Universidade Federal de Juiz de Fora
Departamento de Ciência da Computação
DCC014 - Inteligência Artificial

Fórmula para 24

Trabalho I e II

Profa. Dra. Luciana Conceição Dias Campos

Grupo: 2
Deoclécio Porfírio Ferreira Filho - MAT 201876043
Ivanilson Honório Gonçalves - MAT 201776002
Igor Marchito Zamboni - MAT 201976020
Marcos Aquino Almeida - MAT 201276024

Fevereiro - 2022

PROBLEMA 2: FÓRMULA PARA 24

Usar as 4 cartas do baralho disponíveis e os sinais da matemática para criar uma fórmula que apresente o número 24 como resposta.

Tem que usar as 4 cartas, e os sinais podem repetir na fórmula e não precisa usar todos.

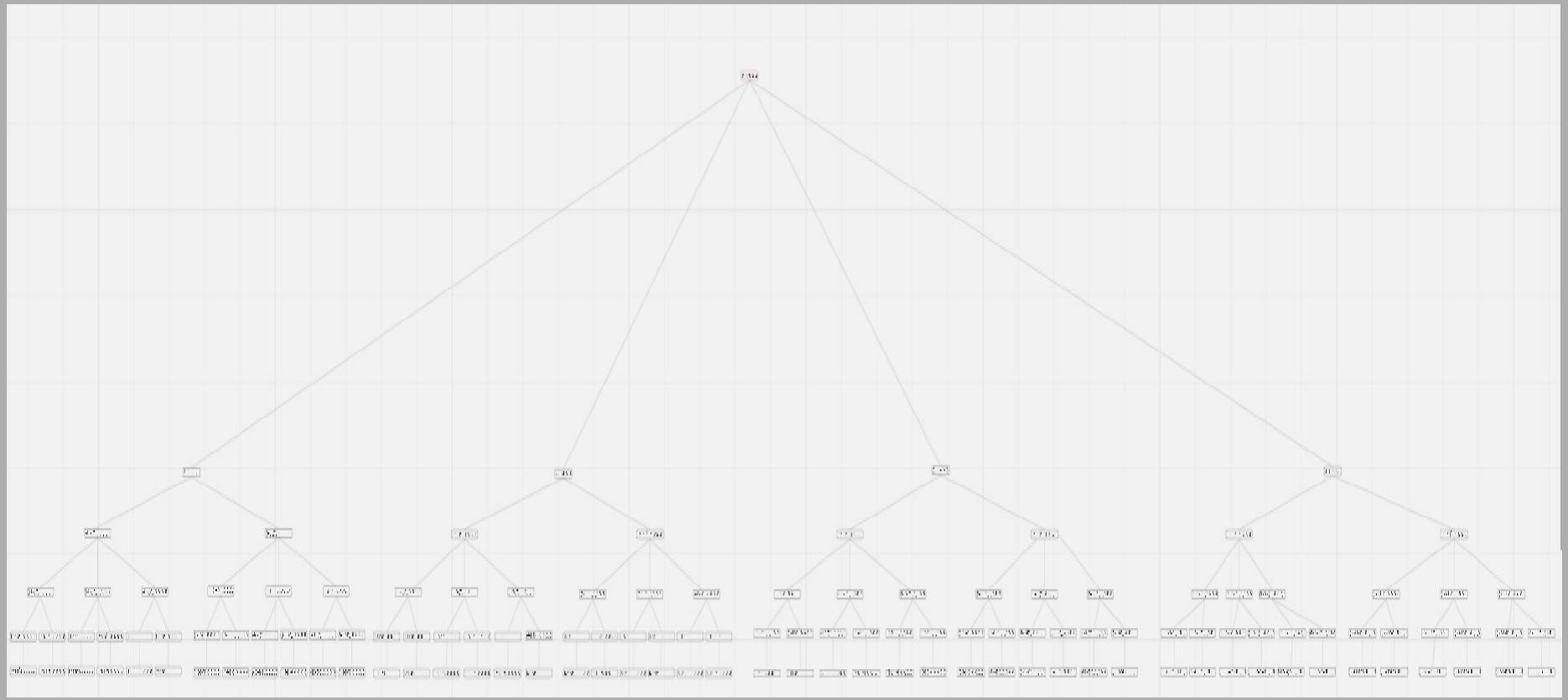
Temos as seguintes considerações:

- A carta Às representa o valor 1.
- A carta Valete (J) representa o valor 11.
- A carta Dama (Q) representa o valor 12.
- A carta Rei (K) representa o valor 13.
- Os sinais que devem ser usados para a fórmula são:
  - Soma: +
  - Subtração: -
  - Multiplicação: x
  - Divisão: /
  - Parenteses: ()

Fórmula para 24 - Adaptado

- Usar as 4 cartas do baralho disponíveis e os sinais da matemática para criar uma fórmula que apresente o número 24 como resposta.
- É obrigatório o uso de todas as 4 cartas.
- Sabe-se que haverá duas operações básica (uma soma e uma subtração) entre 3 cartas dentro de parênteses e esse resultado será multiplicado por uma outra carta.
- Desta forma o problema passa a ter uma aparência do tipo x*(y+z-w) onde, x, y, w e z correspondem às cartas.

Árvore Completa



Busca em Largura

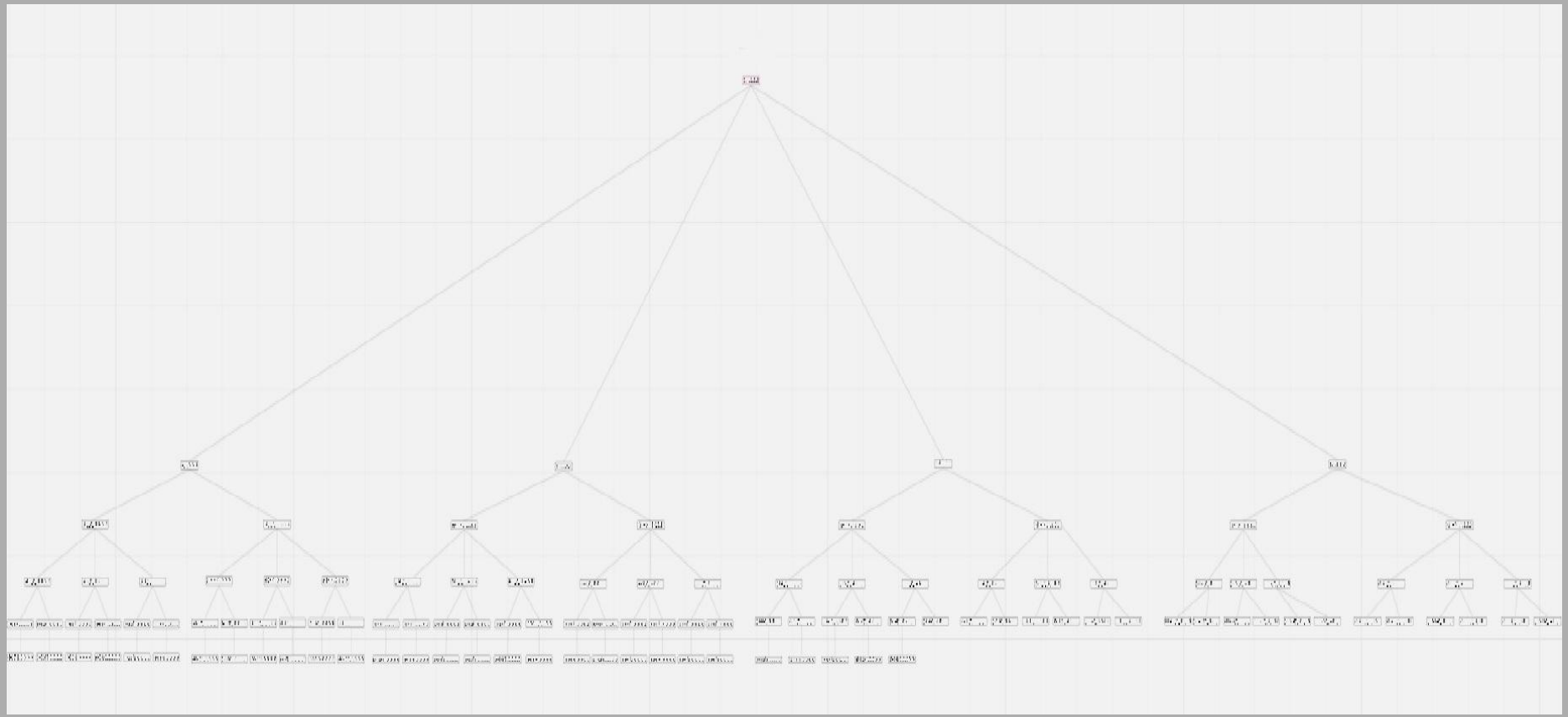
Na Busca em largura (Breadth-First Search), o nó raiz é expandido primeiro, depois todos os seus sucessores são expandidos, depois seus sucessores e assim por diante.

Todos os nós são expandidos em uma determinada profundidade na árvore de busca antes que qualquer nó no próximo nível seja expandido. O que faz com que todos os filhos imediatos de nós são explorados antes que qualquer filho dos filhos seja considerado.

Busca em Largura

- Etapa 1: Coloque o nó inicial (inicial) em uma lista chamada ABERTOS de nós inexplorados. Se o nó inicial for um nó objetivo, uma solução foi encontrada.
- Etapa 2: Se (ABERTOS está vazio) ou (ABERTOS = OBJETIVO) então não existem soluções, então, encerre a pesquisa.
- Passo 3: Remova o primeiro nó **A** de ABERTOS e o coloca em uma lista chamada FECHADOS de nós expandidos.
- Passo 4: Expanda o nó a, Se não tiver sucessor Então vá para o Passo 2.
- Passo 5: Coloque todos os sucessores do nó a, no final de ABERTOS.
- Passo 6: Se algum dos sucessores do nó a for um estado objetivo, então uma solução será encontrada.

Árvore Completa da Busca em Largura



```
void buscaLarg() {
  int custo = 0, filho = RAIZ, contNO = 0, t = 0;
  int jogo[NUM] = {0, 4, 6, 5, 3},
      final[NUM] = {24, 0, 0, 0, 0},
      carta[NUM - 1] = {4, 6, 5, 3},
      gJogo[][NUM] = {{0, 0, 0, 0, 0},
                      {0, 0, 0, 0, 0},
                      {0, 0, 0, 0, 0},
                      {0, 0, 0, 0, 0}};
  string cartaXL[2][NO];
  for (int i = 0; i < 2; i++)
    for (int j = 0; j < NO; j++)
      cartaXL[i][j] = " ";

  vector<char> noFilho;
  vector<int> altValor;
  vector<string> lstAbertos, lstFechados, fila, vetTemp;
  string k = "(0)", r;
  char f = char(RAIZ);
  r = convChar(f);

  cout << "\n\n"
       << "                    *** BUSCA EM LARGURA ***  \n\n"
       << " - Critério de desempate - \n"
       << "Carta de menor valor ou ordem de entrada na lista de abertos.  \n\n";

  cout << "RAIZ DA ÁRVORE → Estado inicial: ";
  impEstado(jogo);
  cout << "\nEstado final: ";
  impEstado(final);
  cout << "\n";
  cout << "Raiz → " << f << k;
  ImpFilhoRaizI(jogo, filho);
  cout << "\n\n";
  //!============================================================

  cout << "Inicio\n";
  impLstAbertos(lstAbertos);
  impLstFechados(lstFechados);
  cout << "\n\n";

  //!=====================================

  cout << "Filhos de  " << f << k;
  vetTemp = gravaEstado(jogo, filho, gJogo, cartaXL, contNO, t);
  for (int i = 0; i < vetTemp.size(); i++)
    fila.push_back(vetTemp[i]);
  cout << "\n";
  lstAbertos.push_back(r);
  impLstAbertos(lstAbertos);
  impLstFechados(lstFechados);
  impLstFila(fila);

  cout << "\n\n";
  lstFechados.push_back(lstAbertos[0]);
  lstAbertos.erase(lstAbertos.begin());
  for (int i = 0; i < vetTemp.size(); i++)
    lstAbertos.push_back(vetTemp[i]);
  limpaFilaL(vetTemp);
  impLstAbertos(lstAbertos);
  impLstFechados(lstFechados);
```

```
  // inicia os filhos de A

  for (size_t j = 1; j < NUM - 1; j++) {
    cout << "\nFilhos de  " << lstAbertos[0];
    if (lstAbertos[0] == cartaXL[0][j]) { //! j = 1: A → 4

      jogo[0] = stoi(cartaXL[1][j]);
      for (size_t i = 0; i < NUM; i++) {
        jogo[i + 1] = carta[i];
        if (jogo[i + 1] == jogo[0])
          jogo[i + 1] = 0;
      }

      vetTemp = gravaEstado(jogo, filho, gJogo, cartaXL, contNO, j);
    }
    limpaFilaL(fila);
    for (int i = 0; i < vetTemp.size(); i++)
      fila.push_back(vetTemp[i]);

    cout << "\n";

    impLstAbertos(lstAbertos);
    impLstFechados(lstFechados);
    impLstFila(fila);

    lstFechados.push_back(lstAbertos[0]);
    lstAbertos.erase(lstAbertos.begin());
    for (int i = 0; i < vetTemp.size(); i++)
      lstAbertos.push_back(vetTemp[i]);
  }

  for (size_t j = 1 + 3; j < NUM + 5; j++) {
    cout << "\nFilhos de  " << lstAbertos[0];
    if (lstAbertos[0] == cartaXL[0][j]) { //!

      jogo[0] = stoi(cartaXL[1][j]);
      for (size_t i = 0; i < NUM; i++) {
        jogo[i + 1] = carta[i];
        if (jogo[i + 1] == jogo[0])
          jogo[i + 1] = 0;
      }

      vetTemp = gravaEstado(jogo, filho, gJogo, cartaXL, contNO, j);
    }
    limpaFilaL(fila);
    for (int i = 0; i < vetTemp.size(); i++)
      fila.push_back(vetTemp[i]);

    cout << "\n";

    impLstAbertos(lstAbertos);
    impLstFechados(lstFechados);
    impLstFila(fila);

    lstFechados.push_back(lstAbertos[0]);
    lstAbertos.erase(lstAbertos.begin());
    for (int i = 0; i < vetTemp.size(); i++)
      lstAbertos.push_back(vetTemp[i]);
  }

  cout << "\n\n";
  for (size_t i = 0; i < 2; i++) {
    for (size_t j = 0; j < NO; j++) {
      if (cartaXL[i][j] != " ")
        cout << " " << cartaXL[i][j] << " ";
    }
    cout << "\n";
  }
  cout << "\n\n";
}
```

Busca em Profundidade

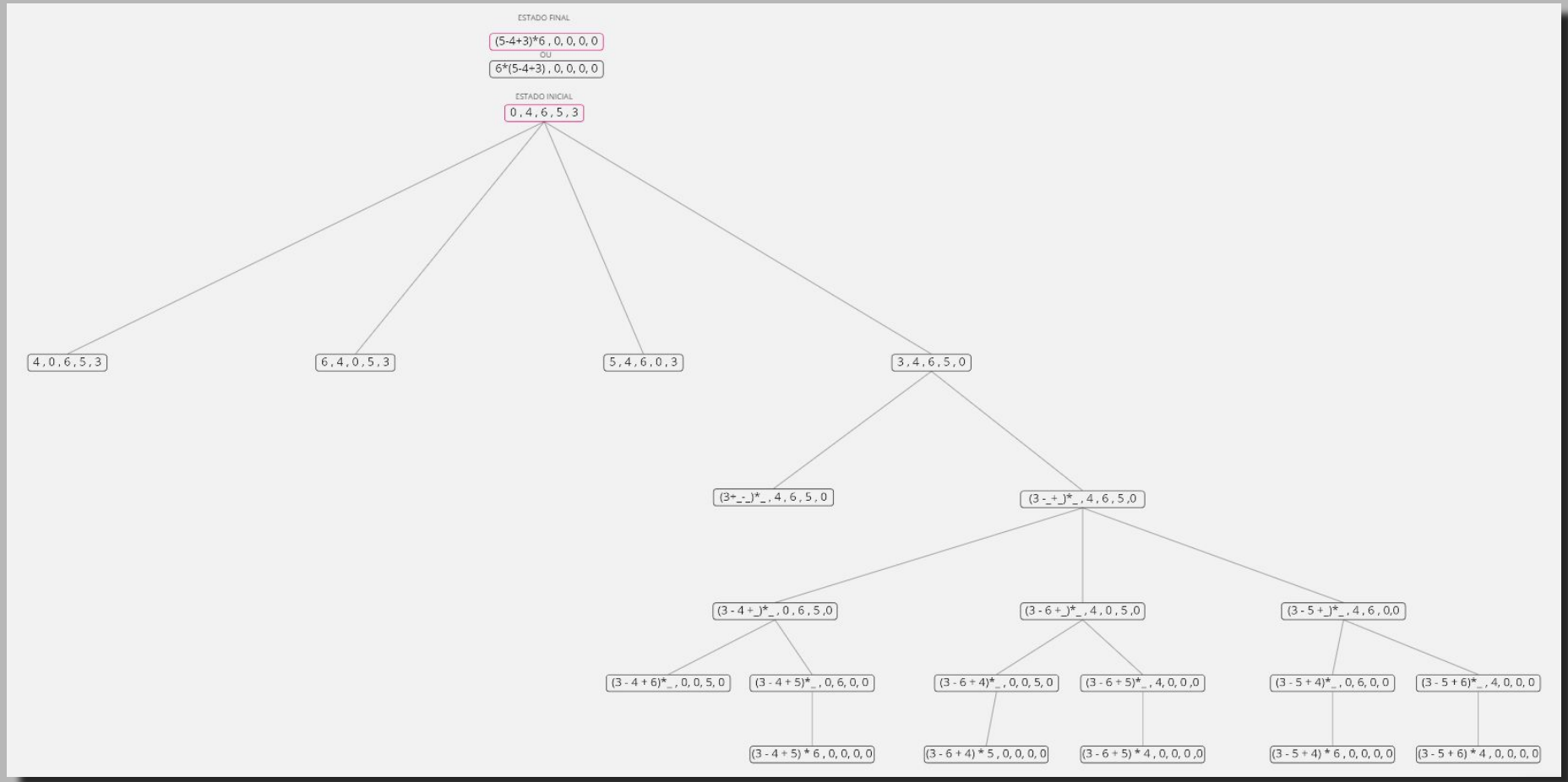
A busca em profundidade começa expandindo o nó inicial usando um operador. Ele gera todos os sucessores do nó inicial e os testa.

O DFS é caracterizado pela expansão do nó mais recentemente gerado ou mais profundo primeiro. Ele precisa armazenar o caminho da raiz até o nó folha, bem como os nós não expandidos. Não é completo nem ideal. Se o DFS cair, uma ramificação infinita não terminará até que um estado de meta seja encontrado. Mesmo que seja encontrado, pode haver uma solução melhor em um nível superior.

Busca em Profundidade

- Como existem apenas quatro números e três operadores que são adição, subtração e multiplicação, o espaço total da solução é constante.
- O algoritmo Busca em profundidade operará em um vetor e forçará todos os pares de números no vetor e aplicará quatro operadores nesses dois números, empurrará o resultado para o vetor e se chamará recursivamente até que haja apenas 1 número no vetor - o que podemos verificar facilmente se é 24. Os tipos no vetor devem ser duplos, convertendo os inteiros de entrada para os duplos primeiro.
- Com a multiplicação e as adições, podemos acelerar um pouco o processo ignorando os pares comparando os índices. Lembrando que estamos seguindo as regras colocadas no estudo de caso fórmula 24.

Árvore Completa da Busca em Profundidade



```
void buscaProf() {

  int custo = 0, filho = RAIZ, contNO = 0, t = 0;
  int jogo[NUM] = {0, 4, 6, 5, 3},
      final[NUM] = {24, 0, 0, 0, 0},
      carta[NUM - 1] = {4, 6, 5, 3},
      gJogo[][NUM] = {{0, 0, 0, 0, 0},
                      {0, 0, 0, 0, 0},
                      {0, 0, 0, 0, 0},
                      {0, 0, 0, 0, 0}};
  string cartaXL[2][NO];
  for (int i = 0; i < 2; i++)
    for (int j = 0; j < NO; j++)
      cartaXL[i][j] = " ";

  vector<char> noFilho;
  vector<int> altValor;
  vector<string> lstAbertos, lstFechados, vetTemp;
  string k = "(0)", r;
  char f = char(RAIZ);
  r = convChar(f);
  stack<string> pilha, pilhaTemp, pt, pFechados, pAbertos;
  //!-------------------------------------------------------------

  cout << "\n\n"
       << "                    *** BUSCA EM PROFUNDIDADE ***  \n\n"
       << " Direção: Direita \n"
       << "PILHA - Primeiro a entrar é ultimo a sair.  \n\n";

  cout << "RAIZ DA ÁRVORE → Estado inicial: ";
  impEstado(jogo);
  cout << "\nEstado final: ";
  impEstado(final);
  cout << "\n";
  cout << "Raiz → " << f << k;
  ImpFilhoRaizI(jogo, filho);
  cout << "\n\n";
  //!=============================================================

  cout << "Inicio\n";
  impPilhaAbertos(pAbertos);
  impPilhaFechados(pFechados);
  cout << "\n\n";

  //!======================================

  cout << "Filhos de  " << f << k;

  vetTemp = gravaEstado(jogo, filho, gJogo, cartaXL, contNO, t);
  for (int i = 0; i < vetTemp.size(); i++)
    pilha.push(vetTemp[i]);
  cout << "\n";
  pAbertos.push(r);
  impPilhaAbertos(pAbertos);
  impPilhaFechados(pFechados);
  impPilha(pilha);

  //!======================================
  cout << "\n\n";
  pFechados.push(pAbertos.top());
  pAbertos.pop();
  for (int i = 0; i < vetTemp.size(); i++)
    pAbertos.push(vetTemp[i]);

  limpaPilha(pilhaTemp);
  impPilhaAbertos(pAbertos);
  impPilhaFechados(pFechados);

  // inicia os filhos de A
  //!======================================

  for (size_t j = 1; j < NUM - 1; j++) {
    cout << "\nFilhos de  " << pAbertos.top();

    if (pAbertos.top() == cartaXL[0][j - 1]) { //! j = 1: A → 4

      cout << "\nAbertos: " << pAbertos.top() << " == "
           << "CartaX: " << cartaXL[0][j] << " ";
      //    cout<< "\nAbertos: "<< pAbertos.top()<< " == "<<"CartaX: "<< cartaXL[0][j]<<" ";

      jogo[0] = stoi(cartaXL[1][j]);
      for (size_t i = 0; i < NUM; i++) {
        jogo[i + 1] = carta[i];
        if (jogo[i + 1] == jogo[0])
          jogo[i + 1] = 0;
      }
      //  jogo[i] = gJogo[0][i]; // grava o valor do jogo atua
      gEstadoP(jogo, filho, gJogo, cartaXL, contNO, j, pt);

      while (!pt.empty()) {
        pilhaTemp.push(pt.top());
        pt.pop();
      }
    }
    //   limpaPilha;
    while (!pilhaTemp.empty()) {
      pilha.push(pilhaTemp.top());
      pilhaTemp.pop();
    }
    cout << "\n";

    impPilhaAbertos(pAbertos);
    impPilhaFechados(pFechados);
    impPilha(pilha);

    cout << "\n\n";
    pFechados.push(pAbertos.top());
    pAbertos.pop();
    while (!pilhaTemp.empty()) {
      pAbertos.push(pilhaTemp.top());
      pilhaTemp.pop();
    }
  }
}
```

Busca Gulosa (Heurística)

Heurística: Valor mais aproximado do objetivo

- Definimos a multiplicação como última operação a ser realizada mantendo as demais operações dentro dos parênteses;
- Um número que seja divisível por 24 dentre as cartas é escolhido, pois como as outras operações são básicas, se o número não for divisível por 24 ele precisaria ser multiplicado por um número fracionário, o que no caso é impossível com duas operações básicas (+ e -) sobre 3 números inteiros não fracionários.
- Assim, essa divisão de 24 por uma das cartas geraria um valor objetivo a ser atingido através de uma soma e uma subtração com as cartas restantes.

Busca Gulosa (Heurística)

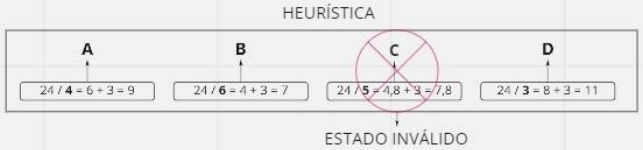
Heurística: Valor mais aproximado do objetivo

- Em caso de mais de uma carta ser divisível por 24, têm-se como critério de desempate usar a que gera o menor valor objetivo.
- Após isso, soma-se o número de cartas que ainda não foram utilizadas.
- A partir do segundo nível, a carta a ser utilizada será subtraída do valor gerado pela divisão do primeiro nível e ao módulo dessa subtração será adicionado o número de cartas restantes.
- O caminho a ser tomado será, novamente, o de menor valor.

Busca Gulosa (Heurística)

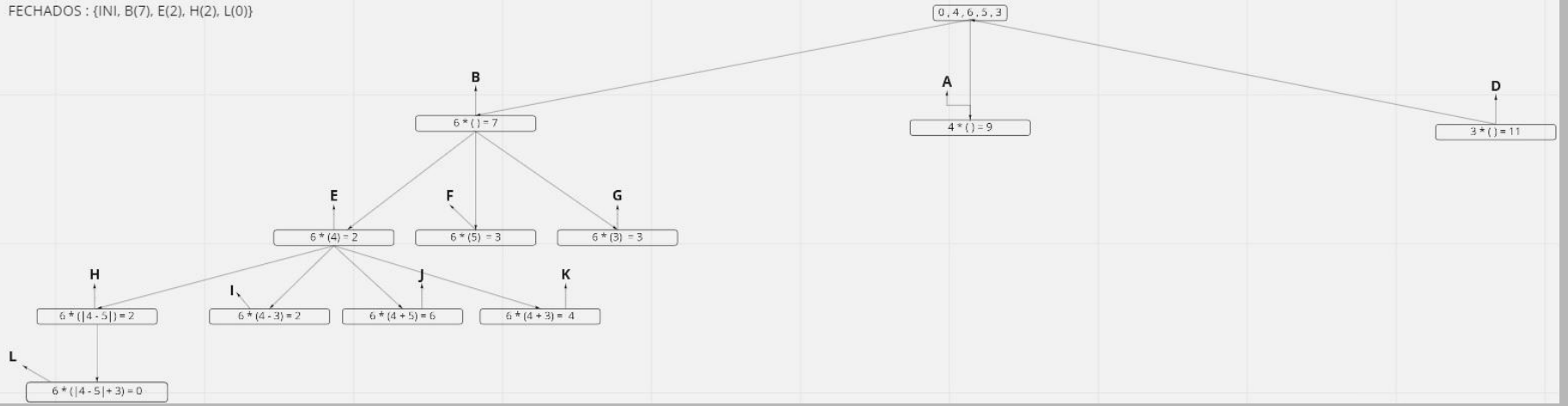
ESTADO INICIAL [0, 4, 6, 5, 3]

ESTADO FINAL [24]



ABERTOS : ~~{I (2), F(3), G(3), K(4), J(6), A(9), D(11)}~~

FECHADOS : {INI, B(7), E(2), H(2), L(0)}



```c
//! Regra1: Inicial  - Raiz
int valorH4(int *jogo, int i) {
  int valor = 0;
  int temp = 0;

  for (int j = 1; j < NUM; j++) { //! soma do numeo de cartas diferente de zero
    if (jogo[j] != 0)
      temp++;
  }
  if (jogo[i] != 0 && RESULT % jogo[i] == 0) {
    valor = (RESULT / jogo[i]) + (temp - 1);
  }
  return valor;
}
```

```
//! Regra2: segunda carta
int valorH3(int *jogo, int i) {
  int valor = 0;
  int temp = 0;

  for (int j = 1; j < NUM; j++) { //! soma do numeo de cartas diferente de zero
    if (jogo[j] != 0)
      temp++;
  }
  if (jogo[i] != 0) {
    // cout<< "\nJogo"<<i<<": " << jogo[i]<< endl;
    valor = (4 - jogo[i]);
    if (valor < 0)
      valor = valor * (-1);
    valor += (temp - 1);
  }

  return valor;
}
```

```
void buscaGulosa() {
  int jogo[NUM] = {0, 4, 6, 5, 3},
      final[NUM] = {24, 0, 0, 0, 0},
      carta[NUM - 1] = {4, 6, 5, 3},
      cartaX[2] = {100, 0};
  int custo = 0, filho = RAIZ;

  vector<char> noFilho;
  vector<int> fila, altValor;
  vector<string> lstAbertos, lstFechados;
  string k = "(0)";
  char f = char(RAIZ);

  cout << "\n\n"
       << "                        *** BUSCA GULOSA ***  \n\n"
       << "                             HEURÍSTICA\n\n"
       << " - Critério de desempate - \n"
       << "Ordem de entrada na lista de abertos.  \n\n";
  cout << "Estado inicial: " << f << " = ";
  impEstado(jogo);
  cout << "\nEstado final: ";
  impEstado(final);
  cout << "\n\n";
  cout << "Inicio\n";
  cout << "Filhos da Raiz - " << f << k;
  ImpFilhoRaizH(jogo, filho);
  fila = gravaFilaFilhoH(jogo, noFilho, filho, carta, cartaX);
  cout << "Filhos de " << f << k << ": { ";
  impFilhos(fila, lstAbertos, noFilho);
  concStrCharF(k, f, lstFechados);
  for (size_t i = 0; i < 4; i++) {
    impLstFechados(lstFechados);
    ordenaFila(fila, lstAbertos);
    impLstAbertos(lstAbertos);
    cout << "Estado atual: ";
    cartaX[0] = 100;
    alteraValor(jogo, cartaX[1]);
    impEstado(jogo);

    switch (cartaX[1]) {···
    cout << "Filhos de  " << lstAbertos[0] << ": { ";
    lstFechados.push_back(lstAbertos[0]);
    lstAbertos.erase(lstAbertos.begin());
    limpaNo(noFilho, fila);
    fila = gravaFilaFilhoH(jogo, noFilho, filho, carta, cartaX);
    impFilhos(fila, lstAbertos, noFilho);
    impLstFechados(lstFechados);
    ordenaFila(fila, lstAbertos);
    impLstAbertos(lstAbertos);
    verificaFinalH(jogo, final, custo, lstFechados);

    cout << "\n";
  }
}
```

Busca Ordenada

Regra 1:

- Deve ser utilizadas todas as cartas, a multiplicação, soma e subtração;
- A multiplicação deve ser utilizada primeiro e as demais não tem ordem de utilização;

Critério de desempate: Carta de menor valor ou a que vier primeiro.

Verificação de estados válidos

Estado Inicial: {0, 4, 6, 5, 3}

Estado Final: {24, 0, 0, 0, 0}

Busca Ordenada

Regra 2:

- 24 / i - sendo i a primeira carta do estado inicial;
- Resultado da divisão deve ser inteiro;
- Soma de todas as cartas menos a carta i;
- Soma de todas as cartas = 18;

{A = (24/4=6 + (18 - 4)), B = (24/6 = 4 + (18 - 6)), C= (24/5 = Inválido), D = (24/3 = 8 + (18 - 3))}

£ = {A(20), B(16), D(23)};

```c
//! Regra: Inicial
int valorI(int *jogo, int i) {
  int valor = 0;
  int temp = 0;

  for (int j = 1; j < NUM; j++) {
    temp += jogo[j];
  }
  if (jogo[i] != 0 && RESULT % jogo[i] == 0) {
    valor = (RESULT / jogo[i]);
    valor += (temp - jogo[i]);
  }
  return valor;
}
```

Busca Ordenada

Filhos da Raiz £(0) = {B(16), A(20), D(23)};

Deve ser escolhido a primeira carta da lista de abertos e atualizar a lista de fechados e estado atual:

Lista de Fechados: { £(0)}

Ordenando a lista de abertos:

Lista de Abertos: { B(16) A(20) D(23) }

Carta escolhida: 6

Verifica-se o valor atual é igual ao estado final:

Estado atual: {6, 4, 0, 5, 3} -> {6 * ()} <> Estado Final: {24, 0, 0, 0, 0}

Busca Ordenada

Regra 3:

- Deve ser verificado se a próxima carta multiplicada pelo primeira escolhida dá o valor 24 ou quanto que falta ou está passando para dar os 24.
- Deve ser somado a esse resultado a soma das cartas ainda não utilizadas, se houver;

Filhos de B(16) : {(6 * (4 - 4) → 0 + (5 + 3) = 8), (6 * (5 - 4) → 1 + (4+3) = 8),

(6 * (3 - 4) → 1 + (5 + 4) = 10)}

Filhos de B(16) : { E(8) F(8) G(12) }

Carta escolhida: **4**

Verifica-se o valor atual é igual ao estado final:

Estado atual: {24, 0, 0, 5, 3} -> {6 * (4)} <> Estado Final: {24, 0, 0, 0, 0}

```
//! Regra2: segunda carta escolhida
int valorDiv(int *jogo, int i) {
  int valor = 0;
  int temp = 0;
  for (int j = 1; j < NUM; j++) {
    temp += jogo[j];
  }
  if (jogo[i] != 0) {
    valor = (4 - jogo[i]);
    if (valor < 0)
      valor = valor * (-1);
    valor += (temp - jogo[i]);
  }
  return valor;
}
```

Busca Ordenada

Filhos de B(16) : { E(8) F(8) G(12) }

Lista de Fechados: { £(0) B(16) }

Ordenando a lista de abertos:

Lista de Abertos: { E(8) F(8) G(12) A(20) D(23) }

Estado atual: {24, 0, 0, 5, 3} -> {6 * (4)}

Escolhido: E(8) → Utilizado o critério desempate.

Busca Ordenada

Regra 4:

- Deve ser verificado se a próxima carta, somado ou subtraído, da carta anterior e multiplicada pela primeira carta escolhido é igual 24 ou quanto que falta ou está passando para os 24.
- Deve ser somado a esse resultado a soma das cartas ainda não utilizadas, se houver;

Filhos de E(8) : {(6 * (|(4 - 5)| - 4) → 3 + 3 = 6), (6 * ((4 - 3) - 4) → 3 + 5=8),

(6 * (|(4+5)| - 4) → 5 + 3 = 6), (6 * ((4 + 3) - 4) → 3 + 5=8)}

Filhos de E(8) : { H(6) I(8) J(8) K(8) }

Carta escolhida: **5**

Verifica-se o valor atual é igual ao estado final:

Estado atual: {6, 0, 0, 0, 3} -> {6 * |(4 - 5)|} <> Estado Final: {24, 0, 0, 0, 0}

```c
//! Regra3: terceira carta escolhida
int valorSoma(int *jogo, int i) {
  int valor = 0;
  int temp = 0;
  for (int j = 1; j < NUM; j++) { //! soma das cartas re
    temp += jogo[j];
  }
  //! soma das cartas restantes
  if (jogo[i] != 0) {
    valor = (4 + jogo[i]);
    valor -= 4;
    valor += (temp - jogo[i]);
  }
  return valor;
}
```

```c
//! Regra3: terceira carta escolhida
int valorSub(int *jogo, int i) {
  int valor = 0;
  int temp = 0;
  for (int j = 1; j < NUM; j++) { //! soma das cartas rest
    temp += jogo[j];
  }
  //! subtração das cartas restantes
  if (jogo[i] != 0) {
    valor = (4 - jogo[i]);
    if (valor < 0)
      valor = valor * (-1);
    valor -= 4;
    valor = valor * (-1);
    valor += (temp - jogo[i]);
  }
  return valor;
}
```

You, seconds ago • U

Busca Ordenada

Filhos de E(8) : { H(6) I(8) J(8) K(8) }

Lista de Fechados: { £(0) B(16) E(8) }

Ordenando a lista de abertos:

Lista de Abertos: { H(6) F(8) I(8) J(8) K(8) G(12) A(20) D(23) }

Estado atual: {6, 0, 0, 0, 3} -> {6 * (|4 - 5|)}

```c
//! Regra4: quarta carta escolhida      You, seconds ago • Uncommitted changes
int valorSomaQ(int *jogo, int i) {
  int valor = 0;
  int temp = 0;
  for (int j = 1; j < NUM; j++) { //! soma das cartas restantes
    temp += jogo[j];
  }

  if (jogo[i] != 0) {
    valor = (1 + jogo[i]);
    if (valor < 0)
      valor = valor * (-1);
    valor -= 4;
    valor = valor * (-1);
    valor += (temp - jogo[i]);
  }
  return valor;
}
```

Busca Ordenada

Regra 5:

- Como foi utilizado a subtração, a próxima carta deve ser somada com o valor anterior e, este valor, multiplicado pela primeira carta escolhido dá 24 ou quanto que falta ou está passando para dar os 24.
- Deve ser somado a esse resultado a soma das cartas ainda não utilizadas, se houver;

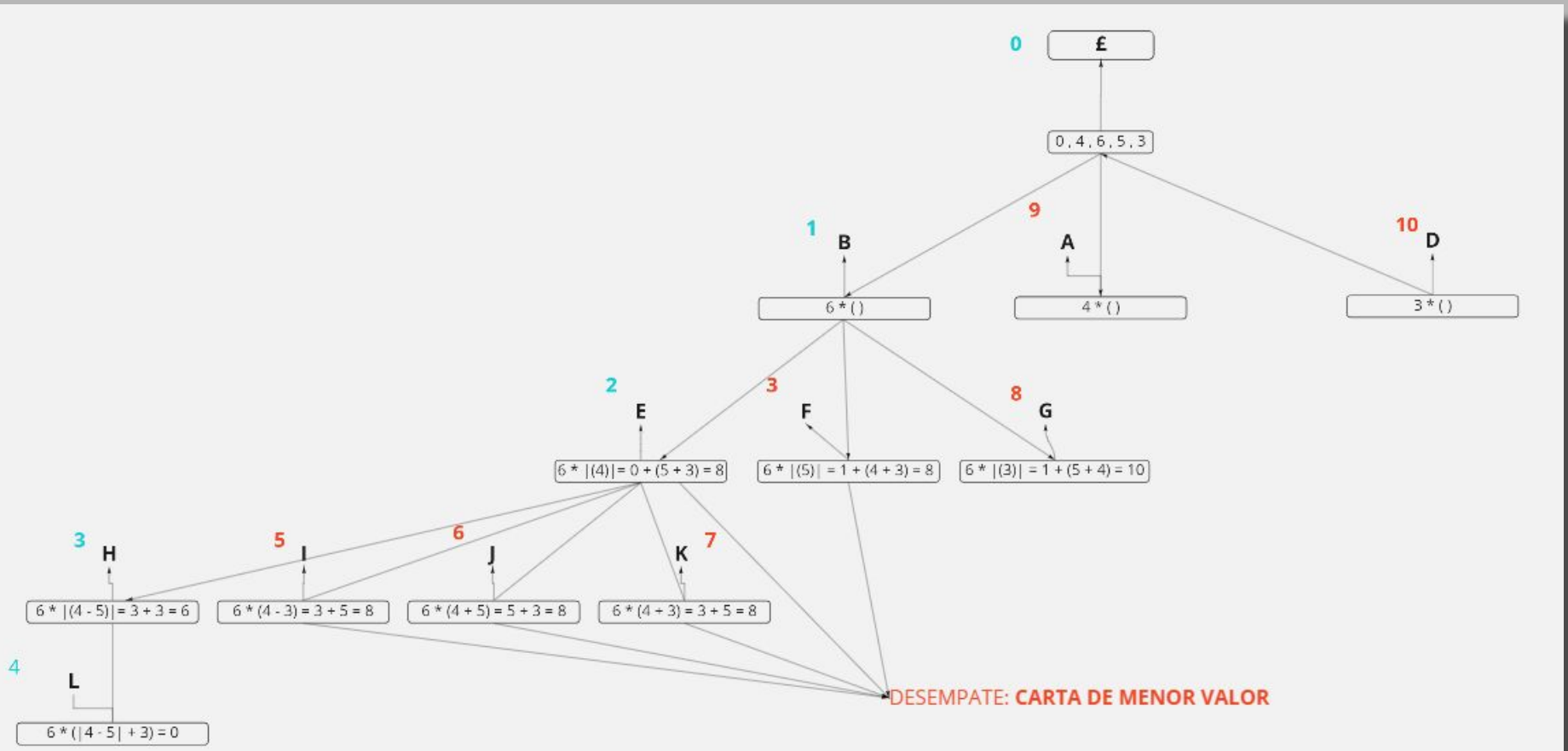
Filhos de H(6) : {(6 * (|(4 - 5)| + 3) → 0 + 0 = 0),

Filhos de H(6) : { L(0) }

Carta escolhida: **3**

Estado atual: {24, 0, 0, 0, 0} -> {6 * (|(4 - 5)| + 3)} <> Estado Final: {24, 0, 0, 0, 0}

Fim de jogo!

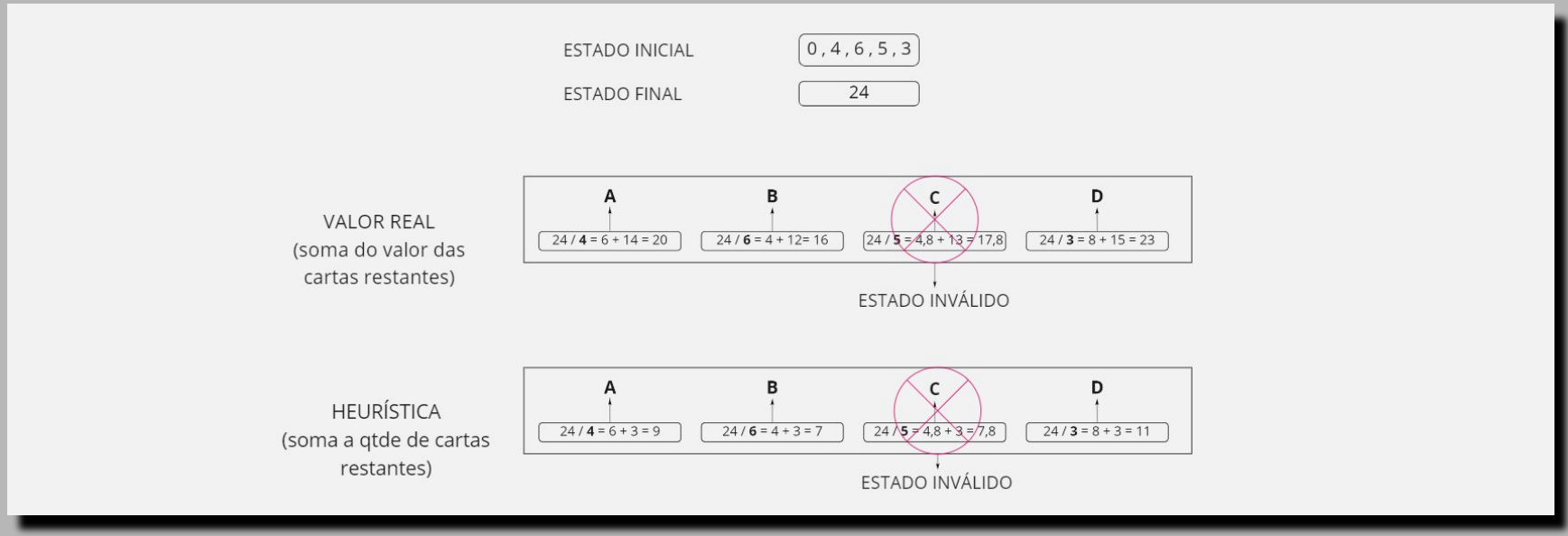

0
£
0 , 4 , 6 , 5 , 3
9
1
B
A
10
D
6 * ( )
4 * ( )
3 * ( )
2
3
8
E
F
G
6 * |(4)| = 0 + (5 + 3) = 8
6 * |(5)| = 1 + (4 + 3) = 8
6 * |(3)| = 1 + (5 + 4) = 10
3
5
6
7
H
I
J
K
6 * |(4 - 5)| = 3 + 3 = 6
6 * (4 - 3) = 3 + 5 = 8
6 * (4 + 5) = 5 + 3 = 8
6 * (4 + 3) = 3 + 5 = 8
4
L
6 * (|4 - 5| + 3) = 0
DESEMPATE: CARTA DE MENOR VALOR

```
void buscaOrdenada() {

  int jogo[NUM] = {0, 4, 6, 5, 3},
      final[NUM] = {24, 0, 0, 0, 0},
      carta[NUM - 1] = {4, 6, 5, 3},
      cartaX[2] = {100, 0};
  int custo = 0, filho = RAIZ;

  vector<char> noFilho;
  vector<int> fila, altValor;
  vector<string> lstAbertos, lstFechados;
  string k = "(0)";
  char f = char(RAIZ);

  cout << "\n\n"
       << "                    *** BUSCA ORDENADA ***  \n\n"
       << " - Critério de desempate - \n"
       << "Carta de menor valor ou ordem de entrada na lista de abertos.  \n\n";

  cout << "Estado inicial: " << f << " = ";
  impEstado(jogo);
  cout << "\nEstado final: ";
  impEstado(final);
  cout << "\n\n";

  //!============================================================
```

```
  cout << "Inicio\n";
  cout << "Filhos da Raiz - " << f << k;
  ImpFilhoRaiz(jogo, filho);
  fila = gravaFilaFilho(jogo, noFilho, filho, carta, cartaX);
  cout << "Filhos de " << f << k << ": { ";
  impFilhos(fila, lstAbertos, noFilho);
  // ordenaFila(fila, lstAbertos);
  cout << "}\n";
  concStrCharF(k, f, lstFechados);
  //!================================================

  for (size_t i = 0; i < 4; i++) {
    impLstFechados(lstFechados);
    ordenaFila(fila, lstAbertos);
    impLstAbertos(lstAbertos);
    cout << "Estado atual: ";
    cartaX[0] = 100;
    alteraValor(jogo, cartaX[1]);
    impEstado(jogo);
    switch (cartaX[1]) {···
    cout << "Filhos de  " << lstAbertos[0] << ": { ";
    lstFechados.push_back(lstAbertos[0]);
    lstAbertos.erase(lstAbertos.begin());
    limpaNo(noFilho, fila);
    fila = gravaFilaFilho(jogo, noFilho, filho, carta, cartaX);
    // custo = custo + fila[0]; // custo
    impFilhos(fila, lstAbertos, noFilho);
    verificaFinal(jogo, final);
    cout << "\n";
  }
}
```

Busca A*

- Função de avaliação = valor real + heurística



Busca A*

Avalia os nós combinando o custo para alcançar cada nó n e o custo para ir do nó n até o nó objetivo: f(n) = g(n) + h(n).

- g(n) custo do caminho do nó inicial até o nó n.
- h(n) custo estimado do caminho de custo mais baixo do nó n até o nó objetivo.
- f(n) custo estimado da solução mais econômica passando por n.

No problema que estamos trabalhando temos g(Valor Real - Busca Ordenada) e h(Resultado da Heurística - Busca Gulosa).

Busca A*

ESTADO ATUAL: **£**

f(A) = g(A) + h(A) = 20 + 9 = 29

f(B) = g(B) + h(B) = 16 + 7 = 23

f(D) = g(D) + h(D) = 23 + 11 = 34

	E	**F**	**G**
VALOR REAL	6 * |(4)| = 0 + (5 + 3) = 8	6* |(5)| = 1 + (4 + 3) = 8	6 * |(3)| = 1 + (5 + 4) = 10
HEURÍSTICA	6 * (4) = 0 + 2 = 2	4 * (5) = 1 + 2 = 3	4 * (3) = 1 + 2 = 3

ESTADO ATUAL: **B**

f(E) = g(E) + h(E) = 8 + 2 = 10

f(F) = g(F) + h(F) = 8+ 3 = 11

f(G) = g(G) + h(G) = 10 + 3 = 13

Busca A*

	H	I	J	K
VALOR REAL	6 * |(4 - 5)| = 3 +	6 * (4 - 3) = 3 + 5	6 * (4 + 5) = 5 + 3	6 * (4 + 3) = 3 + 5
HEURÍSTICA	6 * |(4 - 5)| = 3 +	6 * (4 - 3) = 3 + 1	6 * (4 + 5) = 5 + 1	6 * (4 + 3) = 3 +

ESTADO ATUAL: **E**

f(H) = g(H) + h(H) = 6 + 4 = 10

f(I) = g(I) + h(I) = 8 + 4 = 12

f(J) = g(J) + h(J) = 8 + 6 = 14

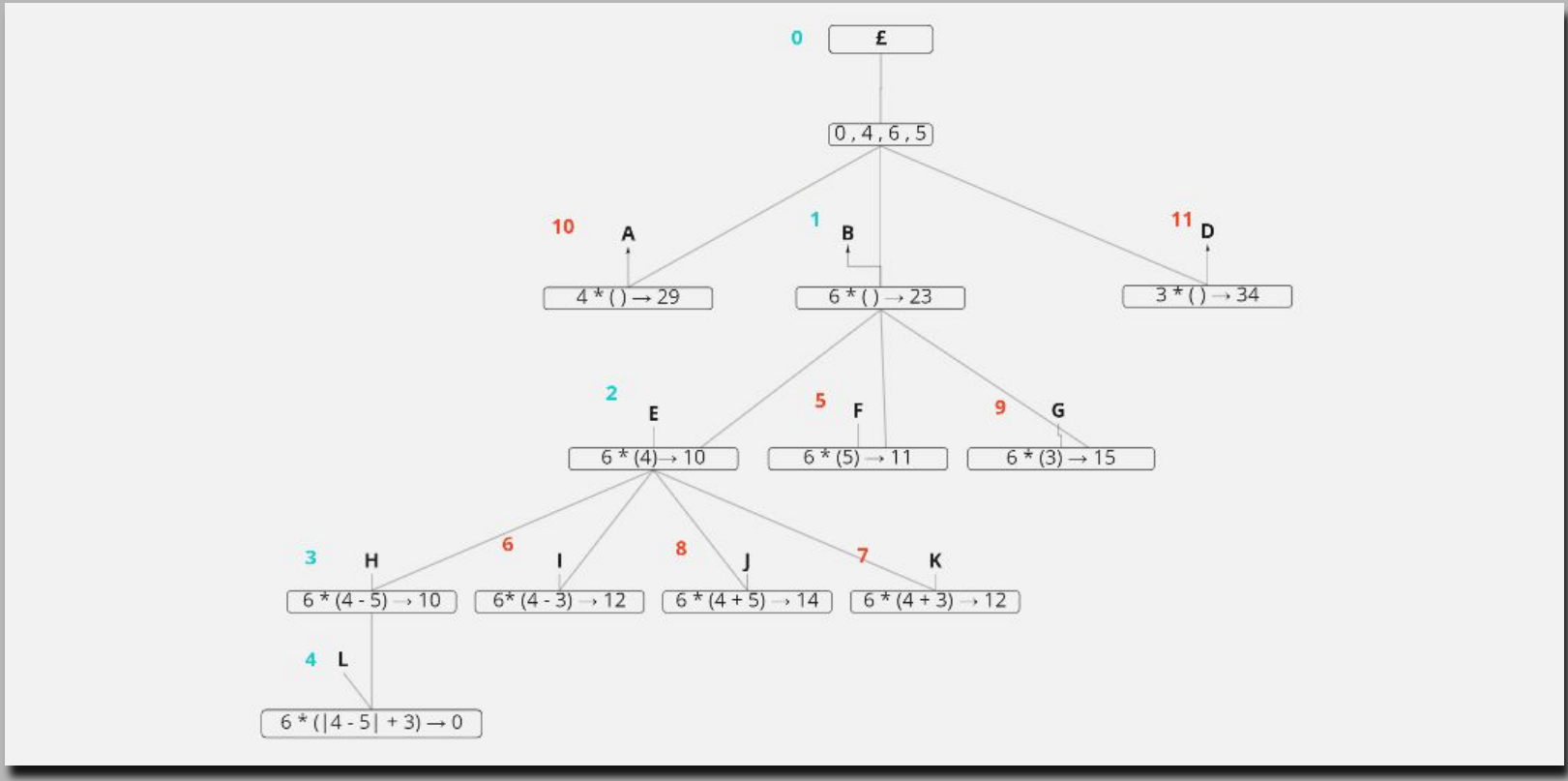
f(G) = g(G) + h(G) = 8 + 4 = 12

	H
VALOR REAL	6 * |(4 - 5 + 3)| = 0
HEURÍSTICA	6 * |(4 - 5 + 3)| = 0

ESTADO ATUAL: **H**

f(L) = g(L) + h(L) = 0 + 0 = 0

Busca A* - Árvore de exemplo



```
void buscaA() {

int jogo[NUM] = {0, 4, 6, 5, 3},
    final[NUM] = {24, 0, 0, 0, 0},
    carta[NUM - 1] = {4, 6, 5, 3},
    cartaX[2] = {100, 0};
 int custo = 0, filho = RAIZ, filhoA = RAIZ;

 vector<char> noFilho, noFilhoA;
 vector<int> filaA, filaH, filaO;
 vector<string> lstAbertos, lstFechados;
 string k = "(0)";
 char f = char(RAIZ);


 cout << "\n\n"
      << "                    *** BUSCA A* ***  \n\n"
      << " - Critério de desempate - \n"
      << "Ordem de entrada na lista de abertos.  \n\n";

 cout << "Estado inicial: " << f << " = ";
 impEstado(jogo);
 cout << "\nEstado final: ";
 impEstado(final);
 cout << "\n\n";

  //!============================================================

 cout << "Inicio\n\n";
 cout << "Valor Real → " ;
 ImpFilhoRaiz(jogo, filhoA);

 cout << "\nHeurística → " ;
 ImpFilhoRaizH(jogo, filhoA);

 filaO = gravaFilaFilho(jogo, noFilho, filho, carta, cartaX);
 filaH = gravaFilaFilhoH(jogo, noFilhoA, filhoA, carta, cartaX);
```

```
for (size_t i = 0; i < filaO.size(); i++)
  filaA.push_back(filaO[i] + filaH[i]);

cout << "\n\nFilhos de " << f << k << ": { ";
impFilhos(filaA, lstAbertos, noFilho);
concStrCharF(k, f, lstFechados);
cout << "\n\n";
for (size_t i = 0; i < 4; i++) {
  impLstFechados(lstFechados);
  ordenaFila(filaA, lstAbertos);
  impLstAbertos(lstAbertos);
  cout << "Estado atual: ";
  cartaX[0] = 100;
  alteraValor(jogo, cartaX[1]);
  impEstado(jogo);

  switch (cartaX[1]) {···
  cout << "Filhos de  " << lstAbertos[0] << ": { ";
  lstFechados.push_back(lstAbertos[0]);
  lstAbertos.erase(lstAbertos.begin());
  limpaNoH(noFilho, noFilhoA, filaA, filaO, filaH);


filaO = gravaFilaFilho(jogo, noFilho, filho, carta, cartaX);
filaH = gravaFilaFilhoH(jogo, noFilhoA, filhoA, carta, cartaX);

for (size_t i = 0; i < filaO.size(); i++)
  filaA.push_back(filaO[i] + filaH[i]);


  //fila = gravaFilaFilhoH(jogo, noFilho, filho, carta, cartaX);

  impFilhos(filaA, lstAbertos, noFilhoA);
  impLstFechados(lstFechados);
  ordenaFila(filaA, lstAbertos);
  impLstAbertos(lstAbertos);
  verificaFinalH(jogo, final, custo, lstFechados);

  cout << "\n";
}
```

Agradecemos a atenção de todos!

Agora vamos a uma breve demonstração da execução do programa.