A Técnica 'Simulated Annealing' aplicada aos problemas de Percurso do Cavalo e Damas Pacíficas

Bruno de Souza Almeida
Paulo E. D. Pinto
Rodrigo Rafael Amaral Soriano
UERJ, Universidade Estadual do Rio de Janeiro, RJ, Brasil.

Abstract

Um grande desafio para a área de Algoritmos é o conjunto de problemas que formam a classe NP-Completo. Trata-se de inúmeros problemas de grande importância teórica e prática, para os quais não são conhecidas soluções computacionais eficientes. Várias técnicas têm sido desenvolvidas para tentar contornar a intratabilidade desses problemas. Dentre elas, uma das primeiras foi o Simulated Annealing*. Este trabalho explica e ilustra a técnica, aplicando-a a dois problemas clássicos, ambos relacionados a um tabuleiro de xadrez: o Percurso do Cavalo e Damas Pacíficas. O objetivo é muito mais didático do que o de buscar soluções eficientes para os problemas, uma vez que essas soluções já existem.*

1 Introdução

Neste trabalho iremos utilizar a técnica 'Simulated Annealing' (SA) para a resolução de dois problemas clássicos, relacionados ao jogo de xadrez: 'Percurso do Cavalo' (Knight Tour) e 'Damas Pacíficas' (Eight Queens). Assim, exemplificaremos como a técnica é utilizada, sua simplicidade e flexibilidade na resolução de problemas. Devemos, entretanto, observar que os dois problemas tratados pertencem à classe P de problemas, pois existem heurísticas conhecidas capazes de resolvê-los facilmente.

Este trabalho está organizado da seguinte maneira: Na Seção 2 abordaremos a complexidade de problemas computacionais e as classes de algoritmos desenvolvidos para solucioná-los. Na Seção 3 estudaremos a técnica *Simulated Annealing*. Na Seção 4 introduziremos os problemas *Percurso do Cavalo e Damas Pacíficas* e aplicaremos a técnica do Simulated Annealing para resolvê-los, coletando os resultados obtidos e fazendo a análise necessária. Finalizaremos nas Conclusões, com nossas observações finais sobre o trabalho.

2 Complexidade de algoritmos

Os conceitos a seguir são de utilidade para este artigo e encontram-se mais detalhados em vários livros clássicos [1] [4].

2.1 Fundamentos

Um algoritmo é um método utilizado para resolver uma classe de problemas em um computador. A complexidade de um algoritmo é o custo, medido em tempo de execução, ou espaço utilizado, ou em quaisquer unidades relevantes, do uso do algoritmo para a resolução de um desses problemas. Alguns problemas requerem uma grande quantidade de tempo, outros são solucionados rapidamente. Alguns problemas parecem precisar de muito tempo, e então alguém descobre uma maneira melhor de solucioná-los (um algoritmo mais rápido). O estudo da quantidade de esforço computacional necessário para realizar certos tipos de computações é o estudo da complexidade computacional.

Naturalmente poderíamos esperar que um problema computacional que requeira milhões de bits de dados de entrada irá ser solucionado mais lentamente que outro problema que precisa apenas de uns poucos itens de entrada. Portanto, a complexidade temporal de um cálculo é medida expressando-se o tempo de execução do mesmo como uma função relacionada à quantidade de dados necessários para descrever o problema no computador.

Por exemplo, um programa inversor de matrizes pode ser capaz de inverter uma matriz n x n em $1.2n^3$ minutos. Essa é uma descrição típica da complexidade de um algoritmo qualquer, onde o tempo de execução é dado em função do tamanho da matriz de entrada. Um programa mais rápido poderia realizar a inversão da matriz em $0.8n^3$ minutos, por exemplo. E se alguém fizesse uma descoberta realmente importante na área, talvez pudéssemos reduzir o expoente da função, ao invés de reduzir apenas a constante de multiplicação.

Assim sendo, podemos introduzir o conceito de problemas fáceis e difíceis: Um problema cujo tempo de execução é expresso no máximo como uma função polinomial da quantidade de dados de entrada é considerado um problema fácil, caso contrário, é um problema difícil. Por exemplo, se o tempo de execução de um problema é representado por cn^3 (onde c é uma constante e n é a quantidade de dados de entrada), este pode ser considerado um problema fácil. Já se uma certa computação realizada em uma matriz n x n levar 2^n minutos, esse seria um problema difícil.

Naturalmente algumas das computações que estamos chamando 'fáceis' podem levar muito tempo para serem efetuadas, mas ainda assim, do nosso ponto de vista atual, a importante distinção a ser mantida é a garantia de tempo polinomial ou não. Muitos problemas na área da Ciência da Computação são considerados fáceis. Para convencer alguém que um problema é fácil, basta descrever uma maneira rápida para solucionar aquele problema.

Já convencer alguém que um problema é difícil é bem complexo, pois é necessário provar a impossibilidade de encontrar uma maneira rápida de realizar o cálculo da solução. Não é suficiente apontar um algoritmo específico e demonstrar a sua lentidão, pois, afinal de contas, pode ser que esse algoritmo específico seja lento, mas isso não impede que exista uma maneira mais rápida de solucionar o problema. O problema de inversão de matrizes n x n é um problema fácil, resolvido pelo método de eliminação gaussiana em tempo polinomial (cn^3).

Para dar um exemplo de um problema difícil podemos citar o seguinte: Suponha que temos infinitos azulejos, todos em formato hexagonal, de mesmo tamanho. Podemos cobrir um plano inteiro com eles, de maneira perfeita, ou seja, sem deixar espaços vagos entre eles. Isso também pode ser feito se os azulejos forem retangulares, mas não é possível fazê-lo se estiverem em formato pentagonal.

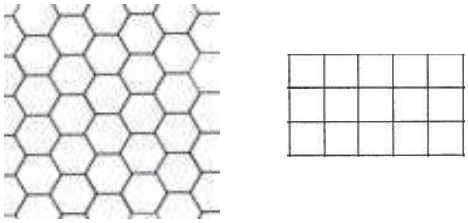

Figura 1. Plano coberto por formas hexagonais/retangulares

Agora, suponha que tenhamos infinitos azulejos, na forma de um polígono qualquer, não necessariamente regular e/ou convexo. É possível ou não preenchermos perfeitamente um plano com eles? Essa questão foi provada computacionalmente impossível de ser resolvida. Em outras palavras, não apenas não sabemos de um meio rápido de resolver esse problema em um computador; além disso, provou-se não haver nenhuma maneira de se fazer isso. Então, procurar por um algoritmo para o problema é inútil. Isso não significa que a questão é difícil para todos os polígonos.

Problemas difíceis podem possuir instâncias fáceis de serem resolvidas. O que foi provado é a inexistência de um método único que resolva a questão para todos os polígonos. Note que a quantidade de dados de entrada para esse problema é mínima: é necessário apenas indicar o formato do polígono desejado. Ainda assim, não apenas é impossível descobrir um algoritmo rápido para esse problema, como também se provou impossível encontrar um algoritmo qualquer que seja capaz de dar uma resposta Sim/Não para o problema após um número finito de passos.

2.2 Problemas fáceis X Problemas difíceis

Agora vamos definir melhor o que queremos dizer quando declaramos uma computação como 'fácil' ou 'difícil'. Pense em um algoritmo como se fosse uma pequena caixa, capaz de resolver uma certa classe de problemas computacionais. Colocamos dentro da caixa a descrição de um problema particular pertencente àquela classe, e após um certo período de tempo a resposta aparece. Um algoritmo rápido seria um que garante uma rápida performance. Seguem alguns exemplos:

1) Se o problema for descrito com b bits de entrada, então uma resposta será dada em no máximo em $6b^3$ minutos.

2) É garantido que qualquer problema definido a partir de b bits de entrada será resolvido em, no máximo, $0.7b$ segundos.

Uma garantia de performance, como as duas citadas acima, é normalmente chamada de 'estimativa de complexidade do pior caso', por motivos óbvios. Por exemplo, se temos um algoritmo que organize uma sequência de números quaisquer em ordem crescente, ele poderá descobrir que certas sequências são mais fáceis de serem organizadas que outras. Por exemplo, a sequência $1, 2, 7, 8, 11, 10, 20$ já está praticamente em ordem, portanto nosso algoritmo poderá, se tirar vantagem desse fato, organizar a sequência rapidamente. Outras sequências podem ser mais complexas para organizar, portanto levarão mais tempo.

Sendo assim, em alguns problemas cuja entrada possui b bits, o algoritmo pode operar em $6b$ unidades de tempo, e em outras, ele pode precisar de, por exemplo, $10b \log b$ unidades de tempo, e ainda para outras instâncias dessa classe

de problemas com entrada de b bits o algoritmo pode precisar de $5b^2$ unidades. Mas o que a garantia nos diz? Ela precisa lidar com a pior possibilidade, pois senão ela não será válida. Ela assegura ao usuário que, se o problema de entrada for descrito por b bits, então uma resposta será dada em, no máximo, $5b^2$ unidades de tempo. Portanto, uma garantia de performance é equivalente a uma estimativa da pior possibilidade: o tempo de resolução mais longo possível para um problema de entrada de b bits.

Uma garantia que não nos assegura uma performance rápida deve conter alguma função de b que cresça mais rapidamente que qualquer polinômio. Como por exemplo e^b, ou $2^{b^{\frac{1}{2}}}$. É a garantia de tempo polinomial que faz a diferença entre as classes de problemas fáceis e difíceis, e entre os algoritmos rápidos e lentos. É altamente desejável trabalhar com algoritmos tais que sua garantia de tempo de execução seja, no máximo, uma função polinomial do número de bits de entrada.

Concluindo, um problema computacional é tratável se existe um algoritmo rápido que solucione todas as suas instâncias. Um problema é intratável se foi provado que não existe um algoritmo rápido para resolvê-lo.

Exemplo: consideremos um problema computacional bem conhecido, e um algoritmo para solucioná-lo. Vejamos se esse algoritmo possui uma garantia de tempo polinomial ou não: Seja n um número inteiro. Queremos descobrir se n é composto e, em caso afirmativo, queremos escrever n como o produto $n = f_1.f_2$, onde f_1 e f_2 são inteiros diferentes de 1. Ou seja, queremos fatorar n.

O algoritmo escolhido é o seguinte: Para cada número inteiro $m = 2, 3, \ldots, n^{\frac{1}{2}}$ perguntaremos se m é divisor exato de n. Se todas as respostas foram 'não', então declaramos que n é um número primo. Caso contrário, n é um número composto e $f_1 = m$, onde m é o menor número que dividiu exatamente n. Então, $f_2 = n/m$. Vamos analisar a complexidade computacional desse algoritmo. Isso quer dizer que iremos determinar quanto trabalho é envolvido na realização desse teste. Para um número inteiro n, o traba- lho que será realizado pode ser medido em unidades de divisões de um número inteiro por outro número inteiro. Nessa medida, teremos $n^{\frac{1}{2}}$ unidades de trabalho. Parece que temos então um problema tratável, pois $n^{\frac{1}{2}}$ cresce polinomialmente em função de n.

Sendo assim, de acordo com a definição anterior de algoritmos lentos e rápidos (cuja distinção foi feita levando em conta o crescimento polinomial x crescimento mais rápido que polinomial da quantidade de trabalho para solucionar o problema), esse problema deve ser fácil. Será?

Não exatamente. Ao retornarmos à definição da distinção entre métodos lentos e rápidos, vemos que temos que medir a quantidade de trabalho realizado em função do número de bits de entrada do problema. Neste exemplo, n não é o número de bits de entrada. Por exemplo, se $n = 59$, nós não precisamos de 59 bits para descrever n, apenas 6. Em geral, o nmero de dígitos binários em um string de bits que representa um número inteiro é próximo de $\log n$.

Portanto, no problema anterior, o tamanho da entrada b é aproximadamente $\log n$. Visto desta maneira, o cálculo da solução passa a se tornar muito longo, pois um string que consiste de apenas $\log n$ dígitos binários faz com que o computador realize $n^{\frac{1}{2}}$ unidades de trabalho. Se expressarmos a quantidade de trabalho realizado em função de b, descobriremos que a complexidade desse cálculo é de aproximadamente $2^{\frac{b}{2}}$, que apresenta um crescimento muito maior que qualquer função polinomial de b.

Consequentemente, o método que discutimos para fatorar um inteiro n é lento, e até hoje ninguém encontrou uma maneira significativamente melhor de realizar a tarefa, mas também ninguém provou que não existe uma maneira mais rápida. A Fatoração pertence à classe de problemas que são aparentemente (mas não asseguradamente) intratáveis.

2.3 NP - Completeza

Nos tópicos anteriores vimos que existem problemas para os quais ainda não foram encontrados algoritmos rápidos, e nem se provou que tais algoritmos não poderão ser encontrados. Agora nós analisaremos uma grande família de problemas, que não são apenas uma lista de problemas computacionais aparentemente difíceis; Na verdade eles estão ligados por fortes laços estruturais. Essa coleção de problemas, chamados de problemas NP-completos, inclui várias questões muito conhecidas e importantes na área da matemática computacional, como as seguintes:

Problema do caixeiro viajante: Dados n pontos em um plano (cidades) e uma distância d. Existe uma rota que passe por todas as n cidades, retorne ao ponto de partida, e cuja distância seja menor ou igual a d?

Coloração de grafos: Dados um grafo G e um inteiro k. Os vértices de G podem ser adequadamente coloridos (de maneira que cada vértice nunca esteja ligado a um vértice de mesma cor que a dele) usando k ou menos cores?

Empacotamento: Dado um conjunto finito S de inteiros positivos, e um inteiro n (o número de pacotes). Existe uma partição de S em n ou menos subconjuntos tal que a soma dos inteiros em cada subconjunto seja menor ou igual a k? Em outras palavras, podemos 'empacotar' os inteiros de S em, no máximo n pacotes com capacidade máxima k?

Estes são problemas computacionais muito difíceis. Por exemplo, ao analisar o problema de coloração de grafos, poderíamos tentar todas as possíveis maneiras de colorir os vértices de G com k cores para verificar se alguma delas funciona. Existem k^n possibilidades, se G possui n vértices. Portanto uma grande quantidade de recursos computacionais será utilizada, quantidade tão grande que, se G possui 50 vértices e tivermos 10 cores disponíveis, o problema estará muito além da capacidade dos computadores mais rápidos existentes atualmente. Entretanto, problemas difíceis podem ter instâncias simples. Se o grafo G não possuir nenhuma aresta, ou um número pequeno delas, será bem simples descobrir se uma coloração adequada é possível. Sendo assim, não existe contradição no fato de que o problema é difícil e de que existem instâncias simples dele. A dificuldade do problema vem da aparente impossibilidade de produzir um algoritmo que garanta solucioná-lo em tempo polinomial, como vimos anteriormente.

Agora vamos retornar às propriedades da família de problemas NP-completos. Abaixo seguem algumas delas:

1) Todos os problemas parecem ser computacionalmente muito difíceis, e nenhum algoritmo de tempo polinomial foi encontrado para nenhum deles.

2) Ainda não foi provada a inexistência de algoritmos com tempo polinomial para estes problemas.

3) Esta família de problemas está ligada por uma propriedade. Se um algoritmo rápido puder ser encontrado para um problema NP-completo então existem algoritmos rápidos para todos os problemas da família.

4) Analogamente, se puder ser provado que não existe um algoritmo rápido para um dos problemas da família, então não haverá algoritmo rápido para nenhum dos outros problemas.

As propriedades acima não pretendem ser uma definição formal do conceito de NP-completeza. Elas são apenas uma lista de características interessantes desses problemas que, quando combinadas com a importância teórica e prática dos mesmos, dão razão ao intenso esforço mundial de pesquisa que tem sido efetuado para estudá-los nos últimos anos. A questão da existência ou não de algoritmos de tempo polinomial para os problemas NP-completos provavelmente é o principal problema teórico ainda não solucionado na área da Ciência da Computação atualmente.

2.4 Alternativas para a intratabilidade

Se tivéssemos que solucionar um problema NP-completo, certamente estaríamos diante de um processo de computação extremamente longo. Existe algo que pode ser feito para reduzir esse problema? Em vários casos foram desenvolvidos algoritmos probabilísticos e de aproximação bastante engenhosos, e de grande valia nessa questão. A seguir são citadas algumas das estratégias que foram desenvolvidas:

Tipo 1: *Quase sempre eficiente*
Os algoritmos desse tipo apresentam as seguintes propriedades:
(A) Sempre operam em tempo polinomial
(B) Quando encontram uma solução é sempre uma solução correta
(C) Nem sempre encontram solução, mas normalmente o fazem, considerando que a taxa de sucessos em relação ao número de casos aumenta de maneira proporcional ao tamanho da entrada.

Tipo 2: *Usualmente rápido*
Nesta categoria estão os algoritmos cuja incerteza está no tempo necessário para se encontrar uma solução.Um algoritmo desse tipo irá:
(A) sempre encontrar uma solução e ela sempre será correta
(B) operar em uma média de tempo sub-exponencial, mas às vezes irá operar em tempo exponencial. A média é feita sobre todas as entradas de um mesmo tamanho.

Tipo 3: *Usualmente correto*
Neste tipo de algoritmo não se consegue a resposta correta, mas se chega próximo à mesma; em compensação, esses algoritmos normalmente são bem rápidos.
Assim, os algoritmos desse tipo irão:
(A) executar em tempo polinomial
(B) sempre encontrar alguma resposta
(C) garantir que a resposta encontrada não difere da ótima além de uma faixa especificada.

Dentre essas abordagens, podemos citar as meta-heurísticas, algoritmos que utilizam métodos engenhosos para garantir a proximidade da resposta ótima ao problema.
Na próxima seção analisaremos um dos primeiros algoritmos meta-heurísticos : *Simulated Annealing*.

3 A técnica Simulated Annealing

3.1 Introdução

Simulated Annealing é uma heurística probabilística para problemas de otimização global que geralmente encontra

uma boa aproximação do ótimo global.

Foi criada em 1983 por Kirkpatrick [2], quando este propôs um método baseado no algoritmo Metropolis Monte Carlo Simulation [2] para encontrar o nível de energia mínimo ou máximo para um determinado sistema.

O termo 'annealing' refere-se a um processo metalúrgico de resfriamento térmico usado para tornar o metal o mais forte possível. Este procedimento começa pelo aquecimento de um metal até uma alta temperatura, de forma que o metal se torne líquido e os átomos possam se mover de forma relativamente livre. A temperatura do metal é então lentamente diminuída de forma que, a cada nova tempe- ratura, os átomos possam se mover o suficiente para começar a adotar uma posição mais estável. Se o cristal é resfriado suficientemente devagar, os átomos estarão aptos a 'relaxar' na posição mais estável. Este processo de resfriamento lento é conhecido como *annealing*, e então o método ficou conhecido como *Simulated Annealing*.

3.2 Simulação Monte Carlo

A simulação Monte Carlo usa movimentos aleatórios para explorar o espaço de busca, a fim de encontrar alguma informação sobre o espaço. Em uma simulação de Monte Carlo, todos os movimentos aleatórios são aceitos, uma vez que a intenção é percorrer todo o espaço, para depois chegar a alguma conclusão baseada na estatística gerada pela função de restrição [7].

O método é conhecido por este nome devido aos jogos de roleta, um dispositivo gerador de números aleatórios, encontrados nos cassinos do principado de Mônaco [10].

Vejamos um exemplo clássico. Considere o seguinte problema: queremos fazer uma simulação que nos permita encontrar o valor de π. Isso será feito da seguinte forma: considere um quadrado que tem um canto na origem de um sistema de coordenadas e que possui lados de tamanho 1. Agora considere inscrever a quarta parte de um círculo de raio 1 neste quadrado, como mostrado na Figura 2. Sabemos que a sua área é $\pi/4$. Podemos então utilizar a simulação de Monte Carlo para encontrar a área relativa da quarta parte do círculo e quadrado e depois multiplicar a área da quarta parte do círculo por 4 para encontrar o valor de π.

Em particular, o modo que encontraremos a área do círculo é baseado no seguinte: para um ponto (x, y) pertencer a um círculo de raio 1, sua distância da origem $(x^2 + y^2)$ deverá ser menor ou igual a 1. Podemos gerar milhares de pontos (x, y) aleatórios e determinar quando cada um deles está dentro do círculo. Cada vez que um ponto estiver dentro do círculo, iremos somar 1 a um contador. Após gerar um grande número de pontos, a razão entre o número de pontos dentro do crculo e o número total de pontos gerados irá se aproximar da razão entre a área do círculo e a área do quadrado. Conseqentemente, o valor de π será, aproximadamente: $\pi = 4d/f$,

Figura 2. Ilustração da simulação Monte Carlo

onde d = número de pontos dentro da quarta parte do círculo e f = número de pontos gerados. Portanto, podemos encontrar uma aproximação para π com uma matemática simples utilizando-se dos dados gerados através da simulação de Monte Carlo.

3.3 Simulação Metropolis-Monte Carlo

Como sabemos, as simulações de Monte Carlo usam movimentos aleatórios para explorar o espaço de busca no intuito de encontrar alguma informação sobre ele. Em 1953, Nicholas Metropolis e seus colaboradores propuseram um novo procedimento de amostragem, o qual incorporava uma temperatura do sistema.

Isto é feito para que o critério de Boltzmann (veremos adiante) possa ser facilmente aplicado. Este método de Monte Carlo modificado é conhecido como uma simulação Metropolis Monte Carlo [6].

Em muitos sistemas físicos, como por exemplo, os sistemas de partículas, a temperatura desempenha um papel fundamental na dinâmica que pode ser observada. O algoritmo de Metropolis é uma técnica de amostragem importante, ele testa o espaço de busca de acordo com a forma da função que é integrada. O algoritmo de Metropolis começa com um estado aleatório do sistema. Então, tomando como base o estado atual, um novo estado aleatório é gerado e proposto como o novo do estado atual. Este estado proposto é aceito ou não com uma determinada pro- babilidade. O critério de aceitação neste método leva em consideração a importância relativa do estado atual e do novo estado proposto. O estado proposto será aceito, incondicionalmente, se for mais importante, e será aceito, mediante uma condição

probabilística, caso seja menos importante que o estado atual. Assim, tanto o novo estado proposto como o estado atual podem ser usados como o novo estado do sistema [9].

Para descrever uma simulação Metropolis Monte Carlo, é primeiramente necessário esclarecer algumas notações. Dado um momento qualquer da execução do algoritmo, o estado atual do sistema (isto é, sua posição atual no espaço da busca) será chamado estado E_0. Esse estado terá uma energia ϵ_0. Após um movimento aleatório, o estado atual será o estado E_1 com uma energia ϵ_1. Há ainda dois outros parâmetros. O primeiro que chamaremos de nT, indica o número de iterações ocorridas e o outro é um valor entre 0.0 e 1.0, gerado aleatoriamente, o qual chamaremos de R, utilizado no critério de Boltzmann.

No começo de uma simulação de Monte Carlo a variável nT recebe valor zero. O sistema está atualmente em algum estado E_0 e em uma temperatura T, e continua a simulação como segue:

1) altera-se aleatoriamente o estado atual do sistema de modo que esteja agora no estado E_1, certificando-se de que as configurações do estado E_0 e do estado E_1 estão salvas (veja adiante).

2) incrementa-se o parâmetro nT de um. Para determinar se o estado proposto será aceito ou não como o novo estado atual, é necessário comparar as energias dos dois estados. As escolhas possíveis são dadas pelas regras seguintes.

2.1) $(\epsilon_1 \leq \epsilon_0)$
A energia do estado E_1 é menor ou igual à do estado E_0. Isto significa que o novo estado será aceito e irá transformar-se no novo estado E_0.

2.2) $(\epsilon_1 > \epsilon_0)$ e $(e^{\frac{-(\epsilon_1-\epsilon_0)}{T}} \geq R)$
A energia de estado E_1 é maior que a do estado E_0, mas a diferença de energia é pequena o bastante para que possa ser aceita probabilisticamente. Isto significa que o novo estado será aceito e irá transformar-se no novo estado E_0.

2.3) $(\epsilon_1 > \epsilon_0)$ e $(e^{\frac{-(\epsilon_1-\epsilon_0)}{T}} < R)$
A energia de estado E_1 é maior que a do estado E_0, mas a diferença de energia é grande o bastante para que seja rejeitada probabilisticamente. Isto significa que o sistema permanece no estado E_0.

3) para continuar a simulação, basta simplesmente retornar à etapa 1. Se não, pare a simulação.

É importante enfatizar que cada tentativa de Monte Carlo aumenta o valor de nT.

Existem dois fatores que controlam o quão bem a simulação examina o espaço de busca do sistema: o tamanho de cada etapa na simulação, ou seja, a diferença de energia entre o estado E_1 e o estado E_0, e o número de iterações da simulação Metropolis Monte Carlo executadas, nT.

Se o tamanho de uma etapa de Monte Carlo for demasiadamente grande, a diferença na energia entre o estado E_0 e o estado E_1, pode tornar-se muito grande e, virtualmente, todas as tentativas serão probabilisticamente rejeitadas. Isto faz com que a simulação fique 'travada' em um ponto particular no espaço de busca.

Por outro lado, se o tamanho da etapa de Monte Carlo for demasiadamente pequeno, o sistema pode levar um tempo muito grande para testar todo o espaço busca disponível. Em ambos os casos, o número total de tentativas pode se tornar excessivamente grande.

Encontrar um tamanho médio bom para a etapa em determinada temperatura depende do sistema que está sendo modelado. Na prática, a quantidade de iterações de Monte Carlo aceitas pode ser usado como um guia e/ou monitoração de um ou mais dos parâmetros que são variados durante uma etapa. Em último caso, os valores dos parâmetros de cada estado podem dar uma indicação de se o espaço de busca está sendo completamente explorado, ou se a simulação varia entre alguns poucos estados em uma região localizada.

3.4 Simulated Annealing

Uma otimização usando Simulated Annealing começa com uma simulação Metropolis Monte Carlo a uma temperatura bem alta. Devido ao critério de Boltzmann, isto significa que uma porcentagem relativamente alta de estados gerados aleatoriamente que aumentam a energia do sistema serão aceitos.

Após um número suficiente de iterações de Metropolis Monte Carlo, a temperatura é diminuída. A simulação Metropolis Monte Carlo é, então, executada mais uma vez com a nova temperatura. Este procedimento é repetido até a temperatura final ser atingida [8].

Um programa Simulated Annealing consiste em um par de loops aninhados. O loop externo controla a temperatura e o loop interno executa a simulação Metropolis Monte Carlo nesta temperatura. O método pelo qual a temperatura é diminuda é conhecido como 'Cooling Schedule'. Na prática, dois Cooling Schedules são predominantemente usados: o *Cooling Schedule linear* ($T_n = T_a - dT$) e o *Cooling Schedule proporcional* ($T_n = c.T_a$), onde T_n = nova temperatura, T_a = antiga temperatura e $0 < c < 1$ [11].

Vejamos um exemplo da estrutura do algoritmo do Simulated Annealing:

```
Procedimento SA;
   Inicializar(T); nT ← 1;
   Selecionar o estado corrente (E0) aleatoriamente;
   Repetir
      Repetir
         Selecionar novo estado(E1) na vizinhança de E0;
         V0 ← Avalia(E0);   V1 ← Avalia(E1);
         Se (V1 < V0) Ou
            (Aleatorio[0, 1) < e^(-(V1-V0)/T)) Então
               E0 ← E1;
         nT ← nT + 1;
      até (condição de parada dessa simulação);
      T ← g(T, nT);
   até (condição de término);
Fim;
```

Existem algumas características que são essenciais para a eficiência e a eficácia da simulação. A forma como são implementadas estas características varia de acordo como o tipo de problema que se quer solucionar. Vamos ver quais são elas.

3.5 Função que gera um novo estado (busca local)

Existe uma grande variedade de algoritmos de busca local para problemas combinatórios. O mais simples é o chamado 2OPT. Começa com uma permutação aleatória de pontos (chame esse estado de T) e tenta melhorá-lo. A vizinhança de T é definida como o conjunto de todas os estados que podem ser alcançados trocando dois pontos não adjacentes em T. Este movimento é conhecido como '2-interchange' e é ilustrado na Figura 3.

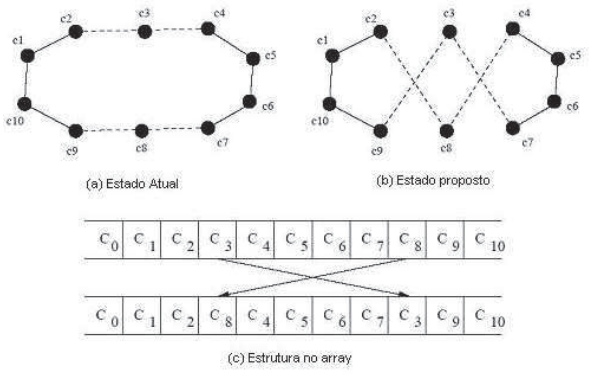

Figura 3. Exemplo de funcionamento de 2OPT

Assim, o novo estado proposto é sempre derivado do estado atual. Este esquema é bem simples e foi usado apenas para exemplificar a busca local. Na implementação dos problemas propostos foi usada uma outra maneira.

3.6 Função que avalia a energia ou custo de um estado

Antes de aceitar um estado existe a necessidade de saber qual é a sua energia, pois será usada no critério de aceitação do novo estado. Esta função é muito importante para a simulação e depende muito do problema em questão.

No caso de problemas combinatórios, ela geralmente consiste em dar pesos para cada par de elementos da sequência em questão.

3.7 Número de iterações por nível de temperatura

Toda temperatura é mantida por algum tempo antes de ser reduzida. De acordo com a analogia a Física, significa que são dadas às moléculas chances de encontrar uma posição entre suas vizinhas, o que produzirá um estado de menor energia total, antes da temperatura diminuir.

Na implementação dos problemas propostos, o número máximo de iterações por nível de temperatura é tratado por dois limites. O primeiro limite é o número máximo de iterações com sucesso, isto é, iterações em que o novo estado proposto foi aceito, que é de 10 vezes o número de elementos da combinação. O segundo limite é o número máximo de iterações por temperatura, tanto com sucesso ou sem sucesso, que é de 100 vezes o número de elementos da combinação.

3.8 Temperatura

De acordo com as descrições físicas do processo de 'annealing', a temperatura inicial deve ser alta o suficiente para que a posição das moléculas no metal líquido seja completamente aleatória [5].

Como isto se traduz para um modelo de algoritmo depende do tipo de problema. Geralmente essa temperatura é derivada do custo do estado inicial do sistema e em muitas implementações é definida como 10% da energia do estado inicial.

Como foi mencionado anteriormente, a temperatura é reduzida após um certo número de iterações. No Simulated Annealing, a temperatura é utilizada para controlar a pro- babilidade de um novo estado proposto, que tenha uma energia ou custo maior do que o estado atual, ser aceito como o novo estado atual. Quanto maior a temperatura, maior a probabilidade e quanto menor a temperatura, menor a probabilidade.

A tabela seguinte exemplifica o comportamento do critério de Boltzmann. $E_0 = 37$ e $E_1 = 50$, logo, $\Delta E = -13$.

Tabela 1. Probabilidade de aceitação do novo estado

T	$e^{\frac{-13}{T}}$	p
1	0.000002	1.00
5	0.0743	0.93
12	0.2725	0.78
20	0.52	0.66
50	0.77	0.56
10^{10}	0.999999	0.5

A conclusão é clara: quanto maior o valor de T, menor a importância da diferença de energia entre os estados comparados E_0 e E_1. Em particular, se T é muito grande (por exemplo, $T = 10^{10}$), a probabilidade de aceitação aproxima-se de 0.5. A procura passa a ser aleatória. Por outro lado, se T muito pequeno (por exemplo, $T = 1$), o novo estado E_1 quase sempre será aceito e o algoritmo perderá seu caráter probabilístico.

Portanto, temos que encontrar um valor apropriado do parâmetro T para um problema particular: nem muito baixo e nem muito alto.

Para termos mais uma perspectiva, suponha $T = 10$ em determinado momento da execução. Vamos supor também que o estado corrente E_0 seja avaliado em 107, isto é, Avalia(E_0) = 107. Então, a probabilidade de aceitação depende somente do valor do novo estado E_1, como mostra a tabela a seguir.

Tabela 2. Probabilidade de aceitação do novo estado

Avalia(E_1)	Δ =Avalia(E_1)- Avalia(E_0)	$e^{\frac{-\Delta}{10}}$	p
80	27	14.88	0.06
100	7	2.01	0.33
107	0	1.00	0.50
120	-13	0.27	0.78
150	-43	0.01	0.99

Analisando a tabela podemos perceber que, se o novo estado tem a mesma energia do estado atual, isto é, Avalia(E_1) = Avalia(E_0), a probabilidade de aceitação é de 50%. Não importa qual será escolhido porque os dois estados são de igual qualidade. Além disso, se o novo estado é melhor, a probabilidade de aceitação é maior que 50%. A probabilidade de aceitação cresce junto com a diferença (negativa) entre estas avaliações.

Em particular, se o novo estado é muito melhor que o estado atual (Avalia(E_1) = 150), a probabilidade de aceitação fica perto de 1.

3.9 Redução da temperatura (Cooling Schedule)

Depois de um número limitado de iterações, com ou sem sucesso, a temperatura é diminuda. A sugestão mais direta de uma função de redução da temperatura é multiplicar T por alguma constante r, $T \leftarrow rT$. Os valores mais comumente usados são $r \in [0.8,\ 0.99]$. Existe um grande número de esquemas de resfriamento, mas nós optamos pela simplicidade. A temperatura inicial junto com a constante r e a temperatura de parada T_F decidem o número de níveis de temperatura (iterações) k. Por exemplo, usando $T_F = 0.00001$, $r = 0.99$, e $T_0 = 20$, teremos $k = 1672$ iterações, isto é, $k = T_F/(rT_0)$. Se o custo da rota ótima é conhecido, ela será adicionada à cláusula que trata do critério de parada.

Outras funções de redução da temperatura podem ser usadas. A Figura 4 ilustra quatro possibilidades. T_i é a temperatura para a iteração i, $0 \leq i \leq N$. As temperaturas inicial, T_0, e final, T_N, são determinadas pelo usuário, bem como o valor de N.

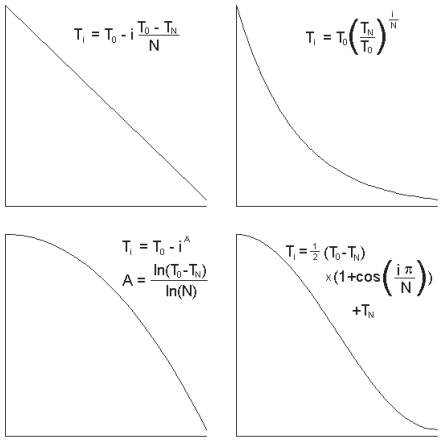

Figura 4. Esquemas de 'Cooling Schedules'

4 Descrição dos problemas

Serão descritos, a seguir, os dois problemas considerados neste trabalho: Percurso do Cavalo e Damas Pacíficas.

4.1 Percurso do Cavalo

O Percurso do Cavalo (Knight Tour) em um tabuleiro de xadrez (ou qualquer outro tabuleiro) consiste de uma sequência de movimentos feitos pela peça de xadrez correspondente ao cavalo, cujo movimento consiste de dois

avanços em relação a um eixo e um avanço em relação ao outro eixo, de tal maneira que cada quadrado do tabuleiro seja visitado exatamente uma vez. Se a posição final do cavalo está a um movimento de distância da posição de partida, temos um caminho reentrante sendo, portanto, um Circuito Hamiltoniano [12].

O Percurso do Cavalo vem sendo estudado há mais de mil anos, e suas origens remontam a países orientais, como China e India [13]. Devemos observar que esse problema é uma pequena variação do Caminho Hamiltoniano, possuindo restrições no acesso às casas. Algoritmos de backtracking (no qual o cavalo se move o mais longe possível até chegar a um beco sem saída, e neste ponto movendo-se para trás um certo número de passos e tentando, então, um caminho diferente) podem ser utilizados para resolver esse problema, mas esses métodos podem ser extremamente lentos [14].

O problema não é difícil em um tabuleiro pequeno. Em um tabuleiro 6*6 o backtracking é uma boa estratégia para resolver o problema, sendo útil, inclusive, para encontrar todas as soluções possíveis do problema. Entretanto, em um tabuleiro comum 8*8, o problema se mostra surpreendentemente complexo, tal que a simples contagem de soluções com backtracking se torna impossível, mesmo para os computadores atuais [15]. No ano de 1997, descobriu-se que o número total de soluções para um tabuleiro 8*8 é de cerca de 13 trilhões. Ainda que utilizemos um computador capaz de encontrar dez milhões de soluções por minuto, ele levaria cerca de 3 anos para encontrar todas as soluções.

Embora o Percurso do Cavalo seja um problema derivado do problema Caminho Hamiltoniano, que pertence à classe NP-Completo de problemas, esse problema, em tabuleiros comuns, pertence à classe P. No ano de 1823, o matemático H. C. Warnsdorff apresentou, no documento intitulado 'Solução simples e genérica para o Percurso do Cavalo', uma heurística simples, mas efetiva, na solução do problema: entre os movimentos possíveis, avançar o cavalo sempre para a posição que permita o menor número de movimentos subsequentes. O objetivo é evitar a criação de 'becos sem saída', quadrados nos quais o cavalo não pode avançar sem chegar a um quadrado já visitado anteriormente; por isso, os próximos movimentos possíveis são examinados antes de cada jogada. Conta-se, então, o número de escolhas possíveis para cada um desses movimentos, e é escolhida a jogada que possui o menor número de movimentos posteriores, evitando assim que ela futuramente se torne um 'beco sem saída' [16].

A heurística de Warnsdorff é extremamente rápida. Entretanto, apresenta alguns pontos negativos. Em primeiro lugar, ela não é totalmente determinística; em algum momento haverá uma indecisão sobre qual será a próxima jogada, pois algumas jogadas terão o mesmo número de movimentos posteriores. Warnsdorff afirmou que, quando uma

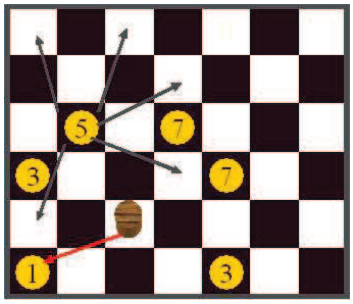

Figura 5. A Heurística de Warnsdorff

situação dessas ocorrer, qualquer uma das rotas mencionadas poderá ser escolhida. Entretanto, existem exemplos para os quais esta regra não funciona. Em segundo lugar, esta heurística não é capaz de encontrar todas as soluções possíveis, ou seja, existem soluções que não obedecem à regra de Warnsdorff. Além disso, a heurística não funciona para tabuleiros muito grandes, mais especificamente de tamanho 76*76 em diante. Tendo em vista esses fatores, devemos considerar a utilização de outras técnicas para solucionar o problema.

4.2 Damas Pacíficas

O problema 'Damas Pacíficas' consiste no posicionamento de oito rainhas do jogo de xadrez em um tabuleiro 8x8, de tal maneira que nenhuma delas consiga atacar qualquer uma das outras sete, utilizando os movimentos adequados da rainha no jogo de xadrez (andar para frente, para trás, para os lados e para as diagonais quantas casas quiser). A cor das peças não tem influência. Sendo assim, uma rainha é capaz de atacar qualquer outra. Uma solução para esse problema requer que não existam duas ou mais rainhas partilhando a mesma linha, coluna ou diagonal no tabuleiro.

Esse problema foi proposto em 1848 pelo enxadrista Max Bazzel [17], e vem sendo estudado desde então por vários matemáticos. Existem diversas heurísticas conhecidas para solucioná-lo, todas elas muito simples e rápidas.

O problema pertence, sem dúvida, à classe P de problemas. Ele possui 92 soluções distintas, mas se contarmos soluções que diferem apenas por operações de simetria (como reflexões e rotações) como se fossem apenas uma solução, então o número de soluções distintas cai para doze. Encontrar todas essas soluções é um bom exemplo de um problema simples, embora longe de ser trivial. Por este motivo, é normalmente utilizado como um problema de demonstração para várias técnicas de programação, incluindo abordagens não tradicionais, como os algoritmos genéticos. Também é bastante utilizado como exemplo de um problema que pode ser solucionado com o auxílio de um

Figura 6. Aplicação para Percurso do Cavalo

Figura 7. Aplicação para Damas Pacíficas

algoritmo recursivo, no qual uma rainha extra é adicionada à solução do problema para $n-1$ rainhas. Essa técnica é muito mais eficiente que o algoritmo de força bruta, que considera todas as n^8 disposições possíveis de oito rainhas, passando então a filtrá-las caso duas rainhas sejam colocadas na mesma posição ou em posições de ameaça mútua. Como alternativas mais eficientes, podemos citar os algoritmos de busca em profundidade, ou o uso de heurísticas.

Veremos a seguir como o Simulated Annealing se sairá na solução dos dois problemas descritos nesta seção.

4.3 Aplicação desenvolvida

Para a resolução desses dois problemas clássicos através da técnica Simulated Annealing, desenvolvemos uma aplicação na plataforma Visual Basic.

Trata-se de um programa simples, dividido em 3 seções principais:

1. Função *SWAP*: Esta função realiza a mudança do estado atual para um de seus estados vizinhos.

2. Função *CUSTO*: Esta função calcula o custo (ou energia) do estado atual.

3. Função *PRINCIPAL*: Este é o bloco principal do programa, onde inicializamos as variáveis correspondentes à temperatura inicial/final, e é nele que está localizado o procedimento do Simulated Annealing em si.

O programa gera aleatoriamente o estado inicial de cada um dos problemas. Para o Percurso do Cavalo, ele popula o tabuleiro com os números de 0 a 63, de maneira aleatória, preenchendo assim os 64 quadrados do tabuleiro 8*8 comum. O objetivo final é que cada número esteja 'ligado' ao seu sucessor por um movimento adequado do cavalo (um avanço em relação a um eixo e dois avanços em relação ao outro eixo).

Para o Damas Pacíficas, oito rainhas são colocadas em posições aleatórias no tabuleiro. O objetivo final é que cada uma delas não seja capaz de 'atacar' nenhuma das outras sete com apenas um movimento.

Para uma análise inicial de como o algoritmo se comporta na resolução desses problemas, utilizamos uma configuração conservadora para cada função, descritas a seguir:

1. *SWAP*: Para o Percurso do Cavalo, a mudança de estado se efetua da seguinte maneira: Dois números, entre 0 e 63, são escolhidos aleatoriamente e suas posições no tabuleiro são trocadas. Para Damas pacíficas, uma das rainhas é escolhida ao acaso, e colocada em uma posição desocupada no tabuleiro de maneira aleatória.

2. *CUSTO*: Para o Percurso do Cavalo, consideramos um custo unitário para cada número que não esteja ligado ao seu sucessor por um movimento do cavalo. Para Damas Pacíficas, consideramos um custo unitário para cada 'ameaça' que ocorra (ou seja, se uma rainha que estiver ameaçando outras 3 rainhas, temos um custo 3). Com essas configurações, assim que o custo zero for alcançado, o problema estará solucionado.

3. Utilizamos nos dois problemas a temperatura inicial com o valor de 10% do custo do primeiro estado gerado, com redução de 1% por iteração. Além disso, estabelecemos o número de loops a cada temperatura com o valor de [100 * número de nós do problema] (que nos dois casos é igual a 64, totalizando 6400 iterações por temperatura), e

[10 * número de nós] para o número de mudanças de estado efetuadas por temperatura. Esses valores são recomendados como valores genéricos para o SA. A princípio não estipulamos um valor para a temperatura final.

Apresentamos a seguir os resultados obtidos quando da execução da aplicação para os dois problemas, nas figuras 8 e 9.

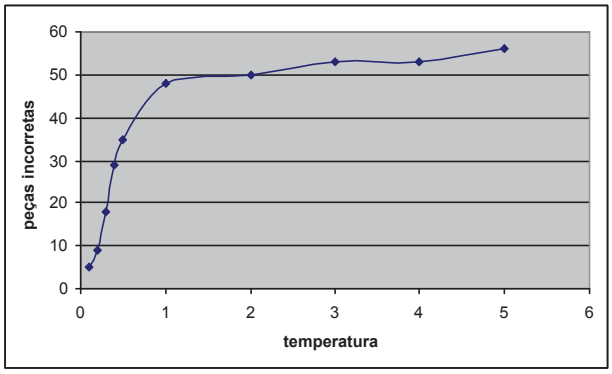

Figura 8. Posições incorretas no Percurso do Cavalo

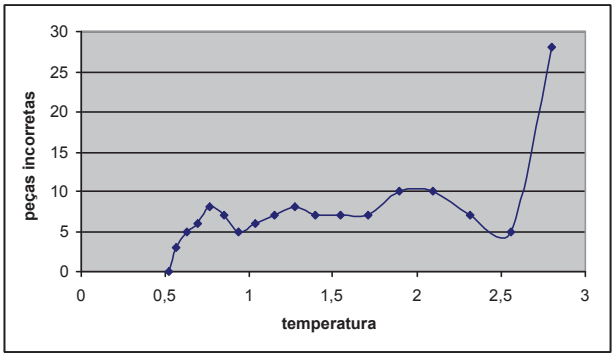

Figura 9. Posições incorretas em Damas Pacíficas

Os gráficos das Figuras 8 e 9 mostram a relação temperatura x peças incorretas utilizando a função *CUSTO* e a função *SWAP* para os problemas Percurso do Cavalo e Damas Pacíficas, respectivamente.

É interessante notar que a abordagem genérica adotada foi suficiente para solucionar o problema das Damas Pacíficas, já que a aplicação resolveu o problema todas as vezes em que foi executado.

Entretanto, ela se mostrou inócua para o Percurso do Cavalo, deixando uma grande quantidade de peças incorretas, na maioria das execuções. Para tentar solucionar esse problema, realizamos uma pequena modificação na função *CUSTO*: o custo de uma jogada incorreta passou a ser $(64 - j)$, onde j indica o número da jogada. A idéia é forçar a ordenação das jogadas de maneira sequencial. Essa modificação foi chamada *CUSTO1* e os novos resultados são mostrados na Figura 10.

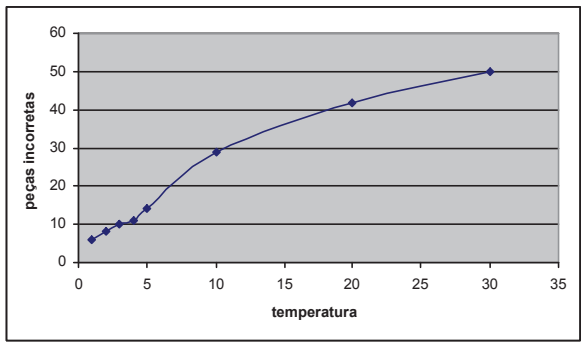

Figura 10. Posições incorretas no Percurso do Cavalo (CUSTO1)

A mudança realizada também não foi efetiva na solução do problema, pois ela apresenta uma eficiência praticamente igual a da abordagem genérica, além de aumentar consideravelmente o tempo de resolução, já que o custo inicial é bem maior do que aquele usado na primeira abordagem.

Testamos outra modificação: usar a função custo genérica, e efetuar alterações na função de troca, passando a realizar uma troca de estado direcionada, conforme o descrito a seguir:

1. Um número entre 0 e 63 é escolhido ao acaso.

2. É feita uma escolha aleatória entre o antecessor e o sucessor do número escolhido no passo anterior, a fim de saber qual dos dois será ordenado (obviamente não existe escolha se o número sorteado for 0 ou 63).

3. A partir da posição no tabuleiro do número escolhido no passo 1, são determinados todos os movimentos possíveis segundo a movimentação do cavalo a partir daquela posição (desde 1 até 8 movimentos), e um deles é selecionado ao acaso.

4. Finalmente, o número que está situado na posição determinada pelo movimento escolhido no passo anterior é trocado de posição com o número escolhido no passo 2.

Os resultados do uso dessa função modificada, (*SWAP1*), são apresentados no gráfico da Figura 11.

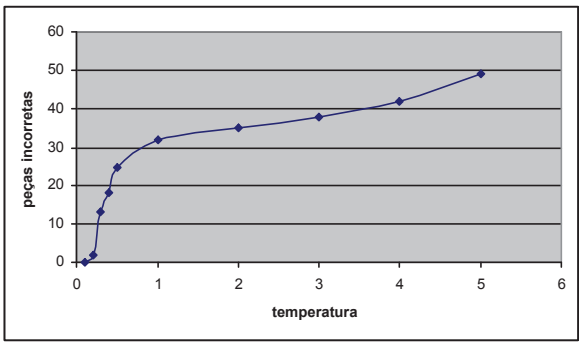

Figura 11. Posições incorretas no Percurso do Cavalo (SWAP1)

A função *SWAP1*, utilizada juntamente com as configurações genéricas de temperatura e iterações, consegue solucionar o problema em algumas das execuções da aplicação. A eficiência do software é de aproximadamente 20% pois, a cada cinco execuções, o problema é solucionado em apenas uma delas.

A utilização de uma meta-heurística como o Simulated Annealing na resolução destes problemas pode trazer essa consequência, pois, como vimos na Seção 1, as meta-heurísticas nem sempre encontram a solução ótima para o problema. Nos dois problemas em que trabalhamos, buscamos apenas uma das soluções ótimas (já que quando o custo for zero as tentativas são interrompidas). Além disso, as configurações genéricas são muito lentas; se executarmos o ajuste cuidadoso das variáveis de inicialização, em conjunto com modificações adicionais nas funções auxiliares certamente é possível diminuir o tempo de resolução dos dois problemas.

4.4 Otimização

A partir de certo ponto, nossa meta foi a de tentar aumentar a taxa de eficiência da solução do Percurso do Cavalo, tentando encontrar uma solução do problema em toda execução. Também tentamos acelerar o tempo de resolução dos dois problemas.

Note que anteriormente utilizamos as configurações genéricas para Damas Pacíficas, já que elas foram suficientes na resolução do problema, enquanto utilizamos a função *SWAP1* para o Percurso do Cavalo. Observamos que o número de peças incorretas em Damas Pacíficas é muito flutuante, variando bastante para cima e para baixo durante a execução da aplicação, até encontrar uma solução. Tipicamente, a solução é encontrada na temperatura por volta de $T = 0.7$, sem apresentar uma taxa aparente de redução, como era esperado.

Já o Percurso do Cavalo apresenta uma taxa gradual de redução, sendo solucionado por volta de $T = 0.15$. Tentamos, em primeiro lugar, reduzir a temperatura inicial para os dois problemas, conforme abaixo:

Percurso do Cavalo: $T = 0.25$

Damas Pacíficas: $T = 0.02$

Os resultados são apresentados nos gráficos das Figuras 12 e 13.

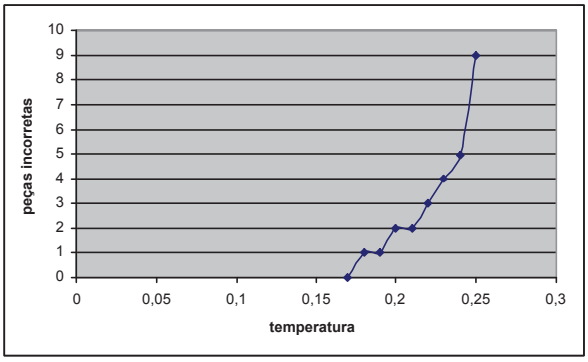

Figura 12. Posições incorretas no Percurso do Cavalo(Ot. 1)

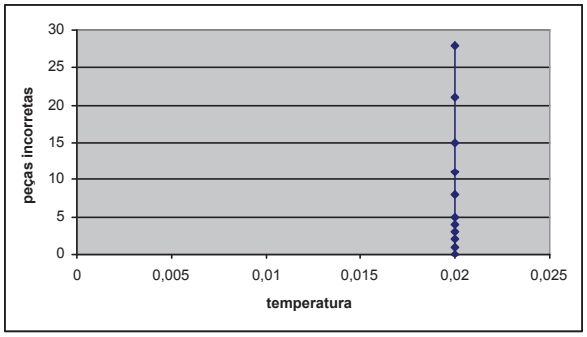

Figura 13. Posições incorretas em Damas Pacíficas(Ot. 1)

A redução da temperatura inicial no Percurso do Cavalo não aumentou a eficiência da aplicação, mas acelerou o tempo de execução, sem prejudicar o resultado final, ou seja, cortou o tempo de execução inútil. Para valores menores de temperatura, a aplicação começou a apresentar diminuição na eficiência, não efetuando mudanças de estado ainda com um número elevado de peças incorretas, ou seja, não conseguindo fugir do ótimo local.

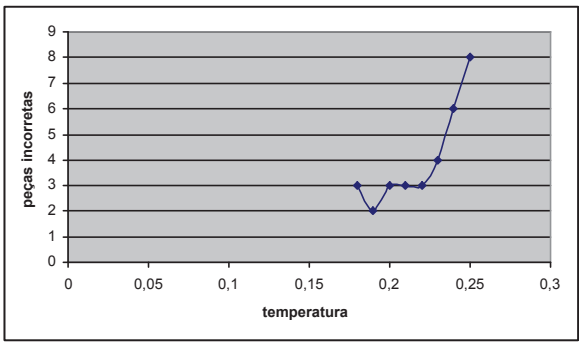

Figura 14. Posições incorretas no Percurso do Cavalo(Ot. 2, T = 0.01)

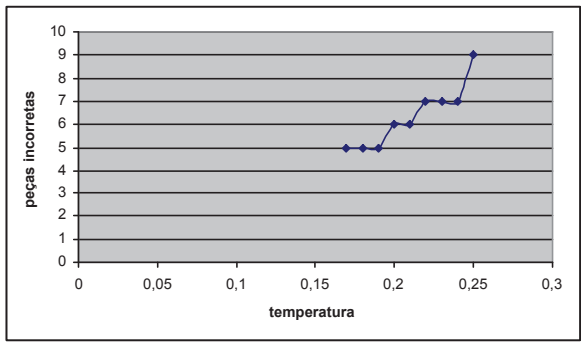

Figura 15. Posições incorretas no Percurso do Cavalo(Ot. 2, T = 1)

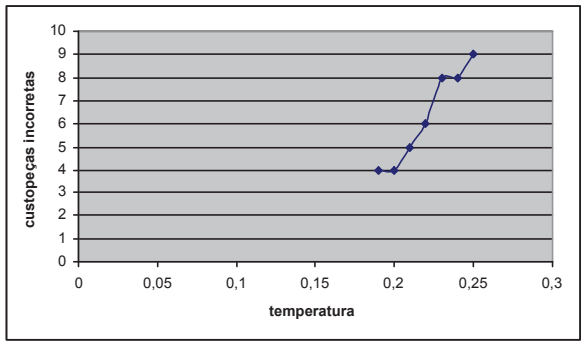

Figura 16. Posições incorretas no Percurso do Cavalo(Ot. 2, T = 10)

Já para Damas Pacíficas, a alteração mostrou-se bastante eficiente, pois o problema passou a ser solucionado quase instantaneamente, e, além disso, continua a ser solucionado toda vez que a aplicação é executada. Isso provavelmente só foi possível graças ao grau de dificuldade desse problema, que é relativamente simples, quando comparado ao Percurso do Cavalo.

Para o Percurso do Cavalo, tentamos, também, criar uma outra função de custo, baseada na heurística de Warnsdorff, discutida anteriormente. A idéia era tentar ocupar prioritariamente os quadrados que possuem menos 'pontos de fuga'. Esta função, que chamaremos *CUSTO2*, é quase idêntica à função de custo genérica, mas considera um custo adicional de acordo com a posição de cada jogada. Nela, o tabuleiro é dividido em 2 setores: O primeiro setor compreende o mini-tabuleiro 4*4 localizado no centro do tabuleiro principal; o segundo setor é a periferia desse mini-tabuleiro central, que compreende os 48 quadrados restantes. As jogadas de números 48 a 63 devem ocupar os quadrados do setor 1, enquanto as jogadas de números 0 a 47 devem ocupar os quadrados do setor 2. Caso uma jogada esteja fora do setor apropriado, um certo custo é adicionado. Podemos utilizar valores altos de custo, para 'forçar' a priorização do posicionamento adequado das jogadas, ou valores baixos, para fazê-lo adequar o posicionamento das peças de maneira menos direta. Utilizamos 3 custos diferentes: 0.1, 1 e 10. Os gráfico das Figura 14 a 16 mostram os resultados obtidos para cada uma das três alternativas de custo respectivas.

Pode ser observada a ineficiência das estratégias de custo muito elevadas, como 10 e 1, que acabam por restringir o espaço de troca de estados, já que às vezes é preciso que as jogadas iniciais ocupem posições proibidas no tabuleiro. Entretanto, para valores baixos a estratégia se torna inócua, já que o software praticamente não considera seu baixo custo durante a execução, só vindo a fazê-lo quando a tem-
peratura se encontra muito baixa, e não há mais espao para trocas.

Valores ainda inferiores, como 0.01 são praticamente desconsiderados, e apresentaram eficiência equivalente à da função custo padrão.

Uma última tentativa foi a de modificar a inicialização, tal que, ao invés de gerar o estado inicial de forma totalmente aleatória, tentou-se criar o maior número de casas em sequência. Esta tentativa não trouxe qualquer melhoria nos resultados.

5 Conclusões

Existe um potencial bastante elevado para evolução no campo dos algoritmos probabilísticos e a meta-heurística *Simulated Annealing* é, sem dúvida, uma ferramenta poderosa na tentativa de contornar os problemas intratáveis.

Neste trabalho demonstramos a versatilidade desta técnica na solução de problemas com tempo de resolução poli-

nomial, embora sua utilidade maior seja, efetivamente, na resolução de problemas ditos difíceis. Percebemos que SA pode ser uma alternativa interessante para as heurísticas existentes na resolução de problemas fáceis, embora dificilmente seja mais rápido que elas; nesse caso sua vantagem reside na facilidade de alteração para resolver problemas distintos.

Na abordagem efetuada para os dois problemas clássicos, o Percurso do Cavalo e Damas Pacíficas, o SA mostrou-se ótimo para o segundo, enquanto insuficiente para o primeiro. O uso de SA conseguiu resolver Damas Pacíficas em tempo médio inferior a 1 segundo na máquina utilizada, o que o torna no máximo marginalmente mais lento que a heurística correspondente, não fazendo diferença em termos práticos. É digno de nota observar que esse resultado foi conseguido com as configurações genéricas das funções de troca e custo, além das variáveis de inicialização, o que indica que provavelmente haveria espaço para maior otimização de tempo de resolução, caso fosse necessário.

Quando utilizado para resolver o Percurso do Cavalo, entretanto, a técnica exibe certas deficiências. Ela se mostra lenta para resolver o problema, e sua eficiência na resolução também mostrou-se insatisfatória, quando comparada à heurística de Warnsdorff, a melhor existente até o momento. Uma caracterstica notável foi a tendência em formar caminhos distintos no tabuleiro, fenômeno observado também na utilização de outros algoritmos na resolução deste problema, como os algoritmos genéticos. Várias modifica- ções foram efetuadas nas funções e nos valores das variáveis de inicialização com o intuito de acelerar o tempo de resolução e a eficiência da aplicação. Somente o primeiro objetivo foi alcançado, o que não se constituiu em um grande resultado.

Apesar disso, este resultado não pode de maneira nenhuma ser considerado conclusivo, pois é certo que existam funções de troca e de custo ainda mais apropriadas para a resolução deste problema, embora suas alterações sejam tão determinísticas para o problema, que passam a corromper a característica básica de SA, que é a sua aleatoriedade.

Como sugestões de trabalhos posteriores na área, podemos citar a utilização do SA na resolução desses problemas em tabuleiros maiores e menores que o de 8*8, além de tabuleiros em 3 dimensões, o que pode trazer luz a novas maneiras de solucioná-los, além de também contornar a deficiência da heurística de Warnsdorff para tabuleiros muito grandes. A busca de caminhos reentrantes para o Percurso do Cavalo é também um adendo interessante para o trabalho realizado. Finalmente, pode-se considerar a resolução de problemas derivados de Damas Pacíficas, utilizando outras peças, como o próprio cavalo, por exemplo.

Referências

[1] C. H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1984.

[2] S. Kirkpatrick Jr, C. Gelatt, M. Vecchi. *Optimization by Simulated Annealing* In *Decision Science*, v. 220, n.4598, pp. 498-516, 1983.

[3] N. Metropolis, A. W.Rosenbluth, M. N. Rosenbluth, A. H.Teller, E. Teller. *Equation of State Calculations by Fast Computing Machines* In *J. Chem. Phys*, 21, pp 1087-1092, 1953. Addison Wesley, 1984.

[4] Herbert S. Wilf. *Algorithm and Complexity*. Online em http://www/cis.upenn.edu/wilf, 01 de agosto de 2005.

[5] Christian Blum; Andrea Roli. *Simulated Annealing Page*. Online em http://www.nist.gov/cgi-bin/exit-nist.cgi, 01 de agosto de 2005.

[6] Brian T. Luke. *Metropolis Monte Carlo Simulation*. Online em http://members.aol.com/btluke/metro01.htm, 01 de agosto de 2005.

[7] Brian T. Luke. *Simple Monte Carlo Simulation*. Online em http://members.aol.com/btluke/smcsim.htm, 01 de agosto de 2005.

[8] Brian T. Luke. *Simulated Annealing*. Online em http://members.aol.com/btluke/featur05.htm, 01 de agosto de 2005.

[9] Franz J. Vesely. *Monte Carlo Method*. Online em http://homepage.univie.ac.at/Franz.Vesely/cp0102/dx/node99.html, 01 de agosto de 2005.

[10] Sabri Pllana. *History of Monte Carlo method*. Online em http://www.geocities.com/CollegePark/Quad/2435/history.html, 01 de agosto de 2005.

[11] Brian T. Luke. *Simulated Annealing*. Online em http://fconyx.ncifcrf.gov/lukeb/simann1.html, 01 de agosto de 2005.

[12] Paulo Feofiloff. *Ciclos hamiltonianos*. Online em http://www.ime.usp.br/pf/mac5827/aulas/hamilton.html, 01 de agosto de 2005.

[13] George Jelliss. *Early History of Knights Tours*. Online em http://www.ktn.freeuk.com/1a.htm, 01 de agosto de 2005.

[14] Eric W. Weisstein. *Knights Tour*. Online em http://mathworld.wolfram.com/KnightsTour.html, 01 de agosto de 2005.

[15] Gunno Tornberg. *Knights Tour*. Online em http://web.telia.com/u85905224/knight/eknight.htm, 01 de agosto de 2005.

[16] Gunno Tornberg. *Warnsdorffs Rule*. Online em http://web.telia.com/u85905224/knight/eWarnsd.htm, 01 de agosto de 2005.

[17] Wikipedia. *Eight Queens Puzzle*. Online em http://en.wikipedia.org/wiki/Eight-queens-puzzle, 01 de agosto de 2005.