Busca com satisfação de restrições (*backtracking*)

João Batista

Baseado várias fontes:

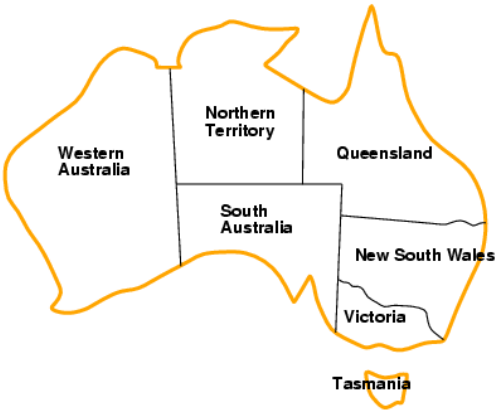
- Gustavo Batista, que se baseou em: Hwee Tou Ng:
  http://aima.eecs.berkeley.edu/slides-ppt/
  Artificial Intelligence: A Modern Approach – Russell & Norvig
- Vários sítios na internet e tambem livro do Halim – Competitive Programming

Numenclatura

- Busca Completa ?
- Força Bruta?
- Backtracking Recursivo ?
  - São sinônimos !
- Consistem de métodos que vasculham o espaço de busca inteiro (ou então parte dele) para obter a soluçao desejada.
  - Durante a busca, podemos podar parte do espaço se entendermos que tais partes não tem o que procuramos !

Exemplo: Coloração de Mapas

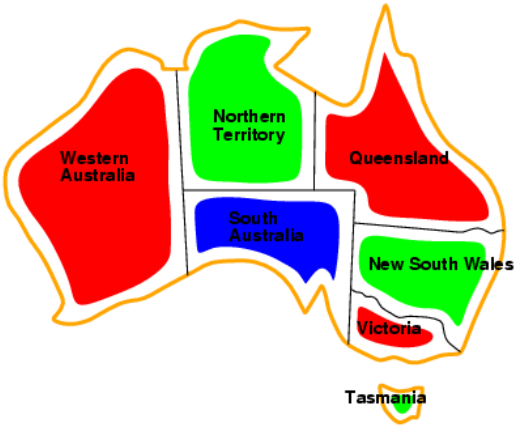
- O objetivo é colorir um mapa utilizando diferentes cores para regiões adjacentes



Estado inicial

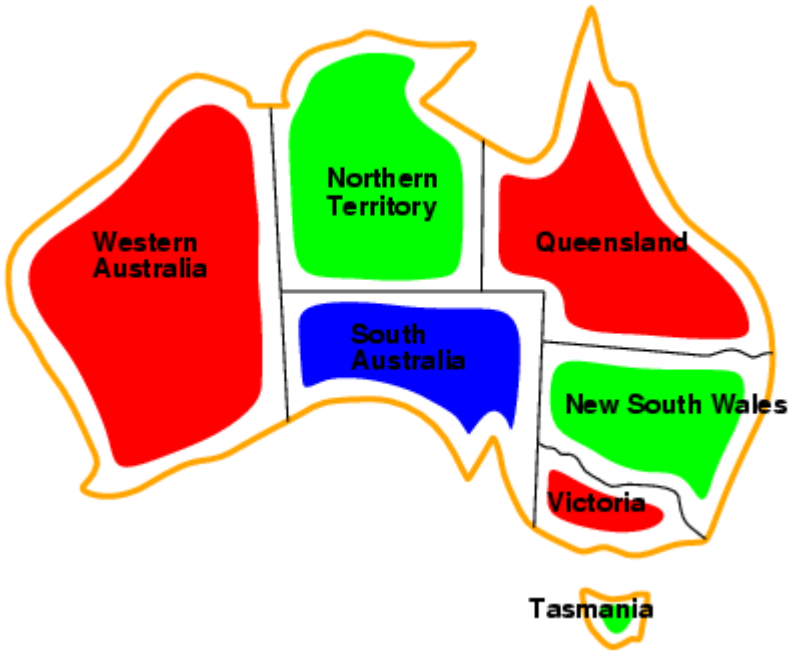
- Podemos modelar esse exemplo como um problema de busca

Estado final



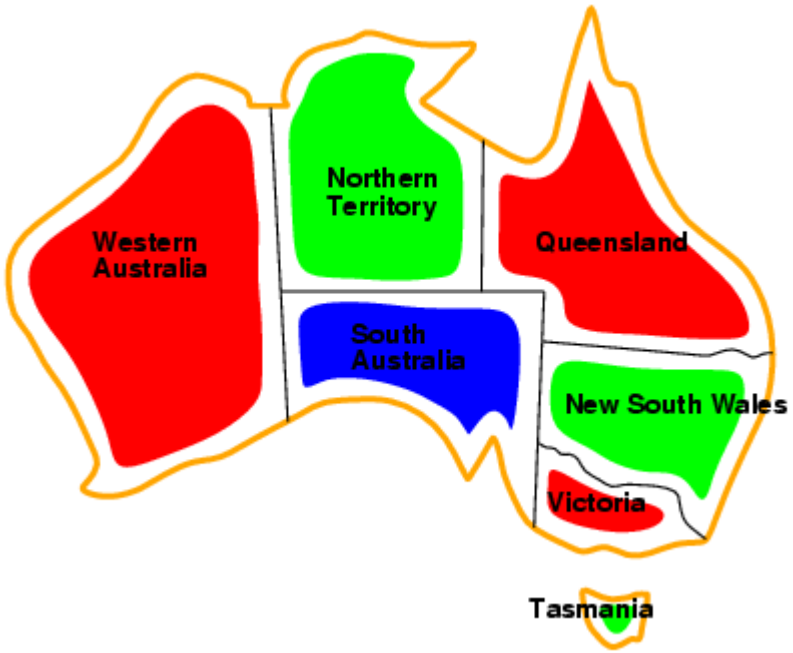
Modelagem: Coloração de Mapas

- Essa modelagem consiste de:
  - Variáveis: WA, NT, SA, Q, NSW, V, T
  - Domínio: Vermelho, verde e azul
  - Restrições: regiões adjacentes devem possuir cores diferentes



Modelagem: Coloração de Mapas

- Objetivo:
  - Atribuir valores a todas as variáveis
  - Cada atribuição deve respeitar as restrições impostas



Problemas Combinatoriais

- Problemas combinatoriais podem ser modelados dessa maneira
  - Envolvem encontrar uma atribuição, ordenação ou agrupamento a um conjunto finito de objetos discretos que satisfaz certas condições
- Diversos problemas em computação são problemas combinatoriais e podem ser modelados da mesma maneira.

Exemplo: Sudoku

- Modelagem:
  - Variáveis: A1...A9, B1...B9, ..., I1...I9
  - Domínio: 1...9
  - Restrições:

Alldiff(A1,A2,A3,A4,A5,A6,A7,A8,A9)

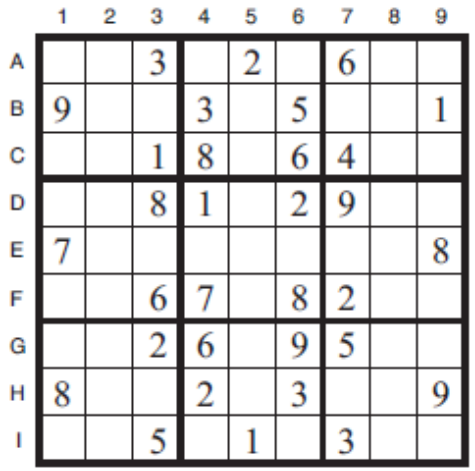
...

Alldiff (A1,B1,C1,D1,E1,F1,G1,H1,I1)

...

Alldiff (A1,A2,A3,B1,B2,B3,C1,C2,C3)

...



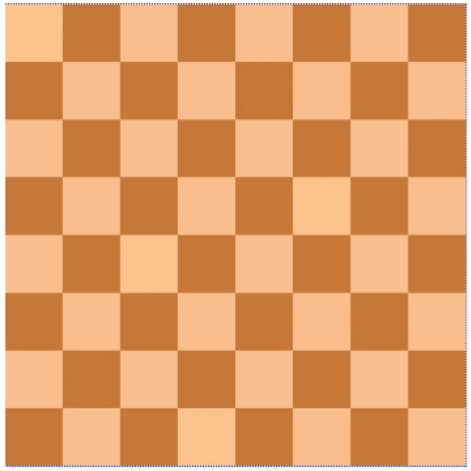
Estado inicial

Estado final

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

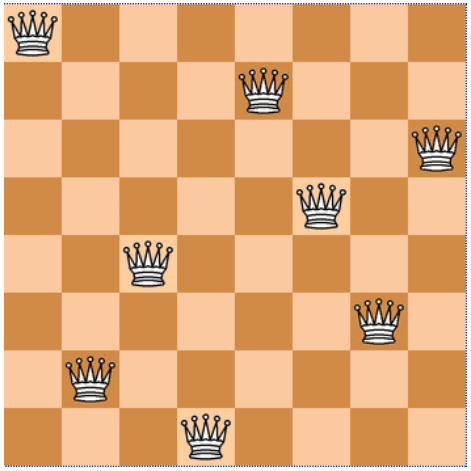
Exemplo: Oito Rainhas

- Modelagem:
  - Variáveis: A1, A2, ... A8, B1, ... B8, ..., H1, ... H8
  - Domínio: 0 e 1
  - Restrições: Não pode ter duas rainhas na mesma linha, coluna ou diagonal



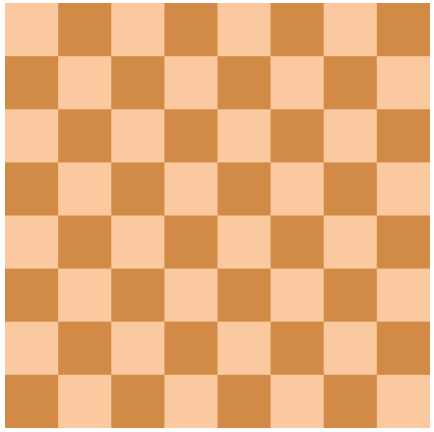
Estado inicial

Estado final



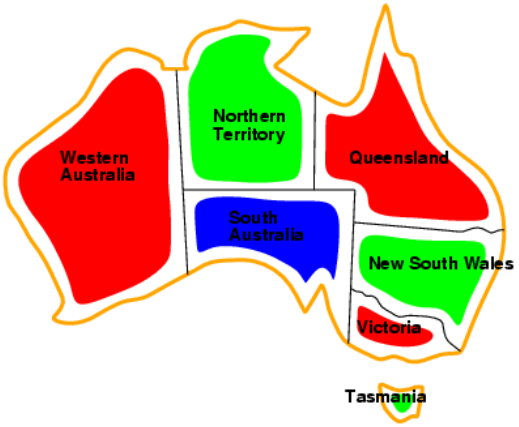
Problema de Satisfação de Restrições (PSR)
Constraint Satisfaction Problem (CSP)

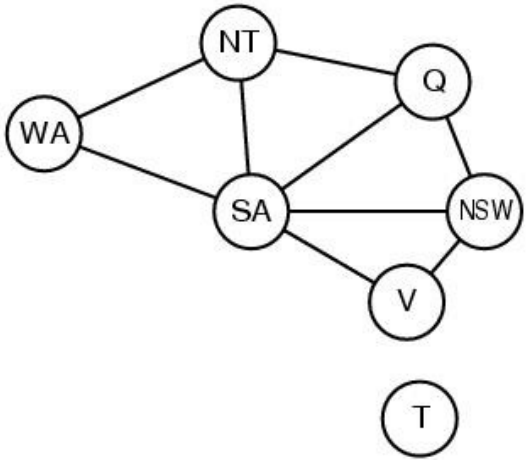
- Pode ser entendido como um problema de busca:
  - Estado inicial: nenhuma variável atribuída
  - Operador: atribuir um valor a uma variável livre, dado que não existe conflito
  - Estado final: atribuição completa e consistente
- *Backtracking*:
  - Busca em profundidade
  - Resolve *n*-rainhas para $n \approx 25$



Como Representar Restrições?

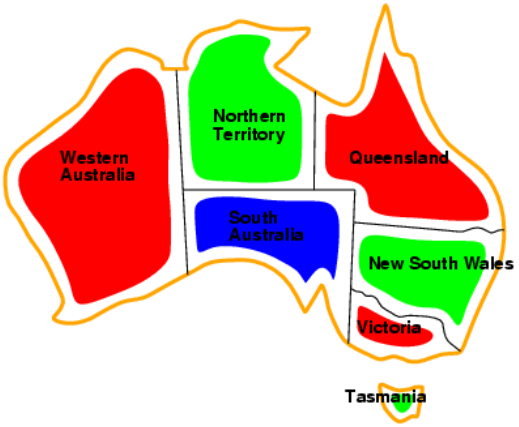
- Restrições podem ser:
  - Unárias: envolvem uma variável. Ex.: SA ≠ verde
  - Binárias: envolvem duas variáveis. Ex.: SA ≠ WA
  - Maior-ordem: envolvem três ou mais variáveis. Ex.: Alldiff(A1...A9)
- Restrições de maior-ordem podem ser quebradas em binárias
- Grafos é uma representação frequente para restrições

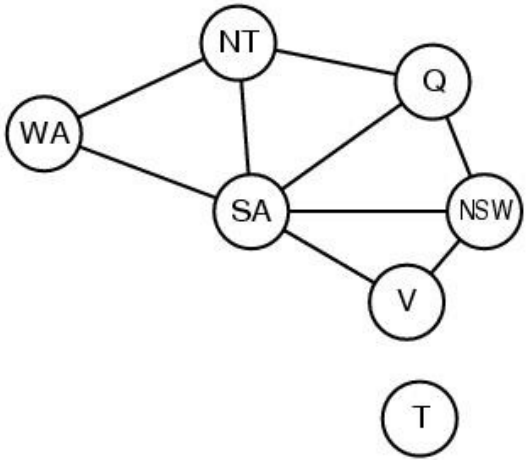




Grafo de Restrições

- Para PSRs binários:
  - Nós são variáveis
  - Arestas são restrições
- Benefício:
  - Padrão de representação
  - Funções genéricas de objetivo e sucessor
  - Pode ser utilizado para simplificar a busca
    - Componentes independentes como Tasmânia





Busca Backtracking

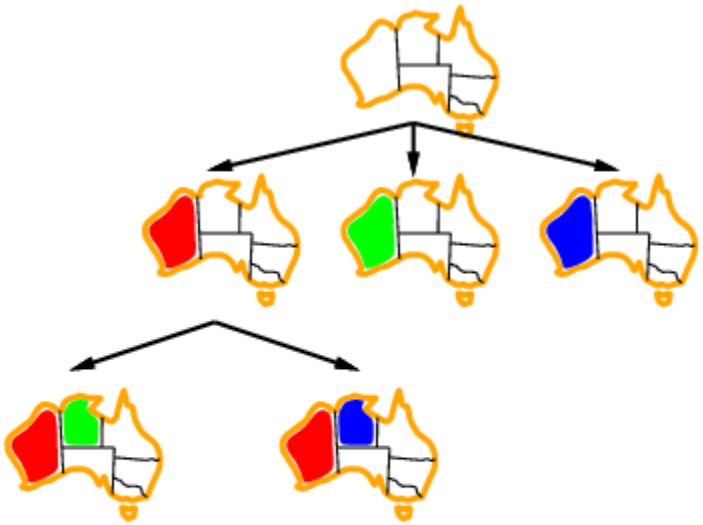
function BACKTRACKING-SEARCH(*csp*) % returns a solution or failure
    return RECURSIVE-BACKTRACKING(*{}* , *csp*)

function RECURSIVE-BACKTRACKING(*assignment*, *csp*) % returns a solution or failure
    if *assignment* is complete **then return** *assignment*
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*],*assignment*,*csp*)
    for each *value* **in** ORDER-DOMAIN-VALUES(*var*, *assignment*, *csp*) **do**
        if *value* is consistent with *assignment* according to CONSTRAINTS[*csp*] **then**
            add *{var=value}* to assignment
            result ← RECURSIVE-BACKTRACKING(*assignment*, *csp*)
            if *result* ≠ *failure* **then return** *result*
            remove *{var=value}* from *assignment*
    return *failure*

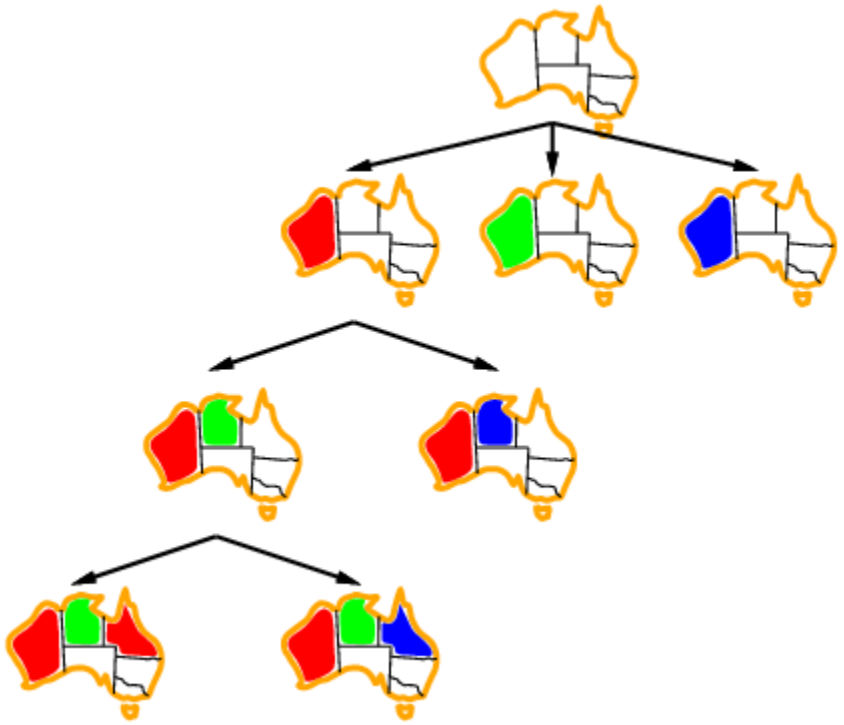
Exemplo Backtracking

Exemplo Backtracking

Exemplo Backtracking

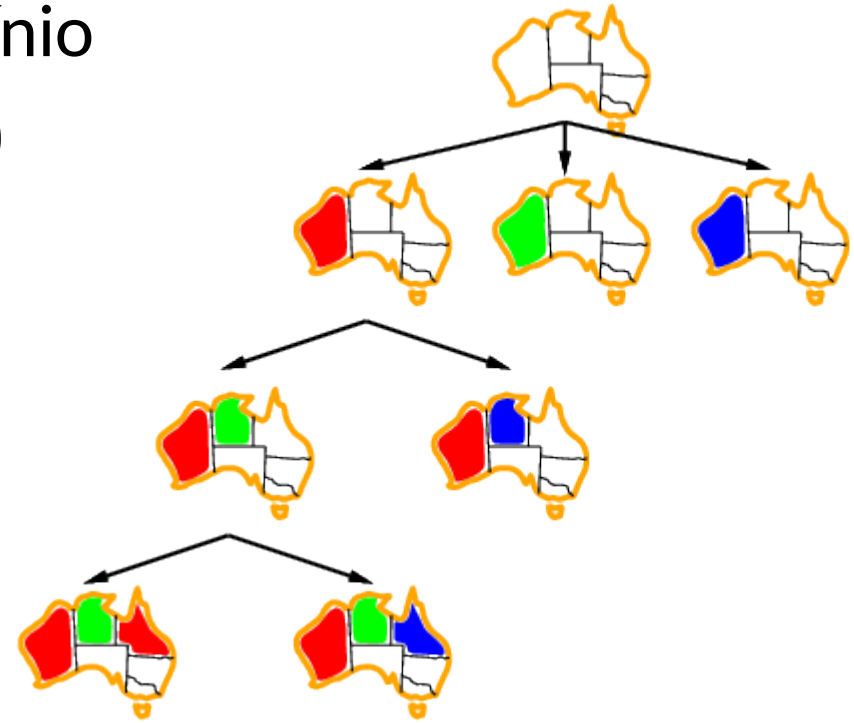


Exemplo Backtracking



Eficiência do Backtracking

- Complexidade de execução do backtracking:
  - n: número de variáveis
  - d: tamanho do domínio
  - Complexidade: $O(d^n)$



Vejamos 2 exemplos práticos de Backtracking

- Calcule todas as palavras (dado alfabeto inteiro) de 5 caracteres. Imprima todas
- Percurso de um cavalo em um tabuleiro de xadrez, de tamanho N x N.

Exemplo 1: Palavras

- Vamos supor queremos gerar todas as possíveis “palavras” de MAX letras utilizando uma string.
- Cada posição do vetor irá armazenar uma letra.
- Tal problema simples pode ser utilizado para ilustrar a construção de um algoritmo de *backtracking*.

Exemplo 1: Palavras

- Todo algoritmo de *backtracking* possui algumas características em comum:
  - Uma condição que verifica se uma solução foi encontrada;
  - Um laço de tenta todos os valores possíveis para uma única variável discreta;
  - Uma recursão que irá atribuir valores às variáveis sem valores até o momento.

Exemplo 1: Palavras

```c
#include<stdio.h>
#define MAX 5

void backtracking(char v[], int k)
{
    char letra;

    if (k == MAX)
        printf("%s\n", v);
    else
        for (letra = 'a'; letra <= 'z'; letra++)
        {
            v[k] = letra;
            backtracking(v, k + 1);
        }
}

int main()
{
    char v[MAX+1];

    v[MAX] = '\0';
    backtracking(v, 0);
    return 0;
}
```

Exemplo 2: Percurso do Cavalo

- Um segundo exemplo de problema que se pode tratar com *backtracking* é de gerar percursos em tabuleiros:
  - Pode-se encontrar um percurso realizado por um cavalo que visite todas as posições de um tabuleiro, sem passar pela mesma posição duas vezes.

Exemplo 2: Percurso do Cavalo

```
#include<stdio.h>
#define SIZE 8

bool marked[SIZE][SIZE];

char moves[8][2] = {-1, -2,
                    -2, -1,
                    -2, 1,
                    -1, 2,
                     1, 2,
                     2, 1,
                     2, -1,
                     1, -2 };

bool valid(char v) {
    return (v >= 0) && (v < SIZE);
}
```

Exemplo 2: Percurso do Cavalo

```
void backtracking(char lin, char col, char k) {
    char new_lin, new_col, i;

    if (k == SIZE*SIZE-1) {
      printf("There exists a path!\n");
    } else
       for (i = 0; i < 8; i++) {
           new_col = col + moves[i][0];
           new_lin = lin + moves[i][1];
           if (valid(new_lin) && valid(new_col) && !marked[new_lin][new_col]) {
               marked[new_lin][new_col] = true;
               backtracking(new_lin, new_col, k+1);
               marked[new_lin][new_col] = false;
           }
       }
}
```

Exemplo 2: Percurso do Cavalo

```
int main() {
    int i, j;
    char c;

    for (i = 0; i < SIZE; i++)
        for (j = 0; j < SIZE; j++)
            marked[i][j] = false;

    marked[SIZE/2][SIZE/2] = true;
    backtracking(SIZE/2, SIZE/2, 0);
}
```

Como Melhorar a Eficiência do Backtracking?

- Métodos de propósito geral podem fornecer grandes ganhos de desempenho:
  - Qual variável deve ser a próxima a ser atribuída?
  - Em qual ordem os valores devem ser tentados?
  - Pode-se detectar falhas inevitáveis mais cedo?

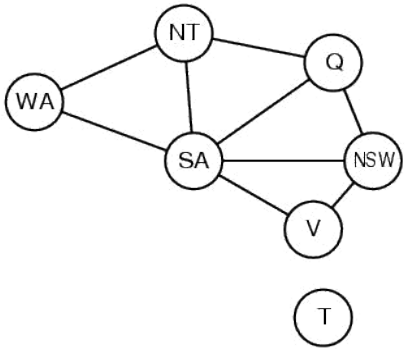
Próxima Variável a ser Atribuída

function BACKTRACKING-SEARCH(*csp*) % returns a solution or failure
    return RECURSIVE-BACKTRACKING(*{}* , *csp*)

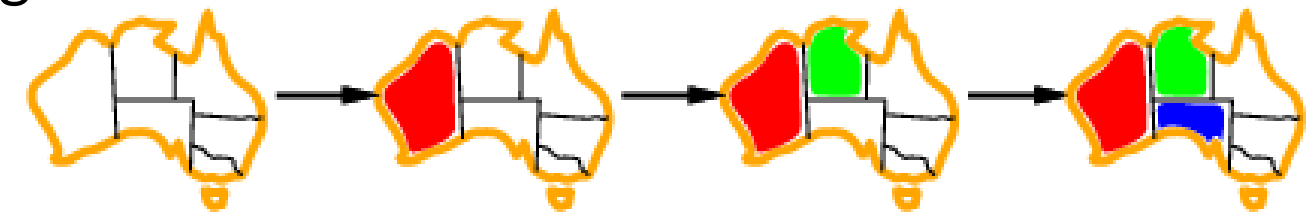
function RECURSIVE-BACKTRACKING(*assignment*, *csp*) % returns a solution or failure
    if *assignment* is complete **then return** *assignment*
    var ← **SELECT-UNASSIGNED-VARIABLE**(VARIABLES[*csp*],*assignment*,*csp*)
    for each *value* **in** ORDER-DOMAIN-VALUES(*var*, *assignment*, *csp*) **do**
        if *value* is consistent with *assignment* according to CONSTRAINTS[*csp*] **then**
            add *{var=value}* to assignment
            result ← RECURSIVE-BACKTRACKING(*assignment*, *csp*)
            if *result* ≠ *failure* **then return** *result*
            remove *{var=value}* from *assignment*
    return *failure*

Variável mais Restrita

Mininum remaining values (MRV)



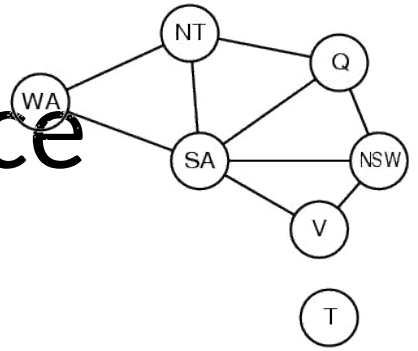
- Variável mais restrita:
  - Escolha a variável com menor número de movimentos legais



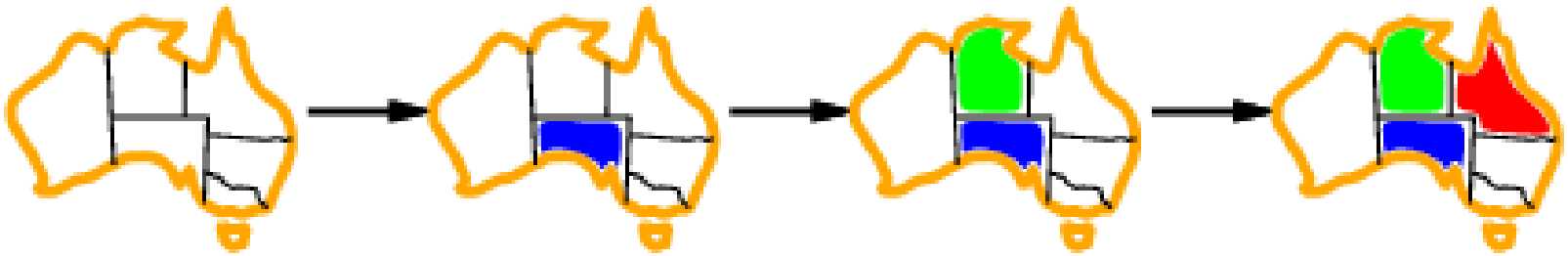
- Também conhecida como heurística mínimo valores remanescentes (MVR)
  - A ideia é eliminar variáveis que vão causar falhar antes, realizando assim uma poda!

Heurística Grau do vértice

degree heuristc



- Suponha o grafo inicial, não colorido:
  - Qual a validade da heurística MVR?
    - Nenhuma !!! todas variáveis podem ser coloridas com 3 cores ...



- Heurística grau: selecione a variável que tenha o maior número de restrições a outras variáveis não atribuídas
  - Vertice de maior grau !
- No geral, MVR é melhor do que grau, então utiliza-se grau como desempate para MVR.

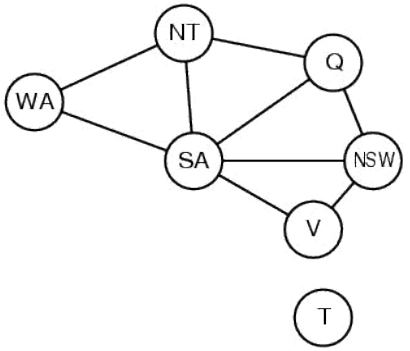
Ordem dos Valores a serem Atribuídos

function BACKTRACKING-SEARCH(*csp*) % returns a solution or failure
    return RECURSIVE-BACKTRACKING(*{}* , *csp*)

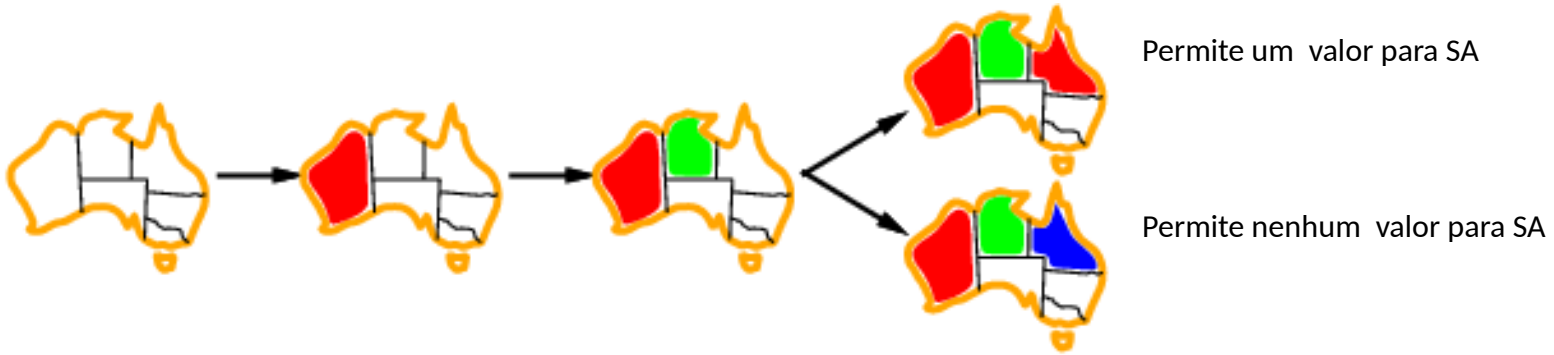
function RECURSIVE-BACKTRACKING(*assignment*, *csp*) % returns a solution or failure
    if *assignment* is complete **then return** *assignment*
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*],*assignment*,*csp*)
    for each *value* **in** **ORDER-DOMAIN-VALUES**(*var*, *assignment*, *csp*) **do**
        if *value* is consistent with *assignment* according to CONSTRAINTS[*csp*] **then**
            add *{var=value}* to assignment
            result ← RECURSIVE-BACKTRACKING(*assignment*, *csp*)
            if *result* ≠ *failure* **then return** *result*
            remove *{var=value}* from *assignment*
    return *failure*

Valor Menos Restritivo

least restrictive value



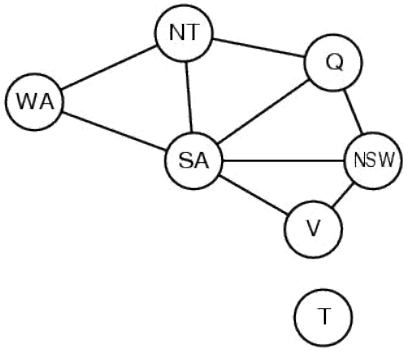
- Dada uma variável, escolha o valor menos restritivo:
  - Aquele que remove o menor número de valores para outras variáveis



- Combinar essas heurísticas faz com que o problema das 1000-rainhas seja possível

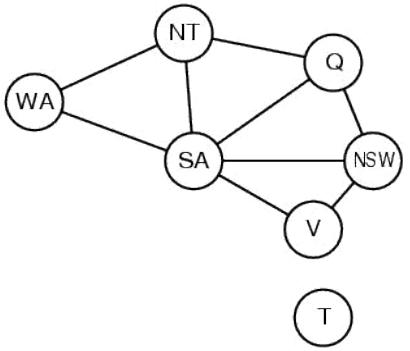
Verificação Adiante

forward checking

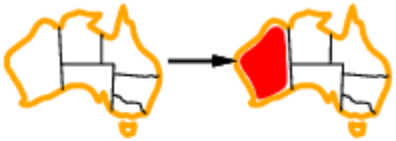


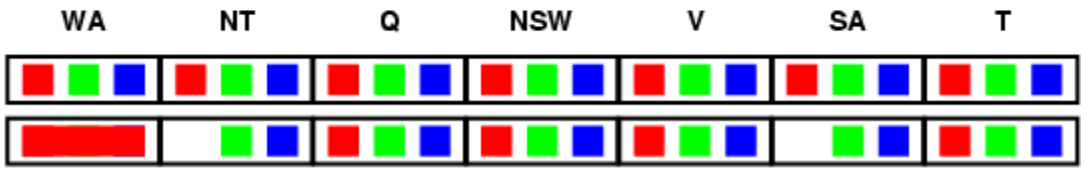
- Manter os valores legais remanescentes para variáveis não atribuídas
- Retroceder a busca quando uma variável não possuir valores legais

Verificação Adiante

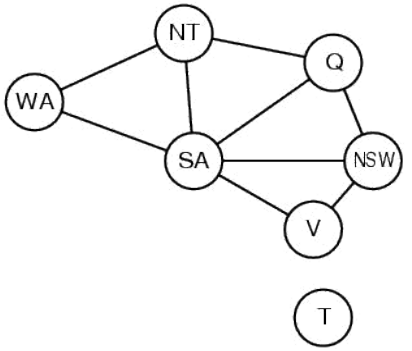


- Manter os valores legais remanescentes para variáveis não atribuídas
- Retroceder a busca quando uma variável não possuir valores legais



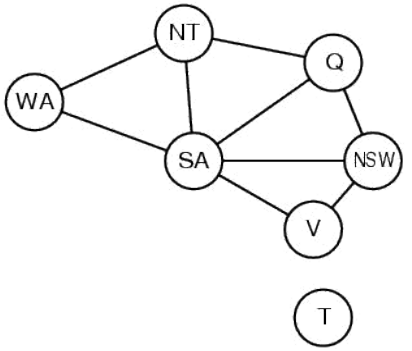


Verificação Adiante

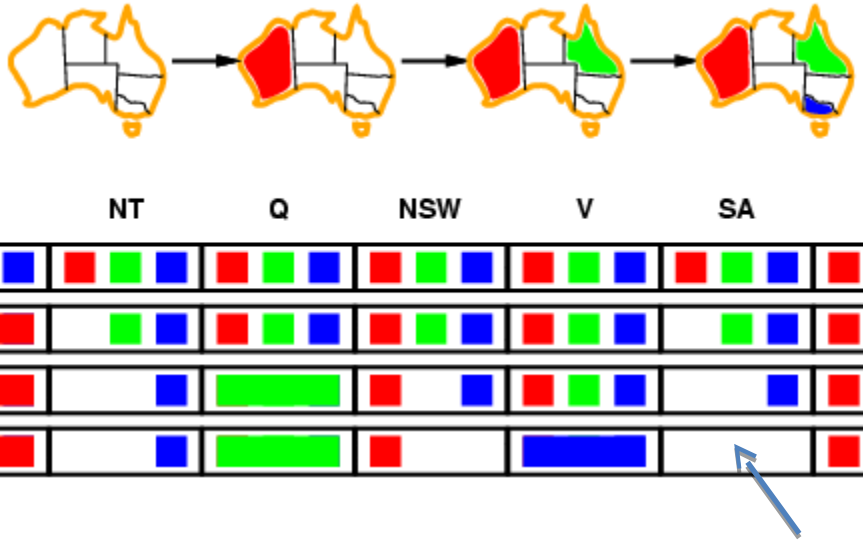


- Manter os valores legais remanescentes para variáveis não atribuídas
- Retroceder a busca quando uma variável não possuir valores legais

Verificação Adiante

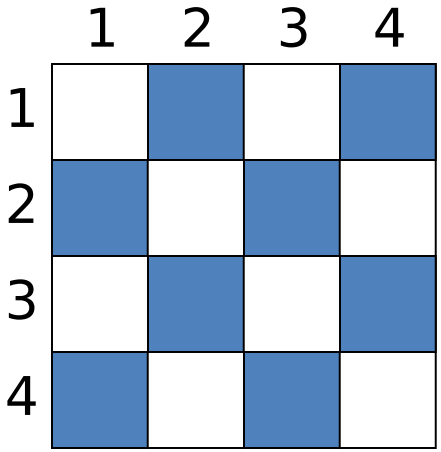


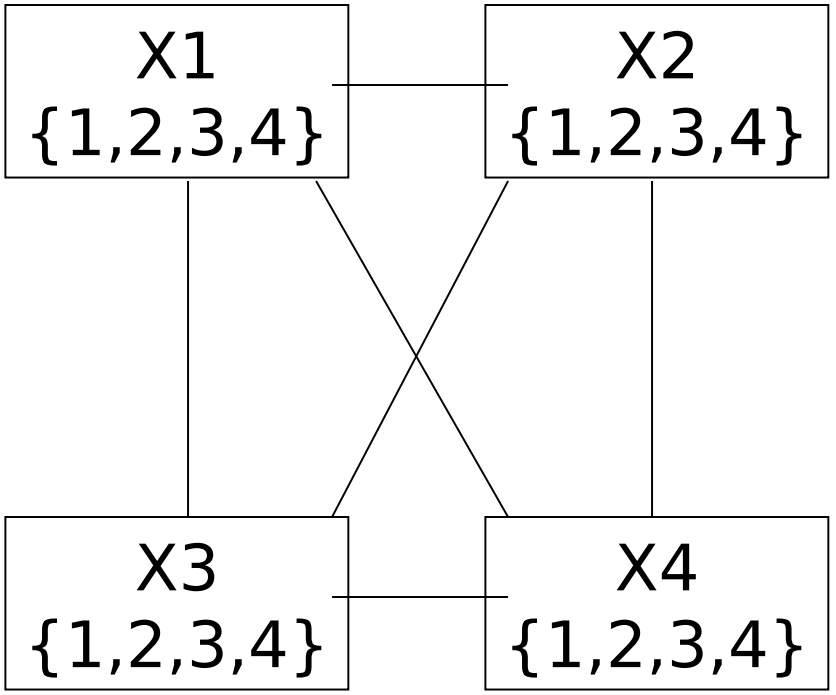
- Manter os valores legais remanescentes para variáveis não atribuídas
- Retroceder a busca quando uma variável não possuir valores legais



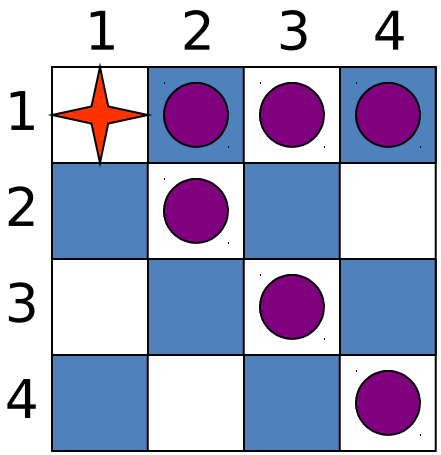
Nenhum valor para SA: backtracking

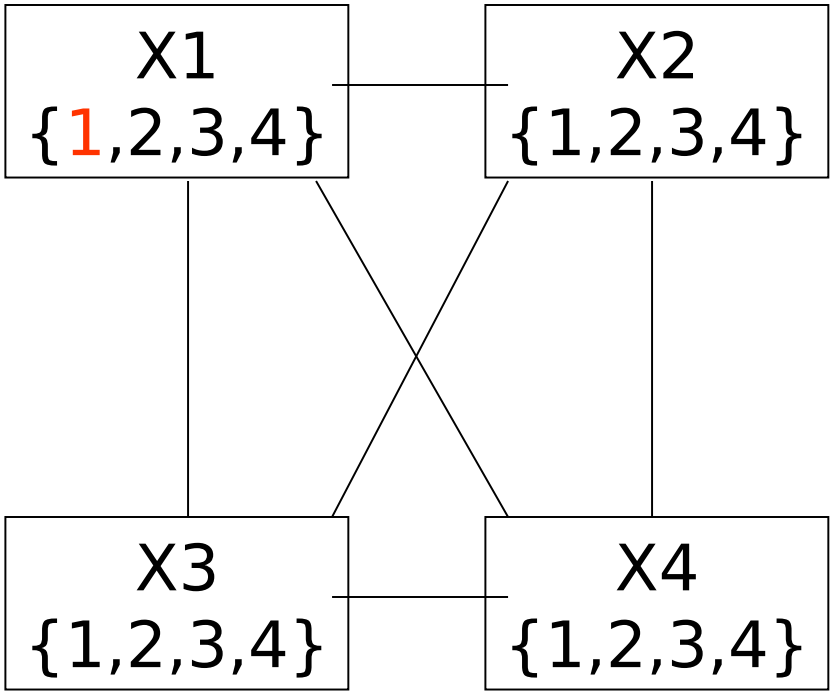
Verificação Adiante: 4-Rainhas



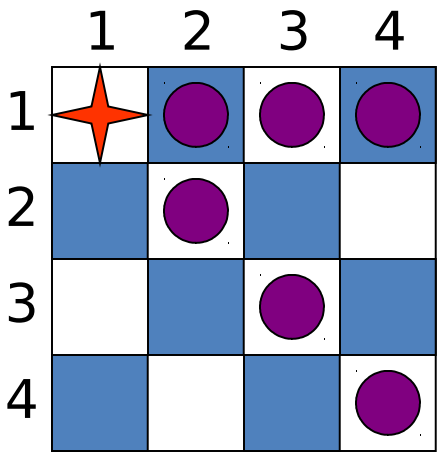


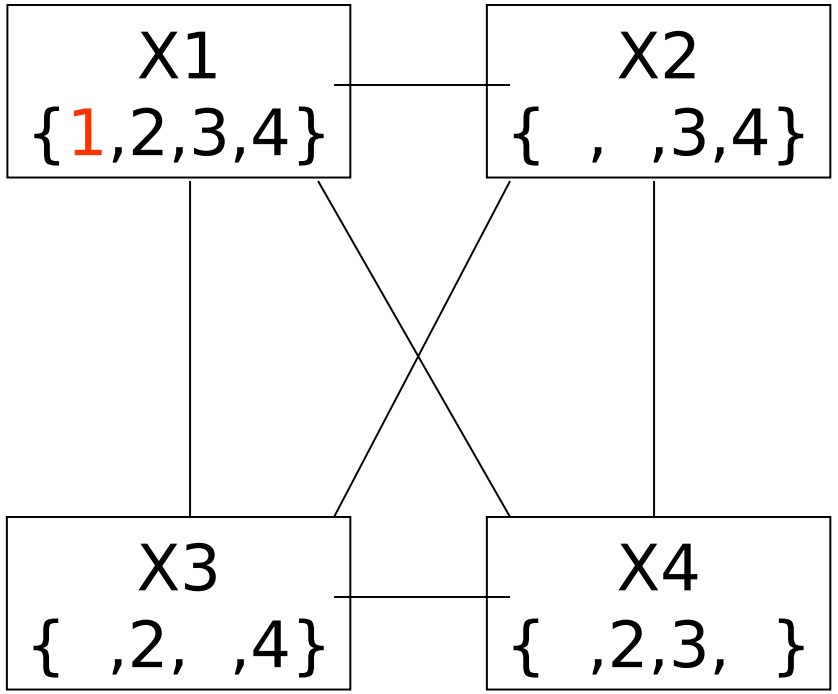
Verificação Adiante: 4-Rainhas



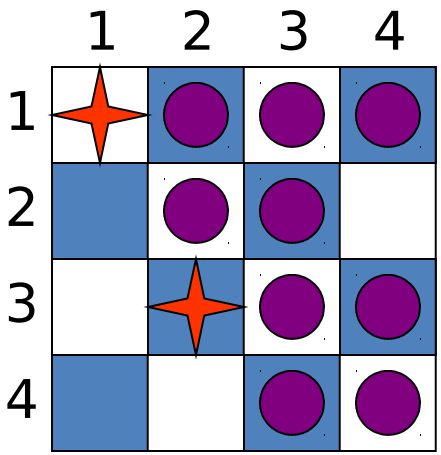


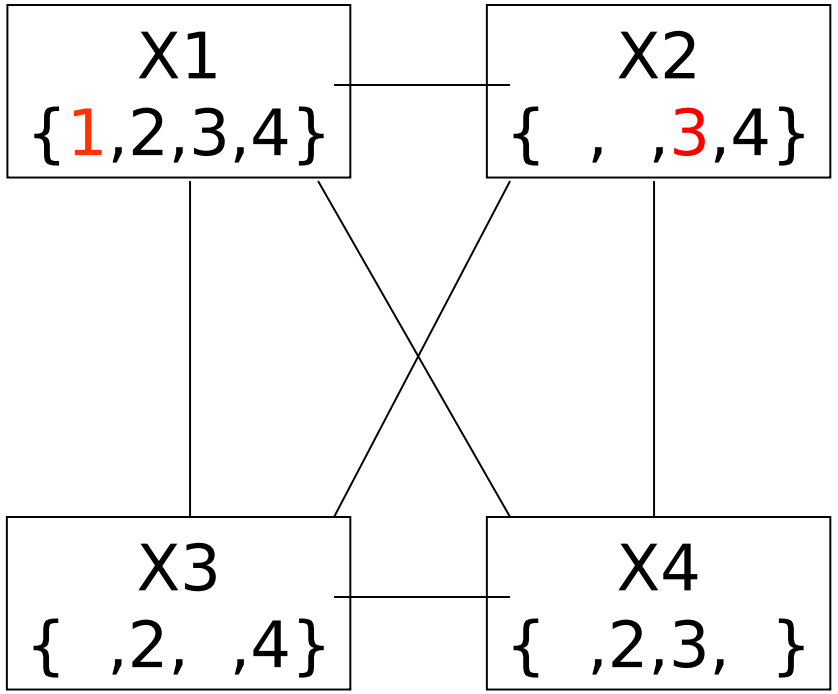
Verificação Adiante: 4-Rainhas



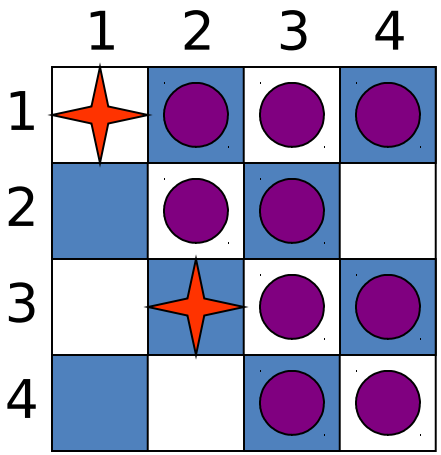


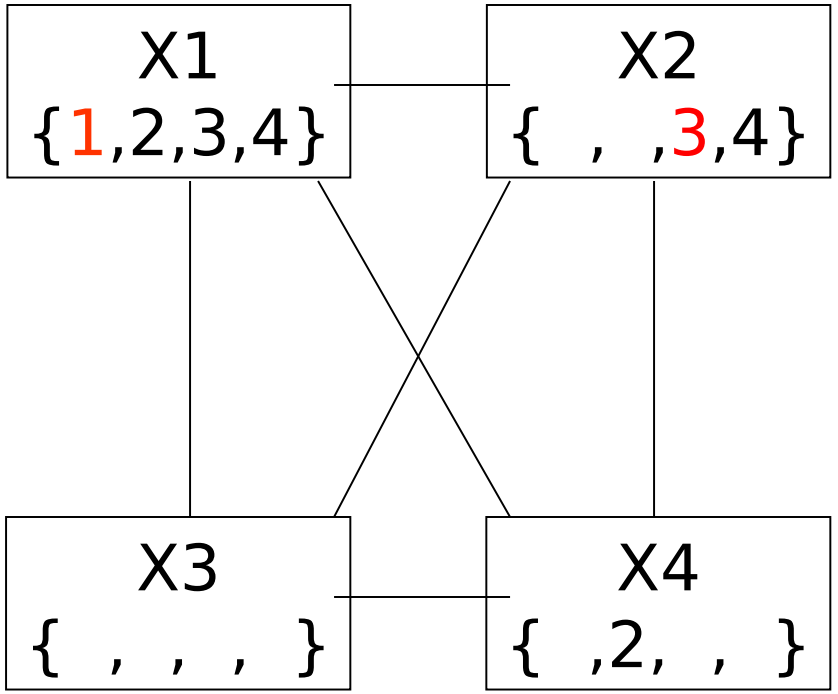
Verificação Adiante: 4-Rainhas



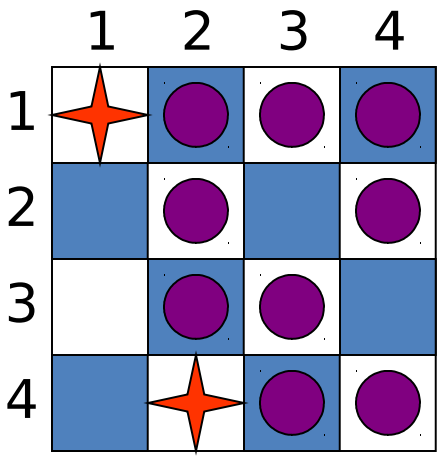


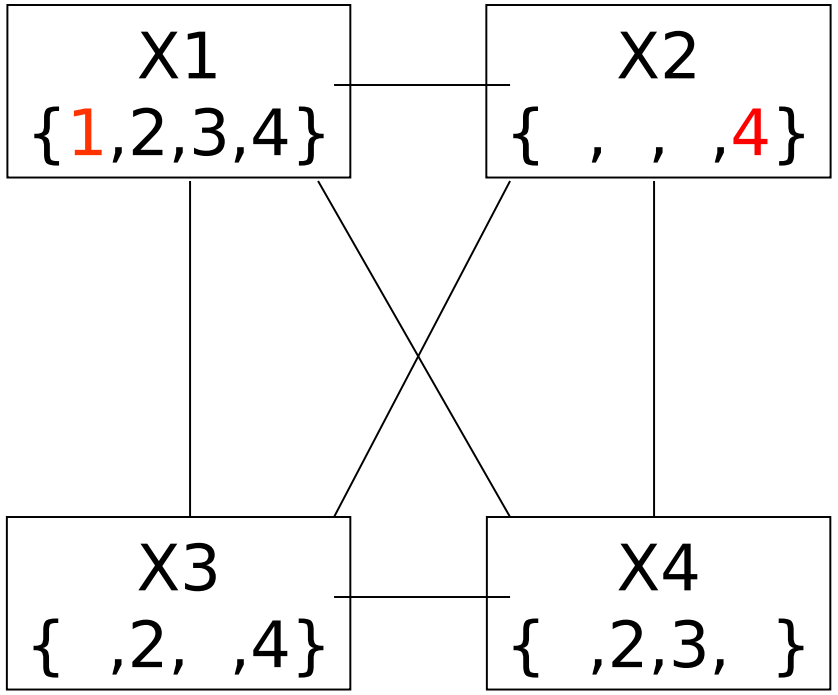
Verificação Adiante: 4-Rainhas



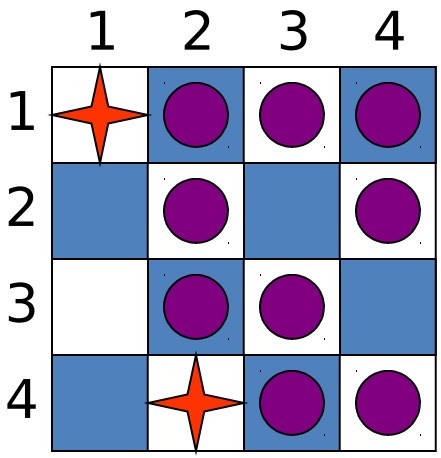


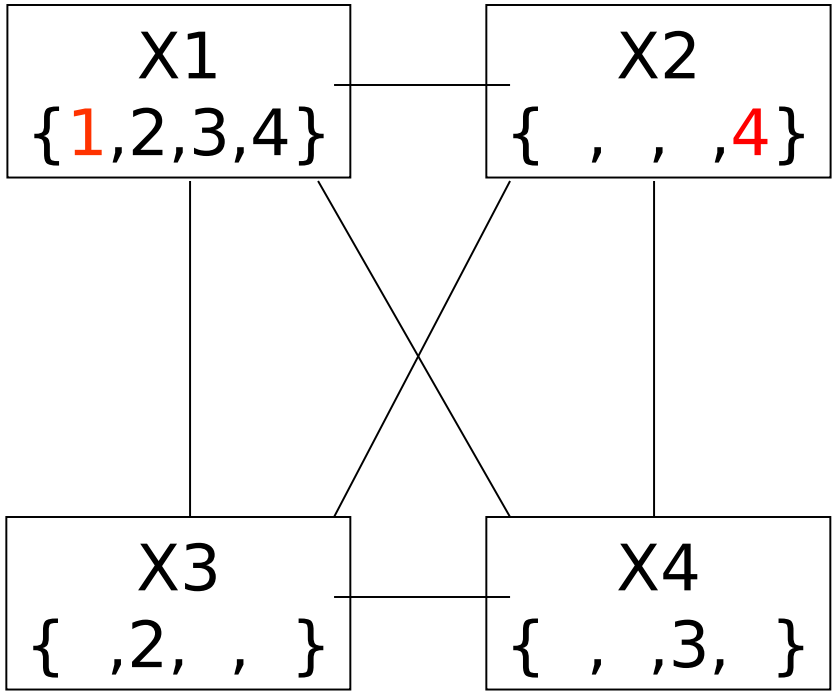
Verificação Adiante: 4-Rainhas



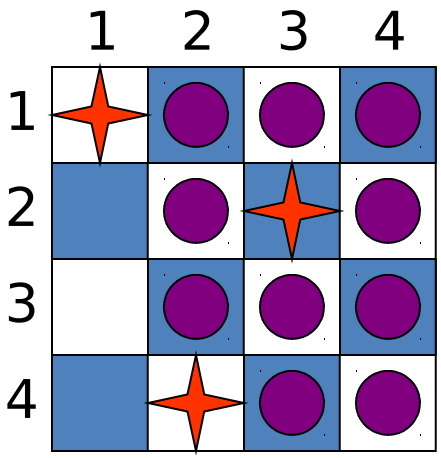


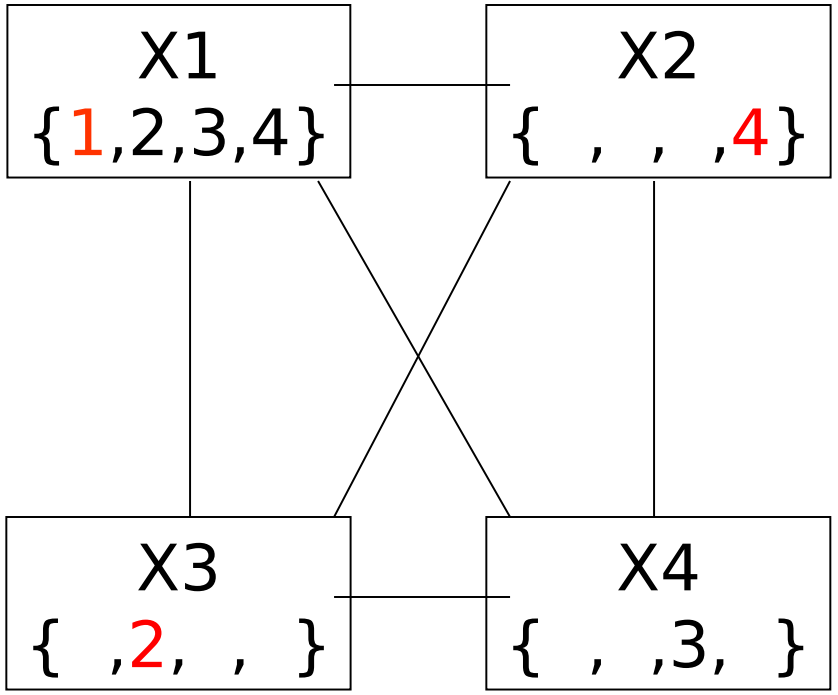
Verificação Adiante: 4-Rainhas



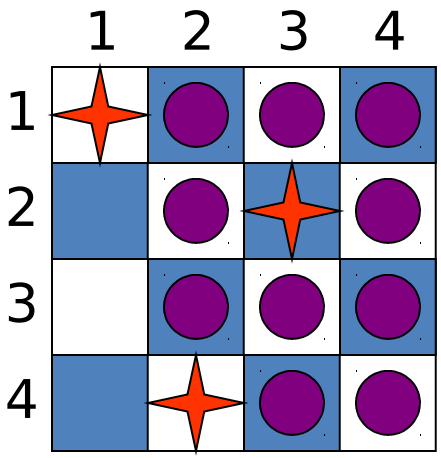


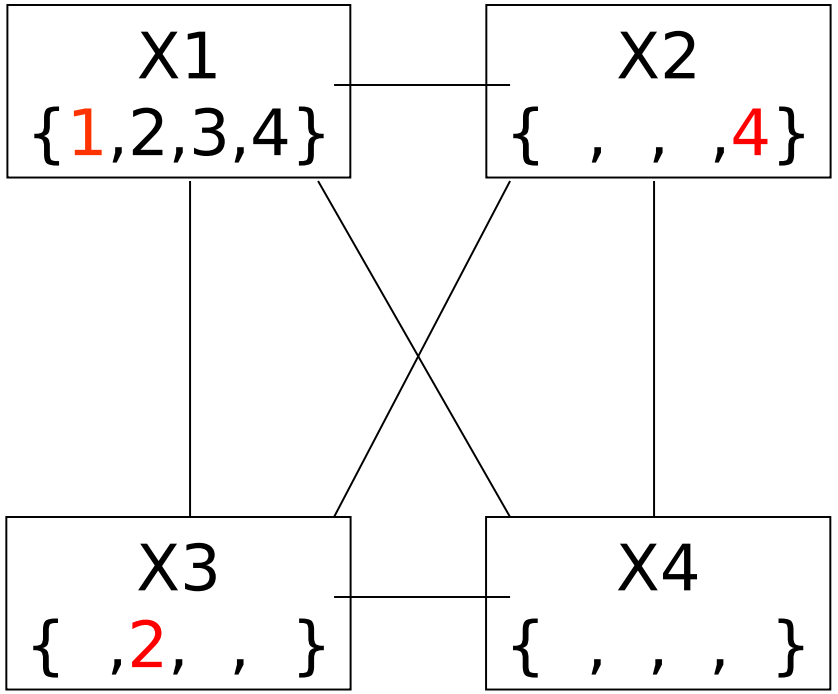
Verificação Adiante: 4-Rainhas



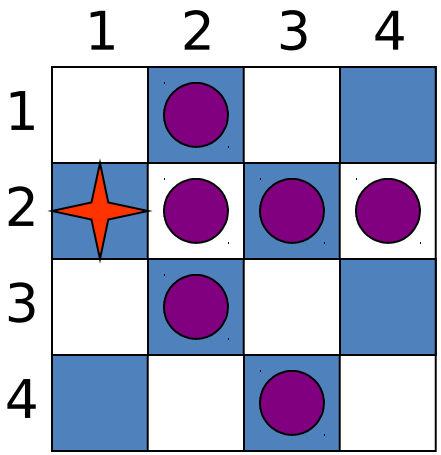


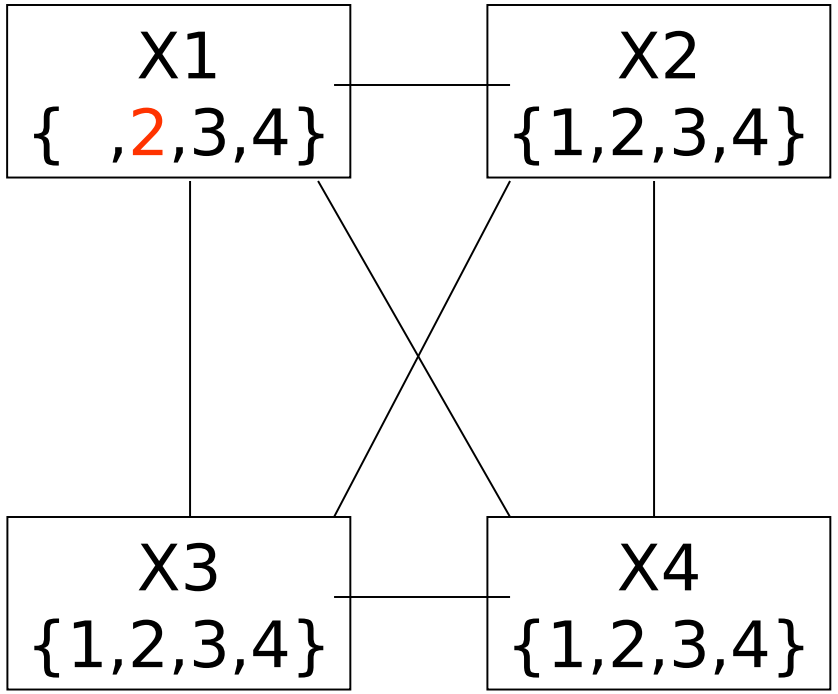
Verificação Adiante: 4-Rainhas



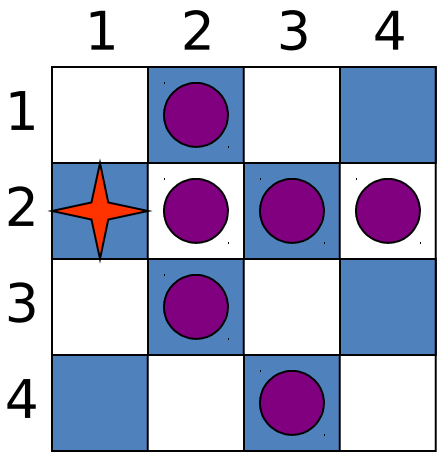


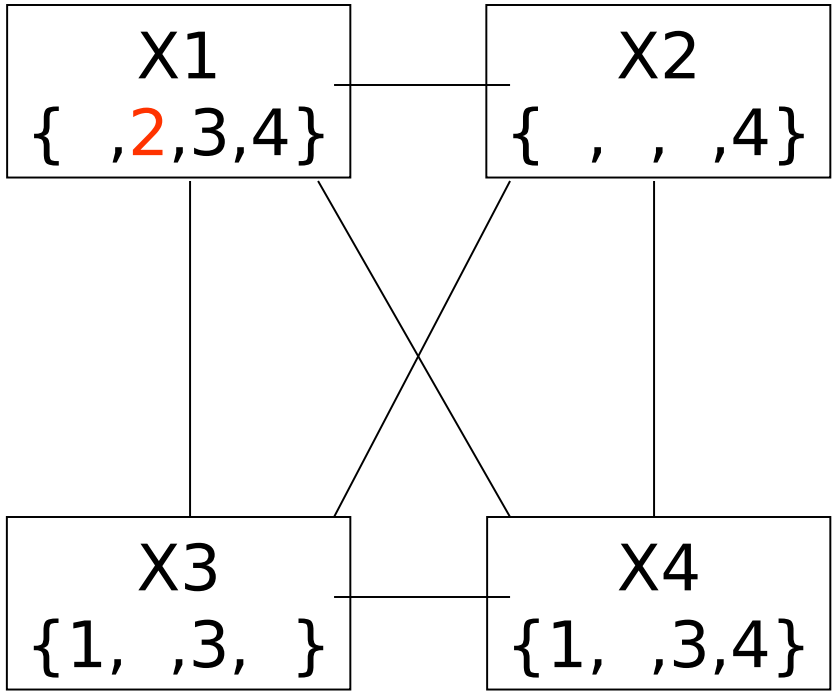
Verificação Adiante: 4-Rainhas



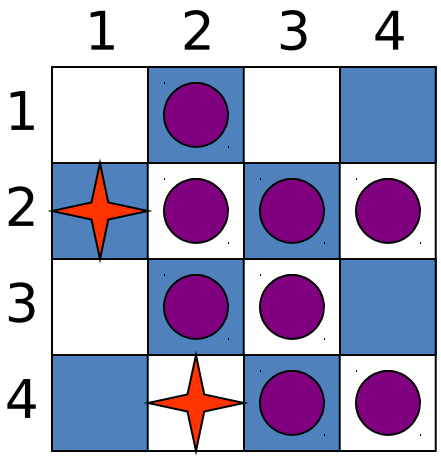


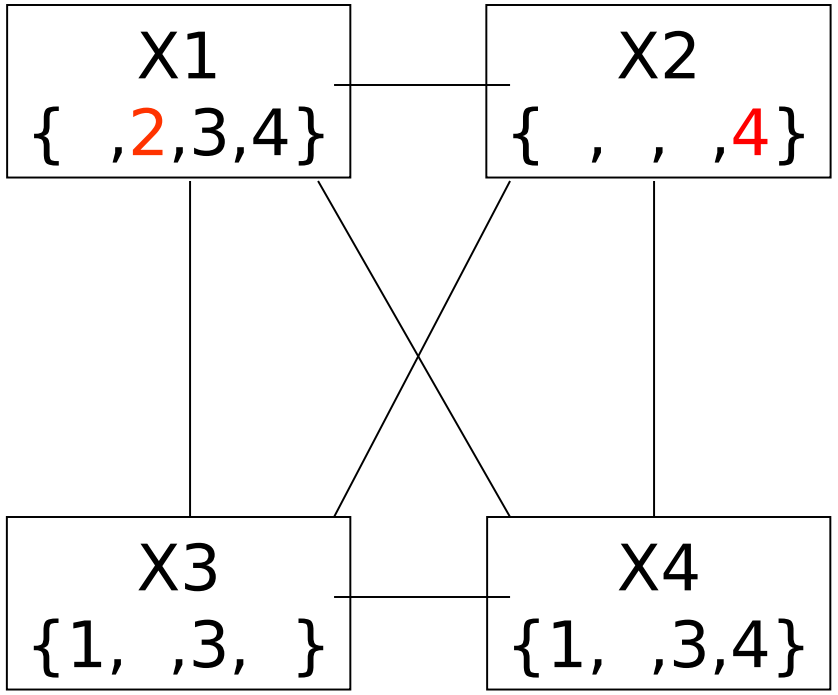
Verificação Adiante: 4-Rainhas



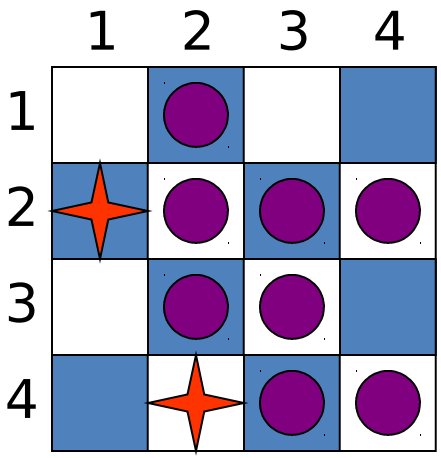


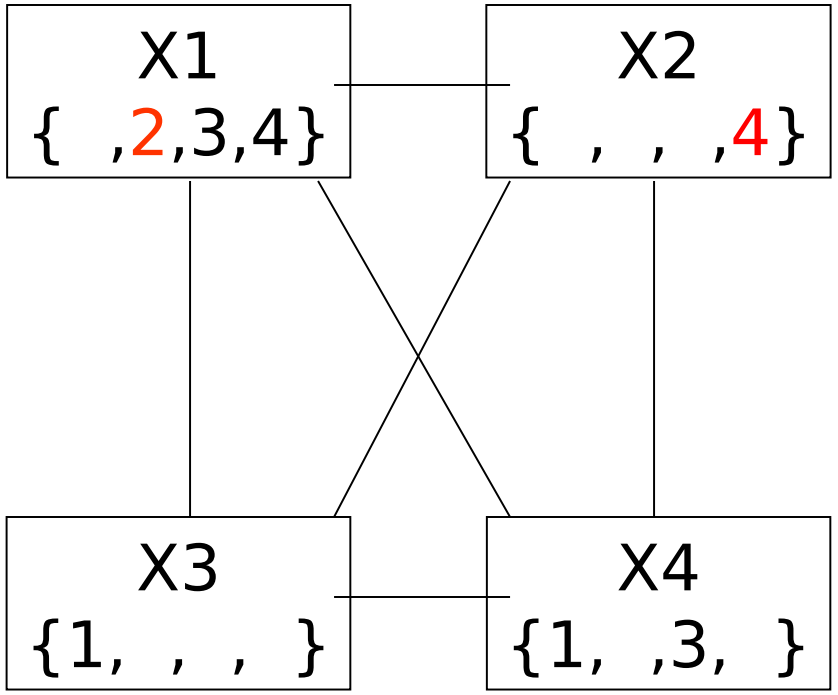
Verificação Adiante: 4-Rainhas



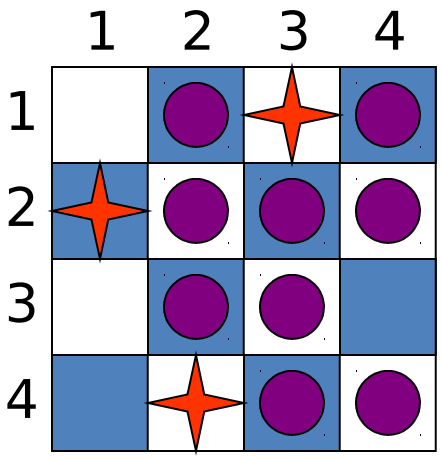


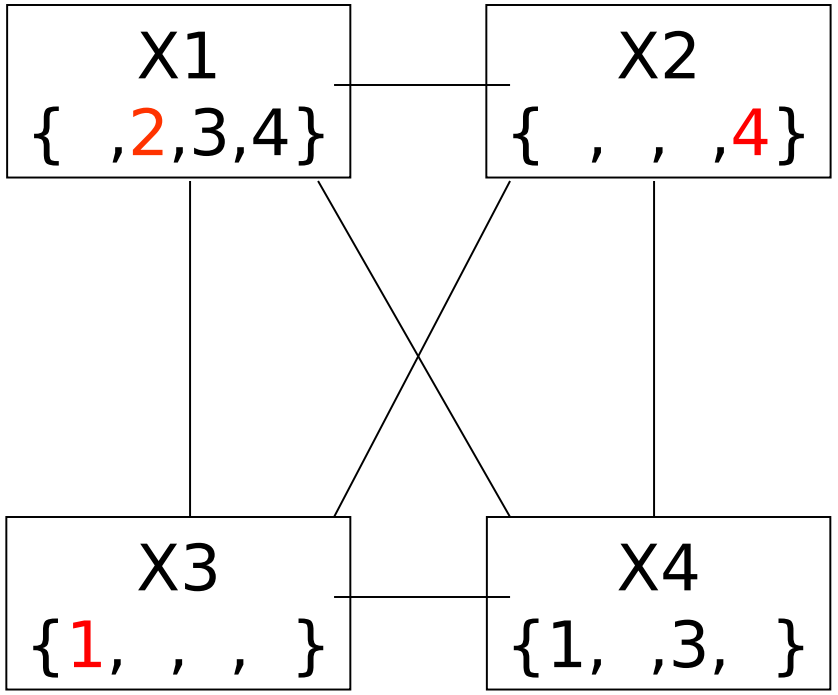
Verificação Adiante: 4-Rainhas



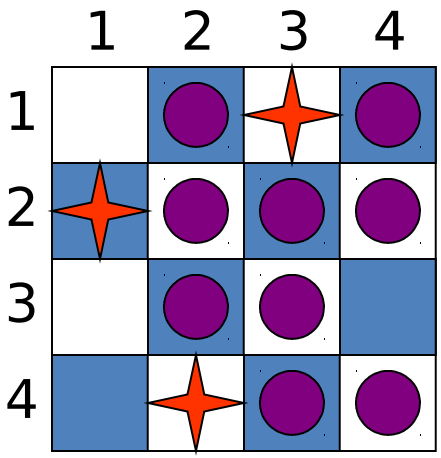


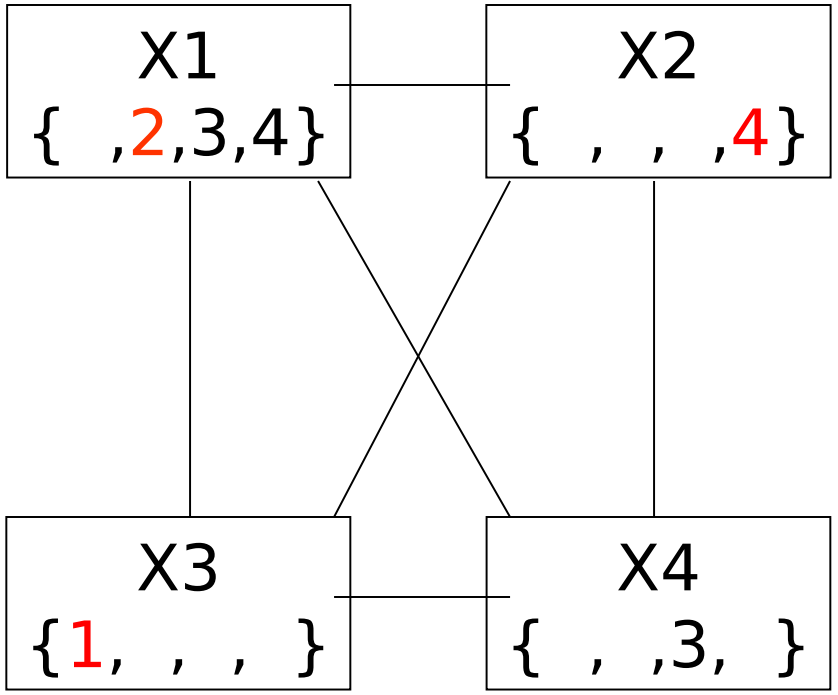
Verificação Adiante: 4-Rainhas



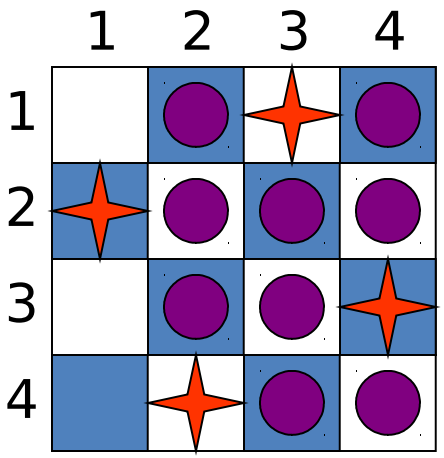


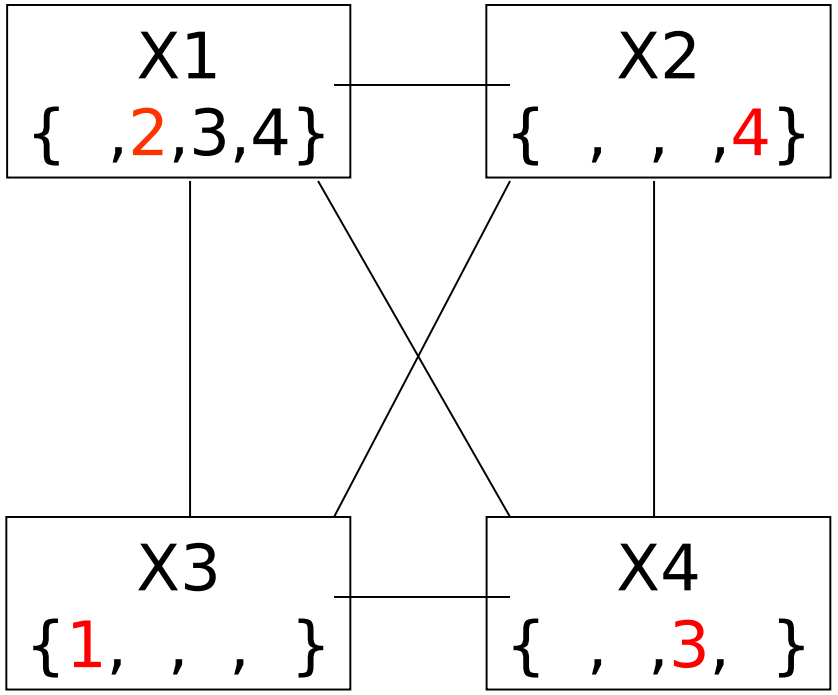
Verificação Adiante: 4-Rainhas





Verificação Adiante: 4-Rainhas





Busca A*

- Esta é uma forma bastante inteligente de poda para problemas que requerem solução rápida
- Quem tiver curiosidade, veja o problema UVA 10067 – Playing with Wheels
  - Este problema é classico backtracking, mas não falha (limite de tempo excedido) com a solução clássica !
- Vejamos o que isso quer dizer...

Quebra- cabeça de 8 (Eight-puzzle)

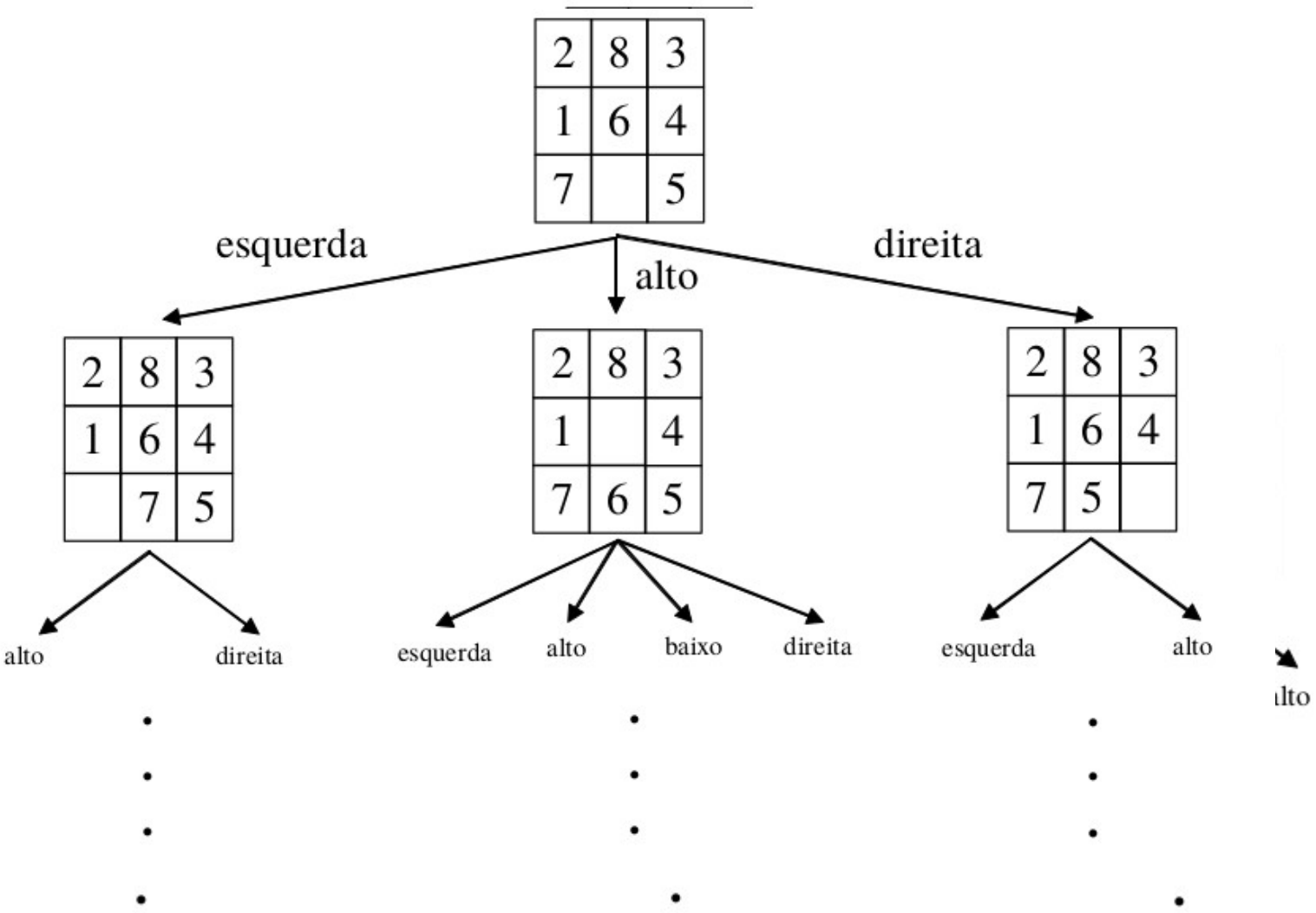
1	2	3
8		4
7	6	5

- O que se vê acima é a configuração final desejada.

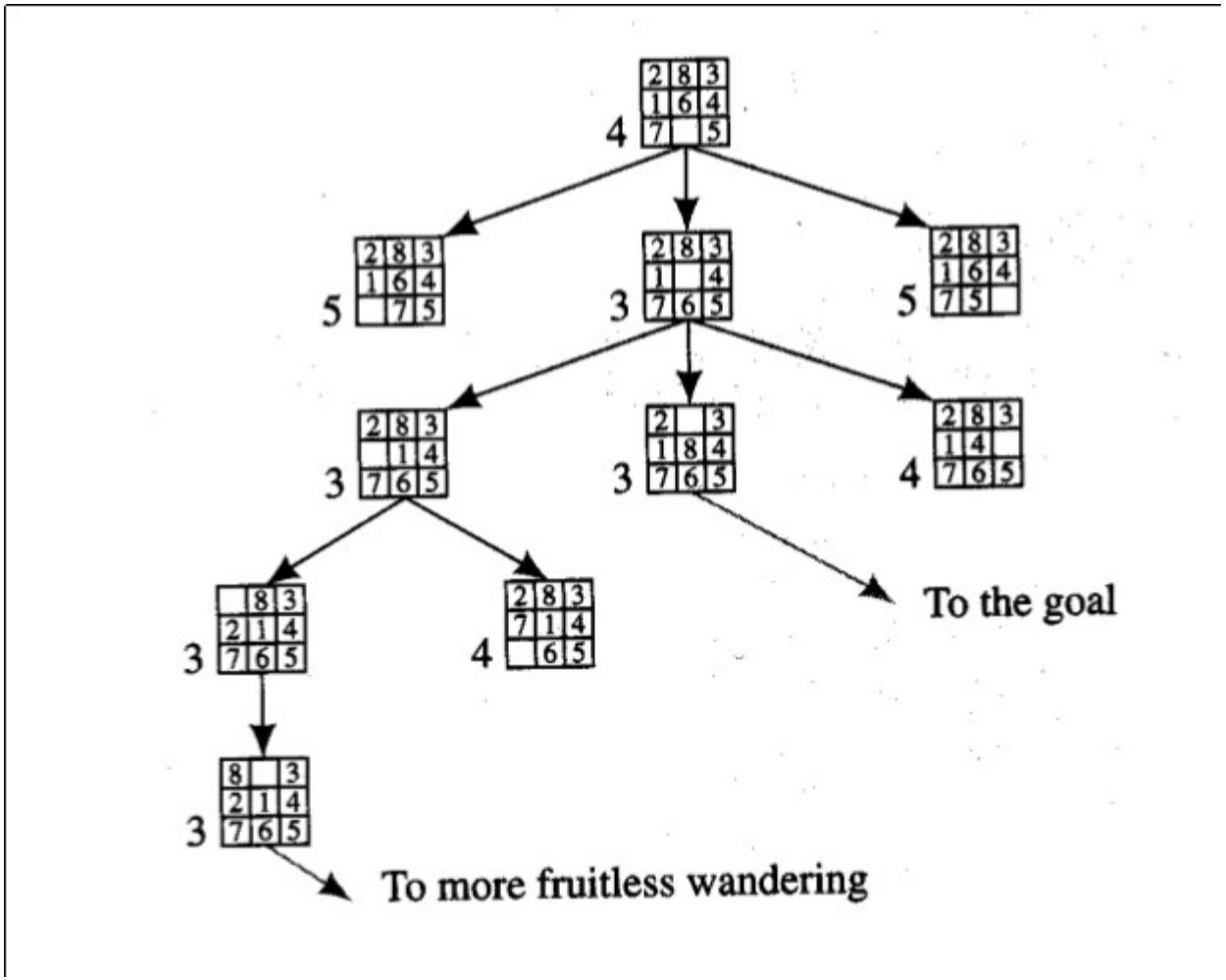
2	8	3
1	6	4
7		5

- Acima, uma configuração n qualquer,:
  - Qual seria uma solução gulosa para este problema?
  - Usaria a seguinte heurística h(n) cujo custo é "distância" de n até a solução final
    - Neste caso, contaria quantos nros estão fora do lugar e escolheria a transição de menor valor.

Movimentos do jogo



Estratégia Gulosa

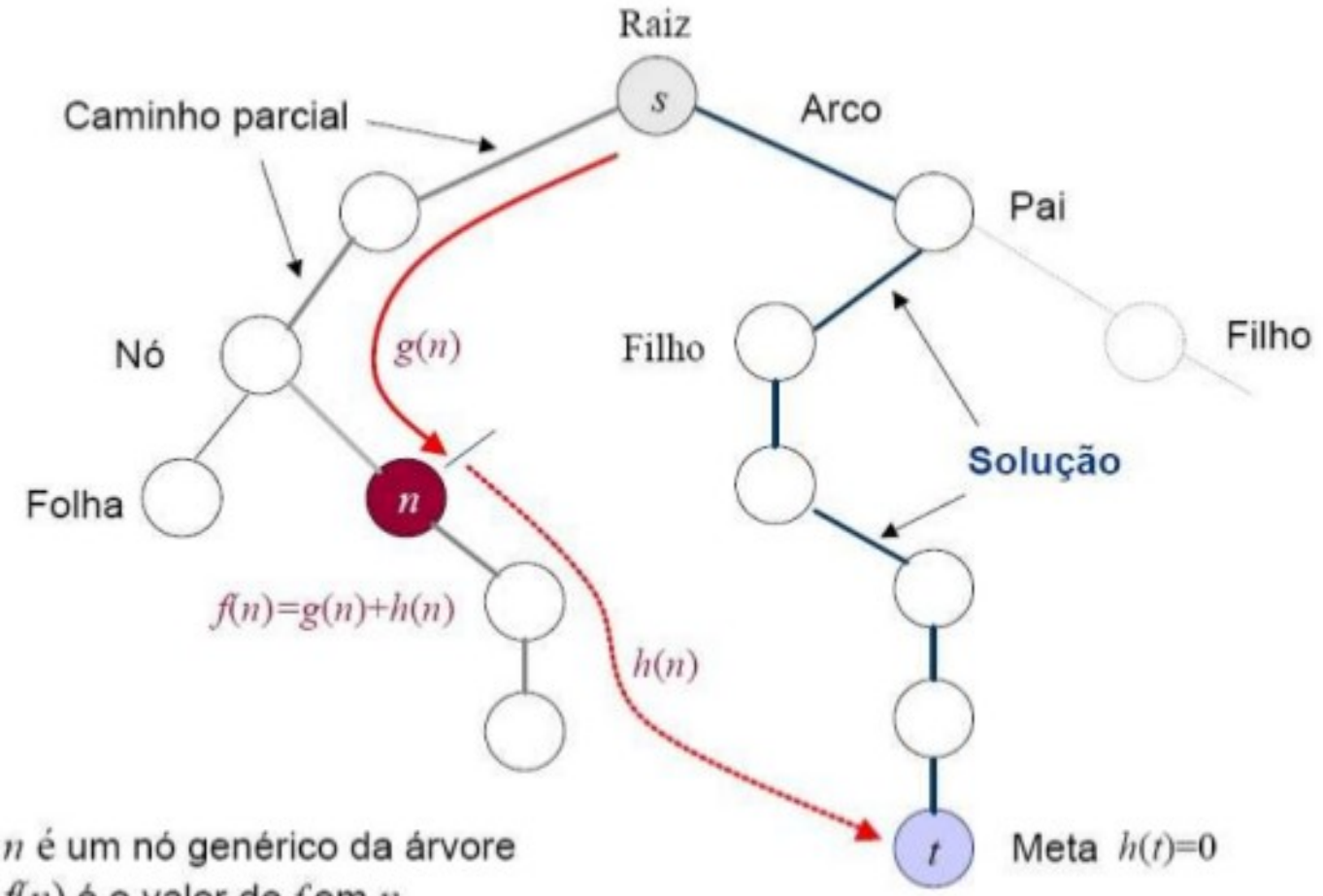


- A estratégia de busca se resumiu em f(n) = h(n), ou seja, a heurística consistiu apenas da estima do custo de n até a meta final !

Busca A*

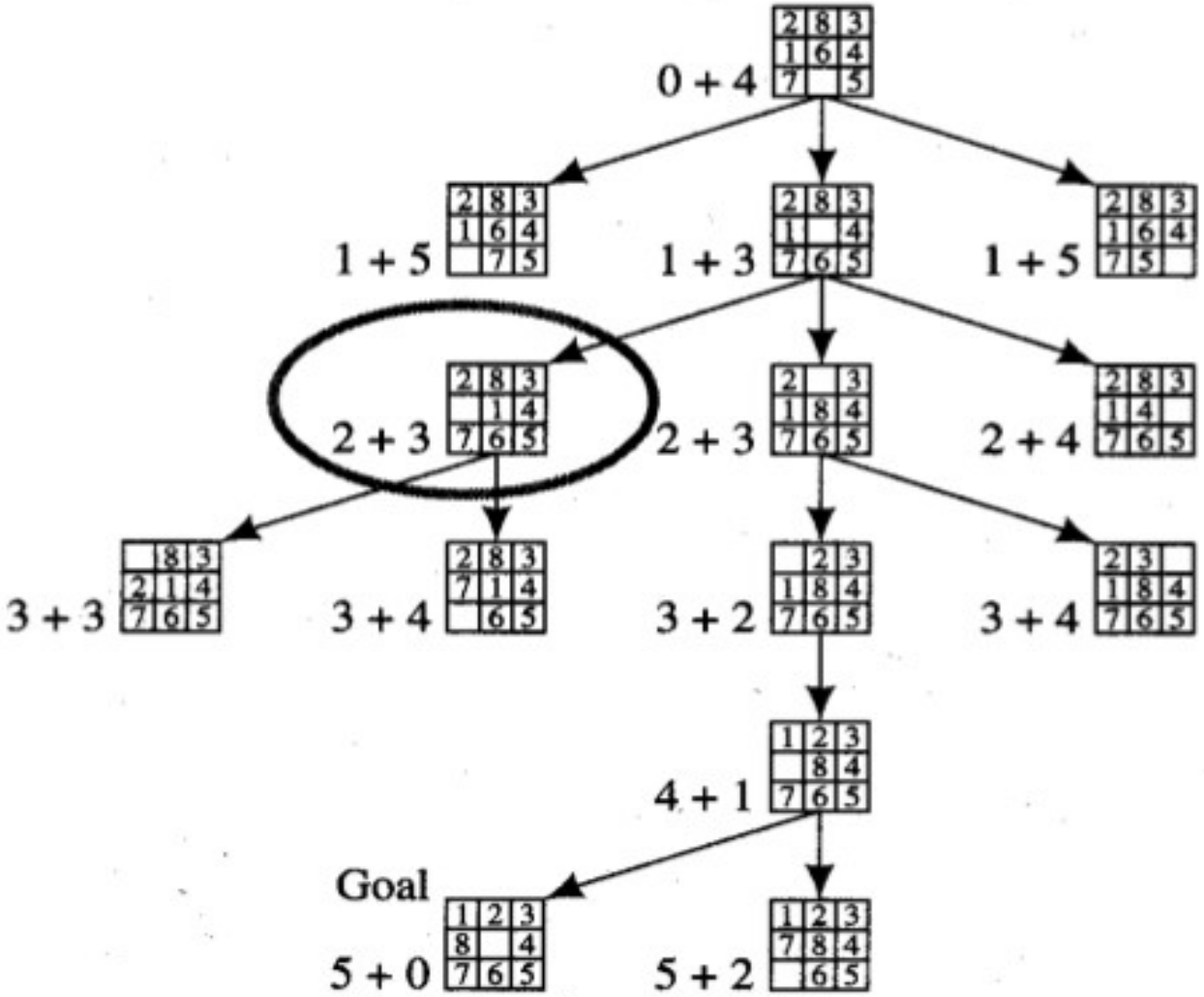
- A procura do melhor caminho até a solução obedece a seguinte fórmula:
  - f(n) = g(n) + h(n)
  - g(n): o custo para se chegar ao nó n, a partir da solução inicial
  - h(n): o quão distante se está da solução final
    - Esta é a mesma eurística utilizada na busca gulosa
    - IMPORTANTE> esta heurística deve ser sempre admissível, ou seja, nunca deve sobre estimar o custo real até o final
      - Escolher h(n) como a quantidade de dígitos for a do lugar é sempre <= ao real custo até o final (para cada nro for a da posição, é necessário ao menos um movimento para colocar o número na posição correta!

Busca A*



n é um nó genérico da árvore
$f(n)$ é o valor de f em n
$g(n)$ é o custo da trajetória até n
$h(n)$ é uma *estimativa* do custo de n até a meta

Busca A* - veja que legal



Busca A* - Propriedades

- h(n): deve ser admssível ou otimista
- f(n) nunca decresce ao longo do caminho → monotonicidade
- A primeira solução encontrada é ótima
- O que vc tem a dizer do uso de memória da A* ??
  - Observe que devemos usar uma priority queue! E ela pode crescer bastante...