SCC-210
Algoritmos Avançados

Capítulo 7
Backtracking

Adaptado por João Luís G. Rosa

Backtracking

- *Backtracking* é uma técnica exaustiva de busca por soluções para problemas combinatórios.
- Problemas combinatórios podem ser definidos como aqueles para os quais existe uma representação tal que as soluções são dadas por uma combinação de valores de um conjunto de variáveis discretas:
  - Exemplo: Encontre os valores lógicos de um conjunto de variáveis que satisfaçam (tornem verdadeira) uma dada forma normal conjuntiva (expressão lógica constituída de conjunções de disjunções das variáveis ou suas negações) (SAT).

Backtracking

- Para muitos problemas dessa natureza, existem algoritmos eficientes (polinomiais). Exemplos:
  - Caminhos mais curtos em grafos,
  - Árvore geradora mínima.
- Para outros, no entanto, não existe algoritmo eficiente conhecido (e.g. SAT e TSP – problemas NP-completos), e não restam muitas outras alternativas senão:
  - Métodos de Busca Exaustiva
    - complexidade exponencial
  - Métodos Heurísticos
    - em geral não garantem solução ótima.

Backtracking

- A evolução dos computadores modernos tem tornado viável a solução por busca exaustiva (força bruta) de instâncias maiores de problemas combinatórios difíceis:
  - Exemplo: *Deep Blue* e sucessores.
- Por exemplo:
  - Computador pessoal de 1GHz $\rightarrow$ 1 bilhão de operações / segundo.
  - Estimando que verificar uma solução demanda algumas centenas de instruções, então pode-se verificar alguns milhões de soluções em 1s.
- No entanto...
  - 1 milhão de permutações equivalem a aproximadamente todos os possíveis arranjos de não mais que 10 objetos (10! = 3.628.800).
  - 1 milhão de subconjuntos equivalem à todas as possíveis combinações de não mais do que 20 itens (2^{20} = 1.048.576).

Exemplo 1: Números

- Vamos supor que queiramos gerar todos os possíveis números de quatro dígitos utilizando um vetor.
- Cada posição do vetor irá armazenar um dígito.
- Tal problema simples pode ser utilizado para ilustrar a construção de um algoritmo de *backtracking*.

Exemplo 1: Números

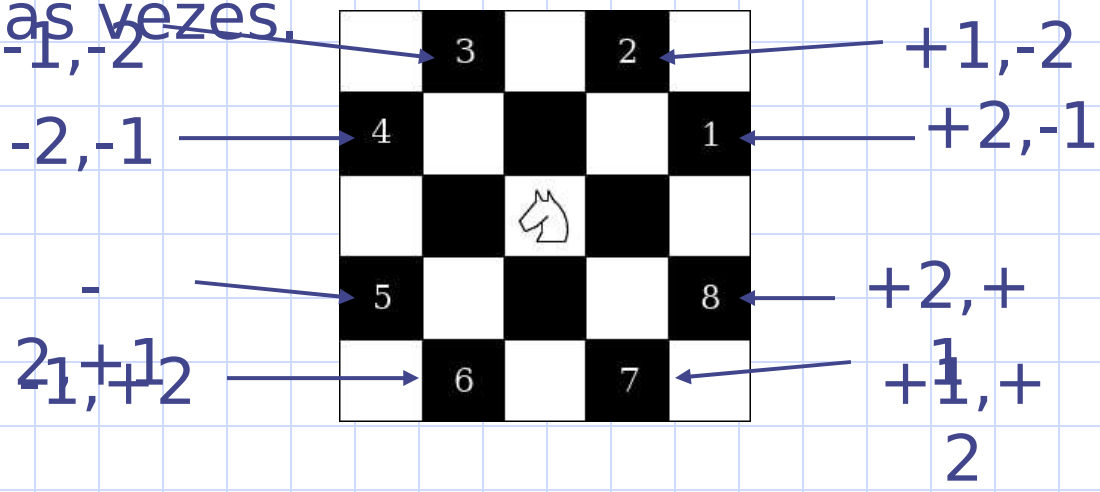
- Todo algoritmo de *backtracking* possui algumas características em comum:
  - Uma condição que verifica se uma solução foi encontrada;
  - Um laço que tenta todos os valores possíveis para uma única variável discreta;
  - Uma recursão que irá investigar se existe uma solução dados os valores atribuídos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

Exemplo 1: Números

```c
#include<stdio.h>
#define MAX 4

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

  if (k == MAX)
    print(v);          // Imprime solução
  else
    for (i = 0; i <= 9; i++) {
      v[k] = i;
      backtracking(v, k+1);
    }
}

int main() {
  int v[MAX];

  backtracking(v, 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

1	60	39	34	31	18	9	64
38	35	32	61	10	63	30	17
59	2	37	40	33	28	19	8
36	49	42	27	62	11	16	29
43	58	3	50	41	24	7	20
48	51	46	55	26	21	12	15
57	44	53	4	23	14	25	6
52	47	56	45	54	5	22	13

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,&
                for (i=0; i<ncandidates; i++) {
                        a[k] = c[i];
                        backtrack(a,k,input);
                }
        }
}
```

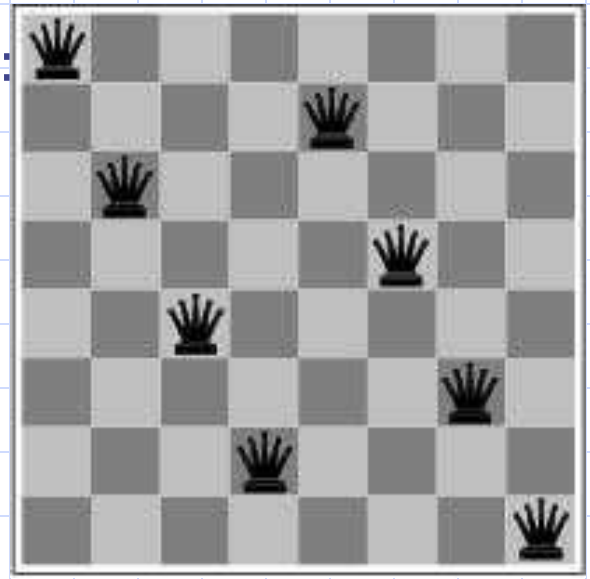
Verifica se é solução

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.

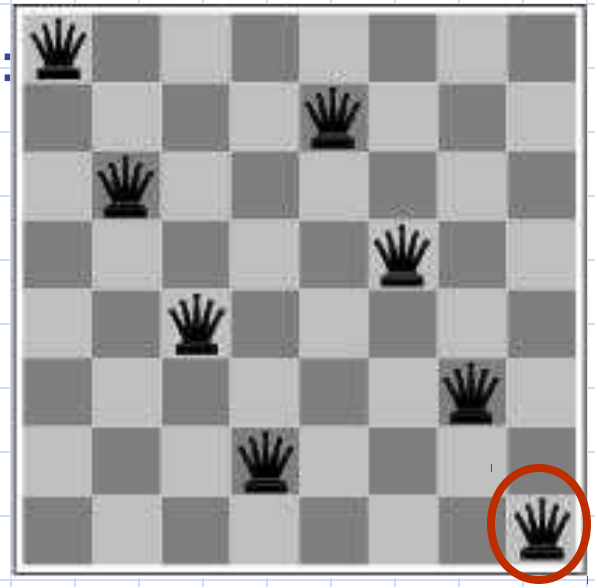
Exemplo: 8 Rainhas

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



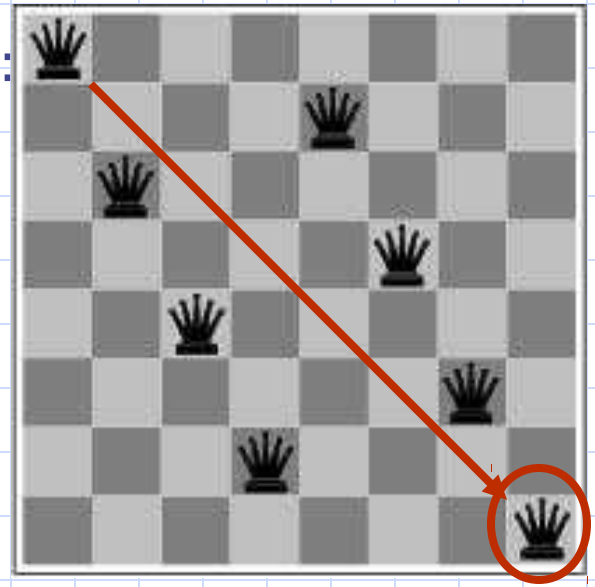
Exemplo: 8 Rainhas

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

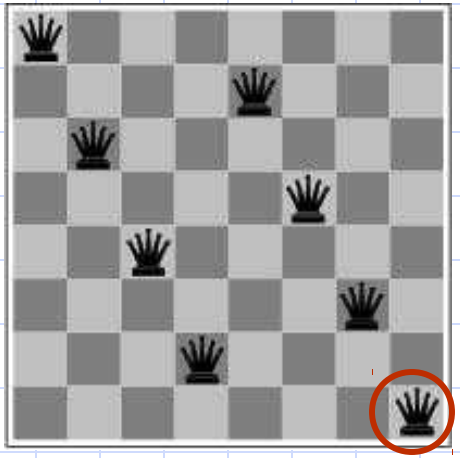


Exemplo: 8 Rainhas

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

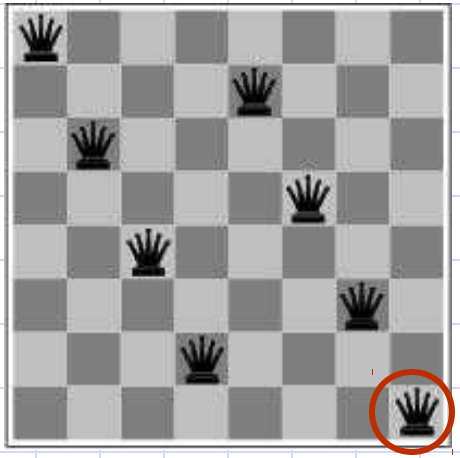


Exemplo: 8 Rainhas



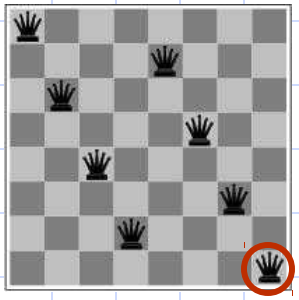
- Representação 1:
  - Vetor binário com $8^2 = 64$ elementos
  - Elemento é 1 caso a posição estiver sob ataque e 0 caso não estiver
  - $2^{64} \approx 1{,}85 \times 10^{19}$ possibilidades...
- Representação 2:
  - Vetor de inteiros com 8 elementos
  - Elemento assume valor $x \in \{1, ..., 64\}$, que indica a posição da dama 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 Rainhas



- Representação 3 (**Poda de Simetrias**):
  - Representação 2 com $a_1 > a_2 > \ldots > a_n$
  - 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 $x \in \{1, \ldots, 8\}$, 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).

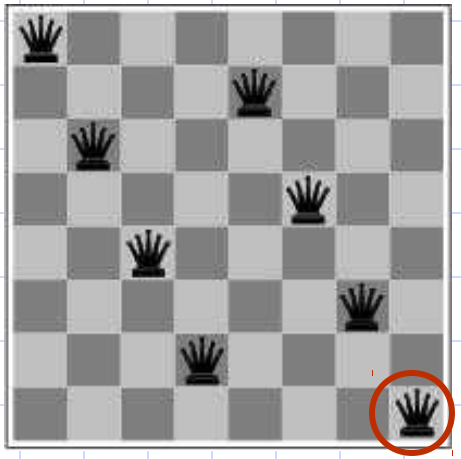
Exemplo: 8 Rainhas



```
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;
        }
    }
}
```

PS. Código poderia ser um pouco mais rápido se o loop interno fosse interrompido asssim que legal_move se tornasse FALSE.

Exemplo: 8 Rainhas

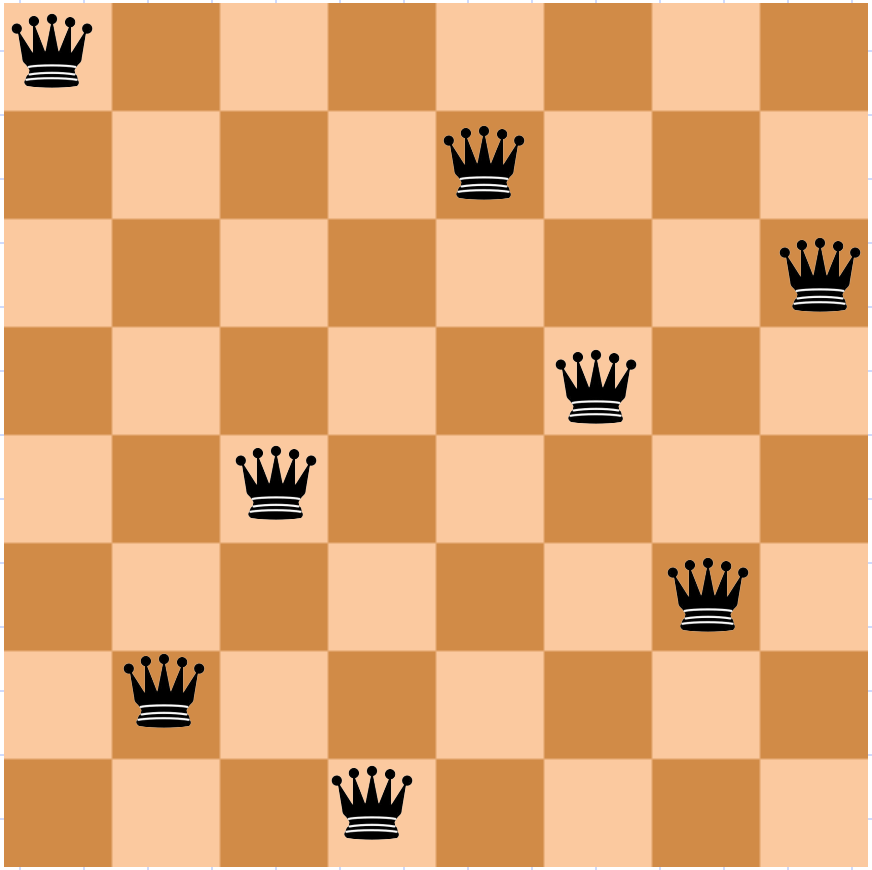


```
is_a_solution(int a[], int k, int n)
{
    return (k == n);
}
```

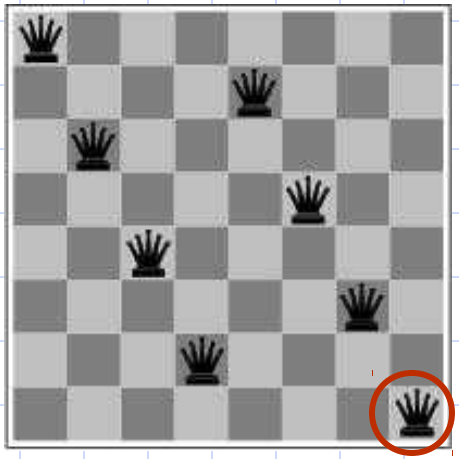
```
int solution_count; /* how many solutions are there? */

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);
}
```

8 Rainhas: Uma solução

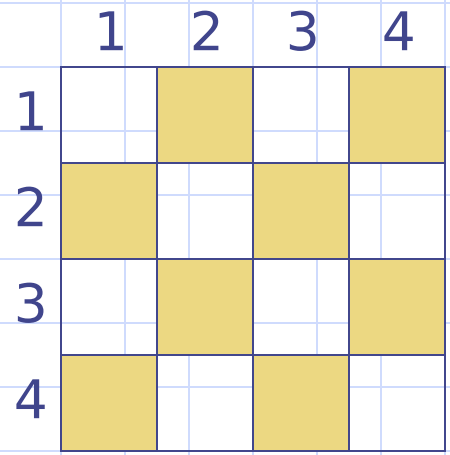


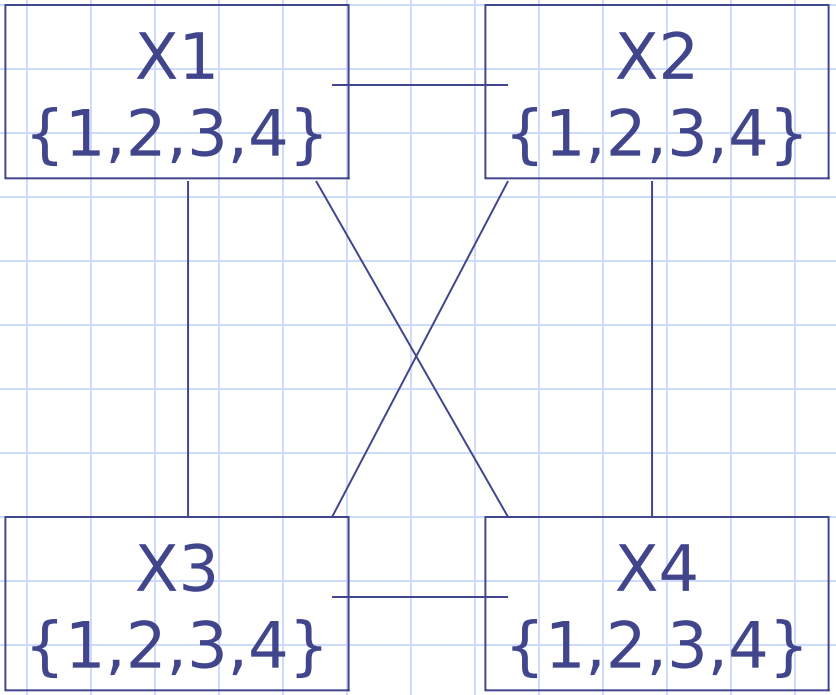
Forward Checking



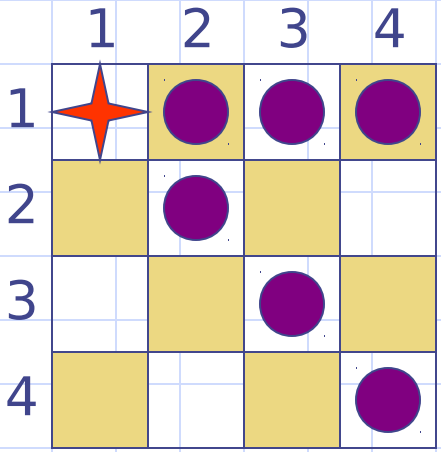
- É possível antecipar fracassos inevitáveis dado o estado corrente de atribuições ?
- ***Forward Checking* (FC)**: mantém controle dos valores remanescentes consistentes para cada variável ainda não atribuída.
- Antecipa o retorno (*backtrack*) quando alguma dessas variáveis se torna infactível, ou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

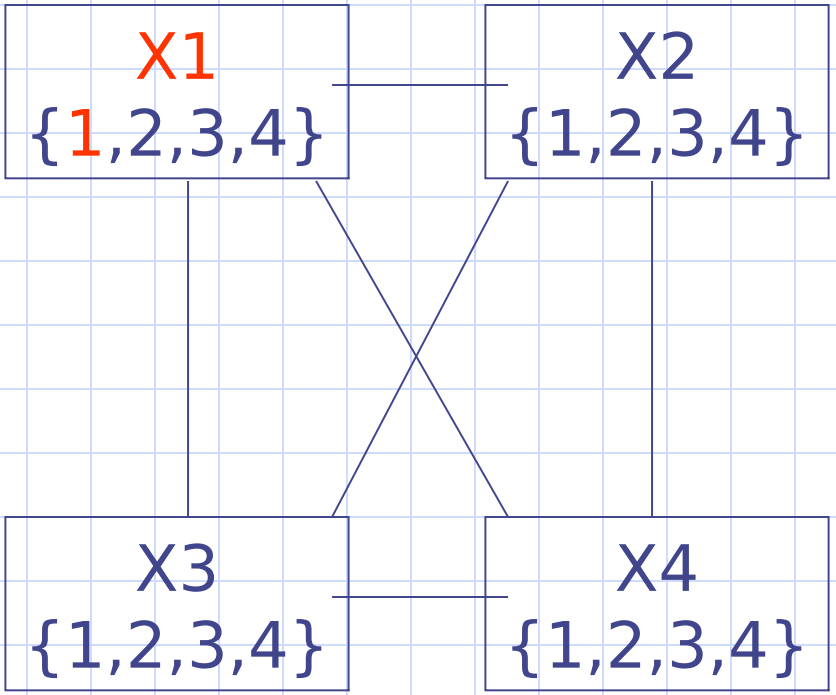
Exemplo: Problema 4 Rainhas



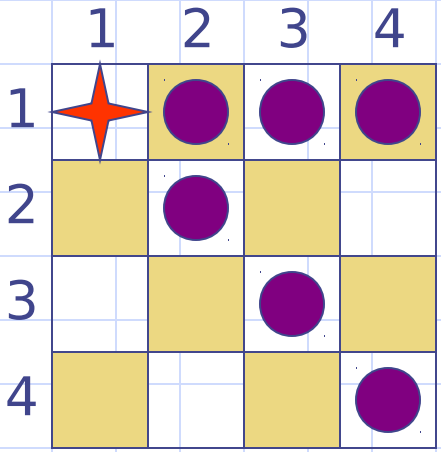


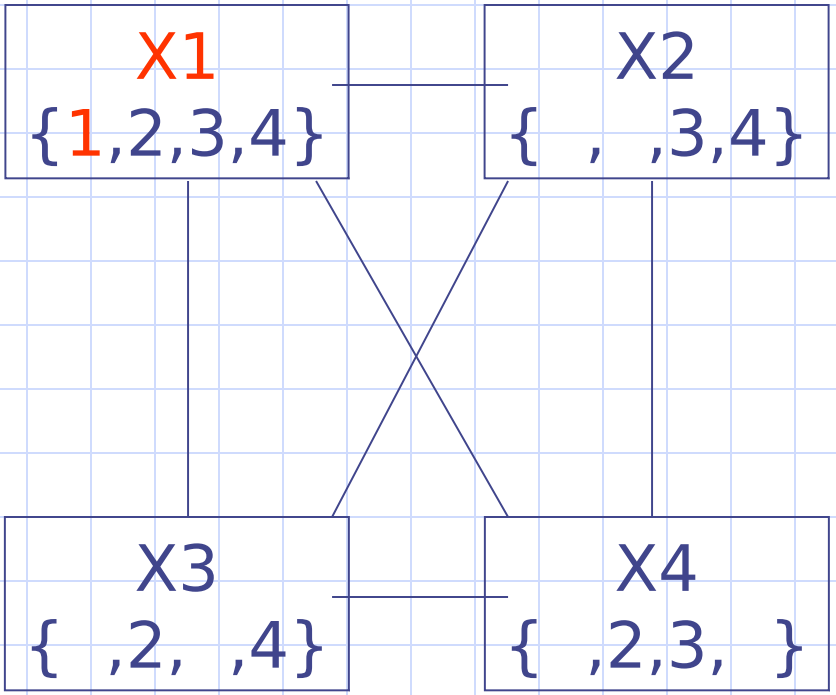
Exemplo: Problema 4 Rainhas



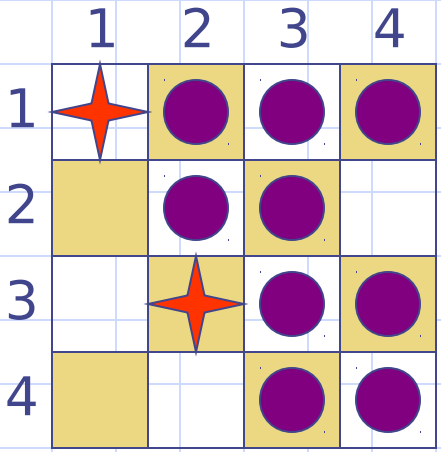


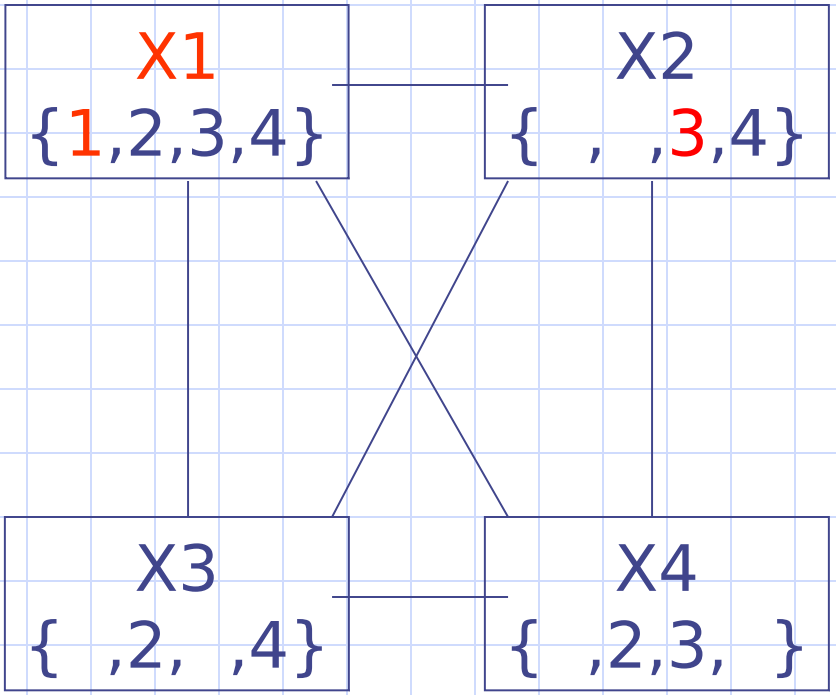
Exemplo: Problema 4 Rainhas



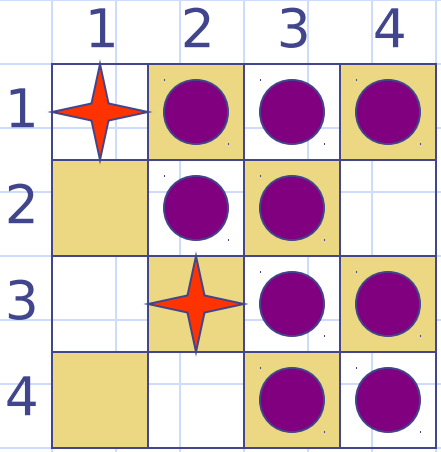


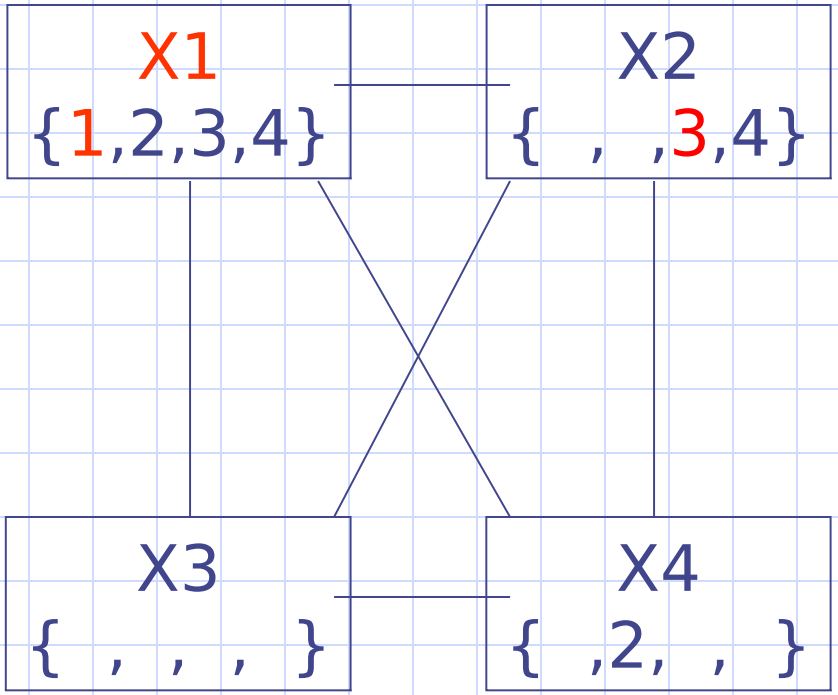
Exemplo: Problema 4 Rainhas





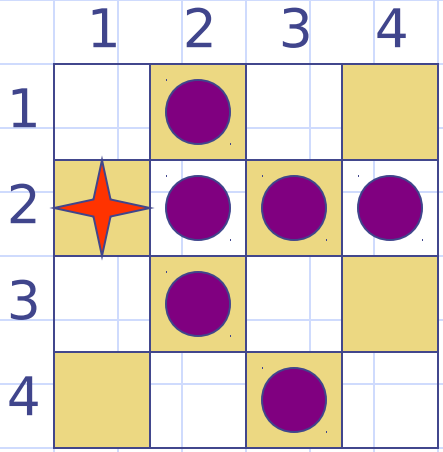
Exemplo: Problema 4 Rainhas

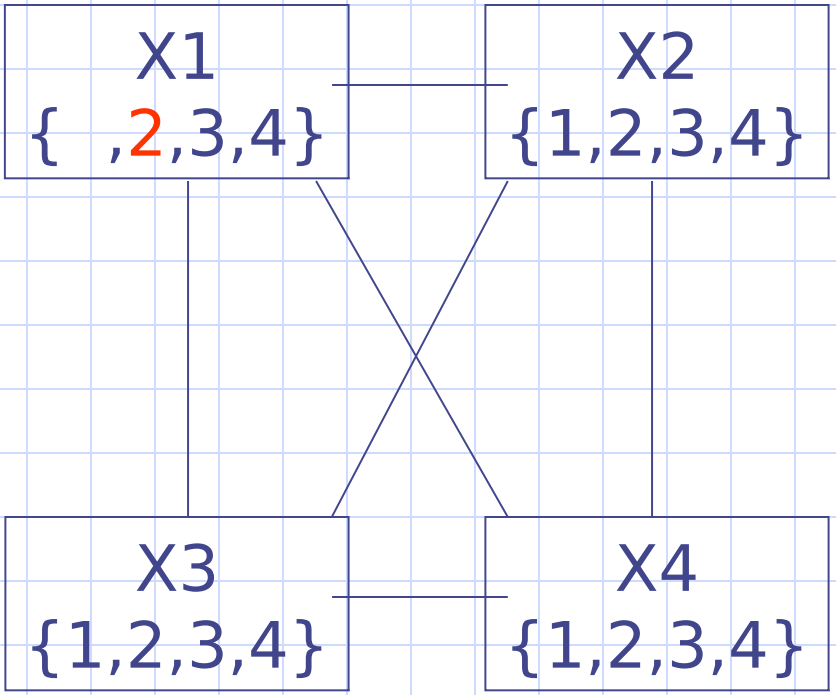




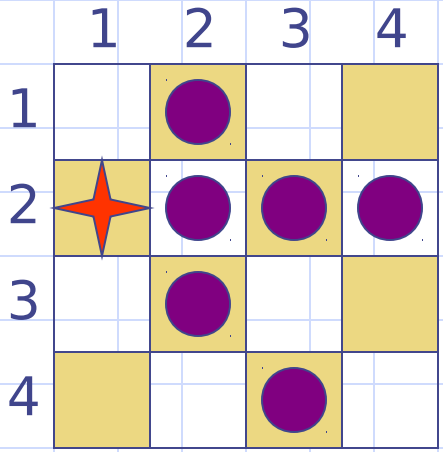
Nesse ponto FC detecta a inconsistência em X3 e antecipa o *backtrack*, indo imediatamente para o próximo valor de X2.

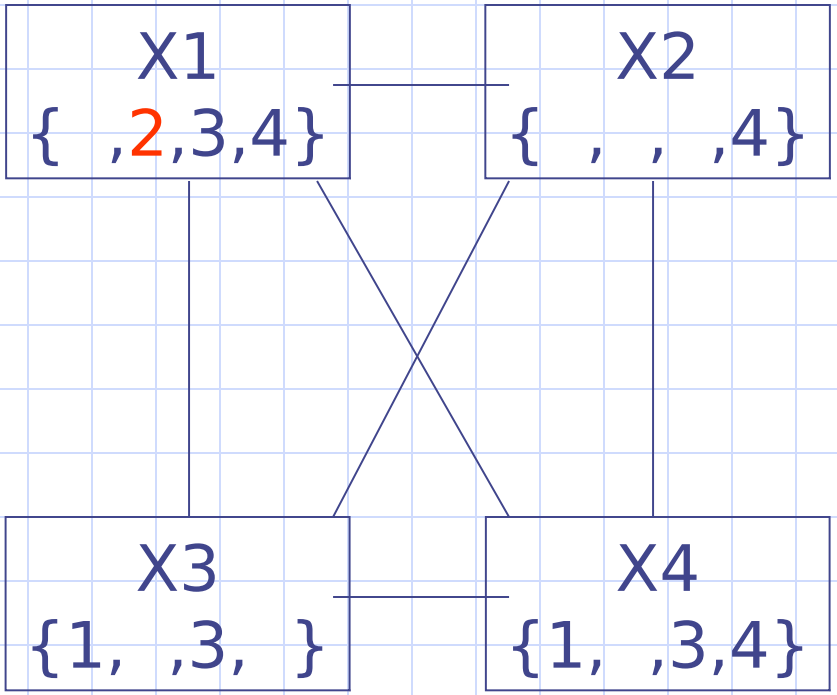
Exemplo: Problema 4 Rainhas



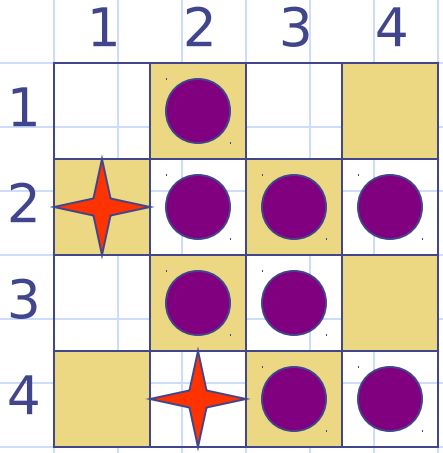


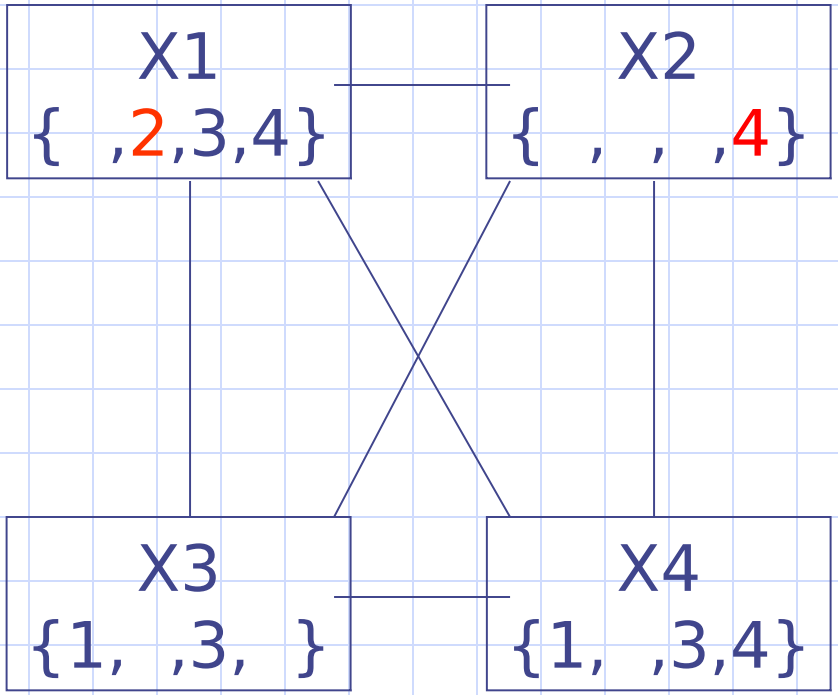
Exemplo: Problema 4 Rainhas



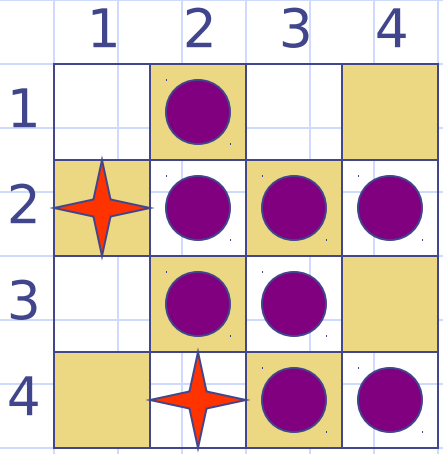


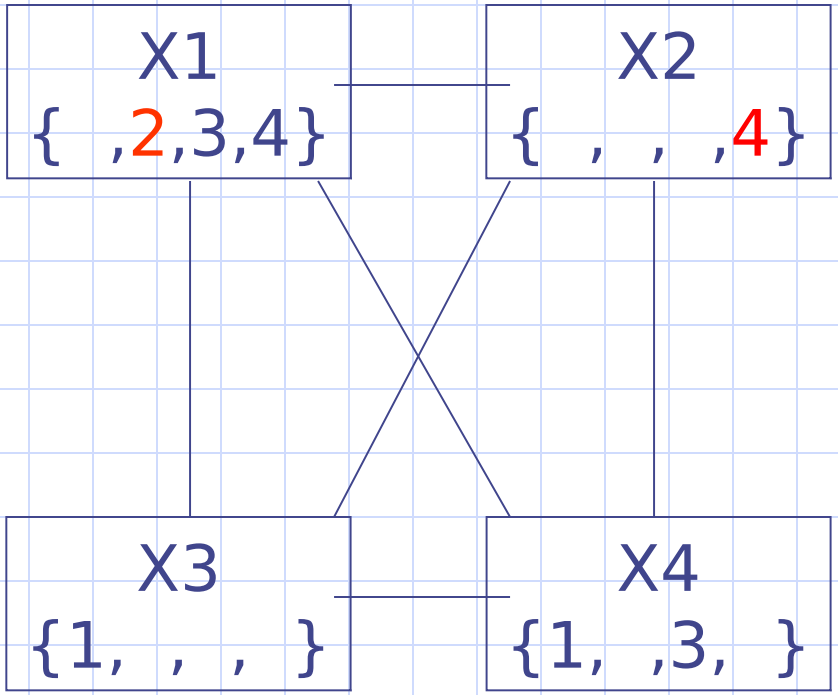
Exemplo: Problema 4 Rainhas



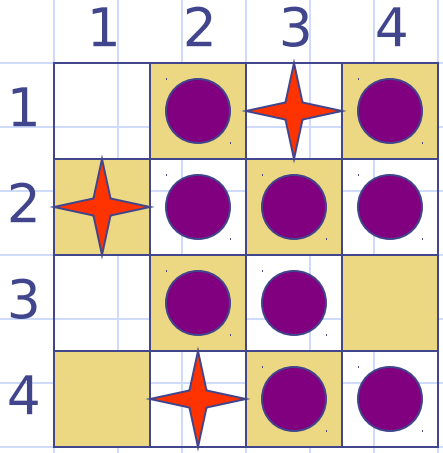


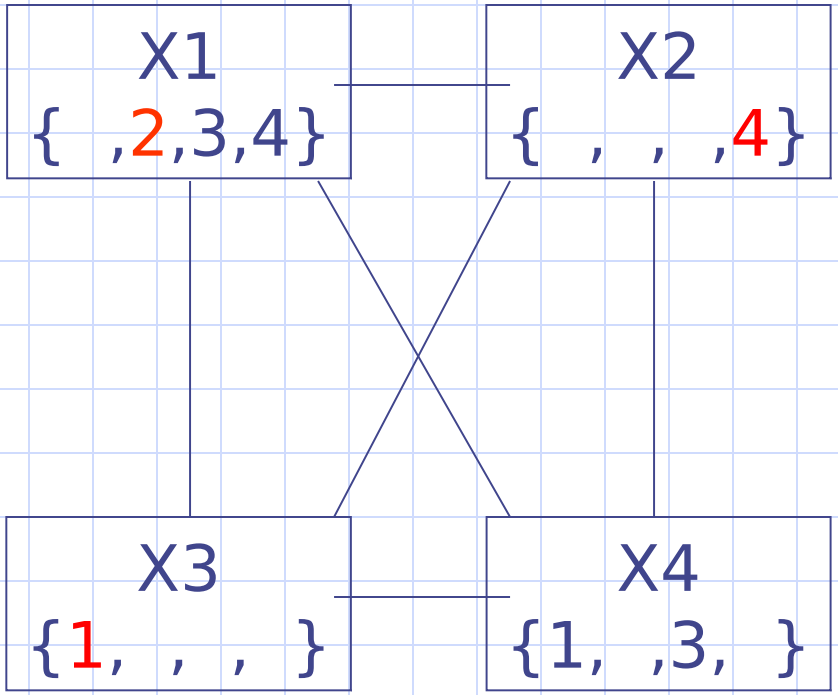
Exemplo: Problema 4 Rainhas



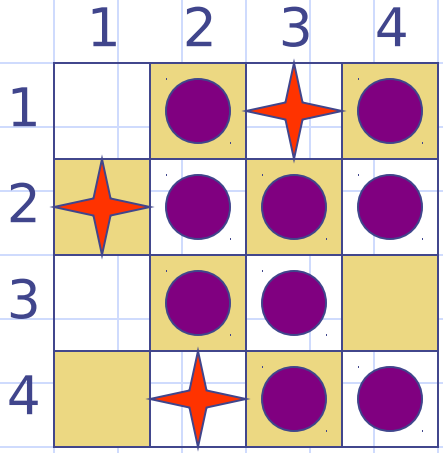


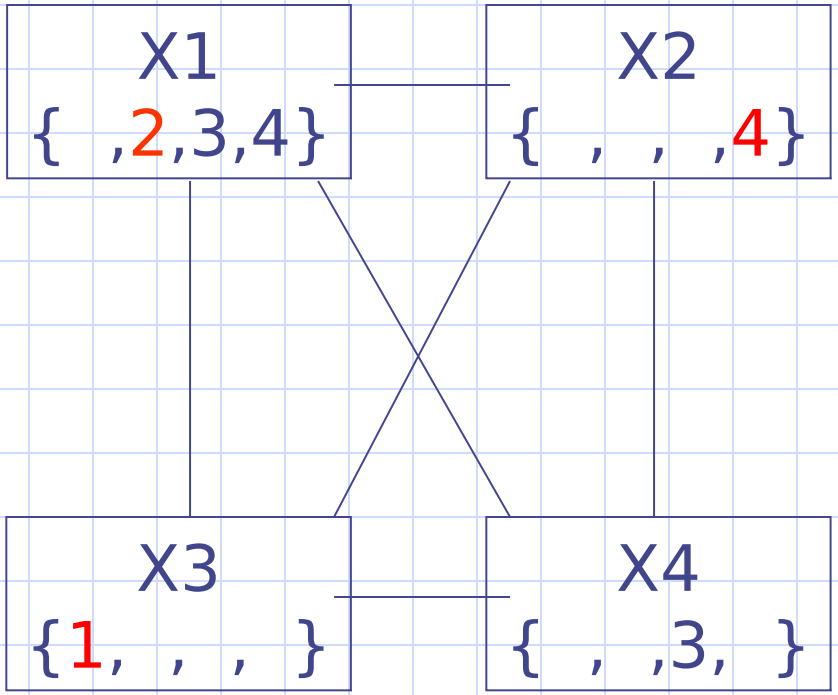
Exemplo: Problema 4 Rainhas



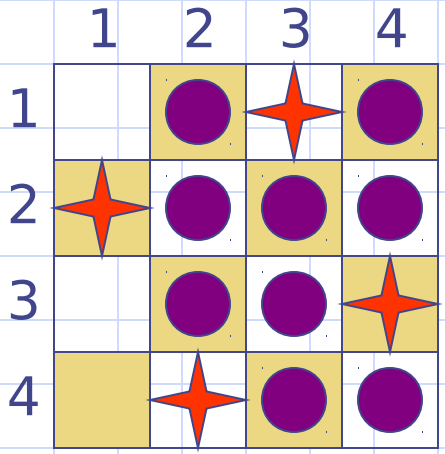


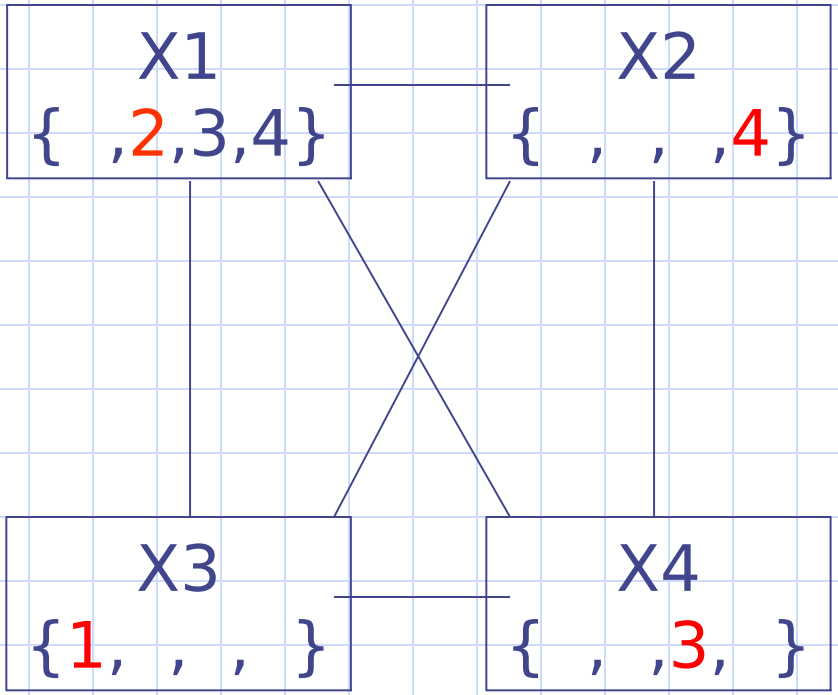
Exemplo: Problema 4 Rainhas



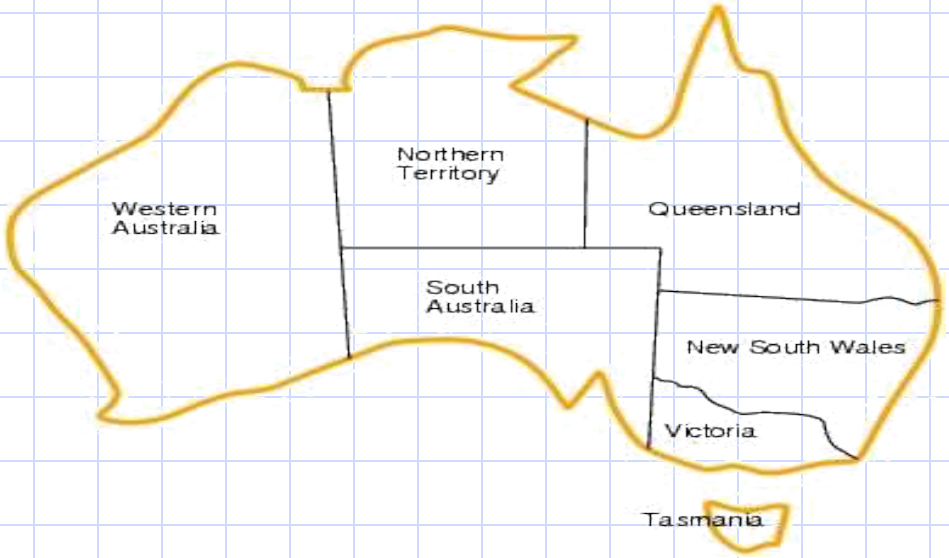


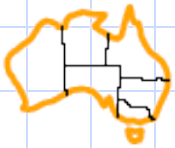
Exemplo: Problema 4 Rainhas

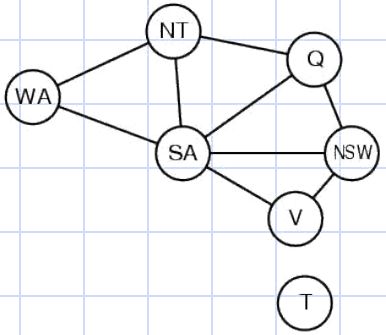




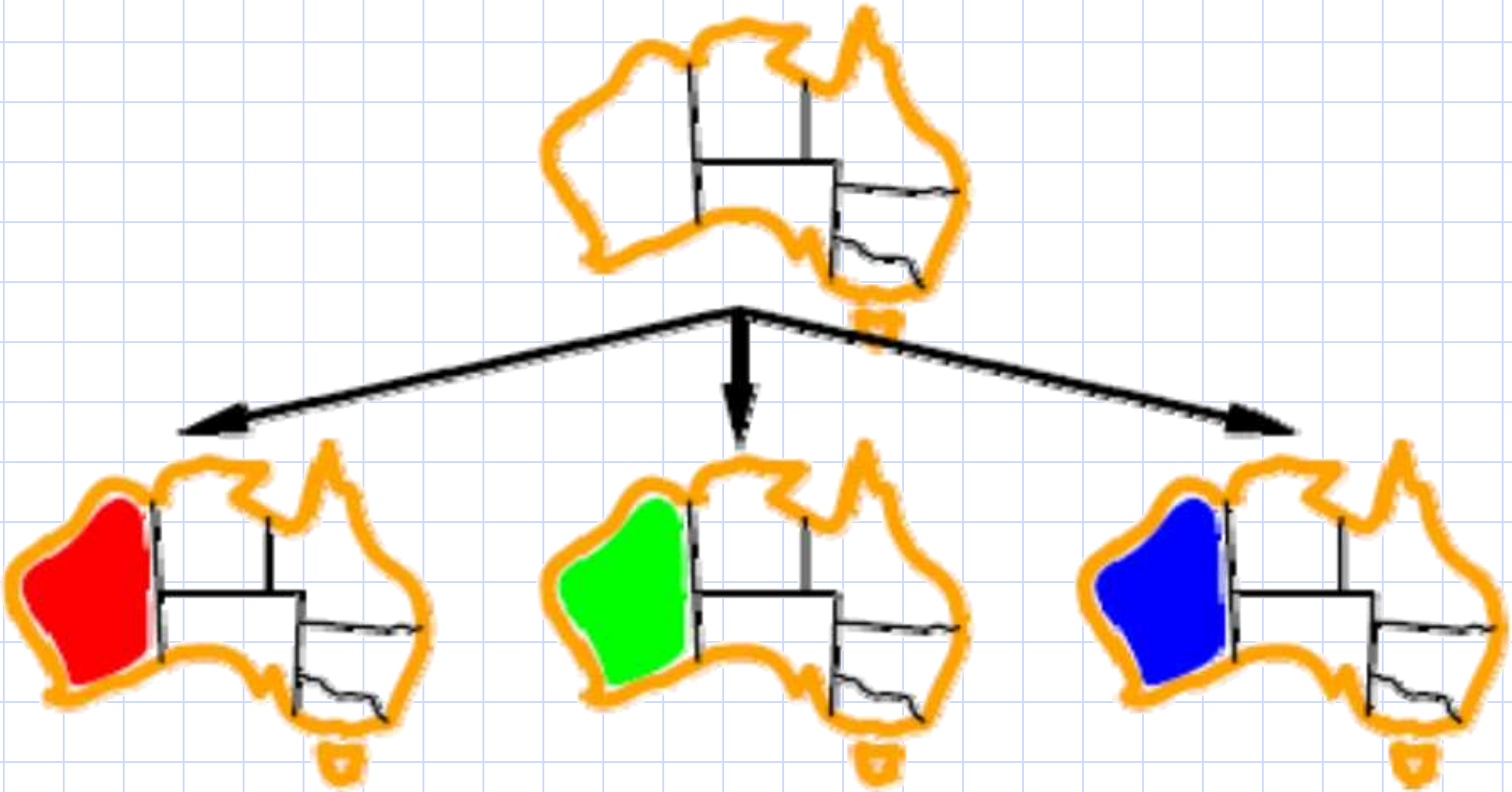
Exemplo – Coloração de Grafos



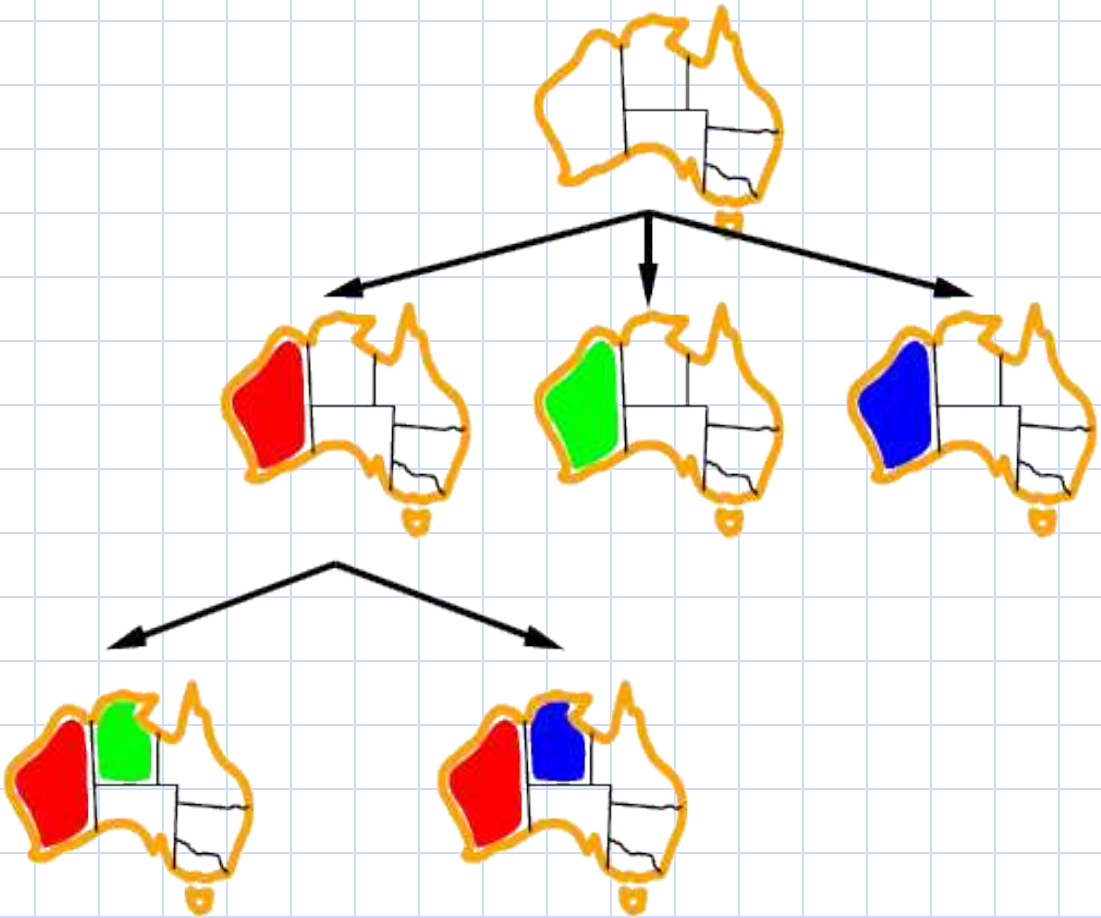




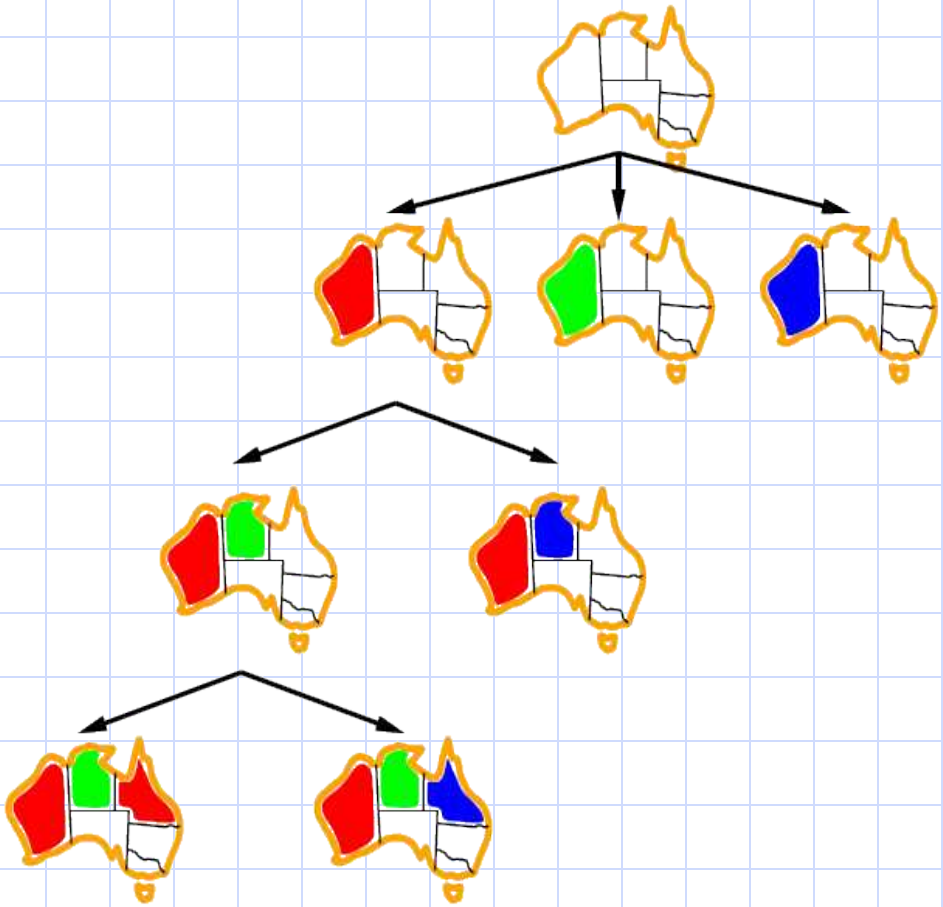
Exemplo – Coloração de Grafos



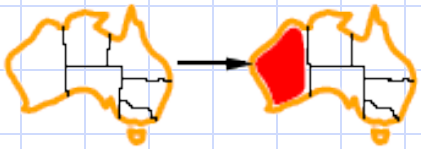
Exemplo – Coloração de Grafos

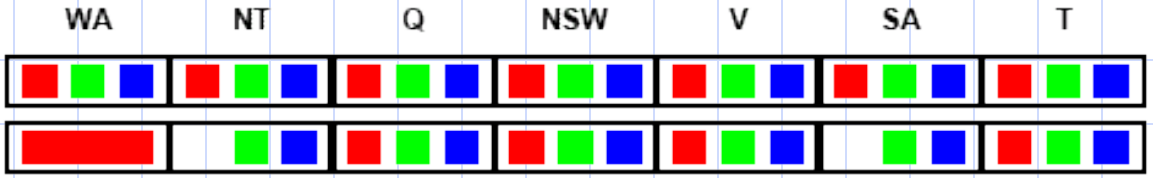


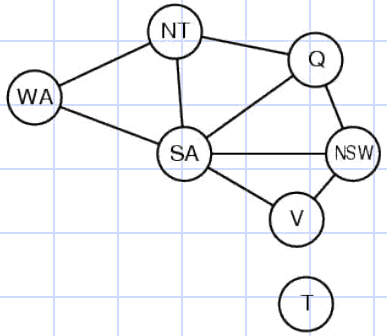
Exemplo – Coloração de Grafos



Forward Checking

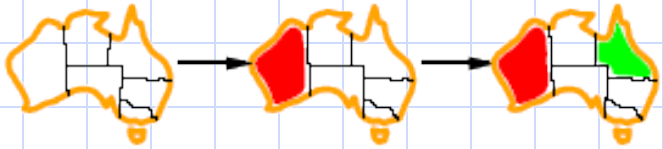


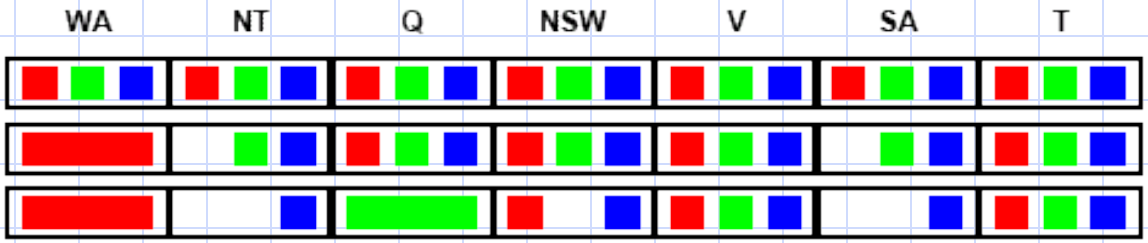


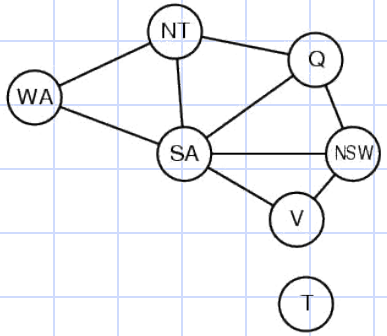


- Atribuição: *{WA = red}*
- Efeito sobre as outras variáveis conectadas por restrições:
  - *NT não pode mais ser red*
  - *SA não pode mais ser red*

Forward Checking

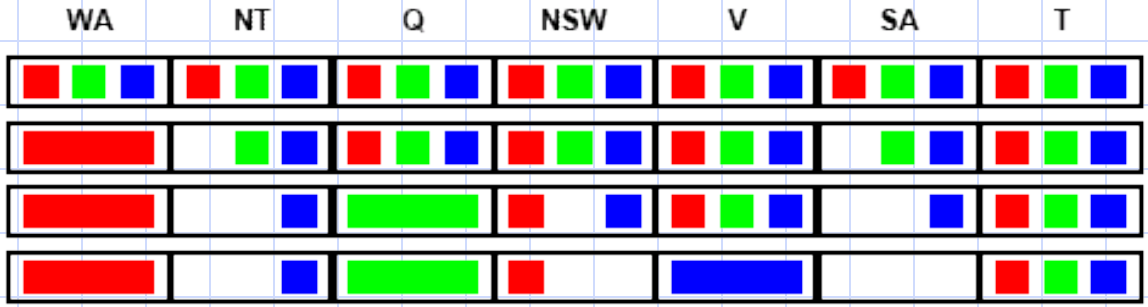


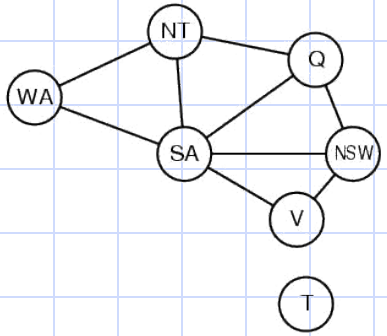




- Atribuição: *{Q = green}*
- Efeito sobre as outras variáveis conectadas por restrições:
  - *NT não pode mais ser green*
  - *NSW não pode mais ser green*
  - *SA não pode mais ser green*

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.

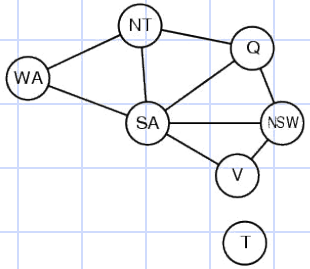
Heurísticas para *Backtracking*

- O uso de heurísticas em algoritmos de *backtracking* pode 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 ?

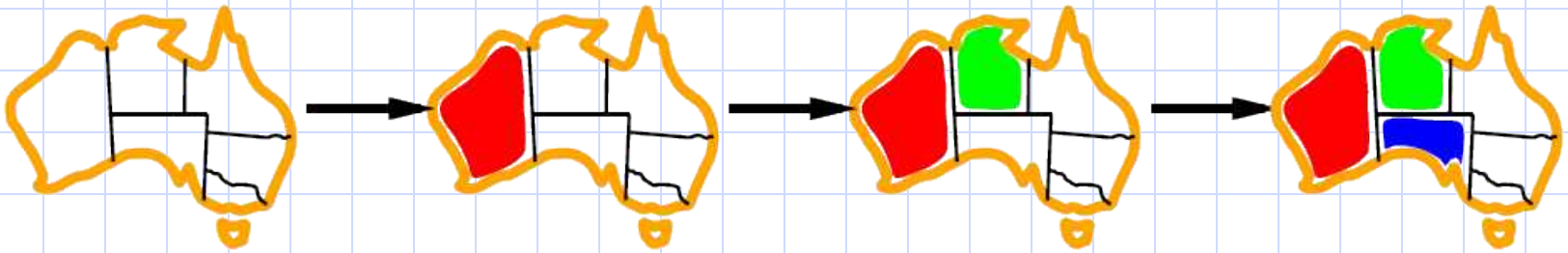
Qual a Próxima Variável?

- 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

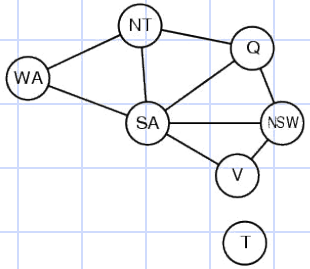


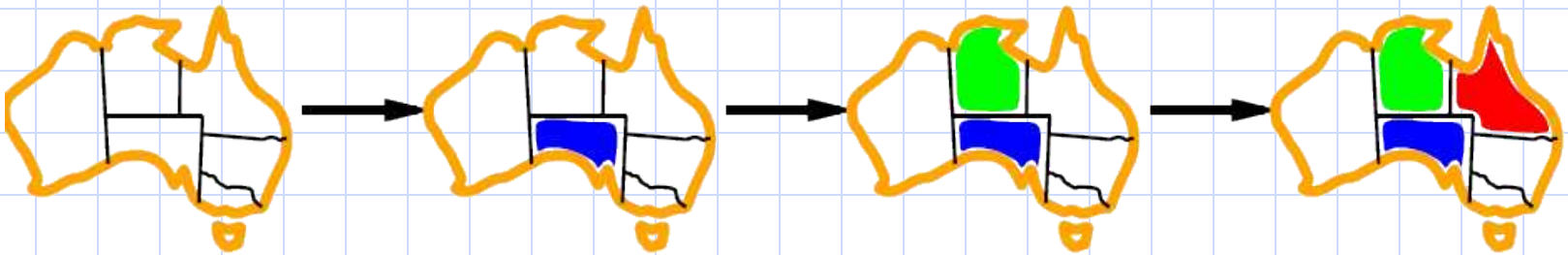
MRV = *Minimum Remaining Values*



- *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*



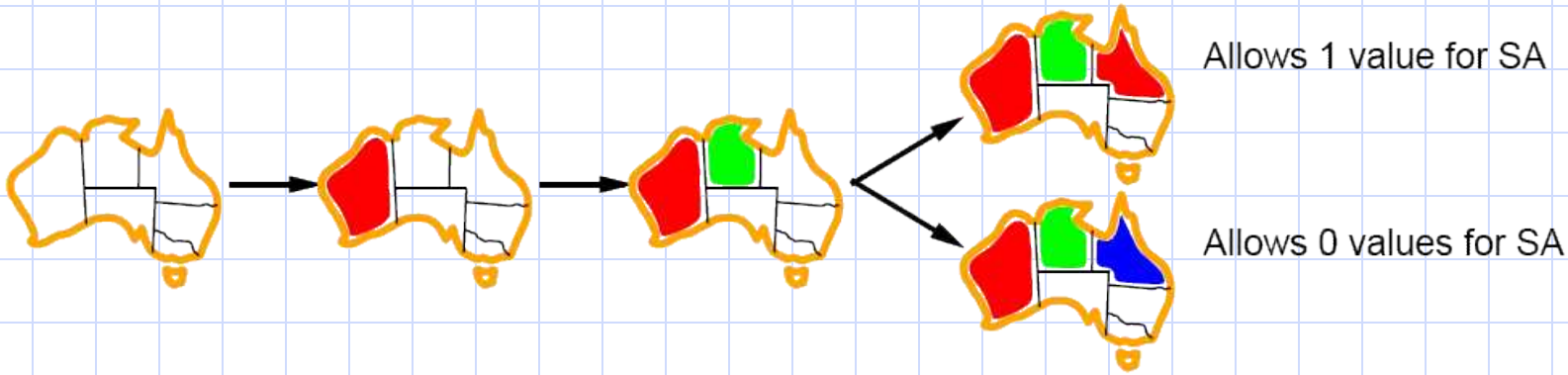


- *Regra*: escolher a variável que está envolvida em um maior número de restrições junto a outras variáveis ainda não atribuídas.
- *Idéia:*
  - Selecionar a variável que possui o maior potencial de se tornar infactível após a atribuição das demais.
  - Essa variável também é aquela que mais irá restringir 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).

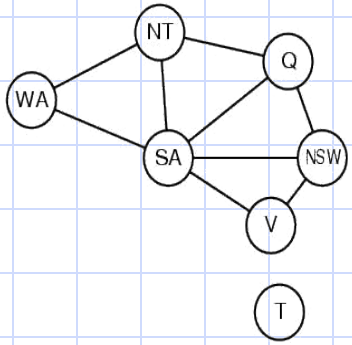
Qual o Próximo Valor ?

- 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.

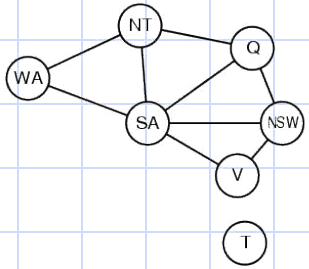
Heurística do Valor Menos Restritivo

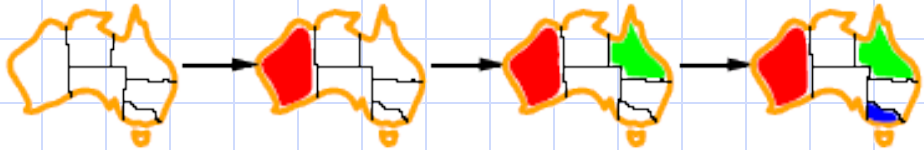


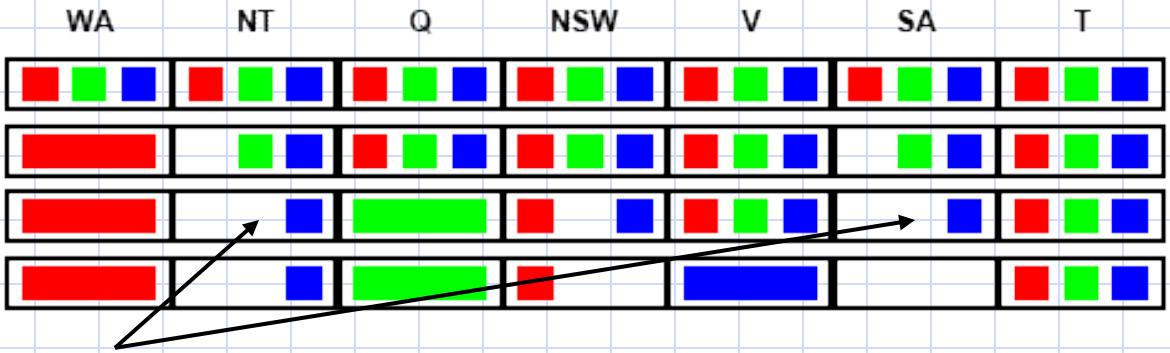
- *Regra*: dada uma variável, escolha os seus valores a partir do menos restritivo, i.e. daquele que deixa o máximo de flexibilidade para as atribuições de variáveis subseqüentes.



Tópicos Avançados





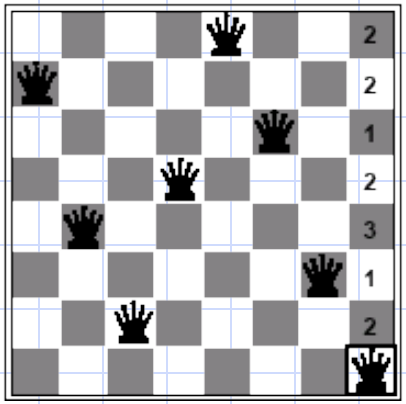


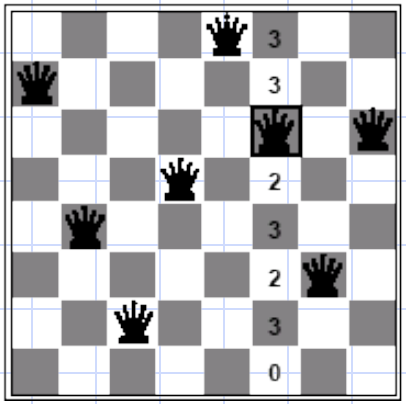
- O tipo de *propagação de restrições* implementado por FC não é capaz de detectar antecipadamente todas as possíveis inconsistências:
  - e.g. NT e SA já não podiam ser azuis antes da atribuição de V !
- Abordagens de propagação de restrições mais sofisticadas:
  - **Consistência de Arcos**
  - ***k*-Consistência**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

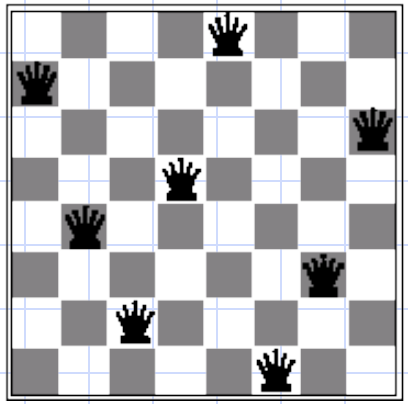
Tópicos Relacionados

- Métodos Heurísticos de Busca Local:
  - Subida de Encosta (*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



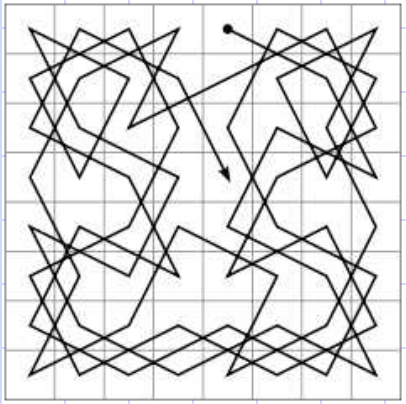


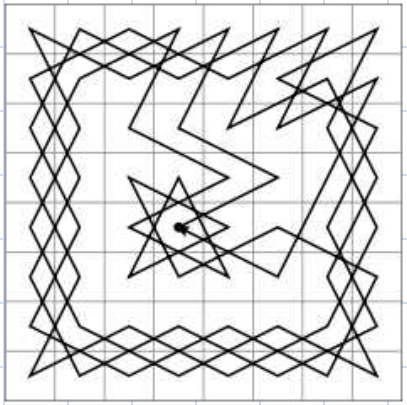


- Uma solução de dois passos para o problema das 8 rainhas utilizando a heurística dos conflitos mínimos.
- Em cada passo uma rainha é re-posicionada em sua coluna.
- O algoritmo move a rainha para o quadrado de conflito mínimo, resolvendo empates aleatoriamente.
- Heurística bastante insensível ao tamanho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

Percurso do Cavalo Revisitado

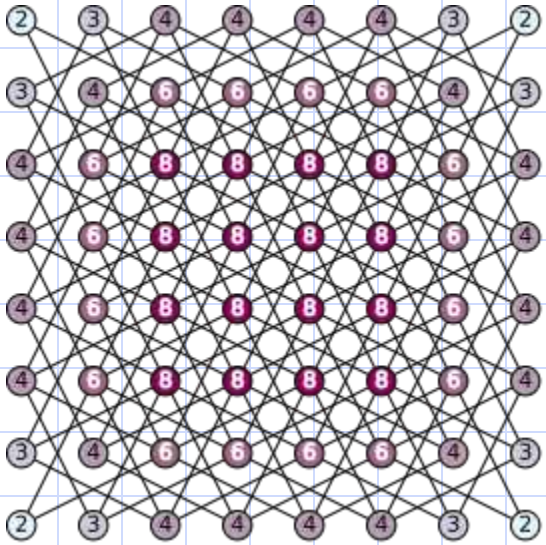
- O problema do percurso do cavalo envolve encontrar um caminho Hamiltoniano (similar ao TSP) e é NP-Completo, portanto exponencial.
- Entretanto, as regularidades existentes permitem encontrar algoritmos lineares.





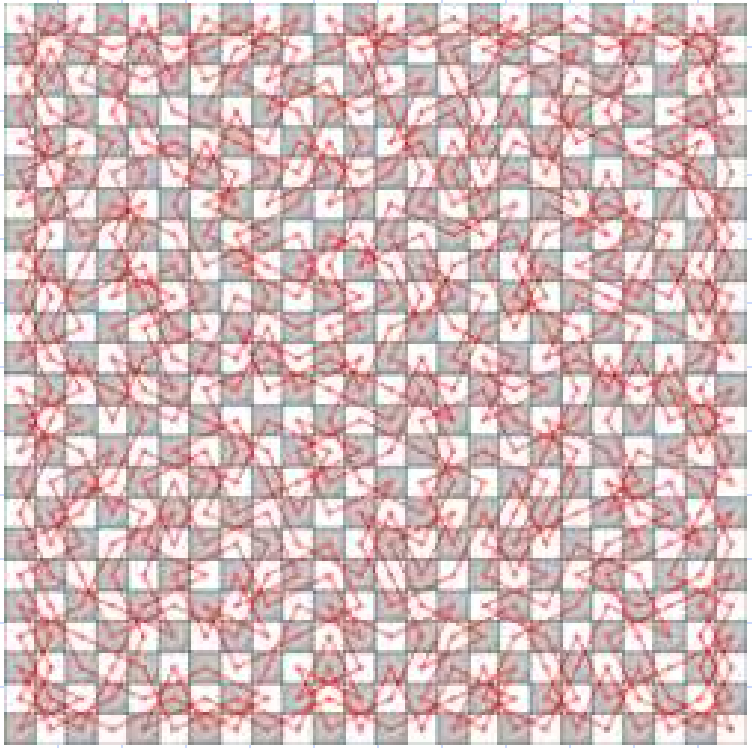
Percurso do Cavalo Revisitado

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



Percurso do Cavalo Revisitado

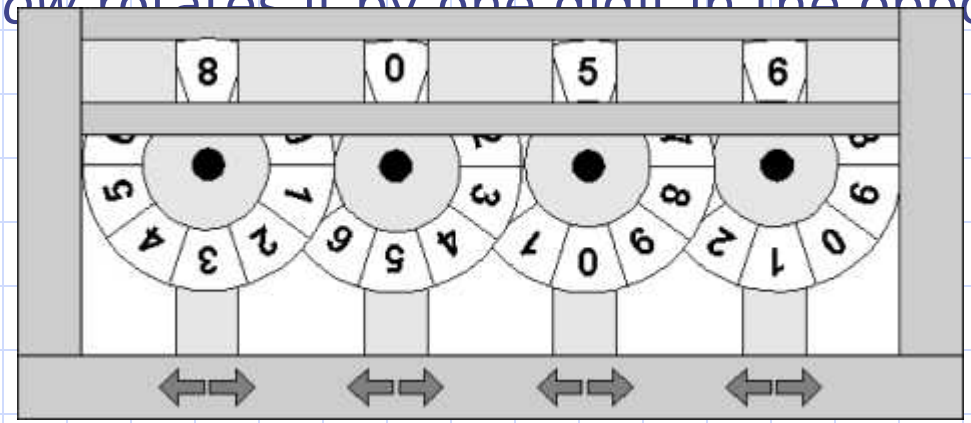
- E outros algoritmos linerares (Conrad *et al.*, 1994) e baseados em redes neurais (Takefuji & Lee, 1992) podem solucionar grandes instâncias.



Playing with Wheels

Popularidade: C, Sucess rate: average, Level: 2

Consider the following mathematical machine. Digits ranging from 0 to 9 are printed consecutively (clockwise) on the periphery of each wheel. The topmost digits of the wheels form a four-digit integer. For example, in the following figure the wheels form the integer 8,056. Each wheel has two buttons associated with it. Pressing the button marked with a *left arrow* rotates the wheel one digit in the clockwise direction and pressing the one marked with the *right arrow* rotates it by one digit in the opposite direction.



Playing with Wheels

We start with an initial configuration of the wheels, with the topmost digits forming the integer $S_1S_2S_3S_4$. You will be given a set of n forbidden configurations $F_{i1}F_{i2}F_{i3}F_{i4}$ $(1 \leq i \leq n)$ and a target configuration $T_1T_2T_3T_4$. Your job is to write a program to calculate the minimum number of button presses required to transform the initial configuration to the target configuration without passing through a forbidden one.

Playing with Wheels

Input

The first line of the input contains an integer *N* giving the number of test cases. A blank line then follows. The first line of each test case contains the initial configuration of the wheels, specified by four digits. Two consecutive digits are separated by a space. The next line contains the target configuration. The third line contains an integer *n* giving the number of forbidden configurations. Each of the following *n* lines contains a forbidden configuration. There is a blank line between two consecutive input sets.

Output

For each test case in the input print a line containing the minimum number of button presses required. If the target configuration is not reachable print “-1”.

Playing with Wheels

- Qual é o grafo por trás deste problema?

Playing with Wheels

- Qual é o grafo por trás deste problema?
  - Vértices: estados (números de 4 dígitos);
  - Arestas: possíveis transições entre estados.

Playing with Wheels

- Qual é o grafo por trás deste problema?
  - Vértices: estados (números de 4 dígitos);
  - Arestas: possíveis transições entre estados.
- Qual é o grau de cada vértice deste grafo?

Playing with Wheels

- Qual é o grafo por trás deste problema?
  - Vértices: estados (números de 4 dígitos);
  - Arestas: possíveis transições entre estados.
- Qual é o grau de cada vértice deste grafo?
  - Exatamente oito, pois existem oito possíveis transições a partir de cada estado.

Playing with Wheels

- Qual é o grafo por trás deste problema?
  - Vértices: estados (números de 4 dígitos);
  - Arestas: possíveis transições entre estados.
- Qual é o grau de cada vértice deste grafo?
  - Exatamente oito, pois existem oito possíveis transições a partir de cada estado.
- Como é possível encontrar o caminho mínimo?

Playing with Wheels

- Qual é o grafo por trás deste problema?
  - Vértices: estados (números de 4 dígitos);
  - Arestas: possíveis transições entre estados.
- Qual é o grau de cada vértice deste grafo?
  - Exatamente oito, pois existem oito possíveis transições a partir de cada estado.
- Como é possível encontrar o caminho mínimo?
  - A forma mais simples é utiliza uma busca em largura, pois o grafo é não-ponderado.

Playing with Wheels

- É necessário representar o grafo explicitamente?

Playing with Wheels

- É necessário representar o grafo explicitamente?
  - Não. Podemos construir uma função que retorna os próximos estados dado o estado atual;
  - Backtracking com fringe organizado em uma fila.
- Os estados já visitados podem ser marcados em uma matriz.

Busca versus *Backtracking*

- Mesmo conceito:
  - Busca em grafos: grafo explícito;
  - *Backtracking*: grafo implícito

- Conseqüência:
  - Busca em grafos: pode-se consultar o grafo para se saber os vértices adjacentes;
  - *Backtracking*: deve-se ter uma sub-rotina que gera os próximos “vértices” e verifica se são válidos.

Playing with Wheels

```
#include<stdio.h>
#include<queue>

using namespace std;

struct state {
    char digit[4];
    int depth;
};

char moves[8][4] = {{-1, 0, 0, 0},
                    {1, 0, 0, 0},
                    {0, -1, 0, 0},
                    {0, 1, 0, 0},
                    {0, 0, -1, 0},
                    {0, 0, 1, 0},
                    {0, 0, 0, -1},
                    {0, 0, 0, 1}};
```

Playing with Wheels

```
void next_states(state s, state nexts[8]) {
     int i,j;

    for (i=0; i < 8; i++) {
        nexts[i] = s;
        nexts[i].depth = s.depth+1;
        for (j = 0; j < 4; j++) {
            nexts[i].digit[j] += moves[i][j];
            if (nexts[i].digit[j] < 0)
                nexts[i].digit[j] = 9;
            if (nexts[i].digit[j] > 9)
                nexts[i].digit[j] = 0;
        }
    }
}
```

```
main() {
    int nr_tests, test, forbidden, i, j, k, l, d;
    char visited[10][10][10][10];
    state initial, final, aux;

    scanf("%d", &nr_tests);
    for (test=0; test < nr_tests; test++) {
        scanf("%d %d %d %d", &initial.digit[0], &initial.digit[1],
                             &initial.digit[2], &initial.digit[3]);
        scanf("%d %d %d %d", &final.digit[0], &final.digit[1],
                             &final.digit[2], &final.digit[3]);
        scanf("%d", &forbidden);
        for(i=0; i<10; i++)
            for(j=0; j<10; j++)
                for(k=0; k<10; k++)
                    for(l=0; l<10; l++)
                        visited[i][j][k][l] = 0;
        for(i=0; i<forbidden; i++) {
            scanf("%d %d %d %d", &aux.digit[0], &aux.digit[1],
                                 &aux.digit[2], &aux.digit[3]);
            visited[aux.digit[0]][aux.digit[1]]
                   [aux.digit[2]][aux.digit[3]] = 1;
        }
        initial.depth=0;
        printf("%d\n", bfs(initial,final,visited));
    }
}
```

```
int bfs(state current, state final, char visited[10][10][10][10]) {
    state nexts[8];
    int i;
    queue<state> q;
                        /* Argh, o estado inicial pode ser proibido */
    if (!visited[current.digit[0]][current.digit[1]]
                [current.digit[2]][current.digit[3]]) {
        q.push(current);
        while (!q.empty()) {
            current = q.front();
            q.pop();
            if (equal(current, final)) return current.depth;
            next_states(current, nexts);
            for (i = 0; i < 8; i++)
                if (visited[nexts[i].digit[0]][nexts[i].digit[1]]
                           [nexts[i].digit[2]][nexts[i].digit[3]]) {
                    visited[nexts[i].digit[0]][nexts[i].digit[1]]
                           [nexts[i].digit[2]][nexts[i].digit[3]] = 1;
                    q.push(nexts[i]);
                }
        }
    }
    return -1;
}
```

```
int bfs(state current, state final, char visited[10][10][10][10]) {
    state nexts[8];
    int i;
    queue<state> q;
                        /* Argh, o estado inicial pode ser proibido */
    if (visited[current.digit[0]][current.digit[1]]
                [current.digit[2]][current.digit[3]]) {
        q.push(current);
        while (!q.empty()) {
            current = q.front();
            q.pop();
            if (equal(current, final)) return current.depth;
            next_states(current, nexts);
            for (i = 0; i < 8; i++)
                if (visited[nexts[i].digit[0]][nexts[i].digit[1]]
                           [nexts[i].digit[2]][nexts[i].digit[3]]) {
                    visited[nexts[i].digit[0]][nexts[i].digit[1]]
                           [n
                    q.push(ne
                }
        }
    }
    return -1;
}
```

```
int equal(state s, state e) {
    int i;

    for (i=0; i < 4; i++)
        if (s.digit[i] != e.digit[i])
            return 0;
    return 1;
}
```

Playing with Wheels

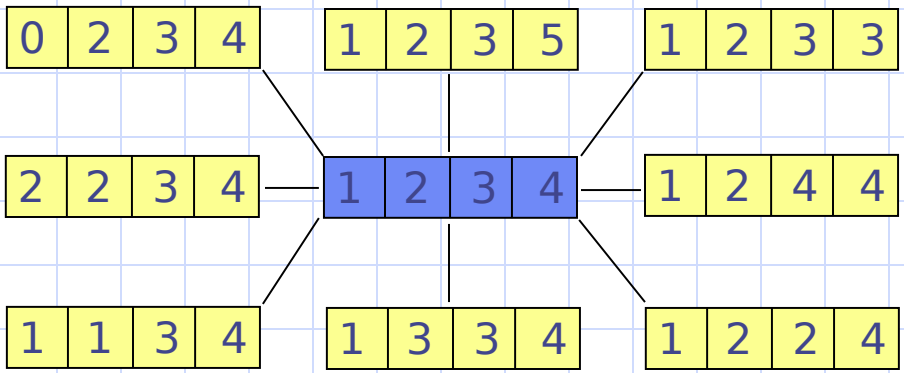
- A solução apresentada necessita de 0.973s para solucionar os casos do teste do UVA.
- É possível encontrar um solução mais eficiente?

Playing with Wheels

- A solução apresentada necessita de 0.973s para solucionar os casos do teste do UVA.
- É possível encontrar um solução mais eficiente?
  - Uma possibilidade é incorporar alguma heurística;
  - Mais isso deve ser feito com cuidado.

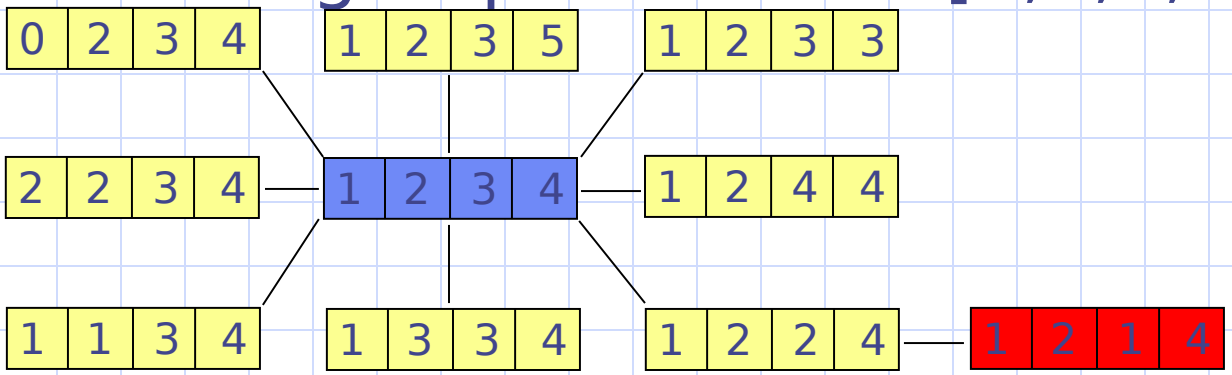
Playing with Wheels: Heurística

- Com a busca em largura, dado um estado (vértice) todos os vértices adjacentes não visitados são inseridos na fila *q*.



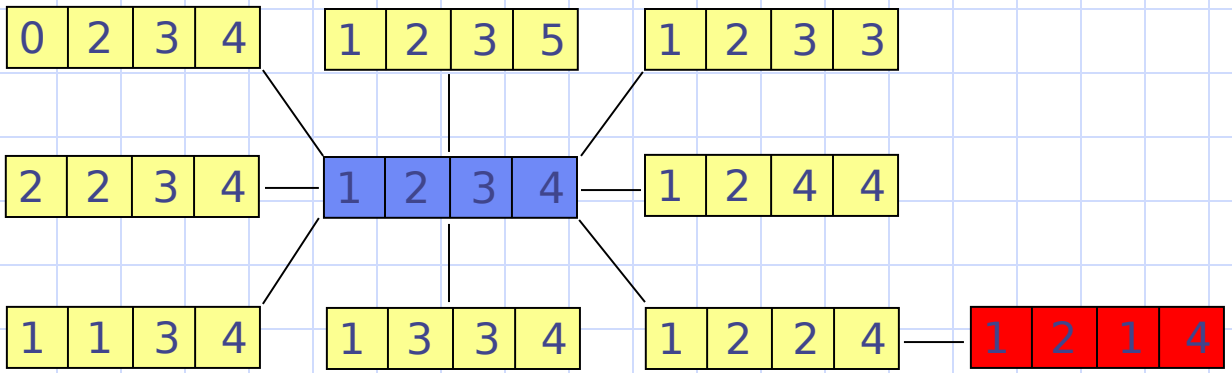
Playing with Wheels: Heurística

- Com a busca em largura, dado um estado (vértice) todos os vértices adjacentes não visitados são inseridos na fila *q*.
- Sendo [1,2,3,4] o estado inicial, imagine que o estado final é [1,2,1,4]. Então é melhor seguir pelo vértice [1,2,2,4].



Playing with Wheels: Heurística

- A função heurística pode ser então a "distância direta" entre o estado analisado e o estado final:
  - d([1,2,3,4], [1,2,2,4]) = 1
  - d([1,2,3,4], [1,2,1,4]) = 2

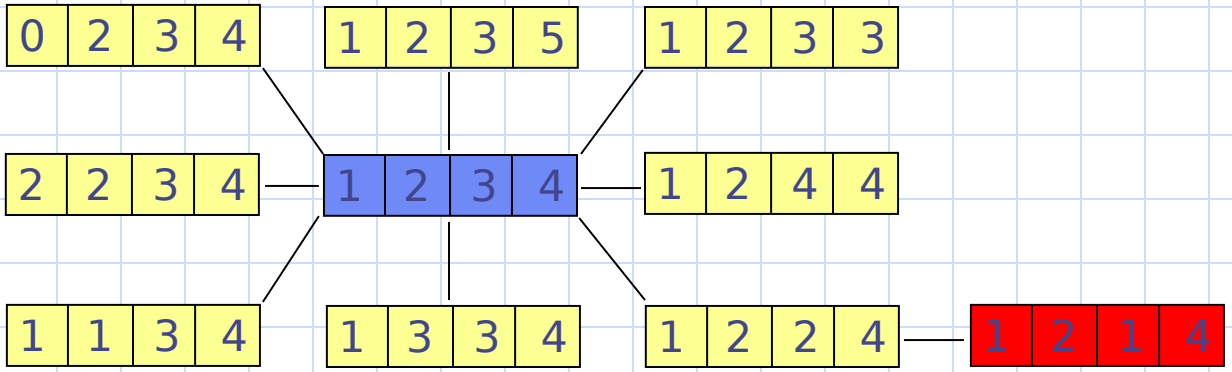


Playing with Wheels: Heurística

- A função heurística pode ser então a "distância direta" entre o estado analisado e o estado final:
  - d([1,2,3,4], [1,2,2,4]) = 1
  - d([1,2,3,4], [1,2,1,4]) = 2
- A existência de estados proibidos faz com que a função heurística seja uma estimativa otimista.

Playing with Wheels: Heurística

- A função heurística pode ser incorporada alterando-se a fila da busca em largura por uma fila de prioridade (ordenada pela função heurística).
  - *pq* = (1:[1,2,2,4], 3:[1,2,4,4], 3:[1,2,3,3], 3:[1,2,3,5], 3:[0,2,3,4], 3:[2,2,3,4], 3:[1,1,3,4], 3:[1,3,3,4])

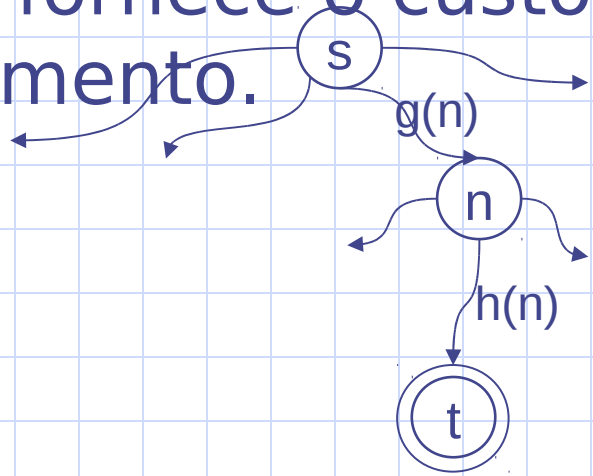


Playing with Wheels: A*

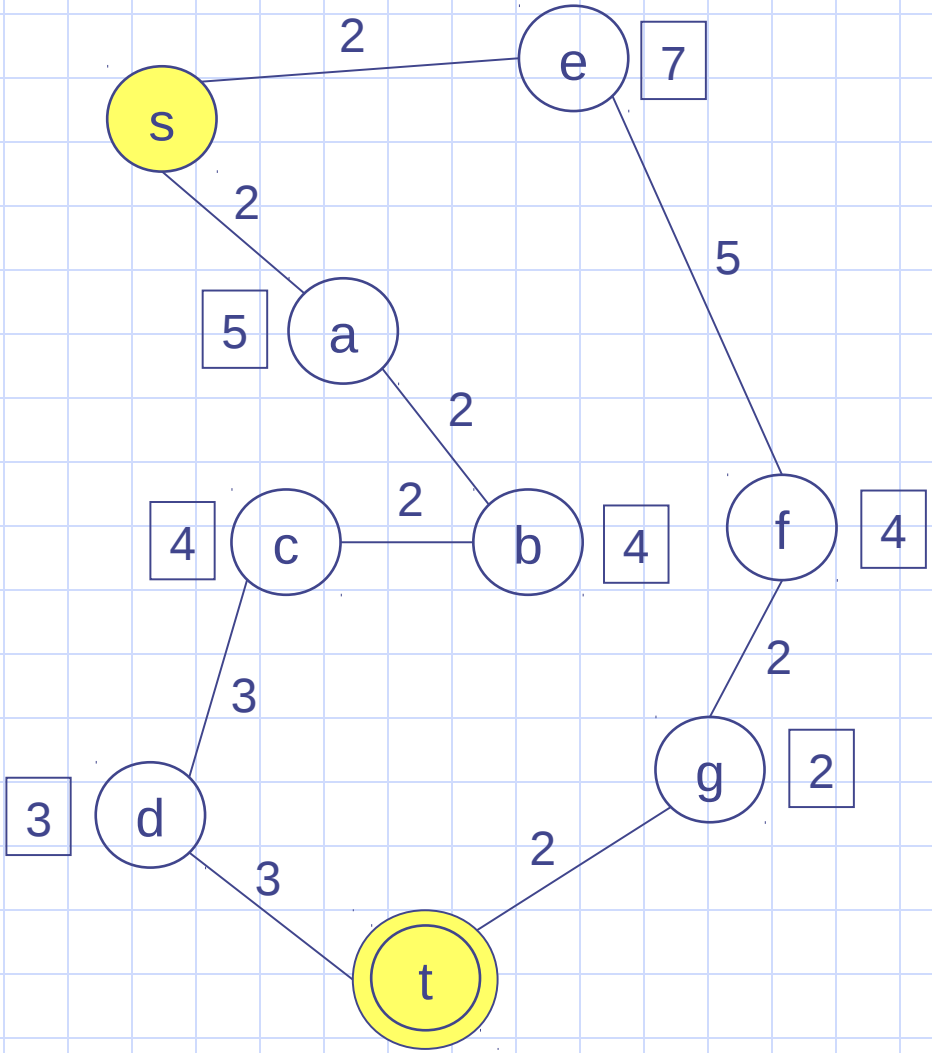
- A busca descrita é chamada de *best-first*.
- Mas essa busca possui uma limitação: ela não garante que o menor caminho será encontrado.
- Uma variação dela, chamada A*, fornece essa garantia.
- A* = *best-first* + função de custo + função heurística **admissível**.

Playing with Wheels: A*

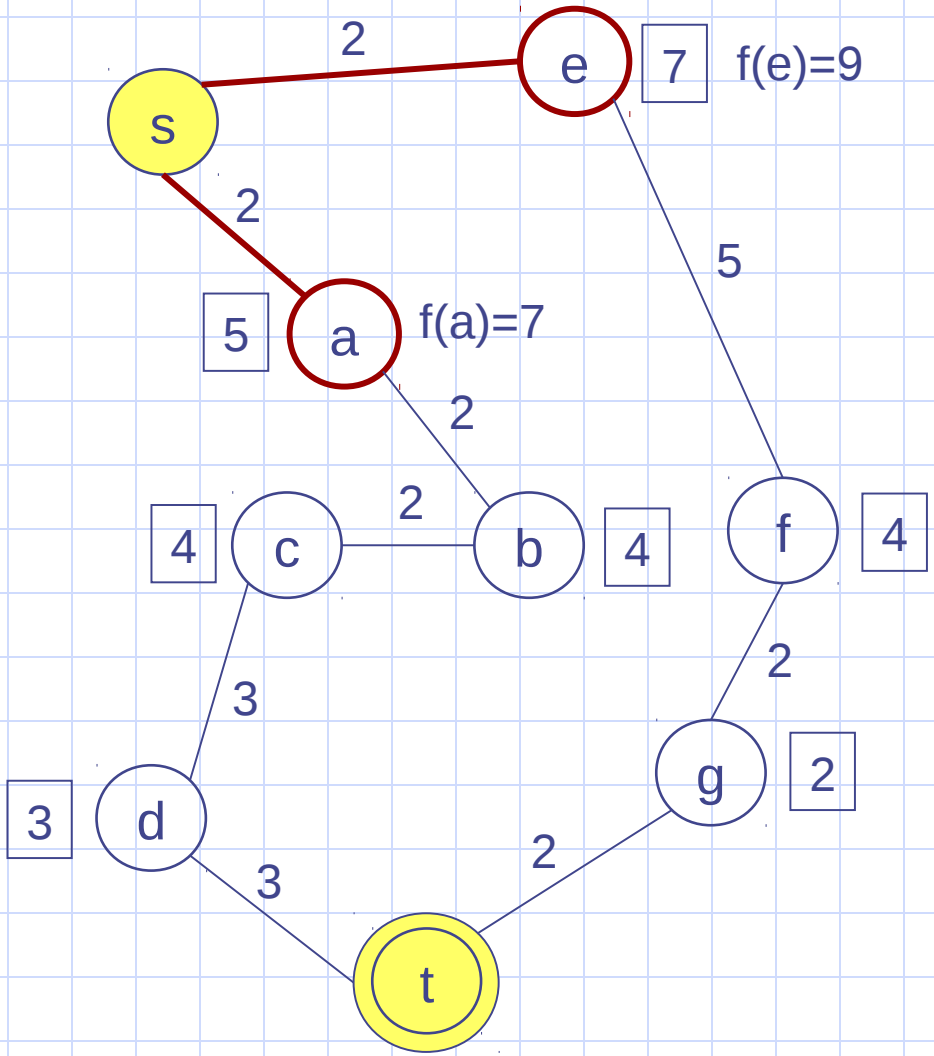
- Uma função heurística admissível nunca superestima a custo real da melhor solução.
- A função heurística fornece uma estimativa do esforço futuro.
- A função de custo fornece o custo do caminho até o momento.



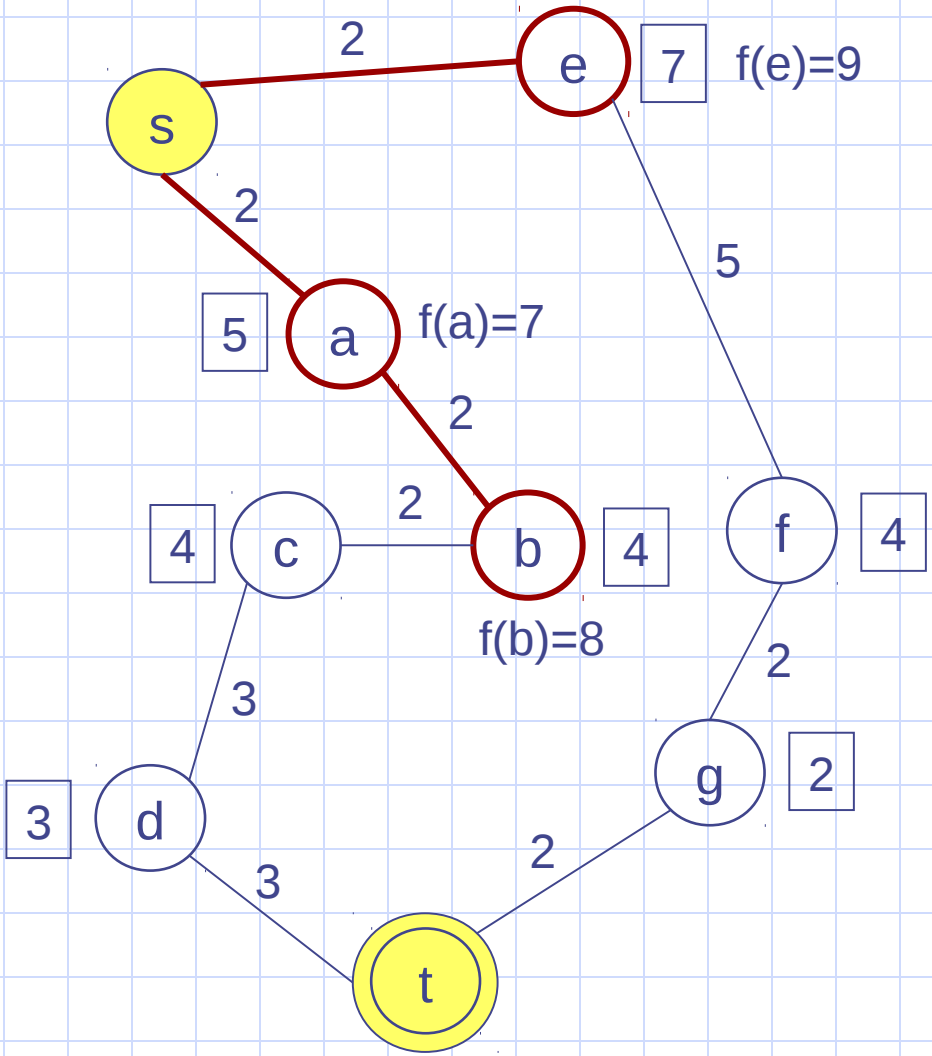
Playing with Wheels: A*



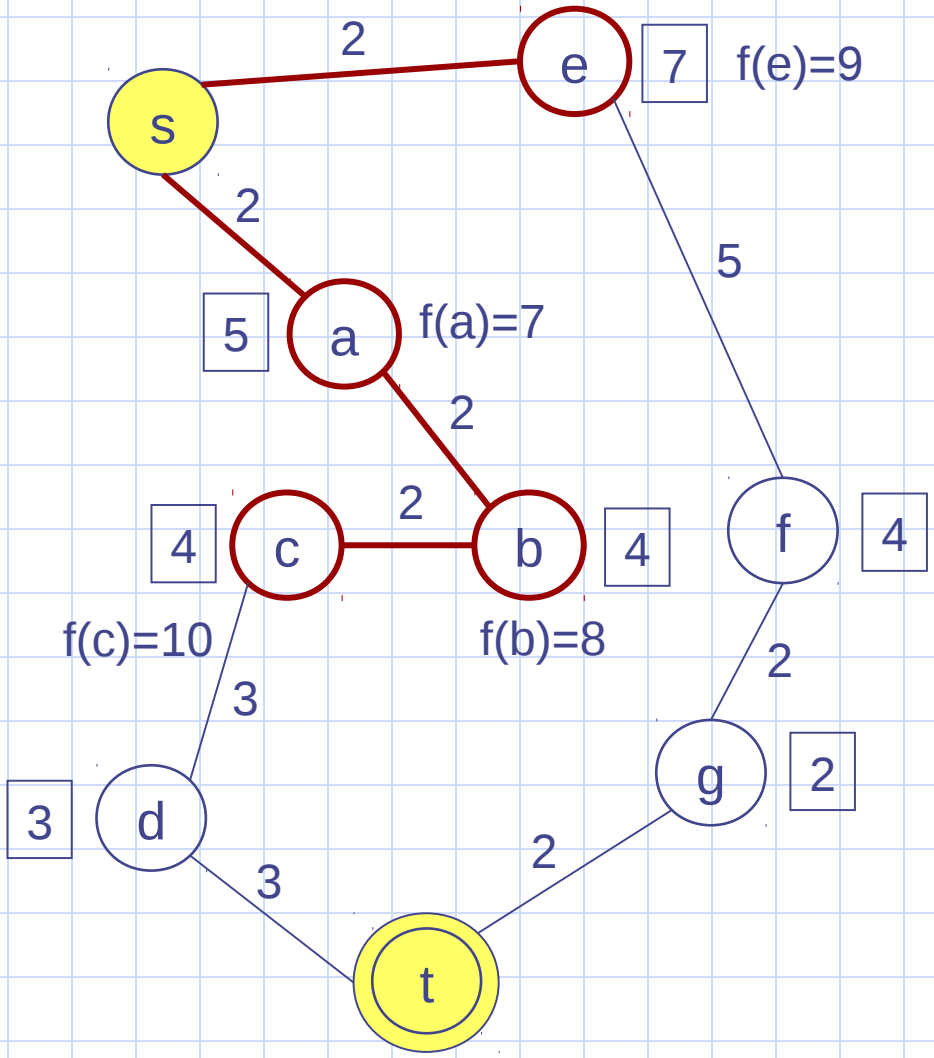
Playing with Wheels: A*



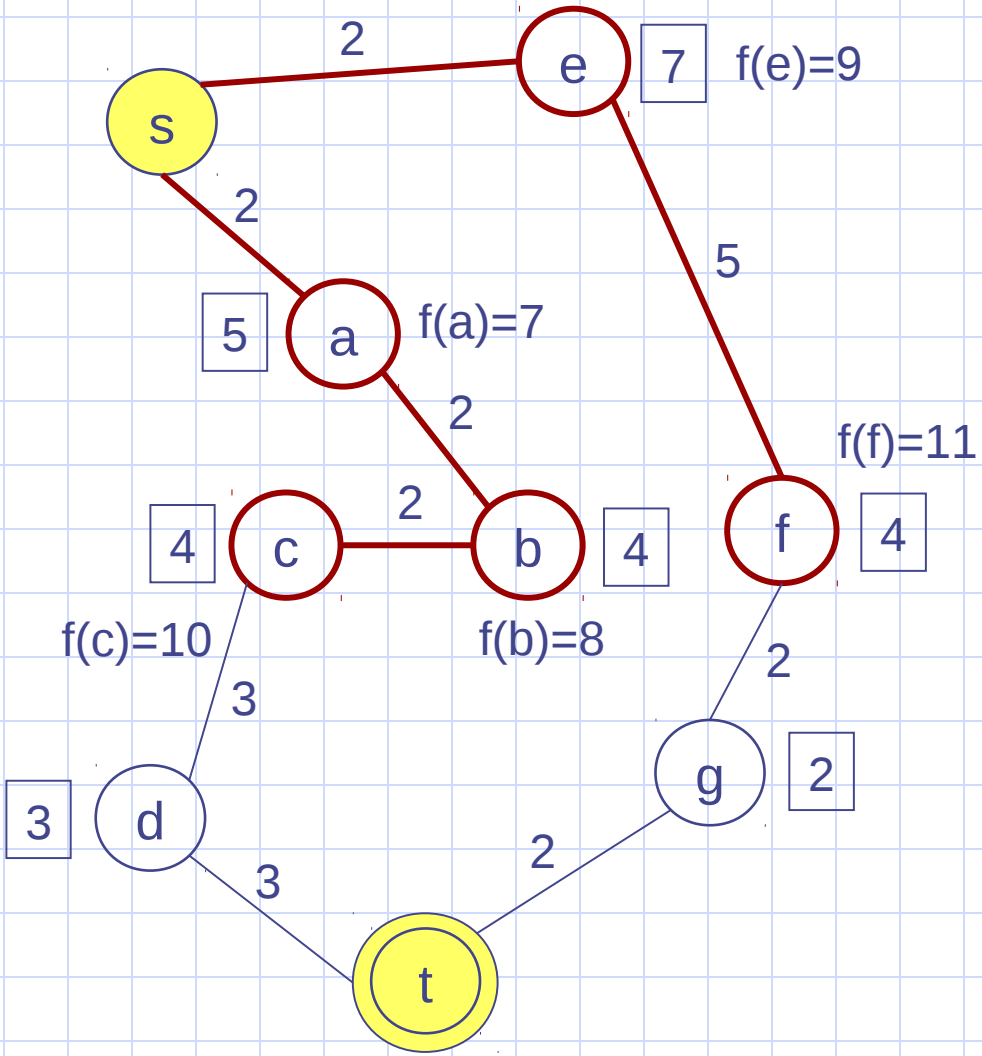
Playing with Wheels: A*



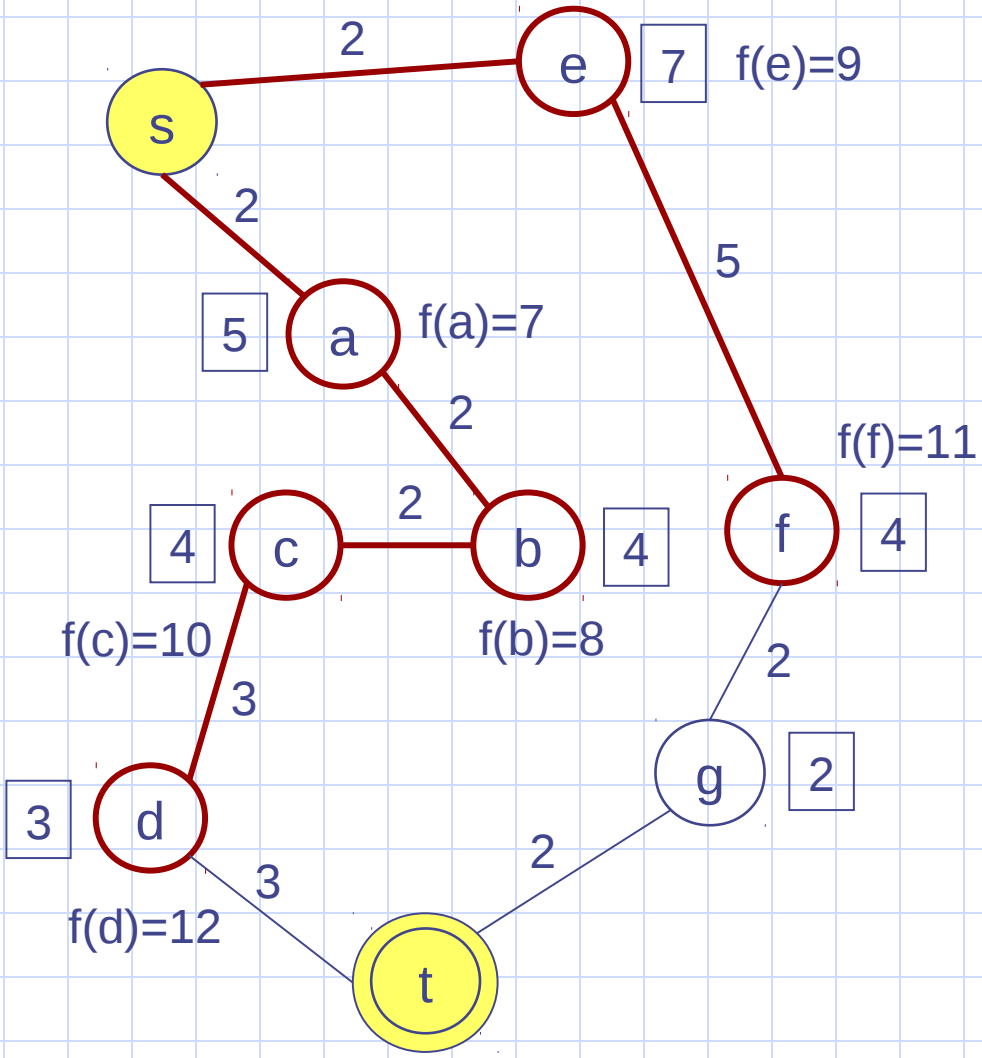
Playing with Wheels: A*



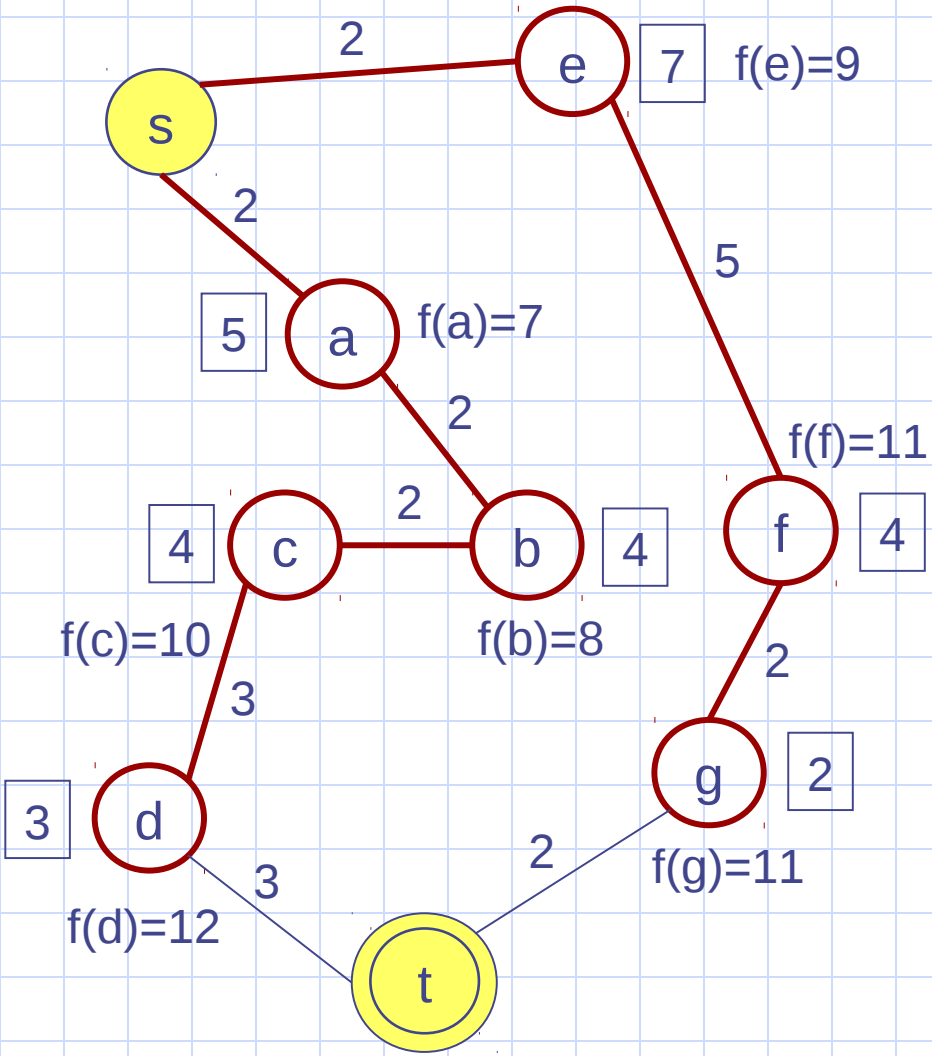
Playing with Wheels: A*



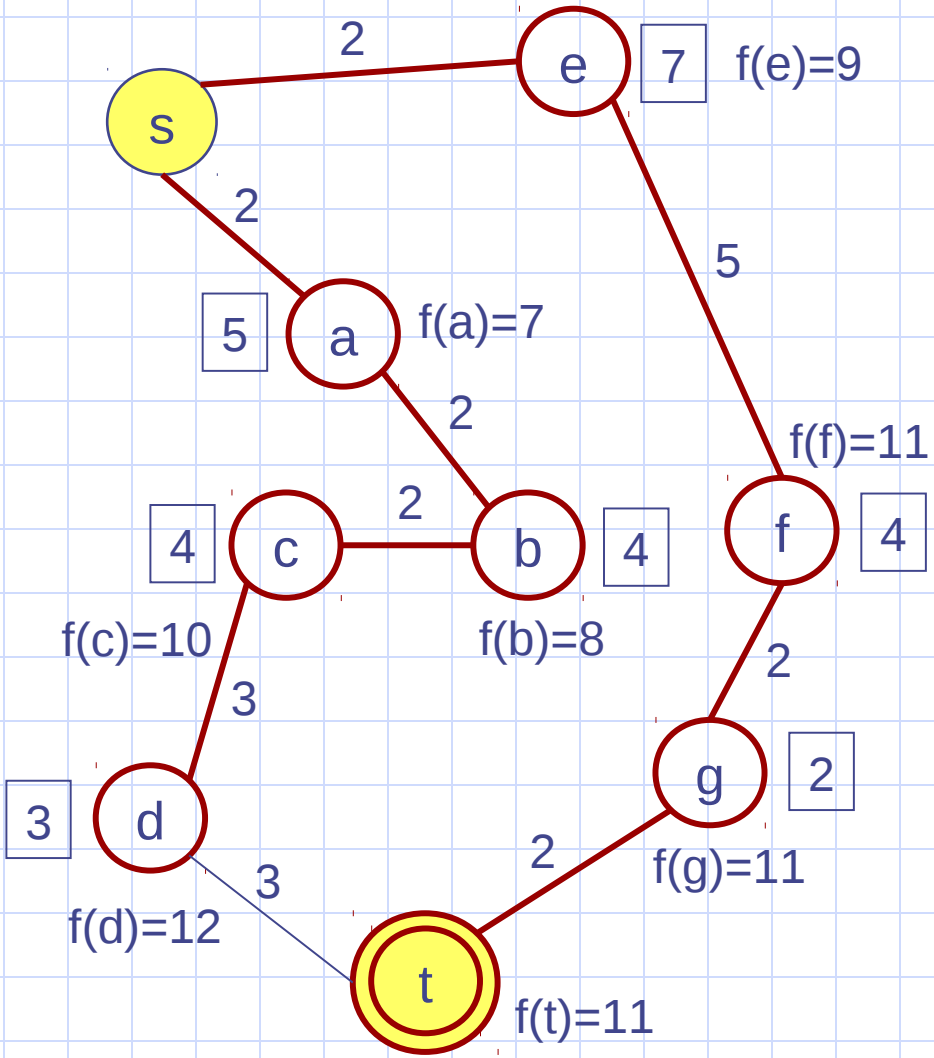
Playing with Wheels: A*



Playing with Wheels: A*



Playing with Wheels: A*



Playing with Wheels: A*

- A* possui as seguintes características:
  - **Completa**: garantia de encontrar uma solução quando existe;
  - **Ótima**: encontra a melhor solução entre várias soluções não ótimas;
  - **Eficiência ótima**: nenhum outro algoritmo da mesma família expande um número menor de nós que o A*.
- Por outro lado, A* pode consumir muita memória.
  - Solução: IDA* e SMA* (Russel e Norvig, 2002).

Playing with Wheels: A*

- Para modificar a solução atual para A* é necessário:
  - Definir uma função heurística e uma função custo;
  - Transformar a fila em uma fila de prioridades, na qual os estados são ordenados pelos valores da função heurística + função de custo;
  - Utilizar como próximo estado a ser processado o estado de menor combinação heurística + custo, fornecido pela fila de prioridades.

Playing with Wheels: A*

```
struct state {
    char digit[4];
    int depth;
    int f;
    bool operator<( const state &a ) const {
        return f > a.f;
    }
};
```

Playing with Wheels: A*

```
int heuristic_cost(state s, state t) {
    int i, smaller, bigger, h = 0;

    for (i=0; i < 4; i++) {
        if (s.digit[i] > t.digit[i]) {
            smaller = t.digit[i];
            bigger = s.digit[i];
        } else {
            smaller = s.digit[i];
            bigger = t.digit[i];
        }
        h += min(bigger-smaller, 10-bigger+smaller);
    }
    return h+s.depth;
}
```

```
int A_star(state current, state final, char visited[10][10][10][10]) {
    state nexts[8];
    int i;
    priority_queue<state> q;

    if (!visited[current.digit[0]][current.digit[1]]
                [current.digit[2]][current.digit[3]]) {
        q.push(current);
        while (!q.empty()) {
            current = q.top(); q.pop();
            if (equal(current, final)) return current.depth;
            next_states(current, nexts);
            for (i = 0; i < 8; i++)
                if (visited[nexts[i].digit[0]][nexts[i].digit[1]]
                           [nexts[i].digit[2]][nexts[i].digit[3]]) {
                    visited[nexts[i].digit[0]][nexts[i].digit[1]]
                           [nexts[i].digit[2]][nexts[i].digit[3]] = 1;
                    nexts[i].f = heuristic_cost(nexts[i], final);
                    q.push(nexts[i]);
                }
        }
    }
    return -1;
}
```

Playing with Wheels

- A solução com busca em largura necessita de 0.973s para solucionar os casos do teste do UVA.
- O método A* e precisa de 0.430s para todos os casos de teste (posição 25 no *ranking* do UVA).

Problemas

- Hoje
  - UVa 291 (The House of Santa Claus)
- Casa próxima aula
  - UVa 439 (Knight Moves);
  - UVa 850 (Crypt Kicker II);
  - UVa 10422 (Knights in FEN).
- Opcional
  - UVa 861 (Little Bishops)

Referências

- Batista, G. & Campello, R.
  - Slides disciplina *Algoritmos Avançados*, ICMC-USP, 2007.
- Lenaerts, T.
  - Vlaams Interuniversitair Instituut voor Biotechnologie. Notas de aula, 2007.
- Skiena, S. S. & Revilla, M. A.
  - *Programming Challenges – The Programming Contest Training Manual*. Springer, 2003.

Referências

- 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