Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia

COPPE UFRJ

O PROBLEMA SNAKE-IN-THE-BOX: ORIGEM, DESAFIOS E TÉCNICAS DE SOLUÇÃO

Girolamo Santoro

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Fábio Happ Botler
Laura Silvia B. da Silva Leite

Rio de Janeiro
Setembro de 2024

O PROBLEMA SNAKE-IN-THE-BOX:
ORIGEM, DESAFIOS E TÉCNICAS DE SOLUÇÃO

Girolamo Santoro

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Orientadores: Fábio Happ Botler
Laura Silvia B. da Silva Leite

Aprovada por: Prof. Laura Silvia B. da Silva Leite
Prof. Fábio Happ Botler
Prof. Carla Negri Lintzmayer
Prof. Daniel Ratton Figueiredo

RIO DE JANEIRO, RJ – BRASIL
SETEMBRO DE 2024

Santoro, Girolamo

O Problema Snake-In-the-Box:

Origem, Desafios e Técnicas de Solução/Girolamo Santoro. – Rio de Janeiro: UFRJ/COPPE, 2024.

XII, 69 p.: il.; 29, 7cm.

Orientadores: Fábio Happ Botler

Laura Silvia B. da Silva Leite

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2024.

Referências Bibliográficas: p. 65 – 69.

1. Snake-In-the-Box. 2. Beam Search. 3. Hipercubos. I. Happ Botler, Fábio *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

"A nossa maior fraqueza está em desistir. O caminho mais certo de vencer é tentar mais uma vez." (Thomas Edison)

Agradecimentos

Dedico este trabalho às três pessoas mais importantes da minha vida, que me incentivaram e possibilitaram que eu chegasse até aqui: meu querido e falecido pai Giuseppe e minha mãe Filomena, que trabalharam incansavelmente para me proporcionar esta oportunidade; e minha esposa Maria Antonieta, que, apesar das longas horas de estudo necessárias, que me privaram de sua companhia, sempre me incentivou para a conclusão de meu mestrado. Espero que este trabalho sirva de inspiração para meus três filhos e seis netos, demonstrando que o estudo e aperfeiçoamento são indispensáveis ao longo de toda a vida, independentemente da idade. Gostaria de expressar meus sinceros agradecimentos a todos os professores do mestrado, que com seu conhecimento e dedicação, contribuíram significativamente para a minha formação acadêmica e pessoal. Agradeço, em especial, ao Prof. Fábio Botler, por sua dedicação e orientação criteriosa, como também pelo apoio constante ao longo deste processo. Sua expertise e paciência foram fundamentais para a realização deste trabalho, além de despertar meu interesse pelo tema da dissertação. Agradeço também à Prof. Celina Miraglia Herrera de Figueiredo, cujas aulas proporcionaram uma base sólida em algoritmos e suas complexidades, incentivando-me nas matérias fundamentais e fortalecendo meu entendimento dos conceitos essenciais necessários ao desenvolvimento deste estudo.

A todos, meu profundo reconhecimento e gratidão.

“DEO GRATIAS”

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

O PROBLEMA SNAKE-IN-THE-BOX:
ORIGEM, DESAFIOS E TÉCNICAS DE SOLUÇÃO

Girolamo Santoro

Setembro/2024

Orientadores: Fábio Happ Botler
Laura Silvia B. da Silva Leite

Programa: Engenharia de Sistemas e Computação

Este trabalho tem como objetivo apresentar o problema *Snake-in-the-Box*, abreviadamente *problema SIB*, que consiste na busca pelos maiores caminhos induzidos abertos (resp. fechados), comumente denominados *snakes* (resp. *coils*), em grafos da família dos hipercubos. A abordagem visa tornar o tema acessível tanto para iniciantes quanto para especialistas, utilizando uma linguagem clara e explicando os conceitos e pré-requisitos necessários. São introduzidos os fundamentos do problema SIB, incluindo sua origem em 1958, sua evolução, principais aplicações práticas e os métodos para representar, validar e buscar *snakes* em hipercubos de dimensão d.

Inicialmente, são apresentados limitantes inferiores e superiores para cada dimensão dos hipercubos, com base em métodos matemáticos e conceitos da teoria dos grafos. Em seguida, discutem-se os algoritmos de busca que foram adaptados, aperfeiçoados e implementados em programas computacionais, permitindo a identificação de *snakes* cada vez maiores. Alguns desses caminhos são comprovadamente máximos absolutos, enquanto outros representam os maiores conhecidos até o momento.

O trabalho também analisa uma ampla base de publicações, incluindo artigos técnicos, dissertações e teses divulgados desde 1958. Esses estudos são examinados para abordar tanto os fundamentos matemáticos que definem tanto os limitantes quanto os diferentes algoritmos empregados para explorar o problema SIB.

Adicionalmente, foi implementado um algoritmo customizado do tipo *Stochastic Beam Search*, um dos vários métodos de busca utilizados por pesquisadores para obter as maiores snakes. Essa implementação oferece aos leitores uma abordagem prática que lhes permite avaliar a complexidade envolvida no problema SIB.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

THE SNAKE-IN-THE-BOX PROBLEM: ORIGIN, CHALLENGES AND SOLUTION TECHNIQUES

Girolamo Santoro

September/2024

Advisors: Fábio Happ Botler
Laura Silvia B. da Silva Leite

Department: Systems Engineering and Computer Science

This work aims to present the *Snake-in-the-Box* problem, abbreviated as *SIB problem*, which consists of searching for the longest induced open (resp. closed) paths, commonly referred to as *snakes* (resp. *coils*), in graphs from the family of the hypercubes. The approach aims to make the topic accessible to both beginners and experts, using clear language and explaining the necessary concepts and prerequisites. The fundamentals of the SIB problem are introduced, including its origin in 1958, its evolution, main practical applications, and methods for representing, validating, and searching for *snakes* in hypercubes of dimension d.

Initially, lower and upper bounds for each dimension of hypercubes are presented, based on mathematical methods and concepts from graph theory. Next, the search algorithms that have been adapted, refined, and implemented in computer programs are discussed, enabling the identification of increasingly larger *snakes*. Some of these paths are proven to be absolute maxima, while others represent the longest known paths to date.

The work also analyzes a broad range of publications, including technical papers, dissertations, and theses published since 1958. These studies are examined to address both the mathematical foundations that define the bounds and the different algorithms used to explore the SIB problem.

Additionally, a custom *Stochastic Beam Search* algorithm was implemented, one of the various search methods used by researchers to find the longest *snakes*. This implementation provides readers with a practical approach, enabling them to assess the complexity involved in the SIB problem.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Esta dissertação é um guia introdutório a respeito do problema *Snake-In-the-Box*, abreviadamente chamado problema *SIB*. O objetivo deste trabalho é descrever o problema *SIB* e apresentar os resultados alcançados até o momento na literatura, detalhando de forma concisa cada um dos resultados e os métodos empregados para sua obtenção. Além disso, apresentamos uma descrição minuciosa da implementação customizada de um de tais métodos, de forma a fornecer uma base para futuras pesquisas.

1.1 Conceitos Básicos em Grafos

Um *grafo* G é um par (V, E), em que V é um conjunto finito, cujos elementos são chamados de *vértices*, e E é um conjunto de pares (não ordenados) de vértices de V, chamados *arestas*. Como cada aresta $e \in E$ é um par de vértices $e = \{u, v\}$, para evitar excesso de notação, frequentemente a representamos apenas pelos dois vértices, i.e., por uv. Nesse caso, os vértices u e v são chamados de *extremidades* da aresta e, e também dizemos que u e v são *adjacentes*; e que a aresta e é *incidente* a u e v. Também dizemos que duas arestas são *adjacentes* se possuem uma extremidade em comum. O *grau* de um vértice u de G é a quantidade de arestas incidentes a u. Quando todos os vértices de G têm o mesmo grau k, dizemos que G é *k-regular*.

Dado um grafo G, um *caminho* P em G é um subgrafo cujos vértices admitem uma ordenação $v_0, v_1, \ldots, v_s$ tal que $E(P) = \{v_{i-1}v_i : i \in \{1, \ldots, s\}\}$. Um *ciclo* C em G é um subgrafo cujos vértices admitem uma ordenação $v_0, v_1, \ldots, v_s$ tal que $E(C) = \{v_{i-1}v_i : i \in \{1, \ldots, s\}\} \cup \{v_0v_s\}$. Por simplicidade, frequentemente, identificamos um caminho apenas por sua sequência de vértices; e um ciclo pela sequência obtida da sua sequência de vértices pela adição de uma ocorrência de seu primeiro vértice ao final.

Dado um inteiro d, o *hipercubo de dimensão* d, denotado por Q_d, é o grafo cujos vértices são as sequências binárias de comprimento d, e no qual dois vértices são adjacentes se suas sequências diferem em apenas uma posição (veja Figura 1). Consequentemente, o número de vértices em Q_d é 2^d, e o número de arestas em Q_d é $d \cdot 2^{d-1}$, pois cada um de seus vértices é adjacente a precisamente d outros vértices. Assim, Q_d é um grafo d-regular. Por exemplo, em Q_3, o vértice 000 conecta-se aos vértices 001, 010 e 100 (veja Figura 2). Podemos também representar os vértices de Q_d pelos números decimais equivalentes a cada sequência, i.e., pelos números de 0 a $2^d - 1$. Nessa notação, 0 é adjacente a 1, 2, e 4.

Os hipercubos também são grafos *bipartidos*, i.e., grafos cujos conjuntos de vértices podem ser particionados em dois conjuntos, A e B, de modo que não existem arestas conectando dois vértices de A, nem arestas conectando dois vértices de B.

De fato, seja A (respectivamente, B) o conjunto de vértices cujas sequências possuem um número ímpar (respectivamente, par) de 1's.

Naturalmente, toda sequência associada a um vértice adjacente a um vértice de A deve possuir um número par de 1's e, portanto, pertence ao conjunto B.

Também podemos obter Q_d recursivamente como o produto cartesiano (veja [1]) do hipercubo Q_{d-1} e do grafo completo K_2. Isso é, $Q_0 = K_1$ é o grafo vazio com apenas um vértice, e $Q_d = Q_{d-1} \times K_2$ para $d \geq 1$.

Exemplo 1. O conjunto de vértices do cubo Q_2 de dimensão 2 é

$$V(Q_2) = V(Q_1 \times K_2) = \{00, 01, 10, 11\}$$

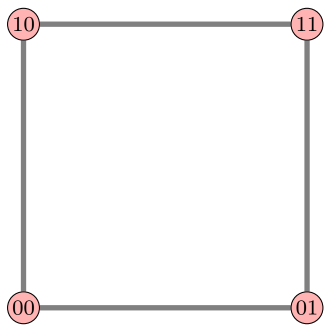


Figura 1: Cubo Q_2.

Exemplo 2. O conjunto de vértices do cubo Q_3 de dimensão 3 é

$$V(Q_3) = V(Q_2 \times K_2) = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

Os hipercubos possuem propriedades interessantes e são objeto de estudo em teoria dos grafos, matemática discreta e ciência da computação, redes de computadores, e em outras áreas, devido à sua simetria e capacidade de expansão para dimensões superiores [2–5]. Na Figura 3 mostramos uma representação em 2D, dos hipercubos, nas dimensões 2, 3 e 4.

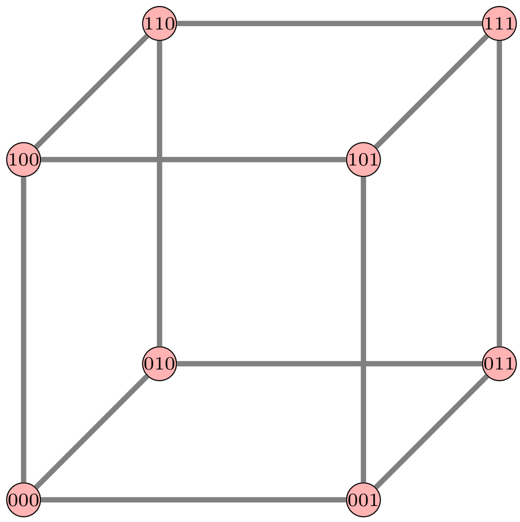
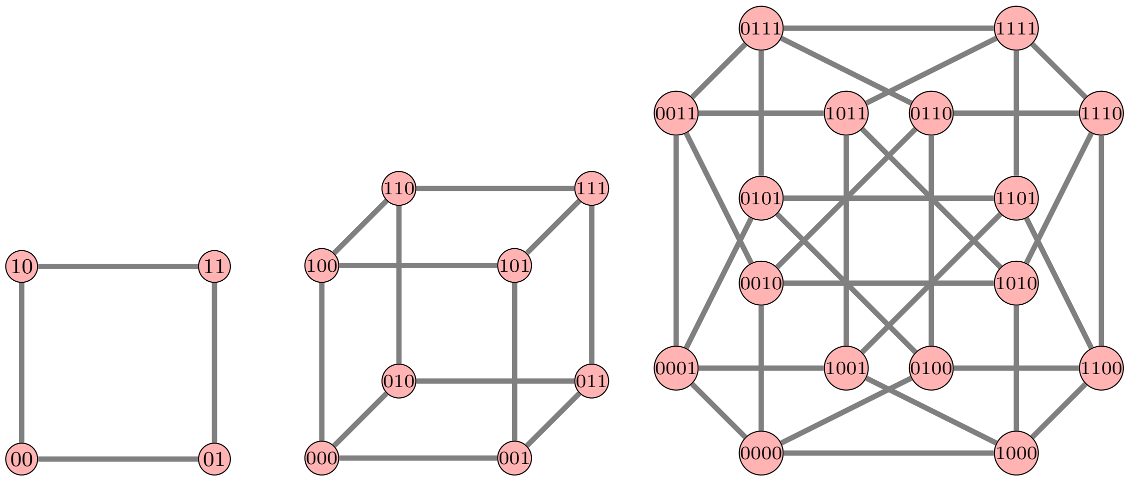


Figura 2: Cubo Q_3.



(a) Quadrado (b) Cubo (c) Q_4 Tesseract

Figura 3: Cubos de dimensões 2, 3 e 4

1.2 Caminhos e ciclos induzidos

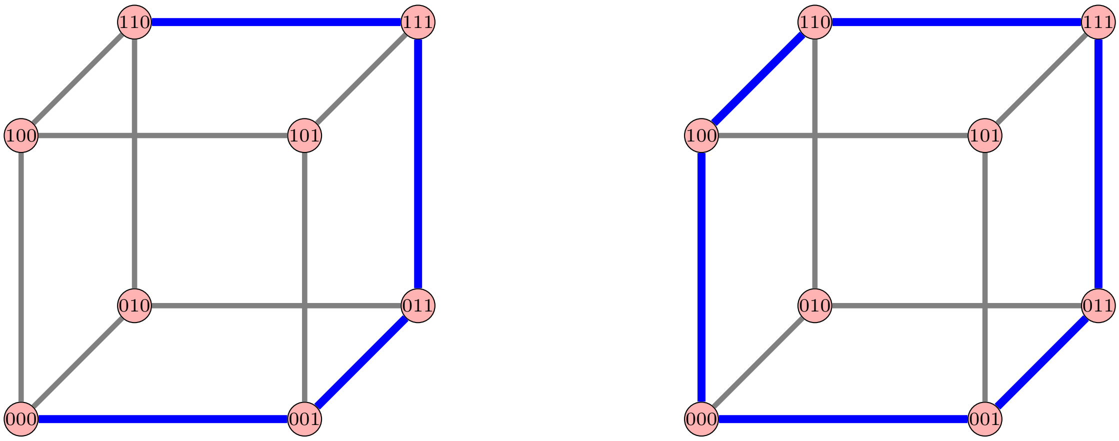
Seja X um caminho ou um ciclo em um grafo G. Uma *corda* de X é uma aresta de G que não pertence a X e que liga dois vértices de X. Em outras palavras, se $X = v_1 v_2 \cdots v_k$ é um caminho, então uma corda de X é qualquer aresta $v_i v_j$ de G com $|i - j| \geq 2$. Dizemos então que X é um *caminho induzido* se G não possui cordas de X. Analogamente, se $X = v_1 v_2 \cdots v_k v_1$ é um ciclo, então, uma corda de X é qualquer aresta $v_i v_j$ de G com $|i - j| \geq 2$ e $\{i, j\} \neq \{1, k\}$, e dizemos que X é um *ciclo induzido* se G não possui cordas de X. Portanto, se X é um caminho ou ciclo induzido, as únicas arestas de G que ligam dois vértices de X são aquelas que ligam vértices consecutivos de X.

Por exemplo, seja G o grafo com vértices $V = \{a, b, c, d\}$ e arestas $E = \{ab, bc, cd, ad\}$. O caminho $P = abc$ é um caminho induzido de G, pois G não possui aresta ligando a a c. Por outro lado, o caminho $P' = abcd$ não é um caminho induzido de G, pois a aresta ad é uma corda de P.

Em termos de complexidade computacional, o problema de determinar os caminhos ou ciclos induzidos mais longos em grafos é um problema desafiador, e é um problema NP-completo mesmo em grafos bipartidos [6]. Portanto, não é conhecido algoritmo polinomial para resolvê-lo. Desta forma, é interessante estudar a sua restrição a classes específicas de grafos [7–9]. Neste trabalho estudamos o problema de determinar caminhos e ciclos induzidos mais longos na família de grafos hipercubo.

1.3 O problema *Snake-In-the-Box*

Seja d um inteiro positivo. Uma *snake* é um caminho induzido em Q_d, e um *coil* é um ciclo induzido em Q_d (veja Figura 4). O problema de encontrar as snakes e coils mais longos em Q_d é conhecido como o *problema Snake-In-the-Box*, ou abreviadamente, o *problema SIB*, e foi primeiramente estudado por Kautz [10] em 1958. Alguns autores diferenciam o problema de encontrar a maior snake e o maior coil em Q_d e chamam o problema de encontrar o maior coil de o *problema Coil-In-the-Box*, abreviadamente CIB. Observe que, como Q_d é um grafo bipartido, todos os ciclos em Q_d têm comprimento par.



(a) Snake em Q_3 (b) Coil em Q_3

Figura 4: Snake e Coil em Q_3.

Lembre-se que os vértices em Q_d são representados por sequências binárias de comprimento d. Dados dois vértices u e v em Q_d, a *distância de Hamming* [11] entre u e v é o número de posições em que as suas sequências binárias diferem, e é denotada por $dist_{Q_d}(u, v)$. Por exemplo, $dist(100, 001) = 2$.

Neste trabalho, dado um inteiro d, definimos o *espaço de busca* $\mathcal{S}(Q_d)$ como o conjunto de todos os subconjuntos de $V(Q_d)$ que formam um caminho induzido (veja Seção 1.2). Cada caminho S começa em um vértice $v_0 \in V(Q_d)$ e é seguido por uma sequência de vértices $v_1, v_2, v_3, \ldots, v_n$, com $n \geq 1$. Formalmente, temos:

$$\mathcal{S}(Q_d) = \{\{v_0, v_1, v_2, \ldots, v_n\} \subset V(Q_d) : n \geq 1, \text{ e } S \text{ é um caminho induzido em } Q_d\}$$

Amplitude de uma snake ou de um coil

Dizemos que uma snake ou coil X tem *amplitude* k se para qualquer par de vértices u e v com distância pelo menos k em X temos $dist_{Q_d}(u, v) \geq k$ (veja Figura 5). Uma snake (resp. um coil) de amplitude k é também chamada de k-*snake* (resp. k-*coil*). Note que toda k-snake é uma $(k-1)$-snake, e todo k-coil é um $(k-1)$-coil. Por exemplo, em uma 3-snake $v_0 \cdots v_t$ de tamanho t, as distâncias (em Q_d) entre quaisquer dois vértices v_i e v_j com $j - i \geq 3$ é sempre pelo menos 3. Assim, uma generalização natural do problema SIB é, dados k e d, encontrar a maior k-snake (resp. o maior k-coil) em Q_d.

Exemplo 3. Considere a 3-snake $S = 0000, 0001, 0011, 0111, 1111, 1110$ no hipercubo de dimensão 4 (veja Figura 5). Para verificar que tal sequência de fato é uma 3-snake devemos checar se todo par de vértices cuja distância em S é pelo menos 3 está a uma distância pelo menos 3 em Q_d, i.e., se a sua distância de Hamming é pelo menos 3. Tais distâncias são apresentados na Tabela 1. Observe, em particular, que o caminho que seria obtido a partir de S pela adição do vértice 1100 ao final formaria uma 2-snake, porém não seria uma 3-snake pois $dist_{Q_4}(0000, 1100) = 2$.

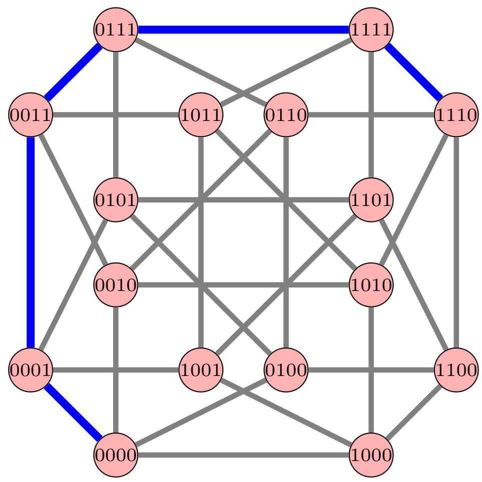


Figura 5: Uma 3-Snake em Q_4.

Apesar dessa potencial generalidade da amplitude das snakes e dos coils, neste trabalho focamos em snakes com amplitude 2.

Tabela 1: Distâncias de Hamming dos vértices da 3-snake
$S = 0000\ 0001\ 0011\ 0111\ 1111\ 1110$ cuja distância em S é pelo menos 3.

Vértices u, v	Distância de u, v em S	Distância de u, v em Q_4
(0000, 0111)	3	3
(0000, 1111)	4	4
(0000, 1110)	5	3
(0001, 1111)	3	4
(0001, 1110)	4	3
(0011, 1110)	3	4

1.4 Um breve histórico

Três pesquisadores foram fundamentais para o surgimento do problema *SIB*: Gray [12], Hamming [11] e Kautz [10]. Esses pesquisadores desenvolveram os conceitos essenciais que deram origem ao problema *SIB*. Para uma melhor compreensão do problema, descreveremos a seguir como suas contribuições estão interligadas. Além disso, faremos um breve relato sobre os desenvolvimentos e estudos paralelos conduzidos na União Soviética.

Gray e o Código de Gray

Criado por Frank Gray, físico e engenheiro da Bell Labs, o Código de Gray foi desenvolvido na década de 1940 e teve sua patente registrada em 1953 [12]. Amplamente utilizado em eletrônica e telecomunicações, ele foi inicialmente empregado em formas primitivas de codificadores que dependiam de interruptores mecânicos e relés[1].

O código de Gray é um sistema de codificação binária em que, diferentemente da codificação binária usual, apenas um bit varia entre números consecutivos, garantindo uma transição suave entre valores. Essa estrutura oferece uma representação precisa e confiável de dados binários, reduzindo significativamente os erros em processos de leitura e transmissão e minimizando a chance de interpretações incorretas nas mudanças de estado, que são comuns em sistemas de medição e comunicação. A principal vantagem do Código de Gray é a redução de erros durante as transições de valores, o que é essencial para displays digitais, encoders rotativos e sistemas digitais, onde mudanças bruscas podem gerar falhas. Em comunicações digitais, o Código de Gray é aplicado para aumentar a confiabilidade na transmissão, permitindo inclusive a detecção e, em alguns casos, a correção de erros.

Em particular, alternar entre a codificação binária usual e o Código de Gray é uma tarefa relativamente simples. Para converter um número binário em Código de Gray, aplica-se uma operação XOR entre cada bit e o bit anterior. Já a conversão

[1] Interruptores eletromecânicos criados por Faraday

inversa também utiliza operações XOR, mas de forma sequencial, permitindo a recuperação do valor binário original e garantindo que o sistema permaneça simples e eficiente para diferentes aplicações (veja, e.g., [13]).

Hamming e os Códigos de Hamming

Richard W. Hamming [11], também atuando nos Bell Labs, desenvolveu os códigos que levam seu nome, os *Códigos de Hamming*, em 1950, enquanto investigava métodos de correção de erros introduzidos por máquinas leitoras de cartões perfurados.

Na ciência da computação e telecomunicações, os Códigos de Hamming constituem uma família de códigos de correção de erros. A distância entre duas palavras-código é definida como o número de bits que precisam ser alterados para transformar uma palavra na outra, sendo essa métrica conhecida como *Distância de Hamming* (veja Seção 1.3).

Diferentemente do código de paridade simples, que adiciona um único bit ao final da palavra para indicar a paridade (par ou ímpar) do número de bits com valor 1, e que pode apenas detectar erros em um número ímpar de bits, os Códigos de Hamming alcançam a maior taxa de detecção de erros para códigos com uma distância mínima de 3. Essa distância mínima permite que os Códigos de Hamming corrijam até um erro e detectem até dois. Já os *Códigos de Hamming Estendidos*, com distância mínima aumentada para 4, podem corrigir um erro e detectar dois ou mais erros. Ambos os tipos pertencem à categoria de códigos *SECDED (Single Error Correction, Double Error Detection).*

Nos Códigos de Hamming, as palavras-código são compostas por bits de dados e bits de paridade. Os bits de dados contêm a informação transmitida, enquanto os bits de paridade, calculados por meio da operação XOR (soma lógica) entre os bits de dados, são estrategicamente posicionados para permitir a detecção e correção de erros. Esses bits garantem que o número total de bits "1" na palavra-código satisfaça uma condição de paridade (par ou ímpar).

A posição dos bits de paridade é tipicamente escolhida como potências de dois (1, 2, 4, etc.), de modo que cada bit de paridade verifica combinações específicas de bits de dados e de outros bits de paridade. Para um número inteiro $r \geq 2$, uma palavra-código possui comprimento $n = 2^r - 1$ e comprimento da mensagem de dados $k = 2^r - r - 1$. Por exemplo, para $r = 3$, temos $n = 7$ e $k = 4$, significando que, em uma palavra-código de sete bits, quatro bits são dedicados aos dados e três aos bits de paridade. Nesse caso, a palavra-código pode ser representada como $p_1p_2d_1p_3d_2d_3d_4$, em que p_1, p_2, p_3 são bits de paridade, e d_1, d_2, d_3, d_4 são bits de dados.

Kautz e os Coils-in-the-Box

Kautz [10], nas décadas de 1950 e 1960, fez avanços significativos na teoria dos códigos ao buscar códigos especiais para a transmissão e armazenamento de informações por sinais elétricos digitais, focando na capacidade de detectar e corrigir erros. Kautz se baseou nas ideias pioneiras de Gray e Hamming, explorando palavras binárias correspondentes aos vértices dos coils em hipercubos de dimensão d. Kautz procurou identificar conjuntos de palavras binárias que pudessem minimizar erros complexos durante a transmissão ou armazenamento de dados, abordando cenários onde múltiplos bits poderiam ser alterados nas palavras binárias transmitidas ou armazenadas. Seu trabalho foi fundamental para o desenvolvimento de técnicas que aumentaram a confiabilidade e eficiência dos sistemas de comunicação digital e armazenamento de dados.

Em particular, snakes e coils são códigos de Gray que permitem a correção de um erro de transmissão e Kautz observou que tais códigos podem ser generalizados para detectar múltiplos erros de transmissão.

Estudos paralelos na União Soviética

Na mesma época em que se estavam desenvolvendo no ocidente as ideias de Gray, Hamming e Kautz, matemáticos na União Soviética estavam pesquisando e investigando problemas semelhantes, mas com foco em diferentes aspectos dos desafios combinatórios.

Desde o início da década de 1960, os matemáticos soviéticos estavam se defrontando com a necessidade de examinar todas as possibilidades em certos problemas combinatórios afim de encontrar uma solução. Eles reconheceram que, sob certas circunstâncias, não havia atalhos eficientes para evitar essa busca exaustiva (do russo "*perebor*"). Isso despertou em Trakhtenbrot [14] um interesse profundo em entender e provar a inevitabilidade dessa abordagem.

Na mesma época em que pesquisavam soluções para os problemas combinatórios, a partir de 1963, alguns matemáticos soviéticos se dedicaram ao estudo da Minimização de Funções Booleanas, com destaque para Vasil'ev [15] e Zhuravlev [16]. Eles exploraram os conceitos relacionados às *snakes-in-the-box* em suas pesquisas sobre a minimização de formas normais disjuntivas de funções booleanas, investigando como esses códigos poderiam explicar as dificuldades encontradas ao usar algoritmos locais para minimização. Essa abordagem foi crucial para aumentar a eficiência dos circuitos lógicos e sistemas computacionais. Suas contribuições incluem o desenvolvimento de métodos avançados para simplificar expressões booleanas complexas, visando reduzir o número de termos ou portas lógicas necessárias. Esses avanços são fundamentais para otimizar circuitos digitais, melhorando significativamente sua eficiência e desempenho.

Em 1969, Evdokimov [17] estendeu o trabalho de Vasil'ev, aprofundando a compreensão dos desafios em minimizar funções booleanas. Evdokimov analisou como a estrutura dos códigos binários dos caminhos snake-in-the-box tornava difícil a utilização de algoritmos locais para minimização, destacando as complexidades intrínsecas desses problemas. Para poder aproveitar os benefícios da utilização dos códigos das snakes e para superar as dificuldades do seu uso, Evdokimov indicou novas possíveis abordagens para lidar com tais complexidades.

Em anos posteriores Emelyanov e Lukito continuaram a examinar esses problemas, consolidando a importância dos códigos SIB na teoria da computação e combinatória [18]. Eles revisitaram os resultados anteriores e aplicaram novos métodos para estudar a minimização de funções booleanas, reforçando a importância das ideias originais de Zhuravlev e Vasil'ev.

Embora os desenvolvimentos soviéticos tenham ocorrido paralelamente e independentemente daqueles de Gray, Hamming e Kautz, houve uma conexão entre os estudos soviéticos e ocidentais, pois ambos os grupos estavam essencialmente lidando com problemas similares de minimização de erros e de eficiência computacional. As soluções e avanços obtidos por ambos os grupos de pesquisadores contribuíram significativamente para a teoria da informação, codificação e otimização combinatória.

1.5 Aplicações de Snakes e Coils

As snakes e coils foram e estão sendo utilizadas para resolver problemas em diversas áreas, com aplicações em Matemática e Ciência da Computação. Dentre outras aplicações destacamos as seguintes:

a. Correção de erros na transmissão de dados: Kautz [19] em 1962 demonstrou que é possível utilizar as sequências de vértices das snakes, as quais depois de manipuladas de forma adequada, geram uma lista de códigos binários, para transmitir dados e poder detectar e corrigir erros de transmissão.

b. Teoria dos Grafos: Em grafos da família dos hipercubos, as snakes são úteis para caracterizar propriedades dos grafos e estudar suas estruturas. Esta aplicação foi abordada em 1986 por Hamming [2].

c. Correção de erros em memória flash: Zhang e Ge [3] em 2015 mostraram que os códigos binários das snakes-in-the-box podem ser utilizados na técnica de Rank Modulation, que aproveita a ordem relativa das células de memória para corrigir erros e melhorar a eficiência da reescrita em dispositivos de memória flash.

d. Sincronização de clocks em microprocessadores: Com o aumento da complexidade dos microprocessadores, distribuir um único sinal de clock por todo o processador sem distorção significativa torna-se inviável. A divisão em várias regiões de clock cria um desafio na comunicação entre essas regiões, pois cada uma pode ter seu próprio sinal de clock, possivelmente dessincronizado com os outros. Em 2022, Bund [4] utilizou os códigos de snakes-in-the-box e demonstrou que sua aplicação nos algoritmos de sincronização de clocks melhorou significativamente a confiabilidade, eficiência e previsibilidade dos sistemas de microprocessadores.

e. Minimização de Formas Normais Disjuntivas de Funções Booleanas: Drapela [5], em 2015, discorreu sobre a equivalência entre a minimização de funções booleanas, como as Formas Normais Disjuntivas (DNF), e o problema (SIB). O número de variáveis em uma fórmula proposicional é equivalente ao número de dimensões em um hipercubo, e o hipercubo d-dimensional é análogo ao complexo espaço de busca onde podem ser encontradas minimizações ótimas. Assim, as técnicas úteis e benefícios para resolver o problema SIB também são aplicáveis e benéficos para a minimização de funções booleanas, e vice-versa.

1.6 Notações padrão para Snakes e Coils

Nesta dissertação, para evitar confusão, quando usarmos os termos *tamanho* ou *posição*, que são usados várias vezes para estruturas diferentes, o termo *tamanho* sempre se refere à quantidade de arestas na estrutura, e o termo *posição* sempre se refere à posição da dimensão que o bit está no vetor binário que representa os vértices dos hipercubos. Por exemplo para o vértice $1010 \in V(Q_4)$, as posições são numeradas de 0 a $d-1$, da direita para a esquerda conforme abaixo:

$$\begin{array}{rcccc} \text{vértice} & 1 & 0 & 1 & 0 \\ \text{posições} & \color{red}{3} & \color{red}{2} & \color{red}{1} & \color{red}{0} \end{array}$$

Desde o surgimento do problema SIB, foram utilizadas diversas formas para descrever e armazenar as snakes e coils. A seguir apresentamos as três notações mais frequentemente utilizadas, para facilitar o entendimento dos trabalhos relacionados e suas soluções. Em todos os exemplos dados a seguir, nas três notações, utilizamos a snake da Figura 6 em Q_4.

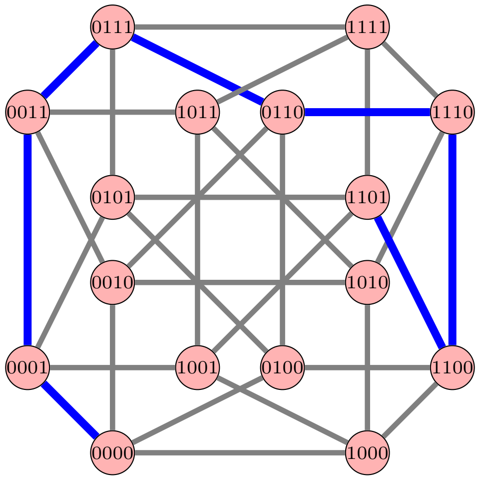


Figura 6: Snake em Q_4.

1.6.1 Sequência de vértices

A sequência dos vértices da snake é a notação utilizada em teoria dos grafos para representar um caminho. Os vértices podem ser identificados por sequências binárias ou decimais. No caso dos hipercubos, a maioria dos trabalhos representa os vértices pelas suas sequências binárias. A snake da Figura 6 pode ser representada pelas seguintes sequências de vértices.

$$\begin{aligned} S &= 0000, 0001, 0011, 0111, 0110, 1110, 1100, 1101 \\ S &= 0, 1, 3, 7, 6, 14, 12, 13. \end{aligned}$$

1.6.2 Sequência de Transições

A notação em sequência de transições, ao invés dos vértices, utiliza uma sequência de números de 0 até $d-1$, em que d é a dimensão do cubo estudado. Nesta notação, todas as snakes se iniciam no vértice $0 \cdots 0$. Como em uma snake a diferença entre dois vértices consecutivos é de precisamente um bit, representamos cada snake pela sequência de posições em que ocorrem tais mudanças. A numeração dos bits na palavra binária do vértice é feita da direita para a esquerda. Como temos uma transição a cada par de vértices adjacentes no caminho, isto faz com que a quantidade de transições corresponda à quantidade de arestas no caminho da snake, ou seja o tamanho da snake. A snake da Figura 6 pode ser representada pela seguinte sequência de transições.

$$S = 0, 1, 2, 0, 3, 1, 0.$$

Observe que a primeira transição é 0, pois os dois primeiros vértices (0000 e 0001) diferem apenas no bit de posição 0.

$$\begin{aligned} \text{vértices} &= 0000 \quad 0001 \\ \text{posições} &= 3210 \quad 3210 \end{aligned}$$

As demais transições seguem essa mesma regra, ou seja, recebem o valor da posição do bit em que houve a transição.

1.6.3 Sequência Binária

Considere uma snake S em Q_d. A notação de S em sequência binária é uma assinatura de S. Mais precisamente, a sequência binária de S é um vetor binário $(x_1, \ldots, x_{2^d})$ de comprimento 2^d, em que cada posição representa um vértice de Q_d, e tal que $x_i = 1$ se e somente se o vértice representado por x_i está em S. A snake da Figura 6 pode ser representada pela seguinte sequência binária.

$$S = (1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0).$$

1.6.4 Uso das notações

Dependendo do algoritmo e das estruturas de dados utilizadas, os pesquisadores usam uma das três notações apresentadas ou até mesmo mais de uma delas. É comum a implementação de funções para traduzir a representação de uma snake de uma notação para outra sempre que em determinada rotina do algoritmo uma outra notação é mais adequada. Para a apresentação dos resultados obtidos, principalmente em snakes de tamanho muito grande, a notação mais conveniente é a de sequência de transições. Para snakes em cubos de dimensão maiores do que 7 é comum também suprimir as vírgulas e representar números com mais de um dígito, como 10 e 11, etc, por outros símbolos (como letras) para tornar a notação mais compacta:

$$S = 0120310.$$

1.7 Organização do trabalho

No Capítulo 2, apresentamos os métodos analíticos utilizados para encontrar as maiores snakes, além de sumarizar e apresentar os limitantes inferiores e superiores, apresentamos também os melhores resultados encontrados com métodos computacionais, permitindo ao leitor analisar a evolução dos máximos ao longo do tempo. No Capítulo 3, fazemos uma análise das abordagens recordistas tanto para snakes quanto para os coils. No Capítulo 4, apresentamos uma solução baseada no método computacional que fundamenta esta dissertação. O algoritmo utilizado é uma variação do *Stochastic Beam Search*, um dos mais promissores tipos de algoritmos, amplamente empregado por diversos pesquisadores para resolver o problema SIB. A versão implementada difere daquela usada por Meyerson [20] sobretudo na forma como a aleatoriedade é aplicada ao comparar, aceitar ou descartar soluções que, em um nível local, parecem ter chances semelhantes de evolução. Finalmente, no Capítulo 5 apresentamos os resultados dos testes realizados, e no Capítulo 6 apresentamos as conclusões do presente trabalho, bem como direções para pesquisa futura.

Capítulo 2

Maiores Snakes e Coils

Os tamanhos das snakes e coils mais longos são estudados desde 1958, e resultados para cubos de várias dimensões foram obtidos pela utilização de vários métodos. Neste capítulo apresentamos os tamanhos das maiores snakes e coils conhecidos até o momento. Para dimensões até 8, já foram encontradas as maiores snakes e os maiores coils possíveis, enquanto para dimensões maiores do que 8, os valores máximos não são conhecidos, e para essas dimensões apresentamos os melhores resultados obtidos até o momento (veja Tabela 4 e Tabela 5). Além disso, apresentamos limitantes inferiores e superiores encontrados, e aprimorados desde o surgimento do problema SIB.

2.1 Abordagens utilizadas

O problema de encontrar as maiores snakes e os maiores coils em hipercubos é um problema clássico em Teoria dos Grafos e combina Otimização Combinatória com outras técnicas computacionais. As abordagens utilizadas para resolver esses problemas podem ser divididas em duas categorias principais: métodos analíticos e métodos computacionais.

Métodos Analíticos. Neste tipo de abordagem, foram utilizados dois métodos baseados em Teoria dos Grafos e Combinatória. O primeiro método, que chamamos de *Análise Teórica*, apoia-se em conceitos de Teoria dos Grafos, como caminhos Hamiltonianos e ciclos em grafos da família dos hipercubos, no qual os pesquisadores analisaram propriedades estruturais dos hipercubos para deduzir limitantes teóricos e construir manualmente exemplos de snakes e coils; e o segundo método, que chamamos de *Construtivo*, envolveu o desenvolvimento de algoritmos manuais que constroem sequências específicas de vértices, formando snakes ou coils com base em padrões reconhecíveis.

Utilizando tais métodos foram encontrados limitantes inferiores e superiores em função da d, a dimensão do hipercubo. Todos esses trabalhos consideram apenas coils. Tais limitantes são apresentados na Tabela 2 e na Tabela 3.

Tabela 2: Limitantes Inferiores. $|C_d|$ representa o tamanho do maior coil possível em Q_d.

Limitante	**Referência bibliográfica**
$\lvert C_d\rvert \geq \lambda \cdot 2^{(d/2)} \quad para \quad \lambda \geq 4$	Kautz 1958 [10]
$\lvert C_d\rvert \geq \left(\frac{3}{2}\right)^d$	Ramanujacharyulu e Menon 1964 [21]
$\lvert C_d\rvert \geq \lambda \cdot \left(\frac{5}{2}\right)^{(d/2)}$	Abbott 1965 [22]
$\lvert C_d\rvert \geq \left(\frac{7}{4}\right) \cdot \frac{2^d}{d-1}$	Danzer e Klee 1967 [23]
$\lvert C_d\rvert \geq \lambda \cdot 2^d \quad para \quad \lambda > 0$	Evdokimow 1969 [17]
$\lvert C_d\rvert \geq \frac{77}{256} \cdot 2^d$	Abbott e Katchalski 1991 [24]

Tabela 3: Limitantes Superiores. $|C_d|$ representa o tamanho do maior coil possível em Q_d

Limitante	**Referência bibliográfica**
$\lvert C_d\rvert \leq \frac{d}{d-1} 2^{(d-1)}$	Kautz 1958 [10]
$\lvert C_d\rvert \leq 2^{d-1}\left(1 - \frac{2}{d^2-d+2}\right)$	Solov'jeva 1987 [25]
$\lvert C_d\rvert \leq 1 + 2^{(d-1)} \cdot \frac{6d}{6d + \frac{\sqrt{d}}{6\sqrt{6}} - 7}$	Abbott e Katchalski 1991 [24]
$\lvert C_d\rvert \leq 2^{d-1}\left(1 - \frac{\sqrt{d}}{89} + O(\frac{1}{d})\right)$	Zémor 1997 [26]

Métodos Computacionais. Como o problema SIB apareceu na década de 1950, período no qual houve um desenvolvimento acelerado de computadores para uso em negócios e na academia, os pesquisadores se utilizaram dos computadores para abordá-lo. Esta prática se estende até os dias de hoje, em que temos disponíveis computadores cada vez mais poderosos.

Inicialmente os pesquisadores implementaram algoritmos de *Busca Exaustiva*, que exploram todas as possibilidades possíveis dentro do espaço disponível. Tais métodos tiveram sucesso nas dimensões de 1 a 6. Entretanto, em dimensões maiores, devido ao crescimento exponencial do espaço de busca das soluções possíveis se torna necessária a utilização de *Heurísticas e Metaheurísticas*, tais como Tabu Search [27], Algoritmos Genéticos [28], Simulated Annealing [29], Stochastic Beam Search [20], Monte Carlo [30], e Ant Colony Optimization [31], para encontrar soluções ótimas ou próximas ao ótimo, sem a necessidade de explorar todo o espaço disponível.

Com o desenvolvimento de redes de computadores e GPUs, que possibilitaram o uso de Computação Distribuída e de Computação Paralela, tais formas foram utilizadas, sempre que possível, nos algoritmos mencionados anteriormente. Desta forma foi possível aumentar a eficiência na busca por soluções, especialmente em hipercubos de dimensões pelo menos 8.

Tabela 4: Snakes recordistas. Os valores com * ‘direita são os valores máximos para a dimensão correspondente. Os valores com # à direita são valores obtidos apenas com métodos analíticos e construtivos

Dimensão	**Snake Recordista**	**Referência Bibliográfica**	**Método ou Algoritmo**
1	1*	Davies, 1965 [32]	busca exaustiva
2	2*	Davies, 1965 [32]	busca exaustiva
3	4*	Davies, 1965 [32]	busca exaustiva
4	7*	Davies, 1965 [32]	busca exaustiva
5	13*	Davies, 1965 [32]	busca exaustiva
6	26*	Davies, 1965 [32]	busca exaustiva
7	50*	Potter *et al*, 1994 [28]	algoritmo genético
8	98*	Carlson e Hougen, 2010 [33]	algoritmo genético
9	190	Wynn, 2012 [34]	busca exaustiva de snakes e construção com suas permutações em d-2 e d-1
10	373	Dang *et al*, 2023 [30]	Monte Carlo + NRPA com parada ótima
11	721	Dang *et al*, 2023 [30]	Monte Carlo + NRPA com parada ótima
12	1383	Dang *et al*, 2023 [30]	Monte Carlo + NRPA com parada ótima
13	2709	Dang *et al*, 2023 [30]	Monte Carlo + NRPA com parada ótima
14	4932#	Abbott e Katchalski, 1991 [24]	método de análise teórica + método construtivo
15	9866#	Abbott e Katchalski, 1991 [24]	método de análise teórica + método construtivo
16	19738#	Abbott e Katchalski, 1991 [24]	método de análise teórica + método construtivo

Tabela 5: Coils Máximos. recordistas. Os valores com * são os valores máximos para a dimensão correspondente. Os valores com # são valores obtidos apenas com métodos analíticos e construtivos

Dimensão	**Coil Recordista**	**Referência Bibliográfica**	**Método ou Algoritmo**
1	0*	Kautz, 1958 [10]	método construtivo
2	4*	Kautz, 1958 [10]	método construtivo
3	6*	Kautz, 1958 [10]	método construtivo
4	8*	Kautz, 1958 [10]	método construtivo
5	14*	Kautz, 1958 [10]	método construtivo
6	26*	Davies, 1965 [32]	busca exaustiva
7	48*	Kochut, 1996 [35]	busca exaustiva
8	96*	Östergård e Pettersson, 2014 [36]	método analítico e buscas exaustivas em d-2 e d-1
9	188	Wynn, 2012 [34]	busca exaustiva de coils e construção com suas permutações em d-2 e d-1
10	366	Allison e Paulusma, 2016 [37]	stochastic beam search
11	692	Allison e Paulusma, 2016 [37]	stochastic beam searchm
12	1344	Allison e Paulusma, 2016 [37]	stochastic beam search
13	2594	Allison e Paulusma, 2016 [37]	stochastic beam search
14	4934#	Abbott e Katchalski, 1991 [24]	método de análise teórica + método construtivo
15	9868#	Abbott e Katchalski, 1991 [24]	método de análise teórica + método construtivo
16	19740#	Abbott e Katchalski, 1991 [24]	método de análise teórica + método construtivo

Capítulo 3

Análise das abordagens recordistas

Neste capítulo, apresentamos resumidamente os métodos utilizados em alguns dos trabalhos para atingir os recordes registrados na Tabela 4 e na Tabela 5.

3.1 Snakes máximas nas dimensões de 1 a 6

Quando trabalhava no National Physical Lab na Inglaterra, inspirado pelo trabalho de Black [38], que descrevia uma operação para abrir um cadeado digital, Davies [32] estabeleceu uma relação entre tal problema e snakes-in-the-box.

Em sistemas de controle de acesso, o *cadeado digital* atua com um conjunto de chaves eletrônicas que permitem a abertura do circuito apenas sob condições específicas. Supondo o uso de d chaves eletrônicas para abrir o cadeado, em que cada chave é ativada por uma palavra binária específica, Black associou o processo de abertura do cadeado ao recebimento de uma sequência única de d palavras binárias com distância de Hamming de 1. Ao receber a sequência adequada, o cadeado digital aciona a chave correspondente a cada código, permitindo a abertura do cadeado, e consequentemente permitindo a utilização do circuito de forma controlada e segura. A implementação em Python desse algoritmo é apresentada no Algoritmo 3.1.

Considere então o cadeado cujo algoritmo de abertura é apresentado no Algoritmo 3.1. Se chamarmos `abre_cadeado` com `sequencia_codigos` com a lista `[000,001,011]`, o algoritmo devolve `aberto`. Por outro lado, se chamarmos `abre_cadeado` com `sequencia_codigos` com a lista `[000,001,010]`, o algoritmo devolve `fechado`, pois `010` não corresponde à terceira chave do cadeado.

Com o objetivo de minimizar a possibilidade do cadeado digital ser aberto indevidamente, Davies [32] aprimorou a ideia de Black, utilizando o conceito de amplitude (veja Seção 1.3), e sugerindo que a sequência de palavras para abertura do cadeado fosse um caminho induzido com amplitude pelo menos 2 (i.e., uma k-snake com $k \geq 2$). Motivado por essa ideia, Davies se dedicou a encontrar as snakes mais longas em hipercubos nas dimensões de 1 até 6. Usando um computador Fer-

Listing 3.1: Algoritmo que decide a abertura de cadeado eletrônico com três chaves comutadoras. O algoritmo possui o segredo do cadeado, e permite a sua abertura caso receba como input uma lista (`sequencia_codigos`) com os três códigos corretos. Neste caso "000", "001", e "011".

```
def abre_cadeado(sequencia_codigos):
    estado = "fechado"
    snake_segredo = ["000" , "001", "011"] # sequência correta
    chaves =       [False , False, False]
    for i in range(3):
        if (sequencia_codigos[i] == snake_segredo[i]):
            chaves[i] = True
    # se todas as chaves foram ligadas, muda o estado
    if (chaves[0] and chaves[1] and chaves [2]):
        estado = "aberto"
    return estado
```

ranti Hermes, Davies obteve, através de um algoritmo de busca exaustiva, os valores máximos para as snakes nestas dimensões.

Como a busca feita pelo algoritmo de Davies é exaustiva, os resultados obtidos são ótimos, i.e., máximos absolutos. Tais resultados são apresentados na Tabela 4 e na Tabela 5. Em particular, tal busca confirma que os coils encontrados por Kautz [10] para a dimensões 1 a 5 são ótimos.

3.2 Snakes máximas na dimensão 7

Em 1994, Potter et al. [28] utilizaram um algoritmo genético para buscar as maiores snakes possíveis nas dimensões 7 e 8.

Os *algoritmos genéticos* foram primeiramente desenvolvidos em 1975 por Holland [39] e são inspirados na teoria da evolução natural. Os algoritmos genéticos implementam um tipo de método de busca estocástica, sendo amplamente aplicados a problemas NP-Difíceis em diversas áreas como, por exemplo, no escalonamento de processos [40] (scheduling).

Em algoritmos genéticos, uma *população* é uma coleção de sequências de cromossomos, na qual cada sequência é uma solução para o problema específico em questão. Os cromossomos são alterados, utilizando operadores genéticos, a fim de criar uma nova população, ou *geração*. Este processo evolutivo é repetido um número predeterminado de vezes ou até que não seja encontrada uma solução melhor para o problema.

Potter et al. [28], ao usarem um algoritmo genético para abordar o problema SIB, representaram cada indivíduo da população, i.e., cada candidata a snake, por uma sequência de vértices do hipercubo expressos em valores decimais (veja Subseção 1.6.1), de 0 a $2^d - 1$, ou por uma sequência de transições (veja Subseção 1.6.2).

As duas formas de codificação foram implementadas.

Naturalmente, devemos testar a *viabilidade* de cada uma dessas sequências, isso é, devemos testar se cada uma dessas sequências é de fato um caminho induzido, e descartá-la em caso negativo. Além disso, para compararmos as soluções viáveis encontradas por um algoritmo utilizamos funções de avaliação, chamadas de *funções de aptidão*. Escolher e formular uma função de aptidão apropriada é crucial para garantir a eficiência do algoritmo genético em sua busca. No problema SIB, em que estamos buscando caminhos induzidos longos, uma função de aptidão baseada no comprimento do caminho é uma função razoavelmente adequada. Neste trabalho exploramos também duas outras funções de aptidão (veja Subseção 4.3.1).

Operadores genéticos são funções que permitem criar filhos (i.e., indivíduos da próxima geração) que diferem de seus pais (i.e., indivíduos da geração corrente). Potter et al. [28] exploraram operadores genéticos fundamentais como seleção de parceiros, cruzamento e mutação, e operadores genéticos avançados inspirados no conhecimento derivado do campo da genética. Os detalhes a respeito desses operadores fogem ao escopo deste trabalho.

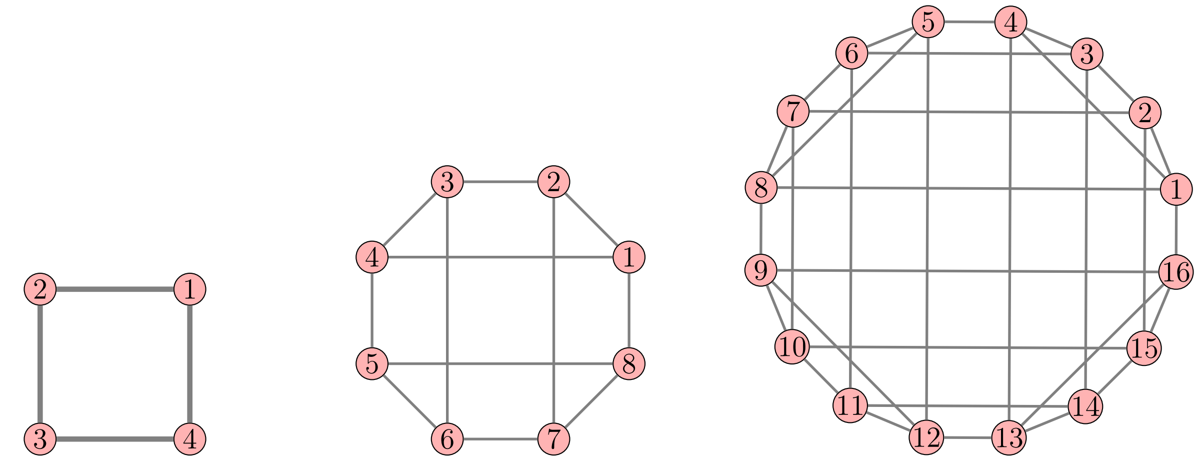
Com a sua implementação, Potter et al. [28] bateram os recordes da época, i.e., encontraram snakes mais longas do que as conhecidas até então, obtendo uma snake de tamanho 50 na dimensão 7 e uma snake de tamanho 89 na dimensão 8.

3.3 Snakes máximas na dimensão 8

Em 2010 Carlson e Hougen [33] uniram as técnicas de algoritmos genéticos e de algoritmos de busca com heurísticas, para construir um algoritmo genético customizado no qual incluíram vinte e duas heurísticas diferentes, também criadas por eles, com o objetivo de resolver o problema SIB na dimensão 8.

A representação gráfica dos hipercubos torna-se desafiadora para as dimensões maiores do que 3, e vários métodos foram utilizados para visualizar o espaço para dimensões superiores. No entanto, para facilitar a análise visual conforme uma snake cresce em dimensões mais altas, Carlson e Hougen desenvolveram uma nova representação bidimensional (2D) que realça as simetrias do espaço de busca.

Na Figura 7 podemos ver uma ilustração da representação gráfica circular em 2D usada por Carlson e Hougen para as dimensões 2, 3 e 4. Nessa representação, todos os nós são dispostos em posição círcular, com os números dos nós aumentando no sentido anti-horário, começando com o nó 1 à direita. À medida que novas dimensões são adicionadas, para construir a metade superior do círculo da nova dimensão movemos o número mais alto da dimensão anterior no sentido horário para a extremidade superior esquerda, ajustando os outros nós ao redor (o nó 1 permanece fixo à direita). Para construir a metade inferior do circulo, colocamos o novo conjunto de nós, incrementando seus números da esquerda para a direita, e as conexões na metade superior do círculo são espelhadas na metade inferior, e conectamos os nós correspondentes da metade superior à inferior, preservando a simetria.



(a) Q_2 Quadrado (b) Q_3 Cubo (c) Q_4 Cubo Tesseract

Figura 7: Hipercubos das dimensões 2, 3 e 4 na representação circular em duas dimensões, conforme proposta por Carlson e Hougen.

Para representar os hipercubos através de seus vértices e arestas em memória, Carlson e Hougen utilizaram a *lista de adjacências* de vértices, uma estrutura de dados bastante usada em Teoria dos Grafos [1], que mantém a vizinhança de cada vértice (veja Tabela 6).

Tabela 6: Lista de adjacências dos hipercubos - usada por Carlson e Hougen

Hipercubos			Vértice	Vértices Adjacentes			
			1	2	4	8	16
		Q_2	2	1	3	7	15
			3	4	2	6	14
	Q_3		4	3	1	5	13
			5	6	8	4	12
			6	5	7	3	11
			7	8	6	2	10
Q_4			8	7	5	1	9
			9	10	12	16	8
			10	9	11	15	7
			11	12	10	14	6
			12	11	9	13	5
			13	14	16	12	4
			14	13	15	11	3
			15	16	14	10	2
			16	15	13	9	1

No passo de mutação, Carlson e Hougen introduziram em seu algoritmo uma inovação importante, criando um operador customizado de cruzamento que combina informações do *genótipo* e do *fenótipo*, i.e., que leva em consideração tanto a codificação das cobras, quanto as suas características consequentes.

A implementação de Carlson e Hougen visava mais explorar a unificação das duas técnicas do que estabelecer algum recorde em alguma dimensão dos hipercubos, mas mostrou-se eficiente e conseguiu encontrar uma snake recorde de tamanho 98 na dimensão 8, superando o recorde anterior que era de 97 arestas e que havia sido encontrado nove anos antes (veja Tabela 4).

3.4 Snake máxima na dimensão 9

Em 2012, Wynn [34] apresentou duas novas construções para coils e snakes no hipercubo e, com elas, conseguiu melhorar os resultados conhecidos para snakes de dimensões 9, 10 e 11, e para coils de dimensões entre 8 e 13.

Na primeira construção, os códigos de coils são gerados a partir de permutações de uma sequência de transições inicial; na segunda construção, dois caminhos de menor dimensão são unidos com apenas uma ou duas alterações na dimensão mais alta; isso requer uma busca por uma permutação da segunda sequência para se ajustar à primeira. A geração de tais permutações é feita por um algoritmo apresentado por Knuth [41, Algoritmo 7.2.1.2X].

Embora em sua abordagem Wynn tenha focado em coils, ele também explorou snakes de amplitude k. Inicialmente, usando busca exaustiva, ele encontrou várias snakes máximas de dimensão 7, e utilizando a sua segunda construção, combinando pares das snakes de dimensão 7, conseguiu obter snakes de dimensão 8, que foram novamente combinadas para obter snakes de dimensão 9. Com isso, Wynn encontrou uma snake recorde de tamanho 190 e um coil de tamanho 188, ultrapassando os recordes anteriores que eram, respectivamente, de 188 e 180.

A ideia de combinar sequências de transições surgiu por intuição ao observar a ocorrência de padrões de repetições com algumas pequenas inversões ou modificações em exemplos práticos de snakes representadas por suas sequências de transições.

3.5 Snakes máximas nas dimensões de 10 a 13

Dang et al. [30] propuseram novas técnicas de uso geral para customizar e aperfeiçoar algoritmos de busca que utilizam o método de Monte Carlo. Tais técnicas se mostraram úteis para melhorar o desempenho de algoritmos em diversas aplicações em otimização combinatória, tais como `Roteamento de Caminho Mais Curto com Mínima Congestão`, `Problema do Caixeiro Viajante com Janela de Tempo` e o problema SIB, e com elas foram obtidos resultados melhores do que os anteriores para esses problemas.

Os algoritmos de busca Monte Carlo utilizam aleatoriedade para descobrir boas soluções para problemas complexos de otimização combinatória. Com isso, os algoritmos de Monte Carlo exploram várias possíveis soluções de maneira eficiente, no vasto espaço de soluções possíveis, ajustando-se dinamicamente com base nos resultados observados, de modo a escapar de soluções sub-ótimas locais e encontre melhores soluções em diferentes áreas do espaço de soluções. Após seu sucesso em jogos, os algoritmos de Monte Carlo foram aplicados com sucesso a vários problemas de otimização combinatória, usados em conjunto com aprendizado de máquina.

Os problemas acima já haviam sido explorados com algum sucesso por outros pesquisadores [42], que utilizaram a técnica de Adaptação de Política de Rollout Aninhada (NRPA padrão) em buscas com método de Monte Carlo. Dang et al. obtiveram resultados melhores daqueles obtidos com a NRPA padrão. Em particular, para o problema SIB, eles conseguiram estabelecer novos limitantes inferiores para as dimensões 10, 11, 12 e 13, obtendo snakes de tamanhos, respectivamente, 373, 721, 1383 e 2709 (veja Tabela 4).

3.6 Snakes máximas nas dimensões a partir de 14

Em 1969, Evdokimov [17] mostrou que existe uma constante $\lambda = 0.125$ para qual a maior snake em Q_d tem tamanho pelo menos $\lambda 2^d$, i.e., que a maior snake ocupa uma fração dos vértices de Q_d. Em 1988, Abbott e Katchalslki [43] apresentaram uma prova mais simples de tal resultado (veja Seção 2.1) com uma constante que depende dos resultados experimentais conhecidos, i.e., que é melhorada a cada novo recorde. Até onde sabemos, atualmente temos $\lambda = 77/256$.

Para dimensões pelo menos 14, não foi encontrado registro de nenhuma snake maior do que os limitantes inferiores correspondentes (veja Tabela 4).

3.7 Coils máximos nas dimensões de 2 a 7

Em 1958, Kautz [10] explorou coils máximos nas dimensões de 1 a 7, e conseguiu determinar através do método construtivo os coils máximos para as dimensões de 2 a 5. bem como obteve resultados próximos aos máximos nas dimensões 6 e 7 (veja também [32, 35])

Tabela 7: Coils obtidos por Kautz. Valores indicados por * correspondem aos valores máximos possíveis de coils na dimensão correspondente.

d	2	3	4	5	6	7
$\lvert C_d \rvert$	4*	6*	8*	14*	24	46

Em 1965, usando um algoritmo de busca exaustiva, em um computador Ferranti Hermes, da mesma forma que determinou as snakes de tamanho máximo nas dimensões de 1 a 6, Davies [32] estabeleceu o tamanho máximo absoluto de 26 para os coils de dimensão 6 (Veja Seção 3.1).

Em 1996, Kochut [35], que havia participado de um trabalho anterior com Potter et al. [28] em 1994, detalhou o processo para a obtenção do tamanho máximo absoluto de 48 para coils em cubos de dimensão 7 utilizando um algoritmo de busca em profundidade, de busca exaustiva, similar ao apresentado em 1994, porém, com otimizações substanciais. Em tal trabalho, Kochut também formalizou as restrições para reduzir o espaço de busca que explicamos em detalhes na Subseção 4.2.1.

3.8 Coils máximos na dimensão 8

Em 2014, Östergård e Pettersson [36] usaram um método misto e criativo, unindo os Métodos Analíticos e Métodos Computacionais, para provar que 96 é o tamanho máximo absoluto de coils na dimensão 8. Dentre os resultados obtidos, eles provaram que não existe um coil com mais de 123 vértices em um cubo de dimensão 8. A partir dessa prova e do resultado de um trabalho anterior que já havia encontrado um coil de tamanho 96 na dimensão 8 [44], restava provar que não existia nenhum coil no intervalo de 98 a 122 (isto porque o tamanho de um coil é sempre par). Eles mostraram que os coils na dimensão d poderiam ser obtidos a partir de snakes das dimensões $d-1$ e $d-2$, que eles conseguiram gerar com um algoritmo de busca exaustiva. O passo seguinte foi dividir o intervalo de 98 a 122 em dois intervalos, sendo o primeiro de $98 \leq t \leq 110$ e o segundo $112 \leq t \leq 122$ (em que t é o tamanho do coil buscado). A partir daí foi testada a geração de coils cujo tamanho estivesse dentro de cada intervalo. Como não foi encontrado nenhum tal coil, Östergård e Pettersson concluíram que o coil anteriormente encontrado, de tamanho 96, é um máximo absoluto na dimensão 8.

3.9 Coils máximos na dimensão de 9 a 13

O coil recorde na dimensão 9, de tamanho 188, foi obtido por Wynn, utilizando os mesmos métodos utilizados para encontrar uma snake de tamanho 190 na dimensão 9 (veja Seção 3.4).

Em 2016, Allison e Paulusma [37] estabeleceram novos recordes de tamanhos de coils nas dimensões de 10 a 13. Eles utilizaram o algoritmo Stochastic Beam Search, o mesmo utilizado por Meyerson [20], e também em nossa implementação (veja Seção 4.2).

3.10 Coils máximos nas dimensões a partir de 14

Abbott e Katchalski [24] utilizaram métodos analíticos para estabelecer limitantes inferiores inferiores para snakes e coils em geral. Novamente, não foi encontrado registro de nenhum coil maior do que as fornecidas por tais limitantes para a dimensões a partir de 14.

Capítulo 4

Uma implementação

Dentre os tipos de algoritmos utilizados até hoje para explorar o problema SIB, implementamos neste capítulo o algoritmo *Stochastic Beam Search* devido ao seu sucesso nas dimensões de 8 a 13. Primeiramente, em 2015, utilizando o Stochastic Beam Search, Meyerson [20, 45] confirmou (i.e., encontrou novamente) todos os recordes obtidos até então, e também obteve onze novos recordes de snakes e coils, o que mostra que o algoritmo é também eficaz para tratar o problema em dimensões maiores do que 8. Em 2016, Allison e Paulusma [37] também usaram um algoritmo do tipo Stochastic Beam Search e bateram os recordes de coils máximos nas dimensões 10, 11, 12 e 13, que ainda não foram ultrapassados.

Ressaltamos que tais trabalhos não apresentam pseudocódigos e nem os programas fonte da suas implementações. Meyerson dá apenas explicações gerais sobre as medidas de avaliação, muitas vezes difíceis de entender, e sem explicar precisamente como fazer a comparação entre as snakes de forma a decidir quais serão mantidas e quais serão descartadas. Da mesma forma, Allison e Paulusma, em um artigo técnico curto, apresentaram quase que somente os resultados obtidos, com uma promessa de publicação futura dos detalhes da implementação, que não pudemos encontrar.

Portanto, projetamos um algoritmo a partir do que entendemos de tais trabalhos, desde a escolha das estruturas de dados utilizadas até as heurísticas e as regras de aplicação da aleatoriedade. Também nos preocupamos em definir as estruturas de dados básicas para armazenar as subsoluções na memória de modo a minimizar o espaço ocupado e otimizar o tempo de processamento. Como o algoritmo escolhido é uma variante do algoritmo Beam Search, neste capítulo apresentamos uma explicação do seu funcionamento, um breve histórico de sua origem e aplicações (veja Seção 4.1) e, em seguida, detalhamos a nossa implementação.

4.1 Algoritmos de Beam Search

O Beam Search é um algoritmo de busca heurístico que explora grandes espaços de soluções mantendo a cada passo um conjunto limitado com as subsoluções mais promissoras. Em vez de manter as subsoluções obtidas em cada passo de execução, o beam search seleciona apenas aquelas subsoluções que apresentam os melhores valores de uma função de avaliação, restringindo o espaço de busca e reduzindo o tempo de processamento.

Esse processo de seleção confere ao Beam Search uma característica de algoritmo guloso. Entretanto, o Beam Search não é um algoritmo guloso no sentido estrito. Diferentemente de algoritmos gulosos, o beam search não se restringe a uma única escolha em cada etapa. Em vez disso, mantém um número fixo de subsoluções, permitindo uma exploração limitada de várias subsoluções em paralelo.

O Beam Search tenta encontrar um equilíbrio entre a exploração (investigar múltiplos caminhos) e a limitação do espaço de busca (para evitar a explosão combinatória). Portanto, o Beam Search não garante encontrar soluções ótimas, já que a exploração é limitada à quantidade máxima de subsoluções que cabem na memória do computador. Tal limite é chamado de *largura do feixe* (*beam width*). Isso, por outro lado, contrasta com algoritmos de busca exaustiva, que exploram completamente o espaço de soluções (como a busca em largura ou em profundidade). O Beam Search é especialmente útil em problemas nos quais a busca por uma solução ótima pode ser muito cara computacionalmente, permitindo um equilíbrio entre precisão e eficiência.

O Beam Search foi apresentado primeiramente em 1976 por Lowerre [46] para melhorar a eficiência e precisão do reconhecimento de padrões de fala. Lowerre apresentou o Beam Search pelo nome de *Locus Model of Search*, e Reddy [47], explorando esse mesmo problema, posteriormente renomeou tal algoritmo para *Beam Search.* O Beam Search, além de ser usado na área de Processamento de Linguagem Natural [48], é também usado nas áreas de Pesquisa Operacional e Logística [49], Reconhecimento de Padrões e Visão Computacional [50], Robótica e Planejamento de Movimentos [51], Otimização Combinatória [52], Bioinformática [53] dentre outras áreas.

4.2 O Algoritmo Stochastic Beam Search

O algoritmo Stochastic Beam Search é uma variante do algoritmo Beam Search. A única diferença entre o Beam Search e o Stochastic Beam Seach é o critério de seleção utilizado para manter ou descartar subsoluções. Enquanto o Beam Search avalia as subsoluções e mantém somente aquelas com as melhores medidas de avaliação, de forma determinística, o Stochastic Beam Search utiliza um critério de aleatoriedade. Dessa forma, as subsoluções mantidas são diferentes em cada execução, permitindo explorar áreas diferentes do espaço de busca, e aumentando a chance de conseguir snakes maiores.

Assim como ocorre em muitos métodos de busca heurísticos, não podemos garantir que o Stochastic Beam Search encontre a solução ótima, mas, em várias implementações, esse algoritmo forneceu snakes e coils que igualaram os tamanhos máximos conhecidos ou que até os superaram.

Antes de descrever o funcionamento do algoritmo Stochastic Beam Search, vamos apresentar quatro conceitos fundamentais para o seu entendimento: (1) a Redução do Espaço de Busca; (2) a Extensão das Subsoluções; (3) a Avaliação das Subsoluções; e (4) a Seleção das Novas Subsoluções.

4.2.1 Redução do Espaço de Busca

Em hipercubos de dimensões superiores a 7, o vasto espaço de busca torna necessário o uso de métodos eficazes para sua redução significativa. O método de redução do espaço de busca mais utilizado e incluído nos diversos tipos de algoritmos usados para abordar o problema SIB é obtido ao observar e aplicar as propriedades decorrentes da alta simetria dos hipercubos. Tal método foi aplicado desde o surgimento do problema por Kautz [19] e posteriormente por Davies [32], Adelson [54], Kochut [35], e por muitos outros pesquisadores mais recentemente.

Dois caminhos em um hipercubo Q_d são considerados *isomorfos* se puderem ser transformados um no outro por uma combinação de:

- Permutação das sequências de transições (i.e., reordenar as transições entre as dimensões do hipercubo); e
- Reflexão total das sequências de transições (i.e., fazer uma inversão da ordem das transições, do fim para o início).

As *classes de equivalência* de caminhos isomorfos são os conjuntos de caminhos que são isomorfos entre si. Essas classes de equivalência particionam o conjunto de todos os caminhos de tamanho t em Q_d em classes. Todos os membros de uma classe compartilham a mesma estrutura, mesmo que suas representações (sequências de vértices ou de transições) sejam diferentes.

Portanto, podemos e devemos considerar um único caminho como representante de cada classe de equivalência, reduzindo assim de forma significativa o espaço de busca para o problema SIB. Para tal devemos observar as seguintes restrições:

1. **Vértice inicial.** Devemos buscar apenas caminhos inciados em um mesmo vértice fixado. Em particular, desta forma, é evitado explorar dois caminhos de

uma mesma classe que iniciem em vértices diferentes. Aplicando esta restrição o espaço de busca total é dividido por 2^d, o que é uma redução importante, especialmente em hipercubos de dimensões maiores do que 6.

2. **Próxima transição.** Devemos explorar uma nova dimensão do hipercubo apenas após explorar todas as dimensões inferiores, i.e., ao buscar uma nova transição para ser incluída em uma snake S, devemos escolher uma transição que já ocorreu em S, ou a menor das transição que ainda não ocorreu em S.

As snakes geradas e que respeitem as restrições acima são ditas estar em *forma canônica* [35]. Em outras palavras, uma snake dada pela sua sequência de transições $t_1 \cdots t_s$ é dita estar na forma canônica se seu vértice inicial é `0` e se para todo $i \in \{1, \dots, s\}$ todos os números menores que t_i ocorrem em $\{t_1, \dots, t_{i-1}\}$. Uma prova formal de que tais restrições são suficientes para evitar explorar caminhos isomorfos foi apresentada em 1998 por McKay [55].

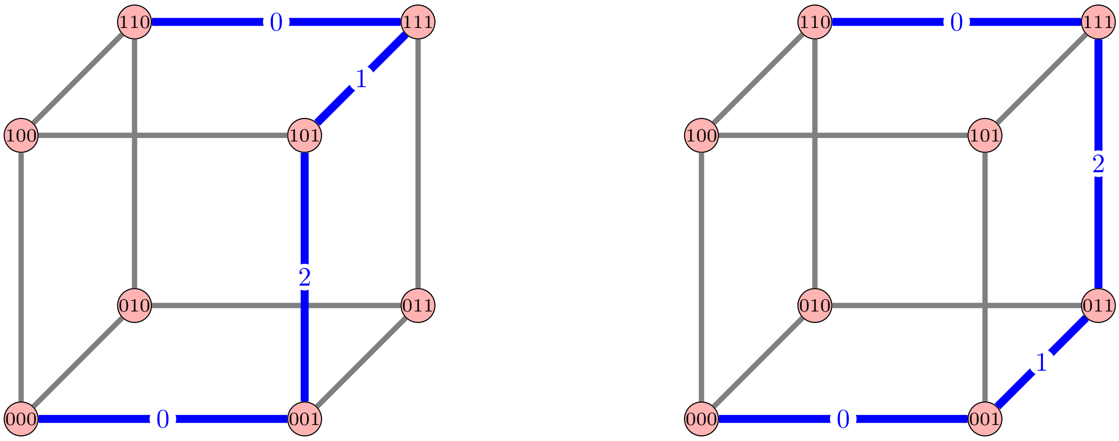
Qualquer sequência de transições em forma não canônica pode ser convertida para a forma canônica de sua classe de equivalência de maneira simples [55]. A seguir mostramos em um exemplo simples tal conversão.

Sequência de Transições em forma não canônica:	`[0,2,1,0]`
Transições fora de ordem :	`[0,2,1,0]`
Como seriam as transições em ordem canônica :	`[0,1,2,0]`

Para transformar a sequência acima da transição em uma sequência canônica, utilizamos uma permutação que troca apenas 1 com 2. Fazendo isso, temos a forma canônica correspondente: `[0,1,2,0]`. Na Figura 8 apresentamos essas duas formas. Note que as duas sequências de transições, apesar de serem diferentes, são permutações de uma mesma sequência de transições, e portanto estão na mesma classe de equivalência.

4.2.2 Extensão das Subsoluções

Para estender uma subsolução S de tamanho t para $t+1$ em Q_d, começamos por encontrar todas as transições possíveis do vértice da cabeça de S para os seus vértices vizinhos, numeradas de 0 até $d-1$. Dentre todas as transições possíveis eliminamos as duas últimas transições de S, para evitar ciclos de comprimento 2 (voltar ao penúltimo vértice) e 4 (voltar a um ciclo com os três últimos vértices e o vértice novo). Além disso eliminamos também todas as transições que violam a propriedade 2 da forma canônica (veja em 4.2.1). Após os dois tipos de eliminação temos então finalmente as transições candidatas a serem exploradas. Para cada transição candidata fazemos a sua inclusão na posição da cabeça da nova subsolução e verificamos se a subsolução obtida forma uma snake, i.e., se o caminho da subsolução é uma snake de



(a) Forma não canônica (b) Forma canônica

Figura 8: Formas Não Canônica e Canônica em Q_3

tamanho $t + 1$ (veja na Seção 1.3). Caso a subsolução seja uma snake, após passar pela próxima etapa de Seleção das Melhores Subsoluções, ela poderá ser incorporada ao conjunto de snakes de tamanho $t + 1$, caso contrário será descartada.

4.2.3 Avaliação das Subsoluções

Visto que a quantidade de snakes cresce exponencialmente com o aumento da dimensão do hipercubo, torna-se inviável armazenar todas as subsoluções na memória. Portanto, para nos prepararmos e dar subsídios para a etapa de seleção, é essencial definir e apurar métricas para avaliar cada subsolução, a fim de decidir quais serão mantidas e quais serão descartadas. Em nossa implementação utilizamos duas métricas, para avaliar e comparar as subsoluções, para decidir qual é a mais promissora para futuras extensões.

As duas métricas são baseadas na quantidade de *vértices alcançáveis* a partir do vértice da cabeça da snake S sendo avaliada. No que segue, dado um vértice $v \in V(Q_d)$, denotamos por $N(v)$ o conjunto de vértices em Q_d que são adjacentes a v; e dado um conjunto de vértices $S \subset V(Q_d)$, denotamos por $N(S)$ o conjunto de vértices em Q_d que são adjacentes a pelo menos um vértice de S.

Agora, seja $S \subset V(Q_d)$ uma snake cuja cabeça é v. Os *vértices alcançáveis* a partir de v são os vértices $u \in V(Q_d) \setminus V(S)$ para os quais existe um caminho P de w a u, em que (i) w é um vizinho de v que não está em S, e tal que (ii) $\big(V(P) \setminus \{u, w\}\big) \cap \big(S \cup N(S)\big) = \emptyset$. O conjunto de vértices alcançáveis é denotado por Alcançáveis(S). Dado um vértice alcançável $u \in$ Alcançáveis(S), dizemos que u é *indisponível* se $u \in N(S)$, e *disponível* caso contrário (veja Figura 9). Em particular, todo vértice alcançável disponível pode ser utilizado para estender S. Finalmente, dizemos que um vértice alcançável disponível u é dito um *nó cego* se u possui apenas um vizinho que é alcançável e disponível. Neste caso, ao estendermos

S até um nó cego, não podemos continuar a estendê-la. O conjunto de nós cegos é denotado por $C(S)$ (veja Figura 10).

Observamos que o conjunto Alcançáveis(S) pode ser obtido através de uma busca (e.g., busca em largura ou busca em profundidade) a partir de $N(v) \setminus S$ na qual não utilizamos os vértices em $S \cup N(S)$ como vértices internos. Um pseudocódigo para computar os vértices alcançáveis e disponíveis é apresentado no Listing 4.1, e um pseudocódigo para computar os nós cegos é apresentado no Listing 4.2.

Listing 4.1: Pseudocódigo para computar o conjunto de vértices alcançáveis.

```
def alcancaveis(v):
    '''
        Assumimos que cubo.in_S (resp. cubo.in_NS) éum vettor booleano de marcação
        que indicam se um dado vértice está em S (resp. N(S))
    '''
    visitado   = [False]*cubo.order # Uma lista que indica os vértices visitados
    alcancavel = [False]*cubo.order # Uma lista que indica os vértices alcançáveis
    disponivel = [False]*cubo.order  # Uma lista que indica os vértices disponíveis

    fila = deque() # Fila para a BFS
    for w in cubo.vizinhos(v): # Os caminhos podem começar pelos vizinhos de u que não estão em S
        if not cubo.in_S[w]:
            quaue.append(w)
            visitado[w], alcancavel[w] = True, True

    while fila: # enquanto a fila não fica vazia
        atual = fila.pop()

        for viz in cubo.vizinhos(atual):
            if visitado[viz]: continue
            if not cubo.in_S(viz):
                alcancavel[viz] = True
                # os vértices que podem ser vértices internos não podem estarm em S U N(S)
                if not cubo.in_NS[viz]:
                    disponivel[viz] = True
                    fila.append(viz)

        visitado[atual] = True

    return alcancavel, disponivel
```

Listing 4.2: Pseudocódigo para computar o conjunto de nós cegos.

```
def nos_cegos(alcancavel, disponivel):
    nos_cegos = [False]*cubo.order
    for u in cubo.vertices:
        if alcancavel[u] and disponivel[u]:
            num_viz_alc_disp = 0
            for viz in cubo.vizinhos(u):
                if alcancavel[viz] and disponivel[viz]:
                    num_viz_alc_disp += 1
            if num_viz_alc_disp == 1:
                nos_cegos[u] = True
    return nos_cegos
```

Fitness. A métrica mais intuitiva para identificar a snake mais promissora é a quantidade de vértices alcançáveis disponíveis que não são nós cegos. Naturalmente, quanto maior for a quantidade de tais vértices para uma cobra S, maior a chance de que S possa continuar a ser estendida. Tal métrica é chamada de `fitness`. Formalmente, dada uma cobra S, temos

$$\texttt{fitness}(S) = |\text{Alcançáveis}(S) \setminus \big(N(S) \cup C(S)\big)|.$$

Um pseudocódigo para computar a fitness de uma snake é apresentado no Listing 4.3. Ao compararmos duas subsoluções, consideramos a mais promissora aquela que tem o maior valor de `fitness`. Quando for necessário, descartamos a subsolução menos promissora.

Listing 4.3: Pseudocódigo para computar a fitness de uma snake.

```
def fitness(alcancavel, nos_cegos):
    fitness = 0
    for u in cubo.vertices:
        if alcancavel[u] and not cubo.in_NS[u] and not nos_cegos[u]:
            fitness += 1
    return fitness
```

Skin. Quando houver empate entre duas snakes na métrica de `fitness`, o critério de desempate será uma segunda métrica, que apresentamos a seguir.

Dada uma snake S cuja cabeça é um vértice v, os vértices alcançáveis e indisponíveis que não são adjacentes a v são chamados de *nós da pele* de S, i.e., os nós alcançáveis que estão em $N(S)$. Observe que a contagem de nós da pele de S pode ser feita na mesma busca em que encontramos os nós alcançáveis e os classificamos entre disponíveis e indisponíveis. Tal métrica é chamada de `skin_fit`. Formalmente, dada uma cobra S, temos

$$\texttt{skin\_fit}(S) = |\big(\text{Alcançáveis}(S) \cap N(S)\big) \setminus N(v)|$$

Um pseudocódigo para computar a fitness de uma snake é apresentado no Listing 4.4.

Listing 4.4: Pseudocódigo para computar a skin_fit de uma snake.

```
def skin_fit(alcancavel, nos_cegos):
    v = snake.cabeça
    skin_fit = 0
    for u in cubo.vertices:
        if alcancavel[u] and cubo.in_NS[u] and u not in cubo.vizinhos(v):
            skin_fit += 1
    return skin_fit
```

Acreditamos que uma snake com mais nós de pele tem maior potencial de ser estendida a snakes mais longas, pois a utilização de vértices adjacentes a muitos vértices da pele atrasa a indisponibilização dos vértices alcançáveis. Em outras palavras, ao incluirmos em S um vértice que é adjacente a muitos nós da pele, o número de vértices alcançáveis que se tornam indisponíveis é menor. Desta forma, mantemos mais vértices no conjunto de vértices alcançáveis, atrasando saturação das áreas do hipercubo e permitindo que as snakes se estendam mais livremente. Ao compararmos duas subsoluções que possuem a mesma `fitness`, consideramos a mais promissora aquela que tem o maior valor de `skin_fit` e, novamente, quando for necessário, descartamos a subsolução menos promissora.

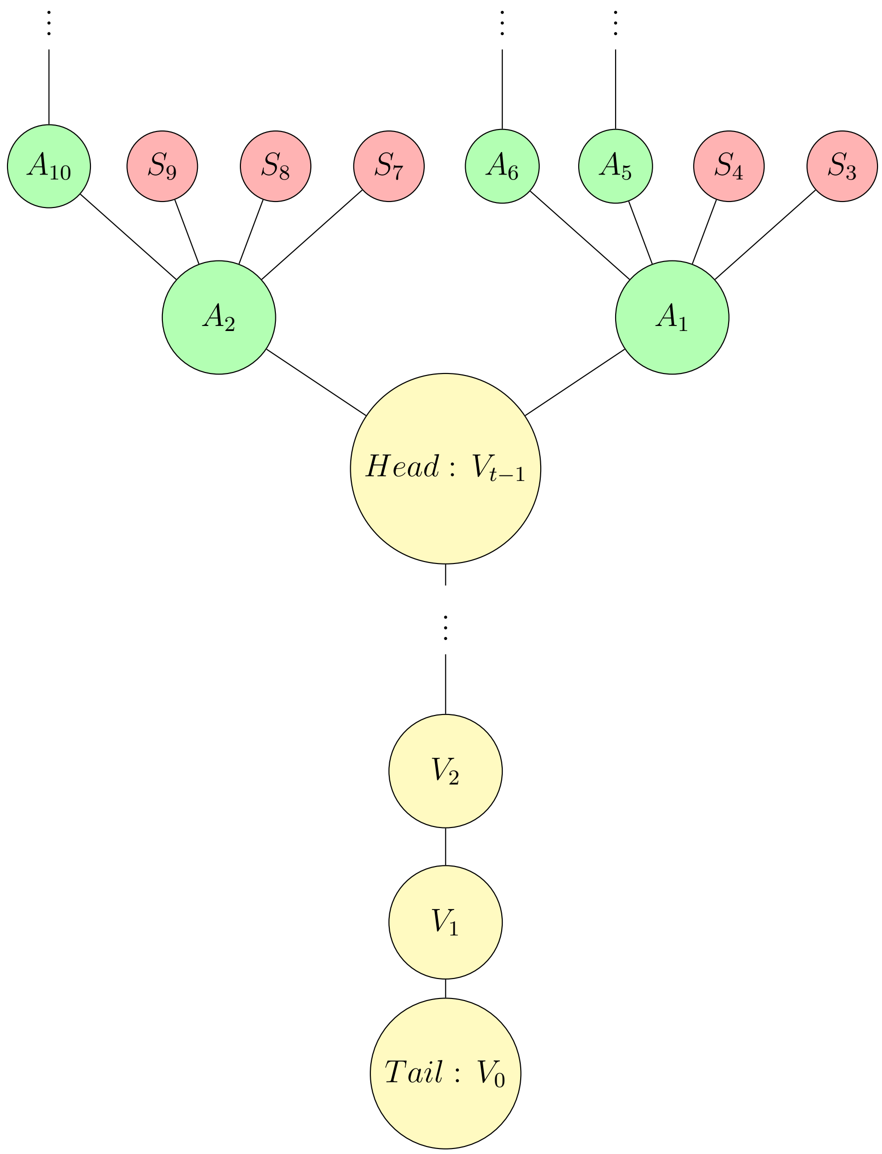


Figura 9: Representação parcial de subsolução em Q_5. Os vértices da cobra S são ilustrados na cor amarela. Os vértices alcançáveis disponíveis e indisponíveis são ilustrados, respectivamente, nas cores verde (A_1, A_2, A_5, A_6 e A_{10}) e vermelha (S_3, S_4, S_7, S_8 e S_9).

Na Figura 10 e nas Tabelas 8 e 9 ilustramos o cálculo da `fitness` e `skin_fit`.

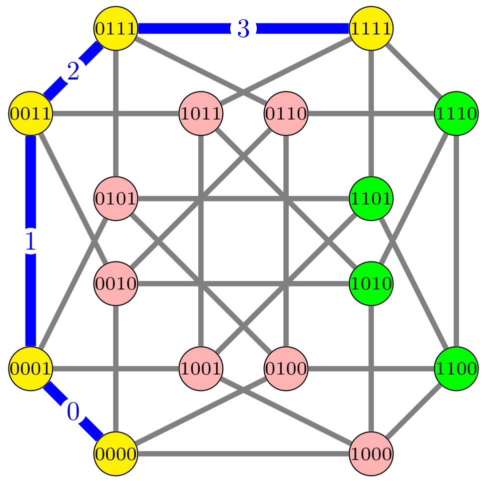


Figura 10: Uma snake S em Q_4 ilustrada na cor azul. A sequência de transições de S é 0123. Nas cores verde e vermelho ilustramos, respectivamente, os vértices alcançáveis disponíveis e indisponíveis.

Tabela 8: A apuração da `fitness`(S) será feita contando a quantidade de vértices alcançáveis e disponíveis em S_{va} que é igual a 4, observando que todos eles têm pelo menos um vértice vizinho alcançável disponível, i.e., nenhum deles é um nó cego.

Vértice(s) alcançável(is) e disponível(is)	Contribuição para `fitness`	Vértice(s) vizinho(s) alcançável(is) e disponível(is)
1110	1	1010, 1100
1010	1	1110
1100	1	1101, 1110
1101	1	1100
`fitness`	4	

4.2.4 Seleção das Novas Subsoluções

Se houver espaço na estrutura que armazena as novas subsoluções, a inclusão de uma nova subsolução é feita de forma imediata. Caso contrário, quando não há mais espaço, devemos realizar a etapa de seleção. Nesta etapa devemos determinar se a subsolução recém-gerada deve substituir a subsolução menos promissora dentre as subsoluções que se encontram na estrutura que armazena as subsoluções novas. Tal substituição observa os seguintes três critérios.

1. Substituir a pior subsolução armazenada pela nova subsolução caso a nova subsolução apresente melhores métricas de avaliação;

2. Substituir a pior subsolução armazenada pela nova subsolução de forma aleatória caso suas métricas de avaliação sejam iguais;

3. Substituir a pior subsolução armazenada pela nova subsolução de forma aleatória, em sorteio com uma baixa probabilidade de ocorrência. Esse critério busca oferecer oportunidade a subsoluções que, embora apresentem valores de

Tabela 9: A apuração da `skin_fit`(S) é calculada pela soma das contribuições dos nós de pele adjacentes aos vértices alcançáveis disponíveis em S_{va}, assegurando que cada nó de pele seja contado apenas uma vez.

Vértice alcançável e disponível	Contribuição para `skin_fit`	Vértice(s) de pele	Observações
1110	1	0110	
1010	3	0010, 1000, 1011	
1100	1	0100, ~~1000~~	vértice já contado antes
1101	2	0101, 1001	
`skin_fit`	7		total das contribuições

`fitness` ou `skin_fit` inferiores à pior subsolução armazenada, possam obter melhores resultados nas próximas iterações, escapando de máximos locais.

Os sorteios realizados nos Itens 3. e 2. utilizam parâmetros específicos fornecidos à função de seleção. Essa abordagem permite explorar uma diversidade de subsoluções, mitigando o risco de convergência prematura e aumentando a chance de encontrar soluções mais promissoras no espaço de busca.

4.3 A implementação

Inicialmente, desenvolvemos um protótipo de algoritmo para o problema Snake-in-the-Box (SIB) utilizando a linguagem Python, implementando o algoritmo Stochastic Beam Search. Essa escolha se deu pela facilidade e rapidez de prototipação oferecida pelo Python, o que é ideal para validar conceitos e ajustar os primeiros detalhes do algoritmo. No entanto, ao lidar com hipercubos de dimensões superiores a 6, percebemos que a execução do programa apresentava uma lentidão significativa. Isso se deve às limitações de desempenho intrínsecas ao Python, especialmente em operações computacionalmente intensivas que envolvem iterações e manipulação de grandes estruturas de dados.

Diante desse desafio, decidimos migrar o programa para a linguagem C++. Escolhemos o C++ por ser uma linguagem de alto desempenho, permitindo uma execução mais eficiente devido à sua proximidade com o hardware e à otimização proporcionada pelos compiladores modernos. Após a conversão, os testes revelaram que o tempo de execução do programa em C++ foi cerca de 7 a 8 vezes menor em comparação à versão em Python, confirmando a eficácia da mudança para atender às demandas de maior escalabilidade e desempenho.

Para um melhor entendimento dos principais passos do algoritmo, que serão apresentados adiante, vamos descrever a notação utilizada para representar os caminhos das snakes e, também, as principais estruturas de dados utilizadas.

Notação

Em nossa implementação representamos o conjunto dos vértices do hipercubo em palavras binárias de d bits, e para as snakes utilizamos duas das notações que apresentamos na Seção 1.6: a sequência de vértices (veja Subseção 1.6.1) e a sequência de transições (veja Subseção 1.6.2). As transições são representadas por objetos da classe `Transition` (veja Subseção 4.3.1). Cada snake tem um *vértice inicial*, *um vértice final*, que são os extremos do caminho induzido, i.e., da snake, e os demais vértices são chamados de *vértices internos*. Neste trabalho também chamamos o vértice inicial de *vértice de cauda* (em inglês *tail*) e o vértice final de *vértice cabeça* (em inglês *head*). A Figura 10 mostra uma snake cuja sequência de vértices é $S = 0000, 0001, 0011, 0111, 1111$, na qual o vértice 0000 é o vértice de cauda, o vértice 1111 é o vértice cabeça.

Como na execução do algoritmo as snakes começam pequenas e vão aumentando de tamanho, por convenção, definimos que o vértice da cauda da snake é o vértice que fica fixo, e o vértice cabeça da snake é o vértice no qual novos vértices serão acrescentados de forma a obter snakes mais longas. Ao adicionarmos um novo vértice adjacente à cabeça da snake, tal vértice passa a ser a cabeça da nova snake obtida, substituindo o vértice cabeça anterior. Neste trabalho utilizamos sempre o vértice $0 = 0 \cdots 0$ como vértice cauda para todas as snakes. Desta forma, respeitamos a restrição de vértice inicial para a redução do espaço de busca (veja Subseção 4.2.1 Item 1). Escolhemos 0 como o vértice de cauda, pois tal vértice é o vértice escolhido pela grande maioria dos pesquisadores, ao que parece influenciados pela escolha de Kochut em suas pesquisas (veja [35]).

4.3.1 Estruturas de Dados

Primeiramente, implementamos a classe `Transition`, essencial para a representação de uma snake pela sua sequência de transições. A classe `Transition` contém os seguintes cinco atributos (veja Listing 4.5). Um pseudocódigo da classe transition é apresentado no Listing 4.6.

- `father` - É um atributo do tipo ponteiro nos permite acessar o objeto pai, implementando uma lista encadeada na qual a partir da última transição podemos obter a sequência completa das transições de uma snake;
- `transition` - É um atributo do tipo `int` que representa a dimensão que mudou entre os dois últimos dois vértices da snake. Este atributo pode variar de 0 a $d - 1$, em que d é a dimensão do cubo explorado;
- `max-seen` - É um atributo do tipo `int` que indica a maior transição já ocorrida na snake atual até o presente momento. Este atributo pode variar de 0 a $d - 1$

assim como `transition`, e nos permite gerar apenas snakes na forma canônica (veja Item 4.2.1);

- `fitness` - É um atributo do tipo `int` na qual é armazenada a principal variável de avaliação para decidir quais são as snakes mais promissoras, aquelas que provavelmente têm mais chances de crescimento (veja Subseção 4.2.3);
- `skin_fit` - É um atributo do tipo `int` na qual é armazenada a segunda variável de avaliação para decidir quais são as snakes mais promissoras, aquelas que provavelmente têm mais chances de crescimento (veja Subseção 4.2.3).

Listing 4.5: Início da definição da classe Transition em C++.

```
class Transition {
public:
Transition * father;     // ponteiro p/ objeto pai (mesma classe Transition)
int transition;          // posição da dimensão em que ocorre a transição
int max_seen;            // transição máxima até esta transição corrente
                         // para garantir forma a forma canônica
int fitness;             // quantidade de vértices alcançáveis
int skin_fit;            // quantidade de vértices tipo skin
 ...
```

Listing 4.6: Pseudocódigo da classe Transition.

```
class Transition:

    def __init__(self, transition, father = None):
        self.transition = transition
        self.max_seen = transition
        if father != None:
            self.max_seen = max(transition, father.max_seen)

        self.father = father

    def is_snake(self):
        t = self
        vertice = (0)*dimension # começa no vértice (0,...,0)
        cubo.mark(vertice)     # marca o vértice inicial
        while t.father != None:
            t = t.father
            vertice[t.transition] = (vertice[t.transition] + 1) % 2
            cubo.mark(vertice)
            for viz in cubo.vizinhos(vertice):
                if cubo.is_marked(viz):
                    return False
        return True

    def children(self):
        result = []
        m = min(self.max_seen + 2, self.dimension)
        for t in range(m):
            new_transition = Transition(self.dimension, t, self)
            if new_transition.is_snake():
                result.append(new_transition)
        return result
```

Além dos objetos da classe `Transition` que representam as transições, utilizamos as seguintes duas estruturas de dados fundamentais para o nosso algoritmo nas quais armazenamos ponteiros para os objetos da classe `Transition`.

- `beam` - É um array de ponteiros no qual armazenamos os ponteiros para os objetos da classe Transition. O comprimento do array `beam` é chamado de `largura do beam`. Neste array armazenamos o conjunto corrente de soluções de tamanho t. Cada ponteiro armazenado aponta para a última transição de cada uma das snakes em tal conjunto.

- `new_beam` - É objeto do tipo `priority_queue min-heap`, no qual armazenamos ponteiros para objetos da classe Transition. Assim como no `beam`, cada ponteiro armazenado aponta para a última transição de cada uma das snakes no conjunto de subsoluções. Enquanto `beam` aponta para subsoluções de tamanho t, `new_beam` aponta para subsoluções de tamanho $t+1$, obtidas a partir das subsoluções que em `beam`. Neste tipo de estrutura, uma árvore binária completa implícita é implementada diretamente em um vetor em C++, fornecendo operações eficientes de inserção de novos elementos (push), de remoção do elemento de prioridade mínima (pop), e de consulta do menor elemento (top). A prioridade de um elemento em `new_beam` é o par $(\texttt{fitness}, \texttt{skin\_fit})$ que são comparados dois-a-dois de forma lexicográfica (i.e., primeiro é comparado o `fitness` e, em caso de empate, é comparado o `skin_fit`). Como foi escolhido o `min_heap`, o elemento de prioridade mínima é armazenado no topo da árvore e aponta para a snake que possui as piores medidas de avaliação, i.e., aponta para a snake de menor $(\texttt{fitness}, \texttt{skin\_fit})$. Assim ao compararmos cada nova snake gerada com a snake no topo de `new_beam`, i.e., com a snake com as piores medidas de avaliação, podemos decidir se devemos descartar a nova snake ou se devemos substituir a snake do topo da `new_beam` pela nova snake gerada (veja Subseção 4.2.4).

4.3.2 Visão geral da implementação.

Nesta seção, apresentamos uma descrição resumida da implementação do algoritmo Stochastic Beam Search que foi utilizada neste trabalho. Um pseudocódigo da visão geral da implementação é apresentado no Listing 4.7, e o código em C++ é apresentado no Seção 6.

Passo 1. **Inicialização:** O algoritmo começa preparando os objetos da Classe `Transition`(nós) para criar a subsolução inicial do problema $S_i = (0, 1, 2)$, que é a subsolução válida para todos os hipercubos de dimensão pelo menos 3. Para isso, são criados três objetos da classe `Transition`, cujos atributos `transition` são respectivamente 0, 1, e 2, e que são encadeados de forma linear pelo atributo `father` (veja Figura 11). Após isso, o ponteiro para o nó correspondente à última transição da subsolução (o nó com o atributo `transition` igual a 2) é incluído em `beam`.

Passo 2. **Extensão das subsoluções:** Para cada subsolução (de tamanho t) S apontada pelos ponteiros em `beam`, são geradas todas as novas subsoluções possíveis de tamanho $t+1$ a partir S (veja Figura 14). Cada nova subsolução gerada será processada nos Passos 3 e 4 a seguir, para determinar se será ou não incluida em `new_beam`.

- **Passo 3.** **Avalia:** Para cada nova subsolução de tamanho $t+1$ gerada, são calculadas as medidas `fitness` e `skin_fit`, que serão usadas para comparar a tal subsolução com aquela de pior avaliação entre todas as subsoluções em `new_beam`.
- **Passo 4.** **Insere/substitui/descarta:** Para cada nova subsolução de tamanho $t+1$ gerada e avaliada, são observados os critérios para sua inserção, descarte ou para que substitua a pior subsolução armazenada em `new_beam` (veja 4.2.4).

Passo 5. **Atualização de `beam`:** Se `new_beam` não está vazio, esvaziamos `beam` e copiamos os ponteiros armazenados em `new_beam` para `beam`. Retornamos ao Passo 2.

Passo 6. **Finaliza:** Se `new_beam` está vazio, o algoritmo apresenta as subsoluções que estão em `beam` e termina sua execução.

Na implementação utilizada, focamos na busca por snakes mais longas. Entretanto, a mesma implementação pode ser modificada para se obter os coils mais longos, inserimos o seguinte teste na execução do Passo 5 acima para cada ponteiro de subsolução que é copiado de `new_beam` para `beam`.

5.1 **Fechamento do coil:** Se o ponteiro a ser copiado apontar para uma solução de tamanho par, verificamos se, a partir da cabeça da subsolução obtida existe um par de vértices consecutivos, alcançáveis e disponíveis cujo segundo vértice seja o vértice 0 (i.e., a cauda da snake), de forma a obter um ciclo induzido (i.e., um coil). Em caso afirmativo, adicionamos as transições correspondentes a tais vértices, apresentamos a solução do coil encontrado e continuamos o processamento do Passo 5.

Listing 4.7: Pseudocódigo da visão geral da implementação.

```
# Parâmetros
dimension, size_beam, size_new_beam, tx_rand1, tx_rand2 = 6, 4000, 4000, 2, 20

# Passo 1 — Inicialização
t0 = Transition(0,None)
t1 = Transition(1,t0)
t2 = Transition(2,t1)

beam = [t2]
new_beam = []

while beam != []:

# Passo 2 — Extensão das subsoluções
    for snake in beam:
        for child in snake.children():

# Passo 3 — Avaliação
            child.calculate_fitness()
            child.calculate_skin_fit()

# Passo 4 — Insere/substitui/descarta
            if len(new_beam) < size_new_beam:
                new_beam.append(child)
            else:
                pior = new_beam.top()
                condicao_1 = (child.fitness,child.skin_fit) > (pior.fitness, pior.skin_fit)
                condicao_2 = ((child.fitness,child.skin_fit) == (pior.fitness, pior.skin_fit)) and
                     (random.random() < tx_rand2)
                condicao_3 = random.random() < tx_rand1
                if condicao_1 or condicao_2 or condicao_3:
                    new_beam.pop() # extrai o topo do min_heap
                    new_beam.insert(child) # insere no min_heap com prioridade (child.fitness,child.skin_fit)

# Passo 5 — Atualização do beam
    if new_beam != []:
        beam = new_beam
        new_beam = []
    else:
        return beam
```

Para facilitar o entendimento, vamos mostrar um exemplo em Q_4. Como pode ser visto na Figura 11 e na Figura 12, na execução do **Passo 1 de Inicialização**, criamos os objetos da Classe `Transition` (nós) para compor a subsolução inicial $S_i = (0, 1, 2)$ do problema, e inserimos em `beam[0]` a referência (endereço) para o nó que representa a transição 2 em S_i. Esta última transição é a transição que leva o vértice interno 0011 e o vértice 0111, que é a da cabeça da snake inical.

No Passo 2, em um processo iterativo, tentamos estender as snakes de tamanho t da população corrente, referenciadas pelos ponteiros armazenados em `beam`, para gerar um novo conjunto de subsoluções de tamanho $t+1$ que podem ser armazenadas em `new_beam`, dependendo dos critérios a serem verificados no Passo 3 e no Passo 4.

Nas Figuras 13 e 14 ilustramos duas snakes estendidas na iteração #1 do Passo 2

Passo 1: Coloca snake inicial em beam[0]

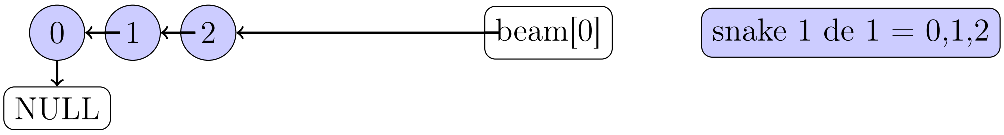


Figura 11: Ilustração do Passo 1 do algoritmo. Os nós representam os objetos da classe `Transition` criados inicialmente.

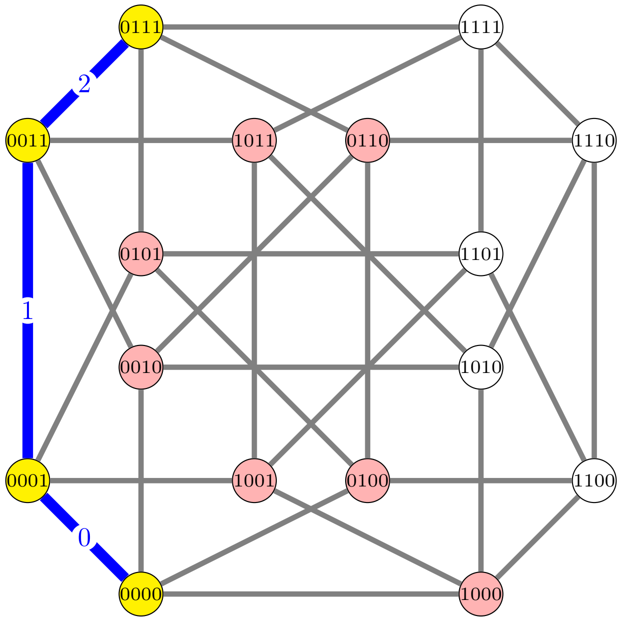
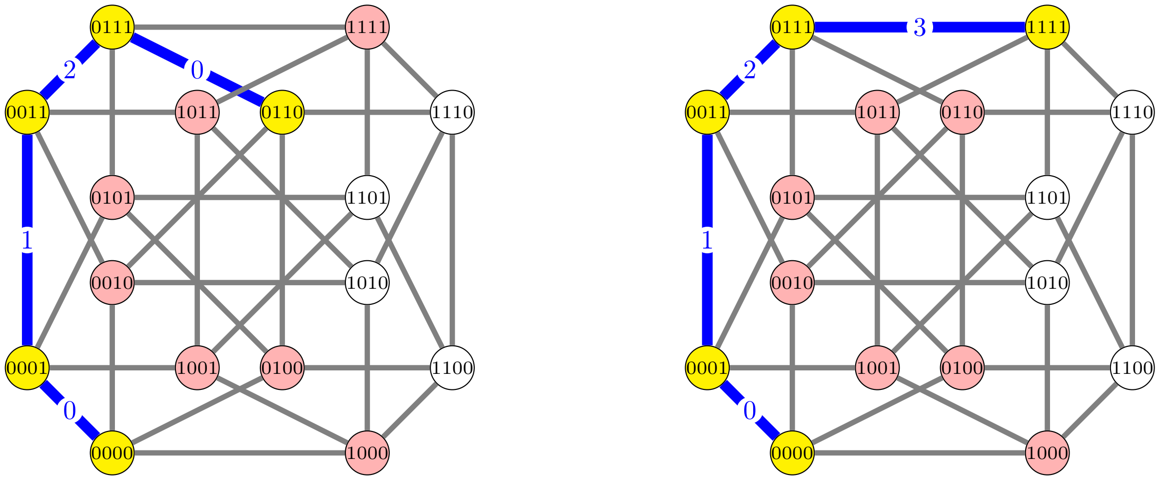


Figura 12: Ilustração do Passo 1 do algoritmo. As arestas em azul representam a snake inicial

do algoritmo. Após essa iteração `beam` contém duas subsoluções de tamanho 4. Na Figura 14, apresentamos a evolução do `beam` do exemplo da Figura 12, após as duas primeiras iterações da execução do Passo 2. Após a segunda iteração temos em `beam` três subsoluções de tamanho 5. Ao final de todas as iterações do Passo 2, após gerar todas as snakes de tamanho $t+1$ possíveis a partir de cada snake armazenada em `beam`, executamos o Passo 5, em que fazemos um teste para verificar se existe pelo menos uma subsolução de tamanho $t+1$ em `new_beam`. Se existir pelo menos uma tal subsolução, esvaziamos `beam` e transferimos (os ponteiros para) as snakes armazenadas em `new_beam` para `beam` e retornamos para o Passo 2. Se não existir nenhuma tal subsolução em `new_beam`, i.e., não foi possível estender as subsoluções correntes, finalizamos o programa.

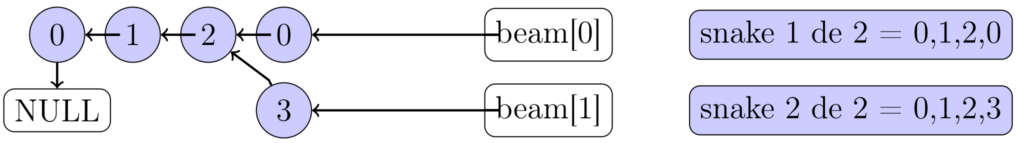


(a) Snake Inicial estendida - 1

(b) Snake Inicial estendida - 2

Figura 13: Passo 2 - Iteração #1 Extensão da Snake Inicial em 2 outras

Passo 2: Chama increase_beam (iteração # 1)



Passo 2: Chama increase_beam (iteração # 2)

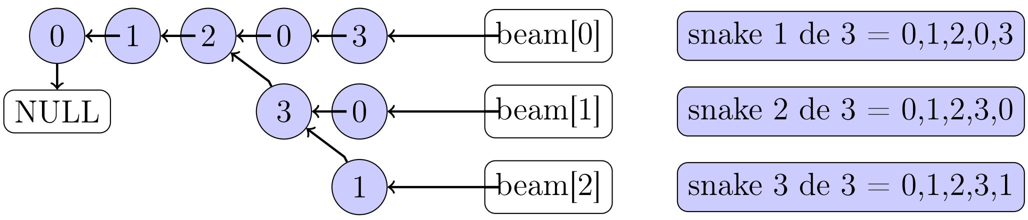


Figura 14: Ilustração do Passo 2 do algoritmo implementado neste trabalho. Evolução de `beam` imediatamente após a inicialização da snake $S = 0, 1, 2$ no Passo 1 após um e duas execuções do laço dos passos 2–5.

Capítulo 5

Testes realizados

Neste capítulo apresentamos os testes que foram realizados com o algoritmo que apresentamos neste trabalho. O algoritmo implementado admite cinco parâmetros que podem ser modificados no arquivo `def.h` (veja Seção 6), e que levam a obtenção de resultados diferentes, bem como tempos de processamento diferentes. Naturalmente, o parâmetro principal é a dimensão `dimension` do hipercubo no qual estamos buscando a maior snake; de forma secundária o algoritmo possui dois parâmetros para controlar o comprimento das estruturas de dados nas quais armazenamos as subsoluções encontradas (`beam` e `new_beam` – veja Subseção 4.3.1); finalmente, o algoritmo possui dois parâmetros relacionados aos sorteios realizados durante a inserção das novas subsoluções em tais estruturas (`txrand1` e `txrand2`). Neste capítulo apresentamos os resultados e tempos de processamento obtidos pela variação destes parâmetros.

1. `dimension` (`int`) – o dimensão do cubo explorado;

2. `SIZE_BEAM` (`int`) – o comprimento de `beam`;

3. `SIZE_NEW_BEAM` (`int`) – o comprimento de `new_beam`;

4. `TXRAND1` (`int`) – taxa referente à probabilidade de troca de snakes para fugir de máximos locais no Passo 4. Uma nova subsolução substitui a pior snake armazenada em `new_beam` com probabilidade `TXRAND1`/100 independentemente das suas medidas de `fitness` e `skin_fit`, conforme Subseção 4.2.4 item 1;

5. `TXRAND2` (`int`) – taxa referente à probabilidade de troca de snakes para fugir de máximos locais no Passo 4. Uma nova subsolução substitui a pior snake armazenada em `new_beam` com probabilidade `TXRAND2`/100 quando suas medidas de `fitness` e `skin_fit` empatam com a pior snake armazenada, conforme Subseção 4.2.4(3).

Observe que ao aumentarmos o valor de `dimension`, os comprimentos das snakes encontradas aumentam (veja Capítulo 2), assim como o tempo de avaliação da validade de cada nova transição (veja Subseção 4.2.2), e o espaço de busca. Dado tal aumento no espaço de busca, se faz necessário aumentar os comprimentos de `beam` e `new_beam` para armazenar uma quantidade adequada de subsoluções. Se `beam` e `new_beam` tiverem comprimentos maiores do que o tamanho do espaço de busca, nenhuma subsolução é descartada e, portanto, todas as snakes possíveis são encontradas e, consequentemente, a snake mais longa é encontrada. Naturalmente, isso não é viável já para dimensões maiores que 6.

Os comprimentos de `beam` e `new_beam` são, portanto, cruciais para a execução do algoritmo. Na medida em que tal comprimento aumenta, aumentamos a probabilidade de guardar snakes promissoras, que levam a subsoluções mais longas, bem como uma maior quantidade de tais subsoluções. Paralelo a isso, o tempo de execução aumenta devido ao número de subsoluções avaliadas em cada iteração (veja Passo 2 na Subseção 4.3.2). Embora a implementação admita que os comprimentos de `beam` e `new_beam` sejam diferentes, os testes realizados com uma tal diferença não demonstraram resultados melhores do que aqueles nos quais os comprimentos foram iguais.

Isso pode ser justificado pelo fato de que as snakes excedentes sempre são descartadas em algum momento. Se `SIZE_BEAM` e `SIZE_NEW_BEAM` são iguais, as snakes excedentes são descartadas de uma por uma na medida em que são geradas (veja Passo 4 na Subseção 4.3.2); e se `SIZE_NEW_BEAM` for maior que `SIZE_BEAM`, o excesso de snakes armazenadas em `new_beam` é descartardo ao final da iteração (veja Passo 5 na Subseção 4.3.2). Portanto, nos resultados dos testes apresentados a seguir, ao nos referirmos ao comprimento do `beam`, estamos nos referindo aos dois comprimentos i.e., ao tamanho do `beam` e ao tamanho do `new_beam`.

No restante do capítulo apresentamos os resultados obtidos dos experimentos de acordo com as dimensões exploradas. O algoritmo Stochastic Beam Search customizado foi executado em um notebook ASUS VivoBook com processador Intel® Core™ i7-1165G7 @ 2.80GHz × 8 cores – 11a. geração, com 16GB de memória RAM, e sistema operacional Ubuntu 22.04.5 LTS.

5.1 Testes nas dimensões de 3 a 6

Nos testes para as dimensões de 3 a 6 o algoritmo mostrou-se muito eficiente. Com relação ao tempo, conseguimos igualar os recordes para essas dimensões em tempos inferiores a 1 seg (veja Tabela 10). Da mesma forma, o algoritmo mostrou-se eficiente com relação à utilização de memória, uma vez que foi necessário apenas comprimento 85 para as estruturas `beam` e `new_beam`, i.e., foi necessário armazenar apenas 85 subsoluções concomitantemente.

Tabela 10: Resultados obtidos de tamanhos de snakes, tempo de execução e comprimento de `beam` para cubos de dimensões de 3 a 6

Dimensão (d)	Snake $\|S_d\|$	Tempo de Processamento	Comprimento do Beam	Quantidade de Snakes
3	4*	0h 0m 0s	85	1
4	7*	0h 0m 0s	85	1
5	13*	0h 0m 0s	85	1
6	26*	0h 0m 0s	85	1

As snakes mais longas encontradas nestas dimensões são apresentadas na Tabela 11.

Tabela 11: Sequências de transições das snakes mais longas encontradas em cubos de dimensões de 3 a 6.

Dimensão	Sequência de transição da Snake mais longa
3	0120
4	0120310
5	0123014021032
6	01231043054013402410431534

5.2 Testes na dimensão 7

Na dimensão 7 o recorde foi obtido em 6 segundos, porém nesta dimensão o custo de memória já se mostrou maior do que o necessário anteriormente. Na Tabela 12 apresentamos os resultados obtidos ao aumentarmos gradativamente o comprimento do `beam` de 85 a 1800, quando então foi possível igualar o recorde. Observe que com comprimentos menores que 1800, a maior snake encontrada teve tamanho 48 ou 49 e, portanto, o recorde não foi alcançado. Com comprimentos de `beam` a partir de 4000 conseguimos obter várias snakes de tamanho recorde em cada processamento, o que mostra a influência do comprimento do `beam` para encontrar uma quantidade maior de soluções (veja Tabela 12). Na Tabela 13 apresentamos algumas das snakes encontradas em cada teste realizado.

5.3 Testes na dimensão 8

Os testes realizados neste trabalho se concentraram no cubo de dimensão 8 devido ao tamanho significativo do espaço de busca, que impõe desafios consideráveis ao algoritmo tanto em termos de memória quanto de tempo de processamento. Neste

Tabela 12: Resultados obtidos de tamanhos de snakes, tempos de execução, comprimentos de `beam` e quantidade de snakes obtidas para o cubo de dimensão 7

Teste #	Tamanho Snake	Tempo de Processamento	Comprimento do Beam	Quantidade de Snakes
01	48	0h 0m 0s	85	1
02	49	0h 0m 2s	800	1
03	50*	0h 0m 6s	1800	1
04	50*	0h 0m 13s	4000	3
05	50*	0h 0m 49s	15000	5
06	50*	0h 5m 28s	100000	8
07	50*	0h 24m 51s	500000	12

caso, foi encontrada uma snake de tamanho 97 utilizando um `beam` de comprimento 80000, enquanto a maior snake em Q_8 tem tamanho 98. Na Tabela 14 apresentamos os resultados obtidos ao aumentarmos gradativamente o comprimento do `beam` até 3500000, mesmo assim não foi possível igualar o recorde.

Embora não tenhamos atingido o recorde absoluto de tamanho 98 na dimensão 8, os testes reforçaram que, ao aumentar o comprimento do `beam`, é possível obter não apenas snakes de maior comprimento, mas também uma quantidade maior delas. Por exemplo, ao utilizarmos comprimento do `beam` entre 80000 e 500000, conseguimos obter duas snakes de tamanho 97, enquanto quando aumentamos o comprimento do `beam` para 1000000, obtivemos quinze snakes de tamanho 97 em um único processamento. Algumas das snakes mais longas encontradas nesta dimensão são apresentadas na Tabela 15.

5.4 Testes na dimensão 9

Os testes na dimensão 9 mostraram que o algoritmo implementado não é eficiente para esta dimensão, pois o tempo de execução é demasiadamente grande, mesmo para comprimentos de `beam` pequenos. Isto pode ser justificado pelo tamanho do espaço de busca no hipercubo de dimensão 9. Na Tabela 16 apresentamos os resultados obtidos ao aumentarmos gradativamente o comprimento do `beam` até 300000. Similarmente aos testes na dimensão 8, não foi possível igualar o recorde (190). Diferentemente dos testes na dimensão 8, nesta dimensão o recorde não é comprovadamente o tamanho da maior snake possível. Algumas das snakes mais longas encontradas nesta dimensão são apresentadas na Tabela 17.

Tabela 13: Sequências de transições das snakes mais longas encontradas no cubo de dimensão 7.

Teste #	Dimensão	Tamanho	Sequência de transição da Snake mais longa
01	7	48	012031430540650345015305436452640143054 364014364
02	7	49	012302410230153023106501320150314302352 1032560230
03	7	50	0123453210325312345321032631230531034530 1230531034
04	7	50	0120310421035012406504203401240350420340 1206240124
04	7	50	0120310421035012406504203401240350420340 1206104210
04	7	50	0123453210325312345321032631230531034530 1230531034
05	7	50	0123014021035023065032103502306403210650 1230150210
05	7	50	0123453210325312345321032631230531034530 1230531034
05	7	50	0123014021035023065032103502306403201650 1230150210
06	7	50	0120310421035012406504203401240350420340 1206104210
06	7	50	0123014021035023065032103502306403201650 1230150210
06	7	50	0123453210325312345231032605430135023103 5430125023
07	7	50	0123014021035023065032103502306403210650 1230150210
07	7	50	0123453210325312345321032631230531034530 1230531034
07	7	50	0123014021035023065032103502306403201650 1230150210

Tabela 14: Resultados obtidos de tamanhos de snakes, tempo de execução, comprimento de `beam`, e quantidade de snakes diferentes no cubo de dimensão 8.

Teste #	Tamanho Snake	Tempo de Processamento	Comprimento do Beam	Quantidade de Snakes
01	93	00h 02m 10s	5000	12
02	94	00h 09m 05s	20000	27
03	97	00h 34m 17s	80000	2
04	97	02h 16m 24s	300000	2
05	97	07h 22m 08s	1000000	15
06	97	24h 23m 27s	3500000	15

Tabela 15: Sequências de transições das snakes mais longas encontradas em cubos de dimensão 8.

Teste #	Dimensão	Tamanho	Sequência de transição da Snake mais longa
01	8	93	01230140210325306507603210267021561034106510470156036102630674063072035236215263 0625632675210
02	8	94	01203240154124015305406532012415476451421023064056152105142647052074023421051701 36146017031641
03	8	97	01231041325106152315612517213254231270123106132510415231541251721325623145132160 12315241231041325
04	8	97	01231041325106152315612517213254231270123106132510415231541251721325632145132160 12315241231041325
05	8	97	01234532103253123452310326054301350231035430125023763201354301350231035430136243 50251035205435201
05	8	97	01234532103253123452310326054301350231035430125023763205210345301320531034526302 53450310251031245
06	8	97	01234532103253123452310326054301350231035430125023763205210345301320531034526302 51304213015201305
06	8	97	01231041325106152315612517213254231270123106132510415231541251721325623145132160 12315241231041325

Tabela 16: Resultados obtidos de tamanhos de snakes, tempo de execução, comprimento de `beam`, e quantidade de snakes diferentes no cubo de dimensão 9.

Teste #	Tamanho Snake	Tempo de Processamento	Comprimento do Beam	Quantidade de Snakes
01	172	00h 14m 45s	5000	13
02	172	00h 44m 03s	20000	15
03	177	05h 36m 13s	100000	157
04	179	15h 02m 05s	300000	88

Tabela 17: Sequências de transições das snakes mais longas encontradas em cubos de dimensão 9.

Teste #	Dimensão	Tamanho	Sequência de transição da Snake mais longa
01	9	172	0120340210540126521076012872102760856876084058678061431608768560876023064260140684021658075061206543584602165860746758742608760570817860756257835740326824580457836035730647
02	9	172	0120340210540126521076012872102760856876084058678061431608768560876023064260140684021658075061206543584602165860746758742608760570817860756270430638745763574032682453046032
03	9	177	012345321024032063012306703210870120610321067012074021541234103470320760234532067023064032083012046021076012354321067012038430238210276257025420340120358302351023401207486283603
04	9	179	01230240123015410321651230176103218712301271321671230156103214512301251321651230154103176130145103163413258230450321561231521032154123016510315864130670327523185741321451230125132
04	9	179	01230240123015410321651230176103218712301271321671230156103214512301251321651230154103275231436130154103168130450321561231520132154123106510315864130670327521358741321451230125132

Capítulo 6

Conclusão e Direções para Pesquisas Futuras

Neste trabalho apresentamos um estudo do problema Snake-In-The-Box, que devido a sua diversa aplicabilidade despertou bastante interesse na comunidade da Ciência da Computação sobretudo até o início do século XXI. Embora tal interesse tenha aparentemente diminuído, novas aplicações em tecnologia de ponta surgiram nos últimos anos, o que motiva a sua constante (re)visitação. Compilamos aqui os melhores resultados obtidos até o presente momento, bem como analisamos as implementações e estratégias algoritmicas utilizadas para a obtenção de tais resultados, e apresentamos uma implementação detalhada de um desses algoritmos.

Devido a diversos contratempos (e.g., Pandemia de COVID-19), algumas direções de pesquisa não puderam ser exploradas. No caso do cubo de dimensão 8, em especial, é possível investigar outras heurísticas para a seleção de snakes a serem mantidas ou descartadas, bem como heurísticas que auxiliem a fuga de soluções ótimas locais. Já para dimensões superiores, além da utilização de comprimentos de `beam` maiores, se torna necessária a utilização de processamento paralelo, pois os espaços de busca são muito maiores do que os explorados até então.

Outra alternativa interessante para pesquisas futuras é a possibilidade da utilização de computação quântica, como proposto por Fuidio, Canale e Sotelo [56], que utilizaram um algoritmo quântico para resolver o problema SIB na dimensão 3.

ANEXO I - Programas-Fonte

Nesta seção apresentamos os códigos-fonte em linguagem C++ elaborados neste trabalho. Estes códigos também estão disponíveis em [57], e estão estão organizados nos seguintes três arquivos.

1. `def.h`:

   - Contém as definições dos parâmetros que podem ser modificados a cada processamento. Este arquivo serve como base para ajustar os valores necessários para as execuções do programa.

2. `transitionClass.cpp`:

   - Contém a definição da classe `Transition`, que permite a criação e manipulação de objetos que representam as transições dos vértices das *snakes* armazenadas em memória. Devido ao seu tamanho, este arquivo foi dividido em sete partes para facilitar a leitura.

3. `beam.cpp`:

   - Contém o algoritmo *Stochastic Beam Search (SBS)* customizado, bem como as funções associadas e o programa principal. Devido ao seu tamanho, este arquivo foi dividido em cinco partes.

Definições de parâmetros – arquivo: def.h

```
// PARÂMETROS QUE PODEM SER ALTERADOS A CADA PROCESSAMENTO
#define SNAKE_DIMENSION 8
#define TXRAND1 2  // PERCENTUAL PARA TROCA DE SNAKES COM QQ
                   // FITNESS E SKIN_FIT P/ FUGIR DE MÁXIMOS
                   // LOCAIS

#define TXRAND2 20      // PERCENTUAL PARA TROCA DE SNAKES COM A
                        // MESMA FITNESS E MESMA SKIN_FIT EM
                        // NEW_BEAM E NEW_SEQUENCES

#define SIZE_BEAM   250000    // TAMANHO DO beam
#define SIZE_NEW_BEAM 250000  // TAMANHO DO new_beam
#define DEBUGAR 4

// PARÂMETROS PARA NÃO SEREM ALTERADOS A CADA PROCESSAMENTO
#define TWO_TO_THE_N (1 << SNAKE_DIMENSION) // CALCULA VÉRTICES DO CUBO
// DEFINE VERTOT E IGUALA A TWO_TO_THE_N
// SNAKE_DIMENSION 3 4   5   6    7    8    9    10     11
// VERTOT         8  16  32  64  128  256  512  1024   2048
#define VERTOT TWO_TO_THE_N
// DIMENSÃO MÁXIMA ( == DIMENSÃO DO CUBO - 1) PORQUE COMEÇA DE 0
#define MAX_DIMENSION SNAKE_DIMENSION -1
extern int new_fitness;      // new_fitness DEFINIDA EM beam.cpp
                             // USADA EM transitionClass.cpp
extern int snake_length;     // snake_length DEFINIDA EM beam.cpp
                             // USADA EM transitionClass.cpp
extern int debugar;          // debugar DEFINIDA EM beam.cpp
                             // USADA EM transitionClass.cpp
extern int new_skin_fit;     // new_skin_fit DEFINIDA EM beam.cpp
                             // USADA EM transitionClass.cpp
```

Classe `Transition` - Parte 1 Variáveis Globais

arquivo: `transitionClass.cpp`

```

// Variáveis GLOBAIS
const int vertot = VERTOT;       // total de vértices do cubo

vector <string> vert_bin;        // vértices do cubo em fmt string binário
                                 // string "-1" vértice não disponível

vector <string> vert_bin_fixo;   // todos os vértices do cubo usado para
                                 // inicializar o vert_bin

string viz_bin[SNAKE_DIMENSION]; // vértices vizinhos de um outro vértice.

vector <string> viz_bin_ind;     // todos os vizinhos indisponíveis de um vértice
                                 // alcançável a partir da cabeça da snake e que
                                 //não estão na própria snake

vector <string> vert_comp;       // todos os vértices da componente conexa
                                 // alcançáveis e disponíveis no hipercubo,
                                 // a partir da cabeça da snake em questão.

int snake_length;                // variável que mantem o tamanho da snake
                                 // corrente sendo processada

vector <int> trans;              // vetor de inteiros c/ todas transições de uma
                                 // determinada snake

vector <string> vert_bins;       // vértices em fmt string binário
                                 // com todos os vértices de uma snake
```

Classe `Transition` - Funções - Parte 2

arquivo: `transitionClass.cpp`

```cpp
// Função converte (vertice) do formato string binário p/ valor inteiro a ser
    retornado
int bin2int(const std::string& vertice) {
    int vint = 0;
    // Inicia c/ bit mais a esquerda
    int bitValue = 1 << (vertice.size() - 1);

    for (char bit : vertice) {
        if (bit == '1') {
            vint |= bitValue; // Põe 1 no bit correspondente
        }
        bitValue >>= 1; // Vai p/ próximo bit
    }
    return vint;
}
// Função que coloca no array viz_bin em fmt string binário,
// os vizinhos de um vértice string v de entrada
void vizinhos(string v){
    string vx;
    for (int i=0; i < SNAKE_DIMENSION; i++)
    {
        vx = v;
        if (vx[i] == '0')
            vx[i] = '1';
        else
            vx[i] = '0';
        viz_bin[i] = vx;
    }
    return;
}
```

Classe Transition - Funções - Parte 3

arquivo: transitionClass.cpp

```
// Função que transforma para fmt string binário os vértices correspondentes
// a um vetor de transições trans em fmt inteiro, devolvendo os vértices
// em fmt string binário no vetor vert_bins
void trans2vertb(const std::vector<int>& trans) {
    vert_bins.clear();
    // Inicializa vert_bins c/ '0'
vert_bins.push_back(std::string(SNAKE_DIMENSION, '0'));
    for (int i = 0; i < trans.size(); i++) {
        // Guarda cópia do último vertice em ult_vertex
        std::string ult_vertex = vert_bins.back();
        int desl = SNAKE_DIMENSION - trans[i] - 1;
        ult_vertex[desl] = (ult_vertex[desl] == '0') ? '1' : '0';
        vert_bins.push_back(ult_vertex);
    }
}
// Cria e coloca em vert_bin todos os vértices do cubo em
// formato string (binário) por exemplo: se dimensão do cubo == 8,
// retorna em vert_bin = ["00000000"], ["00000001"], ... ["11111111"]
void ini_vert_bin(vector <string>& vert_bin, int vertot) {
    for (int i=0; i < vertot; i++)
        {
        string vb = bitset<SNAKE_DIMENSION>(i).to_string();
        vert_bin.push_back(vb);
        }
    return;
}
// Função que imprime um vetor de transições em fmt inteiro
// usada somente para fins de debug e na finalização
string listaTransitions (vector<int> vetrans, string tipo) {
int no = vetrans.size();
string texto = "";
cout<<tipo<<" length = "<<vetrans.size()<< " trans = [ ";
texto = to_string(vetrans.size()) + " " + tipo + " trans = [ ";
for(int i=0; i < no; i++)
     if (i == no-1)
    {
        cout<<vetrans[i]<< "";
        texto = texto + to_string(vetrans[i]) + "";
    }
    else
    {
        cout<<vetrans[i]<< ", ";
        texto = texto + to_string(vetrans[i]) + ", ";
    }
    cout<<" ]"<<endl;
    texto = texto + " ]";
    return texto;
}
```

Classe Transition - Funções - Parte 4

arquivo: transitionClass.cpp

```cpp
// Função que coloca no array viz_bin em fmt string binário,
// os vizinhos de um vértice string v de entrada, colocando
// o string "-1" quando o vértice do vizinho estiver em
// posição indisponível em vert_bin (espaço de busca), e
// também retorna a quantidade de vizinhos disponíveis do vértice v
// no hipercubo.
int vizinhos_spa(string v){
    string vx;
    viz_bin_ind.clear();
    int qtd_viz = 0;
    for (int i = 0; i < SNAKE_DIMENSION; i++)
    {
        vx = v;
        if  (vx[i] == '0')      // modifica um char de cada vez
            vx[i] = '1';        // se '0' troca para '1'
        else
            vx[i] = '0';        // se '1' troca para '0'

        // converte vértice atual vert_bin[i] para (fmt int) x
        int zin = bin2int(vx);
        // verifica se vértice do vizinho atual está em posição proibida

        if ( vert_bin[zin] == "-1")
            {
            // se vértice indisponível e não faz parte dos vértices da snake
            // então éum vértice candidato a ser um nó de pele (skin node)
            if (std::count(vert_bins.begin(), vert_bins.end(), vx) == 0)
                viz_bin_ind.push_back(vx);
            viz_bin[i] = "-1";
            }
        else
            {
            viz_bin[i] = vx; // guarda vizinho alcançável no array viz_bin
            qtd_viz += 1;    // conta vizinhos alcançáveis em qtd_viz
            }
    }
    return qtd_viz;
}
```

Classe `Transition` - Funções - Parte 5

arquivo: `transitionClass.cpp`

```cpp
// Função retorna o tamanho do vetor vert_comp que representa a qtd de vértices
// alcançáveis a partir de vorig (vértice da cabeça da snake sendo processada).
// Computa os vértices de pele (skin nodes) na variável global new_skin_fit.
int find_alcance(std::string vorig)
{   int qtd_viz;  // qtd vértices vizinhos do vértice sendo processado
    int ac_viz_zero = 0;          // contador de vértices Dead End Nodes
    new_skin_fit = 0; // global onde se calcula a qtd total de vértices skin
    vert_comp.clear();             // global (tamanho final é a medida de fitness
    vert_comp.push_back(vorig); // inicializa com vértice cabeça da snake
    // Cria um unordered set (vazio) p/checar se vértice já está em vert_comp
    std::unordered_set<std::string> visit_vert_comp;
    visit_vert_comp.insert(vorig);
    // Cria um unordered set (vazio) p/checar se vértice já computado em skin_fit
    std::unordered_set<std::string> visit_skin_fit;
    // LOOP PRINCIPAL - Processando vert_comp[k]
        for (int k = 0; k < vert_comp.size(); k++)
    {
        // Processando vert_comp[k]
        // acha vizinhos da cabeça ou de qq outro vértice alcançável
        qtd_viz = vizinhos_spa(vert_comp[k]);
        // Rotina para apurar a fitness
        // loop para processar os vizinhos de vert_comp[k]
        for (int i = 0; i < SNAKE_DIMENSION; i++)
        {   // verifica se viz_bin[i] é um Dead End Node e não conta para fitness
            if (qtd_viz == 0)
               {   ac_viz_zero += 1; // contador de vértices Dead End Nodes
                   break; }
            if (viz_bin[i] == "-1")
               continue; // se vizinho é inalcançável -> avança para o próximo
            // Verifica se vértice alcançável em viz_bin[i] já foi visitado
            if (visit_vert_comp.find(viz_bin[i]) == visit_vert_comp.end())
            {   // se viz_bin[i] não está em vert_comp[]
               vert_comp.push_back(viz_bin[i]);  // insere viz_bin[i] em vert_comp[]
               visit_vert_comp.insert(viz_bin[i]); // marca como visitado
            }
        }
        // Rotina para apurar a new_skin_fit
        for (int j = 0; j < viz_bin_ind.size(); j++)
        {   if (vert_comp[k] == vorig)
               break;
            // verifica se viz_bin_ind[j] ainda não foi visitado e computado como nó de pele
            if (visit_skin_fit.find(viz_bin_ind[j]) == visit_skin_fit.end())
            {  new_skin_fit += 1; // se ainda não visitado soma 1 a skin_fit
               visit_skin_fit.insert(viz_bin_ind[j]);} // marca viz_bin_ind[j] como visitado
        }
    }          // FIM DO LOOP PRINCIPAL
// PREPARA PARA RETORNO - Cálculo do fitness: se tamano_fitness for negativo retorna ZERO
        int tamanho_fitness = vert_comp.size() - 1 - ac_viz_zero;
        if (tamanho_fitness < 0)
            tamanho_fitness = 0;    // Garante que o valor não seja negativo
        return tamanho_fitness; // Retorna o valor de fitness ou 0, caso seja negativo
        // retorna a quantidade de vértices alcançáveis (fitness)
        // menos a cabeça da snake e menos os vértices DEAD END NODES
}       // e também os vértices de skin na global new_skin_fit
```

Classe `Transition` - Parte 6: Definição dos Atributos

arquivo: `transitionClass.cpp`

```
// A classe Transition define os objetos que representam as snakes.
// Cada snake éum caminho induzido num grafo da família dos hipercubos.
// Para compactar a representação das snakes encontradas, usa-se
// comumente, representar as snakes com a notação da sequência das transições
// dos vértices expressos na base 2. As transicoes são expressas na base 10.
// A estrutura de dados escolhida para as snakes éuma lista encadeada
// de objetos da Classe Transition, representado uma arvore especial
// Cada snake éum caminho da árvore. Em particular, todos os caminhos nessa
// árvore tem a mesma origem comum no vértice 0 e têm dois vértices terminais
// que tem grau 1, enquanto todos os outros vértices internos têm grau 2.
// Cada objeto Transition aponta para o seu objeto pai (father).

class Transition {
    public:
        Transition * father;
        int transition;
        int max_seen;
        int fitness;
        int skin_fit;

        Transition (Transition * father_in, int transition_in,
                    int max_seen_in, int fitness_in, int skin_fit_in)
        {
            father     = father_in;
            transition = transition_in;
            max_seen   = max_seen_in;
            fitness    = fitness_in;
            skin_fit     = skin_fit_in;
        }

        // Sobrepõe o operador "<" para fins de comparação
        // Min-heap: classificada em ordem ascendente de fitness e de skin_fit
        bool operator<(const Transition& other) const {
            if (fitness != other.fitness) {
                return fitness > other.fitness; // Ordem crescente de fitness
            }
            return skin_fit > other.skin_fit; // Ordem crescente de skin_fit se
                fitness for igual
        }
```

Classe `Transition` - Parte 7: Métodos

arquivo: `transitionClass.cpp`

```cpp
// Método transition_sequence - como chamar: v->transition_sequence()
// retorna o vetor trans com toda a sequência de transições da snake
vector<int> transition_sequence() {
    vector<int>result;
    result.push_back(transition);
    Transition * nextfather = father;
    while (nextfather != nextfather->father)
    {
        result.push_back(nextfather->transition);
        nextfather = nextfather->father;
    }
    vector<int> newvector (result.rbegin(), result.rend());
return newvector;
}
// Método is_snake recebe a próxima transição em transx e o vértice cabeça
// da snake em v. Devolve "true" se após o acréscimo de transx, o caminho
// continuar a ser uma snake. Calcula new_fitness e new_skin_fit as
// duas medidas de adequação de cada nova snake estendida
bool is_snake(int transx, Transition * v) {
// variável que recebe valor decimal inteiro de cada vértice em fmt string binário
int verint;   // para poder marcar vértices indisponíveis no espaço de busca ( vetor vert_bin em fmt string binário)
// Prepara vetor trans c/ transições snake anterior + nova transição (a testar)
trans.clear();
trans = v->transition_sequence();
trans.push_back(transx);
// transforma o vetor de transições trans p/ vetor vert_bins em fmt string binário
trans2vertb(trans);
vert_bin = vert_bin_fixo;      // inicializa vert_bin com vértices do cubo em fmt binário
for (int i = 1; i < vert_bins.size(); i++)
{   // converte vértice atual vert_bin[i] para (fmt int) verint
    verint = bin2int(vert_bins[i]);
    // verifica se vértice atual está em posição proibida
    if ( vert_bin[verint] == "-1")
    {
        return false;
    }
    vizinhos(vert_bins[i-1]);
    for (int j=0; j < SNAKE_DIMENSION; j++)
    {
        // converte vértice vizinho atual viz_bin[i] para (fmt int) verint
        verint = bin2int(viz_bin[j]);
        // marca indisponibilidade em todos os vértices vizinhos do
        // vértice anterior da snake inclusive aquele de transx
        vert_bin[verint] = "-1";
    }
}
// atualiza snake_length para o novo tamanho da snake corrente
snake_length = trans.size();
string vorig = vert_bins[vert_bins.size() - 1]; // atribui a vorig cabeça da snake corrente
new_fitness = find_alcance(vorig);    // vai calcular new_fitness e new_skin_fit
return true;
}
};  // FIM DA CLASSE Transition
```

Stochastic Beam Search (SBS)

Parte 1: Includes e Definição de Globals

arquivo: `beam.cpp`

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <bits/stdc++.h>
#include <stddef.h>
#include <vector>
#include <queue>
#include <algorithm>
#include <time.h>
#include <chrono>
#include <unistd.h>
#include <unordered_set>
#include "def.h"
using namespace std;
#include "transitionClass.cpp" // inclui a Classe Transition e Funções Auxiliares
// VARIÁVEIS E ESTRUTURAS GLOBAIS
int new_fitness;  // total de vértices alcançáveis de uma nova subsolução de snake
int new_skin_fit; // total de vértices de pele (skin nodes) de uma nova subsolução de snake
// valor máximo e valor mínimo de fitness a cada extensão do beam ( tamanho t para t+1)
int max_fit;
int min_fit;
// valor máximo e valor mínimo de skin_fit a cada extensão do beam ( tamanho t para t+1)
int min_skinfit;
int max_skinfit;
// Inteiros gerados aleatóriamente entre 1 e 100 para nossa rotina de randomicidade
int irand1;
int irand2;
// Parâmetros de randomização variáveis ajustados a cada corrida definidos em def.h
int def_irand1 = TXRAND1 + 1;
int def_irand2 = TXRAND2 + 1;
// Totais de substituições feitas ao acaso (irand1 x def_irand1) ou (irand2 x def_irand2)
int tot_rand1 = 0;
int tot_rand2 = 0;
// variáveis auxiliares
Transition *snake;       // variável para guardar o ponteiro para um node snake
int debugar = DEBUGAR; // variável para controle de debug (executar ou não instruções de cout's<< ...)
string nomeArq = "";  // variável para colocar o nome do arquivo onde serão gravados os resultados do processamento
// variáveis de tamanho para controle do beam e definição do array beam
int ind_beam = 0; // ind_beam éo tamanho do beam na iteração atual (pode ser menor do que o máximo)
int max_beam      =      SIZE_BEAM;
int ind_beam_anter; // ind_beam_anter éo tamanho do beam na iteração anterior (pode ser menor do que o máximo)
int ind_beam1;     // ind_beam1 éonde salvo temporariamente o tamanho do beam na iteração atual
Transition * beam     [SIZE_BEAM]; // definição do array beam
// variáveis de tamanho para controle do new_beam e definição da min-heap new_beam
int ind_new_beam; // ind_new_beam éo tamanho do new_beam da iteração atual (pode ser menor do que o máximo)
int max_new_beam =      SIZE_NEW_BEAM; // tamanho máximo do new_beam
// Define vetor new_sequences onde serão colocados os ponteiros para as extensões de uma das snakes do beam
vector<Transition *> new_sequences;
const size_t MAX_NEW_SEQUENCES_SIZE = SNAKE_DIMENSION - 2;
// Define vetor vetortran onde serão colocados os ponteiros para as extensões de uma das snakes do beam
vector<int> vetortran;
// Define new_beam a priority queue de ponteiros para objetos da Classe Transition onde vamos tentar colocar todas as
// extensões das snakes de todas as new_sequences geradas
struct TransitionPtrComparator {
    bool operator()(const Transition* p1, const Transition* p2) const {
        return *p1 < *p2;   //  menor fitness no topo de new_beam
    }
};
priority_queue<Transition*, vector<Transition*>, TransitionPtrComparator> new_beam;
// ver na Classe Transition a sobreposição do operador para transformar a max-heap em min-heap
// variáveis para controlar as quantidades de nodes gerados e deletados
int del_newseq_newbeam = 0; // nodes deletados por não aproveitamento ao passar de newsequ para new_beam
// nodes deletados ao reduzir tamanho de new_beam antes colocar as subsoluções de new_beam em beam
int del_newbeam_beam = 0;
int nodes_del;              // total geral de nodes deletados
int pool_nodes = 4;         // total de nodes gerados
```

SBS - Parte 2: Funções Auxiliares

arquivo: beam.cpp

```
// Reserva o espaço máximo do vetor new_sequences e do vetor
// vetortran para evitar realocações visto que estes vetores
// têm tamanhos dinâmicos
void init_new_sequences_global(){
new_sequences.reserve(MAX_NEW_SEQUENCES_SIZE);
vetortran.reserve(MAX_NEW_SEQUENCES_SIZE);
}

// Gera número inteiro aleatório a cada chamada
int numeroAleatorio(int menor, int maior) {
      return rand()%(maior-menor+1) + menor;
}

// Esta função imprime na console a maior snake obtida
void finaliza(Transition * v){
    trans.clear();
    trans = v->transition_sequence();
    string texto;
    texto = listaTransitions(trans, "LONGEST SNAKE ");
    }
```

SBS - Parte 3: Função increase

arquivo: `beam.cpp`

```
// Esta função recebe um ponteiro ptrans para a transição do vértice da cabeça
// de uma snake de tamanho t a ser estendida em outras snakes de tamanho t + 1.
// Cada ponteiro no vetor de retorno new_sequences representa as snakes estendidas
// a partir de uma subsolução anterior que estava no array beam. Para cada snake
// estendida cria-se um novo objeto node da classe Transition. Ao tentar criar o node
// da extensão, é obedecida a regra para criar snakes na forma canônica: a transição
// do node sendo criado só pode ser uma unidade maior do que as transições anteriores
// dos nodes precedentes na snake. Os próximos nodes criados não podem ter também
// transições iguais às 2 antecedentes trans_current ou trans_previous.
void increase(Transition * ptrans) {
new_sequences.clear();                     // limpa vetor new_sequences a ser usado
vetortran.clear();                         // limpa vetor vetortran    a ser usado
int vmaxseen =     ptrans->max_seen; // coloca o maior valor de transição até então
                                           // na snake base a ser estendida para garantir
                                           // que as novas transições irão ser maiores
                                           // do que a maior das anteriores
int trans_current  = ptrans->transition; // coloca em trans_current a atual transição da subsolução
int trans_previous = ptrans->father->transition; // trans_previous = transição anterior da subsolução
int next_dimension = min(vmaxseen+1, MAX_DIMENSION); // define limite máximo próximas transições
// Rotina que cria as próximas transições possíveis para estender a subsolução apontada em ptrans
for (int i  = 0; i  <= next_dimension; i++)
    {
            if ( i != trans_current and i != trans_previous)
                vetortran.push_back(i);
    }
// Rotina que pega cada nova transição possível em vetortran, chama o método issnake da Classe Transition
// para verificar se é uma snake, e se for cria um novo node da Classe Transition para apontar para a
// nova snake recém criada
for (int i=0; i < vetortran.size(); i++)
    {
        int transx = vetortran[i];         // coloca na variável auxiliar transx a nova transição a ser verificada
        bool issnake = ptrans->is_snake(transx, ptrans); // chama método is_snake p/ verificar se éuma snake
        // se nova transx for uma snake então, atualiza a maior transição até esta nova snake em next_seen,
        //  cria um novo node da Classe Transition e adiciona o ponteiro (u) para ele no vetor new_sequences
        if (issnake)
            {
            int next_seen = max(vetortran[i], vmaxseen); // next_seen = maior transição até esta nova snake
            // cria novo node    Transition (&father, transition, max_seen, fitness,      skin_fit)
            Transition * u = new Transition( ptrans, transx,      next_seen,new_fitness, new_skin_fit);
            new_sequences.push_back(u);
            pool_nodes += 1;
            }
    }
}
```

SBS - Parte 4: Função increase_beam

arquivo: `beam.cpp`

```
// Função para estender todas as snakes no array beam[], chamando a função increase para estender cada
// uma das snakes (tamanho t) retornando as snakes estendidas (tamanho t+1) no vetor new_sequences. Para
// cada vetor // new_sequences recebido a increase_beam pode: 1- se todas as novas snakes em new_sequences
//  couberem em new_beam adiciona-as a new_beam e torna a chamar a função increase.
//  2- se a quantidade máxima de snakes no min-heap new_beam já tiver sido atingida, esta rotina
// faz a seleção e decide se vai usar a nova snake estendida em new_sequences para substituir o topo da min-heap
// new_beam ou se vai descartar a nova snake criada em new_sequences. Ao decidir descartar uma das 2 snakes (a de
// new_sequences ou anterior do topo da min-heap new_beam), a função atualiza os contadores de nodes da forma adequada.
void increase_beam()
{   srand((unsigned)time(NULL));    // para gerar números aleatórios reais
    // VARRE TODO O BEAM PARA TENTAR O CRESCIMENTO DAS SNAKES
    for (int i = 0; i < max_beam; i++)
    {   // SE BEAM INCOMPLETO ( FOI ATINGIDO BEAM[I] = ZERO) INTERROMPE A INCREASE_BEAM
        if (beam[i] == 0)
            break;
        increase(beam[i]);  // chama a função increase
        // retorna da função increase com as snakes de tamanho t+1 em new_sequences
        if (new_sequences.size() == 0) // se não houve nenhuma snake estendida de tamanho t+1 volta ao loop for para
            continue;                   // chamar a increase  para estender a próxima snake do array beam[]
            for (int j=0; j < new_sequences.size();j++) // loop para processar as novas snakes de tamanho t+1
            {   // se ainda não foi atingido o tamanho máximo de new_beam vamos anexar a nova snake no min-heap
                if (ind_new_beam < max_new_beam) // new_beam e atualizar as variáveis necessárias
                {   // insere nova snake de new_sequences corrente em
                new_beam.emplace(new_sequences[j]); // new_beam mantendo a snake de menor fitness no topo de new_beam.
                ind_new_beam +=1;                   // Atualiza o índice que mostra o tamanho atual de new_beam
                // atualiza as variáveis com as informações de máximos e mínimos do ciclo da increase_beam
                max_fit = max(max_fit, new_sequences[j]->fitness);
                min_fit = min(min_fit, new_sequences[j]->fitness);
                max_skinfit = max(max_skinfit, new_sequences[j]->skin_fit);
                min_skinfit = min(min_skinfit, new_sequences[j]->skin_fit);}
                // ind_new_beam >= max_new_beam. Não consegui colocar em new_beam todas as snakes criadas na increase
                else
                {   // obtendo em px o topo atual do min-heap para poder decidir se o topo atual será, mais adiante
                    Transition* px = new_beam.top();  // substituído pela snake apontada p/new_sequences[j]
                    // Gera número aleatório irand1 entre 1 e 100 para ver se vamos trocar o topo da min-heap por uma
                    irand1 = numeroAleatorio(1, 100);   //  snake qualquer, fazemos isso para fugir de snakes máximas locais
                    // se vamos trocar o topo por uma snake qq ao acaso soma 1 a tot_rand1
                    if (irand1 < def_irand1)
                        tot_rand1 += 1;
                    // Gera número aleatório irand2 entre 1 e 100 para ver se vamos
                    // trocar o topo da min-heap por uma snake nova que tem as
                    // mesmas medidas de adequação da snake que está no topo
                    irand2= numeroAleatorio(1, 100);
                    //  verifica se vamos trocar o topo por uma snake qq ao acaso
                    if  ( (irand1 < def_irand1) or
                    // ou se vamos trocar o topo por uma snake c/ mesma fitness e mesma skin_fit
                        ( (irand2 < def_irand2) and
                          (new_sequences[j]->fitness == px->fitness and new_sequences[j]->skin_fit == px->skin_fit) ) or
                    // ou se vamos trocar o topo por uma snake com melhor fitness
                        (new_sequences[j]->fitness > px->fitness) or
                    // ou se vamos trocar o topo por uma snake com igual fitness
                    // mas que tem melhor skin_fit do que a snake no topo do min-heap
                        (new_sequences[j]->fitness == px->fitness and new_sequences[j]->skin_fit > px->skin_fit) )
                    // caso uma das 4 condições seja verdadeira, vamos descartar a snake do topo de new_beam e vamos
                    // substituí-la pela nova snake em new_sequences
                    {    new_beam.pop();
                        delete(px);
                        if ( (irand2 < def_irand2)  and
                            (new_sequences[j]->fitness == px->fitness and new_sequences[j]->skin_fit == px->skin_fit
                                and irand1 >= def_irand1 ) )
                            tot_rand2 += 1;
                        del_newseq_newbeam += 1;
                        new_beam.emplace(new_sequences[j]);  // insere nova snake de new_sequences corrente em new_beam
                                                             // mantendo a snake de menor fitness no topo de new_beam.
                        // atualiza as variáveis com as informações de máximos e mínimos do ciclo da increase_beam
                        max_fit = max(max_fit, new_sequences[j]->fitness);
                        min_fit = min(min_fit, new_sequences[j]->fitness);
                        max_skinfit = max(max_skinfit, new_sequences[j]->skin_fit);
                        min_skinfit = min(min_skinfit, new_sequences[j]->skin_fit);}
                    else    { // se nenhuma das 4 condições for verdadeira
                    delete(new_sequences[j]);  //então descarta a nova snake em new_sequences
                    del_newseq_newbeam += 1; // e atualiza o contador de novas snakes em new_sequences deletadas
                    } } } } }
```

SBS - Parte 5: Programa Principal main()

arquivo: `beam.cpp`

```
// PROGRAMA PRINCIPAL
int main(int argc, char * argv[]) {
// obtem dimensão do cubo e se for < 4 encerra com as soluções triviais
int snake_dimension = int(SNAKE_DIMENSION);
 if (snake_dimension == 1) cout << "LONGEST SNAKE 1 OF 1 length = 1 trans = [ 0 ]" << endl;
 if (snake_dimension == 2) cout << "LONGEST SNAKE 1 OF 1 length = 2 trans = [ 0, 1]" << endl;
 if (snake_dimension == 3) cout << "LONGEST SNAKE 1 OF 1 length = 3 trans = [ 0, 1, 2]" << endl;
 if (snake_dimension < 4) exit(0);
init_new_sequences_global();          // reserva espaço máximo para vetores new_sequences e vetortran
ini_vert_bin(vert_bin_fixo, vertot);    // inicializa vetor vert_bin_fixo
// Cria o primeiro node nf cujo pai (father) aponta para ele mesmo e depois
// cria os 3 primeiros nodes de uma snake, que representam a primeira snake de
// tamanho 3 de um cubo (de qualquer dimensão > do que 3), na forma canônica snake = [0, 1, 2]
Transition nf=Transition(&nf, 0,0,0,0);    // nf nó fantasma que aponta para ele mesmo (NULL)
Transition n0=Transition(&nf, 0,0,1,0);    // n0 (root) aponta para nó fantasma nf
Transition n1=Transition(&n0, 1,1,1,0);    // n1 aponta para o n0 (root)
Transition n2=Transition(&n1, 2,2,1,0);    // n2 aponta para o n1
// Transition (&father, transition, max_seen, fitness, skin_fit)
// cria o array beam (de pointers para as cabeças das snakes) com apenas uma snake S = [0,1,2]
beam[0] = &n2;
// Loop tentando estender todas as snakes de tamanho t no
// array beam[] para tamanho t+1 em new_beam, chamando a função increase_beam
while (true){
     // inicializa as variáveis de controle do loop
     ind_new_beam = 0; // ind_new_beam éo tamanho do new_beam da iteração atual
     max_fit = 0;
     min_fit = 999999999;
     max_skinfit = 0;
     min_skinfit = 999999999;
// Chama a função increase_beam que vai processar as snakes no array global beam[]
     increase_beam();
// ##### FINALIZA AQUI O PROGRAMA SE NEW_BEAM ESTIVER VAZIO POIS
//  NÃO HOUVE CRESCIMENTO DE NENHUMA SNAKE DO BEAM ANTERIOR !!! #####
     if (ind_new_beam == 0) { // se não houve nenhum crescimento -> finaliza com as snakes em beam[]
          snake = beam[0];
          finaliza (snake);
          exit(0);}
// Caso new_beam não esteja vazio, houve algum crescimento, então
// executa a rotina para copiar snakes de new_beam para o próximo beam
     ind_beam = 0; //   inicializa ind_beam com 0 (o indice do beam)
// Rotina que copia todas as snakes de new_beam para beam até o limite máximo = max_beam
// Primeiramente se o tamanho de new_beam for maior do que o máximo possível do array beam[]
// no loop while vamos eliminar todas as snakes de menor fitness de new_beam de forma a ficarem
// em new_beam somente as snakes de maiores fitnesses que couberem no array beam.
// No corpo do while vamos eliminando o topo de new_beam e deletando os nodes correspondentes
// para limpar e conseguir otimizar a memória principal.
     if (new_beam.size() > max_beam) {
               while (new_beam.size() != max_beam)
                    {
                    Transition* px = new_beam.top();
                    new_beam.pop();
                    delete(px);
                    del_newbeam_beam += 1;
                    } }
// No corpo do loop while abaixo vamos copiar as snakes de new_beam para beam[]
// e atualizamos o ind_beam somando 1 a cada nova snake copiada
     while (!new_beam.empty()) {
          Transition* px = new_beam.top();
          beam[ind_beam] = px;
          new_beam.pop();
          ind_beam += 1;}
// Nesta rotina verificamos se o beam ficou incompleto pois existiam
// menos snakes em new_beam do que a capacidade máxima de beam.
// Neste caso temos que zerar os elementos restantes do array beam
// pois podem estar com snakes do ciclo anterior
     ind_beam1 = ind_beam;
     while (ind_beam1 < max_beam) {
               if (beam[ind_beam1] == 0)         // zera o restante do beam (até o final)
                    break;
               beam[ind_beam1] = 0;
               ind_beam1++;}                        // Fim da Rotina beam = new_beam
}              // final do while (true)
return 0; } // final do main()
```

Referências Bibliográficas

[1] BONDY, J. A., MURTY, U. S. R., OTHERS. *Graph theory with applications*, v. 290. University of Waterloo, Ontario, Canada, Macmillan London, 1976.

[2] HAMMING, R. W. *Coding and information theory.* New York, Prentice-Hall, Inc., 1986.

[3] ZHANG, Y., GE, G. "Snake-in-the-Box Codes for Rank Modulation", *IEEE Transactions on Information Theory*, v. 62, n. 1, pp. 151–158, 2015.

[4] BUND, J. "Hazard-free clock synchronization", *Servidor Científico da Universidade de Saarland*, 2022.

[5] DRAPELA, T. E. *The Snake-in-the-Box Problem: a primer*. Tese de Doutorado, Citeseer, 2015.

[6] GAREY, M. R., JOHNSON, D. S. *Computers and intractability*, v. 174. Murray Hill, New Jersey, USA, freeman San Francisco, 1979.

[7] ISHIZEKI, T., OTACHI, Y., YAMAZAKI, K. "An improved algorithm for the longest induced path problem on k-chordal graphs", *Discrete applied mathematics*, v. 156, n. 15, pp. 3057–3059, 2008.

[8] JAFFKE, L., KWON, O., TELLE, J. A., et al. "Polynomial-Time Algorithms for the Longest Induced Path and Induced Disjoint Paths Problems on Graphs of Bounded Mim-Width". In: *12th International Symposium on Parameterized and Exact Computation (IPEC 2017)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2018.

[9] KRATSCH, D., MÜLLER, H., TODINCA, I. "Feedback vertex set and longest induced path on AT-free graphs". In: *Graph-Theoretic Concepts in Computer Science: 29th International Workshop, WG 2003. Elspeet, The Netherlands, June 19-21, 2003. Revised Papers 29*, pp. 309–321. Springer, 2003.

[10] KAUTZ, W. H. "Unit-distance error-checking codes", *IRE Transactions on Electronic Computers*, v. 1, n. 2, pp. 179–180, 1958.

[11] HAMMING, R. W. "Error detecting and error correcting codes", *The Bell system technical journal*, v. 29, n. 2, pp. 147–160, 1950.

[12] GRAY, F. "Pulse code communication", *United States Patent Number 2632058*, 1953.

[13] DORAN, R. W. *The gray code.* Relatório técnico, Citeseer, 2007.

[14] TRAKHTENBROT, B. A. "A survey of Russian approaches to perebor (brute-force searches) algorithms", *Annals of the History of Computing*, v. 6, n. 4, pp. 384–400, 1984.

[15] VASIL'EV, Y. L. "On comparing the complexity of typical and disjunctive normal forms", *Problemy Kibernet*, v. 10, pp. 5–61, 1963.

[16] GUREVICH, I., ZHURAVLEV, Y. I. "Minimization of Boolean functions and effective recognition algorithms", *Cybernetics*, v. 10, n. 3, pp. 393–397, 1974.

[17] EVDOKIMOV, A. A. "Maximal length of circuit in a unitary n-dimensional cube", *Mathematical notes of the Academy of Sciences of the USSR*, v. 6, pp. 642–648, 1969.

[18] EMELYANOV, P. G., LUKITO, A. "On the maximal length of a snake in hypercubes of small dimension", *Discrete Mathematics*, v. 218, n. 1-3, pp. 51–59, 2000.

[19] KAUTZ, W. *Codes and coding circuitry for automatic error correction within digital systems technical report no. 2.* Relatório técnico, NASA, 1962.

[20] MEYERSON, S. J., DRAPELA, T. E., WHITESIDE, W. E., et al. "Finding longest paths in hypercubes: 11 new lower bounds for snakes, coils, and symmetrical coils". In: *Current Approaches in Applied Artificial Intelligence: 28th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2015, Seoul, South Korea, June 10-12, 2015, Proceedings 28*, pp. 23–32. Springer, 2015.

[21] RAMANUJACHARYULU, C., MENON, V. "A note on the snake-in-the box problem". In: *Annales de l'ISUP*, v. 13, pp. 131–135, 1964.

[22] ABBOTT, H. "A Note on the Snake-in-the-Box Problem". In: *Some Problems in Combinatorial Analysis*, University of Alberta, Edmonton, Canada, Edmonton, Alberta, Canada, 1965.

[23] DANZER, L., KLEE, V. “Lengths of snakes in boxes”, *Journal of Combinatorial Theory*, v. 2, n. 3, pp. 258–265, 1967.

[24] ABBOTT, H. L., KATCHALSKI, M. “On the construction of snake in the box codes”, *Utilitas Mathematica*, v. 40, pp. 97–116, 1991.

[25] SOLOV’JEVA, F. “An upper bound for the length of a cycle in an n-dimensional unit cube”, *Diskret. Analiz*, v. 45, pp. 71–76, 1987.

[26] ZÉMOR, G. “An upper bound on the size of the snake-in-the-box”, *Combinatorica*, v. 17, pp. 287–298, 1997.

[27] BROWN, W. J. *An iterated local search with adaptive memory applied to the snake in the box problem.* Tese de Doutorado, University of Georgia, 2005.

[28] POTTER, W. D., ROBINSON, R. W., MILLER, J. A., et al. “Using the Genetic Algorithm to Find Snake-in-the-Box Codes.” In: *IEA/AIE*, pp. 421–426. Citeseer, 1994.

[29] ATLURI, K. *Snake-in-the-box Problem Using Nature Inspired Search.* Tese de Doutorado, University of Georgia, 2009.

[30] DANG, C., BAZGAN, C., CAZENAVE, T., et al. “Warm-starting nested rollout policy adaptation with optimal stopping”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*, v. 37, pp. 12381–12389, 2023.

[31] HARDAS, S. P. *An ant colony approach to the snake-in-the-box problem.* Tese de Doutorado, University of Georgia, 2005.

[32] DAVIES, D. W. “Longest“Separated”Paths and Loops in an N Cube”, *IEEE Transactions on Electronic Computers*, , n. 2, pp. 261–261, 1965.

[33] CARLSON, B. P., HOUGEN, D. F. “Phenotype feedback genetic algorithm operators for heuristic encoding of snakes within hypercubes”. In: *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pp. 791–798, 2010.

[34] WYNN, E. “Constructing circuit codes by permuting initial sequences”, *arXiv preprint arXiv:1201.1647*, 2012.

[35] KOCHUT, K. J. “Snake-in-the-box codes for dimension 7”, *Journal of Combinatorial Mathematics and Combinatorial Computing*, v. 20, pp. 175–185, 1996.

[36] ÖSTERGÅRD, P. R., PETTERSSON, V. H. "On the maximum length of coil-in-the-box codes in dimension 8", *Discrete Applied Mathematics*, v. 179, pp. 193–200, 2014.

[37] ALLISON, D., PAULUSMA, D. "New bounds for the snake-in-the-box problem", *arXiv preprint arXiv:1603.05119*, 2016.

[38] BLACK, W. "Electronic combination locks", *Quarterly Progress Report of the Research Laboratory of Electronics*, , n. 73, pp. 232–233, 1964.

[39] HOLLAND, J. H. "Adaptation in Natural and Artificial Systems. ann arbor: University of michigan press", *Ann Arbor: The University of Michigan Press*, 1975.

[40] OMARA, F. A., ARAFA, M. M. "Genetic algorithms for task scheduling problem", *Journal of Parallel and Distributed computing*, v. 70, n. 1, pp. 13–22, 2010.

[41] KNUTH, D. E. *The art of computer programming, volume 4A: combinatorial algorithms, part 1.* New Delhi, Pearson Education India, 2011.

[42] ROSIN, C. D. "Nested rollout policy adaptation for Monte Carlo tree search". In: *Ijcai*, v. 2011, pp. 649–654, 2011.

[43] ABBOTT, H. L., KATCHALSKI, M. "On the snake in the box problem", *Journal of Combinatorial Theory, Series B*, v. 45, n. 1, pp. 13–24, 1988.

[44] PATERSON, K. G., TULIANI, J. "Some new circuit codes", *IEEE Transactions on Information Theory*, v. 44, n. 3, pp. 1305–1309, 1998.

[45] MEYERSON, S. *Finding longest paths in hypercubes: 11 new lower bounds for snakes, coils, and symmetrical coils.* Dissertação de M.Sc., Department of Computer Sciences/University of Georgia, Atlanta, Georgia, USA, 2015.

[46] LOWERRE, B. T. *The HARPY Speech Recognition System.* Tese de D.Sc., Department of Computer Science/Carnegie-Mellon University, Pittsburgh, Pennsylvania, USA, 1976.

[47] RUBIN, S. M., REDDY, R. "The locus model of search and its use in image interpretation", *Cambridge, Massachusetts*, pp. 590–595, 1977.

[48] BAHDANAU, D. "Neural machine translation by jointly learning to align and translate", *arXiv preprint arXiv:1409.0473*, 2014.

[49] WU, K.-C., TING, C.-J., LAN, W. "A Beam Search Heuristic for the Traveling Salesman Problem with Time Windows", *Journal of the Eastern Asia Society for Transportation Studies*, v. 9, pp. 702–712, 2011. Disponível em: <`https://api.semanticscholar.org/CorpusID:60885667`>.

[50] GUPTA, P., DOERMANN, D., DEMENTHON, D. "Beam search for feature selection in automatic SVM defect classification". In: *2002 International conference on pattern recognition*, v. 2, pp. 212–215. IEEE, 2002.

[51] LI, L., PAGNUCCO, M., SONG, Y. "Graph-based spatial transformer with memory replay for multi-future pedestrian trajectory prediction". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 2231–2241, 2022.

[52] HUBER, M., RAIDL, G. R. "Learning beam search: Utilizing machine learning to guide beam search for solving combinatorial optimization problems". In: *International Conference on Machine Learning, Optimization, and Data Science*, pp. 283–298. Springer, 2021.

[53] HUANG, L., ZHANG, H., DENG, D., et al. "LinearFold: linear-time approximate RNA folding by 5'-to-3'dynamic programming and beam search", *Bioinformatics*, v. 35, n. 14, pp. i295–i304, 2019.

[54] ADELSON, L. E., ALTER, R., CURTZ, T. B. *Long snakes and a characterization of maximal snakes on the d-cube.* University of Kentucky, Lexington, Kentucky, USA, University of Kentucky, 1973.

[55] MCKAY, B. D. "Isomorph-free exhaustive generation", *Journal of Algorithms*, v. 26, n. 2, pp. 306–324, 1998.

[56] FUIDIO, F., CANALE, E., SOTELO, R. "QUBO formulation for the Snake-in-the-box and Coil-in-the-box problems", *arXiv preprint arXiv:2409.04476*, 2024.

[57] SANTORO, G. "snake_in_the_box". `https://github.com/girsant/snake_in_the_box/`, 2024.