monte-Carlo Tree Search y Machine Learning.

3.1. Algoritmos Evolutivos

El término algoritmos evolutivos refiere a un conjunto de modelos computacionales basados en la evolución, este tipo de algoritmos son a menudo vistos como funciones optimizadoras [49], sus aplicaciones abarcan áreas de todo tipo, algunas de ellas son industriales [50], bioinformática [51], electromagnetismo [52], videojuegos e inteligencia artificial [53], entre otras.

Existen muchas implementaciones de GA distintas, pero todas parten de una base similar. La esencia de la implementación toma como punto de partida la idea de la selección natural, en el cuadro 3.1 se puede observar la analogía entre la biología y la manera que GA se adapta a las distintas problemáticas [54]. En la siguiente lista se describe las etapas generales que sigue un algoritmo evolutivo.

1. Inicio: Se genera un número aleatorio de cromosomas.
2. *Fitness Function*: Se evalúa cada cromosoma con la ayuda de una función llamada *Fitness Function*.
3. Nueva población: Se crea la nueva población siguiendo las etapas descritas a continuación.

   a) Selección: Selecciona dos cromosomas teniendo como criterio el valor entregado por la función de aptitud. Un alto valor da al cromosoma mayores probabilidades de ser seleccionado.

   b) Cruza (*Crossover*): Se combina la información de un cromosoma con otro, gracias a este pasos se permite la exploración de nuevas soluciones del espacio de búsqueda.

   c) Mutación: Etapa en la cual se alteran las soluciones para no estancar el proceso evolutivo.

   d) Se coloca la nueva descendencia en la población.

4. Reemplazo: Podemos utilizar métodos de reemplazamiento aleatorios, o deterministas. En esta parte del proceso el programador puede decidir si conservar los mejores cromosomas o no hacerlo, esta acción es llamada elitismo.

5. Prueba: Si se cumple la condición final se detiene el algoritmo y se devuelve la mejor solución de la población actual.

Biología	**Algoritmos evolutivos**
Cromosoma o genotipo	Estructura o cadena de datos (número binario)
Locus	Un particular (bit) o posición en la cadena de datos
Fenotipo	Set de parámetros o vector de soluciones

Cuadro 3.1: Analogía Biología - Algoritmos evolutivos

3.1.1. Quality Diversity Algorithms

Esta familia de algoritmos genera una colección de soluciones de alta calidad en vez de solo una. Los elementos que conforman este tipo de métodos son: un contenedor, un operador de selección y el valor a optimizar [55]. Como se puede apreciar en el cuadro 3.2 hay distintos algoritmos basados en esta idea, estos no se guían por una aptitud enlazada al objetivo del problema, sino que se recompensa la variación en las soluciones por medio de la novedad, la sorpresa y la curiosidad inherentes en ellas.

Algoritmo	**Descripción**
MAP-Elites (ME)	MAP-Elites produce una gran diversidad de soluciones de alto rendimiento. Pertenece a la categoría de Illumination Algorithms. Al igual que GA las fases de ejecución son selección, mutación, evaluación y sustitución, las que se repiten hasta que se alcanza la cobertura deseada.
Constrained Novelty Search (CNS)	Técnica que combina el algoritmo genético de dos poblaciones viables infalibles (FI-2Pop) con NS, con el objetivo de maximizar la distancia espacial de comportamiento de los individuos de la población que cumplen con las condiciones de calidad existentes.
Constrained Surprise Search (CSS)	Similar al algoritmo anterior, este método utiliza la imprevisibilidad como métrica para evaluar la diversidad de los individuos viables en el algoritmo FI-2Pop.
Novelty Search with Local Competition (NS-LC)	Este algoritmo emplea un enfoque de múltiples objetivos que maximiza tanto la valoración de la novedad como el objetivo de competencia local que empuja al individuo a vencer a otros en su nicho.
Surprise Search with Local Competition (SS-LC)	Parecido a NS-LC, mide la desviación de las predicciones realizadas sobre generaciones anteriores, lo que resulta en una medida diferente a la divergencia de la novedad.

Cuadro 3.2: Quality Diversity Algorithms [56]

3.1.2. Novelty Search Algorithm

Novelty Search forma parte de la familia *Quality Diversity Algorithms* [56], los que tienen como meta encontrar diversas soluciones eficientes en lugar de una sola pero de mejor rendimiento [57]. Se ha comprobado que este método de búsqueda puede superar a la búsqueda basada en objetivos, su uso es considerado muy útil para la resolución de problemas de programación genética difíciles [58].

Un problema muy común en la búsqueda de soluciones usando métodos de optimización es la convergencia prematura a óptimos locales, los que impiden que la evolución pueda dirigirse hacia otros espacios de búsqueda [58]. Como respuesta a este comportamiento se desarrolló *Novelty Search* (NS). Este algoritmo trabaja sobre la base de explorar el espacio de búsqueda sin tener en cuenta los objetivos, por medio de un comportamiento aleatorio uniforme en el espacio de estados [57]. En cada iteración, el valor resultante de la *Fitness Function* cambia, ya que es probable que un individuo que se consideraba nuevo en una generación sea mucho menos novedoso en la siguiente.

3.2. Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) es un enfoque general para solucionar problemas asociados a diversos problemas, ha desempeñado un papel protagonista en AlphaZero de Google DeepMind y su predecesor AlphaGo, el cual derrotó al campeón mundial de Go (humano) Lee Sedol en 2016 y al jugador de Go número uno del mundo Ke Jie en 2017 [59], estas son solo algunas evidencias del potencial de esta técnica. A diferencia de los algoritmos clásicos, MCTS no necesita una función heurística de evaluación, ya que ejecuta una exploración aleatoria en el espacio de búsqueda, a medida que avanza en esta exploración construye un árbol con los resultados de las exploraciones [60].

Cada nodo del árbol de búsqueda simboliza un estado del dominio y los enlaces dirigidos a los nodos secundarios representan acciones que conducen a estados posteriores. La construcción del árbol de búsqueda se desarrolla de forma iterativa en 4 etapas [61]:

1. Selección: Iniciando en el nodo raíz, se generan los nodos hijos de forma recursiva con ayuda de la política de selección. Un nodo capaz de expandir es un

nodo no terminal que contiene hijos no visitados.

2. Expansión: Se agregan nodos al árbol tomando en cuenta las acciones disponibles.

3. Simulación: Es ejecutada una simulación a partir de los nuevos nodos de acuerdo con la política predeterminada para producir un resultado.

4. *Backpropagation*: El resultado de la simulación es transmitido hacia los nodos antecesores actualizando las estadísticas.

3.3. Machine Learning

Machine Learning (ML) se puede definir de forma general como un conjunto de métodos computacionales que usan las experiencias pasadas para un mejor rendimiento o la ejecución de predicciones precisas [62]. La idea de aprender en base a experiencias pasadas es algo inherente en la naturaleza del ser humano y es la esencia del *Machine Learning*. El algoritmo que implementa ML descubre y aprende gracias a que se nutre de los datos para seguir puliéndose en su propósito, y es bajo esta idea donde podemos asociar el recuerdo o las experiencias humanas a la habilidad de aprendizaje en los métodos de *Machine Learning* existentes.

Este concepto lo podemos clasificar en base al conjunto de datos que alimentaran nuestro algoritmo y el problema dado.

1. *Supervised Learning*: El algoritmo intenta encontrar las relaciones entre el conjunto de características y el conjunto de etiquetas entregado. Si los datos que nutren el algoritmo son etiquetas, el problema de aprendizaje tiene por nombre clasificación, por otro lado, si el conjunto de datos pertenece al conjunto de los números reales, el problema tiene por nombre regresión. Como se puede apreciar en el cuadro 3.3, hay un conjunto de métodos que permiten realizar aprendizaje supervisado , la categorización está basada en el tipo de conjuntos de la hipótesis [63].

2. *Unsupervised Learning*: Los datos que se le proporcionan al algoritmo no están etiquetados. Tiene como objetivo agrupar la información que se le suministra, la búsqueda de relaciones y la disminución de la dimensionalidad. Al igual que

Category	Important Methods
Linear model	▪ Perceptron, multi-layer perceptron (MLP) ▪ Support vector machine (SVM) ▪ Support vector regression ▪ Linear regressor, Rigid regression ▪ Logistic regression
Non-parametric model	▪ K-nearest neighbors ▪ Kernel denssity estimation ▪ Kernel regression, Local regression
Non-metric model	Clasification and regression tree (CART), decision tree
Parametric model	▪ Naive Bayes ▪ Gaussian discrimant analysis (GDA) ▪ Hidden Markov models (HMM) ▪ Probabilistic graphical models ▪ Convolutional Neural Networks ▪ Generative Adversarial Network ▪ Recurrent Neural Network ▪ Long short-term memory

Cuadro 3.3: Métodos aplicados en *Supervised Learning*

en el ítem anterior, se adjuntan en el cuadro 3.4 los métodos usados con mayor frecuencia.

3. *Reinforcement Learning*: Posee un enfoque en el cual permite que los agentes que actúan en un mundo estocástico desconocido aprendan observando los estados que suceden y evaluando por medio de las recompensas que se dan en cada uno de ellos [64]. Es utilizado para solucionar problemas de toma de elecciones, como podría ser el movimiento de un robot o la conducción automática de autos. Las técnicas de *Reinforcement Learning* más comunes se muestran en el cuadro 3.5 [65].

ML habitualmente está compuesto por un modelado del problema (conjunto de hipótesis y función objetivo) y optimización, la parte necesaria para llevar a cabo ML es un conjunto de datos adecuados para el aprendizaje del algoritmo a desarrollar [63].

Category	Important Methods
Clustering	- K-means clustering - Spectral clustering
Density Estimation	- Gaussian mixture model (GMM) - Graphical models
Dimensionality reduction	- Principal component analysis (PCA) - Factor analysis

Cuadro 3.4: Métodos aplicados en *Unsupervised Learning*

Model	Category	Important Methods
Model-Free RL	Q-Learning	- Deep Q-Learning. - Categorical 51-Atom DQN. - Quantile Regression DQN. - Hindsight Experience Replay.
	Policy Optimization	- Policy gradient. - Asynchronous Advantage Actor-Critic. - Proximal Policy Optimization. - Trust Region Policy Optimization.
Model-Based RL	Learn model	- World Models. - Imagination-Augmented Agents. - Model-Based RL whit Model-Free Fine-Tuning. - Model-Based Value Expansion.
	Given model	- AlphaZero.

Cuadro 3.5: Métodos aplicados en *Reinforcement Learning*

4. Metodología

Tal cómo se ha visto en la sección 2 la mayoría de los investigadores llevan a cabo su generación en fases. Hemos ideado un nuevo proceso bajo 2 etapas para generar niveles de Sokoban. Estas etapas las identificamos como: Etapa de inicialización y Etapa de Evolución, la etapa de Inicialización comprende dos subetapas llamadas Etapa de Construcción y Etapa de Preparación. A continuación, describimos cómo ellas ayudan en la generación de los niveles.

4.1. Etapa de Inicialización

En esta etapa nos enfocamos en crear la base del tablero (solo murallas y espacios vacíos) y el vecindario inicial de nuestro algoritmo evolutivo. El vecindario está compuesto por una cantidad definida de pares caja-meta las cuales serán combinadas eventualmente sobre esta base de murallas. En la siguiente sección explicamos en mayor detalle cada subetapa.

4.1.1. Etapa de Construcción

Esta es la primera fase que se lleva a cabo, el resultado se puede apreciar en la imagen 4.2. Para llegar a esta estructura ejecutamos las siguientes subetapas:

1. Ajuste de dimensión del nivel: La dimensión de los tableros es elegida de forma aleatoria. Primero es seleccionado un factor entre 2 y 5, este valor representa la cantidad de *templates* (ver figura 4.3) por eje. Valores menores a 2 producen tableros demasiado pequeños y valores mayores a 5 generan tableros demasiado grandes, lo que implicaría esperas muy altas para la Fase de Evolución, ya

que que el *solver* es ejecutado reiteradas veces y una dimensión de tablero gigante tampoco garantiza mayor dificultad o entretención. Estos factores serán multiplicados por 3, lo que nos da tableros de un mínimo de 6 y máximo 15 *tiles*. Necesitamos multiplicar por 3 porque la dimensión de los *templates* utilizados en la etapa siguiente es 3, de esta manera la ubicación encaja de forma exacta con las dimensiones aleatorias del tablero. Si multiplicamos por un número distinto de 3 tendríamos que ver otras estrategias para ubicar los *templates* en la base del tablero.

2. Ubicación de *templates*: El proceso se resume gráficamente en la figura 4.1 para un tablero de 3 *templates* de ancho y 2 *templates* de alto. Los *templates* utilizados se pueden observar en la imagen 4.3. Estos son de 5 x 5 *tiles*, su contorno es necesario para que no se originen callejones cerrados cuando vamos combinándolos. Al ubicarlos se van solapando tal como se muestra en la figura 4.1.

   Cuando ocurre el solapamiento entre cada *template* existe la condición de que los espacios libres (*tiles* cafés) no sean solapados por murallas, ya que si esto ocurre es probable que haya un mayor número de laberintos y callejones, por lo que en cada ubicación de los *templates* se debe comprobar si no se producen este tipo de conflictos con los *tiles* que están a su alrededor.

   El proceder general es el siguiente:

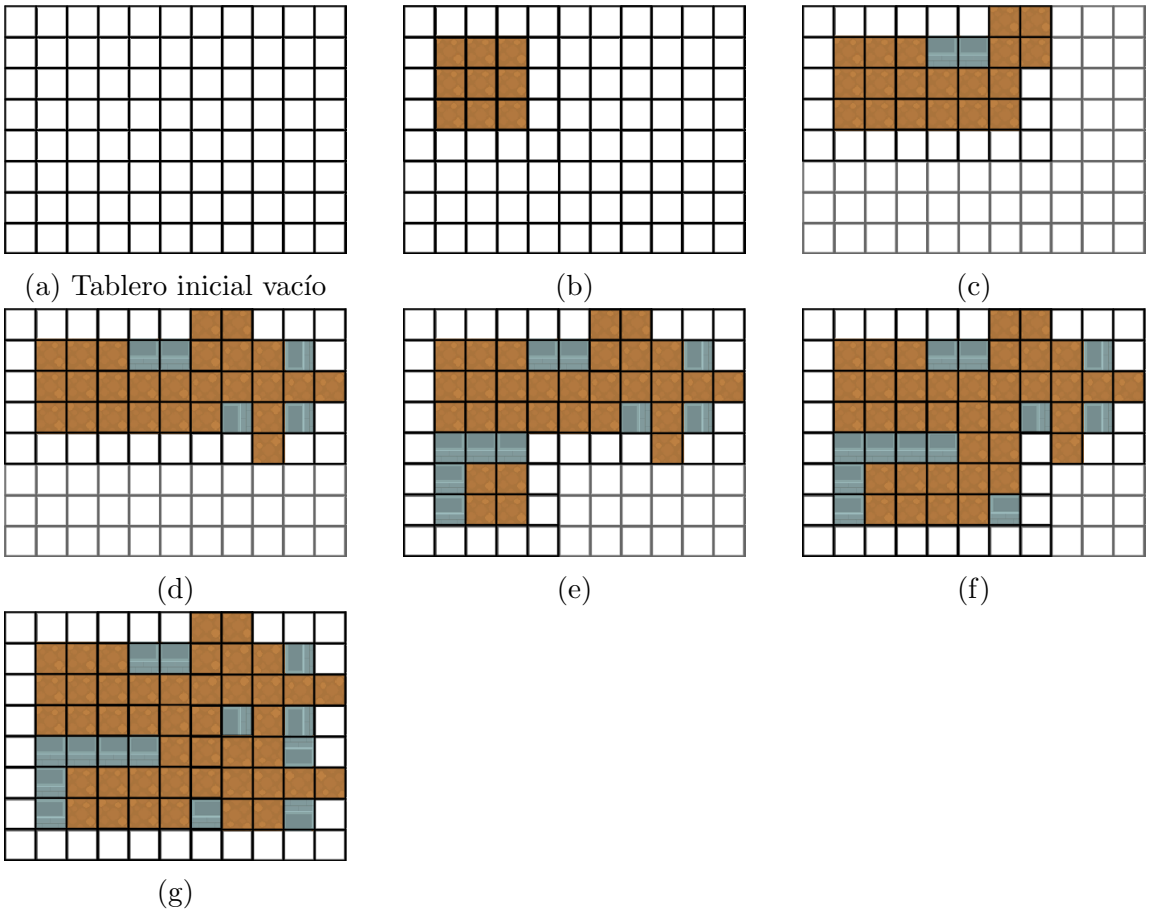
   a) Se itera por cada *template* del conjunto mostrado en la imagen 4.3 analizando si no existen conflictos.

   b) Este proceso de análisis se repite para el *template* rotado y volteado en el eje X e Y.

   c) Si el análisis da como resultado que el *template* puede ser colocado, este es agregado a una lista de candidatos.

   d) Finalmente se escoge y ubica un *template* al azar de la lista de candidatos mencionada en el punto anterior.

3. Post procesado: Las siguientes restricciones son llevadas a cabo para desechar algunos niveles deficientes.



(a) Tablero inicial vacío (b) (c)

(d) (e) (f)

(g)

Figura 4.1: Ubicación de cada *template* sobre las dimensiones de tablero previamente determinadas. Figura b, c, d, e, f y g muestran la ubicación de cada *template* sobre la base de tablero vacía

a) Espacios grandes: Niveles que contenían espacios grandes vacíos de 4x3 y 3x4 *tiles* son descartados por nuestro proceso PCG. Al analizarlos de manera visual nos dimos cuenta de que eran niveles a los cuales nuestro algoritmo genético demoraba en combinar y además eran triviales de resolver.

b) Algoritmo Flood Fill [66]: Otra consideración es que los niveles debían tener todos sus espacios vacíos conectados, a veces se podían crear habitaciones aisladas por una línea de murallas lo que generaría problemas en las etapas posteriores. Con el algoritmo recursivo Flood Fill garantizamos que los niveles fueran recorridos en su totalidad por el agente, ya que se explora la estructura del nivel verificando que todos los espacios vacíos

estén conectados.

c) Exceso de laberintos: El tener demasiados lugares estrechos generaban tableros triviales los que tenían largos de solución reducidas, ya que tableros de juego con espacios vacíos demasiados estrechos impedían la ubicación de cajas y metas sobre este en la etapa de evolución.

d) Exceso de murallas: El exceso de murallas también era un factor ya que al tener demasiado lugares cerrados y ajustados se generaban niveles poco interesantes, en los que regularmente no caían muchas cajas, por lo que estos también eran triviales de solucionar.

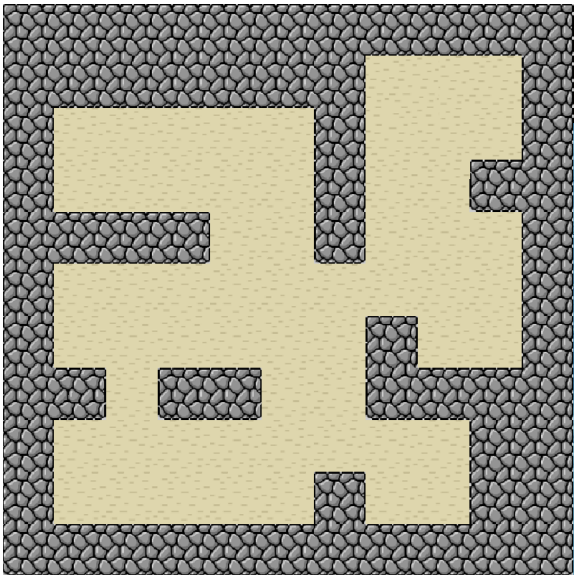


Figura 4.2: Resultado final fase construcción

4.1.2. Etapa de preparación

A continuación, creamos la vecindad inicial de individuos a ser utilizada en el algoritmo genético del proceso de evolución. La idea general tras nuestro algoritmo evolutivo es dividir un problema complejo (tablero de Sokoban con muchas cajas y metas) en sub-problemas (sólo un par caja - meta). Comúnmente los tableros de Sokoban poseen mas de una caja y meta, por lo que llamaremos a cada par caja - meta un subproblema. Y este sub-problema será parte de los genes de nuestros individuos, los que combinaremos generando tableros cada vez mas complejos.

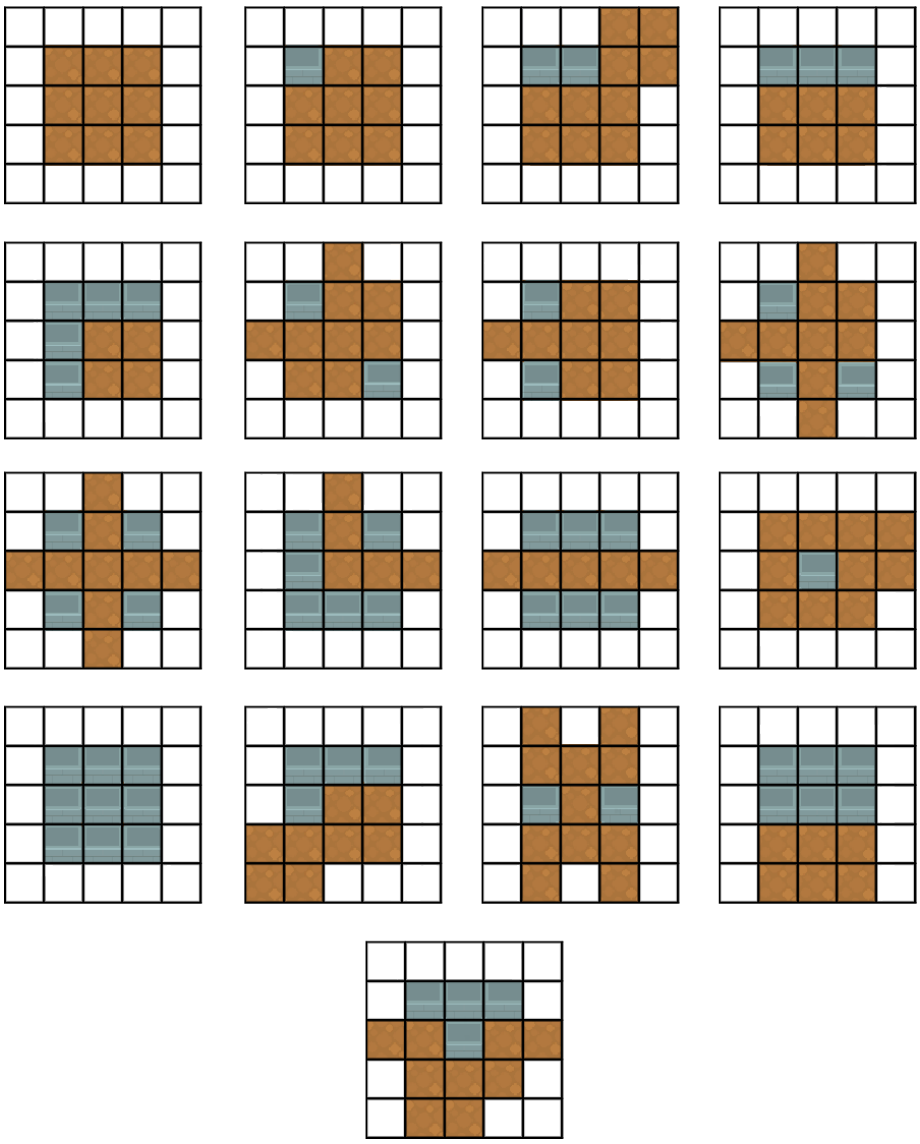


Figura 4.3: *Templates* utilizados en la etapa de construcción

Los datos que tendrán los individuos entonces serán:

Pares caja - meta: Lista de pares caja - meta que cada tablero contendrá.

Ruta solución: Ruta solución de cada par caja - meta. Esto nos será útil en la etapa evolutiva.

Para crear el vecindario inicial de nuestro algoritmo evolutivo utilizaremos un término introducido en el artículo de Murase [24], el *Goal-Range*. Llaman *Goal-Range* a los *tiles* del tablero en el cual para un tile dado (meta) podemos ubicar una caja en un rango de *tiles* de tal manera que aseguremos su solución. En la figura 4.4 se puede ver diversos círculos que componen el *Goal Range*: el nodo 'G' es la meta y el nodo 'B' es la caja. Luego seleccionamos el *tile* más alejado de la meta dentro del *Goal-Range* para posicionar su caja.

Ahora que ya se tiene una manera de crear el vecindario inicial, explicaremos las subetapas de esta fase:

1. Ajustar la capacidad del vecindario inicial: Puede ocurrir que algunos tableros tengan muchas opciones debido al tamaño del *Goal Range*, por lo que los pares caja-meta posibles son muchos para una meta dada. Limitamos esta cantidad a 10 si es que hay demasiados. Creemos que esto puede ser mejorado buscando todos los candidatos posibles y filtrar por ejemplo seleccionando los que presenten una longitud de solución mayor. De esta forma la población inicial podría reflejar niveles más complicados al final de la etapa de evolución.

2. Ruta solución del par caja - meta: Almacenamos la ruta solución para ese par caja-meta, porque nos servirá en la etapa de evolución. Algunos pares caja-meta contienen rutas de solución muy cortas. Las que poseen una longitud menor a 4 son rechazadas y se busca otro par enseguida. La ruta solución es calculada gracias al árbol n-ario que describe el goal-range. Por medio de este árbol podemos iterar recursivamente partiendo desde el *tile* donde está ubicada la caja hasta llegar al *tile* donde está la meta.

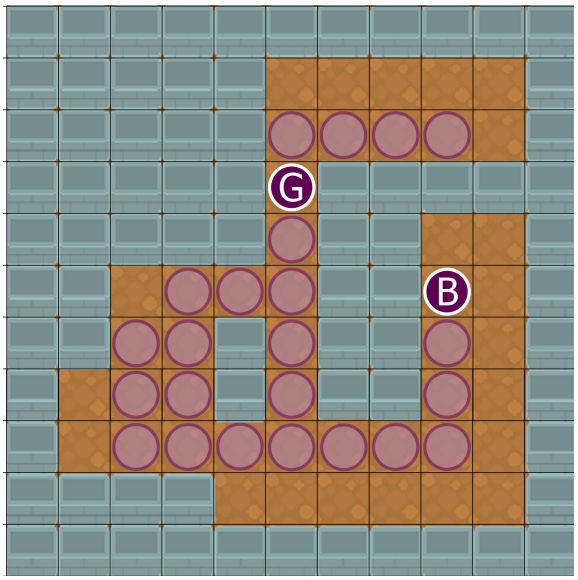


Figura 4.4: *Goal Range*, círculos rosados componen el *Goal Range*. El círculo con la letra G es la meta y el de la letra B es la caja

4.2. Etapa de evolución

En la segunda etapa toma lugar la implementación de un algoritmo genético para ubicar cajas, metas y jugador en el nivel. Hemos hecho uso de la librería Jenes [67][1] y el software de código libre JSoko, el cual es usado generalmente para creación y solución de tableros Sokoban.

Los usos que le hemos dado a cada herramienta son:

1. Jenes: Librería liviana y de fácil implementación desarrollada en lenguaje Java. Nos facilitó la programación del algoritmo evolutivo, proporcionando la estructura base para poder extender el código y así ajustarlo a nuestro problema.

2. JSoko: Herramienta que entre sus características posee la capacidad para solucionar niveles, optimizar soluciones, edición de niveles, jugar Sokoban, entre otros. Nos ha servido mayormente para utilizar el *solver* implementado en ella. También hemos podido hacer depuración de la Etapa de Evolución y tener retroalimentación visual sobre cómo van cambiando los niveles en el proceso.

[1]Librería de bajo peso, fácil uso y de código abierto desarrollado y por Intelligentia s.r.l. En colaboración con el Laboratorio de Ingeniería de Sistemas Inteligentes y Computacionales (CISELab) de la Universidad de Sannio

4.2.1. Genes

Todo algoritmo genético necesita una manera de dar forma a sus individuos, esta debe satisfacer la mayor cantidad de escenarios posibles. Para cumplir con esta tarea se han usado estructuras basadas en matrices, *binary array*, listas de números flotantes, grafos dirigidos y lenguaje natural. Para nuestro problema creemos que usar una matriz de caracteres es una buena idea, ya que es la misma representación que usa JSoko para sus niveles. Otra razón para usar este tipo de representación es que funciona bien con problemas bidimensionales frente otras representaciones como *strings* [68]. La imagen 4.5 muestra un ejemplo de cómo codificaremos los niveles de Sokoban, y la tabla 4.1 presenta la simbología utilizada.

```
########################
#                      #
# ####################$##
# #...#              #  #
# #...# $$$$$$$ # #  #
# #..$  $      $#. $  #
# #   #   $$$$$.# #  #
# #...# # .....#   .###
# ##################$  #
#+                     #
########################
```

Figura 4.5: Ejemplo de codificación de un nivel de Sokoban

4.2.2. Función de fitness

Para saber qué nivel es más apto a seguir formando parte de la evolución, necesitamos una función de evaluación que nos proporcione una estimación de cuan apto es. La métrica que tomaremos en cuenta para conservar o desechar algunos individuos será la dificultad de resolución de estos. Para obtener estos valores haremos uso del *solver* de JSoko (ver apéndice C), específicamente el *solver* AStar el cual entrega la solución que minimiza los empujes realizados sobre las cajas. Haremos uso de este valor como estimación de dificultad.

Otra métrica puede haber sido el conteo de cambios de dirección en el movimiento de las cajas, esta es una buena métrica ya que deja fuera los movimientos en línea recta de las cajas, pero es más complicada de implementar [15].

4.2.3. Fases del algoritmo evolutivo

Cómo se explicó en la sección 3, los algoritmos evolutivos necesitan ciertos componentes básicos para funcionar, una de ellas es la representación del problema que permita identificar a la eventual solución. También hacen uso de una función de *fitness* que permita estimar la calidad de una solución en particular, además de una política de *crossover* que combine la información de dos individuos y se pueda seguir explorando el espacio solución. A continuación, se detallan estos elementos y su consecuencia en la generación de Niveles Interesantes.

1. Selección: En esta etapa debemos seleccionar dos individuos para intercambiar su información genética. La selección es realizada por el método de torneo. Se seleccionan dos individuos de forma aleatoria, y el que tenga mejor valor de *fitness* es el individuo escogido.

2. Crossover: La etapa de *crossover* nos permite combinar la información de los niveles, específicamente sus pares de cajas - metas. El proceder es como se detalla a continuación:

   a) Preparación: Se inicializa el *crossover* tal como se ve en el algoritmo 1. Las etapas son las siguientes:

Elemento	**Símbolo**
Muralla	#
Jugador	@
Jugador sobre meta	+
Caja	$
Caja Sobre meta	*
Meta	.
Suelo	(space)

Cuadro 4.1: Símbolos del formato de niveles de JSoko

1) En la línea 2 y 3 calculamos cuantos pares de cajas cederemos al otro cromosoma y viceversa. Pueden ocurrir las dos siguientes situaciones:
   (I) Cromosomas con 2 o más pares caja-meta: El valor es un número aleatorio entre 2 y la máxima cantidad de pares caja-meta que contiene el cromosoma. El mínimo es 2 porque no queremos dejar a los individuos sin información disponible para ser combinada.
   (II) Cromosomas con menos de 3 pares caja - meta: El valor puede ser 1 o 2 dependiendo de la cantidad de pares caja-meta que contiene el individuo.

   Este intercambio de cajas permite que generación tras generación los pares caja-meta de cada cromosoma vayan cambiando. Al compartir esta información entre los dos individuos que ingresan a la fase de *crossover*, se agregan en las pares caja-meta del cromosoma hermano al cromosoma que recibe cuando sea posible en las etapas posteriores a esta etapa de preparación (si es el nivel sigue teniendo solución). De esta forma los cromosomas comienzan con solo un par caja-meta y culminan el proceso con muchos otros pares más.
2) Selección de pares: En las líneas 4 y 5 seleccionamos aleatoriamente y duplicamos los pares caja-meta de los cromosomas almacenándolos en dos listas auxiliares.
3) Intentar agregar los candidatos de pares caja-meta: En la línea 6 se procede a intentar agregar cada candidato caja-meta a los genes del cromosoma 1. El mismo proceso es realizado en la línea 7 pero con el cromosoma 2 y pares de caja-meta del cromosoma 1.

b) Diagnóstico: Como se aprecia en el algoritmo 2 línea 2, analizamos cada par caja-meta candidato para ver si es posible agregarlo al cromosoma "chrom". En el algoritmo 3 comprobamos si la caja junto a su meta pueden dar origen a un nuevo tablero que siga teniendo solución. A continuación, se describe en mayor detalle este algoritmo:

1) En la línea 8 comenzamos analizando si la caja candidata está colisionando con alguna caja del cromosoma o el agente. Si hay colisión, se traslada la caja un valor entre 1 y 3 en la dirección de la ruta solución. Las constantes 1 y 3 fueron seleccionadas así porque generalmente los

largos de las rutas entre cada caja y meta no son muy extensos, el intentar moverlas lentamente puede dar lugar a una exploración más profunda que si las movemos valores muy altos en su ruta solución, ya que una vez estas llegan al límite de la ruta, no se siguen moviendo si es que no ocurre la mutación que invierte sus posiciones. Este proceso se repite hasta que no haya colisión de la caja ni el agente.

2) Luego en la línea 12 se ejecuta la misma acción anterior pero comprobando que la meta candidata no colisione con alguna meta del cromosoma, Si existe una colisión, trasladamos la meta un valor entre 1 y 3 en la dirección de la ruta solución. Este proceso se repite hasta que no haya colisión de la meta con otras metas del cromosoma.

3) Una vez se haya comprobado que no hay colisiones, actualizamos el nivel con la nueva información del par candidato en una matriz auxiliar.

4) Luego usamos el *solver* de JSoko para averiguar si la nueva configuración del nivel tiene solución. Si tiene solución, copiamos la información del par candidato en el nivel del cromosoma y agregamos este par candidato a la lista de pares caja - meta del cromosoma.

5) Cómo se puede apreciar en la línea 5, si el proceso anterior fue exitoso (se agregó el par candidato al cromosoma), removemos el par candidato del cromosoma al cual pertenece solo si este tiene más de 1 par caja-meta en su lista de pares caja-meta.

El proceso se repite también para el segundo cromosoma, tomando la lista de pares caja-meta del primer cromosoma. Si los procesos de diagnóstico fueron exitosos cada cromosoma tendría nueva información al finalizar la etapa de *crossover*.

3. Mutación: Realizamos dos tipos de mutaciones para reducir las probabilidades de que la evolución se estanque. Son las siguientes:

   a) Cambio de posición del jugador: El jugador es trasladado a una posición vacía del tablero, si el nuevo nivel sigue teniendo solución, se realiza el intercambio, si no, no se hacen cambios. Esta mutación le hemos otorgado 20 % de probabilidades de ser efectuada, si bien su valor de ocurrencia es

```
1: procedure InitCrossover (chromosome1, chromosome2)
2:     boxCount1 ← random number of boxes from chromosome1
3:     boxCount2 ← random number of boxes from chromosome 2
4:     boxexDatas1 ← Get boxes datas from chromosome 1
5:     boxexDatas2 ← Get boxes datas from chromosome 2
6:     Crossover(boxexDatas1, chromosome2, chromosome1).
7:     Crossover(boxexDatas2, chromosome1, chromosome2).
```

Algoritmo 1 Preparación, método llevado a cabo antes del *Crossover* para preparar la información a combinar

```
1: procedure Crossover (boxDataCandidates, chrom, srcChrom)
2:     for all boxDataCandidates do
3:         succes ← TryPutBox(chrom, boxDataCandidate)
4:         boxDatasLen ← length of chrom Box Datas List
5:         if succes == True  & boxDatasLen≥ 2 then
6:             RemoveBoxFromSrc(boxDataCandidate, srcChrom)
7:         AddBox(boxDataCandidate, chromosome)
```

Algoritmo 2 Crossover, método que intenta agregar nuevos pares caja-meta en un cromosoma

alto, esta no es ejecutada con frecuencia ya que el nuevo nivel no tiene solución siempre, por lo que la mutación es descartada.

b) Invertir posición de caja y meta: Seguimos la misma idea de la mutación anterior, pero en este caso invertimos un par caja-meta, también debemos comprobar que el nuevo nivel tiene solución, de forma contraria no hacemos cambios. Esta mutación le hemos otorgado 25 % de probabilidades de ser efectuada, al ser una mutación más agresiva que la del jugador creemos que su ejecución puede ayudarnos a explorar nuevas formas de solucionar un nivel.

Con el objetivo de saber cuántas de estas mutaciones efectivamente eran realizadas, se efectuó una prueba donde ejecutamos 10 veces el generador en un nivel de 3 x 3 *templates*, con un máximo de 12 generaciones para así tener los valores correspondientes al total de veces que son llamados los eventos de las mutaciones y el total de veces que si son realizadas. Ya que recordemos que para que la mutación sea ejecutada, el tablero debe tener solución, si no, esta mutación es desechada. La imagen 4.6 muestra

```
1: procedure TryPutBox (chrom, boxDataCandidate)
2:     routeLen ← length of boxRoute
3:     top:
4:     repeat
5:         boxPos ← boxRoute[boxIndexRoute]
6:         goalPos ← boxRoute[goalIndexRoute]
7:         for all boxesInChrom do
8:             if boxPos collision boxChro then
9:                 boxRoute ← boxRoute + 1;
10:                goto top.
11:        for all goalsInChrom do
12:            if goalPos collision goalChrom then
13:                goalRoute ← goalRoute − 1;
14:                goto top.
15:        hasSolution ← PerformExchange().▷ Check if new board has solution
16:    until hasSolution OR boxIndexRoute quality ≥ routeLen OR goalIndex-
   Route < 0
```

Algoritmo 3 Diagnóstico, método de la clase BoxData. BoxRoute, boxIndexRoute y goalIndexRoute estan contenidos en BoxData y representan respectivamente la ruta solución para la caja y meta de esta clase, el índice de la posición de la caja en la ruta solución y el índice de la posición de la meta en la ruta solución

un resumen de los resultados.

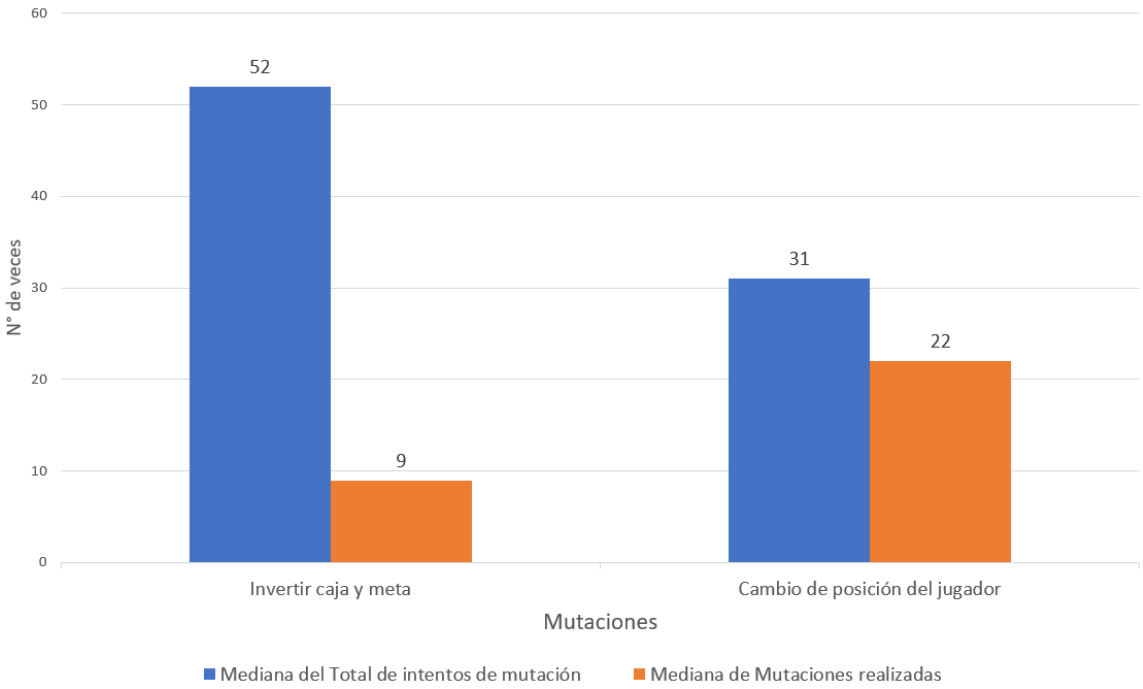


Figura 4.6: Cantidad de mutaciones efectuadas y rechazadas

4. Elitismo: El uso del elitismo en un algoritmo evolutivo no es obligatorio, pero para nuestro problema nos será útil ya que deseamos que la evolución progrese con mejores valores de *fitness* en cada generación. Es por lo que el incluir a los *parents* en la nueva población estaremos incluyendo potenciales individuos para una próxima etapa de *crossover*.

4.2.4. Limitaciones del algoritmo evolutivo

En esta sección listaremos algunas de las limitaciones del generador que hemos desarrollado:

1. El generador usa recurrentemente el *solver* de JSoko, si bien hacemos uso del *solver* en su versión *Any Solution* con el propósito de saber solamente si un nivel tiene solución, los tiempos de espera son extensos como para utilizar nuestro método para generar niveles complicados de forma *online*. Por ejemplo para el nivel que se ve en la figura 4.7 se necesito 1 hora de ejecución. Cabe

recalcar que este nivel tiene 6 cajas, por lo que niveles de estas dimensiones no podrían ser generados en un contexto *online*. Para la función de *fitness* usamos el *solver* en su versión *Optimize Pushes*. Este es más lento que en su versión *Any solution*, pero así obtenemos una medida más acertada sobre el esfuerzo para solucionar cada nivel.

2. Al hacer uso del *solver* de JSoko, hemos limitado la cantidad máxima de cajas a colocar en un nivel a 10. Hemos revisado Sokobano [11] y el *solver* de JSoko no tiene los mejores tiempos entre los *solvers* mayormente conocidos. Sin embargo, los tiempos de espera no son extensos si los comparamos con el esfuerzo al crearlos de forma manual.

3. Utilizamos entre 10 a 15 generaciones para evolucionar nuestra población inicial, valores mayores obligaban a esperar demasiado tiempo con un bajo factor de exploración como ganancia. Debido al *crossover* que hemos implementado, los estados de nivel posibles eran explorados de buena forma en esta cantidad de iteraciones dando como resultado niveles difíciles de solucionar.

4. Si bien la mayoría de los niveles generados tienen grandes posibilidades de ser Niveles Interesantes, se realizó un filtro manual para escoger los que al parecer del profesor Guía y el tesista parecen mas Interesantes que otros. Por ejemplo, los niveles con una base de nivel (murallas y espacios vacíos) similares, eran discutidos para concluir si colocarlos o no en la lista de tableros de Sokoban que los usuarios probarían.

4.3. Estudio de usuario

A continuación, explicamos como llevamos a cabo el estudio de usuario, los datos que nos fueron útiles recopilar y el software utilizado en la actividad.

4.3.1. Registro de usuarios

Es necesario conocer estadísticas de los usuarios con el propósito de analizarlas y sacar conclusiones respecto del contenido generado. Los datos solicitados a los usuarios los describimos a continuación:

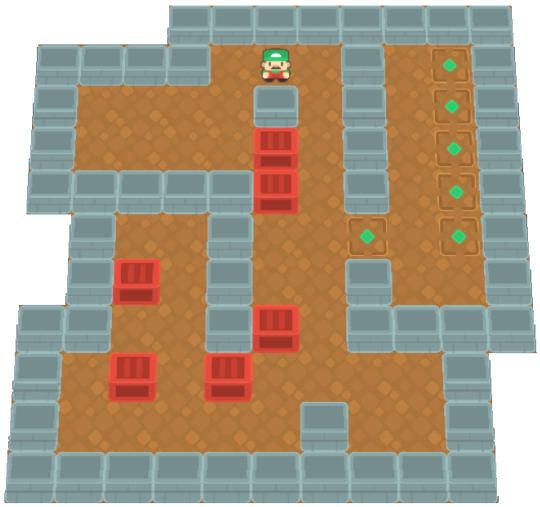


Figura 4.7: Ejemplo de nivel generado en un total de 112 minutos

1. Nombre de usuario: Identificador compuesto del primer nombre y apellido del usuario.
2. Edad: Edad del usuario expresada en años.
3. Nivel educacional: Nivel educacional del usuario.
4. Conjunto de nivel jugado: El identificador del conjunto que aleatoriamente ha sido seleccionado para el usuario.
5. Experiencia en juegos de Puzle: Nivel de experticia que tiene el usuario en juegos de tipo puzzle.
6. Experiencia en juegos de Sokoban: Nivel de experticia que tiene el usuario en juegos de tipo Sokoban.

Hacemos uso de Firebase[2] como servidor para recopilar los datos que los usuarios nos entregan.

4.3.2. Conjunto de niveles

Se crearon dos conjuntos de 10 niveles distintos (ver Apéndice E). El primer nivel que hemos agregado a los dos conjuntos es uno sencillo para que funcione como

[2]Plataforma para el desarrollo de aplicaciones web y aplicaciones móviles lanzada en 2011 y adquirida por Google en 2014

Tutorial, de esta forma las personas que nunca han jugado Sokoban podrán practicar en él.

Son un total de 11 niveles, el nivel inicial que llamaremos nivel Tutorial y 10 niveles adicionales ordenados ascendentemente basados en la cantidad de empujes necesarios para resolverlos. La cantidad de empujes fue entregada por el *solver* JSoko, utilizando su modo optimizar empujes usando AStar (A*).

4.3.3. Recolección de datos por nivel

Cómo se mencionó anteriormente, recabamos algunos datos de la experiencia de juego para su posterior análisis. Los siguientes datos se guardan por cada nivel que el usuario juega:

1. Nivel de dificultad: Valor entre 1 y 5 que representa la dificultad percibida.
2. Nivel de entretención: Valor entre 1 y 5 que representa la entretención percibida.
3. Cantidad de movimientos: Cantidad de movimientos realizados por el usuario.
4. Cantidad de empujes: Cantidad de empujes a cajas realizados por el agente.
5. Cantidad de reinicios: Cantidad de veces que el usuario reinició el nivel.
6. Si el usuario se ha rendido: Variable booleana que nos indica si el usuario se rindió o no antes de pasar al siguiente nivel.
7. Tiempo total de juego: Tiempo total en formato horas/ minutos/ segundos/ milisegundos que le tomó al usuario jugar el nivel.

4.3.4. Experimentos

En esta sección describimos los experimentos realizados para dar respuesta a las sub-hipótesis que hemos planteado. Al usuario se le presentan 10 niveles más el nivel Tutorial para que este responda las actividades. Los detalles son explicados en las siguientes secciones.

Flujo de la experiencia

Los 10 niveles que se presentan al usuario los dividimos en dos subconjuntos. La decisión de dividir los conjuntos fue hecha para crear una experiencia más relajada que a corto plazo nos entregara datos acerca de las 3 métricas. Otra razón es que muchas personas juegan un tiempo reducido, cómo no hay un incentivo detrás de la experiencia de juego, renuncian rápidamente a completar la actividad experimental.

En la figura 4.8 presentamos un diagrama de flujo que muestra las etapas por las que pasan los usuarios al utilizar Sokostigation. Cabe recalcar que los datos son enviados al instante luego de que el usuario termina de jugar un nivel. Gracias a esto no es necesario que terminen la experiencia completamente para que se registren sus respuestas.

Evaluación de métricas por nivel

Al finalizar cada nivel, es presentado al usuario un panel para que evalúe las métricas de dificultad y entretención. Cabe decir que el nivel Tutorial no forma parte del análisis. La imagen 4.9 muestra el panel presentado a los usuarios, contiene escalas Likert con un valor mínimo de 1 y máximo de 5. Una vez es presionado el botón de aceptar se carga el siguiente nivel.

A continuación, detallaremos cómo se evalúa dificultad y entretención.

Dificultad Si el usuario tuvo que reiniciar varias veces o ha demorado mucho en solucionar un nivel en particular, este debería ser puntuado con un valor cercano a 5, en contraste de un nivel trivial, el cual debería ser puntuado con un valor cercano a 1.

Entretención Tal como en la evaluación de la métrica anterior, los usuarios puntúan un nivel entretenido con un valor cercano a 5 y para un nivel aburrido con un valor cercano a 1.

La percepción de dificultad también nos será útil para compararla con nuestra función de *fitness* (cantidad de empujes necesarios para solucionar un nivel).

Evaluación de diversidad

Para evaluar la diversidad de los niveles generados hemos realizado lo siguiente: Después de jugar la primera mitad de niveles, se les presenta a los usuarios un panel

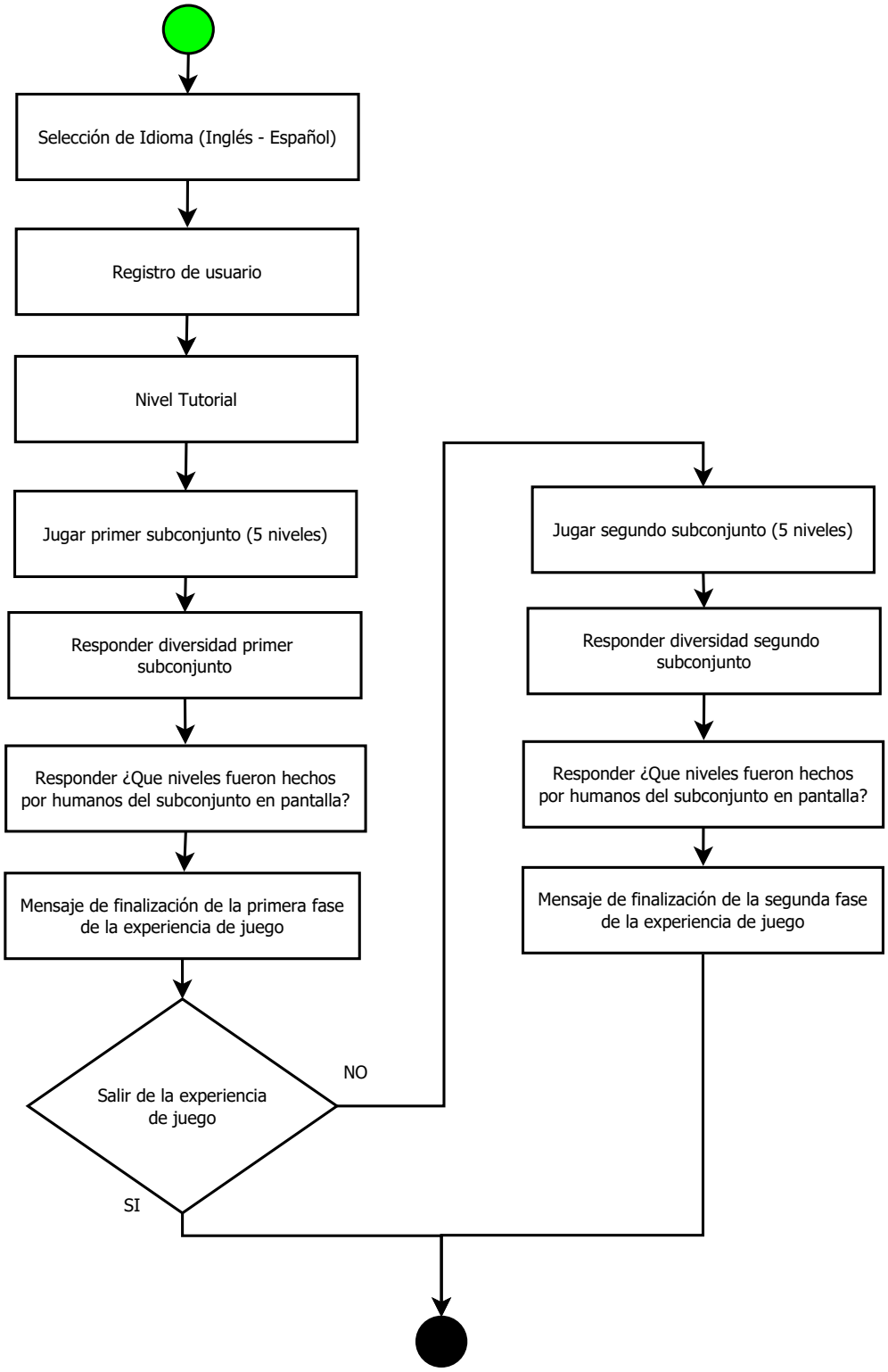


Figura 4.8: Flujo de la experiencia

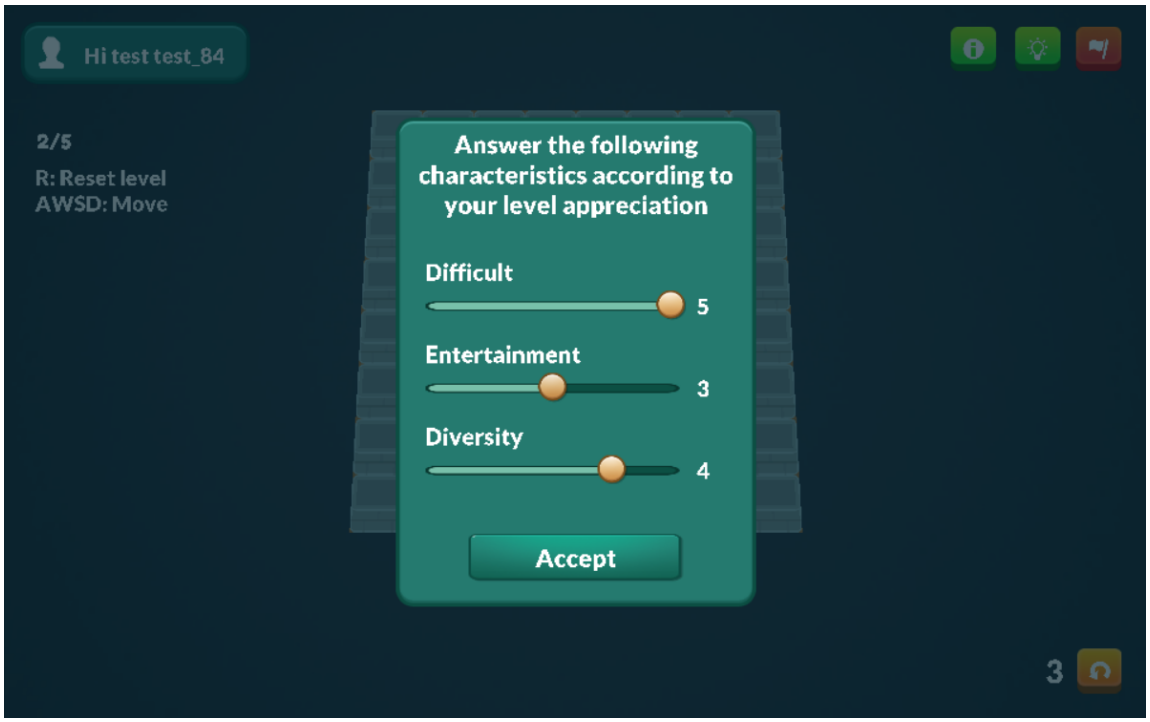


Figura 4.9: Panel de métricas

con el conjunto de niveles jugado. Cómo se aprecia en la imagen 4.10 ellos deben responder a la pregunta: ¿El conjunto de niveles que se muestra es diverso? siendo 5 el valor máximo y 1 el valor mínimo. Una vez contestan, su respuesta es enviada al servidor de Firebase. Posteriormente cuando hayan finalizado la segunda mitad de tableros el proceso se repite, pero ahora con los 5 nuevos niveles jugados.

Evaluación de niveles hechos por humanos versus creados por el algoritmo PCG

Realizamos actividad extra donde los usuarios deben marcar los niveles que creen que fueron creados por humanos de 5 que se les presentan. En la imagen 4.11 se puede apreciar el subconjunto de niveles mostrado en la primera mitad de la experiencia de juego. Al igual que con el panel de diversidad, también pedimos al usuario contestar esta actividad en dos ocasiones.

Para saber si los niveles generados por PCG pueden camuflarse dentro de un conjunto de niveles creados por humanos, mezclamos los niveles generados por PCG con

Figura 4.10: Panel para que los usuarios evalúen diversidad

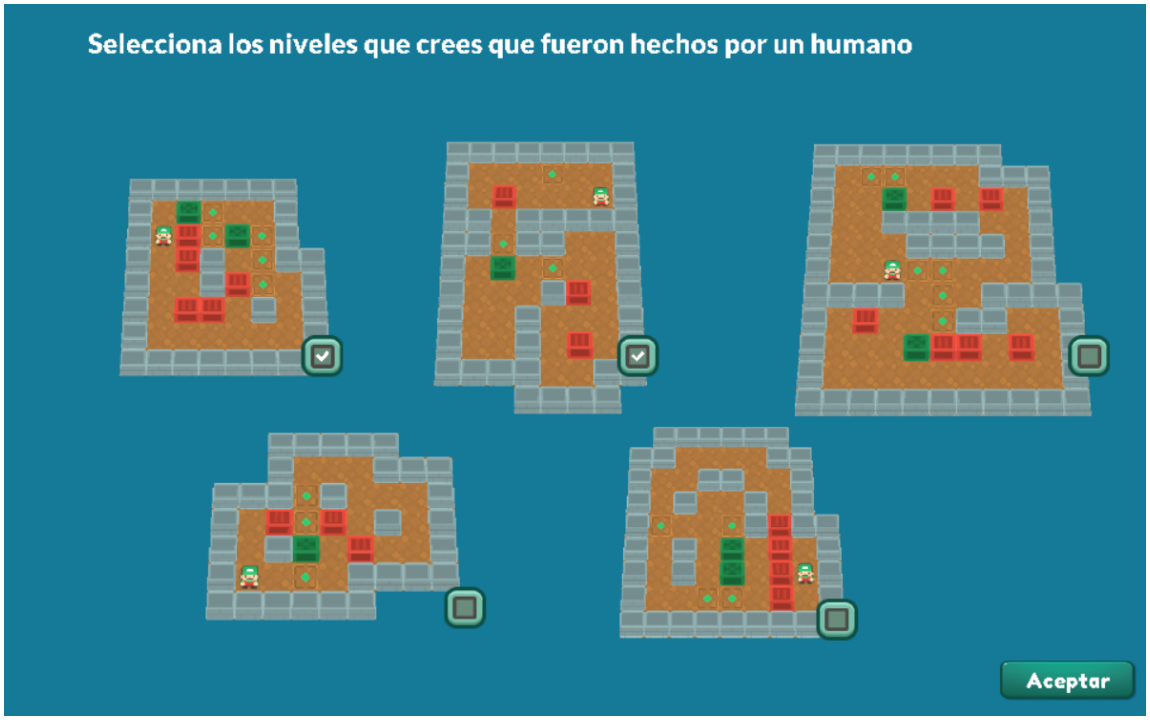


Figura 4.11: Panel donde los usuarios eligen los niveles creados por humanos

dos niveles hechos por humanos[3]. Así hicimos también con el segundo subconjunto en la parte final del videojuego.

4.4. Metodología de desarrollo

Llevamos a cabo una combinación del proceso iterativo descrito en Scrum junto al control y planificación de tareas que plantea Kanban [69]. Cómo se puede apreciar en la imagen 4.12, nuestro proceso iterativo consta de 4 etapas:

1. Planificación: Se extraen las tareas a realizar desde la plataforma elegida para llevar a cabo Kanban. Ocurre una actualización del estado de estas tareas.
2. Desarrollo: Se llevan a cabo las tareas elegidas en la fase anterior.
3. Prueba: Se comprueba que las tareas hayan sido realizadas según lo planificado.
4. Retrospectiva: Se evalúa el progreso, resultados y se sacan conclusiones e ideas respecto del avance para el trabajo futuro.

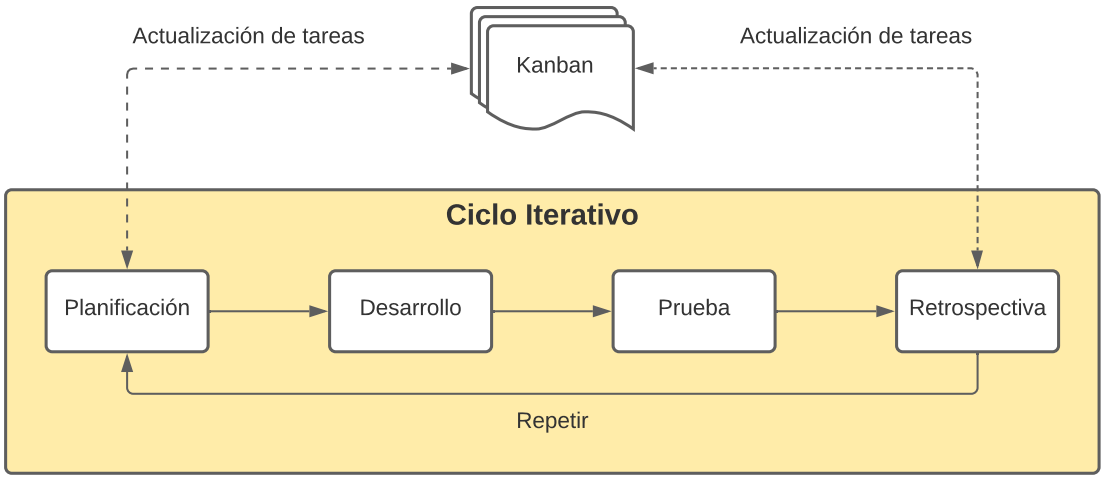


Figura 4.12: Ciclo iterativo y Kanban

En cuanto a las reuniones, las habrá semanalmente los martes a las 16:00 horas por medio de Discord. En ellas se hará revisión junto al profesor guía para discutir los resultados y problemas que van apareciendo.

[3]Extraídos del conjunto de niveles hechos por humanos Yoshio Murase. `https://www.sokobanonline.com/play/web-archive/yoshio-murase/hand-made`

4.4.1. Carta Gantt

El periodo para el cual están planificadas las siguientes tareas es 4 meses. Tal como se puede apreciar en 4.13 existen 3 tareas generales. Las describimos a continuación:

1. Algoritmo genético: Esta etapa del proyecto se refiere a la creación del algoritmo genético que dará lugar a los niveles de Sokoban. Comenzamos por hacer la configuración del proyecto importando la librería Jenes para luego continuar con las fases de creación de la población inicial, selección, *crossover* y mutación.

2. Escritura: Se realizan las mejoras pertinentes al escrito, se actualiza con la nueva información encontrada la metodología, prueba de usuario además de pulir otras secciones.

3. Prueba de usuario: Se usa el prototipo de prueba de usuario Sokostigation para reunir información acerca de las sesiones de juego. Los usuarios califican los niveles respecto a su dificultad, diversidad y entretención. Es realizado una actividad exploratoria donde los usuarios evalúan niveles hechos por humanos frente a los generados por el método PCG expuesto.

En 4.13 se puede encontrar mayor información sobre las subtareas con sus respectivos plazos.

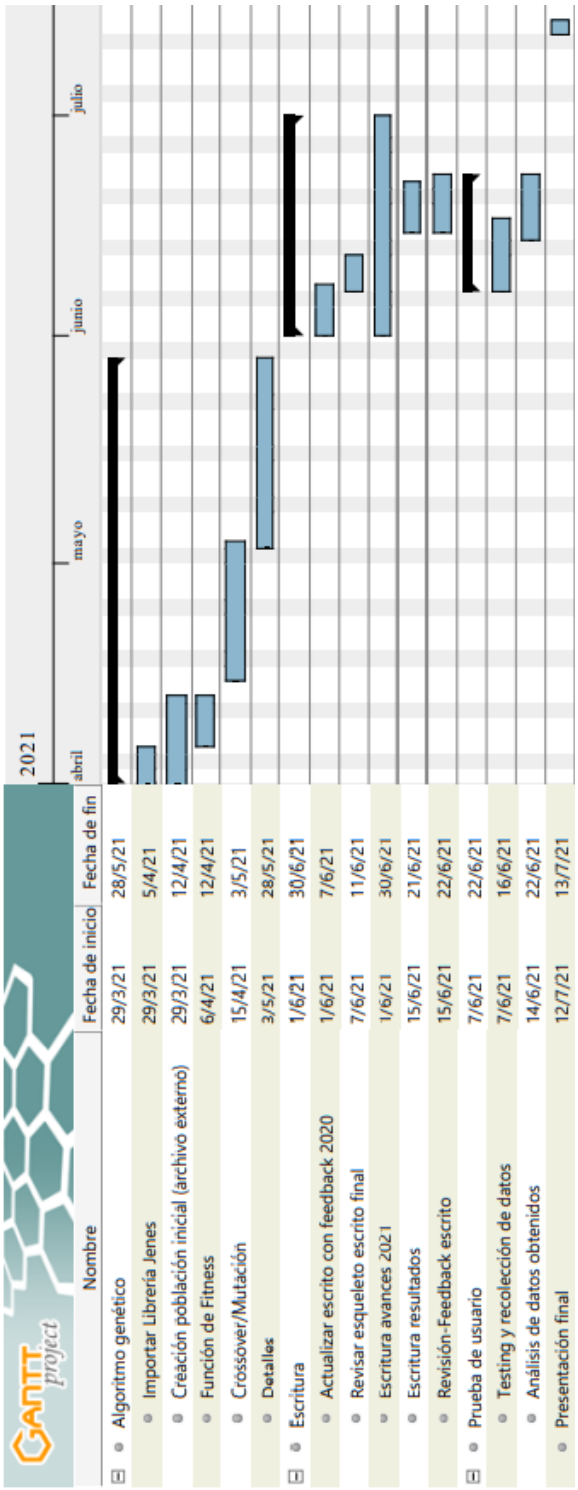


Figura 4.13: Planificación carta Gantt meses abril hasta julio

5. Resultados

En esta sección mostramos los resultados que se obtuvieron con los 4 días de *testing* por medio de la aplicación web Sokostigation creada para efectos de este estudio. Analizamos las métricas obtenidas y los experimentos realizados.

5.1. Métricas de entretención, dificultad y diversidad

Dentro de los parámetros a evaluar se encuentran la entretención, dificultad y la diversidad que los usuarios perciben al jugar los niveles generados. Generamos 20 niveles los que hemos dividido en 2 conjuntos de 10 cada uno, los que fueron jugados parcialmente por 77 jugadores (ver anexo A).

5.1.1. Experimento evaluación de métricas por nivel

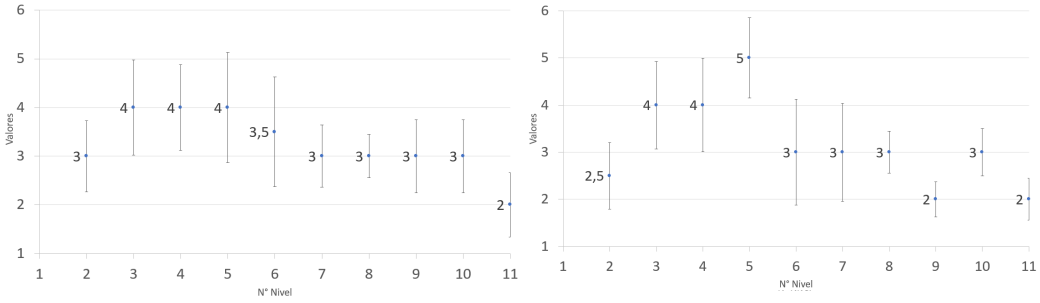
En el cuadro 5.1 resumimos las datos obtenidos de los usuarios frente a los niveles jugados. Hemos sacado las medianas de las evaluaciones las que van acompañadas de la desviación media absoluta entre paréntesis.

En las figuras 5.1a, 5.1b, 5.2a, 5.2b se puede ver la entretención y dificultad percibida junto a sus desviaciones absolutas medias para apreciar la dispersión de los datos. De los datos obtenidos podemos discutir lo siguiente:

Entretención Para el conjunto 1 podemos afirmar que el 90 % de los niveles obtuvo una mediana de 3 o más. Para el conjunto 2 el 100 % de los niveles obtuvo una mediana igual o superior a 3, con un 60 % de los niveles con un valor de 4. Los resultados dan cuenta de una tendencia positiva frente a esta métrica, lo que nos permite confirmar la sub-hipótesis "Es posible generar tableros de Sokoban entretenidos por medio de algoritmos evolutivos".

Nivel	Conjunto 1		Conjunto 2	
	Entretención	**Dificultad**	**Entretención**	**Dificultad**
2	3 *(0,73)*	2.5 *(0,7)*	3 *(0,85)*	4 *(0,79)*
3	4 *(0,97)*	4 *(0,92)*	4 *(0,80)*	3 *(0,79)*
4	4 *(0,87)*	4 *(0,99)*	4 *(0,80)*	3 *(0,92)*
5	4 *(1,13)*	5 *(0,85)*	4 *(0,84)*	3 *(0,68)*
6	3.5 *(1,12)*	3 *(1,12)*	4 *(0,80)*	3 *(0,76)*
7	3 *(0,64)*	3 *(1,04)*	4 *(0,65)*	5 *(0,99)*
8	3 *(0,44)*	3 *(0,44)*	4 *(0,83)*	4.5 *(0,83)*
9	3 *(0,75)*	2 *(0,37)*	3 *(0,44)*	3 *(0,66)*
10	3 *(0,75)*	3 *(0,5)*	3 *(0,44)*	2 *(0,44)*
11	2 *(0,66)*	2 *(0,44)*	3.5 *(0,5)*	3 *(1)*

Cuadro 5.1: Cuadro resumen con las medianas de las métricas obtenidas a partir de las sesiones de juego de los usuarios en Sokostigation. Los valores entre paréntesis son las desviaciones medias absolutas (MAD) para cada conjunto de datos



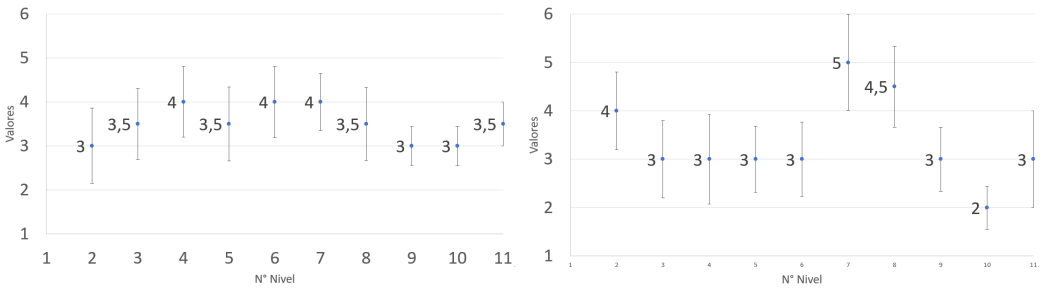
(a) Entretención percibida

(b) Dificultad percibida

Figura 5.1: Gráficas de dificultad y entretención percibida del conjunto de niveles 1. En ella se pueden apreciar las medianas obtenidas para cada nivel junto a las desviaciones absolutas medias (MAD)

Dentro del análisis que los datos nos permiten realizar, evaluamos la correlación de Spearman entre la entretención y la dificultad de todos los niveles (conjunto 1 y 2). El valor-p para la entretención con la dificultad fue un valor de 0,001 junto a un valor R de 0,420. Es decir que existe una relación estadísticamente significativa y positiva media, por ende, a mayor dificultad percibida también es mayor la sensación de entretención que el usuario percibe.

Dificultad Del conjunto 1 en el cuadro 5.1 podemos decir que el 80 % de los niveles logran una dificultad de 3 o mayor. Interesante es la mediana alcanzada del



(a) Entretención percibida (b) Dificultad percibida

Figura 5.2: Gráficas de dificultad y entretención percibida del conjunto de niveles 2. En ella se pueden apreciar las medianas obtenidas para cada nivel junto a las desviaciones absolutas medias (MAD)

quinto nivel, el cual tiene un valor de 5. Es curioso que este nivel haya logrado la dificultad máxima a pesar de aparecer en los primeros puestos del conjunto. Este hecho apela a que el uso de los empujes para clasificar niveles por dificultad no es la mejor idea en todos los casos. Por ejemplo, el undécimo nivel tenía la mayor cantidad de empujes necesarios para ser solucionado, pero este recibió un valor de 2 en dificultad. Para el conjunto número 2 existe una situación similar donde el 90 % de los niveles tiene un valor de 3 o más. Para este conjunto el séptimo nivel alcanza la máxima de 5 y el octavo un valor de 4,5. El nivel 10 a pesar de ser uno de los últimos del conjunto, tiene un valor de 2 en dificultad. Ocurre lo mismo que describíamos con el nivel 11 del conjunto 1.

El conjunto 1 y 2 tienen un promedio de 3.15 y 3.35 (promedio obtenido a partir de las medianas de dificultad) respectivamente dejando al conjunto 2 como el más difícil de jugar. La figura 5.3 muestra la experiencia en Puzles y Sokoban que han marcado los usuarios. Podemos observar que los usuarios con una experiencia en Puzles y Sokoban igual o mejor a la categoría competente es un 55 % y un 81 % respectivamente. Por lo que creemos tener una muestra representativa de usuarios calificados para evaluar dificultad de los niveles generados. Los datos resultantes nos permiten concluir que el generador es capaz de crear niveles complicados de resolver y que a su vez sean entretenidos de jugar.

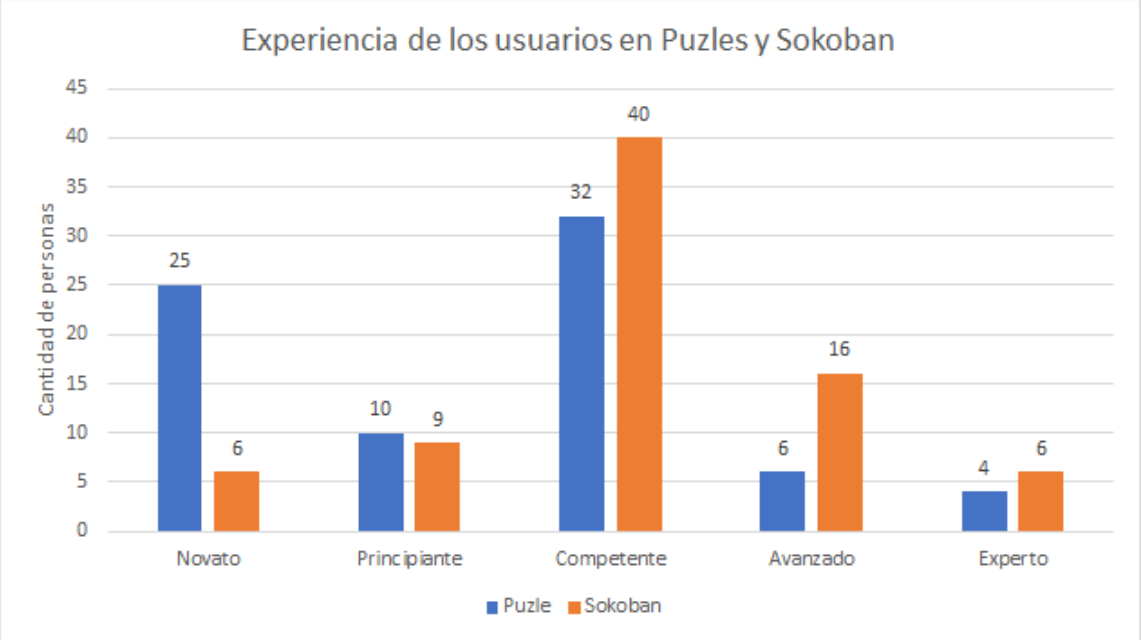


Figura 5.3: Experiencia de los usuarios en Puzles y Sokoban

5.1.2. Experimento evaluación de diversidad

A continuación se presentan los resultados obtenidos del experimento de diversidad presentado en la sección 4.3. En este experimento mostramos al usuario dos subconjuntos de niveles (ver apéndice E) según el conjunto de niveles que se le haya asignado en la fase de registro de usuario. Ellos deben marcar cual es el nivel de diversidad que perciben en el subconjunto en pantalla. En la figura 5.4 se pueden ver los valores obtenidos para la diversidad en cada subconjunto. Cabe decir que si bien los usuarios totales que jugaron parcialmente los niveles son 77, los que efectivamente llegaron hasta las etapas de medición de diversidad son 52.

De la figura 5.4 podemos destacar:

1. El subconjunto 1 del conjunto 1 el que presenta mayor diversidad con una mediana de 4,5.

2. El subconjunto menos diverso de todos es el subconjunto 2 del conjunto 1 con una mediana de 3.

3. El subconjunto 2 del conjunto 1 y 2 obtuvieron poca participación. Aun así, registran valores iguales o mayores a 3.

Debido a estos valores podemos afirmar que es posible crear contenido diverso con el generador de niveles para el puzle Sokoban.

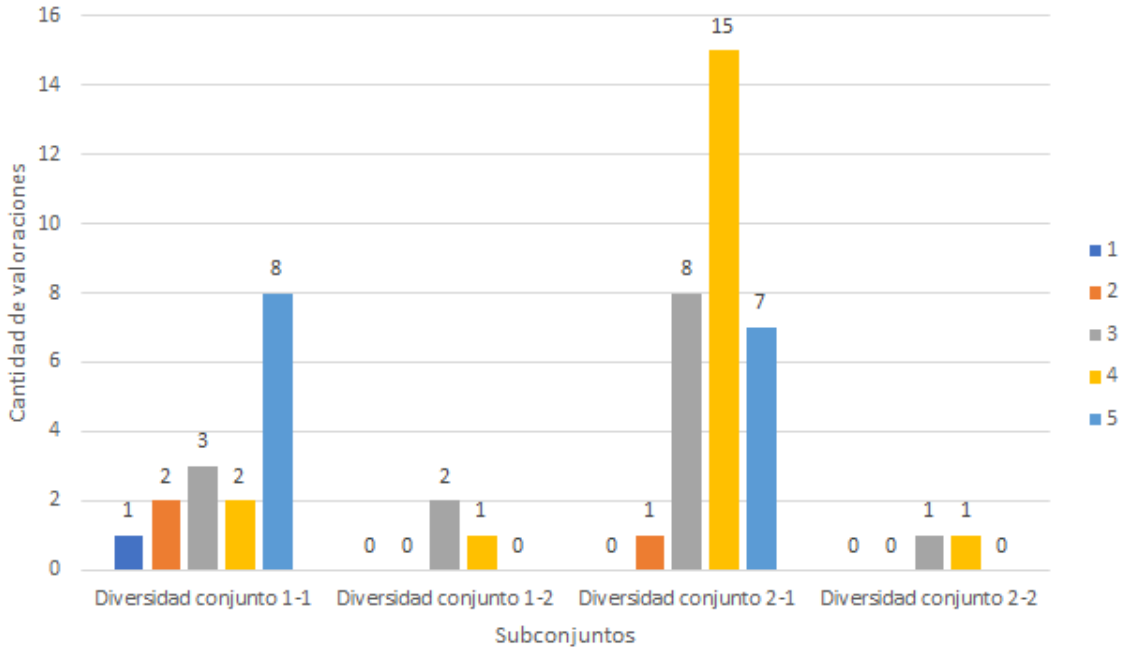


Figura 5.4: Diversidad por cada subconjunto mostrado al usuario

5.2. Correlación de dificultad

A continuación, realizamos algunos experimentos para averiguar cómo se correlacionan los datos de los usuarios y los niveles. Tratamos de encontrar nuevas formas de medir dificultad aplicando el método de Spearman entre distintas variables analizando la correlación con la longitud de la lista cerrada del algoritmo A* implementado en el *solver* de JSoko.

Para correlacionar los datos, hemos optado por tomar el total de ellos de los niveles superados (solo usuarios que no se hayan rendido) para realizar Spearman en el software Jamovi [70]. En el cuadro 5.2 y 5.3 hemos anotado los empujes, movimientos, mediana de la dificultad percibida, valores finales del largo de lista cerrada y abierta del algoritmo A* (variable poco fiable ya que esta cambia durante ejecución) para ver si se correlacionan estas variables en alguna medida.

En el cuadro 5.4 se encuentran los distintos valores de R de cada experimento para encontrar nuevas relaciones. Cada valor en negrita obtuvo un valor-p menor o

Nivel	JSoko Empujes	JSoko Movimientos	Dificultad percibida	Largo Lista cerrada A*	Largo lista abierta A*
2	16	42	2.5	16	0
3	26	93	4	730	275
4	31	95	4	4831	2097
5	34	80	5	714	199
6	39	98	3	95	56
7	45	114	3	260	167
8	48	164	3	812	450
9	49	140	2	2403	1486
10	54	118	3	329	132
11	77	247	2	114	16

Cuadro 5.2: Resumen de los datos obtenidos para el conjunto de niveles 1. JSoko empujes y movimientos fueron otorgados por el *solver* de JSoko mientras que la dificultad percibida corresponde a la mediana de los datos registrados por nivel.

Nivel	JSoko Empujes	JSoko Movimientos	Dificultad percibida	Largo lista cerrada A*	Largo lista abierta A*
2	21	129	4	114	57
3	26	72	3	209	23
4	27	179	3	5016	3472
5	28	75	3	46	14
6	30	77	3	86	28
7	33	114	5	529	105
8	34	122	4.5	133	23
9	39	132	3	1650	424
10	40	139	2	308	210
11	44	139	3	438	71

Cuadro 5.3: Resumen de los datos obtenidos para el conjunto de niveles 2. JSoko empujes y movimientos fueron otorgados por su *solver*, mientras que la dificultad percibida corresponde a la mediana de los datos registrados por nivel.

igual a 0.05 (valores significantes). Para las siguientes conclusiones hemos tomado en cuenta Spearman ya que expone una mejor estabilidad que Pearson [71]. A partir de los datos en el cuadro 5.4 podemos concluir lo siguiente:

1. Casi todas las correlaciones tienen un R positivo (Excepto dificultad percibida - JSoko Optimizer Pushes), lo que indica que el cambio de la variable A puede provocar un cambio en la misma dirección a la variable B mas no en la mis-

	Largo lista cerrada A*		Largo lista abierta A*		JSoko Optimizer Moves		JSoko optimizer Pushes	
	Spearman	**Pearson**	**Spearman**	**Pearson**	**Spearman**	**Pearson**	**Spearman**	**Pearson**
Movimientos de los usuarios	**0.762**	**0.380**	**0.773**	**0.403**	**0.856**	**0.78**	**0.34**	**0.501**
Empujes de los usuarios	**0.518**	**0.174**	**0.547**	**0.160**	**0.591**	**0.749**	**0.867**	**0.935**
Porcentaje de movimientos por sobre el que dio Jsoko	**0.433**	0.089	**0.418**	0.107	**0.408**	**0.219**	**0.216**	0.097
Tiempo en resolver de los usuarios	**0.4**	**0.13**	**0.42**	0.095	**0.437**	**0.225**	**0.34**	**0.173**
Dificultad percibida	**0.286**	0.13	**0.284**	**0.224**	**0.187**	0.107	0.089	-0.003

Cuadro 5.4: Resumen de los valores de R de Spearman y Pearson (valores en negrita tienen un valor-p menor o igual a 0.05) para distintas métricas de dificultad. Fueron utilizados los datos de los usuarios que completaron niveles sin la opción de rendirse

ma magnitud. Hay que recordar que aún existe posibilidad de que ocurra la hipótesis nula (las variables no se correlacionan), pero este valor probabilístico es menor o igual a 0.05 por lo que es poco probable que suceda.

2. Hemos encontrado una correlación muy fuerte y positiva entre los movimientos que hacen los usuarios y el largo de la lista cerrada del algoritmo A* (nodos visitados). Esta última variable puede ser de ayuda para calificar niveles guiando la búsqueda hacia instancias complicadas de resolver.

3. La métrica que hemos utilizado en nuestra función de *fitness* (los empujes dados por JSoko en su versión optimizar empujes) de la fase de evolución, no correlaciona bien con la dificultad percibida por los usuarios. Este valor de 0,089 correlaciona positivamente, pero de forma débil. Por lo que utilizar otra métrica puede ser mejor idea para guiar la búsqueda hacia niveles más complicados.

4. La variable que mejor correlaciona con la dificultad percibida es el largo de la lista cerrada. Por lo que es un buen candidato para usar en la función de *fitness* de la fase evolutiva.

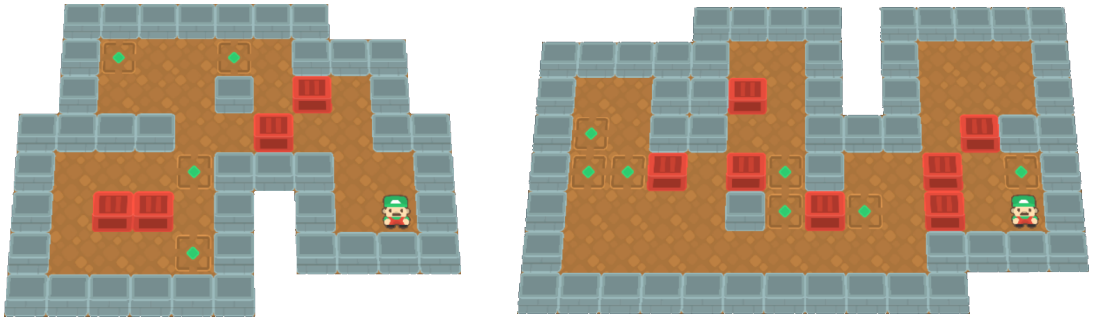
5.3. Resultados experimento niveles hechos por humanos versus PCG

Cómo se mencionó en la sección 4.3 hemos realizado esta actividad con la intención de saber si los niveles generador por PCG se podrían camuflar dentro de un conjunto de niveles en los que también hay otros creados por humanos.

En la figura 5.6 se pueden apreciar las respuestas de los usuarios para los niveles creados por nuestro generador. Luego en la figura 5.7 se exponen las capturas a la pantalla donde es efectuada la actividad. Las observaciones las exponemos a continuación:

1. Los niveles mayormente confundidos con niveles creados por humanos son los que se ven en la figura 5.5, ambos con un 73 % del total de 15 votos.

2. Las personas no pudieron distinguir los niveles creados por nuestro método PCG, ni tampoco los niveles creados por humanos. Por lo tanto, los niveles creados con PCG si se pueden camuflar en un conjunto de niveles de Sokoban creados por humanos.



(a) Nivel 2 Subconjunto 1 Conjunto 1 (b) Nivel 3 Subconjunto 1 Conjunto 1

Figura 5.5: Niveles creados por nuestro método PCG confundidos con niveles creados por un humano

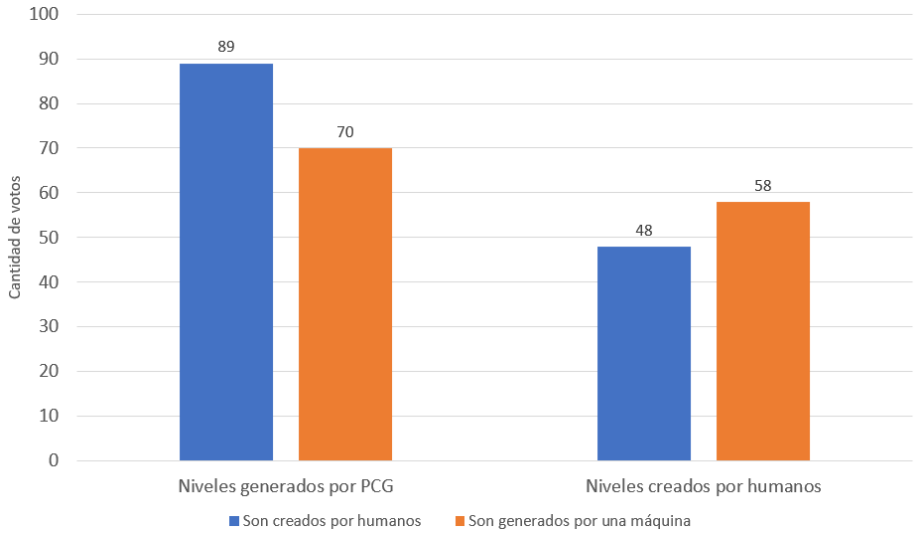
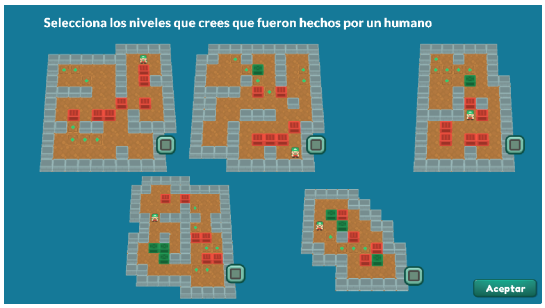
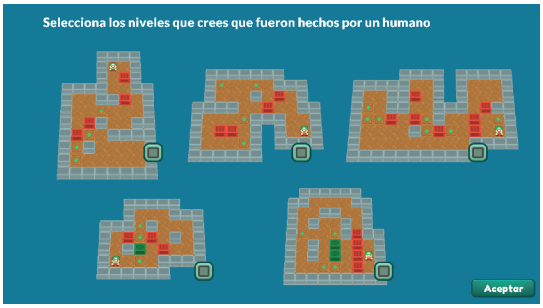


Figura 5.6: Resumen datos obtenidos experimento humanos versus PCG

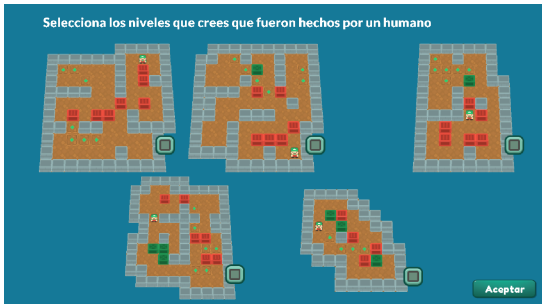
(a) Subconjunto 1 Conjunto 1



(b) Subconjunto 1 Conjunto 2



(c) Subconjunto 2 Conjunto 1



(d) Subconjunto 2 Conjunto 2

Figura 5.7: Actividad niveles hechos por humanos versus el método PCG. La primera y segunda fila de niveles son generados con PCG y hechos por humanos respectivamente

6. Conclusiones

Al comienzo de este manuscrito nos propusimos construir un generador capaz de crear niveles que fueran entretenidos, difíciles y diversos. Luego fueron evaluados por medio de un estudio de usuario con ayuda de una aplicación web donde los usuarios jugaban y respondían algunas actividades. Los datos eran enviados a un servidor web para su posterior análisis y con ello comprobar si las sub-hipótesis se cumplen. Estas son:

1. Es posible generar tableros de Sokoban entretenidos por medio de algoritmos evolutivos.
2. Es posible generar tableros diversos por medio de algoritmos evolutivos.
3. Es posible generar tableros difíciles por medio de algoritmos evolutivos.

6.1. Conclusiones respecto de las sub-hipótesis

Después de analizar las estadísticas obtenidas en la prueba de usuario, donde 77 personas evaluaron estas características por medio de una escala Likert que va desde 1 a 5, siendo 1 el peor valor y 5 el mejor.

Entretención Para el conjunto 1 obtuvimos que el 90 % de los niveles tiene una media de 3 o más. Mientras que para el conjunto 2 todos los niveles mantienen una mediana igual o superior a 3. Por lo que podemos concluir que si es posible generar tableros de Sokoban entretenidos por medio de algoritmos evolutivos.

Dificultad El 80 % de los niveles jugados del conjunto 1 tienen una dificultad media percibida de 3 o más. Para el conjunto 2 la dificultad media percibida fue de 3

o más en el 90 % de los niveles jugados. Por lo que se concluye que es posible generar niveles difíciles por medio de algoritmos evolutivos (lo suficiente para que un usuario con experiencia moderada lo encuentre complicado y divertido).

Diversidad Pocas personas evaluaron los subconjuntos 2 de los conjuntos (menor a 4) por lo que no podemos concluir efectivamente sobre estos datos, aun así, se registraron valores de 3 y 4 para ellos en su diversidad. El subconjunto 1 del conjunto 1 tiene un valor medio en cuanto a diversidad percibida de 4,5. Mientras que el subconjunto 1 del conjunto 2 tiene un valor medio en cuanto a diversidad percibida de 3, siendo el peor subconjunto evaluado. Estos datos nos permiten concluir que si se es posible crear niveles diversos por medio de algoritmos evolutivos.

Los datos sin procesar pueden ser encontrados en el siguiente enlace: datos pruebas de usuario.

¿Se pueden confundir los niveles generados por nuestro método PCG cuando son mezclados con niveles creados por humanos?

Del experimento realizado, los resultados fueron que si hubo confusión en los usuarios respecto de cuales eran niveles creados por nuestro método PCG y cuales eran creados por humanos. Por ejemplo, el caso más interesante son los resultados obtenidos para los niveles 2 y 3 (ver figura 5.5) del subconjunto 1 conjunto 1, ya que un 73 % de las 15 personas que contestaron, creyeron que eran niveles creados por humanos. Puede encontrar más ejemplos de niveles creados con el generador en el apéndice D.

6.2. Ejemplos de otros enfoques para la función de *fitness*

La búsqueda de niveles complicados de nuestro algoritmo evolutivo fue guiada por la cantidad de empujes necesarios para dar solución a los niveles. A partir del análisis estadístico (ver sección 5) nos dimos cuenta que una mejor variable para guiar esta búsqueda puede haber sido el largo de la lista cerrada en el algoritmo A*, ya que correlaciona casi 0,2 puntos mejor que los empujes. Los datos obtenidos para el largo de la lista cerrada anteriormente mencionada son un R de 0,286 (Spearman) y un valor-p menor a 0,01 con la dificultad percibida por los usuarios. En cambio los

valores para la cantidad de empujes necesarios con la dificultad percibida es un R de 0,089 (Spearman) y un valor-p de 0,906. A continuación, planteamos otras variables que pueden guiar esta búsqueda por niveles difíciles en un proceso evolutivo:

1. Box Changes: Cambios de dirección de las cajas, esta métrica puede ser difícil de implementar pero es muy útil para dejar fuera el problema donde las cajas son empujadas en línea recta (métricas basadas en movimientos y empujes del agente) [15].

2. Problem decomposition based [10]: Proponen descomponer el problema (nivel a solucionar) en distintos subproblemas, donde cada uno de estos es puede ser una caja y su meta o un grupo de cajas y metas. Luego se resuelven los subgrupos para obtener una medida de dificultad basado en por ejemplo sus *box-lines*.

3. Closed List Length [72]: Cómo vimos en los primeros párrafos hay una correlación positiva media entre la dificultad percibida y esta variable. El largo de esta lista nos proporciona la cantidad de nodos visitados por A*, esta puede ser una buena medida para guiar la búsqueda hacia niveles complicados.

6.3. Trabajo futuro y consideraciones

Nuestro método puede trabajar con 6 a 8 cajas en un rango de tiempo menor o igual a 4 horas, en contraste del método de Taylor [15] el que puede generar tableros hasta con un máximo de 6 cajas en este mismo rango de tiempo. Esto puede ser por distintos factores como la evolución del hardware, la fuerza bruta que ellos implementan para posicionar cajas y metas o las limitaciones de tiempo en el proceso de generación de los estudios. Hemos limitado nuestro proceso PCG a un máximo de 10 cajas para evitar tiempos largos de espera, ya que utilizamos continuamente el *solver* de JSoko y por ejemplo, buscar la solución a un tablero de 15 x 15 *tiles* con 8 o más cajas puede llegar a tardar horas de procesamiento. La velocidad de generación no estuvo dentro de nuestras sub-hipótesis consideramos interesante medir este tiempo, restando de la evolución las funciones para *debug* implementadas, realizar las optimizaciones posibles para lograr el menor tiempo posible y ocupar un *solver* mas rápido que el de JSoko.

Si bien se obtuvieron buenos valores de diversidad y dificultad desde los subconjuntos, se podría obtener una mejor búsqueda optimizando estos parámetros entre los niveles generados usando un algoritmo perteneciente a la familia de algoritmos *Quality Diversity*, ya que estos dan como resultado un grupo de soluciones de alta calidad en vez de solo una. Creemos que hay nuevas líneas de investigación en esta área ya que hay un gran conjunto de estos algoritmos [73].

Recientemente también se ha utilizado *Wave Function Collapse* [38] como método PCG. Creemos que es un algoritmo que puede dar muy buenos resultados y este puede ser utilizado para crear la base de los tableros de Sokoban en nuestra etapa de Inicialización. Sería interesante ver las diferencias que se producen en el vecindario inicial generado con este algoritmo y nuestro actual estudio, puede ocurrir que haya una convergencia a individuos más diversos al final de la fase de Evolución.

6.4. Conclusión final

Se creó un nuevo método que unifica varios de los enfoques vistos en la sección 2 para la creación de tableros de Sokoban entretenidos, diversos y difíciles de resolver. Logrando en algunos casos niveles de hasta 8 cajas. También realizamos un estudio de usuario que nos permitió afirmar nuestras sub-hipótesis, comprobar que nuestros niveles si se pueden camuflar en medio de un grupo de niveles hechos a mano y encontrar nuevas variables candidatas para que formen parte de la función de *fitness* para eventuales métodos basados en evolución. El método realizado en este trabajo puede ser extrapolado a distintos juegos *sokoban-like* como por ejemplo Zen Puzzle Garden [74] y Stephen's Sausage Roll [75], logrando ahorrar recursos de tiempo y dinero.

Bibliografía

[1] Barbara De Kegel and Mads Haahr. Procedural puzzle generation: a survey. *IEEE Transactions on Games*, 12(1):21–40, 2019.

[2] Gillian Smith. An analog history of procedural content generation. In *Foundations of Digital Games (FDG)*, 2015.

[3] Gwyneth A Bradbury, Il Choi, Cristina Amati, Kenny Mitchell, and Tim Weyrich. Frequency-based controls for terrain editing. In *Proceedings of the 11th European Conference on Visual Media Production*, pages 1–10, 2014.

[4] Robert Dale. Gpt-3: What's it good for? *Natural Language Engineering*, 27(1):113–118, 2021.

[5] Danial Hooshyar, Moslem Yousefi, Minhong Wang, and Heuiseok Lim. A data-driven procedural-content-generation approach for educational games. *Journal of Computer Assisted Learning*, 34(6):731–739, 2018.

[6] Noor Shaker, Julian Togelius, and Mark J Nelson. *Procedural content generation in games*. Springer, 2016.

[7] Games-Stats.com. Baba Is You – Stats on Steam. `https://games-stats.com/steam/game/baba-is-you`, 2020. [Online; accessed 24-november-2020].

[8] Megan Charity, Ahmed Khalifa, and Julian Togelius. Baba is y'all: Collaborative mixed-initiative level design. In *2020 IEEE Conference on Games (CoG)*, pages 542–549. IEEE Computer Society, 2020.

[9] Nathan Sturtevant, Nicolas Decroocq, Aaron Tripodi, and Matthew Guzdial. The unexpected consequence of incremental design changes. In *Proceedings of*

the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), volume 16, pages 130–136, 2020.

[10] Petr Jarušek and Radek Pelánek. Difficulty rating of Sokoban puzzle. In *Proc. of the Fifth Starting AI Researchers' Symposium (STAIRS 2010)*, pages 140–150, 2010.

[11] Sokobano Community. Sokoban Levels. `http://sokobano.de`, 2020. [Online; accessed 30-may-2021].

[12] Joseph Culberson. Sokoban is pspace-complete. 1997.

[13] Dorit Dor and Uri Zwick. Sokoban and other motion planning problems. *Computational Geometry*, 13(4):215–228, 1999.

[14] Erik D Demaine, Martin L Demaine, Michael Hoffmann, and Joseph O'Rourke. Pushing blocks is hard. *Computational Geometry*, 26(1):21–36, 2003.

[15] Joshua Taylor and Ian Parberry. Procedural generation of Sokoban levels. In *Proceedings of the International North American Conference on Intelligent Games and Simulation*, pages 5–12, 2011.

[16] Joshua Taylor, Ian Parberry, and Thomas D Parsons. Comparing player attention on procedurally generated vs. hand crafted sokoban levels with an auditory stroop test. In *Foundations of Digital Games (FDG)*, 2015.

[17] Bilal Kartal, Nick Sohre, and Stephen J Guy. Data driven sokoban puzzle generation with monte carlo tree search. In *InProceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2016.

[18] William L Raffe, Fabio Zambetta, Xiaodong Li, and Kenneth O Stanley. Integrated approach to personalized procedural map generation using evolutionary algorithms. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(2):139–155, 2014.

[19] Dâmaris S Bento, André G Pereira, and Levi HS Lelis. Procedural generation of initial states of Sokoban. *arXiv preprint arXiv:1907.02548*, 2019.

[20] Muhammad Suleman, Farrukh Syed, Tahir Syed, Saqib Arfeen, Sadaf Iqbal, and Behroz Mirza. Generation of sokoban stages using recurrent neural networks. *International Journal of Advanced Computer Science and Applications*, 8, 03 2017.

[21] Hwanhee Kim, Teasung Hahn, Sookyun Kim, and Shinjin Kang. Graph based wave function collapse algorithm for procedural content generation in games. *IEICE Transactions on Information and Systems*, 103(8):1901–1910, 2020.

[22] Graham Kendall, Andrew Parkes, and Kristian Spoerer. A survey of NP-complete puzzles. *ICGA Journal*, 31(1):13–34, 2008.

[23] Joshua Taylor. *The procedural generation of interesting Sokoban levels*. University of North Texas, 2015.

[24] Yoshio Murase, Hitoshi Matsubara, and Yuzuru Hiraga. Automatic making of Sokoban problems. In *Pacific Rim International Conference on Artificial Intelligence*, pages 592–600. Springer, 1996.

[25] Joseph C Culberson and Jonathan Schaeffer. Searching with pattern databases. In *Conference of the Canadian Society for Computational Studies of Intelligence*, pages 402–416. Springer, 1996.

[26] Enrique Naredo, Leonardo Trujillo, Pierrick Legrand, Sara Silva, and Luis Muñoz. Evolving genetic programming classifiers with Novelty Search. *Information Sciences*, 369:347–367, 2016.

[27] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. Sentient sketchbook: computer-assisted game level authoring. 2013.

[28] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. Constrained Novelty Search: A study on game content generation. *Evolutionary computation*, 23(1):101–129, 2015.

[29] Ahmed Khalifa, Michael Cerny Green, Gabriella Barros, and Julian Togelius. Intentional computational level design. In *Proceedings of The Genetic and Evolutionary Computation Conference*, pages 796–803, 2019.

[30] Bilal Kartal, Nick Sohre, and Stephen Guy. Generating Sokoban puzzle game levels with Monte-Carlo Tree Search. In *The IJCAI-16 Workshop on General Game Playing*, volume 47, 2016.

[31] Thorbjørn S Nielsen, Gabriella AB Barros, Julian Togelius, and Mark J Nelson. Towards generating arcade game rules with vgdl. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 185–192. IEEE, 2015.

[32] Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. Pcgrl: Procedural content generation via reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, volume 16, pages 95–101, 2020.

[33] Yaron Shoham and Jonathan Schaeffer. The fess algorithm: A feature based approach to single-agent search. In *2020 IEEE Conference on Games (CoG)*, pages 96–103. IEEE, 2020.

[34] Heiner Marxen and Matthias Meger. Sokoban Solver Statistics - Level Set Summary - XSokoban. `http://sokobano.de`, 2020. [Online; accessed 29-may-2021].

[35] Andreas Hald, Jens Struckmann Hansen, Jeppe Kristensen, and Paolo Burelli. Procedural content generation of puzzle games using conditional generative adversarial networks. In *International Conference on the Foundations of Digital Games (FDG)*, pages 1–9, 2020.

[36] Edoardo Giacomello, Pier Luca Lanzi, and Daniele Loiacono. Doom level generation using generative adversarial networks. In *2018 IEEE Games, Entertainment, Media Conference (GEM)*, pages 316–323. IEEE, 2018.

[37] Vanessa Volz, Jacob Schrum, Jialin Liu, Simon M Lucas, Adam Smith, and Sebastian Risi. Evolving mario levels in the latent space of a deep convolutional generative adversarial network. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 221–228, 2018.

[38] Maxim Gumin. Wave Function Collapse, October 2020.

[39] Hwanhee Kim, Seongtaek Lee, Hyundong Lee, Teasung Hahn, and Shinjin Kang. Automatic generation of game content using a graph-based Wave Function Collapse algorithm. In *2019 IEEE Conference on Games (CoG)*, pages 1–4. IEEE, 2019.

[40] Tobias Nordvig Møller, Jonas Billeskov, and George Palamas. Expanding Wave Function Collapse with growing grids for procedural map generation. In *International Conference on the Foundations of Digital Games (FDG)*, pages 1–4, 2020.

[41] Hugo Scurti and Clark Verbrugge. Generating paths with WFC. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, volume 14, 2018.

[42] Arunpreet Sandhu, Zeyuan Chen, and Joshua McCoy. Enhancing Wave Function Collapse with design-level constraints. In *Proceedings of the 14th International Conference on the Foundations of Digital Games (FDG)*, pages 1–9, 2019.

[43] Timo Mantere and Janne Koljonen. Solving, rating and generating sudoku puzzles with ga. In *IEEE Congress on Evolutionary Computation*, pages 1382–1389. IEEE, 2007.

[44] Daniel Ashlock and Justin Schonfeld. Evolution for automatic assessment of the difficulty of Sokoban boards. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[45] Marc Van Kreveld, Maarten Löffler, and Paul Mutser. Automated puzzle difficulty estimation. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 415–422. IEEE, 2015.

[46] Mattia Crippa and Fabio Marrochi. *Monte-Carlo Tree Search for Sokoban.* Master's thesis, School of Industrial and Information Engineering, April 2018.

[47] Richard E Korf, Michael Reid, and Stefan Edelkamp. Time complexity of iterative-deepening-a. *Artificial Intelligence*, 129(1-2):199–218, 2001.

[48] Debosmita Bhaumik, Ahmed Khalifa, Michael Green, and Julian Togelius. Tree search versus optimization approaches for map generation. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, volume 16, pages 24–30, 2020.

[49] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[50] Oswaldo Velez-Langs. Genetic algorithms in oil industry: An overview. *Journal of petroleum science and engineering*, 47(1-2):15–22, 2005.

[51] Zachary Piserchia. *Applications of Genetic Algorithms in Bioinformatics*. PhD thesis, UC Riverside, 2018.

[52] Randy L Haupt and Douglas H Werner. *Genetic algorithms in electromagnetics*. John Wiley & Sons, 2007.

[53] Daniele Norton, Laura Anna Ripamonti, Mario Ornaghi, Davide Gadia, and Dario Maggiorini. Monsters of darwin: A strategic game based on artificial intelligence and genetic algorithms. In *GHITALY*, volume 1956. CEUR-WS, 2017.

[54] David Alexander Coley. *An introduction to genetic algorithms for scientists and engineers*. World Scientific Publishing Company, 1999.

[55] Midas Schonewille. Evaluating the effectiveness and applicability of novelty search and other quality-diversity algorithms compared to niching. Master's thesis, Utrecht University - Game and Media Technology, September 2020.

[56] Daniele Gravina, Ahmed Khalifa, Antonios Liapis, Julian Togelius, and Georgios N Yannakakis. Procedural content generation through quality diversity. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2019.

[57] Stephane Doncieux, Alban Laflaquière, and Alexandre Coninx. Novelty Search: a theoretical perspective. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 99–106, 2019.

[58] Joel Lehman and Kenneth O Stanley. Efficiently evolving programs through the search for novelty. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 837–844, 2010.

[59] Michael C Fu. Monte-Carlo tree search: A tutorial. In *2018 Winter Simulation Conference (WSC)*, pages 222–236. IEEE, 2018.

[60] Beatriz Nasarre Embid. Método de Monte-Carlo Tree Search (mcts) para resolver problemas de alta complejidad: Jugador virtual para el juego del go, 2012.

[61] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte-Carlo Tree Search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[62] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[63] Wei-Lun Chao. Machine learning tutorial. *Digital Image and Signal Processing*, 2011.

[64] Christopher Amato and Guy Shani. High-level reinforcement learning in strategy games. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 75–82. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[65] Miguel Morales. Kinds of RL Algorithms. `https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html`, 2018. [Online; accessed 11-November-2020].

[66] SV Burtsev and Ye P Kuzmin. An efficient flood-filling algorithm. *Computers & graphics*, 17(5):549–561, 1993.

[67] Luigi Troiano, Pasquale Davide De, and Pasquale Marinaro. Jenes Genetic Algorithms in Java. `http://jenes.intelligentia.it`, 2020. [Online; accessed 31-May-2021].

[68] Bradley C Wallet, David J Marchette, and Jeffrey L Solka. Matrix representation for genetic algorithms. In *Automatic Object Recognition VI*, volume 2756, pages 206–214. International Society for Optics and Photonics, 1996.

[69] Henrik Kniberg, Mattias Skarin, Prólogo de Mary Poppendieck, and David Anderson. Kanban y scrum–obteniendo lo mejor de ambos. *Prólogo de Mary Poppendieck & David Anderson. Estados Unidos de América: C4Media Inc*, 2010.

[70] Danielle Navarro and David Foxcroft. Learning statistics with jamovi: A tutorial for psychology students and other beginners (version 0.70). *Tillgänglig online: http://learnstatswithjamovi. com [Hämtad 14 december]*, 2019.

[71] Petr Jarušek and Radek Pelánek. Human problem solving: Sokoban case study. *Technická zpráva, Fakulta informatiky, Masarykova univerzita, Brno*, 2010.

[72] Elio Valenzuela. Project add: Adaptative difficulty dungeons. Bachelor thesis, Video Games Development and Virtual Reality Engineering, Universidad de Talca, September 2020.

[73] Gravina Daniele. Divergence and quality diversity. `https://github.com/DanieleGravina/divergence-and-quality-diversity`, 2021. [Online; accessed 06-June-2021].

[74] Martyn Amos and Jack Coldridge. A genetic algorithm for the zen puzzle garden game. *Natural Computing*, 11(3):353–359, 2012.

[75] Increpare games. Stephen's Sausage Roll. `https://store.steampowered.com/app/353540/Stephens_Sausage_Roll/`, 2020. [Online; accessed 22-June-2021].

[76] SJ Karman. Generating sokoban levels that are interesting to play using simulation. Master's thesis, 2018.

[77] Nils Froleyks and Tomás Balyo. *Using an algorithm portfolio to solve sokoban.* PhD thesis, Karlsruher Institut für Technologie (KIT), 2016.

[78] Heiner Marxen and Matthias Meger. JSoko Solver. `http://www.sokobano.de/wiki/index.php?title=JSoko_Solver`, 2017. [Online; accessed 28-may-2021].

[79] Daniele Gravina. Divergence-and-quality-diversity. `https://github.com/DanieleGravina/divergence-and-quality-diversity`, 2020. [Online; accessed 22-June-2021].

A. Datos demográficos

Presentaremos algunos datos demográficos sobre las personas que han participado jugando en la aplicación web.

En total han participado 77 personas en la prueba de usuario. Esta duró 4 días y fue compartida en grupos de expertos jugadores de Sokoban, aficionados a los puzles, grupos de videojuegos y la carrera de Ingeniería en Desarrollo de videojuegos y Realidad Virtual de la Universidad de Talca, Chile. En la figura A.1 podemos apreciar que la mayor cantidad de participantes son estudiantes universitarios alcanzando un 48 % del total, le sigue titulado universitario y *bachelor degree* con un 10 % cada uno.

También realizamos un gráfico con las edades de los participantes, de la imagen A.2 podemos apreciar que hay una concentración de edades entre los 20 y 30 años con algunos datos atípicos en los 44 y 14 años. Creemos que es útil identificar a la población que les gusta este tipo de videojuegos para posteriores investigaciones que requieran público objetivo similar.

Registramos la cantidad de niveles completos e incompletos por los usuarios, ya que ellos tenían la opción de rendirse y pasar al siguiente nivel. En la figura A.3 podemos apreciar que un 69 % de los niveles jugados del conjunto 1 fueron completados y para la figura A.4 del conjunto 2 fue un 78 %.

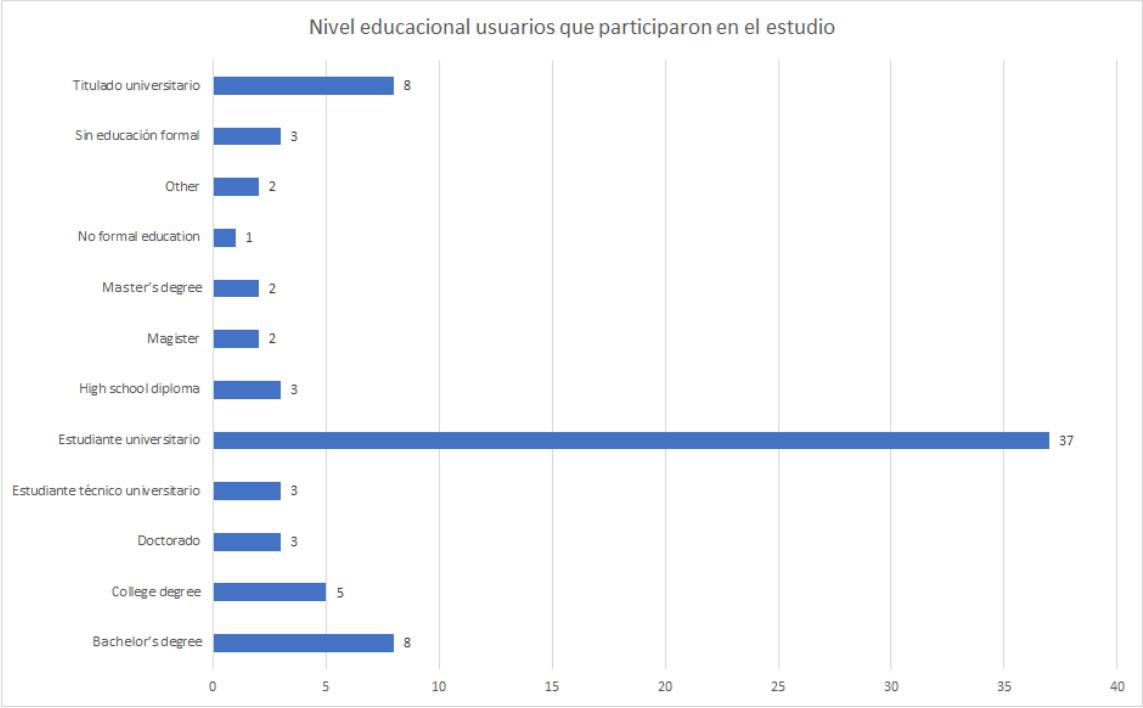


Figura A.1: Nivel educacional de participantes del estudio de usuario

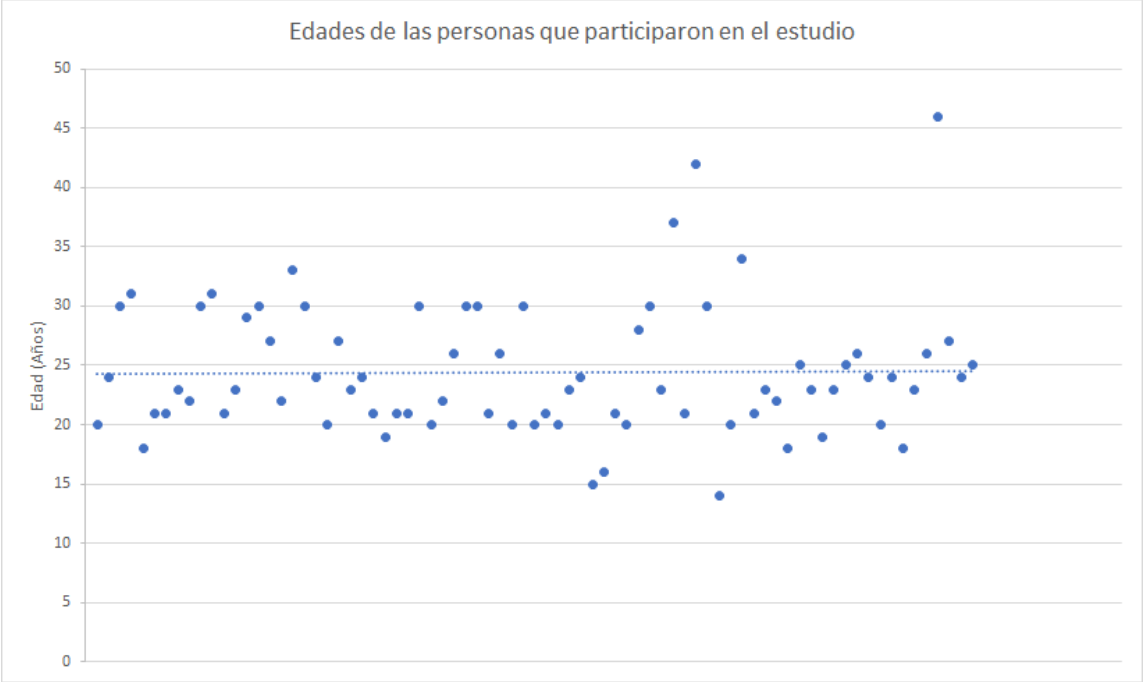


Figura A.2: Edades de los usuarios que han participado jugando en Sokostigation

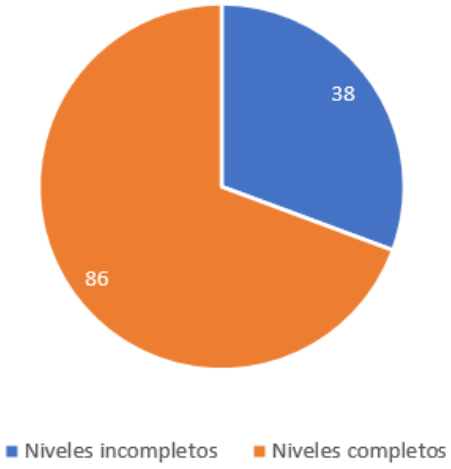


Figura A.3: Cantidad de niveles completos e incompletos en el conjunto de niveles 1

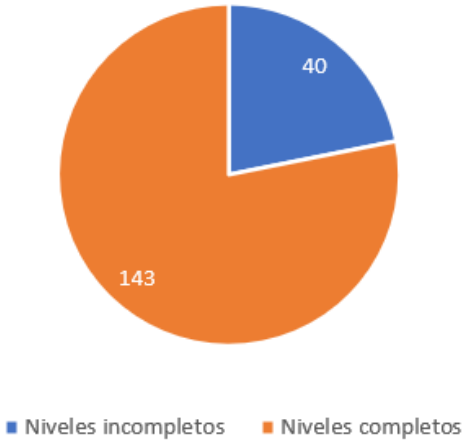


Figura A.4: Cantidad de niveles completos e incompletos en el conjunto de niveles 2

B. Sokoban

Sokoban es un juego del tipo puzle, ampliamente investigado debido a la naturaleza de los subproblemas inherentes en él, su resolución entra en la categoría NP-Hard ya que este problema tiene un alto factor de ramificación [13]. Fue Creado en 1980 por Hiroyuki Imabayashi y publicado por *Thinking Rabbit* en 1982 es uno de los puzles más jugados de su tipo. Con el pasar del tiempo muchos juegos se han visto influenciadas por las mecánicas de Sokoban dando lugar a nuevas preguntas interesantes de investigar [7].

El significado de Sokoban en japonés es "Encargado de almacén". Su concepto de juego está basado en la necesidad de ordenar un almacén, de esta idea desprenden los componentes básicos que dan forma a las mecánicas de juego, un nivel de juego común puede apreciarse en la imagen B.1, algunos de los elementos principales que se pueden distinguir son las cajas, las metas, los obstáculos y el jugador. Como se muestra en la imagen B.2 el objetivo del puzle es empujar todas las cajas hacia sus metas. Las reglas son las siguientes:

1. El jugador empuja las cajas en la dirección que se mueve si en la nueva dirección frente a la caja hay un lugar vacío.

2. El jugador no puede traspasar las murallas ni cajas.

3. El jugador y las cajas solo se mueven un espacio a la vez cuando están siendo empujadas.

4. El nivel está completo cuando todas las cajas están en las metas.

A medida que el usuario está tratando de resolver un nivel, puede darse la situación donde se converge a un estado llamado "deadlock", es decir que ninguno de los

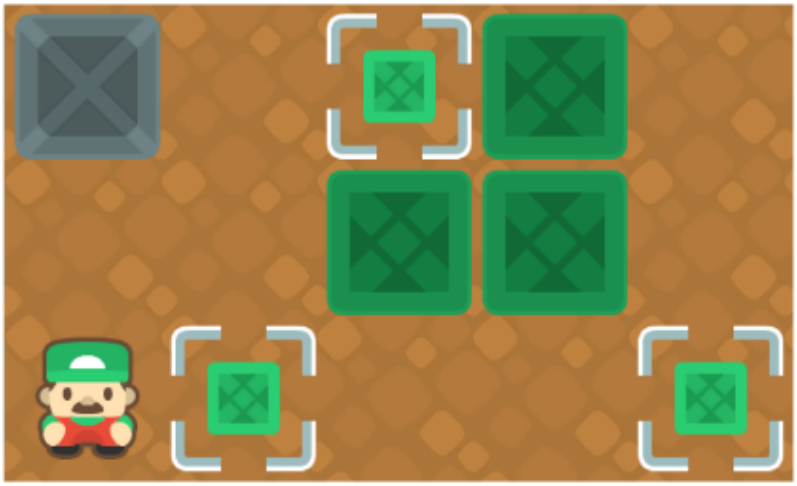


Figura B.1: Ejemplo de un nivel tradicional de Sokoban [76]

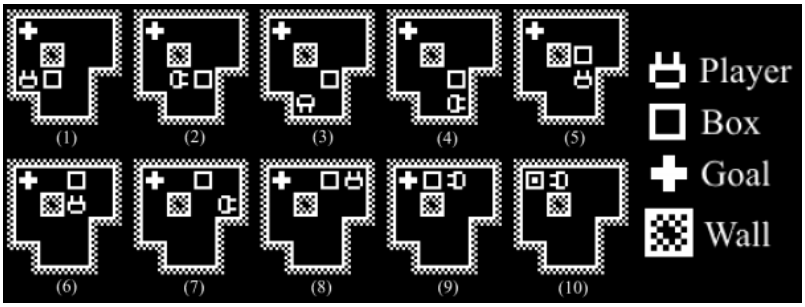


Figura B.2: Pasos para solucionar un nivel de Sokoban [23]

estados posteriores de juego conducirá al estado solución, por ende, se debe reiniciar el nivel y empezar otra vez. En la imagen B.3 se puede observar este tipo de estado de juego.

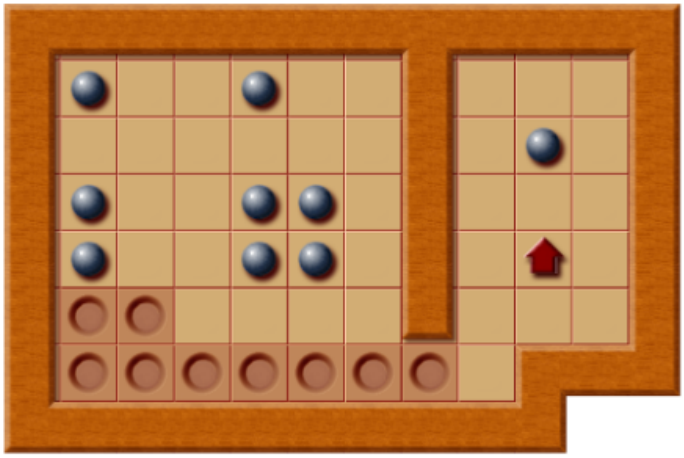


Figura B.3: Ejemplo de deadlock, La caja ubicada en la esquina superior izquierda no se puede empujar a ningún lado. Las cajas en los cuatro grupos en el medio no permiten que se muevan entre sí. Lo mismo ocurre con las dos cajas a lo largo de la pared izquierda. La caja en la pared superior se puede mover, pero nunca alcanzará una meta. Lo mismo se aplica a la caja de la habitación del lado derecho [77]

.

C. JSoko

JSoko es un programa desarrollado en el lenguaje de programación Java, el proyecto es de código abierto y fue realizado por Heiner Marxen y Matthias Meger. Posee opciones para que los niveles sean jugados en el mismo programa, así como también un *solver* capaz de solucionar una gran cantidad de niveles [78], sin embargo hay ciertas limitaciones:

1. Esto significa que solo se puede usar para niveles que no superen los 70 * 70 *tiles*.

2. El *solver* solo intenta encontrar una solución independientemente del número de movimientos o empujes necesarios para resolver el nivel.

3. El *solver* solo se puede usar como parte de JSoko, ya que no existe una versión independiente. Pero existe la opción de separar el código de este del programa completo.

Como se puede apreciar en la imagen C.1 hay 4 criterios para que el *solver* trabaje [79], estos son:

1. Cualquier solución: el *solver* intenta resolver el nivel sin tomar en cuenta los movimientos o empujes necesarios.

2. Empuje óptimo: el *solver* busca una solución de empuje óptima. Sin embargo, es posible que el número de movimientos de la solución encontrada no sea el óptimo.

3. Empuje óptimo con los mejores movimientos: el *solver* busca una solución de empuje óptima que tenga la menor cantidad posible de movimientos. Mover

óptimo con los mejores empujes: el *solver* busca una solución de movimiento óptimo que tenga el menor número posible de empujes.

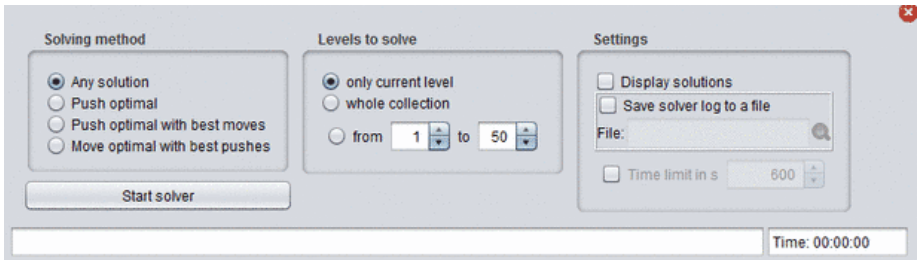


Figura C.1: Configuración posible para dar solución a un nivel de Sokoban en JSoko

En la siguiente sección "Levels to solve" podemos configurar cuales niveles queremos resolver:

1. Sólo el nivel actual, también podemos hacer algunos movimientos y ordenar que el *solver* trate de encontrar una solución desde el estado actual.

2. El *solver* intenta resolver todos los niveles del conjunto cargado.

3. Especificar qué niveles queremos que se resuelvan en particular.

Finalmente, en "Settings" podemos:

1. Mostrar soluciones: Mostrar la solución encontrada.

2. Guardar registro: Se puede seleccionar un archivo para que sirva de contenedor de la/las soluciones.

3. Límite de tiempo: Se puede establecer un límite de tiempo por cada nivel para la ejecución del *solver*. Al superarse este limite el *solver* deja de buscar una solución.

D. Ejemplos de niveles creados

En este apéndice reunimos algunos de los niveles creados con nuestro generador para que sirvan como ejemplos de lo que este puede generar. En la figura D.1 reunimos niveles que fueron creados con la restricción de no tener conjunto de baldosas vacías de 3x3. Luego en la figura D.2 reunimos niveles que fueron creados con la restricción de no tener conjuntos de baldosas vacías de 3x4 ni 4x3.

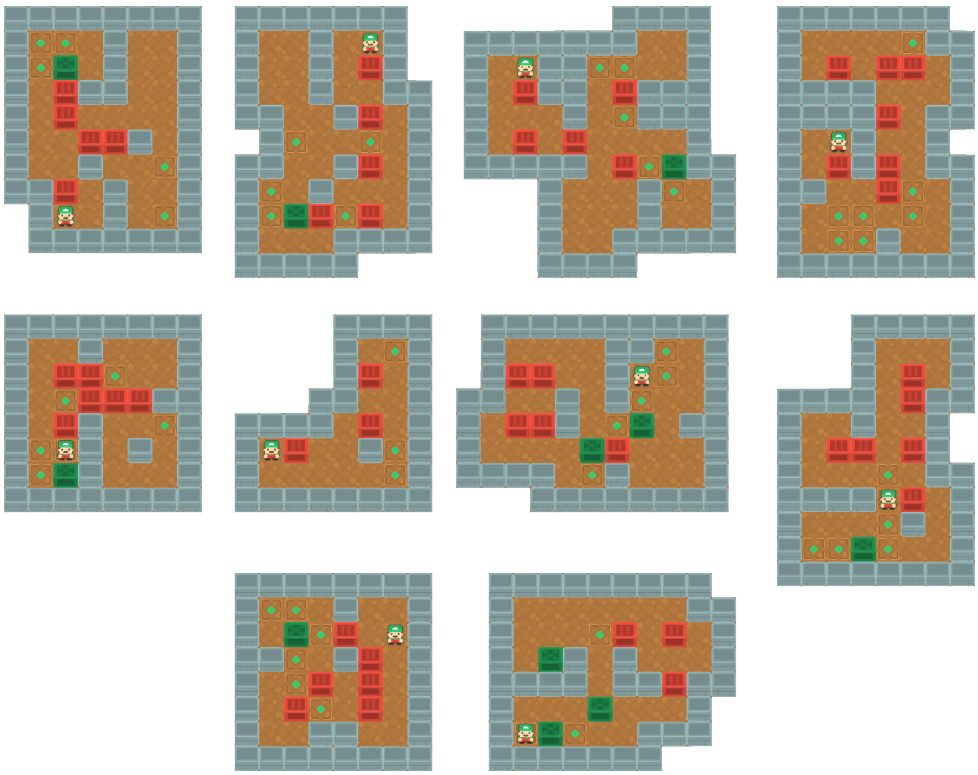


Figura D.1: Conjunto de niveles con restricción de no tener espacios vacíos de 3x3 baldosas

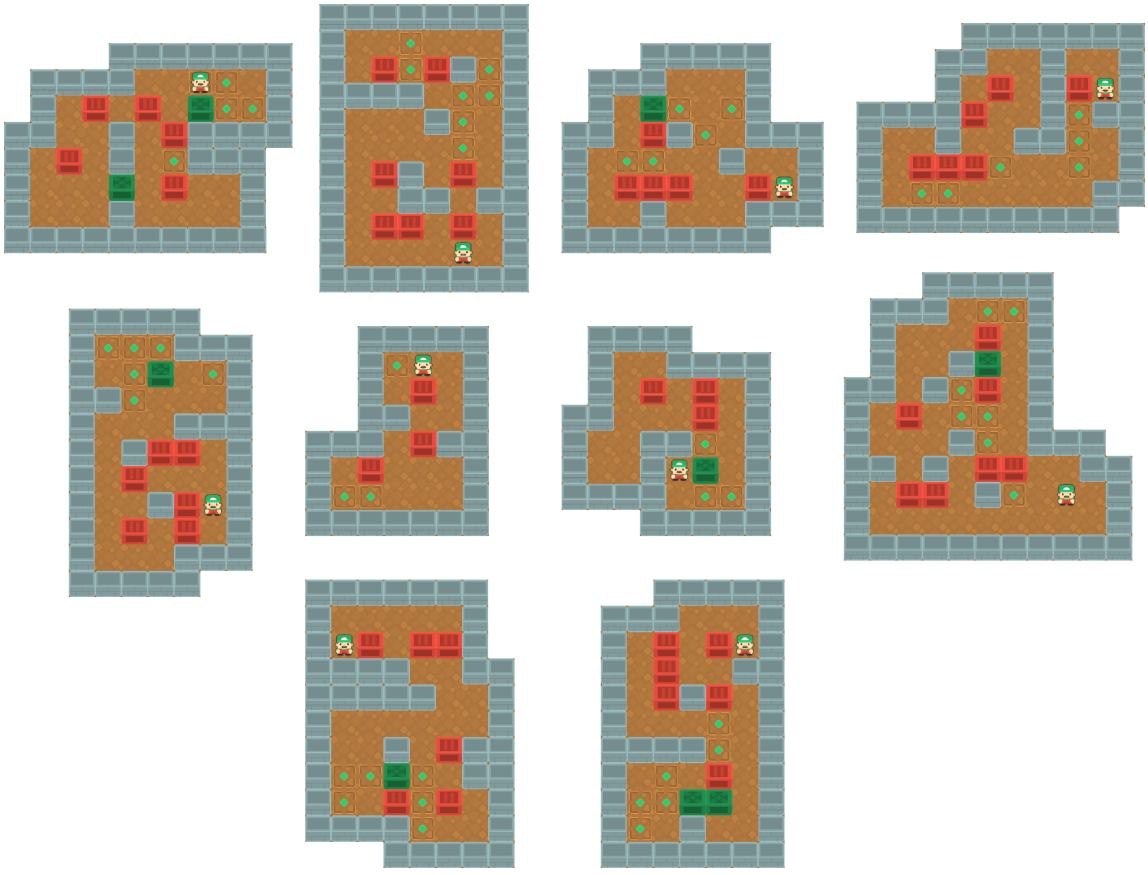


Figura D.2: Conjunto de niveles con restricción de no tener espacios vacíos de 3x4 y 4x3 baldosas

E. Sokostigation: User testing App

A continuación mostramos algunos detalles sobre la aplicación para llevar a cabo *User Testing* bautizada como Sokostigation.

Cómo se mencionó en la sección 4.3.4, hemos dividido los niveles en dos conjuntos. En las figuras E.1, E.2, E.3 y E.4 se pueden observar los distintos subconjuntos jugados por los usuarios.

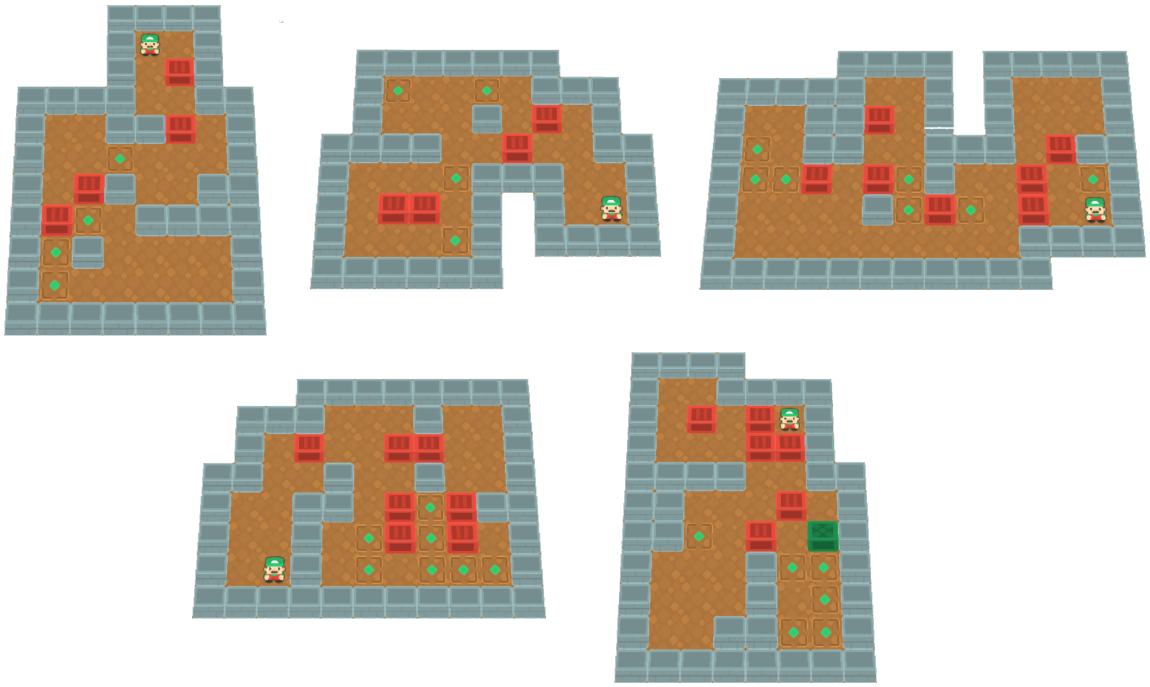


Figura E.1: Subconjunto 1 del conjunto 1 de niveles jugados en Sokostigation

Los datos recabados del estudio de usuario, niveles utilizados, formatos de niveles e imágenes vinculadas a los experimentos, pueden ser encontradas en el siguiente enlace: datos pruebas de usuario.

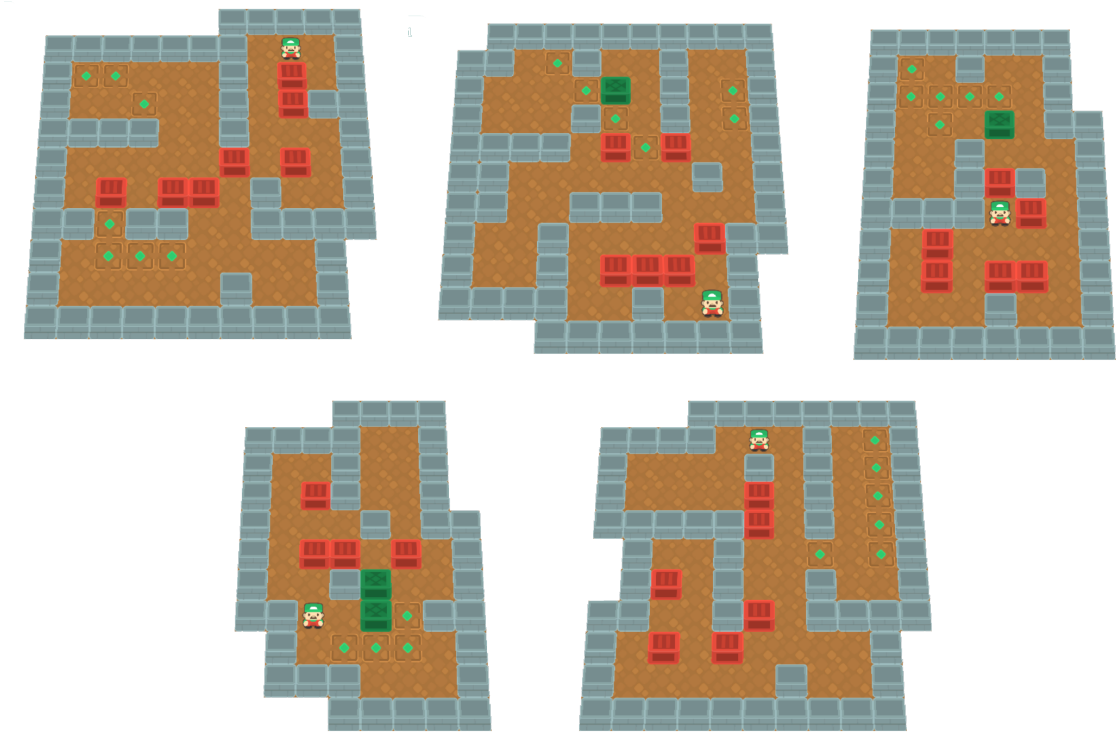


Figura E.2: Subconjunto 2 del conjunto 1 de niveles jugados en Sokostigation

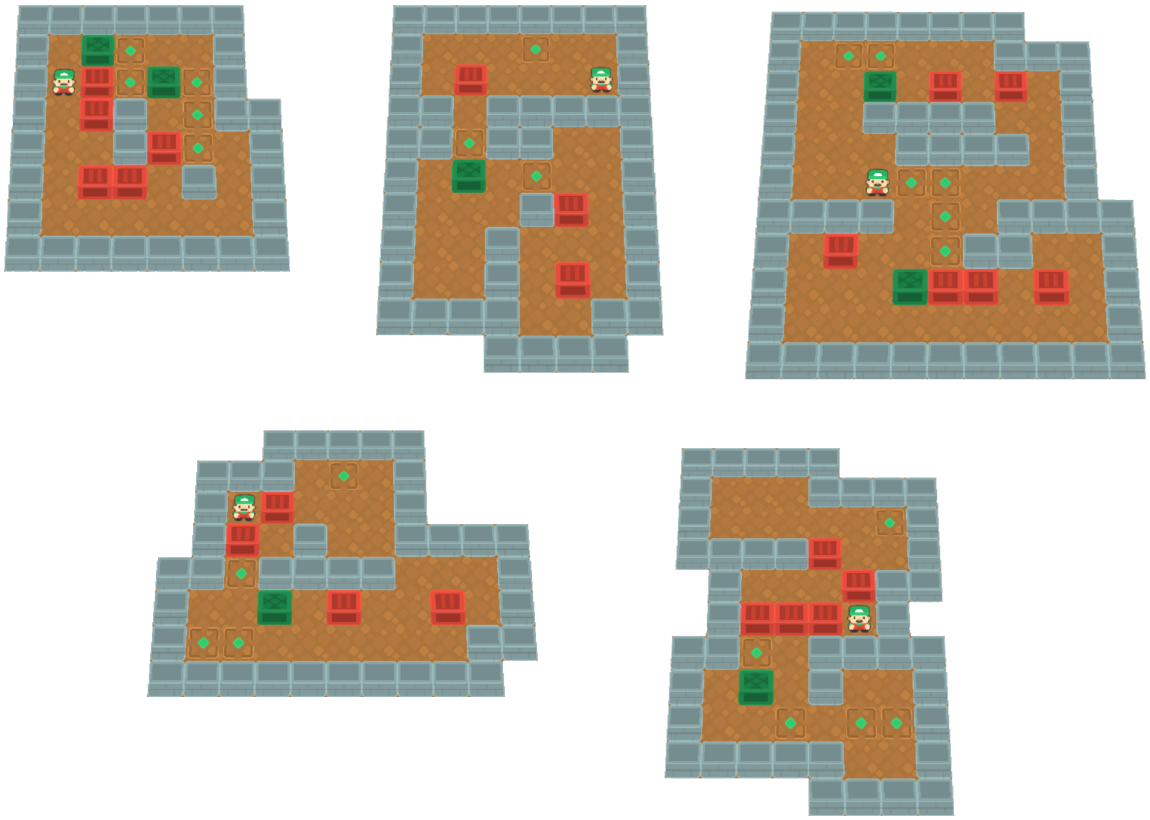


Figura E.3: Subconjunto 1 del conjunto 2 de niveles jugados en Sokostigation

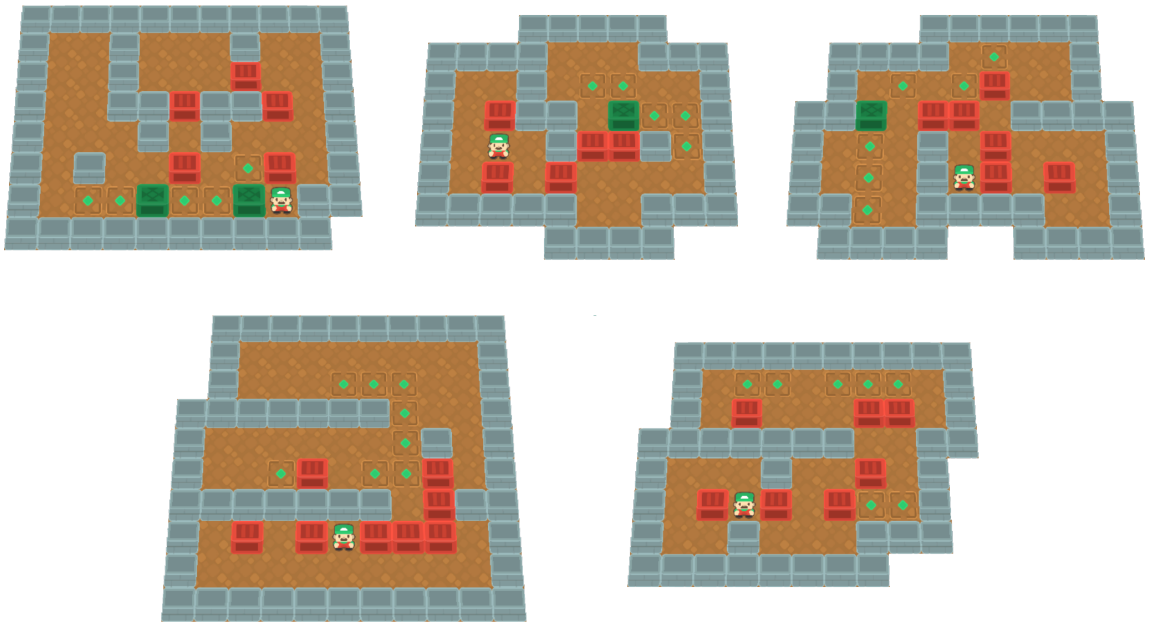


Figura E.4: Subconjunto 2 del conjunto 2 de niveles jugados en Sokostigation