Tarea 2 - Optimización estocástica

Erick Salvador Alvarez Valencia

CIMAT A.C.,
erick.alvarez@cimat.mx

Resumen En el presente reporte se hablará sobre el desarrollo y la implementación de una heurística constructiva y una búsqueda local para la resolución de un sudoku en diferentes niveles de dificultad. Se mostrarán varios resultados obtenidos al ejecutar dichos métodos en el cluster *El Insurgente* y se harán comparaciones con respecto a la heurística constructiva y una solución generada aleatoriamente.

Keywords: Sudoku, heurística constructiva, búsqueda local

1. Motivación

Para comenzar, un sudoku se juega en su versión más común con una matriz de 9x9 en donde se dan inicialmente un conjunto de números fijos en el rango (1-9) que formarán parte de su solución. Una solución en el sudoku es aquella donde se tiene la matriz llena de números que se encuentren en el rango anterior dicho y que cumpla con tres restricciones: Las filas, columnas y celdas de 3x3 no deberán contener dígitos repetidos. Por lo anterior vemos que este juego conlleva a cierto nivel de dificultad para resolverlo, y más en el caso de una computadora, por lo cual se desarrollaron los algoritmos de una heurística constructiva y una búsqueda local, los cuales se integrarán en un futuro a un algoritmo evolutivo que permita resolver rápidamente sudokus.
En las siguientes secciones se hablan más a detalle sobre estos dos algoritmos, pero antes de ello es necesario describir cómo se representarán las instancias de algún sudoku y el por qué de esto.

2. Representación de las instancias de un sudoku

Es importante elegir desde un inicio una buena forma para representar a este juego ya que esta nos puede facilitar o dificultad el trabahar con los algoritmos que se mostrarán más adelante así como algún algoritmo evolutivo. Para representar las instancias de un sudoku se optó por usar una estructura que almacene tres cosas: Los números que hay por defecto en un bloque, las posiciones en una matriz de dichos números y los números que se pueden poner en las celdas restantes en forma de permutación, al final se creaba un vector de 9 estructuras anterior descritas para tener una representación completa del sudoku. Lo más

destacable de esta representación es la última parte, ya que, se optó por guardar los números de esta forma debido a que trae dos ventajas en si, la primera es que al momento de que la función de fitness evalúe una instancia, los bloques de 3x3 están asegurados para estar siempre resueltos, por lo cual sólo deberiamos preocuparnos por resolver las filas y columnas. La otra ventaja es que todas las permutaciones pueden hacerse con esta representación por lo que la o las soluciones también se pueden representar, de esta manera no deberíamos preocuparnos porque no estemos abarcando todo el espacio de búsqueda de las soluciones.
Una clara desventaja que tiene esta representación es que dificulta su evaluación a la función fitness porque se tiene que hacer una representación matricial cada vez que se tiene que evaluar, aunque esto al final no es tan costoso debino a que las instancias del sudoku son de 9x9.

3. Función fitness

Después de la representación, lo primero que se hizo fue generar la función *fitness* la cual se enfocaría en ver que tan "buena.[es] una solución y para ello se enfocó en el error de la misma. Como se dijo en la sección anterior, gracias a la representación que se está utilizando no hay que preocuparse por checar los cuadros de 3x3 debido a que estos siempre estarán sin errores, lo único que hay que ver es el chequeo en filas y columnas. Ahora la manera de ver que tan buena o mala es una solución es penalizar los errores que estos tengan, por lo que si la función de fitness devuelve un cero entonces la solución será correcta para todo el sudoku. La penalización de errores se hace de manera muy intuitiva aunque se hizo una pequeña modificación pensando en un futuro cuando se implemente algún algoritmo evolutivo. Por cada par de números iguales que se encuentren en una fila o columna se agrega una penalización de un punto, pero si uno de los números es de los que venía por defecto en el sudoku la penalización será mayor (para este caso se eligió 50 puntos), esto debido a que en el caso anterior, donde los dos números eran puestos por algún algoritmo, puede que uno de ellos si esté colocado de manera correcta y ahí no habría tanto problema, pero en el segundo caso donde uno de ellos ya estaba por defecto, el que colocó el algoritmo se sabe que está mal por lo que un AE debería darle prioridad a quitarlo lo antes posible.

4. Heurística constructiva

Este algoritmo heurístico se enfocó en generar una solución inicial para el sudoku y que tuviera un error menor al que tendría una instancia completamente aleatoria. La idea principal aquí fue, partiendo de una versión de la representación mencionada anteriormente del sudoku ir construyendo una matriz de 9x9 que al final se transformaría a una representación por permutaciones del sudoku. Lo que se iba haciendo era buscar las filas con menos espacios para colocar números y posteriormente en esas filas buscar las columnas con más números colocados, y las celdas que estuvieran entre estas filas y columnas eran a las

que se les daba prioridad para colocar un dígito el cual se elegía aleatoriamente entre los que estaban permitidos a colocar y que no generaran ningún conflico entre fila y columna. En el caso donde no hubieran dígitos disponibles debido a los conflicos se colocaba alguno aleatoriamente. De esta forma se generó una solución heurística que claramente fue mejor que una construcción estocástica. Más abajo se muestran los resultados de la media de la función fitness y los tiempos de ejecución de este algoritmo con diferentes instancias del sudoku.

5. Búsqueda local

Una vez que se tiene una solución inicial generada ya sea estocásticamente o heurísticamente se procede a realizar una búsqueda local para encontrar mejores soluciones que estén contenidas en algunos de los vecinos de la instancia incial. Un vecino en la representación mencionada anteriormente es aquel que solo contiene una permutación de diferencia, es decir, un intercambio entre dos números, cuidando que no se generen números repetidos en los bloques.
La búsqueda local realizada en este trabajo itera entre los 9 bloques del sudoku generando vecinos y visitándolos, la elección de cuál es el siguiente bloque se hace de manera aleatoria. Ahora, una vez elegido el bloque se generan todos los vecinos con los números disponibles en el mismo, para ello se hacen todos los posible pares de números (sin repetición) y se almacenan en un vector de pares, esto para posteriormente elegirlos de manera aleatoria y una vez elegido un par se aplica el flip a la solución y se verifica si el fitness mejoró, de ser así la búsqueda se ejecuta desde el principio pero con ahora la nueva instancia. La búsqueda termina una vez que fueron generados todos los vecinos y ya no hubo mejores soluciones.

6. Ejecuciones y resultados

A continuación se muestra unas tablas con información resultante de hacer 100 ejecuciones con cada instancia que fue dada para el testing de los algoritmos descritos anteriormente. Una tabla mostrará la información de las ejecuciones con solución aleatoria y la otra mostrará las que tienen la heurística constructiva.

Cuadro 1: Resultados de las 100 ejecuciones usando la heurística constructiva y la búsqueda local

Instancia	Menor Fitness	Media Fitness (Heurística)	Media Fitness (B. Local)	Tiempo medio (Heurística) seg.	Tiempo medio (B. Local) seg.	Tiempo medio (Total) seg.
Easy	0	188	4	9.66e-3	0.0074	0.0075
David Filmer 1	4	216	8	9.66e-5	0.0114	0.0115
David Filmer 2	2	167	6	9.74e-5	0.0085	0.0086
Inkala	6	138	9	9.82e-5	0.0082	0.0083
Medium	6	264	9	0.00010	0.00987	0.00997
SD1	4	138	7	9.97e-5	0.0098	0.0099
SD2	5	253	10	9.947e-5	0.01049	0.01059
SD3	8	225	10	9.716e-5	0.0119	0.0120
Hard	4	336	23	9.754e-5	0.0111	0.0112

Cuadro 2: Resultados de las 100 ejecuciones usando una generación aleatoria y la búsqueda local

Instancia	Menor Fitness	Media Fitness (Aleatoria)	Media Fitness (B. Local)	Tiempo medio (Aleatoria) seg.	Tiempo medio (B. Local) seg.	Tiempo medio (Total) seg.
Easy	2	738	13	5.712e-5	0.0103	0.0104
David Filmer 1	4	672	10	5.277e-5	0.0144	0.0145
David Filmer 2	5	633	9	4.997e-5	0.01593	0.01598
Inkala	12	626	14	5.011e-5	0.01466	0.01471
Medium	7	710	10	4.838e-5	0.01346	0.01351
SD1	7	673	10	5.569e-5	0.01417	0.01423
SD2	10	772	11	5.18e-5	0.0139	0.0140
SD3	10	636	12	5.198e-5	0.01281	0.01286
Hard	4	826	39	5.426e-5	0.0172	0.0173

De las tablas anteriores podemos notar que la heurística constructiva en comparación con la solución aleatoria tuvieron gran diferencia en el costo generado por la función de fitness por lo que se puede concluir que es mejor usar la heurística al momento de adaptar algún algoritmo evolutivo para empezar de una solución medianamente buena. Otra cosa que hay que destacar es que los tiempos de ejecución en ambos casos fueron muy cortos, lo cual indica la eficiencia de las implementaciones, aunque no hay que olvidar que las ejecuciones de los algoritmos se hicieron en el cluster de CIMAT.
Ahora, un dato importante que hay que mencionar es el ratio de éxito de los algoritmos, únicamente la la instancia *Easy* fue resuelta 2 veces de las 100 ejecuciones realizadas, lo cual generaría un ratio de $\frac{2}{100}$ pero hay que mencionar que como los dos algoritmos dependen en gran parte de la generación de números aleatorios y dado el hecho de que las ejecuciones son bastante rápidas, la semilla aleatoria generada en función del tiempo de la máquina probablemente no cambiaría mucho a lo largo de las 100 ejecuciones y por lo tanto los resultados fueron muy parecidos entre sí.

Notas y comentarios. Por lo anterior comentado sobre la similitud en los resultados obtenidos no se puede realizar una comparatiba fiable con los resultados del estado del arte.