9

Algoritmos genéticos para generación de protocolos incondicinalmente seguros

Ignacio Hernández Antón
Fernando Soler Toscano

9.1. Introducción

Podemos definir protocolos que no dependen de la complejidad computacional de la codificación para la transmisión de mensajes; de hecho, la información a transmitir no se encripta. En vez de mensaje encriptados, se utilizan anuncios públicos donde la información es abierta y "entendible" por todos los agentes. Estos protocolos han de garantizar que:

1. Cuando terminan, han sido informativos: cuando se ejecuten se ha de garantizar que la información que se desea transmitir llega íntegramente a los destinatarios elegidos como receptores adecuados. Con *íntegramente* queremos indicar que el destinatario elegido recibe la información suficiente que nos asegura que nuestro mensaje no pierde originalidad.

2. Cuando terminan, han garantizado *totalmente* que los destinatarios no elegidos (los interceptores o intrusos ilícitos) no puedan deducir toda la información contenida en el mensaje.

3. Los mensajes transmitidos mediante anuncios públicos que un agente ejecuta se basan exclusivamente en la información que dicho agente posee.

4. Son independientes del tipo de información a transmitir.

A continuación centramos el estudio de la seguridad en la comunicación en lo que se refiere a ataques en el proceso de comunicación contra el mensaje (eavesdropping), dejando intencionamente de lado los otros tipos de ataques. Estudiaremos, mediante la presentación del caso del juego de las cartas rusas, cómo intentamos garantizar que el mensaje sólo es compartido por aquellos agentes que consideramos legítimos independientemente de que se permita la escucha pasiva de la información transmitida. Existe un tratamiento lógico donde se formaliza el problema con lógica epistémica dinámica, para profundizar en este enfoque ver [1].

9.1.1. Presentación del problema

De un paquete de siete cartas conocidas, dos agentes toman tres cartas cada uno y un tercer agente "espía" coge la séptima y última carta. ¿Cómo podría cada agente de tres cartas informar abiertamente, sin cifrar, al otro sobre sus propias cartas evitando que el espía aprendiera los correspondientes repartos incluso si escucha y entiende la información transmitida?

Para familiarizarnos con este escenario podemos llamar a los agentes a, b y c, las siete cartas 0,...,6; a las "manos" de cada agente las indicamos con una cadena de enteros. Son principales legítimos a y b mientras que c es el agente *espía*. Para estudiar el problema de las cartas rusas (PCR) vamos a suponer que el reparto real se corresponde con la lista: 012.345.6. En este escenario, la comunicacióon entre agentes se realiza mediante anuncios públicos y verdaderos.

Es razonable hacerse la siguiente pregunta: ¿qué consituye un anuncio seguro? Un anuncio seguro es aquel que mantiene la condición de ignorancia del espía c así como el conocimiento común en el grupo de agentes considerados de dicha ignorancia. El espía es ignorante y es conocimiento común que es ignorante.

Observando la definición anterior, se puede pensar que una solución del problema de las cartas rusas es una secuencia de anuncios después de los cuales se verifica que:

- Los principales legítimos, a y b conocen respectivamente sus manos.
- Es conocimiento común, al menos entre los principales legítimos, que conocen sus respectivas manos.
- El espía c permanece ignorante.
- Es conocimiento común de todo el grupo que el espía c permanece ignorante.

Resumimos el escenario del PCR (012.345.6) con el que vamos a trabajar con la siguiente tabla:

$A = \{a, b, c\}$	Agentes
$C = \{0, 1, 2, 3, 4, 5, 6\}$	Baraja
$\#d = \#a.\#b.\#c = 3.3.1$	Distribución del reparto
$\|C\| = \|\#d\| = 7$	Tamaño de la baraja que coincide con el del reparto
$d^{-1}(a) = 012; \#a = 3$	Mano de a; su tamaño
$d^{-1}(b) = 345; \#b = 3$	Mano de b; su tamaño
$d^{-1}(c) = 6; \#c = 1$	Mano de c; su tamaño
$d_0 = (0_a \wedge 1_a \wedge 2_a \wedge 3_b \wedge 4_b \wedge 5_b \wedge 6_c)$	Reparto (poseedores de las cartas)

El objetivo de este trabajo es emplear algoritmos genéticos para buscar soluciones para el caso (3,3,1) del problema de las cartas rusas [1]. Ya que este caso es suficientemente conocido nos servirá para estudiar la adecuación de los algoritmos genéticos para resolver este problema, y tal vez emplearlos en un futuro en repartos mayores.

Los algoritmos genéticos, propuestos por Holland [4], son uno de los métodos de búsqueda heurística más empleados en la inteligencia artificial para resolver problemas de optimización. Parten de una población compuesta por cierto número de posibles soluciones a un problema (cromosomas) generadas aleatoriamente. Generalmente cada cromosoma es una cadena de bits (genes) de longitud fija. Por ello, se debe buscar una representación binaria del espacio de posibles soluciones del problema al que se aplica el algoritmo genético. A la población inicial se le aplican ciertas operaciones para ir produciendo nuevas *generaciones*. Las más importantes:

- *Selección.* En analogía con la selección natural, evalúa los mejores cromosomas que son los que pasarán a la siguiente generación y podrán reproducirse. Para ello se define una función de evaluación, o *fitness* que opera sobre una cadena de bits e indica qué tan buena es una solución.

- *Cruzamiento.* Operación análoga a la reproducción que, a partir de dos cromosomas genera dos nuevos descendientes en los que se combinan características de ambos padres. Con ello se obtienen nuevas soluciones al problema.

- *Mutación.* Con cierta probabilidad, los nuevos descendientes pueden modificar alguno de sus genes. Esta operación evita el estancamiento en máximos locales.

El algoritmo genético se ejecuta hasta un cierto número de generaciones o hasta que aparece una solución que se considera suficientemente buena. La probabilidad de encontrar la mejor solución aumenta tras cada generación.

En este trabajo usamos un algoritmo genético implementado en Haskell. Hemos empleado la librería `hgalib: Haskell Genetic Algorithm Library` creada por Kevin Ellis. Su código se encuentra bajo dominio público en:

`http://hackage.haskell.org/package/hgalib`.

9.2. Especificación del Problema de las Cartas Rusas

La primera decisión que debemos tomar al usar algoritmos genéticos para resolver un problema es cómo vamos a representar una posible solución mediante una cadena de bits (cromosoma).

En el caso de las cartas rusas, todo anuncio es equivalente a una disyunción de manos (ver [1]). Como lo que buscamos es el anuncio de a que comunique a b sus cartas, y sólo hay $\binom{7}{3} = 35$ manos posibles para a, podemos representar las posibles soluciones del problema mediante cadenas de 35 bits (valores booleanos en Haskell). Cada uno de los bits representa la presencia o ausencia de una determinada mano en la disyunción que anuncia a. El siguiente código nos muestra en primer lugar el conjunto de todas las manos posibles de a (también son las que b puede tener) y una función que dada una lista de valores booleanos (bits) devuelve el conjunto de manos (anuncio) representado:

```
abHands :: [[Integer]]
abHands = [[a1, a2, a3] | a1 <- [0..6], a2 <- [a1+1..6],
           a3 <- [a2+1..6]]

aHandsNumber :: [Bool] -> [[Integer]]
aHandsNumber bits = [aH  | (True,aH) <- (zip bits abHands)]
```

Buscamos un anuncio de a tal que b adquiera el mayor conocimiento posible (todas las cartas de a) y c el mínimo (ninguna carta de a ni de b). Además, los requisitos deben ser conocimiento común. Hemos modelado en Haskell estas condiciones para emplearlas en la función de evaluación (*fitness*). A continuación mostramos tres funciones interesantes para modelar el conocimiento que b adquiere tras el anuncio de a.

En primer lugar, `howManyBknows` toma como argumentos un anuncio de a (conjunto de posibles manos) que llamamos `handsA` y `hB`, una mano de b. Devuelve la cantidad de cartas que b aprendería si a hiciera el anuncio `handsA` y las cartas de b fueran `hB`. Lo que hace esta función es descartar todas las manos de `handsA` donde hay alguna carta de `hB` (b sabe que esas no pueden ser manos de a) y contar las cartas que quedan en la intersección de las manos restantes:

```
howManyBknows :: (Enum a, Num a) => [[a]] -> [a] -> Int
howManyBknows handsA hB = length (foldl intersect [0..6]
               [a | a<-handsA, (intersect a hB) == []])
```

La función `handsBcompat` toma como argumento una lista de manos `handsA` que representa el anuncio de a y devuelve la lista de posibles manos que b puede tener compatibles con dicho anuncio. Esto es, las manos de `abHands` que tienen intersección vacía con alguna de las manos de `handsA`:

```
handsBcompat :: [[Integer]] -> [[Integer]]
handsBcompat handsA = [hB | hB <- abHands, (elem []
                         (map (intersect hB) handsA))]
```

Finalmente, nos interesa saber cuál es el número mínimo de cartas que b sabe de a, es decir, el número n tal que tras el anuncio de a es conocimiento común de los tres agentes que b sabe *al menos* n cartas de a. La función `commonKnowBknows` toma como argumento `handsA`, conjunto de manos que anuncia a, y devuelve las mínimas cartas que b aprende. Lo que hace es ver el mínimo valor que devuelve `howManyBknows` para todas las manos de b compatibles (mediante `handsBcompat`) con el anuncio de a:

```
commonKnowBknows :: [[Integer]] -> Int
commonKnowBknows handsA = foldl min 7 (map
                  (howManyBknows handsA) (handsBcompat handsA))
```

Esta función es una de las que nos interesarán para evaluar la calidad de una solución. También queremos garantizar que c no aprende nada. En otro caso, el anuncio de a `[[0,1,2]]` garantiza que b aprende tres cartas de a, pero sabemos que no es seguro. Por tanto, debemos calcular cuántas cartas *puede* aprender c tras el anuncio de a.

En primer lugar, la siguiente función `cardsCcompat` nos devuelve las cartas de c compatibles con cierto anuncio `handsA` de a. Simplemente busca las cartas que no aparecen en alguna de las manos de `handsA`:

```
cardsCcompat :: (Enum a, Num a) => [[a]] -> [[a]]
cardsCcompat handsA = [[cC] | cC <- [0..6],
             (elem [] (map (intersect [cC]) handsA))]
```

La siguiente función `howManyCknowsB` toma un anuncio de a, `handsA`, y una mano de c, `handC`, y devuelve el número de cartas de b que c aprende si `handC` es su mano. Tras descartar c todas las manos de a que contienen su carta, las cartas que no son la de c y tampoco están en las manos de a restantes, son cartas que c sabe que las tiene necesariamente b:

```
howManyCknowsB :: (Enum a, Num a) => [[a]] -> [a] -> Int
howManyCknowsB handsA handC = length (([0..6] \\ handC ) \\
               (foldl union [] [a | a <- handsA,
               (intersect a handC) == []]))
```

La siguiente función `totalCknowledge` nos dice cuál es el número total de cartas que c aprende (máximo seis) si `handsA` son las manos que anuncia a y `handC` es la mano de c (una carta). Para saber las cartas que c aprende de a, reutilizamos la función `howManyBknows` que explicamos más arriba, pero ahora con la mano de c. Las cartas de c aprende de b las obtenemos con `howManyCknowsB`:

```
totalCknowledge :: (Num a, Enum a) => [[a]] -> [a] -> Int
totalCknowledge handsA handC = (howManyBknows handsA handC) +
                (howManyCknowsB handsA handC)
```

La función anterior devolvía las cartas que c aprende si su mano es `handC`, pero nos interesa saber cuál es el máximo número de cartas que c puede saber sea cual sea su mano, que habrá de ser igual a cero es un protocolo incondicionalmente seguro. Eso es lo que hace `maxCKnows`, que dado un anuncio `handsA` de a calcula el máximo número de cartas que c puede aprender del reparto, para todas las manos de c compatibles. Se define a partir de las funciones anteriores:

```
maxCKnows :: (Enum a, Num a) => [[a]] -> Int
maxCKnows handsA = foldl max 0 (map (totalCknowledge handsA)
          (cardsCcompat handsA))
```

Una de las partes fundamentales en el algoritmo genético es la función de evaluación o *fitness* que nos da la calidad de una solución. En nuestro caso, las cadenas de bits (cromosomas) representan anuncios de a, y nos interesa maximizar el número de cartas que b aprende y minimizar las cartas que aprende c. Por tanto, al mínimo número de cartas que b aprende restamos el máximo número de cartas que c podría aprender. Ese es el valor que habrá que maximizar durante la evolución del algoritmo:

```
valuation :: [[Integer]] -> Int
valuation handsA = 5*(commonKnowBknows handsA) -
                   3*(maxCKnows handsA)
```

Los pesos 5 y 3 que aparecen se han fijado experimentalmente, buscando una mejor convergencia.

9.3. Ejecución del algoritmo genético

A continuación mostramos los parámetros fundamentales de inicialización del algoritmo y las opciones de ejecución. Pueden obtenerse más detalles a partir de la documentación de `hgalib` y en los ejemplos comentados que contiene. Nosotros sólo hemos comentado los parámetros más relevantes:

```
-- Parámetros de configuración
myconfig = defaultConfig {
   cConfig = B.config {
      -- Función de evaluación:
      fitness = fromIntegral . booleanVal,
      -- Probabilidad de mutación:
      mutate = B.mutateBits (0.01 :: Double)
   },
   pConfig = L.config,
   newPopulation = mynewPopulation,
   maxGeneration = Just 500, -- Paramos a las 500 generaciones
   gen = mkStdGen 48         -- Semilla generador de nos aleat.
}

mynewPopulation pop = mutateM pop >>= rouletteM

-- La población inicial son 200 cromosomas de 34 bits cada uno
initPop = liftM (map (++[])) $ replicateM 200 (B.randomBits 34)

-- Bucle principal del algoritmo
getBest = do
  pop <- initPop
  before <- bestChromosome pop
  answer <- run pop
  after <- bestChromosome answer
  return (before, after)

-- Función de fitness
booleanVal :: [Bool] -> Int
booleanVal bits = valuation (aHandsNumber ([True]++bits))

-- Función principal
main = do
  let (before, after) = evalState getBest myconfig

  -- Muestra mejor cromosoma inicial y su evaluación
  putStrLn $ show (aHandsNumber ([True]++before))
  putStrLn $ show $ round $ fromIntegral $ booleanVal before

  -- Muestra mejor cromosoma final y su evaluación
  putStrLn $ show (aHandsNumber ([True]++after))
  putStrLn $ show $ round $ fromIntegral $ booleanVal after
```

Se puede observar que los cromosomas que se manejan tienen sólo 34 bits, no los 35 de los que hablamos en la sección anterior. Es así porque la mano `[0,1,2]` (la mano real de a) debe pertenecer a cualquier solución válida del problema, por lo que sólo nos interesa que evolucionen las otras 34 manos posibles. Con ello ahorramos un bit de cada solución, lo cual no supone mucho en 200 cromosomas durante 500 generaciones, pero se impide que por azar pueda mutar este bit que debe ser siempre `True`. Podemos ver que para evaluar las soluciones, así como para mostrarlas, siempre se añade este bit inicial.

Cuando ejecutamos el algoritmo obtenemos la siguiente solución:

```
[[0,1,2], [0,1,4], [0,1,5], [0,2,3], [0,2,5], [0,3,4], [0,3,6],
 [0,4,5], [0,5,6], [1,2,3], [1,2,4], [1,3,6], [3,4,5], [3,4,6],
 [4,5,6]]
5

[[0,1,2], [0,3,6], [0,4,5], [1,3,4], [2,3,5], [2,4,6]]
15
```

El primer anuncio que vemos, de 15 manos, garantiza que b conoce al menos una carta de a y es conocimiento común la total ignorancia de c. Es la mejor solución de las 200 aleatorias que componen la población inicial.

En el anuncio final, sin embargo, es conocimiento común que b conoce todas las cartas de a y que c es completamente ignorante. Por tanto, es una solución incondicionalmente segura al problema de las cartas rusas. Es una de las soluciones minimales recogidas en [1].

9.4. Conclusiones

Podemos ver el problema de las cartas rusas como un caso particular del problema mucho más general de síntesis de programas. Como se muestra en [1], un protocolo es un programa epistémico que modifica un modelo de Kripke a través de actualizaciones (anuncios públicos) que cambian el conocimiento de los agentes implicados. Dadas unas precondiciones (modelo epistémico inicial), la búsqueda de un protocolo pretende encontrar cuáles son las acciones (anuncios públicos) que permiten llegar a un estado final en que se cumplen ciertas postcondiciones (conocimiento e ignorancia de los agentes).

Una de las áreas de aplicación de los algoritmos genéticos es precisamente la síntesis de programas, mediante técnicas como la programación evolutiva. Como hemos visto en este trabajo, estas técnicas se pueden extender al diseño de protocolos incondicionalmente seguros. Es más, la convergencia del algoritmo genético ha sido más rápida que la búsqueda de soluciones en otras pruebas que habíamos realizado anteriormente con DEMO [3], donde generábamos posibles protocolos y los verificábamos uno a uno.

Los dos elementos principales de la programación de un algoritmo genético son, como vimos, la elección de una codificación binaria del espacio de soluciones del problema y la definición de la función de evaluación. Respecto a la codificación, nos parece que la decisión tomada de representar cada posible mano del anuncio de a con un gen es la más acertada, pues permite un fácil cruzamiento entre cromosomas. Además, la extensión del algoritmo a repartos distintos de (3,3,1), incluso a protocolos de más pasos, es directa. Basta con considerar una secuencia de genes por cada uno de los anuncios que integran el protocolo. En cuanto a la función *fitness* que evalúa la calidad de los cromosomas, resulta inevitable que requiera gran cantidad de operaciones combinatorias, dada la naturaleza del problema. Sin embargo, pensamos que para repartos mayores que (3,3,1) debemos buscar alternativas más eficientes que nuestra función `valuation`, que no requieran la evaluación completa en todos los casos.

No podía faltar un párrafo dedicado al que ha sido nuestro profesor y mentor, al que nos inculcó, con su buen hacer pedagógico e investigador, la disciplina de las ciencias formales. Para Ángel Nepomuceno va dedicado con cariño este simbólico presente académico por el cual nos complace transmitir, en estas fechas tan especiales para él, nuestro más sincero agradecimiento y admiración por una trayectoria profesional envidiable y por las constantes muestras de su gran corazón como persona.

Bibliografía

[1] H.P. van Ditmarsch, "The Russian Cards Problem". *Studia Logica*, 75:31-62, 2003.

[2] H.P van Ditmarsch, W. van der Hoek, R. van der Meyden, J. Ruan, "Model Checking Russian Cards". *Electronic Notes in Theoretical Computer Science*, 149:105-123, 2006.

[3] J. van Eijck, *Dynamic Epistemic Modelling*, Technical Report, disponible en `http://homepages.cwi.nl/~jve/`, 2005.

[4] J.H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, The MIT Press, 1992.