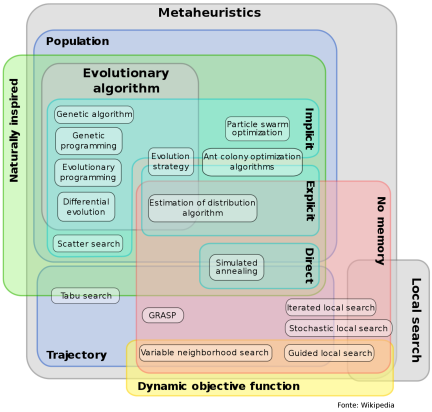


Notas de aula

Marcus Ritt
mrpritt@inf.ufrgs.br

11 de Junho de 2014

Universidade Federal do Rio Grande do Sul
Instituto de Informática
Departamento de Informática Teórica

Versão 5203 do 2014-06-11, compilada em 11 de Junho de 2014.

Agradecimentos Agradeço os estudantes da primeira edição dessa disciplina em 2013 por críticas e comentários e em particular o Tadeu Zubaran por diversas correções e sugestões.

Conteúdo

1. Introdução

Um *problema de busca* é uma relação binária $\mathcal{P} \subseteq I \times S$ com instâncias $x \in I$ e soluções $y \in S$. O par $(x, y) \in \mathcal{P}$ caso y é uma solução para x.

Definição 1.1
A classe de complexidade FNP contém os problemas de busca com relações $\mathcal{P}$ polinomialmente limitadas (ver definição 1.3) tal que $(x, y) \in \mathcal{P}$ pode ser decidido em tempo polinomial.
A classe de complexidade FP contém os problemas em FNP para quais existe um algoritmo polinomial A com

$$A(x) = \begin{cases} y & \text{para um } y \text{ tal que } (x, y) \in \mathcal{P} \\ \text{“insolúvel”} & \text{caso não existe } y \text{ tal que } (x, y) \in \mathcal{P} \end{cases}.$$

Teorema 1.1
FP=FNP se e somente se P=NP.

Prova. Ver por exemplo Papadimitriou (1993, cáp. 10.3). ■

Definição 1.2
Um *problema de otimização* $\Pi = (\mathcal{P}, \varphi, \text{opt})$ é uma relação binária $\mathcal{P} \subseteq I \times S$ com instâncias $x \in I$ e soluções $y \in S$, junto com

- uma função de otimização (função de objetivo) $\varphi : \mathcal{P} \to \mathbb{N}$ (ou $\mathbb{Q}$).
- um objetivo: Encontrar mínimo ou máximo

$$\text{OPT}(x) = \text{opt}\{\varphi(x, y) \mid (x, y) \in \mathcal{P}\}$$

  junto com uma solução y^* tal que $f(x, y^*) = \text{OPT}(x)$.

O par $(x, y) \in \mathcal{P}$ caso y é uma solução para x.

Uma instância x de um problema de otimização possui soluções $S(x) = \{y \mid (x, y) \in \mathcal{P}\}$.

Convenção 1.1
Escrevemos um problema de otimização na forma

NOME

Instância x

Solução y

Objetivo Minimiza ou maximiza $\varphi(x, y)$.

Com um dado problema de otimização correspondem três problemas:

- Construção: Dado x, encontra a solução ótima y^* e seu valor $\mathrm{OPT}(x)$.
- Avaliação: Dado x, encontra valor ótimo $\mathrm{OPT}(x)$.
- Decisão: Dado x e k, decide se $\mathrm{OPT}(x) \geq k$ (maximização) ou $\mathrm{OPT}(x) \leq k$ (minimização).

Definição 1.3
Uma relação binária R é *polinomialmente limitada* se

$$\exists p \in \mathsf{poly} : \forall (x, y) \in R : |y| \leq p(|x|).$$

Definição 1.4 (Classes de complexidade)
A classe PO consiste dos problemas de otimização tal que existe um algoritmo polinomial A com $\varphi(x, A(x)) = \mathrm{OPT}(x)$ para $x \in I$.
A classe NPO consiste dos problemas de otimização tal que

(i) As instâncias $x \in I$ são reconhecíveis em tempo polinomial.

(ii) A relação $\mathcal{P}$ é polinomialmente limitada.

(iii) Para y arbitrário, polinomialmente limitado: $(x, y) \in \mathcal{P}$ é decidível em tempo polinomial.

(iv) φ é computável em tempo polinomial.

1.1. Não tem almoço de graça

> "Sire in eight words I will reveal to you all the wisdom that I have distilled through all these years from all the writings of all the economists who once practiced their science in your kingdom. Here is my text: 'There ain't no such thing as free lunch' " (NN 1938)

A frase "there ain't no such thing as free lunch" (TANSTAFEL) expressa que uma vantagem (p.ex. o almoço de graça em bares dos EUA no século 19) tipicamente é pago de outra forma (p.ex. comida salgada e bebidas caras). Para problemas de busca e de otimização, Wolpert e Macready (1997) provaram teoremas que mostram que uma busca universal não pode ter uma vantagem em todos problemas de otimização.
Para um problema de otimização supõe que $\varphi : \mathcal{P} \to \Phi$ é restrito para um conjunto finito Φ, e seja $\mathcal{F} = \Phi^{S(x)}$ espaço de todas funções objetivos para uma instância do problema. Um algoritmo de otimização avalia pares de soluções com o seu valor $(s, v) \in S(x) \times \Phi$. Seja $D = \cup_{m \geq 0}(S(x) \times \Phi)^m$ o conjunto de todas sequencias de pares. Um algoritmo de otimização que não repete avaliações pode ser modelado por uma função $a : d \in D \to \{s \mid s \neq s_i, \text{para } d_i = (s_i, v_i), i \in [|d|]\}$ que mapeia a sequencia atual para a próxima solução a ser avaliada (observe que o algoritmo toma essa decisão em função das soluções anteriormente visitadas e os seus valores). A avaliação de um algoritmo de otimização é através uma função $\Phi(d)$. Ela pode, por exemplo, atribuir a d o valor mínimo encontrado durante a busca.

Teorema 1.2 (Wolpert e Macready (1997))
Para algoritmos a, a', um número de passos m e uma sequencia de valores $v \in \Phi^m$

$$\sum_{f \in \mathcal{F}} P[v \mid f, m, a] = \sum_{f \in \mathcal{F}} P[v \mid f, m, a'].$$

O teorema mostra que uma busca genérica não vai ser melhor que uma busca aleatória em média sobre todas funções objetivos. Porém, uma grande fração das funções possíveis não ocorrem na prática (uma função aleatória é incompressível, i.e. podemos especificá-la somente por tabulação, funções práticos muitas vezes exibem localidade). Além disso, algoritmos de busca frequentemente aproveitam a estrutura do problema em questão.

1.2. Representação de soluções

A representação de soluções influencia as operações aplicáveis e a sua complexidade. Por isso a escolha de uma representação é importante para o desempenho de uma heurística. A representação também define o tamanho do espaço de busca, e uma representação compacta (e.g. 8 coordenadas versus permutações no problema das 8-rainhas) é preferível. Para problemas restritos uma representação implícita que é transformada para uma representação direta por um algoritmo pode ser vantajoso.

Para uma discussão abstrata usaremos frequentemente duas representações elementares. Na *representação por conjuntos* uma solução é um conjunto $S \subseteq U$ de um universo U. Os conjuntos válidos são dados por uma coleção $\mathcal{V}$ de subconjuntos de U. Na *representação por variáveis* uma instância é um subconjunto $I \subseteq U$, e uma solução é uma atribuição de valores de um universo V aos elementos em I.

Exemplo 1.1 (Representação do PCV por conjuntos)
Uma representação por conjuntos do PCV sobre um grafo $G = (V, A)$ é o universo de arestas $U = A$, com $\mathcal{V}$ todos subconjuntos que formam ciclos. ◊

Exemplo 1.2 (Representação do PCV por variáveis)
Uma representação por variáveis do PCV sobre um grafo $G = (V, A)$ usa um universo de vértices U. Uma instância $I = V$ atribui a cada cidade a próxima cidade no ciclo. Uma representação alternativa usa $I = [n]$ a atribui a cada variável $i \in I$ a i-ésima cidade no ciclo. ◊

Exemplo 1.3 (Representação da coloração de grafos por variáveis)
Seja U um universo de vértices e C um universo de cores. Uma representação da uma instância $G = (V, A)$ do problema da coloração de grafos usa variáveis $V \subseteq Q$ e atribui cores de C às variáveis. ◊

1.2.1. Reduções de problemas

Não todos elementos do universo são usados em soluções ótimas: frequentemente eles tem que satisfazer certos critérios para participar numa solução ótima. Isso permite reduzir o problema para um *núcleo*. No problema do PCV, por exemplo, arestas mais longas tem uma baixa probabilidade de participar de uma solução ótima, mas arestas bem curtas com alta probabilidade aparecem na solução ótima. No problema da mochila elementos de alta eficiência são mais usados, e de baixa eficiência menos. Se soubéssemos o arco de menor distância não usado numa solução ótima, e de maior distância usado, poderíamos reduzir o problema de acordo. Regras de redução para um núcleo são possíveis em diversos problemas (e.g. o problema da mochila (Kellerer et al. 2004)) e são essenciais para problemas tratáveis por parâmetro fixo (Niedermeier 2002).

Princípio de projeto 1.1 (Redução de problemas)
Busca por regras de redução do problema. Procura reduzir o problema para um núcleo heurístico.

1.2.2. Transformações entre representações

Um transformador recebe uma representação de uma solução e transforma ela numa representação diferente. Um algoritmo construtivo randomizado (ver capítulo 3) pode ser visto como um algoritmo que transforma uma sequencia de números aleatórios em uma solução explicita. Ambas são representações válidas da mesma solução. Essa ideia é aplicada também em algoritmos genéticos, onde a representação fonte se chama *fenótipo* e a representação destino *genótipo*. A ideia de representar uma solução por uma sequencia de números aleatórios é usado diretamente em algoritmo genéticos com chaves aleatórias (ver 4.5.6).
Uma transformação é tipicamente sobrejetiva ("many-to-one"), i.e. existem várias representações fonte para uma representação destino. Idealmente, existe o mesmo número de representações fontes para representações destino, para manter a mesma distribuição de soluções nos dois espaços.

Exemplo 1.4 (Representação de permutações por chaves aleatórias)
Uma permutação de n elementos pode ser representada por n números aleatórios reais em $[0, 1]$. Para números aleatórios são $a_1, \ldots, a_n$, seja π uma permutação tal que $a_{\pi(1)} \leq \cdots a_{\pi(n)}$. Logo os números a_i representam a permutação π (ou π^{-1}). ◊

Uma transformação pode ser útil caso o problema possui muitas restrições e o espaço de busca definido por uma representação direta contém muitas soluções inválidas.

Exemplo 1.5 (Coloração de vértices)
Uma representação direta da coloração de vértices pode ser uma atribuição de cores a vértices. Para um limite de no máximo n cores, temos n^n possíveis atribuições, mas várias são infactíveis. Uma representação indireta é uma permutação de vértices. Para uma dada permutação um algoritmo guloso processa os vértices em ordem e atribui o menor cor livre ao vértice atual. A corretude dessa abordagem mostra

Lema 1.1
Para uma dada k-coloração, sejam $C_1 \cup \cdots \cup C_k$ as classes de cores. Ordenando os vértices por classes de cores, o algoritmo guloso produz uma coloração com no máximo k cores.

Prova. Mostraremos por indução que a coloração das primeiras i classes não precisa mais que i cores. Para a primeira classe isso é óbvio. Supõe que na coloração da classe i precisamos usar a cor $i+1$. Logo existe um vizinho com cor i. Mas pela hipótese da indução o vizinho não pode ser de uma classe

menor. Logo, temos uma aresta entre dois vértices da mesma classe, uma contradição. ■
Com essa representação, todas soluções são válidas. Observe que o tamanho do espaço da busca $n! \approx \sqrt{2\pi n}(n/e)^n$ (por A.5) é similar nas duas representações. ◊

Por fim, transformações podem ser úteis caso podemos resolver subproblemas restritos do problema eficientemente.

Exemplo 1.6 (Sequenciamento em máquinas paralelas não relacionadas)
Uma solução para $R \parallel \sum w_j C_j$ direta é uma atribuição das tarefas às máquinas, junto com a ordem das tarefas em cada máquina.

Teorema 1.3
A solução ótima de $1 \parallel \sum w_j C_j$ é uma sequencia em ordem de tempo de processamento ponderado não-decrescente $p_1/w_1 \leq \cdots \leq p_n w_n$.

Prova. Supõe uma sequencia ótima com $p_i/w_i > p_{i+1}/w_{i+1}$. A contribuição das duas tarefas à função objetivo é $w = w_i C_i + w_{i+1} C_{i+1}$. Trocando as duas tarefas a contribuição das restantes tarefas não muda, e a contribuição das duas tarefas é

$$w_{i+1}(C_{i+1} - p_i) + w_i(C_i + p_{i+1}) = w + w_i p_{i+1} - w_{i+1} p_i.$$

Logo a função objetivo muda por $\Delta = w_i p_{i+1} - w_{i+1} p_i$, mas pela hipótese $\Delta < 0$. ■
Logo a ordem ótima de uma máquina pode ser computada em tempo $O(n \log n)$, e uma representação reduzida mantém somente a distribuição das tarefas à máquinas. ◊

As diferentes representações compactas podem ser combinadas.

Exemplo 1.7 (Simple assembly line balancing)
No "simple assembly line balacing problem" do tipo 2 temos que atribuir n tarefas, restritas por precedências, à m de estações de trabalho. Cada tarefa possui um tempo de execução t_i, e o *tempo de estação* é o tempo total das tarefas atribuídas a uma estação. O objetivo é minimizar o maior tempo de estação.
Uma representação direta é uma atribuição de tarefas a estações, mas muitas atribuições são inválidas por não satisfazer as precedências entre as tarefas. Uma representação mais compacta atribui chaves aleatórias às tarefas. Com isso, uma ordem global das tarefas é definida: elas são ordenadas topologicamente, usando as chaves aleatórias como critério de desempate, caso duas

tarefas concorram para a próxima posição. Por fim, para uma dada ordem de tarefas, a solução ótima do problema pode ser obtida via programação dinâmica. Seja $C(i,k)$ o menor tempo de ciclo para tarefas $i,\dots,n$ em k máquinas, a solução ótima é $C(1,m)$ e C satisfaz

$$C(i,k) = \begin{cases} \min_{i\le j\le n} \max\{\sum_{i\le j'\le j} t_{j'}, C(j+1,k+1)\} & \text{para } i \le n,\ k > 0 \\ 0 & \text{para } i > n \\ \infty & \text{para } i \le n \text{ e } k = 0 \end{cases},$$

e logo a solução ótima pode ser obtida em tempo e espaço $O(nm)$ (pré-calculando as somas parciais). ◊

Essa observação é o motivo para o

Princípio de projeto 1.2 (Subproblemas)
Identifica os subproblemas mais difíceis que podem ser resolvidos em tempo polinomial e considera uma representação que contém somente a informação necessária para definir os subproblemas.

1.3. Estratégia de busca: Diversificação e intensificação

No projeto de uma heurística temos que balancear dois objetivos antagonistas: a *diversificação* da busca e a *intensificação* de busca. A diversificação da busca (também chamada *exploration*) procura garantir uma boa cobertura do espaço de busca, evitando que a soluções analisadas fiquem confinadas a uma região pequena do espaço total. A diversificação ideal é um algoritmo que repetidamente gera soluções aleatórias. Em contraste a *intensificação* (também chamada *exploitation*) procura melhorar a solução atual o mais possível. Um exemplo de uma intensificação seria analisar todas soluções dentro uma certa distância da solução atual.
O tema de intensificação e diversificação se encontra na discussão da heurísticas individuais na seções 2 a 4; um procedimento genérico de intensificação e diversificação é apresentado na seção 4.8.

1.4. Notas

Mais informações sobre os teoremas NFL se encontram no artigo original de Wolpert e Macready (1997) e em Burke e Kendall (2005, cáp. 11) e Rothlauf (2011, cáp. 3.4.4). Para um crítica ver p.ex. Hutter (2010). Talbi (2009, cáp. 1.4.1) discute outras representações de soluções.

2. Busca por modificação de soluções

2.1. Vizinhanças

Uma busca local procura melhorar uma solução de uma instância de um problema aplicando uma pequena modificação, chamada *movimento*. O conjunto de soluções que resultam de uma pequena modificação formam os *vizinhos* da solução.

Definição 2.1 (Vizinhança)
Uma *vizinhança* de uma instância x de um problema de otimização Π é uma função $N : S(x) \to 2^{S(x)}$. Para uma solução s, os elementos $N(s)$ são os *vizinhos* de s. Os *vizinhos melhores* de s são $B(s) = \{s' \in N(s) \mid \varphi(s') < \varphi(s)\}$. Uma vizinhança é *simétrica*, caso para $s' \in N(s)$ temos $s \in N(s')$.
Para uma dada vizinhança um *mínimo local* é uma solução s, tal que $\varphi(s) \leq \varphi(s')$ para $s' \in N(s)$ e um *máximo local* caso $\varphi(s) \geq \varphi(s')$ para $s' \in N(s)$. Caso uma solução é estritamente menor ou maior que os seus vizinhos, o ótimo local é *estrito*. Uma vizinhança é *exata*, caso cada ótimo local também é um ótimo global.

Definição 2.2 (Grafo de vizinhança)
O *grafo de vizinhança* $G = (V, E)$ para uma instância x de um problema de otimização Π com vizinhança N possui vértices $V = \{y \mid (x, y) \in P\}$ e arcos (s, s') para $s, s' \in S(x)$, $s' \in N(s)$. Para uma vizinhança simétrica, o grafo de vizinhança é efetivamente não-direcionado. Uma solução s' é alcançável a partir da solução s, caso existe um caminho de s para s' em G. Caso todo vértice é alcançável a partir de qualquer outro, G é *conectado*. Neste caso o *diâmetro* de G é o comprimento do maior caminho mais curto entre dois vértices em G. O grafo G é *fracamente otimamente conectada* caso a partir de cada solução s uma solução ótima é alcançável.

Uma vizinhança é suficiente para definir uma busca local genérica. Ela seleciona um vizinho de acordo com uma distribuição $\hat{P}_s$ sobre a *vizinhança fechada* $\hat{N}(s) = \{s\} \cup N(s)$. Para uma distribuição P_s sobre $N(s)$, a extensão padrão para a vizinhança fechada é definida por

$$\hat{P}_s(s') = \begin{cases} 1 - \sum_{s' \in N(s)} P_s(s') & \text{para } s' = s \\ P_s(s') & \text{caso contrário} \end{cases}$$

Algoritmo 2.1 (LocalSearch)
Entrada Solução inicial s, vizinhança N, distribuição P_s.

Saída Uma solução com valor no máximo $\varphi(s)$.

```
1  LocalSearch(s)=
2    s* := s
3    repeat
4      seleciona s' ∈ N̂(s) de acordo com P̂_s
5      s := s'
6      if φ(s) < φ(s*) then s* := s
7    until critério de parada satisfeito
8    return s*
9  end
```

A complexidade de uma busca local depende da complexidade da seleção e do número de iterações. A complexidade da seleção muitas vezes é proporcional ao tamanho da vizinhança $|N(s)|$.
Duas estratégias básicas para uma busca local são

Caminhada aleatória (ingl. random walk) Para $N(s) \neq \emptyset$, define $P_s(s) = 1/|N(s)|$.

Amostragem aleatória (ingl. random picking) Uma caminhada aleatória com $N(s) = S(x)$ para todo $s \in S(x)$.

Melhor vizinho Para $B(s) \neq \emptyset$, define $B^*(s) = \{s' \in B(s) \mid \varphi(s') = \min_{s'' \in B(s)} \varphi(s''$ e $P_s(s') = 1/|B^*(s)|$ para $s' \in B^*(s)$. Esse estratégia tipicamente não consegue sair de mínimos locais e tem que ser modificado por uma das técnicas discutidas em 2.3, mas supera plateaus.

Exemplo 2.1 (Polítopos e o método Simplex)
O método Simplex define uma vizinhança entre os vértices do polítopo de um programa linear: cada par variável entrante e sainte admissível define um vizinho. Essa vizinhança é simétrica, conectada, fracamente otimamente conectada e exata. Logo o método resolve o problema da programação linear. ◊

Exemplo 2.2 (k-exchange para o PCV)
Uma vizinhança para o PCV é k-exchange Croes (1958): os vizinhos de um ciclo são obtidos removendo k arcos, e conectando os k caminhos resultantes de outra forma. Para qualquer k fixo, essa vizinhança é simétrica, conectada,

fracamente otimamente conectada, mas inexata (por quê?). O tamanho da vizinhança é $O = (\binom{n}{k}k!2^k) = O(n^k)$ para n cidades e k fixo.

$\Diamond$

Exemplo 2.3 (k-SAT)
O problema k-SAT é decidir se existe uma atribuição $x \in \{0,1\}^n$ que satisfaz uma fórmula $\varphi(x)$ da lógica proposicional em forma normal conjuntiva com k literais por cláusula.
Seja $|x-y|_1 = \sum_{i\in[n]}[x_i \neq y_i]$ a distância Hamming entre dois vetores $x, y \in \{0,1\}^n$. Uma vizinhança conhecida para SAT é k-flip: os vizinhos de uma solução são todas soluções de distância Hamming k. A vizinhança é simétrica, fracamente otimamente conectada para $k = 1$, mas inexata. O tamanho da vizinhança é $O(n^k)$.

$\Diamond$

Observação 2.1 (Cálculo eficiente da função objetivo)
Frequentemente é mais eficiente avaliar a diferença $\Delta(s, s') = \varphi(s') - \varphi(s)$ para determinar o valor da função objetivo de um vizinho. No exemplo 2.2 avaliar $\varphi(s)$ custa $O(n)$, mas avaliar $\Delta(s, s')$ custa $O(1)$. Logo, determinar o melhor vizinho na vizinhança 2-exchange, por exemplo, custa $O(n^3)$ na abordagem ingênua, mas é possível em $O(n^2)$ avaliando as diferenças.
Em alguns casos a avaliação da diferença das diferenças é ainda mais eficiente. Um exemplo é a *programação quadrática binária* com função objetivo

$$\varphi(s) = \sum_{i,j\in[n]} q_{ij}x_ix_j$$

e coeficientes simétricos ($Q = Q^t$). Avaliar $\varphi(s)$ custa $\Theta(n^2)$, avaliar a diferença na vizinhança 1-flip que troca $x'_k = 1 - x_k$ para um k fixo

$$\begin{aligned}
\Delta_k(s', s) &= \sum_{i,j\in[n]} q_{ij}x'_ix'_j - \sum_{i,j\in[n]} q_{ij}x_ix_j \\
&= \sum_{j\in[n]\setminus\{k\}} q_{kj}(x'_k - x_k)x_j + \sum_{j\in[n]\setminus\{k\}} q_{jk}x_j(x'_k - x_k) + q_{kk}({x'_k}^2 - x_k^2) \\
&= (1 - 2x_k)\big(q_{kk} + 2\sum_{j\in[n]\setminus\{k\}} q_{jk}x_j\big)
\end{aligned}$$

custa somente $O(n)$.
Atualizando um bit l por $x'_l = 1 - x_l$ obtemos novas diferenças

$$\Delta'_k = \begin{cases} -\Delta_k & \text{caso } l = k \\ \Delta_k + 2q_{lk}(1 - 2x_k)(1 - 2x_l) & \text{caso contrário.} \end{cases} \tag{2.1}$$

Dado os valores Δ_k podemos encontrar o melhor vizinho em tempo $O(n)$. Passando para o melhor vizinho, podemos atualizar todos valores Δ_k em tempo $O(n)$ usando (2.1). Logo, o custo de encontrar o melhor vizinho é $\Theta(n^3)$ avaliando soluções completas, somente $\Theta(n^2)$ calculando as diferenças, e somente $O(n)$ atualizando diferenças. ◊

2.1.1. Vizinhanças reduzidas

Uma técnica comum para melhorar o desempenho de buscas locais é reduzir a vizinhança heuristicamente, excluindo vizinhos com características que com baixa probabilidade se encontram em soluções de boa qualidade. Uma forma comum de reduzir a vizinhança é usar *listas de candidatos* (ingl. candidate lists).

Exemplo 2.4 (Vizinhança reduzida para o PCV)
No caso do 2-exchange para o PCV muitas das $\Theta(n^2)$ vizinhos produzem rotas inferiores, porque eles introduzem uma arestas longas, caso as duas arestas originais ficam muito distantes. Logo é possível reduzir a vizinhança heuristicamente, sem expectativa de perder soluções boas. Uma estratégia de proposto por Johnson e McGeoch (2003) é: escolher uma cidade aleatória, um vizinho aleatório dessa cidade na rota, uma terceira cidade entre os 20 vizinhos mais próximos de segunda cidade, e a quarta cidade como sucessor da terceira na orientação da rota dado pelas primeiras duas cidades. Com isso uma rota tem no máximo $40n$ vizinhos. ◊

A redução de vizinhanças frequentemente é uma estratégia importante para obter resultados de boa qualidade (Johnson e McGeoch 2003; Toth e Vigo 2003; Glover e Laguna 1997), motivo para

Princípio de projeto 2.1 (Redução de vizinhanças)
Considera eliminar das vizinhanças movimentos com baixa probabilidade de melhorar a solução.

2.2. Buscas locais monótonas

Uma busca local monótona permite somente modificações que melhoram a solução atual, i.e. no algoritmo LocalSearch sempre temos $P_s(s') = 0$ para $s' \notin$

$B(s)$. Logo, o algoritmo termina num ótimo local. Pela monotonia também não é necessário guardar a melhor solução encontrada. A busca depende da estratégia de seleção da nova solução s', também conhecida como *regra de pivoteamento*.

Algoritmo 2.2 (LocalDescent)
Entrada Solução inicial s, vizinhança N, distribuição P_s.

Saída Uma solução com valor no máximo $\varphi(s)$.

```
1  LocalDescent(s):=
2    repeat
3      seleciona s' ∈ N̂(s) de acordo com P̂_s
4      s := s'
5    until P_s(s) = 1
6    return s
7  end
```

Descida aleatória (ingl. stochastic hill descent) Para $B(s) \neq \emptyset$ define $P_s(s') = 1/|B(s)|$ para $s' \in B(s)$. Esta estratégia é equivalente com a primeira melhora, mas em ordem aleatória.

Primeira melhora (ingl. first improvement) A primeira melhora supõe uma vizinhança ordenada $B(s) = \{b_1, \ldots, b_k\}$. Ela seleciona $f = \min\{i \mid \varphi(b_i) < \varphi(s)\}$, i.e. $P_s(b_i) = [i = f]$. O método é conhecido pelos nomes "hill climbing" (no caso de maximização) ou "hill descent" (no caso de minimização).

Melhor melhora (ingl. best improvement) Para $B(s) \neq \emptyset$, define $B^*(s) = \{s' \in B(s) \mid \varphi(s') = \min_{s'' \in B(s)} \varphi(s'')\}$ e $P_s(s') = 1/|B^*(s)|$ para $s' \in B^*(s)$. O método é conhecido pelos nomes "steepest ascent" (no caso de maximização) ou "steepest descent" (no caso de minimização).

Busca por amostragem (ingl. sample search) Seleciona um subconjunto $S \subseteq N(x)$ aleatório de tamanho $\alpha|N(x)|$, define $B^*(s) = \{s' \in B(s) \mid \varphi(s') = \min_{s'' \in S} \varphi(s'')$ e $P_s(s') = 1/|B^*(s)|$ para $s' \in B^*(s)$.

As estratégias obviamente podem ser combinadas, por exemplo, aplicar uma estratégia de "primeira melhora" após uma amostragem.
A qualidade de uma busca local depende da vizinhança: para vizinhanças maiores esperamos encontrar ótimos locais melhores. Porém a complexidade da busca cresce com a vizinhança. A arte, então, consiste em balancear estes dois objetivos.

Exemplo 2.5 (Método Simplex)
Não conhecemos regras de pivoteamento para o método Simplex que garantem uma complexidade polinomial. Porém, a programação linear possui soluções polinomiais (que não usam busca local). Por isso, a complexidade de encontrar ótimos locais pode ser menor que a complexidade do método iterativo. ◊

Exemplo 2.6 (Árvore geradora mínima)
Para uma árvore geradora, podemos definir vizinhos como segue: adicione uma aresta, e remove outra do (único) ciclo formado. Uma árvore geradora é mínima se e somente se não existe melhor vizinho (prova: exercício). Por isso a busca local resolve o problema de encontrar a árvore geradora mínima. A vizinhança é simétrica, fracamente otimamente conectada e exata. Porém, a busca local geralmente não é eficiente. ◊

Exemplo 2.7 (OneMax)
Para um $x^* \in \{0,1\}^n$ fixo o problema OneMax consiste encontrar o mínimo de $\varphi(x) = |x - x^*|_1$, i.e. x^*. O número de bits X corretos de uma solução aleatória satisfaz $E[X] = n/2$ e $\Pr[X \leq n/3] \leq e^{-n/36}$ e $\Pr[X \geq 2n/3] \leq e^{-n/54}$ (aplicando limites de Chernoff (A.4)).
Uma descida aleatória precisa tempo $O(n)$ para selecionar um vizinho, avaliando a função objetivo em $O(1)$ e sem repetição, e $O(n)$ passos, para um tempo total de $O(n^2)$. Uma análise mais detalhada do caso médio é a seguinte: para selecionar um vizinho melhor, podemos repetidamente selecionar um vizinho arbitrário, até encontrar um vizinho melhor. Com i bits diferentes, encontramos um vizinho melhor com probabilidade i/n. Logo a seleção precisa esperadamente n/i passos até encontrar um vizinho melhor (ver lema A.5) e logo no máximo

$$\sum_{1 \leq i \leq n} n/i = nH_n \approx n \log n$$

passos até encontrar x^*.
A primeira melhora precisa no pior caso (todos bits diferentes) tempo esperado $\Theta(n/i)$ para encontrar um vizinho melhor, e a melhor melhora tempo $\Theta(n)$. Logo, ambas precisam tempo $\Theta(n^2)$ para encontrar x^*. ◊

Exemplo 2.8 (GSAT)
O algoritmo GSAT (Selman et al. 1992) aplica a estratégia "melhor vizinho" na vizinhança 1-flip com função objetivo sendo o número de cláusulas satisfeitas (observe que é importante escolher entre os melhores uniformemente). Ele periodicamente recomeça a busca a partir de uma solução aleatória. ◊

Exemplo 2.9 (WalkSAT)
WalkSAT usa uma estratégia de seleção mais sofisticada: em cada passo uma cláusula não satisfeita é selecionada, e uma variável aleatória dessa cláusula é invertida. (O WalkSAT proposto por Selman et al. (1994) seleciona uma variável que não invalida nenhuma outra cláusula ou com probabilidade p uma que invalide o menor número e com probabilidade $1-p$ uma aleatória.) Logo a vizinhança é um subconjunto da vizinhança 1-flip. WalkSAT também recomeça a busca a partir de uma solução aleatória periodicamente.

Lema 2.1 (Schöning (1999))
Seja φ uma fórmula em k-CNF satisfatível com n variáveis. O algoritmo WalkSAT com período $3n$ precisa esperadamente $O(n^{3/2}(2(k-1)/k)^n)$ passos até encontrar uma atribuição que satisfaz φ.

Prova. Seja a uma atribuição que satisfaz φ. Vamos determinar a probabilidade q que um período de WalkSAT encontra a. Com $p_j = \binom{n}{j}2^{-n}$ a probabilidade de iniciar com distância Hamming j de a, e q_j a probabilidade de encontrar a a partir da distância j, temos

$$q = \sum_{0\le j\le n} p_j q_j. \qquad (*)$$

A distância Hamming para a diminui com probabilidade pelo menos $1/k$ e aumenta com probabilidade no máximo $1-1/k$. Podemos modelar o WalkSAT como caminhada aleatória entre classes de soluções com distância Hamming j, com uma probabilidade de transição de j para $j-1$ ("para baixo") de $1/k$ e de j para $j+1$ ("para acima") de $1-1/k$. Com isso q_j é pelo menos a probabilidade de chegar na classe 0 a partir da classe j em no máximo $3n$ passos. Para conseguir isso podemos fazer j passos para baixo, ou $j+l$ para baixo e um para acima, e no geral $j+l$ para baixo e l para acima. Logo

$$q_j \ge \max_{0\le l\le (3n-j)/2} \binom{j+2l}{l}\left(\frac{k-1}{k}\right)^l\left(\frac{1}{k}\right)^{j+l}.$$

Para $l=\alpha j$ com $\alpha\in(0,1)$ temos

$$q_j \ge \binom{(1+2\alpha)j}{\alpha j}\left(\left(\frac{k-1}{k}\right)^{\alpha}\left(\frac{1}{k}\right)^{(1+\alpha)}\right)^j.$$

Aplicando o lema A.2 é podemos estimar[1]

$$\binom{(1+2\alpha)j}{\alpha j} \ge (8j)^{-1/2}\left(\left(\frac{1+2\alpha}{\alpha}\right)^{\alpha}\left(\frac{1+2\alpha}{1+\alpha}\right)^{1+\alpha}\right)^j$$

[1] Substituindo diretamente é descartando o fator $\sqrt{(1+2\alpha)/(\alpha(1+\alpha))} \ge 1$.

e logo

$$q_j \geq (8j)^{-1/2} \left(\left(\frac{1+2\alpha}{\alpha} \right)^{\alpha} \left(\frac{1+2\alpha}{1+\alpha} \right)^{1+\alpha} \left(\frac{k-1}{k} \right)^{\alpha} \left(\frac{1}{k} \right)^{(1+\alpha)} \right)^{j}.$$

Escolhendo $\alpha = 1/(k-2)$ e simplificando obtemos

$$q_j \geq (8j)^{-1/2} \left(\frac{1}{k-1} \right)^{j}.$$

Finalmente, substituindo em (*)

$$\begin{aligned} q &\geq 2^{-n} + \sum_{j \in [n]} \binom{n}{j} 2^{-n} (8j)^{-1/2} \left(\frac{1}{k-1} \right)^{j} \\ &\geq 2^{-n} (8n)^{-1/2} \sum_{j \in [n]} \binom{n}{j} \left(\frac{1}{k-1} \right)^{j} 1^{n-j} \\ &= 2^{-n} (8n)^{-1/2} \left(1 + \frac{1}{k-1} \right)^{n} = \frac{1}{\sqrt{8n}} \left(\frac{k}{2(k-1)} \right)^{n}. \end{aligned}$$

Logo, o número esperado de períodos é

$$1/q = \sqrt{8n} \left(\frac{2(k-1)}{k} \right)^{n}$$

e como cada período precisa tempo $O(n)$ o resultado segue. ■
Para uma fórmula satisfatível com $k = 3$, por exemplo, o algoritmo precisa $O(n^{3/2}(4/3)^n)$ passos.
É possível transformar esta algoritmo num algoritmo randomizado que decide se uma fórmula é satisfatível com alta probabilidade. ◊

Exemplo 2.10 (2-opt para o PCV)
A estratégia 2-opt para o PCV é uma descida aleatória na vizinhança 2-exchange. Similarmente, obtemos k-opt na vizinhança k-exchange.

Teorema 2.1 (Chandra et al. (1999))
Para $k \geq 2$, $n \geq 2k + 8$ e para $\alpha > 1/n$ existe uma instância x do PCV com n cidades, tal que

$$\frac{\text{k-opt}(x)}{\text{OPT}(x)} > \alpha.$$

Prova. Para um k par, define distâncias

$$\begin{aligned} d_{12} &= 1 && \\ d_{i,i+1} = d_{n,1} &= 1/n\alpha && i \in [2, n) \\ d_{k+3,2k+4} &= 1/n\alpha && \\ d_{j,2k+4-j} &= 1/n\alpha && j \in [k] \\ d_{i,j} &= kn && \text{caso contrário} \end{aligned}$$

Um ciclo Hamiltoniano ótimo é dado por arestas $(i, \text{próximo}(i))$ com

$$\text{próximo}(i) = \begin{cases} 2k+4-i & \text{para } i \text{ impar e } i < k \\ i+1 & \text{para } i \text{ par e } i < k \\ i+1 & \text{para } i \in [k, k+2] \\ 2k+4 & \text{para } i = k+3 \\ i-1 & \text{para } i \text{ impar e } i \in [k+3, 2k+4) \\ 2k+4-i & \text{para } i \text{ par e } i \in [k+3, 2k+4) \\ i+1 & \text{para } i \in [2k+4, n] \\ 1 & \text{para } i = n \end{cases}$$

A otimalidade segue do fato que todas arestas possuem o peso mínimo $1/n\alpha$. Este ciclo é o único ciclo ótimo (Exercício!). Por outro lado, o ciclo $(1, 2, \ldots, n)$ possui peso total $1 + (n-1)/n\alpha$, mas tem $k+1$ arestas diferentes. Logo este ciclo é um mínimo local para k-exchange e para a instância acima temos

$$\frac{k\text{-opt}(x)}{\text{OPT}(x)} \geq \alpha + 1 - 1/n > \alpha.$$

Para provar o caso para um k impar, podemos observar que um mínimo local para o $k+1$-exchange, também é um mínimo local para k-exchange. ■

Teorema 2.2 (Chandra et al. (1999))
No caso métrico $2\text{-opt}(x)/\text{OPT}(x) \leq 4\sqrt{n}$.

Antes provaremos

Lema 2.2
Seja $(c_1, c_2, \ldots, c_n, c_{n+1} = c_1)$ um mínimo local de 2-opt, e $k \in [n]$ seja $E_k = \{(c_i, c_{i+1}) \mid d_{i,i+1} > 2\text{OPT}(x)/\sqrt{k}\}$. Então $|E_k| < k$.

Prova. Supõe que existe um k tal que $|E_k| \geq k$.

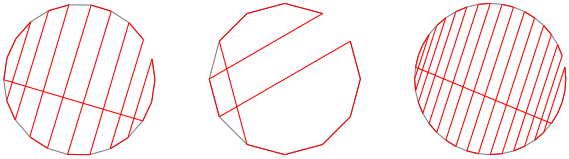


Figura 2.1.: Caminhos construídos na prova do teorema 2.1. Esquerda: $n = 22$, $k = 8$. Meio: $n = 12$, $k = 2$. Direita: $n = 40$, $k = 16$. A figura somente mostra arestas de distância $1/n\alpha$.

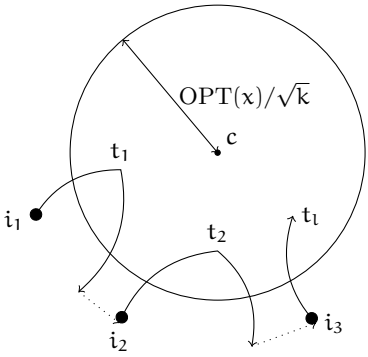


Figura 2.2.: Ilustração para o teorema 2.2.

A densidade de términos de arcos $(c_i, c_{i+1}) \in E_k$[2] não pode ser demais: Supõe que numa bola com centro c e raio $OPT(x)/\sqrt{k}$ temos términos $t_1, \ldots t_l$ com $l \geq \sqrt{k}$. Sejam $i_1, \ldots i_l$ os inícios correspondentes. Nenhum início esta na bola, por ser mais que $2OPT(x)/\sqrt{k}$ distante do término. Os términos, por estarem na bola, possuem distância no máximo $2OPT(x)/\sqrt{k}$ entre si. Logo, os inícios possuem uma distância mais que $2OPT(x)/\sqrt{k}$ entre si: caso contrário, para um par de inícios i_a, i_b com distância menos que $2OPT(x)/\sqrt{k}$ a solução que aplica um 2-exchange substituindo (i_a, t_a) e (i_b, t_b) por (i_a, i_b) e (t_a, t_b) séria melhor, uma contradição com a minimalidade local.
Logo tem pelo menos $\sqrt{k}$ inícios com distância pelo menos $2OPT(x)/\sqrt{k}$. Mas uma rota mínima entre eles possui distância pelo menos $2OPT(x)$, uma contradição. Isso mostra que numa bola de raio $OPT(x)/\sqrt{k}$ temos menos que $\sqrt{k}$ términos.

[2] O término de (u, v) é v, o início u.

Por consequência, em E_k existem pelo menos $\sqrt{k}$ términos com distância mais que $OPT(x)/\sqrt{k}$ entre si: começando com o conjunto de todos términos de arcos em E_k vamos escolher cada vez um, e removê-lo junto com os términos com distância no máximo $OPT(x)/\sqrt{k}$ dele, até nenhum término sobrar. Como em cada passo removeremos no máximo $\sqrt{k}$ términos, o conjunto resultante possui pelo menos $\sqrt{k}$ términos. Mas então uma rota que visita todos possui distância mais que $OPT(x)$, uma contradição. Logo $|E_k| < k$. ■

Com isso podemos provar o teorema 2.2.

Prova. Pelo lema, a distância de i-ésima aresta em ordem não-crescente e no máximo $2OPT(x)/\sqrt{i}$. Logo temos para a distância da rota

$$\sum_{a \in C} d_a \leq 2OPT(x) \sum_{i \in [n]} 1/\sqrt{i} \leq 4OPT(x)\sqrt{n}$$

(porque $\sum_{i \in [n]} 1/\sqrt{i} \leq \int_0^n i^{-1/2} di = 2n^{1/2}$). ■

Observação 2.2

Os teoremas não quantificam a complexidade para encontrar o mínimo local. Chandra et al. (1999) ainda provaram que o número esperado de iterações sobre instâncias Euclidianas aleatórias em $[0,1]^2$ é $O(n^{10} \log n)$. Para $[0,1]^3$ isso se reduz para $O(n^6 \log n)$. Eles também provaram que no caso métrico existem instâncias com mínimos locais cujo valor desvia pelo menos um fator $1/4\sqrt{n}$ da otimalidade, i.e., o teorema assintoticamente é o melhor possível. ◊

Por final observamos que o PCV em geral não é resolúvel por busca local (em contraste com a programação linear e o método Simplex).

Teorema 2.3 (Papadimitriou e Steiglitz (1977))

Caso $P \neq NP$, não existe um algoritmo de busca local com complexidade polinomial por passo que é exato para o PCV.

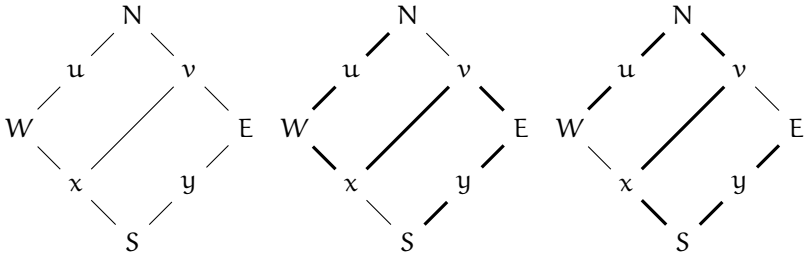
Considere primeiramente o problema

> CICLO HAMILTONIANO RESTRITO
>
> **Entrada** Um grafo não-direcionado $G = (V, A)$ e um caminho Hamiltoniano p em G.
>
> **Decisão** Existe um ciclo Hamiltoniano em G?

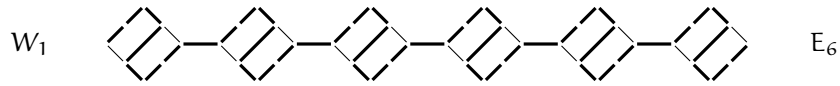
Lema 2.3

Ciclo Hamiltoniano restrito é NP-completo.

Prova. Por redução do problema "Ciclo Hamiltoniano". Considere o grafo "diamante" abaixo com quatro "entradas" norte (N), oeste (W), sul (S) e este (E). Entrando em N, W, S, E ele só pode ser atravessado por um ciclo Hamiltoniano em dois modos, um modo EW e outro modo NS, como mostrado do lado.



Para uma dado instância $G = (V, A)$ do problema do ciclo Hamiltoniano, podemos construir um grafo G' que possui um caminho Hamiltoniano como segue. Introduz um "diamante" d_v para cada vértice em $v \in V$ e chama os quatro entradas N_v, W_v, S_v, e E_v. Conecta os diamantes de oeste ao este linearmente, i.e. $(E_1, W_2), (E_2, W_3), \ldots, (E_{n-1}, W_n)$. Isso garante a existência de um caminho Hamiltoniano começando no oeste do primeiro vértice W_1 e terminado no este do último vértice E_n. Para representar a estrutura do grafo G, introduz para cada aresta $(u, v) \in A$ duas arestas (N_u, S_v) e (N_v, S_U) conectando os diamantes correspondentes a u e v de norte a sul. Caso G possui um ciclo Hamiltoniano, G' também, atravessando os diamantes sempre de modo WE de acordo com o ciclo. Caso G' possui um ciclo Hamiltoniano, ele usa somente os diamantes de modo NS. Caso contrário, o ciclo tem que seguir o modo WE até terminar num dos dois vértices W_1 e E_n. Logo G também possui um ciclo Hamiltoniano.



■

Prova.(do teorema 2.3) Por contradição. Caso existe tal busca local, podemos decidir em tempo polinomial se uma dada solução s é sub-ótima: é suficiente chamar $N(x, s)$. Mas o problema de decidir se uma solução s é sub-ótima é NP-completo, por redução de Ciclo Hamiltoniano restrito. O problema pertence a NP, porque uma solução ótima é um certificado curto da sub-otimalidade. Dado um grafo não-direcionado $G = (V, A)$ define uma instância do PCV com cidades V, e distâncias $d_a = 1$ caso $a \in A$, e $d_a = 2$ caso contrário. O ciclo

Hamiltoniano c fechando p possui distância total $(n-1)+2$. Agora G possui um ciclo Hamiltoniano sse o PCV possui uma solução de valor n sse c é subótimo. ■

♢

As analises de mínimos locais podem trazer informações relevantes sobre a qualidade da solução e sugerem caminhos para melhor mínimos locais. Isso é motivo do

Princípio de projeto 2.2 (Vizinhanças)
Encontra exemplos de mínimos locais e os compara com soluções ótimas. Investiga que tipo de modificação poderia melhorar um mínimo local.

2.2.1. Segue os vencedores

Segue os vencedores (ingl. go with the winners) (Aldous e Vazirani 1994) é uma estratégia que trabalha com múltiplas soluções. Cada solução percorre uma trajetória de uma busca local monótona. Caso uma das trajetórias termina num mínimo local, ela continua no ponto atual de uma das outras trajetórias que ainda não chegaram num mínimo local. A busca termina, caso todas trajetórias terminaram num mínimo local.

Algoritmo 2.3 (Segue os vencedores (SOV))
Entrada Solução inicial s, vizinhança N, distribuição P_s, o número de soluções k.

Saída Uma solução com valor no máximo $\varphi(s)$.

```
1  SV(s)=
2     s_i := s para i ∈ [k]
3     s* = s
4     repeat
5        seja L := {i ∈ [k] | B(s) = ∅} e L̄ := [k] \ L
6        atribui às soluções em L
7           uniformemente soluções em L̄
8        seleciona s'_i ∈ N̂(s_i) de acordo com P̂_{s_i}
9        s_i := s'_i
10       s* = min{s_*, s_1, ..., s_k}
11    until L = [k]
12    return s*
13 end
```

Na atribuição das linhas 6–7 cada solução em $\overline{L}$ é usada no máximo $\lceil|\overline{L}|/|L|\rceil$ vezes.
A motivação para SOV pode ser explicada no exemplo da árvore na figura 2.3. Seja d a variável aleatória da profundidade alcançada por uma partícula numa caminhada aleatória partindo da raiz em direção as folhas. Temos $P[d > k] = 2^{-k}$ (a profundidade da raiz é 0). Com n partículas independentes, seja $d^* = \max\{d_1, \ldots, d_n\}$. Logo

$$\begin{aligned} P[d^* > k] &= 1 - P[d^* \leq k] = 1 - \prod_{i\in[n]} P[d_i \leq k] \\ &= 1 - \prod_{i\in[n]} 1 - P[d_i > k] = 1 - \prod_{i\in[n]} 1 - 2^{-k} = 1 - (1-2^{-k})^n. \end{aligned}$$

Aplicando o lema A.4 obtemos

$$E[d^*] = \sum_{k\geq 0} P[d^* > k] = \sum_{k\geq 0} 1 - (1-2^{-k})^n \leq \sum_{k\geq 0} 1 - (1-2^{-k}n) = 2n$$

(a última estimativa segue pela desigualdade de Bernoulli A.1).
Seja agora d^S a variável aleatória do SOV com n partículas. Temos $P[d^S > k] = (1-2^{-n})^k$ e logo

$$E[d^S] = \sum_{k\geq 0} P[d^S > k] = \sum_{k\geq 0} (1-2^{-n})^k = 2^n.$$

Logo a profundidade esperada do SOV é exponencialmente maior que a profundidade de um número equivalente de explorações com uma partícula neste exemplo. De fato, temos:

Teorema 2.4 (Aldous e Vazirani (1994))
Para uma árvore com profundidade D, sejam V_i os vértices na profundidade i e seja $p(v)$ a probabilidade de visitar vértice v numa caminhada aleatória da raiz na direção das folhas para uma dada distribuição de probabilidade $p(u \mid v)$ entre os filhos u de cada vértice interno v. Define $\kappa = \max_{0\leq i<j\leq D} \kappa_{i,j}$ com

$$\kappa_{i,j} = P[d \geq i]/P[d \geq j]^2 \sum_{v\in V_i} p(v)P[d \geq j \mid v]^2.$$

Então, SOV com $B = \kappa D^{O(1)}$ partículas falha de chegar na profundidade D com probabilidade no máximo 1/4.

O valor κ é uma medida da dificuldade de superar os D níveis. No exemplo da figura 2.3 temos $\kappa = 2$ (para uma profundidade máxima fixa D).

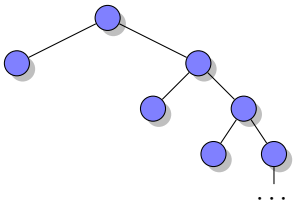


Figura 2.3.: Exemplo de uma árvore em que segue os vencedores é exponencialmente mais eficiente que uma estratégia de múltiplos inícios.

2.2.2. Complexidade

A solução ótima de um problema de otimização também é um mínimo local para qualquer vizinhança. Para problemas em PO podemos encontrar um mínimo global (e local) em tempo polinomial. Porém o exemplo do método Simplex mostra que mesmo em casos em que podemos encontrar um mínimo local em tempo polinomial, isso não precisa ser por uma busca local monótona. Logo, temos o problema de analisar a complexidade de uma das busca local, o problema de encontrar um mínimo local de qualquer forma, e o problema de encontrar o mínimo local que a busca local encontraria.
Para calcular um mínimo local por uma busca local monótona, claramente pelo menos a vizinhança tem que ser analisável em tempo polinomial. A classe de complexidade PLS captura essa ideia.

Definição 2.3 (Johnson et al. (1988))
Um problema de otimização Π com $\mathcal{P}$ polinomialmente limitada, junto com uma vizinhança N (escrito Π/N) pertence à classe de complexidade PLS caso existem algoritmos polinomiais I, V, N tal que

i) $I(x)$ produz uma solução (inicial);

ii) $V(x, s)$ decide se é uma solução válida da instância x, e caso sim, calcula $\varphi(x, s)$;

iii) $N(x, s)$ verifica se s é um mínimo local, e caso contrário produz uma solução vizinha $s' \in N(s)$ estritamente melhor, i.e. $\varphi(s') < \varphi(s)$.

Com isso podemos definir quatro problemas concretas.

Complexidade de uma busca local

Entrada Um problema em PLS com funções I, V, N fixas.

Problema Qual a complexidade pessimista em número de passos sobre todas soluções iniciais em função do tamanho do problema?

PROBLEMA DE BUSCA LOCAL

Entrada Um problema em PLS.

Problema Encontra um mínimo local.

Observações O mínimo local pode ser encontrado com qualquer algoritmo, não necessariamente por busca local.

PROBLEMA DE ENCONTRAR O MÍNIMO LOCAL PADRÃO

Entrada Um problema em PLS com funções I, V, N fixas.

Problema Encontra o mínimo local que a busca local definido por I, V e N encontraria?

Teorema 2.5
$\mathsf{FP} \subseteq \mathsf{PLS} \subseteq \mathsf{FNP}$.

Prova. Supõe que temos um problema em FP com algoritmo A. Então existe Π/N tal que os mínimos local correspondem com as soluções de uma instância: podemos escolher $S(x) = \{y \mid (x, y) \in \mathcal{P}\}$, $\varphi(x, s) = 0$ e $\mathsf{N}(x, s) = \{s\}$. O algoritmo I é o algoritmo A, o algoritmo V decide $(x, y) \in \mathcal{P}$ em tempo polinomial e o algoritmo N sempre retorna "falso".
Caso temos um problema $\Pi/\mathsf{N} \in \mathsf{PLS}$, então o problema de encontrar um mínimo local pertence a FNP: as soluções são limitadas polinomialmente, e podemos usar o algoritmo N para reconhecer soluções. ■
Logo, a questão $\mathsf{PLS} \subseteq \mathsf{FP}$ é "podemos encontrar mínimos locais em tempo polinomial?".
Para relacionar problemas de busca local serve a seguinte noção de redução.

Definição 2.4 (Redução PLS)
Uma problema de busca local Π_1/N_1 é PLS-redutível a um problema de busca local Π_2/N_2 caso existem algoritmo polinomiais S, T tal que:

- Podemos transformar instâncias de Π_1/N_1 para Π_2/N_2: Para $x_1 \in I_1$, $S(x_1) \in I_2$.
- Podemos transformar soluções de Π_2/N_2 para soluções de Π_1/N_1: Para $s_2 \in S(x_2)$, $T(s_2, x_1) \in S(x_1)$.

- Os mínimos locais correspondem: Para um mínimo local $s_2 \in S(x_2)$ de Π_2/N_2, $T(s_2, x_1)$ é um mínimo local de Π_1/N_1.

Com isso obtemos a noção normal de completude. Em particular as reduções são transitivas (ver exercício 2.2).

Definição 2.5 (PLS-completo)
Um problema Π/N em PLS é PLS-complete para todo problema em PLS é PLS-redutível a Π/N.

Considera o problema Circuit/1-flip: Dado um circuito booleano (sobre $\wedge, \vee, \neg$, por exemplo) com n entradas e m saídas encontra um mínimo local para a função objetivo que trata as saídas como número binário de m bits.

Teorema 2.6 (Completude de Circuit/1-flip)
Circuit/1-flip é PLS-completo.

Prova. Ver por exemplo Yannakakis (2003). ■

Teorema 2.7
Para k fixo PCV/k-exchange é PLS-completo.

Fato 2.1
Os problemas MaxCut/Flip a Graph-partitioning/Swap are PLS-complete. Para os problemas Graph-partitioning/Swap, TSP/k-opt e MaxCut/Flip a busca local precisa no caso pessimista um número exponencial de passos para encontrar um mínimo local. Para os mesmos problemas, o problema de encontrar um mínimo local específico é PSPACE-complete.

2.2.3. Notas

Uma boa introdução à busca local encontra-se em Kleinberg e Tardos (2005, cáp. 12) ou Papadimitriou e Steiglitz (1982, cáp. 10). A última referência tem mais material sobre a conexão entre busca local e a busca na vizinhança definida por um politopo. Michiels et al. (2007) apresentam aspectos teoricos da busca local. Em particular o cáp. 5 dessa referência apresenta mais detalhes sobre o PCV métrico e Euclidiano. Neumann e Wegener (2006) analisam mais profundamente o desempenho de uma busca local randomizada no problema da árvore geradora mínima. Um exemplo em que a busca local é melhor que outras abordagens é o problema métrico das k-medianas (ver por exemplo Korte e Vygen (2008, cáp. 22). Dimitriou e Impagliazzo (1996) propõem uma variante do algoritmo SOV que distribui as soluções de acordo com o número de vizinhos melhores. Yannakakis (2009) mostra conexões entre busca local e jogos, Knust (1997) entre busca local e problemas de escalonamento.

2.3. Buscas locais não-monótonas

Uma busca local não-monótona permite piorar a solução atual.

Algoritmo 2.4 (S-LocalSearch)
Entrada Solução inicial s, distribuição P_s

Saída Uma solução com valor no máximo $\varphi(s)$.

```
1  S–LocalSearch(s)=
2    s* := s
3    repeat
4      seleciona s' ∈ N̂(s) de acordo com P̂_s
5      if aceitável(s, s') then s := s'
6      if φ(s) < φ(s*) then s* := s
7    until critério de parada satisfeito
8    return s*
9  end
```

No que segue usaremos $\Delta(s, s') = \varphi(s') - \varphi(s)$. A tabela 2.1 mostra um resumo de estratégias de seleção e aceitação dos métodos discutidos abaixa.

2.3.1. Critérios de parada

Em buscas locais não-monótonas temos que definir um *critério de parada* (ingl. stopping criterion). Exemplos incluem um número máximo de iterações ou um tempo máximo. Ambos são usados frequentemente, por serem simples, e por permitirem comparações da qualidade obtida com os mesmos recursos por métodos diferentes. Porém, eles potencialmente gastem tempo demais em instâncias em que uma boa solução foi encontrada cedo na busca, e provavelmente gastem tempo de menos em instâncias maiores que foram consideradas na definição dos critérios: um bom método precisa ajustar a tempo investido em função do tamanho do problema.
Critérios de parada dinâmicos resolvem estes problemas. Exemplos são: (i) A solução encontrada possui um desvio relativo fixo de algum limite inferior do problema. Este método fornece inclusive uma garantia da qualidade da solução. (ii) Podemos determinar empiricamente, que a probabilidade de melhorar a solução incumbente é baixa. O critério mais simples desse tipo é parar caso o método não faz progresso por um número de iterações ou um tempo fixo. Em função do método critérios mais rigorosos são possíveis (por exemplo por métodos estatísticos em métodos de múltiplos inícios, ver cap. 3.2).

Tabela 2.1.: Estratégias de busca local.

Nome	Estratégia de seleção	Estratégia de aceitação
Aceitação por limite	Cam. aleatória	$\Delta(s, s') < W(t)$
Grande dilúvio	Cam. aleatória	$\varphi(s') < W(t)$
Recorde para recorde	Cam. aleatória	$\Delta(s^*, s') < W(t)$
Algoritmo demônio	Cam. aleatória	$\Delta(s, s') < W(t)$
Aceitação atrasada	Cam. aleatória	$\Delta(s', s_{-k}) < 0$
BLMR	De acordo com (2.2)	Com prob. 1.
Têmpera simulada	Cam. aleatória	Com prob. $\min\{e^{-\Delta(s,s')/T(t)}, 1\}$
Busca Tabu	Unif. em $N(s) \setminus L(t)$	Com prob. 1.

Exemplo 2.11 (Desvio relativo limitado)
O *limitante de Held-Karp* (ingl. Held-Karp bound) HK para o PCV é o valor do programa linear

$$\begin{aligned} \textbf{minimiza} \quad & \sum_{e \in E} c_e x_e & \\ \textbf{sujeito a} \quad & x(\delta(S)) \geq 2 & \text{para } \emptyset \neq S \neq V \\ & x(\delta(c)) = 2 & \text{para } v \in V \\ & 0 \leq x_e \leq 1 & \text{para } e \in E. \end{aligned}$$

e pode ser obtido eficientemente na prática. (Aqui δ é o conjunto de arestas na fronteira do conjunto S e x o valor total deles.) No caso métrico o valor de HK não é menos que 2/3 do valor ótimo (Wolsey 1980). Logo, parando com um valor menos que αHK, para um $\alpha > 3/2$ temos uma α-aproximação da solução ótima. ◊

2.3.2. Aceitação por limite e variantes

Entre os métodos não-monótonos mais simples estão estratégias de aceitação por limite. Eles aceitam uma solução pior, dado que o valor da solução não ultrapassa um certo limite. Eles foram introduzidos como variantes determinísticos da têmpera simulada. A definição concreta do limite difere entre as estratégias de *aceitação por limite* (ingl. threshold accepting) (Dueck e Scheuer 1990), o *grande dilúvio* (ingl. great deluge) (Dueck 1993), *viagem de recorde para recorde* (ing. record-to-record-travel), *aceitação atrasada* (ingl. late acceptance) Burke e Bykov 2012, e *algoritmo demônio* (ingl. demon algorithm (Creutz 1983).
A tabela 2.1 mostra as estratégias de forma resumida. Na tabela, $W(t)$ é um limite que varia com o tempo como segue:

Aceitação por limite $W(t+1) = W(t) - \delta$ caso o algoritmo não faz progresso.

Grande dilúvio $W(t+1) = W(t) - \delta$ em cada aceitação de um movimento. Dueck (1993) sugere que δ seja "um pouco menos que 1% do valor médio de $\Delta(s, W(t))$".

Recorde para recorde $W(t) = W$.

Algoritmo demônio Nesse tipo de algoritmo, o demônio é um *banqueiro*: $W(t+1) = W(t) - \Delta(s, s')$. Variantes incluem *demônios limitados* ($W(t+1) = \min\{W(t) - \Delta(s, s'), W_{\max}\}$), *com inflação* (a "conta" do demônio diminiu com o tempo), ou *com valor aleatória* ($W(t)$ representa a média de uma variável com distribuição Gaussiana e desvio padrão fixo).

Outras formas da variação do limite são possíveis, e de fato, a seleção dos $W(t)$ é um problema em aberto (Aarts e Lenstra 2003).

2.3.3. Buscas locais estocásticas

Em buscas estocásticas o critério de aceitação é probabilístico e geralmente tal que soluções de melhor valor possuam uma probabilidade maior de serem aceitos.

Busca local monótona randomizada (BLMR)

Uma das buscas locais estocásticas mais simples, a *busca local monótona randomizada* (ingl. randomised iterative improvement) seleciona com probabilidade p um vizinho arbitrário, e com $1-p$ um vizinho melhor, i.e.

$$P_s(s') = \begin{cases} p/|N(s)| + (1-p)/|B(s)| & \text{caso } s' \in B(s) \\ p/|N(s)| & \text{caso } s' \in N(s) \setminus B(s) \end{cases}. \tag{2.2}$$

A probabilidade de encontrar a solução ótima para uma vizinhança conectada com uma busca local monótona randomizada converge para 1 com um número de passos crescente (Hoos e Stützle 2004, p. 155).

Algoritmo de Metropolis

O *critério de aceitação de Metropolis* (Metropolis et al. 1953) é

$$P[\text{aceitar } s' \mid s] = \begin{cases} 1 & \text{caso } \Delta(s, s') < 0 \\ e^{-\Delta(s,s')/kT} & \text{caso contrário} \end{cases}. \tag{2.3}$$

(O critério foi introduzido para a simulação da evolução de um sólido para o equilíbrio térmico, e por isso inclui a constante de Boltzmann k. No contexto de otimização ela tipicamente é ignorada, i.e. $k = 1$.) Uma busca local estocástica com temperatura fixa é conhecida como *algoritmo de Metropolis*. Para um $T \to \infty$ o algoritmo se aproxima a uma caminhada aleatória, para $T \to 0$ a uma busca local monótona.

Têmpera simulada

A *têmpera simulada* (ingl. Simulated Annealing) foi proposto por Cerny (1985) e Kirkpatrick et al. (1983). Ela varia a temperatura do algoritmo de Metropolis de acordo com uma programação de resfriamento (ingl. cooling schedule). O motivo é que a temperatura ideal depende da escala da função objetivo e geralmente também da instância.
Um aspecto teoricamente interessante da têmpera simulada é que ela converge para a solução ótima para certos programações de resfriamento. Define a *profundidade* $d(s)$ de um mínimo local s como menor valor tal que uma solução de valor menor que $\varphi(s)$ é alcançável a partir de s via soluções de valor no máximo $\varphi(s) + d(s)$. Com isso temos

Teorema 2.8 (Hajek (1988))
Para uma constante Γ e $T(t) = \Gamma / \log(t+2)$ a têmpera simulada converge assintoticamente para uma solução ótima sse a vizinhança é conectada, simétrica, e $\Gamma \geq D$, sendo D a profundidade máxima de um mínimo local.

Uma heurística concreta usando têmpera simulada precisa definir uma *temperatura inicial*, o número de iterações com temperatura constante *ingl. temperature length*, uma programação de resfriamento, e um critério de parada.
A temperatura inicial e o número de iterações por temperatura dependem fortemente da instância e por isso devem ser calibrados dinamicamente. Para a temperatura inicial, uma técnica é gerar uma série de soluções aleatórias e definir a temperatura inicial tal que $T = \Delta(s_{\min}, s_{\max})$ em que $s_{\min}$ e $s_{\max}$ são as soluções de menor e maior valor encontradas. Uma outra técnica é incrementar uma temperatura baixa inicial, até uma percentagem desejada de movimentos (tipicamente $> 90\%$) é aceito.
O número de iterações por temperatura tipicamente deve ser proporcional ao tamanho da vizinhança para obter bons resultados (Johnson et al. 1989). Uma outra abordagem para garantir um progresso por temperatura, e manter ela constante até um número mínimo de movimentos foi aceito, mas não mais que um limite superior de iterações, para evitar um custo alto para temperaturas baixas.

A programação de resfriamento mais comum é *geométrica*, em que $T(t) = T_0\alpha_t$ com $\alpha \in (0,1)$. Um valor típico é $\alpha \in [0.8, 0.99]$. Johnson et al. (1989) concluem experimentalmente que não há razão para usar outras programações de resfriamento (como p.ex. linear, ou logarítmico).
Como critério de terminação podemos usar uma temperatura final, por exemplo. Um critério adaptativo, que detecta o "domínio" da busca local é ainda melhor. Johnson et al. (1989) propõem, por exemplo, usar uma percentagem mínima de movimentos que pioram: caso menos movimentos são aceitos em mais que um número fixo de níveis de temperatura, sem melhorar a melhor solução encontrada, o método termina. Como o método é estocástico, é indicado aplicar uma busca local depois.

Observação 2.3 (Johnson et al. (1989))
Experimentalmente, parece que

- A têmpera simulada precisa um tempo longo para obter resultados de boa qualidade.
- Tempo gasto no início e no final (domínio de caminhada aleatório e busca local) tipicamente é pouco efetivo.
- Uma execução mais longa da têmpera simulada tende a produzir melhores resultados que diversas repetições mais curtas. Isso provavelmente se aplica também para o "reheating".

◊

2.3.4. Otimização extremal

Otimização extremal (ingl. extremal optimization) (Boettcher e Percus 2003) supõe que uma solução s é representada por variáveis $(x_1, \ldots, x_n)$ (ver seção 1.2) e que cada variável contribui linearmente à função objetivo com um valor $\lambda_i(s)$, i.e. $\varphi(s) = \sum_{i\in[n]} \lambda_i(s)$. A vizinhança na busca local é restrita para vizinhos que alteram o valor uma determinada variável, a *variável extrema*. A probabilidade de uma variável ser a variável extrema é proporcional à sua contribuição $\lambda_i(x_i)$ na função objetivo.

Algoritmo 2.5 (EO)
Entrada Solução inicial s.

Saída Uma solução com valor no máximo $\varphi(s)$.

1 EO(s)=

```
2    s* := s
3    repeat
4      seja s = (x_1, ..., x_n) com λ_1(s) ≥ ··· ≥ λ_n(s)
5      seleciona i ∈ [n] com probabilidade ∝ i^{-τ}
6      seleciona s' ∈ N(s) tal que x_i muda o valor
7      s := s'
8      atualiza s*
9    until critério de parada satisfeito
10   return s*
```

Boettcher e Percus (2003) propõem $\tau = 1 + \Theta(1/\ln n)$.

2.3.5. Busca local guiada

A busca local guidada (ingl. guided local search) penaliza elementos indesejáveis na solução, similar a otimização extremal, mas por modificação da função objetivo. Supõe uma representação por conjuntos e uma função $\lambda_u(s)$ que define o custo do elemento $u \in U$. (Diferente da otimização extremal este custo não precisa entrar diretamente na função objetivo.) Além disso, para cada elemento $u \in U$, p_u é o número de vezes o elemento foi penalizado. A busca local guiada usa a função objetivo

$$\varphi'(s) = \varphi(s) + \sum_{u \in s} p_u.$$

Em cada mínimo local o método penaliza os elementos com uma *utilidade de penalização*

$$P(s,u) = \begin{cases} \lambda_u(s)/(1+p_i) & \text{caso } u \in s \\ 0 & \text{caso contrário} \end{cases}$$

máxima (i.e. aumenta o p_u correspondente por 1) e continua com a busca. Observe que a busca local guiada é independente do método para chegar num mínimo local.

2.3.6. Busca tabu

A ideia central da *busca tabu* é usar memoria adaptativa para guiar uma busca local. Na forma proposta inicialmente por Glover (1986) ela aplica a estratégia "melhor melhora" enquanto $B(s) \neq \emptyset$, e permite soluções piores caso contrário. Uma *memoria de curta duração* (ingl. short-term memory, ou recency-based

memory) serve para excluir soluções candidatas (declará-las "tabu") da vizinhança com o objetivo de evitar ciclagem. A busca tabu demonstrou a sua utilidade em várias aplicações, porém existe pouca fundamentação teórica: não existe prova de convergência para a otimalidade, por exemplo.
Uma busca tabu probabilística relaxa a estratégia "melhor melhoras" para uma busca por amostragem. Isso pode ser indicado em vizinhanças grandes e reduz a probabilidade de ciclagem. Além disso, existem resultados teóricos que mostram a convergência nesse caso (e.g. (Faigle e Schrader 1992)).
O algoritmo 2.6 mostra uma busca local estocástica com memoria genérica.

Algoritmo 2.6 (S-LocalSearchMemory)
Entrada Solução inicial s_0, distribuição P_s

Saída Uma solução com valor no máximo $\varphi(s)$.

```
1  S−LocalSearch(s)=
2     inicializa a memoria M
3     s* := s
4     repeat
5        seleciona s' ∈ N̂(s) de acordo com P̂_{s,M}
6        if aceitável(s', M) then s := s'
7        atualiza a memoria M
8        if φ(s) < φ(s*) then s* := s
9     until critério de parada satisfeito
10    return s*
11 end
```

A busca tabu básica define $P_{s,M}(s') = 1/|B^*(s)|$ para $s' \in B^*(s)$ com $B^*(s) = \{s' \in N(s) \setminus L(s,M) \mid \varphi(s') = \min_{s'' \in N(s) \setminus L(s,M)} \varphi(s'')\}$ e sempre aceita a nova solução s'. Neste caso a lista de soluções tabu $L(s,M)$ resulta (da parte da memoria de curta duração) de M.
A memoria de curta duração mais usada guarda atributos removidos ou inseridos em soluções e trata uma solução que inclui um atributo removido ou exclui um atributo inserido recentemente como "tabu". Na representação por conjuntos (ver cap. 1.2) sejam i_u e r_u o último tempo em que o elemento $u \in U$ foi inserido e removido da solução. Para uma *duração tabu* (ingl. tabu tenure) fixa d, a *regra tabu* define um vizinho s' de s tabu no tempo t caso

$$t \leq \max\{r_u + d \mid u \in s' \setminus s\} \tag{2.4}$$

$$t \leq \max\{i_u + d \mid u \in s \setminus s'\}. \tag{2.5}$$

Aqui a primeira restrição proíbe introduzir elementos removidos em menos

tempo que $\mathtt{d}$, e a segunda remover elementos introduzidos em menos tempo que $\mathtt{d}$. Uma boa duração tabu depende do tamanho da instância e um intervalo adequado $[\mathtt{d}_{\min}(\mathtt{n}), \mathtt{d}_{\max}(\mathtt{n})]$ e tem que ser determinado experimentalmente (Glover e Laguna 1997). Valores mais baixos tendem intensificar a busca, mas resultam em ciclagem no limite, e valores altos tendem a diversificar a busca, mas resultam numa qualidade reduzida no limite.

Observação 2.4 (Implementação memoria de curta duração)
Uma implementação de $\mathtt{r}$ e $\mathtt{u}$ com vetores na estratégia acima acima permite um teste tabu em tempo linear no tamanho da modificação $\mathtt{s} \oplus \mathtt{s}'$, que frequentemente é $\mathtt{O}(1)$. Caso $|\mathtt{U}|$ é grande demais, é preferível usar tabelas hash. ♢

A regra tabu básica permite diversas variações. Entre os mais comuns são

- Considerar um vizinho como tabu somente se ambas condições (2.4) e (2.5) são satisfeitas.
- Considerar somente atributos alterados: com $\mathtt{a}_\mathtt{u}$ o tempo da última alteração (inserção ou remoção), o critério tabu é simplificado para

$$\mathtt{t} \leq \max\{\mathtt{a}_\mathtt{u} + \mathtt{d} \mid \mathtt{u} \in \mathtt{s}' \oplus \mathtt{s}\}.$$

- Usar uma duração tabu diferente em (2.4) e (2.5): quanto mais a proibição de um atributo restringe a solução, quanto menor deve ser a duração tabu (Glover e Laguna 1997).
- Usar uma duração tabu dinâmica, por exemplo um valor aleatório em $[\mathtt{d}_{\min}(\mathtt{n}), \mathtt{d}_{\max}(\mathtt{n})]$ ou uma sequencia fixa (e.g. um múltiplo adequado do prefixo do *ruler function* $(1, 2, 1, 3, 1, 2, 1, 4, 1, 2, \ldots$, (*A001511*)); Galinier et al. (2011) é um exemplo de uma abordagem estado de arte que aplica isso.)
- Declarar diferentes aspectos de um problema tabu, ou usar mais que uma lista tabu.
- Tratar um tabu como penalidade: um atributo tabu $\mathtt{u}$ não é proibido, mas penalizado por $\mathtt{t} - (\mathtt{a}_\mathtt{u} + \mathtt{d})$.

Exemplo 2.12 (PCV)
Na representação do PCV por conjuntos usando 2-exchange arestas removidas ou inseridas se tornam tabu. Considerando critério (2.4) e (2.5) proíbe desfazer o 2-exchange por $\mathtt{d}$ iterações. Um exemplo de um aspecto diferente é declarar todas arestas incidentes com as cidades do último 2-exchange tabu. ♢

Uma consequência de uma memoria de curta duração é um *critério de aspiração* (ingl. aspiration criterion). A exclusão de atributos exclui não somente solução já visitadas, mas também pode excluir soluções ainda não visitadas, inclusive soluções com melhores características ou valores da função objetivo. Para contornar este problema, um critério de aspiração define exceções da regra tabu. Na forma mais simples ele permite aceitar um vizinho que melhora a solução incumbente. Um critério de aspiração pode também permitir escolher o vizinho "menos tabu" caso não existe vizinho não-tabu ("aspiration by default"). Esta condição pode servir alternativamente como critério de parada, além dos critérios genéricos (cap. 2.3.1).

Intensificação e diversificação Para melhorar a solução pode ser útil intensificar a busca perto de soluções de boa qualidade. Isso pode ser alcançado reduzindo o tamanho da lista tabu, fixando partes dos atributos para um determinado tempo, ou aplicando outras formas de buscas (e.g. um solver exato).

Em outras fases é necessário diversificar a busca, i.e. conduzi-la para novas soluções.

Memoria de longa duração Uma *memoria de longa duração* pode ser usada para guiar a busca mais efetivamente, e para intensicá- ou diversificá-la. A memoria pode guardar soluções de boa qualidade ou informações estatísticas. Mais comum para as últimas são frequências de pertinência em soluções (recentemente ou globalmente) e frequências de alteração de status de atributos. Por exemplo, para intensificar a busca podemos fixar elementos que recentemente pertenceram a soluções com alta frequência e aplicar um dos métodos acima ("restarting"). Para diversificar podemos incentivar incluir elementos que globalmente foram usados com baixa frequência, por exemplo incluindo um termo γf_u na função objetivo para um movimento que inclui elemento u, que já foi incluído com frequência f_u, onde γ é um parâmetro que depende do domínio função objetivo.

As observações sobre intensificação e diversificação e os diferentes tipos de memoria motivam

Princípio de projeto 2.3
Identifica os elementos de intensificação e diversificação da heurística. Procure encontrar um equilíbrio entre os dois princípios. Em particular, considere formas de memoria de longa duração para melhorar o desempenho da heurística.

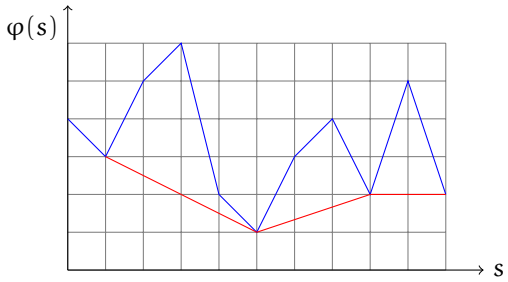


Figura 2.4.: Espaço de soluções (azul) e de mínimos locais (vermelho).

2.4. Buscas locais avançadas

2.4.1. Busca local iterada

A *busca local iterada* (ingl. iterated local search) pode ser vista como uma busca local no espaço de mínimos locais de um problema (ver figura 2.4).

Definição 2.6
O *basin de atração* $\mathcal{B}(s^*)$ associado a um mínimo local s^* e o conjunto de soluções s tal que uma dada busca local iniciada em s termina em s^*.

Logo, para passar de um mínimo local para outro, temos que alterar a solução atual suficientemente para obter uma solução nova que pertence a um basin de atração vizinho. Para isso, a busca local iterada *perturba* a solução atual e aplica a busca local na solução perturbada, para obter um outro mínimo local. A forma específica da perturbação define a vizinhança entre os mínimos locais e a probabilidade de transição. O critério de aceitação pode ser um dos critérios usados em uma busca não-monótona (e.g. o critério de aceitação de Metropolis).
Para perturbar o mínimo local atual podemos, por exemplo, caminhar aleatoriamente para um número de iterações, ou escolher um movimento aleatório numa vizinhança grande. Idealmente a perturbação é na ordem de grandeza do diâmetro do basin da solução atual: perturbações menores levam ao mesmo mínimo local, enquanto perturbações maiores se aproximam a uma caminhada aleatória no espaço de mínimos locais.

2.4.2. Busca local com vizinhança variável

Os métodos usando k vizinhanças $\mathcal{N}_1, \ldots, \mathcal{N}_k$ sempre voltam a usar a primeira vizinhança, caso um movimento melhora a solução atual. Caso contrário eles

passam para próxima vizinhança. Isso é o movimento básico:

Algoritmo 2.7 (Movimento)
Entrada Solução atual s, nova solução s', vizinhança atual k.

Saída Uma nova solução s e uma nova vizinhança k.

```
1  Movimento(s,s',k) :=
2     if φ(s') < φ(s) then
3        s := s'
4        k := 1
5     else
6        k := k + 1
7     end if
8     return (s,k)
```

Com isso podemos definir uma estratégia simples, chamada *Variable Neighborhood Descent* (VND).

Algoritmo 2.8 (VND)
Entrada Solução inicial s, conjunto de vizinhanças $\mathcal{N}_i$, $i \in [m]$.

Saída Uma solução com valor no máximo $\varphi(s)$.

```
1  rVNS(s,{N_i})=
2     k := 1
3     // até chegar num mínimo local
4     // para todas vizinhanças
5     while k ≤ m
6        encontra o melhor vizinho s' em N_k(s)
7        (s,k) := Movimento(s,s',k)
8     end while
9     return s
```

Uma versão randomizada é o *reduced variable neighborhood search.*

Algoritmo 2.9 (rVNS)
Entrada Solução inicial s, conjunto de vizinhanças $\mathcal{N}_i$, $i \in [m]$.

Saída Uma solução com valor no máximo $\varphi(s)$.

```
1  VND(s,{N_i})=
2     until critério de parada satisfeito
```

```
3       k := 1
4       while k ≤ m do
5         { shake }
6         seleciona vizinho aleatório s′ em N_k(s)
7         (s, k) := Movimento(s, s′, k)
8       end while
9     end until
10    return s
```

Uma combinação do rVNS com uma busca local é o *Variable Neighborhood Search* (VNS) básico.

Algoritmo 2.10 (VNS)
Entrada Solução inicial s, um conjunto de vizinhanças $\mathcal{N}_i$, $i \in [m]$.

Saída Uma solução com valor no máximo $\varphi(s)$.

```
1  VNS(s,{N_i})=
2    until critério de parada satisfeito
3      k := 1
4      while k ≤ m do
5        { shake }
6        seleciona vizinho aleatório s′ em N_k(s)
7        s″ := BuscaLocal(s′)
8        (s, k) := Movimento(s, s″, k)
9    end until
10   return s
```

Observação 2.5
A busca local em VNS pode usar uma vizinhança diferente das vizinhanças que perturbam a solução atual. Também é possível usar o VND no lugar da busca local. ◊

2.4.3. Busca local em vizinhanças grandes

Uma vizinhança é considerada *massiva* (ingl. very large scale) caso o número de vizinhos cresce exponencialmente com o tamanho da instância (Pisinger e Ropke 2010). Uma vizinhança massiva tem uma vantagem caso o custo maior de selecionar um vizinho é compensado pela qualidade das soluções. Em particular, isso é possível caso a vizinhança pode ser analisada em tempo

polinomial apesar do seu tamanho exponencial, e.g. por resolver um problema de caminhos mais curtos, fluxo máximo ou emparelhamento.

2.4.4. Detecção de estagnação genérica

Watson et al. (2006) propõem um mecanismo explicito e genérico para detecção de estagnação. Supõe que temos uma heurística H arbitrária, e seja $N_H(s)$ a próxima solução visitada por H dado a solução atual s. O CMF (Core methaheuristics framework) adiciona a essa heurística uma detecção explicita de estagnação.

Algoritmo 2.11 (CMF)
Entrada Uma instância de um problema, uma solução inicial s, uma distância mínima d_{min}, distâncias L_0 e Δ_L e um número de iterações t_{test}.

Saída A melhor solução encontrada.

```
1  CMF(s) :=
2    s_t := s
3    cada t_test iterações:
4      if d(s, s_t) < d_min then
5        if escaping then
6          L := L + Δ_L
7        else
8          L := L_0
9        s_t := s
10       s := randomWalk(s, L)
11       escaping := true
12     else
13       s_t := s
14       escaping := false
15     end if
16   s := N_H(s)
17 end
```

2.4.5. Notas

O livro de Hoos e Stützle (2004) é uma excelente referência para área de busca local estocástica. Os artigos Dueck e Scheuer (1990) e Dueck (1993) que propõem aceitação por limite, o grande dilúvio e viagem de recorde para

recorde são bem acessíveis. Talbi (2009) apresenta um bom resumo desses métodos que inclui o algoritmo demônio. A referência definitiva para a busca tabu ainda é o livro de Glover e Laguna (1997), uma boa introdução é Hertz et al. (2003).

2.5. Exercícios

Exercício 2.1
A vizinhança 2-flip para o k-SAT é simétrico? Fracamente otimamente conectada? Exata? E a vizinhança k-flip para $k > 2$?

Exercício 2.2
Mostra que reduções PLS são transitivas.

3. Busca por construção de soluções

3.1. Construção simples

3.1.1. Algoritmos gulosos

Definição 3.1 (Sistemas de conjuntos)
Um *sistema de conjuntos* é um par $(U, \mathcal{V})$ de um universo U de elementos e uma coleção $\mathcal{V}$ de subconjuntos de U. Caso para cada $S \in \mathcal{V}$ existe um $u \in U$ tal que $S \setminus \{u\} \in \mathcal{V}$ o sistema de conjuntos é *acessível*. Caso $\mathcal{V}$ é fechado sobre inclusão (i.e. caso $S' \subseteq S$ para um $S \in \mathcal{V}$ então $S' \in \mathcal{V}$) o sistema é *independente* e o seus elementos se chamam *conjuntos independentes*.

Definição 3.2 (Matroides e greedoides)
Um sistema de conjuntos satisfaz a *propriedade de troca*, caso para todos $S, T \in \mathcal{V}$ com $|S| > |T|$ existe um $u \in S \setminus T$ tal que $T \cup \{u\} \in \mathcal{V}$. Um *greedoide* é um sistema de conjuntos acessível que satisfaz a propriedade de troca. Um *matroide* é um sistema de conjuntos independente que satisfaz a propriedade de troca.

Definição 3.3 (Problema de otimização de um sistema de conjuntos)
Para um sistema de conjuntos $(U, \mathcal{V})$ com pesos $w_u \in \mathbb{R}_+$ para $u \in U$, o *problema correspondente* de otimização é encontrar um subconjunto independente de maior peso total.

Observação 3.1
Na prática o conjunto $\mathcal{V}$ é especificado por um algoritmo que decide, para cada $S \subseteq U$ se $S \in \mathcal{V}$. ◊

Exemplo 3.1
Muitos problemas de otimização podem ser formulados como sistemas de conjuntos, por exemplo o PCV (com arestas U, e $\mathcal{V}$ subconjuntos de circuitos Hamiltonianos), o problema do conjunto máximo independente (com vértices U e $\mathcal{V}$ os conjuntos independentes do grafo), o problema do caminho s-t mais curto (com arestas U e $\mathcal{V}$ subconjuntos de caminhos s-t), ou o problema da mochila (com itens U, e $\mathcal{V}$ os subconjuntos de itens que cabem na mochila). ◊

Um algoritmo guloso constrói iterativamente uma solução válida de um sistema de conjuntos acessível.

Algoritmo 3.1 (Algoritmo guloso)
Entrada Um sistema de conjuntos $(U, \mathcal{V})$.

Saída Uma solução $S \in \mathcal{V}$.

```
1  Guluso()=
2    S := ∅
3    while U ≠ ∅ do
4      seleciona u ∈ U com w_u maximal
5      U := U \ {u}
6      if S ∪ {u} ∈ V then
7        S := S ∪ {u}
8      end if
9    end while
10   return S
11 end
```

Teorema 3.1 (Edmonds-Rado)
O algoritmo guloso resolve o problema correspondente do sistema de conjuntos independente $S = (U, \mathcal{V})$ se e somente se S é um matroide.

Prova. Supõe S é um matroide. Pela propriedade de troca, todos conjuntos independentes maximais possuem a mesma cardinalidade. Supõe que o algoritmo guloso produz uma solução $S = \{s_1, \ldots, s_n\}$, mas a solução ótima $S^* = \{s'_1, \ldots, s'_n\}$ satisfaz $w(S) < w(S^*)$. Sem perda de generalidade $w_{s_i} \geq w_{s_{i+1}}$ e $w_{s'_i} \geq w_{s'_{i+1}}$ para $1 \leq i < n$. Provaremos por indução que (*) $w_{s_i} \geq w_{s'_i}$, uma contradição com $w(S) < w(S^*)$. Para $i = 1$ (*) é correto pela escolha do algoritmo guloso. Para um $i > 1$ supõe $w_{s_i} < w_{s'_i}$. Pela propriedade de troca existe um elemento de $u \in \{s'_1, \ldots, s'_i\} \setminus \{s_1, \ldots, s_{i-1}\}$ tal que $\{s_1, \ldots, s_{i-1}, u\} \in \mathcal{V}$. Mas $w_{s_i} < w_{s'_i} \leq w_u$, uma contradição com a escolha do algoritmo guloso.
De modo oposto, supõe o algoritmo guloso resolve o problema correspondente de otimização (para pesos arbitrários), mas a propriedade de troca é inválida. Logo existem conjuntos $S, T \in \mathcal{V}$, tal que $|S| = |T| + 1$ mas para nenhum $u \in S \setminus T$ temos $T \cup \{u\} \in \mathcal{V}$. Define

$$w_u = \begin{cases} |T| + 2 & \text{para } u \in T \\ |T| + 1 & \text{para } u \in S \setminus T \\ 0 & \text{caso contrário} \end{cases}.$$

Para essa instância o algoritmo guloso começa escolher todos elementos de T. Depois ele não consegue melhorar o peso total, porque um elemento em $S \setminus T$ não pode ser adicionado, e os restantes elementos possuem peso 0. Logo o valor da solução gulosa é $w(T) = |T|(|T|+2) < (|T|+1)^2 \leq w(S)$, em contradição com o fato que o algoritmo guloso resolve o problema otimamente. ■

Obtemos uma generalização similar com a busca local selecionando o próximo elemento de acordo com uma distribuição de probabilidade P sobre o universo U. Essa distribuição pode ser *adaptativa*, i.e. ela depende dos elementos selecionados anteriormente.

Algoritmo 3.2 (Algoritmo guloso generalizado)
Entrada Um sistema de conjuntos $(U, \mathcal{V})$.

Saída Uma solução $S \in \mathcal{V}$.

```
1  Guluso-Generalizado()=
2    S := ∅
3    while U ≠ ∅ do
4      seleciona u ∈ U de acordo com P
5      U := U \ {u}
6      if S ∪ {u} ∈ V then
7        S := S ∪ {u}
8      end if
9    end while
10   return S
11 end
```

Seja $u^* = \text{argmax}_u\{w(u)|u \in U\}$ e $B(U) = \{u \in U \mid w_u = w_{u^*}\}$. A estratégia gulosa corresponde com $P(u) = 1/|B(U)|$ para $u \in B(u)$. Um algoritmo semi-guloso relaxa este critério. Duas estratégias comuns são:

Guloso-k Seja $U = \{u_1, \ldots, u_n\}$ com $w_i \geq w_{i+1}$. Seleciona $S = \{u_1, \ldots, u_{\min\{k,n\}}\}$ e define $P(u) = 1/|S|$ para $u \in S$. Essa estratégia seleciona um dos k melhores elementos.

Guloso-α Seja $U = \{u_1, \ldots, u_n\}$ com $w_i \geq w_{i+1}$. Para um $0 < \alpha \leq 1$, seleciona $S = \{u_i \mid w_i \geq \alpha w_n + (1-\alpha)w_1\}$ e define $P(u) = 1/|S|$ para $u \in S$. Essa estratégia seleciona um entre os $\alpha\%$ melhores elementos.
Entre distribuições de probabilidade alternativas para o guloso-α temos abordagens que usam o rank r do elemento para definir um peso w_r, e selecionam o elemento com rank r com probabilidade $w_r / \sum w_r$. Exemplos são

- pesos polinomiais $w_r = r^{-\tau}$ (ver 2.3.4 para uma aplicação na otimização extremal);
- pesos lineares $w_e = 1/r$ ou $w_e = n - r$;
- pesos logarítmicos $w_e = 1/\log r + 1$; ou
- pesos exponenciais $w_e = e^{-r}$ (Bresina 1996).

Exemplo 3.2 (Construção gulosa para o PCV)
Exemplos de construções gulosas para o PCV são

- *vizinho mais próximo*: escolhe uma cidade inicial aleatória, e visita sempre a cidade mais próxima não visitada ainda, até fechar o ciclo;
- *algoritmo guloso*: no matroide com U todos arcos e V subconjuntos de arcos de ciclos Hamiltonianos, como acima;
- o *algoritmo de Clarke-Wright*: define uma cidade aleatória como centro e forma "pseudo-rotas" (2-ciclos) do centro para todos outras cidades. Ranqueia todos pares de cidades diferente do centro pela redução de custos ("savings") obtido passando diretamente de uma cidade para outra, não visitando o centro. Processa os pares nessa ordem, aplicando cada redução que mantém uma coleção de pseudo-rotas, até a coleção é reduzida para um único ciclo.
- o *algoritmo de Cristofides* para instâncias métricas: junta uma árvore geradora mínima das cidades com um emparelhamento perfeito de custo mínimo entre os vértices de grau impar da árvore, encontre um caminho Euleriano nesse grafo, e torná-lo um ciclo pulando cidades repetidas.

◊

3.1.2. Algoritmos de prioridade

Supõe uma representação de uma solução por variáveis. Uma solução parcial é um atribuição com *variáveis livres*, i.e. variáveis que ainda não receberam valores. *Algoritmos de prioridade* processam as variáveis em I em alguma ordem definida por uma *função de ordenamento* o que retorna um sequencia das variáveis livres. A variável atual recebe um valor em V de acordo com uma *função de mapeamento* f. Caso o depende somente da instância obtemos um *algoritmo de prioridade fixa*; caso a ordem depende também da atual solução parcial obtemos um *algoritmo de prioridade adaptativa*.

Algoritmo 3.3 (Algoritmo de prioridade)
Entrada Uma instância $I \subseteq U$, uma função de ordenamento o e uma função de mapeamento f.

Saída Uma solução S, i.e. um atribuição de valores em V aos elementos em I.

```
1  Prioridade()=
2    S := ∅
3    while I ≠ ∅ do
4      seja o(I,S) = (x_1,...,x_k)
5      S := S ∪ {x_1 ↦ f(S,x_1)}
6      I := I \ {x_1}
7    end while
8    return S
```

Observação 3.2
Um algoritmo de prioridade pode ser relaxado, da mesma forma que algoritmos gulosos, por selecionar a nova variável a ser fixada entre as $\alpha\%$ ou as k variáveis de maior prioridade. ◊

Exemplo 3.3 (Coloração de grafos)
Com a representação do exemplo 1.3 obtemos um algoritmo de prioridade fixa ordenando os vértices por grau não-crescente e usando uma função de mapeamento que atribui a menor cor livre ao vértice atual. Obtemos uma variante adaptativa ordenando os vértices ainda não coloridos por grau não-crescente com respeito a outros vértices não coloridos, com a mesma função de mapeamento. ◊

Exemplo 3.4 (Empacotamento bidimensional)
No problema de *empacotamento bidimensional* (ingl. 2D strip packing) temos n caixas de dimensões $l_i \times c_i$. O objetivo é empacotar as caixas numa faixa de largura L sem sobreposição, paralelo com os eixos, e sem rotacioná-los, tal que o comprimento total ocupado é minimizado. Um algoritmo de prioridade ordena as caixas por altura, largura, circunferência, ou área não-crescente, e aloca a caixa atual na posição mais para baixo e mais para esquerda possível ("bottom left heuristic"). ◊

3.1.3. Busca por raio

A *busca por raio* (ingl. beam search) mantém k soluções parciais (k é chamada a *largura do raio* (ingl. beam width)). Em cada passo uma solução parcial é

estendida para k' soluções parciais diferentes, e entre as kk' soluções novas, uma função de ranqueamento seleciona as k melhores. A função tipicamente fornece um limite inferior para as soluções completas que podem ser obtidas a partir da solução parcial atual.
Uma busca por raio pode ser entendida como uma busca por largura truncada ou ainda como versão construtiva do algoritmo SOV na busca. O modelo mais simples para definir a busca por raio é numa árvore de soluções parciais, com a solução vazia na raiz. Cada solução s possui uma série $F(s)$ de extensões possíveis (filhos na árvore), que são escolhidos com distribuição de probabilidade P_s. Seja $p(s)$ o pai de s na árvore.

Algoritmo 3.4 (Busca por raio)
Entrada Uma instância de um problema.

Saída Uma solução s, caso for encontrada.

```
1  BeamSearch(k,k′):=
2    B := {∅}
3    while B ≠ ∅ do
4      repete |B|k′ vezes
5        seja F := ∪_{s∈B} F(s)
6        B := ∅
7        seleciona f ∈ F com prob. P_{p(s)}(f)/∑_{f∈F} P_{p(f)}(f)
8        se f é sol. completa: atualiza o incumbente s*
9        se f é sol. parcial: B := B ∪ {f}
10       { alguns autores: F := F \ {f} }
11     end
12     seleciona as melhores soluções em B
13       (no máximo k)
14   end while
15   return s* { eventualmente não encontrado }
```

Observação 3.3
Uma busca por raio BeamSearch(1,1) é equivalente ao algoritmo guloso generalizado. ◊

3.2. Construção repetida independente

A estratégia de *múltiplos inícios* (ingl. multi-start) procura encontrar soluções melhores por construção repetida. No caso mais simples, cada repetição é independente da outra e o algoritmo retorna a melhor solução encontrada. Essa

estratégia pode ser usada com qualquer construção aleatória, por exemplo com os algoritmos Guloso-k e Guloso-α da seção anterior. Usando o algoritmo Guloso-α com $\alpha = 1$ obtemos uma construção totalmente aleatória. Múltiplos inícios também é uma estratégia simples de diversificação para outras heurísticas.

3.2.1. GRASP

A forma mais simples de melhorar uma construção repetida independente é aplicar uma busca local monótona às soluções construídas. Este método foi proposto com o nome GRASP (Greedy randomized adaptive search procedure) por Feo e Resende (1989).
Variantes básicas do GRASP incluem métodos que escolham $\alpha \in \{\alpha_1, \ldots, \alpha_k\}$ de acordo com alguma distribuição de probabilidade (a distribuição uniforme frequentemente é uma primeira escolha razoável), e *GRASP reativo* (ingl. reactive GRASP) que começa com uma distribuição uniforme e periodicamente adapta as prioridades de acordo com

$$P(\alpha_i) = q_i / \sum_{j \in [k]} q_j$$

com $q_i = \varphi(s^*)/\overline{\varphi_i}$ para incumbente s^* e com $\overline{\varphi_i}$ o valor médio encontrado usando α_i (para um problema de minimização).
O *GRASP evolucionário* (ingl. evolutionary GRASP), uma variante que usa uma outra forma memória de longa duração é discutida na seção 4.4.

3.2.2. Bubble search randomizada

Bubble search (Lesh e Mitzenmacher 2006) generaliza algoritmos de prioridade. Considera primeiramente um algoritmo de prioridade fixa. Para melhorá-lo, podemos consideras todas permutações das variáveis I na alocação. O Bubble search faz isso em ordem de distância Kendall-tau crescente da permutação base $o(S)$. A distância Kendall-tau mede o número de inversões entre duas permutações π e ρ de $[n]$, i.e.

$$d(\pi, \rho) = \sum_{1 \le i < j \le n} [\pi(i) < \pi(j) \text{ and } \rho(i) > \rho(j)] + [\pi(i) > \pi(j) \text{ and } \rho(i) < \rho(j)].$$

(A distância Kendall-tau é também conhecida por *distância de Bubble sort*.)
Bubble search randomizada gera uma permutação de distância d com probabilidade proporcional com $(1-p)^d$ para um parâmetro $p \in (0, 1)$.

Observação 3.4 (Geração de permutações no Bubble search)
Uma permutação de acordo com a probabilidade acima pode ser selecionado considerando os elementos ciclicamente na ordem $\mathsf{o}(\mathrm{I})$. Inicia com uma lista em ordem $\mathsf{o}(\mathrm{I})$. Começando com o primeiro elemento, visite os elementos da lista ciclicamente. Selecionando o item atual com probabilidade p, caso contrário continua. Ao selecionar um item, remove-o da lista e repete o processo na lista reduzida, até ela é vazia. A ordem da seleção dos itens define a permutação gerada. ◊

O processo da observação acima pode ser aplicado também em algoritmos de prioridade adaptativa considerando os elementos ciclicamente na ordem $\mathsf{o}(\mathrm{I}, \mathrm{S})$. (Observe que nesse caso não existe uma relação simples da ordem resultante com a distância Kendall-tau.)

3.3. Construção repetida dependente

Uma construção repetida dependente usa informações das iterações anteriores para melhorar a construção em iterações subsequentes. Um exemplo simples é o *Bubble search com reposição* (ingl. Bubble search with replacement): a ordem base é sempre a ordem em que o incumbente foi construído.

3.3.1. Iterated greedy algorithm

Algoritmos gulosos iterados foram introduzidos por Ruiz e Stützle (2006). Depois da primeira construção, o algoritmo repetidamente destrói parte da solução atual, e reconstrói-a gulosamente. A forma mais simples da destruição é remover d elementos na representação por conjuntos, ou resetar d variáveis na representação por variáveis e aplicar um algoritmo guloso, respectivamente um algoritmo prioridade a partir da solução parcial resultante para obter uma nova solução completa.
Um algoritmo guloso iterado é o análogo de uma busca local iterada. Aplicando uma busca local em cada iteração, um algoritmo guloso iterado vira uma busca local iterada, na qual a perturbação é realizada por destruição e reconstrução via um algoritmo guloso.

3.3.2. Squeaky wheel optimization

A otimização da *roda que chia* (ingl. squeaky wheel optimization), introduzida por Joslin e Clements (1999), prioriza na construção elementos que aumentam a função objetivo ("the squeaky wheel gets the grease"). O modelo mais simples para explicar isso é como modificação de um algoritmo de prioridade

cuja função de ordenamento usa pesos w_i para $i \in I$ e produz $o(I, S) = (x_1, \ldots, x_k)$ caso $w_1 \geq \cdots \geq w_k$. Supõe que as variáveis que aumentaram a função objetivo na última construção recebem ainda "penalidades" p_i para $i \in I$. A função de ordenamento $o(I, S) = (x_1, \ldots, x_k)$ tal que $w_1 + p_1 \geq \cdots \geq w_k + p_k$ considera além da ordem base as penalidades. A otimização da roda que chia corresponde com a otimização extremal e a busca local guidada que forçam alterar ou penalizam elementos que aumentam a função objetivo.

Exemplo 3.5
(Continua o exemplo 3.3.) Na coloração de grafos podemos penalizar vértices que usam cores $\geq n$, caso o incumbente tem n cores. ◊

3.3.3. Otimização por colônias de formigas

Algumas espécies de formigas conseguem encontrar caminhos curtos para objetos interessantes comunicando por feromônio deixado nas trilhas. O feromônio é uma forma de memoria de longa duração guiando as formigas. Otimização por colônias de formigas (ingl. ant colony optimization, ACO) (Dorigo et al. 1996) aplica essa ideia na otimização.
De forma mais abstrata, ACO realiza uma construção repetida dependente, com probabilidades de transição dinâmicas, que dependem das iterações anteriores. Concretamente, na representação de variáveis, ACO associa dois valores τ_{iv} e η_{iv} com uma variável $i \in I$ que recebe um valor $v \in V$. O valor τ_{iv} representa a componente dinâmica (o feromônio), e o valor η_{iv} a componente estática da preferência de atribuir o valor v à variável i. Uma fase do ACO constrói soluções $S_1, \ldots, S_m$ de forma independente. Uma construção repetidamente atribui um valor à próxima variável x_1 numa ordem fixa ou dinâmica $o(I, S) = (x_1, \ldots, x_k)$, igual a um algoritmo de prioridade, com probabilidade

$$P(x_1 = v \mid S) \propto \tau_{iv}^{\alpha} \eta_{iv}^{\beta}, \tag{3.1}$$

sendo α e β parâmetros que balanceiam o efeito entre preferência dinâmica e estática. (Logo, para $\alpha = 0$ obtemos um algoritmo guloso randomizado.) ACO atualiza no fim de cada fase os feromônios por

$$\tau_{iv} = (1 - \rho)\tau_{iv} + \sum_{S \in U \mid \{i \mapsto v\} \in S} g(S).$$

O primeiro termo diminui o feromônio com o tempo ("evaporação"), o segundo termo aumenta o feromônio de acordo com uma função de avaliação $g(S)$ das soluções S que atribuem v a i. As soluções S fazem parte de um conjunto

U de soluções candidatas. Os candidatos tipicamente incluem $S_1, \ldots, S_m$ e soluções elites (p.ex. o incumbente S^*). A função $g(S)$ cresce com a qualidade da solução. Concretamente, no exemplo do PCV:

- Sistema de formigas (ingl. ant system): $U = \{S_1, \ldots, S_m\}$, $\eta_{iv} = 1/d_{iv}$, $g(S) = 1/d(S)$.
- Sistema de formigas elitista: $U = \{S_1, \ldots, S_m, S^*\}$, $\eta_{iv} = 1/d_{iv}$,

$$g(S) = \begin{cases} 1/d(S) & \text{para } S_1, \ldots, S_m\} \\ e/d(S) & \text{para } S^* \end{cases}$$

- Sistema de formigas com ranqueamento: um sistema de formigas elitista com $U = \{S_1, \ldots, S_k, S^*\}$, sendo $S_1, \ldots, S_k$ os $k \leq m$ melhores soluções da última fase.
- Sistema de formigas com limites (ingl. min/max ant system): $U = \{S^*\}$ ou $U = \{S_1\}$ com S_1 a melhor solução da última fase ("elitismo forte") com limites $\tau_{min} \leq \tau_{iv} \leq \tau_{max}$, e $\tau_{iv} = \tau_{max}$ inicialmente.
- Sistema de colônia de formigas (ingl. ant colony system): elitismo forte com seleção "pseudo randômica proporcional": com probabilidade q seleciona a variável com $P(x_1 = v|S)$ máximo, senão de acordo com (3.1). O sistema também diversifica a construção reduzindo a quantidade de feromônio em atribuições selecionadas na fase atual.

3.4. Exercícios

Exercício 3.1
Quais sistemas de conjuntos do exemplo 3.1 são acessíveis? Independentes? Quais satisfazem a propriedade de troca?

4. Busca por recombinação de soluções

A recombinação de soluções procura misturar componentes da duas ou mais soluções para produzir uma ou mais novas soluções combinadas. Para algumas recombinações é conveniente ter uma noção de distância entre soluções. Para as nossas representações padrão de conjuntos e variáveis, usaremos as distâncias $d(s, s') = |s \oplus s'|$ e $d(s, s') = \sum_{i \in I}[s_i \neq s'_i]$, respectivamente. Em função do problema e sua representação outras distâncias podem ser adequadas. Tipicamente a representação de variáveis é mais conveniente para formular a recombinação de soluções.

Exemplos de recombinações simples na representação por variáveis de soluções $c = C(s_1, \ldots, s_n)$ são:

Recombinação randomizada Escolhe $c_i = s_{ki}$ com probabilidade p_k. Para $p_k = 1/n$ obtemos uma *recombinação uniforme*. Uma recombinação não-uniforme comum é escolher $p_k \propto \varphi(s_k)$. No contexto de algoritmos genéticos o caso $n = 2$, $V = \{0, 1\}$, $p = 1/2$ é chamada *crossover uniforme*] (Ackley 1987). Outro exemplo é definir $p_k \propto |\{s_{ki} \mid k \in [n]\}|$ na seleção da componente i. Caso a função objetivo é linear nas variáveis, i.e. $\varphi(s_k) = \sum_{i \in I} \varphi(s_{ki})$, um critério melhor pode ser uma seleção com probabilidade $p_k i \propto \varphi(s_{ki})$ para cada componente.

Recombinação por mediano Supondo que V possui uma ordem, escolhe $c_i = \langle s_{1i} \cdots s_{ni} \rangle$ com mediano $\langle \cdot \rangle$. Para n impar e $V = \{0, 1\}$ isso é uma *recombinação maioritária*.

Recombinação linear Supondo que $V = \mathbb{R}$, seleciona $c_i = \sum_{k \in [n]} \lambda_k s_{ik}$ com $\sum_{k \in [n]} \lambda_k = 1$. Para $\lambda_k \geq 0$ obtemos uma *recombinação convexa*.

Recombinação particionada Uma recombinação randomizada aplicada numa partição $\mathcal{S}$ de $[n]$. Para cada parte seleciona uma solução s_i com probabilidade p_i e atribui os valores de toda parte à solução combinada. Um subcaso importante são partições contínuas (i.e. cada parte $p \in \mathcal{S}$ satisfaz $p = [a, b]$ para $a < b$, $a, b \in [n]$.) Para uma partição contínua aleatória com $|\mathcal{S}| = 2$ obtemos o *recombinação em um ponto* (ingl. one-point crossover), caso $|\mathcal{S}| = k$ uma *recombinação em* k *pontos*.

Recombinação para permutações A recombinação tem que satisfazer as restrições do problema. Um caso frequente e por isso importante são permutações, com $I = V = [n]$. Exemplos de estratégias para recombinar permutações são:

Recombinação irrestrita na tabela de inversões Aplica uma das recombinações acima na tabela de inversões.

Recombinação PMX Para permutações $\pi = \pi_1\pi_2\ldots\pi_n$ e $\rho = \rho_1\rho_2\ldots\rho_n$ define $\sigma = \mathrm{PMX}(\pi, \rho)$ como segue (Goldberg e Lingle 1985):

1) Seleciona um intervalo aleatório $I = [a, b] \subseteq [n]$. Para uma permutação π, seja $\pi_I = \{\pi_i \mid i \in I\}$.
2) Define um mapeamento $m : \pi_I \to \rho_I : \pi_i \mapsto \rho_i$.
3) Define um mapeamento $m^* : \pi_I \to \rho_I : m^k(\pi_i)$, com k o menor expoente tal que $m^k(\pi_i) \notin \pi_I$. O mapeamento m^* itera m até o elemento não pertence a π_I.
4) Finalmente define

$$\sigma_i = \begin{cases} \pi_i & i \in I \\ \rho_i & \rho_i \notin \pi_I \\ m^*(\rho_i) & \rho_i \in \pi_I \end{cases}.$$

Exemplo 4.1 (Recombinação PMX)
Seja $\pi = 123456789a$ e $\rho = 49a8173526$ e $I = [3, 6]$. Logo $\pi_I = \{3, 4, 5, 6\}$ e $\rho_I = \{a, 8, 1, 7\}$, e temos os mapeamentos

π_i	3	4	5	6
$m(\pi_i)$	a	8	1	7
$m^*(\pi_i)$	a	8	1	7

,

i.e., o mapeamento iterado m^* é igual a m. Obtemos

Índice i	1	2	3	4	5	6	7	8	9	10
Elem.	$m^*(4)$	ρ_2	π_3	π_4	π_5	π_6	$m^*(3)$	$m^*(5)$	ρ_9	$m^*(6)$
σ_i	8	9	3	4	5	6	a	1	2	7

$\Diamond$

Exemplo 4.2 (Recombinação PMX)
Seja $\pi = 123456789a$ e $\rho = 361a849725$ e $I = [3, 6]$. Logo $\pi_I = \{3, 4, 5, 6\}$ e $\rho_I = \{a, 8, 1, 7\}$, e temos os mapeamentos

π_i	3	4	5	6
$m(\pi_i)$	1	a	8	4
$m^*(\pi_i)$	1	a	8	a

.

Obtemos

Índice i	1	2	3	4	5	6	7	8	9	10
Elem.	$m^*(3)$	$m^*(6)$	π_3	π_4	π_5	π_6	ρ_7	ρ_8	ρ_9	$m^*(5)$
σ_i	1	a	3	4	5	6	9	7	2	8

$\Diamond$

A seleção de um ou mais operadores de recombinação é um parte importante do projeto de uma heurística por recombinação. Além das recombinações genéricas, uma recombinação que aproveita a estrutura do problema deve ser considerada.

Exemplo 4.3 (Recombinação EAX para o PCV)
O *edge assembly crossover* (EAX) (Nagata e Kobayashi 1997) trabalha na representação de rotas por conjuntos de arestas. Para rotas A e B ele forma $A \cup B$ e extrai um conjunto completo de ciclos AB-alternantes (i.e. ciclos com arestas alternadamente e A e B; isso sempre é possível). Seleciona um subconjunto S dos ciclos AB extraídos e gera uma coleção de ciclos $A \oplus S$. Repetidamente reconecta o menor ciclo com um outro ciclo até obter uma rota simples.
Para conectar ciclos C e D (representados por conjuntos de arestas), gulosamente seleciona o par de arestas $uu' \in C$ e $vv' \in D$ tal que $(C \cup D) \oplus \{uu', vv', uv, u'v\}$ tem custo mínimo.

$\Diamond$

4.1. Religamento de caminhos

O *religamento de caminhos* (ingl. path relinking), proposto por Glover (1996) no contexto da busca tabu, explora trajetórias entre uma *solução inicial* s e uma *solução guia* s'. Isso é realizado com uma busca local na vizinhança reduzida ("vizinhança direcionada") $D(s) = \{s'' \in N(s) \mid d(s'', s') < d(s, s')\}$. Logo em no máximo $d(s, s')$ passos a busca transforma s em s'. Qualquer distribuição de probabilidade discutida no cap. 2 pode ser usada para explorar D; tipicamente é usada a estratégia "melhor vizinho". O resultado do religamento de caminhos é a melhor solução s^* encontrada na trajetória explorada. Como a melhor solução da trajetória s^* não necessariamente é um mínimo local de N, é comum aplicar uma busca local em N.

Algoritmo 4.1 (Religamento de caminhos)
Entrada Uma solução inicial s, uma solução guia s'.

Saída Uma solução s^* com $\varphi(s^*) \leq \min\{\varphi(s), \varphi(s')\}$.

```
1  PathRelinking(s,s') :=
2    while D(s) ≠ ∅ ∧ s ≠ s' do
3      s* = argmin{φ(s), φ(s')}
4      seleciona s'' ∈ D(s) com probabilidade P_s(s'')
5      s := s''
6      atualiza o incumbente s*
7    end
8    return s*
```

Observação 4.1 (Conectividade da vizinhança direcionada)
Caso é garantido que na vizinhança D existe um caminho de s para s' podemos simplificar a condição da linha 2 para $s \neq s'$. Um exemplo em que isso não é satisfeito: para o problema do exemplo 1.7 pode ser conveniente restringir a vizinhança N que desloca uma tarefa para outra estação às estações críticas, i.e. as estações com tempo de estação igual ao tempo de ciclo. Logo o religamento de caminhos termina, caso as tarefas alocadas às estações críticas na solução atual e guia são as mesmas. ◊

Variantes comuns são: religamento de caminhos

para frente (ingl. forward path relinking, "uphill") Para soluções s_1 e s_2 com $\varphi(s_1) \leq \varphi(s_2)$ explore a trajetória de s_1 para s_2.

para trás (ingl. backward path relinking, "downhill") Para soluções s_1 e s_2 com $\varphi(s_1) \leq \varphi(s_2)$ explore a trajetória de s_2 para s_1.

para trás e frente (ingl. back-and-forward path relinking) Para soluções s_1 e s_2 com $\varphi(s_1) \leq \varphi(s_2)$ explore a trajetória de s_2 para s_1, seguido da trajetória de s_1 para s_2.

misto (ingl. mixed path relinking) Altera ambas soluções até eles se encontram.

truncado (ingl. truncated path relinking) Explora a trajetória somente no início ou no final. Esse estratégia é justificada por experimentos que mostram que as melhores soluções tendem a ser encontradas no início ou no final da trajetória.

Observação 4.2
O religamento de caminhos explora a vizinhança da solução inicial melhor. Logo, caso somente uma trajetória é explorada, é melhor usar um religamento para frente, que começa da melhor das soluções (Resende e Ribeiro 2005). ◊

Observação 4.3 (Seleção do vizinho)
Qualquer estratégia de busca local pode ser aplicada na seleção da linha 4. Aplicando a estratégia "guloso-α", por exemplo, obtemos um *religamento de caminhos guloso adaptativo* (ingl. greedy randomized adaptive path-relinking, GRAPR). ◊

4.2. Probe

O *population-reinforced optimization-based exploration* (PROBE) trabalha com uma *população* de soluções $S_1, \ldots, S_n$. Sendo $C(\cdot,\cdot)$ algum operador que recombina duas soluções, Probe produz em cada iteração uma nova população $C(S_1, S_2), C(S_2, S_3), \ldots, C(S_n, S_1)$.

Teorema 4.1 (Convergência de Probe)
Caso $\varphi(C(S,T)) \leq \min\{\varphi(S), \varphi(T)\}$ o valor médio da população diminui até todas soluções possuem o mesmo valor.

Prova. Supõe que um par de soluções adjacentes S_j, S_{j+1} não possui o mesmo valor. Logo $\varphi(C(S_j, S_{j+1}) < \varphi(S_j)$ ou $\varphi(C(S_j, S_{j+1}) < \varphi(S_{j+1})$ e como as restantes soluções satisfazem $\varphi(C(S_i, S_{i+1}) \leq \varphi(S_i)$ resp. $\varphi(C(S_i, S_{i+1}) \leq \varphi(S_{i+1})$ o valor médio diminui. ■

Observação 4.4 (Convergência trivial)
Para $C(S,T) = \operatorname{argmin}\{\varphi(S), \varphi(T)\}$ a população converge para a melhor das n soluções inicias. ◊

4.3. Scatter search

A *busca dispersa* (ingl. Scatter search) é um esquema algorítmico que explora o espaço de busca sistematicamente usando um *conjunto de soluções de referência* (ingl. reference set). A enfase da busca dispersa é na exploração determinística e sistemática, similar com a busca tabu, ao contrário de métodos que focam em randomização. Repetidamente a busca dispersa combina um subconjunto das soluções de referência para gerar novas soluções e atualiza as soluções de referência. O método procura incluir elementos de diversificação e intensificação estrategicamente. As soluções de referência R, por exemplo,

tipicamente contém soluções de boa qualidade e soluções diversas. O conjunto de soluções de referência inicial é selecionado entre um número grande de soluções diversas. Depois da recombinação o novo conjunto de soluções de referência é selecionado entre as soluções de referência atuais e as soluções obtidas por recombinação.
Seja $d(p, S) = \min\{d(p, s) \mid s \in S\}$ e distância mínima da solução p para qualquer solução do conjunto S. Um exemplo de uma construção do conjunto de referência que seleciona b_1 soluções de boa qualidade e b_2 soluções diversas é

```
1  refset(P) :=  { seleciona soluções de referência de P }
2    seja P = {p_1, ..., p_n} com φ(p_1) ≤ ··· ≤ φ(p_n)
3    S := {p_1, ..., p_{b_1}}
4    P := P \ S
5    while P ≠ ∅ ∧ |S| ≤ b_1 + b_2 do
6      p := argmax_p{d(p, S) | p ∈ P}
7      S := S ∪ {p}
8      P := P \ {p}
9    end
```

Com isso obtemos

Algoritmo 4.2 (Scatter search)
Entrada Uma instância de um problema.

Saída Uma solução s, caso for encontrada.

```
1  ScatterSearch() :=
2    cria um conjunto de soluções diversas C
3    R := refset(C)
4    do
5      seja 𝒮 uma família de subconjuntos de R
6      C := ∅
7      for S ∈ 𝒮 do
8        T := recombine(S)
9        C := C ∪ improve(T)
10     end for
11     R := refset(R ∪ C) { alternativa: refset(C) }
12   while R changed
```

A tabela 4.1 mostra valores de referência para os parâmetros da busca dispersa.

Observação 4.5 (Atualização do conjunto de referência)
Existem diversas estratégias de atualização do conjunto de soluções de re-

Tabela 4.1.: Valores de referência para os parâmetros da busca dispersa.

Número de soluções de referência $\|R\|$	≈ 20
Número de soluções iniciais $\|C\|$	$\geq 10\|R\|$
Número de soluções elite b_1	$\approx \|R\|/2$
Número de soluções diversas b_2	$\approx \|R\|/2$

ferência. Por exemplo, podemos adicionar uma nova solução ao conjunto de referência R caso (i) $|R| < b$, ou (ii) ela é melhor que o incumbente, ou (iii) ela é melhor que a pior solução de R, dado que ela possui uma distância mínima d das soluções restantes. Em ambos casos a solução de menor distância com a nova solução sai do conjunto de referência. Para implementar isso, podemos modificar o algoritmo 4.2 para

```
11    for each c ∈ C: refset(R, c)
```

usando o procedimento

```
1  refset(R,s) := { atualiza o conjunto R com s }
2    seja R = {r_1, ..., r_n} com φ(r_1) ≤ ··· ≤ φ(r_n)
3    if |R| < b then
4      R := R ∪ {s}
5    else if φ(s) < φ(r_1) ∨ (φ(s) < φ(r_n) ∧ min_i d(s, r_i) > d then
6      seja k = argmin_i d(s, r_i)
7      R := R \ {r_k} ∪ {s}
8    end if
9  end
```

◊

Observação 4.6 (Seleção da família $\mathcal{S}$)
A abordagem mais comum é selecionar todos pares de soluções de referência. Variantes propostas na literatura incluem escolher triplas formadas por todos pares mais a solução de referência melhor que não faz parte do par, ou escolher quadruplas formadas por todas triplas mais a solução de referência melhor que não faz parte da tripla. Essas abordagens são raras, por precisarem uma combinação efetiva entre mais que duas soluções. ◊

4.4. GRASP com religamento de caminhos

GRASP com religamento de caminhos mantém um conjunto de soluções de referência. Este conjunto é alimentado pelas soluções obtidas em cada iteração. Uma proposta típica da atualização é a regra da observação 4.5. Em cada

iteração, GRASP+PR aplica religamento de caminhos entre o mínimo local obtido s e uma solução de referência r. A solução de referência é selecionada, por exemplo, com probabilidade $\propto d(s, r)$, para religar soluções distantes com maior probabilidade.
O *GRASP evolucionário* (ingl. evolutionary GRASP) reconstrói o conjunto de soluções de referência periodicamente. Os candidatos para formar o novo conjunto de soluções são as soluções obtidas por religamento de caminhos entre todos pares de soluções de conjunto de referência do período anterior.

4.5. Algoritmos genéticos e meméticos

Observação 4.7 (Função objetivo e aptidão)
Como algoritmo genéticos e variantes normalmente são formulados para maximizar uma função objetivo – chamada *aptidão* (ingl. fitness) – vamos seguir essa convenção nesta seção. ◊

Algoritmos genéticos (ingl. genetic algorithms) foram propostas por Holland (1975) em analogia com processos evolutivos. Um algoritmo genético mantém uma população $S_1, \ldots, S_n$ de indivíduos e repetidamente seleciona dois indivíduos *pais*, gera novos indivíduos por recombinação dos pais, eventualmente aplica uma mutação em indivíduos selecionados, e atualiza a população. Um algoritmo genético difere da busca dispersa principalmente pelos elementos randomizados: a seleção dos pais é aleatória (mas tipicamente proporcional com a qualidade da solução) bem como a mutação. Obtemos um *algoritmo memético* (ingl. memetic algorithm) caso um indivíduo é melhorado por uma busca local, e um *algoritmo genético Lamarckiano* caso essa melhora é herdável (i.e. a transformação inversa do fenótipo para genótipo existe, ver cáp. 1.2.2).
A terminologia biológica é frequentemente usada em algoritmos genéticos. Numa representação de variáveis, por exemplo, uma variável é chamada *gene* e os valores que ela pode assumir os *alelos*.
O algoritmo 4.3 define um esquema genérico de um algoritmo genético. Ele é definido por (i) uma população inicial, (ii) por uma estratégia de seleção de indivíduos, (iii) operadores de recombinação e mutação, e (iv) uma estratégia de seleção da nova população.

Algoritmo 4.3 (Algoritmo genético)
Entrada Uma instância de um problema.

Saída Uma solução s, caso for encontrada.

```
1  GeneticAlgorithm() :=
```

```
2    cria um conjunto de soluções iniciais P
3    until critério de parada satisfeito
4      C := ∅
5      { recombinação }
6      seja 𝒫 um conjunto de pais selecionados de P
7      for p = (p1, p2) ∈ 𝒫 do
8        T := recombine(p1, p2)
9        C := C ∪ improve(T)
10     end for
11     { mutação }
12     seja ℳ ⊆ P ∪ C de soluções que sofrem mutação
13     for s ∈ ℳ do
14       T := mutate(s)
15       C := C ∪ improve(T) \ {s}
16     end for
17     P := update(P, C) { com update (μ + λ), (μ, λ) }
18   end
```

Exemplo 4.4 (Algoritmo genético básico)
Uma instância básica do algoritmo 4.3 usa

- uma representação por variáveis com $V = \{0, 1\}$;
- uma população inicial com μ indivíduos aleatórios;
- uma seleção de $|\mathcal{P}| = \mu$ pares de pais, cada solução s com probabilidade $\propto \varphi(s)$;
- uma recombinação em um ponto (p. 55) que gera duas novas soluções;
- nenhum procedimento de melhora (improve$(C) = C$);
- uma mutação que inverte cada variável com probabilidade p (frequentemente $p = 1/|I|$) nas novas soluções;
- uma atualização (μ, λ) da população (seleciona os μ melhores entre os novos indivíduos).

$\Diamond$

4.5.1. População inicial

A população é criada por alguma heurística construtiva, frequentemente com indivíduos aleatórios. Reeves (1993) propõe um tamanho mínimo que garante que todas soluções podem ser obtidas por recombinação da população inicial, i.e. todo alelo está presente em todo gene. Para uma inicialização aleatória uniforme na representação por variáveis, temos $|V|^n$ possíveis combinações de alelos num determinado gene, para uma população de tamanho n. Dessas combinações $|V|!\left\{ {n \atop |V|} \right\}$ possuem todos alelos, logo a probabilidade que todos alelos são presentes em todos genes k é

$$\left(|V|! \left\{ {n \atop |V|} \right\} |V|^{-n} \right)^k.$$

Em particular para $|V| = 2$ obtemos a probabilidade $(1-2^{1-n})^k$. Isso permite selecionar um n tal que a probabilidade de que todos alelos estejam presentes é alta.

4.5.2. Seleção de indivíduos

Um indivíduo S é selecionado como pai com probabilidade $\propto \varphi(s)$ ou conforme alguma regra de seleção baseado no rank na população (ver pág. 48). Outro exemplo é uma *seleção por torneio* que seleciona o melhor entre k indivíduos aleatórios, similar da busca por amostragem.

Observação 4.8 (Seleção por torneio)
Um 1-torneio é equivalente com uma seleção aleatória. Num 2-torneio a probabilidade de selecionar o elemento com posto i é $(n-i)/\binom{n}{2}$, logo obtemos uma seleção linear por posto. Em geral a probabilidade de selecionar o elemento com posto i num k-torneio é

$$\binom{n-i}{k-1} / \binom{n}{k} \propto \binom{n-i}{k-1} = \Theta((n-i)^{k-1}).$$

◊

Exemplo 4.5 (Fitness uniform selection scheme (FUSS))
Hutter e Legg (2006) propõem um *esquema de seleção uniforme baseada em aptidão* (ingl. *fitness uniform selection scheme*): escolhe um valor uniforme f no intervalo $[\min_{i \in P} \varphi(i), \max_{i \in P} \varphi(i)]$ e seleciona o indivíduo com valor da função objetivo mais próximo de f. O objetivo da seleção é manter a população diversa: indivíduos em regiões com menor densidade da distribuição dos valores da função objetivo possuem uma probabilidade maior de seleção.

◊

Exemplo 4.6 (Seleção estocástica universal)
Baker (1987) propõe uma *seleção estocástica universal* (ingl. *stochastic uniform selection*): Seja p_i, a probabilidade de selecionar indivíduo $i \in [\mu]$, e $P_i = [\sum_{k\in[i-1]} p_i, \sum_{k\in[i]} p_i)$ o intervalo correspondente, seleciona, para um $r \in 1/\mu$ aleatório, os indivíduos $i_1, \ldots, i_\mu$ tal que $k/\mu \in P_{i_k}$ para $k \in [\mu]$. (A explicação mais simples dessa seleção é por uma roleta com μ seletores de distância $1/\mu$). ◊

4.5.3. Recombinação e mutação

Para recombinação de indivíduos serve qualquer das recombinações discutidas acima, inclusive o religamento de caminhos. Uma mutação é uma pequena perturbação de uma solução. Logo ela pode ser realizada por um passo de uma busca local estocástica 2.1. Recombinação ou mutação podem ser aplicados com probabilidades diferentes, eventualmente dinâmicas.

4.5.4. Seleção da nova população

A população pode ser atualizada depois de criar um número suficiente de novas soluções, selecionando uma nova população entre estes indivíduos, eventualmente incluindo a população antiga. Uma alternativa é atualizar a população constantemente. (Observe que isso corresponde exatamente com as estratégias de seleção da busca dispersa.) As primeiras duas estratégias de seleção levam a um *algoritmo genético geracional* e a última a um *algoritmo genético em estado de equilíbrio* (ingl. steady state genetic algorithm). Para uma população de tamanho μ e λ novos indivíduos eles também são conhecidos por *seleção* (μ, λ) (seleciona os μ melhores dos λ novos indivíduos) ou *seleção* $(\mu+\lambda)$ (seleciona os μ melhores entre a população antiga e os λ novos indivíduos). Caso uma seleção permite soluções da população antiga entre na nova população, e seleciona algumas das melhores soluções, o algoritmo é *elitista*.

Exemplo 4.7 (Estratégias de evolução)
Estratégias de evolução (ingl. evolution strategies) são algoritmos genéticos sem recombinação. Eles recebem o nome da atualização correspondente: (μ, λ) ou $(\mu+\lambda)$. Observe que uma estratégia de evolução $(1+1)$ é uma busca local monótona estocástica. ◊

Uma outra estratégias comum é a deleção randomizada de indivíduos do conjunto de candidatos até μ indivíduos sobram. A variante mais simples delete indivíduos com probabilidade uniforme; uma variante delete com probabilidade $\propto \varphi(s_{\max}) + \varphi(s_{\min}) - \varphi(s)$ com $s_{\max}$ a melhor e $s_{\min}$ a pior solução.

Exemplo 4.8 (Fitness uniform deletion scheme (FUDS))
Hutter e Legg (2006) propõem um *esquema de deleção uniforme baseado em aptidão* (ingl. *fitness uniform deletion scheme*): similar ao FUSS, escolhe um valor uniforme f no intervalo $[\min_{i \in P} \varphi(i), \max_{i \in P} \varphi(i)]$ e deleta o indivíduo com valor da função objetivo mais próximo de f. FUDS favorece uma exploração em regiões de menor densidade da distribuição dos valores da função objetivo. ◊

Observação 4.9 (Resultados experimentais (Levine 1997))
Experimentalmente, parece que

- manter a população em estado de equilíbrio é preferível sobre abordagens geracionais;
- uma recombinação uniforme ou em dois pontos é preferível sobre uma em um único ponto;
- uma seleção proporcional com φ raramente é bom;
- uma taxa de mutação dinâmica é preferível;
- manter a diversidade da população é importante.
- operadores de recombinação e mutação específicos para o problema são mais úteis;

◊

Observação 4.10 (Resultados teóricos)
Pela teoria sabemos que

- o desempenho depende fortemente do problema: existem funções unimodais em que uma determinada estratégia de evolução $(1 + 1)$ precisa tempo exponencial mas também classes de funções que podem ser resolvidos em tempo polinomial (Droste et al. 2002; Jansen e Wegener 2000); e existem instâncias de problemas NP-completos em que uma estratégia de evolução $(1 + 1)$ não possui garantia de aproximação (e.g. cobertura por vértices (Friedrich et al. 2010)), mas também problemas NP-completos em que a estratégia garante uma aproximação (e.g. uma 4/3-aproximação em tempo esperado $O(n^2)$ para o problema de partição[1] (Witt 2005)).

[1] Particionar um conjunto de números $x_1, \ldots, x_k$ tal que a diferença das somas dos partes é mínima.

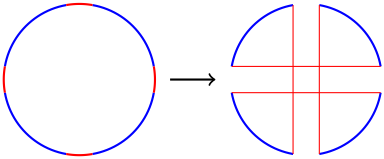


Figura 4.1.: Um movimento 4-opt com dois pontes.

- o tamanho ideal da população depende fortemente do problema: existe uma função em que uma dada estratégia de evolução $(\mu, 1)$[2] precisa tempo exponencial para μ pequeno, mas tempo polinomial para μ grande e vice versa (Witt 2008);
- o desempenho depende fortemente da função objetivo: uma estratégia de evolução $(1+1)$ consegue ordenar n números em tempo $\Theta(n^2 \log n)$, mas existem funções objetivos para medir o grau da ordenação que levam a um tempo exponencial (Scharnow et al. 2002);

$\Diamond$

A última observação experimental, que não é restrito para algoritmos genéticos, em conjunto com os resultados teóricos, é o motivo para conjeturar que (i) para cada solução "genérica" de um problema, existe um algoritmo heurístico específico melhor. (ii) para cada heurística que funciona bem na prática (i.e. resolve o problema em tempo esperado polinomial com garantia de qualidade) deve existir um subproblema do problema em questão em P.

Princípio de projeto 4.1 (Estrutura do problema)
Procure aproveitar a estrutura do problema. Caso a heurística funciona bem: procure identificar quais características das instâncias são responsáveis por isso.

Exemplo 4.9 (Algoritmo genético para o PCV)
Em Johnson e McGeoch (2003) o algoritmo genético melhor é degenerado para uma busca local iterada: a "população" consiste de uma única solução, e o algoritmo aplica repetidamente uma busca local Kernighan-Lin e uma mutação na vizinhança 4-exchange restrito para dois pontes (Fig. 4.1), i.e. a estratégia de atualização é $(1, 1)$. $\Diamond$

Exemplo 4.10 (Algoritmo genético para o PCV)
O algoritmo genético para o PCV de Nagata e Kobayashi (2012) exemplifica o princípio 4.1. Ele usa

[2] A estratégia padrão com atualização por deleção aleatória.

- Uma população inicial de tamanho 300 com rotas aleatórias otimizadas por 2-opt.
- Uma recombinação entre π_i e π_{i+1} para uma permutação aleatória da população.
- A recombinação entre p, q aplica uma variante "localizada" de EAX (i.e. produz soluções mais similares com p) e gerar diversas novas soluções $f_1, \ldots, f_k$ ($k \approx 30$).
- Uma seleção que substitui o p atual pela melhor soluções entre $f_1, \ldots, f_k, p$.
- Uma função objetivo modificada que procura manter a diversidade da população. Para $P_i = (p_{ij})_j$ a distribuição de probabilidade dos arcos (i, j) na população, define a entropia da população por

$$H = \sum_{i \in [n]} H_i; \qquad H_i = -\sum_{j \in [n]} p_{ij} \log p_{ij}$$

e seleciona a solução s de maior valor

$$\varphi(s) = \begin{cases} -\Delta L(s)/\epsilon & \text{caso } \Delta L(s) < 0,\ \Delta H(s) \geq 0 \\ \Delta L(s)/\Delta H(s) & \text{caso } \Delta L(s) < 0,\ \Delta H(s) < 0 \\ -\Delta L(s) & \text{caso } \Delta L(s) \geq 0 \end{cases}$$

com $\Delta L(s)$ o aumento da distância total média da população caso s substitui p, e $\Delta H(s)$ o aumento correspondente da entropia.

$\Diamond$

4.5.5. O algoritmo genético CHC

O "Cross-generational elitist selection, Heterogeneous recombination, and Cataclysmic mutation" (CHC) é um exemplo de uma variante de um algoritmo genético com um foco em intensificação (Eshelman 1990). Ele recombina sistematicamente todos pares da população atual, e procura manter a diversidade por recombinar somente soluções suficientemente diferente com uma recombinação HUX. A recombinação HUX é uniforme, mas troca exatamente a metade das variáveis diferentes entre os pais e gera dois novos filhos. Caso a população convergiu ele é recriada aplicando uma mutação para a melhor solução.

Algoritmo 4.4 (Algoritmo genético CHC)
Entrada Uma instância de um problema, uma taxa de mutação p_m (típico: $p_m = 1/2$).

Saída Uma solução s, caso for encontrada.

```
1  CHC() :=
2    cria um conjunto de soluções iniciais P
3    d := p_m(1 − p_m)|I|
4
5    until critério de parada satisfeito
6      C := ∅
7      for n/2 iterações do
8        seleciona pais p_1, p_2 ∈ P aleatoriamente
9        if d(p_1, p_2) > 2d then
10         T := HUX(p_1, p_2)
11         C := C ∪ T;  P := P \ {p_1, p_2}
12       end
13     end
14     if C = ∅ then
15       d := d − 1
16     else
17       P := (μ + λ)(P ∪ C)
18     end if
19     if d < 0 then
20       { re−criação cataclísmica }
21       reduz P para a melhor solução p em P
22       until |P| = μ do
23         aplica uma mutação em p com prob. 0.35
24         insere o indivíduo obtido em P
25       end
26       d := p_m(1 − p_m)|I|
27     end if
28   end
29 end
```

4.5.6. Algoritmos genéticos com chaves aleatórias

Um "biased random-key genetic algorithm" (BRKGA) é uma extensão do algoritmo genético com chaves aleatórias de Bean (1994). Ambos usam uma

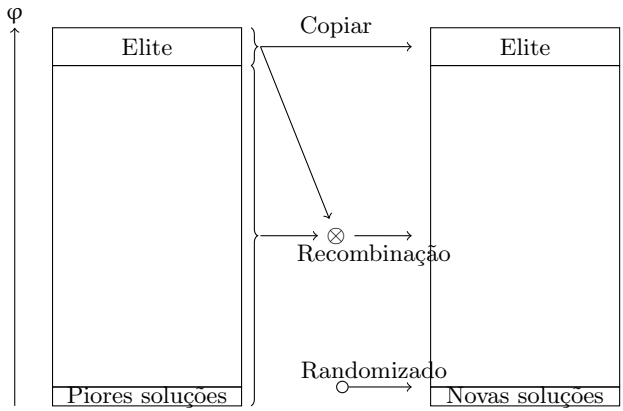


Figura 4.2.: Algoritmo genético com chaves aleatórias.

representação por chaves aleatórias (seção 1.2.2) e uma população com três "castas" (ver Fig. 4.2). A nova população consiste da elite da população antiga, soluções randômicas que substituem as piores soluções e soluções que foram obtidas por recombinação uniforme. No caso do BRKGA a recombinação uniforme é substituída por uma recombinação que passa de cada gene independentemente o alelo do pai elite com probabilidade $p \geq 0.5$ para o filho. Tamanhos típicos para a elite são $10-20\%$ da população, e $1-5\%$ de soluções randômicas.

4.6. Otimização com enxames de partículas

A otimização com enxames de partículas (ingl. particle swarm optimization, PSO) (Eberhart e Kennedy 1995) foi proposta para otimização contínua e mantém uma população de soluções $x_1, \ldots, x_n$ em $\mathbb{R}^k$. Cada solução também possui uma velocidade v_i, $i \in [n]$ e em cada passo a posição é atualizada para $x_i' = x_i + \epsilon v_i$ para um parâmetro $\epsilon \in (0, 1]$. A velocidade v_i é atualizada em direção da melhor solução na trajetoria da solução atual x_i^*, da melhor solução $x_I^* = \max_{i \in I} x_i^*$ encontrada por soluções *informantes* $I \subseteq [n]$ e da melhor solução global $x_{[n]}$ por

$$v_i' = \alpha v_i + \beta(x_i^* - x_i) + \gamma(x_I^* - x_i) + \delta(x_{[n]}^* - x_i). \tag{4.1}$$

Com isso obtemos o esquema genérico

Algoritmo 4.5 (Otimização com enxames de partículas)
Entrada Uma instância de um problema, parâmetros $\alpha, \beta, \gamma, \delta, \epsilon$.

Saída A melhor solução encontrada.

```
1  PSO() :=
2    cria soluções iniciais x_1,...,x_n
3      com velocidades v_1,...,v_n
4
5    until critério de parada satisfeito
6      for cada solução i ∈ [n] do
7        seleciona um conjunto de informantes I
8        atualiza v_i de acordo com (4.1)
9        x_i := x_i + ε v_i
10   end
11   return x*_[n]
12 end
```

Na forma mais comum:

- Aproximadamente 50 soluções e velocidades inicias são escolhidas aleatoriamente.
- O conjunto de informantes é um subconjunto aleatório de $[n]$.

Variantes incluem:

- Selecionar em cada aplicação de (4.1) valores aleatórias em $[0, \beta]$, $[0, \gamma]$ e $[0, \delta]$ para os pesos.

Aplicação para otimização discreta A forma mais simples de aplicar a otimização com enxames de partículas em problemas discretos é trabalhar no espaço real e transformar a solução para uma solução discreta (seção 1.2.2). Uma alternativa é definir uma estratégia de atualização discreta.

Exemplo 4.11 (Variante binária de PSO)
Kennedy e Eberhart (1997) propõem para soluções in $\{0,1\}^k$ mapear as velocidades em $\mathbb{R}^k$ para o $[0,1]^k$ por uma transformação logística $S(x) = (1 + e^{-}x)^{1}$ aplicada a cada elemento do vetor, e interpretar os componentes das velocidades como probabilidades. Em cada passo x_{ij} recebe o valor 1 com probabilidade $S(v_{ij})$. ◊

4.7. Sistemas imunológicos artificiais

Sistemas imunológicos artificiais (ingl. artificial immunological systems) são algoritmos de otimização usando princípios de sistemas imunológicos. Daremos somente um exemplo de um algoritmos comum dessa classe. O princípio natural do algoritmo é a observação que o sistema imunológico se adapta para novas antigenes por clonagem e amadurecimento.

Algoritmo 4.6 (SIA/Clonalg)
Entrada Uma instância de um problema, parâmetros α, β.

Saída A melhor solução encontrada.

```
1  Clonalg() :=
2    seja P = {p_1,...,p_n} aleatória com φ(p_1) ≤ ··· ≤ φ(p_n)
3
4    until critério de parada satisfeito
5      seleciona as α% melhores soluções p_1,...,p_k
6      for i ∈ [k] do
7        { clonagem }
8        cria um conjunto C_i de ∝ 1/i cópias de p_i
9        { amadurecimento por hípermutação }
10       aplica uma mutação a c ∈ C_i com taxa ∝ φ(s)
11     end
12     selecione a nova população entre P e ∪_i C_i
13     substitui as β% piores soluções
14       por soluções aleatórias
15   end
16 end
```

4.8. Intensificação e diversificação revisitada

Uma população de soluções de alta qualidade junto com a recombinação de soluções também serve para realizar uma intensificação e diversificação genérica (Watson et al. 2006). O IMDF (Intensification/Diversification framework) supõe que temos uma heurística de busca $H(x_0, i)$ base arbitrária, que podemos rodar para um número de iterações i numa instância inicial x_0.

Algoritmo 4.7 (IDMF)
Entrada Uma instância de um problema, probabilidade de intensificação

p_i, uma heurística H, iterações $i_0 > i_1$ para intensificação.
Saída A melhor solução encontrada.

```
1  H*(x0) :=
2    x := H(x0, i0)
3    while φ(x) < φ(x0)
4      x0 := x
5      x := H(x0, i1)
6    end
7    return x0
8  end
9
10 IDMF() :=
11   gera uma população E de ótimos locais
12   aplica H*(e) em cada e ∈ E
13   repeat
14     com probabilidade pi: { intensificação }
15       seleciona e ∈ E
16       g := e
17     com probabilidade 1−pi: { diversificação }
18       seleciona e, f ∈ E
19       gera um elemento g no meio entre e e f
20         por religamento de caminhos
21     e' := H*(g)
22     if φ(e') < φ(e)
23       e := e'
24   end
25 end
```

4.9. Notas

Mais sobre a busca dispersa se encontra em Gendreau e Potvin (2010, cáp. 4), Glover e Kochenberger (2002, cáp. 1) e Talbi (2009, cáp. 3.4). Uma aplicação recente do operador EAX num algoritmo genético se encontra em Nagata e Kobayashi (2012).

4.9.1. Até mais, e obrigado pelos peixes!

Para quem não é satisfeito com os métodos discutidos: usa alguma outra *besta de carga* como

> fireflies, monkeys, cuckoos, viruses, bats, bees, frogs, ou competitive imperialists,

ou deixa a física resolver o problema com

> gravitational search, intelligent waterdrops, ou harmony search.

Porém, é importante lembrar que o objetivo da pesquisa em heurísticas não é produzir novos vocabulários para descrever as mesmas estratégias, mas entender quais métodos servem melhor para resolver problemas. Weyland (2010), por exemplo, mostra que a busca de harmonias (ingl. harmony search) é uma forma de uma estratégia de evolução. Para uma crítica geral ver também Sörensen (2013).

5. Tópicos

5.1. Hibridização de heurísticas

A combinação de técnicas de diversas meta-heurísticas ou de uma meta-heurística com técnicas das áreas relacionadas de pesquisa operacional ou inteligência artificial define *heurísticas híbridas*. Um exemplo é a combinação de técnicas usando populações para identificar regiões promissoras no espaço de busca com técnicas de busca local para intensificar a busca. Um outro exemplo é o uso de programação matemática ou constraint programming para resolver subproblemas ou explorar vizinhanças grandes. Isso é um exemplo de *matheuristics*, a combinação de heurísticas com técnicas de programação matemática, também conhecida por *heurísticas baseados em modelos matemáticos* (ingl. model-based heuristics).

5.1.1. Matheuristics

Hibridizações básicas entre heurísticas e programação matemática aplicam as heurísticas para obter limitantes superiores em algoritmos de branch-and-bound ou usam programação matemática para resolver subproblemas em heurísticas. Exemplos de outras hibridizações são relaxações lineares de programas inteiros para gerar soluções inicias ou guiar buscas, e a aplicação de técnicas heurísticas para guiar a exploração de buscas em algoritmos exatos.

Exemplo 5.1 (Diving)
Algoritmos branch-and-bound frequentemente expandem o nodo com o menor limite inferior. *Diving* é uma estratégia que estrategicamente aplica uma busca por profundidade para gerar melhores soluções. ◊

Exemplo 5.2 (Ramificação local)
Ramificação local (ingl. local branching) guia a exploração das soluções de programas inteiras $0{-}1$ de um resolvedor genérico para analisar primeiramente soluções de distância Hamming $\leq k$. A distância Hamming das soluções $x = (x_1, \ldots, x_n) \in \mathbb{B}^n$ e $\overline{x} = (\overline{x}_1, \ldots, \overline{x}_n) \in \mathbb{B}^n$ é

$$\Delta(x, \overline{x}) = \sum_{i \in [n] | x_i = 0} \overline{x}_i + \sum_{i \in [n] | x_i = 1} 1 - \overline{x}_i.$$

Com isso para uma dada solução x_0 uma estratégia global de ramificação resolve primeiramente o programa inteiro $Ax \leq b \wedge \Delta(x, x_0) \leq k$ e só depois $Ax \leq b \wedge \Delta(x, x_0) \geq k + 1$. Essa ramificação continua no primeiro subproblema, caso o resolvedor encontra uma melhor solução. Fischetti e Lodi (2003) sugerem $k \in [10, 20]$. ◊

Exemplo 5.3 (RINS e religamento de caminhos)
O *relaxation induced neighorhood search* (RINS) é uma estratégia para intensificar a busca para melhores soluções viáveis. Para um dado nó na árvore de branch-and-bound da solução de um programa inteiro, ela fixa as variáveis que possuem o mesmo valor no incumbente e na relaxação linear atual, e resolve o subproblema nas restantes variáveis restrito para um valor máximo da função objetivo e com um tempo limite. Danna et al. (2005) propõem aplicar RINS cada $f \gg 1$ nós com um limite de nós explorados, e.g. $f \approx 100$, com limite de ≈ 1000 nós.
Uma forma similar de explorar o espaço entre duas soluções é uma extensão do religamento de caminhos: fixa todas variáveis em comum, e resolve o problema no subespaço resultante de forma exata. ◊

Exemplo 5.4 (Geração heurística de colunas)
Na geração de colunas (usado também em algoritmos de branch-and-price) o subproblema de pricing precisa encontrar uma coluna com custo reduzido negativo. Para melhorar os limitantes inferiores da decomposição de Dantzig-Wolfe, o subproblema de pricing deve ser o mais difícil possível, que pode ser resolvido em tempo aceitável. Uma estratégia diferente resolve o subproblema de pricing heuristicamente. O método continue ser correto caso no final o subproblema de pricing é resolvido pelo menos uma vez exatamente para demonstrar que não existem mais colunas com custo reduzido negativo.
Por exemplo o problema de colorar um grafo não-direcionado $G = (V, E)$ com o menor número de cores

$$
\begin{aligned}
\textbf{minimiza} \quad & \sum_{i \in [n]} c_i, && \\
\textbf{sujeito a} \quad & \sum_{i \in [n]} x_{vi} \geq 1, && \forall v \in V, \\
& x_{ui} + x_{vi} \leq 1, && \forall \{u, v\} \in E, i \in [n], \\
& c_i \geq \sum_{v \in V} x_{vi}/n, && \forall i \in [n], \\
& x_{vi}, c_i \in \mathbb{B}, && \forall v \in V, i \in [n],
\end{aligned}
$$

pode ser decomposto em um problema mestre de cobertura por conjuntos independentes maximais I de G

$$\textbf{minimiza} \quad \sum_{i \in I} x_i \tag{5.1}$$

$$\textbf{sujeito a} \quad \sum_{i \in I | v \in I} x_i \geq 1 \qquad \forall v \in V \tag{5.2}$$

$$x_i \in \mathbb{B} \qquad \forall i \in I. \tag{5.3}$$

Para custos reduzidos λ_v, $v \in V$ o subproblema problema de pricing é encontrar um conjunto independente máximo de maior peso

$$\begin{aligned} \textbf{maximiza} \quad & \sum_{v \in V} \lambda_v z_v \\ \textbf{sujeito a} \quad & z_u + z_v \leq 1 && \forall \{u, v\} \in E \\ & z_v \in \mathbb{B} && v \in V. \end{aligned}$$

Filho e Lorena (2000) propõem um algoritmo genético para resolver o subproblema de pricing. ◊

5.1.2. Dynasearch

Dynasearch determina a melhor combinação de vários movimentos numa vizinhança por programação dinâmica (Congram et al. 2002). Ela pode ser vista como uma busca local com estratégia "melhor melhora" intensificada. A aplicação é limitada para movimentos *independentes*: cada movimento precisa ser aplicável independente dos outros, e contribui linearmente para a função objetivo. Numa representação por variáveis $(x_1, \ldots, x_n)$ seja δ_{ij} a redução da função objetivo aplicando um movimento nas variáveis $x_i, \ldots, x_j$. Logo a maior redução da função objetivo Δ_j por uma combinação de movimentos independentes aplicado a $x_1, \ldots, x_j$ é dado pela recorrência

$$\Delta_j = \max\{\Delta_{j-1}, \max_{1 \leq i \leq j} \Delta_{i-1} + \delta_{ij}\}$$

e a melhor combinação de movimentos reduz a função objetivo por Δ_n.

Exemplo 5.5 (Dynasearch para o PCV)
Para aplicar dynasearch no PCV supõe uma representação por variáveis com $I = \{\pi_i \mid i \in [n]\}$ e valores em $[n]$ que representa uma permutação das cidades. Um movimento 2-exchange entre arestas (π_i, π_{i+1}) e (π_j, π_{j+1}) com $i < j$ é válido caso $i + 1 < j$, i.e. precisa pelo menos quatro vértices. (Todos índices

são modulo n.) Dois movimentos (i, j) e (i', j') com $i < i'$ são independentes caso $j < i$. A redução da função objetivo para um movimento (i, j) é $\delta_{ij} = -d_{ij} - d_{i+1,j+1} + d_{i,i+1} + d_{j,j+1}$. Logo obtemos a recorrência

$$\Delta_j = \begin{cases} 0 & \text{caso } j < 4 \\ \max\{\Delta_{j-1}, \max_{1 \le i \le j-3} \Delta_{i-1} + \delta_{ij}\} & \text{caso contrário.} \end{cases}$$

◊

5.2. Híper-heurísticas

Híper-heurísticas usam ou combinam heurísticas com o objetivo de produzir uma heurística melhor e mais geral (Denzinger et al. 1997; Cowling et al. 2000). A heurísticas podem ser geradas antes da sua aplicação ("offline"), por uma busca no espaço das heurísticas. Uma híper-heurística desse tipo pode ser projetada usando alguma meta-heurística. Importante no projeto é uma representação adequada de uma heurística generalizada para o problema e diversas heurísticas ou heurísticas parametrizadas que instanciam a heurística generalizada. As operações correspondentes modificam, constroem ou recombinam heurísticas. Uma alternativa é aplicar diferentes heurísticas durante a otimização ("online"). Para isso uma híper-heurística precisa decidir qual sub-heurística aplicar quando.

Exemplo 5.6 (Híper-heurística online construtiva)
Considera o empacotamento unidimensional que permite diversas estratégias gulosas para selecionar o próximo item a ser empacotado (na ordem dada ou em ordem não-crescente, no contêiner atual ou no primeiro ou melhor contêiner). Uma híper-heurística pode selecionar a estratégia de acordo com a solução parcial. Um exemplo é Ross et al. (2002): uma solução parcial é representada pelo número de itens, e as percentagens de itens pequenas, médias, grandes e muito grandes e um classificador é treinado para decidir qual de quatro regras candidatas é aplicada. ◊

Exemplo 5.7 (Híper-heurística online por modificação)
Uma híper-heurística pode usar conceitos da busca tabu para a seleção de heurísticas de modificação $H_1, \ldots, H_k$. Associa um valor ν_i com cada heurística H_i. Aplica em cada passo a heurística H_i de maior valor (uma ou mais vezes). Caso ela melhora a solução atual, aumenta ν_i, senão diminui ν_i e declara H_i tabu. ◊

Exemplo 5.8 (Híper-heurística offline)
Fukunaga (2008) apresenta uma abordagem para gerar heurísticas que selecionam uma variável a ser invertida em uma busca local para o problema SAT. A

regra de seleção é representada por uma expressão, que inclui seleções típicas de algoritmos conhecidos como a restrição para cláusulas falsas, a seleção pelo aumento da função objetivo, uma seleção pelo tempo da última modificação ou uma seleção randômica. Essas restrições podem ser combinadas por condições. A regra de seleção do WalkSAT, por exemplo, é representada por

```
(IF-VAR-COND = +NEG-GAIN+ 0
  (GET-VAR +BC0 +NEG-GAIN+)
  (IF-RAND-LTE 0.5
    (GET-VAR +BC0+ +NEG-GAIN+)
    (VAR-RANDOM +BC0+)
  )
)
```

Um algoritmo genético em estado de equilíbrio evolui as regras de seleção. A população inicial consiste de expressões aleatórias restritas por uma gramática que garante que eles selecionam uma variável. O algoritmo seleciona dois pais com uma probabilidade linear no posto na população, e gera 10 filhos. A estratégia de seleção é $(\mu+\lambda)$. A recombinação de pais p_1 e p_2 é "if (condição) then p_1 else p_2" com 10 condições diferentes, p.ex. i) uma seleção randômica com probabilidade $0.1, 0.25, 0.5, 0.75, 0.9$, ii) a variável mais "antiga" entre p_1 e p_2, ou iii) a variável de p_1 caso ela não invalida nenhuma cláusula, senão p_2. Como a recombinação aumenta a profundidade das expressões, uma regra substitui sub-arvóres de altura dois que ultrapassam um limite de profundidade por uma expressão de menor altura. Isso serve também como mutação das expressões. Cada regra é avaliada em até 200 instâncias com 50 variáveis e caso pelo menos 130 execuções tiveram sucesso em mais 400 instâncias com 100 variáveis e recebe uma valor $s_{50} + 5s_{100} + 1/\bar{f}$ com s_i o número de sucessos em instâncias com i variáveis e $\bar{f}$ o número médio de inversões de variáveis em instâncias com sucesso. As heurísticas evoluídas em uma população de 1000 indivíduos, limitado por 5500 avaliações, com limite de profundidade entre 2 e 6 são competitivas com heurísticas criadas manualmente. ♢

5.3. Heurísticas paralelas

Heurísticas podem ser aceleradas por paralelização. A *granularidade* do paralelismo (a relação entre o tempo de computação e comunicação) é importante para obter uma boa aceleração e tipicamente define ou limita a escolha da arquitetura paralela. A paralelização mais básica executa diversas heurísticas (ou a mesma heurística randomizada) em paralelo e retorna a melhor solução

encontrada. Essa estratégia corresponde com repetições independentes, possui uma granularidade alta, tem a vantagem de ser simples de realizar, e gera uma aceleração razoável. Uma variante é uma decomposição do espaço de busca em subespaços.

Exemplo 5.9 (Aceleração de heurísticas de busca)
Supõe um problema de busca com uma função de probabilidade exponencial $\lambda e^{-\lambda t}$ de encontrar uma solução no intervalo $[t, t + dt]$. A distribuição do mínimo de p variáveis distribuídas exponencialmente com $\lambda_1, \ldots, \lambda_k$ é distribuído exponencial com parâmetro $\lambda = \sum_i \lambda_i$. Logo, para p repetições paralelas independentes, obtemos uma nova distribuição exponencial do tempo de sucesso com parâmetro $p\lambda$. O valor esperado de uma distribuição exponencial é λ^{-1}, e assim obtemos uma aceleração esperada de $\lambda^{-1}/(p\lambda)^{-1} = p$. ◊

As três técnicas heurísticas principais permitem algoritmos paralelos de granularidade fina ou média:

- Buscas por modificação: a exploração de uma única trajetória é inerentemente sequencial. Uma paralelização de granularidade fina pode avaliar toda vizinhança em paralelo (ou alguns movimentos, e.g. na tempera simulada). A granularidade pode ser aumentado por vizinhanças grandes.

- Busca por construção: similarmente a construção por elementos é sequencial, mas os candidatos podem ser avaliados em paralelo.

- Busca por recombinação: permite uma granularidade média paralelizando os passos de seleção, recombinação e melhora de subconjuntos de soluções sobre subconjuntos de soluções independentes.

Uma busca por modificação ou construção pode ser paralelizado melhor avaliando diversas trajetórias ou construções em paralelo. Esse tipo de paralelização se aplica diretamente em métodos como segue os vencedores e colônias de formigas.
Uma paralelização com granularidade fina ou média é mais adequada para arquiteturas com memoria compartilhada. Eles podem ser realizadas de forma conveniente com múltiplos threads (explicitamente ou com abordagens semi-automáticos usando diretivas como OpenMP).

Exemplo 5.10 (GSAT paralelo em C++ com OpenMP)
Uma versão simplificada de uma busca "melhor melhora" para o problema SAT (ver exercícios) pode ser paralelizada em OpenMP por

```
#pragma omp parallel shared(bestvalue,bestj)
    private(t_bestvalue,t_bestj)
{
#pragma omp for private(value)
  for(unsigned j=1; j<=I.n; j++) {
    int value = S.flipvalue(j);
    if (value>t_bestvalue) {
      t_bestvalue = value;
      t_bestj = j;
    }
  }
#pragma omp critical
  {
    if (t_bestvalue < bestvalue) {
      bestvalue = t_bestvalue;
      bestj = t_bestj;
    }
  }
}
```

◊

Modelos cooperativos Uma estratégia de granularidade média são modelos cooperativos: a mesma ou diferentes heurísticas ("agentes") que executam em paralelo trocam tempo a tempo informações sobre os resultados da busca. O projeto de uma estratégia inclui a definição

- de uma topologia de comunicação, que define quais agentes trocam informações. Exemplos de topologias são grades (de diferentes dimensões, abertas ou fechadas), estrelas, ou grafos completos.
- da informação trocada. Exemplos incluem incumbentes, memorias de frequência, ou sub-populações.
- de uma estratégia de incluir a informação no recipiente, por exemplo substituindo um parte da população ou combinar memorias de frequência.
- da frequência em que a informação é trocada.

Um exemplo simples de modelos cooperativos é um *conjunto elite compartilhado*, que pode ser implementado de forma mais simples por um esquema de mestre-escravo.

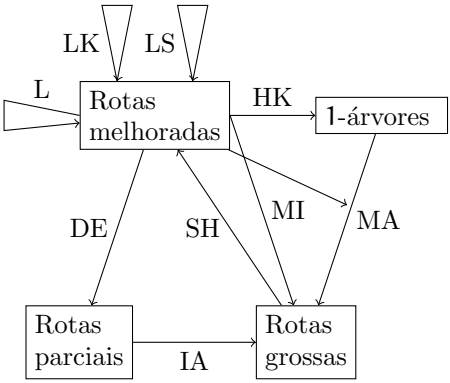


Figura 5.1.: Exemplo de times assíncronos para o PCV (Souza e Talukdar 1993).

Exemplo 5.11 (Colaboração indireta: times assíncronos)
Uma extensão da ideia do conjunto elite compartilhado são *times assíncronos*: uma coleção de diferentes algoritmos (de construção, melhoras, ou recombinação) (chamados de agentes) conectadas por memorias. Cada agente trabalha de forma autônoma e insere, no caso de heurísticas construtivas, ou extrai, modifica e retorna, no caso de heurísticas de melhora ou recombinação, soluções das memorias.

Souza e Talukdar (1993) apresentam um time assíncrono para o PCV com nove agentes: *inserção arbitrária* (IA) completa uma rota parcial por inserção de uma cidade aleatória não-visitada no melhor ponto; *shift* (SH) testa todos deslocamentos de até três cidades consecutivas; *Lin-Kernighan* (LK) aplica o algoritmo do mesmo nome; *Lin-Kernigham simples* (LS) aplica Lin-Kernighan mas termina na primeira melhora encontrada; *misturador* (MI) tenta criar uma nova rota com as arestas de duas rotas (eventualmente completada por demais arestas); *Held-Karp* aplica o algoritmo do mesmo nome para obter um limite inferior e 1-árvores (uma árvore mais um vértice conectado a ela via duas arestas); *misturador de árvores* (MA) mistura uma rota e uma 1-árvore para gerar uma nova rota; *destruidor* (DE) quebra rotas em segmentos, dados pela interseção de duas rotas; *limitador* (L) remove rotas piores ou aleatórias (com uma seleção linear de acordo co a distância, tal que a rota melhor nuca é removida) para limitar o número de rotas. Os agentes são conectados de acordo com a figura 5.1.

◊

Exemplo 5.12 (Algoritmos genéticos no modelo de ilhas)
A metáfora evolutiva naturalmente sugere uma abordagem distribuída em algoritmos genéticos: *populações panmíticas* em quais todos indivíduos da mesma espécia podem ser recombinadas são raras. O *modelo de ilhas* propõe populações com uma evolução independente e uma troca infrequente de indivíduos entre as ilhas.
Luque e Alba (2011) discutem um algoritmo genético distribuído para MAX-SAT com $800/p$ indivíduos em cada um dos p processadores, recombinação em um ponto com probabilidade 0.7 e mutação 1-flip com probabilidade 0.2. Os processadores forma um anel direcionado e cada 20 iterações uma população manda um individuo aleatória para o seu vizinho que incorpora-o caso o valor da função objetivo está maior que a pior indivíduo da população. Numa instância com 100 variáveis e 430 cláusulas eles observam uma aceleração de 1.93, 3.66, 7.41, e 14.7 para $p = 2, 4, 8, 16$ em média sobre 100 replicações. ◊

5.4. Heurísticas para problemas multi-objetivos

Um problema multi-objetivo possui mais que uma função objetivo. O valor de uma solução $\varphi(s) = (\varphi_1(s), \ldots, \varphi_k(s))^t \in \mathbb{R}^k$ *domina* um outro valor $\varphi(s')$ caso $\varphi(s) < \varphi(s')$ (com $<$ tal que existe pelo menos uma componente estritamente menor). Uma solução s cujo valor não é dominado pelo de valor de uma outra solução é *eficiente* (ou *Pareto-ótima*). Diferente da otimização mono-objetivo podem existir valores incomparáveis (e.g. $(1, 2)$ e $(2, 1)$). Tais soluções formam a *fronteira Pareto* (ver fig. 5.3), e um algoritmo multi-objetivo geralmente mantém uma população de soluções não-dominadas. Limites para soluções não-dominadas são o *ponto ideal*

$$\iota = (\min_s \varphi_1(s), \ldots, \min_s \varphi_n(s))$$

dos mínimos em cada dimensão, e o *nadir*

$$\nu = (\max_{s|s \text{ eficiente}} \varphi_1(s), \ldots, \max_{s|s \text{ eficiente}} \varphi_n(s))$$

dos máximos das soluções eficientes em cada dimensão. Um valor $\upsilon \leq \iota$ que domina o valor ideal é *utópico*.
Em problemas difíceis as funções objetivos tendem a ser antagonísticas, i.e., a redução do valor de uma função geralmente aumenta o valor de uma ou mais das outras. Frequentemente um problema multi-objetivo é resolvido por *escalarização*, usando uma função mono-objetivo ponderada $\omega(s) = \sum_i w_i \varphi_i(s)$. Isso geralmente produz somente um subconjunto das soluções eficientes (ver fig. 5.3). Além disso, o conjunto de soluções *suportadas* que podem ser obtidas por otimizar $\omega(s)$ para algum conjunto de pesos w, não inclui todas

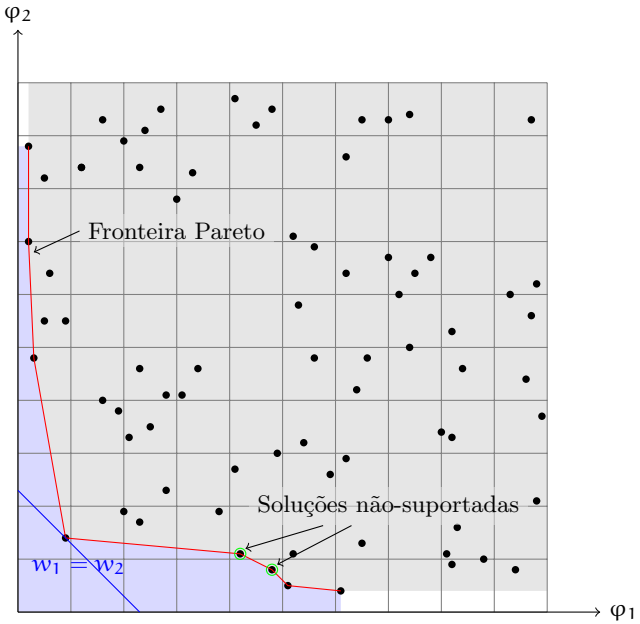


Figura 5.2.: Soluções de um problema com duas funções objetivo. Fronteira eficiente em vermelho. A solução ótima ponderada com pesos $w_1 = w_2$ em azul. Duas soluções eficientes não-suportadas marcadas em verde.

soluções, i.e. existem soluções *não-suportadas* que para nenhuma escolha de w são mínimos de $\omega(s)$.

Exemplo 5.13 (Problema da mochila bi-objetivo)
O problema da mochila bi-objetivo (leia: a versão de decisão correspondente)

$$\begin{aligned} \textbf{maximiza} \quad & cx \\ \textbf{maximiza} \quad & dx \\ \textbf{sujeito a} \quad & wx \leq W \\ & x \in \mathbb{B}^n \end{aligned}$$

é NP-completo por generalizar o problema da mochila. ◊

Claramente uma variante multi-objetivo de um problema é mais difícil que a versão mono-objetiva.

Exemplo 5.14 (Caminhos mais curtos)
Determinar o caminho mais curto entre dois vértices num grafo direcionado conhecidamente permite um algoritmo polinomial (e.g. Dijkstra). A versão (de decisão) bi-objetiva é NP-completo (Serafini 1986): para um problema de mochila $\max\{cx \mid wx \leq W\}$ considera um grafo com vértices $[0, n]$ e arestas $(c_i, 0)$ e $(0, w_i)$ entre $i-1$ e i. O problema da mochila possui uma solução com $cx \geq C$ e $wx \leq W$ sse existe um caminho de 0 para n com distâncias no máximo $\sum_{i\in[n]} c_i - C$ e W. ◊

Avaliação de algoritmos multi-objetivos A comparação de algoritmos multi-objetivos precisa comparar aproximações $\hat{E}$ da fronteira eficiente real E. Caso E é conhecido, uma medida simples é a fração das soluções eficientes encontradas $|\hat{E} \cap E|/|E|$. Porém, isso não conta soluções que são razoavelmente pertas de soluções eficientes. Uma segunda medida aproveita que todas soluções eficientes são soluções suportadas, ou caiem num subespaço "triangular" (ver figura 5.3) de soluções suportadas e mede a fração das soluções em $\hat{E}$ que pertencem a esse espaço. Outros exemplos de medidas de qualidade incluem a distância mínima média para uma solução eficiente

$$\overline{d}(\hat{E}, E) = \sum_{s\in E} \min_{\hat{s}\in\hat{E}} d(s, \hat{s})/|E|$$

e a distância mínima máxima

$$\overline{d}_{\max}(\hat{E}, E) = \max_{s\in E} \min_{\hat{s}\in\hat{E}} d(s, \hat{s})$$

ou medidas baseados no volume coberto. Caso E é desconhecido, uma avaliação aproximada pode ser obtida usando o conjunto de soluções suportadas nas medidas acima. No momento não há consenso sobre a comparação ideal de dois algoritmos multi-objetivos.

5.4.1. Busca por modificação de soluções

Tempera simulada Para aplicar a tempera simulada no caso multi-objetivo, o critério de Metropolis (2.3) precisa ser modificado para comparar valores vetoriais. Uma forma comum é a *escalarização local*: para pesos w a qualidade da nova solução é avaliada pela diferença ponderada das funções objetivos ou das probabilidades (Ulungu et al. 1999). Por exemplo, com $\Delta_w(s, s') = \omega(s') - \omega(s)$ obtemos o critério de Metropolis modificado

$$P[\text{aceitar}] = \begin{cases} 1 & \text{caso } \Delta_w(s,s') \leq 0 \\ e^{-\Delta_w(s,s')/kT} & \text{caso contrário} \end{cases}. \tag{5.4}$$

O algoritmo mantem um conjunto de soluções eficientes durante a busca. Ele aceita uma nova solução caso nenhuma outra solução eficiente dominá-la e aplica critério (5.4) nos outros casos. A tempera simulada é repetida com vários pesos w aleatórios.
Um outro exemplo de um critério de aceitação, proposto por Suppapitnarm et al. (2000), usa um vetor de temperaturas $T \in \mathbb{R}^n$. Com $\Delta_T(s, s') = \sum_{i\in[n]}(s'_i - s'_i)/T_i$ uma solução é aceita com probabilidade

$$\begin{cases} 1 & \text{caso } \Delta_T(s,s') \leq 0 \\ e^{-\Delta_T(s,s')} & \text{caso contrário} \end{cases}$$

Isso é uma variante do critério (5.4) com pesos $w_i = kTT_i^{-1}$ variáveis.

Exemplo 5.15 (MOSA para o problema da mochila bi-objetivo)
O algoritmo descrito acima aplicando o critério (5.4) é conhecido por MOSA (multi-objective simulated annealing). Ulungu et al. (1999) aplicam MOSA no problema da mochila bi-objetivo em comparação com uma solução exata. As instâncias são geradas aleatoriamente com pesos e valores de n itens em $[1, 1000]$ e uma capacidade $W = \sum_{i\in[n]} w_i/r$ com $r \in (0, 1)$. O algoritmo usa uma probabilidade de aceitação inicial de $P_0 = 0.5$, $\alpha = 1-1/40$, $L = \{5, 15, 25\}$ conjuntos de pesos, e $100, 300, 500$ passos por temperatura. A vizinhança remove aleatoriamente itens até todos itens não selecionados cabem na mochila e depois adiciona itens aleatórias até nenhum item cabe mais. ◊

Busca tabu Uma busca tabu multi-objetivo tem que definir a "melhor" solução vizinha. O algoritmo MOTS de Gandibleux et al. (1997) usa a escalarização de Steuer (1986)

$$S(s') = \|\lambda \circ (\upsilon - \varphi(s'))\|_\infty + \rho \|\lambda \circ (\upsilon - \varphi(s'))|_1$$

para selecionar o vizinho não tabu de menor valor S. O valor de um vizinho s' depende um ponto utópico local υ (i.e. um ponto que domina o ponto ideal da vizinhança $N(s)$), um conjunto de pesos λ que define a direção da busca (com $\sum_{i\in[n]} \lambda_i = 1$) e um parâmetro $\rho \ll 1$[1].

Exemplo 5.16 (MOTS para o problema da mochila bi-objetivo)
O algoritmo determina inicialmente limites $[l, u]$ para o número de itens. Na forma mais simples ele busca soluções eficientes com um número de itens $n = u, u-1, \ldots, l$, numa vizinhança que troca um item selecionado x_i por um item não selecionado x_j. A reinserção do item i fica tabu para 7 iterações e a deleção do item j para 3 iterações.
Em cada iteração o algoritmo determina todos vizinhos viáveis não tabu V, que dominam um *ponto de satisfação* σ e não são dominados por uma solução na fronteira eficiente atual $\hat{E}$, e atualiza $\hat{E}$ com estes pontos. O ponto de satisfação σ é 0 para $n = u$ e se aproxima ao nadir η do conjunto eficiente $\hat{E}$ do n anterior de acordo com $\sigma_{n-1} = \sigma_n + (\eta_n - \sigma_n)/\delta$ com um tamanho de passo $\delta \geq 2$. Depois a solução vizinha s' de maior $S(s')$ é selecionada. Caso não existe solução viável que não é tabu, o algoritmo passa para a solução não-tabu que excede a capacidade da mochila menos possível. Um critério de aspiração permite selecionar uma solução tabu que domina todas soluções V ou que domina um número grande de soluções em $\hat{E}$.
A solução inicial é aleatória (com $n = u$ itens selecionados) e cada direção de busca continua com a solução final anterior. Diminuindo n, o item com o menor valor mínimo dos sobre as dimensões da mochila é removido.
A implementação testa 25 conjuntos de pesos $(\lambda, 1-\lambda)$, com $\lambda = i/24$ para $i = 0, \ldots, 24$, aplica no máximo 500 iterações por busca tabu (para cada conjunto de pesos e cada n), e usa $\delta = 2$ na mesmas instâncias do exemplo anterior. A busca para com $n = l$ ou caso na vizinhança não tem solução que domina o ponto de satisfação. ◊

5.4.2. Busca por recombinação de soluções

A maioria das propostas de heurísticas multi-objetivos recombinando soluções são algoritmos genéticos e evolutivos. Num algoritmo genético somente a

[1] A operação $\circ$ é a multiplicação ponto a ponto de dois vetores.

seleção de indivíduos para recombinação depende da função objetivo. Portanto, uma das modificações que torna um algoritmo genético multi-objetivo, é uma seleção proporcional com $\omega(s)$, com um vetor de pesos w selecionado aleatoriamente em cada iteração (Murata et al. 1996). Essa abordagem é simples na implementação, mas tem a desvantagem que ela foca em soluções suportadas. Um dos algoritmos pioneiros trabalho com k subpopulações, e seleciona indivíduos em cada subpopulação de acordo com a i-ésima função objetivo (ver algoritmo 5.1).

Algoritmo 5.1 (Seleção VEGA (Vector-evaluated GA))
Entrada A população atual P.

Saída Uma nova população P.

```
1  para i ∈ [k]
2    seleciona |P|/k indivíduos proporcional com φ_i
3  aplica recombinação e mutação
4    na união S dos indivíduos selecionados
5  retorne a nova população
```

Algoritmos recentes determinam o valor de uma solução de acordo com a proximidade com a fronteira eficiente e a densidade na fronteira eficiente, para uma exploração melhor em direção de soluções eficientes e em regiões esparsas. Para um conjunto de soluções S seja $\hat{E}(S) = \hat{E}_1(S)$ a fronteira eficiente (local) e define recursivamente a $k+1$-ésima fronteira eficiente por

$$\hat{E}_{k+1}(S) = \hat{E}\big(S \setminus \bigcup_{i \in [k]} \hat{E}_k(S)\big). \tag{5.5}$$

(ver o exemplo da Fig. 5.3).
Seja ainda $B(x, S) = \{s \in S \mid x > s\}$ o conjunto de soluções em S que dominam x e $W(x, S) = \{s \in S \mid x > s\}$ o conjunto de soluções dominadas por x em S. Entre as propostas temos algoritmos que ordenam soluções $s \in P$ da população atual P

- pelo nível k da sua fronteira eficiente $s \in \hat{E}_k(P)$ correspondente (non-dominated sorting GA, NSGA, NSGA-II);
- pelo número $1 + |B(s, P)|$ de soluções que dominam s na população atual P (MOGA);
- pela fração total da cobertura por soluções de um conjunto E eficiente atual $1 + \sum_{t \in B(s,E)} |W(t, P)|/(|P| + 1)$ que dominam s (strength Pareto EA, SPEA);

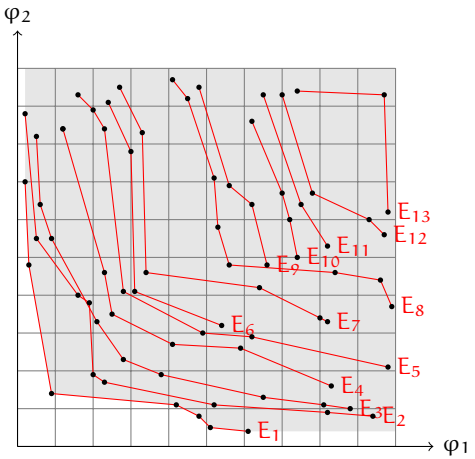


Figura 5.3.: Decomposição de um conjunto de soluções em fronteiras eficientes de acordo com (5.5).

- pelo soma dos postos das soluções que dominam s, $r(s) = 1 + \sum_{t \in B(s,P)} r(t)$.

Técnicas para priorizar a exploração de regiões esparsas incluem

- a redução da função objetivo por um fator $|B_\sigma(s) \cap \hat{\varphi}(P)|^{-1}$ (com $B_r(s)$ um esfera de raio r e centro $\hat{\varphi}(s)$ e $\hat{\varphi}(s)$ a função objetivo normalizada para o intervalo $[0,1]$ em cada dimensão) (MOGA);

- a soma das distâncias normalizadas para os predecessores e sucessores na fronteira atual em cada dimensão ("crowding distance") (NSGA-II). Para cada dimensão $i \in [k]$ supõe que as soluções $x^1, \ldots, x^n$ de uma fronteira são ordenadas pela i-ésima coordenada (i.e. $x_i^1 \leq x_i^2 \leq \cdots \leq x_i^n$). Então o crowding distance normalizada da solução x^s na dimensão i é

$$c_i(x^s) = (\varphi_i(x^{s-1}) - \varphi_i(x^{s+1}))/(\varphi_i^{\max} - \varphi_i^{\min})$$

  para $s \in [2, n-1]$, $c_i(x^1) = c_i(x^n) = \infty$ e a crowding distance da solução é $c(x^s) = \sum_{i \in [k]} c_i(x^s)$.

Formas de elitismo incluem manter uma ou mais fronteiras eficiente $\hat{E}_k(P)$ ou $\hat{E}_k(P \cup C)$ com filhos C.

Exemplo 5.17 (NSGA-II)
O algoritmo NSGA-II segue o algoritmo genético 4.3 com uma seleção por um torneio binário de $\mathcal{P}$: entre duas soluções aleatórias a solução de menor nível k ou, no caso de empate, de menor "crowding distance" é selecionada. Ele sempre aplica mutação ($\mathcal{M} = C$). A função `update` que atualiza a população é realizada por

```
1  R := P ∪ C
2  seja P := Ê_1(R) ∪ ··· ∪ Ê_k(R) com k maximal t.q. |P| ≤ n
3  if |P| < n
4    complete P com as n − |P| soluções de Ê_{k+1}(R)
5      de menor ''crowding distance''
6  end if
```

◊

5.5. Heurísticas para problemas contínuas

Uma forma geral de um problema de otimização contínuo é

$$\begin{array}{lll} \textbf{minimiza} & f(x) & \\ \textbf{sujeito a} & g_i(x) \leq 0 & \forall i \in [m] \\ & h_j(x) = 0 & \forall j \in [l], \end{array}$$

com soluções $x \in \mathbb{R}^n$, uma função objetivo $f : \mathbb{R}^n \to \mathbb{R}$, e restrições $g_i : \mathbb{R}^n \to \mathbb{R}$ e $h_j : \mathbb{R}^n \to \mathbb{R}$. Casos particulares importantes incluem funções lineares e convexas e o caso irrestrito ($m = l = 0$). As definições 2.1 continuam válidas com uma vizinhança $N_\epsilon(x) = \{x' \in \mathbb{R}^n \mid \|x - x'\| \leq \epsilon\}$ e com a condição adicional para um mínimos ou máximos locais deve existir um $\epsilon > 0$ que satisfaz a definição.
Casos simples de um problema de otimização contínua podem ser resolvidos por métodos indiretos. Um método indireto encontra primeiramente todos candidatos para soluções ótimas por critérios necessários para otimalidade local, depois verifica a otimalidade local por critérios suficientes, e finalmente encontra a solução ótima global por comparação dos soluções ótimas locais. Na otimização irrestrita em uma dimensão, por exemplo, temos a condição suficiente $f' = 0$ para otimalidade local, e a condição suficiente $f'' > 0$ para um mínimo local e $f'' < 0$ para um máximo local (dado que as derivadas existem).
Caso resolver $f' = 0$ não é possível técnicas de *busca numa linha* (ingl. *line search*) podem ser usadas. Para um domínio restrito $x \in [a, b]$ um método simples é a *busca regular*: escolhe o melhor entre os pontos $x = a + i\Delta x$, para

$i = 0, \ldots, \lfloor (b - a)/\Delta x \rfloor$, para um tamanho de passo Δx. Um outro exemplo é uma *busca em linha com backtracking*.

Algoritmo 5.2 (Busca em linha com backtracking)
Entrada Um ponto x, uma direção de descida Δx, $\alpha \in (0, 0.5)$, $\beta \in (0, 1)$.

Saída Uma nova solução x.

```
1  t := 1
2  while f(x + tΔx) > f(x) + αtf'(x)Δx do t := βt
3  return x + tΔx
```

O algoritmo precisa uma direção de descida Δx, tal que $f'(x)\Delta x < 0$, por exemplo $\Delta x = -f'(x)$. O parâmetro α define uma perda em qualidade aceitável, o parâmetro β a precisão da busca. A busca termina, porque para um t suficientemente pequeno a condição é satisfeita localmente.
Os dois métodos podem ser generalizadas para o caso irrestrito no $\mathbb{R}^n$. A busca regular limitada para $S = \{x \in \mathbb{R}^n \mid l \leq x \leq u\}$ para um limitante inferior $l \in \mathbb{R}^n$ e superior $u \in \mathbb{R}^n$ avalia todos pontos $x = l + i \circ \Delta x \in S$, com $i \in \mathbb{Z}_+$ para um tamanho de passo $\Delta x \in \mathbb{R}^n$. A busca em linha com backtracking substitui a derivada $f'(x)$ pelo gradiente $\nabla f(x)$; uma direção de busca então é $\Delta x = -\nabla f(x)$.
Métodos de busca em linha são elementos de *métodos univariados* de otimização, que otimizam uma variável por vez, ou mais geral, uma direção de busca por vez. A *busca por relaxação de Southwell* por exemplo repetidamente seleciona a variável x_i que corresponde com o maior valor absoluto do gradiente $|\partial f/\partial x_i|(x)$. Um dos métodos mais comuns é a *descida do gradiente* (ingl. *gradient descent*).

Algoritmo 5.3 (Descida do gradiente)
Entrada Um ponto inicial $x \in \mathbb{R}^n$.

Saída Uma nova solução $x \in \mathbb{R}^n$.

```
1  repeat
2    Δx := −∇f(x)
3    aplica uma busca em linha na direção Δx
4      para obter um tamanho de passo t
5    x := x + tΔx
6  until critério de parada satisfeito
7  return x
```

Um critério de parada comum é $\|\nabla f(x)\|_2 \leq \epsilon$, para um $\epsilon > 0$ pequeno.

Exemplo 5.18 (Redes neurais artificias)
Uma grande classe de *redes neurais artificias* são *redes sem realimentação* (ingl. *feed forward networks*). Eles recebem informação numa *camada de entrada*, que passa por múltiplas *camada internas* até chegar na *camada de saída*. A saída x de um elemento de uma camada é uma função da soma ponderada dos elementos $x'_1, \ldots, x'_n$ da camada anterior:

$$x = g\Big(\sum_{i\in[n]} w_i x'_i\Big). \tag{5.6}$$

A função g é a *função de ativação*. (O modelo simples de um neurônio de McCulloch e Pitts (1943) usa $g(x) = [x > 0]$.) Ela tipicamente é sigmoide (possui forma de "s"), por exemplo

$$g(x) = \frac{1}{1 + \exp(-2\beta h)}$$

com derivada $g' = 2\beta g(1 - g)$. Em geral supõe que temos uma rede com k camadas e a camada i possui n_i elementos. Sejam $W^1, \ldots, W^{k-1}$ as matrizes de pesos entre as camadas, com $W^i \in \mathbb{R}^{n_{i+1}\times n_i}$. Logo uma entrada $x^1 \in \mathbb{R}^{n_1}$ na primeira camada é propagada para frente por

$$h^{i+1} = W^i x^i; \qquad x^{i+1} = g(h^i) \tag{5.7}$$

para $i \in [k-1]$. O valor h^i é a entrada da camada i, o valor $x^i \in \mathbb{R}^{n_i}$ a sua saída. (A função g é aplicada em cada componente.)
O objetivo de uma rede neural artificial é treiná-lo para produzir saídas desejadas (e espera-se que a rede generaliza e produz resultados desejáveis para entradas desconhecidas). Na *aprendizagem supervisionada* a rede repetidamente recebe uma entrada $x^1 = \xi$ e a saída x^k é comparada com uma saída desejada σ. O *erro* é definido por

$$E(W^1, \ldots, W^k) = 1/2 \sum_{i\in[n_k]} (\sigma_i - x_i^k)^2.$$

O treinamento consiste em ajustar o pesos $W^1, \ldots, W^k$ tal que E é minimizado. Isso é um problema de otimização contínua, e nos podemos aplicar a descida de gradiente para obter pesos melhores. No caso de uma rede com somente uma camada interna ($k = 3$) temos

$$E(W^1, W^2) = 1/2 \sum_{k\in[n_3]} \Big(\sigma_k - g\Big(\sum_{j\in[n_2]} W^2_{kj}\, g\Big(\sum_{i\in[n_1]} W^1_{ji} x^1_i\Big)\Big)\Big)^2.$$

e o gradiente para os pesos entre a segunda e a terceira camada é

$$\begin{aligned}\frac{\partial E}{\partial W^2_{kj}} &= -(\sigma_k - x^3_k) g'(h^3_k) x^2_j \\ &= -\delta^2_k x^2_j\end{aligned}$$

com $\delta^2_k = g'(h^3_k)(\sigma_k - x^3_k)$. Similarmente o gradiente para os pesos entre a primeira e a segunda camada é

$$\begin{aligned}\frac{\partial E}{\partial W^1_{ji}} &= -\sum_{k\in[n_3]} (\sigma_k - x^3_k) g'(h^3_k) W^2_{kj} g'(h^2_j) x^1_i \\ &= -\sum_{k\in[n_3]} \delta^2_k W^2_{kj} g'(h^2_j) x^1_i \\ &= -\delta^1_j x^1_i.\end{aligned}$$

com $\delta^1_j = g'(h^2_j) \sum_{k\in[n_3]} \delta^2_k W^2_{kj}$.
Aplicando a descida do gradiente com um tamanho de passo η obtemos a regra simples

$$\Delta W^i_{kj} = -\eta \frac{\partial E}{\partial W^i_{kj}} = \eta \delta^i_k x^i_j \tag{5.8}$$

com

$$\begin{aligned}\delta^2 &= g'(h^3) \circ (\sigma - x^3) \\ \delta^1 &= g'(h^2) \circ \delta^2 W^2.\end{aligned}$$

Isso pode ser generalizado para um número arbitrário de camadas por

$$\begin{aligned}\delta^k &= g'(h^k) \circ (\sigma - x^k) \\ \delta^i &= g'(h^{i+1}) \circ \delta^{i+1} W^{i+1}, \qquad i \in [k-2].\end{aligned} \tag{5.9}$$

Logo enquanto os valores são propagadas para frente, de acordo com (5.7), os erros são propagadas para atrás por (5.9) e o método é chamada *propagação para atrás* (ingl. *backpropagation*).
Para treinar uma rede serve um conjunto de entradas $\xi^1, \ldots, \xi^m$ com saídas desejadas $\sigma^1, \ldots, \sigma^m$. Repetidamente para entrada ξ^i a saída é calculada por propagação para frente, os erros δ são calculados por propagação para atrás e os pesos são ajustados pela regra (5.8).

$\Diamond$

5.5.1. Meta-heurísticas para otimização contínua

A otimização com enxames de partículas da seção 4.6 é um exemplo de uma meta-heurística que pode ser aplicado diretamente na otimização contínua. De fato a maioria das heurísticas por modificação ou recombinação podem ser aplicadas para problemas contínuas com uma definição adequada de uma vizinhança e de uma recombinação. Exemplos de vizinhanças contínuas são a vizinhança uniforme $N_\epsilon(x)$ de acima e a vizinhança Gaussiana $N(x) = N(x, \sigma)$. Recombinações da seção 4 que podem ser aplicadas no caso contínuo são as recombinações randomizadas, lineares e particionadas.
Um exemplo que inclui uma estratégia construtiva para otimização contínua é o *GRASP contínuo* (C-GRASP).

Algoritmo 5.4 (C-GRASP)
Entrada Conjunto de soluções viáveis $S = \{x \in \mathbb{R}^n \mid l \leq x \leq u\}$, parâmetros h_0, h_f, ρ e α.

Saída Uma solução $x \in S$.

```
1  repeat
2    x := U[l,u]
3    h := h0
4    repeat
5      x := construct(x,α,h)
6      x := localsearch(x,ρ,h)
7      if x não melhorou
8        h := h/2
9      endif
10   until h < hf
11 until critério de parada satisfeito
12 return x
```

A construção gulosa é univariada, selecionando entre uma das melhores direções de otimização

```
1  construct(x,α,h) :=
2    S := [n]
3    while S ≠ ∅ do
4      for i ∈ S: zi := buscaregular(xi,li,ui,h)
5      C := {i ∈ S | f(zi) ≤ (1−α) min_i zi + α max_i zi}
6      seleciona j ∈ C aleatório
7      xj := zj
```

```
8         S := S \ {j}
9       end while
10    end
```

A vizinhança da busca local projeta todos pontos da grade regular $R(x) = \{x \mid x = l + i \circ \Delta x \in S,\ i \in \mathbb{Z}_+\}$ numa esfera de raio h com centro x

$$B_h(x) = \{x'' \in S \mid x'' = x + h(x' - x)/\|x' - x\|_2, x' \in R(x) \setminus \{x\}\}$$

e repetidamente busca numa direção aleatória em $B_h(x)$.

```
1   localsearch(x,ρ,h) :=
2     repeat
3       seleciona x' ∈ B_h(x) aleatoriamente
4       if f(x') < f(x): x := x'
5     until ρ|R(x)| pontos examinados sem melhora
6     return x
7   end
```

5.6. Notas

O livro do Talbi (2009, ch. 4) contém uma boa introdução em otimização multi-objetivo. Konak et al. (2006) apresentam estratégias para algoritmos genéticos multi-objetivos. Jaszkiewicz e Dąbrowski (2005) é uma biblioteca (já um pouco antiga) com implementações de diversas meta-heurísticas multi-objetivos. Boyd e Vanderberghe (2004) é uma introdução excelente na otimização convexa.

6. Metodologia para o projeto de heurísticas

> Over the last decade and a half, tabu search algorithms for machine scheduling have gained a near-mythical reputation by consistently equaling or establishing state-of-the-art performance levels on a range of academic and real-world problems. Yet, despite these successes, remarkably little research has been devoted to developing an understanding of why tabu search is so effective on this problem class.
>
> (Watson et al. 2006)

> Despite widespread success, very little is known about why local search metaheuristics work so well and under what conditions. This situation is largely due to the fact that researchers typically focus on demonstrating, and not analyzing, algorithm performance. Most local search metaheuristics are developed in an ad hoc manner. A researcher devises a new search strategy or a modification to an existing strategy, typically arrived at via intuition. The algorithm is implemented, and the resulting performance is compared with that of existing algorithms on sets of widely available benchmark problems. If the new algorithm outperforms existing algorithms, the results are published, advancing the state of the art. Unfortunately, most researchers [...] fail to actually prove that the proposed enhancements actually led to the observed performance increase (as typically, multiple new features are introduced simultaneously) or whether the increase was due to fine tuning of the algorithm or associated parameters, implementation tricks, flaws in the comparative methodology, or some other factors.
>
> Gendreau e Potvin (2010)

> The field of optimization is perhaps unique in that natural or man-made processes completely unrelated to optimization can be used as inspiration, but other than that, what has caused the research field to shoot itself in the foot by allowing the wheel to be invented over and over again? Why is the field of metaheuristics so vulnerable to this pull in an unscientific direction? The field has shifted from a situation in which metaheuristics are used as

> inspiration to one in which they are used as justification, a shift that has far-reaching negative consequences on its credibility as a research area.
> [...]
> The field's fetish with novelty is certainly a likely cause.
> [...]
> A second reason for this research to pass is the fact that the research literature in metaheuristics is positively obsessed with playing the up-the-wall game (Burke et al., 2009). There are no rules in this game, just a goal, which is to get higher up the wall (which translates to "obtain better results") than your opponents. Science, however, is not a game. Although some competition between researchers or research groups can certainly stimulate innovation, the ultimate goal of science is to understand. True innovation in metaheuristics research therefore does not come from yet another method that performs better than its competitors, certainly if [it] is not well understood why exactly this method performs well.
>
> Sörensen (2013)

As citações acima caracterizam o estado metodológico do projeto de heurísticas. Por isso, é necessário enfatizar que o projeto de heurísticas é uma disciplina experimental, e tem que seguir o *método científico*. Em particular, o projeto

i) inicia com uma *questão científica* específica, bem definida e clara; ("Qual o melhor método para resolver o PCV?")

ii) gera um ou mais *hipóteses* para responder essa questão; ("Dado o mesmo tempo, Lin-Kernighan iterado sempre é melhor que tempera simulada.")[1]

iii) projeta *testes experimentais* para verificar (estatisticamente) ou rejeitar as *predições* das hipóteses;

iv) analisa os resultados dos experimentos e conclui; isso pode resultar em novas hipóteses.

6.1. Projeto de heurísticas

O objetivo típico do projeto de uma heurística é obter soluções de boa qualidade em tempo adequado. Os critérios são correlacionados, i.e. mais tempo geralmente produz melhores soluções. O tempo disponível depende da aplicação

[1] Observe que isso é uma ilustração: essa hipótese é quase irrefutável, e precisa ser muito mais específica na prática.

e tipicamente influencia a técnica heurística (pensa: 100 metros rasos vs. maratona). Além disso, pode ser o objetivo do projeto obter uma heurística

- *simples*, i.e. fácil de implementar, entender e explicar;
- *robusta*, i.e. simples de calibrar e pouco sensível aos parâmetros;
- *generalizável*, i.e. aplicável a um grande número de problemas similares

(Barr et al. 1995; Cordeau et al. 2002).
De acordo com a nossa classificação, heurísticas usam três operações principais: construção, por adição de elementos, modificação, por alteração de elementos, e recombinação, por selecionar e unir elementos de mais que uma solução. Essas operações são específicas ao problema, junto com a representação e a função objetivo. A literatura sugere que uma meta-heurística efetiva depende dos seguintes componentes, *em ordem da sua importância* (Watson et al. 2006; Hertz et al. 2003):

1. as técnicas específicas ao problema;
2. a meta-heurística; uma meta-heurística básica precisa técnicas para evitar estagnação (mínimos locais);
3. a intensificação e diversificação estratégica usando memoria que beneficia geralmente cada heurística;
4. os parâmetros dos componentes;
5. a implementação eficiente.

Na prática inversões são possíveis, e todos os pontos tem que ser tratados sistematicamente para obter resultados de estado de arte. Por isso sugerimos uma *metodologia construtiva por componentes* para o projeto de heurísticas.

1. Estuda diferentes representações do problema. Projeta uma estrutura de dados adequada com apoia eficiente para as principais operações (adição, deleção, alteração de elementos e avaliação incremental). Determine a complexidade dessas operações. Considera os princípios 1.1 e 1.2.
2. Propõe diferentes operações de construção, modificação e recombinação. Avalia estatisticamente cada uma das operações e o seus parâmetros separadamente. Para modificação considera os princípios 2.1 e 2.2.
3. Considere uma análise da paisagem de otimização (cáp. 6.2).

4. Combina sistematicamente operações básicas para uma meta-heurística básica que evita mínimos locais ou uma meta-heurística construtiva. Especificamente projeta e testa se as técnicas para evitar mínimos locais são efetivas. Avalia a contribuição e a interação dos componentes e o seus parâmetros. Procede das técnicas mais simples para as mais complexas (e.g. busca local, tempera simulada, busca tabu; resp. construção gulosa, bubble search, colônia de formigas).

5. Adiciona uma estratégia de intensificação e diversificação usando uma forma de memoria de longa duração. Procede das técnicas mais simples para as mais complexas (e.g. Probe, GRASP-PR, algoritmo genético/busca dispersa).

Complementarmente o método científico sugere:

1. Compare durante o projeto com o estado de arte em algoritmos exatos, aproximativos, e heurísticos em tempo e qualidade.

2. Procure não simplesmente produzir "melhores" resultados mas explicações do funcionamento do método.

3. Os experimentos tem que ser reproduzíveis por outros pesquisadores. Consequentemente as instâncias, as saídas, as soluções completas obtidas e o código tem que ser publicado (eventualmente em forma "ilegível" mas compilável, caso investimento em desenvolvimento ou propriedade intelectual tem que ser protegido) (Barr et al. 1995).

Complementarmente a literatura sobre solução de problemas sugere (e.g. Polya (1945))

1. Tenta entender o problema profundamente. Resolve algumas instâncias manualmente, testa heurísticas construtivas, de modificação ou recombinação em alguns exemplos pequenos manualmente. Para heurísticas de modificação estuda exemplos de mínimos locais: porque eles são mínimos locais? Com quais operações daria para escapar desses mínimos (princípio 2.2)?

2. Tenta resolver o problema de melhor forma algoritmicamente, mesmo ele sendo NP-completo. Estuda algoritmos aproximativos e exatos para o problema. Usa as técnicas das melhores algoritmos para construir as operações básicas da heurística.

3. Caso problema é NP-completo: estuda a prova da dificuldade cuidadosamente: quais características do problema torna-o difícil? Eles são

comuns em instâncias práticas? Caso contrário, a prova pode ser simplificada? Ou é possível que o problema não é NP-difícil em instâncias práticas? É possível isolar características que simplificam instâncias?

4. Procure identificar o subproblema mais simples que pode ser resolvido. Procure identificar problemas semelhantes e estudar as suas soluções. Procure generalizar o problema. Dá para transformar o problema para um outro problema similar?

Escolha de uma meta-heurística Dado o metodologia acima, uma guia básica para escolha de uma meta-heurística é

- A meta-heurística é menos importante que as operações básicas. Escolhe a meta-heurística mais tarde possível, e somente depois de estudar as operações básicas.
- Seleciona uma meta-heurística que conhecidamente funciona bem em problemas similares.
- Tendencialmente técnicas construtivas são mais adequadas para problemas mais restritos.
- Tendencialmente intensificação é preferível para uma escala de tempo curta; algoritmos estocásticos (e.g. tempera simulada, construção iterada independente) tendem a precisar mais tempo.
- Tendencialmente métodos mais sistemáticos são preferíveis para problemas maiores. Por exemplo, a probabilidade de encontrar soluções de boa qualidade por construção iterada independente tipicamente diminui com o tamanho da instância (Gendreau e Potvin 2010, cap. 20) ("central limit catastrophe").

6.2. Analise de paisagens de otimização

Para estimar a dificuldade de resolver um problema para uma dada vizinhança temos que responder (empiricamente) perguntas como

- Qual a probabilidade de encontrar uma solução ótima *a priori*?
- O quanto a função objetivo varia entre soluções vizinhas?
- Qual a distância média entre dois mínimos locais?
- O quanto a função objetivo guia uma busca local para soluções ótimas?

Essa perguntas geralmente são difíceis para responder, porque eles supõem que já conhecemos as soluções ótimas do problema. Na prática podemos obter estimativas dessa medidas por amostragem.

Distribuição de tipos de soluções Para uma dada vizinhança podemos classificar a soluções como segue. Seja $E(s) = \{s \in N(s) \mid \varphi(s') = \varphi(s)\}$ o conjunto de vizinhos com o mesmo valor da função objetivo, e $W(s) = N(s) \setminus B(s) \setminus E(s)$ o conjunto de vizinhos piores que s. Com isso obtemos a classificação

$\lvert B(s)\rvert$	$\lvert E(s)\rvert$	$\lvert W(s)\rvert$	Tipo de solução
0	0	0	Solução isolada
> 0	0	0	Máximo local estrito
0	> 0	0	Plateau
> 0	> 0	0	Máximo local
0	0	> 0	Mínimo local estrito
> 0	0	> 0	Declive
0	> 0	> 0	Mínimo local
> 0	> 0	> 0	Patamar

Exemplo 6.1 (Permutation flow shop problem)
Obtemos para as $10! = 3.628.800$ soluções da instância "carlier5" do PFSSP na vizinhança N_1 que insere uma tarefa em qualquer outra posição nova:

Tipo de solução	# (%)	Tipo de solução	# (%)
Solução isolada	0 (0)	Mínimo local estrito	5 (0.00014)
Máximo local estrito	0 (0)	Declive	134784 (3.71)
Plateau	0 (0)	Mínimo local	1743 (0.048)
Máximo local	6 (0.00017)	Patamar	3492262 (96.24)

Existem três mínimos globais com valor 7720. Todos três são não-estritos. Logo a probabilidade *a priori* de um mínimo local ser um mínimo global é 0.0017. A distribuição dos 86 valores dos mínimos locais é (mínimo/quartil inferior/mediana/quartil superior/máximo): 7720, 8039, 8047, 8335, 8591. Um busca local na vizinhança N_1 então é no máximo 11.3% acima do valor ótimo. ◊

Variação entre soluções vizinhas Intuitivamente, uma paisagem de otimização "menos contínua" e "mais curvada" é mais difícil para um algoritmo de busca local. Isso pode ser formalizado pela função de correlação da paisagem (ingl. *lands-*

cape correlation function)

$$\rho(i) = \frac{\mathrm{cov}(\varphi(s)\varphi(s'))_{d(s,s')=i}}{\sigma(\varphi)^2} = \frac{\langle \varphi(s)\varphi(s')\rangle_{d(s,s')=i} - \langle\varphi(s)\rangle^2}{\langle\varphi^2(s)\rangle - \langle\varphi(s)\rangle^2}. \quad (6.1)$$

Temos $\rho(i) \in [-1, 1]$: para valores perto de 1 o valor de soluções vizinhas é perto da valor da solução atual; para um valor perto de 0, o valor de uma solução vizinha não é relacionado com o valor da solução atual.

Exemplo 6.2 (Permutation flow shop problem)
No caso do PFSSP obtemos $\rho(1) \approx 0.79$. Logo existe uma alta correlação entre o valor de uma solução e o valor das soluções vizinhas: podemos esperar que uma busca local funciona razoavelmente bem. ◊

A distância média entre dois mínimos locais pode ser estimado pela *distância de correlação* (ingl. *correlation length*) $l = \sum_{i\geq 0} \rho(i)$. Com $B(r)$ o número de soluções numa distância no máximo r de uma solução esperamos que

$$P[s \text{ é ótimo local}] \approx 1/B(l).$$

Essa relação é conhecida como *conjetura da distância de correlação*.
A função de correlação $\rho(i)$ pode ser determinada empiricamente pela autocorrelação de uma caminhada aleatória. Para uma caminhada aleatória $s_1, s_2, \ldots, s_m$ com $m \gg i$ obtemos o estimador

$$\rho(i) = \rho(\varphi(s_{1:m-i}), \varphi(s_{i+1:m})),$$

onde $s_{a:b} = (s_a, \ldots, s_b)$ e $\varphi(s) = (\varphi(s_1), \ldots, \varphi(s_m))$. Essa estimativa é somente correta, caso uma caminhada aleatória é representativa para toda paisagem de otimização. Tais paisagens são chamadas *isotrópicas*. Frequentemente a correlação diminui exponencialmente com a distância de forma $\rho(i) = \rho(1)^i$ e $\rho(1) = e^{-1/l}$. Neste caso, podemos determinar l por

$$l = (-\ln(|\rho(1)|))^{-1}.$$

Para usar uma $\rho(1)$ estimado por um caminho aleatório na conjetura da distância de correlação, ainda temos que corrigir a distância: caso uma caminhada aleatória de i passos resulta numa solução de distância média $d(i)$, a probabilidade de uma solução ser um ótimo local é $\approx 1/B(d(l))$.

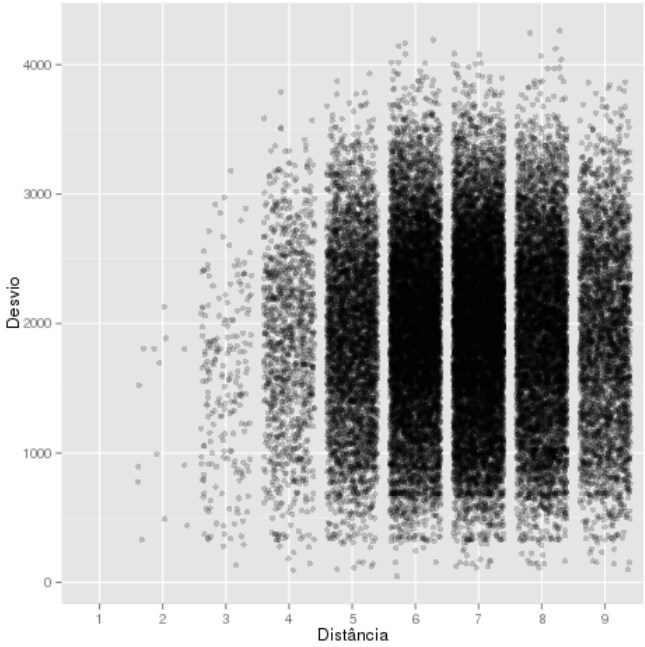
Correlação entre qualidade e distância A função objetivo guia uma busca local para soluções melhores caso a distância $d^*(s)$ para a solução ótima mais

próxima de uma solução s e correlacionada com a valor da função objetivo. A correlação qualidade-distância (ingl. fitness distance correlation)

$$\rho(\varphi, d^*) = \frac{cov(\varphi, d^*)}{\sigma(\varphi)\sigma(d^*)} = \frac{\langle \varphi(s) d^*(s) \rangle - \langle \varphi(s) \rangle \langle d^*(s) \rangle}{\sqrt{\langle \varphi^2(s) \rangle - \langle \varphi(s) \rangle^2} \sqrt{\langle {d^*}^2(s) \rangle - \langle d^*(s) \rangle^2}} \quad (6.2)$$

mede isso. Temos $\rho(\varphi, d^*) \in [-1, 1]$: para valores positivos temos uma estrutura "big valley" com o um extremo de uma correlação linear ideal para um valor de 1; para valores negativos a função objetivo de fato não guia a busca. No primeiro caso intensificação maior, no segundo uma diversificação maior é indicado. A correlação também serve para comparar vizinhanças: muitas vezes a vizinhança que possui uma maior correlação produz resultados melhores.

Exemplo 6.3 (Permutation flow shop problem)
Para a vizinhança "shift" que desloca uma elemento da permutação para qualquer outra posição, obtemos a seguinte distribuição de distância e desvio de uma solução da solução ótima mais perta.



Um $\rho \approx 1.7 \cdot 10^{-5}$ confirma uma fraca correlação entre distância e qualidade. ◊

6.3. Avaliação de heurísticas

Uma heurística, como qualquer algoritmo, transforma determinadas entradas (as instâncias do problema) em saídas ou *resposta* (as soluções viáveis). Essa transformação é influenciada por *fatores* experimentais e pode ser analisado (como qualquer outro processo) com métodos estatísticos adequadas. Os componentes do processo e o seu parâmetros são *fatores controláveis*; além disso o processo sofre fatores incontroláveis (e.g. randomização e as instâncias).
Na avaliação queremos responder perguntas como

- Como os diferentes *níveis* dos fatores controláveis influem a resposta do processo? Quais são os fatores principais? O quanto os fatores influem a resposta? Existe uma interação entre diferentes fatores? Qual escolha de níveis produz resultados bons para uma grande variação dos fatores incontroláveis (i.e. uma heurística *robusta*)?

- Qual o tempo (empírico) para encontrar uma solução viável, de boa qualidade, ou ótima em função do tamanho da instância?

Observação 6.1
Medidas de tempo devem ser acompanhadas por informações detalhadas sobre o ambiente de teste (tipo de processador, memoria, etc.). Uma alternativa é informar o custo computacional em número de operações elementares. ◊

Complexidade empírica de algoritmos A complexidade de tempo de um algoritmo prático com alta probabilidade possui a forma

$$T(n) \sim a b^n n^c \log^d n$$

(ver p.ex. Sedgewick e Wayne (2011, cáp. 1.4) e Sedgewick (2010)). Frequentemente podemos focar em dois casos simples. Para uma série de medidas (n, T) podemos avaliar

uma hipótese exponencial Com $T(n) \sim a b^n$, obtemos $\log T \sim \log a + n \log b$. Logo podemos determinar um modelo por regressão linear entre $\log T$ e n;

uma hipótese polinomial Com $T(n) \sim a n^b$ obtemos $\log T \sim \log a + b \log n$. Logo podemos determinar um modelo por regressão linear entre $\log T$ e $\log n$.

Exemplo 6.4 (Complexidade empírica em GNU R)
Para um arquivo com tamanho da instância n e tempo T da forma

```
n T
100  233.0000
250  689.7667
500 1655.8667
```

podemos determinar a complexidade empírica em GNU R usando

```
d<-read.table("x.dat",header=T)
lm(log(T)~log(n),data=d)
lm(log(T)~n,data=d)
```

◊

Observação 6.2 (Soma de quadrados na regressão linear)
Supõe que temos valores $x \in \mathbb{R}^n$ e m observações $y_i \in \mathbb{R}^m$ para cada $i \in [n]$. A regressão linear determina uma função $\hat{y} = a\hat{x}+b$. Para a soma de quadrados das distâncias dos pontos aproximados $\hat{y}$ e as observações obtemos

$$\begin{aligned}
SS_T &= \sum_{i,j}(y_{ij}-\bar{y})^2 = \sum_{i,j}\left((\bar{y}_i-\bar{y})-(y_{ij}-\bar{y}_i)\right)^2 \\
&= \sum_{i,j}(\bar{y}_i-\bar{y})^2 + 2(\bar{y}_i-\bar{y})(y_{ij}-\bar{y}_i) + (y_{ij}-\bar{y}_i)^2 \\
&= m\sum_i(\bar{y}_i-\bar{y})^2 + 2\sum_i(\bar{y}_i-\bar{y})\underbrace{\sum_j(y_{ij}-\bar{y}_i)}_{n\bar{y}_i-n\bar{y}_i=0!} + \sum_{i,j}(y_{ij}-\bar{y}_i)^2 \\
&= m\sum_i(\bar{y}_i-\bar{y})^2 + + \sum_{i,j}(y_{ij}-\bar{y}_i)^2 \\
&= SS_x + SS_E.
\end{aligned}$$

Isso mostra que podemos decompor a soma de quadrados total SS_T em duas componentes: a soma de quadrados obtida pela variação das médias em cada ponto x da média geral SS_x. Este parte da variação é explicado pela hipótese linear: ele vem da variação da função linear. O segundo termo representa a soma de quadrados obtida pela variação das medidas individuais das médias em cada ponto x. Este parte pode ser atribuído ao erro experimental. Logo a quantidade

$$R^2 = \frac{SS_X}{SS_T} \in [0,1]$$

representa a "fração explicada" da variação dos dados, e serve como medida da qualidade da aproximação linear. Observe que isso é somente possível

aplicando a regressão linear em todos os dados, não nas médias das observações em cada ponto. ◊

Exemplo 6.5 (R^2 em GNU R)
Aplicando a regressão linear nos dados de Rad et al. (2009) obtemos

```
d<-read.table("rad-cpu.dat",header=T)
> lm(log(neht)~log(tasks)+log(machines),data=d)

Call:
lm(formula = log(neht) ~ log(tasks) + log(machines), data = d)

Coefficients:
  (Intercept)     log(tasks)  log(machines)
     -15.0553         1.6194         0.6468

> summary(lm(log(neht)~log(tasks)+log(machines),data=d))

Call:
lm(formula = log(neht) ~ log(tasks) + log(machines), data = d)

Residuals:
     Min       1Q   Median       3Q      Max
-0.46303 -0.20359 -0.05573  0.17781  0.64577

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   -15.0553     0.5960 -25.262 1.15e-09 ***
log(tasks)      1.6194     0.1171  13.830 2.28e-07 ***
log(machines)   0.6468     0.2068   3.128   0.0122 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3767 on 9 degrees of freedom
Multiple R-squared:  0.9657,Adjusted R-squared:  0.9581
F-statistic: 126.7 on 2 and 9 DF,  p-value: 2.562e-07
```

Logo a complexidade empírica do algoritmo NEHT é $T(n) = 289\text{ns}\, n^{1.6} m^{0.6}$ com $R^2 = 0.9657$. ◊

Aplicado à avaliação de uma heurística isso supõe um critério de parada diferente de tempo (e.g. encontrar uma solução em problemas de decisão ou

convergência em problemas de otimização). Essas técnicas podem ser generalizadas para mais que uma variável. Por exemplo, em problemas de grafos com n vértices e m arestas a hipótese $T(n, m) \sim an^b m^c$ gera um modelo linear $\log T \sim \log a + b \log n + c \log m$ e pode ser obtido por regressão linear novamente.

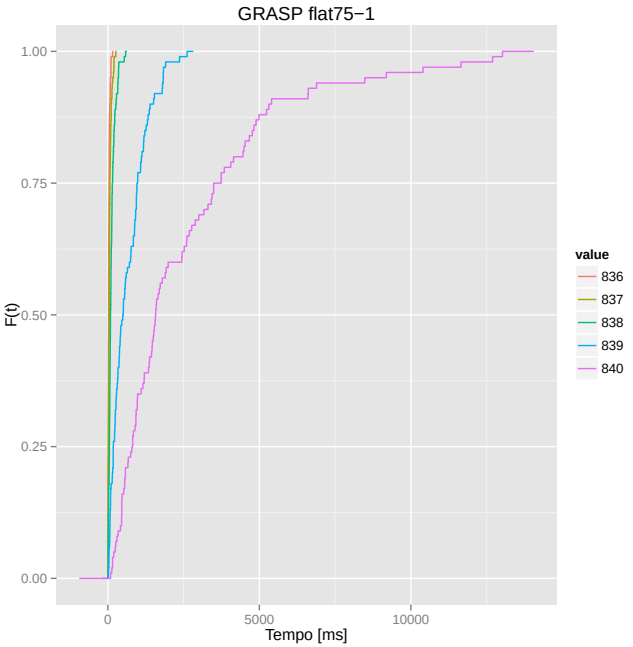
Distribuição de tempo e qualidade Frequentemente a heurística é randomizada e logo o tempo de execução T e a valor V são variáveis aleatórias. Caso a heurística resolve um problema de decisão, e.g. SAT, só consideramos a variável T. Para um problema de decisão obtemos a probabilidade de sucesso pela *função de distribuição acumulada* $F(t) = P[T \leq t]$. O algoritmo encontra um solução em tempo no máximo t com probabilidade $F(t)$.

Para um problema de otimização o tempo depende da qualidade. Logo obtemos a uma probabilidade de sucesso em duas variáveis pela função de distribuição acumulada

$$F(t, v) = P[T \leq t \wedge V \leq v].$$

Para um valor fixo v' obtemos a distribuição restrita de sucesso $F(t) = F(t, v')$. A função $F(t)$ também é chamada o grafo *time-to-target*. Para um tempo fixo t' obtemos a distribuição de qualidade de solução $F(v) = F(t', v)$.

Exemplo 6.6 (Função de distribuição acumulada para SAT)
A seguinte figura mostra a probabilidade de sucesso de um GRASP com $\alpha = 0.8$ na instância flat75-1 e 100 replicações.



◊

Exemplo 6.7 (Distribuição de tempo e qualidade em GNU R)
Dado um arquivo de tempos de execução

```
time
695
2888
...
```

podemos visualizar a distribuição dos tempos e a distribuição acumulada usando

```
d<-read.table("x.dat",header=T)
hist(d$time)
plot(ecdf(d$time),verticals=T,do.points=F)
```

◊

6.3.1. Testes estatísticos

O método básico para comparar a influência de fatores experimentais é o *teste estatístico*. Como podemos tratar o algoritmo usado como um fator

experimental, ele também serve para comparar diferentes heurísticas. Para aplicar um teste temos que

- formular uma hipótese nula e uma hipótese alternativa;
- escolher um teste estatístico adequado;
- definir um nível de significância;
- aplicar o teste e rejeitar ou aceitar a hipótese nula de acordo.

Exemplo 6.8 (Teste binomial)
Queremos descobrir se numa dada população nascem mais homens que mulheres. Seja X a variável aleatória tal que $X = 1$ caso nasce um homem. Logo a hipótese nula é $P[X] = 0.5$ e a hipótese alternativa é $P[X] > 0.5$.
Para decidir essa hipótese, podemos tirar uma amostra $X_1, \ldots, X_{10}$ da população base (de nascimentos). Supondo que as amostras são independentes, $X = \sum_{i \in [n]} X_i$ é distribuído binomialmente.

$$B(k; n, p) = \binom{n}{k} p^k (1-p)^{n-k}$$

a distribuição do $X \sim B(k; 10, 0.5)$ caso a hipótese nula é satisfeito. No exemplo obtemos

k	0/10	1/9	2/8	3/7	4/6	5
$P[X = k]$	0.001	0.010	0.044	0.117	0.205	0.246
$P[X \geq k]$	1.000	0.999	0.989	0.945	0.828	0.623
k	6	7	8	9	10	
$P[X \geq k]$	0.377	0.172	0.055	0.011	0.001	

Para aplicar o teste estatístico, temos que definir um nível de significância. Por exemplo, para um nível de significância $p = 0.05$ temos $P[X \geq 9] \leq p$. Logo podemos rejeitar a hipótese nula, com $p = 0.05$ caso na amostra tem 9 ou 10 nascimentos de homens. Para testar em R:

```
binom.test(9,10,alternative="g")
```

$\Diamond$

No exemplo acima formulas a hipótese alternativa $P[X] > 0.5$. Esse hipótese é *unilateral* (ou monocaudal), porque ela testa em determinada direção do desvio. Similarmente a hipótese alternativa $P[X] < 0.5$ é unilateral. Uma hipótese *bilateral* (ou bicaudal) é $P[X] \neq 0.5$. Neste caso temos que considerar desvios para as duas direções.

O exemplo mostra que o teste estatístico adequado depende das hipóteses sobre a distribuição da quantidade que queremos testar (no exemplo uma distribuição binomial). Um teste estatístico pode falhar em dois casos: num *erro de tipo 1* ele rejeita a hipótese nula, mesmo ela sendo correta; num *erro de tipo 2* ele não rejeita a hipótese nula, mesmo ela sendo falso. Isso pode ser resumido por

	H_0 mantido	H_0 rejeitado
H_0 verdadeiro	Correto	Erro tipo 1
H_1 verdadeiro	Erro tipo 2	Correto

O nível de significância do teste é a probabilidade da fazer um erro de tipo 1 $P[H_0 \text{ rejeitado} \mid H_0 \text{ verdadeiro}]$. A probabilidade condicional de não fazer um erro de tipo 2

$$1 - P[H_0 \text{ mantido} \mid H_1 \text{ verdadeiro} = P[H_0 \text{ rejeitado} \mid H_1 \text{ verdadeiro}]$$

é chamada a *potência* do teste.

Exemplo 6.9 (Teste binomial)
A potência de um teste depende da magnitude do efeito que queremos detectar. Supõe, por exemplo, que estamos interessados em detectar (pelo menos) o efeito caso na hipótese alternativa $P[X] > 0.6$. A distribuição $B(l; 10, 0.6)$ é

k	0	1	2	3	4	5
$P[X = k]$	0.0001	0.002	0.011	0.042	0.111	0.201
$P[X \geq k]$	1.000	0.9999	0.998	0.988	0.945	0.834
k	6	7	8	9	10	
$P[X = k]$	0.251	0.215	0.121	0.040	0.006	
$P[X \geq k]$	0.633	0.382	0.167	0.046	0.006	

Logo a potência do teste é com 0.046 relativamente fraco. Para $P[X] > 0.8$ a potência aumenta para 0.376. ◊

O exemplo mostra que o planejamento do experimento influencia a potência. Para aumentar a potência em geral, podemos

- aumentar o nível de significância: Isso aumenta também o probabilidade de erros do tipo 1.
- aumentar a magnitude de efeito: tipicamente não temos controle direto da magnitude, mas podemos planejar o experimento de acordo com a magnitude do efeito que queremos detectar (e.g. a redução do desvio relativo por 1%).

- diminuir a variança do efeito: tipicamente não temos controle direta da variança.
- aumentar o número de amostras (que diminui a variança): por exemplo para $n = 50$ amostras, com o mesmo nível de significância $p = 0.05$ o teste acima precisa $X \geq 31$ para rejeitar a hipótese nula e a potência do teste acima para detectar o efeito $P[X] > 0.6$ aumenta para 0.336, a para o efeito $P[X] > 0.8$ para 0.997. Uma amostra suficientemente grande que garante uma potência de 0.8 é considerada aceitável.

As características principais para a escolha de um teste adequado são

- o tipo de parâmetro que queremos analisar (e.g. mínimos, médias, medianas);
- testes paramétricos ou não-paramétricos: um teste paramétrico (tipicamente) supõe que a variável estudada é distribuída normalmente;
- o número de fatores e o número de níveis dos fatores;
- testes pareados ou não-pareados: em testes pareados, as amostras são dependentes. Um teste de dois algoritmos numa coleção de instâncias é um exemplo de um teste pareado. Caso as instâncias são geradas aleatoriamente, e cada algoritmo é avaliado em uma séria de instâncias geradas independentemente, o teste é não-pareado. (Testes de diferentes algoritmos com as mesmas sementes randômicos não podem ser considerados pareados, porque não podemos esperar que o semente tem um efeito semelhante nos dois algoritmos.) Em geral para mais que dois níveis de fatores temos um *teste (randomizado) em blocos*.

Testes comuns para comparação de algoritmos Para comparação de dois níveis temos como testes mais relevantes no caso não-paramétrico o teste do sinal (ingl. sign test) e de Wilcoxon de postos com sinais (ingl. Wilcoxon signed-rank test) para dados pareados, e o Wilcoxon da soma dos postos (ingl. Wilcoxon rank-sum test, equivalente com o teste U de Mann-Whitney) para dados não pareados. No caso paramétrico o teste t (pareado ou não pareado) pode ser aplicado.

Teste estatístico 6.1 (Teste do sinal)
Pré-condições Duas amostras pareadas $x_1, \ldots, x_n$ e $y_1, \ldots, y_n$. Os valores $x_i - y_i$ são independentes e distribuídos com mediana comum m.

Hipótese nula H_0: $m = 0$;

Hipótese alternativa H_1: $m > 0$, $m < 0$, $m \neq 0$.

Estatística de teste $B = \sum_{i \in [n]} [x_i > y_i]$.

Observações Valores $z_i = 0$ são descartadas (ou atribuídos pela metade para o grupo com $x_i > y_i$).

Exemplo 6.10 (Teste do sinal)
O teste do sinal de fato é equivalente com um teste binomial. Para estatística de teste B é n amostras

```
binom.test(B,n,alternative="greater")
binom.test(B,n,alternative="less")
binom.test(B,n,alternative="two-sided")
```

testa a hipótese em GNU R (com nível de significância padrão 0.05.). Por exemplo, para comparar os tempos do GSAT com os do WalkSAT (ver exercícios) com hipótese alternative que WalkSAT precisa mais tempo que o GSAT

```
> e
        GSAT   WalkSAT
1 9178.66667 120000.00
2   44.13333  17502.87
3  974.60000 120000.00
4  189.80000 107423.87
> binom.test(sum(e$WalkSAT>e$GSAT),4,alternative="greater")

Exact binomial test

data:  sum(e$WalkSAT > e$GSAT) and 4
number of successes = 4, number of trials = 4, p-value = 0.0625
alternative hypothesis: true probability of success is greater than 0.5
95 percent confidence interval:
 0.4728708 1.0000000
sample estimates:
probability of success
                     1
```

Mesmo o GSAT precisando em todos quatro casos menos tempo que o WalkSAT não podemos rejeitar a hipótese nula com nível de significância $p = 0.05$, pelo número baixo de amostras. ◊

Exemplo 6.11 (Teste do sinal para comparação de modelos matemáticos) Tseng et al. (2004) usam o teste de sinal para testar se pares de modelos matemáticas para o problema do permutation flow shop precisam tempo significadamente diferente.

◊

Teste estatístico 6.2 (Teste de Wilcoxon de postos com sinais)
Pré-condições Duas amostras pareadas $x_1, \ldots, x_n$ e $y_1, \ldots, y_n$. Os valores $z_i = x_i - y_i$ são independentes é distribuídos simétricos relativo a um mediana comum m.

Hipótese nula H_0: $m = 0$.

Hipótese alternativa H_1: $m > 0$, $m < 0$, $m \neq 0$.

Estatística de teste $T^+ = \sum_{i \in [n]} r_i [x_i > y_i]$ com r_i o ranque do valor z_i em ordem crescente de $|z_i|$.

Observações Valores $z_i = 0$ são descartadas. Em caso de empates na ordem de $|z_i|$ cada elemento de um grupo recebe o ranque médio.

Em GNU R `wilcox.test(...,paired=T)`.

Exemplo 6.12 (Teste de Wilcoxon de postos com sinais) (Continuando o exemplo anterior.)

```
wilcox.test(e$WalkSAT,e$GSAT,alternative="greater",paired=T)

Wilcoxon signed rank test

data:  e$WalkSAT and e$GSAT
V = 10, p-value = 0.0625
alternative hypothesis: true location shift is greater than 0
```

◊

Exemplo 6.13 (Gino versus Optisolve) Coffin e Saltzmann (2000) apresentam uma análise de um exemplo de Golden et al. (1986)[2].

[2] A análise na publicação está errada: ela compara o tempo da primeira instância de Gino com o tempos do Optisolve.

```
d<-read.table("golden-etal.dat",header=T)
d<-subset(d,optG==T&optO==T&!is.na(timeO))
plot(d$timeG,d$timeO)
abline(0,1)
binom.test(sum(d$timeO>d$timeG),nrow(e))
wilcox.test(sum(d$timeO>d$timeG),nrow(e),paired=T)
```

◊

Teste estatístico 6.3 (Teste de Wilcoxon da soma dos postos)

Pré-condições Duas amostras não-pareadas $x_1, \ldots, x_n$ e $y_1, \ldots, y_m$. Os x_i são independentes e distribuídos igualmente, os y_i são independentes e distribuídos igualmente, e os x_i e y_i são independentes.

Hipótese nula $F_x(t) = F_y(t)$ para todo t, para distribuições acumuladas F_x e F_y desconhecidas. No modelo mais simples supondo a mesma distribuição $F_x(t) = F_y(t)$, a hipótese alternativa é um deslocamento, i.e.$F_x(t) = F_y(t - \Delta)$. A hipótese nula nessa caso é $\Delta = 0$.

Hipótese alternativa H_1: $\Delta < 0$, $\Delta = 0$, $\Delta > 0$.

Estatística de teste $S = \sum_{i \in [m]} r_i$ com r_i o ranque de y_i na ordem crescente de todos valores x_i e y_i.

Em GNU R `wilcox.test(...,paired=F)`.

Exemplo 6.14 (Teste de Wilcoxon da soma dos postos)
Continuando o exemplo anterior.

```
wilcox.test(e$WalkSAT,e$GSAT,alternative="greater",paired=F)

Wilcoxon rank sum test with continuity correction

data:  e$WalkSAT and e$GSAT
W = 16, p-value = 0.0147
alternative hypothesis: true location shift is greater than 0

Warning message:
In wilcox.test.default(e$WalkSAT, e$GSAT, alternative = "greater",  :
  cannot compute exact p-value with ties
```

◊

Teste estatístico 6.4 (Teste t de Student)
Pré-condições Duas amostras pareadas $x_1, \ldots, x_n$, e $y_1, \ldots y_n$. Os valores $z_i = x_i - y_i$ são distribuídos normalmente $\sim N(\mu, \sigma^2)$. (A normalidade não é necessária para amostras suficientemente grandes, e.g. $n, m < 30$).

Hipótese nula H_0: $\mu = 0$.

Hipótese alternativa H_1: $\mu < 0$, $\mu > 0$, $\mu \neq 0$.

Estatística de teste $t = \bar{z}/S\sqrt{n}$ com $S^2 = \sum_i (z_i - \bar{z})/(n-1)$ uma estimativa da variança da população inteira. A estatística é distribuída t com $n-1$ graus de liberdade.

Em GNU R `t.test`.

Teste estatístico 6.5 (Teste t de Student)
Pré-condições Duas amostras não-pareadas $x_1, \ldots, x_n$, e $y_1, \ldots y_m$. Os x_i são distribuídos normalmente $\sim N(\mu_x, \sigma^2)$, os y_i normalmente $\sim N(\mu_y, \sigma^2)$. (A normalidade não é necessária para amostras suficientemente grandes, e.g. $n, m < 30$).

Hipótese nula H_0: $\mu_x = \mu_y$.

Hipótese alternativa H_1: $\mu_x < \mu_y$, $\mu_x > \mu_y$, $\mu_x \neq \mu_y$.

Estatística de teste $t = (\bar{x} - \bar{y})/(S\sqrt{1/n + 1/m})$ com

$$S = \sqrt{\frac{(n-1)S_x^2 + (m-1)S_y^2}{n+m-2}}$$

uma estimativa do desvio padrão da população inteira. A estatística é distribuída t com $n+m-2$ graus de liberdade.

Em GNU R `t.test(x,y,var.equal=T,paired=F)`; para variânças diferentes: `t.test(x,y,var.equal=F,paired=F)`.

Exemplo 6.15 (MINOS versus OB1)
Coffin e Saltzmann (2000) apresentam uma análise de um exemplo de Lustig et al. (1991). O teste do coeficiente β_1 da regressão linear do exemplo é um

teste t. Neste caso a estatística de teste $t = (\hat{\beta}_1 - \beta_1)/se(\hat{\beta}_1)$ com

$$se(\hat{\beta}_1) = \sqrt{\frac{(\sum_i e_i^2)/(n-2)}{\sum_i (x_i - \bar{x})^2}}$$

e resíduos e_i é distribuída t com $n-2$ graus de liberdade.

```
d<-read.table("lustig-etal.dat",header=T)
attach(d)
plot(minos.time,ob1.time)
plot(log(minos.time),log(ob1.time))
l<-lm(log(ob1.time)~log(minos.time))
summary(lm)
# t-test
es = resid(l)
n = length(es)
se = sqrt(sum(es^2)/(n-2))
se = se/sqrt(sum((log(minos.time)-mean(log(minos.time)))^2))
t=(1-coef(l)[2])/se
pt(t,n-2,lower.tail=F)
```

◊

6.3.2. Escolha de parâmetros

Princípio de projeto 6.1 (Parâmetros (Hertz et al. 2003, p. 127)**)**
O projeto do método em si (vizinhança, função objetivo, etc.) é mais importante que a escolha de parâmetros. Um bom método deve ser robusto: a qualidade das soluções é menos sensível à escolha de parâmetros. Porém, a calibração de parâmetros não compensa um método fraco.

O ponto de partido frequentemente é um conjunto de parâmetros inciais obtidos durante o projeto por testes ad hoc. Para heurísticas robustas e parâmetros simples um tal conjunto frequentemente é uma escolha razoável. Porém robustez tem que ser demonstrada e não podemos esperar robustez sobre a modificação de componentes da heurística (e.g. vizinhanças, operadores de recombinação).
A busca para um conjunto ideal de parâmetros é uma problema de otimização separado, que a princípio pode ser resolvido pelas técnicas discutidas. Porém para obter o valor função objetivo temos que avaliar agora uma heurística (em diversas instâncias e com replicações no caso de algoritmos randomizados).

A estratégia mais simples é analisar um parâmetro por vez (ingl. one factor at a time, OFAT): determine a variação do desempenho da heurística para cada parâmetro independentemente, com os outros parâmetros fixos. Depois seleciona uma combinação de parâmetros que melhora o desempenho e eventualmente repete. Para comparação de diferentes níveis de uma parâmetro pode-se aplicar testes estatísticos. Esse método serve também para analisar o impacto de diversos parâmetros e selecionar um subconjunto para ser calibrado ("screening"). As desvantagens do OFAT são: i) ignorar interações de parâmetros, ii) aumentar os erros de tipo 1 no caso de aplicações de testes estatísticos, e iii) um custo maior que outras formas de experimentos (Montgomery 2009).

Um *projeto fatorial* testa l^k *células*, i.e., combinações dos l níveis de k fatores. Para algoritmos randomizados cada célula precisa algumas replicações do experimento. Projetos fatoriais comuns são o *projeto fatorial completo* 2^k (muitas vezes usado para "screening") e o projeto fatorial completo com um fator em l níveis. Um projeto fatorial geralmente supõe um modelo linear dos efeitos dos fatores. No caso de uma aplicação em instâncias fixas obtemos um *projeto em blocos* que generaliza um projeto pareado. (A aplicação para instâncias geradas aleatoriamente poderia ser tratado como *projeto completamente randomizado*; porém o efeito da instância muitas vezes é significativo, e não pode ser modelado como um erro simples.) A disciplina de *projeto de experimentos* (ingl. design of experiments) oferece mais possibilidades, inclusive projetos fatoriais fracionários que testam menos combinações de parâmetros, mas em contrapartida não conseguem identificar todas interações univocamente.

Projetos fatoriais podem ser avaliados por *analise de variação* (ingl. analysis of variation, ANOVA) no caso paramétrico, e no caso não-paramétrico por um teste Kruskal-Wallis (sem blocos) ou um teste de Friedman (com blocos).

Em exemplo de uma ANOVA com um fator experimental:

Teste estatístico 6.6 (ANOVA)

Pré-condições Um projeto k tratamentos e n replicações por tratamento. O problema é modelado linearmente por

$$x_{ij} = \mu + \tau_i + \epsilon_{ij}.$$

para tratamentos $i \in [k]$ e replicações $j \in [n]$. O valor τ_i é o efeito do tratamento $i \in [k]$. Os error são independentes e distribuídos normalmente como $N(0; \sigma^2)$. (Em particular a variança é constante, i.e. os erros são homoscedasticos).

Hipótese nula H_0: $\tau_1 = \cdots = \tau_k = 0$.

Hipótese alternativa H_1: existe um i com $\tau_i \neq 0$.

Estatística de teste A soma de quadrados total SS_T pode ser decomposta por $SS_T = SS_A + SS_E$ (similar com a observação 6.2) em uma soma de quadrados dos tratamentos SS_A e dos erros SS_E. Os tratamentos possuem $k-1$ graus de liberdade, os erros $kn-k$. As médias das somas de quadrados $MS_A = SS_A/(k-1)$ e $MS_E = SS_E/(kn-k)$ são distribuídos χ e a estatística de teste $F_0 = MS_A/MS_E$ é distribuída F. Caso não existe um efeito dos tratamentos, esperamos $F_0 = 1$, caso contrário $F_0 > 1$.

Em GNU R `aov`.

Exemplo 6.16 (ANOVA)

```
d=read.table("mont-etch.dat",header=T,
             colClasses=c("factor","numeric"))
a=aov(rate~power,data=d)
summary(a)
plot(a)
plot(TukeyHSD(a,ordered=T))
```

◊

Caso a hipótese nula é rejeitada um teste post-hoc pode ser usado para identificar os grupos significativamente diferentes. Uma abordagem simples é comparar todos grupos par a par com um teste simples (e.g. um teste t). Porém a probabilidade de um erro do tipo 1 aumenta com o número de testes. Uma solução para este problema é aplicar uma *correção Bonferroni*: para um nível de significância desejada α e n testes em total, cada teste é aplicado com um nível de significância α/n. Um exemplo de um teste menos conservativo é *Tukey's honest significant differences*, uma generalização do teste t para múltiplas médias.

Teste estatístico 6.7 (Teste de Friedman)

Pré-condições Um projeto em blocos (randomizado) com k tratamentos e n blocos. As variáveis aleatórias x_{ij} seguem distribuições desconhecidas F_{ij} relacionadas por $F_{ij}(u) = F(u - \beta_i - \tau_j)$, com β_i o efeito do bloco $i \in [n]$ e τ_j o efeito do tratamento $j \in [k]$.

Hipótese nula H_0: $\tau_1 = \cdots = \tau_k$.

Hipótese alternativa H_1: não todos τ_j são iguais.

Estatística de teste Com R_{ij} o posto do tratamento j no bloco i e $R_j = \sum_i R_{ij}$

$$T = \frac{(k-1)\sum_{j\in[k]}(R_j - n(k+1)/2)^2}{\sum_{i\in[n],j\in[k]} R_{ij}^2 - nk(k+1)^2/4}.$$

Observações Para amostras suficientemente grandes $T \sim \chi^2$ com $k-1$ graus de liberdade. Caso H_0 é rejeitado, testes post-hoc podem ser usados para identificar o melhor tratamento.

Em GNU R `friedman.test(m)` com matriz m.

Exemplo 6.17 (Teste Friedman)

```
e=data.frame(n=gl(3,3),h=rep(c(1,2,3)),v=runif(9))
with(e,friedman.test(v~h*n))
```

◊

Uma aplicação do teste de Friedman: corridas Testar todas combinações de parâmetros em todas instâncias investe um tempo igual em todas combinações. Uma corrida (ingl. race) aplica as combinações instância por instância e elimina combinações inefetivas da corrida logo, investindo mais tempo de teste em combinações melhores. Uma exemplo de uma estratégia de corrida é F-RACE, um algoritmo que aplica o teste de Friedman para eliminar combinações de parâmetros.

Algoritmo 6.1 (F-RACE)
Entrada Um conjunto de combinações de parâmetros $\Theta = \{\Theta_1, \ldots, \Theta_k\}$.

Saída Um subconjunto $\Theta' \subseteq \Theta$ de combinações de parâmetros efetivas.

```
1  F–RACE(Θ) :=
2  repeat for i = 1,...
3    gera a i–ésima instância I
4    aplica todas combinações de parâmetros em Θ em I
5    aplica o teste de Friedman
6      (na matriz i × |Θ|)
7    if H0 rejeitada then
8      seleciona o Θj de menor posto combinado Rj
9      remove todos tratamentos significadamente
10       pior que Θj (via testes post–hoc) de Θ
```

```
    end if
until |Θ| = 1 ou limite de tempo
return Θ
```

Para gerar a conjunto Θ inicial podemos usar um projeto fatorial completo (F-RACE(FFD)) ou simplesmente gerar amostras aleatórias dos parâmetros (F-RACE(RSD)).

6.3.3. Comparar com que?

> - Quietly employ assembly code and other low-level language constructs.
> - When direct run time comparison are required, compare with an old code on an obsolete system.
>
> "Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers", Bailey (1991)

Uma heurística tem que ser comparado com outros algoritmos existentes; em casos de problemas novos podemos comparar com algoritmos existentes para casos particulares e generalizações do problema, ou com algoritmos mais simples (e.g. uma construção ou busca randomizada simples, ou versões simplificadas do algoritmo proposto) ou genéricos (e.g. CPLEX, localsolver). Isso inclui algoritmos exatos e aproximativos, e evita situações como essa:

> A recent paper (Davidović et al. 2012) presented a bee colony metaheuristic for scheduling independent tasks to identical processors, evaluating its performance on a benchmark set of instances from the literature. We examine two exact algorithms from the literature, the former published in 1995, the latter in 2008 (and not cited by the authors). We show that both such algorithms solve to proven optimality all the considered instances in a computing time that is several orders of magnitude smaller than the time taken by the new algorithm to produce an approximate solution.
>
> Dell'Amico et al. (2012)

6.4. Notas

Barr et al. (1995) e Silberholz e Golden (2010) explicam de forma geral o tem que ser considerado na avaliação de heurísticas. Luke (2011, cáp. 11.) é uma

boa introdução na ideias gerais de comparação de algoritmos e Coffin e Saltzmann (2000) é uma excelente introdução com diversos exemplos práticos. O livro de Bartz-Beielstein et al. (2010) apresenta em grande detalhe a aplicação de métodos estatísticos na avaliação de heurísticas. Hollander e Wolfe (1999) é uma referência detalhada para métodos estatísticos não-paramétricos. (LeVeque 2013) é um ensaio recomendado sobre a publicação de código.

A. Conceitos matemáticos

Definição A.1
Uma função f é *convexa* se ela satisfaz a desigualdade de Jensen

$$f(\Theta x + (1-\Theta)y) \leq \Theta f(x) + (1-\Theta)f(y). \tag{A.1}$$

Similarmente uma função f é *concava* caso $-f$ é convexo, i.e., ela satisfaz

$$f(\Theta x + (1-\Theta)y) \geq \Theta f(x) + (1-\Theta)f(y). \tag{A.2}$$

Exemplo A.1
Exemplos de funções convexas são x^{2k}, $1/x$. Exemplos de funções concavas são $\log x$, $\sqrt{x}$. ◊

Proposição A.1
Para $\sum_{i\in[n]} \Theta_i = 1$ e pontos x_i, $i \in [n]$ uma função convexa satisfaz

$$f\Big(\sum_{i\in[n]} \Theta_i x_i\Big) \leq \sum_{i\in[n]} \Theta_i f(x_i) \tag{A.3}$$

e uma função concava

$$f\Big(\sum_{i\in[n]} \Theta_i x_i\Big) \geq \sum_{i\in[n]} \Theta_i f(x_i) \tag{A.4}$$

Prova. Provaremos somente o caso convexo por indução, o caso concavo sendo similar. Para $n = 1$ a desigualdade é trivial, para $n = 2$ ela é válida por definição. Para $n > 2$ define $\bar{\Theta} = \sum_{i\in[2,n]} \Theta_i$ tal que $\Theta + \bar{\Theta} = 1$. Com isso temos

$$f\Big(\sum_{i\in[n]} \Theta_i x_i\Big) = f\big(\Theta_1 x_1 + \sum_{i\in[2,n]} \Theta_i x_i\big) = f(\Theta_1 x_1 + \bar{\Theta} y)$$

onde $y = \sum_{j\in[2,n]}(\Theta_j/\bar{\Theta})x_j$, logo

$$\begin{aligned} f\Big(\sum_{i\in[n]}\Theta_i x_i\Big) &\le \Theta_1 f(x_1) + \bar{\Theta} f(y) \\ &= \Theta_1 f(x_1) + \bar{\Theta} f\Big(\sum_{j\in[2,n]}(\Theta_j/\bar{\Theta})x_j\Big) \\ &\le \Theta_1 f(x_1) + \bar{\Theta}\sum_{j\in[2,n]}(\Theta_j/\bar{\Theta}) f(x_j) = \sum_{i\in[n]}\Theta_i x_i \end{aligned}$$

■

Definição A.2
O *fatorial* é a função

$$n! : \mathbb{N} \to \mathbb{N} : n \mapsto \prod_{1\le i\le n} i.$$

Temos a seguinte aproximação do fatorial (fórmula de Stirling)

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1 + O(1/n)) \tag{A.5}$$

Uma estimativa menos preciso pode ser obtido estimando

$$e^n = \sum_{i\ge 0}\frac{n^i}{i!} > \frac{n^n}{n!}$$

que implica

$$(n/e)^n \le n! \le n^n.$$

Lema A.1 (Desigualdade de Bernoulli)
Para $x \ge -1$ e $n \in \mathbb{N}$ temos $(1+x)^n \ge 1 + xn$.

Prova. Por indução sobre n.

$$\begin{aligned} (1+x)^{n+1} &= (1+x)(1+x)^n \ge (1+x)(1+xn) \\ &= 1 + xn + x + x^2 n = 1 + x(n+1) + x^2 n \ge 1 + x(n+1). \end{aligned}$$

onde a primeira desigualdade é válida porque $(1+x) \ge 0$. ■

Definição A.3 (Entropia binária)
A entropia binária para $\alpha \in (0,1)$ é $h(\alpha) = -\alpha\log_2\alpha - (1-\alpha)\log_2 1 - \alpha$.

Lema A.2 (Ash (1967))
Para $\alpha \in (0,1)$

$$(8n\alpha(1-\alpha))^{-1/2}\, 2^{h(\alpha)n} \leq \binom{n}{\alpha n} \leq (2\pi n\alpha(1-\alpha))^{-1/2} 2^{h(\alpha)n}$$

Lema A.3
Para $\alpha \in (0, 1/2]$

$$(8n\alpha(1-\alpha))^{-1/2}\, 2^{h(\alpha)n} \leq \sum_{1\leq i\leq n\alpha} \binom{n}{i} \leq 2^{h(\alpha)n}.$$

Prova. A primeira desigualdade é uma consequência do lema A.2. Para a segunda desigualdade temos

$$\begin{aligned}
1 = (\alpha + (1-\alpha))^n &= \sum_{1\leq i\leq n} \binom{n}{i} \alpha^i (1-\alpha)^{n-i} \\
&\geq \sum_{1\leq i\leq n\alpha} \binom{n}{i} \left(\frac{\alpha}{1-\alpha}\right)^i (1-\alpha)^n \\
&\geq \sum_{1\leq i\leq n\alpha} \binom{n}{i} \left(\frac{\alpha}{1-\alpha}\right)^{n\alpha} (1-\alpha)^n \\
&= \alpha^{n\alpha}(1-\alpha)^{(1-\alpha)n} \sum_{1\leq i\leq n\alpha} \binom{n}{i} \\
&= 2^{-nh(\alpha)} \sum_{1\leq i\leq n\alpha} \binom{n}{i}.
\end{aligned}$$

O terceiro passo é valido porque para $\alpha \in (0, 1/2]$ temos $\alpha/(1-\alpha) \leq 1$ e $i \leq n\alpha$. ■

A.1. Probabilidade discreta

Probabilidade: Noções básicas

- *Espaço amostral* finito Ω de *eventos elementares* $e \in \Omega$.
- Distribuição de probabilidade $\Pr[e]$ tal que

$$\Pr[e] \geq 0; \qquad \sum_{e \in \Omega} \Pr[e] = 1$$

- *Eventos* (compostos) $E \subseteq \Omega$ com probabilidade

$$\Pr[E] = \sum_{e \in E} \Pr[e]$$

Exemplo A.2
Para um dado sem bias temos $\Omega = \{1, 2, 3, 4, 5, 6\}$ e $\Pr[i] = 1/6$. O evento $\text{Par} = \{2, 4, 6\}$ tem probabilidade $\Pr[\text{Par}] = \sum_{e \in \text{Par}} \Pr[e] = 1/2$. $\Diamond$

Probabilidade: Noções básicas

- *Variável aleatória*

$$X : \Omega \to \mathbb{N}$$

- Escrevemos $\Pr[X = i]$ para $\Pr[X^{-1}(i)]$.
- Variáveis aleatórias *independentes*

$$P[X = x \text{ e } Y = y] = P[X = x]P[Y = y]$$

- *Valor esperado*

$$E[X] = \sum_{e \in \Omega} \Pr[e]X(e) = \sum_{i \geq 0} i \Pr[X = i]$$

- Linearidade do valor esperado: Para variáveis aleatórias X, Y

$$E[X + Y] = E[X] + E[Y]$$

Prova. (Das formulas equivalentes para o valor esperado.)

$$\begin{aligned}\sum_{0\le i} \Pr[X = i]i &= \sum_{0\le i} \Pr[X^{-1}(i)]i \\ &= \sum_{0\le i} \sum_{e\in X^{-1}(i)} \Pr[e]X(e) = \sum_{e\in\Omega} \Pr[e]X(e)\end{aligned}$$

■

Prova. (Da linearidade.)

$$\begin{aligned}E[X+Y] &= \sum_{e\in\Omega} \Pr[e](X(e)+Y(e)) \\ &= \sum_{e\in\Omega} \Pr[e]X(e) \sum_{e\in\Omega} \Pr[e]Y(e)) = E[X]+E[Y]\end{aligned}$$

■

Exemplo A.3
(Continuando exemplo A.2.)
Seja X a variável aleatório que denota o número sorteado, e Y a variável aleatório tal que $Y =$ [a face em cima do dado tem um ponto no meio].

$$E[X] = \sum_{i\ge 0} \Pr[X = i]i = 1/6 \sum_{1\le i\le 6} i = 21/6 = 7/2$$
$$E[Y] = \sum_{i\ge 0} \Pr[Y = i]i = \Pr[Y = 1] = 1/2 \quad E[X+Y] = E[X]+E[Y] = 4$$

◊

Lema A.4 (Forma alternativa da expectativa)
Para uma variável aleatória X que assume somente valores não-negativos inteiros $E[X] = \sum_{k\ge 1} P[X \ge k] = \sum_{k\ge 0} P[X > k]$.

Prova.

$$E[X] = \sum_{k\ge 1} kP[X = k] = \sum_{k\ge 1} \sum_{j\in[k]} P[X = k] = \sum_{j\ge 1} \sum_{j\le k} P[X = k] = \sum_{j\ge 1} P[X \ge j].$$

■

Lema A.5
Para tentativas repetidas com probabilidade de sucesso p, o número esperado de passos para o primeiro sucesso é $1/p$.

Prova. Seja X o número de passos até o primeiro sucesso. Temos $P[X > k] = (1-p)^k$ e logo pelo lema A.4

$$E[X] = \sum_{k \geq 0} (1-p)^k = 1/p.$$

■

Proposição A.2
Para φ convexo $\varphi(E[X]) \leq E[\varphi(X)]$ e para φ concavo $\varphi(E[X]) \geq E[\varphi(X)]$.

Prova. Pela proposição A.1. ■

Proposição A.3 (Desigualdade de Markov)
Seja X uma variável aleatória com valores não-negativas. Então, para todo $a > 0$

$$Pr[X \geq a] \leq E[X]/a.$$

Prova. Seja $I = [X \geq a]$. Como $X \geq 0$ temos $I \leq X/a$. O valor esperado de I é $E[I] = Pr[I = 1] = Pr[X \geq a]$, logo

$$Pr[X \geq a] = E[I] \leq E[X/a] = E[X]/a.$$

■

Proposição A.4 (Limites de Chernoff (ingl. Chernoff bounds))
Sejam $X_1, \ldots, X_n$ indicadores independentes com $Pr[X_i] = p_i$. Para $X = \sum_i X_i$ temos para todo $\delta > 0$

$$Pr[X \geq (1+\delta)\mu] \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}} \right)^{\mu}$$

para todo $\delta \in (0, 1)$

$$Pr[X \leq (1-\delta)\mu] \leq \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}} \right)^{\mu}$$

para todo $\delta \in (0, 1]$

$$Pr[X \geq (1+\delta)\mu] \leq e^{-\mu\delta^2/3}$$

e para todo $\delta \in (0, 1)$

$$Pr[X \leq (1-\delta)\mu] \leq e^{-\mu\delta^2/2}.$$

Exemplo A.4
Sejam $X_1, \ldots, X_k$ indicadores com $\Pr[X_i = 1] = \alpha$ e $X = \sum_i X_i$. Temos $\mu = \mathsf{E}[X] = \sum_i \mathsf{E}[X_i] = \alpha k$. Qual a probabilidade de ter menos que a metade dos $X_i = 1$?

$$\Pr[X \leq \lfloor k/2 \rfloor] \leq \Pr[X \leq k/2] = \Pr[X \leq \mu/2\alpha] = \\ \Pr[X \leq \mu(1-(1-1/2\alpha))] \leq e^{-\mu\delta^2/2} = e^{-k/2\alpha(\alpha-1/2)^2}.$$

$\Diamond$

Medidas básicas A covariância de duas variáveis aleatórias X e Y é

$$\operatorname{cov}(X,Y) = \mathsf{E}[(X-\mathsf{E}[X])\mathsf{E}[Y-\mathsf{E}[Y]] = \mathsf{E}[XY] - \mathsf{E}[X]\mathsf{E}[Y].$$

A variança de uma variável aleatória X é a covariança com si mesmo

$$\sigma(X) = \operatorname{cov}(X,X) = \mathsf{E}[X^2] - \mathsf{E}[X]^2 \tag{A.6}$$

e o seu *desvio padrão* é $\sigma(X) = \sqrt{\operatorname{cov}(X)}$. A *correlação* entre duas variáveis aleatórias é a covariança normalizada

$$\rho(X,Y) = \operatorname{cov}(X,Y)/(\sigma(X)\sigma(Y)). \tag{A.7}$$

A figura A.1 mostra exemplos de dados com correlações diferentes.

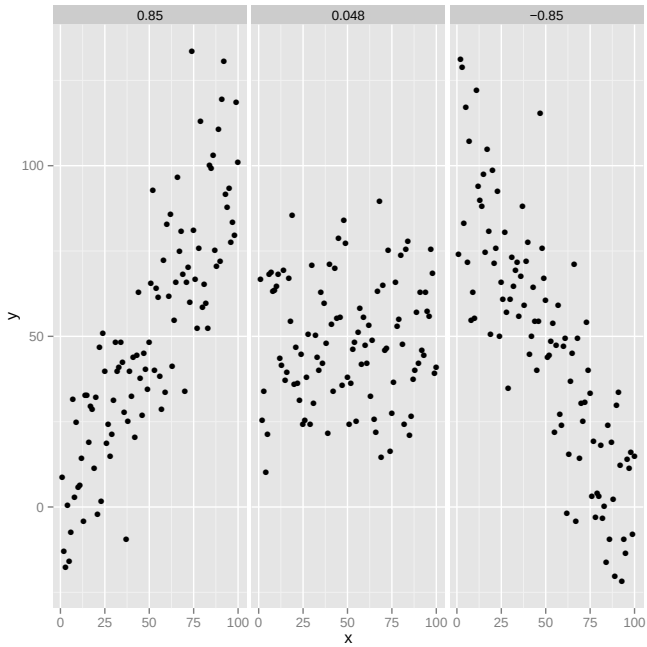


Figura A.1.: Três conjuntos de dados com correlação alta, quase zero, e negativa.

Bibliografia

Aarts, E. e J. K. Lenstra, eds. (2003). *Local Search in Combinatorial Optimization*. Princeton University Press. ISBN: 978-0691115221. URL: http://press.princeton.edu/titles/7564.html (ver p. 32).

Ackley, D. H. (1987). *A connectionist machine for genetic hillclimbing*. Kluwer (ver p. 55).

Aldous, D. e U. Vazirani (1994). ""Go With the Winners" Algorithms". Em: *Proc. 26th STOC* (ver pp. 25, 26).

Ash, R. B. (1967). *Information theory*. Wiley (ver p. 127).

Bailey, D. H. (ago. de 1991). "Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers". Em: *Supercomputing Review* 4.8, pp. 54–55 (ver p. 122).

Baker, J. E. (1987). "Reducing bias and inefficiency in the selection algorithm". Em: *Proceedings of the 2nd International Conference on Genetic Algorithms*. Ed. por J. J. Grefenstette, pp. 14–21 (ver p. 65).

Barr, R. S., B. L. Golden, J. P. Kelly, M. G. C. Resende e W. R. S. Jr. (1995). "Designing and reporting on computational experiments with heuristic methods". Em: *J. Heuristics* 1, pp. 9–32 (ver pp. 99, 100, 122).

Bartz-Beielstein, T., M. Chiarandini, L. Paquete e M. Preuss, eds. (2010). *Experimental methods for the analysis of optimization algorithms*. Springer. ISBN: 978-3-642-02538-9 (ver p. 123).

Bean, J. C. (1994). "Genetic algorithms and random keys for sequencing and optimization". Em: *ORSA J. Comput.* 6, pp. 154–160 (ver p. 69).

Boettcher, S. e A. G. Percus (2003). "Optimization with extremal dynamics". Em: *Complexity* 8.2, pp. 57–62 (ver pp. 34, 35).

Boyd, S. e L. Vanderberghe (2004). *Convex optimization*. Cambridge University Press (ver p. 95).

Bresina, J. L. (1996). "Heuristic-biased stochastic sampling". Em: *Proc. 13th Nat. Conf. on Artificial Intelligence*, pp. 271–278 (ver p. 48).

Burke, E. K. e Y. Bykov (2012). *The Late Acceptance Hill-Climbing Heuristic*. Rel. téc. Computing Science e Mathematics, University of Stirling (ver p. 31).

Burke, E. K. e G. Kendall, eds. (2005). *Search methodologies*. Springer. URL: http://www.springer.com/mathematics/applications/book/978-0-387-23460-1 (ver p. 11).

Cerny, V. (1985). "Thermodynamical approach to the travelling salesman problem: An efficient simulation algorithm". Em: *J. Opt. Theor. Appl.* 45, pp. 41–51 (ver p. 33).

Chandra, B., H. Karloff e C. Tovey (1999). "New results on the old k-opt algorithm for the TSP". Em: *SIAM J. Comput.* 28.6, pp. 1998–2029 (ver pp. 20, 21, 23).

Coffin, M. e M. J. Saltzmann (2000). "Statistical analysis of Computational Tests of Algorithms and Heuristics". Em: *INFORMS J. Comput.* 12.1, pp. 24–44 (ver pp. 114, 116, 123).

Congram, R. K., C. N. Potts e S. L. van de Velde (2002). "An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem". Em: *INFORMS J. Comput.* 14.1, pp. 52–67 (ver p. 77).

Cordeau, J.-F., M. Gendreau, G. Laporte, J.-Y. Potvin e F. Semet (2002). "A guide to vehicle routing heuristics". Em: *J. Oper. Res. Soc.* 53 (ver p. 99).

Cowling, P. I., G. Kendall e E. Soubeiga (2000). "A hyperheuristic approach for scheduling a sales summit". Em: *Selected Papers of the Third International Conference on the Practice And Theory of Automated Timetabling*. LNCS, pp. 176–190 (ver p. 78).

Creutz, M. (mai. de 1983). "Microcanonical Monte Carlo Simulation". Em: *Phys. Rev. Lett.* 50.19, pp. 1411–1414. DOI: 10.1103/PhysRevLett.50.1411 (ver p. 31).

Croes, G. A. (1958). "A Method for Solving Traveling-Salesman Problems". Em: *Oper. Res.* 6, pp. 791–812. DOI: 10.1287/opre.6.6.791 (ver p. 14).

Danna, E., E. Rothberg e C. L. Pape (2005). "Exploring relaxation induced neighborhoods to improve MIP solutions". Em: *Math. Prog. Ser. A* 102, pp. 71–90 (ver p. 76).

Davidović, T., M. Selmić, D. Teodorović e D. Ramljak (2012). "Bee colony optimization for scheduling independent tasks to identical processors". Em: *J. Heuristics* 18, pp. 549–569 (ver p. 122).

Dell'Amico, M., M. Iori, S. Martello e M. Monaci (2012). "A note on exact and heuristic algorithms for the identical parallel machine scheduling problem". Em: *J. Heuristics* 18, pp. 393–942 (ver p. 122).

Denzinger, J., M. Fuchs e M. Fuchs (1997). "High performance ATP systems by combining several AI methods". Em: *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pp. 102–107 (ver p. 78).

Dimitriou, T. e R. Impagliazzo (1996). "Towards an Analysis of Local Optimization Algorithms". Em: *Proc. 28th STOC* (ver p. 29).

Dorigo, M., V. Maniezzo e A. Colorni (1996). "Ant system: optimization by a colony of cooperating agents". Em: *IEEE Trans. Syst. Man Cybern. – Part B* 26.1, pp. 29–41 (ver p. 53).

Droste, S., T. Jansen e I. Wegener (2002). "On the analysis of the (1 + 1) evolutionary algorithm". Em: *Theor. Comput. Sci.* 276.1–2, pp. 51–81 (ver p. 66).

Dueck, G. (1993). "New optimization heuristics". Em: *J. of Comput. Phys.* 104, pp. 86–92 (ver pp. 31, 32, 42).

Dueck, G. e T. Scheuer (1990). "Threshold Accepting. A General Purpose Optimization Algorithm Superior to Simulated Annealing". Em: *J. of Comput. Phys.* 90, pp. 161–175 (ver pp. 31, 42).

Eberhart, R. C. e J. Kennedy (1995). "A new optimizer using particle swarm theory". Em: *In Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pp. 39–43 (ver p. 70).

Eshelman, L. J. (1990). "The CHC Adaptive Search Algorithm: How to Have Safe Search When Engaging in Nontraditional Genetic Recombination". Em: *FOGA*. Ed. por G. J. E. Rawlins. Morgan Kaufmann, pp. 265–283. ISBN: 1-55860-170-8 (ver p. 68).

Faigle, U. e R. Schrader (1992). "Some convergence results for probabilistic tabu search". Em: *ORSA J. Comput.* 4, pp. 32–37 (ver p. 36).

Feo, T. e M. Resende (1989). "A probabilistic heuristic for a computationally difficult set covering problem". Em: *Oper. Res. Lett.* 8, pp. 67–71 (ver p. 51).

Filho, G. R. e L. A. N. Lorena (2000). "Constructive genetic algorithm and column generation: an application to graph coloring". Em: *Proceedings the Fifth Conference of the Association of Asian-Pacific Operations Research Societies within IFORS*. Ed. por L. P. Chuen (ver p. 77).

Fischetti, M. e A. Lodi (2003). "Local branching". Em: *Math. Prog. Ser. B* 98, pp. 23–47 (ver p. 76).

Friedrich, T., J. He, N. Hebbinghaus, F. Neumann e C. Witt (2010). "Approximating covering problems by randomized search heuristics using multi-objective models". Em: *Evolutionary Computation* 18.4, pp. 617–633 (ver p. 66).

Fukunaga, A. S. (2008). "Automated discovery of local search heuristics for satisfiability testing". Em: *IEEE Trans. Evol. Comp.* 16.1, pp. 31–61 (ver p. 78).

Galinier, P., Z. Boujbel e M. C. Fernandes (2011). "An efficient memetic algorithm for the graph partitioning problem". Em: *Ann. Oper. Res.* 191.1 (ver p. 37).

Gandibleux, X., N. Mezdaoui, e A. Fréville (1997). "A tabu search procedure to solve multiobjective combinatorial optimization problems". Em: *Advances in Multiple Objective and Goal Programming*. Ed. por R. Caballero, F. Ruiz e R. Steuer. Vol. 445. LNEM, pp. 291–300 (ver p. 87).

Gendreau, M. e J.-Y. Potvin, eds. (2010). *Handbook of Metaheuristics*. 2nd. Springer. URL: http://www.springer.com/business+%26+management/operations+research/book/978-1-4419-1663-1 (ver pp. 73, 97, 101).

Glover, F. (1996). "Interfaces in Computer Science and Operations Research". Em: ed. por R. S. Barr, R. V. Helgason e J. L. Kennington. Kluwer. Cap. Tabu search and adaptive memory programing – Advances, applications and challenges, pp. 1–75 (ver p. 57).

Glover, F. (1986). "Future paths for integer programming and links to artificial intelligence". Em: *Comput. Oper. Res.* 13, pp. 533–549 (ver p. 35).

Glover, F. e G. A. Kochenberger, eds. (2002). *Handbook of metaheuristics*. INF 65.012.122 H237. Kluwer (ver p. 73).

Glover, F. e M. Laguna (1997). *Tabu Search*. Kluwer (ver pp. 16, 37, 43).

Goldberg, D. e R. Lingle (1985). "Alleles, loci and the Traveling Salesman Problem". Em: *Proceedings of 1st International Conference on Genetic Algorithms and their Applications*, pp. 154–159 (ver p. 56).

Golden, B. L., A. A. Assad, E. A. Wasil e E. Baker (1986). "Experimentation in optimization". Em: *Eur. J. Oper. Res.* 27, pp. 1–16 (ver p. 114).

Hajek, B. (1988). "Cooling schedules for optimal annealing". Em: *Mathematics of Operations Research* 13, pp. 311–329 (ver p. 33).

Hertz, A., E. Taillard e D. de Werra (2003). "Local Search in Combinatorial Optimization". Em: ed. por E. Aarts e J. K. Lenstra. Princeton University Press. Cap. Tabu search. ISBN: 978-0691115221. URL: http://press.princeton.edu/titles/7564.html (ver pp. 43, 99, 117).

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press (ver p. 62).

Hollander, M. e D. A. Wolfe (1999). *Nonparametric statistical methods*. 2nd. Wiley (ver p. 123).

Hoos, H. H. e T. Stützle (2004). *Stochastic Local Search : Foundations & Applications*. Morgan Kaufmann. ISBN: 978-1558608726. URL: http://www.sls-book.net (ver pp. 32, 42).

Hutter, M. (2010). "A complete theory of everything (will be subjective)". Em: *Algorithms* 3.4, pp. 329–350 (ver p. 11).

Hutter, M. e S. Legg (2006). "Fitness uniform optimization". Em: *IEEE Trans. Evol. Comp.* 10.5, pp. 568–589 (ver pp. 64, 66).

Jansen, T. e I. Wegener (2000). "Evolutionary Algorithms: How To Cope With Plateaus of Constant Fitness and When to Reject Strings of the Same Fitness". Em: *IEEE Transactions on Evolutionary Computation* 5.6, pp. 589–599 (ver p. 66).

Jaszkiewicz, A. e G. Dąbrowski (2005). *MOMH: Multiple-Objective MetaHeuristics*. URL: http://home.gna.org/momh (acedido em 04/06/2014) (ver p. 95).

Johnson, D. S., C. R. Aragon, L. A. McGeoch e C. Schevon (1989). "Optimization by Simulated Annealing. Part I, Graph Partitioning". Em: *Oper. Res.* 37, pp. 865–892 (ver pp. 33, 34).

Johnson, D. D., C. H. Papadimitriou e M. Yannakakis (1988). "How easy is local search?" Em: *J. Comput. Syst. Sci.* 37, pp. 79–100 (ver p. 27).

Johnson, D. S. e L. A. McGeoch (2003). "Local Search in Combinatorial Optimization". Em: ed. por E. Aarts e J. K. Lenstra. Princeton University Press. Cap. The traveling salesman problem: a case study. ISBN: 978-0691115221. URL: http://press.princeton.edu/titles/7564.html (ver pp. 16, 67).

Joslin, D. E. e D. P. Clements (1999). "Squeaky wheel optimization". Em: *Journal of Artificial Intelligence Research*, pp. 353–373 (ver p. 52).

Kellerer, H., U. Pferschy e D. Pisinger (2004). *Knapsack problems*. Springer (ver p. 8).

Kennedy, J. e R. C. Eberhart (1997). "A discrete binary version of the particle swarm algorithm". Em: *Conference on Systems, Man, and Cybernetics*, pp. 4104–4109 (ver p. 71).

Kirkpatrick, S., C. D. Gelatt e M. P. Vecchi (1983). "Optimization by simulated annealing". Em: *Science* 220, pp. 671–680 (ver p. 33).

Kleinberg, J. e E. Tardos (2005). *Algorithm design*. Addison-Wesley (ver p. 29).

Knust, S. (1997). *Optimality conditions and exact neighborhoods for sequencing problems*. Rel. téc. Universität Osnabrück (ver p. 29).

Konak, A., D. W. Coit e A. E. Smith (2006). "Multi-objective optimization using genetich algorithms: a tutorial". Em: *Reliability Engineering and System Safety* 91, pp. 992–1007 (ver p. 95).

Korte, B. e J. Vygen (2008). *Combinatorial optimization – Theory and Algorithms*. 4th. Springer (ver p. 29).

Lesh, N. e M. Mitzenmacher (2006). "BubbleSearch: a simple heuristic for improving priority-based greedy algorithms". Em: *Inf. Proc. Lett.* 97, pp. 161–169. DOI: 10.1016/j.ipl.2005.08.013 (ver p. 51).

LeVeque, R. J. (2013). "Top Ten Reasons To Not Share Your Code (and why you should anyway)". Em: *SIAM News* (ver p. 123).

Levine, D. (1997). "Genetic Algorithms: A Practitioner's View". Em: *INFORMS J. Comput.* 9, pp. 256–259 (ver p. 66).

Luke, S. (2011). *Essentials of Metaheuristics*. lulu.com. ISBN: 978-0557148592. URL: http://cs.gmu.edu/~sean/book/metaheuristics (ver p. 122).

Luque, G. e E. Alba (2011). *Parallel Genetic Algorithms: Theory and Real World Applications*. INF: Recurso eletrônico. Springer. ISBN: 978-3642220838. URL: http://link.springer.com/book/10.1007/978-3-642-22084-5/page/1 (ver p. 83).

Lustig, I. J., R. E. Marsten e D. F. Shanno (1991). "Computational experience with a primal-dual interior point method for linear programming". Em: *Linear algebra and its applications* 152, pp. 191–222. DOI: 10.1016/0024-3795(91)90275-2 (ver p. 116).

McCulloch, W. S. e W. Pitts (1943). "A logical calculus of ideas immanent in nervous activity". Em: *Bull. Math. Biophys.* 5, pp. 115–133 (ver p. 92).

Metropolis, N., A. Rosenbluth, M. Rosenbluth, A. Teller e E. Teller (1953). "Equation of state calculations by fast computing machines". Em: *Journal of Chemical Physics* 21, pp. 1087–1092 (ver p. 32).

Michiels, W., E. Aarts e J. Korst (2007). *Theoretical Aspects of Local Search*. INF: Recurso eletrônico. Springer. ISBN: 978-3-540-35853-4. URL: http://link.springer.com/book/10.1007/978-3-540-35854-1/page/1 (ver p. 29).

Montgomery, D. C. (2009). *Design and analysis of experiments*. 7th ed. Wiley (ver p. 118).

Murata, T., H. Ishibuchi e H. Tanaka (1996). "Multi-objective genetic algorithm and its applications to flowshop scheduling". Em: *Comput. Ind. Eng.* 30.4, pp. 957–968 (ver p. 88).

Nagata, Y. e S. Kobayashi (1997). "Edge assembly crossover: A high-power genetic algorithm for the traveling salesman problem". Em: pp. 450–457 (ver p. 57).

— (2012). "A powerful genetic algorithm using edge assembly crossover for the traveling salesman problem". Em: *INFORMS J. Comput.* DOI: /10.1287/ijoc.1120.0506 (ver pp. 67, 73).

Neumann, F. e I. Wegener (2006). "Randomized local search, evolutionary algorithms, and the minimum spanning tree problem". Em: *Nat. Comput.* 5.3, pp. 305–319 (ver p. 29).

Niedermeier, R. (set. de 2002). *Invitation to fixed-parameter algorithms*. Habilitationsschrift, Universität Tübingen, WSI für Informatik, Germany (ver p. 8).

NN (1938). "Economics in eight words". Em: *The Pittsburgh Press* (ver p. 6).

Papadimitriou, C. (1993). *Computational Complexity*. Addison-Wesley (ver p. 5).

Papadimitriou, C. H. e K. Steiglitz (mar. de 1977). "On the complexity of local search for the traveling salesman problem". Em: *SIAM J. Comput.* 6.1, pp. 76–83 (ver p. 23).

— (1982). *Combinatorial optimization: Algorithms and complexity*. Dover. Prentice-Hall (ver p. 29).

Pisinger, D. e S. Ropke (2010). "Handbook of Metaheuristics". Em: ed. por M. Gendreau e J.-Y. Potvin. 2nd. Springer. Cap. Large Neighborhood

Search. URL: http://www.springer.com/business+%26+management/operations+research/book/978-1-4419-1663-1 (ver p. 41).

Polya, G. (1945). *How to solve it.* Princeton University Press (ver p. 100).

Rad, S. F., R. Ruiz e N. Boroojerdian (2009). "New high performing heuristics for minimizing makespan in permutation flowshops". Em: *Omega* 37.2, pp. 331–345 (ver p. 107).

Reeves, C. R. (1993). "Using genetic algorithms with small populations". Em: *Inf. Conf. Genetic Algorithms* (ver p. 64).

Resende, M. G. C. e C. C. Ribeiro (2005). "Metaheuristics: Progress as Real Problem Solvers". Em: ed. por T. Ibaraki, K. Nonobe e M. Yagiura. Springer. Cap. GRASP with path-relinking: Recent advances and applications, pp. 29–63 (ver p. 59).

Ross, P., S. Schulenburg, J. G. Marin-Blázquez e E. Hart (2002). "Hyper-heuristics: learning to combine simple heuristics in bin-packing problem". Em: *Proceedings of the Genetic and Evolutionary Computation Conference* (ver p. 78).

Rothlauf, F. (2011). *Design of Modern Heuristics: Principles and Application.* INF: Recurso eletrônico. Springer. ISBN: 978-3540729617. URL: http://link.springer.com/book/10.1007/978-3-540-72962-4/page/1 (ver p. 11).

Ruiz, R. e T. Stützle (2006). "A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem". Em: *Eur. J. Oper. Res.* (Ver p. 52).

Scharnow, J., K. Tinnefeld e I. Wegener (2002). "Fitness landscapes based on sorting and shortest path problems". Em: *Proceedings of the Parallel Problem Solving from Nature conference VII.* Vol. 2939. LNCS, pp. 54–63 (ver p. 67).

Schöning, U. (1999). "A probabilistic algorithm for k-SAT and constraint satisfaction problems". Em: *Proc. 40th FOCS* (ver p. 19).

Sedgewick, R. (2010). *Algorithms for the masses* (ver p. 105).

Sedgewick, R. e K. Wayne (2011). *Algorithms.* 4th. Addison-Wesley (ver p. 105).

Selman, B., H. Levesque e D. Mitchell (1992). "A new method for solving hard satisfiability problems". Em: *Proc. 10th Nat. Conf. Artif. Intell.* Pp. 440–446 (ver p. 18).

Selman, B., H. Kautz e B. Cohen (1994). "Noise strategies for improving local search". Em: *Proc. 12th Nat. Conf. Artif. Intell.* Pp. 337–343 (ver p. 19).

Serafini, P. (1986). "Some considerations about computational complexity for multi objective combinatorial problems". Em: *Recent advances and historical developments of vector optimization.* Ed. por J. Jahn e W. Krabs. LNEM 294, pp. 222–232 (ver p. 85).

Silberholz, J. e B. Golden (2010). "Handbook of Metaheuristics". Em: ed. por M. Gendreau e J.-Y. Potvin. 2nd. Springer. Cap. Comparison of metaheuristics. URL: http://www.springer.com/business+%26+management/operations+research/book/978-1-4419-1663-1 (ver p. 122).

Sloane, N. *A001511*. URL: http://oeis.org/A001511 (ver p. 37).

Sörensen, K. (2013). "Metaheuristics – the metaphor exposed". Em: *Int. Trans. Oper. Res.* (Ver pp. 74, 98).

Souza, P. S. de e S. N. Talukdar (1993). "Asynchronous organizations for multi-algorithm problems". Em: *Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing*, pp. 286–293 (ver p. 82).

Steuer, R. (1986). *Multiple Criteria Optimization: Theory, Computation and Application*. Wiley (ver p. 87).

Suppapitnarm, A., K. A. Seffen, G. T. Parks e P. J. Clarkson (2000). "Simulated annealing: An alternative approach to true multiobjective optimization". Em: *Engineering Optimization* 33.1 (ver p. 86).

Talbi, E.-G. (2009). *Metaheuristics. From Design to Implementation*. Wiley. ISBN: 978-0-470-27858-1. URL: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470278587.html (ver pp. 11, 43, 73, 95).

Toth, P. e D. Vigo (2003). "The granular tabu search and its application to the vehicle routing problem". Em: *INFORMS J. Comput.* 15, pp. 333–346 (ver p. 16).

Tseng, F. T., E. F. S. Jr. e J. N. D. Gupta (2004). "An empricial analysis of integer programming formulations for the permutation flowshop". Em: *Omega* 32, pp. 285–293. DOI: 10.1016/j.omega.2003.12.001 (ver p. 114).

Ulungu, E. L., J. Teghem, P. H. Fortemps e D. Tuyttens (1999). "MOSA method: a tool for solving multiobjective combinatorial optimization problems". Em: *Journal of multi-criteria decision analysis* 8.4, pp. 221–236. DOI: 10.1002/(SICI)1099-1360(199907)8:4<221::AID-MCDA247>3.0.CO;2-O (ver p. 86).

Watson, J.-P., A. E. Howe e L. D. Whitley (2006). "Deconstructing Nowicki and Smutnicki's i-TSAB tabu search algorithm for the job-shop scheduling problem". Em: *Comput. Oper. Res.* 33.9, pp. 2623–2644 (ver pp. 42, 72, 97, 99).

Weyland, D. (2010). "A Rigorous Analysis of the Harmony Search Algorithm: How the Research Community can be Misled by a "Novel"Methodology". Em: *Int. J. of Applied Metaheuristic Computing* 1.2, pp. 50–60. DOI: 10.4018/jamc.2010040104 (ver p. 74).

Witt, C. (2005). "Worst-Case and Average-Case Approximations by Simple Randomized Search Heuristics". Em: *Proc. of the 22nd Annual Sympo-*

sium on Theoretical Aspects of Computer Science. Vol. 3404. LNCS (ver p. 66).

— (2008). "Population size versus runtime of a simple evolutionary algorithm". Em: *Theor. Comput. Sci.* 403.1, pp. 104–120. DOI: 10.1016/j.tcs.2008.05.011 (ver p. 67).

Wolpert, D. H. e W. G. Macready (1997). "No Free Lunch Theorems for Optimization". Em: *IEEE Trans. Evol. Comp.* Pp. 67–82 (ver pp. 7, 11).

Wolsey, L. A. (1980). "Heuristic analysis, linear programming and branch and bound". Em: *Math. Prog. Stud.* 13, pp. 121–134 (ver p. 31).

Yannakakis, M. (2003). "Local Search in Combinatorial Optimization". Em: ed. por E. Aarts e J. K. Lenstra. Princeton University Press. Cap. Computational complexity. ISBN: 978-0691115221. URL: http://press.princeton.edu/titles/7564.html (ver p. 29).

— (2009). "Equilibria, Fixed Points, and Complexity Classes". Em: *Comput. Sci. Rev.* 3.2, pp. 71–85 (ver p. 29).

Índice