Algoritmos Avançados SCC-210

Backtracking

Joao Batista

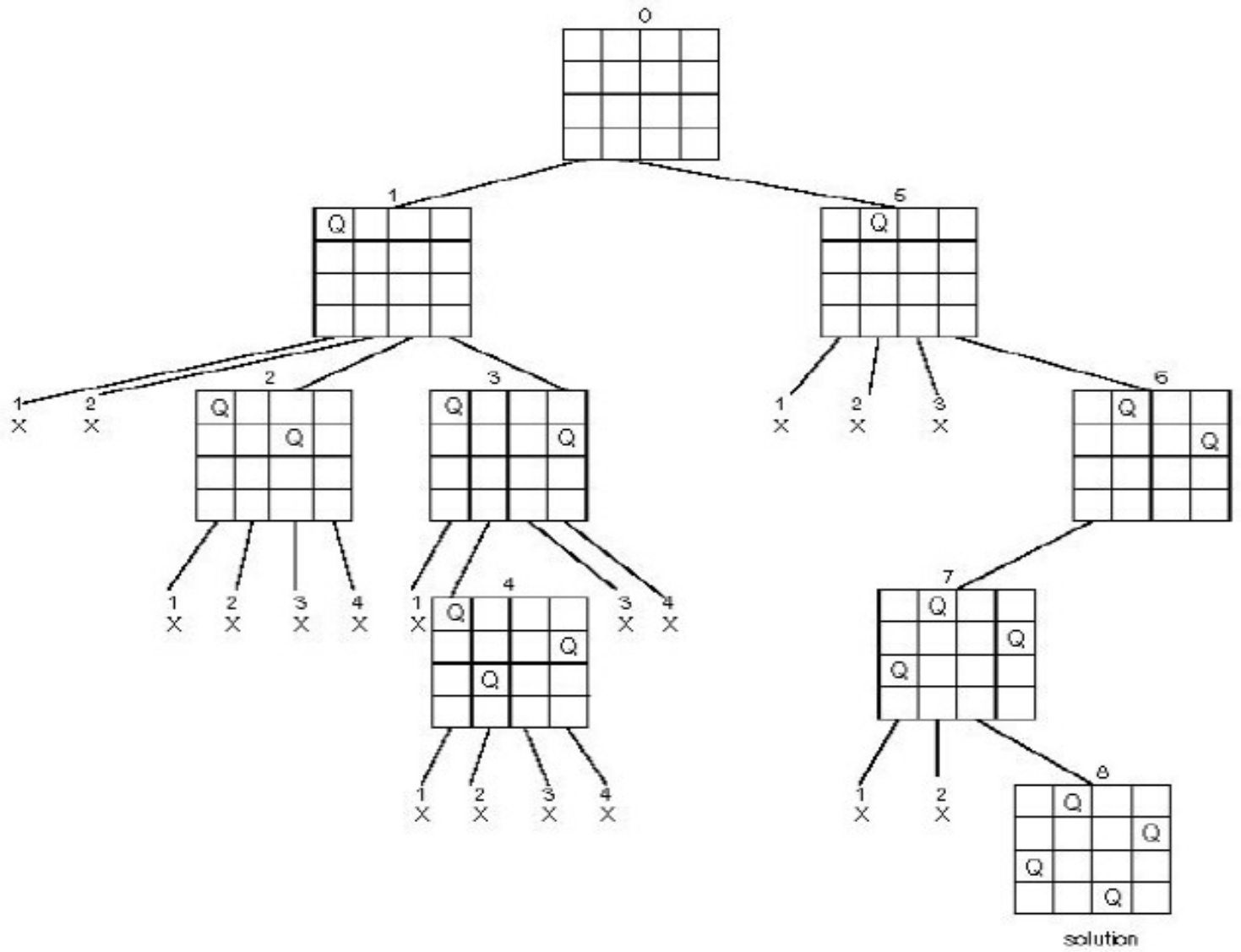
If all you have is a hammer, everything looks like a nail
- Abraham Maslow, 1962

Backtracking

- Técnicas de busca exaustiva geram todas as soluções candidatas e então identificam aquela (ou aquelas) com a propriedade desejada

- A ideia do Backtracking, ao contrário, é construir soluçoes parciais e avaliá-las da seguinte maneira:
  - Se a solução parcial pode ser continuada, sem violar os objetivos, então, faça-o, incorporando um próximo componente legítimo
  - Se nao há nenhuma opção legítima, nenhuma alternativa restante precisa ser considerada.
  - Backtracking usa conceito de árvore de estado-espaço.

Backtracking: 4 Queens!



Backtracking

- Estratégia interessante que pode ser usada para resolver problemas combinatoriais difícies.
- A árvore estado espaço são criadas no estilo DFS (depth first search)

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.

```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);
}
```

Template Backtracking

```
bool finished = FALSE;                  /* found all solutions yet? */

backtrack(int a[], int k, data input)
{
        int c[MAXCANDIDATES];           /* candidates for next position */
        int ncandidates;                /* next position candidate count */
        int i;                          /* counter */

        if (finished) return;   /* terminate early
        if (is_a_solution(a,k,input))
                process_solution(a,k,input);
        else {
                k = k+1;
                construct_candidates(a,k,input,c,&n
                for (i=0; i<ncandidates; i++) {
                        a[k] = c[i];
                        backtrack(a,k,input);
                }
        }
}
```

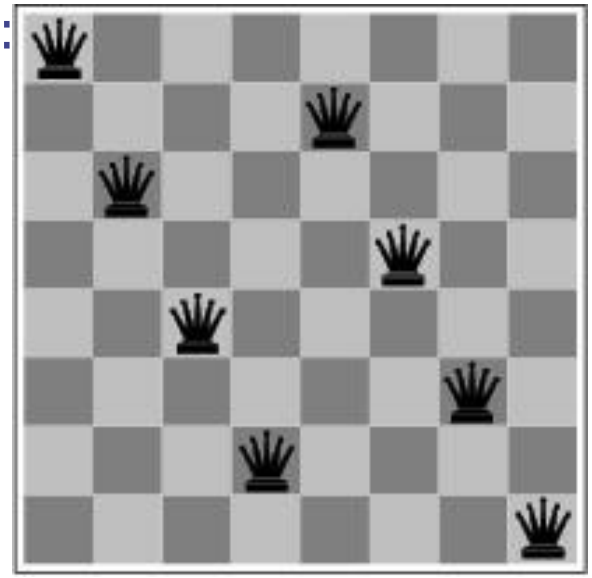
Verifica se é solução

Incrementa contagem e/ou imprime a solução e/ou força interrupção do algoritmo (“finished” = TRUE), etc

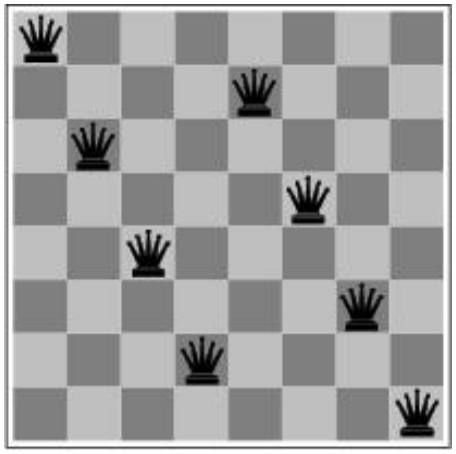
Preenche o vetor ***c*** com no. “*ncandidates*” de valores possíveis para *a[k]*, dados os *k*-1 valores anteriores, e retorna esse no. Note que não é tão eficiente em termos de memória, pois ***c*** é gerado de uma só vez e armazenado na pilha de recursão.

Exemplo – 8 Queens

- Problema:
  - 8 rainhas no tabuleiro
  - Nenhuma sob ataque
- Exemplo de Quase Solução:

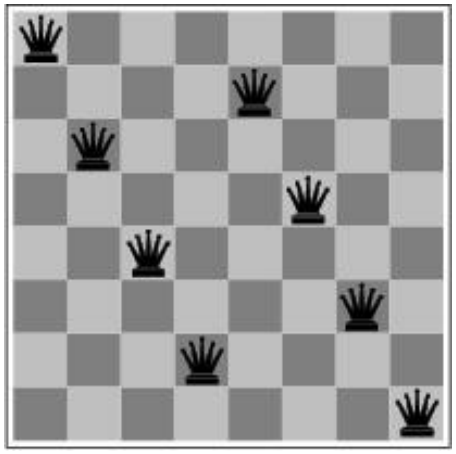


Exemplo – 8 Queens



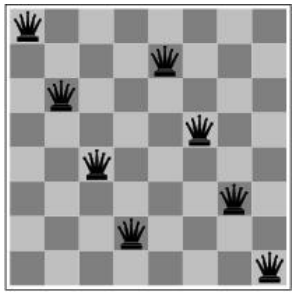
- Representação 1:
  - Vetor binário com $8^2 = 64$ elementos
  - Elemento é 1 caso a posição estiver sob ataque e 0 caso não estiver
  - $2^{64} \approx 1{,}85 \times 10^{19}$ possibilidades...
- Representação 2:
  - Vetor de inteiros com 8 elementos
  - Elemento assume valor $x \in \{1, ..., 64\}$, que indica a posição da rainha no tabuleiro
  - $64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57 \approx 1{,}78 \times 10^{14}$ possibilidades

Exemplo – 8 Queens



- Representação 3 (**Poda de Simetrias**):
  - Representação 2 com $a_1 > a_2 > ... > a_n$
  - Reduz o número de soluções em 8! vezes (permutações): $\approx 4{,}42 \times 10^9$
- Representação 4 (1 rainha em cada linha/coluna):
  - Vetor de inteiros com 8 elementos
  - Elemento assume valor $x \in \{1, ..., 8\}$, que indica posição na linha/col.
  - 8! = 40320 possibilidades!
  - Verificando para cada elemento a_k quais os valores candidatos que não colocam a rainha correspondente sob ataque das k-1 anteriores, reduz-se significativamente esse número para 2057 possíveis seqüências, das quais apenas 92 são soluções (pode-se obter isso experimentalmente).

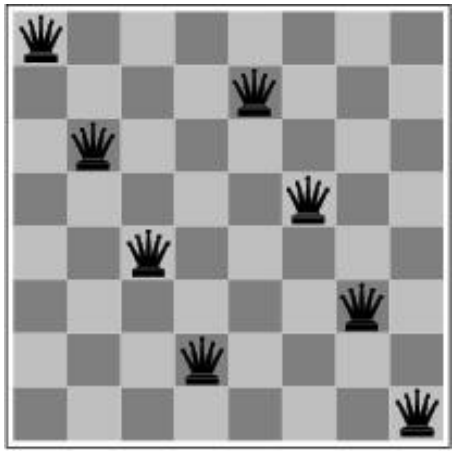
Exemplo – 8 Queens



```
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
    int i,j;              /* counters */
    bool legal_move;      /* might the move be legal? */

    *ncandidates = 0;
    for (i=1; i<=n; i++) {
        legal_move = TRUE;
        for (j=1; j<k; j++) {
            if (abs((k)-j) == abs(i-a[j]))   /* diagonal threat */
                legal_move = FALSE;
            if (i == a[j])                   /* column threat */
                legal_move = FALSE;
        }
        if (legal_move == TRUE) {
            c[*ncandidates] = i;
            *ncandidates = *ncandidates + 1;
        }
    }
}
```

tornasse FALSE.

Exemplo – 8 Queens

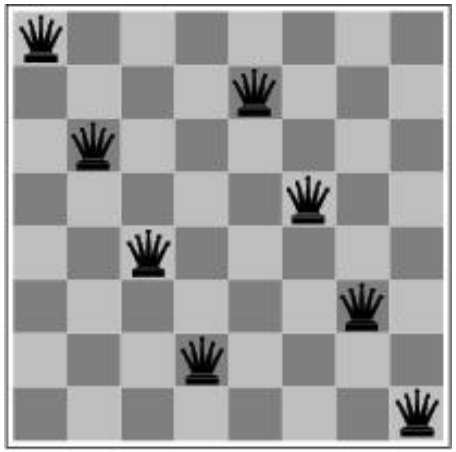


```
is_a_solution(int a[], int k, int n)
{
    return (k == n);
}
```

```
int solution_count; /* how many solutions are there? */

process_solution(int a[], int k)
{
    solution_count ++;
}
```

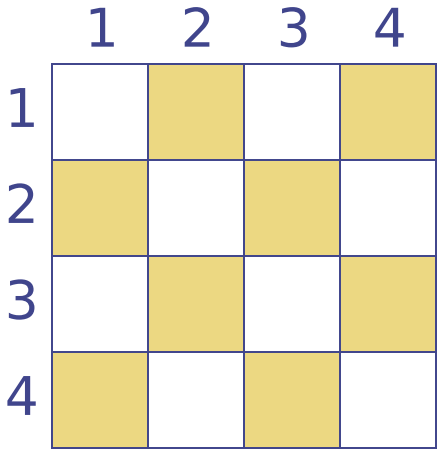
```
#define NMAX 8
#define MAXCANDIDATES 8

main() {
    int a[NMAX+1];          /* solution vector     */
                            /* first cell ignored */
    backtrack(a,0,NMAX);
    printf("Solution_count=%d\n",solution_count);
}
```

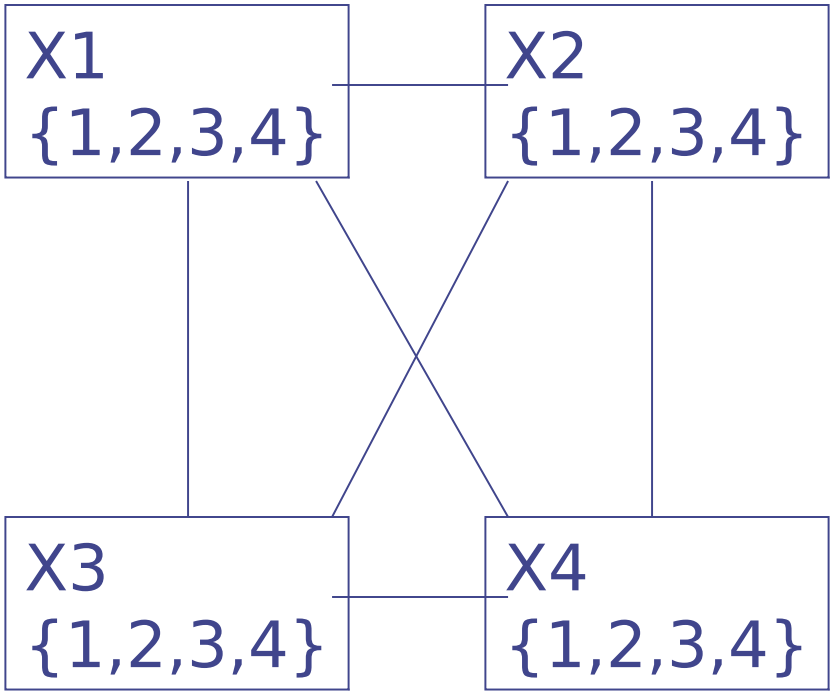
Forward Checking



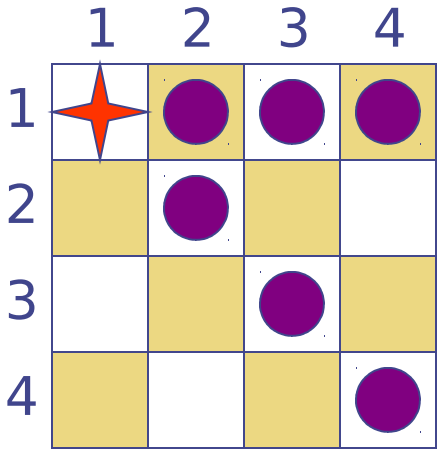
- É possível antecipar fracassos inevitáveis dado o estado corrente de atribuições ?
- ***Forward Checking* (FC)**: mantém controle dos valores remanescentes consistentes para cada variável ainda não atribuída.
- Antecipa o retorno (*backtrack*) quando alguma dessas variáveis se torna infactível, ou seja, fica com um domínio nulo de valores legais.

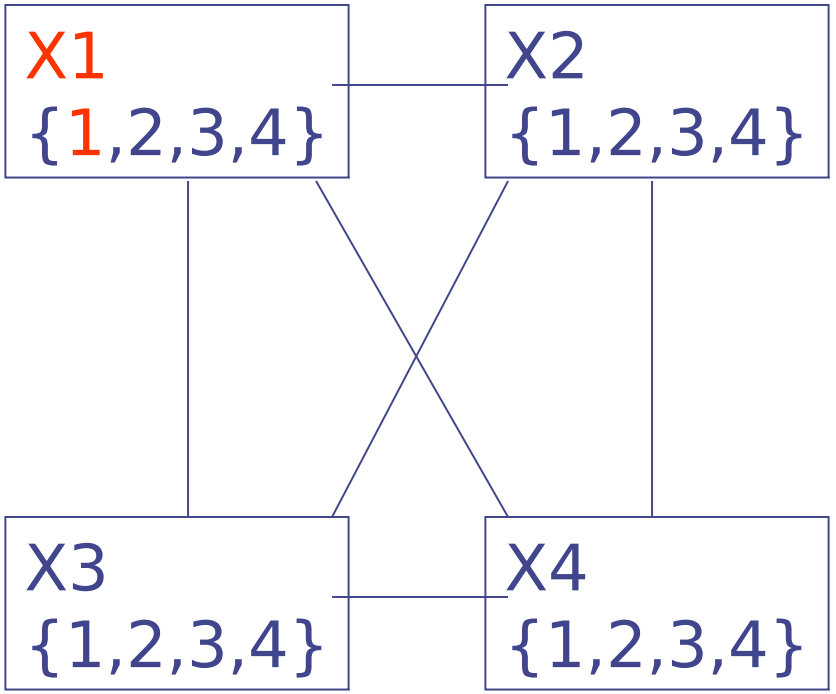
Exemplo: Problema 4 Queens



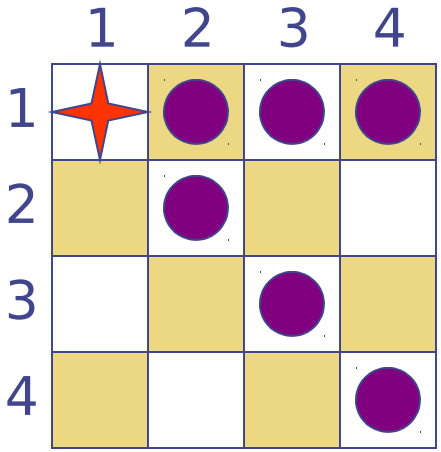


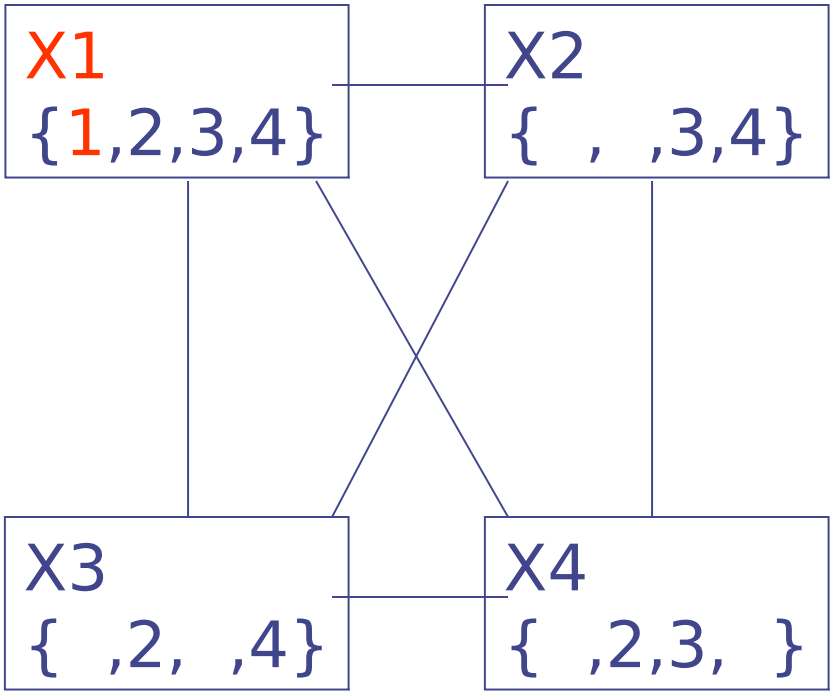
Exemplo: Problema 4 Queens



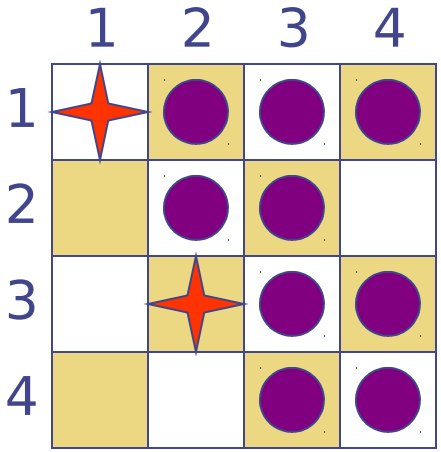


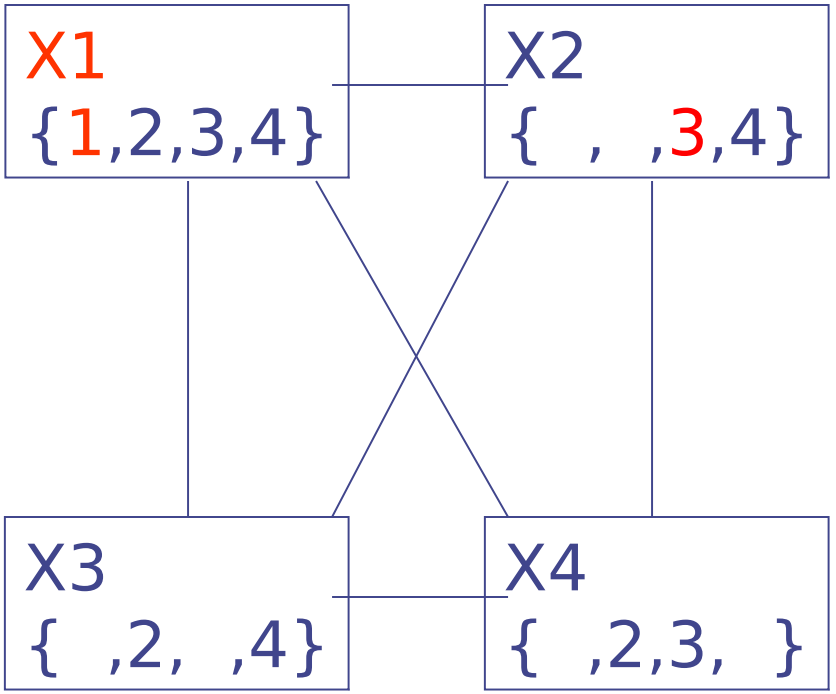
Exemplo: Problema 4 Queens



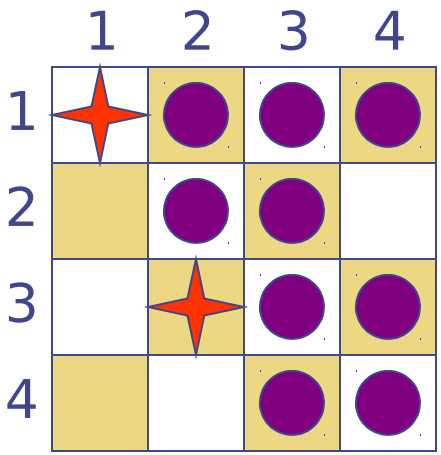


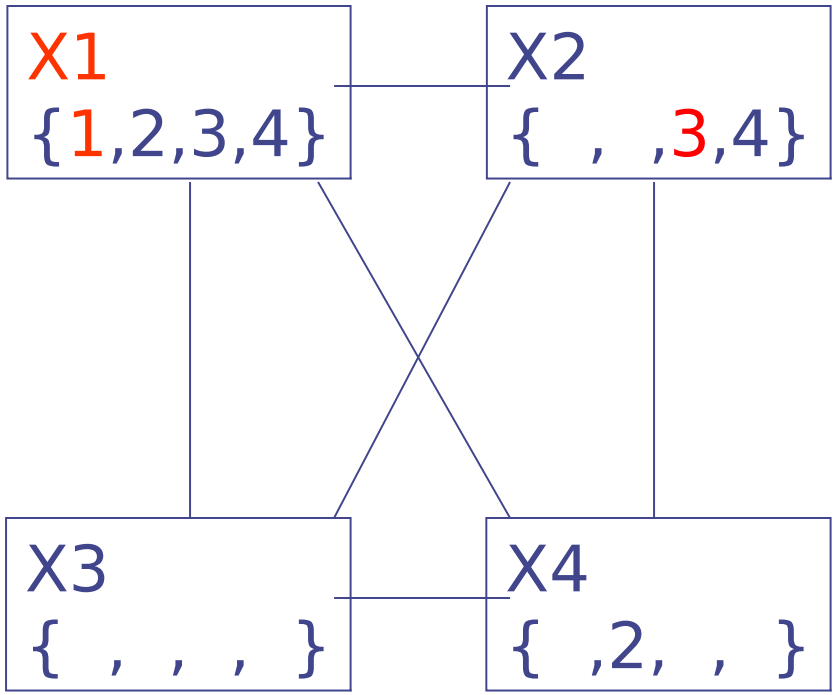
Exemplo: Problema 4 Queens





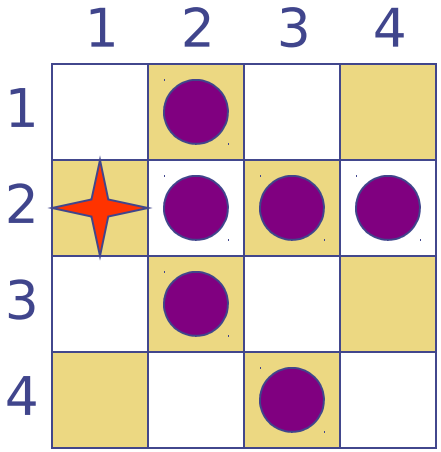
Exemplo: Problema 4 Queens

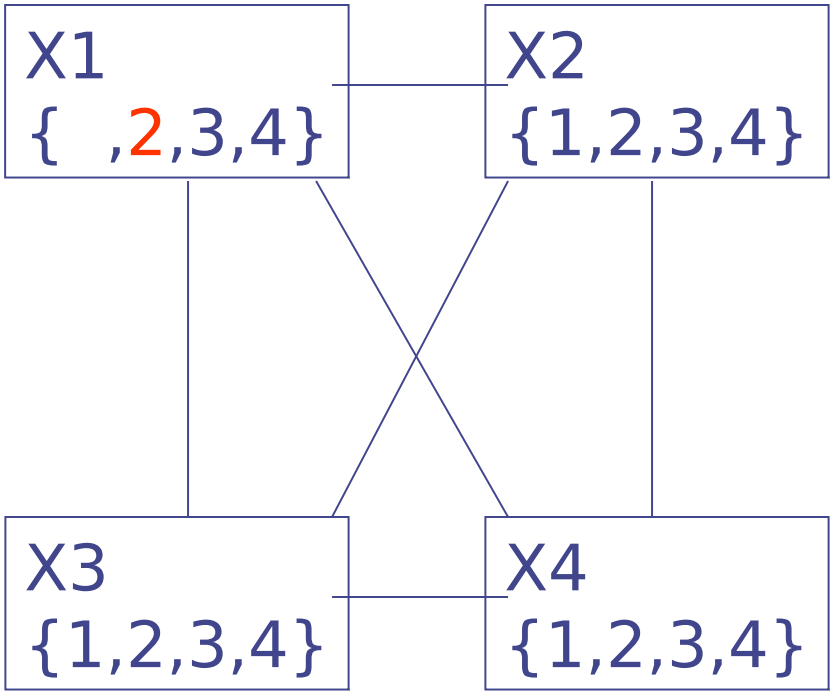




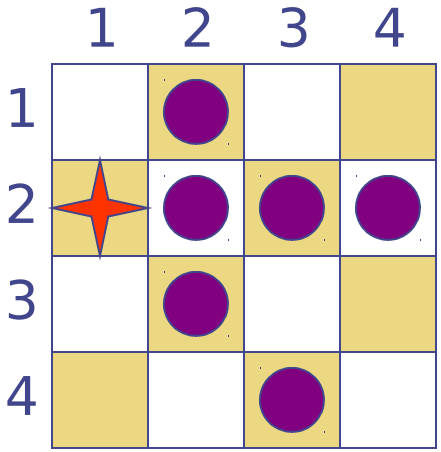
Nesse ponto FC detecta a inconsistência em X3 e antecipa o *backtrack*, indo imediatamente para o próximo valor de X2.

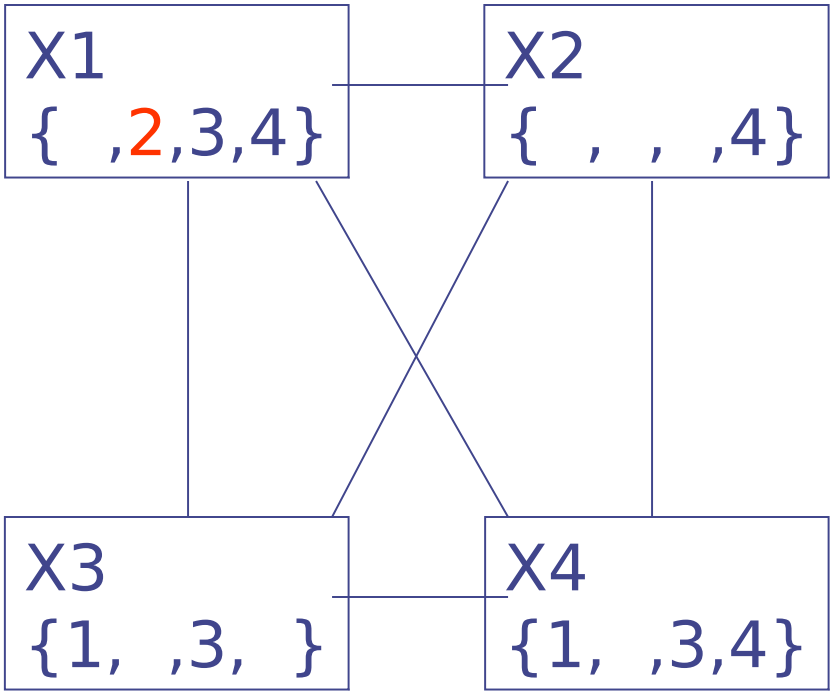
Exemplo: Problema 4 Queens



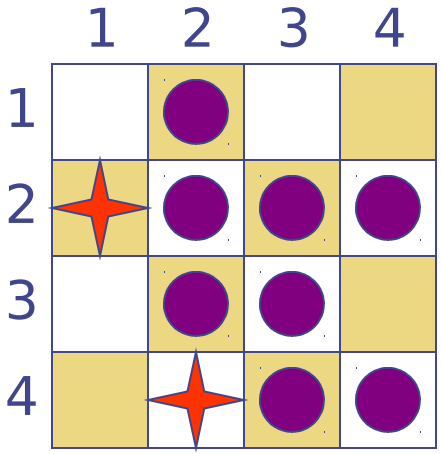


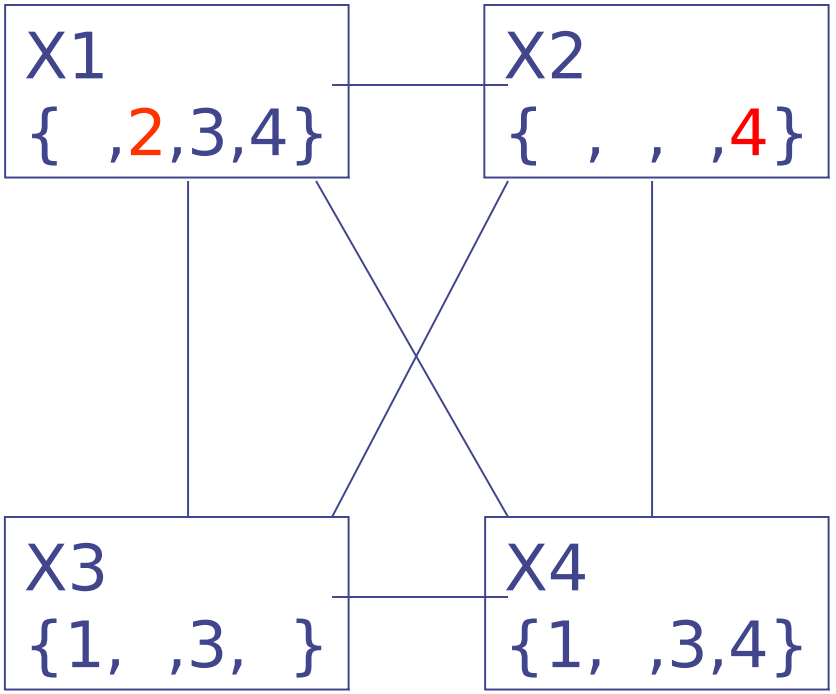
Exemplo: Problema 4 Queens



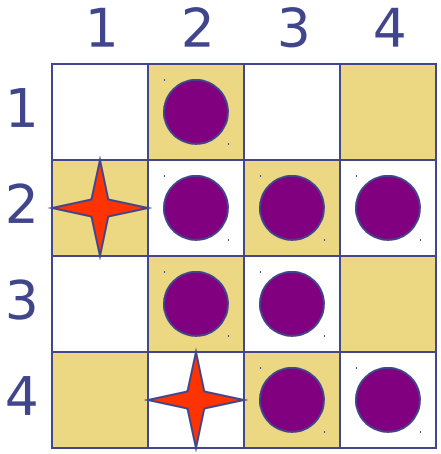


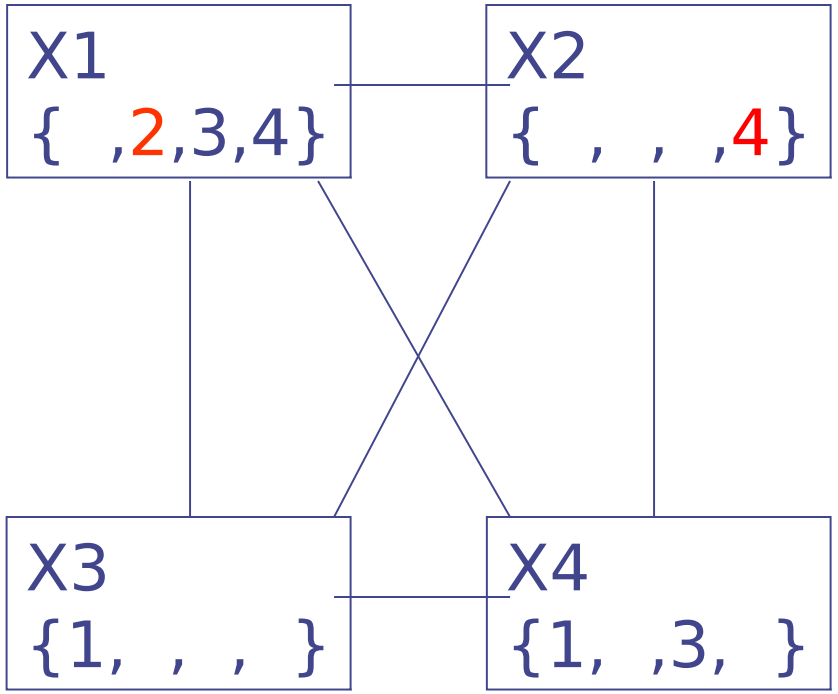
Exemplo: Problema 4 Queens



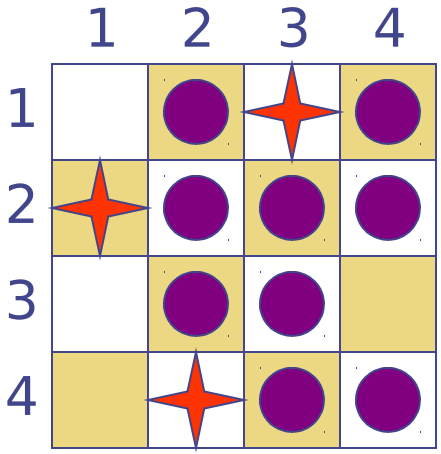


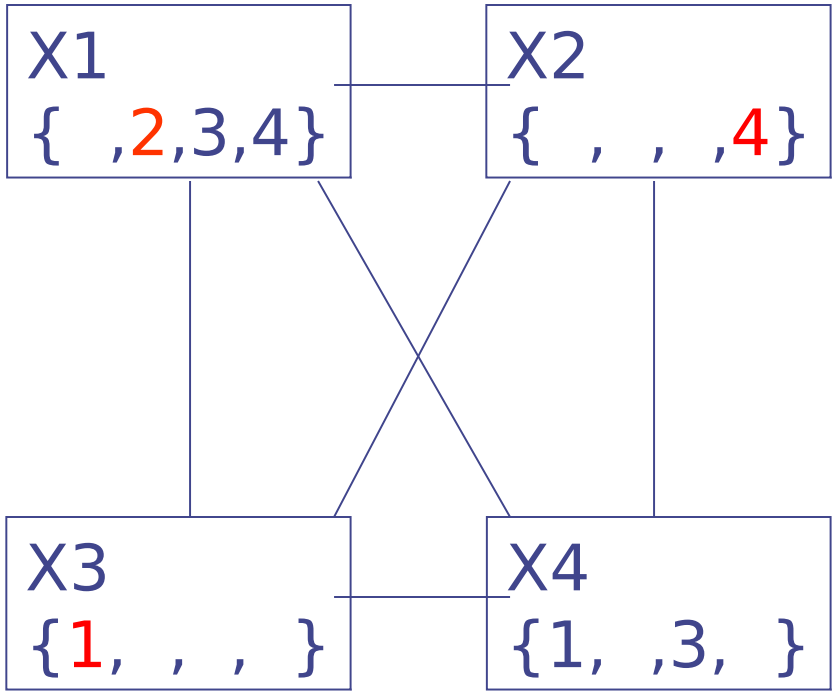
Exemplo: Problema 4 Queens



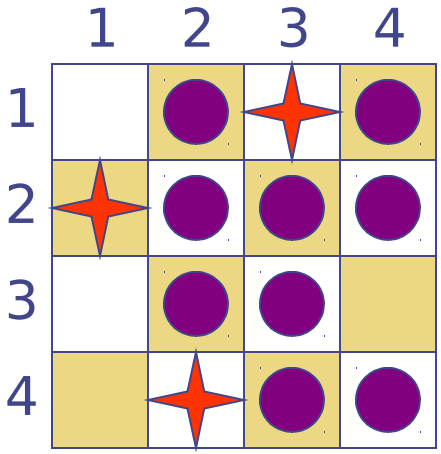


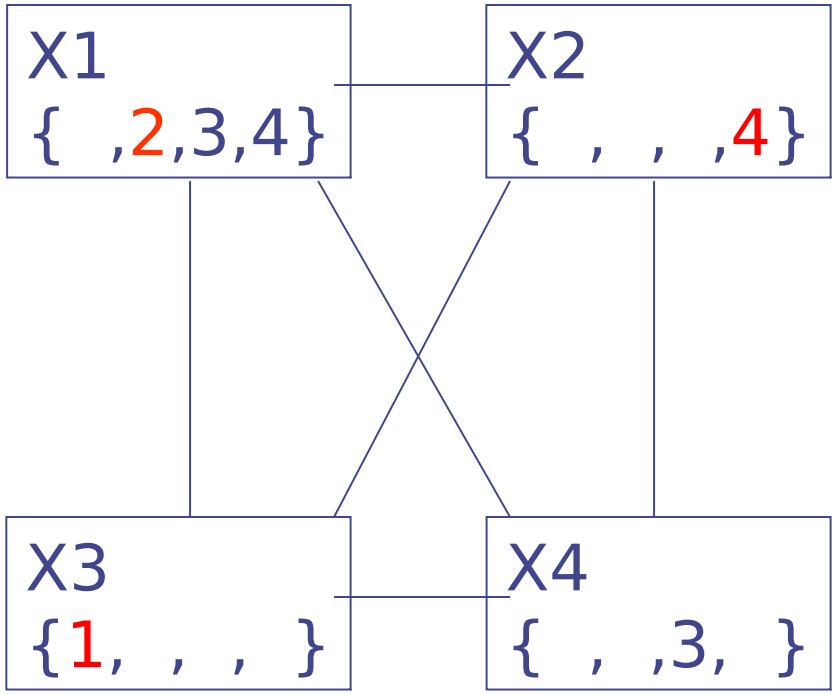
Exemplo: Problema 4 Queens



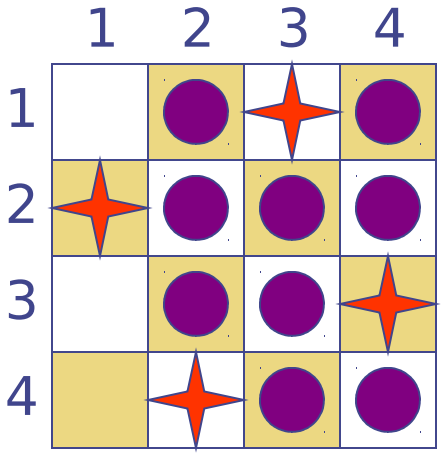


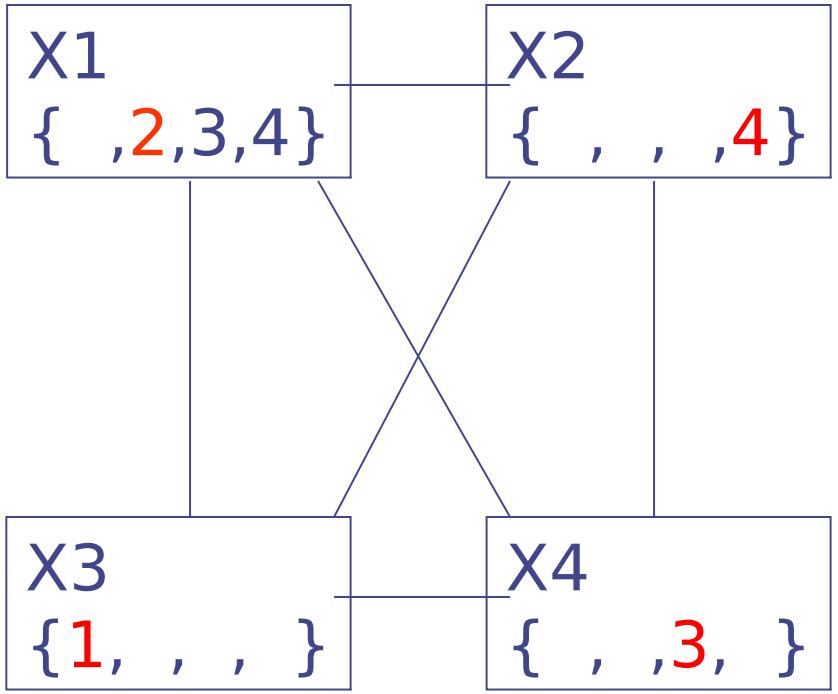
Exemplo: Problema 4 Queens



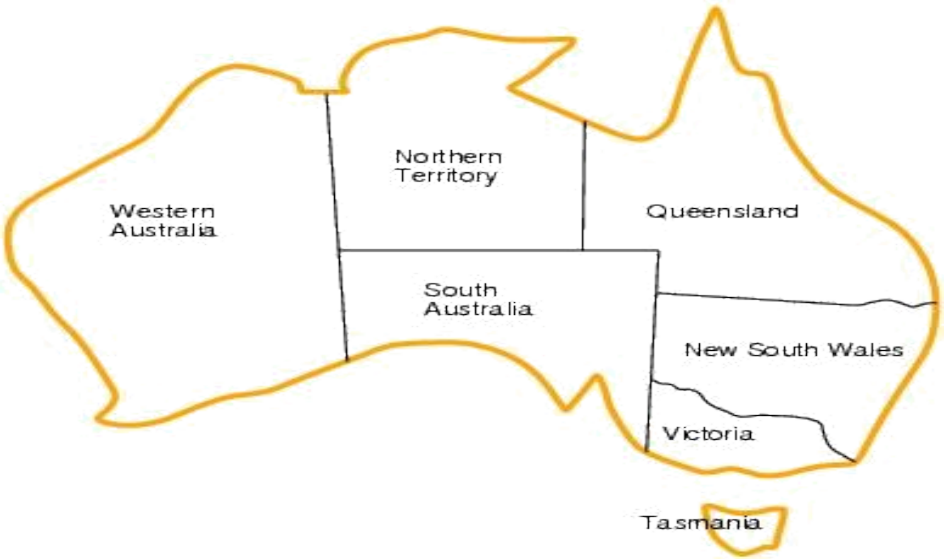


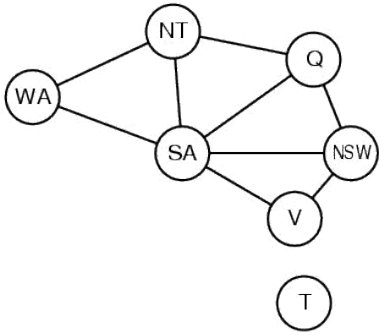
Exemplo: Problema 4 Queens





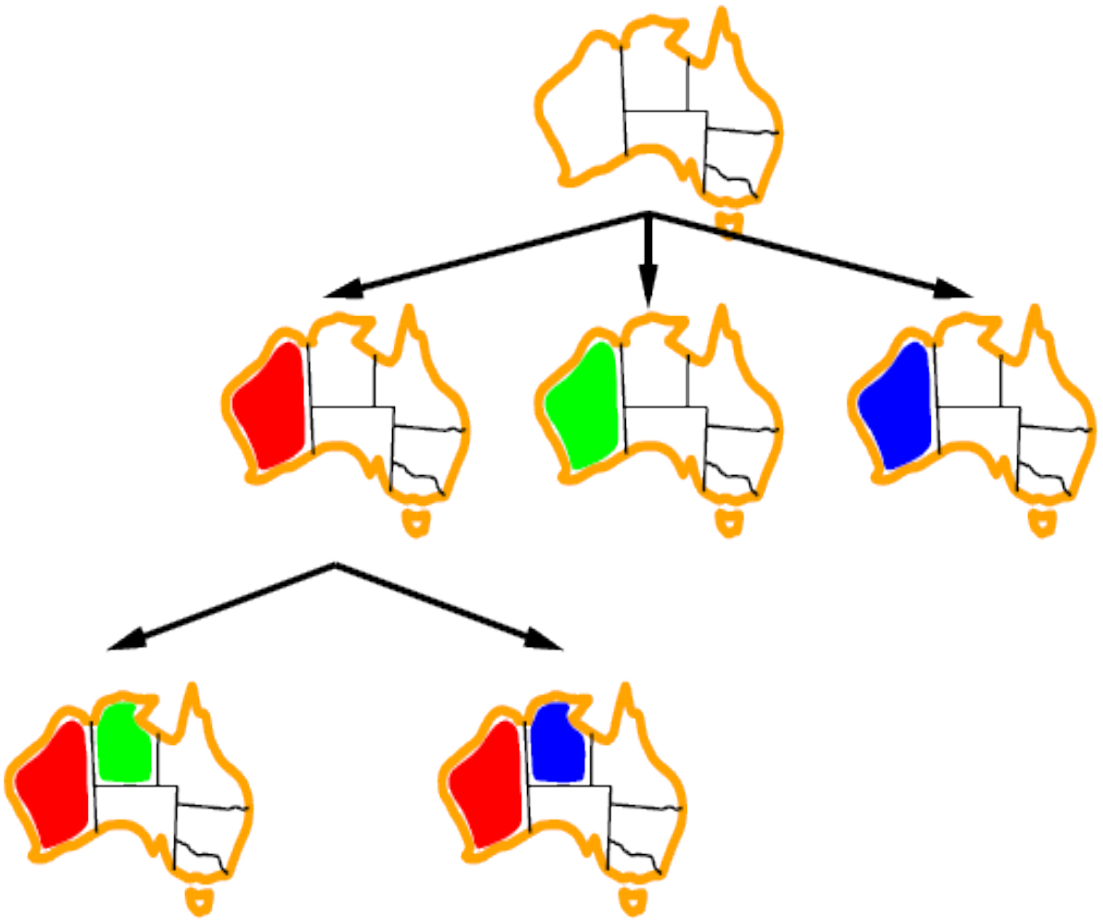
Exemplo – Coloração de Grafos



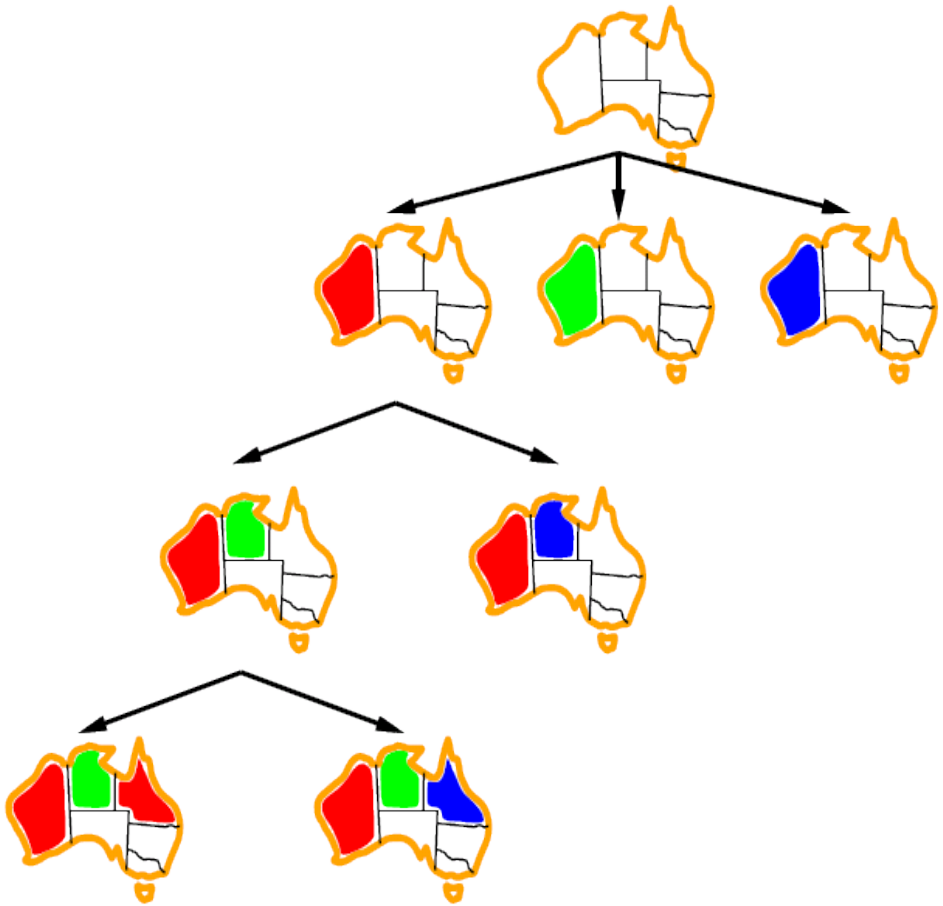


Exemplo – Coloração de Grafos

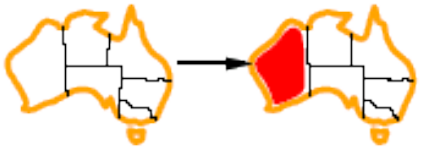
Exemplo – Coloração de Grafos

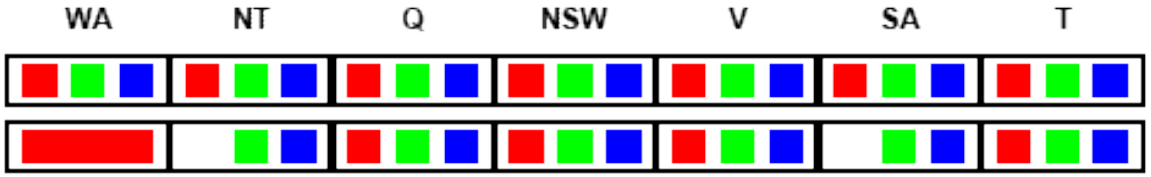


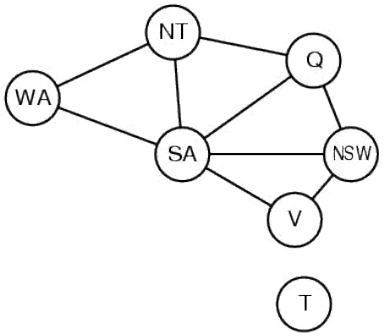
Exemplo – Coloração de Grafos



Forward Checking

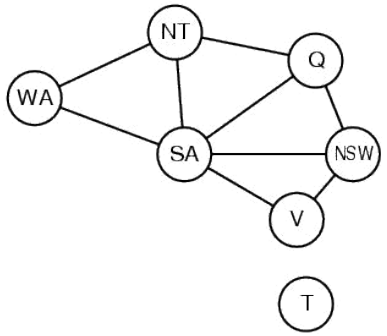






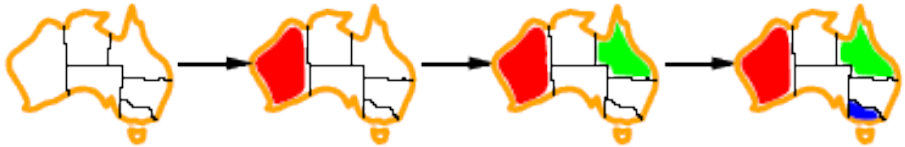
- Atribuição: *{WA = red}*
- Efeito sobre as outras variáveis conectadas por restrições:
  - *NT não pode mais ser red*
  - *SA não pode mais ser red*

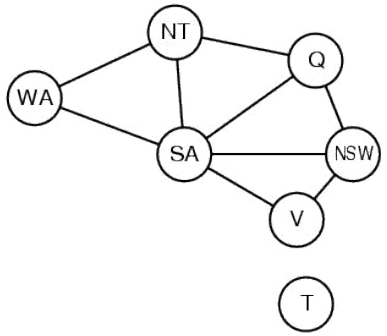
Forward Checking



- Atribuição: *{Q = green}*
- Efeito sobre as outras variáveis conectadas por restrições:
  - *NT não pode mais ser green*
  - *NSW não pode mais ser green*
  - *SA não pode mais ser green*

Forward Checking





- Se *V* é atribuído *blue*
- Efeito sobre as outras variáveis conectadas por restrições:
  - *NSW não pode mais ser blue.*
  - ***Domínio legal de SA é vazio !***
- FC detectou que a atribuição parcial descrita é *inconsistente* com as restrições do problema e *backtrack* será antecipado.

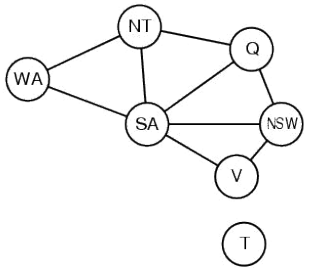
Heurísticas para Backtracking

- O uso de heurísticas em algoritmos de backtracking podem acelerar significativamente o processo de busca, especialmente quando combinadas com mecanismos antecipativos, como FC.
- Duas heurísticas de propósito geral referem-se às seguintes questões:
  - Qual a próxima variável a ser atribuída ?
  - Em qual ordem os valores candidatos devem ser tentados ?

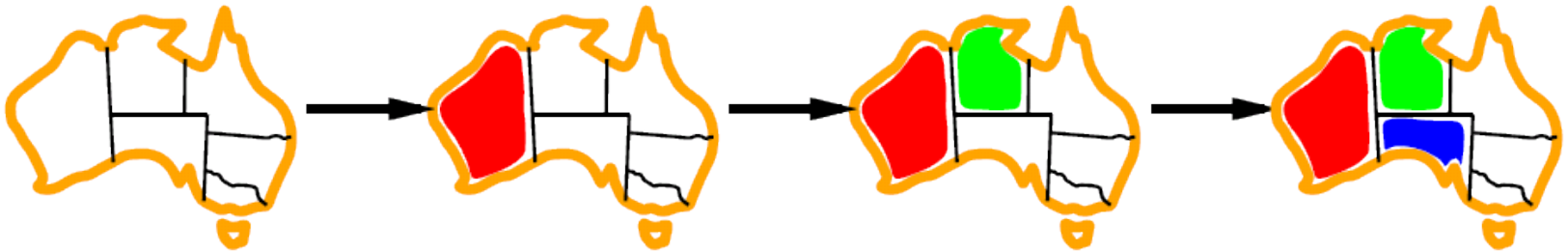
Qual a Próxima Variável?

- Dado um conjunto parcial de atribuições, a escolha da próxima variável a ser atribuída deve ser no sentido de direcionar ao máximo a busca a caminhos com potencial de solução, evitando longos caminhos infrutíferos e retornos desnecessários pela árvore de busca.
- **Fato:** qualquer variável terá necessariamente que ser atribuída em algum momento.
- **Conclusão:** deve-se priorizar variáveis mais críticas com relação a restrições, pois essas são as potenciais causadoras de infactibilidades e *backtracks* na busca.

Heurística MRV

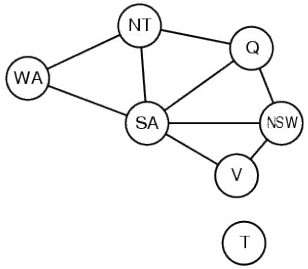


MRV = *Minimum Remaining Values*



- *Regra*: Escolher a variável com o menor número de valores legais a serem atribuídos dadas as atribuições de variáveis anteriores.
- *Idéia:* Selecionar a variável mais restrita, evitando uma provável perda de tempo com a atribuição de outras variáveis que acabariam a tornando infactível e forçando um *backtrack*.

Heurística *Degree*

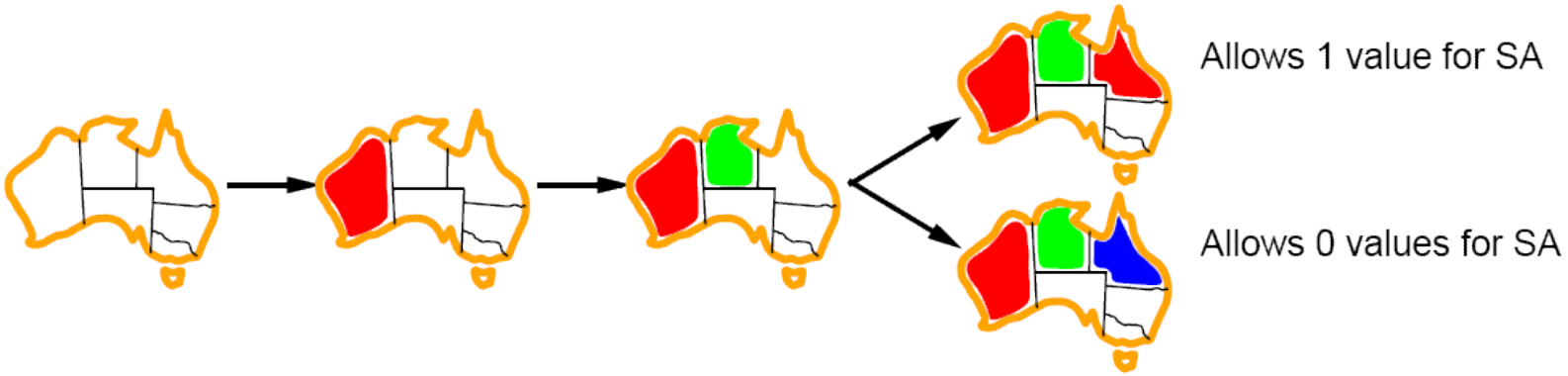


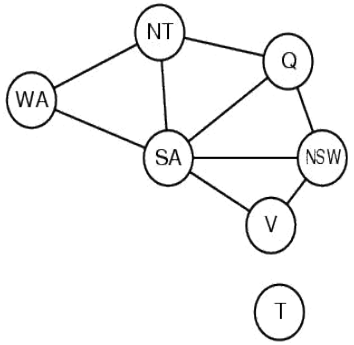
- *Regra*: escolher a variável que está envolvida em um maior número de restrições junto a outras variáveis ainda não atribuídas.
- *Idéia:*
  - Selecionar a variável que possui o maior potencial de se tornar infactível após a atribuição das demais.
  - Essa variável também é aquela que mais irá restringir as demais após sua atribuição, possivelmente antecipando um retrocesso na árvore por detecção de infactibilidade via FC.
- *Aplicação:* Embora menos eficiente que MRV, é usualmente utilizada para decidir empates nessa última (e.g. no caso da primeira variável).

Qual o Próximo Valor ?

- Dado um conjunto parcial de atribuições, a seleção do próximo valor a ser atribuído a uma dada variável deve ser no sentido de direcionar a busca a caminhos com potencial de solução, evitando caminhos infrutíferos e retornos desnecessários pela árvore.
- **Fato:** não necessariamente um único valor pode ser atribuído a uma determinada variável sem que isso implique a inexistência de uma solução.
- **Conclusão:** deve-se priorizar valores menos críticos com relação às demais variáveis ainda não atribuídas.

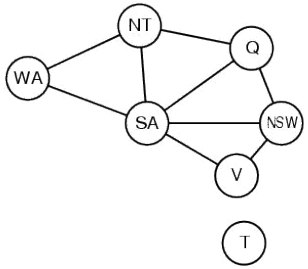
Heurística do Valor Menos Restritivo

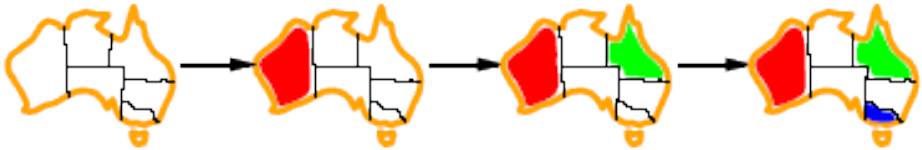


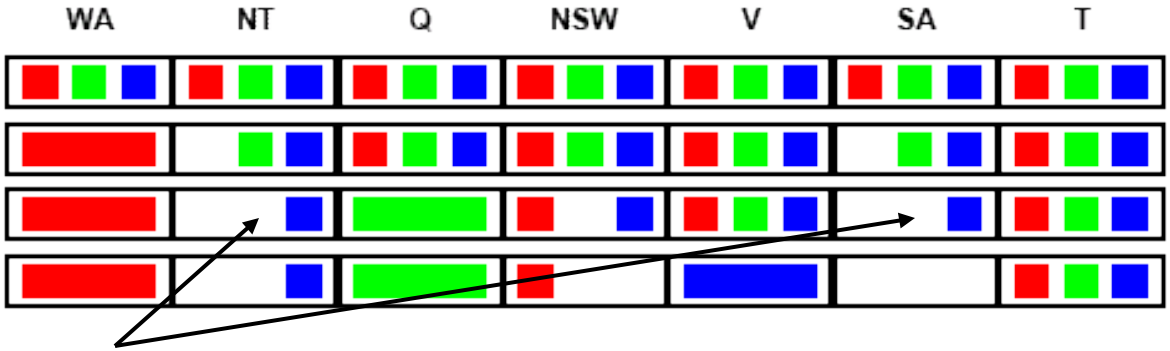


- *Regra*: dada uma variável, escolha os seus valores a partir do menos restritivo, i.e. daquele que deixa o máximo de flexibilidade para as atribuições de variáveis subseqüentes.

Tópicos Avançados





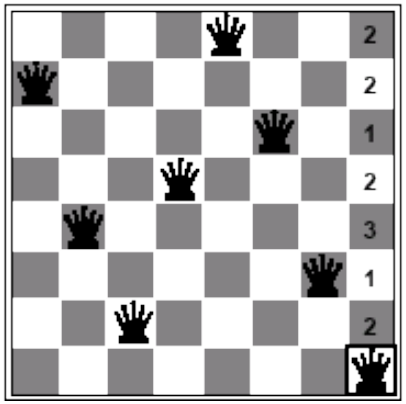


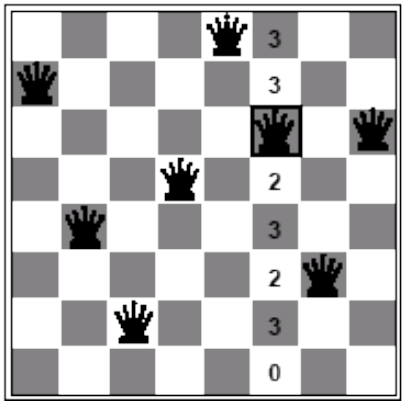
- O tipo de *propagação de restrições* implementado por FC não é capaz de detectar antecipadamente todas as possíveis inconsistências:
  - e.g. NT e SA já não podiam ser azuis antes da atribuição de V !
- Abordagens de propagação de restrições mais sofisticadas:
  - **Consistência de Arcos**
  - ***k*-Consistência**
  - **Consistência de Restrições Especiais**

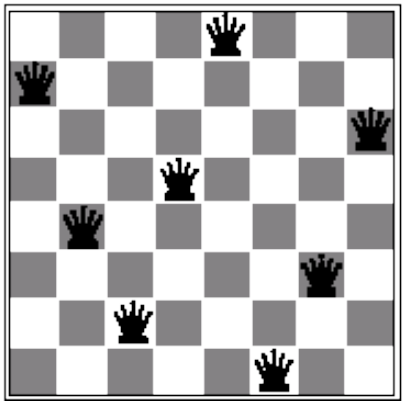
Tópicos Relacionados

- Métodos Heurísticos de Busca Local:
  - Hill Climbing;
  - Busca Tabu;
  - Conflitos Mínimos:
    - Seleciona um novo valor para uma dada variável (e.g. escolhida aleatoriamente) que resulte em um menor número de conflitos com as demais variáveis.
- Métodos Heurísticos de Busca Global:
  - Algoritmos Evolutivos (e.g. GAs);
  - Algoritmos de Enxame (e.g. PSO, ACO, etc).

Tópicos Relacionados







- Uma solução de dois passos para o problema das 8 rainhas utilizando a heurística dos conflitos mínimos.
- Em cada passo uma rainha é re-posicionada em sua coluna.
- O algoritmo move a rainha para o quadrado de conflito mínimo, resolvendo empates aleatoriamente.
- Heurística bastante insensível ao tamanho *n* no problema mais geral das *n*-rainhas: Resolve para *n* = 1 milhão em média em 50 passos !!!

Algoritmo: conflitos mínimos

function MIN-CONFLICTS(*csp, max_steps*) return solution or failure

inputs: *csp*, a constraint satisfaction problem

max_steps, the number of steps allowed before giving up

current ← an initial complete assignment for *csp*

for *i* = 1 to *max_steps* do

if *current* is a solution for *csp* then return *current*

var ← a randomly chosen, conflicted variable from VARIABLES[*csp*]

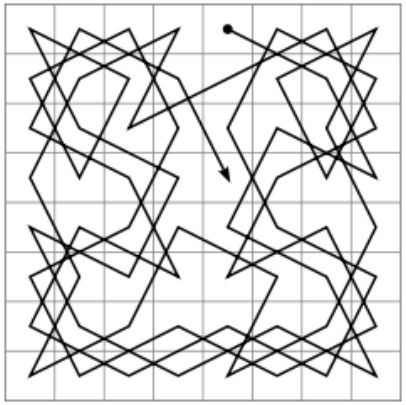
value ← the value *v* for *var* that minimize CONFLICTS(*var,v,current,csp*)

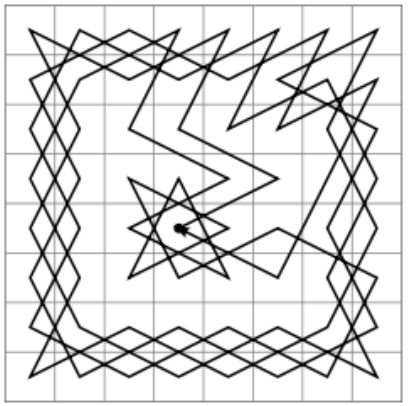
set *var* = *value* in *current*

return *failure*

Percurso do Cavalo Revisitado

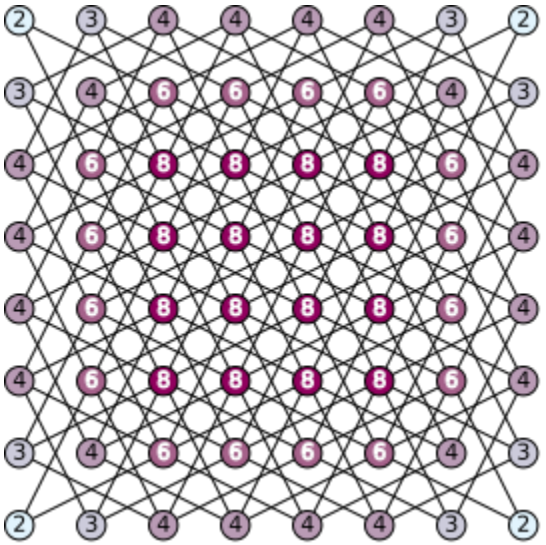
- O problema do percurso do cavalo envolve encontrar um caminho Hamiltoniano (similar ao TSP) e é NP-Completo, portanto exponencial.
- Entretanto, as regularidades existentes permitem encontrar algoritmos lineares.





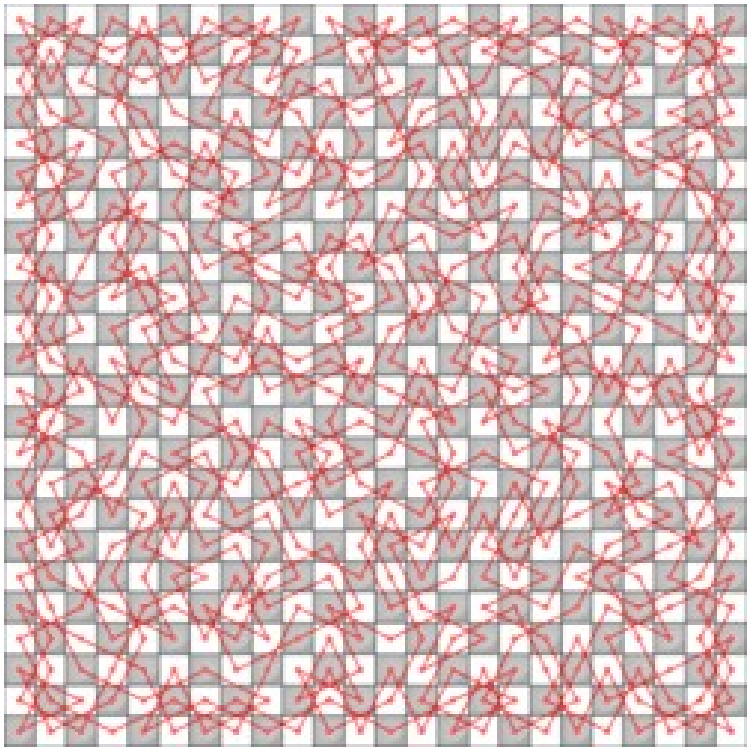
Percurso do Cavalo Revisitado

- O algoritmo de Warnsdorff (1823) funciona bem para tabuleiros até 76x76 e privilegia posições com pouco sucessores (isolados).



Percurso do Cavalo Revisitado

- E outros algoritmos linerares (Conrad *et al.*, 1994) e baseados em redes neurais (Takefuji & Lee, 1992) podem solucionar grandes instâncias.



Bibliografia

- A. Conrad, T. Hindrichs, H. Morsy, and I. Wegener. "Solution of the Knight's Hamiltonian Path Problem on Chessboards." *Discrete Applied Math*, volume 50, no.2, pp.125-134. 1994.
- Y. Takefuji, K. C. Lee. "Neural network computing for knight's tour problems." *Neurocomputing*, 4(5):249-254, 1992.
- H. C. Warnsdorff von *“Des Rösselsprungs einfachste und allgemeinste Lösung.“* Schmalkalden, 1823.
- Wikipedia. “Knight's tour.” http://en.wikipedia.org/wiki/Knight's_tour.
- Wolfram Math World. “Knight's Tour”. http://mathworld.wolfram.com/KnightsTour.html