SIMULATED ANNEALING COM DIVISÃO DE DOMÍNIO APLICADO AO PROBLEMA DO CAIXEIRO VIAJANTE

Giovany F. Teixeira
Universidade Federal do Espírito Santo – UFES
Rua Maria Eleonora Pereira nº 1061 Suri-401, Jardim da Penha, Vitória, ES. Cep 2906180
giovanyfrossard@gmail.com

Flávio Severiano Lamas de Souza
Universidade Federal do Espírito Santo – UFES
flavio-lamas@gmail.com

Resumo

Computar a solução ótima para uma unidade de problema do caixeiro viajante é custoso pois se trata de um dos problemas clássicos do tipo NP-Hard. Nesse artigo, descreveremos uma implementação modificada meta-heurística de *Simulated Annealing* aplicada a unidades de problema do caixeiro viajante, a qual demos o nome de *Simulated Annealing* com Divisão de Domínio. Nosso objetivo é gerar soluções aproximadas e comparar seus resultados com a solução ótima afim de determinarmos o quão boa é essa meta-heurística, para as comparações utilizaremos a heurística de *Arbitrary Insertion Procedure* e o *Simulated Annealing* tradicional.
Palavras Chave: *Simulated Annealing*, caixeiro viajante, meta-heurísticas.

Abstract

To find out the optimal solution for a simple TSP problem is extremely expensive since it is a classic NP-Hard problem. In this paper, we describe a modified implementation of the Simulated Annealing heuristic for the TSP problem that we called Divided Simulated Annealing or Simulated Annealing with Domain Division. Our target is to find out solutions using this new code and compare them with already know optimal solutions so we can measure how good is this new implementation. We will be using Arbitrary Insertion Procedure and the usual Simulated Annealing heuristics to compare with this new implementation.
Key words: Simulated Annealing, TSP, metaheuristic.

1. INTRODUÇÃO

O problema do caixeiro viajante (PCV, ou em inglês, traveler salesman problem, ou TSP) consiste em encontrar o caminho de menor custo passando por todas as cidades, em um tour, uma única vez e depois voltando a cidade inicial.

Esse problema tem sido pesquisado , tratado há muito tempo e até hoje não foram encontrados algoritmos polinomiais que realmente retornem sua resposta ótima. Tanto que os caminhos mais abordados para resolver esse problema na literatura são os algoritmos de aproximação como: Algoritmo Genético [4], Colônia de Formigas [2], *Simulated Annealing* [5].

Nesse trabalho , inicialmente faremos uma breve descrição das principais heurísticas para o problema do caixeiro viajante no capítulo 2, explicamos o que é o problema do caixeiro viajante (PCV) no capitulo 3, abordamos o Simulated Annealing e os motivos que nos levam a utilizar meta-heurísticas no capitulo 4 , no capítulo 5 discutimos a implementação do *Simulated Annealing* com Divisão de Domínio, faremos a comparação dessa implementação

com o *Simulated Annealing* tradicional [5] e com *o Arbitrary Insertion Procedure* [3] mostrando nossos resultados no capitulo 6 e relatamos nossas conclusões no capitulo 7.

2. ALGORITMOS PRINCIPAIS UTILIZADOS PARA RESOLVER O PROBLEMA DO CAIXEIRO VIAJANTE

Existem diversas técnicas utilizadas para resolver o problema do caixeiro viajante, destacaremos nesse artigo algumas das principais meta-heurísticas utilizadas para resolver esse tipo de problema.

2.1. COLÔNIA DE FORMIGAS

A meta-heurística de Colônia de Formigas foi inspirada na observação das colônias de formigas do mundo real. Um comportamento interessante dessas colônias é que as formigas, em geral, encontram o caminho mais curto ou pelo menos um bom caminho entre a fonte de comida e o seu ninho, entende-se por bom caminho um caminho próximo ao mais curto. Isso ocorre porque enquanto caminham, depositam sobre a superfície uma substância chamada feromônio, deixando assim um rastro que indica para as formigas subsequentes o caminho a seguir. Quando uma formiga chega num ponto de decisão, como a interseção entre dois ou mais caminhos, ela sente o cheiro do feromônio acumulado em cada uma das trilhas e tende a escolher o caminho com maior concentração de feromônio.

2.2. TABU SEARCH

A Busca Tabu explora uma coleção de princípios para resolver problemas de maneira inteligente. Um dos elementos fundamentais da BT é o uso da memória flexível [5]. Do ponto de vista da BT, a memória flexível envolve o processo duplo de criar e explorar estruturas para obter vantagem mediante a combinação de atividades de aquisição, avaliação e melhoramento da informação de maneira histórica. Em geral a BT pode ser descrita da seguinte forma: deseja-se mover passo a passo a partir de uma solução inicial de um problema de otimização combinatória até uma solução que proporcione o valor mínimo da função objetivo C. Para tal, pode-se representar cada solução por um ponto s (em algum espaço) e se define uma vizinhança N(s) para cada ponto s como um conjunto de soluções adjacentes à solução s. O passo básico do procedimento consiste em começar de um ponto s e gerar um conjunto de soluções em N(s), de estas se escolhe a melhor s* e se posiciona neste novo ponto, mesmo que C(s*) tenha ou não um melhor valor que C(s). A característica de maior importância da BT é precisamente a construção de uma lista tabu T de movimentos: aqueles movimentos que não sejam permitidos (movimentos tabu) na presente iteração [3]. A razão desta lista é de excluir os movimentos que possam nos retornar a uma iteração anterior. Sendo que, um movimento permanece como tabu somente durante um certo numero de iterações, de forma que se considera que T é uma lista cíclica onde para cada movimento s s* o movimento oposto s* s se adiciona ao final de T onde o movimento mais velho em T se exclui.

2.3. ALGORITMO GENÉTICO

Algoritmos Genéticos são métodos computacionais de busca e otimização inspirados nos mecanismos de evolução natural e da genética. Os Algoritmos Genéticos operam sobre uma população de soluções dos problemas codificados usando transições probabilisticas e não deterministica.

2.3.1. Conceitos básicos

- cromossomo (genótipo) - cadeia de bits que representa uma solução possível para o problema.
- gene - representação de cada parâmetro de acordo com o alfabeto utilizado (binário, inteiro ou real).
- fenótipo - cromossomo codificado

- população - conjunto de pontos (indivíduos) no Espaço de Busca
- geração - iteração completa do AG que gera uma nova população
- aptidão - saída gerada pela função objetivo para um indivíduo da população

Os três aspectos mais importantes do desenvolvimento de um Algoritmo Genético são: (1) definição e implementação da representação genética, (2) definição da função objetivo, e (3) definição e implementação dos operadores genéticos. Uma vez que se tenha definido estes três aspectos, o algoritmo genético deve trabalhar bem. além disto ainda se pode tentar muitas variações para melhorar o funcionamento através do ajuste de parâmetros.

3. ANÁLISE DO PROBLEMA DO CAIXEIRO VIAJANTE

Os problemas matemáticos relacionados ao PCV foram tratados em meados de 1800 por Sir William Rowan Hamilton e por Thomas Penyngton Kirkman. Uma discussão do trabalho inicial de Hamilton e Kirkman pode ser encontrada no livro *Graph Theory [7]*.

Entretanto, as origens do PCV são obscuras. Na década de 20, o matemático e o economista Karl Menger publicou-o entre seus colegas em Viena e Harvard. Na década de 30, o problema re-apareceu nos círculos matemáticos de Princeton, demonstrado por Hassler Whitney e Merrill Flood. Em meados de 1940 foi estudado por estatísticos (Mahalanobis (1940), Jessen (1942), Gosh (1948), Marks (1948)) em relação a uma aplicação para a área agrícola e o matemático Merill Flood popularizou-o entre seus colegas da RAND Corporation [8].

Eventualmente, o PCV ganhou notoriedade como o protótipo de um problema difícil em otimização combinatorial já que examinar as excursões uma por uma é fora de questão devido a seu número imenso de possibilidades (fatorial) e nenhuma outra idéia estava no horizonte por muito tempo. Um tratamento detalhado da conexão entre Menger, Whitney e o crescimento do PCV como tópico de estudo pode ser encontrado no artigo de Alexander Schrijver [9].

O Problema do Caixeiro Viajante é fácil de enunciar: dado um número finito de "cidades" junto com o custo de viagem entre cada par delas, encontre a maneira mais barata de visitar todas as cidades e retornar a seu ponto inicial ou, de uma forma mais matemática, é encontrar em um **grafo** G=(V,A), o **circuito hamiltoniano** de menor custo.

Um **grafo**, numa definição bem simples, é um conjunto de vértices(V) e arestas(A). Os vértices (ou nós) são pontos que podem representar cidades, depósitos, postos de trabalho ou atendimento. Já as arestas são linhas que conectam os vértices, podendo representar ruas ou estradas, por exemplo. Um **circuito hamiltoniano** é um passeio que percorre todos os vértices de um grafo e retorna ao vértice origem (início do passeio), passando por cada vértice apenas uma vez, a figura abaixo retrata um exemplo de um PCV com 13 cidades e com a melhor rota possível.

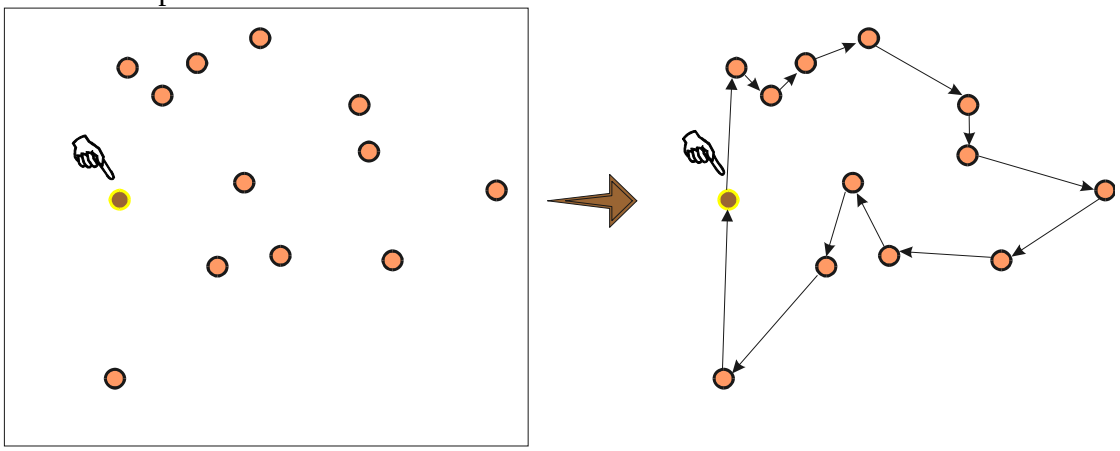

Figura 3.1 – Um exemplo simples do PCV

Tal passeio recebe esse nome devido a **Willian Rowan Hamilton** que, em 1857,

propôs um jogo que denominou ”Around the World”. O jogo foi elaborado sobre um dodecaedro em que cada vértice estava associado a uma cidade importante da época. O objetivo era encontrar uma rota através dos vértices do dodecaedro que iniciasse e terminasse em uma mesma cidade sem nunca repetir uma visita.

O Problema do Caixeiro Viajante é importante devido a pelo menos três de suas características: grande aplicação prática, grande relação com outros modelos e grande dificuldade de solução exata. Em suas diversas versões, o PCV está presente em inúmeros problemas práticos, como por exemplo:

1) Programação de operações de máquinas em manufatura.
2) Programação de transporte entre células de manufatura.
3) Otimização do movimento de ferramentas de corte.
4) Otimização de perfurações de furos em placas de circuitos impressos.
5) Na maioria dos problemas de roteamento de veículos.
6) Na solução de problemas de seqüenciamento de DNA
7) Na solução de problemas de programação e distribuição de tarefas em plantas.
8) Trabalhos administrativos.

O PCV pertence à classe de problemas considerados difíceis ou intratáveis. A busca de uma solução ótima pode "custar caro", pois à medida que aumentamos o número de vértices o tempo para a resolução do problema aumenta de forma gritante (no caso, fatorial) tornando, às vezes, a resolução inviável para os computadores atuais. Para problemas difíceis, são utilizadas algumas técnicas para tentar resolver o problema em tempo hábil como, por exemplo, a elaboração de algoritmos de aproximação.

Algoritmos de aproximação são aqueles que apresentam uma solução correta com a garantia de que esta solução está dentro de uma determinada porcentagem da solução ótima.

Fazendo uma análise do pior caso e do melhor caso que o algoritmo pode produzir, é possível avaliar sua complexidade e a proximidade das suas soluções em relação a aquela que é ótima (difícil é achar a ótima antes). Com o uso de algoritmos de aproximação é possível encontrar, para problemas difíceis, soluções de boa qualidade em tempo computacional aceitável.

3.1. COMPLEXIDADE

Segundo a teoria de NP-completude [10], os problemas de otimização combinatória podem ser classificados segundo sua complexidade em P, NP, NP-difícil e NP-completo. Os problemas pertencentes à classe P são ditos tratáveis computacionalmente, já os demais são considerados intratáveis, ou seja, o número de *computações* executadas no melhor algoritmo conhecido para o problema cresce exponencialmente em função do tamanho da instância. Neste contexto, por *computações* subentende-se operações primitivas (atribuição, soma etc.) [1].

Antes de caracterizar as quatro classes de problemas é necessário saber o que é um algoritmo determinístico e um não-determinístico. Dado o fluxo de controle de um algoritmo, ele é determinístico se, a cada passo deste, o próximo passo é único, enquanto que os algoritmos não-determinísticos fazem uma escolha aleatória do próximo passo, entre um número fixo de possibilidades, e, conseqüentemente, o fluxo do algoritmo depende da alternativa selecionada.

P (*Polinomial Time*): problemas que podem ser resolvidos por algoritmos determinísticos polinomiais em função do tamanho da instância; a complexidade do problema cresce polinomialmente em função do tamanho da instância.

NP (*Non-Deterministic Polinomial Time*): problemas que podem ser resolvidos por algoritmos não-determinísticos polinomiais no tamanho da instância; a complexidade do problema cresce exponencialmente em função do tamanho da instância.

Conclui-se que P $\subseteq$ NP, mas não é sabido se P = NP. Para resolver a questão, todos os problemas pertencentes à classe NP deveriam ser resolvidos em tempo polinomial ou serem reduzidos polinomialmente para os problemas da classe P.

NP-difícil: compostos pelos problemas caracterizados por haver redução polinomial a partir de todo problema pertencente à classe NP [10].

NP-completo: problemas pertencentes à interseção das classes NP e NP-difícil.

Desta forma, para provar que um problema *Q* pertence à classe NP-completo, deve-se mostrar que *Q* está em NP e encontrar um problema *R*, que se sabe estar em NP-difícil, e reduzi-lo polinomialmente ao problema *Q*. O problema do caixeiro viajante pertence à classe NP-difícil.

3.2. APLICAÇÕES

Muitos estudos desenvolvidos com o PCV não foram motivados por aplicações diretas, mas pelo fato deste problema fornecer uma plataforma ideal para o estudo dos métodos gerais que podem ser aplicados a uma grande variedade de problemas discretos de otimização. Ainda assim, o PCV encontra aplicações em muitos outros campos. As numerosas aplicações diretas do PCV certamente trazem a vida à área da pesquisa e ajudam-na a dirigir trabalhos futuros.

O PCV mostra-se como um subproblema em muitas aplicações de transporte e logística como, por exemplo, o problema de criar rotas de ônibus para pegar crianças em um distrito escolar. Esta aplicação é de significado histórico importante para o PCV, pois forneceu a motivação para Merrill Flood, um dos pioneiros da pesquisa de PCV no inicio dos anos 40. Uma segunda aplicação de PCV dos anos 40 envolveu o transporte de equipamento agrícola de um local para outro a fim de testar o solo, conduzindo aos estudos matemáticos em Bengal por P. C. Mahalanobis e em Iowa por R. J. Jessen. As aplicações mais recentes envolvem programação de atendimentos técnicos em firmas de cabeamento, entrega das refeições em casa, programação de guindastes empilhadores nos armazéns, roteamento dos caminhões etc.

Embora as aplicações do transporte fossem naturalmente mais apropriadas para o PCV, a simplicidade do modelo conduziu a muitas aplicações interessantes em outras áreas. Um exemplo clássico é a programação de uma máquina para perfurar furos em placas de circuitos ou em outros objetos. Neste caso os furos a serem perfurados são as cidades e o custo do curso é o tempo que se leva para mover a cabeça da broca de um furo ao próximo. A tecnologia para perfurar varia de uma indústria à outra, mas sempre que o tempo de percurso do dispositivo perfurante é uma parcela significativa do processo total de manufatura, o PCV pode ter um papel importante em reduzir custos.

Alguns exemplos de aplicações do PCV:

3.2.1. Otimização de Varreduras Encadeadas

Um fabricante de semicondutores usou a implementação da heurística “Chained Lin-Kernighan” nas experiências para otimizar varreduras encadeadas em circuitos integrados. Varreduras encadeadas são rotas incluídas em um chip para finalidades de teste e é útil minimizar seu comprimento por razões de sincronismo e de energia. Uma discussão curta de Varreduras encadeadas pode ser encontrada no artigo "ASIC DFT and BIST Alternatives".

3.2.2. Coletar Moedas

Uma aplicação antiga do PCV é programar a coleta das moedas dos telefones pagos de uma dada região. Uma versão modificada da heurística “Chained Lin-Kernighan” foi usada para resolver uma variedade de problemas de coleta de moedas.

Em nossos testes o algoritmo de **Simulated Annealing Modificado** com divisão de domínio obteve soluções cerca de 15% melhores, se comparado com o **Simulated Annealing tradicional** e cerca de 40% melhores se comparados com a heurística gulosa de **Arbitrary**

Insertion Procedure, isso para grandes instâncias de problema.

3.2.3. Cadeias Universais de DNA

Um grupo da AT&T usou resoluções do PCV para computar seqüências do DNA em um projeto de pesquisa de engenharia genética. Na aplicação, uma coleção de *strings* de DNA, cada uma de comprimento "k", foi encaixada em uma *string* universal (isto é, cada uma das *strings* alvo é contida como uma substring na string universal), com o objetivo de minimizar o comprimento da *string* universal. As cidades do PCV são as *strings* alvo e o custo do curso é "k" menos a sobreposição máxima das *strings* correspondentes.

4. SIMULATED ANNEALING X SOLUÇÃO EXATA

Faremos agora a comparação do Simulated Annealing com a solução exata, isso tem como objetivo, mostrar o quão custoso é encontrar a solução exata num problema do tipo NP.

4.1. ALGORITMO DE SOLUÇÃO EXATA

Para encontrar a solução exata do problema do caixeiro viajante podemos utilizar um algoritmo baseado em programação dinâmica, entretanto esse algoritmo trabalhará com uma grande quantidade de combinações e seu crescimento assintótico será exponencial, uma vez que o caixeiro viajante é um problema do tipo NP.

Para mostrar como esse crescimento é custoso do ponto de vista computacional a medida que aumentamos a quantidade de cidades, apresentamos os seguintes dados na tabela abaixo:

Número de cidades	Custo	Tempo (em segundos)
3	**3**	**0**
4	**4**	**0**
5	**5**	**0**
6	**6**	**0,01**
7	**7**	**0,03**
8	**8**	**0,27**
9	**9**	**2,604**
10	**10**	**27,67**
11	**11**	**316,395**
12	**12**	**4042,7**
13	**13**	**56395,665**

Tabela 4.1.1 – Soluções exatas

Obs. Foi utilizado um conjunto de dados que a medida que aumentávamos uma cidade, o custo ótimo aumentava de apenas uma unidade, nosso objetivo nesse sub-tópico é demostrar que para problemas razoavelmente grandes (no nosso caso acima de 11 cidades) o algoritmo exato começa a não ser interessante pois seu custo de processamento para as diversas combinações é extremamente alto.

A título de ilustração mostraremos a seguir, o gráfico 4.1.1 relativo a tabela 4.1.1:

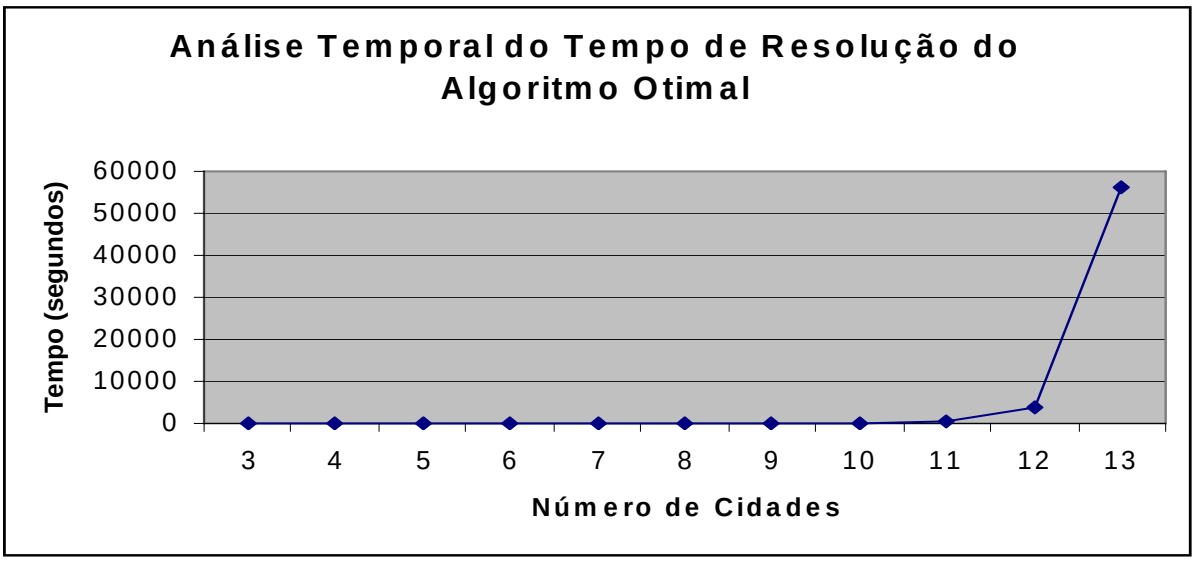

Gráfico 4.1.1 – Soluções Otimais

4.2. ARBITRARY INSERTION PROCEDURE

Consiste em escolher uma cidades aleatoriamente e inserindo-as nas melhores posições possíveis, ou seja, as que forem montando percursos com os mais baixos custos. Será o algoritmo utilizado para a geração do S0 no *Simulated Annealing*. Seu custo de processamento se comparado ao Algoritmo de Solução Exata é insignificante, gera soluções razoáveis e por isso, é um bom ponto de partida para o *Simulated Annealing*. Vejamos então a análise de complexidade desse algoritmo:

Custo	
1) c1	Comece com uma cidade i qualquer
2) c2	Determine a cidade mais próxima de i e insira-a no subtour
3) c3	Pegue uma cidade ainda não escolhida e descubra em que posição ela deve ser inserida de forma a ter o menor custo
4)	Volte ao passo 3 ate todas as cidades serem escolhidas

Onde:

p = número de cidades no tour

a1 = custo do algoritmo que calcula o custo de um sub-tour = O(n)

Algoritmo 4.2.1 - Arbitrary Insertion Procedure:

$$T(n) = c1 + (n-1)*c2 + \left(\sum_{p=2}^{n-1}(p-1)\right) * a1 * c3$$

$$<= c1 - c2 + n*c2 + ((1 + n - 2) * ((n-2)/2)) * O(n) = O(n^3)$$

Ser $O(n^3)$ faz esse algoritmo ser interessante como ponto de partida para o *Simulated Annealing*, uma vez que, o problema atacado (caixeiro viajante) é exponencial. Outro fato interessante é as constantes assintóticas implicítas na notação $O(n^3)$ são pequenas se

comparadas com outros algoritmos como por exemplo o Nearest Addition Heuristic, que também é $O(n^3)$ mas tem um tempo de execução superior ao Arbitrary Insertion Procedure.

4.3. SIMULATED ANNEALING

O algoritmo de *Simulated Annealing* (*SA*) pertence a uma classe de algoritmos de pesquisa local que também são conhecidos sob a designação genérica de "algoritmos de limiar" (do inglês, *Threshold Algorithms*). Estes algoritmos desempenham um papel importante na pesquisa local, por duas razões; em primeiro lugar, porque têm sido bem sucedidos, quando aplicados a um vasto conjunto de problemas práticos, o que lhes atribuiu uma considerável afirmação entre os seus utilizadores. Em segundo lugar, alguns desses algoritmos, tal como é o caso do *SA*, têm uma componente estocástica, que facilita uma análise teórica da sua convergência assintótica e isto tornou-os bastante populares entre os matemáticos.

O algoritmo de *Simulated Annealing* foi originalmente proposto por Kirkpatrick [5], em 1983. O objetivo de um algoritmo desta natureza é encontrar a melhor solução de entre um número finito de soluções possíveis. A técnica de *SA* é uma técnica particularmente atrativa pois permite encontrar soluções próximas da ótima, à custa de um esforço computacional geralmente pouco exagerado. É de notar que neste tipo de algoritmo, não é possível saber se a melhor solução encontrada, é o ótimo global.

Em procedimentos do tipo *SA*, as sequências das soluções não tendem linearmente para um ótimo local, como acontece noutras técnicas de pesquisa local. Sendo assim, verifica-se que as soluções traçam um percurso ou trajeto variável, através de um conjunto S, de soluções possíveis e este percurso tende a ser guiado numa direção "favorável".

Regra geral, poderá dizer-se que o *SA* consiste num procedimento fiável para usar em situações em que o conhecimento é escasso ou se aparenta difícil de aplicar algoritmicamente. Mesmo para dar soluções a problemas complexos, esta técnica é relativamente fácil de implementar e normalmente executa um procedimento do tipo *hill-climbing* com múltiplos recomeços. Tipicamente, esta técnica gera um caminho "Markoviano", em que o sucessor de um ponto actual é escolhido estocasticamente, com uma probabilidade que depende apenas do ponto actual e não da história prévia da busca. A propriedade "Markoviana" é uma bênção misturada, ou seja, permite resultados heurísticos, que são bastante bons em muitos casos, mas torna a análise teórica do algoritmo difícil. O algoritmo de *SA* é claramente desapropriado para a resolução ótima de problemas de otimização.

Em resumo, o algoritmo de *Simulated Anne*aling (*SA*) é um algoritmo estocástico com uma analogia "física" de "fusão" do sistema a ser otimizado (como um sólido) usando "temperaturas elevadas" e posteriormente prosseguindo através de uma redução progressiva e suave da temperatura, até que o sistema "cristalize" e não ocorram mais alterações. O processo de fusão pode ser visto como a busca pela melhor solução.

Vejamos a seguir o pseudo-código do algoritmo de *Simulated Annealing:*

Algoritmo 4.3.1: *Algoritmo de Simulated Annealing*

```
Obtenha uma solução inicial S0
Faça T <- T0 (onde T0 é uma temperatura inicial)
Repita
        For i de 1 até o número de iterações internas faça
                Gere um vizinho de S0 denotado por S1
                Se F(S1) < F(S0)
                        S0 <- S1
                        Smax  <- S0
                Senão
                        Gerar q entre [0,1]
                        Se exponencial(- (F(S1) – F(S0))/T) > q
```

S0 <- S1

Fim for

*T <- Fator de Diminuição da Temperatura * T*

S0 <- Smax

Até que o critério de parada seja satisfeito

Nota: Verificar que ao final do processamento S0 possui a melhor solução encontrada .

5. SIMULATED ANNEALING COM DIVISÃO DE DOMÍNIO

Nesta seção abordamos o algoritmo implementado. Como vimos o *Simulated Annealing* é um algoritmo que pode encontrar soluções próximas ao ótimo mas isso não é garantido, isso porque na função de geração do vizinho do *Simulated Annealing* não existem garantias que um vizinho gerado não possa já ter sido gerado em uma iteração anterior, se isso fosse possível teríamos a garantia que o *Simulated Annealing* sempre encontraria a melhor solução. É nesse aspecto que o *Simulated Annealing* com Divisão de Dominio ataca, determinar se um vizinho novo gerado já foi gerado numa iteração anterior é extremamente custoso e voltaríamos ao estado do algoritmo exato. Nossa proposta é induzir o *Simulated Annealing* a não gerar muitos vizinhos iguais. Isso é feito da seguinte forma:

- ⇨ Geramos todas as combinações possíveis de troca de duas cidades no *tour*.
- ⇨ Determinamos blocos, ou seja, dividimos o domínio de combinações em blocos de combinações.
- ⇨ Cada combinação de troca pertence a um único bloco.
- ⇨ Fazemos então que cada geração de novos vizinhos se baseie em cada momento num bloco diferente, fazendo com que as trocas não fiquem centralizadas num universo restrito de combinações, que poderiam ficar muito dependentes da semente aleatória utilizada.

Vamos explicar esse processo utilizando o algoritmo 4.3.1 e dando um exemplo de como isso seria feito:

Suponhamos que temos o seguinte tour:

1 2 5 6 7 4 8 9 3 (onde cada número representa uma cidade)

As possíveis combinações de troca são:

(1,2), (1,3), (1,4), (1,5), (1,6), (1,7), (1,8), (1,9), (2,3), (2,4), (2,5), (2,6), (2,7), (2,8), (2,9), (3,4), (3,5), (3,6), (3,7), (3,8), (3,9), (4,5), (4,6), (4,7), (4,8), (4,9), (5,6), (5,7), (5,8), (5,9), (6,7), (6,8), (6,9), (7,8), (7,9), (8,9).

Totalizando 36 combinações possíveis de troca.

Suponhamos que utilizassemos o *Simulated Annealing* e ele promove-se as seguintes trocas:

(1,2) (1,3) (1,5) (1,6)

Note que esse comportamento não seja desejado, pois nosso universo de trocas pode ficou restrito a trocas em que a cidade 1 aparecia.

Vejamos como o *Simulated Annealing* com Divisão de Domínio contorna esse possível problema:

1) Divide-se as combinações em blocos, por exemplo de tamanho 9 cada um.

Daí temos o seguinte:

Bloco 1: (1,2), (1,3), (1,4), (1,5), (1,6), (1,7), (1,8), (1,9), (2,3)

Bloco 2: (2,4), (2,5), (2,6), (2,7), (2,8), (2,9), (3,4), (3,5), (3,6)
Bloco 3: (3,7), (3,8), (3,9), (4,5), (4,6), (4,7), (4,8), (4,9), (5,6)
Bloco 4: (5,7), (5,8), (5,9), (6,7), (6,8), (6,9), (7,8), (7,9), (8,9)

2) Escolhemos 1 troca em cada bloco e ao final desse processo voltamos ao bloco inicial e começamos tudo novamente.

Daí poderíamos obter a seguinte sequência de trocas:

(1,2) (3,6) (4,6) (7,9)

Note que aumentamos a chance de obter soluções diferentes diminuíndo assim a dependência de uma boa semente aleatória.

Com essa estratégia esperamos obter resultados melhores que o *Simulated Annealing* Tradicional, isso porque acreditamos ter acrescentado alguma inteligência a mais no processo de geração de vizinhos.

6. RESULTADOS EXPERIMENTAIS

Nesta seção apresentaremos os resultados obtidos para um conjunto de instâncias retiradas de um site que contem um conjunto de arquivos com problemas de tsp já resolvidos com seus melhores caminhos já traçados:

http://www.informatik.uni-heidelberg.de/groups/comopt/software/TSPLIB95/STSP.html.

Todos os testes foram feitos em um Athlon, 1Ghz, , 256mb de memória ram, sendo 64 compartilhados para vídeo , rodando Linux Kurumin 2.21, no compilador g++ 3.3. Para cada instância de problema foram feitas 10 execuções, obtendo-se assim os dados para a análise de custos médio, máximo e mínimo.

É importante ressaltar que a necessidade de serem feitas várias execuções decorre do fato de utilizarmos uma semente aleatória para a geração de randômicos, tal medida visa diminuir a influência que uma semente específica possa ter na análise dos dados.

Os menores custos ditados pelo site do TSPLIB para os problemas são:

- **bays29 : 2020 (29 cidades)**
- **bayg29 : 1610 (29 cidades)**
- **swiss42 : 1273 (42 cidades)**
- **brazil58 : 25395 (58 cidades)**
- **si175 : 21407 (175 cidades)**
- **rat575: 6773 (575 cidades)**
- **rat783 : 8806 (783 cidades)**
- **rl1304 : 252948 (1304 cidades)**
- **pr2392 : 378032 (2392 cidades)**

Experimentalmente, as heurísticas implementadas atingiram os valores abaixo, o primeiro campo indica a porcentagem atingida da solução ótima e o segundo o tempo gasto em segundos. É válido afirmar que todos os parâmetros do *Simulated Annealing* e do *Simulated Annealing* com Divisão de Domínio como temperatura inicial, número de iterações internas, etc, são todos iguais:

Bays29:

Execução	Arbitrary Insertion Procedure		Simulated Annealing		S. A. Divisão de Domínio	
Média	244,25	0,00	118,90	0,50	121,88	0,41
Máxima	270,40	0,00	127,48	0,44	136,36	0,39
Mínima	226,29	0,00	108,12	0,43	109,21	0,38

Bayg29:

Execução	Arbitrary Insertion Procedure		Simulated Annealing		S. A. Divisão de Domínio	
Média	246,28	0,00	112,68	0,30	123,63	0,44
Máxima	268,26	0,00	123,79	0,28	133,35	0,39
Mínima	207,58	0,00	105,22	0,28	111,43	0,51

Swiss42:

Execução	Arbitrary Insertion Procedure		Simulated Annealing		S. A. Divisão de Domínio	
Média	317,12	0,00	136,00	0,37	138,02	0,60
Máxima	382,17	0,00	144,54	0,37	148,00	0,52
Mínima	275,41	0,00	119,80	0,34	132,05	0,52

Brazil58:

Execução	Arbitrary Insertion Procedure		Simulated Annealing		S. A. Divisão de Domínio	
Média	283,00	0,00	148,66	1,07	145,05	1,16
Máxima	387,08	0,00	165,55	0,86	156,16	1,16
Mínima	235,95	0,01	127,76	1,22	133,13	1,18

Si175:

Execução	Arbitrary Insertion Procedure		Simulated Annealing		S. A. Divisão de Domínio	
Média	223,76	0,03	140,09	4,69	135,56	4,01
Máxima	228,36	0,03	143,07	4,65	138,29	4,05
Mínima	216,96	0,02	136,96	4,72	134,25	4,01

Rat575:

Execução	Arbitrary Insertion Procedure		Simulated Annealing		S. A. Divisão de Domínio	
Média	1098,14	1,53	641,28	35,23	569,79	30,27

Máxima	1099,07	2,06	656,88	34,35	591,93	33,52
Mínima	1097,19	0,74	624,99	35,88	547,20	30,22

Rat783:

Execução	**Arbitrary Insertion Procedure**		**Simulated Annealing**		**S. A. Divisão de Domínio**	
Média	1311,74	1,92	845,10	49,06	733,58	41,00
Máxima	1312,51	2,45	866,37	50,10	764,16	40,81
Mínima	1311,15	2,55	803,64	48,43	703,25	42,77

Rl1304:

Execução	**Arbitrary Insertion Procedure**		**Simulated Annealing**		**S. A. Divisão de Domínio**	
Média	2742,37	5,76	1628,68	87,21	1522,61	77,79
Máxima	2742,40	5,86	1667,30	90,31	1560,29	79,49
Mínima	2742,35	5,80	1593,33	88,86	1470,42	78,90

Pr2392:

Execução	**Arbitrary Insertion Procedure**		**Simulated Annealing**		**S. A. Divisão de Domínio**	
Média	2858,31	26,23	2021,82	179,17	1856,04	167,06
Máxima	2858,33	26,51	2051,35	180,17	1879,60	160,56
Mínima	2858,29	26,26	1985,09	181,64	1825,23	170,24

6.1. ANÁLISE DOS DADOS

A estratégia de *Simulated Annealing* Tradicional mostrou-se mais interessante para pequenas instâncias de problema, como: Bays29, Bayg29 e Swiss42. A partir do Brazil58 o *Simulated Annealing* com Divisão de Domínio começa a mostrar-se superior obtendo melhores média e máximo, daí em diante a supremacia do *Simulated Annealing* com Divisão de Domínio é incontestável, para as unidades de problema testadas, obtendo todos os melhores valores em todos os quesitos: média, máximo e mínimo para as unidades de problema Si175, Rat575, Rat783, Rl1304, Pr2392 e Pr2392. A justificativa para não testarmos unidades de problema maiores são as limitações de memória da máquina utilizada, entretanto como podemos ver existe uma tendência que o *Simulated Annealing* com Divisão de Domínio obtenha resultados melhores que o *Simulated Annealing* Tradicional.

7. CONCLUSÕES

Inicialmente é importante ressaltar a dificuldade que é comparar algoritmos de natureza meta-heurística, existem muitos parâmetros envolvidos no processo de obtenção de melhores soluções, no caso do Simulated Annealing podemos destacar: A temperatura inicial, a taxa de decrescimento, a temperatura mínima aceitável, entre outros.

Concluímos, baseado nos dados obtidos, que o *Simulated Annealing* com Divisão de Domínio atendeve as expectativas pois foi superior ao *Simulated Annealing* Tradicional na

maior parte da massa de testes, destacando aí que o melhor desempenho ocorreu em unidades de problema médias e grandes, principal alvo de ataque de algoritmos meta-heurísticos. A idéia de dividir domínios de combinações é interessante, visto que, diminui o grau de aleatoriedade do algoritmo *Simulated Annealing*, possibilitando assim uma distribuição mais uniforme das trocas executadas.

A área de otimização e problemas NP ainda são alvo de muita pesquisa, toda "inteligência" incorporada a algoritmos meta-heurísticos, visa diminuir a aleatorieadade na geração de novas solução, fazendo assim uma busca mais "inteligente". Esse é o foco e objetivo dos cientistas que trabalham nessa área, sendo assim, temos a sensação de termos contribuído de alguma forma para o desenvolvimento desse processo.

8. REFERÊNCIAS BIBLIOGRÁFICAS

[1] CORMEN, Thomas. **Introduction to algorithms**. Cambridge, EUA: Massachusetts Institute of Tecnology, 2001.

[2] DORIGO, Marco, DI CARO, Gianni. **Ant Optimization: A new meta-heuristic**. In Peter J. Angeline, Zbyszek Michalewicz, Marc Shoenauer, Xin Yao and Aly Zalzala, Proceedings of the Congress on Evolutionary Computation, volume 2, pages 1470-1477, Mayflower Hotel, Washington D.C., 6-9 July 1999. IEEE Press.

[3] ROSENKRANTZ, STEARNS, LEWIS. 1977.

[4] GOLDBERG, David. **Genetic Algorithms**. Addison Wesley, Reading, 1989.

[5] KIRKPATRICK, S., GELLATT, D., VECCHI, P. **Simulated Annealing**. Science, 220(671), 1983.

[6] YAGIURA, M., IBARAKI T. **Efficient 2 and 3-Flip neighborhood seach algorihms for the** MAX-SAT. Lecture Notes in Computer Science, 1449, 1998.

[7] N. L. Biggs, E. K. Lloyd & R. J. Wilson . ***Graph Theory***, *1736-1936*, Clarendon Press, Oxford,1976.

[8] http://www.math.princeton.edu/tsp/history.html

[9] SCHRIJVER, Alexander. **On the history of combinatorial optimization**, 1960, http://www.cwi.nl/~lex/files/histco.ps.

[10] GAREY, M.R. & D. S. Jonhson. **Computers and Intractability: A Guide to the Theory of NP-Completeness**, San Francisco: Freeman, 1979.