Metaheurísticas

Segundo Cuatrimestre de 2015

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico Final

Sudoku

Integrante	LU	Correo electrónico
Emiliano Mancuso	597/07	emiliano.mancuso@gmail.com
Gloria Diodati	285/05	gloriadiodati@gmail.com
Emiliano Hoss	664/04	emihoss@gmail.com

Índice

1. Introducción

El juego *Sudoku* es un rompecabezas lógico que trata la ubicación de números en una grilla de N^2xN^2. Este tiene celdas con valores ya fijados, llamados *pistas*, y el objetivo es completar las celdas faltantes con valores de $1..N^2$. La grilla se subdivide en N cuadrantes de NxN y se debe completar cumpliendo con las siguientes reglas:

1. Cada fila debe tener los valores de 1 a N una única vez
2. Cada columna debe tener los valores de 1 a N una única vez
3. Cada subcuadrante de NxN debe tener los valores de 1 a N una única vez

Un ejemplo de este tipo de problemas tan conocido se observa en la figura 1

	1	2	3	4	5	6	7	8	9
a			4	8					
b		9		4	6			7	
c		5					6	1	4
d	2	1		6			5		
e	5	8		7		9		4	1
f			7			8		6	9
g	3	4	5					9	
h		6			3	7		2	
i						4	1		

Figura 1: Sudoku ejemplo

Dado que es un problema NP-Completo, durante los últimos tiempos, han surgido diversos métodos para resolver de manera heurística el problema dada su complejidad.

Los métodos de resolución exacta para este problema con *Fuerza bruta* consisten en asignar posibles valores iterativamente a las celdas e ir verificando si el Sudoku cumple las reglas a medida que se siguen completando los blancos. Este algoritmo de resolución es costoso y a menudo se utiliza con backtracking.

En la actualidad se han encontrado avances en la resolución de Sudoku utilizando algoritmos genéticos [1], Búsqueda armónica [2] y Ant Colony [3] entre otros.

Para este trabajo práctico, implementamos una metaheurística de Ant Colony, y una de búsqueda local, que transforma el Sudoku en un problema de coloreo de grafos y luego utiliza DSatur [4] como heurística.

2. Búsqueda Local - DSatur

Supongamos que tenemos un problema al que queremos encontrarle una solución óptima, este problema toma diversos parámetros y por cada instancia devuelve una solución que puede ser correcta en términos de las restricciones del problema, puede ser óptima en términos de ser la mejor solución, o puede ser ninguna de las anteriores. En este último caso, evaluaremos si siguiendo por esa solución parcial del problema alcanzaremos una solución que sea a la vez correcta y óptima.

Una solución óptima puede no ser siempre alcanzable y en ese caso se busca la mejor solución. Se dice que una solución S_1 es mejor que solución S_2 si dada una función objetivo que toma instancias de solución nos permite compararlas para determinar si una es mejor que la otra.

$$f(S_1) < f(S_2) \tag{1}$$

El método de Búsqueda local entonces, itera sobre el espacio de soluciones buscando en cada paso obtener una mejor solución parcial hasta alcanzar un mínimo. Luego de la estabilización, este se detiene.

¿Qué sucede si el método no alcanzó el mínimo absoluto? Este se estanca en una solución que no se encuentra cercana al valor óptimo y terminará arrojando un mínimo local.

Ahora necesitamos definir como transformamos el problema de resolver un Sudoku a un problema de coloreo de grafos. Ademas, definiremos la función objetivo, la función de vecindad y los parámetros. Para todos los casos, utilizamos los Sudoku de $9x9$.

2.1. Transformación

La transformación es bastante trivial, cada celda del Sudoku es representada por un nodo en el Grafo, y la fila, columna y el subcuadrante asociado son los nodos adyacentes como muestra la figura 2.

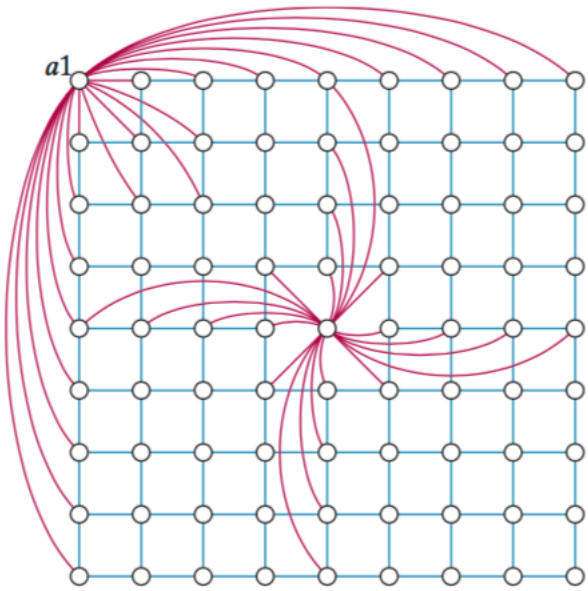


Figura 2: Nodos adyacentes

Esta es la estructura básica del grafo, luego tenemos que transformar las pistas (números que vienen fijos) en colores de manera unívoca (figura 3).

Figura 3: Asociación de Pistas a Colores

Con esto, ya contamos con el grafo inicial 4 y podemos comenzar a colorearlo con el algoritmo que elegimos.

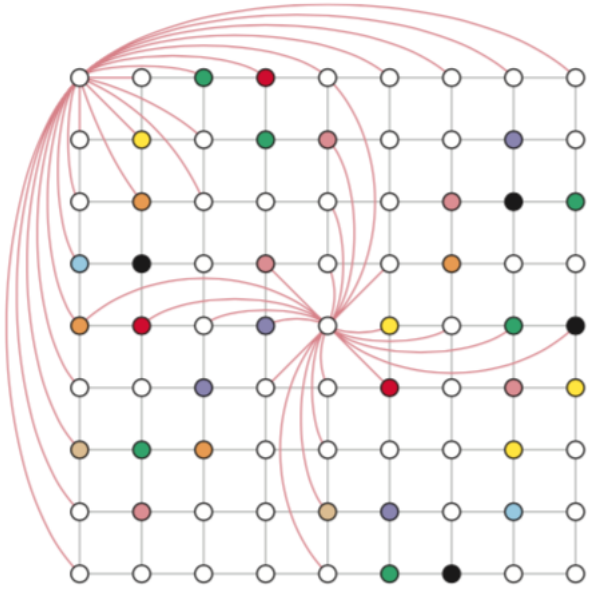


Figura 4: Grafo Inicial

A su vez, de este grafo podemos extraer varias propiedades:

- 81 nodos
- Clique Máxima: 9
- Todo nodo pertenece a una clique máxima
- $grado(n_i) = 20$
- $\chi(G) = 9$

2.2. Solución Inicial

Los algoritmos de coloreo secuencial, también conocidos como coloreo goloso van seleccionando un nodo de acuerdo a algún criterio específico, formando así un *orden* de coloración. Luego, a cada nodo lo van coloreando con un color que no tienen sus vecinos.

El orden en que se seleccionan los nodos afectan al coloreo, así que un buen orden de vértices puede traducirse en una buena coloración. No hay ninguna forma de obtener la solución óptima, sino que, se aplica el algoritmo de coloreo secuencial varias veces hasta obtener la coloración válida.

Los algoritmos de coloreo secuencial proponen las siguiente variantes:

- *Largest Degree Ordering (LDO)* - Los vértices son ordenados en orden descendente de acuerdo a su grado. La idea aquí es que los vértices de mayor grado serán más difíciles de colorear al último, y por eso se colorean primero.
- *Saturation Degree Ordering (SDO)* - A diferencia de *LDO* donde el orden de los vértices se determina de antemano, aquí los vértices son seleccionados durante el proceso de coloración. En saturación, el vértice con el grado de saturación más alta se selecciona para ser el próximo

a colorear. El grado de saturación de un vértice es el número de colores diferentes usados por sus vecinos. La idea es seleccionar el vértice que está más restringido.

- *Incidence Degree Ordering (IDO)* - Es una modificación de *SDO*, el grado de incidencia de un nodo esta definido como el número de vértices adyacentes coloreados.

2.3. SDO

Nuestra solución inicial está definida de manera de elegir los nodos con grado de saturación más alto primero, y asignarles el primer color disponible [5].

```
while (ColoredNodes < 81) {
  for all the nodes {
    node = NodeWithMaxSaturatedDegree

    if not colored(node) {
      AssignFirstAvailableColor(node)
      ColoredNodes = ColoredNodes + 1
    }
  }
}
```

2.4. Nodos Conflictivos

Dado que es un algoritmo goloso, es altamente probable que no de una solución óptima. En los casos que no se pueda asignar un color, o un número del 1-9, el algoritmo utiliza números mas grandes, y continúa buscando una solución.

Decimos que un nodo es conflictivo si cumple la siguiente condición:

$$hasColor \wedge isNotAClue \wedge (color > 9 \vee adjacentUsesSameColor) \tag{2}$$

2.5. Función objetivo

Definiremos la función objetivo como la suma de la cantidad de nodos conflictivos. Esta función vale cero cuando la solución del sudoku es correcta, y justamente queremos buscar una solución que la minimice.

```
for node in SudokuNodes {
  if Conflictive(node)
    count++
}

return count
```

2.6. Función de Vecindad - Repair/Wipeout

Definimos nuestra función de vecindad que toma un Sudoku como fue definido en 2.1.

Intercambiaremos los nodos conflictivos con el color más probable a ser correcto, y removiendo el color de los nodos que entran en conflicto con este nuevo coloreo. Tendremos cuidado de no intercambiar los colores con los ya fijados (*pistas*) por el Sudoku de entrada, pues no producirían

una solución válida. De esta manera, nos vamos moviendo en un conjunto de soluciones cercanas, reduciendo la cantidad de nodos conflictivos, hasta quedarnos con la mínima. A esta parte del algoritmo la llamamos *repair*.

Para evitar quedarnos en un mínimo local, luego del *repair*, si la solución sigue teniendo *nodos conflictivos* pasamos a la fase de *wipeout*.

Ésta consiste en tomar una cantidad dada de filas o columnas que tengan nodos conflictivos y borrarlos por completo (sin sacar las pistas), para luego empezar desde el principio con *dsatur*. Esta parte del algoritmo es esencial para escapar a los mínimos locales que podamos hallar con la *reparación*.

2.7. Parámetros

En esta metaheurística, tenemos tres parámetros para ajustar el comportamiento de la misma. Describiremos a continuación cada uno, con sus implicancias y el valor elegido por nosotros.

2.7.1. WIPEOUT ITERATIONS

Este parámetro, representa las iteraciones totales de la metaheurística. Es decir, la cantidad de veces que va a borrar parte de la Solución parcial para alejarse de un mínimo local.

Un número muy grande, incrementa el tiempo de ejecución del algoritmo y uno muy pequeño no recorre distintas vecindades y se queda con los mínimos locales.

2.7.2. REPAIR ITERATIONS

Una iteración de *repair*, implica intentar resolver los nodos conflictivos de cada solución parcial y reducir la cantidad de los mismos. Es lo que se conoce como explorar la vecindad. Cada reparación puede agregar nuevos conflictos, pero sin modificar la solución de forma significativa.

Por lo tanto un número muy grande, hace que la vecindad crezca también y se demore mucho tiempo explorando la misma, cuando mejores resultados se obtendrían volviendo a empezar.

2.7.3. SETS TO WIPEOUT

Cada vez que tenemos que hacer un *wipeout* al Sudoku, queremos borrar algunas filas o columnas enteras que contienen nodos conflictivos, para alejarnos de los mínimos locales. Éste parámetro justamente indica la cantidad de estos a borrar.

La particularidad de este valor, es que esta acotado por:

$$1 \leq SETS\_TO\_WIPEOUT \leq 18 \tag{3}$$

Las consecuencias de elegir un número muy alto hacen que se aleje más de la solución parcial, y por el contrario, mientras más chico es el número, más parecida es a la solución parcial. Si bien la intención del proceso *wipeout* es alejarse de un mínimo local, y poder ampliar la búsqueda por otra vecindad, encontramos que un valor alto para éste parámetro es perjudicial para el algoritmo, pues por lo general la solución parcial tiene 2 o 3 nodos conflictivos nada más. Si borramos gran parte del Sudoku, aumentamos la posibilidad de generar más conflictos y como consecuencia, el algoritmo le toma mucho más tiempo converger a una solución. Incluso, para Sudokus donde terminaba en menos de 10 segundos, con valores entre 6 y 11 el algoritmo no terminó en varias ocaciones.

2.7.4. Valores seleccionados

Los mejores resultados, tanto de tiempo como de Sudokus resueltos fueron en estos intervalos:

- *WIPEOUT ITERATIONS* $\in [35, 70]$
- *REPAIR ITERATIONS* $\in [70, 95]$
- *SETS TO WIPEOUT* $\in [3, 5]$

2.8. Algoritmo

Luego de haber explicado los detalles, podemos describir en alto nivel como funciona nuestro algoritmo.

```
forEach WIPEOUT_ITERATION {
  dsatur @sudoku

  forEach REPAIR_ITERATION {
    repair @sudoku
  }

  if solved {
    exit
  } else {
    wipeout @sudoku, SETS_TO_WIPEOUT
  }
}
```

3. Ant Colony

Es una metaheurística de la familia de PSO (Particle Swarm Optimization) basada en el comportamiento en grupo de las hormigas para definir el camino a un recurso deseado.

La metaheurística general consiste de lo siguiente:

En principio, todas las hormigas se mueven de manera aleatoria, buscando por si solas un camino al recurso que están buscando (una posible solución).

Una vez encontrada una solución, la hormiga vuelve dejando un rastro de feromonas; este rastro puede ser mayor o menor dependiendo de lo buena que sea la solución encontrada. Utilizando este rastro de feromonas, las hormigas pueden compartir información entre sus distintos pares en la colonia.

Cuando una nueva hormiga inicia su trabajo, es influenciada por la feromona depositada por las hormigas anteriores, y así aumenta las probabilidades de que esta siga los pasos de sus anteriores al acercarse a un recurso previamente encontrado.

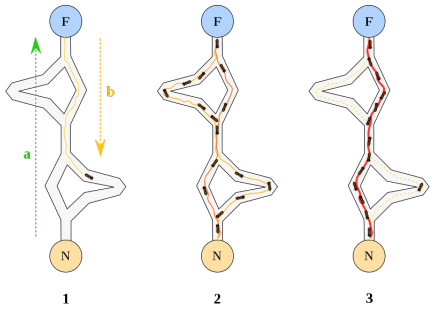


Figura 5: Ant colony

Esta feromona además tiene un factor de evaporación, esto produce que los caminos pierdan su fuerza de atracción, cuanto más largo sea el camino, más tiempo demorará una hormiga en recorrerlo, más se evaporará la feromona y por ende serán menos frecuentado. Por su parte los caminos más cortos (o más óptimos) tendrán mayor cantidad de feromonas, por ende, mayor probabilidad de ser frecuentados. Figura 5.

En computación podemos expandir este concepto biológico de interoperabilidad entre las hormigas, para construír y combinar distintas soluciones parciales a un problema. En particular utilizaremos este concepto para resolver el problema de Sudoku, el cual será representado como una grilla de 9x9 con ciertas restricciones que se detallarán a continuación.

3.1. Ant Colony para Sudoku

Nuestra implementación está fuertemente basada en el paper de Krzysztof Schif [3], tomando decisiónes sobre algunas cuestiones que no estaban debidamente cubiertas por el documento o que parecían ambiguas.

La solución (u objetivo) se establece como un tablero de 9x9 posiciones, todo número que pertenece a la solución debe cumplir las tres reglas de Sudoku. Nos obstante pueden existir posiciones vacías sin ningún valor.

Puede verse a la solución S de Sudoku, entonces, como un conjunto de duplas: [posición, dígito]. En donde no pueden existir posiciones repetidas y tampoco pueden existir un par de posiciones cuyos valores rompan las condiciones de las reglas de Sudoku.

Además, existe un tablero T, de tres dimensiones, 9x9x9. En este tablero se representan las posibles soluciones que van encontrando las hormigas y será utilizado para depositar el valor de feromona de cada elemento de acuerdo a como este pueda o no ser parte de la solución final.

Notar que este tablero no tiene en cuenta las permutaciones de los elementos, solamente sirve como guía para cada hormiga al momento de tener que realizar una elección para su siguiente paso.

Al principio del algoritmo, todos los elemento de T tienen un valor máximo de feromona. A medida que las hormigas realizan su trabajo, van depositando para cada dupla [posición, dígito] un valor de feromona de acuerdo a cuan buena fue la solución que incluye a esa dupla.

El algoritmo está compuesto por dos ciclos principales anidados, uno para repetir el trabajo en conjunto de todas las hormigas y otro para que cada hormiga realice su trabajo individualmente.

El ciclo interno es el que corre por cada hormiga. Estas intentan individualmente encontrar la mejor solución posible utilizando únicamente el nivel de feromona depositado por el conjunto de hormigas anterior.

Cada vez que las hormigas terminan su trabajo, se guarda la mejor solución comparada contra la encontrada por la hormiga anterior. Al momento en el que todas las hormigas finalizan su trabajo, se evalúa la mejor solución encontrada y se deposita un nivel de feromona igual para cada dupla *[posición, dígito]* en el tablero T, al mismo tiempo que se evapora un cierto porcentaje.

Con el objetivo de encontrar una solución al problema, cada hormiga selecciona el próximo ítem a ser agregado a la solución con una probabilidad:

$$p(j) = \frac{t_j n_j}{\sum_{i=1}^{n} t_i n_i} \tag{4}$$

Donde t_j expresa la cantidad de feromona depositada en el elemento j de la solución y n_j expresa el nivel de interés de agregar el elemento j a la solución S, que se detallará más adelante.

Además, para el caso del Sudoku, se considera que los elementos deben cumplir con las restricciones del juego, es decir, no puede elegirse un dígito que ya exista en la fila, columna o subgrilla de la posición a la que se lo quiere agregar. En ese caso se considera que la probabilidad de elegirlo es cero.

3.2. El trabajo de cada hormiga

Al principio del trabajo de cada hormiga, esta puede encontrar que existen posiciones a las cuales únicamente puede asignarles un dígito entre las nueve opciones iniciales. Figura 6.

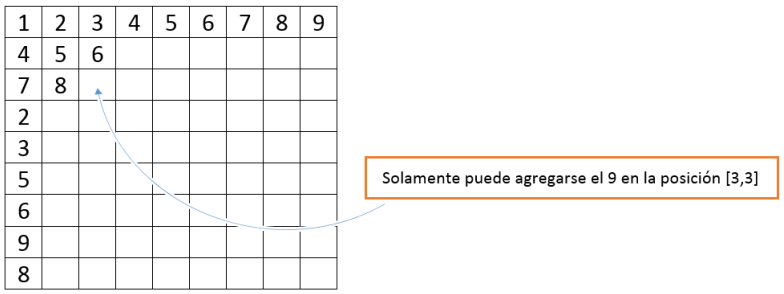


Figura 6: Ant colony

También puede ocurrir que para los dígitos posibles que puede tomar una posición de la solución, exista una subgrilla tal que solamente posea una única posición en la que pueda contener este dígito. Figura 7.

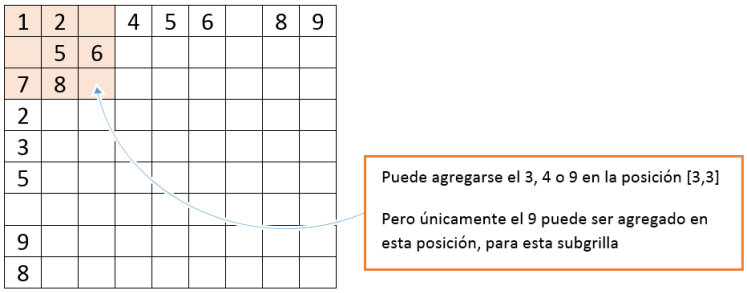


Figura 7: Ant colony

Cuando alguna de estas dos situaciones ocurre, estos valores son asignados inmediatamente.

Finalmente, y si no ocurrió ninguna de las situaciones previamente mencionadas, la hormiga toma la siguiente posición de acuerdo a una probabilidad. Esta probabilidad depende tanto del nivel de feromona como de la composición de la solución que está formando la hormiga.

Lo que la hormona realiza es ponderar aquellas posiciones y dígitos de acuerdo a lo siguiente:

- Cantidad de dígitos que puedo agregar a una posición
- Cantidad de posiciones en los que puedo agregar un dígito dentro de la subgrilla a la que pertenece dicha posición

En función de tal selección, para cada posición y dígito (i, j, k) realiza lo siguiente:

$$p(i,j,k) = (10 - places(i,j,k)) * (10 - digits(i,j)) \tag{5}$$

En donde:

- *places*: Es la función que calcula la cantidad de lugares donde puedo colocar al dígito k en la subgrilla a la que pertenece la posición (i, j)
- *digits*: Es la función que dada una posición (i, j) me devuelve la cantidad de colores que puedo colocar en dicha posición

3.3. Actualización de feromona

Una vez que todas las hormigas realizan su trabajo, se deja un rastro de feromona de acuerdo a cuan buena fue la solución encontrada entre todas.

Este cálculo se realiza comparando a la solución encontrada con la solución ideal, dividiendo la cantidad de selecciones que deberían haber hecho para encontrar la solución ideal con la cantidad de selecciones realizadas efectivamente.

Una selección es una posición dentro de la solución con un valor asignado distinto de cero. La actualización de la feromona queda de la siguiente manera:

$$dt = \frac{mayorSeleccion}{81} \tag{6}$$

Y por último se realiza la evaporación de la feromona de acuerdo a un valor fijado al comienzo del algoritmo. También hay que recordar que la feromona para este caso, no se aplica únicamente a una posición del tablero sino a la dupla *[posición, dígito]* de acuerdo a cuan buena resultó la solución final encontrada por todo un grupo de hormigas, que incluía ese dígito en esa posición.

Inicialmente todos los pares *[posición, dígito]* comienzan con un nivel de feromona de valor 1000.

Para todas las duplas *[posición, dígito]* que no se incluyan en la solución encontrada por el conjunto de hormigas estas posiciones solamente serán afectadas por el factor de evaporación, el cual es constante y es un parámetro de entrada del algoritmo. El factor de evaporación se utiliza para evitar una convergencia a un óptimo local que no sea el que estamos buscando.

3.4. Decisiones de la implementación

3.4.1. Selección random de cada paso

Una de las cuestiones que no estaba aclarada en el paper fue como calcular el siguiente paso a dar por cada hormiga. Es decir, cuales de entre todas las posiciones elegir y que dígito poner en esta posición.

El paper [3] hace referencia a que ciertas posiciones y dígitos tienen más probabilidad de ocurrir que otros. Calcular esto puede no ser trivial y una elección distinta puede llevar a resultados distintos.

Además, en el paper no se hace referencia a si esta selección la hace entre todos los elementos de la grilla o solamente entre los que todavía no fueron actualizados. Tampoco menciona si primero selecciona una posición y luego un dígito para agregar en esta, o vice versa, primero selecciona un dígito posible a agregar a la grilla y luego una posición entre las posibles a agregar este dígito.

En nuestra implementación optamos solamente considerar a los elementos que, para un determinado momento puede ser seleccionados, es decir, entre aquellas posiciones que todavía no tienen un dígito asignado y entre los posibles dígitos para esta posición.

Para implementar la selección random, evaluamos distintas opciones y utilizamos el siguiente método:

```
prob = maxProbabilidad
probabilidadAcumulada = 0

while probabilidadAcumulada < prob {
  pos = posicionRandom(posicionesAunNoSeleccionadas)
  dig = digitoRandom(digitoFactiblePara(pos))

  probabilidadAcumulada += p[pos, dig]
}
```

3.4.2. Cantidad de selecciones por iteración

Una cuestión importante es la cantidad de pasos que cada hormiga puede realizar en cada una de sus iteraciones, si puede realizar más de un paso, si realiza varios y se queda con el mejor, o si puede realizar solamente uno.

En caso de realizar varios pasos además, sera necesario contemplar que ocurre si realiza pasos que no sean compatibles o si antes de realizar cada paso deberá verificar que sea factible realizarlo.

Para nuestra implementación, cada hormiga realiza únicamente pasos que sean factibles, es decir, que no contradigan ninguna regla de Sudoku. Además, por cada iteración la hormiga realiza únicamente un paso.

3.4.3. Finalización del trabajo de la hormiga

Algo que resulta ambiguo en la descripción del algoritmo es como se finaliza el trabajo de una hormiga.

La hormiga puede terminar de realizar su trabajo cuando:

- Ya no tiene posibles posiciones en las cuales agregar un dígito.
- Cuando encuentra al menos una posición en la cual no agregar un dígito.

Además, lo mismo podría ocurrir con los dígitos respecto de las posiciones, si se encuentra un dígito que no puede ser agregado en ninguna posición de una subgrilla de 3x3.

La diferencia entre estos dos casos es que para el primero, la hormiga intentará encontrar la mejor solución posible, aún cuando ya sabe que no podrá resolver el problema por ese camino.

En el segundo caso la hormiga intentará resolver la solución hasta el punto en el que sepa que ese camino no la llevará a una solución deseada, sin intentar mejorar la solución actual (ley del menor esfuerzo).

3.5. Pseudo código

El pseudo-código a grandes rasgos queda de la siguiente manera:

```
forEach ciclo {
  forEach hormiga {
    solucionParcialDeHormiga = realizarTrabajoDeHormiga

    if (solucionParcialDeHormiga es mejor que solucionParcial ) {
      solucionParcial = solucionParcialDeHormiga
    }
  }

  if (solucionParcial es mejor que solucionFinal ) {
    solucionFinal = solucionParcial
  }
}
```

El trabajo de cada hormiga se detalla a continuación con un diagrama de flujo 8.

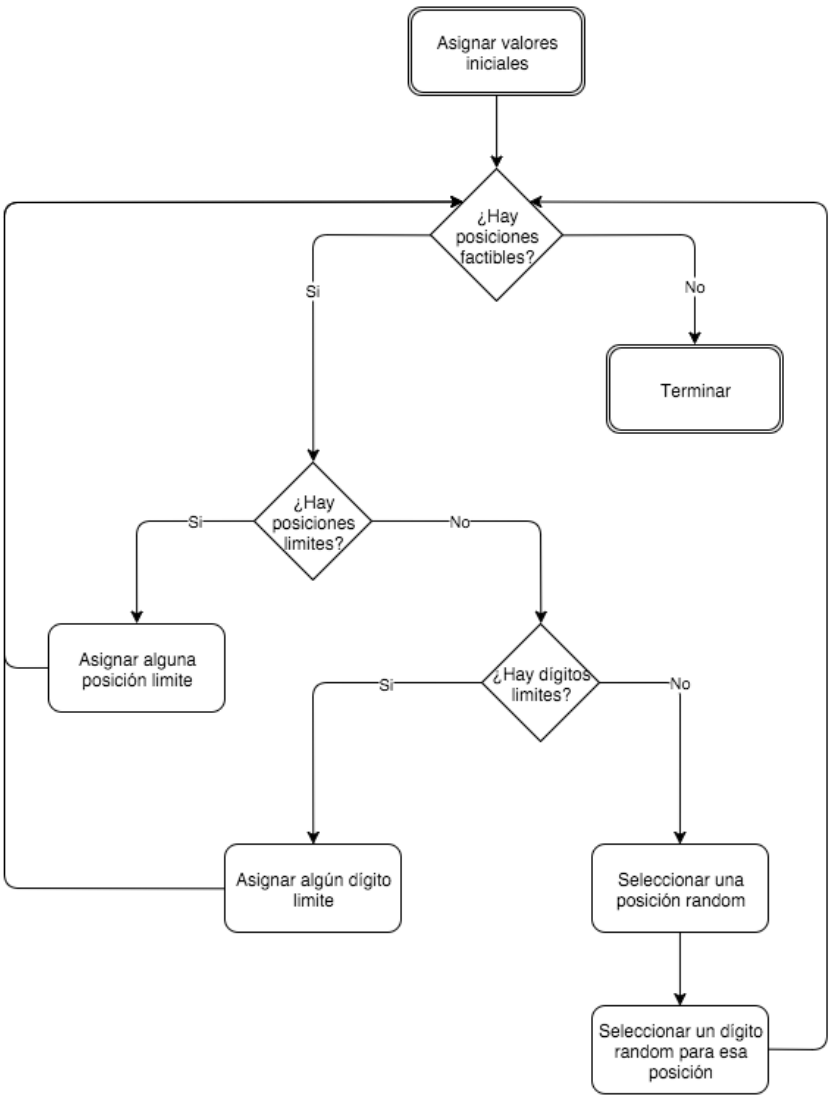


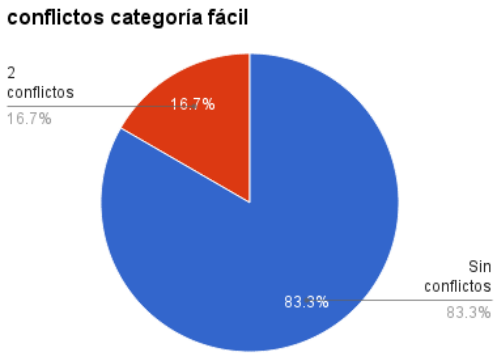
Figura 8: Ant colony - Flowchart

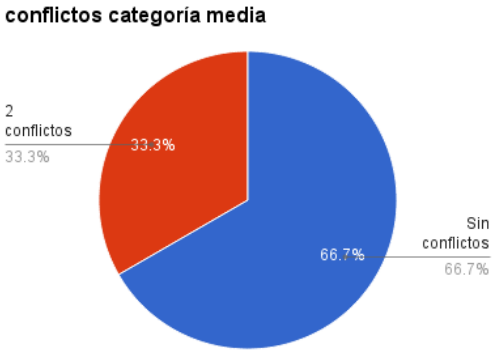
4. Casos de prueba

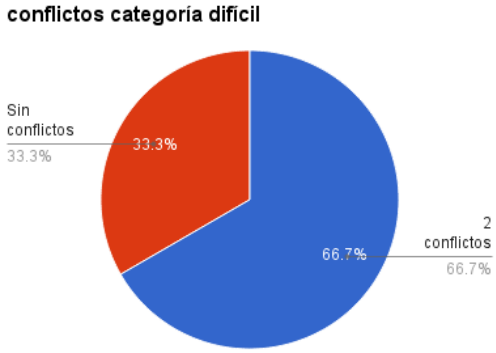
Para evaluar el comportamiento de ambos algoritmos utilizamos los casos de prueba obtenidos de http://lipas.uwasa.fi/~timan/sudoku/, ordenados en cuantro categorías: *fácil, medio, difícil y ultra difícil.*

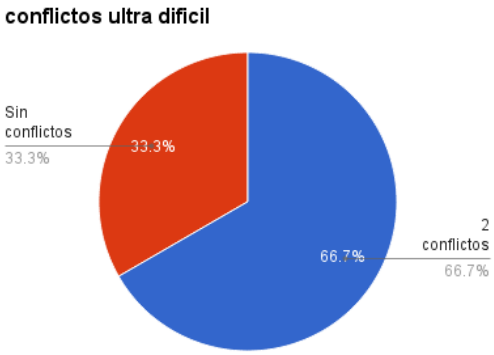
Al discutir la naturaleza de los métodos de resolución, supusimos que aquellos dentro de la categoría *fácil* no iban a presentar dificultades en dicho proceso. Con respecto a aquellos categorizados como dificultad *media*, *difícil* y *ultra difícil* creímos que los algoritmos podrían acercarse a alguna solución pero conservar algunos conflictos.

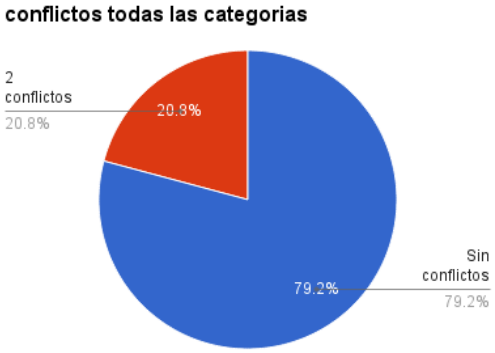
4.0.1. Búsqueda Local - DSatur











Como resultado de la experimentación podemos ver que aunque los casos en los que no se pudo calcular una solución se encuentran en mayor proporción en la categoría *difícil* y *ultra difícil*, en todas las categorías hubo instancias que presentaron conflictos.

4.1. Ant Colony

Para el caso de los test con Ant Colony no incluiremos gráficos de porcentajes de problemas resueltos y no resueltos dado que logramos resolverlos todos.

Entre los problemas que resolvimos, la mayoría los resolvió rápidamente, nunca demorando más de 5 minutos. En la mayoría de los problemas fáciles no se llegaba ni a la primera iteración de un ciclo principal de Ant Colony, es decir, el primer grupo de hormigas encontraba la solución.

Esto es importante de notar dado que significa que el algoritmo está logrando encontrar la solución partiendo desde la heurística de cada hormiga, sin compartir entre sí la feromona. Para problemas más avanzados si se notó la utilización de 1 o más ciclos principales.

Utilizamos estos problemas más avanzados para evaluar los distintos parámetros del algoritmo. En particular nos interesó evaluar como afectaba la cantidad de hormigas y el nivel de feromona compartido entre ellas a la performance del algoritmo.

Comenzamos evaluando cuanto se modificaban los tiempos para encontrar una solución aumentando la cantidad de hormigas de 10 en 10 con una misma instancia (la más difícil de nuestro set de prueba). Los resultados se pueden ver en la figura 9.

Posteriormente evaluamos que ocurría si decrementábamos la cantidad de feromona de 0,999 a 0,809 utilizando el mejor valor obtenido en la prueba anterior para la cantidad de hormigas.

Los resultados son los de la figura 10:

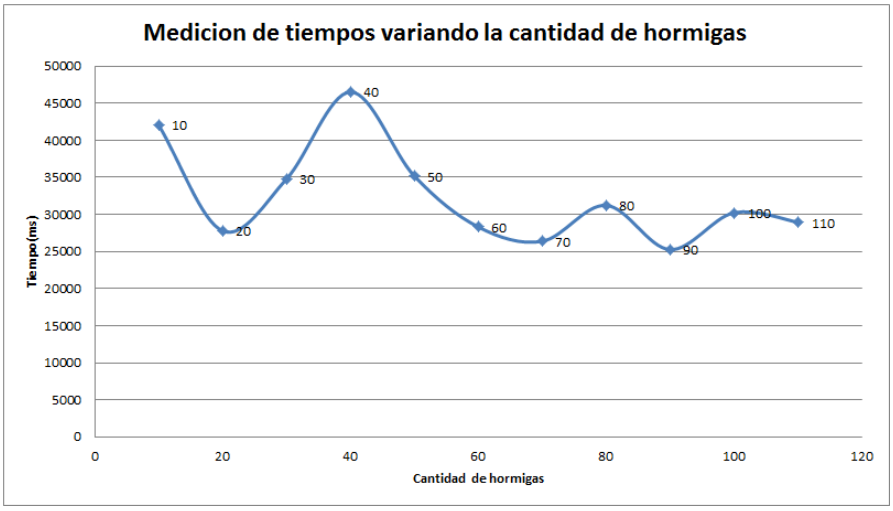


Figura 9: Tiempo / Hormigas

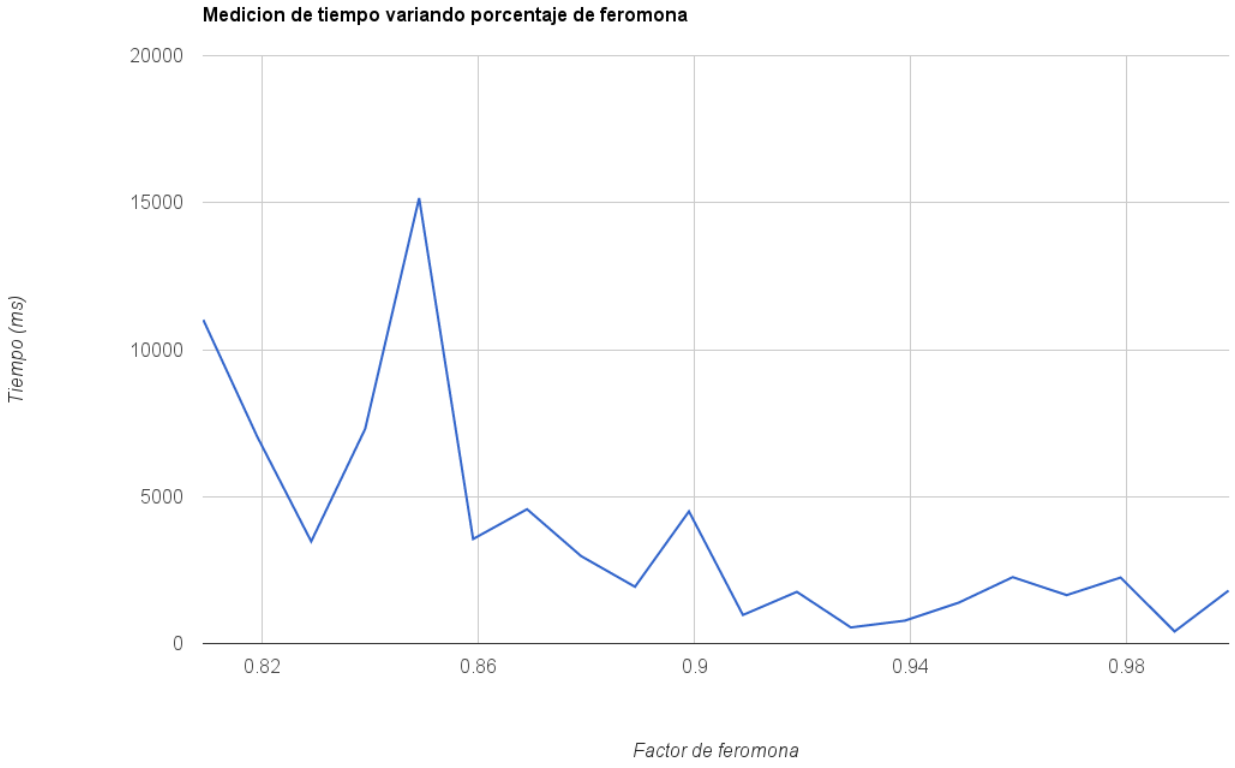


Figura 10: Feromona / Tiempo

4.1.1. Ambas metaheurísticas

Dado que *Ant Colony* resuelve todos los Sudoku que probamos pero la de *Búsqueda Local* no siempre, comparamos estas metaheurísticas con las instancias que los 2 terminaron en un tiempo comparable.

Las instancias más difíciles quedaron fuera de este gráfico, pues la metaheurísticas de *Búsqueda Local* tomaba demasiado tiempo, incluso no termino en algunos casos.

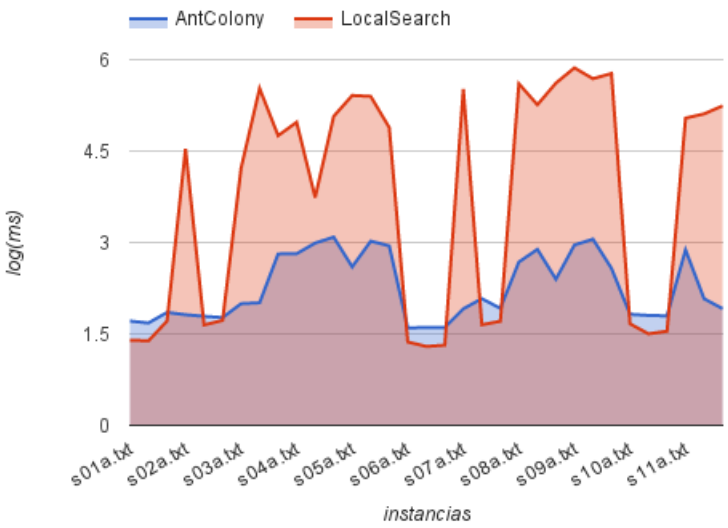


Figura 11: Tiempos de ambas metaheurísticas

Como podemos observar, más allá de la diferencia de los tiempos, el comportamiento o la dificultad de los Sudoku para los algoritmos se corresponden entre sí, pero no con la categoría.

En casi todos los casos más sencillos para los dos, a *DSatur* le va mejor. Pero en los casos que a ambos les cuesta más llegar a una solución, *Ant Colony* converge mucho más rápido que la otra metaheurística. Figura 11.

Esto puede pasar por que a *Búsqueda Local* le cuesta salir de un mínimo local que no es solución al Sudoku o porque *Ant Colony* encuentra una solución inicial más cercana a la solución final.

5. Discusión

En este trabajo experimentamos con la resolución del problema de Sudoku con dos heurísticas distintas: de búsqueda local y Ant Colony.

Encontramos que no hay relación entre la dificultad de resolver los problemas de manera lógico deductivo con la resolución algorítmica utilizando estos métodos.

La facilidad de implementación de estos algoritmos los hacen excelentes candidatos para poder determinar una solución inicial para un problema combinatorio de estas características. Sin embargo es probable que no encuentren el mínimo global dado las limitaciones que presentan en su diseño.

Respecto a Ant Colony lo que notamos es que la cantidad de hormigas a utilizar para problemas categorizados como fáciles es despreciable con respecto a el tiempo de ejecución, dado que encuentra una solución con menos de 30 hormigas promedio.

Por otro lado puede apreciarse una mayor relación respecto a la cantidad de hormigas que se utilizan y su resultado en los tiempos de ejecución en problemas difíciles. Llegamos a que un valor adecuado seria de 75 hormigas para estos casos.

En cuanto a las variaciones al factor de evaporación de la feromona, nuevamente para los casos de la categoría *fácil* este valor no afecta, dado que no llega a utilizarse la feromona en la búsqueda de la solución óptima.

No obstante para los problemas categorizados como *difíciles*, este valor toma mayor significado, dado que al iterar más de una vez, la feromona comienza a ser compartida y utilizada por las iteraciones posteriores de grupos de hormigas. En caso de tener un valor de evaporación de feromona inferior a los 0,95 el tiempo de convergencia e incluso la convergencia misma pueden resultar afectadas negativamente.

Vale la pena mencionar que los resultados son distintos a los que se ven en el paper de donde tomamos la idea del algoritmo. Nuestra implementación resultó funcionar mucho más rápido en términos de cantidad de ciclos y de hormigas.

Creemos que estas diferencias pueden ser por dos cosas:

Por un lado podría ser que no probamos el mismo caso que fue evaluado en el paper. Si bien comparamos nuestras corridas con instancias de pruebas de la base de datos mencionada en el paper original, no supimos con exactitud cuales fueron las instancias que demoraron el algoritmo tantos ciclos como se ve en las gráficas. Por otro lado pensamos que la diferencia pueda deberse a las diferencias de implementación que pueden existir entre nuestra implementación y la original, dado que varios detalles de la misma no se mencionan.

Referencias

[1] Mantere T. Solving, rating and generating sudoku puzzles with ga. *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, 1:1382–1389, 2007.

[2] Zong Woo Geem. Harmony search algorithm for solving sudoku. *11th International Conference, KES 2007, XVII Italian Workshop on Neural Networks, Vietri sul Mare, Italy, September 12-14, 2007. Proceedings, Part I*, 1:371–378, 2007.

[3] Krzysztof Schiff. An ant algorithm for the sudoku problem. *Journal of Automation, Mobile Robotics & Intelligent Systems*, 9(2):24–27, 2015.

[4] D. Brelaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.

[5] Dr. Hussein Al-Omari and Khair Eddin Sabri. New graph coloring algoritms. *American Journal of Mathematics and Statistics*, 2:739–741, 2006.