>> Algoritmos Gulosos

* Aplicáveis no contexto de problemas de otimização.
* Funcionam de forma similar aos de PD no que se refere à forma de construção de soluções
* Tentam economizar tempo ao definir critérios de escolha para a construção de uma solução, em vez de testar várias possibilidades
* Os critérios de escolha constituem o que chamamos de escolha gulosa
* Podem ou não culminar em uma solução ótima, a depender das escolhas feitas
* Sempre devem construir uma solução viável.

Para que um algoritmo guloso seja considerado correto devemos observar:

1. Caracterização de uma solução ótima.
2. Critérios de escolha gulosa.
3. Subestrutura ótima do problema

Nem todos os problemas de otimização aceitam algoritmos gulosos corretos.

>> Mochila Fracionaria

Um ladrão vai entrar num deposito e precisa calcular que joias levar para ganhar mais dinheiro, diante do limite de carga que ele possui.

- Mochila com capacidade M
- Conjunto de itens com n itens
  *Cada item i possui um valor Vi e um peso pi.
  *E diretamente proporcional a relação entre uma fração fi de um item e seus parâmetros.

fi ----------> (fi*pi) (peso ocupado)
(função obtida) '--> (fi*vi) (ganho obtido)

Desejamos maximizar o ganho do que podemos levar na mochila.
Vamos considerar que sempre vamos poder levar a mochila cheia, pois, mesmo que algo possua um peso que ultrapasse capacidade, ele pode ser fracionado para ser carregado.

* Se a soma dos pesos dos itens e menor que a capacidade total da mochila, a sol. ótima é trivialmente levar 100% de todos os itens.
* Se a soma dos pesos dos itens for maior ou igual a M, podemos garantir que a solução ótima preenche a mochila completamente.

Estratégias Gulosas

1. Escolher o item de maior custo/beneficio, Vi/Pi, tomando o máximo que couber, ou seja, incluir o maior sempre, e quando ele não couber mais, incluir o 2º maior
sob a mesma regra.

Exemplo:
M = 5

```
     vi | 2   4   4
     ---|------------
     pi | 3   2   3          >-------------> 2/3 e 4/3 = 4+4/3
  valor | 3º  1º  2º                    2/3 e 4/2 = 6
                           4/2 e 4/3 = 8 ------> maior ganho
```

2. Escolher o item de maior valor, desempatando pelo menor peso, na fração que couber.
M = 5   2/3 4/2 4/3 6/5 ---> não encontra ótimo

3. Escolher o item de menor peso, desempatando pelo maior valor, na fração que couber.
M = 4   1/2 1/2 3/4 ---> não encontra ótimo

Subestrutura ótima:
- Solução como sequencia das frações escolhidas de cada item.

Teorema: A estrategia (1) é correta
Demonstração:
OBS.: Pela forma como a estrategia se comporta, no máximo um item terá valor 0<fi<1

Seja S* uma solução ótima e S uma solução dada pelo guloso. Sem perda de generalidade, imagine que ambas foram reordenadas pelos itens do maior para o menor custo-benefício.

Seja i o primeiro índice onde ocorre divergência entre S* e S, ou seja, fi*=fi.

Perceba que, pela escolha gulosa, é impossível que fi*>fi, pois escolhemos sempre a maior fração
que cabe na mochila. Isso nos permite concluir que 0<=fi<=1. Ou seja, fi>0.

Como S tem no máximo um item 0<fk<1, todos os itens anteriores a i tem que terr sido escolhidos completamente.

Vamos considerar o seguinte procedimento para converter fi em fi*, com o objetivo final de convertermos S em S*. Certamente existe um item j>i em S que não foi escolhido completamente, pois não estamos considerando instancias que cabem completamente na mochila. Vamos reduzir a escolha de i por uma quantidade E e aumentar a escolha de j por uma quantidade E' de forma a manter a mochila preenchida com E,E'>0.

pi * E = pj * E'

Vamos diminuir E de i e aumentar E' de j.
$f1*p1 + f2*p2 + ... + (fi-E)*pi + ... + (fj+E')*pj + ... + fn*pn = M$

$(f1*p1 + ... + fi*pi + ... + fj*pj + ... + fn*pn) - E*pi + E'*pj = M$
$M - E*pi + E'*pj = M$
$E*pi = E'*pj$

(i) ($E/E' = Pj/Pi$)

O valor do ganho dessa nova configuração é dado por:
$(f1*v1 + f2*v2 + ... + fi*vi + ... fj*vj + ... + fn*vn) - E*vi+E'*vj$
(ii) $G - E*vi + E'*vj = G$

Sendo G o valor do ganho da solução S. Pela escolha de i e j, temos que $Vi/Pi >= Vj/Pj$
Dai, temos que $Vj/Vi <= Pj/Pi = E/E'$ -> $Vj/Vi <= E/E'$
$E'*Vj <= E*Vi$ -> $E'*Vj-E*Vi <= 0$ (iii).

Observando (ii) e (iii), $G'<=G$.
O ganho obtido pela nova configuração é menor ou igual ao ganho obtido por S.
Vamos denominar essa nova solução de S1.
Considere agora o processo de repetidamente comparar uma solução Sk com S* e aplicar as transformações que descrevemos para gerar uma nova solução viável Sk+1. Se nos convencermos que esse processo términa, teremos a última solução gerada igual a S.

Para isso, perceba que é impossível gerarmos uma solução obtida anteriormente pois a cada nova solução estamos diminuindo estritamente um dos elementos sem afetar os que ocorrem antes dos pontos de divergência.

Assim, construímos nesse processo a sequencia finita de soluções:
S = S0 -> S1 -> S2 -> ... -> Sr = S*

Porém, considerando v(Sk) como o valor de uma solução viável Sk qualquer, sabemos que v(S) = V(S0) >= V(S1) >= ... >= V(Sr) = V(S*),
portanto, temos que V(S) >= V(S*) (iv).

Entretanto, como S* é ótima e S é viável, temos que v(S) <= V(S*) (V).
De (IV) e (V) concluímos que V(S) = V(S*), ou Seja, S deve ser ótima.

>> Optimal Caching

cache miss -> quando buscamos um dado na cache e não encontramos.
cache eviction -> determinar quem vai sair da cache.

Dada uma memoria cache com M posições, um conjunto D = {d1,d2,...,dm} de posições de memória e uma sequencia L determinando a ordem de leitura das posições D para realizar uma computação queremos determinar uma estrategia de cache eviction para garantir o mínimo de cache misses para L.

Exemplo:
D = {a,b,c,d}
L = <a,b,a,d,a,b,d,c,a>
M = 3

```
|  |  |  |   -->  |a| | | --> |a|b| | --> |a|b| | --> |a|b|d| --> |a|b|d| --> |a|b|d| --> |a|c|d| --> |a|c|d|
                                                                                             |
                                                                                             \/
                                                                                      cache eviction
```

OBS.: Toda estrategia ótima deve decidir por não realizar eviction quando a posição necessária já estiver na cache.

-Algumas possíveis estrategias
(1) Jogar fora a o posição cujo próximo acesso na computação está o mais no futuro.
(2) Jogar fora a que menos ocorrer no futuro
(3) Jogar fora a que foi menos acessada até agora
(4) Jogar fora a acessada mais recentemente
(5) Jogar fora a acessada mais antigamente

Provaremos que a estratégia 1 é ótima(Farthest in Future)

Definição: Uma estratégia de cache-eviction que só traz para a cache uma posição de memória apenas quanto ela é necessária e não está na cache é chamada de reduzida.

Teorema: Toda sequencia de instruções de cache-evction S possui uma equivalente S* em número de cache-misses obtida por uma estratégia reduzida.

Demonstração: Vamos chamar S* redução de S
Vamos criar uma sequenca S* a partir de S da seguinte forma.
* Até o primeiro momento em que S respeita a propriedade, S* será sua cópia.
* No primeiro ponto em que S trouxer uma posição c para a cache diferente da que será executada, vamos instruir S* a não fazer nada e transportar essa movimentação de c para a posição da sua próxima execução em L.
* Entre essa divergência e o próximo acesso a c, qualquer acesso a um conteúdo diferente da posição e jogada fora por S será replicada em S*. Qualquer acesso a e sera transformado em não fazer nada em S*.
Com isso, garantiremos que o número de cache musses dentro desse intervalo será menor ou igual em S* com relação a S.

Teorema: A estratégia Farthest-in-Future é correta.
Demonstração: Vamos demonstrar que a seguinte afirmação é verdadeira

Dada uma solução gulosa Sff e uma outra reduzida Sj que concorda com Sff nas primeiras j decisões, existe uma solução reduzida Sj+1 que concorda nas primeiras j+1 posições e que gera no máximo a mesma quantidade de cache misses de Sj.

Considere a j+1 instrução da computação e suponha que requeremos acesso á memória d. Como Sj e Sff concordaram até a posição j, neste ponto temos o mesmo conteúdo de cache para ambas.

Caso 1: d está na cache

Como Sj é reduzida, ambas devem ter a instrução j+1 = nop (Sj+1 = Sj)

Caso 2: d não estpa na cache

Caso 2.1: Ambas jogam fora o mesmo elemento. Como Sj é reduzida, ela deve trazer d para a mesma posição que Sff traz. Portanto, Sj[j+1] = Sff[j+1] (Sj+1 = Sj)

Caso 2.2: Sj joga fora f e Sff joga fora e != f. (Ambas trazem d para a cache)

Vamos construir uma sequencia Sj+1* identica a Sff nas primeiras j+1 instruções, e completar as demais conforma o caso. Apesar do conteúdo de cache entre Sj+1* e Sj divergirem, vamos tornar Sj+1 equivalente a Sj até que algum dos casos a seguir aconteça pela primeira vez.

Caso 2.2.1: L faz requisição a uma posição g != e, f que não está na cache de Sj e Sj descarta e. Podemos instruir em Sj+1* descartar a posição f e, assim, conseguimos igualar as caches de ambas.

Caso 2.2.2: L faz requisição a f e Sj descarta um item g = e. Instruímos Sj+1* a não fazer nada. Assim, temos as configurações de cache iguais.

Caso 2.2.3: L requer f e Sj descarta um item g != e. Vamos instruir Sj+1* a trazer e para a cache descartando o mesmo elemento g. Daí, conseguimos igualar as caches. Observe que como Sj concorda com Sff nas j primeiras instruções e Sff decide descarte e na próxima instrução em vez de f, deve ser o caso que f ocorre antes de e em L, dada a escolha gulosa. Assim, esses são os únicos casos a considerar. Daí, qualquer que seja o caso, vamos completar Sj+1* com as demais instruções de Sj. Podemos concluir que Sj+1 não aumenta o numero de cache misses de Sj. Podemos concluir que Sj+1 não aumenta o número de cache misses de Sj. Por fim, determinamos Sj+1 como sendo a redução da sequencia Sj+1* construída. Terminando a demonstração.

Perceba que se iniciarmos com uma solução ótima reduzida S'(que sempre deve existir), aplicando repetidamente o processo de obtenção de nova solução descrito anteriormente, junto ao teorema de subestrutura ótima, obteremos uma sequencia de soluções:

S' = S0 → S1 → … → Sn = Sff

de forma que

V(S') = V(S0) >= V(S1) >= … >= V(Sn) = V(Sff)

e daí, V(S') >= V(Sff), porém, como S' é ótima, concluímos que Sff também é ótima.

>> Seleção de Atividades

Dado um conjunto de atividades A, representados por intervalos semiabertos, determinar o maior subconjunto de A que contém apenas atividades compatíveis, ou seja, cujos intervalos não se interceptam.

Teorema: A escolha gulosa pelo intervalo compatível com menor fim é ótima.

Demonstração: Suponha que S^* seja uma solução ótima qualquer para o problema, e imagine que a apresentamos como uma sequência ordenada pelo fim dos seus intervalos.

Se S^* já contem o intervalo de A que termina primeiro, temos inclusa a escolha gulosa.

Suponha então que s_1^* não seja o intervalo t e A que termina primeiro.

Como s_1^* é em S^* o intervalo que tem menor fim, temos que todos os demais devem ter início após o fim deste. Com isso, podemos concluir que os demais são todos compatíveis com t, já que o tempo de fim de t é menor que o de s_1^*.

Portanto, a solução $S'=< t, s_2^*, s_3^*, \dots, s_n^*>$ é viável. Como ela tem a mesma cardinalidade de S^*, ela também é ótima.