

# Constraint Programming

---

## - An overview

- Examples of decision (making) problems
- Declarative Modelling with Constraints
- Finite and Continuous Domains
- Constraint Propagation



# Constraint Problems: Examples

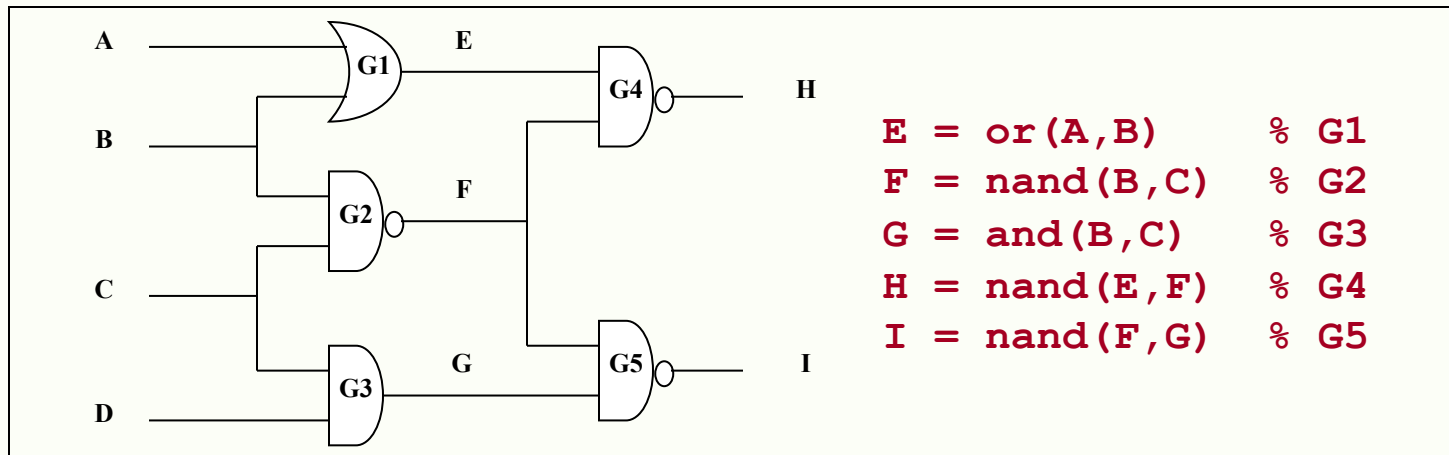
---

- Decision Making Problems include:
  - Modelling of Digital Circuits
  - Production Planning
  - Network Management
  - Scheduling
  - Assignment (Colouring, Latin/ Magic Squares, Sudoku, Circuits, ...)
  - Assignment and Scheduling (Timetabling, Job-shop)
  - Filling and Containment
- Typically a problem may be represented by different models, some of which may be more adequate (ease of modelling, efficiency of solving in a given solver, etc)

# Modeling of Digital Circuits

Goal (Example): Determine a test pattern that detects some faulty gate

- Variables:
  - Signals in the circuit
- Domain:
  - Booleans: 0/1 (or True/False, or High/Low)
- Constraints:
  - Equality constraints between the output of a gate and its “boolean operation” (e.g. and, or, not, nand, ...)



# Production Planning

---

Goal (Example): Determine a production plan

- Variables:
  - Quantities of goods to produce
- Domain:
  - Rational/Reals or Integers
- Constraints:
  - Equality and Inequality (linear) constraints to model resource limitations, minimal quantities to produce, costs not to exceed, balance conditions, etc...

**Find  $x$ ,  $y$  and  $z$  such that**

**$4x + 3y + 6z \leq 1500$       % resources used do not exceed 1500**

**$x + y + z \geq 300$       % production not less than 300 units**

**$x \leq z + 20$       %  $x$  units within  $z \pm 20$  units**

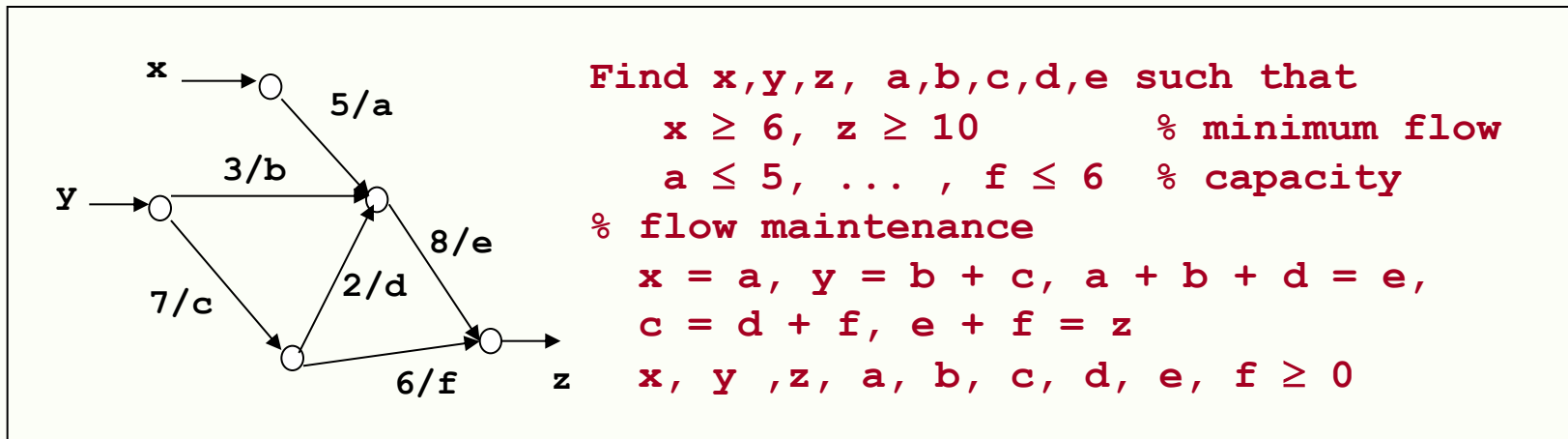
**$x \geq z - 20$**

**$x, y, z \geq 0$       % non negative production**

# Network Management

Goal (Example): Determine acceptable traffic on a network

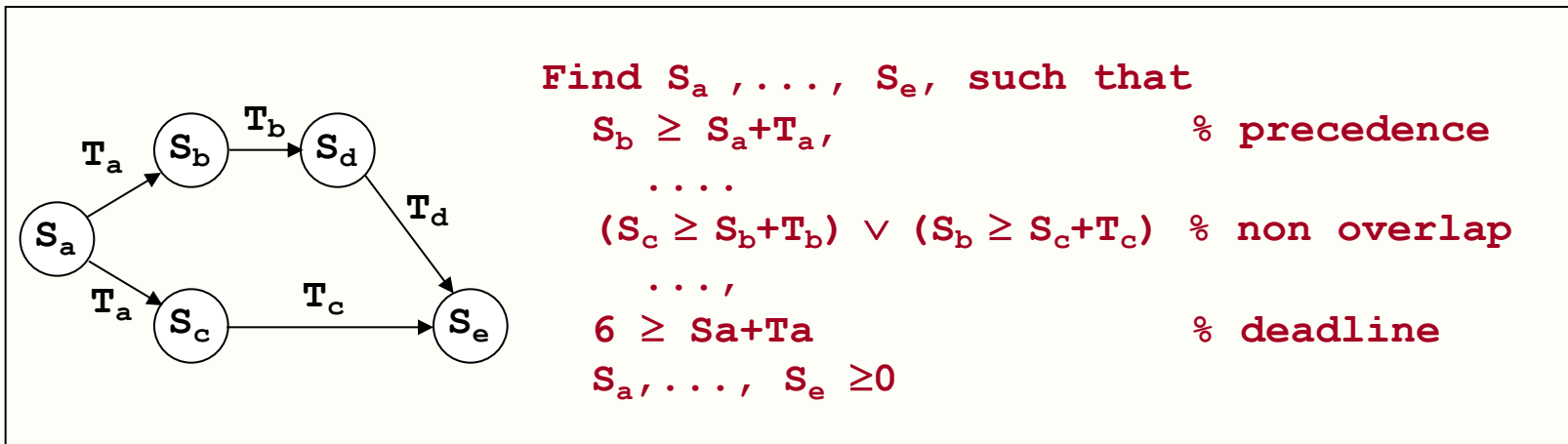
- Variables:
  - Flows in each edge
- Domain:
  - Rational/Reals (or Integers)
- Constraints:
  - Equality and Inequality (linear) constraints to model capacity limitations, flow maintenance, costs, etc...



# Scheduling

Goal (Example): Assign timing/precedence to tasks

- Variables:
  - Start Timing of Tasks, Duration of Tasks
- Domain:
  - Rational/Reals or Integers
- Constraints:
  - Precedence Constraints, Non-overlapping constraints, Deadlines, etc...



# Assignment

---

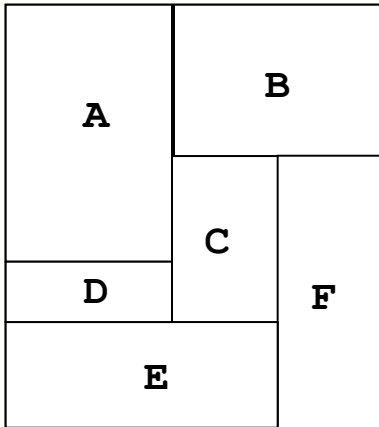
Many constraint problems can be classified as assignment problems. In general all that can be stated is that these problems follow a general CSP goal :

Assign values to the variables to satisfy the relevant constraints.

- Variables:
  - Objects / Properties of objects
  
- Domain:
  - Finite Discrete /Integer or Infinite Continuous /Real or Rational Values
    - colours, numbers, duration, load
  - Booleans for decisions
  
- Constraints:
  - Compatibility (Equality, Difference, No-attack, Arithmetic Relations)

Some examples may help to illustrate this class of problems

# Assignment (2)



## Graph Colouring (Finite Domains)

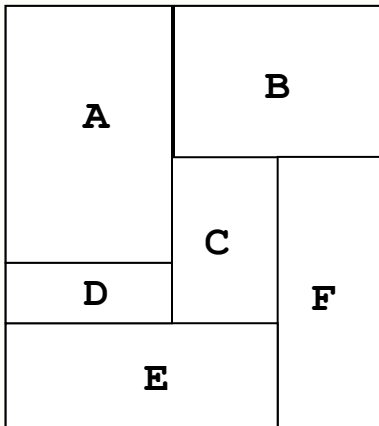
Assign values to A, ..., F,

s.t.  $A, B, \dots, F \in \{\text{red, blue, green}\}$

$A \neq B, A \neq C, A \neq D,$

$B \neq C, B \neq F, C \neq D, C \neq E, C \neq F$

$D \neq E, E \neq F$



## Graph Colouring (0/1 or Booleans – but not SAT)

Assign values to A1, A2, ..., F1, F2

s.t.  $A_r, A_b, A_g, \dots, F_r, F_b, F_g \in \{0, 1\}$

% one and only one colour for A, B, ..., F

$A_r + A_b + A_g = 1;$

.....

% different colours for A and B, ...

$A_r + B_r \leq 1; A_b + B_b \leq 1; A_g + B_g \leq 1;$

.....



# Assignment (3)

Q1		●		
Q2				●
Q3	●			
Q4			●	

## N-queens (Finite Domains):

Assign Values to  $Q_1, \dots, Q_n \in \{1, \dots, n\}$

s.t.  $\forall_{i \neq j}$  noattack ( $Q_i, Q_j$ )

$X_{11}$	$X_{12}$	$X_{13}$
$X_{21}$	$X_{22}$	$X_{23}$
$X_{31}$	$X_{32}$	$X_{33}$

## Latin Squares (similar to Sudoku):

Assign Values to  $X_{11}, \dots, X_{33} \in \{1, \dots, 3\}$

s.t.  $\forall_k \forall_i \forall_{j \neq i} X_{ki} \neq X_{kj}$  % same row

$\forall_k \forall_i \forall_{j \neq i} X_{ik} \neq X_{jk}$  % same column

$X_{11}$	$X_{12}$	$X_{13}$
$X_{21}$	$X_{22}$	$X_{23}$
$X_{31}$	$X_{32}$	$X_{33}$

## Magic Squares:

Assign Values to  $X_{11}, \dots, X_{33} \in \{1, \dots, 9\}$

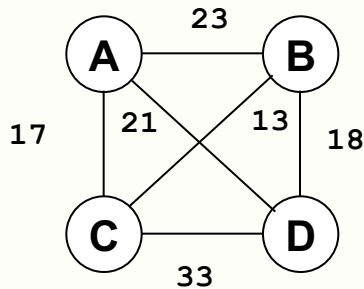
s.t.  $\forall_i \forall_{j \neq i} \sum_k X_{ki} = \sum_k X_{kj} = M$  % same rows sum

$\forall_i \forall_{j \neq i} \sum_k X_{ik} = \sum_k X_{jk} = M$  % same cols sum

$\sum_k X_{kk} = \sum_k X_{k,n-k+1} = M$  % diagonals

$\forall_{i \neq k} \forall_{j \neq l} X_{ij} \neq X_{kl}$  % all different

# Assignment (3)



## Travelling Salesperson (Finite Domains)

Find values for  $A, B, C, D \in \{1, \dots, 4\}$

s.t.  $A \neq B, \dots, C \neq D$

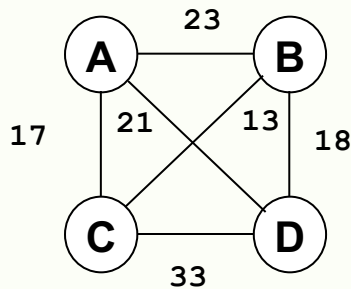
% a permutation of  $[A, B, C, D]$

if  $A = B+1$  then  $X_A = L_{ba}$ ,

...

if  $D = C+1$  then  $X_D = L_{cd}$

$X_A + X_B + X_C + X_D \leq k$



## Travelling Salesperson (0/1 or Booleans – but not SAT)

Find decision values for  $X_{ab} \dots X_{dc} \in \{0, 1\}$

s.t.  $\forall_a \sum_k X_{ak} = 1$

$\forall_a \sum_k X_{ka} = 1$

... no subcycle constraints

$\sum_a \sum_b X_{ab} L_{ab} < k$

# Mixed: Assignment and Scheduling

Goal (Example): Assign values to variables

- Variables:

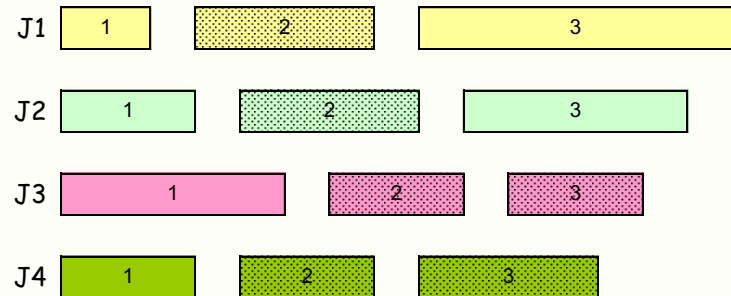
- Start Times, Durations, Resources used

- Domain:

- Integers (typically) or Rationals/Reals

- Constraints:

- Compatibility (Conditional, Disjunctive, Difference, Arithmetic Relations)



## Job-Shop

Assign values to  $S_{ij} \in \{1, \dots, n\}$  % time slots

and to  $M_{ij} \in \{1, \dots, m\}$  % machines available

% precedence within job

$$\forall_j \forall_{i < k} S_{ij} + D_{ij} \leq S_{kj}$$

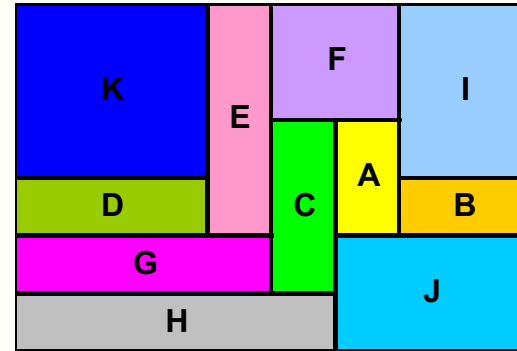
% either no-overlap or different machines

$$\forall_{i,j,k,l} (M_{ij} = M_{kl}) \rightarrow (S_{ij} + D_{ij} \leq S_{kl}) \vee (S_{kl} + D_{kl} \leq S_{ij})$$

# Filling and Containment

Goal (Example): Assign values to variables

- Variables:
  - Point Locations
- Domain:
  - Integers (typically) or Rationals/Reals
- Constraints:
  - Non-overlapping (Disjunctive, Inequality)



## Filling

Assign values to  $X_i \in \{1, \dots, X_{\max}\}$  % X-dimension

$Y_i \in \{1, \dots, Y_{\max}\}$  % Y-dimension

% no-overlapping rectangles

$\forall_{i,j} \quad (X_i + Lx_i \leq X_j)$  % I to the left of J

$(X_j + Lx_j \leq X_i)$  % I to the right of J

$(Y_i + Ly_i \leq Y_j)$  % I in front of J

$(Y_j + Ly_j \leq Y_i)$  % I in back of J

# Constraint Satisfaction Problems

---

- Other Examples (from CP-16):
  - Finding Patterns for DataMining
    - Rather than finding rules (as in ID3 /CS4.5) whole sets must be obtained
    - e.g. sequences of letters in ADN / Protein searches
  - Hospital Residence Problem (with pairs)
    - Kind of Stable Marriage Problem but pairings make it NP-Hard
    - Both Hospitals and Residents (junior doctors) have a list of preferences
    - Pairs of Residents have joint preferences

# Constraint Satisfaction Problems

---

- Formally a constraint satisfaction problem (CSP) can be regarded as a tuple  $\langle X, D, C \rangle$ , where
  - $X = \{ X_1, \dots, X_n \}$  is a set of variables
  - $D = \{ D_1, \dots, D_n \}$  is a set of domains (for the corresponding variables)
  - $C = \{ C_1, \dots, C_m \}$  is a set of constraints (on the variables)
- Solving a constraint problem consists of determining values  $x_i \in D_i$  for each variable  $X_i$ , satisfying all the constraints  $C$ .
- Intuitively, a constraint  $C_i$  is a limitation on the values of its variables.
- More formally, a constraint  $C_i$  (with arity  $k$ ) over variables  $X_{i1}, \dots, X_{ik}$  ranging over domains  $D_{i1}, \dots, D_{ik}$  is a subset of the cartesian cartesian  $D_{j1} \times \dots \times D_{jk}$ .

$$C_i \subseteq D_{j1} \times \dots \times D_{jk}$$

# Constraints and Optimisation Problems

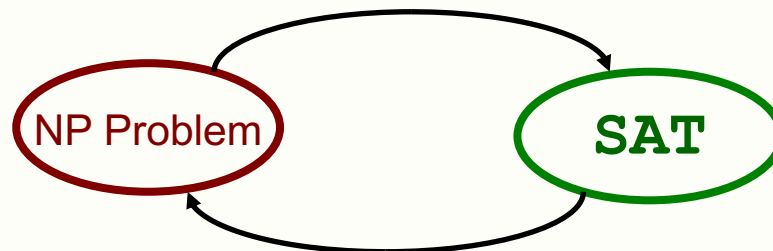
---

- In many cases, one is interested not only in satisfying some set of constraints but also in finding among all solutions those that optimise a certain objective function (minimising a cost or maximising some positive feature).
- Formally a constraint (satisfaction and) optimisation problem (CSOP or COP) can be regarded as a tuple  $\langle V, D, C, F \rangle$ , where
  - $X = \{ X_1, \dots, X_n \}$  is a set of variables
  - $D = \{ D_1, \dots, D_n \}$  is a set of domains (for the corresponding variables)
  - $C = \{ C_1, \dots, C_m \}$  is a set of constraints (on the variables)
  - $F$  is a function on the variables
- Solving a constraint satisfaction and optimisation problem consists of determining values  $x_i \in D_i$  for each variable  $X_i$ , satisfying all the constraints  $C$  and that optimise the objective function.

# Decision Problems are NP-complete

---

- All the problems presented are decision problems in that a decision has to be made regarding the value to assign to each variable.
- Non-trivial decision making problems are untractable, i.e. they lie in the class of NP problems.
- Formally, these are the problems that can be solved in polinomial time by a non-deterministic machine, i.e. one that “guesses the right answer”.
- For example, in the graph colouring problem ( $n$  nodes,  $k$  colours), if one has to assign colours to  $n$  nodes, a non-deterministic machine could guess a solution in  $O(n)$  steps.
- As a class, NP-complete problems may be converted in polinomial time onto other NP-complete problems (SAT, in particular).





# Decision Problems are NP-complete

- No one has already found a polynomial algorithm to solve SAT (or any other NP problem), and hence the conjecture  $P \neq NP$  (perhaps one of the most challenging open problems in computer science) is regarded as true.
- Hence, with real machines and non trivial problems, one has to guess the adequate values for the variables and make mistakes. In the worst case, one has to test  $O(k^n)$  potential solutions.
- Just to have an idea of the complexity, the table below shows the time needed to check  $kn$  solutions, assuming one solution is examined in  $1 \mu\text{sec}$  (times in secs).

$k^n$	$n$					
	10	20	30	40	50	60
2	1.0E-03	1.0E+00	1.1E+03	1.1E+06	1.1E+09	1.2E+12
3	5.9E-02	3.5E+03	2.1E+08	1.2E+13	7.2E+17	4.2E+22
4	1.0E+00	1.1E+06	1.2E+12	1.2E+18	1.3E+24	1.3E+30
5	9.8E+00	9.5E+07	9.3E+14	9.1E+21	8.9E+28	8.7E+35
6	6.0E+01	3.7E+09	2.2E+17	1.3E+25	8.1E+32	4.9E+40

1 hour =  $3.6 * 10^3 \text{ sec}$

1 year =  $3.2 * 10^7 \text{ sec}$

TOUniv =  $4.7 * 10^{17} \text{ sec}$

# Decision Problems are NP-complete

---

- Still, constraint solving problems are NP-complete problems (as SAT is).
- If a non-deterministic machine (that guesses correctly) can **solve** a problem in polynomial time, then a real deterministic machine can **check** in polynomial time whether a potential solution satisfies all the constraints.
- More important: with an appropriate search strategy, many instances of NP-complete problems can be solved in quite acceptable times.
- Hence, search plays a fundamental role in solving this kind of problems. Adequate search methods and appropriate heuristics can often solve large instances of these problems in very acceptable time.

# Search Strategies

---

- There are two main types of search strategies that have been adopted to solve combinatorial problems:

## Complete Backtrack Search Methods:

- Solutions are incrementally **completed**, by assigning values to “undecided” variables and backtrack whenever any constraint is violated;
- These methods are complete: if a solution exists it is found in finite time.
- More importantly, they can proof non-satisfiability.

## Incomplete Local Search Methods:

- Complete “solutions” are incrementally **repaired**, by changing the values assigned to some of the variables until a “real solution” is found;
- These local search methods are not guaranteed to avoid revisiting the same solutions time and again and are therefore incomplete.
- They are often very efficient to find very good solutions (local optima)

# Optimisation Problems are NP-hard

---

- Optimisation problems are typically NP-hard problems in that solving them is at least as difficult as solving the corresponding decision problem.
- In practice these problems cannot be solved in polynomial time by a non-deterministic machine, nor can they be checked by a deterministic machine.
- In fact, to find an optimal solution it is not enough to find it ... It is necessary to show that it is better than all other solutions!
- Being harder than the decision problems, optimisation problems also require adequate search strategies, if larger instances are to be solved.
  - In complete search, detection of failure and subsequent backtracking may be imposed if the partial solution can be proved to be no better than one already found (branch & bound).

# Declarative Programming

---

- Programming a combinatorial problem thus requires
  - The specification of the constraints of the problem; and
  - The specification of a search algorithm
- The separation of these two aspects has for a long time been advocated by several programming paradigms, namely functional programming and logic programming.
- Logic programming in particular has a built-in mechanism for search (backtracking) that makes it easy to extend into constraint (logic) constraint programming, by “replacing” its underlying **resolution** to **constraint propagation**. A number of Constraint Logic Programming languages have been proposed (CHIP, ECLiPSE, GNU Prolog, SICStus) to explore this extension of logic programming.
- More recently, other declarative languages such as Comet (OO-like), Choco (Java Library) and Zinc, provide more convenient data structures for modelling, maintaining a declarative approach.

# Constraint Programming

---

Constraint Programming (and Languages) is driven by a number of goals

- Expressivity
  - Constraint Languages should be able to easily specify the variables, domains and constraints (e.g. conditional, global, etc...);
- Declarative Nature
  - Ideally, programs should specify the constraints to be solved, not the algorithms used to solve them
- Efficiency
  - Solutions should be found as efficiently as possible, i.e. with the minimum possible use of resources (time and space).

These goals are partially conflicting goals and have led to the various developments in this research and development area.

# Search Methods - Pure Backtracking

---

- In this course we will focus on these two aspects of Constraint Programming:
  - **Declarative Modelling**
    - How to specify as naturally as possible the problem we want to solve
  - **Efficient Execution**
    - How to solve the problems thus specified as efficiently as possible, combining, as we should study, Heuristics with constraint propagation.
- These topics will be studied in the context of two types of domains
  - **Finite Domains**
    - discrete domains, basically integer intervals)
  - **Continuous Domains**
    - in principle, the difference between two values can be as small as we may want.

# Constraint Programming - Finite Domains

---

- In Finite domains we will see that the the efficiency obtained in solving a problem with CP depends on many issues that will be addressed in the course:
  1. Formalization of Constraint Propagation
  2. Types of constraints and their main features
  3. Alternative models
    - a. Redundant Constraints
    - b. Symmetry Breaking Constraints
  4. Heuristics that are most commonly used
  5. Testing these techniques with Choco in several non-trivial examples
- These aspects will be studied in the first part of the course (first 6 weeks).



# Constraint Programming - Continuous Domains

---

Continuous constraints require somewhat different methods for constraint propagation as well as enumeration. The main differences to consider are:

1. In a domain  $lo .. hi$  there are infinite values to consider. Hence enumeration cannot be a simple test of the alternative values, backtracking if necessary.
  2. Constraints should consider variables whose domains are intervals, and adapt standard arithmetic to consider such domains – interval arithmetic.
  3. Advanced methods can be used to propagate constraints, more sophisticated than naïve methods adapted from the finite domains (e.g. interval Newton).
  4. Approximations are often necessary (e.g. rounding off arithmetic operations) and care must be taken that errors are not made (so as to lose solutions).
- Constraints in these continuous domains will be covered in the second part of the course, by Prof. Jorge Cruz.

# Constraint Programming - Continuous Domains

---

A summary of this second part:

1. Continuous Constraint Satisfaction Problems
2. Continuous Constraint Reasoning
  - a. Representation of Continuous Domains
  - b. Pruning and Branching
3. Solving Continuous CSPs
  - a. Constraint Propagation
  - b. Consistency Criteria
4. Practical Examples

# Constraint Programming - Continuous Domains

---

A major concern of dealing with continuous constraints regards constraint propagation.

For these part of the course some topics will be dealt more formally, namely:

1. Interval Constraints Overview
2. Intervals, Interval Arithmetic and Interval Functions
3. Interval Newton Method
4. Associating Narrowing Functions to Constraints
5. Constraint Propagation and Consistency Enforcement

# Assessment

---

- Evaluation consists of the following components
  - Project 1 – Finite Domains Problem
  - Mini-Test 1 – Finite Domains Concepts
  - Project 2 – Continuous Domains Problem
  - Mini-Test 2 – Continuous Domains Concepts
- Projects are made in team work (2 students per group) and the tests assess the students individually.
- All components have the same weight for the final grade.
- Students that do not get the minimum grade, are allowed to do a repetition exam if they get at least an average grade of 8/20 in the two projects.
- Exact dates to be announced –
  - Project 1 and Mini-test 1 at mid-term (end October / early November)
  - Project 2 and Mini-test 2 at the end of semester (end December /early January)

# Constraint Propagation

- As mentioned, non trivial constraint satisfaction problems are typically NP-complete so there is no known algorithm to solve them in polynomial time.
- In practice, this means that solving them require some form of **search**.
- Given a problem with  $n$  variables each with  $k$  values in its domain, the number of possible solutions is  $k^n$ . As such brute force algorithms that explore all the possibilities are doomed to be unpractical in instances with a relatively low number of variables.

$k^n$	$n$					
	10	20	30	40	50	60
$k=2$	1.0E-03	1.0E+00	1.1E+03	1.1E+06	1.1E+09	1.2E+12
$k=3$	5.9E-02	3.5E+03	2.1E+08	1.2E+13	7.2E+17	4.2E+22
$k=4$	1.0E+00	1.1E+06	1.2E+12	1.2E+18	1.3E+24	1.3E+30
$k=5$	9.8E+00	9.5E+07	9.3E+14	9.1E+21	8.9E+28	8.7E+35
$k=6$	6.0E+01	3.7E+09	2.2E+17	1.3E+25	8.1E+32	4.9E+40

1 hour =  $3.6 * 10^3$  sec    1 year =  $3.2 * 10^7$  sec    TOUniv =  $4.7 * 10^{17}$  sec

# Constraint Propagation

---

- Given the need for search it is very important to decrease the space of potential solutions that have to be tested.
- This is a key goal of **constraint propagation**: take into account each and all of the constraints of the problem to decrease the possible values a variable might take.
- More specifically, constraint propagation uses constraints **actively**: the domain of a variable should be decreased if it no longer makes it possible to satisfy the constraint.
- Constraint propagation can be illustrated with the well known SENDMORY crypto-arithmetic problem.

C4	C3	C2	C1	
	S	E	N	D
+	M	O	R	E
<hr/>				
M	O	N	E	Y

- We will use the model that constrains the variables in the sums of each column, including the carries.

# Constraint Propagation

- These are the variables and domains of the problem.

```
0. [S,E,N,D,M,O,R,Y] in 0..9
1. [C1, C2, C3, C4] in 0..1
2. alldif([S,E,N,D,M,O,R,Y])
3. M > 0
4. S > 0
5. D + E          = Y + 10 * C1
6. N + R + C1    = E + 10 * C2
7. E + O + C2    = N + 10 * C3
8. S + M + C3    = O + 10 * C4
9. C4 = M
```

C4	C3	C2	C1	
	S	E	N	D
+	M	O	R	E
<hr/>				
M	O	N	E	Y

- With a naïve approach, and assuming that the search is only done in the digit variables (not in the carries, that are implied) the size of the search space is (since there are 8 variables, each with 10 values in the domain)

$$8^{10} = 1\,073\,741\,824 \approx 10^9$$

- Even if we consider during search that the variables are all different the size of the search space is

$$10 \times 9 \times \dots \times 3 = 1\,814\,400 \approx 2 \cdot 10^6$$

# Constraint Propagation

0. [S,E,N,D,M,O,R,Y] in 0..9
1. [C1, C2, C3, C4] in 0..1
2. alldif([S,E,N,D,M,O,R,Y])
3.  $M > 0$
4.  $S > 0$
5.  $D + E = Y + 10 * C1$
6.  $N + R + C1 = E + 10 * C2$
7.  $E + O + C2 = N + 10 * C3$
8.  $S + M + C3 = O + 10 * C4$
9.  $C4 = M$

C4	C3	C2	C1	
	S	E	N	D
+	M	O	R	E
M	O	N	E	Y

- We now analyse how to decrease the search space. We start by noting that both C4 and M must be 1, given constraints 3, 9, and 1.
- In fact
  - M must be greater than 0 (constraint 3);
  - M must be equal to C4 (constraint 9);
- But since
  - C4 may only be 0 or 1 (domain constraint 1)
 It must be
  - $M = C4 = 1$

1	C3	C2	C1	
	S	E	N	D
+	1	O	R	E
1	O	N	E	Y



# Constraint Propagation

0. [S,E,N,D,M,O,R,Y] in 0..9
1. [C1, C2, C3, C4] in 0..1
2. alldif([S,E,N,D,M,O,R,Y])
3.  $M > 0$
4.  $S > 0$
5.  $D + E = Y + 10 * C1$
6.  $N + R + C1 = E + 10 * C2$
7.  $E + O + C2 = N + 10 * C3$
8.  $S + 1 + C3 = O + 10$
9.  $C4 = M$

	1	C3	C2	C1	
		S	E	N	D
+	1	O	R	E	
	1	O	N	E	Y

- Now constraint 8 can be rewritten as  $S = O + 9 - C3$ .
- Since S cannot be greater than 9, there are two possibilities here.
  - $S = 9$  and  $O = C3$ ; or
  - $S = 8$  and  $O = C3 + 1$
- Let us explore the first hypothesis. Since
  - $O \neq 1$  (as it must be different from  $M=1$ ); and
  - $O = C3$
 the only remaining possibility, as  $C3$  must be 0 or 1 is
  - $O = 0$ ; and
  - $C3 = 0$

	1	0	C2	C1	
		9	E	N	D
+	1	0	R	E	
	1	0	N	E	Y

# Constraint Propagation

0. [S,E,N,D,M,O,R,Y] in 0..9
1. [C1, C2, C3, C4] in 0..1
2. alldif([S,E,N,D,M,O,R,Y])
3. M > 0
4. S > 0
5. D + E = Y + 10 \* C1
6. N + R + C1 = E + 10 \* C2
7. E + O + C2 = N + 10 \* C3
8. S + 1 + C3 = O + 10
9. C4 = M

	1	0	C2	C1	
		9	E	N	D
+	1	0	R	E	
	1	0	N	E	Y

- Now constraint 7 can be rewritten as  $N = E + C2$ .
- Since E and N must be different it must be the case that
  - $C2 = 1$  and
  - $N = E + 1$

	1	0	1	C1	
		9	E	N	D
+	1	0	R	E	
	1	0	N	E	Y

# Constraint Propagation

```

0. [S,E,N,D,M,O,R,Y] in 0..9
1. [C1, C2, C3, C4] in 0..1
2. alldif([S,E,N,D,M,O,R,Y])
3. M > 0
4. S > 0
5. D + E          = Y + 10 * C1
6. N + R + C1 = E + 10
7. N = E + 1
8. S + 1 + C3 = O + 10
9. C4 = M
    
```

1	0	1	C1		
	9	E	N	D	
+	1	0	R	E	
1	0	N	E	Y	

- Combining constraints 6 and 7 we obtain

- $R + 1 + C1 = 10$
- $R = 9 - C1$

- Since we know that R cannot be 9 (the value assigned to S) given constraint 2, then the only possible assignment is

- $R = 8$
- $C1 = 1$

1	0	1	1		
	9	E	N	D	
+	1	0	8	E	
1	0	N	E	Y	

# Constraint Propagation

```

0. [S,E,N,D,M,O,R,Y] in 0..9
1. [C1, C2, C3, C4] in 0..1
2. alldif([S,E,N,D,M,O,R,Y])
3. M > 0
4. S > 0
5. D + E = Y + 10
6. N + R + C1 = E + 10
7. N = E + 1
8. S + 1 + C3 = O + 10
9. C4 = M
    
```

	1	0	1	1	
		9	E	N	D
+	1	0	8	E	
	1	0	N	E	Y

- Now, we may note that

- $Y \geq 2$ , since Y must be different from M and O (constraint 2)
- $E \leq 6$ , since  $N = E + 1$  and both E and N must be less than 8, since they must be different from S and R (constraint 2)

- Hence constraint 5 can be rewritten as

- $D = Y - E + 10$ ; and hence
- $D \geq 2 - 6 + 10 = 6$

- Now D can only take values 6 or 7 (given constraint 2), so

- $D = 6$

and rewrite constraint 5 as  $E = Y + 4$ .

	1	0	1	1	
		9	E	N	6
+	1	0	8	E	
	1	0	N	E	Y

# Constraint Propagation

```

0. [S,E,N,D,M,O,R,Y] in 0..9
1. [C1, C2, C3, C4] in 0..1
2. alldif([S,E,N,D,M,O,R,Y])
3. M > 0
4. S > 0
5. E = Y + 4
6. N + R + C1 = E + 10
7. N = E + 1
8. S + 1 + C3 = O + 10
9. C4 = M
    
```

	1	0	1	1	
		9	E	N	6
+	1	0	8	E	
	1	0	N	E	Y

- But this is not possible, since in this case,
  - E must be greater than 6, and
  - N would be even greater, but it cannot since values 8 and 9 are taken by variables R and S (constraint 2)
- Then we must **backtrack** and try instead
  - D = 7

and rewrite constraint 5 as  $E = Y + 3$ .

	1	0	1	1	
		9	E	N	7
+	1	0	8	5	
	1	0	N	E	Y

# Constraint Propagation

0. `[S, E, N, D, M, O, R, Y]` in `0..9`
1. `[C1, C2, C3, C4]` in `0..1`
2. `alldif([S, E, N, D, M, O, R, Y])`
3. `M > 0`
4. `S > 0`
5. `E = Y + 3`
6. `N + R + C1 = E + 10`
7. `N = E + 1`
8. `S + 1 + C3 = O + 10`
9. `C4 = M`

1	0	1	1	
	9	E	N	7
+	1	0	8	5
1	0	N	E	Y

- Trying `E = 5` and propagating we get
  - `N = 6` (through constraint 7) and
  - `Y = 2` (through constraint 5)

thus solving the problem.

1	0	1	1	
	9	5	6	7
+	1	0	8	5
1	0	6	5	2

# Constraint Propagation

---

- In this case, the active use of the constraints of the problem allowed us to solve the problem very efficiently
  - Only 2 choice points
  - Only one backtracking
- In general **constraint propagation** is at the heart of constraint Programming for two main reasons:
  - It decreases the size of the search space
    - the size of the domains and the number of choice points
  - It provides useful information to guide search
    - NP problems still require heuristics, given their exponential size
- However, in this case, we adopted some special purpose reasoning to obtain propagation, namely
  - Combining several constraints
  - Using adequate arithmetic knowledge
- These techniques will be used later, when dealing with **global constraints**.

# Constraint Propagation

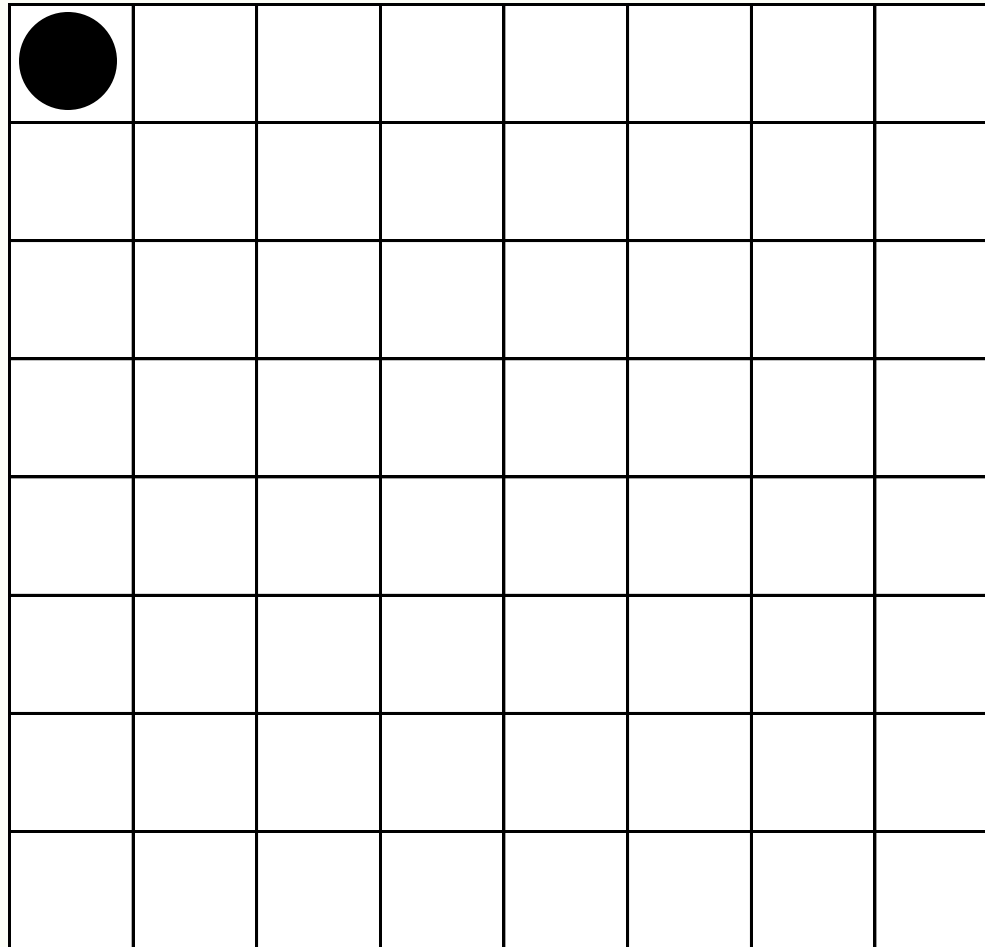
---

- For the moment we may consider a simpler form of reasoning to achieve constraint propagation, that will be illustrated with the 8-queens problem
- First we show the use of backtrack alone towards solving the problem, and compare it later with the combined use of backtrack and constraint propagation.
- The simplest backtracking strategy uses constraints **passively**:
  - Whenever a value is assigned a variable, the constraints whose variables have their variables all assigned are checked for satisfaction
  - If this is not the case, the search backtracks (chronological backtrack).
- This is a typical **generate and test** procedure
  - Firstly, values are generated
  - Secondly, the constraints are tested for satisfaction.
- Of course, tests should be done as soon as possible, i.e. a constraint is checked whenever all its variables are assigned values.



# Backtracking

---



**Tests 0**

**Backtracks 0**

# Backtracking

---

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

●							
●							

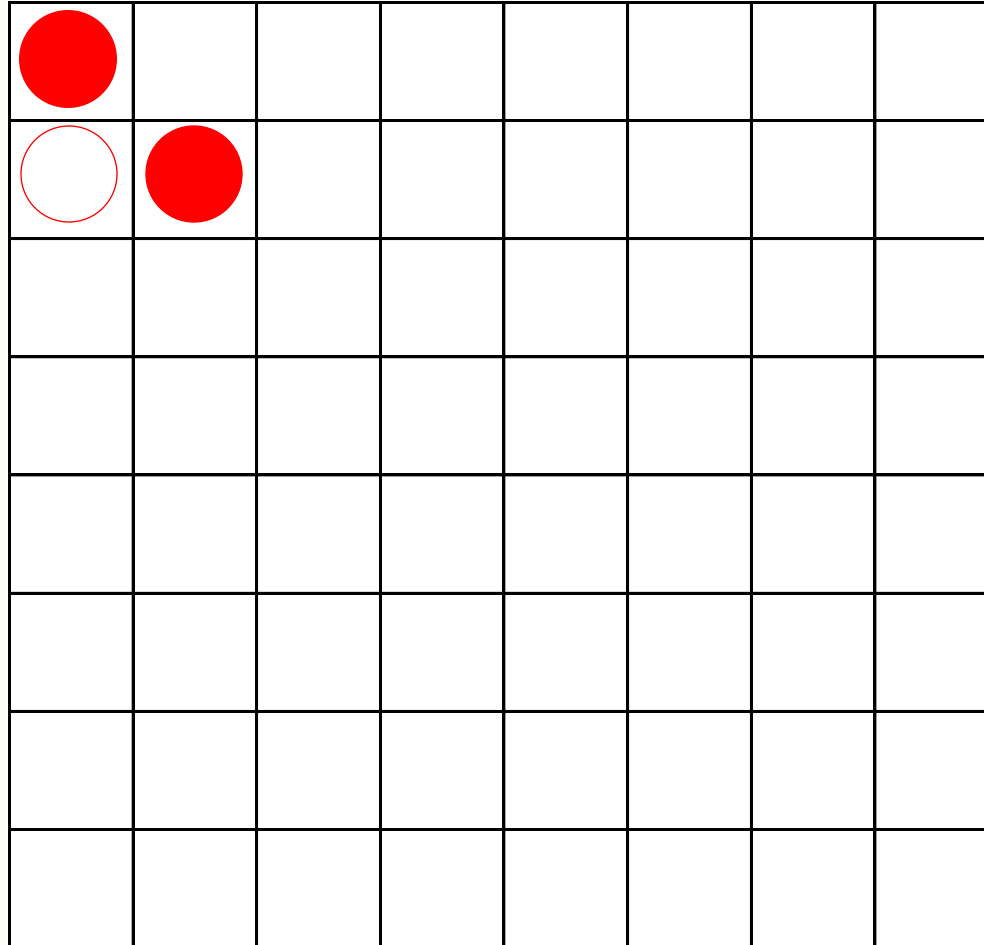
**Tests**  $0 + 1 = 1$

**Backtracks** 0

# Backtracking

---

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



**Tests**  $1 + 1 = 2$

**Backtracks** 0

# Backtracking

---

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

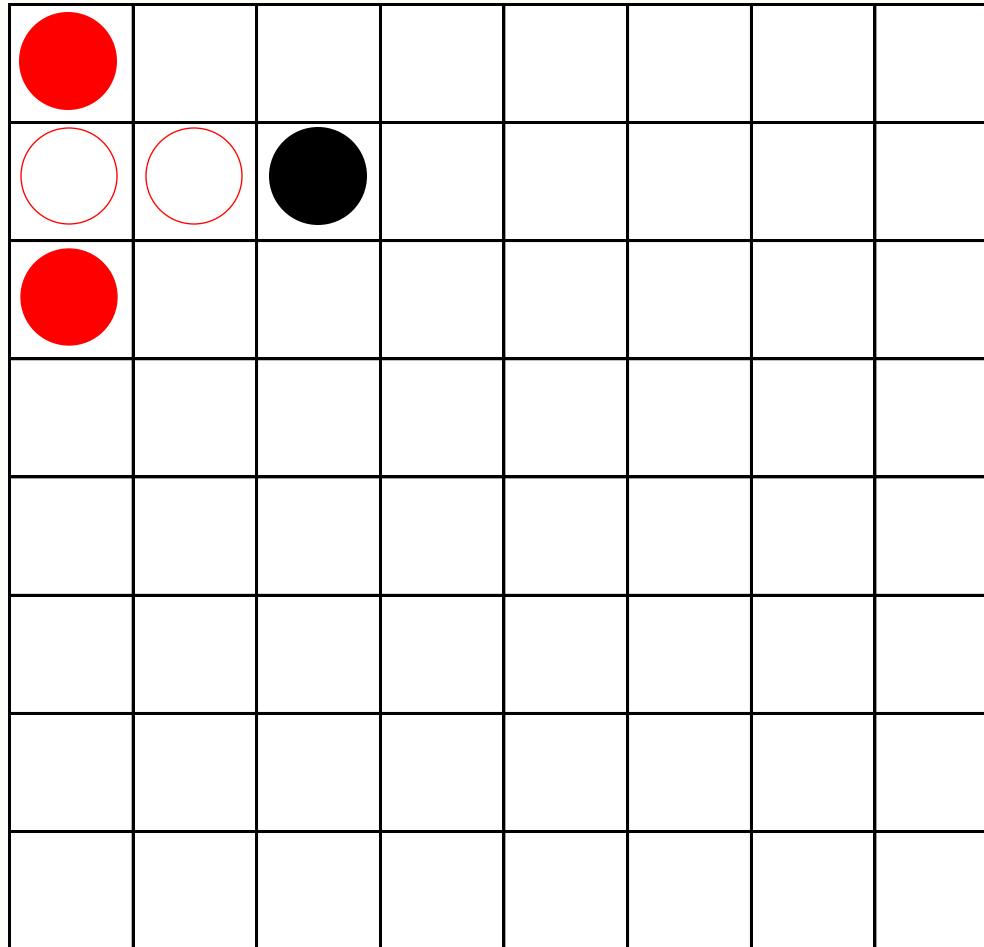
●							
○	○	●					

**Tests  $2 + 1 = 3$**

**Backtracks 0**

# Backtracking

---

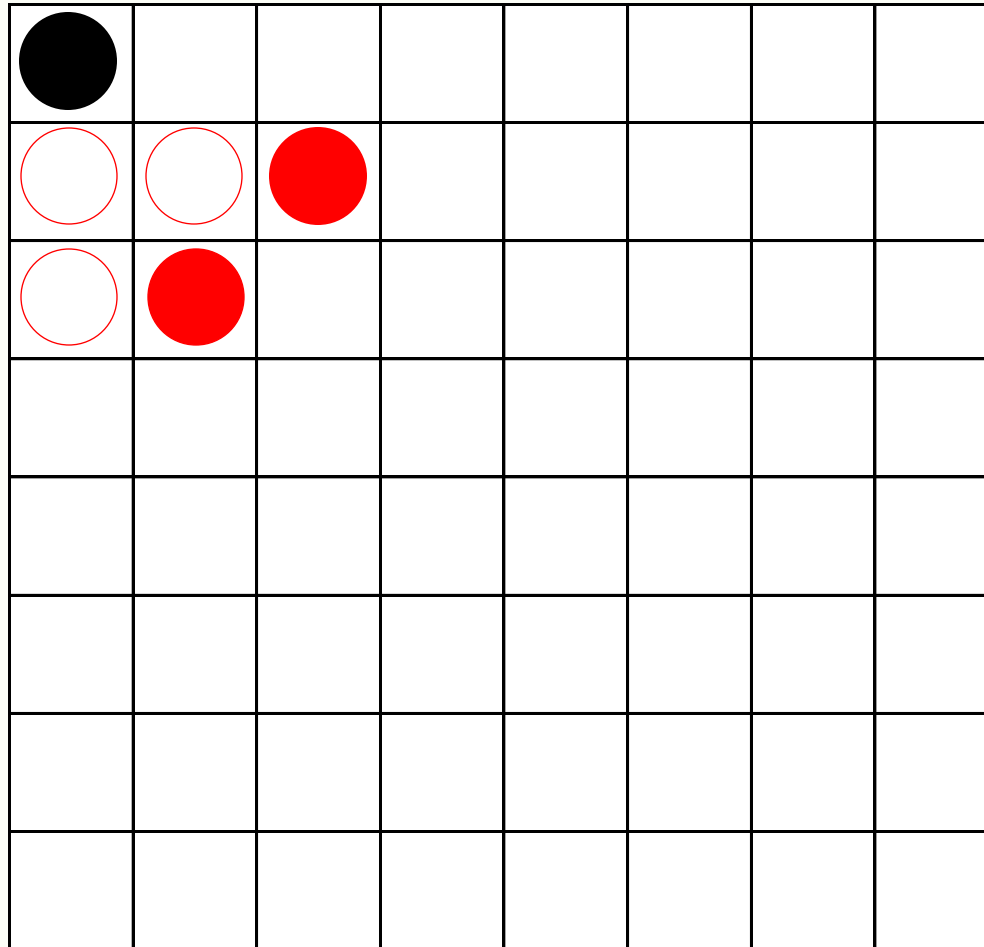


**Tests**  $3 + 1 = 4$

**Backtracks** 0

# Backtracking

---

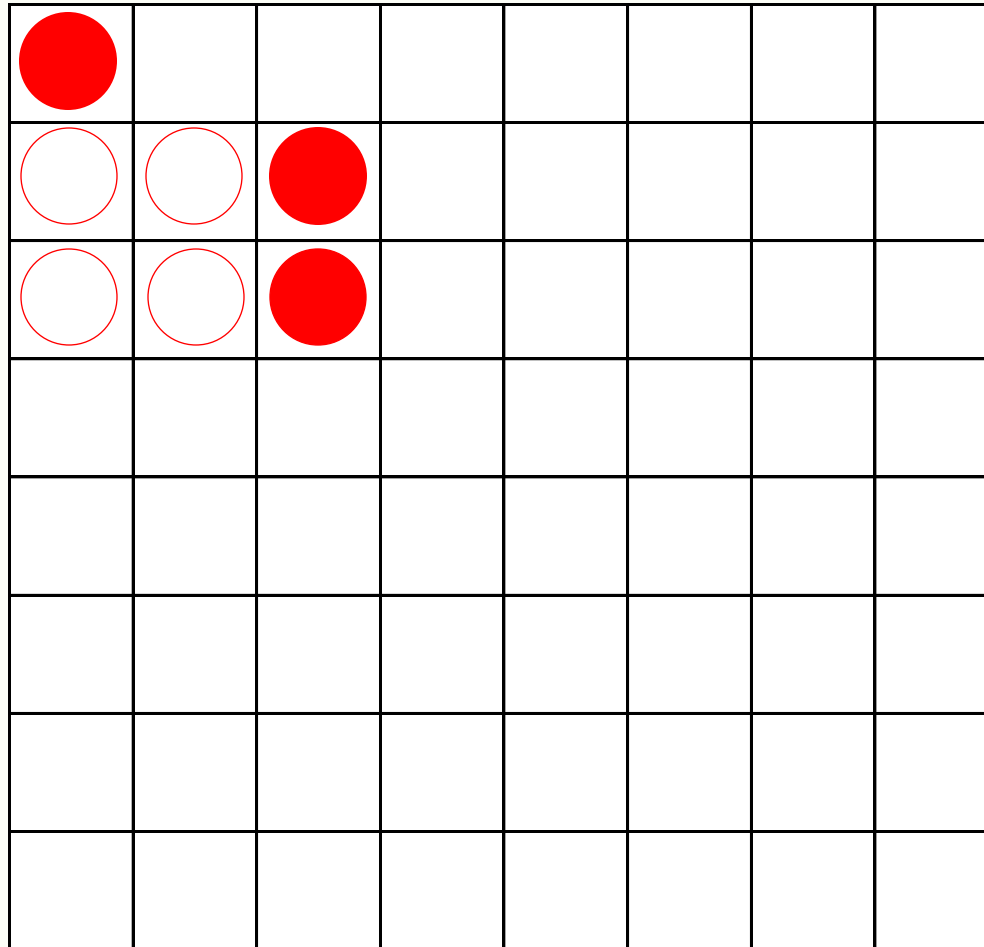


**Tests**  $4 + 2 = 6$

**Backtracks** 0

# Backtracking

---

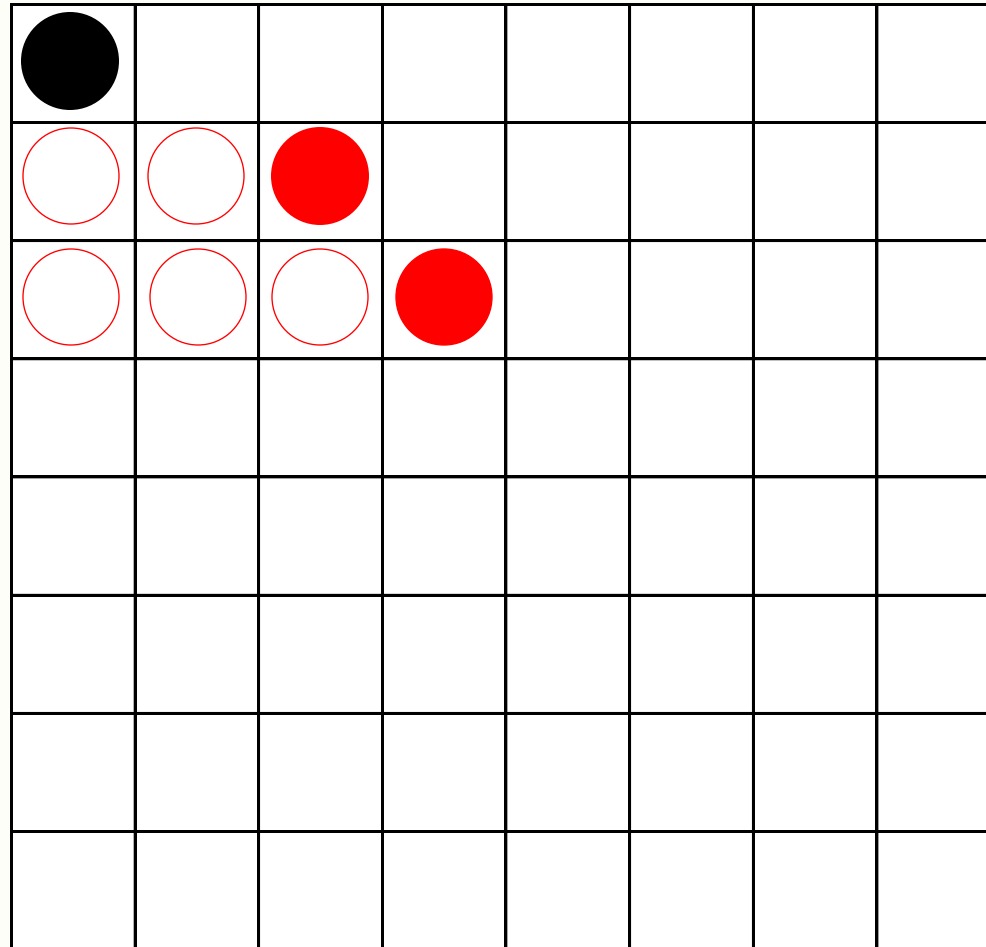


**Tests  $6 + 1 = 7$**

**Backtracks 0**

# Backtracking

---



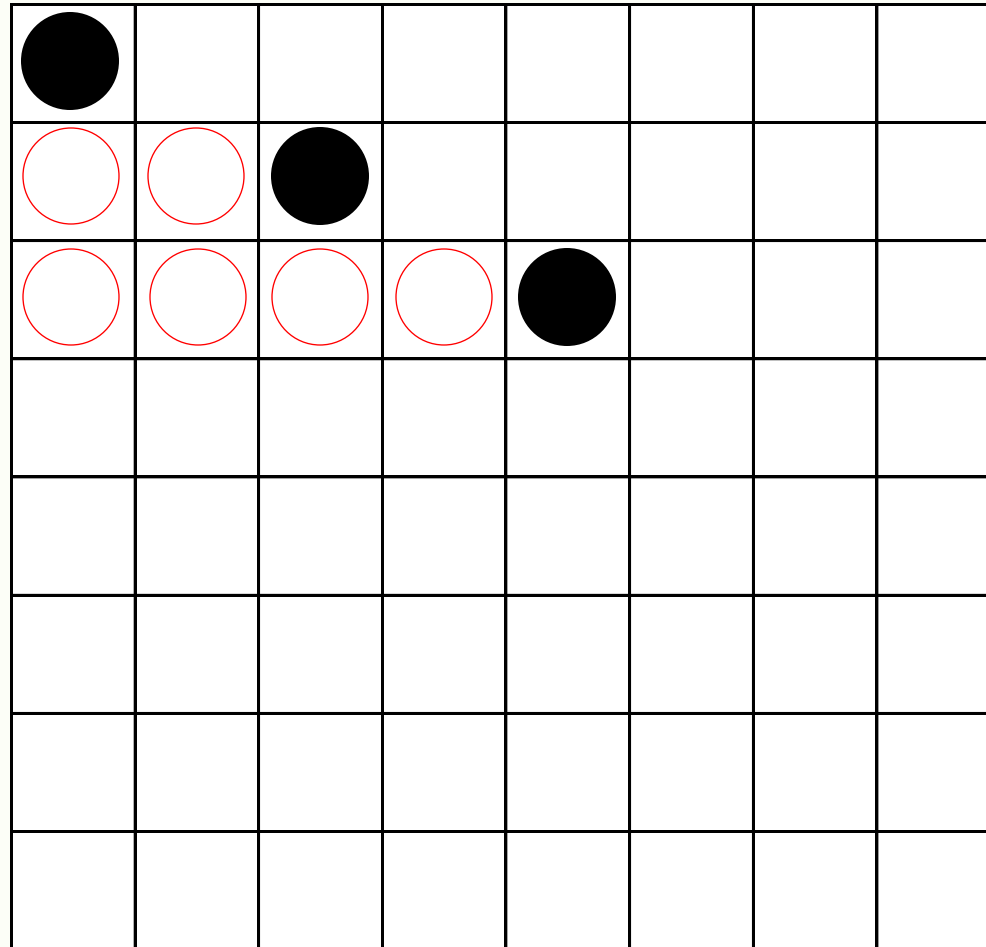
**Tests  $7 + 2 = 9$**

**Backtracks 0**



# Backtracking

---

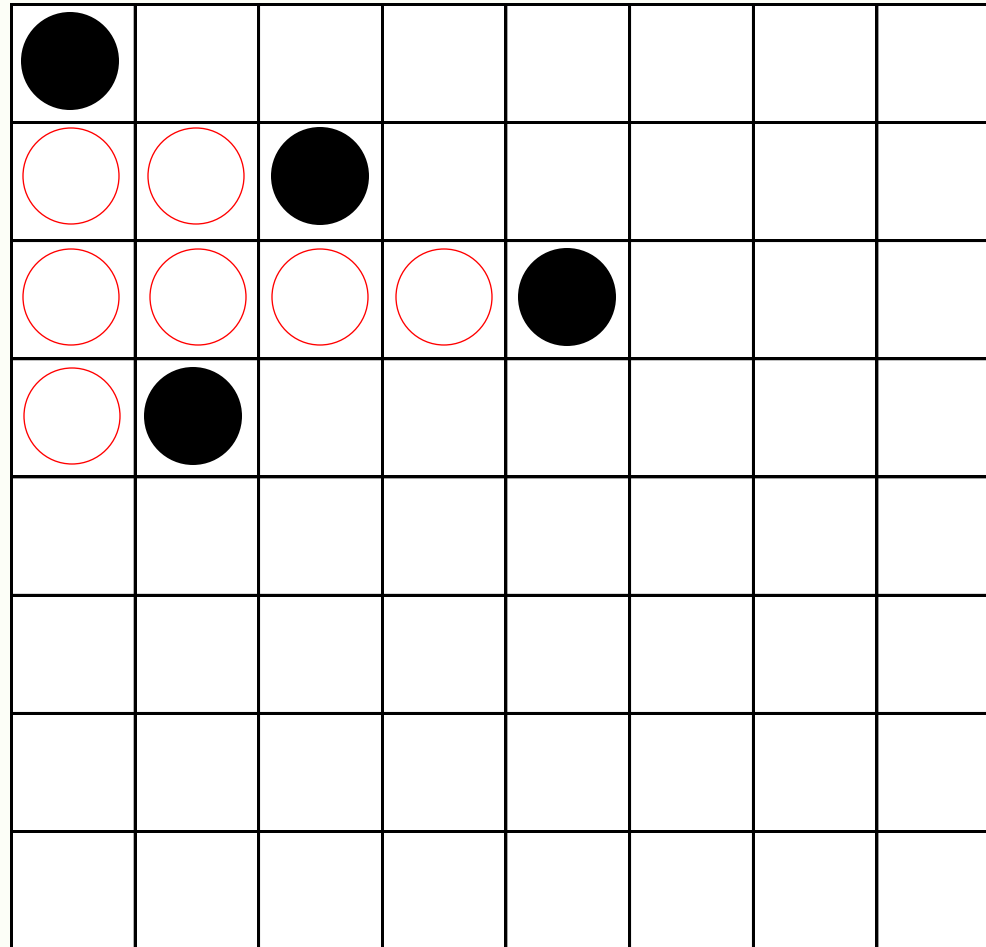


**Tests  $9 + 2 = 11$**

**Backtracks 0**

# Backtracking

---

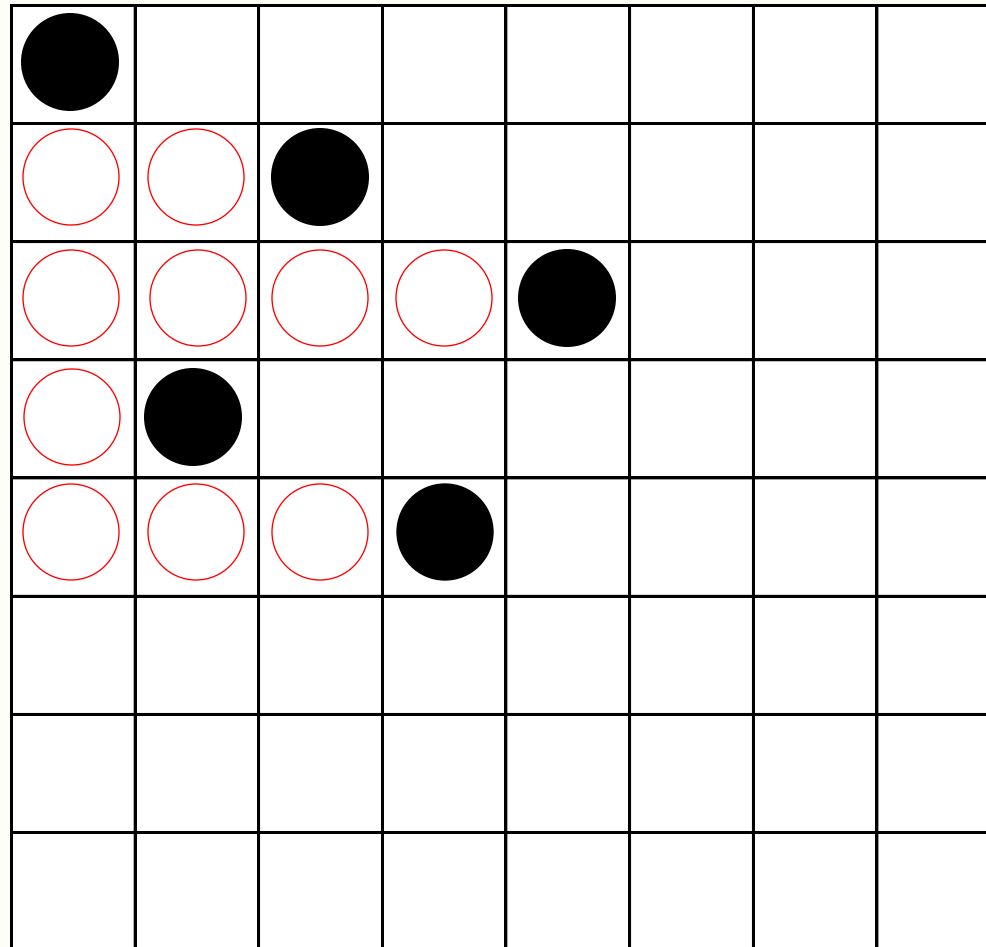


**Tests  $11 + 1 + 3 = 15$**

**Backtracks 0**

# Backtracking

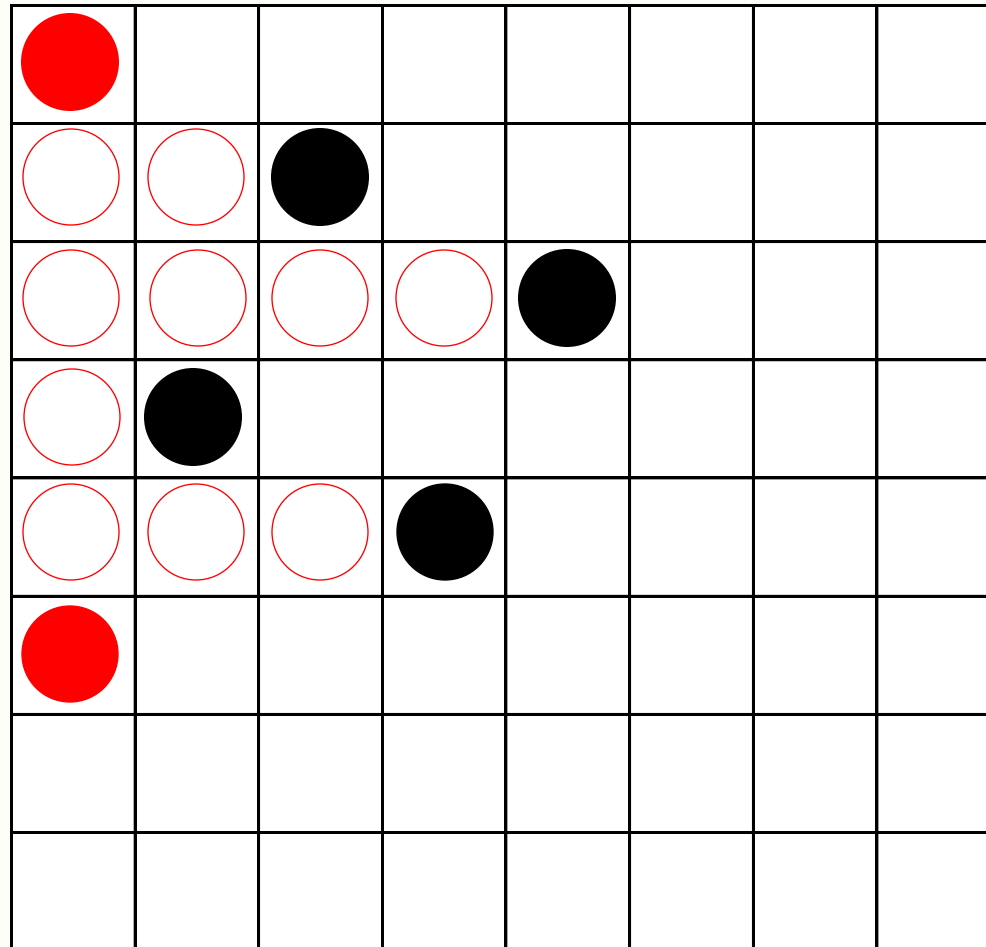
---



**Tests**  $15+1+4+2+4 = 26$       **Backtracks** 0

# Backtracking

---

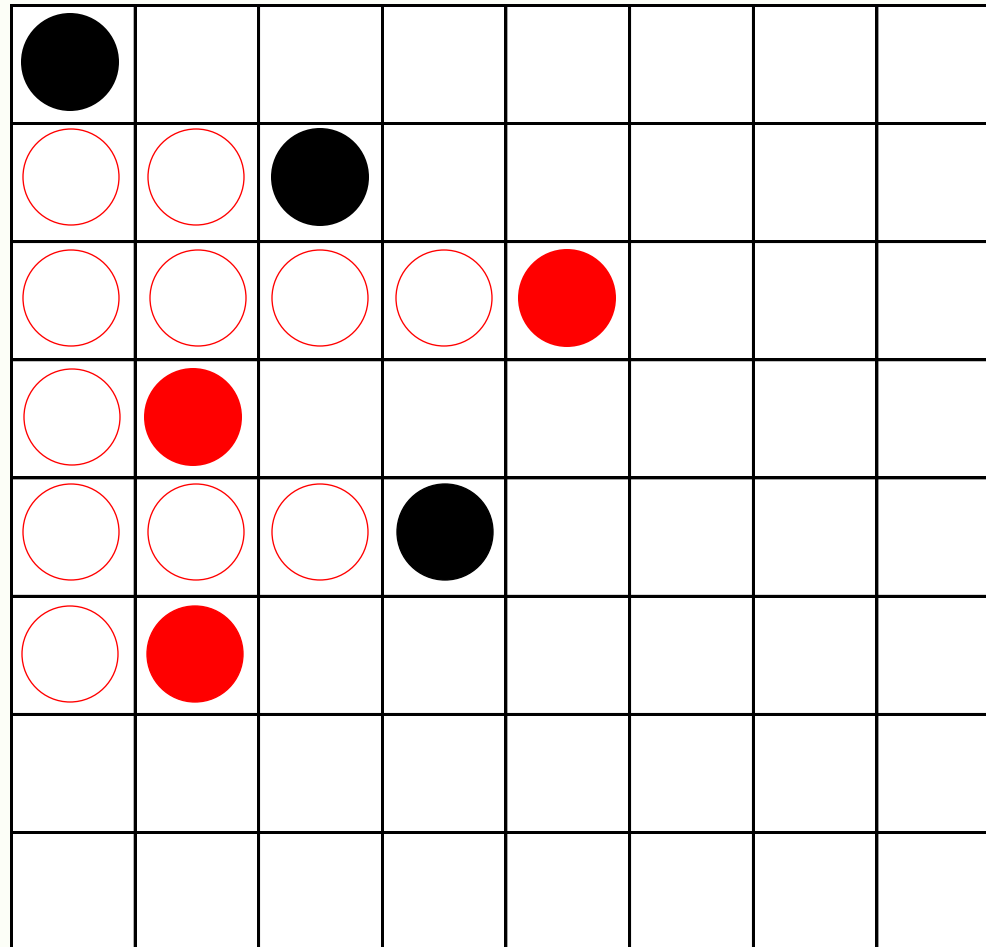


**Tests  $26+1 = 27$**

**Backtracks 0**

# Backtracking

---

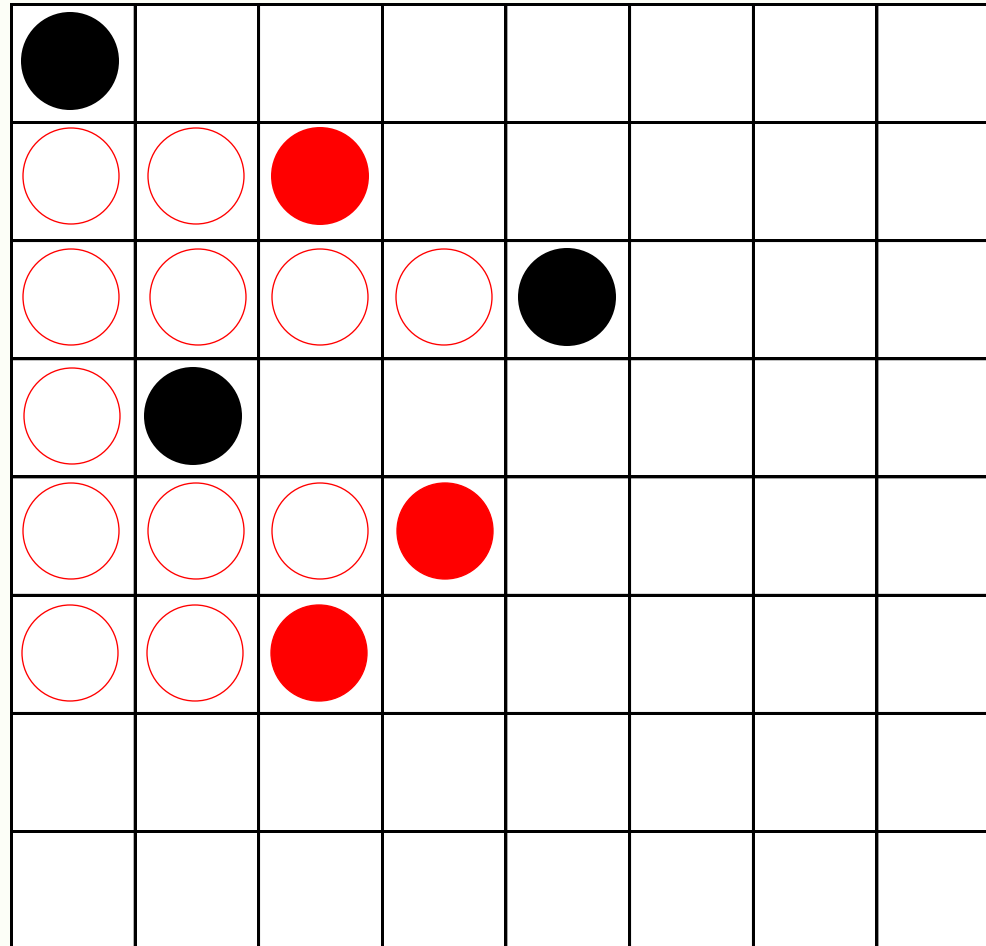


**Tests  $27 + 3 = 30$**

**Backtracks 0**

# Backtracking

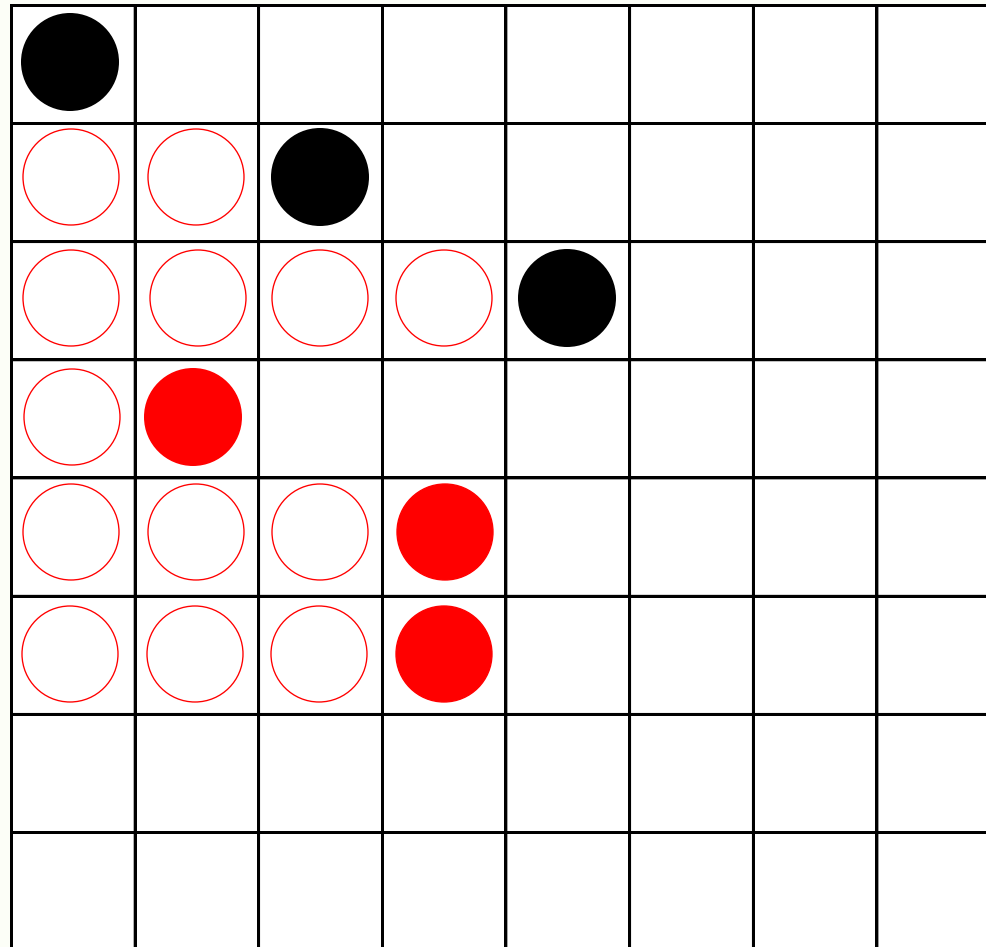
---



**Tests  $30+2 = 32$  Backtracks 0**

# Backtracking

---

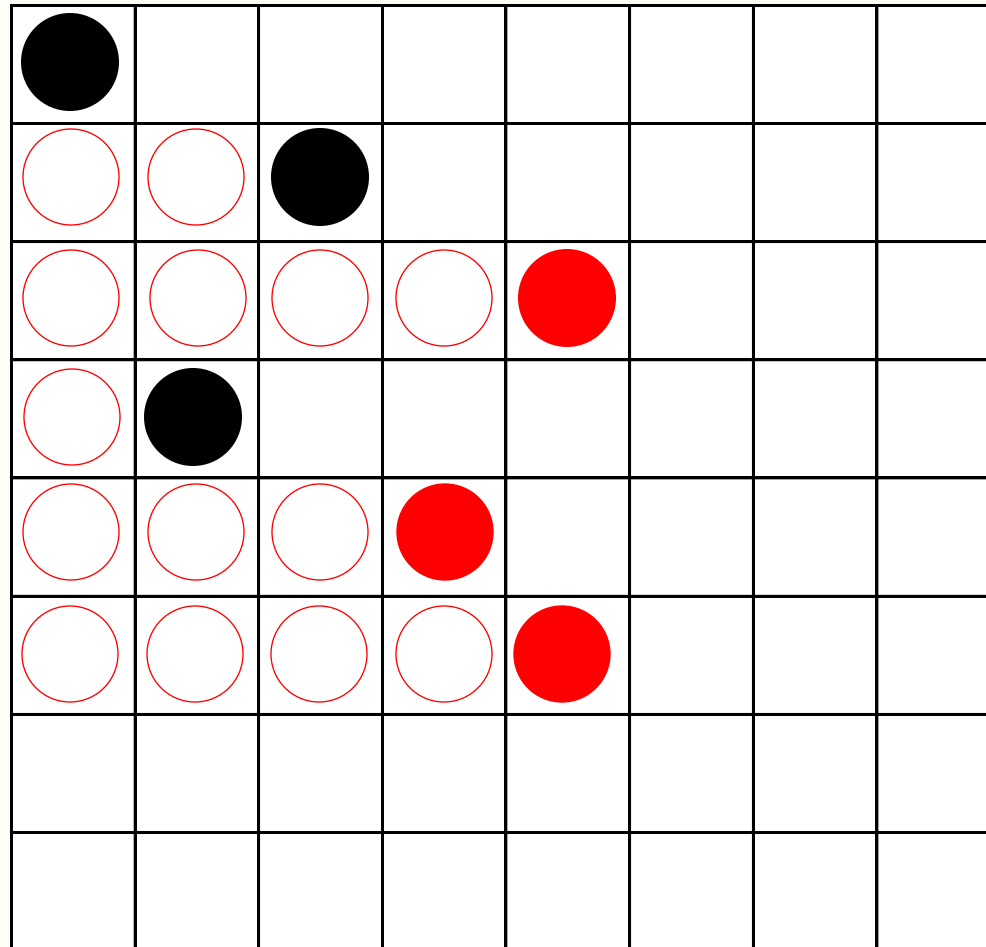


**Tests  $32 + 4 = 36$**

**Backtracks 0**

# Backtracking

---



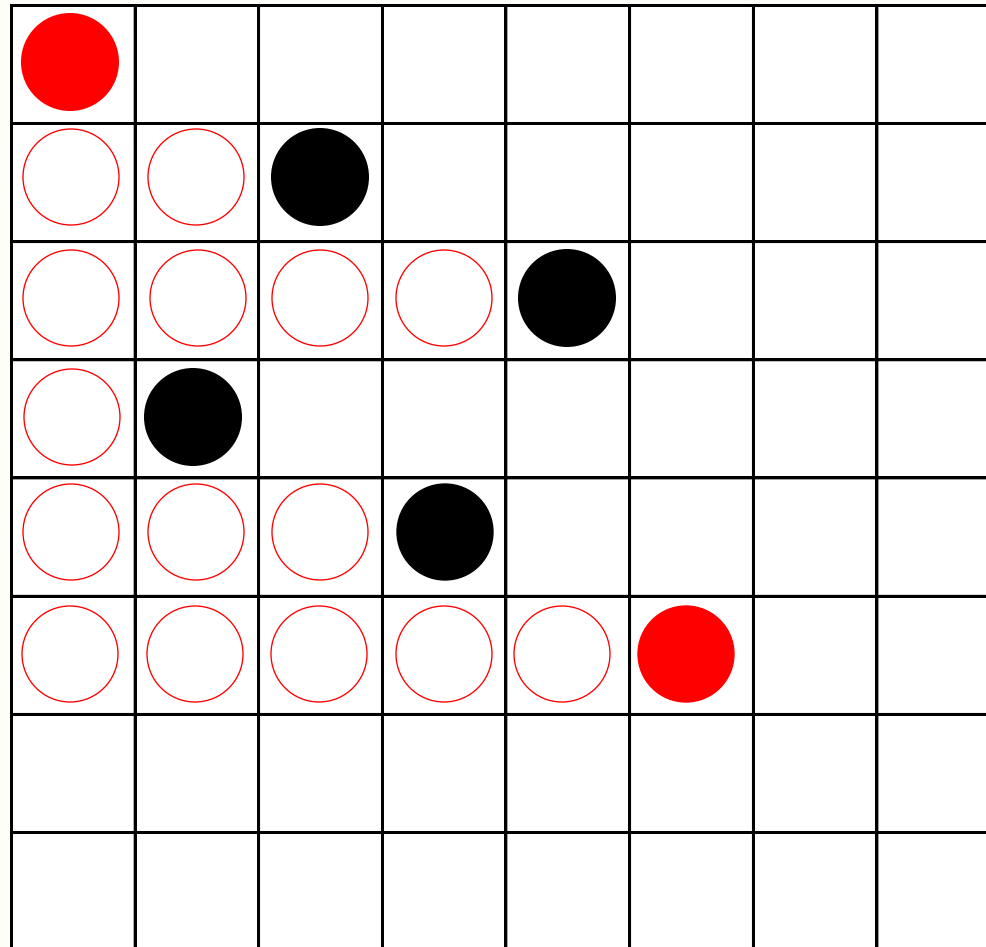
**Tests  $36 + 3 = 39$**

**Backtracks 0**



# Backtracking

---

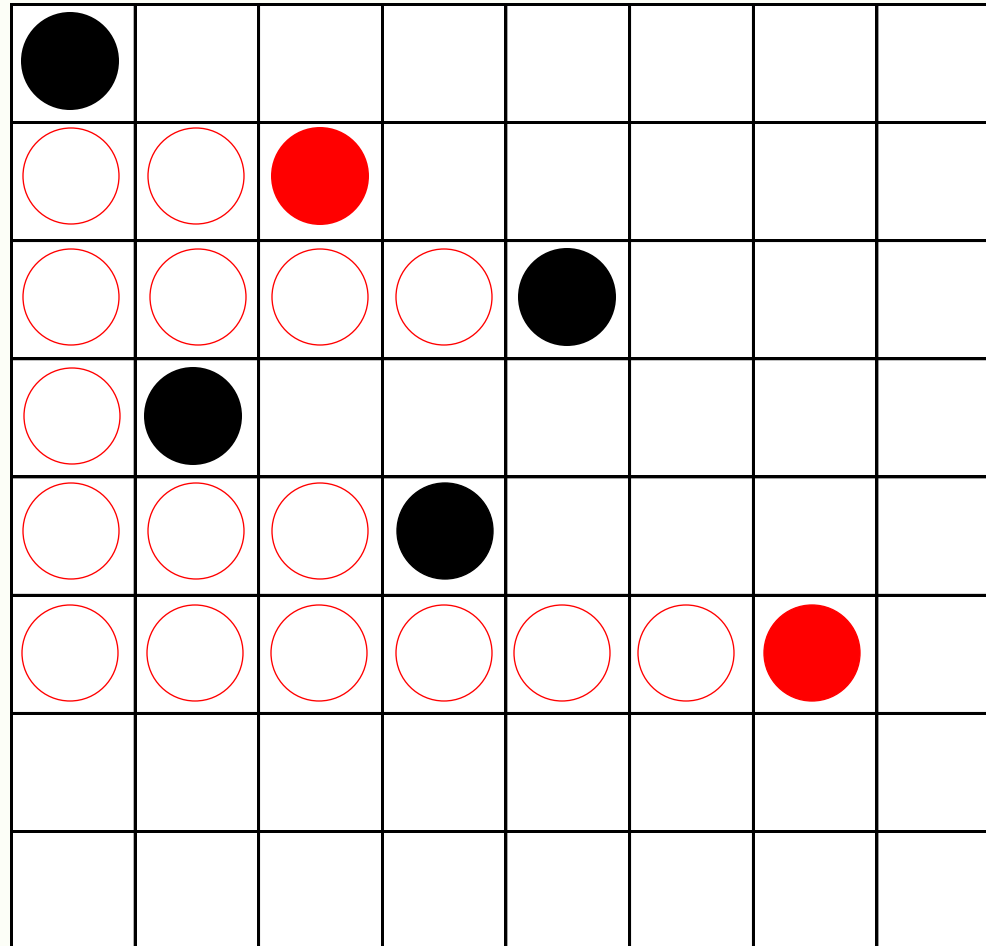


**Tests  $39 + 1 = 40$**

**Backtracks 0**

# Backtracking

---

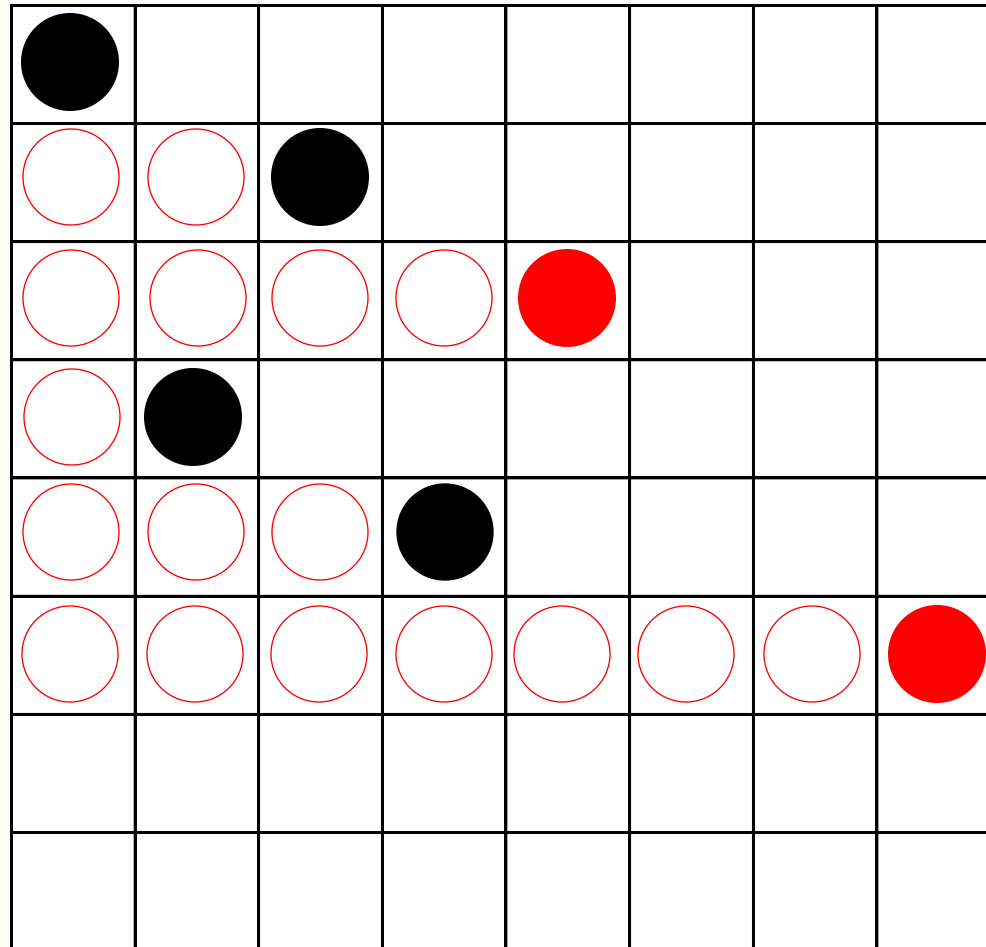


**Tests  $40 + 2 = 42$**

**Backtracks 0**

# Backtracking

---

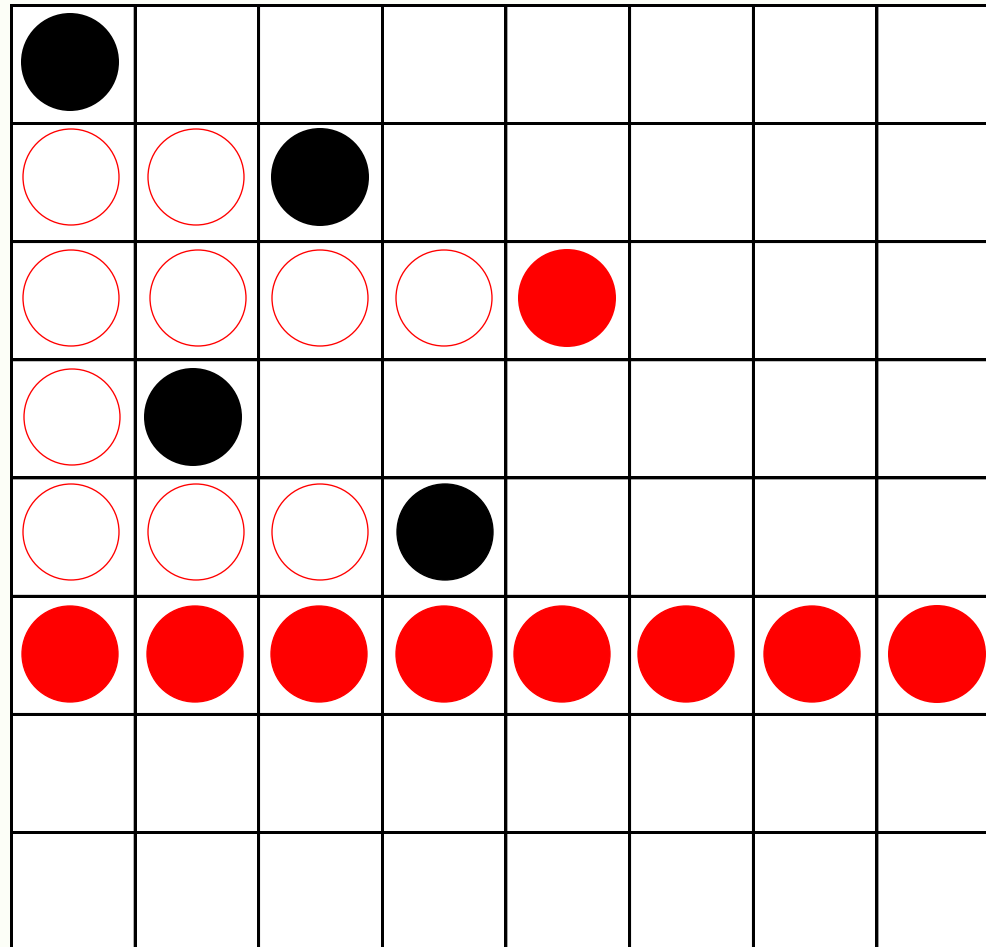


**Tests  $42 + 3 = 45$**

**Backtracks 0**

# Backtracking

**Q6 Fails**  
**Backtracks**  
**to**  
**Q5**

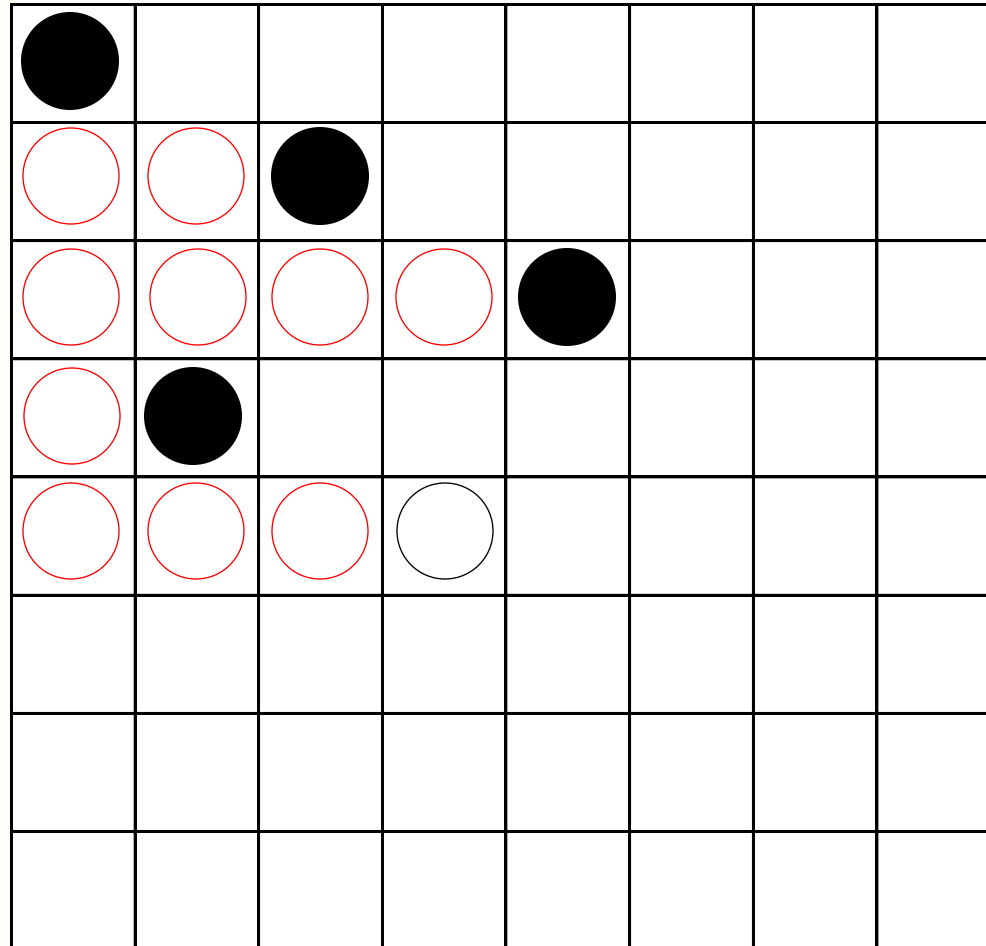


**Tests 45**

**Backtracks  $0+1=1$**

# Backtracking

---

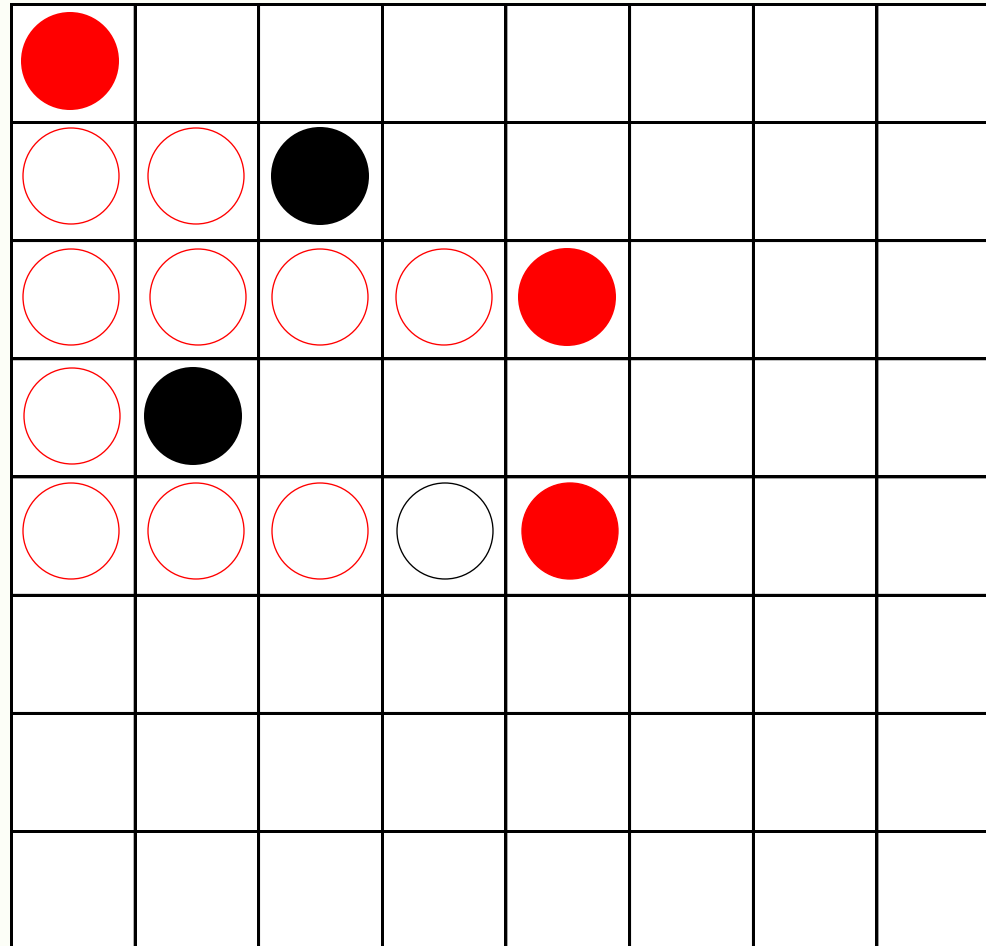


**Tests 45**

**Backtrackings 1**

# Backtracking

---

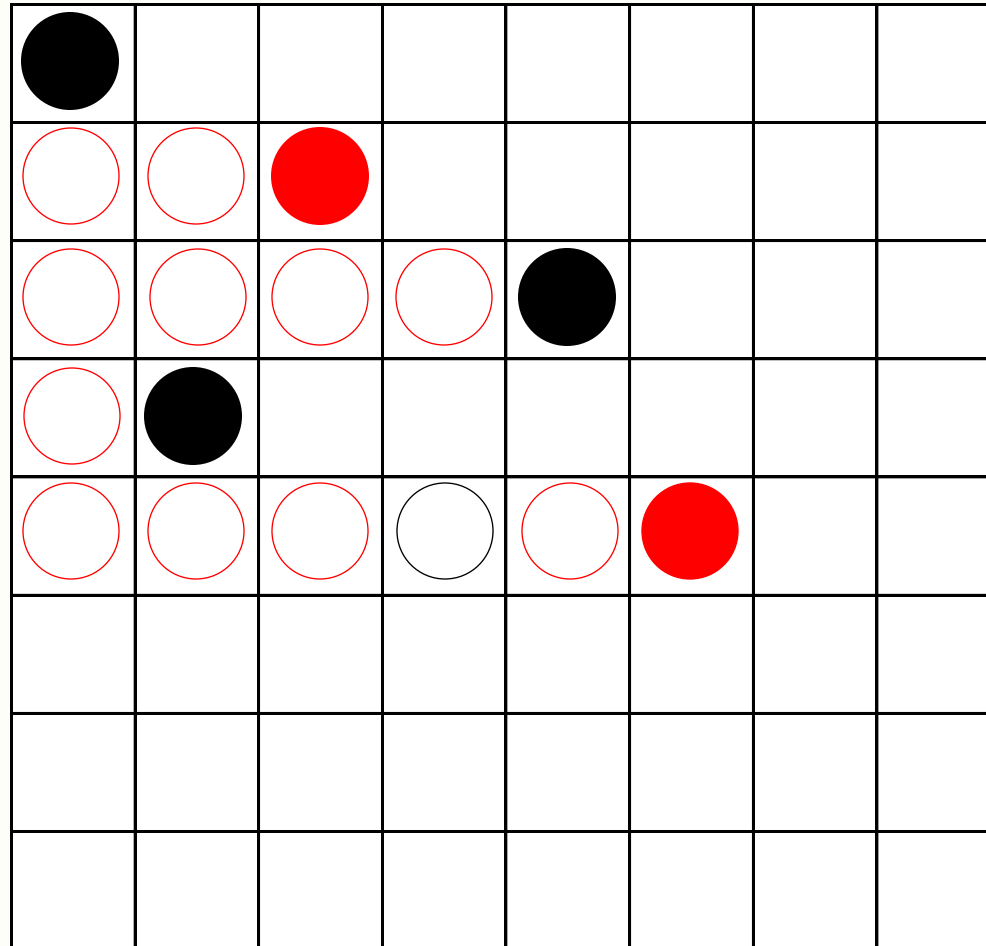


**Tests  $45 + 1 = 46$**

**Backtracks 1**

# Backtracking

---

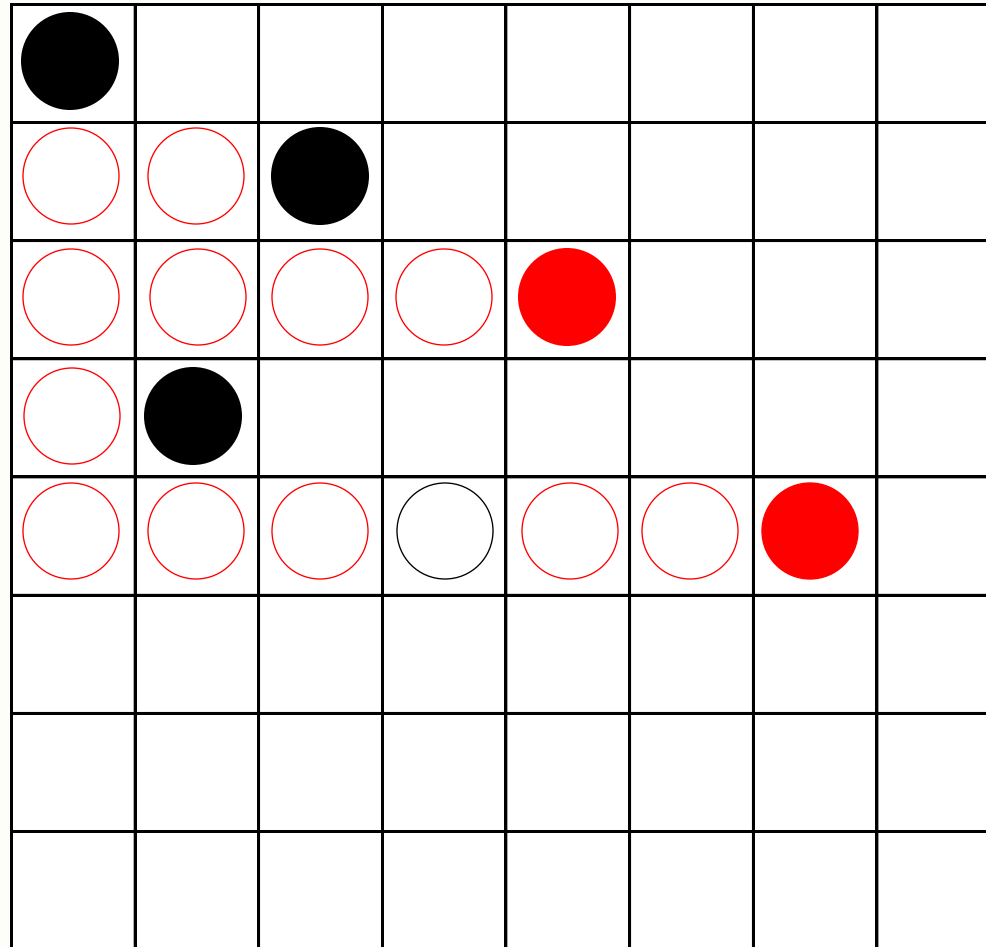


**Tests  $46 + 2 = 48$**

**Backtracks 1**

# Backtracking

---



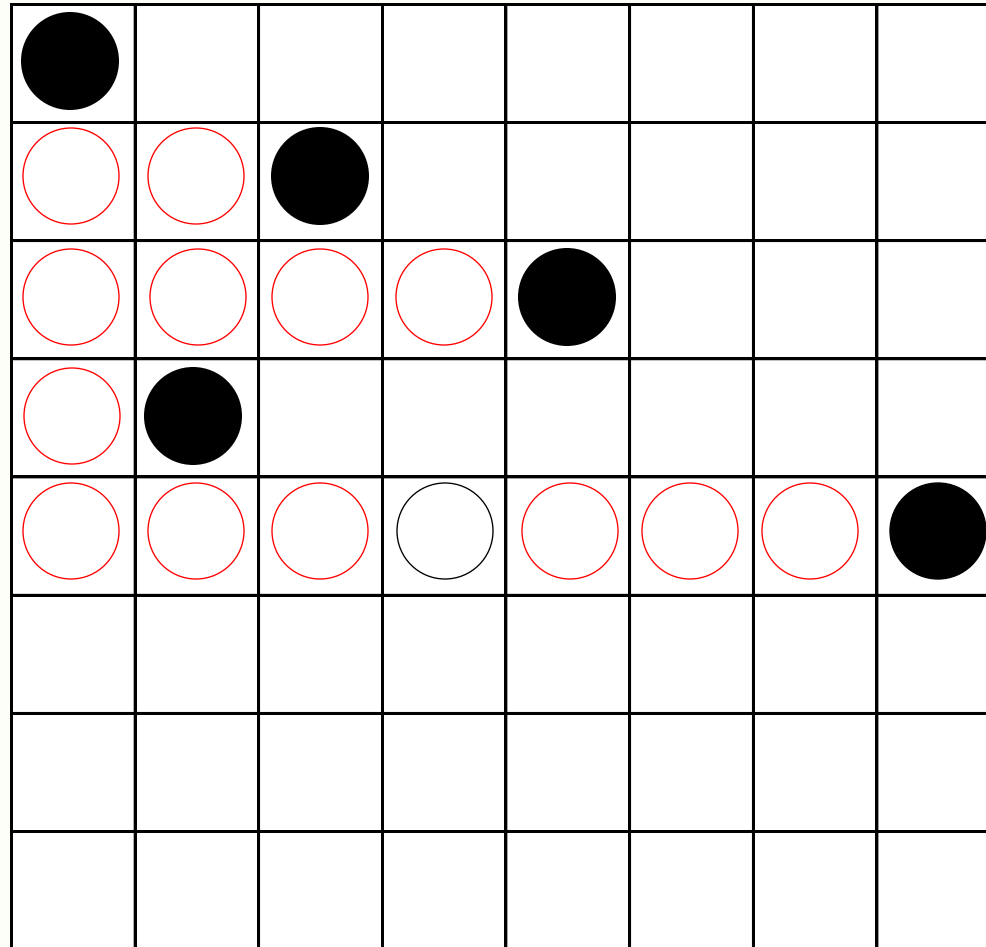
**Tests  $48 + 3 = 51$**

**Backtracks 1**



# Backtracking

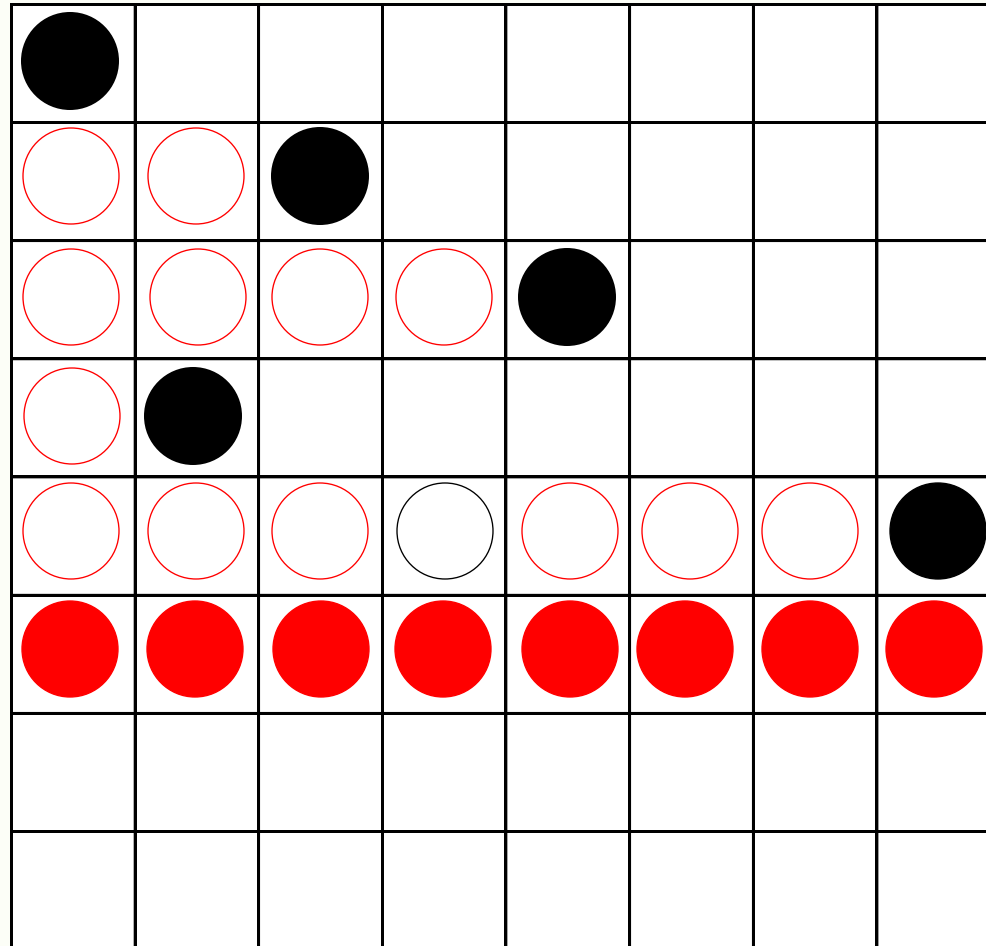
---



**Tests**  $51 + 4 = 55$

**Backtracks** 1

# Backtracking



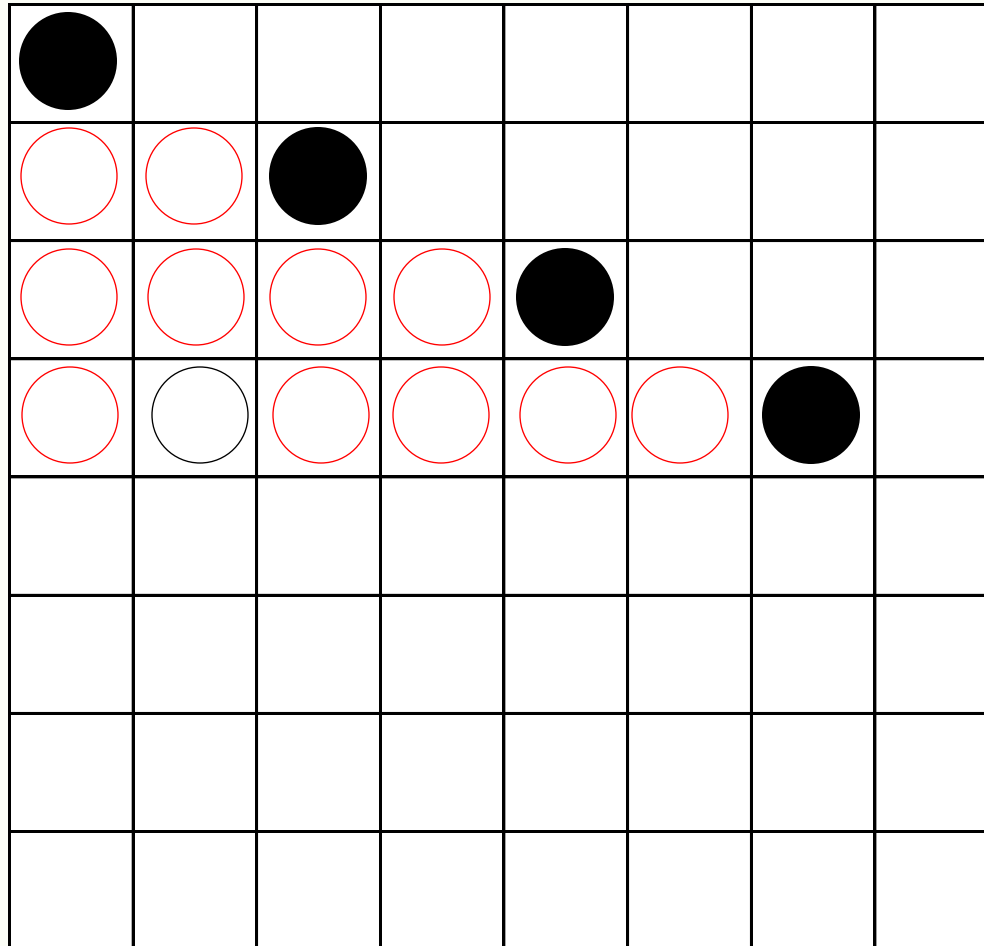
Q6 Fails  
 Backtracks  
 to  
 Q5  
 and next to  
 Q4

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

Backtracks  $1+2 = 3$

# Backtracking

---

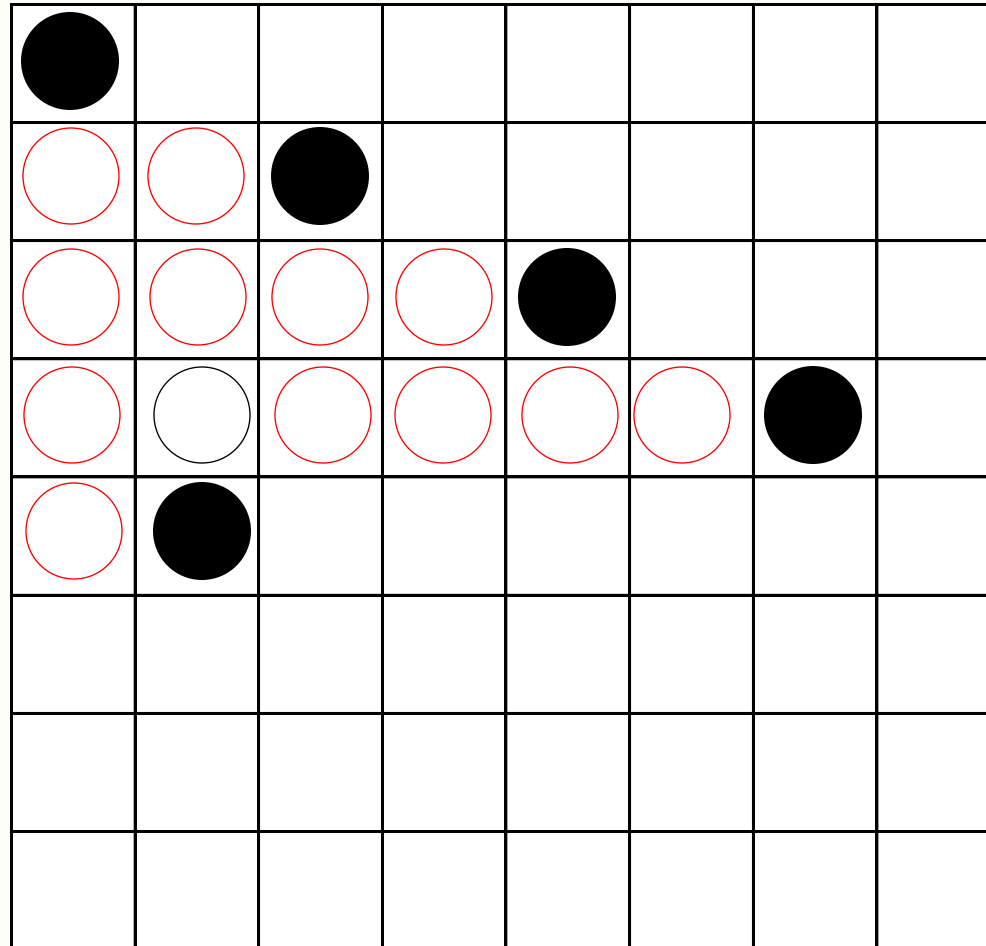


**Tests  $74+2+1+2+3+3= 85$**

**Backtracks 3**

# Backtracking

---

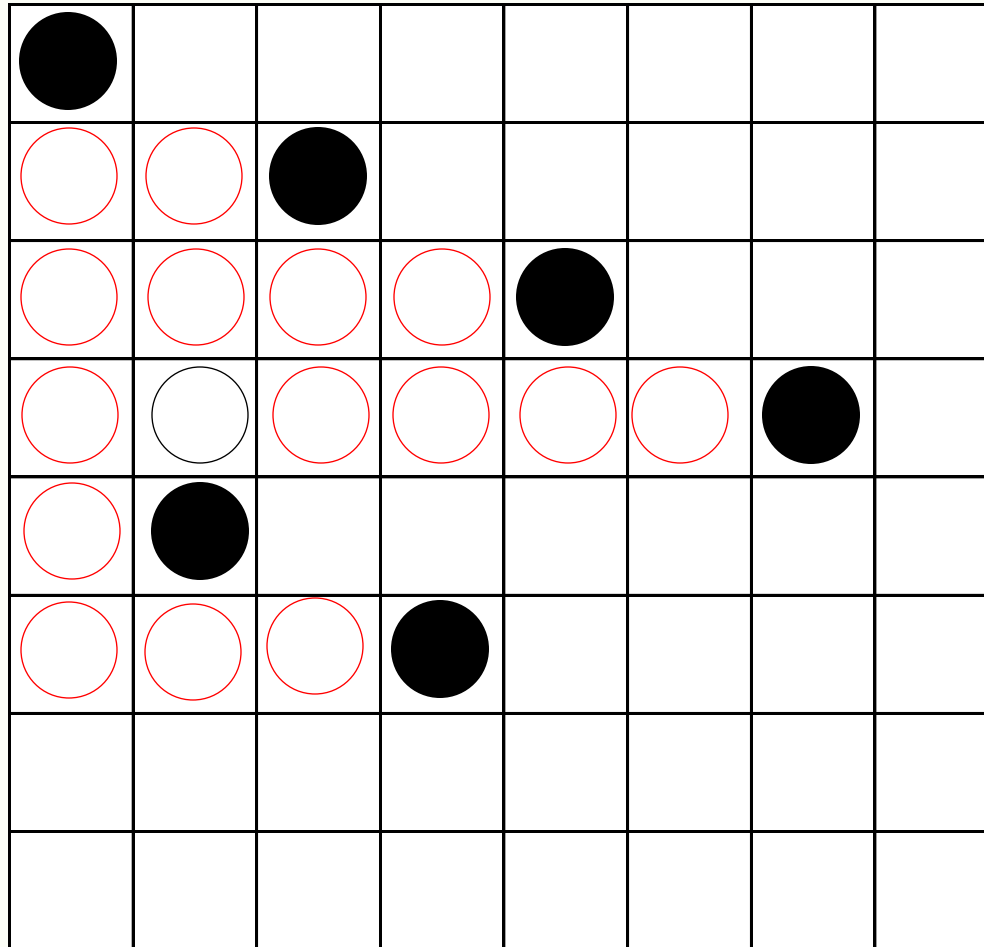


**Tests  $85 + 1 + 4 = 90$**

**Backtracks 3**

# Backtracking

---

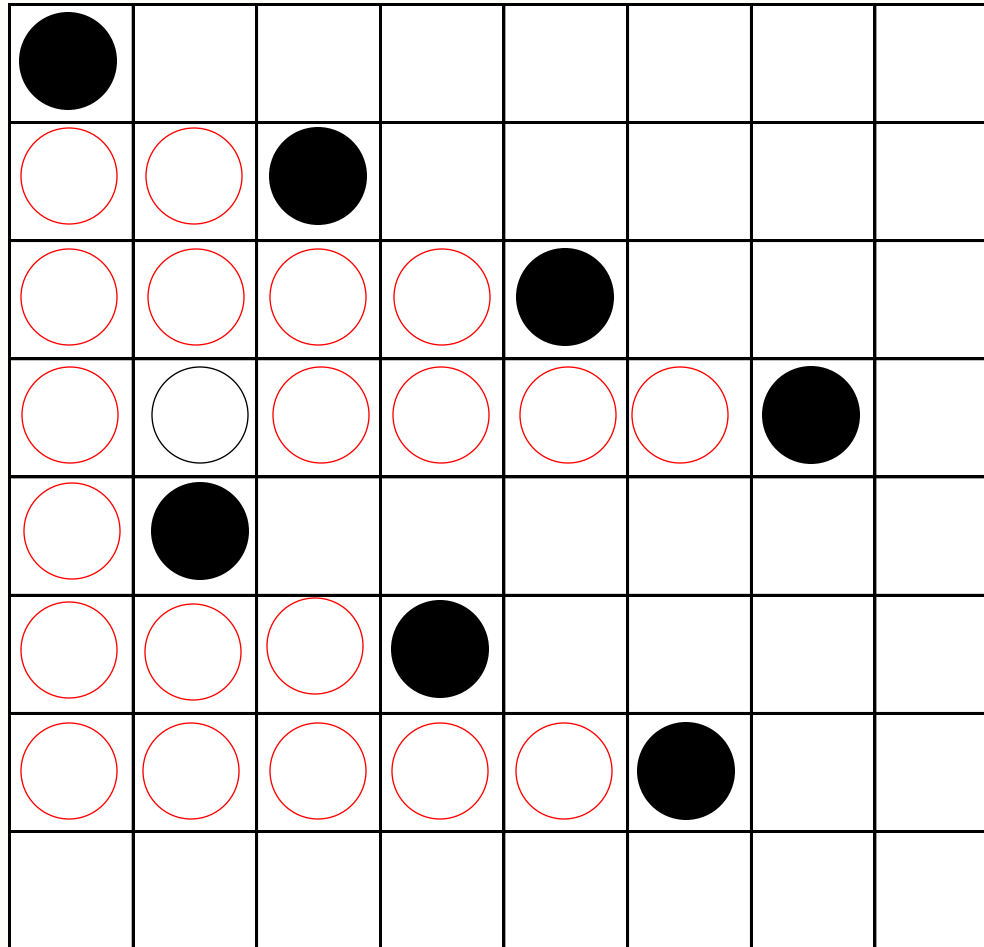


**Tests**  $90 + 1 + 3 + 2 + 5 = 101$

**Backtracks** 3

# Backtracking

---

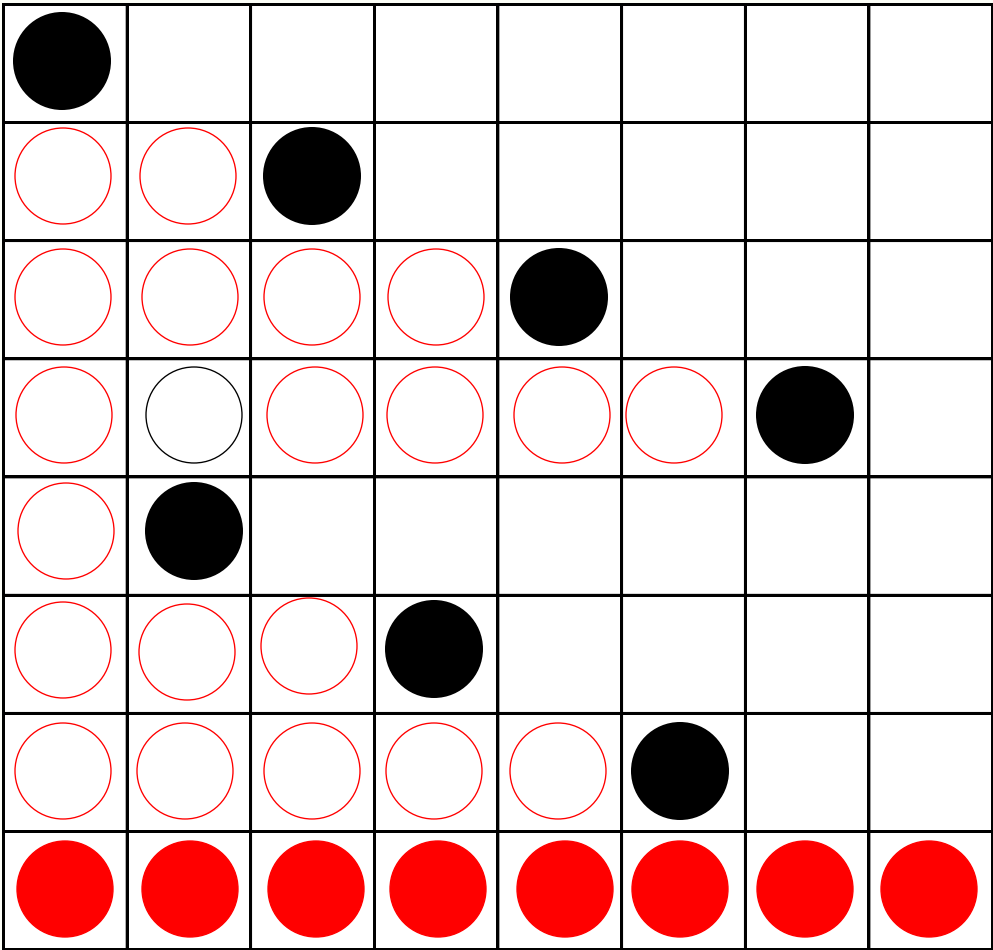


**Tests**  $101+1+5+2+4+3+6= 122$

**Backtracks** 3

# Backtracking

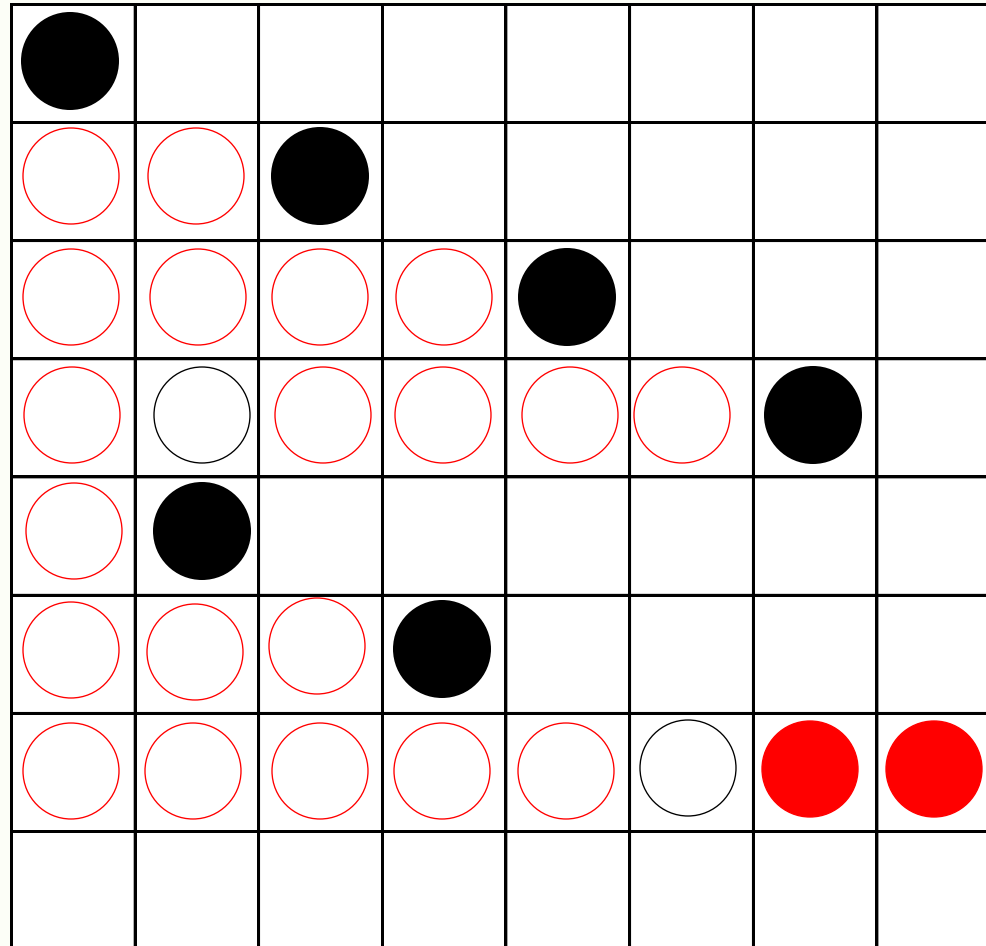
Q8 Fails  
Backtracks  
to  
Q7



Tests  $122+1+5+2+6+3+6+4+1= 150$  Backtracks  $3+1=4$

# Backtracking

**Q7 Fails**  
**Backtracks**  
**to**  
**Q6**



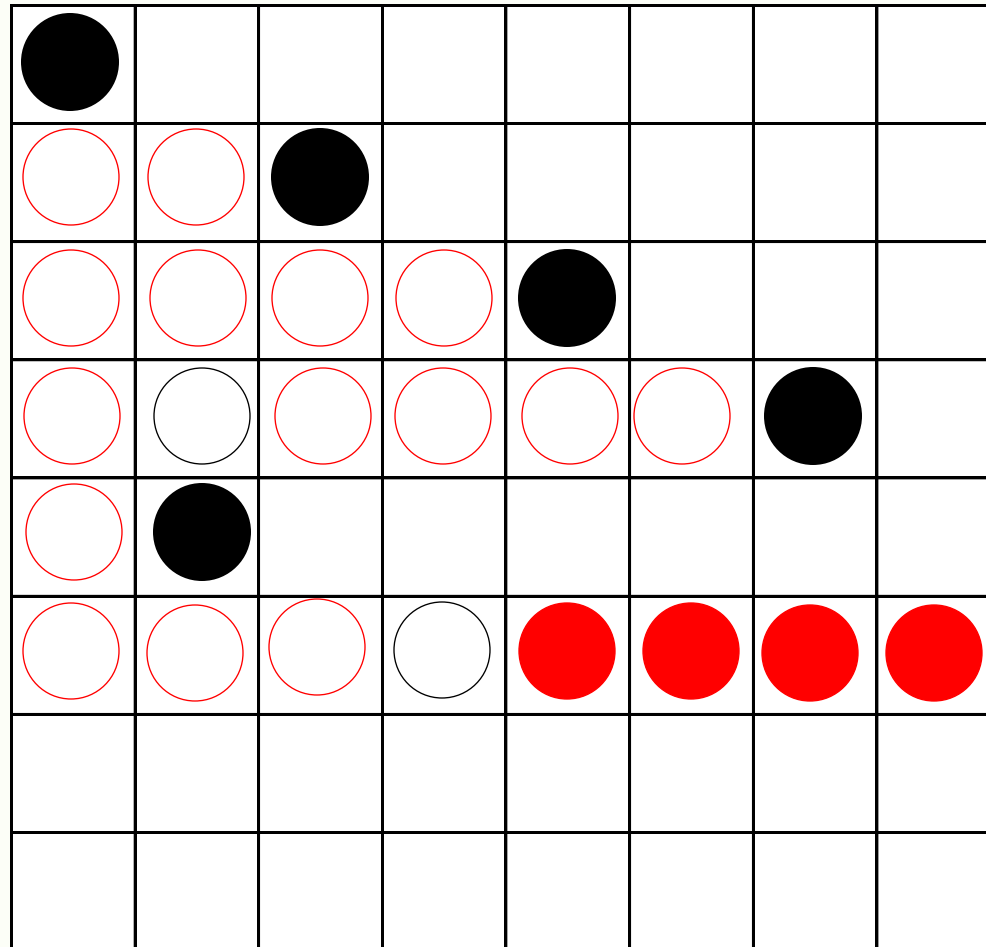
**Tests**  $150+1+2= 153$

**Backtracks**  $4+1=5$



# Backtracking

Q6 Fails  
Backtracks  
to  
Q5

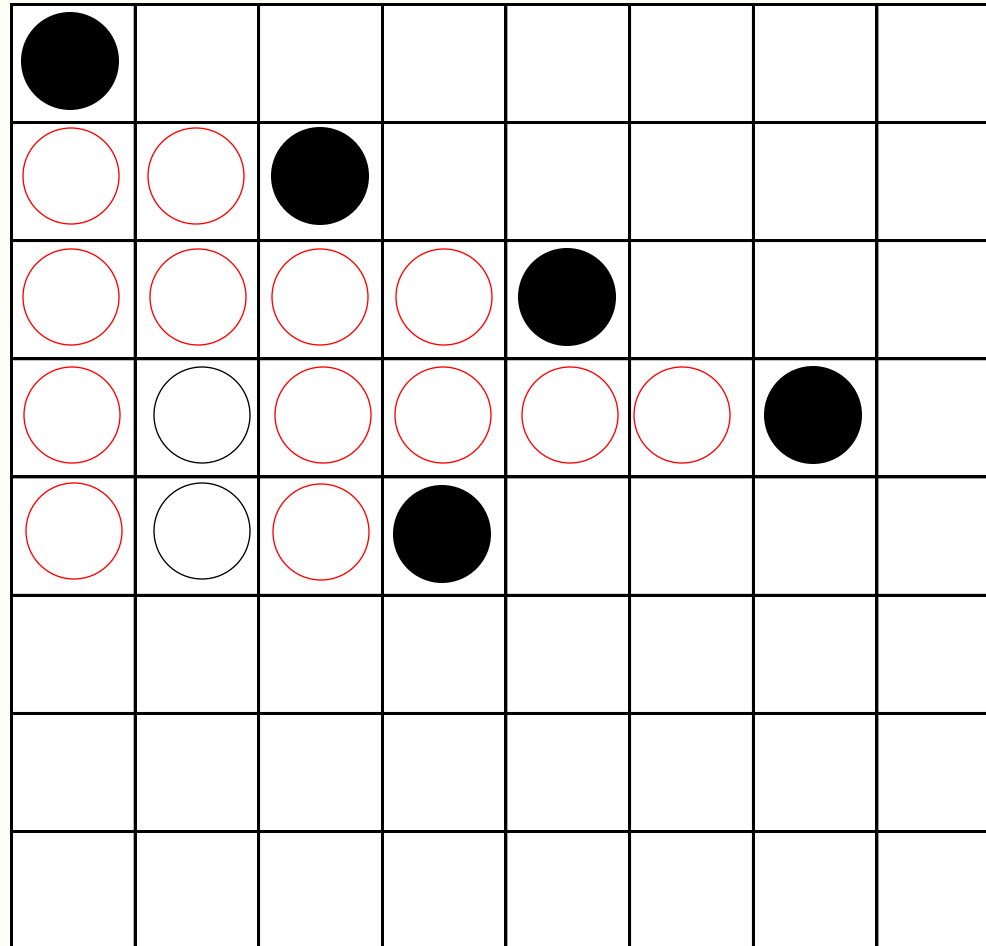


Tests  $153+3+1+2+3= 162$

Backtracks  $5+1=6$

# Backtracking

---

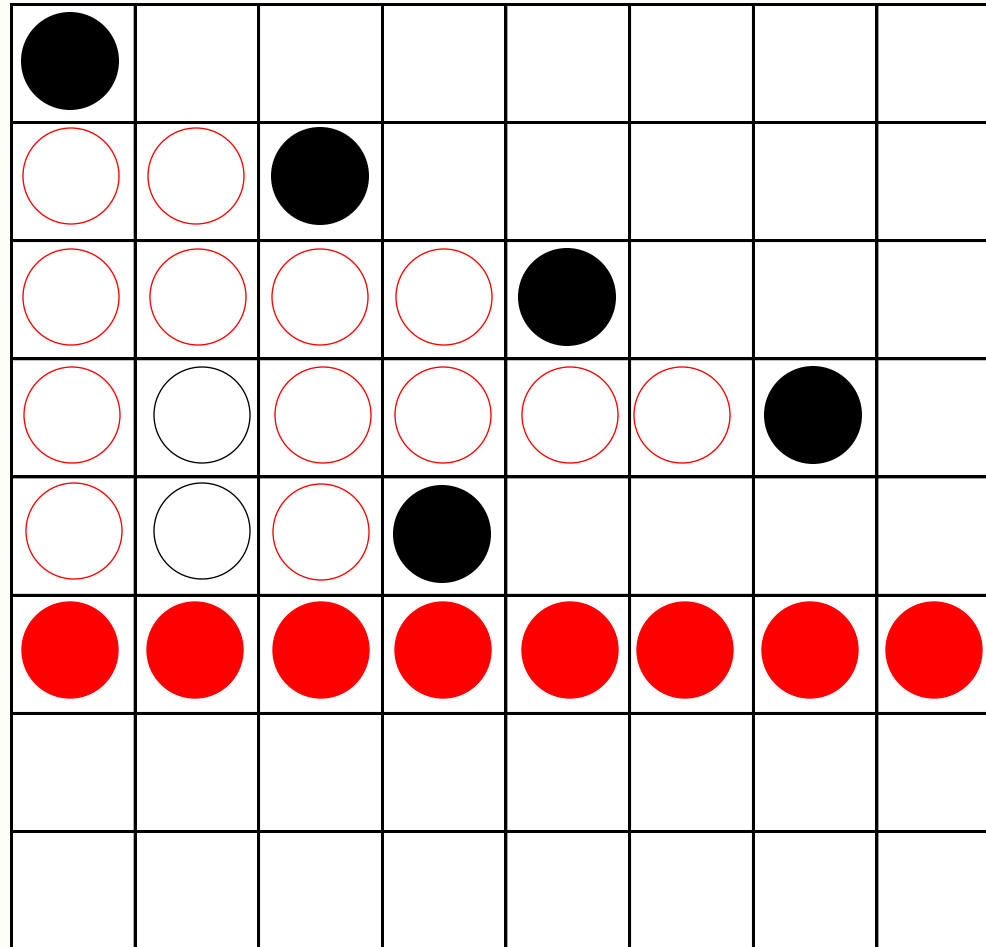


**Tests  $162+2+4= 168$**

**Backtracks 6**

# Backtracking

Q6 Fails  
Backtracks  
to  
Q5

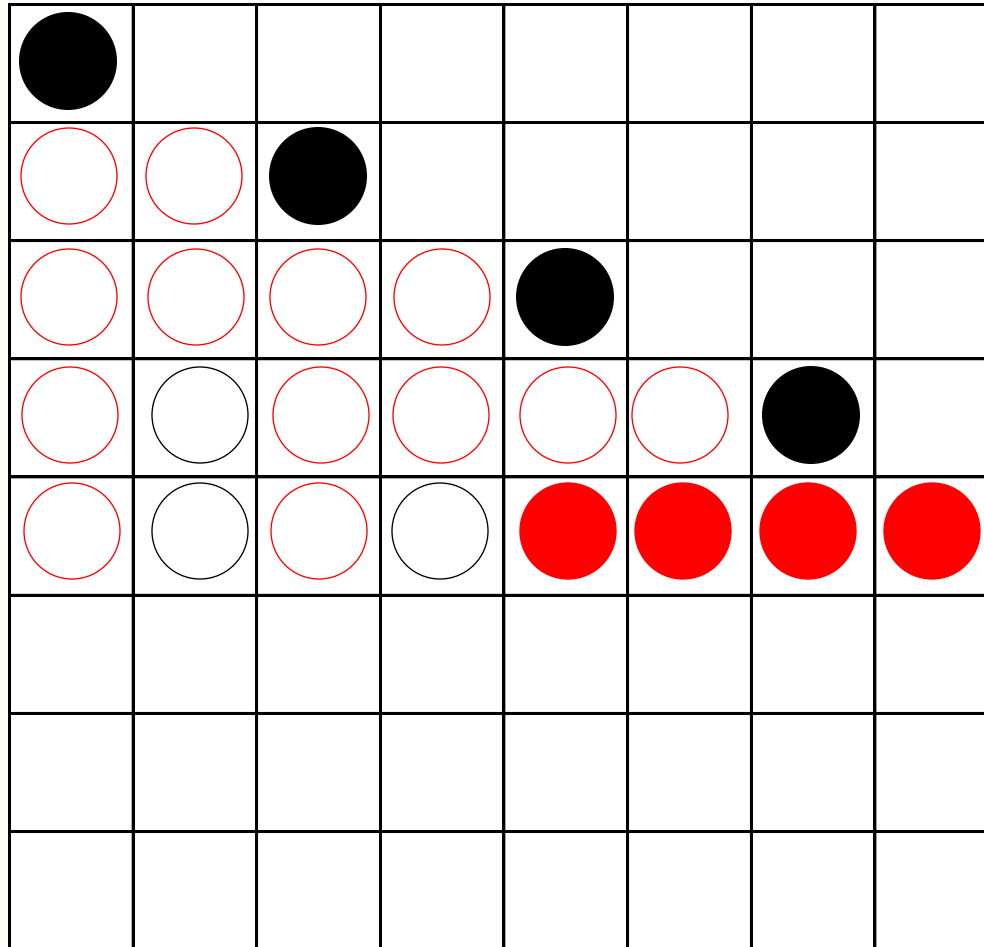


Tests  $168+1+3+2+5+3+1+2+3= 188$

Backtracks  $6+1 = 7$

# Backtracking

**Q5 Fails**  
**Backtracks**  
**to**  
**Q4**

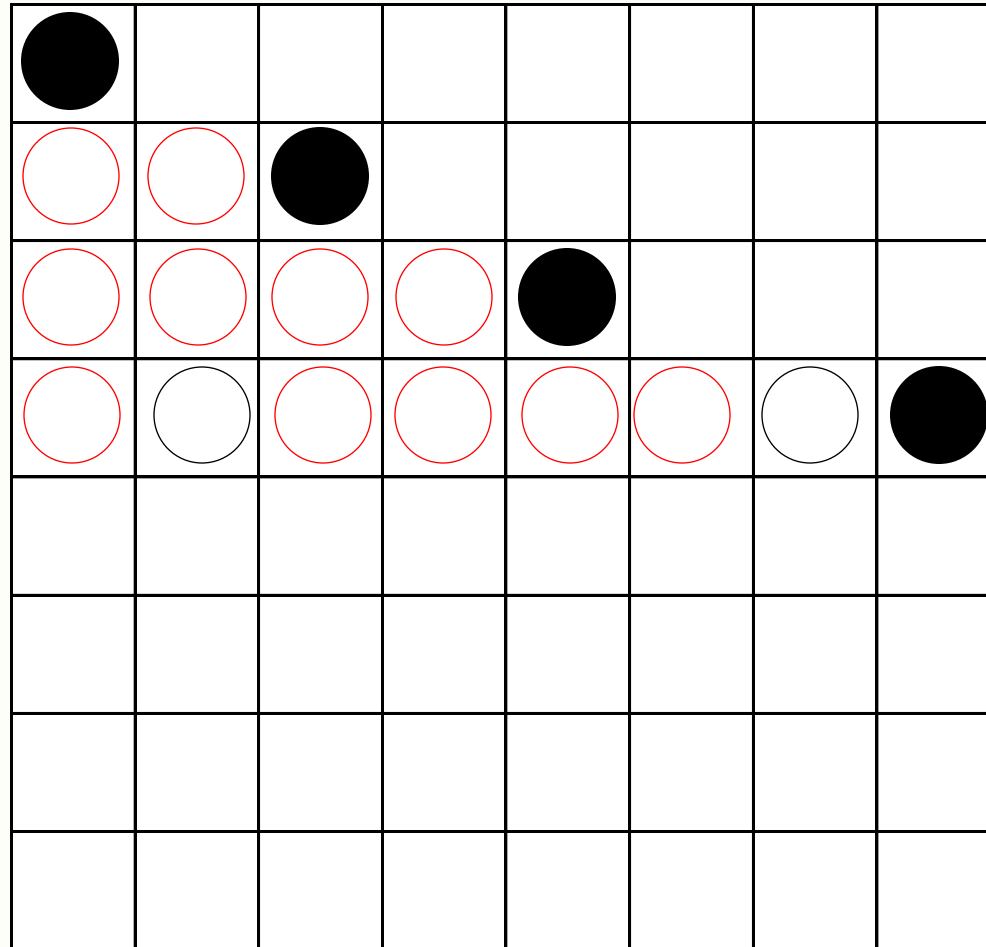


**Tests**  $188+1+2+3+4= 198$

**Backtracks**  $7+1=8$

# Backtracking

---

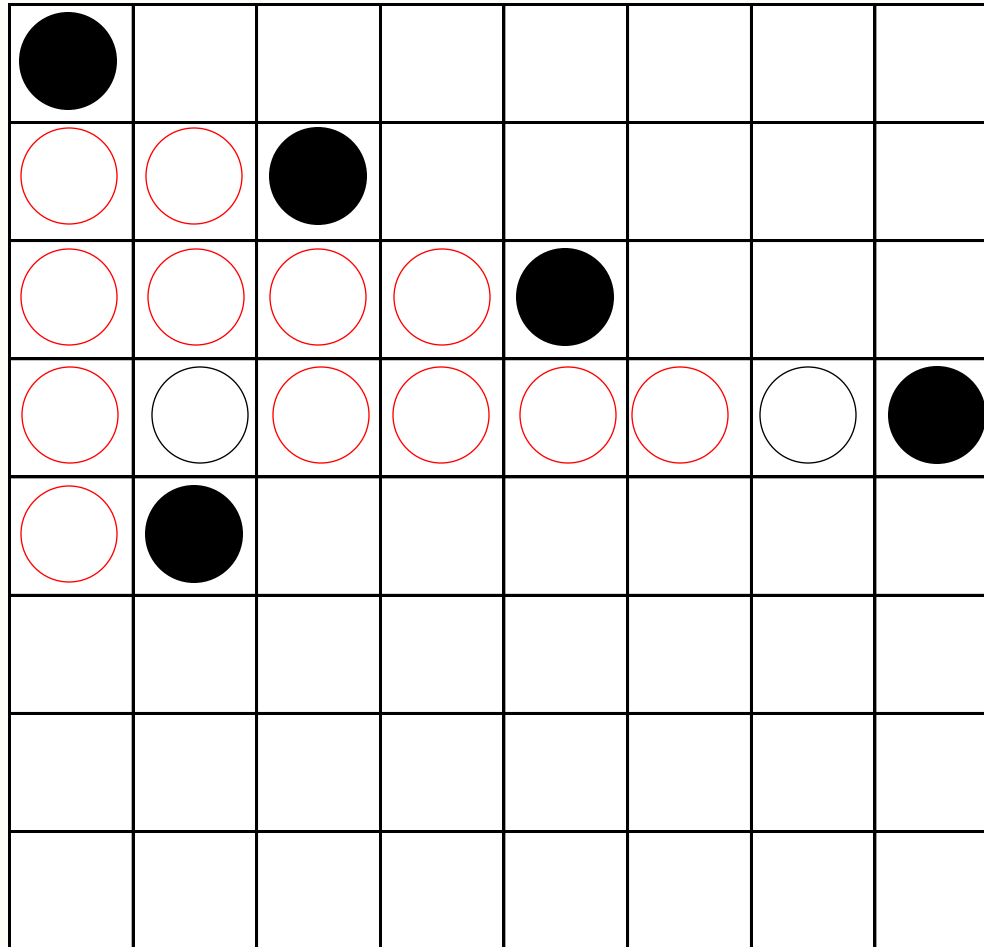


**Tests  $198 + 3 = 201$**

**Backtracks 8**

# Backtracking

---

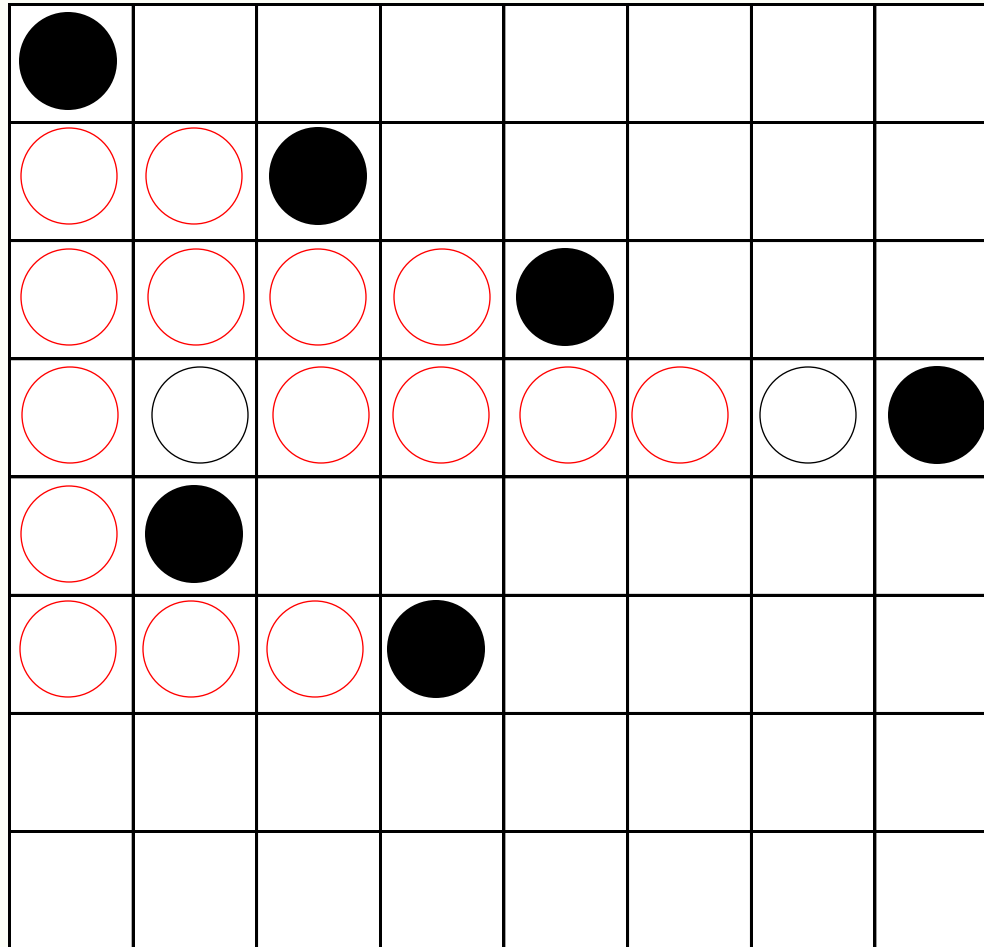


**Tests  $201+1+4 = 206$**

**Backtracks 8**

# Backtracking

---

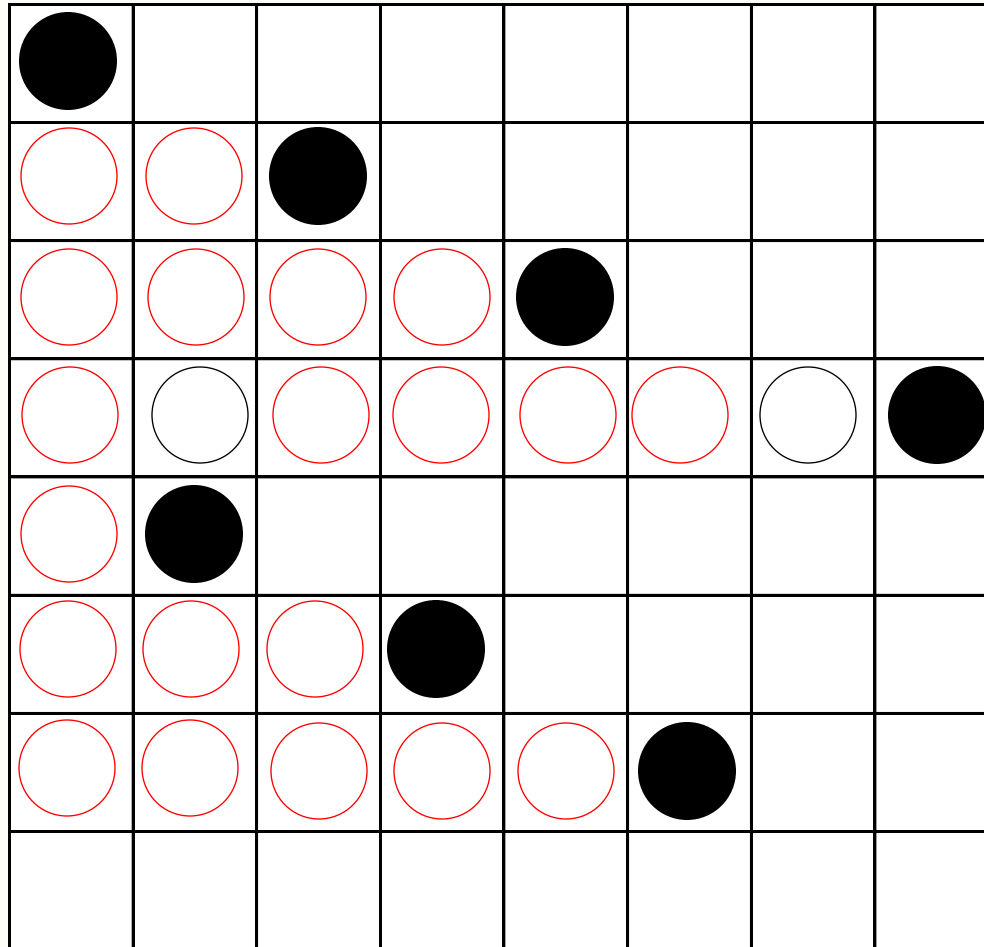


**Tests**  $206+1+3+2+5 = 217$

**Backtracks** 8

# Backtracking

---



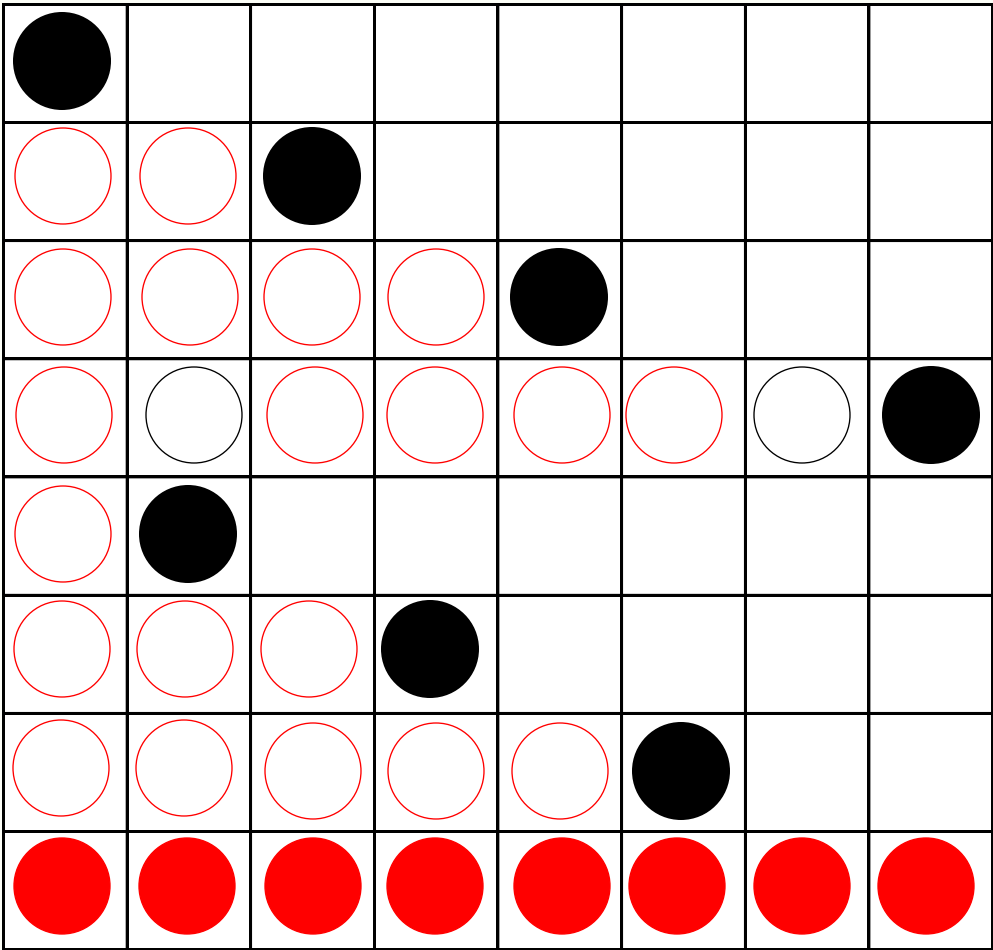
**Tests**  $217+1+5+2+5+3+6 = 239$

**Backtracks** 8



# Backtracking

Q8 Fails  
Backtracks  
to  
Q7



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

Backtracks  $8+1 = 9$

# Backtracking

●							
○	○	●					
○	○	○	○	●			
○	○	○	○	○	○	○	●
○	●						
○	○	○	●				
○	○	○	○	○	○	●	●

**Q7 Fails**  
**Backtracks**  
**to**  
**Q6**

**Tests  $274+1+2= 277$**

**Backtracks  $9+1=10$**

# Backtracking

Q6 Fails  
Backtracks  
to  
Q5

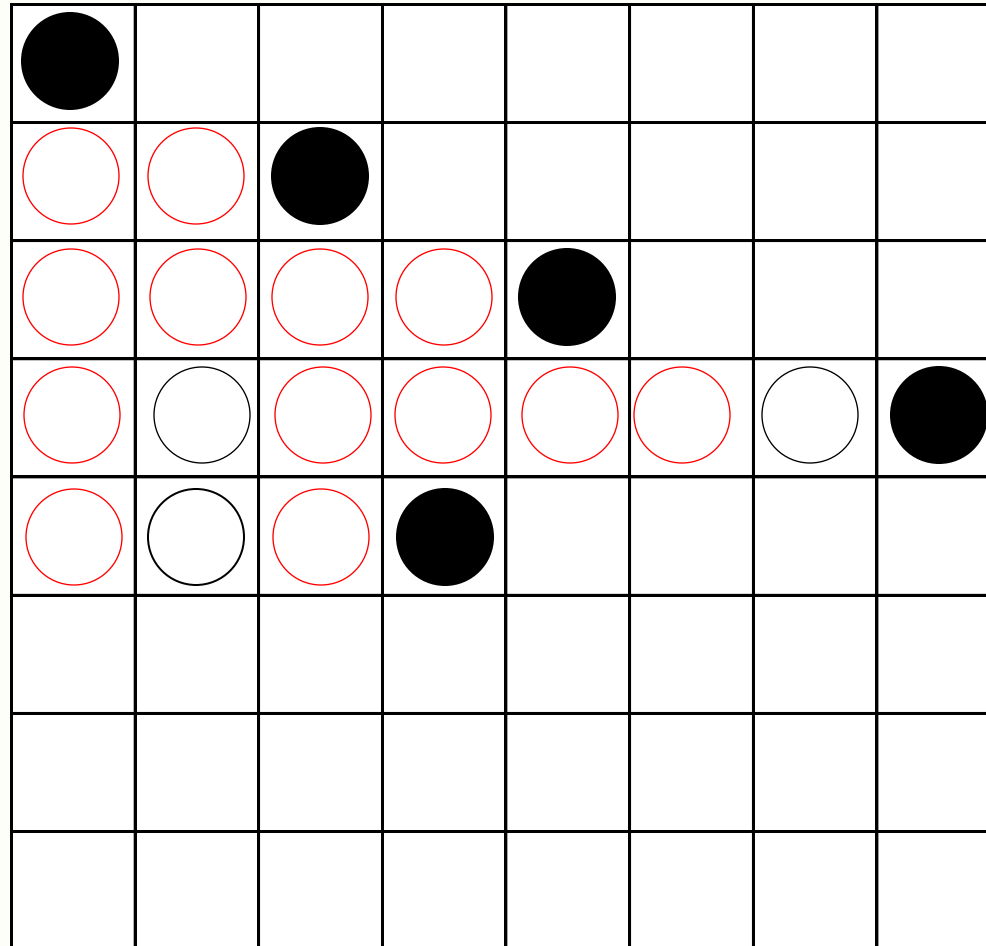
●							
○	○	●					
○	○	○	○	●			
○	○	○	○	○	○	○	●
○	●						
○	○	○	○	●	●	●	●

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

Backtracks  $10+1=11$

# Backtracking

---

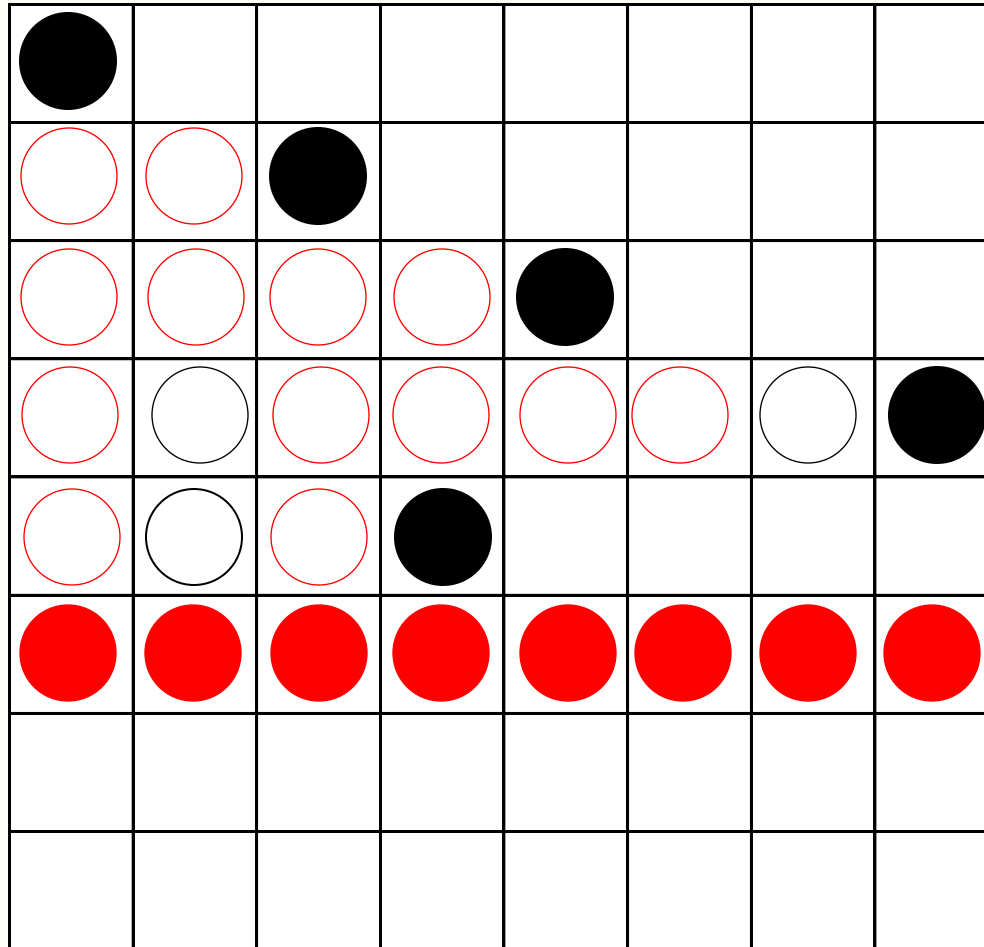


**Tests  $286+2+4= 292$**

**Backtracks 11**

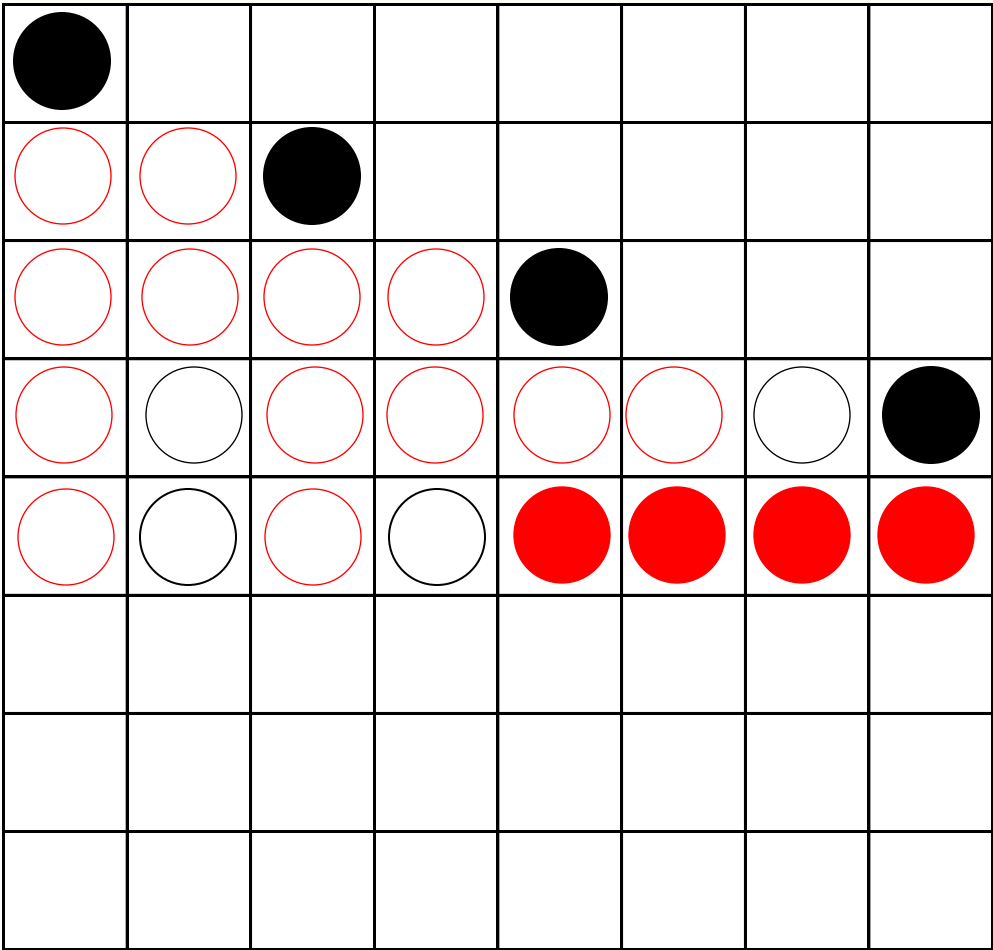
# Backtracking

Q6 Fails  
Backtracks  
to  
Q5



Tests  $292+1+3+2+5+3+1+2+3= 312$  Backtracks  $11+1=12$

# Backtracking

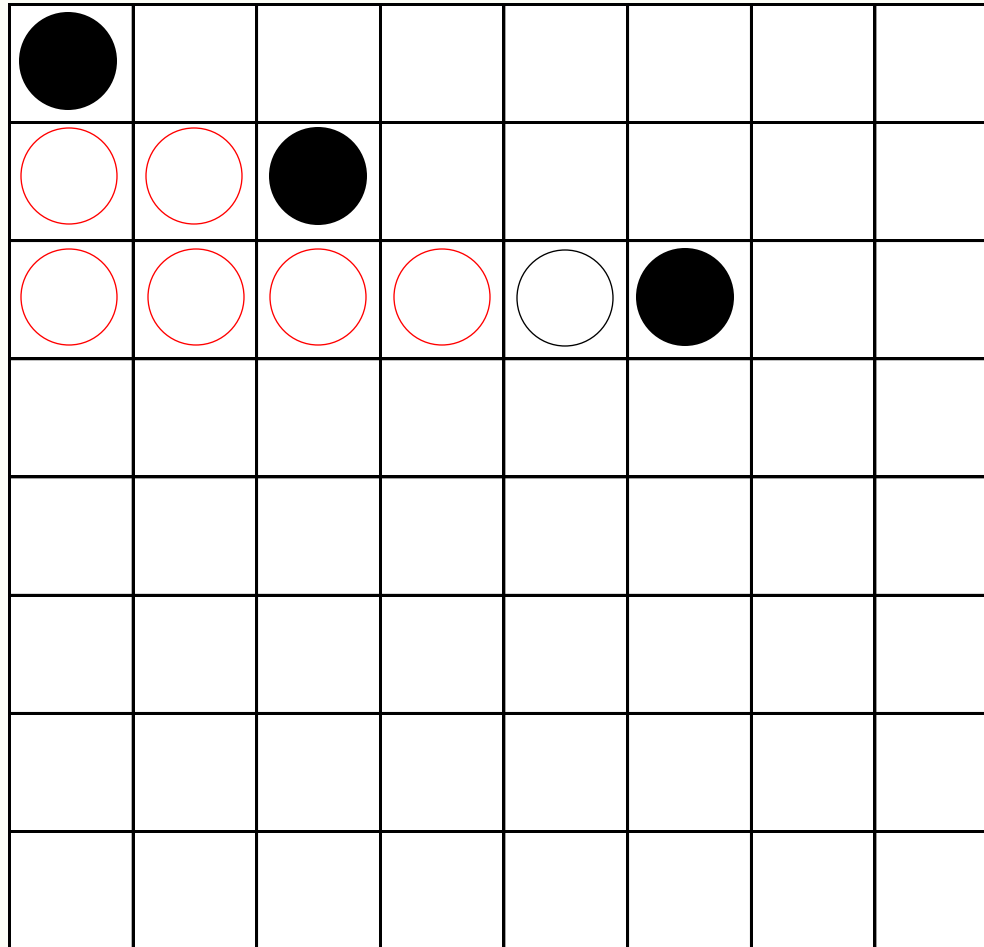


Q5 Fails  
 Backtracks  
 to  
 Q4  
 and next to  
 Q3

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

Backtracks  $12+2=14$

# Backtracking



$Q_1 = 1$

$Q_2 = 3$

$Q_3 = 5$

Impossible !

Tests  $322 + 2 = 324$

Backtracks 14

# Backtracking + Propagation

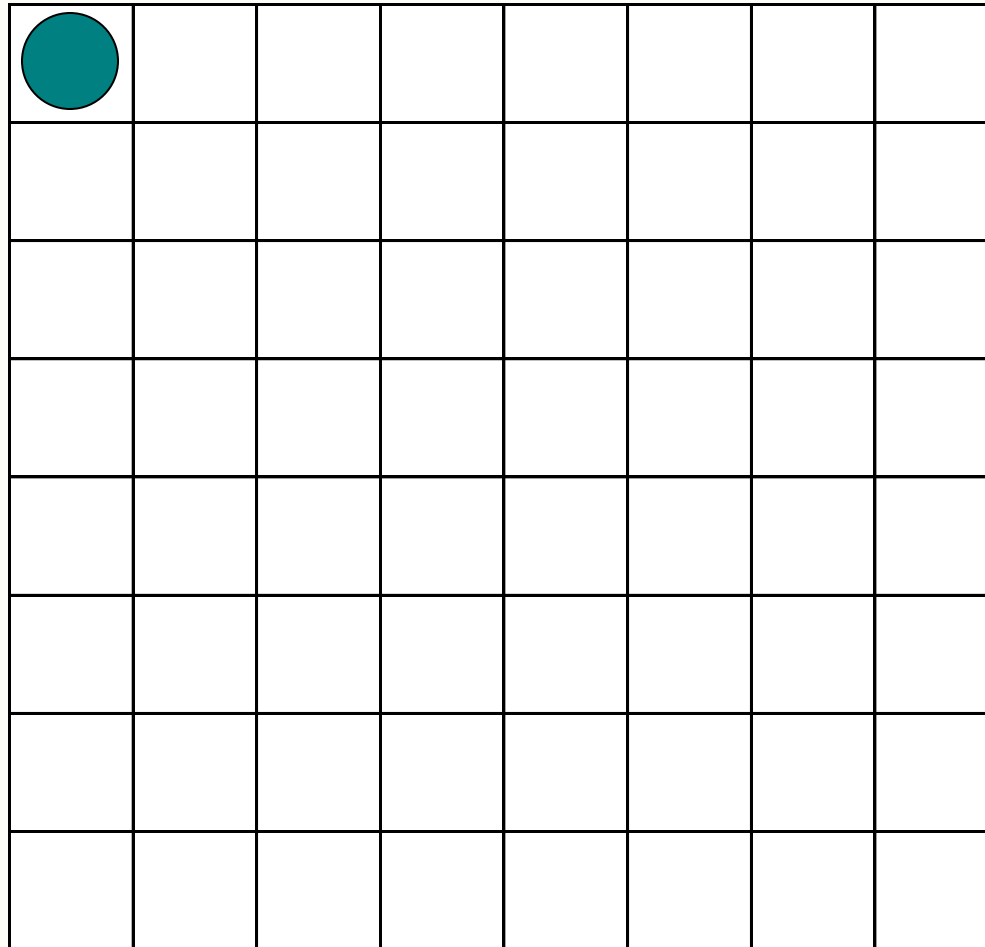
---

- A more efficient backtracking search strategy sees constraints as active constructs and **interleaves backtracking with constraint propagation**:
  - Whenever a variable is assigned a value, the consequences of such assignment are taken into account in all the constraints it appears to narrow the possible values of the variables not yet assigned.
  - If for one such variable there are no values to choose from, then a failure occurs and the search backtracks.
- This is a typical **test and generate** procedure
  - Firstly, values are tested to check their possible use.
  - Secondly, the values are assigned to the variables among the remaining values.
- Clearly, the reasoning that is done should have the adequate complexity otherwise the gains obtained from the narrowing of the search space are offset by the costs of such narrowing.



# Backtracking + Propagation

---




**Tests 0**

**Backtracks 0**

# Backtracking + Propagation

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



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

**Tests  $8 * 7 = 56$**

**Backtracks 0**

# Backtracking + Propagation




---

							
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

**Tests**  $56 + 6 * 6 = 92$

**Backtracks** 0

# Backtracking + Propagation

							
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

**Tests  $92 + 21 = 113$**

**Backtracks 0**

# Backtracking + Propagation + Heuristics

---

- In both types of backtrack search (pure backtracking as well as in backtracking + propagation) there is a need for heuristics.
- After all, in decision problems with  $n$  variables, a perfect heuristics would find a solution (if there is one) in exactly  $n$  steps (i.e. with  $n$  decisions – polynomial time).
- Of course, there are no such perfect heuristics for non-trivial problems (this would imply  $P = NP$ , a quite unlikely situation), but good heuristics can nonetheless significantly decrease the search space. Typically a heuristics consists of
  - **Variable selection:** The selection of the next variable to assign a value
  - **Value selection:** Which value to assign to the variable
- The adoption of a backtrack + propagation search method allows better heuristics to be used, that are not available in pure backtrack search methods.
- In particular a very simple heuristics, **first-fail**, is often very useful: whenever a variable is restricted to take a single value, select that variable and value.

# Backtracking + Propagation + Heuristics

---

Which queen to label?





●							
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

Tests  $92 + 21 = 113$

Backtracks 0

# Backtracking + Propagation + Heuristics





$Q_6$   
 may only  
 take value  
 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

**Tests  $92 + 21 = 113$**

**Backtracks 0**

# Backtracking + Propagation + Heuristics

							
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

Tests  $113+3+3+3+4 = 126$

Backtracks 0



# Backtracking + Propagation + Heuristics

$Q_8$   
 may only  
 take value  
 7




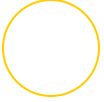

●							
1	1	●					
1	2	1	2	●			
1	6	2	1	2	3		
1		2	6	1	2	3	
1	3	2	○	3	1	2	3
1		2	6	3		1	2
1	6	2	2	3	6		1

Tests 126

Backtracks 0

# Backtracking + Propagation + Heuristics

---






							
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

**Tests 126**

**Backtracks 0**

# Backtracking + Propagation + Heuristics

---




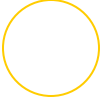
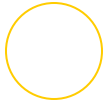
							
1	1						
1	2	1	2				
1	6	2	1	2	3	8	
1		2	6	1	2	3	
1	3	2		3	1	2	3
1		2	6	3	8	1	2
1	6	2	2	3	6		1

Tests  $126+2+2+2=132$

Backtracks 0

# Backtracking + Propagation + Heuristics

$Q_4$   
 may only  
 take value  
 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

Tests 132

Backtracks 0

# Backtracking + Propagation + Heuristics

---





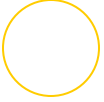

●							
1	1	●					
1	2	1	2	●			
1	6	2	1	2	3	8	●
1		2	6	1	2	3	
1	3	2	○	3	1	2	3
1		2	6	3	8	1	2
1	6	2	2	3	6	○	1

**Tests 132**

**Backtracks 0**

# Backtracking + Propagation + Heuristics

---





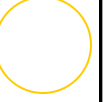
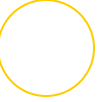
							
1	1						
1	2	1	2				
1	6	2	1	2	3	8	
1		2	6	1	2	3	4
1	3	2		3	1	2	3
1		2	6	3	8	1	2
1	6	2	2	3	6		1

**Tests**  $132+2+1=135$

**Backtracks** 0

# Backtracking + Propagation + Heuristics

$Q_5$   
 may only  
 take value  
 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

Tests 135

Backtracks 0

# Backtracking + Propagation + Heuristics

---

●							
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

**Tests 135**

**Backtracks 0**



# Backtracking + Propagation + Heuristics

---

●							
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

**Tests  $135+1=136$**

**Backtracks 0**

# Backtracking + Propagation + Heuristics

---



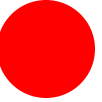




●							
1	1	●					
1	2	1	2	●			
1	6	2	1	2	3	8	○
1	○	2	6	1	2	3	4
1	3	2	○	3	1	2	3
1	5	2	6	3	8	1	2
1	6	2	2	3	6	○	1

**Tests 136**

**Backtracks 0**

# Backtracking + Propagation + Heuristics



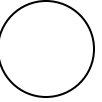

**Q<sub>7</sub>**  
 may take NO  
 value  
**Failure!**  
**Backtracks**  
 ... to Q<sub>3</sub> !

							
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

Tests 136

Backtracks 0+1=1

# Backtracking + Propagation + Heuristics

							
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

$$Q_1 = 1$$

$$Q_2 = 3$$

$$Q_3 = 5$$

Impossible !

Tests

136  
(324)

Backtracks

1  
(14)

Tests 136

Backtracks 1

# Backtracking + Propagation + Heuristics

---

- The adoption of constraint propagation and backtrack is more efficient for three main reasons:
  - Early detection of Failure:
    - In this case, after placing queens  $Q1 = 1$ ,  $Q2 = 3$  and  $Q3 = 5$ , a failure is detected without any backtracking.
  - Relevant backtracking:
    - Although a failure is detected in  $Q7$ , backtracking is done to  $Q3$ , and to none of the other queens ( $Q4$ ,  $Q5$ ,  $Q6$  and  $Q8$ , that are not relevant).
    - With pure backtracking many backtracks were done to undo choices in these queens.
  - Heuristics:
    - Constraint Propagation makes it easy to adopt heuristics based on the remaining values of the unassigned variables.

# Constraints: Basic Concepts

---

- To allow a study of constraint propagation in general, we start with some definitions and notation:

## Definition (**Domain of a Variable**):

- The **domain** of a variable is the set of values that can be assigned to that variable.
  - Given some variable **x**, its domain will be usually referred to as **dom(x)** or, simply, **Dx**.
- **Example:** The N queens problem may be modelled by means of N variables,  $q_1$  to  $q_n$ , all with the domain from 1 to n.

$$\text{Dom}(q_i) = \{1, 2, \dots, n\} \quad \text{or} \quad q_i :: 1..n.$$

- **Note:** In the first part of this course we will deal with Finite Domains, i.e. domains that are finite sets of values.

# Constraints: Basic Concepts

---

- To formalise the notion of the state of a variable (i.e. its assignment with one of the values in its domain) we have the following

## Definition (**Label**):

- A **label** is a Variable-Value pair, where the Value is one of the elements of the domain of the Variable.

- The notion of a partial solution, in which some of the variables of the problem have already assigned values, is captured by the following

## Definition (**Compound Label**):

- A **compound label** is a set of labels with distinct variables.

# Constraints: Basic Concepts

---

- We come now to the formal definition of a constraint

## Definition (**Constraint**):

- Given a set of variables, a **constraint** is a set of compound labels on these variables.
- Alternatively, a constraint may be defined simply as a relation, i.e. a subset of the cartesian product of the domains of the variables involved in that constraint.
- For example, given a constraint  $c_{ijk}$  involving variables  $x_i$ ,  $x_j$  and  $x_k$ , then

$$c_{ijk} \subseteq \text{dom}(x_i) \times \text{dom}(x_j) \times \text{dom}(x_k)$$



# Constraints: Basic Concepts

---

- Given a constraint  $c$ , the set of variables involved in that constraint is denoted by  $\mathbf{vars}(c)$ .
- Symmetrically, the set of constraints in which variable  $x$  participates is denoted by  $\mathbf{cons}(x)$ .
- Notice that a constraint is a relation, not a function, so that it is always  $c_{ij} = c_{ji}$ .
  
- In practice, constraints may be specified by
  - **Extension**: through an explicit enumeration of the allowed compound labels;
  - **Intension**: through some predicate (or procedure) that determines the allowed compound labels.

# Constraints: Basic Concepts

---

- For example, constraint  $c_{13}$  involving queens 1 and 3 in the 4-queens problem, may be specified

• **By extension** (label form):

$$c_{13} = \{ \{q_1-1, q_3-2\}, \{q_1-1, q_3-4\}, \{q_1-2, q_3-1\}, \{q_1-2, q_3-3\}, \\ \{q_1-3, q_3-2\}, \{q_1-3, q_3-4\}, \{q_1-4, q_3-1\}, \{q_1-4, q_3-3\} \} .$$

or, in tuple (relational) form, omitting the variables

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

• **By intension:**

$$c_{13} = (q_1 \neq q_3) \quad \wedge \quad (1+q_1 \neq 3+q_3) \quad \wedge \quad (3+q_1 \neq 1+q_3) .$$

# Constraints: Basic Concepts

---

## Definition (**Constraint Arity**):

- The **arity** of some constraint **c** is the number of variables over which the constraint is defined, i.e. the cardinality of set **Vars(c)**.
- Despite the fact that constraints may have an arbitrary arity, an important subset of the constraints is the set of **binary constraints**.
- The importance of such constraints is two-fold
  - All constraints may be converted into binary constraints
  - A number of concepts and algorithms are appropriate for these constraints.

# Constraints: Basic Concepts

---

## Definition (**Constraint Satisfaction 1**):

- A compound label satisfies a constraint if their variables **are the same** and if the compound label is a member of the constraint.
- In practice, it is convenient to generalise constraint satisfaction to compound labels that strictly contain the constraint variables.

## Definition (**Constraint Satisfaction 2**):

- A compound label satisfies a constraint if its variables **contain** the constraint variables and the projection of the compound label to these variables is a member of the constraint.

# Constraints: Basic Concepts

---

## Definition (**Constraint Satisfaction Problem**):

- A constraint satisfaction problem is a triple  $\langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$  where
  - $\mathbf{X}$  is the set of variables of the problem
  - $\mathbf{D}$  is the domain(s) of its variables
  - $\mathbf{C}$  is the set of constraints of the problem

## Definition (**Problem Solution**):

- A solution to a constraint satisfaction problem  $\mathbf{P}: \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$ , is a compound label over the variables  $\mathbf{X}$  of the problem, which satisfies all constraints in  $\mathbf{C}$ .

# Constraints: Basic Concepts

---

## Definition (**Constrained Optimisation Problem**):

- A constrained optimization problem (**COP**) is a tuple  $\langle \mathbf{X}, \mathbf{D}, \mathbf{C}, \mathbf{F} \rangle$  where
  - $\mathbf{X}$  is the set of variables of the problem
  - $\mathbf{D}$  is the domain(s) of its variables
  - $\mathbf{C}$  is the set of constraints of the problem
  - $\mathbf{F}$  is a function on the variables of the problem

## Definition (**Problem Solution**):

- $\mathbf{S}$  is a solution of a COP  $\mathbf{P}: \langle \mathbf{X}, \mathbf{D}, \mathbf{C}, \mathbf{F} \rangle$ , iff:
  - $\mathbf{S}$  is a solution of the corresponding CSP  $\mathbf{P}': \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$ ;
  - No other solution  $\mathbf{S}'$  has a better value for function  $\mathbf{F}$

# Constraints: Basic Concepts

---

- For convenience, the (binary) constraints of a problem may be considered as forming a special **constraint graph**.

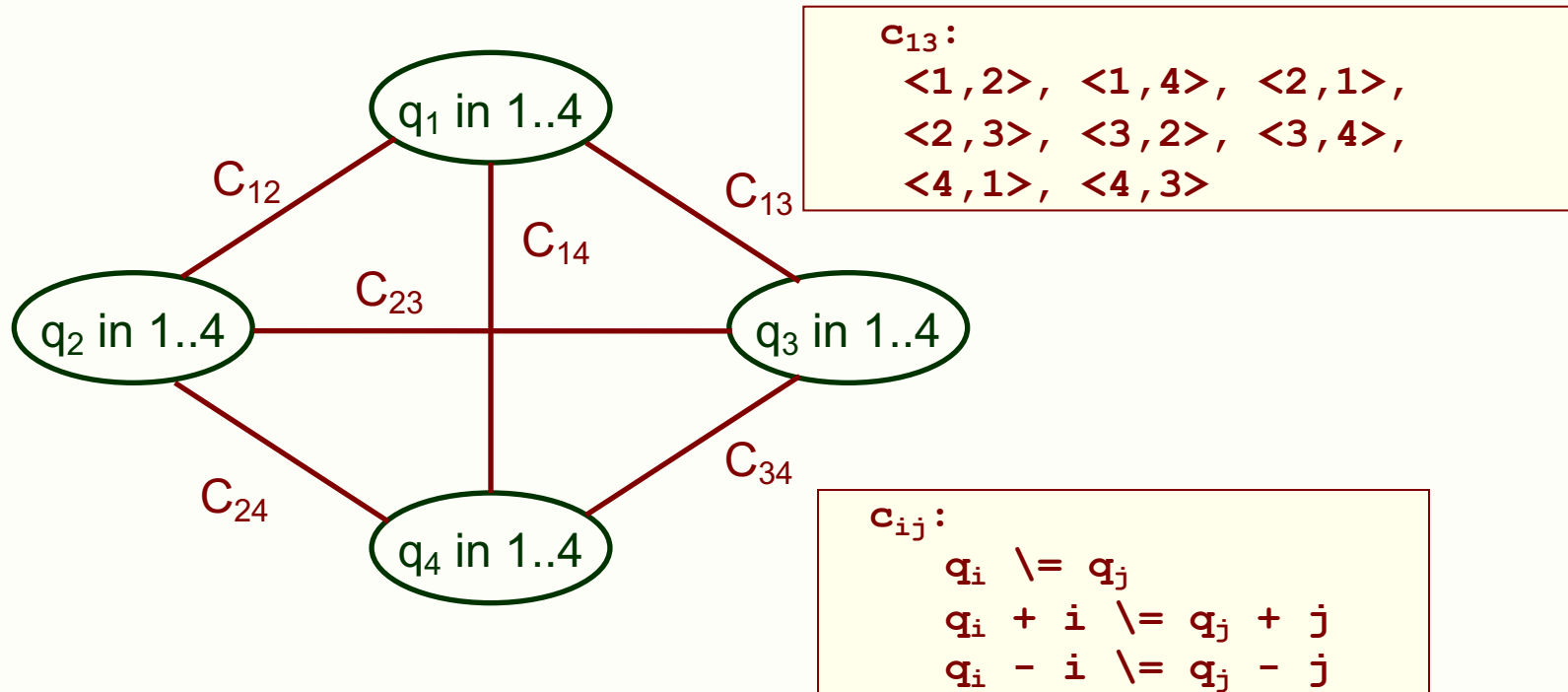
**Definition (Constraint Graph or Constraint Network):**

- The **Constraint Graph** or **Constraint Network** of a binary constraint satisfaction problem is defined as follows
  - There is a node for each of the variables of the problem.
  - For each (non-trivial) constraint of the problem, involving one or two variables, the graph contains an arc linking the corresponding nodes.
- When the problems include constraints with arbitrary arity, the Constraint Network may be formed after converting these constraints on its binary equivalent.

# Constraints: Basic Concepts

## Example:

- The 4-queens problem may be specified by the following **constraint network**:





# Constraints: Basic Concepts

---

- An important issue to consider in solving a constraint satisfaction problem is the existence of redundant values and labels in its constraints.

## Definition (**Redundant Value**):

- A **value** in the domain of a variable is **redundant**, if it does not appear in any solution of the problem.

## Definition (**Redundant Label**):

- A **compound label** of a constraint is **redundant** if it is not the projection to the constraint variables of a solution to the whole problem.
- Redundant values and labels increase the search space uselessly, and should thus be avoided. There is no point in testing a value that does not appear in any solution !

# Constraints: Basic Concepts

---

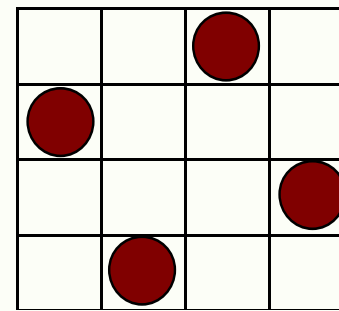
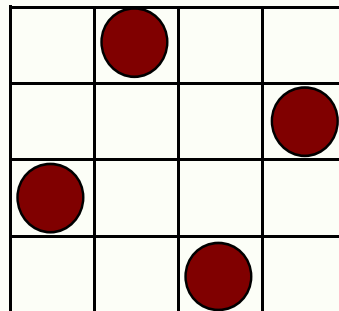
## Example:

- The 4-queens problem only admits two solutions:

$\langle 2,4,1,3 \rangle$

and

$\langle 3,1,4,2 \rangle$ .



- Hence,

- values 1 and 4 are redundant in the domain of variables  $q_1$  and  $q_4$ ; and
- values 2 and 3 are redundant in the domain of variables  $q_2$  and  $q_3$ .

# Constraints: Basic Concepts

---

- Redundant values and labels increase the search space useless, and should thus be avoided (there is no point in testing a value that does not appear in any solution !). Hence, the following definitions:

## Definition (**Equivalent Problems**):

- Two problems  $P_1 = \langle X_1, D_1, C_1 \rangle$  and  $P_2 = \langle X_2, D_2, C_2 \rangle$  are equivalent iff they have the same variables (i.e.  $X_1 = X_2$ ) and the same set of solutions.
- The “simplification” of a problem may also be formalised

## Definition (**Reduced Problem**):

- A problem  $P = \langle X, D, C \rangle$  is reduced to  $P' = \langle X', D', C' \rangle$  if
  - $P$  and  $P'$  are equivalent;
  - The domains  $D'$  are included in  $D$ ; and
  - The constraints  $C'$  are at least as restrictive as those in  $C$ .

# Complexity of Search

---

- Clearly, the more reduced a problem is, the easier it is, in principle, to solve it.
- Given a problem  $\mathbf{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$  with  $n$  variables  $\mathbf{x}_1, \dots, \mathbf{x}_n$  the search space where solutions can potentially be found (i.e. the leaves of the search tree with compound labels  $\{\langle \mathbf{x}_1 - \mathbf{v}_1 \rangle, \dots, \langle \mathbf{x}_n - \mathbf{v}_n \rangle\}$ ) has cardinality

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

- Assuming identical cardinality (or some kind of average of the domains size) for all the variable domains, ( $\#D_i = d$ ) the search space has cardinality

$$\#S = d^n$$

which is exponential on the “size”  $n$  of the problem.

# Complexity of Search

- Given a problem with initial cardinality  $d$  of its variables, and a reduced problem whose domains have lower cardinality  $d'$  ( $<d$ ) the size of the potential search space also decreases exponentially!

$$S'/S = d'^n / d^n = (d'/d)^n$$

- Such exponential decrease may be very significant for “reasonably” large values of  $n$ , as shown in the table.

		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'										

# Propagation in Search

---

- The effort in reducing the domains must be considered within the general scheme to solve the problem.
- In Constraint (Logic) Programming, the specification of the constraints precedes the enumeration of the variables.

**Problem (Vars) :-**

**Declaration of Variables and Domains,  
Specification of Constraints,  
Labelling of the Variables.**

- In general, search is performed exclusively on the labelling of the variables.
- The execution model alternates enumeration with propagation, making it possible to reduce the problem at various stages of the solving process.

# Complexity of Search

---

- In complete search methods, that deal with search through backtracking, the solving method is **constructive** and **incremental**, whereby a compound label is completed (*constructive*) throughout the solving process, one variable at a time (*incremental*), until a solution is reached.
- However, one must check that, at every step in the construction of a solution, the resulting label still has the potential to reach a complete solution.

## Definition (**k-Partial Solution**):

- A **k-partial solution** of a constraint solving problem  $P = \langle X, D, C \rangle$ , is a compound label on a subset of  $k$  of its variables,  $X_k$ , that satisfies all the constraints in  $C$  whose variables are included in  $X_k$ .

# Propagation in Search

---

- Given a problem with  $n$  variables  $x_1$  to  $x_n$ , and assuming a lexicographical variable/value heuristics, the execution model follows the following pattern to incrementally extend partial solutions until a complete solution is obtained:

```
Declaration of Variables and Domains,  
Specification of Constraints,  
    propagation, % reduction of the whole problem  
% Labelling of Variables,  
    label(x1), % variable/value selection with backtracking  
    propagation, % reduction of problem {x2 ... xn}  
    label(x2),  
    propagation, % reduction of problem {x3 ... xn}  
    ...  
    label(xn-1)  
    propagation, % reduction of problem {xn}  
    label(xn)
```



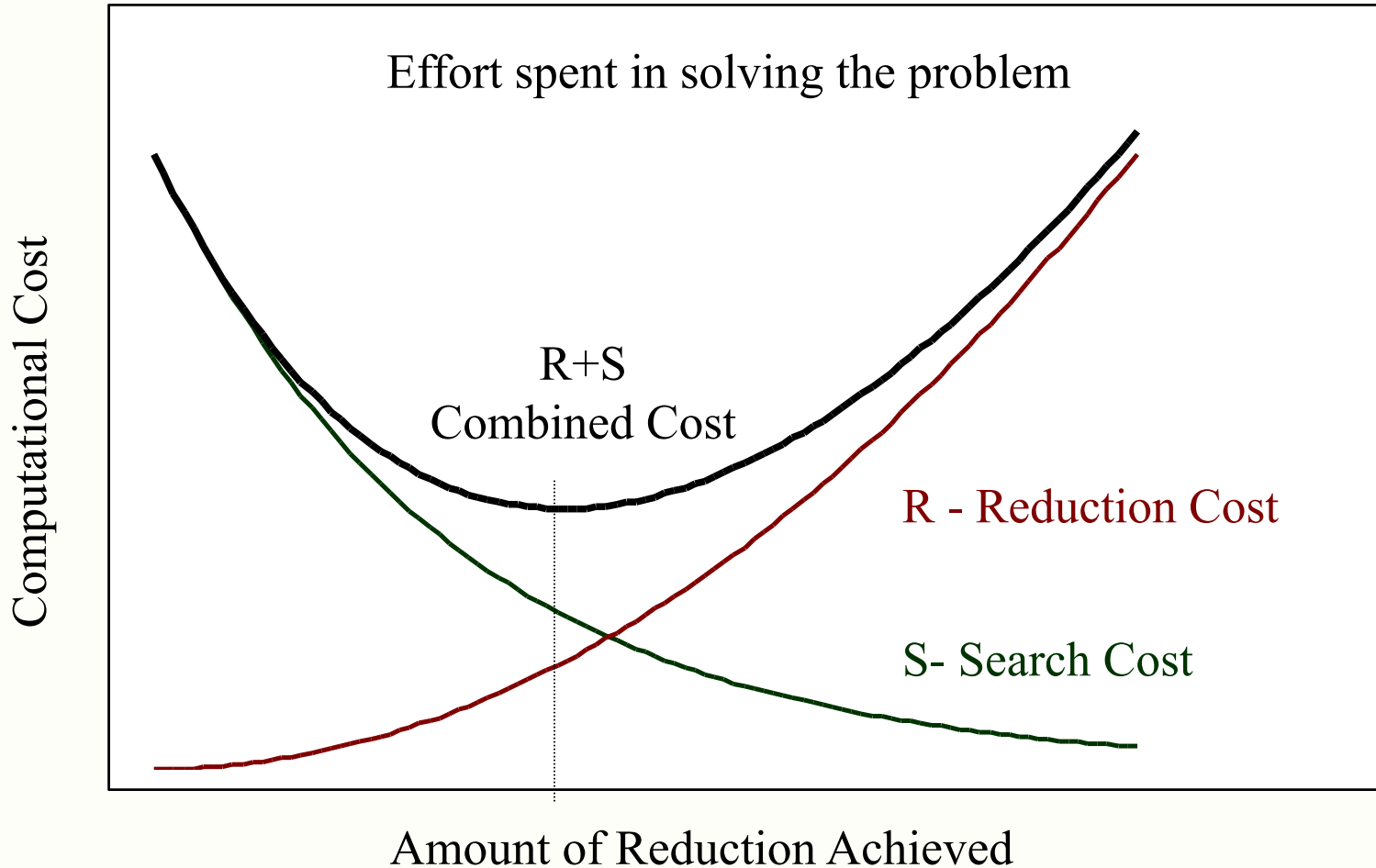
# Complexity of Search

---

- In practice, this potential narrowing of the search space has a cost involved in finding the redundant values (and labels).
- A detailed analysis of the costs and benefits in the general case is extremely complex, since the process depends highly on the instances of the problem to be solved.
- However, it is reasonable to assume that the computational effort spent on problem reduction is not proportional to the reduction achieved, becoming less and less efficient.
- After some point, the gain obtained by the reduction of the search space does not compensate the extra effort required to achieve such reduction.

# Complexity of Search

- Qualitatively, this process may be represented by means of the following picture



# Propagation: Consistency Criteria

---

- Consistency criteria enable to establish redundant values in the variables domains in an indirect form, i.e. requiring no prior knowledge on the set of problem solutions.
- Hence, procedures that maintain these criteria during the “propagation” phases, will eliminate redundant values and so decrease the search space on the variables yet to be enumerated.
- For constraint satisfaction problems with binary constraints, the most usual criteria are, in increasingly complexity order,
  - **Node Consistency**
  - **Arc Consistency**
  - **Path Consistency**
  - **i-Consistency**

# Assessment

---

- Evaluation consists of the following components
  - Project 1 – Finite Domains Problem
  - Mini-Test 1 – Finite Domains Concepts
  - Project 2 – Continuous Domains Problem
  - Mini-Test 2 – Continuous Domains Concepts
- Projects are made in team work (2 students per group) and the tests assess the students individually.
- All components have the same weight for the final grade.
- Students that do not get the minimum grade, are allowed to do a repetition exam if they get at least an average grade of 8/20 in the two projects.
- Exact dates to be announced –
  - Project 1 and Mini-test 1 at mid-term (end October)
  - Project 2 and Mini-test 2 at the end of semester (mid December)