Carlo juega a LightsOut

Miguel Alcázar Valentín
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
migalcval@alum.us.es / alcazar.miguel30@gmail.com

Jose Ángel Rodríguez Durán
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
josroddur@alum.us.es / DLJ7666

***Resumen*—Este trabajo aborda la resolución del juego *LightsOut* mediante técnicas de planificación automática. El sistema desarrollado representa el problema usando el lenguaje PDDL e implementa un algoritmo de búsqueda en Python con una fase común de exploración informada y una fase específica, una basada en el algoritmo de búsqueda de A* y otro en el algoritmo de Monte Carlo Tree Search (MCTS). Se analiza el rendimiento de los algoritmos en tableros de tamaño 5x5, discutiendo los principales desafíos de diseño y los resultados obtenidos.**

***Palabras clave*—Planificación automática, MCTS, LightsOut, Unified Planning, PDDL, Python, A* search.**

I. Introducción

LightsOut es un juego que consiste en una cuadrícula de bombillas (más comumente llamadas celdas) que poseen 2 estados, encendidas o apagadas. En cada jugada o acción, el jugador cambia el estado de una celda y el de sus vecinas ortogonales (formando una cruz). El objetivo es encontrar una secuencia de pulsaciones que permita alcanzar un estado objetivo, normalmente aquel en el que todas las celdas están encendidas.

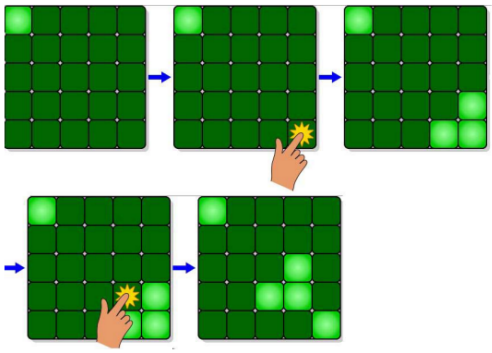

Fig. 1. Representacion del funcionamiento del minijuego *LightsOut*. Imagen obtenida del pdf *https://matematicas.uam.es/ eugenio.hernandez*

Para abordar este problema, se ha seguido una metodología dividida en etapas. En primer lugar, se ha representado formalmente el entorno mediante el lenguaje PDDL (Planning Domain Definition Language). Posteriormente, se ha desarrollado un sistema en Python capaz de interpretar dicha representación, simular transiciones de estado y resolver el problema utilizando dos algoritmos de búsqueda distintos.

- **Búsqueda A*:** Presentado por primera vez en 1968 por Peter E. Hart, Nils J. Nilsson y Bertram Raphael; el algoritmo A* (llamado A estrella, *A star* traducido del inglés) es un algoritmo de búsqueda en grafos de tipo heurístico, es decir, necesita de una heurística (función que aproxima el número de pasos mínimo ideal de un algoritmo para resolver un problema) para funcionar correctamente.
- **Monte Carlo Tree Search (MCTS)** El algoritmo de árbol de búsqueda de Monte Carlo (*Monte Carlo Tree Search*, en inglés, por sus siglas *MCTS*), es un algoritmo de búsqueda en grafos de tipo heurístico que se caracteriza por sus iteraciones divididas en 4 etapas, que serán explicadas más adelante.

II. Preliminares

En esta sección se describen brevemente las técnicas y herramientas empleadas para abordar el problema del juego *LightsOut*, así como una revisión de algoritmos y enfoques similares utilizados previamente en problemas de planificación automática.

A. Estructura del proyecto

La implementación del sistema se ha organizado de manera modular, distribuyendo sus componentes en una estructura de carpetas que facilita la separación de responsabilidades y la mantenibilidad del código. A continuación, se describe brevemente la estructura principal del proyecto:

- `doc/`: Carpeta que contiene la documentación del proyecto, incluyendo el presente informe técnico en formato LaTeX.
- `src/`: Carpeta principal del código fuente del sistema.
  - `pddl/`: Subcarpeta dedicada a almacenar los archivos relacionados con la generación y lectura de dominios y problemas en formato PDDL.
  - `board_solver.py`: Script que implementa el algoritmo de búsqueda A*, encargado de resolver el tablero a partir de un dominio y problema en PDDL.
  - `board_solver_mcts.py`: Script que implementa el algoritmo MCTS, encargado de resolver el tablero a partir de un dominio y problema en PDDL.

- `lights_out_board.py`: Módulo que contiene la lógica del tablero del juego *LightsOut*, incluyendo la representación del estado y las posibles acciones.
- `Node.py`: Define la estructura de los nodos utilizados en el algoritmo de búsqueda A* y en la fase de simulación del algoritmo MCTS.
- `parsers.py`: Implementa el parser personalizado que traduce los archivos PDDL a una representación interna manipulable desde Python.

B. Métodos empleados

El enfoque de este trabajo combina una fase de exploración estructurada del espacio de estados con una estrategia basada en los algoritmos de búsqueda de *A** y *Monte Carlo Tree Search (MCTS)*. A continuación se detallan los componentes más relevantes:

- **Lenguaje PDDL:** Se ha utilizado para modelar el dominio del problema y generar automáticamente instancias del juego. Se definieron los tipos, predicados y acciones necesarias para representar el entorno LightsOut.
- **Unified Planning:** Librería de planificación automática en Python empleada para construir objetos del dominio, definir condiciones iniciales y metas, y generar archivos `.pddl` válidos.
- **Parser y simulador personalizados:** Se ha implementado desde cero un parser de dominio y problema en PDDL, así como un simulador del entorno que permite aplicar acciones y verificar el cumplimiento del objetivo.
- **Algoritmo MCTS:** Técnica basada en simulaciones aleatorias y retropropagación de resultados. En este caso, no nos ha dado tiempo a acabar Monte Carlo Tree Search por falta de tiempo.
- **Heurística h-add simplificada:** Se emplea en la fase inicial para evaluar la distancia estimada al objetivo, calculando el número de celdas aún apagadas en un estado dado.

C. Trabajo relacionado

El algoritmo de búsqueda A* que se ha implementado está basado en el que nos fue aportado en la teoría, elaborado por el departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Sevilla, que será explicado en el apartado de **Metodología**.

III. Representación del problema en PDDL

Para resolver el problema con técnicas de planificación, se ha optado por una modelización en PDDL. En el dominio se definen:

- Tipos de objetos: `cell`, que representa una celda del problema.

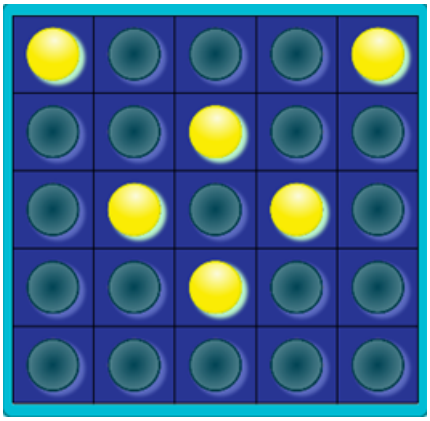

Fig. 2. Ejemplo de un tablero de *Lights Out* de tamaño 5x5. Imagen sacada del juego de *Google Play* del mismo nombre de 2023. Licencia Creative Commons

- Objetos: Las celdas que tiene el tablero, cuyo nombre tiene el formato `ci-j`, donde *i* es la fila y *j* la columna que ocupa la celda. En el caso de nuestro tablero estándar de 5x5, se generan 25 celdas, desde `c0-0` hasta `c4-4`.
- Predicados: `cell_on(cell)`, que representa que una celda concreta está encendida, y `cell_adjacent(cell1, cell2`, que define si 2 celdas concretas son ortogonalmente adyacentes.
- Acción: `press_cell(c)`, que cambia el estado de la celda pulsada `c` y de las celdas ortogonalmente adyacentes a la misma.

El problema se genera automáticamente desde Python mediante la librería Unified Planning, permitiendo definir de forma reproducible:

- Estado inicial: `init`, en el que se indica las celdas que están encendidas al iniciarse el problema, así como las adyacencias ortogonales entre las celdas.
- Estado objetivo: `goal`, donde se indican las celdas que deben estar encendidas para dar por finalizado el problema. Nótese que no se indica ninguna adyacencia, ya que no hay forma de mutar este predicado mediante ninguna acción.

IV. Metodología

El sistema implementado sigue una arquitectura que consta de dos fases principales: una parte común de exploración del problema PDDL y una parte específica que ejecuta el algoritmo correspondiente (ya sea búsqueda A* o MCTS).

En primer lugar, para el funcionamiento óptimo de ambos algoritmos, precisamos de una heurística admisible. Siguiendo lo explicado en la teoría de la asignatura, decidimos decantarnos por aproximar la heurística óptima con una heurística segura, admisible, consciente del objetivo y consistente, mediante el método de h*add* con los planes relajados.

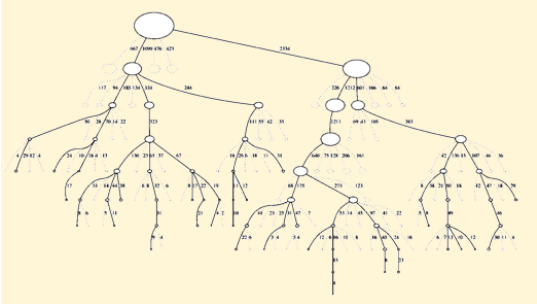

Fig. 3. *Monte Carlo Tree Search.*

Sin embargo, este método no era independiente del dominio, pues dependía del tamaño del tablero seleccionado, por lo que finalmente optamos por aproximar *h_add* al **número de celdas apagadas por cada estado**. Es decir, además de eliminar el *borrado* de la acción `press_cell`, también eliminamos las implicaciones que tiene la acción en las celdas ortogonalmente adyacentes. De esta forma, obtenemos una función sencilla de implementar y con un bajo coste computacional, sin que esta deje de ser suficiente para el correcto funcionamiento de los algoritmos.

A continuación se describe el funcionamiento de ambos algoritmos en forma de pseudocódigo.

La función `solve_board.py` implementa el núcleo del algoritmo A* utilizado para resolver instancias del juego *Lights Out* definidas mediante archivos PDDL. A partir del dominio y el problema proporcionados, se extrae la información relevante sobre las celdas, sus relaciones y el estado inicial. Luego, se realiza una búsqueda informada que intenta alcanzar el objetivo —apagar todas las luces— aplicando una secuencia de acciones válidas. A continuación se muestra el pseudocódigo que resume los pasos principales del proceso [3].

Cada subrutina corresponde a una fase del algoritmo Monte Carlo Tree Search (MCTS):

- **Selección**: se recorren los nodos del árbol hasta encontrar uno aún expandible.
- **Expansión**: se genera un hijo con una acción válida aún no aplicada desde el nodo.
- **Simulación**: se ejecuta una secuencia aleatoria de acciones desde el nodo actual hasta alcanzar una condición de parada:
  - Se alcanza una profundidad igual al número de celdas.
  - Se encuentra una solución.
  - Se detecta un ciclo.
- **Retropropagación**: se actualizan los valores de los nodos del camino recorrido, de forma que:
  - Si el estado en cuestión es el último al que ha llegado el algoritmo.
    - Si ha llegado por encontrar un ciclo, asigna a dicho estado un valor de *-1*.
    - Si ha llegado por alcanzar el número máximo de iteraciones, asigna a dicho estado un valor de *0*.
    - Si el estado es solución, asigna a dicho estado el valor siguiente: *número máximo de iteraciones - longitud del camino hasta la solución + 1*.
  - En caso contrario, asigna a cada estado el valor siguiente: *valor del estado hijo / número de apariciones del estado en las iteraciones*.

SOLVE_BOARD(*domain_file*, *problem_file*)
1 ***Entrada*: archivos `.pddl` de dominio y problema**
2 ***Salida*: secuencia de acciones que resuelve el tablero**
3 **dominio ← parsear archivo de dominio**
4 **problema ← parsear archivo de problema**
5 **celdas ← obtener celdas desde constantes del dominio**
6 **acciones ← `press_cell(c)` para cada celda c**
7 **adyacencias ← extraídas de predicados binarios**
8 **objetivo ← celdas encendidas**
9 **estado_inicial ← celdas encendidas en el estado inicial**
10 **Inicializar:**
11 $g[s] \leftarrow \infty \quad \forall s$**, excepto** g**[estado_inicial]** $\leftarrow 0$
12 $f[s] \leftarrow g[s] + h(s)$
13 **`open` ← {estado_inicial}**
14 **`closed` ← ∅**
15 **`nodes` ← conjunto de nodos**
16 **Mientras `open` no esté vacío:**
17 $s \leftarrow \arg\min_{s \in \texttt{open}} f[s]$
18 **mover s de `open` a `closed`**
19 **Si objetivo $\subseteq s$:**
20 **retornar `get_solution_path(s, nodes)`**
21 **Para cada acción aplicable a en s:**
22 $s' \leftarrow$ **resultado de aplicar a sobre s**
23 **nodo_nuevo ← `Node(s', s, a)`**
24 **Si $s' \in$ `open` $\cup$ `closed` y mejora el coste $g[s']$:**
25 **actualizar $g[s']$, $f[s']$ y nodo correspondiente**
26 **Si $s' \notin$ `open` $\cup$ `closed`:**
27 **inicializar $g[s']$, $f[s']$ y añadir a `open` y `nodes`**
28 **Fin para**
29 **Fin mientras**
30 **retornar lista vacía (no se encontró solución)**

Fig. 4. Pseudocódigo simplificado del algoritmo de búsqueda A* implementado en `board_solver.py`

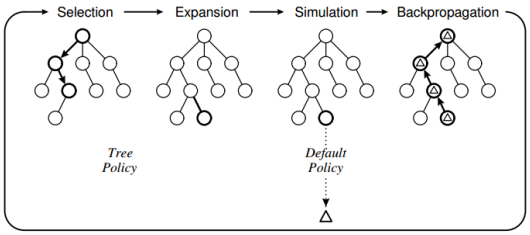

Fig. 5. Diferentes subrutinas del algoritmo *Monte Carlo Tree Search*. Imagen sacada de la página web *https://www.cs.us.es/ fsancho/Blog/posts/MCTS.md*

La implementación se ha hecho desde cero, incluyendo un parser PDDL personalizado, la construcción de nodos, y la

```
Función mcts_algorithm(domain, problem, k, tries)
  values ← None
  Para i ← 0 hasta tries − 1 hacer:
      Para j ← 0 hasta k − 1 hacer:
          Si i = 0 ∧ j = 0:
            values ← mcts_iteration(domain, problem)
          Sino:
            values ← mcts_iteration(domain, problem, values, i)
      Fin Para
  Fin Para
  Retornar values
```

```
Función mcts_iteration(dom, prob, vals=None, extra=0)
  1. Inicialización
      Parsear dom y prob
      s0 ← estado inicial
      Si vals=None: vals ← {}
  2. Selección (3 pasos)
      Para 3 iteraciones hacer:
          a ← acción aleatoria
          s' ← aplicar(a)
          actualizar estructuras
      Si extra> 0: selección por menor valor
  3. Expansión
      Repetir:
          a ← acción aleatoria
          s' ← nuevo estado
          Hasta s' no visitado
    4. Simulación (A*)
          Ejecutar A* limitado con h = celdas apagadas
          Si solución: guardar camino
          Si estado repetido: vals ← −1
    5. Retropropagación
          Si solución: vals ← longitud(camino)
          Actualizar valores recursivamente
    Retornar vals
```

Fig. 6. Algoritmo MCTS adaptado para Lights Out

gestión de estados con conjuntos inmutables.

V. Dificultades encontradas y decisiones de diseño

Durante el desarrollo del proyecto se enfrentaron varios retos técnicos:

- En las etapas iniciales del desarrollo se intentó utilizar la librería *py2pddl* para la generación de dominios PDDL. Sin embargo, debido a diversas limitaciones y errores en su funcionamiento, se optó por migrar a la librería *Unified Planning*, que ofreció mayor flexibilidad y estabilidad.
- Como era importante entender bien los efectos condicionales definidos en los dominios PDDL, se decidió implementar un parser propio que tradujera la estructura del dominio a una representación interna que se pudiera manejar fácilmente en Python.
- Para la representación de los estados del juego se eligió una estructura basada en conjuntos inmutables (`frozenset`), lo que facilitó su gestión eficiente en estructuras de datos como diccionarios y conjuntos hashables.
- En un incio se optó por implementar la heurística h*max*, sin embargo esta resultó ser demasiado optimista ya que siempre devolvía *1* para cualquier estado que no fuera el objetivo, por la propia naturaleza de h*add*.
- Posteriormente se optó por implementar un módulo que calculara los planes relajados del problema y obtuviera el valor de h*add*, sin embargo este resultó ser de un costo computacional muy elevado y de una complejidad demasiado alta para que fuera viable, por lo que finalmente nos decantamos por aplicar la heurística explicada en metodología.

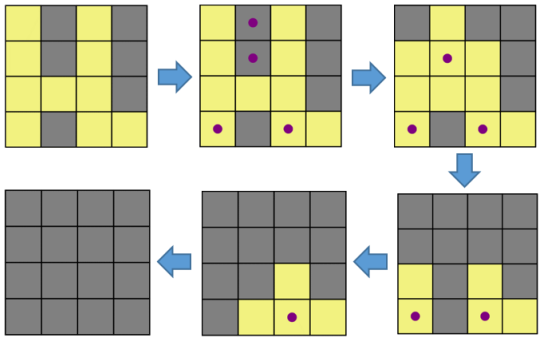

Fig. 7. *Ejemplo de resolución de Lights Out.*

VI. Resultados

El sistema implementado ha sido probado sobre múltiples configuraciones del juego *LightsOut*, con tableros de tamaño 5x5 y estados iniciales generados aleatoriamente. El objetivo principal de estas pruebas ha sido validar el correcto funcionamiento del parser, la simulación del entorno, y la ejecución del algoritmo desarrollado.

A. Funcionalidad del sistema

Las pruebas se han centrado en comprobar que, dado un problema definido en archivos `.pddl`, el sistema es capaz de:

- Interpretar correctamente el dominio y problema mediante el parser desarrollado.
- Representar adecuadamente el estado inicial, incluyendo el estado de cada celda y las relaciones de adyacencia.
- Aplicar de forma correcta las acciones definidas, alterando los estados conforme al efecto esperado.
- Generar una secuencia de acciones que transforma el estado inicial en el estado objetivo.

VII. Conclusiones

A lo largo de este trabajo se ha abordado el problema de resolución del juego *LightsOut* mediante técnicas de planificación automática. Para ello, se ha modelado el dominio del problema en el lenguaje PDDL, y se ha implementado desde

cero un sistema completo en Python que incluye un parser personalizado, un simulador del entorno, y los algoritmos de búsqueda Monte Carlo Tree Search y A*.

El sistema fue diseñado para poder leer descripciones PDDL del dominio y del problema, generar estados, aplicar acciones, y encontrar secuencias que transformen el estado inicial en uno objetivo. Durante las fases de desarrollo, se logró validar funcionalmente gran parte del sistema: el parser interpreta correctamente las estructuras del dominio, el simulador aplica de manera adecuada las acciones según los efectos condicionales, y ambos algoritmos exploran el espacio de estados de forma coherente, con el algoritmo de MCTS dando una solución coherente con un número considerable de iteraciones.

Sin embargo, a pesar de que el sistema no genera errores de ejecución y realiza las búsquedas tal y como fueron planteadas, el algoritmo no logra encontrar una solución válida al problema, incluso en configuraciones simples.

Como ideas de mejoras proponemos:

- Realizar pruebas de unidad más detalladas en cada parte del sistema: parser, generador de acciones, aplicación de efectos, heurística y comparación de estados.
- Incluir una visualización gráfica o paso a paso del tablero tras cada acción aplicada, para observar si el estado evoluciona correctamente.
- Revisar y simplificar la definición de efectos condicionales, dado que su implementación fue una de las partes más complejas y propensas a errores.

REFERENCIAS

[1] Documentacion acerca de Unified Planning. Disponible en: `https://unified-planning.readthedocs.io/en/latest/`.

[2] "Práctica 4: Planificación automática," Universidad de Sevilla, Departamento de Ciencias de la Computación e Inteligencia Artificial. [Material docente].

[3] "Tema 4: Planificación automática," Universidad de Sevilla, Departamento de Ciencias de la Computación e Inteligencia Artificial. [Material docente].

[4] W3Schools. Disponible en: `https://www.w3schools.com/`

[5] Lanshor. Disponible en: `https://www.lanshor.com/pathfinding-a-estrella/`

[6] Khan Academy. Disponible en: `https://es.khanacademy.org`

[7] Introducing Monte Carlo Tree Search. Disponible en: `https://mcts.ai`

[8] OpenAI, ChatGPT, *https://chat.openai.com*. Utilizado para obtener información sobre la sintaxis y estructura de documentos LaTeX.