Universidade Federal do ABC
Centro de Matemática, Computação e Cognição

Algoritmos para o Problema de Realocação de Contêineres

Santo André – SP
Novembro/2021

Matheus Fellone dos Santos

Algoritmos para o Problema de Realocação de Contêineres

Projeto de graduação em computação apresentado como requisito para a obtenção do título de Bacharel em Ciência da Computação pela Universidade Federal do ABC.

Orientadora: Profa. Dra. Carla Negri Lintzmayer

Santo André – SP
Novembro/2021

Resumo

Com o passar dos anos o transporte marítimo vem adquirindo cada vez mais relevância no comércio mundial, sendo que boa parte desse transporte é realizado através do uso de contêineres para armazenar as mercadorias. O armazenamento desses contêineres nos espaços limitados dos portos gera um problema logístico relacionado ao carregamento e descarregamento desses contêineres. Esse trabalho tratará especificamente do *problema da realocação de contêineres*. Nesse problema, dada uma configuração bidimensional contendo várias pilhas de contêineres, cada uma com uma capacidade de contêineres máxima, é desejado remover todos os contêineres desse espaço, respeitando uma ordem de prioridade dada previamente e realizando o menor número de realocações possíveis, que são movimentações de uma pilha para outra. Por ser um problema NP-Difícil, não se tem esperança de resolvê-lo em tempo polinomial, a menos que $P = NP$. Tendo isso em mente, nesse trabalho foram estudadas heurísticas, meta-heurísticas e algoritmos exatos, a fim de compreender o problema e enriquecer o conhecimento do aluno na área de Otimização Combinatória. Mais especificamente, foram estudados 5 métodos de resolução. Também foi feita a implementação de todos eles e a análise desses algoritmos, fazendo comparações entre os resultados obtidos, além de também comparar com os resultados já existentes na literatura.

Palavras-Chave: Otimização combinatória, realocação de blocos, realocação de contêineres, heurísticas.

Sumário

1 Introdução **5**

2 Definição do Problema **7**

3 Levantamento Bibliográfico **12**

3.1 Modelo de Programação Dinâmica 12

3.2 Médodo do Corredor 14

3.3 Branch and Bound 16

3.4 MinMax 20

3.5 Colônia de Formigas 23

4 Testes e Resultados **28**

4.1 Testes dos algoritmos exatos 28

4.2 Testes para o método colônia de formigas 30

4.3 Método Corredor 32

4.4 Comparação entre todos os algoritmos 34

4.5 Resumo dos resultados 34

5 Considerações Finais **37**

Referências **38**

1 Introdução

Nos dias de hoje o transporte marítimo é de extrema importância para o comércio internacional. Segundo *"The Review of Maritime Transport UNCTAD (2019)"* [17], relatório anual sobre o transporte marítimo, mais de 80% do comércio mundial de mercadorias em termos de volume é realizado nos portos de todo o mundo, sendo que os navios porta-contêineres em 2019 representavam cerca de 13.4% da frota mundial de navios.

Essa grande quantidade de contêineres somada à falta de espaço nos portos gera um problema logístico de armazenamento, de forma que os contêineres são empilhados para aumentar a utilização do espaço. Isso gera vários problemas relacionados ao carregamento e descarregamento desses contêineres, pelo fato de a ordem de chegada desses contêineres nos portos poder ser diferente da ordem em que eles precisam ser retirados. Assim, um dos problemas gerados é o problema de **Empilhamento de Blocos** [12]. Entre as abstrações criadas para resolver o problema podemos citar o problema de **Realocação de Blocos** (BRP, de *Blocks Relocation problem*), que será o foco desse trabalho, também conhecido como problema de Realocação de Contêineres. Outros problemas relacionados são o *Pre-Marshalling Problem* e o *Re-Marshalling Problem* [2, 14, 15].

Desde o surgimento do BRP, muita pesquisa já foi feita para tentar achar soluções viáveis para o problema. Caserta, Schwarze e Voß provaram que o BRP é NP-Difícil [4]. Por isso, a menos que $P = NP$, não podemos resolvê-lo em tempo polinomial. Entre os esforços para resolver o problema, temos estudos relacionados ao uso de diferentes *funções objetivos* [6, 13, 19] e conjuntos de restrições [8], além de estudos para o problema em três dimensões [13]. Outros estudos muito importantes para o assunto são os relacionados à versão *online* do problema [16, 18], que considera a chegada de contêineres em tempo real.

Este trabalho teve como objetivo o estudo do BRP em sua versão restrita, considerando algoritmos heurísticos e exatos para o mesmo. Nos aprofundamos em alguns dos métodos desenvolvidos na literatura para resolver o problema, e os implementamos de forma a comparar seus resultados. Vimos uma abordagem baseada em programação dinâmica, que tenta resolver o problema explorando todas as possibilidades de realocações, o método do corredor, que consiste em uma heurística aplicada em cima do método exato de programação dinâmica, a estratégia do *branch and bound*, uma heurística gulosa e a meta-heurística colônia de formigas. Com isso, o aluno foi exposto a diversas técnicas de solução de problemas comuns na área de Otimização Combinatória.

A partir daqui, o documento será divido da seguinte forma. Na Seção 2 será apresentada uma definição geral do problema, que servirá de base para todos os mé-

todos de resolução vistos nesse trabalho. Na Seção 3 são apresentadas, de forma mais elaborada e explicada, algumas metodologias utilizadas para resolver o problema do BRP, além de seus pseudocódigos quando necessário. Na Seção 4 mostraremos os resultados obtidos com os testes desses algoritmos em casos de testes relevantes para o problema. Por fim, a Seção 5 conclui o trabalho com as considerações finais.

2 Definição do Problema

Antes de partir para o levantamento bibliográfico e os resultados obtidos, vamos definir formalmente o problema do BRP restrito, de forma a criar uma notação única para os métodos de resolução estudados mais a frente. Se necessário serão feitas novas definições de notação em seções adiante, específicas para cada seção.

Consideramos então um **compartimento**, que vamos visualizar de forma bidimensional, que irá conter uma quantidade P de **pilhas de contêineres**, e uma **altura máxima** H para cada pilha. Também consideramos como n o número total de contêineres a serem retirados do espaço, sendo que cada contêiner é rotulado com um valor de 1 a n. Quanto menor o rótulo do contêiner, maior sua prioridade, ou seja, mais cedo ele deve ser retirado. Podemos considerar então um contêiner c, sendo $c \in \{1, \ldots, n\}$.

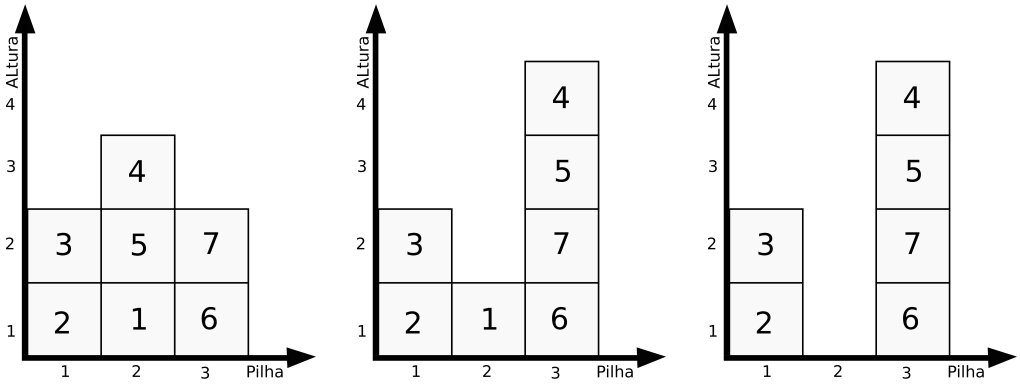
A posição de cada contêiner é conhecida antecipadamente, por isso podemos denotar por $p(c)$ a pilha em que o contêiner c está localizado, sendo $p(c) \in \{1, \ldots, P\}$, e por $h(c)$ a altura, ou nível, em que o contêiner c se encontra, com $h(c) \in \{1, \ldots, H\}$. Com isso definimos o **estado** do compartimento, que indica todas as posições dos contêineres.

Somente o contêiner que estiver no nível mais alto de uma pilha pode ser movido. Além disso, somente o contêiner com o menor valor pode ser removido do espaço onde os contêineres estão localizados. Chamamos também esse contêiner de **alvo**, e vamos designá-lo como T. Caso o alvo esteja bloqueado, o que significa que ele não está no topo da pilha $p(T)$, os contêineres acima dele, chamados de **bloqueantes**, devem ser transferidos para outras pilhas de forma a liberá-lo, sendo que esse contêineres só podem ser movidos para o topo de outras pilhas que tenham menos contêineres que a altura máxima H. Essa ação de mover um contêiner bloqueante de uma pilha para outra é o que chamamos de **realocação**. Caso T não tenha nenhum bloqueante dizemos que ele está **livre** e portanto pode ser removido do compartimento.

O objetivo do BRP é retirar todos os contêineres do compartimento, seguindo a ordem de prioridade, de forma a minimizar o número de realocações. Como dito anteriormente, nesse trabalho consideramos a versão restrita do BRP, onde as realocações permitidas devem ser realizadas somente com os contêineres bloqueantes de T.

Podemos representar uma solução viável do problema como uma sequência $((c_1, p_1), (c_2, p_2), \ldots, (c_x, p_x))$, onde c_i é o contêiner a ser realocado ou removido e p_i será a pilha para onde c_i será realocado, sendo que quando $p_i = 0$ significa que c_i está sendo retirado do compartimento. Note que em qualquer solução temos $x = n + r$, onde r é o número de realocações que ocorre, já que n remoções sempre vão existir.

Para entender melhor, podemos observar na Figura 1a um exemplo de um estado



(a) Configuração inicial. *(b) Realocação de 5 e 4.* *(c) Remoção de 1.*

Figura 1: Exemplo de um compartimento contendo 7 contêineres e com 2 realocações sendo realizadas.

inicial de uma instância do BRP, com $H = 4$, $P = 3$ e $n = 7$. Como definido, temos $T = 1$, ou seja ele deve ser o primeiro contêiner a ser retirado. Como podemos observar, 4 e 5 estão bloqueando-o, então devemos fazer uma realocação de cada um deles para retirar T. Por exemplo, se movermos 5 para a terceira pilha e logo em seguida o 4 também para a mesma pilha teremos um novo estado, mostrado na Figura 1b. Agora com esse novo estado, podemos retirar o contêiner 1 que é nosso alvo, resultando no estado representado na Figura 1c. Nesse novo estado agora atualizamos nosso alvo, $T = 2$. Podemos observar que para retirá-lo devemos mover o contêiner bloqueante 3. Nesse caso, pelas realocações anteriores a terceira pilha atingiu o limite máximo de altura $H = 4$, portanto só podemos mover o bloqueante do novo T para a segunda pilha e então remover T. Assim deve-se seguir até retirarmos todos os contêineres e gerarmos uma solução. Por exemplo, $((5, 3), (4, 3), (1, 0), (3, 2), (2, 0), (3, 0), (4, 0), (5, 0), (7, 2), (6, 0), (7, 0))$ é uma solução para o exemplo dado. Assim tivemos um total de 4 realocações para retirar todos os contêineres. Nesse exemplo não é possível limpar o compartimento com menos realocações, e por isso essa solução é ótima.

Como dito anteriormente, em Caserta, Schwarze e Voß foi provado que o BRP é NP-Difícil [4]. Eles fizeram isso reduzindo o problema conhecido como *Mutual Exclusion Scheduling (MES)*, provado ser NP-completo por Jansen [10] quando usado em grafos de permutação.

O problema MES surge quando temos um número n de tarefas que devem ser completadas por m processadores em um tempo t. Cada processador executa uma tarefa por vez e cada tarefa tem um tempo de execução. Além disso, exitem tarefas que não podem ser executadas ao mesmo tempo, sendo chamadas de concorrentes. O objetivo é encontrar uma forma de executar todas as tarefas dentro do tempo limite t. Para modelar esse problema pode-se usar um grafo G não direcionado, onde cada vértice é uma tarefa e as arestas conectam duas tarefas que são concorrentes.

Para entendermos melhor, primeiro vamos definir alguns conceitos que são utilizados na prova. Dado um grafo não direcionado G, um subconjunto $U \subseteq V(G)$ é chamado independente se não existe nenhum par de vértices $i, j \in U$ tal que $\{i, j\} \in E(G)$. O MES é definido formalmente da seguinte forma: Dado um grafo G e constantes m e t, existe uma partição de $V(G)$ em t subconjuntos independentes U_i, $i = \{1, \ldots, t\}$, tal que $|U_i| \leq m$ para todo i?

Jansen provou que o MES é NP-Completo para grafos de permutação. Dada uma permutação $\pi = (\pi_1, \ldots, \pi_z)$ dos números $1, \ldots, z$, um grafo de permutação tem como conjunto de vértices $V = \{1, \ldots, z\}$ e existe uma aresta $\{i, j\}$ para todo $i < j$ tal que j apareça antes de i na permutação. Por exemplo, para a permutação $\pi = (4, 3, 5, 1, 2)$ teríamos o grafo mostrado na Figura 2.

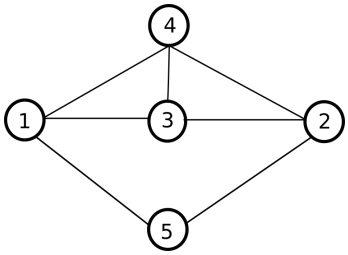


Figura 2: Grafo da permutação $\pi = (4, 3, 5, 1, 2)$.

O BRP foi provado então ser NP-Difícil reduzindo a versão de decisão do MES em grafos de permutação para a versão de decisão do BRP, que é definida da seguinte forma. Dada uma área de empilhamento com P, H, e n finitos e um inteiro k, existe uma solução para o BRP que exija exatamente k realocações? Se considerarmos uma instância do MES, com um grafo G e constantes m e t, onde G é o grafo da permutação $\pi = (\pi_1, \ldots, \pi_z)$ vamos criar uma instância do BRP de decisão da seguinte forma: definimos o número de pilhas como sendo $P = t + 1$, a altura como sendo $H = z + 1$, o número de contêineres $n = PH - tm$, o inteiro $k = z$ e a configuração inicial como mostrado na Tabela 1.

Tabela 1: Regra para criar a configuração inicial do compartimento a partir de um grafo de permutação.

Pilha	Altura	Prioridade do Bloco
$p = 1$	$h = 1$	1
$p = 1$	$h = 2, \ldots, H$	$\pi_{h-1} + 1$
$p = 2, \ldots, P$	$h = 1, \ldots, H - m$	$N + 1 - (h + (i - 2)(H - m))$
$p = 2, \ldots, P$	$h = H - m + 1, \ldots, H$	vazio

Assim, dada a mesma permutação vista anteriormente, $\pi = (4, 3, 5, 1, 2)$, com $t = 3$ e $m = 2$, seguindo a Tabela 1 teríamos como configuração inicial do compartimento o estado mostrado na Figura 3, onde vemos, na pilha 1, que os contêineres acima do alvo $T = 1$ têm os valores equivalentes aos da permutação π somando 1 em todos

os números e o restante das pilhas têm os contêineres empilhados de forma que não há bloqueantes em nenhum deles.

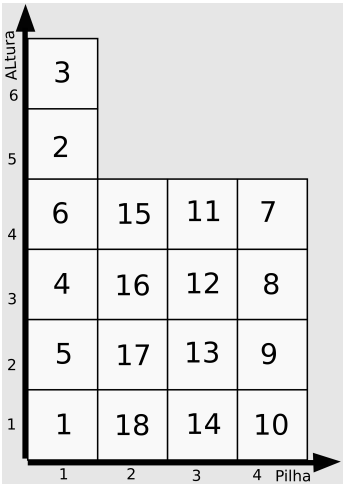


Figura 3: Compartimento criado a partir do grafo de permutação da Figura 2 e considerando $t = 3$ e $m = 2$

Agora resta provar a seguinte afirmação: uma solução do BRP com exatamente k realocações existe se e somente se $V(G)$ pode ser particionado em $t = P - 1$ conjuntos independentes de tamanho no máximo m.

Para o exemplo da Figura 3 é fácil perceber que precisamos de pelo menos $k = z = 5$ realocações para conseguir remover o contêiner 1 do compartimento. Além disso, como visto antes, os contêineres acima do 1 são semelhantes à permutação dada inicialmente. Para ter uma solução com exatamente z realocações, os blocos que forem realocados para as outras pilhas não devem gerar novas realocações. Para realocar dois contêineres para uma pilha de forma que eles não causem mais realocações, é preciso que eles façam parte de um conjunto independente de G. Como existem $P-1 = t$ pilhas restantes e m espaços vazios em cada uma, encontrar uma solução que não causa mais realocações é equivalente a encontrar $P - 1 = t$ conjuntos independentes de tamanho máximo m no grafo G.

Vamos então provar formalmente as seguintes afirmações:

- Se uma solução do BRP com $k = z$ realocações existe, então é possível particionar $V(G)$ em $t = P - 1$ conjuntos independentes de tamanho máximo m;
- Se é possível particionar $V(G)$ em $t = P - 1$ conjuntos independentes de tamanho máximo m, então existe uma solução do BRP com $k = z$ realocações.

Para a primeira afirmação, suponha que existe uma sequência $((c_1, p_1) \dots (c_{k+n}, p_{k+n}))$ que remove os contêineres do compartimento. Note que $(c_1, c_2, \dots, c_z) = (\pi_1 + 1, \pi_2 + 1, \dots, \pi_z + 1)$, pois o bloco 1 estará no final da primeira pilha, portanto todos os blocos referentes à permutação devem ser realocados inicialmente. Como na

solução ocorrem somente $k = z$ realocações, então os contêineres foram realocados para as outras $P - 1$ pilhas de forma que não causaram mais realocações. Note que, quando dois contêineres desse conjunto formado pela permutação não causam realocações entre si, quer dizer que em G eles fazem parte de um mesmo conjunto independente. Portanto, existem $t = P - 1$ conjuntos independentes em $V(G)$ com tamanho até m cada. Portanto a primeira afirmação está correta.

Para a segunda afirmação sejam $U_1, \ldots, U_t$, com $|U_i| \leq m$, conjuntos independentes de G. De forma equivalente, para existir uma solução com k realocações, precisamos realocar os contêineres equivalentes à permutação da primeira pilha para as outras t pilhas sem causar mais nenhuma realocação. Fazemos isso realocando de forma que em cada pilha fiquem os contêineres equivalentes a um U_i. Como temos t conjuntos independentes, cada um de tamanho máximo m, podemos realocar os contêineres paras as $t = P - 1$ pilhas restantes, cada uma com m espaços livres, sem causar mais realocações. Portanto existe uma solução com $k = z$ realocações, e logo a segunda afirmação também está correta.

Com isso temos a prova de que esse problema de decisão é NP-Completo, assim como provado por Jansen [10]. Portanto, se o MES é NP-Completo, temos que a versão de decisão do BRP também é NP-Completo, e logo a versão de otimização do BRP é NP-Difícil.

3 Levantamento Bibliográfico

Nessa seção vamos mostrar alguns algoritmos para o BRP restrito. Vamos ver algoritmos exatos nas Seções 3.1 e 3.3, que portanto sempre resultam na solução ótima, mas nem sempre com um desempenho viável dependendo do tamanho da instância. Veremos também heurísticas nas Seções 3.2, 3.4 e 3.5, que apesar de nem sempre obterem uma solução ótima, conseguem achar soluções viáveis com um desempenho aceitável.

3.1 Modelo de Programação Dinâmica

O primeiro trabalho que vamos abordar será o do artigo *"Applying the corridor method to a blocks relocation problem"*, de Caserta, Voß e Sniedovich [3], onde os autores definem um modelo de programação dinâmica para realizar a busca no espaço de possíveis soluções do problema e aplicam uma heurística para diminuir o espaço de busca desse modelo.

Inicialmente definimos um esquema de programação dinâmica que é capaz de capturar todos os estados possíveis do compartimento de contêineres. Primeiro definimos dois possíveis casos de um estado, sendo um onde o contêiner alvo T está livre, ou seja, não há bloqueantes e T pode ser removido, e o outro quando temos pelo menos um contêiner bloqueando T no estado atual. Nesse último caso, todas as possíveis realocações do bloqueante ao topo da pilha $p(T)$ devem ser enumeradas. Esse processo de enumeração dos estados é repetido até o contêiner alvo estar livre e poder ser retirado. Após isso, o processo é repetido para o próximo contêiner a ser retirado, até o compartimento estar inteiramente vazio.

Os autores também definem alguns elementos básicos para o modelo de programação dinâmica apresentado:

- Uma variável de estado, representada por uma 4-upla $s = (T, p(T), B, C)$, onde $T \in \{1, \ldots, n\}$ é o alvo, $p(T) \in \{1, \ldots, P\}$ é a pilha em que T está localizado, B é a sequência de contêineres bloqueantes do alvo (com o topo à direita) e C é uma sequência de sequências, onde cada sequência interna corresponde a uma pilha do estado, excluindo $p(T)$. Assim, se C_i é a sequência de elementos da pilha i, com $i \in \{1, \ldots, P\}$, então $C = (C_1, C_2, \ldots, C_{p(T)-1}, C_{p(T)+1}, \ldots, C_P)$. Por exemplo, na Figura 1a, $T = 1$, $p(T) = 2$, $B = (4, 5)$, $C_1 = (2, 3)$, $C_3 = (6, 7)$ e $C = ((2, 3), (6, 7))$.

- Uma variável de decisão, para o caso do contêiner T estar bloqueado no estado s. Considerando τ como o contêiner no topo de $p(T)$, definimos $D(s)$ como o conjunto que contém todas as possíveis pilhas para onde τ pode ser re-

alocado. Assim, cada decisão possível a partir do estado s atual é representada por um $x \in D(s)$. Por exemplo, na Figura 1a temos $D(s) = \{1, 3\}$.

- Seja $s' = (T', p(T'), B', C')$ o estado que é obtido após aplicar a decisão x no estado $s = (T, p(T), B, C)$. Definimos $F(s, x)$ como a função de transição de estado, em que $s' = F(s, x)$. Se T está livre em s, temos que $T' = T + 1$, $p(T')$ é a pilha em que T' está, B' é a lista de bloqueantes de T' e C' é a configuração dos contêineres restantes. Caso T esteja bloqueado em s, temos $T' = T$, $p(T') = p(T)$, $B' = B \backslash \{\tau\}$, onde τ é o contêiner a ser realocado, e C' vai ser definido dependendo para onde τ foi realocado. Por exemplo, na Figura 1a, temos $\tau = 5$ e ao movermos ele para a pilha 3 como mostrado na Figura 1b, teremos $s' = (1, 2, \{4\}, C')$, com $C' = ((2, 3), (6, 7, 5))$.

- A função objetivo é dada por $f(s) = 1 + \min_{x \in D(s)}\{f(F(s, x))\}$, onde $s = (T, p(T), B, C)$, e ela indica que o menor número de realocações de s, denotado por $f(s)$, é 1, para a remoção do alvo, somado com o menor número de realocações do estado que se atinge ao realocar um bloqueante, ou remover o alvo. Caso $T = n$, ou seja, temos somente um contêiner no compartimento para remover, temos que $f(s) = 1$. Considerando os casos explicados antes podemos definir a fórmula mais formalmente como:

$$f(T, p(T), B, C) = \begin{cases} 1 + f(T + 1, p(T + 1), B', C) & \text{se } B = \emptyset \\ 1 + \min\limits_{x \in D(T, p(T), B, C)} \{f(T, p(T), B \setminus \{\tau\}, C')\} & \text{se } B \neq \emptyset \\ 1 & \text{se } T = n \end{cases}$$

Assim podemos definir o caso base do problema: para um compartimento com somente um contêiner, temos de forma trivial que vale $f(T, p(T), \emptyset, C) = 1$.

É fácil perceber pela fórmula que, ao implementar essa solução, nós percorreríamos todos os estados possíveis de uma configuração inicial, ou seja, ela iria funcionar de forma ótima. Porém em instâncias de grande escala, esse algoritmo é inviável.

Por isso Caserta, Voß e Sniedovich [3] apresentam uma meta-heurística que, aplicada ao algoritmo, diminui o espaço de buscas dos possíveis estados a partir um estado inicial, e ainda consegue encontrar soluções viáveis. O modelo utilizado é o método do corredor, ou *Corridor Method*, que será apresentado na próxima seção, onde serão definidas variáveis que vão limitar as possíveis decisões de realocações a partir de um estado.

3.2 Médodo do Corredor

Ainda em Caserta, Voß e Sniedovich [3] utilizando o mesmo modelo apresentado na subseção anterior como base, os autores aplicam uma heurística, o método do corredor, para conseguir minimizar os problemas de desempenho apresentados pela solução de programação dinâmica.

Dado um estado $s = (T, p(T), B, C)$, podemos definir valores que vão limitar as possibilidades de realocação dos contêineres bloqueantes no compartimento, tanto horizontalmente, quanto verticalmente. Definimos então uma constante δ, que vai limitar horizontalmente as possibilidades de forma que um bloqueante só pode ser realocado para alguma pilha entre $p(T) - \delta$ e $p(T) + \delta$. Por exemplo, considerando a configuração da Figura 4, se definirmos $\delta = 1$, como $T = 1$ está localizado na pilha 3, o algoritmo só vai gerar estados de realocações para as pilhas 2 e 4. No caso do exemplo, onde temos 5 pilhas, ao invés de gerar quatro diferente configurações, somente duas seriam geradas. Além disso, também é definida outra variável λ, que impõe um corredor vertical no compartimento que limita a altura das pilhas para as quais o contêiner bloqueante pode ser realocado para um valor menor que λ. Por exemplo, na Figura 4, se $\lambda = 3$, então o algoritmo irá impedir de gerar o estado que realocaria 5 para a pilha 1, pois essa já contém 3 contêineres. Assim, somente 3 dos 4 estados possíveis seriam gerados.

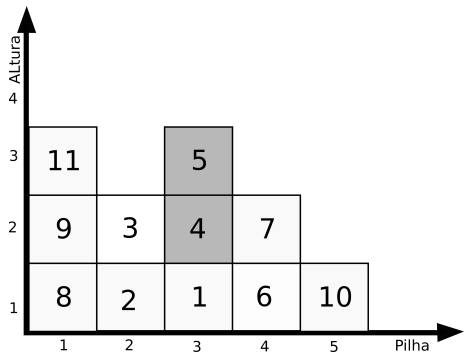


Figura 4: Compartimento contendo 11 contêineres, 5 pilhas e altura máxima 4.

Os autores então usam essas duas variáveis para gerar um corredor bidimensional que vai restringir o conjunto de estados possíveis a serem gerados a partir de $s = (T, p(T), B, C)$ em uma execução no modelo de programação dinâmica definido anteriormente. Também vale ressaltar que para ser possível gerar soluções viáveis é necessário que o corredor definido tenha pelo menos $|B|$ espaços vazios para poder realocar os contêineres bloqueantes. Caso isso não aconteça, o algoritmo irá aleatoriamente incrementar λ ou δ até que haja espaços vazios suficientes dentro do corredor.

Definindo formalmente, seja $s = (T, p(T), B, C)$ o estado atual do compartimento, que contém P pilhas, e para cada pilha $p \in \{1, \ldots, P\}$ seja C_p a sequência

de contêineres que estão em p. Por exemplo, na Figura 4 temos $C_1 = (8, 9, 11)$. Portanto $C = (C_1, \ldots, C_{p(T)-1}, C_{p(T)+1}, \ldots, C_P)$. Considerando $|C_p|$ como o número de contêineres de uma pilha p e dados δ e λ, definimos um novo conjunto de decisões $D(s)$ para o problema: $D(s, \delta, \lambda) = \{x \in \{1, ..., P\} \setminus \{p(T)\} : p(T) - \delta \leq x \leq p(T) + \delta, |C_x| < \lambda\}$.

Obviamente essa estratégia não é ótima para a resolução do problema, pois o corredor limita os estados percorridos pelo algoritmo, fazendo com que seja possível que realocações que criariam uma solução ótima não ocorram, pois a pilha que seria o destino dos bloqueantes estaria fora dos limites determinados pelo corredor.

3.3 Branch and Bound

Como visto anteriormente, o BRP foi definido por Kim e Hong [12], e nesse mesmo artigo os autores apresentam um modelo baseado na estratégia *branch and bound* (B&B) para poder resolver de forma ótima o problema. Então nessa seção vamos apresentar a definição do método.

Um algoritmo de B&B é caracterizado por criar uma árvore de enumeração que explora as possíveis soluções do problema, onde cada nó da árvore representa um estado do problema e o nó raiz representa o estado inicial. O algoritmo utiliza de estratégias para realizar cortes nas expansões desses nós, com o objetivo de descartar a exploração de ramos que não vão gerar soluções ótimas. Essa estratégia é utilizada no BRP para cortar os ramos em que as soluções certamente não conterão a solução com o menor número de realocações. Vale ressaltar que na versão dos autores, cada nó da árvore de busca é definido de uma forma diferente, representando um estado após todos os bloqueantes do alvo atual serem realocados.

Denote por s^k o estado depois de k blocos serem retirados do compartimento, por a^k uma sequência de realocações realizadas para retirar o contêiner k do compartimento, ou seja, o estado s^k é resultante da aplicação das ações a^k em s^{k-1} e denotamos isso por $s^{k-1} \rightarrow^{a^k} s^k$. Também denotamos por $r(a^k|s^{k-1})$ o número de realocações realizadas em a^k a partir do estado s^{k-1} e, por fim, $F(s^k)$ como o número total mínimo de realocações para coletar os $n-k$ contêineres restantes do compartimento do estado s^k. Para facilitar a representação dos estados vamos usar a mesma representação utilizada anteriormente. Por exemplo, o estado representado na Figura 5a seria definido como $s^0 = (1, 2, (6, 3), C)$, com $C = ((4), (5, 2))$. Note que cada s^i será a representação de um estado contido em cada nó da árvore de busca.

O problema então pode ser escrito como encontrar $F(s^0)$, que será igual à quantidade de realocações de uma ação a^1 tal que $r(a^1|s^0)$ possua o menor valor possível, somado com $F(s^1)$, que é a menor quantidade de realocações para retirar os $n-1$ contêineres restantes. Portanto temos $F(s^0) = \min_{a^1}\{r(a^1|s^0) + F(s^1)\}$, onde s^1 é o resultado de $s^0 \rightarrow^{a^k} s^1$. Ao expandirmos a fórmula, se considerarmos um compartimento com k contêineres, teremos

$$F(s^0) = \min_{a^1, a^2, \ldots, a^k} \left\{ \sum_{i=1}^{k} r(a^i|s^{i-1}) + F(s^k) \right\},$$

onde $s^{i-1} \rightarrow^{a^i} s^i$ para $i = 1, 2, \ldots, k$.

Dado um estado s^i qualquer, definimos como **realocações confirmadas** as realocações que com certeza vão ocorrer a partir desse s^i, que são todos os contêineres que estão acima de um contêiner de prioridade maior. Com isso conseguimos definir

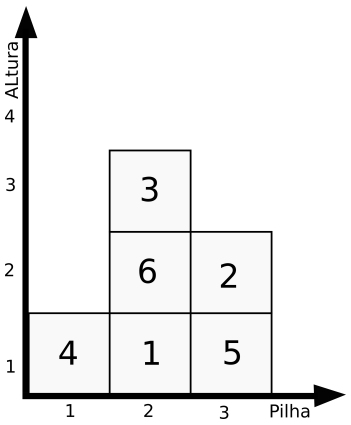
o limite inferior de realocações para s^i, que será a soma do número de realocações que ocorreram até então com o número de realocações confirmadas. Se na árvore de busca o limite inferior do nó atual s^i for maior do que o a melhor solução encontrada, então esse nó para de ser explorado (*bound*). Esses limites são usados na ramificação (*branch*) da árvore, onde serão explorados primeiro os nós filhos que têm o menor limite inferior.

Para ilustrar o funcionamento do algoritmo, vamos considerar o estado inicial s^0 definido pela Figura 5a. Os nós que podem ser gerados a partir desse estado inicial são representados nas Figuras 5b, 5c e 5d. Podemos ver pelos contêineres destacados que, nos estados das Figuras 5b e 5c, após retirar o contêiner 1, teremos pelo menos uma realocação confirmada, que ocorrerá em algum momento posterior, já que o contêiner destacado em cada imagem está bloqueando outros mais prioritários. Já no estado da Figura 5d há duas realocações confirmadas. Levando isso em consideração, ordenamos os nós a serem explorados pelo limite inferior de forma crescente, e iniciamos a busca pelo menor. Como podemos ver nas Figuras 5b e 5c, tivemos duas realocações a partir do estado inicial (dos contêineres 3 e 6) e temos uma realocação confirmada em ambos, portanto seus limites inferiores serão iguais a 3. Já na Figura 5d, temos duas realocações confirmadas a partir do estado inicial, mais as duas realocações já realizadas, portanto o limite inferior será 4. Sendo assim, nesse modelo, os ramos dos estados nas Figuras 5b e 5c serão explorados primeiro, encontrando assim uma solução viável e possivelmente atualizando a melhor solução, enquanto o da Figura 5d será explorado posteriormente, e por ter um limite inferior maior tem uma chance maior de ter a exploração interrompida antes de chegar ao final da árvore de exploração.

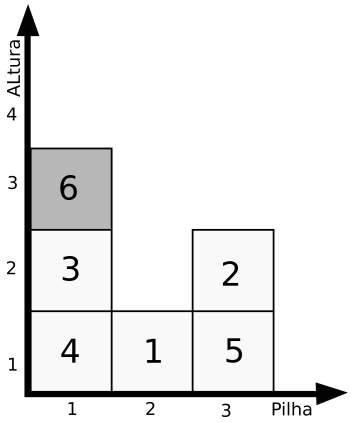
Para entendermos melhor o funcionamento do *branch and bound*, o Algoritmo 1 apresenta o pseudocódigo criado para servir de base para a implementação do código. Esse algoritmo usa as funções REALOCACOESCONFIRMADAS, que devolve o número de realocações confirmadas de um dado estado s, QTDBLOQUEANTES, que devolve a quantidade de bloqueantes do alvo em um dado estado s, e GERARFILHOS, que cria a lista dos próximos estados, que correspondem aos nós da árvore de busca.

A função BRANCHANDBOUND recebe três parâmetros. O primeiro, S, representa o estado atual. O valor *realocs* representa o número de realocações realizadas até aquele estado. E por fim *melhorSol* representa o número de realocações da melhor solução encontrada até então. No algoritmo, primeiro verificamos se o estado está vazio e, caso esteja, significa que temos uma solução completa. Por isso, verificamos se o número realocações *realocs* realizadas até ali é menor que *melhorSol* pois, caso seja, *melhorSol* deve ser atualizada com o valor de *realocs*.

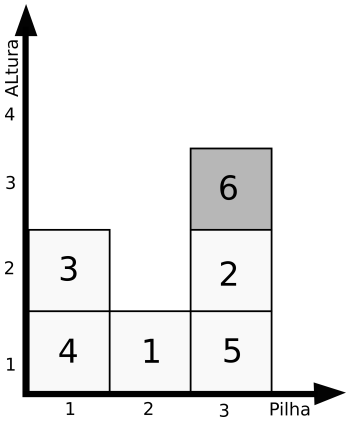
Caso o compartimento ainda não esteja vazio, primeiro verificamos se o limite inferior é maior que a melhor solução encontrada até então. Se for, a exploração



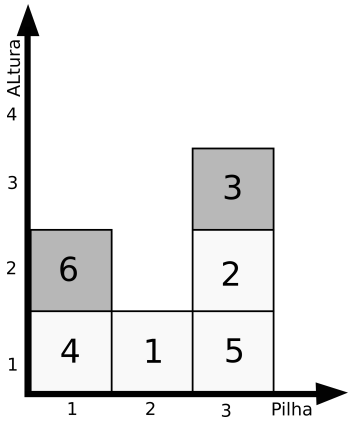
(a) Estado inicial contendo 6 contêineres, com 3 pilhas e altura máxima 3.



(b) Realocação de 3 e 6 para a pilha 1. Limite inferior igual a 3.



(c) Realocação de 3 para a pilha 1 e 6 para a pilha 3. Limite inferior igual a 3.



(d) Realocação de 3 para a pilha 3 e 6 para a pilha 1. Limite inferior igual a 4.

Figura 5: Estado inicial e possíveis estados gerados para poder retirar o bloco 1.

dessa ramificação é interrompida. Caso contrário, o valor de *realocs* é atualizado com as realocações dos bloqueantes que vão ocorrer para gerar os próximos estados, e então essa lista de estados filhos, que são os possíveis estados gerados ao realocar os bloqueantes do alvo, é gerada e a função BranchAndBound é chamada recursivamente para os estados gerados na ordem do limite inferior dos novos estados.

A função gerarFilhos gera os estados filhos de um dado estado e os insere na lista *proximos_estados*. Enquanto o alvo T não está livre, ela realiza as realocações possíveis, e quando encontra um estado em que T está livre ela calcula o número de realocações confirmadas desse estado e o insere na lista *proximos_estados*. Ela faz isso até encontrar todos os estados filhos possíveis. O pseudocódigo dela é apresentado no Algoritmo 2.

Algoritmo 1 Pseudocódigo do procedimento *branch and bound*, que recebe estado atual, número de realocações até o momento, e valor da melhor solução encontrada até o momento.

```
1:  Função BranchAndBound(s, realocs, melhorSol)
2:      seja proximos_estados uma variável global
3:      Se compartimento está vazio então
4:          Se realocs < melhorSol então
5:              melhorSol = realocs
6:          Devolve melhorSol
7:      limiteInferior = realocs + realocacoesConfirmadas(s)
8:      Se limiteInferior ≥ melhorSol então
9:          Devolve melhorSol
10:     realocs = realocs + qtdBloqueantes(S)
11:     gerarFilhos(s)
12:     ordena proximos_estados pelo número de realocações confirmadas
13:     Para cada S' ∈ proximos_estados seguindo a ordenação faça
14:         melhorSol = BranchAndBound(S', realocs, melhorSol)
15:     esvazia proximos_estados
16:     Devolve melhorSol
```

Algoritmo 2 Função que gera todos estados que são filhos de um dado estado.

```
1:  Função gerarFilhos(S)
2:      Se alvo T está livre então
3:          r = realocacoesConfirmadas(S)
4:          adiciona estado S na lista proximos_estados junto ao valor r
5:      Senão
6:          Para cada p' ∈ {1, ..., P} com p' ≠ p(T) faça
7:              Se pilha p' não está cheia então
8:                  atualiza S realocando o primeiro bloqueante c para p'
9:                  gerarFilhos(S)
10:                 atualiza S devolvendo c para a posição original
```

Para a primeira chamada de BranchAndBound, o valor enviado como $realocs$ é definido como zero e o valor inicial de $melhorSol$ é o valor de uma solução viável qualquer. No caso, utilizamos o valor de realocação de uma solução encontrada pelo MinMax, que é o método guloso apresentado na próxima seção.

3.4 MinMax

Dentre as resoluções que existem na literatura para o BRP, existem vários algoritmos simples baseados em heurísticas que conseguem achar uma solução viável. Muitas vezes esses algoritmos são usados para encontrar soluções iniciais para aplicar um outro método, ou para para definir limites inferiores, por exemplo. Dentre essas heurísticas, uma muito utilizada é a que vamos chamar de MinMax, que formalmente foi definida pela primeira vez por Caserta, Schwarzeb e Voß [2], e é uma heurística gulosa.

Essa heurística é baseada na definição de uma pontuação para cada pilha, de forma a definir qual a melhor pilha para realocar um contêiner bloqueante. Para determinar onde cada contêiner deve ser realocado, os autores definem um conceito de atratividade da pilha. Então é definido o valor $min(p)$, que é o valor do contêiner de maior prioridade em uma pilha p (portanto o menor valor), sendo $p \in \{1, \ldots, P\}$. Para pilhas vazias considera-se $min(p) = n + 1$.

Tendo esses valores definidos, consideramos c, o primeiro contêiner a ser retirado da lista de bloqueantes do alvo, e escolhe-se a pilha para qual ele vai ser realocado da seguinte forma. Se existir pelo menos uma pilha p em que $min(p) > c$, então c será realocado para a pilha com o menor $min(p)$ que satisfazer essa condição. Sendo assim, o contêiner estaria bem localizado e não haveria realocação extra para retirá-lo. Caso não exista nenhuma pilha que satisfaça essa condição, deve ser selecionada então a pilha que tenha o maior $min(p)$ tal que $min(p) < c$. Sendo assim, o contêiner c causará mais realocações forçadas mas, maximizando $min(p)$, tenta-se minimizar essas realocações.

Como dito antes, muitas dessas heurísticas como o MinMax são usadas como soluções iniciais ou limites para outras resoluções, como é o caso da resolução com o método de colônia de formigas [11], que veremos na próxima seção, e do *branch and bound*, visto na seção anterior. Inclusive, no mesmo artigo os autores apresentam uma definição mais formal do MinMax através de fórmulas que serão apresentadas a seguir.

Seja R_c um conjunto de pilhas candidatas para realocação de um contêiner c, isto é, $R_c = \{p \in \{1, \ldots, P\} : h(p) < H \text{ e } p \neq p(c)\}$. Definimos formalmente $min(p)$ como sendo:

$$min(p) = \begin{cases} \min_{c \in p} c & \text{se } h(c) > 0 \\ n + 1 & \text{se } h(c) = 0 \end{cases}$$

Para uma pilha p e um contêiner c tal que $p(c) \neq p$, definimos a indesejabilidade da pilha, denotado por $dif(c, p)$, da seguinte forma:

$$dif(c,p) = \begin{cases} min(p) - c & \text{se } min(p) > c \\ 2n + 1 - min(p) & \text{se } min(p) < c \end{cases}$$

Assim, quanto menor o valor de $dif(c,p)$, mais desejável é realizar a realocação do contêiner c para p.

Por fim, definimos $MinMax(c)$ como a melhor escolha de realocação para o contêiner c como sendo:

$$MinMax(c) = \arg\min_{p \in R_c} dif(c,p)\,.$$

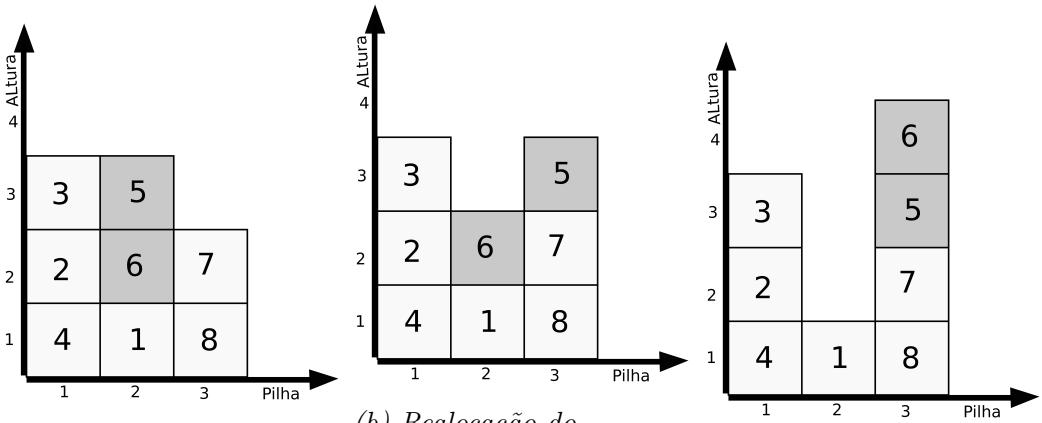
O funcionamento desse algoritmo está formalizado no Algoritmo 3.

Algoritmo 3 Heurística gulosa MinMax.

```
1:  Função MinMax(S)
2:      realocs = 0
3:      T = 1
4:      Enquanto compartimento não estiver vazio faça
5:          Enquanto T não estiver no topo da pilha p(T) faça
6:              seja c o contêiner no topo de p(T)
7:              calcule min(p) para cada p no conjunto R_c das pilhas candidatas
8:              calcule dif(c,p) para todas as pilhas candidatas
9:              realoque c para a pilha com menor dif(c,p)
10:             realocs = realocs + 1
11:         retire o alvo T do compartimento
12:         T = T + 1
13:     Devolve realocs
```

Vamos exemplificar o funcionamento do algoritmo considerando o estado com a configuração inicial da Figura 6a. Observamos que para remover o alvo atual 1, precisamos remover os contêineres 5 e 6. Como nesse caso temos a altura máxima $H = 4$, teremos as pilhas 1 e 3 como candidatas para a realocação do bloco 5. O algoritmo irá calcular o $min(p)$ de ambas as pilhas, e no caso teremos $min(1) = 2$ e $min(3) = 7$. Após calcular esses valores ele calcula o $dif(5,p)$. Seguindo a fórmula definida, teremos que $dif(5,1) = 2\times8+1-min(1) = 15$ e $dif(5,3) = min(3)-5 = 2$. Nesse caso como $dif(5,3)$ é menor, o contêiner 5 será realocado para lá, como vemos na Figura 6b, o que condiz com a heurística definida, já que a pilha 3 é a única que não causaria novas realocações ao realocar o contêiner 5. Em seguida temos que realocar o contêiner 6. Nesse caso podemos ver, pela Figura 6b, que para qualquer pilha que ele fosse realocado ele causaria uma nova realocação posteriormente. Nesse caso ainda seguindo a fórmula definida pelo MinMax, temos agora $min(1) = 2$ e $min(3) = 5$, e portanto $dif(6,1) = 2\times8+1-2 = 15$ e $dif(6,3) = 2\times8+1-5 = 12$. Logo, o contêiner 6 seria também realocado para a terceira pilha já que temos o menor valor de dif, como vemos na Figura 6c. Note que ele foi realocado para a pilha que tem o maior $min(p)$, como antes definido, já que, nesse caso, para todas

as pilhas, $min(p) < c$.



(a) Configuração inicial do compartimento. Note que $min(1) = 2$ *e* $min(3) = 7$.

(b) Realocação do contêiner 5 para a pilha 3. Note que $min(1) = 2$ *e* $min(3) = 5$.

(c) Realocação do contêiner 6 para pilha 3.

Figura 6: Exemplo de execução do MinMax para remoção do primeiro alvo.

3.5 Colônia de Formigas

O próximo método que vamos apresentar é o de otimização por colônia de formigas (ACO, do inglês *Ant Colony Optimization*). Jovanovic, Tuba e Voß [11] utilizam a heurística ACO para encontrar uma solução viável de forma eficiente para o BRP.

Primeiramente, vamos definir em linhas gerais o funcionamento do ACO. Definido em 1997 por Dorigo e Gambardella [7], o algoritmo é inspirado no comportamento de colônias de formiga na natureza para sair da colônia e coletar alimento. Imaginando uma região com vários caminhos possíveis, cada formiga ao sair da colônia para coletar o alimento inicialmente segue um caminho aleatório. Ao percorrer um caminho e retornar, elas vão depositando uma substância produzida por elas para se comunicar, o feromônio. Com o tempo esse feromônio vai evaporando, de forma que ele vai acumulando em maior quantidade no caminho mais curto, que eventualmente se torna o principal. Esse feromônio no algoritmo do ACO normalmente é representado através de uma matriz, a **matriz de feromônio**, onde cada dimensão representa uma informação do problema e a escolha dos caminhos que as formigas vão seguir é feita através de uma função heurística que usa esses valores da matriz.

Esse modelo já foi aplicado para diversos problemas na literatura [5, 9]. No caso do BRP, podemos fazer a analogia do funcionamento do ACO de forma que os caminhos percorridos por cada formiga serão as soluções parciais do problema, que representam estados que ainda têm contêineres para serem retirados, ou seja, cada formiga gera uma solução expandindo essas soluções parciais. O que veremos a seguir é como essa decisão é realizada, e como a matriz de feromônio é definida.

Primeiramente é preciso gerar uma solução inicial para o problema, sendo que os autores, para o caso restrito do problema, utilizam o algoritmo MinMax, já definido na Seção 3.4. Essa solução é usada para inicializar a matriz de feromônio.

O método funciona de forma que, quanto mais vezes o laço principal é executado, mais o algoritmo aprende com os resultados obtidos. A ideia principal consiste em uma colônia de formigas, onde cada formiga vai gerar uma solução nova, baseada na matriz de feromônio, percorrendo as soluções parciais por ela gerada, através de uma regra de transição.

Para compreender melhor, vamos definir inicialmente como é montada uma solução nesse caso. Vamos definir $min^*(p)$ para pilhas não vazias como $min^*(p) = min(p)$ e para pilhas vazias $min^*(p) = n + p$. Nas seções anteriores, representávamos uma solução para o BRP como uma sequência de pares (c, p). Agora, vamos representar uma solução C para o BRP como sendo uma sequência de 4-uplas da forma (c, d_c, m_c, T), onde c corresponde ao contêiner que será realocado, $d_c = min^*(p)$ em que p é a pilha para onde c será realocado, m_c é o número de vezes que o contêiner c

já foi realocado e, por fim, T é o alvo no estado atual. Com esses valores, é possível reconstruir uma solução completa para o problema, em que cada 4-upla de C representa uma realocação realizada no estado atingido pela realocação anterior. Vamos definir também um vetor M, que será usado para controlar o m_c, onde $M[c]$ guarda o número de vezes que um contêiner c foi realocado.

Com isso podemos definir a matriz de feromônio, que será utilizada pelas formigas para decidir qual possível realocação realizar quando em um determinado estado. Seja n o número total de contêineres no estado inicial e m o valor máximo para a quantidade de realocações m_c. Como a matriz deve estar de certa forma conectada com o estado do compartimento, definimos a matriz de feromônio com quatro dimensões, $\sigma_{n\times(n+P)\times m\times n}$, onde cada dimensão representará uma informação do estado, de forma similar à uma realocação de uma solução. Portanto a primeira dimensão armazena os valores possíveis de c, que será o contêiner a ser realocado, a segunda dimensão armazena o valor $min^*(p)$ onde p é a pilha para a qual c será realocado, a terceira armazena valores m_c e por fim a quarta dimensão armazena T para o respectivo estado. Portanto, construindo a matriz dessa forma podemos representar qualquer estado possível de um compartimento, partindo de um estado inicial.

Vamos definir então a regra de transição *select*, que define a forma como é feita a decisão de qual realocação a formiga irá realizar a partir do estado em que a solução que ela está gerando se encontra. Essa seleção poderá ser feita de duas formas: uma determinística outra probabilística. Primeiramente, definimos algumas fórmulas que são utilizadas. Considerando $f(c,p) = \dfrac{1}{1+dif(c,p)}$, temos $g(c,p) = f(c,p) \times \sigma_{c,d_c,m_c,T}$, onde (c,p) representa a realocação de um contêiner c para a pilha p, tal que $(c,p) \in R$ e R é o conjunto de todas as realocações possíveis que podem ser realizadas por c. Definimos então um parâmetro $q_0 \in [0,1]$ e escolhemos um valor q arbitrariamente para cada realocação, que nos auxiliará na seleção. Definimos a regra então da seguinte forma:

- Se $q < q_0$ então a realocação será selecionada de forma determinística. Selecionamos então, de todas as possíveis realocações (c,p) de R, a realocação que tiver o maior $g(c,p)$.
- Se $q \geq q_0$, a seleção é probabilística. Definimos então a probabilidade de cada realocação $(c,p) \in R$ ser selecionada como sendo $prob(c,p) = \dfrac{g(c,p)}{\sum_{(c',p')\in R} g(c,p)}$. Então selecionamos uma relocação (c,p) com base nessas probabilidades.

Para as regras de atualização da matriz de feromônio, seja C uma solução para o BRP. Como cada 4-upla representa uma realocação, consideramos $|C|$ como o número total de realocações realizadas na solução. Definimos então $val(C) =$

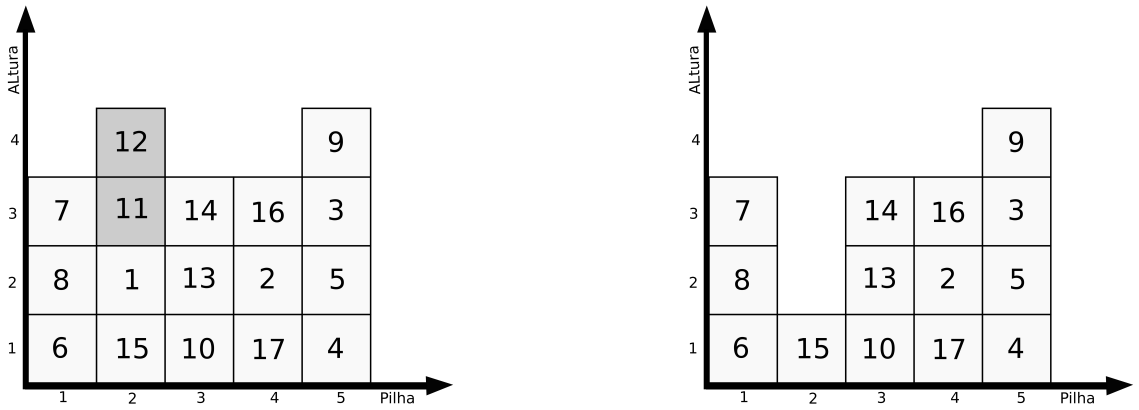
$\frac{1}{|C| + LB + 1}$ como a função que calcula a qualidade de uma solução, onde LB é o limite inferior para a solução. No nosso caso para o problema restrito, utiliza-se os limites propostos por Zhu *et al.* [20], que apresentam três limitantes inferiores diferentes. O primeiro é o número de realocações confirmadas, semelhante ao utilizado no algoritmo *branch and bound* visto na Seção 3.3, e vamos nos referir a ele como LB_1. Para o segundo, que vamos identificar como LB_2, consideramos inicialmente $LB_2 = LB_1$. Definimos então um valor $K = 0$. Seja c um contêiner bloqueante de T. Se para toda pilha p com espaço vazio no compartimento, com exceção de $p(T)$, existir um contêiner com prioridade maior que c, então $K = K + 1$. Verificamos isso para todos os bloqueantes e por fim somamos $LB_2 = LB_2 + K$. Para o LB_3, inicializamos seu valor $LB_3 = LB_2$. Para um estado S e um conjunto de contêineres G, vamos denotar por $S' = S - G$ o estado resultante ao retirar G de S. Se considerarmos S como nosso estado atual e G como o conjunto contendo o alvo e seus bloqueantes, perceba que temos um novo alvo no estado S'. Depois disso calculamos K para S', semelhante ao que fizemos com LB_2 anteriormente, e caso $K > 0$ somamos K a LB_3. Definimos então um novo conjunto G', agora com o alvo e bloqueantes de S', e definimos $S'' = S' - G'$. Calculamos um novo K para S'' e, caso $K > 0$, somamos a LB_3. Fazemos isso sucessivamente somando K a LB_3 até não poder mais, tendo por fim o valor do terceiro limite inferior.

Para entendermos a regra para calcular os limitantes inferiores, vamos observar inicialmente a Figura 7a. Considerando o número de realocações confirmadas, temos $LB_1 = 8$. Definimos $K = 0$ e $LB_2 = LB_1$. O primeiro bloqueante do alvo (contêiner 12), pode ser realocado paras as pilhas 1, 3 ou 4, e vemos que em qualquer dessas possibilidades ele estaria acima de um contêiner de maior prioridade, portanto terá que ser realocado posteriormente, logo $K = 1$. Fazemos essa análise para o contêiner 11 também e vemos que ele também causaria mais uma realocação, portanto incrementamos K em 1 mais uma vez. Temos por fim $K = 2$, logo $LB_2 = 10$. Para o LB_3 retiramos o alvo e os bloqueantes formando o estado mostrado na Figura 7b, onde nosso novo alvo agora é o contêiner 2. Consideramos um novo $K = 0$ e fazendo a mesma análise para o bloqueante 16, vemos que para qualquer pilha que ele fosse realocado ele causaria mais uma realocação, portanto $K = 1$. Seguimos então retirando os contêineres 2 e 16 do novo estado, e fazemos essa mesma análise para o próximo estado. Fazendo essa análise para os próximos estados consecutivamente, temos que, para o estado da Figura 7a, $LB_3 = 11$.

Para a regra de atualização global da matriz de feromônio, seja C_{best} a melhor solução encontrada até então e $\Delta\sigma = val(C_{best})$. Definimos

$$\sigma_{c,d_c,m_c,T} = (1 - k)\sigma_{c,d_c,m_c,T} + k\Delta\sigma, \ \forall (c, d_c, m_c, t) \in \ C_{best}, \tag{1}$$

onde o parâmetro k é um valor predefinido que representa a taxa da influência da



(a) Estado inicial contendo 17 contêineres, com 5 pilhas e altura máxima 4.

(b) Compartimento sem e seus bloqueantes. Novo alvo é o contêiner 2.

Figura 7: Configurações de compartimentos para exemplificar cálculo de LB_2 e LB_3. No estado da Figura (a) temos $LB_1 = 8$, $LB_2 = 10$ e $LB_3 = 11$.

regra global, sendo que $(1-k)$ funcionará como a taxa de evaporação de feromônio. Assim, a cada iteração da colônia atualizaremos os valores da matriz de feromônio seguindo essa regra.

Para a atualização local, seja C a solução obtida por uma formiga. Fazemos

$$\sigma_{c,d_c,m_c,T} = \varphi\ \sigma_{c,d_c,m_c,T},\ \forall (c,d_c,m_c,t) \in\ C, \tag{2}$$

onde φ é a taxa de influência, ou evaporação local.

Tendo todos esses conceitos definidos, por fim definimos somente o valor inicial para a matriz de feromônio como sendo

$$\sigma_0 = \frac{1}{H} val(C_g)\,, \tag{3}$$

onde C_g é a solução inicial encontrada pelo MinMax considerando apenas as realocações. Com isso podemos definir formalmente o método no Algoritmo 4.

O algoritmo começa gerando uma solução, para utilização como a melhor solução inicial e para realizar o cálculo inicial dos valores da matriz de feromônio. No nosso caso utilizaremos o MinMax, apresentado na Seção 3.4.

O algoritmo executa até atingir um critério de parada, que será quantas vezes desejamos que a colônia realize as iterações ou um tempo máximo de execução.

O laço de cada formiga irá executar até o compartimento estar vazio, como vemos na linha 11, ou irá parar caso a soma do número de realocações até então na solução parcial mais o limite inferior do estado atual seja maior que o número de realocações de C_{best}, como vemos na linha 21. Além disso, a cada execução de uma formiga aplicamos a regra de atualização local na matriz de feromônio com base

na solução encontrada por ela, como vemos na linha 23. Por fim quando a formiga termina a execução, verificamos na linha 24 se a solução encontrada é melhor que C_{best} atual, e se for, C_{best} é atualizada.

Por fim quando a iteração terminar para todas as formigas realizamos a regra de atualização global, na linha 28. Nesse ponto também realizamos uma verificação, na linha 26, para caso o algoritmo esteja há muito tempo sem nenhuma melhora na melhor solução encontrada e reiniciamos matriz de feromônio para recomeçar as buscas caso essa condição seja verdadeira.

Algoritmo 4 Pseudocódigo do algoritmo de colônia de formigas.

1: **Função** COLONIA(S)
2: seja C_{best} a solução gerada pelo MINMAX
3: crie a matriz de feromônio $\sigma[1..n][1..n+P][1..m][1..n]$
4: inicialize a matriz de feromônio com o valor σ_0 (3), calculado a partir de C_{best}, em todas as posições
5: defina o parâmetro q_0
6: **Enquanto** não atingir critério de parada **faça**
7: **Para** cada formiga **faça**
8: seja S' uma cópia de S
9: seja $C = ()$ uma solução vazia
10: seja $M[1..n]$ um vetor de tamanho n inicializado com 0
11: **Enquanto** o compartimento de S' não estiver vazio **faça**
12: **Enquanto** o alvo T não estiver no topo de $p(T)$ **faça**
13: seja c o bloco no topo de $p(T)$
14: calcule a lista de realocações (c, p) candidatas R
15: escolha um valor q arbitrário
16: selecione a melhor realocação utilizando *select*, com base na relação entre q e q_0
17: seja (c, p) a realocação escolhida
18: adicione $(c, min^*(p), M[c], T)$ na solução C
19: aplique a realocação (c, p) no compartimento de S'
20: $M[c] = M[c] + 1$
21: **Se** $|C| + LB(S') \geq |C_{best}|$ **então**
22: vá para a próxima formiga
23: atualize localmente a matriz de feromônio com base em C (2)
24: **Se** $|C| > |C_{best}|$ **então**
25: $C_{best} = C$
26: **Se** o algoritmo está há X iterações sem melhora **então**
27: reinicie a matriz de feromônio com σ_0
28: atualize globalmente a matriz de feromônio com base em C_{best} (1)
29: **Devolve** $|C_{best}|$

4 Testes e Resultados

Nessa seção apresentaremos e discutiremos os resultados obtidos nas implementações dos algoritmos definidos na seção anterior, além de comparar o desempenho de cada algoritmo e também verificar se a implementação aqui realizada corresponde de maneira correta às definições presentes na literatura.

Todos os algoritmos foram implementados utilizando a linguagem de programação C[1]. A execução de todos os algoritmos foi realizada no ambiente online de desenvolvimento DevCloud da Intel(R), que utiliza um processador Intel(R) Xeon (Skylake IBRS), com 16 gigabytes de memória RAM rodando com um sistema operacional baseado em Linux[2]. Para cada instância foi definido um tempo limite de execução de 10 minutos, valor ainda acima do que é apresentado nos artigos, mas que podemos considerar aceitável.

Para a realização dos testes, foi utilizado o mesmo conjunto de dados utilizado na maioria dos artigos que tratam do BRP, definido pela primeira vez em Caserta, Voß e Sniedovich [1][3]. Cada instância contém a configuração de um compartimento $(H \times P)$, onde cada pilha está preenchida até a altura $H-2$. Portanto cada instância tem um total de $(H-2) \times P$ contêineres. O conjunto de testes utilizado tem compartimentos com 21 tamanhos diferentes e para cada tamanho de compartimento existem 40 conjuntos de instâncias para a realização dos testes.

4.1 Testes dos algoritmos exatos

Inicialmente fizemos a comparação dos dois métodos exatos que apresentamos, um seguindo o modelo de programação dinâmica que percorre todos os possíveis estados, apresentado na Seção 3.1, e o outro o método do *branch and bound* (B&B), apresentado na Seção 3.3. Como vimos, já foi provado que o problema do BRP é NP-Difícil, portanto não pode ser resolvido em tempo polinomial a menos que $P = NP$. Assim, qualquer método exato já desenvolvido para esse problema tem tempo inviável conforme aumentamos o tamanho das instâncias que se tenta resolver. E isso nós pudemos comprovar pelos testes, que têm os resultados apresentado na Tabela 2, onde apresentamos para cada tamanho de compartimento a média de realocações e de tempo de execução (em segundos) obtidas executando as 40 instâncias de cada tamanho de compartimento.

Primeiramente, para o algoritmo desenvolvido usando o modelo de programação dinâmica, como esperado, para instâncias menores ele se mostra eficiente e consegue

[1] https://github.com/Febinhas/BRP_PGC_Codigo

[2] https://www.intel.com/content/www/us/en/developer/tools/devcloud/overview.html

[3] https://www.bwl.uni-hamburg.de/en/iwi/forschung/projekte/dataprojekte/brp-instances-caserta-etal-2012.zip

resolver os problemas em tempo aceitável. Já para instâncias maiores, a partir dos problemas (3×5) a execução começa a demorar mais que dez minutos para boa parte das instâncias, e por isso não obtivemos solução para todas as instâncias.

Já no B&B, podemos observar, além de um tempo de execução mais rápido, que o algoritmo consegue executar dentro de um tempo aceitável até as instâncias (4×6). Porém, a partir disso, como esperado o método começa se tornar inviável, com exceção das instâncias de tamanho (5×4) que também foram todas executadas. No restante, o algoritmo começa a apresentar problema de memória devido à limitação dos computadores, ou demora demais para executar, fazendo com que somente algumas instância tenham sido resolvidas. Para ambos os algoritmos, os casos que são marcados com "-" na Tabela 2 significam que o algoritmo não conseguiu executar todas as 40 instâncias no compartimento.

Acrescentamos, para fins de comparação, a última coluna, com os resultados obtidos por Expósito-Izquierdo *et al.* [8] com seu algoritmo baseado no algoritmo A*. Vemos que para os casos onde todas as 40 instâncias executaram os valores são iguais, e portanto assumimos que os algoritmos implementados estão corretos. Apesar dos resultados serem iguais, na Tabela 2 só consideramos os casos em que todas as instâncias conseguiram executar, de forma que podemos ver que as nossas implementações, apesar de corretas, conseguem executar menos instâncias que o algoritmo de Expósito-Izquierdo *et al.* [8]. É importante observar que, para o método de programação dinâmica isso já era esperado, mas no caso do B&B nessas primeiras instâncias em que ele começa a falhar, isso se dá mais pela falta de memória, devido à implementação, do que efetivamente por passar do tempo limite.

Tabela 2: Média de realocação e tempo de execução em segundos para os algoritmos de programação dinâmica, branch and bound e A (Expósito-Izquierdo et al.).*

	Prog. Dinâmica		**B&B**		**Exposito (A*)**
Instância	Realocações	Tempo(s)	Realocações	Tempo(s)	Realocações
3×3	5.00	0.000140	5.00	0.000041	5.00
3×4	6.18	0.048223	6.18	0.000178	6.18
3×5	-	-	7.02	0.000574	7.02
3×6	-	-	8.40	0.001590	8.40
3×7	-	-	9.28	0.001672	9.28
3×8	-	-	10.65	0.016001	10.65
4×4	-	-	10.20	0.003918	10.20
4×5	-	-	12.95	0.071347	12.95
4×6	-	-	14.02	1.144459	14.02
4×7	-	-	-	-	16.12
5×4	-	-	15.42	0.121945	15.42
5×5	-	-	-	-	18.85
5×6	-	-	-	-	22.08

Ainda assim, no *branch and bound*, para algumas instâncias maiores dos conjuntos que não executaram completamente, tivemos alguns resultados obtidos. Pode-

Tabela 3: Resultado da execução incompleta do algoritmo branch and bound e número de instâncias que conseguiram ser executadas.

Branch and Bound		
Instância	Realocações	Nº de executados
4×7	15.78	37 (92.5%)
5×5	18.18	34 (85%)
5×6	21.23	26 (65%)
5×7	21.93	15 (37.5%)

mos vê-los na Tabela 3, que mostra também a quantidade de instâncias que foram executadas para cada tamanho de compartimento.

4.2 Testes para o método colônia de formigas

O segundo teste realizado foi com o método colônia de formigas. Por se tratar de uma heurística, vemos que o algoritmo consegue encontrar solução para todas as instancias do problema, porém obviamente não temos garantia de que o resultado será o ótimo. Como visto anteriormente, o algoritmo colônia de formigas utiliza-se de alguns parâmetros que são definidos previamente antes da execução. Temos k representando a taxa de influência da regra global de transição, φ representando a taxa de influência da regra local e por fim a taxa de exploração que representamos por q_0. Para os primeiros testes realizados para esse método utilizamos os valores para esses parâmetros definidos por Jovanovic, Tuba e Voß [11], portanto temos $k = 0.1$, $\varphi = 0.9$ e $q_0 = 0.9$. Além disso também foram utilizados os mesmo valores do artigo para a quantidade de formigas da colônia, 10, e para o critério de parada que representa o número total de iterações a serem realizadas, que é 5000. Também quando o algoritmo executa 100 iterações sem haver nenhuma melhora a matriz de feromônio é reiniciada.

Primeiramente foi realizado o teste comparando os resultados utilizando os três limites inferiores apresentados em *"Iterative Deepening A* Algorithms for the Container Relocation Problem"* de Zhu *et al.* [20]. De forma geral pudemos observar, conforme mostrado na Tabela 4, que na maioria dos casos de teste houve uma pequena melhora nos resultados ao mudar o limite inferior, com exceção de algumas instâncias, como as (10×6), que apresentaram uma ligeira piora ao comparar o LB_1 e o LB_2 com o LB_3. Porém, apesar dessas pequenas melhoras, não podemos garantir de fato que os limites inferiores afetam totalmente os resultados para o algoritmo implementado, pois as mudanças notadas são bem pequenas. Além disso, pela forma que o algoritmo é desenvolvido é esperada uma variação de resultados a cada execução de uma mesma instância.

Além disso, outro ponto que podemos observar é o significativo aumento na média do tempo de execução dos problemas ao compararmos LB_1 com LB_2 e LB_3, o que

Tabela 4: Resultado da execução do algoritmo colônia de formigas com os três diferentes limites inferiores.

Colônia de formigas						
	Lower Bound 1		Lower Bound 2		Lower Bound 3	
Instância	Realocações	Tempo	Realocações	Tempo	Realocações	Tempo
3×3	5.00	0.174	5.00	0.404	5.00	0.487
3×4	6.18	0.290	6.18	0.644	6.18	0.836
3×5	7.02	0.463	7.02	1.082	7.02	1.645
3×6	8.40	0.693	8.40	1.652	8.40	2.807
3×7	9.28	1.007	9.28	1.996	9.28	3.794
3×8	10.65	1.398	10.65	2.995	10.65	4.961
4×4	10.22	0.412	10.22	1.142	10.20	1.597
4×5	12.95	0.719	12.95	1.786	12.95	2.953
4×6	14.02	1.085	14.02	2.510	14.02	4.989
4×7	16.12	1.525	16.12	3.918	16.12	6.453
5×4	15.45	0.611	15.45	1.693	15.45	2.317
5×5	19.00	0.903	19.00	2.395	19.02	3.535
5×6	22.22	1.975	22.25	3.347	22.20	4.765
5×7	24.50	1.636	24.40	4.956	24.45	8.175
5×8	27.85	2.519	27.85	7.075	27.82	11.712
5×9	30.52	2.931	30.52	8.953	30.50	14.209
5×10	33.50	3.987	33.52	10.877	33.45	18.708
6×6	31.65	1.296	31.7	5.506	31.55	6.283
6×10	46.62	3.716	46.5	16.316	46.52	21.652
10×6	85.72	3.180	85.68	24.619	85.82	21.809
10×10	124.18	7.237	124.18	85.341	123.95	67.433

é esperado pela forma como esses limites são calculados. Vemos que nas instâncias maiores esse aumento pode chegar a mais de 1179%, como nos casos (10×10) onde utilizando LB_1 a média de execução fica na casa dos 7 segundos, enquanto para o LB_2 esse valor chega a mais de 85 segundos. Se extrapolarmos para problemas ainda maiores, esse tempo deve ficar maior ainda, o que pode tornar uso dos limites inferiores 2 e 3 não muito vantajoso visto que a melhora ao usá-los não foi muito significativa nos resultados dos testes realizados.

Como dito anteriormente, os testes foram realizadas utilizando os valores definidos originalmente no artigo pelos autores para os parâmetros utilizados no algoritmo. Porém, um teste válido seria variar esses valores para verificar se é possível obter melhores resultados. Fizemos isso com diversas combinações de valores, porém sem obter resultados positivos. Inicialmente os valores de k e φ foram fixados iguais aos originais e variamos o valor de q_0, e em todos os testes realizados os resultados obtidos foram maiores que os mostrado na Tabela 4. Portanto assume-se que $q_0 = 0.9$ é o melhor valor a ser utilizado para esse parâmetro. Tendo q_0 fixado, foram realizados testes variados, com diferentes combinações para os valores de k e φ. Porém, novamente, para todos os casos, quando considerando a média total do número de realocações para um conjunto de instâncias, não foi observada nenhuma melhoria nos valores obtidos.

Finalizando os testes do ACO, mostramos na Tabela 5 a comparação dos resultados obtidos pela nossa implementação, com os obtidos no artigo original [11]. Vale ressaltar que é esperado haver diferença nos valores, já que o algoritmo não é exato e pode obter resultados diferentes a cada execução. Podemos observar que para as instâncias maiores essa diferença foi consideravelmente maior, mas a maioria dos resultados obtidos foram condizentes com os obtidos no artigo. Para os casos em que a diferença é mais discrepante, acreditamos que a diferença na implementações deve ser o maior motivo.

Tabela 5: Comparação dos resultados do algoritmo Colônia de formigas. A coluna "Implementado" representa os resultados do algoritmo implementado no nosso trabalho e a coluna "Artigo" apresenta os resultados do artigo original.

Colônia de formigas		
Instância	Implementado	Artigo
3×3	5.00	5.00
3×4	6.18	6.18
3×5	7.02	7.02
3×6	8.40	8.40
3×7	9.28	9.28
3×8	10.65	10.65
4×4	10.22	10.20
4×5	12.95	12.95
4×6	14.02	14.02
4×7	16.12	16.12
5×4	15.45	15.42
5×5	19.02	18.95
5×6	22.20	22.15
5×7	24.45	24.33
5×8	27.82	27.73
5×9	30.50	30.50
5×10	33.45	33.40
6×6	31.55	31.05
6×10	46.52	45.93
10×6	85.82	79.50
10×10	123.95	113.45

4.3 Método Corredor

Antes de mostrarmos a comparação entre todos os resultados obtidos, vale ressaltar um detalhe da implementação do método do corredor. A nossa implementação foi realizada seguindo uma interpretação do artigo *"Applying the corridor method to a blocks relocation problem"* [3], que descreve o método como sendo baseado em um modelo dinâmico como visto anteriormente. Porém, como podemos observar na Tabela 6, os resultados obtidos pela nossa implementação não foram satisfatórios quando comparados aos resultados obtidos pelo método do corredor nos testes do artigo *"An efficient ant colony optimization algorithm for the blocks relocation*

Tabela 6: Resultados obtidos pelo método do corredor com o algoritmo implementado e resultados do artigo do método colônia de formigas.

Método corredor		
Instância	Implementado	Artigo
3×3	5.98	5.00
3×4	7.60	6.18
3×5	9.02	7.02

problem" [11], já que na maioria dos casos o tempo máximo de execução era atingido. Como possível é observar, somente 3 tamanhos de instâncias conseguiram ser totalmente executados.

Com objetivo de entender os motivos dessa disparidade entre desempenho e resultados, foi consultada a versão do código disponibilizada pelo autor[4]. Ela parece ser a mesma utilizada nos testes no artigo *"An efficient ant colony optimization algorithm for the blocks relocation problem"* [11], porém é possível perceber uma grande diferença entre as implementações. Na nossa implementação exploramos todas as possíveis soluções para uma instância, sempre considerando o corredor definido, de fora que o código passa por diversas recursões, o que causa um problema de desempenho. Já no código desenvolvido pelo autor, podemos perceber que na verdade a cada contêiner a ser realocado da pilha do alvo se realiza uma heurística, semelhante ao MinMax, para selecionar a possível melhor realocação dentre as opções do corredor, o que difere bastante da interpretação obtida no artigo original do método.

Como mencionado, essa é uma versão mais recente do código e a única encontrada, já que os links para o código fonte original já não funcionam mais atualmente. Portanto, não é possível garantir se o código originalmente implementado no artigo utiliza-se da mesma estratégia. Inclusive há um comentário no código mais recente que pode corroborar isso. Primeiramente, com a implementação do artigo original algumas instâncias não foram resolvidas, semelhante ao caso da nossa implementação como mostrado na Tabela 7, fazendo com que a média apresentada no artigo fosse parcial para alguns tamanhos de compartimento. Porém o autor não indica em quais tamanhos não foi possível executar as 40 instâncias. Além disso, o autor cita que no artigo original o limite de altura máxima do compartimento não era respeitado, portanto alguns resultados apresentados no artigo aparentam ser melhores que os mais recentes, porém isso se dá devido a erros na implementação. É por esse motivo que os resultados utilizados na comparação foram os obtidos do artigo do método colônia de formigas [11].

[4] https://github.com/marcocaserta/BRP

Tabela 7: Média de realocações para as instâncias executadas com o método corredor.

Método Corredor		
Instância	Realocações	Nº de executados
3×3	5.98	40
3×4	7.60	40
3×5	9.02	40
3×6	9.15	34
3×7	9.26	19
3×8	10.69	16
4×4	10.86	35

4.4 Comparação entre todos os algoritmos

Por fim, comparamos os resultados obtidos entre todos os métodos implementados nesse trabalho. Como já esperado, o método colônia de formigas, que também é o método mais recente do todos os estudados, é o que consegue melhores resultados, sendo que até as instâncias (4×6) ele conseguiu resolver de forma ótima, o que torna o ACO muito vantajoso, pois ele conseguiu resolver de forma ótima quase todas as instâncias que conseguiram serem resolvidas pelo *branch and bound*. Podemos ver os resultados na Tabela 8. Podemos reparar que no caso do MinMax, nenhuma das instâncias foram resolvidas de forma ótima, mas nas instâncias menores os valores ficam bem próximos. Para fazer uma análise mais aprofundada dos casos que não são resolvidos de forma ótima, apresentamos, nas Tabelas 9 e 10, a porcentagem das instâncias que conseguiram ser resolvidas otimamente, considerando as instâncias que conseguiram ser resolvidas também pelo *branch and bound*.

Vemos que no caso do MinMax, até o tamanho (3×8) quase todos os casos obtiveram resultados ótimos, mas a partir dos problemas (4×4) essa proporção cai bastante, e boa parte não consegue esses resultados. Já no caso do colônia de formigas, no único caso em que podemos comparar com o *branch and bound* e que não foi resolvido de forma ótima, somente uma das instâncias não conseguiu esse resultado ótimo, aumentando um pouco a média do número de realocações.

4.5 Resumo dos resultados

Como podemos observar, os testes refletiram, na grande maioria dos casos. o que era esperado. Foi possível observar os algoritmos exatos em ação e comprovar que a partir de um certo tamanho de instância eles já não conseguem resolver em um tempo aceitável, considerando uma aplicação prática do algoritmo. Além disso, vimos que as heurísticas estudadas durante o trabalho, com exceção do método do corredor, já conseguem resolver problemas maiores, porém sem garantia de resultados ótimos. Ainda assim, conseguiram resolver otimamente algumas das instâncias, como é o caso da colônia de formigas, que por sua vez se provou como a melhor

Tabela 8: Média do número de realocações para todos os algoritmos implementados. A coluna "OPT" apresenta a média do número ótimo de realocações.

Instância	Prog. Dinâmica	B&B	Corredor	MinMax	ACO	OPT
3×3	5.00	5.00	5.98	5.08	5.00	5.00
3×4	6.18	6.18	7.60	6.30	6.18	6.18
3×5	-	7.02	9.02	7.08	7.02	7.02
3×6	-	8.40	-	8.48	8.40	8.40
3×7	-	9.28	-	9.32	9.28	9.28
3×8	-	10.65	-	10.75	10.65	10.65
4×4	-	10.20	-	11.00	10.20	10.20
4×5	-	12.95	-	13.68	12.95	12.95
4×6	-	14.02	-	14.72	14.02	14.02
4×7	-	-	-	17.00	16.12	16.12
5×4	-	15.42	-	16.68	15.45	15.42
5×5	-	-	-	21.28	19.02	18.85
5×6	-	-	-	24.40	22.22	22.08
5×7	-	-	-	26.45	24.45	-
5×8	-	-	-	29.82	27.82	-
5×9	-	-	-	32.62	30.55	-
5×10	-	-	-	35.62	33.45	-
6×6	-	-	-	36.10	31.65	-
6×10	-	-	-	50.00	46.52	-
10×6	-	-	-	101.75	85.88	-
10×10	-	-	-	139.52	124.2	-

Tabela 9: Porcentagem de instâncias resolvidas de forma ótima pelo MinMax.

MinMax		
Instância	Realocações	% ótimos
3×3	5.08	92.5%
3×4	6.30	90%
3×5	8.48	95%
3×6	9.32	92.5%
3×7	9.26	95%
3×8	10.75	90%
4×4	11.00	47.5%
4×5	13.68	55%
4×6	14.72	62.5%
5×4	16.68	30%

Tabela 10: Porcentagem de instâncias resolvidas em tempo ótimo pelo colônia de formigas.

Colônia de formigas		
Instância	Realocações	% ótimos
5×4	15.45	97.5%

opção dos algoritmos estudados, conseguindo os melhores resultados em um tempo relativamente pequeno.

Os testes foram realizados utilizando as instâncias utilizadas em grande parte dos artigos da literatura e vão até o tamanho (10×10). Os testes poderiam ainda ser extrapolados para instâncias maiores, por exemplo (100×100), mas além de provavelmente consumir muito tempo para serem executados, quando pensamos na aplicação prática do problema da Realocação de Contêineres, fica inviável considerar uma aplicação no mundo real de um problema desse tamanho. Por isso nos restringimos às instâncias encontradas.

5 Considerações Finais

O objetivo geral desse trabalho foi fazer um estudo e implementação para comparação de alguns algoritmos da literatura para o problema de realocação de contêineres, ou BRP (do inglês *"Blocks relocation problem"*). Entre os métodos vistos temos dois exatos, sendo um baseado em programação dinâmica e outro no algoritmo *branch and bound*, e três heurísticas, sendo uma o método do corredor, outra o MinMax e a terceiro a colônia de formigas. Por fim, foram realizados testes e comparações entre os resultados obtidos usando bancos de dados clássicos para o problema.

Nos métodos exatos foram estudadas formas de resolver o problema sempre de forma ótima. Em instâncias menores do problema esse objetivo foi atingido com sucesso, porém pelo problema ser NP-Difícil essas abordagens se mostraram de fato ineficazes em instâncias maiores, apresentando sempre problemas relacionados ao tempo de execução ou falta de memória. Nos algoritmos heurísticos, encontramos formas confiáveis de resolver a maioria dos casos do BRP, porém agora sem a garantia da otimalidade dos resultados. Porém, como observado, foi possível obter resultados satisfatórios principalmente no caso da colônia de formigas.

No decorrer do trabalho, também houve algumas dificuldades relacionadas à implementação, principalmente no caso do método do corredor, onde houve uma disparidade entre o algoritmo desenvolvido, que foi baseado na interpretação do artigo em que ele foi apresentado, e o código disponibilizado pelos autores originais do artigo. Devido a esse problema não foi possível realizar uma análise completa do método, já que o contato com os autores não foi bem sucedido. Porém apesar dos imprevistos, foi possível obter sucesso na implementação dos outros algoritmos e realizar uma comparação satisfatória entre eles..

De forma geral esse estudo serviu como uma forma de aplicar parte do conhecimento adquirido na graduação. Foi possível observar como é feita a modelagem de um problema prático do mundo real em um problema de otimização de forma a se conseguir aplicar técnicas de projeto de algoritmos para esse tipo de problema. Foi possível aplicar modelos clássicos estudados, como é o caso do *branch and bound*, em que foi possível ver sua aplicação prática e observar que ele pode ser utilizado na resolução de diversos problemas, assim como modelos mais modernos e rebuscados, como é o caso do algoritmo colônia de formigas, que utiliza de uma abstração de um comportamento da natureza resolver problemas.

Por fim, é possível afirmar que esse trabalho tenta contribuir como uma forma de condensar parte do conhecimento que foi desenvolvido nos últimos anos para o problema do BRP restrito, e serve como uma motivação para o desenvolvimento de métodos novos. Em pesquisas futuras podemos propor esse estudo a fim de se conseguir resultados ainda melhores que os já presentes na literatura.

Referências

[1] M. Caserta, S. Schwarze, and S. Voß. A new binary description of the blocks relocation problem and benefits in a look ahead heuristic. pages 37–48, 2009. URL https://doi.org/10.1007/978-3-642-01009-5_4.

[2] M. Caserta, S. Schwarze, and S. Voß. *Container Rehandling at Maritime Container Terminals*, pages 247–269. Springer New York, New York, NY, 2011. ISBN 978-1-4419-8408-1. URL https://doi.org/10.1007/978-1-4419-8408-1_13.

[3] M. Caserta, S. Voß, and M. Sniedovich. Applying the corridor method to a blocks relocation problem. *OR Spectrum*, 33(4):915–929, 2011. URL https://doi.org/10.1007/s00291-009-0176-5.

[4] M. Caserta, S. Schwarze, and S. Voß. A mathematical formulation and complexity considerations for the blocks relocation problem. *European Journal of Operational Research*, 219(1):96–104, 2012. ISSN 0377-2217. URL https://doi.org/10.1016/j.ejor.2011.12.039.

[5] W.-N. Chen and J. Zhang. An ant colony optimization approach to a grid workflow scheduling problem with various qos requirements. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 39(1): 29–43, 2009. doi: 10.1109/TSMCC.2008.2001722.

[6] A. d. S. Firmino, R. M. de Abreu Silva, and V. C. Times. An exact approach for the container retrieval problem to reduce crane's trajectory. In *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*, pages 933–938, 2016. URL https://doi.org/10.1109/ITSC.2016.7795667.

[7] M. Dorigo and L. M. Gambardella. Ant colonies for the travelling salesman problem. *Biosystems*, 43(2):73–81, 1997. URL https://doi.org/10.1016/S0303-2647(97)01708-5.

[8] C. Expósito-Izquierdo, B. Melián-Batista, and J. M. Moreno-Vega. A domain-specific knowledge-based heuristic for the Blocks Relocation Problem. *Advanced Engineering Informatics*, 28(4):327–343, 2014. ISSN 1474-0346. URL https://doi.org/10.1016/j.aei.2014.03.003.

[9] M. Guntsch, M. Middendorf, and H. Schmeck. An ant colony optimization approach to dynamic tsp. page 860–867, 2001.

[10] K. Jansen. The mutual exclusion scheduling problem for permutation and comparability graphs. *Information and Computation*, 180(2):71–81, 2003. URL https://doi.org/10.1016/S0890-5401(02)00028-7.

[11] R. Jovanovic, M. Tuba, and S. Voß. An efficient ant colony optimization algorithm for the blocks relocation problem. *European Journal of Operational Research*, 274(1):78–90, 2019. ISSN 0377-2217. URL https://doi.org/10.1016/j.ejor.2018.09.038.

[12] K. H. Kim and G.-P. Hong. A heuristic rule for relocating blocks. *Computers & Operations Research*, 33(4):940–954, 2006. ISSN 0305-0548. URL https://doi.org/10.1016/j.cor.2004.08.005. Part Special Issue: Optimization Days 2003.

[13] Y. Lee and Y.-J. Lee. A heuristic for retrieving containers from a yard. *Computers & Operations Research*, 37(6):1139–1147, 2010. ISSN 0305-0548. URL https://doi.org/10.1016/j.cor.2009.10.005.

[14] J. Lehnfeld and S. Knust. Loading, unloading and premarshalling of stacks in storage areas: Survey and classification. *European Journal of Operational Research*, 239(2):297–312, 2014. ISSN 0377-2217. URL https://doi.org/10.1016/j.ejor.2014.03.011.

[15] D. Steenken, S. Voß, and R. Stahlbock. Container terminal operation and operations research - A classification and literature review. *OR Spectrum*, 26(1):3–49, 2004. ISSN 1436-6304. URL https://doi.org/10.1007/s00291-003-0157-z.

[16] L. Tang, W. Jiang, J. Liu, and Y. Dong. Research into container reshuffling and stacking problems in container terminal yards. *IIE Transactions*, 47(7): 751–766, 2015. doi: 10.1080/0740817X.2014.971201. URL https://doi.org/10.1080/0740817X.2014.971201.

[17] R. S. UNCTAD. Review of maritime transport 2019, 2019. URL https://unctad.org/en/PublicationsLibrary/rmt2019_en.pdf. Geneva-New York.

[18] E. Zehendner, D. Feillet, and P. Jaillet. An algorithm with performance guarantee for the Online Container Relocation Problem. *European Journal of Operational Research*, 259(1):48–62, 2017. ISSN 0377-2217. URL https://doi.org/10.1016/j.ejor.2016.09.011.

[19] M. Zhu, X. Fan, and Q. He. A heuristic approach for transportation planning optimization in container yard. In *2010 IEEE International Conference on Industrial Engineering and Engineering Management*, pages 1766–1770, 2010. URL https://doi.org/10.1109/IEEM.2010.5674565.

[20] W. Zhu, H. Qin, A. Lim, and H. Zhang. Iterative Deepening A* Algorithms for the Container Relocation Problem. *IEEE Transactions on Automation Science and Engineering*, 9(4):710–722, 2012. URL https://doi.org/10.1109/TASE.2012.2198642.