Memoria de la práctica de PDA

Sergio Ballesteros Navajas
Héctor Alejandro Martos Gómez

Cita:

“The way you learn anything is that something fails, and you figure out how not to have it fail
again”.
— Robert S. Arrighi, Pursuit of Power: NASA's Propulsion Systems Laboratory No. 1 and 2

Introducción

En esta memoria vamos a describir en detalle el desarrollo de la práctica realizada para la asignatura Programación Declarativa Avanzada.

Nuestra elección para la práctica ha sido el juego de las Siete y Media. Las Siete y Media es un juego popular español que consiste en conseguir que la suma del valor de las cartas del jugador sume siete puntos y medio o bien se acerque al máximo a este valor sin pasarse. La banca juega contra cada jugador individualmente y su jugada ha de superar o igualar la de cada uno de ellos. En caso de empate gana la banca.

Nosotros desarrollaremos una versión del juego con un único jugador y una banca que seguirá una heurística de juego.

Para el desarrollo de la práctica, aprovechando las similitudes con otros juegos de cartas como el Poker o el BlackJack, nos hemos inspirado con algunos repositorios de código abierto, así como blogs, páginas de la documentación de Haskell, etc. Todas estas referencias se pueden encontrar en la sección Bibliografía.

Desarrollo

Flujo de juego

Para iniciar el juego se reparte una carta a cada uno de los jugadores (en nuestro caso sólo un jugador) y una carta a la banca. La carta de la banca debe ser visible para los jugadores.

Empieza el turno del jugador, que puede elegir entre pedir carta o plantarse. Si se planta el turno pasa al siguiente jugador (en nuestro caso pasará directamente a la banca). Si el jugador elige pedir carta se le repartirá una nueva carta y pueden suceder tres cosas:

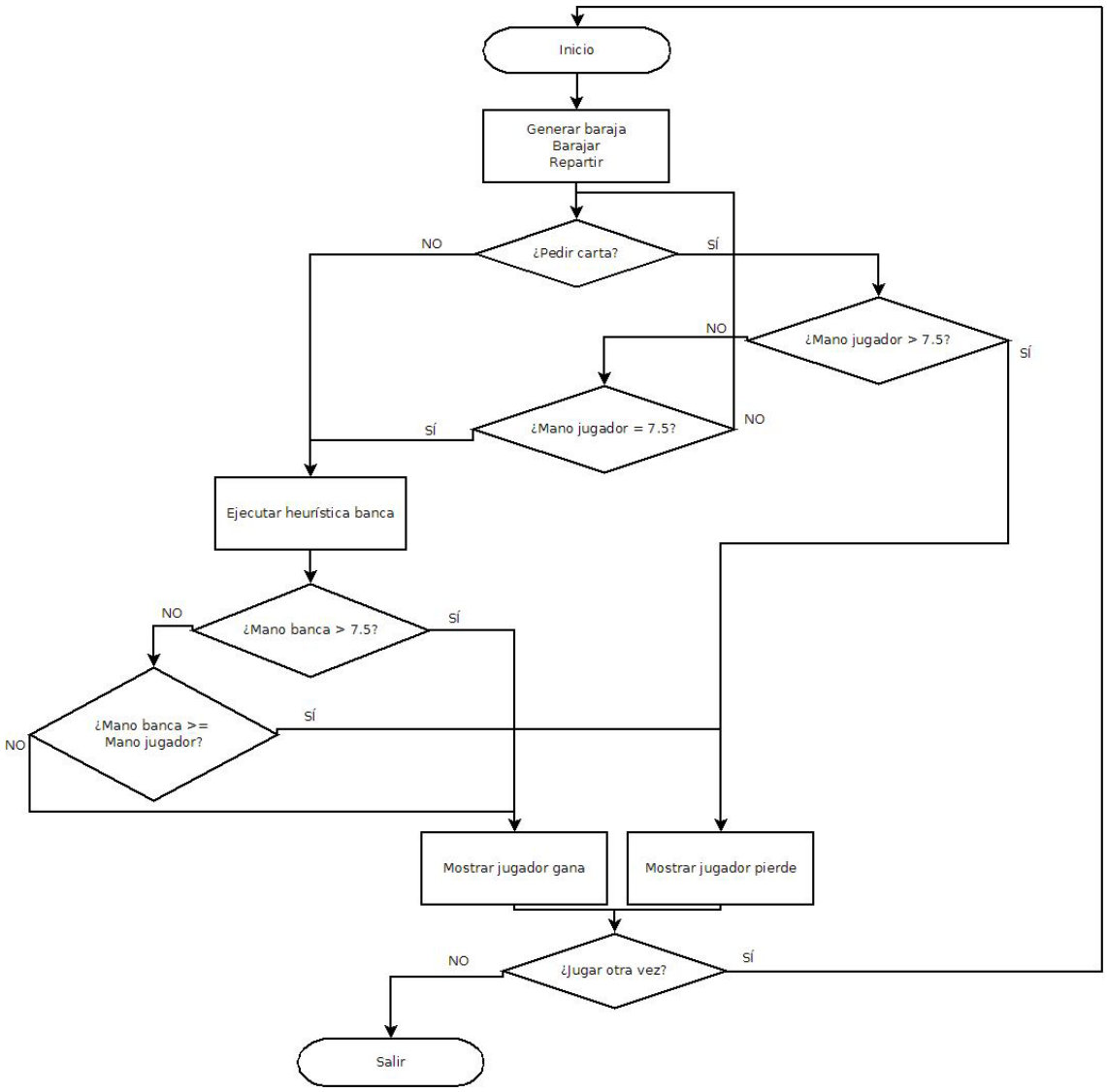
1. El valor total de las cartas del jugador supera siete puntos y medio. En este caso, el jugador ha perdido, y debe esperar a la siguiente ronda para volver a jugar. En nuestro caso, una vez que el jugador pierda se le dará la opción de jugar una nueva ronda.

2. El valor total de las cartas del jugador es estrictamente menor que siete puntos y medio. En este caso el jugador puede volver a elegir entre pedir otra carta o plantarse.

3. El valor total de las cartas del jugador suma exactamente siete puntos y medio. En este caso, el jugador se planta automáticamente y se pasa el turno a la banca. El jugador ganará la partida si la banca no consigue sumar siete y media o se pasa (en caso de empate gana la banca).

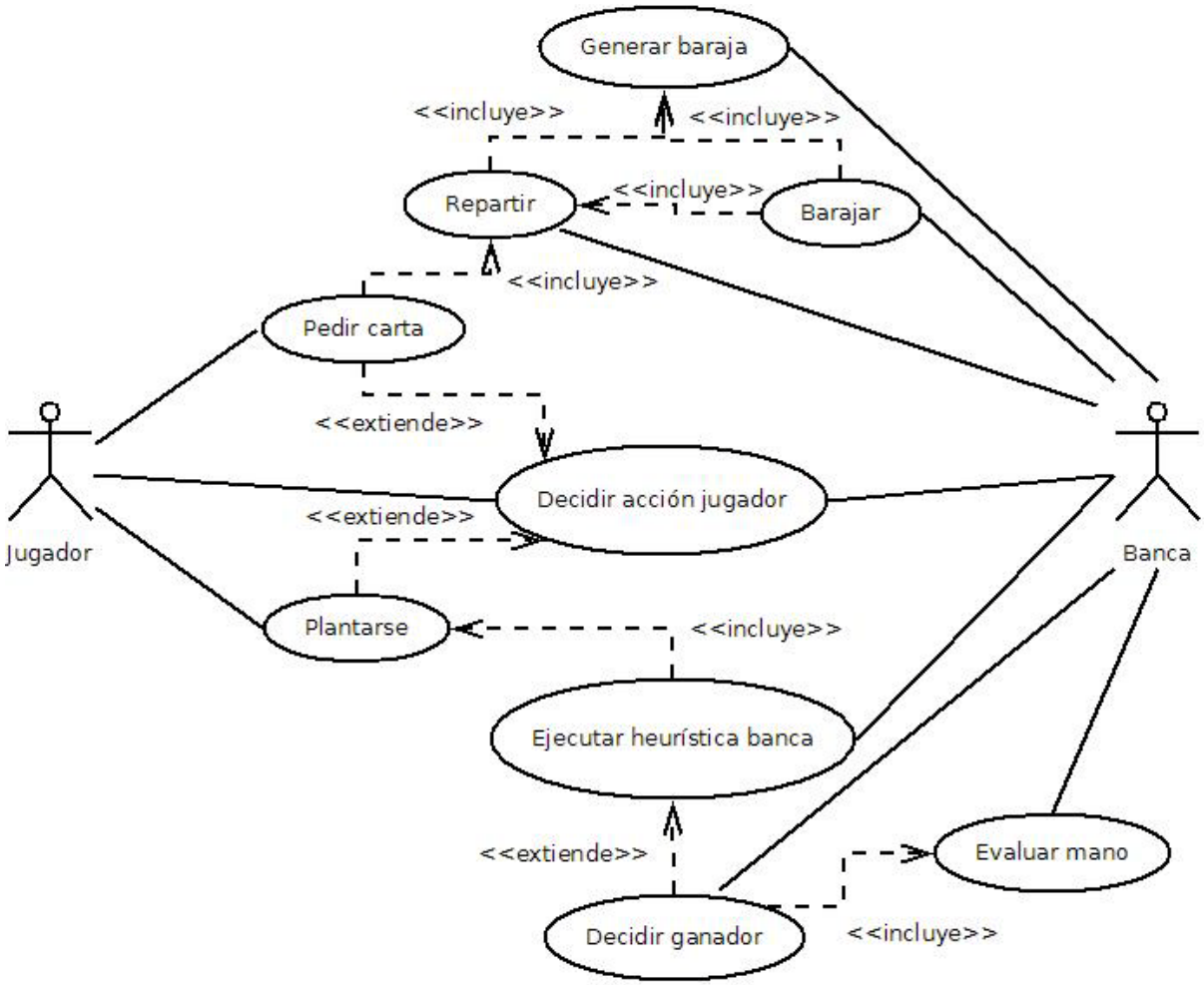
Podemos ver el flujo de juego de forma más visual con la ayuda de un diagrama de flujo

Diagrama de flujo



Casos de uso

Diagrama de casos de uso



Caso de uso: Generar baraja

Descripción breve:
Este caso de uso describe la funcionalidad que permite generar una baraja española de 40 cartas para poder jugar a las Siete y Media

Camino principal:
Este caso de uso comienza al inicio del juego, y es la primera acción que se ejecuta.

Precondiciones:
Haber ejecutado el juego.

Postcondiciones:
Queda generada una baraja española de 40 cartas sin repeticiones.

Caso de uso: Barajar

Descripción breve:
Este caso de uso describe la funcionalidad que permite barajar un mazo.

Camino principal:
Se llega a este caso de uso después de generar la baraja, y antes de repartir cartas a los jugadores.

Precondiciones:
Debemos tener una baraja de cartas.

Postcondiciones:
Queda alterado el orden de las cartas en el mazo, quedando en el mazo barajado el mismo número de cartas que en el mazo sin barajar. Es una permutación del mazo de entrada.

Caso de uso: Repartir

Descripción breve:
Este caso de uso describe la funcionalidad que permite repartir una carta a cada jugador (humano o banca).

Camino principal:
Se llega a este caso de uso después de barajar el mazo inicial.

Caminos secundarios:
También se puede llegar a este caso de uso desde el caso de uso pedir carta.

Precondiciones:
Debemos haber generado y barajado una baraja de cartas.

Postcondiciones:
Las manos del jugador y de la banca reciben una carta, y ambas cartas son extraídas del mazo.

Caso de uso: Decidir acción jugador

Descripción breve:
Este caso de uso describe la funcionalidad que permite decidir la acción que tomará el jugador durante la ejecución de una ronda.

Camino principal:
Se llega a este caso de uso después del caso de uso repartir.
El juego queda en espera hasta que el jugador decide que acción va a tomar, a elegir entre pedir carta o plantarse. El jugador debe elegir una opción válida, de lo contrario, el juego quedará bloqueado hasta que introduzca una opción válida.
Después de la elección vamos al caso de uso correspondiente (pedir carta o plantarse).

Precondiciones:
Debe haberse ejecutado el caso de uso repartir.

Postcondiciones:
Se ejecuta el caso de uso correspondiente a la acción elegida por el jugador.

Caso de uso: Pedir carta

Descripción breve:
Este caso de uso describe la funcionalidad que permite a un jugador pedir otra carta para tratar de sumar siete puntos y medio en su mano.

Camino principal:
Se llega a este caso de uso después del caso de uso decidir acción jugador, si el jugador ha elegido la opción pedir carta.

Caminos secundarios:
Si al pedir la carta la evaluación de la nueva mano del jugador es mayor que 7.5, el jugador automáticamente pierde la partida y se le pregunta si quiere jugar otra ronda.

Si al pedir la carta la evaluación de la nueva mano del jugador es 7.5, el jugador automáticamente se planta, y se ejecuta el caso de uso ejecutar heurística banca.

Precondiciones:
El jugador debe haber seleccionado la opción pedir carta en el caso de uso decidir acción jugador. Se añade una carta a la mano de un jugador.
El valor de la mano actual del jugador debe ser estrictamente menor que 7 puntos y medio.

Postcondiciones:
La mano del jugador queda en posesión de una carta más de las que tenía antes de iniciar el caso de uso. La carta que recibe el jugador debe extraerse del mazo.

Caso de uso: Plantarse

Descripción breve:
Este caso de uso describe la funcionalidad que permite a un jugador plantarse con la mano que tiene actualmente.

Camino principal:
Se llega a este caso de uso después del caso de uso decidir acción jugador, si el jugador ha elegido la opción plantarse.

Precondiciones:
El jugador debe tener en su mano al menos una carta.

Postcondiciones:
Queda definida la mano final del jugador, y se ejecuta el caso de uso ejecutar heurística banca.

Caso de uso: Ejecutar heurística banca

Descripción breve:
Este caso de uso describe la funcionalidad que permite ejecutar la jugada de la banca (jugador máquina).

Camino principal:
Se llega a este caso de uso después del caso de uso plantarse, pasarse o conseguir Siete y Media, por lo que las manos de los jugadores ya están determinadas.
La banca ejecuta su heurística de acuerdo al siguiente algoritmo:
Si va perdiendo con respecto a la mano del jugador, pide carta.
Si va ganando con respecto a la mano del jugador, se planta, para ganar al jugador.
Si el jugador se ha pasado, ejecuta su heurística contra la mano del jugador IA.

Precondiciones:
El jugador debe haberse plantado, haberse pasado, o haber conseguido Siete y Media.

Postcondiciones:
Queda definida la mano final de la banca, de forma que se puede determinar quién ha ganado la partida.

Caso de uso: Evaluar mano

Descripción breve:
Este caso de uso describe la funcionalidad que permite calcular el valor numérico de la mano de un jugador.

Camino principal:
Se llega a este caso de uso después del caso de uso repartir carta.
Se calcula el valor de la mano de un jugador como la suma del valor de cada una de las cartas que la componen, fijándonos en su valor (el palo es independiente) de acuerdo a la siguiente tabla:

Valor carta	***Valor numérico***
As	1
Dos	2
Tres	3
Cuatro	4
Cinco	5
Seis	6
Siete	7
Sota, Caballo, Rey	0.5

Caminos secundarios:
También se puede llegar a este caso de uso desde el caso de uso ejecutar heurística banca.

Precondiciones:
La mano del jugador debe tener al menos una carta.

Postcondiciones:
Queda calculado el valor numérico de la mano de un jugador.

Caso de uso: Decidir ganador

Descripción breve:
Este caso de uso describe la funcionalidad que permite decidir el jugador ganador de la ronda (jugador o máquina).

Camino principal:
Se llega a este caso de uso después de ejecutar heurística banca.
Se calcula el valor de la mano del jugador y la mano de la máquina con el caso de uso evaluar mano y se decide el ganador de la ronda.
Si la evaluación de la mano del jugador es mayor que la evaluación de la mano de la banca el jugador gana la partida. En otro caso el jugador pierde la partida (la banca gana los empates).

Precondiciones:
Las manos del jugador y de la banca tienen que tener al menos una carta.
El jugador debe haberse plantado o haberse pasado.
La banca debe haber ejecutado su heurística de juego.

Postcondiciones:
Queda decidido el ganador de la ronda y se muestra el resultado.

Implementación

Tipos de datos

Tipo Carta:

```
data Carta = Carta Valor Palo
                deriving (Eq,Show)
```

Es un tipo unión o recubrimiento, que agrupa en un constructor (Carta) varios argumentos (Valor, Palo). Los constructores de datos son funciones que devuelven un valor del tipo para el que fueron definidos.
Una carta tiene dos argumentos, un valor y un palo, que los definimos como tipos enumerados. Deriva de las clases Eq y Show para tener igualdad entre cartas y poder imprimirlas por pantalla.

Tipos Valor y Palo:

```
data Valor = As
                | Dos
                | Tres
                | Cuatro
                | Cinco
                | Seis
                | Siete
                | Sota
                | Caballo
                | Rey
                 deriving (Eq,Enum,Show)

data Palo = Oros
                | Copas
                | Espadas
                | Bastos
                  deriving (Eq,Enum,Show)
```

El tipo Valor es un tipo enumerado con todos los valores de una baraja española y el tipo Palo es un tipo enumerado con todos los palos de la baraja española. Derivan de Eq para tener igualdad entre tipos, de Enum por ser tipos enumerados y de Show para poder imprimirlos por pantalla.

Tipos Mano y Mazo:

```
type Mano = [Carta]

type Mazo = [Carta]
```

Ya tenemos definido el tipo de datos Carta, por lo que podemos definir el tipo Mano, entendido como el conjunto de cartas en posesión de un jugador en un momento determinado de la partida, como una lista de cartas. También podemos definir el tipo Mazo, entendido como el resto de cartas de la baraja que no está en posesión de ningún jugador.
Los tipos de datos Mano y Mazo son sinónimos de tipo, asignamos un nombre distinto a un tipo ya existente.

Funciones

Función generarBaraja:

```
generarBaraja :: Mazo
generarBaraja = [Carta x y | x <- [As .. Rey],  y <- [Oros .. Bastos]]
```

En esta función, que es la primera del programa generamos una baraja española con una lista intensional. La función devuelve un tipo de datos Mazo.

Función barajar:

```
barajar :: Mazo -> [Int] -> Mazo
```

Esta función genera una permutación de la baraja de entrada, usando una lista de enteros aleatorios. Utiliza varias funciones auxiliares para generar una lista de números aleatorios, de la que vamos extrayendo la cabeza y una función auxiliar mezclarAleatorio que mezcla recursivamente dos mitades de la baraja.

Función repartir:

```
repartir :: Mano -> Mazo -> (Mano, Mazo)
repartir mano mazo = (mano ++ (sacarCarta(mazo)),
sacarCartaDelMazo(mazo))
```

La función repartir recibe como entrada una mano y un mazo y devuelve un par formado por la mano más una carta extraída del mazo y el mazo sin la carta extraída.
Usamos dos funciones auxiliares sacarCarta y sacarCartaDelMazo cuyas cabeceras son:
sacarCarta :: Mazo -> Mano y sacarCartaDelMazo:: Mazo -> Mazo

Función heuristicaBanca

heuristicaBanca :: Mano -> Mano -> Mazo -> Mano

Esta función define la heurística que sigue la banca.
Si va perdiendo con respecto a la mano del jugador, pide carta.
Si va ganando con respecto a la mano del jugador, se planta, para ganar al jugador.
Si el jugador se ha pasado, ejecuta su heurística contra la mano del jugador IA.

Función heurísticaJugadorIA

heuristicaJugadorIA :: Mano -> Mano -> Mazo -> Mano

Ejecuta la heurística del jugador IA. El jugador IA hace un cálculo probabilístico para saber cuál es la probabilidad de pasarse pidiendo una carta. El jugador IA pide carta si la probabilidad de pasarse es menor del 50%.

Para el cálculo estadístico utiliza las funciones auxiliares
probabilidadPasarse :: Float -> Float y
valorSinPasarse :: [Carta] -> Float
que determinan la probabilidad de pasarse pidiendo una carta y el máximo valor que puede recibir el jugador sin pasarse.

Función jugar:

jugar :: Mazo -> Mano -> Mano -> IO ()

La función jugar implementa el flujo del juego, las acciones del jugador, la heurística de la banca y los mensajes que se muestran al usuario. Utiliza el

sistema de entrada/salida para recoger la decisión del jugador y mostrar la situación actual de la partida. Se llama recursivamente a esta función cada vez que el jugador pide una carta.

Función evaluarMano:

```
evaluarMano :: Mano -> Float
evaluarMano [] = 0
evaluarMano (x:xs) = valorCarta x + evaluarMano xs
```

La función evaluarMano devuelve la puntuación correspondiente a la mano que recibe como entrada. Se calcula como la suma de las puntuaciones de cada una de sus cartas de acuerdo a las puntuaciones definidas para el juego de las Siete y Media (ver caso de uso Evaluar mano).
Utiliza una función auxiliar valorCarta que devuelve la puntuación de una carta.

Función decidirGanador:

```
decidirGanador :: Mano -> Mano -> Int
decidirGanador manoJugador manoBanca
        | evaluarMano manoJugador > evaluarMano manoBanca = 1
        | otherwise = 0
```

La función decidirGanador evalúa las manos del jugador y de la máquina para determinar quién gana la partida. Devuelve 1 si gana el jugador y 0 si gana la banca. La banca gana los empates.

Anexo 1: Estudio de la estrategia óptima para el juego de las Siete y Media

En la función heuristicaIA veíamos como el jugador IA hacía un cálculo estadístico de la probabilidad de pasarse al pedir carta, y pedía carta si la probabilidad era menor del 50%.

Además de esta estrategia hay otra muy interesante que consiste en simular n partidas viendo qué pasaría si el jugador pidiera una carta, y ejecutar la acción del jugador en consecuencia. El valor de n debe ser un valor asumible para el coste computacional.

Para este estudio hemos implementado una serie de funciones que simulan el escenario de repartir una carta de la baraja a la mano del jugador IA y obtienen el resultado. Las funciones son las siguientes:

sePasa :: Mano -> Int
Función que devuelve 1 si la mano pasada como parámetro es mayor que 7.5 y 0 en caso contrario

simulacion :: Mano -> Mazo -> Int
Función que devuelve lo que pasaría con la mano si le repartimos una carta. Devuelve 1 si se pasa y 0 en caso contrario

simular :: Mano -> IO Int
Función que devuelve un entero (IO) con el resultado de simular la acción repartir una carta del mazo a la mano

generarSimulacion :: Int -> Mano -> [Int] -> IO [Int]
Función que genera una simulación n veces sobre una mano y devuelve una lista de enteros (IO) con el resultado de la simulación

resultadoSimulacion :: Int -> Mano -> IO ()
Función que devuelve el número de aciertos en la simulación (casos en los que el jugador no se pasa al pedir carta)

La mónada IO() aparece en el conjunto ya que para generar n barajas distintas para la simulación necesitamos hacer uso de los números aleatorios.

La siguiente tabla muestra los resultados positivos de la simulación para cada valor de mano posible. Cada simulación simula 1000 veces la acción de repartir una carta a la mano del jugador, devolviendo los casos favorables (número de veces que repartiendo una carta a la mano no se pasa). La última columna nos da el porcentaje de veces en las que el jugador no se pasaría si pidiera otra carta teniendo una mano del valor de la primera columna (probabilidad de no pasarse).

Valor mano	Simu1	Simu2	Simu3	Simu4	Simu5	Media	Porcentaje
0	1000	1000	1000	1000	1000	1000	100%
0.5	1000	1000	1000	1000	1000	1000	100%
1	933	952	937	925	940	937,4	93,74%
1.5	938	954	939	930	938	939,8	93,98%
2	865	867	872	872	861	867,4	86,74%
2.5	873	857	884	850	860	894,8	86,48%
3	784	782	791	778	797	787	78,7%
3.5	797	797	791	773	787	789	78,9%
4	702	738	728	741	729	727,6	72,76%
4.5	743	728	728	742	715	731,2	73,12%
5	651	670	638	663	662	656,8	65,68%
5.5	645	635	642	642	666	646	64,6%
6	568	563	557	561	568	563,4	56,34%
6.5	563	591	572	576	559	572,2	57,22%
7	362	376	380	367	364	369,8	36,98%
7.5	0	0	0	0	0	0	0%

Podemos apreciar con los resultados de la tabla que, asintóticamente, la probabilidad de pasarnos con una mano con un valor entero es igual a la probabilidad de pasarnos con una mano con el mismo valor entero más medio punto (excepto en el valor 7.5). Esto podemos comprobarlo con la función probabilidadPasarse

probabilidadPasarse 5 = probabilidad 5.5

Además, podemos observar que el valor óptimo para plantarse es el valor 6 (o el 6.5, como acabamos de deducir). Por tanto, un jugador IA que se plante en estos valores tiene mayor probabilidad de ganar a la banca. Esto demuestra que la estrategia que sigue la heurística del jugador es óptima, ya que la probabilidad de pasarse teniendo una mano de valor 6 ó 6.5 es del 60%, y es la primera mano que no cumple la condición para pedir otra carta (El jugador IA se plantará si consigue una mano con esos valores).

Anexo 2: Implementación de una interfaz gráfica de usuario

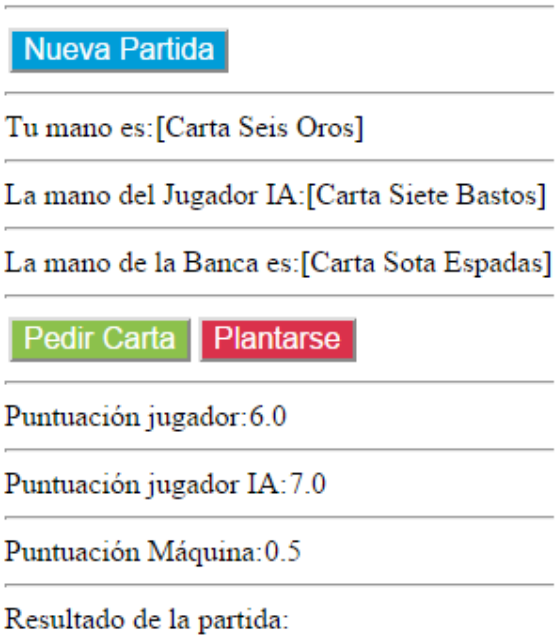
Para la implementación de la interfaz gráfica de usuario decidimos utilizar alguno de las librerías disponibles en https://hackage.haskell.org
La primera librería que investigamos fue la librería Gtk2Hs, que permite la creación de interfaces gráficas utilizando un sistema de contenedores y ventanas. Esta librería nos pareció demasiado compleja para el tiempo del que disponíamos y decidimos investigar la librería Threepenny-gui.

Threepenny-GUI es un framework que utiliza el navegador como una pantalla. Internamente la librería implementa un servidor web que sirve páginas HTML, y se puede manejar el DOM y los eventos del navegador desde el código en Haskell. La librería viene con algunos ejemplos, y también nos ha ayudado nuestro conocimiento de los lenguajes web como el HTML, JavaScript y CSS, con los que trabajamos a diario.

Para instalar el framework Threepenny-GUI podemos ejecutar cabal (la base de datos de paquetes de Haskell)

```
cabal install threepenny-gui -fbuildExamples
```

Por defecto el servidor web viene configurado para trabajar en el puerto 8023. Para cargar el código compilamos con ghc o cargamos con ghci la práctica y ejecutamos la función main. Si todo ha funcionado correctamente podemos ver el resultado en cualquier navegador accediendo a la dirección localhost:8023.



Conclusiones y trabajo futuro

Conclusiones

Haskell es un lenguaje de programación puramente funcional, esto obliga al desarrollador a cambiar radicalmente la forma de abordar un problema con respecto a la programación imperativa. Estamos tan acostumbrados a los lenguajes imperativos que al principio cuesta creer que se pueda programar sin asignaciones o declaraciones de variables, y este fue el primer problema que se nos presentó.

Para realizar un algoritmo es Haskell hay que pensar más en la especificación de un problema que en su implementación, aunque una vez hecha la especificación la implementación es prácticamente traducir la especificación. Como un programa en Haskell se construye como un conjunto de funciones que se evaluarán durante la ejecución del programa, lo mejor a la hora de abordar un problema es descomponerlo en funciones.

Una de las ventajas de Haskell es que es sencillo y claro, las funciones pequeñas se explican por sí solas, y al generar menos líneas de código hace que los programas más fáciles de mantener. Por otra parte, los mensajes de error del compilador son poco descriptivos y a veces te pierden más, y la tarea de depuración es algo más compleja que con un IDE como Eclipse.

En cuanto a la práctica, hemos elegido algo sencillo debido a limitaciones de tiempo, pero es una práctica que se puede ampliar para darle un valor añadido. Una ampliación podría ser la implementación de un sistema de apuestas, para hacer el juego más interesante, permitiendo al usuario ingresar más dinero virtual si se le acaba. El sistema pagaría al usuario la cantidad apostada en caso de que gane a la banca y el doble de la cantidad apostada en el caso de que gane con Siete y Media.

La experiencia con los lenguajes declarativos ha sido muy buena, ya que aporta una visión distinta de cara a la resolución de problemas, y te hace pensar si en ocasiones no sería mucho más sencillo utilizar un lenguaje declarativo en lugar de uno imperativo, algo que raras veces nos planteamos en el análisis de un problema.

Bibliografía

Como bibliografía hemos utilizado principalmente Internet y los apuntes de programación funcional.

Proyectos de código abierto en GitHub
https://github.com/mpeltier25/BlackJack

Documentación de Haskell, Haskell Wiki
http://www.haskell.org/haskellwiki/Haskell

Reglamento del juego de las Siete y Media,
http://www.nhfournier.es/public/assets/Reglamentos_22/Reglamento_Las_Siete_y__Media.pdf

¡Aprende Haskell por el bien de todos!
http://aprendehaskell.es/

Jeremy Shaw, Learn Haskell in just 5 minutes per day,
http://learnhaskell.blogspot.com.es/

Steven Cheng, Experimenting with game engine concepts in Haskell,
http://free-idea-monoid.blogspot.com.es/2014/03/experimenting-with-game-engine-concepts.html

Jeff Foster, Playing cards with Haskell,
http://www.fatvat.co.uk/2009/12/playing-cards-with-haskell.html

Functional Programming Lab 2. A Simple Black Jack Variant, Universidad Tecnológica Chalmers, Suecia,
http://www.cse.chalmers.se/edu/year/2010/course/TDA451_Functional_Programming/labs/2/lab2.pdf

Documentación de la librería Gtk2Hs
http://projects.haskell.org/gtk2hs/documentation/

Documentación del framework Threepenny-gui
http://www.haskell.org/haskellwiki/Threepenny-gui

Números aleatorios en Haskell
http://hackage.haskell.org/package/random-1.0.0.2/docs/System-Random.html

Haskell and random numbers
http://stackoverflow.com/questions/2738581/haskell-and-random-numbers

Mónada IO
http://www.haskell.org/haskellwiki/IO_inside

Operaciones con la mónada IO (recuperar un valor de un tipo IO Tipo para pasárselo a una función)
http://stackoverflow.com/questions/4235348/converting-io-int-to-int

Estudio de la estrategia óptima para el black-jack, Juan Tejada Cazorla y Javier Yañez Gestoso, Departamento de Estadística e Investigación Operativa, Facultad Ciencias Matemáticas. Universidad Complutense. Madrid.
https://www.google.es/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0CCEQFjAA&url=http%3A%2F%2Fwww.ine.es%2Fss%2FSatellite%3Fblobcol%3Durldata%26blobheader%3Dapplication%252Fpdf%26blobheadername1%3DContent-Disposition%26blobheadervalue1%3Dattachment%253B%2Bfilename%253D309%252F33%252F107_8.pdf%26blobkey%3Durldata%26blobtable%3DMungoBlobs%26blobwhere%3D309%252F33%252F107_8.pdf%26ssbinary%3Dtrue&ei=XIMVVPjjE9GI7AbGk4CoBw&usg=AFQjCNF9pzwVOY7HeE8zpCKJDbQNAVbE2g&sig2=yg7FpBDECIViOD5xEjPqRA&bvm=bv.75097201,d.ZWU