

Informe de Laboratorio

2: Implementación de juego «Dobble» en Prolog



PARADIGMAS DE PROGRAMACIÓN

08/05/2022



Índice general

1	Introducción	3
1.1	Descripción del Problema	3
1.2	Descripción del Paradigma	3
2	Desarrollo	5
2.1	Análisis del Problema	5
2.2	Diseño de la Solución	6
2.2.1	TDA CardsSet	6
2.2.2	TDA GAME	7
2.3	Aspectos de Implementación	8
2.4	Instrucciones de uso:	9
2.4.1	Resultados Esperados	9
2.4.2	Posibles errores	9
2.5	Resultados y autoevaluación	10
3	Conclusiones	11
4	Referencias	12
5	Anexos	13
5.1	Anexo 1: Algoritmo generación de cartas	13
5.2	Anexo 2: Tabla de Pruebas y autoevaluación	14

CAPÍTULO Introducción 1

Este informe corresponde al 2º laboratorio de la asignatura *Paradigmas de Programación*. En este primer laboratorio se empleará el Paradigma Lógico, usando el lenguaje de programación **SWI-Prolog**, una implementación de el lenguaje **Prolog**. Para el uso de este, se usará **GNU-Emacs** para la edición de texto y se harán las consultas a través de **SWI-Prolog**.

1.1. Descripción del Problema

Se pide crear una implementación de Dobble. Este juego consiste en un mazo de cartas con 55 cartas (57 originalmente) donde dos cartas cualesquiera comparten **solo** un elemento entre sí. Para jugar Dobble existen distintos tipos de modos. El modo más conocido consiste en que los jugadores deben tomar dos cartas del mazo y el primero que identifique cuál símbolo se encuentre en ambas cartas toma la primera carta y la guarda. Después, se toma otra carta y se repite el proceso hasta que se acabe el mazo. Una vez que se acaben las cartas, el jugador que tenga la mayor cantidad de cartas ganará. Para representar esto en Prolog hay que tener en cuenta que la generación de cartas deberá respetar la condición de que solo se repita el símbolo una vez entre cualquier par de cartas. Además hay que tomar en cuenta que tipo de modo de juego seleccionará el usuario.

1.2. Descripción del Paradigma

Para crear la representación de «Dobble» se usará la Programación Lógica, Un paradigma de la programación con fundamentos en las matemáticas. En este paradigma, los programas tendrán una base de conocimiento, en la que se expresarán predicados, hechos y reglas y estos se podrán consultar. A diferencia de los otros paradigmas, se enfoca en que busquemos responder, en vez de cómo lo podemos responder. Estas consultas solo tendrán dos posibles resultados, **true** o **false**. Las principales características de este paradigma son:

- **Backtracking:** Prolog para responder las consultas hechas por el usuario hace uso de el algoritmo Backtracking, el cual va construyendo las posibles soluciones y las abandona (backtrack) en cuanto estas no puedan llegar a una solución válida.
- **Recursión:** No se hace uso de ciclos **for** o ciclos **while**, y en cambio se hará uso de la recursión, donde se llamará la función una y otra vez hasta que se llegue a un caso base.

- **Unificación:** Es un proceso que Prolog realizara al buscar una variable no instanciada con otra variable, y si es que es posible que haya un valor que las haga idénticas según el predicado, termina tomando el valor de la otra.
- **Inmutabilidad:** Las variables serán inmutables, osea que no se podrán modificar una vez sean creadas.

Estas características tendrán sus ventajas y desventajas. Las ventajas de estas es que en nuestros programas podremos fácilmente armar una base de verdades y poder saber si estas son verdad o falsas simplemente haciendo una consulta. Además, no nos tendremos que fijar específicamente en como llegaremos al resultado Pero tendrán la desventaja que podrían requerir de mas rendimiento en caso de mal utilizar la recursión. Además, usualmente se esta acostumbrado a usar loops para programar, lo cual hará pasar a solamente usar recursión algo difícil. Otra desventaja es que, al trabajar usualmente con lenguajes declarativos, se hace difícil el cambio a este paradigma.



CAPÍTULO Desarrollo 2

2.1. Análisis del Problema

Para este proyecto se tendrá que tener en cuenta los aspectos fundamentales del juego «Dobble». Este juego, como ya ha sido mencionado, tendrá un mazo de cartas con 57 cartas, donde 2 cartas cualesquiera tendrán siempre un elemento en común. Para este hay que tener en cuenta las bases matemáticas del juego, pero en este caso se nos entrega un algoritmo escrito en **Javascript** el cual facilitara la creación del algoritmo en **Prolog**. El mazo creado sera representado como una lista de listas, donde cada lista sera una carta diferente. El usuario podrá pedir diferentes acciones respecto al mazo o a una carta [TDA cardsSet]:

- Verificar si un mazo es valido [dobble?]
- Ver el numero de cartas en el mazo [numCards]
- Buscar que carta hay en x posición [nthCard]
- Encontrar cuantas cartas necesita para un mazo valido a partir de una carta [findTotalCards]
- Encontrar el numero de elementos que necesita para armar un mazo valido a partir de una carta [requiredElemets]
- Ver que cartas faltan de un mazo para armar uno valido [missingCards]
- Transformar el mazo a una representación en strings. [cardsSet→string]

Para permitir al usuario poder jugar el juego hay que tener en cuenta que este podrá [TDA game]:

- Ingresar cantidad de jugadores [register y numPlayers]
- Elegir modo de juego [mode, stackMode, myMode y emptyHandsStackMode]
- Elegir que hacer en su turno [play y pass]
- Terminar el juego cuando quiera
- Ver de quien es el turno y el puntaje [whoseTurnIsIt? y status]

2.2. Diseño de la Solución

Para la solución de este problema, se tienen en cuenta todas las acciones que el usuario podrá tomar, ya sea para armar el mazo o para las acciones que podrá tomar al jugar el juego una vez ya tenga el mazo armado.

2.2.1. TDA CardsSet

Creación de cartas (Constructor cardsSet)

El mazo de cartas creado sera representado como una lista de listas, y cada elemento de la carta podrá ser o un numero, o un símbolo introducido por el usuario. El principal problema sera la creación de cartas, para ello se nos entrega el siguiente algoritmo escrito en Javascript. (Ver Anexo 1)

Debido a la diferencias del lenguaje y requerimientos del proyecto, nuestra solución no podrá hacer uso de variables ni de ciclos `while` o `for`, por lo que se necesitara hacer uso de recursión para poder implementarlo. En el algoritmo se puede observar que este posee ciclos `for` anidados, por lo que en la solución crearemos predicados recursivos que llamen a otros predicados recursivos las cuales generaran el numero requerido para las cartas y los agregaran a una lista. Con los predicados que generaran las cartas creadas, hace falta dejarlas en una sola lista, lo cual se puede lograr con la función de Prolog `append`. Ahora se obtiene un mazo de cartas, pero este esta representando por numero y hay que tener en cuenta que el usuario podrá pedir en la generación de cartas que este se genere con una lista de símbolos ingresada por el usuario (Elements), un numero de elementos en cada carta (numE), un máximo de cartas (maxC) y una función que aleatorizara el ordenamiento de las cartas (rndFn). Para generar el mazo de cartas con los símbolos, se tomara generara el mazo y se remplazara el símbolo que este en la posición `i` y se remplazaran todos los `i` que hayan en el mazo con ese símbolo y se repetirá el proceso hasta que no queden mas símbolos o mas cartas. El numero de elementos sera entregado al algoritmo para que los genere y después se separara la lista según la cantidad de elementos en cada carta. Y para el máximo de cartas generado solo bastara con tomar los maxC elementos de la lista. Y finalmente con todo ya listo, se procede a aleatorizar el ordenamiento de las cartas con la función proporcionada por el enunciado.

Pertenencia

Como predicado de pertenencia tendremos que verificar que el mazo de Dobble entregado por el usuario sea un mazo valido para jugar. Para ello tendremos que verificar que la



cantidad de cartas es la correcta, lo cual se puede verificar con la formula $(n * (n - 1) + 1)$, donde n es el numero de elementos. Además habrá que asegurarnos que los símbolos de las cartas solo se repitan una vez en cada par de cartas.

Selectores

Nuestra predicado selector sera el predicado `cardsSetNthCard`, el cual nos permitirá encontrar la carta n (elegida por el usuario). Como nuestro mazo esta representado por una lista de lista solo basta con hacer uso de `nth0`, un predicado nativo de Prolog.

Modificadores

El predicado modificador en este TDA sera el predicado `cardsSetToString`, el cual nos permitirá representar el mazo (lista de listas) como strings.

Otras Funciones

Se tienen otros predicados como `cardsSetFindTotalCards` y `cardsSetMissingCards`. En el primer predicado bastara con usar el predicado `length` y para el ultimo predicado, se hará una intersección entre el mazo ingresado y un mazo generado (valido), donde el resultado serán las cartas faltantes.

2.2.2. TDA game

El TDA game permitirá al usuario jugar con el set de cartas anteriormente creado. El usuario podrá registrar usuarios y elegir un modo de juego.

Constructor

El predicado constructor para el TDA game consistirá en una predicado que podrá recibir como argumentos un Entero, una lista de listas (Mazo), un string (modo de juego) y una semilla (Entero) para la aleatorización (seed). Este predicado devolverá una lista de listas representando todo lo ingresado.



Selectores

Se tendrán varios predicados selectores, ya que hay que poder obtener de quien es el turno actual (`dobbleGameWhoseTurnIsIt`), obtener el status del juego (`dobbleGameStatus`) y conseguir el puntaje de cada jugador (`dobbleGameScore`).

Modificadores

Se hará uso de este tipo de predicados para poder registrar a los usuarios que irán a jugar el juego (`dobbleGameRegister`) y para poder mostrar el juego en forma de string (`dobbleGameToString`).

Otras Funciones

El usuario podrá accionar en el juego con la función (`dobbleGamePlay`), y podrá elegir si pasar su turno o seleccionar que simbolo se repite en las dos cartas mostradas. El modo de juego sera `stackMode`.

2.3. Aspectos de Implementación

- Estructura del proyecto: El archivo `Dobble20960400villarroelGonzalez.pl` contendr los predicados necesarios *Prolog*, pudiendo hacer uso de los *TD* para armar el mazoy jugar con este. *Bibliotecas utilizadas :* *Para este proyecto solo se hizo uso de la libreria base de Prolog.*
- Interprete utilizado: Se hizo uso de SWI-Prolog v8.4.2
- Razones de elección: Se mantiene los predicados en un mismo archivo debido a que no fue tan extenso como la implementacion de este juego en Racket. Se hizo uso de la biblioteca base debido a que no se requirió tener que hacer uso especifico de otra y el uso de SWI-Prolog fue debido a que el entorno que este tiene esta bien integrado con Prolog y es el interprete por defecto que viene con la instalación.



2.4. Instrucciones de uso:

```
%% CardsSet
cardsSet(["A","B","C","D","E","F","G","H","I"],3,-1,Seed,CS).
%% cardsSetNthCard
cardsSet(["A","B","C","D","E","F","G","H","I"],3,-1,Seed,CS),
cardsSetNthCard(CS,5,CS1).
%% cardsSetFindTotalCards
cardsSet(["A","B","C","D","E","F","G","H","I"],3,-1,Seed,CS),
cardsSetNthCard(CS,5,CS1),
cardsSetFindTotalCards(CS1,CS2).
%% cardsSetMissingCards
cardsSetMissingCards([[1,2,3,4],[7,6,5,1]],CS2).
%% cardsSetToString
cardsSet(["A","B","C","D","E","F","G","H","I"],3,-1,Seed,CS),
cardsSetToString(CS,CS1).
%% dobbbleGame
cardsSet(["A","B","C","D","E","F","G","H","I"],3,-1,Seed,CS),
dobbbleGame(2,CS,"STACK",Seed,G).
%% dobbbleGameRegister
cardsSet(["A","B","C","D","E","F","G","H","I"],3,-1,Seed,CS),
dobbbleGame(2,CS,"STACK",Seed,G),
dobbbleGameRegister("DIINF",G,G1),
dobbbleGameRegister("Prolog",G1,G2).
```

2.4.1. Resultados Esperados

Se espera que el mazo se genere sin errores dado que el conjunto de elementos ingresados sea valido y que se puede generar el juego y registrar a los usuarios que se quieran ingresar.

2.4.2. Posibles errores

Es posible que `cardsSetMissingCards` no genere bien las cartas faltantes si es que se ingresan símbolos, además el predicado `dobbbleGameRegister` podría lanzar un error si es que se superan el numero máximo de usuarios.



2.5. Resultados y autoevaluación

Probando el programa creado con los ejemplos demostrados, se puede ver que los predicados implementados de el TDA cardsSet funcionan correctamente y que el TDA game solamente funciona parcialmente debido a que solamente unas pocos predicado de este TDA fueron implementadas. La autoevaluación (Ver Anexo 1) se evaluara de la siguiente forma:

1. Nombre de la función
2. Tipo de Prueba
3. Razón de fallos
4. Completado?
5. Evaluación

La columna de evaluación sera evaluada según lo siguiente:

1. 0: No realizado.
2. 0.25: Implementación con problemas mayores (funciona 25 % de las veces o no funciona)
3. 0.5: Implementación con funcionamiento irregular (funciona 50 % de las veces)
4. 0.75: Implementación con problemas menores (funciona 75 % de las veces)
5. 1: Implementación completa sin problemas (funciona 100 % de las veces)

El predicado cardsSetMissingCards tiene 0.5 de puntaje debido a que funciona solamente cuando se entrega un mazo donde los elementos sean un número entero. En el TDA game el predicado dobbbleGameRegister tendrá 0.75 de puntaje ya que puede fallar al pasarse de el numero máximo de usuarios especificados. El resto de las funciones del TDA game tienen 0 puntaje debido a que no fueron implementadas.





CAPÍTULO

Conclusiones 3

El Paradigma Lógico en Prolog puede facilitar mucho algunas partes de el programa, pero en otras partes se vuelve algo difícil de pensar, debido a que se debe implementar de una manera completamente diferente a lo que uno acostumbra. Además, debido a que uno se encuentra acostumbrado a la sintaxis de otros lenguajes como `python` y `javascript`, las cuales son mas parecidas al lenguaje humano, además que estos usualmente hacen uso de ciclos `for` o `while` y el cambio de pasar de estos a solamente tener que usar recursión y no usar variables hizo difícil el comienzo del proyecto. A cambio de el Paradigma Funcional en Racket, este paradigma fue mas fácil de entender en un principio, debido a la facilidad de crear hechos y reglas, pero una vez se pone más complejo, se vuelve mas difícil de imaginarse la solución, en cambio en Racket, uno igualmente podía aplicar una lógica parecida a la declarativa. Otra complicación fue que el TDA game no fue completado al 100% debido a complicaciones de tiempo. Aun con estas complicaciones, este proyecto fue de mucha utilidad ya que permitió aprender sobre un diferente paradigma de programación y hacer uso de tecnologías como git.

CAPÍTULO

Referencias 4

1. SWI Prolog Reference Manual. (2012). SWI-Prolog Documentation. Recuperado 2022, de <https://www.swi-prolog.org/pldoc/index.html>
2. Flores, V. (2022). Paradigmas de Programación - Proyecto semestral de laboratorio. Recuperado 2022, de 

3. Dore, M. (2021, 29 diciembre). The maths behind Dobble - Micky Dore. Medium. Recuperado 2022, de <https://mickydore.medium.com/dobble-theory-and-implementation-ff21ddb5318>
4. Dore, M. (2021, diciembre 30). The Dobble Algorithm - Micky Dore. Medium. <https://mickydore.medium.com/the-dobble-algorithm-b9c9018afc52>

5

5.1. Anexo 1: Algoritmo generación de cartas

```

//Generacion Dobble Javascript
//
let i, j, k
let n = 3 //order of plane, must be a prime number
let numOfSymbols = n + 1 //order of plane + 1
let cards = [] //the deck of cards
let card = []; //the current card we are building//to start, we build the first
↪ card
let cards = [];

for (i = 1; i <= n+1; i++) {
    card.push(i)
}
cards.push(card)//then we build the next n number of cards

for (j=1; j<=n; j++) {
    card = []
    card.push(1)

    for (k=1; k<=n; k++) {
        card.push(n * j + (k+1))
    }
    cards.push(card)
} //finally we build the next n^2 number of cards

for (i= 1; i<=n; i++) {
    for (j=1; j<=n; j++) {
        card = []
        card.push(i+1)

        for (k=1; k<= n; k++) {
            card.push(n+2+n*(k-1)+(((i-1)*(k-1)+j-1)%n))
        }
        cards.push(card)
    }
}

```

5.2. Anexo 2: Tabla de Pruebas y autoevaluación

Función	Tipo de prueba	Razón de fallos	Completado?	Evaluación
cardsSet	Generación de cartas con distintos argumentos.	No hubo fallos	Si	1
cardsSetIsDobble	No hubo pruebas	No se logro implementar	No	0
cardsSetNthCard	Se busca la carta con distintos mazos	No hubo fallos	Si	1
cardsSetFindTotalCards	Se busca la carta con distintas cartas	No hubo fallos	Si	1
CardsSetMissingCards	Se busca las cartas faltantes con distintos mazos	Hubo fallos cuando cartas contenían símbolos	No	0.5
cardsSetToString	Se imprimen como strings distintos mazos	No hubo fallos	Si	1
dobbleGame	Se crea con distintos mazos	No hubo errores	Si	1
stackMode	No hubo pruebas	No se logro implementar	No	0
dobbleGameRegister	Se hace prueba con distintos números de usuarios	Error cuando >numPlayers	Si	0.75
DobbleGameWhoseTurn	No hubo pruebas	No se logro implementar	No	0
DobbleGamePlay	No hubo pruebas	No se logro implementar	No	0
DobbleGameStatus	No hubo pruebas	No se logro implementar	No	0
DobbleGameScore	No hubo pruebas	No se logro implementar	No	0
DobbleGameToString	No hubo pruebas	No se logro implementar	No	0

