**UNIVERSIDADE FEDERAL DE ITAJUBÁ**

**INSTITUTO DE MATEMÁTICA E COMPUTAÇÃO**

**FLÁVIO MOTA GOMES - 2018005379**

**RAFAEL ANTUNES VIEIRA - 2018000980**

**RAFAEL GRECA VIEIRA - 2018000434**

**PROJETO - PARADIGMAS DE PROGRAMAÇÃO**

**ITAJUBÁ**

**NOVEMBRO / 2020**

**UNIVERSIDADE FEDERAL DE ITAJUBÁ**

**INSTITUTO DE MATEMÁTICA E COMPUTAÇÃO**

**FLÁVIO MOTA GOMES - 2018005379**

**RAFAEL ANTUNES VIEIRA - 2018000980**

**RAFAEL GRECA VIEIRA - 2018000434**

**PROJETO - PARADIGMAS DE PROGRAMAÇÃO**

Relatório apresentado à professora de Paradigmas de Programação, Dra. Isabela Neves Drummond, como requisito parcial para obtenção de nota para a disciplina de Paradigmas de Programação, do curso de Ciência da Computação da Universidade Federal de Itajubá.

**ITAJUBÁ**

**NOVEMBRO / 2020**

**SUMÁRIO**

[**1. HISTÓRICO DAS LINGUAGENS**](#_8adyebo7gp0n) **3**

[1.1. LINGUAGEM C](#_schamrg5ytk6) 3

[1.2. LINGUAGEM JAVA](#_387yckygb601) 3

[1.3. LINGUAGEM LISP](#_ljow11ogevyd) 4

[1.4. LINGUAGEM PROLOG](#_eu5rtcm4vsmt) 4

[**2. AMBIENTE DE PROGRAMAÇÃO**](#_mm6n1qlducz4) **4**

[**3. DESCRIÇÃO DA SOLUÇÃO PROPOSTA PARA CADA PROBLEMA**](#_si01b97t3y29) **5**

[3.1. QUEBRA-CABEÇA DE OITO](#_e1ashcrpcvdb) 5

[3.1.1. Linguagens C e Java](#_6rtel2o6at7q) 5

[3.1.2. Linguagens LISP e PROLOG](#_9tfvutnix3vf) 6

[3.2. OITO RAINHAS](#_lkoedjf4dy7u) 7

[3.3. O MACACO E AS BANANAS](#_xwnbccmi63q0) 8

[3.3.1. Linguagens C, Java e Lisp](#_8oteej9tl5ye) 8

[3.3.2. Linguagem PROLOG](#_2b75gfsbdvb4) 8

[**4. DESCRIÇÃO DA IMPLEMENTAÇÃO REALIZADA**](#_joi3vtwn6gi9) **8**

[4.1. QUEBRA-CABEÇA DE OITO](#_cw4542qd1524) 8

[4.1.1. Quebra-cabeça de oito em C](#_ap3qop2j1hy2) 9

[4.1.2. Quebra-cabeça de oito em JAVA](#_vp516la4wn52) 11

[4.1.3. Quebra-cabeça de oito em LISP](#_pow519z03dbm) 17

[4.1.4. Quebra-cabeça de oito em PROLOG](#_sf74bvjjafne) 18

[4.2. OITO RAINHAS](#_qxf4oh2084hi) 22

[4.2.1. Oito Rainhas em C](#_jeuneh31hlju) 22

[4.2.2. Oito Rainhas em JAVA](#_gqnhi76owz8v) 24

[4.2.3. Oito Rainhas em LISP](#_3wbwe9jws10n) 25

[4.2.4. Oito Rainhas em PROLOG](#_wchtrnheke2w) 26

[4.3. O MACACO E AS BANANAS](#_4aahx9qzekfv) 27

[4.3.1. O macaco e as bananas em C](#_5ociy33jlhzv) 27

[4.3.2. O macaco e as bananas em JAVA](#_tgl1h4vp823k) 30

[4.3.3. O macaco e as bananas em LISP](#_hqffiwxgrsgj) 30

[4.3.4. O macaco e as bananas em PROLOG](#_mff6prszrcee) 31

[**REFERÊNCIAS BIBLIOGRÁFICAS**](#_hvi69axyhfhe) **33**

# 1. HISTÓRICO DAS LINGUAGENS

A seguir, realiza-se um panorama das quatro linguagens de programação utilizadas neste trabalho, sendo elas: Linguagem C, Linguagem JAVA, Linguagem LISP, Linguagem Prolog.

## 1.1. LINGUAGEM C

Linguagem de programação compilada de propósito geral, estruturada, imperativa, procedural, padronizada pela Organização Internacional para Padronização (ISO), criada em 1972 por Dennis Ritchie na empresa AT&T Bell Labs para desenvolvimento do sistema operacional Unix (originalmente escrito em Assembly).

A Linguagem C foi derivada de outras duas, a BCPL e a Algol 68. Suporta os paradigmas estruturado, imperativo e procedural. Neste trabalho, utilizou-se a Linguagem C com o propósito de explorar o paradigma estruturado.

Trata-se de uma linguagem das mais populares do mercado, tendo se tornado muito popular ainda nos anos 1980. Tamanha popularidade torna difícil encontrar arquiteturas onde não haja compiladores para ela, o que a torna uma linguagem com alto nível de portabilidade.

Além disso, dentre as características da Linguagem C destacam-se a simplicidade, confiabilidade, facilidade de uso, entre outras. A Linguagem C influenciou a estrutura de outras linguagens, como C++ e C#.

## 1.2. LINGUAGEM JAVA

Para o paradigma de orientação a objetos, a linguagem escolhida foi a Linguagem Java. A linguagem foi desenvolvida nos anos 1990 nos laboratórios da Sun Microsystems.

No início, o Java foi desenvolvido para ser mais simples e eficiente e para estar em produtos eletrônicos de consumo, como agendas eletrônicas. Com a popularização da internet, seu código foi adaptado para ser utilizado em computadores conectados à internet, o que deu popularidade à linguagem.

A linguagem caracteriza-se por ser poderosa, versátil, segura, simples e eficiente. Além disso, seus códigos, executados em máquina virtual, a garantem portabilidade.

## 1.3. LINGUAGEM LISP

Representante do paradigma funcional neste trabalho, a linguagem LISP teve sua primeira implementação por John McCarthy, em 1958, a partir da ideia de desenvolver uma linguagem algébrica para processamento de listas para trabalho em IA. A motivação de McCarthy surgiu da ideia de desenvolver uma linguagem algébrica para processamento de listas para trabalho em inteligência artificial.

O nome LISP oriunda-se de “**LIS**t **P**rocessing”. De fato, a lista é a estrutura de dados fundamental da linguagem, já que os dados e o próprio programa são representados nessa estrutura, permitindo que a linguagem manipule o código fonte como qualquer outro tipo de dado.

Além do uso em aplicações de inteligência artificial, outras aplicações são AutoCad, Editor Emacs. O LISP puro não possui atribuições, o que o torna livre de efeitos colaterais. Além disso, não é compilado, mas sim interpretado.

Como curiosidade, há uma brincadeira que diz que LISP significa “**L**ot of **I**rritating **S**tupid **P**arenthesis” (muitos parênteses estúpidos e irritantes).

## 1.4. LINGUAGEM PROLOG

O nome Prolog para a linguagem concreta foi escolhido por Philippe Roussel como uma abreviação de “PROgrammation en LOGique”. Foi criada em meados de 1972 por Alain Colmerauer e Philippe Roussel, baseados no conceito de Robert Kowalski da interpretação procedimental das cláusulas de Horn.

O Prolog caracteriza-se por ser uma linguagem descritiva e prescritiva; ao mesmo tempo que descreve o que deve ser feito, descreve também como isso deve ser feito.

Baseada na lógica de predicados, a linguagem Prolog, concebida na década de 1970, é usada nos domínios de inteligência artificial, linguística computacional e sistemas de recomendação.

# 2. AMBIENTE DE PROGRAMAÇÃO

Para cada um dos três problemas deste trabalho - Quebra-cabeça de oito, Oito rainhas, O macaco e as bananas - foi elaborada uma solução em quatro diferentes paradigmas de programação: procedimental, orientado a objetos, lógico e funcional. Para cada um desses paradigmas, foi utilizada uma linguagem diferente, sendo elas, respectivamente, a Linguagem C, Linguagem Java, Linguagem LISP e PROLOG.

As execuções basearam-se em ferramentas online para escrita e compilação de código. Os códigos em Linguagem C e em Linguagem Java utilizaram da ferramenta **Repl.it**. Essa ferramenta permite aos usuários escrever códigos e construir aplicações utilizando um navegador, além de permitir que usuários compartilhem projetos e editem em conjunto. Há a possibilidade de criar espaços de trabalho nas mais populares linguagens de programação e os códigos são executados em uma máquina virtual.

As demais linguagens também foram utilizadas em ambiente virtual online. Para os códigos em LISP, utilizou-se o LISP Online (GNU CLISP v2.49+), disponível no site <https://www.tutorialspoint.com/execute\_lisp\_online.php>.

Por fim, os códigos em PROLOG utilizaram-se da estrutura online do SWI-Prolog, disponível no site <https://www.swi-prolog.org/>.

# 3. DESCRIÇÃO DA SOLUÇÃO PROPOSTA PARA CADA PROBLEMA

A seguir, descrevem-se as soluções propostas para cada um dos três problemas propostos para este trabalho.

## 3.1. QUEBRA-CABEÇA DE OITO

Nos próximos tópicos, seguem as descrições para o problema do quebra-cabeça de oito nas linguagens utilizadas.

### 3.1.1. Linguagens C e Java

O problema implementado nas linguagens JAVA e C foi implementado utilizando o algoritmo Branch and bound.

Ele consiste em fazer uma enumeração dos pontos que são candidatos a uma solução ótima do um problema. O termo branch refere-se ao fato de que o método efetua partições no espaço das soluções. O termo bound ressalta que a prova da otimização da solução utiliza-se de limites calculados ao longo da enumeração.

PSEUDOCÓDIGO Branch-and-Bound

Seja:

**M** = melhor caminho atualmente (menos custoso)

Faça:

1 Iniciar com algum problema P0

2 Faça S = P0, o conjunto de subproblemas ativos

3 **M** = oo (infinito ou valor muito grande)

4 i = 0

5 Repita enquanto S não está vazio:

6 Escolha um subproblema (solução parcial) P pertencente a S e o remova de S

7 Expandí-lo em subproblemas menores P1, P2,..., Pk

8 Para cada Pi:

9 Se Pi é uma solução completa

10 atualizar **M**

11 senão se Limite\_Inferior(Pi) < **M**

12 adicione Pi para S

13 Retorne **M**

Quando aplicado o paradigma descrito acima no problema do quebra cabeça de oito, o algoritmo de Branch and Bound vai verificando, ao decorrer da execução do problema, quais são os movimentos que, somados, resultam no menor custo, do estado inicial até o estado final estabelecido.

### 3.1.2. Linguagens LISP e PROLOG

O algoritmo utilizado para resolver o problema foi o A\* (A estrela). Ele consiste em encontrar caminhos em um determinado grafo através de um vértice/nó inicial e um final.

Pseudocódigo: A star

Sejam

**Q** = conjunto de nós a serem pesquisados;

**S** = o estado inicial da busca

**N** = estado atual

Faça:

1. Inicialize **Q** com o nó de busca **(S)** como única entrada;
2. Se **Q** está vazio, interrompa. Se não, escolha o melhor elemento de **Q**;
3. Se o estado **(n)** é um objetivo, retorne **n**;
4. (De outro modo) Remova **n** de **Q**;
5. Encontre os descendentes do estado **(n)** que não estão em visitados e crie todas as extensões de **n** para cada descendente;
6. Adicione os caminhos estendidos a **Q** e vá ao passo **2**;

No problema em questão, o estado inicial do quebra cabeça será o ponto de partida do grafo, que será construído através das possibilidades de jogadas a partir dos estados, até o estado final, que será o resultado final que queremos obter. Ambos os estados serão dados pelo usuário.

Além disso, o algoritmo também utilizará uma função heurística que será responsável por guiá-lo até a solução do problema. Ela irá calcular o quão distante o estado atual do quebra-cabeça de oito está do estado final. A função escolhida para resolver o problema em questão é o cálculo da quantidade de movimentos que serão necessários executar para que cada peça esteja no seu lugar de destino.

Por fim, assim que houver um estado que seja igual ao estado final, quer dizer que encontramos uma solução.

## 

## 3.2. OITO RAINHAS

O algoritmo utilizado nas quatro linguagens para resolver o problema foi o backtracking. Ele consiste em encontrar todas as soluções possíveis através da utilização da força bruta.

No problema em questão, o das oito rainhas, serão testadas todas as combinações possíveis das posições que elas podem estar. Quando o algoritmo se encontrada em um estado que não irá resultar em uma resposta válida, ele volta para o estado anterior. O mesmo acontece quando é alcançada uma extremidade, como por exemplo o nó folha duma árvore. Esse retrocesso entre os estados é feito por meio da recursão.

Por fim serão retornadas todas as possibilidades que irão satisfazer o problema descrito, onde uma rainha não pode estar na mesma coluna, linha e diagonal que outra.

## 3.3. O MACACO E AS BANANAS

A descrição deste problema para cada uma das implementações dá-se a seguir:

### 3.3.1. Linguagens C, Java e Lisp

Para resolver o problema em questão foi utilizado uma matriz, que representa a sala, que possui um tamanho definido e onde serão colocados o macaco, a banana e a caixa. O algoritmo considera que o macaco sempre começará na posição (0, 0), a banana sempre estará no centro da sala e a caixa poderá estar em qualquer lugar definido pelo usuário.

Para resolver o problema foi considerado que o macaco sabe onde a caixa está e andará em linha reta até chegar nela, depois empurra a caixa, também em linha reta, até onde a banana se encontrada para, enfim, pegá-la.

A solução irá mostrar cada ação feita pelo macaco e a coordenada, no plano cartesiano, onde ele se encontra depois de ter feito a ação.

### 3.3.2. Linguagem PROLOG

O algoritmo utilizado foi o backtracking, como foi já explicado e utilizado nas subseções anteriores.

No problema em questão, o algoritmo será utilizado para fazer com que o macaco encontre um caminho até a caixa, depois a carregue até debaixo do centro da sala, onde está localizada a banana, para que, enfim, o macaco consiga subir nela e alcance a banana que está pendurada no teto.

A solução irá mostrar as ações que o macaco precisará realizar para que consiga chegar ao seu objetivo final.

# 4. DESCRIÇÃO DA IMPLEMENTAÇÃO REALIZADA

A seguir, descrevem-se características sobre as implementações realizadas para cada um dos problemas em cada um dos quatro paradigmas de programação utilizados.

## 4.1. QUEBRA-CABEÇA DE OITO

A descrição deste problema para cada uma das implementações dá-se a seguir:

### 4.1.1. Quebra-cabeça de oito em C

A implementação em C deste problema inicializa com a criação de uma fila de prioridade, um quebra-cabeça 3x3 em uma estrutura matricial que demonstra o estado inicial e outro que demonstra o estado final. Esse tabuleiro é preenchido com os respectivos valores e um valor zero, que representa o espaço em branco. Depois será chamada a função de que irá resolver o problema.

|  |
| --- |
| **int** **main**(){   fila \*f = criaFila();   *//estado inicial*  *//o valor zero representa o espaço vazio*  **int** inicial[3][3] = {  {1, 2, 3},  {5, 6, 0},  {7, 8, 4}  };    *//estado final*  **int** final[3][3] = {  {1, 2, 3},  {5, 8, 6},  {0, 7, 4}  };    *//posição da matriz onde está o espaço em branco*  **int** x = 1, y = 2;    solucao(f, inicial, x, y, final); } |

A função solução irá receber a fila, que ainda está vazia, o estado inicial, final e as coordenadas do espaço vazio. Cada nó terá o estado onde o quebra-cabeça se encontra, as coordenadas do espaço vazio, o pai do estado, a profundidade (ou a quantidade de movimentos), o antecessor dele, o custo e as coordenadas do espaço vazio que serão utilizadas para movimentar o espaço. Primeiramente será criado o nó raíz utilizando o estado inicial e calculando seu custo. Depois disso ele será inserido na fila de prioridade. Enquanto a fila não estiver vazia, será retirado o nó com menor custo, que é o primeiro da fila. Se o custo é zero, quer dizer que uma solução foi encontrada e as movimentações serão mostradas na tela. Se não, ele criará quatro filhos para o nó: um que irá representar a movimentação do espaço vazio para cima, outro para esquerda, direita e para baixo. Após realizar uma verificação se as coordenadas são válidas, os filhos serão criados da mesma maneira que foi feito o nó raiz e com a movimentação do espaço em branco feita. Os filhos serão inseridos na lista de prioridades e isso será feito até achar um nó que possua um custo igual a zero.

|  |
| --- |
| *//ir para baixo, esquerda, cima e direita* *//será utilizada para movimentar o espaço branco de lugar* **int** linha[] = {1, 0, -1, 0}; **int** coluna[] = {0, -1, 0, 1};  *//função que irá utilizar o algoritmo branch and bound para* *//encontrar a solução* **void** **solucao**(fila \*f, **int** inicial[3][3], **int** x, **int** y, **int** final[3][3]){    *//cria o nó raíz*  no\* raiz = criaNo(inicial, x, y, 0, NULL, x, y);  raiz->custo = calcula\_custo(inicial, final);   *//insere o nó raíz na fila de prioridade;*  insereNo(f, raiz);    *//enquanto a fila não estiver vazia, vai continuar procurando pelo nó*  *//que possui o menor custo*  **while** (!filaVazia(f)){    *//retira o nó com menor custo da fila de prioridade*  no \*menor = removeNo(f);    *//se o custo for zero quer dizer que uma solução foi encontrada*  **if** (menor->custo == 0){   imprime\_caminho(menor);  **return**;  }    *//será criado quatro nós filhos para o nó de menor custo*  **for** (**int** i = 0; i < 4; i++){   **if** (verifica(menor->x + linha[i], menor->y + coluna[i])){   no\* filho = criaNo(menor->mat, menor->x, menor->y, menor->nivel + 1, menor, menor->x + linha[i], menor->y + coluna[i]);  filho->custo = calcula\_custo(filho->mat, final);   *//adiciona os nós filhos na lista de prioridade*  insereNo(f, filho);  }  }   } } |

### 4.1.2. Quebra-cabeça de oito em JAVA

A implementação desse problema em JAVA foi feita utilizando a estrutura de árvore de busca que é utilizada no algoritmo Branch and Bound utilizado para resolver o problema, explicado no tópico 3.1.1.

O programa contém duas classes: uma que contém o objeto (No\_arvore) que seriam os pontos da estrutura da árvore, e outra classe que é a classe principal do quebra cabeça, com a árvore em si e os métodos de execução dela.

Primeiramente, será apresentado a primeira classe que contém o objeto (No\_arvore). Ela contém os campos que receberão o nó pai do nó atual verificado, a matriz que contém as peças do tabuleiro, as coordenadas, o valor do custo que aquele nó tem e o nível em que se encontra o nó na estrutura da árvore. Depois da criação dessas variáveis citadas, existe um método chamado: “No\_arvore”. Ele atua como um construtor dos nós que são criados na execução do problema. Assim, quando um nó é criado este método é responsável por colocar as informaçẽos no lugar correto do objeto (No\_arvore).

|  |
| --- |
| **public** **class** **No\_arvore** {   **public** No\_arvore pai;  **public** **int**[][] matriz;   *// Coordenadas vazias*  **public** **int** x, y;   *// variavel que recebe o custo*  **public** **int** custo;   *// variavel que recebe a etapa*   **public** **int** nivel\_arvore;   **public** **No\_arvore**(**int**[][] matriz, **int** x, **int** y, **int** Xaux, **int** Yaux, **int** nivel\_arvore, No\_arvore pai) {  **this**.pai = pai;  **this**.matriz = **new** **int**[matriz.length][];  **for** (**int** i = 0; i < matriz.length; i++) {  **this**.matriz[i] = matriz[i].clone();  }   *// Faz a troca dos valores*  **this**.matriz[x][y] = **this**.matriz[x][y] + **this**.matriz[Xaux][Yaux];  **this**.matriz[Xaux][Yaux] = **this**.matriz[x][y] - **this**.matriz[Xaux][Yaux];  **this**.matriz[x][y] = **this**.matriz[x][y] - **this**.matriz[Xaux][Yaux];   *// Calcula os novos custos de acordo com o nivel da arvore*  **this**.custo = Integer.MAX\_VALUE;  **this**.nivel\_arvore = nivel\_arvore;  **this**.x = Xaux;  **this**.y = Yaux;  } } |

Agora será descrito para a segunda classe implementada no problema, classe principal do quebra cabeça.

Inicialmente, são definidas algumas variáveis globais. Uma para definir o tamanho da matriz alocada na execução do problema que contem as peças, ela é definido com o valor 3 porque a matriz será 3x3, e, também é definido 2 vetores. Esses vetores guardam os movimentos que serão possíveis de serem executados na linha da matriz e na coluna dela. A linha representa os movimentos para a esquerda ou direita e a coluna representa os movimentos para cima e para baixo.

|  |
| --- |
| **public** **class** **quebra\_cabeca** {   *// tamanho da matriz*   **public** **int** tamanho\_matriz = 3;   *// opções de movimentos na coluna e na linha*  **int**[] linha = {1, 0, -1, 0};  **int**[] coluna = {0, -1, 0, 1}; |

Logo em seguida, existem métodos que fazem a impressão de partes da árvore, sendo eles: “imprime” e “imprime\_caminho\_recursivo” .

O método “imprime” é responsável por imprimir os elementos contidos na matriz que contém as peças do quebra cabeça.

|  |
| --- |
| *// Metodo que imprime a matriz com os numeros*  **public** **void** **imprime**(**int**[][] matriz) {  **for** (**int**[] vet\_aux : matriz) {  **for** (**int** j = 0; j < matriz.length; j++) {  System.out.print(vet\_aux[j] + " ");  }  System.out.println();  }  } |

O método “imprime\_caminho\_recursivo” vai imprimir o caminho feito pelo programa até o nó raiz da árvore.

|  |
| --- |
| *// Imprimir o caminho do nó raiz até nó de destino de forma recursiva*  **public** **void** **imprime\_caminho\_recursivo**(No\_arvore raiz) {  *// se a raiz for igual a NULL, quer dizer que chegou no final ou não existe*  **if** (raiz == **null**) {  **return**;  }  *// Chama novamente a função passsando o nó raiz anterior*   imprime\_caminho\_recursivo(raiz.pai);  *// imprime o nó atual da matriz*  imprime(raiz.matriz);  *// Espaco entre os numeros ( estetica )*  System.out.println();  } |

O método “verifica\_posicao\_matriz” verifica se o programa não acessou uma posição que não existe dentro da matriz.

|  |
| --- |
| *// Verifica se a posição existe dentro da matriz*  **public** **boolean** **verifica\_posicao\_matriz**(**int** x, **int** y) {  *// Verifica se não estourou as dimensões da matriz*  **if** (x >= 0 && x < tamanho\_matriz && y >= 0 && y < tamanho\_matriz){  **return** TRUE;  }  *// Se estourou as dimensões da matriz, retorna false ( reprovado )*  **else**{  **return** FALSE;  }  } |

O método “ calculo\_dos\_custos “ efetua o calculo dos custos de cada nó de acordo com a posição do nó na árvore.

|  |
| --- |
| *// Função que efetua o cálculo dos custos*   **public** **int** **calculo\_dos\_custos**(**int**[][] inicio, **int**[][] fim) {  *// variável que que guarda os custos*  **int** aux = 0;  *// percorre a matriz até o final coluna e linha*  **for** (**int** i = 0; i < inicio.length; i++) {  **for** (**int** j = 0; j < inicio.length; j++) {  *// se início for diferente do bloco vazio e o elemento fina da matriz de estado incial*   *// for diferente da matriz do estado final acrescente 1 no custo*  **if** (inicio[i][j] != 0 && inicio[i][j] != fim[i][j]) {  aux++;  }  }  }  **return** aux;  } |

O método “sequencia” verifica se os nós estão na sequencia correta do estado final estabelecido do programa.

|  |
| --- |
| *// Método que verifica se os nós estão em sequência*  **public** **boolean** **sequencia**(**int**[][] matriz) {  *// Define as variaveis*  **int** aux = 0;  *// Cria uma lista de inteiro*  List<Integer> array = **new** ArrayList<>();    *// Cria uma lista2 de inteiro*  Integer[] array2 = **new** Integer[array.size()];      *// Passa os valores da matriz para o array*  **for** (**int**[] vet\_aux : matriz) {  **for** (**int** j = 0; j < matriz.length; j++) {  array.add(vet\_aux[j]);  }  } |

O método “Resolucao” contém a implementação do algoritmo Branch and Bound utilizado para resolver o problema, explicado no tópico 3.1.1.

|  |
| --- |
| *// Metodo que gerencia toda a resoluçaõ do problema*  *// Recebe as posições x e y estado inicial e o final*   **public** **void** **Resolucao**(**int**[][] inicio, **int**[][] fim, **int** x, **int** y) {  *// Os objetos devem ser processados​com base na prioridade os que tiverem menos custos*  PriorityQueue<No\_arvore> pq = **new** PriorityQueue<>(1000, (a, b) -> (a.custo + a.nivel\_arvore) - (b.custo + b.nivel\_arvore));  *// Cria um novo nó com as posições de menor custo*   No\_arvore raiz = **new** No\_arvore(inicio, x, y, x, y, 0, **null**);  raiz.custo = calculo\_dos\_custos(inicio, fim);  pq.add(raiz);    *// Enquanto a lista não chegou no final, faça:*  **while** (!pq.isEmpty()) {  *// Retira-se o elemento mais custoso*   No\_arvore min = pq.poll();  *// Se o custo for zero, o melhor caminho é ele, então basta imprimir*  **if** (min.custo == 0) {  imprime\_caminho\_recursivo(min);  **return**;  }    **for** (**int** i = 0; i < 4; i++) {  *// Verifica se a posição lida está dentro dos limites da matriz*   **if** (verifica\_posicao\_matriz(min.x + linha[i], min.y + coluna[i])) {  *// cria um novo nó na arvore*   No\_arvore no\_filho = **new** No\_arvore(min.matriz, min.x, min.y, min.x + linha[i], min.y + coluna[i], min.nivel\_arvore + 1, min);  *// calcula o custo*   no\_filho.custo = calculo\_dos\_custos(no\_filho.matriz, fim);  *// adiciona ele*   pq.add(no\_filho);  }  }  }  } |

E, por fim, a main do programa que contém os estados definidos iniciais e finais do programa.

|  |
| --- |
| *// MAIN DO PROGRAMA*  **public** **static** **void** **main**(String[] args) {  *// Define estado inicial e final*  **int**[][] inicio = {{1, 8, 2}, {0, 4, 3}, {7, 6, 5}};  **int**[][] fim = {{1, 2, 3}, {4, 5, 6}, {7, 8, 0}};   *// Inicia a coordenada*  **int** x = 1, y = 0;  *// Chama a classe*   quebra\_cabeca jogo = **new** quebra\_cabeca();  *// Passa os estados e a posição deles*   *// Caso os dados estejam incompletos, recusa*   **if** (jogo.sequencia(inicio)) {  jogo.Resolucao(inicio, fim, x, y);  } **else** {  System.out.println("O modelo inicial não é possível de resolver");  }  } |

### 4.1.3. Quebra-cabeça de oito em LISP

O programa LISP para o quebra-cabeça de oito possui uma estrutura que representa um estado para o problema.

|  |
| --- |
| (defstruct state  no  g\_val  h\_val  f\_val  no\_pai ) |

Além dessa estrutura, também consta no programa a declaração da variável global de estado inicial e de estado final do quebra-cabeça.

|  |
| --- |
| *;; estado final* (defvar \*estado\_final\* (list '1 '2 '3 '4 '5 '6 '7 '8 '0 '8))  *;; estado inicial* (defvar \*estado\_inicial\* (list '4 '2 '5 '0 '1 '3 '7 '8 '6 '3)) |

Há também uma função chamada move\_pecas. Essa função tem a responsabilidade de realizar as movimentações das peças nos sentidos superior, inferior, esquerdo e direito, bem como a de checar quais são as peças que se situam nessas direções. Essa informação é importante para determinar quais são as peças que têm capacidade de locomoção, bem como para qual direção essa locomoção pode ser feita.

Há também função para impressão de estado, ou seja, impressão do estado do quebra-cabeça.

|  |
| --- |
| (defun imprime\_estado(x) *;; função que imprime o estado em uma placa adequada como representação*  (format t "~a ~a ~a ~%" (first x) (second x) (third x))  (format t "~a ~a ~a ~%" (fourth x) (fifth x) (sixth x))  (format t "~a ~a ~a ~%" (seventh x) (eighth x) (ninth x))  (format t "~v@{~A~:\*~}~%~%" 30 "-")  ) |

O programa também conta com funções de verificação de situação, para conferir se o objetivo final já foi atingido.

### 4.1.4. Quebra-cabeça de oito em PROLOG

A implementação em PROLOG deste problema inicializa com a declaração dos fatos e das regras que serão utilizadas no algoritmo A\* (A estrela). Primeiro iremos instanciar a notação do node para facilitar na definição das lógicas e regras. A regra da função consiste na utilização da função custo total de um determinado, a sua busca entre os nós e salva o seu caminho na variável “Solucao”.

|  |
| --- |
| :- op(400,yfx,'#'). solucao(State, Solucao) :- funcao\_custo(State, 0, F),  busca([State#0#F#[]], S), reverse(S, Solucao). |

A função total custo é determinada pela soma do custo do estado e a profundidade que ele se encontra.

|  |
| --- |
| funcao\_custo(State, P, F) :- f\_custo(State, C),  F is P + C. |

A seguinte regra é utilizado para verificar se um estado, que está na cabeça da lista, é igual ao estado final.

|  |
| --- |
| busca([State#\_#\_#Solucao|\_], Solucao) :- final(State). |

A regra busca irá pegar o primeiro elemento da lista e realizar uma busca. Fazendo isso para todos os elementos dela. A regra expandir irá pegar todos os nós que apresenta uma determinada característica, nesse caso representada pelo segundo argumento da função bagof. A regra inserir irá inserir os nós na lista de acordo com o seu custo, colocando os menores primeiro de forma crescente e, se necessário, realizará o backtracking. A regra inserir tudo irá inserir todos os nós na lista.

|  |
| --- |
| *%pega o primeiro elemento da lista e realiza a busca até o último elemento* busca([B|R], S) :- expandir(B, Children), inserir\_tudo(Children, R, Open), busca(Open, S).  inserir\_tudo([F|R], Open1, Open3) :- inserir(F, Open1, Open2), inserir\_tudo(R, Open2, Open3).  inserir\_tudo([], Open, Open).  inserir(B, Open, Open) :- repetir\_no(B, Open), ! . inserir(B, [C|R], [B,C|R]) :- menor\_custo(B, C), ! . inserir(B, [B1|R], [B1|S]) :- inserir(B, R, S), !. inserir(B, [], [B]).  *%repete o nó* repetir\_no(P#\_#\_#\_, [P#\_#\_#\_|\_]).  *%utilizada para ordenar a ordem dos nós de forma crescente* menor\_custo( \_#\_#F1#\_ , \_#\_#F2#\_ ) :- F1 < F2.  expandir(State#D#\_#S,All\_My\_Children) :-  bagof(Child#D1#F#[Move|S],  (D1 is D+1,  movimenta\_peca(State,Child,Move),  funcao\_custo(Child,D1,F)),  All\_My\_Children). |

Após isso iremos declarar todos o estado final que queremos chegar e todas os fatos em relação a movimentação do espaço branco do quebra-cabeça. Devemos fazer isso para os quatro movimentos: cima, baixo, esquerda e direita. A regra para o cálculo do custo irá retornar a quantidade de movimentos necessários para colocar cada elemento em sua posição correta.

|  |
| --- |
| final(1/2/3/5/8/6/0/7/4).  direita(A/0/C/D/E/F/H/I/J, A/C/0/D/E/F/H/I/J). direita(A/B/C/D/0/F/H/I/J, A/B/C/D/F/0/H/I/J). direita(A/B/C/D/E/F/H/0/J, A/B/C/D/E/F/H/J/0). direita(0/B/C/D/E/F/H/I/J, B/0/C/D/E/F/H/I/J). direita(A/B/C/0/E/F/H/I/J, A/B/C/E/0/F/H/I/J). direita(A/B/C/D/E/F/0/I/J, A/B/C/D/E/F/I/0/J).  esquerda(A/0/C/D/E/F/H/I/J, 0/A/C/D/E/F/H/I/J). esquerda(A/B/C/D/0/F/H/I/J, A/B/C/0/D/F/H/I/J). esquerda(A/B/C/D/E/F/H/0/J, A/B/C/D/E/F/0/H/J). esquerda(A/B/0/D/E/F/H/I/J, A/0/B/D/E/F/H/I/J). esquerda(A/B/C/D/E/0/H/I/J, A/B/C/D/0/E/H/I/J). esquerda(A/B/C/D/E/F/H/I/0, A/B/C/D/E/F/H/0/I).  baixo(A/B/C/0/E/F/H/I/J, A/B/C/H/E/F/0/I/J). baixo(A/B/C/D/0/F/H/I/J, A/B/C/D/I/F/H/0/J). baixo(A/B/C/D/E/0/H/I/J, A/B/C/D/E/J/H/I/0). baixo(0/B/C/D/E/F/H/I/J, D/B/C/0/E/F/H/I/J). baixo(A/0/C/D/E/F/H/I/J, A/E/C/D/0/F/H/I/J). baixo(A/B/0/D/E/F/H/I/J, A/B/F/D/E/0/H/I/J).  cima(A/B/C/0/E/F/H/I/J, 0/B/C/A/E/F/H/I/J). cima(A/B/C/D/0/F/H/I/J, A/0/C/D/B/F/H/I/J). cima(A/B/C/D/E/0/H/I/J, A/B/0/D/E/C/H/I/J). cima(A/B/C/D/E/F/0/I/J, A/B/C/0/E/F/D/I/J). cima(A/B/C/D/E/F/H/0/J, A/B/C/D/0/F/H/E/J). cima(A/B/C/D/E/F/H/I/0, A/B/C/D/E/0/H/I/F).  f\_custo(Puzz, H) :- custo(Puzz, P),  H is P.  movimenta\_peca(P, C, esquerda) :- esquerda(P, C). movimenta\_peca(P, C, direita) :- direita(P, C). movimenta\_peca(P, C, baixo) :- baixo(P, C). movimenta\_peca(P, C, cima) :- cima(P, C).  custo(A/B/C/D/E/F/G/H/I, P) :-  a(A,Pa), b(B,Pb), c(C,Pc),  d(D,Pd), e(E,Pe), f(F,Pf),  g(G,Pg), h(H,Ph), i(I,Pi),  P is Pa+Pb+Pc+Pd+Pe+Pf+Pg+Ph+Pg+Pi.  a(0,0). a(1,0). a(2,1). a(3,2). a(4,4). a(5,1). a(6,3). a(7,3). a(8,2). b(0,0). b(1,1). b(2,0). b(3,1). b(4,3). b(5,2). b(6,2). b(7,2). b(8,1). c(0,0). c(1,2). c(2,1). c(3,0). c(4,2). c(5,3). c(6,1). c(7,3). c(8,2). d(0,0). d(1,1). d(2,2). d(3,3). d(4,3). d(5,0). d(6,2). d(7,2). d(8,1). e(0,0). e(1,2). e(2,1). e(3,2). e(4,2). e(5,1). e(6,1). e(7,1). e(8,0). f(0,0). f(1,3). f(2,2). f(3,1). f(4,1). f(5,2). f(6,0). f(7,2). f(8,2). g(0,0). g(1,2). g(2,3). g(3,4). g(4,2). g(5,1). g(6,3). g(7,1). g(8,2). h(0,0). h(1,3). h(2,2). h(3,3). h(4,1). h(5,2). h(6,2). h(7,0). h(8,1). i(0,0). i(1,4). i(2,3). i(3,2). i(4,0). i(5,3). i(6,1). i(7,1). i(8,2). |

## 4.2. OITO RAINHAS

A descrição deste problema para cada uma das implementações dá-se a seguir:

### 4.2.1. Oito Rainhas em C

A implementação em C deste problema inicializa criando um tabuleiro 8x8 em uma estrutura matricial. Esse tabuleiro é totalmente preenchido com zeros, que indicam a não existência de rainhas na posição em questão. Depois da chamada da função de resolução do problema, as oito rainhas serão colocadas em posições adequadas às especificações do problema e, nessas posições, constará o número 1 em vez do 0.

|  |
| --- |
| *// Main:* **int** **main**()  {   *// Cria o tabuleiro 8x8 e preenche-o com valores*  *// 0. Onde houver uma rainha, o valor é marcado como 1.*  **int** tabuleiro[8][8] = { { 0, 0, 0, 0, 0, 0, 0, 0 },   { 0, 0, 0, 0, 0, 0, 0, 0 },   { 0, 0, 0, 0, 0, 0, 0, 0 },   { 0, 0, 0, 0, 0, 0, 0, 0 },  { 0, 0, 0, 0, 0, 0, 0, 0 },  { 0, 0, 0, 0, 0, 0, 0, 0 },  { 0, 0, 0, 0, 0, 0, 0, 0 },  { 0, 0, 0, 0, 0, 0, 0, 0 } };   *//Chama função para resolver:*  **int** res = resolveProblema(tabuleiro, 0);   **if** (res == 1){  imprimir(tabuleiro);   }  **else**{  printf("\nErro.\n");  }  **return** 0;  } |

A função resolveProblema utiliza-se de técnica de recursão para tentar alocar as rainhas nas posições. Para isso, percorre o tabuleiro em uma coluna, varrendo todas suas linhas. Após alocar a rainha desta coluna em uma das linhas, segue à próxima coluna.

|  |
| --- |
| **int** **resolveProblema**(**int** tabuleiro[8][8], **int** coluna){    *//A função aloca as rainhas por coluna (segundo argumento passado)*  **if** (coluna >= 8) **return** 1; *//Isso porque já foram alocadas todas as rainhas*    *//Para cada coluna, o for testará colocar a rainha linha a linha:*  **for** (**int** i = 0; i < 8; i++) {   *//Na coluna passada como argumento, testa-se todas as linhas i*  **if** (verifica(tabuleiro, i, coluna)) {   *//Se verifica permite, coloca-se a rainha nesta posição:*  tabuleiro[i][coluna] = 1;    *//Recursão para a próxima coluna*  **if** (resolveProblema(tabuleiro, coluna + 1))   **return** 1;    *//Se o if acima não obteve sucesso, colocar a rainha nesta posição foi um erro.*  *//Portanto, retiramo-a da posição:*   tabuleiro[i][coluna] = 0;   }   }   **return** 0;  } |

A função verifica confere se a posição pode receber uma rainha, verificando à esquerda apenas, pois é a ordem de colocação (em resolveProblema, varremos o tabuleiro no sentido esquerda-direita). O cumprimento desses códigos garante que as rainhas serão colocadas em uma posição adequada. Este código retorna o primeiro tabuleiro encontrado cuja solução é válida.

### 4.2.2. Oito Rainhas em JAVA

A solução em JAVA é análoga à solução em C. Diferenciam-se no paradigma de programação, que, aqui, é orientado a objetos. Por conta disso, o código contém dois arquivos. Um deles é onde está a classe main. Nela, cria-se um objeto do tipo Rainha e inicializa-se a chamada para solução do problema.

|  |
| --- |
| **public** **class** **Main**{  *// Função Principal:*  **public** **static** **void** **main**(String args[])  {  Rainha R1 = **new** Rainha();  R1.montaTabuleiro();  } } |

No outro arquivo, consta a classe rainha. A classe rainha possui três métodos. O método montaTabuleiro(), invocado pela main, cria o tabuleiro de dimensão 8x8, como criado na Linguagem C, preenchendo-o em todas as posições com o número 0. Em seguida, chama o método resolveProblema. Esse método funciona da mesma forma que o método homônimo empregado na implementação em C, incluindo na chamada por verifica, que atestará se uma rainha pode ser colocada na posição analisada.

Por fim, como na implementação em C, há uma impressão do resultado. Essa impressão, realizada pelo método imprimir da classe rainha, apresenta a primeira solução válida encontrada pelo problema. Essa solução, recapitulando, tem a posição das rainhas marcada pelo número 1 no tabuleiro.

|  |
| --- |
| **void** **imprimir**(**int** tabuleiro[][])  {  **for** (**int** i = 0; i < 8; i++) {  **for** (**int** j = 0; j < 8; j++){  System.out.print(" " + tabuleiro[i][j] + " ");  }  System.out.println();  }  } |

### 4.2.3. Oito Rainhas em LISP

A seguir, descreve-se elementos da implementação do oito rainhas em LISP. As atividades de cada trecho de código estão destacadas e explicadas nos comentários que se seguem.

|  |
| --- |
| *; O codigo tenta colocar uma rainha em casa posicao x* *; Se conseguir, segue em frente com as demais utilizando recursão* *; Se não conseguir, retrocede na recursão e vai para o proximo caso* *; O tabuleiro tem dimensão 8x8.* (defun solve-loop (x y rainhas max) *; Condicionais mais complexos podem ser construídos através do form cond, que é equivalente a if ... else if ... fi.* *; Um cond consiste de símbolo con seguido por um número de cláusulas-cond, cada qual é uma lista. O primeiro elemento de uma cláusula-cond é a condição, os lementos restantes são a ação.* *; O cond encontra a primeira cláusula que avalia para true, executando a ação respectiva e retornando o valor resultante. Nenhuma das restantes é avaliada.*  (cond  ((= max (length rainhas)) (print (list 'solucao rainhas)) (cdr rainhas))  *; imprimir resposta e retroceder (tentar resolver mais de 1 resposta)*  *; ((= max (comprimento das rainhas)) rainhas)*  ((or (> x max) (> y max)) (cdr rainhas)) *; backtrack*  ((can-place x y rainhas) *; testa se pode colocar rainha na posicao (recursao)*  *; append concatena listas.*  (setq rec (solve-loop (+ 1 x) 1 (append (list (list x y)) rainhas) max))  *; para cada valor de y, ele testa x = 1 até x = max (sendo que max corresponde a 8)*  (cond  ((equal rainhas rec) (solve-loop x (+ 1 y) rainhas max)) *; backtracked*  (t rec)))  (t (solve-loop x (+ 1 y) rainhas max))))  *; para cada valor de x, ele testa y = 1 até y = max (sendo que max corresponde a 8)*  *; Utiliza-se do tamanho máximo do tabuleiro, que é 8, e chama a função solve-loop iniciando pela primeira posição do tabuleiro (1,1), que serão passados para (x,y) na função.* (defun solve (max rainhas)  (solve-loop 1 1 rainhas max)) |

### 4.2.4. Oito Rainhas em PROLOG

A solução para o problema das oito rainhas utilizando o PROLOG, baseia-se em uma consulta ?- tabuleiro(S), solucao(S), solucionar(S). . Essa consulta direciona para o código nessas três estruturas: tabuleiro, solucao e solucionar.

A estrutura dessa primeira é apenas a montagem do que seria o tabuleiro. Nela, estão marcadas as linhas e, com um caracter \_, o que será a coluna ocupada pela rainha na referida linha.

|  |
| --- |
| tabuleiro([1,\_,2,\_,3,\_,4,\_,5,\_,6,\_,7,\_,8,\_]). |

Já solucao verifica se os valores estão entre 1 e 8, para que não haja ultrapassagem da dimensão do tabuleiro. Essa checagem é feita por pertence.

|  |
| --- |
| solucao([]).  solucao([X,Y|Aux2]):-  solucao(Aux2),  pertence(Y,[1,2,3,4,5,6,7,8]),  block([X,Y],Aux2).  pertence(X,[X|\_]).  pertence(X,[\_|Aux]):- pertence(X,Aux). |

A chamada de block garante que a colocação da rainha em uma posição será válida.

|  |
| --- |
| block(\_,[]).  block([X,Y],[X1,Y1|Aux2]):-  Y =\= Y1,  Y1 - Y =\= X1 - X,  Y1 - Y =\= X - X1,  block([X,Y],Aux2). |

Por fim, solucionar apresenta um print com as soluções encontradas. São as soluções porque não há nenhuma limitação para backtracking, ou seja, o código é capaz de listar todas as soluções possíveis se isso for um desejo.

solucionar([X, Y | Aux]) :- write('|'), write(X), write(', '), write(Y), solucionar(Aux).

## 4.3. O MACACO E AS BANANAS

A descrição deste problema para cada uma das implementações dá-se a seguir:

### 4.3.1. O macaco e as bananas em C

Primeiramente, o programa irá utilizar uma matriz para representar a sala, no nosso caso ela terá tamanho 10x10, e inserir os agentes dentro dela. O macaco começará no ponto (0, 0), a banana no centro da sala e a caixa em um lugar pré-definido pelo usuário. Depois disso será chamada a função que irá resolver o problema.

|  |
| --- |
| **int** **main**(){   **int** sala[N][N];  **int** x\_caixa = 3, y\_caixa = 8;   *//1 = macaco*  *//2 = banana*  *//3 = caixa*  **for**(**int** i=0; i<10; i++){  **for**(**int** j=0; j<10; j++){  *//aloca o macaco na posição 0,0*  **if**(i==0 && j==0){  sala[i][j] = 1;  }**else**{  *//aloca a banana no centro da sala*  **if**(i==(N/2)-1 && j==(N/2)-1){  sala[i][j] = 2;  }**else**{  *//aloca a caixa na sala*  **if**(i==x\_caixa && j==y\_caixa){  sala[i][j] = 3;  }**else**{  sala[i][j] = 0;  }  }  }  }  }   solucao(sala, x\_caixa, y\_caixa);   **return** 1; } |

A solução é feita da seguinte maneira: sabendo onde a caixa está, o macaco vai até ela andando em linhas retas pelo eixo X e Y. Será marcado como 1 na matriz da sala para representar onde o macaco já passou. Depois de chegar na caixa, ele irá arrastá-la até estar localizada embaixo da banana, que o macaco também sabe onde está, já que sempre estará no centro da sala. Depois ele irá andar com a caixa, também simulando pela matriz da sala, até a banana. Então, por fim, ele irá subir na caixa e pegá-la.

|  |
| --- |
| **int** **solucao**(**int** sala[N][N], **int** x\_caixa, **int** y\_caixa){   **int** distancia\_caixa\_x = x\_caixa;  **int** distancia\_caixa\_y = y\_caixa;   **for**(**int** i=1; i<=distancia\_caixa\_x; i++){  *//anda para frente no eixo x*  sala[i][0] = 1;  printf("O macaco deu um passo para frente. Posição atual: [%d][0]\n", i);  }   **for**(**int** i=1; i<=distancia\_caixa\_y; i++){  *//anda para frente no eixo x*  sala[distancia\_caixa\_x][i] = 1;  printf("O macaco deu um passo para a esquerda. Posição atual: [%d][%d]\n", distancia\_caixa\_x, i);  }   printf("O macaco chegou na caixa!\n");   **int** distancia\_banana\_x = x\_caixa - ((N/2)-1);  **int** distancia\_banana\_y = y\_caixa - ((N/2)-1);    *//o macaco precisa voltar*  **if**(distancia\_banana\_x > 0){  **for**(**int** i=abs(distancia\_banana\_x); i>0; i--){  sala[distancia\_caixa\_x--][distancia\_caixa\_y] = 1;  printf("O macaco empurrou a caixa para trás. Posição atual: [%d][%d]\n", distancia\_caixa\_x, distancia\_caixa\_y);  }  }**else**{  **if**(distancia\_banana\_x < 0){  *//o macaco precisa andar para frente*  **for**(**int** i=1; i<=abs(distancia\_banana\_x); i++){  sala[distancia\_caixa\_x++][distancia\_caixa\_y] = 1;  printf("O macaco empurrou a caixa para frente. Posição atual: [%d][%d]\n", distancia\_caixa\_x, distancia\_caixa\_y);  }  }  }   *//o macaco precisa voltar*  **if**(distancia\_banana\_y > 0){  **for**(inti=abs(distancia\_banana\_y);i>0;i--){   sala[distancia\_caixa\_x][distancia\_caixa\_y--]=1;   printf("O macaco empurrou a caixa para direita. Posição atual: [%d][%d]\n", distancia\_caixa\_x, distancia\_caixa\_y);  }  }**else**{   **if**(distancia\_banana\_y < 0){  *//o macaco precisa andar para frente*  **for**(**int** i=1; i<=abs(distancia\_banana\_y); i++){  sala[distancia\_caixa\_x][distancia\_caixa\_y++] = 1;  printf("O macaco empurrou a caixa para esquerda. Posição atual: [%d][%d]\n", distancia\_caixa\_x, distancia\_caixa\_y);  }  }  }  printf("O macaco subiu na caixa!\n");  printf("O macaco pegou a banana!\n"); } |

### 4.3.2. O macaco e as bananas em JAVA

A lógica de implementação é a mesma apresentada no tópico anterior. Difere-se aqui, logicamente, pela questão das classes.

O código implementa uma classe Main. Nessa classe, cria-se o objeto M1 do tipo macaco. Utiliza-se o método montaSala() da classe macaco para montar a sala 10 x 10. Em seguida, convoca-se o método solucao, passando a configuração da sala e as coordenadas da caixa. Veja a seguir.

|  |
| --- |
| **public** **class** **Main** {  **public** **static** **void** **main**(String[] args) {  **int** sala[][] = **new** **int**[10][10];  System.out.println("Problema do macaco e da banana!");  Macaco M1 = **new** Macaco();    sala = M1.montaSala();   M1.solucao(sala, 3, 8);  } } |

A classe macaco possui dos métodos. O método montaSala() tem código idêntico ao código em Linguagem C, apresentado anteriormente. Já o método solucao é quase idêntico, apenas implementa algumas características típicas da linguagem, como o System.out.println para exibir os textos. Além disso, a classe importa a função Math do Java para realizar o cálculo de valor absoluto.

### 4.3.3. O macaco e as bananas em LISP

A solução não se difere muito em relação à solução procedural. A quantidade de golpes necessária para derrubar as bananas do cacho é determinada por um número randomizado de 1 à 5, que determina quantas pancadas o cacho deverá levar até cair. A solução é a seguinte:

|  |
| --- |
| (defvar n 10) *;Tamanho da sala* (defvar pos\_cacho (random n)) *;Posição do cacho* (defvar pos\_macaco 0) *;Posição do macaco* (print "Tamanho da sala") (print n) (print "Posição do cacho:") (print pos\_cacho) (loop for i from 1 to pos\_cacho  do (setf pos\_macaco (+ pos\_macaco 1)) (print "O macaco empurrou a caixa. Posição do macaco:")  (print pos\_macaco) ) (print "O macaco está embaixo do cacho") (defvar qnt\_golpes (random 5)) (loop for i from 1 to qnt\_golpes  do (print "O macaco golpeou o cacho") ) (print "As bananas caíram") (print "O macaco comeu a banana") |

### 4.3.4. O macaco e as bananas em PROLOG

A resolução do problema dos macacos e das bananas em PROLOG baseia-se na seguinte consulta: ?- consegue(estado(na\_porta,no\_chao,na\_janela,nao\_tem), X). . Basicamente, essa consulta diz onde está o macaco (na porta), em que posição vertical (no caso, no chão), onde está a caixa (na janela) e se ele tem a banana ou não (nesse caso, não tem).

O programa em Prolog contém uma sequência de movimentos. Esses movimentos baseiam-se em um estado inicial, uma ação, e o estado resultado dessa ação. Essas regras demarcam apenas situações possíveis. Por exemplo: o macaco estando no centro, acima da caixa, que também está no centro, e não tendo a banana consigo, pode utilizar seu bastão para pegar a banana. O mesmo não seria possível, por exemplo, se ele estivesse no chão em vez de acima da caixa, ou se ele estivesse no centro e a caixa na porta ou na janela. Essas regras são cruciais porque denotam o que é e o que não é possível realizar nesse ambiente.

|  |
| --- |
| move(estado(no\_centro,acima\_caixa,no\_centro,nao\_tem), pegar\_banana, estado(no\_centro,acima\_caixa,no\_centro,tem) ).  move(estado(P,no\_chao,P,Banana), subir, *% subir na caixa* estado(P,acima\_caixa,P,Banana) ).  move(estado(P1,no\_chao,P1,Banana), empurrar(P1,P2), *% empurrar caixa de P1 para P2* estado(P2,no\_chao,P2,Banana) ).  move(estado(P1,no\_chao,Caixa,Banana), caminhar(P1,P2), *% caminhar de P1 para P2* estado(P2,no\_chao,Caixa,Banana) ). |

Além dessas regras, há também uma que indica que o macaco já tem a banana em suas mãos. Nesse caso, obviamente, tanto faz o lugar em que ele ou a caixa estejam.

|  |
| --- |
| consegue(estado(\_,\_,\_,tem),[]). |

Tem-se, ainda, a regra que vai indicar como o macaco conseguirá a banana.

|  |
| --- |
| consegue(Estado1,[Movimento|Resto]) :- move(Estado1,Movimento,Estado2), consegue(Estado2,Resto). |

# 

Por fim, apresenta-se o caminho X, que apareceu na consulta apresentada no início desta solução. X é o caminho que faz o macaco conseguir pegar a banana a partir do estado inicial deste problema. A execução do código dirá se é possível o macaco chegar à banana.

|  |
| --- |
| X = (caminhar(na\_porta,na\_janela), empurrar(na\_janela, no\_centro), subir, pegar\_banana). |

# REFERÊNCIAS BIBLIOGRÁFICAS

Ana Cristina Vieira de Melo e Flávio Soares Corrêa da Silva. Princípios de Linguagens de Programação. 3ª edição. Blucher. 2003.

AVELINO, Thiago. História do Lisp: abra os olhos para programação funcional. **iMasters**. Disponível em: <https://imasters.com.br/desenvolvimento/historia-lisp-abra-os-olhos-para-programacao-funcional>. Acesso em: 15 nov. 2020.

GOMES, R. A. M; SANTOS, H. G. Branch-and-Bound para problemas de Otimização Combinatória. **PPGCC - Programa de Pós-Graduação em Ciência da Computação UFOP.** Disponível em: <http://www.decom.ufop.br/menotti/paa111/files/PCC104-111-pts-11.1-RafaelAntonioMarquesGomes.pdf>. Acesso em: 17 nov. 2020.

INDRUSIAK, Leandro Soares. Linguagem Java. **JavaRS**. Disponível em: <https://www.cin.ufpe.br/~arfs/introjava.pdf>. Acesso em: 15 nov. 2020.

MEIDANIS, João. Paradigmas de programação - Prolog. **Universidade Estadual de Campinas**. Disponível em: <https://www.ic.unicamp.br/~meidanis/courses/mc346/2017s2/prolog/apostila-prolog.pdf>. Acesso em: 15 nov. 2020.

NOLETO, Cairo. Linguagem C: o que é e quais os principais fundamentos? **Trybe**. Disponível em: <https://blog.betrybe.com/linguagem-de-programacao/linguagem-c/>. Acesso em: 15 nov. 2020.

SONG, Siang Wun. Uma Aula Prática sobre LISP (Notas de Aula). **Universidade de São Paulo**. Disponível em: <https://www.ime.usp.br/~song/mac5710/slides/10lisp.pdf>. Acesso em: 15 nov. 2020.