DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Final

Resolviendo Sudokus usando un Algoritmo Genético

Metaheurísticas
Segundo Cuatrimestre de 2017

Integrante	LU	Correo electrónico
Maddonni, Axel	200/14	axel.maddonni@gmail.com
Patané, Federico	683/10	fedepatane20@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Contents

1 Introducción

1.1 Descripción del Problema

En este trabajo estudiaremos si los Sudokus pueden ser resueltos eficientemente usando una heurística basada en un algoritmo genético, un método de optimización que imita la evolución Darwiniana que ocurre en la naturaleza.

El Sudoku es un juego lógico de origen japonés que se ha vuelto muy popular en los últimos tiempos debido a que es muy desafiante y tiene reglas muy simples. Consta de un tablero de 9x9 posiciones, divididos en 9 subcuadrados de 3x3. El objetivo del mismo es completar, dado un tablero y algunas posiciones iniciales ya marcadas (que permanecen estáticas), los siguientes ítems:

- Completar cada uno de los subcuadrados desde 1 a 9, sin repetir.
- Completar cada una de las filas desde 1 a 9, sin repetir.
- Completar cada una de las columnas de 1 a 9, sin repetir.

El nivel de dificultad de un Sudoku no solo está determinado por la cantidad de elementos iniciales ya marcados [7], sino también por al menos otros 20 factores, como explican en su paper Mantere y Koljonen [1]. Además, fue reportado que el número total de sudokus únicos de 9x9 que pueden ser generados es de $6,670,903,752,021,072,936,960 (\sim 6.6710^{21})$, cada uno con una solución única [8].

Computacionalmente, el problema de resolver sudokus de nxn fue probado ser NP-Completo [6]. Sin embargo, existen algoritmos exactos que pueden resolverlo eficientemente como el algoritmo de "Dancing Links". En este trabajo nos enfocaremos en trabajar con metaheurísticas, en particular algoritmos genéticos, y comparar resultados con otros algoritmos genéticos y otras metaheurísticas.

1.2 Algoritmos Genéticos

Los algoritmos genéticos se basan en la evolución Darwiniana, y son construidos como una analogía al proceso evolutivo. Suelen ser útiles en problemas de optimización donde se realizan búsquedas en espacios muy amplios y poco definidos. Fueron introducidos por John Holland en los 60s, y son actualmente aplicados a muchos tipos de problemas.

En un algoritmo genético, las soluciones posibles a los problemas son codificadas como individuos de una población. Cada individuo representa un cromosoma compuesto por muchos genes. Estos individuos son testeados contra nuestro problema usando una función de **fitness**, que define cuán buena es la solución según algún criterio relacionado con el problema. Cuanto mejor sea el fitness de la solución, mayores son las chances de ser seleccionado como un padre para generar nuevos individuos, y continuar así el proceso evolutivo. Los peores individuos se eliminan de la población para hacer lugar para la nueva generación de individuos.

Usando **crossover** y **mutaciones** el algoritmo genera los nuevos individuos. En el crossover, se combinan los genes de dos padres usando algún método preseleccionado, como por ejemplo crossover de 1 punto (elegir un único punto en los padres y dividir por ese punto) o crossover uniforme (los genes son seleccionados aleatoriamente entre los padres). En la mutación, algunos genes random son mutados usando alguna estrategia. De esta manera, este tipo de algoritmos sigue el principio de "supervivencia de los mejores".

El algoritmo continúa hasta que se llega al mejor individuo posible (que maximice la función de fitness), o cuando se llegue a un criterio de parada. Sin embargo, para utilizar este tipo de algoritmos hay determinadas circunstancias que deben tenerse en cuenta.

Si la función a optimizar tiene muchos máximos/mínimos locales se requerirán más iteraciones del algoritmo para alcanzar el máximo/mínimo global, esto puede provocar que el algoritmo tarde demasiado en converger. Además, puede que se "estanque" en uno de estos máximos/mínimos locales, del cual no pueda volver a salir.

También, otro punto fundamental es formular un diseño eficiente del algoritmo para un problema en particular. En caso de no hacerlo de manera inteligente, el algoritmo puede llegar a tener un consumo de memoria muy grande, y volverse aún más lento.

```
Empezar
t := 0
inicializar P(t)
evaluar P(t)
Mientras no se cumpla la condición de parada hacer
    Empezar
        t:= t + 1
        construir P (t) a partir de P( t-1)
        modificar P(t)
        evaluar P (t)
    Fin
Fin
```

Esquema general para algoritmos genéticos

1.2.1 Ejemplos de Aplicación

Algunos de los campos donde se aplican este tipo de algoritmos son:

- Análisis lingüístico, incluyendo inducción gramática, y otros aspectos de procesamiento de lenguajes naturales, tales como eliminación de ambigüedad de sentido.
- Problema del viajante de comercio, dilema del prisionero.
- Organización de la infraestructura en redes de comunicaciones.
- Asignación de recursos de manera óptima, como por ejemplo profesores con horarios a materias. También, en asignación de tripulaciones a vuelos, o enfermeras a turnos de hospitales.
- En bioinformática, alineamiento múltiple de secuencias o predicción de estructura de ARN.
- Problemas de clustering.
- Etcétera

En particular, uno de los ejemplos vistos en clase que nos llamó la atención fue la aplicación en Generación Automática de Tests. Para resolverlo, utilizaban una población compuesta por posibles configuraciones de las variables en una porción de código, para evaluar las distintas ramas de ejecución del mismo, usando funciones de fitness como "Branch Distance" (distancia hasta ejecutar un predicado de una de las ramas), rank-selection para la selección de padres, Single-Point crossover y Bit-Flip Mutation.

1.2.2 Otras Variantes

Una variante del mismo puede ser el uso de **elitismo**. Este proceso permite que las mejores soluciones sobrevivan en lo que es la creación de la nueva generación.

Los algoritmos genéticos con **parámetros adaptativos** es otra de las variantes. No mantienen el valor de fitness fijo para un individuo, sino que este mismo se puede ir adaptando por diferentes motivos, uno de los cuales puede ser la longevidad. Esto hace posible que soluciones consideradas "buenas" que se encuentren estancadas en un mínimo local, puedan perder valor de fitness a lo largo de las iteraciones, lo cual nos permite descartarla para mantener la población distribuida de una mejor manera, con mas diversidad.

Otra de las variantes del mismo puede ser la implementación del algoritmo de manera **paralela**. Esto surge como una ventaja para intentar resolver problemas de gran complejidad computacional.

2 Nuestra Implementación

2.1 Interpretación del Problema

El problema del Sudoku puede entenderse como un problema de coloreo, donde cada nodo (posición del tablero) estará conectado con sus vecinos de su cuadrante, los de su fila y los de su columna. Una vez armado el grafo de esta manera, se busca encontrar un 9-coloreo válido sobre el mismo.

Esta interpretación nos ayudó a plantear nuevas ideas o heurísticas para abordar el problema. Es por eso también que en varias secciones del informe referenciamos a las posiciones del tablero como "nodos" y a los números de 1 al 9 como "colores" indistintamente.

Además, hicimos una investigación sobre otros trabajos realizados sobre algoritmos genéticos para resolver Sudokus para poder comparar e idear nuestro algoritmo, aprovechando ideas de otras implementaciones. Estos se encuentran disponibles en la sección de Bibliografía y serán referenciados a lo largo de informe.

2.2 Algoritmo Genético

2.2.1 Población

Representamos un individuo (cromosoma) con un array de 81 posiciones, que representan las 81 posiciones del tablero en el orden de las filas. Las posiciones del tablero que ya vienen definidas se setean en el array y no pueden ser modificadas a lo largo del algoritmo. Las posiciones libres se marcan en el input del algoritmo con un 0. La implementación de estos individuos se realizó con una clase llamada `SudokuChromosome`.

Nuestro algoritmo trabaja con una población de **100 individuos**. A diferencia de la resolución propuesta en [1], donde utilizan sólo 21 individuos, nos inclinamos por mantener una población de mayor tamaño para asegurar una mayor diversidad y disminuir las chances de caer en un mínimo local. Algo similar plantean en el trabajo [2].

2.2.2 Inicialización

Antes de la primera iteración del algoritmo, inicializamos la población de una manera pseudo-aleatoria, es decir, incluimos algo de inteligencia en la manera de armar las soluciones iniciales.

Para cada subcuadrado, se completa el sudoku con números del 1 al 9 random sin repetir, es decir, todos los subcuadrados deben cumplir la condición de mantener sus 9 valores distintos, sin importar los demás valores de las filas y columnas.

A medida que avancen las generaciones del algoritmo, este invariante se mantendrá inviolable. Es decir, durante toda la ejecución del algoritmo los subcuadrados cumplirán esta condición. Esto se observará con detalle cuando mencionemos los operadores de crossover y mutación. Además, es importante mencionar que las posiciones "fijas" (que vienen en el input) tampoco se modifican durante la ejecución del algoritmo.

2.2.3 Fitness

La función de fitness está definida por la cantidad de ejes que conectan dos nodos que contienen el mismo color. Una manera eficiente de calcular este valor es realizando el siguiente cálculo:

$$A = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$Fitness_{filas} = \sum_{n=1}^{9} |A - F_i|$$

$$Fitness_{cols} = \sum_{n=1}^{9} |A - C_i|$$

$$Fitness = Fitnes_{filas} + Fitness_{cols}$$

donde:

- A es la lista de números disponibles
- F_i y C_i son los conjuntos de números ubicados en la i-ésima fila y columna respectivamente

Notar que no es necesario sumar el fitness por zona (subcuadrados) ya que siempre es 0 debido a que siempre se mantienen todos los números distintos, como se mencionó anteriormente.

El objetivo del algoritmo será, en este caso, **minimizar** el fitness de las soluciones de la población hasta encontrar una con fitness 0 (todos los números distintos en todas las filas, columnas y zonas).

En cada generación del algoritmo, la población es ordenada de menor a mayor fitness, de manera de tener fácil acceso a la mejor solución y al mejor fitness obtenido. Además, esto hace posible que la selección de padres y el reemplazo por los nuevos individuos se vuelva más eficiente, como se verá a continuación.

Leyendo a otros autores, comprobamos que la mayoría está de acuerdo en que una función de fitness simple, como la que utilizamos nosotros, permite obtener soluciones tan eficazmente como otras funciones más complejas, como pueden ser, funciones que agregan peso al valor del fitness según el valor de la sumatoria/productoria de cada fila/columna.

2.2.4 Selección de Padres

Para la selección de padres se implementó un **Tournament Selection**, de tamaño 3. Se eligió este mecanismo ya que es más simple e igualmente eficiente que otros mecanismos con los que experimentamos, como Rank Selection.

El pseudocódigo para elegir cada padre es el siguiente:

```
int SudokuSolver::tournamentSelection() {
   int tournament_size = 3;
   int chosen = population_size;
   for (int i = 0; i < tournament_size; ++i) {
      int participant = rand() % population_size;
      if (participant < chosen) {
         chosen = participant;
      }
   }
   return chosen;
}
```

Notar que como la población se encuentra ordenada de menor a mayor fitness al seleccionar los padres, basta con elegir el de menor índice obtenido entre los participantes del torneo. Si aumentáramos el tamaño del torneo, daríamos menos chances a los individuos de menor fitness de ser elegidos como padres, pero decidimos evitar esto para mantener una población lo más diversa posible y evitar caer en mínimos locales.

2.2.5 Crossover

Una vez seleccionados los dos padres, utilizamos **crossover uniforme**, donde el nuevo individuo generado hereda algunas zonas (subcuadrados) de uno de los padres, y otras zonas de otro de los padres. Esto lo realizamos tirando una moneda para cada zona, de manera que sea lo más uniforme posible. Notar que utilizando este crossover mantenemos el invariante en todas las zonas, ya que al heredar zonas completas siempre van a seguir conteniendo números distintos del 1 al 9.

Durante el diseño del algoritmo experimentamos con crossovers con filas y con columnas, pero probablemente debido al método de inicialización por zonas, éstos métodos de crossover resultaron menos eficientes.

2.2.6 Mutaciones

Para pensar las mutaciones tomamos ideas de varios autores y las combinamos en una sola. Decidimos implementar **swap-mutation**, es decir swapeo de dos posiciones en el tablero que se encuentren en una misma zona.

Esta mutación se realiza con una probabilidad de mutación fijada experimentalmente en 0.6. Al swapear números siempre dentro de un mismo sub-cuadrado mantenemos el invariante inicial (reduciendo el espacio de búsqueda).

A esta primera versión le agregamos algunas características más. Al inicializar el tablero, dadas las posiciones fijas que vienen en el input, definimos los números "habilitados" por cada posición. Estos números habilitados corresponden a aquellos que no se encuentren inicialmente en la misma fila, columna o zona de cada posición. Notamos que colocar un valor numérico que rompe el invariante con una de las

posiciones fijas (que no se pueden modificar en el algoritmo) no ayuda a acercarse a la solución y hace que el algoritmo tarde más iteraciones en converger.

Por lo tanto, una vez definidas las dos posiciones distintas que vamos a swapear en cada zona, chequeamos si los nuevos valores que van a tomar estas posiciones se encuentran dentro de los números habilitados para cada una. En caso contrario, el swap no se realiza. Notar que esto produce que en realidad la probabilidad de swapeo en cada zona sea menor a la probabilidad fijada en 0.6.

Por último, la última modificación que le hicimos al algoritmo fue la de requerir que una de las dos posiciones que vamos a swapear se encuentre "mal coloreada". Esto quiere decir que dicha posición tiene el mismo color que alguno de sus vecinos (en su fila, columna o zona), y por lo tanto, es necesario que tome un nuevo valor. Con la experimentación realizada notamos un incremento en la velocidad de convergencia al agregar este último requerimiento, dando mejores resultados en el caso promedio.

El algoritmo para la mutación queda:

```
void SudokuChromosome::mutate() {
     if (getRandomNumber() <= solver->getMutationProba()) {
     std::vector<int> indexes = solver->getFreePositions();
     bool bad_colored = false;
     int index1;
     while(not bad_colored and indexes.size() > 0) {
        index1 = getSampleFromVector(indexes, true);
        if (count_by_row[solver->getRowNumberByPos(index1)][values[index1]-1] != 1 or
           count_by_col[solver->getColNumberByPos(index1)][values[index1]-1] != 1) {
           bad_colored = true;
        }
     }

     if (bad_colored) {
        int zone_number = solver->getZoneNumberByPos(index1);
        std::vector<int> zone_indexes = solver->getZoneIndexesByNumber(zone_number);
        int index2 = getSampleFromVector(zone_indexes, true);
        while (index1 == index2 or solver->getFixedPositions().count(index2) != 0) {
           index2 = getSampleFromVector(zone_indexes, true);
        }
        if (solver->getAvailableNumbersByPos(index1).count(this->values[index2]) != 0 and
           solver->getAvailableNumbersByPos(index2).count(this->values[index1]) != 0) {
           int tmp = this->values[index1];
           this->values[index1] = this->values[index2];
           this->values[index2] = tmp;
           updateFitness();
        }
     }
  }
}
```

El algortimo primero genera un número random para ver si corresponde o no realizar la mutación, según la probabilidad de mutación seteada en 0.6. Si el valor cae en el umbral procede con la mutación de la siguiente manera. Primero, se ocupa de encontrar una posición no-fija (FreePositions) que esté mal coloreada, usando los ya precomputados arrays de cantidad de apariciones por fila y columna de cada valor, para saber si hay algún vecino con su mismo color. Una vez encontrada esta posición se procede a calcular otra posición (distinta) dentro de su zona para hacer el swapeo, y por último se chequea si el swapeo es válido (chequeando los valores habilitados por posición). En caso de encontrarlo, realiza el swap y actualiza el fitness de la solución.

Otras mutaciones que también testeamos (y descartamos en el camino) fueron:

- Seleccionar posiciones random y recolorear usando colores habilitados.
- Seleccionar sólo las posiciones mal coloreadas y recolorear usando colores habilitadas.

2.2.7 Elitismo

En cada generación, la población se renueva a excepción de un cierto grupo de individuos que se preservan a la siguiente iteración. La cantidad de miembros de esta elite fue seleccionada experimentalmente, y llegamos a la conclusión que tomando un índice de 0.005 de elitismo (es decir, 5 individuos de

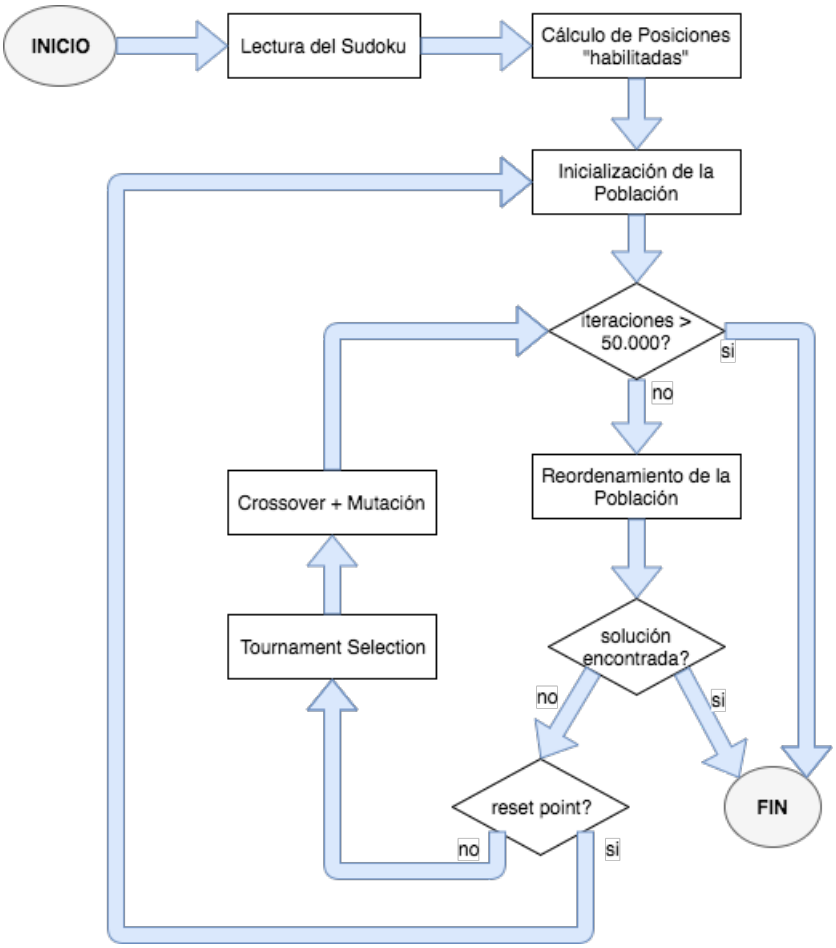
los 100) retorna los mejores resultados. Es interesante remarcar que este es el mismo índice de elitismo que usan en [1] aunque con otro tamaño de población total.

2.2.8 Mejoras

A medida que avanzábamos con la implementación nos topamos con algunos problemas que fuimos resolviendo añadiendo algunas mejoras al algoritmo. El principal problema que encontramos fue el de la rápida convergencia a mínimos locales. El problema del Sudoku, investigando en otras fuentes, tiene un **alto grado de mínimo local**, es decir, hay demasiadas soluciones factibles cercanas al fitness ideal, pero que en realidad se encuentran muy lejanas a la verdadera solución única del tablero. Esto hace que el algoritmo se estanque en muchas oportunidades por lo que es importante asegurar una convergencia no demasiado rápida, manteniendo la población lo más diversa posible. Algunas mejoras que implementamos fueron:

- **Resets** Cuando el algoritmo llega a un numero de generaciones, fijado en 300, sin mejorar el fitness de su mejor solución, la población es completamente re-inicializada, y el algoritmo comienza nuevamente a intentar buscar una solución. Varios autores recurren a esta técnica y por eso llaman a sus algoritmos "Algoritmos Genéticos Recuperables".
- **Población repetida** Otro inconveniente que nos llamó la atención fue el de que una vez que el algoritmo se acerca a converger, se comienzan a repetir los hijos generados, haciendo que sea más probable que el algoritmo se estanque. Para esto, una vez generado un nuevo individuo chequeamos que no sea repetido en la población, y en caso afirmativo, lo descartamos, generando uno nuevo.
- **Envejecimiento de la población** Para evitar caer en mínimos locales, si un individuo permanece como mejor solución durante mas de una generación, lo "envejecemos", es decir, aumentamos el valor de su fitness en 1 por cada generación que se mantenga como mejor solución.

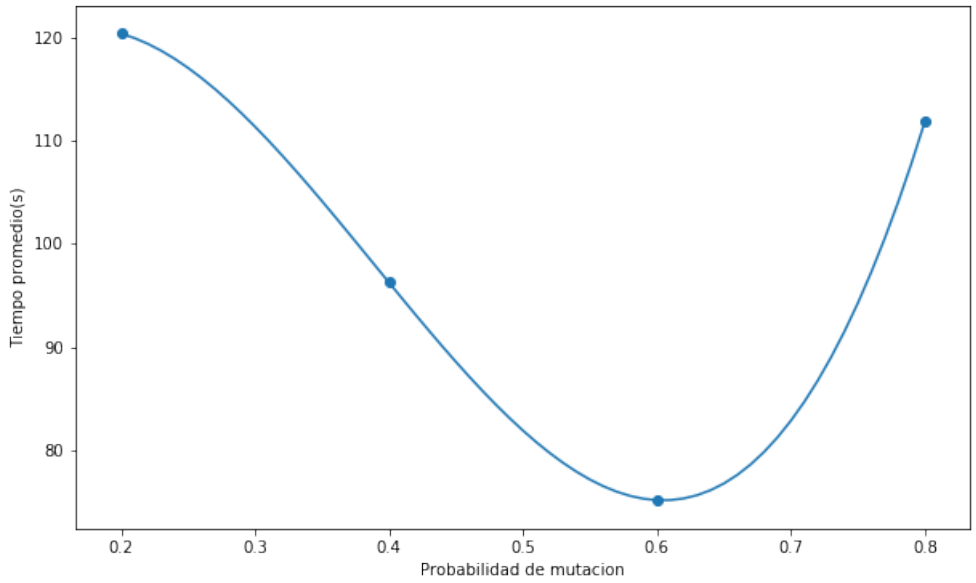
2.2.9 Diagrama General



2.2.10 Optimización de los parámetros

Los parámetros definidos para esta implementación fueron optimizados usando como medida los tiempos obtenidos al resolver 30 iteraciones de un Sudoku de dificultad media, fijando el resto de los parámetros. De esta manera, probamos con poblaciones entre 20 y 200 individuos, índices de elitismo de 0, 0.05, 0.5, probabilidades de mutación entre 0.2 y 0.8, distintos métodos de mutación/crossover ya mencionados y selección de padres (rank selection, tournament selection).

Un ejemplo de optimización de parámetros que realizamos fue el de la probabilidad de mutación. Para eso, fijamos el resto de los parámetros; en este caso, tamaño de población en 20, *restart point* en 2000 iteraciones, crossover por zonas y tournament selection de tamaño 3, y calculamos los promedios de tiempos obtenidos al resolver 30 iteraciones del sudoku *Challenging*. Graficamos los resultados que obtuvimos para este caso, variando la probabilidad de mutación entre 0.2 y 0.8. Tomamos algunos valores discretos en ese rango y graficamos una interpolación entre los puntos.



Tiempos promedio al optimizar probabilidad de mutación

En el gráfico se ve claramente que se alcanza el mínimo de tiempo de ejecución tomando probabilidad = 0.6. De esta manera, fijamos este valor y procedimos a optimizar el resto de los parámetros, obteniendo los valores ya mencionados para cada caso.

2.2.11 Consideraciones

La implementación fue realizada utilizando el lenguaje **C++ 11**, y para las métricas y gráficos de resultados utilizamos Python3.

3 Resultados

3.1 Comparación con otros algoritmos genéticos

En esta sección se comparan los resultados obtenidos por nuestro algoritmo con los publicados en los siguientes papers, con otras implementaciones de algoritmos genéticos: [1](2006), [2](2012) y [3](2014). En todos ellos se utiliza el mismo set de Sudokus con distintos niveles de dificultad, que se encuentran disponibles para descargar en la página [4].

En total se trata de 9 sudokus recolectados por Mantere y Koljonen (autores del paper [1]), donde los primeros 5 de ellos fueron sacados del diario "Helsingin Sgnomat" (2006) y fueron diferenciados de 1 a 5 estrellas, mientras que los otros 4 son del diario "Aamulethi"(2006) diferenciados entre dificultades: *Easy*, *Challenging*, *Difficult* y *Super Difficult*. Utilizamos este set para poder realizar una comparación lo más justa posible, ya que los 3 autores con los que vamos a comparar publicaron sus resultados usando este mismo set de tableros.

Cada uno de estos sudokus se resolvió 100 veces (al igual que en los otros papers) para calcular los resultados, usando una notebook con procesador Intel Core i7 quad core de 2.8 GHz y 16gb de RAM, con un timeout de 180 segundos para cada ejecución.

3.1.1 Rendimiento

Primero, comparamos la cantidad de sudokus resueltos para cada dificultad entre las 100 corridas, contra los resultados obtenidos por los demás algoritmos:

Comparación de efectividad entre los algoritmos genéticos de distintos autores para 100 corridas									
Dificultad	1⋆	2⋆	3⋆	4⋆	5⋆	Easy	Challenging	Difficult	SuperDifficult
Nuestro Algoritmo	100	98	98	66	77	100	**97**	**45**	**41**
[1](2006)	100	69	46	26	23	100	30	4	6
[2](2012)	100	100	100	100	100	100	94	16	9
[3](2014)	100	100	100	**97**	**79**	100	51	17	20

Como se puede ver en la tabla, nuestro algoritmo resuelve aproximadamente todos los sudokus de dificultad baja (hasta 3⋆), al igual que la mayoría de los algoritmos con los que lo comparamos. Con respecto a los de dificultad 4⋆ y 5⋆, la eficiencia de nuestra implementación está por debajo de los otros algoritmos. Sin embargo, obtuvimos mejores resultados, en cantidad de sudokus resueltos, para las dificultades más altas (*Challenging, Difficult y Super Difficult*), resolviendo aproximadamente la mitad de las instancias probadas, muy por encima de los demás. Creemos que esto se debe a los cambios que aplicamos para acelerar la convergencia del algoritmo, provocando que haya más probabilidad de converger a una solución aunque también haya más probabilidad de caer en mínimos locales. Este nivelamiento fue clave a la hora de definir los detalles de nuestro algoritmo, ya que intentamos acelerar el algoritmo sin hacerlo demasiado propenso a caer en mínimos locales (que es muy fácil para el problema del Sudoku).

3.1.2 Cantidad de Generaciones

Luego, realizamos una comparación por cantidad de generaciones del algoritmo (cada vez que se renueva la población). La comparación, en este caso, se realizó solamente con el algoritmo desarrollado en [2], ya que utilizan una población de tamaño similar a la nuestra (90 individuos).

Comparación por cantidad de generaciones hasta alcanzar la solución en 100 corridas												
D	**Resueltos**		**Mín**		**Máx**		**Prom**		**Mediana**		**Desv Std**	
	[2]	AG	[2]	AG	[2]	AG	[2]	AG	[2]	AG	[2]	AG
1⋆	100	100	39	15	1100	1196	159.24	139.13	98	40	159.95	186.23
2⋆	100	98	99	33	7459	4077	1864.94	1085.41	1441	639	1728.25	1040.92
3⋆	100	98	152	24	21595	4359	4338.30	839.91	2755	649	3906.40	782.00
4⋆	100	66	198	23	42382	4558	10716.73	2045.21	6966	1908	11285.80	1336.00
5⋆	100	77	117	25	48336	4454	13569.76	1614.77	9393	1294	12832.50	1259.04
E	100	100	27	12	305	201	70.34	41.61	52	25	55.70	38.95
C	94	97	357	27	41769	4601	25932.87	1028.78	27786	781	19153.54	987.37
D	16	45	3699	35	45676	5066	40466.13	2130.46	33760	1467	11940.66	1583.77
SD	9	41	21424	51	49832	5278	42354.86	2335.34	36318	2192	11198.42	1644.17

En la tabla anterior se puede observar que nuestro algoritmo (referenciado como AG en la tabla), tal como esperábamos, converge mucho más velozmente que el algoritmo de [2], ya que la cantidad de generaciones necesarias para llegar a la solución es en promedio menor, para todos los casos (y la diferencia se hace mayor cuanto mayor es la dificultad). Efectivamente, la propuesta de estos autores fue la de utilizar un espacio de búsqueda mucho más amplio (sin mantener el invariante por zona) con el fin de generar poblaciones más variadas, provocando una convergencia más lenta. Sin embargo, nuestro algoritmo permitió resolver una mayor cantidad de sudokus difíciles.

Por otro lado, esto no nos alcanza para poder decir cuál de los dos algoritmos resuelve los problemas en menos tiempo, ya que el tiempo de ejecución de una generación depende del funcionamiento particular del algoritmo y de los parámetros seleccionados. Lamentablemente, ninguno de los autores ya mencionados muestra en sus papers los tiempos de corrida de sus algoritmos (nuestros tiempos se muestran promediados por dificultad en la siguiente sección).

3.2 Comparación por dificultad

3.2.1 Tiempos Obtenidos

Los tiempos obtenidos (promedio y máximo, en segundos) para cada nivel de dificultad se pueden observar en el siguiente gráfico:

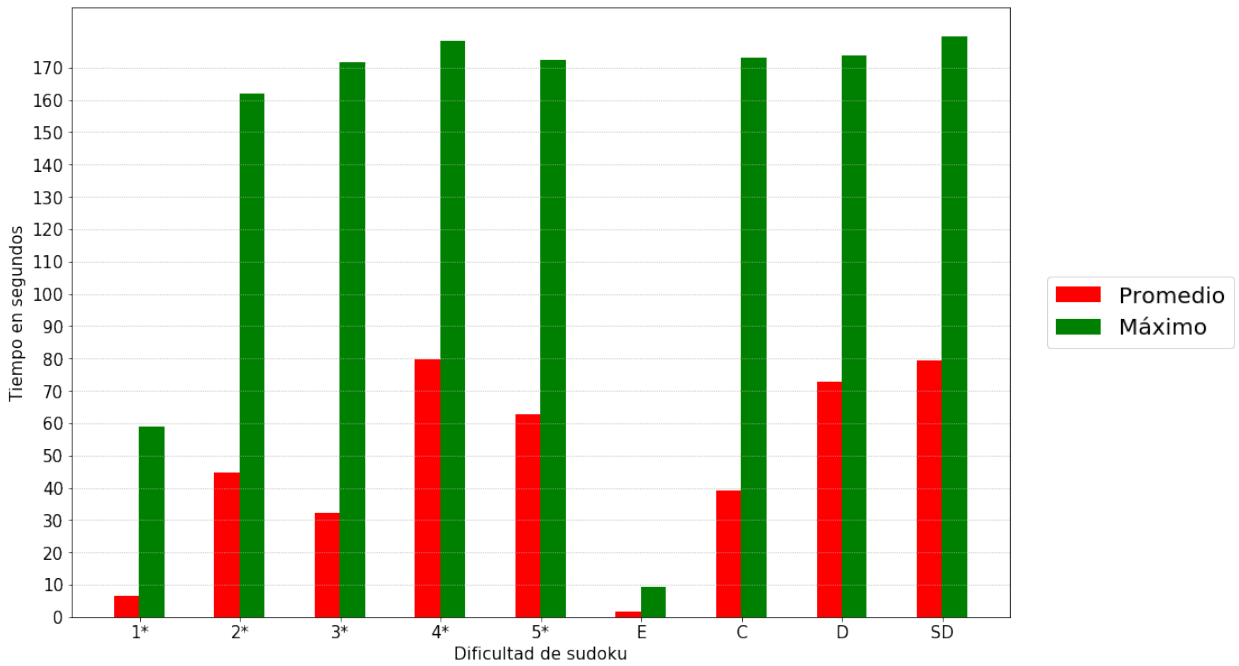


Gráfico 1: Tiempos obtenidos para cada dificultad en 100 corridas

Los tiempos mínimos no se muestran en el gráfico por claridad, ya que en todos los casos son tiempos muy pequeños (del orden de los milisegundos), y se deben a ejecuciones excepcionales del algoritmo donde aleatoriamente se llega a la solución en las primeras iteraciones.

Es interesante notar que al aumentar el nivel de dificultad de *Easy* a *Super Difficult*, el promedio aumenta casi linealmente en función de la dificultad. Sin embargo, en la distinción por estrellas, no se corresponde el orden de los promedios obtenidos con el orden de dificultad. Por ejemplo, los sudokus de 3$\star$ se resolvieron en menor tiempo promedio que los de 2$\star$.

Otro punto interesante del gráfico es el salto que pegan los máximos obtenidos. Al aumentar un poco la dificultad de los sudokus, el máximo despega hasta alcanzar casi el valor fijado como timeout (180 segundos).

Realizando las ejecuciones sin timeout, algunas ejecuciones de los sudokus más difíciles tuvieron que ser paradas debido a que se estancaban, mientras que los de dificultad media sí lograban resolverse en su totalidad. Se eligió dicho timeout para poder comparar la efectividad con los otros algoritmos en un rango razonable de tiempo.

3.2.2 Cantidad de Generaciones

En esta sección realizamos una comparación por dificultad según cantidad de generaciones necesarias para resolver los distintos sudokus. En el siguiente gráfico se puede observar cómo varía la cantidad de generaciones en cada caso tomando el promedio como eje x:

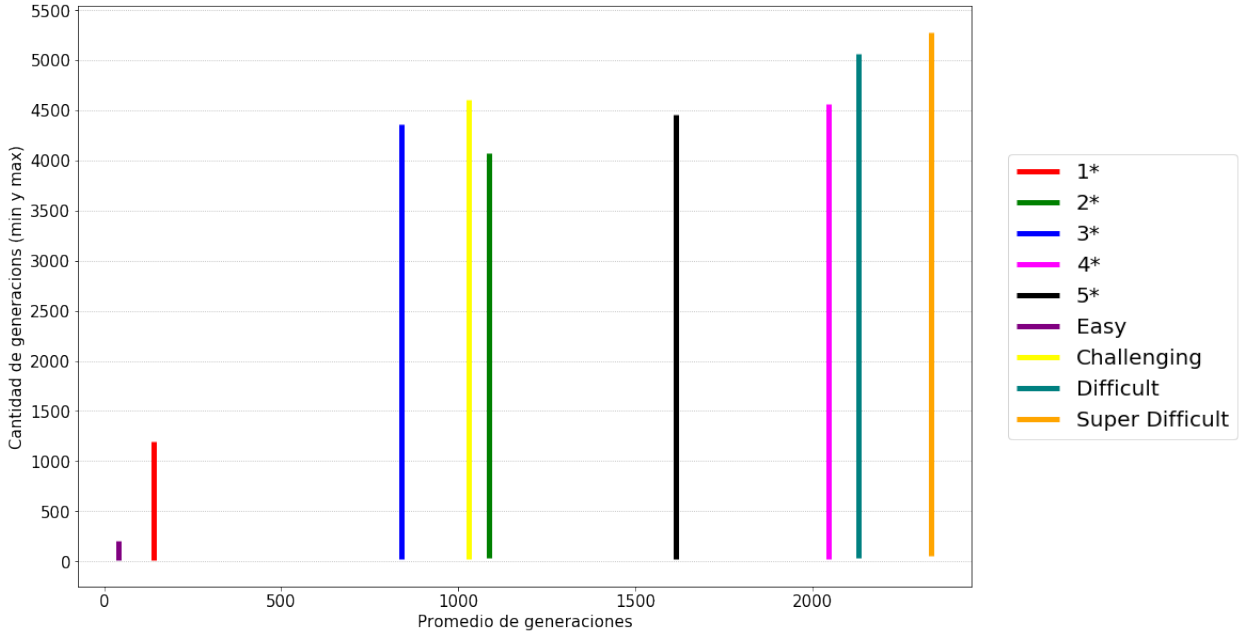


Gráfico 2: Generaciones necesarias para cada dificultad en 100 corridas

De este gráfico se puede determinar fácilmente un ranking global de dificultad entre los 9 distintos sudokus que usamos para testear. Los sudokus de tipo *Super Difficult*, se ubican como los más difíciles de resolver, con el rango de iteraciones y promedio más altos, mientras que los sudokus de calificación *Easy* son los de menor rango y promedio. Otra conclusión que se puede sacar es que los sudokus de nivel *Challenging* se ubicaron entre el nivel 2⋆ y 3⋆.

Mirando estos resultados, podríamos separar nuestro set de sudokus en 3 subconjuntos separados de dificultad: **Alto** (4⋆, 5⋆, *Difficult* y *Super Difficult*), **Medio** (2⋆, 3⋆, *Challenging*) y **Bajo** (1⋆, *Easy*). De esta manera, dado un sudoku nuevo podríamos usar nuestro algoritmo como "calificador" e incluirlo en alguno de estos 3 niveles.

3.3 Comparación con otras heurísticas

Por último, incluimos una comparación con los tiempos obtenidos por un algoritmo basado en Tabú Search de la literatura, [5], también para resolver sudokus. En este caso, el set de sudokus es distinto, aunque también se puede conseguir de la misma página que en el caso anterior [4], y la cantidad de pruebas por sudoku fue de 30. Veamos una comparación con los resultados obtenidos por el algoritmo Tabú según el nivel de dificultad de cada sudoku:

Resultados de nuestro algoritmo genético					
Dificultad	**Intentos**	**Solucionados**	**Mínimo(s)**	**Máximo(s)**	**Promedio(s)**
Easy	30	30	0.806	25.526	5.990
Medium	30	29	1.351	350.195	98.794
Hard	30	30	4.064	239.082	73.371

Resultados del Tabu Search de [5]					
Dificultad	**Intentos**	**Solucionados**	**Mínimo(s)**	**Máximo(s)**	**Promedio(s)**
Easy	30	30	1.026	1.049	1.032
Medium	30	30	5.151	245.346	69.807
Hard	49	30	3.875	309.799	112.667

Para estas pruebas, se aumentó el timeout a 360 segundos, ya que los valores máximos del tabú rondan el orden de los 300 segundos.

Comparando los resultados entre uno y otro, se puede ver que ambos son efectivos para obtener una solución del sudoku para los 3 niveles testeados, sin embargo, hay algunas diferencias en los tiempos de ejecución. Ambos algoritmos tienen similares tiempos mínimos, debido a que corresponden a casos en los que aleatoriamente se consigue la solución en las primeras iteraciones.

Para el caso *Easy*, nuestro algoritmo tiene un tiempo promedio mayor, debido a que en algunos casos probablemente necesite restartear la población para salir de un mínimo local.

En los sudokus de dificultad *Medium*, el algoritmo de Tabú obtiene mejores resultados, dando un promedio menor. Mientras que en los algoritmos *Hard*, nuestra implementación obtuvo mejores resultados. Este resultado se relaciona con lo mencionado anteriormente, donde llegamos a la conclusión de que nuestra implementación tiene una mejor performance para sudokus difíciles en comparación con otras implementaciones, aunque a veces no llegue a resolver los problemas más simples en tiempo óptimo. Esto se podría mejorar incluyendo otra heurística mas apropiada para construir las soluciones iniciales, utilizando mas información del tablero, o mezclando con algoritmos exactos en parte de la ejecución.

4 Posible implementación usando Simulated Annealing

La técnica de *Simulated Annealing* es un algoritmo de búsqueda meta-heurística para problemas de optimización, basado en el proceso de recocido del acero y cerámicas; una técnica que consiste en calentar y luego enfriar lentamente el material para variar sus propiedades físicas. El calor causa que los átomos aumenten su energía y que puedan así desplazarse de sus posiciones iniciales (un mínimo local de energía); mientras que el enfriamiento lento les da mayores probabilidades de recristalizar en configuraciones con menor energía que la inicial (mínimo global).

En cada iteración, el método de recocido simulado evalúa algunos vecinos del estado actual s y probabilísticamente decide entre efectuar una transición a un nuevo estado s' o quedarse en el estado s. En el ejemplo de recocido de metales descrito arriba, el estado s se podría definir en función de la posición de todos los átomos del material en el momento actual; el desplazamiento de un átomo se consideraría como un estado vecino del primero en este ejemplo. Típicamente la comparación entre estados vecinos se repite hasta que se encuentre un estado óptimo que minimice la energía del sistema o hasta que se cumpla cierto tiempo computacional u otras condiciones.

Para adaptar esta heurística al problema del Sudoku, podemos tomar como **función de energía** a nuestra función de fitness definida para el algoritmo genético, y al **operador de vecindad** como un swap entre dos posiciones dentro de una de las zonas del tablero, similar a nuestro operador de mutación.

El algoritmo se inicializa con una solución candidata (creada de manera random o usando, por ejemplo, nuestro algoritmo de inicialización explicado anteriormente), y luego explora el espacio de vecindad aplicando iterativamente el operador de vecindad. Dado un candidato s, un vecino s' puede ser aceptado según las siguientes condiciones:

- si la energía del vecino es menor que la del candidato actual, se acepta con probabilidad 1
- si la energía del vecino es mayor o igual, se acepta con probabilidad:

$$exp(-\delta/t)$$

  donde δ corresponde a la diferencia entre la energía del vecino y la energía del candidato actual, y t es una variable de control comúnmente denominada **temperatura**.

En general, la manera en que se elige t es de vital importancia para el éxito de un algoritmo de SA. Se cree que la temperatura inicial debería permitir aceptar la mayoría de los movimientos, y luego, durante la ejecución, reducirla lentamente (**enfriamiento**). Eventualmente, se llegaría a converger a una solución óptima.

Un valor de temperatura inicial alto puede provocar que se pierda tiempo recorriendo aleatoriamente el espacio de búsqueda, mientras que una temperatura inicial baja puede llegar a convertir al algoritmo en un algoritmo demasiado goloso, y susceptible a estancamientos en mínimos locales. Una propuesta para la implementación podría ser tomar como temperatura inicial un valor tal que permita realizar aproximadamente el 80% de los posibles movimientos (a ser estimada experimentalmente), y luego durante la ejecución, reducir la temperatura usando una técnica de enfriamiento geométrica, donde la temperatura t_i es modificada a una temperatura t_{i+1} siguiendo la fórmula:

$$t_{i+1} = \alpha * t_i \tag{1}$$

donde α es una variable de control denominada **grado de enfriamiento**, con un valor entre 0 y 1. Obviamente, un valor muy alto de α haría que a temperatura se reduzca muy lentamente, mientras que un valor bajo permite un enfriamiento mas rápido. Para una primera implementación se podría tomar un α de 0.9, aunque por experiencia, luego debería pasar por un proceso de optimización de parámetros para intentar mejorar los resultados de dicha implementación.

Por último, una mejora posible para este algoritmo de SA sería introducir la posibilidad de un **recalentamiento**, similar a los restarts de algoritmos genéticos, donde si dada una cantidad determinada de iteraciones no se encontró una solución óptima, resetear la temperatura y la solución candidata por una solución inicial.

5 Planteo del problema con Programación Lineal Entera

Un problema de programación lineal entera es un problema de programación lineal con la restricción adicional de que algunas de las variables deben tomar valores enteros. Si esta variable solo puede tomar valores entre 0 y 1 diremos entonces que se trata de una variable binaria. Una gran cantidad de problemas de combinatoria y optimización pueden ser planteados bajo esta técnica de programación, en particular el caso del sudoku.

La idea del planteo es transformar un tablero de 9x9 en un cubo de 9x9x9 con valores binarios 0 o 1. Es equivalente a pensarlo como si fueran 9 tableros apilados uno sobre otro, donde cada capa corresponde a un entero entre 1 y 9. El tablero superior, tiene un 1 en las posiciones que la solución al sudoku tengan un 1. El segundo tablero tiene un 1 donde la solución tiene un 2. En general, el n-ésimo tablero tiene un 1 en las posiciones donde la solución tiene valor N.

Esta formulación es adecuada para aplicar programación lineal entera **binaria**. El problema, entonces, consiste en encontrar una solución que satisfaga las siguientes reglas:

Suponiendo que la solución está representada con una estructura x de tamaño 9x9x9,

- Cada posición de la solución final al sudoku tiene un sólo valor, por lo que entre todas las capas debe haber un sólo 1 para cada posición (i, j). Es decir:

$$\sum_{k=1}^{9} x(i,j,k) = 1 \quad \forall\,(i,j),\ 1 \leq i \leq 9,\ 1 \leq j \leq 9 \tag{2}$$

- En cada fila de la solución final debe haber un sólo valor por cada dígito del 1 al 9. Es decir, en cada fila i tiene que haber exactamente un 1 en cada uno de los k niveles:

$$\sum_{j=1}^{9} x(i,j,k) = 1 \quad \forall\,(i,k),\ 1 \leq i \leq 9,\ 1 \leq k \leq 9 \tag{3}$$

- Análogamente al anterior, en cada columna de la solución final debe haber un sólo valor para cada digito del 1 al 9. Es decir, en cada columna j tiene que haber exactamente un 1 en cada uno de los k niveles:

$$\sum_{i=1}^{9} x(i,j,k) = 1 \quad \forall\,(j,k),\ 1 \leq j \leq 9,\ 1 \leq k \leq 9 \tag{4}$$

- Además, en cada una de las zonas del sudoku se debe cumplir la misma restricción que antes. Es decir, en cada uno de los k niveles, hay exactamente un 1 en cada zona:

$$\sum_{i=1}^{3}\sum_{j=1}^{3} x(i+U, j+V, k) = 1 \quad \forall\, U, V \in \{0,3,6\},\ 1 \leq k \leq 9 \tag{5}$$

- Por último, por cada posición inicial (i, j) que ya viene dada con valor m, agrego la siguiente constante:

$$x(i,j,m) = 1 \tag{6}$$

La solución va a ser el tablero que se desprende del x que satisfaga todas estas condiciones, y para encontrarlo se deberá aplicar alguno de los métodos de resolución de programación lineal.

6 Conclusiones

En este trabajo pudimos experimentar cómo es el proceso de diseño de algoritmos al trabajar con metaheurísticas, pensando ideas propias y combinando ideas de otros autores y heurísticas para intentar encontrar una solución óptima en tiempo razonable a un problema NP-Completo como lo es el Sudoku.

Fue necesario investigar bastante sobre el problema, y sobre las distintas variantes de algoritmos genéticos posibles, para intentar esquivar las dificultades que nos fuimos encontrando, mejorando el algoritmo iterativamente probando nuevas propuestas que fueron surgiendo hasta llegar a una implementación razonable.

Además, nos sirvió de mucha ayuda leer a otros autores que trabajaron sobre metaheurísticas para problemas similares, y revisar los ejemplos de clase, para entender cómo encarar los problemas, las formas de optimizar los algoritmos, y por último comparar los resultados obtenidos.

A pesar de que existan algoritmos exactos de tipo CSP que resuelven Sudokus más eficientemente, pudimos comprobar que es posible resolver muchas instancias de Sudokus usando metaheurísticas y es un problema muy interesante para seguir investigando.

Como trabajo futuro, queda pendiente y nos interesaría mezclar conceptos de otras metaheurísticas e integrarlos a nuestro algoritmo, como por ejemplo, utilizar la propuesta usando SA que se explicó en el informe para intentar mejorar la mejor solución sub-óptima encontrada antes de resetear nuestro algoritmo, o añadir una lista tabú para no repetir movimientos y evitar caer en mínimos locales.

7 Bibliografía

[1] Mantere, T. and Koljonen, J. (2006). Solving and Rating Sudoku Puzzles with Genetic Algorithms. New Development of Artificial Intelligence and the Semantic Web, Proceeding of the 12th Finnish Artificial Intelligence Conferences, (STeP 2006), Octubre 26-27, pp: 86-92.
https://pdfs.semanticscholar.org/1331/c019866526aad677279d1791a23805742ce7.pdf

[2] K. N. Das, S. Bhatia, S. Puri, and K. Deep, "A retrievable GA for solving Sudoku puzzles," Citeseer, 2012.
http://sumitbhatia.net/papers/sudokuTechReport.pdf

[3] Seyed Mehran Kazemi, Bahare Fatemi. A Retrievable Genetic Algorithm for Efficient Solving of Sudoku Puzzles. World Academy of Science, Engineering and Technology International Journal of Computer, Electrical, Automation, Control and Information Engineering Vol:8, No:5, 2014.
https://pdfs.semanticscholar.org/2dfb/c695e2f2bf1e6a66da7a7242a26be3fcffa9.pdf

[4] Mantere, T., Koljonen, J. (2008). Sudoku research page.
http:// lipas.uwasa.fi/ timan/sudoku/

[5] Ricardo Soto, Broderick Crawford, Cristian Galleguillos, Eric Monfroy, Fernando Paredes. A hybrid AC3-tabu search algorithm for solving Sudoku puzzles (2013).
https://pdfs.semanticscholar.org/8614/d90f2989e13787da2361f700fd3889dd170a.pdf

[6] "NP complete – Sudoku" (PDF). Imai.is.su-tokyo.ac.jp.
http://www-imai.is.s.u-tokyo.ac.jp/ yato/data2/SIGAL87-2.pdf

[7] Semeniuk, I. (2005). Stuck on you. Publicado en New Scientist 24, pp: 45-47.

[8] Felgenhauer, B. and Jarvis, A. F. (2006). Mathematics of Sudoku I. Mathematical Spectrum 39 (2006), pp:15–22