Universidad EAFIT

Maestría en Ciencia de Datos y Analítica

Aprendizaje Automático

S2261-0136

Aprendizaje Evolutivo Solucionador Evolutivo de Sudoku

Autores:

Alejandro Barrientos-Osorio

Luis Miguel Caicedo-Jimenez

Omar Alejandro Henao-Zapata

26 de abril de 2022

I. INTRODUCCIÓN

Sudoku es un acertijo de reemplazo numérico, fue adaptado de un concepto matemático llamado *Latin Square*. Para solucionarlo se debe completar una matriz, en su caso más general, de una dimensión de 9x9 celdas con números del 1 al 9, sujeto a reglas:

- En las filas, no debe haber números repetidos.
- En las columnas, no debe haber números repetidos.
- En cada subcuadro 3x3, no debe haber números repetidos.

El sudoku inicialmente entrega unas pistas o valores dados para poder dar solución al problema, los valores dados aunque son pistas, también acotan el problema de optimización.

- Si se dan menos números de los mínimos necesarios para resolver el problema, podrían haber infinitas soluciones, pues no estaría lo suficientemente acotado.
- Con el número estrictamente necesario de números dados, sería un problema con única solución, pero de elevada dificultad.
- Si se dan más números de los necesarios, el problema está sobre especificado, lo cual volvería más fácil el problema, sin embargo, sigue teniendo una única solución.

Los sudokus son problemas considerados NP completos, para verificar que una solución es correcta, en una malla nxn, podríamos simplemente recorrer cada columna, fila y cuadro y verificar números duplicados. Si hay, se rechaza la solución. El verificador de soluciones corre en tiempo $O(n^2)$, confirmando que el problema es NP completo.

Los algoritmos tradicionales para solucionar problemas NP completos son considerados de fuerza bruta, donde se prueban todas las posibles combinaciones hasta que se llega a una posible solución, que tiene obvios problemas de computación, especialmente cuando se reducen los números dados al comienzo del problema, pues el número de combinaciones aumenta o cuando la dimensión del sudoku aumenta.

Una solución alternativa son los algoritmos genéticos (GA), que se derivan de los algoritmos evolutivos que están basados en los conceptos fundamentales de la evolución en Biología. Un GA clásico empieza con una población inicial de posibles soluciones. Estas soluciones son ordenadas por medio de una función de ajuste o *fitness function*, que determina que tan

buena es la solución. Las mejores soluciones o individuos son seleccionados y recombinados o apareados usando un proceso similar al cruce en la recombinación de ADN. Luego, una mutación es aplicada, mutando un porcentaje de la población, la mutación igual que en la naturaleza, para un cromosoma o individuo padre cambia una parte de su genoma o solución de manera aleatoria. Las nuevas soluciones o individuos, junto con los mejores candidatos de la generación anterior, componen la nueva población. Este proceso se aplica hasta que una solución al sistema haya sido encontrada o hasta que haya llegado al número máximo de generaciones. [1]

Los GA son buenos cuando se enfrentan a espacios de búsqueda gigantes y deben navegarlos, buscando por las combinaciones óptimas, soluciones que son difíciles de encontrar para un humano o por fuerza bruta. Un GA es una iteración de pasos de búsqueda, optimización y adaptación. Es una técnica del machine learning que se basa en los principios de la naturaleza y que ha sido probada util en la solución de problemas NP difíciles. [1]

En este trabajo, extrajimos algunas partes del algoritmo propuesto por [2] para generar un GA que sea capaz de resolver sudokus aplicando metodologías evolutivas, en el notebook entregado junto con este trabajo se puede ver que solo se utilizó el marco del algoritmo y su paso de mutación, pues la función de ajuste, recombinación, verificación de solución y lectura del problema fue desarrollo propio.

A. Objetivo general

- Estudio de algoritmos evolutivos para solución de problemas NP difíciles

B. Objetivos Específicos

- Desarrollo de una función de fitness adecuada para el problema de sudoku.
- Desarrollo del cross-over elitista.
- Optimización de hiper parámetros para encontrar los mejores parámetros del modelos en términos de generaciones requeridas para resolver el sudoku.

Dentro de las próximas secciones se hablará de la metodología de solución y desarrollo, un breve análisis de los resultados y finalmente se concluirá.

II. METODOLOGÍA

La metodología que se usó fue CRISP-DM, conocida para afrontar problemas de desarrollo de analítica, sin embargo, el paso de implementación no fue llevado a cabo debido a que el propósito del trabajo es un estudio de los GA.

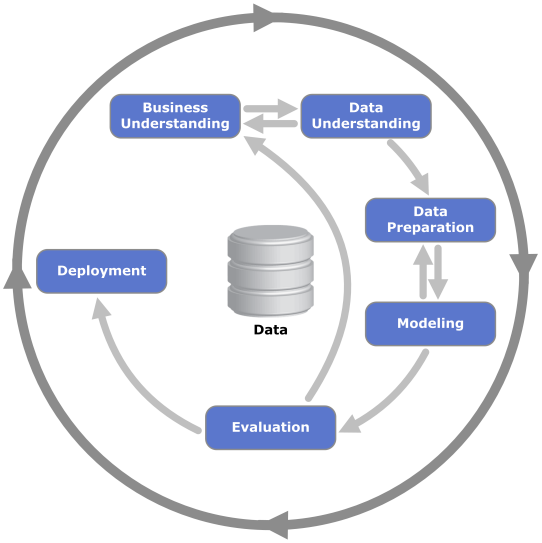


Figura 1: Metodología CRISP-DM

En la figura 1 se puede ver una representación gráfica de la metodología usada.

A. Entendimiento del Negocio

Los problemas de sudoku son problemas de solución compleja, tiene bastantes restricciones y su solución es única, además, debido al gran espacio de búsqueda, se corre una alta posibilidad de caer en mínimos locales. Debido a la naturaleza del problema, por las altas restricciones que tiene cada individuo, la generación de nuevos individuos facilitaría la solución del problema, esto es discutido más ampliamente en la sección de modelado.

B. Entendimiento de los datos

En el problema particular que intentamos resolver, tenemos sudokus de 9x9 que son matrices compuestas de submatrices con dimensión 3x3, que llamaremos de ahora en adelante cuadros. Las condiciones que se deben cumplir ya fueron mencionadas

en la introducción. Dentro del desarrollo utilizamos una base datos obtenida en Kaggle (`https://www.kaggle.com/datasets/bryanpark/sudoku`) la base de datos se llama "1 millón de sudokus"se extrajeron algunos de estos para probar el algoritmo.

C. Preparación de los datos

La preparación de los datos consistió de un cambio de forma para verificar los subcuadros que hacen parte de las restricción y para favorecer el crossover, que se hace entre subcuadros.

8	6	4	3	7	1	2	5	9
3	2	5	8	4	9	7	6	1
9	7	1	2	6	5	8	4	3
4	3	6	1	9	2	5	8	7
1	9	8	6	5	7	4	3	2
2	5	7	4	8	3	9	1	6
6	8	9	7	3	4	1	2	5
7	1	3	5	2	8	6	9	4
5	4	2	9	1	6	3	7	8

8	6	4	3	2	5	9	7	1
3	7	1	8	4	9	2	6	5
2	5	9	7	6	1	8	4	3
4	3	6	1	9	8	2	5	7
1	9	2	6	5	7	4	8	3
5	8	7	4	3	2	9	1	6
6	8	9	7	1	3	5	4	2
7	3	4	5	2	8	9	1	6
1	2	5	6	9	4	3	7	8

Figura 2: Transformación al sudoku

Ambas matrices vistas en la figura 2 son usadas para encontrar la solución. Está transformación permite hacer verificaciones fila a fila y columna a columna de manera más eficiente.

D. Modelado

El modelo parte de una malla llena por completo de 9x9, siendo 0 los valores que no se conocen, este es el formato en como están almacenados el millón de sudokus.

1. Generar población inicial

El primer paso consiste en crear una población de posibles soluciones, para esto se crea el parámetro llamado **population size** que indica de que tamaño será la población inicial, una población inicial más grande podría otorgar más soluciones posibles, pero incrementa también el costo computacional.

0	0	4	3	0	0	2	0	9
0	0	5	0	0	9	0	0	1
0	7	0	0	6	0	0	4	3
0	0	6	0	0	2	0	8	7
1	9	0	0	0	7	4	0	0
0	5	0	0	8	3	0	0	0
6	0	0	0	0	0	1	0	5
0	0	3	5	0	8	6	9	0
0	4	2	9	1	0	3	0	0

0	0	4	0	0	5	0	7	0
3	0	0	0	0	9	0	6	0
2	0	9	0	0	1	0	4	3
0	0	6	1	9	0	0	5	0
0	0	2	0	0	7	0	8	3
0	8	7	4	0	0	0	0	0
6	0	0	0	0	3	0	4	2
0	0	0	5	0	8	9	1	0
1	0	5	6	9	0	3	0	0

Figura 3: Malla inicial y transformada de entrada. 9x9 - 35 números dados

La generación de población inicial es aleatoria, sin embargo, parte de los valores ya dados por el problema, pues estos son números que sabemos son correctos, el algoritmo que explica este paso se ve así:

- Determinar **population size**.
- Crear tantas mallas de 9x9 como indique el parámetro, con números enteros aleatorios del 1 al 9.
- Reemplazar los valores ya dados por el problema en las posiciones correctas.
- Añadir la malla a la población.

El tamaño de la población inicial no fue un parámetro optimizado debido a que no afectaba significativamente el resultado, excepto cuando era muy pequeño pues no había suficiente espacio de búsqueda para encontrar la solución, se estableció como un valor fijo de 10000.

2. Fitness

Este es el segundo paso, en donde se le asigna un número llamado fitness a cada individuo de la población. Este será un número que indica que tan cerca de la solución final está, en nuestro caso es inverso, es decir, mientras mas cercano a cero sea el número, más cerca está de ser solución.

Nuestra función de fitness se calcula como la suma entre tres restas:

- 9 - el número de números únicos en una fila.

- 9 - el número de números únicos en una columna.
- 9 - el número de números únicos en una cuadro.

Si toda la malla está compuesta del mismo número, el número de números únicos es 1. Para la comparación de filas tendríamos un resultado de 72, 72 de nuevo para las columnas y 72 para los cuadros. Con esto podemos verificar que el número máximo que fitness podría ser es 216 en caso de que se trate de una malla del mismo número, en caso de ser todos números únicos la función arrojara 0, indicando que se alcanzó la solución.

3. Selección

Este es el tercer paso, en donde entra el segundo parámetro del algoritmo **selection rate** que indica cual será el porcentaje de la población inicial que se seleccionará, el algoritmo de selección se ve así.

- Calcular el fitness de cada individuo de la población.
- Ordenar de menor a mayor fitness.
- Determinar **selection rate**
- Escoger solo los primeros $selectionrate * populationsize$ individuos

Para este parámetro usamos un grid-search entre $\{0{,}1, 0{,}3, 0{,}5\}$ y el criterio de selección fue el número de generaciones requerido para llegar a la solución en un problema de test.

4. Generación aleatoria

Este es el cuarto paso y es opcional, dentro de los GA convencionales se menciona solo cross-over y mutación, sin embargo, por la naturaleza del problema y debido a que contiene tantos mínimos locales, este mecanismo ayudo a acelerar el proceso de aprendizaje.

En el paso anterior, se seleccionó la proporción de individuos con mejor fitness o más cercanos a la solución, dada por el parámetro selection rate. Sin embargo, esto redujo el número total de la población, eliminando la proporción $(1 - selection\_rate)$ de individuos menos cercanos a la solución. Para poder llegar de nuevo al número original de la población,

hay que generar nuevos individuos. Aquí entra el tercer parámetro estudiado **random proportion** que determina cuantos de estos individuos faltantes serán generados de manera aleatoria, igual que en paso 1. El resto de los individuos faltantes, se generan en el siguiente paso.

El parámetro **random proportion** fue optimizado por medio de un grid search en el que se le otorgaron posible valores entre $\{0{,}3, 0{,}6\}$

5. Crossover

Este es el quinto paso, acá se generan los individuos restantes para completar de nuevo el tamaño de la población inicial, crossover se refería al apareamiento entre dos soluciones para crear una nueva, el crossover se aplica solo con los mejores padres, aplicando una selección y apareamiento elitista.

El algoritmo que explica el crossover elitista, se ve así:

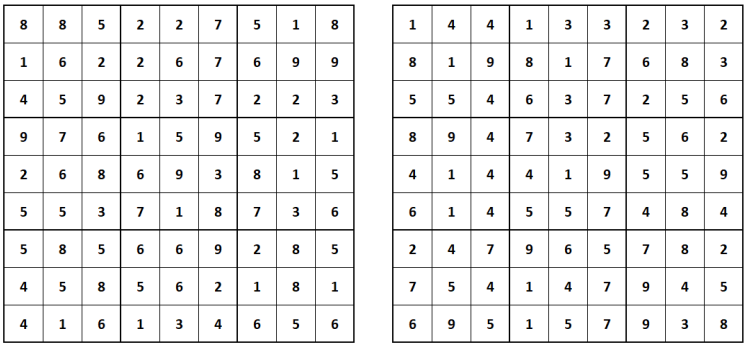
- Seleccionar dos individuos aleatorias de la selección del paso 3.
- Seleccionar dos índices aleatorios.
- Hijo 1 tendrá todas las filas del padre 1, menos una, llenada con una fila del padre 2.
- Hijo 2 tendrá todas las filas del padre 2, menos una, llenada con una fila del padre 1.
- Se selecciona el hijo con menor fitness y se adiciona a la población.

En la figura 4, se ve de manera gráfica el procedimiento.

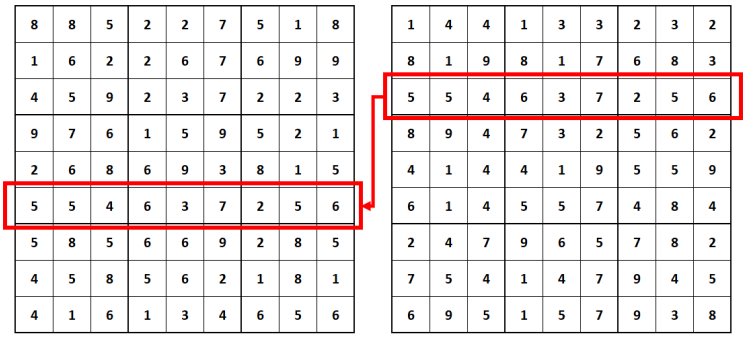
6. Mutación

El último paso a realizar es la mutación de los individuos, aquí entra el último parámetro usado, **mutation rate** que indica la proporción de individuos que mutaran. El algoritmo de mutación se ve así:

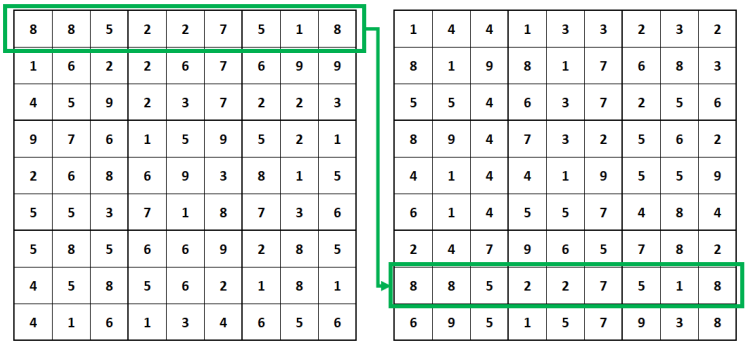
- Escoge un cuadro aleatorio.
- Revisa cuantos de los números en el cuadro aleatorio pueden ser modificados, es decir, si el problema no los ha especificado.



((a)) Padre 1(izq) y Padre 2(Der) que realizaran un crossover.



((b)) Hijo 1(izq) y Padre 2(Der).



((c)) Padre 1(izq) y Hijo 2(Der).

Figura 4: Procedimiento de apareamiento aleatorio.

- Si el cuadro tiene 2 o más números no especificados continua.
- Escoge una posición no especificada aleatoriamente y guarda el número como número 1 y la posición como índice 1.
- Escoge una posición no especificada aleatoriamente y guarda el número como número

2 y la posición como índice 2.

- Intercambia los números, asignando a índice 1 el número 2 y asignando a índice 2, el número 1.
- Adiciona este individuo a la población.

De nuevo, como es un problema NP difícil se beneficia de tener nuevas y diferentes soluciones para poder abarcar el mayor posible espacio de búsqueda, la optimización del parámetro de mutación se acotó en un grid search entre $\{0{,}1, 0{,}5, 0{,}9\}$.

Todos los pasos anteriormente mencionados se repiten un número fijo de veces, definido como **maximum generation count** o máximo número de generaciones, es fijo debido a que casi todos los problemas se resuelven antes de las 200 generaciones y si antes no se resuelve, es porque entro en un mínimo local del que no saldrá, por lo menos en los resultados estudiados.

7. Reset

Como se mencionó durante toda esta sección, el problema del mínimo local es recurrente y la solución propuesta para este tipo de restricciones en la optimización utilizamos un algoritmo de reseto, se ve así:

- Se verifica si lleva más de 60 generaciones.
- Se verifica si lleva más de 40 generaciones, con la mejor generación teniendo el mismo fitness.
- Se recalcula toda la población de nuevo como una población totalmente aleatoria.
- Se continua con el GA.

E. Evaluación

Como se menciono en la sección de modelación, se realizó un grid search para la optimización de hiperparámetros del algoritmo, para cada escenario presentado se solucionó el mismo sudoku 5 veces y se obtuvo la mediana de las generaciones requeridas para alcanzar la solución como variable de respuesta.

Escenario	selection rate	mutation rate	random proportion	Generaciones
1	**0.1**	**0.1**	**0.3**	**10**
2	0.1	0.1	0.6	30
3	0.1	0.5	0.3	11
4	0.1	0.5	0.6	13
5	0.1	0.9	0.3	12
6	0.1	0.9	0.6	13
7	0.3	0.1	0.3	18
8	0.3	0.1	0.6	23
9	0.3	0.5	0.3	16
10	0.3	0.5	0.6	22
11	0.3	0.9	0.3	20
12	0.3	0.9	0.6	28
13	0.5	0.1	0.3	24
14	0.5	0.1	0.6	40
15	0.5	0.5	0.3	28
16	0.5	0.5	0.6	42
17	0.5	0.9	0.3	34
18	0.5	0.9	0.6	50

Cuadro I: Grid search para la optimización de parámetros, la columna generaciones es el promedio de generaciones necesario para resolver el sudoku.

III. ANÁLISIS

Los GA son herramientas que según demostramos, pueden resolver problemas NP difíciles como el sudoku, sin embargo, la naturaleza del sudoku lo vuelve complejo para este método de solución debido al gran número de mínimos locales, al espacio tan amplio de búsqueda, etc. Algo que se puede notar en el cuadro I en general, una random proportion más alta, induce un número más alto de generaciones para resolver el sudoku. En partícular, un mutation rate más alto también lleva a tener que usar más generaciones para resolver el problema. Y además, todas las pruebas donde el selection rate es 1, es donde se concentra el número más

bajo de generaciones requeridas. Esto se puede explicar como que el problema se beneficia de una búsqueda más amplia de soluciones, entonces quedarse solo con el 10 % de los mejores candidatos de la población le permite explicar más soluciones con el crossover.

La configuración de hiperparámetros óptima resuelve en un cuarto de las generaciones el sudoku con respecto a las generaciones necesarias para la configuración más lenta. Sin embargo, incluso en la configuración óptima el GA propuesto es aún muy ineficiente comparado con el benchmark actual para solución de sudokus llamado algoritmo de backtracking.

En general, otros autores han encontrado que la solución por medio de GA a sudokus es generalmente ineficiente [3]. Sin embargo, una conclusión común es que en 9x9 backtracking es más rápido, en comparación de segundos en backtracking a minutos en GA, sin embargo, cuando la escala crece por encima de problemas 9x9 a nxn GA empieza a tener ventaja, debido al costo computacional elevado de backtracking.

IV. CONCLUSIONES

- Encontramos que los GA son algoritmos válidos para la solución de problemas NP difíciles.
- Verificamos que para la solución de problemas con espacios de búsqueda restringidos, es más favorable tener parámetros de exploración altos como random proportion y mutation rate.
- Observamos que para sudokus 9x9 GA no es el mejor modelo, pero podría ser escalado fácilmente a problemas nxn donde si es competidor de los algortimos actuales.

V. REFERENCIAS

[1] G. Panchal and D. Panchal, "Solving np hard problems using genetic algorithm," *International Journal of Computer Science and Information Technologies*, vol. 6, pp. 1824–1827, 2015. [Online]. Available: www.ijcsit.com

[2] J. M. Weiss, "Genetic algorithms and sudoku," 2009.

[3] T. Mantere and J. Koljonen, "Solving and rating sudoku puzzles with genetic algorithms," 2006.