Universidad de Santiago de Chile
Facultad de Ingeniería
Departamento de Ingeniería en Informática

Proyecto Semestral de Laboratorio: Dobble en Prolog

Profesor Encargado: Victor Flores
Alumno: Nicolás Rojas
Rut: 20.058.348-5

1. Introducción

En el siguiente informe se trata la problemática de crear un juego de cartas llamado Dobble dentro del paradigma lógico utilizando el lenguaje de programación Prolog. Este proyecto busca aumentar el conocimiento sobre los alcances que tienen los paradigmas de la programación, si facilitan o dificultan la problemática, si las herramientas a utilizar permiten una mejor implementación y conocer un poco sobre la matemática detrás de Dobble.
Se comienza con una descripción del problema, en el que se da a conocer las reglas del juego Dobble, cómo se juega y las características básicas para desarrollar en la problemática. Continua con una descripción del paradigma, en la que se introducen los conceptos más importantes de la programación lógica en Prolog. Se analiza el problema en el que se muestran todos los requerimientos necesarios para implementar una solución. Se describe y se presenta un modelo de solución para crear partidas de Dobble con usuarios por turnos. Se presenta la estructura del proyecto junto con su implementación bajo el lenguaje de programación solicitado. Se agregan las instrucciones de uso para manejar el programa y finalmente comprobar junto con los ejemplos entregados para concluir el uso potencial del paradigma de programación dentro de una problemática como esta.

2. Desarrollo

2.1 Descripción del problema:

Dobble es un juego de cartas con un número determinado de símbolos, las cuales entre si solo pueden tener un único elemento en común. Este juego tiene una variedad de minijuegos, pero para este proyecto solo se contempla una versión modificada del minijuego “Torre Infernal”. Para jugarlo se deben colocar 2 cartas al azar sobre la mesa a la vista de todos los jugadores. Los jugadores deben reconocer el símbolo igual que hay entre las 2 cartas. El primer jugador que las reconozca robará 2 nuevas cartas aleatorias y ganará 2 puntos. Finalmente se acaba la partida cuando los jugadores lo deciden, y el ganador será el que tenga la máxima cantidad de puntos.
La complejidad de la creación de Dobble es el algoritmo para la generación de cartas con un único elemento en común, ya que esto implica tener conocimiento en propiedades matemáticas como los planos proyectivos finitos, plano de fano, geometría, entre otras cosas. Para esto se utiliza la documentación de un algoritmo en JavaScript proporcionado por Micky Dore[1], el cual explica a través de listas, matrices e iteraciones la forma de generar cartas según una cantidad de elementos dados como símbolos en una variable.

2.2 Descripción del Paradigma:

El paradigma de programación utilizado en este proyecto es el lógico, también llamado como programación predicativa, el cual se basa en la lógica matemática. En lugar de una sucesión de instrucciones, este paradigma contiene una recopilación de hechos y suposiciones.
Un lenguaje lógico como Prolog permite contestar preguntas con True o False. Para que el lenguaje sea capaz de responder tiene que haber una serie de hechos y relaciones, los cuales se pueden utilizar para realizar procesos de inferencia.
Prolog usa 3 mecanismos de inferencia:

1) Unificación: Es un proceso que consiste en encontrar una asignación de variables que haga idénticas a las fórmulas que se desea unificar.
2) Backtracking: Recuerda los momentos de la ejecución donde un objetivo tenía varias soluciones para posteriormente dar marcha atrás y seguir la ejecución usando otra solución como alternativa.
3) Árboles SLD: Árbol el cual sus ramas son derivaciones SLD[2] utilizando unificación y backtracking.

[1] Dore, M. (2021, 30 diciembre). The Dobble Algorithm - Micky Dore. https://mickydore.medium.com/the-dobble-algorithm-b9c9018afc52
[2] Las siglas SLD vienen de reglas de Selección, resolución Lineal y cláusulas Definidas.

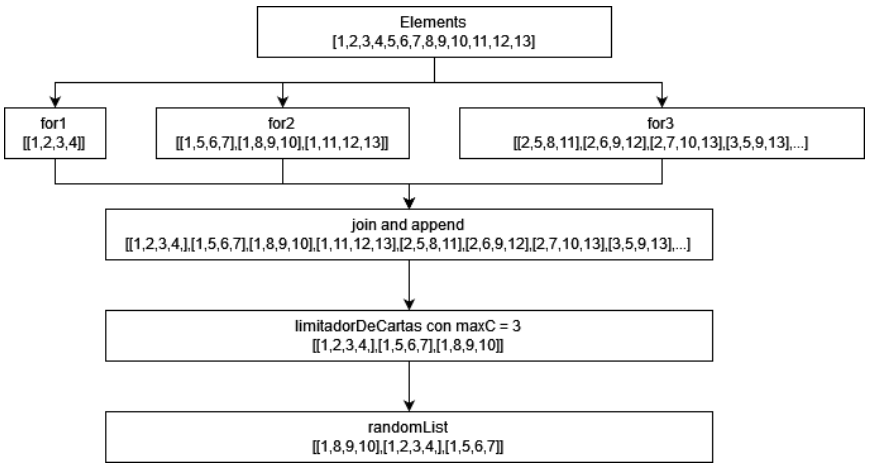
2.3 Análisis del Problema:

1) TDAs: Teniendo en cuenta que Dobble es un juego de cartas, hay que pensar en una estructura que nos permita almacenar los valores de las cartas, es decir, los elementos que se utilizarán como símbolos, y el mazo final. Además del mazo de cartas, se necesita una mesa en la cual dejar el mazo, en la cual estarán los jugadores, sus puntuaciones, el modo juego, etc.
   Con lo anterior pensado, crearemos un TDA Cards para las cartas y un TDA Game para la partida con sus respectivos selectores, modificadores y constructores en casos necesarios.
2) cardsSetIsDobble: Algo que define a Dobble es su mazo con cartas con un solo elemento en común, por lo que se debe crear un predicado que compruebe que un mazo es un conjunto válido para jugar.
3) cardsSetNthCard: Vamos a necesitar un predicado que nos entregue una carta en una determinada posición, desde 0 hasta el número total de cartas generadas menos 1.
4) cardsSetFindTotalCards: Para saber la cantidad de cartas que un conjunto debe tener, se tiene que crear un predicado que a partir de una carta de muestra entregue el número máximo de cartas a generar
5) cardsSetMissingCards: En el caso que se limite la cantidad de cartas a generar, se crea un predicado que sea capaz de generar el resto de cartas necesarias sin repetir para un mazo válido.
6) cardsSetToString: Una vez el mazo esté completo y listo para ser usado, se crea un predicado capaz de convertir todas las cartas en formato de texto para luego ser mostrado en pantalla.
7) dobbleGame: Como se había dicho anteriormente, se necesita un constructor para el TDA Game, en el cual se puedan contener la cantidad de jugadores, el nombre de los jugadores, el mazo, el modo de juego, la mesa, el estado del juego, el turno de los jugadores, y un número que sirva para sacar cartas aleatoriamente.
8) dobbleGameRegister: Los jugadores que vayan a jugar se tienen que identificar con un nombre único en la partida, por eso se crea un predicado que añade jugadores siempre y cuando no se haya registrado antes o supere el límite de usuarios por partida.
9) dobbleGameWhoseTurnIsIt: Dado por las limitaciones del paradigma, el juego debe ser en turnos, por lo que debe existir un predicado que diga a qué jugador le toca jugar.
10) dobbleGamePlay: Cuando ya esté todo establecido en la partida, los jugadores tienen que comenzar a jugar, por lo que este predicado permitirá dar vuelta 2 cartas aleatorias para que el jugador que tiene el turno pueda entregar un elemento para comparar entre ambas cartas, en la cual si acierta, ganará 2 puntos. También se tiene que poder pasar de turno y finalizar el juego.
11) dobbleGameStatus: Este predicado se necesita para saber el estado del juego, ya sea en partida o terminado.

12) dobbleGameScore: Los jugadores tienen que saber sus puntajes, por lo que con el nombre de cada jugador registrado se debe poder preguntar a un predicado qué puntaje tiene el jugador seleccionado. Con esto se podrá determinar un ganador, empate o perdedor.
13) dobbleGameToString: Finalmente, al igual que las cartas, la partida también necesita un predicado que sea capaz de traspasar toda la información a texto para luego ser mostrada en pantalla.

2.4 Diseño de la solución:

El primer desafío es escribir el código que permite la generación de las cartas. Lo realizado para el proyecto consta de predicados con llamadas recursivas que irán generando las cartas necesarias para el mazo, donde for1 representa las cartas de orden 1, for2 las cartas de orden n y for3 las cartas de orden n cuadrado.



Cuando se habla de cartas sabemos que tenemos que agruparlas en un lugar. Ese lugar será el mazo, el cual es representado como una lista con listas de elementos.
El TDA Cards es una lista que contiene los elementos que se utilizaron para hacer el mazo, con la finalidad de poder usarlo más adelante en caso de que se quieran generar las cartas restantes con el predicado cardsSetMissingCards. El TDA también contiene el mazo de cartas que se utilizará dentro de la partida.
La gran mayoría de predicados utilizados para conseguir una carta, verificar los elementos, encontrar la máxima cantidad de cartas que puede tener un mazo, la aleatorización de las cartas y demás son puramente predicados relacionados con la modificación de listas.
La regla principal del juego, la cual consiste en encontrar el elemento en común que tienen las cartas se puede programar cómodamente gracias al uso del backtracking de Prolog.
Un ejemplo de TDA Cards es el de la siguiente imagen:

Elements	Cards
[1,2,3,4,5,6,7,...]	[[1,2,3,4],[1,5,6,7],[1,8,9,10]]

En donde se muestra que los elementos son una lista del 1 al 13 y existen 3 cartas en el mazo.

El TDA Game contiene una lista con el número de jugadores, el mazo de cartas generado, el nombre del modo, el número de aleatorización, la lista de jugadores, el turno de la persona, la puntuación de los jugadores, el estado de la partida y las cartas de la mesa. Cuando genera la lista de jugadores, esta se va rellenando a medida que se van ingresando con el predicado dobbleGameRegister, sin embargo, la lista de puntuaciones se rellena automáticamente con ceros. En el momento en que se agregan los jugadores, a partir de las operaciones en el predicado dobbleGamePlay se irán agregando los puntos según las posiciones de los jugadores.
Un ejemplo de tdaGame es el de la siguiente imagen:

NP	CS	M	R	PL	T	PO	S	Me
3	[[1,2,3,4],[1,5,6,7],[1,8,9,10]]	"SC2"	1234	["Zerg","Protos","Terran"]	"Zerg"	[4,2,0]	"En Partida"	[[1,2,3,4],[1,5,6,7]]

En este ejemplo se puede apreciar que el jugador Zerg hasta ahora tiene ventaja por sobre los demás, ya que el juego sigue en partida.

2.5 Aspectos de implementación:

Para la implementación de este proyecto usamos SWI-Prolog 8.4.2 perteneciente a la programación lógica. Todos los predicados y TDAs basados en listas se encuentran dentro del mismo archivo llamado “labDobble_20058348_RojasGonzalez.pl”, el cual tiene las implementaciones y ejemplos separados por comentarios.
Usamos el algoritmo de Micky Dore como referencia para crear nuestra versión en Prolog mediante llamadas recursivas y unificación. Una vez el mazo de cartas está creado, se utiliza una función de aleatorización con números pseudoaleatorios (seed) que entrega un número muy grande que va disminuyendo considerablemente para simular la aleatorización a la hora de mezclar las cartas o distribuirlas por la mesa.
Debido a que las limitaciones del paradigma no permite que haya varios jugadores simultáneamente, se ha establecido que el juego sea a través de turnos, por lo que los jugadores deberán escribir las instrucciones como se muestra en los ejemplos de uso. En cuanto a la generación de cartas, se asume que la lista de elementos que se le entrega al programa es lo suficientemente larga como para generar una cantidad máxima de cartas necesarias para la partida especificada. Otro supuesto para este proyecto es que el juego utilizará el mismo mazo hasta que los jugadores decidan finalizar el juego, es decir, que cuando se colocan las 2 cartas sobre la mesa, estas vuelven al mazo y pueden volver a aparecer en la mesa.

2.6 Instrucciones de uso:

Para el correcto uso de este programa debe seguir cuidadosamente las siguientes reglas:

1) Lo primero que se debe hacer es la generación de las cartas, por lo que el usuario deberá llamar al predicado cardsSet(Elements,NumE,MaxC,Seed,CS). Donde Elements debe ser una lista lo suficientemente larga como para generar una gran cantidad de cartas con respecto al número de elementos NumE que se le solicitará al programa. El usuario puede limitar la cantidad de cartas que quiere que se genere en MaxC, pero en caso de que no se quiera limitar, basta con colocar una variable, y finalmente se recomienda usar un número superior a 1000 para la Seed, de esta manera el sistema de aleatorización podrá tener más posibilidades de aleatoriedad.
   Un ejemplo para esto puede ser cardsSet([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z], 3, 4, 92175, CS). Donde Elements es una lista con las letras del abecedario, 3 es la cantidad de elementos que debe tener una carta, 4 son las cartas máximas a generar y 92175 es la seed de aleatorización.
2) Una vez las cartas estén generadas, se tiene que crear la partida usando el predicado dobbleGame(NumPlayers,CardsSet,Mode,Seed,Game). Donde NumPlayers es la cantidad de usuarios a registrar, CardsSet es el mazo de cartas creado anteriormente, Mode es el modo de juego (que para este proyecto es irrelevante, ya que solo tiene 1), y un número que se recomienda que sea superior a 1000 para la aleatorización de las cartas a colocar sobre la mesa.
   Un ejemplo de esto puede ser dobbleGame(2,CS,"modoX",4222221,G). Donde se da a conocer que se van a registrar 2 usuarios, se le pasa el CardsSet ya generado, se entrega el modo de juego “modoX” y el número superior a 1000.
3) Para registrar usuarios usamos dobbleGameRegister(User,GameIn,GameOut). Donde User es el nombre del usuario y GameIn la partida creada.
   Un ejemplo es dobbleGameRegister("Comodín",G,G2), donde “Comodín” es el jugador 1, pero hay que tener en cuenta que si el nombre “Comodín” ya está registrado, no se puede volver a registrar. También cabe decir que si se añaden más jugadores de lo que se especificó en la partida, el programa dirá false.
4) Cuando ya estén los usuarios registrados, se usará dobbleGamePlay(G2,null,G3), para dar vuelta 2 cartas sobre la mesa. Luego se usa el predicado dobbleGameWhoseTurnIsIt(G3,Username) para saber a qué jugador le toca jugar. Una vez se sepa quién debe jugar, se usa dobbleGamePlay(G3,[spotit,Username,Elemento],G4), donde Username debe ser el nombre del jugador y Elemento el símbolo a comparar en ambas cartas. Por ejemplo: dobbleGamePlay(G3,[spotit,"Comodín",a],G5). En caso de querer pasar al siguiente turno se usa dobbleGamePlay(G5,[pass],G6) y para finalizar la partida dobbleGamePlay(G6,[finish],G7).
   Hay que tener cuidado con colocar mal al usuario que tiene el turno, ya que si un jugador juega sin tener el turno, el programa dirá false.

2.7 Resultados:

En este proyecto se pudo implementar todo lo necesario para que el juego funcione, siempre y cuando se sigan las instrucciones correctamente. El programa funciona bien para cualquier lista de elementos, ya sean símbolos, números o texto. Se abordaron todos los requerimientos satisfactoriamente, usando al menos 39 ejemplos en los cuales todos salieron positivos.
La siguiente tabla de autoevaluación funciona de la siguiente manera:
a) 0: No realizado
b) 0.25: Funciona a lo más el 25% de las veces.
c) 0.5: Funciona el 50% de las veces.
d) 0.75: Funciona el 75% de las veces.
e) 1: Funciona el 100% de las veces.

Predicado	Puntaje
cardsSet	1
cardsSetIsDobble	1
cardsSetNthCard	1
cardsSetFindTotalCards	1
cardsSetMissingCards	1
cardsSetToString	1
dobbleGame	1
dobbleGameRegister	1
dobbleGameWhoseTurnIsIt	1
dobbleGamePlay	1
dobbleGameStatus	1
dobbleGameScore	1
dobbleGameToString	1

3. Conclusión

Se han podido cumplir todos los objetivos que requería el proyecto. La mayor complicación que hubo fue pensar en cómo construir el algoritmo presentado en JavaScript a un lenguaje de programación lógico como Prolog. Ya se había tenido ese problema antes en el paradigma funcional, por lo que la experiencia anterior ayudó a cambiar la forma de pensar en esta ocasión, permitiendo ver los algoritmos de otra manera en la que se pudieran implementar. El uso de predicados junto con las llamadas recursivas hicieron que el desarrollo del proyecto fuese mucho más ameno que usando Scheme. Las limitaciones del paradigma incluso ayudaron a que la implementación del juego en base a turnos fuese más ordenada y sencilla. El uso de listas y la construcción de TDAs permitieron una correcta implementación en las cartas y la partida.

En este proyecto se pudo aprender el potencial del paradigma lógico, la resolución de problemas, el uso del backtracking para la comparación de elementos en listas, la unificación entre predicados para hacer preguntas y las llamadas recursivas dan una gran referencia de todo lo que se puede hacer con este lenguaje. Es más, incluso luego de ver las limitaciones que tiene el paradigma en sí, permite trabajar con lo que vendrían a ser variables de una manera mucho más cómoda que en el paradigma funcional.

4. Referencias

1) Dore, M. (2021, 30 diciembre). *The Dobble Algorithm - Micky Dore*. https://mickydore.medium.com/the-dobble-algorithm-b9c9018afc52

2) *Unificación en programación lógica | | UPV*. (2018, 22 octubre). YouTube. https://www.youtube.com/watch?v=Zm5m8defbbI

3) *Dobble Kids cómo jugar*. (2019, 18 agosto). YouTube. https://www.youtube.com/watch?v=7XTXe7FqAM8

4) *How to make the card game Dobble (and the Maths behind it!)*. (2021, 8 abril). YouTube. https://www.youtube.com/watch?v=oyqD1Sg5M4Q