1 O Problema do General Bárbaro

Para ver se realmente compreendemos as ideias que vimos anteriormente, é sempre útil ver se conseguimos aplicá-las a algo concreto. Para isso, desenvolveremos aqui um problema específico (o "problema do general bárbaro"[*]). Mas faremos isso não com vistas em colocá-lo diretamente no *framework* desenvolvido até agora, mas sim em entendê-lo, pensar em diversos métodos de resolvê-lo e comparar esses métodos.

1.1 O Problema

O general romano Sulla, em suas campanhas pela Grécia, fez (entre outras coisas) o que qualquer outro general bárbaro faria em seu lugar, isto é, saqueou tesouros locais para obter vantagem na guerra (diferente de outros generais bárbaros, Sulla prometeu devolver parte do saque). Mas isso constituiu um problema, já que ele tinha uma grande variedade de tesouros para escolher levar ou não, enquanto que a capacidade de carga de seu exército era muito limitada, umas vez que dispunha de (relativamente) poucos navios. Mas o que criava a maior dificuldade nesse projeto era não o volume dos tesouros, mas o seu peso.

O valor de cada peça de tesouro é dado por uma função complicada de interesses internos e externos (no que influenciam, entre outros fatores, o preço pelo qual Sulla poderia vender dado item, no valor que cada soldado atribui ao item, no valor que seus conterrâneos atribuiriam ao item e na reputação que carregar um item lhe conferiria) e é sumarizada no que chamamos de *valor Sulla*, $\mathscr{S}$, de um item. Depois de calcular o $\mathscr{S}$ de cada item relevante, Sulla compilou a seguinte tabela preliminar com o peso médio e valor médio de cada item[†]:

Item	Valor	Peso	Quantidade
Livros	10	5	1000
Barras de Ouro	100	500	85
Estátuas de Ouro	500	4000	7
Jóias	50	50	200

Sem muito sucesso em resolver o problema, Sulla mandou que chamassem um matemático local para resolvê-lo. Theon de Smyrna estava por perto e foi convocado. No início, se sentiu incomodado por ter que deixar de lado seu trabalho com os sólidos de Platão em favor de ajudar um bárbaro a resolver um problema desse tipo inferior, como considerava, mas, notando que pode ser mais difícil para um morto desenvolver trabalhos com os sólidos de Platão, decidiu ajudar Sulla com seu problema.

[*] Também conhecido como "o problema da mochila", ou *knapsack problem*, em inglês.

[†] Não há evidências de que tal tabela tenha sido feita, mas é mantida aqui pela conveniência ao exemplo.

Ao ter a situação explicada, Theon deu uma leve risada e disse o que se segue. "Você me dá um problema grandioso e difícil, oh grande Sulla! Os Deuses devem ter nos posto juntos, porque eu devo ser um dos poucos helenos, se não o único, que sabe resolvê-lo! Isto porque vi meu pai o resolvendo quando era criança. Espero que não se incomode em ouvir por alguns momentos um velho como eu relembrar a infância, ainda mais que isso te será útil."

"Continue." - disse Sulla.

"Meu pai era um fazendeiro muito forte, mas também tinha o seus limites. Às vezes, precisava realizar trabalhos em partes distantes e, para isso, precisava levar suas ferramentas. Cada ferramenta faria o seu trabalho mais fácil, ou o ajudaria a terminá-lo mais rápido, ou lhe proporcionaria maior conforto ao realizá-lo. Cada uma, também, tinha o seu peso, e meu pai, forte como era, não conseguia carregar todas. Então, ele compilou uma tabela parecida com a sua, com um "valor" que cada ferramenta lhe proporcionaria, assim como seu peso."

"Mas, meu caro, esse problema parece muito mais simples do que o meu: seu pai só precisava decidir entre levar uma ferramenta ou não, enquanto que eu preciso decidir entre quantos de cada tipo de item levar!" - disse Sulla.

"É verdade que o problema de meu falecido pai parece muito mais simples do que o seu. Mas toda pessoa bem educada, como tenho certeza ser o seu caso, oh conquistador de nações!, sabe que não se pode confiar nas aparências. Neste caso, ver que os problemas são essencialmente os mesmos é muito simples. Toda criança sabe que qualquer número natural pode ser decomposto como uma soma de potências de dois. Por exemplo, $11 = 2^0 + 2^1 + 2^3$. Além disso, essa decomposição é única, cada potência aparecendo no máximo uma vez. Como pode ver, então, o seu problema é um problema de decisão: decidir quais potências de 2 comporão as quantidades de cada tesouro que levará. Esse problema só tem o posssível incoveniente de fazer uso de mais variáveis. Mas não muito mais: $\lfloor log_2(q_i) \rfloor$, em que q_i é a quantidade a ser levada do item i.*"

"Mas esse problema adiciona complicações extras que não me parecem tão necessárias." - disse Sulla.

"Não tanto assim, mas tudo bem. Não quero tomar muito mais tempo seu, então vou direto ao ponto. Um modelo mais geral ao problema que deseja resolver é o seguinte:

$$\begin{aligned} maximizar \sum_{i=1}^{n} q_i \mathscr{S}_i, \\ tal\ que \sum q_i p_i \leq C, \\ 0 \leq q_i \leq l_i,\ q_i\ inteiro \end{aligned}$$

*Estamos adicionando, aqui e mais adiante, notação um pouco mais moderna do que a que o velho Theon poderia ter usado, por conveniência ao exemplo.

em que q_i é a quantidade a ser levada do item i, p_i o peso do item i (em $\mathscr{S}$), C a capacidade máxima a ser levada e l_i limite máximo do item i a ser levado (não podemos levar mais do que l_i do item i, talvez porque não existam mais do que l_i desses itens).

Esse é um problema de programação inteira. Existem muitas formas de resolver problemas de programação inteira. Se tiver algum tempo sobrando, eu poderia desenvolver uma outra forma com você. Será divertido e intrutivo."

"Vá em frente." - disse Sulla.

"Imagine que você sabe resolver esse problema para $C' < C$ e para todo subconjunto dos itens (um subconjunto de $\{1, ..., n\}$) e tenha em mãos uma solução para ele. Se retirarmos dessa solução um item i, de peso p_i, deve ser claro que a solução restante é ótima para o problema cuja capacidade de $C - p_i$ e conjunto de itens $\{1, ..., n\} - \{i\}$. Isso indica que uma solução ótima goza de uma subestrutura ótima. Isto é, a remoção de itens dessa solução gera uma solução que é ótima para outro problema, de fato um subproblema, mais simples. Se temos a solução para algum desses subproblemas, só precisamos expandi-lo, considerando itens que não foram considerados. Para cada um desses itens, sabendo que a nova solução ótima $op(i, c)$ para uma capacidade c, ao considerar a adição de um item i, é:

$$op(i,c) = \begin{cases} op(i-1,c), & se\ p_i > c \\ max\{op(i-1,c), op(i-1,c-p_i) + \mathscr{S}_i\} & se\ p_i \leq c \end{cases}$$

Fazendo $op(0, C) := 0$ e computando os valores de $op(i, c)$ por essa recursão* temos o valor ótimo que o meu pai queria. Se quisermos resolver o seu problema, precisamos fazer apenas uma pequena alteração, como a seguinte:

$$op(i,c) = max_{q_i}\{op(i-1, c-q_i p_i) + q_i \mathscr{S}_i \mid c \geq q_i p_i\}$$

Não deve ser difícil de ver como isso nos leva ao resultado desejado."

"Oh, mas Theon! Devo ter me enganado ao achar que era um matemático, pois você me está parecendo mais um ator de teatro! Isso porque qualquer ator de teatro me daria uma resposta como essa, mas ela é longe de satisfatória. Mas é pior do que um ator de teatro, porque quem quer que fosse incumbido de realizar a tarefa pelo método que você propôs ficaria um tempo intolerável fazendo contas, tempo esse que é, de fato, morto. E você seria o responsável pela morte." - disse Sulla. tempo passado faz parte da nossa morte.

Theon, percebendo a gravidade da situação, disse o seguinte. "Essa formulação, como eu disse antes, tinha um caráter mais introdutório e instrutivo. Se quer uma mais séria, existem várias e em vários sabores. Uma delas em particular faz uso extenso de técnicas de programação inteira, que são desen-

*Conhecida como recursão de Bellman.

volvidas há muitas décadas*. Fazendo uso dessas técnicas, podemos gerar o seguinte programa:

```
:- lib(eplex).

modelo(Pesos, Valores, Limitantes, Capacidade, Quantidades, Peso, Valor) :-
    ( foreach(Limitante,Limitantes), foreach(Quantidade,Quantidades) do
      Quantidade $:: 0..Limitante
    ),
    integers(Quantidades),
    Quantidades*Pesos $= Peso,
    Peso $=< Capacidade,
    Quantidades*Valores $= Valor.

resolve(Pesos, Valores, Limitantes, Capacidade, Quantidades, Peso, Valor) :-
    modelo(Pesos, Valores, Limitantes, Capacidade, Quantidades, Peso, Valor),
    optimize(max(Valor), Valor).

principal :-
    dados(Nomes, Pesos, Valores, Limitantes, Capacidade),
    resolve(Pesos, Valores, Limitantes, Capacidade, Quantidades, Peso, Valor),
    ( foreach(Quantidade,Quantidades), foreach(Nome,Nomes) do
      ( Quantidade >0 -> printf("%d of %w%n", [Quantidade,Nome]) ; true )
    ),
    writeln(peso = Peso ; valor = Valor).
```

Código 1: Conquistador Bárbaro Eplex

Podemos, analogamente, fazer uso da extensa pesquisa em otimização por restrições, com algo como o seguinte programa:

*Infelizmente, não há evidências de desenvolvimentos de programação inteira na época de Theon, mas a fala é mantida aqui por sua conveniência.

```
:- lib(ic).
:- lib(branch_and_bound).

model(Pesos, Valores, Limitantes, Capacidade, Quantidades, Peso, Valor) :-
        ( foreach(Limitante,Limitantes), foreach(Quantidade,Quantidades) do
                Quantidade $:: 0..Limitante
        ),
        integers(Quantidades),
        Quantidades * Pesos $= Peso,
        Peso $=< Capacidade,
        Quantidades * Valores $= Valor.

resolve(Pesos, Valores, Limitantes, Capacidade, Quantidades, Peso, Valor) :-
        model(Pesos, Valores, Limitantes, Capacidade, Quantidades, Peso, Valor),
        Custo #= -Valor,
        minimize(search(Quantidades, 0, largest, indomain_reverse_split, complete, []),
                 Custo).

principal :-
        dados(Nomes, Pesos, Valores, Limitantes, Capacidade),
        resolve(Pesos, Valores, Limitantes, Capacidade, Quantidades, Peso, Valor),
        ( foreach(Quantidade,Quantidades), foreach(Nome,Nomes) do
                ( Quantidade > 0 -> printf("%d of %w%n", [Quantidade,Nome]) ;
                  true )
        ),
        writeln(peso=Peso ; valor=Valor).
```

Código 2: Conquistador Bárbaro IC

Para saber qual dos métodos é o mais interessante, podemos simplesmente tomar uma pequena quantidade dos itens iniciais, fazer um teste, e ver qual resolvemos mais rápido.

"Isso pode ser uma dificuldade, porque, como ainda não alcançamos todos os lugares de onde os itens serão subtraídos, só temos uma ideia muito vaga sobre quais itens existem e qual o valor de cada." - disse Sulla.

"Um problema facilmente remediado. Podemos fazer testes com valores aleatórios, como exemplificado aqui." e, ao dizer isso, Theon primeiro explicou a notação que usaria e, depois, mostrou algo equivalente ao seguinte código a Sulla:

```
:- lib(ic_prop_test_util).

aleat_dados(dados([],[],[],[],_), 0).
aleat_dados(dados([Contador|Ls], [P|Ps], [V|Vs], [Q|Qs], _), Contador) :-
        random_int_between(1, 50,  P),
        random_int_between(1, 100, V),
        random_int_between(1, 15,  Q),
        Novo_Contador is Contador - 1,
        aleat_dados(dados(Ls, Ps, Vs, Qs, _), Novo_Contador).


inic_dados(dados(Nomes, Pesos, Valores, Qtd, _)) :-
        aleat_dados(dados(Nomes, Pesos, Valores, Qtd, _), 100).


dados(Nomes, Pesos, Valores, Qtd, 400) :-
        init_dados(dados(Nomes, Pesos, Valores, Qtd, 400)).
```

Código 3: Problemas Aleatórios

"Parece que, pelo menos por ora, nosso problema está resolvido." - disse Sulla.

"Sim, o seu problema está."

1.2 Os Testes

Levando em conta a discussão anterior, foram feitos testes para checar a eficiência dos métodos discutidos. Eles foram realizados em um computador Inspiron 5447, versão A06, com barramento de 64 bits, produzido pela Dell. Esse computador conta com um processador Intel Core i7-4510U, a 2GHz e sistema operaiconal Debian Strech. A versão do sistema de programação usado é ECL^iPS^e 7.0 (de Julho de 2018).

Os testes foram executados da seguinte forma. O método que faz uso de programação inteira foi testado 8 vezes, com o teste i contando com 2^{i+4} tipos de itens.

Cada tipo de item tem um peso, um volume que ocupa e uma quantidade de itens disponíveis. Nos testes que fazem uso de programação inteira, o peso é escolhido aleatoriamente como um valor entre 1 e 50 (unidades de peso), o volume como um valor entre 1 e 100 (unidades de volume) e, a quantidade, como um valor entre 1 e 15. Dada a natureza aleatória dos dados, é necessário fazer o teste com a mesma quantidade de tipos de itens mais de uma vez, e os que têm mais tipos de itens devem ser testados mais do que os que têm menos tipos de itens. A maneira escolhida foi de executar j vezes o teste com 2^j itens.

Infelizmente, realizar os mesmos testes com o método que faz uso de *branch and bound* foi impraticável, dado o tempo computacional necessário para tanto ser, no geral, proibitivo para $i \geq 6$*. No lugar disso, então, foram executados três testes para esse, os dois primeiros como explicados anteriormente (no caso, fazendo $i + 4 = 4$ e $i + 4 = 5$) e o terceiro caso de teste foi feito com 50 itens, o qual foi executado 6 vezes (sendo a quantidade de itens e de testes realizados as únicas diferenças entre esta modalidade e a explanada acima).

Os resultados são como segue na Tabela 1 (medições de tempo com dois valores significativos). Os valores na coluna "Programação Inteira" são referentes aos respectivos tempos para a resolução dos problemas por meio de restrições provindas da biblioteca *eplex* de programação inteira†, enquanto que os valores na coluna "Branch and Bound" são referentes aos respectivos tempos para a resolução dos problemas por meio de restrições provindas da biblioteca *branch_and_bound*‡ . Como os testes para 2^7 itens ou mais se tornaram inviáveis se fazendo uso da biblioteca ECL^iPS^e *branch_and_bound*, os respectivos espaços na tabela foram substituídos por "NA".

* "No geral", porque, os dados sendo aleatórios, às vezes o programa termina rapidamente. Esse, entretanto, não é o caso geral.

†Essa biblioteca ECL^iPS^e é uma maneira geral de fazer interface com uma variedade de *solvers* para problemas de programação linear e inteira. Neste caso em particular, os *solvers* carregados são o *COIN-OR CLP*, versão 1.6 (linear) com *CBC* versão 2.9 (inteiro misto)[4].

‡A biblioteca ECL^iPS^e *branch_and_bound* provê predicados genéricos que implementam a busca por *branch and bound*, isto é,

1. achando uma solução,
2. adicionando uma restrição que requer uma solução melhor do que a melhor vista até então e
3. achar uma solução que satisfaz essa nova restrição.

Tempos	Programação inteira	Branch and Bound
Menor tempo para 2^4	0,00s	0,00s
Maior tempo para 2^4	0,01s	0,09s
Tempo médio para 2^4	0,00s	0,02s
Menor tempo para 2^5	0,00s	0,00s
Maior tempo para 2^5	0,01s	0,07s
Tempo médio para 2^5	0,00s	0,01s
Menor tempo para $2^6/50$	0,00s	0,08s
Maior tempo para $2^6/50$	0,01s	4,70s
Tempo médio para $2^6/50$	0,01s	1.81s
Menor tempo para 2^7	0,01s	NA
Maior tempo para 2^7	0,01s	NA
Tempo médio para 2^7	0,01s	NA
Menor tempo para 2^8	0,02s	NA
Maior tempo para 2^8	0,02s	NA
Tempo médio para 2^8	0,02s	NA
Menor tempo para 2^9	0,02s	NA
Maior tempo para 2^9	0,04s	NA
Tempo médio para 2^9	0,03s	NA
Menor tempo para 2^{10}	0,05s	NA
Maior tempo para 2^{10}	0,07s	NA
Tempo médio para 2^{10}	0,06s	NA
Menor tempo para 2^{11}	0,60s	NA
Maior tempo para 2^{11}	0,80s	NA
Tempo médio para 2^{11}	0,70s	NA
Menor tempo para 2^{12}	0,14s	NA
Maior tempo para 2^{12}	0,16s	NA
Tempo médio para 2^{12}	0,14s	NA

Tabela 1: Resultados

Referências

[1] Kellerer, Hans e Pferschy, Ulrich e Pisinger, David, "Knapsack Problems", Springer

[2] Plutarch, "Sulla", traduzido por John Dryden

[3] ECL^iPS^e, http://www.eclipseclp.org/examples/

[4] COIN-OR, https://projects.coin-or.org/CoinBinary