Constraint Programming

- An overview

  - Examples, Satisfaction vs. Optimization
  - Different Domains
  - Constraint Propagation –
    - » Kinds of Consistencies
  - Global Constraints
  - Heuristics
  - Symmetries

Constraint Solving Problems

In general, constraint satisfaction problems (CSPs) can be regarded as assignment problems. Solving them correspond to assigning values to the variables within their domains so as to satisfy the intended constraints.

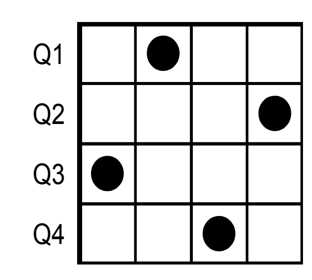
- Variables:
  - Objects / Properties of objects
- Domain:
  - Integer, enumerated or Real
  - Booleans for decisions
- Constraints:
  - Compatibility (Equality, Difference, No-attack, Arithmetic Relations)

Some examples may help to illustrate this class of problems

Constraint Problems: Examples

- Assignment (Graph Colouring, Latin Squares, Magic Squares, ...)
- Scheduling (timetabling, job-shop, ...)
- Traveling Salesperson
- Knapsack
- Filling
- Model-checking
- etc...

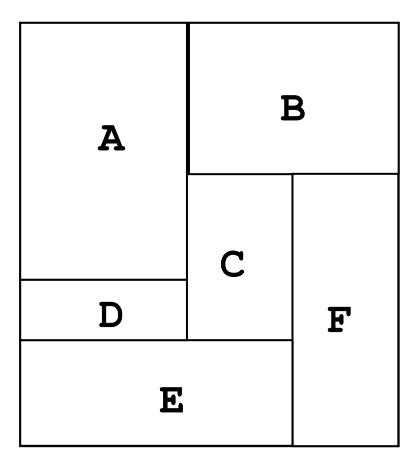
Assignment (2)



N-queens (Finite Domains):

`Assign Values to Q1,..., Qn ∈ {1,.., n}`

`s.t.` $\forall_{i \neq j}$ `noattack (Qi, Qj)`



Graph Colouring (Finite Domains)

`Assign values to A, .., F,`

`s.t. A, B, .., F ∈ {red, blue, green}`
`A ≠ B, A ≠ C, A ≠ D,`
`B ≠ C, B ≠ F, C ≠ D, C ≠ E, C ≠ F`
`D ≠ E, E ≠ F`

Assignment (3)

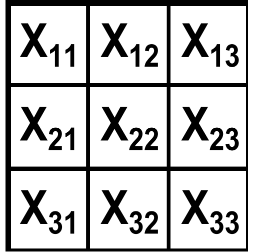
X_{11}	X_{12}	X_{13}
X_{21}	X_{22}	X_{23}
X_{31}	X_{32}	X_{33}

Latin Squares (similar to **Sudoku**):

Assign Values to X11,..., X33 $\in$ {1,.., 3}

s.t. $\forall_i \ \forall_{j \neq k} \ X_{ij} \neq X_{ik}$ % same row

$\forall_j \ \forall_{i \neq k} \ X_{ij} \neq X_{kj}$ % same column



Magic Squares:

Assign Values to X11,..., X33 $\in$ {1,..,9}

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

$\forall i \neq j \ \Sigma_k \ X_{ik} = \Sigma_k \ X_{jk} = M$ % same cols sum

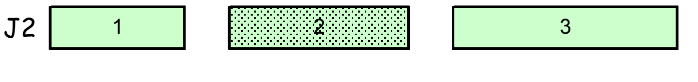
$\Sigma_k \ X_{kk} = \Sigma_k \ X_{k,n-k+1} = M$ % diagonals

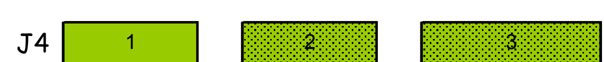
$\forall i \neq k \vee j \neq l \ Xij \neq Xkl$ % all different

Mixed: Assignment and Scheduling

Goal (Example): Assign values to variables

- Variables:
  - Start Times, Durations, Resources used
- Domain:
  - Integers (typicaly) or Rationals/Reals
- Constraints:
  - Compatibility (Disjunctive, Difference, Arithmetic Relations)





J1 1 2 3
J2 1 2 3
J3 1 2 3
J4 1 2 3

Job-Shop

Assign values to $S_{ij} \in \{1,..,n\}$ % time slots
and to $M_{ij} \in \{1,..,m\}$ % machines available

% precedence within job

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

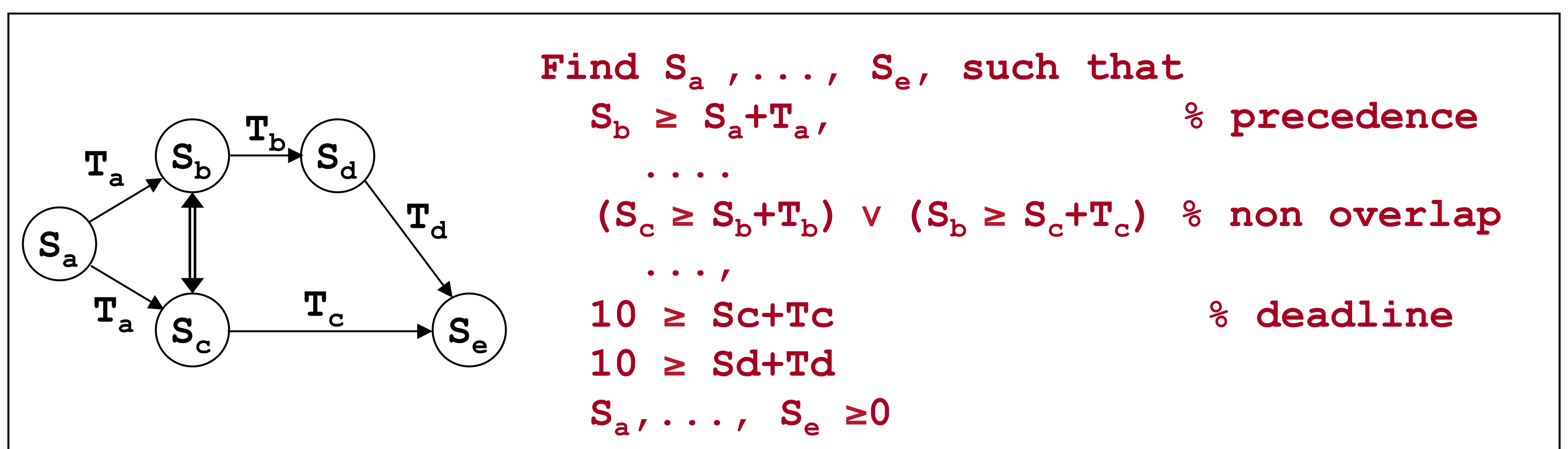
% either no-overlap or different machines

$\forall_{i,j,k,l} \ (M_{ij} \neq M_{kl}) \lor (S_{ij} + D_{ij} \leq S_{kl}) \lor (S_{kl} + D_{kl} \leq S_{ij})$

Schedulling

Goal (Example): Assign timing/precedence to tasks

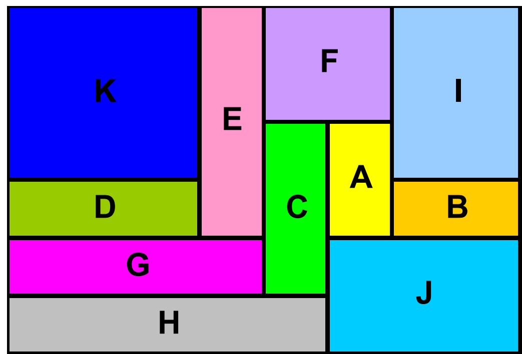
- Variables:
  - Start Timing of Tasks, Duration of Tasks
- Domain:
  - Rational/Reals or Integers
- Constraints:
  - Precedence Constraints, Non-overlapping constraints, Deadlines, etc...



Filling and Containment

Goal (Example): Assign values to variables

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



Fiiling

Assign values to $X_i \in \{1,..,Xmax\}$ % X-dimension

$Y_i \in \{1,..,Ymax\}$ % Y-dimension

% no-overlapping rectangles

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

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

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

$(Y_j+Lx_j \leq X_i)$ % I in back of J

Constraint Satisfaction Problems

- Formally a constraint satisfaction problem (CSP) can be regarded as a tuple <X, D, C>, where
  - $X = \{ X_1, \ldots , X_n\}$ is a set of variables
  - $D = \{ D_1, \ldots , D_n\}$ is a set of domains (for the corresponding variables)
  - $C = \{ C_1, \ldots , C_m\}$ is a set of constraints (on the variables)

- Solving a constraint problem consists of determining values $x_i \in D_i$ for each variable X_i, satisfying all the constraints C.

- Intuitively, a constraint C_i is a limitation on the values of its variables.

- More formally, a constraint C_i (with arity k) over variables $X_{i1}, \ldots, X_{ik}$ ranging over domains $D_{i1}, \ldots, D_{ik}$ is a subset of the cartesian cartesian $D_{j1} \times \ldots \times D_{jk}$.

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

Constraints and Optimisation Problems

- In many cases, one is interested not only in satisfying some set of constraints but also in findind among all solutions those that optimise a certain objective function (minimising a cost or maximising some positive feature).

- Formally a constraint (satisfaction and) optimisation problem (CSOP) can be regarded as a tuple <X, D, C, F>, where
  - $X = \{ X_1, \dots , X_n\}$ is a set of variables
  - $D = \{ D_1, \dots , D_n\}$ is a set of domains (for the corresponding variables)
  - $C = \{ C_1, \dots , C_m\}$ is a set of constraints (on the variables)
  - F is a function on the variables

- Solving a constraint satisfaction and optimsation problem consists of determining values $x_i \in D_i$ for each variable X_i, satisfying all the constraints C and that optimise the objective function.

Issues to be Addressed

- **Modelling**
  - What are the best models (variables, domains, constraints)
- **Complexity Issues**
  - CSPs are typically NP-complete / NP-Hard
- **Search Methods**
  - Backtrack Search vs. Local Search
  - Improved Backtracking through Constraint Propagation
    - Consistency Types
    - Simple vs. Global Constraints
    - Redundant Constraints
  - Heuristics
  - Simmetry Breaking
  - Randomisation and Restarts

Modelling with Different Domains

- In many cases,many alternative formulations exist with different types of domains that, although equivalent, may lead to important differences in execution.

Example: **Set Partition** – Fiind a partition of a set S in three (disjoint) sets of three elements such that the sum of all their elements are the same.

Modelling with **set variables (**values of variables are **sets)**

```
Given S = { 1, 3, 7, 8, 9, 10, 14, 22, 25},
Find sets S1, S2 and s3 such that
       S1 ∪ S2 ∪ s3 = S;
       #S1 = #S2 = #S3 = 3;
       S1 ∩ S2 = S1 ∩ S2 = S1 ∩ S2 = ∅;
       sum(S1) = sum(S2) = sum(S3)
```

Modelling with Different Domains

Example: **Set Partition** – Fiind a partition of a set S in three (disjoint) sets of three elements such that the sum of all their elements are the same.

Modelling with **finite domain** variables (values of variables are finitely enumerated, e.g integers in limited ranges)

```
Given D = [ 1, 3, 7, 8, 9, 10, 14, 22, 25],
Find X1 = [X11,X12,X13], ..., X3 = [X31,X32,X33]
Such that
      X11, X12, .. , X33 have domain D;
      X11 ≠ X12, ..., X32 ≠ X33 % all_different;
      sum(S1) = sum(S2) = sum(S3)
```

Modelling with Different Domains

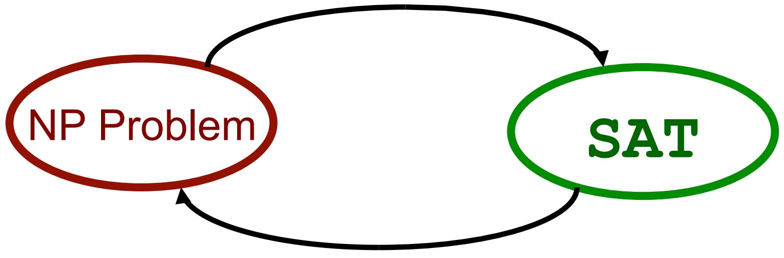
Example: **Set Partition** – Fiind a partition of a set S in three (disjoint) sets of three elements such that the sum of all their elements are the same.

Modelling with **boolean domain** variables (values of variables are booleans)

```
Given S = [ 1, 3, 7, 8, 9, 10, 14, 22, 25],
Find X1 = [X11,..,X19], ..., X3 = [X31,...,X39]
Such that
    X11, X12, .. , X39 have domain 0/1;
    sum([X11,...,X19]) = 3; ..., sum([X31, .., X39]) = 3;
    sum([X11,X21,X31]) = 1; ..., sum([X19,X29,X39] = 1;
    sumproduct(X1,S) = sumproduct(X2,S) = sumproduct(X3,S).
```

Complexity of Decision Problems

- All the problems presented are decision making problems in that a decision has to be made regarding the value to assign to each variable.
- No trivial decision making problems are untractable, i.e. they lie in the class of NP problems.
- Formally, these are the class of problems that can be solved in polinomial time by a non-deterministic machine, i.e. one that "guesses the right answer".
- For example, in the graph colouring problem (n nodes, k colours), if one has to assign colours to n nodes, a non-deterministic machine could guess a solution in O(n) steps.
- Formally, NP-complete problems are a class of problems that may be converted in polinomial time on other NP-complete problems (SAT, in particular).



Complexity of Decision Problems

- No one has already found a polinomial algorithm to solve SAT (or any other NP problem), and hence the conjecture P ≠ NP (perhaps one of the most challenging open problems in computer science) is regarded as true.

- Hence, with real machines and non trivial problems, one has to guess the adequate values for the variables and make mistakes. In the worst case, one has to test $O(k^n)$ potential solutions.

- Just to have an idea of the complexity, the table below shows the time needed to check kn solutions, assuming one solution is examined in 1 μsec (times in secs).

d^n		n					
		10	20	30	40	50	60
d	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

Complexity of Decision Problems

- Still, constraint solving problems are NP-complete problems (as SAT is). This means that one may check in polinomial time whether a potential solution satisfies all the constraints.
- More important: with an appropriate search strategy, many instances of NP-complete problems can be solved in quite acceptable times.
- Hence, search plays a fundamental role in solving this kind of problems. Adequate search methods and appropriate heuristics can often solve large instances of these problems in very acceptable time.
- Optimisation problems are typically NP-Hard problems in that solving them is at least as difficult as solving the corresponding decision problem.
- Being harder than the decision problems, optimisation problems also require adequate search strategies, if larger instances are to be solved.

Search Strategies

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

Complete Backtrack Search Methods:

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

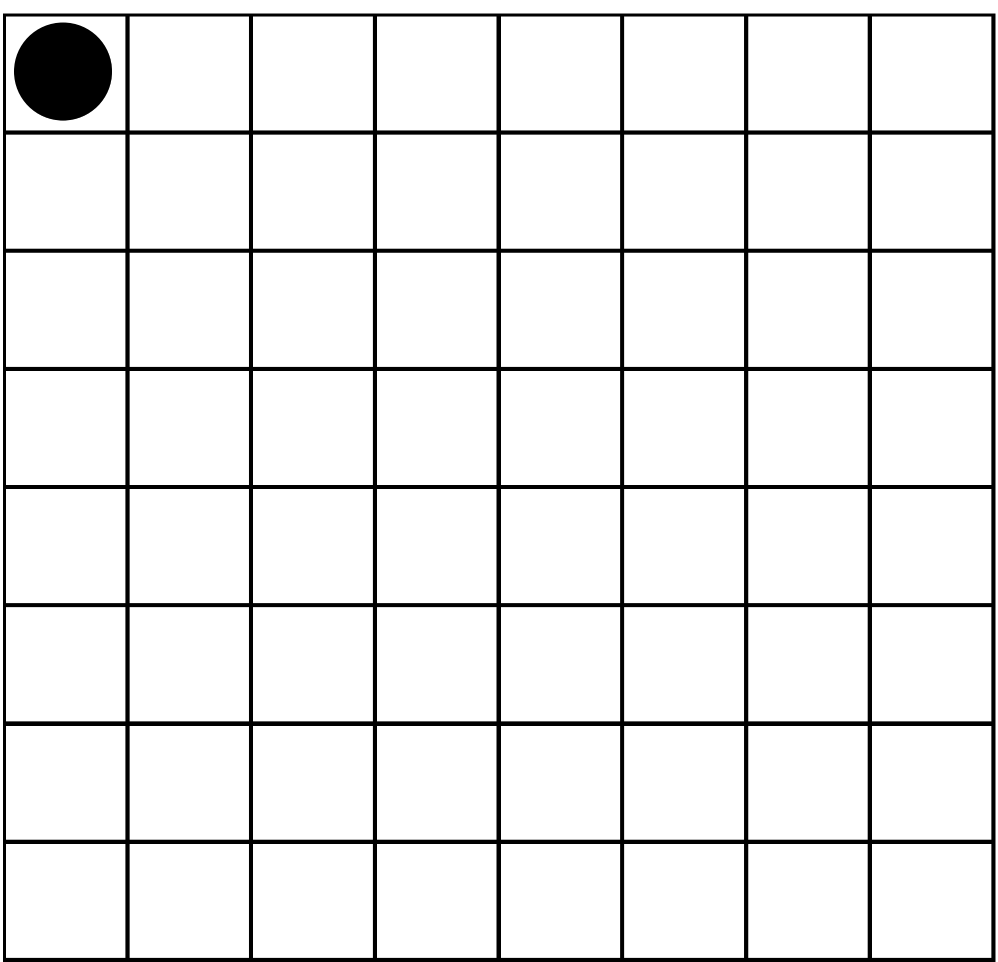
Incomplete Local Search Methods:

- Complete Solutions are incrementally **repaired**, coformed, by changing the values assigned to some of the variables until a solution is found;
- These local serach methods are not guaranteed to avoid revisiting the same solutions time and again and are therefore incomplete.
- They are often more efficient to find very good solutions.

Search Methods – Pure Backtracking

- The same specification can lead to different search strategies.
- 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.
- This is a typical **generate and test** procedure
  - First, values are generated
  - Second , 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



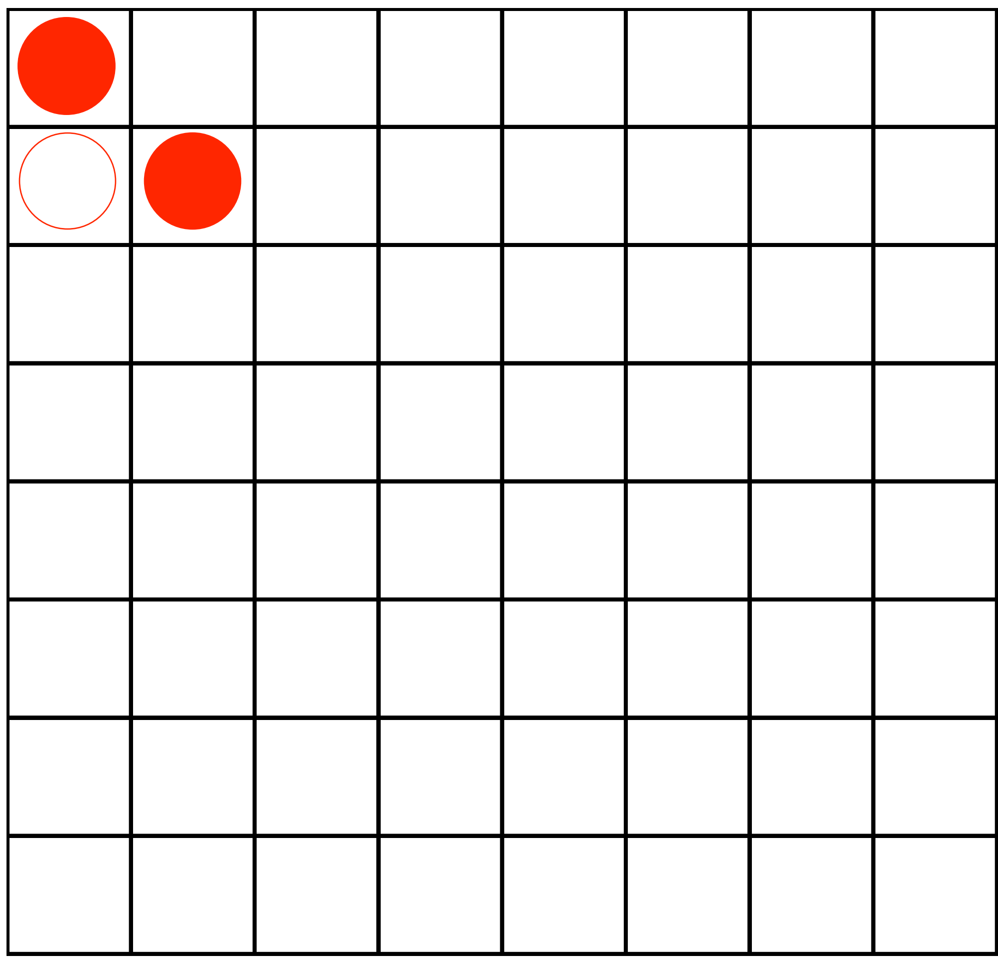
Tests 0 **Backtracks 0**

Backtracking

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

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

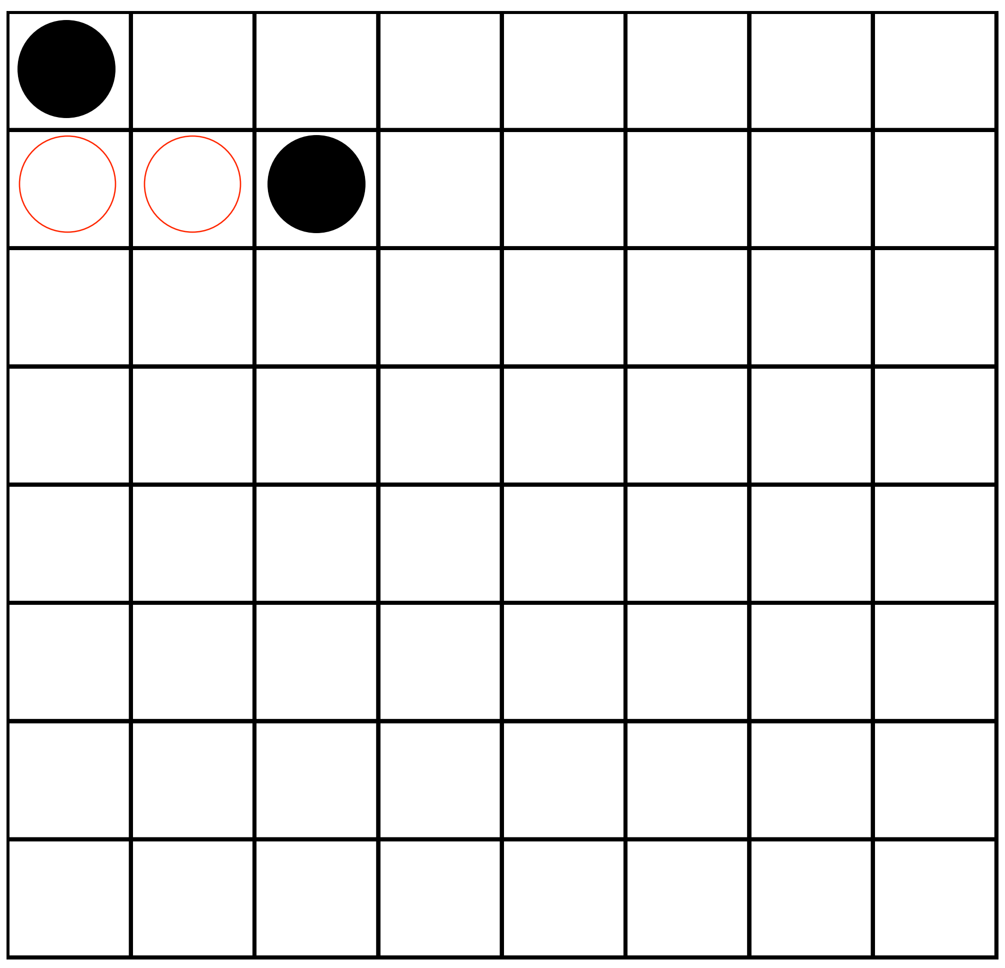
Backtracking



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

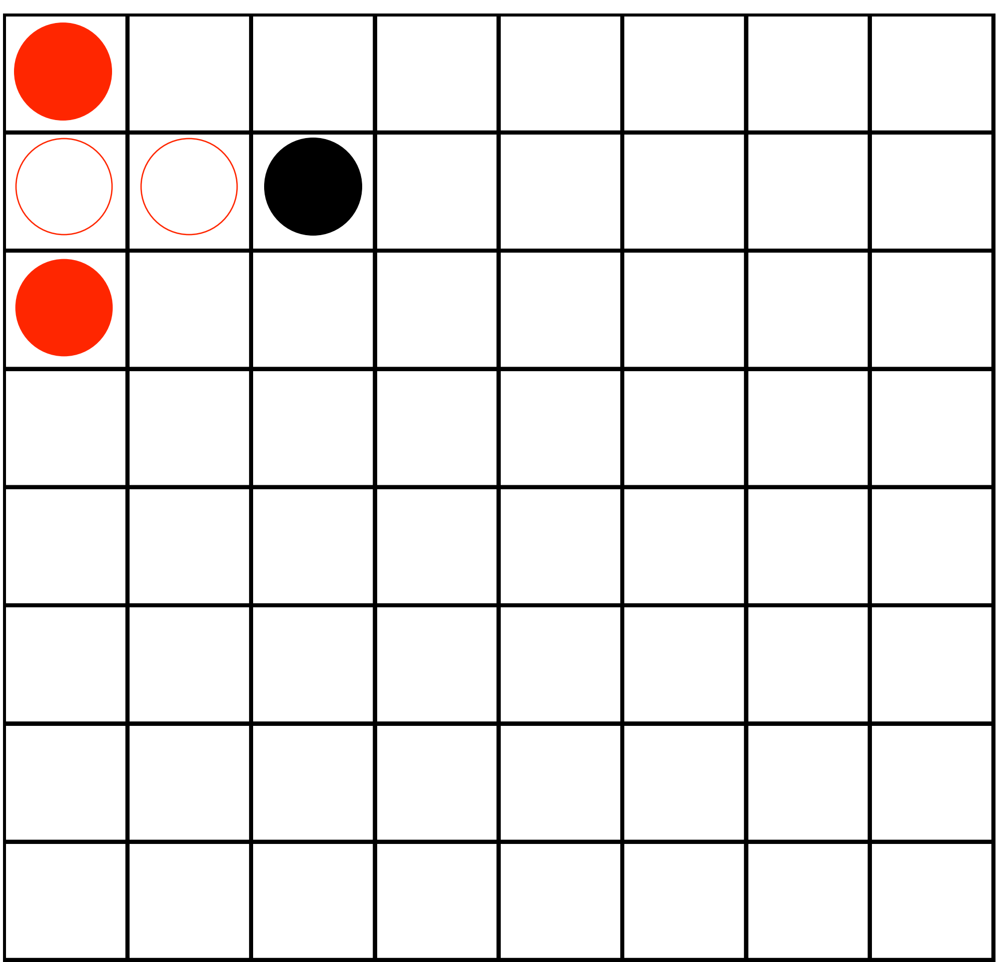
Backtracking

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



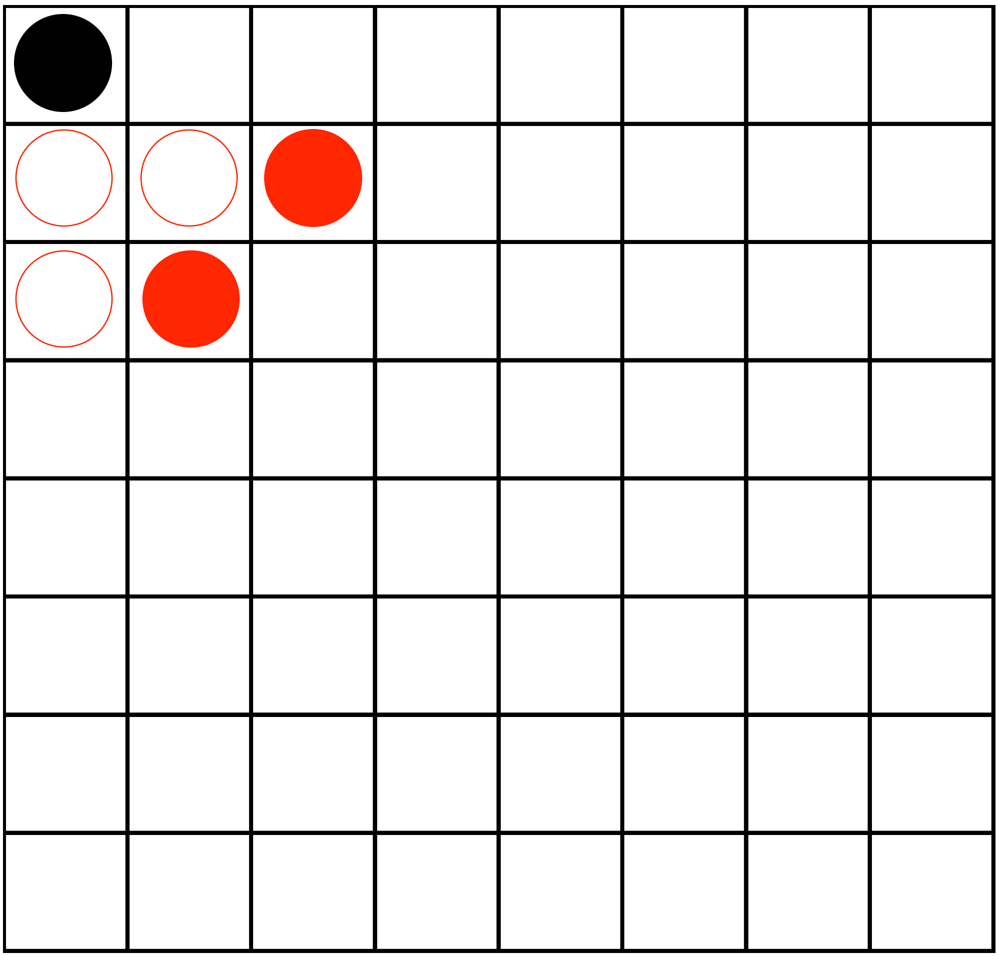
Tests 2 +1 = 3 **Backtracks 0**

Backtracking



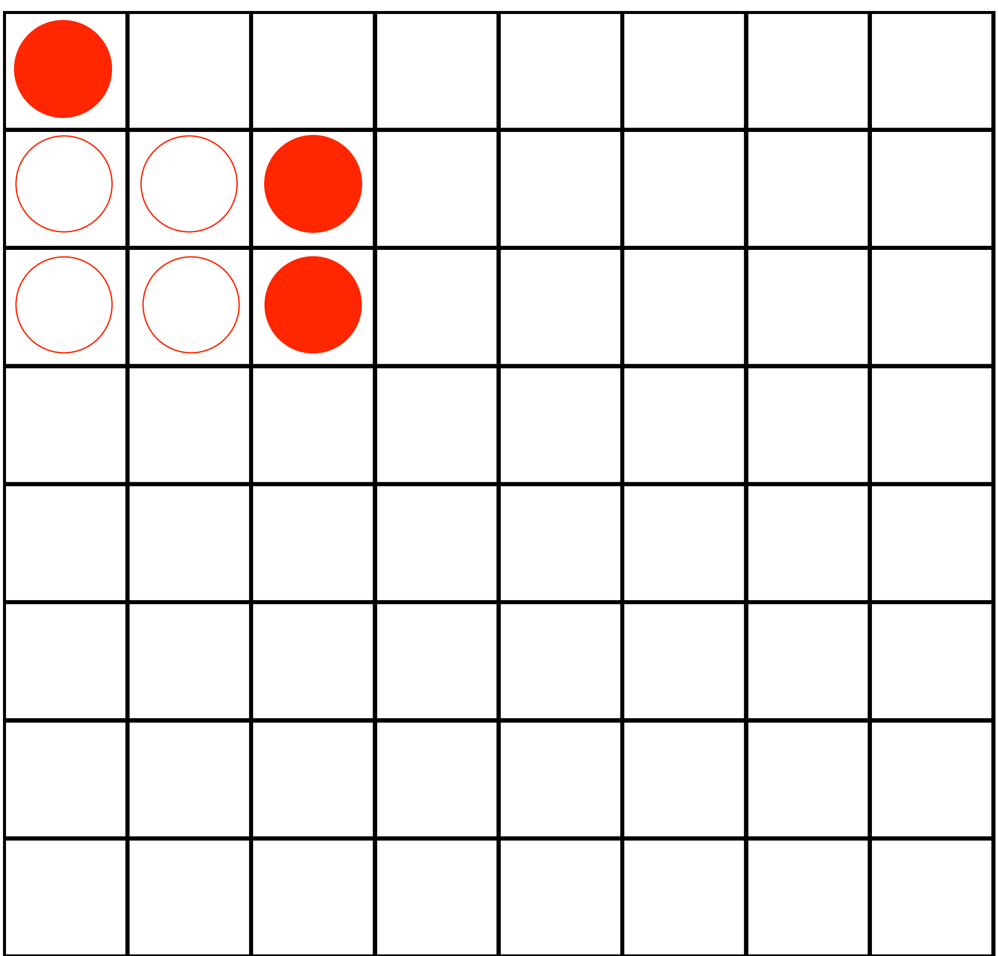
Tests 3 +1 = 4 **Backtracks 0**

Backtracking



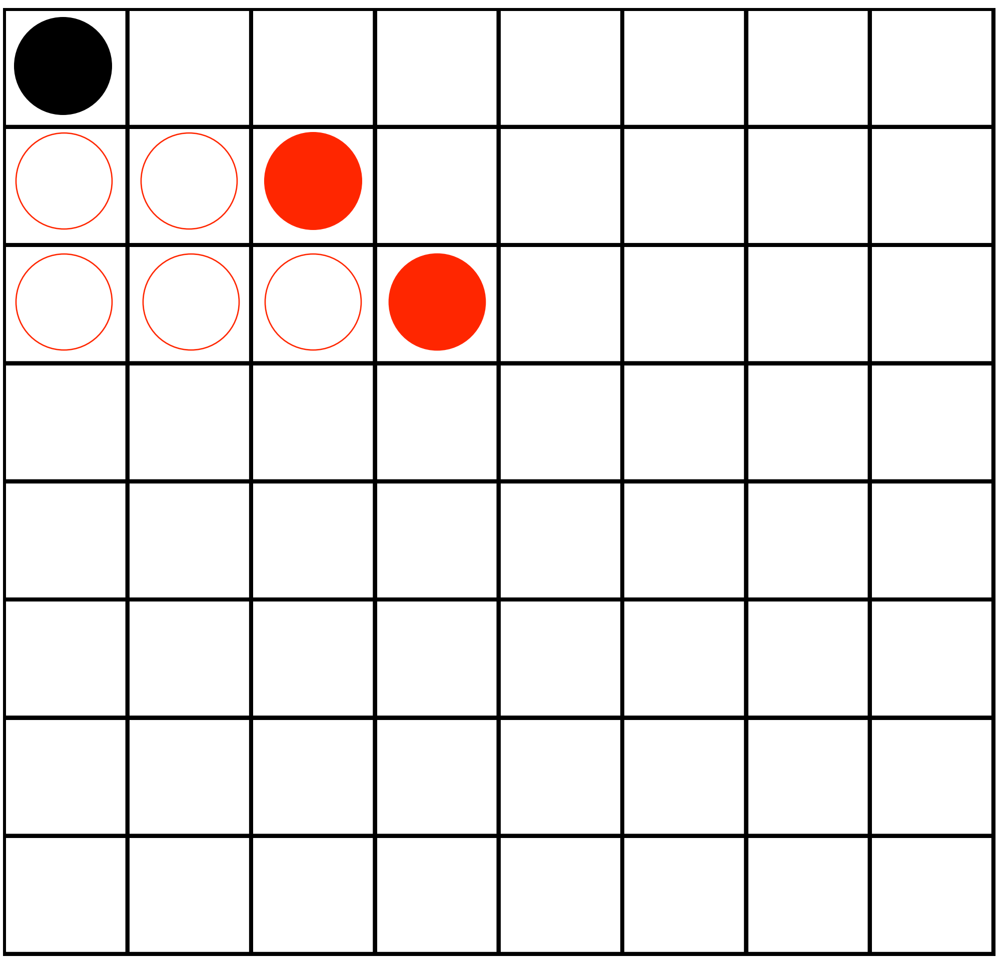
Tests 4 +2 = 6 **Backtracks 0**

Backtracking



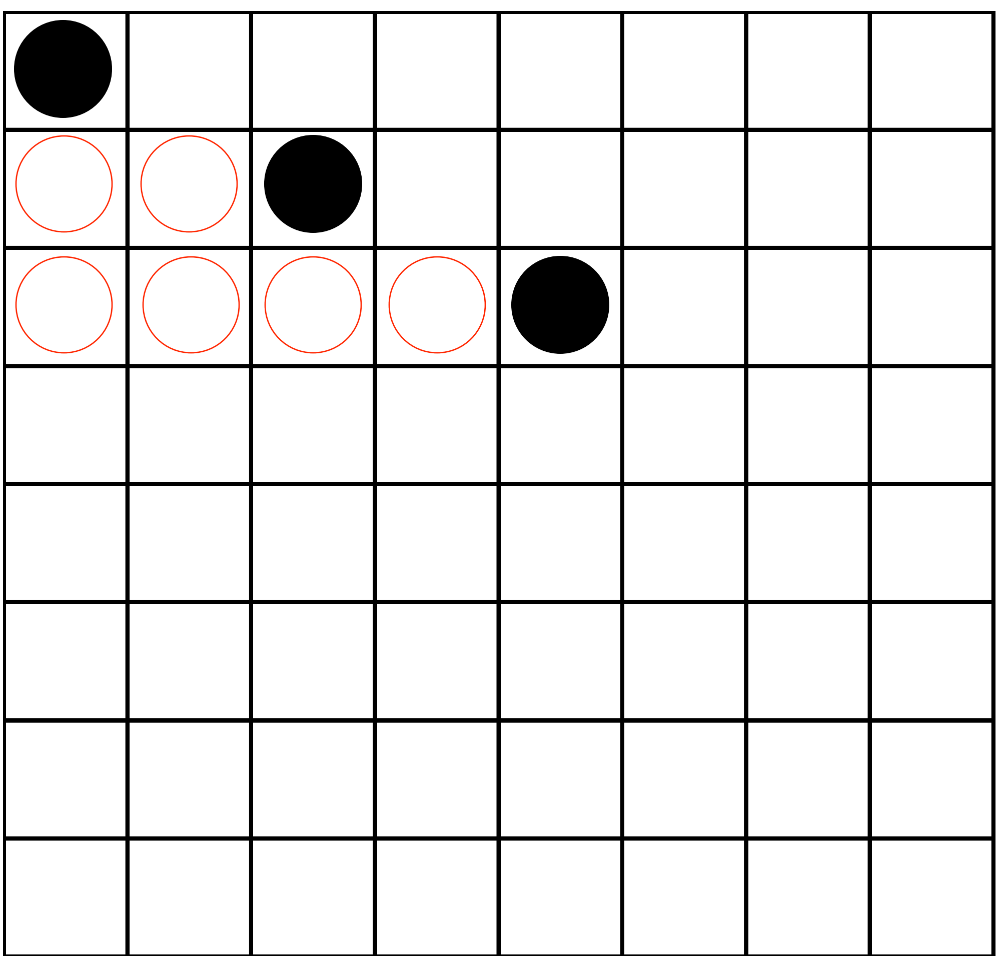
Tests 6 + 1 = 7 **Backtracks 0**

Backtracking



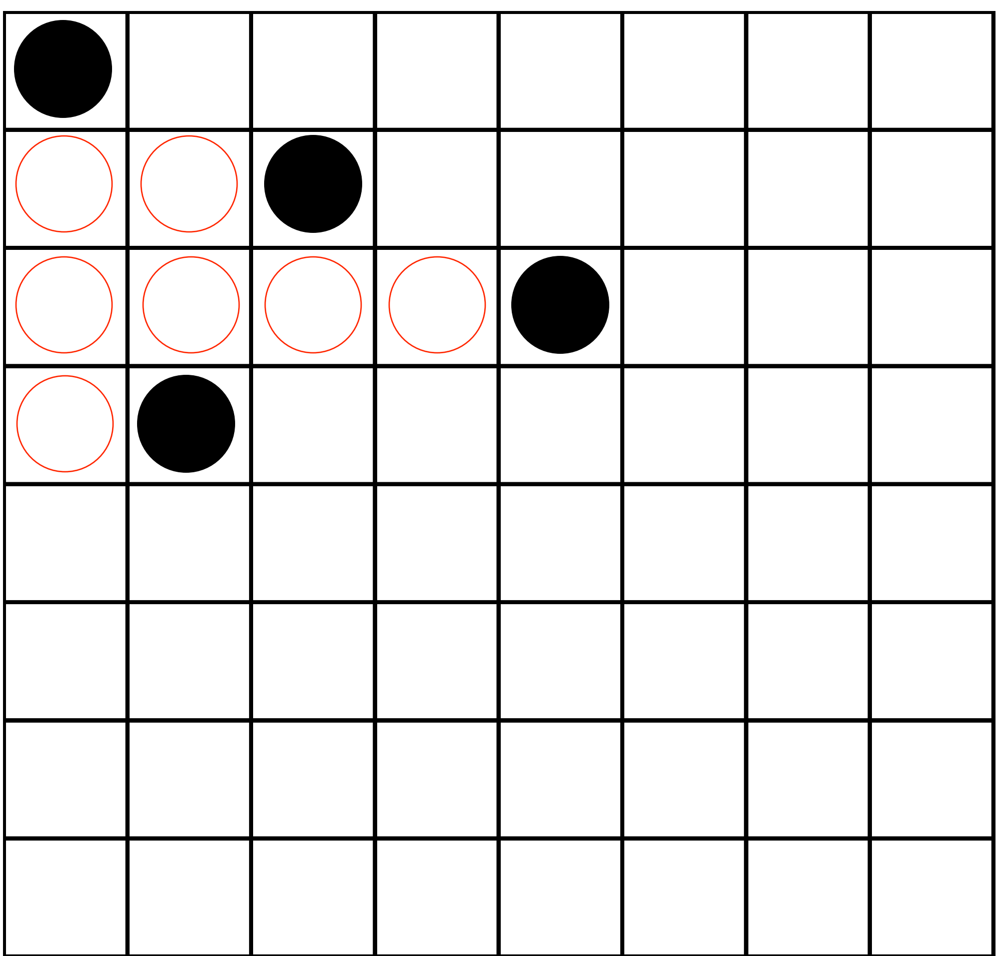
Tests 7 + 2 = 9 **Backtracks 0**

Backtracking



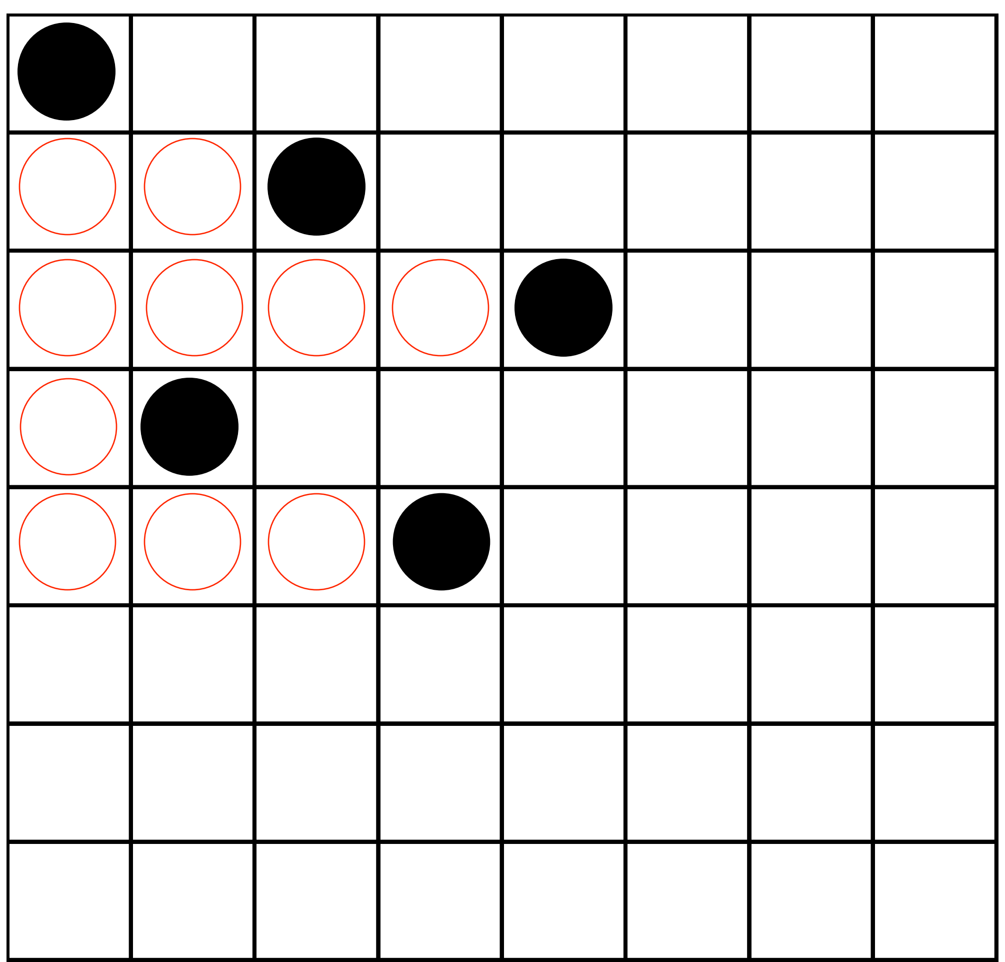
Tests 9 + 2 = 11 **Backtracks 0**

Backtracking



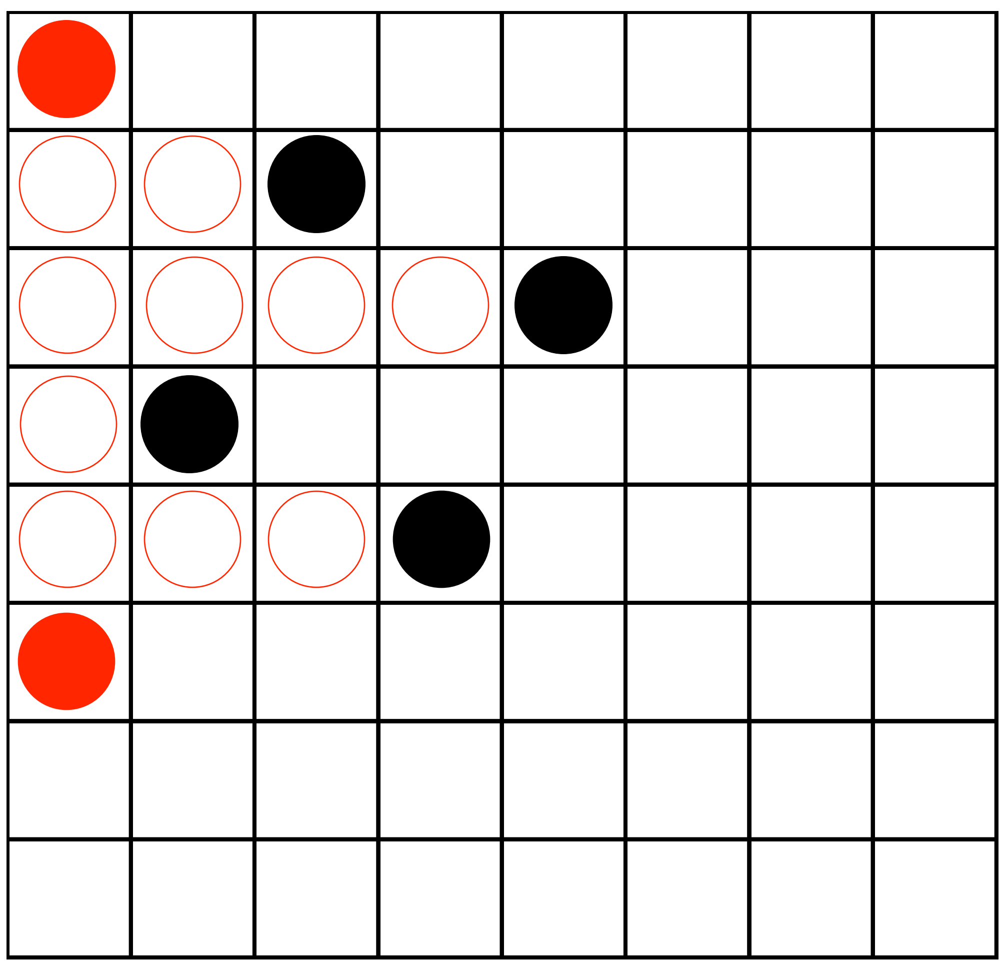
Tests 11 + 1 + 3 = 15 **Backtracks 0**

Backtracking



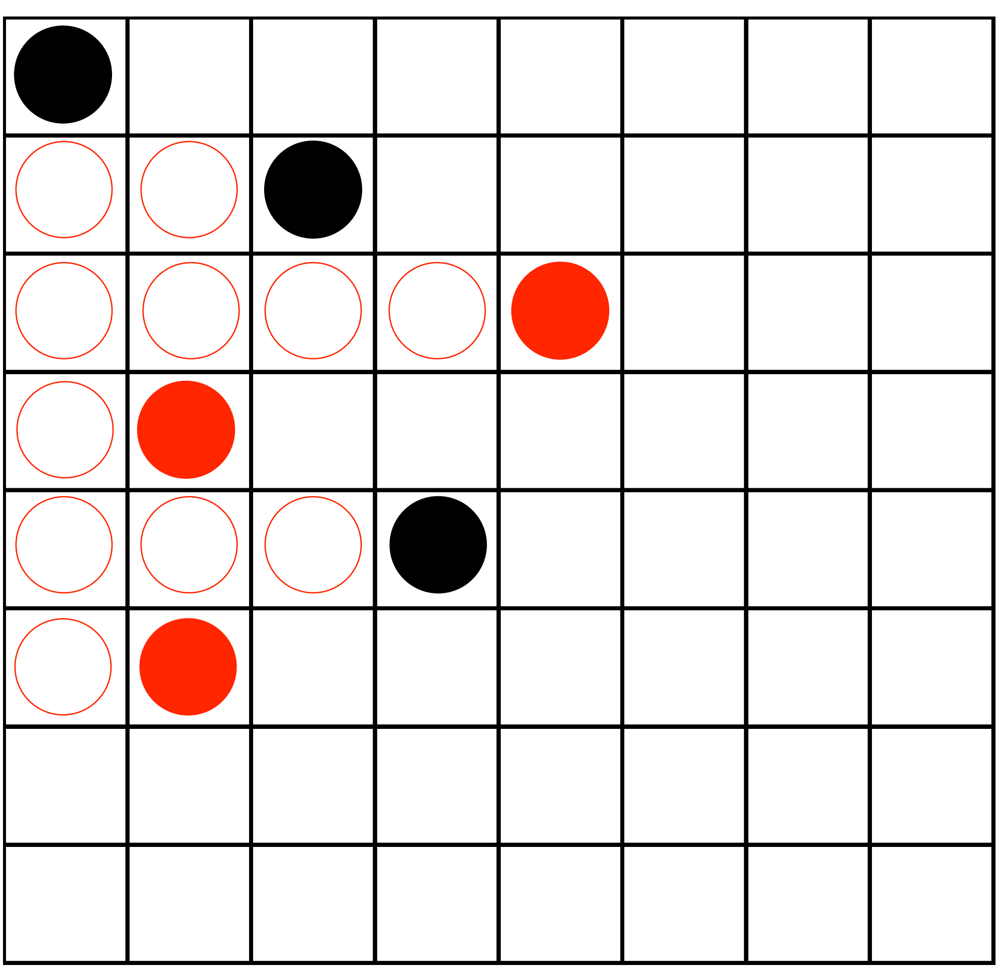
Tests 15+1+4+2+4 = 26 Backtracks 0

Backtracking



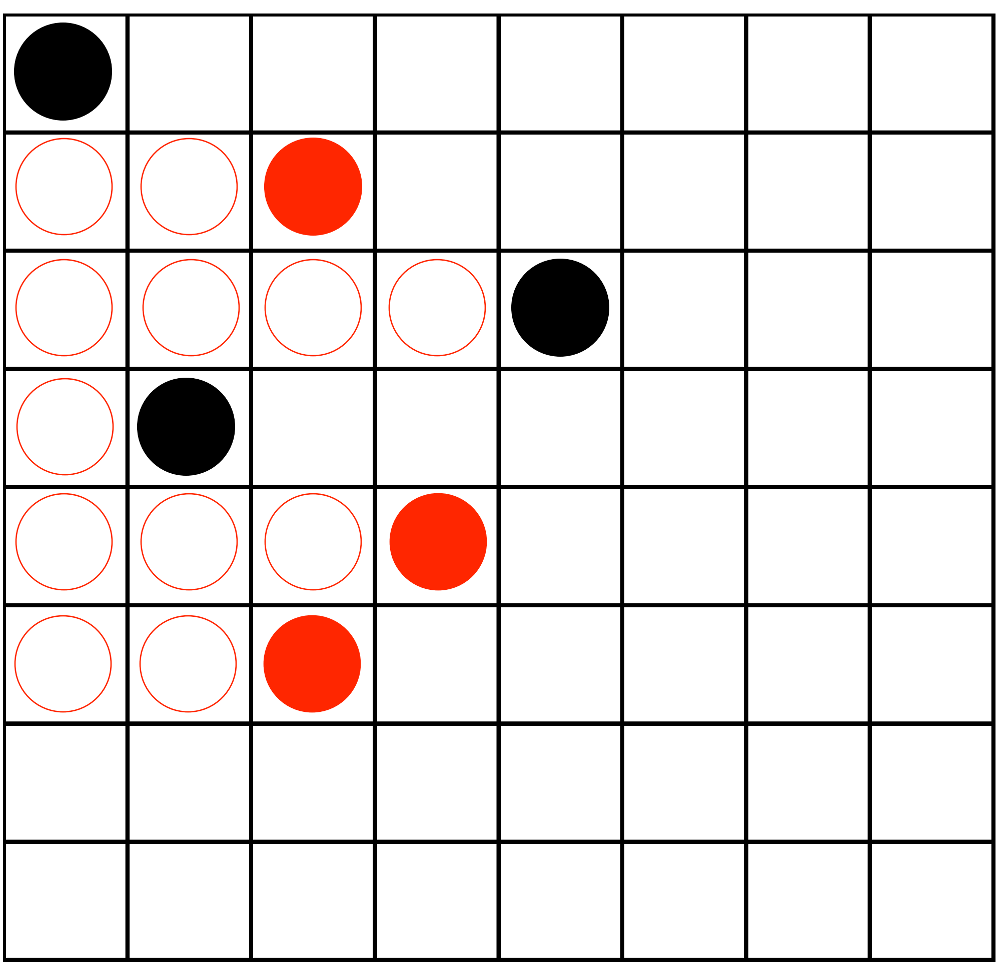
Tests 26+1 = 27 **Backtracks 0**

Backtracking



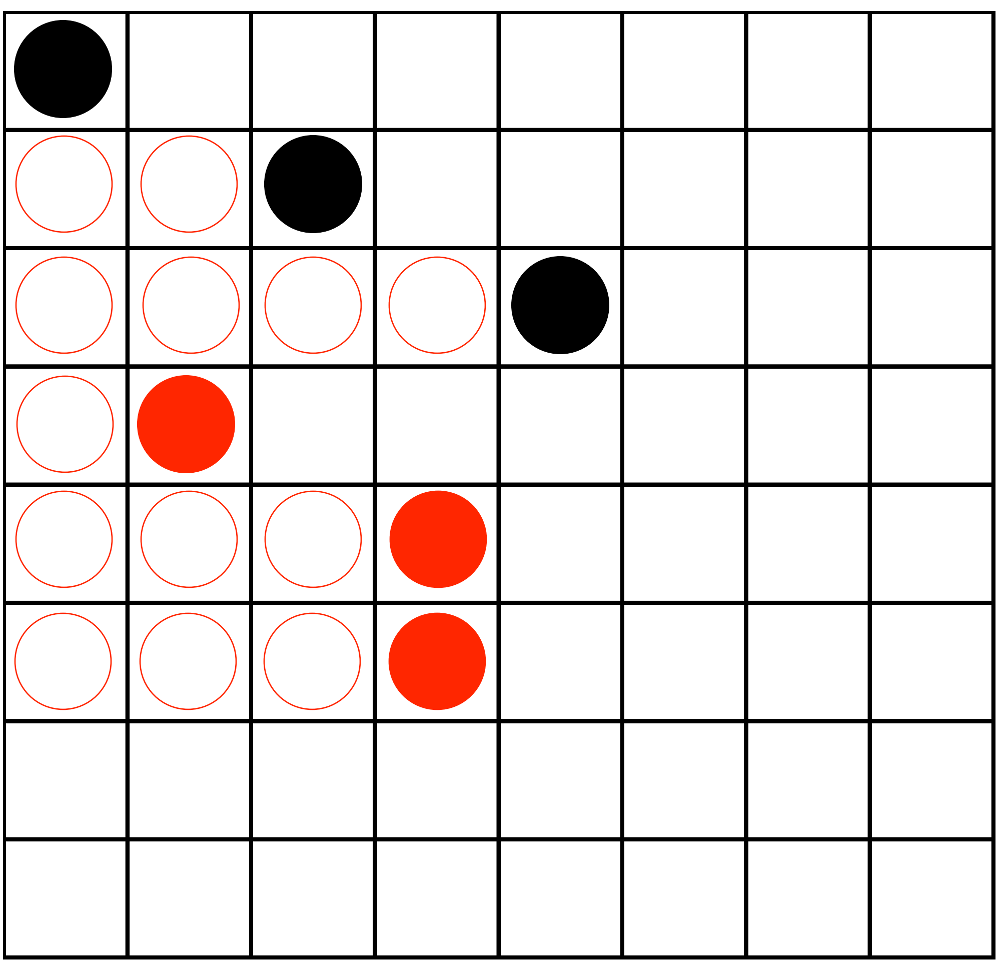
Tests 27 + 3 = 30 Backtracks 0

Backtracking

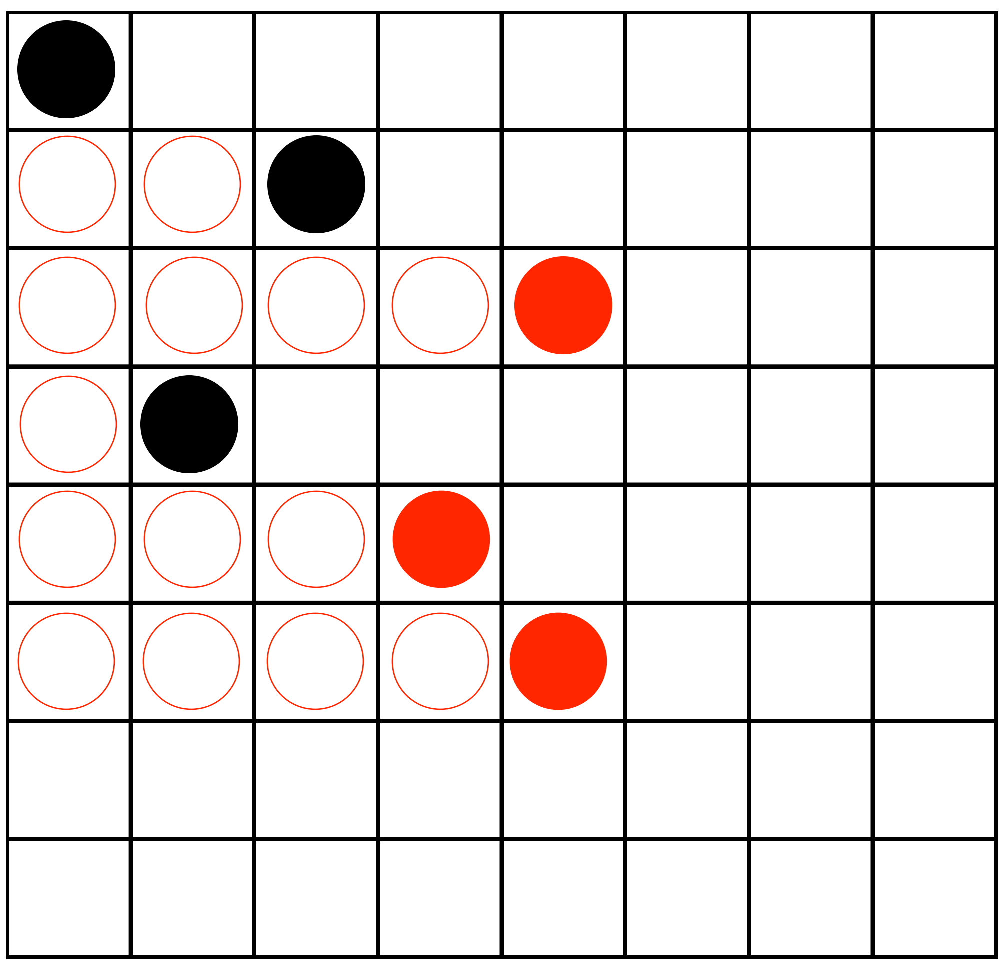


Tests 30+2 = 32 Backtracks 0

Backtracking

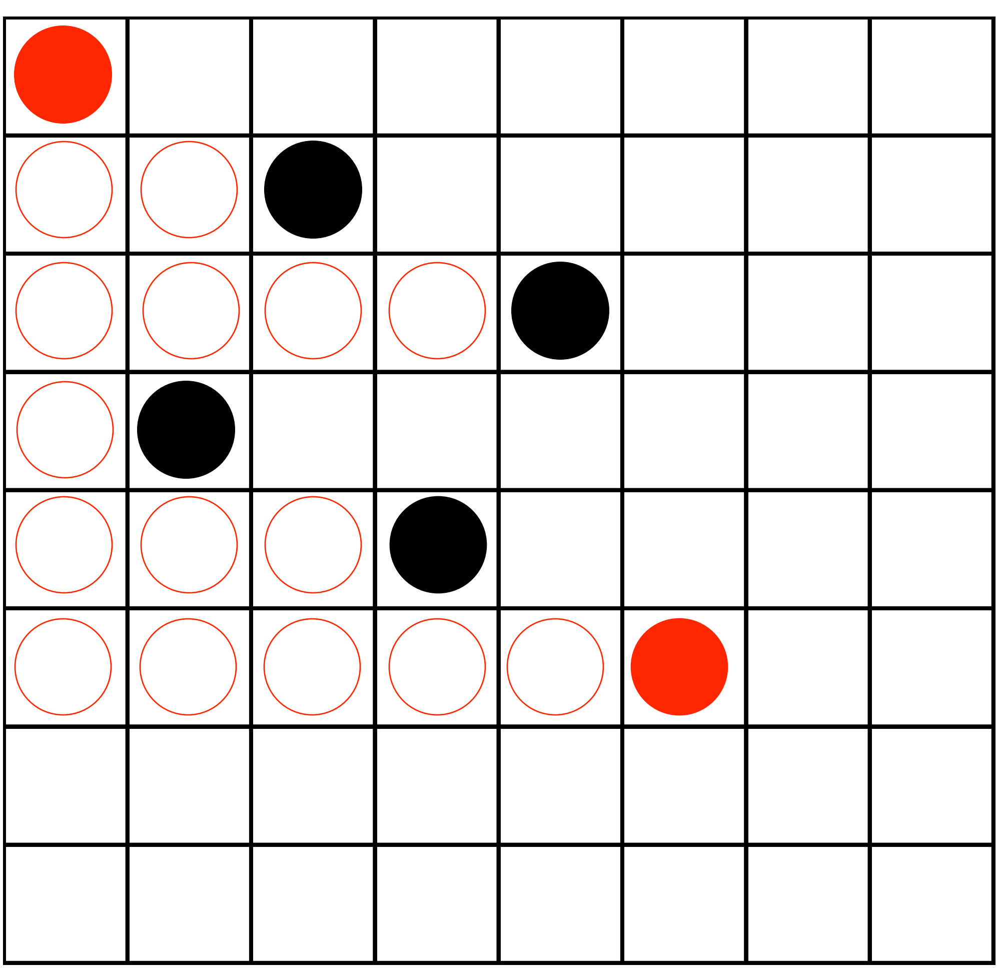


Backtracking

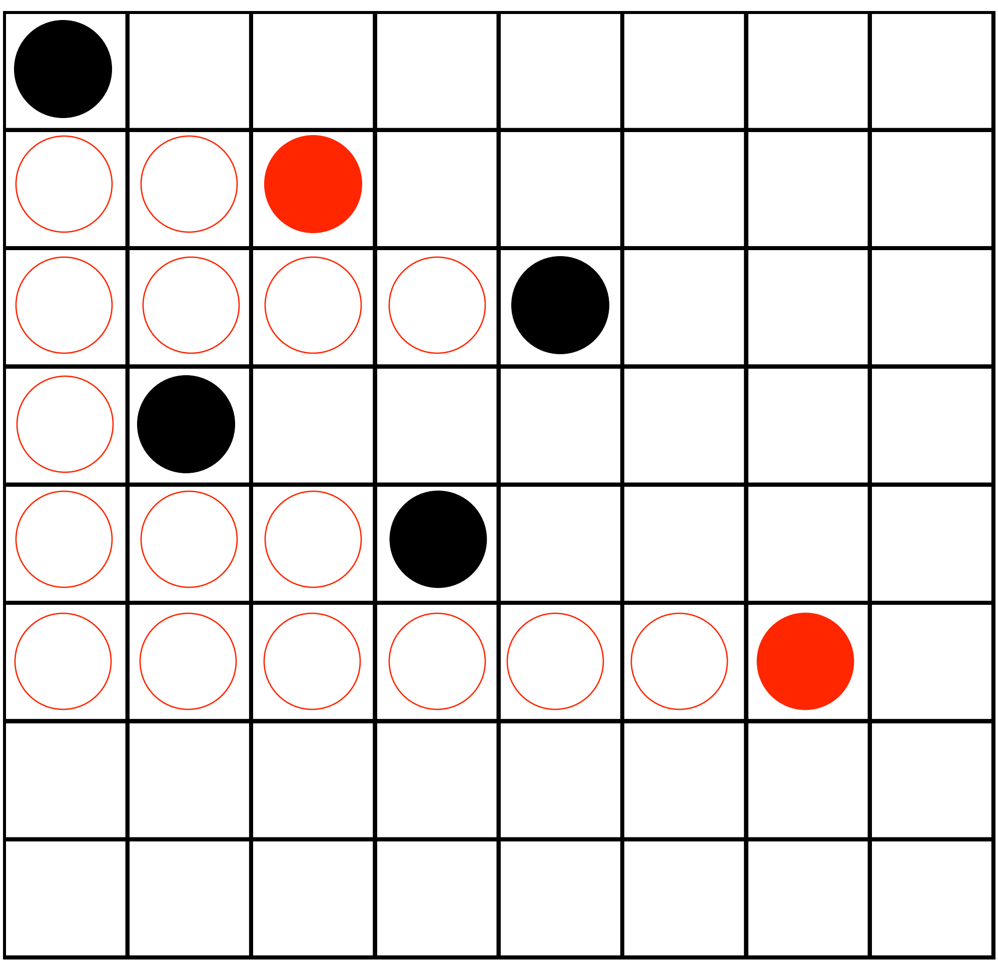


Tests 36 + 3 = 39 **Backtracks 0**

Backtracking

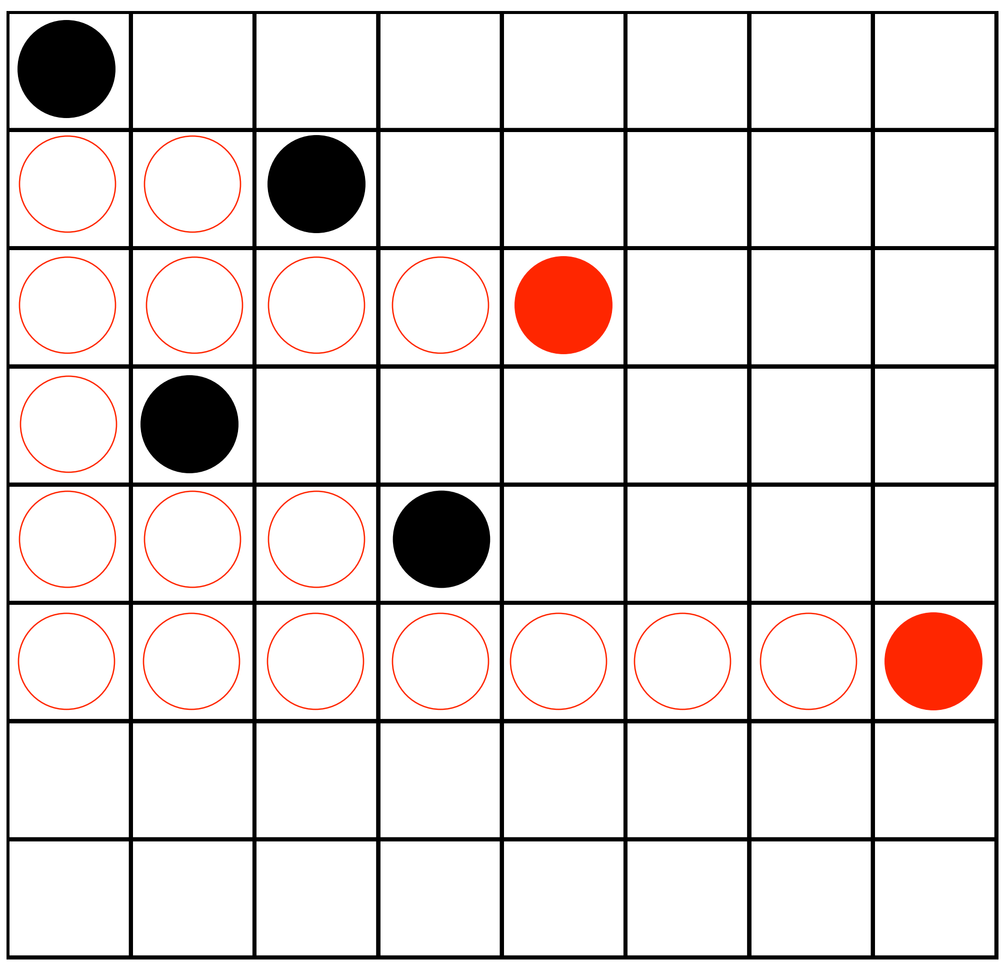


Backtracking



Tests 40 + 2 = 42 **Backtracks 0**

Backtracking

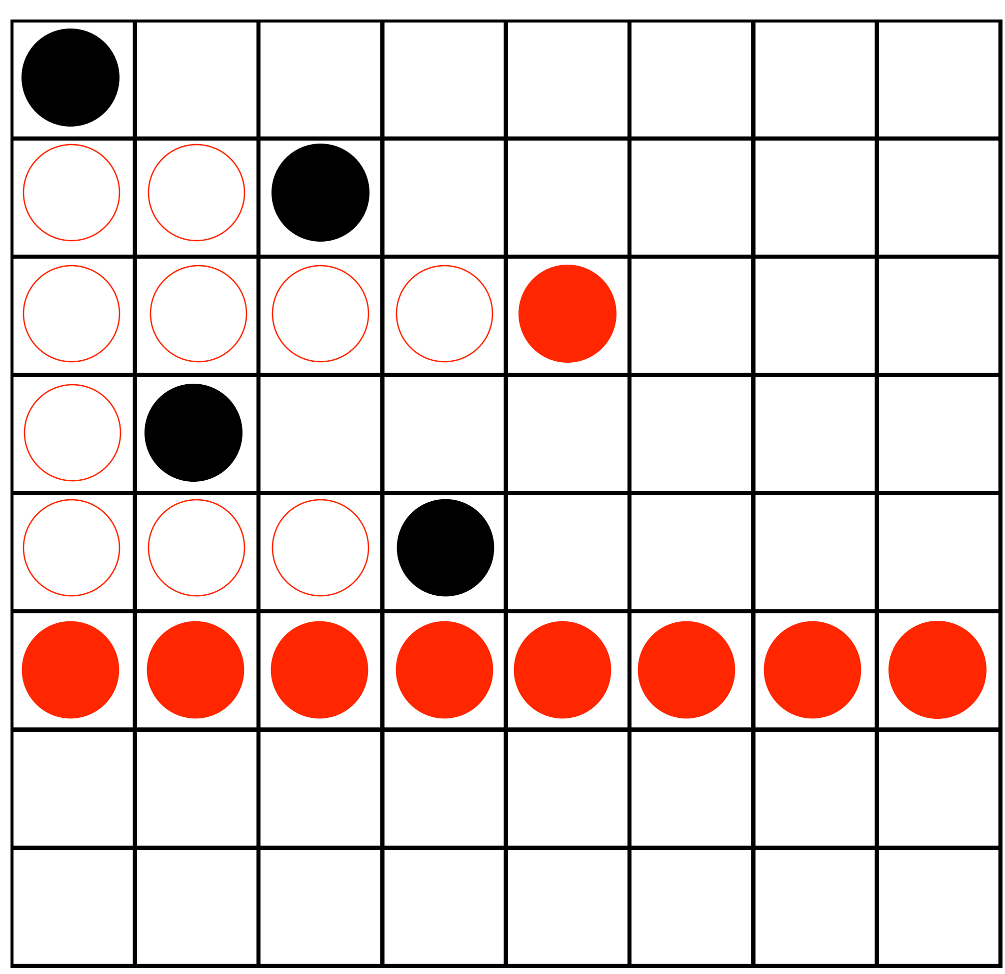


Tests 42 + 3 = 45 Backtracks 0

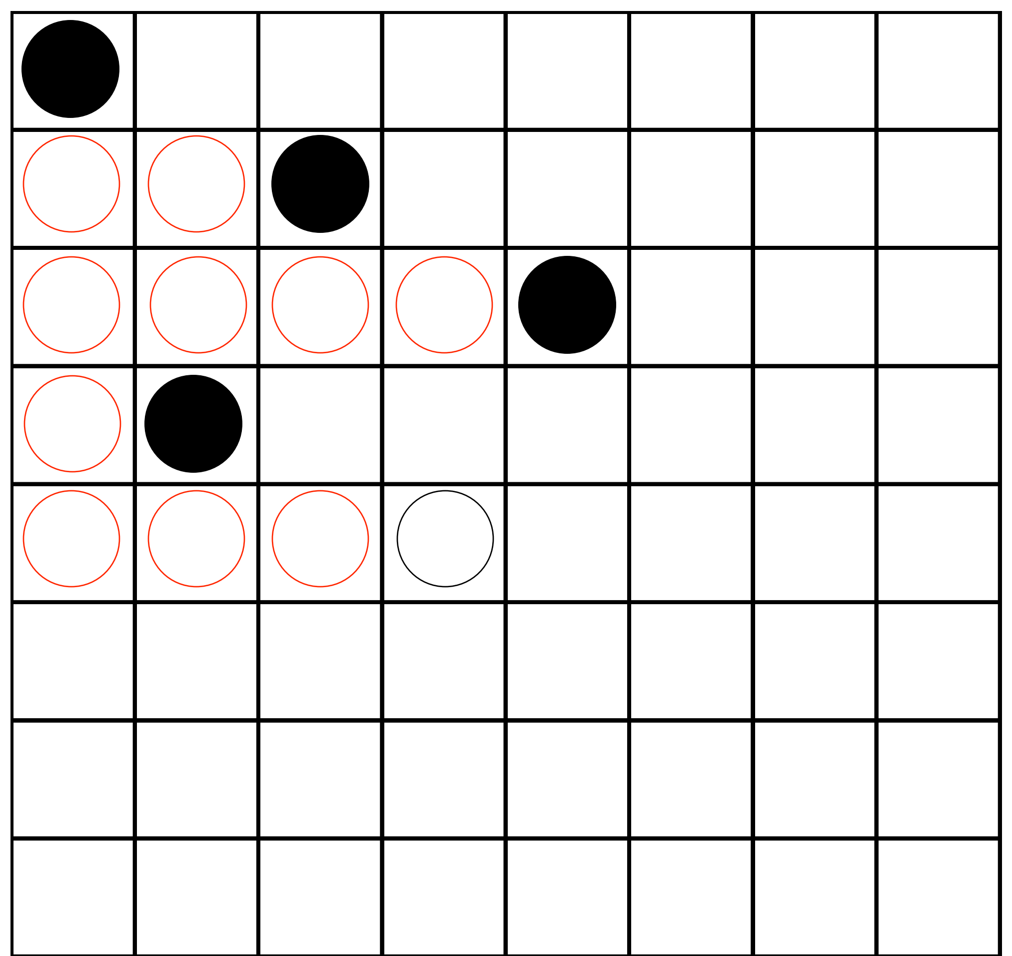
Backtracking

Q6 Fails

Backtracks to Q5

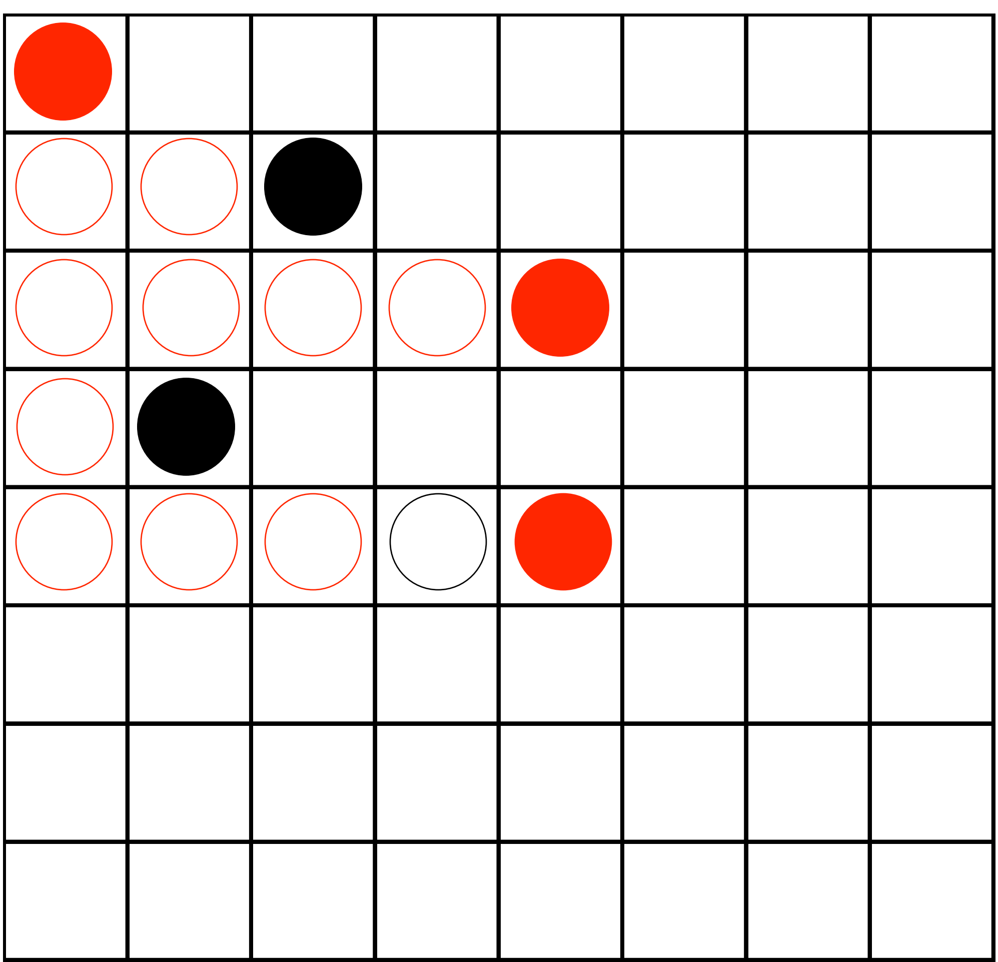


Backtracking



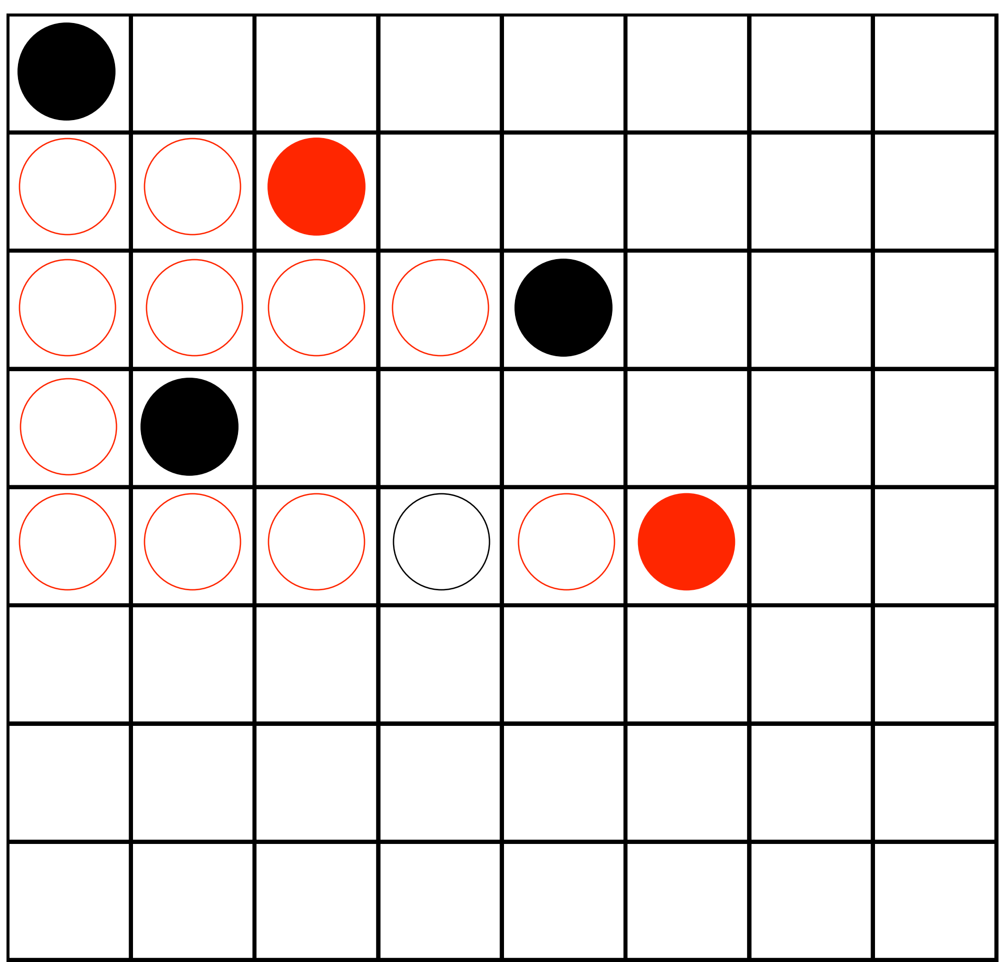
Tests 45 **Backtrackings 1**

Backtracking



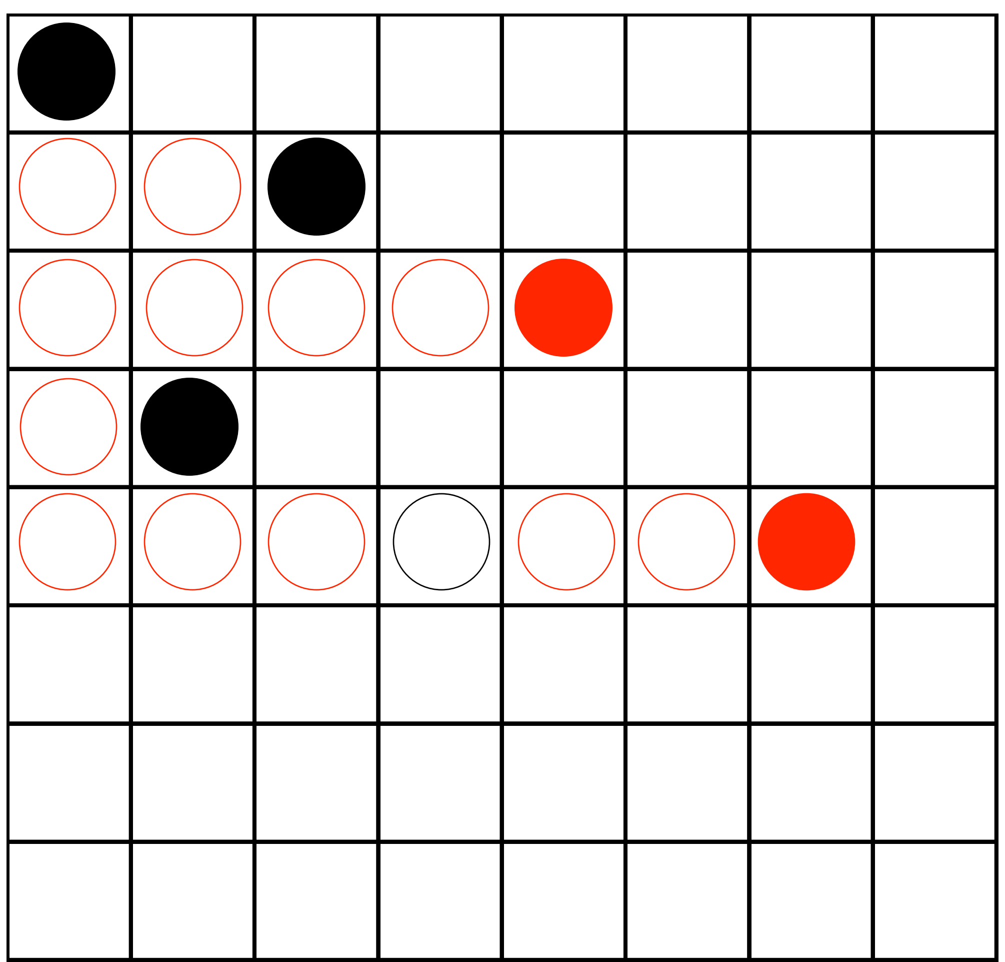
Tests 45 + 1 = 46 **Backtracks 1**

Backtracking



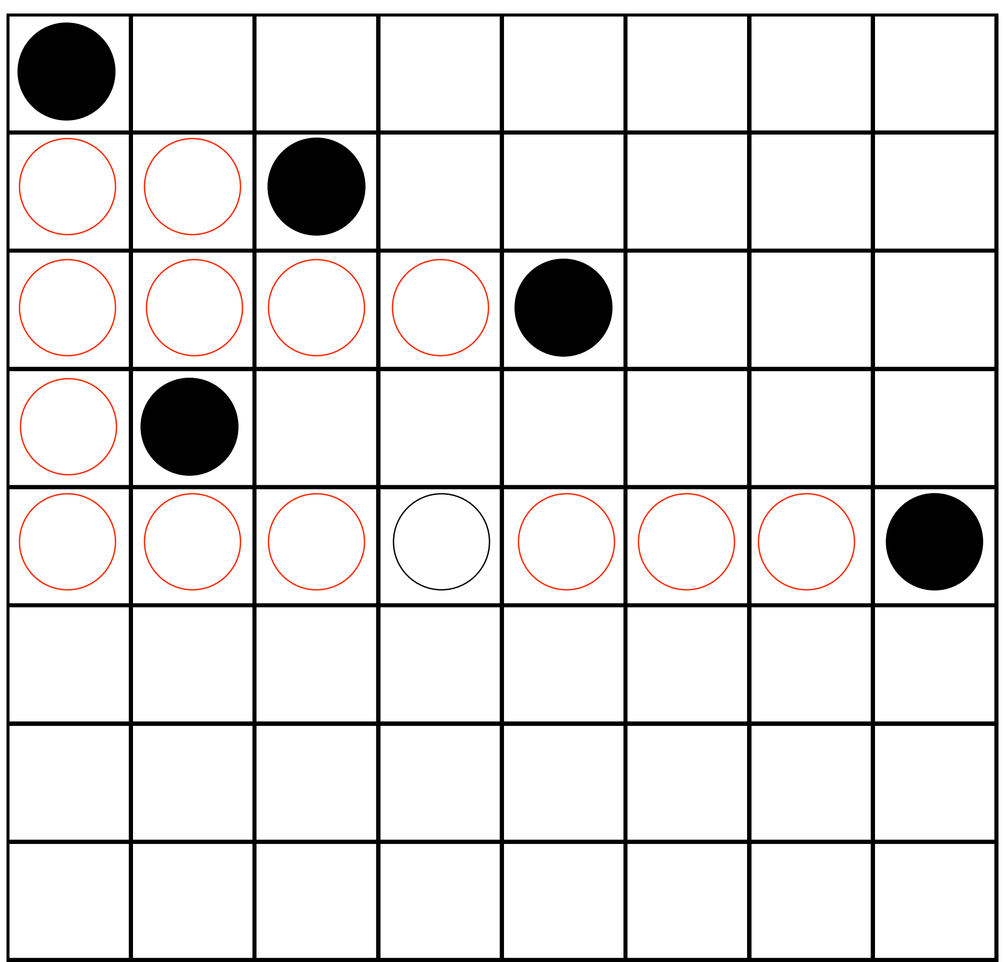
Tests 46 + 2 = 48 **Backtracks 1**

Backtracking



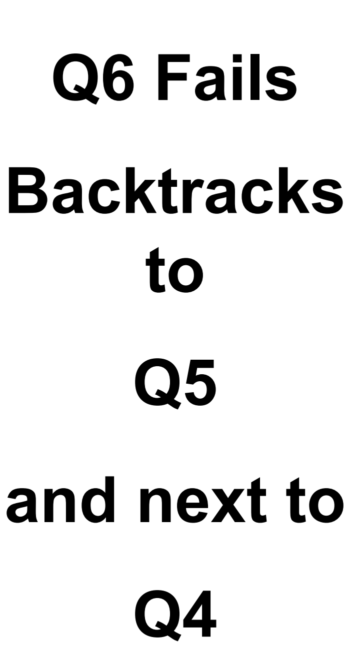
Tests 48 + 3 = 51 **Backtracks 1**

Backtracking



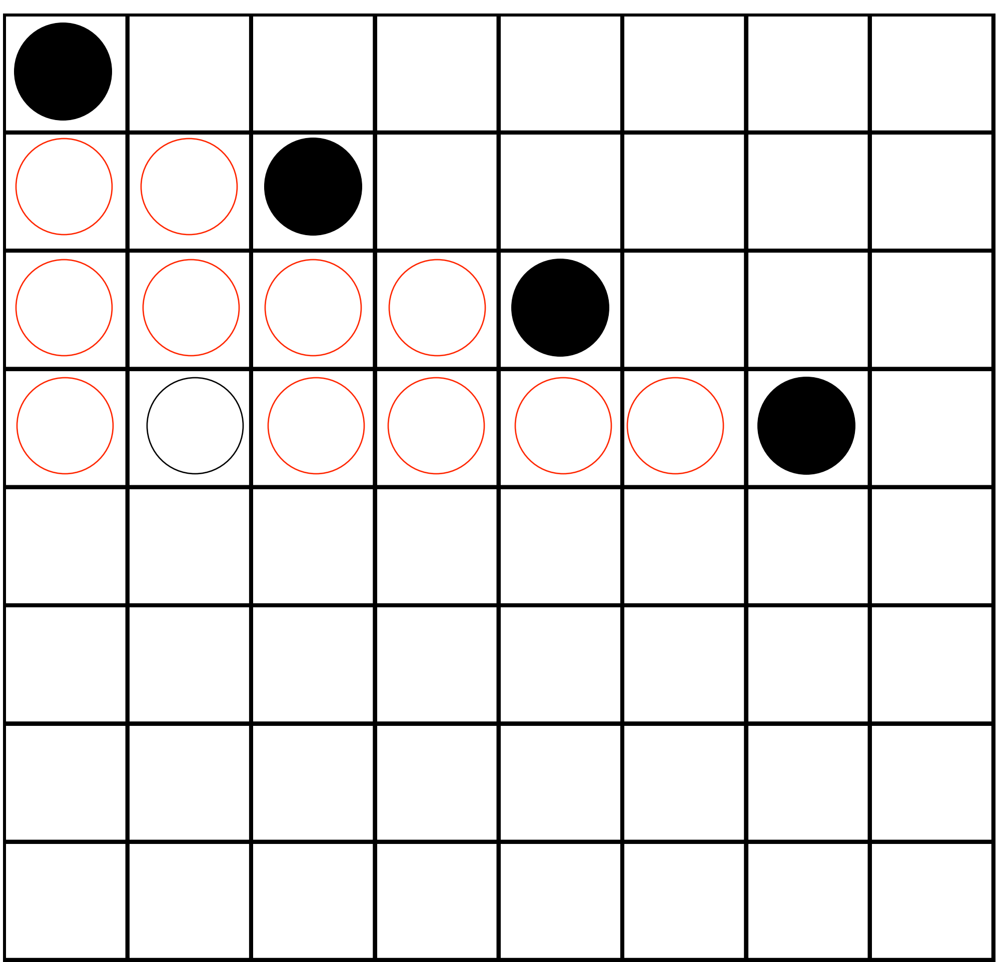
Tests 51 + 4 = 55 **Backtracks 1**

Backtracking



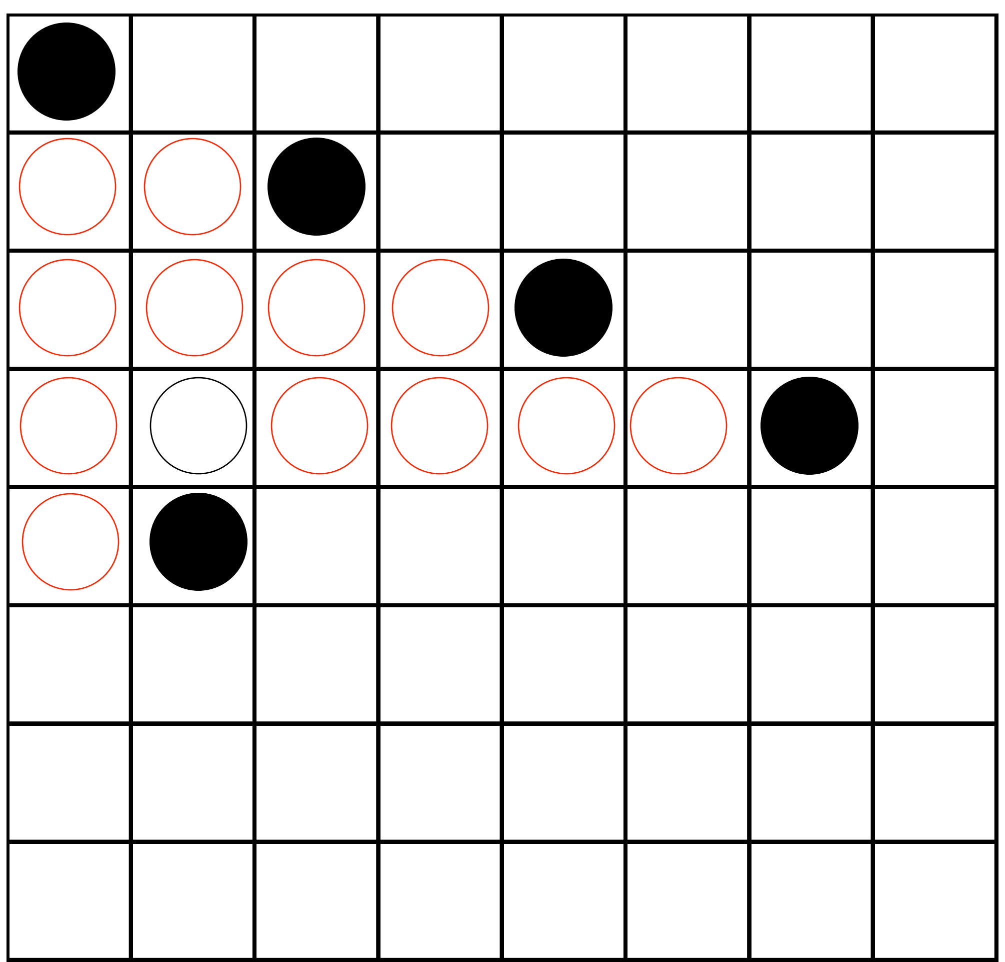
Tests 55+1+3+2+4+3+1+2+3 = 74 **Backtracks 1+2 = 3**

Backtracking



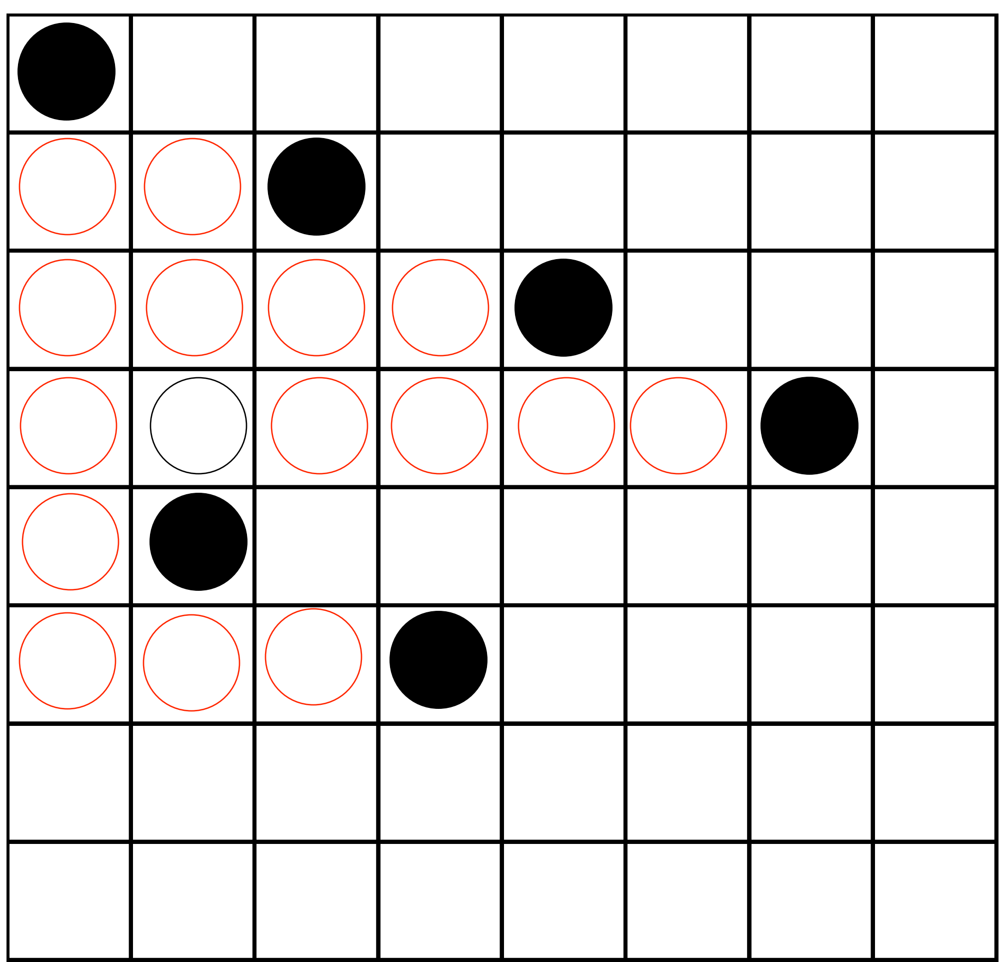
Tests 74+2+1+2+3+3= 85 Backtracks 3

Backtracking



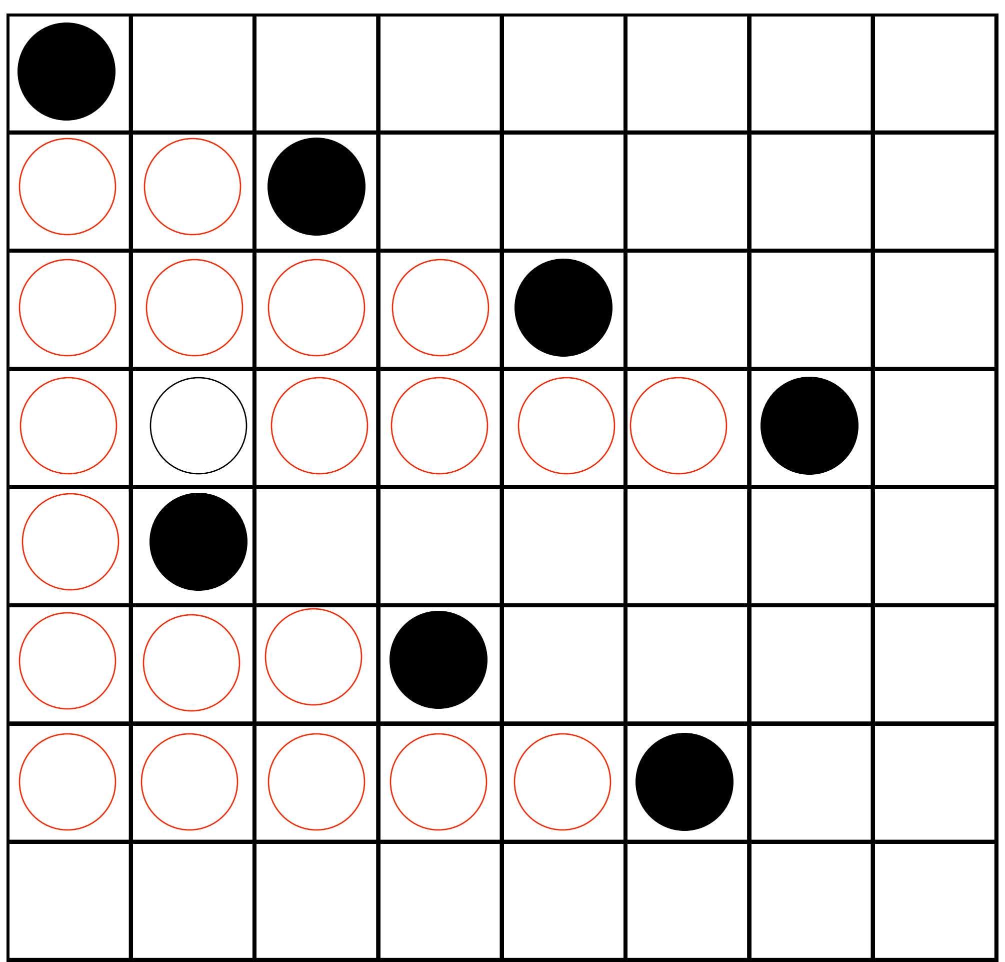
Tests 85 + 1 + 4 = 90 **Backtracks 3**

Backtracking



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

Backtracking

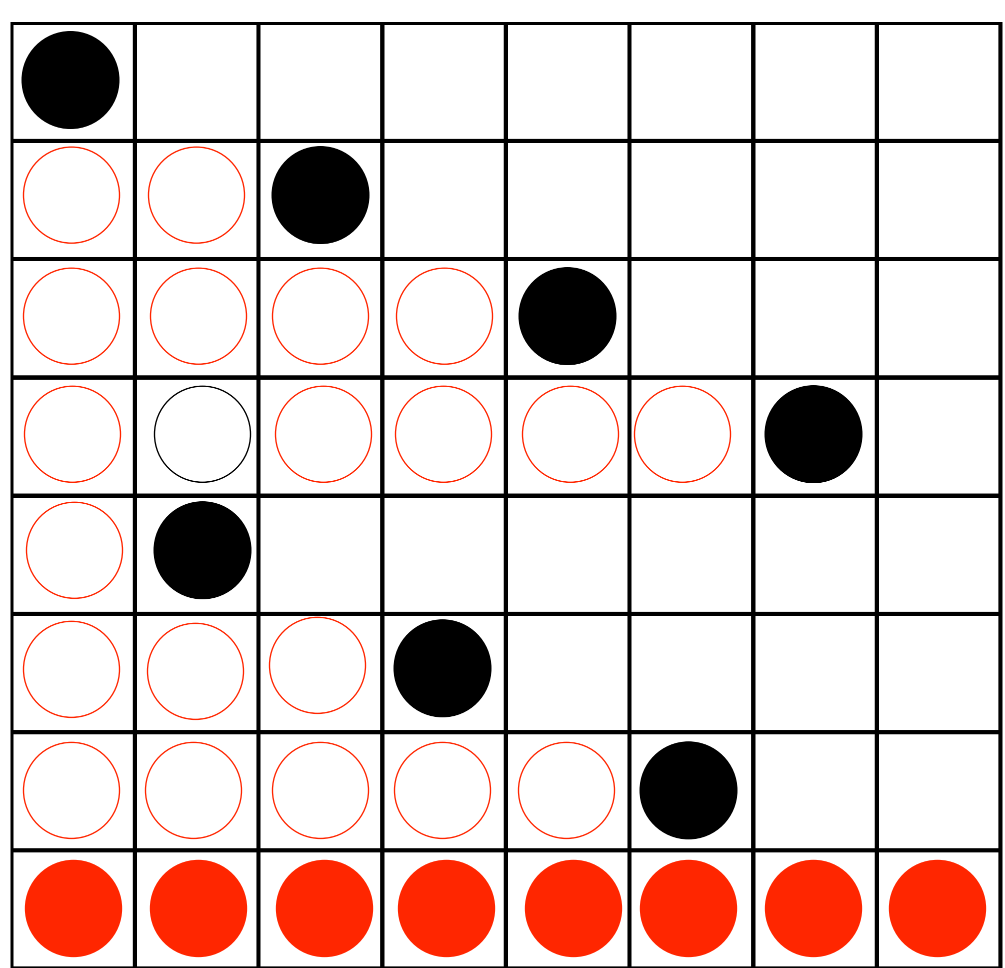


Tests 101+1+5+2+4+3+6= 122 **Backtracks 3**

Backtracking

Q8 Fails

Backtracks to Q7

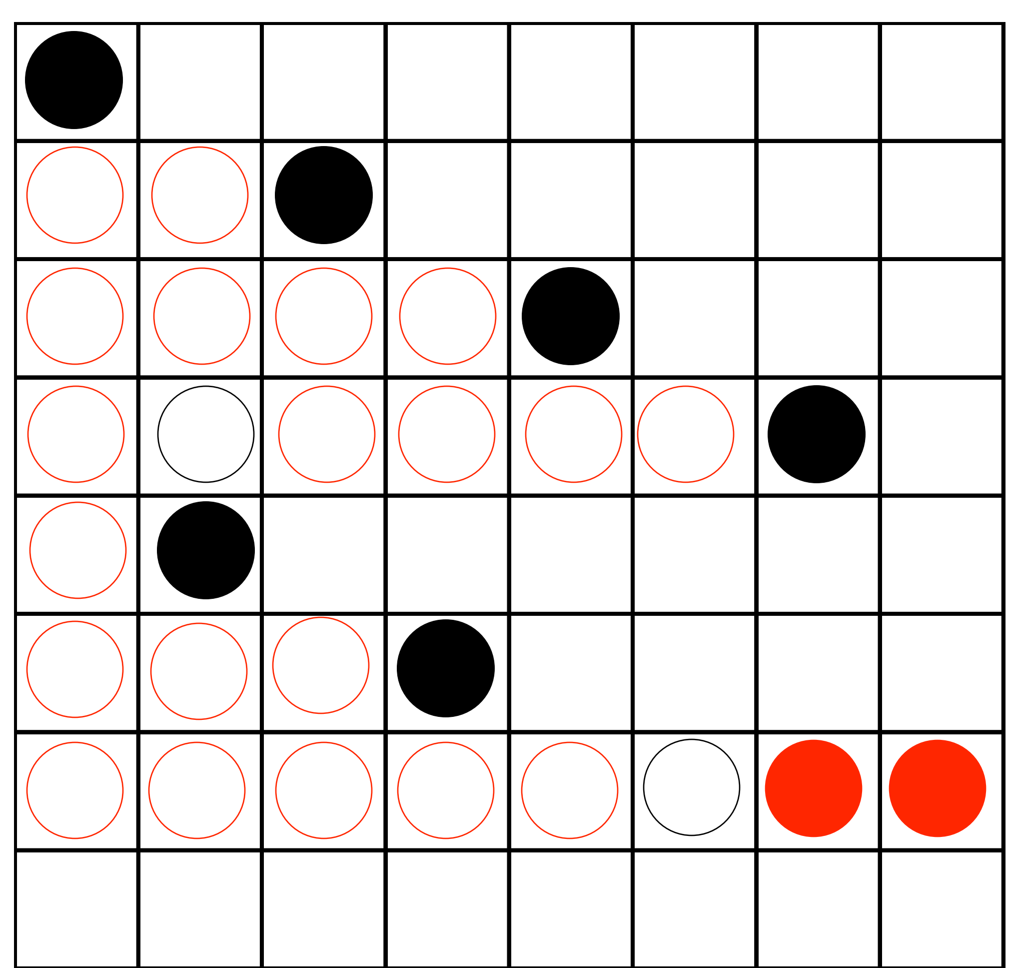


Tests 122+1+5+2+6+3+6+4+1= 150 Backtracks 3+1=4

Backtracking

Q7 Fails

Backtracks
to
Q6

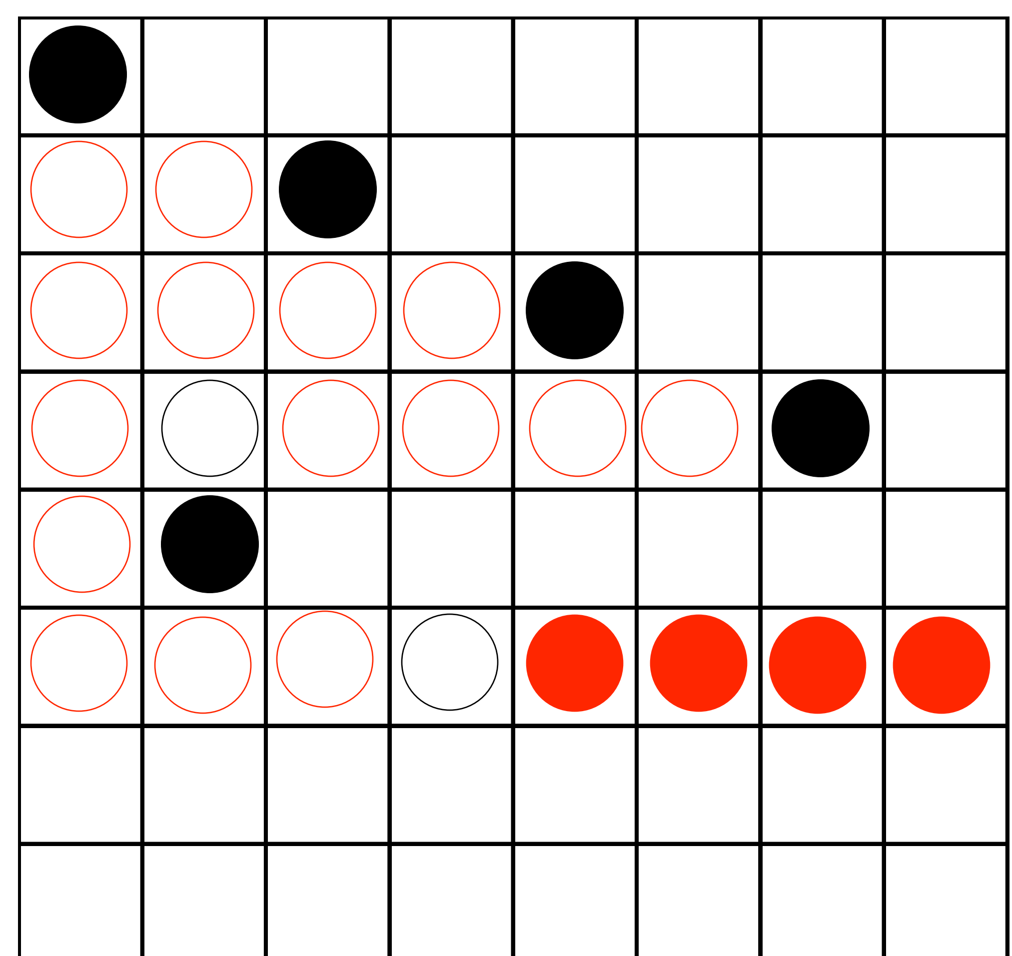


Tests 150+1+2= 153 Backtracks 4+1=5

Backtracking

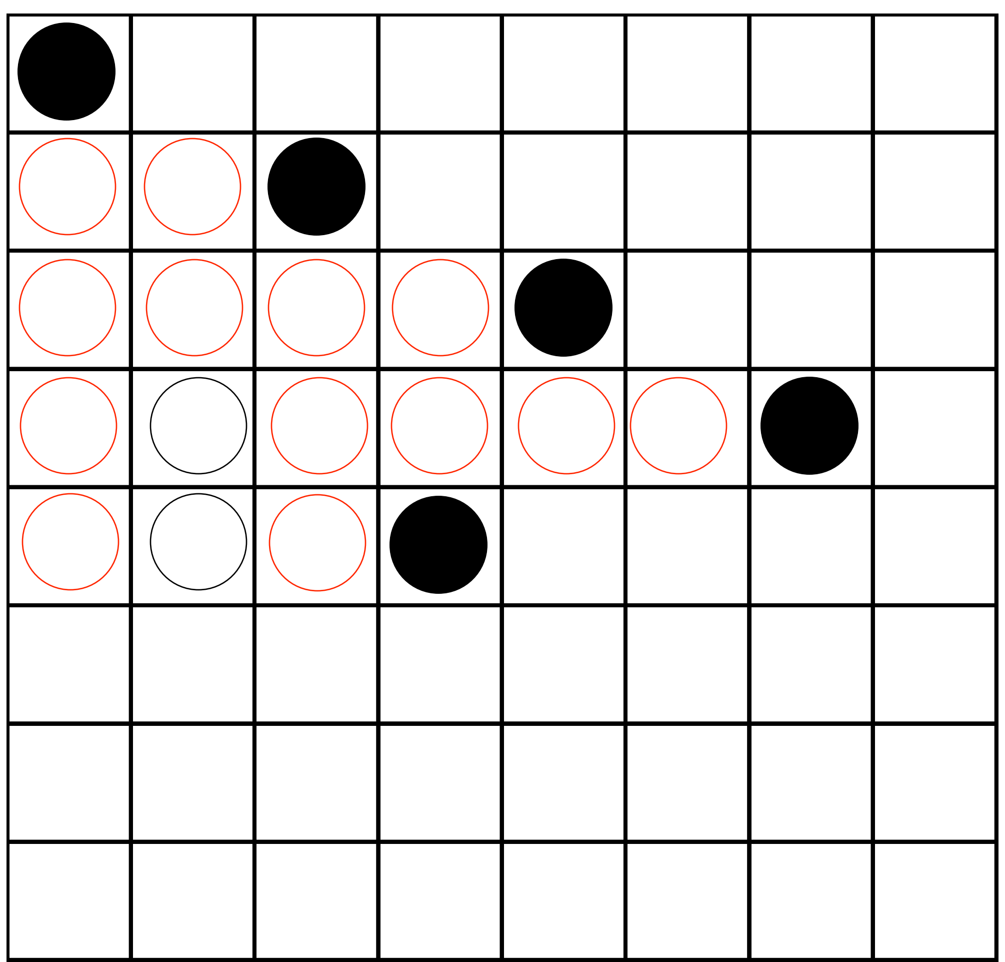
Q6 Fails

Backtracks to Q5



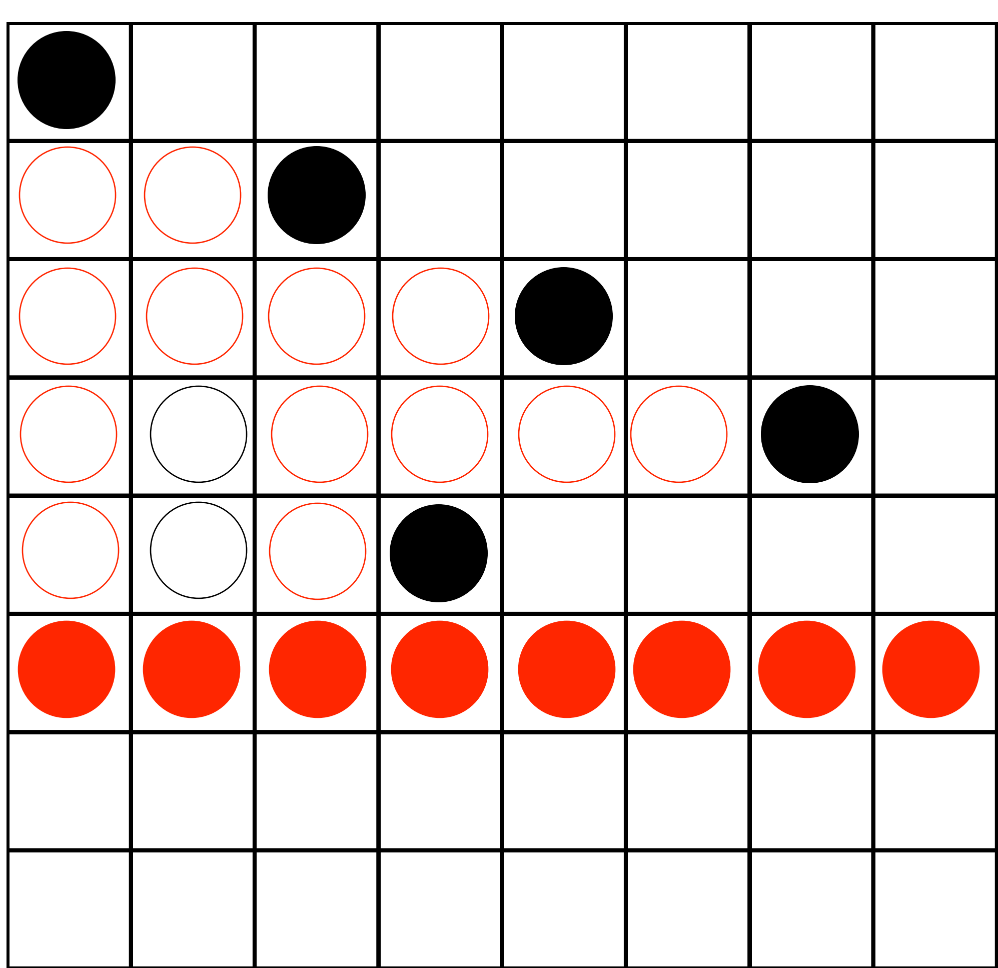
Tests 153+3+1+2+3= 162 Backtracks 5+1=6

Backtracking



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

Backtracking



Q6 Fails

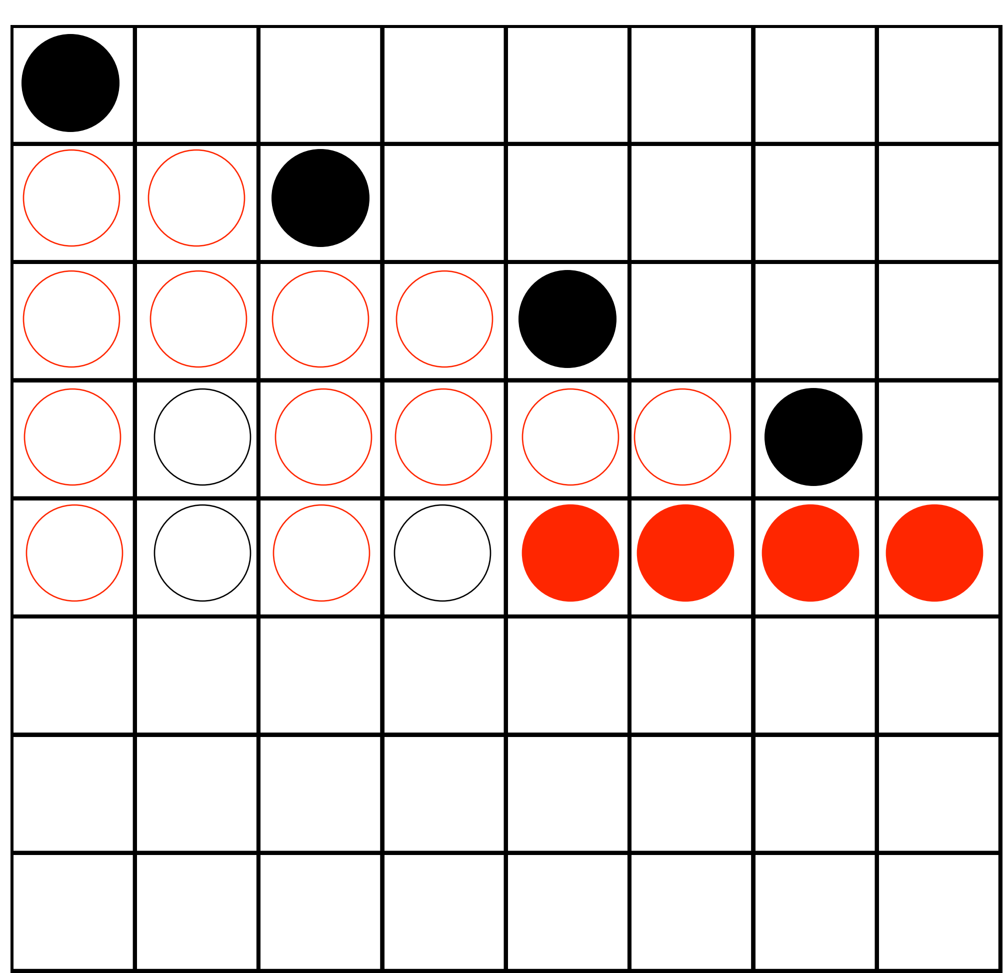
Backtracks to Q5

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

Backtracking

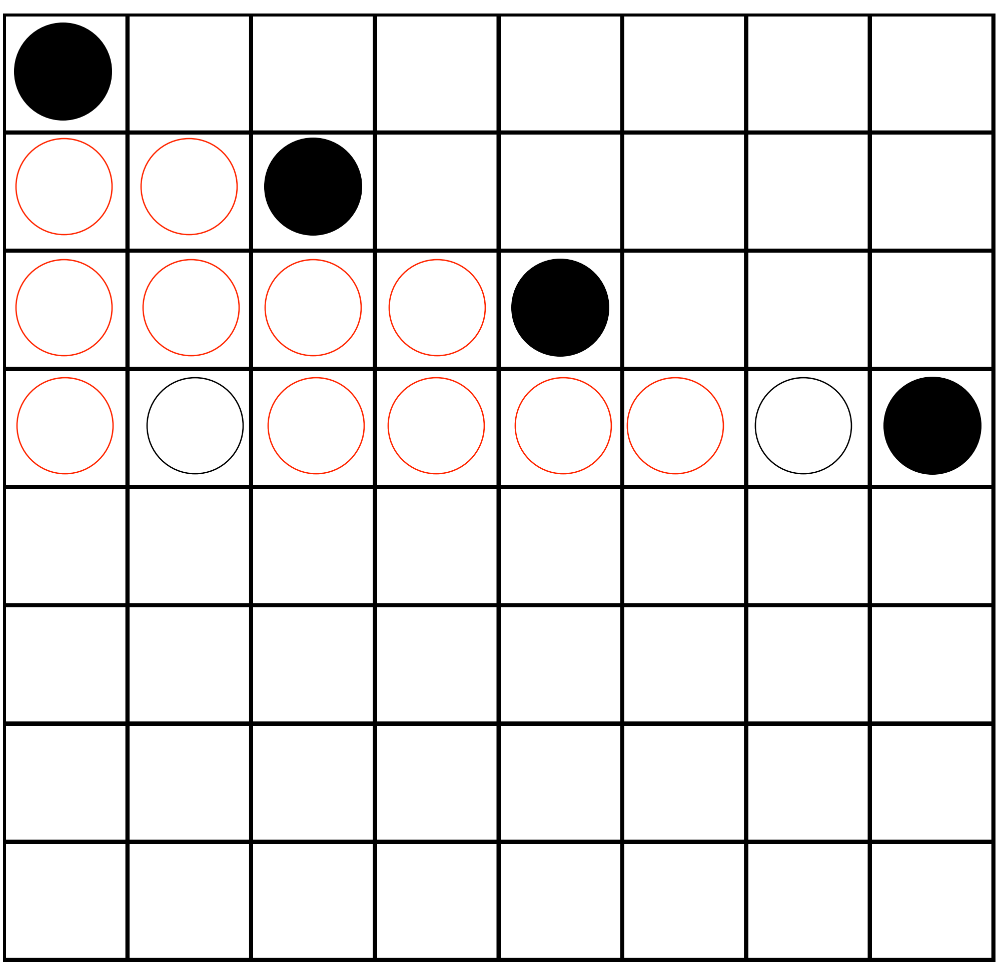
Q5 Fails

Backtracks to Q4



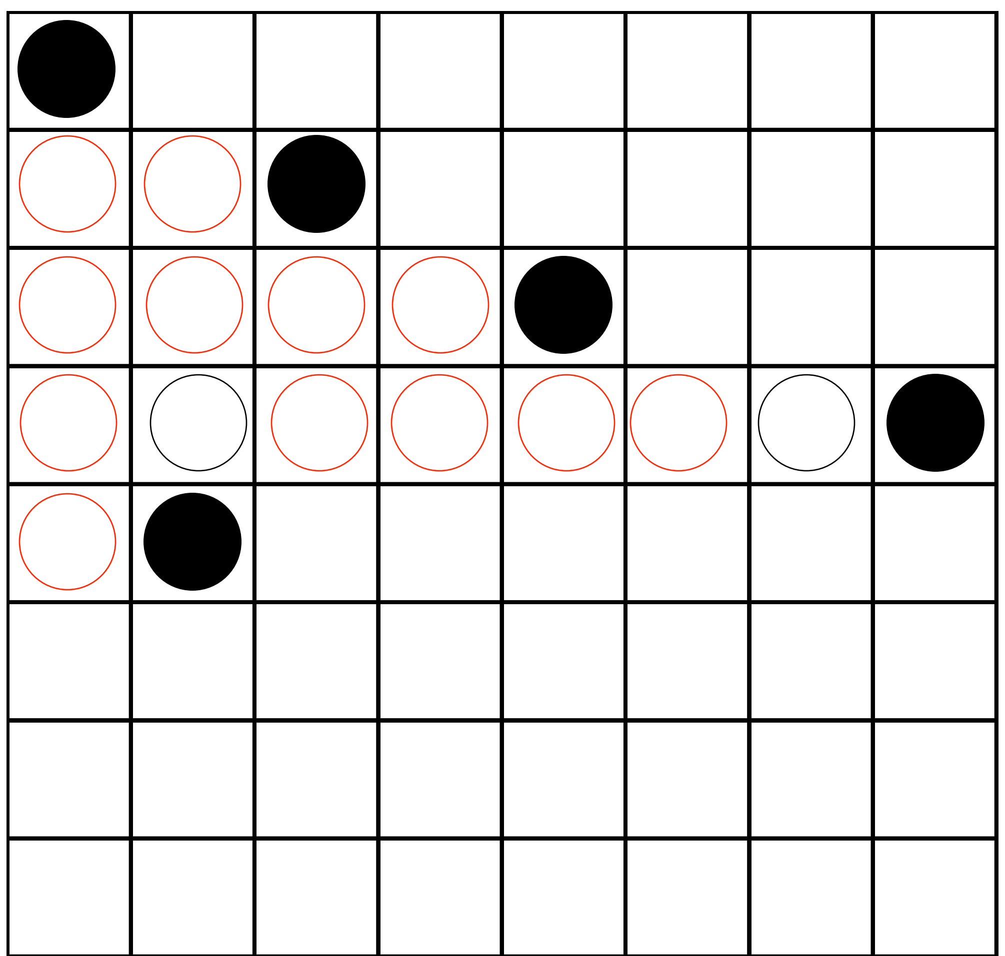
Tests 188+1+2+3+4= 198 **Backtracks 7+1=8**

Backtracking



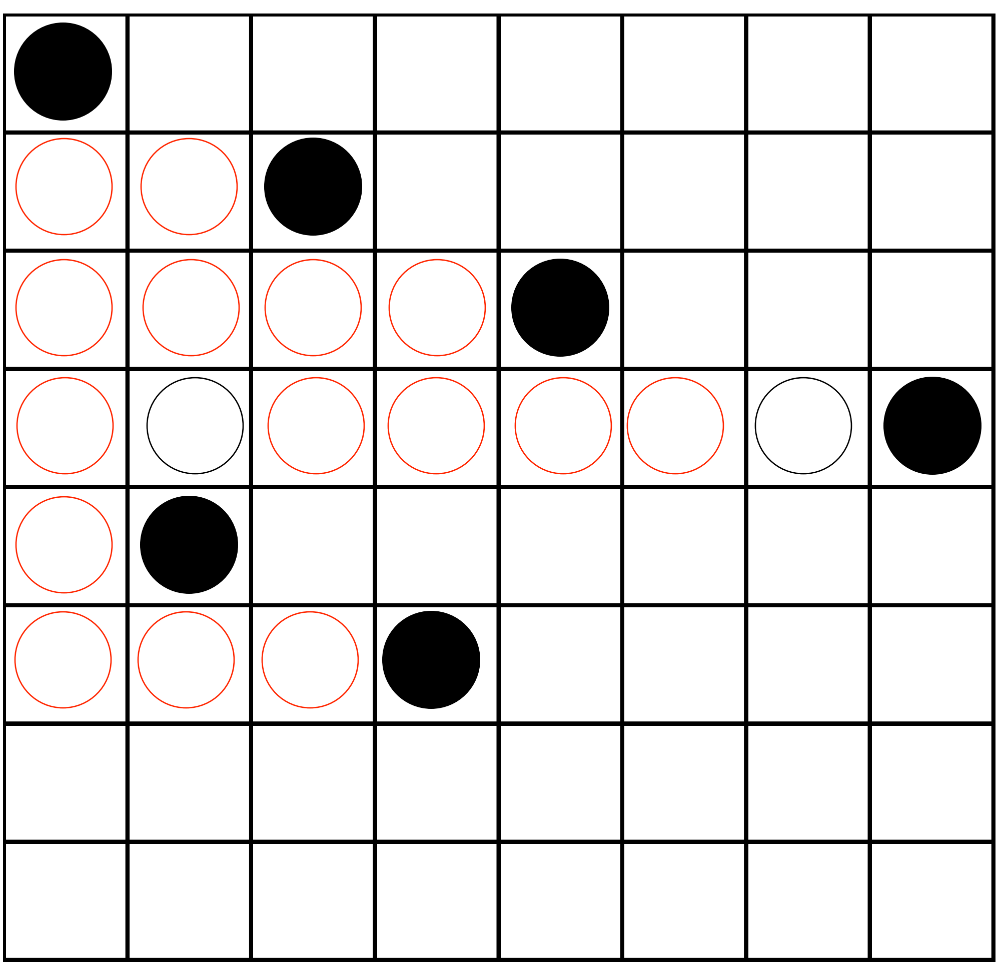
Tests 198 + 3 = 201 **Backtracks 8**

Backtracking



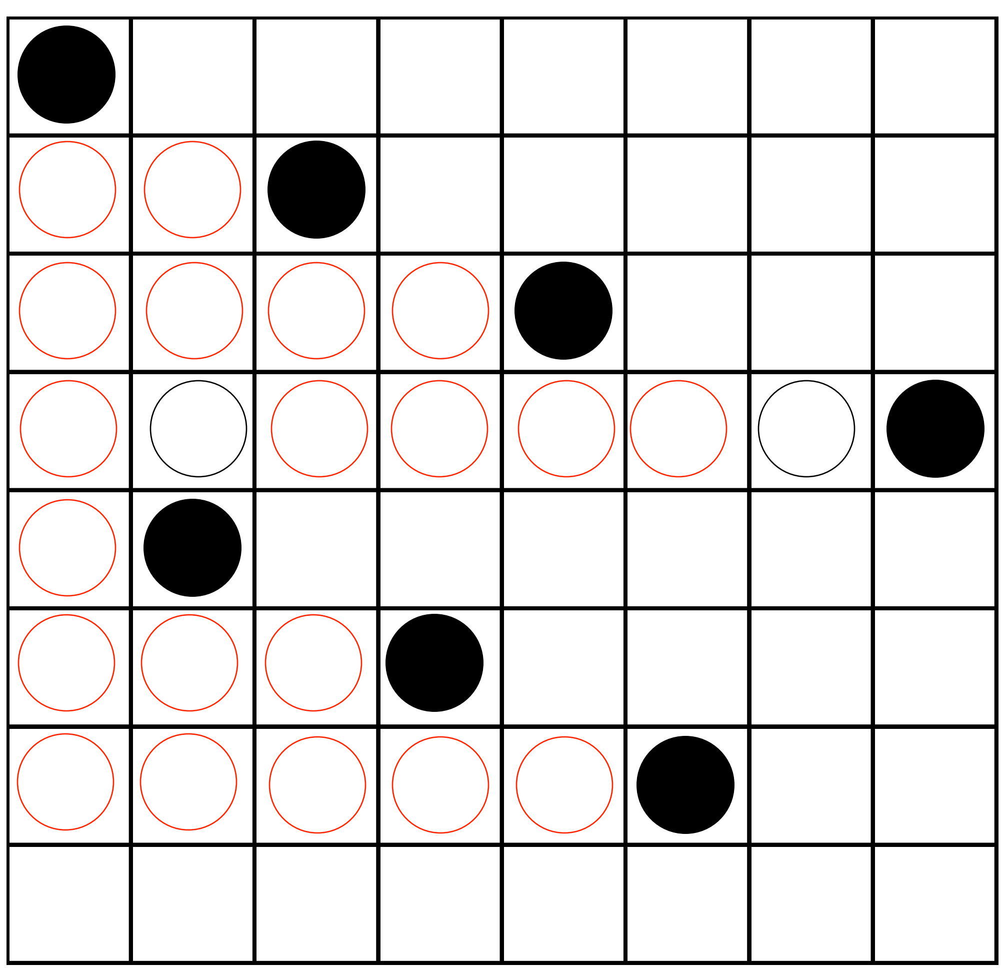
Tests 201+1+4 = 206 **Backtracks 8**

Backtracking



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

Backtracking

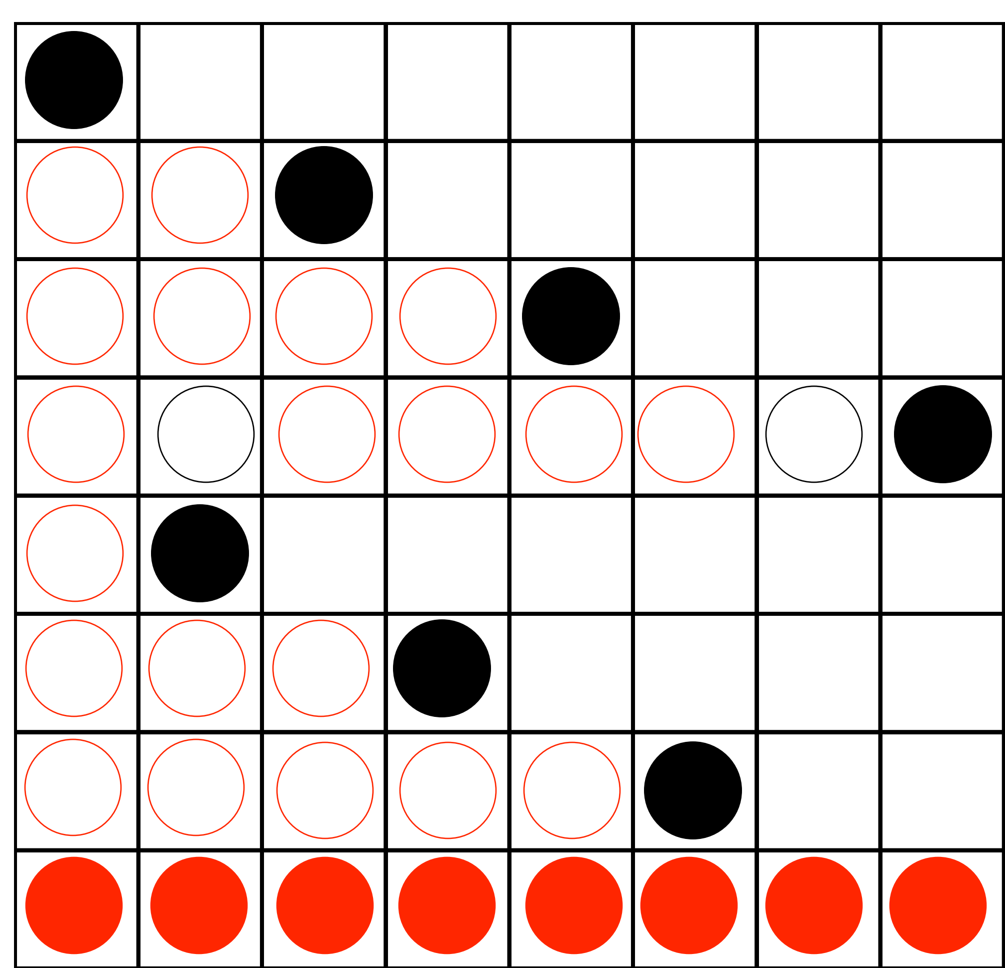


Tests 217+1+5+2+5+3+6 = 239 **Backtracks 8**

Backtracking

Q8 Fails

Backtracks to Q7

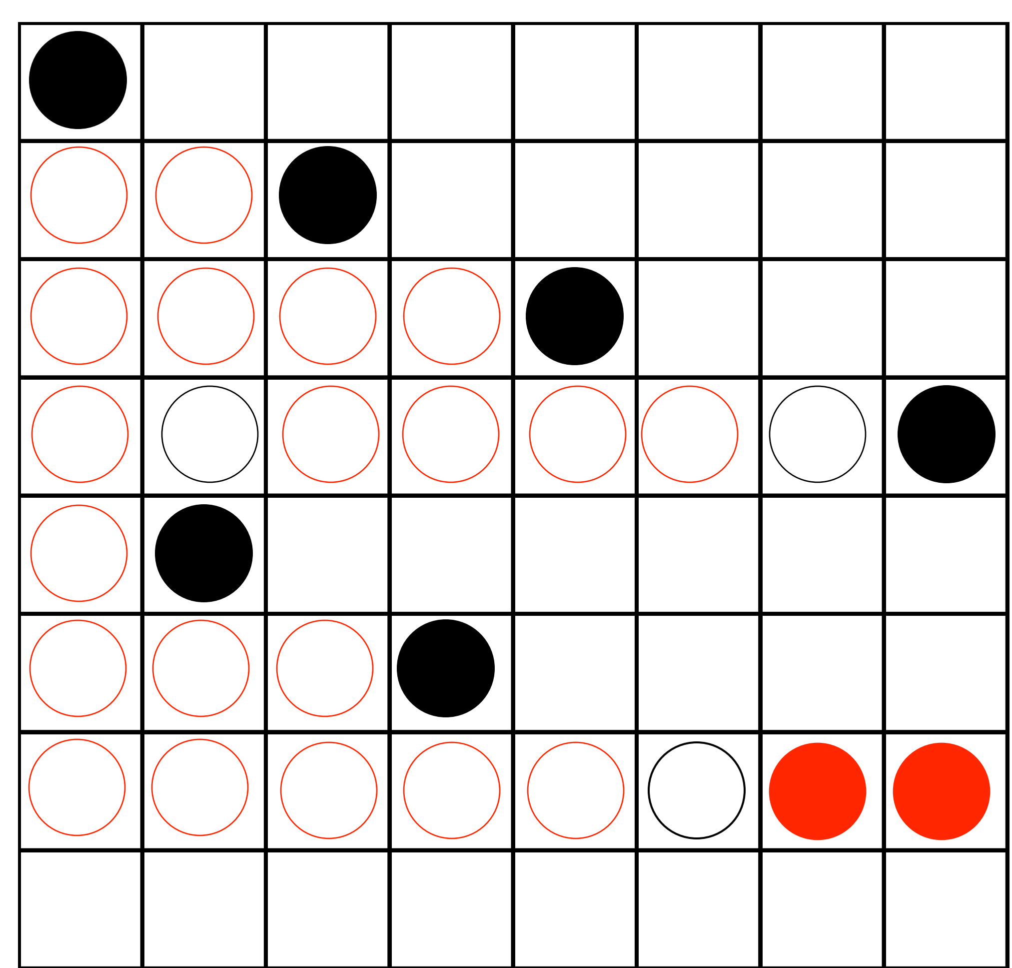


Tests 239+1+5+2+4+3+6+7+7= 274 Backtracks 8+1 = 9

Backtracking

Q7 Fails

Backtracks to Q6

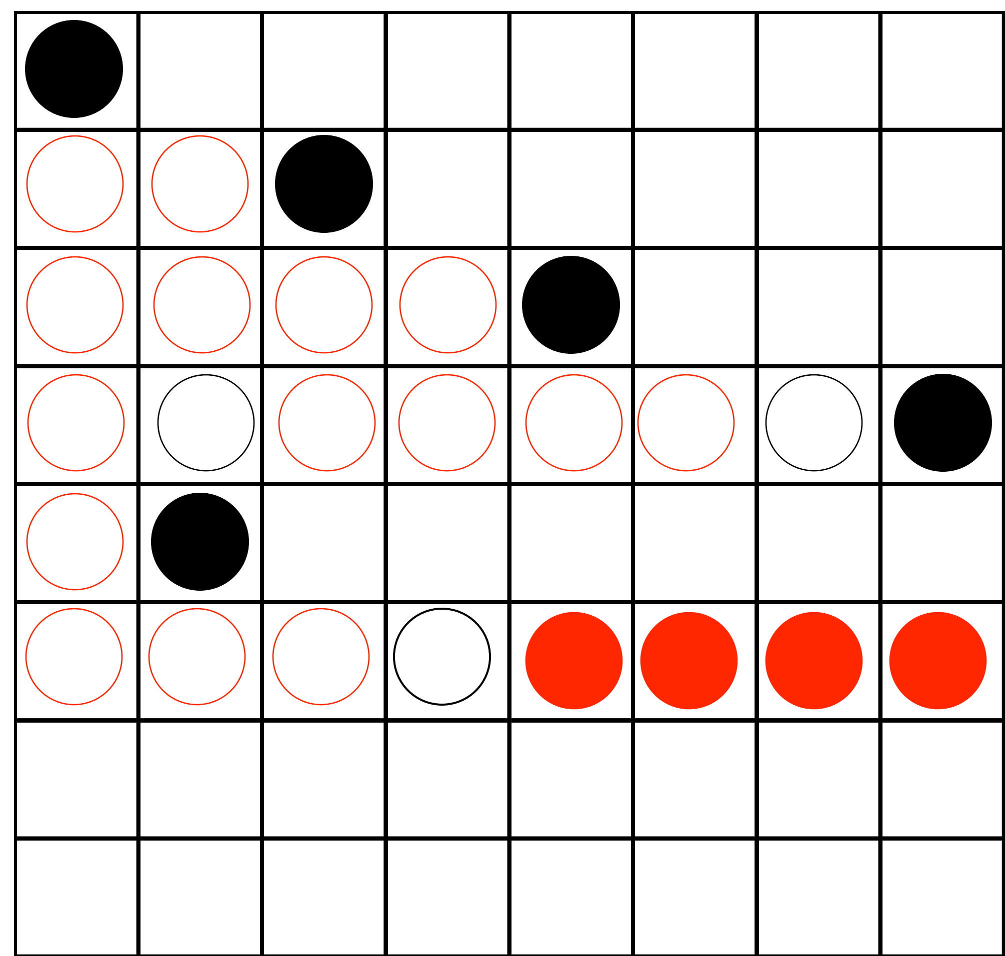


Tests 274+1+2= 277 Backtracks 9+1=10

Backtracking

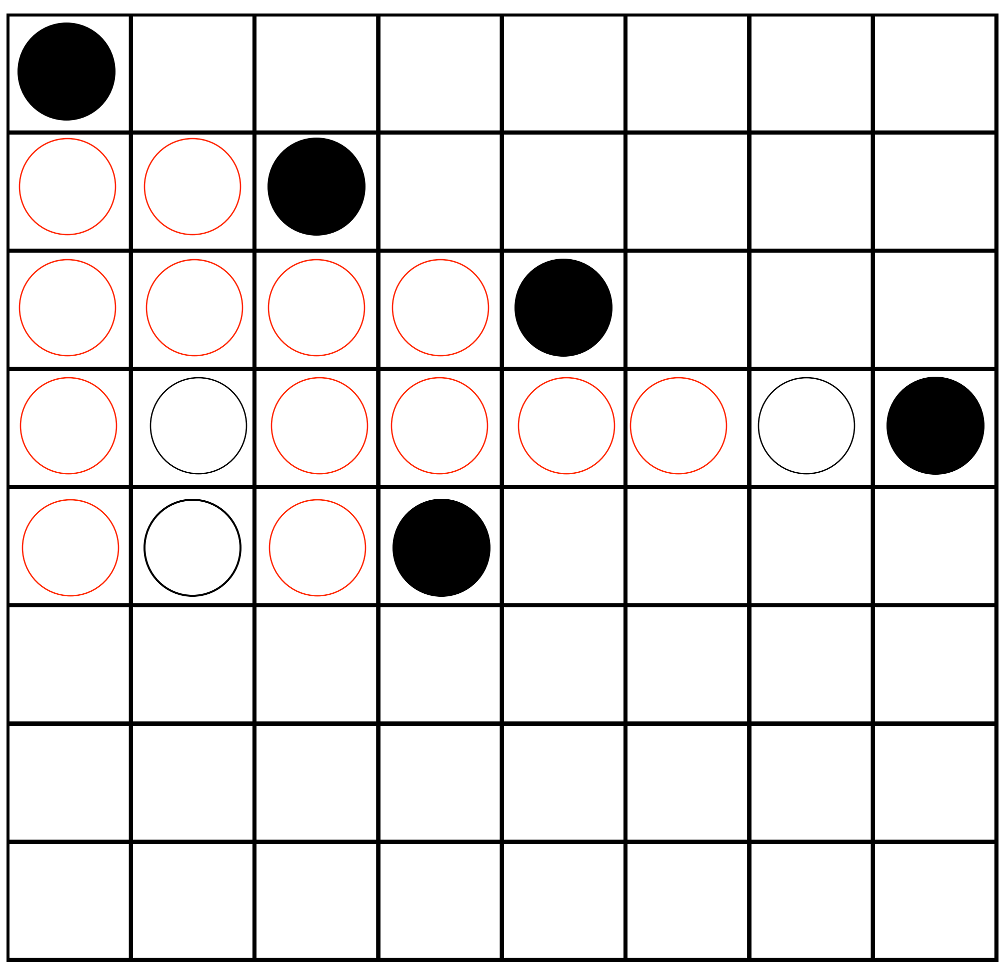
Q6 Fails

Backtracks to Q5



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

Backtracking

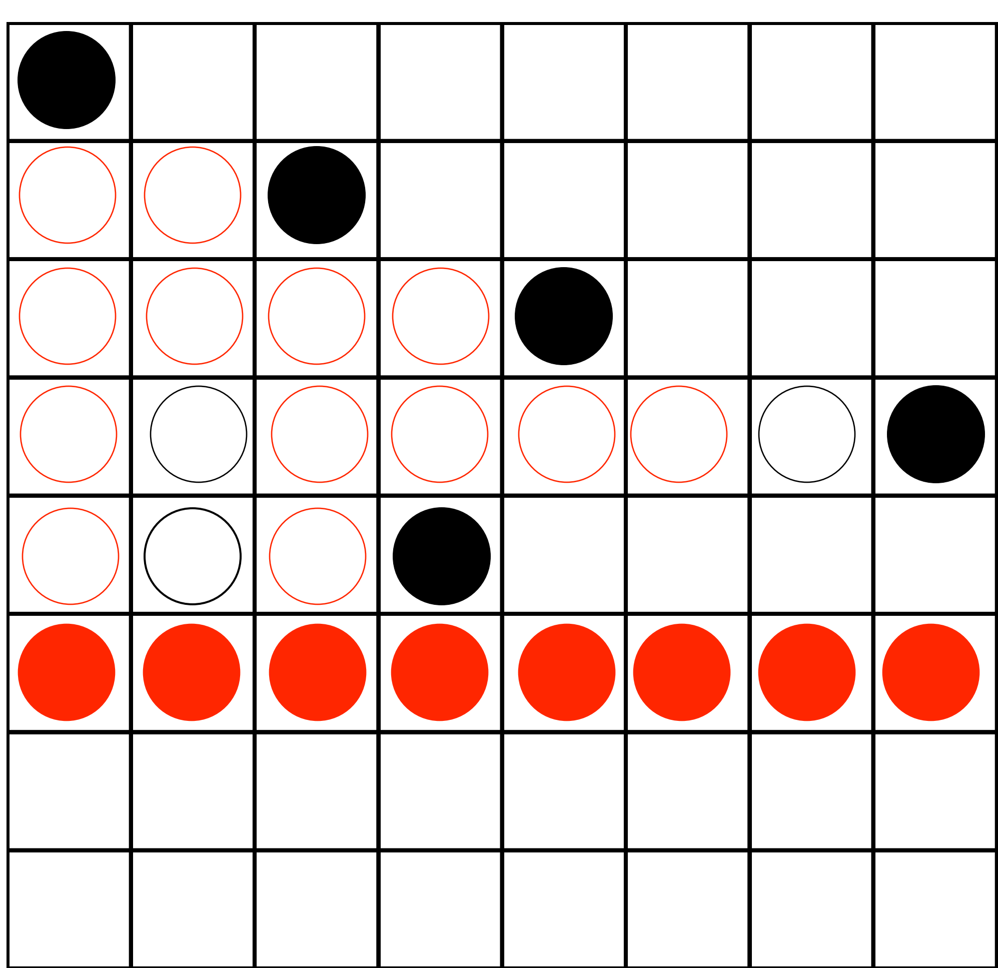


Tests 286+2+4= 292 **Backtracks 11**

Backtracking

Q6 Fails

Backtracks to Q5

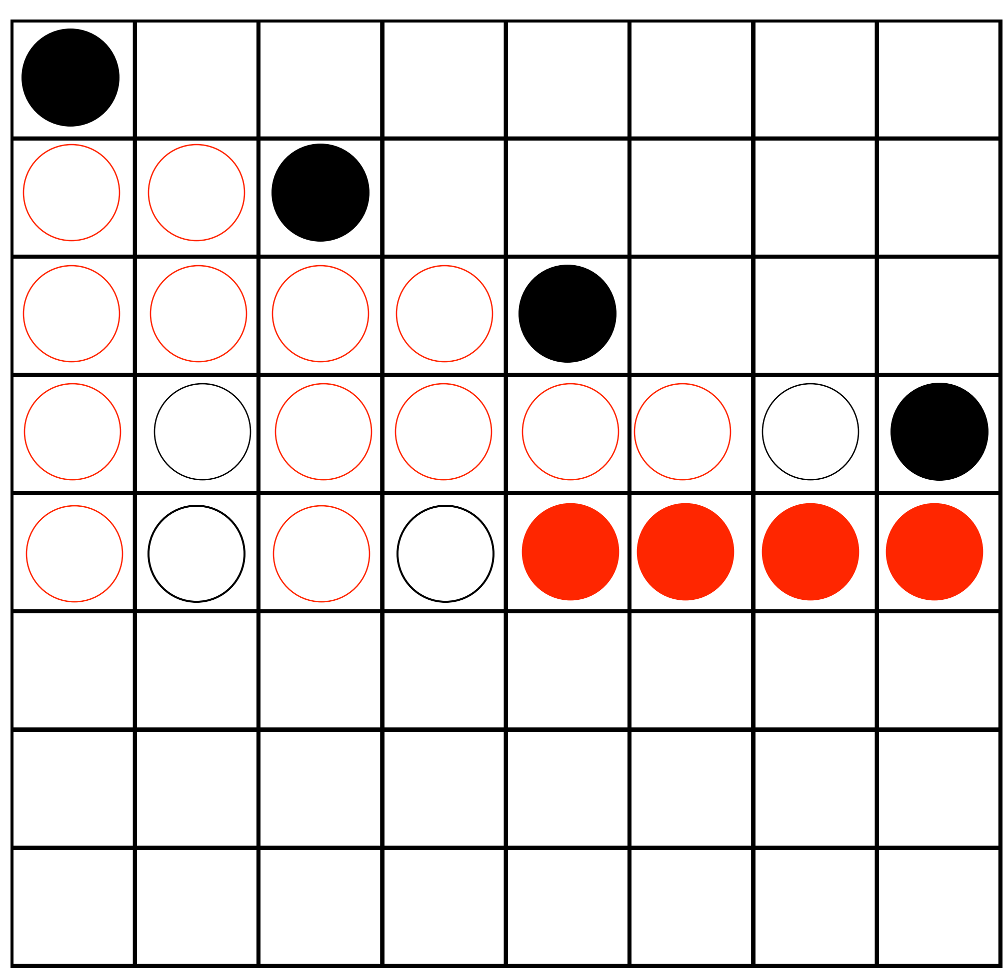


Tests 292+1+3+2+5+3+1+2+3= 312 Backtracks 11+1=12

Backtracking

Q5 Fails

Backtracks to Q4 and next to Q3



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

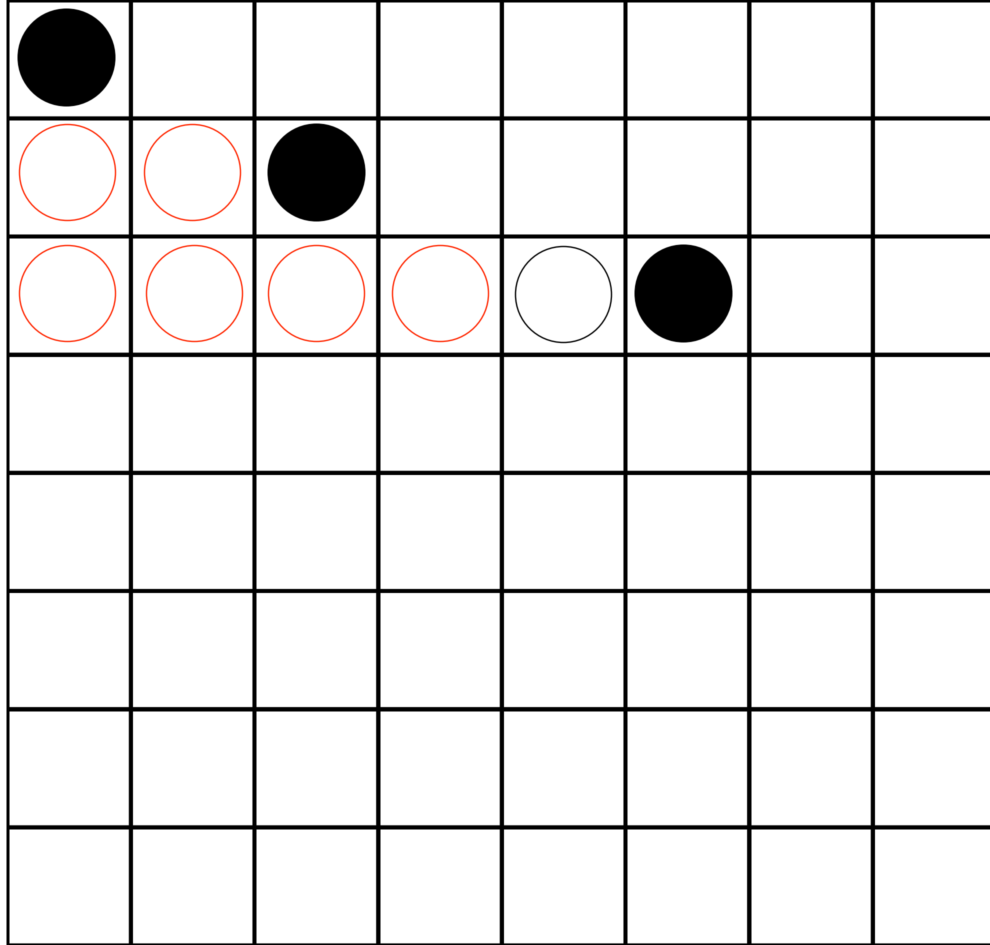
Backtracking

$Q_1 = 1$

$Q_2 = 3$

$Q_3 = 5$

Impossible !

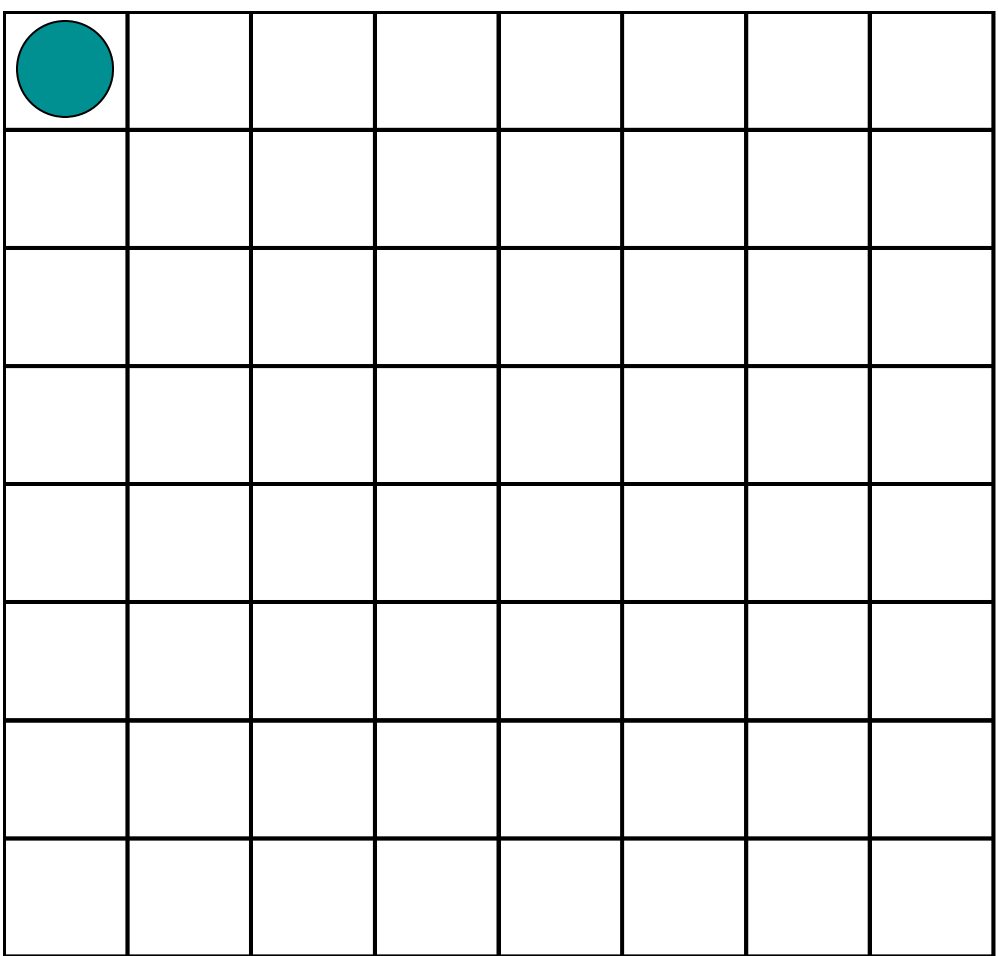


Tests 322 + 2 = 324 **Backtracks 14**

Search Methods – Backtracking + Propagation

- A more efficient backtracking search strategy sees constraints as **active** constructs.
- Whenever a variable is assigned to 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
  - First, values are tested to check their possible use.
  - Second , the values are adopted for 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 - Backtracking + Propagation



Tests 0

Backtracks 0

Search Methods - 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 - 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**

Search Methods - 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 + 4 * 5 = 112 **Backtracks 0**

Search Methods - Heuristics

- In both types of backtrack search (pure backtracking and 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 steps (i.e. taking n decisions).
- Of course, there are no such perfect heuristics for non-trivial problems (this would imply P = NP, a quite unlikely result), but good heuristics can nonetheless significantly decrease the search space. Typically a heuristics require
  - **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 some heuristics to be used, that are not available in pute backtrack search methods.*
- In particular a very simple heuristics is often very useful: Whenever a variable is restricted to take a single value, select that variable and value.
- This procedure is illustrated again with the 8-queens problem.

Search Methods - 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 + 4 * 5 = 112 **Backtracks 0**

Search Methods – 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 112+3+3+3+4 = 125 **Backtracks 0**

Search Methods - B+P w/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 125

Backtracks 0

Search Methods - 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 125 **Backtracks 0**

Search Methods - 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 125+2+2+2=131 **Backtracks 0**

Search Methods - 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 131 **Backtracks 0**

Search Methods – 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 131 **Backtracks 0**

Search Methods – 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 131+2+2=135 **Backtracks 0**

Search Methods – 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 - B+P w/Heuristics

<table>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>1</td><td>1</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>1</td><td>2</td><td>1</td><td>2</td><td></td><td></td><td></td><td></td></tr>
<tr><td>1</td><td>6</td><td>2</td><td>1</td><td>2</td><td>3</td><td>8</td><td></td></tr>
<tr><td>1</td><td></td><td>2</td><td>6</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td>1</td><td>3</td><td>2</td><td></td><td>3</td><td>1</td><td>2</td><td>3</td></tr>
<tr><td>1</td><td></td><td>2</td><td>6</td><td>3</td><td>8</td><td>1</td><td>2</td></tr>
<tr><td>1</td><td>6</td><td>2</td><td>2</td><td>3</td><td>6</td><td></td><td>1</td></tr>
</table>

Tests 135 **Backtracks 0**

Search Methods - 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 – 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 - B+P w/Heuristics

Q_7

may take NO value

Failure!

Backtracks

to Q_7

●							
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 - B+P w/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

Search Methods – Backtracking + Propagation

- 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 none to 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.

Constraint Programming

Constraint Programming 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 possile, i.e. With the minimum possible use of resources (time and space).

Declarative Programming

- Programming declaratively a combinatorial problem thus requires
  - the specification of the constraints of the problem
  - The specification of a search algorithm
- The separation of these two aspects has for a long time been advocated by several programming paradigms, namely functional programming and logic programming.
- Logic programming (LP) has a built in mechanism for search (backtracking) and has been extended to constraint logic programming (CLP). All that is required is to extend its underlying **resolution** to **constraint propagation** (CHiP, ECLiPse, SICStus).
- Alternatively, specialised object-oriented languages (COMET, ZINC) have been recently proposed to ease some modeling issues, namely the specification of data structures such as multi-dimensional arrays, as well as interfacing with other search methods (e.g. local search).

An Example – N queens

- A declarative specification (in COMET):

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

Solver<CP> cp();                    // declaration of a CP solver
var<CP>{int} q[i in S](cp,S);    // … and decision variables

solve<cp> {
  forall(i in 1..n-1, j in i+1..n){
    cp.post(q[i] != q[j]);     // no 2 queens in the same column
    cp.post(q[i]+i != q[j]+j); // no 2 queens in the same / diag
    cp.post(q[i]-i != q[j]-j); // no 2 queens in the same \ diag
  }
}
using {
  labelFF(q);                 // separate search specification
}
```

Constraint Programming

Although the expressivity and declarativity of Constraint Programming languages has driven significant research, most research in this area is being devoted to make it **efficient**.

This topic has been addressed through many different research directions:

- What **kind of consistency** is maintained?
- What **kinds of constraints** are used?
- Would **redundancy** help?
- What **heuristics** to adopt?
- Are there **symmetries** to break?