unesp UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
CAMPUS DE GUARATINGUETÁ

FÁBIO MASCAGNA BITTENCOURT LIMA

RESOLUÇÃO DO PROBLEMA DE CARREGAMENTO E DESCARREGAMENTO DE CONTÊINERES EM TERMINAIS PORTUÁRIOS VIA BEAM SEARCH

Guaratinguetá
2011

FÁBIO MASCAGNA BITTENCOURT LIMA

RESOLUÇÃO DO PROBLEMA DE CARREGAMENTO E DESCARREGAMENTO DE CONTÊINERES EM TERMINAIS PORTUÁRIOS VIA BEAM SEARCH

Trabalho de Graduação apresentado ao Conselho de Curso de Graduação em Engenharia Elétrica da Faculdade de Engenharia do Campus de Guaratinguetá, Universidade Estadual Paulista, como parte dos requisitos para obtenção do diploma de Graduação em Engenharia Elétrica.

Orientador: Prof. Dr. Anibal Tavares de Azevedo

Guaratinguetá
2011

L73
2r

Lima, Fábio Mascagna Bittencourt
Resolução do problema de carregamento e descarregamento de contêineres em terminais portuários via Beam Search / Fábio Mascagna Bittencourt Lima – Guaratinguetá : [s.n], 2011.
48 f : il.

Bibliografia: f. 46-48

Trabalho de Graduação em Engenharia Elétrica – Universidade Estadual Paulista, Faculdade de Engenharia de Guaratinguetá, 2011.

Orientador: Anibal Tavares de Azevedo

1. Algoritmos 2. Java (Linguagem de programação de computador) I. Título

CDU 519.712

unesp UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
CAMPUS DE GUARATINGUETÁ

RESOLUÇÃO DO PROBLEMA DE CARREGAMENTO E DESCARREGAMENTO DE CONTÊINERS EM TERMINAIS PORTUÁRIOS VIA BEAM SEARCH

FÁBIO MASCAGNA BITTENCOURT LIMA

ESTE TRABALHO DE GRADUAÇÃO FOI JULGADO ADEQUADO COMO PARTE DO REQUISITO PARA OBTENÇÃO DO DIPLOMA DE
"GRADUADO EM ENGENHARIA ELÉTRICA"

APROVADO EM SUA FORMA FINAL PELO CONSELHO DE CURSO DE GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Prof.Dr. SAMUEL EUZÉDICE DE LUCENA
Coordenador

BANCA EXAMINADORA:

Prof. Dr. ANIBAL TAVARES DE AZEVEDO
Orientador/UNESP-FEG

Prof. Dr. SAMUEL EUZÉDICE DE LUCENA
UNESP-FEG

Prof. Msc. FERNANDO RIBEIRO FILADELFO
UNESP-FEG

Novembro de 2011

DADOS CURRICULARES

FÁBIO MASCAGNA BITTENCOURT LIMA

NASCIMENTO	27.08.1989 – CASA BRANCA / SP
FILIAÇÃO	Benedito Luiz Bittencourt Lima Janete Isabel Mascagna Correa Bittencourt Lima
2007/2011	Curso de Graduação Engenharia Elétrica – Universidade Estadual Paulista – Faculdade de Engenharia campus Guaratinguetá

de modo especial à minha família, que sempre me apoiou me ajudando a superar os desafios, contribuindo muito para me tornar quem sou e chegar onde eu estou.

AGRADECIMENTOS

Agradeço a Deus, por ter sempre me ajudado nos momentos difíceis com extrema paciência e compaixão.

à minha família, por sempre estar ao meu lado nos momentos que mais precisei com ótimos conselhos e muito amor.

ao meu orientador Prof. Dr. Anibal Tavares de Azevedo, por ter acreditado na minha capacidade para realizar este trabalho.

aos meus amigos, em especial aos da República Amoribunda, que sempre me apoiaram quando precisei tanto na vida acadêmica quanto na pessoal, permitindo momentos muito bons na minha graduação.

LIMA, F. M. B. **Resolução do problema de carregamento e descarregamento de contêineres em terminais portuários via Beam Search.** 2011. Trabalho de Graduação em Engenharia Elétrica – Faculdade de Engenharia do Campus de Guaratinguetá, Universidade Estadual Paulista, Guaratinguetá, 2011.

RESUMO

Um problema que vem acontecendo com frequência nos terminais portuários é o mau planejamento do carregamento e descarregamento de contêineres. Este problema ocorre pela falta de um método eficiente que forneça os melhores modos destas operações serem realizadas. Este trabalho tem por finalidade a implementação de um método que forneça as melhores formas de se realizar o carregamento e o descarregamento de contêineres em cada porto e assim trazer uma grande economia para estes terminais, visto que o número de movimentos é diretamente proporcional ao custo. Para se realizar este programa foi utilizado a ideia de que os contêineres são colocados em pilhas verticais nas quais o acesso a eles é feito somente através do topo da pilha, assim o navio foi tratado como uma matriz e para o seu preenchimento foram criadas duas regras para o carregamento e duas para o descarregamento. Como o problema resultante é combinatório, é necessário empregar uma heurística na tentativa de se obter a melhor sequência de regras. Para tanto, foi utilizado um método do tipo enumeração implícita que analisa apenas alguns dos nós da árvore de acordo com uma heurística gulosa, o *Beam Search*. Assim, o programa desenvolvido na linguagem Java fornece uma boa solução para o carregamento e descarregamento nos portos em um tempo computacional adequado. Além disso, a interface gráfica permite a visualização do estado do navio em cada porto.

PALAVRAS-CHAVE: *Beam Search*. Carregamento de Contêiner. Java.

LIMA, F. M. B. **Solving the container ship stowage problem by Beam Search.** 2011. Undergraduate Conclusion Work in Electrical Engineering – Faculdade de Engenharia do Campus de Guaratinguetá, Universidade Estadual Paulista, Guaratinguetá, 2011.

ABSTRACT

One problem that has been happening frequently in port terminals is the poor planning of the loading and unloading of containers. The reason of this problem is the lack of an efficient method that provides the best means of these operations. The main goal of this work is, to implement a method that provides the best ways to perform the loading and unloading of containers, at each port and thus bring a great saving for these terminals, since the number of moves is directly proportional to cost. To carry out this program was used the idea that the containers are placed in vertical stacks, where the access can be done only by the top of the stack, so the ship was treated as an matrix and to fill it, two rules were created for loading and two for unloading. To obtain the best sequence of rules was used Beam Search method, which is an enumeration type implicit method that analyzes only the best solution of the tree generated. Thus, the program developed in the Java language, provides the best way to perform the loading and unloading ports and the way as the ship leaves each port using a graphical interface.

KEYWORDS: Container Ship Stowage. Beam Search. Java.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 – INTRODUÇÃO

Este item foi fundamentado na referência AZEVEDO; RIBEIRO; LIMA, 2009. É conveniente ressaltar que o orientador e o aluno autor deste trabalho de graduação também são autores desta referência.

Este trabalho de graduação tem por objetivo minimizar o custo do carregamento e descarregamento de contêineres em terminais portuários via *Beam Search*.

Neste trabalho de graduação apresentaremos o método *Beam Search* para resolver este problema. Em um navio porta contêiner os contêineres são colocados em pilhas verticais, localizadas em diversas seções (baias). O acesso aos contêineres é feito somente através do topo da pilha. Muitas vezes para se descarregar um contêiner em um determinado porto *j*, é necessário remover o contêiner cujo destino é o porto *j*+1, porque ele está acima do contêiner que se deseja descarregar, esta operação pode ser chamada de remanejamento. Um navio porta contêiner transportando carga para vários portos pode necessitar de muitas operações de remanejamento. Esses remanejamentos possuem custo e despendem tempo, contudo alguns deles podem ser evitados através de um planejamento eficiente.

A eficiência de um terminal portuário especializado em movimentação de contêineres depende da ordenação e agilidade do processo de lidar com os contêineres, especialmente durante o processo de carregamento dos navios. A estiva e o plano de carregamento associado são determinados fundamentalmente por dois critérios: estabilidade do navio e o número mínimo de remanejo requerido nos diversos pontos de entrega (AMBROSINO; SCIOMACHEN; TANFANI, 2006*;* AVRIEL; PENN; SHPIRER, 2000*;* WILSON; ROACH, 2000). O último critério é baseado no fato de que muitos navios possuem uma estrutura celular, conforme pode ser observado na Figura 1, e os contêineres devem ser carregados de modo a formarem pilhas verticais, o que acarreta, em muitos casos, a necessidade de movimentar alguns contêineres para descarregar outros posicionados na parte inferior da pilha.

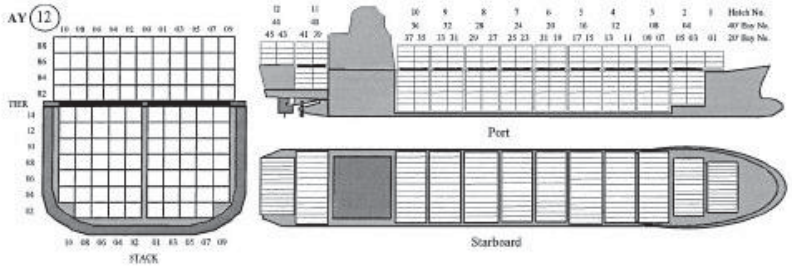

FIGURA 1: *Estrutura celular de um navio (WILSON; ROACH, 2000).*

Concomitantemente, outra restrição emerge durante a escolha dos contêineres para carregamento no pátio do terminal, onde geralmente os contêineres são empilhados formando blocos a espera do momento de serem carregados. Se os contêineres alvos, que devem ser carregados mais tarde, são posicionados nas pilhas abaixo de outros, então a tarefa de carregamento requer remanejo de modo a remover e reposicionar os contêineres alvos. Esta situação ocorre com frequência, uma vez que a ordem de carregamento não é conhecida quando as cargas chegam ao pátio do terminal. No entanto, mesmo quando esta informação é disponibilizada a tempo, o arranjo ideal de contêineres na área de armazenamento é praticamente impossível de ser obtido devido à chegada aleatória de diversas outras cargas.

Mais especificamente, o problema de carregamento de contêineres em terminais portuários (PCCTP) consiste em determinar como carregar um conjunto de contêineres de diferentes tipos em um navio porta-contêiner (*containership*), respeitando restrições operacionais relacionadas aos contêineres, navio e pátio do terminal portuário. Neste trabalho de graduação será apresentado um *Beam Search* para a solução do PCCTP, ou seja, para a minimização do custo de carregamento e posterior descarregamento dos contêineres em terminais portuários.

1.1 - Apresentação do Problema

Um navio porta contêiner tem sua capacidade medida em TEU (*Twenty-foot Equivalent Units*) ou Unidade Equivalente de Vinte Pés. Por exemplo, um navio com capacidade de 8000 TEUs pode carregar 8000 contêineres de vinte pés. Os navios têm uma estrutura celular onde são alojados os contêineres. Essas células são agrupadas por seções ou baias (*bays*) e os contêineres são empilhados nessas seções formando pilhas verticais. Então uma baia é um agrupamento de células, com capacidade de se empilhar certo número de contêineres. Em geral cada baia tem capacidade para alocar quarenta contêineres de vinte pés. A baia tem então linhas horizontais numeradas r = 1, 2,... R, (a linha 1 é a linha que está em baixo, e a linha R é a linha do topo da pilha) e colunas numeradas c= 1, 2, ..., C (coluna 1 é a primeira coluna da esquerda).

O problema PCCTP resolvido aqui consiste em reduzir o número de remanejamentos dos contêineres para um número N de portos. Define-se remanejamento como sendo o descarregamento temporário de contêineres, da pilha de contêineres, com a finalidade de descarregar, em um terminal portuário p, um contêiner que está na parte inferior da pilha. Isto é necessário porque os contêineres que estão numa pilha só podem ser acessados pelo topo.

Então um contêiner que está no meio da pilha só pode ser descarregado em um determinado porto *p* se os contêineres que estão acima dele forem removidos. A seguir será apresentada a formulação deste problema como sendo um problema de programação linear inteira. Esta formulação respeita as restrições operacionais relacionadas aos contêineres, navio e pátio do terminal portuário e aparecem de modo mais detalhado em Avriel, Penn, Shpirer e Wittenboon (1998)*;* Avriel e Penn (1993) *e* Botter e Brinati (1992).

Considere um navio de transporte de contêineres que possui uma única baia. A baia tem *R* linhas horizontais numeradas *r* = 1, 2, ...*R*, (a linha 1 é a linha que está em baixo, e a linha *R* é a linha do topo da pilha) e *C* colunas verticais numeradas *c*= 1, 2, ..., *C* (coluna 1 é a primeira coluna da esquerda). Apesar de a baia ter um formato tridimensional, a mesma pode ser representada, sem perda de generalidade, por um formato bidimensional, em particular uma matriz. Então, uma baia pode alocar no máximo $R \times C$ contêineres. É assumido também que todos os contêineres têm o mesmo tamanho. O navio chega ao porto 1 completamente vazio e sequencialmente ele visita os portos 2, 3, ..., *N*. Em cada porto *i* = 1,...,*N*-1, o navio recebe o carregamento de contêineres com destino aos portos *i*+1,...,*N*. No último porto ele descarrega os contêineres e fica totalmente vazio. Seja $T=[T_{ij}]$ a matriz de transporte de dimensão (*N*-1)×(*N*-1), onde T_{ij} é o número de contêineres com origem em *i* e destino em *j*. Esta matriz é triangular superior porque $T_{ij}=0$ para todo $i \geq j$.

A formulação de programação linear inteira do PCCTP é dada pelas Equações (1)-(6).

$$Min: f(x) = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \sum_{v=i+1}^{j-1} \sum_{r=1}^{R} \sum_{c=1}^{C} x_{ijv}(r,c) \qquad i = 1,\cdots,N-1,\ j = i+1,\cdots,N; \qquad (1)$$

$$\text{s.a:} \quad \sum_{v=i+1}^{j} \sum_{r=1}^{R} \sum_{c=1}^{C} x_{ijv}(r,c) - \sum_{k=1}^{i-1} \sum_{r=1}^{R} \sum_{c=1}^{C} x_{kji}(r,c) = T_{ij} \qquad i = 1,\cdots,N-1,\ j = i+1,\cdots,N; \qquad (2)$$

$$\sum_{k=1}^{j} \sum_{j=i+1}^{N} \sum_{v=i+1}^{j} x_{kjv}(r,c) = y_i(r,c) \qquad i = 1,\cdots,N-1, r = 1,\cdots,R; \qquad (3)$$
$$c = 1,\cdots,C;$$

$$y_i(r,c) - y_i(r+1,c) \geq 0 \qquad i = 1,\cdots,N-1, r = 1,\cdots,R-1; \qquad (4)$$
$$c = 1,\cdots,C;$$

$$\sum_{i=1}^{j-1}\sum_{p=j}^{N} x_{ipj}(r,c) + \sum_{i=1}^{j-1}\sum_{p=j+1}^{N}\sum_{v=j+1}^{p} x_{ipv}(r+1,c) \le 1 \qquad i = 2,\cdots,N, r = 1,\cdots,R-1; \quad c = 1,\cdots,C; \tag{5}$$

$$x_{ijv}(r,c) = 0 \text{ ou } 1;\ y_i(r,c) = 0 \text{ ou } 1 \tag{6}$$

A variável binária $x_{ijv}(r,c)$ é definida de forma que assume o valor 1 se existir um contêiner no compartimento (*r*,*c*) que foi ocupado no porto *i* e tem como destino final o porto *j* e movido no porto *v*; caso contrário assume valor zero. Por compartimento (*r*,*c*) entende-se a linha *r* e a coluna *c* no compartimento de carga do navio. É necessário salientar que a numeração das linhas é feita de baixo para cima, assim a linha de número 5 está acima da linha de número 4, e a numeração das colunas é feita da esquerda para a direita.

Similarmente, a variável $y_i(r,c)$ possui valor 1 se saindo do porto *i* o compartimento (*r*, *c*) for ocupado por um contêiner; caso contrário assume valor 0. A função objetivo da Equação (1) fornece o custo total de movimentação dos contêineres (assumindo que a movimentação de um contêiner possui um custo unitário e igual para todos os portos) em todos os portos. A restrição (2) é a restrição de conservação de fluxo, onde T_{ij} é o elemento da matriz de transporte que representa o número de contêineres que embarcam no porto *i* com destino ao porto *j*. A restrição (3) garante que cada compartimento (*r*,*c*) tem no máximo um único contêiner. A restrição (4) é necessária para garantir que existem contêineres embaixo do contêiner que ocupa o compartimento (*r*,*c*). A restrição (5) é responsável por definir a movimentação dos contêineres: se um contêiner que ocupa a posição (*r*,*c*) é descarregado no porto *j*, então, ou não existem contêineres acima dele, ou o índice *v* do contêiner que ocupa o compartimento (*r*+1,*c*) não é maior que *j*. Infelizmente, o tamanho que o problema assume com a formulação dada pelas Equações de (1) a (6) é proibitivo para problemas reais e só pode ser resolvido para problemas pequenos. Além disso, em Avriel e Penn (1993) é demonstrado que o PCCTP é um problema NP-Completo, justificando o emprego de heurísticas para encontrar boas soluções.

Outro problema é relacionado com a questão da representação da solução por meio de variáveis binárias. A formulação (1) a (6) é tal que para se representar uma solução de uma instância com $R = 6$, $C = 50$, $P = 30$ serão necessárias $R*C*P^3$ variáveis $x_{ijv}(r,c)$, ou seja, 900000 variáveis *x*, e $R*C*P$ variáveis $y_i(r,c)$, ou seja, 9000 variáveis *y*. Ou seja, um total de 909000 variáveis para representar uma única solução.

2 – DESENVOLVIMENTO

Este item foi fundamentado na referência AZEVEDO; RIBEIRO; LIMA, 2009. É conveniente ressaltar que o orientador e o aluno autor deste trabalho de graduação também são autores desta referência.

2.1 - Representações por Regras

Aqui será apresentada a representação por regras desenvolvida para a resolução PCCTP. Esta representação tem a vantagem de ser compacta e de assegurar que todas as soluções geradas pelo método sejam factíveis.

Na Figura 1 viu-se que os navios possuem uma estrutura celular de modo que os locais onde os contêineres serão alocados são pré-determinados fazendo com que os contêineres sejam empilhados verticalmente. Este empilhamento sugere uma representação matricial dos contêineres no navio. Deste modo pode-se definir uma matriz de ocupação *B* que fornece a quantidade de espaços disponíveis e a localização dos contêineres no navio em cada porto.

Para tanto, cada elemento da matriz *Brc* representa o estado de uma célula (*r*,*c*), isto é se *Brc*=0 significa que a célula (*r*,*c*) está vazia e se *Brc*=*j* significa que a célula contém um contêiner cujo destino é o porto *j*. Assim, no Exemplo se o elemento (1,1) é igual a 5 significa que neste local existe um contêiner que será descarregado no porto 5. De modo análogo, o elemento (3,3) = 0 significa que esta célula está vazia. Lembrando que a linha 0 da matriz representa o topo da pilha de carregamento.

Esta matriz de carregamento é modificada em todos os portos devido à entrada e saída de novos contêineres em todos os portos, pois quando o navio chega num porto *j* é necessário realizar dois movimentos obrigatórios, a saber: descarregar os contêineres cujo destino é o porto *j* em questão e carregar os contêineres com destinos aos portos *j*+1, *j*+2,..., *N*. Então, para todos os portos *j* foram estabelecidas regras para se fazer o carregamento e descarregamento dos contêineres.

Muitas vezes, para se fazer o descarregamento no porto *j*, de um contêiner cujo destino é o porto *j*, é necessário fazer operações de remanejamento dos contêineres cujos destinos são os portos de *j*+1 até *N*, porque a posição que eles ocupam na pilha está acima da posição do contêiner do porto *j*.

O carregamento de contêineres num dado porto *j* deve-se levar em conta os contêineres que já estão no navio, porque foram embarcados nos portos anteriores (portos de 1 até *j*-1)

com destino aos portos *j*+1 até *N*. Observe então que existe uma relação íntima entre as operações de carregamento e descarregamento, tendo em vista que, a maneira como é feita o carregamento num porto *j* vai influenciar no descarregamento a ser feito nos demais portos (portos de *j*+1 até *N*). Portanto, para reduzir as operações de remanejamento é necessário estabelecer regras para o carregamento e descarregamento de contêineres em todos os portos que levam em conta esta relação. Para tanto foram criadas quatro regras, sendo duas para o carregamento (*Rc1*, *Rc2*) e duas para o descarregamento (*Rd1*, *Rd2*). A combinação de uma regra de carregamento com uma de descarregamento fornece a regra *k* para o porto *j*.

Regra *k* usada no porto *j*	**Regra de carregamento**	**Regra de descarregamento**
1	*Rc1*	*Rd1*
2	*Rc1*	*Rd2*
3	*Rc2*	*Rd1*
4	*Rc2*	*Rd2*

TABELA 1: *Regras k a serem utilizadas em cada porto j (Fonte: autor).*

Observe na Tabela 1 que a regra 2 foi obtida utilizando a regra *Rc1* para o carregamento dos contêineres e a regra *Rd2* para o descarregamento. Isto foi feito com objetivo de se obter uma representação compacta da solução.

As aplicações destas regras em cada porto *j* vai atualizar a Matriz de Ocupação no porto *j*. Vale lembrar que inicialmente a matriz *M* está com todos seus elementos iguais a zero e ela começa a ser preenchida no porto 1. Para a realização das ações é utilizada a matriz de transporte *T*, que fornece a quantidade de contêineres que devem ser embarcados em cada porto *i* com destino a cada porto *j*.

- **Regra *Rc1*:** Esta regra preenche a matriz de ocupação *B* (no porto *p*) por linha, da esquerda para a direita, colocando na parte inferior da pilha as cargas cujo destino é mais distante.
- **Regra *Rc2*:** Esta regra faz o preenchimento da matriz de ocupação *B* em um porto *p* preenchendo cada coluna até a linha *qp*, colocando, inicialmente, as cargas cujo destino é mais distante. A linha *qp* é calculada através da equação:

$$qp = \frac{(\sum ContExistentes + \sum contEmbarcados + \sum contDesembarcados)}{coluna} \qquad (7)$$

Na qual, *qp* é a linha limite para o carregamento, *contExistentes* são os contêineres já existentes no navio, *contEmbarcados* e *contDesembarcados* são os contêineres embarcados e desembarcados no porto em questão e *coluna* é o número de colunas da matriz B.

Como *qp* indica a linha limite de carregamento, esta deve ser um número inteiro. Assim, caso a equação retornar um valor com casas decimais, este valor deve ser arredondado para cima.

- **Regra *Rd1*:** Nesta regra quando o navio chega a um porto *p*, são removidos todos os contêineres cujo destino é *p* e todos os contêineres que estão acima dos contêineres do porto *p* e cujos destinos são os portos *p+j*, para *j=1, ...,N-p*. Estes contêineres removidos com destino a portos mais adiantes, deverão ser incrementados na matriz transporte *T*. Assim, estes serão carregados da melhor maneira possível juntamente com os contêineres já estabelecidos para aquele porto.
- **Regra *Rd2*:** Nesta regra quando o navio chega ao porto *p*, todos os contêineres são removidos. Os contêineres removidos cujos destinos são portos mais adiantes, deverão ser incrementados na matriz transporte *T*. Assim, estes serão carregados da melhor maneira possível, juntamente com os contêineres já estabelecidos para aquele porto.

As vantagens do emprego de regras são:

- A facilidade de incorporar conhecimento prévio do decisor sob a forma de regras;
- As regras só podem produzir matrizes de Ocupação factíveis, facilitando e garantindo, a obtenção de soluções factíveis por métodos heurísticos (neste trabalho de graduação, um *Beam Search*).

2.2 - Algoritmos Beam Search

O *Beam Search* foi usado pela primeira vez pela comunidade de Inteligência Artificial para tratar problemas de reconhecimento de fala (LOWERRE, 1976). A literatura fornece várias aplicações deste método em problemas de sequenciamento da produção (SABUNCUOGLU; BAVIZ, 1999; DELLA CROCE; T'KINDT, 2002; FOX, 1983; OW; MORTON, 1988; VALENTE; ALVES, 2005). O algoritmo do *Beam Search* é um método do tipo Enumeração Implícita para resolver problemas de Otimização Combinatória. Pode-se

dizer que ele é uma adaptação do método de *Branch and Bound* onde somente os nós mais promissores de cada nível da árvore de decisões (atribuições) são guardados na memória para serem visitados, enquanto que os demais nós são descartados permanentemente. Como uma grande parte dos nós da árvore de atribuições é descartada, isto é, somente alguns poucos nós são selecionados para serem analisados; o tempo de execução do método é polinomial com relação ao tamanho do problema. Em resumo pode-se dizer que o *Beam Search* é uma técnica de busca em árvore de decisão que em cada nível da árvore é analisado um número fixo de nós e, por conseguinte um número fixo de soluções. O número de nós analisados em cada nível é chamado de **largura da busca** e é denotado por β.

Para se construir a árvore de decisões é necessário estabelecer as seguintes definições:

(D.1) A árvore é construída por nível e em cada **nível *i*** é feita à atribuição de uma regra ***k*** ao ***i-ésimo*** porto.

(D.2) Os nós que estão no **nível 1** da árvore são chamados de nós semente porque cada um deles vai gerar uma sub-árvore de decisões.

(D.3) A cada nível ***i,*** ao se realizar a atribuição no ***i*-ésimo** porto de uma regra ***k***, são contabilizados os movimentos de carregamento e descarregamento da regra ***k*** adotada no porto ***i*** mais os movimentos de carregamento e descarregamento das regras utilizadas nas (***i*-1**) atribuições anteriores.

Tendo em vista que a árvore possui ***N*-1** níveis **(D.1)**, onde ***N*** é o número de portos, uma solução completa, com a atribuição de ***m*** regras à todos os ***N*** portos só será obtida ao se definir as atribuições até o nível ***N*-1** da árvore.

A regra *k* é aplicada para todos os portos, com exceção do porto *N*, no qual apenas a regra de descarregamento é utilizada e para o porto 1 no qual apenas a regra de carregamento é realizada.

Para gerar a árvore de soluções, é possível perceber que o número de nós é diretamente proporcional às quantidades de regras e ao número de portos do problema abordado. O número de nós pode ser obtido através de equacionamento matemático, assim como o número de soluções factíveis. A seguir serão colocadas estas equações segundo Ribeiro e Azevedo (2010).

Assim para se obter o número de nós total da árvore, será utilizada a equação dada por:

$$Qn = \frac{m(m^{(N-1)} - 1)}{(m-1)}; \tag{8}$$

Na qual ***m***= número de regras e ***N***= número total de portos.

O número total de soluções factíveis (*Fs*) será dado por:

$$Fs = m^{(N-1)}; \quad (9)$$

Porém o método *Beam Search* analisa apenas os nós mais promissores e assim o número de nós analisados será dado por:

$$Qn = m(N-1); \quad (10)$$

E o número de total de soluções factíveis analisadas será dado por:

$$Fs = \beta; \quad (11)$$

2.3 - Árvores de Soluções Geradas pelo *Beam Search*

Para se construir a árvore o algoritmo utilizado será mostrado a seguir:

Inicio

nível = 0

Enquanto (**nível <** número de portos **N-1**) faça:

1º Passo:

Faça **nível = nível + 1**

P1.1. Criar os nós deste novo nível , um nó para cada regra.

P1.2. Para cada nó criado no **nível *i***, faz-se:

- Armazenar a identificação da regra e do porto;
- Calcular e armazenar o número de movimentos de descarregamento e carregamento relativa à atribuição efetuada no nó;
- Calcular e armazenar o custo parcial da solução, isto é, o número de movimentos de descarregamento e carregamento relativa à atribuição efetuada do nó semente até o nó criado;

Se (**nível** < número de portos ***N*-1**)

Então

2º Passo.

Para cada nó criado no ***1º. Passo*** **(P1.1)** e considerando-se os custos calculados até então, ache uma **solução gulosa** para o problema, através de uma **busca em profundidade** na árvore, partindo deste nó.

3o. Passo

- Conserve na árvore, neste nível, os nós que geraram as β melhores soluções gulosas;
- Descarte os demais nós.

Fim Enquanto

Inicialmente o algoritmo encontra-se no **nível zero** de solução, pois as atribuições começarão a serem feitas agora. A seguir faz-se **nível = 1** e começam a serem criados os nós **(P1.1)** deste nível. Os nós do **nível** 1 são os nós sementes e é criado um nó semente para cada regra. Em **(P1.2**), para cada nó criado em **(P1.1)** é atribuída uma regra no vetor de portos. A seguir, calculam-se o número de movimentos de descarregamento e carregamento do nó.

A busca em profundidade realizada no ***2º Passo***, é feita utilizando um algoritmo do tipo guloso e ela visa escolher, no ***3o. Passo***, quais os nós, do ***nível i***, devem permanecer na árvore de soluções factíveis, de modo a respeitar a largura de busca β. Por exemplo se β**=2**, em cada nível vão permanecer somente dois nós. A forma que esta busca em profundidade é realizada será demonstrada no próximo item.

2.4 - Busca em Profundidade do Passo2- Algoritmo Guloso

Para cada nó ***j*** criado no ***1º. passo*** **(P1.1)** fazer uma arborescência em profundidade, a partir deste nó, descendo até o último nível, isto é:

- Seja um nó ***j*** criado no **nível *i*** com custo de solução parcial ***Sj***, calculada até o **nível *i***;
- Escolha para fazer parte desta solução o nó ***p*** do **nível *i+1*** que é factível e que gere a menor solução parcial ***Sp***, onde ***Sp* = *Sj*** + número de movimentos de descarregamento +número de movimentos de carregamento relativa à atribuição efetuada no nó ***p;***
- Repetir este processo a partir do nó ***p***, descendo até o último nível. Somente após descer no último nó do ramo é que se terá o custo total de uma solução gulosa.

Como já foi dito anteriormente, as soluções gulosas servem para escolher dentre os nós do **nível *i*** que foram criados no ***1º. passo,*** aqueles que vão permanecer na árvore. Essa escolha é feita da seguinte maneira:

- Ordenar os nós criados no **nível *i***, em ordem crescente, de acordo com os resultados dos custos das soluções gulosas que cada um deles gerou;

- Entre os nós do **nível *i*** escolher para permanecer na árvore os ***β*** nós que geraram as soluções gulosas de menor custo;
- Cortar os demais nós do **nível *i***, isto é: Se largura de busca é ***β*=2**, só vão continuar a fazer parte da árvore os dois nós que geraram as duas soluções gulosas de menor custo. O valor da menor solução gulosa calculada no ***2º. Passo***, também é usado como limitante superior (corte) para se gerar ou não os outros nós da árvore, no ***1º. passo***. Este limitante superior deve ser atualizado à medida que forem encontradas soluções gulosas menores. Com isto têm-se como critério de corte o custo da menor solução gulosa, isto é, se o custo parcial da solução no nó que acabou de ser criado, for maior que o custo da melhor solução gulosa, este nó deve ser excluído da solução.

Depois de terminada a Busca em Profundidade deve-se verificar se o nível da árvore de decisões é igual ao número de portos. Se este for o caso, o algoritmo termina, pois já se chegou ao final da árvore. Senão, o algoritmo volta ao ***1º. passo*** para gerar os nós do próximo nível.

Assim, com o *Beam Search* as soluções são codificadas em uma notação compacta que sempre fornece soluções factíveis. Esta estratégia permite o tratamento de problemas de maior porte em tempo computacional razoável.

3 – PROGRAMA DESENVOLVIDO

As seções anteriores forneceram subsídios que permitem a resolução do modelo matemático do PCCTP por meio da combinação da representação por regras e do *Beam Search*. A partir da representação das soluções por regras foi elaborado um conjunto de classes, implementadas em linguagem Java. Esta versão já passou por etapas de testes e ajustes e gerou a resposta esperada. A seguir será detalhado o diagrama UML, bem como o código associado, das classes que compõem o programa.

3.1 - Diagramas UML

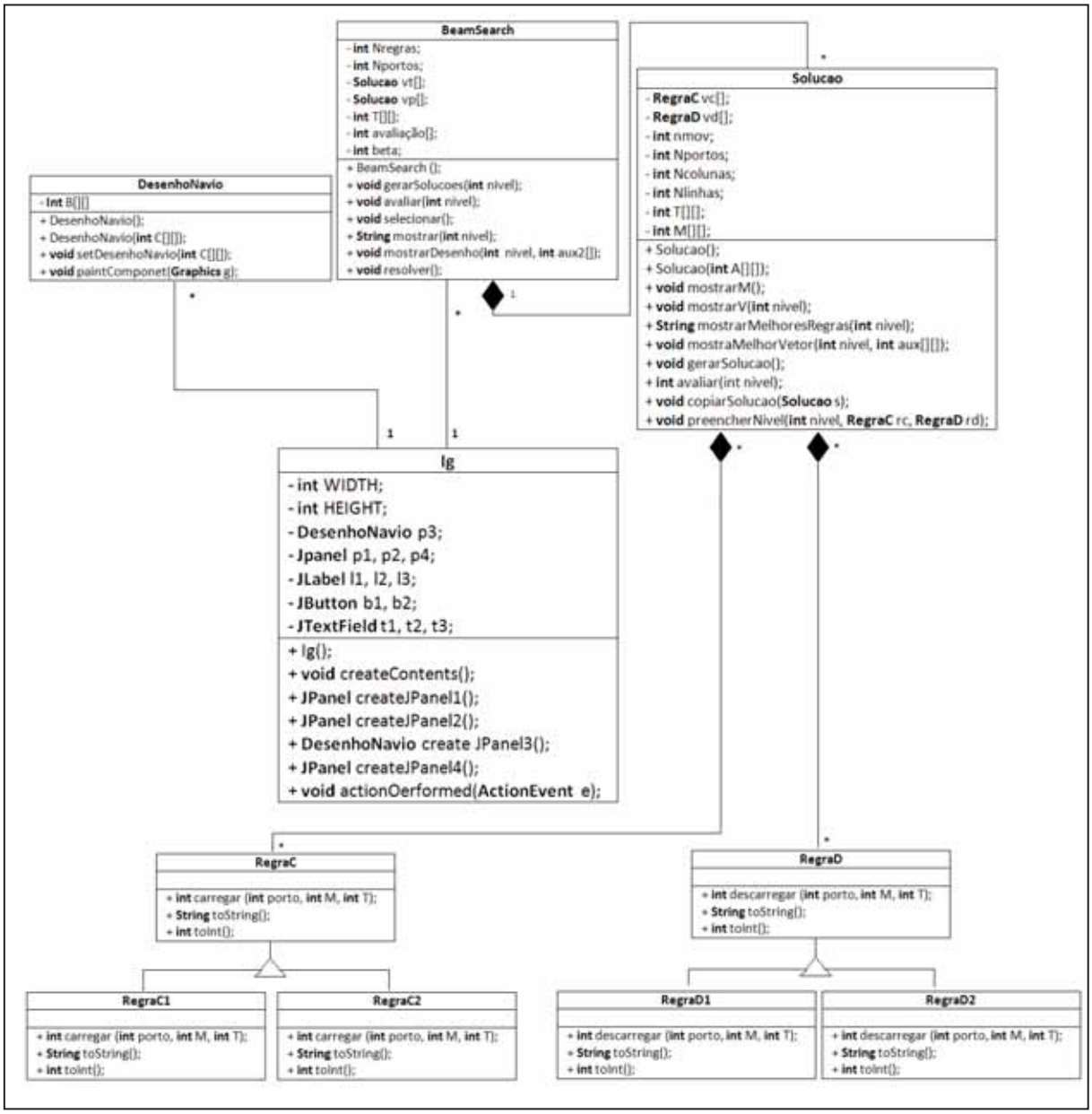

FIGURA 2: *Diagrama UML (Fonte: autor).*

3.2 - Descrição dos métodos de cada classe

3.2.1 - Classes de Carregamento

Neste item serão descritos os métodos referentes ao carregamento do navio.

3.2.1.1 - Classe RegraC1

Método: Carregar.

Descrição: Este método recebe como parâmetros o *porto*, a matriz ocupação *M* e a matriz transporte *T* e tem por função inicializar os campos linhas, coluna e o porto atual. Após isto ele percorre a matriz *M* (matriz ocupação) por linha, da esquerda para a direita, colocando na parte inferior os contêineres com destino mais distantes. Este método retorna o número de movimentos.

Método: toString.

Descrição: Este método não recebe nenhum parâmetro e tem por função retornar a *String* "Regra C1".

Método: toInt.

Descrição: Este método não recebe nenhum parâmetro e tem por função retornar o número inteiro 1.

3.2.1.2 - Classe RegraC2

Método: Carregar.

Descrição: Este método recebe como parâmetros o *porto*, a matriz ocupação *M* e a matriz transporte *T* e inicializa os campos somatória, que corresponde ao número de contêineres no navio, *limite*, que corresponde à última linha que os contêineres poderão ser alocados, *temp* que armazenará a linha da matriz *T*, *controlelinha*, que armazena limite para este não ser perdido após este ser decrementado, além de *linha* e *coluna*. Este método preenche a matiz *M* percorrendo cada coluna, da esquerda para a direita, até a linha limite (Equação 8).

Nem sempre *limite* é um valor inteiro, porém este deverá ser inteiro, então é utilizada a

conversão de *float* para inteiro e posteriormente é conferido a igualdade entre ambos os termos: se forem diferentes, é somado um ao termo convertido a int e este valor é atribuído a *limite* e *controlelinha*, caso contrario o valor *int* é apenas atribuído a *limite* e *controlelinha*.

Este método retorna o número de movimentos.

Método: toString.

Descrição: Este método não recebe nenhum parâmetro e tem por função retornar a *String* "Regra C2".

Método: toInt.

Descrição: Este método não recebe nenhum parâmetro e tem por função retornar o número inteiro 2.

3.2.2 - Classes de Descarregamento

Neste item serão descritos os métodos referentes ao descarregamento do navio.

3.2.2.1 - Classe RegraD1

Método: Descarregar.

Descrição: Este método recebe como parâmetros o *porto*, a matriz ocupação *M* e a matriz transporte *T* e inicializa o número de movimentos e percorre a matriz em busca de contêineres que devem ser descarregados no porto "*porto*", se encontrado, é conferido a existências de outro contêiner acima deste, caso encontrado este também é retirado e colocado na matriz *T*.

Este método retorna o número de movimentos.

Método: toString.

Descrição: Este método não recebe nenhum parâmetro e tem por função retornar a *String* "Regra D1".

Método: toInt.

Descrição: Este método não recebe nenhum parâmetro e tem por função retornar o número inteiro 3.

3.2.2.2- Classe RegraD2

Método: Descarregar.

Descrição: Este método recebe como parâmetros *porto*, a matriz ocupação *M* e a matriz transporte *T* e inicializa o numero de movimentos, e a linha e coluna que o contêiner que não deveria ser descarregado assumirá na matriz transporte *T*. Este percorre toda a matriz descarregando todos os contêineres, porém se encontrado o contêineres para o porto em questão, apenas número de movimentos é incrementado, mas se for encontrado um contêineres com outro destino, o número de movimentos é incrementado e o contêineres deverá ser colocado na matriz *T* para que no próximo carregamento este volte à matriz *M*.

Assim, a cada movimento de retirada de um container o número de movimentos é incrementado e o valor deste parâmetro é retornado no final do método.

Método: toString.

Descrição: Este método não recebe nenhum parâmetro e tem por função retornar a *String* "Regra D2".

Método: toInt.

Descrição: Este método não recebe nenhum parâmetro e tem por função retornar o número inteiro 4.

3.3- Classe referente a Solução

3.3.1 - Classe Solucao

Método: solução.

Descrição: Este método não recebe parâmetro nenhum e tem por função inicializar o número de colunas (*Ncolunas*), o número de linhas (*Nlinhas*), número de portos (*Nportos*) além de *vc* e *vd* que são definidos como objetos de *RegraC* e *RegraD* respectivamente e também *M* que é definido como *new int*.

Este método percorre toda a matriz *M* preenchendo-a com zeros.

Método: solução.

Descrição: Este método recebe como parâmetros *T* e inicializa *vc* e *vd* que são objetos de *RegraC* e *RegraD*, número de portos (*Nportos*) e *M* que é definido como *new int*. Este método percorre a matriz *M* preenchendo-a com zeros.

Método: mostrarM.

Descrição: Este método não recebe nenhum parâmetro e tem como função percorrer a matriz *M* escrevendo na tela o seu conteúdo.

Método: mostrarV.

Descrição: Este método recebe como parâmetro o nível até o qual os vetores serão percorridos. Este método tem como função percorrer os vetores *vc* e *vd* escrevendo seus conteúdos na tela.

Método: gerarSolucao.

Descrição: Este método não recebe parâmetros e possui um campo opção, o qual recebe a opção de carregamento e descarregamento. Este método preenche os vetores *vc* e *vd* de acordo com as regras, assim estes vetores são preenchidos de 0 a *Nportos*.

Método: avaliar.

Descrição: Este método recebe como parâmetros o campo *nivel* e inicializa o número de movimentos com zero. Além disto, este percorre a árvore até o nível em questão adicionando a número de movimentos o número de movimentos dos descarregamentos e carregamentos caso estes existirem.

Método: copiarSolucao.

Descrição: Este método tem como função passar as informações contidas na solução que é parâmetro de entrada do método para o objeto solução que o invoca.

Método: preencherNivel.

Descrição: Este método tem como função realizar uma cópia de objetos de regra de entrada e saída, passada por parâmetro, para o vetor que armazena as regras de entrada e saída em um dado nível da árvore.

Método: mostrarMelhoresRegras.

Descrição: Este método recebe como parâmetro o nível até o qual o método será realizado. Este método tem como função retornar os valores *String* dos vetores *Vc* e *Vd* objetos das classes *RegraC* e *RegraD* respectivamente. Este método é requisitado pela interface gráfica.

Método: mostrarMelhorVetor.

Descrição: Este método recebe como parâmetro o nível até o qual o método será realizado. Este método tem como função retornar os valores int dos vetores *Vc* e *Vd* objetos das classes *RegraC* e *RegraD* respectivamente. Este método é requisitado pela interface gráfica.

3.4 - Classe referente ao Beam Search

3.4.1 - Classe Beam Search

Método: beamSearch.

Descrição: Este método não recebe parâmetros e inicializa o *Nregras* (número de combinações entre as regras existentes), o número de portos (*Nportos*), *beta* (largura de busca) e define dois objetos de *Solucao vt* e *vp*, dos quais *vp* é preenchido com zeros de 0 a beta e *vt* também é preenchido com zeros de 0 a *beta*Nregras*. Este método também inicializa com zeros o campo avaliação o qual possui a dimensão equivalente a *Nregras*beta*.

Método: gerarSolucao.

Descrição: Recebe como parâmetros o nível desejado. Caso este for igual a zero deve realizar o método *preencherNivel* com os parâmetros *nivel*, *RegraC* e *RegraD* definidas através da largura de busca que varia de 0 a *beta*. Caso o nível for diferente de zero deve-se gerar *Nregras*beta* soluções novas e para cada solução o campo *ind* (índice de *vt*) é definido por *i*(que varia de 0 a *Nregras*) e *j* (que varia de 0 a *beta*).

Assim para o índice *ind* de *vt* é utilizado o método *copiarSolucao* para copiar a solução de *vp[i]*, e então deve-se preencher o nível atual assim como o caso *nivel*=0, porém *Nregras* vezes.

Método: avaliar.

Descrição: Recebe como parâmetro o campo *nivel*. Este método utiliza o método avaliar da classe *Solucao* para preencher o vetor avaliação.

Caso o *nivel* for igual a zero, este vetor avaliação é preenchido de zero a beta. Caso contrario este preenchimento ocorre de zero a *vt.length*, que corresponde a dimensão do vetor *vt*.

Método: selecionar.

Descrição: Não recebe nenhum parâmetro e tem como função colocar *vt* em ordem crescente utilizando como critério o número de movimentos (método *avaliacao*). Deve-se agora selecionar as *betas* melhores soluções.

Método: resolver.

Descrição: Não recebe nenhum parâmetro e tem por função utilizar os demais métodos para resolver o problema. Inicializa o *nivel* para zero e percorre até *Nportos* realizando os métodos *gerarSolucoes*(*nivel*), *avaliar*() e *selecionar*().

3.5 - Códigos das Classes

```
public abstract class RegraC
{
    public abstract int carregar (int porto, int M[][], int T[][]);
    public abstract  String toString();
    public abstract int toInt();
}
```

***FIGURA 3**: Classe RegraC (Fonte: autor).*

```
public class RegraC1 extends RegraC
{
   public int carregar (int porto, int M[][], int T[][])
    public String toString()..
    public int toInt()..
}
```

***FIGURA 4:** Classe RegraC1 (Fonte: autor).*

```
public class RegraC2 extends RegraC
{
    public int carregar (int porto, int M[][], int T[][])
    public String toString()..
    public int toInt()..
}
```

***FIGURA 5:** Classe RegraC2 (Fonte: autor).*

```
public abstract class RegraD
{
  public abstract int descarregar(int porto,int M[][], int T[][]);
  public abstract  String toString();
  public abstract int toInt();
}
```

FIGURA 6: *Classe RegraD (Fonte: autor).*

```
class RegraD1 extends RegraD
{
  public int descarregar(int porto, int M[][], int T[][])
  public String toString()..
  public int toInt()..
}
```

FIGURA 7: *Classe RegraD1 (Fonte: autor).*

```
class RegraD2 extends RegraD
{
  public int descarregar(int porto, int M[][], int T[][]) ..
  public String toString()..
  public int toInt()..
}
```

FIGURA 8: *Classe RegraD2 (Fonte: autor).*

```
import java.util.Scanner;
public class Solucao
{
    private RegraC Vc[];
    private RegraD Vd[];
    private int nmov = 0;
    private int Nportos;
    private int Ncolunas;
    private int Nlinhas;
    private int T[][] = {{2, 5, 0, 0}, {0, 2, 3, 1}, {0, 0, 2, 2}, {0, 0, 0, 1}};
    private int M[][] = {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}};

    public  Solucao()..
    public Solucao(int A[][])..
    public void mostrarM() ..
    public void mostrarV(int nivel)..
    public String mostrarMelhoresRegras(int nivel)..
    public void mostrarMelhorVetor(int nivel, int aux[])..
    public void gerarSolucao()..
    public int avaliar(int nivel)..
    public void copiarSolucao(Solucao s)..
    public void preencherNivel(int nivel, RegraC rc, RegraD rd)..
}
```

FIGURA 9: *Classe Solucao (Fonte: autor).*

```
public class BeamSearch
{
    private int Nregras;
    private int Nportos;
    private Solucao vt[];
    private Solucao vp[];
    private int T[][] = {{2, 5, 0, 0}, {0, 2, 3, 1}, {0, 0, 2, 2}, {0, 0, 0, 1}};
    private int avaliacao[];
    private int beta;

    public BeamSearch() ..
    public void gerarSolucoes(int nivel)..
    public void avaliar(int nivel)..
    public void selecionar()..
    public String mostrar(int nivel)..
    public void mostrarDesenho(int nivel,int aux2[])..
    public void resolver()..
}
```

FIGURA 10: *Classe BeamSearch (Fonte: autor).*

3.6 - Resultados de Validação do Programa

Primeiramente será mostrada a matriz transporte utilizada para a validação do programa:

T[][] =

2	5	0	0
0	2	3	1
0	0	2	2
0	0	0	1

Esta matriz fornece a quantidade de contêineres com destino a determinado porto. Por exemplo, na primeira linha é indicado a existência de 2 contêineres com destino ao porto 2 e de 5 contêineres com destino ao porto 3;

Ou seja, de forma mais geral, o destino é a coluna em que este se encontra adicionada de duas unidades, lembrando que no programa em questão a primeira coluna foi considerada com índice zero.

Para o carregamento de forma otimizada do navio podemos utilizar qualquer combinação de regras e a melhor solução será encontrada comparando-se o número de movimentos necessários para realizá-la.

Inicialmente será fornecida a quantidade de movimentos necessários para cada combinação de regras possível. Como o programa possui quatro regras serão obtidas dezesseis combinações possíveis e estas, para o exemplo em questão, são mostradas na figura 11:

Obs: A figura 11 mostra o resultado obtido para o nível igual a quatro.

Vt(0): O valor do vetorC(0) eh: Regra C1 O valor do vetorD(0) eh: Regra D1 O valor do vetorC(1) eh: Regra C1 O valor do vetorD(1) eh: Regra D1 O valor do vetorC(2) eh: Regra C1 O valor do vetorD(2) eh: Regra D1 O valor do vetorC(3) eh: Regra C1 O valor do vetorD(3) eh: Regra D1 O valor do vetorC(4) eh: Regra C1 O valor do vetorD(4) eh: Regra D1 **nmov**=[40]	**Vt(1):** O valor do vetorC(0) eh: Regra C1 O valor do vetorD(0) eh: Regra D2 O valor do vetorC(1) eh: Regra C1 O valor do vetorD(1) eh: Regra D1 O valor do vetorC(2) eh: Regra C1 O valor do vetorD(2) eh: Regra D1 O valor do vetorC(3) eh: Regra C1 O valor do vetorD(3) eh: Regra D1 O valor do vetorC(4) eh: Regra C1 O valor do vetorD(4) eh: Regra D1 **nmov**=[40]
Regras e número de movimentos de vt(0).	Regras e número de movimentos de vt(1).
Vt(2): O valor do vetorC(0) eh: Regra C2 O valor do vetorD(0) eh: Regra D1 O valor do vetorC(1) eh: Regra C1 O valor do vetorD(1) eh: Regra D1 O valor do vetorC(2) eh: Regra C1 O valor do vetorD(2) eh: Regra D1 O valor do vetorC(3) eh: Regra C1 O valor do vetorD(3) eh: Regra D1 O valor do vetorC(4) eh: Regra C1 O valor do vetorD(4) eh: Regra D1 **nmov=[36]**	**Vt(3):** O valor do vetorC(0) eh: Regra C2 O valor do vetorD(0) eh: Regra D2 O valor do vetorC(1) eh: Regra C1 O valor do vetorD(1) eh: Regra D1 O valor do vetorC(2) eh: Regra C1 O valor do vetorD(2) eh: Regra D1 O valor do vetorC(3) eh: Regra C1 O valor do vetorD(3) eh: Regra D1 O valor do vetorC(4) eh: Regra C1 O valor do vetorD(4) eh: Regra D1 **nmov=[36]**
Regras e número de movimentos de vt(3).	Regras e número de movimentos de vt(4).
Vt(4): O valor do vetorC(0) eh: Regra C1 O valor do vetorD(0) eh: Regra D1 O valor do vetorC(1) eh: Regra C1 O valor do vetorD(1) eh: Regra D1 O valor do vetorC(2) eh: Regra C1 O valor do vetorD(2) eh: Regra D1 O valor do vetorC(3) eh: Regra C1 O valor do vetorD(3) eh: Regra D1 O valor do vetorC(4) eh: Regra C1 O valor do vetorD(4) eh: Regra D2 **nmov=[49]**	**Vt(5):** O valor do vetorC(0) eh: Regra C1 O valor do vetorD(0) eh: Regra D2 O valor do vetorC(1) eh: Regra C1 O valor do vetorD(1) eh: Regra D1 O valor do vetorC(2) eh: Regra C1 O valor do vetorD(2) eh: Regra D1 O valor do vetorC(3) eh: Regra C1 O valor do vetorD(3) eh: Regra D1 O valor do vetorC(4) eh: Regra C1 O valor do vetorD(4) eh: Regra D2 **nmov=[49]**
Regras e número de movimentos de vt(4).	Regras e número de movimentos de vt(5).

Vt(6): O valor do vetorC(0) eh: Regra C2 O valor do vetorD(0) eh: Regra D1 O valor do vetorC(1) eh: Regra C1 O valor do vetorD(1) eh: Regra D1 O valor do vetorC(2) eh: Regra C1 O valor do vetorD(2) eh: Regra D1 O valor do vetorC(3) eh: Regra C1 O valor do vetorD(3) eh: Regra D1 O valor do vetorC(4) eh: Regra C1 O valor do vetorD(4) eh: Regra D2 **nmov=[49]**	**Vt(7):** O valor do vetorC(0) eh: Regra C2 O valor do vetorD(0) eh: Regra D2 O valor do vetorC(1) eh: Regra C1 O valor do vetorD(1) eh: Regra D1 O valor do vetorC(2) eh: Regra C1 O valor do vetorD(2) eh: Regra D1 O valor do vetorC(3) eh: Regra C1 O valor do vetorD(3) eh: Regra D1 O valor do vetorC(4) eh: Regra C1 O valor do vetorD(4) eh: Regra D2 **nmov=[49]**
Regras e número de movimentos de vt(6).	Regras e número de movimentos de vt(7).
Vt(8): O valor do vetorC(0) eh: Regra C1 O valor do vetorD(0) eh: Regra D1 O valor do vetorC(1) eh: Regra C1 O valor do vetorD(1) eh: Regra D1 O valor do vetorC(2) eh: Regra C1 O valor do vetorD(2) eh: Regra D1 O valor do vetorC(3) eh: Regra C1 O valor do vetorD(3) eh: Regra D1 O valor do vetorC(4) eh: Regra C2 O valor do vetorD(4) eh: Regra D1 **nmov=[33]**	**Vt(9) :** O valor do vetorC(0) eh: Regra C1 O valor do vetorD(0) eh: Regra D2 O valor do vetorC(1) eh: Regra C1 O valor do vetorD(1) eh: Regra D1 O valor do vetorC(2) eh: Regra C1 O valor do vetorD(2) eh: Regra D1 O valor do vetorC(3) eh: Regra C1 O valor do vetorD(3) eh: Regra D1 O valor do vetorC(4) eh: Regra C2 O valor do vetorD(4) eh: Regra D1 **nmov=[33]**
Regras e número de movimentos de vt(8).	Regras e número de movimentos de vt(9).
Vt(10): O valor do vetorC(0) eh: Regra C2 O valor do vetorD(0) eh: Regra D1 O valor do vetorC(1) eh: Regra C1 O valor do vetorD(1) eh: Regra D1 O valor do vetorC(2) eh: Regra C1 O valor do vetorD(2) eh: Regra D1 O valor do vetorC(3) eh: Regra C1 O valor do vetorD(3) eh: Regra D1 O valor do vetorC(4) eh: Regra C2 O valor do vetorD(4) eh: Regra D1 **nmov=[29]**	**Vt(11):** O valor do vetorC(0) eh: Regra C2 O valor do vetorD(0) eh: Regra D2 O valor do vetorC(1) eh: Regra C1 O valor do vetorD(1) eh: Regra D1 O valor do vetorC(2) eh: Regra C1 O valor do vetorD(2) eh: Regra D1 O valor do vetorC(3) eh: Regra C1 O valor do vetorD(3) eh: Regra D1 O valor do vetorC(4) eh: Regra C2 O valor do vetorD(4) eh: Regra D1 **nmov=[29]**
Regras e número de movimentos de vt(10)	Regras e número de movimentos de vt(11)

Vt(12) : O valor do vetorC(0) eh: Regra C1 O valor do vetorD(0) eh: Regra D1 O valor do vetorC(1) eh: Regra C1 O valor do vetorD(1) eh: Regra D1 O valor do vetorC(2) eh: Regra C1 O valor do vetorD(2) eh: Regra D1 O valor do vetorC(3) eh: Regra C1 O valor do vetorD(3) eh: Regra D1 O valor do vetorC(4) eh: Regra C2 O valor do vetorD(4) eh: Regra D2 **nmov=[49]**	**Vt(13):** O valor do vetorC(0) eh: Regra C1 O valor do vetorD(0) eh: Regra D2 O valor do vetorC(1) eh: Regra C1 O valor do vetorD(1) eh: Regra D1 O valor do vetorC(2) eh: Regra C1 O valor do vetorD(2) eh: Regra D1 O valor do vetorC(3) eh: Regra C1 O valor do vetorD(3) eh: Regra D1 O valor do vetorC(4) eh: Regra C2 O valor do vetorD(4) eh: Regra D2 **nmov=[49]**
Regras e número de movimentos de vt(12).	Regras e número de movimentos de vt(13).
Vt(14): O valor do vetorC(0) eh: Regra C2 O valor do vetorD(0) eh: Regra D1 O valor do vetorC(1) eh: Regra C1 O valor do vetorD(1) eh: Regra D1 O valor do vetorC(2) eh: Regra C1 O valor do vetorD(2) eh: Regra D1 O valor do vetorC(3) eh: Regra C1 O valor do vetorD(3) eh: Regra D1 O valor do vetorC(4) eh: Regra C2 O valor do vetorD(4) eh: Regra D2 **nmov=[49]**	**Vt(15):** O valor do vetorC(0) eh: Regra C2 O valor do vetorD(0) eh: Regra D2 O valor do vetorC(1) eh: Regra C1 O valor do vetorD(1) eh: Regra D1 O valor do vetorC(2) eh: Regra C1 O valor do vetorD(2) eh: Regra D1 O valor do vetorC(3) eh: Regra C1 O valor do vetorD(3) eh: Regra D1 O valor do vetorC(4) eh: Regra C2 O valor do vetorD(4) eh: Regra D2 **nmov=[49]**
Regras e número de movimentos de vt(14).	Regras e número de movimentos de vt(15).

FIGURA 11: *Regras e número de movimentos de vt(i) (Fonte: autor).*

Assim o programa retorna todas as combinações possíveis, bem como o custo (em forma de número de movimentos) de cada ramo da árvore.

3.7 - Exemplo Ilustrativo de Construção da Árvore

Com o intuito de ilustrar a elaboração de uma árvore, foi criado um exemplo da resolução do problema para uma matriz transporte de pequenas dimensões e a consideração de apenas duas regras de carregamento (RC1 e RC2) e duas regras de descarregamento (RD1 e RD2). Para o exemplo em questão, a árvore resultante permite a verificação detalhada do processo de obtenção do número de movimento para cada solução. O número de movimentos

resultantes após a aplicação de cada regra é destacado por número dentro dos círculos.

A matriz transporte utilizada será dada por:

$$T = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix}$$

A árvore obtida será mostrada nas Figuras 12 e 13:

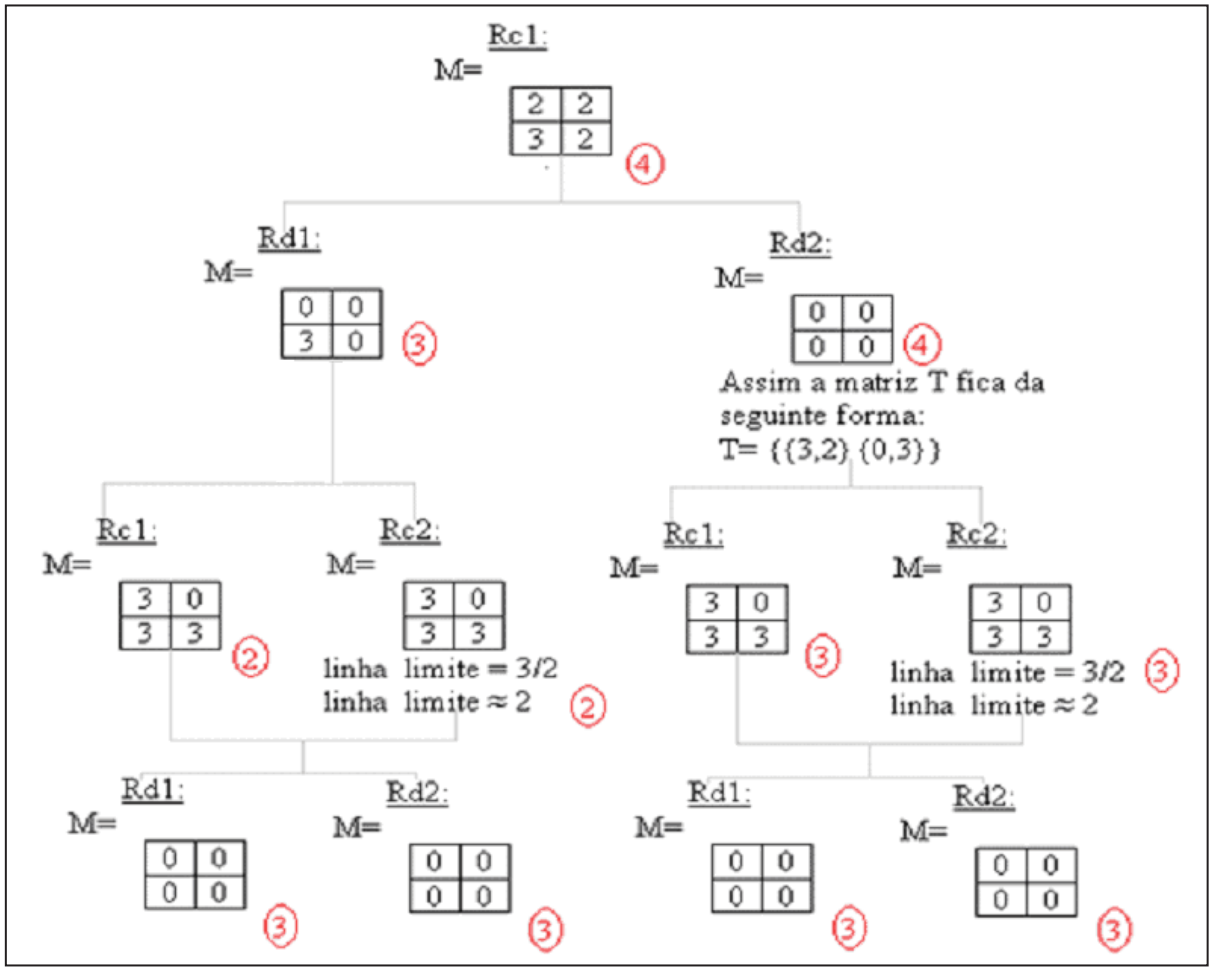

FIGURA 12: *Representação da resolução manual (I) (Fonte: autor).*

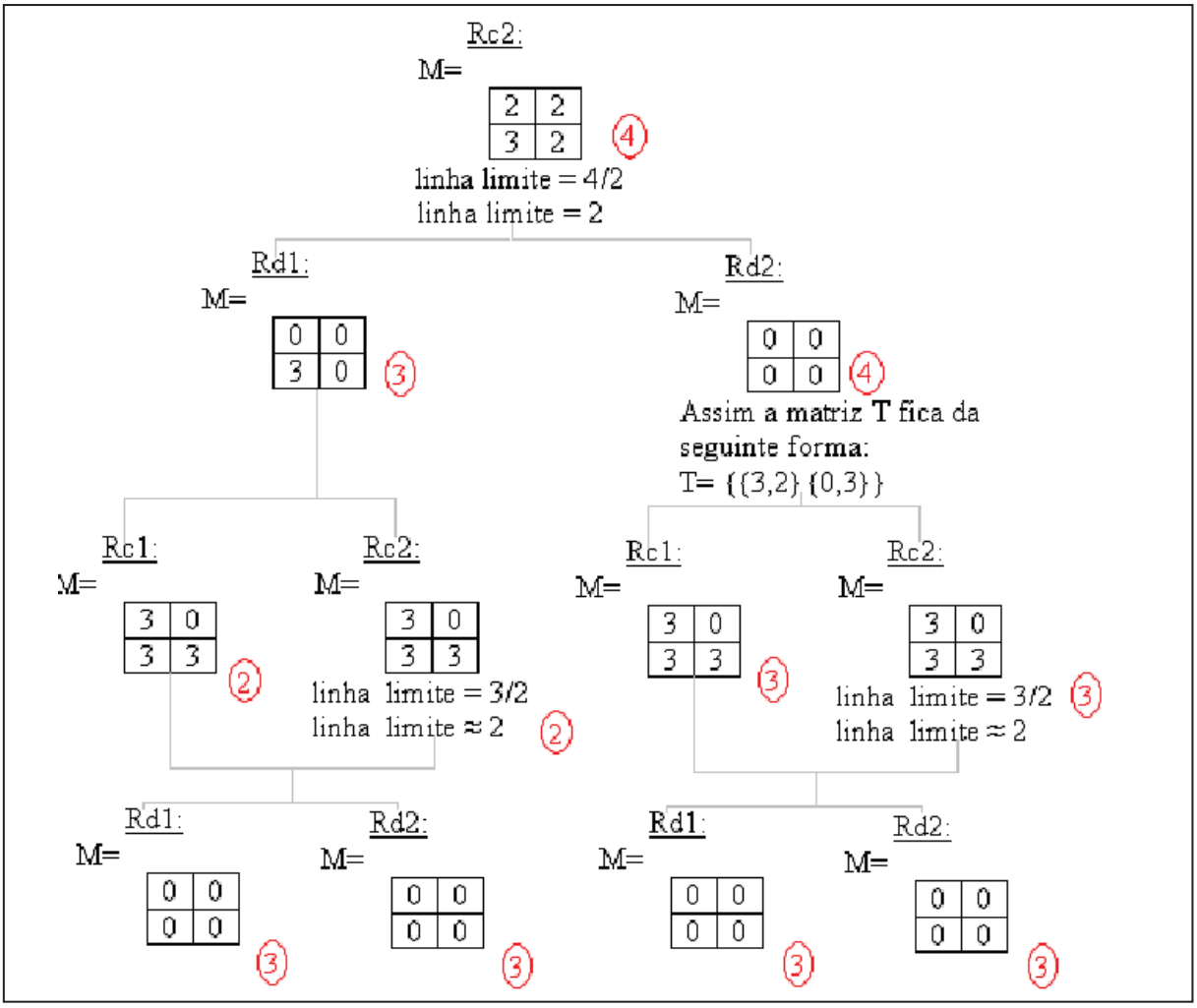

FIGURA 13: *Representação da resolução manual (II) (Fonte: autor).*

Este exemplo permitiu a validação dos resultados obtidos com o programa desenvolvido em Java, dado que exemplos com maior número de portos e regras resultam em árvore de difícil conferência visual.

4 – INTERFACE GRÁFICA

Para uma melhor interface entre o usuário e o programa, uma interface gráfica foi desenvolvida.

Através desta, é possível obter a melhor sequência de regras para obtermos carregamentos e descarregamentos com o menor número de remanejamentos possíveis e, consequentemente, com menor custo. A interface gráfica fornece, também, como o navio ficará ao sair de determinado porto definido pelo usuário, fornecendo assim uma maneira de conferir se os movimentos realizados foram corretos.

4.1 – Classes Desenvolvidas

```
import java.awt.Graphics;
import javax.swing.JPanel;
import java.awt.Color;

public class DesenhoNavio extends JPanel
{
  private int B[][];
  public DesenhoNavio()..
  public DesenhoNavio(int C[][])..
  public void setDesenhoNavio(int C[][])..
  public void paintComponent(Graphics g)..
}
```

FIGURA 14: *Classe DesenhoNavio (Fonte: autor).*

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import java.io.*;
import java.util.Scanner;

public class Ig extends JFrame implements ActionListener
{
    private final int WIDTH = 450;
    private final int HEIGHT = 300;
    private DesenhoNavio p3;
    private JPanel p1, p2, p4;
    private JLabel l1, l2, l3;
    private JButton b1, b2;
    private JTextField t1, t2, t3;

    public Ig()..
    public void createContents()..
    private JPanel createJPanel1()..
    private JPanel createJPanel2()..
    private DesenhoNavio createJPanel3()..
    private JPanel createJPanel4()..
    public void actionPerformed(ActionEvent e)..
}
```

FIGURA 15: *Classe Ig (Interface Gráfica) (Fonte: autor).*

4.2 - Descrição dos Métodos

4.2.1 – Classes Desenho do Navio

Método: DesenhoNavio.

Descrição: Não recebe nenhum parâmetro. Este método tem como função a inicialização da matriz a qual será desenhada.

Método: DesenhoNavio.

Descrição: Recebe como parâmetro a matriz ocupação *M*. Este método tem por função inicializar a matriz que será desenhada com a matriz *M*.

Método: setDesenhoNavio.

Descrição: Recebe como parâmetro a matriz ocupação *M*. Este método tem como função copiar a matriz *M* e redesenhar sua cópia na tela.

Método: paintComponent.

Descrição: Este método é responsável pelo desenho de cada posição da matriz, bem como pintar cada um com sua cor correspondente, visto que a cada destino foi atribuído uma cor diferente para a facilitação do entendimento.

4.2.2 – Classes Interface Gráfica (Ig)

Método: Ig.

Descrição: Não recebe nenhum parâmetro. Este método é responsável pela inicialização dos parâmetros referentes à janela, como por exemplo, as suas dimensões.

Método: createContents.

Descrição: Não recebe nenhum parâmetro. Este método é responsável pela criação dos *JPanel* necessários.

Método: createJPanel1.

Descrição: Não recebe nenhum parâmetro. Este método é responsável pela criação e localização dos componentes da *JPanel1*.

Método: createJPanel2.

Descrição: Não recebe nenhum parâmetro. Este método é responsável pela criação e localização dos componentes da *JPanel2*.

Método: createJPane3.

Descrição: Não recebe nenhum parâmetro. Este método é responsável pela criação e localização dos componentes da *JPanel3*.

Método: createJPane4.

Descrição: Não recebe nenhum parâmetro. Este método é responsável pela criação e localização dos componentes da *JPanel4*.

Método: actionPerformed.

Descrição: Este método é responsável pela definição da função de cada componente

existente na janela. Neste método ocorre também o tratamento de exceção.

4.3 – Apresentação da Interface Gráfica

A Figura 16 apresenta a interface gráfica desenvolvida.

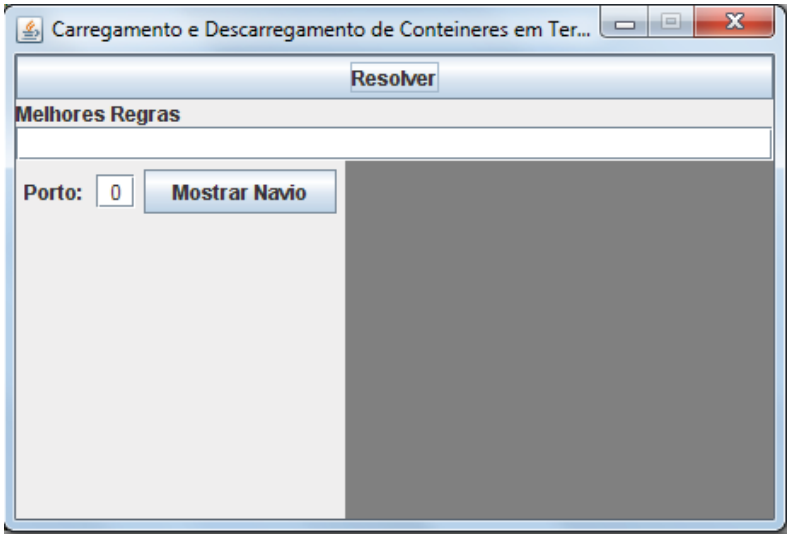

FIGURA 16: *Tela da Interface Gráfica (Fonte: autor).*

Cada componente da interface gráfica será explicado abaixo:

a) Botão "Resolver": Este botão tem por funcionalidade resolver o problema desejado, ou seja, ele realiza o *Beam Search*, compara as soluções encontradas e retorna as melhores soluções para serem aplicadas no decorrer do trajeto do navio.
b) Porto: Tem por finalidade permitir que o usuário digite até que nível (*porto*) as melhores regras serão aplicadas para que por fim o navio seja representado na tela.
c) Botão "Mostrar Navio": Desenha o navio carregado até o nível pré estabelecido pelo usuário no campo porto.

4.4 – Validação das resposta da interface gráfica

Para a validação das respostas da interface gráfica será utilizado o mesmo problema do item 3.6, assim o programa avaliará a melhor sequência de regras entre todas (figura 11) e a exibirá na tela. Para a exibição do navio após a saída do porto definido pelo usuário, será aplicada esta sequência de regras encontrada até o porto definido.

4.4.1 Melhor sequência de regras:

Neste item, será utilizada a interface gráfica desenvolvida para se obter a melhor sequência de regras para a resolução do problema cuja a matriz transporte é: T[][] = {{2, 5, 0, 0}, {0, 2, 3, 1}, {0, 0, 2, 2}, {0, 0, 0, 1}}.

Assim, como se trata do mesmo problema abordado no item 3.6 deste trabalho de graduação, será realizada a comparação entre as respostas obtidas.

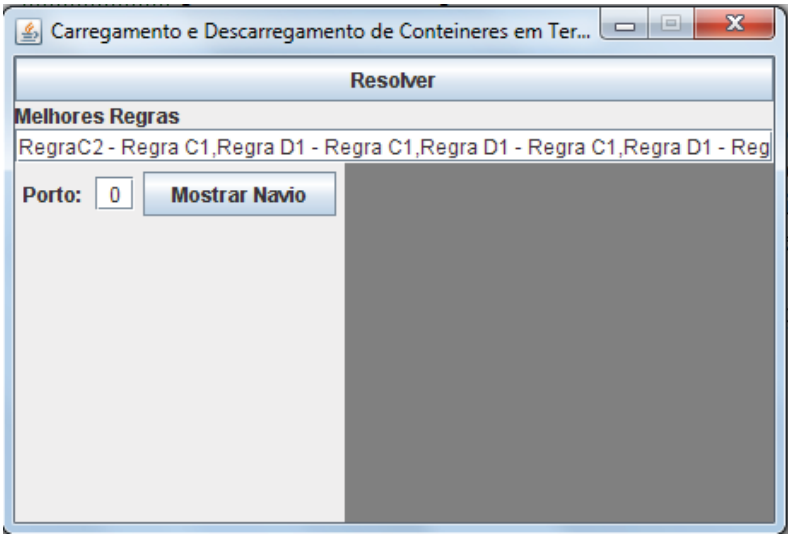

FIGURA 17: *Melhor sequência de regras – interface gráfica (Fonte: autor).*

A melhor sequência obtida foi: Regra C2 - Regra C1, Regra D1 - Regra C1, Regra D1 - Regra C1, Regra D1 - Regra C2, Regra D1. Esta resposta pode ser compara com os resultados Vt(10) da figura 11 que se trata da sequência com menor número de movimentos como desejado.

4.4.2 - Desenho do Navio após cada porto

FIGURA 18: *Navio após o porto 1 (Fonte: autor).*

FIGURA 19: *Navio após o porto 2 (Fonte: autor).*

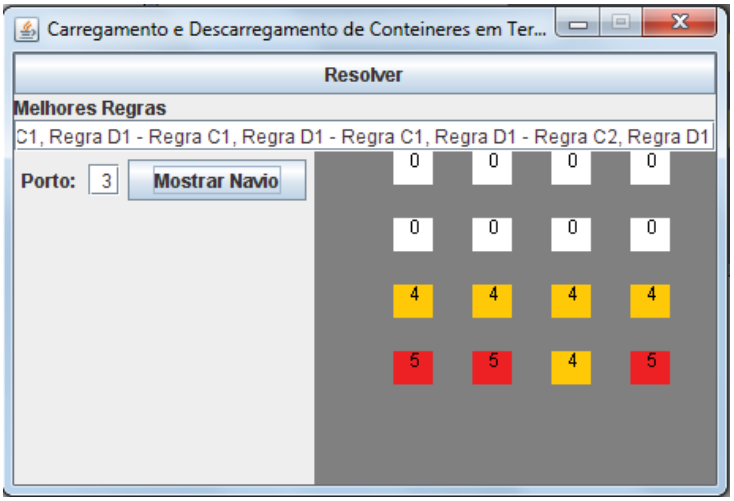

FIGURA 20: *Navio após o porto 3 (Fonte: autor).*

FIGURA 21: *Navio após o porto 4 (Fonte: autor).*

Com as Figuras 18 a 21 é possível observar e comprovar o funcionamento adequado do programa, visto que cada figura respeita as melhores regras obtidas.

4.5 -Tratamento de exceção

Em uma interface gráfica, deve ser realizada a previsão de casos em que a utilização desta não ocorra da forma programada e assim, estas exceções devem ser tratadas.

Na interface gráfica apresentada na Figura 17, é possível observar que o usuário deve inserir um número inteiro no campo Porto para que o desenho do navio ocorra de forma correta. Assim, caso o usuário não insira um número inteiro o programa pararia de funcionar, por este motivo o tratamento de exceção é necessário.

Abaixo, na figura 22, será mostrado o tratamento de exceção realizado.

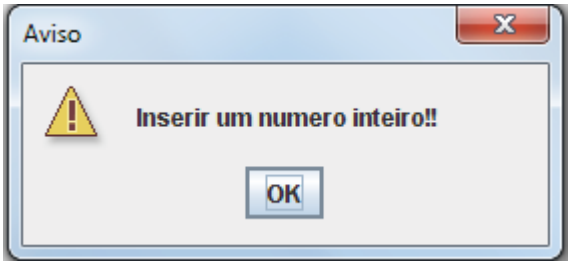

FIGURA 22: *Tratamento de exceção (Fonte: autor).*

Assim, caso o usuário insira uma letra, uma janela alertando a inserção de parâmetro inválido é exibida e o programa continua aguardando a entrada de um parâmetro válido.

5 - CONCLUSÃO

Neste trabalho inicialmente foram estudados os métodos implementados, que são a utilização de regras e o *Beam Search*. Após esta etapa, o programa foi desenvolvido na linguagem Java e testes para a sua validação foram realizados. Estes testes consistiram na simulação de casos conhecidos e casos em que o teste de mesa foi realizado. Para uma melhor interface com o usuário, uma interface gráfica foi desenvolvida.

Assim este trabalho de graduação sobre a resolução dos problemas de carregamento e descarregamento de contêineres em terminais portuários obteve um resultado bastante significativo conseguindo retornar uma solução que seja capaz de suprir o problema principal em questão que é redução dos custos e consequentemente do número de movimentos para o transporte dos contêineres em portos.

Ademais, o uso de representação por regras no programa desenvolvido apresenta três vantagens em relação à representação binária adotada na literatura Avriel (1998):

- A facilidade de incorporar conhecimento prévio do planejador sob a forma de regras;
- Todas as matrizes de ocupação produzidas pelas regras são factíveis. Isto facilita e garante que todas as soluções obtidas pelo método heurístico são factíveis. Neste trabalho a heurística utilizada é o *Beam Search*, mas futuramente serão utilizadas outras heurísticas para que se possa fazer uma comparação entre elas;
- A codificação da solução que determina como será realizado o carregamento e o descarregamento de um navio para N portos é um vetor de tamanho N-1. Esta representação é muito mais compacta se comparada com outras abordagens da literatura, como por exemplo, a utilizada em Avriel (1998).

REFERÊNCIAS BIBLIOGRÁFICAS

AMBROSINO, D., SCIOMACHEN A.,TANFANI, E.. **A decomposition heuristics for the container ship stowage problem, *J. Heuristics.*** 2006. v.12, p. 211–233.

AVRIEL, M.; PENN, M. **Container ship stowage problem, Computers and Industrial Engineering**. 1993. v. 25, p. 271- 274.

AVRIEL, M.; PENN, M.; SHPIRER, N.. **Container ship stowage problem: complexity and connection to the coloring of circle graphs, *Discrete Applied Mathematics.*** 2000. v.103, p. 271-279.

AVRIEL, M.; PENN, M.; SHPIRER, N.; WITTENBOON, S.. **Stowage planning for container ships to reduce the number of shifts, *Annals of Operations Research***. 1998. v. 76, p. 55-71.

BOTTER, R. C. ; BRINATI, M. A. . **Stowage Container Planing: a model for getting optimal solution**. 1992. p. 217-229.

DELLA CROCE, F.; T'KINDT, V.. **A Recovering Beam Search Algorithm for the One-Machine Dynamic Total Completion Time Scheduling Problem**. *Journal of the Operational Research Society*: 2002. vol 54, pp. 1275-1280.

FOX, M.S. **Constraint-Directed Search: A case Study of Job-Shop Scheduling.** *PhD. thesis*, Carnegie-Mellon University, USA: 1983.

LOWERRE, B. T.. **The HARPY Speech Recognition System.** PhD. Thesis, Carnegie-Mellon University, USA: 1976.

MATOS, A.V. **Unified Modeling Language UML – Prático e descomplicado.** 2ºedição. Érica, 2002.

OW, P.S, MORTON T.E.. **Filtered Beam Search in Scheduling**. 1988. vol. 26, pp. 35-62.

RIBEIRO, C.M.; AZEVEDO, A.T.. **Resolução do Problema de Carregamento e Descarregamento de Contêineres em Terminais Portuários via Beam Search.** XLII Simpósio Brasileiro de Pesquisa Operacional, Bento Gonçalves, RS, Brasil: 2010.

SABUNCUOGLU, I., BAVIZ, M.. **Job Shop Scheduling with Beam Search.** European Journal of Operational Research: 1999. vol. 118, pp. 390-412.

VALENTE, J. M. S; ALVES, R. A. F. S.. **Filtered and Recovering Beam Search Algorithm for the Early/Tardy Scheduling Problem with No Idle Time.** Computers & Industrial Engineering: 2005. vol. 48, pp. 363-375.

WILSON, I.; ROACH, P.. **Container stowage planning: a methodology for generating computerised solutions**. Journal of the Operational Research Society: 2000. v. 51, p. 1248-1255.

ARTIGOS PUBLICADOS

AZEVEDO, A.T.; RIBEIRO, C.M.; LIMA, F.M.B.. **Resolução do Problema de Carregamento e Descarregamento de Contêineres em Terminais Portuários via Beam Search.** *XVI Simpósio de Engenharia de Produção - SIMPEP – Bauru, SP*: 2009.

AZEVEDO, A.T.; RIBEIRO, C.M.; DEUS, N.M.R.; *LIMA, F.M.B.*. **Comparação de Heurísticas na Resolução do Problema de Carregamento e Descarregamento de Contêineres em Terminais Portuários via Representação por Regras.** Simpósio de Pesquisa Operacional da Marinha - SPOLM, Rio de Janeiro, RJ, Brasil: 2010.