Solucionando Sudoku utilizando CSP

André Robert Molin
Ciência da Computação - Inteligência Artificial
Insituto Federal Catarinense
Videira, Brasil
andrerobert4096@gmail.com

***Resumo*—A utilização do método de satisfação de restrições no intuito de resolver problemas abre muitas possibilidades, podendo ser utilizados através de inúmeros métodos já conhecidos, com algoritmos genéricos padrões com possibilidade de alterações para a adequação ao problema em questão. Logo, o objetivo deste artigo é compreender o funcionamento do método de satisfação de restrição, bem como seu funcionamento e técnicas de resolução de problemas, através de um experimento, sendo este o desenvolvimento de uma proposta de solução do jogo "Sudoku", qual pode ser resolvido através de técnicas de CSP,utilizando-se do método Backtracking e Arc-Consistency-3 para esta proposta. Com a implementação, será exibido a metodologia utilizada no desenvolvimento do algoritmo, resultados encontrados, além da conclusão elaborada após resultados do desenvolvimento, comparando os dois métodos utilizados.**

***Palavras-chave*—Inteligência Artificial, CSP, Sudoku, Backtracking, AC-3**

I. INTRODUÇÃO

A CSP(Problema da satisfação de restrições, do inglês *Constraint satisfaction problems*), é um método geral de formulação de problemas em que seu objetivo é encontrar valores para variáveis de forma que estes não violem restrições que se mantém entre as variáveis. Esta formulação de problema pode ser usada para resolver muitos problemas em inteligência artificial, ciência da computação e engenharia [2].

Problemas de satisfações de restrições apresentam alta complexidade e exigem que sejam usados métodos heurísticos e de busca combinatória para que se possa resolver tais problemas em tempo aceitável. SAT(problema de satisfatibilidade booleana), SMT(satisfatibilidade de módulos teóricos) e ASP(programação de conjunto de respostas) aproximam-se do problema de satisfação de restrições [4].

Em inteligência artificial e pesquisa operacional, a satisfação das restrições é o processo de encontrar uma solução para um conjunto de restrições que impõem condições que as variáveis devem satisfazer . Portando uma solução é um conjunto de valores para as variáveis que satisfaz todas as restrições, ou seja, um ponto na região viável.

O jogo Sudoku é um quebra-cabeça combinatório em que os números são colocados em uma grade de 81 blocos, divididos em 9 blocos (9x9), que é dividida em 9 sub-grades (3x3). Existem algumas variações com 144 blocos (12x12), divididos em 12 sub-grades(4x3). O jogo se incia com grades parcialmente preenchidas, e cada Sudoku possui uma solução. O objetivo do jogo é que cada um dos 9 blocos deve conter todos os números de 1 a 9 em seus respectivos quadrados [3].

II. MÉTODOLOGIA CSP

Consiste de classes de problemas que satisfazem algumas propriedades estruturais além dos requisitos básicos para problemas em geral. O problema da satisfação de restrições sobre domínios finitos é resolvido usado uma forma de algoritmo de busca, como variantes de algoritmos de backtracking, busca local e propagação de restrições. [2].

A formula básica para compreensão da satisfação das restrições consiste de:

- Conjunto de variáveis
- Domínio para cada variável
- Conjunto de restrições

CSP tem o objetivo de escolher um valor para cada variável de forma que o mundo possível resultado atenda suas restrições. Logo, um CSP finito possui um conjunto finito de variáveis e um domínio finito para cada variável. Cada variável pode ser vista como uma dimensão separada, tornando difíceis de resolver, porém fornecendo uma estrutura qual pode ser explorada [8].

É definido com uma tripla *(X,D,C)*, onde *X* é um conjunto de variáveis, *C* é o domínio dos valores e *D* é um conjunto de restrições. Então, toda restrição c $\in$ C é um par (τ,R), normalmente representado por matriz, onde τ é um *n*-tupla de variáveis e R é uma relação matemática *n*-aria em *D*. Uma avaliação das variáveis é uma função de variáveis para o domínio de valores ν: *X*$\rightarrow$*D*. Uma avaliação ν satisfaz as restrições $\langle$(x',...,xn),R$\rangle$ se ($\nu(x')$,...,$\nu(xn)$) $\in$ R. Só pode ser considerado uma solução quando as restrições do problema são satisfeitas [4].

A. Problemas populares com CSP

Os problemas a seguir são alguns dos problemas que podem ser solucionados usando a metodologia de satisfação de restrições.

- CryptArithmetic (codificação de alfabetos em números)
- O problema das oito rainhas
- Palavras-cruzadas
- Coloração de mapas(grafos)
- Sudoku
- Diversos puzzles lógicos

III. RESOLVENDO SUDOKU COM CSP

Para resolver o jogo Sudoku utilizando o método de satisfação de restrições, foi utilizado das metodologias de *backtracking*, e Arc-Consistency 3 [7].

Para este problema, as variavéis são indicadas pela linha, e o digito indica a coluna X = $\{X1, X2, ..., X81\}$, sendo 81 possibilidades de números. Para o domínio, cada variável pode possuir valores de 1 a 9. D= {D1,...,D81}, logo que Di = {1,2,4,5,6,7,8,9}. Como restrições, o valor de cada variável Xi não pode ser igual a nenhum valor em sua linha, coluna e quadro.

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

Fig. 1. Exemplo - Jogo Sudoku

A. *Mas o que é backtracking?*

Consiste em um refinamento do algoritmo de busca por força bruta (ou enumeração exaustiva), no qual boa parte das soluções podem ser eliminadas sem serem explicitamente examinadas, mantendo apenas soluções que satisfazem as restrições do problema inicial. De maneira incremental, busca por candidatos à soluções e abandona cada candidato parcial C quando C não pode resultar em uma solução válida.Quando sua busca chega a uma extremidade da estrutura de dados, como um nó terminal de uma árvore, o algoritmo realiza um retrocesso tipicamente implementado através de uma recursão [8].

Exemplo de código backtracking:

```
bool acabou = FALSE;

backtrack(int a[], int k, int n) {
int c[MAXCANDIDATOS];
int ncandidatos;
int i;

if (e_uma_solucao(a, k, n)) {
processar_solucao(a, k, n);
} else {
k = k + 1;
construir_candidatos(a, k, n, c, &ncandidatos);
for (i=0; i<ncandidatos; i++) {
        a[k] = c[i];
        backtrack(a, k, n);
        if (acabou) return;
  }
  }
}
```

B. *Mas o que é Arc-Consistency 3?*

O algoritmo AC-3 é um de uma série de algoritmos usados para a solução de problemas de satisfação de restrições. [6] Sua implementação é feita através de um loop simples que seleciona e analisa as restrições armazenadas em Q até que nenhuma mudança ocorra (Q está vazio) ou o domínio de uma variável se torna vazio. O primeiro caso garante que todos os valores dos domínios sejam consistentes com todas as restrições, e o segundo caso retorna que o problema não tem solução. [1] O arco esta entre duas variáveis que possuem a mesma restrição, sendo consistente entre elas quando respeita as seguintes restrições.

- A $\rightarrow$ B é consistente: para cada valor a que A pode assumir, há um valor b que B pode assumir respeitando a restrição.
- B $\rightarrow$ A é consistente: para cada valor b que B pode assumir, há um valor a que A pode assumir respeitando a restrição.

Basicamente, analisa arcos em pares, apagando conflitos detectados durante o retrocesso.

Exemplo de código arc consistency:

```
Entrada:
  Um conjunto de variáveis X
  Um conjunto de domínios D (x) para cada variável
  ↪  x em X. D (x) contém vx0, vx1 ... vxn, os
  ↪  valores possíveis de x
  Um conjunto de restrições unárias R1 (x) na
  ↪  variável x  deve ser satisfeita
  Um conjunto de restrições binárias R2 (x, y) nas
  ↪  variáveis x e y que devem ser satisfeitas

Resultado:
#Domínios de arco consistentes para cada variável.

função ac3 (X, D, R1, R2)
#Domínios iniciais são feitos consistentes com
↪  restrições unárias.
    para cada x em X
        D (x): = {vx em D (x) | vx satisfaz R1
        ↪  (x)}
    // 'lista de trabalho' contém todos os arcos
    ↪  que desejamos provar consistentes ou não.
    lista de trabalho: = {(x, y) | existe uma
    ↪  relação R2 (x, y) ou uma relação R2 (y,
    ↪  x)}

    Faz
        selecione qualquer arco (x, y) da lista de
        ↪  trabalho
        lista de trabalho: = lista de trabalho -
        ↪  (x, y)
        se arco-reduzir (x, y)
             se D (x) estiver vazio,
```

```
                retornar a falha
            else
               lista de trabalho: = lista de
               ↪  trabalho + {(z, x) | z! = y e
               ↪  existe uma relação R2 (x, z)
               ↪  ou uma relação R2 (z, x)}
    enquanto a lista de trabalho não está vazia

função arco-reduzir (x, y)
     mudança bool = falso
    para cada vx em D (x)
        encontre um valor vy em D (y) de modo que
        ↪  vx e vy satisfaçam a restrição R2 (x,
        ↪  y)
        se não houver tal vy {
            D (x): = D (x) - vx
            mudança: = verdadeiro
        }
    mudança de
retorno
```

C. *Implementação do algoritmo*

O algorítimo foi desenvolvido em Python, com os inputs dos dados iniciais através de tabelas em TXT, do nível fácil ao nível difícil.

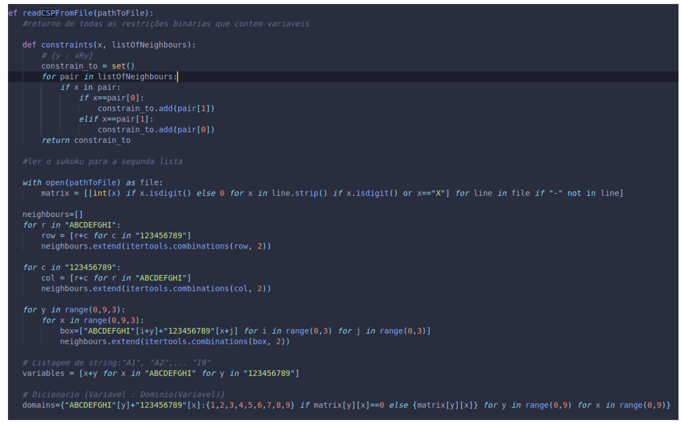

```
8 X X | X X X | X X X
X X 3 | 6 X X | X X X
X 7 X | X 9 X | 2 X X
- - - + - - - + - - -
X 5 X | X X 7 | X X X
X X X | X 4 5 | 7 X X
X X X | 1 X X | X 3 X
- - - + - - - + - - -
X X 1 | X X X | X 6 8
X X 8 | 5 X X | X 1 X
X 9 X | X X X | 4 X X
```

Fig. 2. Exemplo difícil- Tabela inicial sudoku

Após o arquivo de sudoku ser passado como parâmetro para a classe main, qual é passado para o método de leitura do arquivo, onde são definidas suas variáveis, domínios e restrições, validação e varredura do arquivo de input, definindo onde são as linhas e onde são possíveis jogadas a serem feitas, sua definição de vizinhos por colunas, linhas e grades, quais passam valores para a classe csp(variáveis, domínios e restrições.

Na classe padrão, se encontra a formatação do de saída, instanciação de restrições, domínios e verificação de jogo solucionado.

Com a chamada do método AC3, a fila é iniciada como vazia, adicionando todas as restrições presentes em y de cada variável X. Enquanto há itens na fila, x,y recebem uma remoção da fila, b,r recebem valores da redução da consistência destes parâmetros X,y, qual retorna todas as remoções dos domínios de x e y, bem como se seu status foi alterado.


```python
ef readCSPFromFile(pathToFile):
    #returno de todas as restrições binárias que contem variaveis

    def constraints(x, listOfNeighbours):
        # {y : xRy}
        constrain_to = set()
        for pair in listOfNeighbours:
            if x in pair:
                if x==pair[0]:
                    constrain_to.add(pair[1])
                elif x==pair[1]:
                    constrain_to.add(pair[0])
        return constrain_to

    #ler o sukoku para a segunda lista

    with open(pathToFile) as file:
        matrix = [[int(x) if x.isdigit() else 0 for x in line.strip() if x.isdigit() or x=="X"] for line in file if "-" not in line]

    neighbours=[]
    for r in "ABCDEFGHI":
        row = [r+c for c in "123456789"]
        neighbours.extend(itertools.combinations(row, 2))

    for c in "123456789":
        col = [r+c for r in "ABCDEFGHI"]
        neighbours.extend(itertools.combinations(col, 2))

    for y in range(0,9,3):
        for x in range(0,9,3):
            box=["ABCDEFGHI"[i+y]+"123456789"[x+j] for i in range(0,3) for j in range(0,3)]
            neighbours.extend(itertools.combinations(box, 2))

    # Listagem de string:"A1", "A2",... "I9"
    variables = [x+y for x in "ABCDEFGHI" for y in "123456789"]

    # Dicionario {Variavel : Dominio(Variavel)}
    domains={"ABCDEFGHI"[y]+"123456789"[x]:{1,2,3,4,5,6,7,8,9} if matrix[y][x]==0 else {matrix[y][x]} for y in range(0,9) for x in range(0,9)}
```

Fig. 3. Código - Definição de domínios, variáveis e restrições


```python
row = "ABCDEFGHI"
col = "123456789"

class csp:

    def __init__(self, variables, domains, constraints):
        self.variables = variables
        self.domains = domains
        self.constraints = constraints

    def solved(self):
        return not any(len(self.domains[var])!=1 for var in self.variables)

    def __str__(self):
        output=""
        for i in range(0,9):
            if(i%3==0 and i!=0):
                output+=("- - - + - - - + - - - \n")
            for j in range(0,9):
                var="ABCDEFGHI"[i]+"123456789"[j]
                if(j%3==0 and j!=0):
                    output+="| "
                if len(self.domains[var])==1:
                    value=self.domains[var].pop()
                    output+=str(value) + " "
                    self.domains[var].add(value)
                else:
                    output+='X '
            output+="\n"
        return(output)
```

Fig. 4. Código - Classe CSP

Se r for verdadeiro, ele é iterado no array das remoções, logo se b for verdadeiro, arc não é consistente, checando novamente se o domínio é equivalente a zero, caso contrário feito uma nova varredura na vizinhança para remover os valores(adicionado na fila).

Caso o jogo não consiga ser resolvido apenas com o método AC3, é passado como parâmetro o Sudoku parcialmente resolvido para o método de backtracking para que seja continuado em busca da solução.

O método de backtrack é iniciado com uma validação de o CSP foi resolvido, logo, valida quais são as variáveis não definidas(será 0 para a primeira iteração),e valores de domínio pedidos(também será 0 para a primeira iteração).

Primeiramente as variáveis e removidos são associadas, iterando sobre novos removidos. Se AC3 for consistente, continua a busca, e se o resultado encontrado for verdadeiro o resultado final é retornado. Logo, se AC3 não for consistente, os valores removidos pela inferencia são restaurados. Por fim, caso não consiga encontrar nenhuma solução disponível,

```python
def AC3 (csp, queue=None):

    def arc_reduce(x,y):
        removals=[]
        change=False
        for vx in csp.domains[x].copy():
            found=False

            for vy in csp.domains[y]:
                if diff(vx,vy):
                    found=True
            if(not found):
                csp.domains[x].remove(vx)
                removals.append((x,vx))
                change=True

        return change,removals
    removals=[]

    if queue is None:
        queue=[]
        for x in csp.variables:
            queue = queue + [(x, y) for y in csp.constraints[x]]

    while queue:
        x,y= queue.pop()

        b,r=arc_reduce(x,y)

        if r:
            removals.extend(r)
        if(b):
            #nao é arc consistente
            if(len(csp.domains[x])==0):
                return False, removals
            #Checar vizinhança ao remover valores
            else:
                queue = queue + [(x, z) for z in csp.constraints[x] if z!=y]

    return True, removals
```

Fig. 5. Código- Método AC3

```python
if(Sudoku.solved()):
    print("Solução Sudoku com apenas o método AC-3: ")
    print(Sudoku)
else:
    print("Sudoku parcialmente resolvido por AC-3")
    print(Sudoku)

    print("Tentando pelo método de backtracking...")
    solution = backTrackingSearch(Sudoku)

    if(solution):
        print("Solução encontrada por backtracking: \n")
        print(solution)
    else:
        print("A busca backtrack  não conseguiu encontrar a solução.")
```

Fig. 6. Código- Chamada de métodos AC3 e Backtracking

```python
def backTrackingSearch(csp): #retorna solução ou falha
    return backtrack({},csp)


def backtrack(assignment, csp): #retorna solução ou falha
    if csp.solved():
        return csp

    var = selectUnassignedVariable(csp, assignment)

    for value in OrderDomainValues(csp, assignment, var):

        assignment[var] = value

        removals = [(var, a) for a in csp.domains[var] if a != value]

        #Assumir que Var = Value => D(Var) = Value
        csp.domains[var] = {value}

        consistent, removed = AC3(csp, [(x,var) for x in csp.constraints[var]])
        #Se valores forem removidos pelo AC3, adivionar a lista para ser restaurados
        if removed:
            removals.extend(removed)

        #Se AC3 consistente
        if(consistent):
            #continue a busca
            result = backtrack(assignment,csp)
            #Se não houver falha, retornar a solução.
            if(result!=False):
                return result

        #Se o CSP nao for AC3 consistente, restaures os valores removidos pela inferencia
        for variable, value in removals:
            csp.domains[variable].add(value)

    #Incapaz de  encontrar solução disponível para  este caminho, retornar um passo e tentar caminho diferente
    del assignment[var]
    return False
```

Fig. 7. Código- Método Backtracking

```
-----------------------------
Tentativa com AC-3

Sudoku parcialmente resolvido por AC-3
8 X X | X X X | X X X
X X 3 | 6 X X | X X X
X 7 X | X 9 X | 2 X X
- - - + - - - + - - -
X 5 X | X X 7 | X X X
X X X | X 4 5 | 7 X X
X X X | 1 X X | X 3 X
- - - + - - - + - - -
X X 1 | X X X | X 6 8
X X 8 | 5 X X | X 1 X
X 9 X | X X X | 4 X X

Tentando pelo método de backtracking...
Tempo gasto 0.56s
Solução encontrada por backtracking:

8 1 2 | 7 5 3 | 6 4 9
9 4 3 | 6 8 2 | 1 7 5
6 7 5 | 4 9 1 | 2 8 3
- - - + - - - + - - -
1 5 4 | 2 3 7 | 8 9 6
3 6 9 | 8 4 5 | 7 2 1
2 8 7 | 1 6 9 | 5 3 4
- - - + - - - + - - -
5 2 1 | 9 7 4 | 3 6 8
4 3 8 | 5 2 6 | 9 1 7
7 9 6 | 3 1 8 | 4 5 2
```

Fig. 8. Resultado para sudoku difícil - AC3 e Backtracking

o mesmo retorna um passo e tenta um caminho de busca diferente para a solução.

IV. Resultados

Algumas das soluções via AC-3 acabam resultando em um sudoku incompleto, pois esta metodologia não consegue satisfazer algumas restrições, tendo em vista que a mesma não testa as possibilidades de variáveis como como acontece com o backtracking.

Com o auxilio do backtracking o mesmo é resolvido com sucesso, pois os valores retornados pelo AC-3 preencheram um significativo número de linhas, colunas e grades, reduzindo assim a quantidade de restrições a serem testadas pelo algoritmo.

Conclusão

Portanto o AC3 serve para que valores consistentes sejam inseridos na solução do CSP do Sudoku, reduzindo o tamanho do domínio das variáveis e garantindo todas as atribuições futuras sejam consistentes/possíveis, logo o backtracking parte da força bruta deste valor resultado para buscar a solução final;

Com isso, a aplicação do backtracking pode ser prática de ser implementada em problemas, comparados a outras formas, sendo que estes algoritmos tendam a resolver praticamente qualquer problema. Caso não aplicado juntamente à restrições, ele executa uma busca exaustiva, por força bruta, buscando soluções possíveis, podendo ocasionar em explosão combinatória. [8]

Desse modo algoritmos de retrocesso,requerem um grande espaço de memória disponível na máquina, pois a quantidade de variáveis locais passadas a cada chamada recursiva é proporcional ao tamanho do problema, podendo aumentar exponencialmente o consumo de recursos e tempo de execução.

Resumidamente, o uso de CSP é muito comum e vale a pena encontrar maneiras eficientes de resolve-los. Não existem algoritmos conhecidos para resolver CSP com domínios NP-Difíceis, pois o tempo de solução acaba sendo exponencial e inviável.

REFERENCES

[1] Arangu, Marlene Salido, Miguel & Barber, Federico. (2009). AC3-OP: An Arc-Consistency Algorithm for Arithmetic Constraints.. Frontiers in Artificial Intelligence and Applications. 202. 293-300. 10.3233/978-1-60750-061-2-293.

[2] Machado, Marlos Chaimowicz, Luiz. (2011). Combining Meta-heuristics and CSP Algorithms to Solve Sudoku. Brazilian Symposium on Games and Digital Entertainment, SBGAMES. 124-131. 10.1109/SBGAMES.2011.18.

[3] Pt.wikipedia.org. 2020. Sudoku. [online] Disponível em: https://pt.wikipedia.org/wiki/Sudoku [Acesso 13 Ago 2020].

[4] Pt.wikipedia.org. (2020). Problema da satisfação de restrições. [online] Disponível em: [Acesso 13 Ago 2020].

[5] Pt.wikipedia.org. (2020). Backtracking. [online] Disponível em: [Acesso 14 Ago 2020].

[6] Pt.wikipedia.org. (2020). AC-3 Algorithm. [online] Disponível em: [Acesso 14 Ago 2020].

[7] Prehn, Donavan.2017.Sudoku. GitHub [online] Disponível em: [Acesso 14 Ago 2020].

[8] Toussaint, Marc. (2016). Artificial Intelligence - Constraint satisfaction Problems. [online] Disponível em: [Acesso em 13 Ago 2020].