UNIVERSIDADE DE SÃO PAULO
FACULDADE DE CIÊNCIAS FARMACÊUTICAS
Curso de Graduação em Farmácia-Bioquímica

Inferência de Redes Booleanas para o ciclo intra-eritrocítico do parasita da Malária

Raul Granja Adoglio

Trabalho de Conclusão do Curso de Farmácia-Bioquímica da Faculdade de Ciências Farmacêuticas da Universidade de São Paulo.

Orientador: Prof. Dr. Ronaldo Fumio Hashimoto

São Paulo
2024

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

IDC Ciclo de Desenvolvimento Intra-eritrocítico

OMS Organização Mundial da Saúde

STG State Transition Graph (Grafo de Transição de Estados)

TBN Rede Booleana com limiar (Thresholded Boolean Network)

TP Ponto Temporal (Temporal Point)

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

ADOGLIO, R. G. **Inferência de Redes Booleanas para o ciclo intra-eritrocítico do parasita da Malária.** 2024. 58 p. Trabalho de Conclusão de Curso de Farmácia-Bioquímica – Faculdade de Ciências Farmacêuticas – Universidade de São Paulo, São Paulo, 2024.

Palavras-chave: Malária, Redes, Booleanas, Bioinformática.

Introdução: A malária, causada por parasitas do gênero *Plasmodium*, especialmente *Plasmodium falciparum*, é uma das principais causas de mortalidade em regiões tropicais. A resistência crescente do parasita aos tratamentos existentes demanda novas estratégias terapêuticas. Neste contexto, as Redes Booleanas se apresentam como uma ferramenta promissora para modelar sistemas biológicos complexos, permitindo simular interações entre genes e proteínas essenciais no ciclo de desenvolvimento intra-eritrocítico (IDC) do parasita. A inferência de Redes Booleanas através dessa abordagem computacional pode contribuir para o desenvolvimento de tratamentos mais eficazes contra a malária.

Objetivos: Este trabalho tem como objetivo geral inferir redes booleanas no ciclo de vida do *P. falciparum*, parasita causador da malária. A abordagem computacional visa modelar as interações biológicas do parasita e identificar possíveis alvos terapêuticos. Entre os objetivos específicos, estão a modelagem e simulação de Redes Booleanas baseadas em dados biológicos, focando nas interações genéticas durante o IDC. A análise dessas redes permitirá avaliar a viabilidade de intervenções em pontos críticos para interromper o ciclo do parasita.

Metodologia: A metodologia deste trabalho envolve a binarização dos dados de expressão gênica do *P. falciparum*. Os dados, organizados em 46 pontos temporais, foram transformados em valores binários (0 ou 1) para indicar variações significativas de expressão. Após filtrar genes com dados incompletos, foi aplicada uma binarização baseada na média dos grupos funcionais, permitindo identificar padrões de regulação gênica. Esses dados binarizados foram utilizados para inferir Redes Booleanas, visando identificar alvos terapêuticos no IDC do parasita.

Resultados: Foi possível construir um Grafo de Transição de Estados (STG) para uma Rede Booleana com 12 grupos funcionais, utilizando matrizes regulatórias

aleatórias. A matriz com o menor número de componentes conexos foi selecionada para criar o STG, revelando 14 bacias de atração em uma primeira tentativa com 20 matrizes. Após gerar 100.000 matrizes, foi obtida uma rede otimizada com 4 bacias de atração. Gráficos de barras e dispersão foram utilizados para visualizar a frequência e a relação entre o número de bacias de atração e a quantidade de componentes conexos.

Conclusão: Este trabalho inferiu uma Rede Booleana para identificar alvos terapêuticos no IDC do *P. falciparum*. Resultados destacaram os transcritos do estágio anelar inicial, sugerindo sua relevância terapêutica. No entanto, devido à complexidade biológica, novos estudos são necessários, com maior capacidade computacional e métodos de otimização para gerar matrizes que levem à configuração ideal com uma única bacia de atração. Além disso, investigações adicionais são essenciais para determinar em qual grupo funcional o Apicoplasto tem maior representatividade. Este estudo ressalta a importância da modelagem computacional para futuras estratégias terapêuticas.

1. INTRODUÇÃO

1.1. MALÁRIA

A malária é uma das principais causas de mortalidade em regiões tropicais e continua sendo um desafio global significativo. A doença é causada por parasitas do gênero *Plasmodium*, e a transmissão ocorre por meio da picada de mosquitos *Anopheles*, que são vetores da doença (Departamento de Articulação Estratégica de Vigilância em Saúde e Ambiente, 2023). A crescente resistência do parasita a tratamentos farmacológicos tradicionais é um grande desafio (NADEEM et al., 2023), fazendo necessário o desenvolvimento de novas estratégias terapêuticas para combater a doença (PANDEY et al., 2023).

As espécies *P. falciparum* e *P. vivax* são as mais frequentes entre as espécies que infectam humanos, sendo a primeira a mais letal (Departamento de Articulação Estratégica de Vigilância em Saúde e Ambiente, 2023). A infecção pelo *Plasmodium* ocorre quando o mosquito infectado injeta esporozoítos no organismo humano, que, ao atingir o fígado, passam por estágios de desenvolvimento antes de invadir as células vermelhas do sangue (FARROW et al., 2011). Esse ciclo complexo, aliado à capacidade de adaptação do parasita e à crescente resistência a medicamentos, torna

a descoberta de novos alvos terapêuticos uma prioridade na luta contra a malária (PANDEY et al., 2023).

1.2. CICLO DE DESENVOLVIMENTO INTRA-ERITROCÍTICO

O ciclo de desenvolvimento intra-eritrocítico (IDC) do *Plasmodium* é o estágio responsável pelos sintomas clínicos da malária, como febre, anemia e, em casos graves, complicações fatais (Departamento de Articulação Estratégica de Vigilância em Saúde e Ambiente, 2023). Durante esse ciclo, o parasita invade os eritrócitos (glóbulos vermelhos), onde se replica e destrói as células hospedeiras, liberando novos parasitas para infectar outros eritrócitos (FARROW et al., 2011). Esse estágio de vida do parasita é um alvo importante para intervenções terapêuticas, uma vez que é durante esse período que a maioria dos fármacos antimaláricos exercem seus efeitos (SAWYER; KHAN; LE, 2023).

1.3. REDES BOOLEANAS

Nesse contexto, a modelagem computacional tem se mostrado uma ferramenta promissora para a descoberta de novos alvos terapêuticos e o desenvolvimento de fármacos (LIN; LI; LIN, 2020). Em particular, as redes booleanas oferecem uma abordagem eficiente para simular e analisar sistemas biológicos complexos (HEMEDAN et al., 2022), permitindo a compreensão de interações entre genes e proteínas essenciais para o ciclo de vida do *Plasmodium*.

As Redes Booleanas foram idealizadas por Stuart Kauffman na década de 1960, como uma maneira de modelar a regulação genética e entender como os genes interagem para controlar processos biológicos, portanto, são modelos matemáticos que utilizam variáveis discretas, como “ligado” ou “desligado”, para representar o comportamento de genes e outras biomoléculas, facilitando a identificação de potenciais pontos de intervenção terapêutica (KAUFFMAN, 1969). Essa abordagem permite realizar simulações de redes biológicas e identificar nós críticos dessas redes, na tentativa de interromper o ciclo de vida do parasita.

O código em Python escrito para este trabalho pode ser encontrado no anexo, ou no link para o GitHub: https://github.com/raulgranja/malaria-boolean-network

JUSTIFICATIVA:

Em 2022, a Organização Mundial da Saúde (OMS) estimou cerca de 249 milhões de casos de malária, com aproximadamente 608 mil mortes principalmente em países da África subsaariana (World Health Organization, 2023). Isso mostra a urgência em desenvolver novas abordagens terapêuticas. Dessa forma, por meio de uma abordagem computacional, esse trabalho tem o objetivo de buscar uma nova solução, na tentativa de contribuir para a melhoria da qualidade de vida de populações vulneráveis.

A aplicação de Redes Booleanas oferece uma abordagem simplificada para a modelagem de sistemas biológicos complexos (HEMEDAN et al., 2022). Essa metodologia computacional é capaz de oferecer uma análise sistemática e eficiente de redes de interações genéticas. Simulando essas interações é possível identificar alvos terapêuticos que podem ser explorados no desenvolvimento de novos fármacos (LIN; LI; LIN, 2020).

Além disso, há uma justificativa pessoal para a escolha do tema: o estudante desenvolveu interesse por programação durante a graduação e percebeu a importância de aplicar a tecnologia na resolução de problemas da saúde. A malária foi escolhida por ser uma das doenças infecciosas mais relevantes e com maior impacto socioeconômico global. Essa união entre o interesse por programação e o compromisso em contribuir com a saúde pública foi determinante na escolha do tema deste trabalho.

Portanto, este trabalho reúne uma abordagem interdisciplinar, com uma clara relevância social e acadêmica, impulsionada por uma motivação pessoal de explorar a união entre a computação e a saúde.

2. OBJETIVOS

O objetivo geral deste trabalho é calcular Redes Booleanas para investigar possíveis intervenções farmacêuticas no tratamento da malária, utilizando uma abordagem computacional. Essa modelagem visa representar as interações biológicas do *P. falciparum*, o parasita responsável pela forma mais grave da malária, com foco em identificar alvos terapêuticos potenciais. Através da construção dessas redes, espera-se capturar de forma simplificada o comportamento dos genes ou proteínas envolvidos no ciclo de vida do parasita. Esse processo permitirá avaliar

como a interrupção de certos nós críticos nas redes pode impactar a viabilidade do parasita e, assim, auxiliar no estudo para desenvolver novas alternativas terapêuticas.

Entre os objetivos específicos, o trabalho busca, primeiramente, modelar Redes Booleanas que representem as interações biológicas do *P. falciparum* durante o seu IDC. Em seguida, essas redes serão construídas e simuladas com base em dados biológicos disponíveis, permitindo a análise de como as interações genéticas podem ser manipuladas para interromper o ciclo de vida do parasita.

3. MATERIAL E MÉTODOS

3.1 FONTE DE DADOS

A metodologia na qual este trabalho será focado é construir e simular Redes Booleanas de interações biológicas do *P. falciparum* durante seu IDC. Este trabalho modelará expressões de genes e proteínas essenciais para a sobrevivência do parasita usando dados biológicos disponíveis. Além disso, a simulação para encontrar “nós” críticos nessas redes será feita usando programação Python por meio de ferramentas computacionais.

Os dados para este trabalho foram derivados de Bozdech et al. (2003) e produzidos por uma cultura in vitro muito bem sincronizada em um biorreator. A cepa sensível à cloroquina - HB3 foi escolhida devido à sua extensa caracterização em experimentos anteriores. Os parasitas foram sincronizados de forma que permaneceram na mesma fase do ciclo de vida. As amostras foram coletadas por 48 horas, começando uma hora após a invasão dos glóbulos vermelhos. A hibridização competitiva de dois núcleos foi utilizada na medição da expressão gênica; os dois núcleos comparam a abundância de mRNA em cada estágio do IDC (BOZDECH et al., 2003).

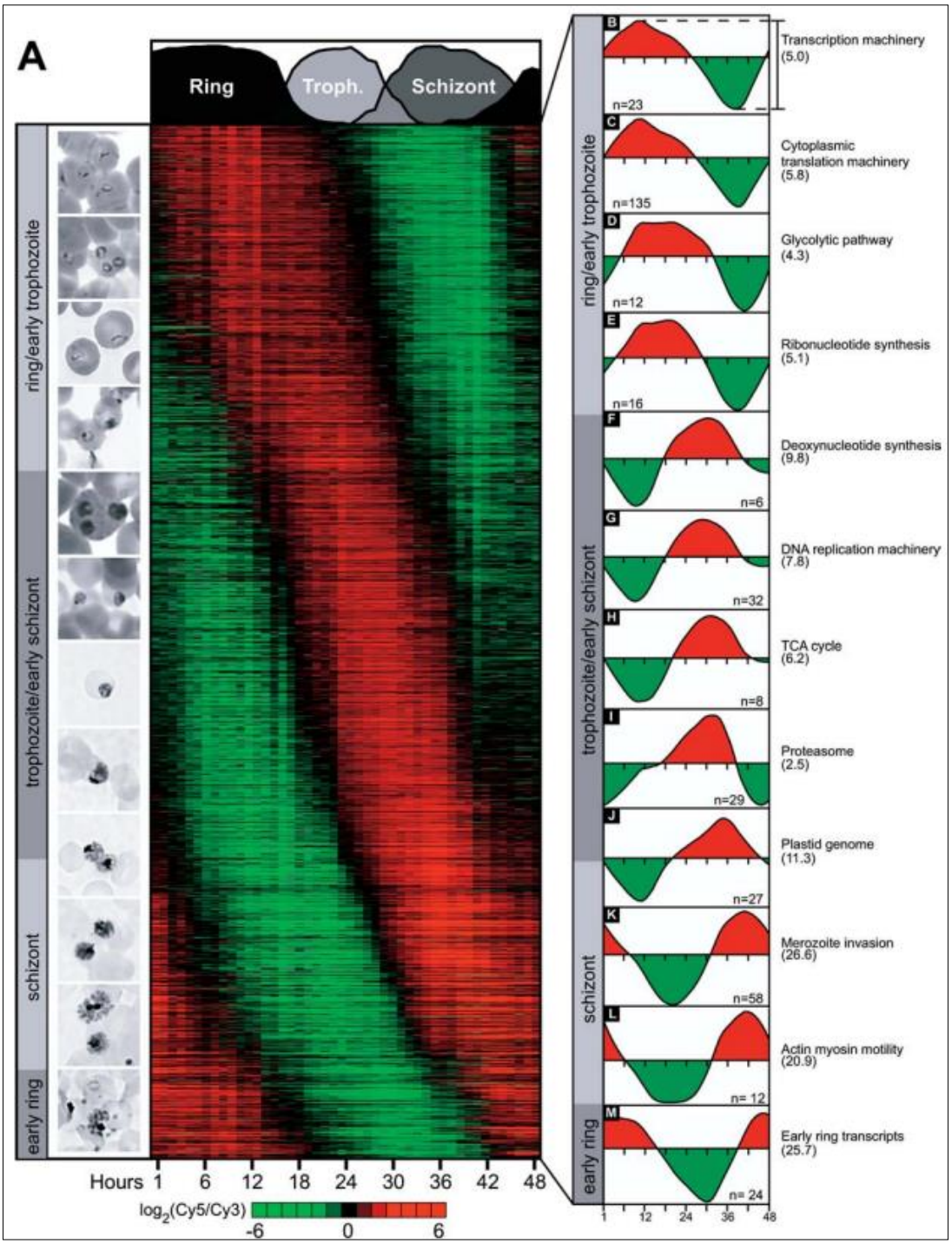
Apesar de existirem estudos mais recentes, como o de Llinás et al. (2006), optei por utilizar o trabalho de Bozdech et al. (2003) por apresentar um conjunto de dados robusto e simplificado, além de focar exclusivamente na cepa HB3. Essa simplificação e o elevado número de citações tornam este estudo uma referência relevante, mesmo sendo mais antigo.

Os resultados foram normalizados e os valores de expressão gênica calculados como uma razão entre a intensidade do cDNA da amostra (Cy5) e a intensidade do cDNA do conjunto de referência (Cy3), gerando assim valores da expressão relativa

para cada gene em cada ponto do ciclo de vida (BOZDECH et al., 2003). Assim, quanto mais próximos esses valores estiverem de 0, menor será a expressão — ou nenhuma expressão em um nível detectável — naquele ponto do ciclo de vida do parasita. Valores positivos indicam que o gene é expresso em um nível mais alto em relação ao conjunto de referência, enquanto valores negativos refletem que sua expressão foi menor do que aquele conjunto. Quanto mais próximo de 0, menor será a atividade do gene (BOZDECH et al., 2003).

A Figura 1 ilustra um diagrama de fases do transcriptoma do IDC do *P. falciparum* (BOZDECH et al., 2003), que mostra os genes ordenados no eixo vertical de acordo com a fase de expressão durante o IDC (anel, trofozoíto, esquizonte), evidenciando a variação na expressão dos genes em cada fase morfológica do parasita; à direita, são mostrados gráficos correlacionando a expressão gênica com processos metabólicos e funções específicas.

Figura 1 - Visão geral do transcriptoma do IDC de *P. falciparum*.



FONTE: BOZDECH et al., 2003.

3.2. BINARIZAÇÃO

Para o início do trabalho, foi realizada a binarização utilizando o conjunto de dados completo (Bozdech et al., 2003), que consiste em dados de 7091 genes ao longo de 48 horas. Ressalta-se que os dados referentes às horas 23 e 29 do ciclo de

vida do *Plasmodium* não foram obtidos devido a problemas técnicos nas placas de microarray (Bozdech et al., 2003). Dessa forma, a matriz inicial analisada neste trabalho apresenta dimensões de 7091 por 46, desconsiderando as colunas categóricas e o cabeçalho.

Essa primeira tentativa de binarização está documentada no anexo deste trabalho, pois não gerou resultados satisfatórios. O gráfico gerado como comparativo com a Figura 1 não apresentou semelhanças significativas com esta, o que levou à conclusão de que esse método não era o mais adequado. Assim, acreditou-se que ele não atenderia às especificidades do conjunto de dados utilizado neste estudo.

Em seguida, foi aplicada outra abordagem de binarização, que, ao final, foi a abordagem mais adequada para a utilização na inferência de Redes Booleanas.

Para essa outra abordagem, iniciou-se com a seleção dos 530 genes, previamente divididos em seus respectivos grupos funcionais conforme exemplificado na Tabela 1. A coluna “Manual annotation” refere-se à interpretação do oligonucleotídeo pelos autores (Bozdech et al., 2003).

Tabela 1 – Cinco primeiras linhas do conjunto de dados dos grupos funcionais.

functional_ group	oligo_ID	Manual annotation	TP 1	TP 2	TP 3	TP 4	TP 5
transcription_ machinery	a3310_4	DNA-directed RNA polymerase 2 subunit, putative	0,91	1,08	1,2	1,72	1,38
transcription_ machinery	b182	transcription factor, putative	1,18	1,39	2,31	1,61	1,58
transcription_ machinery	b471	DNA-directed RNA polymerase II second largest subunit, putative	1,07	1,09	1,28	1,89	1,24
transcription_ machinery	c105	DNA-directed RNA polymerase subunit I, putative	1,34	1,11	1,55	1,68	1,72
transcription_ machinery	c527	DNA-directed RNA polymerase II, putative	1,29	1,42	1,63	1,98	1,63

FONTE: Elaborado pelo autor.

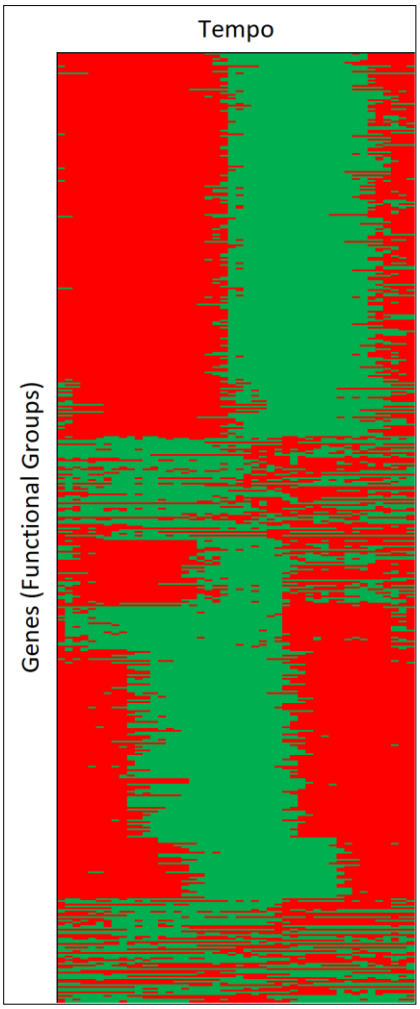
O algoritmo começa com um conjunto de dados D, onde a primeira coluna contém os nomes dos grupos funcionais e as colunas subsequentes a partir da quarta contêm valores numéricos. O primeiro passo é dividir esse conjunto de dados em subconjuntos de genes G_k, onde cada G_k representa um grupo funcional específico.

Para cada grupo de genes G_k, são calculadas as médias $\overline{x}_k^j$ dos valores numéricos presentes em cada coluna numérica j com a média sendo dada pela soma dos valores da coluna dividida pelo número total de genes no grupo funcional.

Uma vez que as médias para cada grupo funcional foram calculadas, o próximo passo é binarizar os dados. Para isso, cada valor $D_{i,j}$ na tabela original é comparado à média $\overline{x}_k^j$ da coluna correspondente ao seu grupo funcional G_k. Se o valor for maior ou igual à média, ele é substituído por 1; se for menor, é substituído por 0. O resultado é uma matriz binarizada B, onde cada elemento $B_{i,j}$ é igual a 1 ou 0, dependendo da comparação com a média calculada.

Aplicando esse algoritmo foi possível construir o gráfico da Figura 2, e, após a comparação com a Figura 1, optou-se por seguir com esse método de binarização. Essa escolha foi feita devido ao maior equilíbrio observado entre os estados dos genes.

Figura 2 - Gráfico que representa os estados binarizados dos 530 genes seguindo o algoritmo proposto pelo autor. Verde: 0 (sem atividade); Vermelho: 1 (com atividade).



FONTE: Elaborado pelo autor.

3.3. FILTRAGEM E OTIMIZAÇÃO DE DADOS

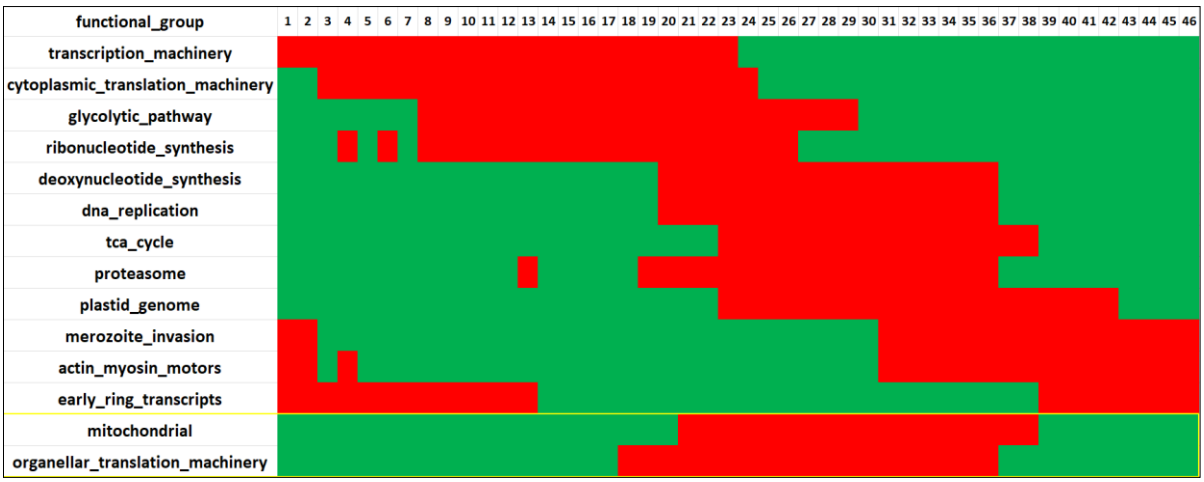
No entanto, essa não será a matriz final utilizada para inferir as Redes Booleanas, uma vez que é necessário otimizar o conjunto de dados para trabalhar com uma menor quantidade de estados. Para isso, todos os genes de cada grupo funcional serão agrupados e tratados como um único estado (um para cada grupo funcional), com o objetivo de simplificar o modelo e otimizar o processamento.

Após o cálculo das médias $\overline{x}_k^j$ para cada coluna dentro de seus respectivos grupos funcionais G_k, foi realizada uma etapa adicional de processamento. As médias calculadas por grupo funcional foram agrupadas de forma que cada grupo fosse representado por uma única linha. Em seguida, foi calculada a média dos valores presentes em cada linha.

Com base nesses novos valores médios por linha, foi realizada uma binarização dos dados. As colunas com valores menores que a média de sua respectiva linha foram definidas como 0, enquanto as colunas com valores iguais ou superiores à média receberam o valor 1.

O resultado visual desse algoritmo está representado abaixo na Figura 3, excluindo os grupos Genes Mitocondriais e Maquinaria de Tradução das Organelas, pois notou-se que aparentemente ambos não demonstraram uma atividade cíclica.

Figura 3 - Gráfico que representa os estados binarizados dos grupos funcionais seguindo o algoritmo proposto pelo autor.



FONTE: Elaborado pelo autor.

Após isso, para trabalhar com a transição de estados, foi criada a matriz transposta desses dados de forma que o tempo esteja no eixo vertical e os grupos no eixo horizontal. Além disso, foram realizados ajustes manuais em elementos específicos da matriz, em pontos onde os dados apresentavam discrepâncias, possivelmente devido a erros de medição ou cálculo na fonte original. Por exemplo, elementos localizados nas posições relativas à síntese de ribonucleotídeos (tempos 4 e 6), proteassoma (tempos 13 e 19) e motores de actina-miosina (tempo 4) foram ajustados para zero, visando corrigir inconsistências que poderiam comprometer a análise subsequente, deixando os dados mais coerentes com as expectativas e eliminando ruídos ou anomalias. O resultado desse processamento está ilustrado na Figura 4.

Figura 4 - Gráfico que representa os estados binarizados transpostos dos grupos funcionais seguindo o algoritmo proposto pelo autor. Verde: 0 (sem atividade); Vermelho: 1 (com atividade).

	transc. machinery	cytopl transl machinery	glycolytic pathway	ribonuc. synthesis	deoxynuc. synthesis	dna replic.	tca cycle	proteasome	plastid genome	merozoite invasion	actin myosin motors	early ring transcripts
TP 1	1	0	0	0	0	0	0	0	0	1	1	1
TP 2	1	0	0	0	0	0	0	0	0	1	1	1
TP 3	1	1	0	0	0	0	0	0	0	0	0	1
TP 4	1	1	0	0	0	0	0	0	0	0	0	1
TP 5	1	1	0	0	0	0	0	0	0	0	0	1
TP 6	1	1	0	0	0	0	0	0	0	0	0	1
TP 7	1	1	0	0	0	0	0	0	0	0	0	1
TP 8	1	1	1	1	0	0	0	0	0	0	0	1
TP 9	1	1	1	1	0	0	0	0	0	0	0	1
TP 10	1	1	1	1	0	0	0	0	0	0	0	1
TP 11	1	1	1	1	0	0	0	0	0	0	0	1
TP 12	1	1	1	1	0	0	0	0	0	0	0	1
TP 13	1	1	1	1	0	0	0	0	0	0	0	1
TP 14	1	1	1	1	0	0	0	0	0	0	0	0
TP 15	1	1	1	1	0	0	0	0	0	0	0	0
TP 16	1	1	1	1	0	0	0	0	0	0	0	0
TP 17	1	1	1	1	0	0	0	0	0	0	0	0
TP 18	1	1	1	1	0	0	0	0	0	0	0	0
TP 19	1	1	1	1	0	0	0	0	0	0	0	0
TP 20	1	1	1	1	1	1	0	1	0	0	0	0
TP 21	1	1	1	1	1	1	0	1	0	0	0	0
TP 22	1	1	1	1	1	1	0	1	0	0	0	0
TP 24	1	1	1	1	1	1	1	1	1	0	0	0
TP 25	0	1	1	1	1	1	1	1	1	0	0	0
TP 26	0	0	1	1	1	1	1	1	1	0	0	0
TP 27	0	0	1	1	1	1	1	1	1	0	0	0
TP 28	0	0	1	0	1	1	1	1	1	0	0	0
TP 30	0	0	1	0	1	1	1	1	1	0	0	0
TP 31	0	0	1	0	1	1	1	1	1	0	0	0
TP 32	0	0	0	0	1	1	1	1	1	0	0	0
TP 33	0	0	0	0	1	1	1	1	1	1	1	0
TP 34	0	0	0	0	1	1	1	1	1	1	1	0
TP 35	0	0	0	0	1	1	1	1	1	1	1	0
TP 36	0	0	0	0	1	1	1	1	1	1	1	0
TP 37	0	0	0	0	1	1	1	1	1	1	1	0
TP 38	0	0	0	0	1	1	1	1	1	1	1	0
TP 39	0	0	0	0	0	0	1	0	1	1	1	0
TP 40	0	0	0	0	0	0	1	0	1	1	1	0
TP 41	0	0	0	0	0	0	0	0	1	1	1	1
TP 42	0	0	0	0	0	0	0	0	1	1	1	1
TP 43	0	0	0	0	0	0	0	0	1	1	1	1
TP 44	0	0	0	0	0	0	0	0	1	1	1	1
TP 45	0	0	0	0	0	0	0	0	0	1	1	1
TP 46	0	0	0	0	0	0	0	0	0	1	1	1
TP 47	0	0	0	0	0	0	0	0	0	1	1	1
TP 48	0	0	0	0	0	0	0	0	0	1	1	1

FONTE: Elaborado pelo autor.

Posteriormente, foi realizada uma etapa de otimização dos dados, eliminando linhas duplicadas da matriz. O objetivo é manter uma quantidade mínima de estados, evitando que repetições prejudiquem a interpretação dos resultados. Além disso, a primeira linha da matriz foi replicada ao final para ilustrar a formação de um ciclo completo, resultando na matriz final pronta para a análise subsequente (Figura 5). Nesse momento, as linhas não mais representam tempos, mas sim estados.

Figura 5 - Matriz de estados otimizada.

transcription machinery	cytoplasmic translation machinery	glycolytic pathway	ribonuc. synthesis	deoxynuc. synthesis	dna replication	tca cycle	proteasome	plastid genome	merozoite invasion	actin myosin motors	early ring transcripts
1	0	0	0	0	0	0	0	0	1	1	1
1	1	0	0	0	0	0	0	0	0	0	1
1	1	1	1	0	0	0	0	0	0	0	1
1	1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	1	1	0	1	0	0	0	0
1	1	1	1	1	1	1	1	1	0	0	0
0	1	1	1	1	1	1	1	1	0	0	0
0	0	1	1	1	1	1	1	1	0	0	0
0	0	1	0	1	1	1	1	1	0	0	0
0	0	0	0	1	1	1	1	1	0	0	0
0	0	0	0	1	1	1	1	1	1	1	0
0	0	0	0	0	0	1	0	1	1	1	0
0	0	0	0	0	0	0	0	1	1	1	1
0	0	0	0	0	0	0	0	0	1	1	1
1	0	0	0	0	0	0	0	0	1	1	1

FONTE: Elaborado pelo autor.

3.4. INFERÊNCIA DE REDES

Foi utilizada uma abordagem para identificar padrões regulatórios válidos em uma Rede Booleana, considerando as transições de estado dos genes ao longo do tempo conforme o modelo proposto por Li et al., 2004.

$$S_i(t+1) = \begin{cases} 1, & se \sum_j a_{ij} S_j(t) > 0 \\ 0, & se \sum_j a_{ij} S_j(t) < 0 \\ S_i(t), & se \sum_j a_{ij} S_j(t) = 0 \end{cases}$$

Onde S_i é o estado de um determinado grupo funcional i e a_{ij} representa uma célula da matriz de regulação. Ou seja, um estado S_2 será determinado pelo estado S_1 e sua matriz de regulação. Neste trabalho, temos os estados dos grupos já definidos, e será preciso encontrar as possíveis matrizes de regulação.

Como exemplo, pode-se levar em consideração a matriz de transição de estados, onde cada linha representa um estado e cada coluna representa um grupo funcional:

$$S = \begin{matrix} S_1 \\ S_2 \\ S_3 \\ S_4 \end{matrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

Sendo A a matriz de regulação, para a transição de S_1 para S_2, temos:

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \rightarrow A \,. \overset{S_1}{\begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}} = \begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \\ a_{2,0} & a_{2,1} \\ a_{3,0} & a_{3,1} \end{bmatrix} \rightarrow \overset{S_2}{\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}}$$

Seguindo o modelo de LI et al., 2004., definimos o seguinte conjunto de restrições:

$$\begin{cases} a_{0,0} + a_{0,1} \geq 0 \\ a_{0,0} + a_{1,1} < 0 \\ a_{0,0} + a_{2,1} \leq 0 \\ a_{0,0} + a_{3,1} \leq 0 \end{cases}$$

Da mesma maneira, outros conjuntos de restrições deverão ser definidos para as outras duas transições de estados da matriz, ou seja, mais dois conjuntos, formando o seguinte conjunto de restrições:

$$\begin{cases} a_{0,0} + a_{0,1} \geq 0 \\ a_{1,0} + a_{1,1} < 0 \\ a_{2,0} + a_{2,1} \leq 0 \\ a_{3,0} + a_{3,1} \leq 0 \end{cases} \qquad \begin{cases} a_{0,0} \geq 0 \\ a_{1,0} < 0 \\ a_{2,0} \leq 0 \\ a_{3,0} \leq 0 \end{cases} \qquad \begin{cases} a_{0,0} + a_{0,2} \geq 0 \\ a_{1,0} + a_{1,2} < 0 \\ a_{2,0} + a_{2,2} \leq 0 \\ a_{3,0} + a_{3,2} \leq 0 \end{cases}$$

Somente levando em consideração as restrições para o primeiro grupo funcional, temos:

$$\begin{cases} a_{0,0} + a_{0,1} \geq 0 \\ a_{0,0} \geq 0 \\ a_{0,0} + a_{0,2} \geq 0 \end{cases}$$

Com isso, teremos 39 soluções possíveis para compor a primeira linha da matriz de regulação, sendo que 3 delas estão exemplificadas a seguir:

Solução 1: 0 0 1 -1

Solução 2: 1 -1 1 -1

Solução 3: 1 1 0 1

Com isso, o objetivo com a matriz de transição de estados do *Plasmodium* é gerar todas as possíveis combinações de influências entre os grupos funcionais, representadas por valores que indicam inibição, ausência de influência ou ativação, e testá-las em relação às mudanças de estado observadas em diferentes momentos.

Padrões que não apresentam influência regulatória são descartados, enquanto os padrões restantes são avaliados para verificar se respeitam as transições de estado exigidas. Os padrões considerados válidos são armazenados de forma organizada para cada gene, e o total de combinações regulatórias válidas é calculado.

Considerando uma sequência de 12 dígitos (número de grupos funcionais), onde cada dígito pode assumir um dos três valores possíveis (-1, 0, 1), o número total de combinações é calculado através do princípio da multiplicação.

Como cada posição da sequência pode ser preenchida independentemente com um dos três valores, o número de combinações possíveis é dado por 3^{12}, que resulta em 531.441 padrões regulatórios potenciais.

Cada valor da sequência numérica tem um significado biológico: -1 representa inibição, 0 indica ausência de influência e 1 corresponde à ativação de um gene. O objetivo é testar essas combinações para identificar quais delas respeitam as transições de estado observadas nas expressões gênicas.

As soluções (padrões de regulação) para cada grupo funcional foram calculadas computacionalmente e estão contabilizadas na Figura 6.

Figura 6 - Possibilidades de padrões de regulação calculados. A quantidade de matrizes possíveis é de aproximadamente $5{,}7 \times 10^{41}$

```
Functional Group 01 has 3973 possible regulation patterns
Functional Group 02 has 3302 possible regulation patterns
Functional Group 03 has 3146 possible regulation patterns
Functional Group 04 has 5010 possible regulation patterns
Functional Group 05 has 2358 possible regulation patterns
Functional Group 06 has 2358 possible regulation patterns
Functional Group 07 has 8778 possible regulation patterns
Functional Group 08 has 2358 possible regulation patterns
Functional Group 09 has 5893 possible regulation patterns
Functional Group 10 has 990 possible regulation patterns
Functional Group 11 has 990 possible regulation patterns
Functional Group 12 has 4148 possible regulation patterns
Total possible matrices = 5.701178e+41
57011784507699614512667749140292279526400O
```

FONTE: Elaborado pelo autor.

Um exemplo de padrão regulatório válido para o grupo funcional Maquinaria de Transcrição, que possui 3.973 padrões de regulação, é a seguinte sequência:

-1 0 0 1 -1 0 -1 1 -1 -1 1 1

Esse padrão descreve as interações regulatórias entre os genes desse grupo, onde alguns genes são ativados, outros inibidos e alguns não apresentam influência direta. Neste exemplo, levando em consideração que cada dígito corresponde a um grupo funcional, na mesma ordem em que estão na Figura 5, a Maquinaria de Transcrição exerce autorregulação de maneira inibitória, além de inibir também a Síntese de Desoxirribonucleotídeos, o Ciclo de Krebs, o Genoma do Plastídio e a Invasão por Merozoítos. Por outro lado, essa maquinaria ativa a Síntese de Ribonucleotídeos, o Proteassoma, o Mecanismo de Actina-Miosina e os Transcritos do Estágio Anelar Inicial. Não há influência regulatória sobre os demais grupos funcionais.

Dessa forma, com mais 11 linhas semelhantes ao exemplo acima, construímos uma possível matriz de regulação.

Para selecionar as matrizes mais otimizadas, temos de levar em conta um raciocínio biológico, de acordo com Kauffman (1993) e Li et al. (2004), redes biológicas estáveis tendem a ter poucos atratores, sendo que um deles possui uma bacia de

atração significativa. Mesmo que essa bacia seja extensa, apenas uma fração dos estados faz parte do atrator, e os estados transientes, que não pertencem a ele, são biologicamente irrelevantes. Seguindo essa lógica, o objetivo é manter apenas as linhas que apresentam a maior quantidade de zeros ou que estejam dentro de uma margem de tolerância próxima a esse valor máximo.

Portanto, como o processo de regulação gênica é geralmente otimizado para economizar energia, foi implementada uma estratégia para filtrar as linhas que contêm o maior número de zeros. A ideia central é que a célula evita gastar energia desnecessária, ativando apenas os processos regulatórios essenciais, o que resulta em matrizes mais esparsas, com menos ativações gênicas.

Esse processo é aplicado individualmente a cada grupo funcional, onde o número de colunas da matriz reflete os diferentes genes analisados. Para cada grupo funcional, são gerados arquivos contendo apenas as linhas mais esparsas (Figura 7), preservando aquelas que melhor refletem um comportamento biológico otimizado, ao eliminar combinações menos plausíveis ou mais "custosas" do ponto de vista energético.

Figura 7 - Possibilidades de padrões de regulação selecionando as linhas que representam um menor custo energético.

```
small_01.txt has 03 rows
small_02.txt has 08 rows
small_03.txt has 06 rows
small_04.txt has 07 rows
small_05.txt has 08 rows
small_06.txt has 08 rows
small_07.txt has 12 rows
small_08.txt has 08 rows
small_09.txt has 03 rows
small_10.txt has 05 rows
small_11.txt has 05 rows
small_12.txt has 24 rows
Total possible matrices: 11147673600
```

FONTE: Elaborado pelo autor.

4. RESULTADOS

Foi possível criar um Grafo de Transição de Estados (STG) para uma Rede Booleana com doze estados (ou grupos funcionais). Inicialmente, são geradas diversas matrizes regulatórias aleatórias para esses grupos funcionais a partir dos

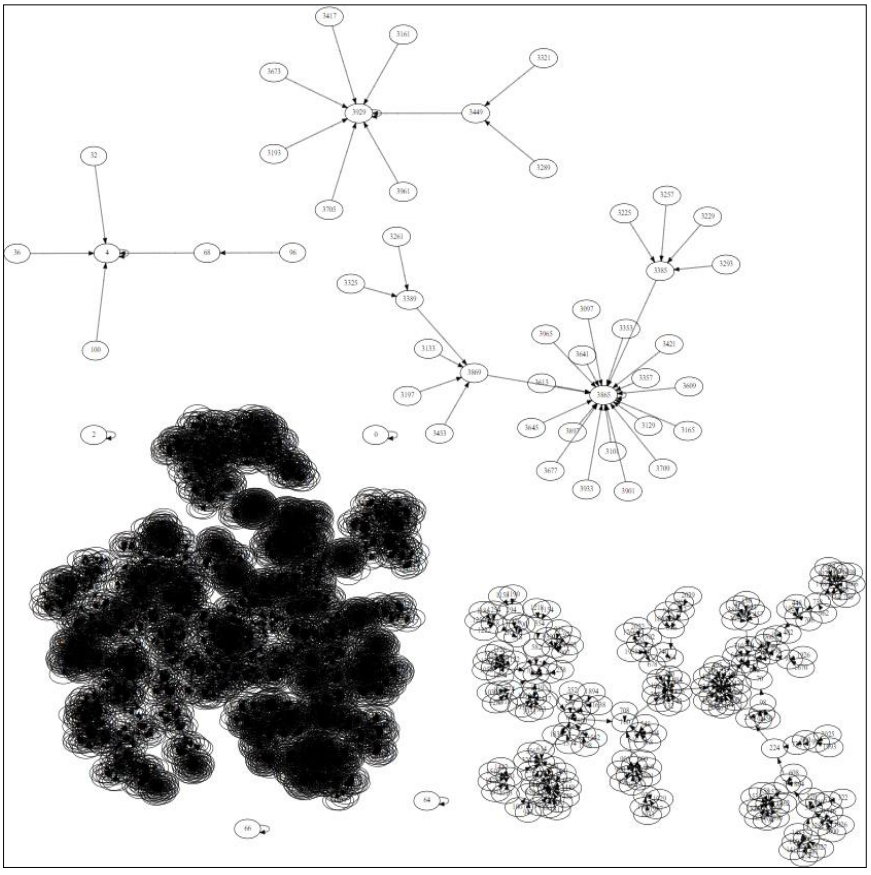
arquivos gerados, garantindo a reprodutibilidade dos resultados através do uso de uma “semente” definida. A partir dessas matrizes, é selecionada aquela com um menor número de bacias de atração, ou seja, grupos de estados que se conectam entre si. Então o STG é construído, representando as transições entre os diferentes estados da rede ao longo do tempo.

O grafo gerado é então analisado com o objetivo de identificar seus componentes conexos. Para cada bacia de atração, calcula-se o seu tamanho, o número total de bacias de atração distintas e para fins de análise, consideramos também o tamanho da maior bacia de atração.

Ao final, o número de componentes distintos é exibido, juntamente com o tamanho da maior bacia de atração.

Para uma primeira tentativa, processamos apenas 20 matrizes aleatórias, e o resultado foi uma rede com 9 bacias de atração (Figura 8).

Figura 8 - Esse STG apresenta 9 Bacias de Atração.

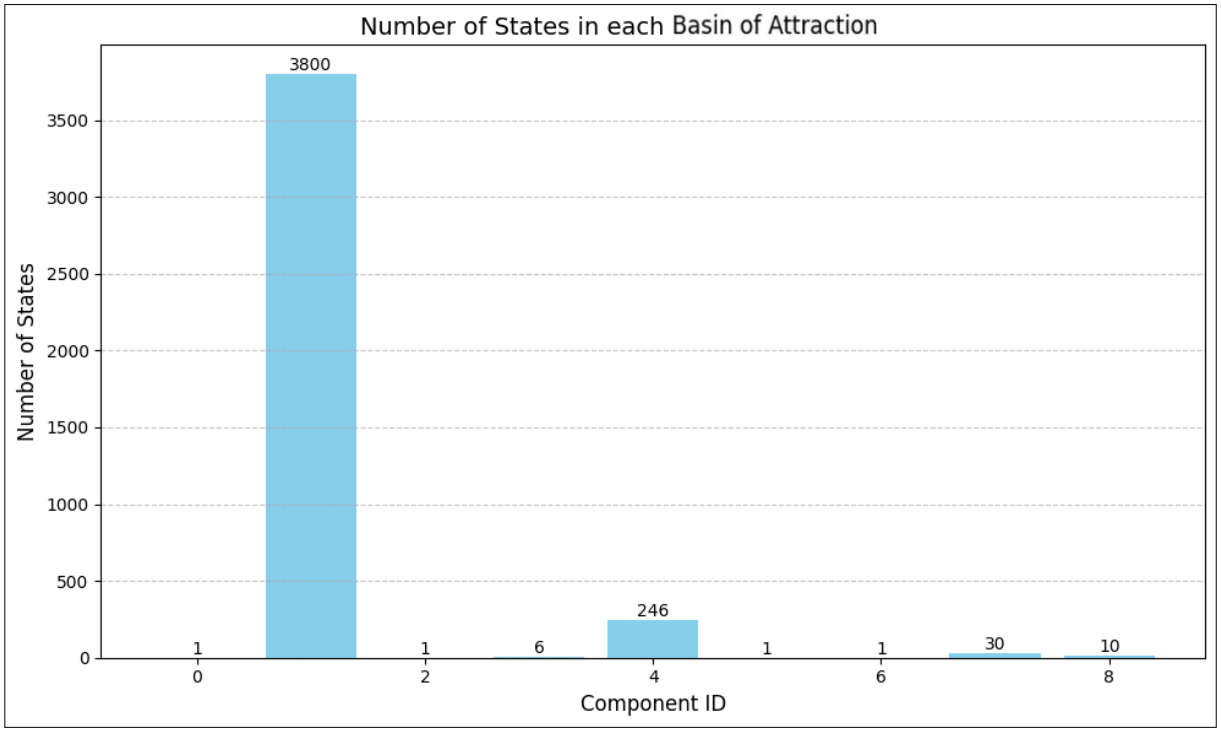


FONTE: Elaborado pelo autor.

Em seguida foi construído um gráfico de barras que representa os tamanhos dos componentes conexos em um STG. O gráfico é gerado com barras que representam o número de estados em cada componente conexo. Cada barra recebe uma anotação, exibindo o valor exato da frequência acima da respectiva barra, o que facilita a leitura dos resultados.

Dessa forma, pode-se concluir que o componente conexo que apresenta o maior número de nós na Figura 8 apresenta 3800 estados (Figura 9).

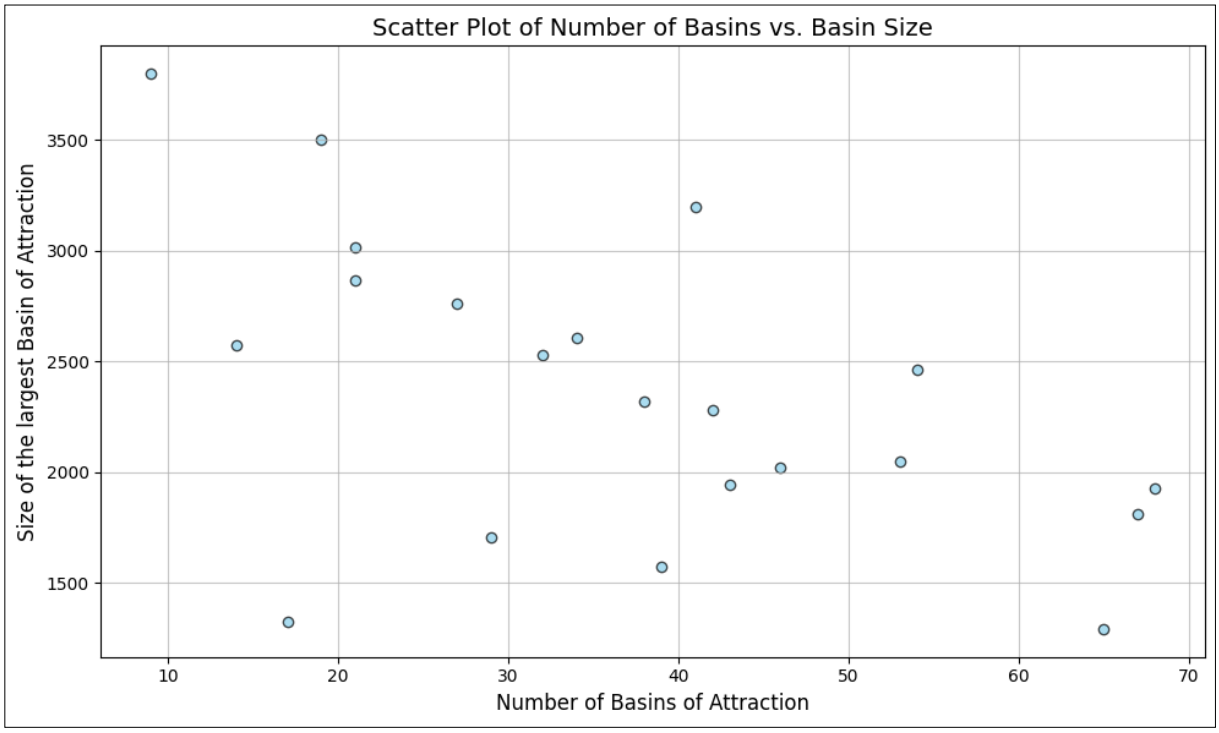
Figura 9 - Frequência de tamanhos componentes conexos para 9 bacias de atração.



FONTE: Elaborado pelo autor.

Com isso foi possível criar um gráfico de dispersão (Figura 10) para visualizar a relação entre o número de bacias de atração e o tamanho da bacia que contém a sequência temporal de estados representada na Figura 5. Quanto mais próxima do quadrante superior esquerdo for a matriz, mais próxima do ideal ela será, ou seja, um menor número de bacias de atração possível, contendo muitos componentes conexos, que indica uma bacia de atração grande.

Figura 10 - Gráfico de dispersão representando 20 redes.

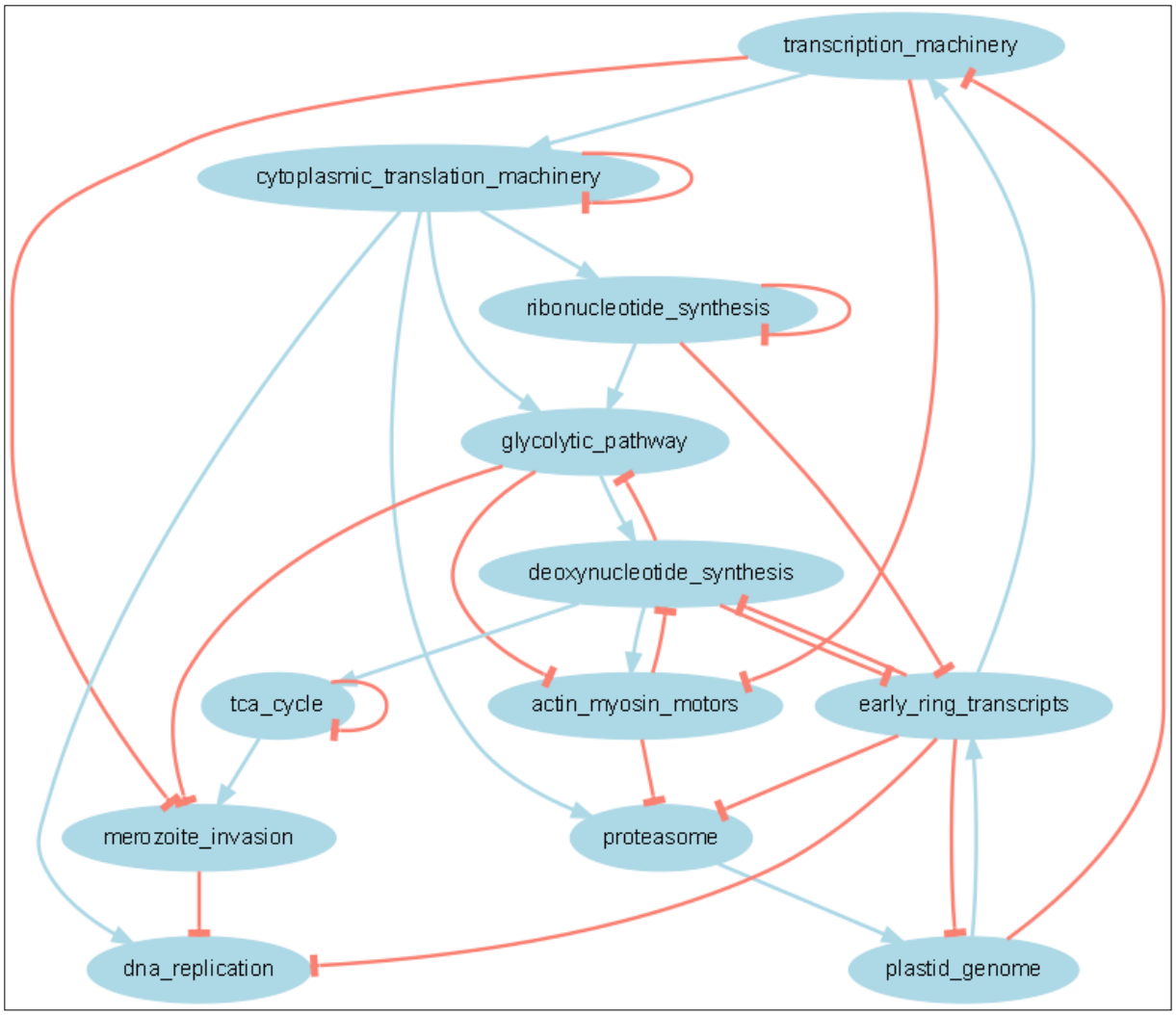


FONTE: Elaborado pelo autor.

Por fim, foi possível criar uma Rede Booleana com limiar (TBN) com base em uma matriz de adjacência que representa as interações entre módulos (Figura 11).

No gráfico, o azul claro indica interações excitatórias e o salmão representa interações inibitórias.

Figura 11 - TBN para a rede com 9 bacias de atração



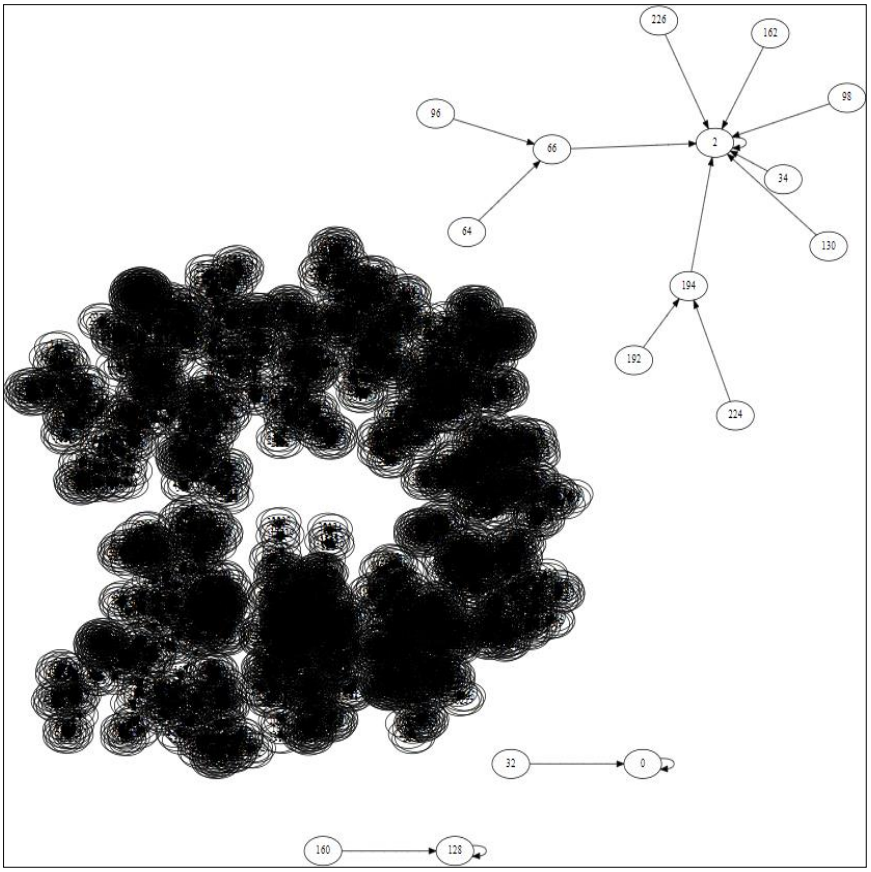
FONTE: Elaborado pelo autor.

Para uma segunda tentativa, foi exigido um processamento maior e mais tempo, pois geramos 100.000 matrizes aleatórias na tentativa de otimizar ainda mais e chegar em uma rede ideal.

Com isso foi possível chegar em uma rede otimizada, com 4 bacias de atração apenas, representada pela matriz de regulação abaixo. Os gráficos estão ilustrados nas figuras 12 a 15.

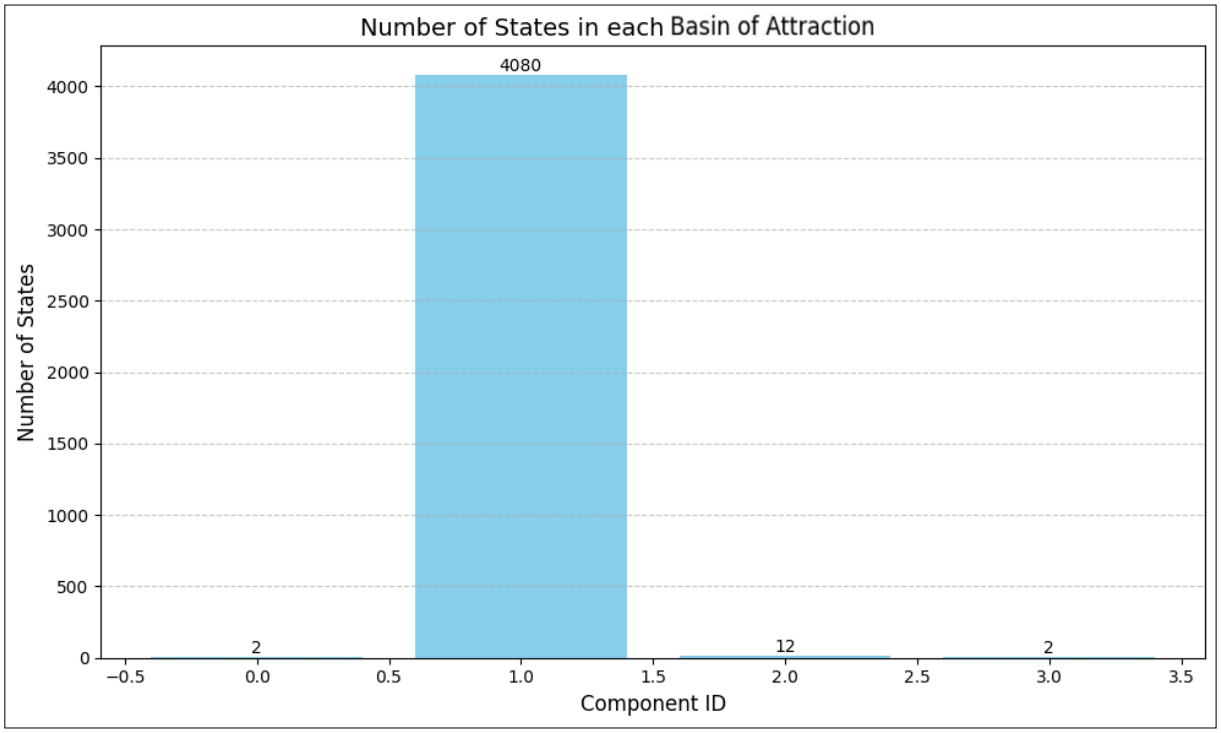
$$A = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 \\
1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\
-1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
-1 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0
\end{bmatrix}$$

Figura 12 – A rede obtida A apresenta 4 Bacias de Atração.



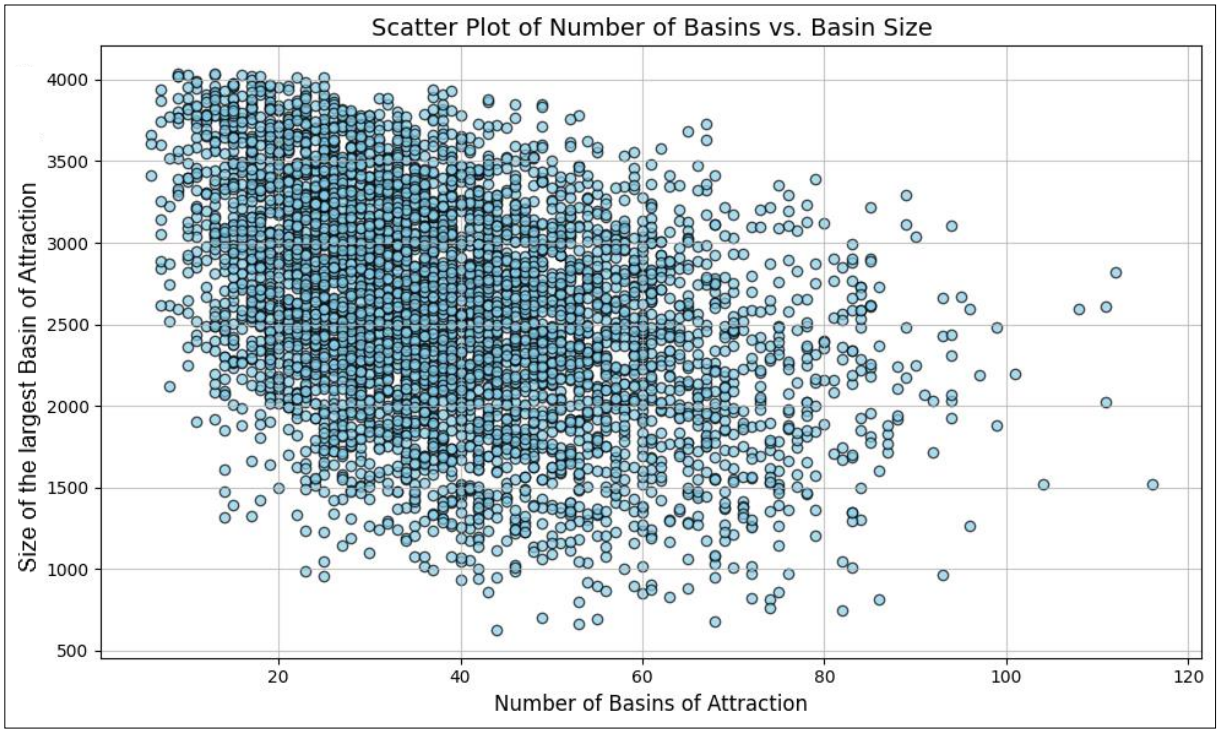
FONTE: Elaborado pelo autor.

Figura 13 - Tamanhos das 4 bacias de atração.



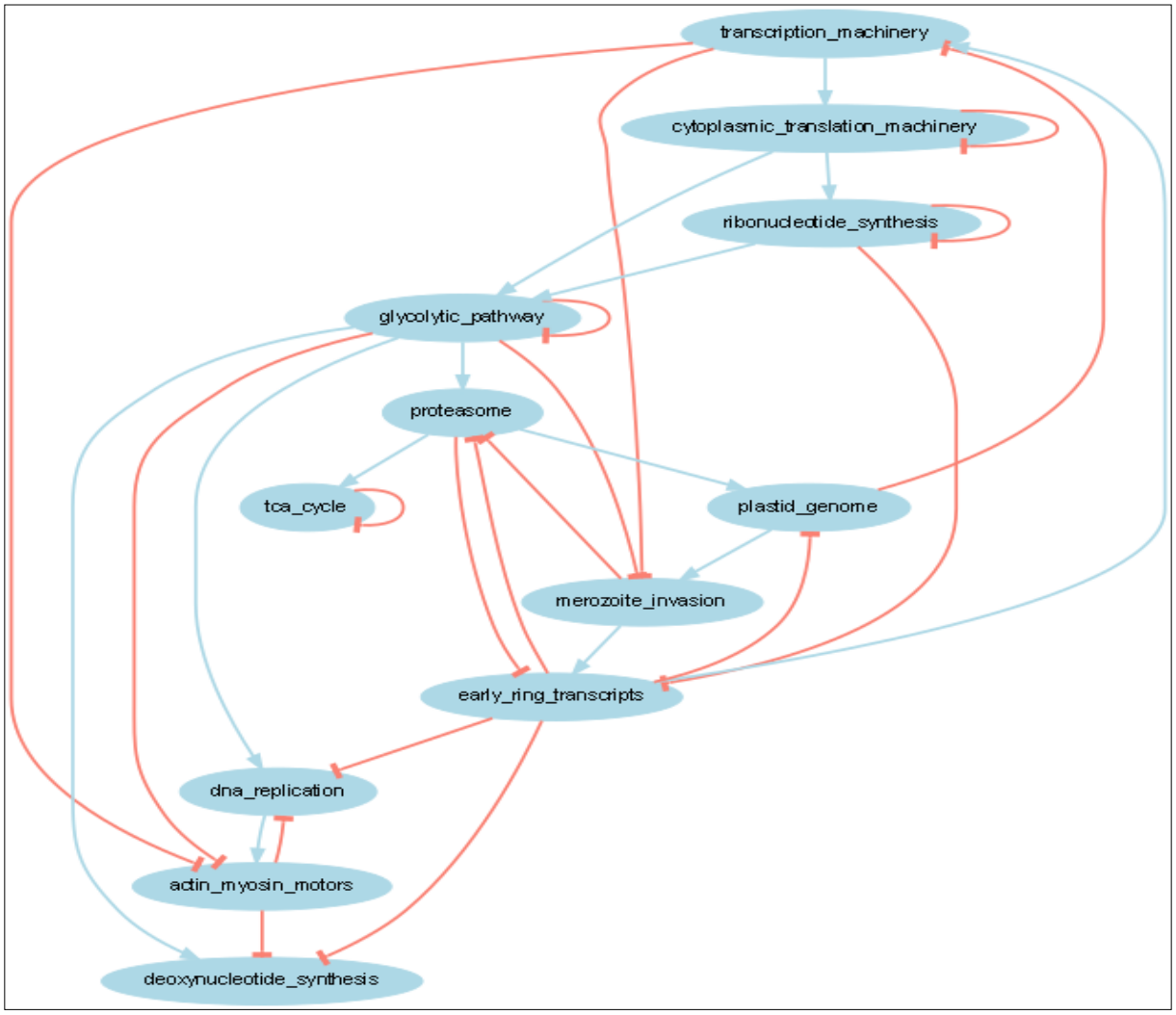
FONTE: Elaborado pelo autor.

Figura 14 - Gráfico de dispersão representando uma amostragem de 5.000 redes. A inclusão das 100.000 redes totais não foi viabilizada devido às limitações no processamento e ao risco de comprometer a clareza visual do gráfico, tornando-o excessivamente sobrecarregado.



FONTE: Elaborado pelo autor.

Figura 15 - TBN para a rede com 4 bacias de atração.



FONTE: Elaborado pelo autor.

5. DISCUSSÃO

Ao analisar a Rede Booleana inferida de forma mais robusta, é possível observar que certos grupos funcionais aparentam desempenhar um papel central na regulação de outros grupos, ou são fortemente regulados por eles. Entre esses grupos destacam-se os transcritos do estágio anelar inicial, o proteassoma e a via glicolítica.

Adicionalmente, de acordo com revisões na literatura, sabe-se que o Apicoplasto se configura como um alvo terapêutico de grande interesse (RALPH; D'OMBRAIN; MCFADDEN, 2001), uma vez que é uma organela exclusiva do parasita e ausente nas células humanas (LIM; MCFADDEN, 2010). O Genoma do Plastídio do Plasmodium falciparum está intimamente relacionado ao Apicoplasto. Este genoma,

que é circular e extracromossômico, contém genes que codificam proteínas fundamentais para funções como a síntese de lipídios e a produção de precursores de aminoácidos (BOZDECH et al., 2003). Devido à sua importância funcional e à presença de genes de função desconhecida, o genoma plastidial do Apicoplasto pode ser considerado um alvo promissor para o desenvolvimento de novas terapias antimaláricas (BOZDECH et al., 2003).

Além disso, entre as $5{,}7 \times 10^{41}$ matrizes possíveis, não foi possível identificar uma matriz ideal, ou seja, uma configuração que apresentasse apenas uma bacia de atração. Esse resultado está diretamente relacionado ao fato de que, neste estudo, buscamos exclusivamente uma única bacia de atração, já que o foco é analisar a evolução do IDC do *Plasmodium*. Essa abordagem não leva em conta outras variáveis importantes, como divisão celular e outros mecanismos biológicos fundamentais que a célula exerce. Esse nível de simplificação reflete a limitação do modelo computacional, que, embora útil para explorar aspectos específicos da dinâmica do IDC, não abrange a vasta complexidade e infinidade de variáveis presentes em um sistema biológico real. Portanto, os resultados apresentados devem ser interpretados como uma representação simplificada do comportamento dinâmico do IDC do *Plasmodium*.

6. CONCLUSÃO

Neste trabalho, buscamos inferir uma Rede Booleana que permita a identificação de alvos terapêuticos promissores para intervenções farmacêuticas no IDC do *P. falciparum*. Apesar de nossos resultados indicarem a relevância dos transcritos do estágio anelar inicial e o genoma plastidial, a complexidade intrínseca do sistema biológico sugere que mais estudos são necessários.

Recomenda-se o uso de uma capacidade computacional mais robusta para gerar um maior número de matrizes, visando a descoberta de uma configuração que apresente uma única bacia de atração, idealmente. Uma outra possibilidade seria uma maneira mais otimizada de selecionar as matrizes, como por exemplo selecionar as soluções que mais se repetem em uma grande quantidade de matrizes geradas, ou por meio da utilização de algoritmos genéticos.

Adicionalmente, é fundamental realizar investigações adicionais em como o Genoma do Plastídio se comporta no ciclo de vida do *Plasmodium*, já que sua

exclusividade no parasita o torna um alvo terapêutico de grande interesse. Por isso é de suma importância que haja um estudo interdisciplinar com pesquisadores cuja linha de pesquisa tenha enfoque no *P. falciparum*.

Por fim, este estudo evidencia a importância da modelagem computacional no entendimento das dinâmicas biológicas do *Plasmodium*, destacando que, embora os resultados obtidos sejam uma simplificação, eles abrem caminhos promissores para o desenvolvimento de novas estratégias terapêuticas e para a pesquisa futura na área.

O código em Python escrito para este trabalho pode ser encontrado no anexo, ou no link para o GitHub: https://github.com/raulgranja/malaria-boolean-network

7. REFERÊNCIAS

ANDRADE, T. P. DE. Interações gênicas usando redes booleanas limiarizadas modeladas como um problema de satisfação de restrições. [s.d.].

BOZDECH, Z. et al. The Transcriptome of the Intraerythrocytic Developmental Cycle of Plasmodium falciparum. **PLoS Biology**, v. 1, n. 1, p. e5, 18 ago. 2003.

BRASIL. Ministério da Saúde. Secretaria de Vigilância em Saúde. *Guia de vigilância em saúde*. Vol. 2. 5. ed. Brasília: Ministério da Saúde, 2021. p. 921-954.

ELLSON, J. et al. **Graphviz - Graph Visualization Software**. 2004. Disponível em: https://graphviz.org/. Acesso em: 13 out. 2024.

FARROW, R. E. et al. The mechanism of erythrocyte invasion by the malarial parasite, Plasmodium falciparum. **Seminars in Cell & Developmental Biology**, v. 22, n. 9, p. 953–960, 1 dez. 2011.

GOOGLE LLC. **Google Colaboratory: Google Colab**. Disponível em: https://colab.research.google.com/. Acesso em: 13 out. 2024.

HARRIS, C. R. et al. **Array programming with NumPy**. Nature, v. 585, p. 357–362, 2020. Disponível em: https://numpy.org. Acesso em: 13 out. 2024.

HEMEDAN, A. A. et al. Boolean modelling as a logic-based dynamic approach in systems medicine. **Computational and Structural Biotechnology Journal**, v. 20, p. 3161–3172, 2022.

HIGA, C. H. A. **Inferência de redes de regulação gênica utilizando o paradigma de crescimento de sementes.** 2012. 78 f. Tese (Doutorado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2012.

HUNTER, J. D. **Matplotlib: A 2D Graphics Environment**. Computing in Science & Engineering, v. 9, n. 3, p. 90-95, 2007. Disponível em: https://matplotlib.org. Acesso em: 13 out. 2024.

IPYTHON DEVELOPMENT TEAM. **IPython display module**. Disponível em: https://ipython.readthedocs.io/en/stable/api/generated/IPython.display.html. Acesso em: 13 out. 2024.

JACQUELINE et al. A snapshot of a representative Brazilian state of illegal mining in indigenous areas during the era of malaria elimination. **Cadernos de Saúde Pública**, v. 40, n. 6, 1 jan. 2024.

KAUFFMAN, S. A. Metabolic stability and epigenesis in randomly constructed genetic nets. **Journal of Theoretical Biology**, v. 22, n. 3, p. 437–467, mar. 1969.

LI, F. et al. The yeast cell-cycle network is robustly designed. **Proceedings of the National Academy of Sciences**, v. 101, n. 14, p. 4781–4786, 22 mar. 2004.

LIM, L.; MCFADDEN, G. I. The evolution, metabolism and functions of the apicoplast. **Philosophical Transactions of the Royal Society B: Biological Sciences**, v. 365, n. 1541, p. 749–763, 12 mar. 2010.

LIN, X.; LI, X.; LIN, X. A Review on Applications of Computational Methods in Drug Screening and Design. **Molecules**, v. 25, n. 6, p. 1375, 18 mar. 2020.

LLINÁS, M.; BOZDECH, Z.; WONG, E. D.; ADAI, A. T.; DERISI, J. L. Comparative whole genome transcriptome analysis of three Plasmodium falciparum strains. **Nucleic Acids Research, Oxford**, v. 34, n. 4, p. 1166–1173, 21 fev. 2006. DOI: 10.1093/nar/gkj517.

NADEEM, M. F. et al. Fixation of pfcrt chloroquine resistance alleles in Plasmodium falciparum clinical isolates collected from unrest tribal agencies of Pakistan. **Brazilian Journal of Biology**, v. 83, 2023.

OLLIARO, P. Mode of action and mechanisms of resistance for antimalarial drugs. **Pharmacology & Therapeutics**, v. 89, n. 2, p. 207–219, fev. 2001.

PYTHON SOFTWARE FOUNDATION. **collections — Container datatypes**. Python v3.10. Disponível em: https://docs.python.org/3/library/collections.html. Acesso em: 13 out. 2024.

PYTHON SOFTWARE FOUNDATION. **itertools — Functions creating iterators for efficient looping**. Python v3.10. Disponível em: https://docs.python.org/3/library/itertools.html. Acesso em: 13 out. 2024.

PYTHON SOFTWARE FOUNDATION. **python-docx: Create and update Microsoft Word (.docx) files**. Disponível em: https://python-docx.readthedocs.io/. Acesso em: 13 out. 2024.

PYTHON SOFTWARE FOUNDATION. **random — Generate pseudo-random numbers**. Python v3.10. Disponível em: https://docs.python.org/3/library/random.html. Acesso em: 13 out. 2024.

RALPH, S. A.; D'OMBRAIN, M. C.; MCFADDEN, G. I. The apicoplast as an antimalarial drug target. **Drug Resistance Updates**, v. 4, n. 3, p. 145–151, jun. 2001.

SAWYER, B. J.; KHAN, H.; LE, H. V. Antimalarial drugs. p. 363–396, 1 jan. 2023.

SHMULEVICH, I.; ZHANG, W. Binary analysis and optimization-based normalization of gene expression data. **Bioinformatics**, v. 18, n. 4, p. 555–565, 1 abr. 2002.

SU, C.; PANG, J. Target Control of Asynchronous Boolean Networks. **IEEE/ACM Transactions on Computational Biology and Bioinformatics**, p. 1–1, 1 jan. 2021.

SWAROOP KUMAR PANDEY et al. Drug Development Strategies for Malaria: With the Hope for New Antimalarial Drug Discovery—An Update. **Advances in medicine**, v. 2023, p. 1–10, 14 mar. 2023.

TEIXEIRA, P. **dataframe_image: Embed pandas DataFrames as images in documents**. Disponível em: https://github.com/dexplo/dataframe_image. Acesso em: 13 out. 2024.

THE PANDAS DEVELOPMENT TEAM. **pandas: powerful Python data analysis toolkit**. Versão 1.5.0, 2024. Disponível em: https://pandas.pydata.org/. Acesso em: 13 out. 2024.

WORLD HEALTH ORGANIZATION. World malaria report 2023. Geneva: World Health Organization, 2023.

São Paulo, 15 de outubro de 2024.

Raul Granja Adoglio

Prof. Dr. Ronaldo Fumio Hashimoto

8 ANEXOS

8.1. BINARIZAÇÃO COM ALGORITMO DA BIBLIOGRAFIA

A Tabela 2 apresenta as cinco primeiras linhas do conjunto de dados após um processamento básico para aprimorar a visualização. Observa-se a presença de valores ausentes em alguns genes no conjunto de dados original. “Oligo_ID” é um identificador para diferenciar os oligonucleotídeos que correspondem a diferentes genes ou regiões do genoma.

Tabela 2 - Conjunto de dados inicial.

Oligo_ID	TP 1	TP 2	TP 3	TP 4	TP 5
a10325_1	nulo	nulo	nulo	nulo	nulo
a10325_16	nulo	nulo	nulo	nulo	nulo
a10325_20	1.106	0.842683478	1.566988255	1.00821535	1.281087784
a10325_26	0.827	0.933344535	1.483985684	0.834908613	0.784240008
a10325_29	2.676	2.836065747	2.155552193	2.435149965	1.658594554

FONTE: Elaborado pelo autor.

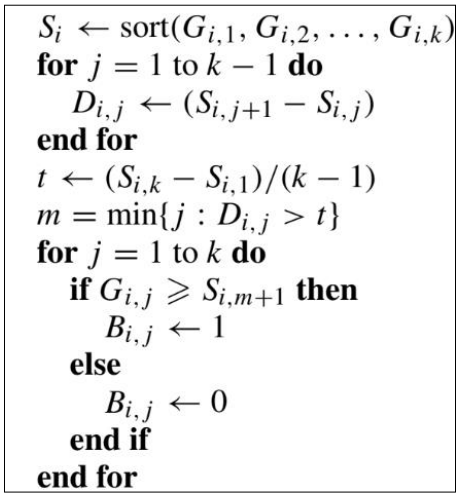
Com a Tabela 1, foi utilizado um algoritmo proposto por SHMULEVICH; ZHANG, 2002, cujo objetivo é realizar a binarização dos estados dos genes, transformando dados contínuos em valores binários (0 ou 1) com base nas variações relativas de expressão gênica. Esse algoritmo está ilustrado na Figura 17. A entrada de dados é um vetor G_i, que representa a expressão de um gene i em k condições diferentes (SHMULEVICH; ZHANG, 2002).

Primeiramente, o vetor G_i é ordenado em ordem crescente, resultando em S_i. Em seguida, o algoritmo calcula as diferenças $D_{i,j}$ entre os valores consecutivos do vetor ordenado. O valor t é calculado como a variação média entre o primeiro e o último valores de S_i, normalizada pelo número de diferenças, servindo como um parâmetro que representa a mudança média na expressão (SHMULEVICH; ZHANG, 2002).

O algoritmo identifica o índice m, que corresponde à primeira diferença $D_{i,j}$ maior que t. Finalmente, os dados de expressão são binarizados: se $G_{i,j}$ for maior que o valor na posição $m+1$ de S_i, o valor binarizado $B_{i,j}$ será 1; caso contrário, será 0. Assim, o vetor binário B_i indica a presença ou ausência de expressão significativa do

gene nas diferentes condições, permitindo a identificação de mudanças relevantes na expressão gênica (SHMULEVICH; ZHANG, 2002).

Figura 17 - Visão geral do transcriptoma do IDC de *P. falciparum*.



$S_i \leftarrow \text{sort}(G_{i,1}, G_{i,2}, \ldots, G_{i,k})$
for $j = 1$ to $k - 1$ **do**
  $D_{i,j} \leftarrow (S_{i,j+1} - S_{i,j})$
end for
$t \leftarrow (S_{i,k} - S_{i,1})/(k - 1)$
$m = \min\{j : D_{i,j} > t\}$
for $j = 1$ to k **do**
  if $G_{i,j} \geqslant S_{i,m+1}$ **then**
    $B_{i,j} \leftarrow 1$
  else
    $B_{i,j} \leftarrow 0$
  end if
end for

FONTE: SHMULEVICH; ZHANG, 2002

Para iniciar a aplicação deste algoritmo, primeiramente, foi feito um processamento para retirar as linhas com dados faltantes, resultando em uma tabela com atividades de 4158 genes em 46 horários distintos (Tabela 3).

Tabela 3 – Conjunto de dados sem a linhas que apresentavam dados nulos.

Oligo_ID	TP 1	TP 2	TP 3	TP 4	TP 5
a10325_29	2,676	2,836065747	2,155552193	2,435149965	1,658594554
a10325_30	2,896	3,068530202	2,827118502	2,273228425	1,818121689
a10325_30j	2,392	1,931779095	1,945949693	1,673612141	1,530729311
a10325_32	2,392	3,168489854	2,192442158	1,881074317	2,171273203
a10935_1	1,559	1,499395336	1,127661474	1,295373374	1,485672115

FONTE: Elaborado pelo autor.

Foi realizado um processamento dos dados aplicando uma função de ordenação crescente às linhas e organizando-as em listas. Em seguida, os nomes originais das colunas foram recuperados e uma tabela contendo os dados ordenados foi criada, preservando esses nomes. Além disso, as colunas correspondentes aos pontos temporais (TP) foram renomeadas e numeradas de 1 a 48. Como resultado da ordenação, os títulos das colunas não correspondem mais às horas originais (Tabela 4).

Tabela 4 – Conjunto de dados ordenados, sem a linhas que apresentavam dados nulos.

Oligo_ID	1	2	3	4	5
a10325_29	0,04930541	0,061	0,069063135	0,091778527	0,111867838
a10325_30	0,027790322	0,033	0,039940128	0,049180739	0,064053362
a10325_30j	0,019722164	0,021	0,027458838	0,036065876	0,037
a10325_32	0,025100937	0,028	0,041375565	0,041803628	0,044100557
a10935_1	0,59	0,590540636	0,611237907	0,65496817	0,66689742

FONTE: Elaborado pelo autor.

Foi realizado um cálculo para obter a diferença entre os valores consecutivos nas colunas numéricas da tabela ordenada, gerando uma nova tabela com uma coluna a menos (Tabela 5). As duas primeiras colunas foram mantidas como identificadores, enquanto as diferenças foram calculadas para as colunas subsequentes. Por fim, as colunas foram renomeadas de acordo com as novas diferenças calculadas.

Tabela 5 – Diferença entre valores consecutivos do conjunto de dados ordenado.

Oligo_ID	1	2	3	4	5
a10325_29	0,01169459	0,008063135	0,022715392	0,020089311	0,003288957
a10325_30	0,005209678	0,006940128	0,009240611	0,014872623	0,000642978
a10325_30j	0,001277836	0,006458838	0,008607038	0,000934124	0,006632413
a10325_32	0,002899063	0,013375565	0,000428063	0,002296929	0,003899443
a10935_1	0,000540636	0,020697271	0,043730263	0,01192925	0,021644281
a11025_2	0,008102067	0,038297633	0,021702367	0,008578662	0,005645849

FONTE: Elaborado pelo autor.

Foi realizado um processamento para separar os dados qualitativos e numéricos, com a seleção das duas primeiras colunas contendo informações qualitativas, que foram armazenadas em uma nova tabela. Posteriormente, foram selecionadas as colunas numéricas remanescentes e calculada a diferença entre o último e o primeiro valor de cada linha. Essa diferença foi dividida pelo número de colunas numéricas menos uma, a fim de obter a média por coluna. Por fim, foi criada uma tabela combinando os dados qualitativos com as médias das diferenças calculadas (Tabela 6).

Tabela 6 – Médias das diferenças entre valores consecutivos.

Oligo_ID	t
a10325_29	0,1556435224
a10325_30	0,1184077114
a10325_30j	0,1223471384
a10325_32	0,1335503073
a10935_1	0,03033275171

FONTE: Elaborado pelo autor.

Foi realizado um processamento para comparar as linhas de duas tabelas e armazenar os resultados em uma lista. Para cada linha de uma das tabelas, foram recuperados determinados valores e buscada a linha correspondente na outra tabela. Em seguida, identificou-se a primeira coluna na segunda tabela onde o valor era superior ao valor recuperado, e o índice dessa coluna foi armazenado. Os resultados foram adicionados à lista e, posteriormente, uma tabela foi criada com as informações coletadas (Tabela 7).

Tabela 7 – Diferença entre valores consecutivos do conjunto de dados ordenado.

Oligo_ID	MIN_INDEX
a10325_29	27
a10325_30	21
a10325_30j	20
a10325_32	20
a10935_1	3

FONTE: Elaborado pelo autor.

Posteriormente, os valores da tabela original foram comparados com os valores correspondentes da tabela ordenada. Essa comparação permitiu a criação da nova tabela para indicar a presença de atividade gênica significativa, onde cada posição foi preenchida com 1 caso a condição fosse satisfeita (ou seja, se o valor no conjunto de dados original fosse maior ou igual ao valor no conjunto de dados ordenados), e 0 caso contrário. Assim, o processamento resultou em uma representação binária das atividades gênicas (Tabela 8).

Tabela 8 – Tabela com dados binarizados.

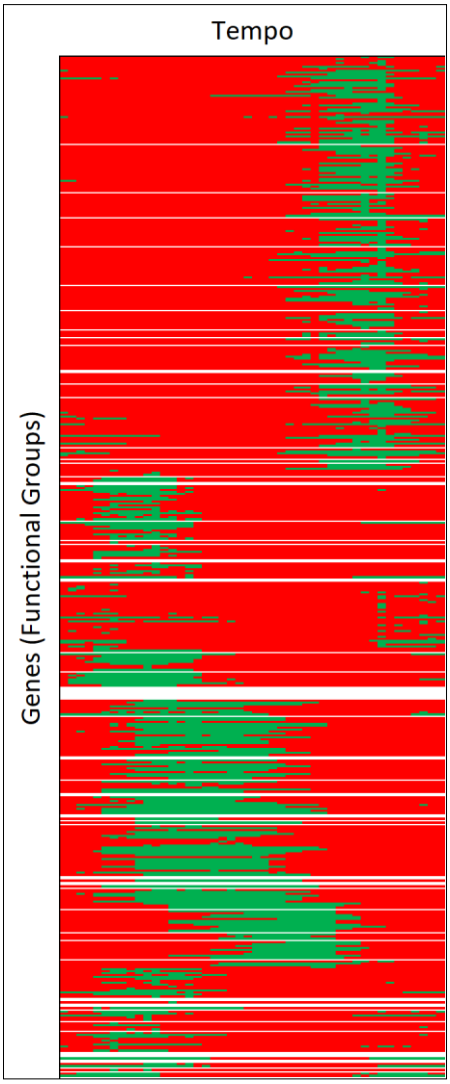
Oligo_ID	TP 1	TP 2	TP 3	TP 4	TP 5
b558	1	1	0	1	0
c345	1	1	0	1	1
c97	1	1	0	1	0
d12635_1	1	1	0	1	1
d49176_44	1	1	0	1	1

FONTE: Elaborado pelo autor.

Após esse passo, foi realizada uma filtragem específica para os genes presentes em um outro conjunto de dados disponibilizado (BOZDECH et al., 2003), que contém uma seleção de 530 genes envolvidos nos principais processos celulares do *Plasmodium*. Esses genes foram organizados em grupos funcionais, tais como: (1) Maquinaria de transcrição, (2) Maquinaria de tradução citoplasmática, (3) Via glicolítica, (4) Síntese de ribonucleotídeos, (5) Síntese de desoxirribonucleotídeos, (6) Replicação do DNA, (7) Ciclo do ácido tricarboxílico (ou ciclo de Krebs), (8) Proteassoma, (9) Genoma do plastídio, (10) Invasão por merozoítos, (11) Mecanismo de actina-miosina, (12) Transcritos do estágio anelar inicial, (13) Genes mitocondriais, (14) Maquinaria de tradução das organelas. Esses grupos foram denominados "grupos funcionais" (BOZDECH et al., 2003).

Isso possibilitou a elaboração de um gráfico com as mesmas cores da Figura 1, com o objetivo de analisar possíveis semelhanças (Figura 16).

Figura 16 - Gráfico que representa os estados binarizados dos 530 genes seguindo o algoritmo encontrado na literatura. Verde: 0 (sem atividade); Vermelho: 1 (com atividade).



FONTE: Elaborado pelo autor

8.2. CÓDIGO PYTHON

Link para o GitHub: https://github.com/raulgranja/malaria-boolean-network

Código desenvolvido e executado no Google Colab:

```
!pip install python-docx
!pip install matplotlib
!pip install dataframe_image

import numpy as np
from graphviz import Digraph
```

```
from itertools import product
import pandas as pd
from google.colab import drive
from docx import Document
import matplotlib.pyplot as plt
import dataframe_image as dfi
from IPython.display import Image
import random
from collections import Counter

def sort_row(row):
    # Extract the ID
    oligo_id = row['Oligo_ID']

    # Extract the name
    name = row['NAME']

    # Sort the numeric values starting from the third column
    sorted_values = sorted(row[2:])

    # Return a new row with 'Oligo_ID', 'NAME', and the sorted numeric values
    return [oligo_id, name] + sorted_values


def list_num(list_s):
    """
    Converts a binary list into the corresponding decimal number.
    Example: [0, 0, 0, 1] --> 1 * 2^0 = 1
             [1, 0, 0, 1] --> 1 * 2^3 + 0 + 0 + 1 * 2^0 = 9

    :param list_s: A list of binary digits (0s and 1s).
    :return: Decimal number (int).
    """
    # Use the built-in int function with base 2 to convert binary to decimal
    return int("".join(map(str, list_s)), 2)


def check_zero(state):
    """
    Identifies which indexes have a value of 1 in the given state and returns
them in a list.

    :param state: A list representing the state to analyze.
    :return: A list of indexes where the elements are equal to 1.

    Example:
```

```
    - check_zero([0, 1, 0, 1, 1, 1, 0, 0, 0]) returns [1, 3, 4, 5].
    """
    # Use list comprehension to get the indexes where the value is 1
    ones_index = [i for i, elem in enumerate(state) if elem == 1]

    return ones_index


def sum_values(state, attempt):
    """
    Sums the values from 'attempt' at the indices where 'state' has 1s.

    :param state: A list representing the state to analyze.
    :param attempt: A list with values corresponding to the indices in the
state.
    :return: Sum of the values in 'attempt' at the indices where 'state' has
1s.

    Example:
    - sum_values([0, 1, 0, 1, 1, 1, 0, 0, 0], [-1, -1, -1, 0, 1, 1, 0, 1, 0])
returns 1.
    """
    # Use sum with a list comprehension to add values of attempt at indices
from check_zero(state)
    return sum(attempt[i] for i in check_zero(state))


def smallest_lines(input_file, output_file, gap=0):
    """
    Filters the lines of a given file, keeping only those that have the
highest number of zeros within a given range.

    :param input_file: File to be filtered.
    :param output_file: The file where the filtered lines will be written.
    :param gap: The allowed range from the maximum number of zeros (default is
0).
    :return: output_file with the filtered lines.
    """
    # Read the input file and determine the maximum number of zeros
    with open(input_file, 'r') as file1:
        lines = file1.readlines()
        lm = max(line.count('0') for line in lines)

    # Filter and write lines with zeros count within the gap range
    with open(output_file, 'w') as file2:
        for line in lines:
```

```
            zeros = line.count('0')
            if zeros >= lm - gap:
                file2.write(line)

    return output_file


def remove_repeated_rows(np_array):
    """
    Removes repeated rows from a 2D NumPy array, retaining only the first
occurrence of each unique row
    while preserving the original order of the rows.

    :param np_array: A 2D NumPy array from which repeated rows will be
removed.
    :return: A new 2D NumPy array containing only the unique rows, in their
original order.
    """
    # Identify unique rows and their original indices
    unique_array, unique_indices = np.unique(np_array, axis=0,
return_index=True)

    # Sort the unique indices to maintain the original order of rows
    sorted_unique_indices = np.sort(unique_indices)

    # Extract the unique rows in their original order
    new_array = np_array[sorted_unique_indices]

    return new_array

def readFile(fileName):
    """
    Reads a file that specifies the possibilities of a line and returns a list
with integer values.

    :param fileName: The file that will be read.
    :return: List containing all the lines with integer values.
    """
    lines_new = []

    # Open the file using 'with' to ensure it's properly closed
    with open(fileName, "r") as file_obj:
        # Read and process each line
        for line in file_obj:
            line_new = line.strip()
            line_new = [int(i) for i in line_new.split()]
```

```
            lines_new.append(line_new)

    return lines_new

def random_reg(n):
    """
    Chooses a random regulation matrix based on the lines files.

    :param n: The number of lines of the matrix, or the number of functional
groups, or the number of files.
    :return: A random regulation matrix.
    """
    regulation = []

    for i in range(1, n + 1):
        # Read the file for the current functional group
        file_name = f'small_{i:02}.txt'
        try:
            lis = readFile(file_name)
            # Choose a random line from the list
            regulation.append(random.choice(lis))
        except FileNotFoundError:
            print(f"Warning: {file_name} not found.")
        except Exception as e:
            print(f"An error occurred while processing {file_name}: {e}")

    return regulation

def draw_trajectory (np_array):
    """
    Draws a directed graph using Graphviz where the vertices represent the
rows of a NumPy array,
    and the directed edges represent transitions from one row to the next in
sequence. The graph
    is saved as a PNG image and displayed in the notebook.

    :param np_array: A 2D NumPy array where each row represents a vertex, and
directed edges are created
                     between consecutive rows.
    :return: None. The function generates and displays a PNG image of the
graph.
    """
    dot = Digraph(comment='My Graph')

    for i in range(len(np_array) - 1):
      dot.node(str(np_array[i]))
```

```
        dot.node(str(np_array[i+1]))
        dot.edge(str(np_array[i]), str(np_array[i+1]))

    dot.render('my_graph', view=True, format='png')

    # Display the image in the notebook
    from IPython.display import Image
    Image(filename='my_graph.png')


def solutions_02(D):
    """
    Calculates and writes the valid regulatory patterns for each functional
group in a Boolean network based on
    the given state transitions matrix. The function iterates through all
possible regulatory patterns
    (-1, 0, 1) and checks their validity based on transitions between states.

    :param D: A 2D NumPy array representing the state transitions, where rows
are time points and
              columns represent functional groups. Each element is either 0 or
1, indicating functional group states.
    :return: The total number of valid regulatory matrices.
    """
    n_times, n_modules = D.shape
    characters = [-1, 0, 1]
    total = 1
    square = lambda x: x ** 2

    # Generate all possible combinations for regulatory patterns
    attempts = np.array(list(product(characters, repeat=n_modules)))

    for functional_group in range(n_modules):
        valid_regulations = []  # To store valid regulation attempts

        for attempt in attempts:
            # Skip attempts where all elements are zero (i.e., no regulation)
            if np.sum(attempt ** 2) == 0:
                continue

            # Count valid transitions
            valid = True
            for j in range(n_times - 1):
                current_state = D[j]
                next_state = D[j + 1]
```

```
                # Calculate h based on current state and regulation attempt
                h = sum_values(current_state, attempt)

                # Check if the regulation attempt satisfies conditions for
functional group
                if not (
                    (next_state[functional_group] == 0 and h < 0) or
                    (next_state[functional_group] == 1 and h > 0) or
                    (current_state[functional_group] ==
next_state[functional_group] and h == 0)
                ):
                    valid = False
                    break  # No need to check further if this attempt fails

            # If the attempt is valid for all transitions, store it
            if valid:
                valid_regulations.append(attempt)

        # Write valid regulations for the current functional group to a file
        filename = f'functional_group_{functional_group + 1:02}.txt'
        with open(filename, 'w') as f:
            for regulation in valid_regulations:
                f.write(' '.join(map(str, regulation)) + '\n')

        # Output functional group information and update total count
        num_solutions = len(valid_regulations)
        print(f'Functional Group {functional_group + 1:02} has {num_solutions}
possible regulation patterns')
        total *= num_solutions

    print(f'Total possible matrices = {total:e}')
    return total


def next_state(actual_state, regulation):
    """
    Calculate the next functional group state based on the current state and a
regulation matrix.

    :param actual_state: The current state of functional groups (list or array
of 0s and 1s).
    :param regulation: A regulation matrix where each row corresponds to how
the state of one functional group is regulated by others.
    :return: The next functional group state as a list of 0s and 1s.
    """
    s_old = np.array(actual_state)
```

```
    s_new = []

    for row in regulation:
        sm = np.dot(s_old, row)
        if sm > 0:
            s_new.append(1)
        elif sm < 0:
            s_new.append(0)
        else:
            s_new.append(s_old[len(s_new)])

    return s_new


def num_list(int_s):
    """
    Converts a decimal number to binary, separates all digits, and stores them
in a list.

    :param int_s: Any integer number from 0 to 2047 (inclusive).
    :return: A binary list of eleven digits corresponding to the binary
representation of the number.

    Example:
    - Converts 2 into [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0].
    - Converts 2047 into [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1].
    """
    str_s = f'{int_s:012b}'

    list_s = [int(s) for s in str_s]

    return list_s


def next_state_dec(s_dec, R):
    """
    Computes the next state of a functional group regulation system given the
current state in decimal form.

    :param s_dec: The current state of the functional group in decimal format.
    :param R: The regulation matrix used to determine the next state.
    :return: The next state of the functional group in decimal format.
    """
    s_bin = num_list(s_dec)

    next_state_bin = next_state(s_bin, R)
```

```
    n_state_dec = list_num(next_state_bin)

    return n_state_dec

def generate_STG(R, n=12):
    """
    Generates a State Transition Graph (STG) for a functional group regulation
system.

    :param R: The regulation matrix used to determine state transitions.
    :param n: The number of functional groups (or length of each state) in the
system. Default is 7.
    :return: A list representing the State Transition Graph, where each index
corresponds to a state,
             and the value at that index is the next state in decimal form.
    """
    if n <= 0:
        raise ValueError("The number of functional groups (n) must be a
positive integer.")

    g = [0] * (2 ** n)

    for s in range(2 ** n):
        g[s] = next_state_dec(s, R)

    return g


def draw_STG(M, n=12, STG=None, engine='sfdp', bin_state=False):
    """
    Draws a State Transition Graph (STG) based on a transition matrix and
state configurations.

    :param M: The transition matrix that defines state transitions.
    :param n: The number of states to consider (default is 12).
    :param STG: A predefined state transition graph (optional). If not
provided, it will be generated from M.
    :param engine: The layout engine to use for the graph visualization
(default is 'sfdp').
    :param bin_state: Boolean flag to display states in binary (True) or
decimal (False) format (default is False).
    :return: A Digraph object representing the state transition graph.
    """
    if STG is None:
        STG = generate_STG(M, n)
```

```
    g = Digraph('G', engine=engine)
    g.attr(size='16,10')

    for ini_state, fim_state in enumerate(STG):
        if bin_state:
            ini_str = ''.join(map(str, num_list(ini_state)))
            fim_str = ''.join(map(str, num_list(fim_state)))
        else:
            ini_str = str(ini_state)
            fim_str = str(fim_state)

        g.edge(ini_str, fim_str)

    return g


def component_sizes(STG):
    """
    Calculates the size of each connected component in a State Transition
Graph (STG).

    Args:
        STG: A list representing the State Transition Graph, where each index
corresponds to a state,
             and the value at that index is the next state in decimal form.

    Returns:
        component_size_counts: A dictionary where keys are component IDs and
values are the sizes of each component.
        total_components: The total number of distinct connected components in
the graph.
        state_to_component_map: A list where each index represents a state,
and the value is the component ID
                                to which the state belongs.
    """
    # Number of states in the STG
    num_states = len(STG)

    # List to track whether a state has been visited
    visited_states = [False] * num_states

    # List to store the component ID for each state
    state_component_ids = [-1] * num_states

    # Initialize the component counter (component ID)
```

```
    current_component_id = 0

    # Loop through each state to explore its connected component
    for state in range(num_states):
        if not visited_states[state]:
            # Start a queue to perform BFS and track the component trajectory
            bfs_queue = [state]

            # Mark the current state as visited and assign it a component ID
            visited_states[state] = True
            state_component_ids[state] = current_component_id

            # Store the current trajectory of states in this component
            current_trajectory = []

            # BFS to explore all states in the connected component
            while bfs_queue:
                current_state = bfs_queue.pop(0)  # Get the current state from
the queue
                current_trajectory.append(current_state)

                # Get the next state according to the transition graph
                next_state = STG[current_state]

                # If the next state has not been visited, add it to the queue
and mark it as visited
                if not visited_states[next_state]:
                    visited_states[next_state] = True
                    bfs_queue.append(next_state)
                    state_component_ids[next_state] = current_component_id
                else:
                    # If we reach an already visited state, update all states
in the trajectory
                    for trajectory_state in current_trajectory:
                        state_component_ids[trajectory_state] =
state_component_ids[next_state]
                    break

            # If the BFS queue is empty, increment the component ID for the
next group of states
            if not bfs_queue:
                current_component_id += 1

    # Get the unique component IDs and map them to a continuous range
    unique_component_ids = sorted(set(state_component_ids))
```

```
    component_id_mapping = {old_id: new_id for new_id, old_id in
enumerate(unique_component_ids)}

    # Remap the component IDs to the continuous range
    state_to_component_map = [component_id_mapping[id] for id in
state_component_ids]

    # Count the number of states in each component
    component_size_counts = Counter(state_to_component_map)

    return component_size_counts, len(unique_component_ids),
state_to_component_map


def generate_random_regulation(num_functional_groups, seed_value=10):
    """
    Generates a random regulation matrix for a given number of functional
groups.

    Args:
        num_functional_groups (int): The number of functional groups for which
to generate the regulation matrix.
        seed_value (int, optional): The seed for random number generation to
ensure reproducibility. Defaults to 10.

    Returns:
        list: A matrix representing the functional group regulation rules
where each row indicates how the state of one functional group is influenced
by others.
    """
    random.seed(seed_value)
    regulation_matrix = random_reg(num_functional_groups)
    return regulation_matrix


def generate_and_analyze_STG(num_functional_groups=12, seed_value=10):
    """
    Generates a State Transition Graph (STG) for a given number of functional
groups, analyzes its connected components,
    and prints the number of distinct connected components and their size
frequencies.

    Args:
        num_functional_groups (int, optional): The number of functional groups
in the regulation matrix. Defaults to 12.
```

```
        seed_value (int, optional): The seed for random number generation to
ensure reproducibility. Defaults to 10.

    Returns:
        None: The function prints the results directly.
    """
    regulation_matrix = generate_random_regulation(num_functional_groups,
seed_value)
    STG = generate_STG(regulation_matrix, n=num_functional_groups)
    frequency_counts, num_distinct_components, mapped_component_ids =
component_sizes(STG)
    print(f"Number of distinct connected components:
{num_distinct_components}")
    print("Component size frequencies:", frequency_counts)
    return frequency_counts


def plot_component_frequencies(frequency_counts):
    """
    Plots a bar chart representing the frequency of each connected component
in a State Transition Graph (STG),
    excluding components with zero frequency. Each bar shows the number of
states within each component.

    Args:
        frequency_counts (dict): A dictionary where keys are component IDs and
values are the corresponding frequencies.

    Returns:
        None: Displays a bar chart of the component sizes.
    """
    filtered_counts = {id: count for id, count in frequency_counts.items() if
count > 0}
    ids = sorted(filtered_counts.keys())
    frequencies = [filtered_counts[id] for id in ids]

    plt.figure(figsize=(10, 6))
    bars = plt.bar(ids, frequencies, color='skyblue')

    plt.xlabel("Component ID", fontsize=12)
    plt.ylabel("Number of States", fontsize=12)
    plt.title("Number of States in each Connected Component", fontsize=14)

    for bar in bars:
        yval = bar.get_height()
```

```
        plt.text(bar.get_x() + bar.get_width()/2, yval, int(yval),
ha='center', va='bottom', fontsize=10)

    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.tight_layout()
    plt.show()


def draw_tbn(adjacency_matrix, module_names):
    """
    Draws a thresholded Boolean network (TBN) based on the provided adjacency
matrix.

    Args:
        adjacency_matrix (list of list of float): A square matrix representing
the transitions between modules.
        module_names (list of str): A list containing names for each module.

    Returns:
        Digraph: A Graphviz Digraph object representing the TBN.
    """
    graph = Digraph('G', node_attr={'style': 'filled', 'fontname':
'Helvetica', 'fontsize': '12'})
    graph.attr(size='10,7', layout='dot')  # Set layout and size for the graph
    graph.attr(rankdir='TB')  # Top to bottom layout

    num_modules = len(adjacency_matrix)

    # Define colors for positive and negative edges
    positive_color = 'lightblue'
    negative_color = 'salmon'

    # Create nodes with customized colors
    for i, name in enumerate(module_names):
        graph.node(str(name), color=positive_color, shape='ellipse',
fontcolor='black', style='filled')

    for i in range(num_modules):
        for j in range(num_modules):
            if adjacency_matrix[i][j] < 0:
                graph.edge(str(module_names[j]), str(module_names[i]),
arrowhead="tee", color=negative_color, penwidth='2')
            elif adjacency_matrix[i][j] > 0:
                graph.edge(str(module_names[j]), str(module_names[i]),
color=positive_color, penwidth='2')
```

```
    return graph

drive.mount('/content/drive')

file_path = 'dados_malaria.csv'

df_original = pd.read_csv(file_path, delimiter = '\t')

cols_to_convert = df_original.columns[2:]

df_original[cols_to_convert] =
df_original[cols_to_convert].apply(pd.to_numeric, errors='coerce')

df_original_dropped = pd.concat([df_original[df_original.columns[:2]],
df_original[cols_to_convert]],
axis=1).dropna(how='any').reset_index(drop=True)

sorted_data = df_original_dropped.apply(sort_row, axis=1).tolist()

columns = df_original_dropped.columns.tolist()

df_sorted = pd.DataFrame(sorted_data, columns=columns)

new_columns_names = {f'TP {i}': f'{i}' for i in range(1, 49)}

df_sorted.rename(columns=new_columns_names, inplace=True)

fixed_columns = df_sorted.iloc[:, :2]

columns_to_diff = df_sorted.iloc[:, 2:]

columns_diff = columns_to_diff.diff(axis=1)

df_diff = pd.concat([fixed_columns, columns_diff], axis=1)

df_diff = df_diff.drop('1', axis=1)

new_columns_name = {f'{i}': f'{i - 1}' for i in range(0, 47)}

df_diff.rename(columns=new_columns_name, inplace=True)

df_sorted_str = df_sorted.iloc[:, :2]
df_sorted_num = df_sorted.iloc[:, 2:]

diff = df_sorted_num.iloc[:, -1] - df_sorted_num.iloc[:, 0]
```

```
result = diff / (df_sorted_num.shape[1] - 1)

df_t = df_sorted_str.copy()
df_t['t'] = result

result_data = []

for _, row2 in df_t.iterrows():
    oligo_id = row2['Oligo_ID']
    name_val = row2['NAME']
    t_value = row2['t']

    row1 = df_diff[(df_diff['Oligo_ID'] == oligo_id) & (df_diff['NAME'] ==
name_val)]

    min_index = None
    for col in df_diff.columns[2:]:
        if row1[col].values[0] > t_value:
            min_index = col
            break

    result_data.append([oligo_id, name_val, min_index])

df_min = pd.DataFrame(result_data, columns=['Oligo_ID', 'NAME', 'MIN_INDEX'])
df_min['MIN_INDEX'] = pd.to_numeric(df_min['MIN_INDEX'],
errors='coerce').astype(pd.Int64Dtype())

new_columns_names = {f'{i}': f'TP {i}' for i in range(1, 49)}
df_sorted.rename(columns=new_columns_names, inplace=True)

df_B = pd.concat([df_original_dropped.iloc[:, :2], pd.DataFrame(0,
index=df_original_dropped.index, columns=df_original_dropped.columns[2:],
dtype=int)], axis=1)

for i in range(len(df_original_dropped)):
    m = df_min.loc[i, 'MIN_INDEX']

    for j in df_original_dropped.columns[2:]:
        m_plus_1 = m + 1

        if (m_plus_1 == 23) or (m_plus_1 == 29):
            m_plus_1 = m + 2
        elif m_plus_1 == 49:
            m_plus_1 = 48

        col = 'TP ' + str(m_plus_1)
```

```
        if df_original_dropped.loc[i, j] >= df_sorted.loc[i, col]:
            df_B.loc[i, j] = 1
        else:
            df_B.loc[i, j] = 0

file_path = 'functional_groups_pandas.xlsx'

df_functional_groups = pd.read_excel(file_path)

functional_groups_first_col = df_functional_groups.iloc[:, 0:1]
df_functional_groups_numbers = df_functional_groups.iloc[:, 4:]

df_functional_groups_averages = pd.concat([functional_groups_first_col,
df_functional_groups_numbers], axis=1).groupby('functional_group',
sort=False).mean().reset_index()

df_functional_groups_averages_transposed =
df_functional_groups_averages.transpose()
df_functional_groups_averages_transposed.columns =
df_functional_groups_averages_transposed.iloc[0]
df_functional_groups_averages_transposed =
df_functional_groups_averages_transposed.drop(df_functional_groups_averages_tr
ansposed.index[0])
df_functional_groups_averages_transposed =
df_functional_groups_averages_transposed.iloc[:, :-2]

df_binarized = df_functional_groups_averages_transposed.copy()
column_means = df_binarized.mean()

for column in df_binarized.columns:
    df_binarized[column] = (df_binarized[column] >=
column_means[column]).astype(int)

df_binarized_np = df_binarized.to_numpy()

# Create a copy of the original array to avoid modifying the original
new_bin_array = df_binarized_np.copy()

# Update specific elements in the array to zero
new_bin_array[5, 3] = 0   # Set element at row 5, column 3 to 0
new_bin_array[3, 3] = 0   # Set element at row 3, column 3 to 0
new_bin_array[3, 10] = 0  # Set element at row 3, column 10 to 0
new_bin_array[12, 7] = 0  # Set element at row 12, column 7 to 0
new_bin_array[18, 7] = 0  # Set element at row 18, column 7 to 0
```

```
cleaned_bin_array = remove_repeated_rows(new_bin_array)

# Appending the first row of 'new_array' to the end of 'new_array'
appended_array = np.append(cleaned_bin_array, [cleaned_bin_array[0]], axis=0)

trajetoria = [list_num(list(row)) for row in appended_array]

draw_trajectory(appended_array)
Image(filename='my_graph.png')

solutions_02 (appended_array)

n_times, n_modules = appended_array.shape

for gene in range(n_modules):
    input_file = f'functional_group_{gene + 1:02}.txt'
    output_file = f'small_{gene + 1:02}.txt'

    # Print the filenames for debugging purposes
    print(f'Processing input file: {input_file}, output file: {output_file}')
    smallest_lines(input_file, output_file, gap=0)

n_times, n_modules = appended_array.shape
number_matrices = 1  # Initialize the product for counting matrices

for gene in range(n_modules):
    input_file = f'small_{gene + 1:02}.txt'

    # Efficiently count the number of lines without loading the entire file
into memory
    with open(input_file, 'r') as file:
        line_count = sum(1 for _ in file)

    print(f'{input_file} has {line_count:02} rows')
    number_matrices *= line_count  # Multiply to keep track of total possible
matrices

print(f'Total possible matrices: {number_matrices}')

frequency_counts = generate_and_analyze_STG(num_functional_groups=12,
seed_value=10)

R = generate_random_regulation(12, seed_value=10)
draw_STG(R)

plot_component_frequencies(frequency_counts)
```

```
STG = generate_STG(R, n=12)
frequency_counts, num_distinct_ids, mapped_component_ids =
component_sizes(STG)

# Convert each row of appended_array from binary to decimal
decimal_values = [int(''.join(map(str, row)), 2) for row in appended_array]

random.seed(10)

# Initialize counters and lists to store results
attempt_count = 0
basin_counts = []
malaria_basin_sizes = []
min_distinct_ids = 4096

while attempt_count < 50:
    regulation_matrix = random_reg(12)
    state_transition_graph = generate_STG(regulation_matrix, n=12)

    # Calculate component sizes from the State Transition Graph
    frequency_counts, num_distinct_ids, mapped_component_ids =
component_sizes(state_transition_graph)

    # Get the size of the basin of attraction containing the malaria sequence
    basin_size = frequency_counts[mapped_component_ids[decimal_values[0]]]

    basin_counts.append(num_distinct_ids)
    malaria_basin_sizes.append(basin_size)

    # Update the minimum distinct IDs and associated basin size if necessary
    if num_distinct_ids < min_distinct_ids:
        min_distinct_ids = num_distinct_ids
        current_basin_size = basin_size
        best_regulation_matrix = regulation_matrix

    print(attempt_count)
    attempt_count += 1

# Create the scatter plot
plt.figure(figsize=(10, 6))
plt.scatter(basin_counts, malaria_basin_sizes, color='skyblue',
edgecolor='black', alpha=0.7)

# Add labels and title
plt.xlabel("Number of Basins of Attraction", fontsize=12)
```

```
plt.ylabel("Size of Basin of Attraction Containing Malaria Sequence",
fontsize=12)
plt.title("Scatter Plot of Number of Basins vs. Basin Size", fontsize=14)

# Add a grid
plt.grid(True, alpha=0.7)

# Display the plot
plt.tight_layout()
plt.show()

plot_component_frequencies(frequency_counts)

best_regulation_matrix

print(f"This network has {min_distinct_ids} basins of attraction.")
print(f"The size of the basin containing the malaria cycle is:
{current_basin_size}.")

module_names = df_functional_groups_averages_transposed.columns.tolist()
tbn_graph = draw_tbn(best_regulation_matrix, module_names)
tbn_graph.render('tbn_graph', format='png', cleanup=True)  # Save the graph as
a PNG file

tbn_graph
```