Coordenadoria do Curso de Ciência da Computação

Universidade Estadual de Mato Grosso do Sul

Investigação de técnicas de computação natural aplicadas no problema de empacotamento unidimensional

Fabiano Correia de Oliveira
Renan Cardena de Souza

Prof. Dr. Osvaldo Vargas Jaques
(Orientador)

Prof. Alcione Ferreira
(Coorientador)

Dourados
Novembro de 2014

Investigação de técnicas de computação natural aplicadas no problema de empacotamento unidimensional

Fabiano Correia de Oliveira
Renan Cardena de Souza

Monografia da disciplina Projeto Final de Curso devidamente corrigida e defendida por Fabiano Correia de Oliveira e Renan Cardena de Souza e aprovada pela Banca Examinadora, como parte dos requisitos para a obtenção do título de Bacharel em Ciência da Computação.

Dourados, 7 de Novembro de 2014.

Prof. Dr. Osvaldo Vargas Jacques
(Orientador)

Prof. Alcione Ferreira
(Coorientador)

Agradecimentos

Primeiramente a Deus, por deixar que eu concluísse mais essa etapa da minha vida.

Aos meus pais, Maria Betania Araujo Correia de Oliveira e Hugo João Borges de Oliveira, pelo apoio e incentivo e a Miqueline Ribeiro de Souza que está ao meu lado a um tempo já e que continuou ao meu lado me incentivando e ajudando.

Ao Prof. Dr. Osvaldo Vargas Jaques pela paciência, compreensão e orientação com este trabalho e ao Prof. Dr. Rubens Barbosa Filho pela motivação, ajuda e paciência passada a mim durante o ano todo.

Ao meu amigo e colega de turma, Renan Cardena de Souza, que me ajudou a concluir essa etapa.

Às pessoas que contribuíram diretamente ou indiretamente a esse trabalho.

Fabiano Correia de Oliveira

A Deus, por me dar forças e saúde para superar as dificuldades e permitir que tudo isso acontecesse em minha vida.

A todos os professores do curso, que foram tão importantes na minha vida acadêmica, em especial aos professores Osvaldo Vargas Jacques e Alcione Ferreira pela orientação e paciência no desenvolvimento deste trabalho e ao professor Rubens, pelas críticas e sugestões dadas durante este ano.

A minha mulher, Daniela Pereira de Toledo Cardena, por acrescentar razão e beleza aos meus dias e me apoiar em todos os momentos deste curso.

Aos meus pais, Raul Oliveira de Souza e Nidene Cardena Souza, heróis que me deram apoio nas horas difíceis, e por todo investimento dado à minha formação acadêmica e, principalmente, pessoal.

Ao meu amigo e colega de turma desde o ensino fundamental, Fabiano Correia de Oliveira, que me ajudou na conclusão, não apenas deste trabalho, mas também deste curso.

A todas as pessoas que contribuíram, direta ou indiretamente, para a conclusão deste trabalho.

Renan Cardena de Souza

Resumo

No atual andamento da indústria, a eficiência do uso de matéria prima em indústrias no geral é imprescindível. Buscando reduzir o descarte de material no corte unidimensional, foi proposto o uso de algoritmos naturais que, mesmo possuindo um tempo de processamento mais alto, muitas vezes conseguem resultados melhores que soluções computacionais tradicionais.

Esta monografia apresenta um breve estudo sobre a computação natural e o problema de empacotamento. Serão abordados dois algoritmos principais: os Algoritmos Genéticos e o *Simulated Annealing*. Estes serão aplicados no resultado do algoritmo *Best Fit Decreasing*, com o objetivo de otimizar esta solução.

Palavras-chaves: *Algoritmos Genéticos, Simulated Annealing, Empacotamento, Unidimensional, Computação Natural.*

Abstract

Nowadays, at the economy, the efficiency of the resources using in the industries, is indispensable. Trying to reduce the waste of material in the one-dimensional cutting, the use of natural algorithms was proposed. Even with a bigger processing time, they usually have better results than the traditional computing solutions.

This work shows a short study about the natural computing and the scaling problem. Two main algorithms are going to be brought up: the Genetic Algorithms and the Simulated Annealing. They are going to be applied in the result of Best Fit Decreasing, to optimize this solution.

Key-words: Genetic Algorithm, Simulated Annealing, One-directional Scaling, Natural Computing.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

A curiosidade humana sempre impulsionou a busca por melhores soluções de diversos problemas, levando o homem a tentar imitar a natureza para alcançar objetivos mais eficientes. Então surgiu a computação natural, onde a natureza tem grande influência no desenvolvimento de novos mecanismos computacionais.

Uma das áreas da computação natural, tem como o objetivo desenvolver, a partir de modelos naturais, soluções para problemas os quais modelos computacionais tradicionais não foram suficientes.

A Inteligência Artificial, ao longo dos anos, vem sido massivamente estudada. Dentro de Inteligência Artificial temos alguns algoritmos que merecem a nossa atenção por, muitas vezes, oferecerem resultados mais eficientes em relação à soluções tradicionais. São eles os Algoritmos Genéticos (AG) e o *Simulated Annealing (SA).*

1.1 Justificativa

O algoritmo *Best Fit Decreasing* (BFD) é um algoritmo que serve para maximizar cortes de tamanhos l_i de barras de tamanho pré-definidos $L \geq l_i$, resultando no mínimo possível de sobras. Este algoritmo pode ser aplicado em repartições que vendem cortes de materiais para com isso maximizar o lucro evitando perdas o máximo possível.

1.2 Objetivos

A partir dos resultados do algoritmo BFD, queremos aplicar os algoritmos de computação natural visando obter um melhor resultado que o BFD sozinho. Teremos então dois novos algoritmos: o algoritmo BFD-SA e o algoritmo BFD-AG. A proposta é melhor entendida conforme a Figura 1.1.

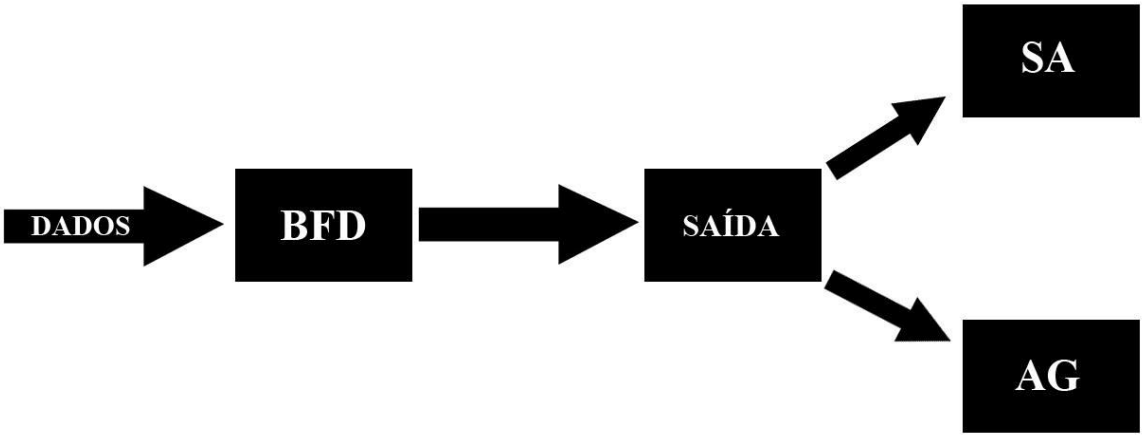

Figura 1.1 - Ilustração do objetivo geral do trabalho

1.2.1 Objetivos específicos

a) Estudo e caracterização dos Algoritmos Genéticos e *Simulated Annealing*, presentes na área da Inteligência Artificial;
b) Análise e comparação dos resultados gerados pelos algoritmos citados neste texto, ou seja, os algoritmos BFD, BFD-SA e BFD-AG.

1.3 Conteúdo e organização do trabalho

Para a realização deste trabalho, foram utilizadas bibliografias da grande área da Computação Natural e, dentro da mesma, a Inteligência artificial, bem como modelos gerados aleatoriamente pelos autores deste texto.

No Capítulo 2, é feita uma revisão literária para que o leitor desta monografia seja introduzido aos conceitos utilizados no mesmo. Em suas seções, este capítulo descreve brevemente a computação natural, a pesquisa operacional, a inteligência artificial e o problema de

empacotamento unidimensional e expõe as características dos algoritmos utilizados, os Algoritmos Genéticos (GA), *Simulated Annealing* (SA) e *Best Fit Decreasing* (BFD).

No Capítulo 3, apresentamos o algoritmo usado para gerar os resultados iniciais. A seguir, apresentamos as modificações feitas nos algoritmos naturais para que pudessem aplicados ao BFD com o intuito otimizar os resultados.

Os resultados e a análise dos resultados apresentados pelos algoritmos tratados neste trabalho são apresentados no Capítulo 4. Finalmente, no Capítulo 5, temos as considerações finais e conclusão.

2 Revisão Literária

2.1 Computação Natural

Neste capítulo serão abordados conceitos e principais motivos para o surgimento da computação natural. A seção 2.1.1 apresenta um breve histórico sobre a computação natural, bem como seus conceitos e os uso da mesma neste trabalho.

2.1.1 Histórico

Logo em seu início, a humanidade buscou na Natureza seu abrigo e seu alimento e, rapidamente aprendeu a manipular os recursos oferecidos por ela. Assim, passou a observar e estudar os padrões físicos, biológicos e químicos.

Estudando os fenômenos naturais, o homem potencializou a capacidade de manipular a natureza. Paralelamente, percebeu que a mesma possui uma grande capacidade de resolver problemas. Então esta entidade passa a ser enxergada como uma fonte de inspiração para o desenvolvimento de sistemas e artefatos.

Entre os mais antigos exemplos de modelos de computação natural, temos o Autômato Celular, concebido por Ulam e von Neumann na década de 40. John von Neumann investigou o uso deste modelo como uma estrutura para a compreensão do comportamento de sistemas complexos. (KALI E ROZENBERG, 2008).

Desde então, ideias baseadas em sistemas naturais vêm sido utilizadas com muito sucesso no desenvolvimento de ferramentas com o objetivo de solucionar problemas não necessariamente otimizados e considerados difíceis e custosos.

2.1.2 Conceito

A terminologia Computação Natural vem sendo empregada na literatura para descrever todos os sistemas construídos tendo como inspiração ou a utilização de mecanismos naturais ou

biológicos para o processamento de informação (BALLARD, 1999; FLAKE, 2000; CASTRO & VON ZUBEN, 2004).

Esta área na computação pode ser vista como uma versão computacional de processos de análise e síntese da natureza para o desenvolvimento de sistemas computacionais artificiais.

2.1.3 As três áreas da Computação Natural

De acordo com Castro e Von Zuben (2004), a área da Computação Natural pode ser dividida em três subáreas: Computação Inspirada na Natureza, Estudos sobre a Natureza através da Computação e Computação com mecanismos naturais.

2.1.3.1 Computação Inspirada na Natureza

Esta subárea abrange as estratégias construídas ou inspiradas em mecanismos naturais. Por exemplo, as redes neurais artificiais (HAYKIN, 1999) e a Inteligência coletiva (BONABEAU et. al. 1999; KENNEDY, 2001).

Vale destacar que o próprio raciocínio humano representa um mecanismo natural de fundamental importância. Então, a inteligência artificial simbólica (RUSSELL & NORVIG, 2004) e os sistemas nebulosos (PEDRYCZ & GOMIDE, 1998), também podem ser inseridos no escopo da computação natural.

Esta será a área da computação natural estudada neste trabalho, mais especificamente os Algoritmos Genéticos e *Simulated Annealing*.

2.1.3.2 Estudos sobre a Natureza através da computação

De maneira oposta à computação inspirada na natureza, esta subárea tende a analisar fenômenos naturais, através da computação. Como linha de atuação temos o estudo sobre a vida e organismos artificiais (ADAMI, 1998; LANGTON, 1989; LEVY, 1993), e a geometria fractal (MANDELBROT, 1982; PEITGEN et al., 1993).

Esta subárea não é relevante para o entendimento deste trabalho, mas tem enorme importância no campo da computação natural.

2.1.3.3 Computação com mecanismos naturais

Enfim, Castro e Von Zuben (2004) citam a computação com mecanismos naturais, a qual trata de um paradigma onde mecanismos naturais, como o DNA, são utilizados como estruturas de dados para o desenvolvimento de "computadores naturais".

Por décadas vemos mecanismos computacionais que tem seu desenvolvimento baseado na Lei de Moore que, em 1954, observou a existência de um crescimento exponencial na quantidade de transistores que são colocados em um circuito integrado. De acordo com Moore, o número de transistores em um chip tende a dobrar a cada par de anos ou ano e meio. Assim, até o final da segunda década deste século, a tecnologia baseada no silício terá atingido seu limite.

Neste momento, surgem as técnicas que fogem ao material silício. Esses meios alternativos possuem duas abordagens: as baseadas em biomoléculas e as baseadas em bits quânticos. Estas abordagens tem como objetivo, revolucionar a computação como um todo, mas ainda estão em fase inicial de desenvolvimento.

2.2 Pesquisa Operacional

A Pesquisa Operacional nasceu durante a Segunda Guerra Mundial, quando os aliados se viram confrontados com problemas (de natureza logística e de tática e estratégia militar) de grande dimensão e complexidade (CAMPOS, 1998).

Segundo Medeiros (1998), pesquisa Operacional é um método científico de tomada de decisões. Trata-se de um estudo voltado para a resolução de problemas reais, em que se procura trazer para o campo da tomada de decisões (sobre a concepção, o planejamento ou a operação de sistemas) a atitude e os métodos próprios de outras áreas científicas.

Dentro desta área, encontra-se a Otimização Combinatória que se baseia em estudos

matemáticos para encontrar um agrupamento ótimo de objetos. Em Otimização Combinatória encontram-se vários algoritmos com desempenhos satisfatórios.

Para Vieira (1999), dentre os problemas clássicos da Otimização Combinatória estão o Problema de Empacotamento *(Bin-Packing-Problem)* e o Problema de Cortes que foram exaustivamente estudados por serem problemas de difícil resolução. Nesses problemas, a possibilidade de existir algoritmos exatos que os resolvam em tempo de execução razoável é muito pequena. Com isso, tornou-se comum o uso de algoritmos aproximados que buscam encontrar soluções muito próximas da ótima em tempos de execução razoáveis, conhecidos como Métodos Heurísticos.

2.2.1 Problema de Empacotamento

Problemas de Empacotamento (*bin packing problem)* constituem um grupo de problemas de otimização combinatória que vêm sendo intensamente estudados há várias décadas, atraindo teóricos e práticos de várias áreas. O grande interesse de estudar essa área deve-se, principalmente, à estrutura simples do problema, o que permite, além da exploração de diversas propriedades do mesmo, a sua resolução como parte de problemas mais complexos. Do ponto de vista prático, estes problemas aparecem em diversas aplicações reais, incluindo, entre outros, carregamento de veículos (EILON e CHRISTOFIDES, 1971; GEHRING et al., 1990) e corte e empacotamento (DYCKHOFF, 1990; MARQUES e ARENALES, 2007; HOTO et al., 2007).

2.2.2 Problema de Empacotamento Unidimensional

Neste trabalho abordaremos o problema de empacotamento unidimensional que consiste do processo de cortar peças maiores, disponíveis em estoque, para a produção de peças menores (MARTELLO & TOTH, 1990). Estes problemas são aparentemente simples, com grande aplicabilidade prática, porém são problemas NP-difíceis. Dentre suas várias aplicações práticas podemos citar o corte de barras de aço e o de bobinas de papel.

Esse problema tem como objetivo reduzir o número de barras usadas para o

acomodamento(corte) de uma lista de cortes predefinida, considerando apenas a dimensão.

Podemos definir três instâncias do problema: a capacidade da barra completa, uma lista de objetos e um número de barras. Assim, o problema consiste em alocar os cortes no menor número de barras possível, sem que a capacidade máxima das barras seja ultrapassado.

De acordo com Johnson (1979), este problema é NP-Completo, o que significa que é pouco provável que exista até mesmo um algoritmo de otimização, para ele, em tempo pseudo-polinomial.

Neste trabalho, foi usado um Algoritmo Genético e o Simulated Annealing partindo de uma solução considerada boa, resultante do algoritmo *Best Fit Decreasing* (BFD).

2.3 Algoritmos de Aproximação para o Problema de Empacotamento Unidimensional (PEU)

Podemos dividir os algoritmos de aproximação para o Problema de Empacotamento Unidimensional (PEU) em duas classes: *on-line* e *off-line*.

Os algoritmos *on-line* não possuem nenhuma alteração ou ordenação na lista inicial do problema. Já os algoritmos *off-line*, ordenam os itens na instância inicial.

Nesta seção serão apresentados os algoritmos *off-line* e o *Best Fit Decreasing*, derivado *Best Fit*. Apenas este último será utilizado neste trabalho, para gerar soluções iniciais boas e, a partir daí, otimiza-las com o Algoritmo Genético e o *Simulated Annealing.*

2.3.1 Algoritmo *Best Fit* (BF)

De acordo com Bernardi (2001), algoritmo *Best Fit* procura uma sublista L_i tal que $w(L_i) + w(p_j) \leq C$ e $w(L_i)$ é o máximo entre as sublistas L_i da partição corrente, onde $w(L_i)$ é o espaço já ocupado da barra, $w(p_j)$ é o corte a ser inserido na barra e C é capacidade total da barra. Caso

haja duas sublistas da partição onde as duas condições sejam satisfeitas, é escolhida a sublista de menor índice.

O algoritmo abaixo, retirado do artigo de Bernardi(2001), ilustra a execução do modelo:

```
inicio do algoritmo
        Part ← ∅
        L1← ∅
        i ← 1
        para j ← 1 até n faça
                k ← 1
                enquanto ( w(Lk) + w(pj) > C) e (k < i) faça
                        k ← k + 1
                fim enquanto
                se k > i
                então           // Começa a agrupar os itens em uma nova sublista
                        i ← i + 1
                        Li← ∅
                        Part ← Part ∪ Li
                fim se
                Li ← Li | | pj
        fim para
        Part ← Li
        retorna Part
fim do algoritmo
```

2.3.2 Algoritmo *Best Fit Decreasing* (BFD)

O algoritmo BFD é derivado do algoritmo BF, com a diferença que o BFD executa, primeiramente uma ordenação decrescente na lista L antes de iniciar o empacotamento propriamente dito (BERNARDI, 2001). Optamos pelo BFD, pois inserir cortes sem uma ordem de tamanho geraria uma desvantagem considerando o nosso problema. É mais fácil alocar um corte pequeno em qualquer lugar do que um corte grande, por isso inserir peças começando das maiores e partindo para as menores seria melhor e o BFD nos possibilita isso.

O pseudocódigo abaixo, retirado do artigo de Bernardi(2001), descreve o funcionamento deste algoritmo:

inicio do algoritmo
    -Faz uma ordenação decrescente da lista L.
    -Aplica o algoritmo BF à nova lista.
    -Retorna a solução encontrada
fim do algoritmo

2.4 Algoritmos Genéticos

Nesta seção será apresentado um breve histórico dos algoritmos genéticos, seus conceitos, implementação e o uso destes para solução do problema proposto por este trabalho.

2.4.1 Histórico

Problemas de elevado nível computacional tem sido um grande desafio para pesquisadores de diversas áreas, principalmente na pesquisa operacional, otimização, matemática e engenharias. Frequentemente nos deparamos com problemas altamente complexos, onde a solução ótima é alcançada somente em pequenas instâncias ou nunca alcançada.

Métodos tradicionais de otimização são caracterizados pela rigidez de seus modelos matemáticos representados por teoremas. Essa rigidez foi, com o tempo, sendo reduzida com o uso de técnicas de Inteligência Artificial, principalmente com ferramentas de busca heurística.

Mas as heurísticas isoladas também possuem limitações. Principalmente pela falta de base teórica dos métodos heurísticos, que desenvolvem algoritmos muito especializados. Assim a sua flexibilidade para a adaptar um programa a um problema diferente é afetada.

A reunião dos modelos rígidos da otimização com os métodos flexíveis da busca heurística permitiu o surgimento dos chamados Métodos Inteligentemente Flexíveis, que buscam desenvolver técnicas com alguma rigidez matemática e com facilidades em incorporar novas situações, sem chegar a uma flexibilidade excessiva que, às vezes, é encontrada em métodos heurísticos.

A década de 80 foi marcada com o surgimento de novos métodos heurísticos carregando novas ferramentas buscando superar as limitações das heurísticas convencionais. Dentre as diversas técnicas produzidas para tentar reduzir o risco de paradas prematuras, destacamos a Programação Evolutiva incluindo: os Algoritmos Genéticos (AGs) inicialmente propostos por Holland (1975), o Scatter Search (SS), proposto por Glover (1977), a Programação Genética, proposta por Koza (1992), e a Programação Evolutiva proposta por Fogel (1966).

Embora com bases distintas, estas metaheurísticas possuem características em comum, que as distinguem das heurísticas convencionais. Temos como exemplo os Algoritmos Genéticos, que têm se destacado na solução de muitos problemas devido a sua simplicidade e sua facilidade em resolver problemas sem exigir muito conhecimento destes.

Os Algoritmos Genéticos (AGs) são algoritmos probabilísticos e, inicialmente, foram propostos pelo Professor John Holland (1975) da Universidade de Michigan, mas somente a partir dos anos 80, é que realmente começaram a se popularizar.

A ideia inicial de Holland (1975) foi tentar imitar algumas etapas do processo de evolução natural das espécies fundindo-as a um algoritmo computacional.

O ponto de referência foi gerar, a partir de uma população de cromossomos, novos cromossomos com propriedades genéticas superiores às de seus antecedentes. Esta ideia foi então associada a soluções de um problema onde, a partir de um conjunto de soluções atuais, são geradas novas soluções superiores às antecedentes, sob algum critério pré-estabelecido.

2.4.2 Características

De acordo com Luger (2013), os Algoritmos Genéticos (AGs) são algoritmos de otimização, que tem base em ferramentas de seleção natural e de genética. Eles tem como estratégia a busca paralela e estruturada, mas aleatória, direcionada ao reforço da busca de pontos de "boa aptidão", isto é, pontos em que a função a ser minimizada (ou maximizada) tem valores baixos (ou altos).

Ainda que aleatórios, eles não são caminhadas aleatórias e não direcionadas, pois trabalham com informação histórica para encontrar novos pontos de busca onde são esperados melhores desempenhos. Isto é feito através de processos iterativos, onde cada iteração é chamada de geração.

Durante cada iteração, os princípios de seleção e reprodução são aplicados a uma população de candidatos que pode variar, dependendo da complexidade do problema e dos recursos computacionais disponíveis. Através da seleção, determina-se quais os indivíduos que conseguirão reproduzir-se, gerando um número determinado de descendentes para a próxima geração, com uma probabilidade determinada pelo índice de aptidão. Por outras palavras, os indivíduos com maior adaptação relativa têm maiores probabilidades de reproduzir-se.

O ponto de partida para a utilização de Algoritmos Genéticos, como ferramenta para solução de problemas, é a representação destes problemas de maneira que os AGs possam trabalhar adequadamente sobre eles.

O princípio básico do funcionamento dos AGs é que um critério de seleção vai fazer com que, depois de muitas gerações, o conjunto inicial de indivíduos gere indivíduos mais aptos. A maioria dos métodos de seleção são projetados para escolher preferencialmente indivíduos com maiores índices de aptidão, embora não exclusivamente, a fim de manter a diversidade da população (LUGER, 2013).

2.4.3 Operadores Genéticos

Os operadores são necessários para que, dada uma população, se consiga gerar populações sucessivas e que melhorem sua aptidão com o tempo. São eles: a criação da população inicial, o cruzamento (*crossover*), que combina soluções existentes em novas soluções; a mutação, que tem como objetivo manter a diversidade genética da população e a seleção, onde certos indivíduos são escolhidos em uma população baseado na sua aptidão.

2.4.3.1 População inicial

Um algoritmo genético começa com uma grande variedade de indivíduos gerados aleatoriamente. As funções são desenvolvidas pelo desenvolvedor do algoritmo e podem incluir uma enorme variedade de funções aritméticas e operadores condicionais.

Neste trabalho, a população inicial será definida a partir da saída do algoritmo *Best Fit Decreasing* (BFD).

2.4.3.2 Cruzamento (crossover)

De acordo com Koza (1992), o cruzamento permite a criação de novos indivíduos, o que permite novas soluções serem criadas. Mesmo a operação de reprodução agindo em apenas um indivíduo, a operação de cruzamento começa com dois pais (strings). Estes pais são selecionados de acordo com sua aptidão e produz dois filhos (strings). Estes filhos são, geralmente, diferentes entre si e de seus dois pais. Cada prole contém material genético de seus pais.

Para ilustrar a operação de *crossover*, considere a tabela 2.1.

A operação começa selecionando aleatoriamente um número entre 1 e L – 1, usando uma distribuição probabilística uniforme. Existem L – 1 = 2 pontos de cruzamento em uma string de tamalho L = 3.

Suponha que o 2 seja o ponto selecionado. Este ponto se torna o ponto de cruzamento. Cada pai é, então, dividido neste ponto de cruzamento em um fragmento e o restante.

O fragmento do cruzamento dos pais 1 e 2 são descritos na tabela 2.2.

Pai 1	Pai 2
011	110

Tabela 2.1 - Dois pais selecionados proporcionalmente à aptidão

Fragmento 1	Fragmento 2
01-	11-

Tabela 2.2 - Fragmento do cruzamento dos dois pais

Depois que o fragmento é identificado, algo sobra de cada pai. Esta sobra é descrita na tabela 2.3.

Quando o restante do pai 1 é combinado com o fragmento do cruzamento do pai 2, resultamos no filho 1. Logo após, o restante do pai 2 é combinado com o fragmento do cruzamento

do pai 1, resultando no filho 2. Na tabela 2.4 são mostrados os filhos gerados pela operação de *crossover*.

Sobra 1	Sobra 2
--1	--0

Tabela 2.3 - Sobra genética dos dois pais

Filho 1	Filho 2
111	010

Tabela 2.4 - Dois filhos gerados pelo cruzamento

2.4.3.3 Mutação

Outro operador usado em algoritmos genéticos é a mutação. Neste, um único pai (string) é selecionado de acordo com a sua aptidão. Um ponto de mutação é aleatoriamente selecionado, é descartado o código genético a partir do ponto selecionado e um novo código é gerado usando o mesmo método de criação aleatória que foi usada para a população inicial.

Neste trabalho, não será aplicado este operador pois geraria muitos soluções inviáveis e, como será tratado no Capítulo Implementação, não são plausíveis de uso.

2.4.4 Implementação

Abaixo é mostrado um pseudocódigo de um AG clássico:

```
Procedimento AG{
        t =0;
        inicia_população (P, t);
        calcula_aptidão(P, t);
```

```
        repita até (t = d)
                {I= I +1;
                seleção_dos_pais (P, t);
                recombinação (P, t);
                mutação (P, t):
                calcula_aptidão(P, t);
                sobrevivem (P, t);
        }
}
```

Onde:

t - tempo atual;

d - tempo determinado para finalizar o algoritmo;

P – população;

2.5 *Simulated Annealing* (SA)

Neste capítulo será abordado o algoritmo *Simulated Annealing* (recozimento simulado), com um breve histórico, conceitos e a sua implementação no problema proposto.

2.5.1 Histórico

De acordo com Soeiro (2000), o *Simulated Annealing* (SA) tem sua origem no processo físico do resfriamento de um metal em estado de fusão e o problema de otimização. Tendo como base, ideias da mecânica estatística e no algoritmo proposto por Metropolis(1953), onde AS foi apresentado como uma técnica de otimização combinatória e utilizado no projeto de sistemas eletrônicos.

2.5.2 Características

O algoritmo *Simulated Annealing* pode ser descrito como um procedimento iterativo composto por dois *loops* (laços de repetição) relacionados. O laço interno simula a realização do equilíbrio térmico em uma temperatura dada, então, ele será tratado como o laço de equilíbrio térmico. (BANCHS, 1997).

O laço exterior realiza o processo de resfriamento, no qual a temperatura é reduzida a partir de seu valor inicial, tendendo a zero até o certo critério de convergência ser atingido e a busca ser parada. Este *loop* será tratado como laço de resfriamento.

2.5.3 O laço de equilíbrio térmico

A operação do laço de equilíbrio começa com um modelo inicial. A cada iteração do laço, um novo modelo será ou não aceito, dependendo de certa probabilidade. Três elementos devem ser identificados, a perturbação, o critério de aceitação e o equilíbrio térmico.

2.5.3.1 Perturbação

A perturbação define a forma o qual o modelo é atualizado. Primeiramente, ela é calculada e então adicionada ao modelo existente que chamamos de $\overline{x_i}$ para obter um novo modelo que chamamos de $\overline{x_u}$. A maneira mais simples realizar esse cálculo é usar uma distribuição uniforme em um conjunto viável, no entanto, desta forma, é necessário um processo de resfriamento muito lento.

A Figura 2.1 ilustra a variação da disposição das moléculas em um meio hipotético, onde cada estado representa uma variação de energia do sistema em cada momento. Pode-se perceber que, a cada momento ou perturbação, as moléculas se dispõem de maneira diferente e, consequentemente, o modelo assume uma energia nova. Geralmente, conforme o sistema vai resfriando há uma tendência das moléculas ficarem mais estáveis. Contudo, alguns configurações

de energia mais alta podem ser mais prováveis conforme o tipo de moléculas no sistema. Existe, para cada estado uma probabilidade de estar em um determinado nível energético. Dado um conjunto de estados E_1, E_2, ..., E_j,...E_m, dizemos que um estado E_j é mais estável se a probabilidade $P_j > P_i$, para qualquer $i \neq j$.

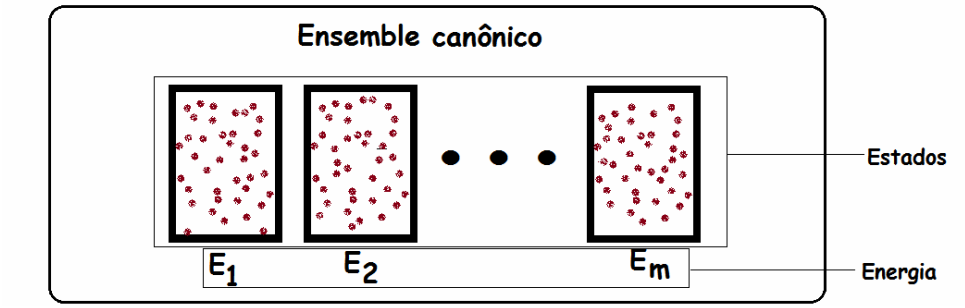

Figura 2.1 - Exemplo da variação da disposição das moléculas em um meio

2.5.3.2 Critério de aceitação

Este elemento determina se o novo modelo será aceito ou descartado. O critério de aceitação mais popular e comum foi proposto por Metropolis (1953). No algoritmo de Metropolis, uma 'variação energética' ΔE é calculada subtraindo a energia da função no modelo inicial com a energia no modelo atual:

$$\Delta E = E(\overline{x_u}) - E(\bar{x}_i)$$

Onde $E(\bar{x})$ é o energia avaliada no modelo $\bar{x}$. Então, se $\Delta E < 0$ o modelo atual $\overline{x_u}$ é aceito, pois a energia atual será menor que a inicial, caracterizando uma condição mais estável das moléculas. Mas, se $\Delta E \geq 0$, o modelo atual é aceito de acordo com a probabilidade:

$$P(\Delta E) = \exp\left(-\frac{\Delta E}{T}\right)$$

onde T é a temperatura. Por outro lado, se o modelo não é aceito, a nova iteração procederá com o mesmo modelo inicial $\bar{x}_i$.

2.5.3.3 O equilíbrio térmico

A cada iteração do laço de equilíbrio térmico uma nova perturbação é calculada. Assim que o modelo é aceito ou rejeitado de acordo com o critério de aceitação, uma nova iteração é iniciada. Este processo é repedido diversas vezes até um certo “equilíbrio térmico” ser alcançado. Neste ponto o laço é encerrado.

Existem diversas estratégias para se calcular e chegar ao equilíbrio térmico. Neste trabalho usamos uma das mais comuns, que tem como base o número de sucessos ou de fracassos.

2.5.4 Laço de resfriamento

O laço de resfriamento começa com um modelo inicial selecionado aleatoriamente e uma temperatura inicial. Em cada iteração, a temperatura é progressivamente reduzida, tendendo a zero, até atingir certo critério de convergência.

2.5.4.1 Temperatura inicial

A temperatura inicial é de extrema importância para o sucesso do algoritmo. Um valor muito baixo pode resultar na baixa abrangência da pesquisa por resultados melhores, afetando a eficiência do algoritmo. Por outro lado, uma temperatura inicial muito alta pode ocasionar ‘buscas aleatórias’ com resultados não relevantes em um número muito grande de interações. Isso terá um alto custo computacional e não terá um resultado satisfatório, principalmente quando o número de iterações é limitado.

Na prática, é complicado definir uma temperatura inicial ideal. Uma forma muito utilizada é calcular os valores de uma função objetivo a partir de modelos selecionados aleatoriamente. Então as variações energéticas são calculadas e o valor da temperatura inicial é estimada de acordo com o critério de aceitação.

2.5.4.2 Resfriamento

A forma que a temperatura é decrementada é crucial para a eficiência da busca. Se o resfriamento for muito lento, o número de iterações será muito alto e, em um algoritmo com

iterações limitadas, pode apresentar buscas sem sucesso. Já se o resfriamento for muito rápido, o algoritmo pode apresentar resultados não eficientes e até piores que a solução inicial.

2.5.4.3 Critério de parada

São inúmeros os critérios de parada que podem sem aplicados no Simulated Annealing, mas como o algoritmo implementado neste trabalho começa com resultados considerados 'bons', pode-se usar o mais comum de todos, o qual se baseia no número de fracassos ou sucessos.

Nesta monografia foram usados os seguintes parâmetros: enquanto o número de sucessos for menor ou igual a 10 vezes o número de barras usadas, o laço é interrompido. Ou, se o número de fracassos menor ou igual a 100 vezes o número de barras usadas, o laço é interrompido.

2.5.5 Implementação

Abaixo, é mostrado um pseudocódigo do Algoritmo *Simulated Annealing:*

procedimento SA (f (.), N (.), α, SAmax, T0, s)

s* ←s {Melhor solução obtida até então}

IterT← 0 {Número de iterações na temperatura T}

T ←T_0 {temperatura corrente}

enquanto (T > 0.0001)

enquanto (IterT < SAmax) **faça**

IterT← IterT+ 1

Gerar um vizinho (s') aleatoriamente na vizinhança N^K(s)

Δ = f (s') – f(s)

se (Δ < 0) **então**

s←s'

se (f(s') < f(s*)) **então** s* ← s'

fim-se

senão

Tome x ∈ [0,1]

se (x < e-Δ/T) **então**

s = s’

fim-senão

fim-se

fim-enquanto

T = T x α

IterT = 0

fim-enquanto

retorne s*

fim-procedimento

3 Implementação

Esse capítulo apresenta uma descrição dos aspectos de implementação dos algoritmos apresentados nesse trabalho. Foram usados o algoritmo BFD acompanhado do Algoritmo Genético e no segundo momento o BFD acompanhado do algoritmo *Simmulated Annealing*, buscando a otimização do resultado do Problema de Empacotamento Unidimensional.

3.1 Ambiente de desenvolvimento

Os algoritmos foram implementados da linguagem C, e compilados utilizando o GCC no sistema operacional Windows 8. Todos os testes apresentados neste trabalho foram realizados em um computador com processador Core i7 de 1.73 GHz e 4 GB de memória RAM.

3.2 Implementação dos métodos

3.2.1 BFD (*Best Fit Decreasing*)

Inicialmente foi implementado o algoritmo BFD para que tanto o AG quanto o SA começassem de um ponto de partida bom, pois ele é considerado clássico e é utilizado até hoje como ponto de partida no desenvolvimento de novos algoritmos. A versão apresentada aqui tem algumas particularidades. Uma delas é a de possuir uma entrada de dados extra contento barras usadas que podem ser adicionadas a solução antes de executa-la. O tamanho máximo de uma barra que poderá ser usada nesta implementação é de 6000mm.

L1: Lista encadeada contendo a solução final (inicialmente contendo apenas um nó cabeça apontando para NULL).

L2: Lista encadeada contendo os cortes que farão parte da solução (em ordem decrescente).

L3: Lista encadeada contendo as barras usadas que farão parte da solução (em ordem crescente).

Nas listas L2 e L3 cada nó possui espaço para guardar corte, pai1, pai2, aproveitamento1, aproveitamento2 e um ponteiro apontando para o próximo nó. Em algumas situações espaços poderão nem serem usados, foi feito assim para reaproveitamento da estrutura.

Na lista L1 cada nó possui espaço para guardar id da barra, corte, quanto resta para chegar ao tamanho máximo (6000mm), tamanho inicial da barra, ponteiro apontando para novo corte (se existir) e ponteiro apontando nova barra (se existir). Ficará mais claro adiante.

A FUNÇÃO_BUSCA verifica se há barra que caiba o novo corte em L1, se há, ele guarda essa posição apontando para barra (ApontamentoL1).

A entrada foi definida consultando uma empresa que trabalha no ramo de cortes de barras de ferro da cidade de Dourados. Com isso podemos ter um resultado final baseando-se na prática do dia-a-dia.

Abaixo encontra-se o pseudocódigo do BFD modificado.

Enquanto (corte != NULL)
Se (Nenhum corte foi feito ainda)
Se (existem barras usadas em L3 que caibam o corte)
Então remove barra usada de L3 e adiciona em L1;
return;
Senao Insere novo corte em L1 usando barra nova;
return;
Se (tem alguma barra usada em L3 que caiba o novo corte)
Então guardo essa informação em forma de apontamento (ApontamentoL3!=NULL);
FUNÇÃO_BUSCA;
Se (a posição retornada da busca não couber o corte)
Então: **Se** (ApontamentoL3 != NULL)
Então Remove barra usada de L3 e adiciona em L1 com o novo corte
return;
Senão Insere novo corte no fim de L1 usando barra nova

return;

Se (ApontamentoL3 == NULL)

Então Insire novo corte em barra já cortada em L1

return;

Senão Se (ApontamentoL3 em L3 for maior que ApontamentoL1 em L1)

Então insiro novo corte em barra de L1

return;

Senão remove barra usada de L3 e adiciona em L1

return;

As Figuras de 3.1 a 3.6, ilustram o funcionamento do algoritmo acima:

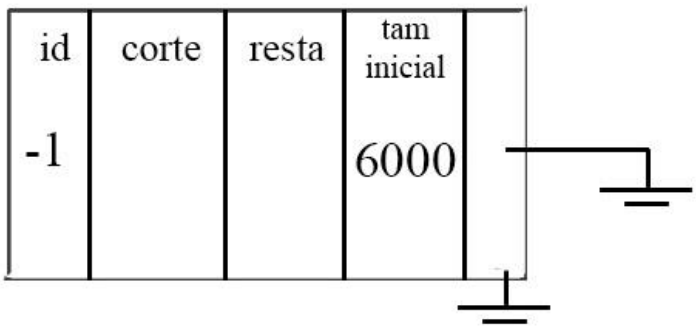

Figura 3.1 - Solução com apenas o nó cabeça

Inicialmente a solução contém apenas o nó cabeça da lista encadeada.

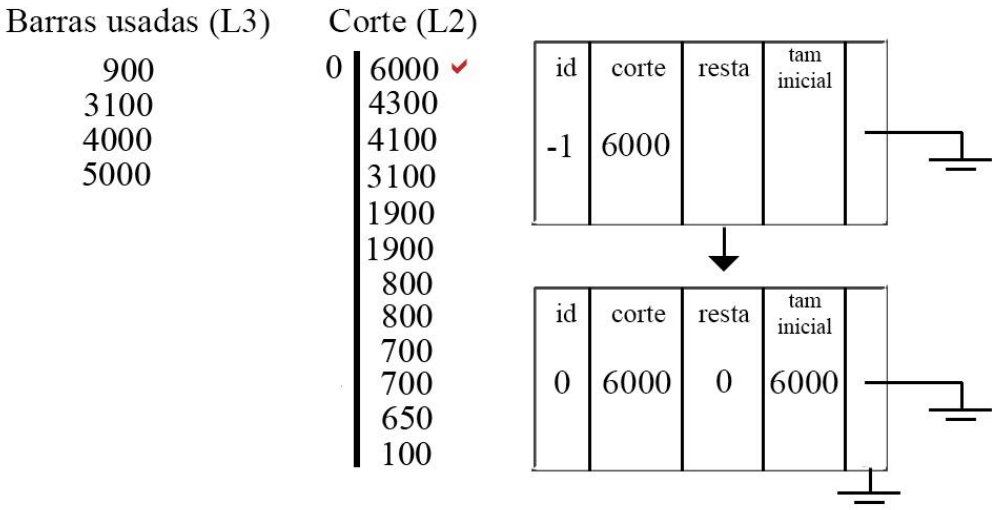

Figura 3.2 - Solução com inserção do corte de 6000

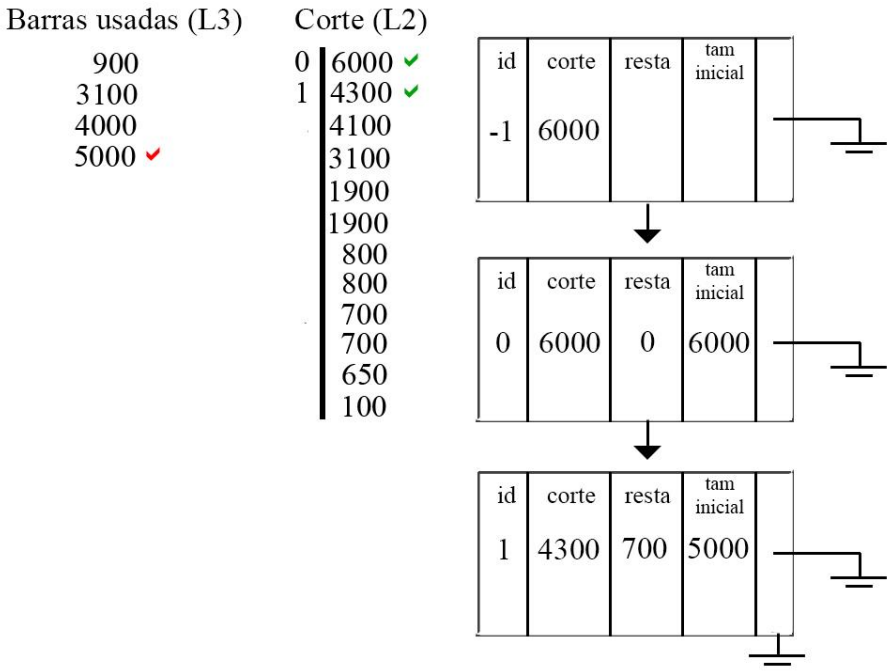

Figura 3.3 - Solução com inserção do corte de 4300

Na Figura 3.3 temos a inserção do corte 4300 em uma barra usada de 5000, onde ela é retirada da lista L3 e alocada na lista L1.

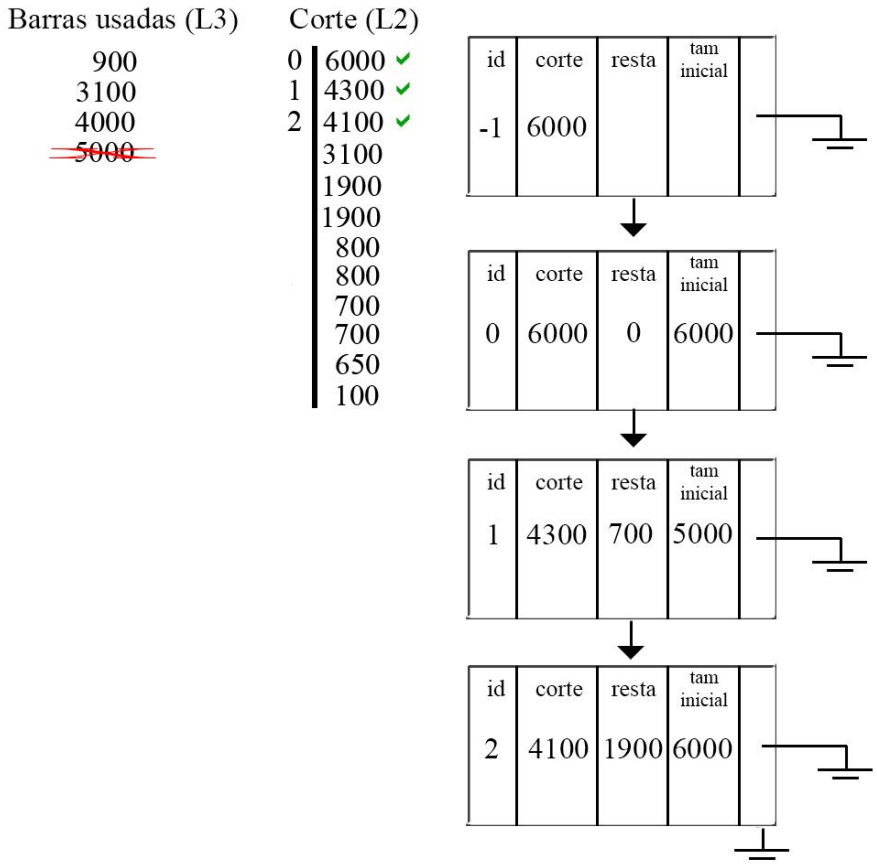

Figura 3.4 - Solução com inserção do corte 4100

Na Figura 3.4 temos a inserção do corte 4100 em uma barra nova.

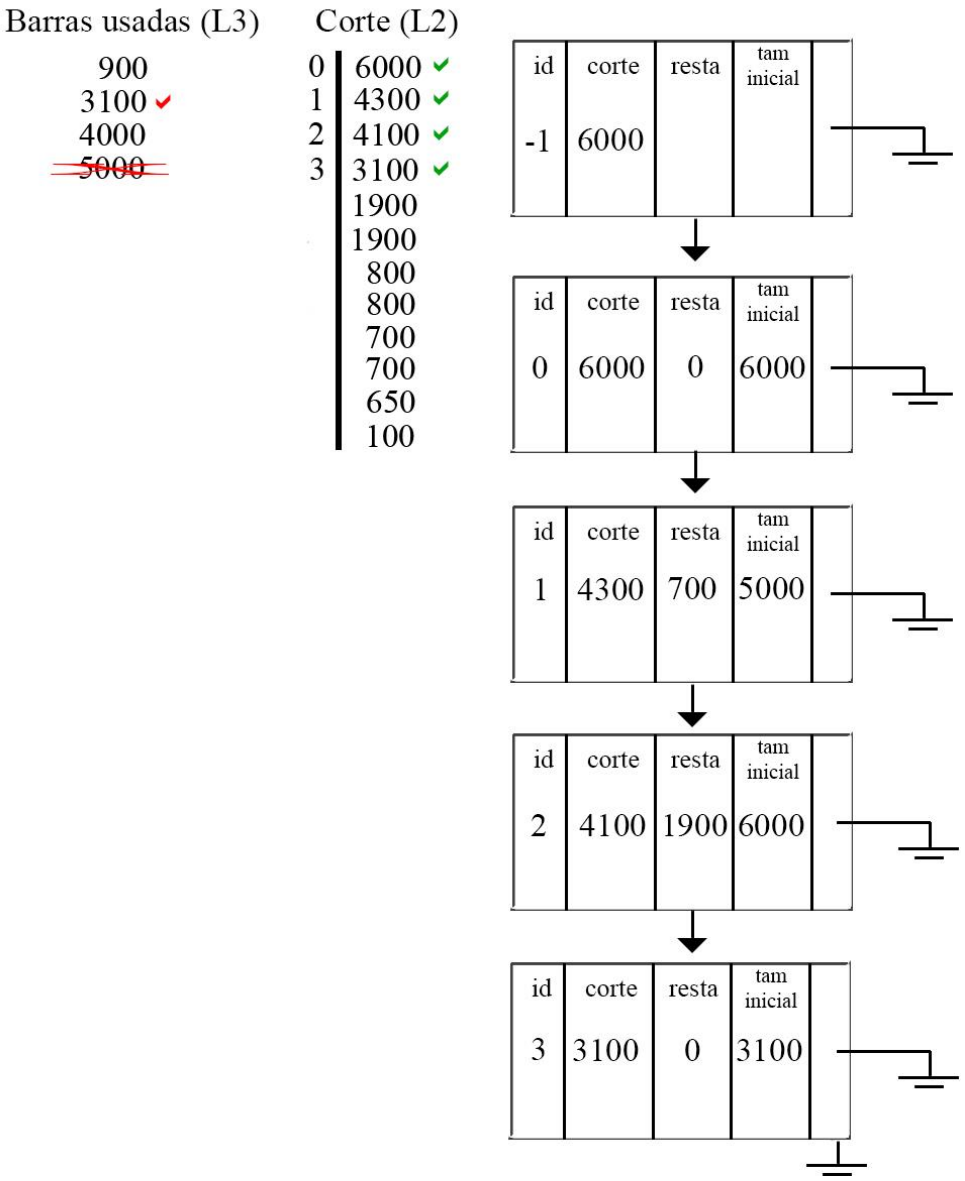

Figura 3.5 - Solução com inserção do corte de 3100

Na Figura 3.5 temos a inserção do corte 3100 em uma barra usada de 3100, onde ela é retirada da lista L3 e alocada na lista L1.

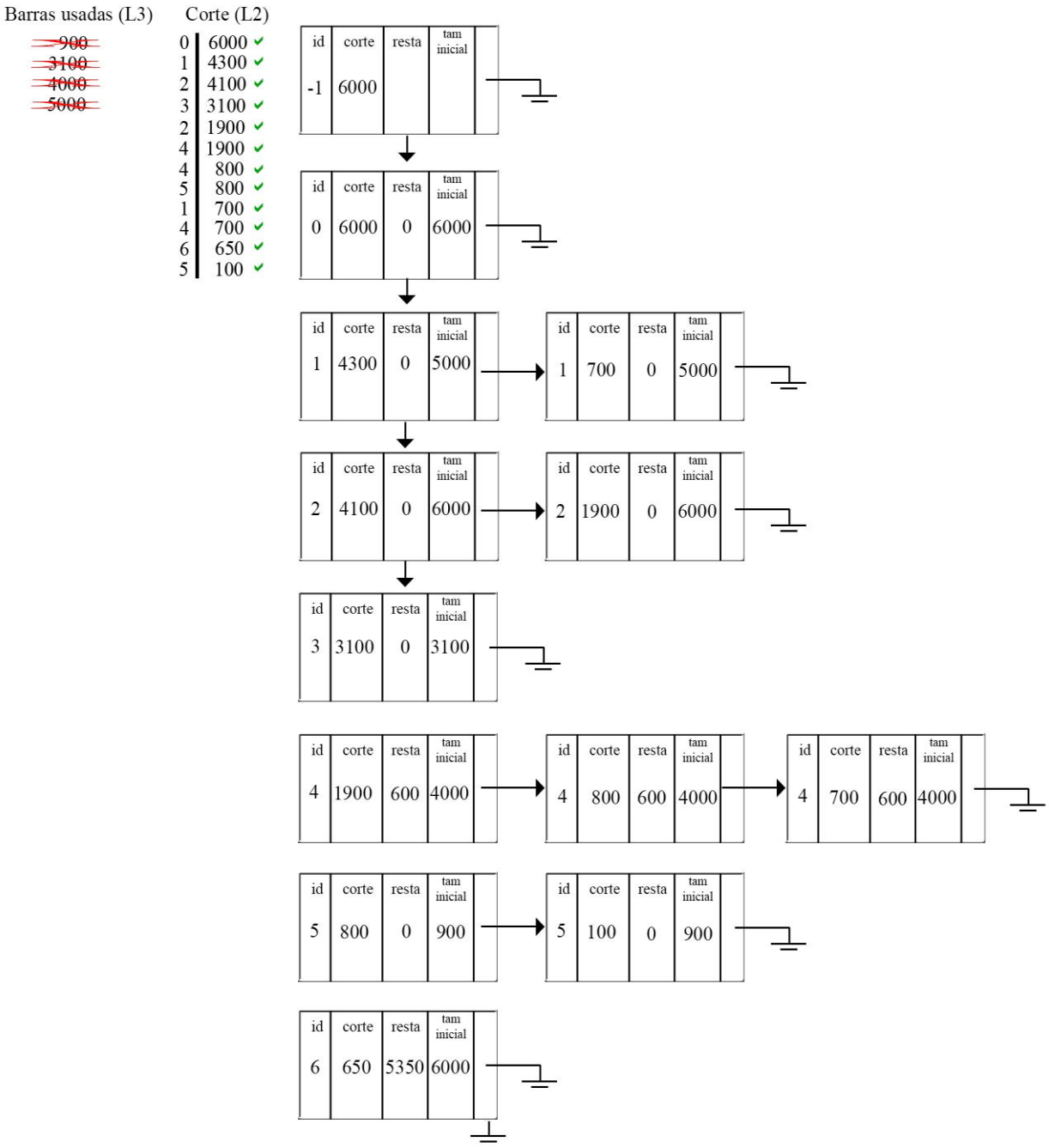

Figura 3.6 - Solução final das respectivas entradas

3.2.2 Algoritmo Genético (AG)

Um AG clássico não é suficiente para a solução do problema proposto. Algumas mudanças foram feitas.

Nosso objetivo aqui não é explicar o funcionamento de um AG e sim apresentar particularidades da nossa implementação em relação um AG comum para resolver o problema proposto. Tendo essa breve introdução à cima, apresentaremos as particularidades abaixo.

Supõe-se que existem 5 barras livres para serem usadas na solução, lembrando que o tamanho máximo de uma barra que poderá ser usada nesta implementação é de 6000mm, ilustrado na Figura 3.7:

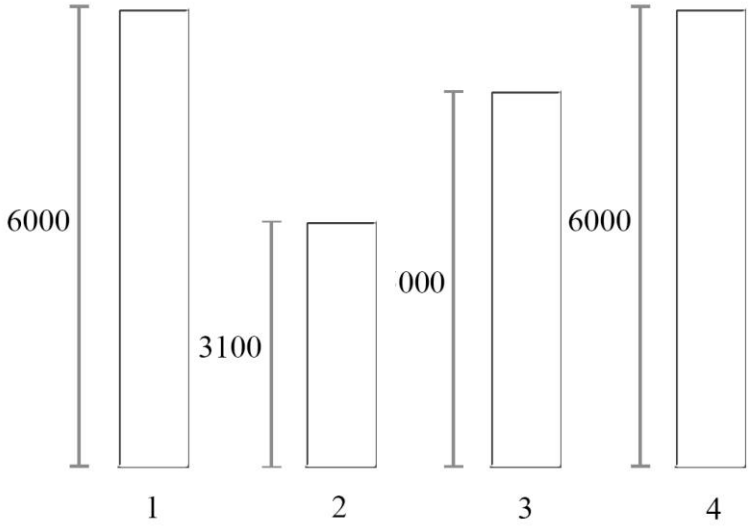

Figura 3.7 - Barras usadas no exemplo

Para o nosso exemplo usaremos os seguintes cortes, ilustrados na Figura 3.8:

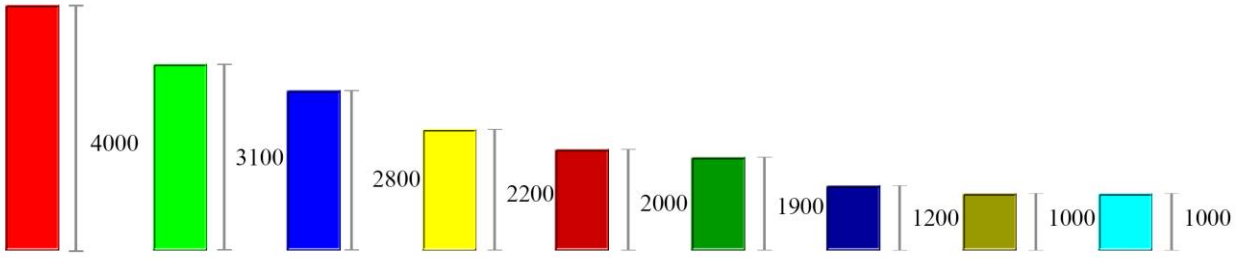

Figura 3.8 - Cortes usados no exemplo

Nossa objetivo é inserir todos os elementos nas barras que estão livres. O algoritmo BFD faz toda a primeira parte que envolve a inserção dos cortes nas barras, com isso ele gera uma cadeia de números inteiros que define em qual barra cada corte pertence. Desse modo, tem-se a seguir uma população de soluções que pode ser gerada pelo exemplo a cima:

P1 (1;2;3;4;1;3;4;4;4)

P2 (2;4;1;2;2;2;1;3;3)

P3 (1;1;2;4;4;2;4;3;3)

P4 (3;1;4;2;2;1;1;3;3)

A Figura 3.9, demonstra a distribuição dos cortes na barras de acordo com a solução P1.

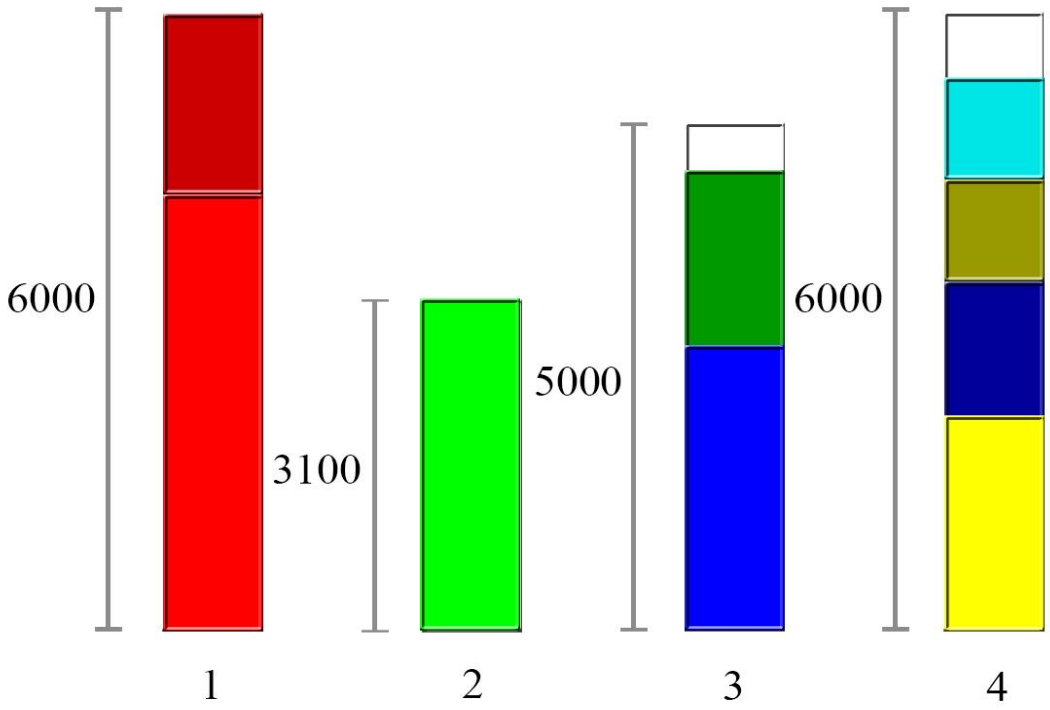

Figura 3.9 - Distribuição P1

Descrevendo:

Barra 1 possui: 4000 e 2000;

Barra 2 possui 3100;

Barra 3 possui 2800 e1900;

Barra 4 possui 2200, 1200, 1000, 1000;

Feita a disposição dos cortes nas barras, calculamos a aptidão da solução P1 que nesse caso resulta em 91,81%. Após isso é gerada uma segunda solução aleatória dentro do campo da solução de P1, isso significa que a cadeia de números correspondente a P1 sofre uma randomização criando assim uma nova cadeia de números correspondente a um P2.

O próximo passo consiste em aplicar o *crossover* em P1 e P2. O que ele faz é escolher dois pontos aleatoriamente na cadeia de números de P1 e trocar todo esse intervalo pelos referentes em P2. Após isso duas novas soluções são geradas P1' e P2'.

P1 (1;2;3;4;1;3;**1;3;3**)

P2 (3;1;4;2;2;1;**4;4;4**)

P1' (3;1;1;2;2;4;1;3;3)

P2' (3;1;4;2;2;1;4;3;3)

Um diferencial da nossa implementação é que não foram admitidas soluções inviáveis. Uma solução inviável no nosso problema significa que a cadeia de números gerada por uma solução não atende a condição básica do problema de empacotamento unidimensional, que é acomodar os cortes nas barras sem exceder a capacidade das mesmas. No nosso caso, P1' e P2' são consideradas inviáveis e com isso temos que aplicar a "função reparo" que retira os excedentes de uma barra e joga para a barra seguinte repetindo esse processo até que não tenha excedentes em nenhuma barra.

Com P1' e P2' devidamente "arrumados", o próximo passo é calcular a aptidão deles, somar, e comparar com a soma da aptidão de P1 e P2, decidindo assim qual dupla de soluções é mais eficiente para assim aceitarmos ela como a melhor solução, gerando, assim, a solução final.

3.2.3 *Simmulated Annealing* (SA)

No caso do SA, o algoritmo clássico foi suficiente para a solução do problema proposto. A

única mudança foi aplicar uma mesma “função reparo” explicada no tópico 3.2.1, justamente pelo fato de não trabalharmos no campo de soluções inviáveis.

Assim como na aplicação do AG, o algoritmo BFD faz toda a primeira parte que envolve a inserção dos cortes nas barras, gerando também uma cadeia de números inteiros que define em qual barra cada corte pertence. Desse modo, tem-se a seguir uma população de soluções iniciais:

P1 (3;1;1;2;2;4;4;3;3)

P2 (2;4;1;2;2;2;1;3;3)

P3 (1;1;2;4;4;2;4;3;3)

P4 (3;1;4;2;2;1;1;3;3)

Feita a disposição dos cortes nas barras, calculamos a aptidão da solução P1. Após isso é gerada uma segunda solução aleatória dentro do campo da solução de P1, isso significa que a cadeia de números correspondente a P1 sofre uma randomização criando assim uma nova cadeia de números correspondente a um P2.

O próximo passo consiste em aplicar as operações que o SA possui (inversão, translação e troca) em P2 e em seguida aplicar a “função reparo” torna-la viável, caso não seja. Com P2 devidamente “arrumado”, o SA segue seu processo normal calculando a aptidão de P2, fazendo a diferença entre as aptidões de P1 e P2, calculando a probabilidade de P2, fazendo o chute e aceitando melhores soluções por melhor resultado (aptidão), ou por probabilidade de acontecer até que o critério de parada seja satisfeito.

4 Resultados

Neste trabalho, era esperada uma diferença maior em relação ao rendimento do algoritmo BFD, com os métodos naturais utilizados. Porém, resultados apresentaram divergências ao que era esperado. O BFD se mostrou muito eficiente quando aplicado com o número de cortes superior a aproximadamente 100, deixando assim o AG e o AS sem eficiência. O fato de barras usadas serem reutilizadas também interferiu no rendimento geral, isso é bom pois essas barras usadas são retalhos que possivelmente iriam para o lixo em uma empresa.

O BFD apresentou um resultado quase que simultâneo com todas as entradas e por isso não tem seu tempo considerado, já o AG e o SA tem médias de tempos bem próximas, por mais que tenhas alguns picos com certas entradas.

Abaixo estão alguns gráficos que demonstram visualmente o que aconteceu nos testes.

A Figura 4.1 apresenta um gráfico mostra o rendimento dos 3 algoritmos com um número definido de cortes usando barras usadas.

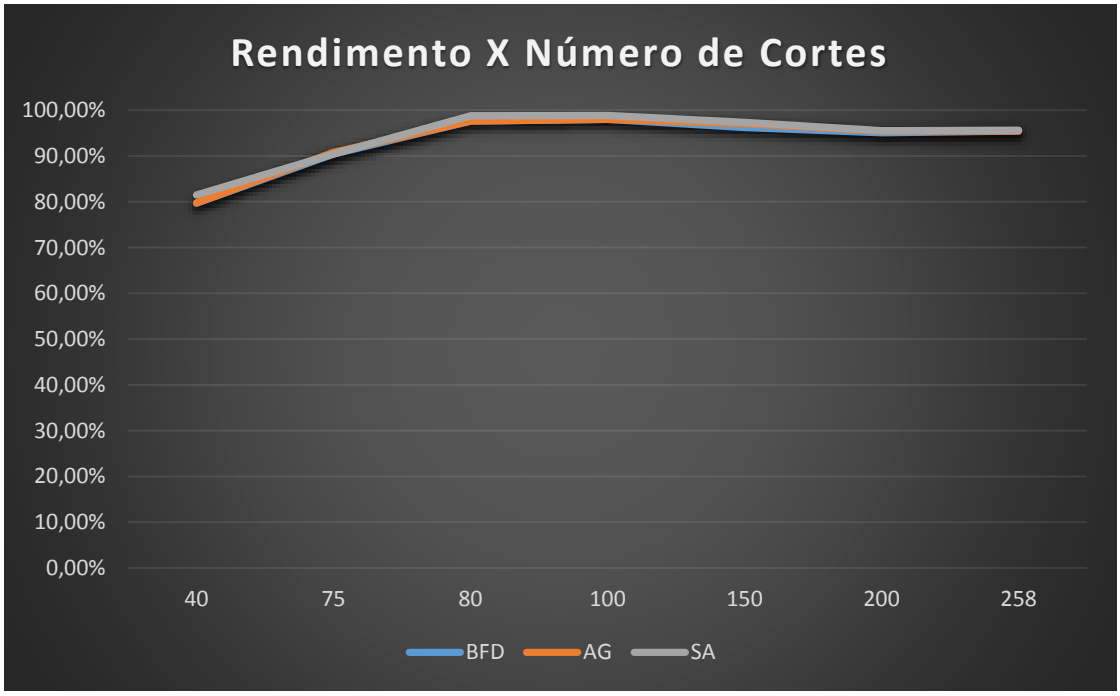

Figura 4.1 - Gráfico relacionando o Rendimento com o Número de Cortes

Na Figura 4.2, temos o gráfico demonstrando o rendimento dos 3 algoritmos com um número definidos de barras mas sem o uso de barras usadas.

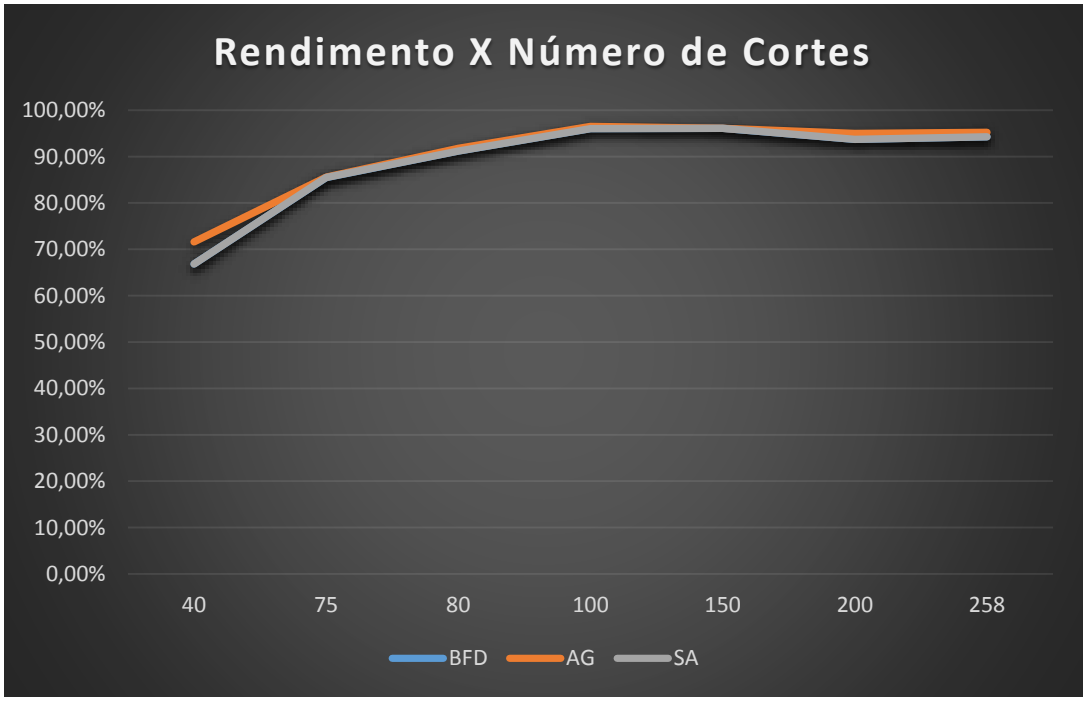

Figura 4.2 - Gráfico do rendimento dos 3 algoritmos sem o uso de barras usadas

Podemos ver que com o uso de barras usadas temos uma diferença significativa no rendimento usando menos de aproximadamente 100 cortes.

Nas figuras 4.3 e 4.4 temos os gráficos onde são demonstrados os tempos de execução em relação ao número de entradas.

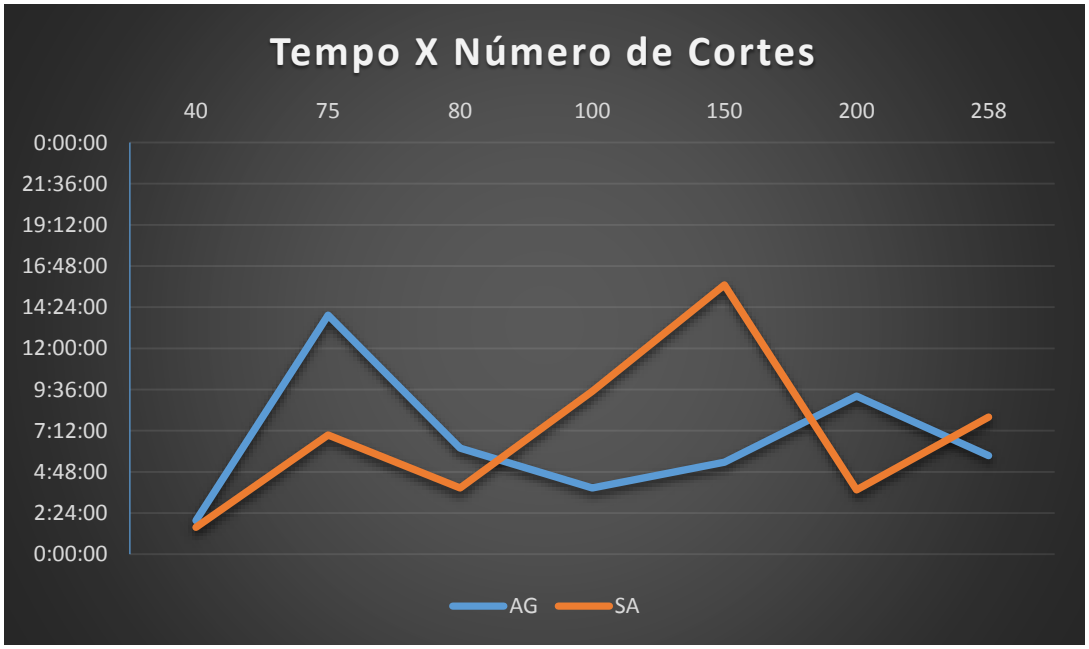

Figura 4.3 - Gráfico do rendimento dos 3 algoritmos sem o uso de barras usadas

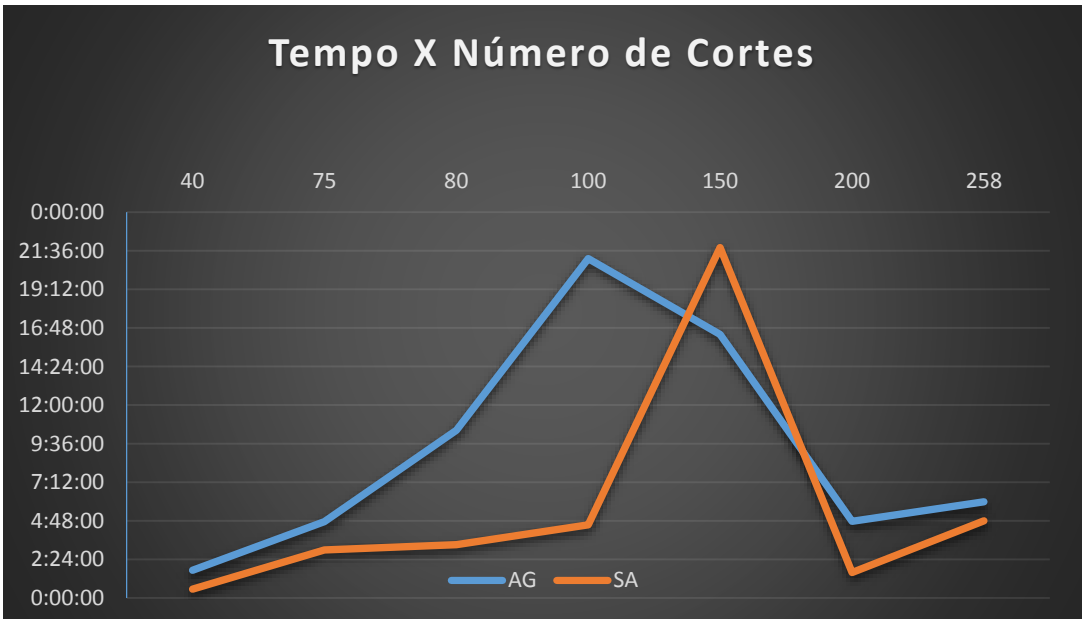

Figura 4.4 - Gráfico demonstrando os tempos de execução em relação ao número de cortes

O AG possui uma média de tempo de aproximadamente 6 minutos e 30 segundos e o SA de 7 minutos quando usamos barras usadas. Já sem barras usadas temos o AG com uma média de tempo de aproximadamente 9 minutos e o SA de 5 minutos.

5 Conclusão

Neste trabalho pudemos analisar a real capacidade dos algoritmos naturais de otimizar a solução de problemas NP-Completos.

Como podemos ver na Figura 4.1 a diferença de rendimento entre os 3 algoritmos é mínima com barras usadas. Já na Figura 4.2, quando usamos apenas barras novas, temos uma diferença significativa, mas nada extraordinário.

Na Figura 4.3 e 4.4, é exposto o tempo em relação ao número de cortes sendo que, na primeira figura usamos barras usadas e na segunda não.

Analisando os resultados, podemos concluir que o uso dos Algoritmos Genéticos e do algoritmo *Simulated Annealing* para a solução do problema de empacotamento unidimensional não é viável pois o tempo de espera por uma solução melhor que a inicial, gerada pelo algoritmo *Best Fit Decreasing*, é muito alto.

5.1 Trabalhos Futuros

Nesta seção serão apresentadas algumas possíveis melhorias que podem ser agregadas ao trabalho e estudadas mais adiante.

- Concorrer o *Simulated Annealing* com o Algoritmo Genético de forma paralela, usando threads ou apenas rodando-os de forma assíncrona. A cada iteração dos algoritmos, as saídas seriam comparadas e uma nova iteração seria realizada utilizando a melhor saída como entrada na nova iteração. Assim, poderíamos obter melhores resultados.

- Utilizar alocação estática visando melhora a velocidade do algoritmo aumentaria.

- Aumentar a população gerando mais pais aleatórios, ou partir direto do algoritmo genético sem fazer uso do BFD considerando inicialmente um corte para cada barra.

Referências Bibliográficas

ADAMI, C. **Introduction to Artificial Life.** New York, Springer Verlag, 1998.

BALLARD, D. H. **An introduction to natural computing.** Cambridge, MIT Press, 1999.

BANCHS, R. E. **Simulated Annealing**, Research progress report, on Time Harmonic Field Electric Logging Austin, University of Texas at Austin, 1997

BERNARDI, R. **Aplicando a técnica de times assíncronos na otimização de problemas de empacotamento unidimensional.** [Dissertação de mestrado], São Paulo, Escola Politécnica da Universidade de São Paulo, 2001.

BONABEAU, E.; DORIGO, M.; THERAULAZ, G. **Swarm Intelligence: From Natural to Artificial Systems.** New York, Oxford University Press, 1999

CAMPOS, D. F. **Introdução à pesquisa operacional**. Apostila / Módulo I, 1998.

CASTRO, L. N.; VON ZUBEN, F. **Recent Developments In Biologically Inspired computing.** Hershey, Idea Group Publishing, 2004.

CHRISTOFIDES, N.; EILON, S. **An algorithm for the vehicle Dispatching Problem.** Birmingham**,** Operational Research Society, 1969.

COPPIN, Ben. **Inteligência artificial**. Rio de Janeiro, RJ: LTC, 2010.

DYCKHOFF, H. **A New Linear Programming Approach to the Cutting Stock Problem.** Fernuniversität Hagen, Operations Research, 1981.

EILON, S.; CHRISTOFIDES, N. **The Loading Problem.** Denver, Management Sci, 1971.

FLAKE, G. W. **The Computational Beauty of Nature.** Cambridge, The MIT Press, 1998.

FOGEL, L.J.; OWENS, A.J.; WALSH, M.J. **Artificial Intelligence through Simulated Evolution.** New York, John Wiley, 1966.

GEHRING, H.; MENSCHNER, K.; MEYER, M. **A Computer-Based Heuristic for Packing Pooled Shipment Containers.** European Journal of Operational Research, 1990.

GLOVER, F. **Heuristics for integer programming using surrogate constraints.** Blackwell Publishing, 1977.

HAYKIN, S. **Redes Neurais Princípios e prática.** Porto Alegre, Bookman, 1999.

HOLLAND, J. H. **Adaptation in Natural and Artificial Systems.** Ann Arbor, The University of Michigan Press, 1975.

HOTO, R.; ARENALES, M. N.; MACULAN, N. **The one dimensional compartmentalized knapsack problem: A case study.** European Journal of Operational Research, 2007.

JACKSON JUNIOR, P. C. **Introduction to artificial intelligence.** New York, Dover, 1985.

JOHNSON, D. S.; GAREY, M. R. **Computers and Intractability: a guide to the theory of NP-Completeness.** New Jersey, Bell Telephone Laboratories Inc, 1979.

KARI, L.; ROZENBERG, G. **The Many Facets of Natural Computing.** London, ACM, 2008.

KENNEDY, J.; EBERHART, R.C.; SHI, Y. **Swarm intelligence.** San Francisco, Morgan Kaufmann Publishers, 2001.

KOZA, J. R. **Genetic programming: on the programming of computers by means of natural selection.** Cambridge, MIT Press, 1992.

LANGTON, C. G. **Artificial Life (Santa Fe Institute Studies in the Sciences of Complexity).** Redwood, Addison-Wesley, 1989.

LEVY, P. **As tecnologias da Inteligência: o futuro do pensamento na era da informática.** Rio de Janeiro, Editora 34, 1993.

LUGER, G. F. **Inteligência Artificial** / George F. Luger; Tradução: Danieli Vieira; Revisão técnica: Andréa I. Tavares, São Paulo, 6ª edição, Pearson Education do Brasil, 2013.

MANDELBROT, B. B. **The fractal geometry of nature.** New York: W. H. Freeman,1982.

MARQUES, F. P.; ARENALES, M. N. **The constrained compartmentalized knapsack problem.** Computers & Operations Research, 2007.

MARTELLO, S. e TOTH, P. **Knapsack Problems: Algorithms and Computer Implementations.** New York, Wiley, 1990.

METROPOLIS, N., A. ROSENBLUTH, M. ROSENBLUTH, A. Teller, E. Teller, **Equation of State Calculations by Fast Computing Machines.** Journal of Chemical Physics, 21, 1953.

PEDRYCZ, W.; GOMIDE. F. **An introduction to fuzzy set: Analysis and design.** Cambridge, MIT Press, 1998.

POLI, R.; LANGDON, W. B.; MCPHEE, N. F. **A field guide to genetic programming.** UK, Wales License, 2008.

RICH, E. **Inteligência artificial.** Sao Paulo, SP, McGraw-Hill, 1988.

RUSSELL, S. J.; NOVIG, P. **Inteligência artificial: referência completa para cursos de computacão.** Rio de Janeiro, Elsevier, 2004.

SOEIRO, F. J. C. P.; CAMPOS, H. F.; SILVA, A. J. **A Combination of Artificial Neural Networks and The Levenberg-Marquardt Method for the Solution of Inverse Heat Conduction Problems.** VI Encontro de Modelagem Computacional, Nova Friburgo, 2003.

SOEIRO, F. J. C. P.; CARVALHO, G.; SILVA NETO, **A. J. Thermal Properties Estimation of Polymeric Materials with the Simulated Annealing Method.** Porto Alegre, Proc. 8th Brazilian Congress of Engineering and Thermal Sciences, 2000.

VAN LAARHOVEN, P.; AARTS, E., **Simulated annealing: Theory and Applications.** Boston, Kluwer Academic Publishers, 1988.

VIEIRA, V.G. **Problema da Mochila.** Santa Maria, Trabalho de Graduação UFSM, 1999.

Apêndice A – Principais funções

A.1 Cruzamento (Algoritmo Genético)

```
void crossover(struct No2 **Ant2, struct No2 **Pont2, struct No2 **Ptlista2)

{

   int aux;

   int n1, n2, i,j;

   (*Ant2) = (*Ptlista2)->prox;

                    //INVERTE

                    srand(time(NULL));

                        n1=rand()% (*Ptlista2)->corte;

                        n2=rand()% (*Ptlista2)->corte;

                      if(n1<n2)

                      {

                        i=n1;

                        j=n2;

                        while(n1)

                        {

                           (*Ant2) = (*Ant2)->prox;

                           n1--;

                        }

                        while(i<=j)

                        {
```

```
            aux = (*Ant2)->p1;
           (*Ant2)->p1 = (*Ant2)->p2;
           (*Ant2)->p2 = aux;
            i++;
        }
    }else{
        i=n2;
        j=n1;
        while(n2)
        {
            (*Ant2) = (*Ant2)->prox;
            n2--;
        }
        while(i<=j)
        {
            aux = (*Ant2)->p1;
           (*Ant2)->p1 = (*Ant2)->p2;
           (*Ant2)->p2 = aux;
            i++;
        }
    }
}
```

A.2 Lógica do Algoritmo BFD

```
void InsereLista(int chave, struct No **Ant, struct No **Pont, struct No **Ptlista, struct No2 **Ant2, struct No2 **Pont2, struct No2 **Ptlista2, struct No2 **Ant3, struct No2 **Pont3, struct No2 **Ptlista3)

{

No *Pt; //ponteiro axuliar

No *Ptr;// ponteiro auxiliar

No *Pi;

No2 *Pi3; // ponteiro pra indicar qual sublista tem menos espaço

Pi3 = NULL;

*Ant = *Ptlista;

Ptr = (*Ptlista)->nova_barra;

(*Ant3) = (*Ptlista3);

(*Pont3) = (*Ptlista3)->prox;


//PRIMEIRA PARTE

      if(Ptr == NULL)

      {

       while((*Pont3) != NULL)

       {

       if(chave <= (*Pont3)->corte )

          {
```

```
        Pt = new struct No;
    if( !Pt )
        {
        cout << endl << "Erro na alocação de memória!";
        exit(0);
        }
// Atualizar ponteiros
Pt->id = ((*Ant)->id)+1;
Pt->corte = chave;
Pt->resta = (*Pont3)->corte - (Pt->corte);
Pt->tam_barra = (*Pont3)->corte;
Pt->nova_barra = NULL;
Pt->novo_corte = NULL;
(*Ant)->nova_barra = Pt;
(*Ant3)->prox = (*Pont3)->prox;
if(Pt->id > (*Ptlista2)->corte)
{
    (*Ptlista2)->corte = Pt->id;
}
(*Ant2)->p1 = Pt->id;
delete (*Pont3);
return;
```

```
    }
    (*Ant3)=(*Pont3);
    (*Pont3)=(*Pont3)->prox;
}
Pt = new struct No;
if( !Pt )
    {
        cout << endl << "Erro na alocação de memória!";
        exit(0);
    }
// Atualizar ponteiros
Pt->id = ((*Ant)->id)+1;
Pt->corte = chave;
Pt->resta = ((*Ant)->resta)-(Pt->corte);
Pt->tam_barra = barra_estoque;
Pt->nova_barra = NULL;
Pt->novo_corte = NULL;
(*Ant)->nova_barra = Pt;
if(Pt->id > (*Ptlista2)->corte)
{
    (*Ptlista2)->corte = Pt->id;
}
```

```
        (*Ant2)->p1 = Pt->id;
        return;
        }
//SEGUNDA PARTE
while(Ptr != NULL)
{
   (*Ant) = Ptr;
   Ptr = Ptr->nova_barra;
}

(*Ant3) = (*Ptlista3);
(*Pont3) = (*Ptlista3)->prox;

while((*Pont3) != NULL)
      {
         if(chave <= (*Pont3)->corte )
         {
               Pi3 = (*Pont3);
               (*Pont3) = NULL;

         }else{
         (*Ant3)=(*Pont3);
```

```
            (*Pont3)=(*Pont3)->prox;
            }
        }
//TERCEIRA PARTE
*Ant = (*Ptlista)->nova_barra;
Ptr = (*Ant)->nova_barra;

//faz a busca pra definir se cria uma sublista,
//ou insere em uma ja existente.
Busca(chave, &Ptr, &Pi, Ant, Pont, Ptlista );
//se elemento indicado por PI(ponteiro que indica onde deve ser inserido
//o elemento) nao couber a inserçao

if(Pi->resta < chave)
 {
    if(Pi3 != NULL)
    {
          Pt = new struct No;
          if( !Pt )
             {
             cout << endl << "Erro na alocação de memória!";
             exit(0);
```

```
        }
    Pt->id = ((*Pont)->id)+1;
    Pt->corte = chave;
    Pt->resta = (Pi3)->corte - (Pt->corte);
    Pt->tam_barra = (Pi3)->corte;
    Pt->nova_barra = NULL;
    Pt->novo_corte = NULL;
    (*Pont)->nova_barra = Pt;
    (*Ant3)->prox = (Pi3)->prox;
    delete (Pi3);
    if(Pt->id > (*Ptlista2)->corte)
    {
       (*Ptlista2)->corte = Pt->id;
    }
    (*Ant2)->p1 = Pt->id;
    return;
}else{
       Pt = new struct No;
       if( !Pt )
          {
          cout << endl << "Erro na alocação de memória!";
          exit(0);
```

```
            }
        // Atualizar ponteiros
        Pt->id = ((*Pont)->id)+1;
        Pt->corte = chave;
        Pt->resta = barra_estoque -(Pt->corte);
        Pt->tam_barra = barra_estoque;
        Pt->nova_barra = NULL;
        Pt->novo_corte = NULL;
        (*Pont)->nova_barra = Pt;
        if(Pt->id > (*Ptlista2)->corte)
        {
           (*Ptlista2)->corte = Pt->id;
        }
        (*Ant2)->p1 = Pt->id;
       return;
     }
}
//QUARTA PARTE
  if(Pi3 == NULL)
  {
      Ptr = Pi->novo_corte;
        Pt = new struct No;
```

```
        if( !Pt )
            {
                //cout << endl << "Erro na alocação de memória!";
                exit(0);
            }
    // Atualizar ponteiros
    Pt->id = Pi->id;
    Pt->corte = chave;
    (Pi)->resta = (Pi)->resta-(Pt->corte);
    Pt->resta = (Pi)->resta;
    Pt->nova_barra = NULL;
    Pt->novo_corte = Ptr;
    (Pi)->novo_corte = Pt;
    if(Pt->id > (*Ptlista2)->corte)
    {
       (*Ptlista2)->corte = Pt->id;
    }
    (*Ant2)->p1 = Pt->id;
    return;
}else{
    if(Pi3->corte > Pi->resta)
    {
```

```
   Pt = new struct No;
   if( !Pt )
      {
      cout << endl << "Erro na alocação de memória!";
      exit(0);
      }
// Atualizar ponteiros
Pt->id = Pi->id;
Pt->corte = chave;
(Pi)->resta = (Pi)->resta-(Pt->corte);
Pt->resta = (Pi)->resta;
Pt->nova_barra = NULL;
Pt->novo_corte = Ptr;
(Pi)->novo_corte = Pt;
if(Pt->id > (*Ptlista2)->corte)
{
   (*Ptlista2)->corte = Pt->id;
}
(*Ant2)->p1 = Pt->id;
return;
}else{
      while((*Pont)->nova_barra != NULL)
```

```
    {
       (*Pont) = (*Pont)->nova_barra;
    }
    Pt = new struct No;
    if( !Pt )
       {
       cout << endl << "Erro na alocação de memória!";
       exit(0);
       }
Pt->id = ((*Pont)->id)+1;
Pt->corte = chave;
Pt->resta = (Pi3)->corte - (Pt->corte);
Pt->tam_barra = (Pi3)->corte;
Pt->nova_barra = NULL;
Pt->novo_corte = NULL;
(*Pont)->nova_barra = Pt;
(*Ant3)->prox = (Pi3)->prox;
delete (Pi3);
if(Pt->id > (*Ptlista2)->corte)
{
   (*Ptlista2)->corte = Pt->id;
}
```

```
        (*Ant2)->p1 = Pt->id;
        return;
        }
    }
    }
```

A.3 Procedimento do Simulated Annealing

```
    void annealing (struct No2 **Ant2, struct No2 **Pont2, struct No2 **Ptlista2, struct No **Ptlista, struct No **Ant)
    {

       bool s=false;
       int k=0;
       float d,daux,dE;
       float t = 300;
       float dec=0.1,chute,Pa;
       float kb=1.38*pow(10,-23);
       int i, j, total;


       srand(time(NULL));
       total = 0;
```

```
j=1;
do{
    i=1;
    sucesso = 0;
    fracasso=0;
    do{
            aptidao1(Ant2, Pont2, Ptlista2, Ptlista, Ant);
            operacoes(Ant2, Pont2, Ptlista2);
            reparo2 (Ant2, Pont2, Ptlista2, Ptlista, Ant);
            aptidao2(Ant2, Pont2, Ptlista2, Ptlista, Ant);
            d=(*Ptlista2)->aproveitamentop1; //SO
            daux=(*Ptlista2)->aproveitamentop2; //recebe distancia aux  APTIDAO AUX SA
            dE= daux - d; //faz a diferença entre distancias  DIFERENÇAS ENTRE APTIDAO
            Pa=exp((dE)/(kb*t)); //probabilidade com a diferença das dist PROBABILIDADE COM DIFERENÇA DE APTIDAO COM FORMAULA PRE DEFINIDA
            chute = (float)(rand()/RAND_MAX);
            if(daux>d || Pa > chute)//aceitou por menor energia
              {
                p1recebe(Ant2, Pont2, Ptlista2); //original recebe aux FAZ COPIA DE P2 EM P1
```

```
                    sucesso++;
                    fracasso=0;
                  }else{
                    fracasso++;
                    sucesso=0;
                    p2recebe(Ant2, Pont2, Ptlista2);
                    }
                i++;
        }while(sucesso <=10*(*Ptlista2)->corte && fracasso <=100*(*Ptlista2)->corte);
        t=t*(1-dec);
        total++;
        j++;
    }while(total<10);
}
```

A.4 Cálculo da Aptidão

```
//calcula aptidao do "pai2"
//o procedimento aptidao1 faz o mesmo processo só que utilizando a primeira string da solução
void aptidao2(struct No2 **Ant2, struct No2 **Pont2, struct No2 **Ptlista2, struct No **Ptlista, struct No **Ant )
{
   No2 *Pt;
```

```
No2 *Ptr;
int i, somab;
float somat;
Ptr = new struct No2;
Ptr->corte = 0;
Ptr->prox = NULL;
//Cria lista PTR com valores de barras usadas e define se nova suoluçao tem menos barras que P1
for(i=(*Ptlista2)->corte;i>=0; i--)
{
   (*Ant2) = (*Ptlista2)->prox;

  while((*Ant2) != NULL)
   {
   if(i == (*Ant2)->p2)
   {
      Pt = new struct No2;
      if( !Pt )
      {
         cout << endl << "Erro na alocação de memória!";
         exit(0);
      }
```

```
    Pt->prox = Ptr->prox;
    Ptr->prox = Pt;
    Pt->corte = (*Ant2)->p2;
    Pt->p1 = 1;
    Ptr->corte = Ptr->corte +1;
    (*Ant2) = NULL;
}else{
        (*Ant2) = (*Ant2)->prox;
        if((*Ant2) == NULL)
        {
            Pt = new struct No2;
            if( !Pt )
            {
                cout << endl << "Erro na alocação de memória!";
                exit(0);
            }
            Pt->prox = Ptr->prox;
            Ptr->prox = Pt;
            Pt->corte = i;
            Pt->p1 = 0;
        }
    }
```

```
        }
    }
    Pt = Ptr->prox;
    somat=0;
    (*Ant) = (*Ptlista);

float a, aux;
    while(Pt != NULL)
        {
            (*Ant2) = (*Ptlista2)->prox;
            (*Ant) = (*Ant)->nova_barra;
            somab = 0;
            while((*Ant2) != NULL)
            {
                if(Pt->p1 == 1)
                {
                    if(((*Ant2)->p2 == Pt->corte) && (Pt->p1 == 1))
                        {
                            somab = somab + (*Ant2)->corte;
                            (*Ant2) = (*Ant2)->prox;
                        }else{
                                (*Ant2) = (*Ant2)->prox;
```

```
                }
            }else{
                (*Ant2) = NULL;
            }
        }
        if(Pt->p1 == 1)
            {
                aux = (*Ant)->tam_barra;
                a = ((somab/aux) * (somab/aux)) / Ptr->corte;
                somat = somat + a;
                Pt = Pt->prox;
            }else {
                Pt = Pt->prox;
                }
    }
(*Ptlista2)->aproveitamentop2 = somat;

//desaloca memoria
 Pt = (Ptr);
while (Pt != NULL)
{
    (Ptr) = (Pt)->prox;
```

```
            delete(Pt);
            (Pt) = (Ptr);
        }
    }
```

A.5 Função Repara

```
    //repara "pai1" deixando-o viável
// o procedimento reparo2 faz o mesmo, só que usando a segunda string da solução.
    void reparo1 (struct No2 **Ant2, struct No2 **Pont2, struct No2 **Ptlista2, struct No **Ptlista, struct No **Ant)
    {
        int i, t, cont = 0;

        *Ant = *Ptlista;
        for(i=0; i<= (*Ptlista2)->corte ;i++)//ate numm de barras usadas
        {
            t=0;
            *Ant2 = (*Ptlista2)->prox;
            *Ant = (*Ant)->nova_barra;
            while((*Ant2) != NULL)
            {

                if((*Ant2)->p1 == i)
```

```
{
   t=t+ (*Ant2)->corte;
   if(t > (*Ant)->tam_barra)
   {
      t = t - (*Ant2)->corte;
      if(((*Ant2)->p1)+1 > (*Ptlista2)->corte)
      {
         (*Ant2)->p1 = 0;
         i=-1;
         cont ++;
         if(cont > (*Ptlista2)->corte)
         {
            randomiza1(Ant2, Pont2, Ptlista2);
            cont = 0;
            i=-1;
         }
         (*Ant) = (*Ptlista);
         (*Ant2) = NULL;
      }else{
            (*Ant2)->p1 = ((*Ant2)->p1) +1;
         }
   }else{
```

```
                    (*Ant2) = (*Ant2)->prox;
                }
            }else{
                (*Ant2) = (*Ant2)->prox;
            }
        }
    }
    //repara "pai1" deixando-o viável
// o procedimento reparo2 faz o mesmo, só que usando a segunda string da solução.
    void reparo1 (struct No2 **Ant2, struct No2 **Pont2, struct No2 **Ptlista2, struct No **Ptlista, struct No **Ant)
    {
        int i, t, cont = 0;

        *Ant = *Ptlista;
        for(i=0; i<= (*Ptlista2)->corte ;i++)//ate numm de barras usadas
        {
            t=0;
            *Ant2 = (*Ptlista2)->prox;
            *Ant = (*Ant)->nova_barra;
            while((*Ant2) != NULL)
            {
```

```
if((*Ant2)->p1 == i)
{
   t=t+ (*Ant2)->corte;
   if(t > (*Ant)->tam_barra)
   {
      t = t - (*Ant2)->corte;
      if(((*Ant2)->p1)+1 > (*Ptlista2)->corte)
      {
         (*Ant2)->p1 = 0;
         i=-1;
         cont ++;
         if(cont > (*Ptlista2)->corte)
         {
            randomiza1(Ant2, Pont2, Ptlista2);
            cont = 0;
            i=-1;
         }
         (*Ant) = (*Ptlista);
         (*Ant2) = NULL;
      }else{
            (*Ant2)->p1 = ((*Ant2)->p1) +1;
```

```
                }
            }else{
                (*Ant2) = (*Ant2)->prox;
            }
        }else{
            (*Ant2) = (*Ant2)->prox;
        }
    }
}
```