Solución eficiente de Sudokus por medio del problema de satisfacción de restricciones o CSP

Adelina Figueira 15-10484

Universidad Simón Bolívar

Abril 2024

Objetivos del proyecto

- Conocer que heurísticas son capaces de retornar el resultado de un Sudoku en el menor tiempo posible.
- Encontrar que algoritmos de propagación de restricciones son más eficientes para la resolución del problema del Sudoku.
- Verificar si el uso de forward-checking durante el backtracking mejora o no el rendimiento de los algoritmos CSP para la resolución del Sudoku.

Técnicas utilizadas

- Heurísticas : MRV (Valores mínimos restantes), primera variable no asignada, valores de dominio no ordenados y LCV (Valores con menos restricciones).
- Forward Checking
- MAC (Mantener la consistencia de arco)
- AC3
- AC4
- Backtracking

Backtracking

Propagación de las restricción al resolver un problema CSP

```
def backtracking_search(csp, select_unassigned_variable=first_unassigned_variable,
                        order_domain_values=unordered_domain_values, inference=no_inference):
    def backtrack(assignment):
        if len(assignment) == len(csp.variables):
            return assignment
        var = select_unassigned_variable(assignment, csp)
        for value in order_domain_values(var, assignment, csp):
            if 0 == csp.nconflicts(var, value, assignment):
                csp.assign(var, value, assignment)
                removals = csp.suppose(var, value)
                if inference(csp, var, value, assignment, removals):
                    result = backtrack(assignment)
                    if result is not None:
                        return result
                csp.restore(removals)
        csp.unassign(var, assignment)
        return None

    result = backtrack({})
    assert result is None or csp.goal_test(result)
```

Forward-checking

Variación del algoritmo de backtracking que disminuye el espacio de búsqueda haciendo uso de un tipo de consistencia local.

```
def forward_checking(csp, var, value, assignment, removals):
    """Prune neighbor values inconsistent with var=value."""
    csp.support_pruning()
    for B in csp.neighbors[var]:
        if B not in assignment:
            for b in csp.curr_domains[B][:]:
                if not csp.constraints(var, value, B, b):
                    csp.prune(B, b, removals)
            if not csp.curr_domains[B]:
                return False
    return True
```

Propagación de restricciones AC3

El algoritmo AC3 se encarga de examinar los arcos entre los pares de variables (x,y).

```
def AC3(csp, queue=None, removals=None, arc_heuristic=dom_j_up):
    """[Figure 6.3]"""
    if queue is None:
        queue = {(Xi, Xk) for Xi in csp.variables for Xk in csp.neighbors[Xi]}
    csp.support_pruning()
    queue = arc_heuristic(csp, queue)
    checks = 0
    while queue:
        (Xi, Xj) = queue.pop()
        revised, checks = revise(csp, Xi, Xj, removals, checks)
        if revised:
            if not csp.curr_domains[Xi]:
                return False, checks  # CSP is inconsistent
            for Xk in csp.neighbors[Xi]:
                if Xk != Xj:
                    queue.add((Xk, Xi))
    return True, checks  # CSP is satisfiable
def revise(csp, Xi, Xj, removals, checks=0):
    """Return true if we remove a value."""
    revised = False
    for x in csp.curr_domains[Xi][:]:
        # If Xi=x conflicts with Xj=y for every possible y, eliminate Xi=x
        # if all(not csp.constraints(Xi, x, Xj, y) for y in csp.curr_domains[Xj]):
        conflict = True
        for y in csp.curr_domains[Xj]:
            if csp.constraints(Xi, x, Xj, y):
                conflict = False
            checks += 1
            if not conflict:
                break
        if conflict:
            csp.prune(Xi, x, removals)
            revised = True
    return revised, checks
```

Propagación de restricciones AC4

Se creó para mejorar el algoritmo AC3.
Dado un valor a para la variable X_i, decimos que a es soportada por X_j si existe al menos un valor b en $Dom(X_j)$ tal que $X_i = a$ y $X_j = b$ son compatibles.

En el contador para cada arco hay un número que registra el número de soportes que recibe de cada restricción que incluya la variable, si este contador llega a 0, se elimina del dominio de la variable. El conjunto de soporte para el valor de cada variable contiene todas las variables que lo soportan.

MAC (Mantaining Arc Consistency)

Es una implementación de AC3 con doble soporte en la heurística de dominio
Utiliza como heurística un conjunto de los valores restantes consistentes de una variable.
Con esta heurística se rellena la cola queue que se inicializa al comenzar el algoritmo.
La heurística que se utiliza para este algoritmo es la heurística Domjup que ordena por tamaño los dominios para cada uno de los arcos que forman parte del problema

Seleccionar variable no asignada

MRV

Consiste de tomar el valor mínimo restante para una asignación que está en las variables del problema y no incumple las restricciones establecidas.

Primera variable no asignada

Solo toma la primera variable que se consiga y no esté en los valores ya asignados.

Orden de los valores del dominio

LCV

Selecciona los valores que no han sido verificados para una variable en particular y retorna estos valores ordenados.

Valores del dominio no ordenados

Retorna los valores del dominio de una variable sin verificar si se trata o no de valores ya tomados y tampoco son ordenados.

Antecedentes

'Why AC-3 is Almost Always Better than AC-4 for Establishing Arc Consistency in CSPs' - Richard Wallace

AC3 es menos eficiente que AC4, este algoritmo es mejor que AC4 para CSPs basados en relaciones generalizadas.
Los problemas que minimizan la eficiencia de los algoritmos de consistencia de arco, como dominios grandes o un gran número de restricciones, suelen tener un mayor efecto en AC4 que en AC3.

Antecedentes

AC3 puede:

- Simplificar un problema al reducir el dominio de las variables.
- Puede manejar varios tipos de restricciones.
- Se puede utilizar junto a otros algoritmos.
- La poda puede ser agresiva, llegando a la pérdida de datos.
- Solo para restricciones binarias.

Herramientas y librerías

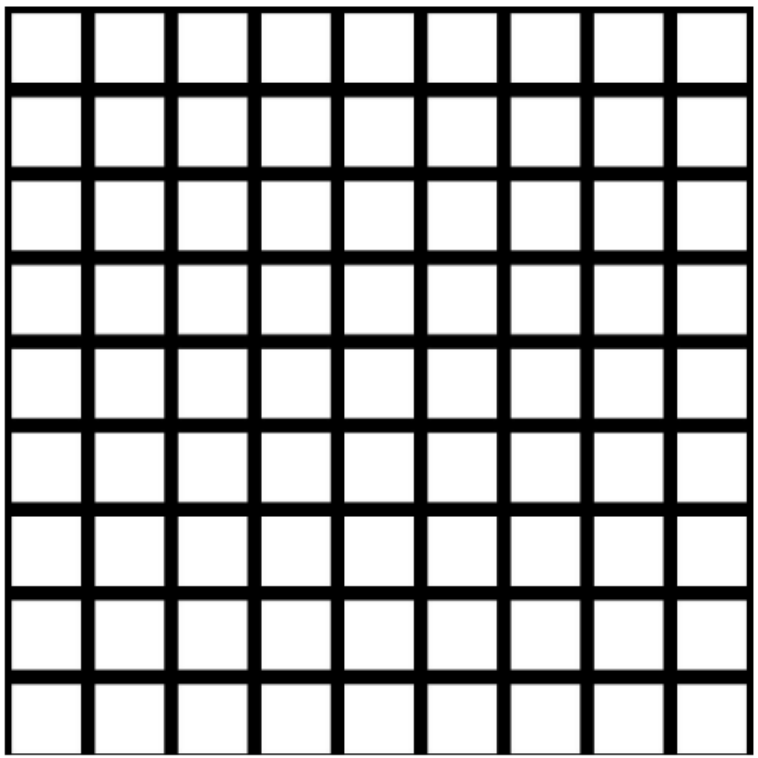
Se utilizó la librería AIMA de Peter Norvig.

La librería AIMA desarrolla el problema de Sudoku con el uso de la restricción que indica que dos cuadros vecinos no deben tener el mismo valor, hacia arriba o hacia abajo.

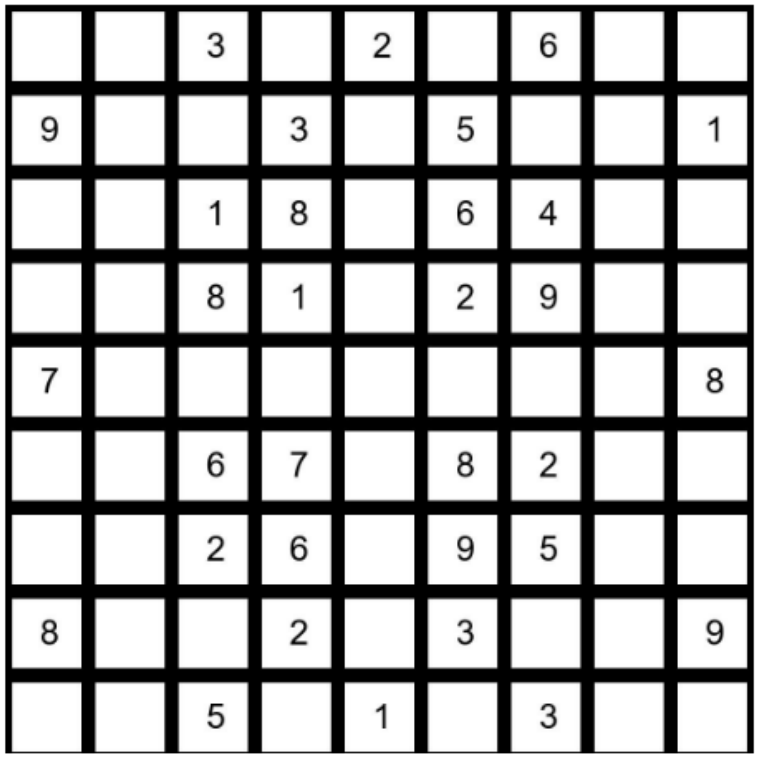
Librería multiprocessing de Python para hacer timeout a corridas con un tiempo mayor a 120 segundos.

Experimentos

Tableros 9x9



EMPTY



EASY1

Experimentos

Tableros 9x9

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

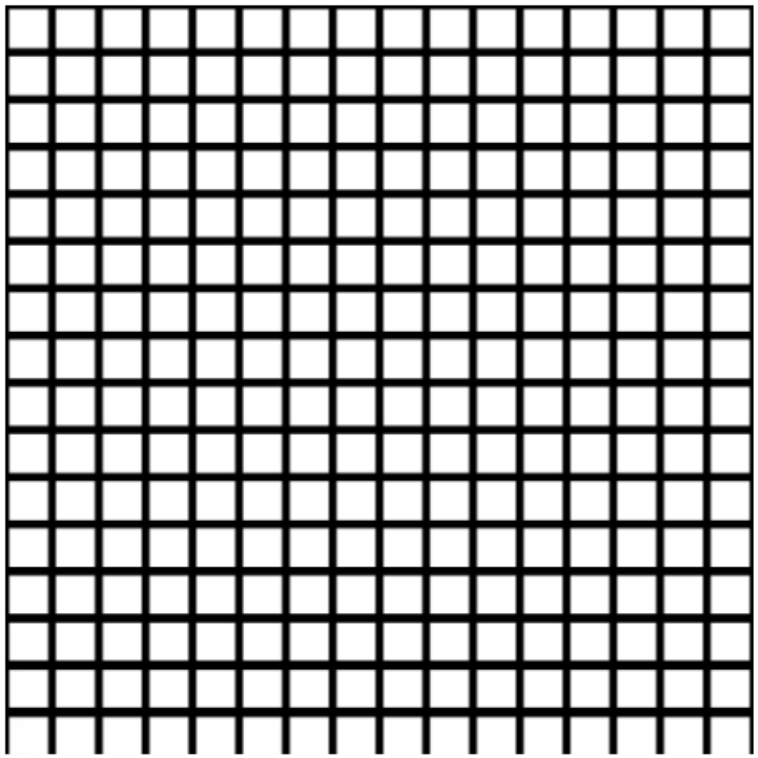
HARD1

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

HARD2

Experimentos

Tableros 16x16

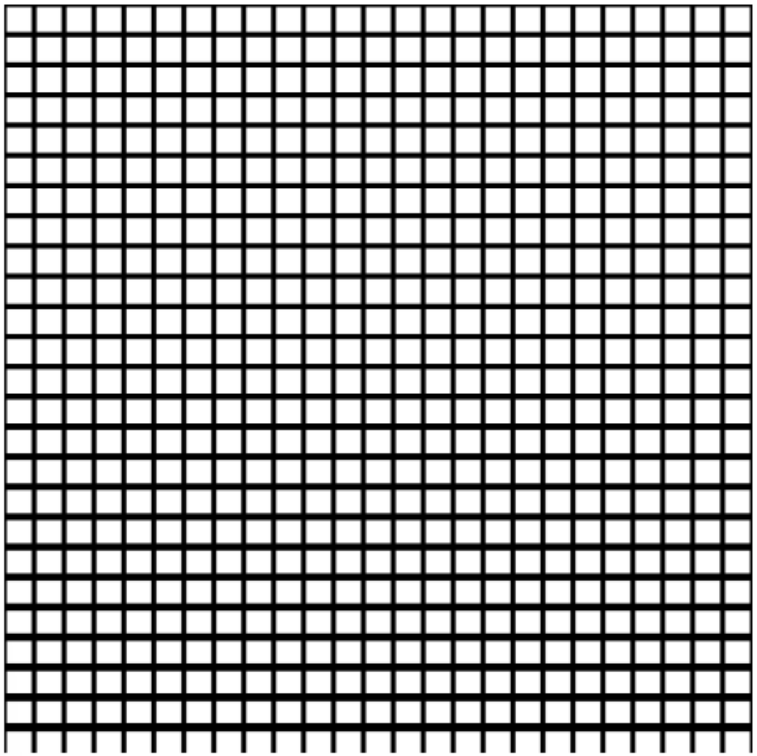


EMPTY

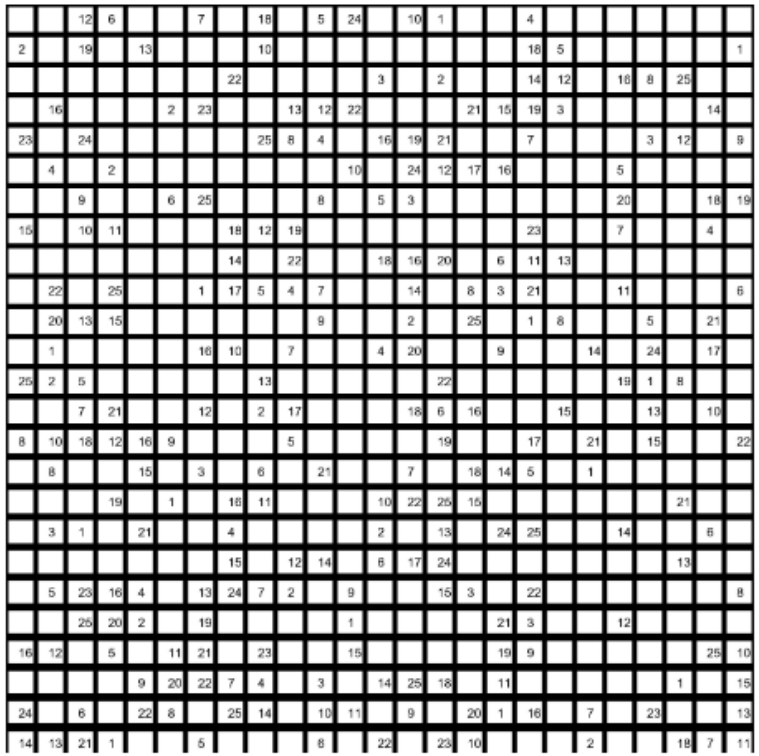
1			2	3	4			12		6				7	
		8				7			3			9	10	6	11
	12			10			1		13		11			14	
3			15	2			14				9			12	
13				8			10		12	2		1	15		
	11	7	6				16				15			5	13
			10		5	15			4		8			11	
16			5	9	12			1						8	
	2						13			12	5	8			3
	13			15		3			14	8		16			
5	8			1				2				13	9	15	
		12	4		6	16		13			7				5
	3			12				6			4	11			16
	7			16		5		14			1			2	
11	1	15	9			13			2				14		
	14				11		2			13	3	5			12

HARD16

Experimentos



EMPTY



HARD25

Resultados experimentos tablero 9x9

Para los tableros de 9x9, se obtienen resultados menores a 1 segundo para cualquier tablero, ya que se trabaja con una cantidad pequeña de datos y todas las técnicas pueden ser utilizadas e igualmente obtener un resultado en un corto período de tiempo.

Resultados experimentos tablero 16x16

Para el tablero Hard16 se obtiene el peor tiempo al aplicar las heurísticas de dominios no ordenados y de la primera variable no asignada, con el uso del algoritmo AC4.

Para el tablero vacío, también ocurre lo mismo.

El mejor tiempo para el tablero vacío toma en cuenta las heurísticas MRV, LCV y el algoritmo MAC.

El mejor tiempo para el tablero Hard16, también se logra con las mismas heurísticas y el algoritmo MAC.

Para el tablero vacío y para el tablero Hard16 al utilizar el algoritmo AC4 se toma un tiempo mayor que para el algoritmo AC3.

Resultados experimentos tablero 25x25

Para el **tablero vacío** el mínimo ocurre cuando se aplican las heurísticas MRV, LCV y se hace el backtracking.
El máximo se alcanza cuando se hace uso de las heurísticas de primera variable no asignada, dominios no ordenados y se hace uso del algoritmo AC4.
Se observó que los resultados que utilizan AC4 tienen un tiempo mayor a los que usan el algoritmo AC3, además al usar AC3 se requieren de heurísticas específicas, si se toma la primera variable no asignada o el dominio sin ningún orden, el algoritmo tiene un rendimiento deficiente.

Resultados experimentos tablero 25x25

Para el tablero **Hard25** se obtiene un tiempo menor al utilizar las heurísticas MRV, LCV y backtracking.
Mientras que el que tiene el peor tiempo con el algoritmo AC4, las heurísticas de primera variable no asignada, dominios no ordenados y forward-checking.
El forward-checking se beneficia de ciertas heurísticas como MRV y LCV, al igual que de AC3.
Para este tablero el uso de AC4 tiene en promedio un tiempo de ejecución mayor al de AC3.

Análisis de Resultados

Para el tamaño 25x25 el algoritmo backtracking con las heurísticas de MRV y LCV nos permiten obtener el resultado más rápido.

Los Sudokus vacíos son las versiones de menor dificultad para cualquiera de los algoritmos que se han utilizado. Pero para los casos 9x9 y 16x16 al usar el algoritmo AC4 con diferentes heurísticas se obtiene un tiempo mayor en comparación a otros algoritmos.

El forward-checking puede ser de utilidad siempre que no se utilice junto a la heurística de la primera variable no asignada.

El algoritmo MAC tuvo un buen rendimiento para Sudokus de gran tamaño.

Conclusiones

Una vez realizados los experimentos para Sudokus de 9x9, 16x16 y 25x25 podemos afirmar que la mejor solución para Sudokus de cualquier tamaño viene dada por:

- AC3 o MAC
- Backtracking
- Heurística MRV para seleccionar la variable no asignada durante el backtracking
- Heurísticas LCV para el orden de los dominios

Para el problema de Sudoku, AC3 obtiene mejores resultados que AC4.

Referencias

- Wallace, R (1993). Why AC-3 is Almost Always Better than AC-4 for Establishing Arc Consistency in CSPs. Department of Computer Science, University of New Hampshire Durham, NH 03824, U. S. A. https://www.ijcai.org/Proceedings/93-1/Papers/034.pdf
- Peter Norvig et al. AIMA library. https://github.com/aimacode/aima-python
- Capítulo 10. Problema de satisfacción de restricciones (CSP). https://personales.upv.es/misagre/papers/capitulo.pdf
- Russell, S. J. 1., & Norvig, P. (2003). Artificial intelligence: a modern approach. Prentice Hall.