Constraint Programming

- An overview

- Examples of decision (making) problems
- Complexity and need for search
- Declarative Modelling with Constraints
  - Constraint Programming
- Finite and Continuous Domains

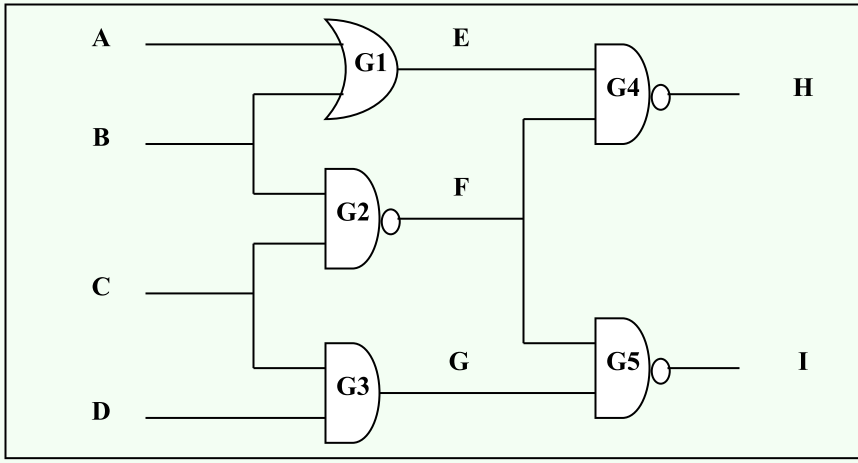
Constraint Problems: Examples

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

Modeling of Digital Circuits

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

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



```
E = or(A,B)     % G1
F = nand(B,C)   % G2
G = and(B,C)    % G3
H = nand(E,F)   % G4
I = nand(F,G)   % G5
```

Production Planning

Goal (Example): Determine a production plan

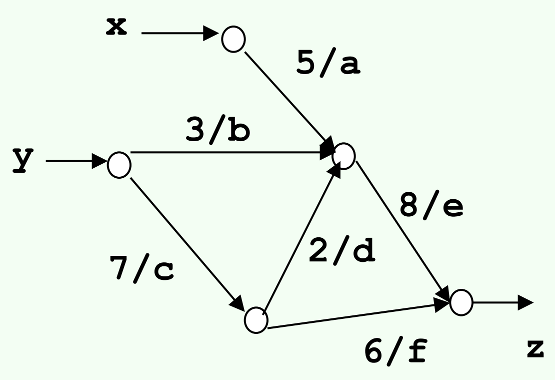
- Variables:
  - Quantities of goods to produce
- Domain:
  - Rational/Reals or Integers
- Constraints:
  - Equality and Inequality (linear) constraints to model resource limitations, minimal quantities to produce, costs not to exceed, balance conditions, etc...

```
Find x, y and z such that
 4x+ 3y + 6z ≤ 1500    % resources used do not exceed 1500
  x + y + z >= 300     % production not less than 300 units
  x ≤ z + 20           % x units within z ± 20 units
  x ≥ z - 20
  x, y, z ≥ 0          % non negative production
```

Network Management

Goal (Example): Determine acceptable traffic on a netwok

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



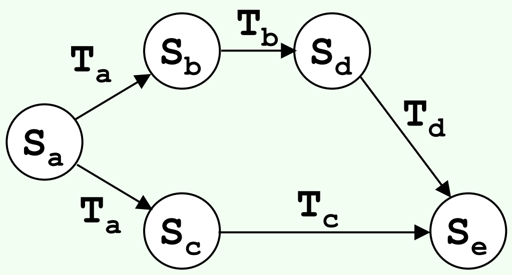
```
Find x,y,z, a,b,c,d,e such that
   x ≥ 6, z ≥ 10          % minimum flow
   a ≤ 5, ... , f ≤ 6     % capacity
% flow maintenance
  x = a, y = b + c, a + b + d = e,
  c = d + f, e + f = z
  x, y ,z, a, b, c, d, e, f ≥ 0
```

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...

$$\begin{tikzcd} & S_b \arrow[r, "T_b"] & S_d \arrow[dr, "T_d"] & \\ S_a \arrow[ur, "T_a"] \arrow[dr, "T_a"'] & & & S_e \\ & S_c \arrow[urr, "T_c"'] & & \end{tikzcd}$$



Find S_a ,..., S_e, such that
  $S_b \geq S_a+T_a$,   % precedence
    ....
  $(S_c \geq S_b+T_b) \vee (S_b \geq S_c+T_c)$ % non overlap
    ...,
  6 ≥ Sa+Ta   % deadline
  S_a,..., $S_e \geq 0$

Assignment

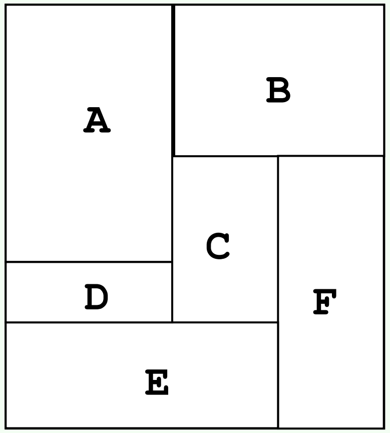
Many constraint problems can be classified as assignment problems. In general all that can be stated is that these problems follow a general CSP goal :

Assign values to the variables to satisfy the relevant constraints.

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

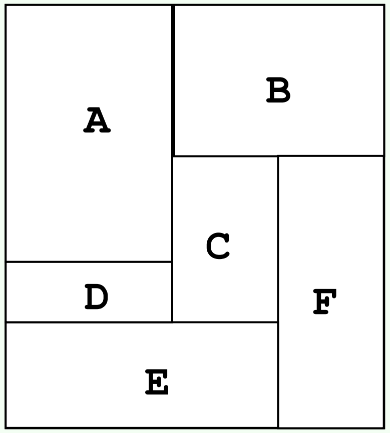
Some examples may help to illustrate this class of problems

Assignment (2)



Graph Colouring (Finite Domains)

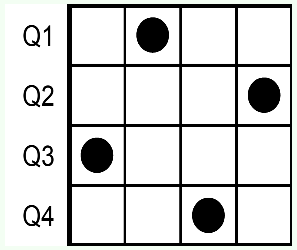
```
Assign values to A, .., F,
  s.t.  A, B, .., F ∈ {red, blue, green}
        A ≠ B, A ≠ C, A ≠ D,
        B ≠ C, B ≠ F, C ≠ D, C ≠ E, C ≠ F
        D ≠ E, E ≠ F
```



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

```
Assign values to A1,A2, .., F1,F2
  s.t.  Ar, Ab, Ag, .., Fr, Fb, Fg ∈ {0,1}
% one and only one colour for A, B, ..., F
        Ar + Ab + Ag = 1;
        ....
% different colours for A and B, ...
Ar + Br <= 1; Ab + Bb <= 1; Ag + Bg <= 1;
        ....
```

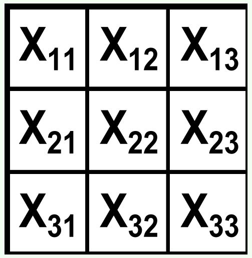
Assignment (3)



N-queens (Finite Domains):

Assign Values to Q1,..., Qn $\in$ {1,.., n}

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

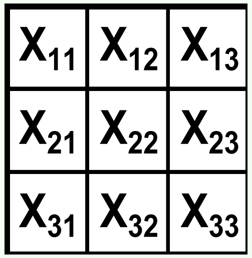

X_{11} X_{12} X_{13}
X_{21} X_{22} X_{23}
X_{31} X_{32} X_{33}

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

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

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

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


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

Magic Squares:

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

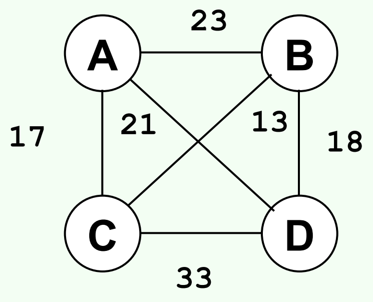
s.t. $\forall_i \; \forall_{j \neq i} \; \Sigma_k \; X_{ki} = \Sigma_k \; X_{kj} = M$ % same rows sum

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

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

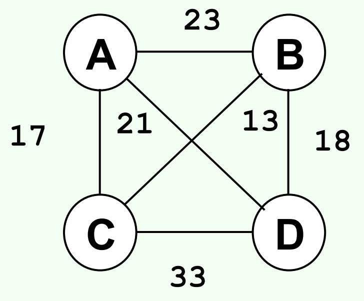
$\forall_{i \neq k} \vee \forall_{j \neq l} \; X_{ij} \neq X_{kl}$ % all different

Assignment (3)



Travelling Salesperson (Finite Domains)

```
Find values for A, B, C, D ∈ {1,..,4}
    s.t. A ≠ B, ..., C ≠ D
         % a permutation of [A,B,C,D]
         if A = B+1 then X_A = L_ba,
         ...
         if D = C+1 then X_D = L_cd
         X_A + X_B + X_C + X_D  ≤ k
```



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

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

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

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

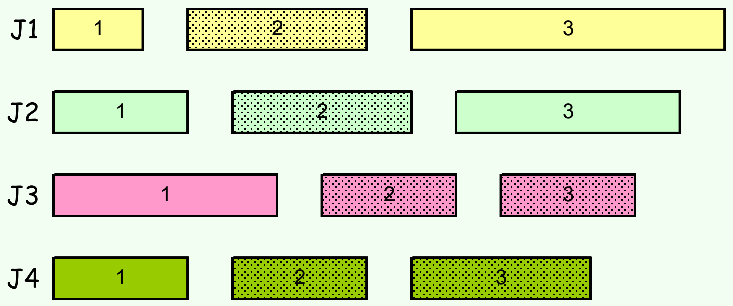
... no subcycle constraints

$\Sigma_a \Sigma_b X_{ab} L_{ab} < k$

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)



Job-Shop

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

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

% precedence within job

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

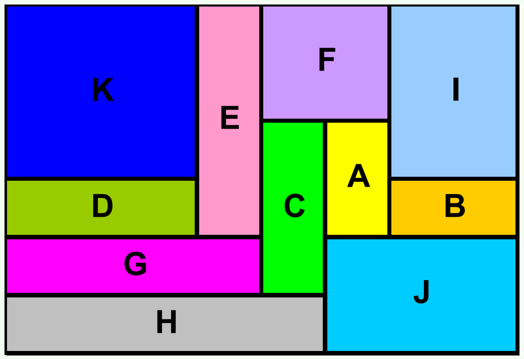
% either no-overlap or different machines

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

Filling and Containment

Goal (Example): Assign values to variables

- Variables:
  - Point Locations
- Domain:
  - Integers (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
        $\forall_{i,j}$    $(X_i+Lx_i \le X_j)$   % I to the left of J
                $(X_j+Lx_j \le X_i)$   % I to the right of J
                $(Y_i+Ly_i \le Y_j)$   % I in front of J
                $(Y_j+Lx_j \le X_i)$   % I in back of J
```

Constraint Satisfaction Problems

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

Constraint Satisfaction Problems

- Formally a constraint satisfaction problem (CSP) can be regarded as a tuple <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

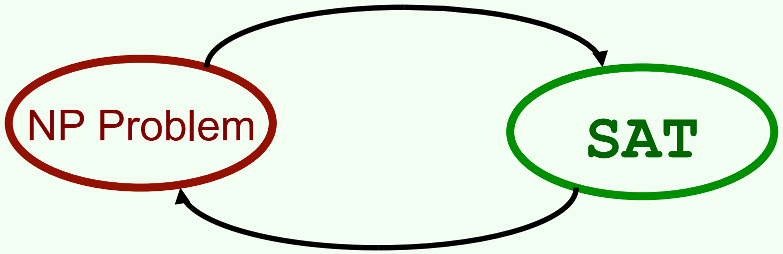
- In many cases, one is interested not only in satisfying some set of constraints but also in finding among all solutions those that optimise a certain objective function (minimising a cost or maximising some positive feature).

- Formally a constraint (satisfaction and) optimisation problem (CSOP or COP) can be regarded as a tuple <V, D, C, F>, 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)
    - F is a function on the variables

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

Decision Problems are NP-complete

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



Decision Problems are NP-complete

- No one has already found a polynomial algorithm to solve SAT (or any other NP problem), and hence the conjecture P ≠ 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).

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

Decision Problems are NP-complete

- Still, constraint solving problems are NP-complete problems (as SAT is).
- If a non-deterministic machine (that guesses correctly) can solve a problem in polynomial time, then a real deterministic machine can check in 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.

Search Strategies

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

Complete Backtrack Search Methods:

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

Incomplete Local Search Methods:

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

Optimisation Problems are NP-hard

- Optimisation problems are typically NP-hard problems in that solving them is at least as difficult as solving the corresponding decision problem.
- In practice these problems cannot be solved in polynomial time by a non-deterministic machine, not can they be checked by a deterministic machine.
- In fact, to find an optimal solution it is not enough to find it ... It is necessary to show that it is better than all other solutions!
- Being harder than the decision problems, optimisation problems also require adequate search strategies, if larger instances are to be solved.
    - In complete search, detection of failure and subsequent backtracking may be imposed if the partial solution can be proved to be no better than one already found (branch & bound).

Constraint Programming

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

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

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

Declarative Programming

- Programming 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 in particular has a built-in mechanism for search (backtracking) that makes it easy to extend into constraint (logic) constraint programming, by “replacing” its underlying **resolution** to **constraint propagation.** A number of Constraint Logic Programming languages have been proposed (CHIP, ECLiPSE, GNU Prolog, SICStus) to explore this extension of logic programming.
- More recently, other declarative languages such as Comet (OO-like), Choco (Java Library) and Zinc, provide more convenient data structures for modelling, maintaining a declarative approach.

Constraint Programming

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

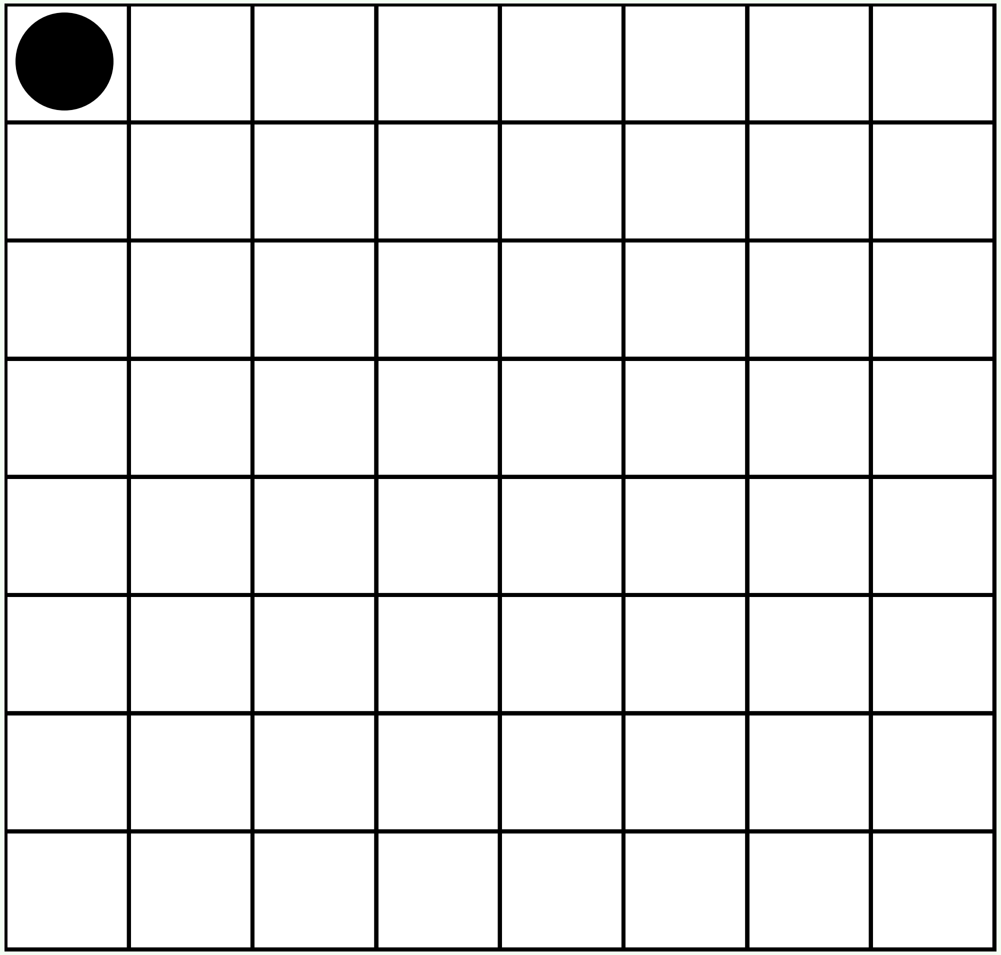
- Expressivity
    - Constraint Languages should be able to easily specify the variables, domains and constraints (e.g. conditional, global, etc...);
- Declarative Nature
    - Ideally, programs should specify the constraints to be solved, not the algorithms used to solve them
- Efficiency
    - Solutions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

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

Search Methods – Pure Backtracking

- The same specification can lead to different search strategies when sequentially assigning values to variables.
- The simplest backtracking strategy sees constraints in a passive form:
  - Whenever a variable is assigned a variable, the constraints whose variables are assigned variables are checked for satisfaction
  - If this is not the case, the search backtracks (chronological backtrack).
- This is a typical **generate and test** procedure
  - Firstly, values are generated
  - Secondly, the constraints are tested for satisfaction.
- Of course, tests should be done as soon as possible, i.e. a constraint is checked whenever all its variables are assigned values.
- This procedure is illustrated in the 8-queens problem.

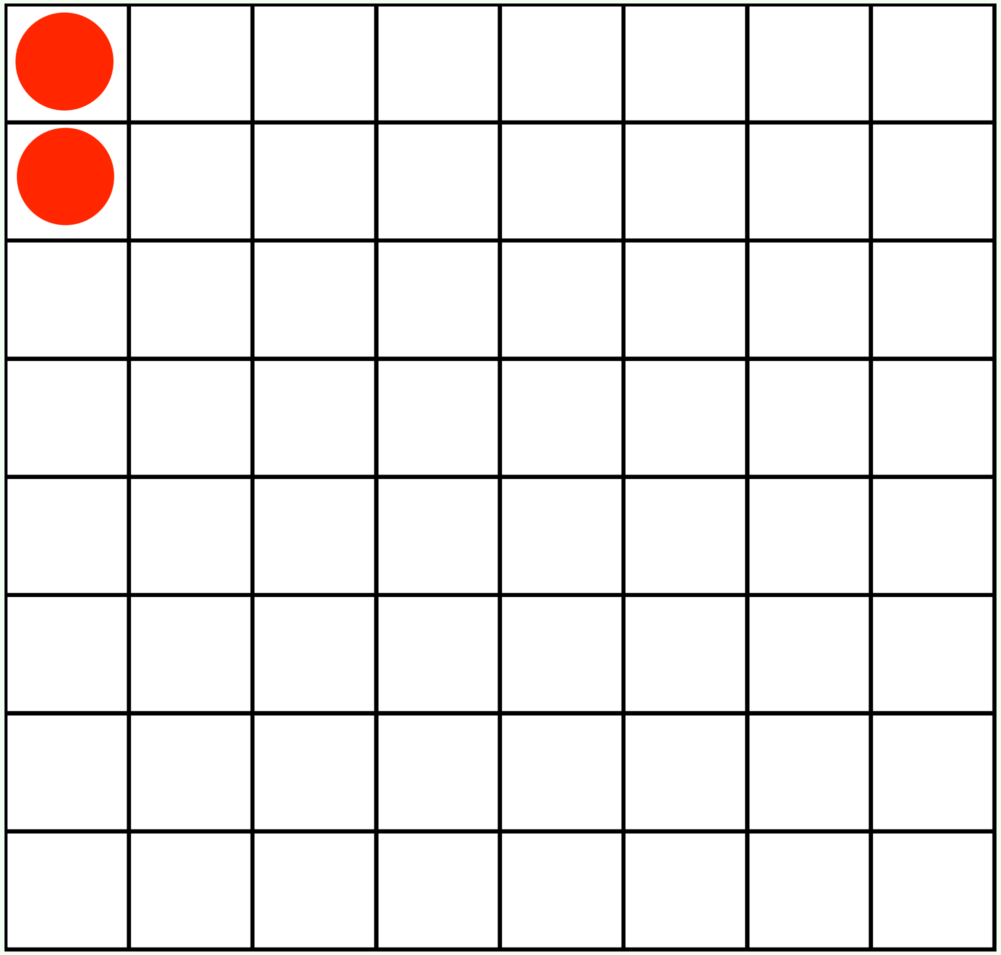
Backtracking



Tests 0 **Backtracks 0**

Backtracking

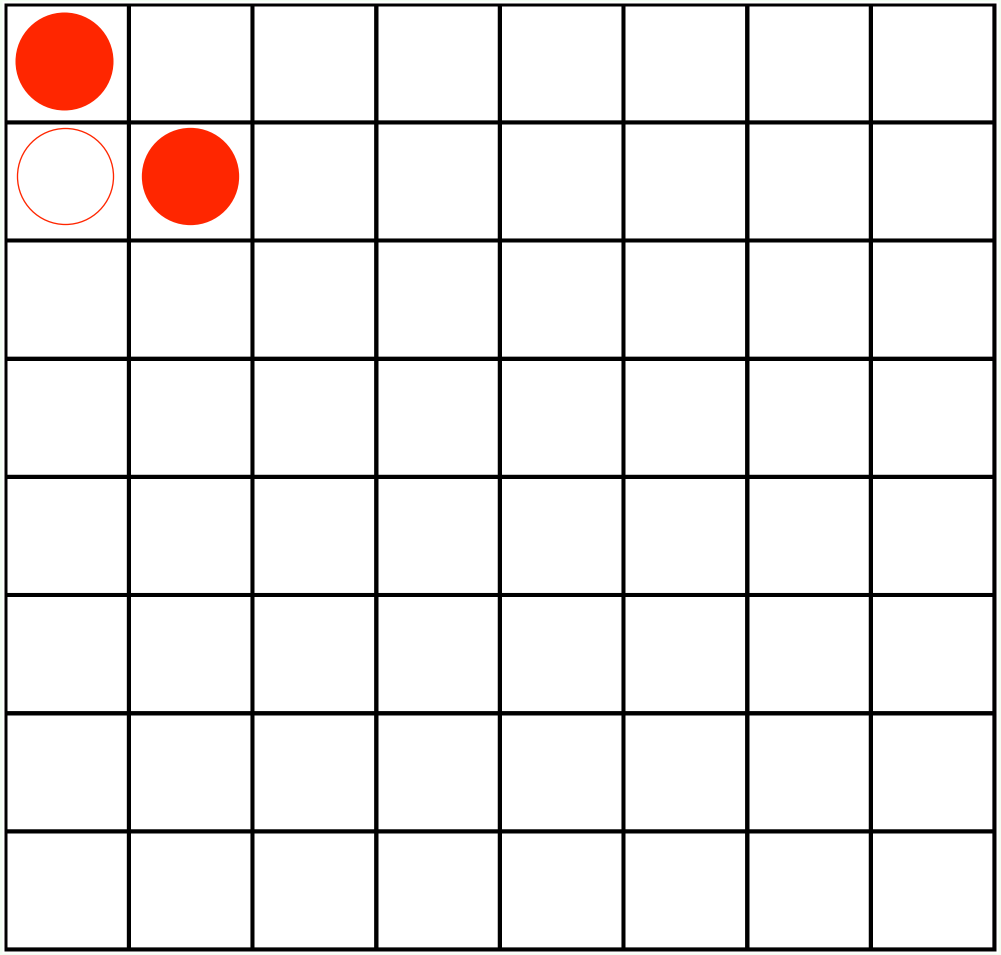
`Q1 \= Q2,  L1+Q1 \= L2+Q2,  L1+Q2 \= L2+Q1.`



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

Backtracking

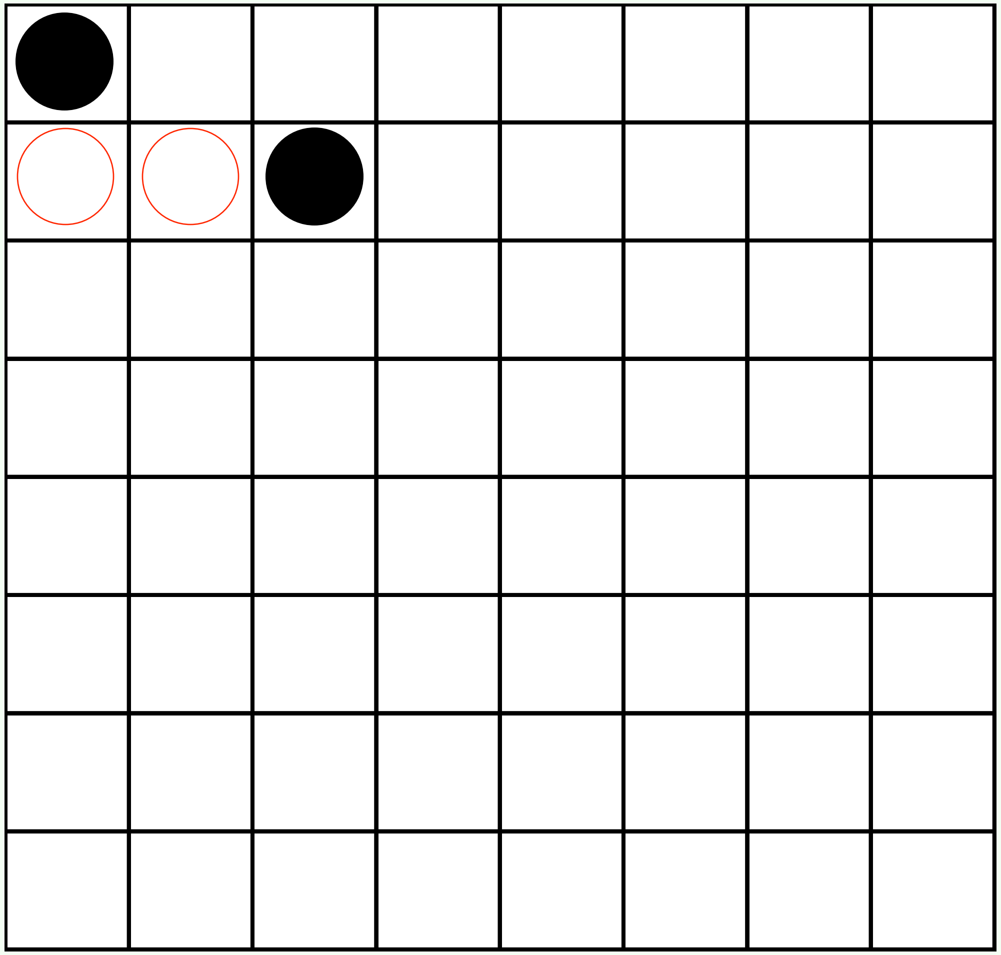
```
Q1 \= Q2,   L1+Q1 \= L2+Q2,   L1+Q2 \= L2+Q1.
```



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

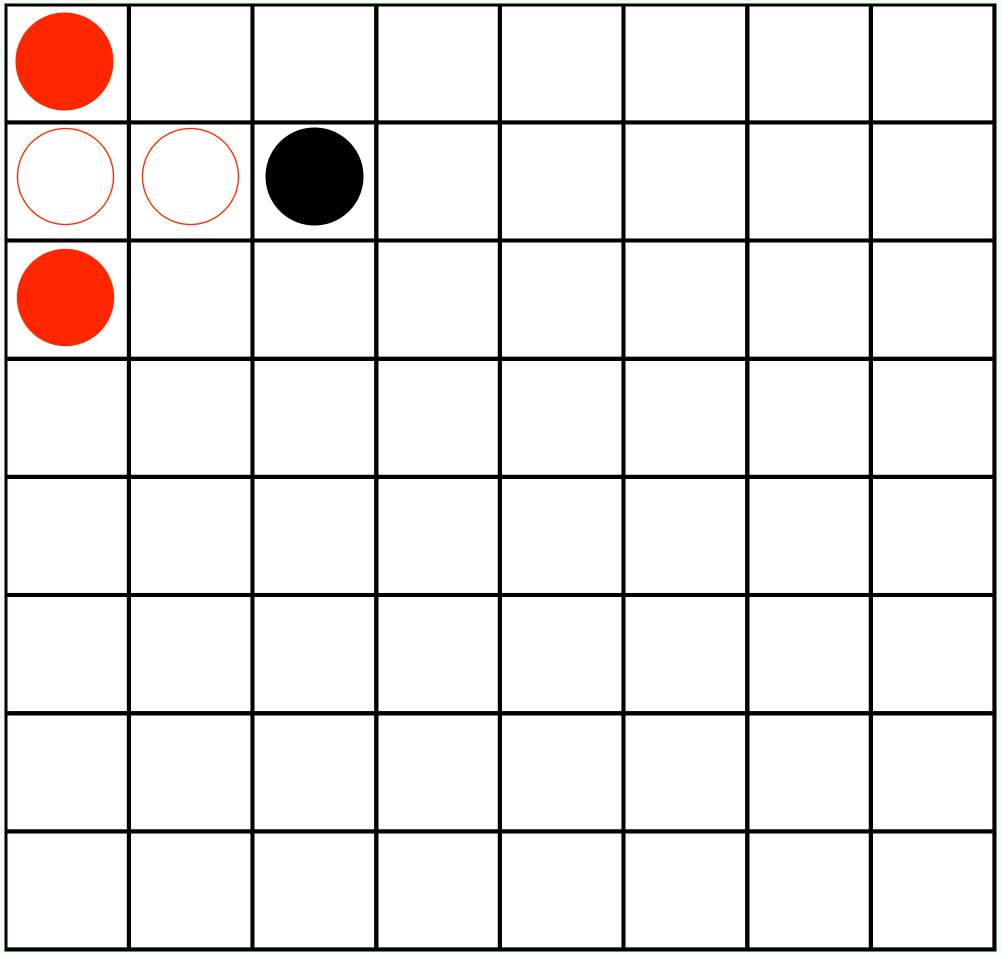
Backtracking

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



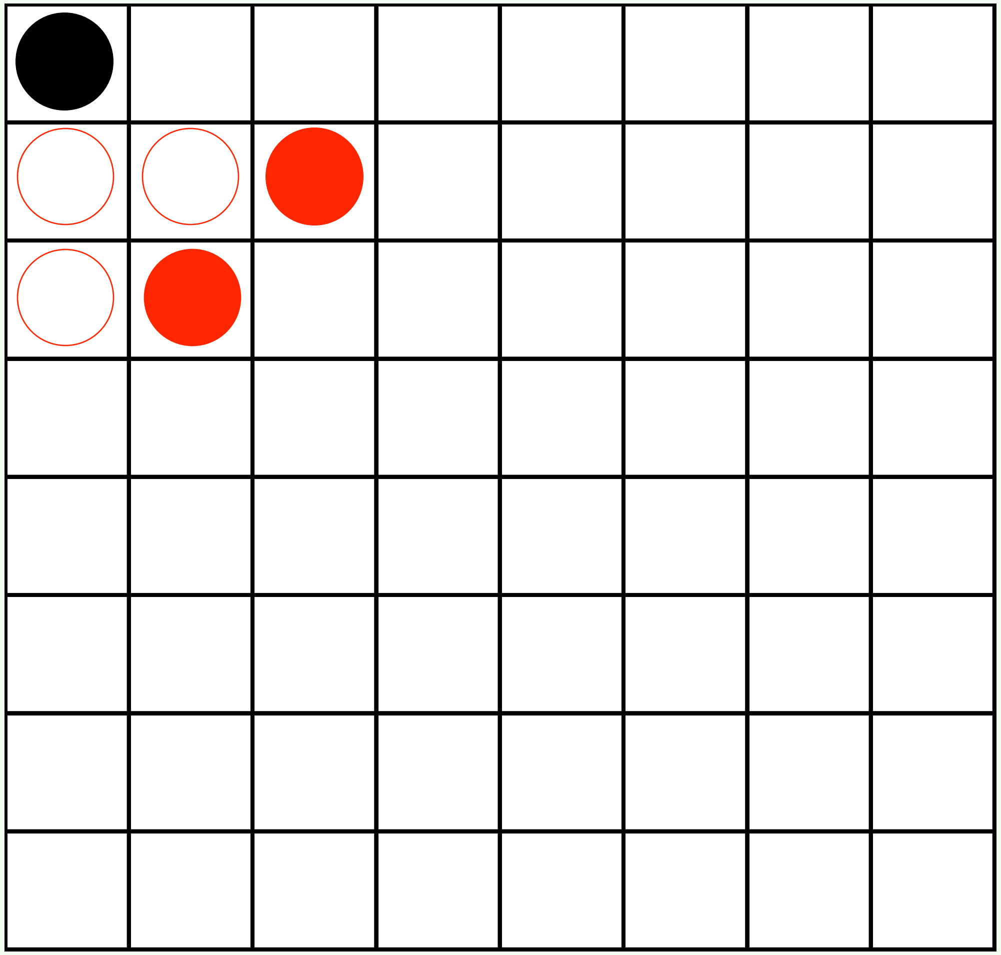
Tests 2 +1 = 3 **Backtracks 0**

Backtracking



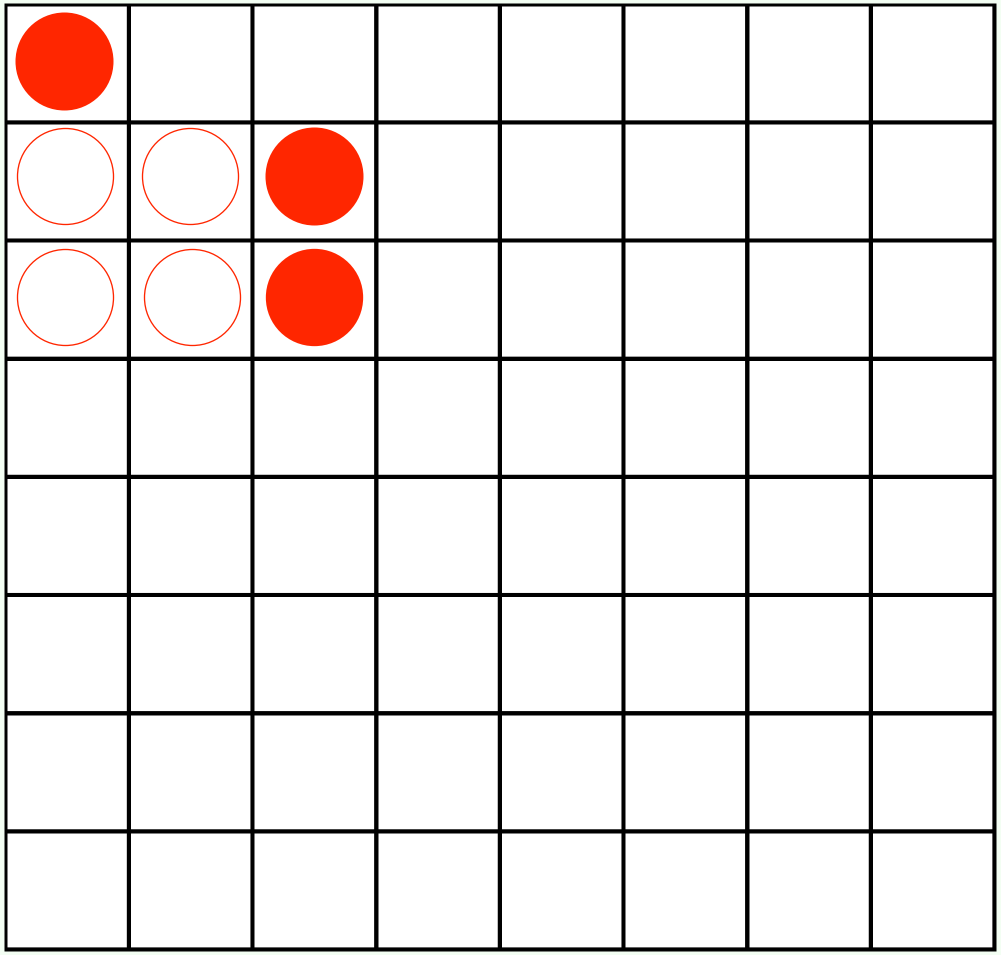
Tests 3 +1 = 4 **Backtracks 0**

Backtracking



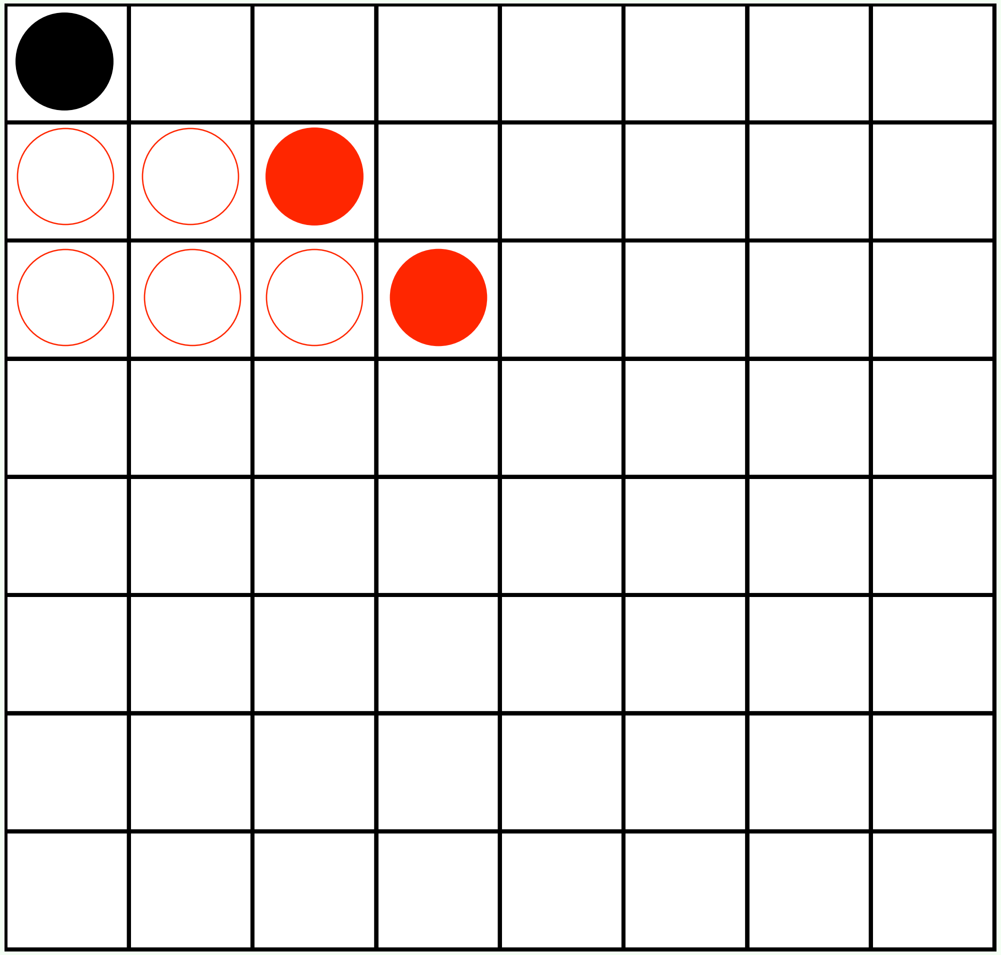
Tests 4 +2 = 6 **Backtracks 0**

Backtracking



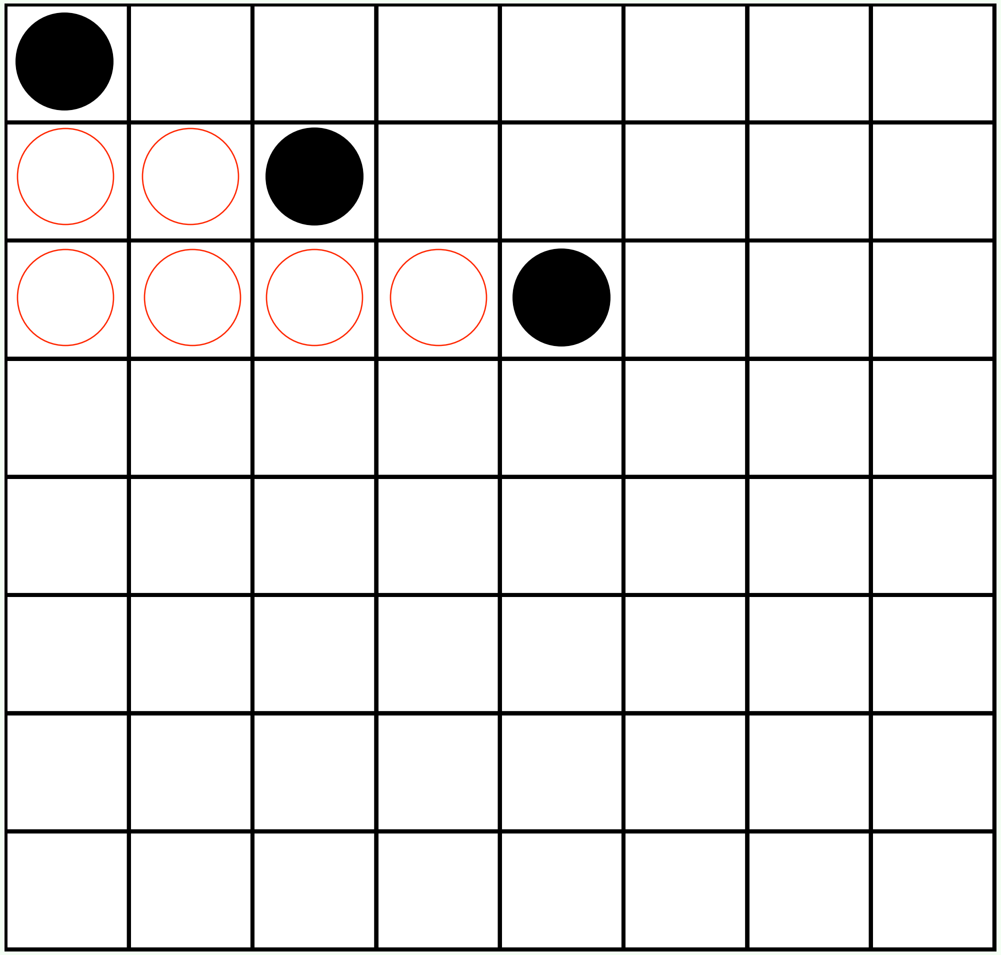
Tests 6 + 1 = 7 **Backtracks 0**

Backtracking



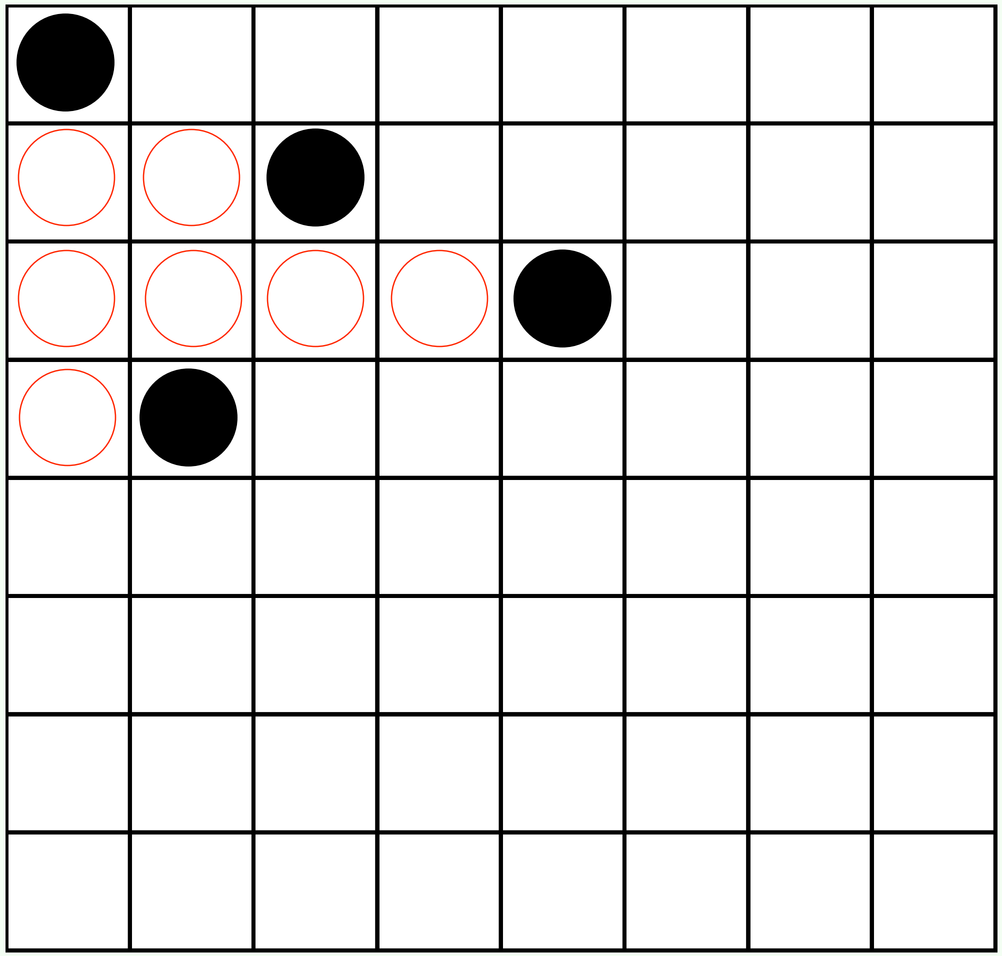
Tests 7 + 2 = 9 **Backtracks 0**

Backtracking



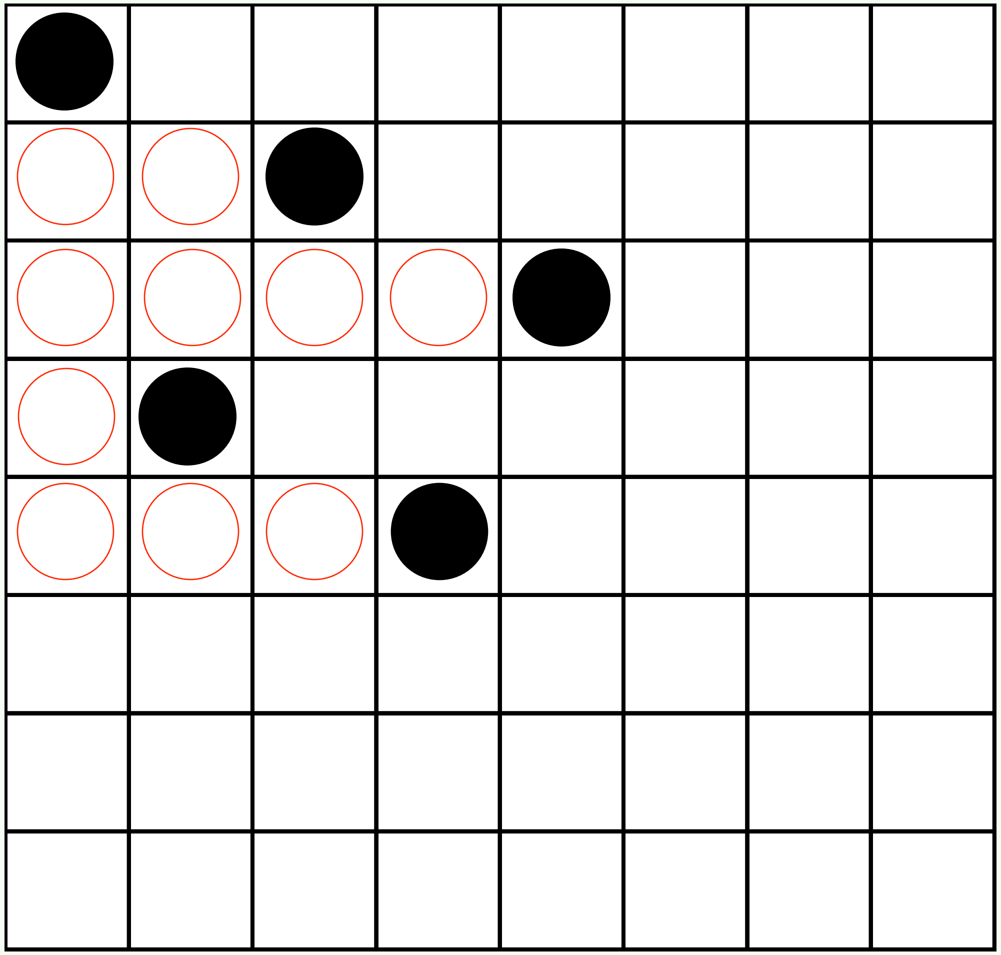
Tests 9 + 2 = 11 **Backtracks 0**

Backtracking



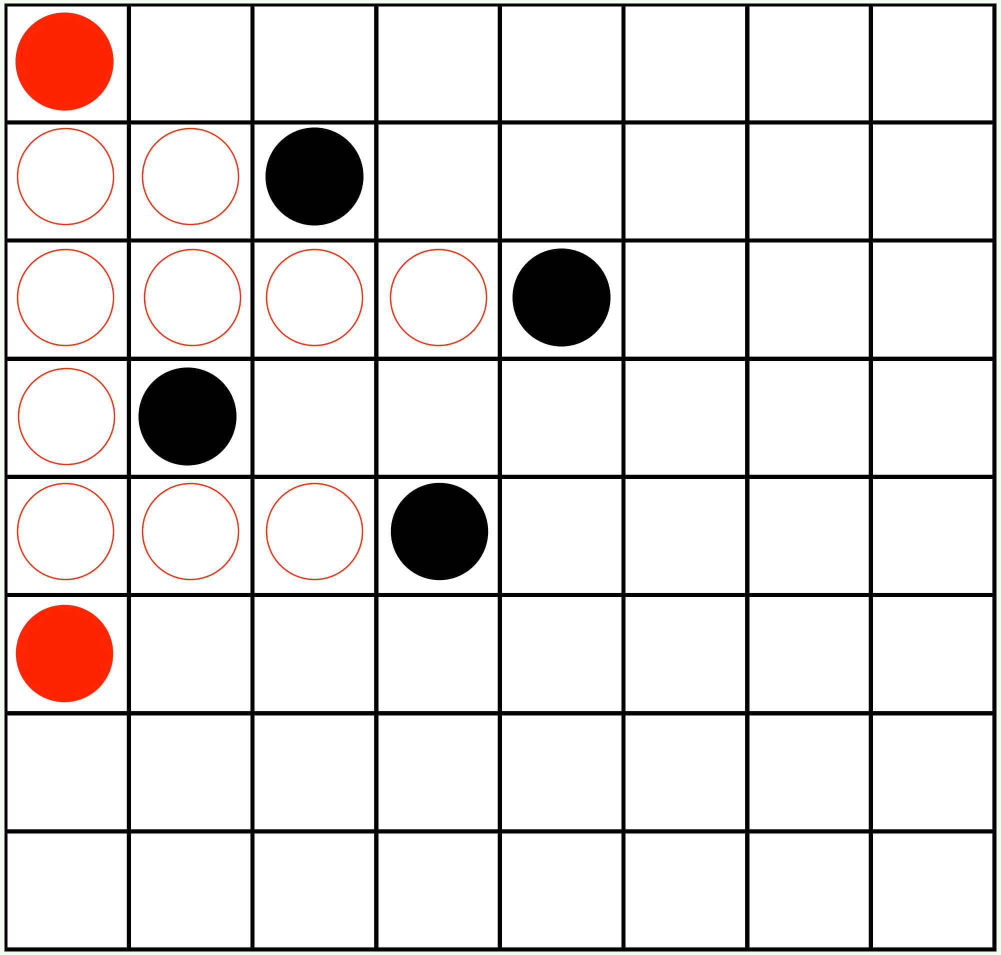
Tests 11 + 1 + 3 = 15 **Backtracks 0**

Backtracking



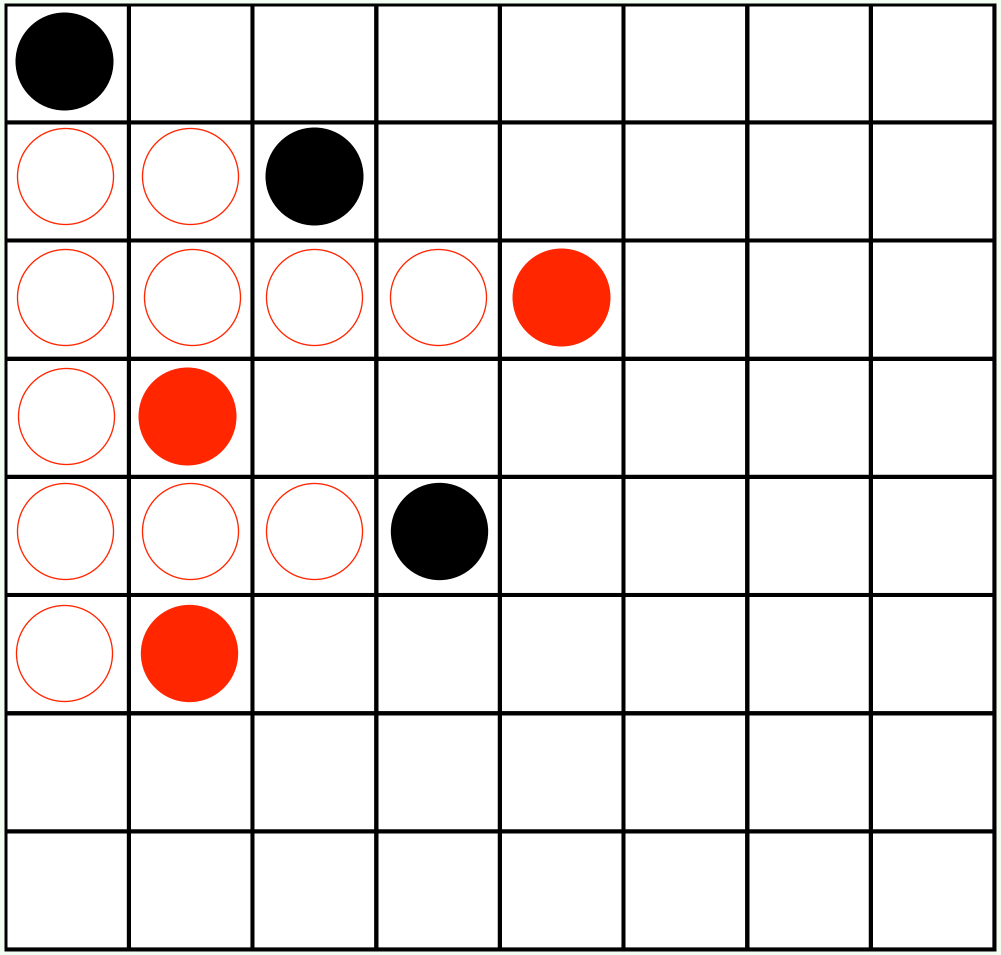
Tests 15+1+4+2+4 = 26 Backtracks 0

Backtracking



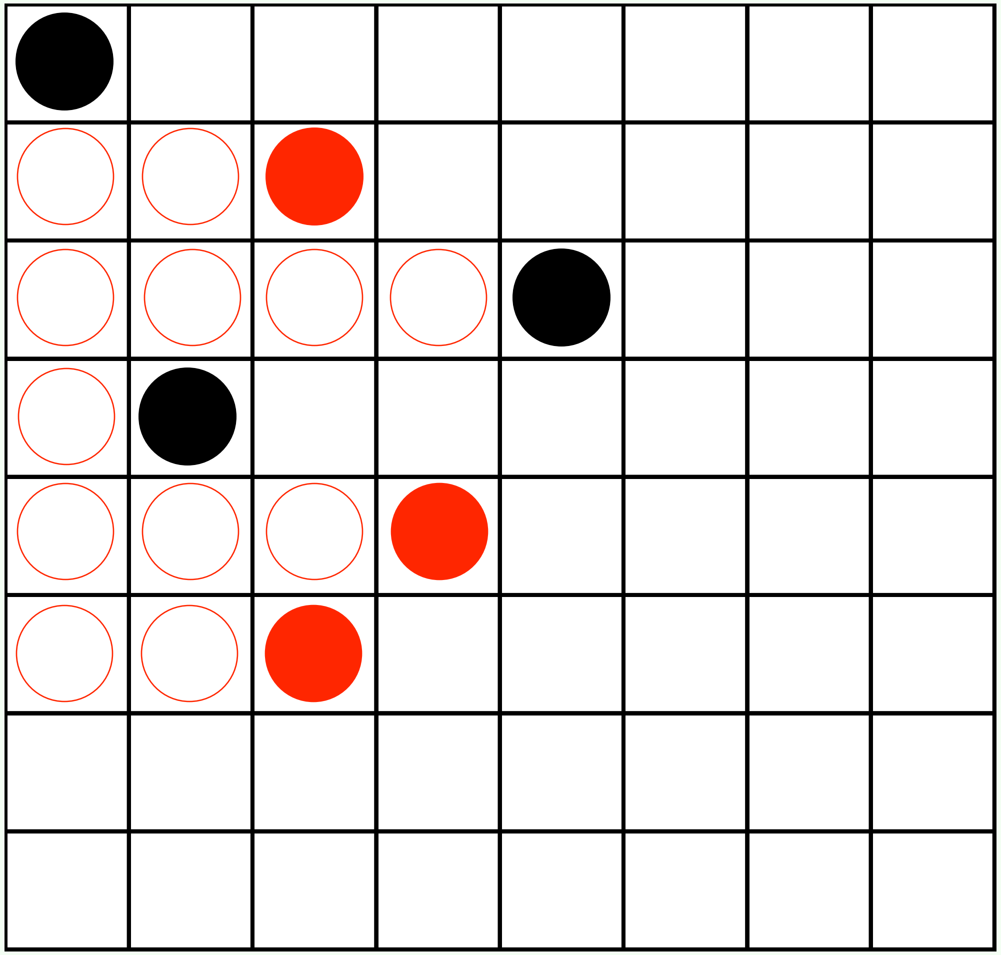
Tests 26+1 = 27 **Backtracks 0**

Backtracking



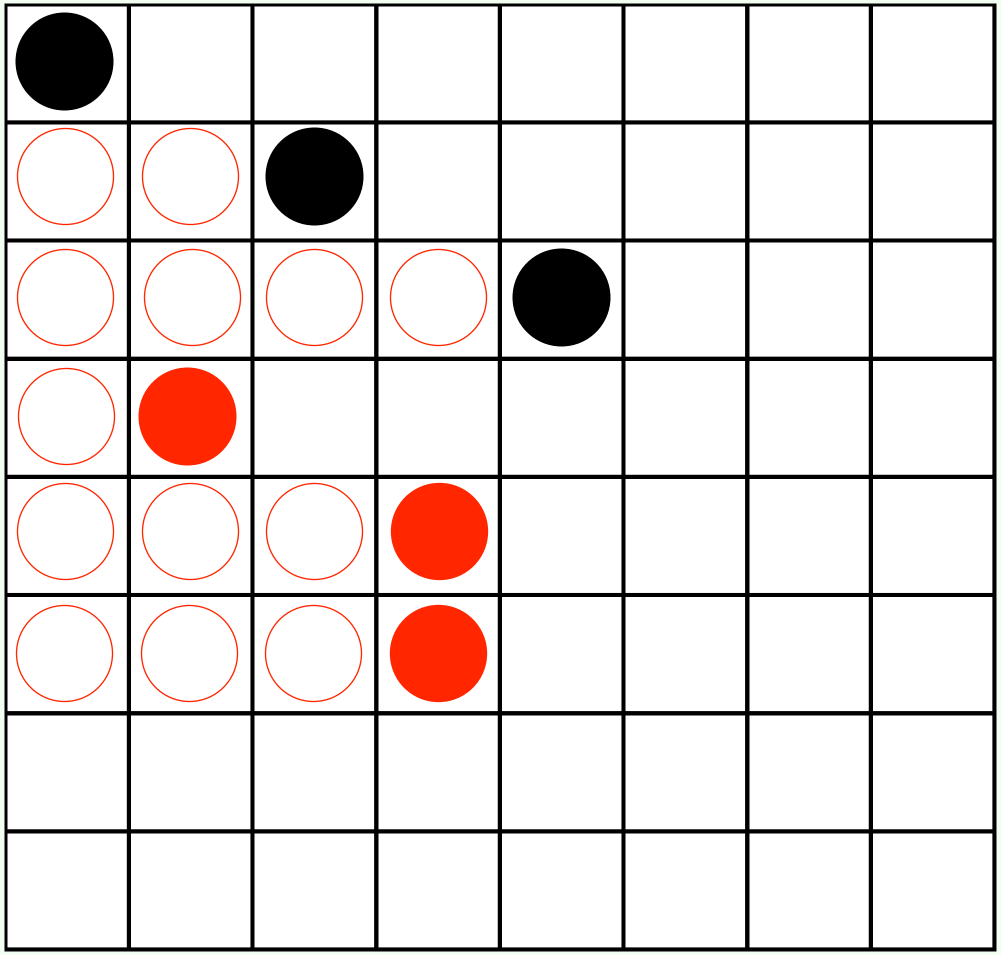
Tests 27 + 3 = 30 Backtracks 0

Backtracking



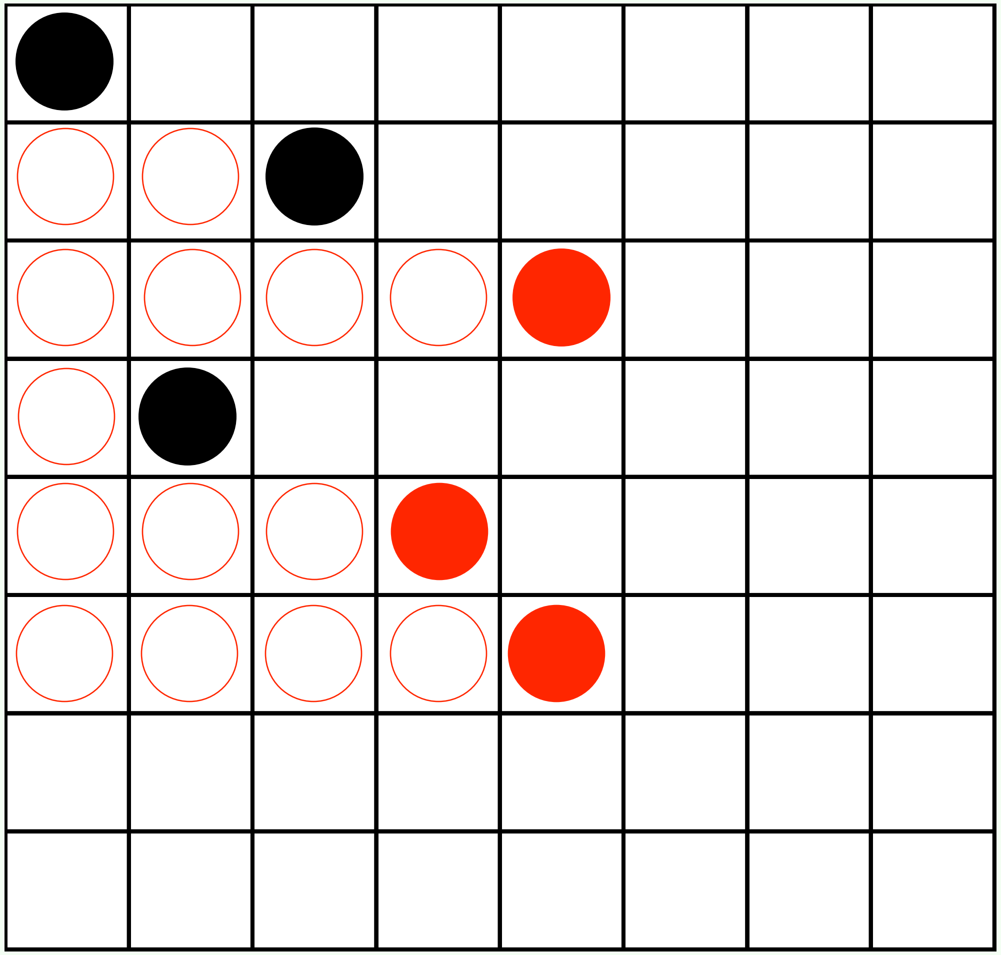
Tests 30+2 = 32 Backtracks 0

Backtracking



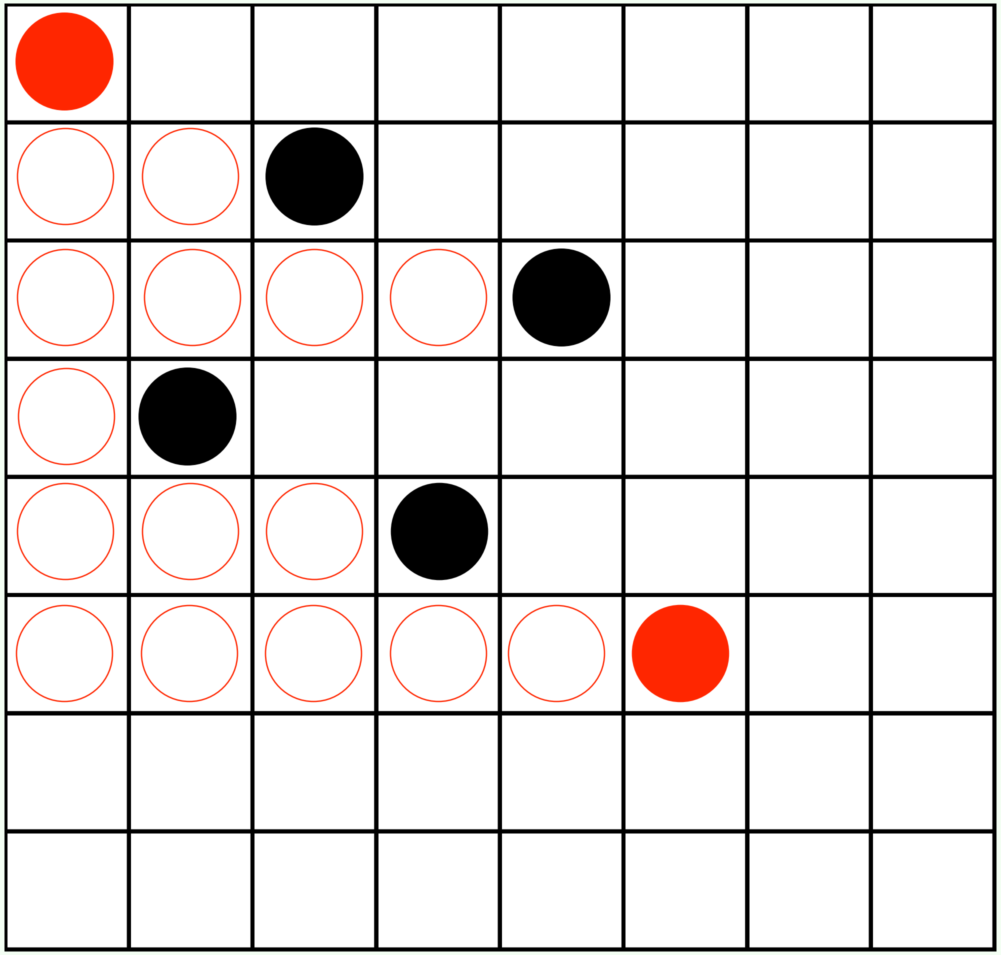
Tests 32 + 4 = 36   Backtracks 0

Backtracking



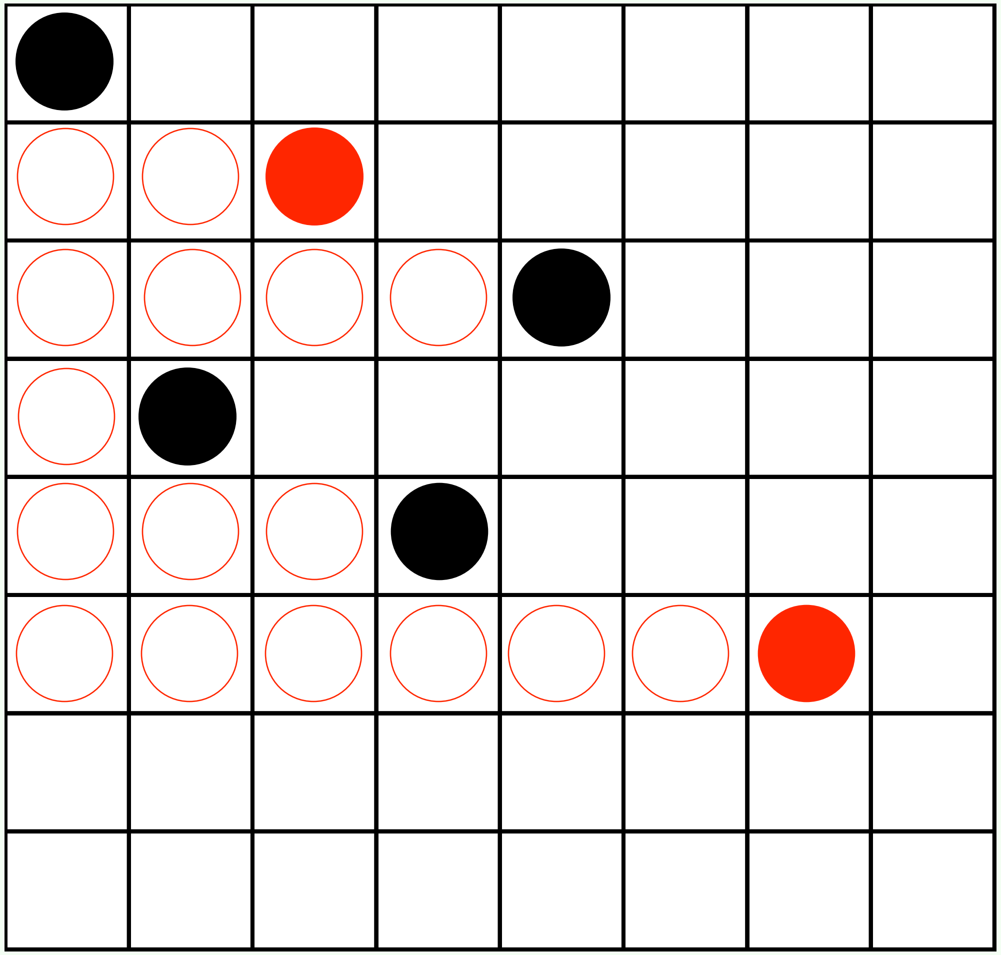
Tests 36 + 3 = 39 **Backtracks 0**

Backtracking



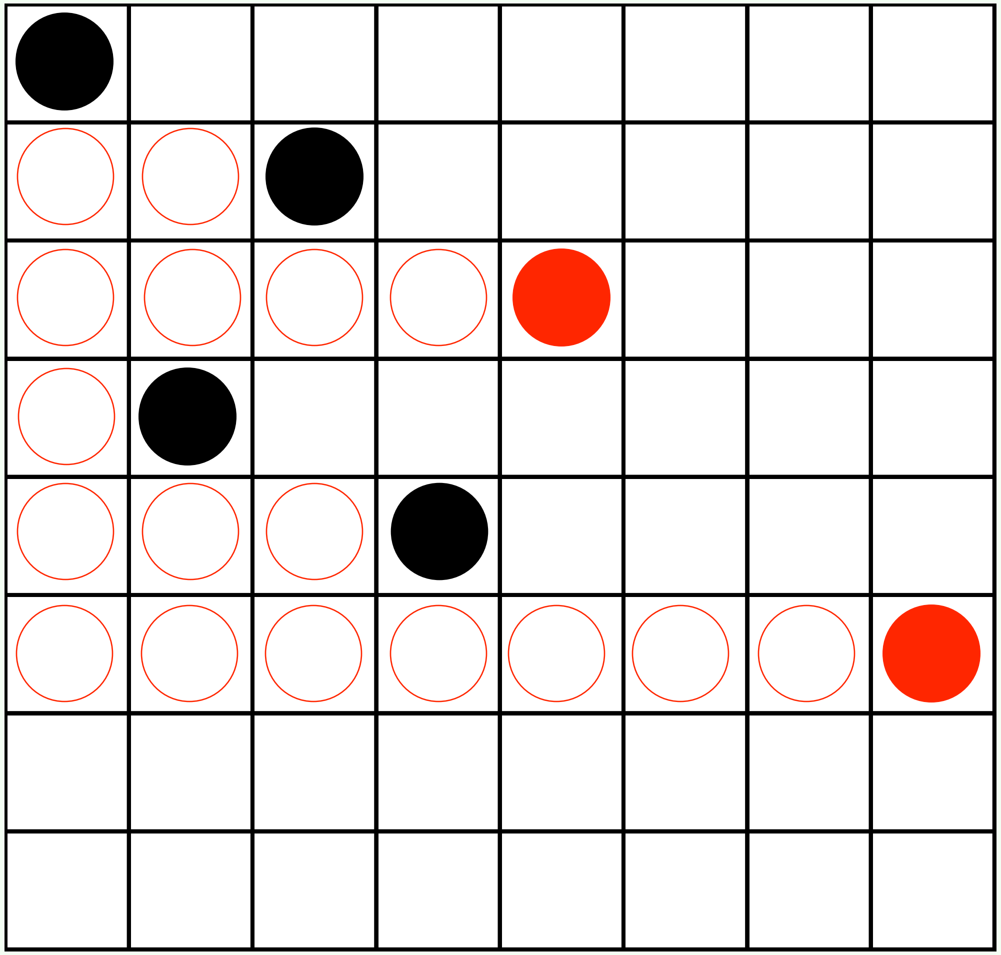
Tests 39 + 1 = 40 **Backtracks 0**

Backtracking



Tests 40 + 2 = 42 Backtracks 0

Backtracking

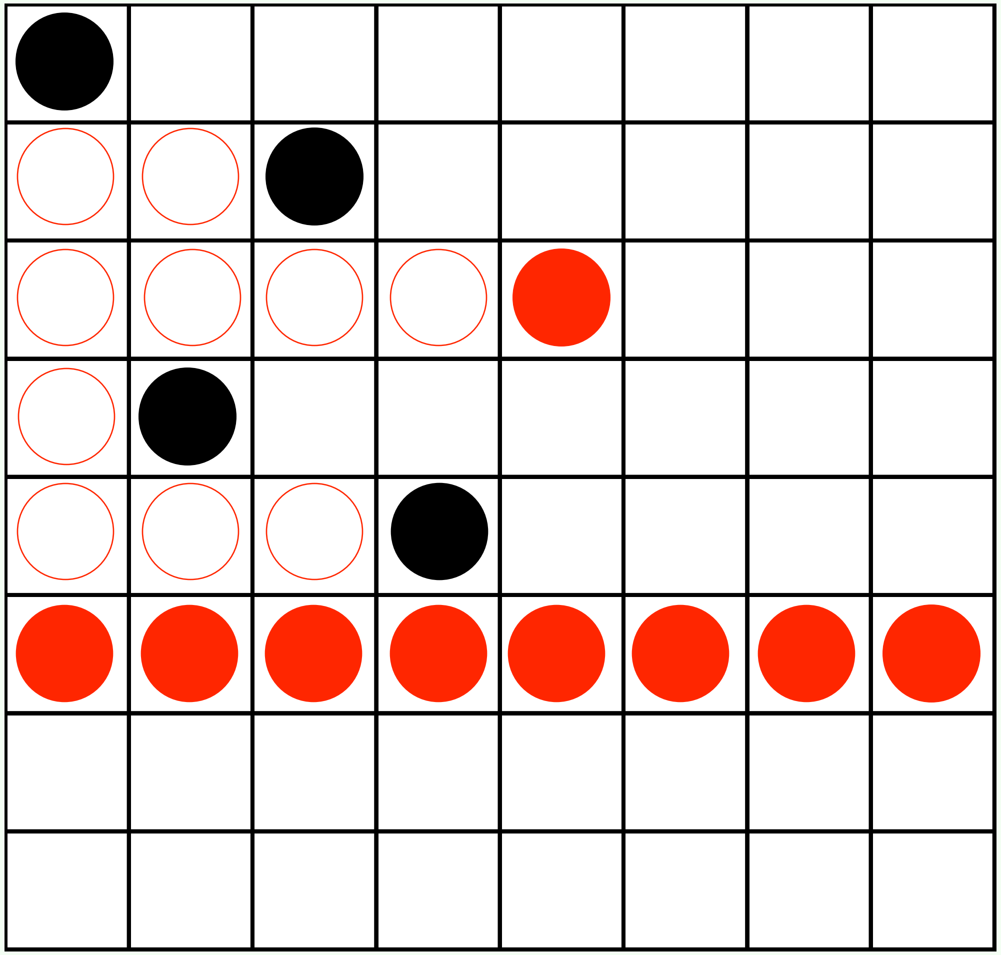


Tests 42 + 3 = 45 Backtracks 0

Backtracking

Q6 Fails

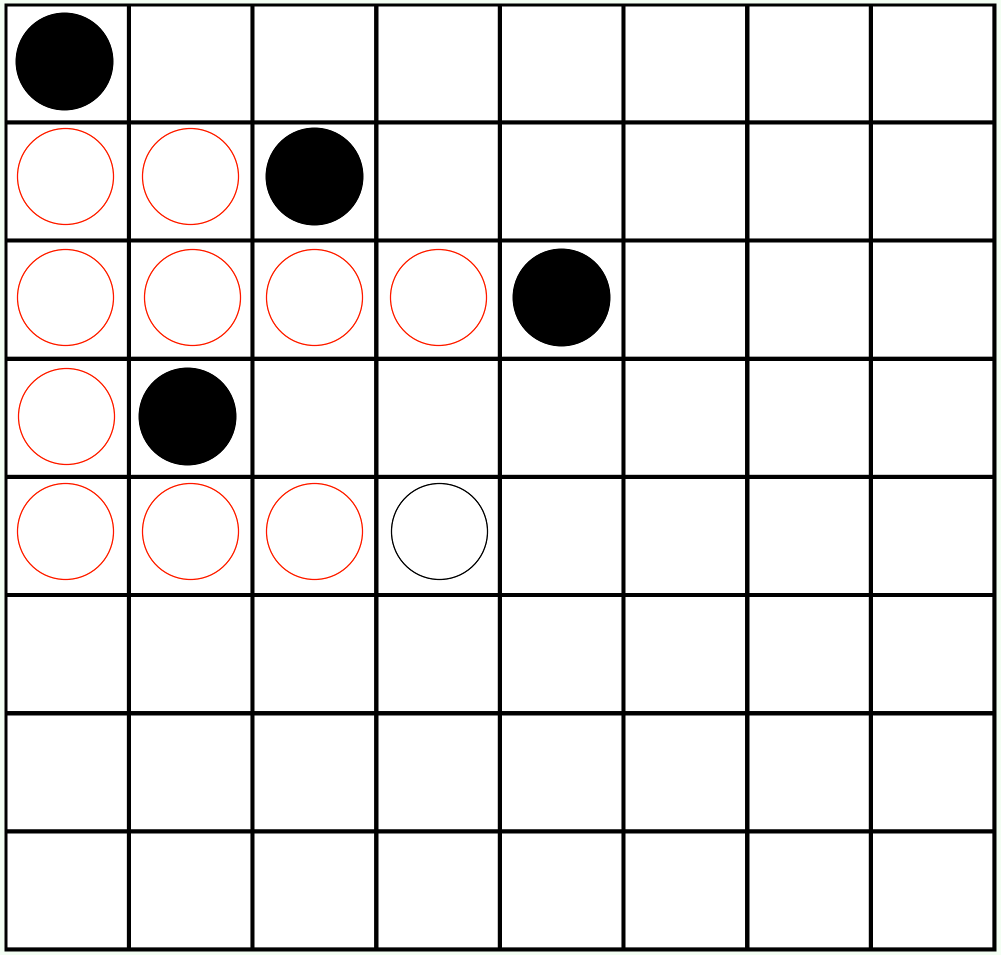
Backtracks to Q5



Tests 45

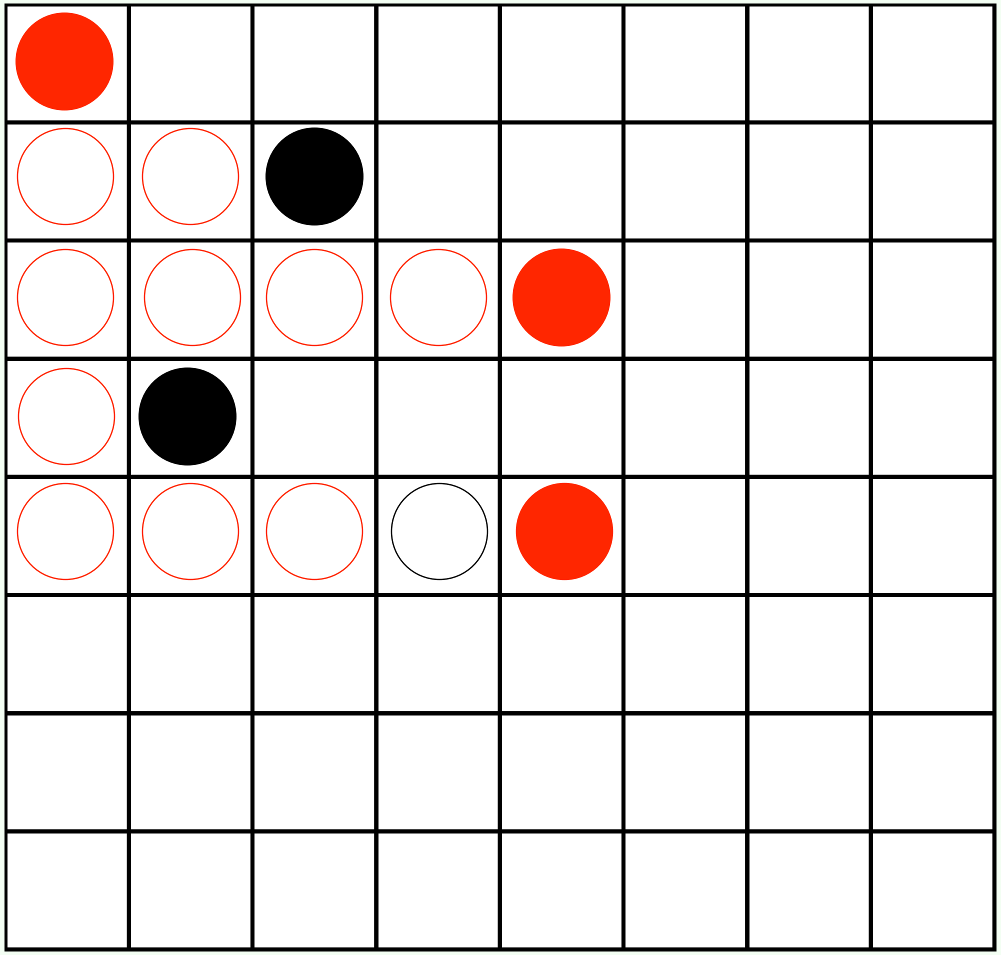
Backtracks 0+ 1 = 1

Backtracking



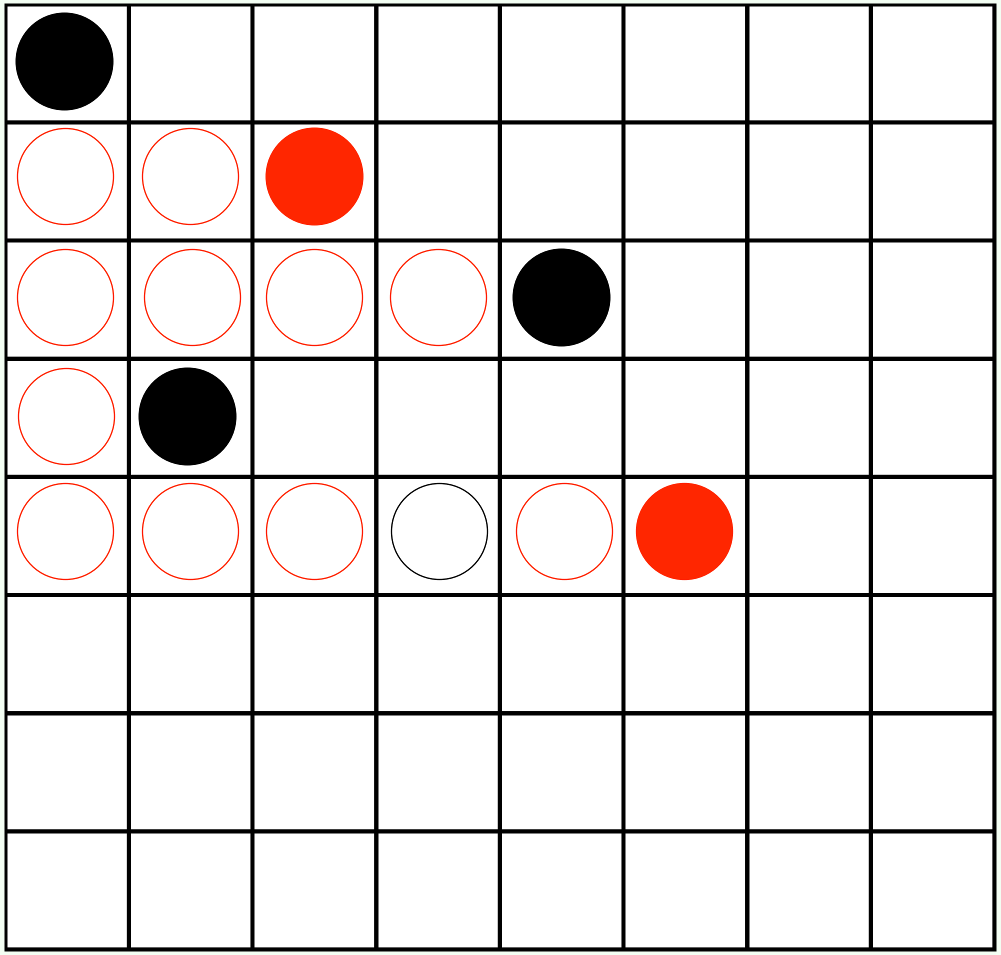
Tests 45 **Backtrackings 1**

Backtracking



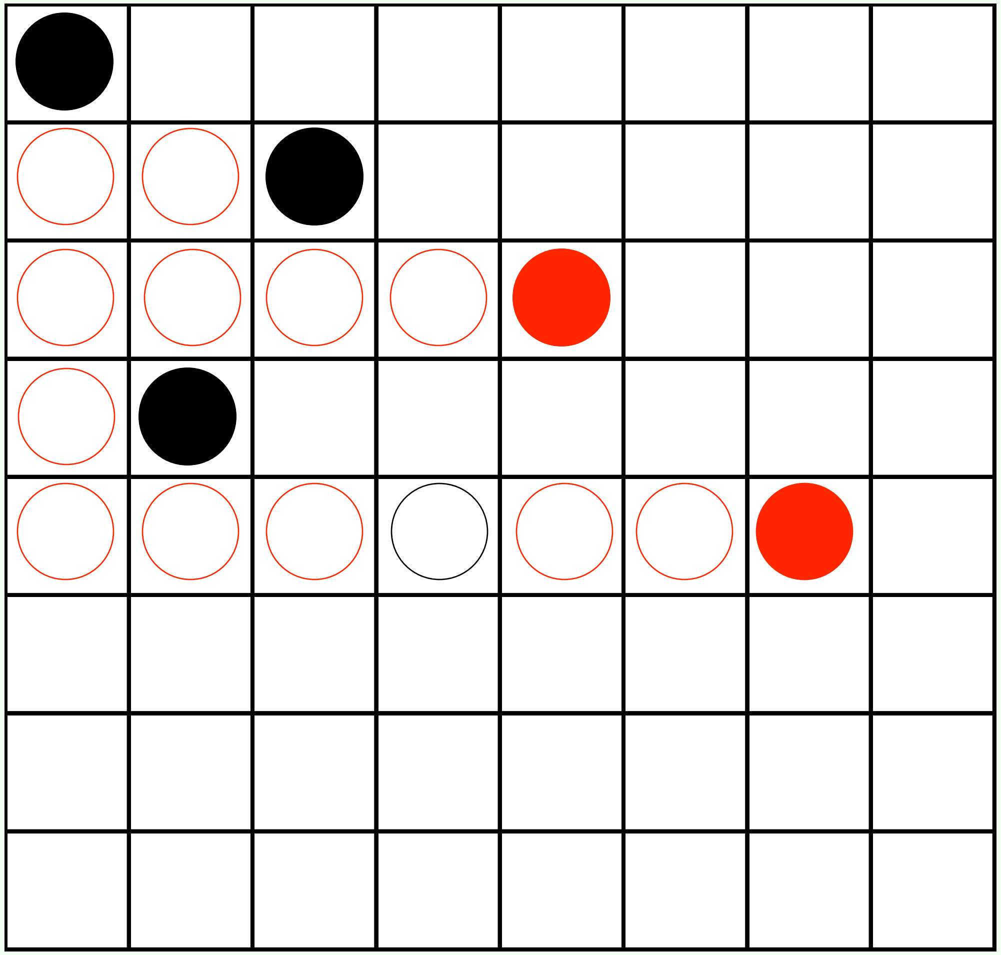
Tests 45 + 1 = 46 **Backtracks 1**

Backtracking



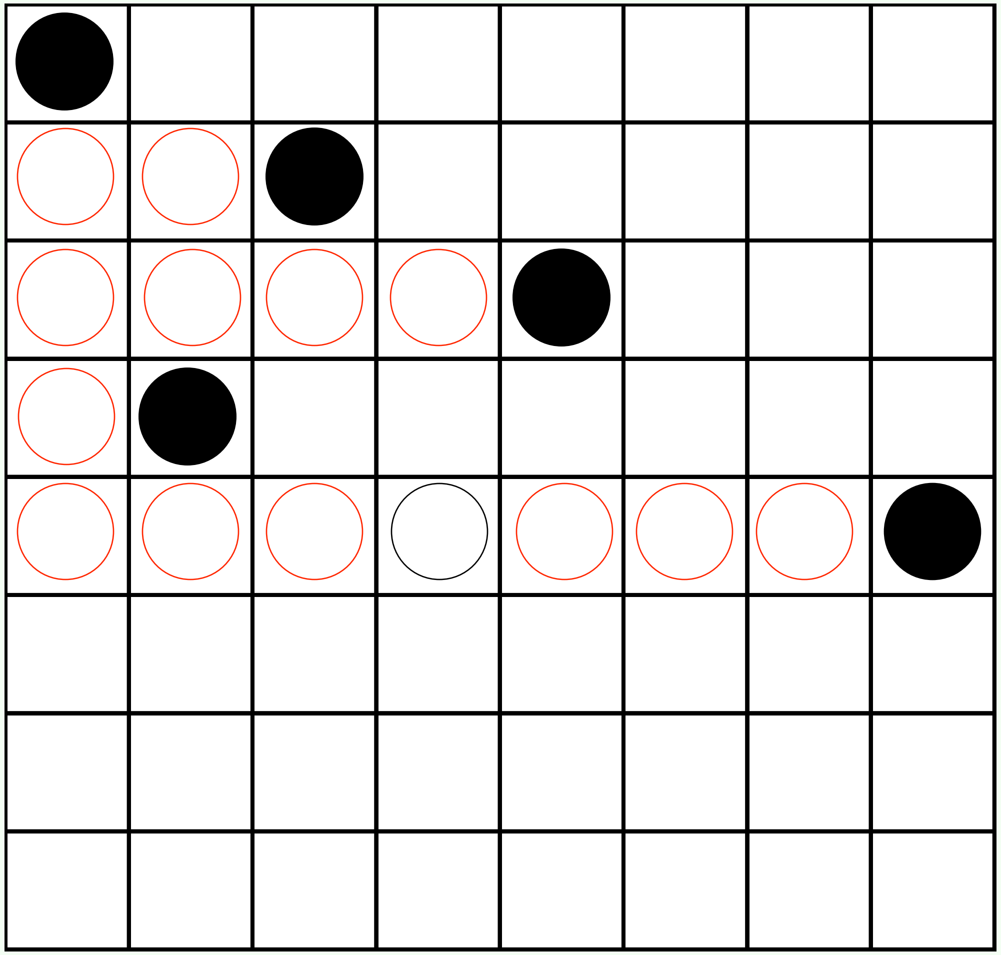
Tests 46 + 2 = 48 **Backtracks 1**

Backtracking



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

Backtracking



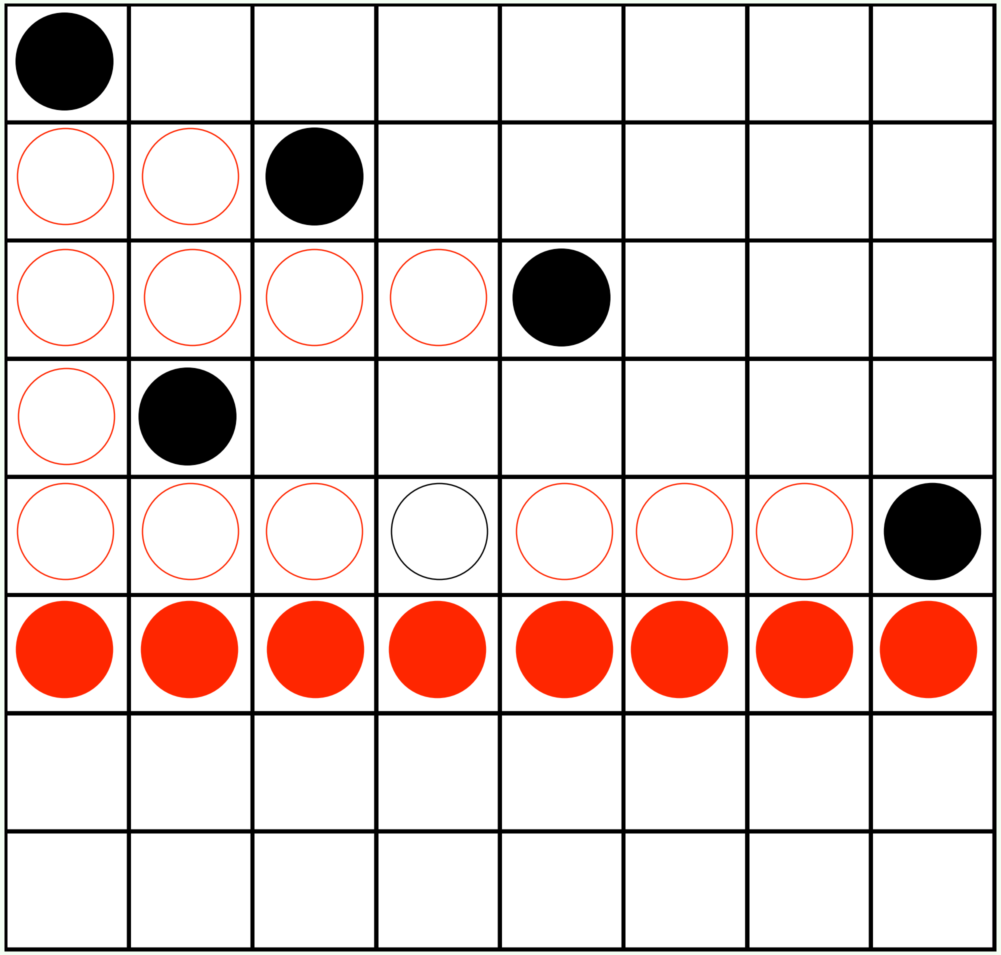
Tests 51 + 4 = 55 **Backtracks 1**

Backtracking

Q6 Fails

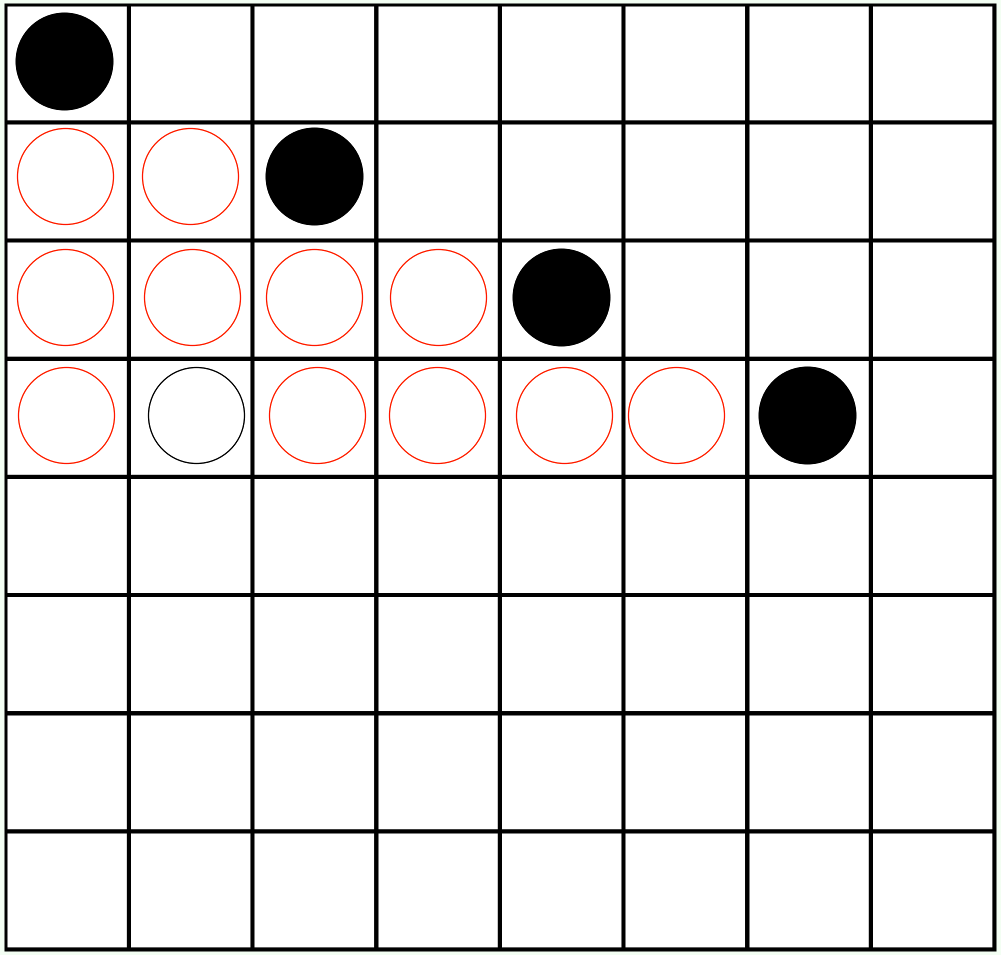
Backtracks to Q5

and next to Q4



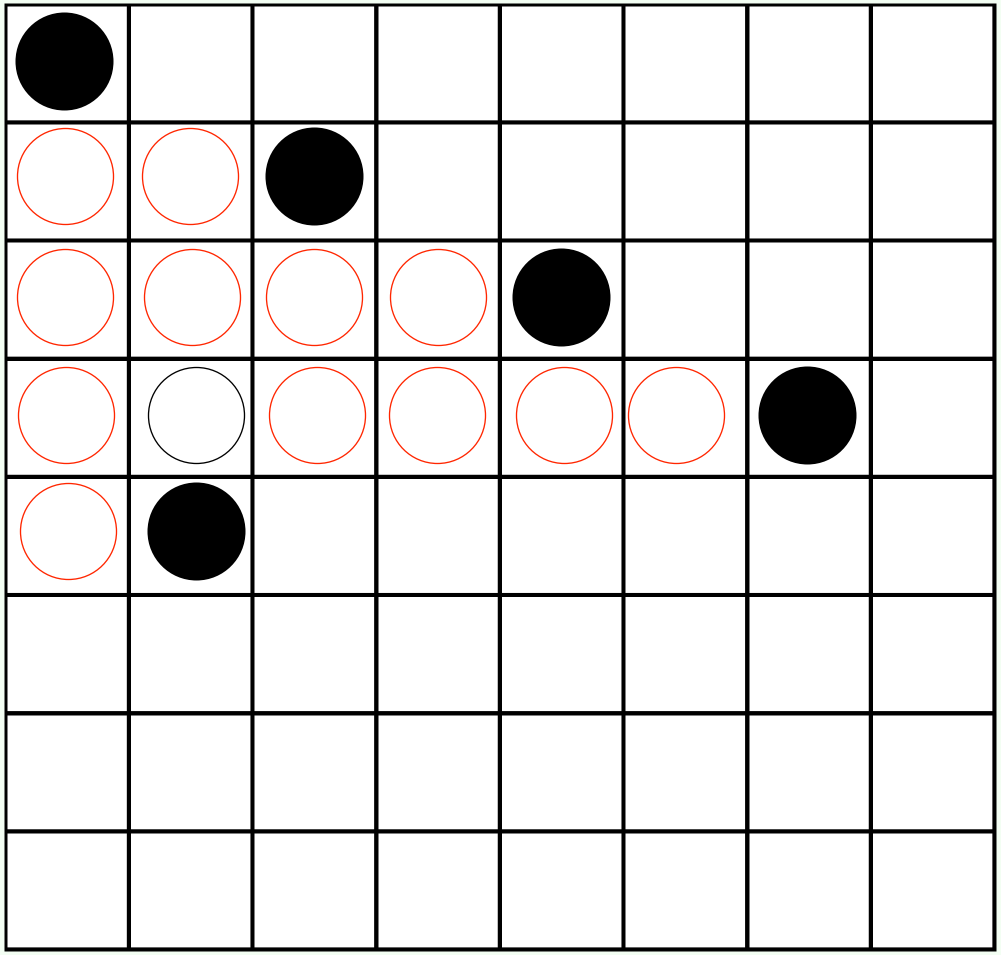
Tests 55+1+3+2+4+3+1+2+3 = 74 Backtracks 1+2 = 3

Backtracking



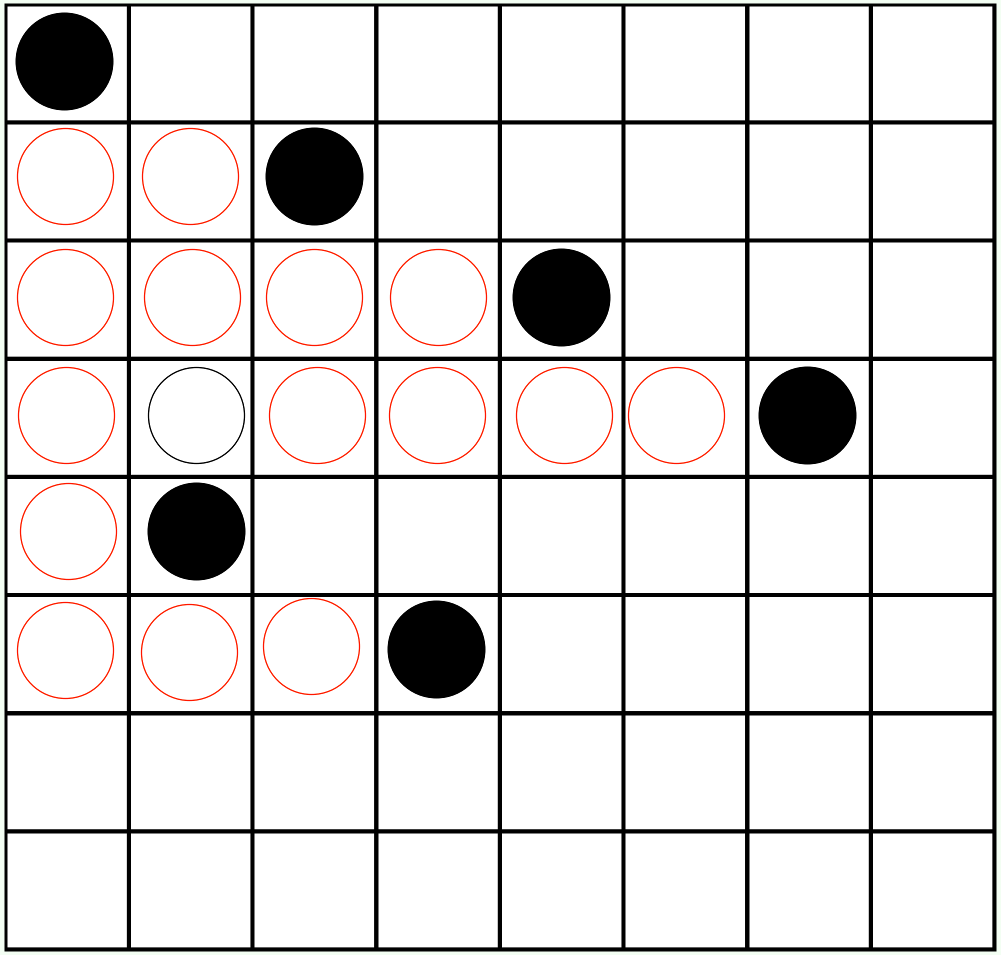
Tests 74+2+1+2+3+3= 85 Backtracks 3

Backtracking



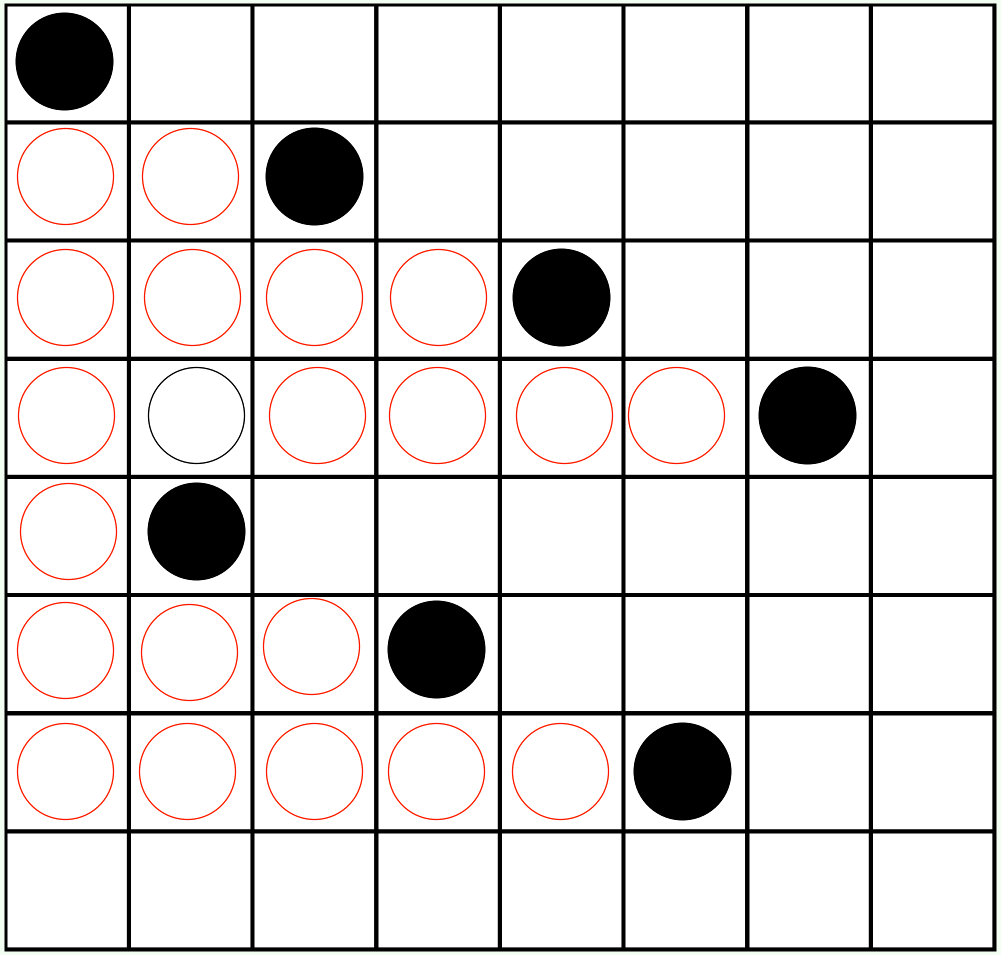
Tests 85 + 1 + 4 = 90 **Backtracks 3**

Backtracking



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

Backtracking

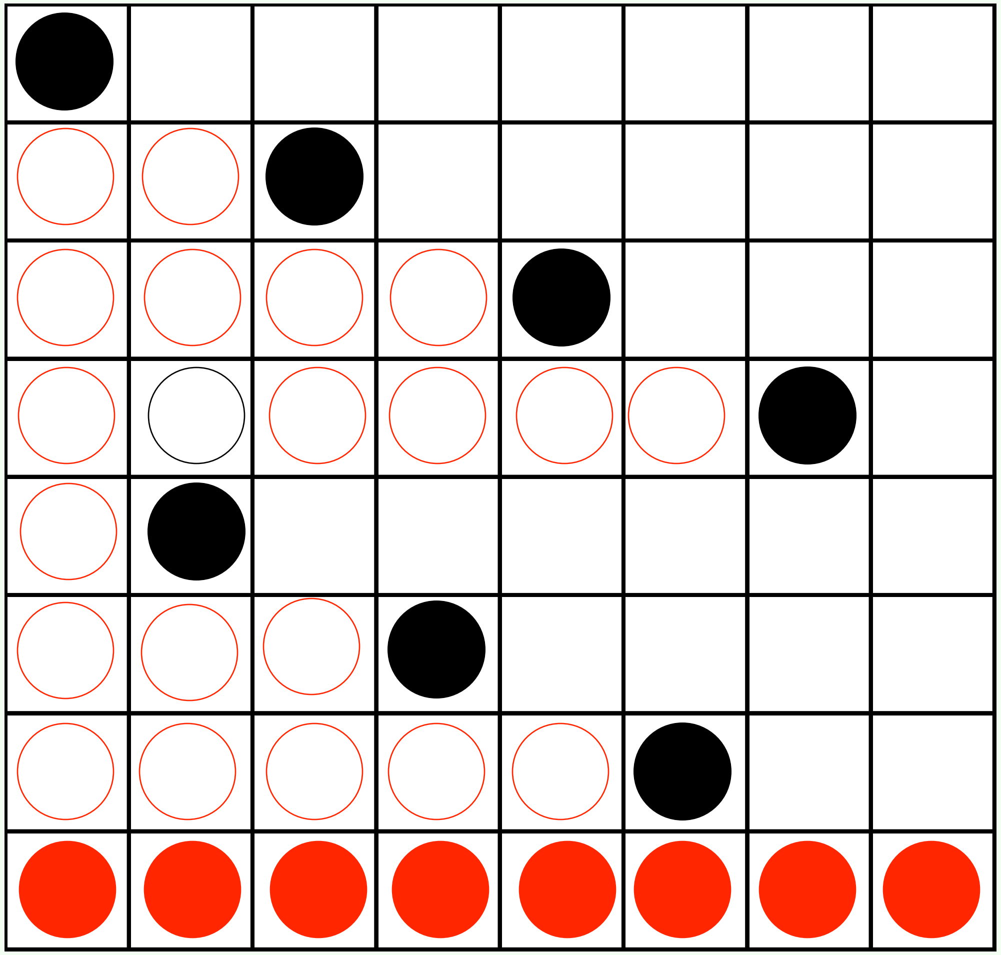


Tests 101+1+5+2+4+3+6= 122 Backtracks 3

Backtracking

Q8 Fails

Backtracks to Q7

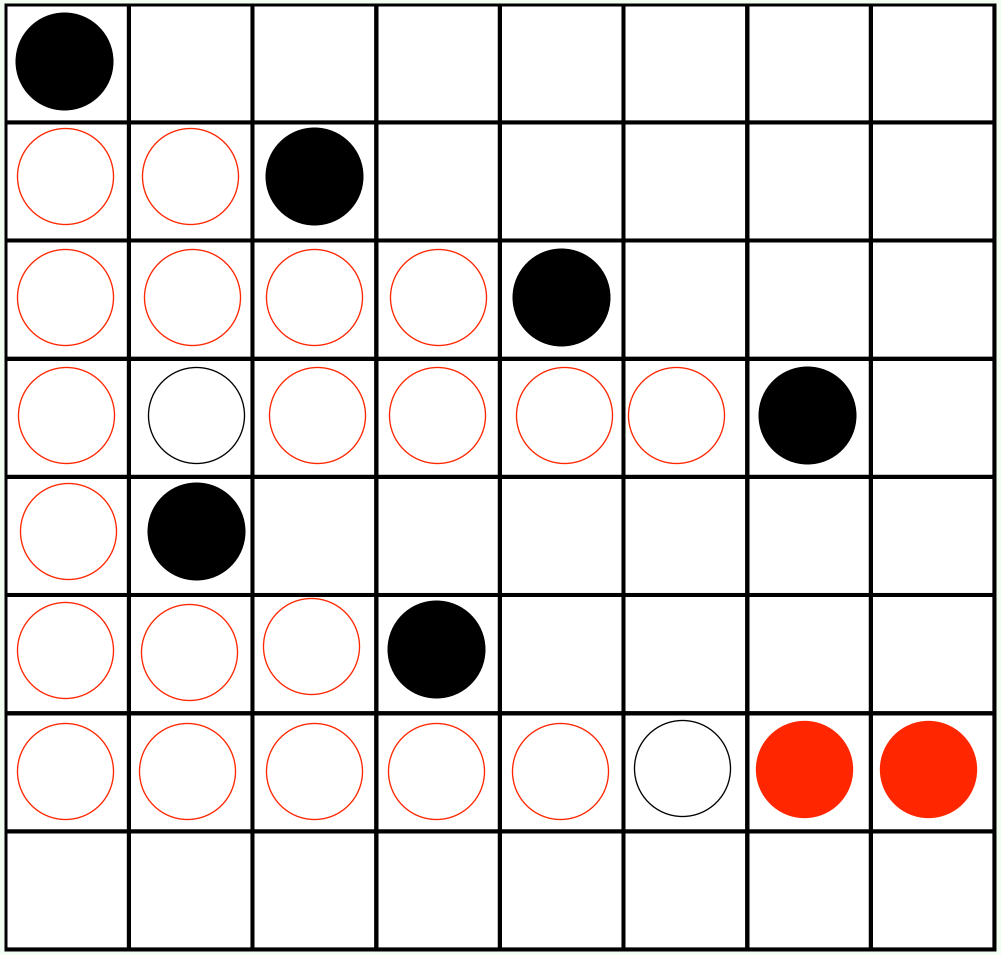


Tests 122+1+5+2+6+3+6+4+1= 150 Backtracks 3+1=4

Backtracking

Q7 Fails

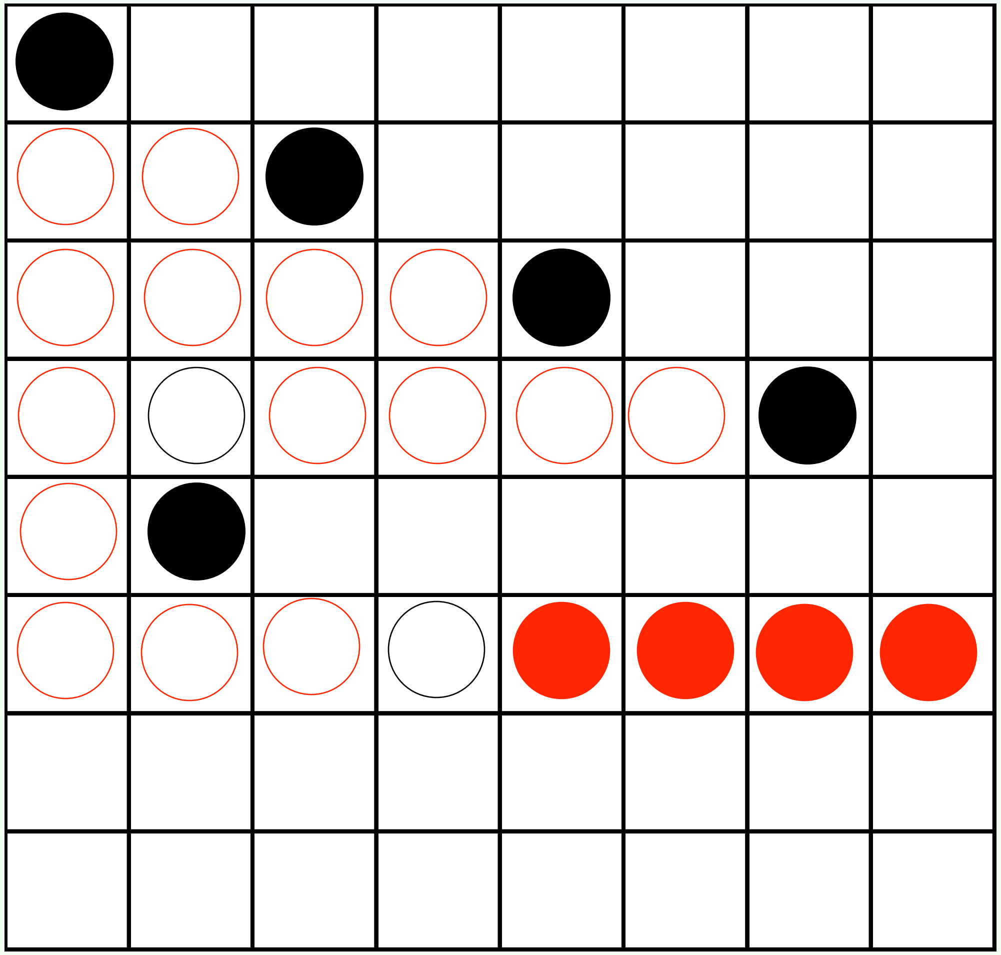
Backtracks to Q6



Tests 150+1+2= 153

Backtracks 4+1=5

Backtracking

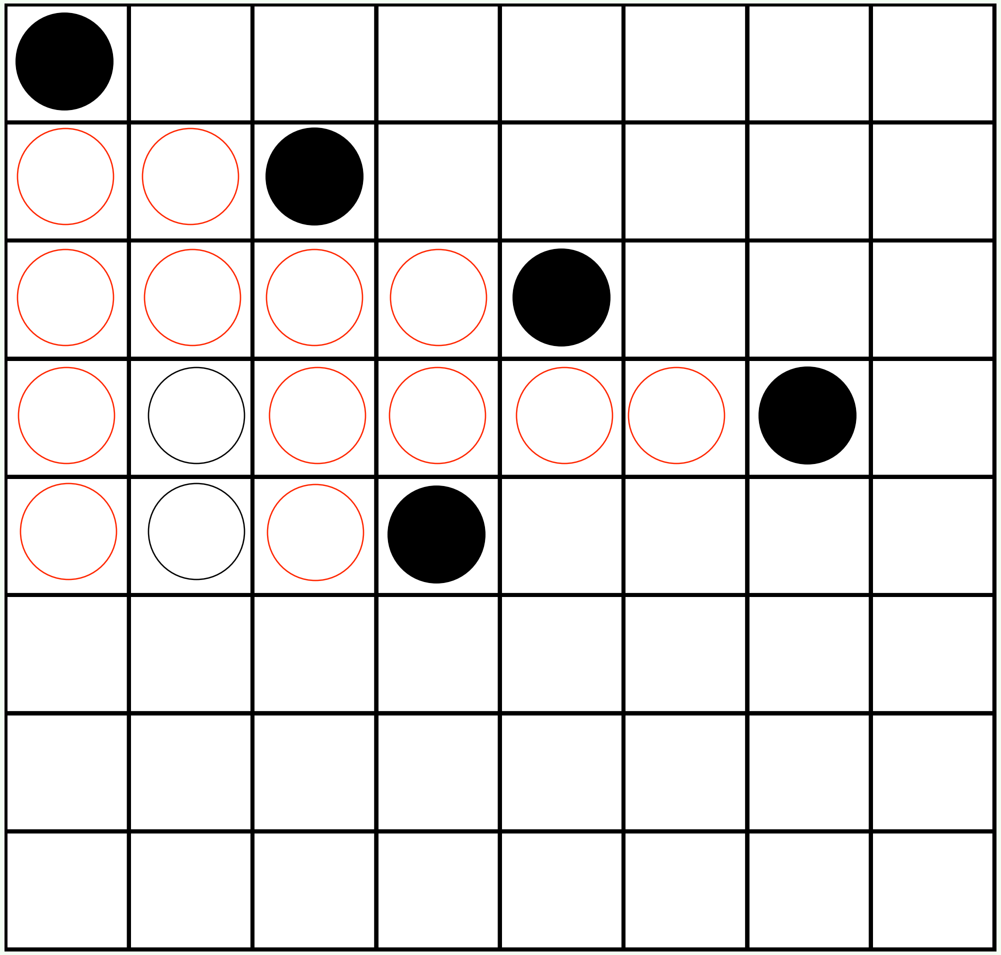


Q6 Fails

Backtracks
to
Q5

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

Backtracking

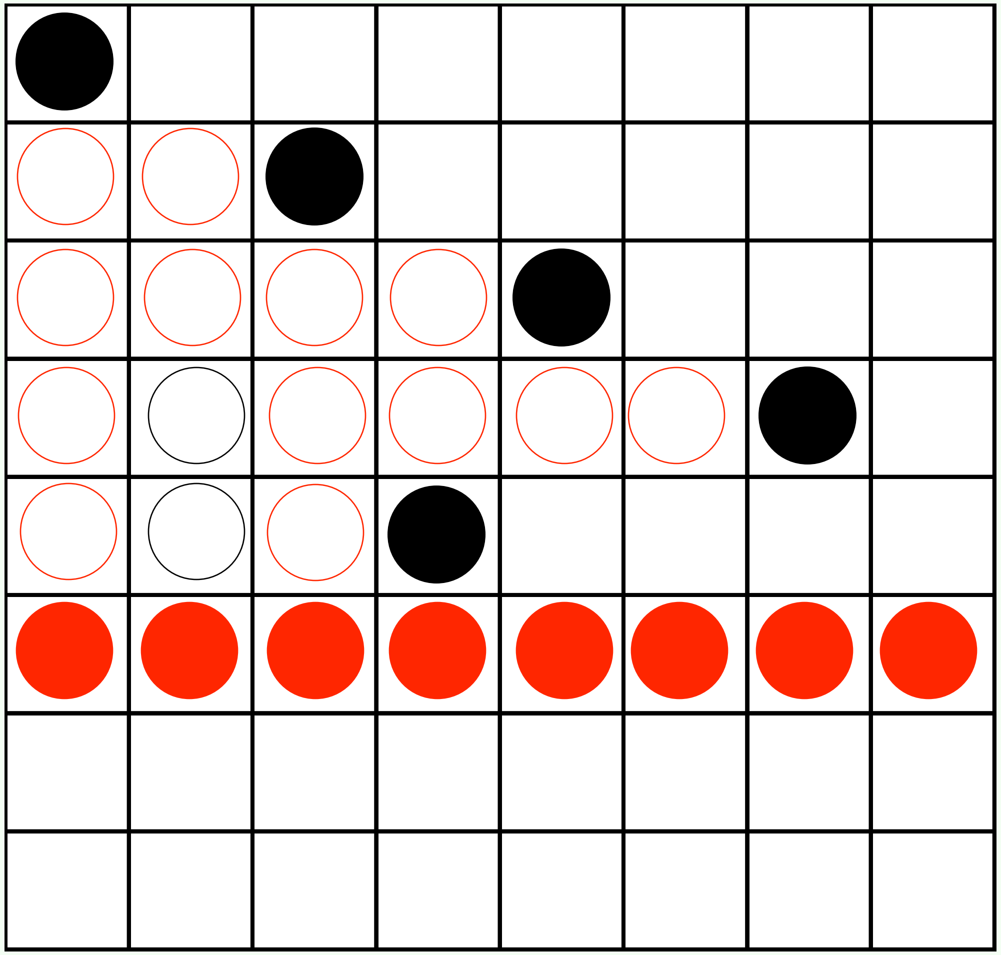


Tests 162+2+4= 168 **Backtracks 6**

Backtracking

Q6 Fails

Backtracks to Q5

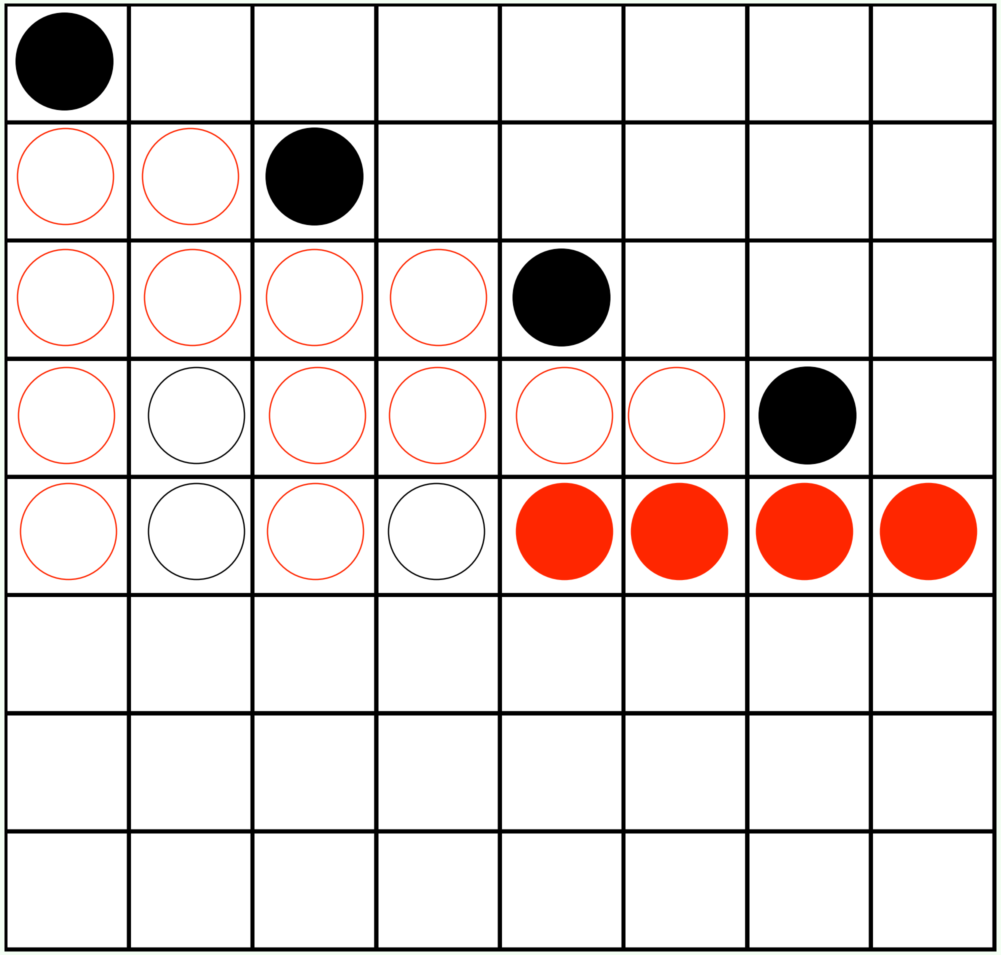


Tests 168+1+3+2+5+3+1+2+3= 188 Backtracks 6+1 = 7

Backtracking

Q5 Fails

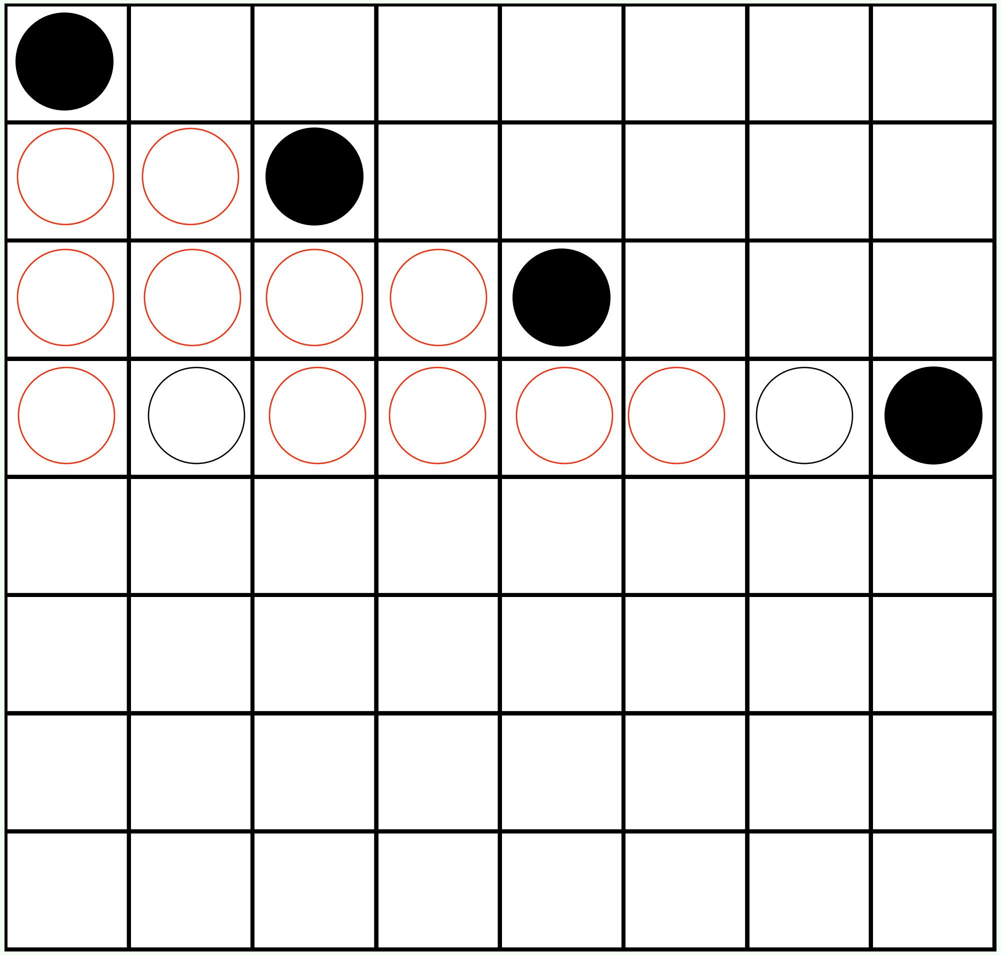
Backtracks to Q4



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

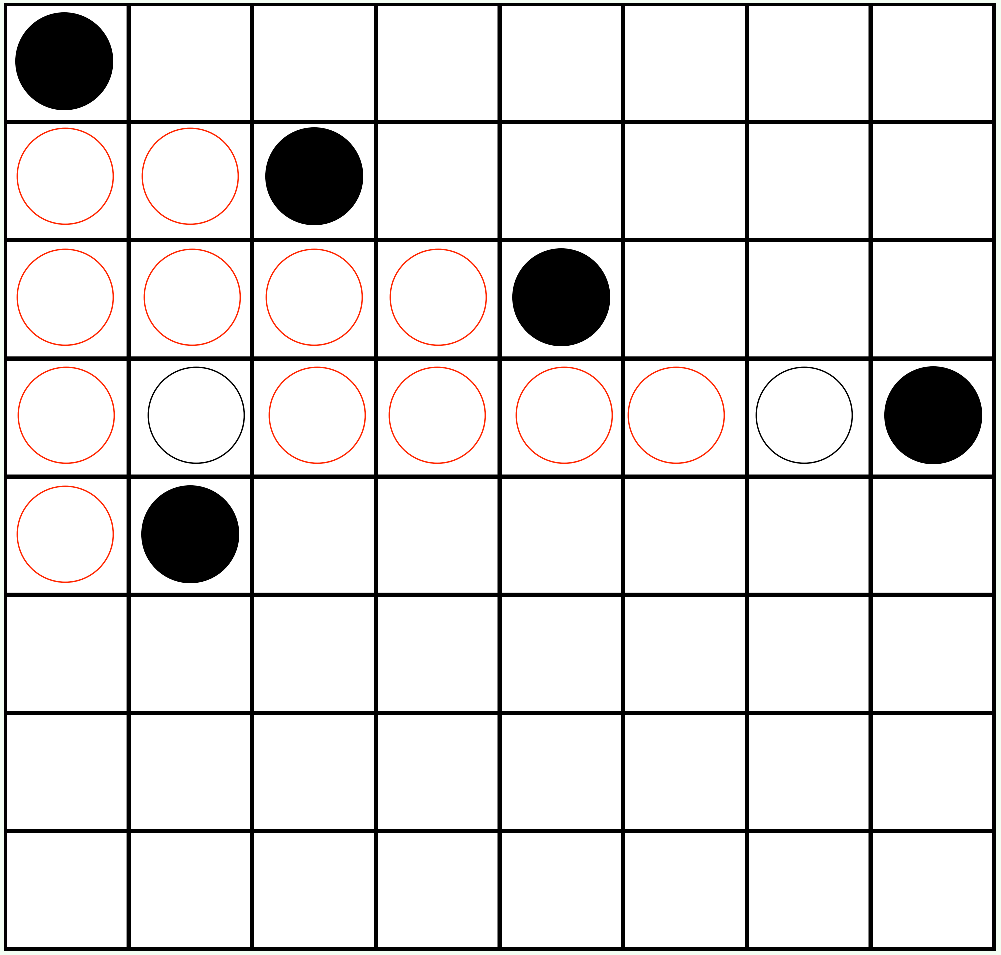
Backtracks 7+1=8

Backtracking



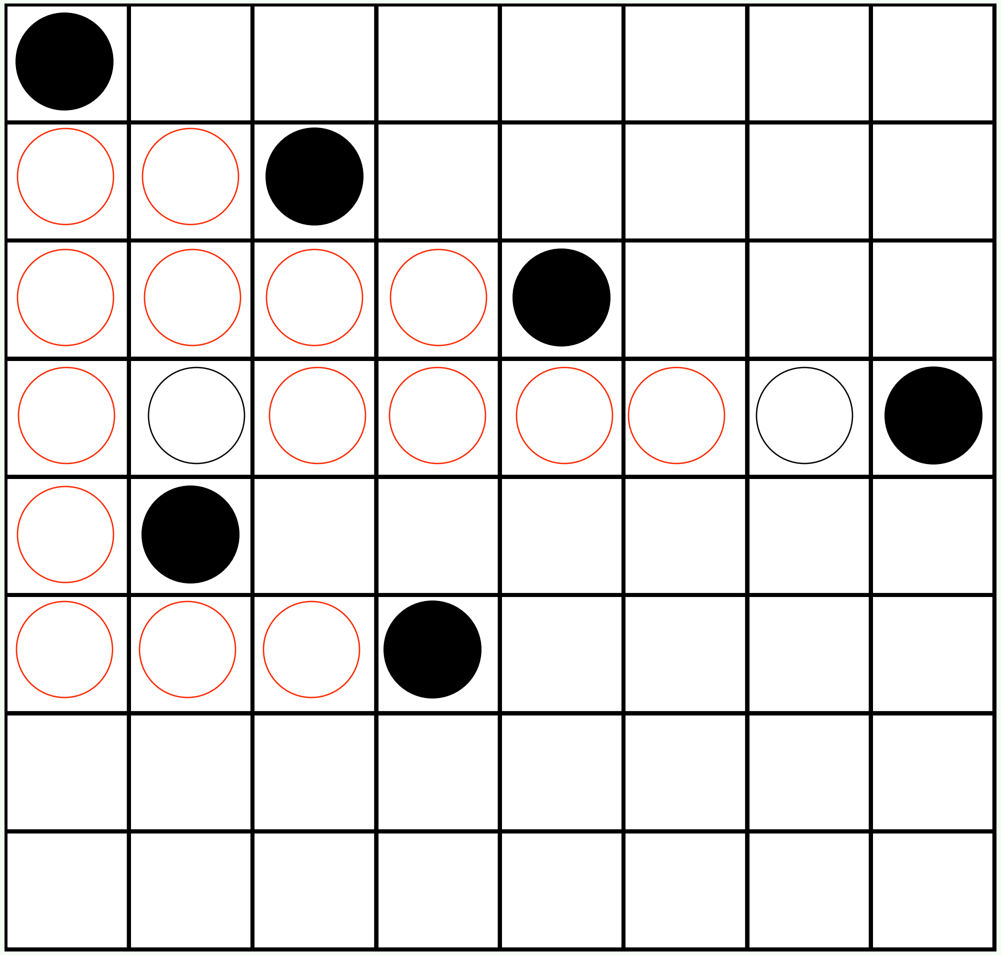
Tests 198 + 3 = 201 **Backtracks 8**

Backtracking



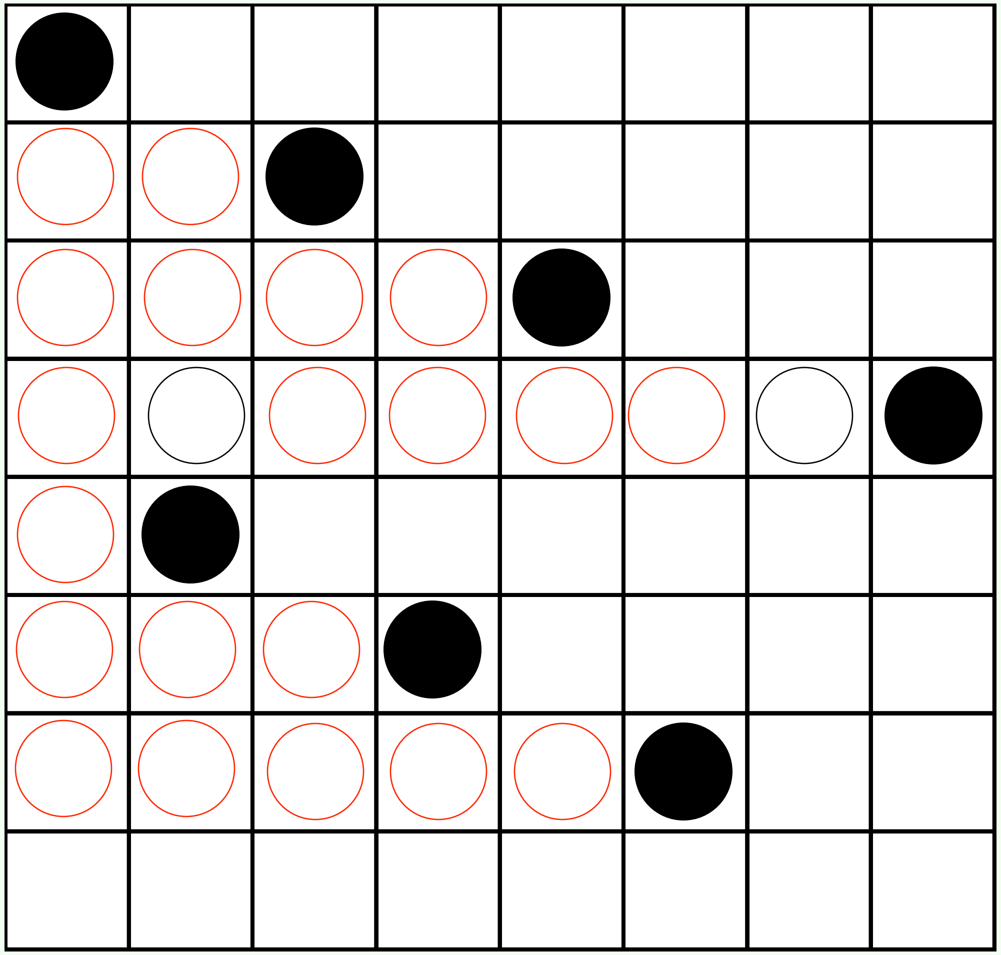
Tests 201+1+4 = 206 **Backtracks 8**

Backtracking



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

Backtracking

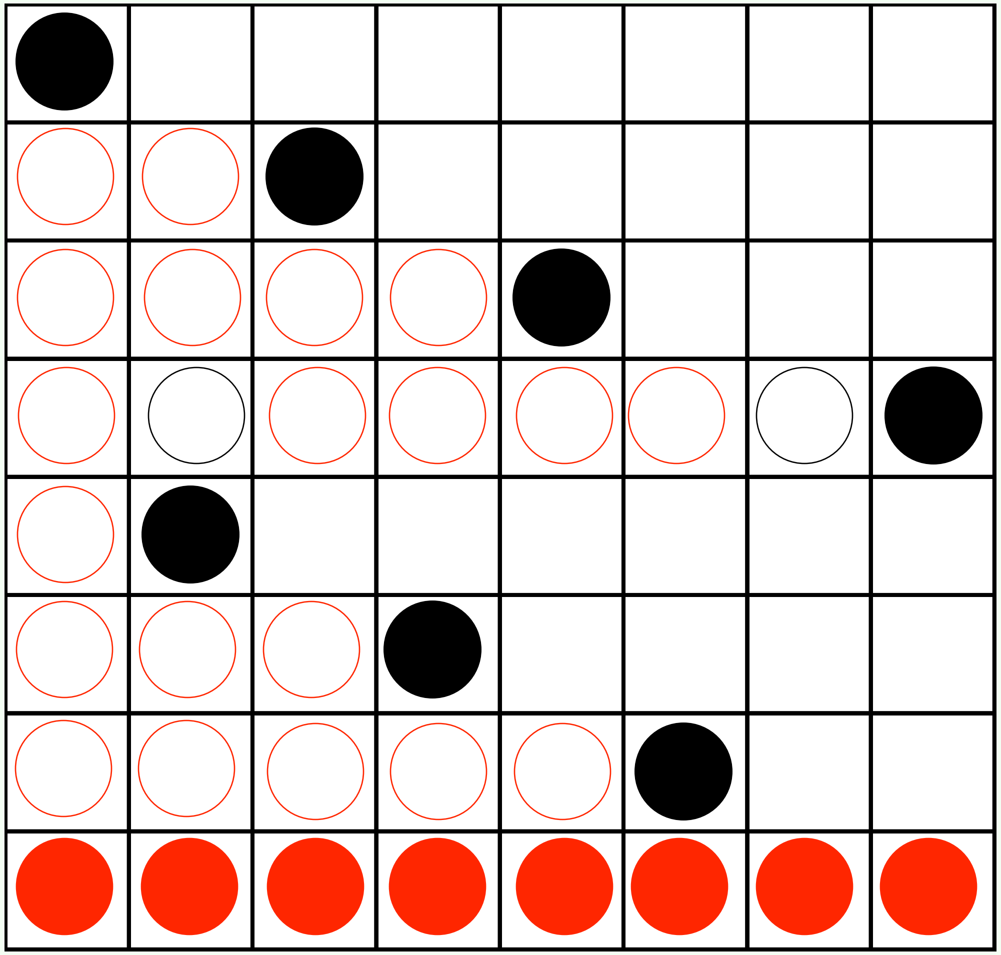


Tests 217+1+5+2+5+3+6 = 239 Backtracks 8

Backtracking

Q8 Fails

Backtracks to Q7

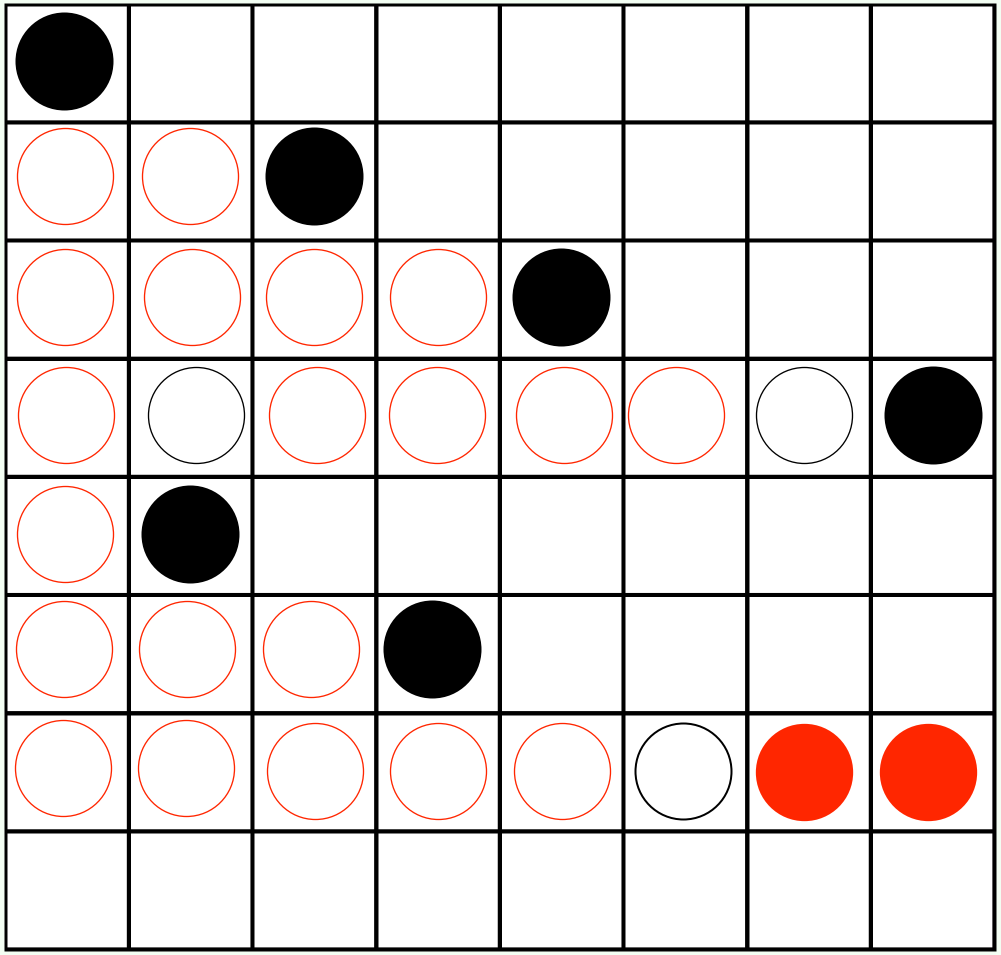


Tests 239+1+5+2+4+3+6+7+7= 274 Backtracks 8+1 = 9

Backtracking

Q7 Fails

Backtracks to Q6

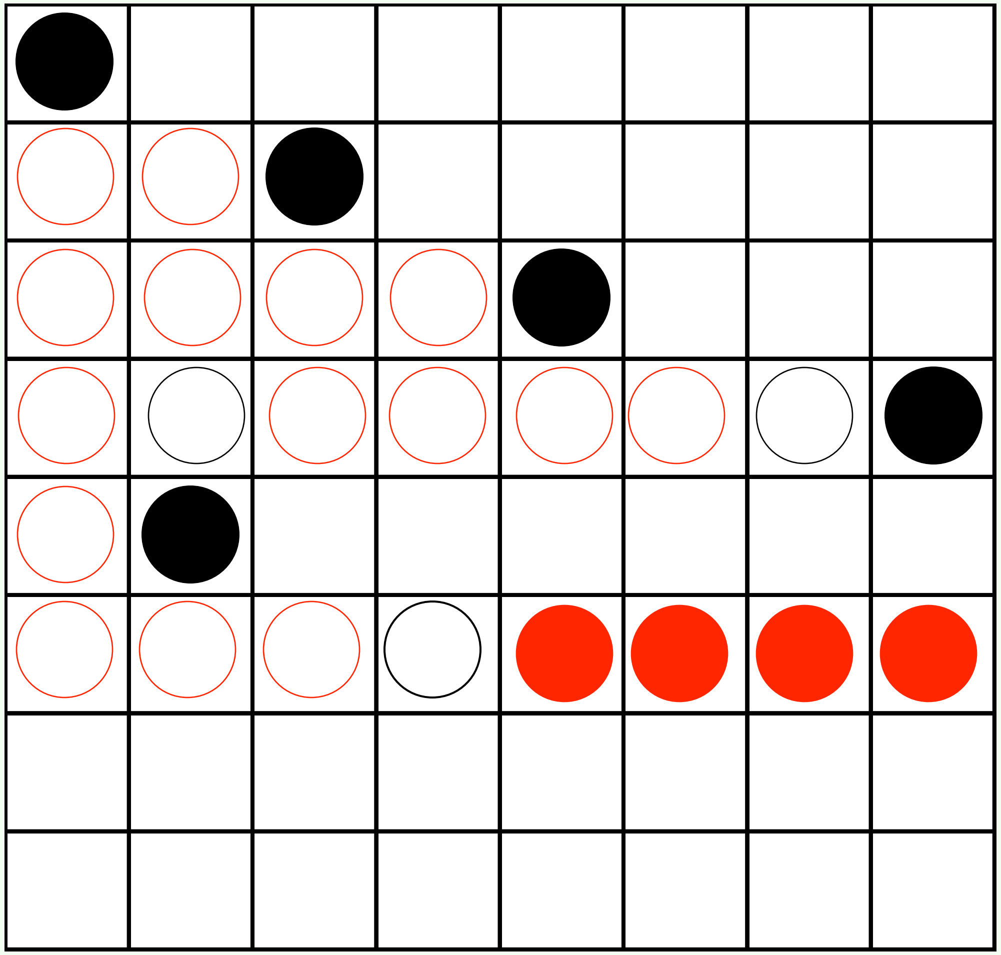


Tests 274+1+2= 277 Backtracks 9+1=10

Backtracking

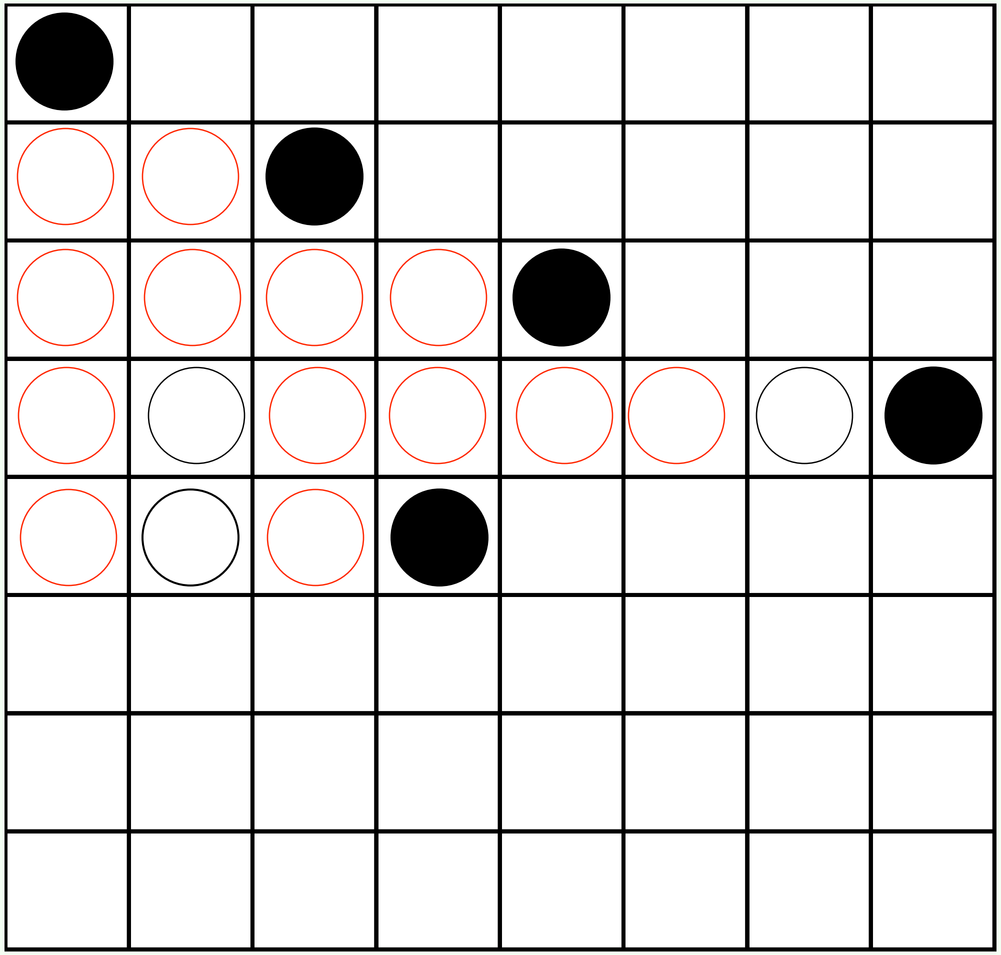
Q6 Fails

Backtracks
to
Q5



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

Backtracking

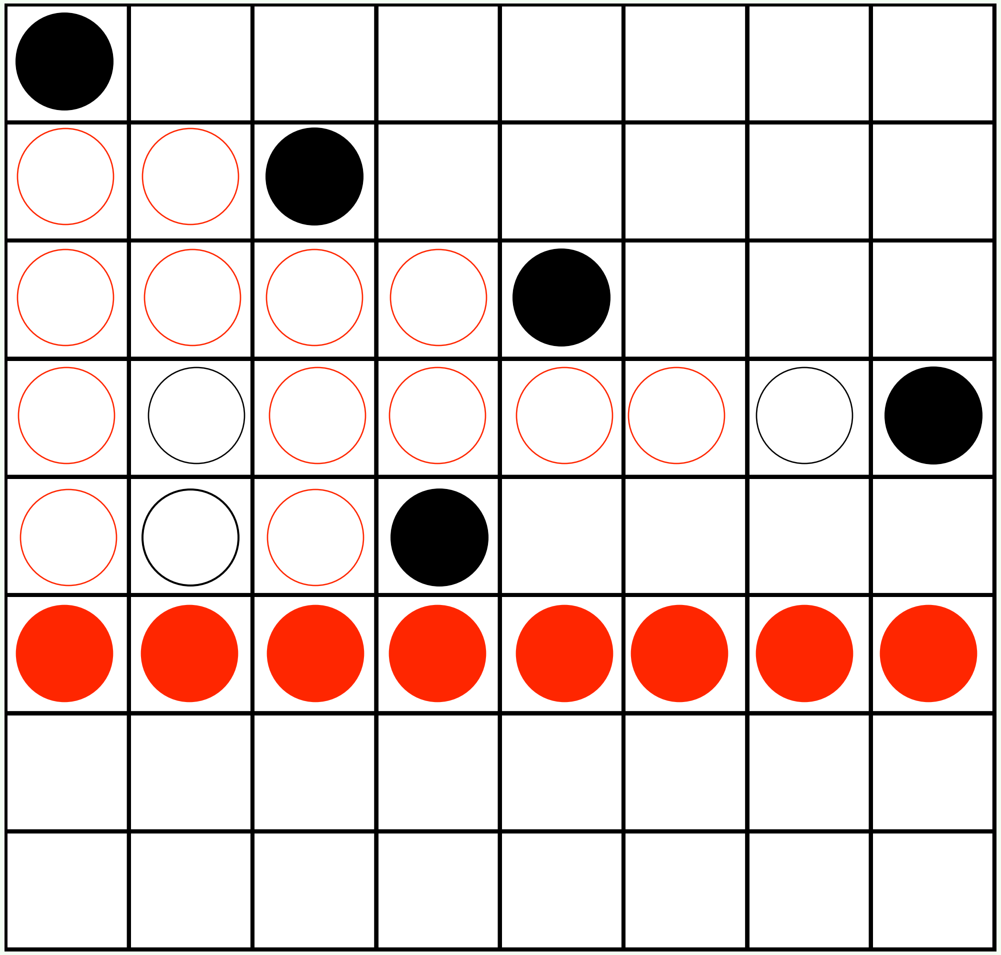


Tests 286+2+4= 292 **Backtracks 11**

Backtracking

Q6 Fails

Backtracks to Q5



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

Backtracking

Q5 Fails

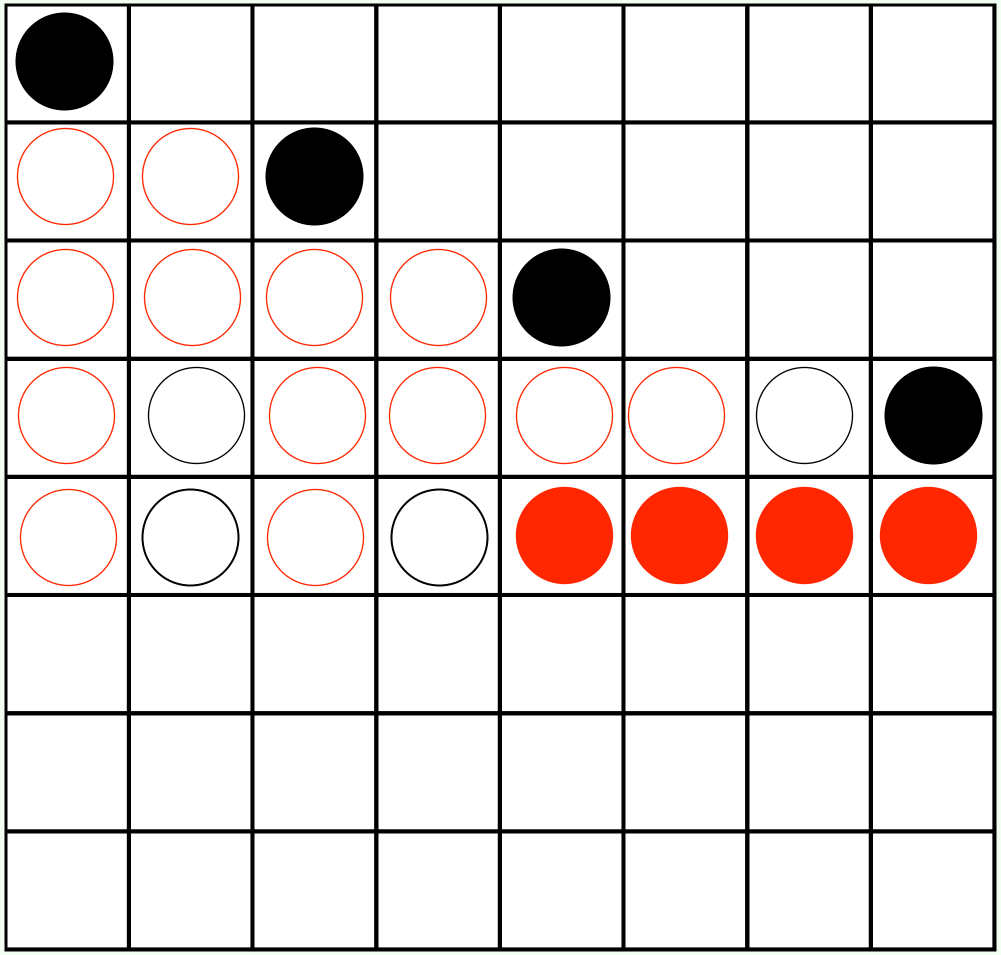
Backtracks

to

Q4

and next to

Q3



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

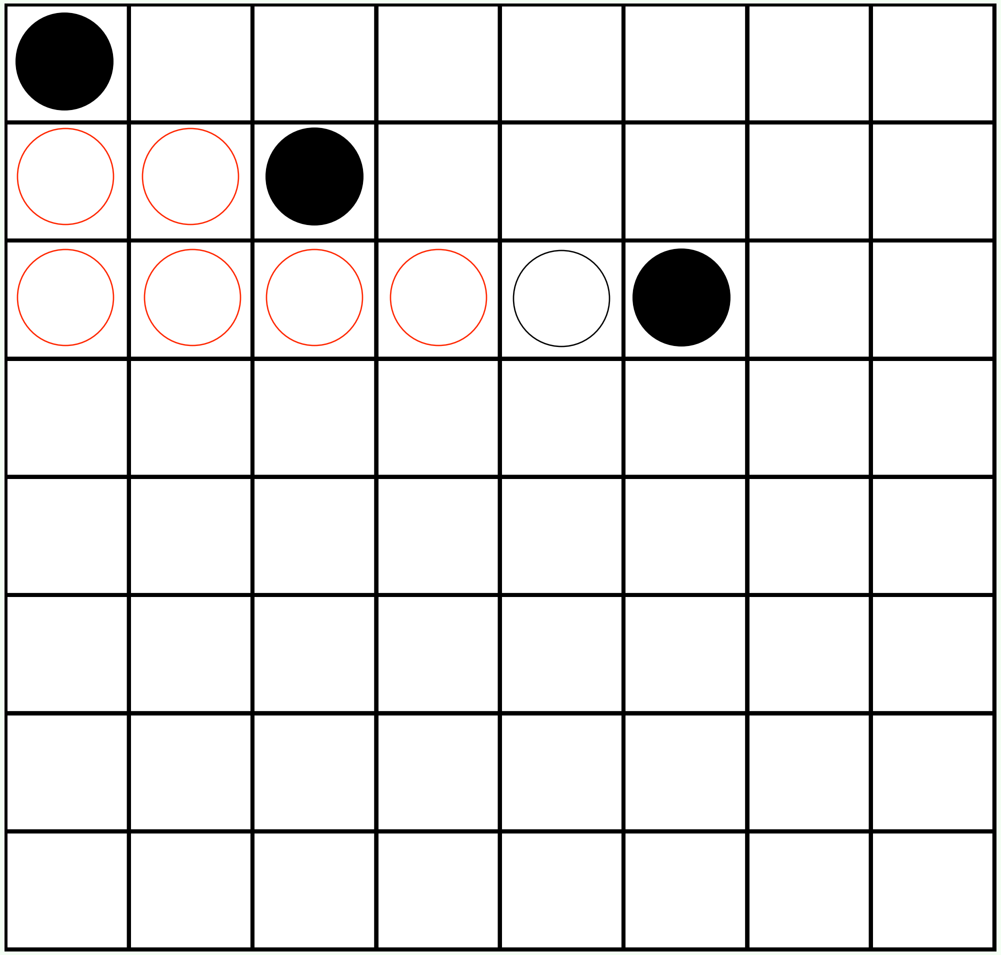
Backtracking

$Q_1 = 1$

$Q_2 = 3$

$Q_3 = 5$

Impossible !

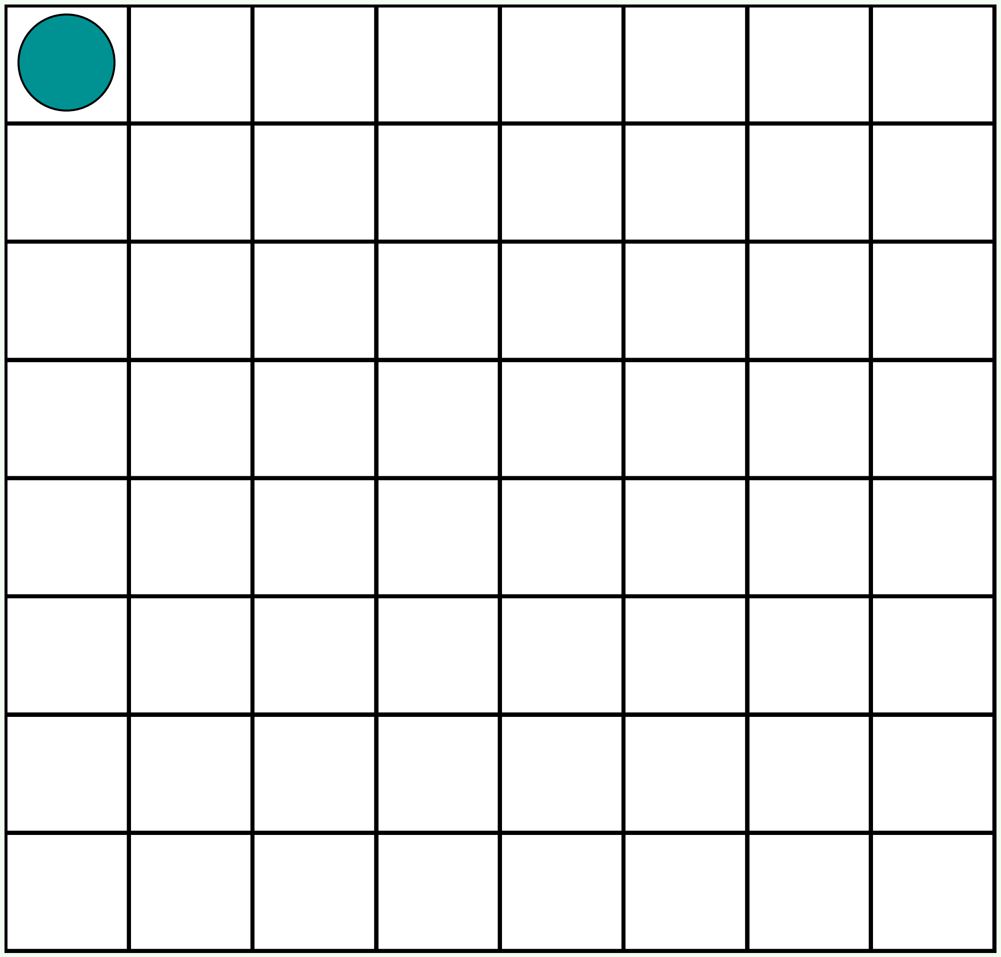


Tests 322 + 2 = 324 **Backtracks 14**

Search Methods (2) – Backtracking + Propagation

- A more efficient backtracking search strategy sees constraints as active constructs:
    - Whenever a variable is assigned a variable, the consequences of such assignment are taken into account to narrow the possible values of the variables not yet assigned.
    - If for one such variable there are no values to chose from, then a failure occurs and the search backtracks.
- This is a typical **test and generate** procedure
    - Firstly, values are tested to check their possible use.
    - Secondly, the values are assigned to the variables.
- Clearly, the reasoning that is done should have the adequate complexity otherwise the gains obtained from the narrowing of the search space are offset by the costs of such narrowing.
- This procedure is illustrated again with the 8-queens problem.

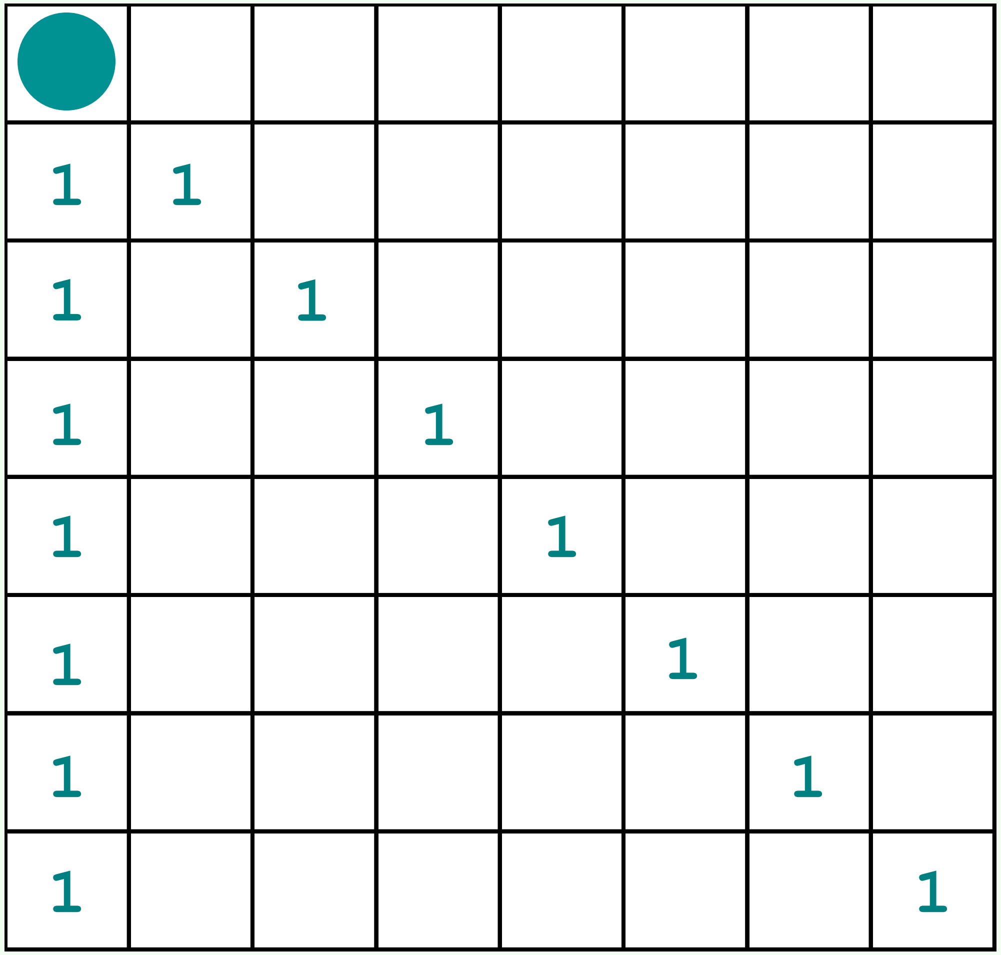
Search Methods (2) - Backtracking + Propagation



Tests 0 **Backtracks 0**

Search Methods (2) - Backtracking + Propagation

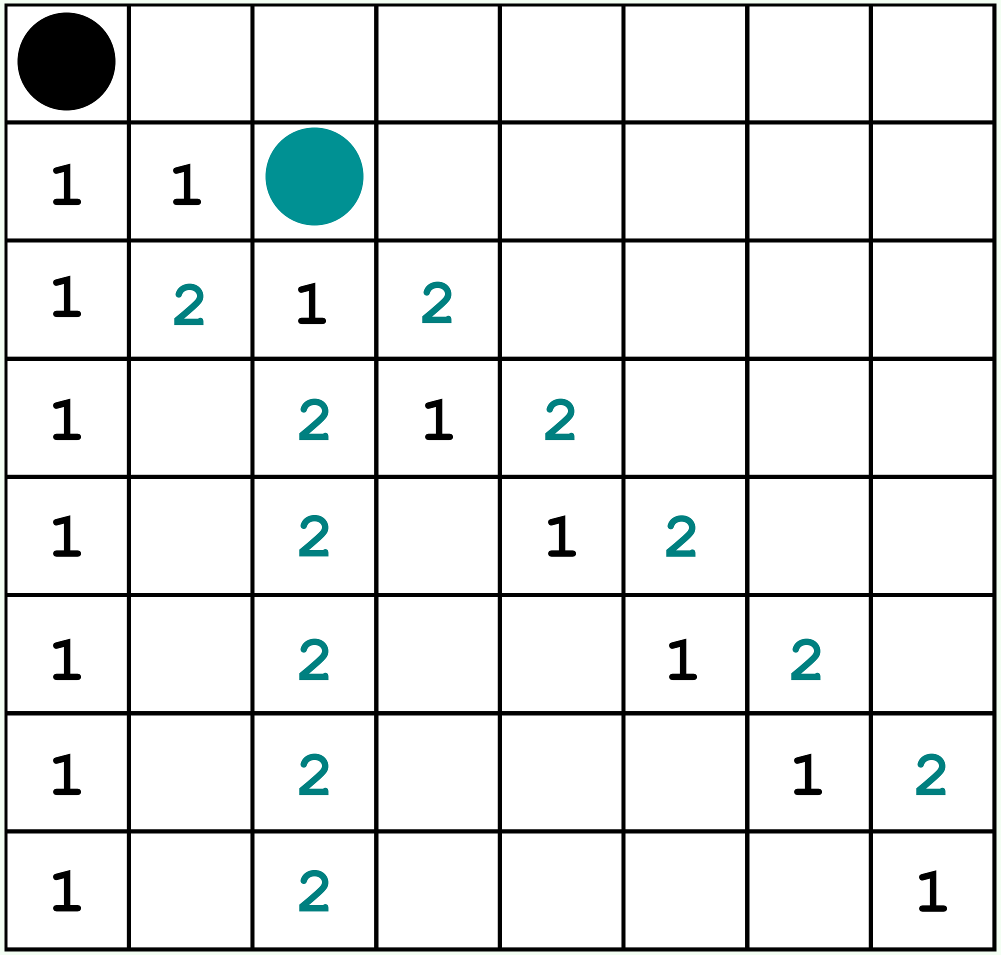
```
Q1 #\= Q2,  L1+Q1 #\= L2+Q2,  L1+Q2 #\= L2+Q1.
```



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

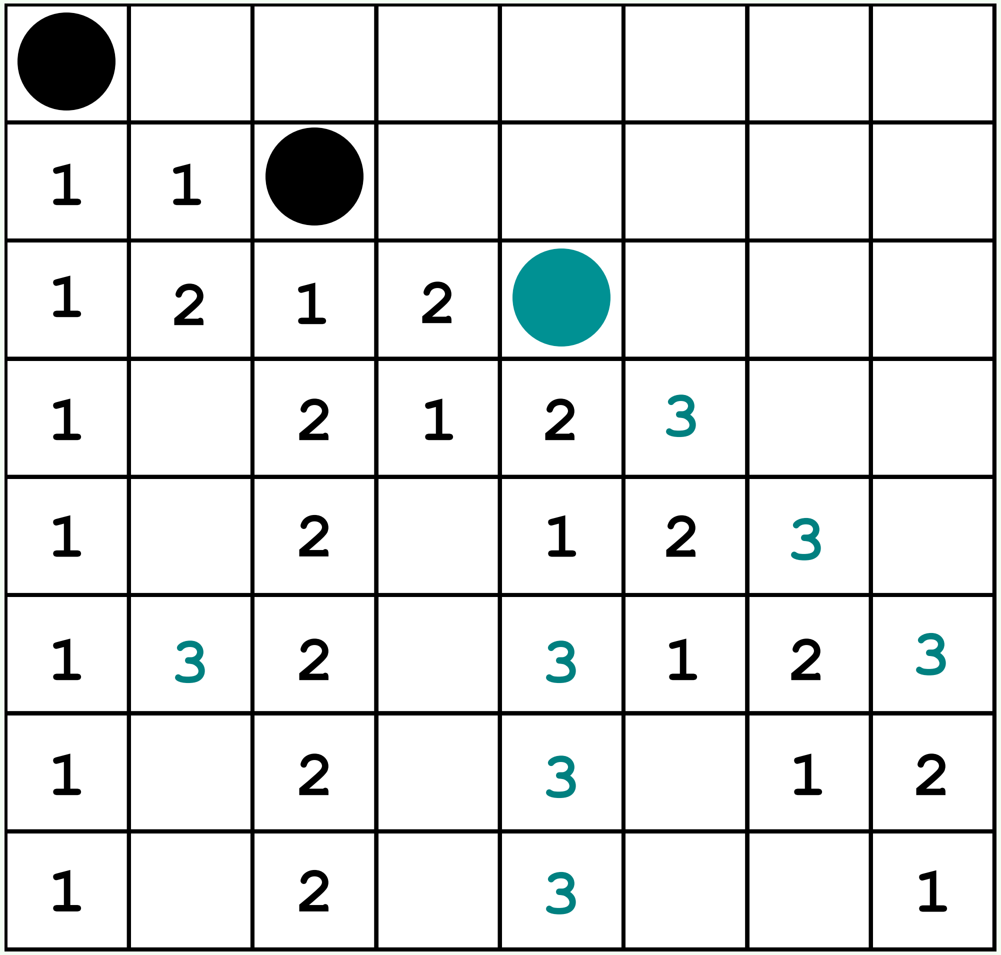
Tests 8 * 7 = 56 **Backtracks 0**

Search Methods (2) - Backtracking + Propagation



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

Search Methods (2) - Backtracking + Propagation



●							
1	1	●					
1	2	1	2	●			
1		2	1	2	3		
1		2		1	2	3	
1	3	2		3	1	2	3
1		2		3		1	2
1		2		3			1

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

Search Methods(2a) – B+P w/Heuristics

- In both types of backtrack search (pure backtracking as well as in backtracking + propagation) there is a *need* for heuristics.
- After all, in decision problems with n variables, a perfect heuristics would find a solution (if there is one) in exactly **n** steps (i.e. with **n** decisions – polinomial time).
- Of course, there are no such perfect heuristics for non-trivial problems (this would imply P = NP, a quite unlikely situation), but good heuristics can nonetheless significantly decrease the search space. Typically a heuristics consists of
  - **Variable selection**: The selection of the next variable to assign a value
  - **Value selection**: Which value to assign to the variable
- The adoption of a backtrack + propagation search method allows better heuristics to be used, that are not available in pure backtrack search methods.
- In particular a very simple heuristics, **first-fail**, is often very useful: whenever a variable is restricted to take a single value, select that variable and value.
- This procedure is again illustrated with the 8-queens problem.

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

Which queen to label?

●							
1	1	●					
1	2	1	2	●			
1		2	1	2	3		
1		2		1	2	3	
1	3	2		3	1	2	3
1		2		3		1	2
1		2		3			1

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

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

Q_6

may only take value

4

●							
1	1	●					
1	2	1	2	●			
1		2	1	2	3		
1		2		1	2	3	
1	3	2	🟡	3	1	2	3
1		2		3		1	2
1		2		3			1

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

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

●							
1	1	●					
1	2	1	2	●			
1	6	2	1	2	3		
1		2	6	1	2	3	
1	3	2	🟡	3	1	2	3
1		2	6	3		1	2
1	6	2	6	3	6		1

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

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

Q_8

may only take value

7

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

Tests 126 **Backtracks 0**

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

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

Tests 126 **Backtracks 0**

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

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

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

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

Q_4

may only take value

8

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

Tests 132 **Backtracks 0**

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

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

Tests 132 **Backtracks 0**

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

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

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

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

Q_5

may only take value

2

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

Tests 135 **Backtracks 0**

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

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

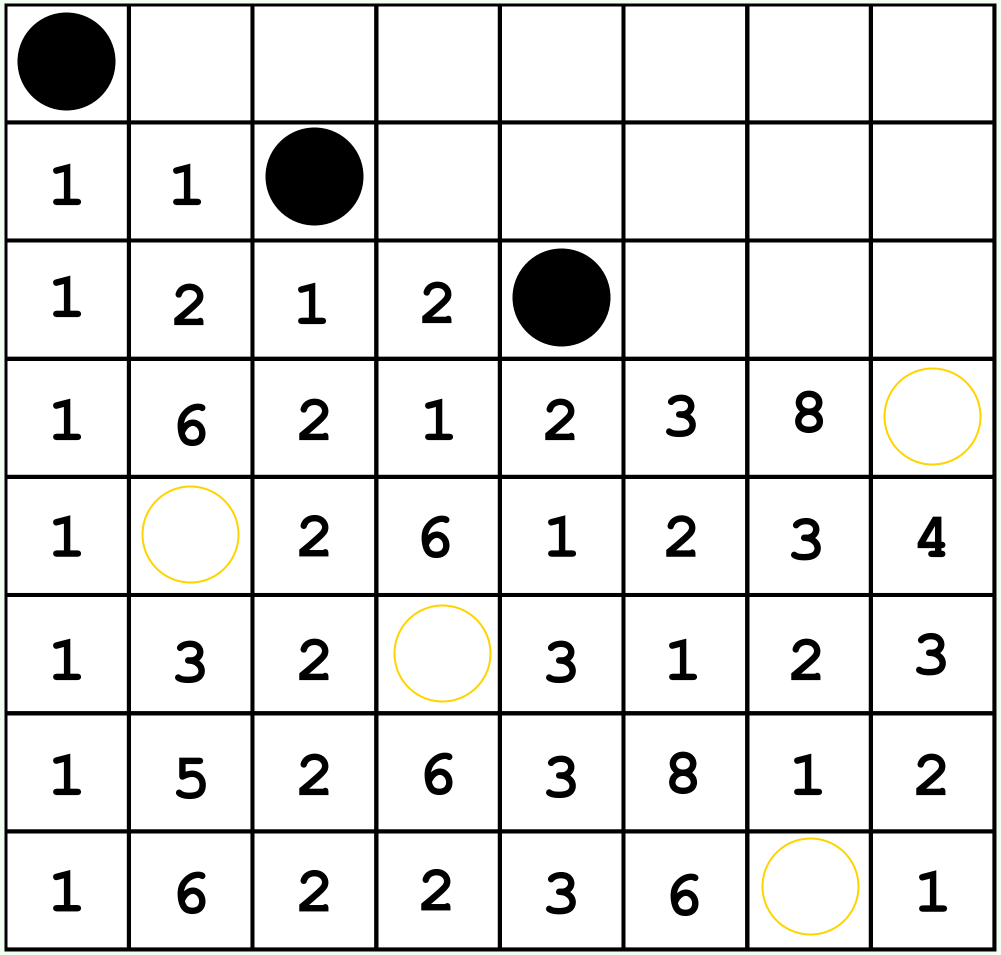
Tests 135 **Backtracks 0**

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

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

Tests 135+1=136 **Backtracks 0**

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



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

Tests 136 **Backtracks 0**

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

Q_7

may take NO value

Failure!

Backtracks

... to Q_3 !

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

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

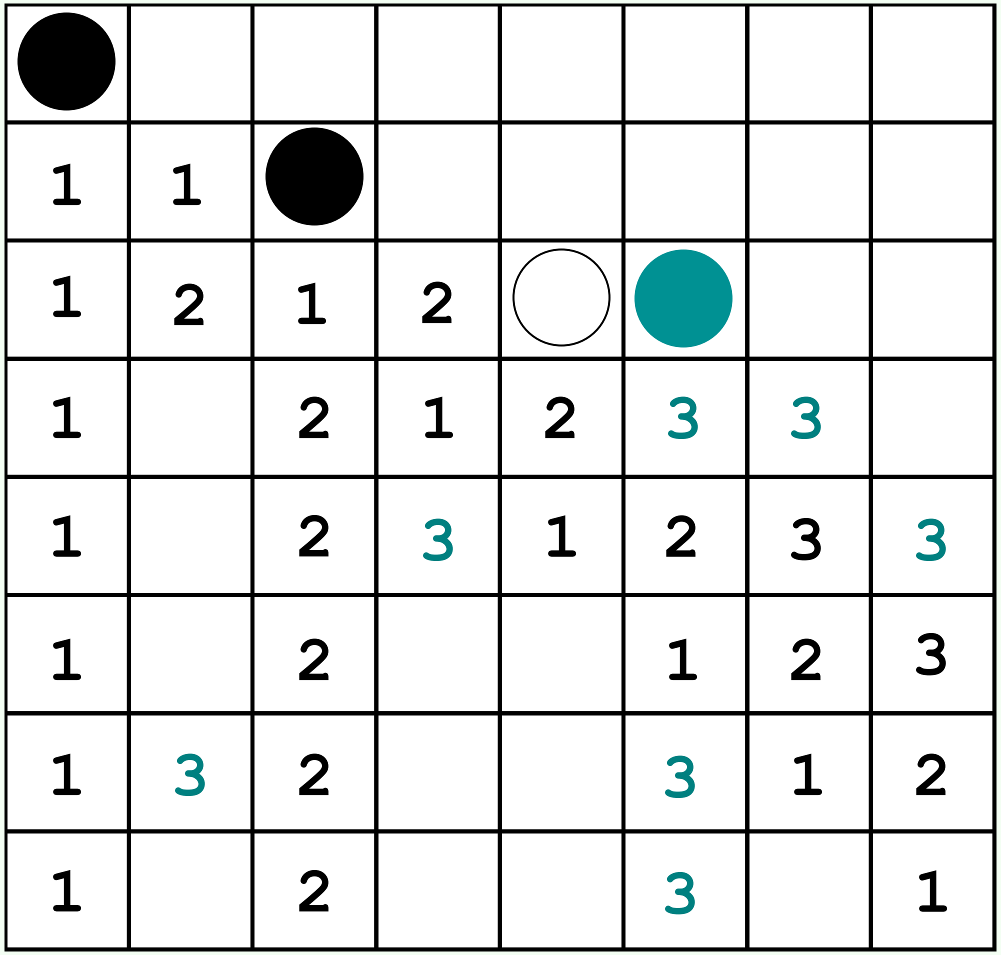
Search Methods(2a) - B+P w/Heuristics

$Q_1 = 1$

$Q_2 = 3$

$Q_3 = 5$

Impossible !



Tests

136

(324)

Backtracks

1

(14)

Tests 136

Backtracks 1

Search Methods – B+P w/Heuristics

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

Constraint Programming by Example

An example: SEND+MORE = MONEY

- Find the digits encoded by letters, where different letters stand for different digits, and the symbolic sum below stands (the leftmost digits are not zero):

```
  S E N D
+ M O R E
---------
M O N E Y
```

- Similarly to all combinatorial problems, a declarative approach (as taken by Constraint Logic Programming) solves this problem by separating the two components:

  - **Model**: What are the variables that will be chosen for the problem unknowns, and the constraints that must be satisfied

  - **Search**: What strategies are used to assign values to variables

Constraint Programming by Example

Modelling

```
   S E N D
 + M O R E
 ---------
 M O N E Y
```

- There are two main steps in modelling a problem:
  1. Choose variables to represent the unknowns
     - What are the variables
     - What values can they take
  2. Select the constraints that these variables must satisfy according to the conditions of the problem;
     - How to constrain the variables
     - Are there alternative (more efficient?) sets of constraints?
- These decisions are often interdependent as illustrated in this problem.

Constraint Programming by Example

Model 1 :

- Variables Adopted:
  - One variable for each letter (we use the letter as the name of the variable)
  - Each variable takes values in 0 to 9
- Constraints to be Satisfied:
  - All variables must be different;
  - The sum must be correct
  - No leading zeros

```
  S E N D
+ M O R E
---------
M O N E Y
```

Constraint Programming by Example

Model 1 : In Comet, this model is specified as follows

```
% // import cotfd;
enum Letters = {s,e,n,d,m,o,r,y};
range Rng = 0 .. 9;
Solver<CP> cp();
var<CP>{int} q[Letters](cp,Rng);
solve<cp>{
   cp.post(q[s] != 0);          % No leading zeros
   cp.post(q[m] != 0);
   cp.post(alldifferent(q));    % All variables are different
                                % The sum must be correct
   cp.post(1000*q[s]+100*q[e]+10*q[n]+q[d]
         +  1000*q[m]+100*q[o]+10*q[r]+q[e]
          == 10000*q[m]+1000*q[o]+100*q[n]+10*q[e]+q[y]);
} using
   labelFF(q);
```

$$\begin{array}{ccccc} C4 & C3 & C2 & C1 & \\ & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

Constraint Programming by Example

Model 2 :

- There is an alternative modelling, that represents the total sum as it is usually operated with “carries”
  - One variable for each letter (we use the letter as the name of the variable)
    - Each variable takes values in 0 to 9
  - 4 Carries
    - Each carry takes value 0 or 1

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

- Constraints to be Satisfied:
  - All variables must be different;
  - All the sums (digit by digit, including carries) must be correct
  - No leading zeros

Constraint Programming by Example

Model 2 : This alternative model can also be expressed in Comet

```
enum Letters = {s,e,n,d,m,o,r,y};
range Rng = 0 .. 9;
Solver<CP> cp();
  var<CP>{int} q[Letters](cp,Rng);
  var<CP>{int} c[1..4](cp,0..1);
solve<cp>{
  cp.post(q[s] != 0);       % No leading zeros
  cp.post(q[m] != 0);
  cp.post(alldifferent(q));  % All variables are different
                             % The sum must be correct
  cp.post(q[d] + q[e]        == q[y] +10 * c[1]);
  cp.post(q[n] + q[r] + c[1] == q[e] +10 * c[2]);
  cp.post(q[e] + q[o] + c[2] == q[n] +10 * c[3]);
  cp.post(q[s] + q[m] + c[3] == q[o] +10 * c[4]);
  cp.post(             c[4] == q[m]);
} using { labelFF(q); labelFF (c)}
```

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

Constraint Programming by Example

Model 3 :

- Even the constraints might often be expressed in alternative ways. The constraint that all letters are different can be expressed by the primitive global constraint **alldifferent(q)**, or obtained by conjunction of the individual pairwise different (!=) constraints:

```
// cp.post(alldifferent(q));

forall(i in letters, j in letters: i!=j)
     cp.post(q[i] != q[j]);
```

- As will be seen later, when available, global constraints typically lead to much more efficient execution.

Constraint Programming by Example

Enumeration :

- Once the variables are declared and the constraints posted, the constraint solver should find values for the variables in some efficient way.
- This is because the underlying constraint propagation process does not guarantee that the problem has a solution!
- It simply removes values from the domain of variables that guaranteedly do not belong to any solution.
- The enumeration is typically achieved in Comet with function **label/1**, that assigns values to the input variables and backtracks when this is impossible.
- The labelling process may be more or less efficient, depending on the heuristics used. A fairly good heuristic is the fail-first that assigns values to the variables with less values in their domains. In Comet, that may be expressed by function **labelFF/1**.
- More sophisticated heuristics may nevertheless be programmed by the user.

Constraint Programming - Finite Domains

- The efficiency obtained in solving a problem with CP depends on many issues that will be addressed in the course:
  1. Formalization of Constraint Propagation
  2. Types of constraints and their main features
  3. Alternative models
     a. Redundant Constraints
     b. Symmetry Breaking Constraints
  4. Heuristics that are most commonly used
  5. Testing these techniques with Comet in several non-trivial examples

Constraint Programming – Continuous Domains

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

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

Constraints in these continuous domains will be covered in the second part of the course, by Prof. Jorge Cruz.

Constraint Programming – Continuous Domains

A summary of this second part:

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

Constraint Programming – Continuous Domains

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

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

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

Assessment

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