# Relatório Exercício-Programa 1

MAC0239 – Introdução à Lógica e Verificação de Programas

Gabriel Baptista - 8941300
Helio Hideki Assakura Moreira - 8941064

## 1 Introdução

O programa consiste na resolução do problema das n-rainhas, o qual deve-se ler dois números inteiros, n que representa as dimensões de um tabuleiro de xadrez e k que representa o número de rainhas já colocadas no mesmo. Para ambos os casos, a primeira linha de saída do programa deve ser "SAT", se existe solução para tal tabuleiro ou "UNSAT", se não existir.
Para desenvolvimento do mesmo, foi proposto que programássemos utilizando a linguagem de programação Python, a qual possui a biblioteca Pyeda, muito útil para manusear BDDs, Binary Decision Diagrams e para utilização de símbolos lógicos, que são alguns de nossos objetos de estudos nesta matéria.

## 2 A lógica do programa

Para resolver tal problema, considera-se as regras derivadas do xadrez, uma rainha pode executar um ataque na linha em que se encontra, na coluna ou diagonais superiores e inferiores a ela, então, utilizando as operações lógicas,desenvolvemos o programa dividindo nas seguintes funções:

- *expression lines (x)*, que recebe as variavéis de x, as analisa e então retorna a expressão correspondente ao não ataque das rainhas nas *linhas* e a existência de uma rainha na *linha.*
- *expression columns (x)*, segue o mesmo princípio do expression lines, porém considerando as colunas.
- *expression diagonal (x, n, full expression)*,que recebe as exprvars de x, o tamanho do tabuleiro e a expressão parcialmente montada, e então coloca na expressão todas as combinações possíveis duas a duas de rainhas que podem se atacar nas diagonais.

Com estas funções, consegue-se formar expressões e assim, verifica se existe ou não solução, caso exista, com a utilização da função *satisfy one*, tem-se uma e apenas uma solução para o tabuleiro de tamanho n x n.

# 3 Testes

Os testes foram feitos em um processador Intel i7 2600K CPU 3.40GHz e 7,8 GiB de Memória. Para concluir alguma coisa além da solução do problema, considera-se duas maneiras de solucioná-lo, um que cria o BDD e outra que apenas responde se há ou não solução, caso positivo a coloca na tela:

| ***n*** | ***SEM*** | ***COM*** | ***SAT?*** |
|---|---|---|---|
| 1 | $0m0.043s$ | $0m0.049s$ | $SAT$ |
| 2 | $0m0.041s$ | $0m0.043s$ | $UNSAT$ |
| 3 | $0m0.054s$ | $0m0.053s$ | $UNSAT$ |
| 4 | $0m0.053s$ | $0m0.062s$ | $SAT$ |
| 6 | $0m0.165s$ | $0m0.507s$ | $SAT$ |
| 8 | $0m1.420s$ | $0m16.054s$ | $SAT$ |
| 9 | $0m2.426s$ | $1m37.873s$ | $SAT$ |
| 10 | $0m2.694s$ | $>$ | $SAT$ |
| 13 | $0m9.672s$ | $>$ | $SAT$ |
| 15 | $2m46.160s$ | $>$ | $SAT$ |
| 17 | $17m0.32.411s$ | $>$ | $SAT$ |

Tabela 1: Tempos com BDD e sem BDD, os campos preenchidos com o sinal de maior, não foram observados, pois levou um tempo muito grande.

# 4 Considerações Finais

Com este EP, conclui-se que para n's relativamente grandes(maiores que 10) a criação do BDD é algo muito custoso computacionalmente,pois o número de variáveis cresce de forma quadrática,talvez em um computador mais potente obteria-se resultados mais expressivos para estes n's. Além disso, a utilização do restrict para valorações 0 ou 1 auxilia na criação do BDD reduzido referente a expressão que é a resposta do problema encontrada. Porém,utilizando o método *satisfy count()*, que mostra a quantidade de soluções, temos os seguintes resultados:

- Direto da expressão:

  n = 8: 0m16.540s, 92 solucoes n = 9: 1m41.069s, 352 solucoes

- Construindo o BDD:

  n = 8: 0m15.701s, 92 solucoes n = 9: 1m39.333s, 352 solucoes

Observa-se uma semelhança entre os tempos, porém, quando resolve-se direto da expressão, deve-se usar SAT-solvers, enquanto o *satisfy count()*, monta o BDD e o resolve, o que o torna ligeiramente melhor para mostrar a quantidade de soluções.

## 5 Imagens

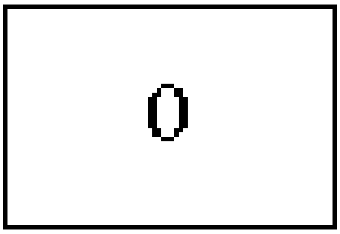


Figura 1: BDD para n=3, UNSAT

Figura 2: BDD para n=6,sem restrict

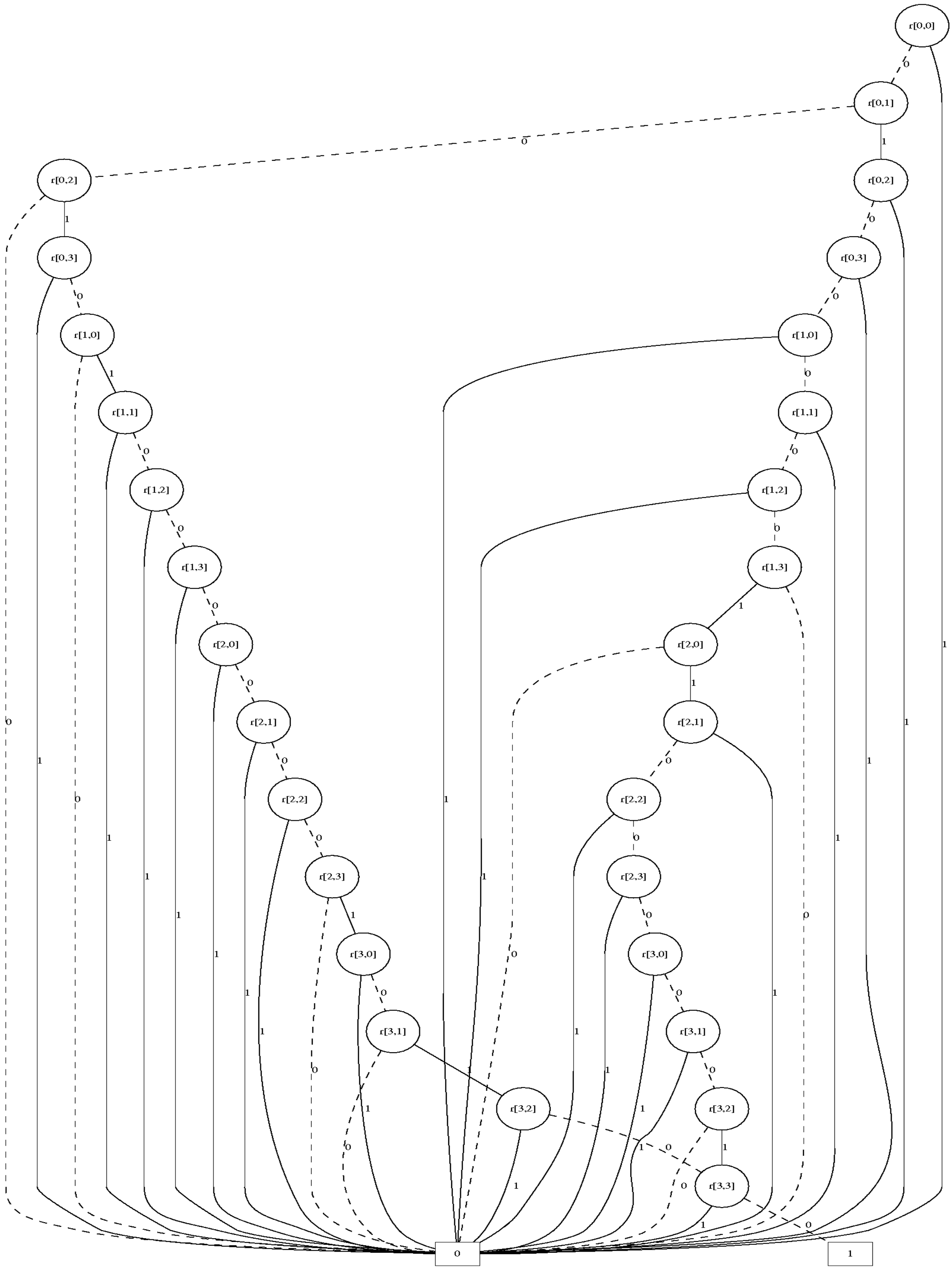


Figura 3: BDD para n=4,sem restrict

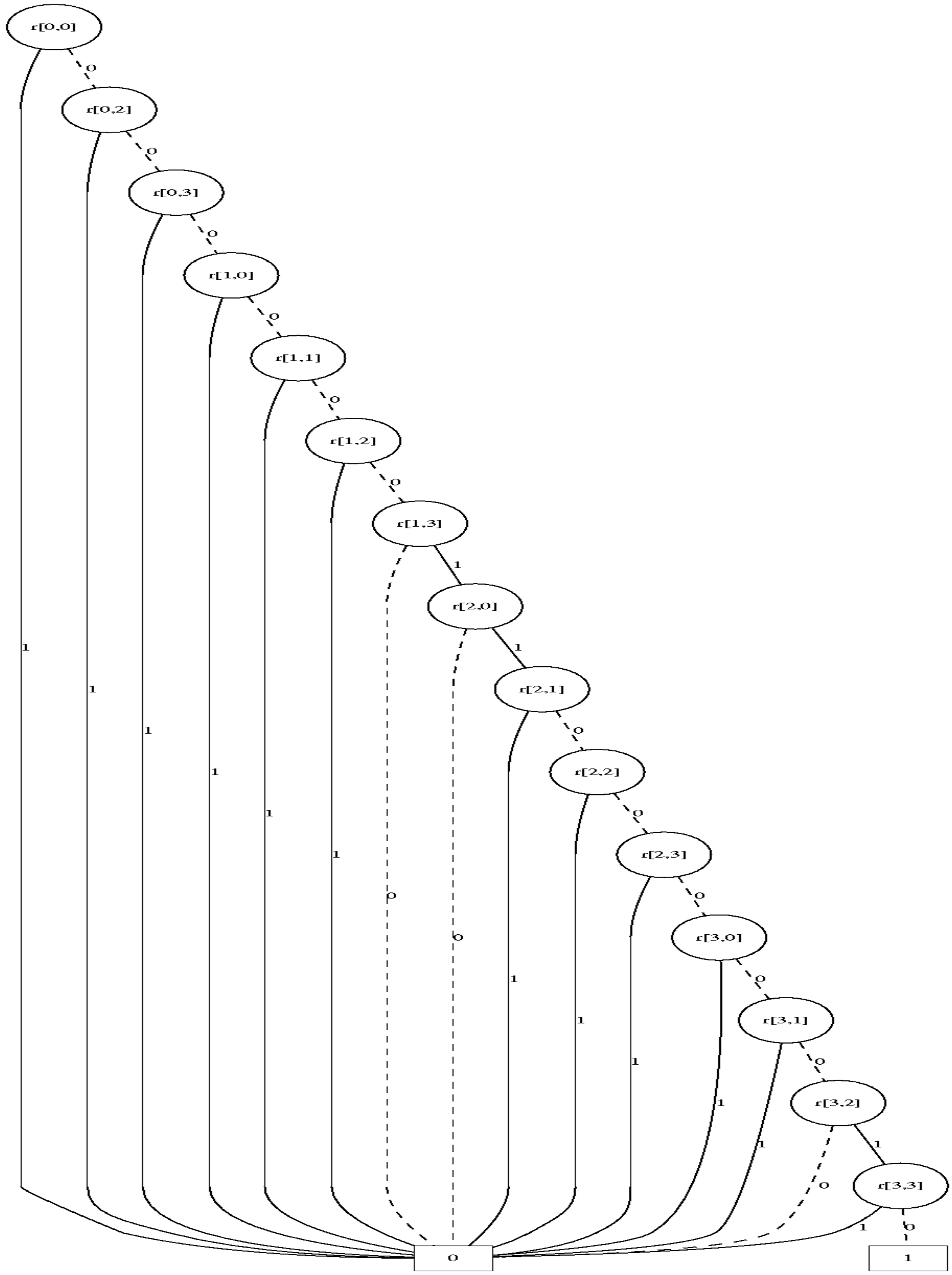


Figura 4: BDD para n=4,com restrict,diminuindo o tamanho do BDD