MAC5758
Introdução ao Escalonamento e Aplicações

Estudo sobre as Metaheurísticas

Leandro Ferro Luzia - 5118636
Mauricio Chui Rodrigues - 5122930

Índice

Capítulo I

Introdução

Otimização

É interessante que alguns fatos e conceitos sobre otimização sejam conhecidos antes de se apresentar a definição de metaheurística. A seguir está uma breve descrição desses pré-requisitos, de forma que mais detalhes devem ser consultados em suas respectivas fontes.

Otimização Estocástica

Otimização, ou programação matemática, se refere a escolher o melhor elemento em um conjunto de alternativas disponíveis, por meio de uma função objetivo específica que apresenta ou se aproxima de um valor desejado [4]. O melhor elemento encontrado pode variar de um simples valor inteiro que maximiza ou minimiza uma função de variável real até estruturas complexas, como o melhor arranjo de um time de robôs que leva ao maior número de gols em uma partida de futebol.

A técnica mais antiga de otimização é conhecida como *steepest descent*, ou gradiente descendente, e foi desenvolvida por Gauss. Nos anos 40, o termo "programação linear" foi introduzido por George Dantzig, matemático criador do algoritmo *simplex*.

Atualmente, a área de otimização é composta por diversos campos, porém há algumas intersecções entre seus subcampos. Algumas sub-áreas são:

- **Programação Linear**: a função objetivo é linear e há um conjunto de restrições relacionadas às variáveis da função, que são especificadas como igualdades ou desigualdades lineares;
- **Programação Inteira**: apresenta problemas geralmente mais complexos que os de programação linear, com variáveis que se restringem a valores inteiros;
- **Otimização Combinatória**: o conjunto de soluções factíveis é ou pode ser reduzido a um conjunto discreto;
- **Otimização Estocástica**: estuda os casos em que as restrições ou os parâmetros dependem de variáveis aleatórias.

As metaheurísticas estão inclusas nos métodos de otimização estocástica e, para justificar seus usos, primeiro será abordado o método do gradiente, uma técnica geral e de suma importância nos estudos de otimização.

Otimização Baseada em Gradiente

O método do gradiente é um tradicional algoritmo de aproximação de primeira-ordem. A idéia do algoritmo é identificar a inclinação da função em um ponto inicial, seguindo então a função em direção ao objetivo (máximo ou mínimo da função), por meio de incrementos ou decrementos proporcionais ao gradiente da função [2].

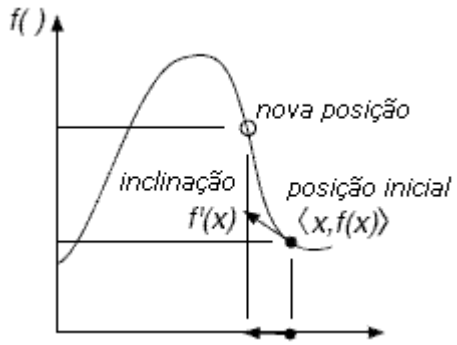

Figura 1: O incremento em x é proporcional à inclinação
da curva da função no ponto inicial

A Figura 1 ilustra o caso simples de uma função de uma variável, em que o gradiente é a derivada da função. As funções objetivo são geralmente multidimensionais, com sua inclinação representada pelo gradiente (representado por $\nabla f(\vec{x})$).

O algoritmo do gradiente é simples, como pode ser visto no Algoritmo 1.

1: inicialize $\vec{x}$ com um valor fornecido ou aleatório
2: repita
3: $\quad \vec{x} \leftarrow \vec{x} + \alpha \cdot \nabla f(\vec{x})$
4: até que $\vec{x}$ seja a solução ideal ou que o tempo tenha se esgotado
5: devolva $\vec{x}$

Algoritmo 1: Algoritmo do Gradiente

Para determinar se o candidato à solução $\vec{x}$ é ideal, em geral analisa-se o gradiente da função - um gradiente igual a $\vec{0}$ indica que a função atingiu o seu máximo ou mínimo.

Um problema com este algoritmo é que, ao se aproximar do ponto máximo, o valor de $\vec{x}$ oscila em torno do ponto devido à variação de sinal do gradiente, o que pode consumir um tempo considerável até que o valor procurado seja atingido. O tempo de convergência está normalmente relacionado ao coeficiente α da equação, ajustável pelo usuário.

Outro grave problema com este algoritmo é que não somente os máximos ou mínimos da função possuem o gradiente igual a $\vec{0}$. Máximos e mínimos locais e pontos de inflexão também apresentam esse valor para o gradiente, como pode ser visto na Figura 2. Estes falsos pontos podem ser erroneamente tomados como a resposta que maximiza/minimiza a função.

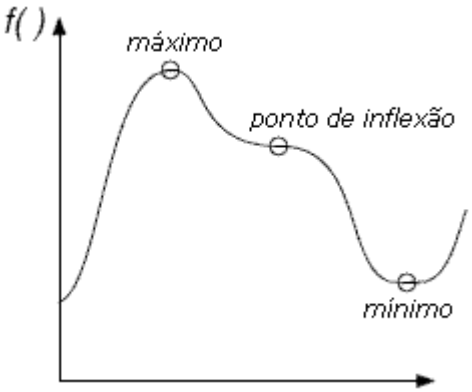

Figura 2: O gradiente de uma função é $\vec{0}$
em máximos, mínimos e pontos de inflexão

Outros métodos, como o de Newton, utilizam outras informações, como a segunda derivada da função (ou, no caso de funções multidimensionais, a matriz Hessiana), para ter uma convergência mais rápida do algoritmo. Porém, isso não resolve o problema central do método do gradiente - a armadilha dos ótimos locais.

Há alternativas baseadas no método do gradiente para escapar dos ótimos locais, mas necessitam de um tempo de execução muito grande, em geral proibitório. É neste contexto que entram abordagens menos determinísticas, que aproveitam elementos aleatórios para evitar a armadilha dos ótimos locais. Entre tais abordagens estão as metaheurísticas, definidas na próxima seção.

Definição de Metaheurística

A área que estuda as metaheurísticas é considerada um subcampo primário da área de otimização estocástica, classe geral de algoritmos e técnicas que empregam algum grau de aleatoriedade para encontrar soluções tão ótimas quanto possível para problemas reconhecidamente difíceis [2].

Segundo a definição original, metaheurísticas são métodos de solução que coordenam procedimentos de busca locais com estratégias de mais alto nível, de modo a criar um processo capaz de escapar de mínimos locais e realizar uma busca robusta no espaço de soluções de um problema [3].

Posteriormente, a definição passou a abranger quaisquer procedimentos que empregassem estratégias para escapar de mínimos locais em espaços de busca de soluções complexas. Em especial, foram incorporados procedimentos que utilizam o conceito de vizinhança para estabelecer meios de fugir dos mínimos locais. Uma metaheurística, portanto, visa produzir um resultado satisfatório para um problema, porém sem qualquer garantia de otimalidade.

Metaheurísticas são aplicadas para encontrar respostas a problemas sobre os quais há poucas informações: não se sabe como é a aparência de uma solução ótima, há pouca informação heurística disponível e força-bruta é desconsiderada devido ao espaço de solução ser muito grande. Porém, dada uma solução candidata ao problema, esta pode ser testada e sua otimalidade, averiguada.

Heurísticas Cobertas pelo Trabalho

O objetivo neste trabalho é estudar e descrever as seguintes metaheurísticas, abordando itens como definição, vantagens e desvantagens de uso, exemplos de aplicação e, quando possível, algumas formas de implementação.

- *Best-first Search* (*Greedy* BFS e A*)
- *Hill Climbing*
- *Tabu Search*
- *Simulated Annealing*
- *Ant Colony Optimization*
- GRASP
- *Memetic Algorithms*
- *Genetic Algorithms*
- *Particle Swarm Optimization*
- *Harmony Search*

Optou-se por não traduzir os nomes das metaheurísticas para o português, visto que nem todas apresentam tradução exata para o idioma.

Organização do Trabalho

As metaheurísticas foram divididas em grupos de acordo com suas propriedades. A classificação utilizada não é oficial e foi adotada apenas para melhor organização.

O Capítulo 2 apresenta as metaheurísticas que correspondem a estratégias de busca informada e ainda as de busca local. O Capítulo 3 contém as estratégias ditas construtivas. O Capítulo 4 introduz os métodos baseados em população. O Capítulo 5 descreve metaheurísticas baseadas em música. Por fim, no Capítulo 6 consta a conclusão dos autores sobre o estudo.

Capítulo II

Estratégias de Busca Informada e de Busca Local

Best-first Search

Definição

A ***Best-first Search*** **(BFS)** é uma abordagem geral de estratégia de busca informada, ou seja, ela aproveita informações específicas do problema que vão além do problema em si [5]. Tais informações são determinadas por uma função $f(n)$, que é aplicada sobre nós de um grafo (cada nó representa um estado possível na solução do problema).

O seguinte princípio é a base da **BFS**: devem ser expandidos e analisados apenas os estados mais promissores, com maior valor para $f(n)$. Mais precisamente, apenas o estado mais promissor deve ser avaliado, por estar mais próximo do estado desejado, chamado de meta. Obviamente, se isso fosse possível, não se trataria de uma estratégia de busca, mas de um caminho direto até a meta. Assim, o mais correto a se afirmar é que o estado escolhido é aquele que aparenta ser o mais próximo da meta.

Na área de escalonamentos, todo estado corresponde a um escalonamento com um número qualquer de tarefas. Já o termo "meta" é o mesmo que "escalonamento ótimo". Pode haver mais de uma meta para um determinado problema, da mesma forma que pode haver mais de um escalonamento ótimo.

Segundo [6], **BFS** pode ser mais precisamente definida como a estimativa de quão promissor é um estado com base na avaliação de uma função $f(n)$, a qual geralmente depende não só de n, mas da descrição da meta, das informações obtidas até tal ponto e de toda e qualquer informação extra sobre o domínio do problema.

Há uma família inteira de algoritmos **BFS**, cada um com uma diferente função. Neste estudo, serão avaliados os dois algoritmos mais conhecidos: ***Greedy*** **BFS** e **A***.

Greedy Best-first Search

O ***Greedy*** **BFS** é descrito em [5] como um algoritmo com função $f(n) = h(n)$, onde $h(n)$ é o custo estimado do melhor caminho de n até a meta. Isto significa que, para determinar o valor de um nó, o único fator considerado é a diferença estimada do mesmo para a meta.

Segundo [5], tal definição é empregada por alguns autores como a definição da própria **BFS**. Outro nome possível para o algoritmo é ***Greedy Search***. O termo "*greedy*" é o mesmo empregado em "*greedy algorithms*", ou "algoritmos gulosos".

Este algoritmo se comporta como uma busca em profundidade: um caminho do grafo é percorrido o máximo possível, da raiz até as folhas, e, caso a meta não tenha sido atingida e não seja mais possível seguir no caminho, outro caminho é escolhido para continuar a busca. O consumo do algoritmo, no pior caso, é $O(b^m)$, onde b é o fator de ramificação médio e m é a maior profundidade no espaço de buscas.

Na área de Inteligência Artificial, o ***Greedy*** **BFS** é considerado um algoritmo não-ótimo e incompleto. A classificação como não-ótimo se deve à definição de otimalidade em tal área: "solução ótima é aquela com o caminho de menor custo entre todas as soluções" [5]. Já a classificação como

incompleto tem como causa a incapacidade do algoritmo de escapar de um caminho infinito no grafo (o que ocorre quando há estados revisitados infinitamente ao longo do caminho).

É importante notar que, considerada a busca de soluções para escalonamentos, o algoritmo é ótimo. Neste caso, não há a condição de custo imposta pela área de IA, apenas há o interesse em se atingir uma meta. Além disso, um mesmo estado não deve aparecer duas vezes em um caminho quando se trata de escalonamentos, pois os sucessores de um estado S sempre apresentam uma tarefa a mais do que S e o número de tarefas é finito. Assim, não há o risco de o ***Greedy* BFS** seguir infinitamente num mesmo caminho.

A* *Search*

O tipo mais conhecido e empregado de **BFS** é o algoritmo **A***, proposto em 1968 por Peter Hart, Nils Nilsson e Bertram Raphael [12]. Ele utiliza a combinação de duas funções para avaliar os nós: $g(n)$, o custo para se chegar até o nó, e $h(n)$, o custo estimado do melhor caminho de n até a meta. Portanto, $f(n) = g(n) + h(n)$ é dito o custo estimado da melhor solução que passa por n.

Embora o comportamento do **A*** seja similar ao do ***Greedy* BFS**, a definição de $f(n)$ proporciona notáveis melhorias. Sob determinadas condições, o **A*** é ótimo e completo na área de IA. Mais do que isso, também é otimamente eficiente para qualquer heurística empregada, ou seja, não há outro algoritmo ótimo que garanta expandir menos nós do que o **A*** sem correr o risco de perder a solução ótima. Como a prova dessas propriedades foge do escopo deste trabalho, sugere-se a consulta a [5] para maiores informações.

Uma propriedade interessante do algoritmo, inclusive na busca por soluções para escalonamentos, é a da poda de sub-árvores do grafo de estados. Dada uma heurística admissível, ou seja, que nunca superestima o valor real, o **A*** nunca expande os nós com $f(n) > C^*$, onde C^* é o custo do caminho para a solução ótima. A implicação disso é que, se um nó apresenta valor que ultrapassa o custo ótimo, não há a expansão da sub-árvore da qual é raiz.

Vale notar que, apesar das melhorias em relação ao ***Greedy* BFS**, o **A*** mantém a expansão exponencial de nós para a grande maioria das heurísticas. Frequentemente, o estouro de memória impossibilita que sejam encontradas soluções ótimas para diversos problemas de larga escala.

Formas de Implementação

Um algoritmo básico para se implementar as variações de **BFS** é o Algoritmo 2, considerando-se os nós em um grafo e supondo que cada nó corresponda a um estado. Há muitas outras formas de implementação, como, por exemplo, as que visam otimizar o consumo de memória durante a execução.

```
1: comece com nós = {estado-inicial}
2: enquanto nós ≠ ∅ faça
3:     escolha o nó com o melhor valor para f(n)
4:     se o nó escolhido for a meta, então
5:         devolva o nó
6:     senão
7:         encontre os sucessores do nó
8:         defina o valor de cada sucessor para f(n) e adicione-os a nós
```

Algoritmo 2: Algoritmo básico para se implementar BFS

Vantagens e Desvantagens

A vantagem de se aplicar **BFS** como metaheurística na busca de soluções para escalonamentos é que, se a função que atribui valores aos nós for bem escolhida e os valores estimados forem próximos dos reais, o algoritmo encontra um escalonamento ótimo e não precisa de muito tempo para isso quando os problemas são pequenos ou médios. Além disso, trata-se de um algoritmo bastante intuitivo.

A maior desvantagem está na expansão exponencial de nós ao longo da busca, o que faz com que os algoritmos não se apliquem a problemas grandes e precisem de otimizações. Em alguns casos, otimizar significa sacrificar propriedades dos algoritmos em troca de uma solução que nem sempre é a ótima. Também pode ser complexa a tarefa de encontrar uma heurística admissível e precisa para definir $f(n)$.

Uma desvantagem relacionada ao ***Greedy* BFS** é a característica de busca exaustiva, pois é possível haver uma varredura completa do grafo, percorrendo todos os caminhos, enquanto uma solução ótima não for encontrada. Conforme descrito anteriormente, isso não ocorre com o **A*** devido às podas de sub-árvores desnecessárias.

Exemplos de Aplicação

Alguns estudos sobre escalonamentos abordam a **BFS** apenas para comparar resultados com algoritmos propostos, como é o caso de [13]. Já os que realmente empregam esse tipo de busca em seus estudos parecem optar pelo **A***.

Um exemplo de aplicação do **A*** para resolver o problema *Job-Shop* é [14], no qual os autores definem uma heurística e buscam testar os limites da otimalidade do algoritmo. Eles ainda propõem meios de se atingir soluções sub-ótimas quando não é possível encontrar soluções ótimas dentro de um limite de tempo.

Outro exemplo é [15], dos mesmos autores e ainda considerando o *Job-Shop*. Um dos objetivos deste recente estudo é demonstrar que o **A*** com "podas por dominância" pode atingir escalonamentos ótimos que não são encontrados pelo **A*** normal em um problema proposto. Há ainda a comparação com algoritmos genéticos.

Job-Shop é um problema reconhecidamente difícil da área de escalonamento, em que serviços (*jobs*) ideais são atribuídos a recursos em determinados momentos no tempo. Cada *job* possui um conjunto de tarefas que devem ser executadas com recursos específicos, durante um período fixo de tempo.

Hill Climbing

Definição

Hill Climbing é uma técnica simples de busca local (não armazena o caminho percorrido até a solução corrente e sim a solução propriamente dita como estado), relacionada ao método do gradiente, que não requer que seja conhecido o gradiente ou sua direção - novos candidatos são gerados posteriormente na região da solução atual.

O algoritmo inicia com uma solução randômica, potencialmente ruim, e iterativamente efetua pequenas modificações nesta solução, buscando melhorias no resultado da função objetivo. O algoritmo termina quando não encontra nenhuma melhoria possível em uma iteração. A solução, ao término do algoritmo, é idealmente ótima, mas não há qualquer garantia de otimalidade.

Há basicamente três variações para o algoritmo de ***Hill Climbing***. A versão mais básica será exposta a seguir, para comparação com o método do gradiente. As duas outras variações serão descritas na próxima seção.

```
1: S ← solução inicial
2: repita
3:     R ← NovaSolução(S)
4:     se (Qualidade(R) > Qualidade(S)) então
5:         S ←R
6: até que S seja a solução ideal ou o tempo tenha se esgotado
7: devolva S
```

Algoritmo 3: *Hill Climbing*, versão básica

No Algoritmo 3, a função NovaSolução efetuará alguma pequena modificação ao seu parâmetro, gerando assim uma nova solução candidata.

Pode-se notar a semelhança com o método do gradiente. Essencialmente, a única diferença está em como se calcula R, a nova solução candidata. No caso do método do gradiente, R é calculado a partir de S, utilizando um valor proporcional ao gradiente. No ***Hill Climbing***, R é obtido por alguma transformação arbitrária, em geral associada a uma abordagem estocástica.

Formas de Implementação

Baseadas no Algoritmo 3, existem diversas variações para o ***Hill Climbing***. Uma é o *Steepest Ascent Hill Climbing*, que utiliza uma abordagem similar à do ***Best-first Search*** ao explorar os sucessores da solução atual e escolher o que oferecer o melhor resultado.

```
01: n ← número de extensões a serem geradas
02: S ← solução candidata inicial qualquer
03: repita
04:     R ← NovaSolução( S )
05:     repita n - 1 vezes
06:         W ← NovaSolução( S )
07:         se Qualidade(W) > Qualidade(R) então
08:             R ← W
09:     se Qualidade(R) > Qualidade(S) então
10:         S ← R
11: até que S seja a solução ideal ou o tempo tenha se esgotado
12: devolva S
```

Algoritmo 4: *Steepest Ascent Hill Climbing*

Outra variação, denominada *Hill Climbing with Random Restart*, visa forçar o ***Hill Climbing*** a cobrir uma maior área do espaço de busca. NovoTempo() é uma função que obtém um número para o período a partir de uma distribuição de probabilidade qualquer fornecida pelo usuário.

```
01: S ← solução candidata inicial qualquer
02: Melhor ← S
03: repita
04:     período ← NovoTempo()
05:     repita
06:         R ← NovaSolução( S )
07:         se Qualidade(R) > Qualidade(S) então
08:             S ← R
09:     até que S seja ideal, ou que período ou o tempo total tenham se esgotado
10:     se Qualidade(S) > Qualidade(Melhor) então
```

```
11:        Melhor ← S
12:     S ← solução candidata aleatória
13: até que S seja a solução ideal ou o tempo tenha se esgotado
14: devolva Melhor
```

Algoritmo 5: *Hill Climbing with Random Restart*

A idéia do Algoritmo 5 é executar o ***Hill Climbing*** por um certo tempo e então efetuar uma busca aleatória (passo 12), reiniciando o algoritmo com a solução aleatória. Isto faz com que o *Hill Climbing with Random Restart* seja classificado entre um método de busca local e um de busca global.

Se o tempo obtido no passo 4 for muito grande, o algoritmo se comportará essencialmente como o ***Hill Climbing*** convencional. Por outro lado, se for muito pequeno, será basicamente uma busca aleatória no espaço de solução. Esta flexibilidade revela ser muito útil em diversos casos.

Vantagens e Desvantagens

A principal vantagem do ***Hill Climbing*** é sua facilidade de implementação. Outra vantagem está no fato de ser utilizado como base para o desenvolvimento de diversos outros métodos mais sofisticados, que apresentam melhores resultados. Portanto, esta técnica deve ser estudada e vista com um ponto de partida, sobre o qual diversas outras metaheurísticas foram desenvolvidas.

Uma desvantagem deste método é a possibilidade de ficar preso em um máximo ou mínimo local que possua uma região de vizinhança muito grande, ou seja, a efetividade do método depende muito da função em que está sendo aplicado. Regiões planas podem fazer com que o método termine prematuramente, pois não encontra nenhuma melhoria nos arredores do ponto em que está trabalhando.

Exemplos de Aplicação

Encontraram-se referências ao uso do ***Hill Climbing*** em diversos problemas de escalonamento. Porém, aparentaram ser mais interessantes aqueles que tentam resolver questões relacionadas ao uso de recursos computacionais em sistemas distribuídos ou multiprocessados, conforme descrito em [10] e [11].

Tabu Search

Definição

Proposta por Fred Glover em 1986, a ***Tabu Search*** é um método de busca local que utiliza o conceito de tabu para tentar encontrar a melhor solução para uma função objetivo. Glover a considera uma metaheurística por atuar como uma estratégia geral para guiar e controlar heurísticas específicas do problema em questão.

A ***Tabu Search*** mantém uma lista de soluções candidatas visitadas recentemente (Lista Tabu) e se recusa a revisitar estas candidatas até que um determinado tempo se passe. Quando um máximo (mínimo) é encontrado, força-se uma busca para um ponto distante do corrente, pois não é permitida a permanência ou retorno para o máximo (mínimo). Isto faz com que uma área mais abrangente do espaço de soluções seja visitada.

Para cada problema tratado por esta metaheurística, é importante definir três elementos: espaço de busca, vizinhança e tabu.

O espaço de busca é o espaço de todas as possíveis soluções que podem ser consideradas durante a busca. Existem maneiras diferentes de se definir um espaço de busca para um determinado problema e, em algumas ocasiões, espaços "não-triviais" devem ser considerados de forma que a ***Tabu Search*** encontre uma boa solução.

Vizinhança é um elemento fortemente relacionado ao espaço de busca. A cada iteração da ***Tabu Search***, as modificações que podem ser aplicadas na solução atual formam um conjunto de soluções vizinhas no espaço de busca, definindo a vizinhança do ponto transformado. Para cada espaço de busca considerado para um problema, diferentes vizinhanças deverão surgir.

Tabus são elementos que previnem visitas cíclicas no espaço de busca quando se está tentando fugir de um máximo (mínimo) local e nenhuma transformação aplicada traz melhorias à solução atual. A idéia-chave para resolver este problema é proibir (considerar tabu) movimentos que revertam o efeito de movimentos recentes.

Os tabus são armazenados em *short-term memory* (Lista Tabu) e normalmente possuem um número fixo e limitado de entradas. Para cada problema, há diversas possibilidades para o que pode ser considerado e armazenado como tabu, devendo cada uma ser considerada em relação ao problema em questão e a como será a ação com cada tabu definido.

Formas de Implementação

O Algoritmo 6 armazena as melhores soluções encontradas recentemente na Lista Tabu.

```
01: n ← número de vizinhos a serem gerados
02: S ← solução candidata inicial qualquer
03: Melhor ← S
04: L ← lista tabu vazia
05: insira S em L
06: repita
07:     R ← GerarVizinho( S )
08:     repita n - 1 vezes
09:         W ← GerarVizinho( S )
10:         se W não está em L e (Qualidade(W) > Qualidade(R) ou R está em L) então
11:             R ← W
12:     se R não está em L e Qualidade(R) > Qualidade(S) então
13:         S ← R
14:         insira R em L
15:     se Qualidade(S) > Qualidade(Melhor) então
16:         Melhor ← S
17: até que Melhor seja a solução ideal ou o tempo tenha esgotado
18: devolva Melhor
```

Algoritmo 6: *Tabu Search*

No Algoritmo 6, GerarVizinho e Qualidade são fornecidos pelo usuário à ***Tabu Search*** e devem, respectivamente, aplicar transformações a uma solução, criando um novo ponto no espaço de busca, e gerar uma medida de avaliação de quão boa é a solução gerada por um ponto no espaço. A Lista Tabu L possui tamanho limitado: quando um novo elemento é inserido e o tamanho especificado é ultrapassado, remove-se o elemento mais antigo na lista.

Vantagens e Desvantagens

Trata-se de uma abordagem algorítmica poderosa que tem sido aplicada com grande sucesso a muitos diferentes e difíceis problemas combinatoriais. Um característica positiva é a capacidade de

lidar com restrições complicadas que surgem em problemas da vida real, sendo portanto uma abordagem de uso realmente prático.

Um grande problema da ***Tabu Search*** é que, se o espaço de busca for muito grande, e particularmente, se for de alta dimensionalidade, a busca poderá ficar presa em uma mesma vizinhança, mesmo que utilize uma extensa Lista Tabu - com um imenso espaço de busca, pode simplesmente existir um excesso posições para serem avaliadas. Como abordagem alternativa, pode-se criar uma Lista Tabu não com as soluções candidatas, mas com as alterações recentemente efetuadas ao gerar vizinhos de uma solução.

Exemplos de Aplicação

Há referências na literatura de aplicações da ***Tabu Search*** em problemas do tipo *Job-Shop* em [7], [8] e [9]. Este tipo de problema foi previamente descrito nos exemplos de aplicação da *Best-first Search*.

Para aplicar esta metaheurística a problemas *Job-Shop*, as soluções factíveis do *Job-Shop* são mapeadas para estados, cada qual associado a um custo. O objetivo é maximizar ou minimizar tal custo associado [7].

Simulated Annealing

Definição

Simulated Annealing é um algoritmo de busca local baseado no conceito de recozimento ("*annealing*"), um processo que consiste em aquecer um metal até o ponto de fusão e então resfriá-lo, lentamente, permitindo que as moléculas alcancem uma configuração de baixa energia e formem uma estrutura cristalina, livre de defeitos [3].

Se o resfriamento for suficientemente lento, a configuração final resulta em um sólido com alta integridade estrutural. ***Simulated Annealing*** estabelece uma conexão entre este comportamento termodinâmico e a busca pelo máximo/mínimo global de um problema de otimização discreto.

A cada iteração, a função objetivo gera valores para duas soluções, a atual e uma escolhida, que são comparadas. Soluções melhores que a atual são sempre aceitas, enquanto que uma fração das soluções piores que a atual são aceitas na esperança de se escapar de um mínimo/máximo local, na busca pelo mínimo/máximo global.

A probabilidade de se aceitar uma solução R de qualidade inferior à solução atual S é dada pela seguinte fórmula (T é um parâmetro chamado temperatura, em analogia ao recozimento, e é tipicamente decrementado a cada iteração do algoritmo):

$$P(R,S,T) = exp(Qualidade(R) - Qualidade(S)) \div T$$

O ponto chave do algoritmo para evitar mínimos/máximos locais é permitir movimentos ascendentes (*hill climbing*) a partir de soluções que pioram a qualidade da solução atual, na esperança de encontrar o máximo/mínimo global. A cada iteração, a temperatura é reduzida, o que diminui a probabilidade de escolha de uma solução menos promissora e aumenta a tendência de se melhorar a solução atual.

Formas de Implementação

No Algoritmo 7, P(R,S,T) é a expressão descrita na seção anterior e Aleatorio() é uma função que gera um número aleatório com valor entra 0 e 1.

```
01: T ← temperatura com valor elevado
02: S ← solução candidata inicial qualquer
03: Melhor ← S
04: repita
05:     R ← GerarVizinho( S )
06:     se Qualidade(R) > Qualidade(S) ou se Aleatorio() < P(R,S,T) então
07:         S ← R
08:     T ← NovaTemperatura(T)
09:     se Qualidade(S) > Qualidade(Melhor) então
10:         Melhor ← S
11: até que Melhor seja a solução ideal, ou o tempo tenha esgotado, ou T < 0
12: devolva Melhor
```

Algoritmo 7: *Simulated Annealing*

A função NovaTemperatura() é fornecida pelo usuário e define a taxa com que a temperatura é decrementada - esta taxa é chamada de *scheduling* [2]. Quanto mais o *scheduling* da temperatura é prolongado, mais o algoritmo se aproxima de uma busca aleatória no espaço de soluções.

Vantagens e Desvantagens

A probabilidade de aceitação P(R,S,T) é o elemento básico do mecanismo de busca do ***Simulated Annealing***. Se a temperatura T é reduzida de forma suficientemente lenta, então o sistema pode atingir o equilíbrio. Este equilíbrio segue a distribuição de Bolztmann, que descreve a probabilidade de um sistema estar em um estado S com energia f(S) e temperatura T. A distribuição de probabilidade da transição entre estados define uma sequência de soluções geradas por uma cadeia de Markov, a partir da qual o algoritmo tem sua convergência provada. Portanto, a vantagem do ***Simulated Annealing*** é sua garantia de convergência.

Embora não tenham sido encontradas evidências de desvantagens na literatura consultada, pela definição do algoritmo podemos supor que, apesar da convergência ser garantida, ela pode ser muito lenta, uma vez que a garantia surge quando o *scheduling* faz a temperatura decrementar lentamente.

Exemplos de Aplicação

Em [16] e [17], o ***Simulated Annealing*** é aplicado para resolver problemas do tipo *Job-Shop*. Já em [18], o algoritmo é utilizado para resolver problemas do tipo *Flow-Shop*, em que a ordem de processamento das tarefas deve ser a mesma em cada máquina e o objetivo é minimizar o maior tempo de término.

Uma aplicação interessante da metaheurística é apresentada em [19], trabalho no qual o ***Simulated Annealing*** é utilizado para o escalonamento das partidas do campeonato brasileiro de futebol.

Capítulo III

Estratégias Construtivas

Ant Colony Optimization

Definição

***Ant Colony Optimization* (ACO)** é uma abordagem construtiva (soluções são construídas de forma iterativa através da adição de componentes a uma solução inicial nula, sem *backtracking*, até que uma solução completa seja encontrada) inspirada no uso que as formigas fazem dos feromônios, um sinal químico liberado por um animal que dispara uma resposta natural em outros membros da mesma espécie, como um canal de comunicação [3].

Analogamente às formigas reais, **ACO** é baseada na comunicação indireta de uma colônia de agentes simples, chamados de formigas artificiais, mediado por rastros de feromônios artificiais. Os rastros de feromônio na **ACO** funcionam como uma informação numérica distribuída que as formigas utilizam para construir, probabilisticamente, soluções para um problema e à qual as formigas se adaptam durante a execução do algoritmo para refletir sua experiência de busca.

As formigas artificiais utilizadas no **ACO** são procedimentos estocásticos que constroem uma solução por meio da adição iterativa de componentes a uma solução parcial, levando em conta (i) informações heurísticas sobre a instância do problema, se disponível, e (ii) trilhas de feromônio artificiais que mudam dinamicamente, em tempo de execução, para refletir a experiência de busca adquirida pelos agentes.

O componente estocástico do **ACO** permite que as formigas construam uma grande variedade de soluções diferentes e, assim, explorem um número muito maior de soluções do que as heurísticas gulosas. Ao mesmo tempo, o uso de informação heurística pode guiar as formigas em direção a soluções mais promissoras.

A experiência de busca das formigas ainda pode ser usada para influenciar a construção da solução em iterações futuras do algoritmo, e o uso de uma colônia de formigas pode fornecer ao algoritmo um grau elevado de robustez e eficiência na resolução dos problemas.

Formas de Implementação

Vamos considerar o problema (S,f,Ω), onde S é o conjunto de soluções candidatas, f é a função objetivo que atribui a cada $s \in S$ um valor $f(s,t)$ (f pode depender do tempo t em problemas dinâmicos) e Ω é um conjunto de restrições. O objetivo é encontrar uma solução global ótima $s_{opt} \in S$ que minimiza/maximiza a função f e satisfaz as restrições Ω.

Este problema pode ser representado por:

- Um conjunto $C = \{c_1, c_2, \ldots, c_{Nc}\}$ de componentes
- Os estados do problema são definidos em termos de sequências $x = \langle c_i, c_j, \ldots, c_k, \ldots \rangle$ sobre os elementos de C. O conjunto de todas as possíveis sequências é denotado por χ
- O conjunto de soluções candidatas S é um subconjunto de χ
- As restrições de Ω definem o conjunto de estados factíveis $\overline{\chi} \subseteq \chi$

Dada a representação anterior, as formigas artificiais constroem soluções ao se moverem num grafo de construção totalmente conexo $G = (C,L)$. Estes movimentos não são arbitrários, pois seguem uma política de construção definida em função das restrições contidas em Ω, possibilitando um certo grau de flexibilidade. Ademais, em certos problemas, pode ser razoável implementar as restrições de forma que as formigas só construam soluções factíveis, enquanto em outros pode ser mais vantajoso permitir que construam soluções não-factíveis, a serem penalizadas posteriormente.

Os componentes $c_i \in C$ e as conexões $l_{ij} \in L$ podem ser associados a uma trilha de feromônio τ que codifica uma memória de longa duração sobre todo o processo de busca, bem como a um valor heurístico η que representa informações sobre a instância do problema *a priori*, ou informações sobre o tempo de execução, fornecidas por uma fonte diferente das formigas.

A **ACO** faz com que a colônia de formigas se mova de forma concorrente e assíncrona através de estados adjacentes de um problema, construindo caminhos sobre o grafo G. O movimento é dado pela aplicação de uma política de decisão estocástica local que faz uso das trilhas de feromônio e de informações heurísticas. Ao se moverem, as formigas constroem soluções para o problema de forma incremental. Uma vez que uma formiga tenha construído uma solução, ou enquanto a solução está em construção, a formiga avalia a solução (parcial) e deposita feromônio nos componentes e/ou nas conexões usadas. Este feromônio irá guiar a busca de outras formigas.

Além da atividade das formigas, um algoritmo **ACO** inclui dois procedimentos adicionais: evaporação de feromônio e ações de segundo plano.

- Evaporação de feromônio é o processo pelo qual a quantidade de feromônio depositado pelas formigas é decrementado ao longo do tempo. Este processo é necessário para evitar uma rápida convergência do algoritmo em direção a uma solução sub-ótima.

- Ações de segundo plano são usadas para implementar ações centralizadas que não podem ser efetuadas por formigas individuais. Exemplos são a coleta de informações globais ou um aumento na trilha de feromônio deixada pela formiga que encontrou a melhor solução.

O Algoritmo 8 descreve a forma geral de um algoritmo de **ACO** [2]. As funções EvaporarFeromônio() e AçõesDeSegundoPlano() efetuam os procedimentos supracitados.

```
01: C ← {C1, ..., Cn} componentes
02: t ← número de trilhas para construir de uma só vez
03: f ← <f1,...,fn> feromônios dos componentes
04: Melhor ← nulo
05: repita
06:     P ← t trilhas, construídas por seleção iterativa de componentes baseada nos
feromônios e nas informações heurísticas
07:    para cada Pi em P faça
08:         se Melhor = nulo ou Qualidade(Pi) > Qualidade (Melhor) então
09:             Melhor ← Pi
10:     atualize f para os componentes baseado na qualidade para cada Pi em P em que
        eles participaram
11:     EvaporarFeromônio()
12:     AçõesDeSegundoPlano()
13: até que Melhor seja a solução ideal ou o tempo tenha se esgotado
14: devolva Melhor
```

Algoritmo 8: *Ant Colony Optimization*

Vantagens e Desvantagens

Uma vantagem do **ACO** é sua flexibilidade. O algoritmo pode ser adaptado para diversas situações, por meio de alterações na maneira estocástica como as formigas se movimentam e como o feromônio evapora.

Outra vantagem é o seu uso em problemas dinâmicos, cujas características se alteram ao longo da execução do algoritmo. Tal aplicação tem sido estudada mais recentemente e demonstra o grau de adaptação que o **ACO** consegue atingir.

Algumas desvantagens do **ACO** são citadas em [34]:

- Estagnação;
- Alto tempo computacional;
- Convergência prematura.

Exemplos de Aplicação

A primeira aplicação do **ACO** foi para resolver o problema do caixeiro viajante. Esta primeira implementação é conhecida como *Ant System* e foi desenvolvida antes mesmo da própria definição de **ACO** [35].

O *AntNet System* é um algoritmo **ACO** para problemas dinâmicos que é aplicado com sucesso ao roteamento em redes de comutação de pacotes (como a Internet), e tem demonstrado experimentalmente uma performance melhor do que algoritmos no estado da arte em estudos comparatórios.

Na área de escalonamentos, o **ACO** já foi aplicado ao problema *Single Machine Total Weighted Tardiness Scheduling Problem*, em que n tarefas devem ser processadas sequencialmente sem interrupção em uma única máquina [36]. Outro exemplo é [37], em que um algoritmo **ACO** é usado para resolver um problema do tipo *Flow-Shop*.

GRASP

Definição

Uma das técnicas mais promissoras na busca de soluções para problemas de otimização combinatória, o **GRASP** (*Greedy Randomized Adaptive Search Procedures*) é um método cuja introdução ocorreu em 1989, por Feo e Resende [20], que estudavam problemas difíceis de cobertura de conjuntos. As demais informações presentes nesta seção têm como base [21], um artigo de Festa e Resende sobre o desenvolvimento do método ao longo do tempo.

O **GRASP** é um processo iterativo em que cada iteração consiste em duas fases: a primeira é uma fase de construção e a segunda é uma fase de busca local. A solução que se mostrar a melhor de maneira geral é mantida como resultado. Detalhes sobre as fases serão descritos a seguir.

- **Construção:** nesta fase, produz-se uma solução factível, também de forma iterativa (os elementos da solução são escolhidos um a um). A cada iteração de construção, a escolha do próximo elemento a ser adicionado à solução é determinada pela ordenação de todos os elementos candidatos, com base em uma função $g:C \rightarrow \mathbb{R}$. Esta função recebe como entrada a lista de candidatos e verifica quais são os benefícios de se escolher cada um. Pode-se afirmar, então, que se trata de uma heurística adaptativa, já que os benefícios associados aos elementos são atualizados a cada iteração de construção, para que sempre se leve em consideração a última escolha realizada.

- **Busca local:** o objetivo nesta fase é encontrar o máximo local na vizinhança da solução obtida na fase anterior. A busca é útil na grande maioria das vezes, dado que o **GRASP** não garante que a solução da primeira fase seja localmente ótima. Assim, testam-se as soluções vizinhas sucessivamente, até que se encontre a melhor. É importante que se escolha uma boa estrutura para representar a vizinhança e que as técnicas de busca nessa estrutura sejam eficientes.

O **GRASP** ainda apresenta um componente probabilístico que se caracteriza pela escolha aleatória de um entre os melhores candidatos na lista da fase de construção. Isto permite que

diferentes soluções sejam obtidas a cada iteração do método, mas não compromete seu potencial adaptativo.

Formas de Implementação

A implementação geral, extraída de [2], é simples: cria-se a solução factível pela construção utilizando os componentes de maior valor (menor custo) e usa-se o *Hill Climbing*, previamente descrito, na fase de busca local. A função Variar() serve para alterar parcialmente uma cópia da solução recebida, de modo a obter uma nova solução factível.

```
01: C ← {C1 , ..., Cn } // componentes
02: p ← porcentagem dos componentes serem inclusos em cada iteração
03: m ← tempo para se realizar Hill Climbing
04: Melhor ← 2
05: repita
06:     S ← {} // solução candidata
07:     repita
08:         C' ← elementos em C − S que podem ser postos em S sem torná-la infactível
09:         se C' for vazio, então
10:             S ← {} // tente novamente
11:         senão
12:             C'' ← componentes em C' com o maior valor (ou menor custo) para p%
13:             S ← S ∪ {componente escolhido uniforme e aleatoriamente em C''}
14:     até que S seja uma solução completa
15:     faça m vezes
16:         R ← Variar(Copiar(S))
17:         se Qualidade(R) > Qualidade(S), então
18:             S←R
19:     se Melhor = 2 ou Qualidade(S) > Qualidade(Melhor), então
20:         Melhor ← S
21: até que Melhor seja a solução ideal ou o tempo tenha se esgotado
22: devolva Melhor
```

Algoritmo 9: Implementação geral do GRASP

Vantagens e Desvantagens

O **GRASP** possui a vantagem de ser facilmente implementável, com um ajuste ou outro de parâmetros do método. Assim, pode-se concentrar esforços em estruturas de dados eficientes que agilizem o processo iterativo.

Segundo [21], outra grande vantagem do **GRASP** é a implementação trivial em paralelo: cada processador pode ser iniciado com sua própria cópia do procedimento, bem como com os dados da instância e uma sequência numérica aleatória independente. As iterações são executadas em paralelo e apenas se utiliza uma variável global que guarda a melhor solução encontrada entre os processadores.

A flexibilidade oferecida por esta metaheurística ainda permite adaptações conforme a necessidade. Desde sua criação, foram propostas muitas variantes e especializações, como funções de probabilidade de escolha de acordo com o posicionamento dos elementos na lista de candidatos, o tamanho variável dessa lista, entre outras [22].

No entanto, a flexibilidade também traz consigo uma desvantagem. A análise formal da qualidade das soluções obtidas pelo **GRASP** é complexa, visto que cada iteração resulta em uma solução de distribuição desconhecida em relação às soluções possíveis, com média e variância dependentes da lista de candidatos. Por exemplo, se o tamanho da lista for restrito a um elemento e uma função gulosa eficiente for usada, haverá apenas uma solução, a variância será nula e a média provavelmente será um bom valor, mesmo que abaixo do ótimo. Por outro lado, uma lista maior implicará no aumento

da variância e em possível redução da média, por comprometer a função.

Exemplos de Aplicação

O **GRASP** tem aplicação aos mais diversos problemas de escalonamento e são muitas as publicações que tratam desta metaheurística. Alguns exemplos são:

- Em 1989, Feo e Bard [23] apresentaram um modelo para localizar estações de manutenção e ainda desenvolver escalonamentos de voos de acordo com a demanda cíclica por manutenção. Dada a difícil obtenção de soluções factíveis por meio do relaxamento do problema de programação linear, os autores propõem o uso de **GRASP**.

- Em 1996, Xu e Chiu [24] trataram o problema de escalonar serviços em diferentes locais e com janelas de tempo para técnicos com diferentes habilidades de trabalho. A escolha adaptativa para o **GRASP** proposto é selecionar os serviços (tarefas) com maior peso unitário. A busca local implementa quatro diferentes movimentos, entre eles a troca de um serviço alocado por outro ainda não-alocado.

- Em 1998, Lourenço, Paixão e Portugal [25] apresentaram diversas metaheurísticas para solucionar problemas de escalonamento de tripulação e que podem ser aplicadas em um sistema de planejamento de transportes, em tempo real e com um ambiente de fácil uso. Entre as abordagens estão a Tabu Search, o **GRASP** e algoritmos genéticos.

- Também em 1998, a tese de Rivera [26] apresentou diversas implementações paralelas de metaheurísticas, entre elas o **GRASP**, para o problema do escalonamento de cursos.

- Em 2001, Binato, Hery, Loewenstern e Resende [27] propuseram o **GRASP** para o problema de escalonamento do *Job-Shop*. A metaheurística aproveita um esquema de intensificação baseado em memória, que melhora a fase de busca local, e a função adaptativa é o makespan (tempo máximo em que se completa uma sequência de serviços) resultante de se adicionar uma operação não-escalonada às que já foram escalonadas.

Capítulo IV

Métodos Baseados em População

Memetic Algorithms

Definição

As informações presentes nesta definição foram extraídas de [38], um ótimo artigo introdutório sobre o assunto.

"***Memetic Algorithms***" (**MAs**) é o nome de uma vasta classe de metaheurísticas baseadas em população, introduzida em 1989 por Moscato [39]. Enquanto a técnica de busca local parte de uma configuração (solução) inicial e itera com o objetivo de atingir uma configuração ótima, algoritmos de busca baseados em população são técnicas que trabalham duas ou mais configurações simultaneamente.

Diferente de outros métodos tradicionais de Computação Evolutiva, a maior preocupação dos **MAs** está em explorar todo o conhecimento disponível sobre um dado problema, ou seja, é fundamental adquirir qualquer informação sobre seu domínio. Esta característica é representada pelo termo "*memetic*" (ou "memético", em português), o qual provém de "meme", conceito definido por Dawkins [40] como uma informação que é propagada pelos cérebros dos indivíduos.

Esta é a definição, traduzida, de Dawkins para um meme, em 1976: "Exemplos de memes são melodias, idéias, frases de efeito, modas, modos de fazer potes ou de construir arcos. Assim como genes se propagam no conjunto de genes ao passarem de corpo em corpo por meio de espermatozóides ou óvulos, memes se propagam no conjunto de memes ao passarem de cérebro em cérebro via um processo que genericamente pode ser chamado de imitação". Em [38], afirma-se que isso significa que a informação não é simplesmente transmitida, mas sim influenciada por cada parte envolvida. No entanto, há argumentos contrários a isso em [2].

Reflete-se nessa influência sobre informações a capacidade dos **MAs** de incorporar heurísticas, algoritmos de aproximação, técnicas de busca local, entre outros. Essencialmente, podem-se identificar **MAs** como uma estratégia de busca em que uma população de agentes otimizadores cooperam e competem entre si. O sucesso desta metaheurística é consequência direta dessa sinergia entre diferentes abordagens e, por esta característica, há inúmeras citações do algoritmo como um algoritmo híbrido.

O conceito de recombinação é importante para as operações que os **MAs** realizam com as configurações disponíveis em um dado instante. Trata-se de um processo em que um conjunto S_{pai} de n configurações é manipulado para gerar um outro conjunto S_{filho} de m novas configurações. Tal criação de descendentes envolve a identificação e a combinação de características extraídas dos pais. Operadores de recombinação que utilizam o conhecimento sobre um problema são chamados de heurísticos ou híbridos e, neles, as informações servem como guia na construção de descendentes.

Um aspecto digno de atenção é a complexidade computacional da recombinação. Combinar características de diversas soluções é claramente mais caro do que apenas modificar uma solução (alteração denominada "mutação"). Ademais, a operação de recombinação geralmente será invocada um grande número de vezes, então é conveniente, e muitas vezes obrigatório, manter um custo baixo por operação. Uma recomendação razoável é considerar o limite superior $O(NlogN)$ para a complexidade, onde N é o tamanho da entrada.

Formas de Implementação

Uma forma genérica de implementação dos **MAs** será vista a seguir, ainda de acordo com [38]. Ela é bastante similar a uma busca local, porém com efeito sobre um conjunto de configurações, como pode ser visto no Algoritmo 10.

```
01: pop ← GerarPopulacaoInicial()
02: repita
03:     novapop ← GerarNovaPopulacao(pop)
04:     pop ← AtualizarPopulacao(pop, novapop)
05:     se pop convergiu então
06:         pop ← ReiniciarPopulacao(pop)
07: até que CriterioDeTermino()
```

Algoritmo 10: Implementação genérica de *Memetic Algorithm*

Algumas observações são necessárias para um melhor entendimento sobre como funciona o algoritmo. Encontra-se a seguir uma breve descrição sobre cada função auxiliar.

- A função GerarPopulacaoInicial é responsável por criar o conjunto inicial com $|pop|$ configurações, que podem ser geradas aleatoriamente ou por meio de algum mecanismo mais sofisticado, como uma heurística construtiva. Outra opção é utilizar uma estratégia de busca local (vide Algoritmo 11).

```
01: pop ← CriarPopulacaoVazia()
02: para j ← 1 até tamanhopop faça paralelamente
03:     i ← GerarConfiguracaoAleatoria()
04:     i ← MecanismoDeBuscaLocal(i)
05:     AdicionarIndividuo(i,pop)
06: devolva pop
```

Algoritmo 11: Possível implementação para gerar populações iniciais

- A função GerarNovaPopulacao é o coração do algoritmo e pode ser vista como um processo de n_{op} fases. Cada fase consiste em receber, como entrada, $aridade_E^j$ configurações da fase anterior, e gerar $aridade_S^j$ novas configurações como saída, após aplicar um operador op^j. O processo é restrito a ter $aridade_E^1 = tamanhopop$, ou seja, o número de configurações de entrada na primeira fase equivale ao tamanho da população.

```
01: buffer^0 ← pop
02: para j ← 1 até n_op faça paralelamente
03:     buffer^j ← CriarPopulacaoVazia()
04: para j ← 1 até n_op faça paralelamente
05:     s^j_pai ← ExtrairDoBuffer(buffer^(j-1), aridade^j_E)
06:     s^j_filho ← AplicarOperador(op^j, s^j_pai)
07:     para z ← 1 até aridade^j_S faça
08:         AdicionarIndividuo(s^j_filho[z], buffer^j)
09: devolva buffer^(n_op)
```

Algoritmo 12: Possível implementação para gerar novas populações

- A função AtualizarPopulacao reconstrói a população atual com base nas populações antiga e nova. Há duas possibilidades principais para essa reconstrução: uma é selecionar as

configurações com melhor $tamanhopop$ de $pop \cup novapop$, outra é extrair apenas as melhores de $novapop$.

- A função ReiniciarPopulacao pode ser implementada de várias formas, entretanto tipicamente consiste em manter uma fração da população atual e substituir as configurações restantes por novas soluções geradas. Vale notar que ReiniciarPopulacao só é executada quando se considera que a população está em estado degenerado, o que pode ser determinado por meio de diversas medidas.

```
01: novapop ← CriarPopulacaoVazia()
02: numpreservados ← tamanhopop * porcentagempreservacao
03: para j ← 1 até numpreservados faça
04:     i ← ExtrairMelhorDaPopulacao(pop)
05:     AdicionarIndividuo(i, novapop)
06: para j ← (numpreservados+1) até tamanhopop faça paralelamente
07:     i ← GerarConfiguracaoAleatoria()
08:     i ← MecanismoDeBuscaLocal(i)
09:     AdicionarIndividuo(i, novapop)
10: devolva novapop
```

Algoritmo 13: Possível implementação para reiniciar populações

- A função CriterioDeTermino, assim como GerarPopulacaoInicial, também pode ser definida de forma similar à busca local, isto é, definindo um limite para o número total de iterações, ou um número máximo de iterações sem melhorias, ou um limite para o número de reinícios, etc.

Vantagens e Desvantagens

A principal vantagem de se usar **MAs** está na possibilidade de se realizar várias buscas locais em paralelo, utilizando as soluções de cada uma para decidir o que fazer e inclusive recombinando-as. Finalmente, por haver a obtenção de mais informações do que simplesmente realizar uma única busca local, a tendência é que haja maior base na busca por soluções ótimas.

A maior desvantagem notada ao longo da análise sobre esta metaheurística é a dependência da operação de recombinação, a qual é necessária para se combinar as configurações encontradas em um dado momento. Conforme descrito na definição de **MAs**, tal operação apresenta um limite sugerido para complexidade, por isso é preciso encontrar uma implementação que atenda à demanda do problema em questão e que não ultrapasse o limiar.

Exemplos de Aplicação

Os **MAs** atualmente são uma abordagem bastante comum para resolver problemas clássicos de otimização, inclusive os NP-difíceis, como o problema do caixeiro viajante, o de particionamento de conjuntos e também o de particionamento de grafos.

Um exemplo de uso da metaheurística para problemas de escalonamento é [47], em que os autores retratam o problema de executar programas usando vários processadores. O objetivo é minimizar o *makespan*, ou seja, o programa deve ter sua execução concluída o quanto antes. Neste artigo, refere-se ao **MA** como uma combinação de *Simulated Annealing* com *Genetic Algorithms*.

Outro exemplo é [48], no qual é proposto um **MA** para o escalonamento de manutenções planejadas de uma rede de transmissão elétrica em que toda linha de transmissão precisa ser tratada alguma vez em um período específico. O objetivo é evitar situações em que seções da rede são desconectadas e também para minimizar a sobrecarga de linhas que estão em uso. A combinação emprega buscas locais como *Tabu Search* e *Simulated Annealing*.

Mais exemplos na área de escalonamentos envolvem problemas com:

- Tempos de *setup* e datas de entrega em uma máquina [49,50,51];
- Máquinas paralelas [52,53,54];
- Escalonamentos de projetos [55,56,57];
- *Open-Shop* [58,59,60], flow shop [61,62] e *Job-Shop* [63];
- Escalonamento de jogos de hóquei [64].

Genetic Algorithms

Definição

Os ***Genetic Algorithms*** (**GAs**) foram citados pela primeira vez em 1975, por Holland [65]. Posteriormente, passaram a ser relacionados com os termos de computação evolucionária ou algoritmos evolucionários. Porém, diferente de outros algoritmos que a precederam, esta metaheurística não se comporta apenas como uma estratégia de busca local baseada no conceito de mutações: os **GAs** incorporaram a noção de recombinação.

A motivação original para a metaheurística foi uma analogia biológica [66]. Na criação de plantas e animais, existe a seleção com base em características desejáveis, as quais são determinadas geneticamente, de acordo com a forma como se combinam os cromossomos dos pais. No caso dos **GAs**, a população utilizada é composta por diversas sequências, relacionadas como cromossomos, e a recombinação das sequências comumente se dá pelo uso de simples analogias com mutações e o *crossover*:

- Mutação é um operador que realiza a escolha aleatória de um subconjunto de genes e seus valores são trocados pelos de seus genes alelos (aqueles que pareiam com os primeiros, ocupando posições correspondentes em cromossomos homólogos);

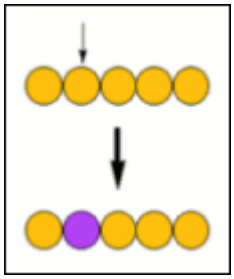

Figura 3: Mutação (Fonte: [67])

- *Crossover* é um operador que substitui os genes de um pai pelos genes correspondentes de outro, ou seja, realiza uma combinação entre os genes dos pais.

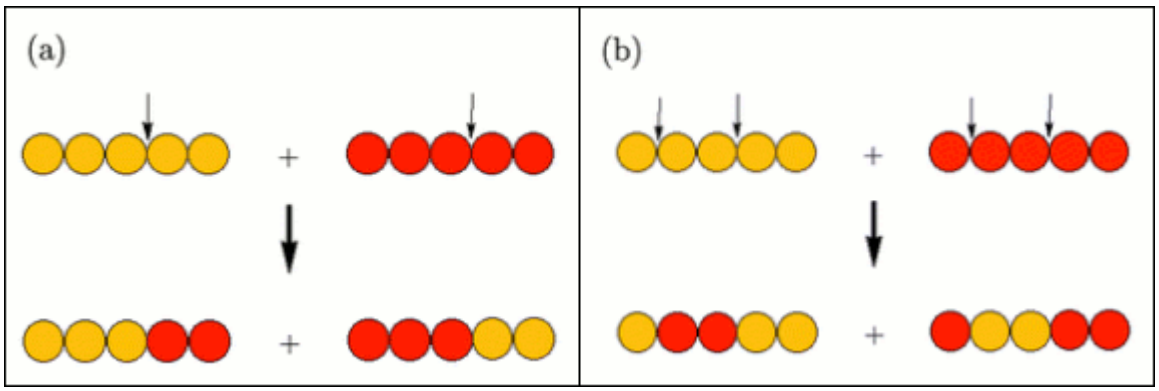

Figura 4: *Crossover* em (a) um ponto do cromossomo e em (b) dois pontos (Fonte: [67])

A busca é, enfim, guiada pelos resultados obtidos da avaliação de uma função objetiva para cada sequência da população, de modo que as melhores sequências identificadas tendem a ser

selecionadas.

Basicamente, o problema a ser resolvido por um **GA** é otimizar (maximizar ou minimizar) uma função objetivo $f:X \to \mathbb{R}$, onde X é um espaço discreto de busca. Um aspecto peculiar dos **GAs** é a possibilidade de separar a representação do problema das variáveis reais com as quais foi originalmente formulado [66]. Trata-se de uma separação análoga à que ocorre na Biologia, onde os genótipos (representação codificada das variáveis) são diferenciados dos fenótipos (conjuntos de variáveis propriamente ditos). Assim, um vetor $x \in X$ é representado por uma sequência s com l símbolos de um alfabeto, por meio de um mapeamento $c:A^l \to X$.

Na prática, é preciso considerar um espaço de busca $S \subseteq A^l$ para representar que algumas sequências podem ser soluções inválidas para o problema. O comprimento l depende tanto de X quanto de A e os elementos da sequência são como genes, enquanto os valores desses genes podem levar aos alelos. Isto frequentemente se denomina "mapeamento genótipo-fenótipo" e transforma o problema na otimização de uma função $g(s) = f(c(s))$. Em alguns casos, isto dificulta o encontro de soluções de problemas de otimização por um **GA** [68]. Vale a pena ressaltar que nem sempre basta o uso de uma função composta $f(c(s))$: pode ser necessário o uso de $h(f(c(s)))$, tal que $h:\mathbb{R} \to \mathbb{R}$.

Pode-se dizer que um algoritmo genérico para **GA** é composto pelas seguintes fases [66]:

- **Inicialização da população:** os dois principais pontos considerados nesta fase são a escolha do tamanho da população e a definição do método que será aplicado na seleção dos indivíduos. A população deve ter um tamanho que não seja muito pequeno (não seria suficiente para a efetiva exploração do espaço de busca) nem muito grande (a eficiência do método seria afetada, pois uma solução pode não ser encontrada em um intervalo razoável de tempo). Alguns trabalhos tentaram, sem sucesso, encontrar a chave para o tamanho ideal, como é o caso de [69,70]. A inicialização propriamente dita é tipicamente aleatória, porém estudos como [71,72] mostram ser possível o uso de outras técnicas heurísticas para isso, com o objetivo de iniciar a partir de uma solução inicial melhor.

- **Condição de término:** trata-se da verificação antes de se realizar cada iteração. Como os **GAs** são métodos de busca estocástica, é preciso impor limites para que a execução não seja infinita. Abordagens comuns são o limite por tempo, pela quantidade de avaliações de soluções ou por alguma propriedade atingida pela população.

- **Condições de mutação e *crossover*:** é possível escolher entre diferentes estratégias para a execução do algoritmo, uma é dita *crossover-AND-mutation* e outra, *crossover-OR-mutation*. A primeira tenta sempre um *crossover* e depois a mutação, de modo que cada operação seja aplicada se o limite for excedido em uma distribuição uniforme pseudo-aleatória. A última oferece a possibilidade de variação nas proporções entre os operadores ao longo da busca. Segundo Davis [73], diferentes taxas são apropriadas para diferentes momentos: no início é interessante empregar uma alta taxa de *crossover*, aumentando a taxa de mutação de acordo com a convergência da população.

- **Mutação e *crossover*:** estas operações, descritas previamente, podem ser computacionalmente planejadas de diversas formas. Tipicamente utilizam-se máscaras de bits do tamanho das soluções e geradas com alguma distribuição estatística. Por exemplo, uma máscara "1 0 0 1" adquire diferente significado para cada operação: na mutação, indica que a primeira e a última posições receberão novos valores; já no *crossover*, pode-se considerar que a primeira e a última posições serão valores adquiridos de um pai, enquanto as posições intermediárias serão do outro pai. O *crossover* pode ocorrer de várias formas, seja o cruzamento em apenas um ponto da solução ou em mais de um ponto.

- **Seleção:** a cada geração, uma proporção da população é selecionada, com base em valores obtidos da avaliação de uma função, para preparar a próxima geração. Alguns métodos avaliam cada solução para escolher as melhores, enquanto outros analisam apenas uma amostra aleatória da população, visto que o processo total pode consumir bastante tempo. Entre os métodos mais empregados estão a seleção por roleta ("*roulette wheel selection*") e a por campeonato ("*tournament selection*"). A maioria das funções é estocástica e desenvolvida de modo a serem selecionadas pequenas proporções das soluções menos favorecidas: isto permite uma maior diversidade da população e ainda previne que ocorra convergência prematura para soluções ruins [74].

Formas de Implementação

Definidos os principais conceitos para o desenvolvimento de um algoritmo de **GA**, no Algoritmo 14 está um algoritmo genérico que serve como modelo para implementações. O algoritmo foi adaptado a partir do encontrado em [66].

```
01: EscolherPopulacaoInicial()
02: enquanto CondicaoDeTermino() for falsa, faça
03:     repita
04:         se CondicaoDeCrossover() for verdadeira, então
05:             SelecionarSolucoesDosPais()
06:             DefinirFormaDeCrossover()
07:             RealizarCrossover()
08:         se CondicaoDeMutacao() for verdadeira, então
09:             DefinirPontosDeMutacao()
10:             RealizarMutacao()
11:         avaliarValorDaNovaSolucao()
12:     até haver quantidade suficiente de novas soluções
13:     SelecionarNovaPopulacao()
14: devolva a melhor solução encontrada
```

Algoritmo 14: Implementação genérica de *Genetic Algorithm*

Vantagens e Desvantagens

A grande vantagem dos **GAs** está na implementação simples que oferece uma poderosa fonte de obtenção de soluções. As combinações que seus operadores conseguem gerar podem ser de difícil alcance para algoritmos de busca local.

Ironicamente, a grande desvantagem da metaheurística está diretamente relacionada à vantagem: o comportamento dos **GAs** é tão complexo e imprevisível que foge ao controle de quem os utiliza. Consequência direta disso é, até o momento da escrita deste texto, não existir a prova de como soluções tão boas são encontradas. A explicação oferecida para o comportamento se baseia na hipótese dos blocos de construção (*building block hypothesis*), que consiste em [74]:

- Uma descrição de um mecanismo adaptativo abstrato que recombina "blocos de construção", isto é, esquemas de baixa ordem com valor acima da média;
- Uma hipótese de que um **GA** realiza suas adaptações ao implementar tal mecanismo de forma implícita e eficiente, encontrando soluções.

No entanto, a hipótese dos blocos é criticada por diversos autores. Por exemplo, Wright *et al.* [75] afirmam que a hipótese não tem embasamento teórico e, em alguns casos, é simplesmente incoerente.

Exemplos de Aplicação

Os **GAs** têm inúmeros exemplos de sucesso de aplicação na área de otimização combinatória e também são aplicáveis para a engenharia. Alguns exemplos de aplicação na área de escalonamentos são:

- Escalonamento com restrição de recursos [76];
- Aplicação de **GA** paralelo para escalonar tarefas em um *cluster* [77];
- Escalonamento de tripulações aéreas [78];
- Escalonamento de enfermeiras [79].

Há inclusive exemplos implementados disponíveis na rede, como é o caso da solução proposta para o escalonamento de horários de aulas [80].

Particle Swarm Optimization

Definição

Particle Swarm Optimization **(PSO)** é uma técnica de otimização estocástica populacional baseada no comportamento coletivo de animais (como cardumes de peixes ou bandos de pássaros) [2].

O **PSO** foi apresentado por Kennedy e Eberhart [41] como um método para otimização de funções não lineares relacionado à metodologia de enxame de partículas, descoberto durante a simulação de um modelo social simplificado: a simulação de um bando ou revoada de pássaros em torno de um ponto contendo comida ou local para descanso.

Assim como algoritmos genéticos e redes neurais se baseiam em sistemas biológicos (evolução humana e um modelo simplificado do cérebro humano), o **PSO** é fundamentado em sistemas sociais. Mais especificamente, baseia-se no comportamento coletivo de indivíduos interagindo entre si e com o ambiente [42]. Este tipo de sistema é conhecido também como inteligência de enxame (*swarm intelligence*).

Diferentemente de outros métodos baseados em população, **PSO** não cria novos indivíduos (e descarta antigos) durante as iterações. Ao invés disso, uma única população estática é mantida, cujos membros possuem propriedades modificadas (como posição e velocidade) em resposta a novas descobertas sobre o espaço.

Formas de Implementação

O algoritmo utilizado pelo **PSO** é simples, mas depende de alguns conceitos, descritos a seguir.

- Partículas: são as soluções candidatas do problema. As partículas nunca morrem, isto é, não há seleção. Uma partícula consiste em duas partes:
  - A localização da partícula no espaço $\vec{x} = \langle x_1, x_{2,\dots} \rangle$;
  - A velocidade da partícula $\vec{v} = \langle v_1, v_{2,\dots} \rangle$, que indica a direção e a velocidade em que a partícula está viajando na iteração, equivalente a $\vec{v} = \vec{x}^{(t)} - \vec{x}^{(t-1)}$.

As partículas são inicializadas em posições escolhidas aleatoriamente e com velocidades iniciais também aleatórias. Três informações adicionais também são armazenadas:

- Para cada partícula, a melhor posição (o melhor valor obtido da função objetivo) $\vec{x}^*$ da própria partícula;
- Para cada partícula, a melhor posição $\vec{x}^+$ de qualquer um dos informantes da partícula - um pequeno conjunto de partículas escolhidas aleatoriamente a cada iteração (uma partícula é sempre informante dela mesma);
- A melhor posição de qualquer partícula até o momento $\vec{x}^!$.

A cada iteração do algoritmo são executadas as seguintes operações:

- Julgamento da qualidade da posição de cada partícula e atualização da melhor posição, caso necessário;
- Atualização do vetor de velocidade $\vec{v}$ para cada partícula, adicionando três vetores que apontam para $\vec{x}^*$, $\vec{x}^+$ e $\vec{x}^!$;

- Atualização da posição de cada partícula com base no vetor de velocidade calculado no passo anterior.

Com estas definições, segue o algoritmo do **PSO**, ilustrado no Algoritmo 15.

01: n ← tamanho do enxame
02: α ← proporção de $\vec{v}$ a ser mantida
03: β ← proporção de $\vec{x}^*$ a ser mantida
04: γ ← proporção de $\vec{x}^+$ a ser mantida
05: δ ← proporção de $\vec{x}^!$ a ser mantida
06: ϵ ← fator multiplicativo para a aplicação da velocidade
07: P ← conjunto de n partículas, inicializadas com velocidade e posição aleatórias
08: Melhor ← **nulo**
09: repita
10: para cada p em P faça
11: se Melhor = **nulo** ou Qualidade(p) > Qualidade(Melhor) então
12: Melhor ← p
13: para cada p em P faça
14: para cada dimensão i de $\vec{x}$ faça
15: b ← número aleatório no intervalo [0, β]
16: c ← número aleatório no intervalo [0, γ]
17: d ← número aleatório no intervalo [0, δ]
18: $v_i \leftarrow \alpha v_i + b(x_i^* - x_i) + c(x_i^+ - x_i) + d(x_i^! - x_i)$
19: para cada p em P faça
20: $\vec{x} \leftarrow \vec{x} + \epsilon\vec{v}$
21: até que Melhor seja ideal ou o tempo tenha esgotado
22: devolva Melhor

Algoritmo 15: *Particle Swarm Optimization*

Na fórmula que atualiza o v_i, pode-se notar as componentes:

- $b(x_i^* - x_i)$: representa o componente cognitivo, ou seja, a experiência individual da partícula;
- $d(x_i^! - x_i)$: representa o componente social, ou seja, a experiência de todo o enxame globalmente;
- $c(x_i^+ - x_i)$: posiciona-se em um meio termo entre a experiência pessoal e a global.

O parâmetro β indica quanto da experiência pessoal deve ser mantida. Quando ele é muito alto, as partículas atuam basicamente como escaladores autônomos (*hill-climbers*), ao invés de um conjunto de exploradores.

O parâmetro δ representa o quanto da experiência global deve ser levada em conta. Quanto maior o valor, mais as partículas se moverão em direção ao melhor valor conhecido.

Vantagens e Desvantagens

A facilidade de implementação e os resultados com qualidade similar ao dos algoritmos genéticos são citados na literatura como vantagens do **PSO** [41]. Esta metaheurística não requer o cálculo de derivadas para a sua implementação. Outra vantagem é a possibilidade de implementação em computadores paralelos.

Como desvantagens há o fato de o algoritmo poder convergir prematuramente (dependendo dos valores utilizados como proporções no algoritmo), porém demorar muito para encontrar o máximo/ mínimo. Isto significa que as partículas podem permanecer girando em torno do máximo/mínimo por muito tempo, até finalmente "pousarem" sobre o ponto.

Exemplos de Aplicação

Em [43] é descrita a utilização do **PSO** como um algoritmo para o escalonamento de aplicações em ambientes computacionais distribuídos. Uma versão modificada do **PSO** é apresentada em [44] para trabalhar em um espaço de busca discreto e resolver problemas do tipo *Job-Shop* e, em [45] e [46], o **PSO** é utilizado para resolver problemas de escalonamento do tipo *Flow-Shop*.

Capítulo V

Metaheurísticas Baseadas em Música

Harmony Search

Definição

Esta recente metaheurística foi introduzida em 2001 por Geem, Kim e Loganathan [28] e, desde então, variantes e melhorias foram propostas. Nesta seção, entretanto, será abordada a forma básica da ***Harmony Search*** (**HS**) no estado-da-arte, de acordo com o artigo de um dos criadores [29].

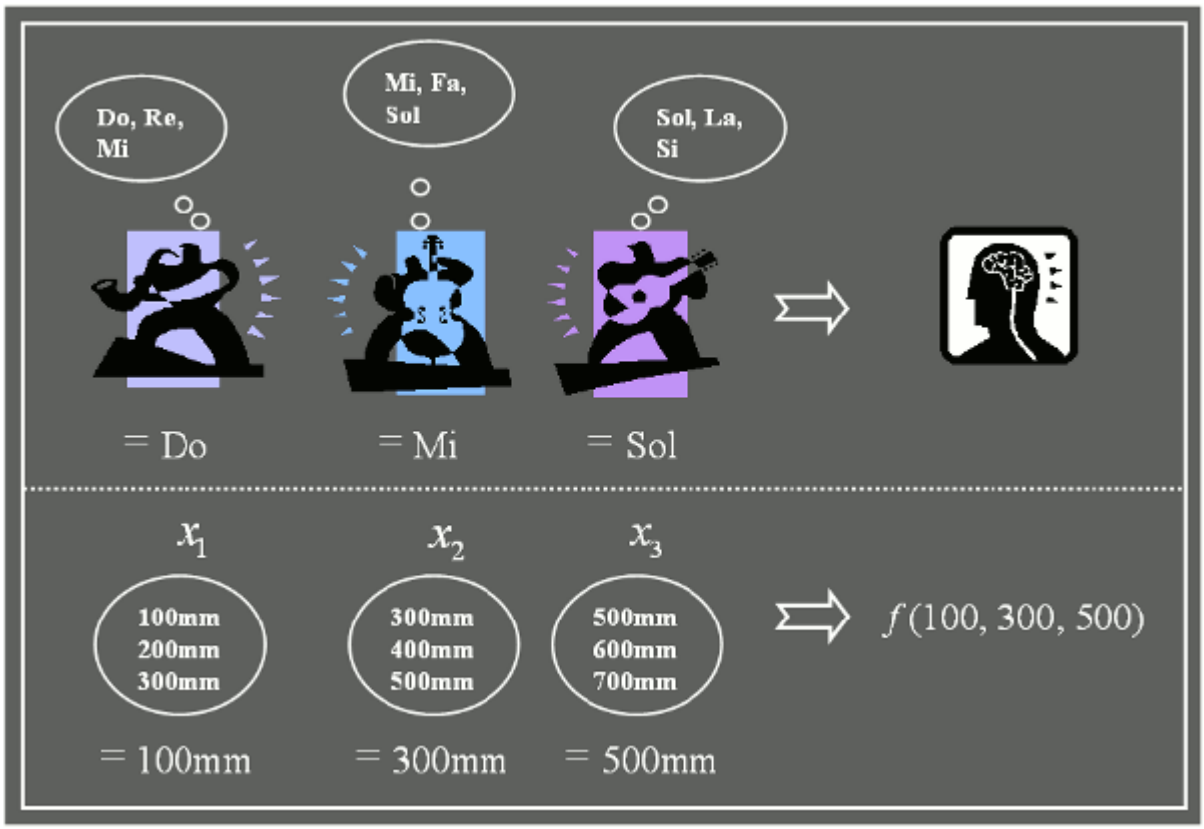

Figura 5: Analogia entre o improviso musical e a otimização (Fonte: [29])

Tradicionalmente, as soluções ótimas para problemas são procuradas por meio de algoritmos baseados em cálculo, mas isso não se aplica aos casos em que as variáveis são discretas e, assim, não há como derivá-las. A **HS** se propõe a resolver essa situação por meio de uma derivada estocástica [30] que pode ser aplicada a variáveis discretas. Tal derivada utiliza o conhecimento obtido na área da música, mais especificamente em experiências de músicos com o processo de improviso no Jazz. O autor de [29] não somente representou a analogia entre o improviso e a otimização (Figura 5), como promoveu a seguinte correspondência entre os fatores envolvidos:

- Músico → variável de decisão;
- Alcance do *pitch* de um instrumento musical → intervalo com valores de uma variável;
- Harmonia musical → vetor de soluções em uma determinada iteração;
- Estética auditiva → função objetivo;

- Prática (leva a melhor harmonia) → iteratividade (leva a melhor solução);
- Experiência → matriz de memória.

Ao invés de buscar informações na inclinação de uma função objetivo, a derivada estocástica traz à **HS** a probabilidade de seleção para cada valor de uma variável de decisão, de forma que a probabilidade seja atualizada a cada iteração. Claro, espera-se que, ao longo das iterações, aumentem as chances de escolha de um valor presente na solução ótima.

O algoritmo da **HS** é composto por sete passos [29]. São eles:

- **Formulação do problema:** para a aplicação do algoritmo, é necessária a formulação do problema com uma função objetivo e restrições, como a presente na Figura 6. Nela, é possível notar que tanto igualdades quanto desigualdades são aceitas e, em (4), estão representadas as definições dos conjuntos de valores discretos ou contínuos das variáveis. O algoritmo considera basicamente a função objetivo, porém duas atitudes podem ser tomadas em caso de violação de restrições: uma é abandonar o vetor de soluções e a outra é continuar com o vetor, desde que o valor da função receba uma penalidade.

Otimizar (minimizar ou maximizar) $f(x)$ (1)

Sujeita a:

$h_i(x) = 0; \quad i = 1, \dots, p;$ (2)

$g_i(x) \geq 0; \quad i = 1, \dots, q;$ (3)

$x_i \in X_i = \{x_i(1), \dots, x_i(k), \dots, x_i(K_i)\}$ ou $x_i^L \leq x_i \leq x_i^U$ (4)

Figura 6: Formulação de problema

- **Definição dos parâmetros do algoritmo:** o algoritmo lida com parâmetros que precisam receber valores, entre eles:
  - tamanho da memória de harmonia (número de vetores de solução com os quais o algoritmo lida simultaneamente);
  - memória de harmonia considerando taxas (taxa entre 0 e 1 com a qual a **HS** obtém um valor da memória, complementada pela taxa com a qual se obtém um valor do intervalo disponível);
  - taxa de ajuste de *pitch* (também entre 0 e 1, representa a taxa com a qual a **HS** altera um valor obtido da memória);
  - improviso máximo (número de iterações); e
  - largura das casas (comprimento arbitrário apenas para variáveis contínuas).

Figura 7: Trastes no braço de um violão, onde cada casa é o espaço entre dois trastes

- **Tuning aleatório para iniciar a memória:** em dado momento de concertos, os instrumentos tocam *pitches* aleatoriamente, fora do alcance normal. Analogamente, o algoritmo improvisa harmonias aleatórias, em uma quantidade igual ou maior ao tamanho da memória da harmonia. As melhores harmonias são então escolhidas como os vetores iniciais.

- **Improviso de harmonia:** no improviso do Jazz, um músico toca uma nota após selecioná-la aleatoriamente no alcance possível ou na sua memória. No segundo caso, ainda há a opção de usar a nota diretamente ou de realizar alguma alteração na mesma. Analogamente, a **HS** improvisa um valor por escolha aleatória no intervalo, por seleção na memória ou por alteração de um valor também obtido da memória. Como descrito previamente, a nova solução pode violar alguma restrição, quando é abandonada ou então adiciona uma penalidade ao valor final.

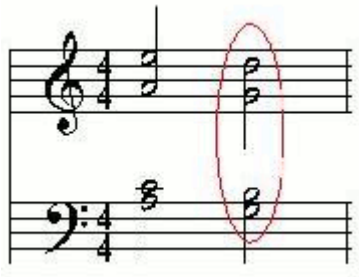

Figura 8: Violação de restrição
no contexto musical

- **Atualização da memória:** se a harmonia obtida for melhor do que a pior harmonia na memória, em termos de função objetiva, descarta-se a pior harmonia e integra-se a obtida à memória (outros critérios também podem ser aplicados na escolha da harmonia descartada, como o número de harmonias idênticas). No entanto, se a harmonia for melhor do que todas as presentes na memória, ainda pode ocorrer um processo de ajuste das notas dessa harmonia, para possivelmente se encontrar uma solução ainda melhor.

- **Execução da terminação:** se a **HS** satisfizer algum critério de terminação (por exemplo, atingir o improviso máximo), a computação é encerrada. Caso contrário, uma nova harmonia é improvisada e tem início outra iteração.

- **Cadência:** trata-se da passagem musical que ocorre antes do encerramento. No contexto do algoritmo, o termo pode se referir à execução de algum processo após a **HS**. Neste processo, o algoritmo devolve a melhor harmonia encontrada e armazenada na memória de harmonia.

Formas de Implementação

O Algoritmo 16 é uma variação do algoritmo presente em [31] e apresenta uma notação específica para os parâmetros de busca descritos na definição da **HS**, bem como os valores típicos para cada parâmetro.

- k é o tamanho da memória de harmonia, com valores típicos na ordem de 1 a 50;
- p_{hmcr} é a taxa de escolha de um valor na memória, tipicamente de 0,7 a 0,99;
- p_{par} é a taxa de ajuste de valores obtidos, tipicamente entre 0,1 e 0,5;
- δ é a distância entre dois valores da vizinhança em um conjunto discreto de candidatos;
- *fw* (ou *bw*) é o comprimento para variáveis contínuas.

É possível variar tais parâmetros ao longo da busca, de maneira similar ao que ocorre com a metaheurística *Simulated Annealing*.

01: inicie a memória de harmonia, selecionando *k* vetores aleatórios $x_1 \ldots x_k$.
02: repita
03: crie um novo vetor *x*'
04: para cada componente *x*'$_i$, faça
05: com probabilidade p_{hmcr}, faça $x'_i \leftarrow x_i^{int(rand(0,1)*k)+1}$
06: com probabilidade 1 - p_{hmcr}, escolha um novo valor aleatório no intervalo
07: para cada componente *x*'$_i$ escolhido da memória, faça

08: com probabilidade p_{par}, altere x'_i com uma pequena quantia. Para variáveis discretas, $x'_i \leftarrow x'_i \pm \delta$; para contínuas, $x'_i \leftarrow x'_i \pm bw \cdot rand(0,1)$

09: com probabilidade $1 - p_{par}$, nada faça

10: se x' for melhor do que o pior x^j na memória, troque x^j por x'
11: até que o improviso máximo seja atingido
12: devolva a melhor harmonia na memória

Algoritmo 16: *Harmony Search*

Vantagens e Desvantagens

Uma das maiores vantagens de se utilizar a **HS** está em sua própria definição, é a facilidade de se tratar variáveis discretas, bem como variáveis contínuas. Isto a coloca um passo à frente das técnicas baseadas em gradientes, que lidam apenas com as contínuas. Ademais, esta metaheurística apresenta uma estrutura básica de baixa complexidade e oferece, assim, flexibilidade suficiente para que os usuários possam adaptar o algoritmo de acordo com seus problemas.

A **HS** ainda possui uma vantagem no que diz respeito à pesquisa de informações. Um dos autores da metaheurística administra um site na Internet com materiais e tópicos relacionados, bem como indicações de livros. Até o momento da escrita deste texto, em 2009, o endereço do site é http://www.hydroteq.com/ .

Há ainda outras vantagens, mais propriamente relacionadas ao algoritmo, que são ressaltadas em [31]:

- O algoritmo não demanda cálculos complexos e, assim, é livre de divergências;
- Não é necessário estabelecer valores iniciais para as variáveis, propriedade que oferece uma boa probabilidade de o algoritmo escapar de ótimos locais;
- A **HS** é capaz de evitar um inconveniente dos algoritmos genéticos, a hipótese dos blocos de construção (maiores informações na seção correspondente deste texto).

Uma desvantagem do algoritmo está em sua compreensão teórica, uma vez que sua definição é totalmente baseada em analogias e termos provenientes do contexto musical. Leitores leigos em tal contexto precisam, então, buscar outras fontes de conhecimento para que possam compreender melhor a **HS**.

Outra desvantagem é a necessidade de se formular formalmente um problema antes da execução do algoritmo. Dependendo do problema, a formulação pode ser complexa devido às restrições envolvidas e, caso o problema não seja bem formulado, a solução obtida, ainda que ótima, pode não ser a solução esperada.

Exemplos de Aplicação

Embora a **HS** seja utilizada em áreas de Inteligência Artificial, Engenharia, Visão Computacional e até mesmo na Medicina e na Biologia, há poucos relatos de emprego em escalonamentos. Isto se deve, possivelmente, ao surgimento recente da metaheurística.

As ocorrências encontradas referentes a escalonamentos remetem a [32], trabalho publicado em 2007 e de autoria de um dos criadores da **HS**. O algoritmo é utilizado, com bons resultados, para se encontrar soluções ótimas no escalonamento de diques, extraindo assim o máximo benefício na geração de energia hídrica e na irrigação. Ainda é feita a comparação com o desempenho do modelo de algoritmos genéticos, que apenas se aproximam das soluções ótimas.

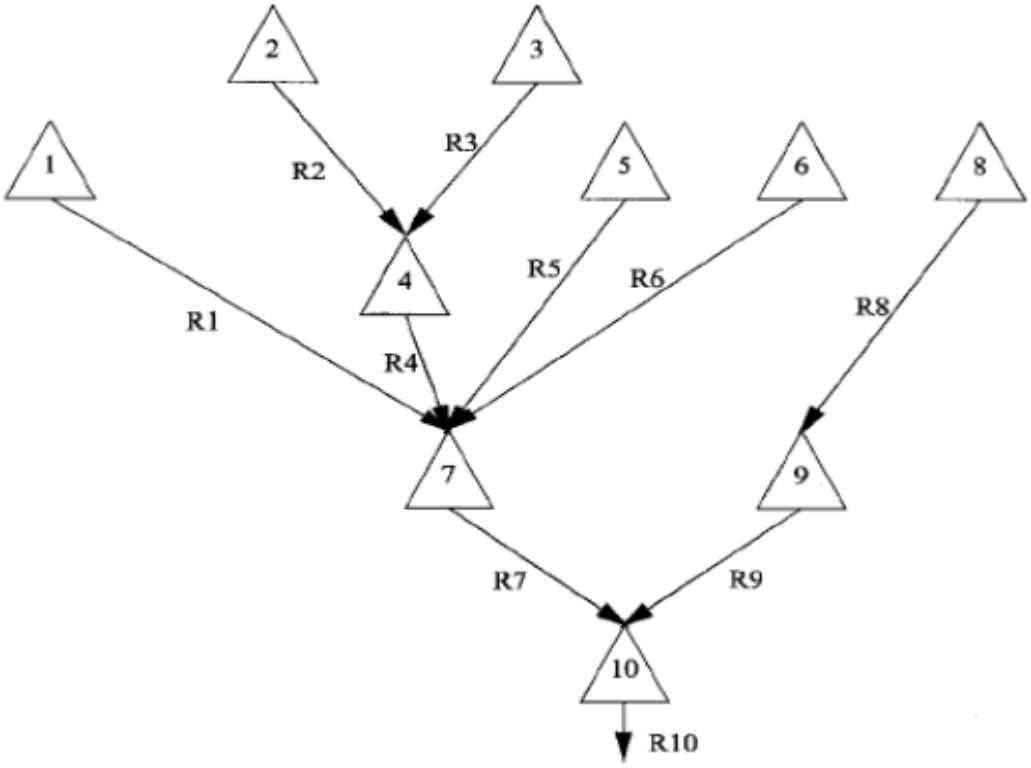

Figura 9: Representação do problema de escalonamento de diques (Fonte: [33])

Capítulo VI

Conclusão

As metaheurísticas compõem uma área de pesquisa bastante ativa, uma vez que são aplicáveis às mais diversas áreas, nem sempre pertencentes a ciências exatas. Os problemas de escalonamentos, em particular, inúmeras vezes usufruem o potencial oferecido por esses métodos de busca por soluções.

O contraste entre as inspirações dos métodos é notável. O desenvolvimento de uma metaheurística pode ser iniciado tanto do cálculo, por meio de estratégias planejadas, quanto pode apresentar as mais inusitadas e inesperadas origens. Exemplos são as comparações entre *Best-first Search* e *Ant Colony Optimization* ou entre *Tabu Search* e *Particle Swarm Optimization*. Alguns casos inclusive demandam novas aquisições de conhecimento, como é o caso da *Harmony Search* e seus conceitos referentes a música.

Há ainda diferenças entre as demonstrações do comportamento de metaheurísticas: enquanto algumas podem ter seu funcionamento comprovado, como a *Hill Climbing*, outras nem sequer atingiram tal etapa, caso dos *Genetic Algorithms*. A ausência de provas teóricas, no entanto, não é suficiente para que uma metaheurística seja abandonada: são diversas as situações em que o interesse está apenas nos resultados obtidos e não na forma ou na teoria de como isso ocorre.

O estudo realizado para a elaboração desta monografia permite afirmar que, apesar de toda a variedade, não existe uma metaheurística suprema, ou seja, melhor do que todas as outras. Há métodos mais empregados e outros menos, porém o domínio do problema para o qual se busca uma solução é o que efetivamente determina a utilidade de cada metaheurística.

Referências Bibliográficas

1. http://en.wikipedia.org/wiki/Metaheuristic
2. LUKE, S. (2009). *Essentials of Metaheuristics.* Disponível em http://cs.gmu.edu/~sean/book/metaheuristics/
3. GLOVER, F. e KOCHENBERGER, G. A. (2003). *Handbook of Metaheuristics*. Kluwer Academic Publishers, Boston.
4. http://en.wikipedia.org/wiki/Optimization_(mathematics)
5. RUSSEL, S. J. e NORVIG, P. (2003). *Artificial Intelligence: A Modern Approach*. 2nd ed. Prentice Hall, Upper Saddle River, NJ.
6. PEARL, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.
7. DELL'AMICO, M. e TRUBIAN, M. (1993). *Applying tabu search to the job-shop scheduling problem.* Springer Netherlands.
8. BARNES J. W. e CHAMBERS J. B. (1995). *Solving the job shop scheduling problem with tabu search.* IIE transactions.
9. BRANDIMARTE, P. (1993). *Routing and scheduling in a flexible job shop by tabu search.* Springer Netherlands.
10. WANG, Q., GAO, Y., e LIU, P. (2006). *Hill Climbing-Based Decentralized Job Scheduling on Computational Grids.* Proceedings of the First International Multi-Symposiums on Computer and Computational Sciences - Volume 1 (IMSCCS'06) - Volume 01.
11. CHOI, S. e YEUNG, D. (2005). *Hill-Climbing SMT Processor Resource Scheduler*. University of Maryland Technical Report CS-TR-4723 and UMIACS-TR-2005-30
12. HART, P. E.; NILSSON, N. J.; RAPHAEL, B. (1968). *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. IEEE Transactions on Systems Science and Cybernetics SSC4.
13. MINTON, S., JOHNSTON, M. D., PHILIPS, A. B., LAIRD, P.. (1990). *Solving Large-Scale Constraint-Satisfaction and Scheduling Problems Using a Heuristic Repair Method*. National Conference on Artificial Intelligence - AAAI.
14. SIERRA, M. R. e VARELA, R. (2005). *Optimal Scheduling with Heuristic Best First Search*. Lecture Notes in Computer Science. Springer Berlin / Heidelberg.
15. SIERRA, M. R. e VARELA, R. (2008). *Pruning by dominance in best-first search for the job shop Scheduling problem with total flow time*. Journal of Intelligent Manufacturing. Springer Netherlands.
16. VAN LAARHOVEN, P. J. M., AARTS, E. H. L., LENSTRA, J. K. (1990). *Job Shop Scheduling by Simulated Annealing*. Operations Reasearch, Vol. 40, No. 1. (Jan. - Feb., 1992), pp. 13-125.
17. KOLONKO, M. (1999). *Some new results on simulated annealing applied to the job shop scheduling problem.* European Journal of Operational Research. Volume 113, Issue 1, 16 February 1999, Pages 123-13. Elsevier Science.
18. OSMAN, I. H, POTTS, C. N. (1989). *Simulated annealing for permutation flow-shop scheduling.* Vol. 17, No. 6, pp. 551-557. Elsevier.
19. BIAJOLI, F. L., SOUZA, M. J. F., CHAVES, A. A. , MINE, O. M., CABRAL, L. A. F. e PONTES, R. C. *Scheduling the Brazilian Soccer Championship: A Simulated Annealing Approach.* Disponível em http://www.lac.inpe.br/~chaves/arquivos/patat2004.pdf
20. FEO, T.A. e RESENDE M.G.C. (1989). *A probabilistic heuristic for a computationally difficult set covering problem*. Operations Research Letters, 8:67–71.
21. FESTA, P. e RESENDE M.G.C. (2002). *GRASP: An Annotated Bibliography*. Em RIBEIRO, C.C. e HANSEN P., editores, *Essays and Surveys on Metaheuristics*, Kluwer Academic Publishers, 2002.
22. http://pt.wikipedia.org/wiki/GRASP_(algoritmo)
23. FEO, T.A. e BARD, J.F. (1989). *Flight scheduling and maintenance base planning*. Management Science, 35:1415–1432.
24. XU, J. e CHIU, S. (1996). *Solving a real-world field technician scheduling problem*. Em Proceedings of the International Conference on Management Science and the Economic Development of China, pp 240–248.
25. LOURENÇO, H.R., PAIXÃO, J.P. e PORTUGAL R. (1998). *Metaheuristics for the bus-driver scheduling problem*. Relatório técnico, Department of Economics and Management, Universitat Pompeu Fabra, Barcelona, Espanha.
26. RIVERA, L.I.D. (1998). *Evaluation of parallel implementations of heuristics for the course scheduling problem*. Tese de mestrado, Instituto Tecnologico y de Estudios Superiores de Monterrey, Monterrey, México.

27. BINATO, S., HERY, W.J., LOEWENSTERN, D. e RESENDE, M.G.C. (2001). *A greedy randomized adaptive search procedure for job shop scheduling*. Em HANSEN, P. e RIBEIRO, C.C., editores, *Essays and surveys on metaheuristics*, Kluwer Academic Publishers, 2001.
28. GEEM, Z.W., KIM, J.H. e LOGANATHAN, G.V. (2001). *A New Heuristic Optimization Algorithm: Harmony Search*, Simulation.
29. GEEM, Z.W.. *State-of-the-Art in the Structure of Harmony Search Algorithm*. Disponível em: http://www.hydroteq.com/HS_Structure.pdf.
30. GEEM, Z.W. (2008). *Novel derivative of harmony search algorithm for discrete design variables*. Applied Mathematics and Computation 199:223–230.
31. http://en.wikipedia.org/wiki/Harmony_search
32. GEEM, Z.W. (2007). *Optimal Scheduling of Multiple Dam System Using Harmony Search Algorithm*. Lecture Notes in Computer Science.
33. GEEM, Z.W.. *Music-Inspired Optimization Algorithm Harmony Search*. Apresentação. Disponível em: http://www.hydroteq.com/HS_Intro.ppt.
34. HUNG, K., SU, S. e LEE, Z. (2007). *Improving Ant Colony Optimization Algorithms for Solving Traveling Salesman Problems*. Journal of Advanced Computational Intelligence and Intelligent Informatics, Vol.11, No.4 pp. 433-442.
35. DORIGO, M., MANIEZZO, V. e COLORNI, A. (1991). *The Ant System: An autocatalytic optimizing process*. Technical Report 91-016 Revised, Dipartimento di Elettronica, Politecnico di Milano, Italy.
36. DEN BESTEN, M., STÜTZLE, T. e DORIGO, M. (2000). *Ant Colony Optimization for the Total Weighted Tardiness Problem*. Volume 1917/2000, pp. 611-620.
37. T'KINDT, V., MONMARCHÉ, N., TERCINET, F. e LAÜGT, D.. (2002). *An Ant Colony Optimization algorithm to solve a 2-machine bicriteria flowshop scheduling problem*. European Journal of Operational Research. Volume 142, Issue 2, pp. 250-257.
38. MOSCATO, P. e COTTA, C.. *A gentle introduction to memetic algorithms*. Em: GLOVER, F. e KOCHENBERGER, G.. *Handbook of Metaheuristics*, Kluwer Academic Publishers, Boston, MA. pp. 105-144.
39. MOSCATO, P.. (1989). *On evolution, search, optimization, genetic algorithms and martial arts: towards memetic algorithms*. Relatório técnico. California Institute of Technology, Pasadena, California, USA.
40. DAWKINS, R.. (1976). *The Selfish Gene*. Clarendon Press, Oxford.
41. KENNEDY, J. e EBERHART, R.. (1995). *Particle Swarm Optimization*. Proceedings of IEEE International Conference on Neural Networks, pp 1942–1948.
42. *Particle Swarm Optimization Tutorial*. Disponível em http://www.swarmintelligence.org/tutorials.php
43. ABRAHAM, A., HONGBO, L. e ZHAO, M.. (2008). *Particle Swarm Scheduling for Work-Flow Applications in Distributed Computing Environments*. Capítulo do Livro "*Metaheuristics for Scheduling in Industrial and Manufacturing Applications*", editora Springer.
44. SHA, D. Y. e HSU, C.. (2006). *A hybrid particle swarm optimization for job shop scheduling problem*. Computers and Industrial Engineering. Vol. 51, Issue 4, pp. 791-808.
45. LIAO, C., TSENG, C., LUARN, P.. (2007). *A discrete version of particle swarm optimization for flowshop scheduling problems*. Computers and Operations Research, Vol. 34, Issue 10, pp. 3099-3111
46. KUO, I., HORNG, S., KAO, T., LIN, T. e FAN, P.. (2007). *An efficient flow-shop scheduling algorithm based on a hybrid particle swarm optimization model*. Lecture Notes in Computer Science. Volume 4570, pp. 303-312.
47. SUTAR, S.R., SAWANT J.P. e JADHAV J.R.. (2008). *Task Scheduling For Multiprocessor Systems Using Memetic Algorithms*. Disponível em: http://www.comp.brad.ac.uk/het-net/tutorials/P27.pdf.
48. BURKE, E. e SMITH, A.. (1999). *A memetic algorithm to schedule planned maintenance for the national grid*. Journal of Experimental Algorithmics 4 (4): 1–13.
49. FRANÇA, P.M., MENDES, A.S. e MOSCATO P.. (1999). *Memetic algorithms to minimize tardiness on a single machine with sequence-dependent setup times*. Em: Proceedings of the 5th International Conference of the Decision Sciences Institute, Atenas, Grécia, pp. 1708–1710.
50. LEE, C.Y.. (1994). *Genetic algorithms for single machine job scheduling with common due date and symmetric penalties*. Journal of the Operations Research Society of Japan, 37(2), 83–95.
51. MILLER, D.M., CHEN, H.C., MATSON J. e LIU Q.. (1999). *A hybrid genetic algorithm for the single machine scheduling problem*. Journal of Heuristics, 5(4), 437–454.
52. CHENG R. e GEN M.. (1996). *Parallel machine scheduling problems using memetic algorithms.* Em: 1996 IEEE International Conference on Systems, Man and Cybernetics. Information Intelligence and Systems, Vol. 4. IEEE, Nova Iorque, NY, pp. 2665–2670.

53. MENDES, A.S., MULLER F.M., FRANÇA P.M. e MOSCATO P.. (1999). *Comparing metaheuristic approaches for parallel machine scheduling problems with sequence-dependent setup times*. Em: Proceedings of the 15th International Conference on CAD/CAM Robotics & Factories of the Future, Águas de Lindóia, Brasil.
54. MIN, L. e CHENG, W.. (1998). *Identical parallel machine scheduling problem for minimizing the makespan using genetic algorithm combined with simulated annealing*. Chinese Journal of Electronics, 7(4), 317–321.
55. NORDSTROM A.L. e TUFEKCI, S.. (1994). *A genetic algorithm for the talent scheduling problem*. Computers & Operations-Research, 21(8), 927–940.
56. OZDAMAR, L.. (1999). *A genetic algorithm approach to a general category project scheduling problem*. IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews), 29(1), 44–59.
57. RAMAT, E., VENTURINI, G., LENTE, C. e SLIMANE, M.. (1997). *Solving the multiple resource constrained project scheduling problem with a hybrid genetic algorithm*. Em: Proceedings of the Seventh International Conference on Genetic Algorithms. Morgan Kaufmann, San Francisco CA, pp. 489–496.
58. CHENG, R., GEN, M. e TSUJIMURA, Y.. (1999). *A tutorial survey of job-shop scheduling problems using genetic algorithms, ii. hybrid genetic search strategies*. Computers & Industrial Engineering, 37(1–2), 51–55.
59. FANG, J. e XI, Y.. (1997). *A rolling horizon job shop rescheduling strategy in the dynamic environment*. International Journal of Advanced Manufacturing Technology, 13(3), 227–232.
60. LIAW, C.F.. (2000). *A hybrid genetic algorithm for the open shop scheduling problem*. European Journal of Oprational Research, 124(1), 28–42.
61. MURATA, T. e ISHIBUCHI, H.. (1994). *Performance evaluation of genetic algorithms for flowshop scheduling problems*. Em: Proceedings of the First IEEE Conference on Evolutionary Computation, Vol. 2. IEEE, New York, NY, pp. 812–817.
62. MURATA, T., ISHIBUCHI, H. e TANAKA, H.. (1996). *Genetic algorithms for flowshop scheduling problems*. Computers & Industrial Engineering, 30(4), 1061-1071.
63. YANG, J.H., LIANG, S., LEE, H.P., YANG, Q. e LIANG, Y.C.. *Clonal Selection Based Memetic Algorithm for Job Shop scheduling problems*. Journal of Bionic Engineering, 5 (2008): 111-119. China.
64. COSTA, D.. (1995). *An evolutionary tabu search algorithm and the NHL scheduling problem*. INFOR, 33(3), 161–178.
65. HOLLAND, J.H.. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, Michigan.
66. REEVES, C.R.. (2003). *Genetic Algorithms*. Em: GLOVER, F. e KOCHENBERGER, G.. *Handbook of Metaheuristics*, Kluwer Academic Publishers, Boston, MA. pp. 55-82.
67. http://www.tc.bham.ac.uk/~roy/Research/ga.html
68. REEVES, C.R.. (1997). *Genetic algorithms for the Operations Researcher*. INFORMS Journal on Computing, 9, 231–250.
69. GOLDBERG, D.E.. (1985). *Optimal initial population size for binary-coded genetic algorithms*. TCGA Report 85001, University of Alabama, Tuscaloosa.
70. GOLDBERG, D.E.. (1989). *Sizing populations for serial and parallel genetic algorithms*. Em: Proceedings of 3rd International Conference on Genetic Algorithms, Morgan Kaufmann, San Mateo, CA.
71. REEVES, C.R.. (1995). *A genetic algorithm for flowshop sequencing*. Computers & Operations Research, 22, 5–13.
72. AHUJA, R.K. e ORLIN J.B.. (1997). *Developing fitter GAs*. INFORMS Journal on Computing, 9, 251–253.
73. DAVIS, L.. (1991). *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, Nova Iorque.
74. http://en.wikipedia.org/wiki/Genetic_algorithm
75. WRIGHT, A.H.; et al. (2003). *Implicit Parallelism*. Em: Proceedings of the Genetic and Evolutionary Computation Conference.
76. WALL, M.. (1996). *A Genetic Algorithm for Resource-Constrained Scheduling*. Tese de Doutorado. Massachusetts Institute of Technology.
77. MOORE, M.. 2004. *An Accurate Parallel Genetic Algorithm to Schedule Tasks on a Cluster*. Em: *Parallel Computing*, 30, pp 567-583, 41.
78. KOTECHA, K., SANGHANI, G. e GAMBHAVA, N.. (2004). *Genetic Algorithm for Airline Crew Scheduling Problem Using Cost-Based Uniform Crossover*. Em: AACC 2004, pp 84-91.
79. DUENAS, A., TUTUNCU, G.Y. e CHILCOTT, J.B.. (2008). *A genetic algorithm approach to the nurse scheduling problem with fuzzy preferences*. IMA Journal of Management Mathematics.
80. http://www.codeproject.com/KB/recipes/GaClassSchedule.aspx