Search and Optimisation

- An overview

- Backtrack Search and Constraint Propagation
- Constraint Networks
- Consistency Criteria
- Node-, Arc- and Path-consistency

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

- The same specification can lead to different search strategies when sequentially assigning values to variables.

- The simplest backtracking strategy sees constraints in a passive form:

  - Whenever a variable is assigned a variable, the constraints whose variables are assigned variables are checked for satisfaction

  - If this is not the case, the search backtracks (chronological backtrack).

- This is a typical **generate and test** procedure

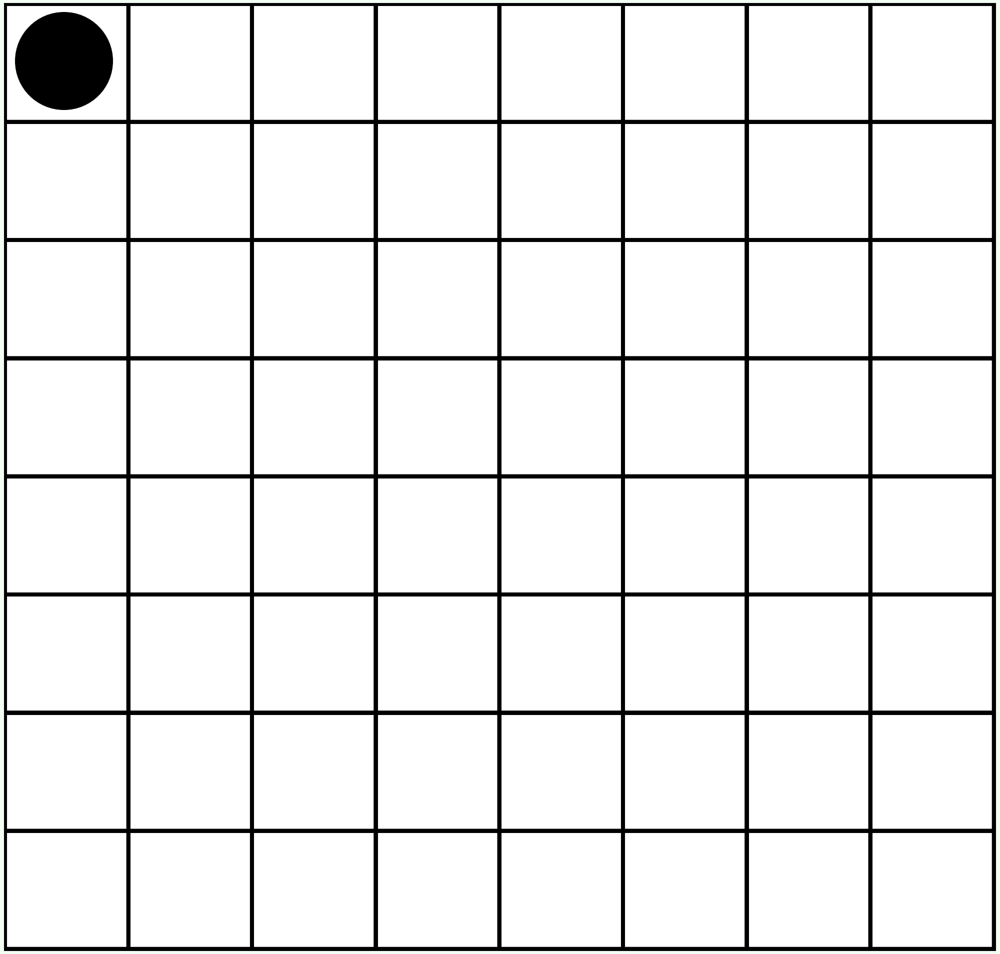
  - Firstly, values are generated

  - 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.

- This procedure is illustrated in the 8-queens problem.

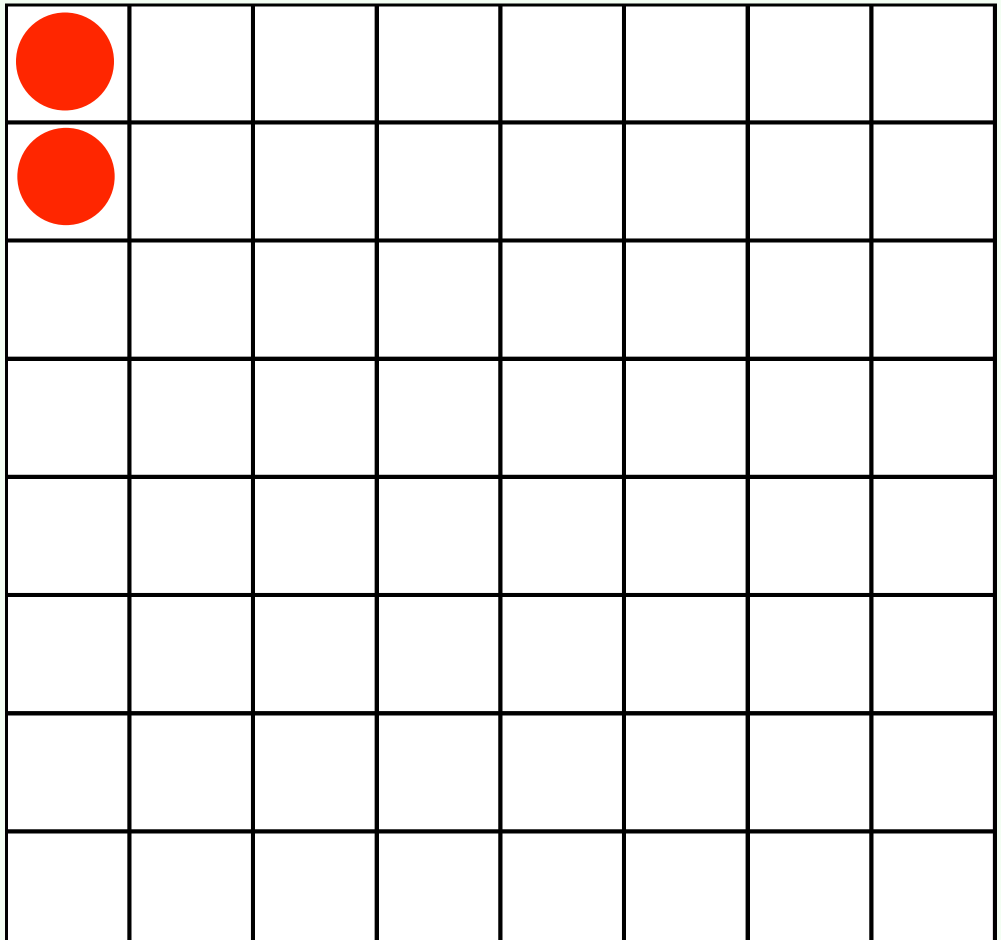
Backtracking



Tests 0 **Backtracks 0**

Backtracking

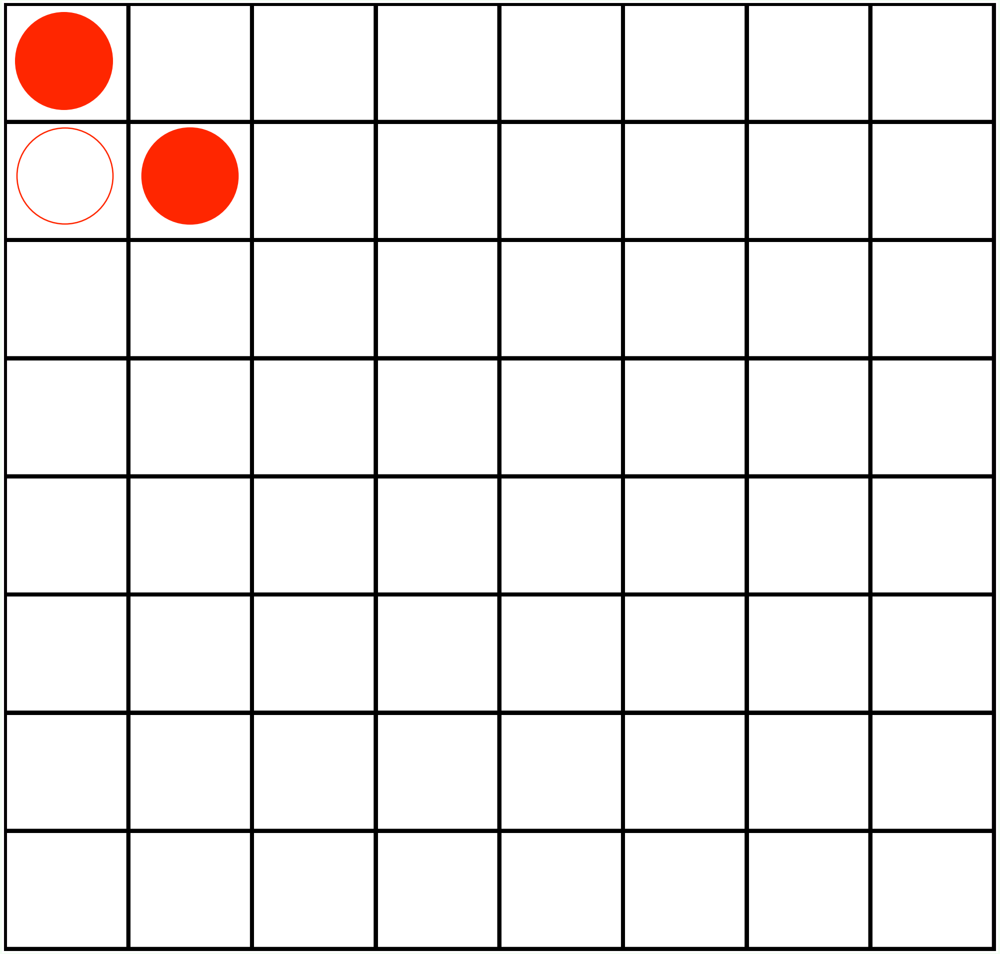
```
Q1 \= Q2,  L1+Q1 \= L2+Q2,  L1+Q2 \= L2+Q1.
```



Tests 0 +1 = 1 **Backtracks 0**

Backtracking

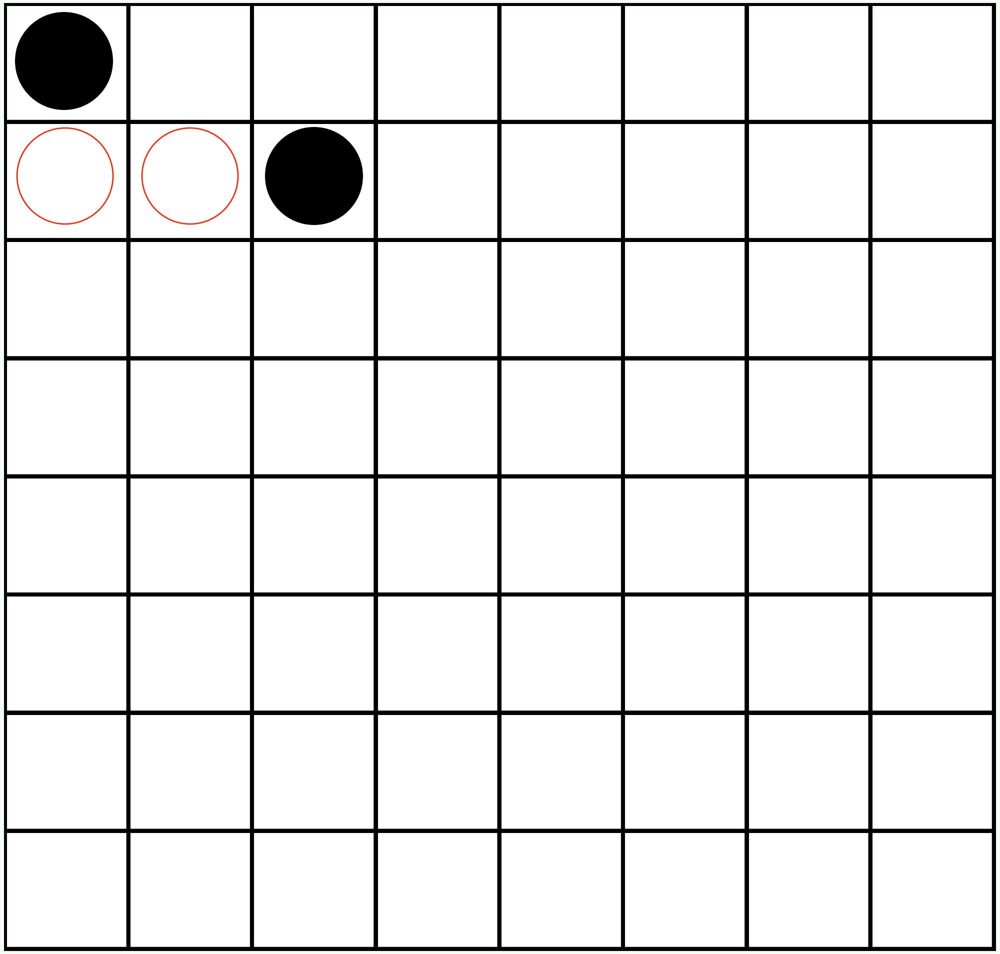
```
Q1 \= Q2,  L1+Q1 \= L2+Q2,  L1+Q2 \= L2+Q1.
```



Tests 1 +1 = 2 **Backtracks 0**

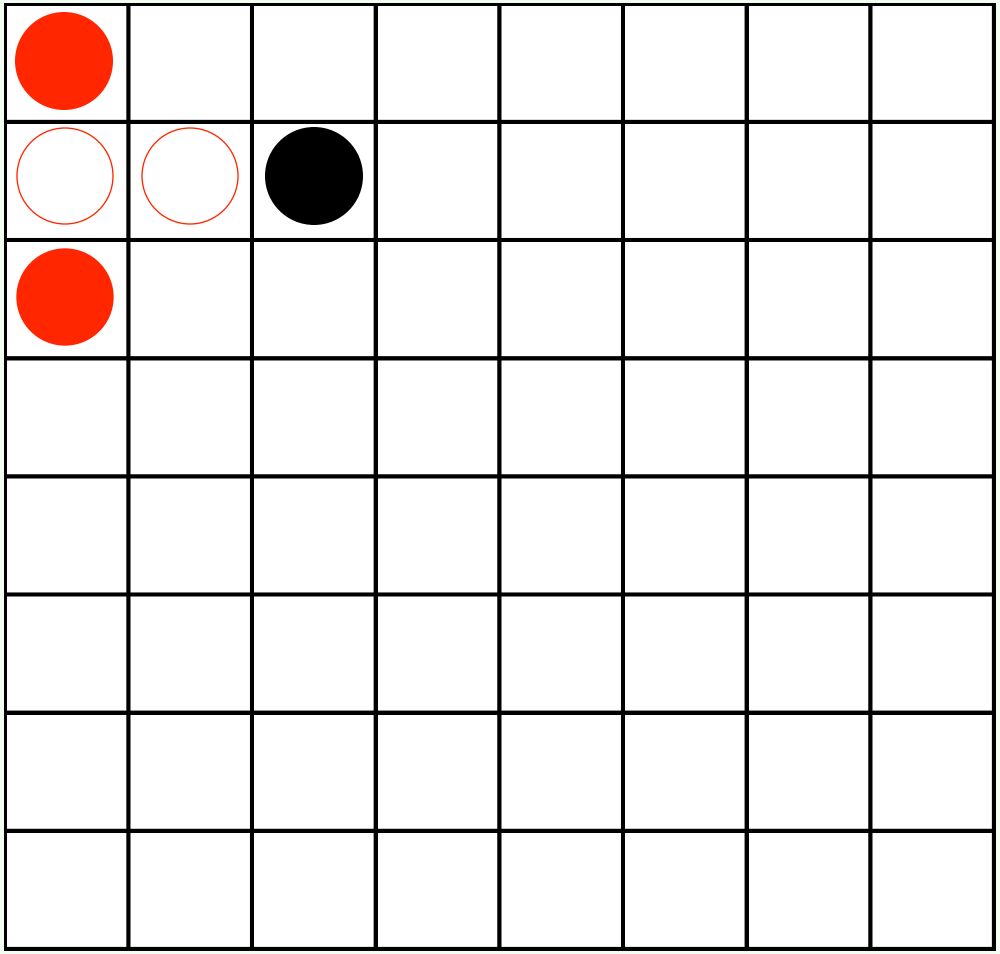
Backtracking

```
Q1 \= Q2,  L1+Q1 \= L2+Q2,  L1+Q2 \= L2+Q1.
```



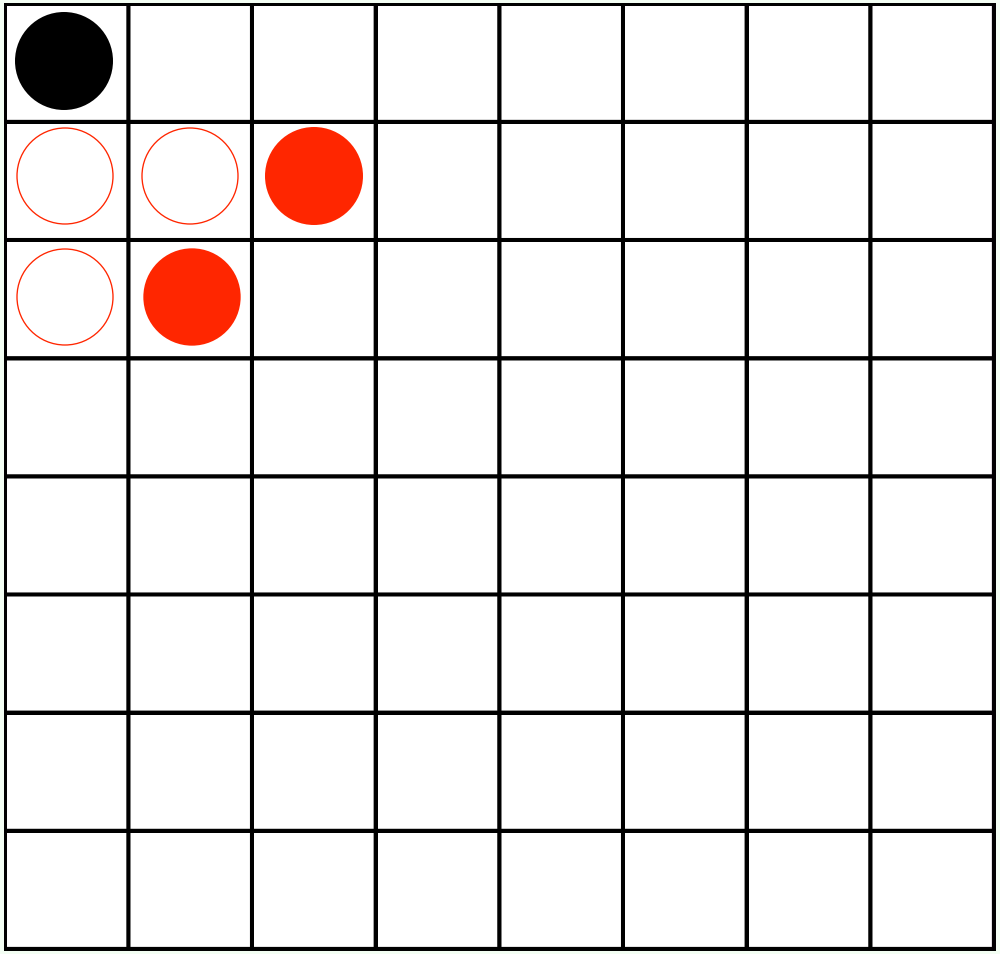
Tests 2 +1 = 3 **Backtracks 0**

Backtracking



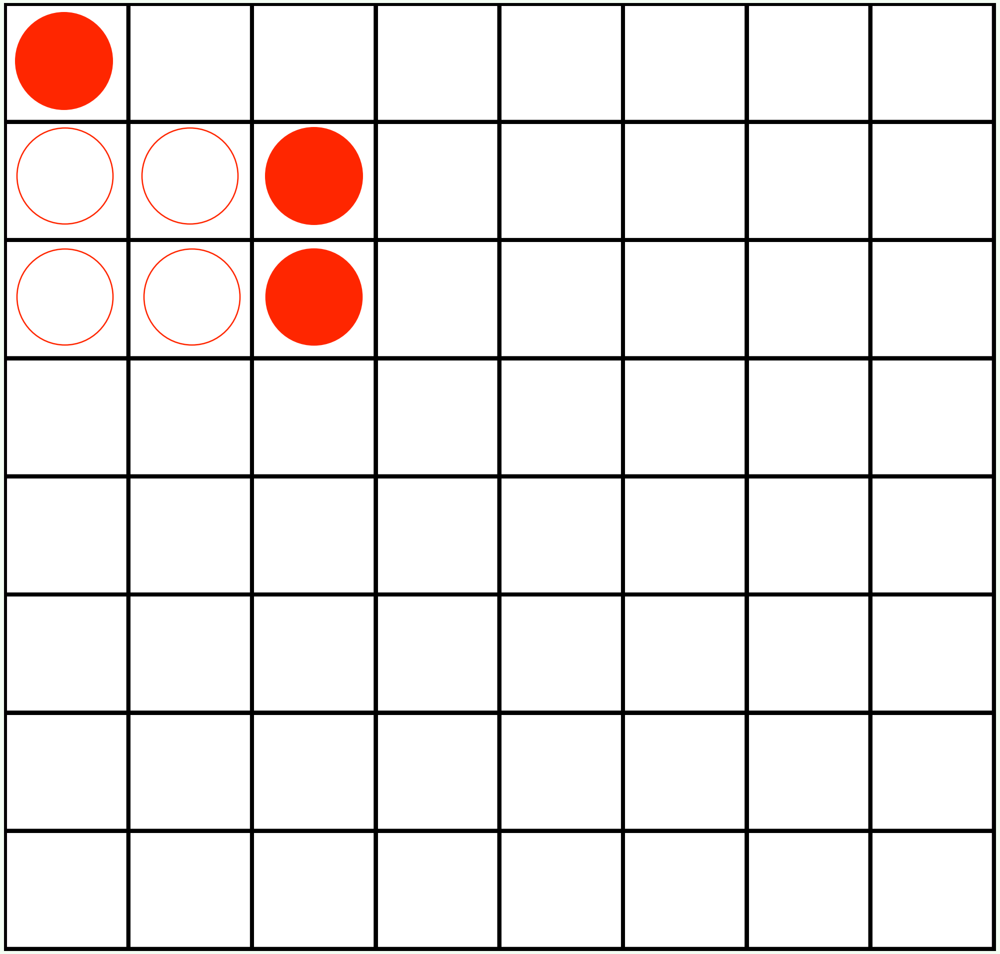
Tests 3 +1 = 4 **Backtracks 0**

Backtracking



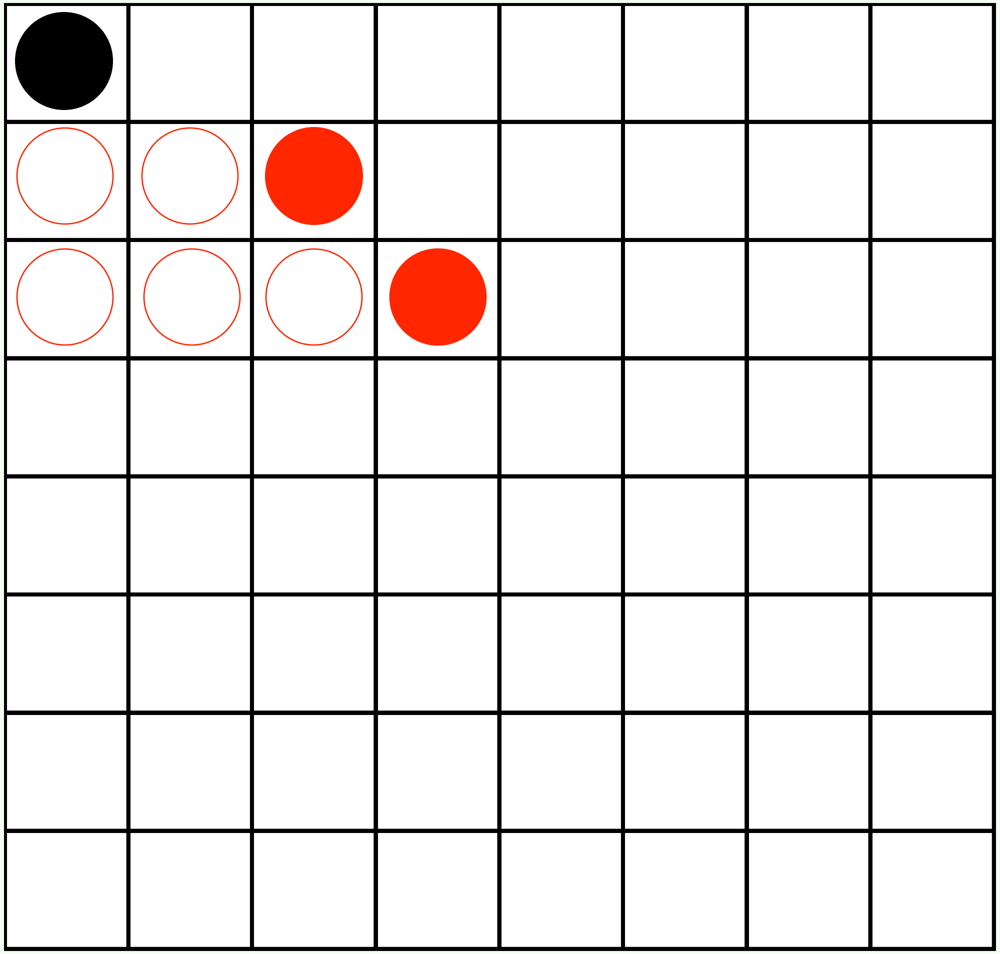
Tests 4 +2 = 6 **Backtracks 0**

Backtracking



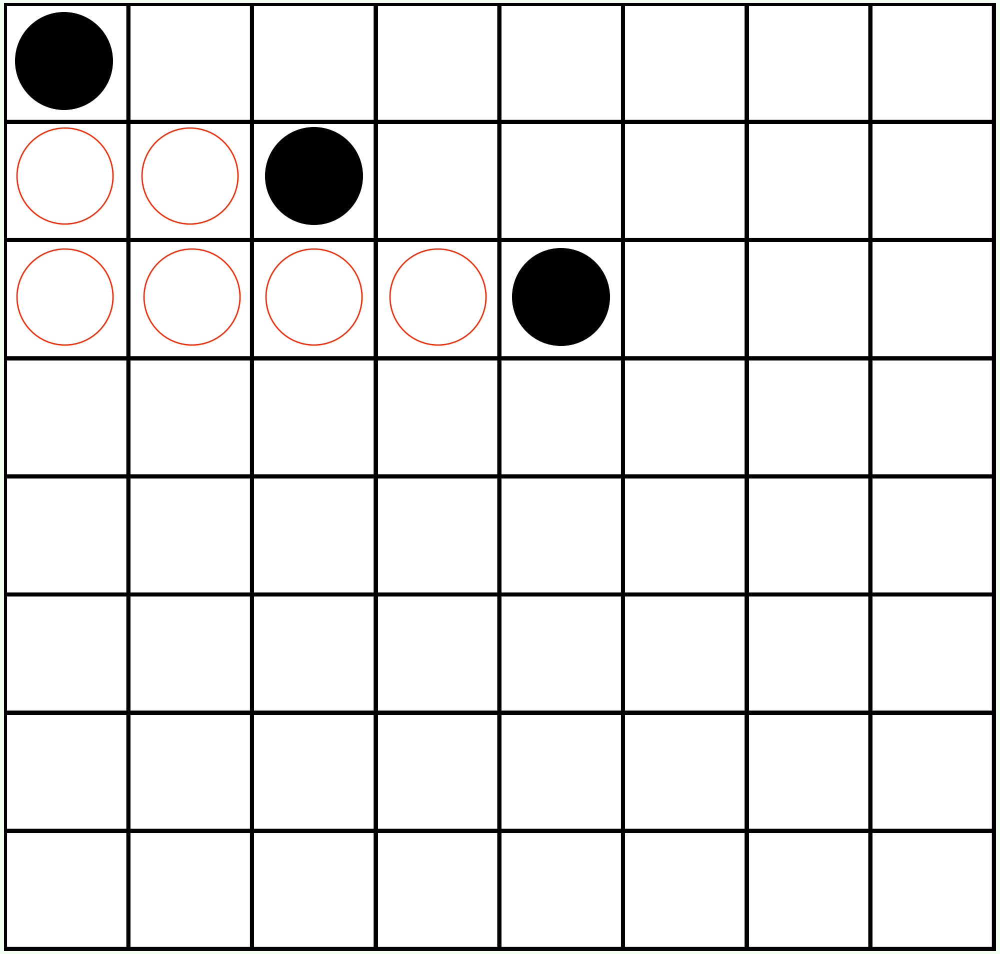
Tests 6 + 1 = 7 **Backtracks 0**

Backtracking



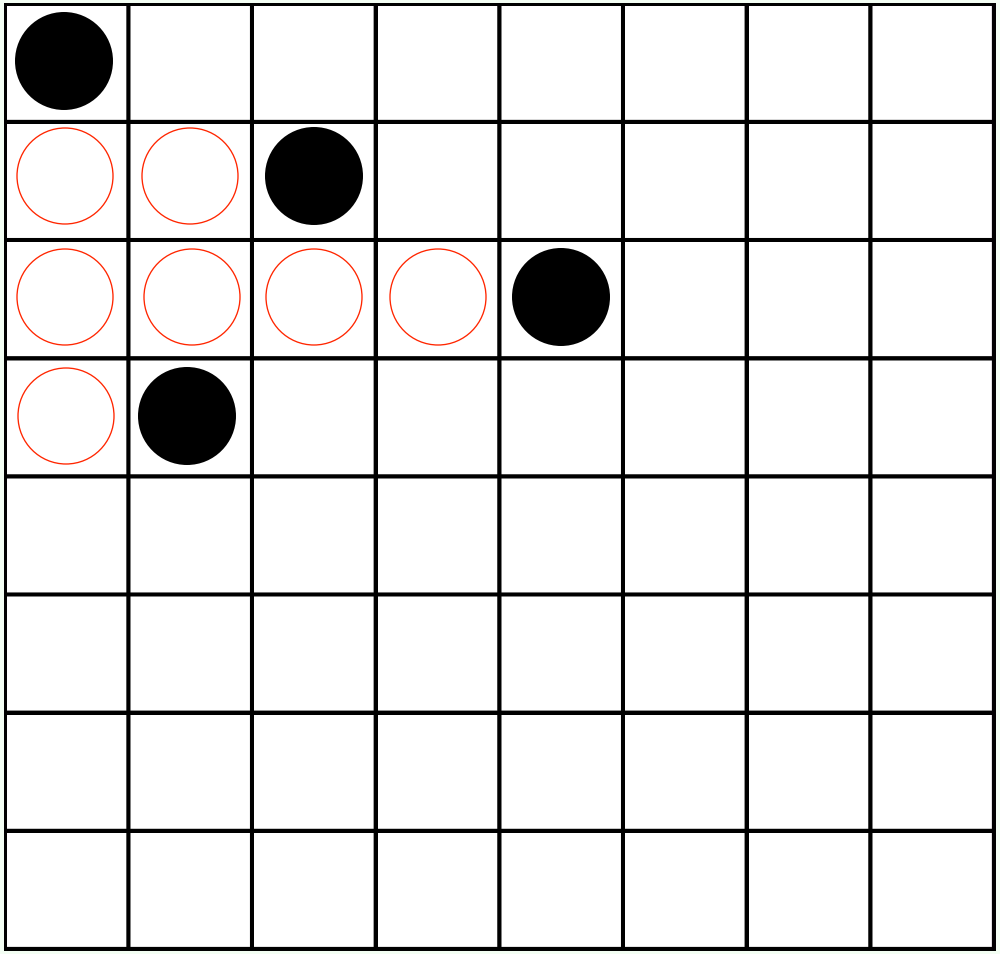
Tests 7 + 2 = 9 **Backtracks 0**

Backtracking



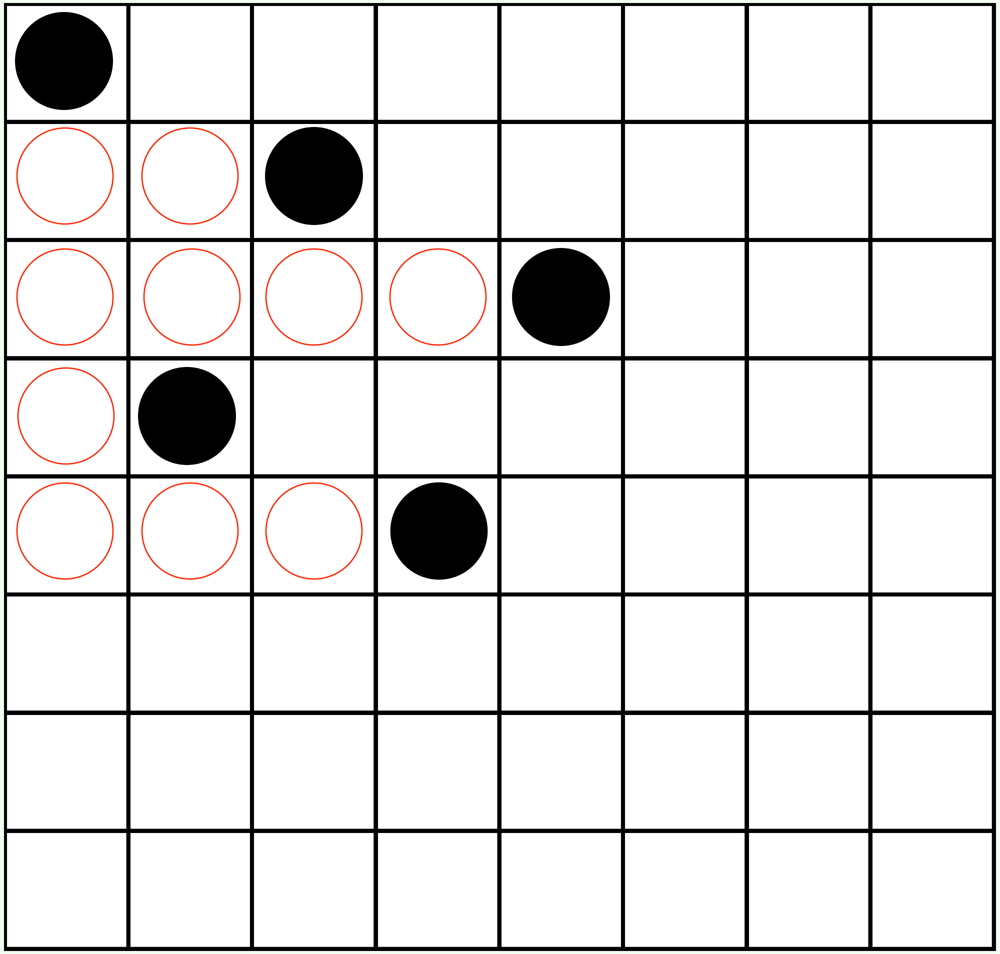
Tests 9 + 2 = 11 **Backtracks 0**

Backtracking



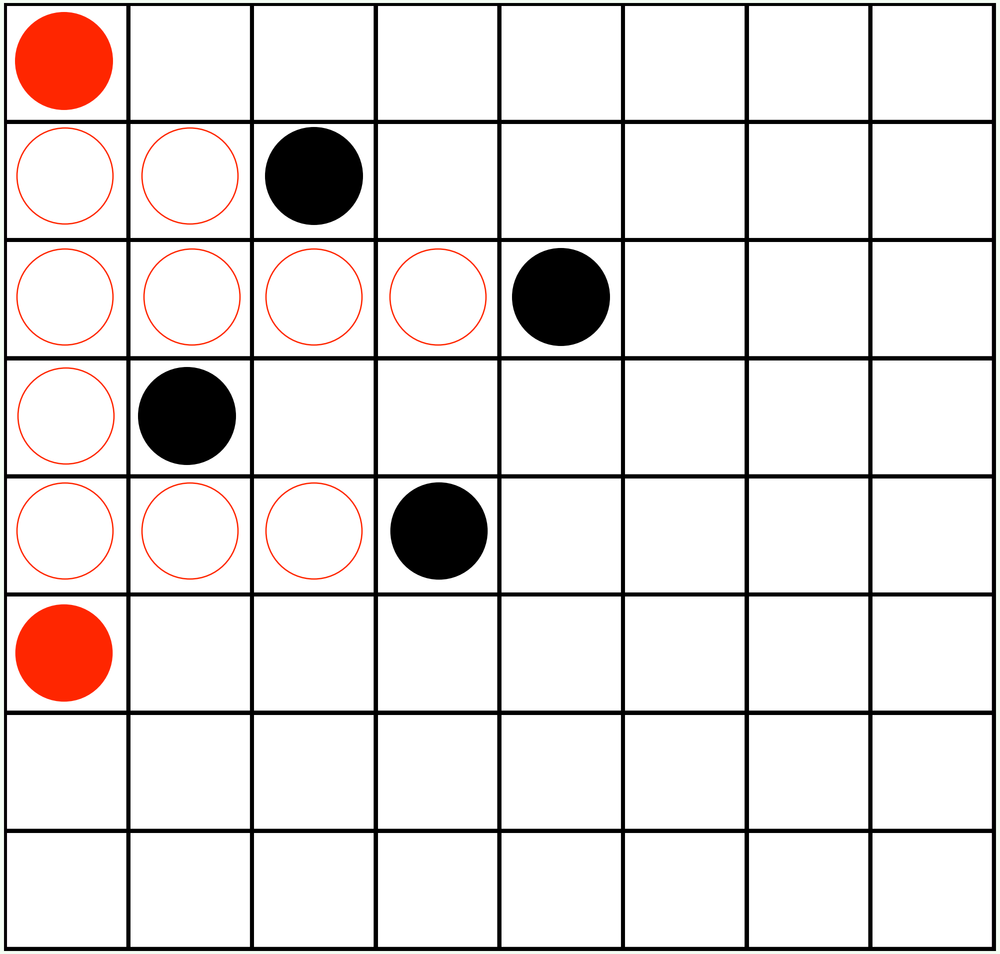
Tests 11 + 1 + 3 = 15 **Backtracks 0**

Backtracking



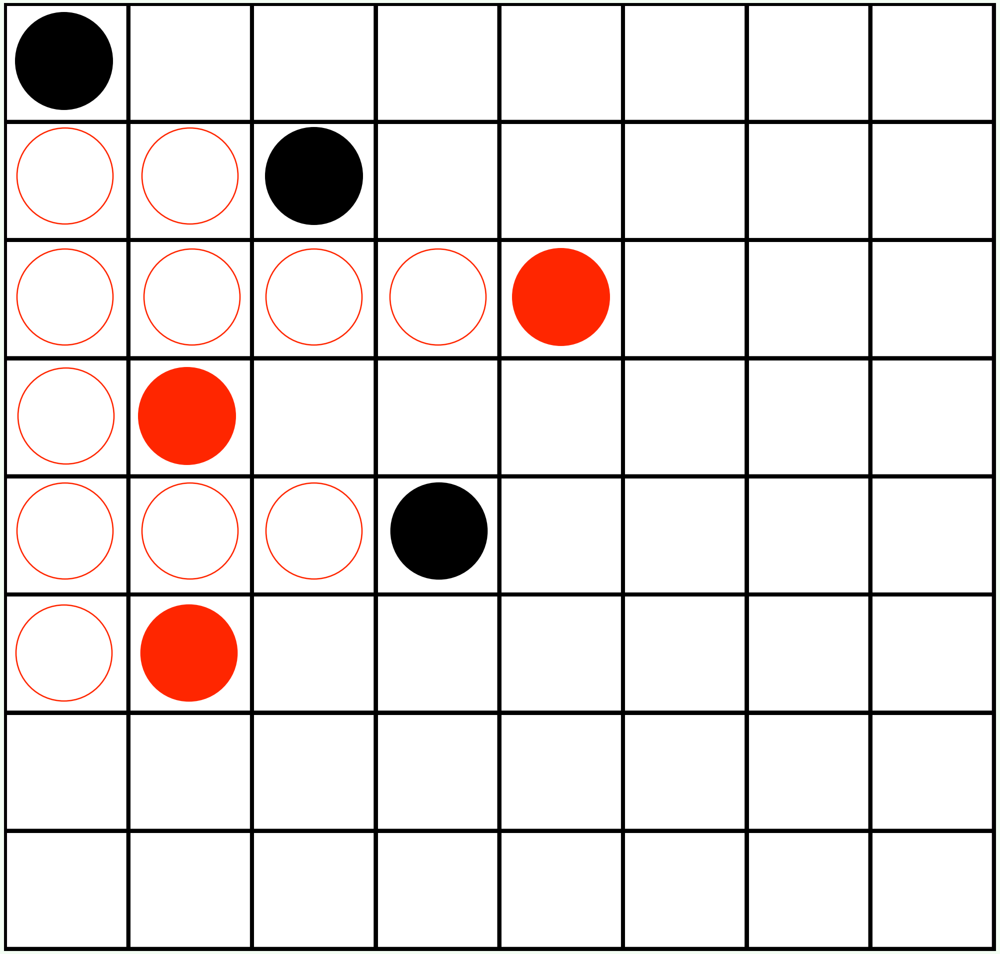
Tests 15+1+4+2+4 = 26 Backtracks 0

Backtracking



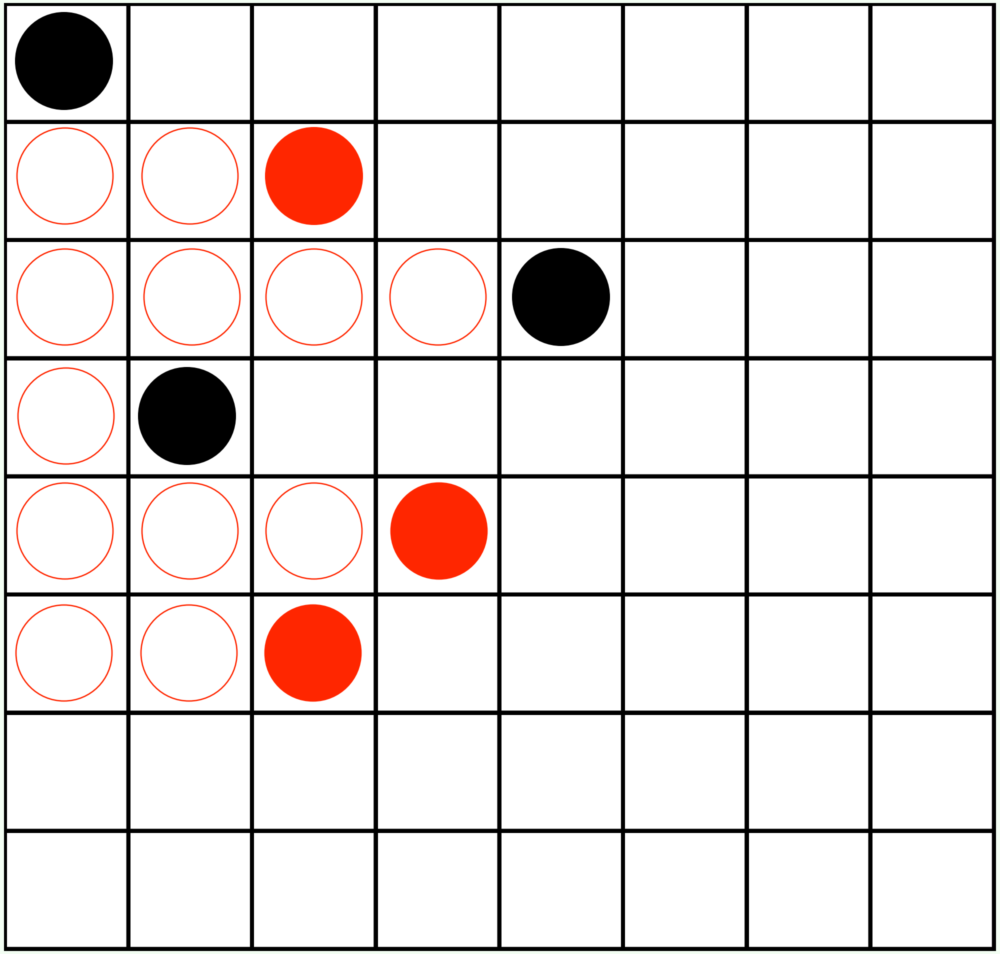
Tests 26+1 = 27 **Backtracks 0**

Backtracking



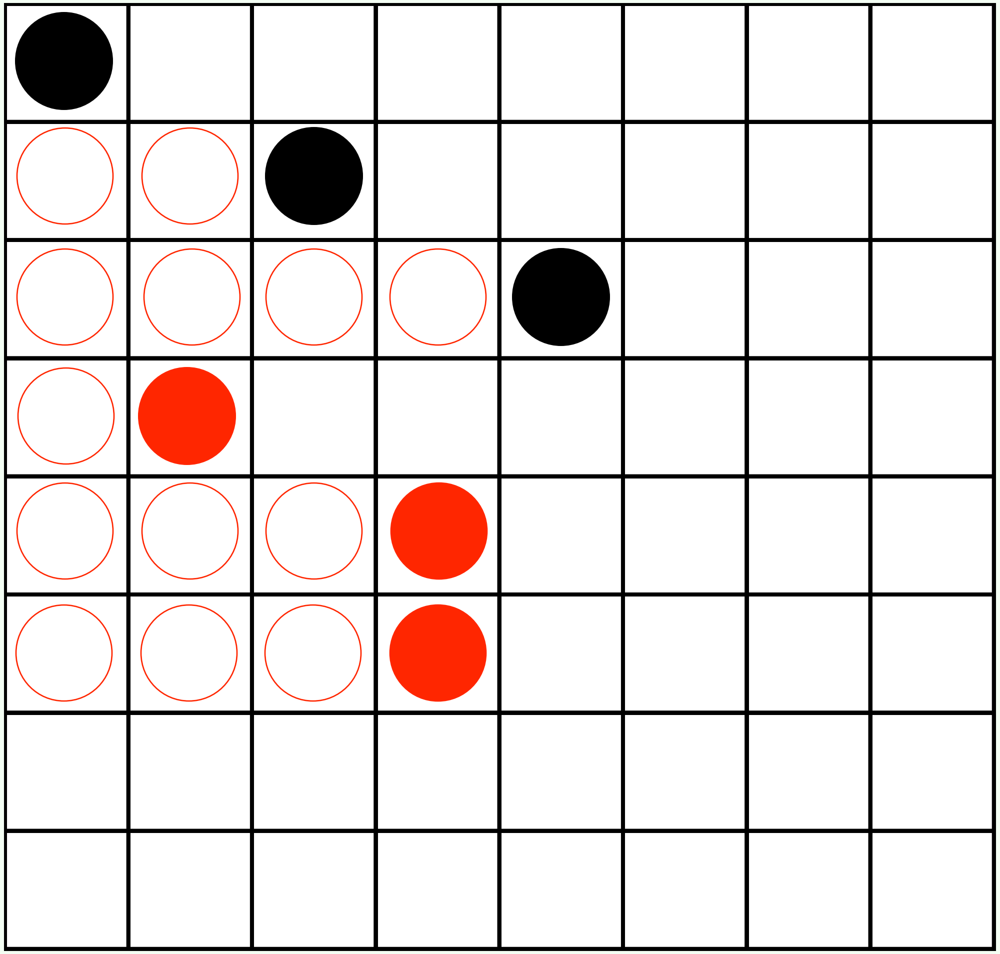
Tests 27 + 3 = 30 Backtracks 0

Backtracking



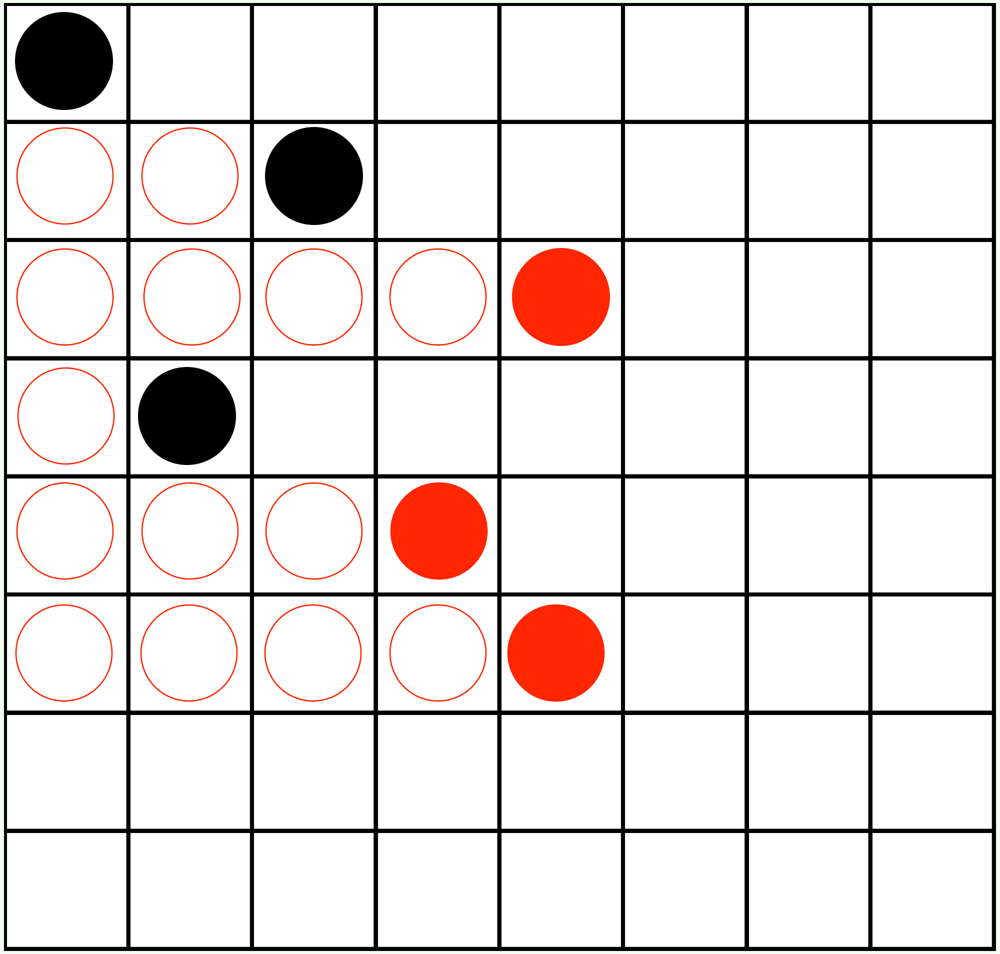
Tests 30+2 = 32 Backtracks 0

Backtracking



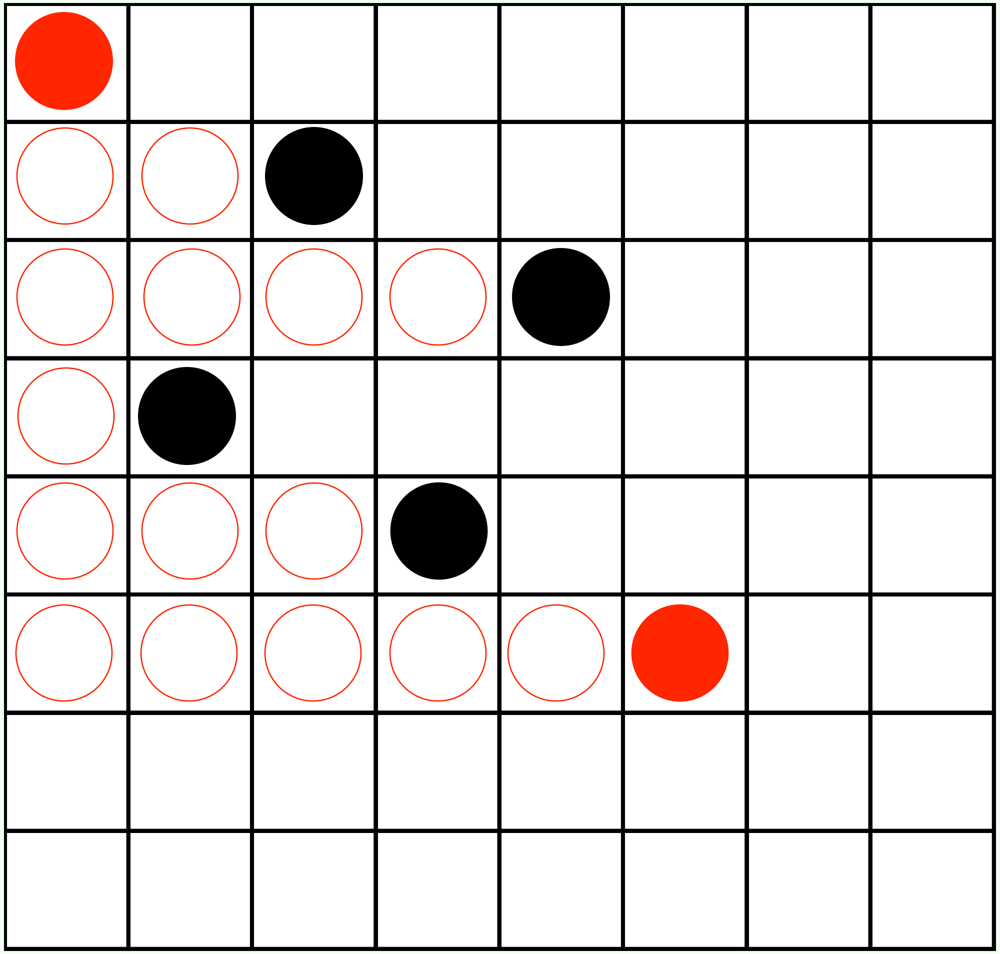
Tests 32 + 4 = 36   Backtracks 0

Backtracking



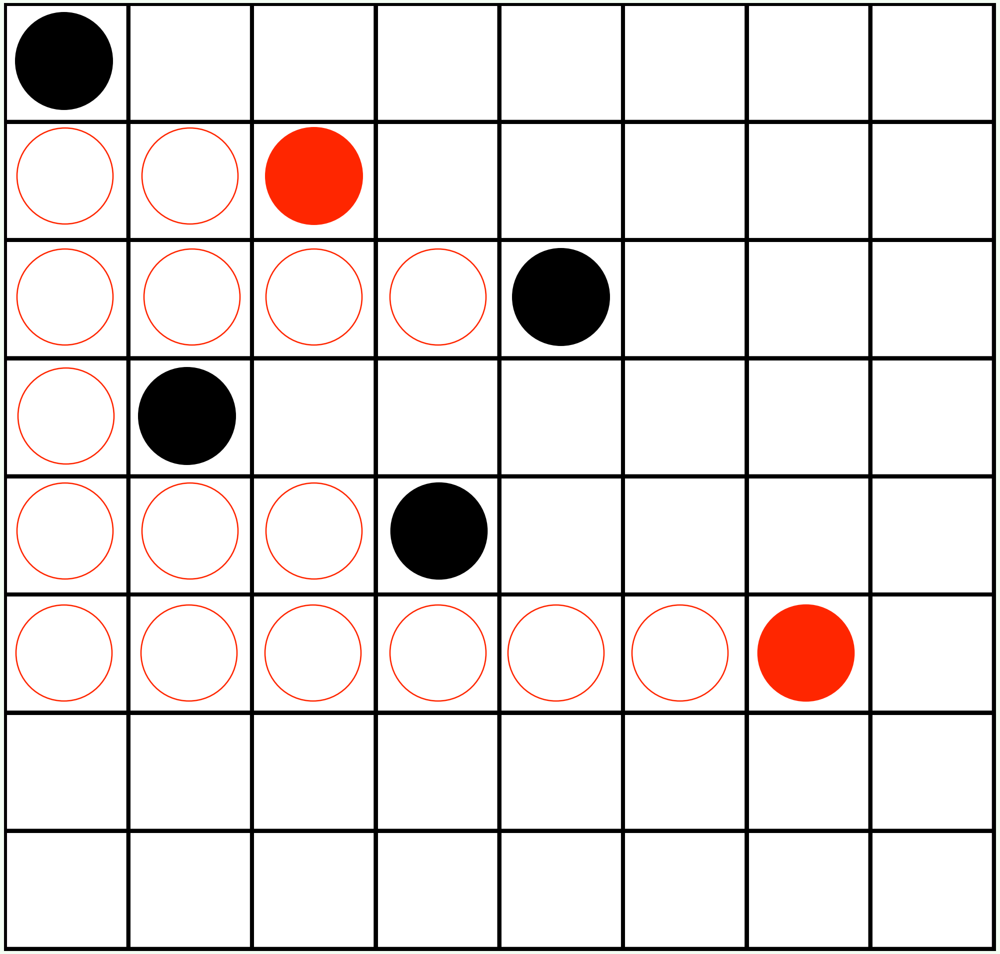
Tests 36 + 3 = 39 Backtracks 0

Backtracking



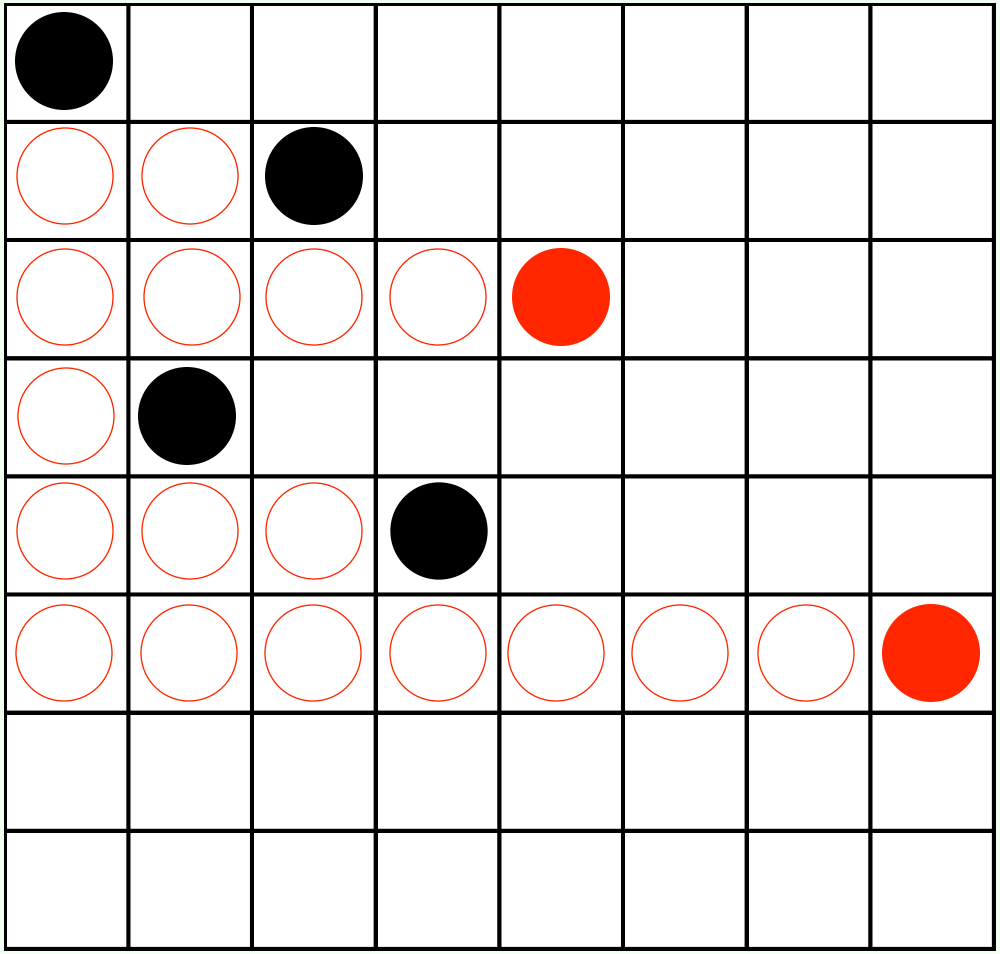
Tests 39 + 1 = 40   Backtracks 0

Backtracking



Tests 40 + 2 = 42 Backtracks 0

Backtracking

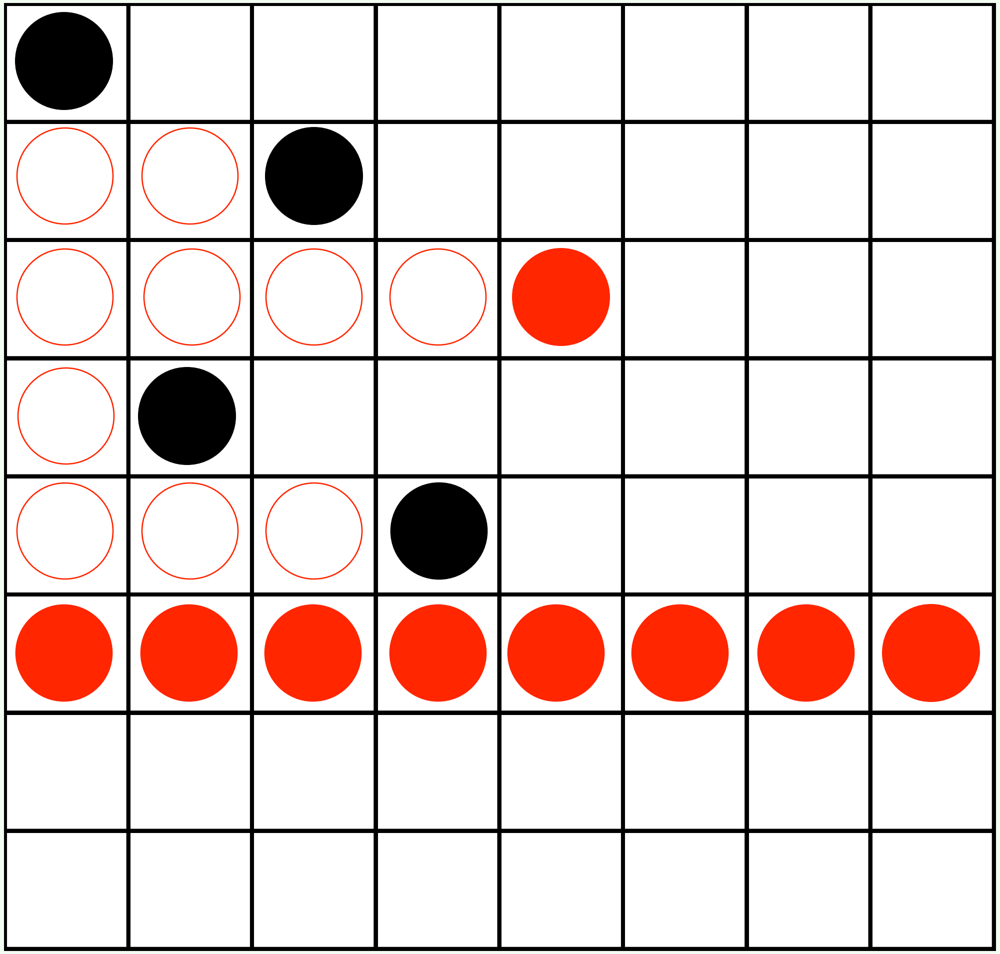


Tests 42 + 3 = 45 Backtracks 0

Backtracking

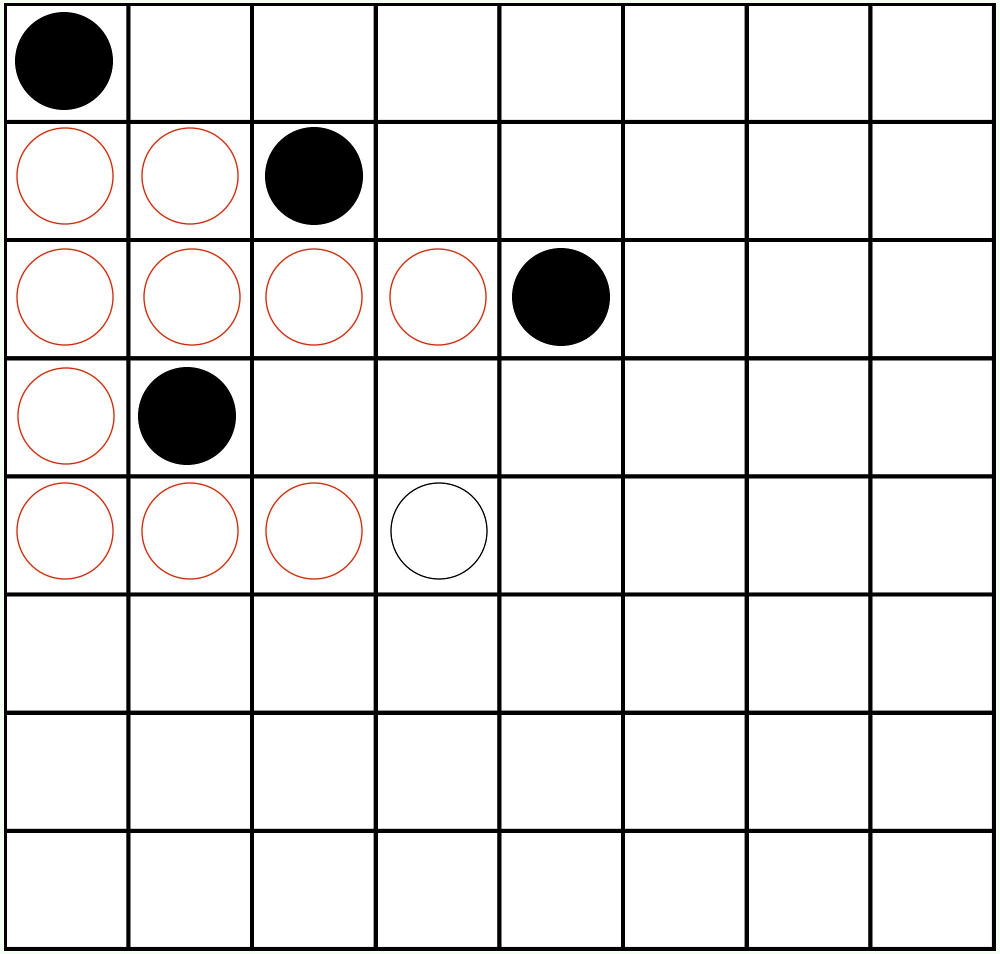
Q6 Fails

Backtracks to Q5



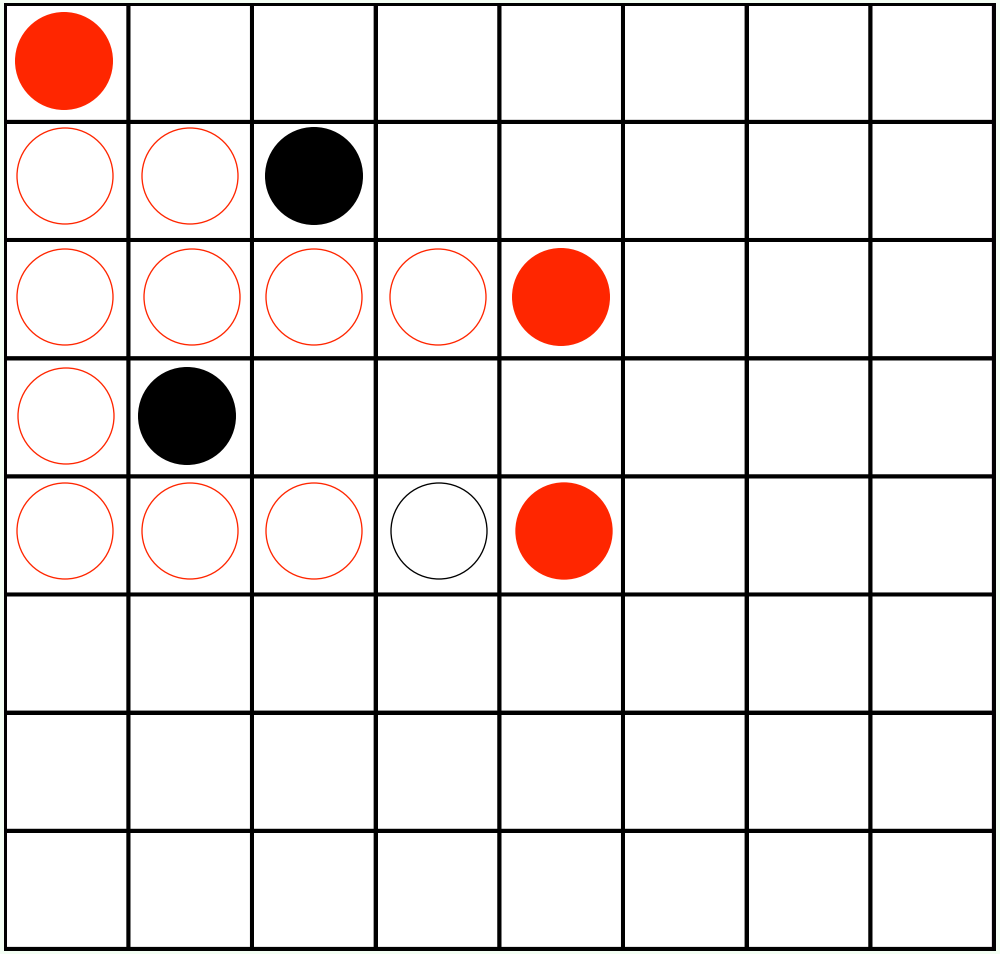
Tests 45 **Backtracks 0+ 1 = 1**

Backtracking



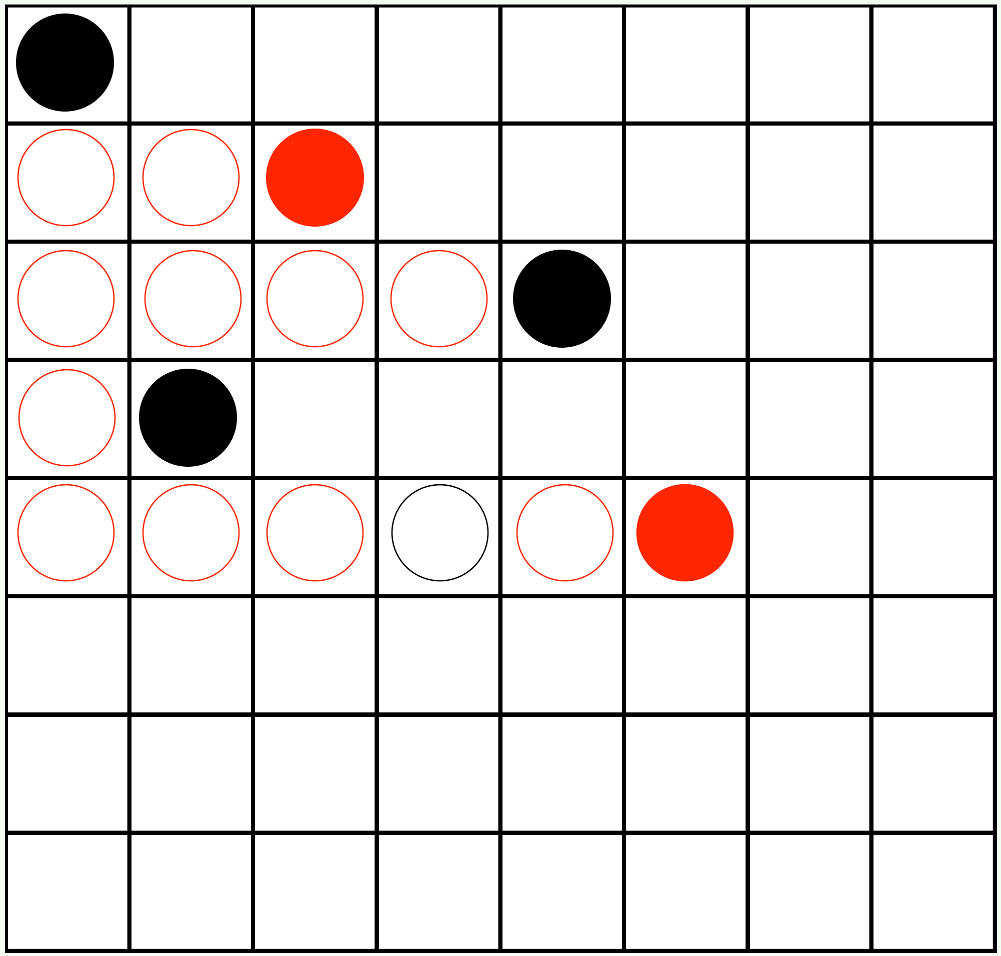
Tests 45 **Backtrackings 1**

Backtracking



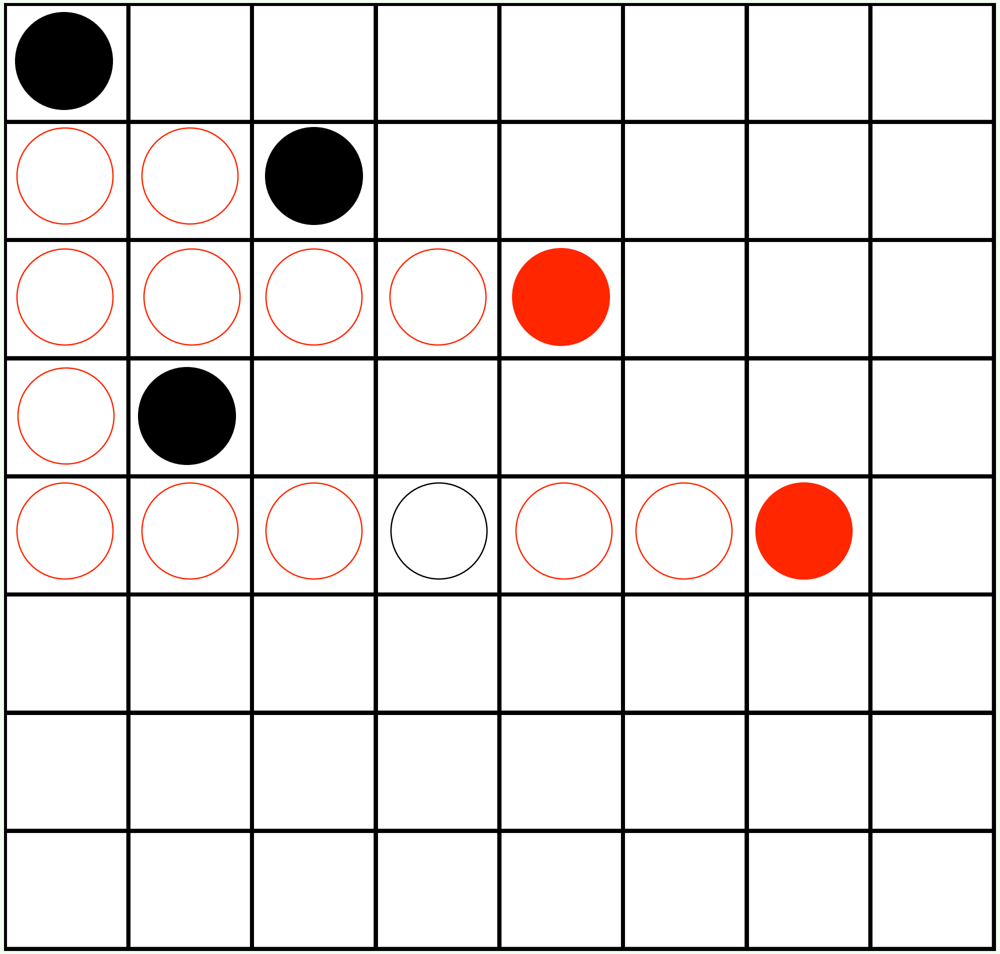
Tests 45 + 1 = 46 **Backtracks 1**

Backtracking



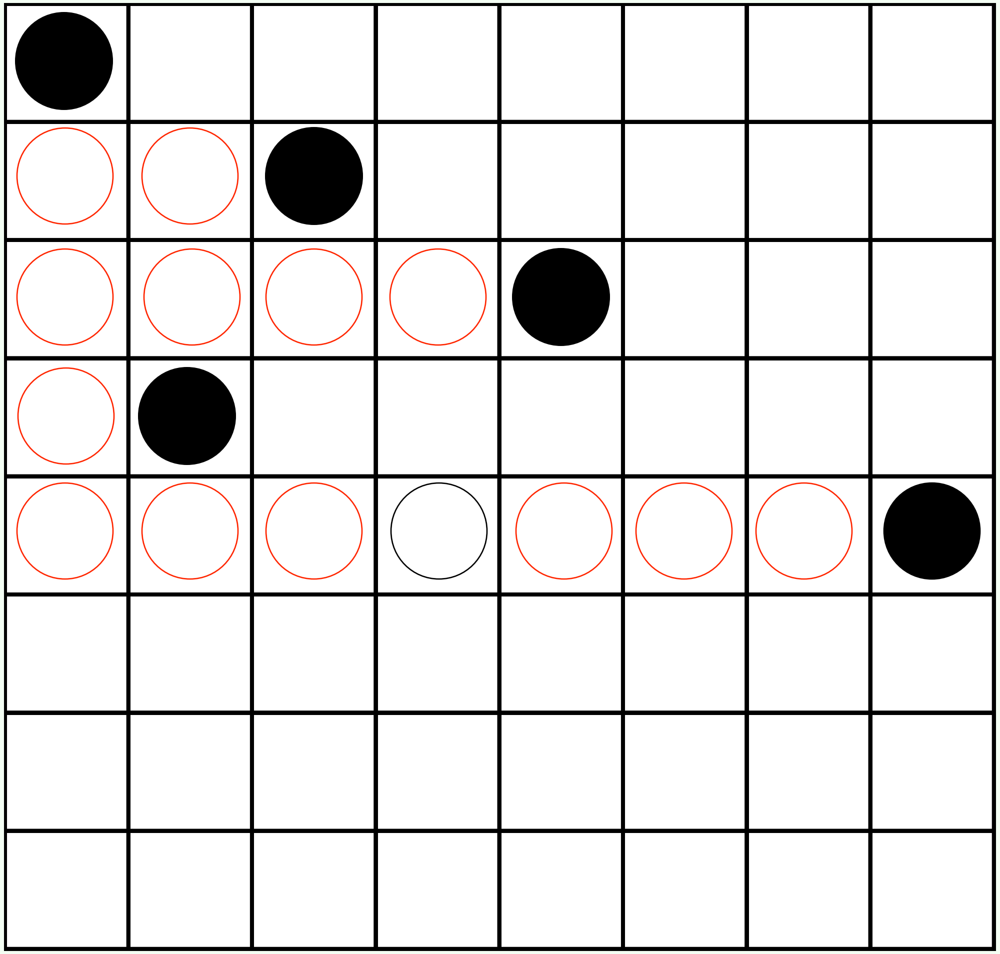
Tests 46 + 2 = 48 **Backtracks 1**

Backtracking



Tests 48 + 3 = 51 **Backtracks 1**

Backtracking

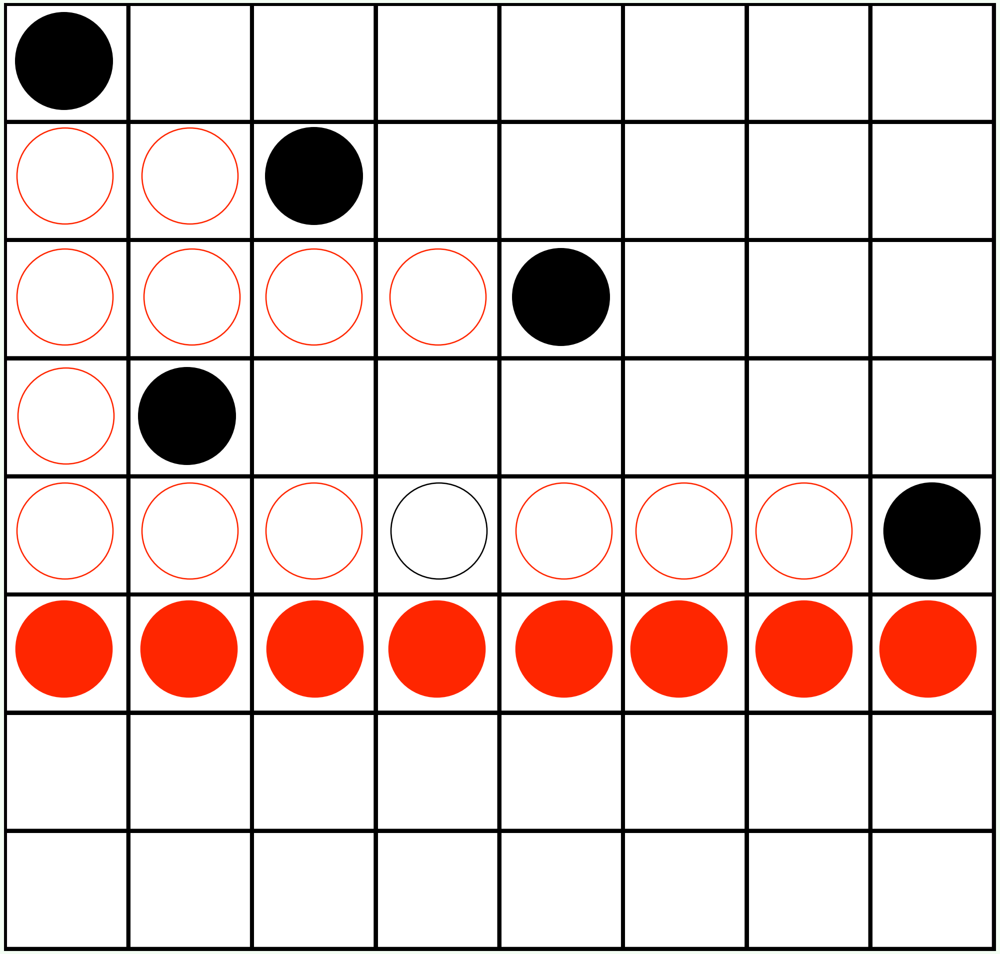


Tests 51 + 4 = 55 **Backtracks 1**

Backtracking

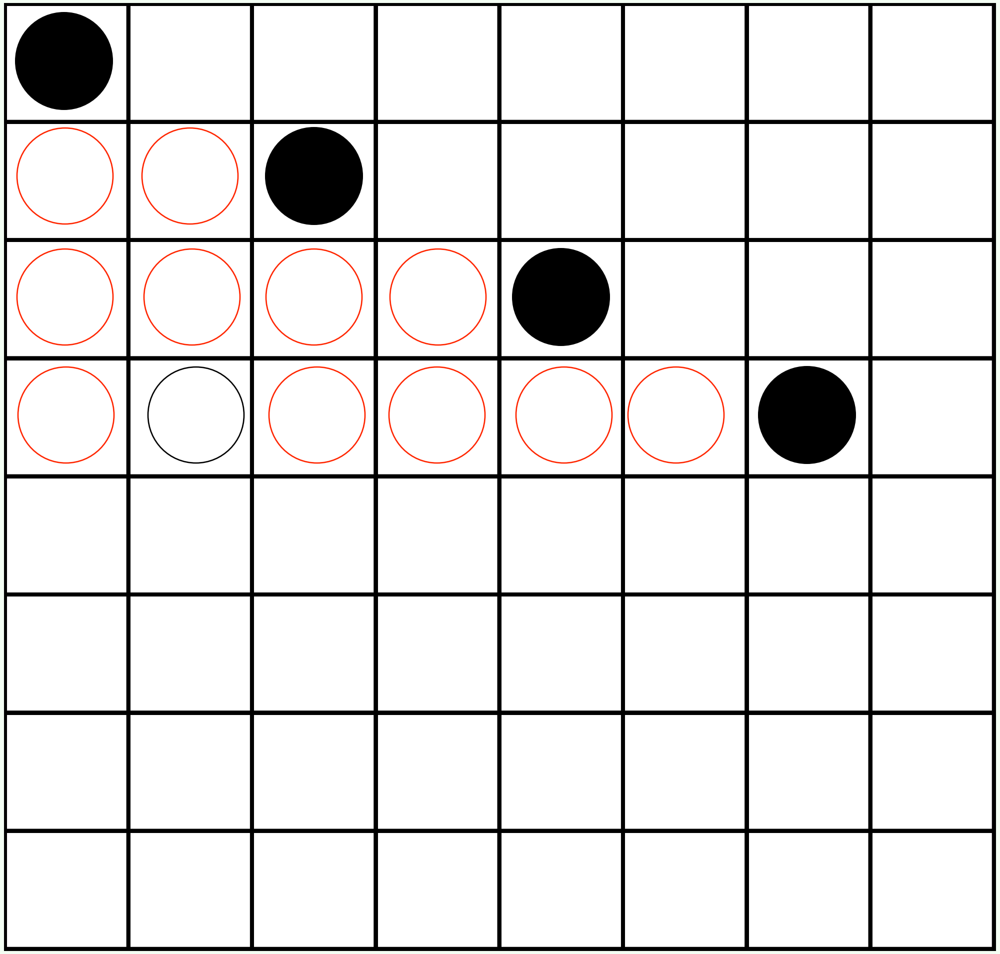
Q6 Fails

Backtracks to Q5 and next to Q4



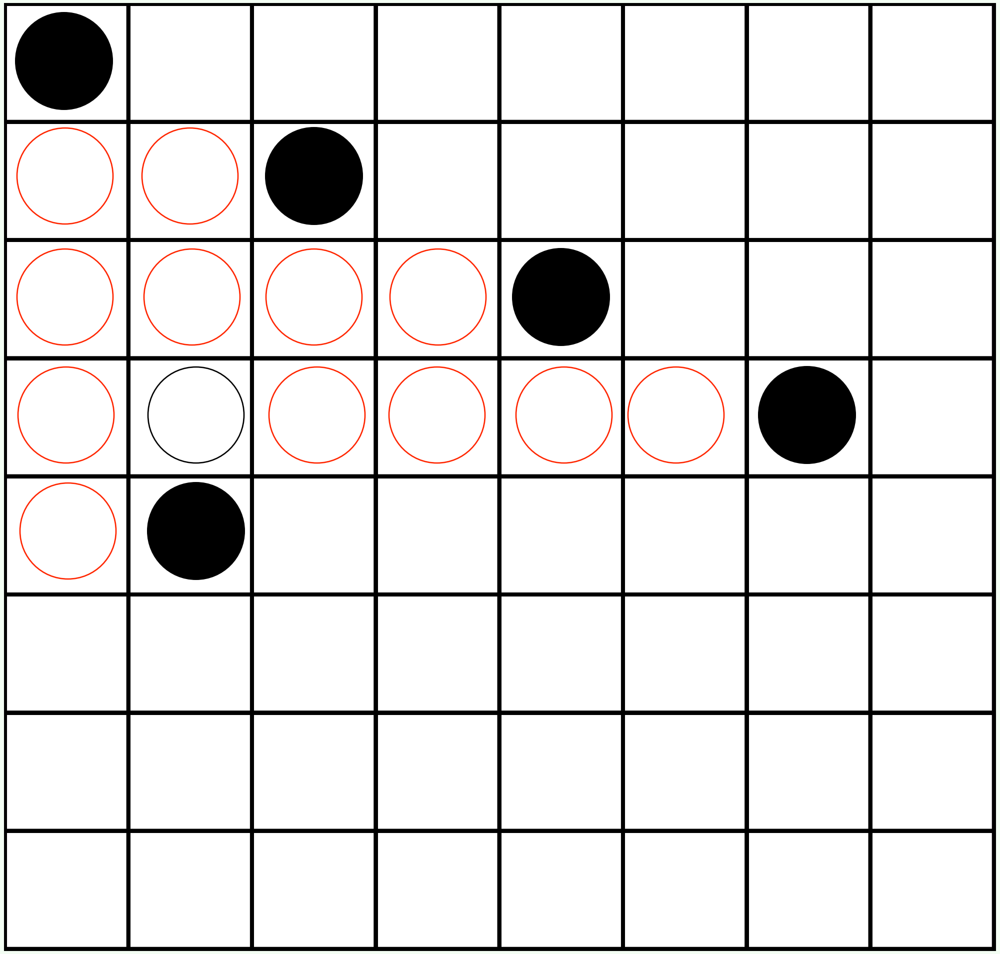
Tests 55+1+3+2+4+3+1+2+3 = 74 Backtracks 1+2 = 3

Backtracking



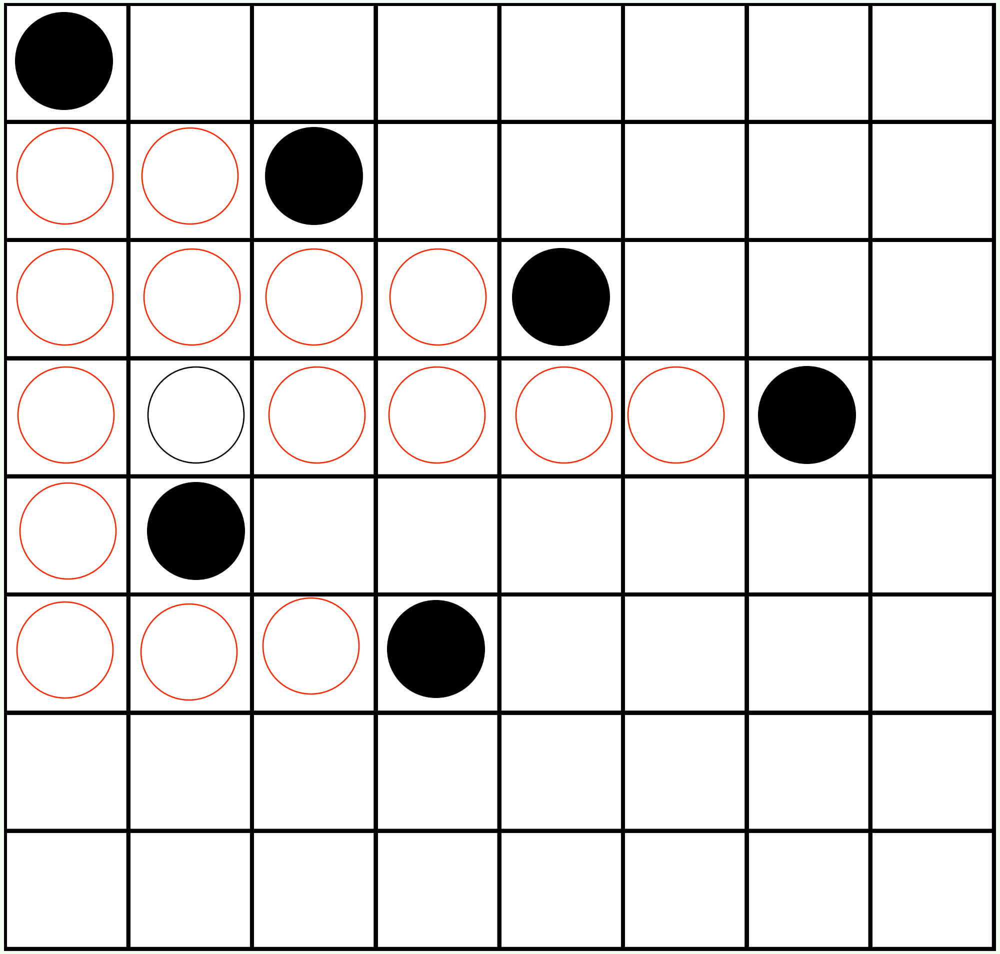
Tests 74+2+1+2+3+3= 85 Backtracks 3

Backtracking



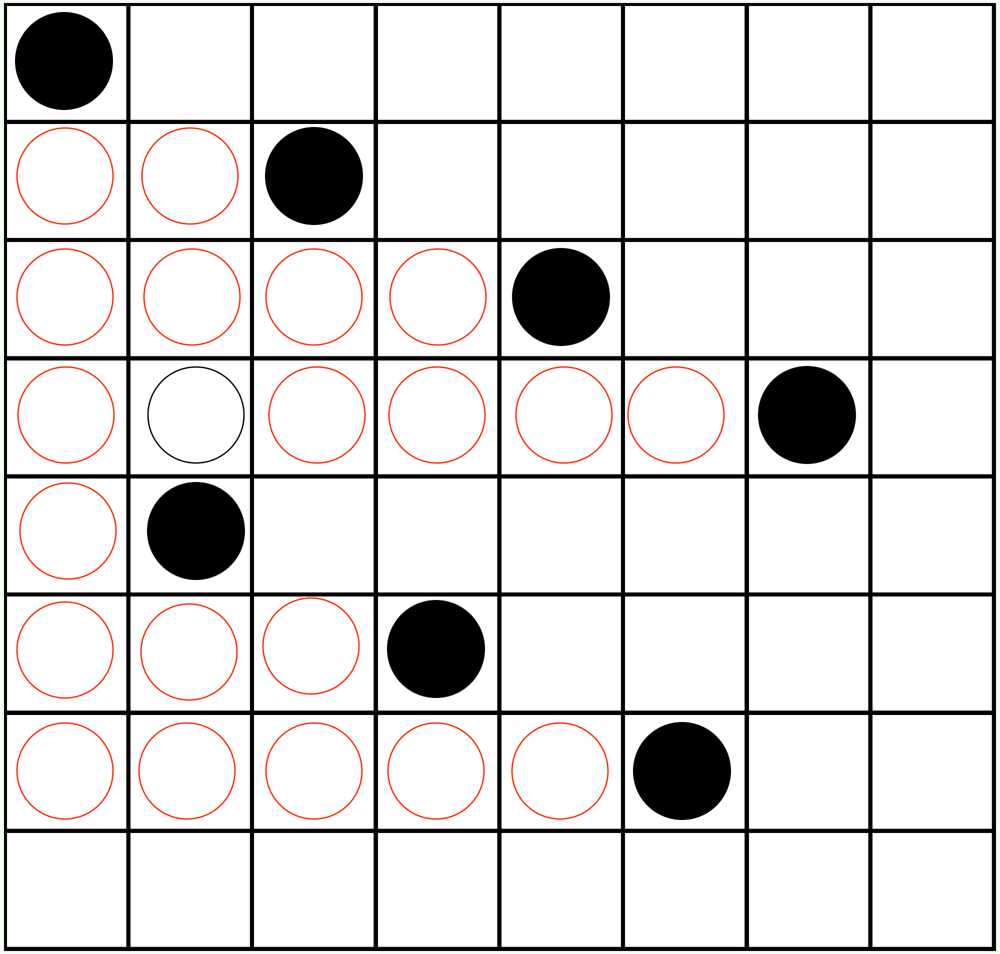
Tests 85 + 1 + 4 = 90 **Backtracks 3**

Backtracking



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

Backtracking

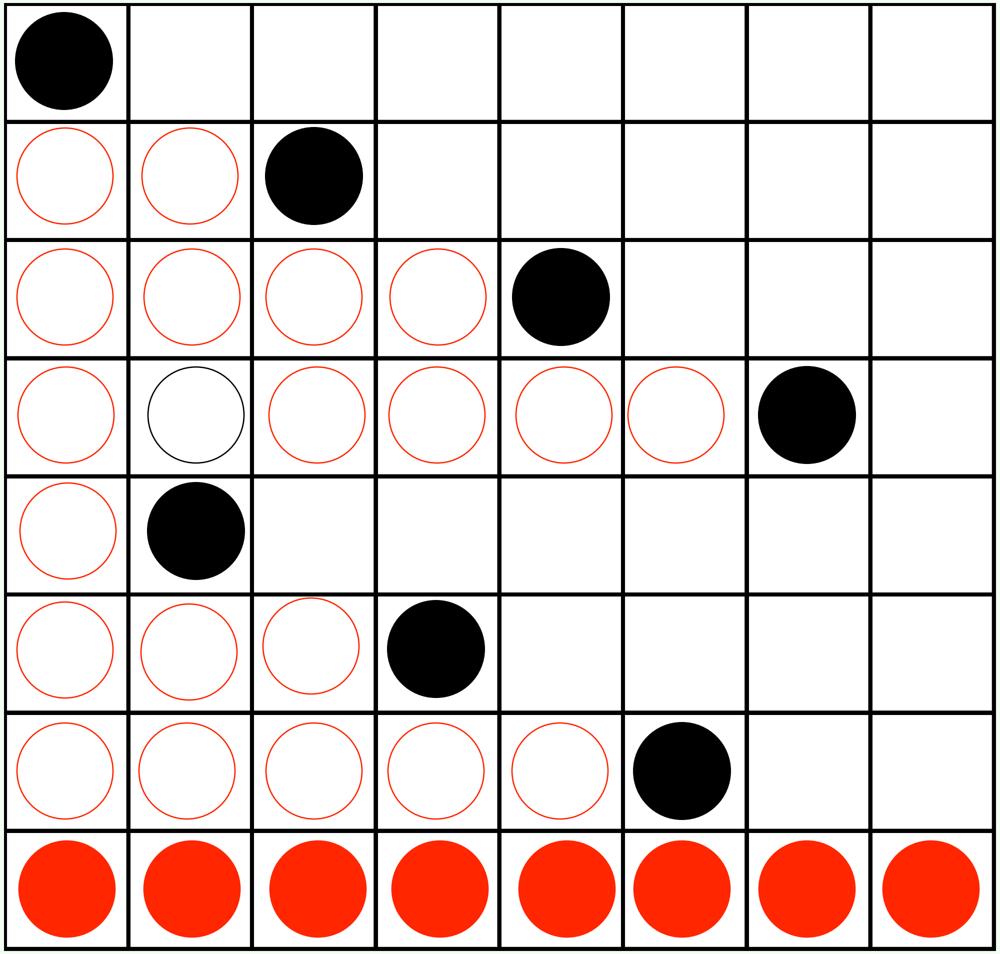


Tests 101+1+5+2+4+3+6= 122 Backtracks 3

Backtracking

Q8 Fails

Backtracks
to
Q7

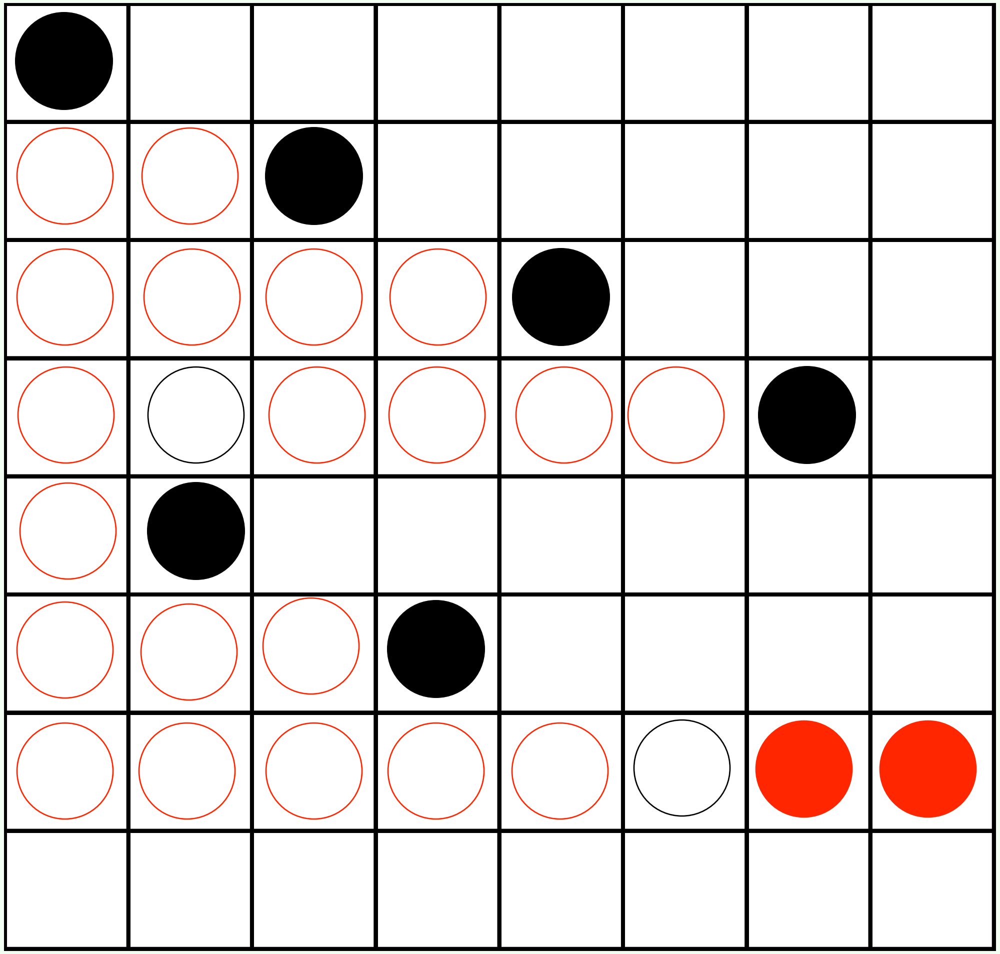


Tests 122+1+5+2+6+3+6+4+1= 150 Backtracks 3+1=4

Backtracking

Q7 Fails

Backtracks to Q6



Tests 150+1+2= 153 Backtracks 4+1=5

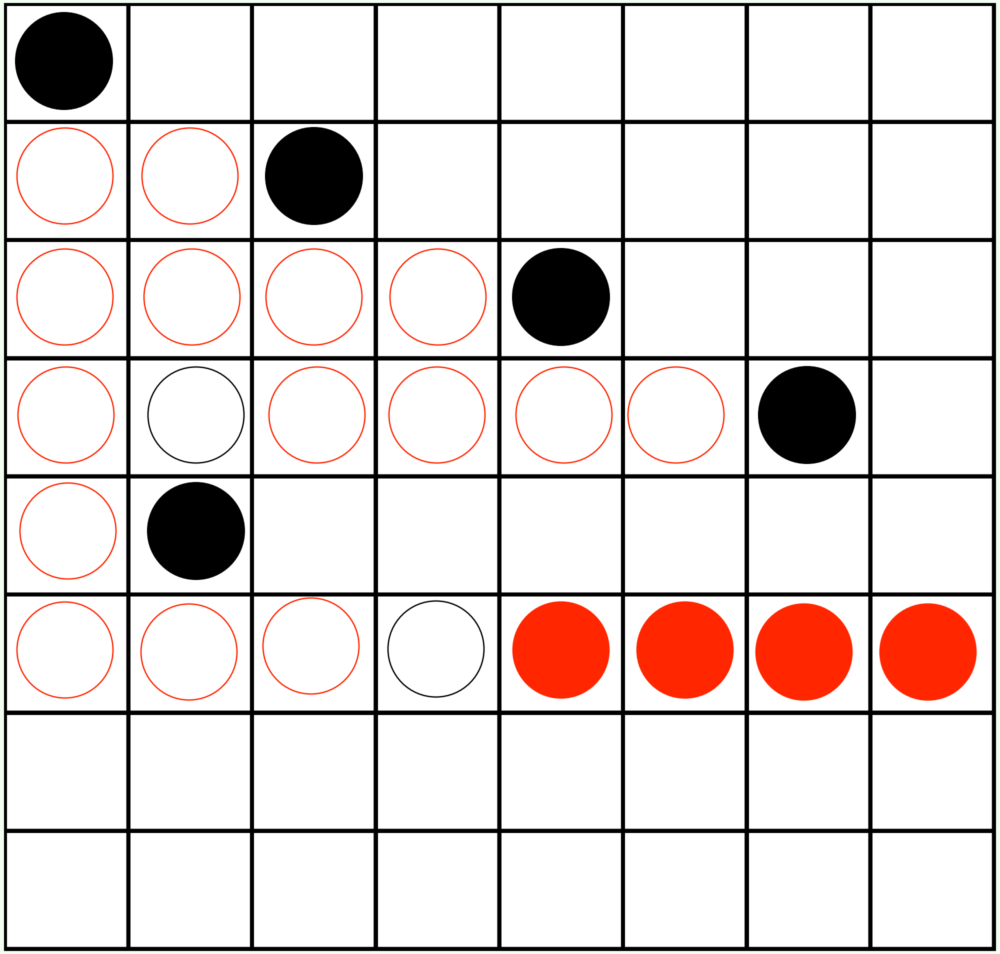
Backtracking

Q6 Fails

Backtracks

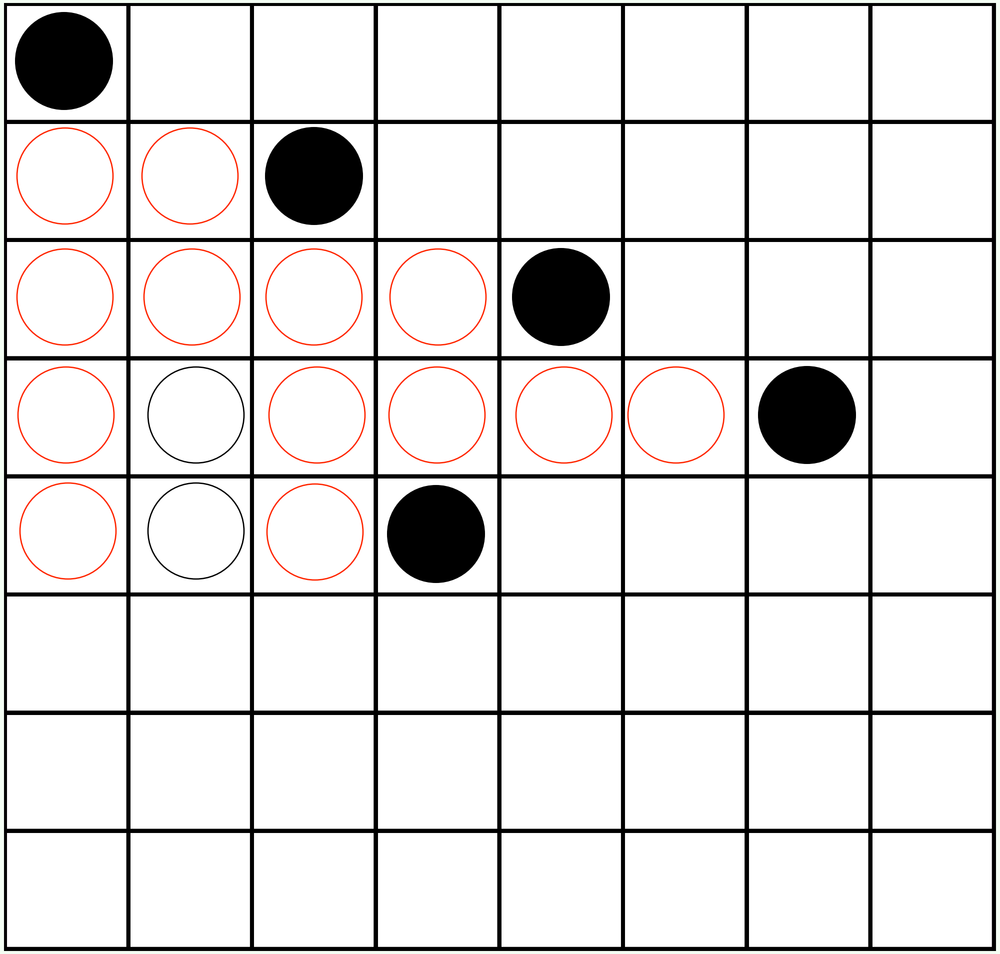
to

Q5



Tests 153+3+1+2+3= 162 **Backtracks 5+1=6**

Backtracking

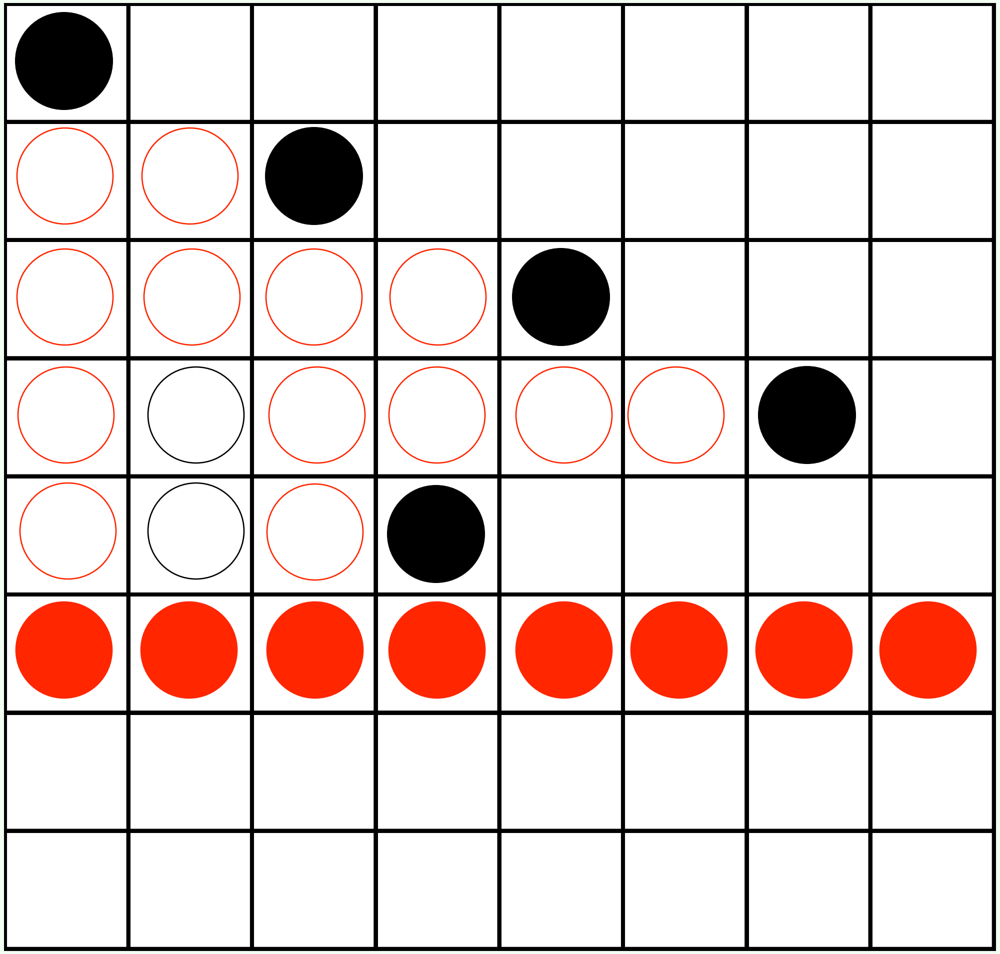


Tests 162+2+4= 168 **Backtracks 6**

Backtracking

Q6 Fails

Backtracks
to
Q5

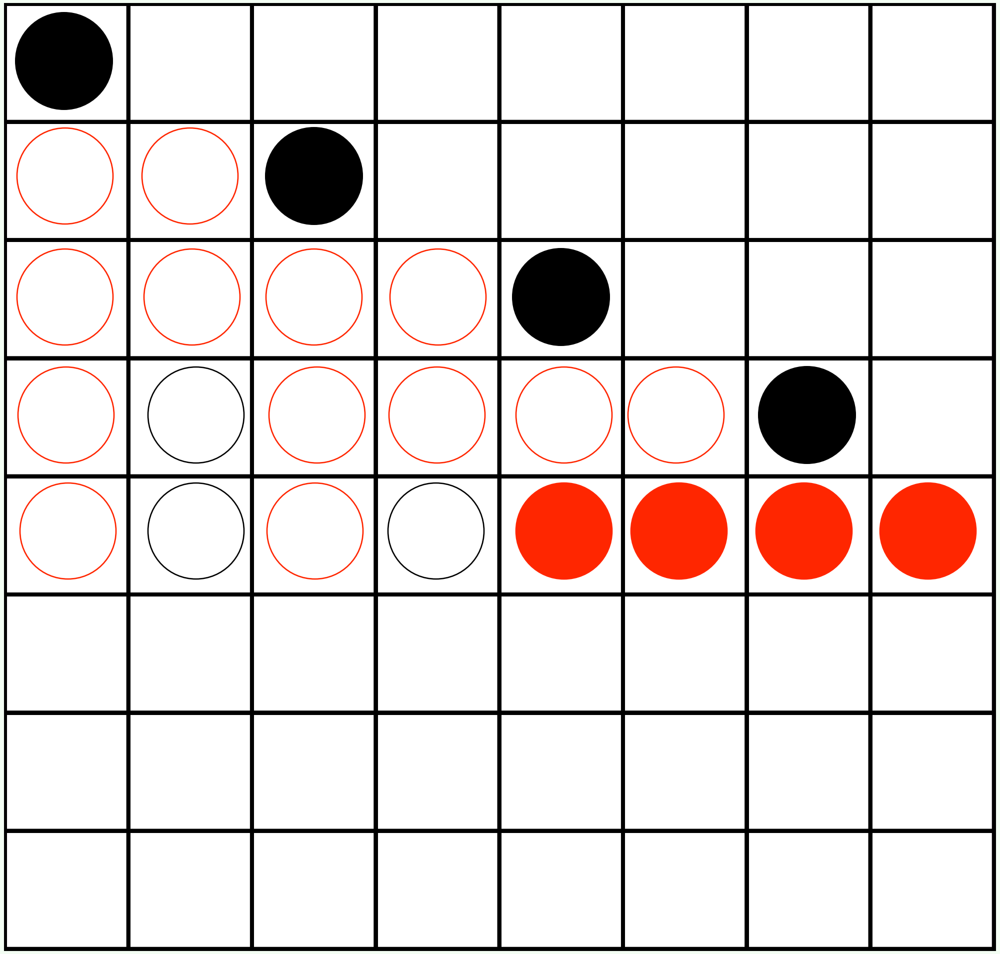


Tests 168+1+3+2+5+3+1+2+3= 188 Backtracks 6+1 = 7

Backtracking

Q5 Fails

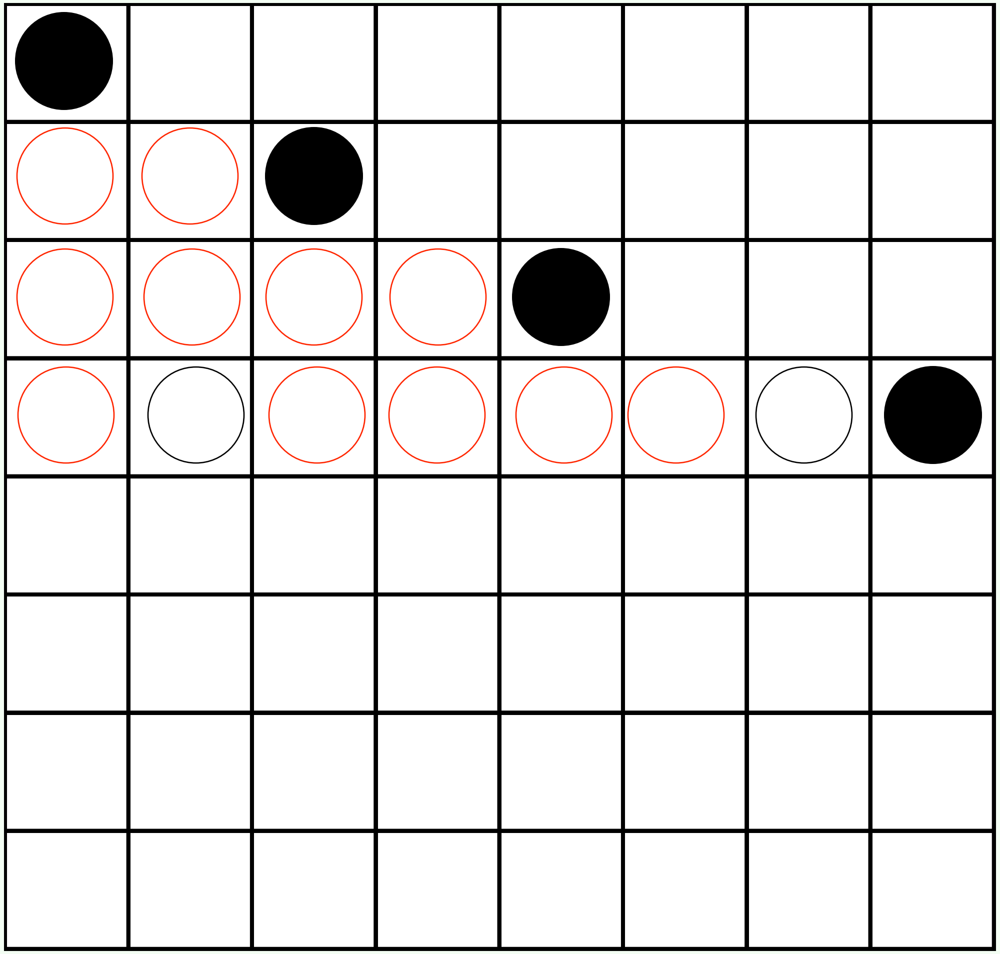
Backtracks to Q4



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

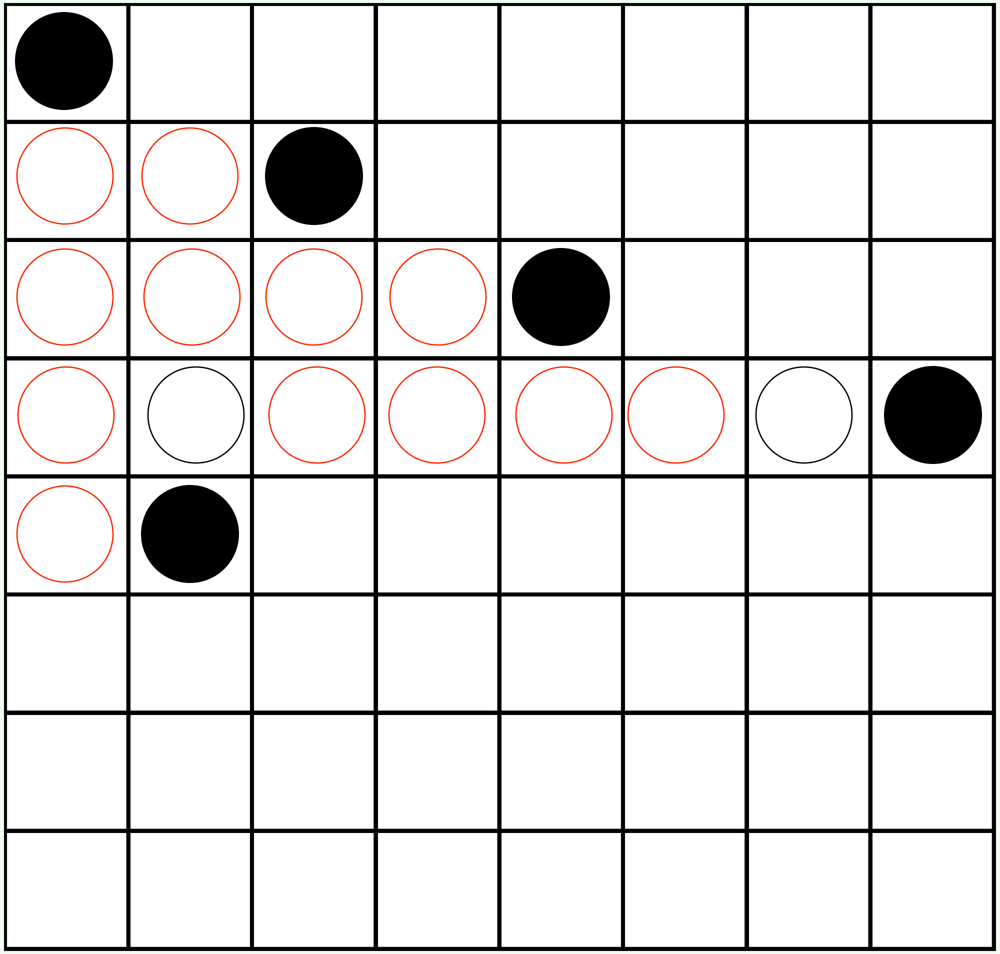
Backtracks 7+1=8

Backtracking



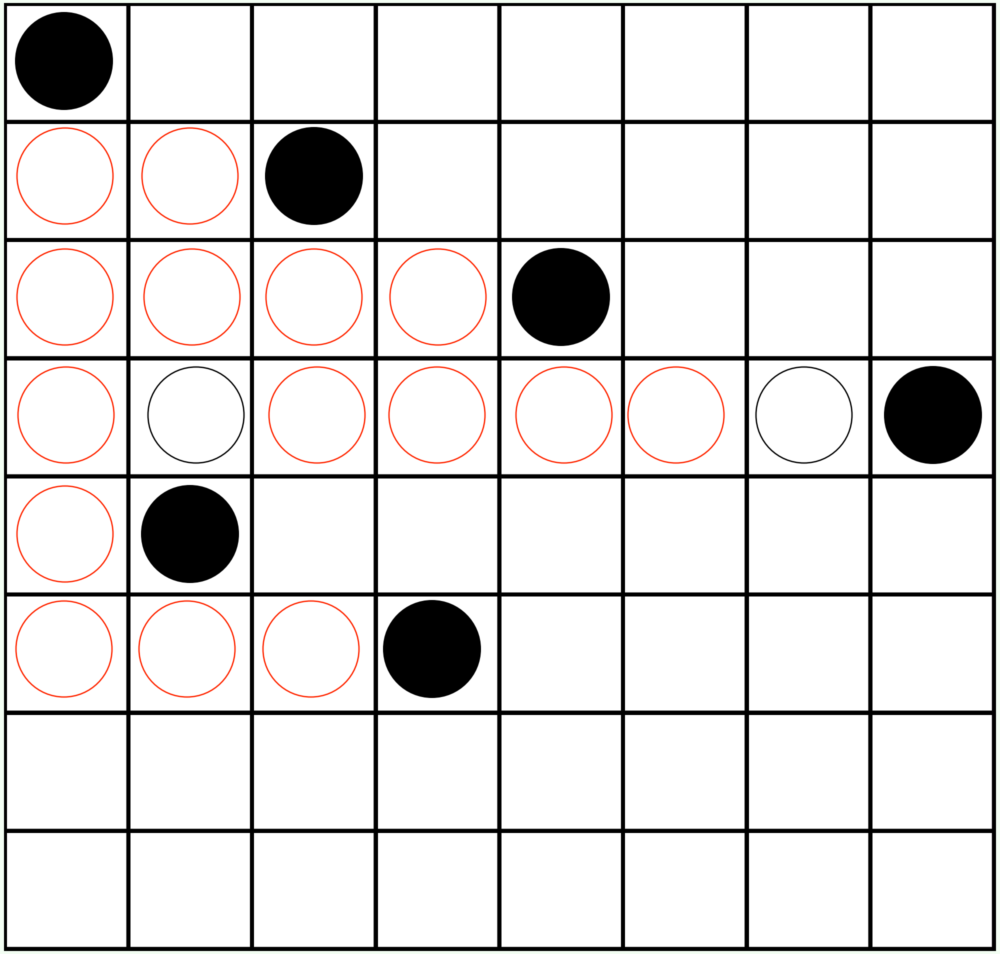
Tests 198 + 3 = 201 **Backtracks 8**

Backtracking



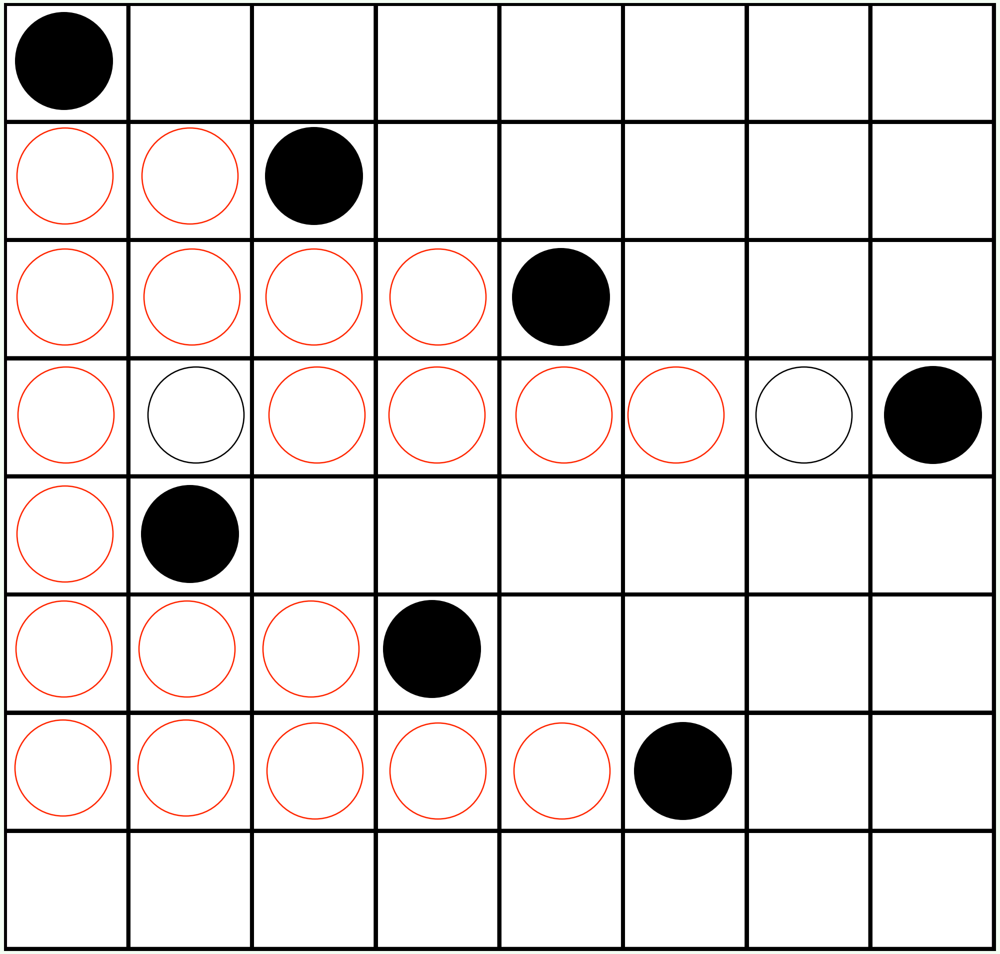
Tests 201+1+4 = 206 **Backtracks 8**

Backtracking



Tests 206+1+3+2+5 = 217 **Backtracks 8**

Backtracking

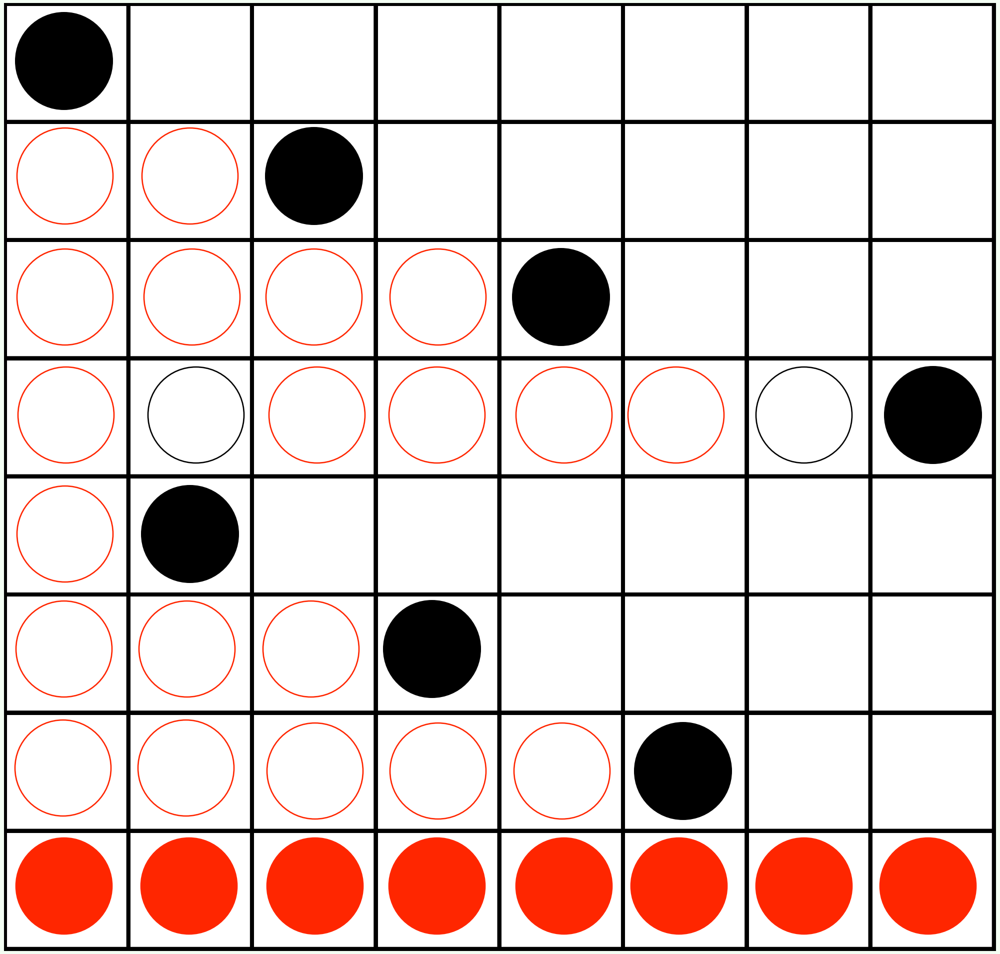


Tests 217+1+5+2+5+3+6 = 239 Backtracks 8

Backtracking

Q8 Fails

Backtracks to Q7



Tests 239+1+5+2+4+3+6+7+7= 274 Backtracks 8+1 = 9

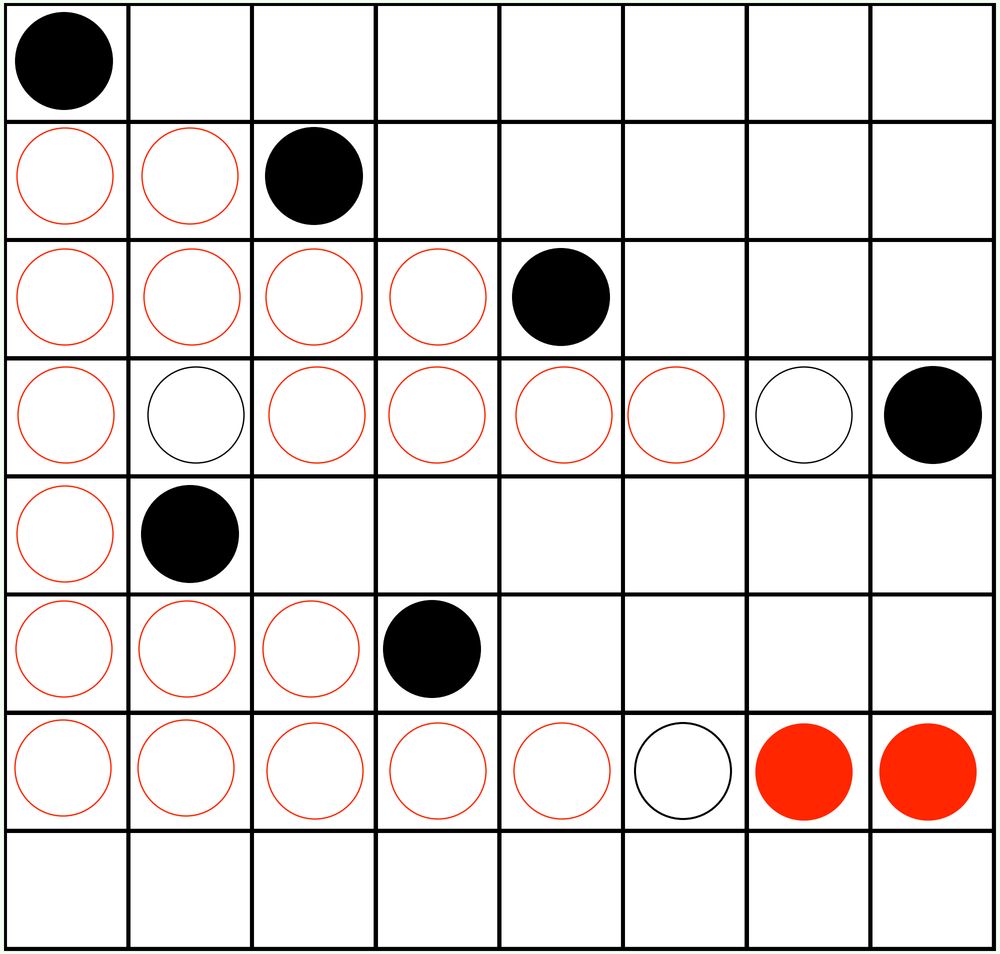
Backtracking

Q7 Fails

Backtracks

to

Q6



Tests 274+1+2= 277 **Backtracks 9+1=10**

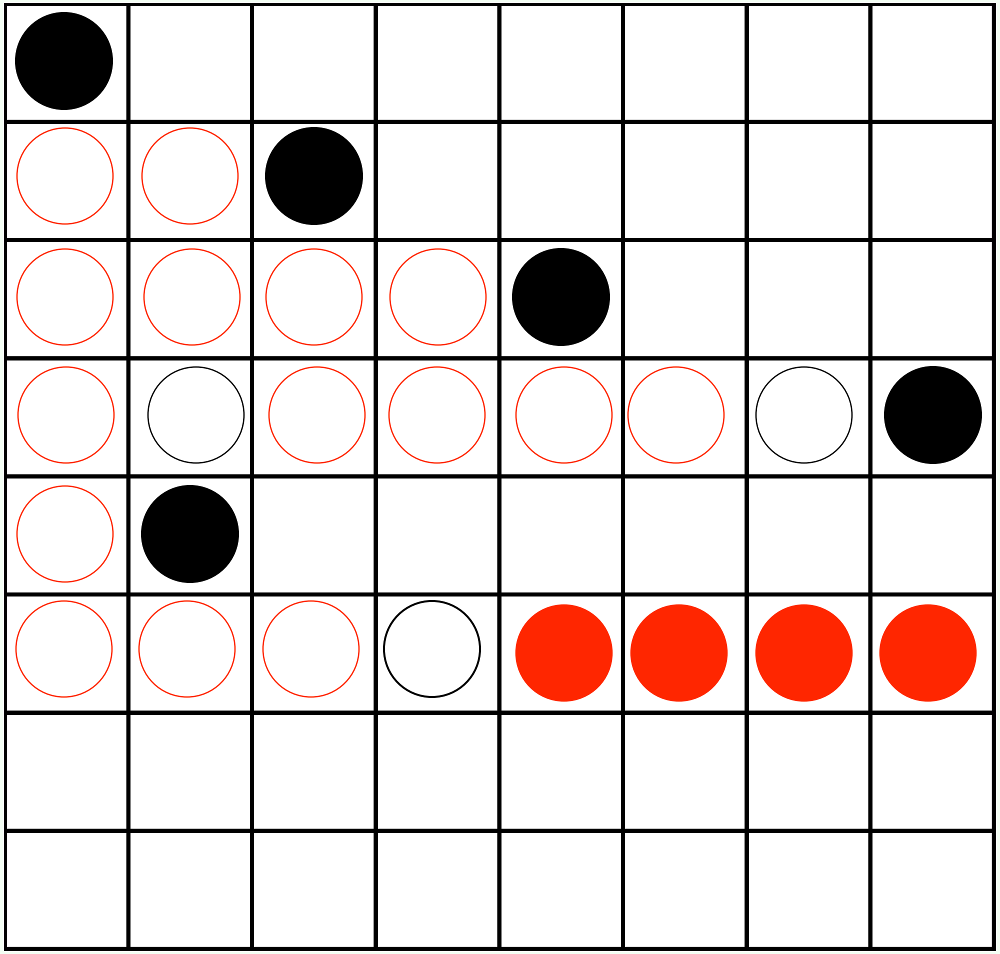
Backtracking

Q6 Fails

Backtracks

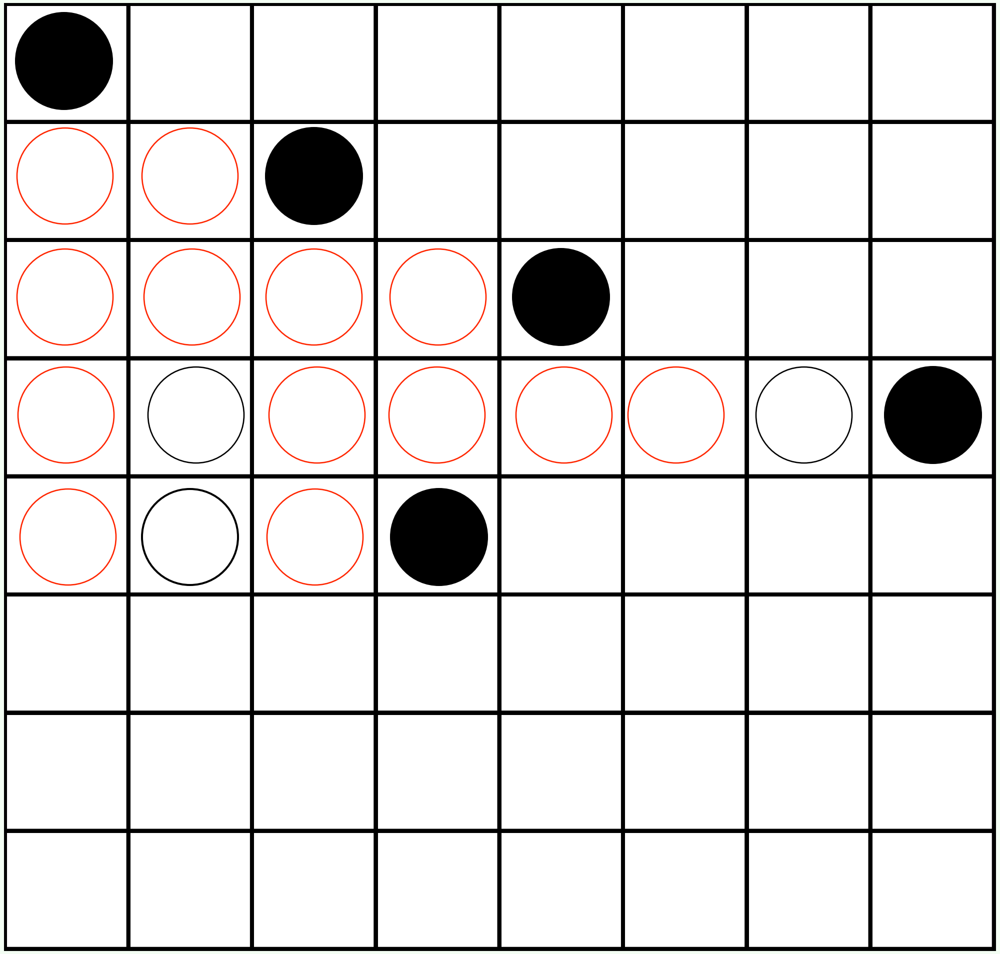
to

Q5



Tests 277+3+1+2+3= 286 Backtracks 10+1=11

Backtracking

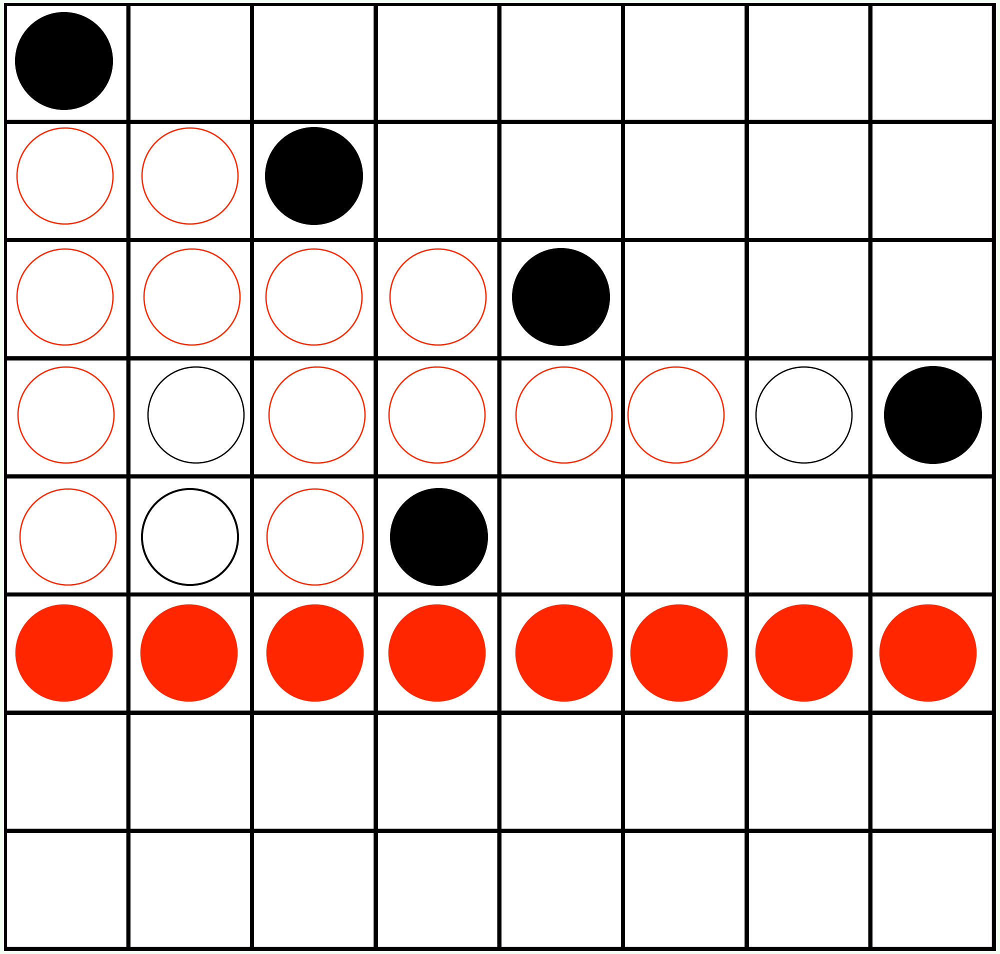


Tests 286+2+4= 292 **Backtracks 11**

Backtracking

Q6 Fails

Backtracks to Q5



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

Backtracking

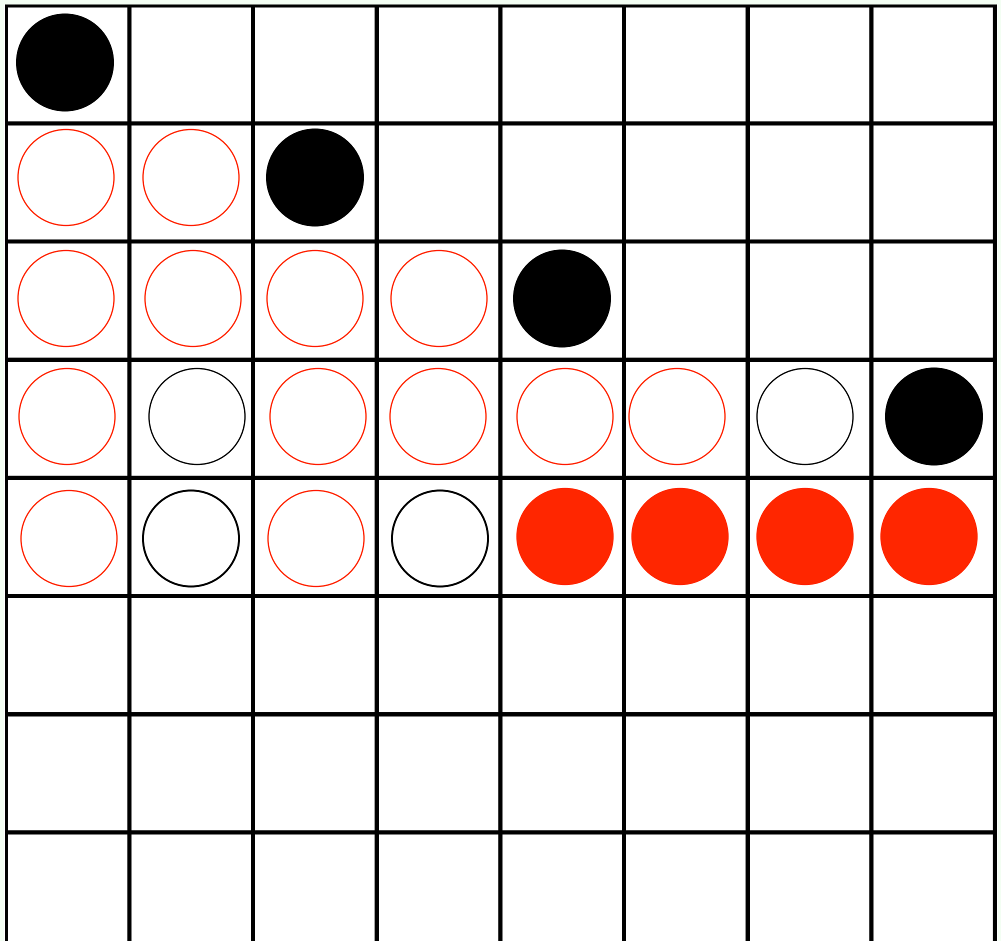
Q5 Fails

Backtracks to

Q4

and next to

Q3



Tests 312+1+2+3+4= 322 Backtracks 12+2=14

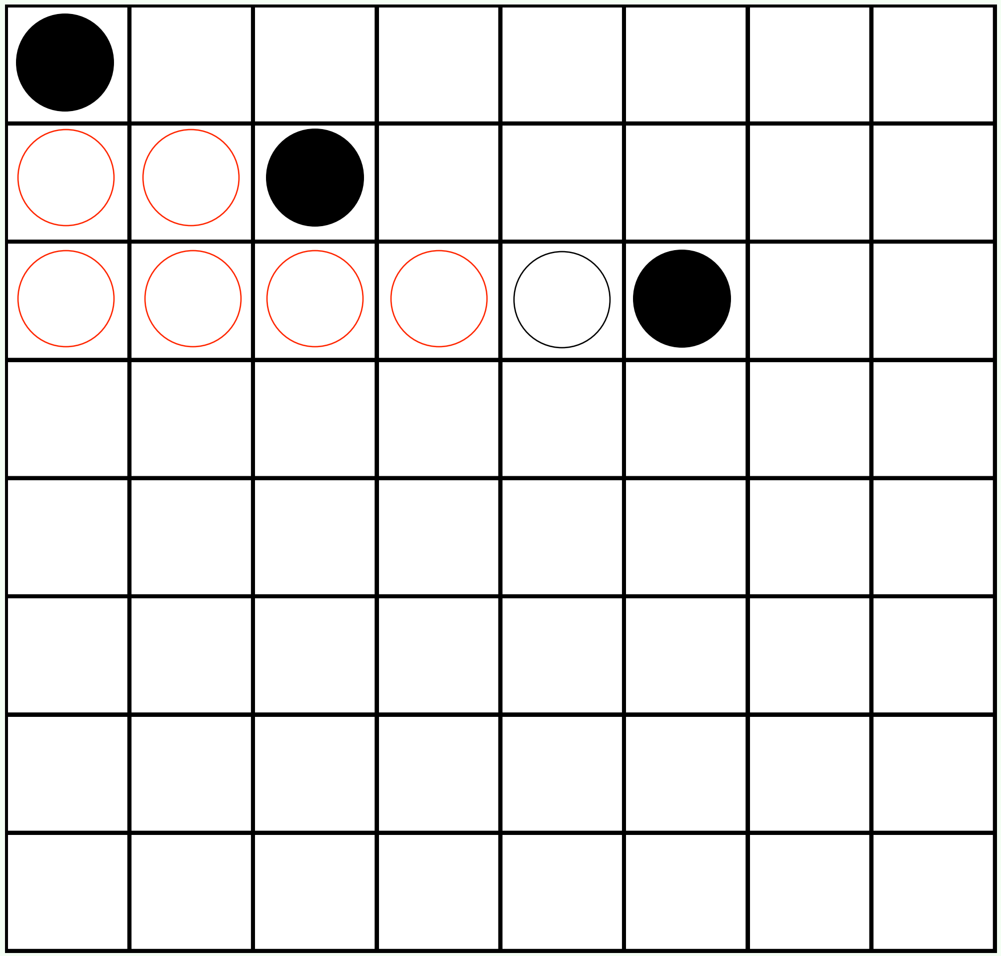
Backtracking

$Q_1 = 1$

$Q_2 = 3$

$Q_3 = 5$

Impossible !

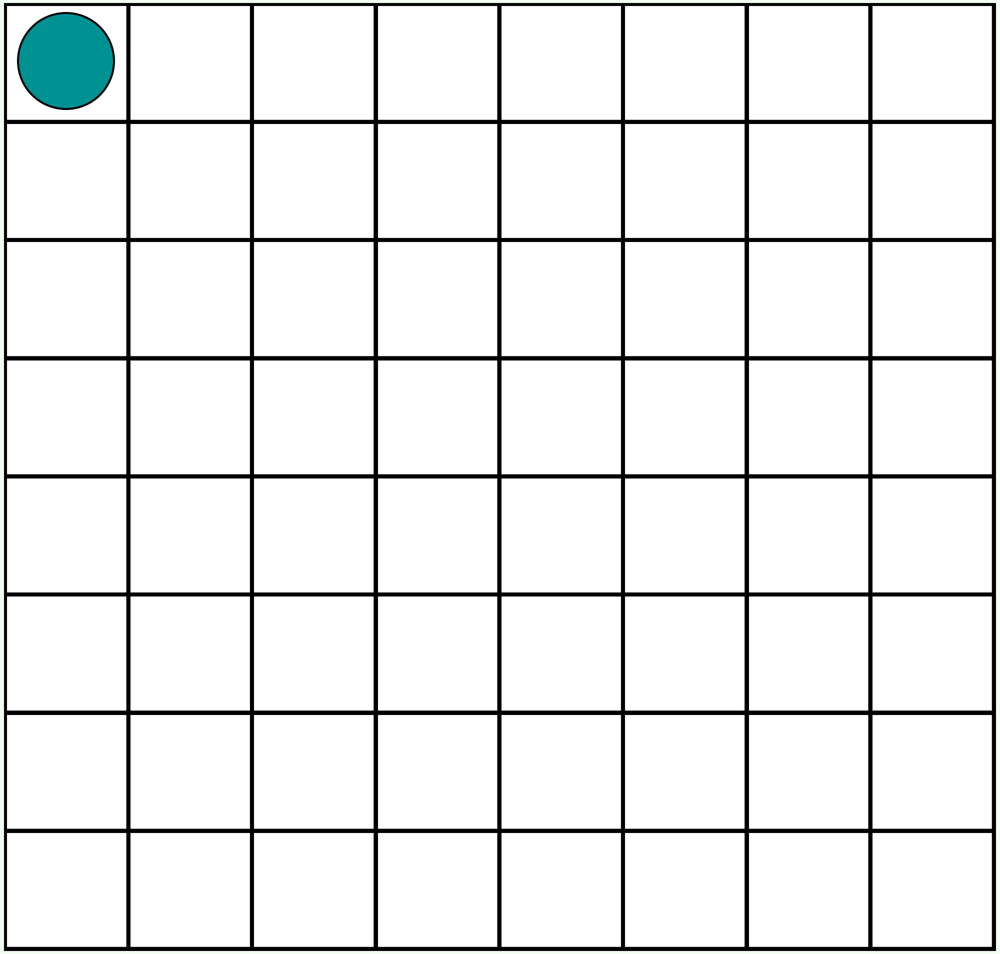


Tests 322 + 2 = 324 **Backtracks 14**

Search Methods (2) – Backtracking + Propagation

- A more efficient backtracking search strategy sees constraints as active constructs:
  - Whenever a variable is assigned a variable, the consequences of such assignment are taken into account to narrow the possible values of the variables not yet assigned.
  - If for one such variable there are no values to chose 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.
- 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.
- This procedure is illustrated again with the 8-queens problem.

Search Methods (2) - Backtracking + Propagation



Tests 0 **Backtracks 0**

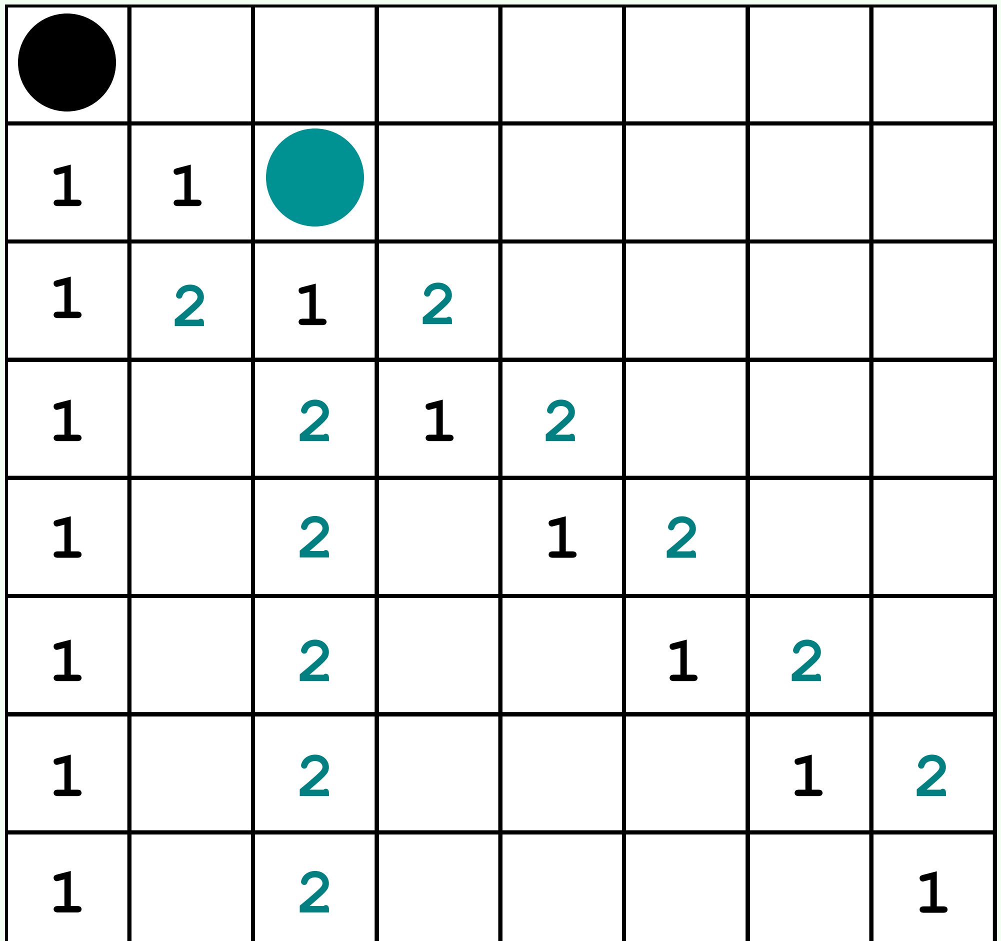
Search Methods (2) - Backtracking + Propagation

```
Q1 #\= Q2,  L1+Q1 #\= L2+Q2,  L1+Q2 #\= L2+Q1.
```

●							
1	1						
1		1					
1			1				
1				1			
1					1		
1						1	
1							1

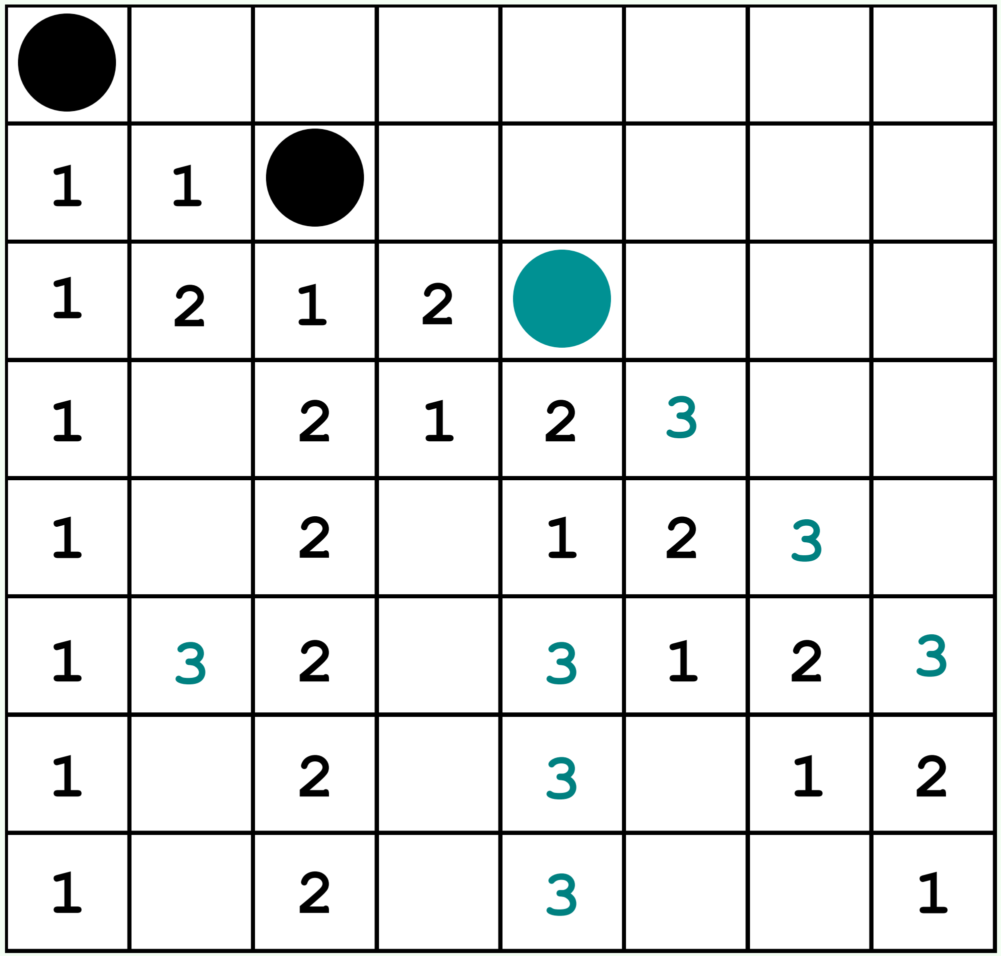
Tests 8 * 7 = 56 **Backtracks 0**

Search Methods (2) - Backtracking + Propagation



Tests 56 + 6 * 6 = 92 **Backtracks 0**

Search Methods (2) - Backtracking + Propagation



Tests 92 + 21 = 113 **Backtracks 0**

Search Methods(2a) – B+P w/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 – polinomial 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.
- This procedure is again illustrated with the 8-queens problem.

Search Methods(2a) - B+P w/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**

Search Methods(2a) - B+P w/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**

Search Methods(2a) - B+P w/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

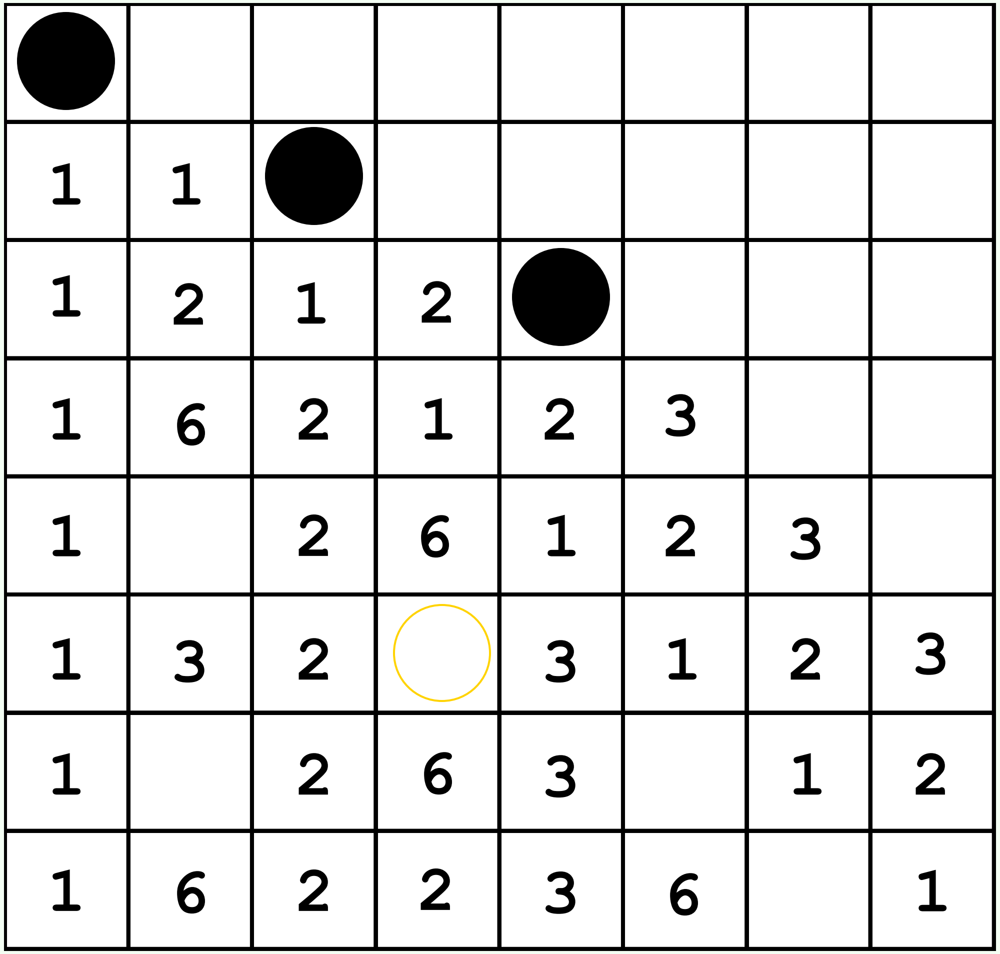
Tests 113+3+3+3+4 = 126 **Backtracks 0**

Search Methods(2a) - B+P w/Heuristics

Q_8

may only take value

7



Tests 126 **Backtracks 0**

Search Methods(2a) - B+P w/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**

Search Methods(2a) - B+P w/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**

Search Methods(2a) - B+P w/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**

Search Methods(2a) - B+P w/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**

Search Methods(2a) - B+P w/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**

Search Methods(2a) - B+P w/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**

Search Methods(2a) - B+P w/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**

Search Methods(2a) - B+P w/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**

Search Methods(2a) - B+P w/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**

Search Methods(2a) - B+P w/Heuristics

Q_7

may take NO value

Failure!

Backtracks

... to Q_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

Tests 136 **Backtracks 0+1=1**

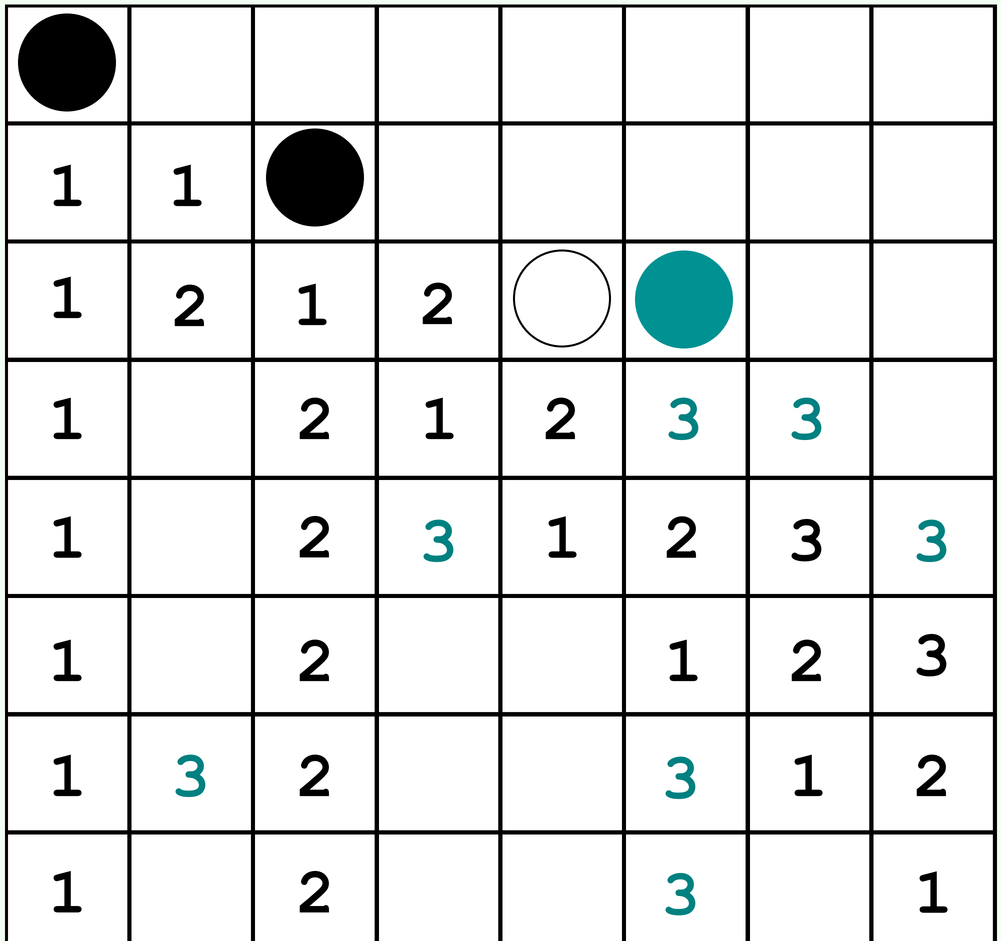
Search Methods(2a) - B+P w/Heuristics

$Q_1 = 1$

$Q_2 = 3$

$Q_3 = 5$

Impossible !



Tests

136

(324)

Backtracks

1

(14)

Tests 136 **Backtracks 1**

Search Methods – B+P w/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

- Before addressing concepts and definitions we will informally see how this type of applications can be programmed in COMET.

- COMET is an Object Oriented language, with a syntax similar to JAVA, but with special classes and methods to deal with
  - **CP** - Contraint Programming; and
  - **LS** - Constrained Local Search

- In COMET, a CSP (Constraint Satisfaction Problem) is typically solved in **CP** with a program with the following structure

```
import cotfd;

Solver<CP> cp();
   //declare the variables
solve<cp> {
   //post the constraints }
using {
   //non deterministic search }
```

Constraints: Basic Concepts

- Solver<CP> is a class with methods to associate variables and constraints as well as nondeterministic search. The constraints are declared within the solve<cp>{ } section.

```
solve<cp> { // post the constraints  }
```

- In this case, only a **single** solution is obtained for the CSP. There are two alternatives for this section:

- To obtain **all** solutions of a CSP problem:

```
solveall<cp>   {post the constraints }
```

- To obtain an **optimal** solution of a CSOP (Constraint Satisfaction and Optimisation Problem)

```
minimize<cp>
   //expression or variable
subject to
   {  //post the constraints }
```

Constraints: Basic Concepts

- Variables are objects, declared by identifying their
  - Type,
  - Domain, and
  - Associated solver
- We will be mostly concerned with Finite Domain (FD) variables, whose type is var<CP>{int}, and have a domain that restricts the values that can appear in a solution of the problem.
- Typically the domain is defined as a range of integers, as in

```
var<CP>{int} x(cp,1..10);
```

- Alternatively, the domain can be a set of integers

```
set{int} dom = {1,3,7};
var<CP>{int} y(cp,dom);
```

- Ranges are defined over integers, sets over integers or enumerated

```
enum country = {Belgium, USA, France, Portugal};
var<CP>{country} z(cp,country);
```

Constraints: Basic Concepts

- FD Boolean variables are a special case of FD variables. They could be regarded as numeric 0/1 Fd variable (and are often recast as such) but have different syntax. As expected, Its domain is the set {false, true}.

```
var<CP>{bool} b(cp);
```

- In most cases, it is convenient to organize FD (or basic) variables in array data structures.

```
range Rng = 1..5;
range Dom = 1..10;
var<CP>{int} a[Rng](cp,Dom);
```

- As expected array data structures are usually associated with loop constructs for flow control, namely the forall construct.

Constraints: Basic Concepts

- Many types of constraints are defined in the language as primitives. They belong to the class constraint and are declared in a solver with its post method.

- The most common constraints are arithmetic constraints, imposing a relation (==, !=, >, >=, <, <=) on arithmetic expressions built over CP and basic variables and values with the arithmetic operators +, -, *, / (integer division) and % (modulo) .

```
int a = 4;
cp.post( x-a > y+2) ;
```

- Usually, a problem is defined as a conjunction of constraints. Nevertheless other logical combinations of constraints are often possible to define, not only conjunctive, but also disjunctive, conditional and equivalence constraints).

```
int a = 4;
cp.post( (x > y) && (x > z)) ;
cp.post( (x > y) || (x > z)) ;
cp.post( (x > y) => (x > z)) ;
cp.post( (x > y) == (x > z)) ;
```

Constraints: Basic Concepts

- COMET supports standard operators, such as **if**, **for** and **while**, along with more advanced loop control capabilities, namely the **forall** construct.

- Note that **if**, **while** and **for** conditions must be decided at compile time, and may not contain FD variables. So the following snipet is valid

```
var<CP>{int} a (cp,Dom);
int i = 1;
if (i <= n)  cp.post(a == i+1);
else         cp.post(a == i-1);
```

… but not

```
var<CP>{int} a (cp,Dom);
var<CP>{int} b (cp,Dom);
if (b <= n)  cp.post(a == i+1);
else         cp.post(a == i-1);
```

- The reason is simple: The constructs are meant to post the relevant constraints, and these must be determined before a solution is obtained (because it depends on the constraints that were posted!).

Constraints: Basic Concepts

- Of course, conditional constraints may be used for this purpose. Instead of the invalid declaration

```
var<CP>{int} a (cp,Dom);
var<CP>{int} b (cp,Dom);
if (b <= n)   cp.post(a == i+1);
else          cp.post(a == i-1);
```

… a valid declaration obeying to the same "logic" can be made with conditional constraints

```
var<CP>{int} a (cp,Dom);
var<CP>{int} b (cp,Dom);
cp.post(  (b <= n) => (a == i+1) );
cp.post( !(b <= n) => (a == i-1) );
```

Constraints: Basic Concepts

- The forall construct in COMET can be associated to *universal quantification* and is usually used with array data structures, as in

```
var<CP>{int} a[Rng] (cp,Dom);
forall (i in Rng) cp.post(a[i] == …);
```

- Special aggregation operators (sum and prod) also exist implementing the corresponding mathematical operations. For example,

```
var<CP>{int} a[1..10] (cp,Dom);
cp.post( x == sum(i in 1..10) a[i]);
```

… is equivalent but more efficient than the *iterated sum* below

```
var<CP>{int} a[1..10] (cp,Dom);
var<CP>{int} s[2..10] (cp,Dom);
cp.post( s[1] == a[1]);
forall(i in 2..10) (s[i] = s[i-1]+ a[i]);
cp.post( x == s[i]);
```

Constraints: Basic Concepts

- Many useful constraints are not easy to decompose into simpler arithmetic and logical constraints.
- Even when they are, there are some specialised algorithms that achieve better propagation.
- These are usually known as Global Constraints, and COMET supports a number of those that have been proposed in the literature:
  - Element
  - Table
  - Alldifferent
  - Cardinality
  - Knapsack
  - Circuit
  - Sequence
  - Stretch
  - Regular
  - Cumulative

Constraints: Basic Concepts

- We finish this brief introduction to COMET with the nondeterministic search that occurs in the using {…} section.
- In this section a non-deterministic search is declared, where alternative values for the value of a variable are explored in some order and backtracked if they lead to failure.
- This is specified in Comet with the tryall<cp> method, that tries all values of the domain of some variable in some arbitrary order (actually, increasing)

```
var<CP>{int} x(cp,Dom);
...
tryall<cp>(v in Dom) cp.label(x,v);
```

- That is equivalent to the call of function label/1.

```
var<CP>{int} x(cp,Dom);
...
label(x);
```

Constraints: Basic Concepts

- Of course, many variables may exist that must be labelled. Often there the variables to label are in one array, x. In this case, one may label all elements of the array in increasing order as in (again equivalent to label(x).)

```
var<CP>{int} x[Rng](cp,Dom);
...
forall(i in Rng)
   tryall<cp>(v in Dom) cp.label(x,v); }
```

equivalent to

```
label(x);
```

- A more efficient policy (heuristics) is to label variables by increasing number of elements in their domain as in

```
var<CP>{int} x[Rng](cp,Dom);
...
forall(i in Rng) by (x[i].getSize())
     tryall<cp>(v in Dom) cp.label(q[i],v);
```

- This policy is so common that there is a built in function equivalent to it, namely

```
labelFF(x);
```

Constraints: Basic Concepts

- Finally to label two or more (arrays of) variables, the labeling may be done with many different policies:
- In sequence

```
var<CP>{int} x[Rng](cp,Dom);
var<CP>{int} y[Rng](cp,Dom);
...
   label(x);
   label(y);
```

… or interleaving

```
...
forall(i in Rng) {
   tryall<cp>(v in Dom) cp.label(x[i],v);
   tryall<cp>(v in Dom) cp.label(y[i],v);
}
```

… or leaving the choice of order to the solver

```
...
label(cp)
```

… or even with more sophisticated heuristics.

```
...
labelFF(cp)
```

Constraints: Other Languages

- Comet is a language that supports both CP (Complete Backtrack Search) and CBLS (Constrained-Based Local Search) and is thus adopted in the course, although not exclusively.
- The major problem with this language is that it is being discontinued, and replaced (soon?) by Objective-CP (designed by the same authors – Pascal Van Hentenryck and Laurent Michel.
- Meanwhile, the language that is becoming quite standard, for CP alone, is Zinc / Minizinc.
- In particular, it provides an interface (Flat-Zinc) that almost all existing CP solvers can support (Gecode, Choco, SICStus, … CaSPER).
- This makes it possible to test solvers in a competition held annually with the CP conferences.
- However, heuristics cannot be fully specified (a number of annotations are available but they are not sufficent for some problems) and no support for local search is available.

Constraints: Other Languages

- The declarative nature of ZINC is easily illustrated with the n-queens problem:

```
int: n = 24;

array [1..n] of var 1..n: q;

include "alldifferent.mzn”;

constraint alldifferent(q);                              % rows
constraint alldifferent(i in 1..n)(q[i] + i-1);    % / diagonal
constraint alldifferent(i in 1..n)(q[i] + n-i);    % \ diagonal


solve   :: int_search( q, first_fail,indomain_min, complete)
  satisfy;

output   ["8 queens, CP version:\n"] ++
         [        if fix(q[i]) = j then "Q " else ". " endif ++
                  if j = n then "\n" else "" endif
         |        i, j in 1..n
         ];
```

Constraints: Other Languages

… which can be compared with the Comet version:

```
import cotfd;
int t0 = System.getCPUTime();

int n = 1000; range S = 1..n;

Solver<CP> cp();
   var<CP>{int} q[i in S](cp,S);

solve<cp> {
   cp.post(alldifferent(q));
   cp.post(alldifferent(all(i in S) q[i] + i));
   cp.post(alldifferent(all(i in S) q[i] - i));
}
using {
   forall(i in S) by(q[i].getSize())
      tryall<cp>(v in S) cp.label(q[i],v);
}

int t1 = System.getCPUTime();
cout << q << endl;
cout << " cpu time (ms) = " << t1-t0 <<endl;
cout << " number of fails = " << cp.getNFail() << endl;
```

Constraints: Basic Concepts

- As discussed when searching for a solution CP interleaves propagation of constraints with labelling, i.e.
  - It propagates all constraints, removing values from the domain of variables that do not belong to a solution.
    - For example if variables **`X`** and **`Y`** have domain **`1..8`** and there is a constraint **`X > Y`**, then their domains are pruned to **`X:2..8`** and **`Y::1..7`**.
  - When no more propagation is possible (i.e. a fixpoint has been reached) , a new variable is labelled (its domain reduced, usually to a single value) and step 1 is repeated.
- Of course, it is important that there is a good trade-off between the cost of propagating constraints and the pruning that results from it.
- To analyse such trade-off we will do a more theoretical and abstract discussion on these issues and will discuss later more practical issues.

Constraints: Basic Concepts

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, $\mathbf{x_1}$ to $\mathbf{x_n}$, all with the domain from **1** to **n**.

$$\mathbf{Dom(x_i) = \{1,2, ..., n\}} \qquad \text{or} \qquad \mathbf{x_i :: 1..n.}$$

- **Note:** In 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 $\mathbf{C_{ijk}}$ involving variables $\mathbf{X_i}$, $\mathbf{X_j}$ and $\mathbf{X_k}$, then

$$\mathbf{C_{ijk} \subseteq dom(X_i) \times dom(X_j) \times dom(X_k)}$$

Constraints: Basic Concepts

- Given a constraint C, the set of variables involved in that constraint is denoted by **vars(C).**

- Simetrically, the set of constraints in which variable X participates is denoted by **cons(X).**

- Notice that a constraint is a relation, not a function, so that it is always $\mathbf{C_{ij} = C_{ji}}$.

- 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, the constraint $\mathbf{C_{13}}$ involving $\mathbf{Q_1}$ and $\mathbf{Q_3}$ in the 4-queens problem, may be specified

- **By extension** (label form):

  $C_{13} = \{\{Q_1\text{-}1, Q_3\text{-}2\}, \{Q_1\text{-}1, Q_3\text{-}4\}, \{Q_1\text{-}2, Q_3\text{-}1\}, \{Q_1\text{-}2, Q_3\text{-}3\},$

  $\{Q_1\text{-}3, Q_3\text{-}2\}, \{Q_1\text{-}3, Q_3\text{-}4\}, \{Q_1\text{-}4, Q_3\text{-}1\}, \{Q_1\text{-}4, Q_3\text{-}3\}\}.$

  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**:

  $C13 = (Q_1 \neq Q_3) \land (1+Q_1 \neq 3+Q_3) \land (3+Q_1 \neq 1+Q_3).$

Constraints: Basic Concepts

Definition (**Constraint Arity**):

- The **constraint arity** of some constraint C is the number of variables over which the constraint is defined, i.e. the cardinality of the 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 **<X, D, C>** where

- **X** is the set of variables of the problem
- **D** is the domain(s) of its variables
- **C** is the set of constraints of the problem

Definition (**Problem Solution**):

A **solution** to a Constraint Satisfaction Problem **P**: **<X, D, C>**, is a compound label over the variables **X** of the problem, which satisfies all constraints in **C**.

Constraints: Basic Concepts

Definition (**Constraint Satisfaction and Optimisation Problem**):

A **constraint satisfaction problem** is a tuple **< X, D, C, F >** where

- **X** is the set of variables of the problem
- **D** is the domain(s) of its variables
- **C** is the set of constraints of the problem
- **F** is a function on the variables of the problem

Definition (**Problem Solution**):

S is a **solution** of a **CSOP P**: **<X, D, C, F >**, iff:

- **S** is a solution of the corresponding **CSP P': <X, D, C>;**
- No other solution S' has a better value for function F

Constraints: Basic Concepts

- For convenience, the 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

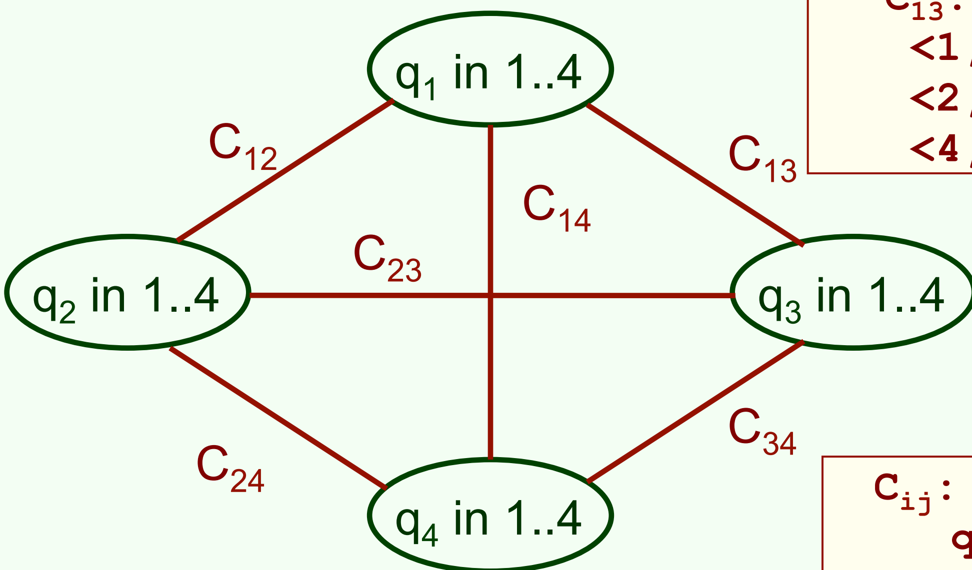
- There is a node for each of the variables of the problem.
- For each non-trivial constraint of the problem, involving one or two variables, the graph contains an arc linking the corresponding nodes.

- 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**:



C_{13}:
<1,2>, <1,4>, <2,1>,
<2,3>, <3,2>, <3,4>,
<4,1>, <4,3>

C_{ij}:
$q_i \neq q_j$ (`qi \= qj`)
$q_i + i \neq q_j + j$ (`qi + i \= qj + j`)
$q_i - i \neq q_j - j$ (`qi - i \= qj - j`)

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

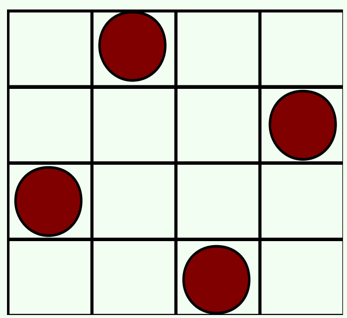
- An important issue to consider in solving a constraint satisfaction problem is the existence of redundant values and labels in its constraints.

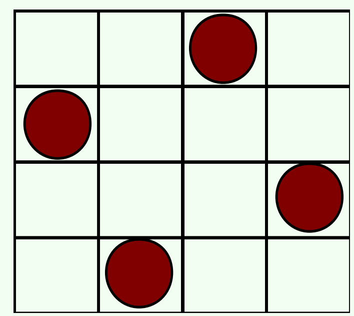
Definition (**Redundant Value**):

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

Example: The 4 queens problem only admits two solutions:

<2,4,1,3> and **<3,1,4,2>.**





- 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 **P1 = <X_1, D_1, C_1>** and **P2 = <X_2, D_2, C_2>** 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=<X, D, C>** is reduced to **P'=<X', D', C'>** 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 a problem is reduced, the easier it is, in principle, to solve it.

- Given a problem **P = <X, D, C>** with **n** variables $x_1, .., x_n$ the potential search space where solutions can be found (i.e. the leaves of the search tree with compound labels $\{<x_1\text{-}v_1>, ..., <x_n\text{-}v_n>\}$) has cardinality

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

- 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.

S/S'		n: 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 usually 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 = <X, D, C>**, is a compound label on a subset of **k** of its variables, $\mathbf{X_k}$, that satisfies all the constraints in **C** whose variables are included in $\mathbf{X_k}$.

Propagation in Search

- Given a problem with **n** variables $\mathbf{x_1}$ to $\mathbf{x_n}$, 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:

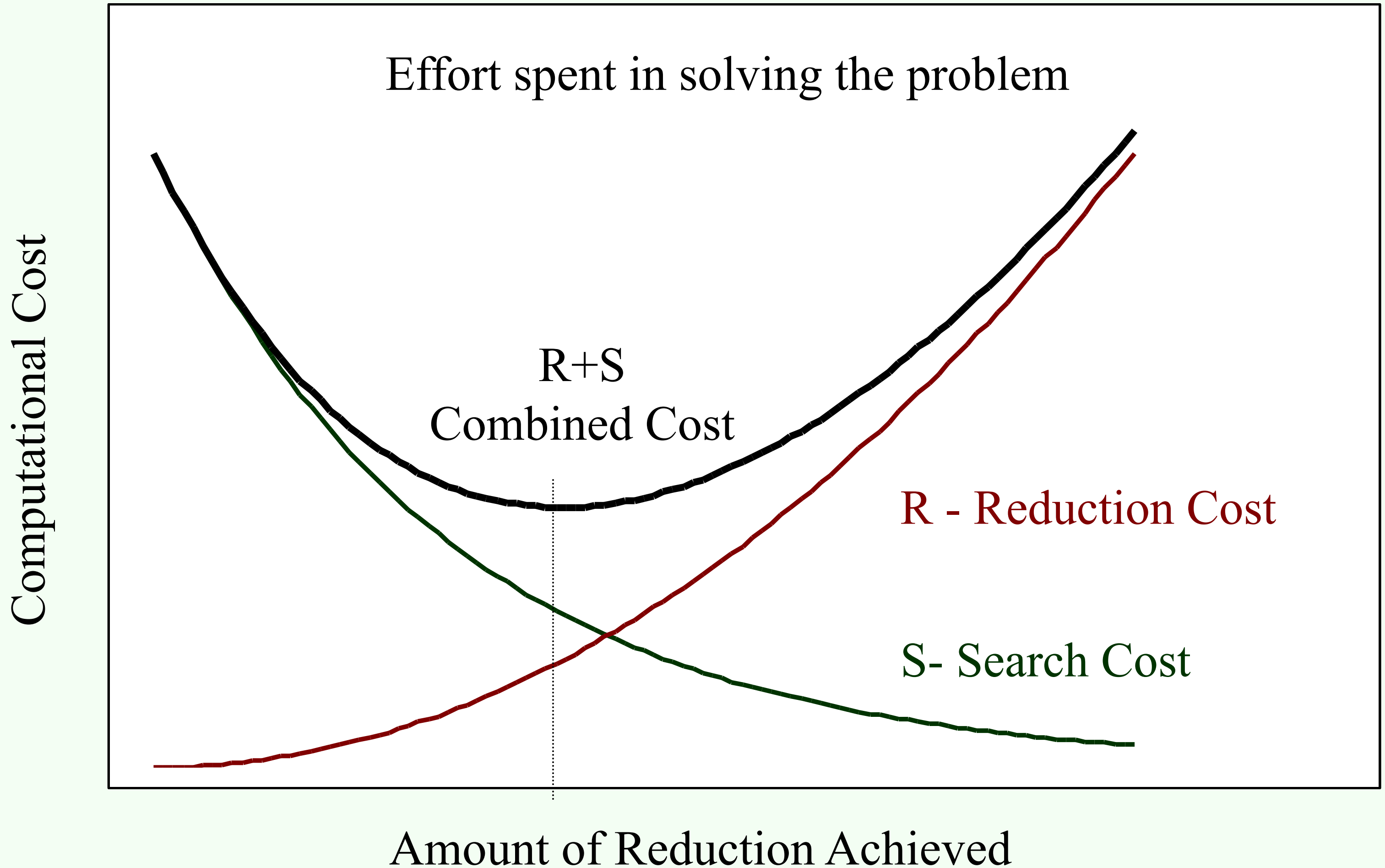
```
Declaration of Variables and Domains,
Specification of Constraints,
   propagation,  % reduction of the whole problem
% Labelling of Variables,
   label(x1),    % variable/value selection with backtraking
   propagation,  % reduction of problem {x2 ... xn}
   label(x2),
   propagation,  % reduction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



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**
  - **Consistency-i**

Node - Consistency

Definition (**Node Consistency**):

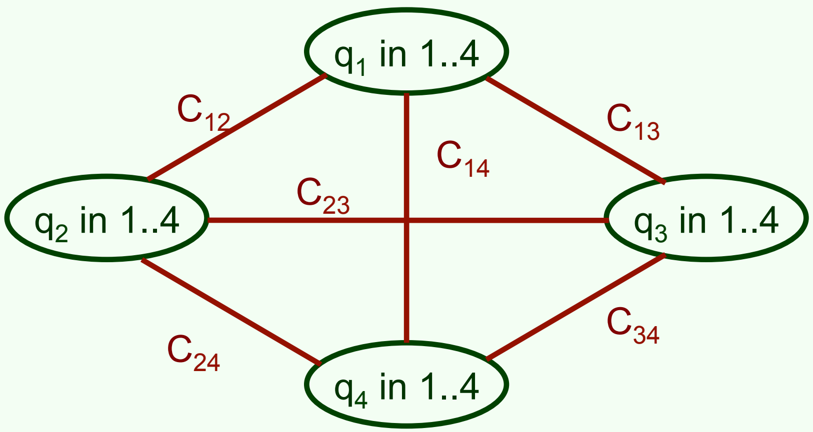
A constraint satisfaction problem is **node-consistent** if no value on the domain of its variables violates the **unary** constraints.

- This criterion may seem both obvious and useless. After all, who would specify a domain that violates the unary constraints ?!
- However, this criterion must be regarded within the context of the execution model that incrementally completes partial solutions. Constraints that were not unary in the initial problem become so when one (or more) variables are enumerated.

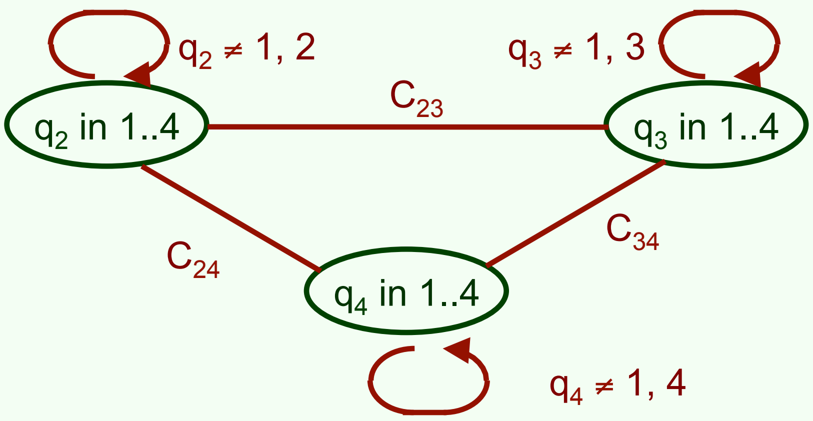
Node - Consistency

Example:

- After the initial posting of the constraints, the constraint network model at the right represents the 4-queens problem.

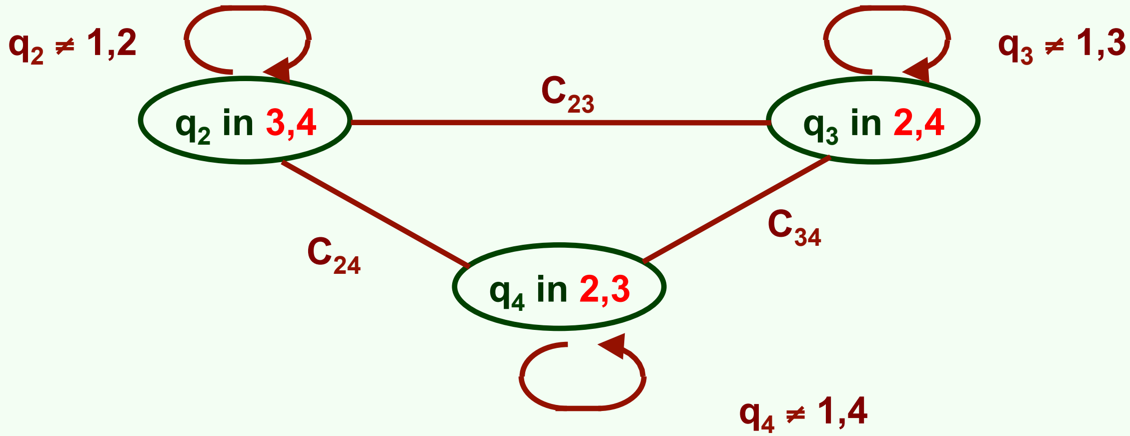


- After enumeration of variable Q_1, i.e. X_1=1, constraints C_{12}, C_{13} and C_{14} become **unary** !!



Node - Consistency

- An algorith that maintains node consistency should remove from the domains of the “future” variables the appropriate values.



- Maintaining node consistency achieves the following domain reduction.

●			
1	1		
1		1	
1			1

$q_2 \neq 1,2$

$q_3 \neq 1,3$

$q_4 \neq 1,4$

Arc - Consistency

- A **more** demanding and complex criterion of consistency is that of arc-consistency

Definition (**Arc Consistency**):

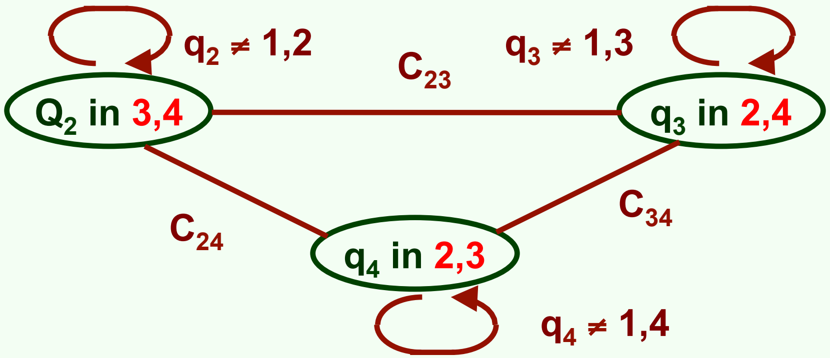
A constraint satisfaction problem is arc-consistent if,

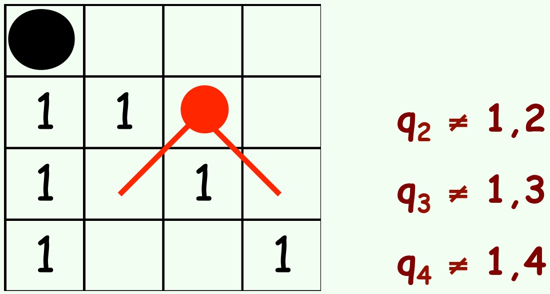
- It is node-consistent; and
- For every label x_i-v_i of every variable x_i, and for all constraints C_{ij}, defined over variables x_i and x_j, there must exist a value v_j that **supports** v_i, i.e. such that the compound label $\{x_i\text{-}v_i, x_j\text{-}v_j\}$ satisfies constraint C_{ij}.

Arc - Consistency

Example:

- After enumeration of variable q_1=1, and making the network node-consistent, the 4 queens problem has the following constraint network:



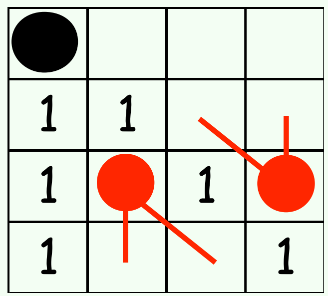


- However, label q_2-3 has **no support** in variable q_3, since neither the compound label {q_2-3 , q_3-2} nor {q_2-3 , q_3-4} will satisfy constraint C_{23}.

- Therefore, value 3 can be safely removed from the domain of q_2.

Example (cont.):

- In fact, none (!) of the values of q_3 has support in variables q_2 and q_4, as shown below:



$q_2 \neq 1,2$

$q_3 \neq 1,3$

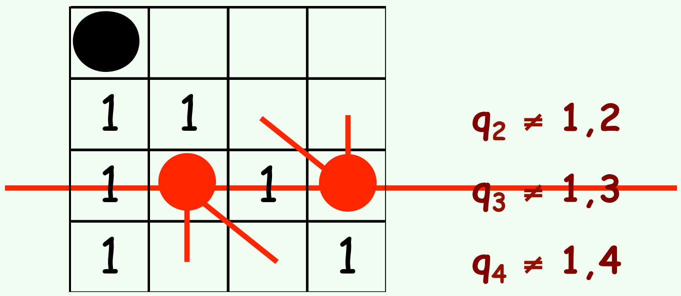
$q_4 \neq 1,4$

  - Label q_3-4 has no support in variable q_2, since none of the compound labels {q_2-3, q_3-4} and {q_2-4, q_3-4} satisfy constraint C_{23}.
  - Label q_3-2 has no support in variable q_4, since none of the compound labels {q_3-2, q_4-2} and {q_3-2, q_4-3} satisfy constraint C_{34}.

Arc - Consistency

Example (cont.):

- Since none of the values from the domain of q_3 has support in variables q_2 and q_4, maintenance of arc-consistency **empties** the domain of q_3!



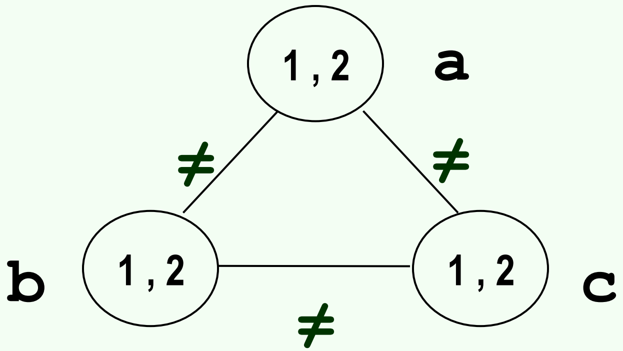
- Hence, maintenance of arc-consistency not only prunes the domain of the variables but also antecipates the detection of unsatisfiability in variable q_3 !

- In this case, backtracking of q_1=1 may be started even before the enumeration of variable q_2.

- Given the good trade-of between pruning power and simplicity of arc-consistency, a number of algorithms have been proposed to maintain it.

Path-Consistency

- The following constraint network is obviously inconsistent:



- Nevertheless, it is arc-consistent: every binary constraint of difference (≠) is arc-consistent whenever the constraint variables have at least 2 elements in their domains.

- However, is is not path-consistent: **no** label {<a-v_a>, <b-v_b>} that is consistent (i.e. does not violate any constraint) can be extended to the third variable (c).

  {<a-1>, <b-2>} → c ≠ 1, 2 ; {<a-1>, <b-2>} → c ≠1, 2

- This property is captured by the notion of path-consistency.

Path-Consistency

Definition (**Path Consistency**):

A constraint satisfaction problem is path-consistent if,

- It is arc-consistent; and
- Every consistent 2-compound label {x_i-v_i, x_{ij}-v_j,} can be extended to a consistent label with a third variable x_k (k ≠ i and k ≠j }.

The second condition is more easily understood as

- For every compound label {x_i-v_i, x_{ij}-v_j,} there must be a value v_k that **supports** {x_i-v_i, x_{ij}-v_j,}, i.e. the compound label {x_i-v_i, x_j-$v_{j,}$ x_k-v_k} satisfies constraints C_{ij}, C_{ik}, and C_{kj}.

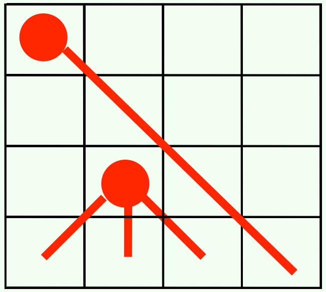
Path-Consistency

Example:

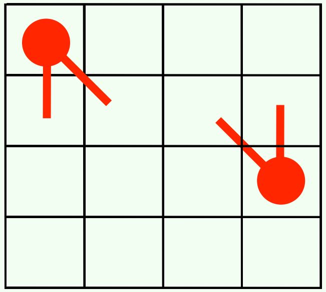
- By enforcing path consistency it is possible to avoid backtracking in the 4-Queens problem.
- In fact, q_1-1 has only two supports in variable q_3, namely q_3-2 and q_3-4.

However:

- <q_1-1, q_3-2> cannot be extended to variable q_4



- <q_1-1, q_3-4> cannot be extended to variable q_2



- Hence, q_1-1 can be safely removed from the domain of variable q_1.
- With similar reasoning, it may be shown that none of the corners, and none of the centre positions can have a queen.

Path-Consistency

- In general, and despite the previous example, maintaining path consistency does not prune the domain of a variable, but rather "forbids" compound labels with cardinality 2.
- This means that imposing arc-consistency on variables x_i and x_j through variable x_k, will tighten the (possible non-existing) constraint between x_i and x_j.

- In the example, a constraint of equality is imposed on variables **b** and c, because the compound labels { b-1 , c-1 } and { b-2 , c-2 } cannot be extended to variable **a**.