Algoritmos e Otimização Combinatória

Carla Negri Lintzmayer

Centro de Matemática, Computação e Cognição, Universidade Federal do ABC

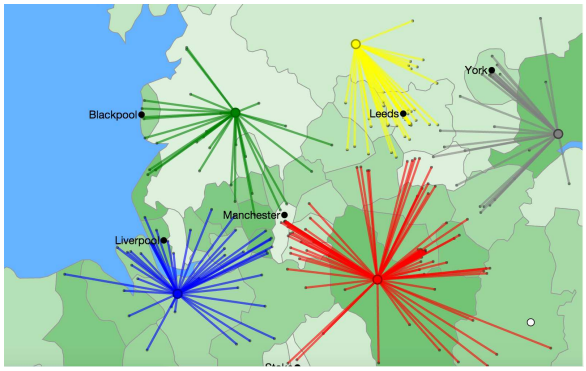
(Versão de 10 de janeiro de 2020)

Problemas em otimização combinatória têm como objetivo encontrar a melhor solução dentro de um enorme mas finito conjunto de soluções possíveis. O que caracteriza uma melhor solução varia de problema para problema e em geral considera custos ou lucros envolvidos. Em geral, um problema desses possui um conjunto de restrições que define o que é uma solução possível e uma função objetivo que determina o valor de cada solução. Assim, a melhor solução é uma solução que minimiza ou maximiza o valor da função objetivo. Essas características são inerentes a vários problemas importantes e provenientes de diversas áreas. Como exemplo podemos citar alocação de recursos, corte de materiais, roteamento em redes, localização de centros de distribuição, montagem de fragmentos de DNA, entre tantos outros. Veja a Figura 1.

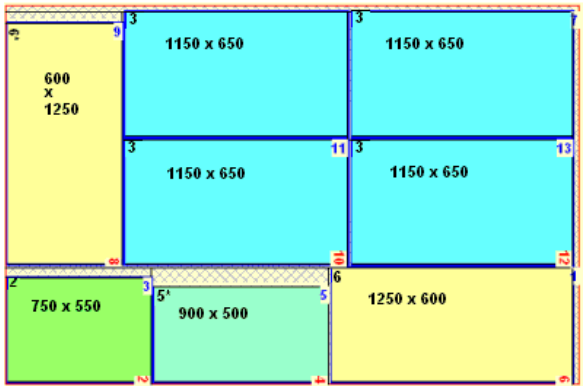
Apesar do conjunto de soluções possíveis ser finito, encontrar a melhor delas usando algoritmos ingênuos ou mesmo enumerando todas as soluções tende a não ser algo possível na prática. Assim, são necessárias técnicas mais elaboradas para encontrar soluções ótimas ou então soluções boas o suficiente em tempo razoável. De forma geral, existem três abordagens principais para tratá-los. Em uma delas desenvolvemos algoritmos exatos (Seção 3.1), que encontram uma solução ótima, mas no pior caso requerem tempo exponencial no tamanho da entrada. Outra abordagem consiste em desenvolver algoritmos que buscam soluções de boa qualidade mas não dão garantia sobre o valor da mesma, como heurísticas e metaheurísticas (Seção 3.2). A terceira abordagem é o desenvolvimento de algoritmos de aproximação (Seção 3.3), que são eficientes e produzem uma solução cujo valor está dentro de um fator do valor de uma solução ótima.

A Seção 1 define alguns termos e notações que serão usadas no decorrer do texto. A Seção 2 descreve algumas técnicas gerais de projeto de algoritmos. A Seção 3 descreve as três técnicas mencionadas acima.

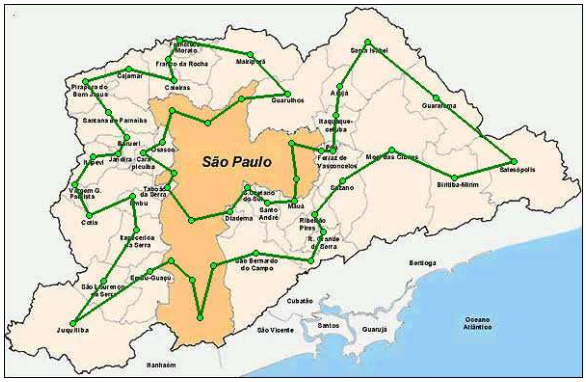
Dados os possíveis locais de abertura de centro de distribuição com os custos para abri-los e os locais dos clientes, decidir onde abrir os centros de forma a minimizar gastos de abertura e distâncias entre os clientes e os centros.



Dadas as dimensões de cada pedaço de vidro que irá compor as janelas de um prédio, corte tais pedaços placas de vidro de tamanho fixo desperdiçando o mínimo possível da placa.



Dada uma lista de cidades que se deseja visitar, planeje uma rota que sai da cidade atual, visita todas as cidades uma única vez, volta para a cidade de origem e que tenha o menor percurso total.



Dada uma região retangular do website que irá mostrar propagandas e uma lista de propagandas com os respectivos tamanhos e lucros, escolher quais propagandas cabem no espaço disponível de forma a se obter o maior lucro possível.

Lorem ipsum per vulputate malesuada ullamcorper viverra sollicitudin risus sapien, torquent turpis metus ultricies rutrum pretium sollicitudin per aliquam, elit dapibus euismod quis mollis odio platea morbi. nam molestie tortor accumsan eros viverra nulla, nibh sodales maecenas sapien felis, mattis libero id fusce porta. consequat nisi semper neque nam tincidunt congue dolor elementum malesuada, netus orci nulla vulputate aenean ante iaculis imperdiet conubia lorem, hendrerit habitasse aliquam quis scelerisque pharetra mauris aliquet. erat a tincidunt vitae tristique hendrerit lacus etiam elit habitasse, pharetra elementum cubilia sapien facilisis egestas curabitur semper, placerat ut rhoncus metus donec inceptos potenti sed. Donec conubia phasellus cubilia luctus curae dictum est quisque at proin nam justo, tempus condimentum morbi mi rutrum pellentesque non pharetra habitant turpis. feugiat sem faucibus donec fringilla maecenas molestie ultricies aenean, imperdiet lacus iaculis vel mi scelerisque venenatis vivamus, porttitor integer etiam molestie porttitor condimentum rhoncus. facilisis placerat nam hendrerit convallis curabitur aenean aliquam aenean, at

Figura 1: Exemplos informais de alguns problemas de otimização.

1 Definições importantes

Consideraremos que um problema de otimização combinatória P consiste de um conjunto $\mathcal{I}_P$ de instâncias, uma função objetivo $val_P \colon \mathcal{S}(I) \to \mathbb{R}$ onde $\mathcal{S}(I)$ é o conjunto de *soluções viáveis* da instância $I \in \mathcal{I}_P$, e um indicador de objetivo (minimização ou maximização). Uma solução $S^* \in \mathcal{S}(I)$ é dita *ótima* se para toda solução $S \in \mathcal{S}(I)$ temos que $val_P(S^*) \leq val_P(S)$ e o problema é de minimização ou se $val_P(S^*) \geq val_P(S)$ e o problema é de maximização. Denotamos por $OPT_P(I)$ o valor de uma solução ótima para P, isto é, $OPT_P(I) = val_P(S^*)$ para uma solução ótima S^*. Veja a Figura 2 para vários exemplos de problemas de otimização (ela formaliza os problemas descritos na Figura 1).

Seja P um problema e seja A_P um algoritmo que recebe uma instância de $\mathcal{I}_P$ como entrada. Denotamos por $A_P(I)$ a solução produzida por A_P quando recebe $I \in \mathcal{I}_P$. Se $A_P(I) \in \mathcal{S}(I)$ para todo $I \in \mathcal{S}(I)$, então dizemos que esse algoritmo *trata* o problema P. Se A_P trata P e $A_P(I)$ é uma solução ótima para todo $I \in \mathcal{S}(I)$, então dizemos que esse algoritmo *resolve* o problema P.

Outros conceitos importantes que são utilizados o tempo todo na pesquisa em otimização combinatória são:

- Análise de algoritmos (para verificar se um algoritmo resolve um problema e, em caso positivo, em quanto tempo ele o faz);
- Complexidade (para verificar se não há esperanças em se resolver um problema em tempo polinomial – classes P, NP, NP-difícil, NP-completo);
- Teoria dos grafos (muitos problemas interessantes podem ser modelados em grafos e precisam de conhecimentos estruturais para serem tratados).

Problema de Localização de Centros de Distribuição (Facility Location)
Entrada: um grafo completo G com custos $c(e)$ para toda aresta $e \in E(G)$, um conjunto $F \subseteq V(G)$ de vértices com custos $f(v)$ para todo vértice $v \in F$, e uma demanda $d(v)$ para todo vértice $v \in V(G)$.
Solução viável: locais para construir os centros de distribuição, isto é, $F' \subseteq F$.
Função objetivo: custo de abrir os centros de F' mais o custo de atender as demandas, isto é,

$$\sum_{v \in F'} f(v) + \sum_{v \in F'} \sum_{u \in V(G)} d(u)c(uv)\,.$$

Objetivo: minimização.

Problema de Empacotamento
Entrada: um conjunto $L = \{1, \ldots, n\}$ de itens retangulares em que cada item i tem largura w_i e altura h_i, e dois números W e H que indicam a largura e altura, respectivamente, de um recipiente retangular.
Solução viável: partição $L_1, \ldots, L_q$ de L tal que os itens em L_j cabem completamente em um recipiente $W \times H$ e não se sobrepõem.
Função objetivo: quantidade de recipientes utilizados (q).
Objetivo: minimização.

Problema do Caixeiro Viajante
Entrada: um grafo G completo com custos $w(e)$ para toda aresta $e \in E(G)$.
Solução viável: ciclo hamiltoniano em G (um passeio fechado que passa por todos os vértices de G uma única vez).
Função objetivo: soma dos custos das arestas no ciclo.
Objetivo: minimização.

Problema da Mochila
Entrada: conjunto de n itens $\{1, \ldots, n\}$, cada um com um peso w_i e um valor v_i, e um peso W.
Solução viável: subconjunto de itens $S \subseteq \{1, \ldots, n\}$ tais que $\sum_{i \in S} w_i \leq W$.
Função objetivo: valor dos itens escolhidos, isto é, $\sum_{i \in S} v_i$.
Objetivo: maximização.

Figura 2: Abstração e formalização dos problemas apresentados na Figura 1.

2 Projeto de algoritmos

As três principais técnicas de projeto de algoritmos descritas a seguir são a base para qualquer pesquisa em algoritmos e otimização combinatória.

Divisão e conquista. A divisão e conquista é uma técnica de projeto de algoritmos que faz uso de recursão. De forma geral, ela consiste em dividir o problema em subproblemas menores, "conquistar" (resolver) os subproblemas por meio de recursão e combinar as soluções dos subproblemas em uma solução para o problema original. Algoritmos clássicos de ordenação, como Mergesort e Quicksort, fazem uso dessa técnica. Um outro exemplo legal e menos trivial é o algoritmo de Karatsuba para multiplicação de inteiros.

Algoritmos gulosos. Um algoritmo guloso é aquele que constrói uma solução através de uma sequência de decisões que visam o melhor cenário de curto prazo, sem garantia de que isso levará ao melhor resultado global. Algoritmos gulosos são muito usados porque costumam ser rápidos e fáceis de implementar, além de em alguns casos ser possível mostrar que suas soluções apresentam algum fator de aproximação. Em geral, é fácil descrever um algoritmo guloso que forneça uma solução viável e tenha complexidade de tempo fácil de ser analisada. A dificuldade normalmente se encontra em provar se a solução obtida é de fato ótima. Na maioria das vezes, inclusive, elas não são ótimas, mas em alguns casos é possível mostrar que elas têm valor próximo ao de uma solução ótima. Algoritmos clássicos em grafos como Kruskal e Dijkstra fazem uso dessa técnica.

Programação dinâmica. Programação dinâmica também é uma técnica de projeto de algoritmos que faz uso de recursão. Assim, como na divisão e conquista, um problema gera subproblemas que serão resolvidos recursivamente. Porém, quando a solução de um subproblema precisa ser utilizada várias vezes em um algoritmo de divisão e conquista, a programação dinâmica pode ser uma eficiente alternativa no desenvolvimento de um algoritmo para o problema. Isso porque a característica mais marcante da programação dinâmica é evitar resolver o mesmo subproblema diversas vezes. Para isso, os algoritmos fazem uso de memória extra para armazenar as soluções dos subproblemas. Algoritmos clássicos em grafos como Bellman-Ford e Floyd-Warshall fazem uso dessa técnica.

3 Outras técnicas de projeto de algoritmos

As técnicas que veremos nessa seção serão exemplificadas por meio de algoritmos para o problema da MOCHILA, que é definido por:

- **Entrada:** conjunto de n itens $\{1, \ldots, n\}$, cada um com um peso w_i e um valor v_i, e um peso W.
- **Solução viável:** subconjunto de itens $S \subseteq \{1, \ldots, n\}$ tais que $\sum_{i \in S} w_i \leq W$.
- **Função objetivo:** valor dos itens escolhidos, isto é, $\sum_{i \in S} v_i$.
- **Objetivo:** maximização.

3.1 Algoritmos exatos

Apesar da maioria dos problemas de otimização combinatória estarem, em NP-difícil, ainda podemos considerar algoritmos exatos para os mesmos. Mesmo executando em tempo não polinomial no pior caso, podem funcionar bem na prática.

Um algoritmo que enumera todas as soluções possíveis, verifica se são viáveis e calcula seus custos com a função objetivo é chamado de *algoritmo de força bruta*. De fato, eles são bem simples e encontram a solução ótima, uma vez que passam por todas as soluções possíveis. No entanto, utilizam muito esforço computacional para encontrar uma solução ótima e ignoram quaisquer estruturas combinatórias do problema.

Alguns problemas admitem soluções exatas por algoritmos gulosos ou de programação dinâmica. No entanto, pode ser que não o façam em tempo polinomial.

Uma outra possibilidade para o projeto de algoritmos exatos é utilizar a modelagem através da *programação linear*. Programação linear tem um papel central no projeto de algoritmos de aproximação, uma vez que diversas técnicas de projeto para tais algoritmos, bem como para análise de qualidade dos mesmos, têm como ponto de partida a formulação do problema de otimização combinatória como um programa linear inteiro.

Um *programa linear* possui um conjunto de variáveis de decisão, um conjunto de restrições e uma função objetivo. Cada restrição corresponde a uma desigualdade (ou equação) linear sobre as variáveis de decisão. Dizemos que uma atribuição de valores para as variáveis de decisão é uma solução viável se todas as restrições são satisfeitas. O valor de uma solução é dado pela função objetivo, que é uma expressão linear sobre as variáveis de decisão. Em um problema de minimização, uma solução é ótima se tem valor mínimo dentre todas as soluções viáveis. A forma padrão de um programa linear para um problema de minimização que tem n variáveis e m restrições é

$$
\begin{aligned}
\text{minimizar} \quad & \sum_{j=1}^{n} c_j x_j & \\
\text{sujeito a} \quad & \sum_{j=1}^{n} a_{ij} x_j \geq b_i & \forall i \in \{1, \ldots, m\} \\
& x_j \geq 0 & \forall j \in \{1, \ldots, n\}
\end{aligned}
$$

onde a_{ij}, b_i e c_j são constantes, x_j são as variáveis de decisão, $\sum_{j=1}^{n} a_{ij} x_j \geq b_i$ são restrições e $\sum_{j=1}^{n} c_j x_j$ é a função objetivo.

Do programa linear acima, chamado de *primal*, pode-se obter um programa linear relacionado, chamado de *dual*:

$$
\begin{aligned}
\text{maximizar} \quad & \sum_{i=1}^{m} b_i y_i & \\
\text{sujeito a} \quad & \sum_{i=1}^{m} a_{ij} y_i \leq c_j & \forall j \in \{1, \ldots, n\} \\
& y_i \geq 0 & \forall i \in \{1, \ldots, m\}
\end{aligned}
$$

onde cada variável y_i está associada a uma restrição do primal, e cada restrição do dual está associada a uma variável do primal.

Um *programa linear inteiro* é definido de modo semelhante a um programa linear, com a restrição adicional de que suas variáveis de decisão devem receber valores inteiros, o que é o caso da maioria dos problemas de otimização combinatória. Esta diferença é bastante relevante, uma vez que são conhecidos algoritmos eficientes para resolver programas lineares, enquanto que resolver programas inteiros também é um problema NP-difícil [4]. Apesar disso, existem implementações eficientes dos métodos *branch-and-bound*, *branch-and-cut*, *branch-and-price* e *branch-and-cut-and-price*, que obtêm bons resultados em situações reais para programas lineares inteiros.

Outra opção no projeto de algoritmos exatos relativamente recente são os algoritmos *parametrizados* [1]. Neste caso, considera-se parâmetros que capturam a dificuldade de um determinado problema. Algoritmos parametrizados são aqueles que resolvem o problema em tempo polinomial sobre o tamanho da entrada mas provavelmente não polinomial sobre esse parâmetro. Se um problema admite tal algoritmo, dizemos que ele é *tratável com parâmetro fixo* e que tal é algoritmo é um FPT (do inglês, *fixed-parameter tractable*).

Do ponto de vista prático, é interessante projetar algoritmos FPT, pois em muitos casos o parâmetro das instâncias é pequeno. Do ponto de vista teórico, é necessário compreender bem a estrutura combinatória do problema, o que pode levar a novos algoritmos para o mesmo ou então ao desenvolvimento de novas técnicas para outros problemas.

3.1.1 Algoritmo exato de força bruta para Mochila

Esse algoritmo enumera todos os subconjuntos possíveis de itens, verifica se eles cabem na mochila, calcula o valor total deles e compara com o melhor subconjunto encontrado até o momento. Note que existem 2^n subconjuntos diferentes de itens pois, para cada item, temos a opção de colocá-lo ou não em um subconjunto. Para cada subconjunto, levamos tempo $O(n)$ para checar se os itens cabem na mochila e calcular seu valor total. Ou seja, esse algoritmo leva tempo $O(n2^n)$ e, portanto, não é polinomial no tamanho da entrada.

3.1.2 Formulação PLI da Mochila

Sejam x_i variáveis binárias que indicam se o item i foi escolhido ou não ($x_i \in \{0, 1\}$). Considere o seguinte programa linear inteiro:

$$\begin{aligned} \text{maximizar} \quad & \sum_{i=1}^{n} v_i x_i \\ \text{sujeito a} \quad & \sum_{i=1}^{n} w_i x_i \leq W \\ & x_i \in \{0, 1\} \qquad \forall i \in \{1, \ldots, n\} \end{aligned}$$

Note que ele formula o problema da mochila, pois tenta maximizar a soma dos valores dos itens escolhidos ($\sum_{i=1}^{n} v_i x_i$) enquanto atende à restrição de que os itens escolhidos devem caber na mochila ($\sum_{i=1}^{n} w_i x_i \leq W$).

3.1.3 Algoritmo exato de programação dinâmica para Mochila

Para facilitar a discussão a seguir, vamos dizer que uma instância da mochila inteira é dada pelo par (n, W), que indica que temos n itens e capacidade W de mochila e deixa os valores e pesos dos itens escondidos. Podemos tentar uma abordagem recursiva para construir uma solução S para (n, W) da seguinte forma. Note que um determinado item i pode ser utilizado ou não na sua solução. Se você decidir por não utilizá-lo, então a capacidade da mochila não se altera e você pode usar a recursão para encontrar uma solução S' para $(n-1, W)$. Assim, $S = S'$ é uma solução para (n, W). Se você decidir por utilizá-lo, então a capacidade da mochila reduz de w_i unidades, mas você também pode usar a recursão para encontrar uma solução S' para $(n-1, W-w_i)$ e usar $S = S' \cup \{i\}$ como solução para (n, W). Veja que um caso base simples aqui seria aquele em que não temos nenhum item para escolher. Independente do tamanho da mochila, não é possível obter nenhum valor.

Mas qual decisão tomar? Escolhemos o item i ou não? Veja que são apenas duas

possibilidades: colocamos i na mochila ou não. Podemos então tentar ambas e devolver a melhor opção das duas. Portanto, se $V_{n,W}$ é o valor de uma solução para (n, W), então

$$V_{n,W} = \begin{cases} \max\{V_{n-1,W}, V_{n-1,W-w_i} + v_i\} & \text{se } w_n \leq W \\ V_{n-1,W} & \text{se } w_n > W \end{cases} \quad (1)$$

E qual item i escolher dentre os n disponíveis? Uma opção simples seria fazer uma escolha gulosa, pelo item com melhor razão v_i/w_i ou então pelo item com maior v_i ou mesmo pelo item com menor w_i. É possível construir contraexemplos que mostram que essas escolhas não dariam soluções ótimas sempre. Mas note que pela recursividade da estratégia, escolher um item i qualquer apenas o remove da instância da chamada atual, deixando qualquer outro item j como possibilidade de escolha para as próximas chamadas. Note também que qualquer solução está descrita em algum caminho que vai da raiz até uma folha dessa árvore. Por isso, podemos escolher qualquer item i que quisermos. Essa análise também mostra que o algoritmo de fato encontra uma solução ótima, uma vez que analisa todas as possíveis. Por comodidades que facilitam a implementação, vamos escolher $i = n$. O algoritmo recursivo descrito é formalizado no Algoritmo 1.

Algoritmo 1: MOCHILA(n, v, w, W)

```
1 se n == 0 então
2     retorna 0
3 se w_n > W então
4     retorna MOCHILA(n − 1, v, w, W)
5 senão
6     usa = v_n + MOCHILA(n − 1, v, w, W − w_n)
7     naousa = MOCHILA(n − 1, v, w, W)
8     retorna max{usa, naousa}
```

Não é difícil perceber que o tempo de execução $T(n)$ de MOCHILA é, no pior caso, descrito pela recorrência $T(n) = 2T(n-1) + \Theta(1)$, cuja solução é $O(2^n)$. Também não é difícil perceber que o problema desse algoritmo está no fato de ele realizar as mesmas chamadas recursivas diversas vezes. Assim, existem no máximo $(n+1) \times (W+1)$ subproblemas diferentes: um subproblema é totalmente descrito por (j, x), onde $0 \leq j \leq n$ e $0 \leq x \leq W$. Por isso, podemos usar uma estrutura de dados para manter seus valores e acessá-los diretamente sempre que necessário ao invés de recalculá-los. Poderíamos utilizar um vetor com $(n+1) \times (W+1)$ entradas, uma para cada subproblema, porém utilizar uma matriz de dimensões $(n+1) \times (W+1)$ nos permite um acesso mais intuitivo. A ideia é armazenar em $M[j][x]$ o valor $V_{j,x}$, de forma que nosso objetivo é calcular $M[n][W]$. O

Algoritmo 2 formaliza a ideia dessa estratégia de programação dinâmica com a abordagem top-down enquanto que o Algoritmo 4 o faz com a abordagem bottom-up.

Algoritmo 2: MOCHILA-TOPDOWN(n, v, w, W)

```
1 Seja M[0..n][0..W] uma matriz global
2 para x = 0 até W faça
3     M[0][x] = 0
4     para j = 1 até n faça
5         M[j][x] = −1
6 retorna MochilaRecursivo-TopDown(n, v, w, W)
```

Algoritmo 3: MOCHILARECURSIVO-TOPDOWN(j, v, w, x)

```
1 se M[j][x] == −1 então
2     se w_j > x então
3         M[j][x] = Mochila(j − 1, v, w, x)
4     senão
5         usa = v_j + Mochila(j − 1, v, w, x − w_j)
6         naousa = Mochila(j − 1, v, w, x)
7         M[j][x] = max{usa, naousa}
8 retorna M[j][x]
```

Algoritmo 4: MOCHILA-BOTTOMUP(n, v, w, W)

```
1  Seja M[0..n][0..W] uma matriz
2  para x = 0 até W faça
3      M[0][x] = 0
4  para j = 1 até n faça
5      para x = 0 até W faça
6          se w_j > x então
7              M[j][x] = M[j − 1][x]
8          senão
9              usa = v_j + M[j − 1][x − w_j]
10             naousa = M[j − 1][x]
11             M[j][x] = max{usa, naousa}
12 retorna M[n][W]
```

3.2 Heurísticas e metaheurísticas

Heurísticas são algoritmos que executam em tempo polinomial e procuram soluções de boa qualidade para um determinado problema, dentro das limitações dos recursos existentes para sua obtenção (tempo ou memória, por exemplo). Para muitos problemas de natureza prática elas obtêm bons resultados com a vantagem de computar uma solução rapidamente. Elas podem ser *construtivas*, que normalmente adotam estratégias gulosas para construir soluções, ou *de busca local*, que partem de uma solução inicial e tentam modificá-la, melhorando-a, até atingir algum critério de parada.

Metaheurísticas são estratégias de alto nível que não necessariamente são definidos para problemas específicos e que podem ser aplicadas a vários problemas. Elas também procuram soluções de boa qualidade mas não dão garantias com relação ao custo da solução encontrada. Podem ser usadas como passo intermediário para resolver subproblemas ou encontrar soluções iniciais para desenvolver algoritmos exatos. Exemplos clássicos são os próprios algoritmos de aproximação e as metaheurísticas GRASP (do inglês *Greedy Randomized Adaptive Search Procedure*), BRKGA (do inglês *Biased Random-Key Genetic Algorithm*), *Simulated Annealing*, Colônia de Formigas e Algoritmos Genéticos [3, 2].

3.2.1 Heurística construtiva para Mochila

O Algoritmo 5 apresenta uma heurística construtiva para o problema da Mochila. Ele se baseia no clássico algoritmo guloso para o problema Mochila Fracionária, em que podemos tomar partes de um item ao invés de pegá-lo por inteiro. Intuitivamente, em ambos problemas, o que queremos é escolher itens de maior valor que ao mesmo tempo tenham pouco peso, isto é, que tenham melhor custo-benefício. Assim, uma estratégia gulosa é sempre escolher o item com a maior razão v/w (valor/peso), enquanto ele couber na mochila. O que o algoritmo Mochila faz então é tomar os q primeiros itens com maior razão v/w que cabem na mochila. Essa estratégia rende uma solução ótima para Mochila Fracionária, porém não rende uma solução ótima para Mochila. Por exemplo, veja o que acontece quando temos $n = 2$, $v_1 = 2$, $w_1 = 1$, $v_2 = 1000$, $w_2 = 1000$ e $W = 1000$.

Algoritmo 5: Mochila(n, v, w, W)

1 Ordene e renomeie os itens para que $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}$

2 Seja q um inteiro tal que $\sum_{i=1}^{q} w_i \leq W$ e $\sum_{i=1}^{q+1} w_i > W$

3 **retorna** $v_1 + v_2 + \ldots + v_q$

3.3 Algoritmos de aproximação

Intuitivamente, dizemos que um algoritmo é de aproximação se, para cada instância de um determinado problema, ele devolve uma solução cujo valor está garantidamente a um fator de distância do valor de uma solução ótima para aquela instância. No que segue, seja P um problema qualquer e A_P um algoritmo que trata P.

Dizemos que A_P é um *algoritmo de aproximação* para P se existe um valor $\alpha > 0$, chamado *fator de aproximação*, tal que

- $val_P(A_P(I)) \leq \alpha OPT_P(I)$ para todo $I \in \mathcal{I}_P$ e o problema é de minimização; ou
- $val_P(A_P(I)) \geq \alpha OPT_P(I)$ para todo $I \in \mathcal{I}_P$ e o problema é de maximização.

Nesse caso, dizemos também que A_P é uma *α-aproximação* para P.

Dizemos P admite um *esquema de aproximação em tempo polinomial* (PTAS, de *polynomial time approximation scheme*) se para toda constante $\epsilon > 0$ existe um algoritmo de tempo polinomial em no tamanho n da entrada com fator de aproximação $(1 + \epsilon)$. Se o tempo de execução do algoritmo é polinomial em ambos n e $1/\epsilon$, então dizemos que o problema admite um *esquema de aproximação em tempo completamente polinomial* (FPTAS, de *fully polynomial time approximation scheme*). Assumindo que P $\neq$ NP, um PTAS é o melhor resultado que podemos obter para um problema fortemente NP-difícil.

Quando $val_P(A_P(I)) \leq \alpha OPT_P(I) + \delta$ para alguma constante δ independente de I e para todo $I \in \mathcal{I}_P$, dizemos que A_P é um *algoritmo de aproximação assintótica* para P, ou que A_P é uma *α-aproximação assintótica*. Dizemos ainda que o problema admite um *esquema assintótico de aproximação em tempo polinomial* (APTAS, de *asymptotic polynomial time approximation scheme*) se para todo $\epsilon > 0$ existe um algoritmo de tempo polinomial em n com fator assintótico de aproximação $(1 + \epsilon)$. De forma equivalente, se o tempo de execução é polinomial em n e em $1/\epsilon$ dizemos que o problema admite um *esquema assintótico de aproximação em tempo completamente polinomial* (AFPTAS, de *asymptotic fully polynomial time approximation scheme*).

Utilizar diferentes técnicas de projeto para obter algoritmos para um determinado problema e buscar garantias de qualidade para as soluções destes, nos ajudam a entender melhor a estrutura e as dificuldades típicas do problema, bem como as peculiaridades e limitações de cada técnica, possibilitando o surgimento de novas ideias de projeto que resultam em algoritmos não triviais e, muitas vezes, bem sucedidos na prática [5]. Outras análises importantes nessa área envolvem encontrar bons limitantes inferiores para o fator de aproximação de qualquer algoritmo que venha a ser desenvolvido para um determinado problema ou mesmo definir se um certo problema pode admitir algum algoritmo de aproximação.

O fator de aproximação de um algoritmo é calculado com base em uma análise de pior caso. Assim, é natural que as soluções encontradas na prática sejam muito melhores do que o esperado, já que muitas vezes o pior caso ocorre em instâncias com estruturas muito específicas que são raras na prática. Por isso, é comum também que algoritmos de aproximação sejam inspiração para heurísticas que retornem bons resultados na prática [6].

Apesar de não haver uma fórmula específica para criação de algoritmos de aproximação, existem algumas técnicas bem conhecidas. Dado um programa linear inteiro que modela um problema de otimização combinatória, podemos relaxá-lo, removendo as restrições de integralidade, obtendo um programa linear. Por vezes, é possível derivar um algoritmo de aproximação a partir da solução ótima deste programa linear, transformando tal solução fracionária em uma solução inteira viável, através do *arredondamento* das variáveis de decisão por algum critério de forma a respeitar as restrições do programa inteiro. Feito este arredondamento, é necessário encontrar uma relação entre o valor da solução inteira criada e o valor da solução fracionária original. Como o valor da solução fracionária é um limite inferior do valor da solução ótima do problema, a partir dessa relação é possível calcular a razão de aproximação do algoritmo obtido.

Outro método conhecido é chamado *primal-dual*. Como existe uma correspondência entre as restrições do dual e as variáveis do primal, uma ideia é aumentar o valor de certas variáveis duais até que alguma restrição fique justa, i.e., se torne uma igualdade. Quando isto acontece, a variável de decisão primal correspondente recebe valor 1. Algoritmos projetados assim sempre mantêm uma solução dual viável e executam até conseguirem obter uma solução primal viável. A razão de aproximação é obtida pela comparação entre o custo da solução de ambas, uma vez que qualquer solução dual viável é um limitante para a solução ótima do primal. Um método relacionado, chamado *dual-fitting*, também explora a relação entre o programa primal e seu dual. No entanto, ele se permite violar a viabilidade da solução dual sendo construída. Neste caso, para obter a razão de aproximação é necessário reescalar a solução dual de modo a torná-la viável.

3.3.1 Algoritmo de aproximação para Mochila

O Algoritmo 6 apresenta um algoritmo que trata o problema da Mochila de forma muito semelhante à heurística apresentada na Seção 3.2.1. A ideia também é considerar a razão v/w (valor/peso) na escolha dos itens. A diferença está apenas em verificar o que é melhor: tomar os q primeiros itens com maior razão v/w que cabem na mochila ou tomar apenas o $(q+1)$-ésimo item de maior razão v/w?

Essa pequena estratégia, que não parece nada intuitiva, garante que para qualquer instância de entrada, a solução devolvida por Mochila-Aproximacao será no máximo 2

Algoritmo 6: MOCHILA-APROXIMACAO(n, v, w, W)

1 Ordene e renomeie os itens para que $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$

2 Seja q um inteiro tal que $\sum_{i=1}^{q} w_i \leq W$ e $\sum_{i=1}^{q+1} w_i > W$

3 **retorna** $\max\{v_1 + v_2 + \ldots + v_q, v_{q+1}\}$

vezes pior do que a solução ótima. Isso é verdade porque

$$\begin{aligned} val(\text{MOCHILA-APROXIMACAO}(I)) &= \max\{v_1 + v_2 + \ldots + v_q, v_{q+1}\} && (2) \\ &\geq \frac{1}{2}(v_1 + \cdots + v_q + v_{q+1}) && (3) \\ &\geq \frac{1}{2} OPT_{MochilaFrac}(I) && (4) \\ &\geq \frac{1}{2} OPT_{Mochila}(I)\,, && (5) \end{aligned}$$

onde (2) vale pelo funcionamento do algoritmo, (3) vale porque o máximo entre dois valores é sempre maior ou igual à média desses valores, (4) vale porque tomar os q primeiros itens com maior razão v/w que cabem na mochila e tomar uma fração do $(q+1)$-ésimo item é igual à solução ótima para MOCHILA FRACIONÁRIA e, finalmente, (5) vale porque uma solução ótima para MOCHILA é uma solução viável para MOCHILA FRACIONÁRIA (e uma solução ótima em um problema de maximização sempre tem valor maior do que qualquer solução viável). Formalizando, temos que MOCHILA-APROXIMACAO é uma (1/2)-aproximação para MOCHILA.

Referências

[1] R. G. Downey and M. R. Fellows. *Fundamentals of Parameterized Complexity*, volume 4. Springer, 2013.

[2] M. Gendreau and J.-Y. Potvin. *Handbook of metaheuristics*. Springer, 3 edition, 2010.

[3] R. Martí, P. M. Pardalos, and M. G. C. Resende. *Handbook of Heuristics*. Springer, 2018.

[4] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1998.

[5] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag, Berlin, Heidelberg, 2001.

[6] D. P. Williamson and D. B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, New York, NY, USA, 1 edition, 2011.