Universidad Santiago de Chile
Facultad de Ingeniería
Departamento de Ingeniería
Informatica

Paradigmas de Programación
Informe Nº 2: "Connect 4"
«Simplificado en Prolog»

Nombre:
Alfonso Palacios Vergara

Profesor:

Gonzalo Martinez Ramirez

Fecha de entrega:

29 de Noviembre de 2024

Índice

1. Introducción

Connect 4 es un popular juego de estrategia y habilidad creado en 1974 por Howard Wexler y Ned Strongin, el cual pasaría a ser propiedad de la empresa Hasbro en 1984 y hasta la actualidad. El juego consiste en la confrontación entre dos jugadores utilizando un tablero vertical de 6 filas por 7 columnas. El objetivo es ser el primero en alinear cuatro fichas del mismo color, ya sea horizontal, vertical o diagonalmente.
Cada jugador elige un color de ficha y, en turnos alternos, deben dejar caer una ficha en una de las columnas del tablero. Las fichas caen hasta el fondo de la columna elegida, lo que añade un elemento de estrategia, ya que cada movimiento afecta tanto a la posición del jugador como a la del oponente. Por esta razón, ha sido utilizado en entornos educativos para desarrollar habilidades de resolución de problemas y pensamiento lógico.
Es este contexto en el cual se encuentra el contenido del presente informe, el cual tiene como objetivo presentar y detallar una implementación en lenguaje Prolog del juego Connect 4 desarrollado dentro de la idea del paradigma lógico, el cual se basa en la lógica matemática y en la lógica de predicados para su funcionamiento.
El presente informe está dividido en 3 secciones: Introducción, Desarrollo y Conclusión. En la sección de Introducción se realizará la descripción del problema, en la cual se describirá el contexto del problema a resolver, y la descripción del paradigma, en la cual se caracterizará el paradigma utilizado en la realización de la solución. Por su parte, en la sección de Desarrollo se analizará el problema presentado y los requerimientos funcionales que presentó, se explicará el diseño de la solución realizada con respecto a estos mismos requerimientos, se detallarán aspectos de la implementación realizada, se presentarán las instrucciones de uso del programa y, por último, se expondrán los resultados de la experiencia. Finalmente, en la sección de Conclusión se procederá a sintetizar y concluir respecto al paradigma y el problema resuelto.

1.1. Descripción del problema

El problema abordado en este trabajo consiste en diseñar e implementar un juego de Connect 4 (o Conecta 4) de manera digital de tal modo que esté representado (por lo menos) cada juego con sus jugadores (cada uno con su nombre, color de fichas, estadísticas y fichas restantes), su tablero y su historial de juegos.
En el juego se debe permitir crear y jugar los movimientos correspondientes por cada juego, realizar la verificación de victoria, derrota o empate en el tablero y actualizar las estadísticas e información de cada jugador luego de cada movimiento.
Existen ciertas limitaciones con el problema, principalmente debido a la exigencia de desarrollar el problema utilizando el paradigma lógico. Este paradigma tiene ciertas características que presentan desafíos al momento de estructurar y resolver el problema, como lo son su bajo nivel de intuitividad, el bajo rendimiento del lenguaje del paradigma en cuestiones de memoria y tiempo (debido a los fundamentos del mismo paradigma), el gran nivel de escalabilidad que el paradigma suscita, etc.

1.2. Descripción del paradigma

El paradigma en el cual la experiencia está sustentada es el **paradigma lógico**. El paradigma lógico tiene su origen en la matemática y en la filosofía, especialmente en la lógica de predicados desarrollada por Gottlob Frege en sus obras '*Grundlagen der Arithmetik*' [2] y '*Begriffsschrift*' [3] y el filósofo y matemático Bertrand Russell, con '*Principia Mathematica*' [4] y su desarrollo del atomismo lógico. Estos conceptos serían tomados por Alain Coulmeauer y Philippe Roussel en la creación del lenguaje Prolog, en 1987, el primer lenguaje en implementar el paradigma lógico.
La lógica aplicada, **lógica de predicados**, se centra en el uso de predicados y cláusulas. Las **cláusulas** (cláusulas de Horn) son sentencias que expresan un hecho o **afirmación lógica**. Las mismas son formadas por un nombre (que identifique la relación entre sus argumentos) e individuos (o argumentos) y pueden estar formadas por condiciones, que son a su vez formadas por condiciones atómicas (condiciones irreductibles) que nombran relaciones entre elementos. Un **predicado**, por otra parte, es una relación lógica que se compone de un nombre y una lista de individuos (o argumentos) y representa reglas o consultas, así como también puede representar hechos. Esta relación lógica, a diferencia de una cláusula, **no es necesariamente un hecho**, lo que significa que puede no ser verdad (*Kowalski, R. (1974). Logic for problem solving*) [1].
Por otra parte, el paradigma lógico integra una técnica llamada **inferencia lógica**, según la cual se infiere la veracidad de ciertas afirmaciones basadas en hechos y reglas. Este proceso se basa en la **resolución lógica**, que permite derivar conclusiones a partir de un conjunto de premisas, así como en la **unificación** con la cual se busca encontrar una correspondencia entre dos expresiones, y el **backtracking**, que permite explorar múltiples soluciones en busca de la correcta (*Lloyd, J. W. (2012). Foundations of logic programming*) [5].

2. Desarrollo

2.1. Análisis del problema

El problema, como se mencionó anteriormente, reside en realizar un programa que pueda emular el juego Connect 4 en lenguaje Prolog. Con esto en mente, son presentados 17 requerimientos funcionales (ver Cuadro 2) que, en conjunto, permiten la correcta implementación de un juego de Connect 4. Cada uno de ellos se centra en realizar una función específica, necesaria para el correcto funcionamiento del programa.
Los RFs constructores (RF01, RF02, RF03 y RF10) corresponden a predicados que deben permitir crear un tipo de dato abstracto (player, piece, board y game respectivamente).
Los selectores, por su parte, permiten seleccionar ciertos elementos del tipo de dato abstracto en cuestión. RFs selectores hay 2: RF14, que requiere permitir la selección del jugador cuyo turno está en curso, y RF15, que exige la selección del estado actual del tablero (ambos aplicados al tipo de dato abstracto game).
Los modificadores, de los cuales hay 3, realizan modificaciones a estos datos abstractos. RF05, RF16 y RF17 son modificadores: el predicado de RF05 consiste en modificar un tablero (board) para jugar una ficha, RF16 especifica que se debe implementar un predicado que actualice el juego (game) y las estadísticas de sus jugadores (players) cuando el mismo

termine, y RF17 se refiere a un predicado que permita realizar un movimiento en el juego (game), modificando el tablero (board) y los jugadores (players) según corresponda.
Por último, aquellos RFs que no encajan en los otros grupos se encargan de realizar diversas operaciones: RF06, RF07 y RF08 requieren la implementación de predicados que verifiquen a un ganador vertical, horizontal y diagonal (en ese orden) en un tablero (board), mientras que RF09 requiere la verificación general de un ganador en un tablero (board). Por otra parte, RF11 pide el historial de movimientos de un juego (game), RF12 exige la verificación de si hay un empate en el juego (game), y RF13 solicita la actualización de las estadísticas de un jugador (player) en un juego (game).

2.2. Diseño de la solución

Dados los requisitos funcionales que se deben cubrir, lo primero que se debe realizar es identificar los TDA (Tipos de Datos Abstractos) necesarios. Se identificaron 3 elementos fundamentales para todo juego de Connect 4, expuestos en el Cuadro 1. La estructura sigue una dependencia jerárquica: cada juego debe poseer 1 tablero y 2 jugadores.
Existen 4 tipos de métodos para los TDA: constructor, que permite crear un tipo de dato abstracto teniendo en consideración los requisitos y tipos de datos que se requieren; modificadores, que realizan modificaciones a la cantidad o el contenido de un tipo de dato abstracto; selectores, que buscan y retornan información específica dentro de diferentes tipos de datos abstractos; y otros, que se encargan de realizar operaciones que no encajan en alguna de las categorías anteriormente mencionadas. Todos los métodos constructores, al ser RFs, están presentes en el archivo '*main*' en lugar de en sus respectivos archivos de TDAs
El enfoque principal de la solución es realizar predicados que permitan gestionar los datos a través de combinaciones de listas o pares. Se separaron las estructuras generales en función de lo mostrado en el Cuadro 1 en sus propios archivos, agregando dentro sus predicados constructores, modificadores y de pertenencia. Tener las estructuras separadas de esta forma permite obtener una organización modular del código.
Para la solución se decidió no aprovechar el uso del backtracking natural de Prolog, debido a 2 razones: la primera, porque en todos los casos sólo nos interesa 1 de los posibles casos que pueden tomar los predicados, y la segunda, ya que suscitaba un uso demasiado intensivo de la memoria. Se hizo uso de la técnica de recursión para moverse a través de las listas (en los RFs: RF04, RF05, RF06, RF07, RF08, RF09, RF12, RF16 y RF17) y se aprovechó la inferencia de Prolog para aplicar unificación a los argumentos de cada predicado (en los selectores). A su vez, como es característico en el uso del paradigma, se utilizó la característica del polimorfismo de predicados de Prolog para manejar los casos '*if/else*', ya que en Prolog dos predicados pueden tener el mismo nombre si poseen distintas metas secundarias o diferente cantidad de argumentos.

2.3. Aspectos de implementación

Teniendo en cuenta lo anterior, las implementaciones de los RF mencionados en el Cuadro 2 se encuentran todas en un único archivo '*main*', mientras que cada TDA posee su archivo independiente organizado en predicados constructores, modificadores, selectores y otros.
Se utilizó el editor e interpretador online de Prolog de la página SWISH – SWI - Prolog

(*https://swish.swi-prolog.org/*) para la realización del código y las consultas de la base de conocimientos anterior a la separación en archivos distintos para los TDAs, y se hizo uso de la versión 9.2.8 de Prolog para las modificaciones posteriores. A su vez, se utilizó un notebook HP Pavillon x360 con Windows 10 para la totalidad del proceso de programación.

2.4. Instrucciones de uso

Para utilizar los archivos se deben tener todos dentro del mismo directorio, ya que se requieren entre ellos (en específico, el archivo 'main' requiere de todos los archivos de TDA y los archivos de prueba requieren del archivo 'main'). Esto implica que ciertos predicados que están en un archivo son utilizados en otro (en especial de TDA a main).
Se debe presionar ctrl+C y ctrl+B en el archivo que se quiera ejecutar para abrir el menú de ejecución y realizar consultas. Alternativamente, se puede ir a '*File* → *Edit ...*' desde la terminal de Prolog para abrir el explorador de archivos y seleccionar el archivo a ejecutar.
Los archivos de prueba permiten visualizar la ejecución de cada predicado y verificar los procedimientos de los mismos teniendo en cuenta sus dominios. Los mismos importan todas las funciones del archivo/módulo '*main*' (que contiene los RFs) y ejecutan diferentes procedimientos que sirven para probar estos mismos. Se recomienda revisar la documentación para cada uno, pues todos poseen un dominio específico con parámetros de entrada particulares. Los archivos que no son los scripts de prueba se encuentran listos para ser corridos, interpretados y sus predicados consultados, mientras que los archivos de prueba deben ser llevados a las consultas del archivo '*main*'. Asimismo, se recomienda tener conocimiento específicamente de los constructores, para tener clara la implementación utilizada para cada TDA.

2.5. Resultados y autoevaluación

En general, se pudo implementar cada RF de manera efectiva, complementándose de buena manera con los TDAs y permitiendo representar fielmente un juego de Connect 4. Se realizaron pruebas en todos los casos borde, sólo retornando algo no esperado cuando se introducía a la función algo fuera de su dominio, lo cual es aceptable, siendo que cada función se especializa en un tipo de operación y nunca serán llamadas con algo indebido (y de todas formas es responsabilidad del usuario velar por que esto no pase). El resultado de la autoevaluación tiene puntaje máximo en todos los ítems excepto en aquellos que sé que pude haber implementado mejor, dado más tiempo y madurez con el funcionamiento de Prolog.

3. Conclusiones

Los alcances de la implementación se consideran aceptables para una buena representación digital en Prolog de Connect 4. Así mismo, se da por concluido el objetivo del informe.
El paradigma posee el beneficio de ofrecer listas heterogéneas, así como el de la inferencia automática mediante unificación y backtracking. Así mismo, la lógica declarativa también permite reutilizar código de manera sencilla, ya que se puede definir hechos y reglas que se pueden aplicar a diferentes problemas o contextos. En general, el paradigma es muy útil para proyectos que impliquen bases de datos deductivas y en tareas donde es necesario realizar

inferencias sobre los datos.
Sin embargo, Prolog es un lenguaje muy poco eficiente en cuestión de memoria y tiempo de ejecución en muchos casos, debido al uso exhaustivo de búsquedas muchas veces innecesarias a través del backtracking. A su vez, la representación de datos complejos es más desafiante en Prolog, como aquellos que trabajan con el manejo de grandes cantidades de datos en tiempo real o estructuras de datos mutables, ya que proyectos grandes con muchas reglas y hechos pueden volverse difíciles de manejar. Además, los programas lógicos dependen de la eficiencia del motor de inferencia para su correcta y efectiva ejecución.
Todas estas características afectaron de gran manera a la implementación de la solución, puesto que la inferencia significó que realizar selectores fuera mucho más sencillo, mientras que el tiempo de ejecución del motor de inferencia dejó mucho que desear la mayoría de las veces, por nombrar algunos ejemplos.

Referencias

[1] Kowalski, R. (1974). *Logic for problem solving*. Imperial College of Science and Technology, University of London. https://www.researchgate.net/publication/277669992_Logic_for_Problem_Solving (Accessed: 2024-11-29)

[2] Frege, G. (2013). *Gottlob Frege: Basic laws of arithmetic* (Vol. 1). OUP Oxford. (Accessed: 2024-29-11)

[3] Frege, G. (1879). *Begriffsschrift*. Harvard University. (Accessed : 2024-11-29)

[4] Whitehead, A. Russell, B. (1910) *Principia Mathematica*. (Accessed : 2024-11-29)

[5] Lloyd, J. W. (2012). *Foundations of logic programming*. Springer Science & Business Media. (Accessed: 2024-11-29)

4. Anexos

Elemento	**Descripción**
Game	Juego de Connect 4 compuesto por 2 jugadores, un tablero y un historial de juego
Board	Tablero de juego de Connect 4 compuesto por filas y columnas, con 42 espacios totales
Player	Jugador de un juego de Connect 4 compuesto por su id, nombre, el color de sus piezas, cantidad de victorias, derrotas y empates, y piezas restantes

Cuadro 1: Elementos principales Connect 4

RF	Descripción
1	TDA Player - constructor. Predicado que permite crear un jugador.
2	TDA Piece - constructor. Predicado que crea una ficha de Conecta4.
3	TDA Board - constructor. Predicado que permite crear un tablero de Conecta4.
4	TDA Board - otros - sePuedeJugar?. Predicado que permite verificar si se puede realizar más jugadas en el tablero.
5	TDA Board - modificador - jugar ficha. Predicado que permite jugar una ficha en el tablero
6	TDA Board - otros - verificar victoria vertical. Predicado que permite verificar el estado actual del tablero y entregar el posible ganador vertical.
7	TDA Board - otros - verificar victoria horizontal. Predicado que permite verificar el estado actual del tablero y entregar el posible ganador horizontal.
8	TDA Board - otros - verificar victoria diagonal. Predicado que permite verificar el estado actual del tablero y entregar el posible ganador diagonal.
9	TDA Board - otros - entregarGanador. Predicado que permite verificar el estado actual del tablero y entregar el posible ganador.
10	TDA Game - constructor. Predicado que permite crear una nueva partida
11	TDA Game - otros - history. Predicado que genera un historial de movimientos de la partida.
12	TDA Game - otros - esEmpate?. Predicado que verifica si el estado actual del juego es empate
13	TDA Player - otros - actualizarEstadisticas. Predicado que actualiza las estadísticas del jugador, ya sea victoria, derrotas o empates.
14	TDA Game - selector - getCurrentPlayer. Predicado que obtiene el jugador cuyo turno está en curso.
15	TDA Game - selector - board-get-state. Predicado que entrega el estado actual del tablero en el juego.
16	TDA Game - modificador - game-set-end. Predicado finaliza el juego actualizando las estadísticas de los jugadores según el resultado.
17	TDA Game - modificador - realizarMovimiento. Función que realiza un movimiento.

Cuadro 2: Requerimientos funcionales