Universidade Federal do Amazonas

Instituto de Computação

3º TRABALHO DE INTELIGÊNCIA ARTIFICIAL (LATTICE BOLTZMANN METHOD - MODEL COUNTING)

Manaus - AM
2023

ALUNOS: DARLYSSON MELO DE LIMA - 21954316
EVANDRO SALVADOR MARINHO DA SILVA - 22052988
KRISTHIAN ALBUQUERQUE DA SILVA - 21950517

3º TRABALHO DE INTELIGÊNCIA ARTIFICIAL (LATTICE BOLTZMANN METHOD - MODEL COUNTING)

Trabalho apresentado à Universidade Federal do Amazonas, como requisito para obtenção de 3º nota de inteligência artificial adiminitrada pelo Professor Edjard Mota, do curso de Engenharia da computação.

Manaus - AM
2023

Conteúdo

1 Introdução

O método Lattice Boltzmann (LBM) é uma abordagem inovadora e poderosa para a simulação de fenômenos fluidodinâmicos complexos. Desenvolvido inicialmente como uma técnica numérica para resolver equações de transporte em meios porosos, o LBM evoluiu para se tornar uma ferramenta versátil para modelagem de fluxos em uma variedade de contextos, desde escoamentos clássicos até problemas multifásicos e interações fluido-estrutura.

A singularidade do LBM reside na sua formulação baseada em uma grade discreta e na abordagem estocástica para representar a distribuição de partículas. Ao discretizar o espaço de velocidade em uma rede regular, o LBM simplifica a resolução de equações de transporte, permitindo uma abordagem mais eficiente para a simulação de fenômenos físicos. Essa característica torna o LBM particularmente adequado para problemas complexos, nos quais a discretização tradicional se torna onerosa computacionalmente.

Além disso, o LBM destaca-se por sua natureza paralelizável, tornando-o uma escolha atrativa para a computação de alto desempenho (HPC). Isso possibilita a simulação eficiente de fenômenos fluidodinâmicos em escalas temporais e espaciais significativas, ampliando as capacidades de análise e otimização de sistemas complexos.

No contexto deste trabalho, exploraremos a aplicação do LBM na resolução de problemas de contagem de modelos (Model Counting) em lógica booleana. A capacidade do LBM em lidar com problemas fluidodinâmicos complexos será aproveitada para abordar desafios computacionais associados ao Model Counting, proporcionando uma nova perspectiva e eficiência na análise de fórmulas booleanas complexas. Este estudo visa demonstrar a adaptação e a aplicação inovadora do LBM em um domínio além do seu escopo tradicional, destacando seu potencial para impulsionar avanços na resolução de problemas computacionais desafiadores.

2 Forma Normal Conjuntiva (CNF)

O model counting é uma tarefa computacional que envolve contar o número de modelos (atribuições de verdadeiro ou falso) que satisfazem uma dada fórmula booleana. O problema é conhecido por ser computacionalmente desafiador, especialmente para fórmulas complexas.

Em geral, um modelo para model counting pode ser representado como uma fórmula booleana na forma normal conjuntiva (CNF), que é uma conjunção (AND) de cláusulas, onde cada cláusula é uma disjunção (OR) de literais. Os literais são variáveis booleanas ou suas negações.

A forma normal conjuntiva (CNF) é uma maneira de representar fórmulas lógicas como uma conjunção de cláusulas, onde cada cláusula é uma disjunção de literais (variáveis booleanas ou suas negações). Para SAT (Satisfabilidade Booleana), você quer criar uma CNF que seja satisfazível se e somente se a fórmula booleana original for verdadeira.

Aqui está um exemplo simples:

Suponha que você tenha a seguinte fórmula booleana:

$$(A \text{ ou } B) \text{ e } (C \text{ ou } D)$$

Esta fórmula é uma conjunção de duas cláusulas, cada uma contendo literais. Você pode representar isso em formato CNF da seguinte maneira:

```
1 p cnf 4 2
  1 2 0
  3 4 0
```

Listing 1: *Arquivo DIMACS*

Onde:

A primeira linha especifica que é um problema CNF com 4 variáveis (numeradas de 1 a 4) e 2 cláusulas.

Cada linha subsequente representa uma cláusula, onde os números positivos representam variáveis e os números negativos representam a negação dessas variáveis. O zero no final de cada linha indica o fim da cláusula.

Este é apenas um exemplo simples. Fórmulas CNF podem ser muito mais complexas, dependendo do problema lógico que você está tentando representar.

3 Implementação

Segue a mundaça no codigo run.py com objetivo de salvar diferente resultados de SAT. Para assim computalo atraves de script:

```
import sys, glob, os
import numpy as np
from pathlib import Path
import argparse

from cnf import DIMACSReader
from cnf2rbm import RBMSAT
from gibbs import *
from femin import *
from libs.pysat.pysat_wrapper import PySatWrapper

import time
from distutils.util import strtobool

parser = argparse.ArgumentParser()
parser.add_argument('-dnf','--dnf', action='store', type=str, help='path
    to dnf file',required=True)

parser.add_argument('-search','--search',action='store',type=str,help='
    search type',default="pysat")

parser.add_argument('-maxiter','--maxiter',action='store',type=int,help=
    'max iter')

parser.add_argument('-optimizer','--optimizer',action='store',type=str,
    help='optimizer, only use with optimization approachs')

parser.add_argument('-learning_rate','--learning_rate',action='store',
    type=float,help='learning rate for the optimiser')

parser.add_argument('-batch_size','--batch_size',action='store',type=int
    ,help='number of seed (initial) assignments')

parser.add_argument('-binarisation','--binarisation',action='store',type
    =str,help='binarisation method')

parser.add_argument('-cvalue','--cvalue',action='store',type=float,help=
    'confidence value')

parser.add_argument('-oplib','--oplib',action='store',type=str,help='
    optimization library')

parser.add_argument('-sigmoid_scale','--sigmoid_scale',action='store',
    type=float,help='sigmoid scale')
```

```
# params for dual annealing
parser.add_argument('-initial_temp','--initial_temp',action='store',type
    =float,help='imitial temperature',nargs="?",const=5230.0,default
    =5230.0)
parser.add_argument('-no_local_search','--no_local_search',action='store
    ',type=lambda x:bool(strtobool(x)),help='no local search',nargs="?",
    const=True,default=True)
parser.add_argument('-restart_temp_ratio','--restart_temp_ratio',action=
    'store',type=float,help='restart temperature ratio',nargs="?",const=2
    e-05,default=2e-05)
parser.add_argument('-visit','--visit',action='store',type=float,help='
    visit',nargs="?",const=2.62,default=2.62)
parser.add_argument('-accept','--accept',action='store',type=float,help=
    'accept',nargs="?",const=-5.0,default=-5.0)
parser.add_argument('-maxfun','--maxfun',action='store',type=float,help=
    'maxfun',nargs="?",const=10000000.0,default=10000000.0)

args = parser.parse_args()

opt_params = {"initial_temp":args.initial_temp,
              "no_local_search":args.no_local_search,
              "restart_temp_ratio":args.restart_temp_ratio,
              "visit":args.visit,
              "accept":args.accept,
              "maxfun":args.maxfun
              }

if __name__=="__main__":
    fname = os.path.basename(args.dnf)
    search = args.search
    optimizer = args.optimizer
    problem_dir_name = os.path.dirname(args.dnf)

    stime = time.time()

    if search=="pysat":
        log_dir = problem_dir_name.replace("data","code/sat-lbm/results"
            )
        log_dir += "/"+fname+"_log"
        log_dir += "/"+search

        if not os.path.exists(log_dir):
            print("Creating logging directory %s"%(log_dir))
            os.makedirs(log_dir)

        stime = time.time()
        reasoner = PySatWrapper(args.dnf)
        #print(reasoner.formula.nv)
        issat,model = reasoner.solve()
        np.savetxt(os.path.join(log_dir,"log.csv"),[issat,time.time()-
            stime])
        if model is None:
            model = [0]
```

```
        np.savetxt(os.path.join(log_dir,"solution.csv"),model)

        #print(model)
        #print(time.time()-stime)
    else:
        dnf_r = DIMACSReader(args.dnf)
        rbmsat = RBMSAT(dnf_r,conf_val=args.cvalue)
        for a in range(1000):
            if search=="emin":
                reasoner = Gibbs(rbmsat, log_dir)
            elif search=="femin":
                log_dir = "MC/solution"+str(a)

                reasoner = FEMin(rbmsat,
                                 log_dir,
                                 batch_size=args.batch_size,
                                 optimizer=args.optimizer,
                                 lr=args.learning_rate,
                                 binarisation=args.binarisation,
                                 oplib=args.oplib,
                                 sigmoid_scale=args.sigmoid_scale,
                                 maxiter=args.maxiter,
                                 opt_params=opt_params)
            else:
                raise ValueError("Error")
            is_sat = reasoner.run()
```

Código 2: *Alteração no arquivo Run.py*

No codigo acima foi realizado a incrementação do laço For na linha 85 para coletar uma quantidade 1000 amostras de resultados e assim utilizar do script abaixo para computadar e contar as possiveis soluções para o problema de SAT demostrado em (Código 1). A contagem é realizada atraves de um mapeamento dos resultados obtidos, onde valores positivos são mapeados para o valor 1 enquanto valores negativos são mapeados para 0.

Assim obtemos os resultado a seguir que satisfazem a tabela verdade da formula booleana (A ou B) e (C ou D).

Com isso revela as seguintes soluções que satisfazem True, onde contabilizam 9 possiveis associações as variaveis que satisafazem a formula:

```
$ python script2.py
1 1 1 0
1 1 0 1
1 0 1 1
0 1 1 0
1 0 1 0
1 1 1 1
0 1 1 1
0 1 0 1
1 0 0 1
```

Código 3: *Resultado obtido*

Onde o resultado pode ser confirmado na tabela verdade a seguir:

A	B	C	D	(A ∨ B) ∧ (C ∨ D)
1	1	1	1	1
1	1	1	0	1
1	1	0	1	1
1	0	1	1	1
0	1	1	1	1
0	0	0	1	0
0	0	1	0	0
0	1	0	0	0
1	0	0	0	0
0	0	0	0	0
1	1	0	0	0
0	0	1	1	0
1	0	0	1	1
0	1	1	0	1
1	0	1	0	1
0	1	0	1	1

Figura 1: *Tabela verdade*

```
import csv
import os
import subprocess

# Comando que voc  deseja executar
comando = [
    'python',
    'copyrun.py',
    '--dnf',
    './MC/model_sat=0.dimacs',
    '--search',
    'femin',
    '--optimizer=dual_annealing',
    '--binarisation=soft_sigmoid',
    '--oplib=scipy',
    '--cvalue=5',
    '--sigmoid_scale=1',
    '--maxiter=50000',
    '--initial_temp=10000',
    '--no_local_search=True'
]

# Executar o comando
subprocess.run(comando)

```

```python
# Diret rio contendo as subpastas enumeradas
diretorio_raiz = './MC/'
# Nome da subpasta contendo os arquivos CSV de sa da
nome_subpasta_solution = 'solution'
# Nome do arquivo de texto de sa da
nome_arquivo_txt = 'saida.txt'
# Lista para armazenar os vetores
vetores = []

# Loop pelas subpastas numeradas
for nome_subpasta in os.listdir(diretorio_raiz):
    caminho_subpasta = os.path.join(diretorio_raiz, nome_subpasta)

    if os.path.isdir(caminho_subpasta) and nome_subpasta.startswith(
        nome_subpasta_solution):
        caminho_arquivo_csv = os.path.join(caminho_subpasta, 'solution.
            csv')

        # Leitura do arquivo CSV
        with open(caminho_arquivo_csv, 'r') as arquivo_csv:
            leitor_csv = csv.reader(arquivo_csv)

            for linha in leitor_csv:
                # Converte os valores para 1 ou 0
                vetor = [1 if float(valor) > 0 else 0 for valor in linha
                    ]
                vetores.append(vetor)

# Escreve os vetores no arquivo de texto
with open(nome_arquivo_txt, 'w') as arquivo_txt:
    for i, vetor in enumerate(vetores, start=1):
        # Converte os valores para strings e os une em uma linha
        linha = ' '.join(map(str, vetor))
        arquivo_txt.write(linha)

        # Adiciona uma quebra de linha a cada 4 valores ou no final do
            vetor
        if i % 4 == 0 or i == len(vetores):
            arquivo_txt.write('\n')
        else:
            arquivo_txt.write(' ')

# L   as linhas do arquivo
with open(nome_arquivo_txt, 'r') as arquivo_txt:
    linhas = arquivo_txt.readlines()

# Identifica e imprime as linhas   nicas
linhas_unicas = set(linhas)
for linha in linhas_unicas:
    print(linha.strip())
```

Código 4: *Código Python para realizar a leitura das saídas do modelo*

4 Conclusão

Em conclusão, o trabalho abordou a problemática do model counting em lógica booleana, enfocando a representação das fórmulas na Forma Normal Conjuntiva (CNF). A complexidade computacional associada ao model counting foi reconhecida, especialmente para fórmulas booleanas intricadas.

A implementação apresentada no código Run.py introduziu modificações visando salvar resultados diferentes de SAT para posterior análise. A extensão do script incluiu a coleta de 1000 amostras de resultados usando abordagens como PySat, Gibbs e FEMin. A análise dos resultados foi realizada por meio de um script que mapeou os valores obtidos, identificando associações de variáveis que satisfaziam a fórmula booleana.

O resultado final demonstrou no Código 3 nove possíveis soluções que satisfazem a tabela verdade da fórmula booleana fornecida. A abordagem de contagem de modelos ofereceu uma visão mais detalhada das diferentes combinações de variáveis que levam à satisfação da fórmula.

Além disso, o código Python apresentado na seção final oferece uma maneira eficiente de analisar e consolidar os resultados em um arquivo de saída, facilitando a interpretação e visualização das soluções encontradas.

Portanto, o trabalho proporcionou uma compreensão mais aprofundada da abordagem Lattice Boltzmann Method - Model Counting para resolver problemas de model counting em lógica booleana, contribuindo para a análise e otimização de fórmulas complexas.