DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

Sudoku

16 de diciembre de 2015

Metaheurísticas
2do Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Kujawski, Kevin	459/10	kevinkuja@gmail.com
Ortiz de Zarate, Juan Manuel	45/10	jmanuoz@gmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

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) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

Sudoku es un juego de lógica cuyo objetivo es rellenar una cuadricula de tamaño 9x9, divididas a su vez en 9 cajas de 3x3, estas últimas llamadas cajas. Inicialmente contiene cierta cantidad de celdas con valores fijos pre-definidos y válidos, es decir, sus valores están entre [1,9], no hay repetidos por filas, columnas o cajas.
Las reglas del juego son:

1. Cada fila debe contener todos los números en el intervalo [1,9]
2. Cada columna debe contener todos los números en el intervalo [1,9]
3. Cada caja debe contener todos los números en el intervalo [1,9]

El objetivo del juego es rellenar las celdas en blanco de la cuadricula inicial respetando las reglas del juego. A pesar de las simpleza del objetivo y las reglas del juego, hay 6670903752021072936960 formas posibles de ser rellenada una cuadricula. Buscar una solución intentando todas las formas posibles es claramente imposible, y eso incentiva la utilización de una metaheurística para solucionar el juego.
En el presente trabajo, presentaremos una posible formulación matemática del Sudoku como un problema de optimización e dos propuestas de implementación utilizando dos metaheurísticas: Simulated annealing y Colonia de hormigas

2. Descripción y formulación matemática

Nuestra formulación matemática es una adaptación de (2). En esa formulación, las variables X_{ijk} son variables de decisión, definidas de la siguiente manera:

$$X_{ijk} = \begin{cases} 1 & \text{si el elemento (i,j) de la cuadricula contiene el valor k} \\ 0 & \text{en caso contrario} \end{cases}$$

La formulación como programación lineal entera es la siguiente:

$$\begin{aligned}
\min \quad & funcionobjetivo \\
\text{sujeto a} \quad & \sum_{i=1}^{9} X_{ijk} = 1, j = 1:9, k = 1:9 && (1) \\
& \sum_{j=1}^{9} X_{ijk} = 1, i = 1:9, k = 1:9 && (2) \\
& \sum_{j=3q-2}^{3q} \sum_{i=3p-2}^{3p} X_{ijk} = 1, k = 1:9, p = 1:3, q = 1:3 && (3) \\
& \sum_{k=1}^{9} X_{ijk} = 1, i = 1:9, j = 1:9 && (4) \\
& X_{ijk} = 1 \forall (i,j,k) \in INICIALES && (5) \\
& X_{ijk} \in 0,1
\end{aligned}$$

Hay que notar que como se trata de un problema de satisfacibilidad, la formulación no necesita de una función objetivo, por eso la definimos como 0. Las restricciones (1),(2) y (3) garantizan que cada numero en el intervalo posible de la instancia solo aparezca una vez en cada columna, fila, y caja, respectivamente. La restricción (4) garantiza que todas las posiciones de la martiz esten rellenas. La restricción (5) fuerza que las variables fijas de la instancia permanezcas sin alterar.

3. Descripción de la Solución

3.1. Simulated annealing

3.1.1. Definición

Simulated annealing (Recocido Simulado en español), respecto a las ciencias de la computación, es una metaheuristica para problemas de optimización global, aplicandose a la heurística de Búsqueda Local Aleatoria. La técnica fue presentada por Kirkpatrick, Gelatt y Vecci (1982) y ha sido muy empleado para resolver problemas combinatorios. La idea surge del proceso físico conocido como recocido en el cual, se eleva la temperatura de un sólido hasta el punto que se vuelve líquido, a continuación la temperatura se disminuye de forma paulatina para obtener una estructura cristalina sin defectos y que puede considerarse como un estado de mínima energía. Cada descenso de temperatura debe ser lo suficientemente pequeño para que el sistema no adquiera una estructura cristalina con defectos, además el sistema debe permanecer un tiempo suficiente a una misma temperatura para permitir alcanzar un estado estacionario, en otras palabras, que las partículas vuelvan a reacomodarse.

Se parte de una solución inicial posible S_i, luego se selecciona una solución posible S_j dentro de una vecindad, enseguida se evalúa la calidad de la solución posible empleando una función de costo f(x) asociada a cada posible solución S del problema.

Si la nueva solución posible S_j es mejor que la actual (de acuerdo al costo), se acepta, de lo contrario se selecciona de acuerdo a una probabilidad, para Simulated annealing dicha probabilidad de selección está dada por la ecuación (1) que se conoce como el criterio de Metropolis; a *temp* se le conoce como el parámetro de control; este valor se inicia con un valor suficientemente grande para que cualquier solución tenga una probabilidad alta de ser seleccionada, a medida que *temp* disminuye, la probabilidad de aceptar soluciones factibles de mala calidad disminuye. El enfriamiento se realiza empleando el sistema geométrico dado por la ecuación (2), donde α es un parámetro fijo.

$$criterio(S_i, S_j) = exp((costo(S_i) - costo(S_j))/temp) \quad (1)$$

$$temp_{k+1} = temp_k * \alpha \quad (2)$$

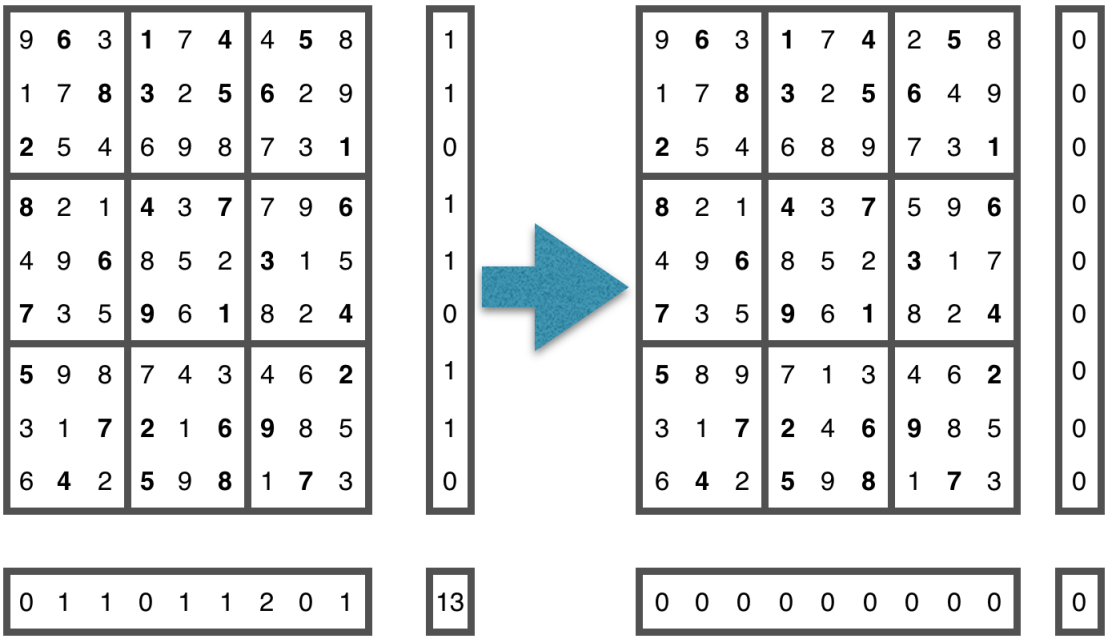
3.1.2. Implementación para resolver Sudoku

Antes de ir directo al algoritmo hay que tener en cuenta las diferentes elecciones que hicimos para encontrar una solución inicial, mejorarla y decidir cuando una solución es mejor que otra.

Solución inicial Como el algoritmo así lo requiere, debemos partir de una solución inicial para intentar mejorarla y en el mejor de los casos poder resolverla. Una primera opción que se puede pensar es, a partir de las celdas iniciales, rellenar todas las celdas vacías con valores aleatorios directamente. Sin descartar totalmente esta idea fuimos un poco más allá. Básicamente intentamos resolver el Sudoku lo más que podamos, estando siempre seguro de que los valores que pongamos en las celdas son los correctos. Para esto analizamos los posibles valores que pueden ir en cada celda, eliminando de esas posibilidades valores que estén en la misma fila, columna o caja. Luego de esto, en caso de detectar que una celda tiene un solo valor posible, se rellenará la celda con ese valor y se lo considerará correcto ya que fue deducido de la configuración inicial. En caso de que no haya valores posibles para esa celda, indicará que esa configuración inicial no tiene solución posible. Por cada celda que se rellene se deberá recalcular todo el proceso de nuevo, hasta que en algún ciclo no haya modificaciones.
Una vez que se terminó de deducir lo máximo posible será el momento de rellenar las celdas que quedaron vacias. Para este punto intentaremos ser lo menos aleatorio posible, y para tal caso seleccionaremos entre los posibles valores que pueden tener la celda, que ya habíamos calculado

en el punto anterior. Y en caso de que para una celda ya no haya posibles valores, es el momento en que se le asignará uno aleatorio, siempre respetando la formulación (3) que se menciona en la sección anterior, es decir, se respetará en todo momento que no haya celdas inválidas en las cajas.

Función de costo El costo (o penalización) va a depender de la cantidad de celdas ïlegales"que tenga la cuadricula. Es decir, cuando haya una celda que se repita en la fila, columna o caja se sumará una penalidad. El caso ideal, que indica que el tablero está solucionado, es cuando el costo devuelve 0.



Solución vecina Partiendo de una posible solución parcial, se procede a generar una solución vecina diferente pero a su vez cercana. Para ello realizamos entre 1 y 9 cambios de celdas, elegidas de manera aleatoria, siempre intercambiando entre la misma caja para seguir asegurando la formulación (3), siempre y cuando las celdas no sean las iniciales o las correctas. Estos "pequeñosçambios, son parte de la búsqueda local, y como se verá más adelante, puede no ser descartada como camino a seguir.

Algoritmo

Algorithm 1 *Sudoku con Simulated annealing*

1: ***function*** SASudokuSolver*(Inicio, α, MaxIter, $Temp_0$)*
2: $sol_0 \leftarrow$ *generarSolucionInicial(inicio)*
3: *solucionActual* $\leftarrow$ *mejorSudoku* $\leftarrow sol_0$
4: *costoActual* $\leftarrow$ *mejorCosto* $\leftarrow$ *costo(mejorSudoku)*
5: ***Mientras*** *no exceda las iteraciones máximas y no haya encontrado una solución*
6: *nuevaSol* $\leftarrow$ *busquedaLocal(solucionActual)* ▷ *Se busca solución vecina*
7: *nuevoCosto* $\leftarrow$ *costo(nuevaSol)* ▷ *Se calcula el nuevo costo*
8: *aceptacion* $\leftarrow$ *Rand(0,1)* $<$ *exp((costoActual - nuevoCosto)/temp)* ▷ *Se calcula la aceptación, con un valor aleatorio entre (0,1)*
9: ***Si*** *se acepta como la solución actual y no fue usada antes*
10: *solucionActual* $\leftarrow$ *nuevaSol*
11: *costoActual* $\leftarrow$ *nuevoCosto*
12: *Marcar la solución como ya usada*

```
        Fin Si
        Si el costo de la actual es menor que la mejor solución
            mejorSudoku ← solucionActual
            mejorCosto ← costoActual
            temp ← Temp_0                          ▷ Reiniciamos la temperatura
        Fin Si
        Si el costo de la nueva solución vecina es 0
            Devolver nuevaSol                      ▷ Devolvemos esa solución del Sudoku
            Finalizamos la función
        Fin Si
        temp ← Máximo entre 0.1 y α * temp         ▷ Le aseguramos un piso mínimo a la temperatura
    Fin Mientras
    Devolver mejorSudoku        ▷ Devolvemos la mejor solución del Sudoku que hayamos encontrado
end function
end
```

Algorithm 2 *Búsqueda local*

```
function BusquedaLocal(Solucion)
    Repetir entre 1 y 9 veces
        celdaAleatoria ← seleccionarCeldaAleatoria(Solucion)   ▷ Seleccionar una celda aleatoria
        otraCelda ← seleccionarOtraCeldaAleatoria(celdaAleatoria)   ▷ Seleccionar otra celda aleatoria del mismo cuadrante
        Si ninguna de las dos celdas seleccionadas está protegida
            Swap(Solucion, celdaAleatoria, otraCelda)          ▷ Intercambiar celdas
        Fin Si
    Fin del ciclo
    Devolver solución vecina
end function
end
```

Algorithm 3 *Generar solución inicial*

```
function GenerarSolucionInicial(SudokuInicial)
    Mientras haya cambios
        Por cada celda del suduoku
            Recorrer toda su fila y toda su columna recolectando los valores
            Intersectar la recolección con los valores del 1 al 9 ▷ Para calcular los posibles valores de la fila
            Si sólo queda un posible valor
                Asignarselo a la celda
                Proteger la celda
                Marcar que hubo cambio
            Fin Si
        Fin del ciclo
    Fin Mientras
    Por cada celda vacia
        Seleccionar uno de los posibles valores que puede tener la celda
        En caso de que no tenga, elegir un valor aleatoria que falte en la misma caja que se encuentra la celda
    Fin del ciclo
end function
end
```

4. Decisiones tomadas y su justificación

5. Experimentos

6. Conclusiones

7. Referencias

1. B. Felgenhauer, e F. Jarvis. (2006, January). Mathematics of Sudoku I.

2. A. Bartlett, T. Chartier, A. N. Langville, e T. Rankin. (2008). An Integer Pro- gramming Model for the Sudoku Problem. Journal of Online Mathematics and its Applications MAA, (8):1-14.