

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

# Programação em Lógica com Restrições

**Luís Paulo Reis e Daniel Castro Silva**

**Março de 2022**

Parcialmente baseado em slides anteriores de Henrique L. Cardoso ([hlc@fe.up.pt](mailto:hlc@fe.up.pt)), Luís Paulo Reis ([lpreis@fe.up.pt](mailto:lpreis@fe.up.pt)), Daniel Castro Silva ([dcs@fe.up.pt](mailto:dcs@fe.up.pt)), Pedro Barahona, John Hooker, Willem-Jan van Hoeve e outros autores

# Conteúdo

1. Introdução à Programação com Restrições
2. Exemplos de Problemas de Satisfação/Decisão e Otimização
3. Exemplos de Resolução de Problemas em PLR
4. Complexidade e Métodos de Pesquisa
5. Definições Formais e Conceitos
6. Manutenção de Consistência
7. Pesquisa, Otimização e Eficiência
8. Sistemas de PR e PLR
9. Conclusões e Leituras Adicionais

Programação em Lógica com Restrições

# 1. INTRODUÇÃO À PROGRAMAÇÃO 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 com Restrições

*“Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.”*

Eugene Freuder, 1997 (‘In Pursuit of the Holy Grail’,  
Constraints: An International Journal, 2, 57-61)

# Contexto Histórico

- Anos '60 e '70: Alguns desenvolvimentos prévios em tópicos e aplicações relacionadas
- Anos '70: Prolog (*'PROgrammation en LOGique'*)
  - Roussel, Colmerauer, Kowalsky et al.
- Anos '80: Programação (em Lógica) com Restrições
  - Prolog III, CHIP, ...
  - *Constraint & Generate vs Generate & Test*
- Anos '90: Programação por restrições, primeiros *solvers* industriais (ILOG, ...) e aplicações
- Anos 2000: Restrições globais, linguagens de modelação, ...

# Aplicações Industriais / Comerciais

- Aplicações usando programação com restrições:
  - Lufthansa
    - Escalonamento de pessoal a curto prazo
  - Renault
    - Planeamento de produção a curto prazo
  - Nokia
    - Configuração de software para telemóveis
  - Airbus
    - Layout de cabine
  - Siemens
    - Verificação de circuitos
  - Ver Helmut Simonis (1999), [Building Industrial Applications with Constraint Programming](#), in Constraints in Computational Logics (CCL 1999)

# Aplicações Industriais / Comerciais

- Escalonamento desportivo
  - Vários tipos de ligas / torneios
- 1997/1998 *ACC basketball league* (9 equipas)
  - Várias restrições laterais complicadas
  - As 179 soluções existentes foram encontradas em 24h usando enumeração e programação linear inteira [Nemhauser & Trick, 1998]
  - As 179 soluções foram encontradas em menos de um minuto usando programação com restrições [Henz, 1999, 2001]



# Aplicações Industriais / Comerciais

- Escalonamento de operações
- Aeroporto de Hong Kong
  - Alocação de *gates* no Aeroporto Internacional de Hong Kong
  - Sistema foi implementado em apenas quatro meses, e inclui tecnologia de programação com restrições (ILOG)
  - Faz o escalonamento de ~1100 voos por dia (mais de 70 milhões de passageiros em 2016)





# Aplicações Industriais / Comerciais

- Escalonamento de operações
- Porto de Singapura
  - Um dos maiores centros de transbordo do mundo
  - Faz a ligação a uma rede de 200 empresas de expedição com ligações a 600 portos em 123 países
  - Necessidade de atribuir localizações e planos de carga / descarga sob restrições operacionais e de segurança
  - Sistema de planeamento, baseado em programação com restrições (ILOG)



# Aplicações Industriais / Comerciais

- Escalonamento de operações
- Netherlands Railways
  - Uma das redes ferroviárias mais densas do mundo, com mais de 5.500 comboios por dia
  - Programação com restrições é uma das partes do software de planeamento, usado para obter um novo horário a partir do zero (2008)
  - Escalonamento mais robusto e eficiente, leva a lucro anual adicional de 75M
  - Vencedor do prémio INFORMS Edelman (2008)



# 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 problemas NP-completos):
  - Escalonamento (*scheduling*), geração de horários (*timetabling*), alocação de recursos, planeamento, gestão da produção, verificação de circuitos, ...

# Programação em Lógica com Restrições

- 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 um subconjunto das variáveis
- Vantagens:
  - Clareza e brevidade dos programas
  - Representação muito próxima da original
  - Reduzido tempo de desenvolvimento
  - Facilidade de manutenção
  - Maior garantia de correção dos programas
  - Algoritmos para resolver CSPs são muito eficientes

# Problema de Satisfação de Restrições

- Um Problema de Satisfação de Restrições - **PSR**, ou **CSP** (*Constraint Satisfaction Problem*) - é modelizado por:
  - **Variáveis** representando diferentes aspetos do problema, juntamente com os seus **domínios**
  - **Restrições** que limitam os valores que as variáveis podem tomar dentro dos seus domínios
- 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
  - A solução é encontrada por uma **pesquisa** sistemática, usualmente guiada por uma heurística, através de todas as atribuições possíveis de valores às variáveis



# Definição Formal de um CSP

- Mais formalmente, 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
    - $D: V \rightarrow \text{conjunto finito de valores}$
    - $D_{x_i}$ : domínio de  $x_i$
  - $C = \{C_1, C_2, \dots, C_n\}$  é o conjunto de restrições que afetam um subconjunto arbitrário de variáveis de  $V$ 
    - Para cada restrição  $c_i \in C$ 
      - $Vars(c_i)$  é o conjunto de variáveis envolvidas em  $c_i$
      - Simetricamente,  $Cons(x_i)$  é o conjunto de restrições nas quais a variável  $x_i \in V$  está envolvida

# Definição Formal de um CSP

- Solução de um CSP: atribuição de um valor a cada variável de forma a que todas as restrições sejam respeitadas:
  - $Sol = \{\langle x_1, v_1 \rangle, \langle x_2, v_2 \rangle, \dots, \langle x_n, v_n \rangle\}$ :
    - $\forall x_i \in V (\langle x_i, v_i \rangle \in Sol \wedge v_i \in D_{x_i})$
    - $\forall c_k \in C \text{ satisfaz } (Sol, c_k)$

# Problema de Otimização com Restrições

- Um Problema de Otimização com Restrições – **POR**, ou **COP** (*Constraint Optimization Problem*) – é um CSP com a adição de uma função objetivo a ser otimizada (maximizada / minimizada)
- A solução de um COP é uma atribuição de um valor (do seu domínio) a cada variável, de forma a que todas as restrições sejam satisfeitas e não haja nenhuma outra atribuição que resulte num valor superior ( / inferior) da função de avaliação



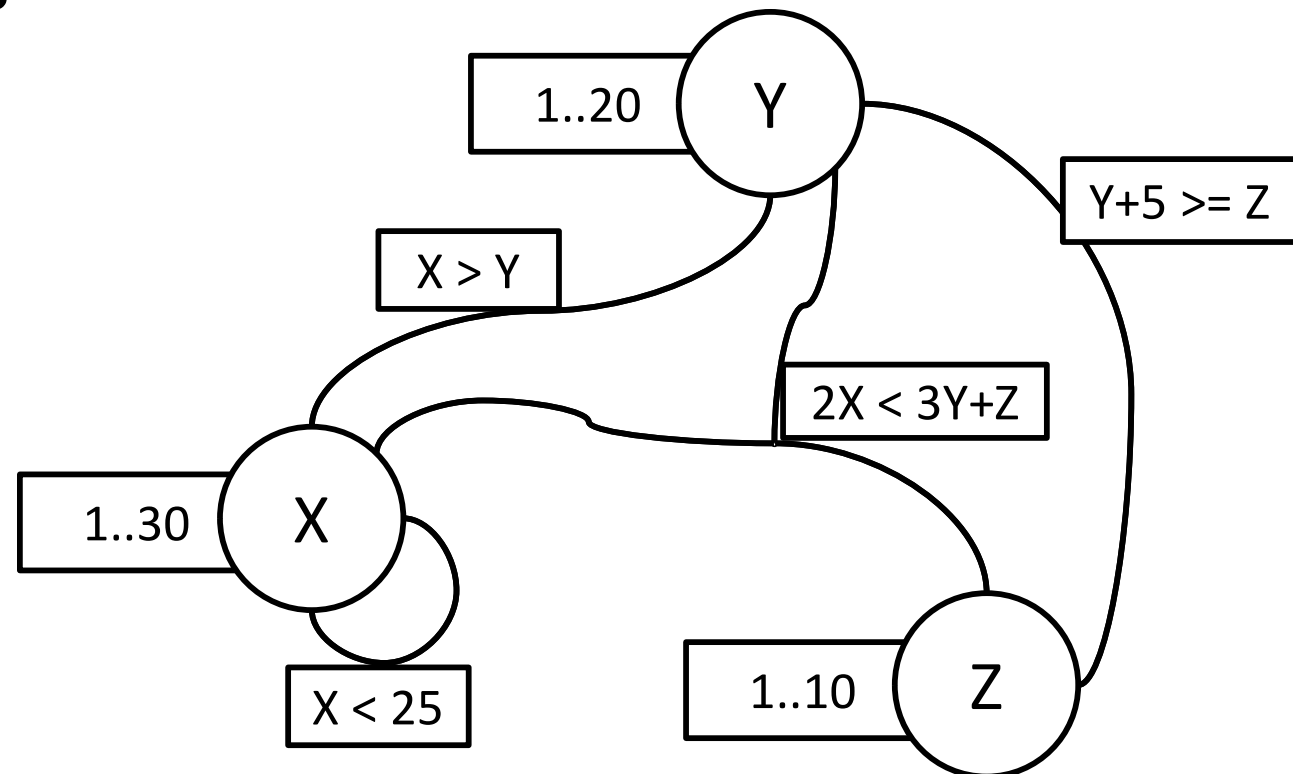
# Restrições em CSPs

## 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
    - $W*Z + 3*X > Y - Z*Z$

# Restrições em CSPs

- Um CSP pode ser visto como um híper-grafo, em que os nós representam variáveis (com domínios associados), e as restrições são (híper-)arcos que ligam os nós

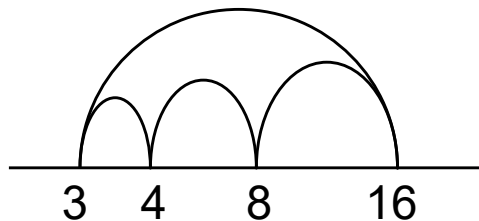


# Restrições

- As restrições de um CSP/COP podem ser:
  - **Rígidas** (*Hard Constraints*): são aquelas que têm obrigatoriamente de ser cumpridas
    - Todas as restrições num CSP são restrições rígidas
  - **Flexíveis** (*Soft Constraints*): são aquelas que podem ser quebradas
    - Deve existir um custo associado a quebrar este tipo de restrições, o qual deve ser somado na função de avaliação do COP
    - Restrições flexíveis permitem relaxamento
      - Aumenta complexidade do modelo

# Resolução de um CSP

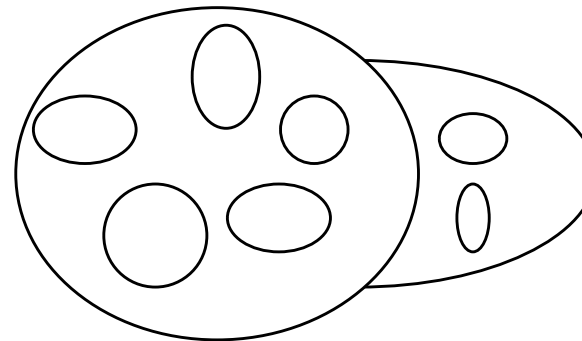
- Passos envolvidos:
  - Declarar as ***variáveis*** e os seus ***domínios*** (finitos)
  - Especificar as ***restrições*** existentes
  - ***Pesquisar*** para encontrar a solução



Variáveis Lógicas:

$x = 3$

$y = 8$



Domínios das variáveis:

$x :: 1..16$

$y :: 1..10$

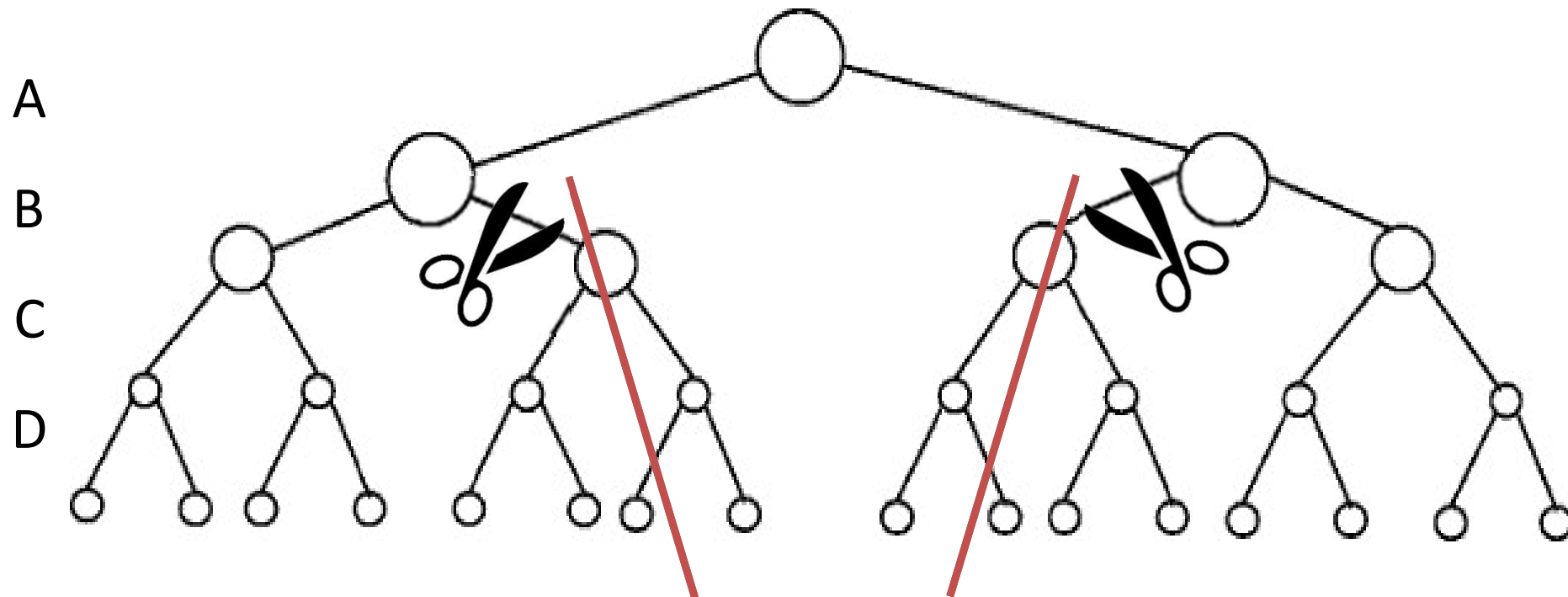
# Resolução de um CSP

- Técnicas utilizadas na resolução:
  - Determinísticas: técnicas de **consistência**
  - Não determinísticas: **pesquisa** (“search”)
- Forma de resolução:
  - Computação determinística é efetuada sempre que possível
    - Durante a propagação, sempre que é colocada uma restrição
  - Pesquisa é efetuada quando não é possível efetuar mais propagação

# 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***”

# Mecanismo *Constrain and Generate*



- Pesquisa do tipo “***generate and test***” é a típica pesquisa em profundidade (na árvore completa)
- Mecanismo “***constrain and generate***” permite eliminar ramos da árvore que se sabem não conduzir a soluções válidas

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]



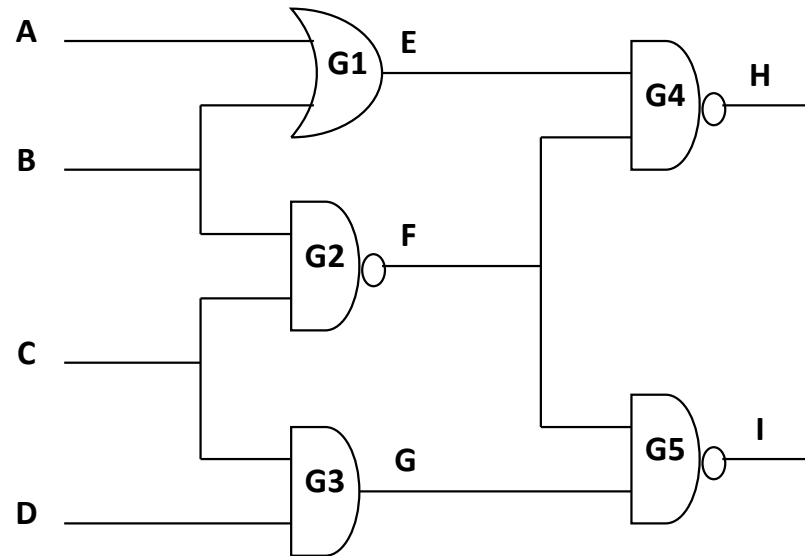
# Restrições: Exemplos de Problemas

- 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

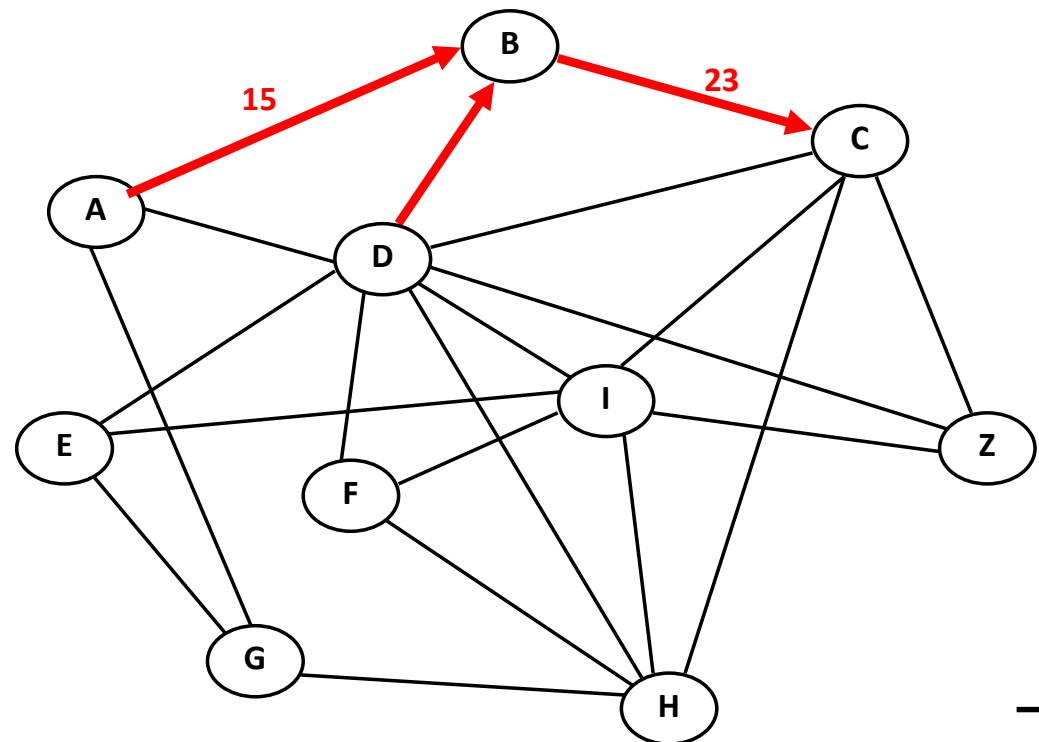
- Determinação do tráfego em redes de telecomunicações, transporte de água/eletricidade, ...
  - Objetivo:
    - Determinar a quantidade de informação que flui em cada um dos troços de uma rede de comunicações
  - Variáveis:
    - Modelam o fluxo nos diversos troços
  - Domínio:
    - Reais não negativos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

$$X_{ab} \leq 15$$

$$X_{ab} + X_{db} = X_{bc}$$



# 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

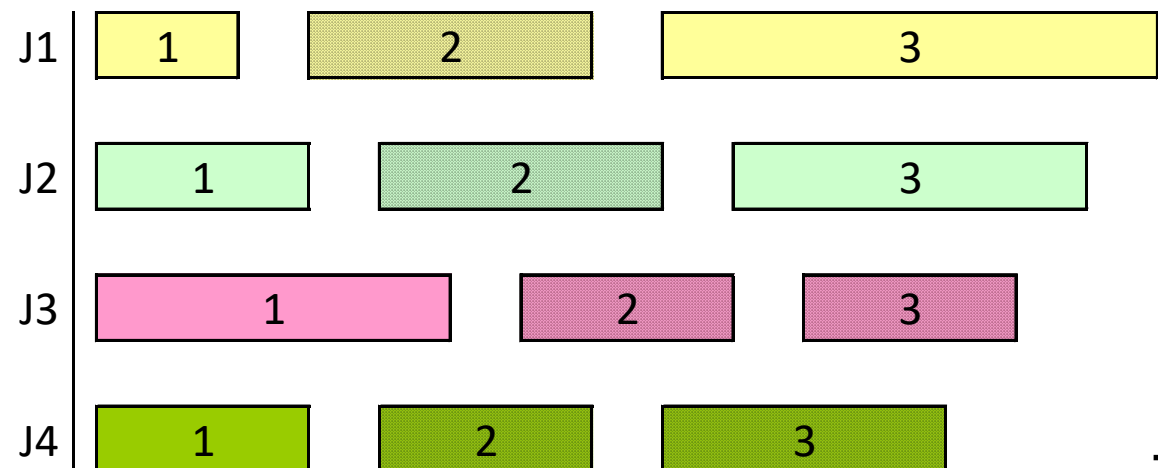


# Escalonamento de Tarefas

- Job-Shop

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

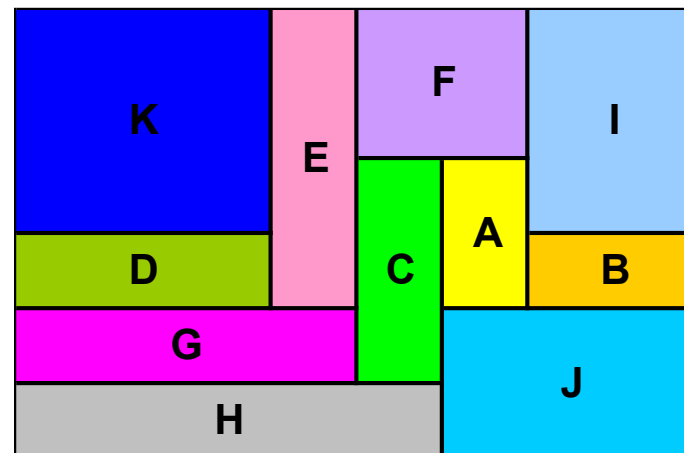
- Não ultrapassar os limites do “contentor”

$$X_a + Lx_a \leq X_{\max} \qquad Y_a + Ly_a \leq Y_{\max}$$

- Não sobrepor componentes

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

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



## Decisão vs. Otimização

- Em certos (muitos) casos o que se pretende determinar não é uma solução qualquer para o problema, mas sim a *melhor solução* (segundo um dado critério)
  - Necessário definir uma função de avaliação

# 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 custo / distância percorrida
- Empacotamento ou Corte
  - Corte / colocação do maior número de peças
  - Menor desperdício de espaço / 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:
  - Soma e Produto
  - Criptograma SEND + MORE = MONEY
  - Quadrado Mágico
  - N-Rainhas
  - Zebra Puzzle
- Variáveis e Domínios?
- Restrições?
- Solução do problema?

# Soma e Produto

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

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

# 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	
-----					
	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
-----				
M	O	N	E	Y

	1	0	1	1
		9	5	6
				7
+		1	0	8
				5
-----				
	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:

```
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, % M #= 1
```

```
    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).
```

# 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:

```
magic3(Vars):-
```

```
  Vars=[A,B,C,D,E,F,G,H,I],
```

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

```
  all_distinct(Vars),
```

```
  A + B + C #= Soma,   A + D + G #= Soma,
```

```
  D + E + F #= Soma,   B + E + H #= Soma,
```

```
  G + H + I #= Soma,   C + F + I #= Soma,
```

```
  A + E + I #= Soma,
```

```
  C + E + G #= Soma,
```

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

8	3	4
1	5	9
6	7	2

6	1	8
7	5	3
2	9	4

4	3	8
9	5	1
2	7	6

8	1	6
3	5	7
4	9	2

6	7	2
1	5	9
8	3	4

2	7	6
9	5	1
4	3	8

2	9	4
7	5	3
6	1	8

4	9	2
3	5	7
8	1	6

# 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:

```
magic3(Vars):-  
    Vars=[A,B,C,D,E,F,G,H,I],  
    domain(Vars,1,9),  
    % Soma is (9+1)*3//2, % aumenta a eficiência  
    all_distinct(Vars),  
    A + B + C #= Soma,  A + D + G #= Soma,  
    D + E + F #= Soma,  B + E + H #= Soma,  
    G + H + I #= Soma,  C + F + I #= Soma,  
    A + E + I #= Soma,  C + E + G #= Soma,  
    % A #< B, A #< C, A #< D, B #< D, % eliminar simetrias  
    labeling([], Vars).
```

2	7	6
9	5	1
4	3	8

# 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+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),  
    all_distinct(Cols), % redundante mas diminui tempo  
    constrain(Cols),  
    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, todos também diferentes. 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 Luky 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_distinct(Nac), all_distinct(Ani), all_distinct(Beb),  
    all_distinct(Cor), all_distinct(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

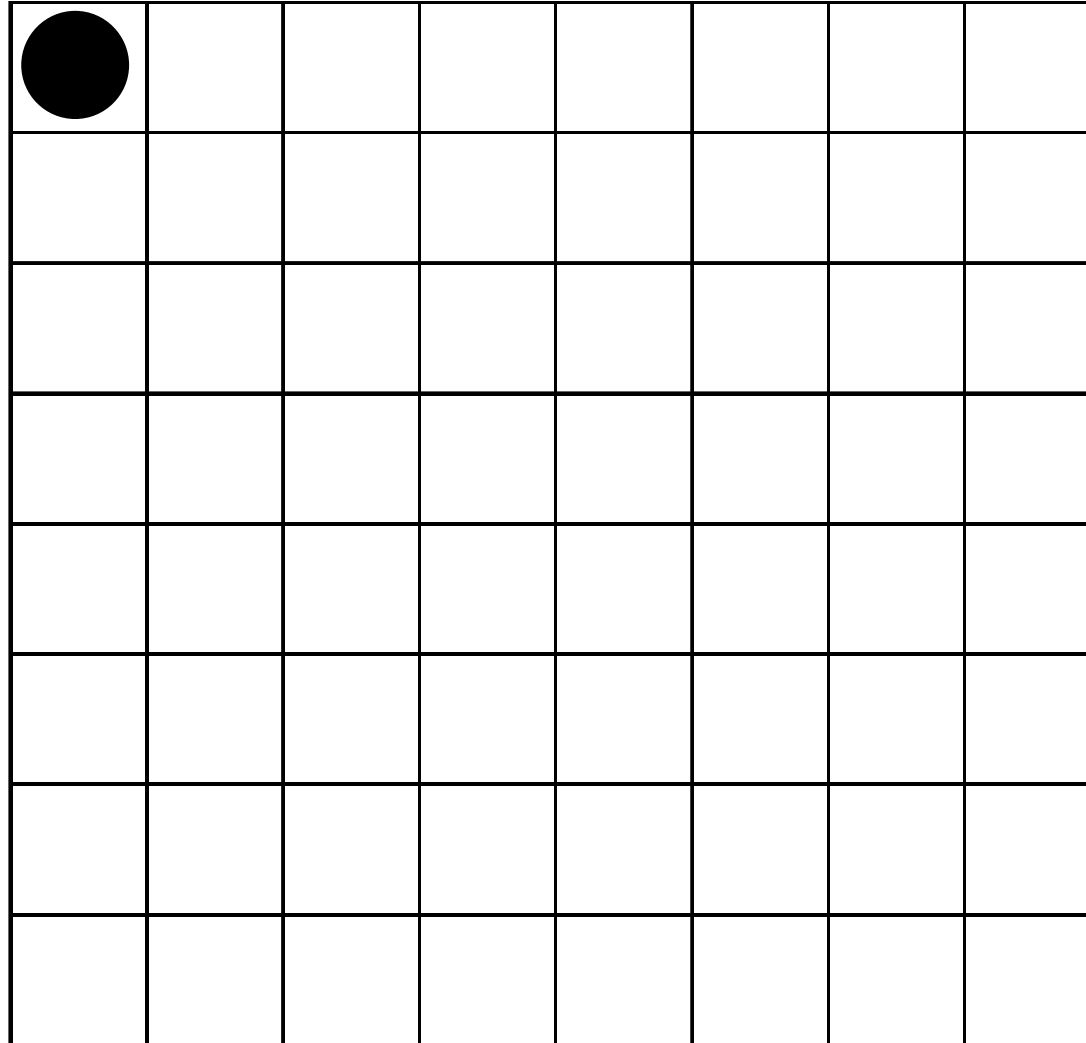
- Complexidade cresce exponencialmente
- Os problemas de interesse são geralmente NP-completos
- Tempo de pesquisa exaustiva das  $k^n$  possíveis soluções:  
(assumindo duração de 1  $\mu$ s para cada operação elementar)

$k \backslash n$	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

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

# Retrocesso



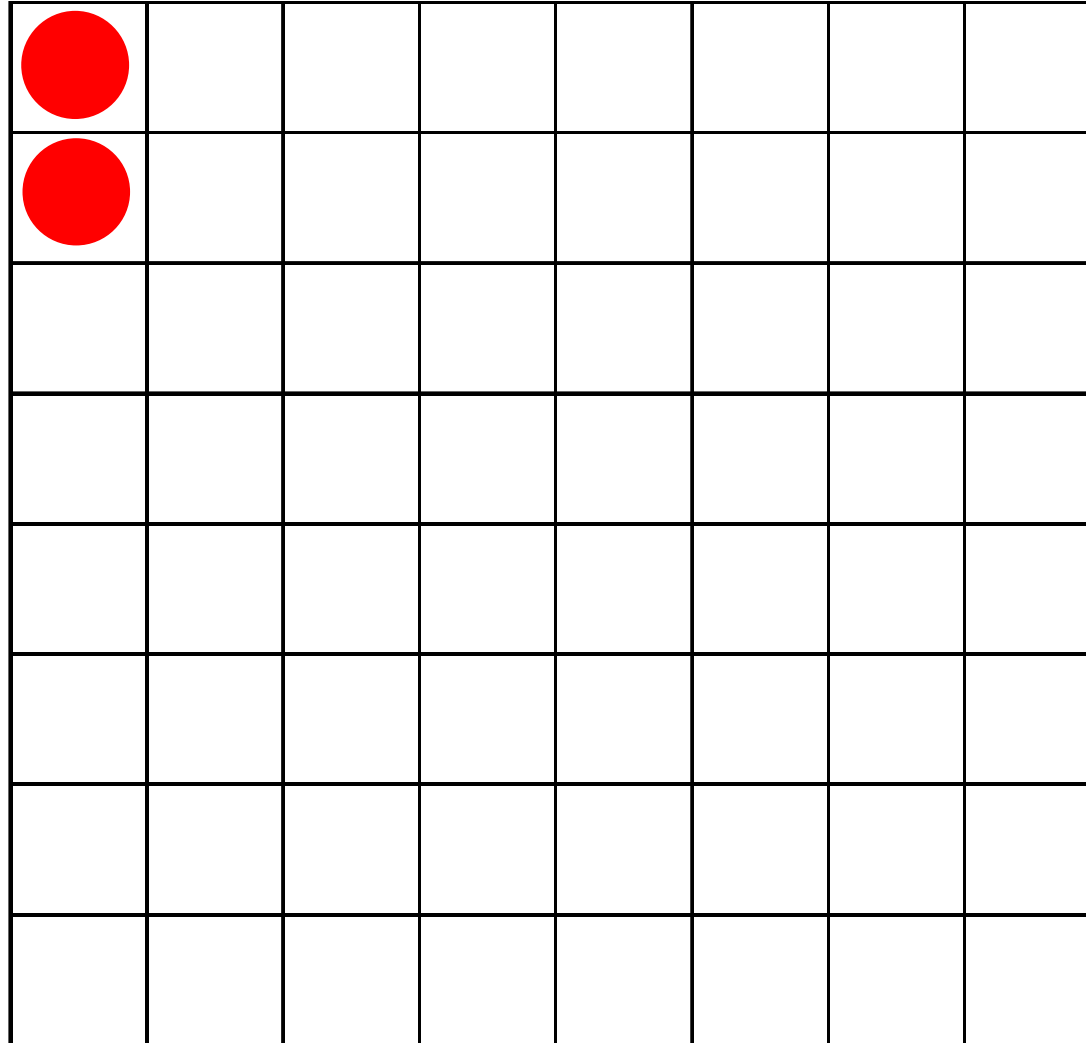
Testes 0

Retrocessos 0



# Retrocesso

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

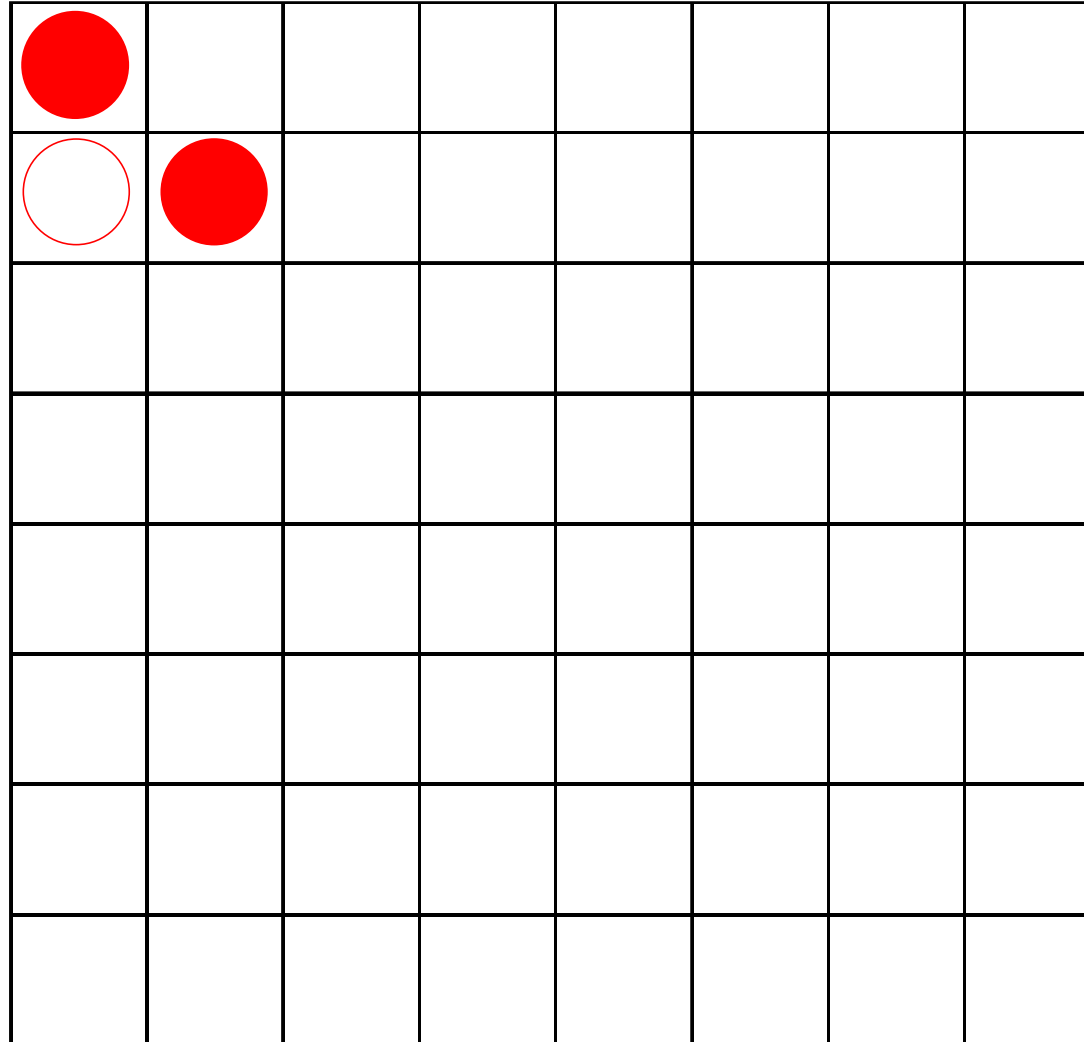


Testes  $0 + 1 = 1$

Retrocessos 0

# Retrocesso

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

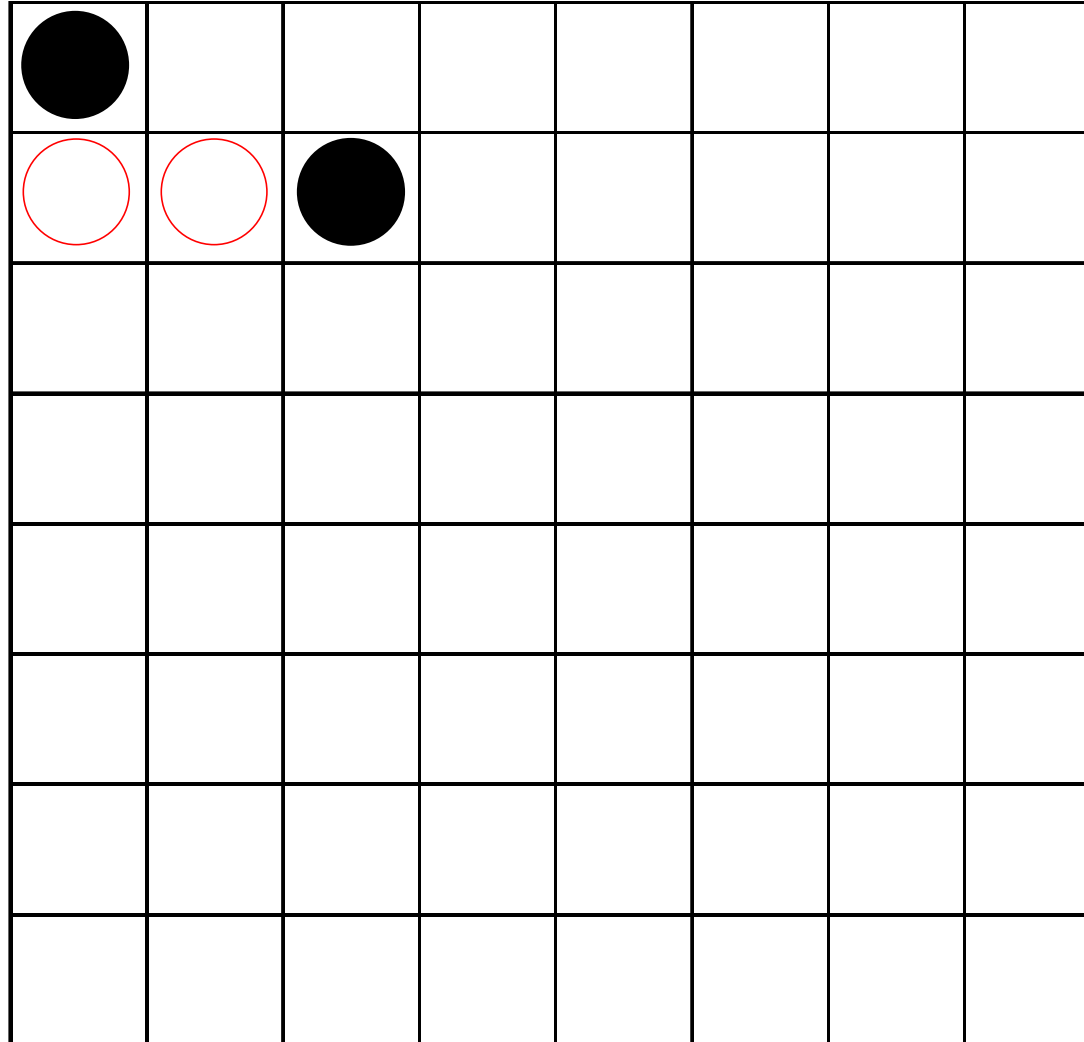


Testes  $1 + 1 = 2$

Retrocessos 0

# Retrocesso

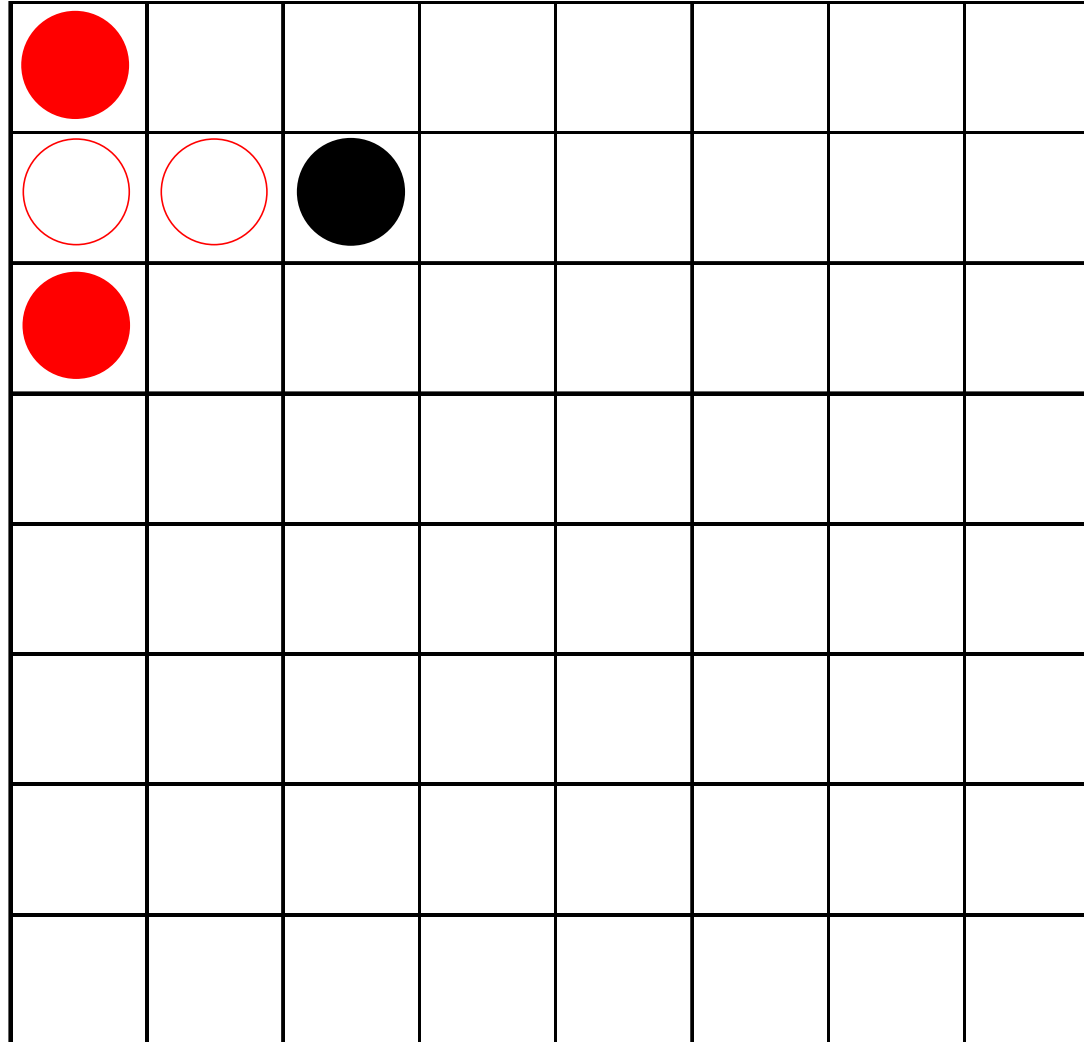
$Q1 \neq Q2, \quad L1+Q1 \neq L2+Q2, \quad L1+Q2 \neq L2+Q1.$



Testes  $2 + 1 = 3$

Retrocessos 0

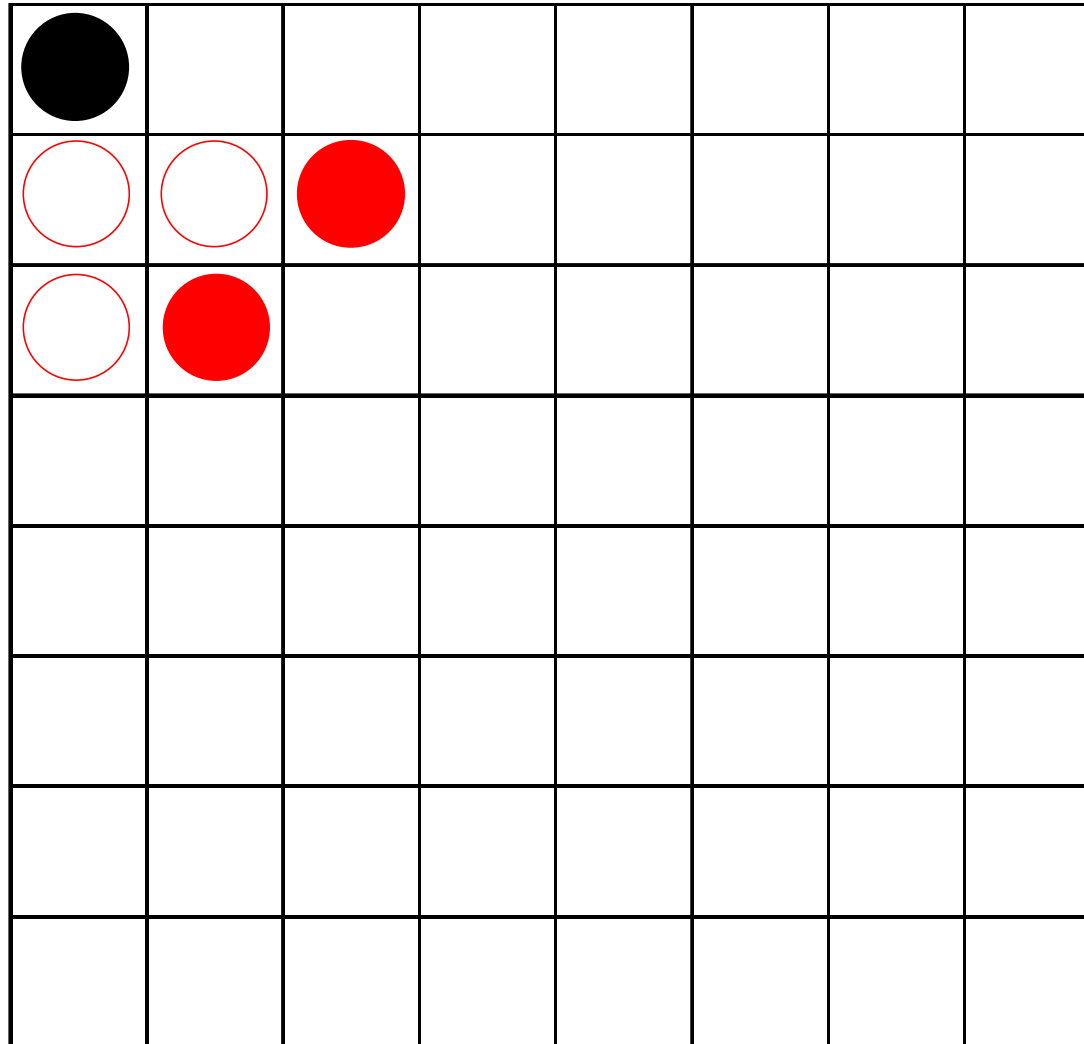
# Retrocesso



Testes  $3 + 1 = 4$

Retrocessos 0

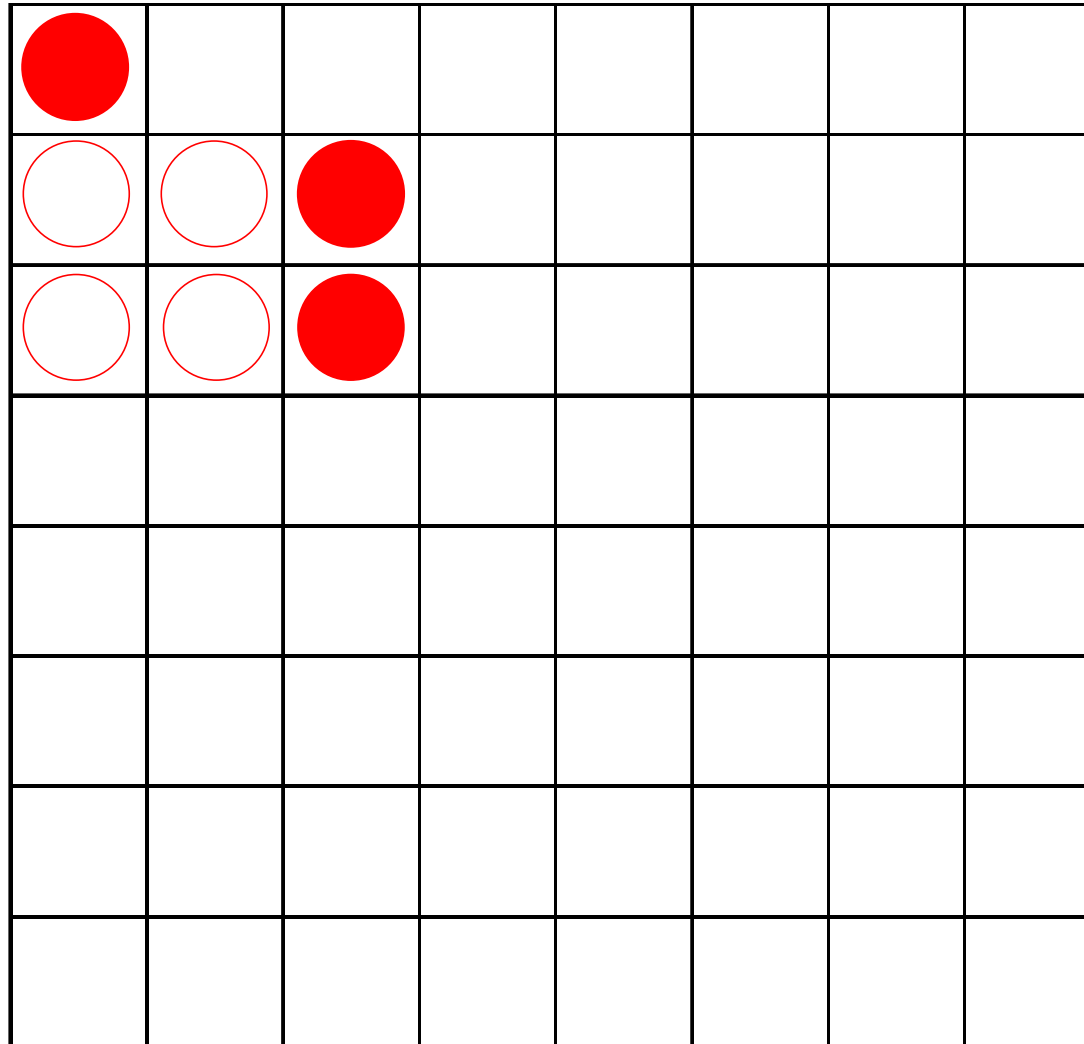
# Retrocesso



Testes  $4 + 2 = 6$

Retrocessos 0

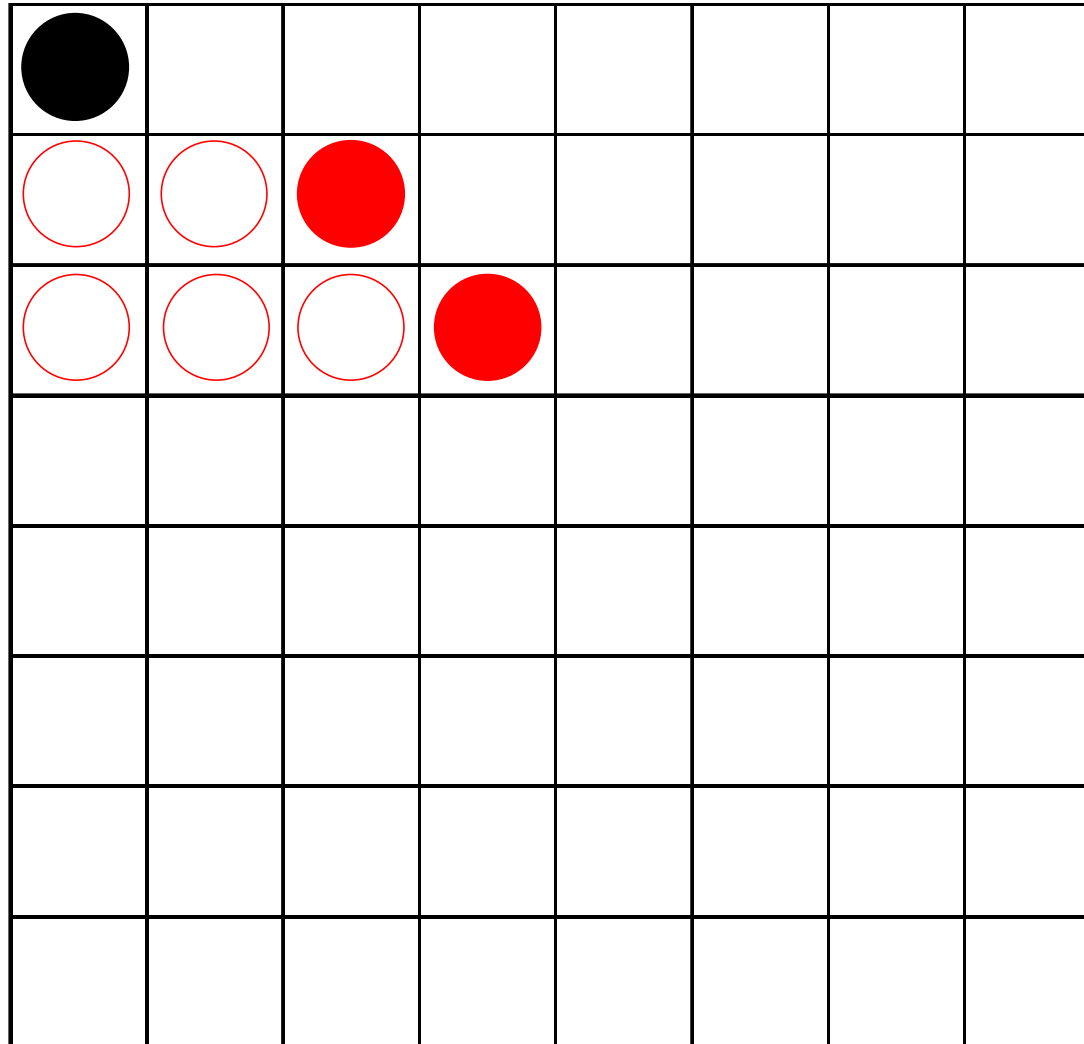
# Retrocesso



Testes  $6 + 1 = 7$

Retrocessos 0

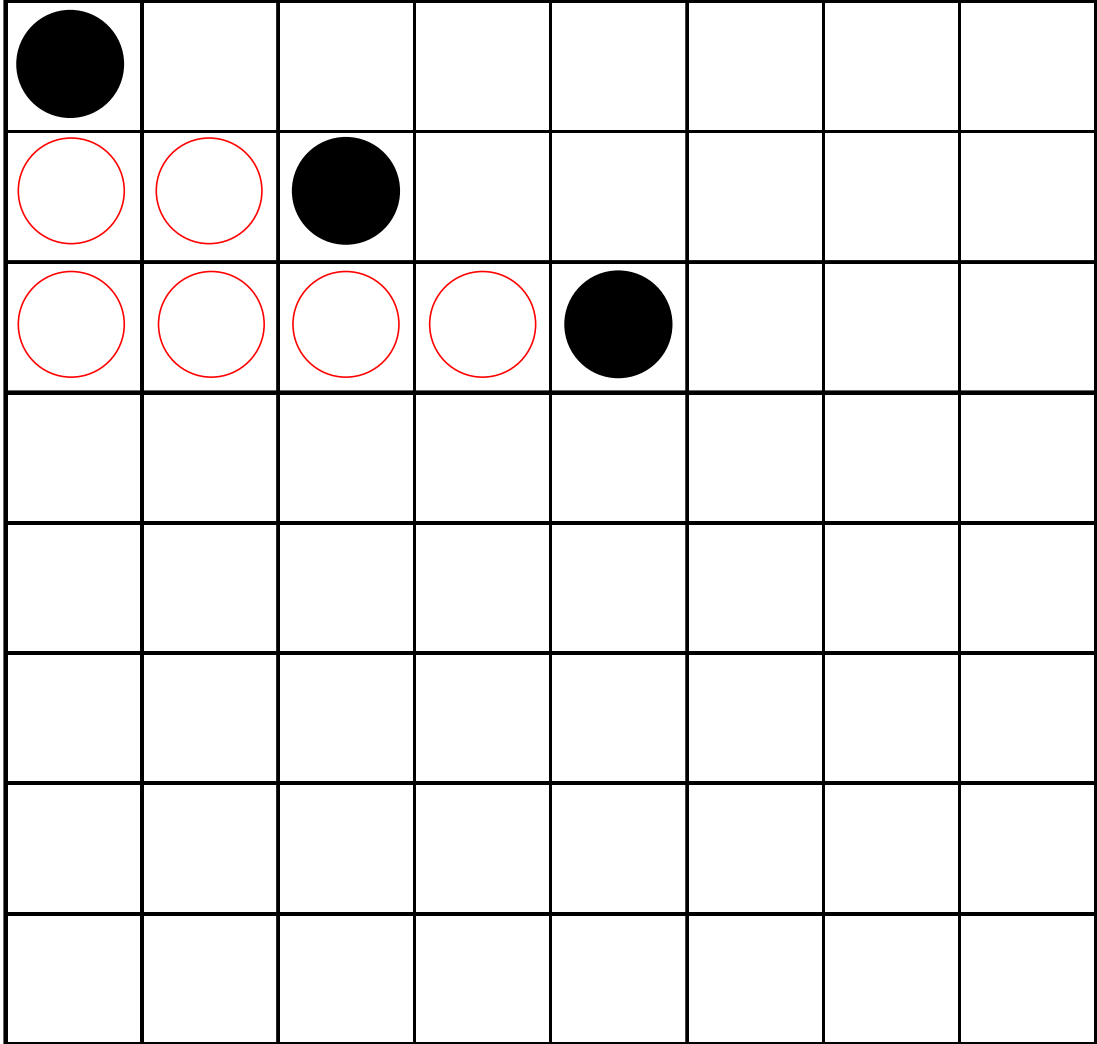
# Retrocesso



Testes  $7 + 2 = 9$

Retrocessos 0

# Retrocesso

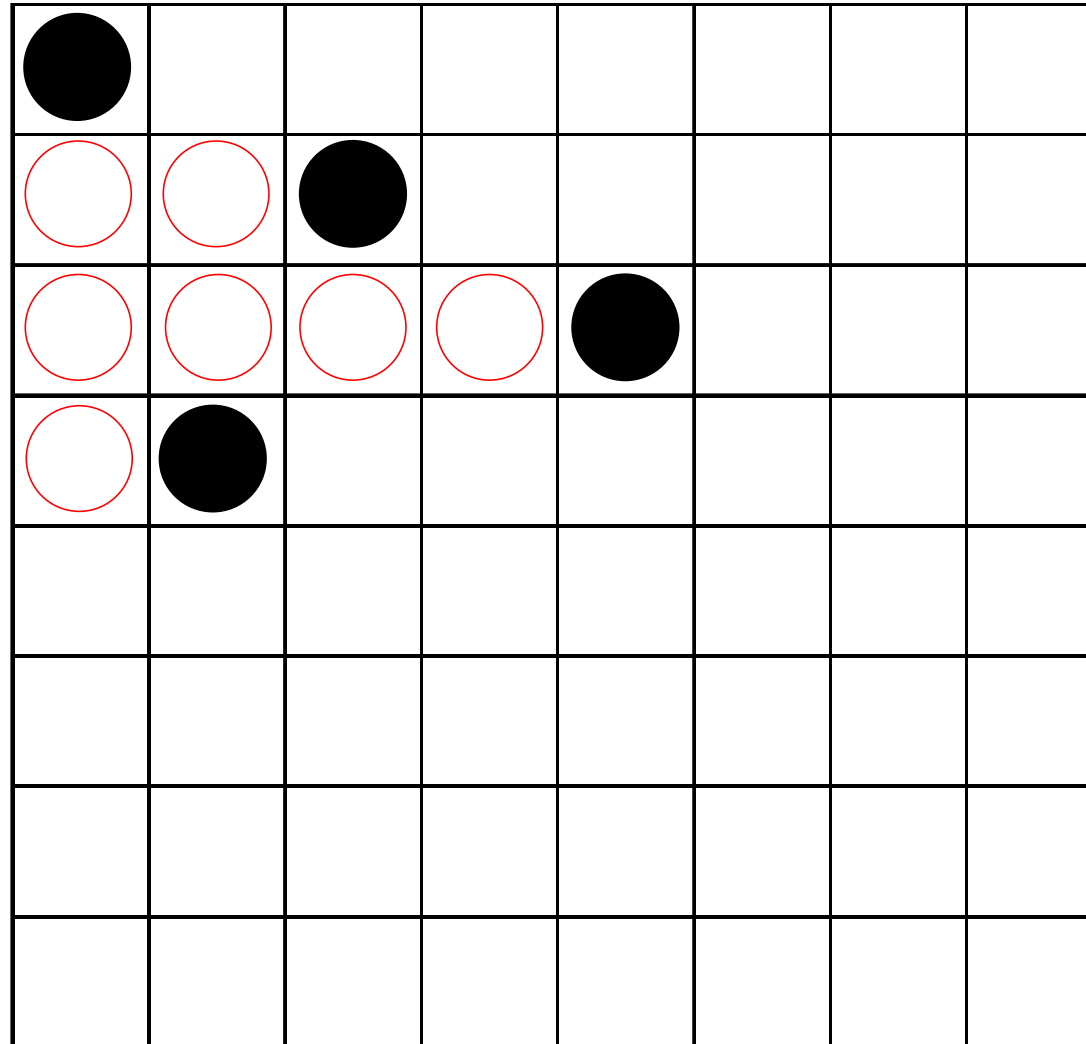


Testes  $9 + 2 = 11$

# Retrocessos 0



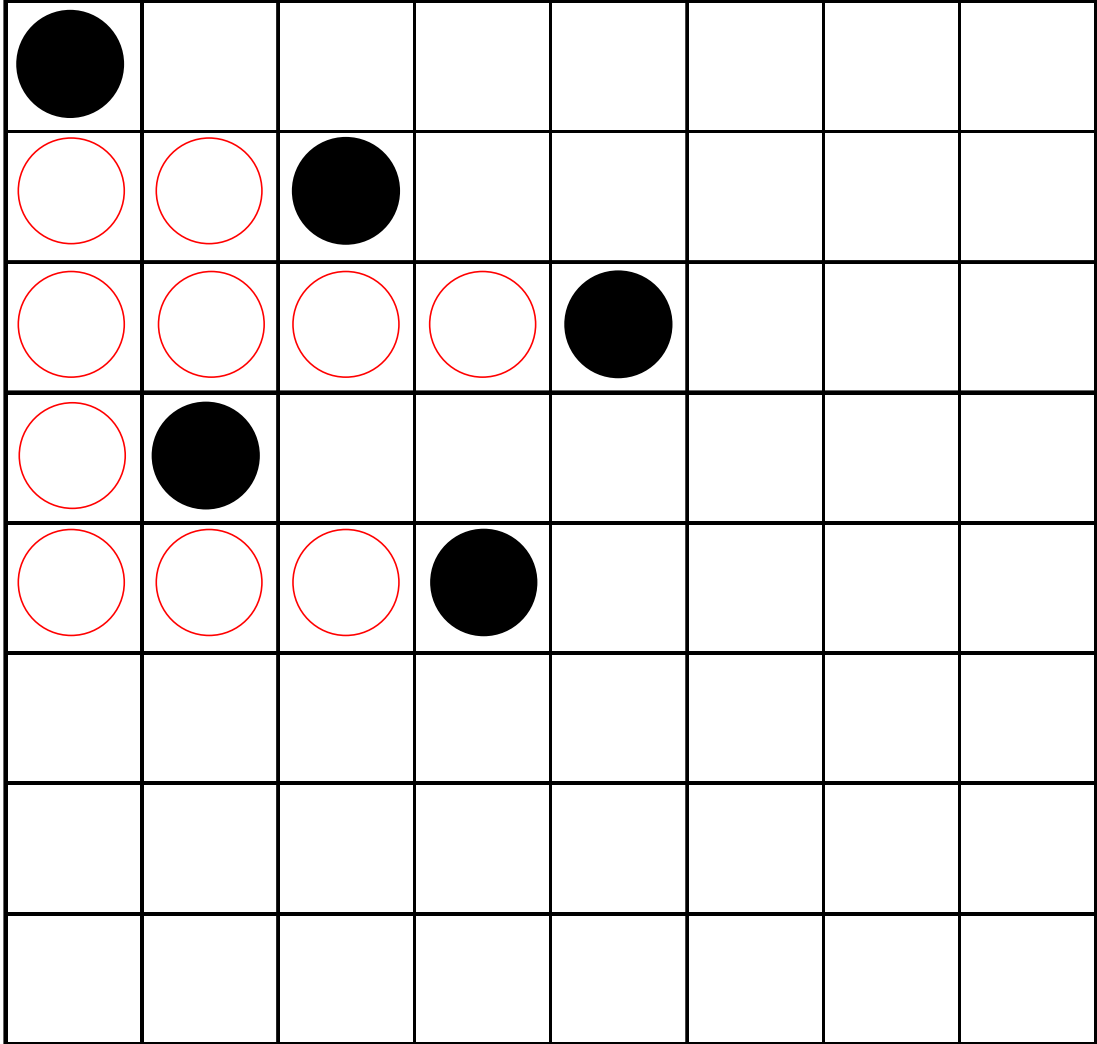
# Retrocesso



Testes  $11 + 1 + 3 = 15$

Retrocessos 0

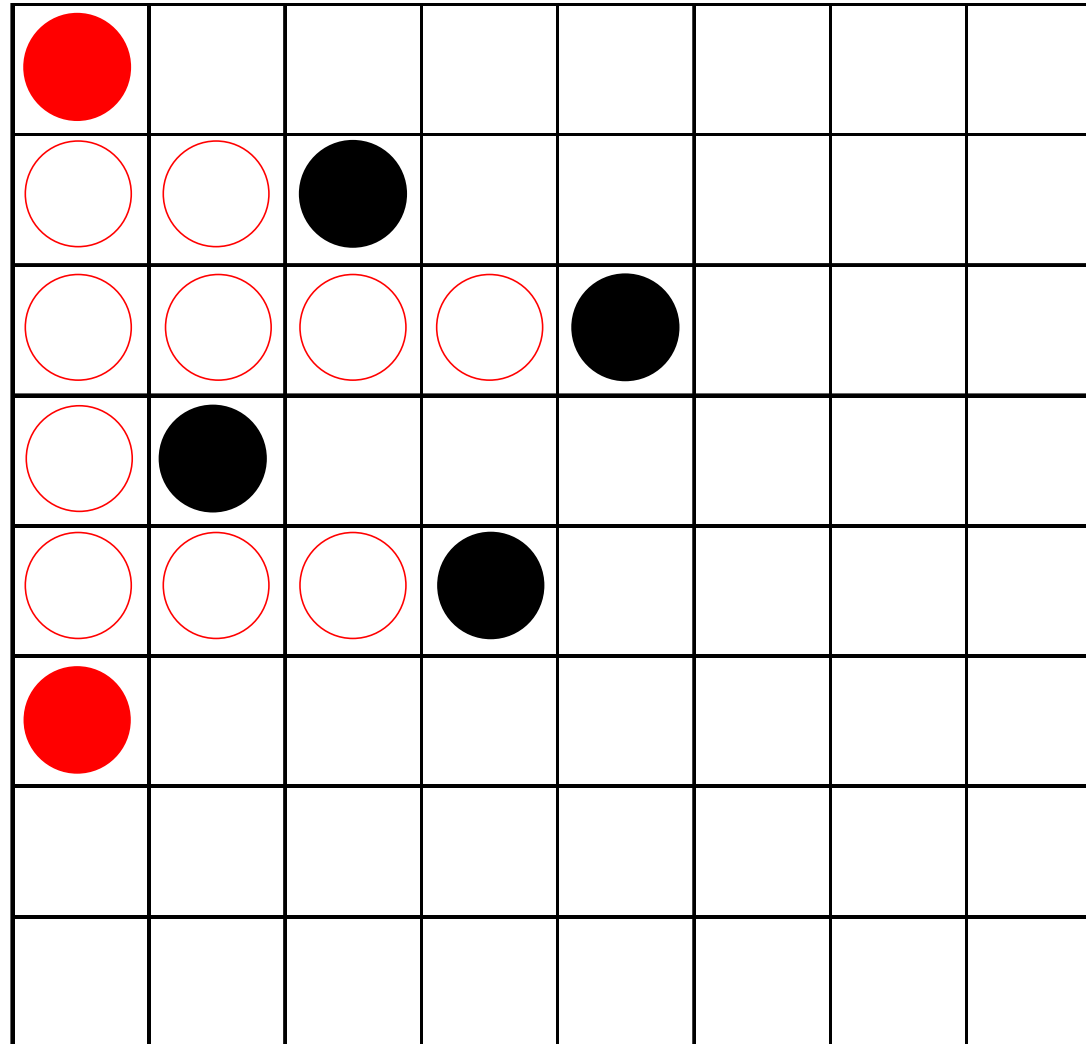
# Retrocesso



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

# Retrocessos 0

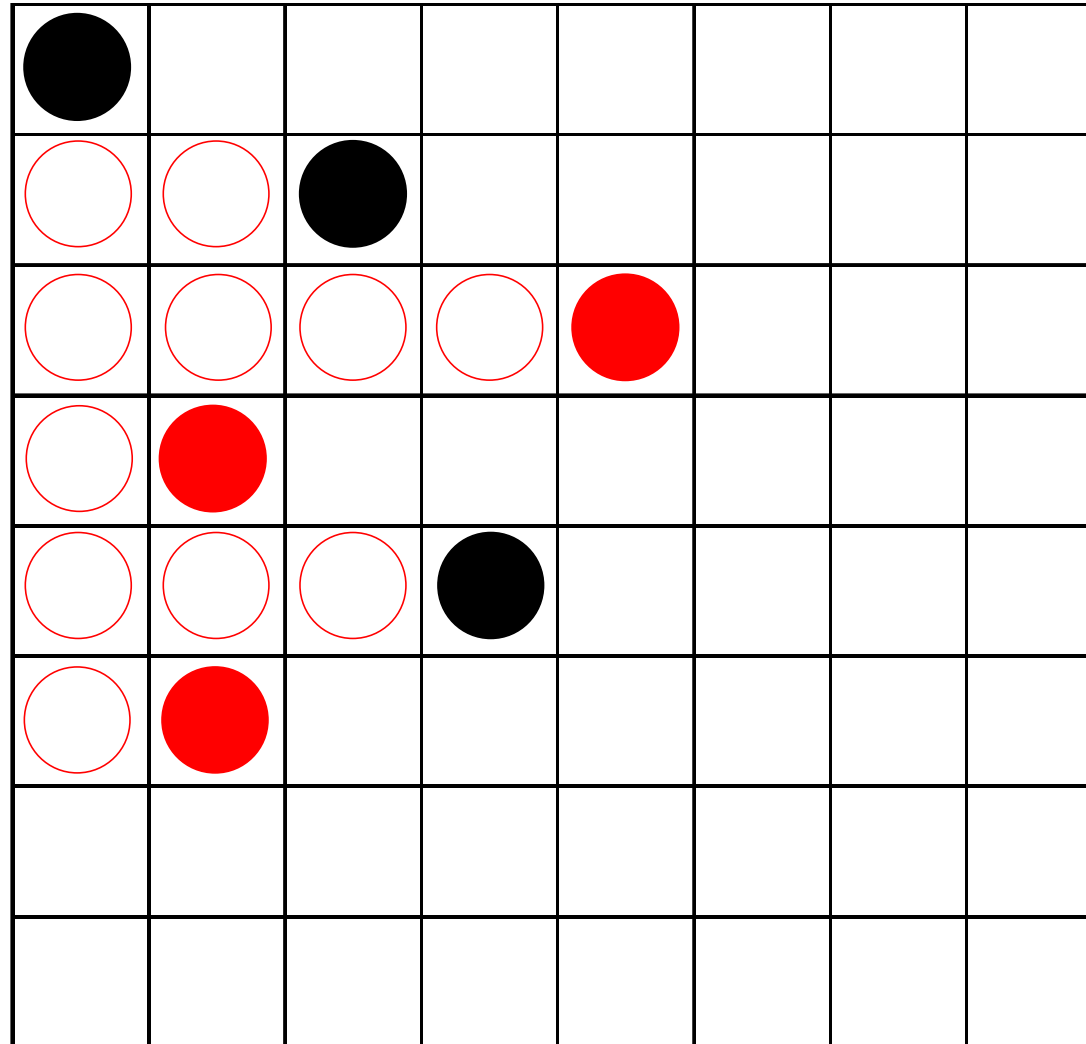
# Retrocesso



Testes  $26 + 1 = 27$

Retrocessos 0

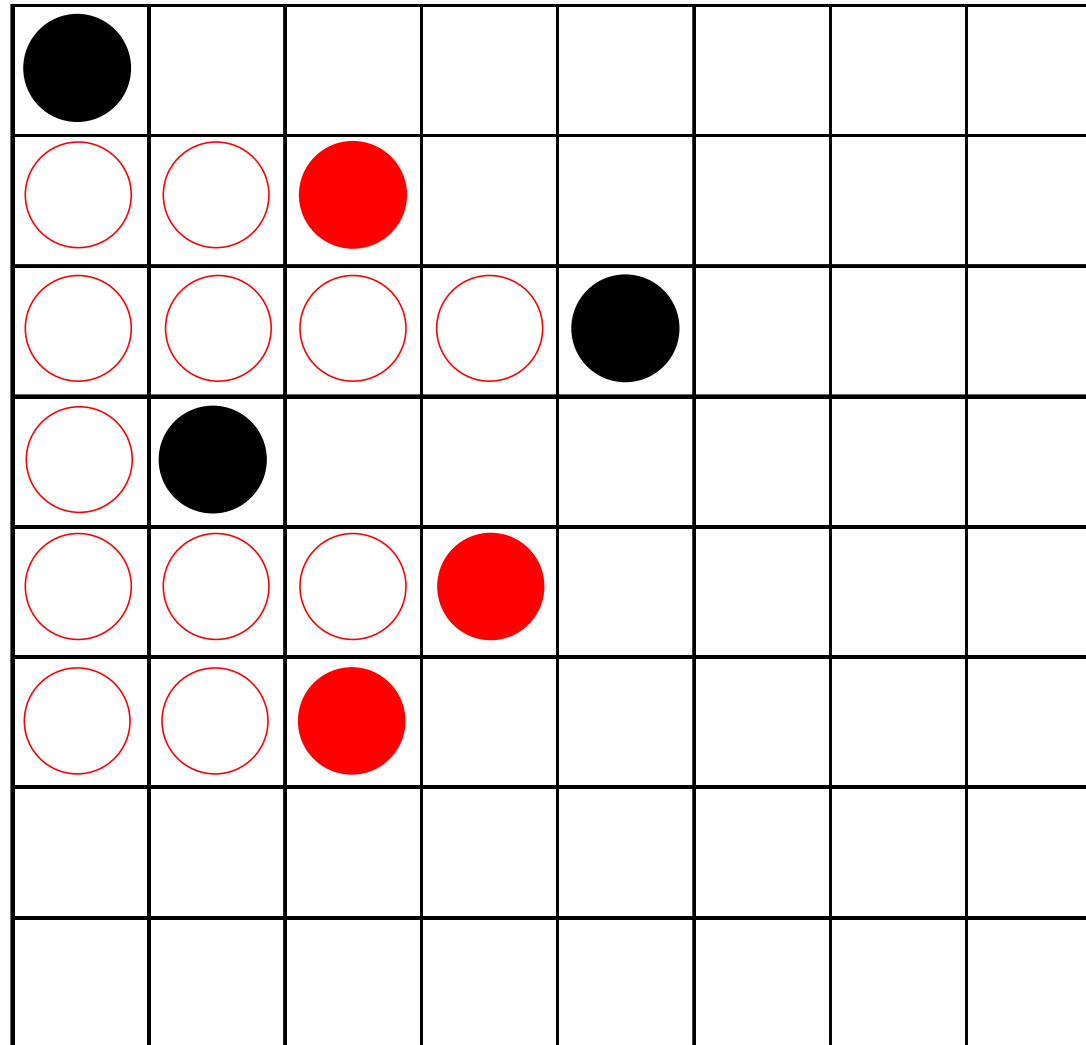
# Retrocesso



Testes  $27 + 3 = 30$

Retrocessos 0

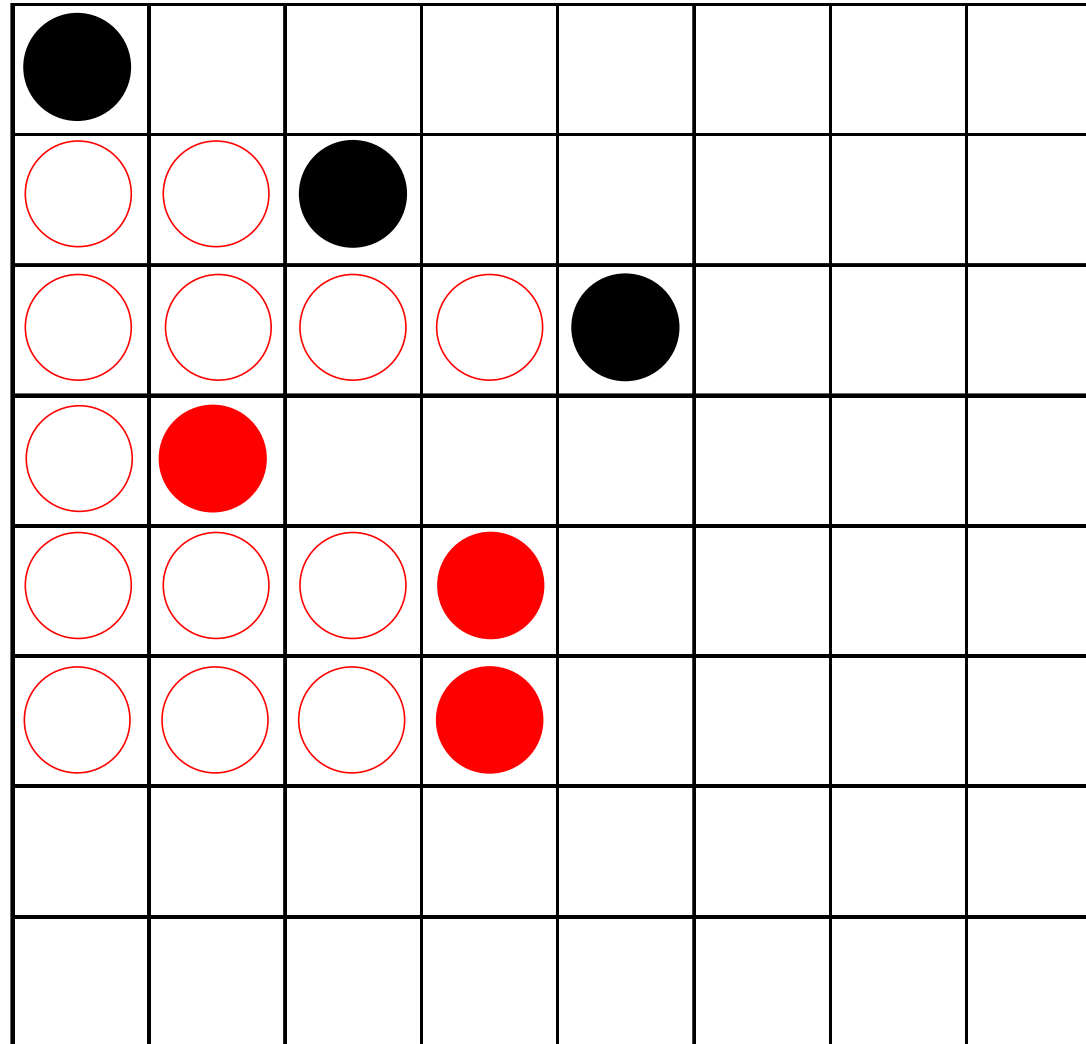
# Retrocesso



Testes  $30 + 2 = 32$

Retrocessos 0

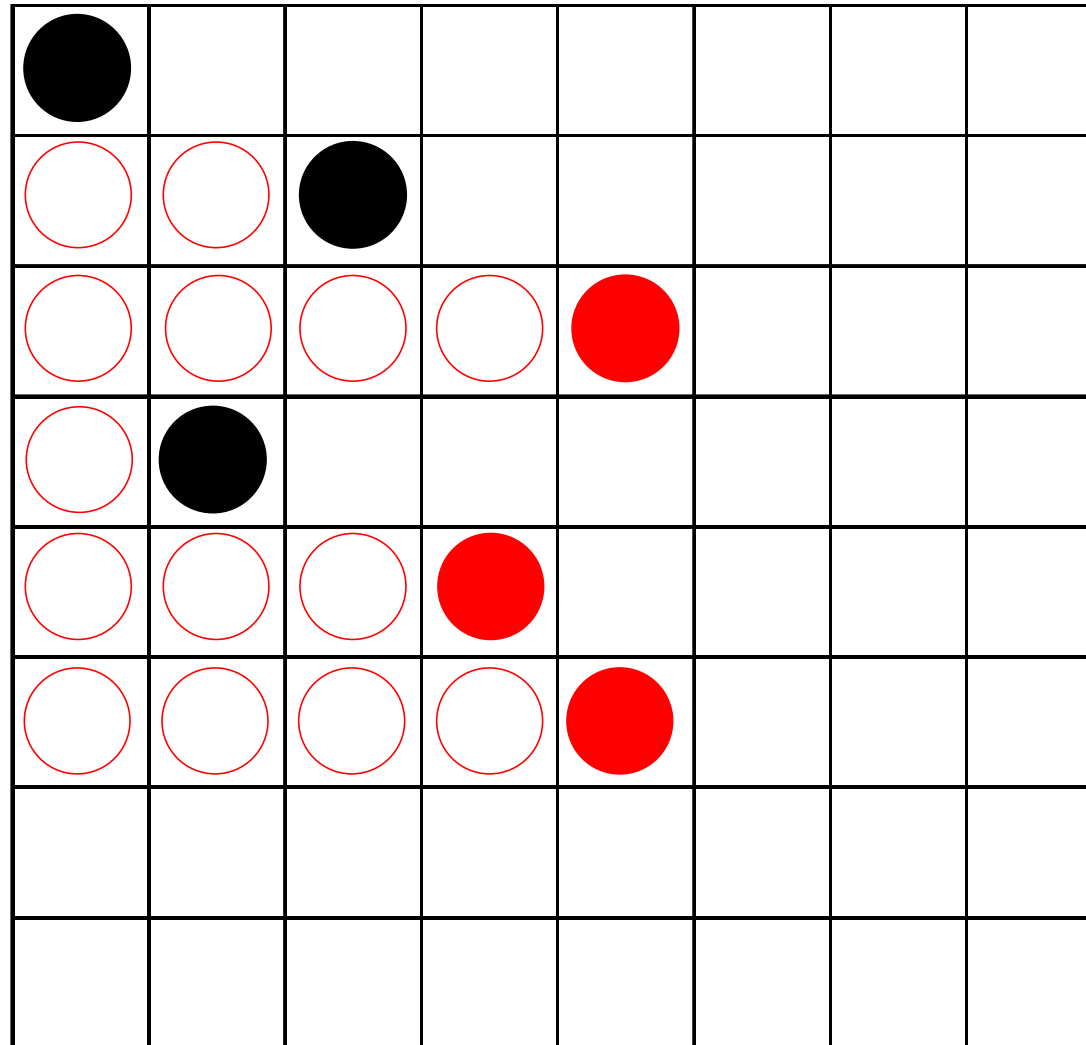
# Retrocesso



Testes  $32 + 4 = 36$

Retrocessos 0

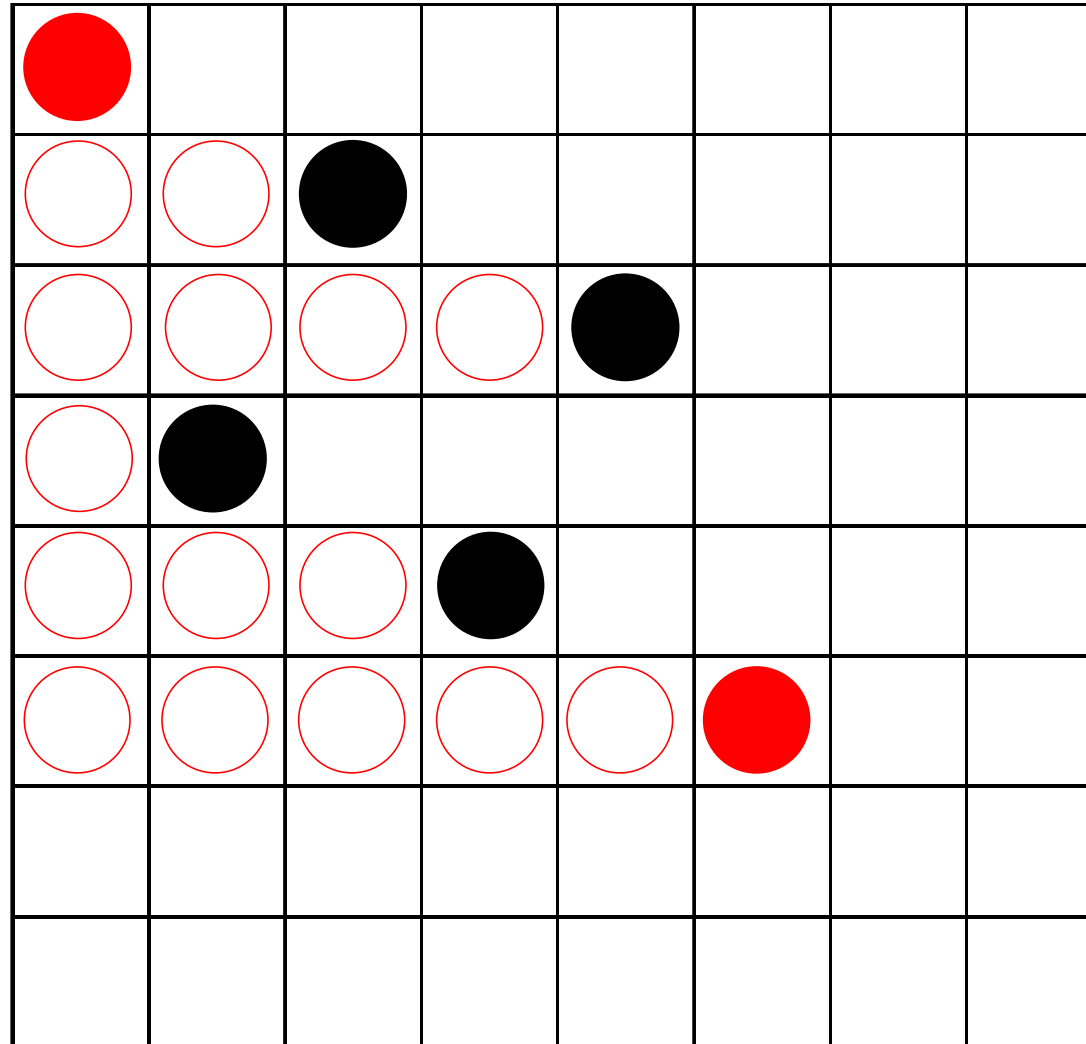
# Retrocesso



Testes  $36 + 3 = 39$

Retrocessos 0

# Retrocesso

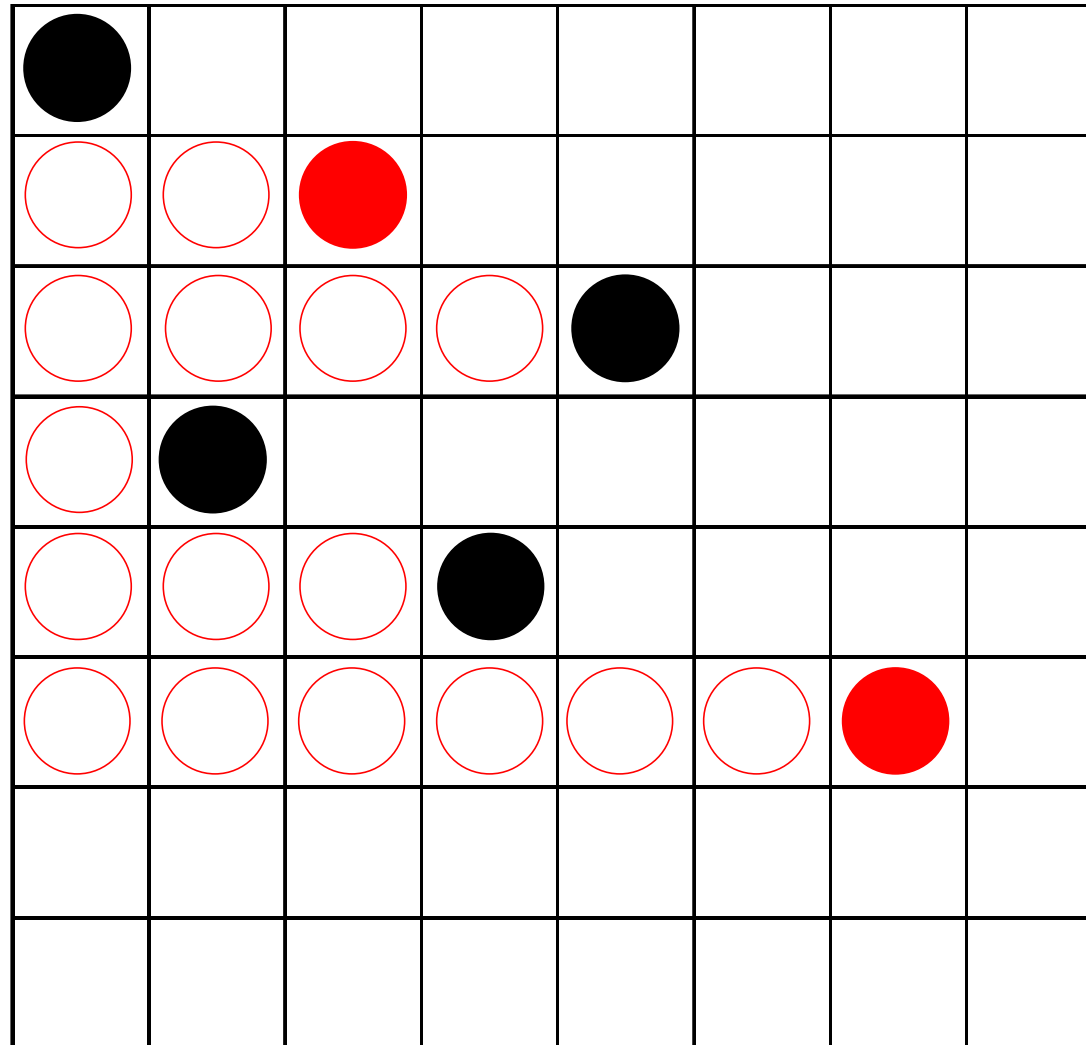


Testes  $39 + 1 = 40$

Retrocessos 0



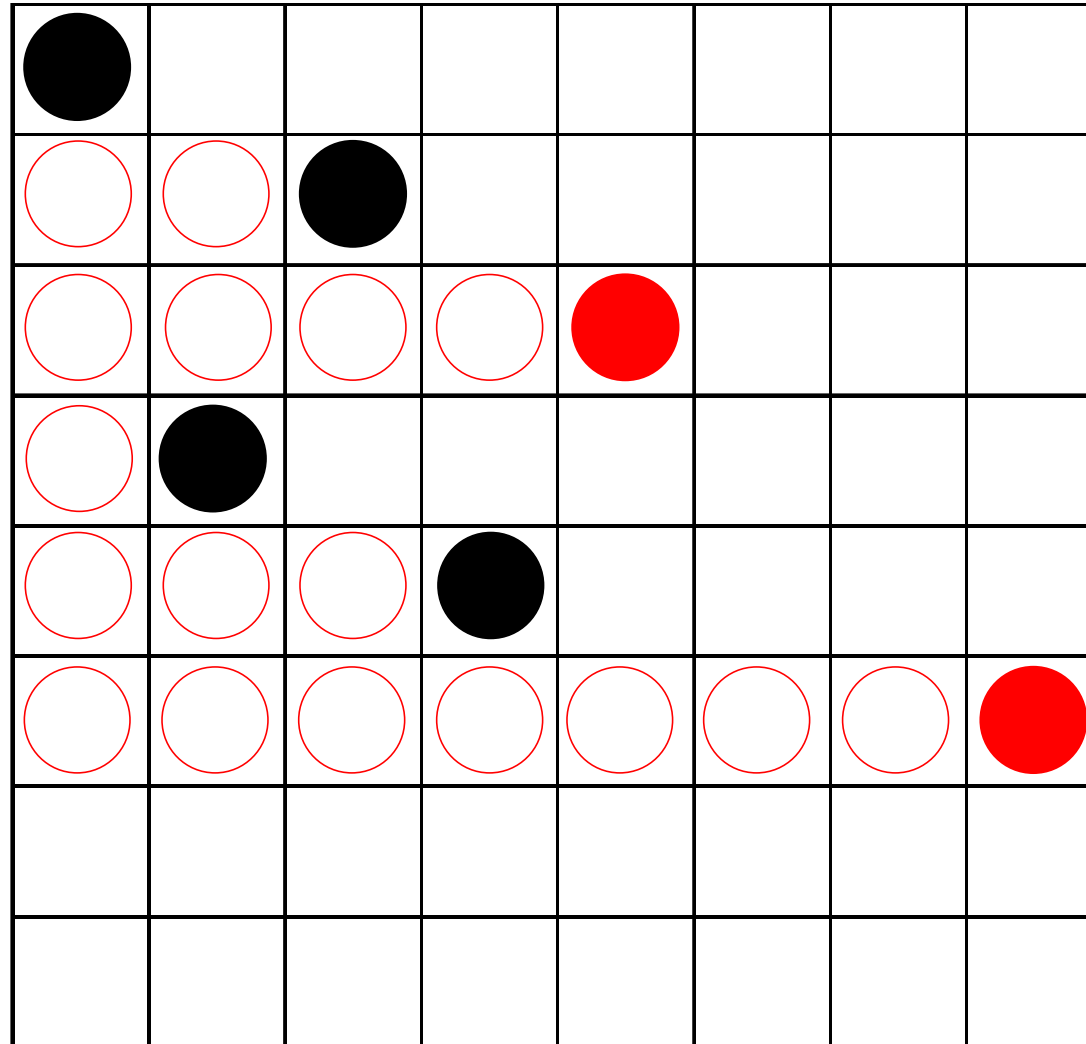
# Retrocesso



Testes  $40 + 2 = 42$

Retrocessos 0

# Retrocesso

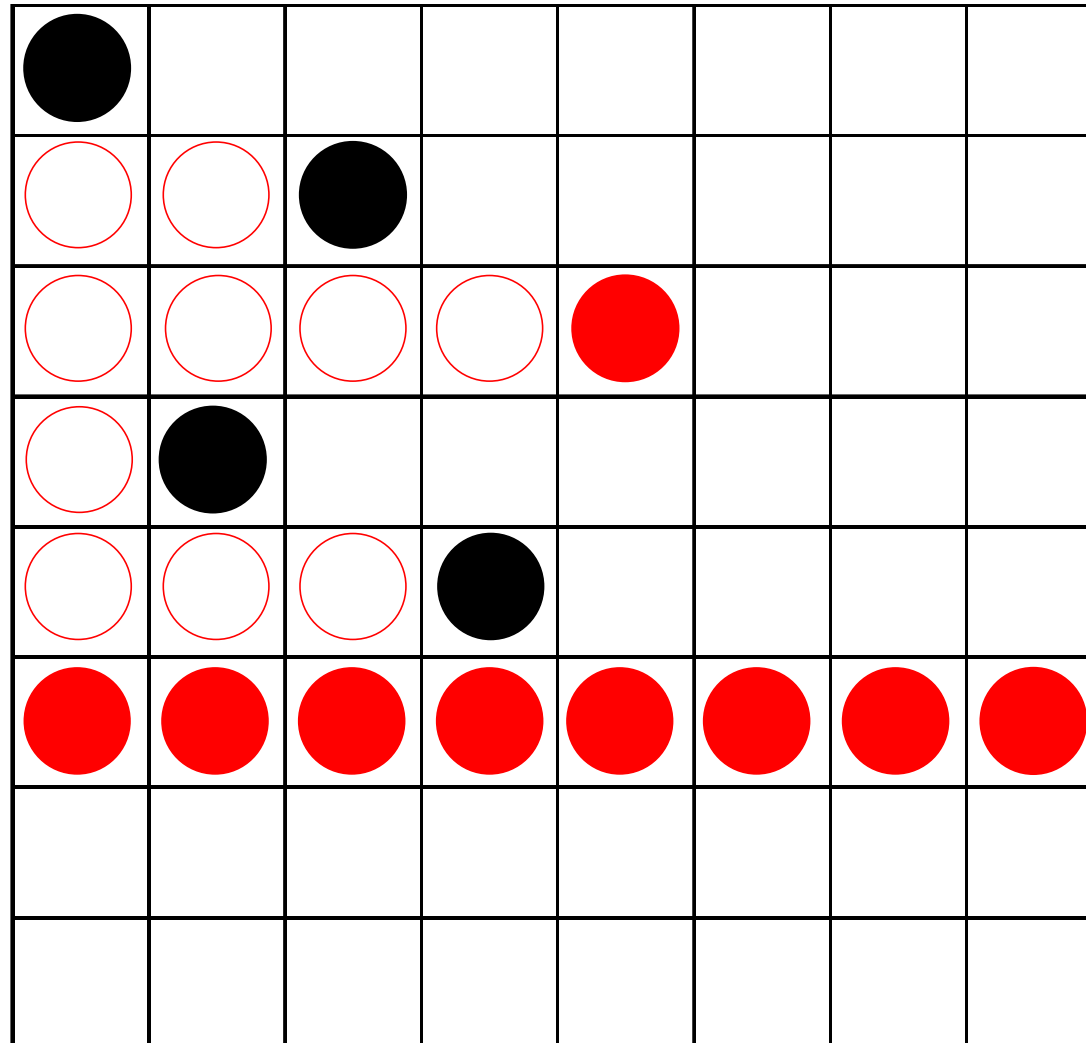


Testes  $42 + 3 = 45$

Retrocessos 0

# Retrocesso

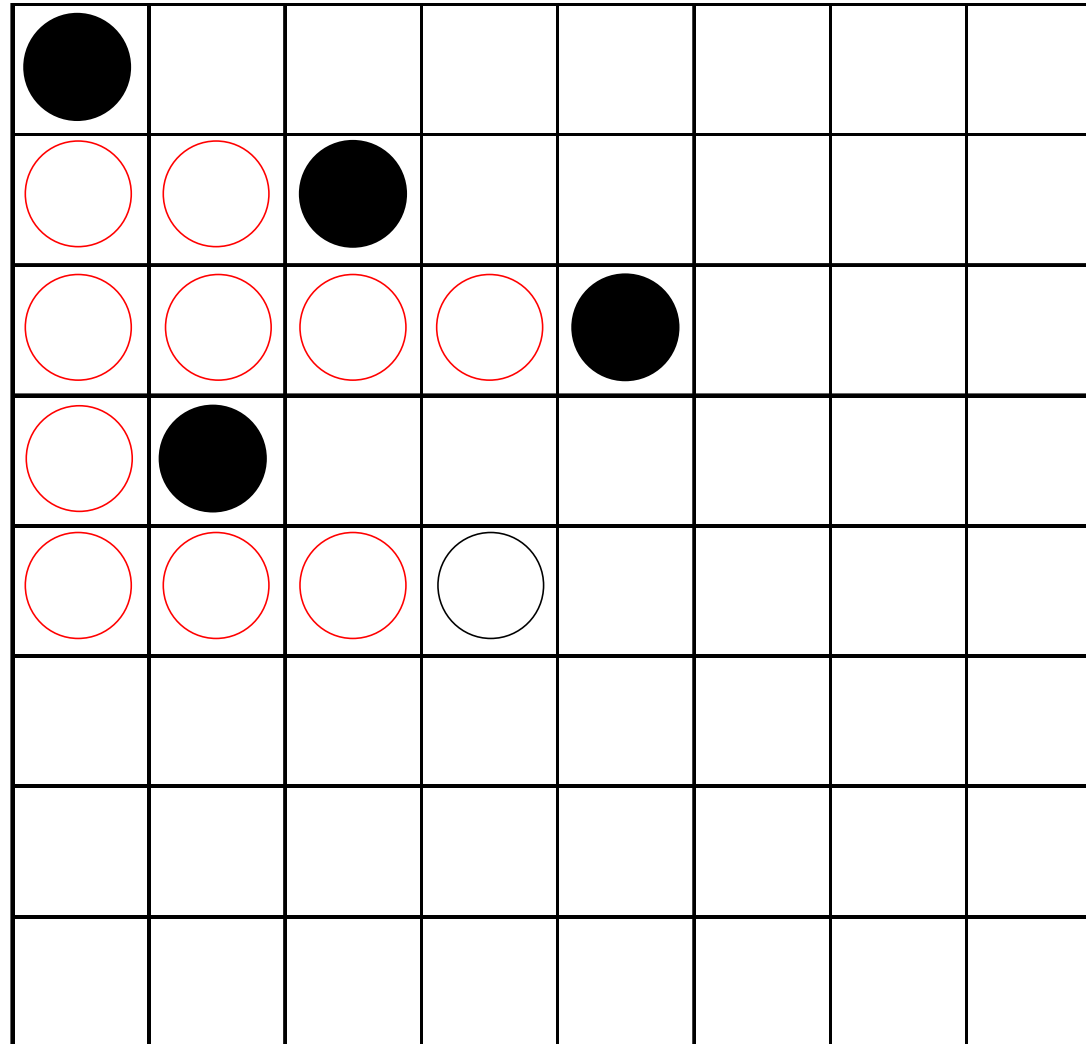
Falha  
6  
Retrocede  
5



Testes 45

Retrocessos 0

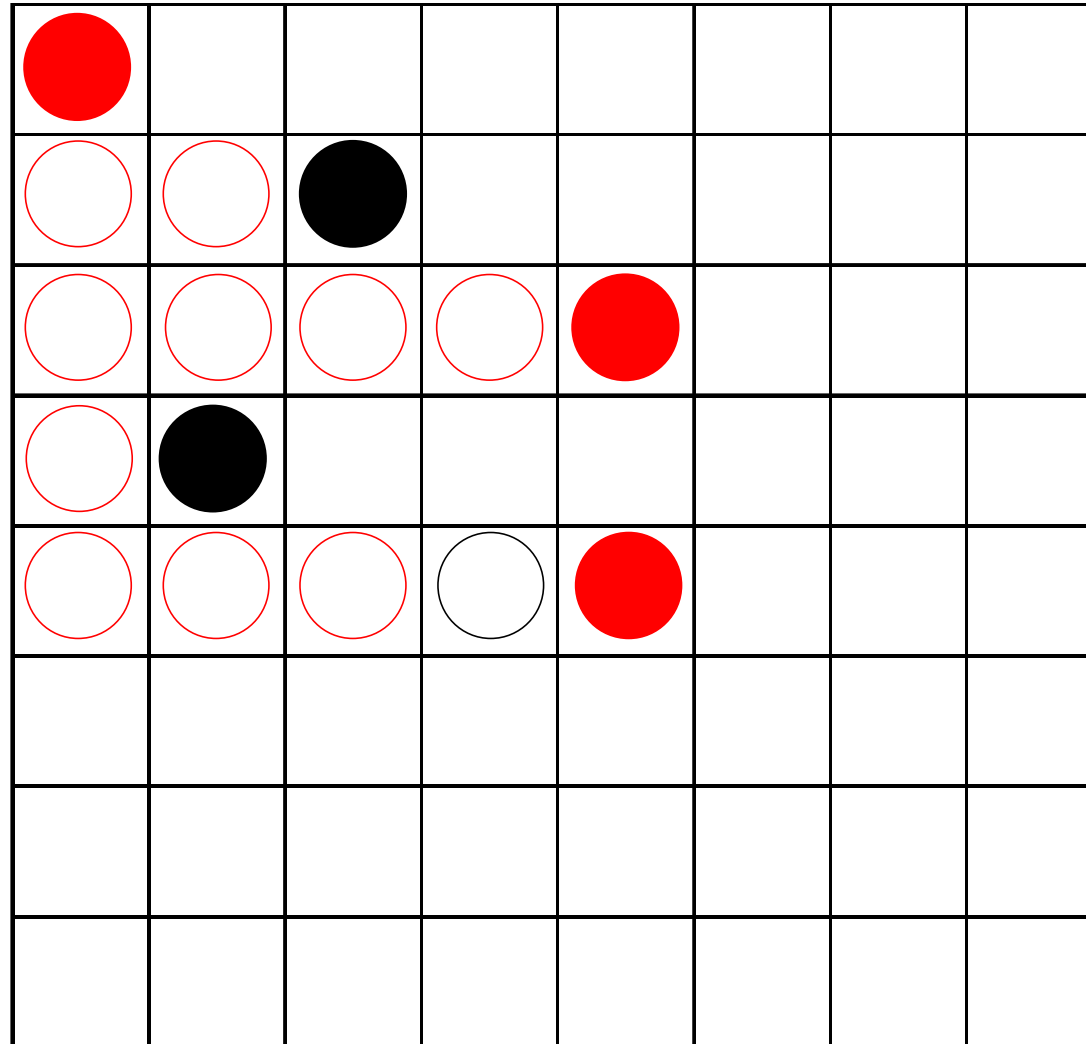
# Retrocesso



Testes 45

Retrocessos 1

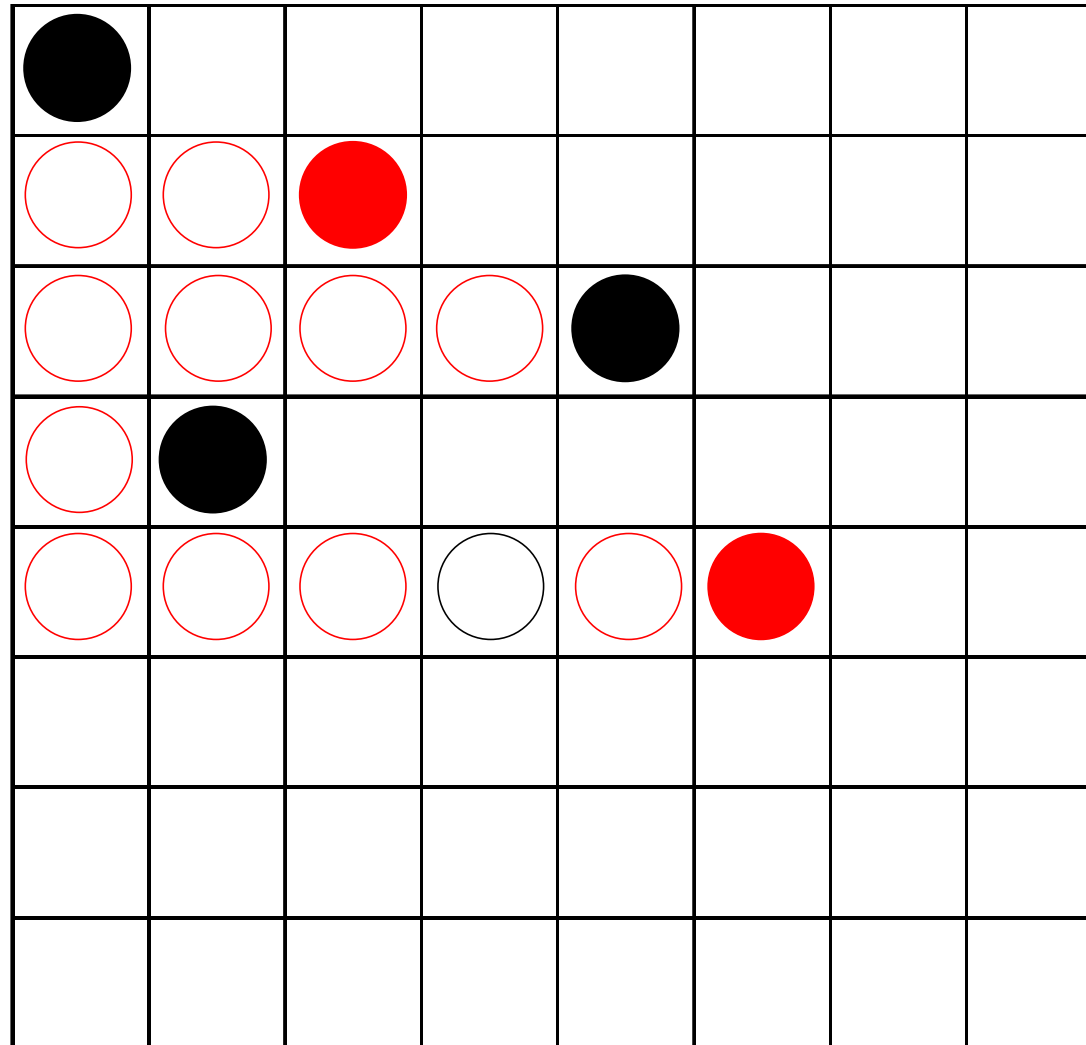
# Retrocesso



Testes  $45 + 1 = 46$

## Retrocessos 1

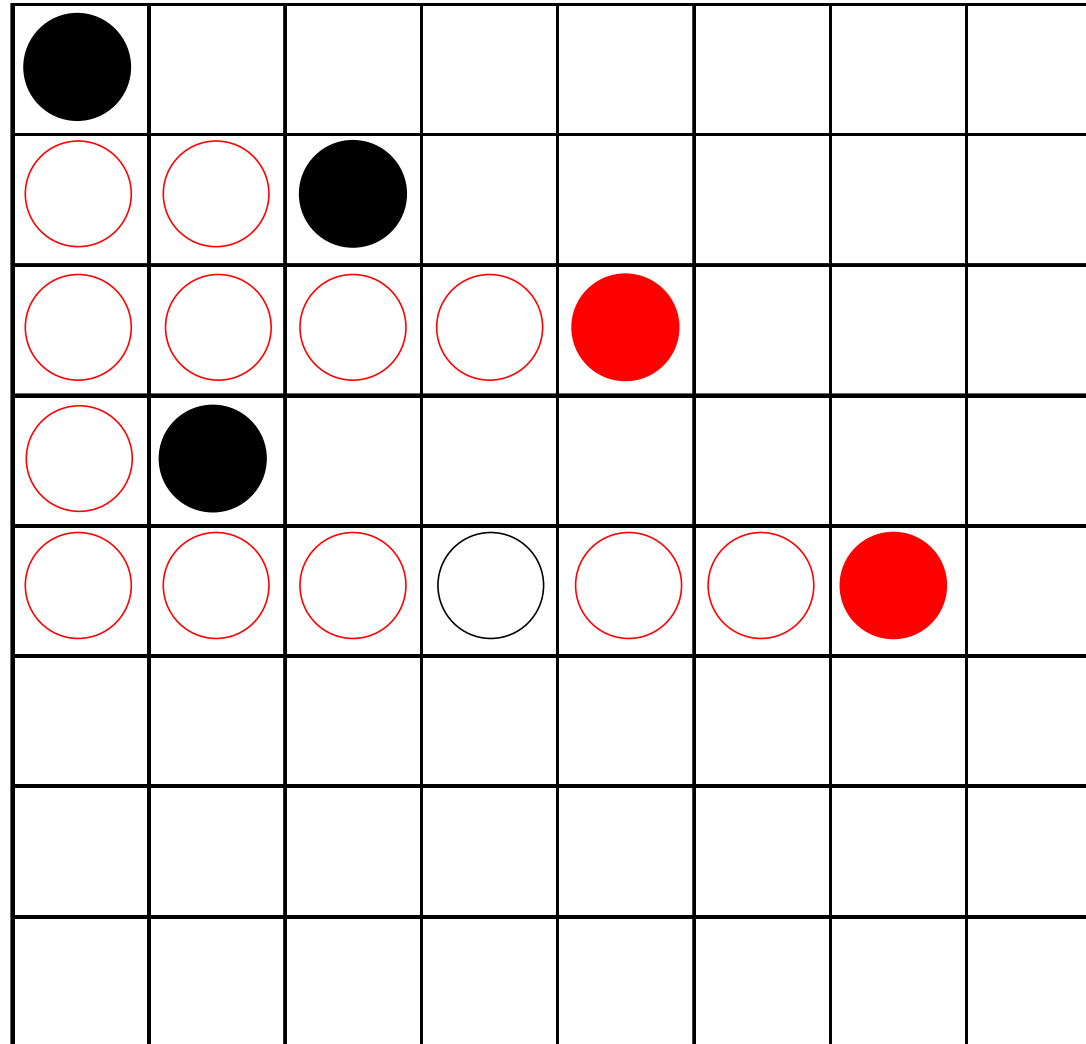
# Retrocesso



Testes  $46 + 2 = 48$

Retrocessos 1

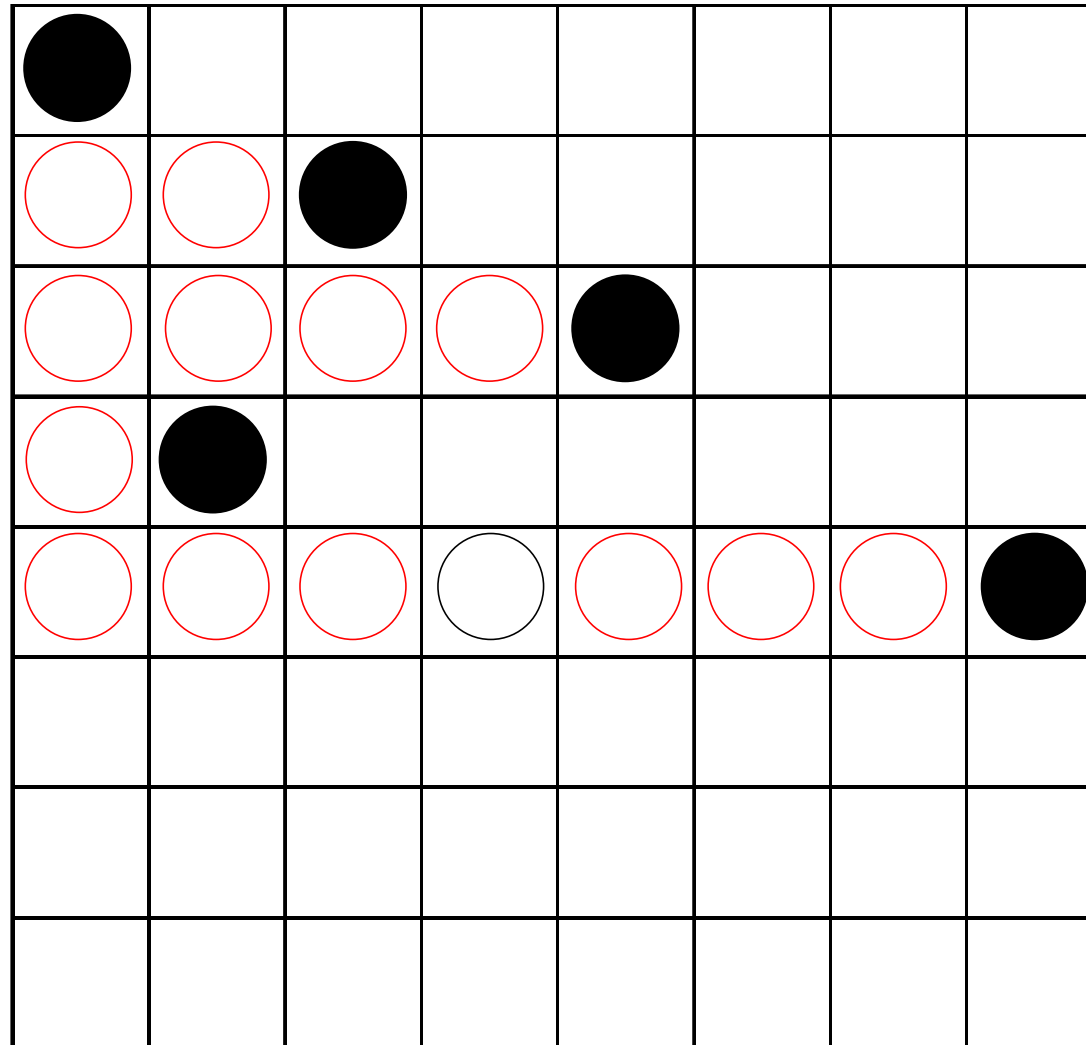
# Retrocesso



Testes  $48 + 3 = 51$

Retrocessos 1

# Retrocesso



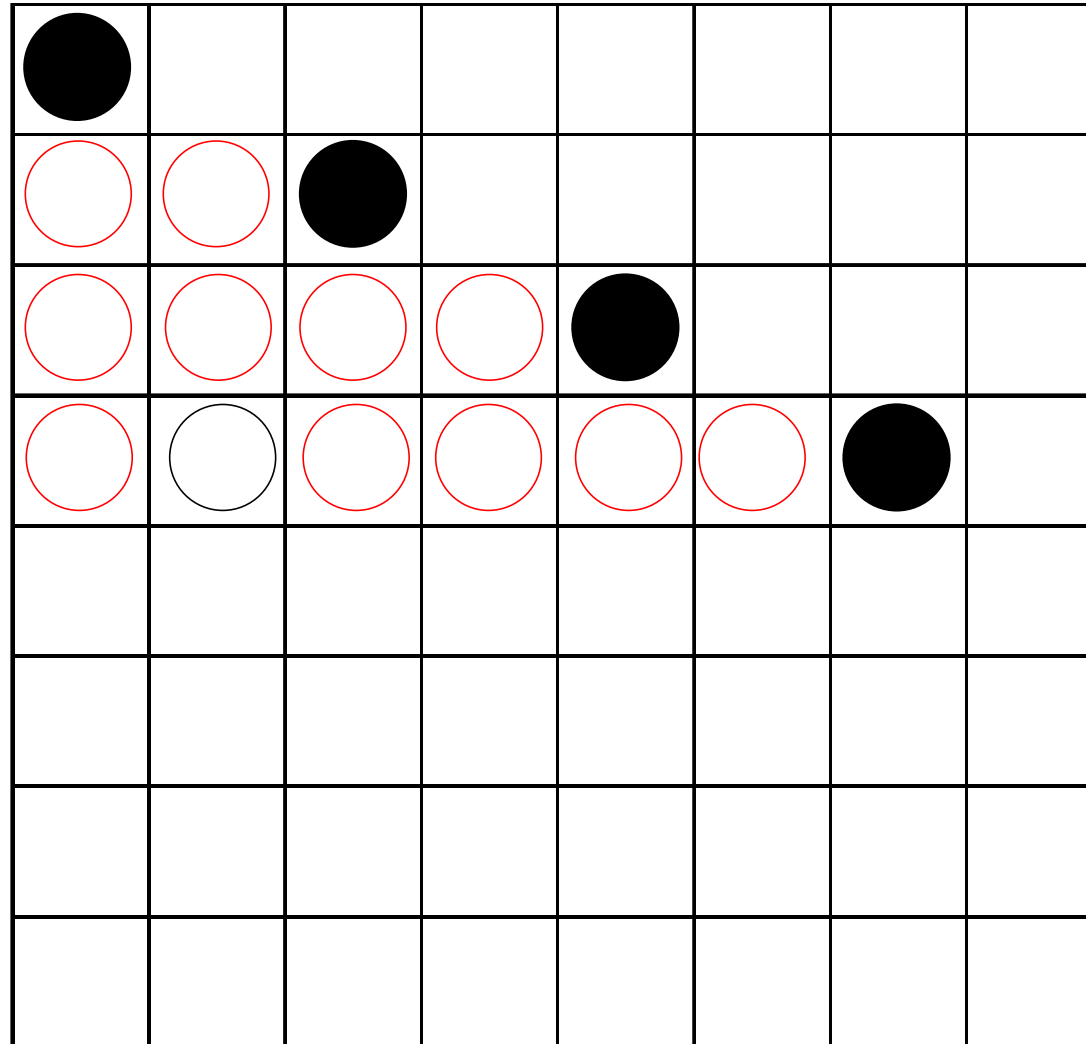
Testes  $51 + 4 = 55$

Retrocessos 1





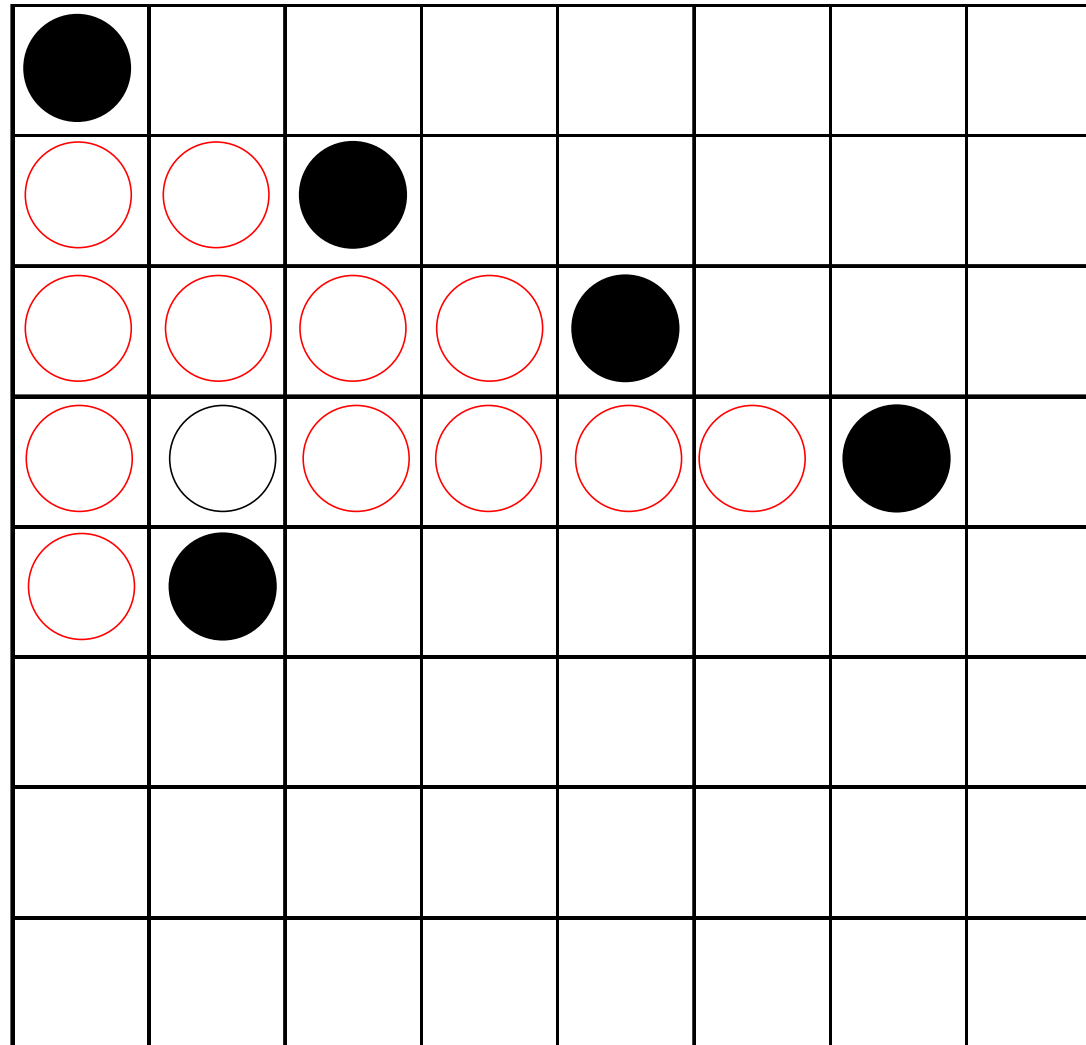
# Retrocesso



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

Retrocessos  $1+2 = 3$

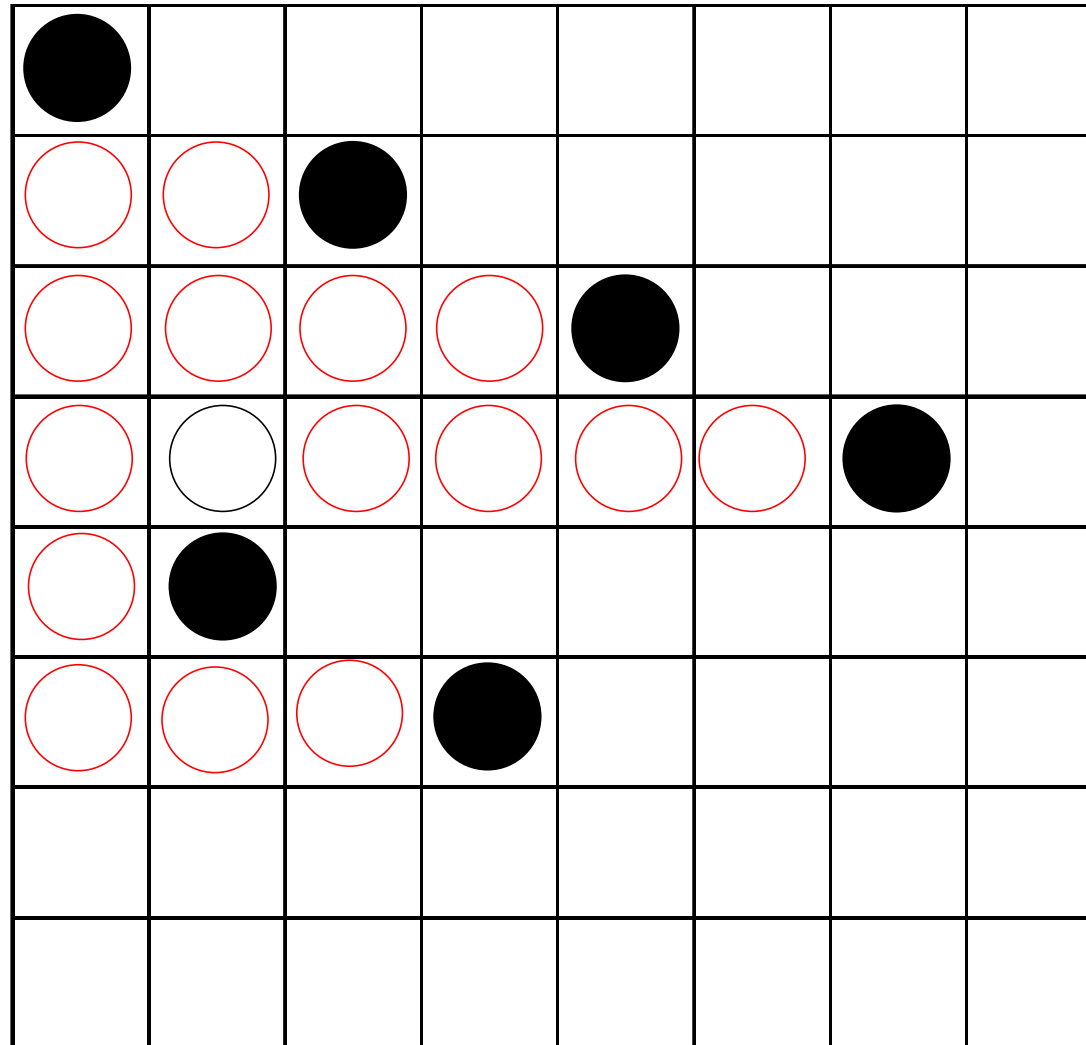
# Retrocesso



Testes  $85 + 1 + 4 = 90$

Retrocessos 3

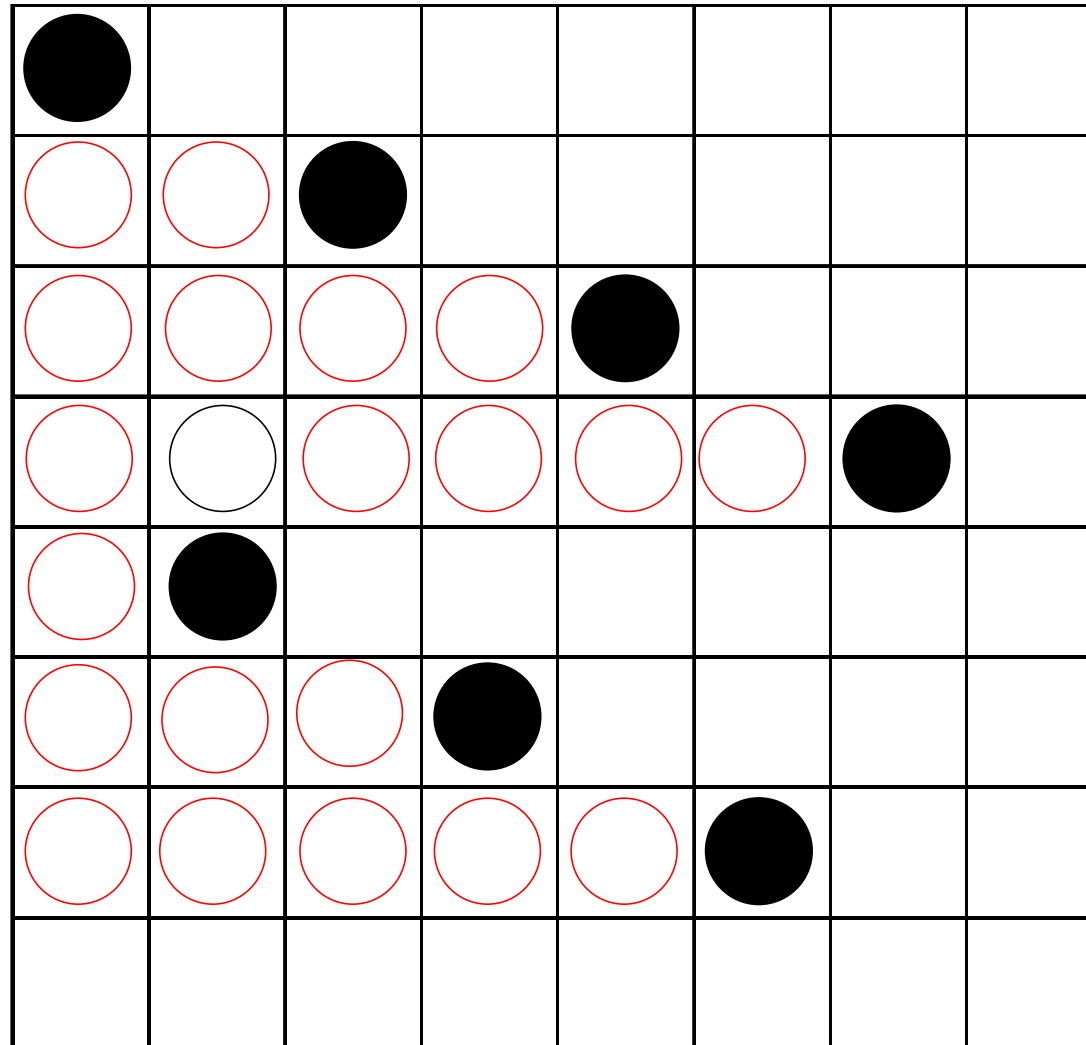
# Retrocesso



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

Retrocessos 3

# Retrocesso

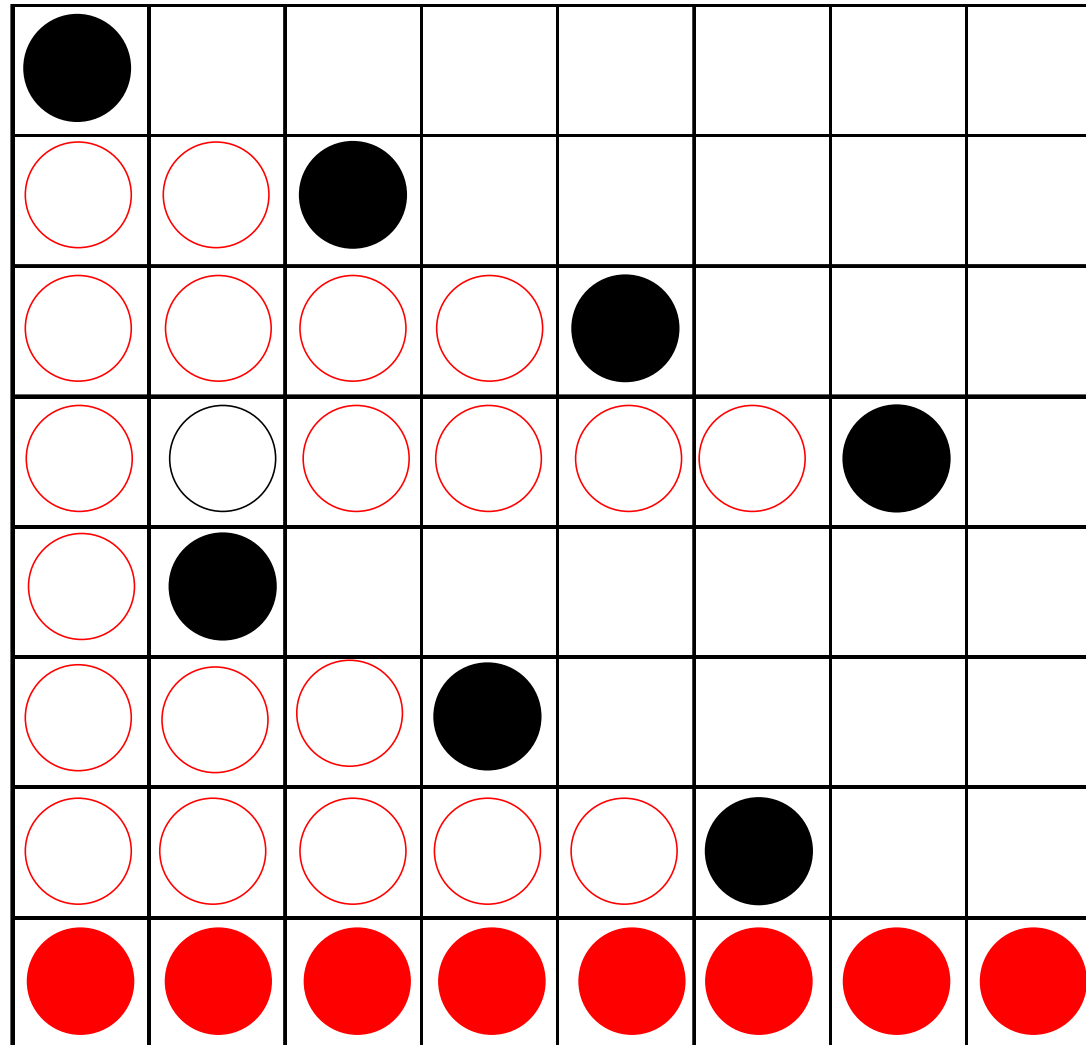


Testes  $10+1+5+2+4+3+6= 122$

Retrocessos 3

# Retrocesso

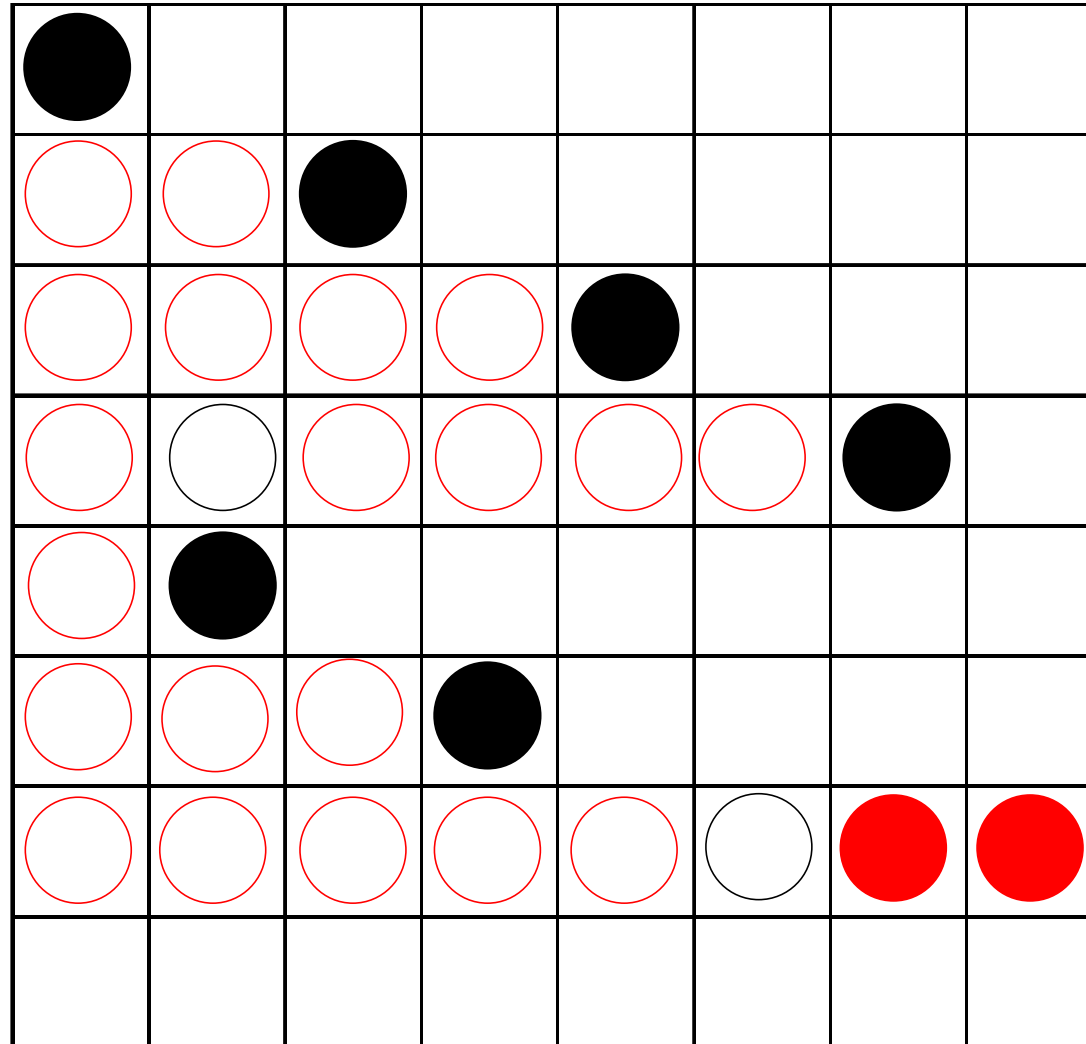
Falha  
8  
Retrocede  
7



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

# Retrocesso

Falha  
7  
Retrocede  
6

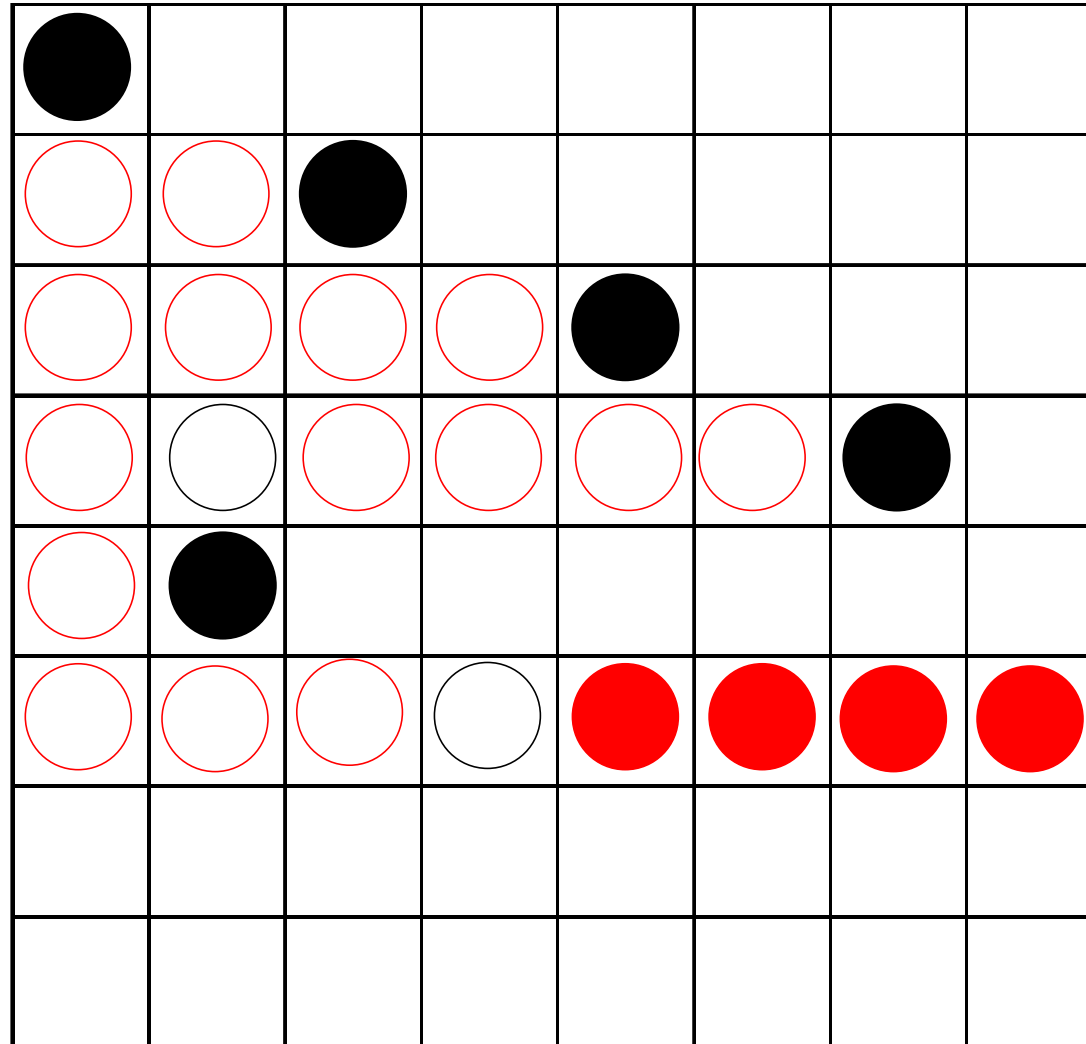


Testes  $150+1+2=153$

Retrocessos  $4+1=5$

# Retrocesso

Falha  
6  
Retrocede  
5

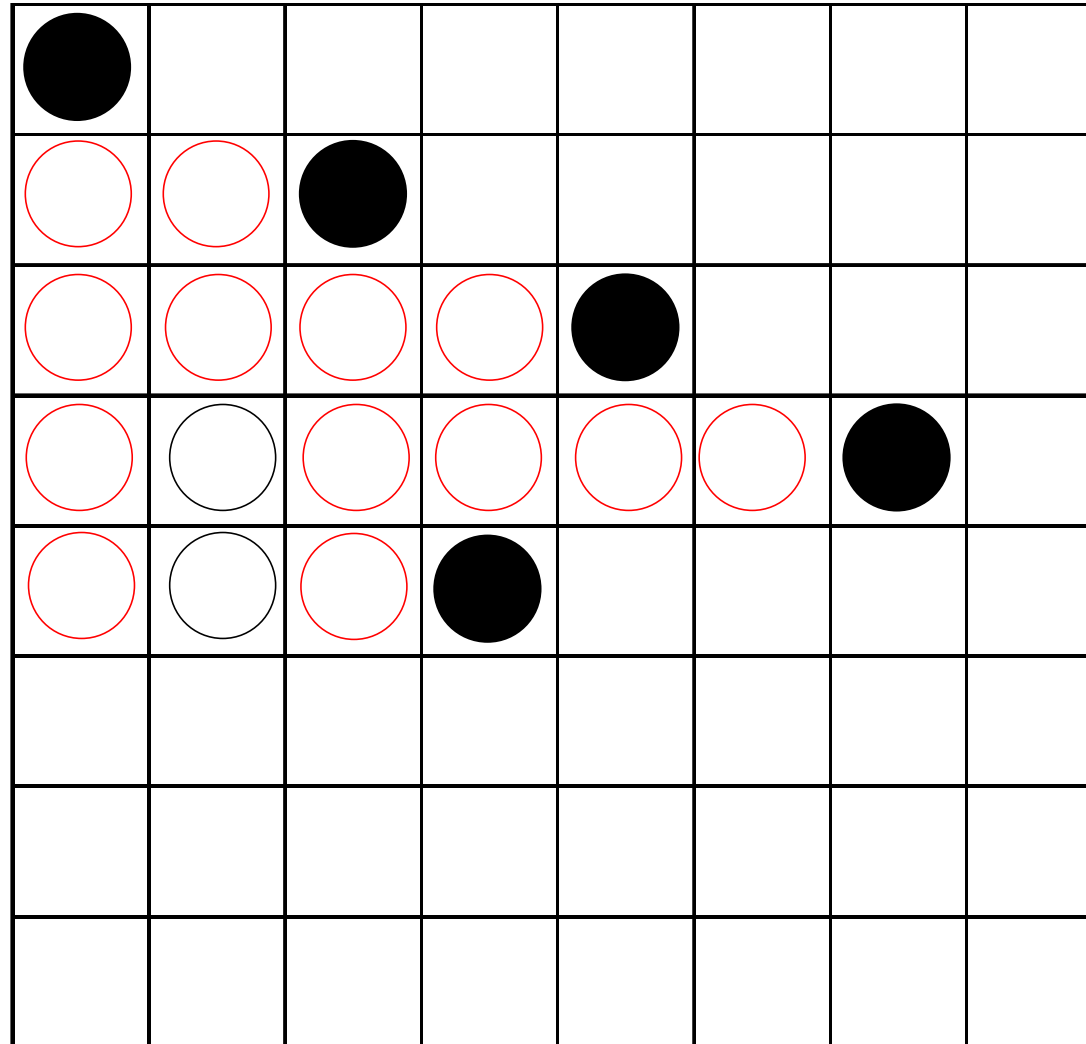


Testes  $153+3+1+2+3= 162$

Retrocessos  $5+1=6$



# Retrocesso

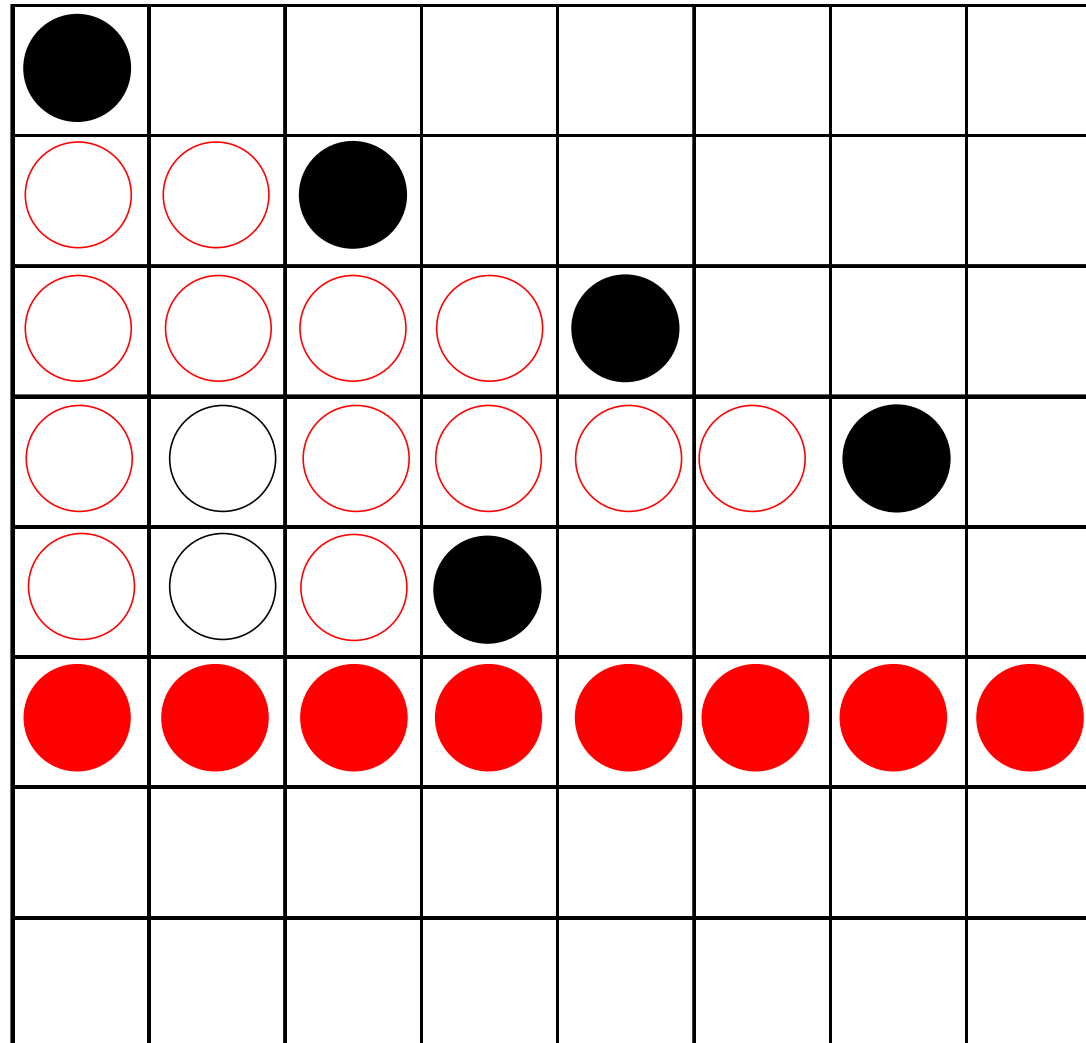


Testes  $162+2+4=168$

Retrocessos  $6+1=7$

# Retrocesso

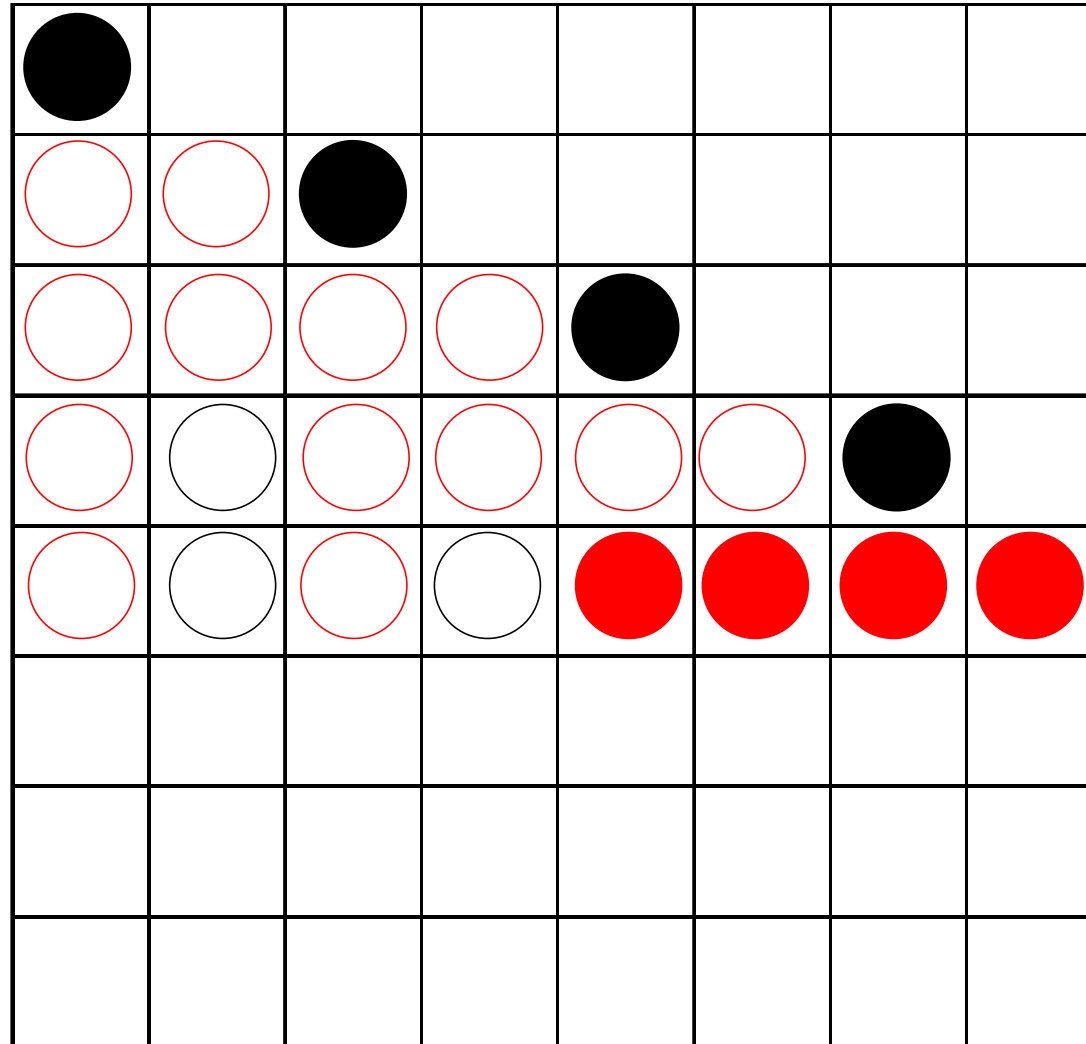
Falha  
6  
Retrocede  
5



Testes  $168+1+3+2+5+3+1+2+3= 188$  Retrocessos 7

# Retrocesso

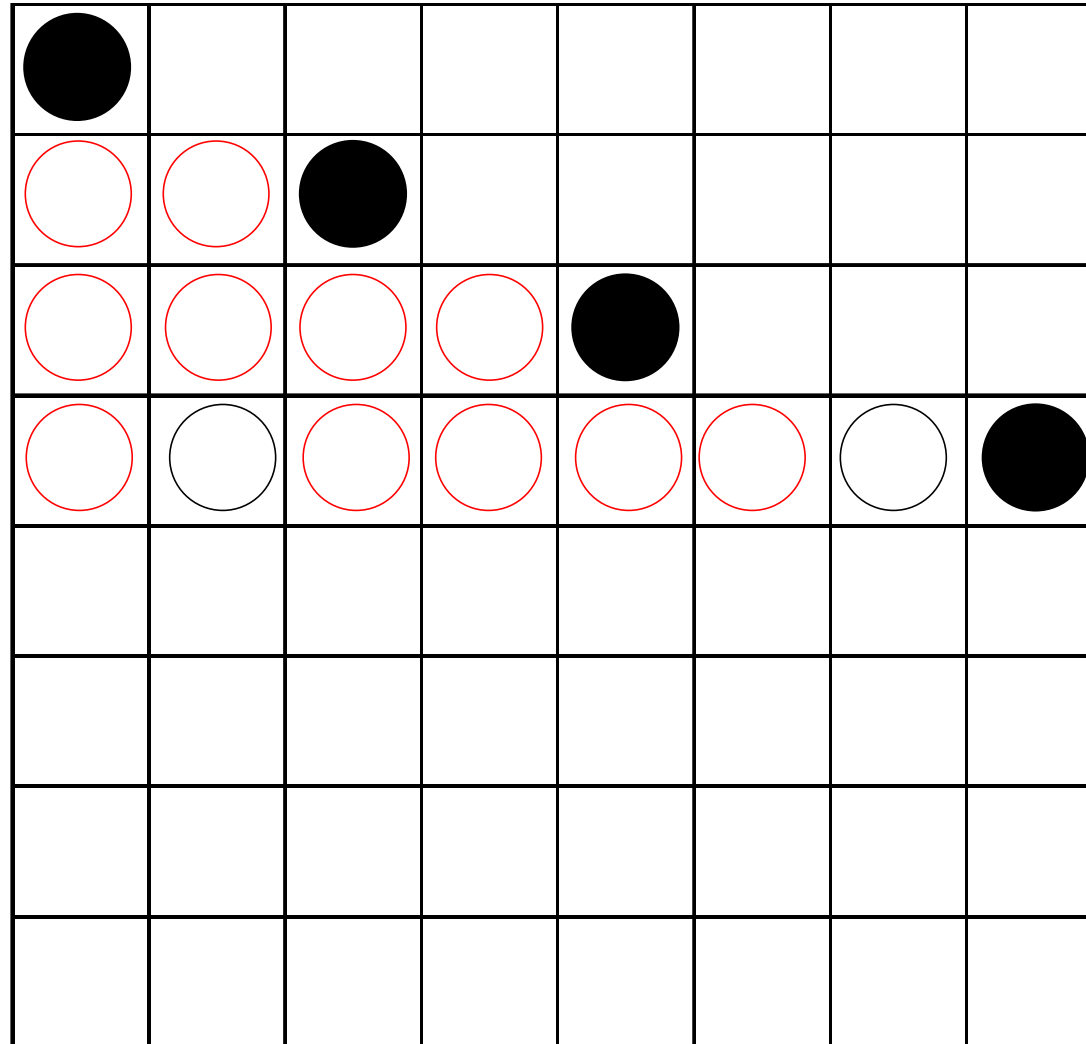
Falha  
5  
Retrocede  
4



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

Retrocessos  $7+1=8$

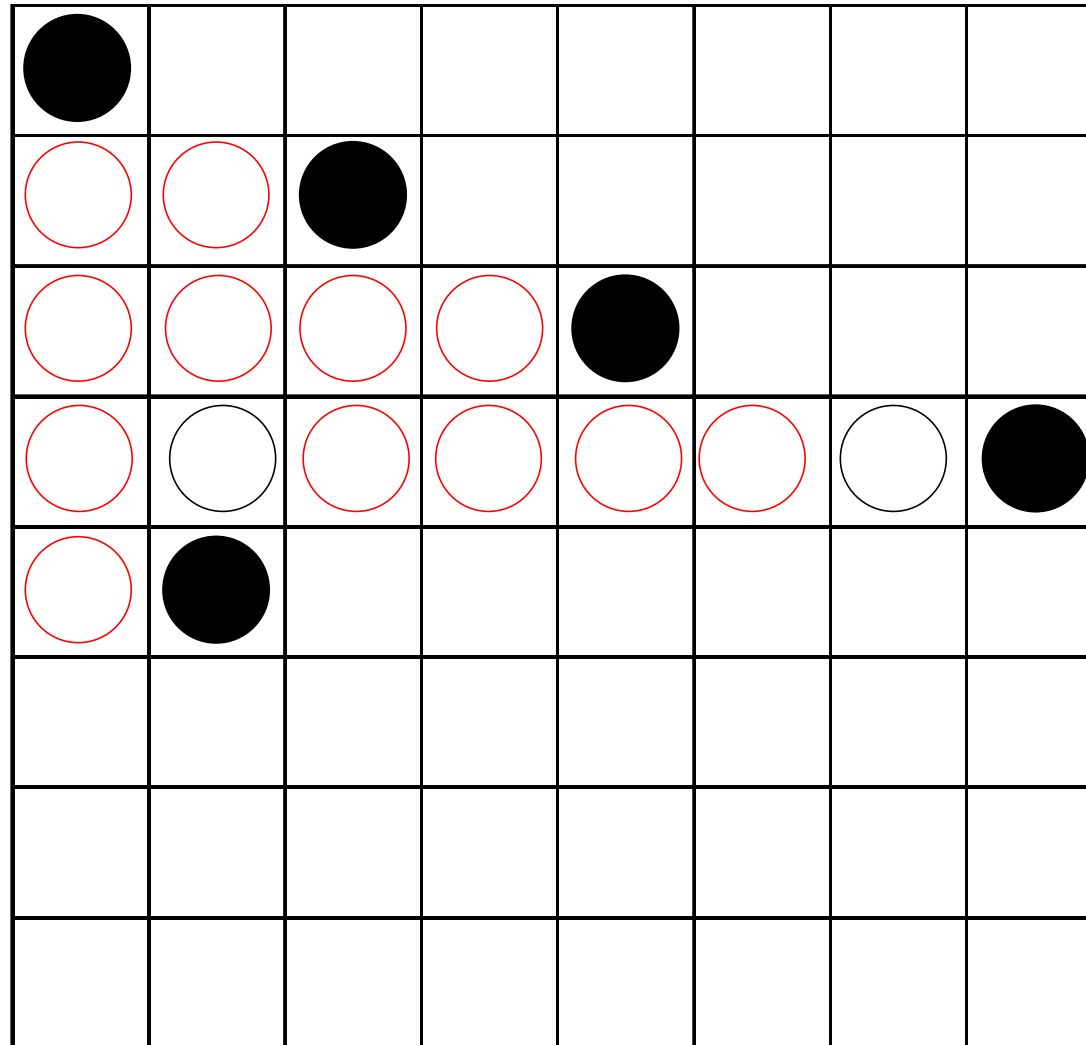
# Retrocesso



Testes  $198 + 3 = 201$

Retrocessos  $8+1=9$

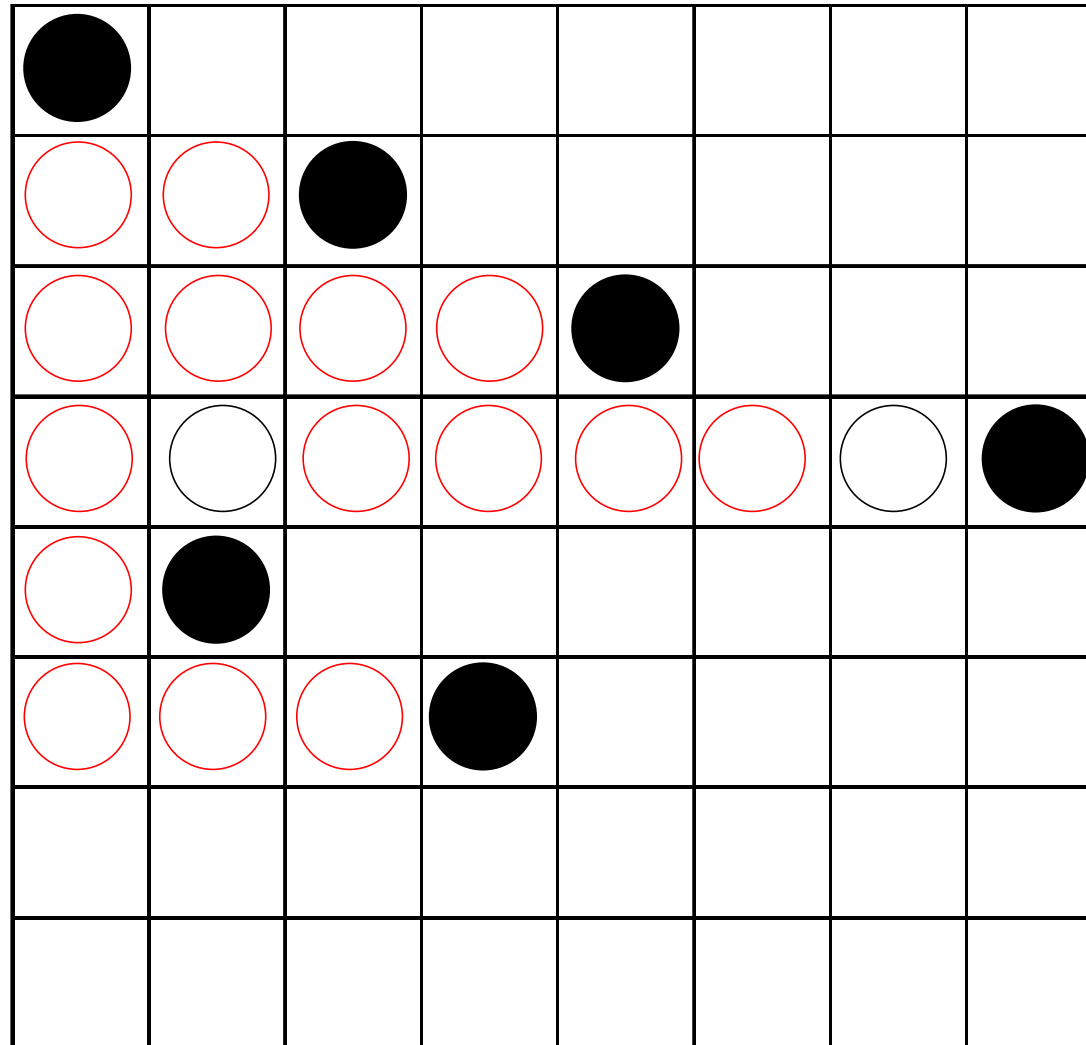
# Retrocesso



Testes  $201+1+4 = 206$

Retrocessos 9

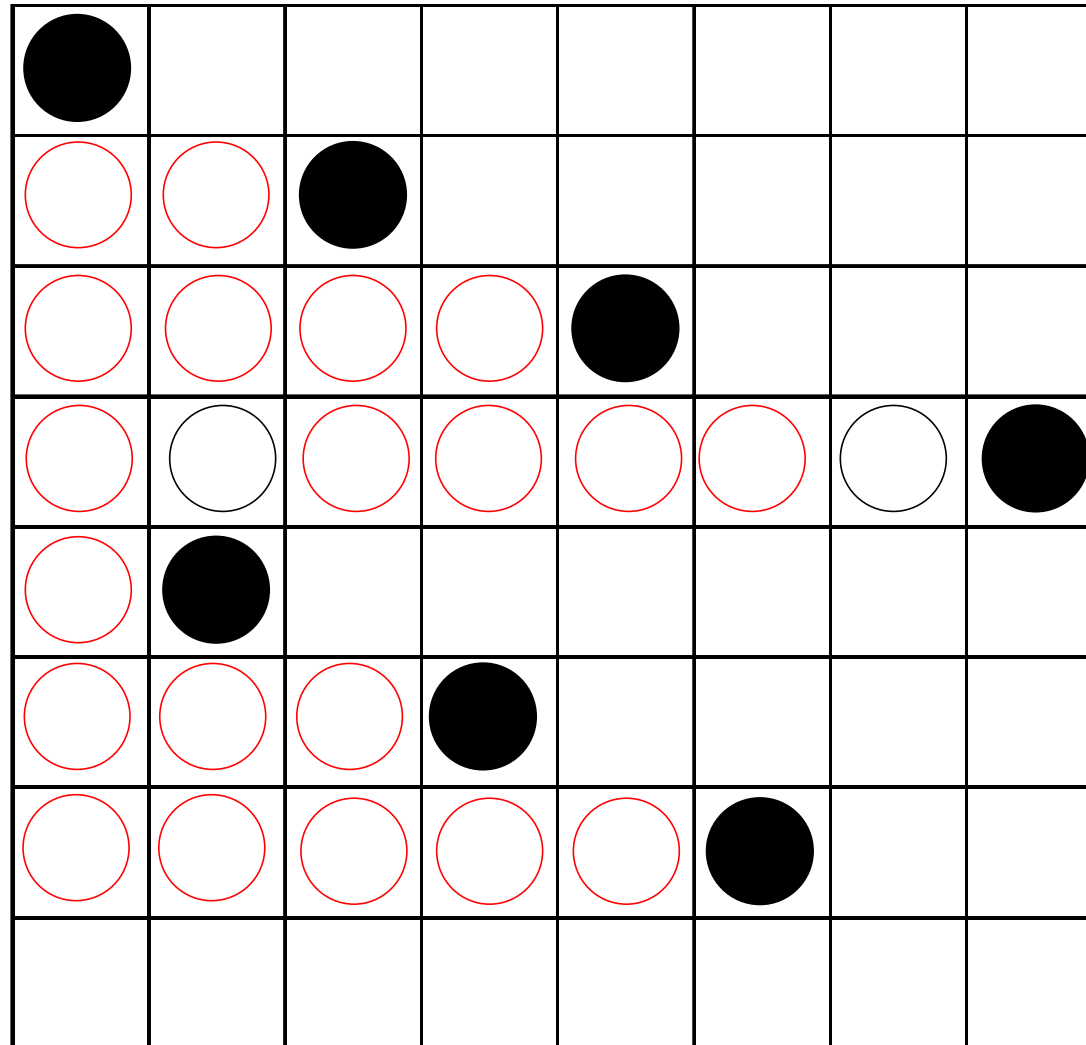
# Retrocesso



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

Retrocessos 9

# Retrocesso

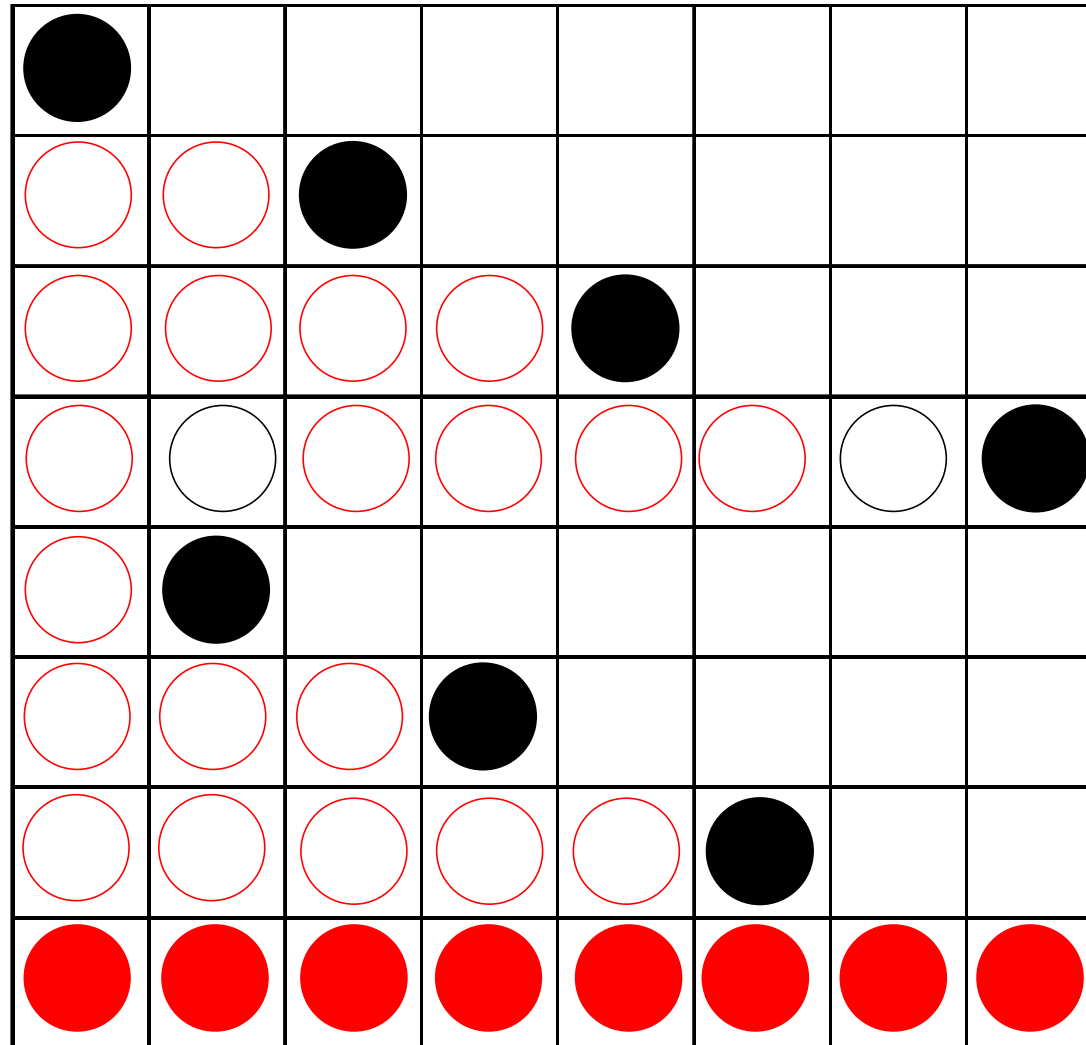


Testes  $217+1+5+2+5+3+6 = 239$

Retrocessos 9

# Retrocesso

Falha  
8  
Retrocede  
7



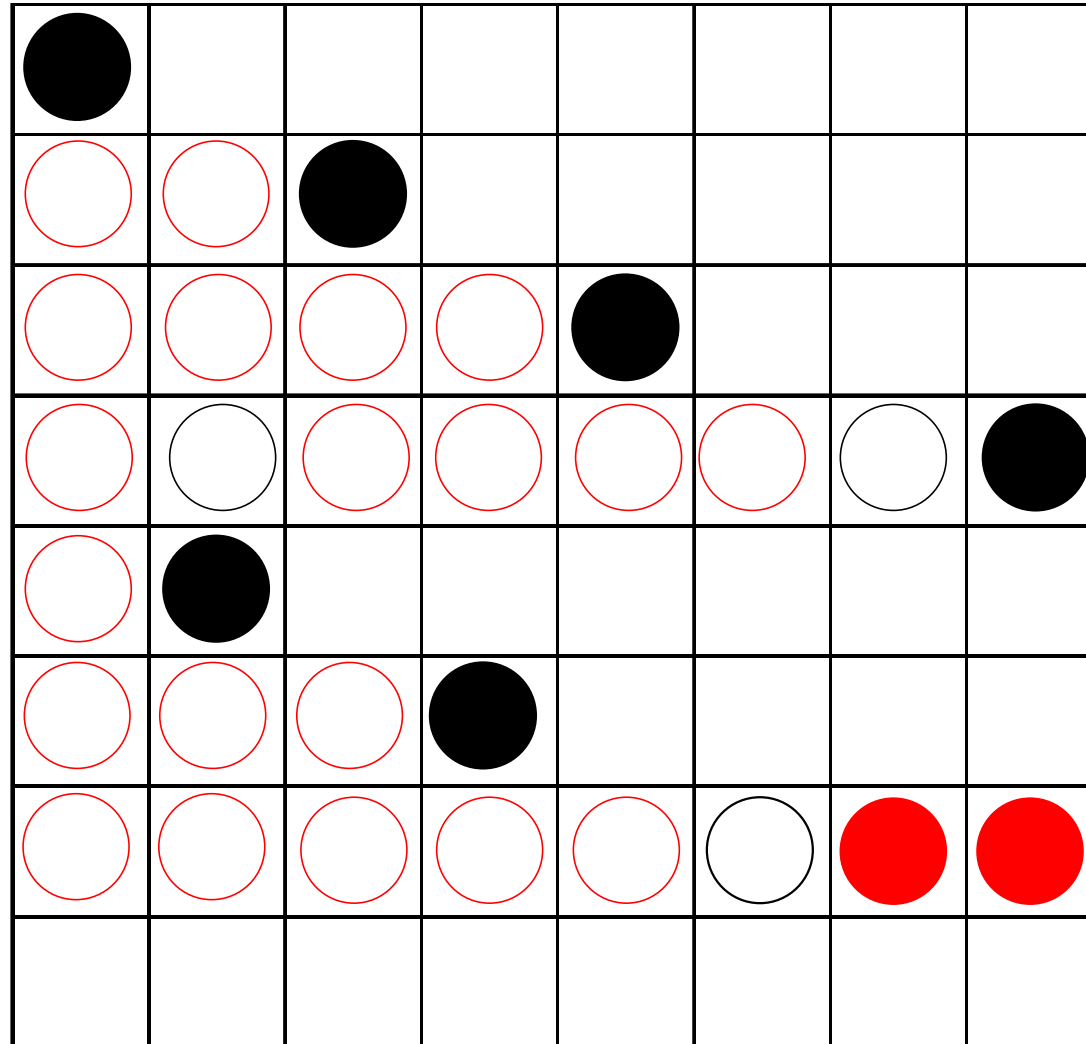
Testes  $239+1+5+2+4+3+6+7+7= 274$

Retrocessos  $9+1=10$



# Retrocesso

Falha  
7  
Retrocede  
6

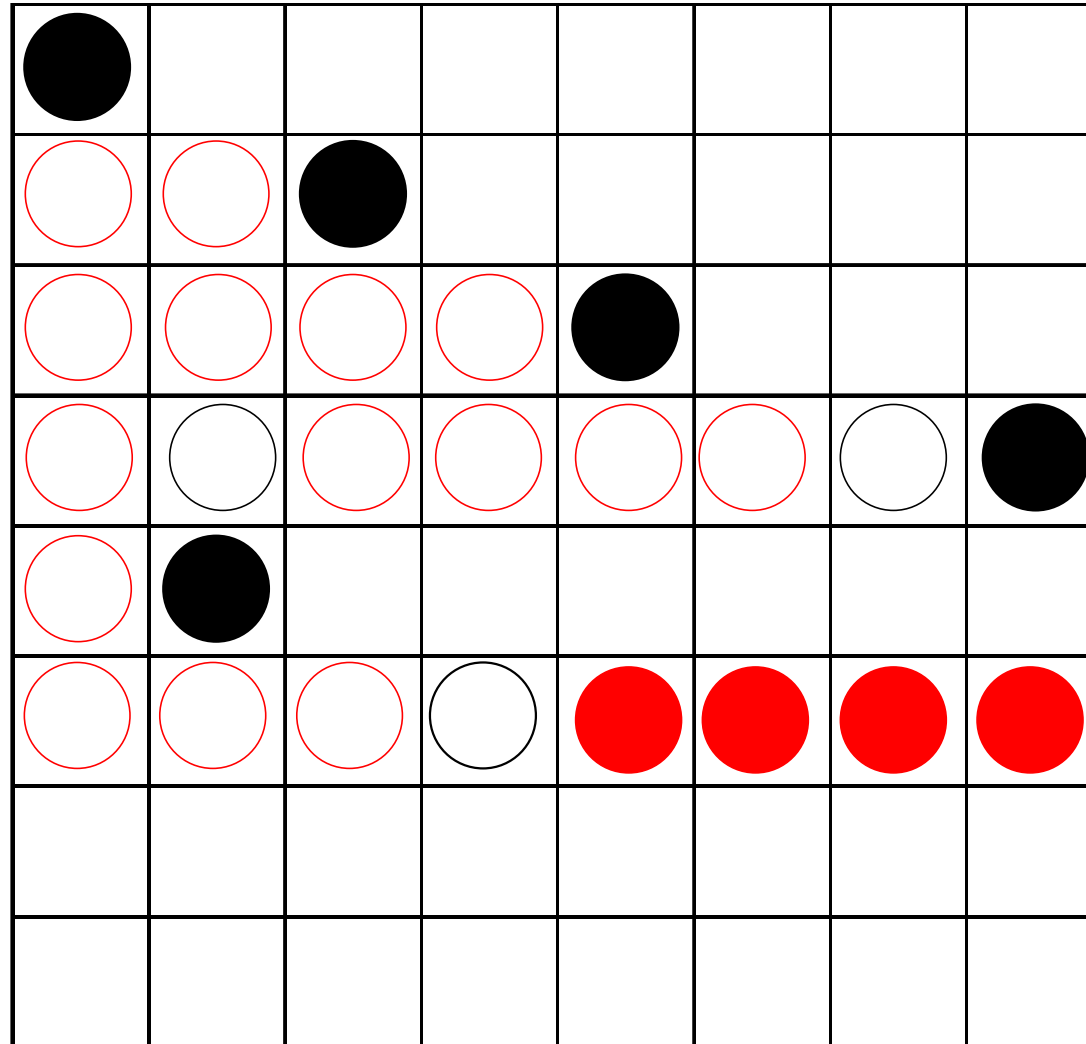


Testes  $274+1+2= 277$

Retrocessos  $10+1=11$

# Retrocesso

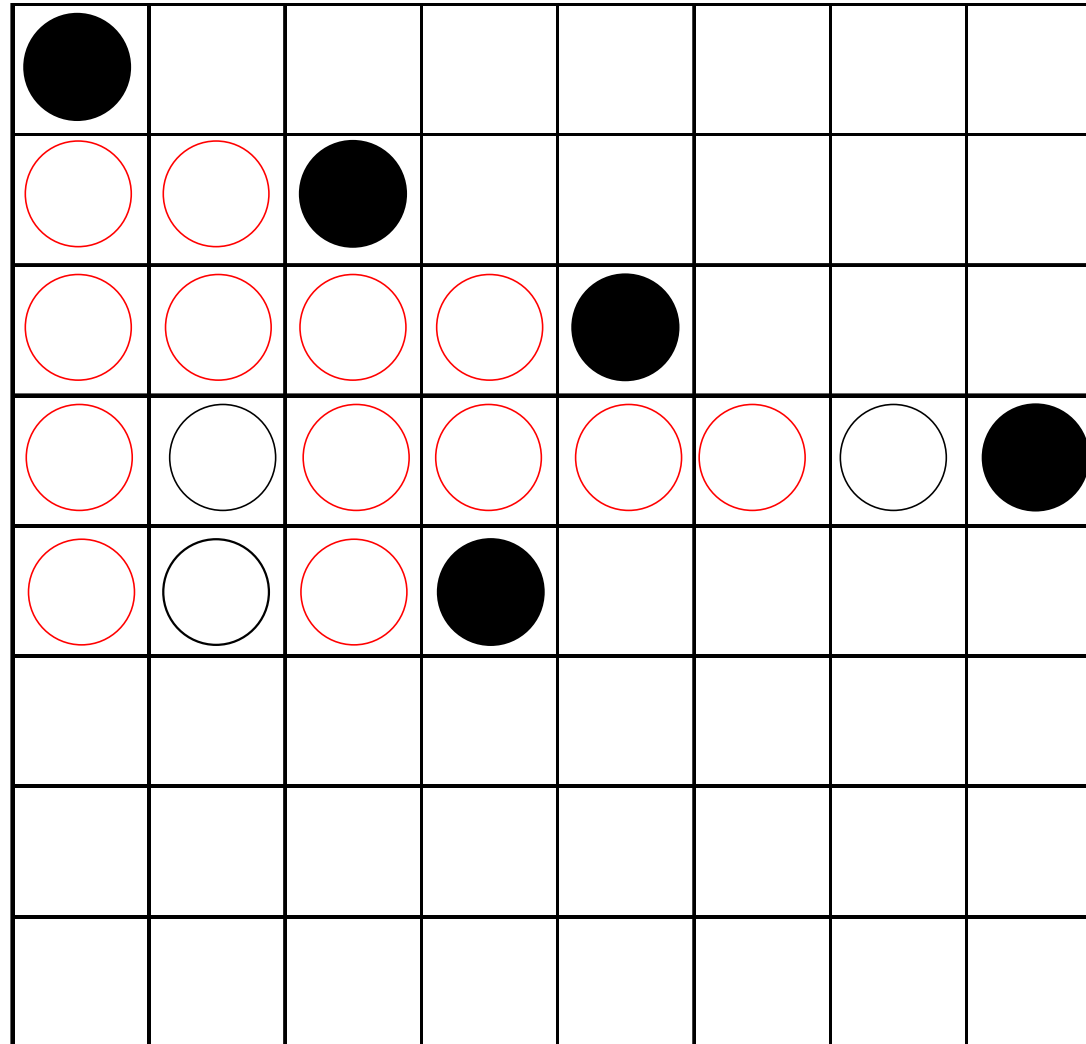
Falha  
6  
Retrocede  
5



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

Retrocessos  $11+1=12$

# Retrocesso

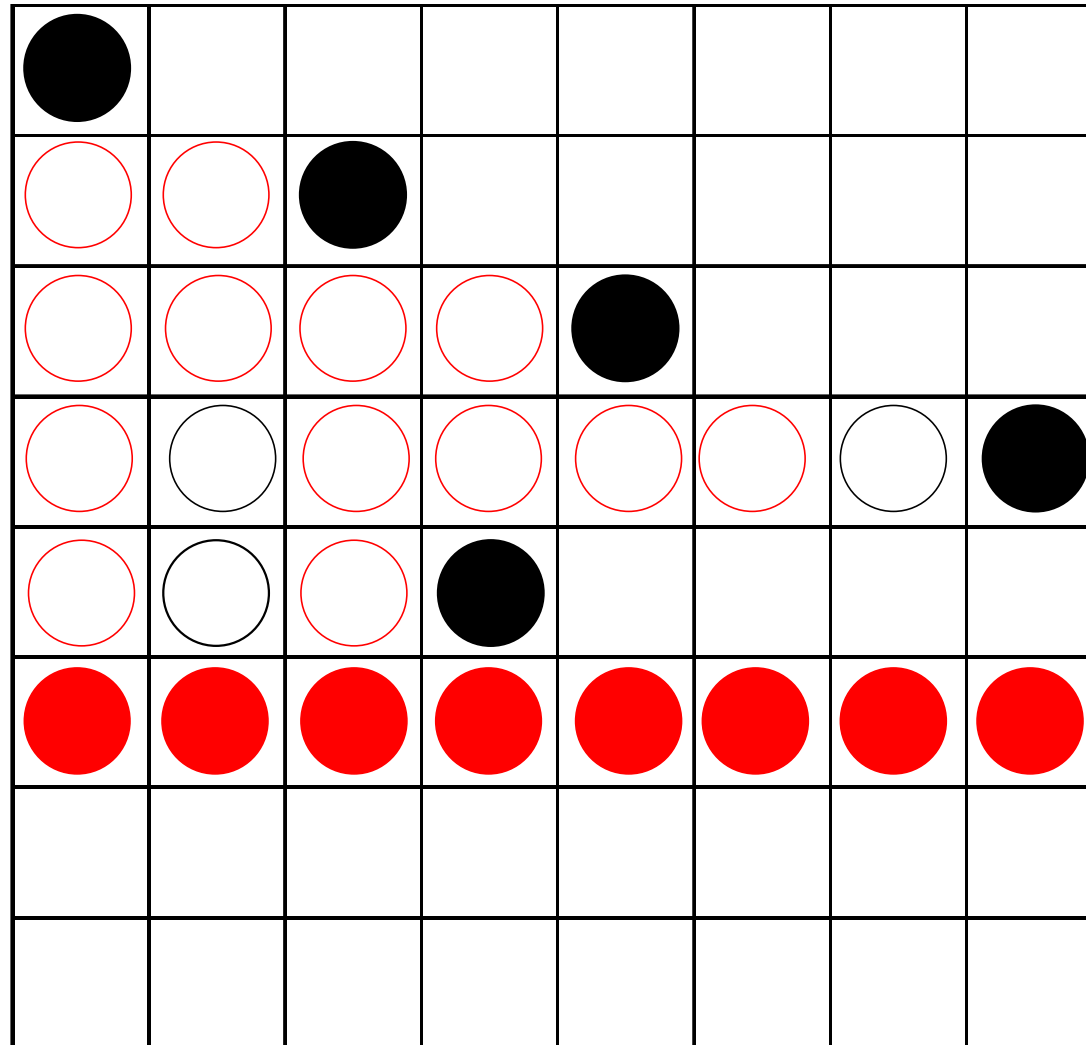


Testes  $286+2+4= 292$

Retrocessos 12

# Retrocesso

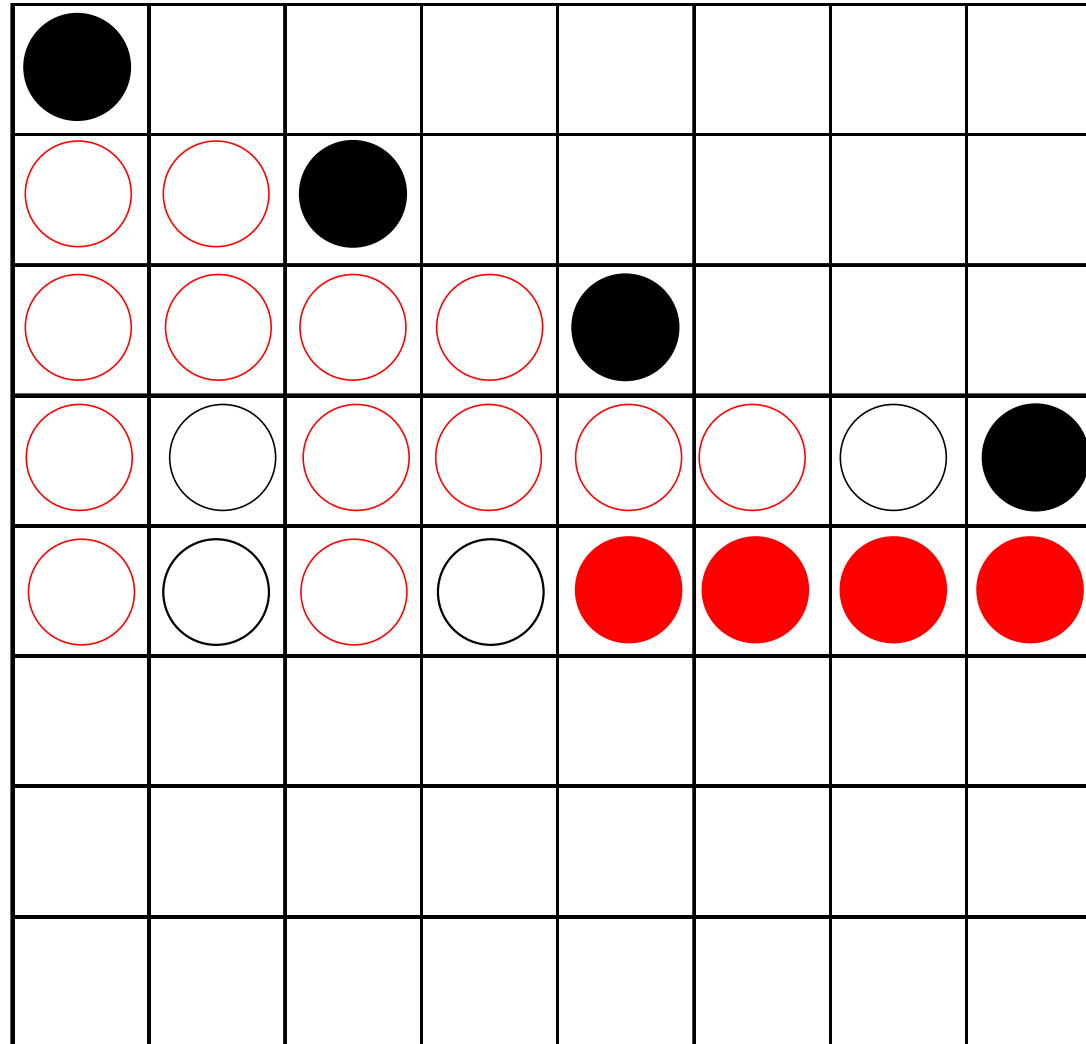
Falha  
6  
Retrocede  
5



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

# Retrocesso

Falha  
5  
Retrocede  
4 e 3



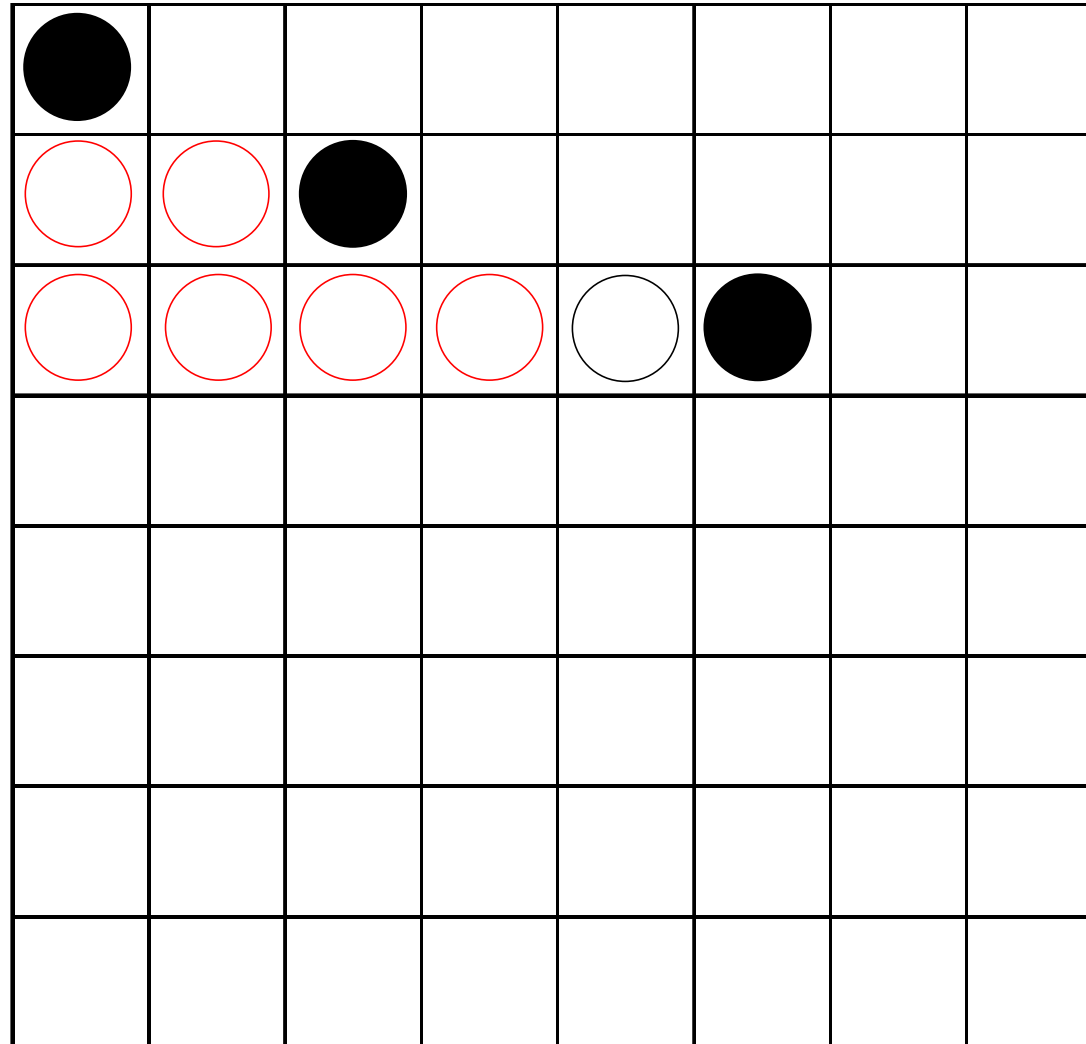
Testes  $3+2+1+0+0+0+0+0=6$

Retrocessos  $3+2=5$

# Retrocesso

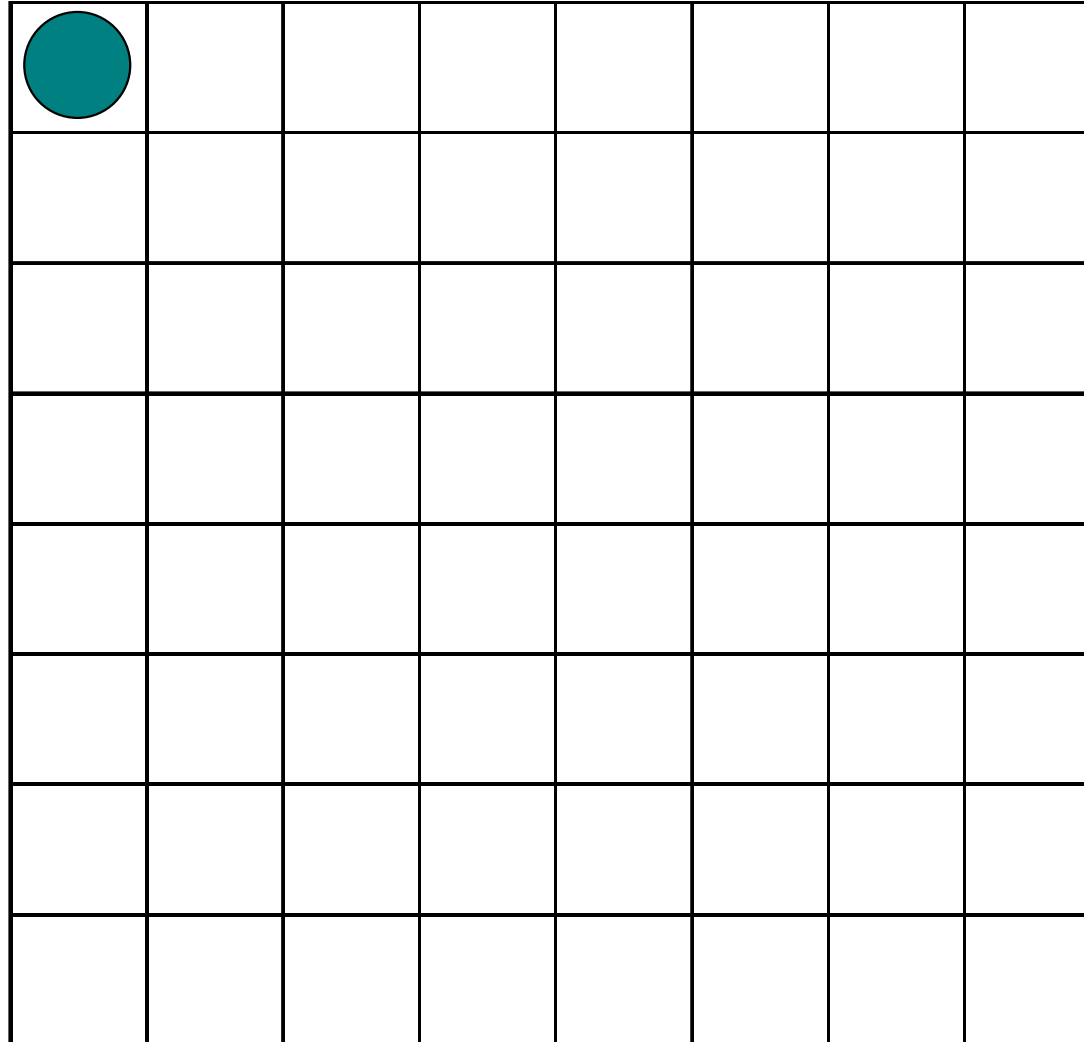
 $X_1=1$  $X_2=3$  $X_3=5$ 

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

Testes 125

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	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






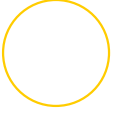
							
1	1						
1	2	1	2				
1	6	2	1	2	3	8	
1		2	6	1	2	3	
1	3	2		3	1	2	3
1		2	6	3	8	1	2
1	6	2	2	3	6		1

Testes 131

Retrocessos 0








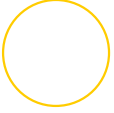
# Propagação 2

							
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 131

Retrocessos 0

# Propagação 2






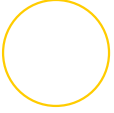
							
1	1						
1	2	1	2				
1	6	2	1	2	3	8	
1		2	6	1	2	3	4
1	3	2		3	1	2	3
1		2	6	3	8	1	2
1	6	2	2	3	6		1

Testes  $131+2+2=135$ 

Retrocessos 0

# Propagação 2







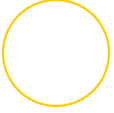
$X_5$  só pode  
tomar o valor 2

							
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

Testes 135

Retrocessos 0







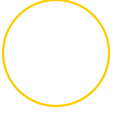
# Propagação 2

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

Testes 135

Retrocessos 0







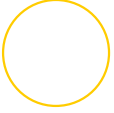
# Propagação 2

							
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







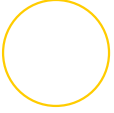
							
1	1						
1	2	1	2				
1	6	2	1	2	3	8	
1		2	6	1	2	3	4
1	3	2		3	1	2	3
1	5	2	6	3	8	1	2
1	6	2	2	3	6		1

Testes 136

Retrocessos 0

# Propagação 2

Falha  
7  
Retrocede  
3 !



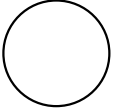

							
1	1						
1	2	1	2				
1	6	2	1	2	3	8	
1		2	6	1	2	3	4
1	3	2		3	1	2	3
1	5	2	6	3	8	1	2
1	6	2	2	3	6		1

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



# Programação por Restrições

- Foco na propagação de restrições (“*forward checking*”) 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:
  - IBM ILOG CP Optimizer, ...
  - Google OR Tools, ...
  - OptaPlanner, ...

Programação em Lógica com Restrições

# 5. 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]

# 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

- Uma restrição que envolva um conjunto de variáveis limita as etiquetas compostas para essas variáveis
- Uma restrição  $C_{ijk}$  envolvendo as variáveis  $X_i$ ,  $X_j$  e  $X_k$  definirá um subconjunto do produto cartesiano dos domínios das variáveis envolvidas
  - $C_{ijk} \subseteq \text{dom}(X_i) \times \text{dom}(X_j) \times \text{dom}(X_k)$
- 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  $X_1$  e  $X_3$  pode ser definida:

- Em Extensão:

$$C_{13} = \{ \{X_1-1, X_3-2\}, \{X_1-1, X_3-4\}, \{X_1-2, X_3-1\}, \{X_1-2, X_3-3\}, \\ \{X_1-3, X_3-2\}, \{X_1-3, X_3-4\}, \{X_1-4, X_3-1\}, \{X_1-4, X_3-3\} \}$$

- Ou de forma mais simples:

$$C_{13} = \{ \langle 1,2 \rangle, \langle 1,4 \rangle, \langle 2,1 \rangle, \langle 2,3 \rangle, \\ \langle 3,2 \rangle, \langle 3,4 \rangle, \langle 4,1 \rangle, \langle 4,3 \rangle \}$$

- Ou implicitamente através da fórmula respetiva:

$$C_{13} = (X_1 \neq X_3) \wedge (X_1 \neq X_3+2) \wedge (X_1 \neq X_3-2)$$

# Aridade de Restrições

- 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  $i$ -ésimo *label* composto que define  $C$

# Restrições Binárias

- Exemplo:
  - Dadas as variáveis  $X_1$ ,  $X_2$  e  $X_3$ , com domínio 1 a 3, a seguinte restrição ternária  $C$  impõe valores diferentes para as três variáveis e é composta por 6 *labels* compostos:
$$C(X_1, X_2, X_3) = \{ \langle 1,2,3 \rangle, \langle 1,3,2 \rangle, \langle 2,1,3 \rangle, \langle 2,3,1 \rangle, \langle 3,1,2 \rangle, \langle 3,2,1 \rangle \}$$
  - Cada um dos *labels* pode ter um valor associado entre 1 e 6:
    - 1:  $\langle 1,2,3 \rangle$ , 2:  $\langle 1,3,2 \rangle$ , 3:  $\langle 2,1,3 \rangle$ , ..., 6:  $\langle 3,2,1 \rangle$
  - As seguintes restrições binárias  $B_1$  a  $B_3$  são equivalentes à restrição ternária inicial  $C$ :
    - $B_1(Z, X_1) = \{ \langle 1,1 \rangle, \langle 2,1 \rangle, \langle 3,2 \rangle, \langle 4,2 \rangle, \langle 5,3 \rangle, \langle 6,3 \rangle \}$
    - $B_2(Z, X_2) = \{ \langle 1,2 \rangle, \langle 2,3 \rangle, \langle 3,1 \rangle, \langle 4,3 \rangle, \langle 5,1 \rangle, \langle 6,2 \rangle \}$
    - $B_3(Z, X_3) = \{ \langle 1,3 \rangle, \langle 2,2 \rangle, \langle 3,3 \rangle, \langle 4,1 \rangle, \langle 5,2 \rangle, \langle 6,1 \rangle \}$



# Satisfação de Restrições

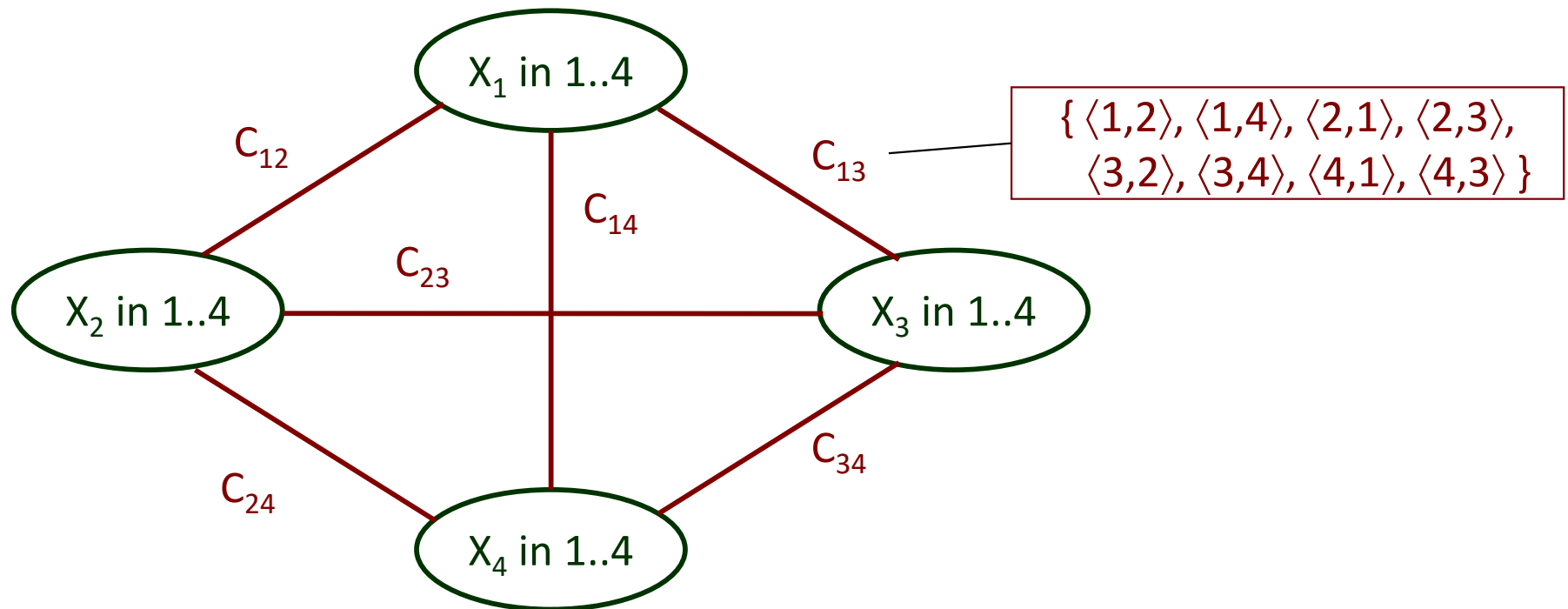
- Um *label* composto **satisfaz uma restrição** se as suas **variáveis são as mesmas** da restrição e se o *label* composto é **membro da restrição**
- Superconjunto de variáveis:
  - Um *label* composto **satisfaz uma restrição** se o seu conjunto de variáveis **contém as variáveis** da restrição e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

# Rede de Restrições

- **Rede (Grafo) de Restrições**
  - Contém em cada nó uma variável
  - 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 seguinte rede de restrições:



# 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

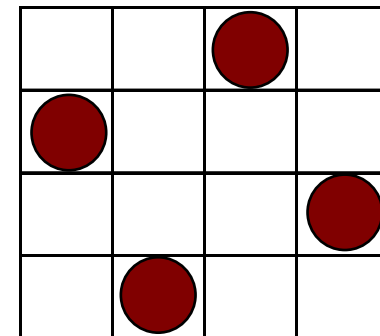
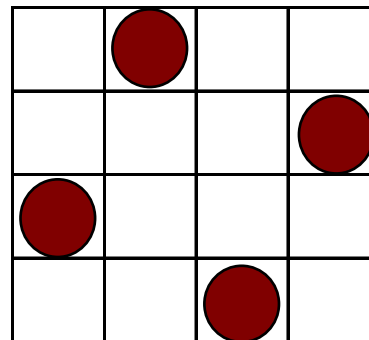
- A **dificuldade de resolução** de um PSR está normalmente relacionada com a densidade da sua rede de restrições:
  - Quanto maior a densidade mais difícil é o problema pois haverá mais possibilidades de invalidar possíveis soluções
- Convém distinguir:
  - Dificuldade do problema
  - Dificuldade de resolver o problema
- Por vezes, um problema difícil pode ser trivialmente 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:
    - $C_{13} = \{ \langle 1,2 \rangle, \langle 1,4 \rangle, \langle 2,1 \rangle, \langle 2,3 \rangle, \langle 3,2 \rangle, \langle 3,4 \rangle, \langle 4,1 \rangle, \langle 4,3 \rangle \}$
    - $Tightness = 8 / (4 * 4) = 1/2$

# *Tightness* de um Problema

- A noção de *tightness* pode ser generalizada para o problema completo
- 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 relacionada com a densidade da sua rede de restrições e com a sua *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 exibem habitualmente 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...

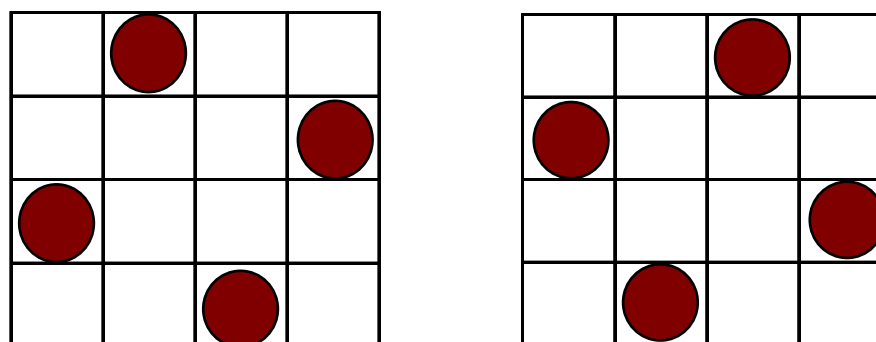


# Redundância

- 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

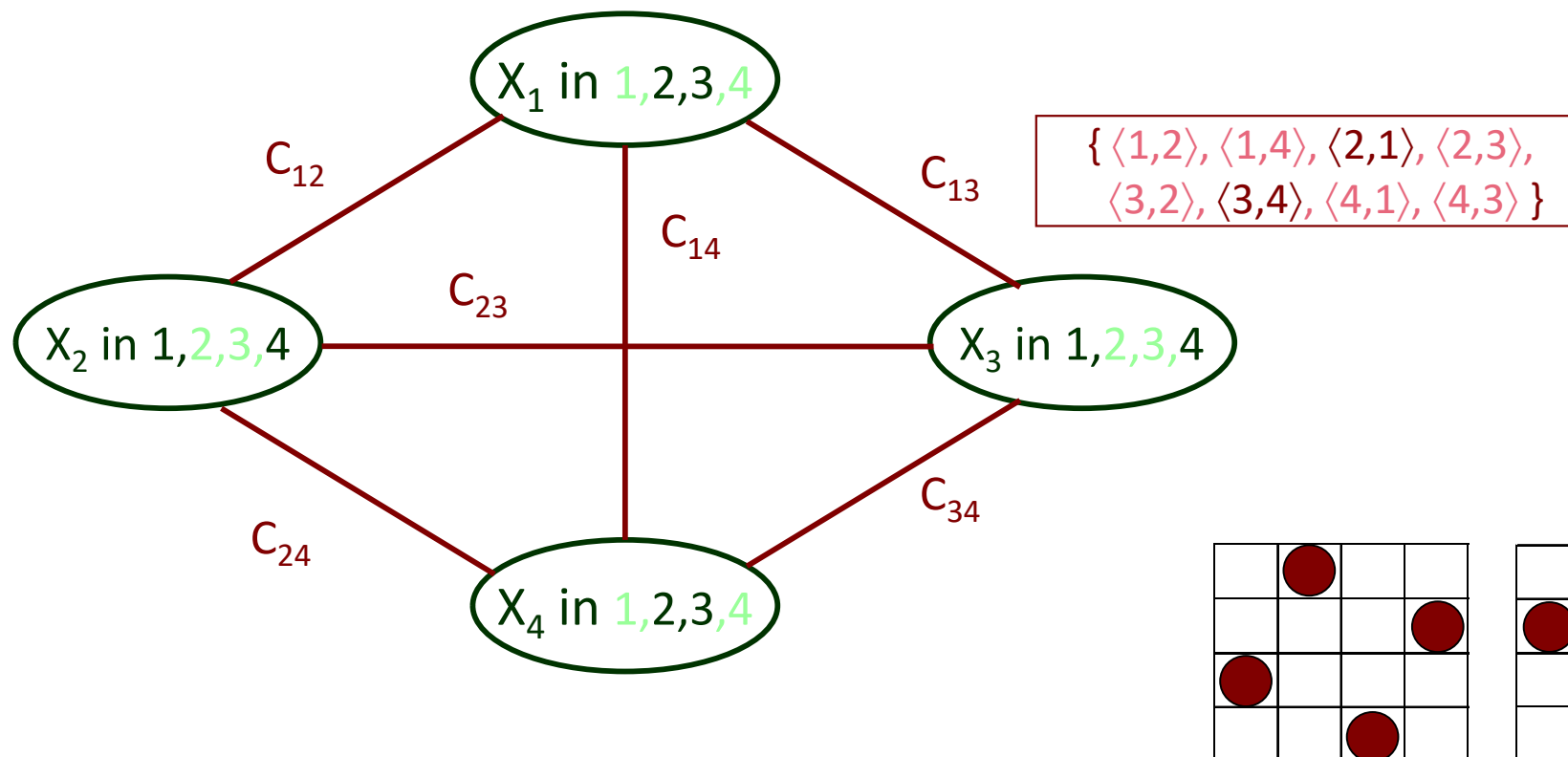
- Exemplo:
  - O problema das 4-Rainhas só admite duas soluções,  $\langle 2,4,1,3 \rangle$  e  $\langle 3,1,4,2 \rangle$



- 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** sse 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  $C$

# 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:

$$\#S = \#D_1 \times \#D_2 \times \dots \times \#D_n$$

- 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  $n$ :

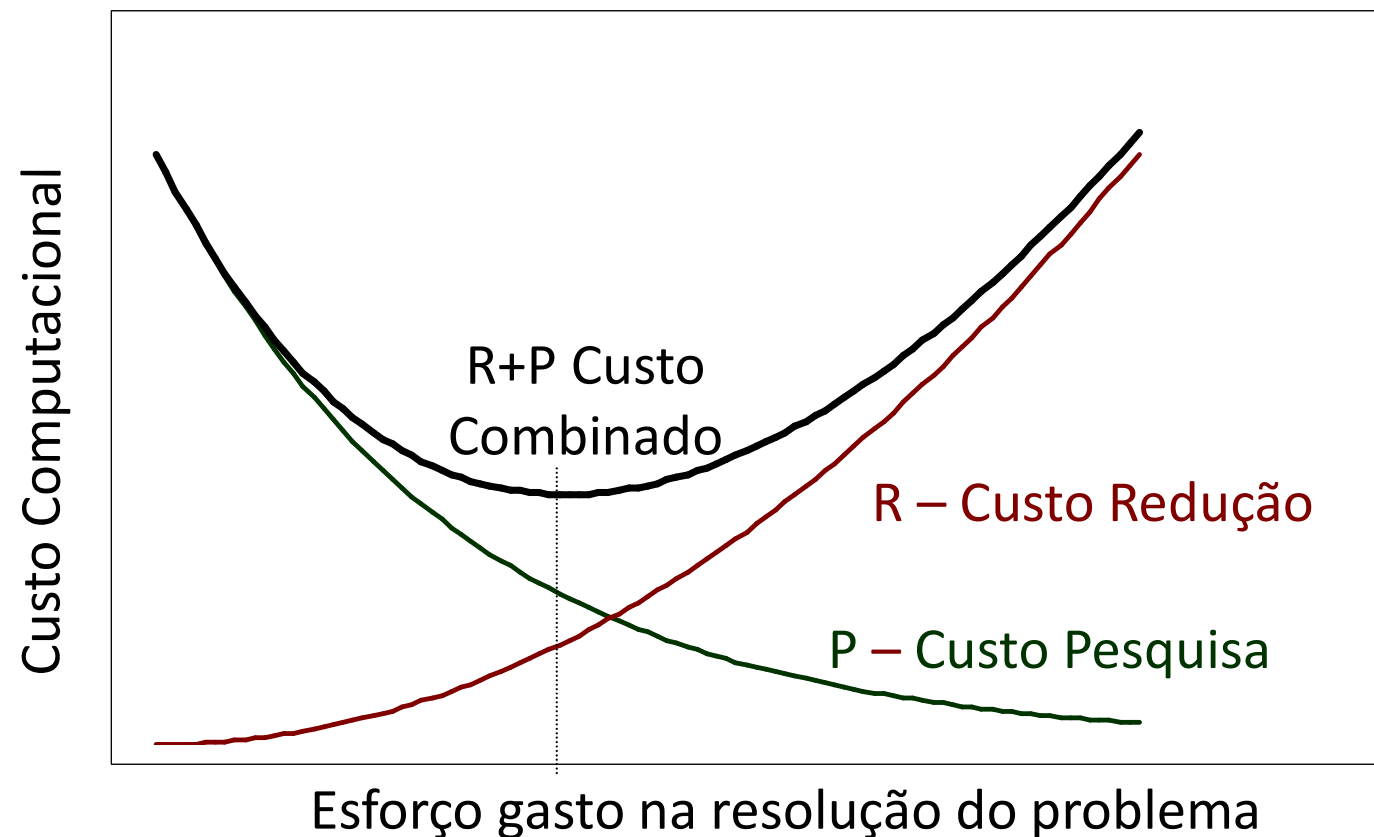
		n									
S/S'		10	20	30	40	50	60	70	80	90	100
7	6	4.6716	21.824	101.95	476.29	2225	10395	48560	226852	1E+06	5E+06
6	5	6.1917	38.338	237.38	1469.8	9100.4	56348	348889	2E+06	1E+07	8E+07
5	4	9.3132	86.736	807.79	7523.2	70065	652530	6E+06	6E+07	5E+08	5E+09
4	3	17.758	315.34	5599.7	99437	2E+06	3E+07	6E+08	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 muito 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 PLR a especificação das restrições 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

## 6. 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

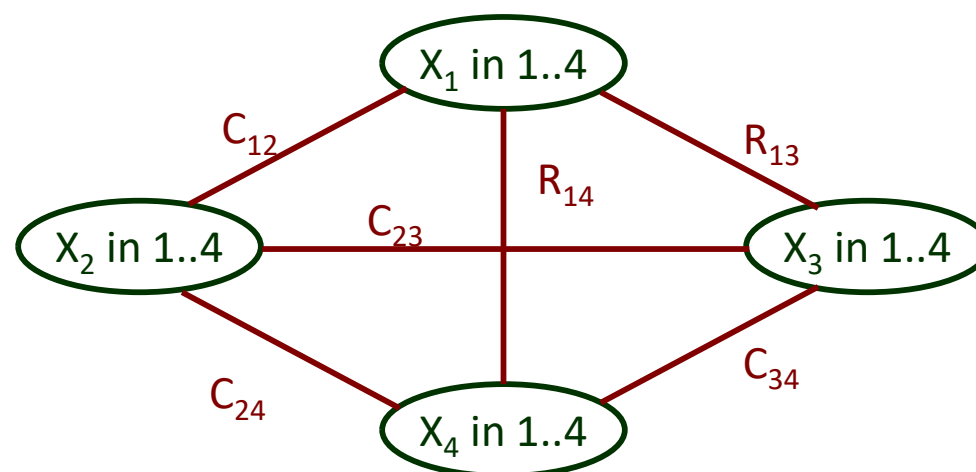
- Critérios de Consistência permitem estabelecer valores redundantes nos domínios das variáveis de forma indireta, i.e. não necessitando de conhecimento à-priori das soluções
- Procedimentos que mantenham esses critérios durante a propagação vão eliminar valores redundantes e, como tal, reduzir o espaço 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

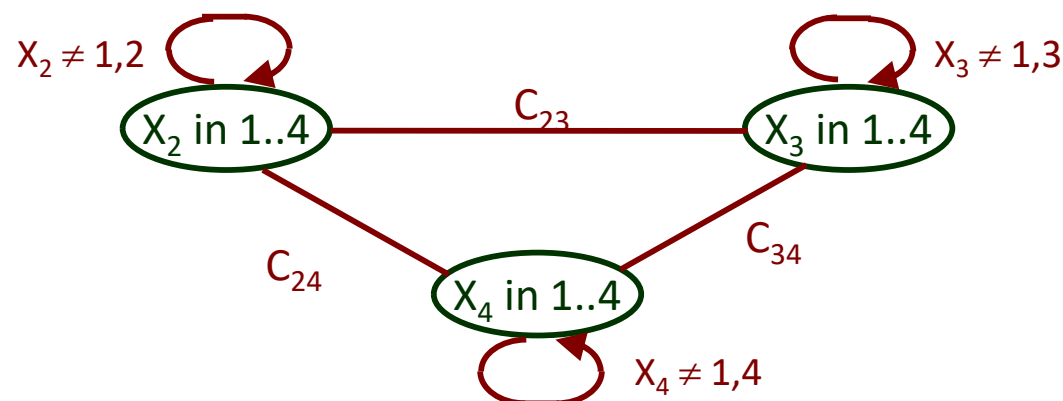
- Um PSR é **consistente nos nós** (“*node-consistent*”) se nenhum valor no domínio das suas variáveis viola as restrições unárias
- Este critério parece óbvio e inútil: quem iria especificar um domínio que violasse as restrições unárias?!
- No entanto o critério é útil (indispensável) durante o processo 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 espacial:  $O(nd)$ ; temporal:  $O(nd)$
- 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

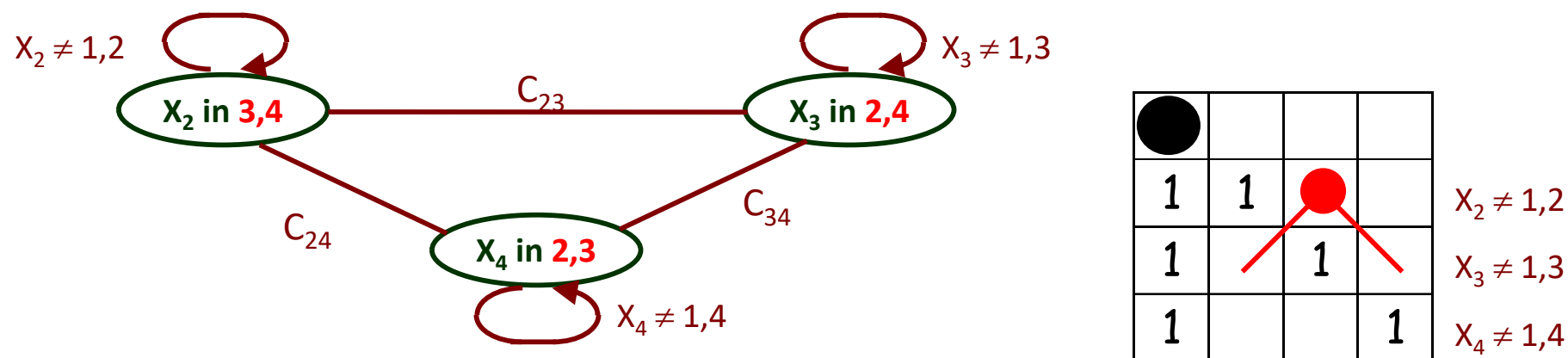


# Consistência dos Arcos

- Critério mais complexo de consistência
- Um PSR é **consistente nos arcos** (“*arc-consistent*”) se
  - É consistente nos nós
  - 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:



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

# Consistência dos Arcos

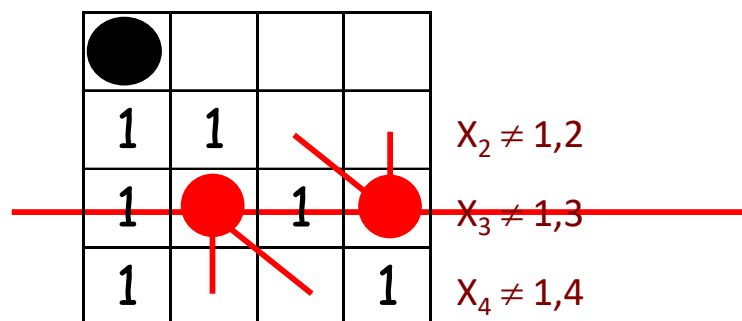
- No exemplo, nenhum dos valores de  $X_3$  tem suporte nas variáveis  $X_2$  e  $X_4$ :

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

- O *label*  $X_3 - 2$  não tem suporte na variável  $X_4$ , pois nenhum dos *labels* compostos  $\{X_3 - 2, X_4 - 2\}$  e  $\{X_3 - 2, X_4 - 3\}$  satisfazem a restrição  $C_{34}$
- O *label*  $X_3 - 4$  não tem suporte na variável  $X_2$ , pois nenhum dos *labels* compostos  $\{X_2 - 3, X_3 - 4\}$  e  $\{X_2 - 4, X_3 - 4\}$  satisfazem a restrição  $C_{23}$

# 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};    % 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  $C_{ij}$  são considerados dois arcos dirigidos  $a_{ij}$  e  $a_{ji}$

# Consistência dos Arcos

- O predicado *revise\_dom*(*a<sub>ij</sub>*,*V*,*D*,*C*) sucede sse reduzir o domínio de *X<sub>i</sub>*:

```
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<sub>ij</sub>, 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};  
  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 uma menor complexidade no pior caso e 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, AC-2000, AC-8, AC-\*, ...
  - Ver Jean-Charles Régin, “[AC-\\*: A Configurable, Generic and Adaptive Arc Consistency Algorithm](#)”, in Proceedings of CP 2005



# 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

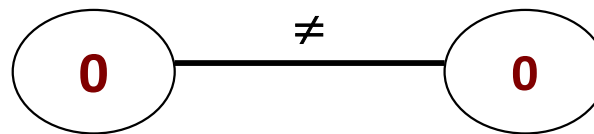
- Um PSR é **consistente nos caminhos** (“*path-consistent*”) se:
  - É consistente nos arcos
  - Para cada restrição  $C_{ij}$  nas variáveis  $X_i$  e  $X_j$ , se existirem restrições  $C_{ik}$  e  $C_{jk}$  entre estas variáveis e uma terceira  $X_k$ , então para todos os *labels* compostos  $\{X_i-v_i, X_j-v_j\}$  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

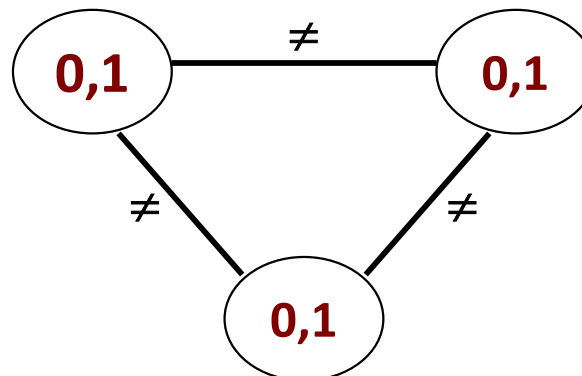
- Consistência dos Nós, Arcos e Caminhos são instâncias do caso geral da **k-consistência**
- Informalmente, uma rede de restrições é **k-consistente** quando para um grupo de  $k$  variáveis  $X_{i_1}, X_{i_2}, \dots, X_{i_k}$ , os valores em cada domínio têm suporte nos valores dos domínios das outras variáveis, considerando este suporte de forma global

# K-Consistência

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

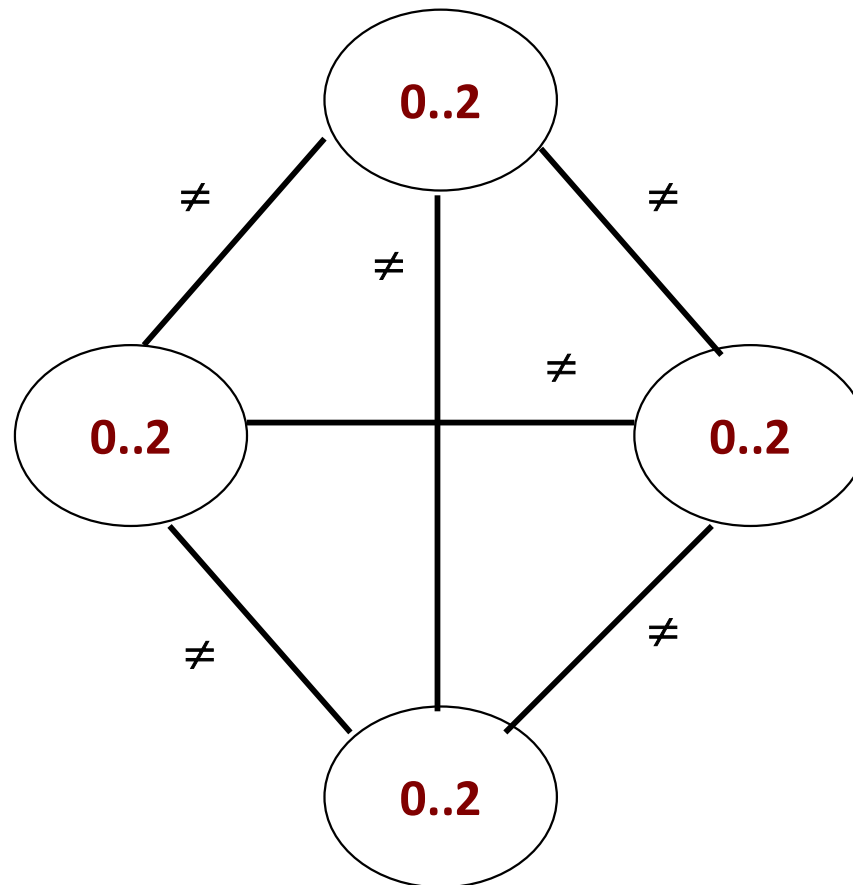


- Rede consistente nos arcos mas não nos caminhos:



# 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

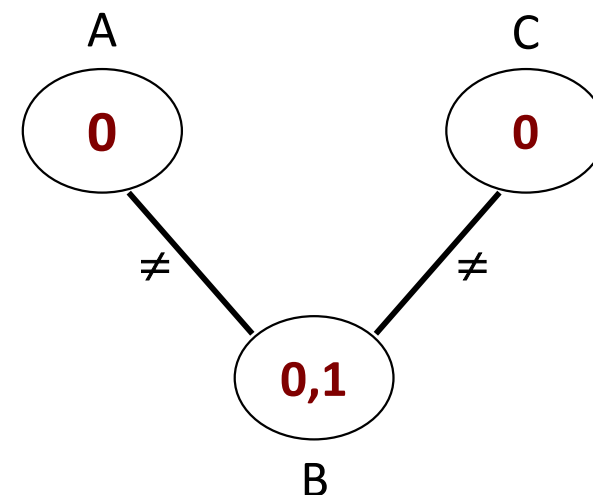
– Os 3 *labels* compostos

$\{A-0, B-1\}$ ,  $\{A-0, C-0\}$  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  $\{B-0, C-0\}$  !



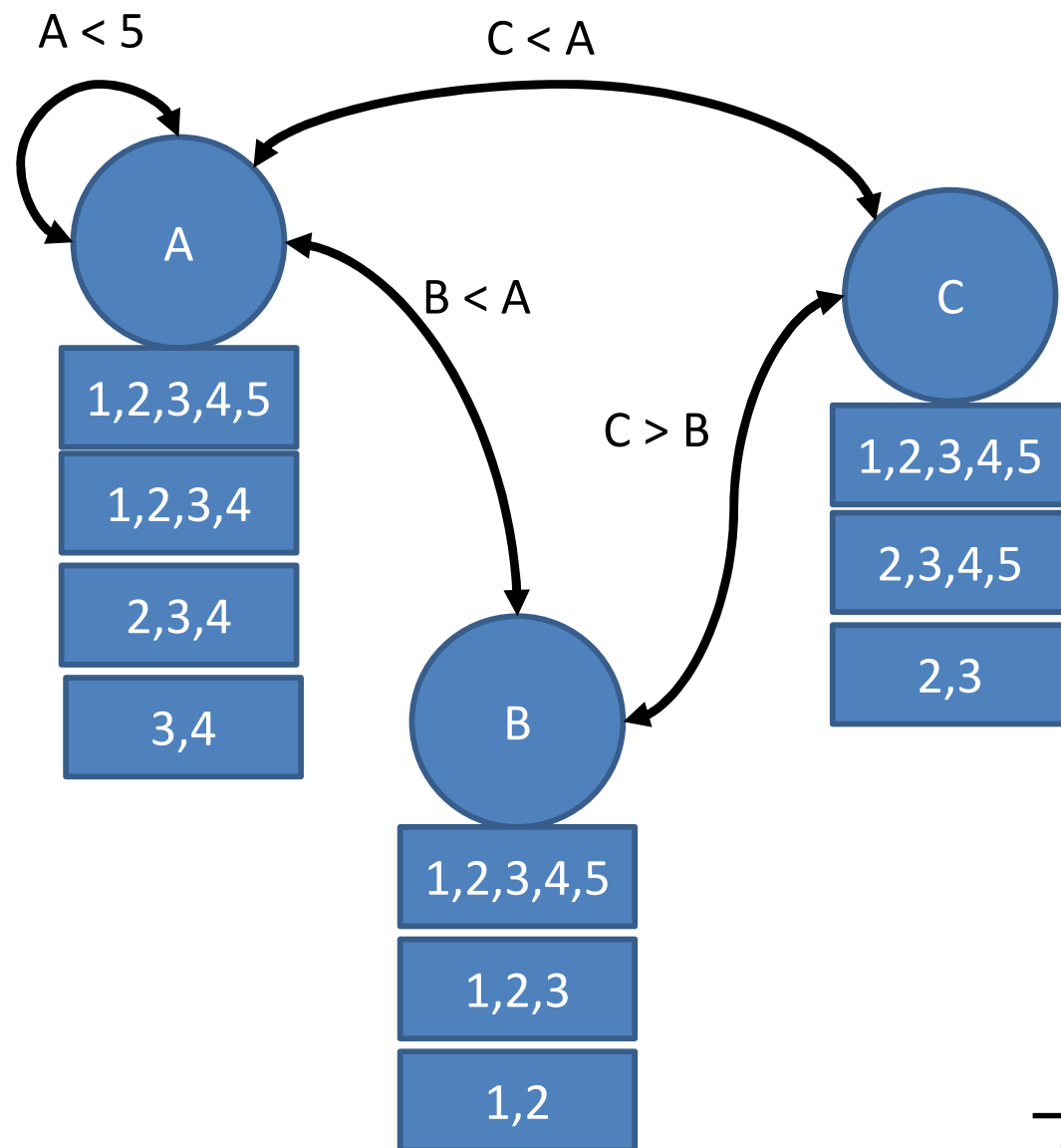
# K-Consistência

- 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



# Consistência e Propagação

$\text{domain}([A, B, C], 1, 5),$   
 $A \neq 5,$   
 $B \neq A,$   
 $C \neq B,$   
 $C \neq A,$   
 $\text{labeling}([\text{max}], [A, B, C]).$



Programação em Lógica com Restrições

## 7. 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
  - As suposições são introduzidas no esquema de raciocínio com restrições, permitindo obter 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

- Efetuar 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 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

# “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

# 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

## 8. 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 – 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 -> 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 – 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 – OZ

- DFKI - German Research Center for Artificial Intelligence
- Baseada num novo modelo computacional que fornece uma base uniforme para programação funcional, POO, 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 – 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 – YAP Prolog

- Excelente sistema de Prolog, desenvolvido pela Universidade do Porto: Vítor Santos Costa, Rogério Reis *et al.*
- 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

# Sistemas de CLP – SICStus

- Swedish Institute of Computer Science
- Extremamente eficiente
- Muitas restrições pré-definidas
- “*Constraint Solvers*”:
  - CLP(R,Q), CLP(B), CLP(FD)
- CHR
- Debugger CLP
- (Ver bloco de slides separado sobre “PLR no SICStus Prolog”)

## Outros Sistemas de Programação com Restrições

- Programação com restrições tem sido adaptada a outras linguagens, existindo diversos sistemas disponíveis (tipicamente como bibliotecas) para linguagens como Java, C / C++, Python, ...
  - Google OR-Tools (C++, Python, Java, .Net)
  - Z3 (C, Java, Python, C#)
  - OptaPlanner (Java, Kotlin, Scala, Jruby, Groovy)
  - IBM ILOG CPLEX (OPL, C++, Python, Java, .Net)
  - ...

Programação em Lógica com Restrições

## **9. 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

# Conferências e Revistas

- Existem várias conferências e revistas científicas focadas em programação (em lógica) com restrições:
  - Constraints - An International Journal. Springer US (Online ISSN: 1572-9354)
  - International Conference on Principles and Practice of Constraint Programming (27<sup>th</sup> edition - [CP 2021](#))
  - International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (18<sup>th</sup> edition – [CPAIOR 2021](#))
  - Vários workshops e outros eventos regulares co-localizados com outras conferências



# Leitura Adicional

- Marriot, Stuckey: “Programming with Constraints: An Introduction”, MIT press, 1998
- Edward Tsang: “Foundations of Constraint Satisfaction”, Academic Press, 1993
- Rossi, F., van Beek, P. and Walsh, T. (Eds.): “Handbook of Constraint Programming”, Elsevier, 2006
- Dechter, R.: “Constraint Processing”, Morgan Kaufmann, 2003
- 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>

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

# Programação em Lógica com Restrições

**Luís Paulo Reis e Daniel Castro Silva**

**Março de 2022**

Parcialmente baseado em slides anteriores de Henrique L. Cardoso ([hlc@fe.up.pt](mailto:hlc@fe.up.pt)), Luís Paulo Reis ([lpreis@fe.up.pt](mailto:lpreis@fe.up.pt)), Daniel Castro Silva ([dcs@fe.up.pt](mailto:dcs@fe.up.pt)), Pedro Barahona, John Hooker, Willem-Jan van Hoeve e outros autores