PCC104 - Projeto e Análise de Algoritmos

Marco Antonio M. Carvalho

Departamento de Computação
Instituto de Ciências Exatas e Biológicas
Universidade Federal de Ouro Preto

Conteúdo

Fonte

Este material é baseado nos livros

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
- S. Halim. *Competitive Programming*. 3rd Edition, 2013.
- Ian Parberry and William Gasarch. *Problems on Algorithms*. Second Edition, 2002.
- Ian Parberry *Lecture Notes on Algorithm Analysis and Complexity Theory*. Fourth Edition, 2001.

Licença

Backtracking

Ian Parberry e William Gasarch, *Problems on Algorithms*

"*Backtracking is a type of exhaustive search in which the combinatorial object is constructed recursively, and the recursion tree is pruned, that is, recursive calls are not made when the part of the current object that has been constructed cannot lead to a valid or optimal solution*"

Definição

O paradigma *Backtracking* (ou "Tentativa e Erro") é relacionado à Busca Completa, no tangente a explorar o espaço de soluções inteiro, se necessário, em busca da solução desejada.

É considerado um refinamento da busca completa, uma vez que pode eliminar parte do espaço de soluções sem examiná-lo.

O princípio é construir uma solução gradativamente na base da tentativa e erro, desmanchando partes da solução quando necessário.

Normalmente, este paradigma é associado a algoritmos recursivos.

Aplicação

O *Backtracking* é aplicado em problemas cuja solução pode ser definida construtivamente, por meio de uma sequência de decisões.

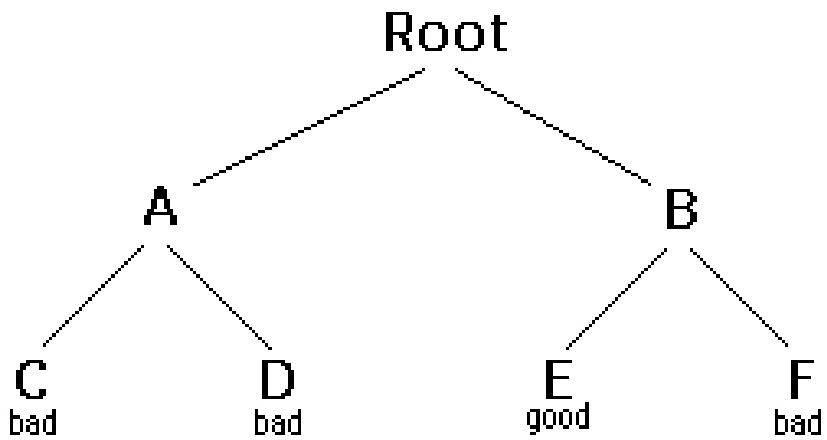
Outra característica dos problemas passíveis de solução por *Backtracking* é a capacidade de modelagem por uma árvore que representa todas as possíveis sequências de decisão.

Exemplo

Suponhamos um problema genérico em que começamos no nó raiz de uma árvore, que pode ter nós folhas "bons" e "ruins", em qualquer proporção e distribuição. Queremos chegar a um nó folha "bom".

A partir da raiz, escolhemos sucessivamente nós filhos, até chegarmos a um nó folha:

- Se o nó folha é "bom", paramos;
- Se o nó folha é "ruim", desfazemos nossa última escolha e tomamos outra opção
  - Caso não haja outra opção, desfazemos a penúltima escolha, e assim sucessivamente.
  - Se voltarmos à raiz da árvore e não houver mais opções, não há folhas "boas" na árvore.


Root
A
B
C
bad
D
bad
E
good
F
bad

Justificativa

Se houver mais do que uma opção disponível para cada uma das n decisões, a busca completa será exponencial.

A eficiência do *Backtracking* depende da capacidade de limitar esta busca, ou seja, podar a árvore, eliminando as ramificações não promissoras.

Desta forma, é necessário definir o espaço de busca para o problema:

- Que inclua a solução ótima;
- Que possa ser pesquisada de forma organizada, tipicamente, como uma árvore.

Princípio

O *Backtracking* é baseado na tentativa e erro.

De acordo com a característica do problema computacional tratado, construímos uma solução gradativamente, passo a passo.

A cada passo, tomamos uma decisão que adicionará um novo elemento à solução parcial.

Esta nova solução, na verdade é uma tentativa de estender a solução parcial atual, no intuito de termos uma solução completa.

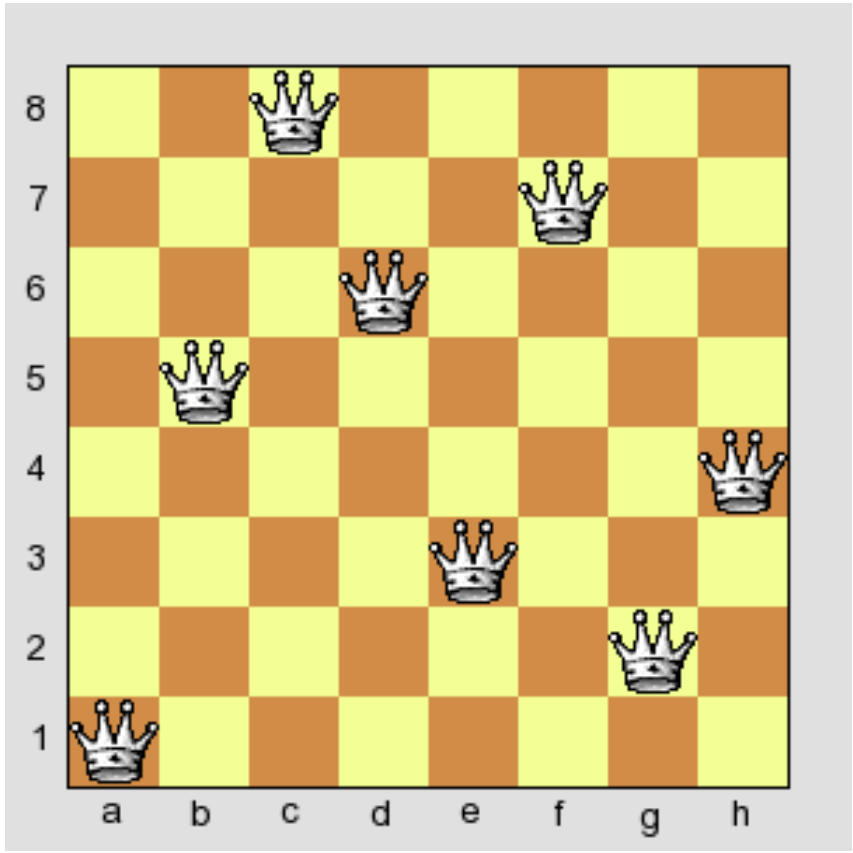
Se chegarmos em algum ponto em que, por algum motivo, não seja possível estender a solução e não tivermos uma solução completa (erro), podemos "voltar atrás" (*backtrack*) e desfazer a tentativa.

Versão Genérica

backtracking(v)
Entrada: vértice inicial v
se *promissor(v)* **então**
  se *solucao_completa(v)* **então**
    armazena solucao(v);
  fim
fim
para cada *filho de v* **faça**
  backtracking(filho);
fim

Exemplo - O Problema das 8 Rainhas

Em um tabuleiro 8×8 é possível posicionar oito rainhas tal que elas não se ataquem umas às outras? Escreva um programa que, dada a posição inicial de uma rainha, escreva **todas** as soluções possíveis.



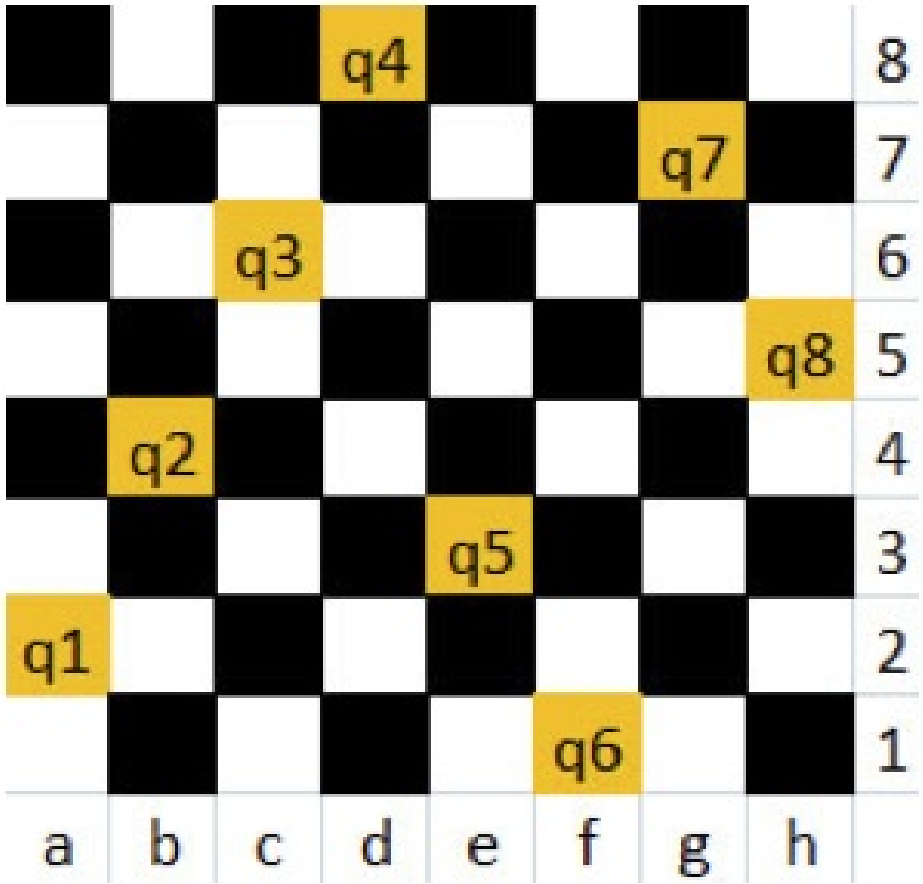
O Problema das 8 Rainhas - Análise

Uma solução ingênua para o problema tentaria todas as $8^8 \approx 17M$ de possibilidades, colocando cada rainha em uma casa e filtrando as soluções inviáveis.

Sabemos que duas rainhas não podem estar na mesma coluna, portanto, podemos reduzir o problema a permutar a posição das rainhas nas linhas.

Por exemplo, `linha = [2, 4, 6, 8, 3, 1, 7, 5]` representaria a solução da figura seguinte, em que `linha[1]`=2 indica que a rainha da coluna 1 está posicionada na linha 2.

Modelando o problema desta forma, diminuímos o espaço de busca de $8^8 \approx 17M$ para $8! = 40K$.



Exemplo de solução para o problema das 8 Rainhas: [2, 4, 6, 8, 3, 1, 7, 5].

O Problema das 8 Rainhas - Análise

Sabemos também que duas rainhas não podem ocupar a mesma diagonal.

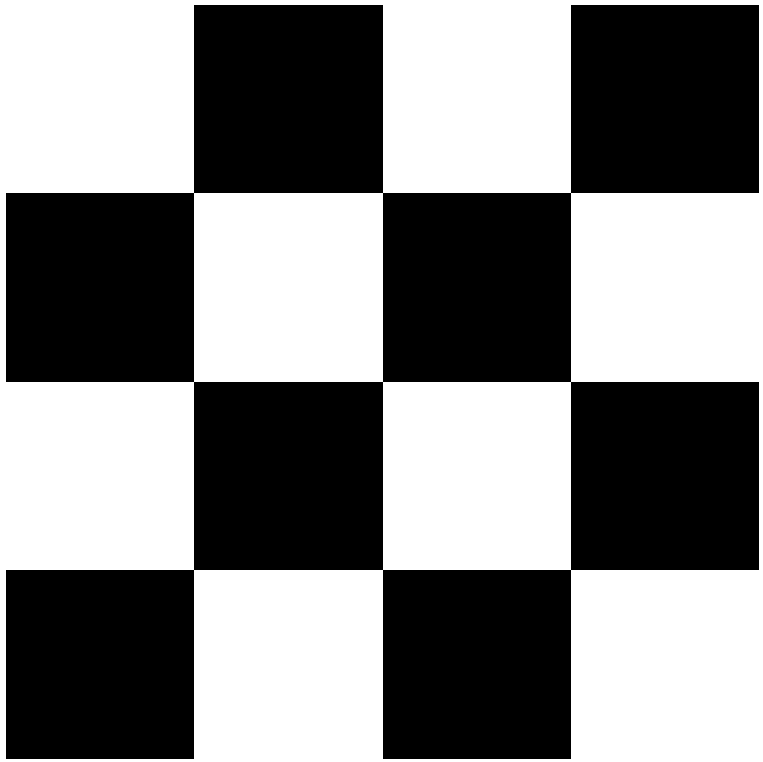
Supondo que a rainha A está posicionada em (i, j) e que a rainha B está posicionada em (k, l), elas se atacarão se $|i - k| = |j - l|$.

Uma solução por *backtracking* posicionaria as rainhas uma por uma, da coluna 1 até a coluna 8 respeitando as restrições do problema.

Caso o posicionamento de uma rainha viole alguma das restrições (**erro**), o último posicionamento (**tentativa**) é desfeito e realizado em outra casa (**nova tentativa**).

No caso de não haver uma nova tentativa, o posicionamento da rainha anterior também é desfeito, e assim sucessivamente, até haver a possibilidade de uma nova tentativa.

Por fim é necessário verificar a condição de que uma das rainhas está na posição original descrita pelo problema (dependendo do enunciado).



Exemplos de tentativas, erros e *backtracking* para a versão com 4 rainhas.

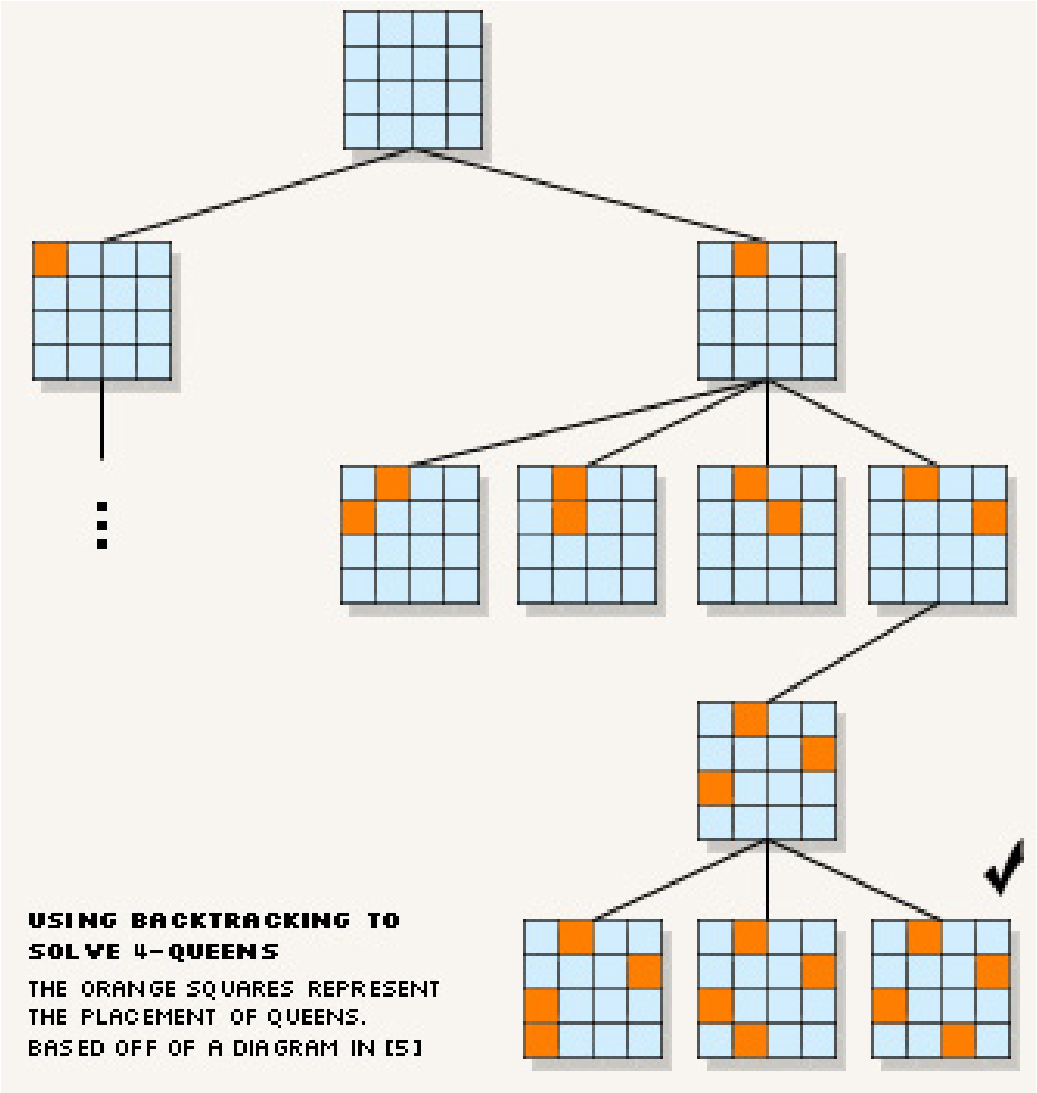


Ilustração parcial do *bracktracking* para a versão com 4 rainhas.

Código

Ver o código 8queens.cpp

Melhorias

Algumas melhorias podem ser aplicadas ao *Backtracking*, de modo a não realizar uma busca completa:

- Poda do espaço de soluções;
- Utilização de simetrias;
- Pré-computação.

Poda do Espaço de Soluções

Ao gerarmos soluções, podemos nos deparar com uma solução parcial que nunca se tornará uma solução completa viável.

Podemos podar esta parte do espaço de soluções e nos concentrarmos em explorar outras partes.

No problema das 8 rainhas, se posicionarmos uma rainha em `linha[1]=2` e depois posicionarmos outra rainha em `linha[2]=1` ou `linha[2]=3` teremos um conflito em diagonal.

Continuar a buscar a partir de uma destas situações jamais nos levará a uma solução viável.

Desta forma, podamos estas ramificações e nos concentramos apenas nas posições válidas `linha[2]=[4, 5, 6, 7, 8]`, economizando tempo de execução.

Utilização de Simetrias

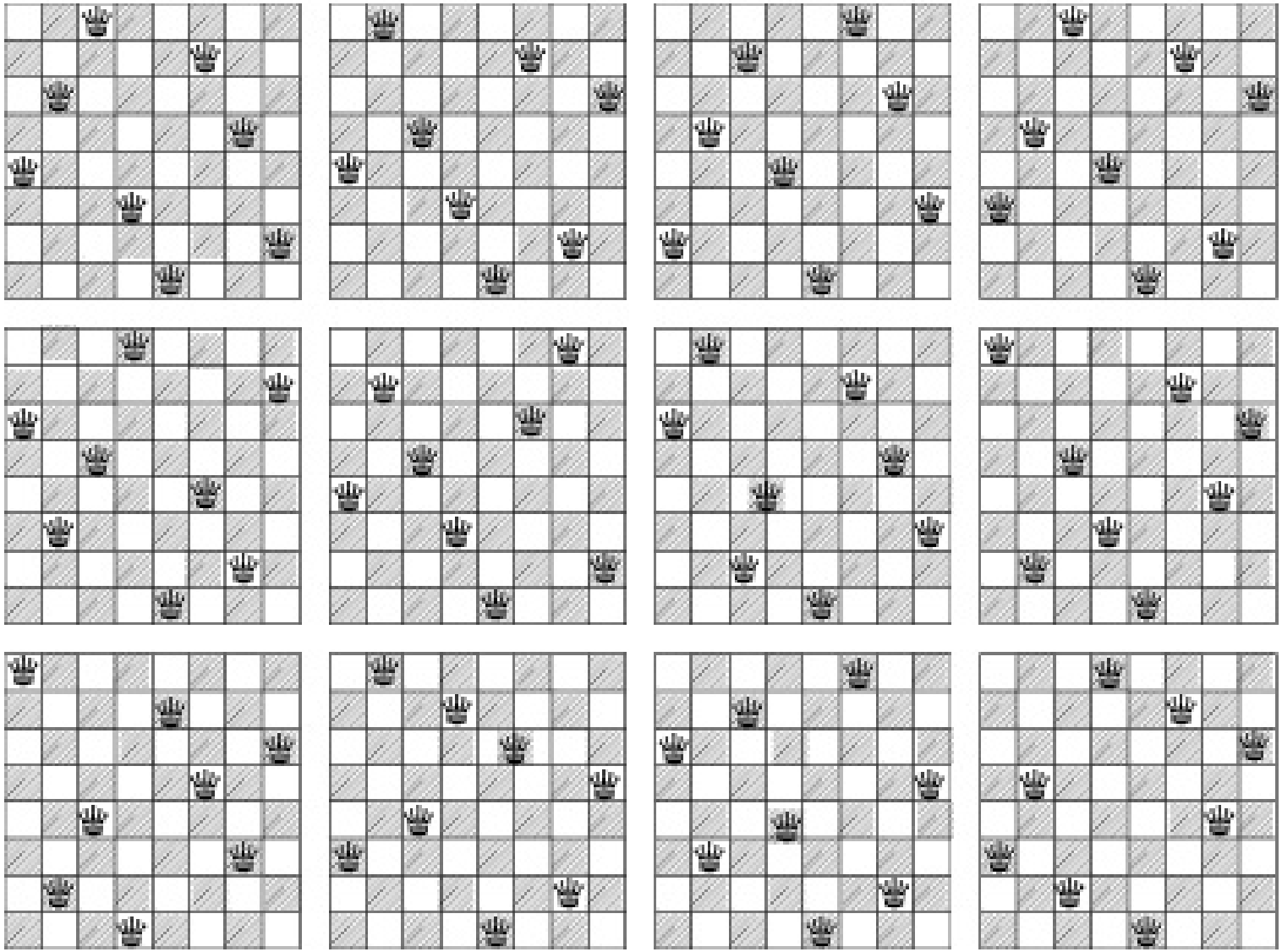
Alguns problemas apresentam simetrias em sua estrutura e podemos utilizá-las em nosso favor.

Desta forma podemos utilizar uma versão reduzida do espaço de soluções, reduzindo o tempo de execução do código.

Exemplo

No problema das 8 rainhas original, sabemos que existem somente 92 soluções, porém, apenas 12 delas são únicas (as outras são rotações e reflexões).

Podemos gerar as 12 soluções únicas e, se necessário, todas as 92 por meio das rotações e reflexões das 12 originais.



As 12 soluções originais do problema das 8 rainhas.

Pré-Computação

Pode ser útil gerar e preencher tabelas ou outras estruturas de dados que nos permitam uma busca rápida por valores, antes mesmo da execução do programa em si.

A pré-computação implica em uma relação de aumento de consumo de memória e diminuição do tempo de execução.

Exemplo

Supondo uma solução de *backtracking* para o problema do Caixeiro Viajante, podemos pré-computar limitantes inferiores e superiores para a solução e utilizar estes valores para guiar a busca.

Nesta mesma solução, podemos pré-computar soluções parciais avaliadas como "ruins" e utilizar esta informação durante o *backtracking*.

Pré-Computação

Novamente considerando o caso das 8 rainhas, sabendo que existem 92 soluções, podemos criar uma matriz 92×8 e então preenchê-la com todas as 92 permutações válidas das 8 rainhas.

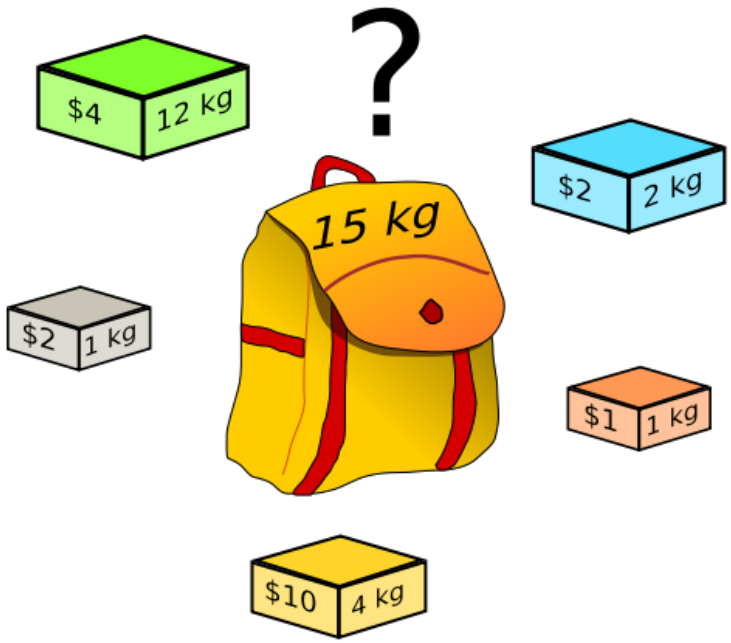
O programa **gerador** levará algum tempo para preencher a matriz, mas será executado uma única vez – outro programa pode simplesmente buscar os valores na tabela posteriormente.

Desta forma, o *backtracking*, ao invés de computar as soluções em tempo real, seria utilizado como um gerador.

A mesma lógica de pré-computação pode ser aplicada a problemas bem caracterizados e com a entrada limitada de alguma forma, como a geração de números da sequência *Fibonnaci*, etc.

Exemplo - O Problema da Mochila 0-1

Dadas uma mochila de capacidade W e uma lista de n itens distintos e únicos (enumerados de x_1 a x_n), cada um com um peso $w_1, w_2, \ldots, w_n$ e um valor $v_1, v_2, \ldots, v_n$, maximizar o valor carregado na mochila, respeitando sua capacidade.



Backtracking Ingênuo - O Problema da Mochila 0-1

O espaço de soluções do problema da mochila possui tamanho 2^n, pois cada um dos itens pode ou não ser selecionado.

Note que este espaço de soluções inclui soluções inviáveis, como por exemplo, aquela em que todos os elementos são selecionados.

O algoritmo apresentado a seguir é uma versão ingênua, pois explorará todas as 2^n possíveis soluções.

Na modelagem utilizada, é necessário determinar uma n-tupla $[x_1, x_2, \ldots, x_n] \in \{0, 1\}^n$, indicando quais itens foram selecionados, de acordo com as restrições e objetivo do problema.

Na nomenclatura utilizada, $Tupla_{opt}$ e V_{opt} representam os itens selecionados para a solução e o valor total, respectivamente.

Ainda, P e V armazenam, respectivamente, o peso e o valor de uma solução parcial.

Backtracking - Versão Ingênua

1 **MochilaBacktrackingIngenuo**
Entrada: indice do objeto atual i, peso atual P, Valor atual V
2 **se** $i = n$ **então**
3 | **se** $P \leq W$ ***e*** $V > V_{opt}$ **então**
4 | | $V_{opt} \leftarrow V$;
5 | | $Tupla_{opt} \leftarrow [x_1, x_2, \ldots, x_n]$;
6 | **fim**
7 **senão**
8 | $x_i \leftarrow 1$; **MochilaBacktrackingIngenuo**$(i + 1,\ P + w_i,$
| $V + v_i)$;
9 | $x_i \leftarrow 0$; **MochilaBacktrackingIngenuo**$(i + 1, P, V)$;
10 **fim**

Backtracking com Poda - O Problema da Mochila 0-1

A versão a seguir inclui uma poda simples no *Backtracking*: se a adição de um novo objeto exceder a capacidade da mochila, então a ramificação não será estendida.

Na nomenclatura utilizada, adiciona-se C_i representando o conjunto de opções para cada objeto:

- 0: o objeto não será incluído na mochila;
- 1: o objeto será incluído na mochila.

Backtracking - Versão com Poda

```
1  MochilaBacktrackingPoda
   Entrada: indice do objeto atual i, peso atual P, Valor atual V
2  se i = n então
3  |   se V > V_opt então
4  |   |   V_opt ← V;
5  |   |   Tupla_opt ← [x_1, x_2, ..., x_n];
6  |   fim
7  senão
8  |   se P + w_i ≤ W então
9  |   |   C_i ← {0, 1};
10 |   senão
11 |   |   C_i ← {0};
12 |   fim
13 |   para cada j ∈ C_i faça
14 |   |   x_i ← j;
15 |   |   MochilaBacktrackingPoda(i + 1, P + w_i x_i, V + v_i x_i);
16 |   fim
17 fim
```

Comentários Finais

O *backtracking* é um paradigma utilizado quando é necessário obter soluções ótimas, enumerar soluções, ou quando não se sabe qual caminho seguir para buscar uma solução.

É necessário que o problema possibilite a construção gradual de uma solução, e que se organize como uma árvore.

Deve ser levado em consideração o fator de ramificação do problema, o que pode levar a um crescimento rápido do espaço utilizado.

Melhorias ao *backtracking*, modelagem correta e truques de programação auxiliam a reduzir o espaço de soluções, salvando tempo de execução.

Exercício

1. Indique variações do problema da mochila que dificultariam ou impediriam a formulação por *backtracking*.
2. Apresente um segundo esquema de poda para o problema da mochila binária.
3. Apresente a árvore de exploração do *backtracking* com poda para o problema da mochila binária para:
   - $W = 10$;
   - P=[8, 1, 5, 4]; e
   - V=[500, 1000, 300, 210].

Dúvidas?