Informe Electivo de Heurística
Tarea 1 - Brandub

Javier Mahana
Joaquín Herrera

14 Mayo 2023

Abstract

En este documento se aborda la implementación del juego de mesa Brandub en C++. Para esto el tablero fue representado con bitsets, debido a su rapidez de ejecución y su eficiencia en el uso de memoria. Una vez que el juego base fue implementado se creo una función heurística con la cual se evalúa el tablero, aspecto desafiante al tratarse de un juego no simétrico. Finalmente se intento utilizar algoritmos de búsqueda adversaria para hacer que dos jugadores controlados por la maquina jueguen.

1 Introducción

Brandub es un antiguo juego de mesa irlandés, asimétrico, de suma cero para dos jugadores, uno ocupará piezas blancas y el otro piezas negras.

Reglas:

- El tablero tiene un tamaño de 7x7 cuadrados, con el cuadrado central y las cuatro esquinas marcados.
- Hay trece piezas en total: un rey y sus cuatro defensores, y ocho atacantes.
- Las piezas se mueven ortogonalmente cualquier distancia, sin aterrizar ni saltar sobre otras piezas en el tablero.
- Una pieza es capturada cuando es rodeada ortogonalmente en dos cuadrados opuestos por piezas enemigas. El rey puede ayudar a capturar piezas en asociación con un defensor.

- Una pieza también puede ser capturada si se encuentra entre un enemigo y el cuadrado central vacío o un cuadrado de esquina.
- El rey gana el juego al llegar a cualquier cuadrado de esquina marcado. Los atacantes ganan si capturan al rey.
- El juego termina en empate si se repite una posición.

2 Implementación

Se utilizó C++ para implementar el juego Brandub. Uno de los principales desafíos fue representar el tablero de juego, que tiene un tamaño de 7x7 y contiene piezas blancas, negras y un rey blanco, junto con casillas especiales que tienen lógicas de juego diferentes. Para resolver este problema, se aprovechó el hecho de que el tablero ocupa solo 49 casillas, que se pueden representar fácilmente como bits en un unsigned long long, que tiene 64 bits disponibles, dejando 15 bits de sobra. Para esto, se utilizó una estructura de datos de C++ llamada "bitsets", que permite utilizar bits de manera flexible y fácil con un solo unsigned long long como base. Se usaron 3 bitsets para representar la posición de las piezas en el tablero: uno para las piezas blancas, otro para las negras y otro para el rey blanco. Además, se aprovechó el espacio de bits sobrante para incluir un bit en 0 al final de cada fila del tablero. De esta manera, las operaciones de movimiento de piezas dentro del tablero se realizaron utilizando solo una instrucción de máquina (un solo bit shift).

Las clases que fueron utilizadas fueron para representar el juego fueron:

- **Bitboard**: ésta clase representará el tablero, contiene los 3 bitsets con la información del tablero, además de contar con todas las funciones para modificarlo, hacer consultas y operaciones en su estado. Además cuenta con una multitud de constantes, las cuales representan valores como la posición de las esquinas o las posiciones iniciales del juego.
- **Game**: éste tendrá el juego actual, poseerá una instancia de Bitboard del tablero actual, sera en donde ocurre el bucle del juego en donde se llama a toda la lógica del juego como las funciones de búsqueda adversaria o se procesa el input del jugador.
- **Move**: ésta clase es simple y su función es la creación de movimientos que se aplicarán a la instancia de bitboard.

Para la generación de movidas legales en un turno, se revisó pieza del color actual y en sus cuatro direcciones se reviso bit por bit si es que estaba vacía la casilla. Para almacenar estos movimientos se usaron stacks de Bitboards, cada bitboard simbolizaba el tablero que resultaba al aplicar dicha movida.

La función de evaluación del tablero, utiliza 3 aspectos fundamentales, evaluación del rey (movilidad, peligro y su posición), el peligro de cada pieza y el material de la partida.

3 Algoritmos de búsqueda usados

3.1 NegaMax

Para su implementación se uso un *template* de pseudocódigo y para que funcionase con nuestra lógica, se creó un *struct* que contendría el valor máximo y el movimiento por hacer. Luego se la búsqueda de movimientos legales dentro de Negamax.

Sin embargo, una vez implementado notamos que una de la función que obtenía todos los movimientos legales a usar por el agente, tenía una fuga de memoria.

4 Conclusiones

Con la implementación se pudo testificar que el uso de bits para juegos de mesas es lo óptimo al igual que lograr una abstracción buena. Las funciones de evaluación tomaron una gran cantidad de tiempo en comparación con la implementación de Negamax, sin embargo, correr los algoritmos para que una partida fuese jugada por inteligencia artificial llevó a una fuga de memoria que no se pudo resolver. Por lo tanto todas las funciones de evaluación y Negamax quedaron en dentro del código fuente pero, sin hacer uso de ellos; la versión de entrega funciona perfectamente en base a entradas a la consola por el usuario.