Constraint Programming

- An overview

- Constraint Propagation
- Constraint Networks and Consistency Criteria
- Node- and Arc-consistency
- Enforcing Algorithms and their Complexity

Constraint Propagation

- Non trivial constraint satisfaction problems are typically NP-complete and as such 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
	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 = $3.6 * 10^3$ 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 cripto-arithmetic problem.

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

$$10 \times 9 \times \ldots \times 3 = 1\,814\,400 \approx 2*10^6$$

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 ≠ 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), s
  - $D = 6$

  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
```

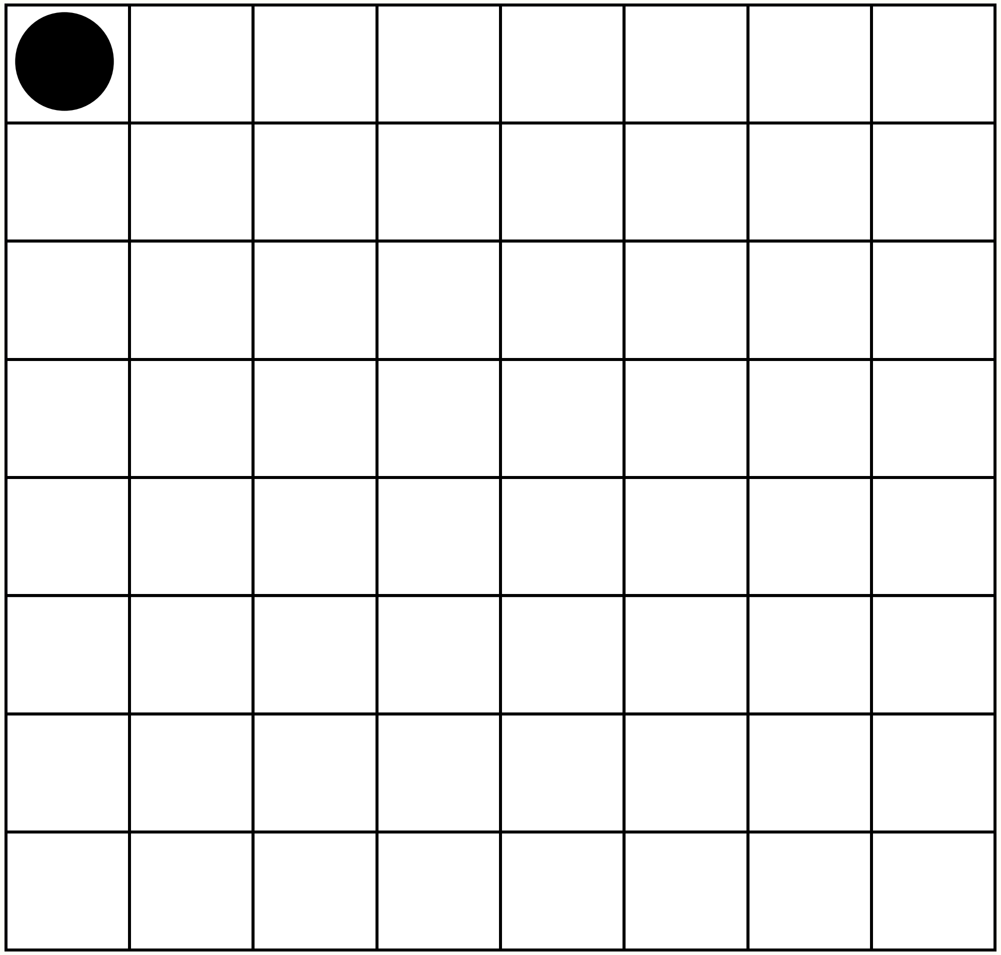
Constraint Propagation

- 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.

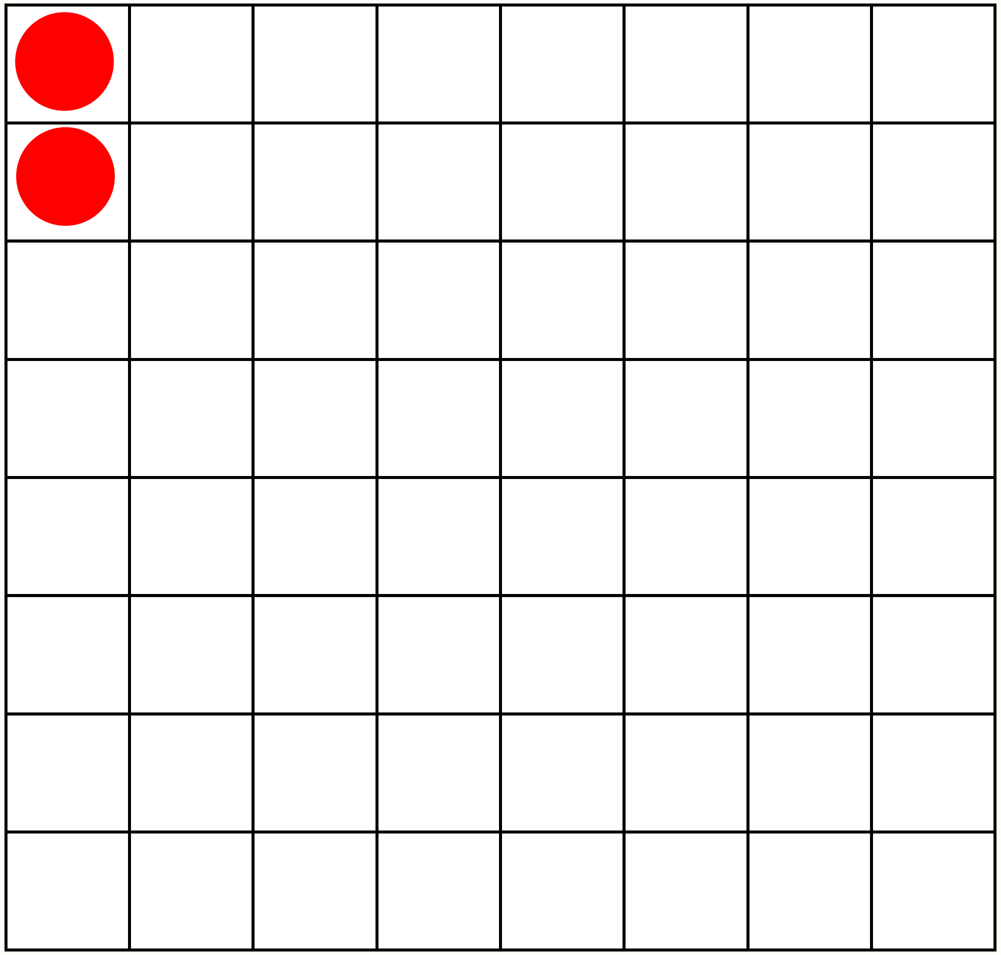
Backtracking



Tests 0 **Backtracks 0**

Backtracking

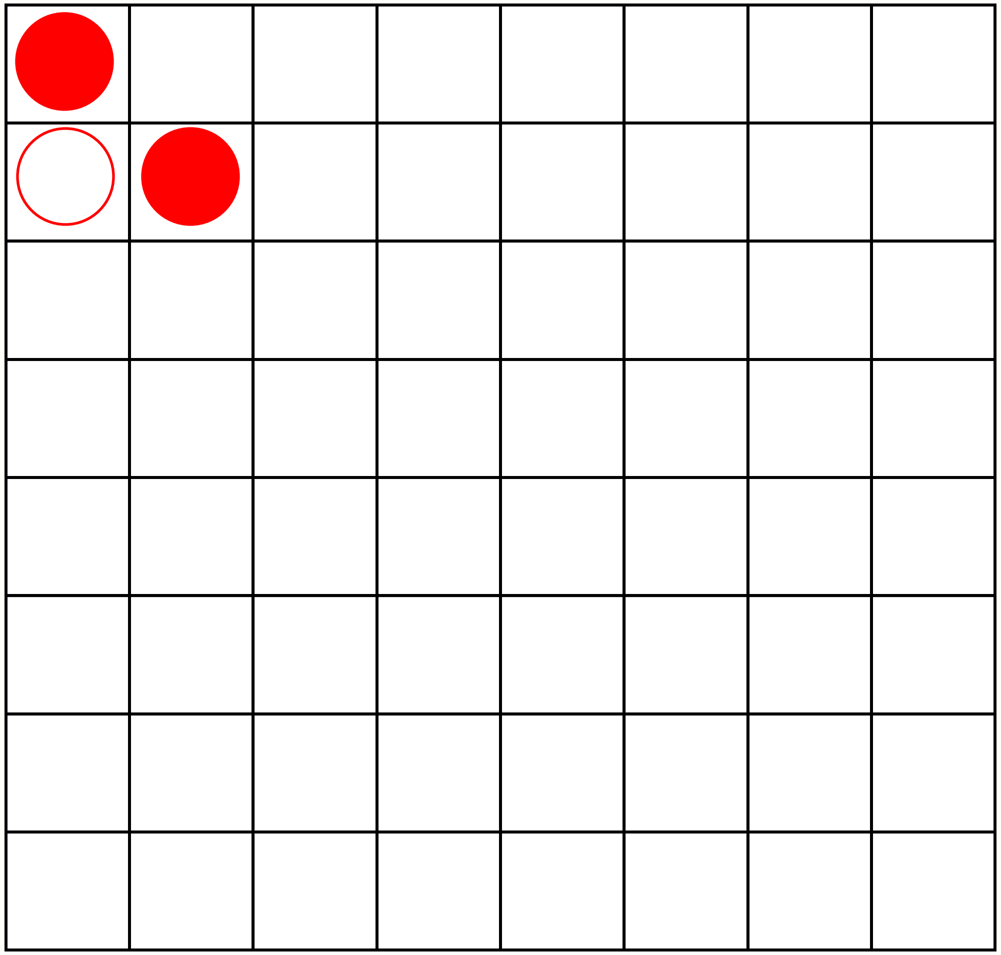
```
Q1 \= Q2,   L1+Q1 \= L2+Q2,   L1+Q2 \= L2+Q1.
```



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

Backtracking

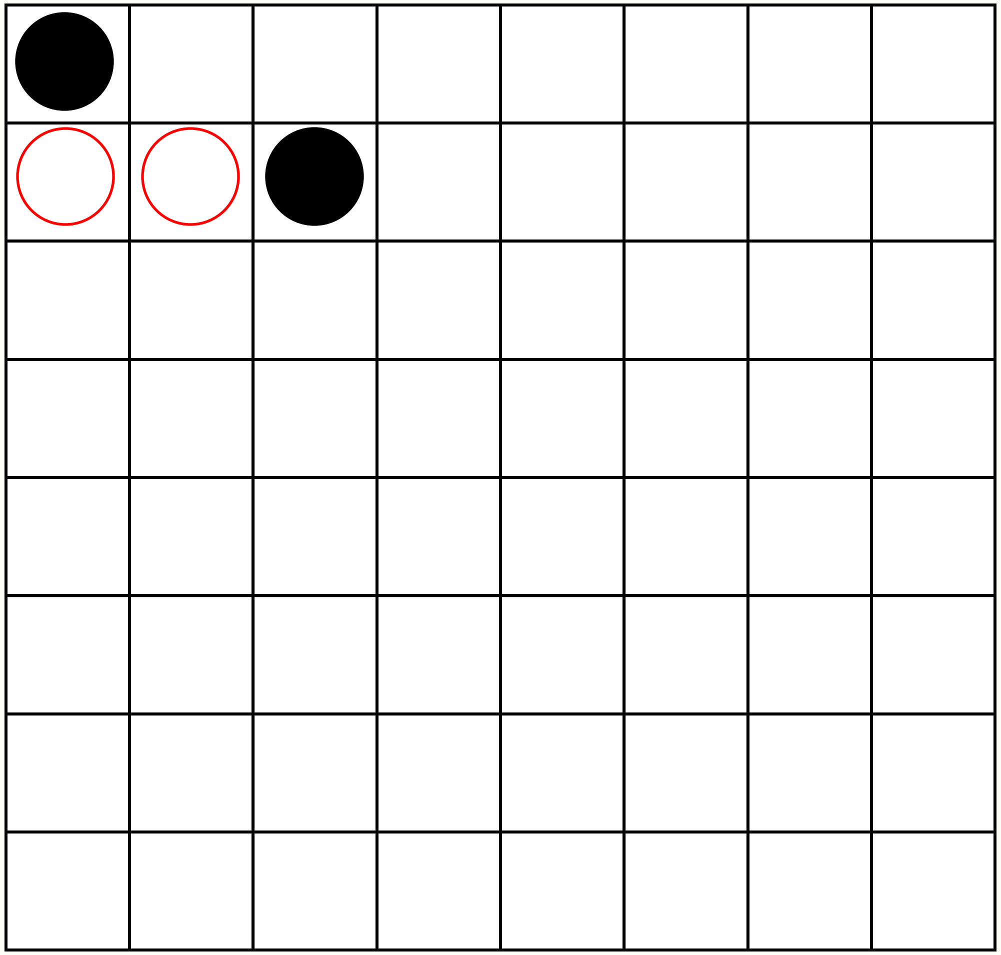
```
Q1 \= Q2,  L1+Q1 \= L2+Q2,  L1+Q2 \= L2+Q1.
```



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

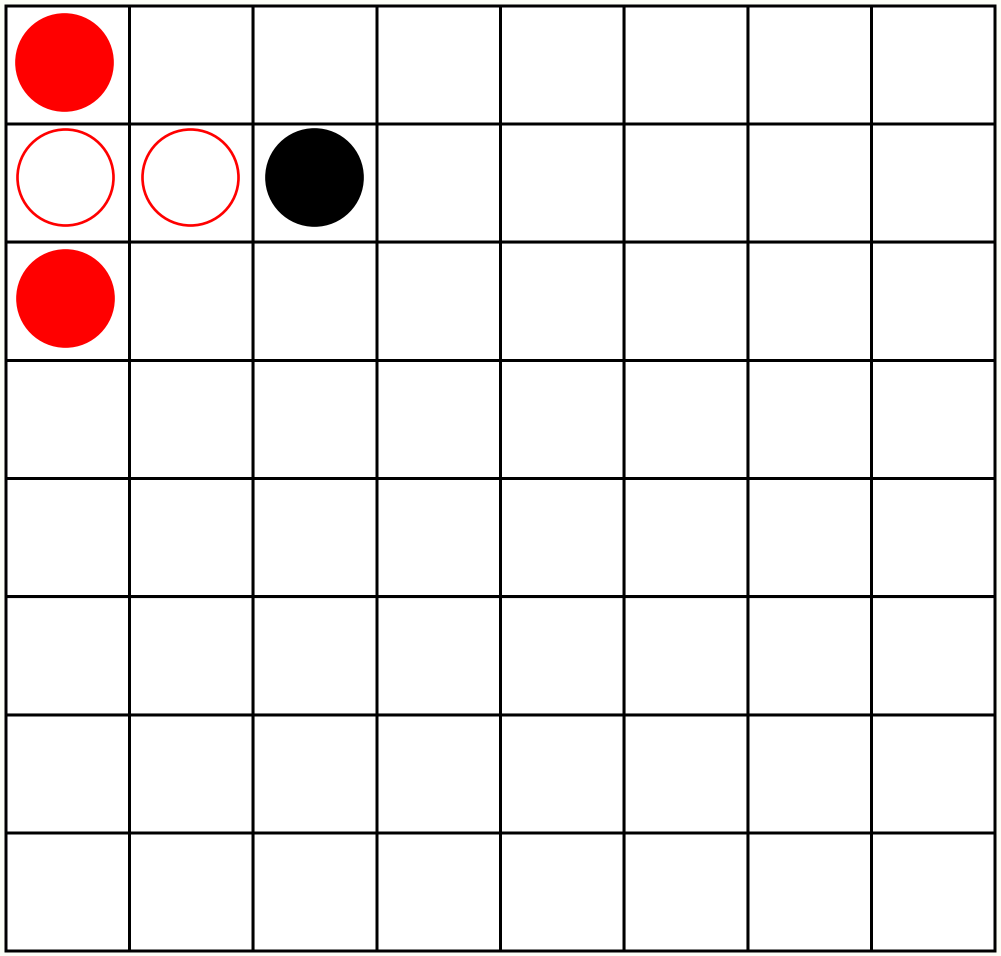
Backtracking

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



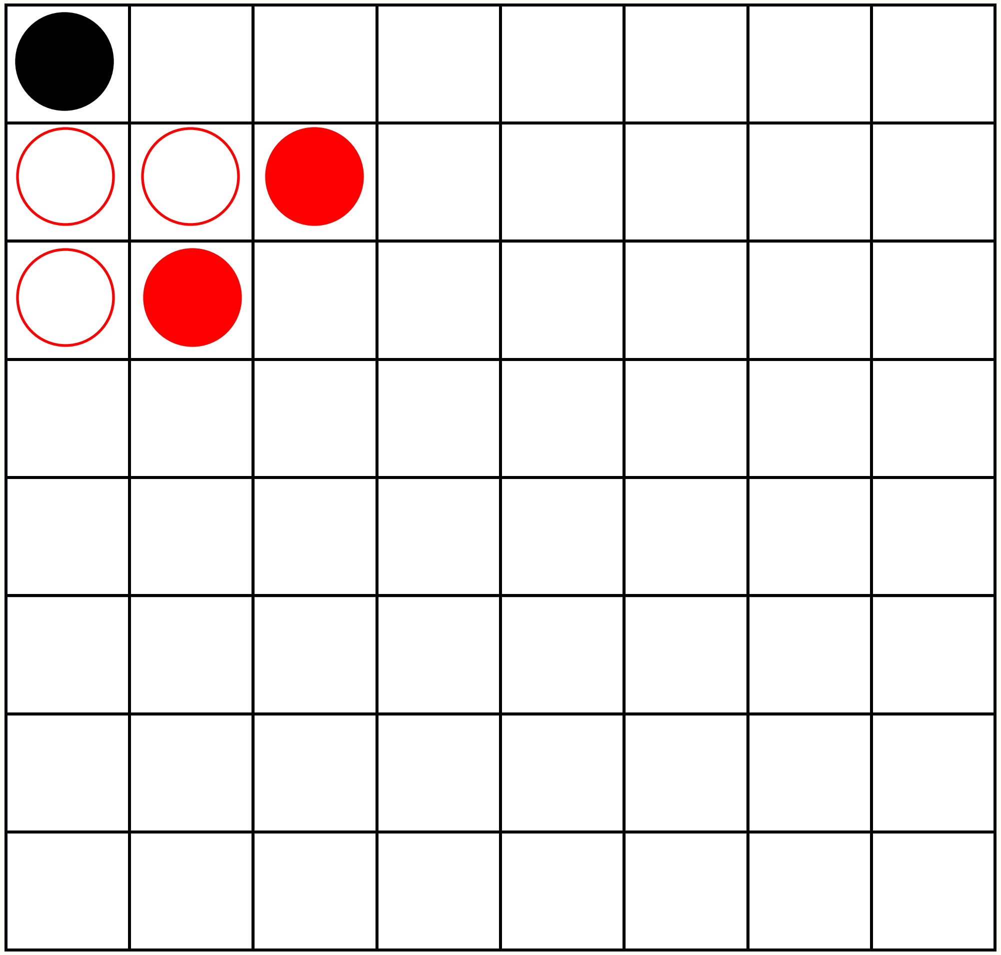
Tests 2 +1 = 3 **Backtracks 0**

Backtracking



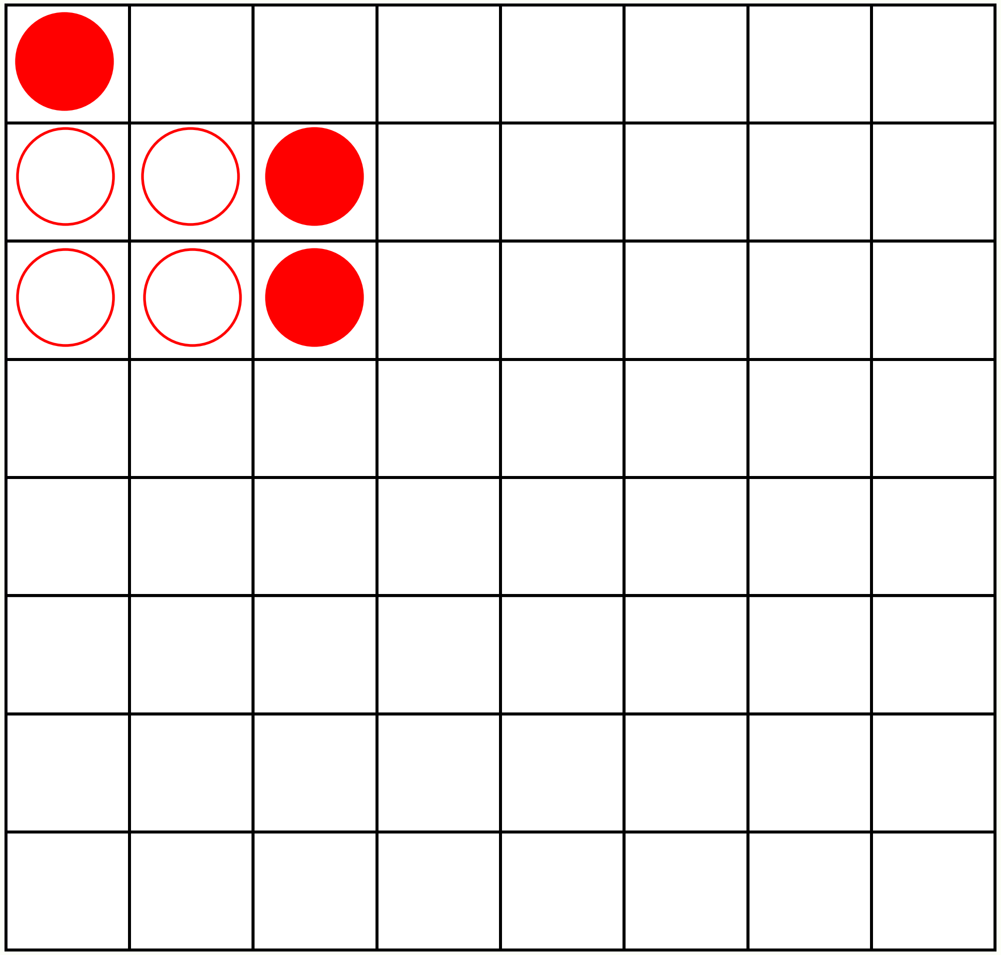
Tests 3 +1 = 4 **Backtracks 0**

Backtracking



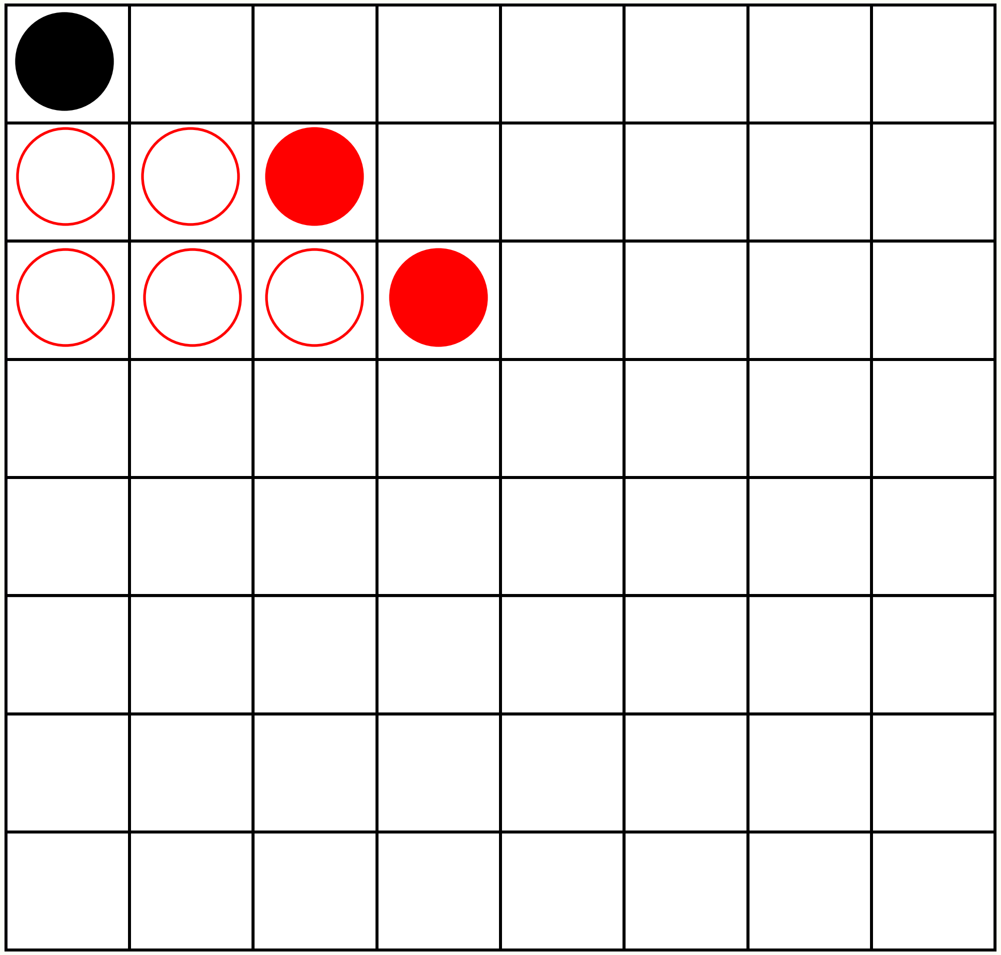
Tests 4 +2 = 6 **Backtracks 0**

Backtracking



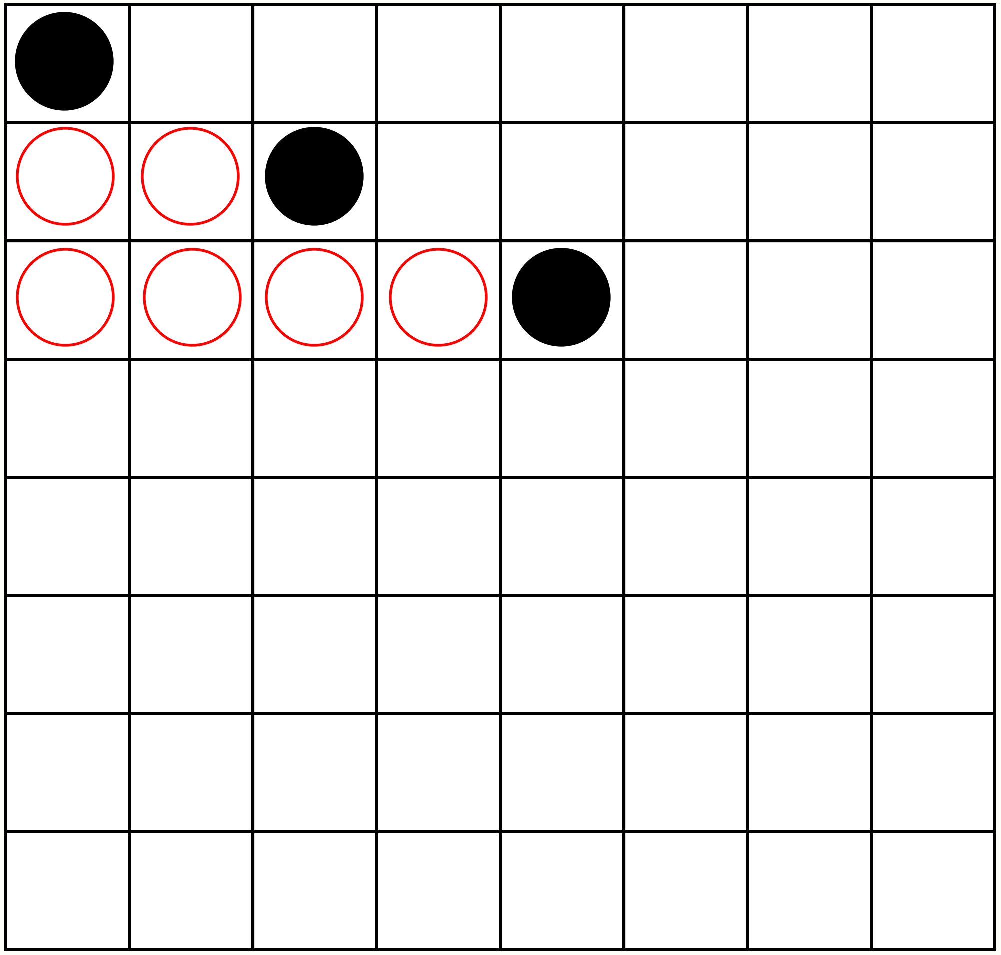
Tests 6 + 1 = 7 **Backtracks 0**

Backtracking



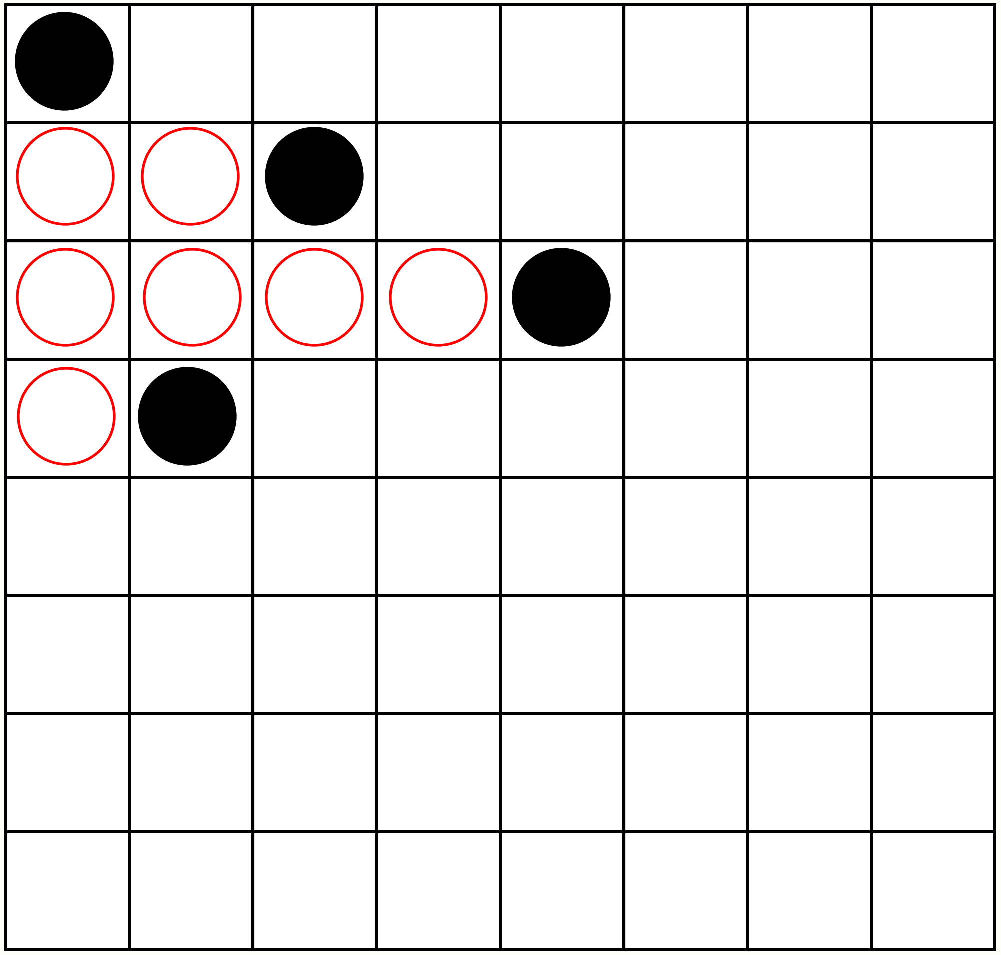
Tests 7 + 2 = 9 **Backtracks 0**

Backtracking



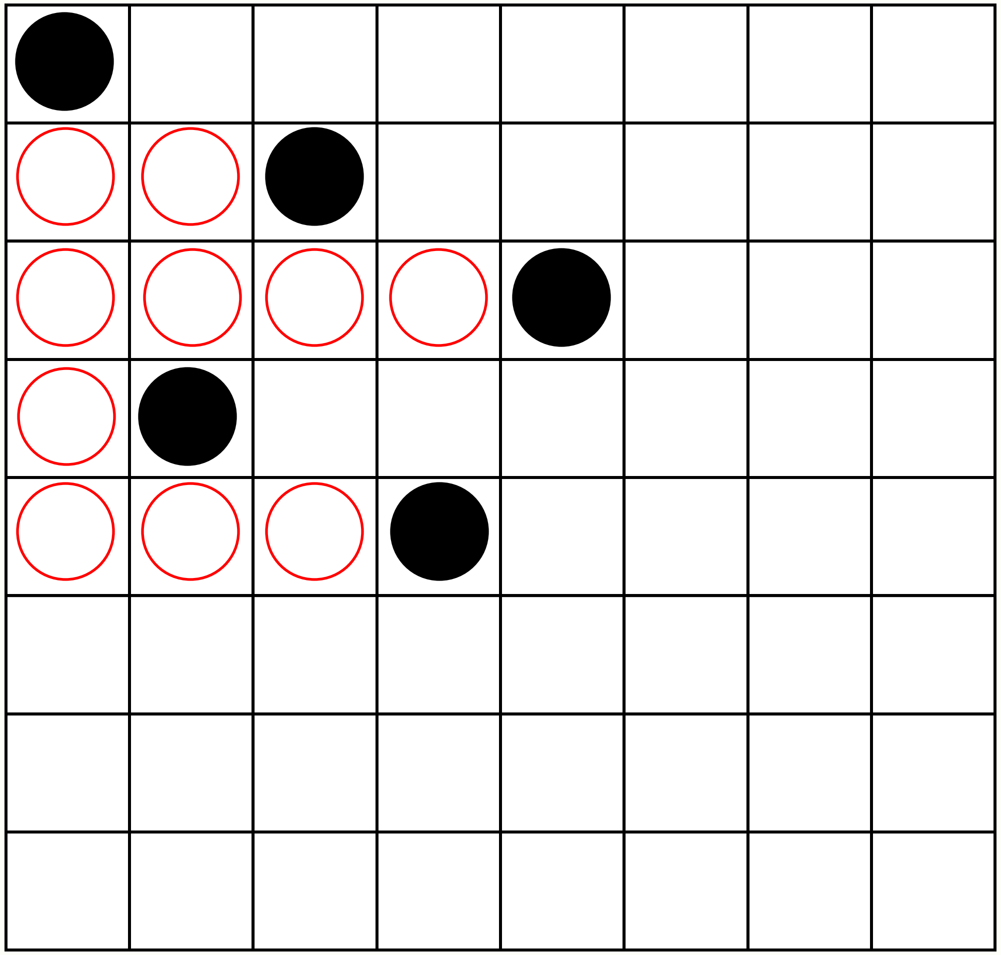
Tests 9 + 2 = 11 **Backtracks 0**

Backtracking



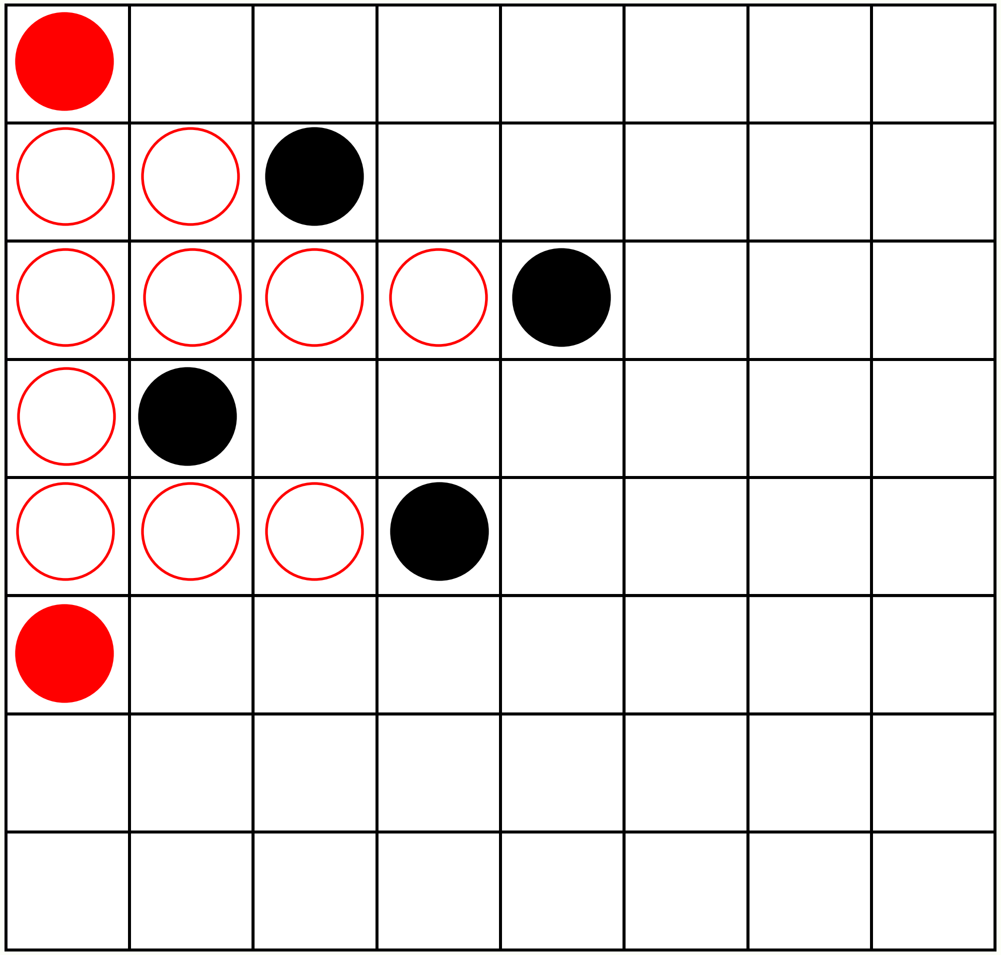
Tests 11 + 1 + 3 = 15 **Backtracks 0**

Backtracking



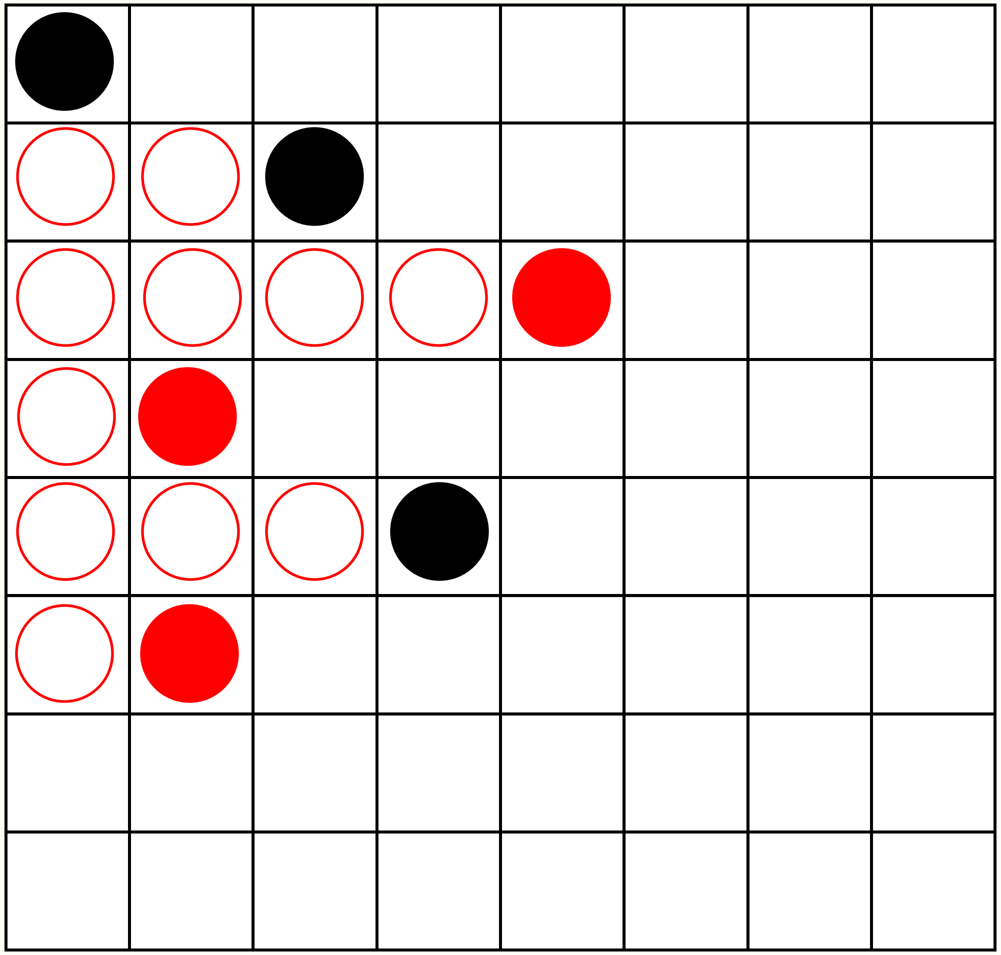
Tests 15+1+4+2+4 = 26 Backtracks 0

Backtracking



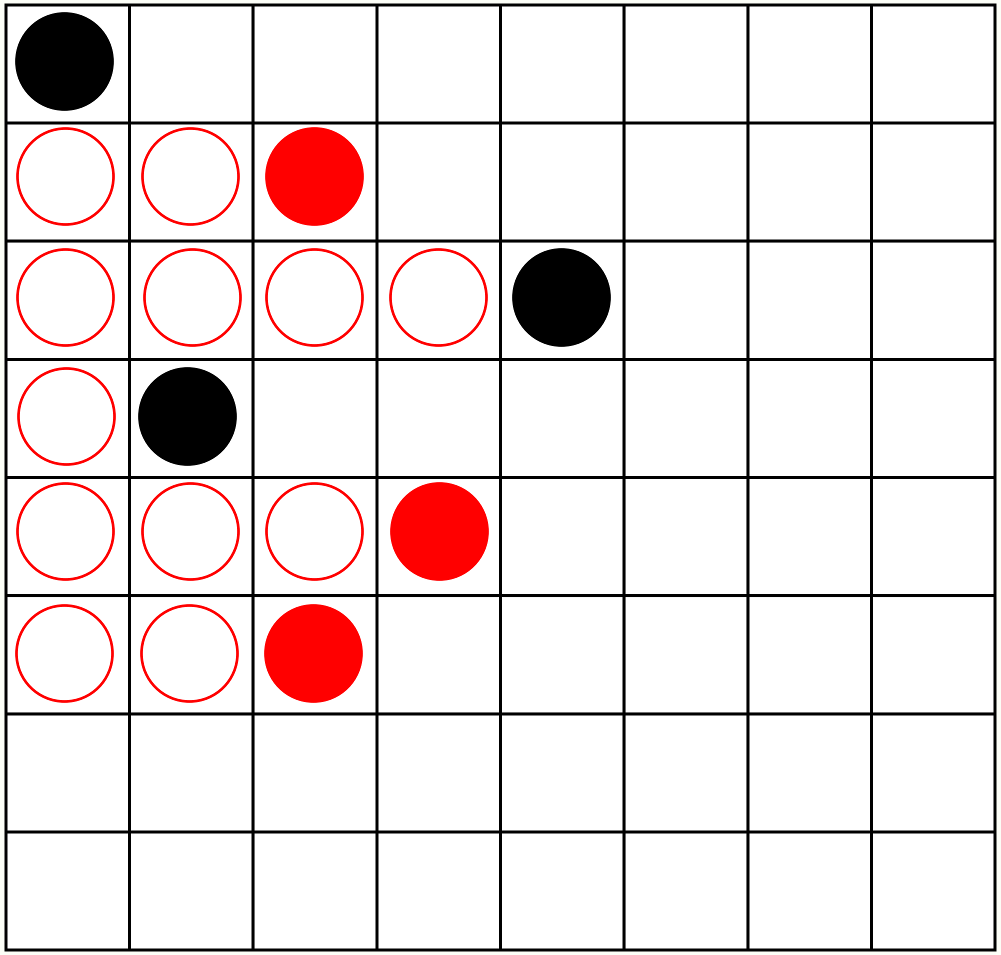
Tests 26+1 = 27 **Backtracks 0**

Backtracking



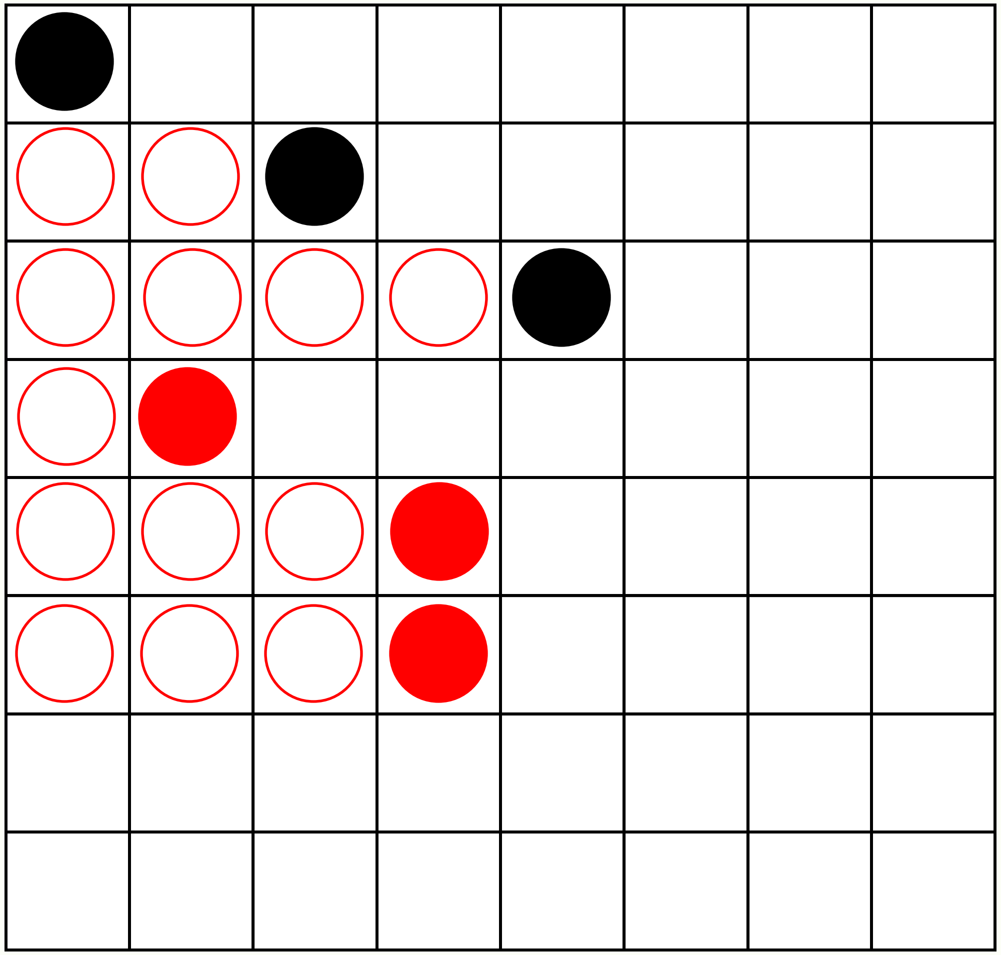
Tests 27 + 3 = 30 Backtracks 0

Backtracking



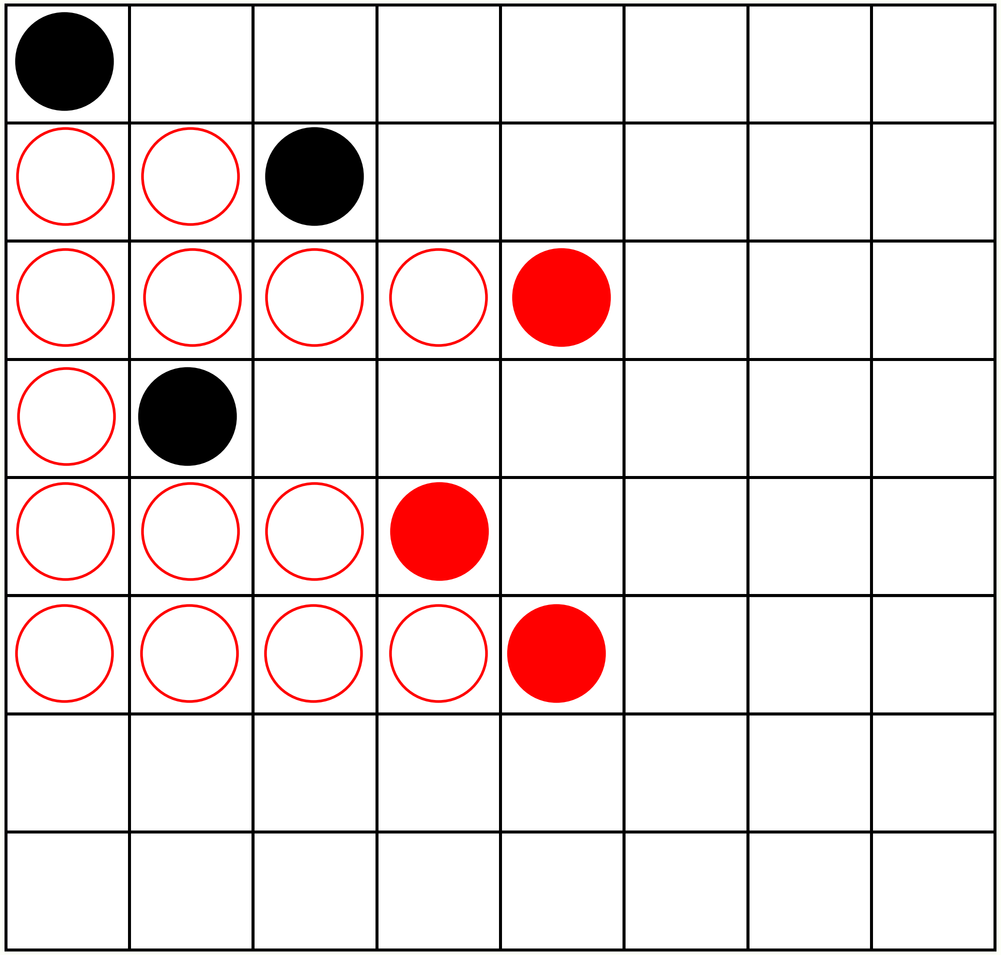
Tests 30+2 = 32 Backtracks 0

Backtracking



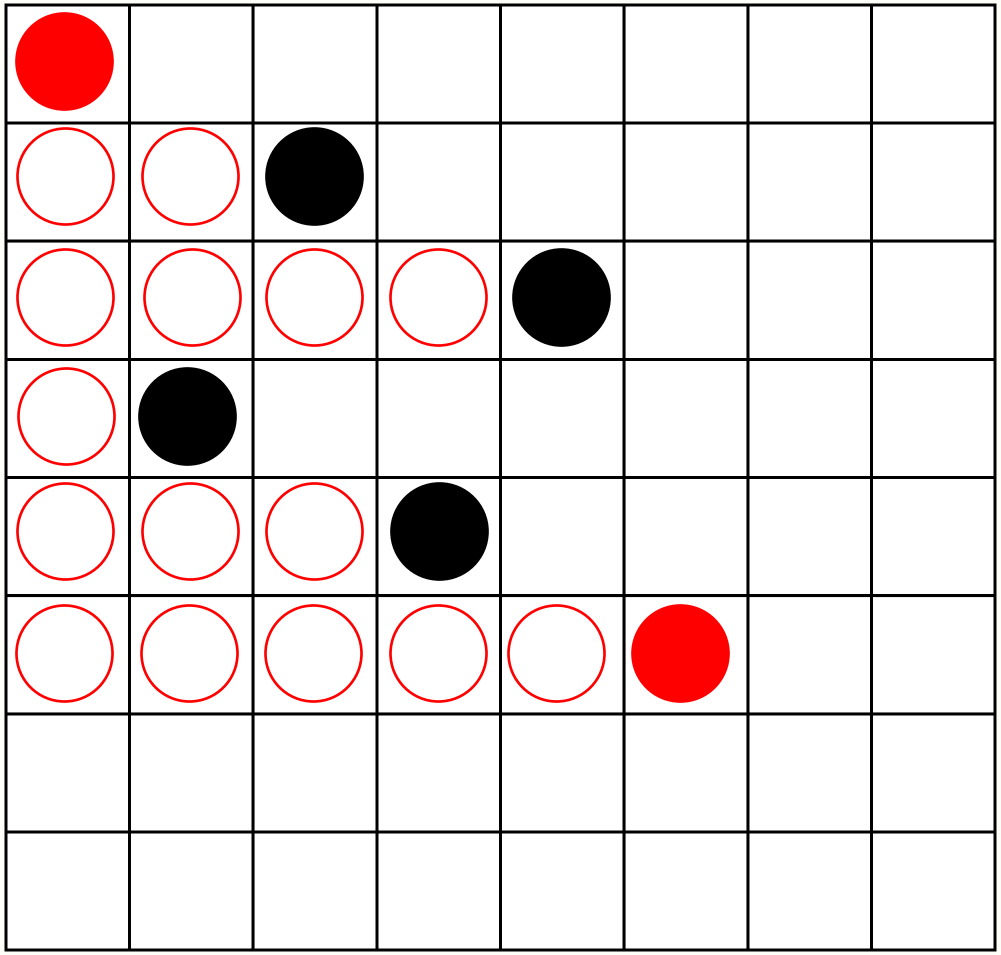
Tests 32 + 4 = 36 Backtracks 0

Backtracking



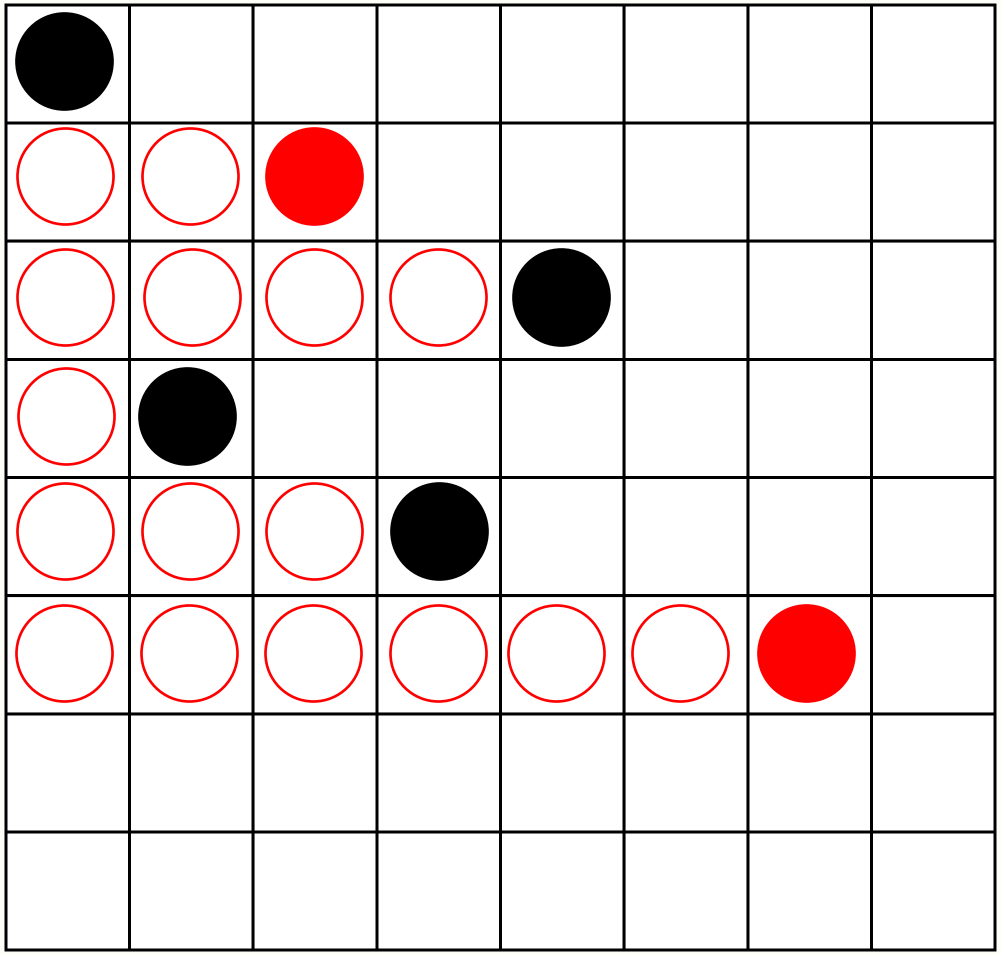
Tests 36 + 3 = 39 Backtracks 0

Backtracking



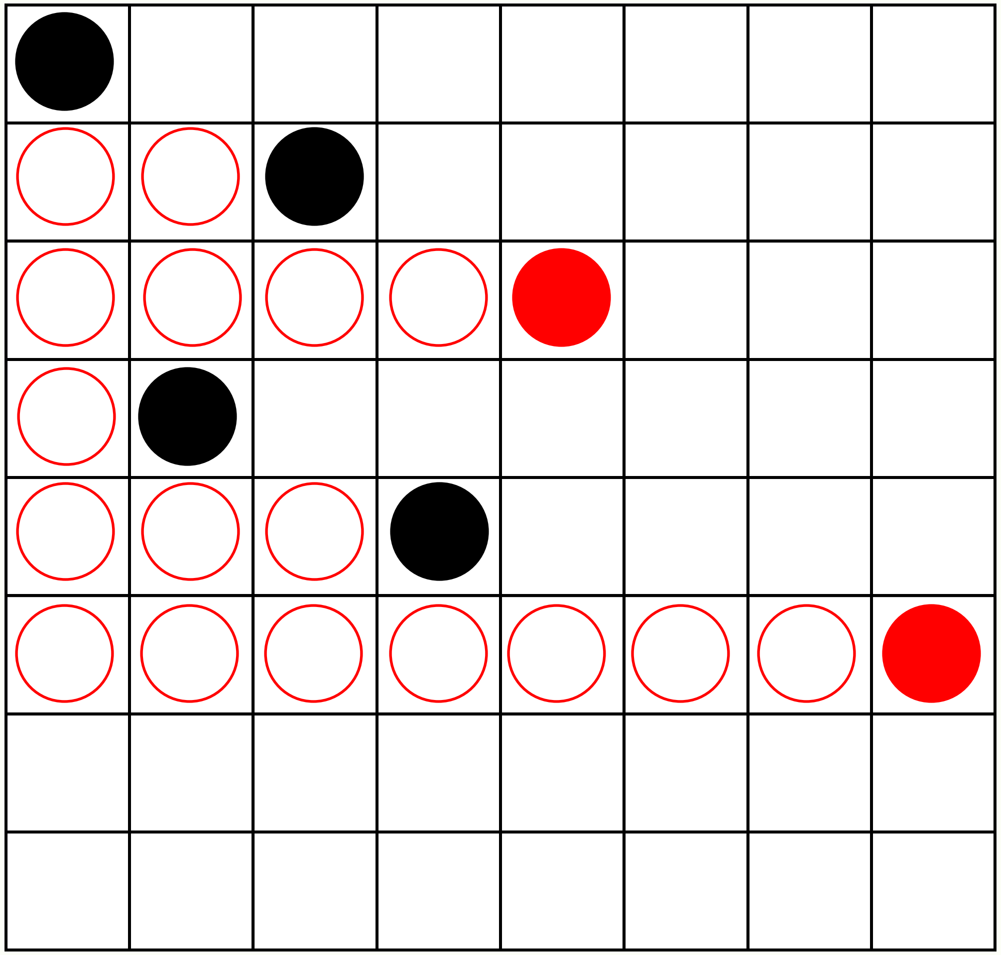
Tests 39 + 1 = 40   Backtracks 0

Backtracking



Tests 40 + 2 = 42 Backtracks 0

Backtracking



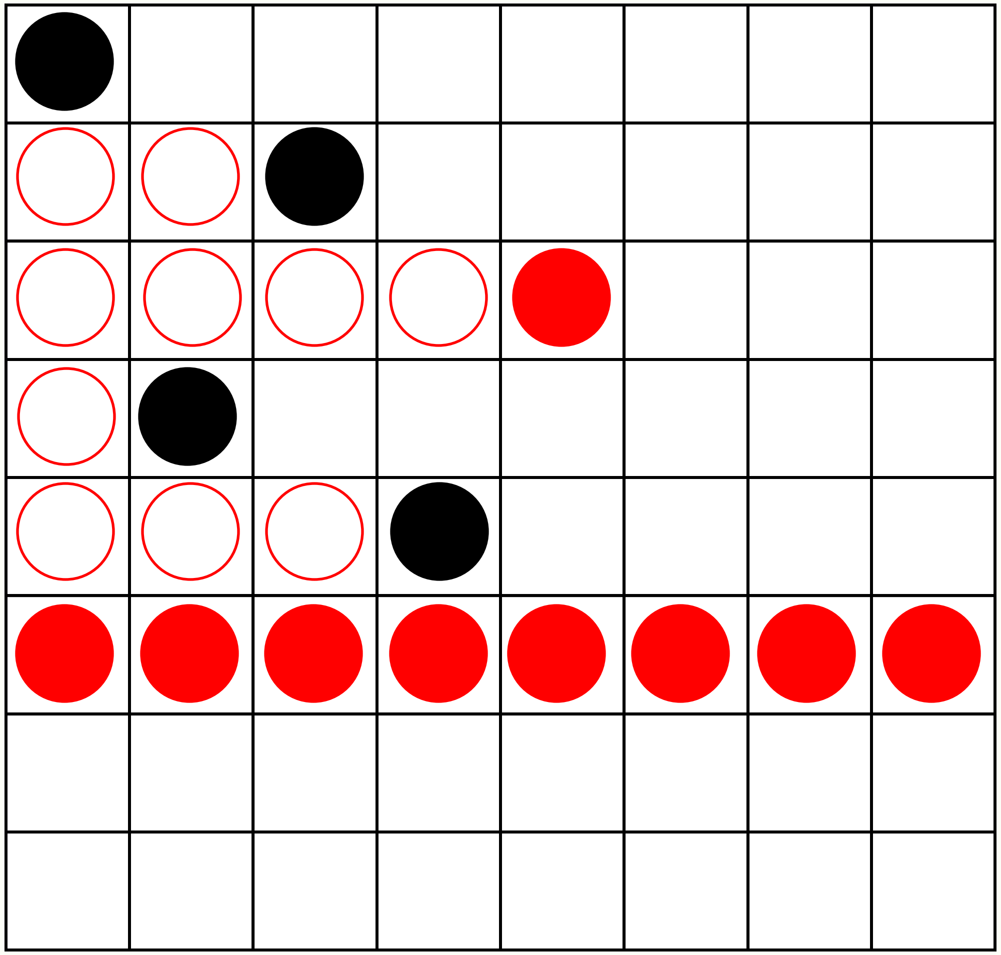
Tests 42 + 3 = 45 Backtracks 0

Backtracking

Q6 Fails

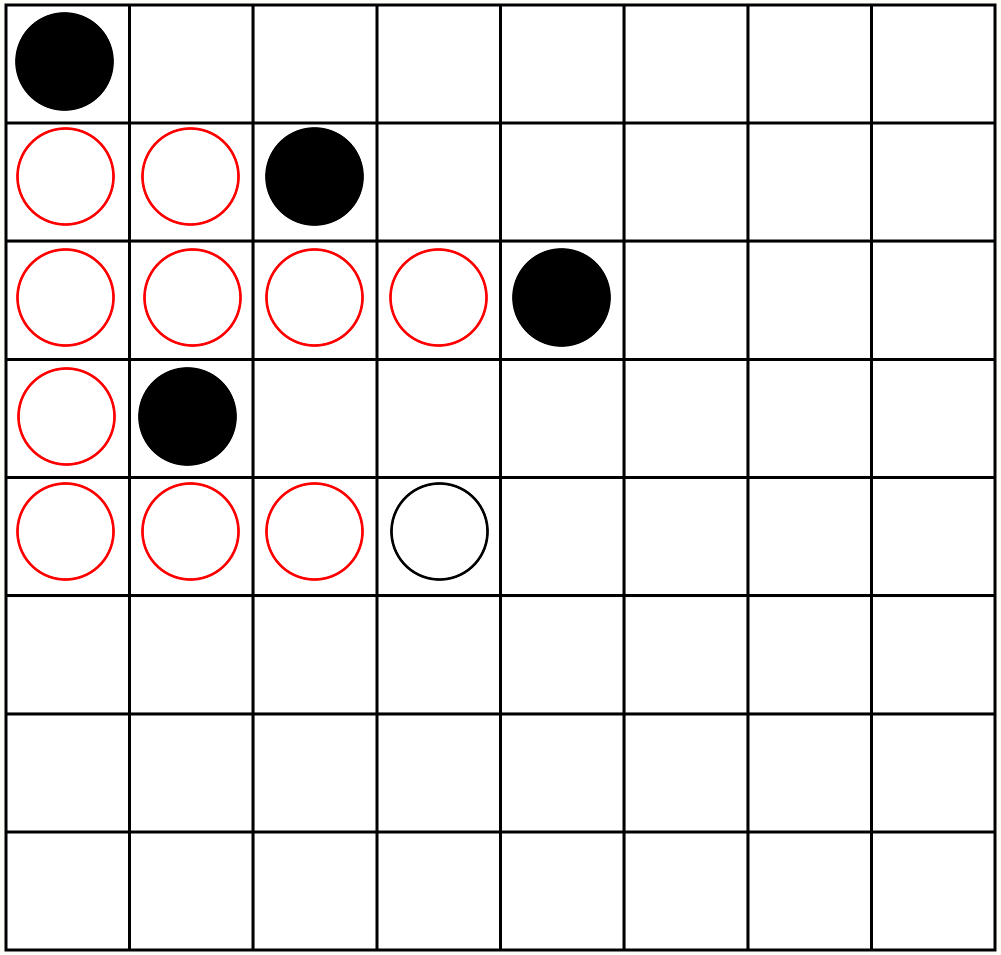
Backtracks to

Q5



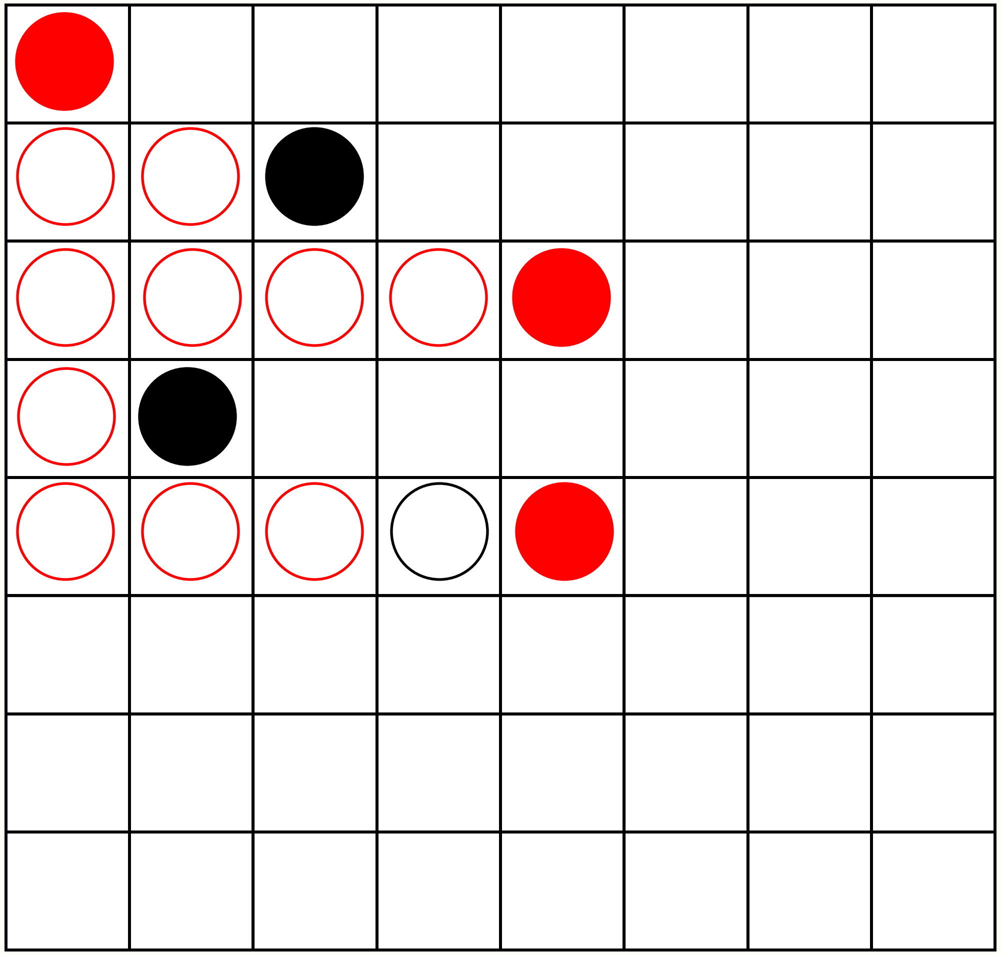
Tests 45 Backtracks 0+ 1 = 1

Backtracking



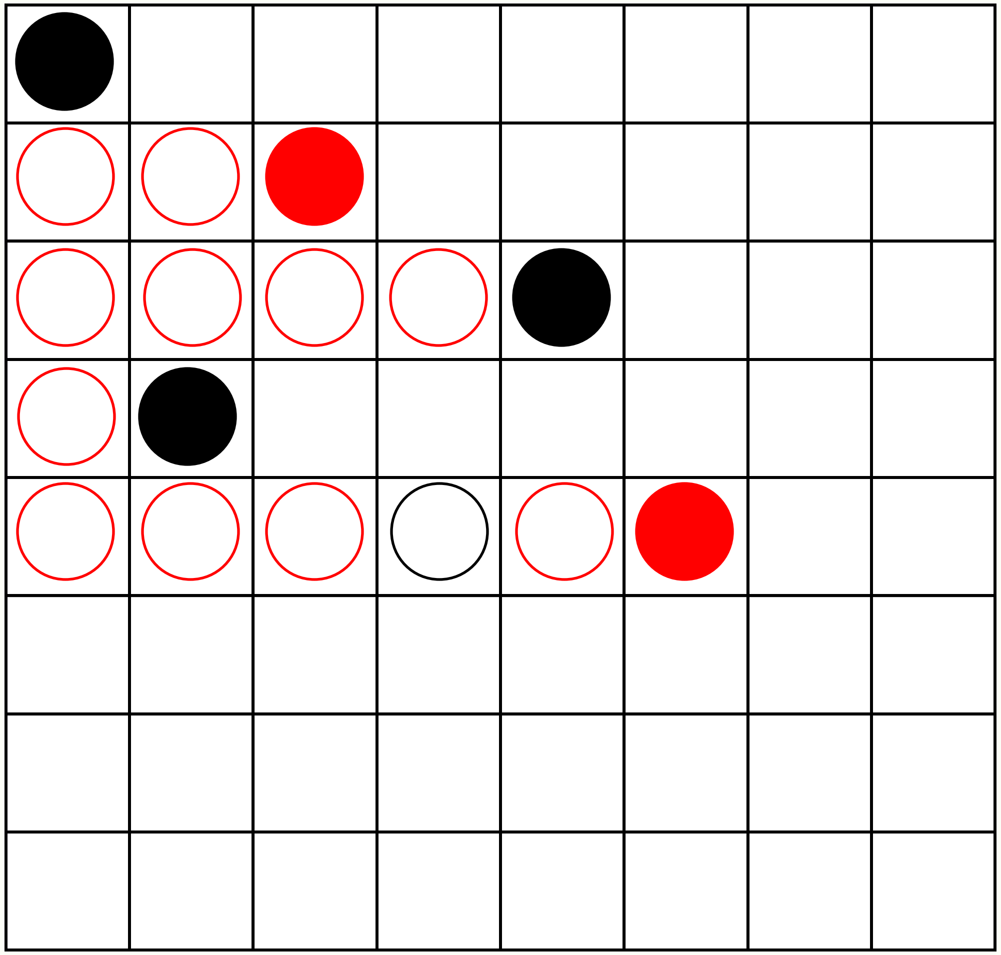
Tests 45 **Backtrackings 1**

Backtracking



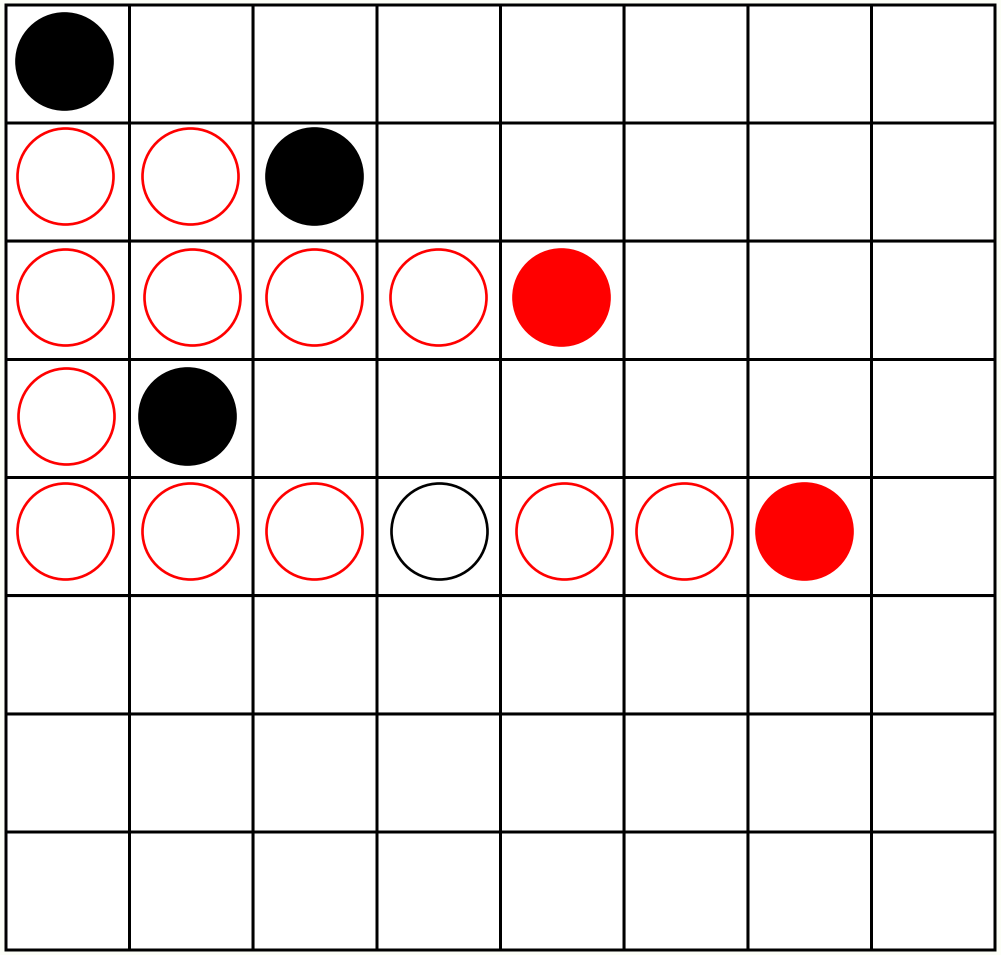
Tests 45 + 1 = 46 **Backtracks 1**

Backtracking



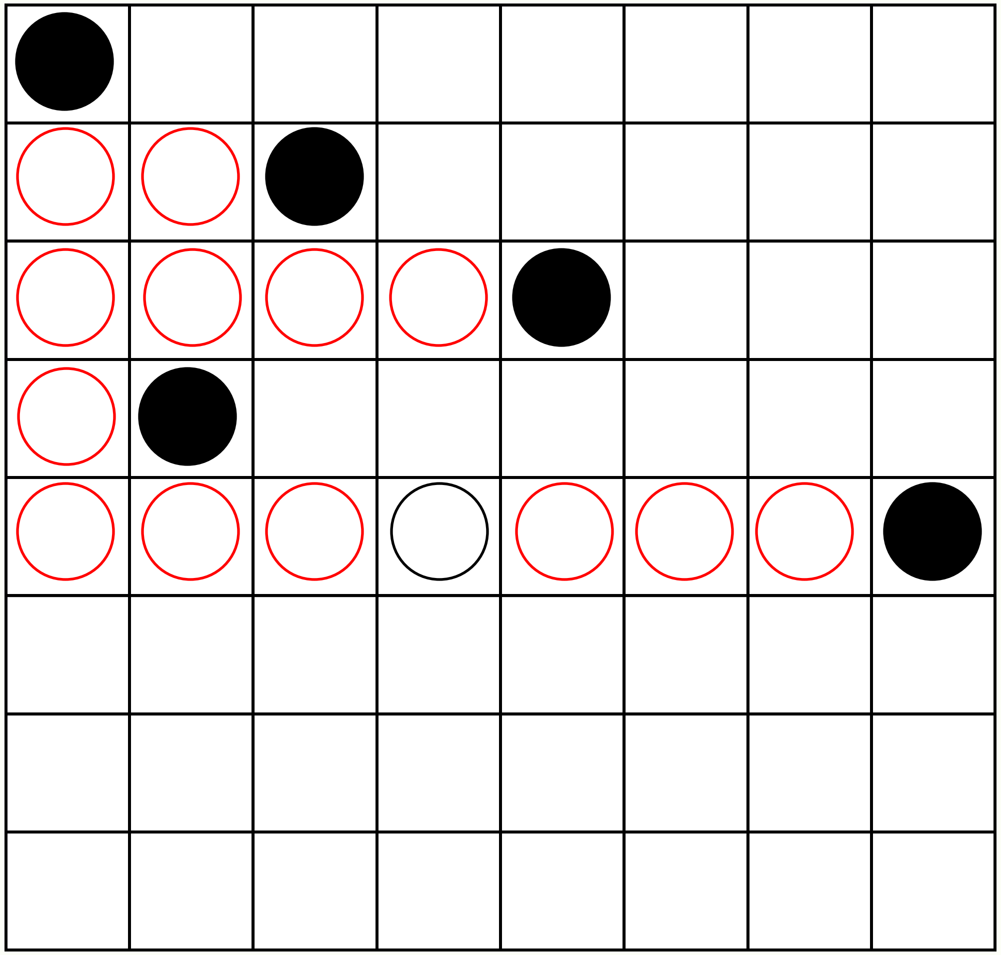
Tests 46 + 2 = 48 **Backtracks 1**

Backtracking



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

Backtracking



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

Backtracking

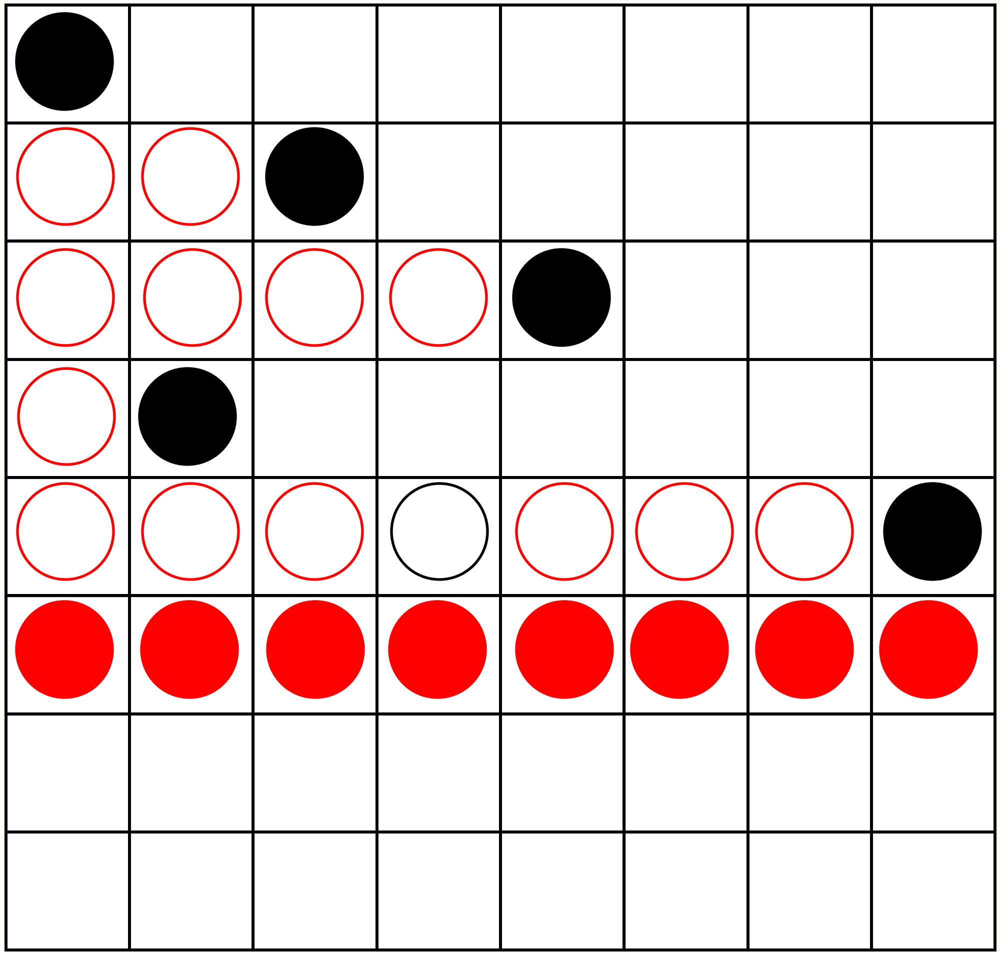
Q6 Fails

Backtracks to

Q5

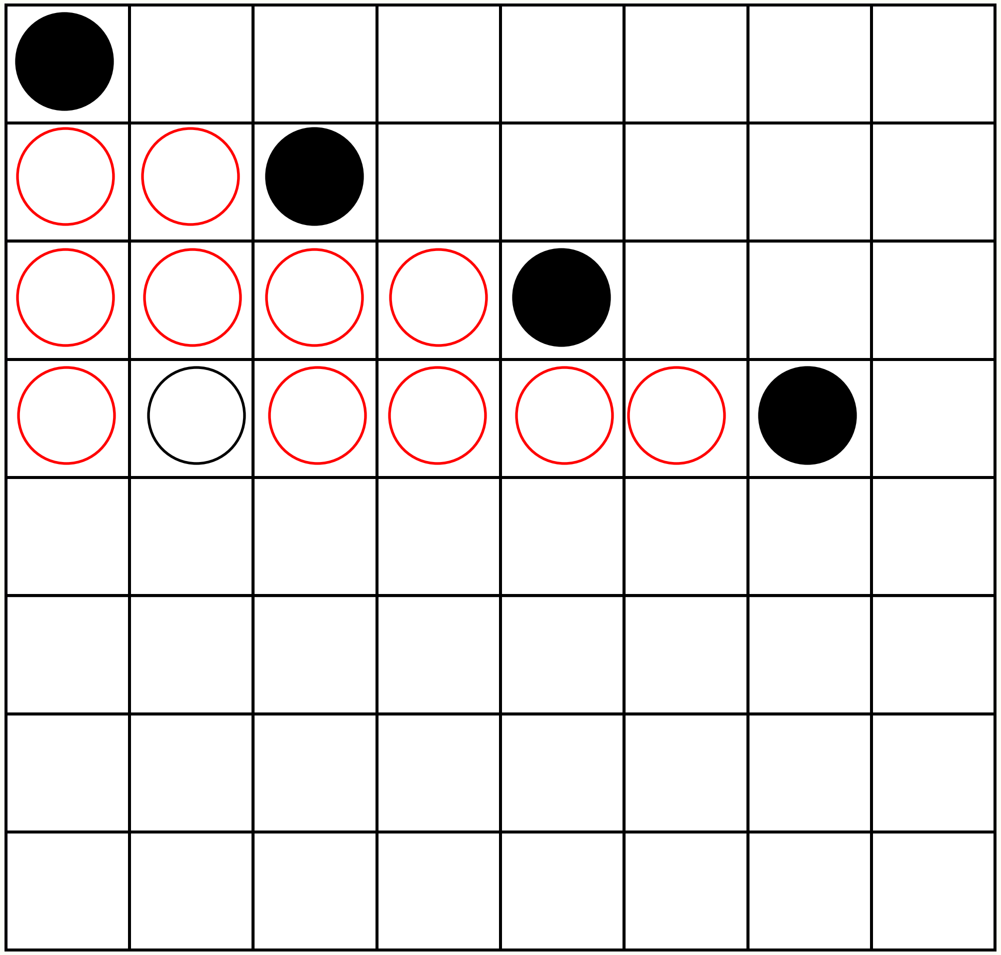
and next to

Q4



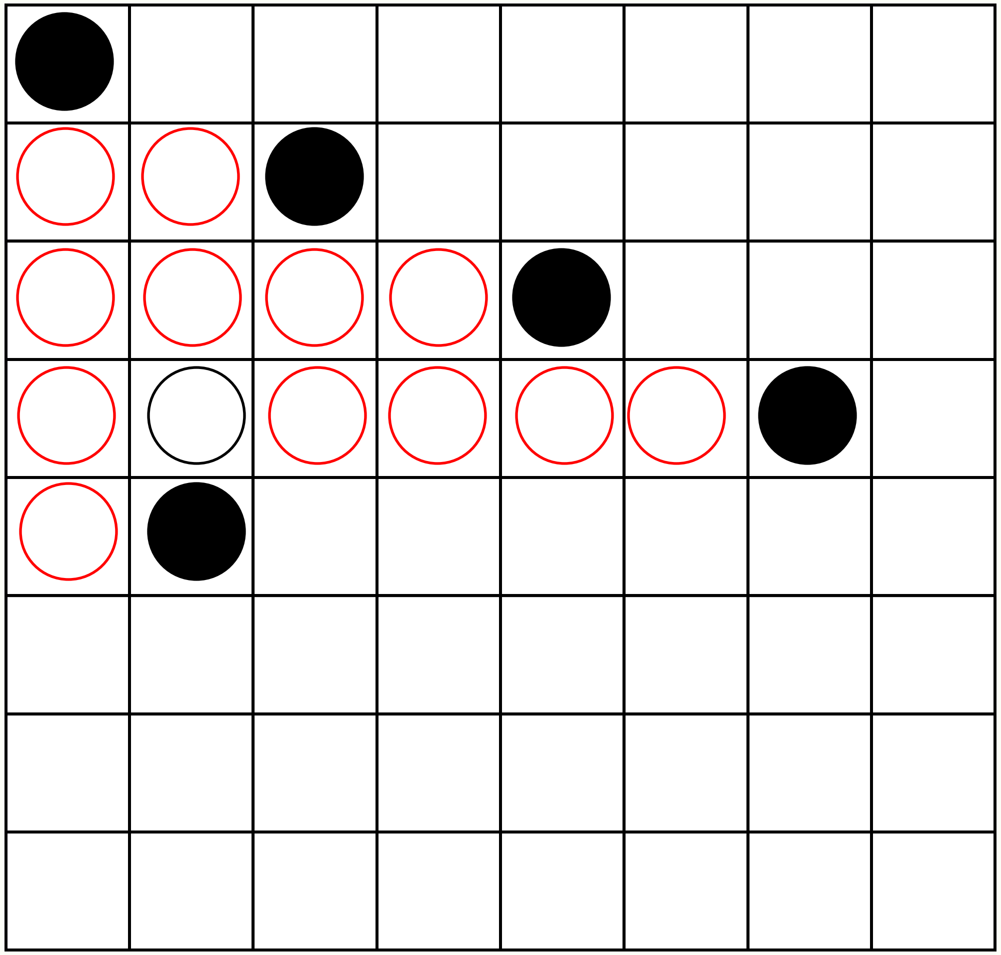
Tests 55+1+3+2+4+3+1+2+3 = 74   Backtracks 1+2 = 3

Backtracking



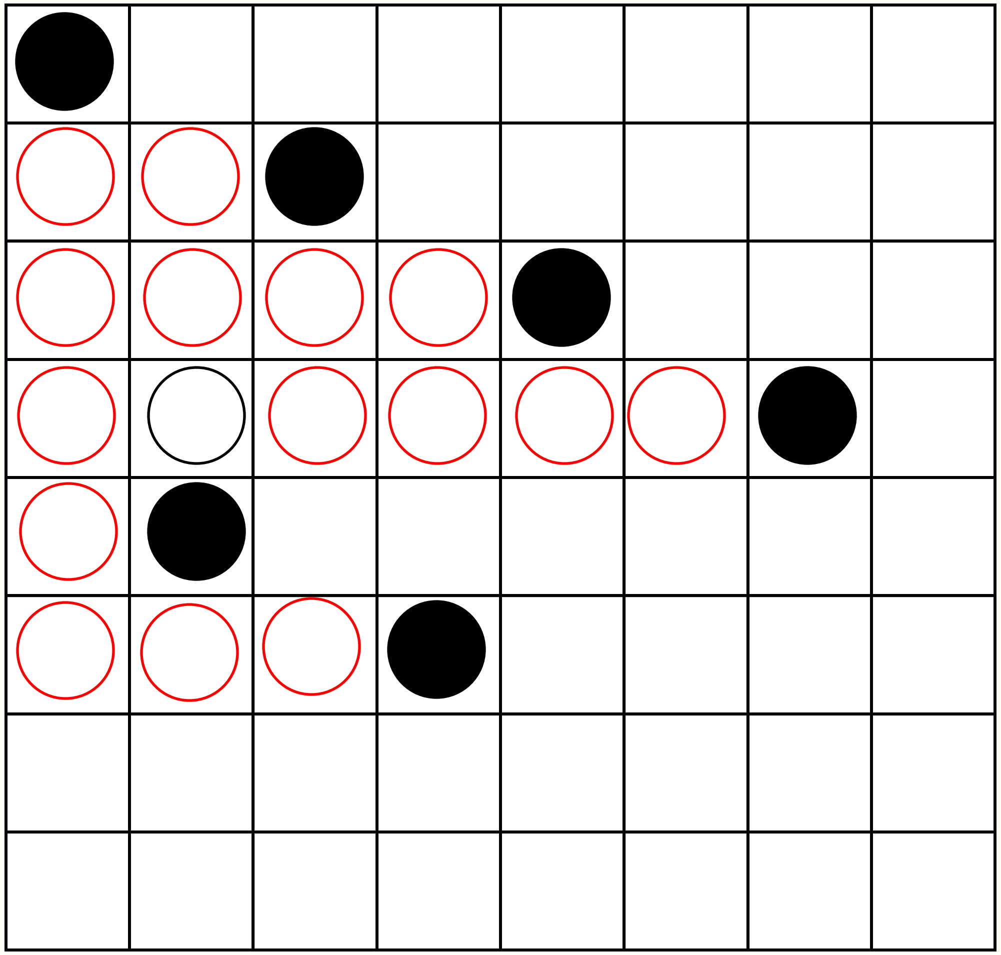
Tests 74+2+1+2+3+3= 85 **Backtracks 3**

Backtracking



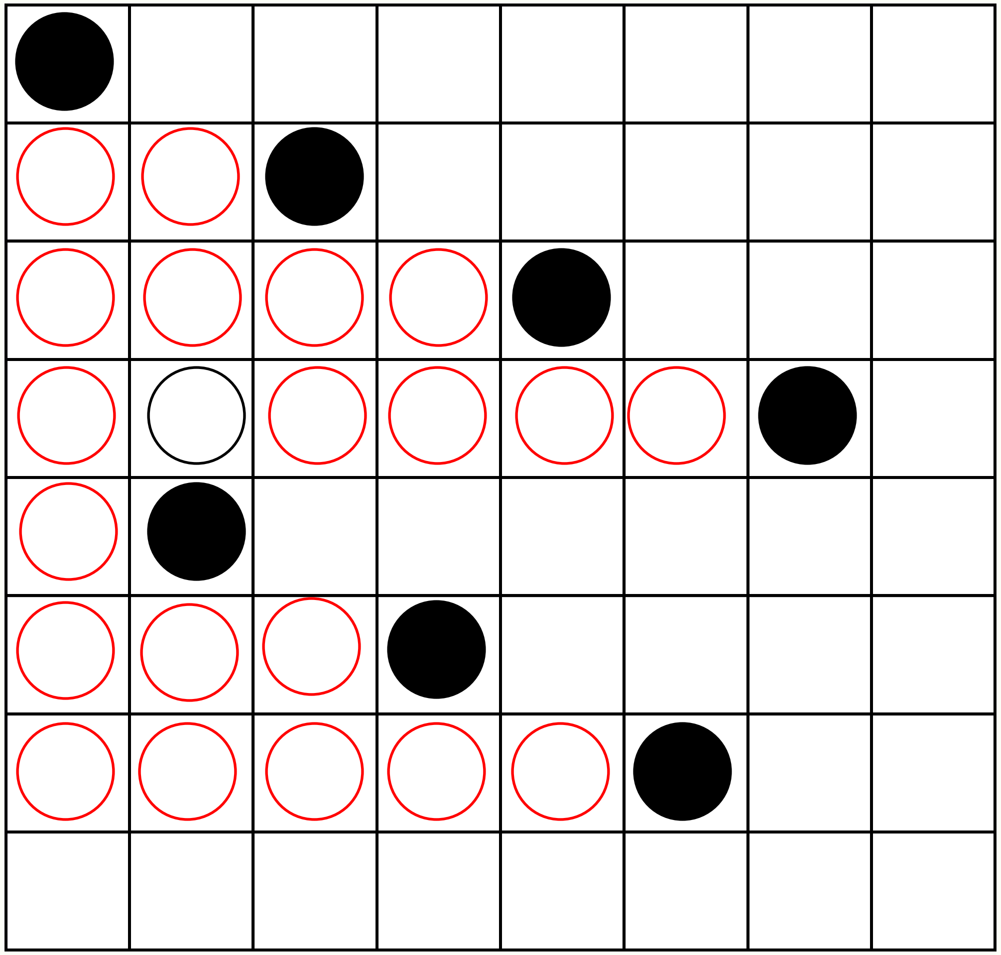
Tests 85 + 1 + 4 = 90 **Backtracks 3**

Backtracking



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

Backtracking



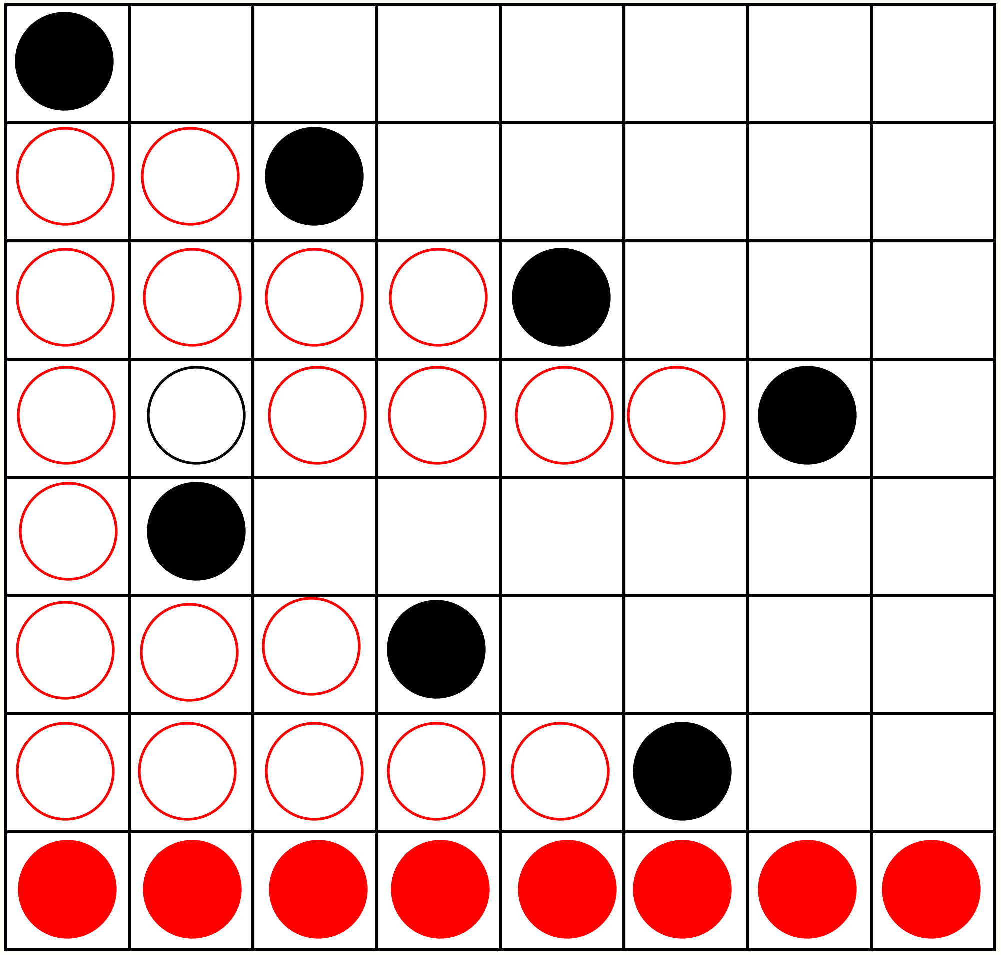
Tests 101+1+5+2+4+3+6= 122 **Backtracks 3**

Backtracking

Q8 Fails

Backtracks to

Q7



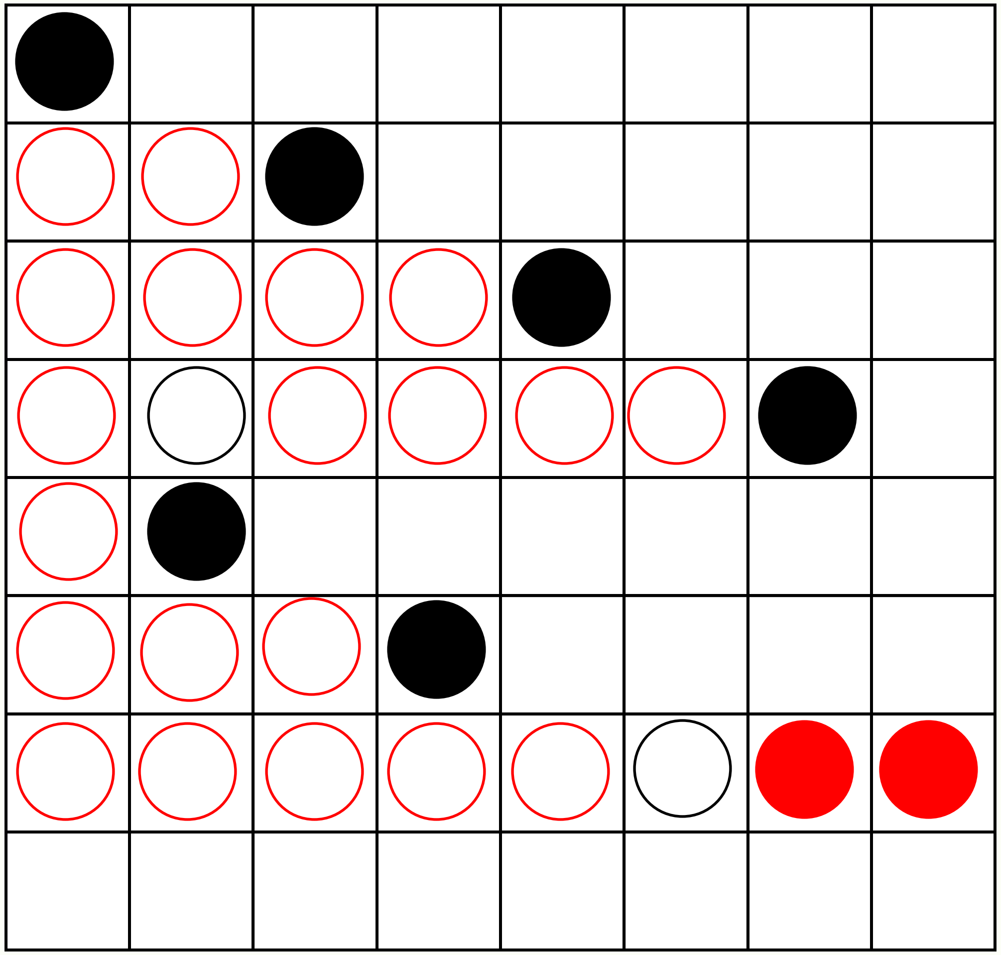
Tests 122+1+5+2+6+3+6+4+1= 150 Backtracks 3+1=4

Backtracking

Q7 Fails

Backtracks to

Q6



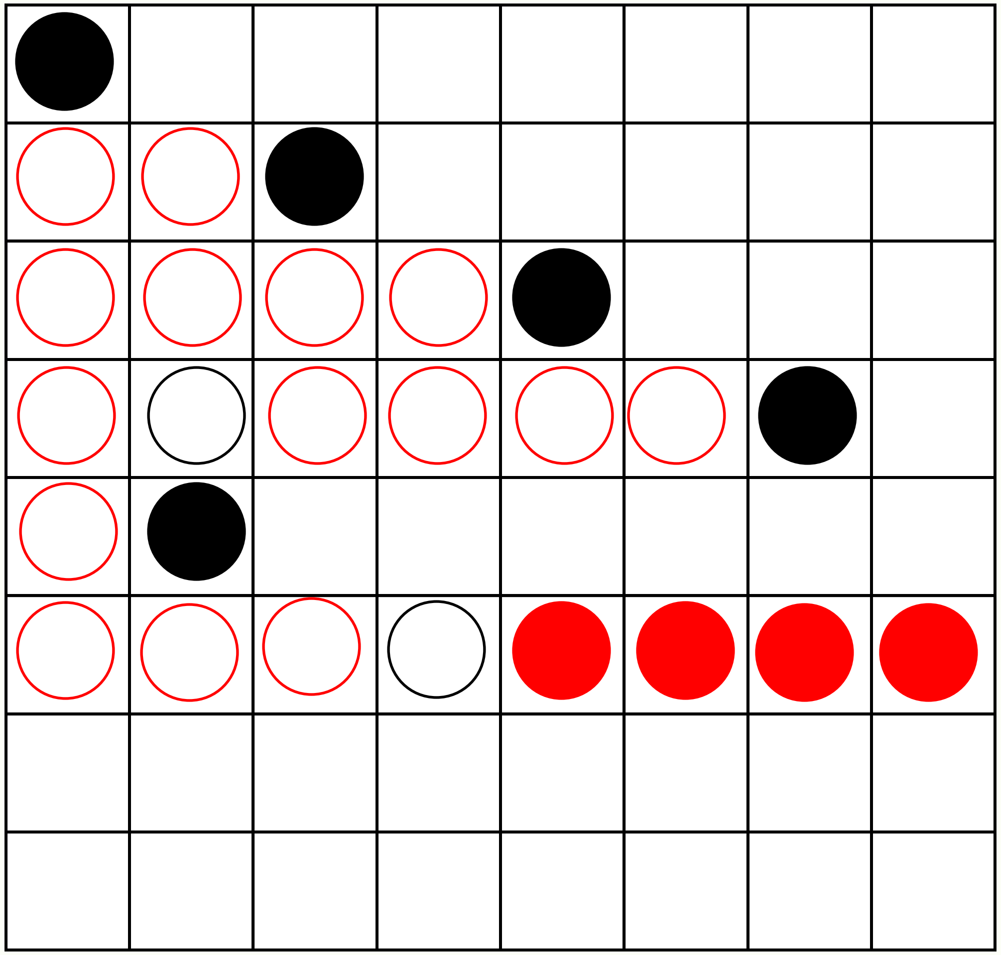
Tests 150+1+2= 153 Backtracks 4+1=5

Backtracking

Q6 Fails

Backtracks to

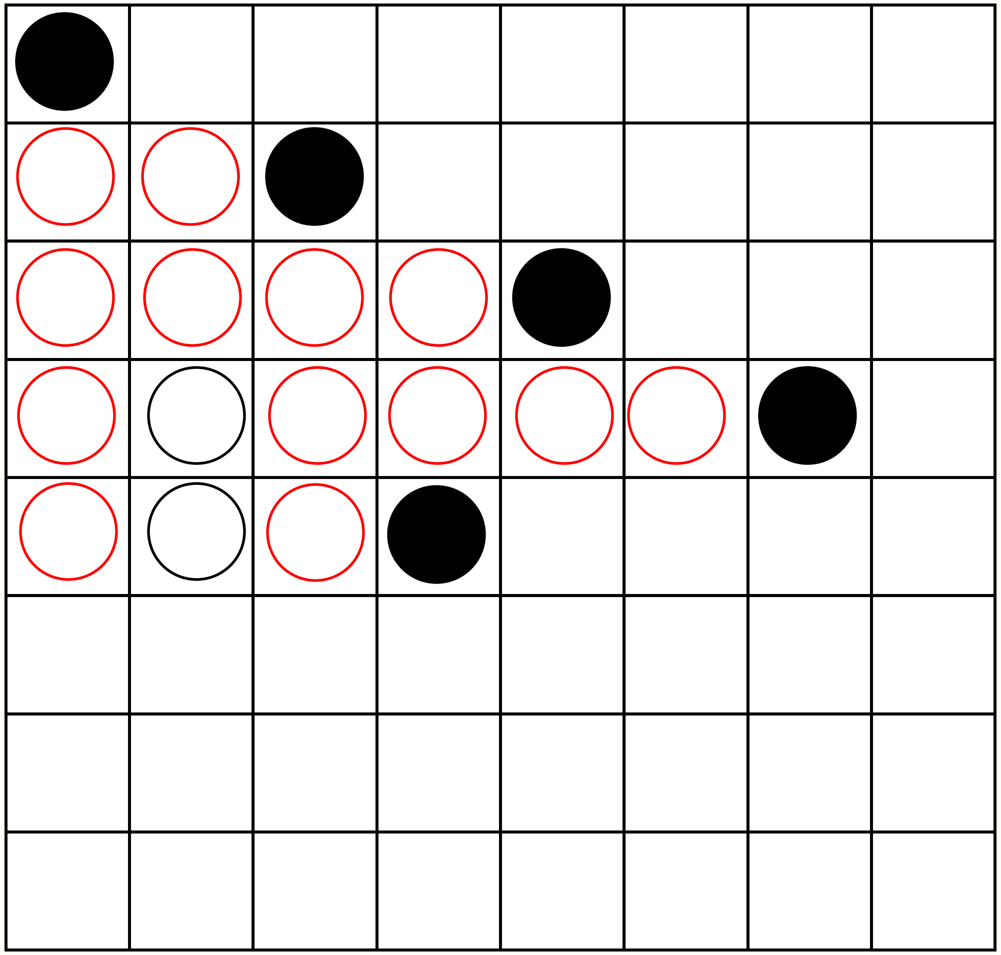
Q5



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

Backtracks 5+1=6

Backtracking



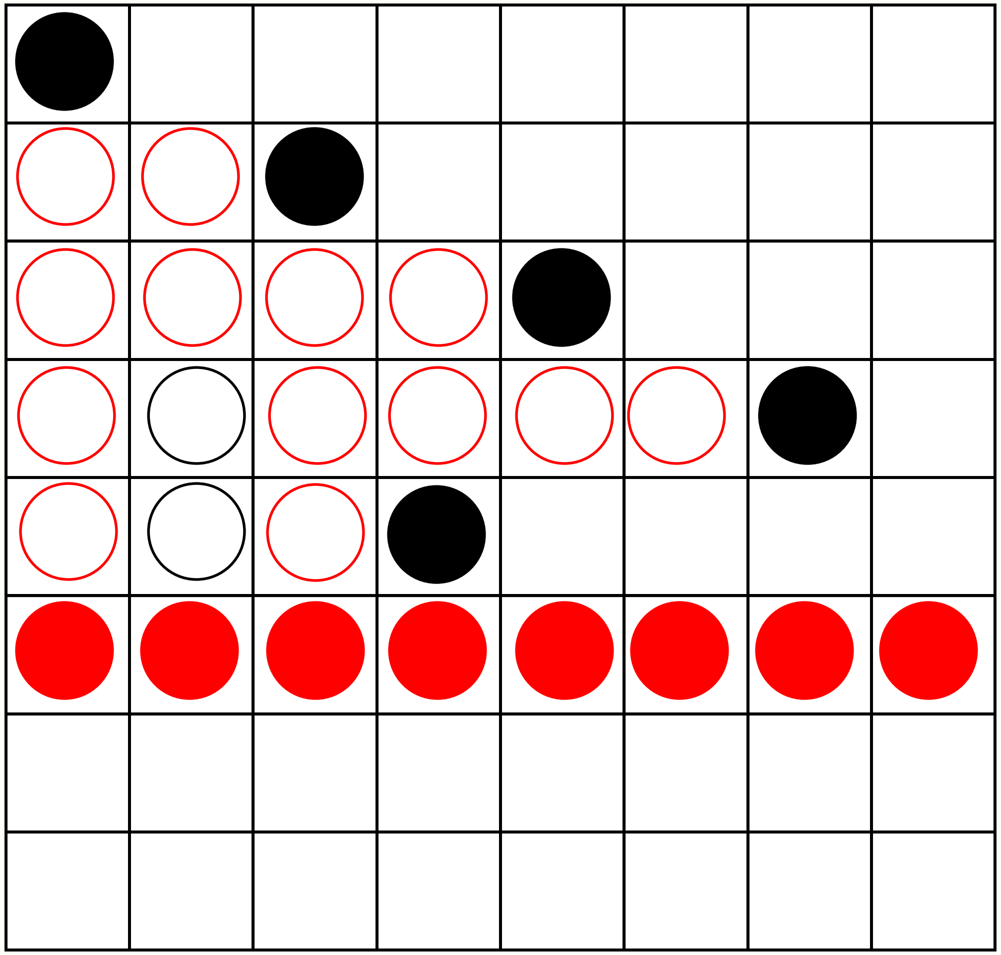
Tests 162+2+4= 168 **Backtracks 6**

Backtracking

Q6 Fails

Backtracks to

Q5



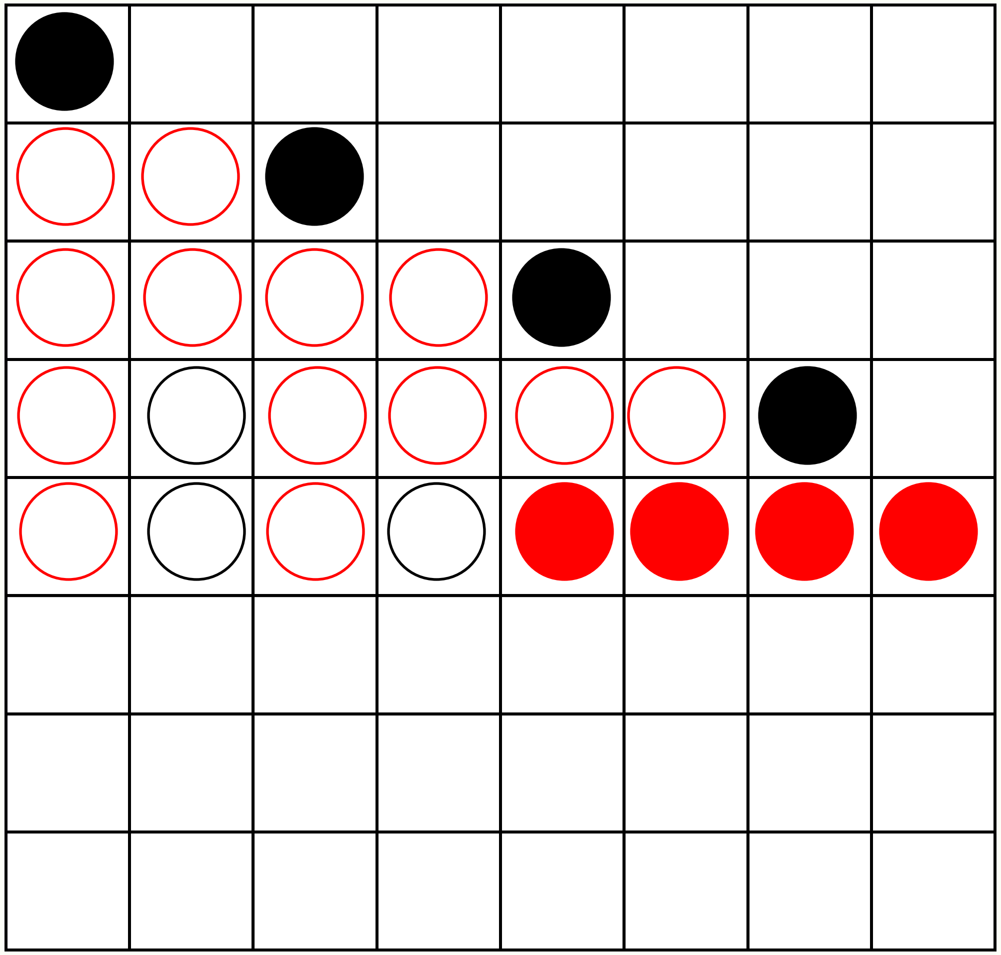
Tests 168+1+3+2+5+3+1+2+3= 188 Backtracks 6+1 = 7

Backtracking

Q5 Fails

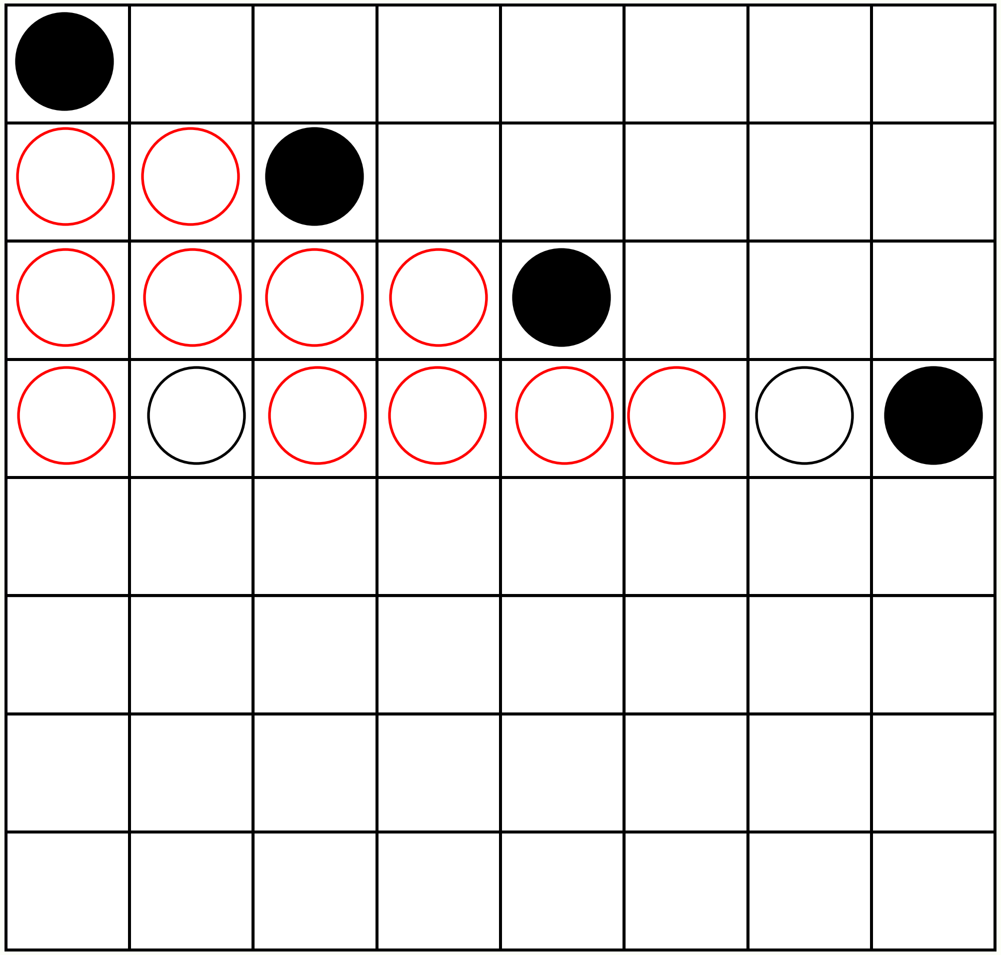
Backtracks to

Q4



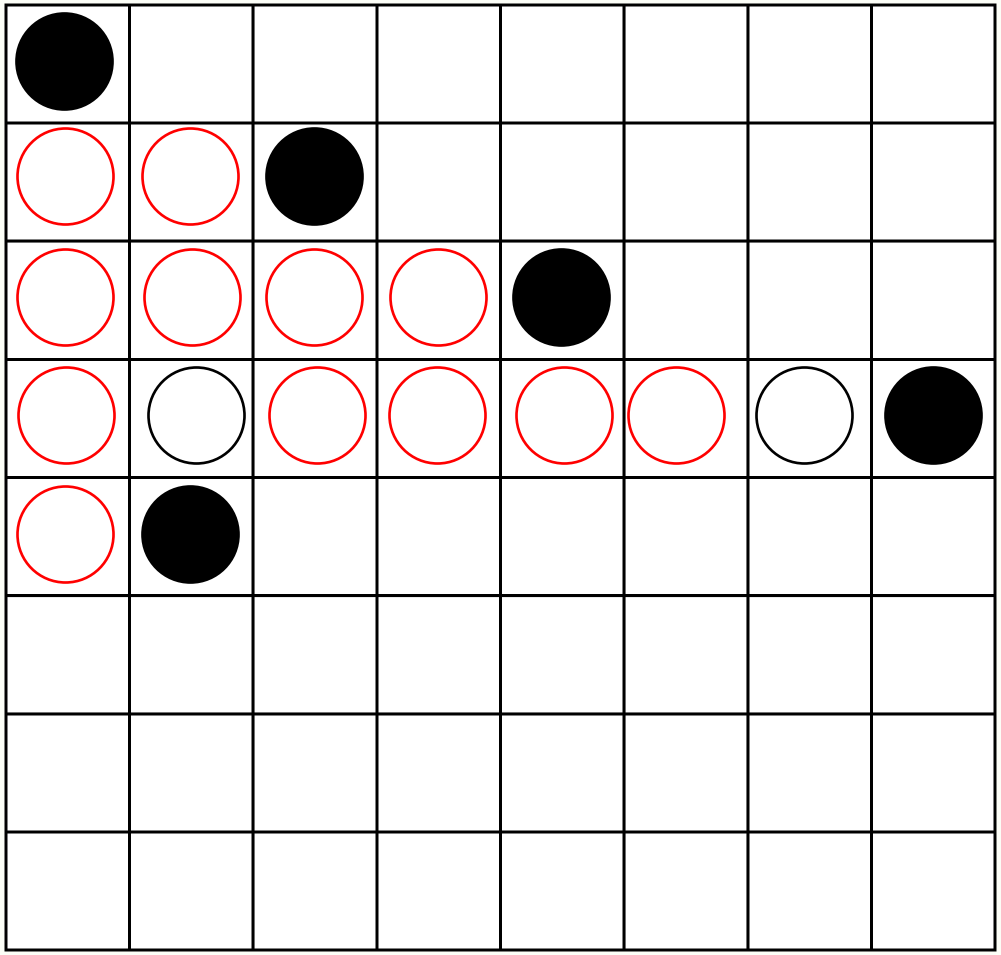
Tests 188+1+2+3+4= 198 Backtracks 7+1=8

Backtracking



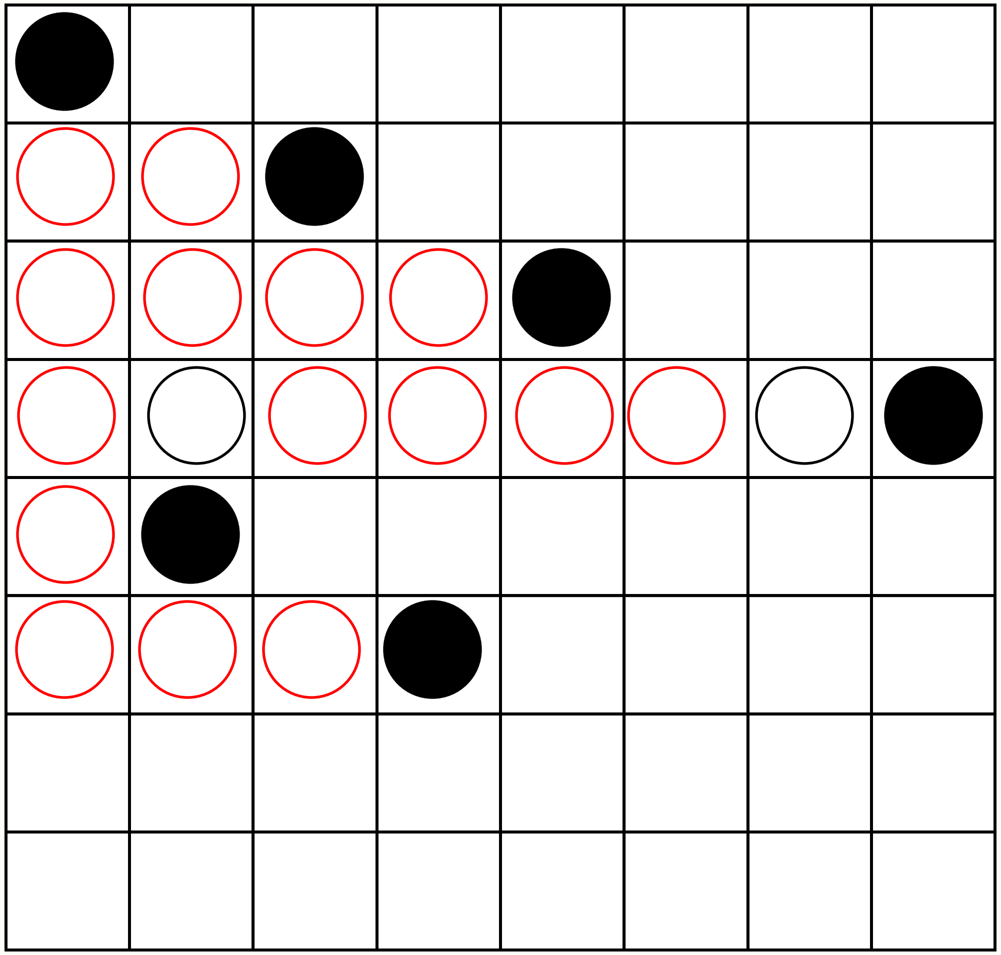
Tests 198 + 3 = 201 **Backtracks 8**

Backtracking



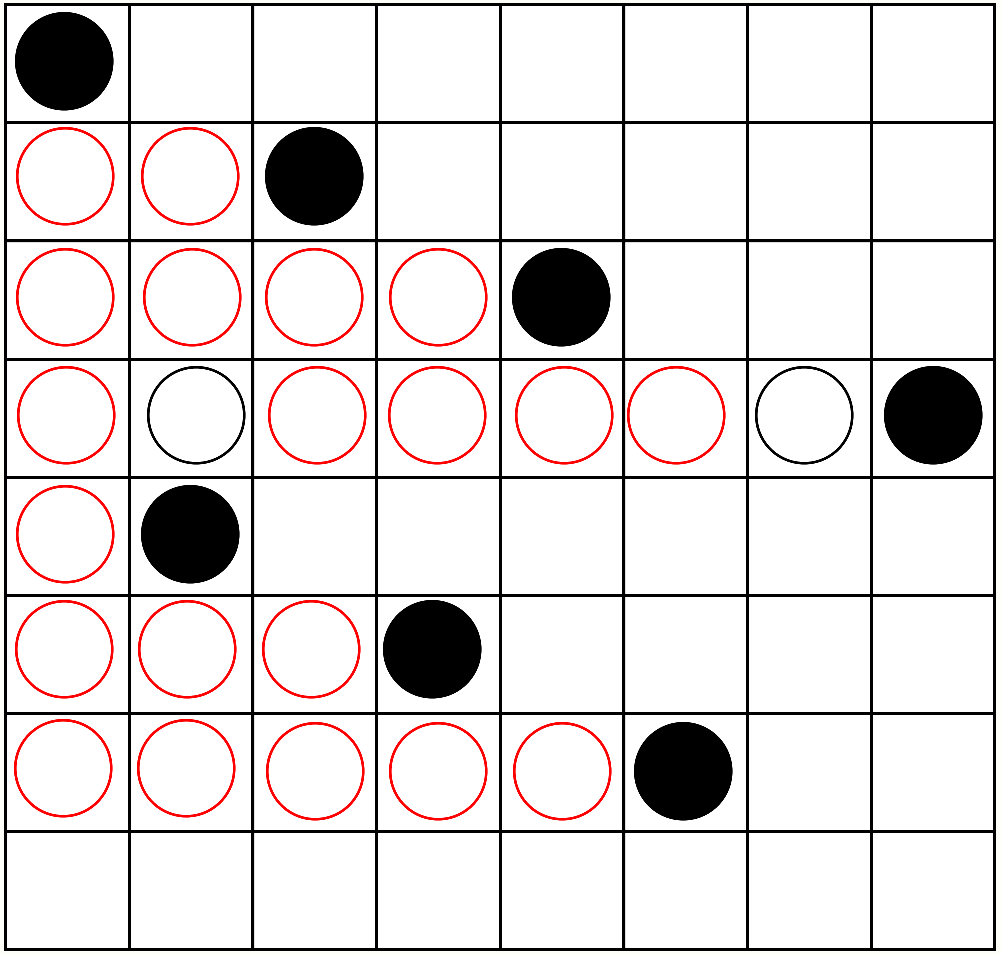
Tests 201+1+4 = 206 **Backtracks 8**

Backtracking



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

Backtracking

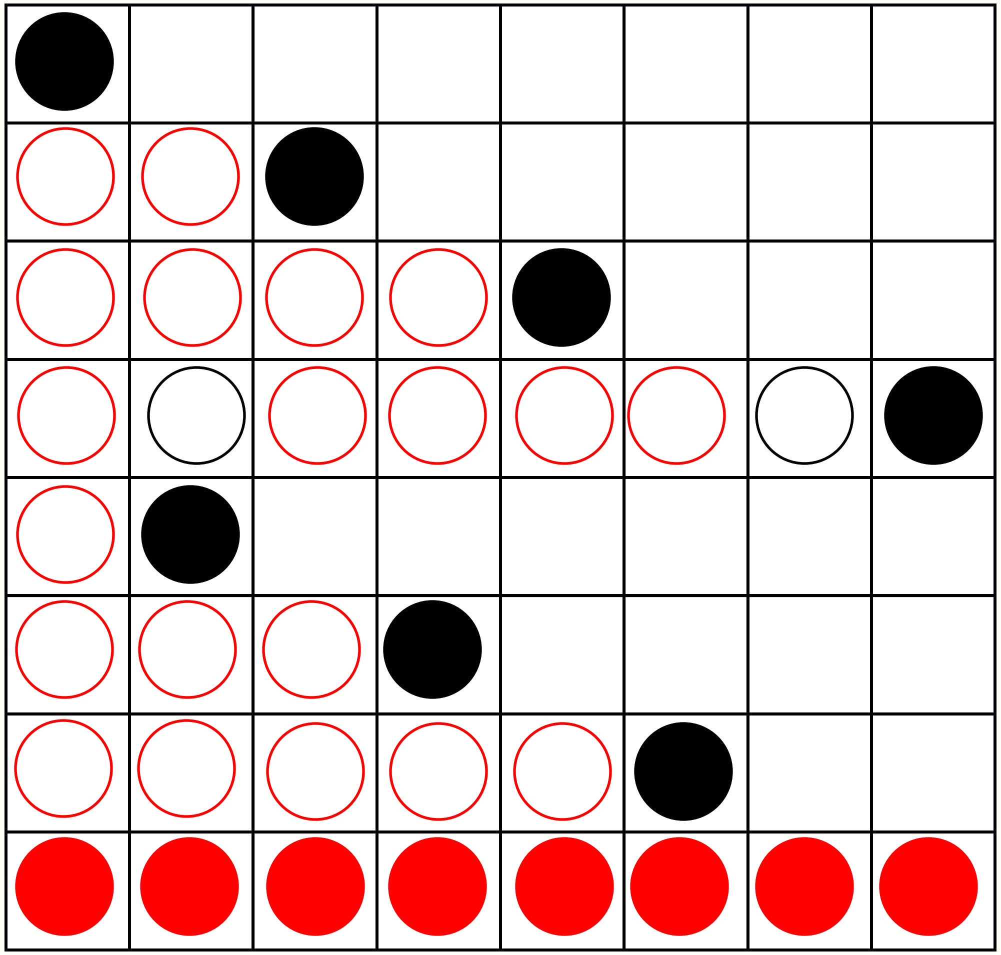


Tests 217+1+5+2+5+3+6 = 239 Backtracks 8

Backtracking

Q8 Fails

Backtracks to Q7



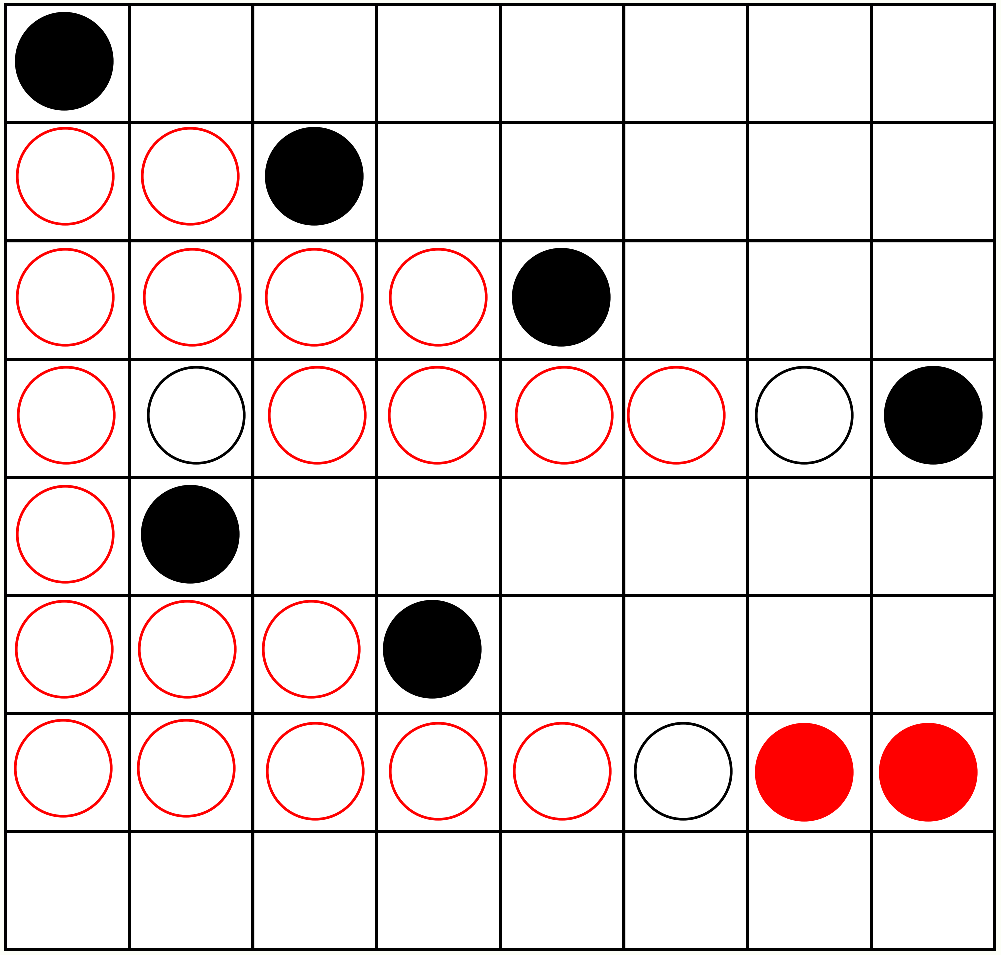
Tests 239+1+5+2+4+3+6+7+7= 274 Backtracks 8+1 = 9

Backtracking

Q7 Fails

Backtracks to

Q6



Tests 274+1+2= 277

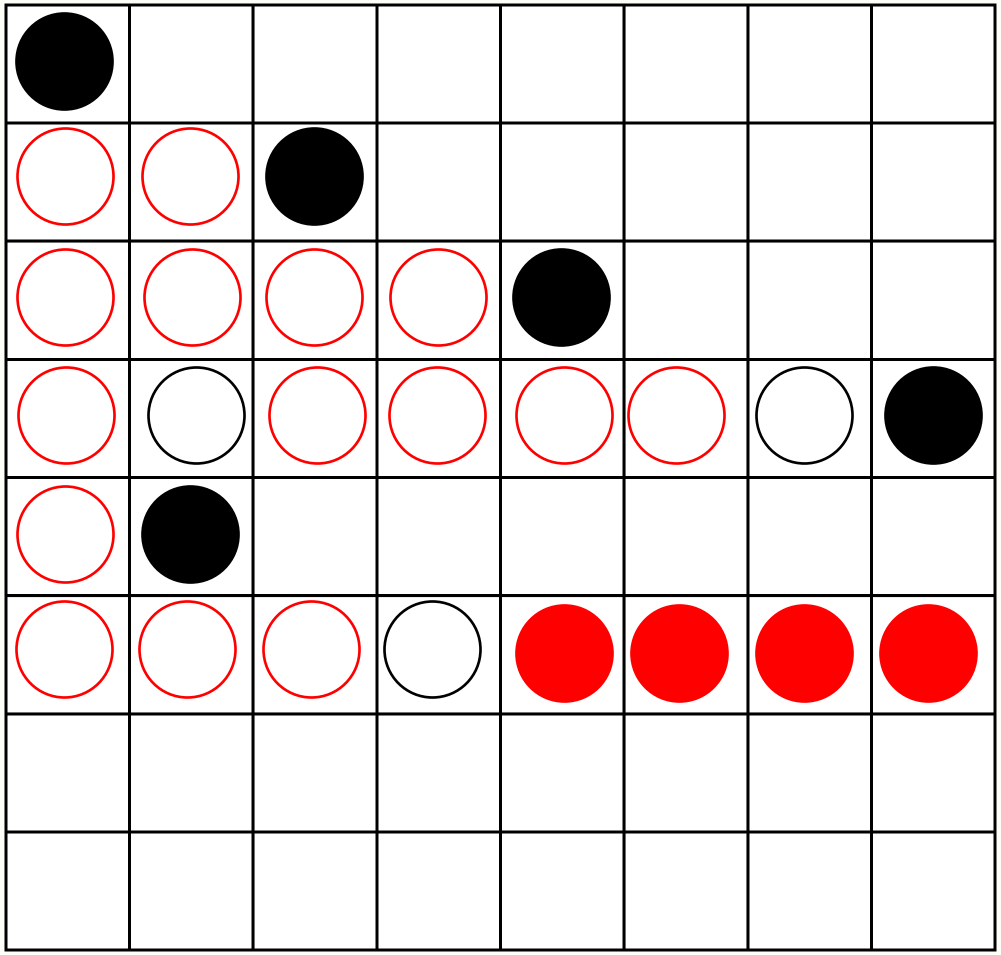
Backtracks 9+1=10

Backtracking

Q6 Fails

Backtracks to

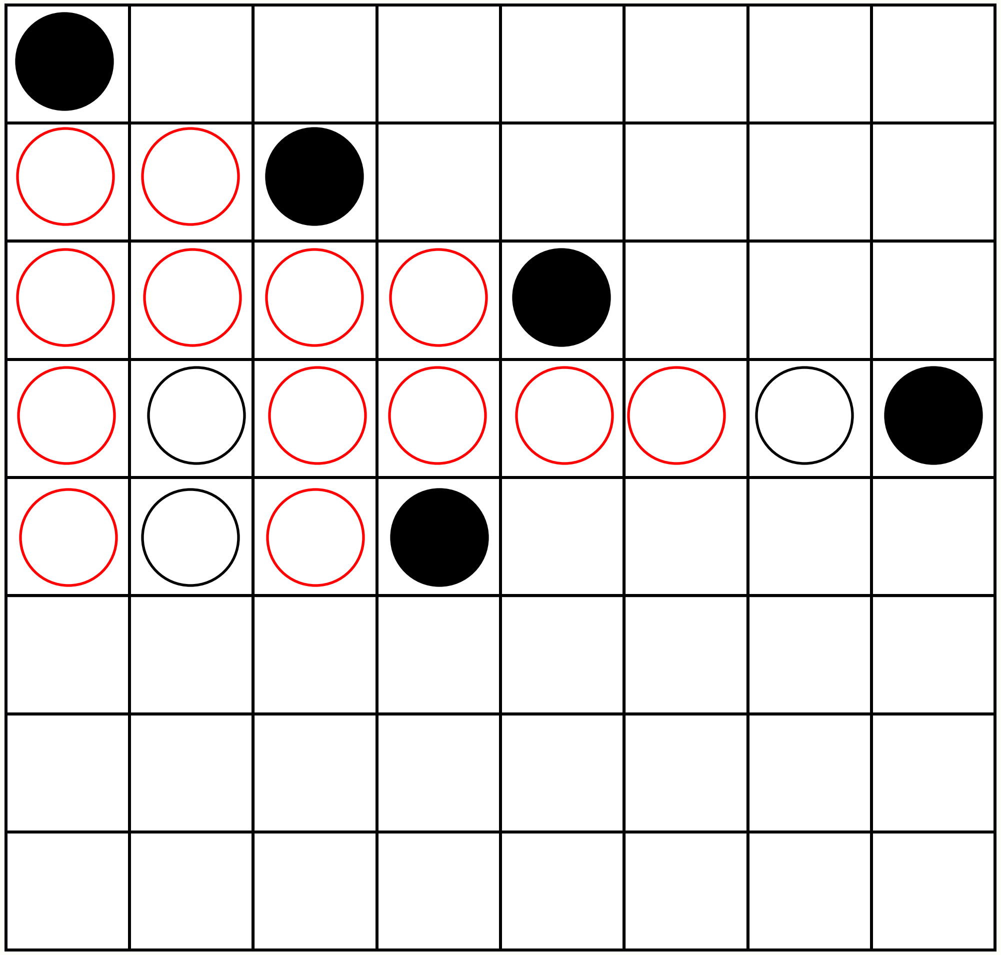
Q5



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

Backtracks 10+1=11

Backtracking



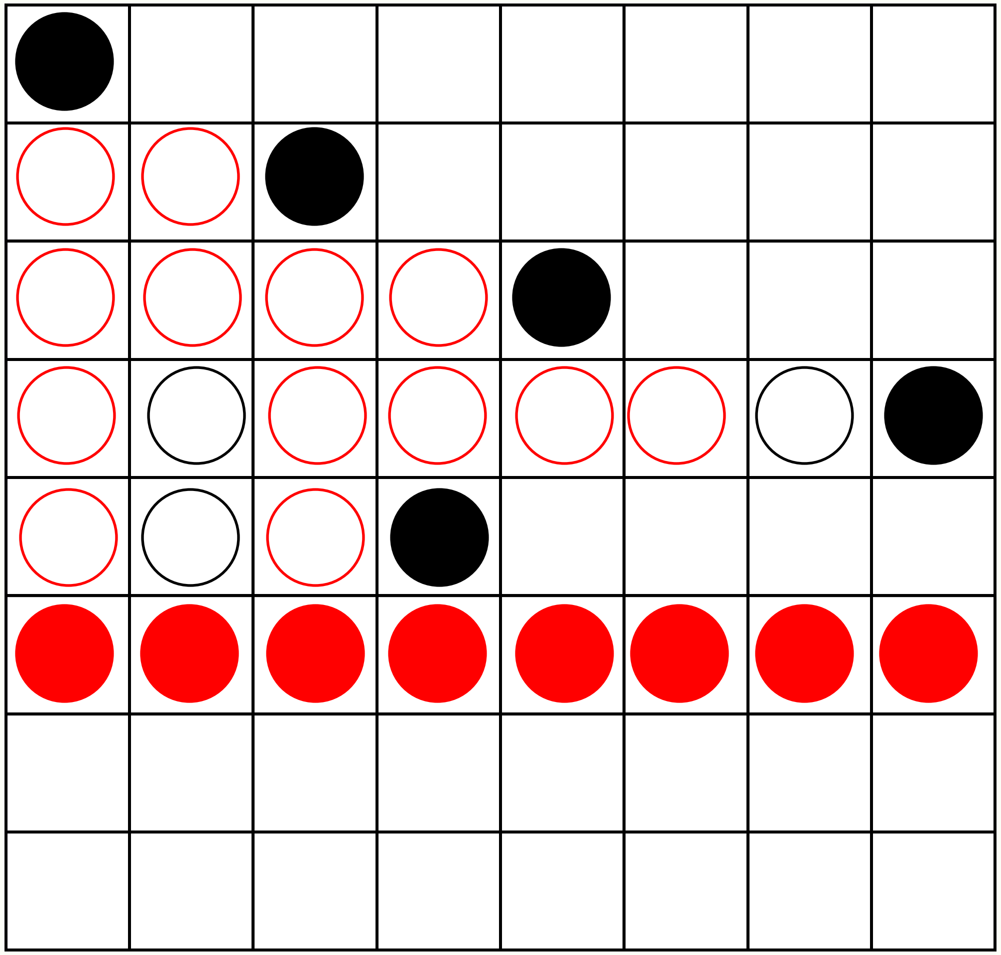
Tests 286+2+4= 292 **Backtracks 11**

Backtracking

Q6 Fails

Backtracks to

Q5



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

Backtracking

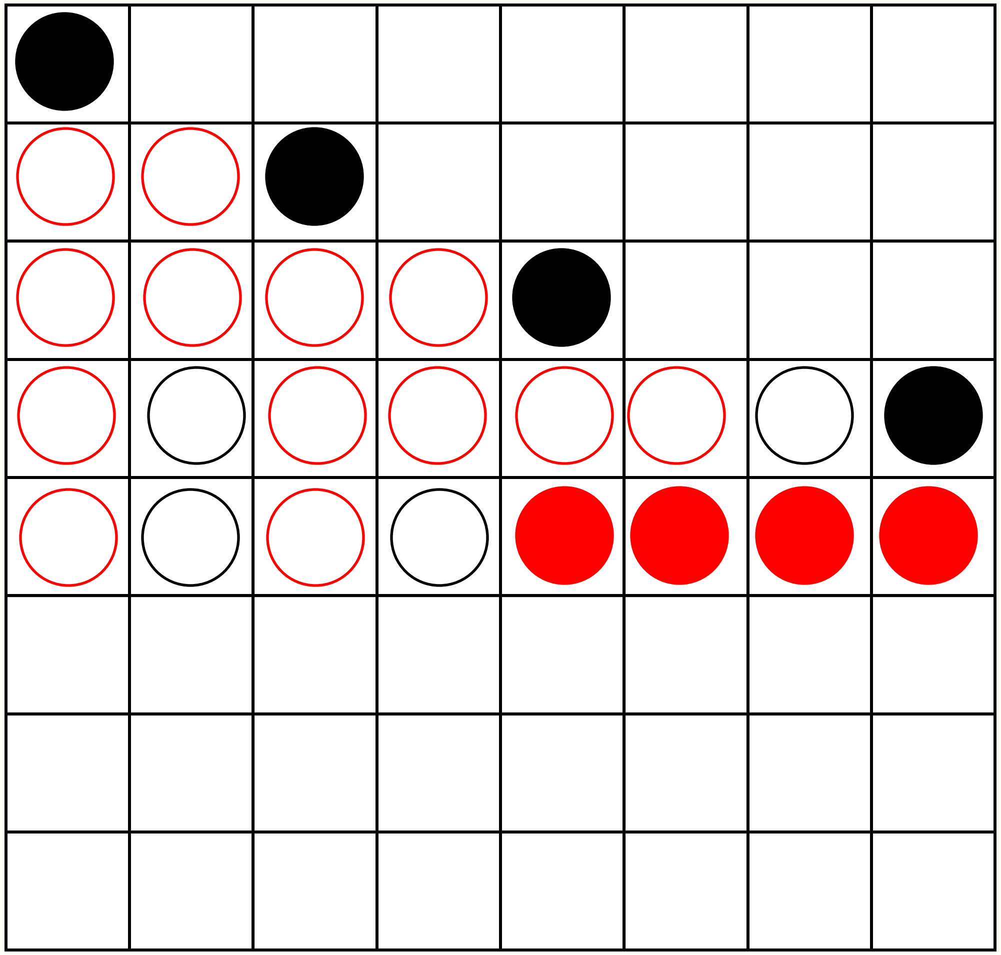
Q5 Fails

Backtracks to

Q4

and next to

Q3



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

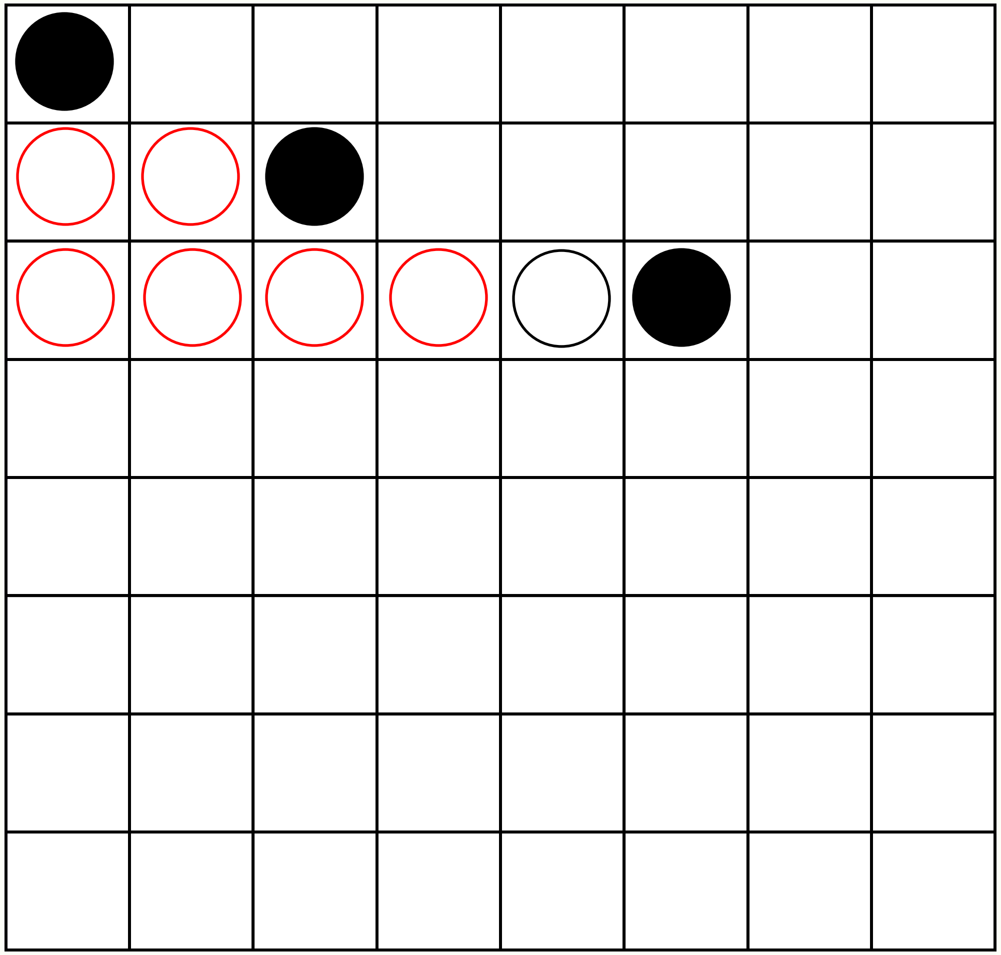
Backtracking

$Q_1 = 1$

$Q_2 = 3$

$Q_3 = 5$

Impossible !

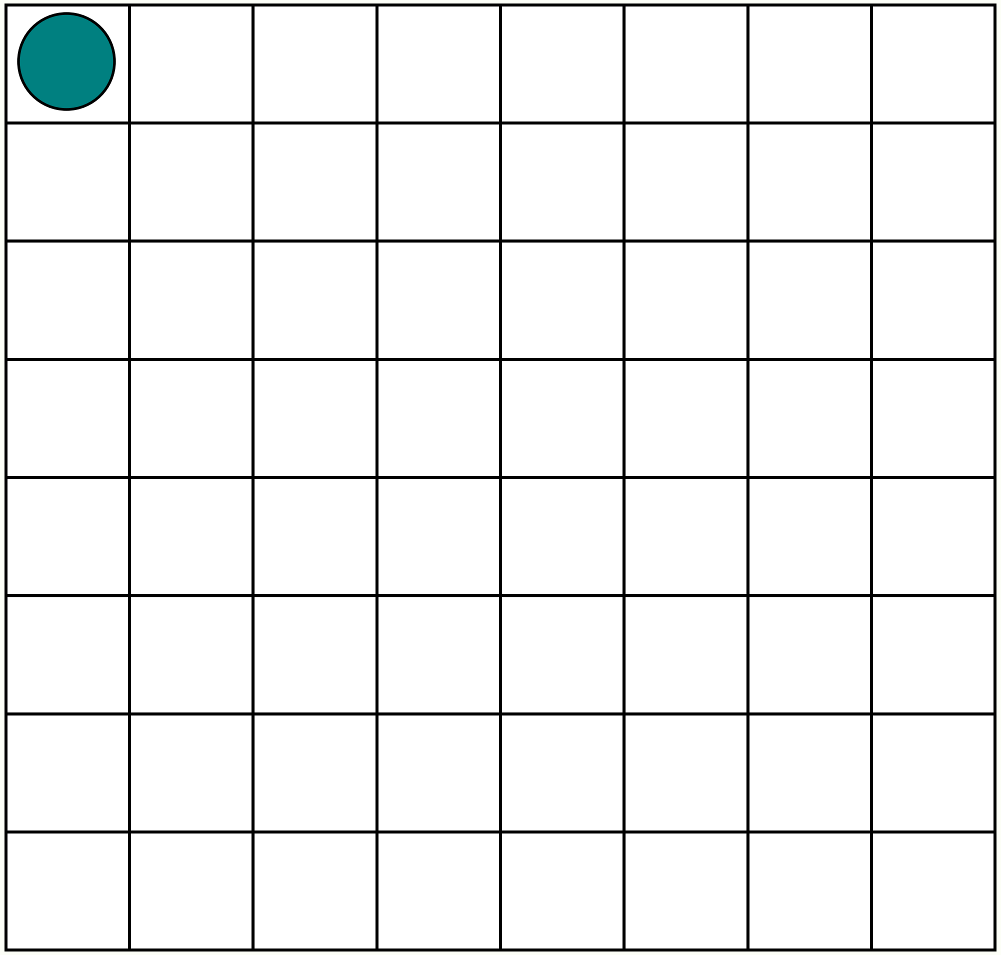


Tests 322 + 2 = 324 **Backtracks 14**

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 variable, the consequences of such assignment are taken into account in all the constraints it apperas 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 amopng 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 #\= 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**

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

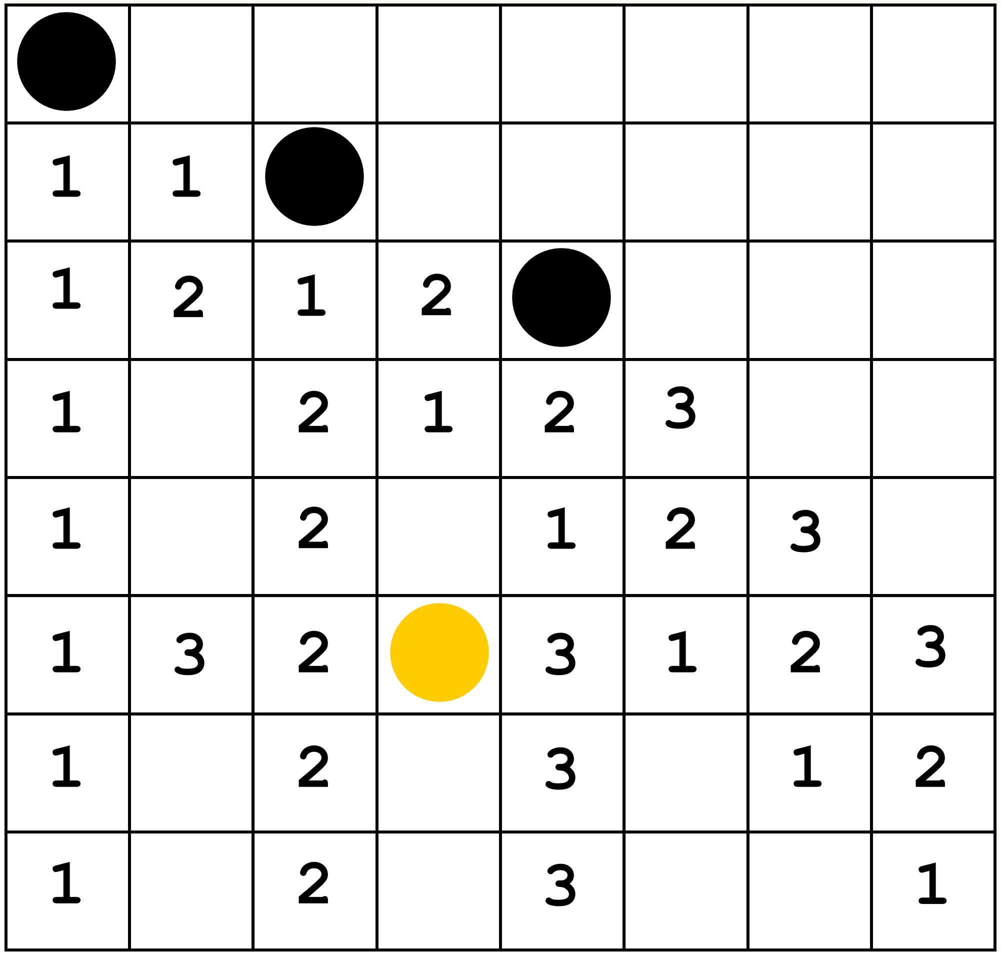
Tests 92 + 21 = 113 **Backtracks 0**

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

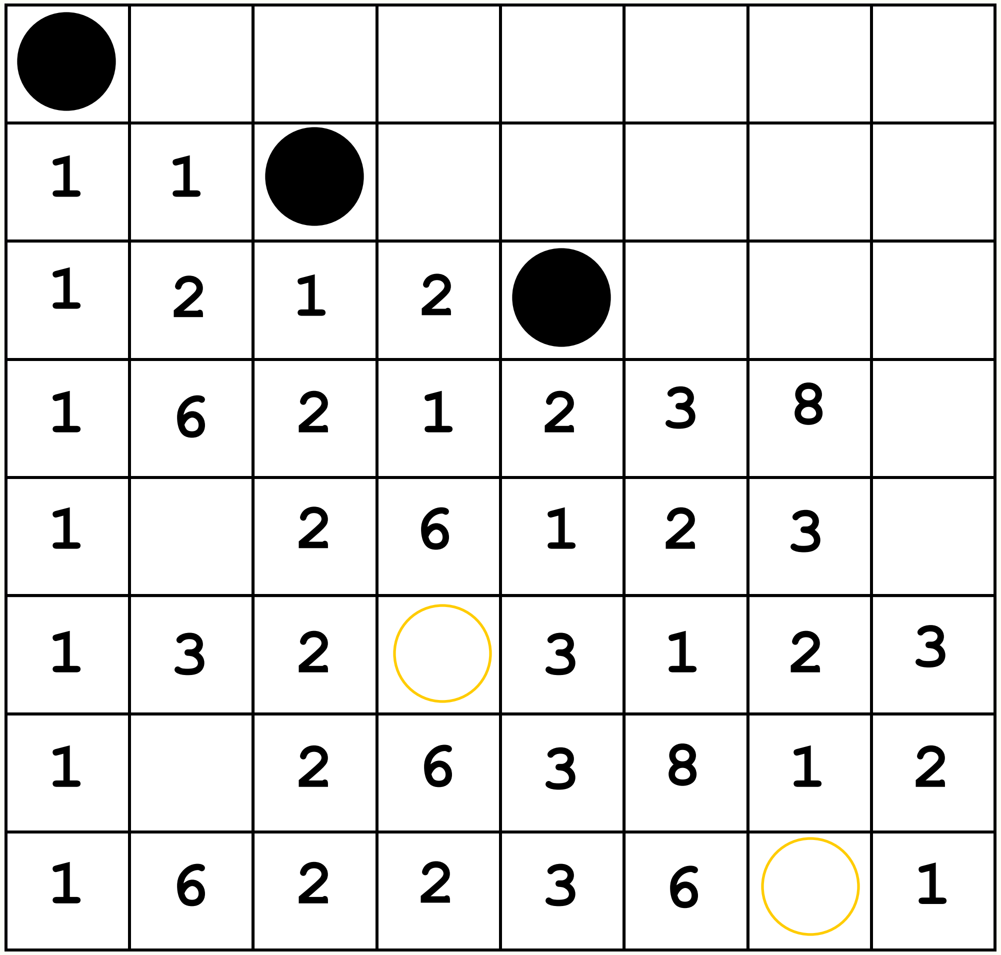
Tests 126+2+2+2=132 **Backtracks 0**

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

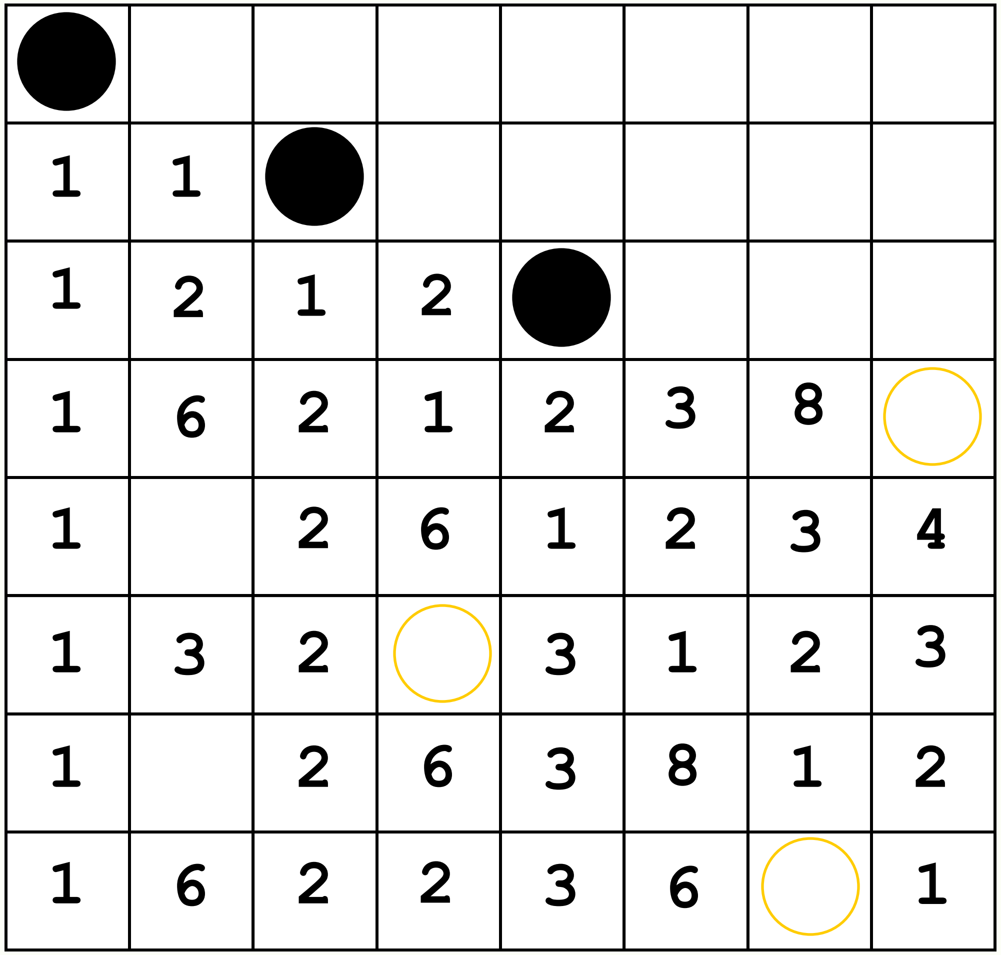
Tests 132+2+1=135 **Backtracks 0**

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_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**

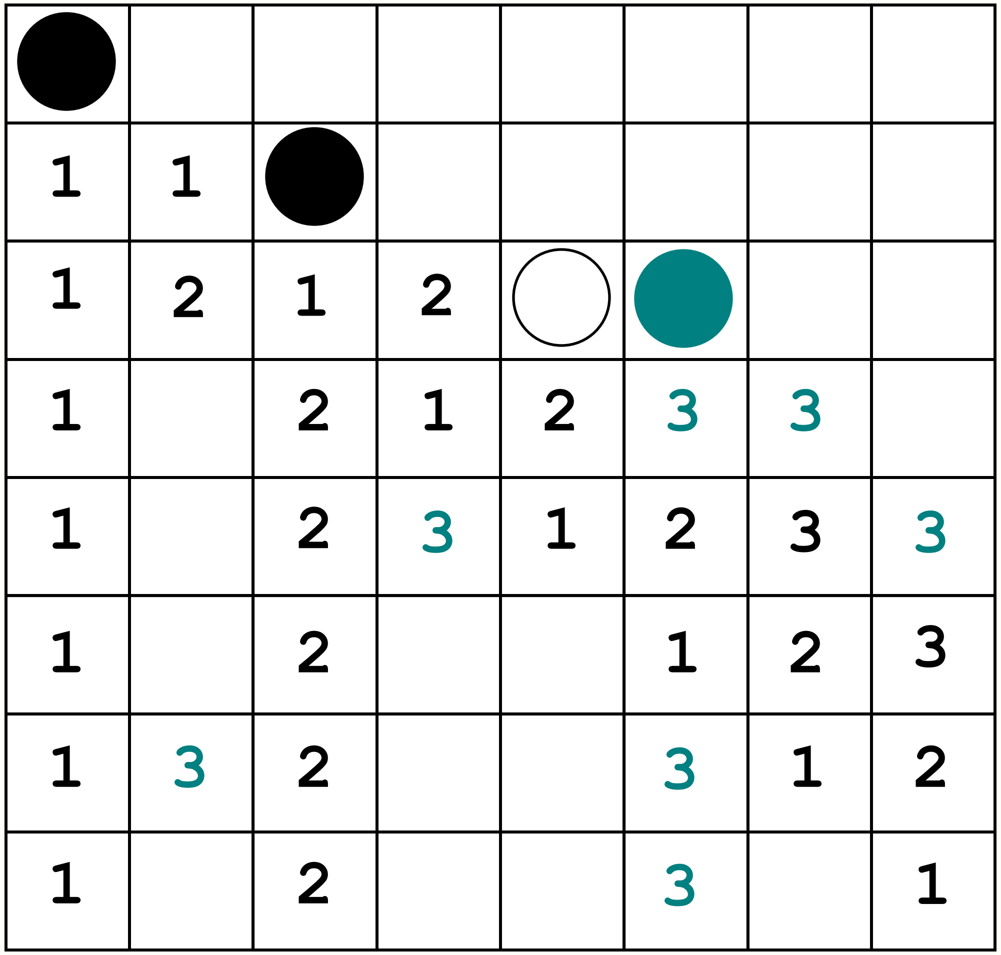
Backtracking + Propagation + Heuristics

$Q_1 = 1$

$Q_2 = 3$

$Q_3 = 5$

Impossible !



●							
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

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.

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

- **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 $\mathbf{c_{ijk}}$ 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, constraint $\mathbf{c_{13}}$ 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

$$c_{13} = \{<1,2>,<1,4>,<2,1>,<2,3>,<3,2>,<3,4>,<4,1>,<4,3>\}.$$

  - **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 **<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 (**Constrained Optimisation Problem**):

- A constrained optimization problem (**COP**) 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 COP **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 (binary) constraints of a problem may be considered as forming a special **constraint graph**.

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

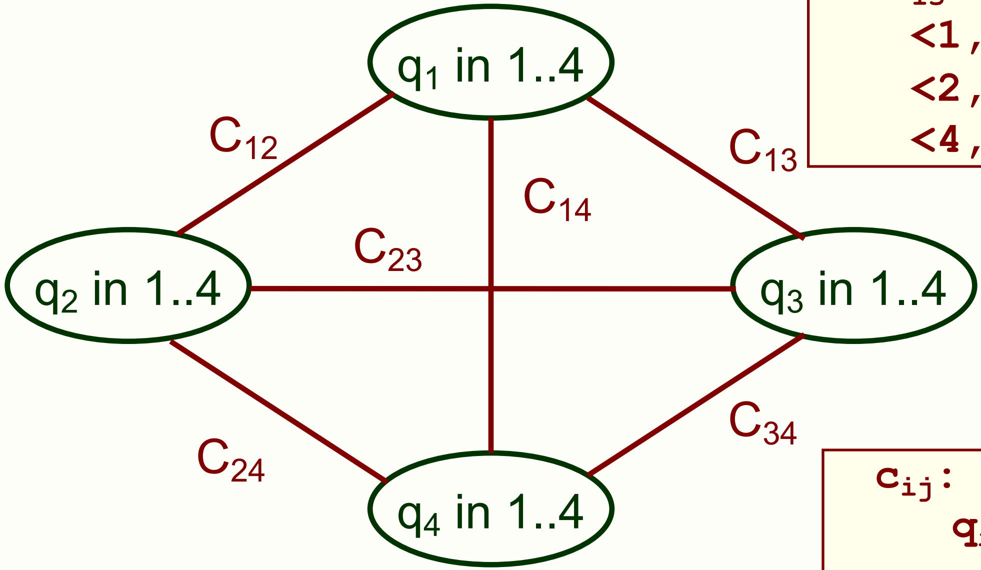
- The **Constraint Graph** or **Constraint Network** of a binary constraint satisfaction problem is defined as follows
  - There is a node for each of the variables of the problem.
  - For each (non-trivial) constraint of the problem, involving one or two variables, the graph contains an arc linking 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**:



```
c13:
 <1,2>, <1,4>, <2,1>,
 <2,3>, <3,2>, <3,4>,
 <4,1>, <4,3>
```

```
cij:
   qi \= qj
   qi + i \= qj + 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.

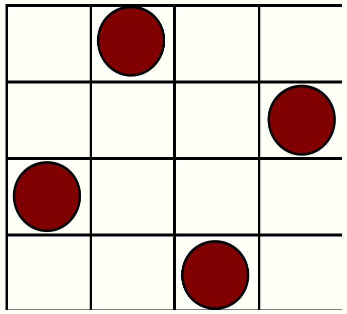
- Redundant values and labels increase the search space uselessly, and should thus be avoided. There is no point in testing a value that does not appear in any solution !

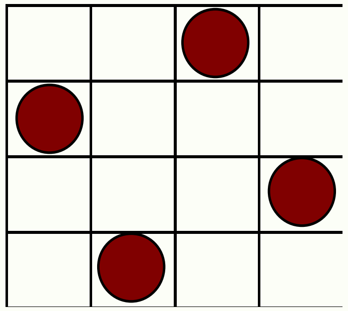
Constraints: Basic Concepts

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 $\mathbf{P_1 = <X_1, D_1, C_1>}$ and $\mathbf{P2 = <X_2, D_2, C_2>}$ are equivalent iff they have the same variables (i.e. $\mathbf{X_1 = X_2}$) 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 reduced a problem is, the easier it is, in principle, to solve it.
- Given a problem **P = <X, D, C>** with n variables $x_1, .., x_n$ the search space where solutions can potentially 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

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

- 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 = <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 **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:

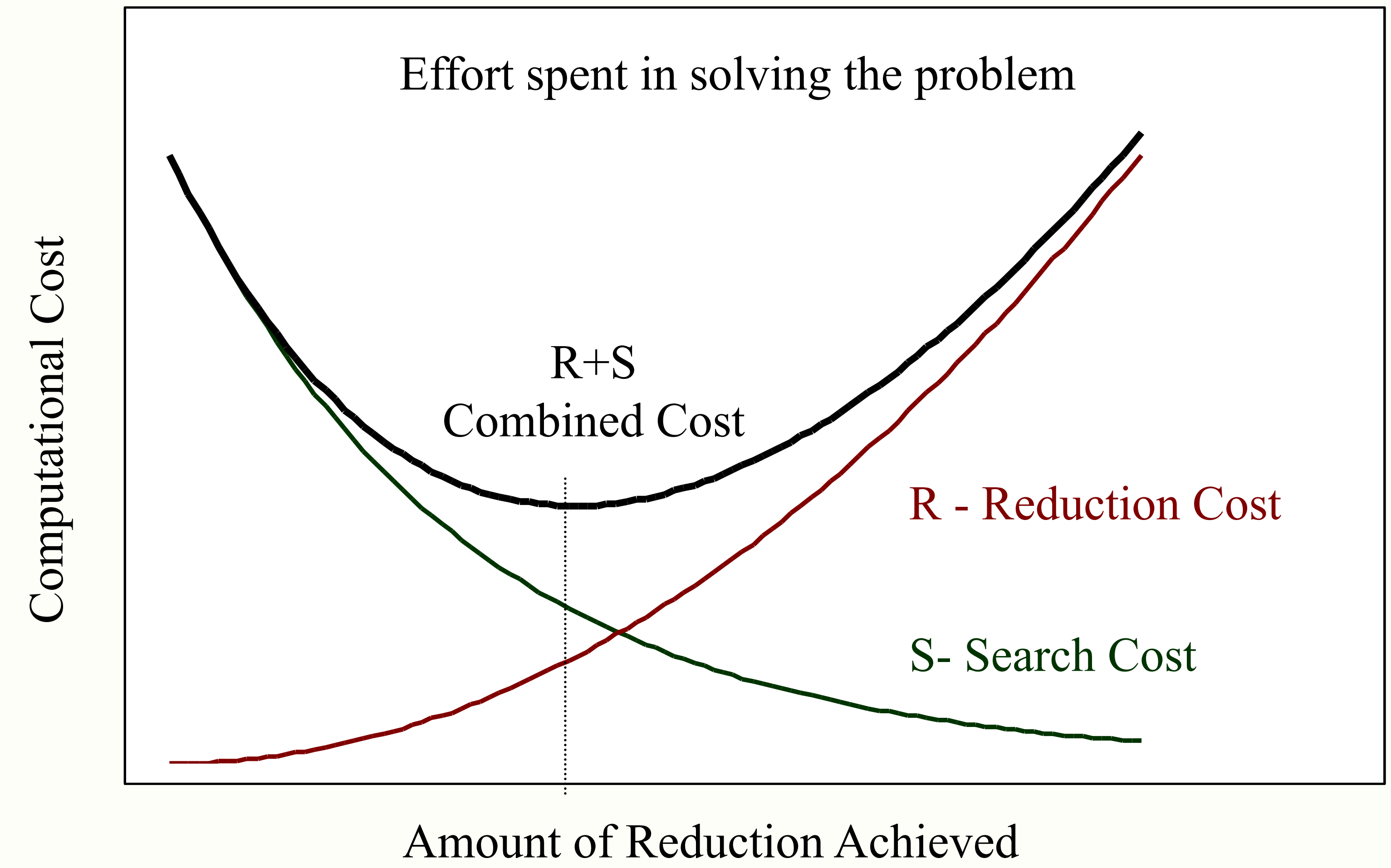
```
Declaration of Variables and Domains,
Specification of Constraints,
   propagation,  % reduction of the whole problem
% Labelling of Variables,
   label(x1),    % variable/value selection with backtraking
   propagation,  % reduction of problem {x2 ... xn}
   label(x2),
   propagation,  % reduction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

Node - Consistency

Definition (**Node Consistency**):

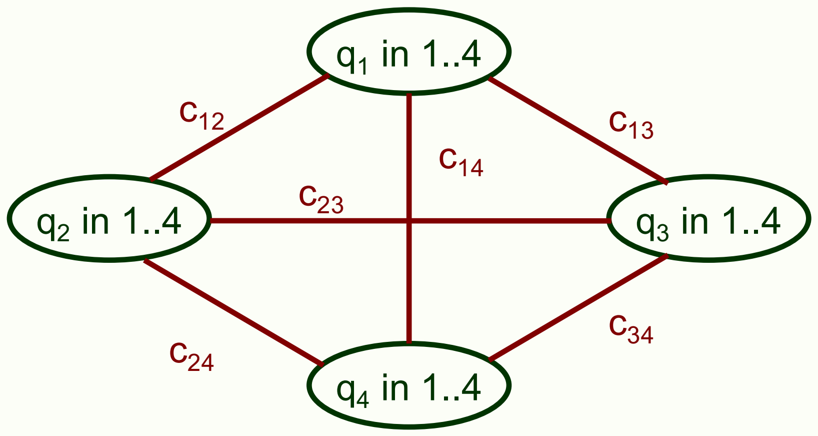
- A constraint satisfaction problem is **node-consistent** if no value in the domain of its variables violates the **unary** constraints.

- 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.

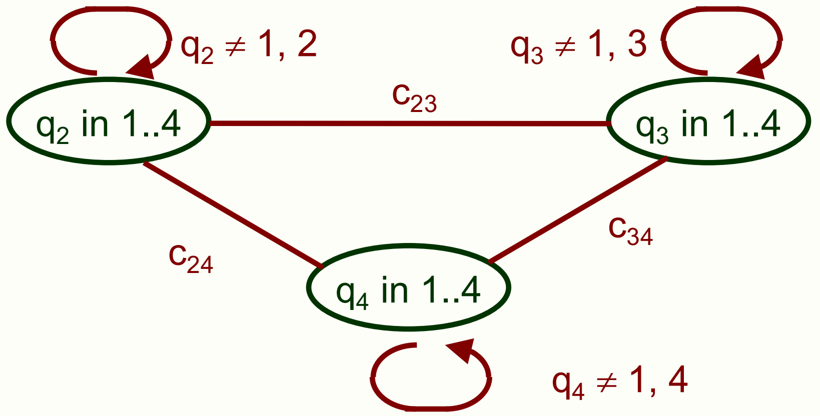
Node - Consistency

Example:

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

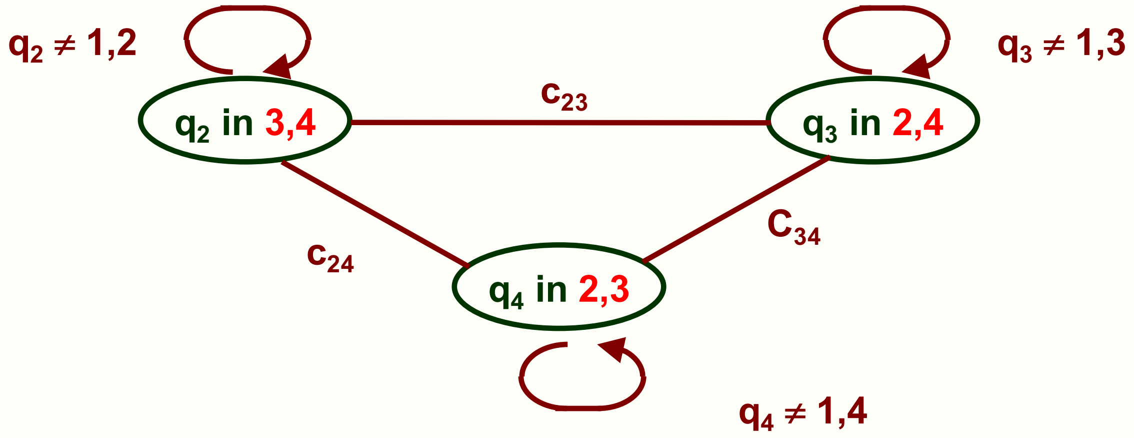


- After enumeration of variable q_1, i.e. q_1=1, constraints c_{12}, c_{13} and c_{14} become **unary** !!



Node - Consistency

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



- Maintaining node consistency thus achieves the following domain reduction.

●			
1	1		
1		1	
1			1

$q_2 \neq 1,2$

$q_3 \neq 1,3$

$q_4 \neq 1,4$

Enforcing Node-Consistency

Definition (Node Consistency):

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

Enforcing node consistency: Algorithm NC-1

- Node-consistency can be enforced by the very simple algorithm shown below:

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

Enforcing Node-Consistency

Space Complexity of NC-1: O(nd).

- Assuming **n** variables in the problem, each with **d** values in its domain, and assuming that the variable's domains are represented by extension, a space **nd** is required to keep explicitely the domains of the variables.
- Algorithm **NC-1** does not require additional space, so its space complexity is **O(nd)**.

Time Complexity of NC-1: O(nd).

- Assuming **n** variables in the problem, each with **d** values in its domain, and taking into account that each value is evaluated one single time, it is easy to conclude that algorithm NC-1 has time complexity **O(nd)**.

- The low complexity, both temporal and spatial, of algorithm NC-1, makes it suitable to be used in virtual all situations by a solver, despite the low pruning power of node-consistency.

Arc - Consistency

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

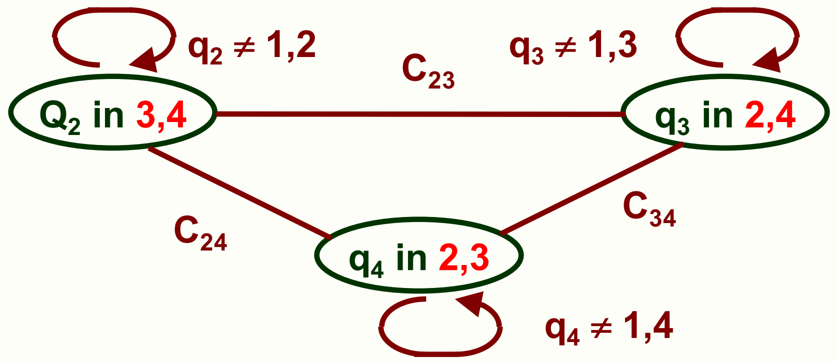
Definition (**Arc Consistency**):

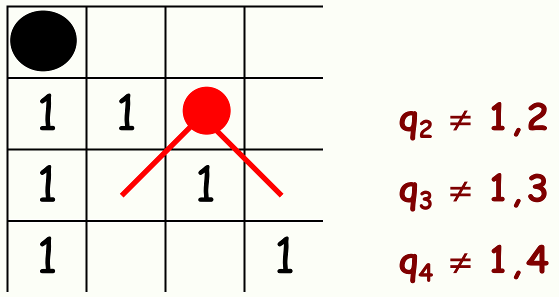
- A constraint satisfaction problem is arc-consistent if,
    - It is node-consistent; and
    - For every label x_i-v_i of every variable x_i, and for all constraints c_{ij}, defined over variables x_i and x_j, there must exist a value v_j that supports v_i, i.e. such that the compound label $\{x_i\text{-}v_i,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





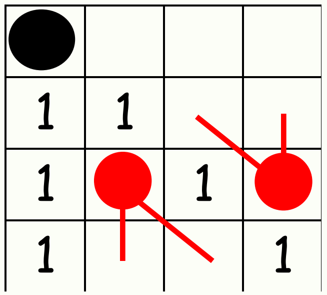
- However, label q_2-3 has **no support** in variable q_3, since neither the compound label {q_2-3 , q_3-2} nor {q_2-3 , q_3-4} will satisfy constraint C_{23}.

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

Arc - Consistency

Example (cont.):

- In fact, none (!) of the values of q3 has support in variables q2 and q4, as shown below:



$q_2 \neq 1,2$

$q_3 \neq 1,3$

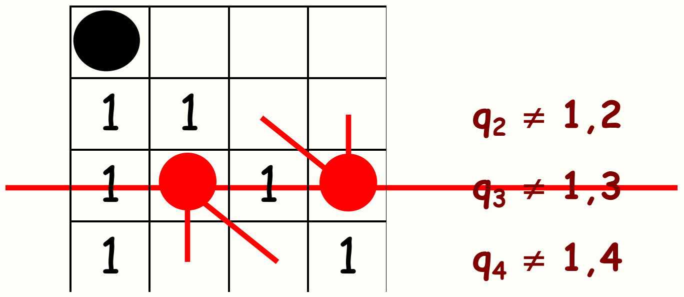
$q_4 \neq 1,4$

- 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.

Enforcing Arc-Consistency: AC-1

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 $\{x_i$-v_i, x_j-$v_j\}$ satisfies constraint c_{ij}.

Enforcing arc-consistency: Algorithm AC-1

- The following simple (and inefficient) algorithm enforces arc-consistency:

```
procedure AC-1(V, D, C);
   NC-1(V,D,C);          % node consistency
   Q = {a_ij | c_ij ∈ C ∨ c_ji ∈ C }; % see note
   repeat
     changed <- false;
     for a_ij in Q do
          changed <- changed or revise_dom(a_ij,V,D,C)
     end for
   until not change
end procedure
```

Enforcing Arc-Consistency: AC-1

Revise-Domain

- Algorithm AC-1 (and others) uses predicate **revise-domain** on some arc a_{ij}, that succeeds if some value is removed from the domain of variable x_i (a side-effect of the predicate).

```
predicate revise_dom(a_ij,V,D,C): Boolean;
   success <- false;
   for v in dom(x_i) do
      if ¬ ∃v_j in dom(x_j): satisfies({x_i-v,x_j-v_j},c_ij) then
         dom(x_i) <- dom(x_i) \ {v};
         success <- true;
      end if
   end for
   revise_dom <- success;
end predicate
```

Enforcing Arc-Consistency: AC-1

Space Complexity of AC-1: $O(ad^2)$

- AC-1 must maintain a queue **Q**, with maximum size **2a**. Hence the inherent spacial complexity of AC-1 is **O(a)**.

- To this space, one has to add the space required to represent the domains **O(nd)** and the constraints of the problem. Assuming **a** constraints and **d** values in each variable domain the space required is **$O(ad^2)$**, and the total space requirement of

$$O(nd + ad^2)$$

  which dominates **O(a)**.

- For "dense" constraint networks", **$a \approx n^2/2$**. This is then the dominant term, and the space complexity becomes

$$O(ad^2) = O(n^2d^2)$$

Enforcing Arc-Consistency: AC-1

Time Complexity of AC-1: O(nad^3)

- Assuming **n** variables in the problem, each with **d** values in its domain, and a total of **a** arcs, in the worst case, predicate revise_dom, checks **d^2** pairs of values.

- The number of arcs **a_{ij}** in queue **Q** is **2a** (2 directed arcs **a_{ij}** and **a_{ji}** are considered for each constraint **c_{ij}**). For each value removed from one domain, revise_dom is called **2a** times.

- In the worst case, only one value from one variable is removed in each cycle, and the cycle is executed **nd** times.

- Therefore, the worst-case time complexity of AC-1 is O(d^2 *2a*nd), i.e.

O(nad^3)

Enforcing Arc-Consistency: AC-3

Enforcing node consistency: Algorithm AC-3

- In AC-1, whenever a value $\mathbf{v_i}$ is removed from the domain of some $\mathbf{x_i}$, all arcs are re-examined. However, only the arcs $\mathbf{a_{ki}}$ (for $k \neq i$) should be re-examined.
- This is because the removal of $\mathbf{v_i}$ may eliminate the support from some value $\mathbf{v_k}$ of some variable $\mathbf{x_k}$ for which there is a constraint $\mathbf{c_{ik}}$ (or $\mathbf{c_{ki}}$).
- Such inefficiency of AC-1 is avoided in AC-3 below

```
procedure AC-3(V, D, C);
   NC-1(V,D,C);           % node consistency
   Q = {$a_{ij}$ | $c_{ij}$ ∈ C ∨ $c_{ji}$ ∈ C };
   while Q ≠ ∅ do
      Q = Q \ {$a_{ij}$}    % removes an element from Q
      if revise_dom($a_{ij}$,V,D,C) then    % revised $x_i$
         Q = Q ∪ {$a_{ki}$ | ($c_{ik}$ ∈ C ∨ $c_{ki}$ ∈ C )∧  k ≠ i}
      end if
   end while
end procedure
```

Enforcing Arc-Consistency: AC-3

Space Complexity of AC-3: $O(ad^2)$

- AC-3 has the same requirements than AC-1, and the same worst-case space complexity of $\mathbf{O(ad^2)} \approx \mathbf{O(n^2d^2)}$, due to the representation of constraints by extension.

Time Complexity of AC-3: $O(ad^3)$

- Each arc a_{ki} is only added to **Q** when some value $\mathbf{v_i}$ is removed from the domain of $\mathbf{x_i}$.
- In total, each of the **2a** arcs may be added to **Q** (and removed from **Q**) **d** times.
- Every time that an arc is removed, predicate revise_dom is called, to check at most $\mathbf{d^2}$ pairs of values.
- All things considered, and in contrast with AC-1, with temporal complexity $\mathbf{O(nad^3)}$, the time complexity of AC-3, in the worst case, is $\mathbf{O(2ad * d^2)}$, i.e.

$$O(ad^3)$$

Enforcing Arc-Consistency: AC-4

Counting Supports: AC-4

- Every time a value $\mathbf{v_i}$ is removed from the domain of some variable $\mathbf{x_i}$, all arcs $\mathbf{a_{ki}}$ ($k \neq i$) leading to that variable are re-examined.
- Nevertheless, only some of these arcs should be examined.
- Although the removal of vi may eliminate one support for some value $\mathbf{v_k}$ of another variable $\mathbf{x_k}$ (given constraint c_{ki}), other values in the domain of $\mathbf{x_i}$ may support the pair $\mathbf{x_k}$-$\mathbf{v_k}$!

- This idea is exploited in algorithm AC-4, that uses a number of new data-structures to **count** supporting values, which contrary to AC-3, with time complexity of $O(ad^2)$, achieves a time-complexity of

$$\mathbf{O(ad^2)}$$

- This is in fact an optimal **assymptotical worst-case complexity**, since checking all the pairs of values in all the binary constraints require $\mathbf{ad^2}$ operations.

Enforcing Arc-Consistency: AC-4

Detecting last Supports: AC-6

- Whereas AC-4 maintains a number of counters to check whether there are support for variables in the domain, the same effect can be achieved by simply maintaining one value that witnesses this support.
- Every time this witness is removed, a next witness is sought.
- Hence there is no need to maintain expensive counters, and the goal of maintaining arc-consistency can be made more efficiently.

- This is the idea exploited in algorithm AC-6, that uses “lighter” data-structures to detect **next** supporting values which, like AC-4, has a time complexity of

$$O(ad^2)$$

- Again this was already seen as the optimal assymptotical worst-case complexity, and AC-6 could not beat it.
- However …

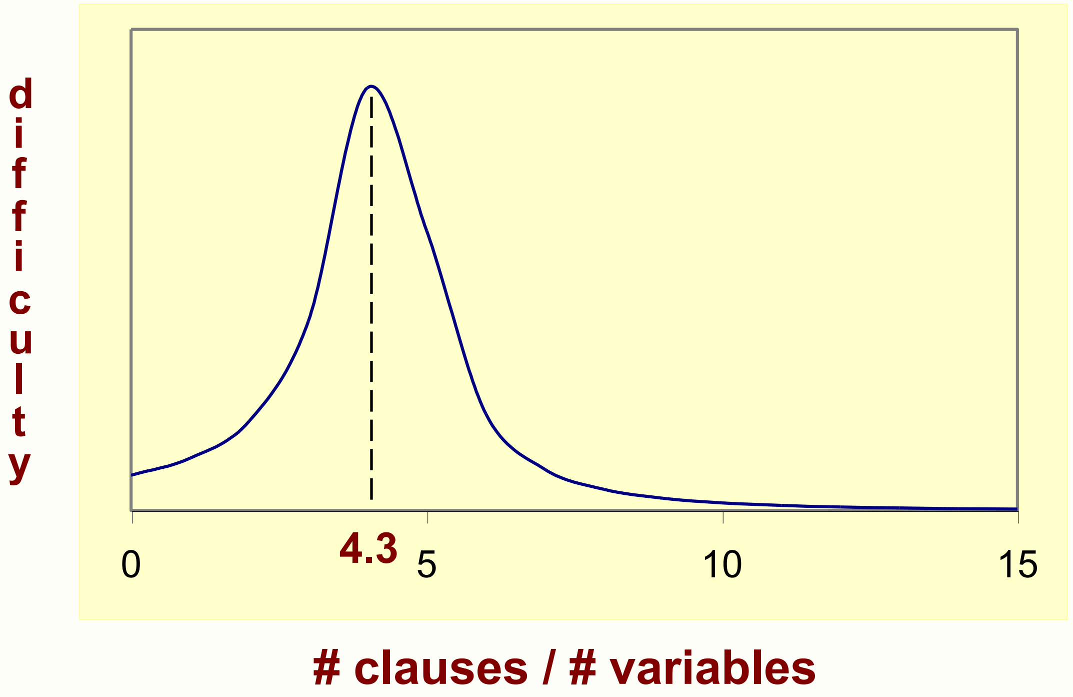
Assessing Typical Complexity

Typical complexity of AC-x algorithms

- The **worst-case** time complexity that can be inferred from the algorithms that maintain arc-consistency do not give a precise idea of their average behaviour in typical situations. For such study, either one tests the algorithms in:
  - A set of "benchmarks", i.e. problems that are supposedly representative of everyday situations (e.g. N-queens); or
  - Randomly generated instances parameterised by
    - their **size** (number of variables and cardinality of the domains) ; and
    - their **difficulty** measured by
      - density of the constraint network - % existing/ possible constraints; and
      - tightness of the constraints - % of allowed / all tuples.
- The study of these issues has led to the conclusion that constraint satisfaction problems often exhibit a phase transition, which should be taken into account in the study of the algorithms.

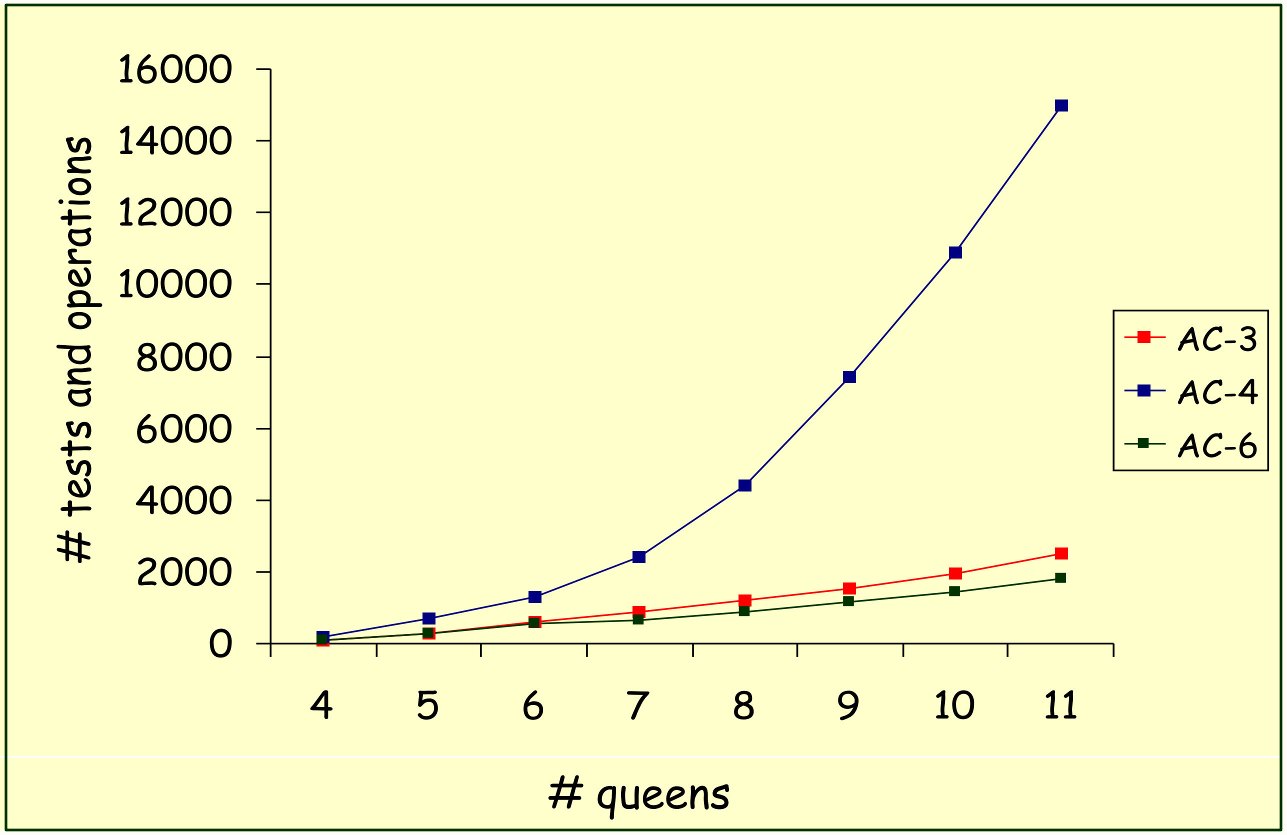
Assessing Typical Complexity: Phase Transition

- This phase transition typically contains the most difficult instances of the problem, and separates the instances that are trivially satisfied from those that are trivially insatisfiable.
- For example, in **SAT** problems, it has been found that the phase transition occurs when the ratio of clauses to variables is around 4.3.



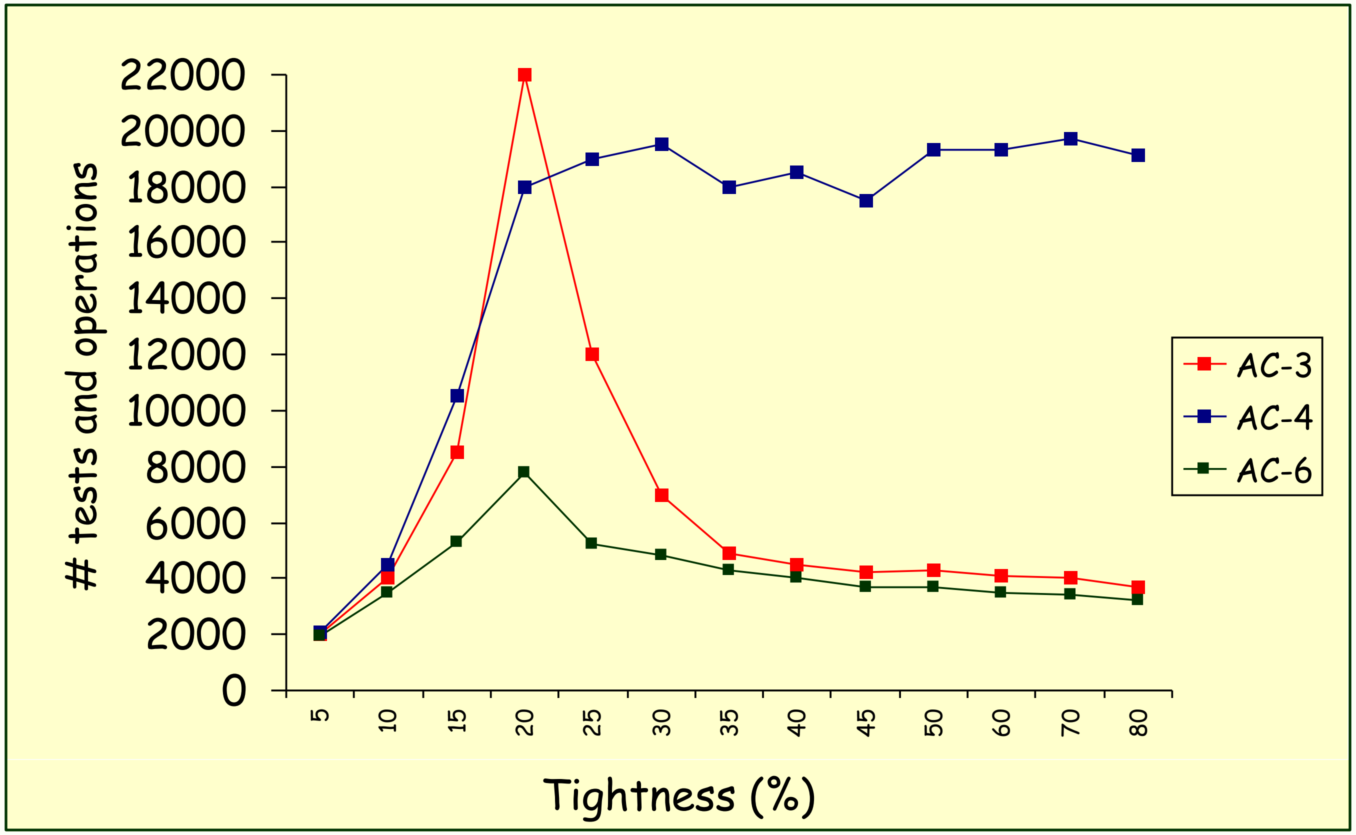
Assessing Typical Complexity

- **Typical Complexity** of algorithms AC-3, AC-4 e AC-6
  - (N-queens)



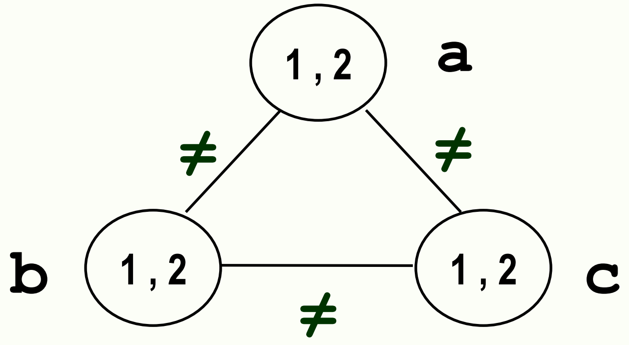
Assessing Typical Complexity

Typical Complexity of algorithms AC-3, AC-4 e AC-6
(randomly generated problems)
n = 12 variables, d= 16 values, density = 50%



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>} $\rightarrow$ c ≠ 1, 2 ; {<a-1>, <b-2>} $\rightarrow$ c ≠1, 2

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