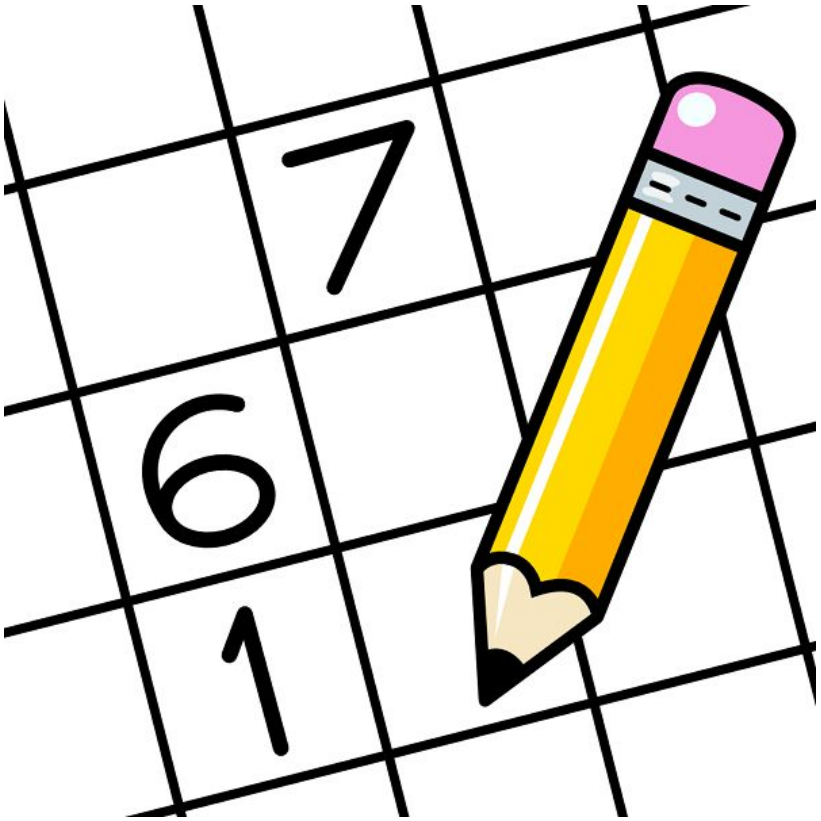
El rompecabezas de sudoku



Javier García Jiménez, David Moreno López, Isabel Martínez Gómez
Laboratorio Inteligencia Artificial - Práctica 1
Universidad de Alcalá
Curso 2019-2020

1. Introducción

El trabajo a realizar consiste en la implementación de un código en Racket que resuelva el problema del Sudoku.

El Sudoku es un juego que consiste normalmente en una tabla compuesta de 9 filas y 9 columnas, que a su vez se divide en 9 sub-tablas de dimensión 3x3.
Cada casilla de la tabla irá etiquetada por un número siendo este número un valor de 0 a 9. El valor 0 indicará que una casilla está vacía mientras que los valores del 1 al 9 indicarán el valor de la casilla, es decir, las pistas que un jugador dispone para resolver el Sudoku.
Para que un Sudoku quede resuelto se han de rellenar todas las casillas del mismo sin que un mismo número se repita en una misma fila, columna o cuadrante.

Para la resolución de este problema se pueden usar numerosos algoritmos, en nuestro caso resolveremos el problema del Sudoku aplicando los algoritmos de búsqueda en profundidad y búsqueda en anchura.

2. Discusión y formulación del problema

Para la posible y correcta resolución del problema, un Sudoku dado debe cumplir las siguientes reglas:

- Un Sudoku debe tener una única solución y para ello, un Sudoku con dimensión 9x9 debe tener al menos 17 números.
- Un Sudoku no debe tener el mismo valor repetido en una misma fila, columna o cuadrante.

Teniendo en cuenta las reglas anteriores y como hemos dicho anteriormente, resolveremos el Sudoku con los siguientes métodos:

- Búsqueda en Profundidad, DFS (Depth-First Search)
    - Con búsqueda en profundidad trabajaremos con una lista de nodos abiertos que llevará el control de todos los sudokus generados y que será tratada como si fuese una pila sacando primero al último sudoku que entró en la lista.

- Búsqueda en Anchura, BFS (Breadth-First Search)
    - Con búsqueda en anchura trabajaremos con una lista de nodos abiertos que llevará el control de todos los sudokus generados y que será tratada como si fuese una cola, sacando a los sudokus de la lista de abiertos en forma de FIFO (First In First Out).

3. Método de resolución empleado

En este apartado se dará una explicación detallada de todas las funciones existentes en el programa.

- **Función imprimir:**

```
(define (imprimir sudoku)
  (for*([fila dimension]
        [columna dimension])
    (when (and (equal? (remainder fila 3)0) (= columna 0))
      (printf "+-------+-------+-------+\n"))
    (when(equal?(remainder columna 3) 0)
      (printf "~a" "| "))
    (printf "~a" (getValorCasilla fila columna sudoku))
    (printf " ")
    (when (= columna 8) (printf "~a" "|\n"))
    (when (and (= fila 8) (= columna 8))(printf  "+-------+-------+-------+\n\n"))))
```

Esta función como su propio nombre indica se encarga de imprimir el sudoku. El parámetro de esta función llamado sudoku será una lista de listas.

La forma de imprimir el sudoku está realizada con el objetivo de mostrar el mismo dividido por cuadrantes. Para ello, se realiza un for*, éste funciona igual que un for pero tiene un #:when #t implícito entre cada sentencia del for. Con este método recorremos todas las casillas del sudoku y, en cada una de ellas se comprueba:

- Si es una casilla que está en el medio de un cuadrante, se imprime el valor.
- Si es una casilla que está en la primera columna de un cuadrante, se imprime una "|" antes de imprimir su valor.
- Si es una casilla que está en la primera fila de un cuadrante, se imprime una línea divisoria antes de imprimir su valor.
- Si es una casilla que está en la última fila y columna del sudoku, se imprimirá una línea divisoria.

Para saber cuando una casilla se encuentra en el límite de un cuadrado, ya sea en su fila o en su columna, se utiliza la función remainder la cual devuelve el resto de dividir la fila o la columna. En este caso, se divide la fila o la columna por 3 porque, si el resto de dicha división es cero, se trata de una casilla límite, si no, se trata de una casilla normal.

- **Función getValorCasilla:**

```
(define (getValorCasilla fila columna sudoku)
  (list-ref (list-ref sudoku fila) columna))
```

Esta función se encarga de devolver el valor de una casilla del sudoku. Se saca a través de una fila, columna y un sudoku dado.

- **Función comprobarCuadrante:**

```
(define (comprobarCuadrante fila columna valor sudoku)
  (define x (remainder fila 3))
  (define y (remainder columna 3))
  (define a (- fila x))
  (define b (- columna y))
  (for/or ([i (/ dimension 3)])
    (for/or  ([j (/ dimension 3)])
      (equal? (getValorCasilla (+ a i) (+ b j) sudoku) valor))))
```

Esta función se encarga de comprobar si un valor se encuentra dentro de un cuadrante o no. Para ello, dados los parámetros fila, columna, valor y sudoku, se conoce la posición de la casilla, el valor de la misma y el sudoku sobre el cual se va a realizar la comprobación.

Para poder recorrer el cuadrado al que pertenece la casilla, primero se define un valor x que se corresponde al resto de dividir la fila entre 3. Lo mismo se realiza con una variable "y", cambiando la fila por la columna. Gracias a esto se puede saber la posición de una casilla dentro de un cuadrante.

Tras esto, se define un valor "a" el cual se corresponde a restarle el valor de "x" a la fila dada. Con esto se consigue conocer la fila donde comienza el cuadrante, ya que si la casilla se encuentra por ejemplo en la fila 5, el resto de dividir 5 entre 3 es 2, por tanto el valor de a sería 3, esto es, la primera fila del cuadrante donde se encuentra la casilla.

Lo mismo se hace para un valor "b", cambiando la fila por la columna. Con esto conseguimos saber con exactitud dónde empieza el cuadrante exactamente.

Para terminar, recorremos a través de dos for/or anidados un cuadrante. Esto se hace a través de dos valores "i" y "j", que recorren el cuadrante tomando los valores 0 1 y 2. Cada vez que entramos en una casilla nueva, sacamos el valor de la casilla gracias a la función getValorCasilla pasándole como parámetros la fila (que resulta de sumar "a" e "i"), la columna (que resulta de sumar "b" a "j") y el sudoku.

La suma de "a" y "b" con "i" y "j" respectivamente, se realiza con el objetivo de conocer la casilla real que se quiere mirar, ya que "i" y "j" solo recorren un cuadrante de la posición 0 a la 2, pero "a" y "b", guardan los valores de comienzo de un cuadrante, así que, si por ejemplo,

se quiere mirar la casilla que se encuentra en la fila 7 y la columna 4, "a" tendría el valor 6 y b el valor 3, mientras que "i" tendría el valor 1 y "j" también, por tanto, se accede a la casilla que se quiere comprobar.

Dado que se recorre el cuadrante con dos for/or anidados, si el resultado de comparar un valor de una casilla con el valor introducido por parámetros es el mismo, entonces directamente se termina la ejecución de los for/or y se llega a la conclusión de que el valor introducido por parámetros ya existe en el cuadrante.

- **Función comprobarFila**

```
(define (comprobarFila fila valor sudoku)
  (for/or ([columna dimension])
    (equal? (getValorCasilla fila columna sudoku) valor)))
```

Esta función recibe una lista con una fila del sudoku, un valor y el sudoku y comprueba si ese valor dado se encuentra en esa fila. Para ello utiliza un bucle for/or que va comprobando desde la posición 0 de la lista hasta la 9 si algún valor de esa fila es igual al valor dado como parámetro mediante la función equal?, por tanto:

- Si el valor dado coincide con un valor existente en la fila la función equal? devuelve #t y se sale del bucle for/or para que la función finalmente devuelva true.
- Si por lo contrario, el valor dado no coincide con ningún valor devuelve #f.

- **Función comprobarColumna**

```
(define (comprobarColumna columna valor sudoku)
  (for/or ([fila dimension])
    (equal? (getValorCasilla fila columna sudoku) valor)))
```

Esta función recibe una lista con una columna del sudoku, un valor y el sudoku y comprueba si el valor dado por parámetros se encuentra en la columna dada. Para ello utiliza un bucle for/or que va comprobando desde la posición 0 de la lista hasta la 9 si algún valor de la columna es igual al valor dado mediante la función equal?, por tanto:

- Si un valor dado coincide con un valor existente en la columna la función equal? devuelve #t y se sale del bucle for/or para que la función finalmente devuelva true.
- Si por lo contrario, el valor dado no coincide con ningún valor devuelve #f.

- **Función getFila**

```
(define (getFila fila sudoku)
  (list-ref sudoku fila))
```

Esta función recibe como parámetro un número que indica el número de fila que quiere obtener y el sudoku. La función devuelve una lista con la fila indicada del sudoku dado.

- **Función getColumna**

```
(define (getColumna fila columna sudoku)
  (if (> fila 8 ) (append '())
      (append (list (getValorCasilla fila columna sudoku)) (getColumna (+ 1 fila) columna sudoku))))
```

Función que devuelve una lista con una columna del sudoku. Esta función recorre de manera recursiva una columna del sudoku, añadiendo a una lista todos los valores de dicha columna.

Recibe como parámetros la fila (para llevar un control sobre la fila de la que hay que sacar el valor), la columna (para conocer la columna de la que se van a sacar los valores) y el sudoku.

- **Función getLimites**

```
(define (getLimites numero)
  (if (< numero 3 ) (append '( 0 2))
      (if (< numero 6) (append '(3 5))
            (append '(6 8)))))
```

Esta función recibe un número del 0 al 8 que indica un número de fila o columna.
Esta función devuelve una lista cuya primera posición y segunda posición vendrá marcada por los límites inferior y superior de la fila o columna respectivamente.

Por ejemplo, si le pasas como parámetro un 4 indicando la fila número 4 la función te devuelve una lista con el 3 y el 5 ya que la fila 4 se encuentra entre el cuadrante que empieza en la fila 3 y acaba en la fila 5.

- **Función insertarValor**

```
(define (insertarValor fila columna valor sudoku)
  (cond ((equal? (or (comprobarFila fila valor sudoku)
                     (comprobarColumna columna valor sudoku)
                     (comprobarCuadrante fila columna valor sudoku)) #t)
         (printf "~a" "No se puede insertar el valor en la posicion recibida"))
        (else (list-set sudoku fila (list-set (getFila fila sudoku) columna valor)))))
```

Función que devuelve el sudoku resultante tras insertar un valor en el mismo en caso de que se pueda.

Esta función inserta un valor dentro de un sudoku siempre que se cumplan tres condiciones. Para poder insertar un valor, dicho valor no puede encontrarse en una fila, columna o cuadrante.

Por tanto, se utiliza una función equal que compara el resultado de un "or" con #t (true).

- Si el resultado del "or" es true, esto indica que el valor se encuentra en la fila, en la columna o en el cuadrante de la casilla donde se quiere insertar dicho valor. En ese caso, el valor no se inserta, ya que esto rompería las reglas del sudoku.
- En caso contrario, la función devuelve una lista que es igual a sudoku excepto el índice especificado por la fila y columna que tendrá el valor insertado, esto se consigue haciendo uso del método list-set.

- **Función posicionVacia**

```
(define (posicionVacia fila columna sudoku)
  (if (equal? 0 (getValorCasilla fila columna sudoku)) #t #f))
```

Esta función recibe como parámetros un número de fila, un número de columna y el sudoku. Compara el valor de la casilla dada con 0 devolviendo:

- #t en caso de que la casilla indicada por parámetros se encuentre vacía.
- En caso contrario la función devuelve #f.

- **Función listaSucesores**

```
(define (listaSucesores fila columna valor sudoku)
  (if (> valor 9) (append '())
      (if (equal? (or (comprobarFila fila valor sudoku)
                      (comprobarColumna columna valor sudoku)
                      (comprobarCuadrante fila columna valor sudoku)) #t)
          (listaSucesores fila columna (+ 1 valor) sudoku)
          (append (list(insertarValor fila columna valor sudoku))
                  (listaSucesores fila columna (+ 1 valor) sudoku)))))
```

Esta función recibe como parámetros un número de fila, un número de columna, un valor que por defecto siempre que se llame por primera vez a la función es 1 y un sudoku.
Se trata de una función recursiva que irá sacando los posibles sudokus para una casilla y añadiendoles a una lista.

El caso base de la función es comprobar si el valor es mayor que 9, en el caso de que así sea se añade una lista vacía y acaba la función devolviendo una lista con los posibles sudokus formados para esa casilla dada.

El cuerpo de la función recursiva se encarga de comprobar si en una fila, columna o cuadrante un valor dado es correcto mediante las respectivas funciones de comprobarFila, comprobarColumna y comprobarCuadrante. El objetivo por tanto, es devolver una lista con los posibles sudokus sucesores de una casilla y para ello:

- Si un valor no es correcto para una fila, columna y cuadrante se hace una llamada recursiva a la función con los mismos parámetros pero sumando uno al valor.
- Si un valor es correcto para una casilla en un sudoku se añade esa posible solución a la lista de sucesores y se hace una llamada recursiva a la función con los mismos parámetros pero sumando uno al valor.

- **Función siguienteVacia**

```
(define (siguienteVacia fila columna sudoku)
  (if (> fila 8) (append '())
      (if (> columna 8) (siguienteVacia (+ 1 fila ) 0 sudoku)
          (if( equal? 0 (getValorCasilla fila columna sudoku )) (append (list fila columna))
             (siguienteVacia fila (+ 1 columna) sudoku)))))
```

Función que devuelve una lista con la primera fila y columna que se encuentra vacía en el sudoku. Para ello, recibe una fila, una columna y un sudoku.

Esta función realizará un recorrido de forma recursiva a través del sudoku, empezando por la fila y la columna dadas y terminando cuando se encuentre una casilla vacía. En caso de que no exista ninguna casilla vacía, la función devuelve una lista vacía.

- **Función busquedaProfundidadImprimir**

```
(define (busquedaProfundidadImprimir sudoku listaAbiertos)
  (imprimir sudoku)
  (define fila
    (if (equal? '() (siguienteVacia 0 0 sudoku)) '()
        (car (siguienteVacia 0 0 sudoku))))
  (define columna
    (if (equal? '() (siguienteVacia 0 0 sudoku)) '()
        (cadr (siguienteVacia 0 0 sudoku))))
  (if (equal? '() (siguienteVacia 0 0 sudoku)) #t
      (if (equal? '() (listaSucesores fila columna 1 sudoku)) (busquedaProfundidadImprimir (car listaAbiertos)
                                                                                           (cdr listaAbiertos))
          (busquedaProfundidadImprimir (car (append (listaSucesores fila columna 0 sudoku) listaAbiertos))
                                       (cdr (append (listaSucesores fila columna 1 sudoku) listaAbiertos))))))
```

Esta función se encarga de realizar la búsqueda en profundidad para resolver el sudoku imprimiendo los pasos para conseguirlo. Para ello, recibe como parámetros el estado del sudoku, y la lista de nodos abiertos para llevar el control de la ejecución del algoritmo.

Para realizar esta búsqueda, primero se obtiene la posición de la primera casilla que está vacía y también se definen los valores "fila" y "columna" que se corresponden con la fila y la columna de la primera casilla vacía.

Tras esto, se realiza la función listaSucesores, la cual devuelve una lista con todos los nodos resultantes al añadir todos los valores posibles a dicha casilla vacía. En caso de que esta lista de sucesores esté vacía, esto significa que hay que hacer "backtracking" y volver a un nodo anterior ya que no se puede continuar por el nodo por donde se estaba resolviendo el sudoku. Por tanto, si la lista de sucesores está vacía, se realiza la recursividad, poniendo como sudoku a analizar en el siguiente paso, el primer valor de la lista de nodos abiertos. Asimismo, la lista de nodos abiertos en el siguiente paso, será el "cdr" de la lista de abiertos en el paso actual, ya que el primer valor de la lista de abiertos actual, va a pasar a ser el nodo analizado en el siguiente paso.

En caso de que la lista de sucesores no esté vacía, eso significa que si que existen nodos sucesores del nodo actual. Por tanto, se realiza la recursión llamando a la función con los parámetros:

- (car (append (listaSucesores fila columna 0 sudoku) listaAbiertos)): este parámetro es el nuevo sudoku a analizar. Como estamos en una búsqueda en profundidad, el siguiente nodo a analizar se corresponde con el primer nodo de la lista de sucesores, ya que la búsqueda en profundidad recorre el árbol de solución de manera recursiva mirando siempre los hijos del nodo actual antes que los hermanos del mismo.
- (cdr (append (listaSucesores fila columna 1 sudoku) listaAbiertos)): este parámetro se corresponde con la lista de nodos abiertos resultantes. Esta lista se compone de la lista de abiertos del nodo actual más los nuevos nodos sucesores, sin contar con el primero de los nodos sucesores, que va a pasar a ser el nuevo nodo actual (es el anterior parámetro).

De esta manera se consigue realizar una búsqueda en profundidad de manera recursiva que encuentra la solución de un sudoku en caso de que exista.

Por último, ya que el sudoku debe ser impreso cada vez que se realiza un paso, se llama a la función imprimir sudoku al principio de la función de busquedaEnProfundidadImprimir.

- **Función busquedaProfundidad**

```
(define (busquedaProfundidad sudoku listaAbiertos)
  (define t0 (current-seconds))
  (define fila
    (if (equal? '() (siguienteVacia 0 0 sudoku)) '()
                  (car (siguienteVacia 0 0 sudoku))))
  (define columna
    (if (equal? '() (siguienteVacia 0 0 sudoku)) '()
                    (cadr (siguienteVacia 0 0 sudoku))))
  (if (equal? '() (siguienteVacia 0 0 sudoku)) (imprimir sudoku)
      (if (equal? '() (listaSucesores fila columna 1 sudoku))
          (busquedaProfundidad (car listaAbiertos) (cdr listaAbiertos))
          (busquedaProfundidad (car (append (listaSucesores fila columna 0 sudoku) listaAbiertos))
                               (cdr (append (listaSucesores fila columna 1 sudoku) listaAbiertos))))))
```

Esta función se encarga de realizar la búsqueda en profundidad para resolver el sudoku. Para ello, recibe como parámetros el estado del sudoku, y la lista de nodos abiertos para llevar el control de la ejecución del algoritmo.

Para realizar esta búsqueda, primero se obtiene la posición de la primera casilla que está vacía. Se definen los valores fila y columna que se corresponden con la fila y la columna de la primera casilla vacía.

Tras esto, se realiza la función listaSucesores, la cual devuelve una lista con los nodos resultantes al añadir todos los valores posibles a dicha casilla vacía. En caso de que esta lista de sucesores esté vacía, esto significa que hay que hacer "backtracking" y volver a un nodo anterior ya que no se puede continuar por el nodo por donde se estaba resolviendo el sudoku. Por tanto, si la lista de sucesores está vacía, se realiza la recursividad, poniendo como sudoku a analizar en el siguiente paso, el primer valor de la lista de nodos abiertos. Asimismo, la lista de nodos abiertos en el siguiente paso, será el "cdr" de la lista de abiertos

en el paso actual, ya que el primer valor de la lista de abiertos actual, va a pasar a ser el nodo analizado en el siguiente paso.

En caso de que la lista de sucesores no esté vacía, eso significa que si que existen nodos sucesores del nodo actual. Por tanto, se realiza la recursión llamando a la función con los parámetros:

- (car (append (listaSucesores fila columna 0 sudoku) listaAbiertos)): este parámetro es el nuevo sudoku a analizar. Como estamos en una búsqueda en profundidad, el siguiente nodo a analizar se corresponde con el primer nodo de la lista de sucesores, ya que la búsqueda en profundidad recorre el árbol de solución de manera recursiva mirando siempre los hijos del nodo actual antes que los hermanos del mismo.
- (cdr (append (listaSucesores fila columna 1 sudoku) listaAbiertos)): este parámetro se corresponde con la lista de nodos abiertos resultantes. Esta lista se compone de la lista de abiertos del nodo actual más los nuevos nodos sucesores, sin contar con el primero de los nodos sucesores, que va a pasar a ser el nuevo nodo actual (es el anterior parámetro).

De esta manera se consigue realizar una búsqueda en profundidad de manera recursiva que encuentra la solución de un sudoku en caso de que exista. Por último imprime el sudoku solución.

- **Función busquedaAnchuraImprimir**

```
(define (busquedaAnchuraImprimir sudoku listaAbiertos)
  (imprimir sudoku)
  (define
    fila (if (equal? '() (siguienteVacia 0 0 sudoku)) '()
             (car (siguienteVacia 0 0 sudoku))))
  (define columna
    (if (equal? '() (siguienteVacia 0 0 sudoku)) '()
             (cadr (siguienteVacia 0 0 sudoku))))
  (if (equal? '() (siguienteVacia 0 0 sudoku)) #t
      (if (equal? '() (listaSucesores fila columna 1 sudoku))
          (busquedaAnchuraImprimir (car listaAbiertos) (cdr listaAbiertos))
          (busquedaAnchuraImprimir (car (append listaAbiertos (listaSucesores fila columna 0 sudoku) ))
                                   (cdr(append listaAbiertos (listaSucesores fila columna 0 sudoku) ))))))
```

Esta función se encarga de realizar la búsqueda en anchura para resolver el sudoku. Para ello, recibe como parámetros el sudoku y la lista de nodos abiertos para llevar el control de la ejecución del algoritmo.

Para realizar la búsqueda en anchura, hemos realizado una función recursiva cuyo caso base es que si la siguiente posición vacía es igual a una lista vacía entonces, termina la búsqueda y se devuelve el resultado.

Como cuerpo de la función recursiva, se realiza la función listaSucesores, la cual devuelve una lista con los nodos resultantes al añadir todos los valores posibles a dicha casilla vacía. En caso de que la lista de sucesores esté vacía, se llama a la función pponiendo como sudoku a analizar en el siguiente paso, el primer valor de la lista de nodos abiertos. Asimismo, la lista de nodos abiertos en el siguiente paso, será el "cdr" de la lista de abiertos en el paso

actual, ya que el primer valor de la lista de abiertos actual, va a pasar a ser el nodo analizado en el siguiente paso.

En caso de que la lista de sucesores no esté vacía, eso significa que si que existen nodos sucesores del nodo actual. Por tanto, se realiza la recursión llamando a la función con los parámetros:

- (car (append listaAbiertos (listaSucesores fila columna 0 sudoku))): este parámetro es el nuevo sudoku a analizar. Como estamos en una búsqueda en anchura, el siguiente nodo a analizar se corresponde con el primer nodo de la lista de abiertos, ya que la búsqueda en anchura recorre el árbol de solución de manera recursiva mirando siempre los hermanos del nodo actual antes que los hijos del mismo.
- (cdr (append listaAbiertos (listaSucesores fila columna 1 sudoku))): este parámetro se corresponde con la lista de nodos abiertos resultantes. Esta lista se compone de la lista de nodos abiertos del nodo actual (sin contar con el primero de la lista de nodos abiertos ya que este pasará a ser el nodo actual del siguiente paso) más los nuevos nodos sucesores.

De esta manera se consigue realizar una búsqueda en anchura de manera recursiva que encuentra la solución de un sudoku en caso de que exista.
En este caso, como la función debe imprimir cada paso de la resolución del sudoku, cada vez que se llama a esta función, se llama a imprimir sudoku, que se encarga de imprimir el mismo.

- **Función busquedaAnchura**

```
(define (busquedaAnchura sudoku listaAbiertos)
  (define fila (if (equal? '() (siguienteVacia 0 0 sudoku)) '()
                   (car (siguienteVacia 0 0 sudoku))))
  (define columna (if (equal? '() (siguienteVacia 0 0 sudoku)) '()
                      (cadr (siguienteVacia 0 0 sudoku))))
  (if (equal? '() (siguienteVacia 0 0 sudoku)) (imprimir sudoku)
      (if (equal? '() (listaSucesores fila columna 1 sudoku))
          (busquedaAnchura (car listaAbiertos) (cdr listaAbiertos))
          (busquedaAnchura (car (append listaAbiertos (listaSucesores fila columna 0 sudoku)))
                           (cdr (append listaAbiertos (listaSucesores fila columna 0 sudoku)))))))
```

Esta función se encarga de realizar la búsqueda en anchura para resolver el sudoku.
Para ello, recibe como parámetros el estado del sudoku, y la lista de nodos abiertos para llevar el control de la ejecución del algoritmo.

Para realizar esta búsqueda, primero se obtiene la posición de la primera casilla que está vacía. Se definen los valores fila y columna que se corresponden con la fila y la columna de la primera casilla vacía.

Tras esto, se realiza la función listaSucesores, la cual devuelve una lista con los nodos resultantes al añadir todos los valores posibles a dicha casilla vacía. En caso de que esta lista de sucesores esté vacía, se realiza la recursividad, poniendo como sudoku a analizar en el siguiente paso, el primer valor de la lista de nodos abiertos. Asimismo, la lista de nodos abiertos en el siguiente paso, será el "cdr" de la lista de abiertos en el paso actual, ya que el

primer valor de la lista de abiertos actual, va a pasar a ser el nodo analizado en el siguiente paso.

En caso de que la lista de sucesores no esté vacía, eso significa que si que existen nodos sucesores del nodo actual. Por tanto, se realiza la recursión llamando a la función con los parámetros:

- (car (append listaAbiertos (listaSucesores fila columna 0 sudoku))): este parámetro es el nuevo sudoku a analizar. Como estamos en una búsqueda en anchura, el siguiente nodo a analizar se corresponde con el primer nodo de la lista de abiertos, ya que la búsqueda en anchura recorre el árbol de solución de manera recursiva mirando siempre los hermanos del nodo actual antes que los hijos del mismo.
- (cdr (append listaAbiertos (listaSucesores fila columna 1 sudoku))): este parámetro se corresponde con la lista de nodos abiertos resultantes. Esta lista se compone de la lista de nodos abiertos del nodo actual (sin contar con el primero de la lista de nodos abiertos ya que este pasará a ser el nodo actual del siguiente paso) más los nuevos nodos sucesores.

De esta manera se consigue realizar una búsqueda en anchura de manera recursiva que encuentra la solución de un sudoku en caso de que exista. Por último imprime el sudoku solución.

- **Función comprobarSudokuValido**

```
(define (comprobarSudokuValido sudoku)
  (for/and ([fila 9])
    (for/and ([columna 9])
      (if (equal? (getValorCasilla fila columna sudoku) 0) #t
          (if (or (comprobarSiRepetido (getValorCasilla fila columna sudoku) (getFila fila sudoku))
                  (comprobarSiRepetido (getValorCasilla fila columna sudoku)  (getColumna fila columna sudoku))
                  (comprobarSiRepetido (getValorCasilla fila columna sudoku)
                                       (getCuadradante fila columna sudoku (getLimites fila) (getLimites columna)))
                  (menorQue17 0 0 (expt dimension 2) sudoku)) #f #t)))))
```

Esta función recibe como parámetro un sudoku y devuelve true o false dependiendo si el sudoku es válido o no. Para ello empieza en un bucle for/and dentro de otro bucle for/and que le sirve para evaluar todas las casillas del sudoku. Después comprueba si la casilla en la que está es un cero o no, en caso de serlo devuelve true y continúa el bucle, en caso contrario comprueba si el valor de la casilla actual se vuelve a repetir en la fila, columna o el cuadrante mediante el método comprobarSiRepetido al que le pasas como parámetros el valor de la casilla actual mediante el método getValorCasilla y la fila columna o cuadrado en el que te encuentras mediante los métodos getFila, getColumna y getCuadrado respectivamente. En caso de que se repita el valor significará que el sudoku no es válido por lo que el if devuelve un false que hace que se salga del bucle for/and y la función devuelva false indicando que el sudoku no es válido, en caso contrario, es decir, que el valor no esté repetido ni en la fila ni en la columna ni en el cuadrante el if devolverá true y continuará el bucle for/and hasta que se encuentre una casilla no válida o hasta que llegue al final y acabe devolviendo true indicando que el sudoku es válido.

- **Función pistasIniciales**

```
(define (pistasIniciales fila columna contador sudoku)
  (if (> fila 8) contador
      (if (> columna 8) (pistasIniciales (+ fila 1) 0 contador sudoku)
          (if (equal? 0 (getValorCasilla fila columna sudoku))
              (pistasIniciales fila (+ 1 columna) (- contador 1) sudoku)
              (pistasIniciales fila (+ 1 columna) contador sudoku)))))
```

Esta función recibe como parámetro un número de fila (fila 0 por defecto), un número de columna (columna 0 por defecto) , un contador (que vale 18 por defecto) y un sudoku. Su objetivo es decrementar el contador cada vez que se encuentra un cero en el sudoku y devolver el número resultante. Empieza mirando si el número de fila es mayor que 8, ya que en el caso de que lo sea significa que ha llegado al final del sudoku y devuelve el contador, en caso contrario comprueba si el número de columna es mayor que 8, si es mayor que 8 hace una llamada recursiva a la función pasando los mismos parámetros pero aumentando en uno el número de fila y poniendo el número de columna en 0, en caso contrario mira si la casilla en la que se encuentra es un 0, si es así llama recursivamente a la función con la siguiente casilla y decrementando en uno el valor de contador, en caso contrario llama recursivamente a la función con la siguiente casilla y el resto de parámetros igual.

- **Función menorQue17**

```
(define (menorQue17 fila columna contador sudoku )
  (define pistas (pistasIniciales fila columna contador sudoku))
  (if (< pistas 17) #t #f))
```

Esta función recibe como parámetros un número de fila (fila 0 por defecto), un número de columna (columna 0 por defecto), un contador (que vale 18 por defecto) y un sudoku. Esta función llama a la función pistasIniciales con los mismos parámetros con los que es llamada y comprueba si el número que devuelve es menor que 17, en caso de que así sea devuelve true indicando que el sudoku no es válido, ya que un sudoku no puede tener un número menor que 17 de números que no son 0, en caso contrario devuelve false indicando que el sudoku es apto.

- **Función comprobarSiRepetido**

```
(define (comprobarSiRepetido valor lista)
  ( if (equal? (member valor (remove valor lista)) #f) #f #t))
```

Esta función recibe como parámetro un número y una lista y devuelve true en caso de que ese número se encuentre en esa lista y false en caso contrario. Para ello usa la función member que devuelve false si un valor se encuentra en una lista, para ello le pasamos el valor dado como parámetro y las lista dada por parámetro pero borrando el valor de esa lista (esto se hace para ver únicamente si el valor sale más de una vez en la lista). El resultado de esta función se compara mediante la función equal con un #f y en caso de ser

ambos false significa que el valor no se repite en la lista y la función devuelve false, en cualquier otro caso la función devuelve true.

- **Función getCuadradante**

```
(define (getCuadradante fila columna sudoku limitesFila limitesColumna)
  (if (> fila (cadr limitesFila)) (append '())
      (if ( > columna (cadr limitesColumna ))
          (getCuadradante (+ 1 fila)(car limitesColumna) sudoku limitesFila limitesColumna)
          (append ( list (getValorCasilla fila columna sudoku))
                  (getCuadradante fila (+ 1 columna) sudoku limitesFila limitesColumna)))))
```

Esta función recibe como parámetros un número de fila, un número de columna, un sudoku, una lista con los límites de la fila y una lista con los límites de la columna. Lo primero que hace es comprobar si el número de fila es mayor que el límite superior del cuadrante, en caso de que así sea significará que la función ha acabado y devolverá una lista con los números del cuadrante, en caso contrario comprueba si el número de la columna es mayor que el límite superior del cuadrante , en caso de que así sea significa que se ha pasado al siguiente cuadrante y se hace una llamada recursiva a la función pasándole como parámetros la fila siguiente, la columna que marca el límite inferior del cuadrante y los límites de la fila y la columna, en caso contrario se añade el valor de la casilla a una lista a la que también se añade el valor que devuelva la llamada recursiva a la función con parámetros la fila actual, la siguiente columna y los límites de la fila y la columna.

- **Función resolverSudoku**

```
(define (resolverSudoku sudoku imprimir metodo)
  (if (equal? imprimir #t)
      (if (equal? metodo "anchura") (busquedaAnchuraImprimir sudoku '())
          (if (equal? metodo "profundidad")(busquedaProfundidadImprimir sudoku '())
              (display "Metodo incorrecto")))
      (if (equal? imprimir #f)
          (if (equal? metodo "anchura") (busquedaAnchura sudoku '())
          (if (equal? metodo "profundidad")(busquedaProfundidad sudoku '())
              (display "Metodo incorrecto")))
          (display "Debes indicar mediante #t o #f si quieres imprimir los pasos o no"))))
```

Esta función recibe como parámetros el sudoku, un booleano puesto a true si quieres que la solución del sudoku se imprima paso a paso o puesto a false si únicamente quieres la solución final y el método que puede ser “anchura” si quieres utilizar el método de búsqueda en anchura o “profundidad” si quieres utilizar el método de búsqueda en profundidad. La función mediante if’s y el método equal va comparando estos parámetros para devolver la búsqueda seleccionada o un mensaje de error en caso de que uno de los parámetros sea incorrecto.

4. Resumen y conclusiones

Una vez realizada la implementación de los dos métodos de búsqueda se ha llegado a la conclusión de que en la mayoría de los casos el método búsqueda en profundidad es más rápido calculando la solución del Sudoku dado que el método de búsqueda en anchura.

La búsqueda en anchura es recomendable en problemas en los que se necesita encontrar el camino más corto para encontrar una solución al problema. Sin embargo, la búsqueda en profundidad busca la solución siguiendo una rama y haciendo backtracking en caso de que no encuentre la solución por ese camino.

Al ser éste un problema en el que nos da igual el coste y que además estamos resolviendo un problema mediante una búsqueda no informada entonces, la búsqueda en profundidad es más eficaz que la búsqueda en anchura ya que no necesita generar todos y cada uno de los posibles sudokus para cada casilla, si no que genera un posible sudoku para una casilla y va siguiendo por ese camino, volviendo para atrás en el caso de que llegue a un camino erróneo o sin solución.

Dado que el problema del sudoku es un problema con nodos limitados, es decir, los nodos que se pueden comprobar no son infinitos, (ya que el sudoku tiene un 9x9 casillas que contienen un valor del 1 al 9) el problema está acotado. Cuando un problema como este está acotado, la búsqueda en profundidad siempre va a llegar a la solución si esta existe. Si a esto le sumamos que en este problema el coste de la resolución es insignificante, el resultado es que la búsqueda en profundidad es mucho más adecuada que la búsqueda en anchura para resolver sudokus.

5. Código Racket

Archivo Sudoku.rkt

```
#lang racket

#|=========================================================================
 Práctica 1 - EL ROMPECABEZAS DE SUDOKU
 Inteligencia Artificial
 Curso 2019/20
 Universidad de Alcalá
 Javier García Jiménez, David Moreno López, Isabel Martínez Gómez
 =========================================================================|#
(provide (all-defined-out))

(define dimension 9)

(define test1
  '((5 0 0 0 0 0 0 0 0)
    (0 2 8 4 0 0 5 0 3)
    (1 0 0 2 7 0 0 0 6)
    (0 0 3 0 5 2 1 9 0)
    (7 0 6 0 1 0 2 0 8)
    (0 1 9 7 4 0 3 0 0)
    (6 0 0 0 9 4 0 0 2)
    (8 0 1 0 0 6 7 5 0)
    (0 0 0 0 0 0 0 0 4)))

(define test19
  '((0 0 0 0 0 8 0 0 0)
    (3 0 8 0 0 0 7 0 0)
    (2 9 4 7 0 0 0 8 1)
    (4 0 0 8 6 0 0 7 0)
    (0 0 0 0 0 0 0 0 0)
    (0 7 0 0 9 2 0 0 0)
    (6 5 0 0 0 1 2 4 3)
    (0 0 3 0 0 0 8 0 9)
    (0 0 0 5 0 0 0 0 0)))

(define sudokuMasDificilDelMundo
  '((8 0 0 0 0 0 0 0 0)
    (0 0 3 6 0 0 0 0 0)
    (0 7 0 0 9 0 2 0 0)
    (0 5 0 0 0 7 0 0 0)
    (0 0 0 0 4 5 7 0 0)
    (0 0 0 1 0 0 0 3 0)
    (0 0 1 0 0 0 0 6 8)
```

```
   (0 0 8 5 0 0 0 1 0)
   (0 9 0 0 0 0 4 0 0)))

;Imprime el formato del sudoku
(define (imprimir sudoku)
  (for*([fila dimension]
        [columna dimension])
    (when (and (equal? 0 (remainder fila 3)) (= columna 0))
      (printf "+-------+-------+-------+\n"))
    (when(equal?(remainder columna 3) 0)
      (printf "~a" "| "))
    (printf "~a" (getValorCasilla fila columna sudoku))
    (printf " ")
    (when (= columna 8) (printf "~a" "|\n"))
    (when (and (= fila 8) (= columna 8))(printf  "+-------+-------+-------+\n\n"))))

;Devuelve el valor de una casilla dada la fila, la columna y el sudoku
(define (getValorCasilla fila columna sudoku)
  (list-ref (list-ref sudoku fila) columna))

;Comprueba si existe un valor dado en un cuadrante
(define (comprobarCuadrante fila columna valor sudoku)
  (define x (remainder fila 3))
  (define y (remainder columna 3))
  (define a (- fila x))
  (define b (- columna y))
  (for/or ([i (/ dimension 3)])
    (for/or  ([j (/ dimension 3)])
      (equal? (getValorCasilla (+ a i) (+ b j) sudoku) valor))))

;Comprueba si un valor existe en una fila dada
(define (comprobarFila fila valor sudoku)
  (for/or ([columna dimension])
    (equal? (getValorCasilla fila columna sudoku) valor)))

;Comprueba si un valor dado existe en una columna dada
(define (comprobarColumna columna valor sudoku)
  (for/or ([fila dimension])
    (equal? (getValorCasilla fila columna sudoku) valor)))

;Devuelve una fila como una lista
(define (getFila fila sudoku)
  (list-ref sudoku fila))

;Devuelve una columna como una lista
(define (getColumna fila columna sudoku)
  (if (> fila 8 ) (append '())
      (append (list (getValorCasilla fila columna sudoku)) (getColumna (+ 1 fila) columna sudoku))))

;Calcula los limites de una fila o columna en un cuadrante
(define (getLimites numero)
```

```
 (if (< numero 3 ) (append '( 0 2))
   (if (< numero 6) (append '(3 5))
       (append '(6 8)))))

;inserta un valor, en caso de que se pueda, en una posicion dada de un sudoku
(define (insertarValor fila columna valor sudoku)
 (cond ((equal? (or (comprobarFila fila valor sudoku)
             (comprobarColumna columna valor sudoku)
             (comprobarCuadrante fila columna valor sudoku)) #t)
     (printf "~a" "No se puede insertar el valor en la posicion recibida"))
    (else (list-set sudoku fila (list-set (getFila fila sudoku) columna valor)))))

;Comprueba si una posicion esta vacia
(define (posicionVacia fila columna sudoku)
 (if (equal? 0 (getValorCasilla fila columna sudoku)) #t #f))

;Devuelve una lista con los posibles sudokus que puede haber insertando los posibles valores de una
determinada casilla
(define (listaSucesores fila columna valor sudoku)
 (if (> valor 9) (append '())
   (if (equal? (or (comprobarFila fila valor sudoku)
             (comprobarColumna columna valor sudoku)
             (comprobarCuadrante fila columna valor sudoku)) #t)
      (listaSucesores fila columna (+ 1 valor) sudoku)
      (append (list (insertarValor fila columna valor sudoku))
          (listaSucesores fila columna (+ 1 valor) sudoku)))))

;Busca la siguiente posicion vacia de una fila y columna dada en un sudoku
(define (siguienteVacia fila columna sudoku)
 (if (> fila 8) (append '())
   (if (> columna 8) (siguienteVacia (+ 1 fila ) 0 sudoku)
     (if(equal? 0 (getValorCasilla fila columna sudoku )) (append (list fila columna))
       (siguienteVacia fila (+ 1 columna) sudoku)))))

;Imprime paso a paso la resolucion del sudoku mediante busqueda en profundidad con backtracking
(define (busquedaProfundidadImprimir sudoku listaAbiertos)
 (imprimir sudoku)
 (define fila
  (if (equal? '() (siguienteVacia 0 0 sudoku)) '()
    (car (siguienteVacia 0 0 sudoku))))
 (define columna
  (if (equal? '() (siguienteVacia 0 0 sudoku)) '()
    (cadr (siguienteVacia 0 0 sudoku))))
 (if (equal? '() (siguienteVacia 0 0 sudoku)) #t
   (if (equal? '() (listaSucesores fila columna 1 sudoku)) (busquedaProfundidadImprimir (car
listaAbiertos)
                                                (cdr listaAbiertos))
      (busquedaProfundidadImprimir (car (append (listaSucesores fila columna 0 sudoku)
listaAbiertos))
                    (cdr (append (listaSucesores fila columna 1 sudoku) listaAbiertos))))))
```

```
;Imprime la solucion del sudoku echa mediante busqueda en profundidad con backtracking
(define (busquedaProfundidad sudoku listaAbiertos)
 (define t0 (current-seconds))
 (define fila
  (if (equal? '() (siguienteVacia 0 0 sudoku)) '()
          (car (siguienteVacia 0 0 sudoku))))
 (define columna
  (if (equal? '() (siguienteVacia 0 0 sudoku)) '()
           (cadr (siguienteVacia 0 0 sudoku))))
 (if (equal? '() (siguienteVacia 0 0 sudoku)) (imprimir sudoku)
   (if (equal? '() (listaSucesores fila columna 1 sudoku))
     (busquedaProfundidad (car listaAbiertos) (cdr listaAbiertos))
     (busquedaProfundidad (car (append (listaSucesores fila columna 0 sudoku) listaAbiertos))
                 (cdr (append (listaSucesores fila columna 1 sudoku) listaAbiertos))))))

;Imprime paso a paso la solucion encontrada del sudoku por busqueda en anchura
(define (busquedaAnchuraImprimir sudoku listaAbiertos)
 (imprimir sudoku)
 (define
  fila (if (equal? '() (siguienteVacia 0 0 sudoku)) '()
       (car (siguienteVacia 0 0 sudoku))))
 (define columna
  (if (equal? '() (siguienteVacia 0 0 sudoku)) '()
       (cadr (siguienteVacia 0 0 sudoku))))
 (if (equal? '() (siguienteVacia 0 0 sudoku)) #t
   (if (equal? '() (listaSucesores fila columna 1 sudoku))
     (busquedaAnchuraImprimir (car listaAbiertos) (cdr listaAbiertos))
     (busquedaAnchuraImprimir (car (append listaAbiertos (listaSucesores fila columna 0 sudoku) ))
                 (cdr(append listaAbiertos (listaSucesores fila columna 0 sudoku) ))))))

;Imprime la solucion encontrada del sudoku por busqueda en anchura
(define (busquedaAnchura sudoku listaAbiertos)
 (define fila (if (equal? '() (siguienteVacia 0 0 sudoku)) '()
          (car (siguienteVacia 0 0 sudoku))))
 (define columna (if (equal? '() (siguienteVacia 0 0 sudoku)) '()
           (cadr (siguienteVacia 0 0 sudoku))))
 (if (equal? '() (siguienteVacia 0 0 sudoku)) (imprimir sudoku)
   (if (equal? '() (listaSucesores fila columna 1 sudoku))
     (busquedaAnchura (car listaAbiertos) (cdr listaAbiertos))
     (busquedaAnchura (car (append listaAbiertos (listaSucesores fila columna 0 sudoku)))
               (cdr (append listaAbiertos (listaSucesores fila columna 0 sudoku)))))))

;Comprueba si un sudoku dado es valido
(define (comprobarSudokuValido sudoku)
 (for/and ([fila 9])
  (for/and ([columna 9])
   (if (equal? (getValorCasilla fila columna sudoku) 0) #t
     (if (or (comprobarSiRepetido (getValorCasilla fila columna sudoku) (getFila fila sudoku))
         (comprobarSiRepetido (getValorCasilla fila columna sudoku)  (getColumna fila columna
sudoku))
         (comprobarSiRepetido (getValorCasilla fila columna sudoku)
```

```
                    (getCuadradante fila columna sudoku (getLimites fila) (getLimites columna)))
          (menorQue17 0 0 (expt dimension 2) sudoku)) #f #t)))))

;Devuelve el numero de valores que no son cero de un sudoku dado
(define (pistasIniciales fila columna contador sudoku)
 (if (> fila 8) contador
    (if (> columna 8) (pistasIniciales (+ fila 1) 0 contador sudoku)
      (if (equal? 0 (getValorCasilla fila columna sudoku))
        (pistasIniciales fila (+ 1 columna) (- contador 1) sudoku)
        (pistasIniciales fila (+ 1 columna) contador sudoku)))))

;Devuelve #t si el numero de pistas de un sudoku inicial es menor que 17
(define (menorQue17 fila columna contador sudoku )
 (define pistas (pistasIniciales fila columna contador sudoku))
 (if (< pistas 17) #t #f))

;Comprueba si se repite un valor
(define (comprobarSiRepetido valor lista)
 ( if (equal? (member valor (remove valor lista)) #f) #f #t)) ; devuelve #t si un valor esta repetido en
una lista

;Devuelve el cuadrante al que pertenece una casilla
(define (getCuadradante fila columna sudoku limitesFila limitesColumna)
 (if (> fila (cadr limitesFila)) (append '())
    (if ( > columna (cadr limitesColumna ))
      (getCuadradante (+ 1 fila)(car limitesColumna) sudoku limitesFila limitesColumna)
      (append ( list (getValorCasilla fila columna sudoku))
          (getCuadradante fila (+ 1 columna) sudoku limitesFila limitesColumna)))))

;Funcion definitiva para resolver el sudoku
(define (resolverSudoku sudoku imprimir metodo)
 (if (equal? imprimir #t)
    (if (equal? metodo "anchura") (busquedaAnchuraImprimir sudoku '())
      (if (equal? metodo "profundidad")(busquedaProfundidadImprimir sudoku '())
        (display "Metodo incorrecto")))
    (if (equal? imprimir #f)
      (if (equal? metodo "anchura") (busquedaAnchura sudoku '())
      (if (equal? metodo "profundidad")(busquedaProfundidad sudoku '())
        (display "Metodo incorrecto")))
      (display "Debes indicar mediante #t o #f si quieres imprimir los pasos o no"))))
```

Archivo testUnitarios.rkt

```
#lang racket
(require (file "Sudoku.rkt"))


(getValorCasilla 5 6 test1) ;devuelve 3
(comprobarCuadrante 2 3 4 test1) ;devuelve #t porque el 4 ya se encontraba en el cuadrante
(comprobarCuadrante 2 3 1 test1) ;devuelve #f porque el 1 no se encunetra en el cuadrante
(comprobarFila 3 5 test1) ;devuelve #t porque el 5 se encuentra en la fila 3
(comprobarFila 3 7 test1) ;devuelve #f porque el 7 no se encuentra en la fila 3
(comprobarColumna 8 6 test1) ;devuelve #t porque el 6 se encuentra en la columna 8
(comprobarColumna 8 1 test1) ;devuelve #f porque el 1 no se encuentra en la columna 8
(getFila 0 test1) ;devuelve la fila 0 del sudoku pasado
(getColumna 0 0 test1) ;devuelve la columna 0 del sudoku pasado
(getLimites 6) ;devuelve '(6 8) porque son los límites inferior y superior del cuadrante de la fila o columna 6
(insertarValor 0 1 3 test1) ;devuelve un sudoku con el valor 3 insertado en la fila 0 columna 1 siempre y cuando sea posible insertarlo
(posicionVacia 1 4 test1) ;devuelve #t porque en la fila 1 columna 4 hay un 0
(posicionVacia 0 0 test1) ;devuelve #f porque en la fila 0 columna 0 no hay un 0
(listaSucesores 3 1 1 test1) ;devuelve una lista con dos posibles sudokus para la casilla de la fila 3 columna 1
(siguienteVacia 2 3 test1) ;devuelve '(2 5) que es la siguiente casilla vacía a la dada en la fila 2 columna 3
(busquedaProfundidadImprimir test1 '()) ;imprime paso a paso la resolución mediante búsqueda por profundidad con bracktracking del sudoku
(busquedaProfundidad test1 '()) ;imprime la solución del sudoku resuelto mediante búsqueda por profundidad con backtracking
(busquedaAnchuraImprimir test1 '()) ;imprime paso a paso la resolución mediante búsqueda por anchura del sudoku
(busquedaAnchura test1 '()) ;imprime la solución del sudoku resuelto mediante búsqueda por anchura
(comprobarSudokuValido test1); devuelve #t porque el sudoku es válido
(pistasIniciales 0 0 81 test1);devuelve 35 ya que el sudoku definido como "test1" tiene 35 pistas iniciales
(menorQue17 0 0 81 test1) ;devuelve #f ya que el sudoku pasado por parámetros tiene más de 17 valores
(comprobarSiRepetido 5 '(1 2 3 5 4 6 5)) ;devuelve #t porque el 5 se encuentra más de una vez en la lista
(comprobarSiRepetido 5 '(1 2 3 5 4 6)) ;devuelve #f porque el 5 no se encuentra más de una vez en la lista
(getCuadradante 3 3 test1 '(3 5) '(3 5)) ;devuelve una lista con los números del cuadrante a partir de la posición de la fila 3 columna 3
(resolverSudoku test1 #f "profundidad") ;imprime la solución del sudoku mediante el método profundidad
(resolverSudoku test1 #t "anchura") ;imprime paso a paso la resolución del problema del sudoku mediante el método anchura
```