

Programação em Lógica com Restrições

Henrique Lopes Cardoso

hlc@fe.up.pt

DEI/FEUP

Novembro 2015

Baseado nos slides “Programação em Lógica com Restrições”, Luís Paulo Reis, FEUP, 2010

Conteúdo

1. Introdução à Programação em Lógica com Restrições
2. Exemplos de Problemas de Satisfação e Otimização
3. Exemplos de Resolução de Problemas em PLR
4. Complexidade e Métodos de Pesquisa
5. Restrições em Booleanos, Reais e Racionais
6. Definições Formais e Conceitos
7. Manutenção de Consistência
8. Pesquisa, Otimização e Eficiência
9. Sistemas de PLR
10. Conclusões e Leitura Adicional



Programação em Lógica com Restrições

1. INTRODUÇÃO À PROGRAMAÇÃO EM LÓGICA COM RESTRIÇÕES

Bibliografia base:

Edward Tsang, *Foundations of Constraint Satisfaction*, Academic Press, London and San Diego, 1993

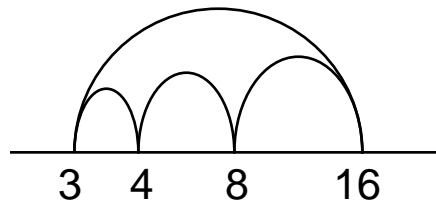
Kim Marriott e Peter J. Stuckey. *Programming with Constraints: an Introduction*, MIT Press, 1998

Programação em Lógica com Restrições

- A Programação em Lógica com Restrições – **PLR**, ou **CLP** (*Constraint Logic Programming*) – é uma classe de linguagens de programação combinando:
 - **declaratividade** da programação em lógica
 - **eficiência** da resolução de restrições
- Aplicações principais na resolução de **problemas de pesquisa/otimização combinatória** (tipicamente NP-completos):
 - escalonamento (*scheduling*), geração de horários (*timetabling*), alocação de recursos, planeamento, gestão da produção, verificação de circuitos, ...
- Vantagens:
 - Reduzido tempo de desenvolvimento
 - Facilidade de manutenção
 - Eficiência na Resolução
 - Clareza e brevidade dos programas

Mecanismo Constrain and Generate

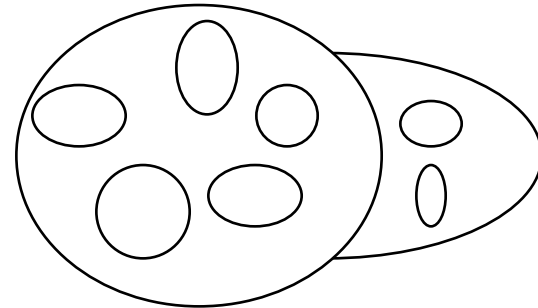
- O mecanismo de **unificação** do Prolog é substituído por um mecanismo de **manipulação de restrições** num dado domínio
- A pesquisa do tipo “**generate and test**” do Prolog (pouco eficiente) é substituída por técnicas mais inteligentes de pesquisa (técnicas de consistência), resultando num mecanismo do tipo “**constrain and generate**”



Variáveis Lógicas:

$$x = 3$$

$$y = 8$$



Domínios das variáveis:

$$x :: 1..16$$

$$y :: 1..10$$

Problema de Satisfação de Restrições

- Um Problema de Satisfação de Restrições – **PSR**, ou **CSP** (*Constraint Satisfaction Problem*) é modelizado através de:
 - **variáveis** representando diferentes aspectos do problema, juntamente com os seus **domínios**
 - **restrições** que limitam os valores que as variáveis podem tomar dentro dos seus domínios
- Porquê representar problemas como CSPs?
 - Representação muito próxima da original
 - Algoritmos para resolver CSPs são muito eficientes
 - Clareza e brevidade dos programas
 - Maior garantia de correção dos programas
 - Rapidez no desenvolvimento de programas

Solução de um CSP

- A **solução** de um CSP é uma atribuição de um valor (do seu domínio) a cada variável, de forma a que todas as restrições sejam satisfeitas
- Podemos estar interessados em:
 - Descobrir **se** o problema tem solução
 - Encontrar unicamente **uma** solução
 - Encontrar **todas** as soluções do problema
 - Encontrar uma solução **ótima**, de acordo com uma função objetivo, definida em termos de algumas ou de todas as variáveis
- A solução de um CSP é encontrada através de uma **pesquisa** sistemática, usualmente guiada por uma heurística, através de todas as atribuições possíveis de valores às variáveis

Definição Formal de um CSP

- Um CSP é um tuplo $\langle V, D, C \rangle$:
 - $V = \{x_1, x_2, \dots, x_n\}$ é o conjunto de **variáveis de domínio**
 - D é uma função que mapeia cada variável de V num conjunto de valores – os **domínios** das variáveis
 - $D : V \rightarrow$ conjunto finito de valores
 - Dx_i : domínio de x_i
 - $C = \{C_1, C_2, \dots, C_m\}$ é o conjunto de **restrições** que afetam um subconjunto arbitrário de variáveis de V
- Solução de um CSP: atribuição de um valor a cada variável de forma a que todas as restrições sejam respeitadas:
 - $Sol = \{\langle x_1, v_1 \rangle, \dots, \langle x_n, v_n \rangle\}$:
 - $\forall x_i \in V (\langle x_i, v_i \rangle \in Sol \wedge v_i \in Dx_i)$
 - $\forall c_k \in C \text{ Satisfaz}(Sol, c_k)$

Restrições em CSPs

- Tipos de Restrições:
 - Unárias
 - $X > 10$
 - $X \in \{1, 2, 5\}$
 - Binárias
 - $X < Y$
 - $Y + 20 \geq X$
 - Qualquer restrição de aridade superior pode ser transformada num conjunto de restrições binárias
 - Logo, os CSPs binários podem ser considerados como representativos de todos os CSPs
- Domínios:
 - Números complexos
 - Reais
 - Racionais
 - Inteiros
 - Booleanos
- Tipos Simples de Restrições:
 - Lineares
 - $2 * X + 4 < 4 * Y + Z$
 - Não Lineares
 - $W * Z + 3 * X > Y - Z * Z$

Resolução de um CSP

- Passos:
 1. Declarar as **variáveis** e os seus **domínios** finitos
 2. Especificar as **restrições**
 3. **Pesquisar** para encontrar a solução
- Técnicas utilizadas:
 - Determinísticas: técnicas de **consistência**
 - Não determinísticas: **pesquisa** (“search”)
- Forma de resolução:
 - Computação determinística é efectuada sempre que possível (durante a propagação)
 - Pesquisa é efectuada quando não é possível efetuar mais propagação

Introdução às Restrições:

Propagação e Consistência

- Se existir uma restrição binária C_{ij} entre as variáveis x_i e x_j , então o arco (x_i, x_j) é **consistente** se para cada valor $a \in Dx_i$, existe um valor $b \in Dx_j$, tal que as atribuições $x_i=a$ e $x_j=b$ satisfaçam a restrição C_{ij}

- Qualquer valor $a \in Dx_i$ para o qual isto não se verifique pode ser removido de Dx_i (pois não pode fazer parte de nenhuma solução consistente)

- A remoção de todos estes valores torna o arco (x_i, x_j) consistente

- Se todos os arcos de um CSP binário forem tornados consistentes, então todo o problema é **consistente nos arcos**

$$\begin{array}{ccc} x & \xrightarrow{x+2 < y} & y \\ \{1..5\} & & \{1..5\} \end{array}$$

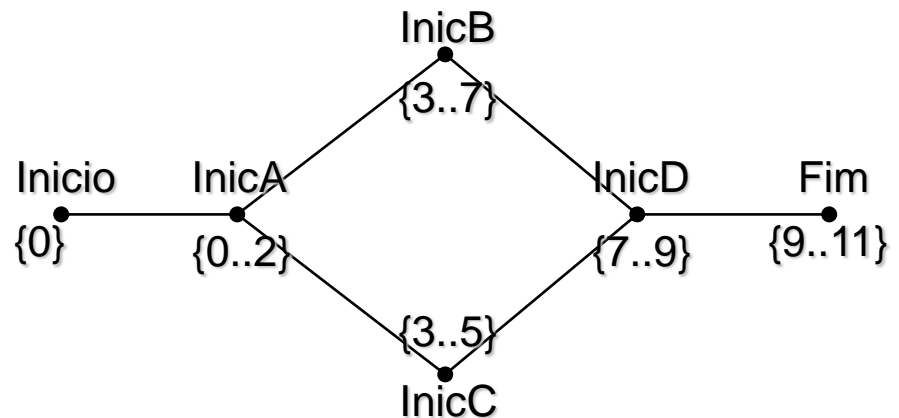
$$\begin{array}{ccc} x & \xrightarrow{x+2 < y} & y \\ \{1,2\} & & \{1..5\} \end{array}$$

$$\begin{array}{ccc} x & \xrightarrow{x+2 < y} & y \\ \{1,2\} & & \{4,5\} \end{array}$$

Introdução às Restrições: Propagação e Consistência

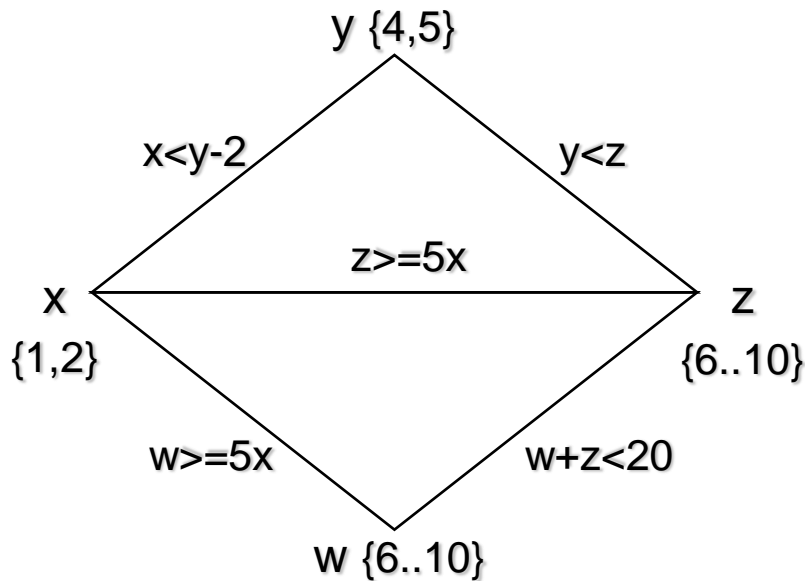
Tarefa	Duração	Precede
A	3	B,C
B	2	D
C	4	D
D	2	

Variável	Domínio
Inicio	{0}
InicA	{0..11}
InicB	{0..11}
InicC	{0..11}
InicD	{0..11}
Fim	{0..11}

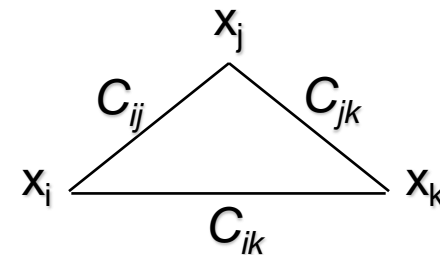


Introdução às Restrições:

Propagação e Consistência



- Problema é consistente nos arcos, mas será que é possível $x=2$?
- Próximo passo: considerar triplos de variáveis em que dois pares de variáveis têm uma restrição não trivial entre eles



- Caminho (x_i, x_j, x_k) é **consistente nos caminhos** se para cada par de valores $v_i \in Dx_i$ e $v_k \in Dx_k$, permitidos pela restrição C_{ik} , existe um valor $v_j \in Dx_j$ tal que $(v_i, v_j) \in C_{ij}$ e $(v_j, v_k) \in C_{jk}$
- Se não existir um tal valor então (v_i, v_k) deve ser removido da restrição C_{ik}

Programação em Lógica com Restrições

2. EXEMPLOS DE PROBLEMAS DE SATISFAÇÃO E OTIMIZAÇÃO

Adaptado de:

Pedro Barahona, *Página da Disciplina de Programação por Restrições, Ano Lectivo 2003/04*, disponível em:

<http://ssdi.di.fct.unl.pt/~pb/cadeiras/pr/0304/index.htm>

[consultado em Setembro de 2004]

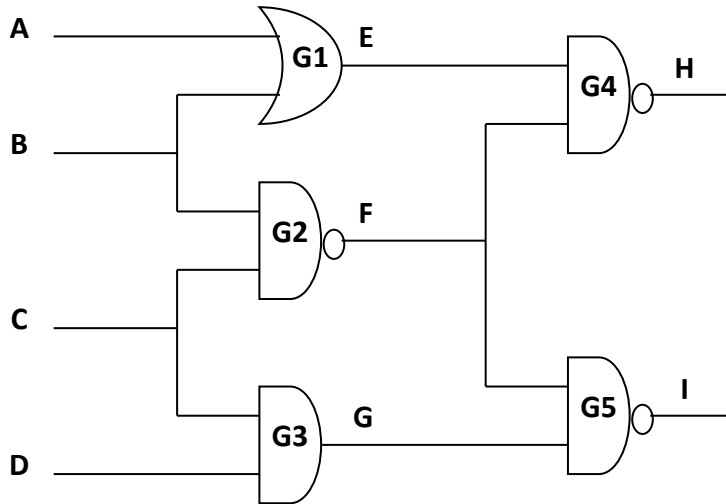
Restrições: Exemplos de Problemas

- Planeamento de Testes em Circuitos Digitais
 - Detecção de Avarias
- Gestão de Tráfego em Redes
- Gestão da Produção
- Escalonamento de Tarefas (*scheduling*)
- Geração de Horários (*timetabling*)
- Caixeiro Viajante
 - Simples ou Múltiplo
- Colocação de Objetos
 - 2D: Corte de peças (tecido, vidro, madeira, etc.)
 - 3D: Preenchimento de contentores / Empacotamento

Circuitos Digitais

- Planeamento de testes para deteção de avarias
- **Objetivo:**
 - Determinar um padrão de entrada que permita detetar se uma “gate” está avariada
- **Variáveis:**
 - Modelizam o valor do nível elétrico (high/low) nos vários fios do circuito
- **Domínios:**
 - Variáveis Booleanas: 0/1.
- **Restrições:**
 - Restrições de igualdade entre o sinal de saída e o esperado pelo funcionamento das “gates”

Circuitos Digitais



- $E = A \oplus B \oplus A \cdot B$ % $E = \text{or}(A, B)$
- $F = 1 \oplus B \cdot C$ % $F = \text{nand}(B, C)$
- $G = C \cdot D$ % $G = \text{and}(C, D)$
- $H = 1 \oplus E \cdot F$ % $H = \text{nand}(E, F)$
- $I = 1 \oplus F \cdot G$ % $I = \text{nand}(F, G)$

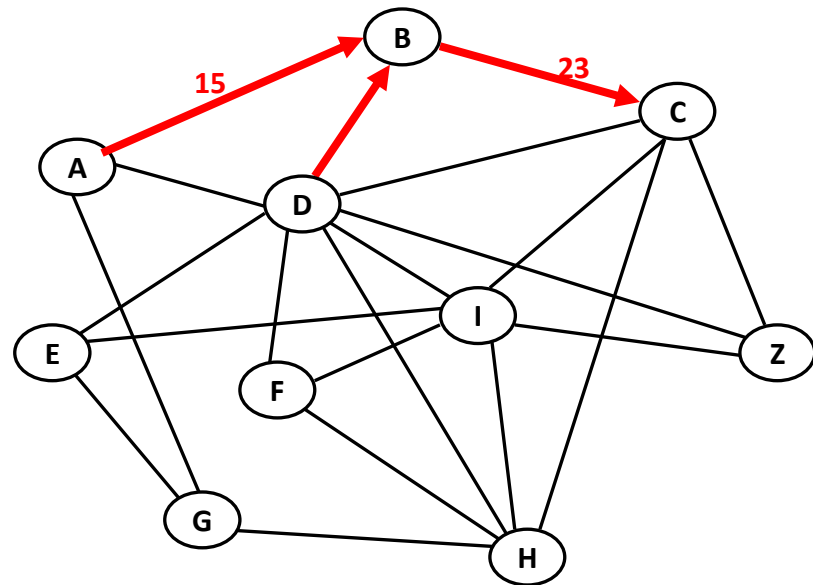
(\oplus representa *xor*; \cdot representa *and*)

Gestão de Tráfego em Redes

- **Determinação do tráfego em redes** de telecomunicações, transporte de água/eletricidade, ...
- **Objetivo:**
 - Determinar a quantidade de informação que flui em cada um dos troços de uma rede de comunicações
- **Variáveis:**
 - Modelam o fluxo nos diversos troços
- **Domínio:**
 - Reais não negativos
- **Restrições:**
 - Limites de capacidade dos troços, não perda de informação nos vários nós

Gestão de Tráfego em Redes

- X_{ij} é o fluxo entre os nós i e j
 - Restrições de capacidade
$$X_{ab} \leq 15$$
 - Restrições de manutenção de informação
$$X_{ab} + X_{db} = X_{bc}$$



Gestão da Produção

- Determinação das quantidades ideais de itens a produzir
- **Objetivo:**
 - Determinar as quantidades de itens que devem ser produzidos
- **Variáveis:**
 - Modelizam o número de exemplares de cada produto
- **Domínio:**
 - Inteiros / Reais não negativos
- **Restrições:**
 - Limites dos recursos existentes, restrições sobre o plano produzido

Gestão da Produção

- Limites dos recursos existentes
 - R_i é a quantidade de unidades do recurso i

$$a_{i1} X_1 + a_{i2} X_2 + a_{i3} X_3 + \dots \leq R_i$$

- a_{ij} é a quantidade do recurso i necessária para produzir uma unidade do produto j
- X_j é a quantidade do produto j produzida

- Restrições sobre o plano produzido
 - Objetivos mínimos de produção

$$X_1 + X_2 \geq 50$$

- Equilíbrio na produção

$$| X_4 - X_5 | < 20$$

Escalonamento de Tarefas

- **Determinação do escalonamento de um conjunto de tarefas** no tempo (eventualmente espaço)
- **Objetivo:**
 - Determinar os tempos de início de um conjunto de tarefas (eventualmente os recursos utilizados também)
- **Variáveis:**
 - Modelam o início da execução das tarefas
- **Domínio:**
 - Inteiros/Reais não negativos
- **Restrições:**
 - Não sobreposição de tarefas que utilizam os mesmos recursos
 - Precedências de tarefas

Escalonamento de Tarefas

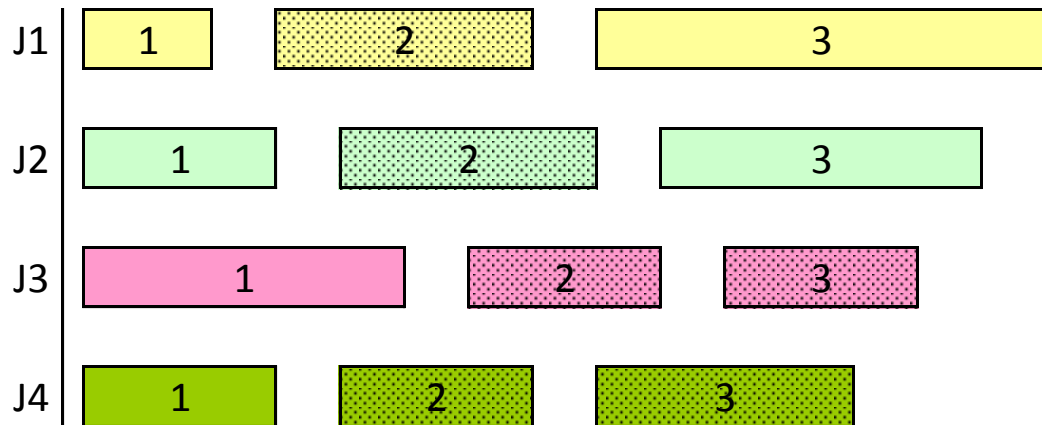
- Job-Shop

- $S_{ij} / D_{ij} / M_{ij}$: início/duração/máquina da tarefa i para o trabalho j
- Precedência entre tarefas

$$S_{ij} + D_{ij} \leq S_{kj} \quad \text{para } i < k$$

- Não sobreposição de tarefas na mesma máquina

$$(M_{ij} \neq M_{kl}) \vee (S_{ij} + D_{ij} \leq S_{kl}) \vee (S_{kl} + D_{kl} \leq S_{ij})$$



Geração de Horários

- **Especificação dos horários** (e.g. escolares ou universitários) de um conjunto de eventos/tarefas
- **Objetivo:**
 - Determinar o início das várias aulas/eventos num horário
- **Variáveis:**
 - Modelam o tempo de início da aula, e eventualmente a sala, o professor, outros recursos, etc.
- **Domínio:**
 - Finitos (horas certas) para os tempos de início
 - Finitos para as salas/recursos/professores
- **Restrições:**
 - Não sobreposição de aulas na mesma sala , nem com o mesmo professor, respeito pelas impossibilidades, etc.

Caixeiro Viajante

- **Determinação de percursos** de caixeiros viajantes (frotas de carros, empresas de distribuição, etc.)
- **Objetivo:**
 - Determinar o melhor (mais curto/mais rápido) caminho para ser seguido pelos veículos de uma empresa de distribuição
- **Variáveis:**
 - Ordem em que cada localidade é atingida
- **Domínio:**
 - Finitos (número de localidades existentes)
- **Restrições:**
 - Não repetir localidades, garantir ciclo global, não existência de sub-ciclos

Empacotamento

- Colocação de componentes no espaço (2D/3D) sem sobreposição
- Objetivo:
 - Determinar formas de conseguir colocar componentes num dado espaço
- Variáveis:
 - Coordenadas (X,Y) dos elementos
 - Rotações/espelhamentos dos elementos
- Domínio:
 - Reais / Finitos (grelha)
- Restrições:
 - Não sobrepor componentes, não ultrapassar os limites do “contentor”

Empacotamento

- Não ultrapassar os limites do “contentor”

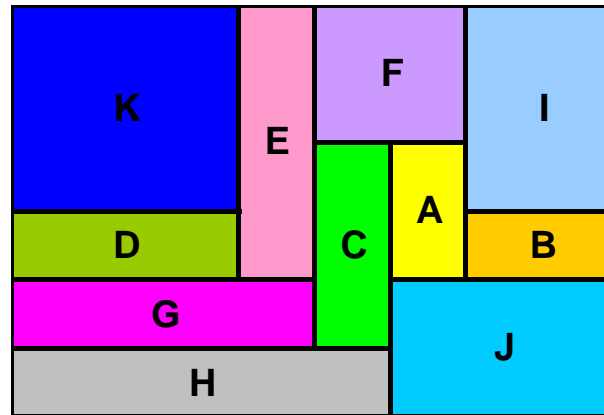
$$X_a + Lx_a \leq X_{\max}$$

$$Y_a + Ly_a \leq Y_{\max}$$

- Não sobrepor componentes

$$X_a + Lx_a \leq X_b \quad \vee \quad X_b + Lx_b \leq X_a$$

$$Y_a + Ly_a \leq Y_b \quad \vee \quad Y_b + Ly_b \leq Y_a$$



Decisão vs. Otimização

- Em certos (muitos) casos o que se pretende determinar não é uma solução qualquer, mas sim **a melhor solução** (segundo um dado critério)
- Um **problema de otimização** com restrições pode ser especificado por um tuplo $\langle V, D, C, F \rangle$ em que:
 - V, D, C tal como anteriormente
 - **F** : é uma função das soluções para um domínio ordenado

Otimização: Exemplos

- Planeamento de Testes em Circuitos Digitais
 - Teste com **menos** entradas especificadas
- Gestão de Tráfego em Redes
 - Tráfego com **menor** custo / mais curto
- Gestão da Produção
 - Plano com **maior** lucro / menores custos
- Escalonamento de Tarefas
 - Solução com o final **mais cedo**
- Geração de Horários
 - Solução com **menos** furos
 - Solução que **melhor** respeita preferências de horário
- Caixeiro Viajante
 - Solução com **menor** distância/custo percorrida
- Empacotamento ou Corte
 - Corte / Colocação do **maior** número de peças
 - **Menor** desperdício de material

Programação em Lógica com Restrições

3. EXEMPLOS DE RESOLUÇÃO DE PROBLEMAS EM PLR

Bibliografia base:

Luís Paulo Reis, *Caderno de Exercícios de Programação em Lógica com Restrições*, FEUP, Setembro de 2007

Exemplos Simples

- Alguns exemplos simples:
 - Criptograma $SEND + MORE = MONEY$
 - Soma e Produto
 - Quadrado Mágico
 - N-Rainhas
 - Zebra Puzzle
- Variáveis e Domínios?
- Restrições?
- Solução do problema?

Criptograma SEND+MORE=MONEY

- Formulação simples:

– Variáveis e domínios:

$S, E, N, D, M, O, R, Y \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

– Restrições:

$S \neq E \neq N \neq D \neq M \neq O \neq R \neq Y$

$S \neq 0, M \neq 0$

$$\begin{aligned} S*1000+E*100+N*10+D &+ M*1000+O*100+R*10+E \\ &= M*10000+O*1000+N*100+E*10+Y \end{aligned}$$

	S	E	N	D	
+	M	O	R	E	
<hr/>					
	M	O	N	E	Y

Criptograma SEND+MORE=MONEY

- Formulação mais eficiente:

- Variáveis e domínios:

$S, E, N, D, M, O, R, Y \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

$C_1, C_2, C_3, C_4 \in \{0,1\}$

- Restrições:

$S \neq E \neq N \neq D \neq M \neq O \neq R \neq Y$

$S \neq 0, M \neq 0$

$D + E = Y + C_1 * 10$

$N + R + C_1 = E + C_2 * 10$

$E + O + C_2 = N + C_3 * 10$

$S + M + C_3 = O + C_4 * 10$

$C_4 = M$

C_4	C_3	C_2	C_1	
	S	E	N	D
+	M	O	R	E
<hr/>				
M	O	N	E	Y

	1	0	1	1	
		9	5	6	7
+		1	0	8	5
<hr/>					
	1	0	6	5	2

Criptograma SEND+MORE=MONEY

- Solução simples:

```
:- use_module(library(clpfd)).
```

```
send(Vars) :-
```

```
    Vars=[S,E,N,D,M,O,R,Y],  
    domain(Vars,0,9),  
    all_different(Vars),  
    S #\= 0, M #\= 0,  
    S*1000+E*100+N*10+D +  
    M*1000+O*100+R*10+E #=  
    M*10000+O*1000+N*100+E*10+Y,  
    labeling([],Vars).
```

- Solução mais eficiente:

```
:- use_module(library(clpfd)).
```

```
send(Vars) :-
```

```
    Vars=[S,E,N,D,M,O,R,Y],  
    domain(Vars,0,9),  
    domain([C1,C2,C3,C4],0,1),  
    all_distinct(Vars),  
    S #\= 0, M #\= 0,  
    D + E #= Y + C1*10,  
    N + R + C1 #= E + C2*10,  
    E + O + C2 #= N + C3*10,  
    S + M + C3 #= O + C4*10,  
    C4 #= M,  
    labeling([ff],Vars).
```

Soma e Produto

- Que conjuntos de três números têm a sua soma igual ao seu produto?
- Solução:

```
somaproduct(A,B,C) :-  
    domain([A,B,C],1,1000),  
    A*B*C #= A+B+C,  
    % C#>=B, B#>=A,      % eliminar simetrias  
    labeling([], [A,B,C]).
```

Quadrado Mágico NxN

- Preencher um quadrado com NxN casas, com números únicos entre 1 e NxN de forma a que a soma dos elementos de cada linha, coluna ou diagonal principal seja sempre a mesma

- Solução 3x3:

```
magic(Vars) :-
```

```
Vars=[A1,A2,A3,A4,A5,A6,A7,A8,A9],
```

```
domain(Vars,1,9),
```

```
% Soma is (9+1)*3//2, % aumenta a eficiência
```

```
all_distinct(Vars),
```

```
A1+A2+A3 #= Soma,      A4+A5+A6 #= Soma,      A7+A8+A9 #= Soma,
```

```
A1+A4+A7 #= Soma,      A2+A5+A8 #= Soma,      A3+A6+A9 #= Soma,
```

```
A1+A5+A9 #= Soma,      A3+A5+A7 #= Soma,
```

```
% A1 #< A2, A1 #< A3, A1 #< A4, A2 #< A4, % eliminar simetrias
```

```
labeling([],Vars).
```

8	3	4
1	5	9
6	7	2

N-Rainhas

- Colocar, num tabuleiro com $N \times N$ casas, N rainhas (de xadrez), sem que nenhuma rainha ataque nenhuma outra (não podem estar na mesma linha horizontal, vertical ou diagonal)
- Solução 4-Rainhas

```
four_queens(Cols) :-  
    Cols=[A1,A2,A3,A4],  
    domain(Cols,1,4),  
    all_distinct(Cols),      % A1#\=A2, A1#\A3, A1#\A4, A2#\A3, A2#\A4, A3#\A4,  
    A1 #\= A2+1, A1 #\= A2-1,  
    A1 #\= A3+2, A1 #\= A3-2,  
    A1 #\= A4+3, A1 #\= A4-3,  
    A2 #\= A3+1, A2 #\= A3-1,  
    A2 #\= A4+2, A2 #\= A4-2,  
    A3 #\= A4+1, A3 #\= A4-1,  
    labeling([],Cols).
```

N-Rainhas

- Solução N-Rainhas

```
nqueens(N, Cols) :-  
    length(Cols, N),  
    domain(Cols, 1, N),  
    constrain(Cols),  
    % all_distinct(Cols), % redundante mas diminui tempo  
    labeling([], Cols).  
  
constrain([]).  
constrain([H|RCols]) :- safe(H, RCols, 1), constrain(RCols).  
  
safe(_, [], _).  
safe(X, [Y|T], K) :- noattack(X, Y, K), K1 is K + 1, safe(X, T, K1).  
  
noattack(X, Y, K) :- X #\= Y, X + K #\= Y, X - K #\= Y.
```

Zebra Puzzle

- Há cinco casas com cinco cores diferentes. Em cada casa, vive uma pessoa de nacionalidade diferente, tendo uma bebida, uma marca de cigarros e um animal favoritos. Pistas:
 - O Inglês vive na casa vermelha
 - O Espanhol tem um cão
 - O Norueguês vive na primeira casa a contar da esquerda
 - Na casa amarela, o dono gosta de Marlboro
 - Quem fuma Chesterfields vive na casa ao lado do homem que tem uma raposa
 - O Norueguês vive ao lado da casa Azul
 - O homem que fuma Winston tem uma iguana
 - O fumador de Lucky Strike bebe sumo de laranja
 - O Ucrâniano bebe chá
 - O Português fuma SG Lights
 - Fuma-se Marlboro na casa ao lado da casa onde há um cavalo
 - Na casa verde, a bebida preferida é o café
 - A casa verde é imediatamente à direita (à sua direita) da casa branca
 - Bebe-se leite na casa do meio
- A pergunta é: Onde vive a Zebra, e em que casa se bebe água?

Zebra Puzzle

```
zebra(Zeb,Agu) :-  
    Nac = [Ing, Esp, Nor, Ucr, Por],  
    Ani = [Cao, Rap, Igu, Cav, Zeb],  
    Beb = [Sum, Cha, Caf, Lei, Agu],  
    Cor = [Verm, Verd, Bran, Amar, Azul],  
    Tab = [Che, Win, LS, SG, Mar],  
    append([Nac,Ani,Beb,Cor,Tab], List),  
    %  
    domain(List,1,5),  
    all_different(Nac), all_different(Ani), all_different(Beb),  
    all_different(Cor), all_different(Tab),  
    Ing #= Verm, Esp #= Cao, Nor #= 1, Amar #= Mar,  
    abs(Che-Rap) #= 1, abs(Nor-Azul) #= 1,  
    Win #= Igu, LS #= Sum, Ucr #= Cha, Por #= SG,  
    abs(Mar-Cav) #= 1,  
    Verd #= Caf, Verd #= Bran+1, Lei #= 3,  
    %  
    labeling([],List).
```


Programação em Lógica com Restrições

4. COMPLEXIDADE E MÉTODOS DE PESQUISA

Adaptado de:

Pedro Barahona, *Página da Disciplina de Programação por Restrições, Ano Lectivo 2003/04*, disponível em:

<http://ssdi.di.fct.unl.pt/~pb/cadeiras/pr/0304/index.htm>

[consultado em Setembro de 2004]

Restrições: Complexidade

- A dificuldade em resolver PSRs reside na sua **complexidade exponencial**
- **Domínio Booleano**
 - Número de variáveis: n
 - Dimensão do Domínio: 2
 - Espaço de Pesquisa: 2^n
- **Domínios Finitos**
 - Número de variáveis: n
 - Dimensão do Domínio: k
 - Espaço de Pesquisa: k^n
- **Domínios Racionais / Reais**
 - Em teoria, infinitas soluções potenciais
 - Na prática, número finito limitado à precisão utilizada
 - Métodos diferentes dos domínios finitos

Restrições: Complexidade

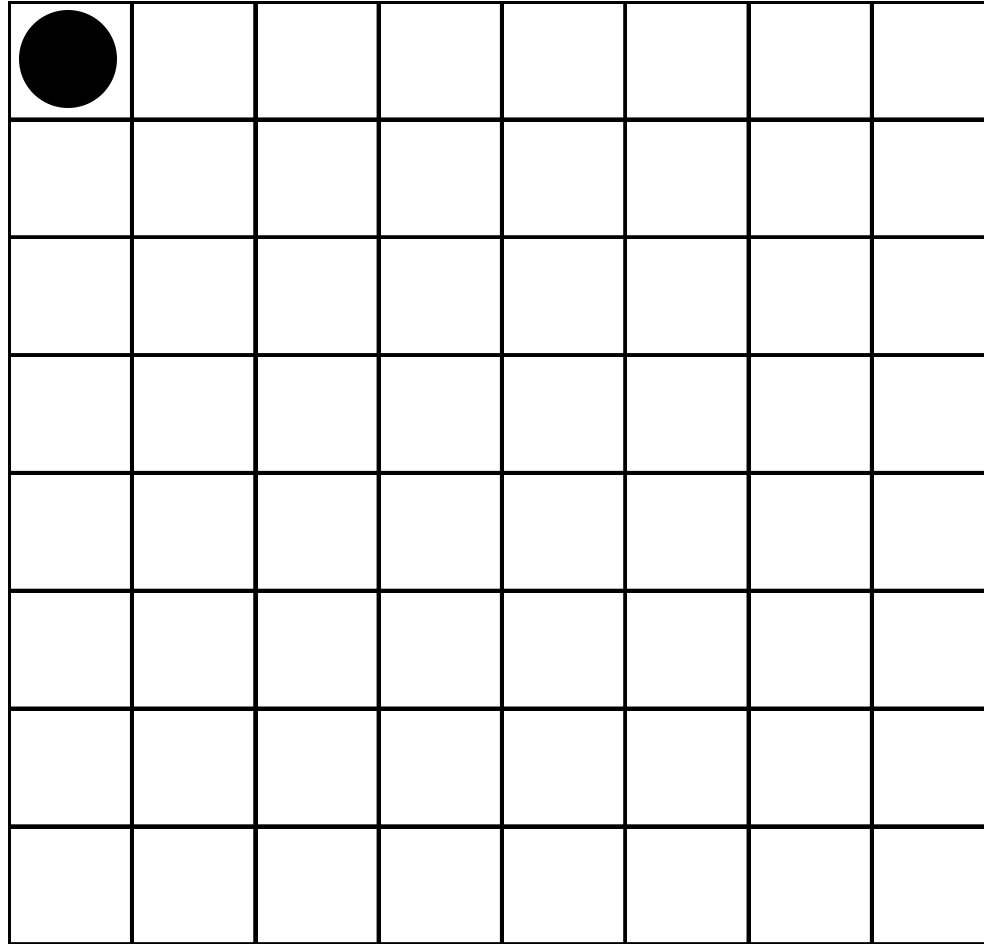
- Complexidade cresce exponencialmente
- Os problemas de interesse são geralmente NP-completos
- Tempo de pesquisa exaustiva das k^n possíveis soluções:
(assumindo duração de 1 μ s para cada operação elementar)

k n	10	20	30	40	50	60
2	1 mseg	1 seg	18 min	12,7 dias	35,7 anos	365 séculos
3	50 mseg	1 hora	6,5 anos	3855 séculos		
4	1 seg	12,6 dias	365 séculos			
5	9,8 seg	1103 dias	295 Kséculos			
6	1 min	116 anos				

Restrições: Construção vs. Reparação

- Construção:
 - Retrocesso: *“generate and test”*
 - Propagação: *“forward checking”*
- Comparação com algoritmos reparativos
- Exemplo: N-Rainhas, comprovando a eficiência do mecanismo *“forward checking”*

Retrocesso

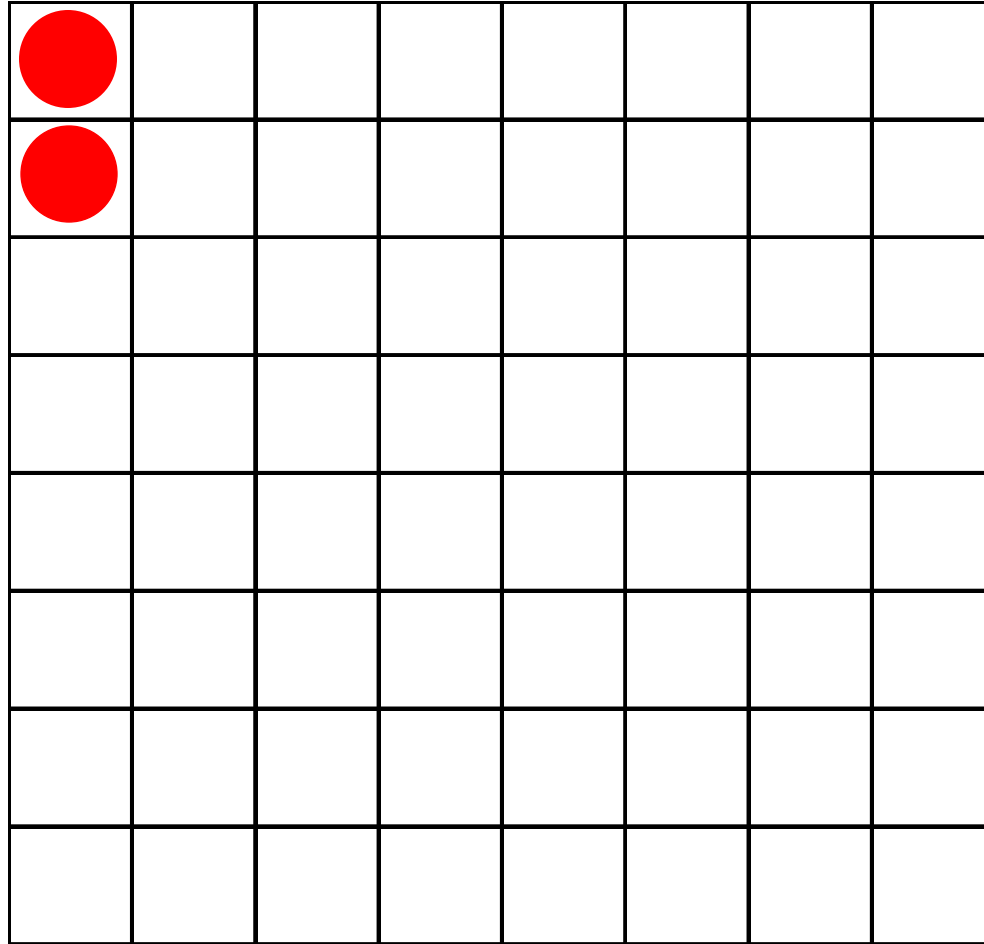


Testes 0

Retrocessos 0

Retrocesso

$Q1 \neq Q2, \quad L1+Q1 \neq L2+Q2, \quad L1+Q2 \neq L2+Q1.$

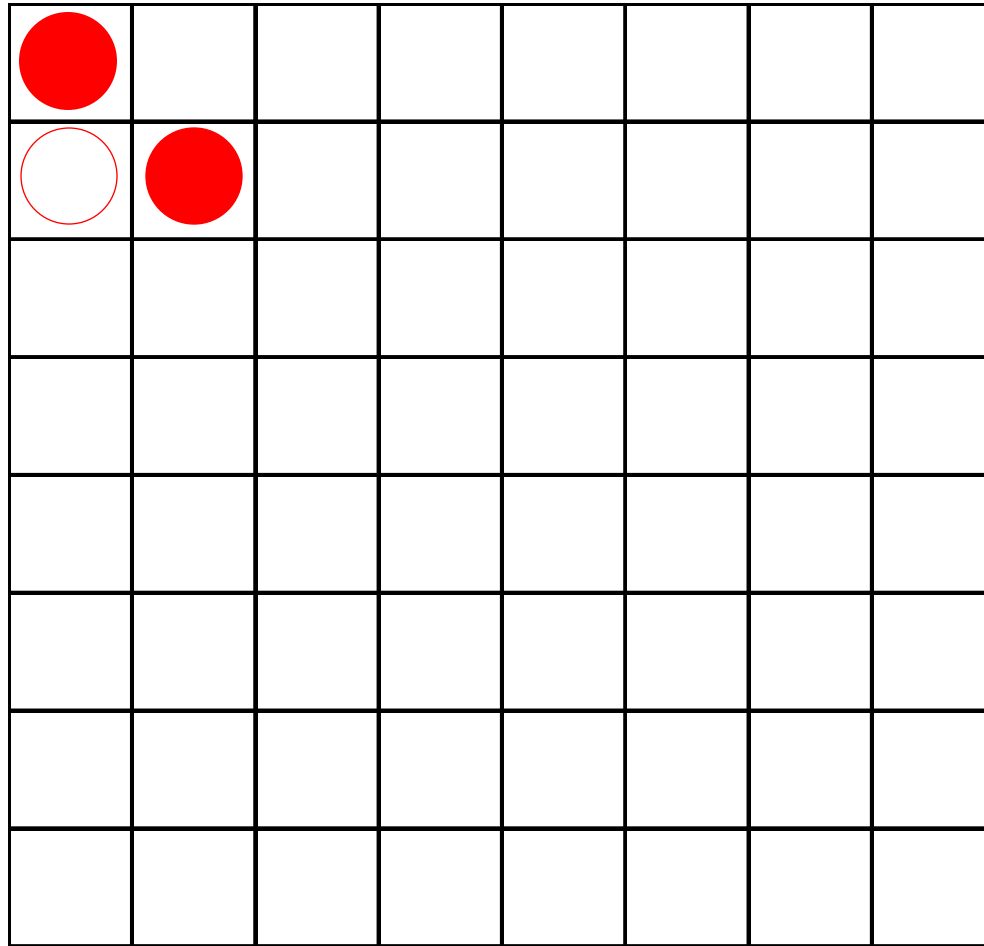


Testes $0 + 1 = 1$

Retrocessos 0

Retrocesso

$Q1 \neq Q2$, $L1+Q1 \neq L2+Q2$, $L1+Q2 \neq L2+Q1$.

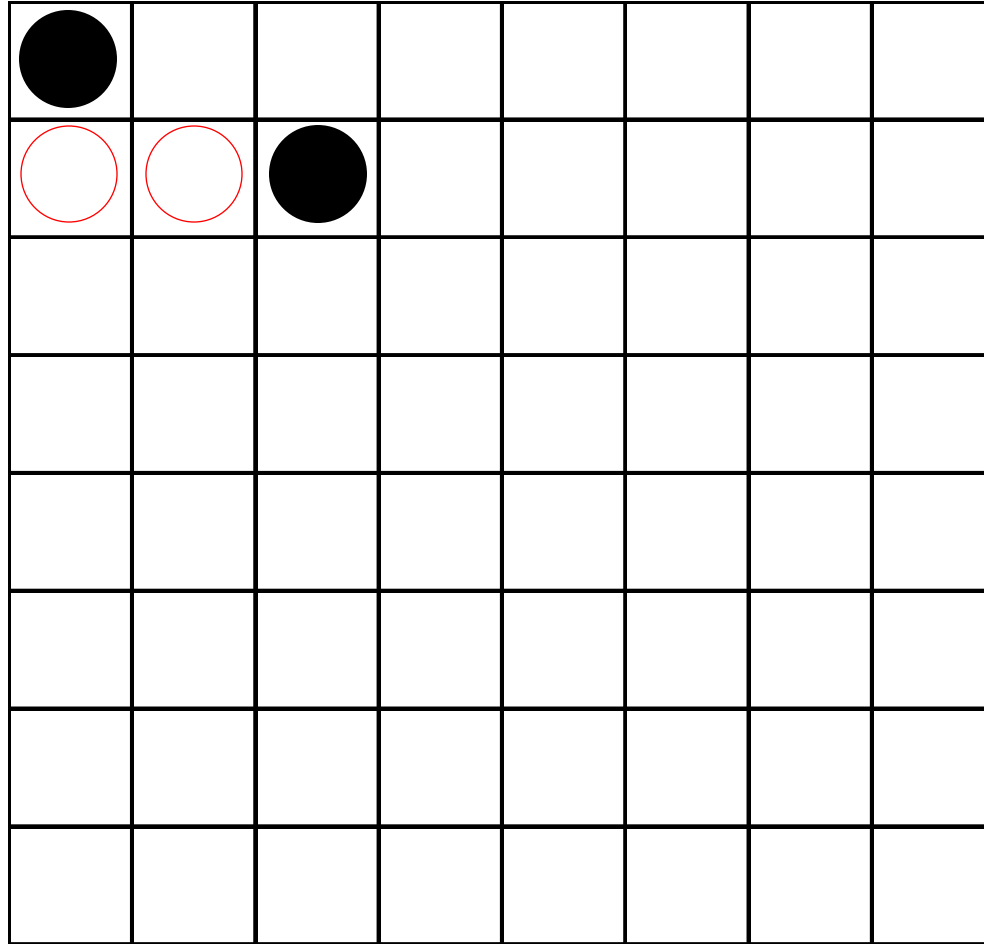


Testes $1 + 1 = 2$

Retrocessos 0

Retrocesso

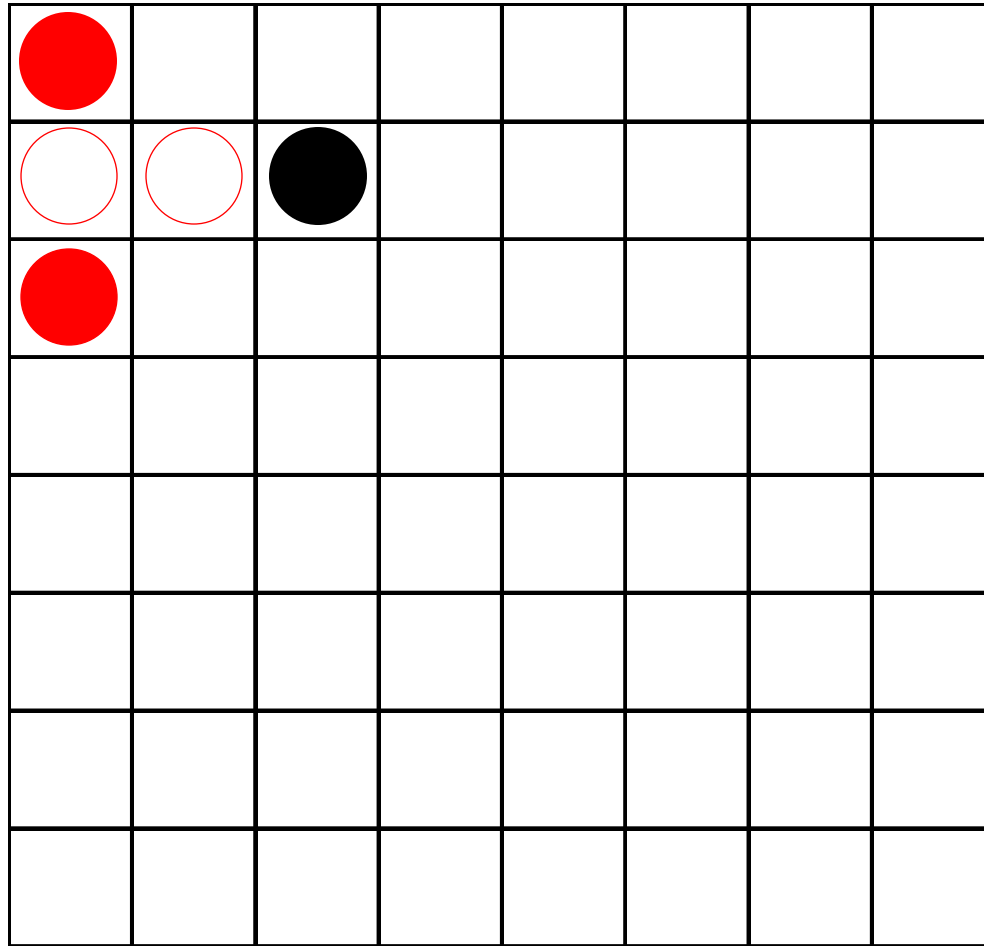
$Q1 \neq Q2, \quad L1+Q1 \neq L2+Q2, \quad L1+Q2 \neq L2+Q1.$



Testes $2 + 1 = 3$

Retrocessos 0

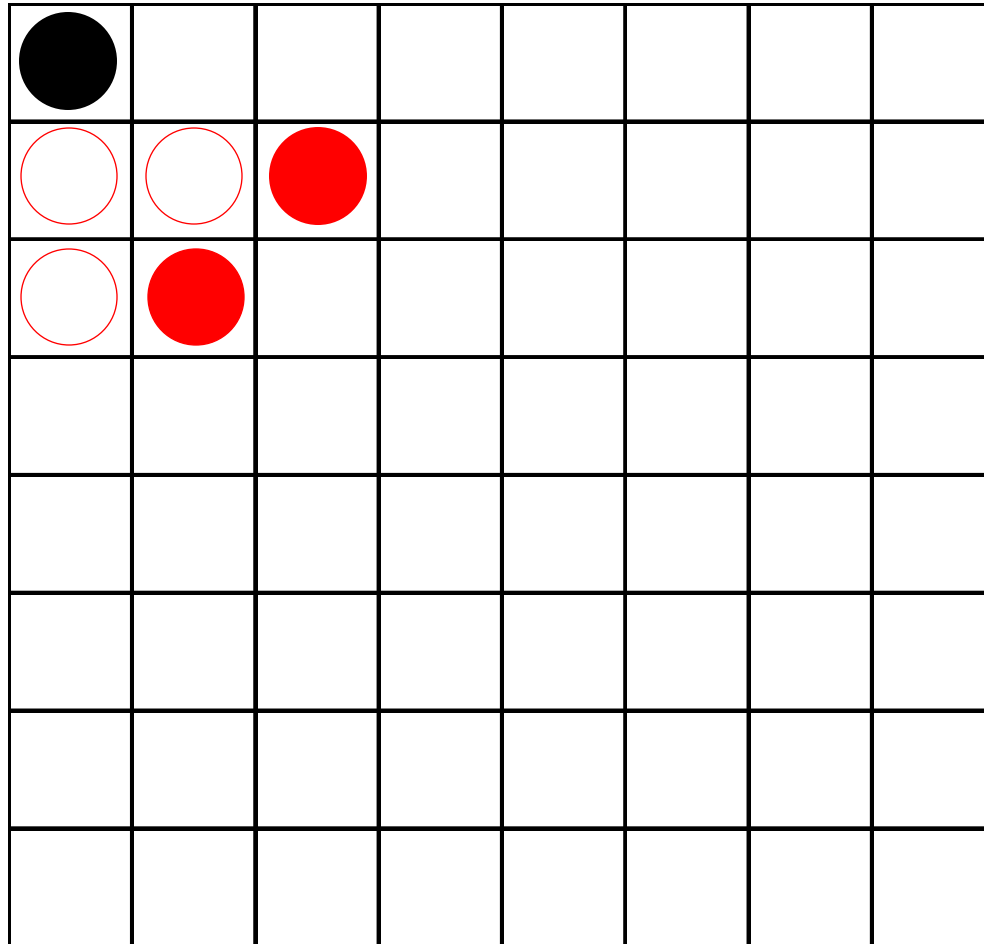
Retrocesso



Testes $3 + 1 = 4$

Retrocessos 0

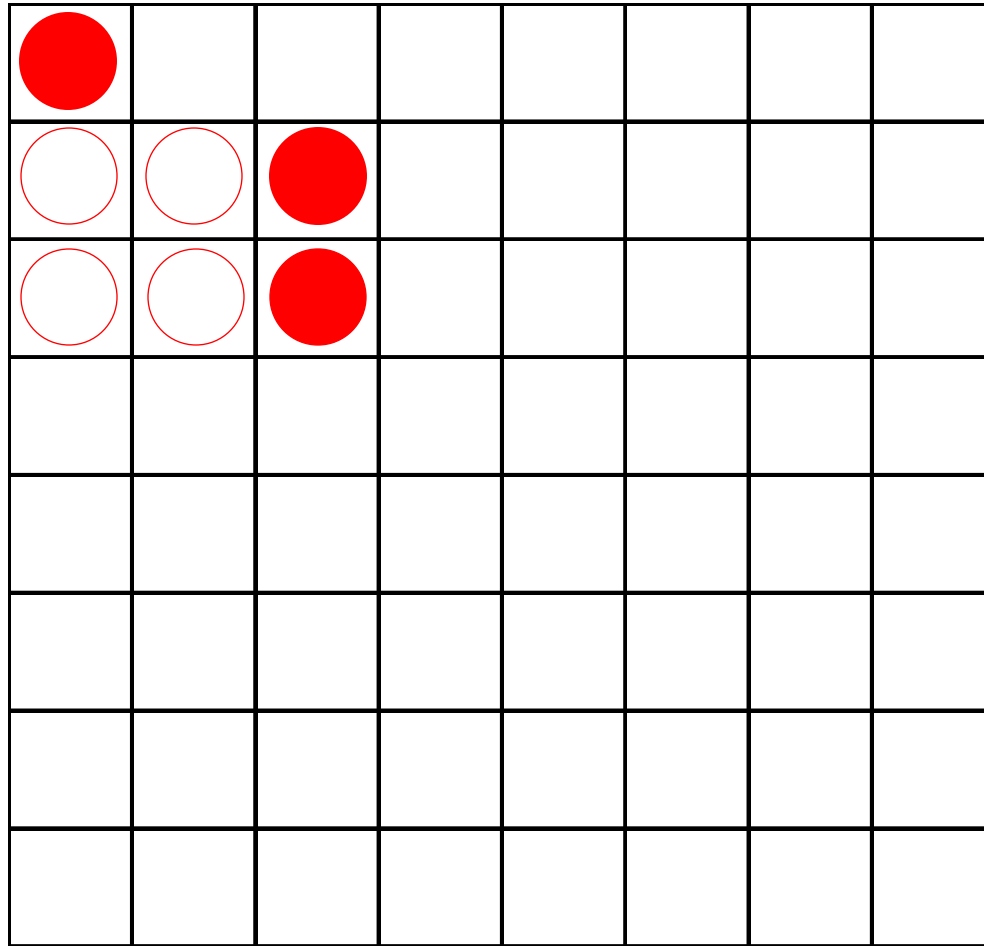
Retrocesso



Testes $4 + 2 = 6$

Retrocessos 0

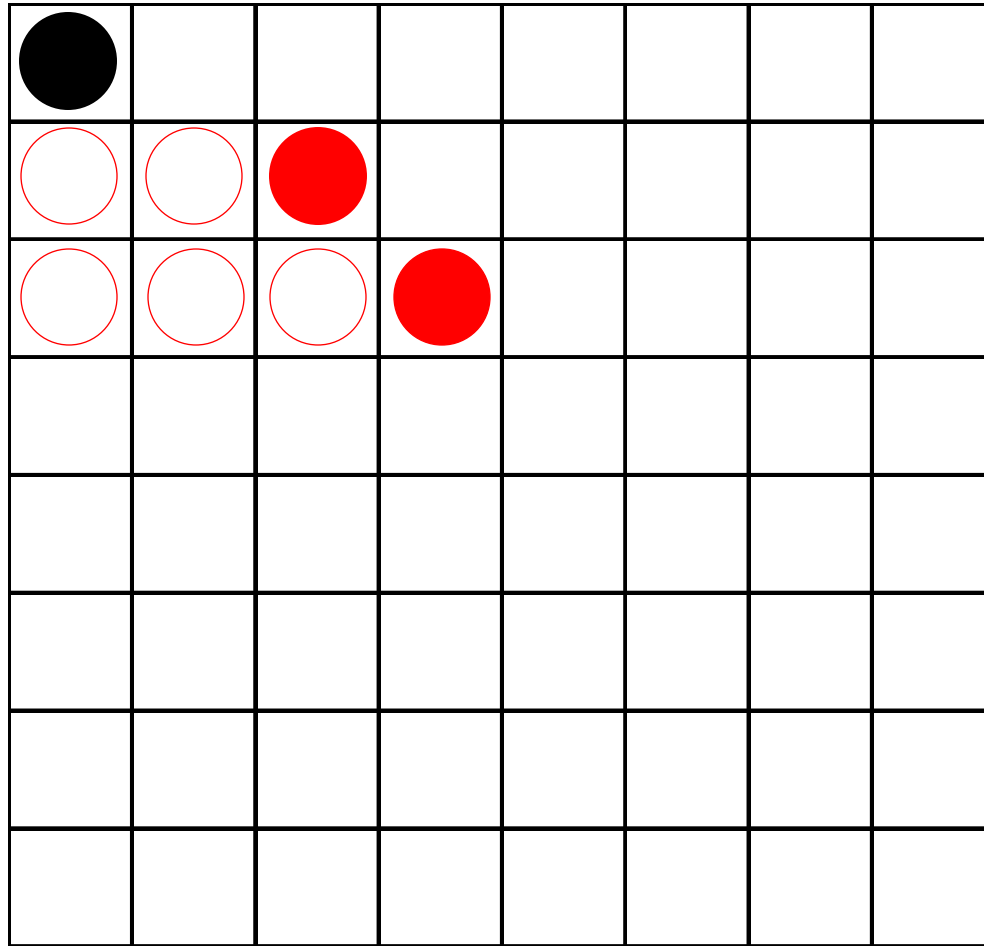
Retrocesso



Testes $6 + 1 = 7$

Retrocessos 0

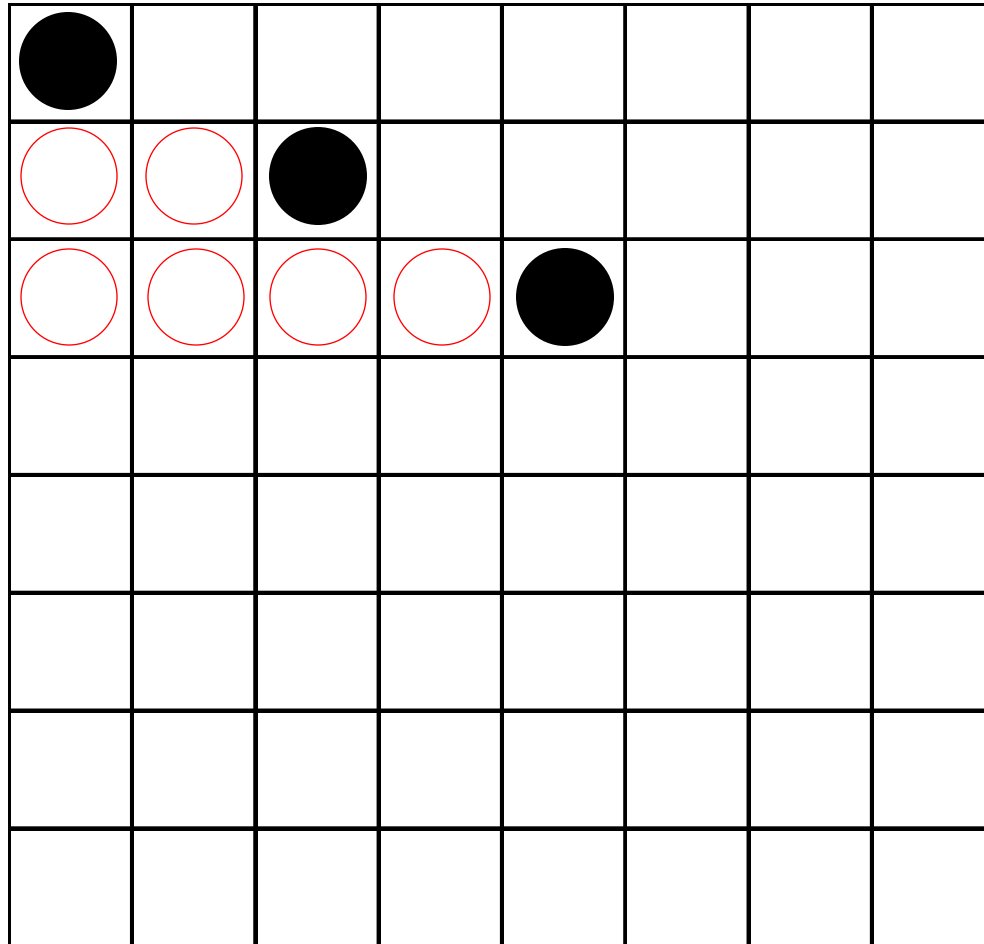
Retrocesso



Testes $7 + 2 = 9$

Retrocessos 0

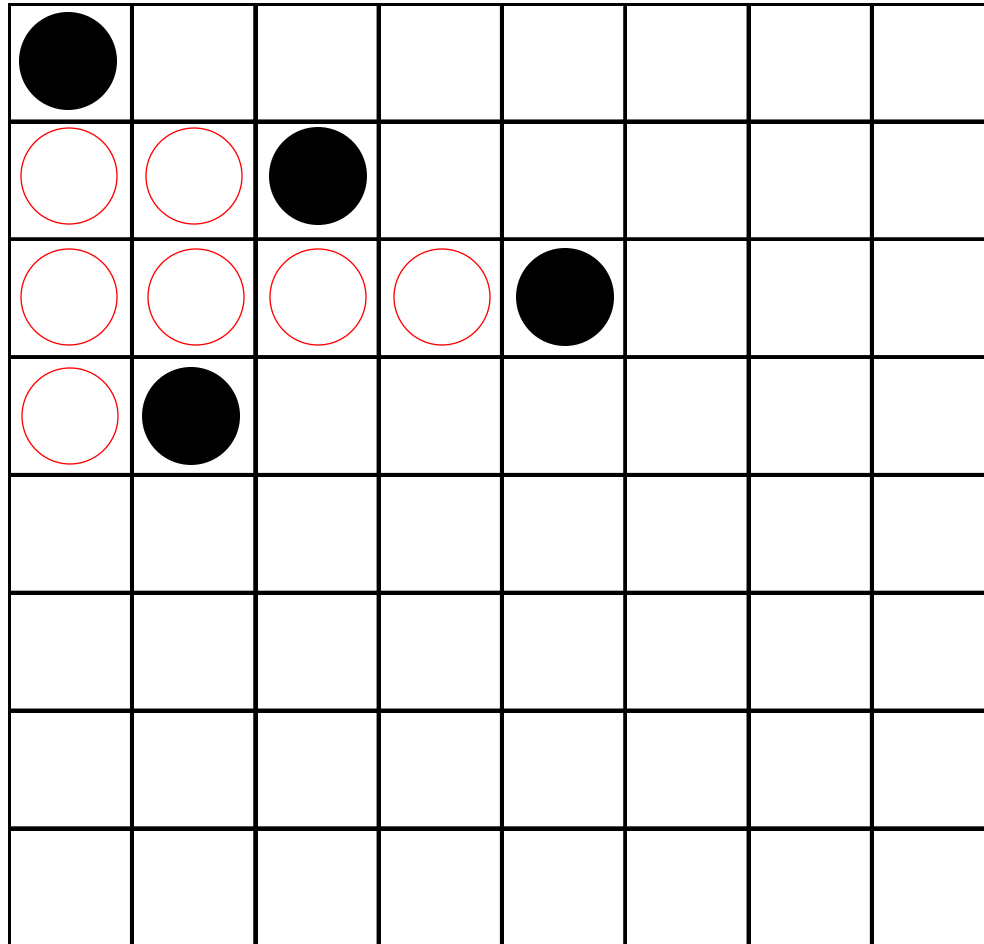
Retrocesso



Testes $9 + 2 = 11$

Retrocessos 0

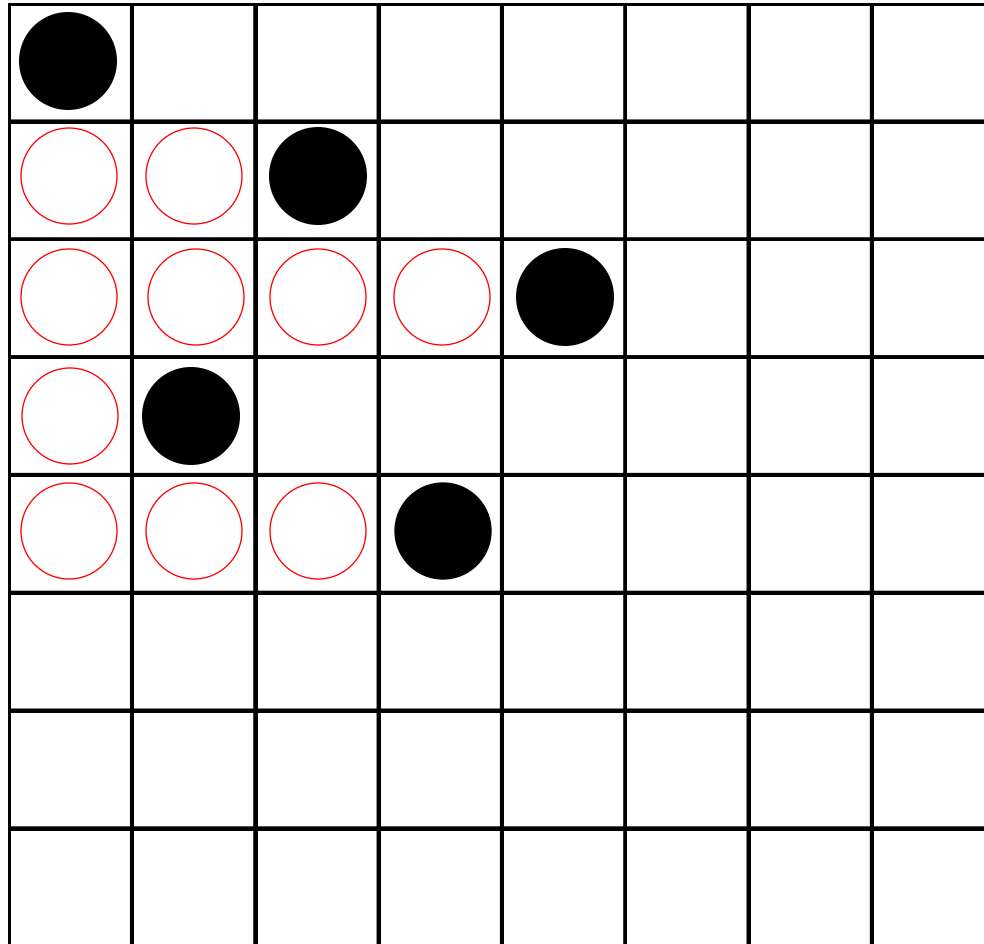
Retrocesso



Testes $11 + 1 + 3 = 15$

Retrocessos 0

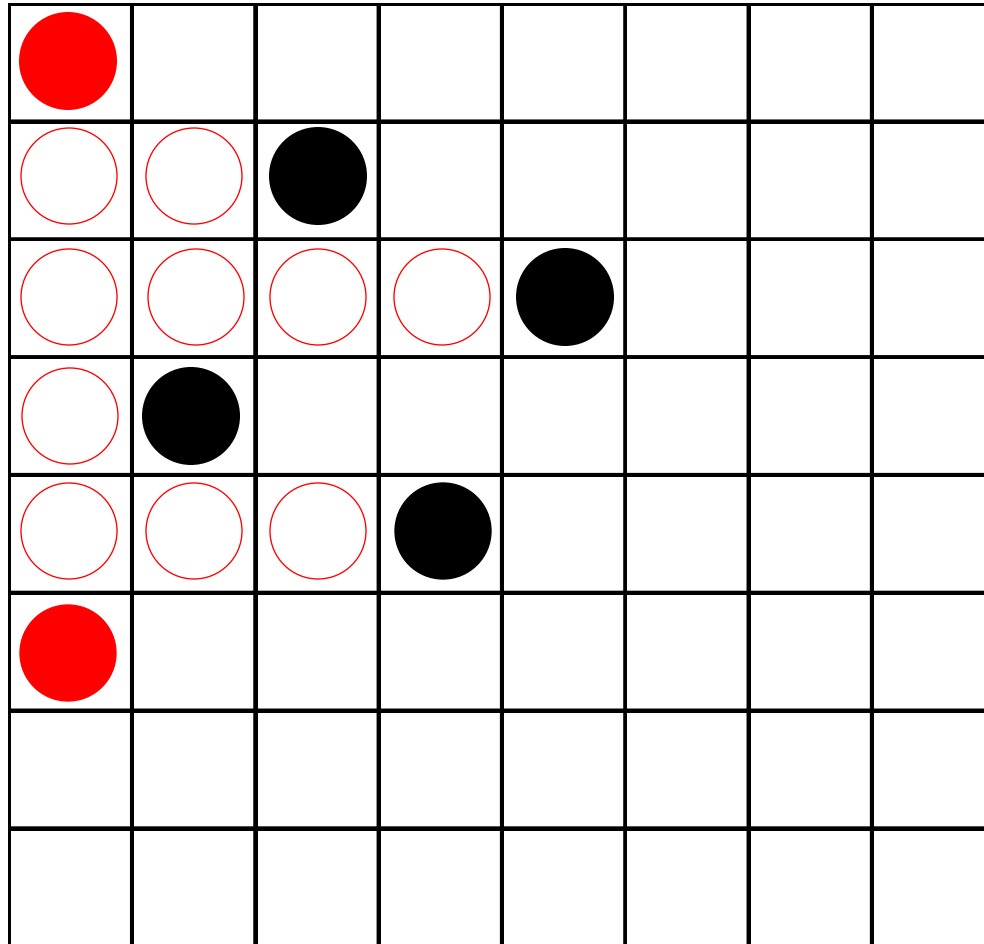
Retrocesso



Testes $15 + 1 + 4 + 2 + 4 = 26$

Retrocessos 0

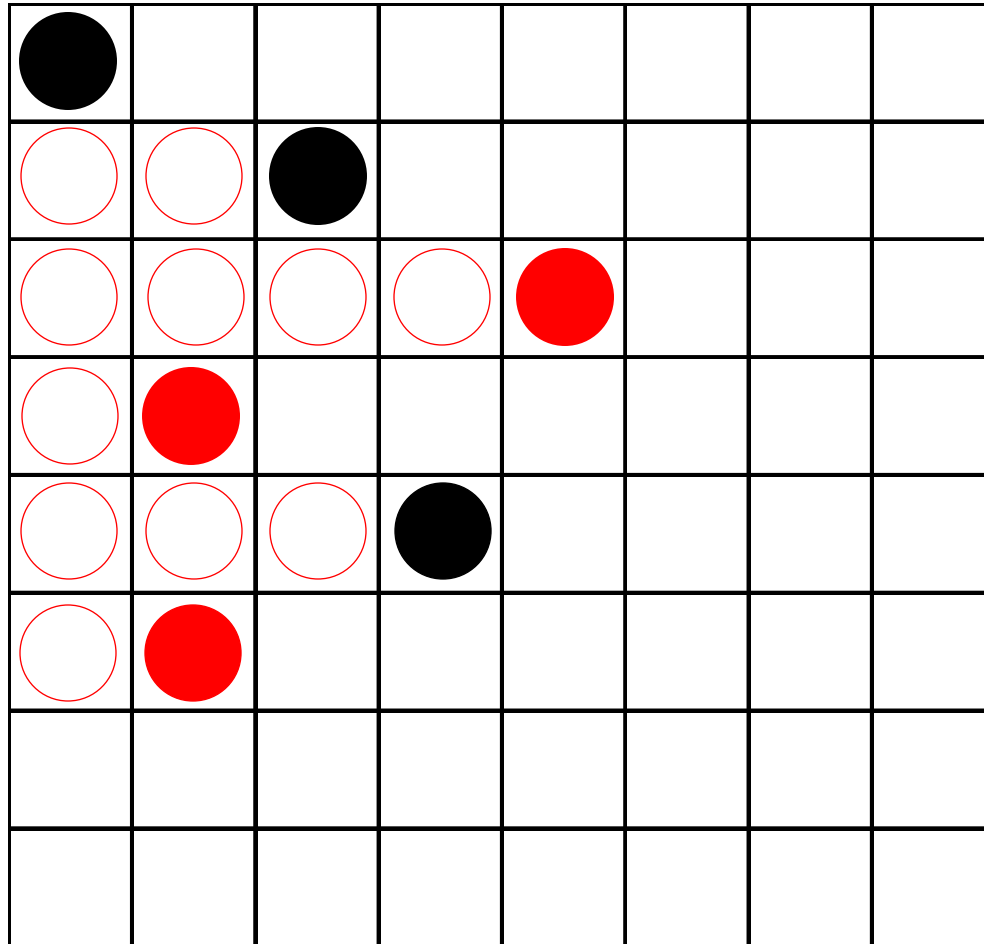
Retrocesso



Testes 26 + 1 = 27

Retrocessos 0

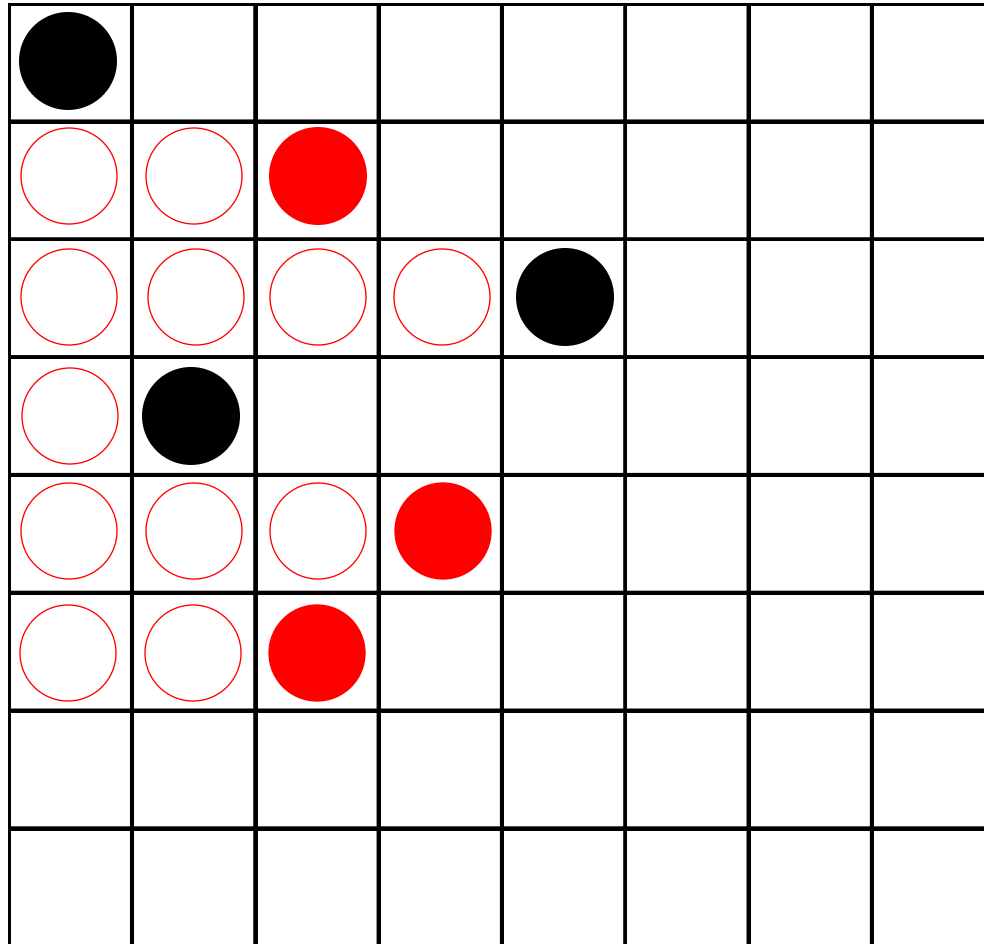
Retrocesso



Testes 27 + 3 = 30

Retrocessos 0

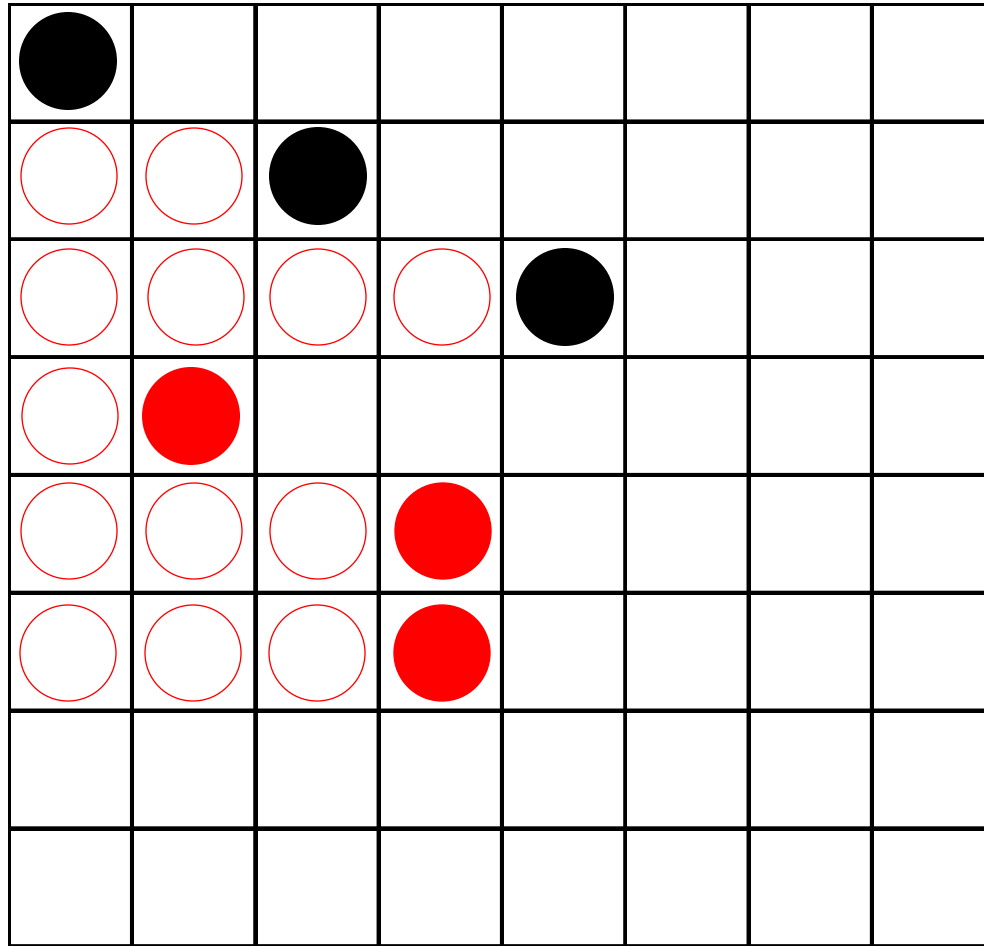
Retrocesso



Testes 30 + 2 = 32

Retrocessos 0

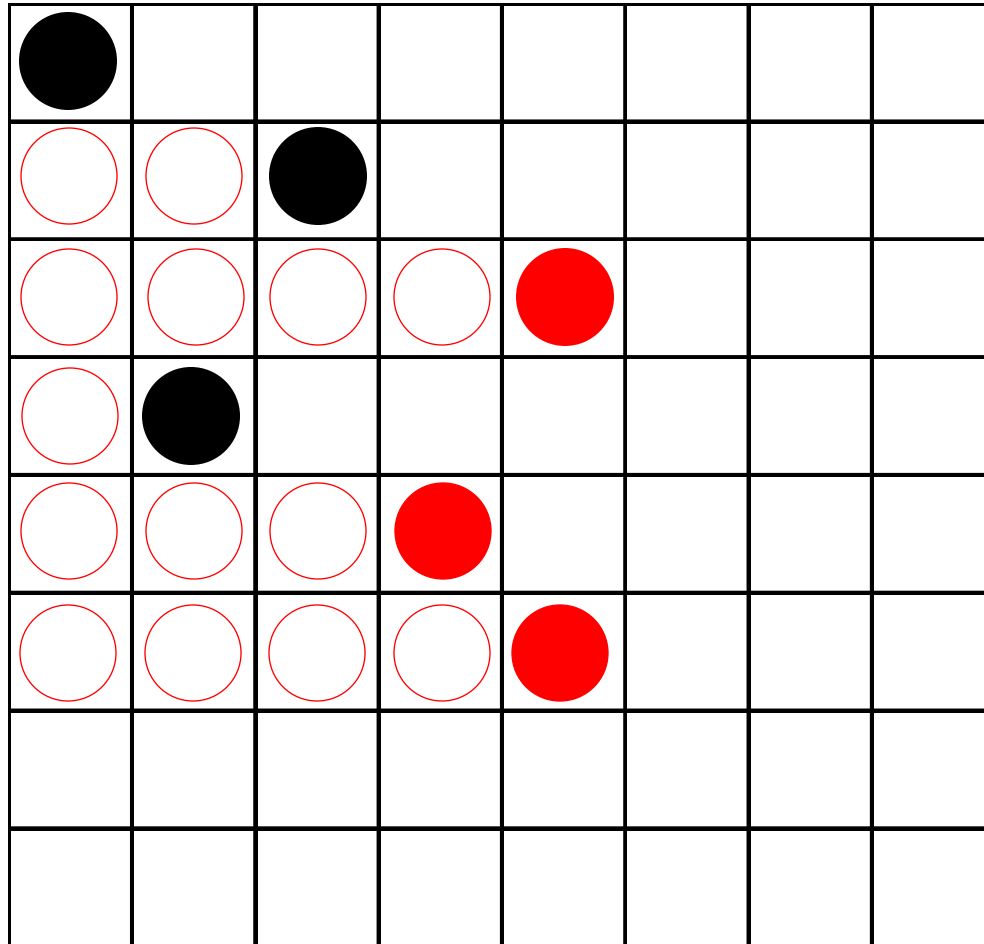
Retrocesso



Testes 32 + 4 = 36

Retrocessos 0

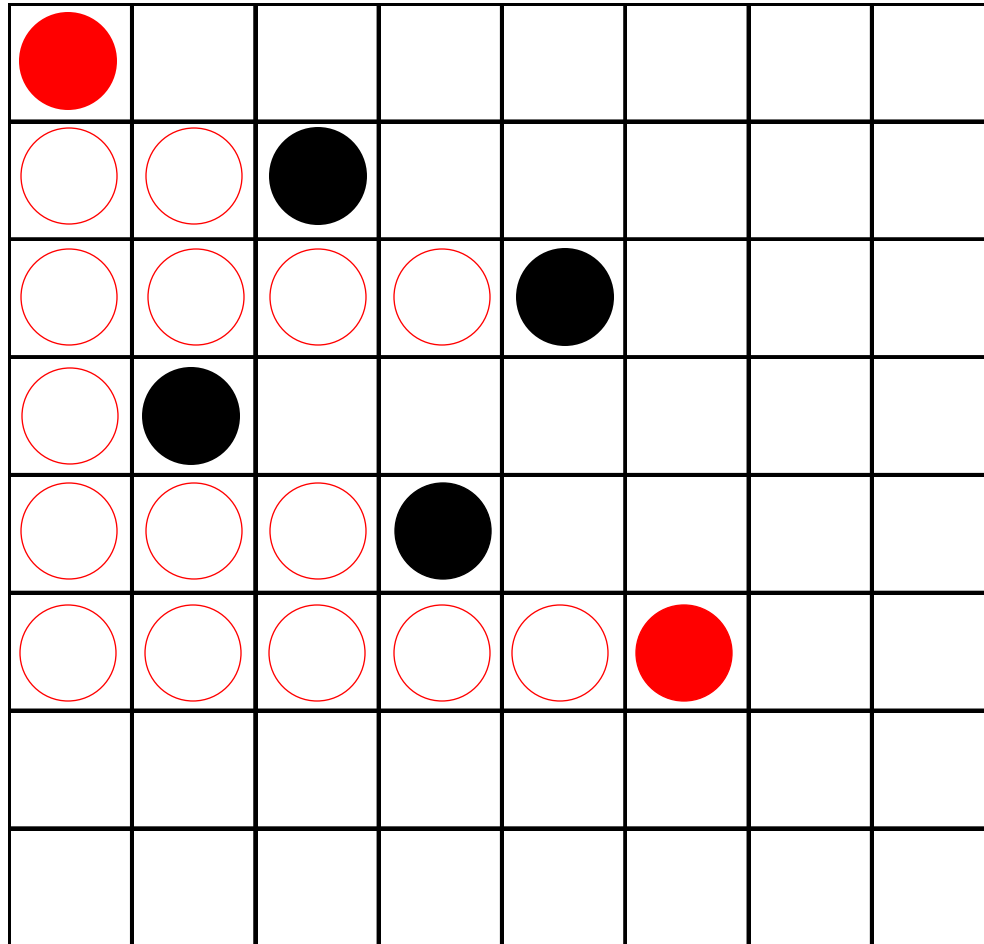
Retrocesso



Testes $36 + 3 = 39$

Retrocessos 0

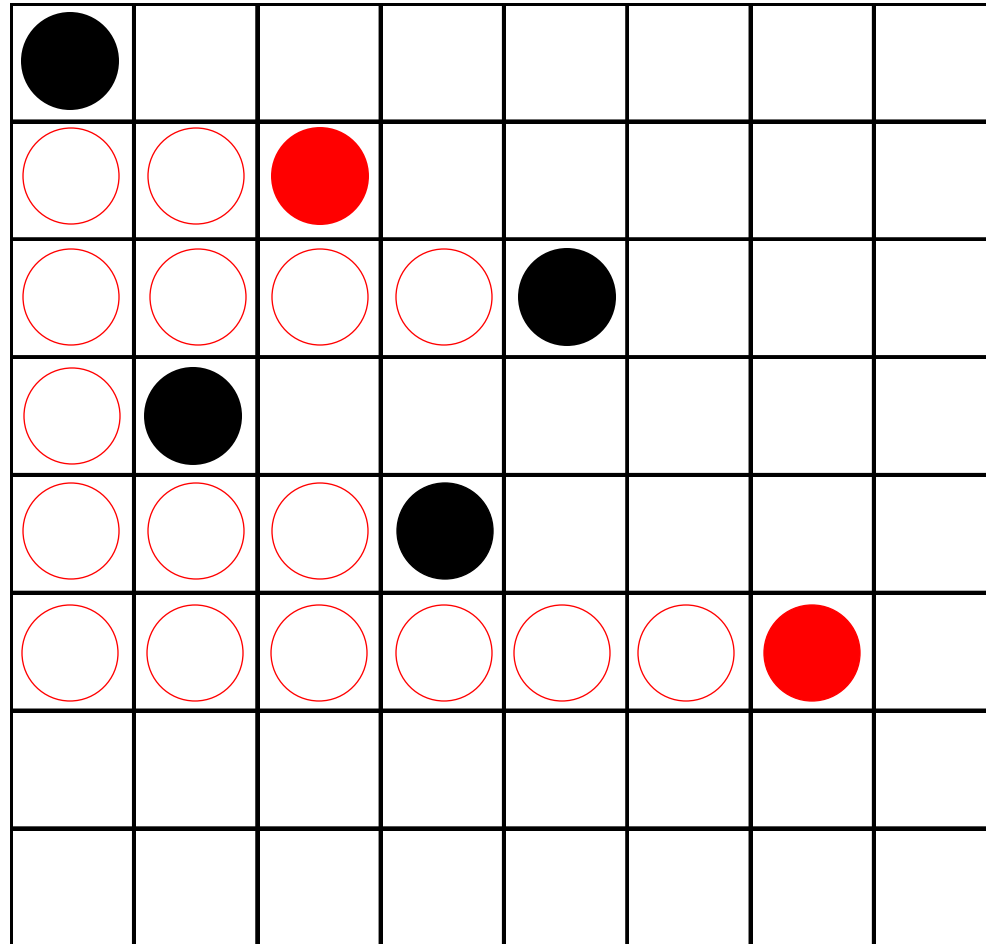
Retrocesso



Testes 39 + 1 = 40

Retrocessos 0

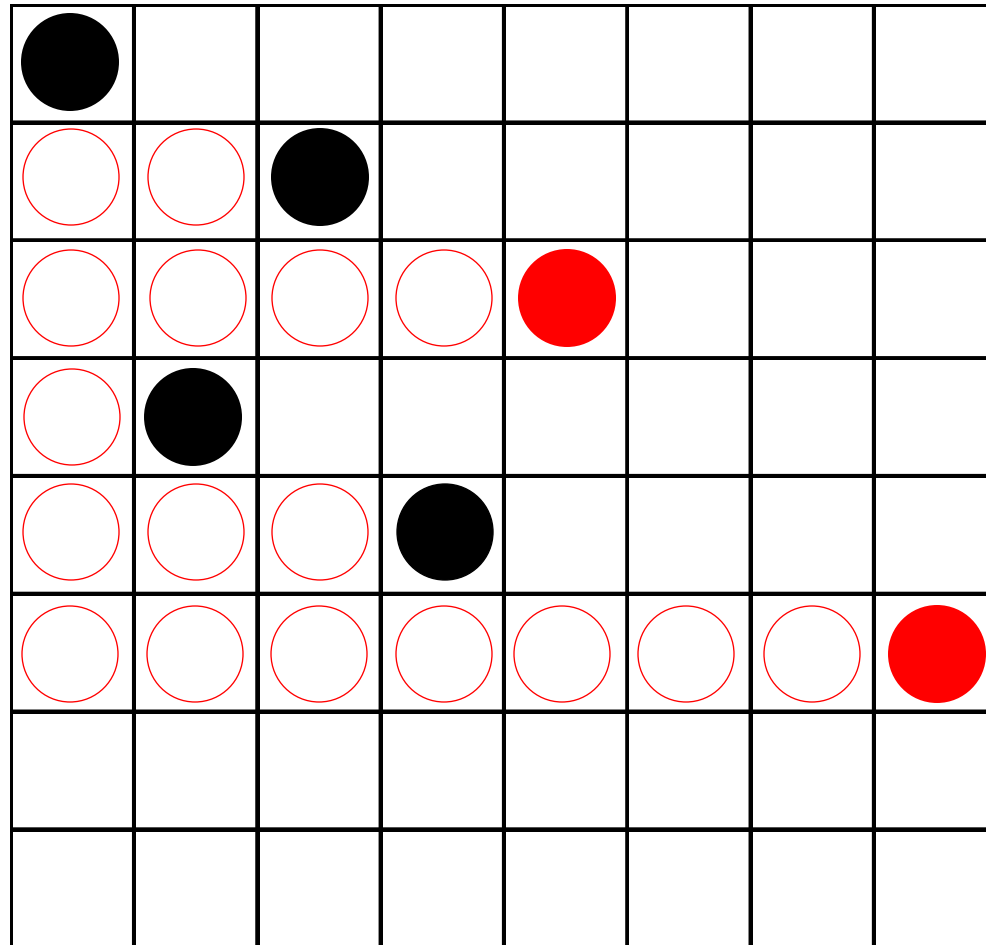
Retrocesso



Testes $40 + 2 = 42$

Retrocessos 0

Retrocesso

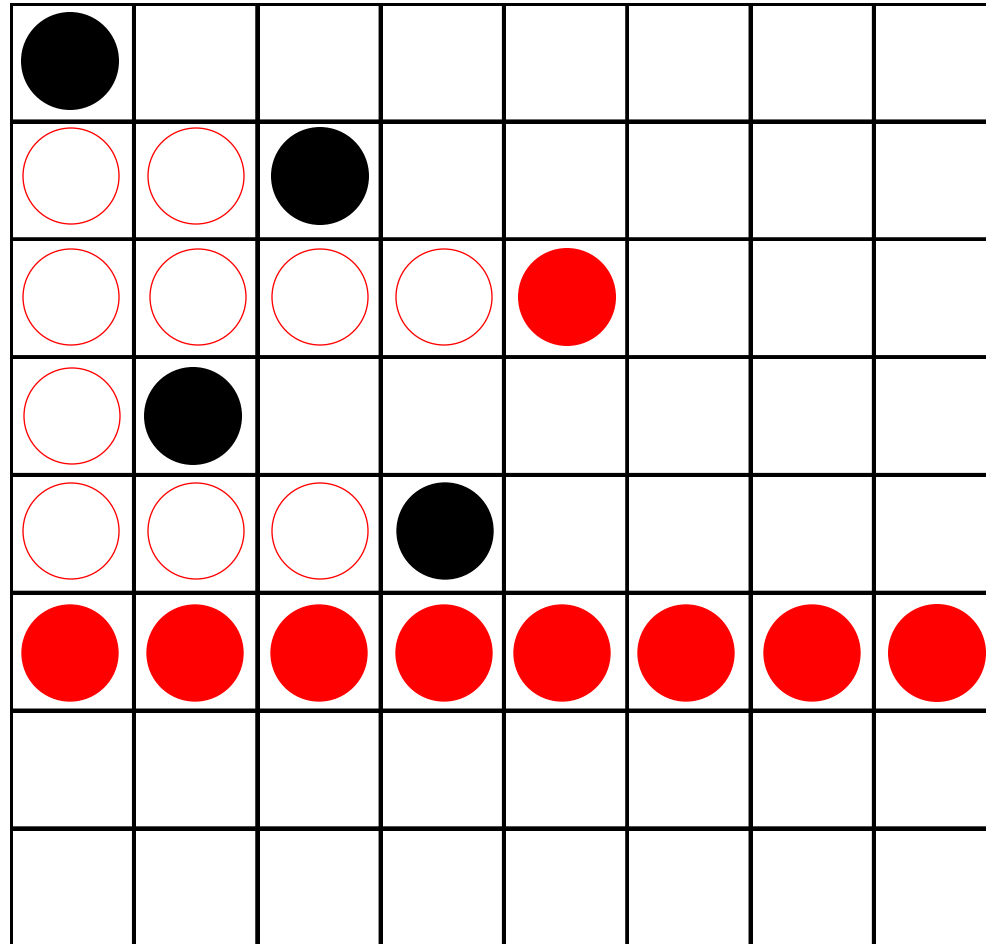


Testes $42 + 3 = 45$

Retrocessos 0

Retrocesso

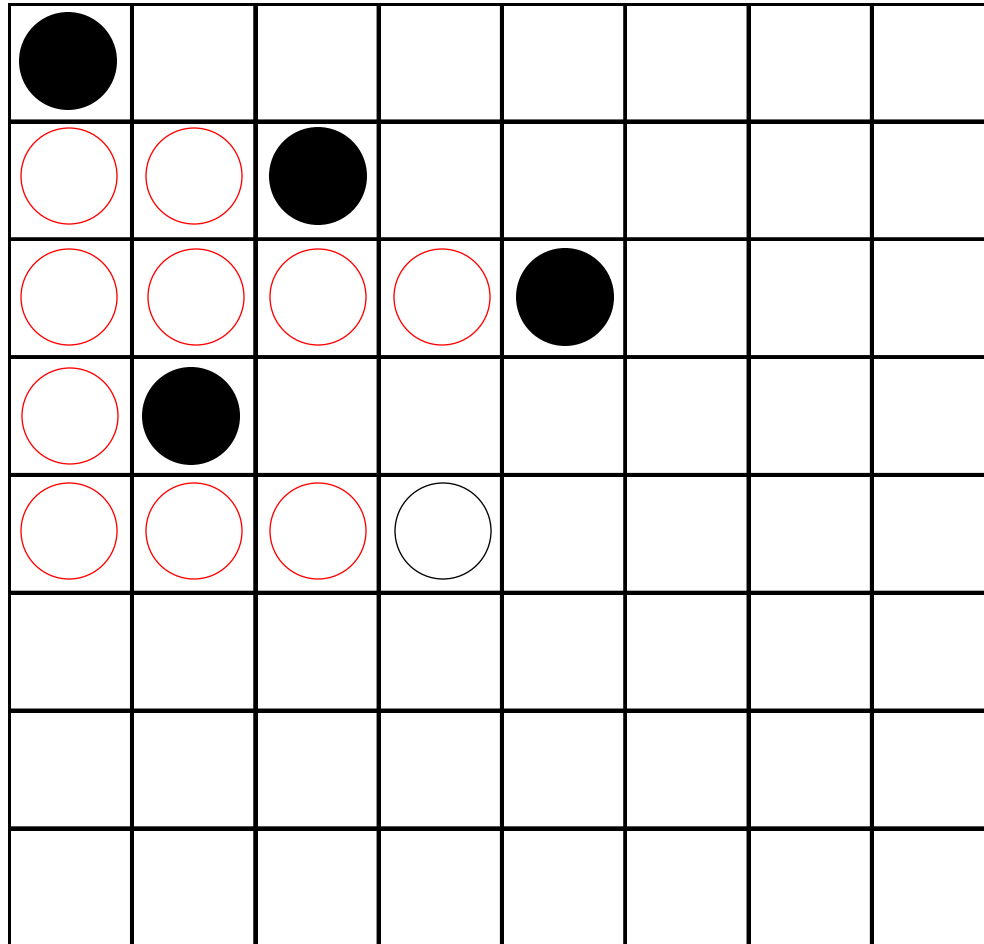
Falha
6
Retrocede
5



Testes 45

Retrocessos 0

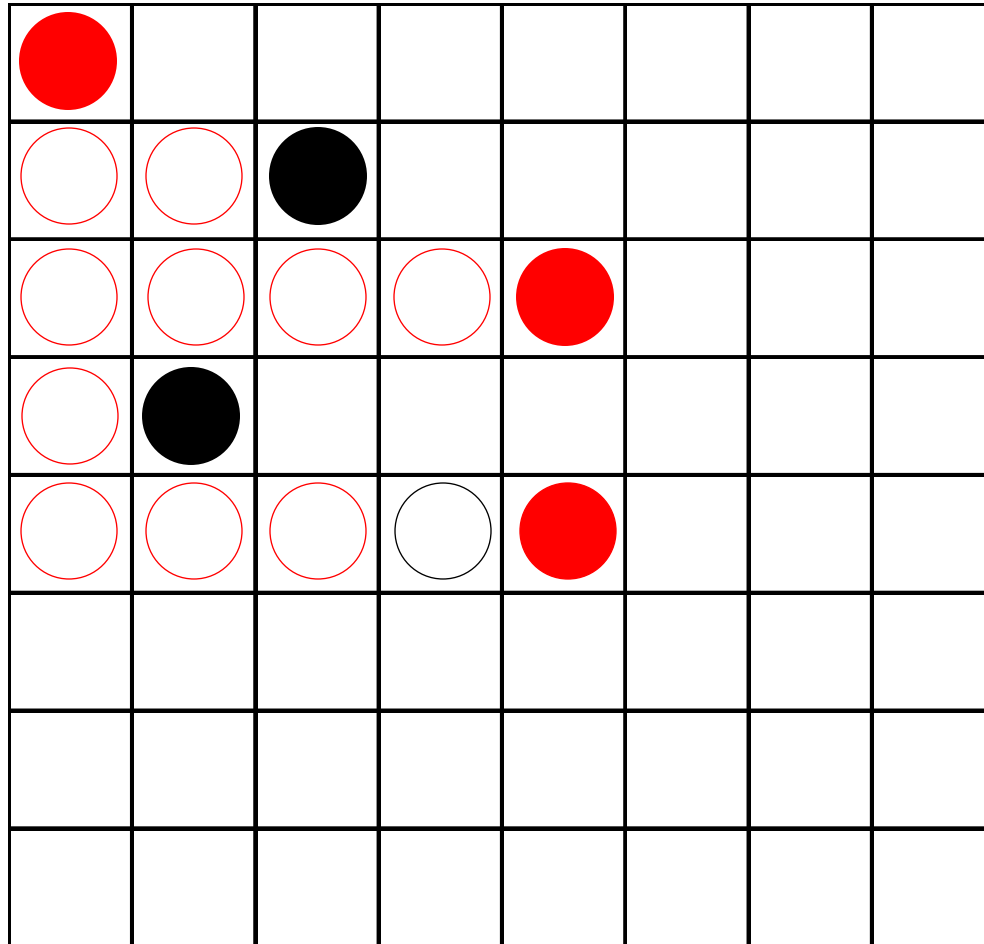
Retrocesso



Testes 45

Retrocessos 1

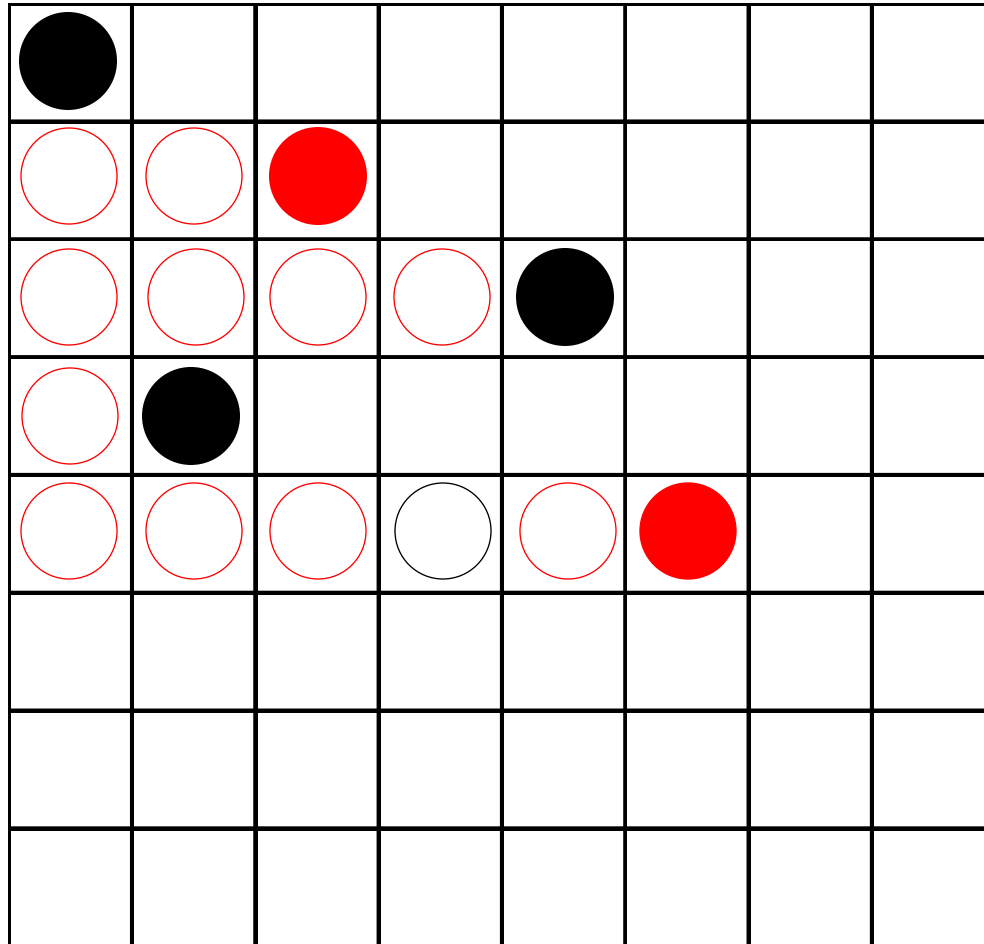
Retrocesso



Testes $45 + 1 = 46$

Retrocessos 1

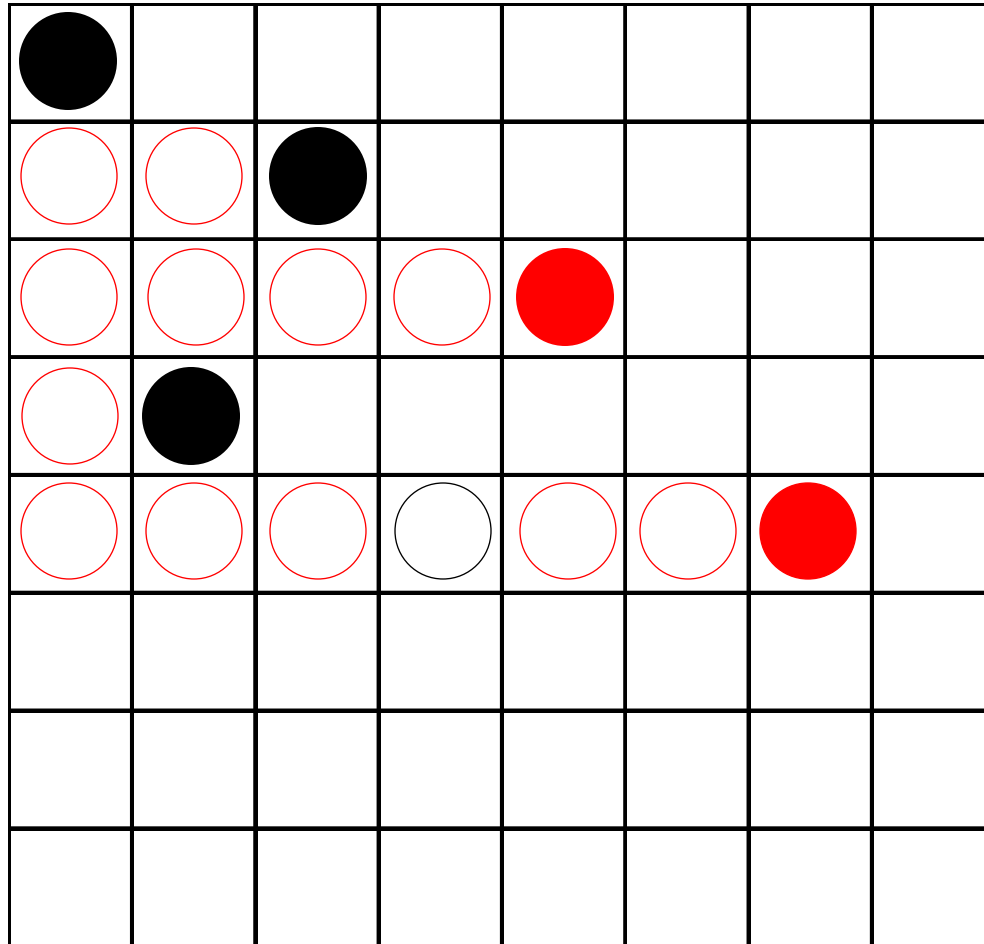
Retrocesso



Testes $46 + 2 = 48$

Retrocessos 1

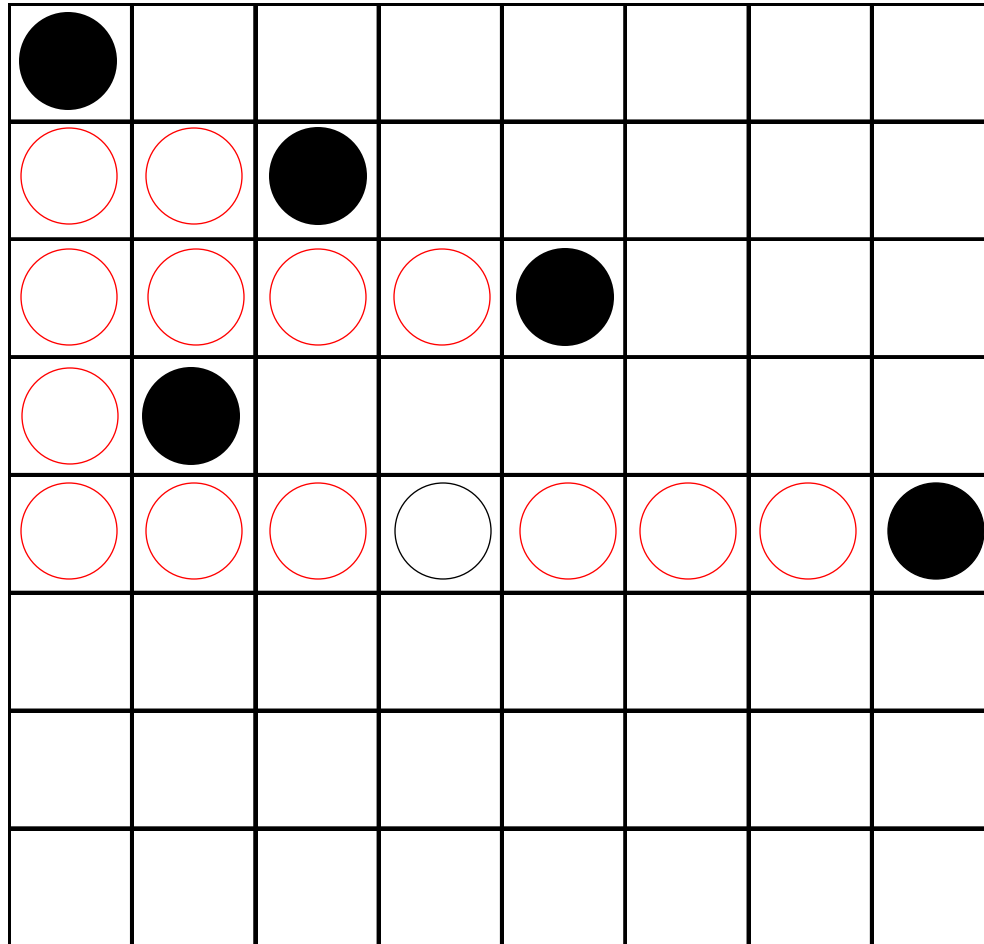
Retrocesso



Testes 48 + 3 = 51

Retrocessos 1

Retrocesso

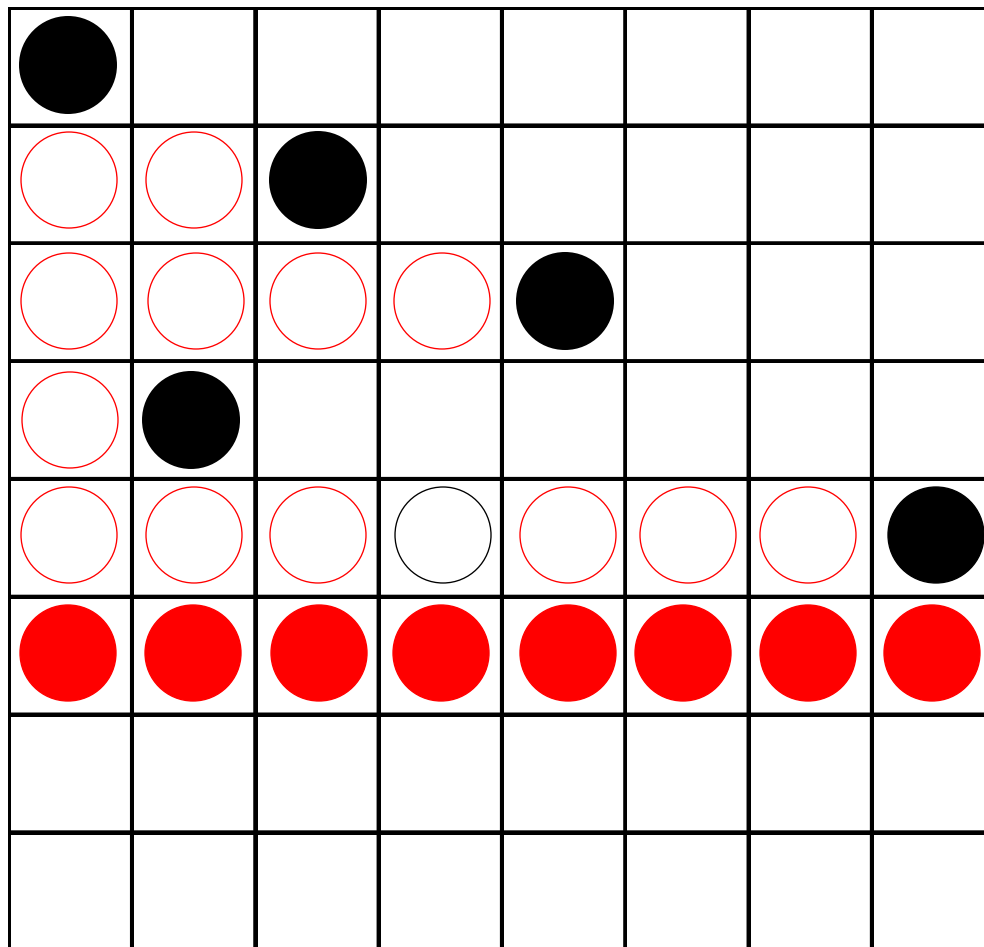


Testes $51 + 4 = 55$

Retrocessos 1

Retrocesso

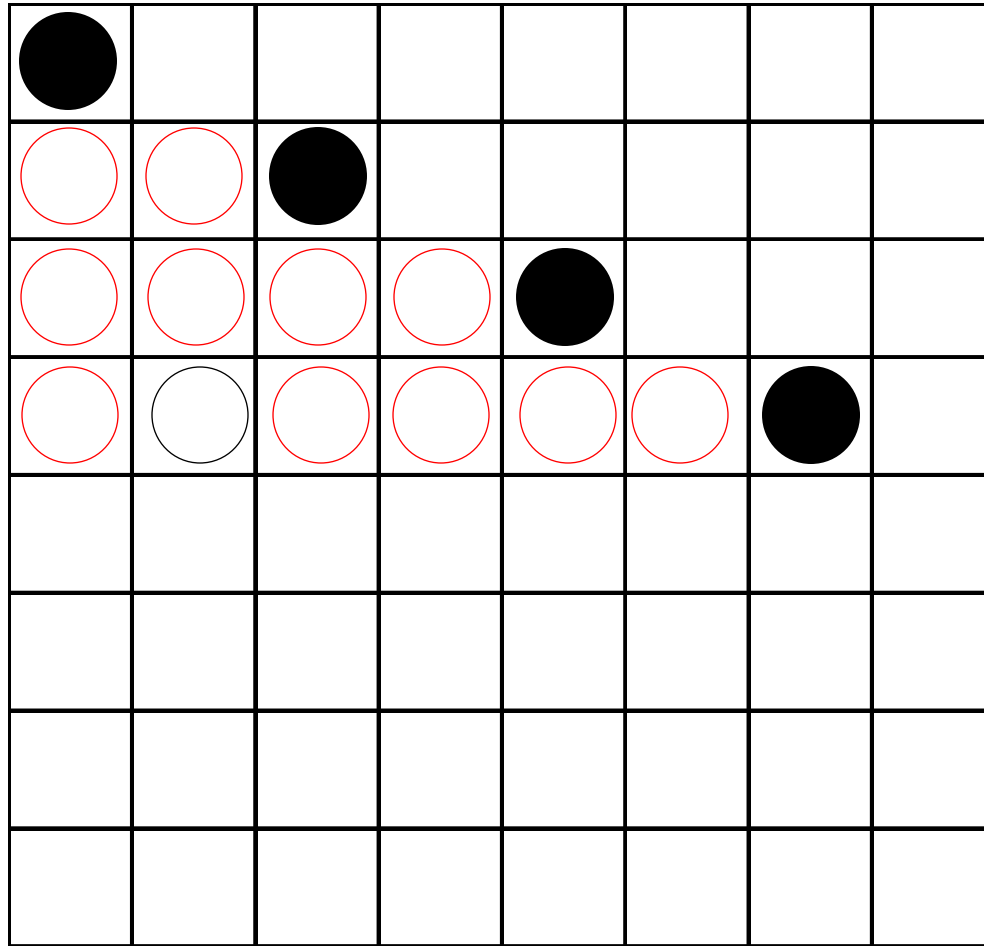
Falha
6
Retrocede
5 e 4



Testes $55+1+3+2+4+3+1+2+3 = 74$

Retrocessos 1

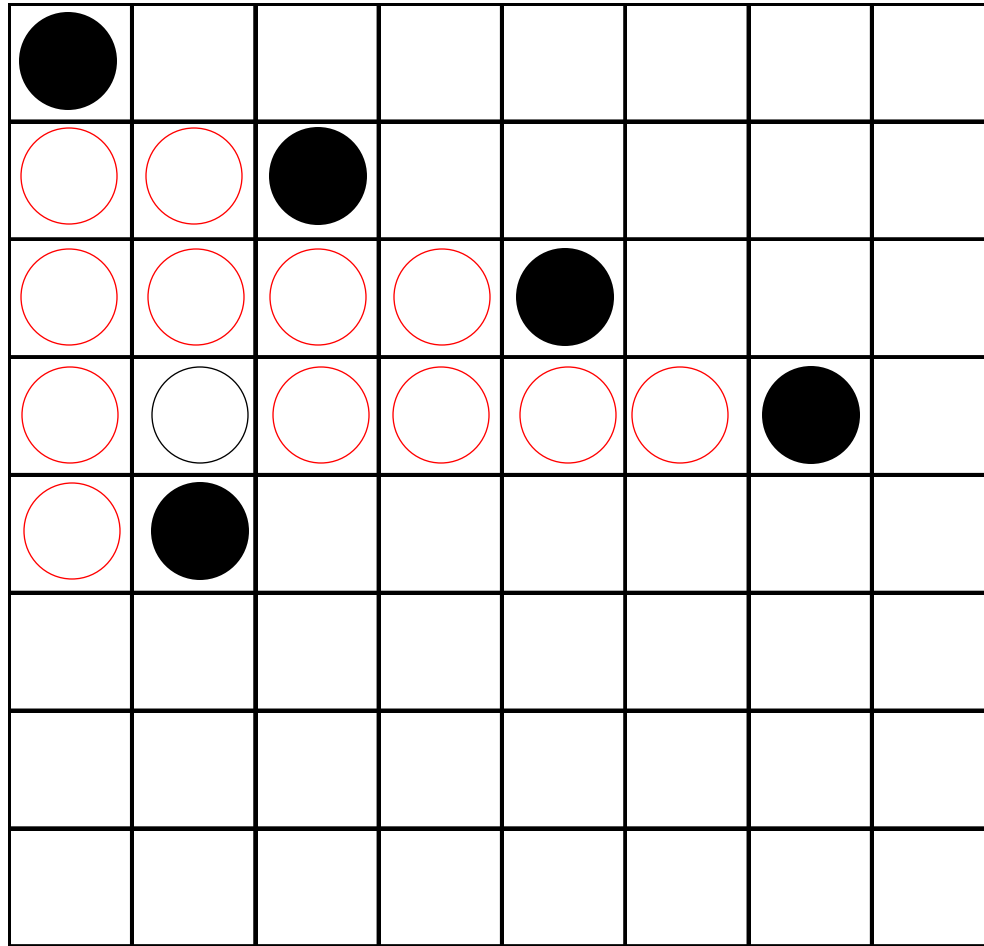
Retrocesso



Testes $74+2+1+2+3+3= 85$

Retrocessos $1+2 = 3$

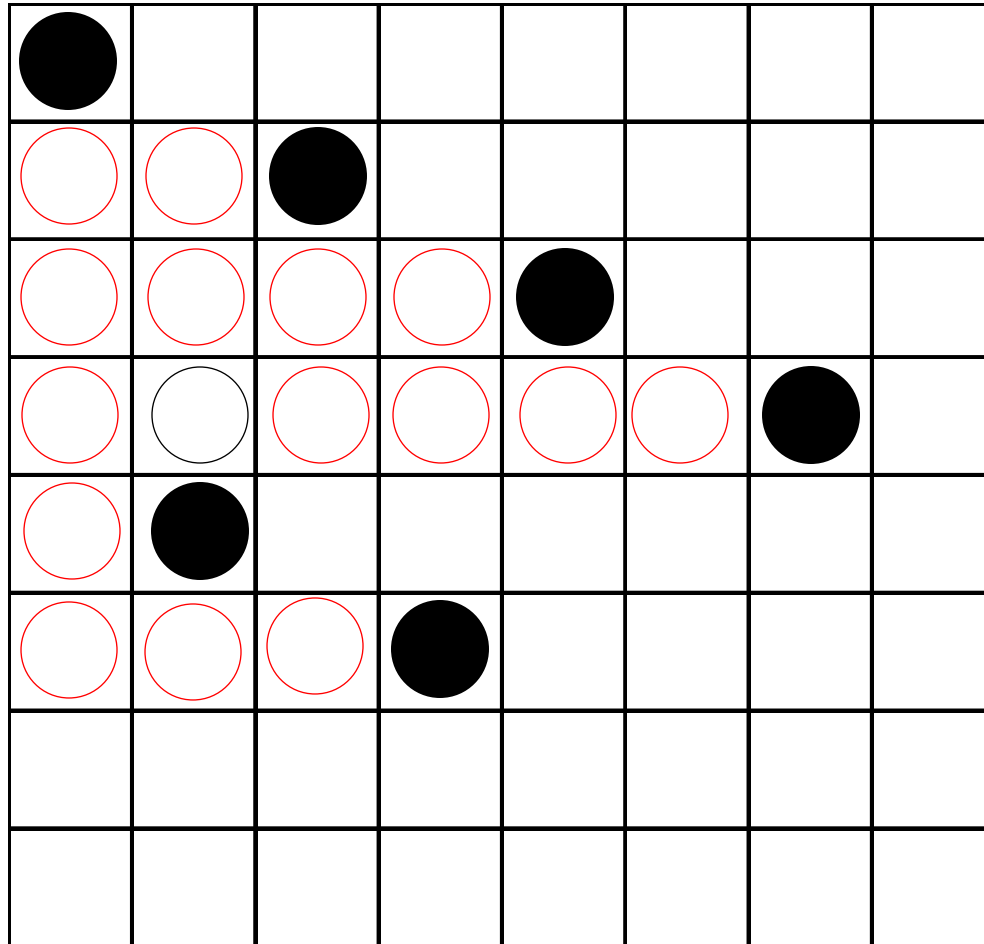
Retrocesso



Testes $85 + 1 + 4 = 90$

Retrocessos 3

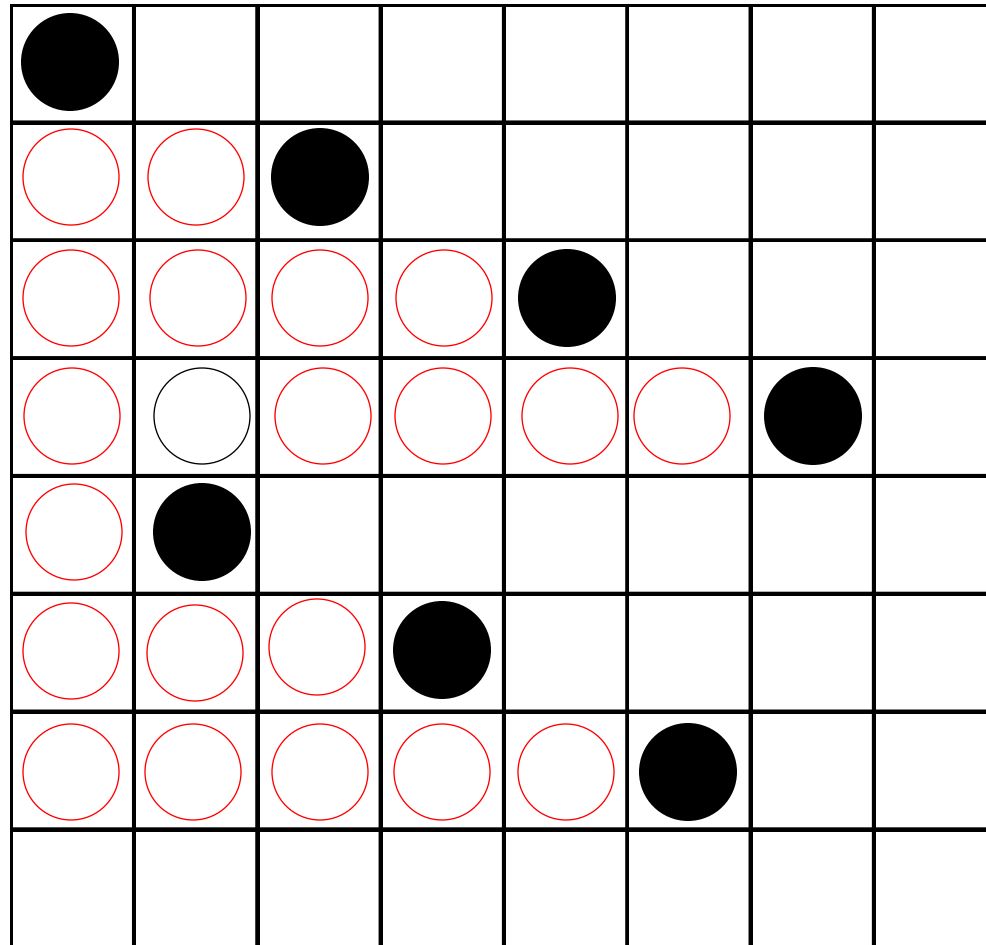
Retrocesso



Testes $90 + 1 + 3 + 2 + 5 = 101$

Retrocessos 3

Retrocesso

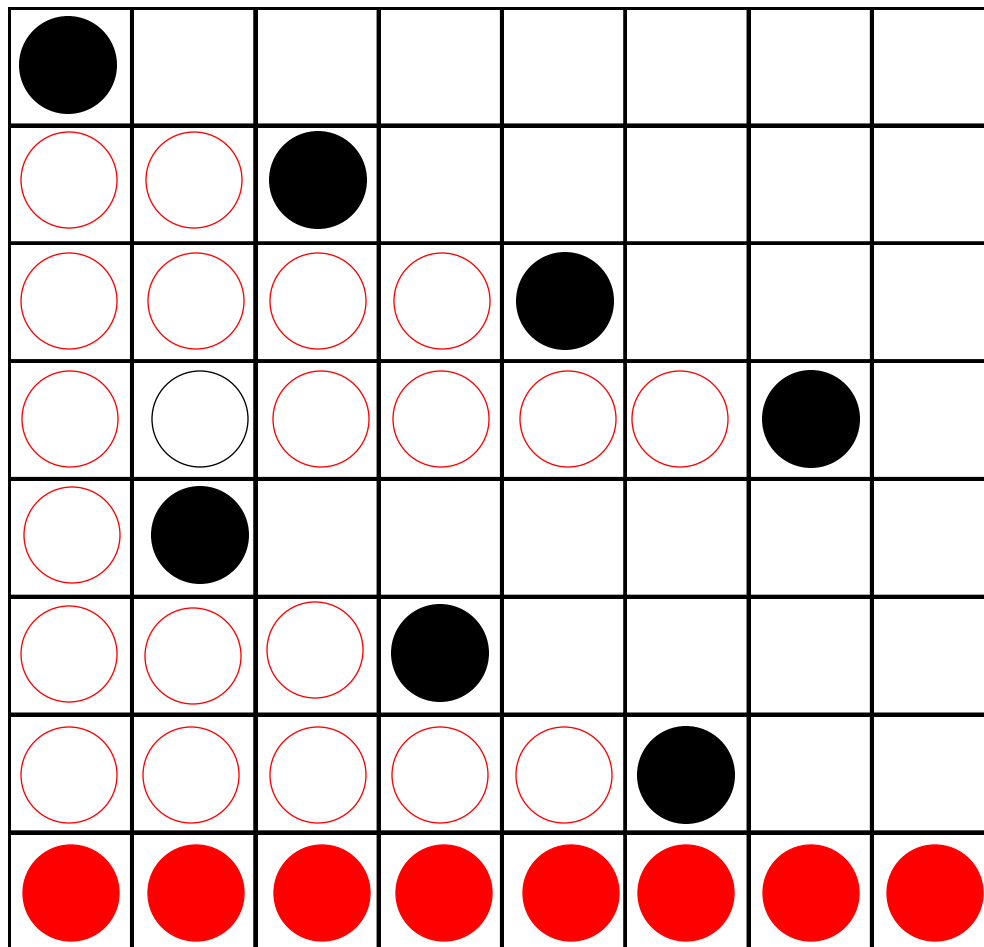


Testes $101+1+5+2+4+3+6= 122$

Retrocessos 3

Retrocesso

Falha
8
Retrocede
7

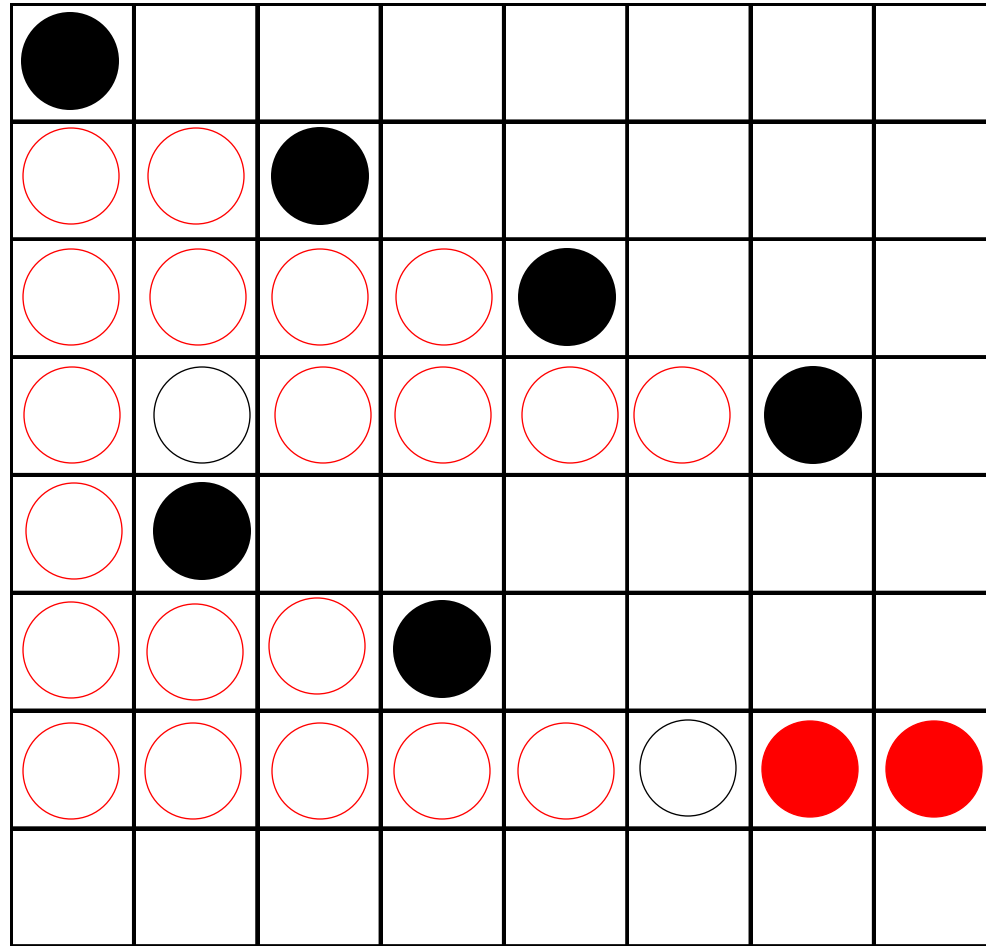


Testes $122+1+5+2+6+3+6+4+1= 150$

Retrocessos $3+1=4$

Retrocesso

Falha
7
Retrocede
6

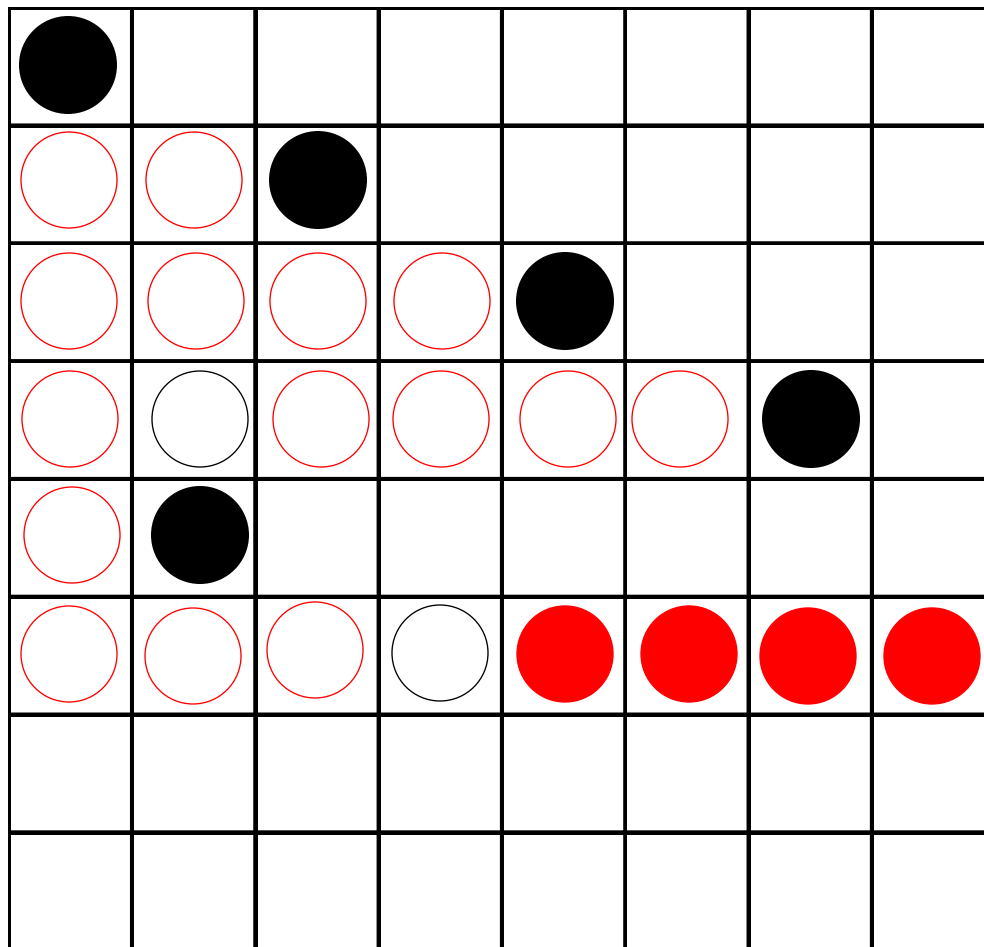


Testes $150+1+2=153$

Retrocessos $4+1=5$

Retrocesso

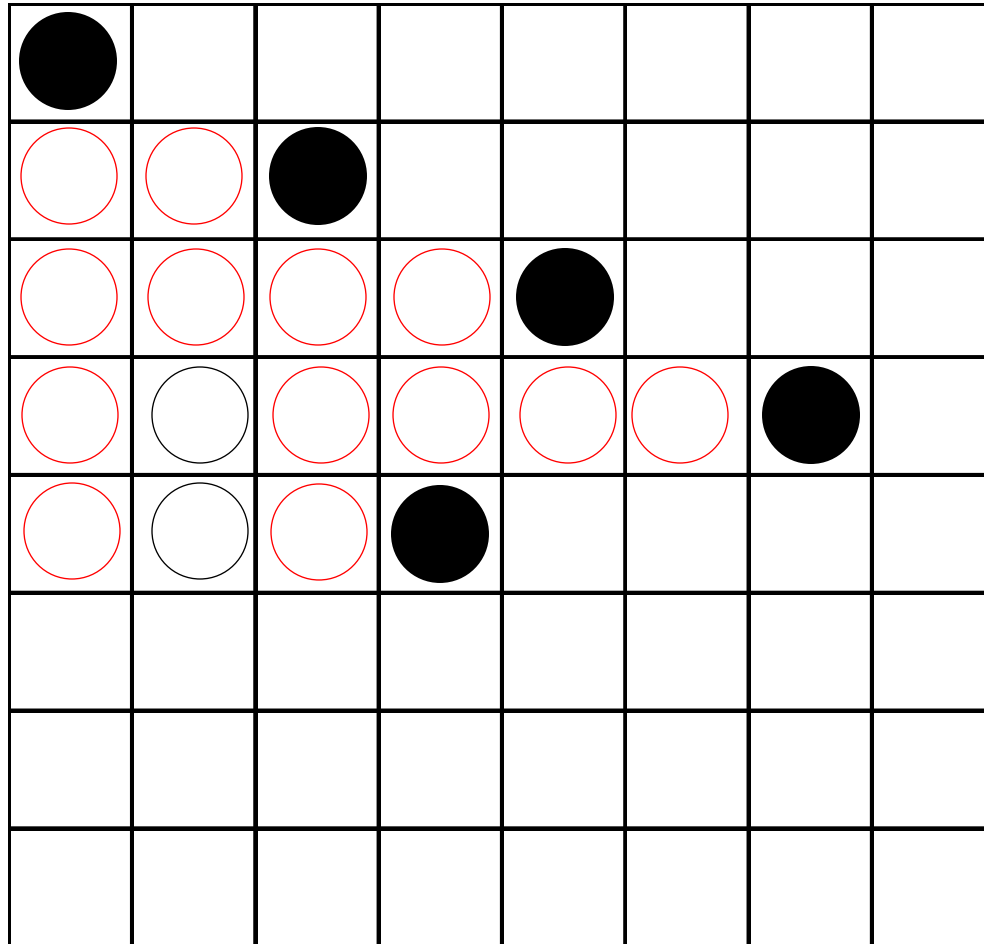
Falha
6
Retrocede
5



Testes $153+3+1+2+3= 162$

Retrocessos $5+1=6$

Retrocesso

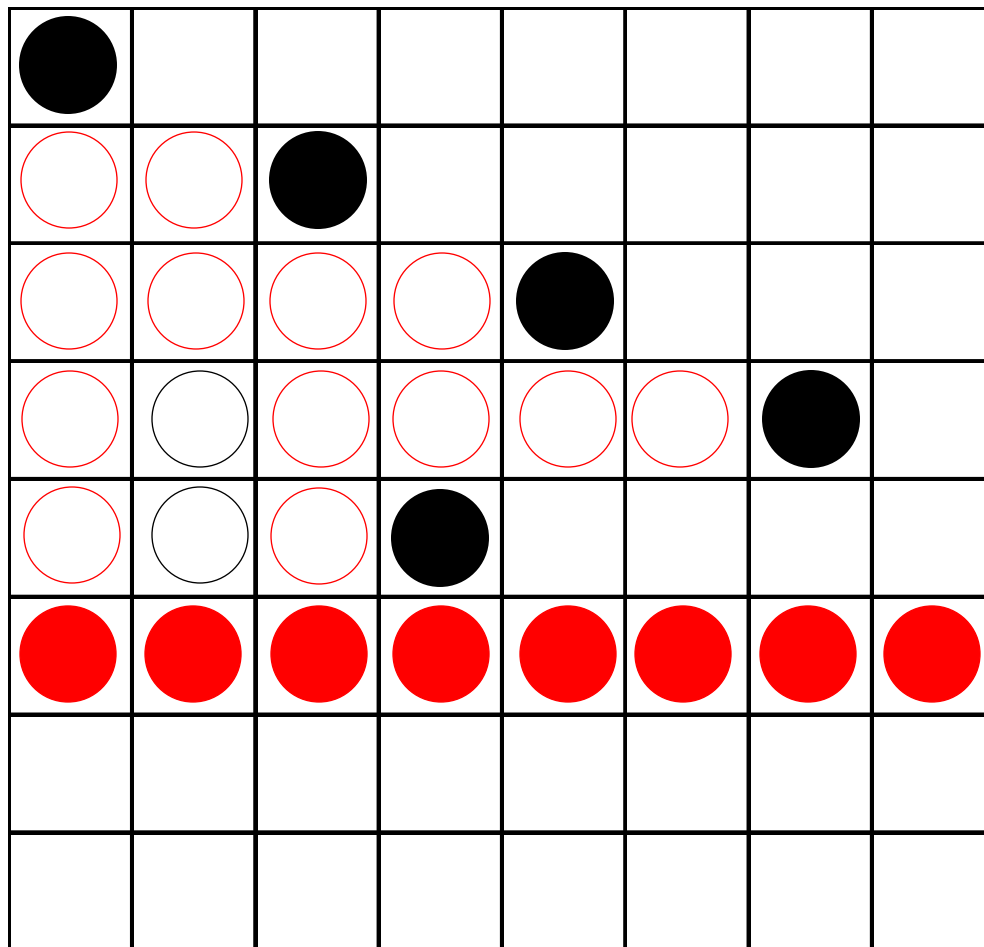


Testes $162+2+4=168$

Retrocessos $6+1=7$

Retrocesso

Falha
6
Retrocede
5

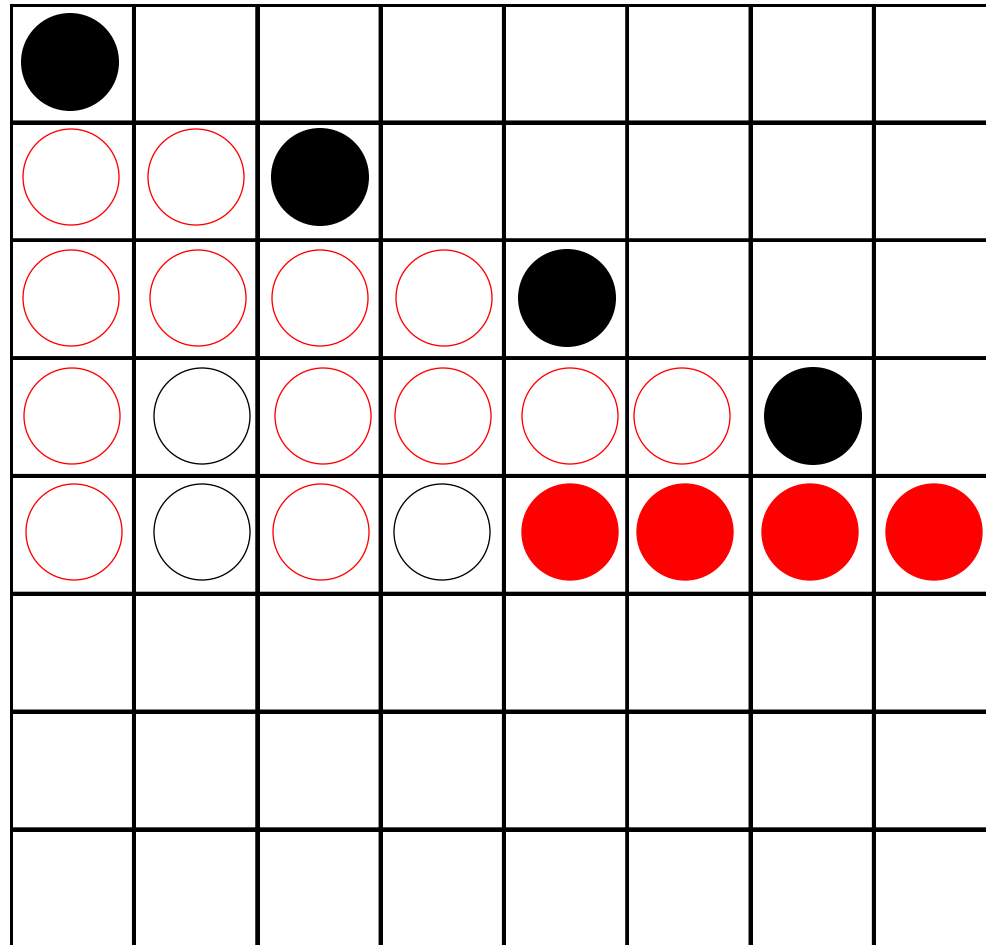


Testes $168+1+3+2+5+3+1+2+3= 188$

Retrocessos 7

Retrocesso

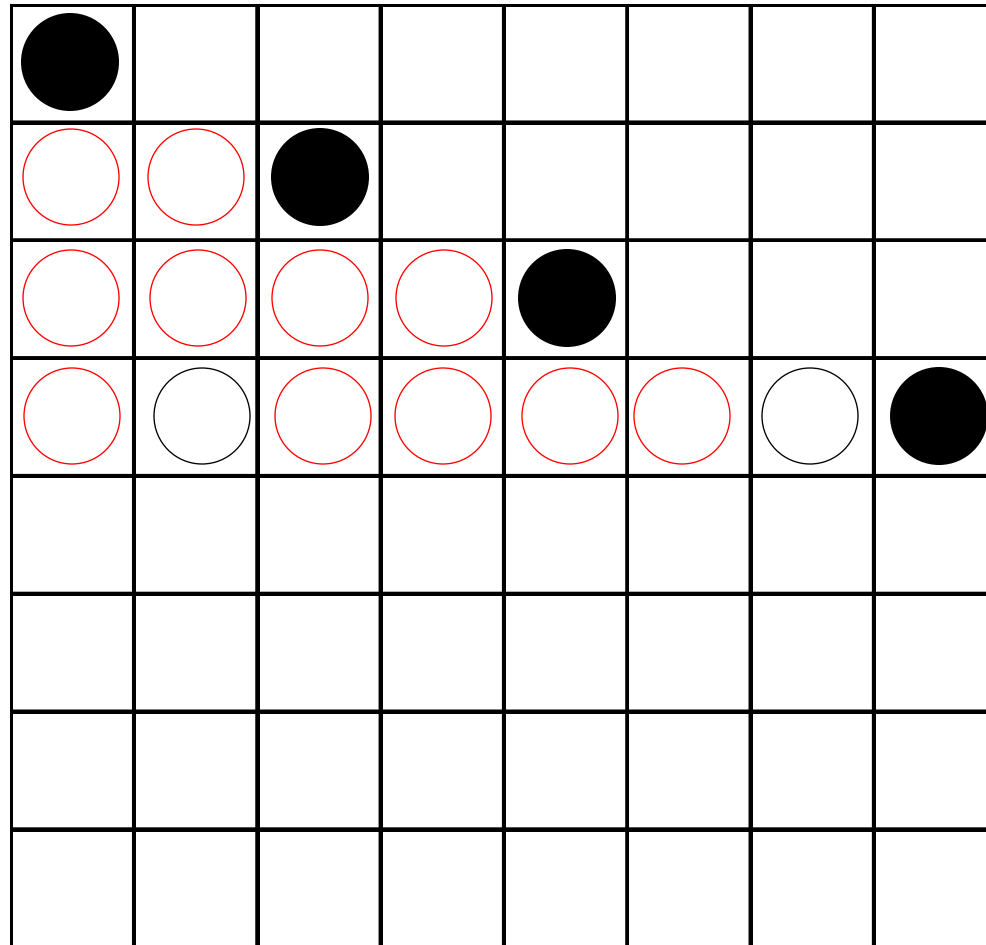
Falha
5
Retrocede
4



Testes $188+1+2+3+4=198$

Retrocessos $7+1=8$

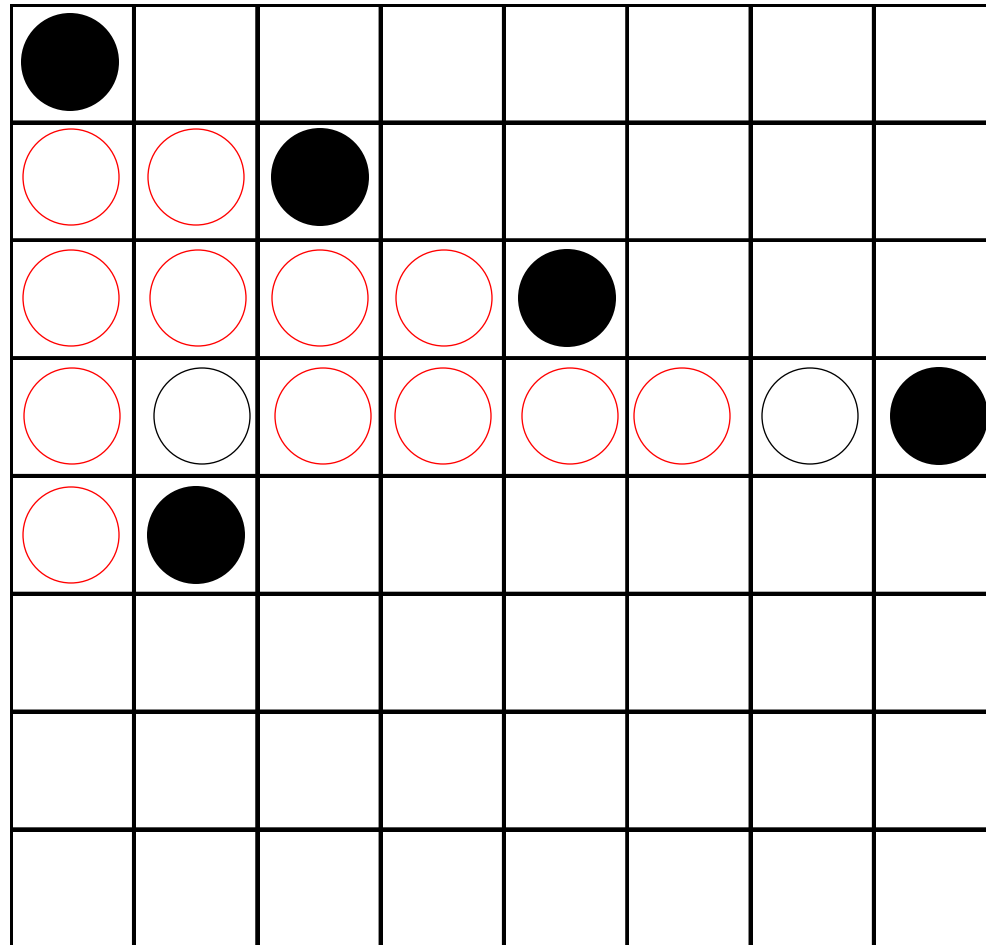
Retrocesso



Testes $198 + 3 = 201$

Retrocessos $8+1=9$

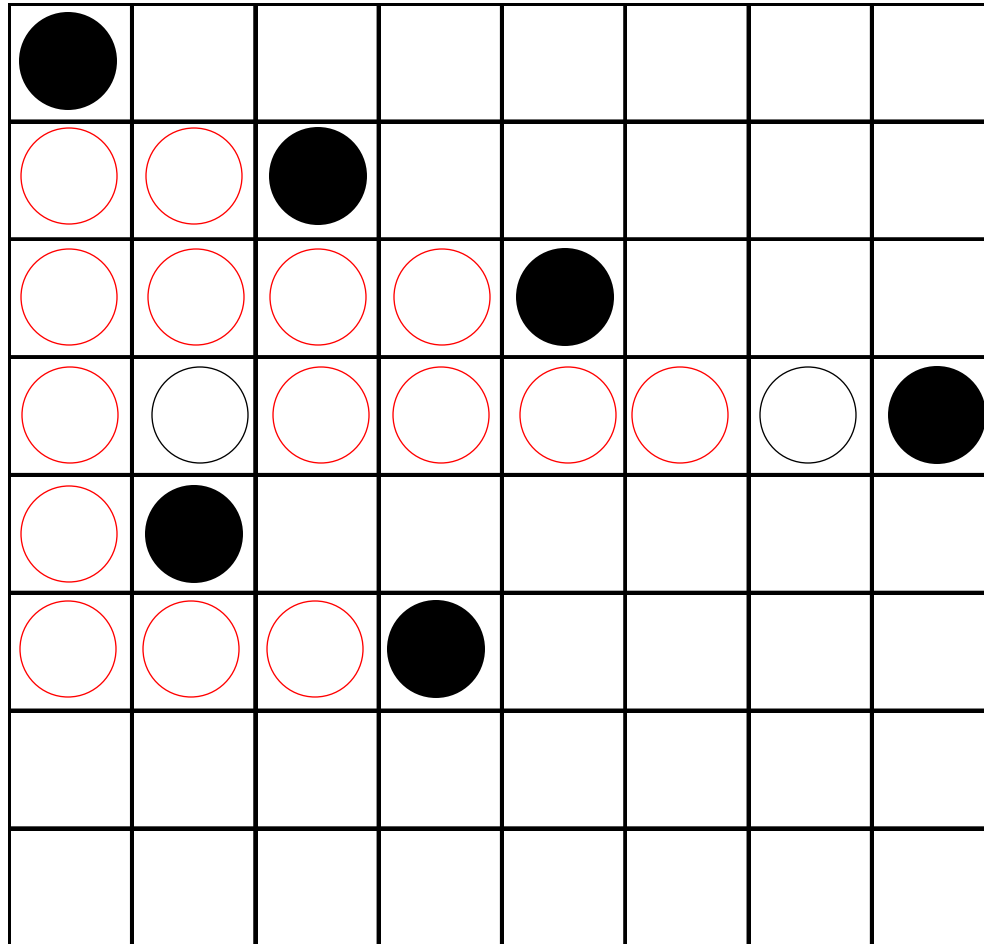
Retrocesso



Testes 201+1+4 = 206

Retrocessos 9

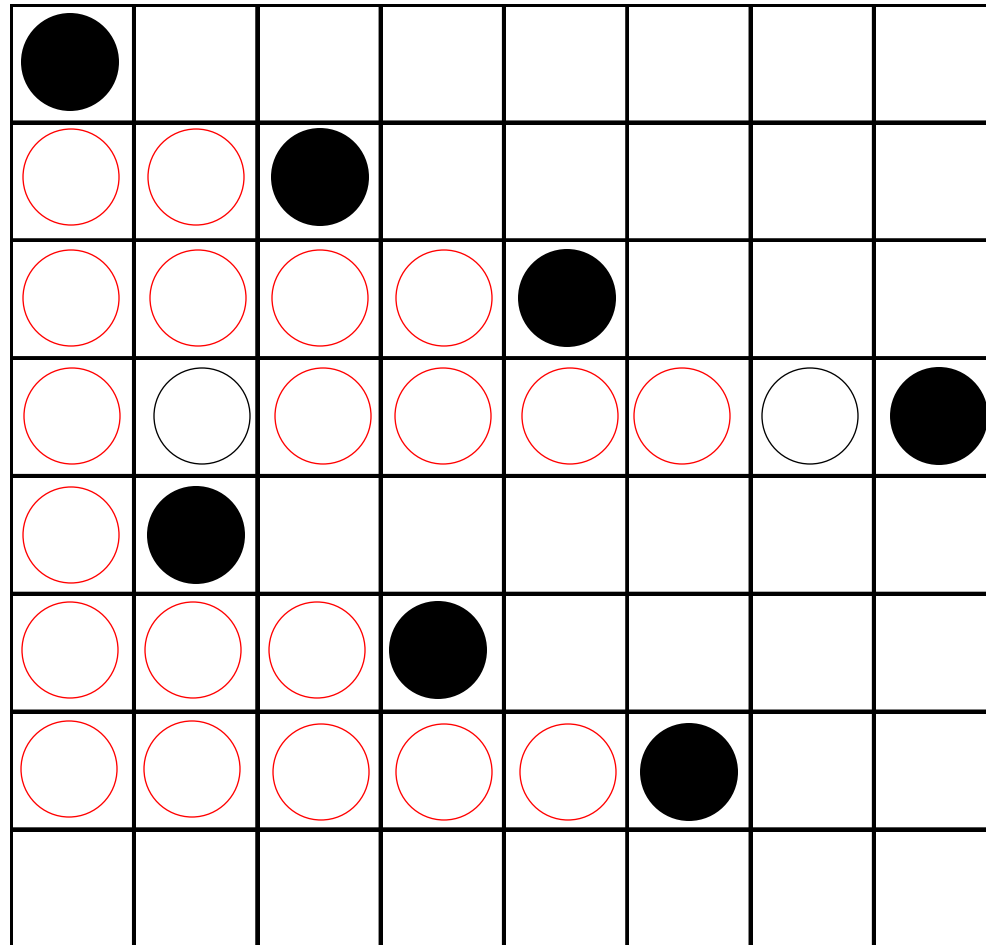
Retrocesso



Testes $206+1+3+2+5 = 217$

Retrocessos 9

Retrocesso

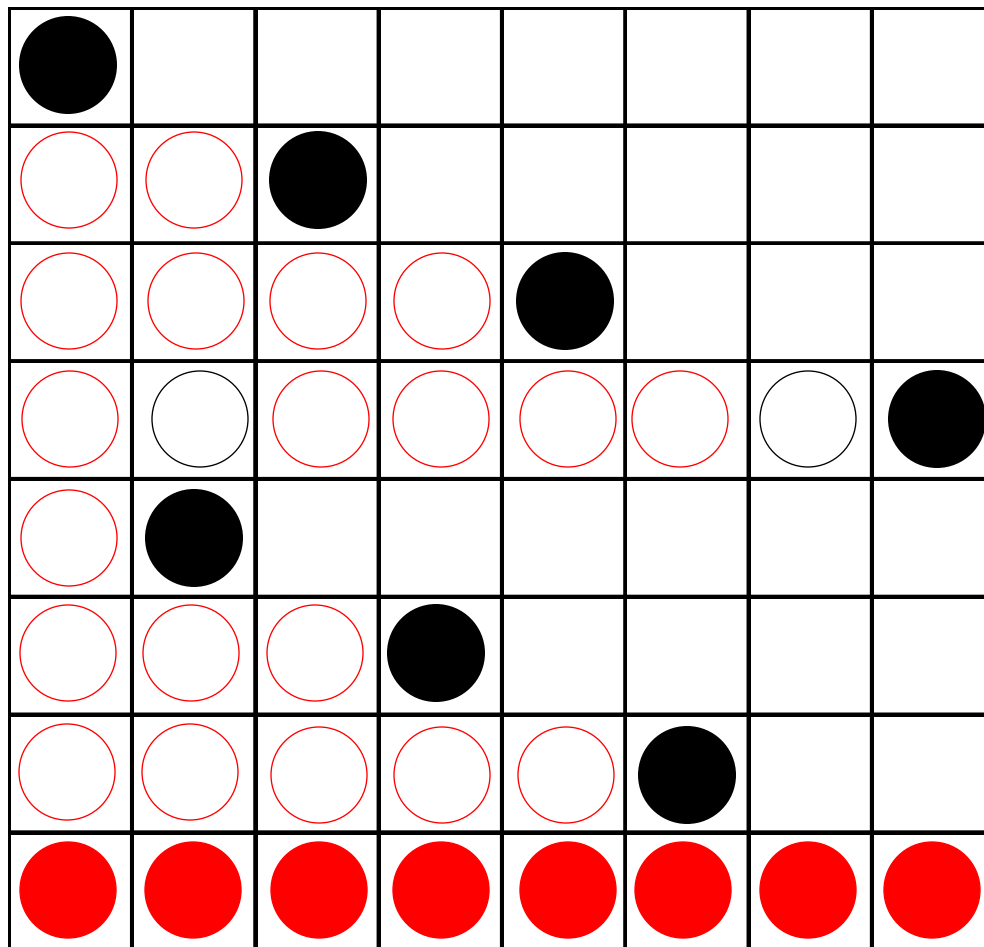


Testes $217+1+5+2+5+3+6 = 239$

Retrocessos 9

Retrocesso

Falha
8
Retrocede
7

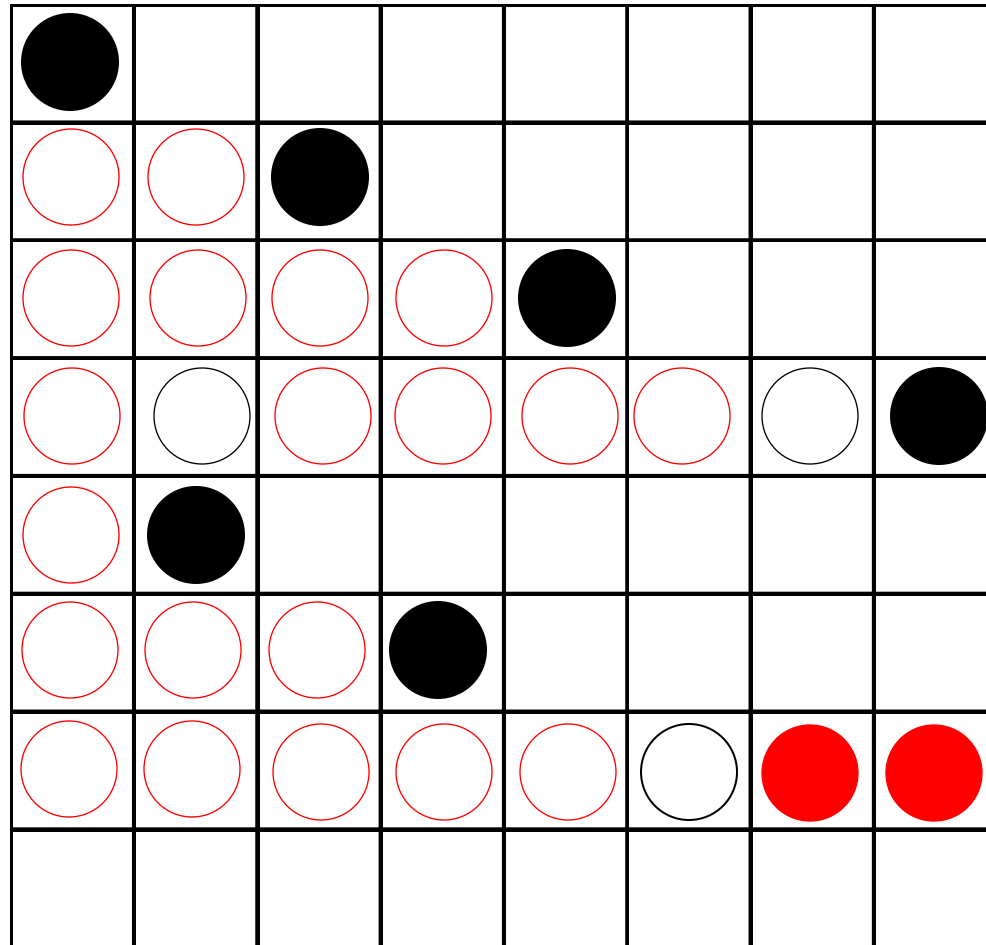


Testes $239+1+5+2+4+3+6+7+7= 274$

Retrocessos $9+1=10$

Retrocesso

Falha
7
Retrocede
6

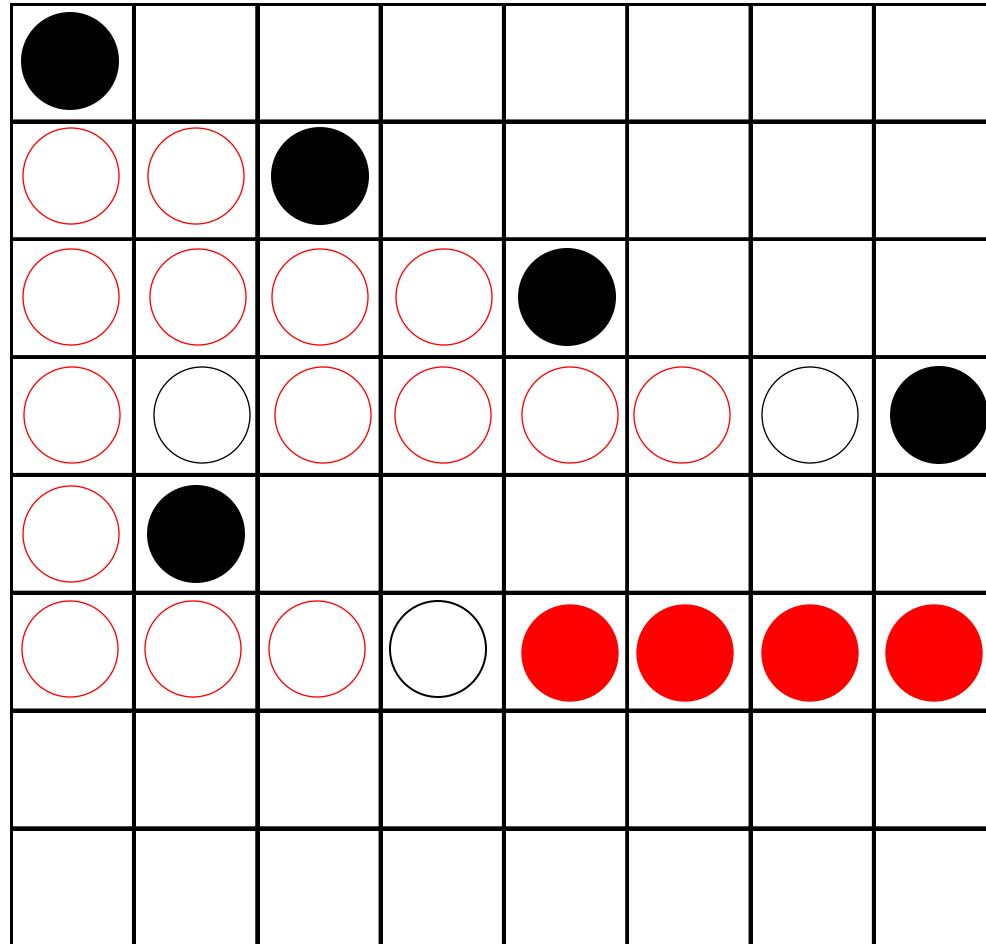


Testes $274+1+2=277$

Retrocessos $10+1=11$

Retrocesso

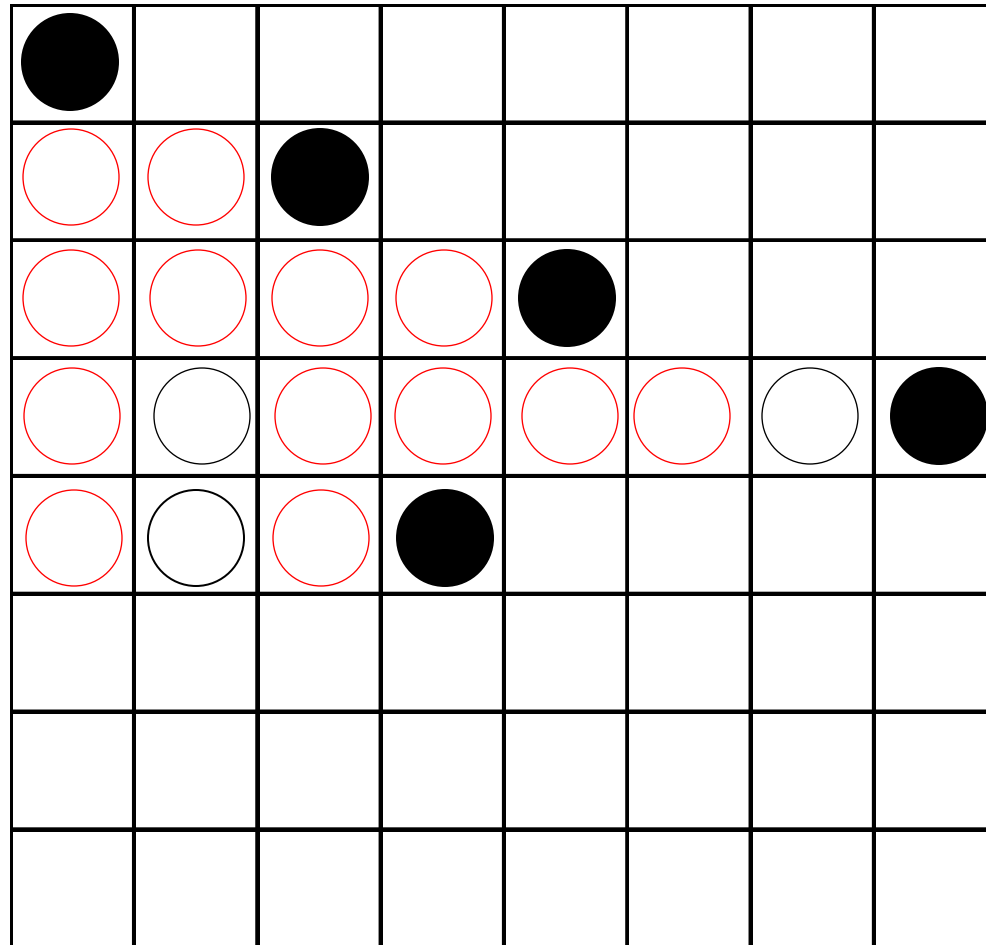
Falha
6
Retrocede
5



Testes $277+3+1+2+3= 286$

Retrocessos $11+1=12$

Retrocesso

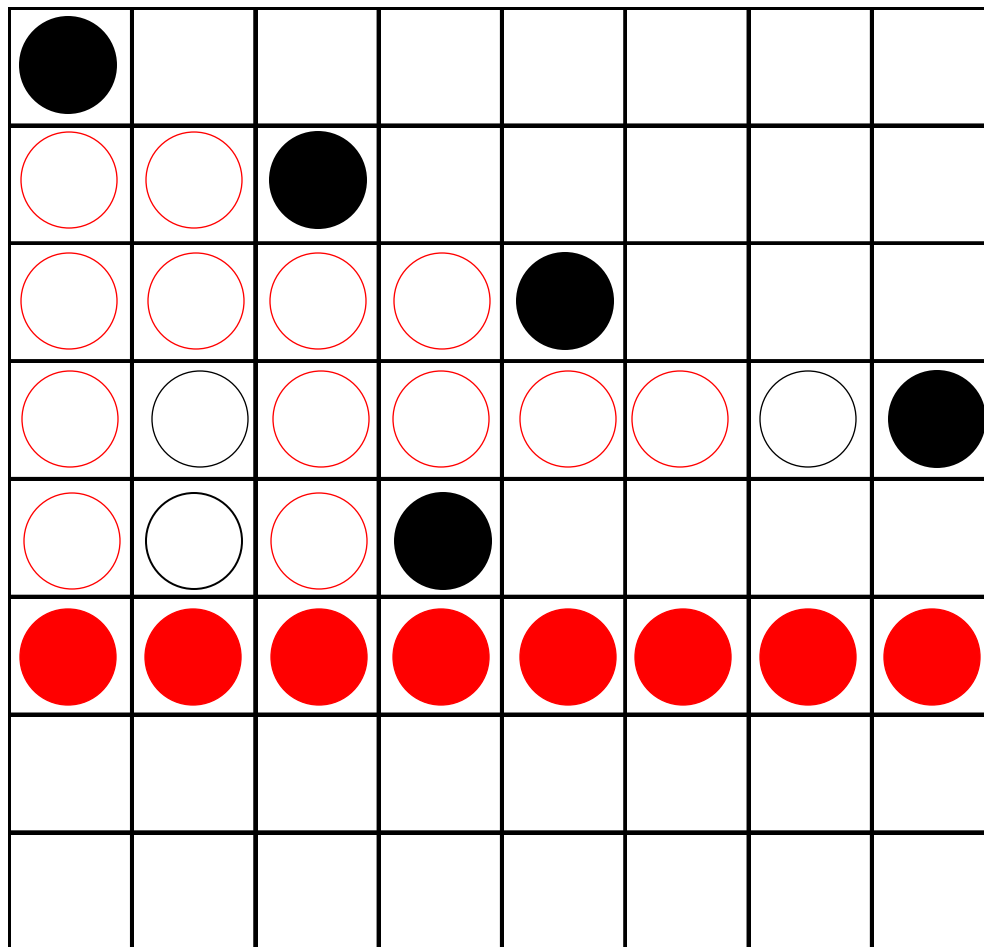


Testes 286+2+4= 292

Retrocessos 12

Retrocesso

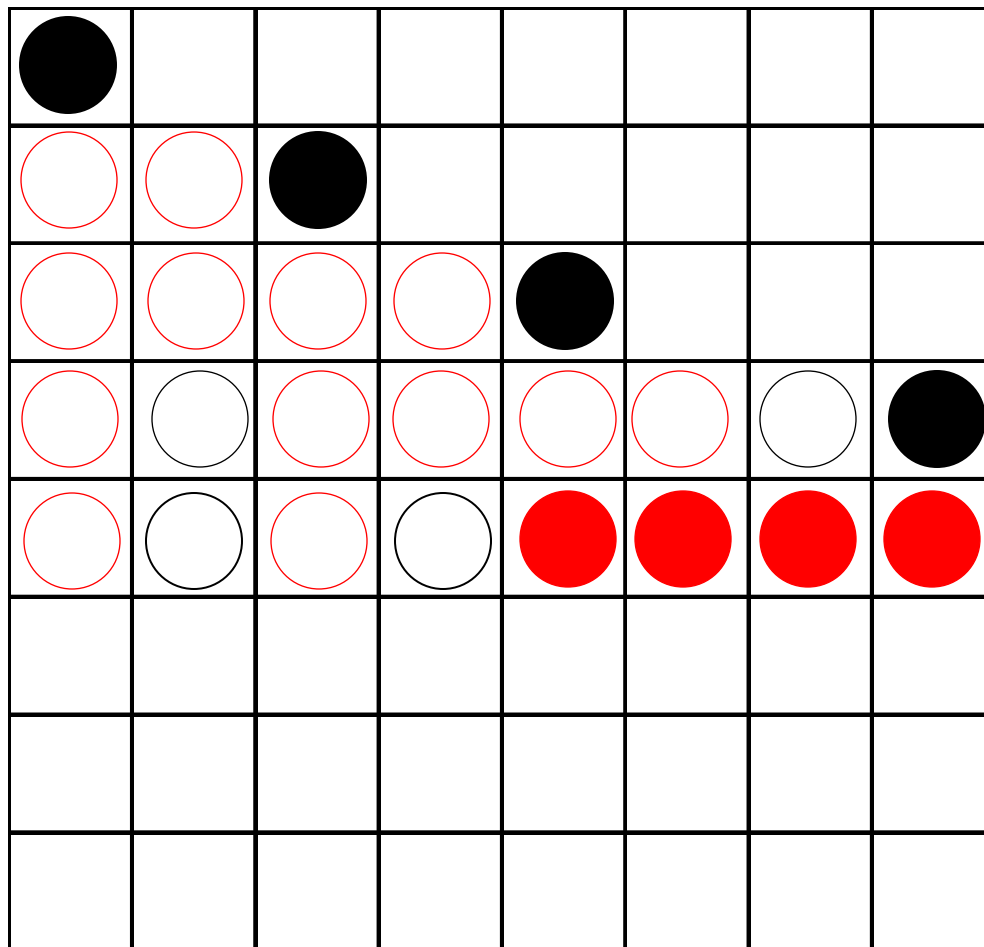
Falha
6
Retrocede
5



Testes $292+1+3+2+5+3+1+2+3= 312$ Retrocessos $12+1=13$

Retrocesso

Falha
5
Retrocede
4 e 3



Testes $312+1+2+3+4= 322$

Retrocessos $13+2=15$

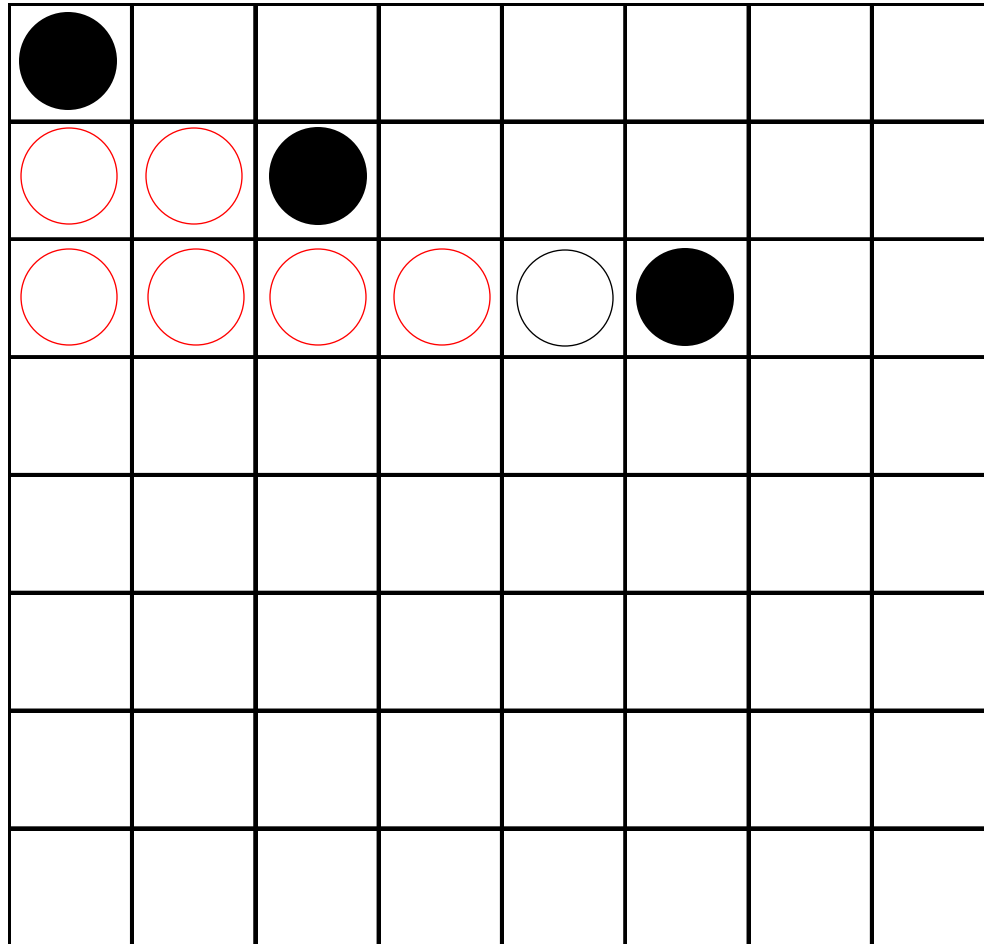
Retrocesso

$$X_1=1$$

$$X_2=3$$

$$X_3=5$$

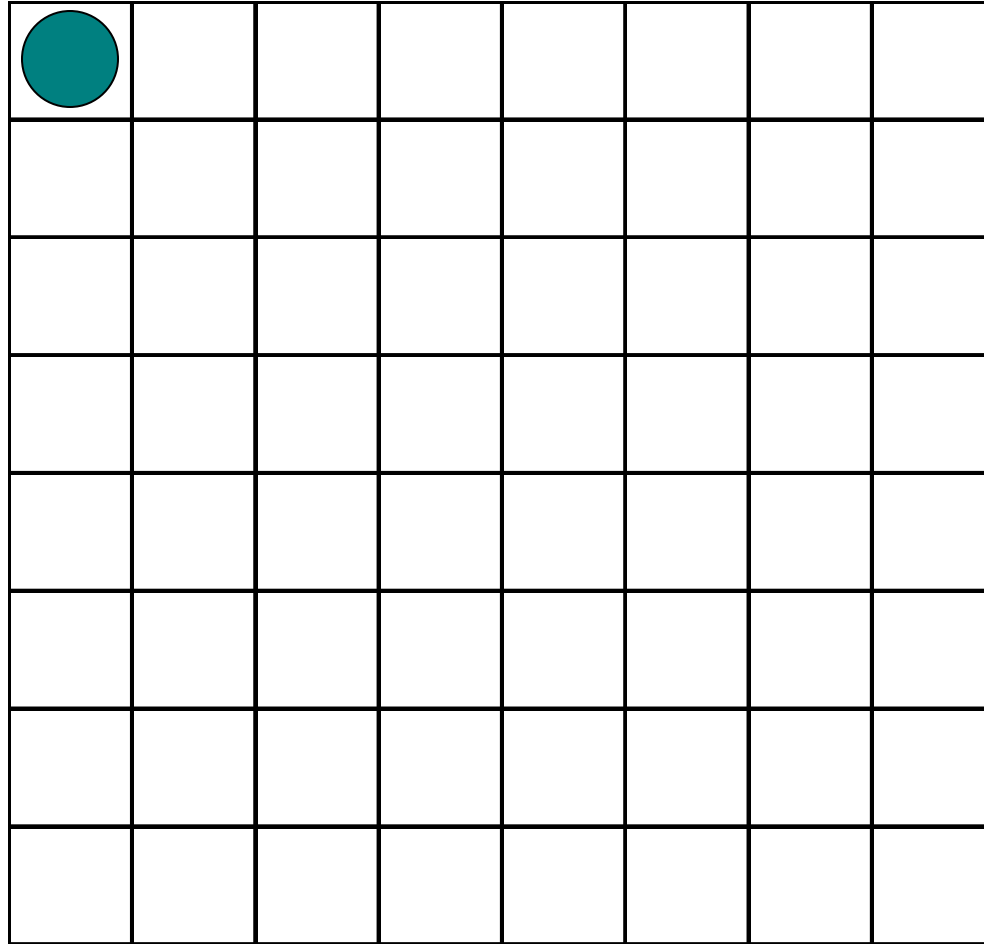
Impossível



Testes 322 + 2 = 324

Retrocessos 15

Propagação




Testes 0

Retrocessos 0

Propagação



$Q1 \neq Q2, \quad L1+Q1 \neq L2+Q2, \quad L1+Q2 \neq L2+Q1.$

							
1	1						
1		1					
1			1				
1				1			
1					1		
1						1	
1							1

Testes $8 * 7 = 56$

Retrocessos 0




Propagação

							
1	1						
1	2	1	2				
1		2	1	2			
1		2		1	2		
1		2			1	2	
1		2				1	2
1		2					1

Testes $56 + 6 * 6 = 92$

Retrocessos 0

Propagação





							
1	1						
1	2	1	2				
1		2	1	2	3		
1		2		1	2	3	
1	3	2		3	1	2	3
1		2		3		1	2
1		2		3			1

Testes $92 + 4 * 5 = 112$

Retrocessos 0

Propagação 2





X_6 só pode
tomar o valor 4

							
1	1						
1	2	1	2				
1		2	1	2	3		
1		2		1	2	3	
1	3	2		3	1	2	3
1		2		3		1	2
1		2		3			1

Testes $92 + 4 * 5 = 112$

Retrocessos 0

Propagação 2





							
1	1						
1	2	1	2				
1	6	2	1	2	3		
1		2	6	1	2	3	
1	3	2		3	1	2	3
1		2	6	3		1	2
1	6	2	6	3	6		1

Testes $112+3+3+3+4 = 125$




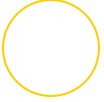

Retrocessos 0

Propagação 2

X_8 só pode
tomar o valor 7

							
1	1						
1	2	1	2				
1	6	2	1	2	3		
1		2	6	1	2	3	
1	3	2		3	1	2	3
1		2	6	3		1	2
1	6	2	2	3	6		1




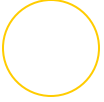

Propagação 2

							
1	1						
1	2	1	2				
1	6	2	1	2	3		
1		2	6	1	2	3	
1	3	2		3	1	2	3
1		2	6	3		1	2
1	6	2	2	3	6		1

Testes 125

Retrocessos 0

Propagação 2




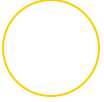
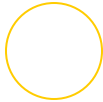
							
1	1						
1	2	1	2				
1	6	2	1	2	3	8	
1		2	6	1	2	3	
1	3	2		3	1	2	3
1		2	6	3	8	1	2
1	6	2	2	3	6		1

Testes $125+2+2+2=131$







Retrocessos 0

Propagação 2







X_4 só pode
tomar o valor 8

							
1	1						
1	2	1	2				
1	6	2	1	2	3	8	
1		2	6	1	2	3	
1	3	2		3	1	2	3
1		2	6	3	8	1	2
1	6	2	2	3	6		1

Propagação 2

							
1	1						
1	2	1	2				
1	6	2	1	2	3	8	
1		2	6	1	2	3	
1	3	2		3	1	2	3
1		2	6	3	8	1	2
1	6	2	2	3	6		1

Propagação 2





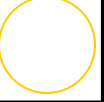

							
1	1						
1	2	1	2				
1	6	2	1	2	3	8	
1		2	6	1	2	3	4
1	3	2		3	1	2	3
1		2	6	3	8	1	2
1	6	2	2	3	6		1

Testes $131+2+2=135$








Retrocessos 0

Propagação 2

X_5 só pode
tomar o valor 2

							
1	1						
1	2	1	2				
1	6	2	1	2	3	8	
1		2	6	1	2	3	4
1	3	2		3	1	2	3
1		2	6	3	8	1	2
1	6	2	2	3	6		1








Propagação 2

							
1	1						
1	2	1	2				
1	6	2	1	2	3	8	
1		2	6	1	2	3	4
1	3	2		3	1	2	3
1		2	6	3	8	1	2
1	6	2	2	3	6		1

Testes 135

Retrocessos 0








Propagação 2

							
1	1						
1	2	1	2				
1	6	2	1	2	3	8	
1		2	6	1	2	3	4
1	3	2		3	1	2	3
1	5	2	6	3	8	1	2
1	6	2	2	3	6		1

Testes 135+1=136

Retrocessos 0

Propagação 2








							
1	1						
1	2	1	2				
1	6	2	1	2	3	8	
1		2	6	1	2	3	4
1	3	2		3	1	2	3
1	5	2	6	3	8	1	2
1	6	2	2	3	6		1

Testes 136

Retrocessos 0

Propagação 2

Falha
7
Retrocede
3 !

							
1	1						
1	2	1	2				
1	6	2	1	2	3	8	
1		2	6	1	2	3	4
1	3	2		3	1	2	3
1	5	2	6	3	8	1	2
1	6	2	2	3	6		1

Testes 136

Retrocessos 0+1=1



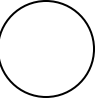

Propagação 2

$$X_1=1$$

$$X_2=3$$

$$X_3=5$$

Impossível

							
1	1						
1	2	1	2				
1		2	1	2	3	3	
1		2	3	1	2	3	3
1		2			1	2	3
1	3	2			3	1	2
1		2			3		1

Testes

136

(324)

Retrocessos

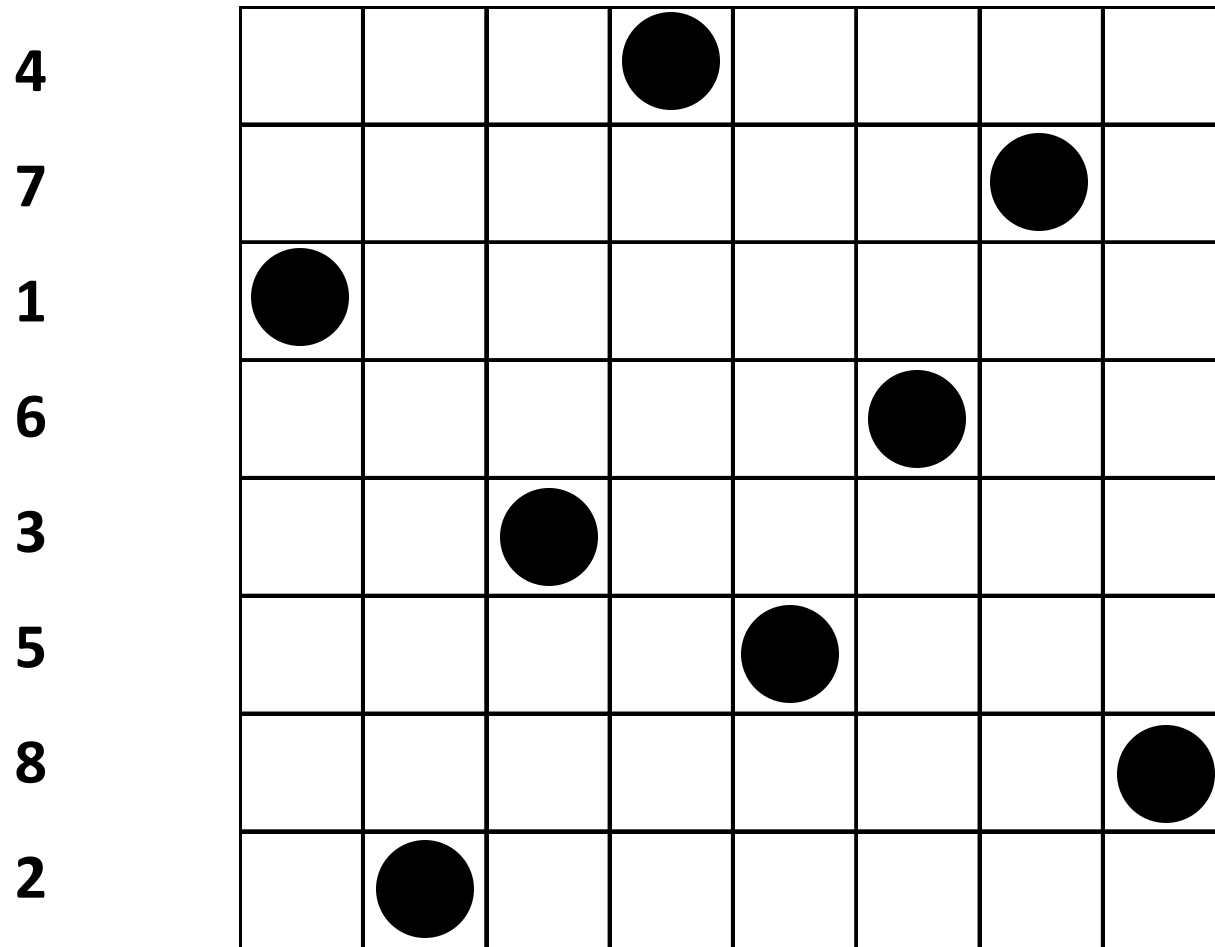
1

(15)

Testes 136

Retrocessos 1

Pesquisa Local - Reparação



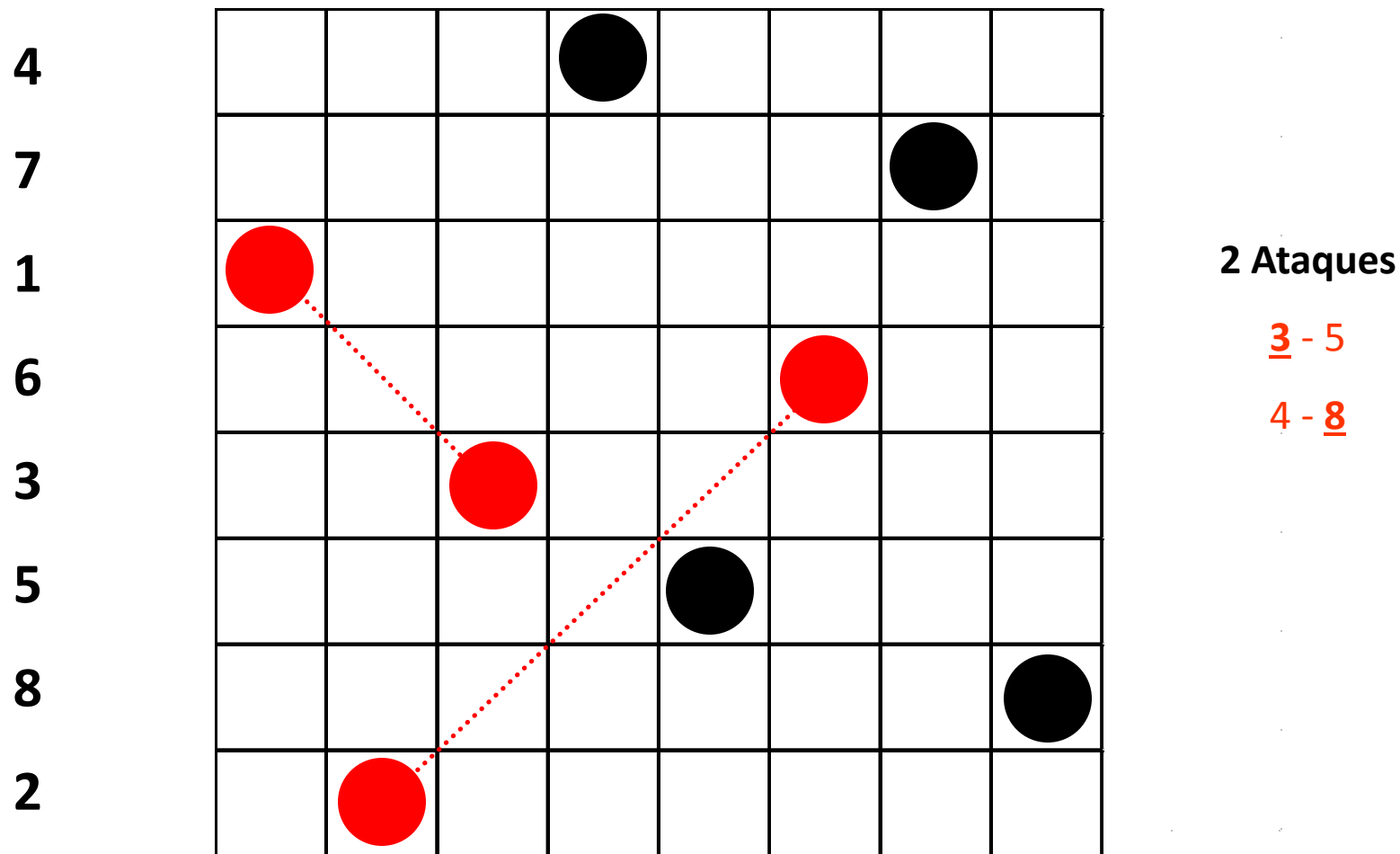
Solução

Permutação

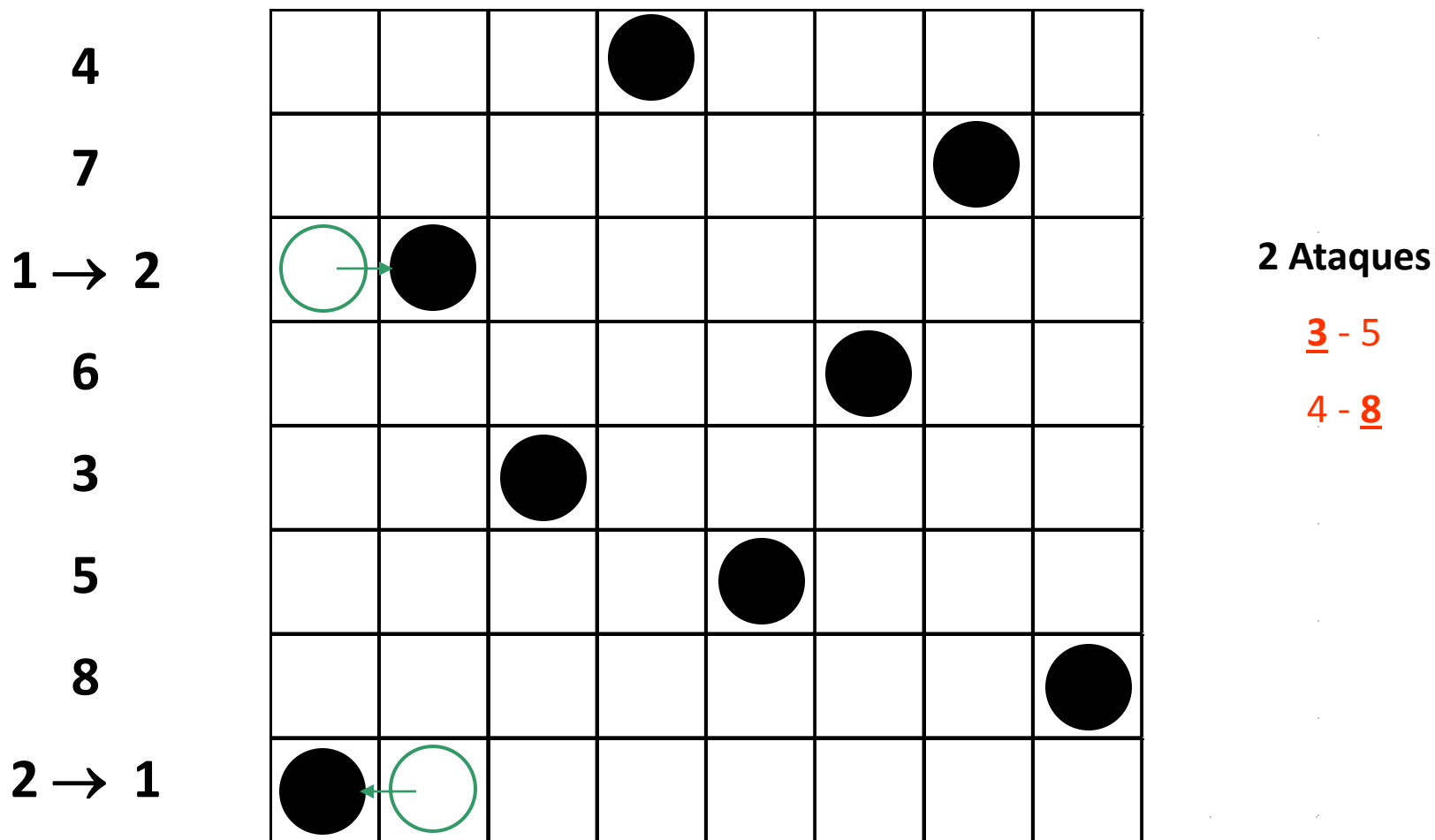
Vizinhança

Troca

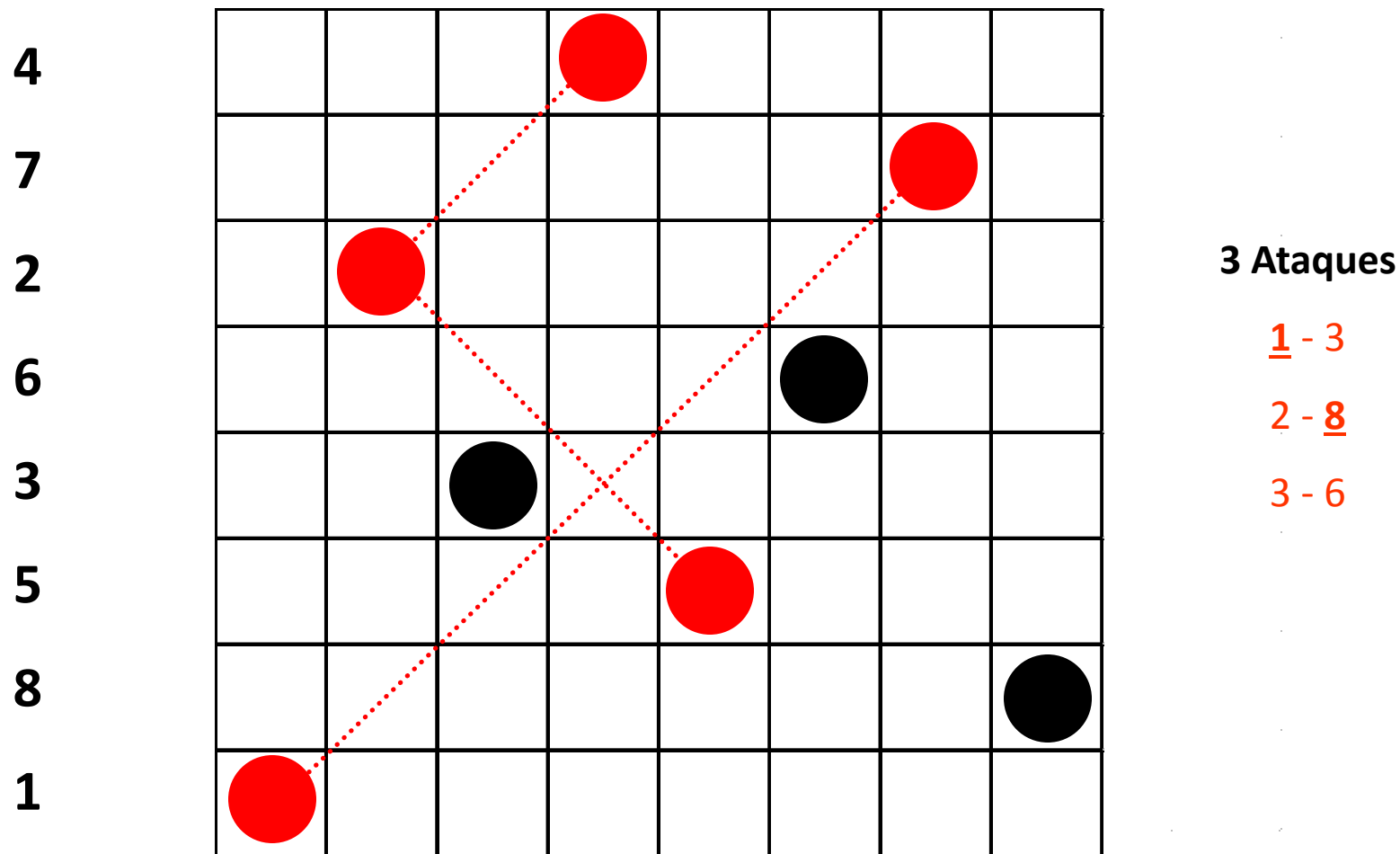
Pesquisa Local - Reparação



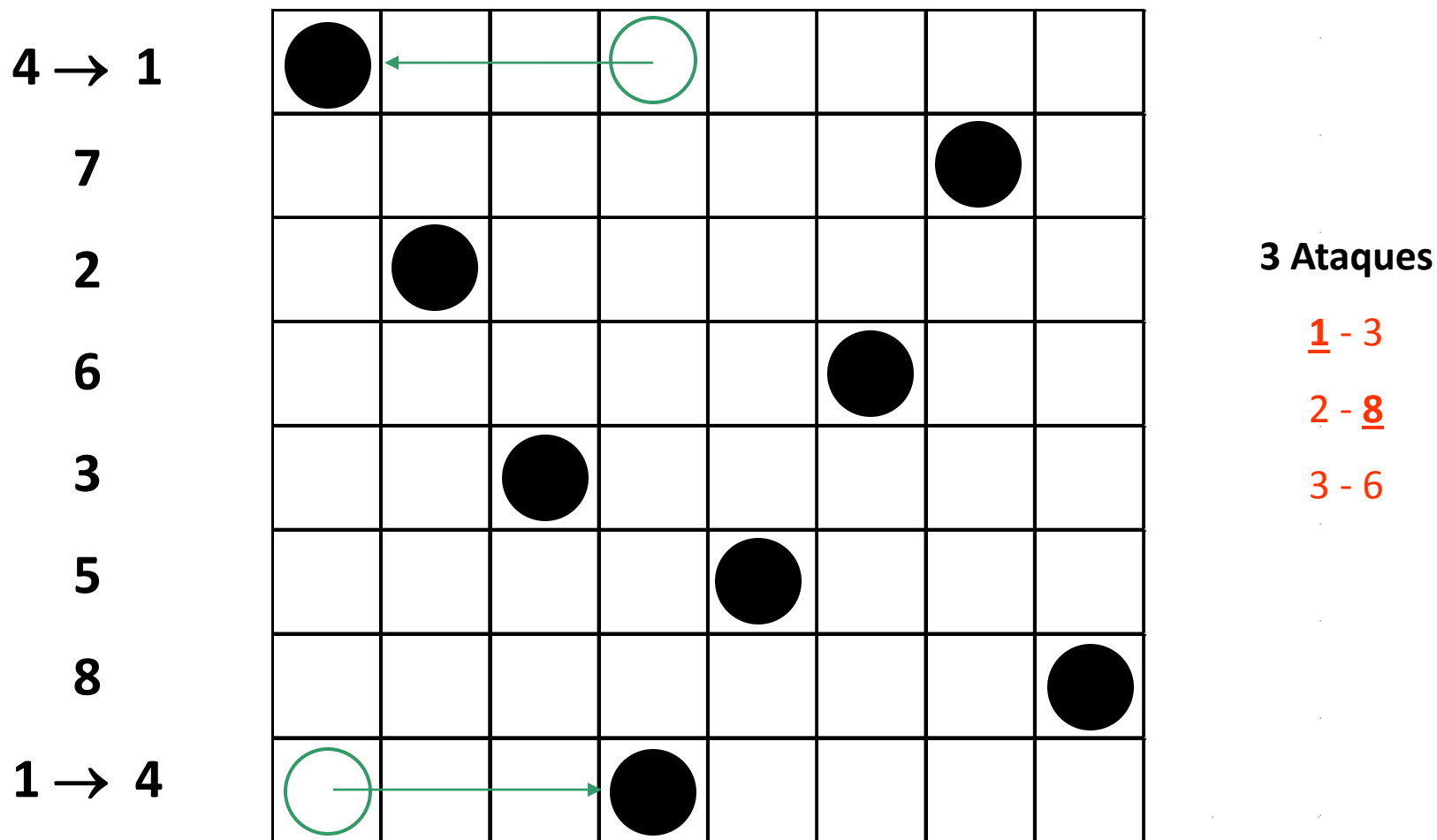
Pesquisa Local - Reparação



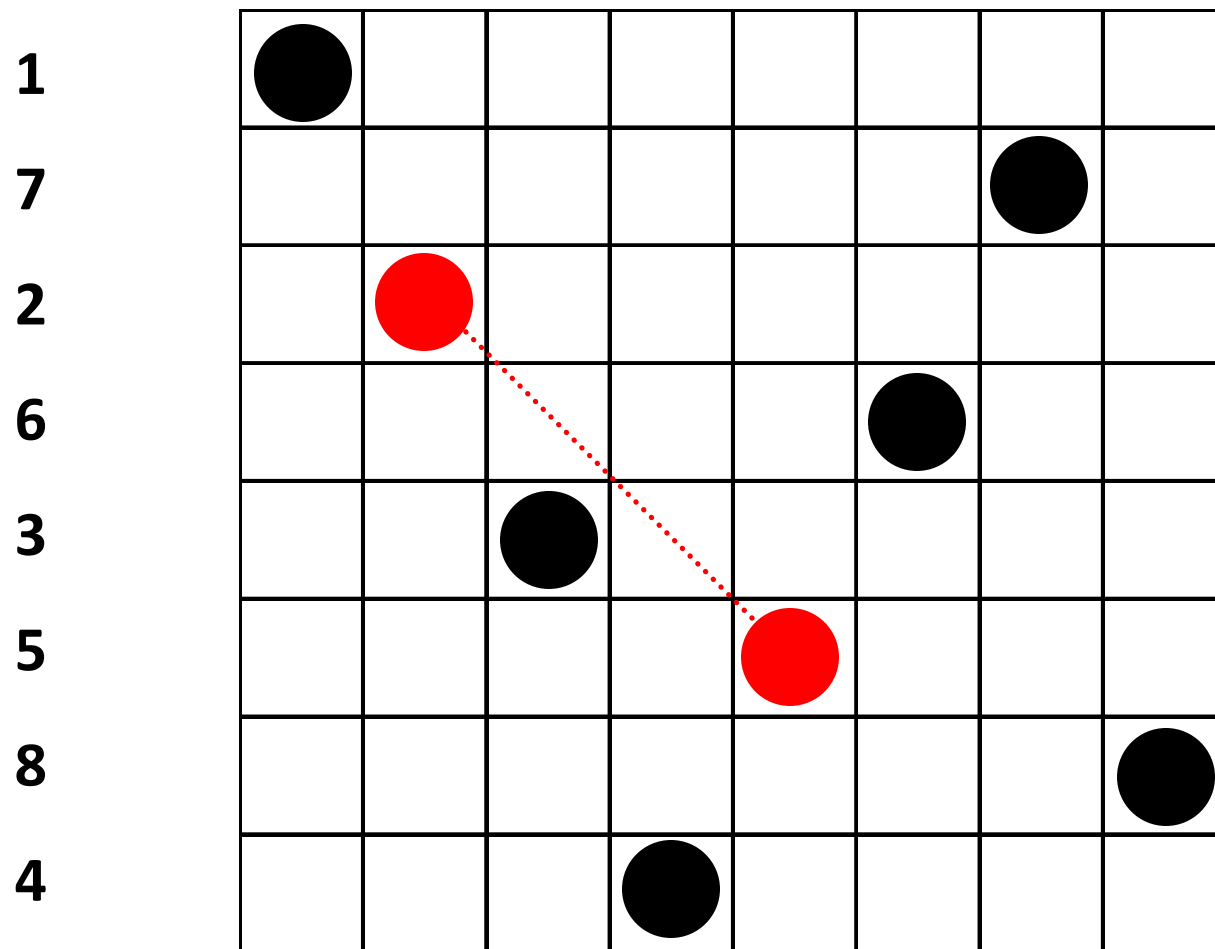
Pesquisa Local - Reparação



Pesquisa Local - Reparação



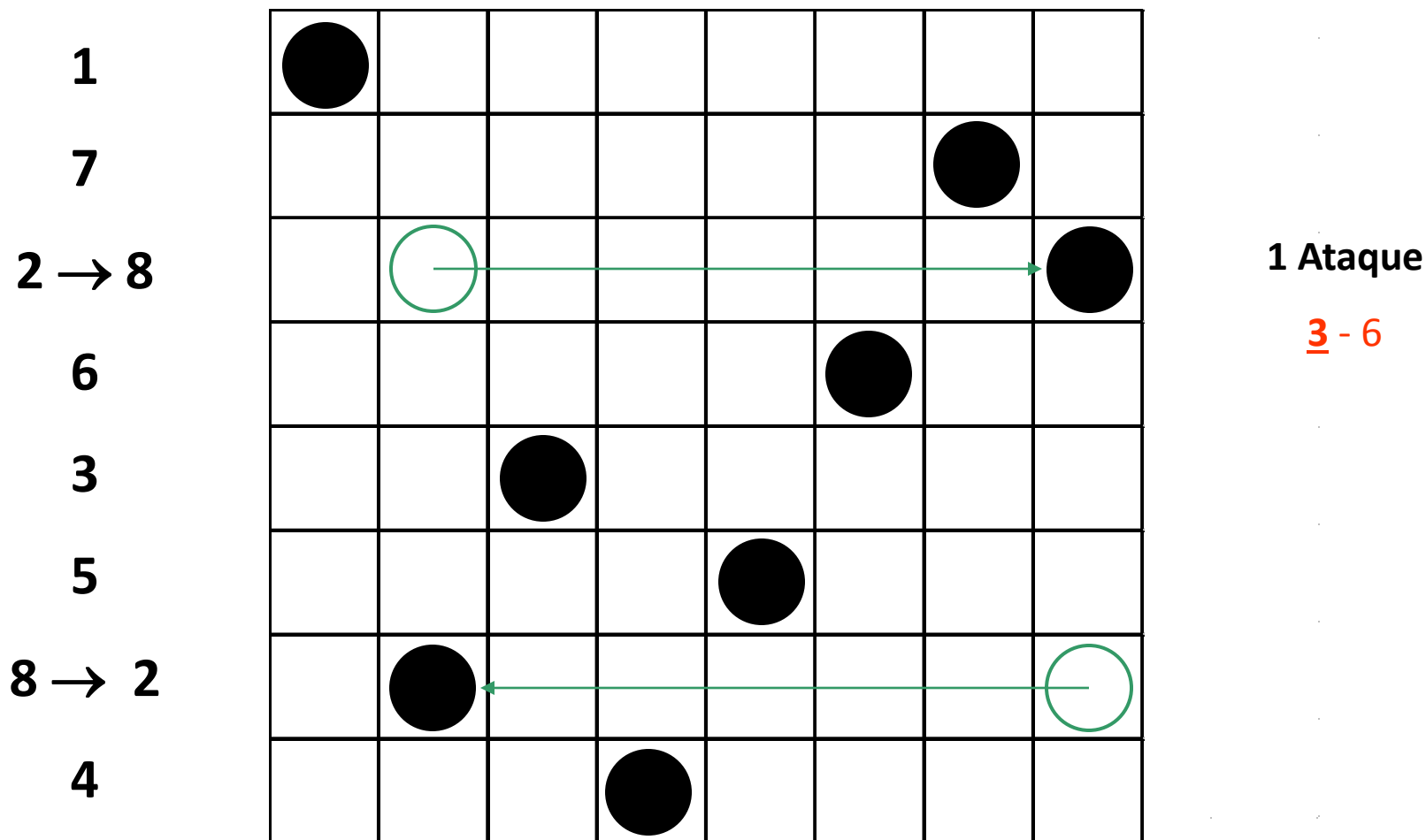
Pesquisa Local - Reparação



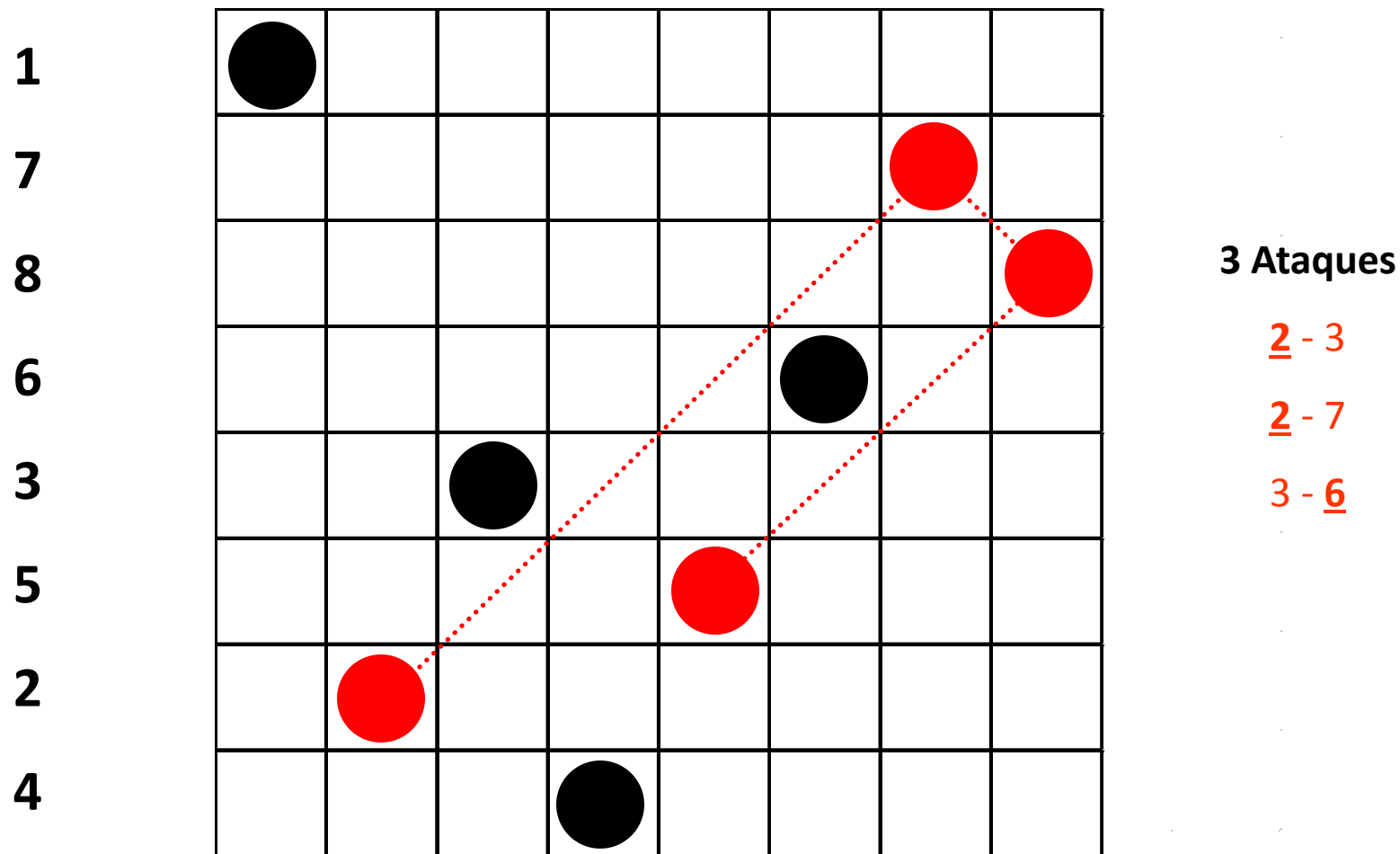
1 Ataque

3 - 6

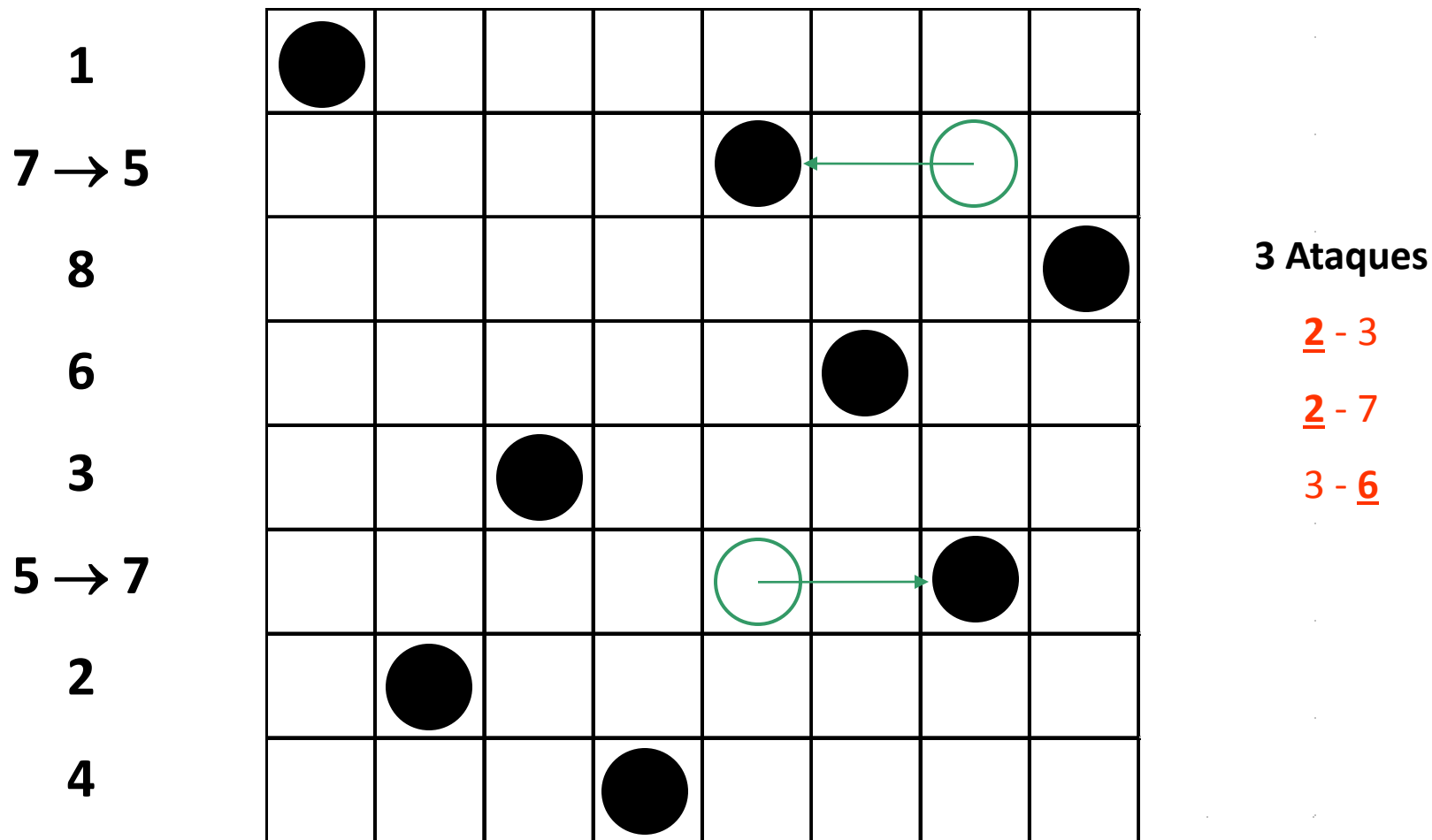
Pesquisa Local - Reparação



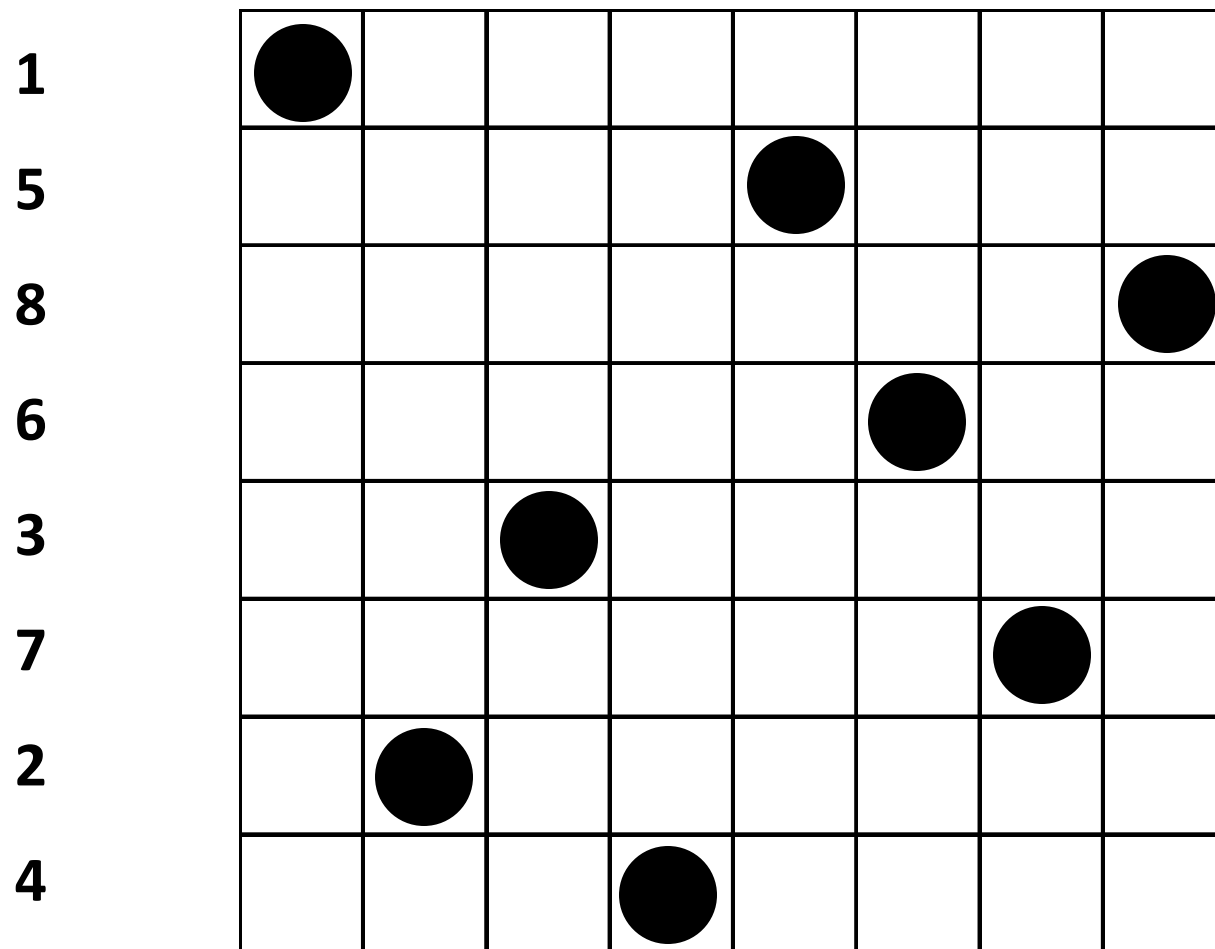
Pesquisa Local - Reparação



Pesquisa Local - Reparação



Pesquisa Local - Reparação



0 Ataques

Programação por Restrições

- Foco na propagação de restrições associada a pesquisa (“*search*”) com retrocesso (“*backtracking*”)
- Extensão ao Prolog:
 - CLP: *Constraint Logic Programming*
 - PLR: Programação em Lógica com Restrições
 - Sistemas: **SICStus**, CHIP, ECLiPSe, ...
- Outros Sistemas:
 - C++: IBM ILOG CP Optimizer, ...
 - Java: OptaPlanner, ...
 - ...

Programação em Lógica com Restrições

5. RESTRIÇÕES EM BOOLEANOS, REAIS E RACIONAIS

Adaptado de:

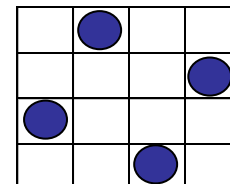
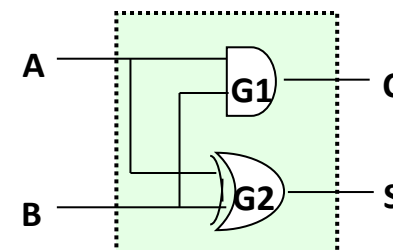
Pedro Barahona, *Página da Disciplina de Programação por Restrições, Ano Lectivo 2003/04*, disponível em:

<http://ssdi.di.fct.unl.pt/~pb/cadeiras/pr/0304/index.htm>

[consultado em Setembro de 2004]

Restrições Booleanas

- O domínio dos Booleanos (ou variáveis 0/1) tem especial aplicação em aplicações:
 - Envolvendo circuitos digitais
 - Exemplo: Circuito semi-somador
 - Em problemas envolvendo escolhas binárias
 - Exemplo: Rainhas / Alocação de Trabalhadores
 - Em problemas que envolvam conjuntos



Restrições Lineares sobre Reais/Racionais

- Muitos problemas podem ser modelados através de variáveis reais (ou racionais), denominadas variáveis de decisão:
 - Típico em Investigação Operacional
 - Todas as restrições sobre essas variáveis são lineares
- Geralmente pretendem-se soluções que otimizem uma função (linear) objetivo
 - Otimização condicionada
- Exemplos:
 - Gestão da produção
 - Redes de distribuição
 - Transportes
 - Alocação de recursos

Restrições Lineares

Exemplo: Gestão da Produção

- **Dados:**
 - Um conjunto de itens P_1, \dots, P_n que se pretende produzir
 - Um conjunto de recursos R_1, \dots, R_m disponíveis para os produzir
 - Uma matriz A , cujos elementos a_{ij} representam a quantidade do recurso i necessário para produzir uma unidade do item j
 - Um vetor C , cujos elementos c_j representam o lucro obtido por cada unidade do item j produzido
 - Um vetor B , cujos elementos b_i representam a quantidade máxima do recurso i que pode ser utilizado
- **Objetivo:**
 - Determinar a quantidade a produzir de cada item j de modo a maximizar o Lucro

Restrições Lineares

Exemplo: Gestão da Produção

- Restrições base:

- Designando por X_i a quantidade do item P_i produzido, o problema de não sobre-utilização dos recursos é modelado por:

$$a_{11} X_1 + a_{12} X_2 + \dots + a_{1n} X_n \leq b_1$$

...

$$a_{m1} X_1 + a_{m2} X_2 + \dots + a_{mn} X_n \leq b_m$$

- Exemplo: Sendo necessários 3 minutos de uma máquina para fabricar 1 sapato e 4 minutos para 1 bota, representando por NS e NB o número de sapatos e botas e estando a máquina disponível 4 horas temos:

$$3 \text{ NS} + 4 \text{ NB} \leq 4 \cdot 60$$

Restrições Lineares

Exemplo: Gestão da Produção

- **Satisfação** (de um dado lucro):

- Para modelizar que um dado lucro L é atingido, adiciona-se ao modelo anterior a restrição:

$$c_1 X_1 + c_2 X_2 + \dots + c_n X_n \geq L$$

- Exemplo, se se obtém um lucro de 5€ por cada sapato e 6€ por cada bota vendida, pretender que o lucro seja pelo menos de 3000€ é modelada pela restrição:

$$5 NS + 6 NB \geq 3000$$

- **Otimização** (do lucro obtido):

- Para modelizar a **maximização** do lucro obtido, adiciona-se ao modelo inicial a função objetivo L (a maximizar):

$$\text{Max } L = c_1 X_1 + c_2 X_2 + \dots + c_n X_n$$

- Exemplo: Considerando os lucros referidos:

$$\text{Max } L = 5 NS + 6 NB$$

Restrições Lineares

Exemplo: Redes de Distribuição

- **Dados:**
 - Um conjunto de nós P_1, \dots, P_n , dos quais P_1 é emissor e P_n o recetor
 - Uma matriz M , cujos elementos $m_{i,j}$ representam a capacidade máxima da ligação entre os nós i e j
 - Uma matriz C , cujos elementos $c_{i,j}$ representam o custo da transmissão entre os nós i e j
- **Objetivo:**
 - Avaliar o fluxo de informação que a rede é capaz de transmitir entre os nós emissor (P_1) e recetor (P_n)

Restrições Lineares

Exemplo: Redes de Distribuição

- **Restrições base:**

- Designando por $X_{i,j}$ a quantidade debitada entre os nós i e j , as restrições de capacidade são:

$$X_{1,1} \leq m_{1,1} \quad \% \text{ Em geral} =$$

...

$$X_{m,n} \leq m_{m,n}$$

- As restrições de igualdade de fluxo de entrada e saída em todos os k nós (exceto nos nós P_1 e P_n) podem ser modelizadas por

$$X_{1,2} + X_{2,2} + X_{3,2} + \dots + X_{n,2} = X_{2,1} + X_{2,2} + \dots + X_{2,n}$$

$$X_{1,3} + X_{2,3} + X_{3,3} + \dots + X_{n,3} = X_{3,1} + X_{3,2} + \dots + X_{3,n}$$

...

$$X_{1,k} + X_{2,k} + X_{3,k} + \dots + X_{n,k} = X_{k,1} + X_{k,2} + \dots + X_{k,n}$$

Restrições Lineares

Exemplo: Redes de Distribuição

- **Satisfação** (de um fluxo F entre P_1 e P_n):
 - Verificação de que é possível transmitir um determinado fluxo igual ou superior a F entre os nós emissor (P_1) e recetor (P_n) pode ser modelizada pela restrição:

$$X_{1,2} + X_{1,3} + X_{1,4} + \dots + X_{1,n} \geq F$$
$$(\text{ou } X_{1,n} + X_{2,n} + X_{3,n} + \dots + X_{n-1,n} \geq F)$$

- **Otimização** (do custo total):
 - Para modelizar o máximo fluxo que a rede é capaz de transmitir entre os nós P_1 e P_n , adiciona-se uma função objectivo F :

$$\text{Max } F = X_{1,2} + X_{1,3} + X_{1,4} + \dots + X_{1,n}$$

Restrições Lineares: Formalização

- Os problemas de **satisfação** de restrições lineares sobre variáveis (de decisão) X_i reais (ou racionais, se todos os parâmetros a_{ij} são números racionais) têm sempre a forma:

$$a_{11} X_1 + a_{12} X_2 + \dots + a_{1n} X_n \text{ op}_1 b_1$$

$$a_{21} X_1 + a_{22} X_2 + \dots + a_{2n} X_n \text{ op}_2 b_2$$

...

$$a_{m1} X_1 + a_{m2} X_2 + \dots + a_{mn} X_n \text{ op}_m b_m$$

- em que $\text{op}_1.. \text{op}_m$ são quaisquer do conjunto $\{\leq, =, \geq, <, >, \neq\}$

- Se se pretender a **otimização** das variáveis de decisão, inclui-se a otimização de uma função (linear) objetivo F:

$$\text{Opt } F = c_1 X_1 + c_2 X_2 + \dots + c_n X_n$$

- em que Opt usualmente $\in \{\text{Maximizar}, \text{Minimizar}\}$

Restrições Lineares: Interpretação Geométrica

- Dado um espaço com n dimensões, a restrição

$$a_{i1} X_1 + a_{i2} X_2 + \dots + a_{in} X_n = b_i$$

define uma região admissível, correspondente aos pontos de um **híper-plano** desse espaço

- Como casos particulares temos um espaço tridimensional ($n=3$), em que o híper-plano corresponde ao **plano** usual, e um espaço bidimensional (ou vulgar plano, $n=2$) em que o híper-plano se reduz a uma **reta**
- A restrição \neq define uma região admissível que corresponde a todos os pontos do espaço n -dimensional, exceto os pontos do híper-plano

Restrições Lineares: Interpretação Geométrica

- As restrições \leq e \geq definem regiões admissíveis do tipo **semi-híper-espaço** limitado pelo correspondente hiper-plano, incluído na região admissível
 - Com $n=3$ temos um **semi-espaço** e com $n=2$ um **semi-plano**. As restrições $<$ e $>$ definem regiões semelhantes, excluindo a fronteira
- O conjunto de restrições define, num espaço a n dimensões, um **híper-poliedro** (**poliedro** com $n=3$ e **polígono** com $n=2$) correspondente à intersecção das m regiões admissíveis
- Sem restrições \neq , a região admissível é **convexa**

Restrições Lineares: Interpretação Geométrica

- Em problemas de otimização, a função

$$C = c_1 X_1 + c_2 X_2 + \dots + c_n X_n$$

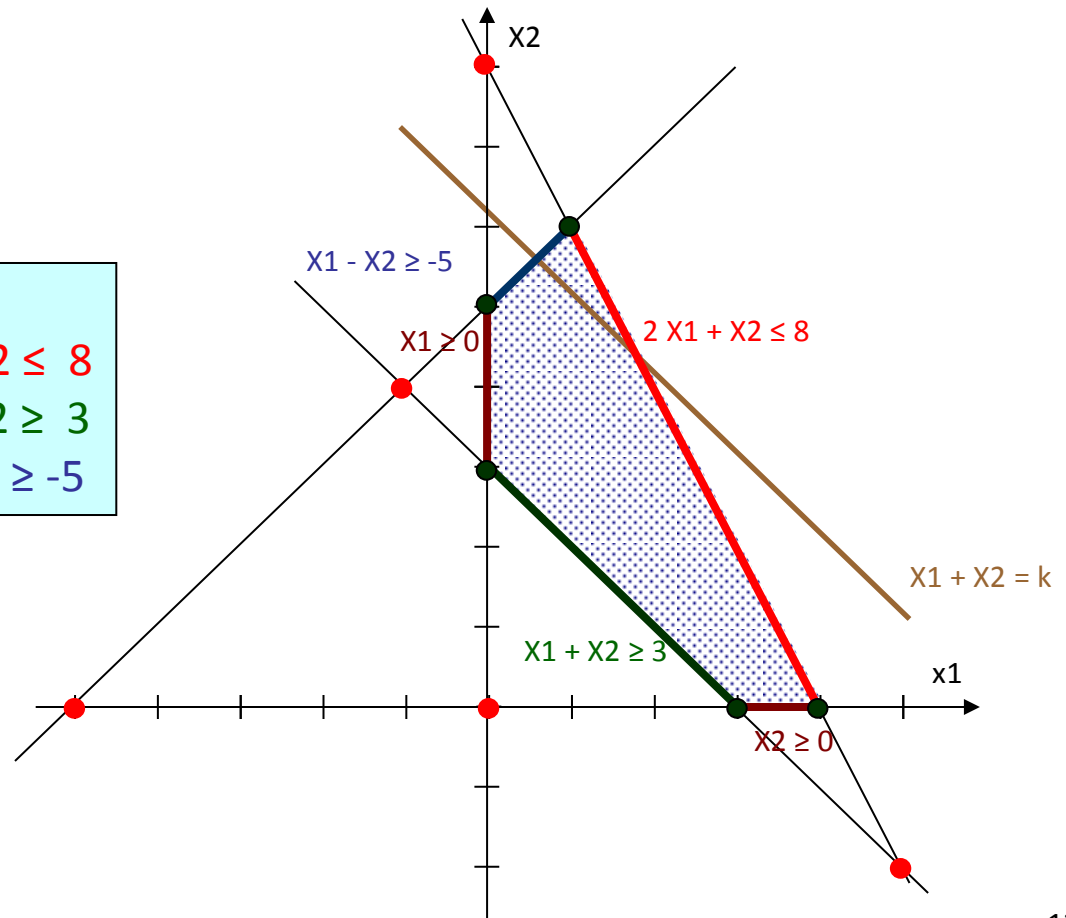
define uma família de hiper-planos paralelos (um para cada valor de C)

- O max/min dessa função na região admissível corresponde ao “último” ponto em que um hiper-plano da função objetivo toca a região admissível quando se desloca para valores crescentes/decrescentes de C
- No caso em que o vértice de ótimo pertence a uma aresta ou um lado do hiper-poliedro paralelos à família de hiper-planos de otimização, haverá em geral mais do que um ponto ótimo, já que a função objetivo tem o mesmo valor em todos os pontos dessas arestas e lados

Restrições Lineares: Interpretação Geométrica

- Problema simples:
 - Maximizar $X_1 + X_2$

$$\begin{array}{ll}\max & X_1 + X_2 \\ \text{Suj.} & 2X_1 + X_2 \leq 8 \\ & X_1 + X_2 \geq 3 \\ & X_1 - X_2 \geq -5\end{array}$$



Programação em Lógica com Restrições

6. DEFINIÇÕES FORMAIS E CONCEITOS

Adaptado de:

Pedro Barahona, *Página da Disciplina de Programação por Restrições, Ano Lectivo 2003/04*, disponível em:

<http://ssdi.di.fct.unl.pt/~pb/cadeiras/pr/0304/index.htm>

[consultado em Setembro de 2004]

Domínios Finitos: Variáveis/Domínios

- Domínio de uma Variável – $dom(X)$ ou D_x
 - Conjunto finito de valores que podem ser atribuídos a essa variável
- Exemplo: N-Rainhas:
 - Modelizado através de:
 - n variáveis, X_1 a X_n , com domínio 1 a n
 - $dom(X_i) = \{1, 2, \dots, n\}$ ou $X_i :: 1..n$

Atribuição de Valor a uma Variável

- Atribuição de Valores a Variáveis:
 - *Label* (Etiqueta)
 - Um *label* é um par Variável-Valor, onde Valor é um dos elementos do domínio da variável
- Solução parcial em que algumas das variáveis já têm valores atribuídos:
 - *Compound Label* (Etiqueta Composta)
 - Conjunto de *labels* incluindo variáveis distintas

Restrições

- Restrição

- Dado um conjunto de variáveis, uma restrição limita as etiquetas compostas para essas variáveis
- Uma restrição R_{ijk} envolvendo as variáveis X_i , X_j e X_k definirá um subconjunto do produto cartesiano dos domínios das variáveis envolvidas
 - $R_{ijk} \subseteq \text{dom}(X_i) \times \text{dom}(X_j) \times \text{dom}(X_k)$

Restrições

- Dada uma restrição C , o conjunto de variáveis envolvidas é descrito por $\text{Vars}(C)$
- Simetricamente, o conjunto de restrições em que a variável X participa é denotado por $\text{Cons}(X)$
- Uma restrição é uma relação, pelo que $C_{ij} = C_{ji}$
- Na prática, as restrições podem ser especificadas:
 - **Em Extensão** (Explicitamente): através da enumeração de todas as etiquetas compostas admissíveis
 - **Implicitamente**: através de um predicado ou procedimento que determine as etiquetas compostas

Restrições

- Por exemplo, no problema das 4-Rainhas a restrição que envolve as variáveis **X1** e **X3** pode ser definida:

– Em Extensão:

$$C13 = \{ \quad \{X1-1, X3-2\}, \{X1-1, X3-4\}, \{X1-2, X3-1\}, \{X1-2, X3-3\}, \\ \{X1-3, X3-2\}, \{X1-3, X3-4\}, \{X1-4, X3-1\}, \{X1-4, X3-3\} \quad \}$$

- Ou de forma mais simples:

$$C13 = \{ \quad \langle 1,2 \rangle, \langle 1,4 \rangle, \langle 2,1 \rangle, \langle 2,3 \rangle, \\ \langle 3,2 \rangle, \langle 3,4 \rangle, \langle 4,1 \rangle, \langle 4,3 \rangle \quad \}$$

– Ou implicitamente através da fórmula respetiva:

$$C13 = (X1 \neq X3) \wedge (X1 \neq X3+2) \wedge (X1 \neq X3-2)$$

Aridade de Restrições

- **Aridade de uma Restrição**
 - A aridade de uma restrição C é o número de variáveis sobre o qual a restrição está definida, ou seja, a cardinalidade do conjunto $\text{Vars}(C)$
- Embora as restrições possam ter qualquer aridade, a aridade mais importante é a binária
- Importância das restrições binárias:
 - Todas as restrições podem ser convertidas em restrições binárias
 - Diversos conceitos e algoritmos são apropriados para restrições binárias

Restrições Binárias

- Conversão para Restrições Binárias:
 - Uma **restrição n-ária** C , definida por k etiquetas compostas nas suas variáveis X_1 a X_n , é **equivalente a n restrições binárias**, B_i , através da adição de uma nova variável Z , cujo domínio é o conjunto 1 a k
- Justificação:
 - Os k *labels* n -ários podem ser ordenados em qualquer ordem
 - Cada uma das restrições binárias B_i relaciona a nova variável Z com a variável X_i
 - O *label* composto $\{X_i-v_i, Z-z\}$ pertence à restrição B_i sse X_i-v_i pertence ao n -ésimo *label* composto que define C

Restrições Binárias

- Exemplo:
 - Dadas as variáveis $X1$, $X2$ e $X3$, com domínio 1 a 3, a restrição ternária seguinte C impõe valores diferentes para as três variáveis e é composto por 6 *labels* compostos:
$$C(X1, X2, X3) = \{ \langle 1,2,3 \rangle, \langle 1,3,2 \rangle, \langle 2,1,3 \rangle, \langle 2,3,1 \rangle, \langle 3,1,2 \rangle, \langle 3,2,1 \rangle \}$$
 - Cada um dos *labels* pode ter um valor associado entre 1 e 6:
 - 1: $\langle 1,2,3 \rangle$, 2: $\langle 1,3,2 \rangle$, ... 6: $\langle 3,2,1 \rangle$
 - As seguintes restrições binárias $B1$ a $B3$, são equivalentes à restrição ternária inicial C :
 - $B1(Z, X1) = \{ \langle 1,1 \rangle, \langle 2,1 \rangle, \langle 3,2 \rangle, \langle 4,2 \rangle, \langle 5,3 \rangle, \langle 6,3 \rangle \}$
 - $B2(Z, X2) = \{ \langle 1,2 \rangle, \langle 2,3 \rangle, \langle 3,1 \rangle, \langle 4,3 \rangle, \langle 5,1 \rangle, \langle 6,2 \rangle \}$
 - $B3(Z, X3) = \{ \langle 1,3 \rangle, \langle 2,2 \rangle, \langle 3,3 \rangle, \langle 4,1 \rangle, \langle 5,2 \rangle, \langle 6,1 \rangle \}$

Satisfação de Restrições

- Um *label* composto **satisfaz uma restrição** se as suas **variáveis são as mesmas** da restrição e se o *label* composto é **membro da restrição**
- Superconjunto de variáveis:
 - Um *label* composto **satisfaz uma restrição** se o seu conjunto de variáveis **contém as variáveis** da restrição e se a **projeção** do *label* composto nestas variáveis é **membro da restrição**

Problema de Satisfação de Restrições

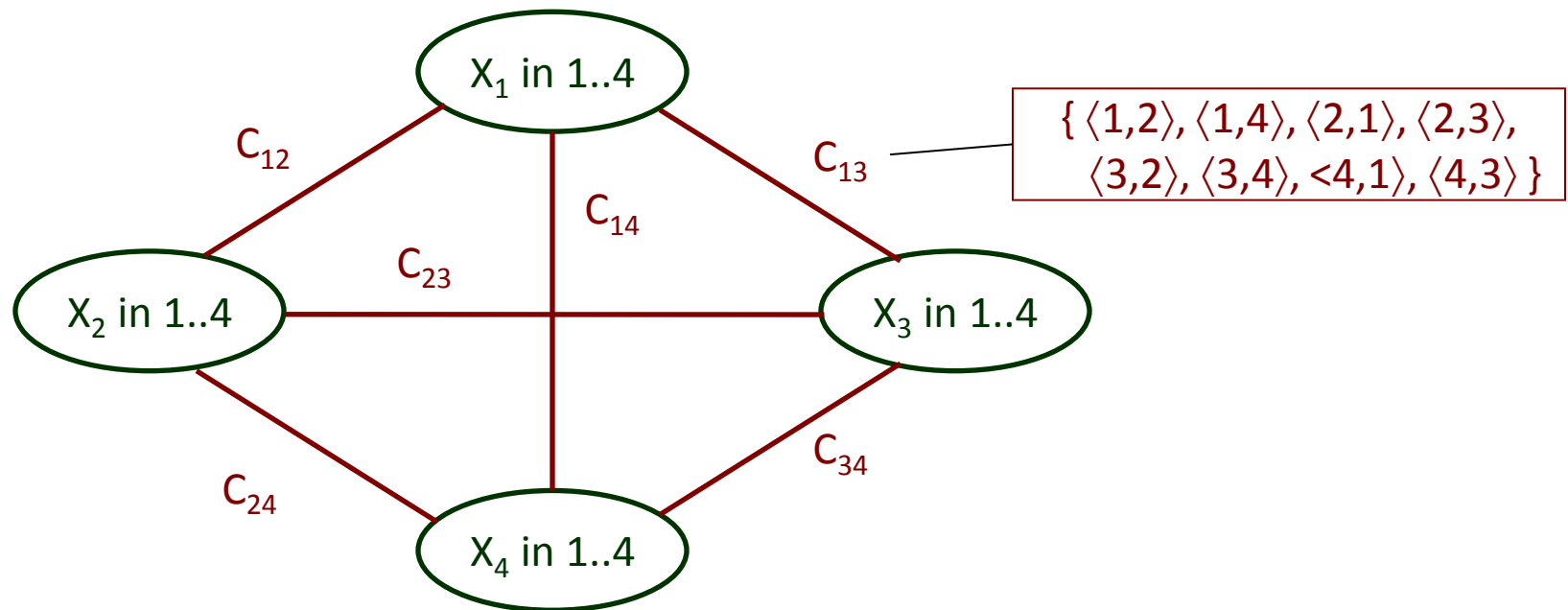
- **Problema de Satisfação de Restrições (PSR)**
 - É um triplo $\langle V, D, C \rangle$ onde:
 - V é um conjunto de variáveis do problema
 - D são os domínios dessas variáveis
 - C é um conjunto de restrições do problema
- **Solução de um PSR**
 - Uma solução de um PSR é definida como um tuplo $\langle V, D, C \rangle$, em que D é um *label* composto sobre as variáveis V do problema que satisfaz as restrições em C

Rede de Restrições

- **Rede (Grafo) de Restrições**
 - Contém em cada nó uma variável e para cada restrição entre duas variáveis contém um arco ligando os nós correspondentes a essas variáveis
- Quando os problemas contêm restrições de aridade superior, a rede de restrições pode ser formada após converter as restrições para restrições binárias equivalentes
- O problema pode também ser representado por um Híper-Grafo
- **Híper-Grafo de Restrições**
 - Um híper-grafo de restrições inclui nos nós as variáveis e para cada restrição, um híper-arco ligando os nós das variáveis que participam na restrição

Rede de Restrições

- Exemplo:
 - O problema das 4-Rainhas pode ser representado pela rede de restrições seguinte:



Rede de Restrições Completa

- **Rede de Restrições Completa**
 - Uma rede de restrições é completa quando, para quaisquer dois nós da rede, existe sempre um arco que os une (ou seja, existe uma restrição entre qualquer par de variáveis)
- O Problema das N-Rainhas tem uma rede de restrições completa para qualquer N
- No entanto não é esta a regra geral em PSRs e é importante medir a densidade da rede de restrições
- **Densidade da Rede de Restrições**
 - A densidade de uma rede de restrições é a razão entre o número de arcos da rede e o número de arcos de uma rede completa com o mesmo número de nós

Dificuldade de um PSR

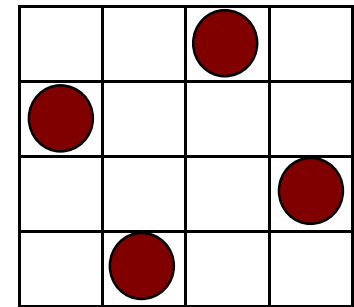
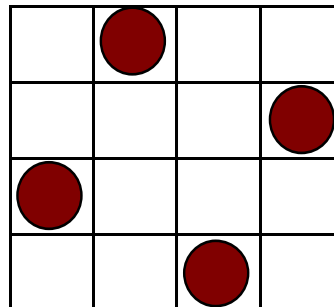
- A **dificuldade de resolução** de um PSR está normalmente relacionada com a densidade da sua rede de restrições:
 - Quanto maior a densidade mais difícil é o problema pois haverá mais possibilidades de invalidar possíveis soluções
- Convém distinguir:
 - Dificuldade do problema
 - Dificuldade de resolver o problema
- Por vezes um problema difícil pode ser de forma trivial provado impossível de resolver!
- A dificuldade de um problema está também relacionada com a dificuldade de satisfazer cada uma das suas restrições (que pode ser medida através da “*constraint tightness*”)

Tightness de uma Restrição

- Dada uma restrição C nas variáveis $X_1 \dots X_n$, com domínios D_1 a D_n , a *tightness* de C é definida como a razão entre o número de *labels* que define a restrição e a dimensão (cardinalidade) do produto cartesiano $D_1 \times D_2 \times \dots D_n$
- Exemplo:
 - *Tightness* da restrição C_{13} do problema das 4-Rainhas nas variáveis X_1 e X_3 com domínios $1..4$:
 - $C_{13} = \{\langle 1,2 \rangle, \langle 1,4 \rangle, \langle 2,1 \rangle, \langle 2,3 \rangle, \langle 3,2 \rangle, \langle 3,4 \rangle, \langle 4,1 \rangle, \langle 4,3 \rangle\}$
 - $Tightness = 8 / (4 \times 4) = 1/2$
- A noção de *tightness* pode ser generalizada para o problema completo

Tightness de um Problema

- A *tightness* de um PSR com variáveis $X_1 \dots X_n$, é a razão entre o número de soluções e a cardinalidade do produto cartesiano $D_1 \times D_2 \times \dots \times D_n$
- Exemplo:
 - No problema das 4-Rainhas só existem duas soluções: $\langle 2,4,1,3 \rangle$ e $\langle 3,1,4,2 \rangle$
 - Logo, $Tightness = 2 / (4*4*4*4) = 1/128$



Dificuldade de um Problema

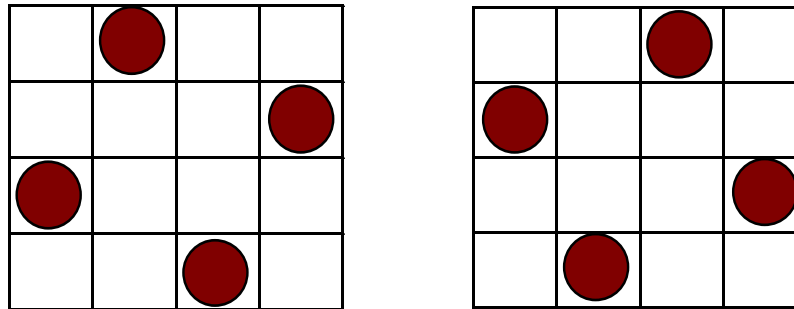
- A dificuldade de resolução de um PSR está então relacionada com a densidade da sua rede de restrições e com o seu *tightness*
- Como tal, quando se testam algoritmos para resolver este tipo de problemas é usual gerar instâncias aleatórias do problema, parametrizadas pelo número de nós e arcos, densidade da rede de restrições e *tightness* das restrições
- Os PSRs usualmente exibem uma transição de fase separando problemas de resolução trivial de problemas de prova trivial de inexistência de solução. Os problemas entre estes dois tipos são os problemas realmente complexos...

Redundância

- Na resolução de um PSR é conveniente considerar a existência potencial de valores e *labels* redundantes nas suas restrições
 - Valor Redundante
 - Um valor do domínio de uma variável é redundante se não aparece em nenhuma solução do problema
 - Label Redundante
 - Um *label* composto de uma restrição é redundante se não é a projeção nas variáveis da restrição de uma solução para o problema global

Redundância

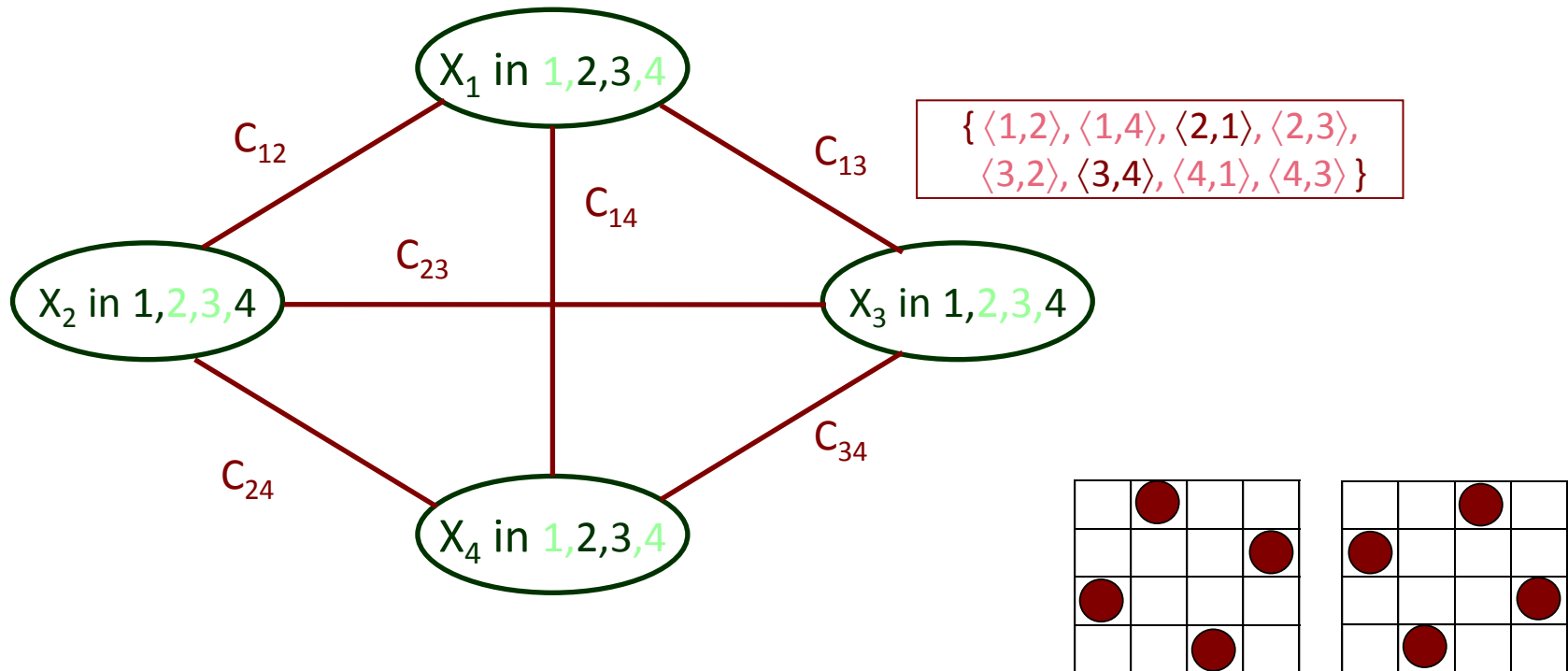
- Exemplo:
 - O problema das 4-Rainhas só admite duas soluções, $\langle 2,4,1,3 \rangle$ e $\langle 3,1,4,2 \rangle$



- Valores Redundantes:
 - Os valores 1 e 4 são redundantes nos domínios das variáveis de X_1 e X_4 e os valores 2 e 3 são redundantes no domínio das variáveis X_2 e X_3
- Labels Redundantes:
 - Os labels $\langle 2,3 \rangle$ e $\langle 3,2 \rangle$ são redundantes na restrição C13

Redundância

- Exemplo:
 - O problema das 4-Rainhas, que só admite duas soluções ($\langle 2,4,1,3 \rangle$ e $\langle 3,1,4,2 \rangle$), pode ser simplificado eliminando valores e *labels* redundantes:



Problemas Equivalentes e Reduzidos

- Qualquer problema é equivalente a qualquer das suas versões simplificadas
- **Problemas Equivalentes**
 - Dois problemas $P1 = \langle V1, D1, C1 \rangle$ e $P2 = \langle V2, D2, C2 \rangle$ são **equivalentes** se têm as mesmas variáveis (i.e. $V1 = V2$) e o mesmo conjunto de soluções
- “Simplificando” um problema: **Problema Reduzido**
 - Um problema $P = \langle V, D, C \rangle$ é **reduzido** para $P' = \langle V', D', C' \rangle$ se
 - P e P' são equivalentes
 - Os domínios D'_x estão incluídos em D_x
 - As restrições C' são pelo menos tão restritivas como as de C

Construção vs. Reparação

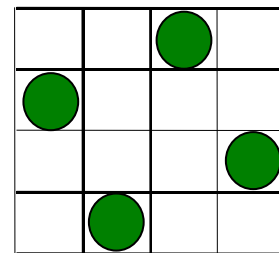
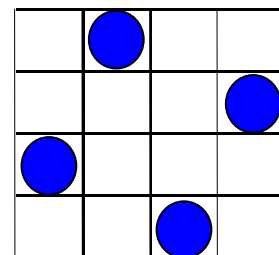
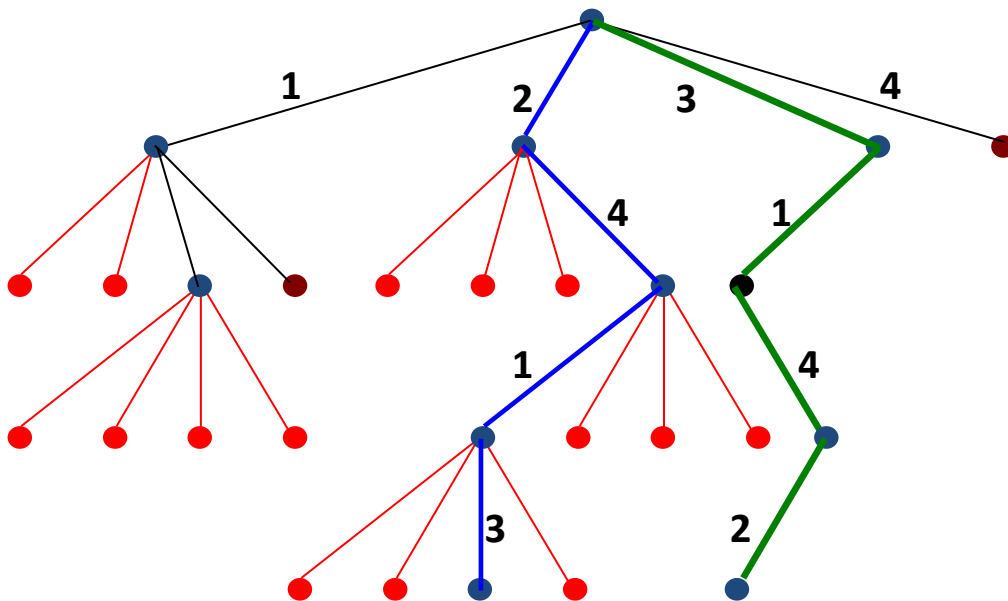
- Como foi demonstrado anteriormente, independentemente da redução do problema, em problemas de decisão (satisfação), um mecanismo do tipo “*generate and test*” para resolver um PSR é usualmente muito ineficiente
- No entanto esta é a abordagem dos métodos de pesquisa local tais como arrefecimento simulado (“*simulated annealing*”), pesquisa tabu ou algoritmos genéticos
- Na maioria dos casos é preferível utilizar um método de resolução construtivo e incremental, onde a solução vai sendo construída, uma variável de cada vez (incremental) até a solução ser atingida
- No entanto, é preciso garantir, em cada passo de construção da solução, que a solução parcial ainda tem potencial para chegar a ser uma solução completa!

Solução Parcial

- **Solução K-Parcial:**
 - Uma solução k-parcial de um PSR $P = \langle V, D, C \rangle$ pode ser definida como um label composto de um subconjunto de k das suas variáveis, V_k , que satisfaz todas as restrições em C cujas variáveis estão incluídas em V_k

Espaço de Pesquisa

- A pesquisa é basicamente uma pesquisa em árvore com backtracking em que as soluções k-parciais correspondem aos nós internos e as soluções completas às folhas da árvore



Espaço de Pesquisa

- Quanto **mais reduzido** estiver um problema, **mais fácil é de resolver**
- Dado um problema $P = \langle V, D, C \rangle$ com n variáveis X_1, \dots, X_n , o espaço de pesquisa (i.e. as folhas da árvore de pesquisa com labels compostos $\{\langle X_1-v_1 \rangle, \dots, \langle X_n-v_n \rangle\}$) tem cardinalidade:
$$\#S = \#D_1 * \#D_2 * \dots * \#D_n$$
- Em média, se as variáveis tiverem dimensão do domínio $\#D_i = d$, o espaço de pesquisa terá a dimensão:
$$\#S = d^n$$
- Ou seja, cresce exponencialmente com a dimensão do problema

Redução vs. Pesquisa

- Se, em vez da cardinalidade d do problema inicial, for resolvido um **problema reduzido**, cujos domínios têm uma cardinalidade inferior d' ($<d$), a **dimensão do espaço de pesquisa diminui exponencialmente!**
- $S'/S = d'^n / d^n = (d'/d)^n$
- Este decréscimo exponencial pode ser muito significativo para valores elevados de n :

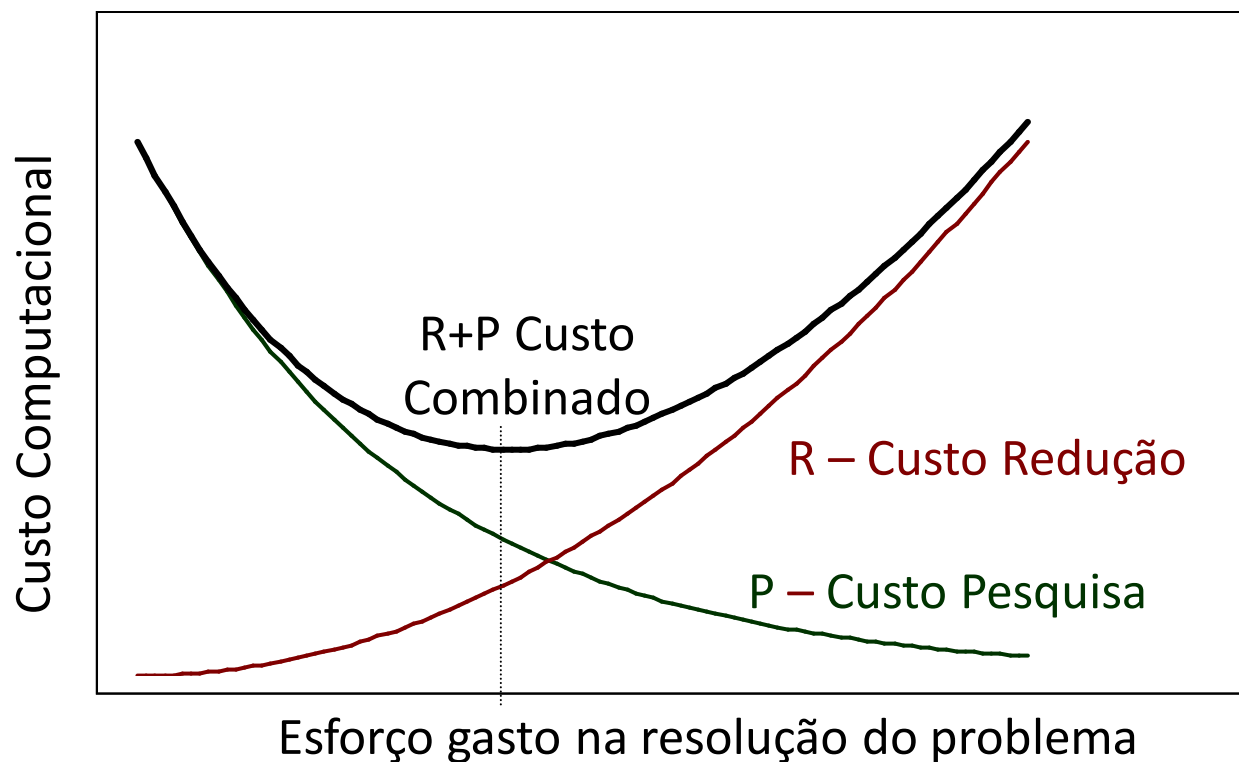
		n									
S/S'		10	20	30	40	50	60	70	80	90	100
7	6	4.6716	21.824	101.95	476.29	2225	10395	48560	226852	1E+06	5E+06
6	5	6.1917	38.338	237.38	1469.8	9100.4	56348	348889	2E+06	1E+07	8E+07
5	4	9.3132	86.736	807.79	7523.2	70065	652530	6E+06	6E+07	5E+08	5E+09
4	3	17.758	315.34	5599.7	99437	2E+06	3E+07	6E+08	1E+10	2E+11	3E+12
3	2	57.665	3325.3	191751	1E+07	6E+08	4E+10	2E+12	1E+14	7E+15	4E+17
d	d'										

Redução vs. Pesquisa

- Na prática, no entanto, esta diminuição potencial do espaço de pesquisa tem um custo envolvido para encontrar os valores e etiquetas redundantes
- A análise detalhada dos custos/benefícios é, em geral muito complexa, pois o processo depende imenso da instância do problema a resolver
- É normal que o esforço computacional despendido na redução do problema seja inicialmente compensador, mas se torne cada vez menos eficiente; a partir de um determinado ponto deixará de compensar em termos de redução do espaço de pesquisa

Redução vs. Pesquisa

- O processo de redução/pesquisa pode ser representado pelo seguinte gráfico:



Redução vs. Pesquisa

- Em CLP a especificação das restrições usualmente precede o processo de pesquisa (enumeração):

Solve_Problem(Vars):-

**Declaração das Variáveis e Domínios,
Colocação das Restrições,
Pesquisa da Solução (enumeração das variáveis).**

- No entanto, o modelo de execução vai alternando a pesquisa (enumeração) com a propagação, tornando possível a redução consecutiva do problema ao longo do processo de pesquisa da solução

Redução vs. Pesquisa

- Dado um problema com n variáveis X_1 a X_n , o modelo de execução segue o padrão seguinte:

Declaração das Variáveis e Domínios,

Colocação das Restrições,

indomain(X_1), % selecção de um valor para X_1 com retrocesso

propagação, % redução do problema (X_2 a X_n)

indomain(X_2),

propagação, % redução do problema (X_3 a X_n)

...

indomain(X_{n-1}),

propagação, % redução do problema (X_n)

indomain(X_n).

Redução vs. Pesquisa

- Definida formalmente a noção de redução do problema, convém compreender os métodos possíveis para a sua execução
- É necessário garantir que, qualquer que seja o método utilizado, a redução mantém o problema equivalente ao inicial:
 - Soluções têm de ser as mesmas!
 - Como garantir? Ainda não sabemos as soluções!
 - Garantir que na redução nenhuma solução é perdida!
 - Diversos critérios para o fazer

Programação em Lógica com Restrições

7. MANUTENÇÃO DE CONSISTÊNCIA

Adaptado de:

Pedro Barahona, *Página da Disciplina de Programação por Restrições, Ano Lectivo 2003/04*, disponível em:

<http://ssdi.di.fct.unl.pt/~pb/cadeiras/pr/0304/index.htm>

[consultado em Setembro de 2004]

Consistência

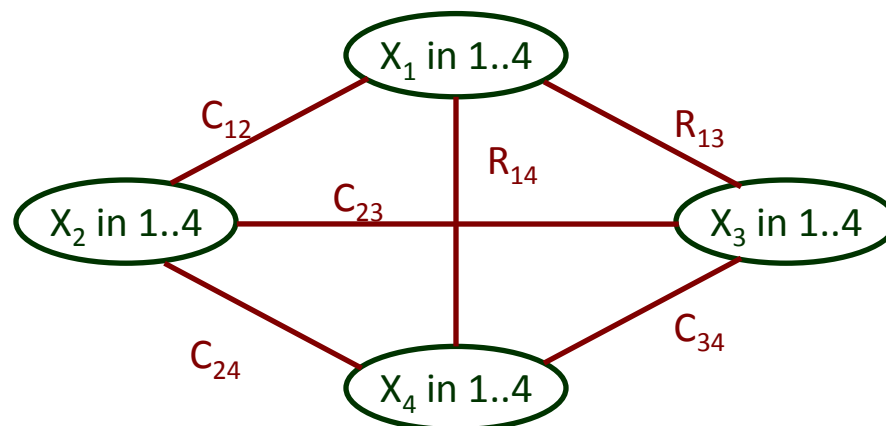
- Critérios de Consistência permitem estabelecer valores redundantes nos domínios das variáveis de forma indireta, i.e. não necessitando de conhecimento à-priori das soluções.
- Procedimentos que mantenham esses critérios durante a propagação vão eliminar valores redundantes e, como tal, reduzir o espaço de pesquisa das variáveis a serem enumeradas.
- Em PSRs com restrições binárias, os critérios mais usuais são:
 - Consistência dos Nós (*“Node Consistency”*) - o mais simples
 - Consistência dos Arcos (*“Arc Consistency”*)
 - Consistência dos Caminhos (*“Path Consistency”*) – o mais complexo

Consistência dos Nós

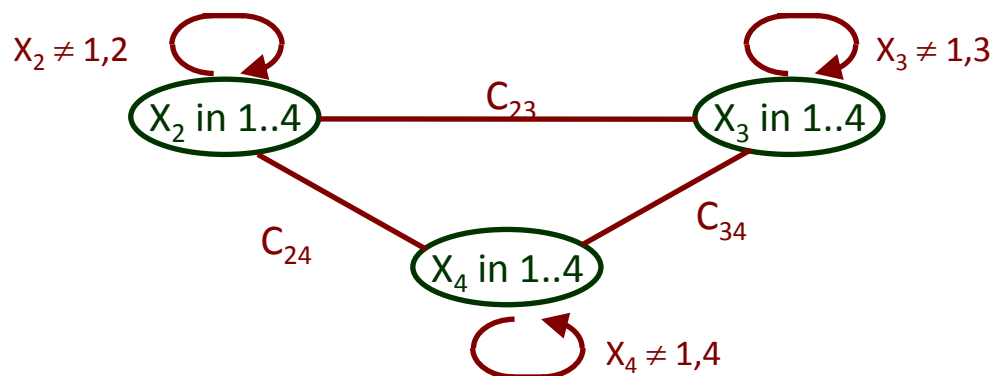
- Definição:
 - Um PSR é **consistente nos nós** (“*node-consistent*”) se nenhum valor no domínio das suas variáveis viola as restrições unárias
- Este critério parece óbvio e inútil. Quem ia especificar um domínio que violasse as restrições unárias?!
- No entanto o critério é útil (indispensável) durante o processo de execução que incrementalmente vai construindo a solução atribuindo valores às variáveis:
 - Restrições que não eram unárias no início do processo, tornam-se unárias a determinada altura quando outras variáveis já foram enumeradas

Consistência dos Nós

- Exemplo:
 - Depois de colocadas as restrições do problema das N-Rainhas temos a seguinte rede de restrições:



- Depois de enumerar a variável X_1 , (como $X_1=1$), as restrições C_{12} , C_{13} e C_{14} tornam-se unárias:



Consistência dos Nós

- Um algoritmo possível é o NC-1:

```
procedure NC-1(V, D, C);  
  for X in V  
    for v in Dx do  
      for Cx in {Cons(X) : Vars(Cx) = {X}} do  
        if not satisfy(X-v, Cx) then  
          Dx <- Dx \ {v}  
        end for  
      end for  
    end for  
  end procedure
```

- Complexidade: Espaço $O(nd)$, Tempo: $O(nd)$
- Baixa complexidade torna o NC-1 útil em virtualmente todas as situações.
- No entanto, a consistência dos nós é muito incompleta não sendo capaz de detetar muitas reduções possíveis

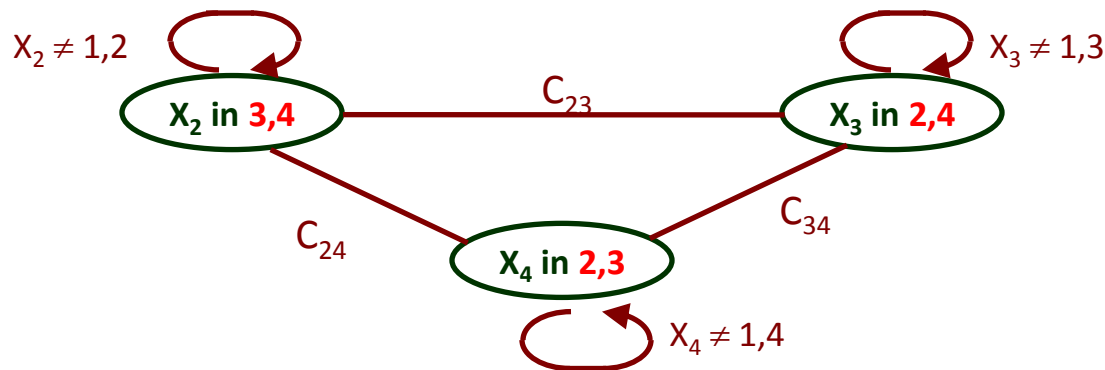
Consistência dos Arcos

- Critério mais complexo de consistência
- Definição:
 - Um PSR é **consistente nos arcos** (“*arc-consistent*”) se
 - é consistente nos nós; e
 - para cada label X_i-v_i de cada variável X_i e para todas as restrições C_{ij} , que envolvam as variáveis X_i e X_j , existir um valor v_j que suporte v_i , i.e. tal que o label composto $\{X_i-v_i, X_j-v_j\}$ satisfaça a restrição C_{ij}

Consistência dos Arcos

- Exemplo:

- Com $X_1=1$, fazendo a rede consistente nos nós, o problema das 4-rainhas tem a seguinte rede de restrições:






1	1			$X_2 \neq 1,2$
1		1		$X_3 \neq 1,3$
1			1	$X_4 \neq 1,4$

- No entanto o label X_2-3 não tem suporte na variável X_3 , uma vez que nenhum dos labels compostos: $\{X_2-3, X_3-2\}$ nem $\{X_2-3, X_3-4\}$, satisfazem a restrição C_{23}
- Portanto, o valor 3 pode ser removido com segurança do domínio de X_2

Consistência dos Arcos

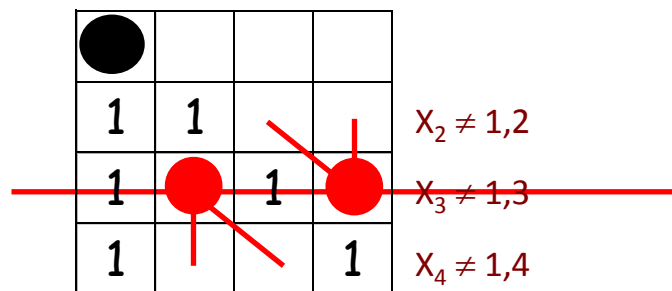
- No exemplo, nenhum dos valores de X3 tem suporte nas variáveis X2 e X4:

				
1	1			$X_2 \neq 1,2$
1		1		$X_3 \neq 1,3$
1			1	$X_4 \neq 1,4$

- O label X3-4 não tem suporte na variável X2, pois nenhum dos labels compostos $\{X2-3, X3-4\}$ e $\{X2-4, X3-4\}$ satisfazem a restrição C23
- O label X3-2 não tem suporte na variável X4, pois nenhum dos labels compostos $\{X3-2, X4-2\}$ e $\{X3-2, X4-3\}$ satisfazem a restrição C34

Consistência dos Arcos

- Como nenhum dos valores do domínio de X_3 tem suporte para as variáveis X_2 e X_4 , a manutenção da consistência nos arcos esvazia o domínio de X_3 !



- A manutenção da consistência dos arcos não só reduz os domínios das variáveis como também antecipa a detecção de que a variável X_3 não terá um valor que satisfaça as restrições
- Desta forma, o retrocesso de $X_1=1$ pode ser começado mesmo antes da enumeração da variável X_2

Consistência dos Arcos

- Algoritmo muito simples AC-1, para manutenção da Consistência dos Arcos:

```
procedure AC-1(V, D, C);  
  NC-1(V,D,C);           % consistência dos nós  
  Q = {aij | Cij ∈ C ∨ Cji ∈ C }; % ver nota  
  repeat  
    changed <- false;  
    for aij in Q do  
      changed <- changed or revise_dom(aij,V,D,C)  
    end for  
  until not changed  
end procedure
```

- Nota: para Cij são considerados dois arcos dirigidos aij e aji

Consistência dos Arcos

- O predicado `revise_dom(aij, V, D, C)` sucede sse reduzir o domínio de X_i :

```
predicate revise_dom(aij, V, D, C): Boolean;  
    success <- false;  
    for vi in dom(Xi) do  
        if there is no vj in dom(Xj) such that  
            satisfies({Xi-vi, Xj-vj}, Cij) then  
                dom(Xi) <- dom(Xi) \ {vi};  
                success <- true;  
            end if  
        end for  
        revise_dom <- success;  
    end predicate
```

Consistência dos Arcos

- Complexidade:
 - Tempo: $O(d^2 * 2a * nd) = O(nad^3)$
 - Espaço: $O(nd + ad^2) = O(ad^2)$
em que $a = n^o$ arcos; $n = n^o$ variáveis; $d = n^o$ valores
- Algoritmo AC-1 é muito pouco eficiente:
 - Sempre que um valor v_i é removido do domínio de uma variável X_i por **revise_dom(a_{ij}, V, D, R)**, **todos** os arcos são reexaminados
 - No entanto só os arcos a_{ki} (para $k \neq i$ e $k \neq j$) devem ser reexaminados
 - Isto deve-se a que a remoção de v_i pode eliminar o suporte de um dado valor v_k de uma variável X_k para a qual existia uma restrição C_{ki} (ou C_{ik})
 - Esta falta de eficiência é eliminada pelo algoritmo AC-3

Consistência dos Arcos

- Algoritmo AC-3, para manutenção da Consistência dos Arcos:

```
procedure AC-3(V, D, C);  
  NC-1(V,D,C);      % consistência dos nós  
  Q = {aij | Cij ∈ C ∨ Cji ∈ C };  
  while Q ≠ ∅ do  
    Q = Q \ {aij}      % remove um elemento de Q  
    if revise_dom(aij,V,D,C) then      % Xi revisto  
      Q = Q ∪ {aki | Cki ∈ C ∧ k ≠ i ∧ k ≠ j}  
    end if  
  end while  
end procedure
```

- AC-3 tem não só uma menor complexidade no pior caso mas também uma menor complexidade no caso típico em relação a AC-1.
- Complexidade no Tempo: $O(2ad^3) = O(ad^3)$
- Melhores algoritmos: AC-4, AC-6, AC-7, ...

Consistência dos Caminhos

- Ideia base da consistência dos caminhos (“*path consistency*”): para além de verificar o suporte entre as variáveis X_i e X_j nos arcos da rede de restrições, verificar também o suporte nas variáveis $X_{k_1}, X_{k_2}, \dots, X_{k_m}$ que formam um caminho entre X_i e X_j , se existem restrições $C_{ik_1}, C_{k_1k_2}, \dots, C_{k_mj}$
- Isto é equivalente a procurar o suporte em qualquer variável X_k conectada simultaneamente a X_i e X_j
- Manter este tipo de consistência tem um custo muito elevado

Consistência dos Caminhos

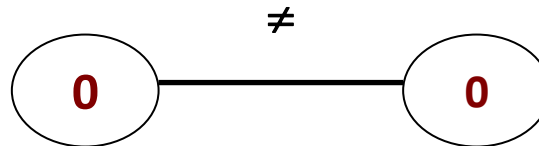
- Definição:
 - Um PSR é **consistente nos caminhos** (“*path-consistent*”) se:
 - é consistente nos arcos; e
 - para cada restrição C_{ij} nas variáveis X_i e X_j , se existirem restrições C_{ik} e C_{jk} entre estas variáveis e uma terceira X_k , então para todos os *labels* compostos $\{X_i-v_i, X_j-v_j\}$ tem de existir um valor v_k tal que os *labels* compostos $\{X_i-v_i, X_k-v_k\}$ e $\{X_j-v_j, X_k-v_k\}$ satisfazem as restrições C_{ik} e C_{jk}

K-Consistência

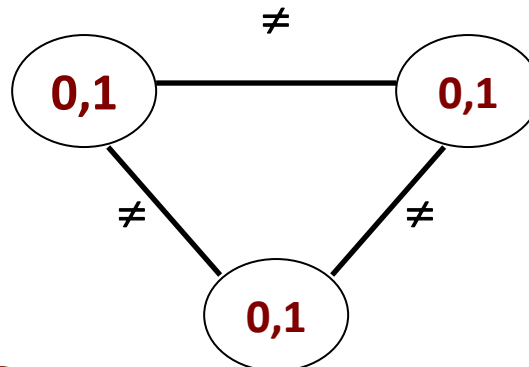
- Consistência dos Nós, Arcos e Caminhos são instâncias do caso geral da **k-consistência**
- Informalmente, uma rede de restrições é **k-consistente** quando para um grupo de k variáveis $X_{i1}, X_{i2}, \dots, X_{ik}$, os valores em cada domínio têm suporte nos valores dos domínios das outras variáveis, considerando este suporte de forma global

K-Consistência

- Rede consistente nos nós mas que não é consistente nos arcos:

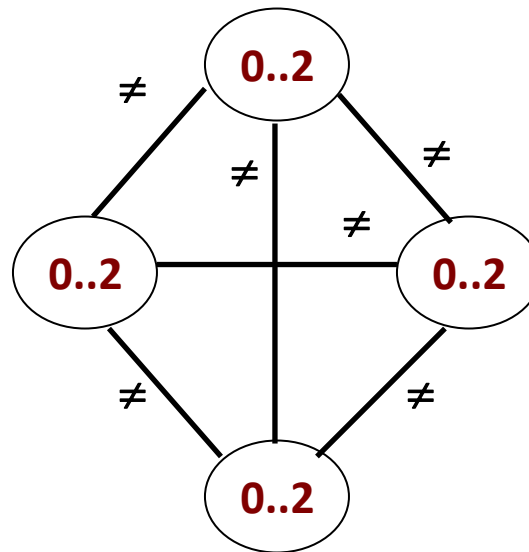


- Rede consistente nos arcos mas não nos caminhos:



K-Consistência

- Rede consistente nos caminhos mas que não é 4-consistente:



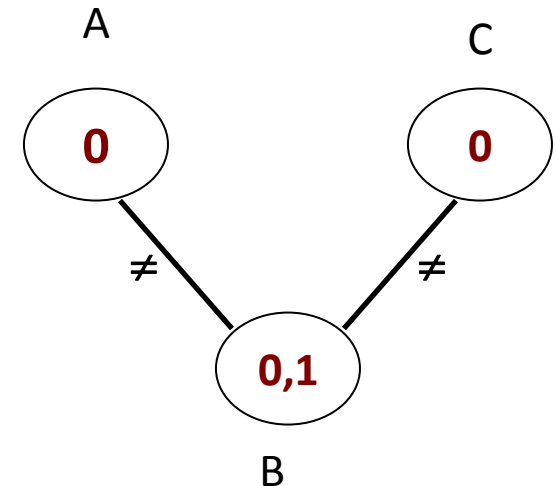
K-Consistência

- Definição de **k-Consistência**:
 - Um PSR é **1-consistente** se os valores dos domínios das variáveis satisfazem as suas restrições unárias
 - Um PSR é **k-consistente** sse todos os seus $(k-1)$ -*labels* compostos (formados por $k-1$ pares $X-v$) que satisfazem as restrições relevantes podem ser estendidos com um *label* de outra variável para formar um *label* k -composto que ainda satisfaz as restrições relevantes
- Definição de **k-Consistência forte**:
 - Um PSR é **k-Consistente forte** sse é i -consistente, para qualquer $i \in 1..k$

K-Consistência

- Exemplo: Rede 3-consistente, mas não 2-consistente. Logo, não é 3-consistente forte

- Os 3 *labels* compostos:
 $\{A-0, B-1\}$, $\{A-0, C-0\}$ e $\{B-1, C-0\}$
satisfazem as restrições e podem ser
estendidos com a variável restante:
 $\{A-0, B-1, C-0\}$.



- No entanto, o *label* 1-composto $\{B-0\}$ não pode ser
estendido às variáveis A ou C: $\{A-0, B-0\}$ ou $\{B-0, C-0\}$!

K-Consistência

- As consistências dos nós, arcos e caminhos são todas instâncias da, mais geral, **k-consistência** (ou melhor, **k-consistência forte**):
 - Um PSR é consistente nos nós se e só se é 1-consistente
 - Um PSR é consistente nos arcos sse é 2-consistente forte
 - Um PSR é consistente nos caminhos sse é 3-consistente forte

Programação em Lógica com Restrições

8. PESQUISA, OTIMIZAÇÃO E EFICIÊNCIA

Bibliografia base:

Edward Tsang, *Foundations of Constraint Satisfaction*, Academic Press, London and San Diego, 1993

Kim Marriott e Peter J. Stuckey. *Programming with Constraints: an Introduction*, MIT Press, 1998

Pesquisa em PLR

- A ideia principal da PLR consiste em resolver problemas complexos, **inferindo incrementalmente propriedades das soluções** e utilizando esta informação para forçar a **consistência**
 - Este conhecimento determinístico é adquirido numa forma explícita, sendo possível **reduzir o espaço de pesquisa**, excluindo certos casos que não podem conduzir à solução
- Normalmente a solução não pode ser obtida diretamente dos passos determinísticos sendo **necessário efetuar suposições** sobre as soluções do problema
 - Estas suposições são introduzidas no esquema de raciocínio com restrições, permitindo a obtenção de informação adicional sobre a solução
 - Este processo é repetido continuamente até que a solução seja obtida
 - Se a rede de restrições se tornar **inconsistente**, as suposições têm de ser retiradas, sendo colocadas suposições alternativas (“**backtracking**”)

Pesquisa em PLR

- O esquema de pesquisa com restrições depende crucialmente de dois aspectos:
 - Capacidade de **inferência** do mecanismo de raciocínio com restrições (**capacidade de propagação**)
 - **Estratégia** utilizada para fazer as suposições sobre a solução (variáveis e valores)
- Em IA, a resolução de problemas é classicamente vista como uma pesquisa num espaço de estados:
 - Resolver um problema consiste em encontrar um caminho desde o estado inicial até ao estado final desejado (representando a solução)
- Em PLR: a **pesquisa** é efetuada se o manipulador de restrições não puder fornecer mais informação
 - Lidamos com **soluções parciais**, ou seja, em cada estado da pesquisa, conhecemos os valores de diversas variáveis mas não de todas

Eficiência em PLR

- Efectuar um passo de pesquisa consiste em tomar duas decisões:
 - Em que aspecto do problema fazer a suposição?
 - (Que **variável** escolher?)
 - Qual deve ser a suposição?
 - (Que **valor** escolher para essa variável?)
- A eficiência da PLR em domínios finitos depende pois de:
 - **Algoritmos de propagação** adequados
 - **Heurísticas** apropriadas de selecção de:
 - Variável a utilizar no processo de enumeração
 - Exemplos: *First Fail*, *First Fail Constrained*
 - Valor a atribuir a essa variável

Algoritmos de Pesquisa:

“Backtracking” vs. “Forward Checking”

- *Backtracking*

- Variável é instanciada com um valor do seu domínio, sendo a atribuição verificada tendo em conta a solução parcial atual
- Se alguma das restrições for violada, a atribuição é abandonada e é escolhido outro valor
- Se todos os valores foram já experimentados para a variável, o algoritmo retrocede para a variável anterior e atribui-lhe um novo valor
- Se um PSR tiver n variáveis, cada qual com m possíveis valores, existem m^n possíveis atribuições
- Algoritmo de *Backtracking* só verifica as restrições entre a variável corrente e as variáveis anteriores

- *Forward Checking*

- Verifica as restrições entre a variável corrente (e anteriores) e as variáveis futuras
- Quando um valor é atribuído à variável corrente, qualquer valor de uma variável futura que entre em conflito com esta atribuição é (temporariamente) removido do seu domínio dessa variável

Heurísticas de Ordenação de Variáveis

- Seleção da próxima **variável a instanciar**
 - Dois métodos possíveis:
 - Ordenação Estática
 - Ordenação Dinâmica
 - Com **Backtracking** simples não existe informação adicional que permita ordenação dinâmica
 - Com **Forward Checking**, o estado corrente inclui os domínios das variáveis, reduzidos tendo em conta o conjunto corrente de instanciações
 - Heurística comum: “**First Fail Principle**”
 - “*To succeed, try first where you are most likely to fail*”
 - Escolher a variável com menos valores ainda possíveis no domínio
 - Outra hipótese: escolher as variáveis com mais restrições ou com restrições mais apertadas
 - Tratar os casos mais difíceis primeiro, pois eles só podem tornar-se piores se forem colocados de parte

Heurísticas de Ordenação de Valores

- Seleção de um **valor para instanciar a variável** atual
 - Dois métodos possíveis:
 - **Ordenação Estática**
 - **Ordenação Dinâmica**
 - Heurística comum:
 - Escolher o valor que tem maior probabilidade de sucesso!
 - Problema: Como determinar esse valor?
 - Calculando a percentagem de valores nos domínios futuros que não poderão ser utilizados (como aproximação ao custo de fazer esta seleção)
 - Calculando a promessa de cada valor como o produto dos tamanhos dos domínios das variáveis futuras depois de efetuar esta seleção
 - Cortar o domínio em duas partes, (por exemplo ao meio)
 - Questão: Será que compensa efetuar o “*forward checking*” antecipado?
 - Normalmente utiliza-se conhecimento específico do problema

Simetrias

- Em muitos problemas, se existirem soluções, existem classes de **soluções equivalentes**
- Exemplo:
 - No problema das “N-Rainhas”, invertendo o tabuleiro na horizontal ou vertical, obtemos novas soluções equivalentes
- **Simetrias** provocam dificuldades ao algoritmo de pesquisa
- Evitar simetrias
 - Introduzir restrições adicionais que permitam encontrar uma única solução para cada classe de soluções
 - Utilizar “*nogoods*”: um conjunto de atribuições a um subconjunto de variáveis que não conduz a uma solução

Eficiência em PLR

- Alterações de eficiência nos programas de PLR são devidas a diferentes:
 - Modelizações do Problema
 - Restrições Utilizadas
 - Algoritmos de Propagação
 - Heurísticas de Seleção de Variáveis
 - Heurísticas de Seleção de valores
- Alguns conceitos utilizados:
 - Restrições e Redes de Restrições
 - Soluções Parciais e Totais
 - Satisfação e Consistência

Programação em Lógica com Restrições

9. SISTEMAS DE PLR

Bibliografia base:

Krzysztof Apt, *Principles of Constraint Programming*, Cambridge University Press; 1ª edição, 2003

SICS-AB, *SICStus Prolog Home Page*, <https://sicstus.sics.se/>

[consultado em Dezembro de 2008]

Os Primeiros Sistemas de CLP

- CLP(R)
- CHIP
- CLP(FD) de Daniel Diaz
- Prolog III
- OZ
- Ilog Solver
- BProlog
- SICStus
- YAP Prolog

Sistemas de CLP – CLP(R)

- Monash University, IBM YorkTown Heights, CMU, 1992
- Desenvolvido como demonstração do esquema CLP(X) definido por Jaffar e Lassez
- “Constraint Solvers”:
 - Racionais Reais: Algoritmo Simplex Estendido
- Potencialidades:
 - Linguagem de programação em lógica completa
 - Restrições não lineares são retardadas
 - Algumas facilidades de Meta-programação com restrições

Sistemas de CLP – CHIP

- CHIP - Constraint Handling in Prolog
- European Computer Industry Research Centre (ECRC)
- “Constraint Solvers”
 - Finite Domains (FD): Técnicas de Consistência
 - Booleanos: Algoritmo de Unificação Booleana
 - Racionais (reais): Algoritmo Simplex Estendido
- Potencialidades:
 - Restrições simbólicas poderosas (element, atmost, etc) e restrições cumulativas
 - Utilizador pode definir as suas próprias restrições
 - Interface para X11, Dos graphics, BDs Oracle e Ingres, linguagem C

Sistemas de CLP – Prolog III

- Substituto do Prolog III que foi a primeira linguagem CLP comercial
- University of Marseille, Prologia na França
- “Constraint Solvers”:
 - Finite Domains (FD), Booleanos, Racionais (reais) e Listas
- Potencialidades:
 - Tratamento de restrições disjuntivas e potencialidades para problemas inteiros/reais
 - Compilador integrado num ambiente gráfico completo
 - Editor de projetos e editor de código, *debugger* e auxílio “on-line”

Sistemas de CLP – CLP(FD)

- Daniel Diaz - INRIA Rocquencourt, 1994
- CLP Domínios Finitos, baseado no compilador Prolog wamcc que traduz Prolog para C
- “Constraint Solvers”
 - Finite Domains (FD): Booleanos tratados como FD
- Potencialidades:
 - Compilador Prolog wamcc (estilo Edimburgh)
 - Extremamente rápido (tradução Prolog \rightarrow C) embora muito simples
 - Restrições simbólicas: `alldifferent(+L)`, `element(?I,+L,?V)`, `atmost(+N,+L,+V)`, `relation(+Tup,+Vars)`
 - Algumas facilidades para construção de novas restrições

Sistemas de CLP – ECLiPse

- ECLiPse - ECRC Logic Programming System (IC Parc - London)
- Sucessor do CHIP, combina funcionalidades do Sepia, Megalog e CHIP
- Bibliotecas com esquemas de manipulação de restrições:
 - CHR - Constraint Handling Rules (18 “*solvers*”), propagação generalizada (“*generalised propagation*”), racionais reais, listas, conjuntos, árvores, termos, domínios finitos e infinitos, “*path consistency*” incremental
- Potencialidades:
 - Acesso a Bases de Dados (sistema BANG)
 - Interface para Tcl/Tk X11 toolkit
 - Poderosas técnicas de propagação generalizada
 - Definição de novas restrições e novos “*solvers*”

Sistemas de CLP – OZ

- DFKI - German Research Center for Artificial Intelligence
- Baseada num novo modelo computacional que fornece uma base uniforme para programação funcional, programação orientada por objetos, programação em lógica, programação em lógica com restrições e programação concorrente
- Pesquisa encontra-se encapsulada (não existe *backtracking* visível)
- “Constraint Solvers”:
 - Finite Domains (FD), Booleanos , racionais (reais), imensas restrições simbólicas poderosas sobre diversas estruturas de dados
- Potencialidades:
 - Adequada à implementação de sistemas Multi-Agente e de sistemas de resolução de CSPs
 - Interface baseada em Emacs, Browser, Explorer e Painel de controle concorrentes
 - Tcl/Tk, sockets, ligação com C e C++
 - Definição fácil de restrições e métodos de pesquisa

Sistemas de CLP – YAP Prolog

- Excelente sistema de Prolog
- Extremamente eficiente – muito rápido
- Stream I/O, sockets, modules, exceptions, Prolog debugger, interface C, código dinâmico, DCGs, saved states e arrays
- Restrições: CLP(Q,R)
- CHR – Constraint Handling Rules
- Desenvolvido pela Universidade do Porto: Vítor Santos Costa, Rogério Reis *et al.*

Sistemas de CLP – SICStus

- Extremamente eficiente
- Muitas restrições pré-definidas
- $\text{CLP}(\text{R}, \text{Q})$
- $\text{CLP}(\text{B})$
- $\text{CLP}(\text{FD})$
- CHR
- Debugger CLP
- (Ver bloco de slides separado sobre “PLR no SICStus Prolog”)

Programação em Lógica com Restrições

10. CONCLUSÕES E LEITURA ADICIONAL

Conclusões

- Paradigma PLR combina:
 - **Declaratividade** da PL
 - **Eficiência** da resolução de restrições
- Resolução de problemas de pesquisa combinatória:
 - “Scheduling”, “timetabling”, alocação de recursos, planeamento, empacotamento, verificação de circuitos, etc.
- Vantagens principais:
 - Reduzido tempo de desenvolvimento
 - Facilidade de manutenção
 - Eficiência na Resolução
 - Clareza e brevidade dos programas

Leitura Adicional

- Marriot, Stuckey: “Programming with Constraints: An Introduction”, MIT press, 1998
- Edward Tsang: “Foundations of Constraint Satisfaction”, Academic Press, 1993
- Roman Barták: “On-line Guide to Constraint Programming”,
<http://kti.mff.cuni.cz/~bartak/constraints/>
- Documentação on-line do SICStus Prolog:
<https://sicstus.sics.se/documentation.html>