Comparación de algoritmos de búsqueda para resolver Sudoku

Esteban Cacavelos
Universidad Católica de Asunción
Capitán Lezcano 2335 c/ Pascual Toledo
+595 981 220 429
estebancacavelos@gmail.com

Juan Santa Cruz
Universidad Católica de Asunción
De la conquista 1350 c/ Mayor Martinez
+595 981 538 078
jotasanta@gmail.com

ABSTRACT
El trabajo se basa en probar la resolución de tableros de sudoku con un algoritmo genético y un algoritmo de búsqueda no informada llamado backtracking. El objetivo es poder sacar conclusiones acerca de la naturaleza de estos algoritmos respecto al problema del sudoku. A través de pruebas realizadas con diferentes tableros, de diferentes dificultades para humanos se quiere llegar a conclusiones que muestren la eficiencia de cada algoritmo.

Categories and Subject Descriptors
I.2 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search – *backtrackings, control theory, dynamic programming, graph and tree search strategies, heuristic methods, plan execution, formation, and generation, scheduling.*

General Terms
Algorithms, Measurement, Experimentation.

Keywords
GA, Sudoku, optimización, algoritmos genéticos, poda, sub-bloque, pistas.

1. INTRODUCCION
De acuerdo a Wikipedia Sudoku es un pasatiempo estadounidense que se popularizó en Japón en 1986, y se dio a conocer en el ámbito internacional en 2005 [1]. Es altamente popular y adictivo debido a que es un juego con reglas muy simples y fáciles de entender, y a su vez representa un desafío lógico muy interesante.

1.1 Propiedades generales
Sudoku puzzle se compone de un cuadrado 9 × 9 que a su vez se dividen en nueve sub cuadrados 3 × 3. La solución del Sudoku es tal que cada fila, columna y sub cuadrado contiene cada número entero del 1 al 9 una y sólo una vez. Al principio hay algunos números estáticos (previamente dados) en el rompecabezas que se dan de acuerdo con la calificación de dificultad. Normalmente el Sudoku se clasifica en niveles de dificultad, según la relevancia y posición de los números iniciales y no de la cantidad de estos. Efectivamente, la cantidad de números dados apenas afecta la dificultad del Sudoku e incluso puede no afectar en absoluto ya que un Sudoku con un mínimo de números iniciales puede ser muy fácil de resolver por la lógica humana y uno con más números de la media puede ser extremadamente complicado de resolver. Además, se dice que un Sudoku está bien planteado si la solución es única.

1.2 Análisis del Problema como un problema de Búsqueda.
El problema de resolver un tablero de Sudoku puede verse como un problema de búsqueda: el encontrar una o todas las secuencias de dígitos del 1 al 9 que completen la grilla cumpliendo con las restricciones mencionadas anteriormente. Para disminuir la cantidad de posibilidades a examinar, es necesario de alguna forma construir solo las secuencias que sean validas. Esto es lo que hacemos en los dos algoritmos utilizados para buscar las soluciones en este trabajo, que son el Bactracking (o vuelta atrás) y un Algoritmo Genético.

Aunque al método de Backtracking que es básicamente ir probando todas las opciones validas (y si llega a un estado donde ya no puede ubicar ningún numero vuelve atrás para seguir probando las demás opciones) hasta encontrar una solución se le puede agregar métodos para aumentar su eficiencia, veremos que igualmente en muchos aspectos tiene ventajas sobre el Algoritmo Genético implementado.

1.3 Análisis del espacio de estados del problema.
Sin importar la cantidad de Sudokus completos (aquellos que cumplen las restricciones) que existen, la dificultad radica en la explosión combinatoria que presenta un sudoku con determinada cantidad de números iniciales (pistas), es decir, entre menos pistas tenga el Sudoku, mayor es el espacio de soluciones (factibles e infactibles) que existen como podemos apreciar en la tabla 1 [2].

Pistas	Cuadros en blanco	Espacio de soluciones
75	6	5,3144E+05
70	11	3,1300E+10
65	16	1,8530E+15
60	21	1,0942E+20
55	26	6,4611E+39
50	31	3,8152E+44
45	36	2,2528E+49
40	41	1,3303E+54
35	46	7,8552E+58
30	51	4,6384E+63
25	56	2,7389E+68
20	61	1,6173E+73
15	66	9,5500E+77
10	71	5,6392E+82
5	76	3,3299E+87
0	81	1,9663E+92

Tabla 1. Espacio de soluciones según la cantidad de pistas para el Sudoku de orden 9.

Por lo tanto, dadas una cierta cantidad de pistas (Xp), el espacio de soluciones (Es) es igual a:

$$Es = 9^{(81-Xp)}$$

Como podemos observar en la ecuación, el problema del Sudoku presenta el fenómeno de explosión combinatorial, lo cual significa un incremento en el espacio de soluciones y también en el esfuerzo computacional para buscar dichas soluciones [2].

En cuanto a la existencia de solución única, no ha sido posible demostrar cuál es la cantidad mínima de pistas que debe tener un Sudoku para garantizar solución única, aunque es fácil determinar que con 78, 79 y 80 pistas se garantiza una solución única. Respecto a 77, en [3] se muestra dos posibles soluciones para un Sudoku.

1.4 Análisis matemático del problema.

A muchos nos resulta natural buscar una forma matemática de encarar el problema, en [4] vemos un estudio exhaustivo de cómo encarar la resolución de sudokus mediante más de 350 ecuaciones. En síntesis el estudio expuesto es este:

Se necesitan variables binarias de la forma x_{ijk} donde 1 significa que el símbolo *k* (de 1 a 9) va en la celda *(i,j)* de la solución y 0 significa que no está. Esto genera un total de *729 variables* (9^3). Veamos las ecuaciones:

$$\sum_{k=1}^{9} x_{ijk} = 1 \quad \text{for all } (i,j) \in \{1,\dots,9\}^2, \tag{1}$$

$$\sum_{j=1}^{9} x_{ijk} = 1 \quad \text{for all } (i,k) \in \{1,\dots,9\}^2, \tag{2}$$

$$\sum_{i=1}^{9} x_{ijk} = 1 \quad \text{for all } (j,k) \in \{1,\dots,9\}^2, \tag{3}$$

$$\sum_{i=1}^{3}\sum_{j=1}^{3} x_{(m-1)*3+i,(n-1)*3+j,k} = 1 \quad \text{for } m,n \in \{1,2,3\},\ k \in \{1,\dots,9\}, \tag{4}$$

$$x_{ijk} = 1 \quad \text{for all } (i,j,k) \in P, \tag{5}$$

$$x_{ijk} \in \{0,1\} \quad \text{for all } (i,j,k) \in \{1,\dots,9\}^3? \tag{6}$$

(1) Para que cada celda *(i,j)* tenga un símbolo y éste sea único. Se necesitan 81 ecuaciones como estas (una por cada celda). **(2)** Para que en la fila *i* cada símbolo esté una vez y que en todas las columnas sea distinto. Se necesitan 81 ecuaciones como estas (una por cada columna y cada símbolo posible). **(3)** Para que en la columna *j* cada símbolo esté una vez y que en todas las filas sea distinto. Se necesitan 81 ecuaciones como estas (una por cada fila y cada símbolo posible). **(4)** Para que en cada región cada símbolo esté una y solo una vez. Se necesitan 81 ecuaciones como estas (una por cada región y cada símbolo posible). **(5)** Representa el enunciado, es decir, el problema a resolver. Por ejemplo, $x_{115}=1$ significa que en la celda (1,1) hay un 5. Dependiendo cuantos casilleros vengan asignados es la cantidad de igualdades como estas que se necesitan; típicamente, ~30. **(6)** Restringe las variables al conjunto binario.

Aunque hay propuestas con menos cantidad de ecuaciones, siempre serán demasiadas como para intentar encarar la solución mediante el método simplex visto en Investigación Operativa [5], existen ya métodos algorítmicos que ayudan a resolver problemas este tipo de problemas [6].

2. DESCRIPCIÓN Y RESUMEN DEL ESTADO DEL ARTE.

2.1 Algoritmos utilizados comúnmente.

Backtracking: En su forma básica, la idea de backtracking se asemeja a un recorrido en profundidad dentro de un grafo dirigido. El grafo en cuestión suele ser un árbol, o por lo menos no contiene ciclos. Sea cual sea su estructura, existe sólo implícitamente. El objetivo del recorrido es encontrar soluciones para algún problema. Esto se consigue construyendo soluciones parciales a medida que progresa el recorrido; estas soluciones parciales limitan las regiones en las que se puede encontrar una solución completa.

El recorrido tiene éxito si, procediendo de esta forma, se puede definir por completo una solución. En este caso el algoritmo puede bien detenerse (si lo único que se necesita es una solución del problema) o bien seguir buscando soluciones alternativas (si deseamos examinarlas todas).

Por otra parte, el recorrido no tiene éxito si en alguna etapa la solución parcial construida hasta el momento no se puede completar. En tal caso, el recorrido vuelve atrás exactamente igual que en un recorrido en profundidad, eliminando sobre la marcha los elementos que se hubieran añadido en cada fase. Cuando vuelve a un nodo que tiene uno o más vecinos sin explorar, prosigue el recorrido de una solución. [7]

Ramificación y Poda: Esta técnica se suele interpretar como un árbol de soluciones donde cada rama nos lleva a una posible solución posterior a la actual. La característica fundamental de esta técnica con respecto a otras(y a lo que debe su nombre) es que el algoritmo se encarga de detectar en qué ramificación las soluciones dadas ya no están siendo óptimas, para podar esa rama del árbol y no continuar malgastando recursos y procesos en casos que se alejan de la solución óptima.
Podemos ver una solución mediante este algoritmo aquí [8].

2.2 Descripción de la técnica de I.A utilizada.

Algoritmos genéticos: Es una variante de la búsqueda de haz estocástica en la que los estados sucesores se generan combinando dos estados padres, más que modificar un solo estado. La analogía a la selección natural es la misma que con la búsqueda de haz estocástica, excepto que se trata con reproducción sexual más que con la reproducción asexual [9]. No es un algoritmo muy clásico para resolver este tipo de problemas, pero se encuentra en muchos trabajos principalmente como fuente de comparación y aprendizaje del funcionamiento de los algoritmos evolutivos.

3. DESCRIPCION DE LOS ALGORITMOS IMPLEMENTADOS

3.1 Backtracking

El algoritmo de backtracking implementado es una técnica de programación de hacer búsqueda sistemática a través de todas las configuraciones posibles dentro de un espacio de búsqueda.

Para lograr esto, nuestro algoritmo construye posibles soluciones candidatas de manera sistemática. En general, dado una solución candidata X hacemos lo siguiente:

1. Verificamos si X es solución. Si lo es termina el algoritmo y devuelve el tablero resuelto.

2. Si X no es solución, seguimos construyendo todas las posibles extensiones de X, e invocamos recursivamente al algoritmo. Las extensiones que vamos armando son todas validas, de esta manera reducimos el espacio de estados.

A veces los algoritmos de este tipo se usan para encontrar una sola solución como en nuestro caso, pero también se podría modificar para revisar todas las soluciones posibles al problema.

Los tableros se modelan con una matriz de números enteros y las soluciones candidatas se extienden agregando elementos validos a la matriz.

3.2 Algoritmo Genético

Para la implementación definimos las piezas fundamentales del algoritmo de la siguiente manera:

3.2.1 El cromosoma

El cromosoma se representa con una cadena de 81 elementos en donde cada carácter puede contener los valores del 1 al 9. La cadena se divide conceptualmente en 9 sub-bloques de 9 elementos que representan las regiones del tablero de izquierda a derecha, de arriba a abajo.

3.2.2 Población

La población inicial se genera aleatoriamente teniendo en cuenta, en primer lugar que los números ubicados en los sub-bloques sean todos diferentes, y segundo, que los números fijos dados en el tablero inicial sean respetados, de esta manera se asegura la condición de que en cada región existan todos números diferentes del 1 al 9. Como la función es aleatoria, esta puede ser llamada muchas veces y generará habitualmente poblaciones distintas.

3.2.3 Función Idoneidad

Ya que la condición anterior asegura la restricción de región, *la función idoneidad* [10] se establece penalizando el no cumplimiento de las restricciones de fila y columna. Para tal objetivo se tienen en cuenta que la suma de cada fila y cada columna del tablero debe ser igual a 45 en una configuración ideal, así también, la multiplicación debe ser igual a 9! y la el conjunto de números en cada fila y columna debe ser igual a {1,2,3,4,5,6,7,8,9}.

$$g_{i1}(x) = \left| 45 - \sum_{j=1}^{9} x_{i,j} \right| \qquad g_{j1}(x) = \left| 45 - \sum_{i=1}^{9} x_{i,j} \right| \qquad (1)$$

$$g_{i2}(x) = \left| 9! - \prod_{j=1}^{9} x_{i,j} \right| \qquad g_{j2}(x) = \left| 9! - \prod_{i=1}^{9} x_{i,j} \right| \qquad (2)$$

Las dos primeras ecuaciones marcan las dos condiciones de suma y multiplicación en filas y columnas.

$$A = \{1,2,3,4,5,6,7,8,9\}$$
$$g_{i3}(x) = |A - x_i| \quad , \quad g_{j3}(x) = |A - x_j| \qquad (3)$$

La tercera es una expresión derivada de la teoría de conjuntos, que requiere que cada fila xi y cada columna xj debe ser igual al conjunto A = {1,2,3,4,5,6,7,8,9}. A continuación se expone la función f(x) que toma como valor la suma de estas tres ecuaciones:

$$f(x) = 10*(\sum_i g_{i1}(x) + \sum_j g_{j1}(x)) + \sum_i \sqrt{g_{i2}(x)} + \sum_j \sqrt{g_{j2}(x)} + 50*(\sum_i g_{i3}(x) + \sum_j g_{j3}(x)) \qquad (4)$$

Con esta expresión podemos obtener la *función idoneidad* que consiste en restar un numero grande que supere el máximo valor que puede devolver f(x), con esto se obtienen mayores valores para estados más aptos.

3.2.4 Selección

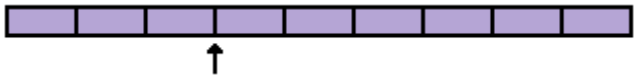
La selección se realiza prácticamente de manera aleatoria pero se utiliza un método por el cual un cromosoma con valor mas alto de función idoneidad tenga un poco más de posibilidades de ser elegido, esto se hace mediante el API utilizado para el desarrollo del proyecto [11].

3.2.5 Cruzamiento

El cruce lo realizamos de dos maneras distintas:

1. Con una probabilidad de 80%, se realiza un cruce de tipo simple que consiste en elegir aleatoriamente un punto de cruce (que tiene que ser en el comienzo de un sub-bloque) y entonces se cruzan los dos cromosomas seleccionados como vemos en el siguiente ejemplo:

Individuo 1:



Suponiendo que el punto de cruce es el cuarto sub-bloque.

Individuo 2:

Y por último el nuevo individuo 3 cruzado:

2. Con probabilidad de 20%, se realiza un cruce de tipo compuesto que es básicamente lo mismo que el cruce simple pero con dos puntos de cruce de la siguiente manera:

Individuo 1:

Suponiendo que los puntos de cruce están en el cuarto y en el séptimo bloque.

Individuo 2:

Y el nuevo individuo 3 cruzado seria:

Las probabilidades de cruzar de una u otra forma (80% y 20%) fueron seleccionadas empíricamente, porque fuimos notando que el funcionamiento ideal tendía a estos números.

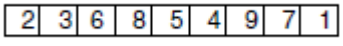
3.2.6 Mutación

La mutación se realiza teniendo en cuenta una probabilidad de mutación, tal como se entiende el proceso natural de mutación.

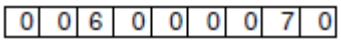
El funcionamiento es el siguiente, bajo la probabilidad de mutación que se ingreso al programa se llama a un procedimiento en donde con dicha probabilidad se muta o no cada una de las 9 regiones.

Las mutaciones en cada sub-bloque ocurren de la siguiente manera [10].

Suponiendo que este es un sub-bloque del tablero

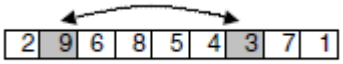


Y que las pistas ubicadas en el tablero inicial (que siempre quedan fijas) eran estas:

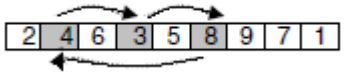


Tendremos básicamente dos tipos de mutaciones:

1. Una en la cual se intercambian dos números



2. Y otra en la cual se intercambian tres números



Cuando se van mutando cada una de las regiones, los tipos de mutación 1 ó 2 descritas ocurren con un 50% de probabilidad cada una.

3.2.7 Condición de parada

El algoritmo buscara un individuo solución hasta encontrarlo o hasta que se acabe el tiempo máximo de busca que se determina al iniciar el programa.

3.3 Optimizaciones para el Algoritmo Genético

El algoritmo buscara un individuo solución hasta encontrarlo o hasta que se acabe el tiempo máximo de busca predeterminado en el programa.

3.3.1 Con respecto a la probabilidad de mutación

Encaramos esta optimización por el método de "Mutación Adaptativa por Temperatura" que es un derivado del algoritmo Simulated Annealing [9]. La probabilidad de Mutación está definida previamente y es dependiente de la cantidad de generaciones que realizo el AG, teniendo un máximo y un mínimo.

Encaramos ambas formas de este tipo de mutación que son:

Mutación Adaptativa por Temperatura Descendente: La mutación disminuye por disminuir la Probabilidad de Mutación, el AG comienza teniendo una alta Probabilidad de Mutación y gradualmente va disminuyendo hasta un mínimo. Esta implementación busca explorar al máximo el universo de individuos al comenzar el algoritmo para encontrar lo más rápido posible los máximos o mínimos locales y asegurar que el AG utilice estos espacios de búsqueda como población, luego al disminuir la probabilidad de mutación se sigue explorando el conjunto de soluciones para asegurar que el AG encuentre la mejor solución.

Y la *Mutación Adaptativa por Temperatura Ascendente***.** La probabilidad de mutación aumenta a medida que avanza el AG es decir que comienza con la probabilidad de mutación más baja. De esta forma se exploran las áreas adyacentes a los primeros máximos o mínimos buscando estos localmente y luego al aumentar la probabilidad de mutación se explora en más detalle el universo completo de búsqueda para encontrar los máximos o mínimos globales de existir en el nuevo espacio en el que se buscan.

El efecto que queremos lograr con estos tipos de mutaciones sobre la población es este:

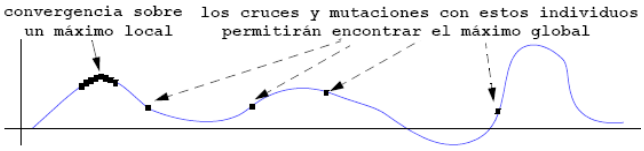


Figura 1. Efecto de la mutación sobre una población.

Luego de una serie de pruebas, concluimos que la mejor optimización para nuestro caso particular era la "Mutación Adaptativa por Temperatura Ascendente". Entonces hacemos que la probabilidad de mutación aumente hasta un máximo de 80%.

3.3.2 Regeneración de la población

Otro efecto interesante de "tipo" Simulated Annealing que nos pareció interesante implementar fue regenerar la población (manteniendo la población de elite) cada vez que en 100 iteraciones se repite el mismo mejor individuo, es decir el individuo con mayor valor de la función idoneidad.

Con esto lo que buscamos es salir de un máximo local y tratar de hacer tender al algoritmo a una convergencia con el máximo global, es decir, la solución.

3.3.3 Elitismo

Lo que hacemos es guardar los N (valor parametrizable en el programa, N << la cantidad de individuos de la población) mejores individuos de una generación para introducir en la siguiente, asegurándonos claramente de que estos también deben pasar por el proceso de mutación, cruce, etc.

Con esto ayudamos al algoritmo a que tenga más probabilidades de converger al óptimo y también logramos que el valor de la función idoneidad del mejor individuo de una generación no disminuya en la sucesión de generaciones.

3.3.4 Reducir el tamaño de la población a un mínimo N.

En un principio pensamos que el tamaño de la población no tenía por qué ser constante, entonces deducimos que una buena práctica para ayudar a la convergencia a la solución sería reducir el tamaño de la población a medida que transcurrían las generaciones.

Intentamos optimizar el algoritmo mediante esta técnica pero tuvo resultados negativos en nuestro caso particular entonces removimos de la implementación final. Lo que pudimos observar aquí fue que perdíamos diversidad en la población y que por esto empeoraba el rendimiento del algoritmo.

4. DESCRIPCION DEL EXPERIMENTO REALIZADO

4.1 Prueba sobre el Backtracking

Para hacer las pruebas para medir la capacidad de resolución del backtracking probamos con 15 tableros de los 100 que teníamos para resolver, los tableros, los códigos fuentes del proyecto y la forma de utilizar el programa se encuentra aquí [12].

Nivel	tablero nro	O(n) Temporal	Nivel	tablero nro	O(n) Temporal
Muy facil	1	77	Medio	52	923
	11	71		58	1787
	19	91		46	1358
Facil	25	85	Dificil	70	2125
	27	260		61	3434
	30	139		72	3558
			Muy dificil	100	2927
				88	6186
				82	10685

Figura 2.Prueba sobre el backtracking.

En la figura 2 podemos ver el numero de tablero de los 100 dados, y también la complejidad temporal que esta medida con la cantidad de veces que verifica si un nodo dado es un nodo solución.

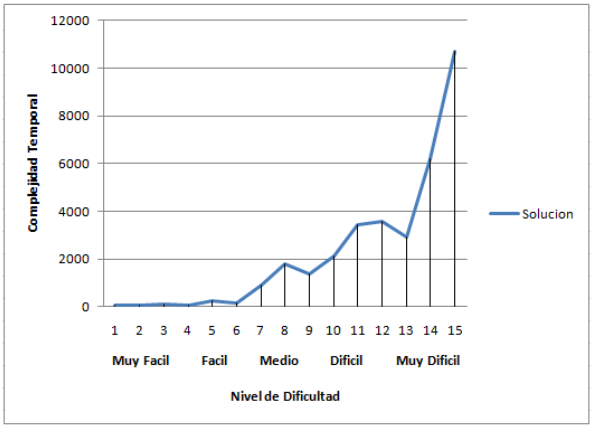


Figura 3. Complejidad Temporal vs Nivel de Dificultad

Los tableros están divididos de acuerdo a la dificultad que tienen cada uno de estos para ser resueltos por la lógica del pensamiento humano. En la figura 3 podemos observar que la complejidad temporal del algoritmo es creciente cuando la dificultad va en aumento.

4.2 Pruebas sobre el Algoritmo Genético

Para realizar las pruebas sobre este algoritmo utilizamos 10 tableros de los 100 dados. Siendo los siguientes 3 parámetros la configuración principal del programa:

-Población total

-Población de elite, un numero de N individuos a mantener.

-Índice de mutación inicial

Lo que hicimos fue intentar resolver 3 veces cada uno de los 10 tableros con 4 configuraciones distintas como se observa en la figura 4.

	Poblacion total	200	400	600	800
	Poblacion elite	50	100	150	200
	I.M.I	0,6	0,6	0,6	0,6
Nivel	tablero numero	O(n) Temporal	O(n) Temporal	O(n) Temporal	O(n) Temporal
Muy facil	1	232.427.991	332.376.403	335.632.218	452.199.527
	19	74.045.498	159.476.424	366.949.157	288.207.077
Facil	25	94.790.618	360.629.876	305.630.738	222.403.080
	30	XXX	342.431.638	XXX	XXX
Medio	46	XXX	XXX	XXX	XXX
	52	XXX	357.869.058	XXX	XXX
Dificil	61	XXX	XXX	506.194.765	XXX
	72	XXX	XXX	XXX	XXX
Muy dificil	82	XXX	XXX	XXX	XXX
	100	XXX	XXX	XXX	XXX

Figura 4. Pruebas sobre el AG

Como podemos ver, logramos que bajo al menos una de las configuraciones nos resuelva 6 de los 10 tableros, con límite de tiempo de ejecución de T=90 minutos.

La complejidad temporal del algoritmo definimos como el número de evaluaciones que necesita para resolver el problema, es decir la cantidad de veces que verifica si un nodo dado es un nodo solución. En nuestro caso, el algoritmo hace una evaluación cada vez que la función idoneidad calcula el valor de un cromosoma.

En la figura 4 I.M.I significa "Índice de Mutación Inicial" Los cuadros que contienen "XXX" denotan que se intento resolver 3 veces y no se consiguió la resolución en ninguno de los tres intentos.

A continuación se expone el gráfico de complejidad temporal vs. nivel de dificultad. Para la realización del gráfico se tuvo en cuenta el mejor resultado que se obtuvo en cada tablero. Por ejemplo, para el segundo tablero que se probo (numero 19), el mejor resultado que se obtuvo fue 74.045.498 que corresponde a la prueba con población inicial de 200 en la figura 4. Para los tableros en los que no se logro una solución el grafico llega a 0, como es el caso del quinto tablero probado, que corresponde al tablero 46 en la figura 4.

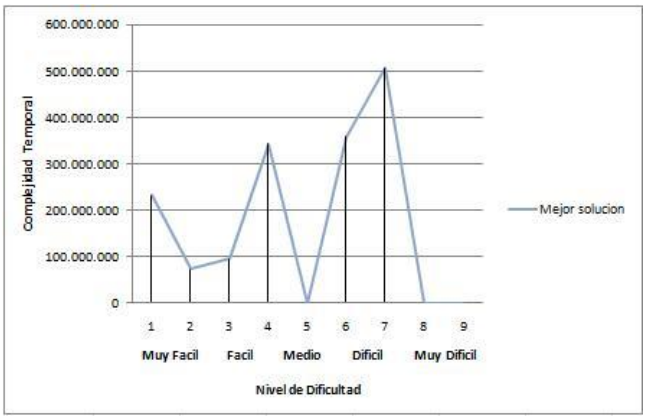


Figura 5. Complejidad Temporal vs Nivel de Dificultad

4.3 Comparación entre los algoritmos

Concluimos que el algoritmo de backtracking es más eficiente que el AG, ya que este, como máximo tardó 15 segundos en resolver un tablero. Al hablar de eficiencia, nos referimos a la complejidad temporal y también a la complejidad espacial, ya que este consume mucho menos recursos.

Los gráficos de complejidad temporal vs. Nivel de dificultad de los dos algoritmos marcan la diferencia en cuanto a la complejidad temporal. El backtracking llega como máximo a 10.000 evaluaciones, mientras que el algoritmo genético llega a aproximadamente 500.000.000 de evaluaciones en el peor de los casos.

En términos de complejidad espacial es extremadamente grande la diferencia. El backtracking se representa con una matriz y se evalúan ciertas restricciones. El algoritmo genético utiliza la misma cantidad de espacio para representar a cada individuo y estos se generan proporcionalmente a la cantidad de población que se desee, de modo que a mayor población, mayor complejidad espacial. Esta diferencia es notoria cuando se realizan las pruebas, aumentando el uso de los recursos de la computadora al ejecutar los algoritmos.

Es importante hacer la observación que en ambos casos el nivel de dificultad para el ser humano es el mismo que para los dos algoritmos. El algoritmo genético depende en gran medida (al menos para este problema, según observamos) del tipo de mutación que se haga, puesto que en este proceso es donde mayormente se van consiguiendo individuos mejores, de modo que, si la mutación no fuese buena, esta no tendría la capacidad de generar un individuo solución, como es el caso del Sudoku.

Claramente, el algoritmo genético no es apto para el problema del sudoku. El backtracking se podría considerar mucho más apto para resolver el sudoku respecto al algoritmo genético. Creemos que en problemas en los que el espacio de estados es muy grande, como es el caso de los tableros difíciles del sudoku, los algoritmos se podrían ver afectados en la eficiencia, lo cual se podría solucionar implementando otro tipo de técnicas para la solución.

5. CONCLUSIONES

De los distintos análisis realizados con el algoritmo genético se pueden deducir, para este problema y algoritmo particular, que:

-Tamaño de la población: Tener una población grande aumenta la posibilidad de encontrar excelentes individuos en cada generación, pero por contra requiere más tiempo de procesamiento, lo que reduce el número de generaciones que podemos obtener. Tener una población pequeña es más rápido, pero puede hacer a la población converger demasiado pronto a tableros no muy buenos y similares. El tamaño de la población tiene cierta incidencia en el resultado del algoritmo.

-Número de generaciones máximo: Calculamos que 90 minutos de ejecución del algoritmo era un buen tiempo como para sacar conclusiones de si solucionaba o si se quedaba estancado en un máximo local. Si hubiésemos dejado más tiempo tal vez conseguíamos resolver más tableros, aunque esto también puede ser falso, porque teóricamente podría también quedarse eternamente en un máximo local.

-Índice de mutación: Un índice demasiado alto dificulta el desarrollo de buenas soluciones, uno demasiado bajo puede producir que la población converja demasiado pronto en nodos de bajo nivel. Es interesante modificarlo durante el transcurso del algoritmo, comenzando con un índice bajo e incrementarlo a medida que avanza, aunque este hecho tampoco influye decisivamente en el rendimiento del algoritmo.

-Una mayor diversidad en la población ayuda a evitar el estancamiento en el progreso hacia una mejor calidad de soluciones.

-Lo que pudimos observar con mucha frecuencia es que los más aptos tienden a homogeneizarse en una solución que no suele ser la óptima. Lo que ocurría muchas veces era que llegaba a un valor altísimo en la función de idoneidad (como faltando un pasito para resolver), y luego se quedaba estancado ahí. Esto nos ocurrió en la gran mayoría de los tableros que no nos resolvió.

-La dificultad para resolver un sudoku mediante la lógica humana y los algoritmos genéticos son similares, ya que observamos que mientras crecía la dificultad de los tableros(para los humanos) también crecía la complejidad temporal del algoritmo.

-Con la práctica vimos que lo recomendado es utilizar un 25% de la población total como población de elite. Sin esta optimización de utilizar población de elite, la resolución de tableros se vuelve mucho más complicada.

-Hay ciertos puntos que se deben tener en cuenta al buscar resolver un problema mediante algoritmos genéticos, debemos observar primero si la función a optimizar tiene muchos máximos locales podrían requerirse demasiadas iteraciones del algoritmo para "asegurar"(porque en realidad no podemos asegurar) el máximo global, también debemos fijarnos si la función a optimizar contiene varios puntos muy cercanos en valor al óptimo, solamente podríamos "asegurar" que encontraremos uno de ellos (no necesariamente el óptimo).

Los códigos fuentes del proyecto están publicados en un servidor [12] como para poder recibir opiniones y/o críticas.

6. REFERENCES

[1] Wikipedia. Sudoku. Wikipedia, la enciclopedia libre. http://es.wikipedia.org/wiki/Sudoku

[2] Franco, John F., Gómez, O., Gallego, Ramon A., 2007 “Aplicación de técnicas de optimización combinatorial a la solución del sudoku”.

[3] Delahaye Jean-Paul. “The science behind Sudoku”. Scientific American. 2006. Pg. 80-87.

[4] Bello, Luciano.

http://www.lucianobello.com.ar/post/sudoku-algoritmos-geneticos-la-estrategia-backtracking-programacion-lineal-y-r/

[5] Hillier F., Lieberman G., Investigacion de Operaciones-7ma Ed. 2002

[6] Bernard Dantzig, George http://www.phpsimplex.com/teoria_metodo_simplex.htm

[7] Wikipedia. Backtracking. Wikipedia, la enciclopedia libre. http://es.wikipedia.org/wiki/Backtracking

[8] Wikipedia. Sudoku ramificación y poda. Wikipedia, la enciclopedia libre. http://es.wikipedia.org/wiki/Sudoku_ramificación_y_poda

[9] Russel, J., Stuart and Norvig Peter. 2004. Inteligencia Artificial, un enfoque moderno. http://aima.cs.berkeley.edu/

[10] Mantere, T. K.(2007) “Solving and Rating Sudoku Puzzles with Genetic Algorithms”

[11] API Java de AIMA http://code.google.com/p/aima-java/

[12] Santa Cruz, Juan I., Cacavelos, Esteban L., Bosch, Diego J., http://code.google.com/p/busqueda-sudoku/