U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

DEPARTMENT OF INFORMATICS ENGINEERING

Logic and Constraint Programming

Masters in Informatics and Computing Engineering
2022/2023 – 2nd Semester

Constraint Programming

Based on slides from Pedro Barahona, John Hooker, Willem-Jan van Hoeve, Luís Paulo Reis, and other authors

Daniel Castro Silva
dcs@fe.up.pt

Presentation Outline

- Introduction to Constraint Programming
- Constraint Programming
- Concepts and Formalisms
- Complexity Analysis
- Constraint Satisfaction and Optimization Problem Examples
- Consistency
- Efficiency in Constraint Programming

Introduction to Constraint Programming

Constraint Programming

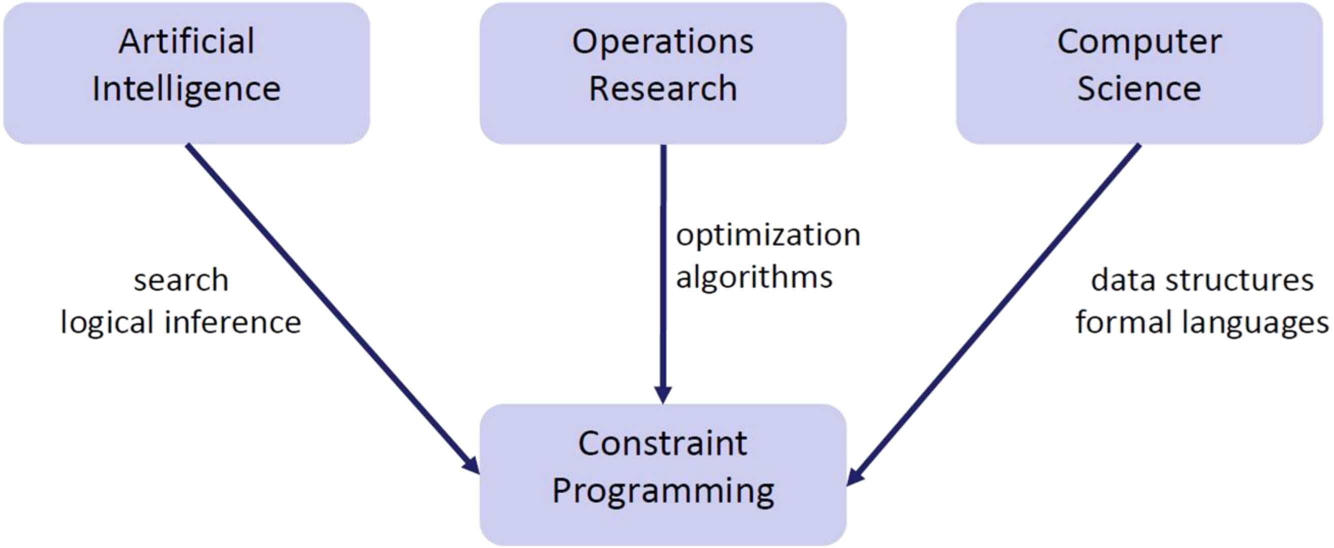
Introduction to Constraint Programming

Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.

Eugene Freuder, 1997 (‘In Pursuit of the Holy Grail’, Constraints: An International Journal, 2, 57-61)

What is Constraint Programming?

- An alternative to (more traditional) optimization methods in operations research
- Developed in the computer science and artificial intelligence communities
- Particularly successful in scheduling and logistics



Constraint (Logic) Programming

- Constraint Programming (CP) is a class of programming languages combining
  - Declarative programming
  - Efficiency of constraint resolution
- First appeared within the Logic Programming community
  - Now known as Constraint Logic Programming (CLP)
- Applied mainly to solving combinatorial search / optimization problems (typically NP-complete problems):
  - Scheduling, timetabling, resource allocation, planning, production management, etc.

Some History

- 1960s and 70s: Some early developments on related topics and applications
- 1970s: Prolog (‘*PROgrammation en LOGique*’)
  - Colmerauer, Roussel, Kowalsky et al.
- 1980s: Constraint (Logic) Programming
  - Prolog III, CHIP, …
  - Constraint and Generate vs Generate and Test
- 1990s: Constraint Programming and first industrial solvers (ILOG, …) and applications
- 2000s: Global constraints, modeling languages, …

Applications and Successes

- Some early Commercial Applications
  - Lufthansa
    - Short-term staff planning
  - Renault
    - Short-term production planning
  - Nokia
    - Software configuration for mobile phones
  - Airbus
    - Cabin layout
  - Siemens
    - Circuit verification

  - See Helmut Simonis (1999), Building Industrial Applications with Constraint Programming, in Constraints in Computational Logics (CCL 1999)

Applications and Successes

- Job shop scheduling
- Production scheduling (chemicals, oil refining, aviation, steel, lumber, ...)
- Maintenance planning
- Warehouse management
- Transport scheduling (food, nuclear fuel, ...)
- Airline crew rostering and scheduling
- Nurse scheduling
- Shift planning
- Course timetabling
- Cellular frequency assignment

Applications and Successes

- Sports Scheduling
  - Several types of leagues/tournaments
- 1997/1998 ACC basketball league (9 teams)
  - Various complicated side constraints
  - All 179 solutions were found in 24h using enumeration and integer linear programming [Nemhauser & Trick, 1998]
  - All 179 solutions were found in less than a minute using constraint programming [Henz, 1999, 2001]

Applications and Successes

- Operations Scheduling

- Hong Kong Airport
  - Gate allocation at Hong Kong International Airport
  - System was implemented in only four months, and includes constraint programming technology (ILOG)
  - Schedules ~1100 flights per day
    (over 70 million passengers in 2016)

Applications and Successes

- Operations Scheduling

- Port of Singapore
  - One of world's largest container transshipment hubs
  - Links shippers to a network of 200 shipping lines with connections to 600 ports in 123 countries
  - Need to assign yard locations and loading plans under operational and safety requirements
  - Yard planning system, based on constraint programming (ILOG)

Applications and Successes

- Operations Scheduling

- Netherlands Railways
  - Among the world's densest rail networks, with 5,500 trains per day
  - Constraint programming as part of railway planning software, used to design a new timetable from scratch (2008)
  - More robust and effective schedule, with $75M additional annual profit
  - INFORMS Edelman Award winner (2008)

Constraint Programming

Constraint Programming

Constraint Programming

- The programmer doesn’t specify the steps to solve the problem
- Instead, the focus is on modeling the problem, specifying
  - **Variables** that represent the structure of the problem
  - Values that the different variables can take - **domains**
  - **Constraints** between variables that must hold in valid solutions
  - Optionally, a **search strategy** to be used by the solver
- **Global constraints** are used to exploit problem structure
- **Filtering** and constraint **propagation** can reduce the search space
  - filtering = reduce variable domains
  - propagation = propagate domain to other constraints

Example

- $3A + B + C = 10$
- A, B, C pairwise distinct
- $A, B \in \{1,2\}$, $C \in \{1,2,3\}$

- Purely procedural approach

```
for A = 1,2:
    for B = 1,2:
        if A ≠ B then
            for C = 1,2,3:
                if A ≠ C and B ≠ C then
                    if 3A + B + C = 10 then
                        print A, B, C
```

Example

- $3A + B + C = 10$
- A, B, C pairwise distinct
- $A, B \in \{1,2\}$, $C \in \{1,2,3\}$

- Purely declarative approach

```
3A + B + C = 10
A ≠ B
A ≠ C
B ≠ C
A, B ∈ {1,2}, C ∈ {1,2,3}
```

How to implement?

Example

- $3A + B + C = 10$
- A, B, C pairwise distinct
- $A, B \in \{1,2\}$, $C \in \{1,2,3\}$

- CP model

```
A,B ∈ {1,2}, C ∈ {1,2,3}
all_distinct(A, B, C)
3A + B + C = 10
```

This **global constraint** (all_distinct) enforces $A \neq B$, $A \neq C$, and $B \neq C$.

CP Model

- CP model

```
A,B ∈ {1,2}, C ∈ {1,2,3}
all_distinct(A, B, C)
3A + B + C = 10
```

- The model is intuitive and looks **declarative**
  - It consists of variables/domains and constraints
  - Constraints can be written in any order
- But each constraint invokes a **procedure**
  - The procedure reduces the search space by **filtering** and **propagation**

CP Model - Filtering

- CP model

```
A,B ∈ {1,2}, C ∈ {1,2,3}
all_distinct(A, B, C)
3A + B + C = 10
```

- Variable Domains:

```
A ∈ {1, 2}
B ∈ {1, 2}
C ∈ {1, 2, 3}
```

- Use the *all_distinct* constraint to **filter** the domains (remove infeasible values)
  - A and B must use the values 1,2

CP Model - Filtering

- CP model

```
A,B ∈ {1,2}, C ∈ {1,2,3}
all_distinct(A, B, C)
3A + B + C = 10
```

- Variable Domains:

```
A ∈ {1, 2}
B ∈ {1, 2}
C ∈ { ,  , 3}
```

- Use the *all_distinct* constraint to **filter** the domains (remove infeasible values)
  - A and B must use the values 1,2
  - So we **filter** these values from the domain of C

CP Model - Filtering

- CP model

```
A,B ∈ {1,2}, C ∈ {1,2,3}
all_distinct(A, B, C)
3A + B + C = 10
```

- Variable Domains:

```
A ∈ {1, 2}
B ∈ {1, 2}
C ∈ { ,  , 3}
```

- Use the *all_distinct* constraint to **filter** the domains (remove infeasible values)
- Removing all infeasible values achieves **domain consistency**

CP Model - Propagation

- CP model

```
A,B ∈ {1,2}, C ∈ {1,2,3}
all_distinct(A, B, C)
3A + B + C = 10
```

- Variable Domains:

```
A ∈ {1, 2}
B ∈ {1, 2}
C ∈ { ,  , 3}
```

- We now **propagate** the reduced domains to other constraints
  - Arithmetic constraints can be manipulated into inequalities to find limits
  - Must have $3A \geq 10 - \max\{1,2\} - \max\{3\}$ ⇔ $3A \geq 5$ ⇔ $A \geq 1{,}667$

Domain of B Domain of C

CP Model - Propagation

- CP model

```
A,B ∈ {1,2}, C ∈ {1,2,3}
all_distinct(A, B, C)
3A + B + C = 10
```

- Variable Domains:

```
A ∈ { , 2}
B ∈ {1, 2}
C ∈ { , , 3}
```

- We now **propagate** the reduced domains to other constraints
  - Arithmetic constraints can be manipulated into inequalities to find limits
  - Must have $3A \geq 10 - \max\{1,2\} - \max\{3\}$ ⇔ $3A \geq 5$ ⇔ $A \geq 1{,}667$
  - We can filter the domain of A

CP Model - Propagation

- CP model

```
A,B ∈ {1,2}, C ∈ {1,2,3}
all_distinct(A, B, C)
3A + B + C = 10
```

- Variable Domains:

```
A ∈ { , 2}
B ∈ {1,  }
C ∈ { ,  , 3}
```

- We again **propagate** to the *all_distinct* constraint
  - We can filter the domain of B

CP Model - Solution Found

- CP model

```
A,B ∈ {1,2}, C ∈ {1,2,3}
all_distinct(A, B, C)
3A + B + C = 10
```

- Variable Domains:

```
A ∈ { , 2}
B ∈ {1,  }
C ∈ { ,  , 3}
```

- Because each domain is a **singleton**, we have a solution
  - No more propagation needed

CP Model - Search

- CP model

```
A,B ∈ {1,2}, C ∈ {1,2,3}
all_distinct(A, B, C)
3A + B + C = 10
```

- Variable Domains:

```
A ∈ { , 2}
B ∈ {1, 2}
C ∈ { , , 3}
```

- Search is often required
  - If B had no filtered domain, we would have to search
    - Choose B=1 and repeat process
    - Choose B=2 and repeat process

Global Constraints

- Global constraints like *all_distinct* **exploit problem structure**
  - Filtering for a global constraint takes advantage of the “global” structure of the elementary constraints it represents
  - This is more effective than propagating the individual constraints

```
A,B ∈ {1,2}, C ∈ {1,2,3}
A != B
A != C
B != C

A,B ∈ {1,2}, C ∈ {1,2,3}
```

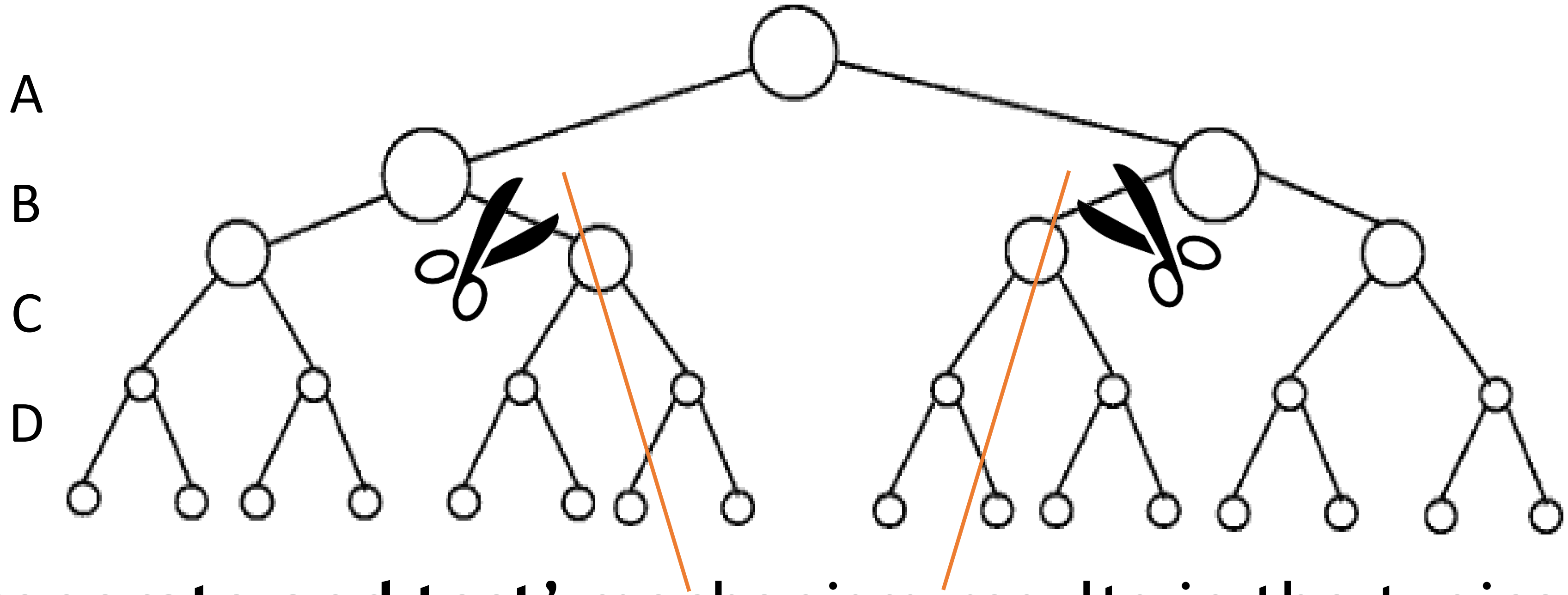
```
A,B ∈ {1,2}, C ∈ {1,2,3}
all_distinct(A, B, C)

A,B ∈ {1,2}, C ∈ {3}
```

Generate and Test vs Constrain and Generate

- In CP, the Prolog’s **unification** mechanism is replaced by a **constraint manipulation** mechanism for a given domain
- Prolog’s standard (and not so efficient) ‘**generate and test**’ search is replaced by more intelligent search techniques (consistency maintenance techniques), resulting in a ‘**constrain and generate**’ mechanism

Generate and Test vs Constrain and Generate



- The ‘**generate and test**’ mechanism results in the typical depth-first search (in the full tree)
- The ‘**constrain and generate**’ mechanism allows eliminating branches of the tree that are known not to lead to valid solutions

Usual CP Program Structure

- Declaration of variables and domains
- Specification of constraints
    - Deterministic filtering and propagation methods reduce variable domains
- Search for solution
    - When no more filtering and propagation can be done, the solver searches for a possible solution

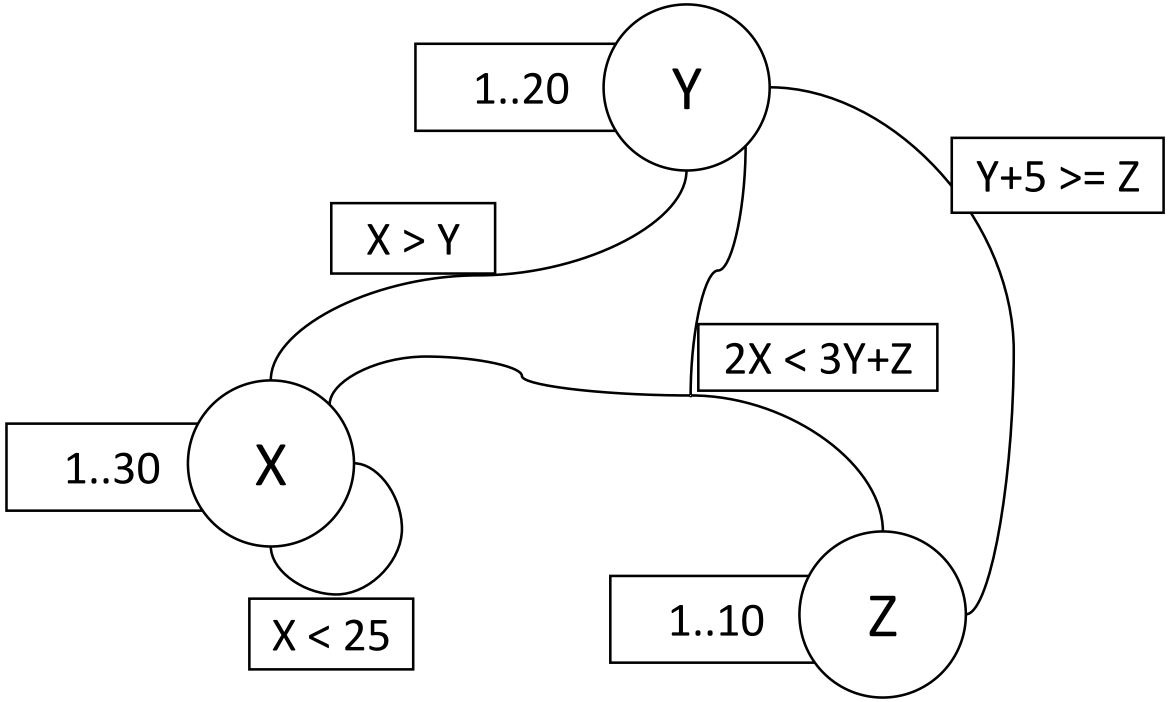
CP Solving

- The solution process of CP interleaves
  - **Domain filtering**
    - Remove inconsistent values from the domains of the variables, based on individual constraints
  - **Constraint propagation**
    - Propagate the filtered domains through the constraints, by re-evaluating them until there are no more changes in the domains
- Possibly (most of the times) followed by **search**
  - Implicitly all possible variable-value combinations are enumerated, but the search tree is kept small due to domain filtering and constraint propagation

Constraint Programming

- A model can be represented by a hyper-graph, where the nodes represent variables (with their associated domains), and the constraints are (hyper-)edges connecting the nodes

```
X in 1..30
Y in 1..20
Z in 1..10
X < 25
X > Y
Y+5 >= Z
2X < 3Y + Z
```



A Small Exercise

```
A,B,C ∈ {1,2,3,4,5}
A < 5
B < A
C > B
C < A
```

Constraint Programming

- Can be used to
  - Determine **if** a problem has a solution
  - Find (any) **one** solution to a problem
  - Find **all** solutions to a problem
  - Find the **optimal** solution to a problem, according to an objective function, defined in terms of a variable subset

Advantages of CP

- Good at scheduling, logistics
  - …where other optimization methods may fail
- Adding messy constraints makes the problem easier to be solved
  - The more constraints, the better (usually prunes the search space)
- More powerful modeling paradigm
  - Simpler models (due to global constraints)
  - Constraints convey problem structure to the solver
  - Clarity and brevity of programs
    - Representation of the problem close to natural language description
    - Advantages in terms of development time, program verification, maintainability, ...

Concepts and Formalisms

Constraint Programming

Constraint Programming

- A Constraint Satisfaction Problem (CSP) is modeled through
  - **Variables** that represent the different aspects of the problem, together with their respective **domains**
  - **Constraints** that limit the values each variable may take (within their respective domains)

- A **solution** to a CSP is an attribution of a domain value to each variable (**labeling**), such that all constraints are satisfied
  - Found through a systematic **search** (usually guided by a heuristic), of all possible value attributions

Constraint Programming

- More formally, a CSP is a tuple $\langle V, D, C \rangle$, where
  - $V = \{x_1, x_2, \dots, x_n\}$ is the set of domain variables
  - D is a function that maps each variable in V to a set of possible values (the variable domain)
    - $D: V \rightarrow finite\ set\ of\ values$
    - $D_{x_i}: domain\ of\ x_i$
  - $C = \{C_1, C_2, \dots, C_n\}$ is the set of constraints that affect (a subset of) variables in V
    - For each constraint $c_i \in C$
      - $Vars(c_i)$ is the set of variables involved in c_i
      - Symmetrically, $Cons(x_i)$ is the set of constraints in which variable $x_i \in V$ is involved

Constraint Programming

- A solution is an attribution of values to variables in V respecting all constraints:
  - $Sol = \{\langle x_1, v_1\rangle, \langle x_2, v_2\rangle, \dots, \langle x_n, v_n\rangle\}$:
    - $\forall x_i \in V \left(\langle x_i, v_i\rangle \in Sol \ \wedge \ v_i \in D_{x_i}\right)$
    - $\forall c_k \in C \ satisfies(Sol, c_k)$

Constraint Programming

- A **Constraint Optimization Problem** (COP) is a CSP with an added objective function to be optimized (maximized / minimized)

- A solution to a COP is an attribution of a domain value to each variable such that all constraints are satisfied and there is no other attribution that results in a larger(/smaller) value for the evaluation function

Constraint Programming

- A **label** is a *Variable-Value* pair, where *Value* belongs to the domain of *Variable*
- A **compound label** constitutes a partial solution to a CSP/COP and is composed of a set of labels for different variables

Constraint Programming

- A **constraint** involves a set of variables and limits the compound labels for those variables
  - A constraint C_{ijk} involving variables X_i, X_j e X_k defines a subset of the Cartesian product of the domains of the variables
    - $C_{ijk} \subseteq dom(X_i) \times dom(X_j) \times dom(X_k)$
- Constraints can be expressed:
  - In extension, through the enumeration of all admissible compound labels
    - $C_{12} = \{ \langle 1,2 \rangle, \langle 2,3 \rangle, \langle 3,4 \rangle, \langle 4,5 \rangle, \langle 5,6 \rangle, \langle 6,7 \rangle, \langle 7,8 \rangle, \langle 8,9 \rangle \}$
  - Implicitly, through an equation or procedure that determines the compound labels
    - $C_{12} = (X_1 = X_2 - 1)$

Constraint Programming

- The **arity** of a constraint *C* is the number of variables involved in that constraint, ie, the cardinality of *Vars(C)*

- Constraints can be of any arity, but unary and binary constraints are usually considered when modeling a CSP/COP
  - Any constraint of higher dimensionality can be converted into a set of binary constraints
    - As such, binary CSPs are representative of all CSPs
  - Several concepts and algorithms are appropriate for binary constraints

Constraint Programming

- Conversion to Binary Constraints:
  - An n-ary Constraint C, defined by k compound labels in its variables X_1 to X_n, is equivalent to n binary constraints, B_i, through the introduction of a new variable Z, whose domain is the set 1 to k
- Rationale:
  - The k n-ary labels can be ordered in any order
  - Each of the binary constraints B_i relates the new variable Z with variable X_i
  - The compound label $\{X_i\text{-}v_i, Z\text{-}z\}$ belongs to constraint B_i iff $X_i\text{-}v_i$ belongs to the i^{th} compound label that defines C

Constraint Programming

- Example:
  - Given variables X_1, X_2 and X_3, with domains 1 to 3, the ternary constraint C imposes different values for all three variables, being composed of 6 compound labels:

$C(X_1, X_2, X_3) = \{ \langle 1,2,3\rangle, \langle 1,3,2\rangle, \langle 2,1,3\rangle, \langle 2,3,1\rangle, \langle 3,1,2\rangle, \langle 3,2,1\rangle \}$

  - Each of the labels can have an associated value from 1 to 6:
    - 1: $\langle 1,2,3\rangle$, 2: $\langle 1,3,2\rangle$, 3: $\langle 2,1,3\rangle$, ..., 6: $\langle 3,2,1\rangle$
  - The following binary constraints B_1 to B_3 are equivalent to the original ternary constraint C:
    - $B_1(Z, X_1) = \{ \langle 1,1\rangle, \langle 2,1\rangle, \langle 3,2\rangle, \langle 4,2\rangle, \langle 5,3\rangle, \langle 6,3\rangle \}$
    - $B_2(Z, X_2) = \{ \langle 1,2\rangle, \langle 2,3\rangle, \langle 3,1\rangle, \langle 4,3\rangle, \langle 5,1\rangle, \langle 6,2\rangle \}$
    - $B_3(Z, X_3) = \{ \langle 1,3\rangle, \langle 2,2\rangle, \langle 3,3\rangle, \langle 4,1\rangle, \langle 5,2\rangle, \langle 6,1\rangle \}$

Constraint Programming

- **Hard Constraints** are those that must be met
  - All constraints in a CSP are hard constraints

- **Soft Constraints** are those that can be broken
  - There must be costs associated with breaking these constraints, adding up to the evaluation function of the COP
  - Relaxation can be achieved with soft constraints
    - Increases complexity of the model

Constraint Programming

- Constraints can be linear or non-linear
  - $2X + 4 < 3Y + Z$
  - $WZ + 3X > Y - Z*Z$

- CSPs can be modeled in different domains:
  - Booleans (SAT)
  - Integers (finite domains)
  - Intervals (scheduling)
  - Complex Numbers
  - Rationals
  - Reals

Constraint Programming

- The difficulty of solving a CSP is usually related to two factors:
  - Density of the constraint network
  - Difficulty of satisfying constraints (which can be approximated by constraint tightness)

- The difficulty of solving a CSP is different from the difficulty of the problem itself
  - Sometimes a difficult problem can easily be proven impossible to be solved

Constraint Programming

- The **density of the constraint network** is the ratio between then number of edges in the network and the total number of edges in a full network with the same number of nodes
  - Usually a higher density implies a higher difficulty in solving the problem, as there are more constraints that possibly invalidate solutions

Constraint Programming

- The **tightness of a constraint** C_{ij} is defined as the ratio between the number of solutions (X_i-v_i, X_j-v_j) that satisfy the constraint and the cardinality of the Cartesian product of the variable domains
  - The same concept if true for non-binary constraints
  - The notion of tightness can also be generalized for the entire problem

Constraint Programming

- Since difficulty in solving a CSP is related to these two dimensions, usually algorithms and models are tested with randomly generated instances of the problem, parameterized by the number of nodes (variables), arcs (constraints), density of the constraint network and constraint tightness

- CSPs usually show a phase transition separating problems that are easily solved from problems that are easily proven to have no solution
  - The problems in between these two are the really complex ones

Complexity Analysis

Constraint Programming

Complexity Analysis

- The difficulty in solving CSPs (and COPs) resides in their exponential complexity
- Boolean Domain
  - Number of variables: n
  - Cardinality of domain: 2
  - Search space: 2^n
- Finite Domain
  - Number of variables: n
  - Cardinality of domain: d
  - Search space: d^n
- Rational / Real Domain
  - In theory, potentially infinite solutions
  - In practice, finite number limited to used precision
  - Different methods from those used in finite domains

Complexity Analysis

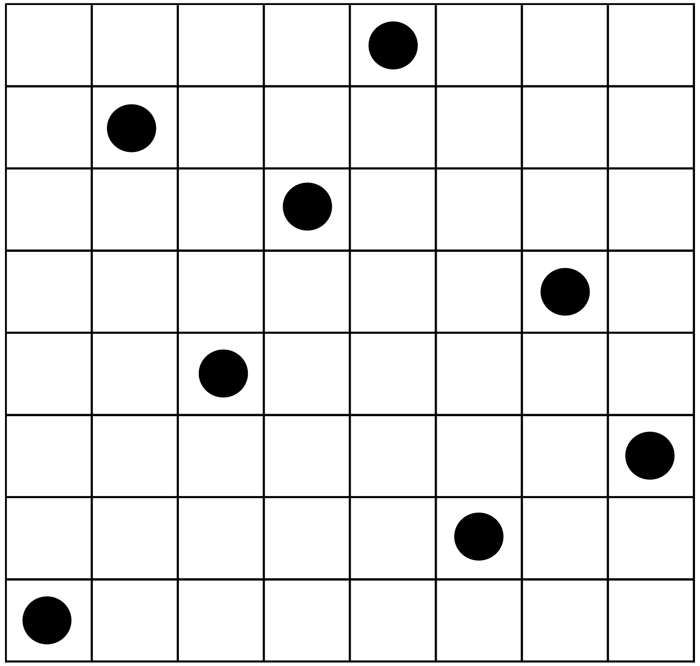
- Complexity grows exponentially
- Interesting problems are usually NP-complete
- Time for exhaustive search of the d^n possible solutions:

  (assuming a duration of 1 μs for each elementary operation)

d \ n	10	20	30	40	50	60
2	1 msec	1 sec	18 min	12,7 days	35,7 years	365 centuries
3	50 msec	1 hour	6,5 years	3855 centuries		
4	1 sec	12,6 days	365 centuries			
5	9,8 sec	1103 days	295 Kcenturies			
6	1 min	116 years				

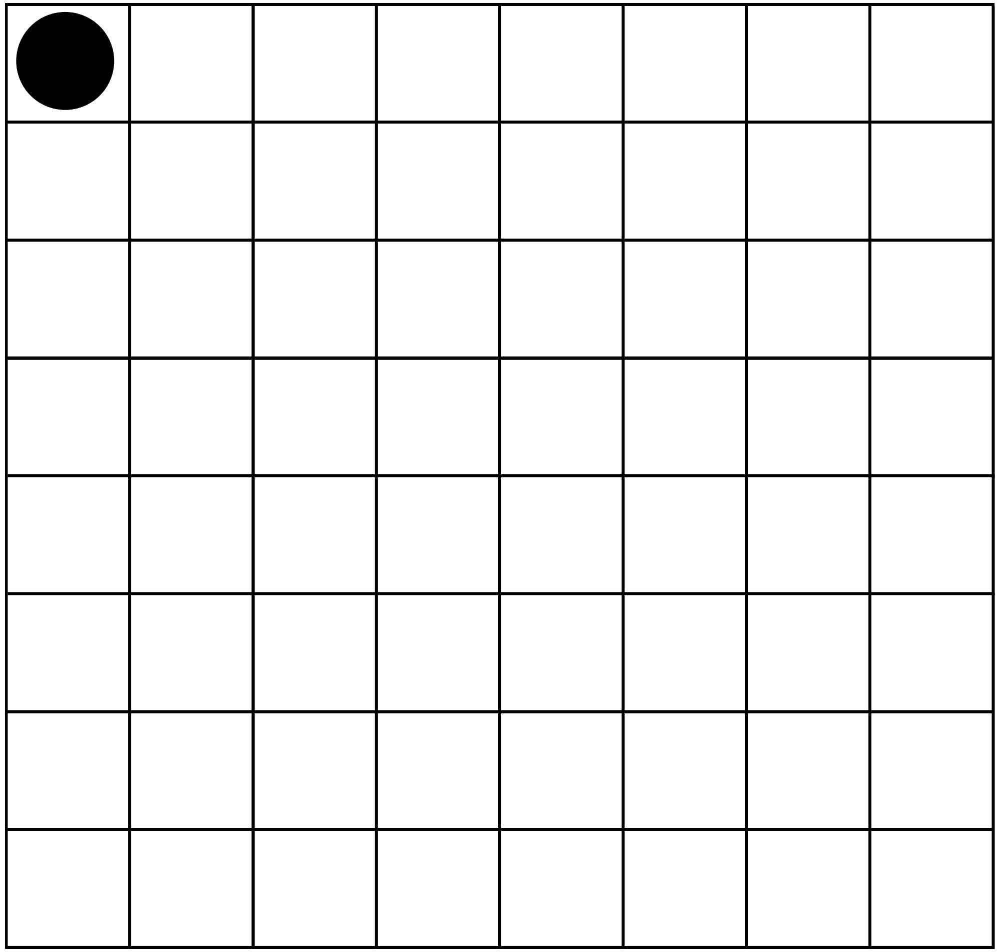
Example: N-Queens

- Fill in an NxN board with N chess queens such that no two queens attack each other
  - Two queens attack each if they are:
    - On the same line
    - On the same column
    - On the same diagonal



- Strategy:
  - Consider exactly one queen per each line of the board

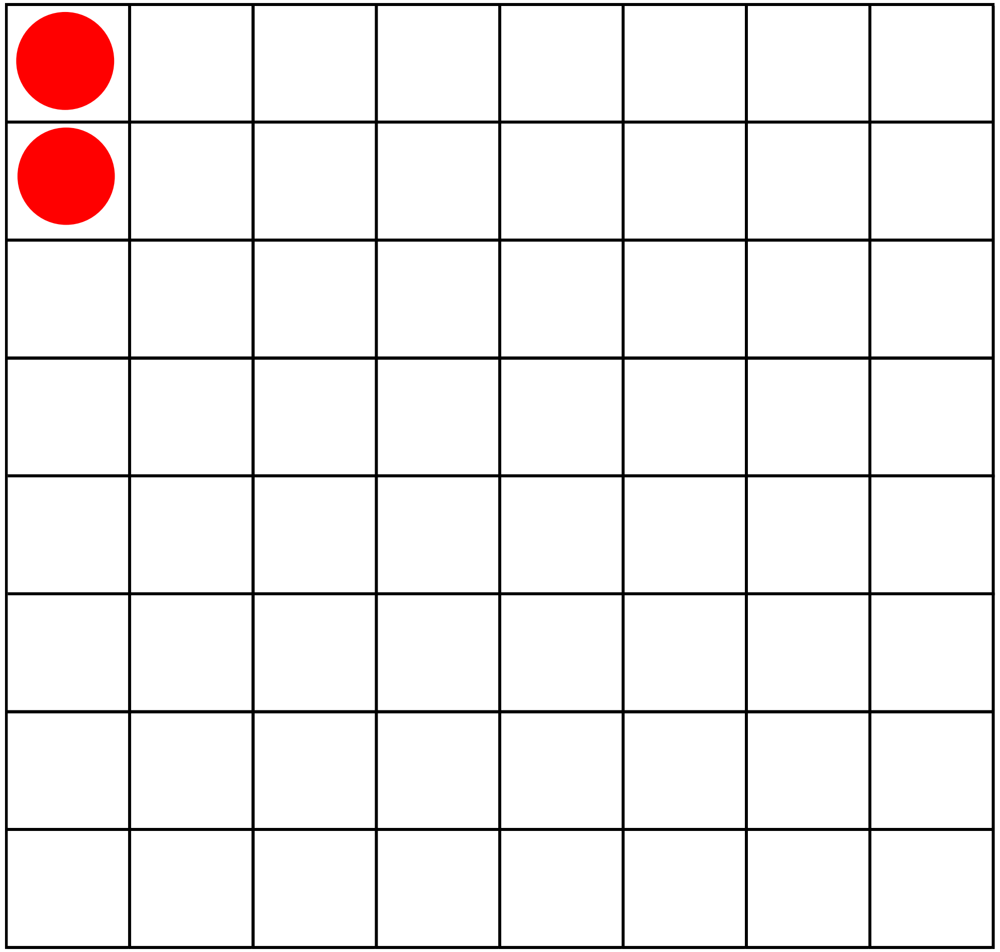
Backtracking



Tests 0 Backtracks 0

Backtracking

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

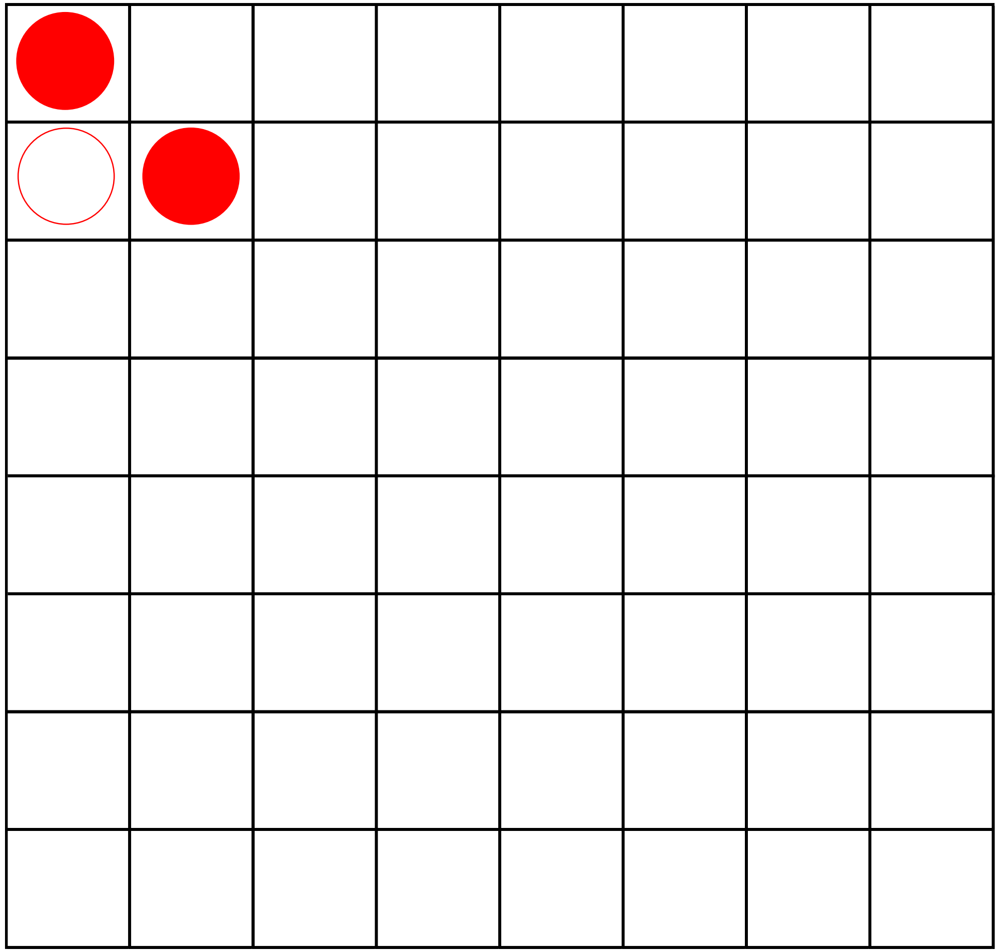


Tests 0 + 1 = 1

Backtracks 0

Backtracking

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

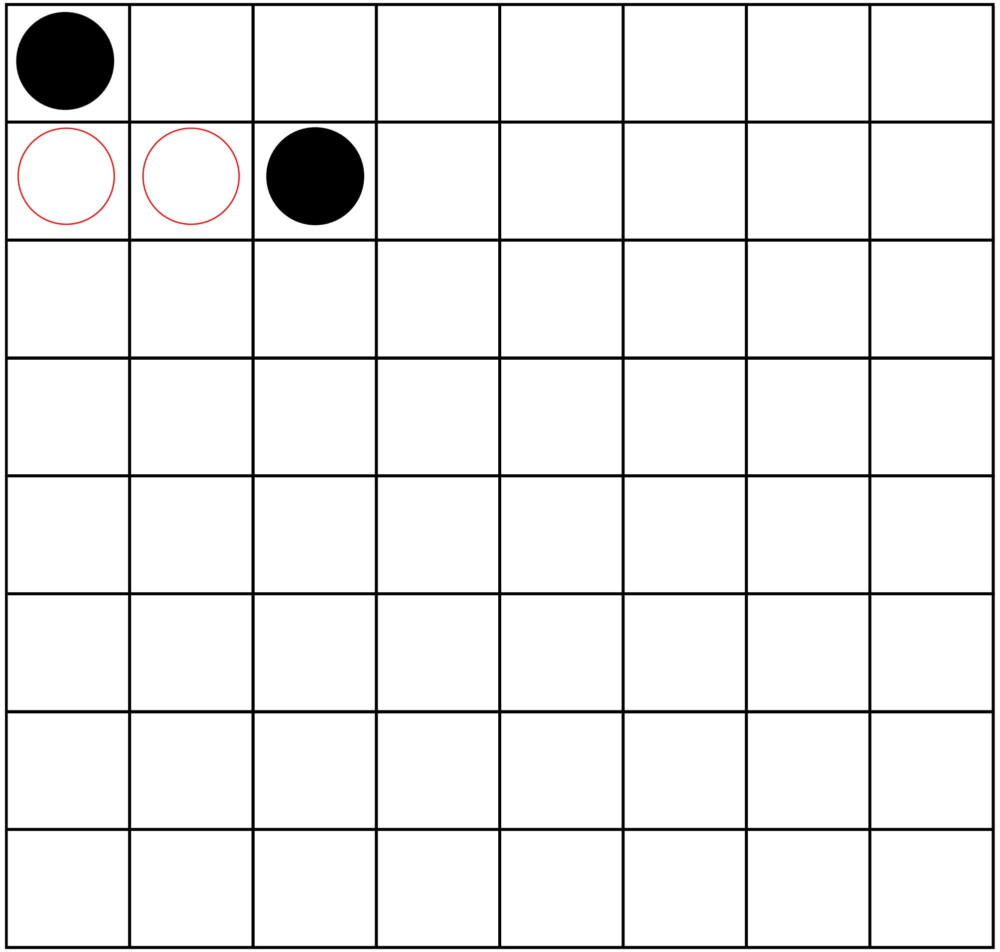


Tests 1 + 1 = 2

Backtracks 0

Backtracking

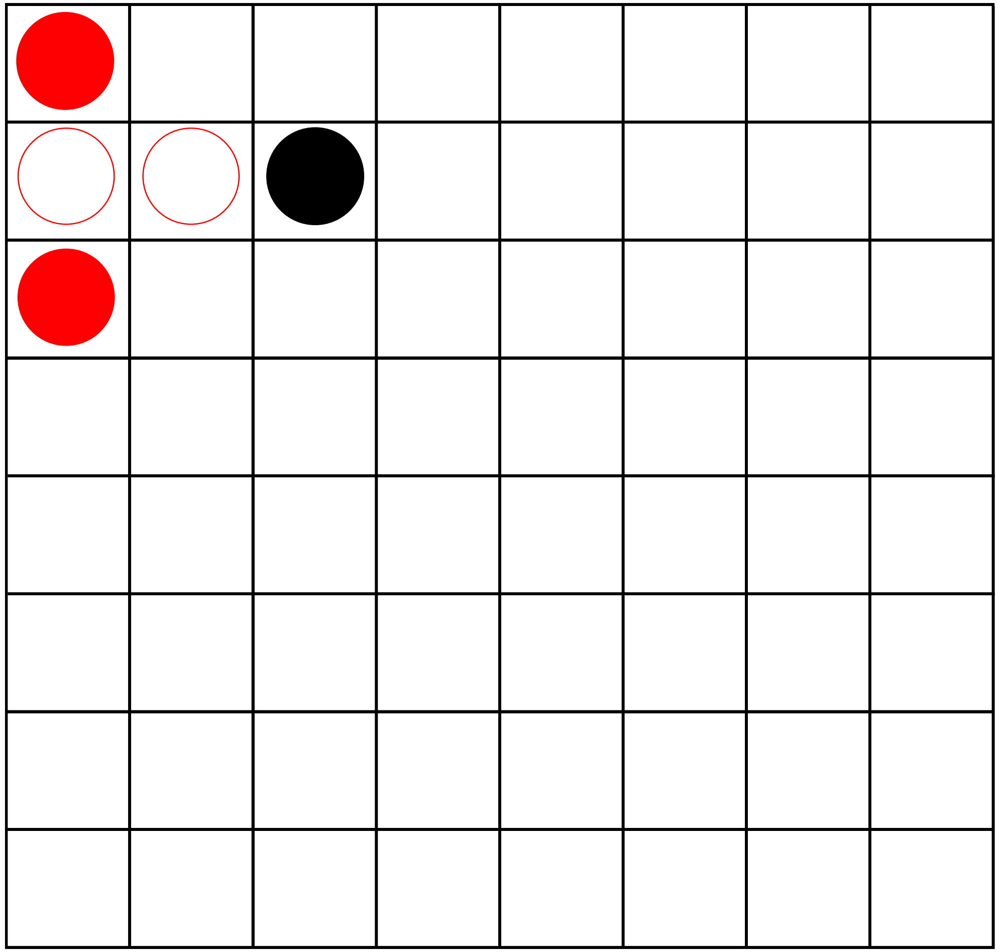
```
Q1 \= Q2,  L1+Q1 \= L2+Q2,  L1+Q2 \= L2+Q1.
```



Tests 2 + 1 = 3

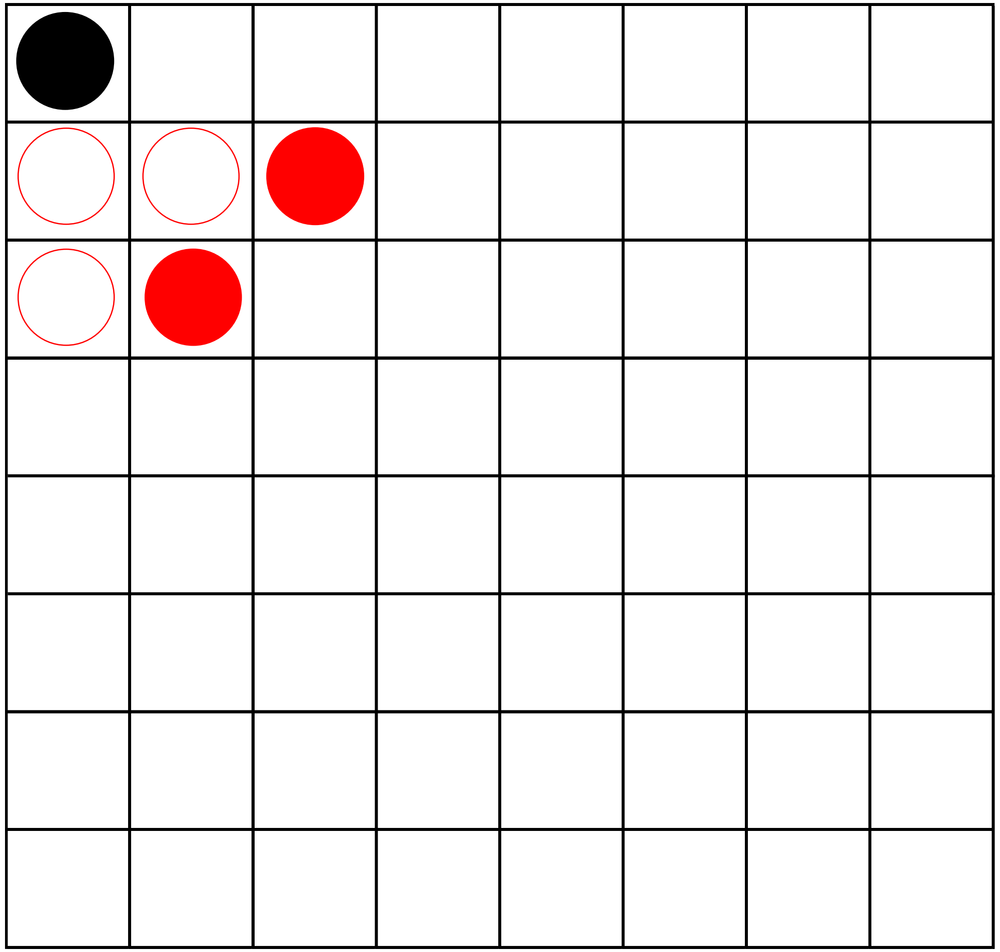
Backtracks 0

Backtracking



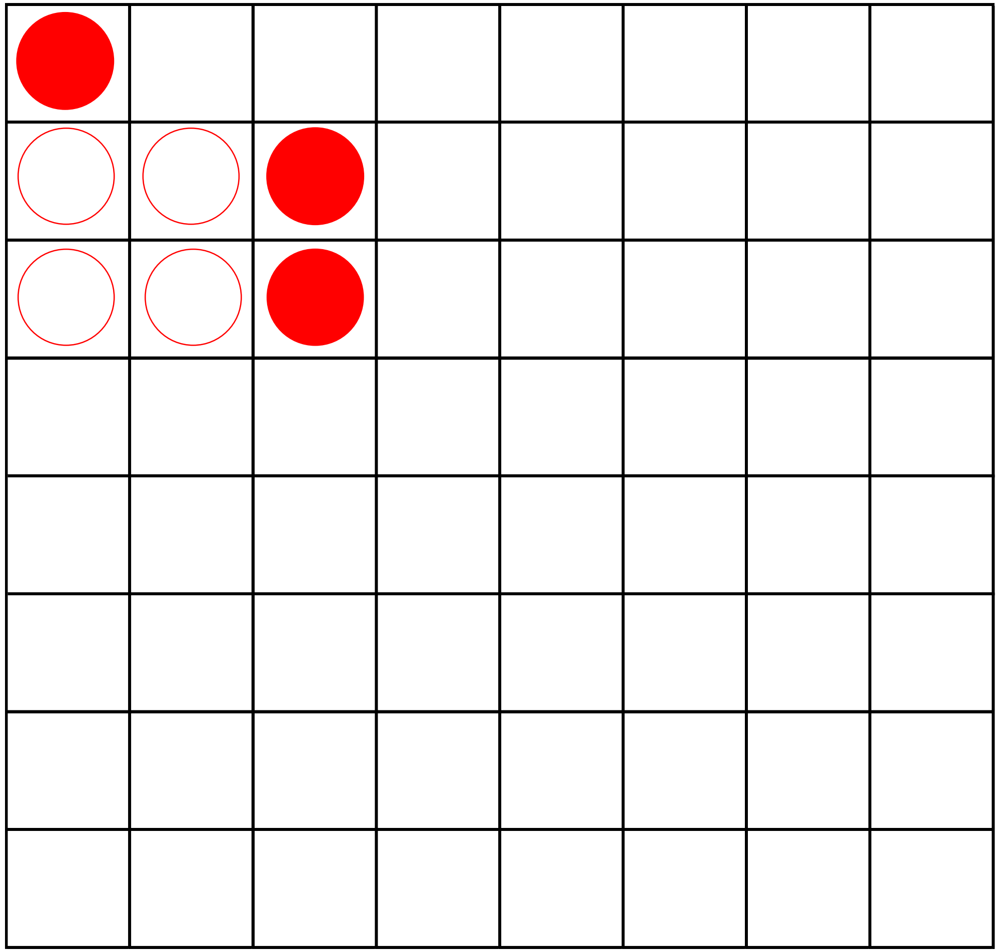
Tests 3 + 1 = 4 Backtracks 0

Backtracking



Tests 4 + 2 = 6 Backtracks 0

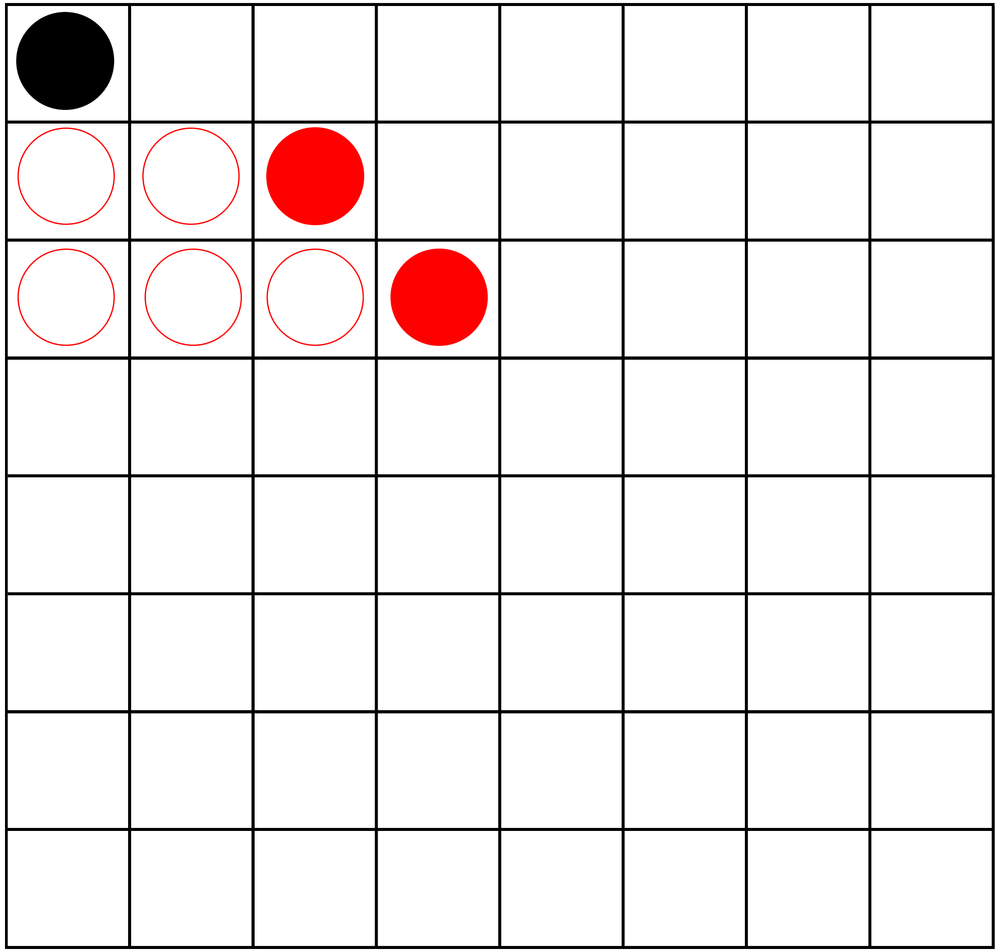
Backtracking



Tests 6 + 1 = 7

Backtracks 0

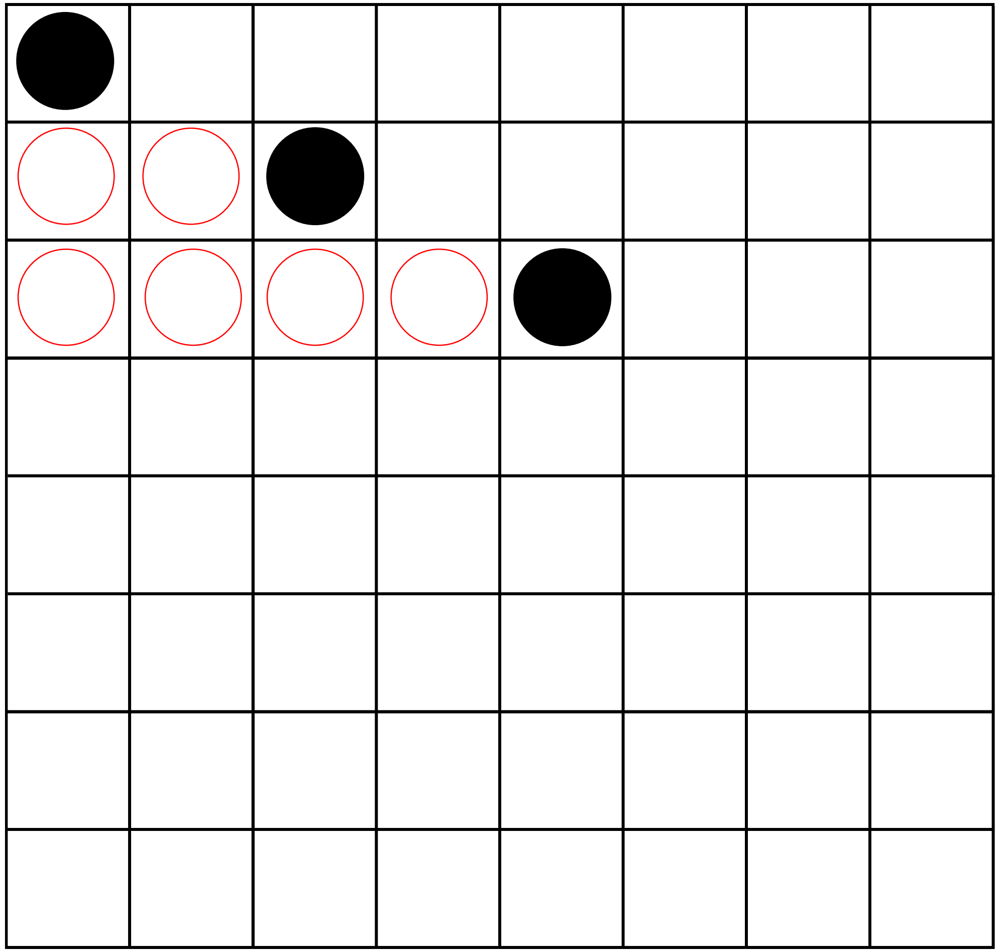
Backtracking



Tests 7 + 2 = 9

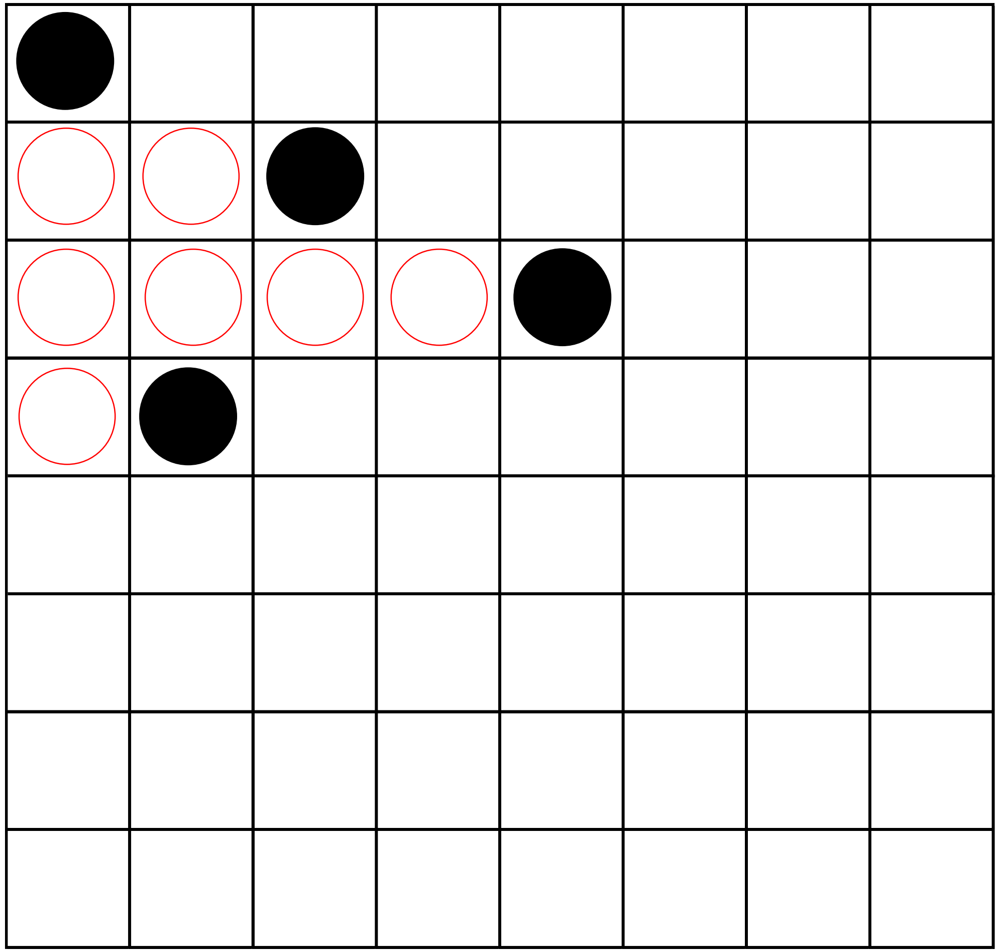
Backtracks 0

Backtracking



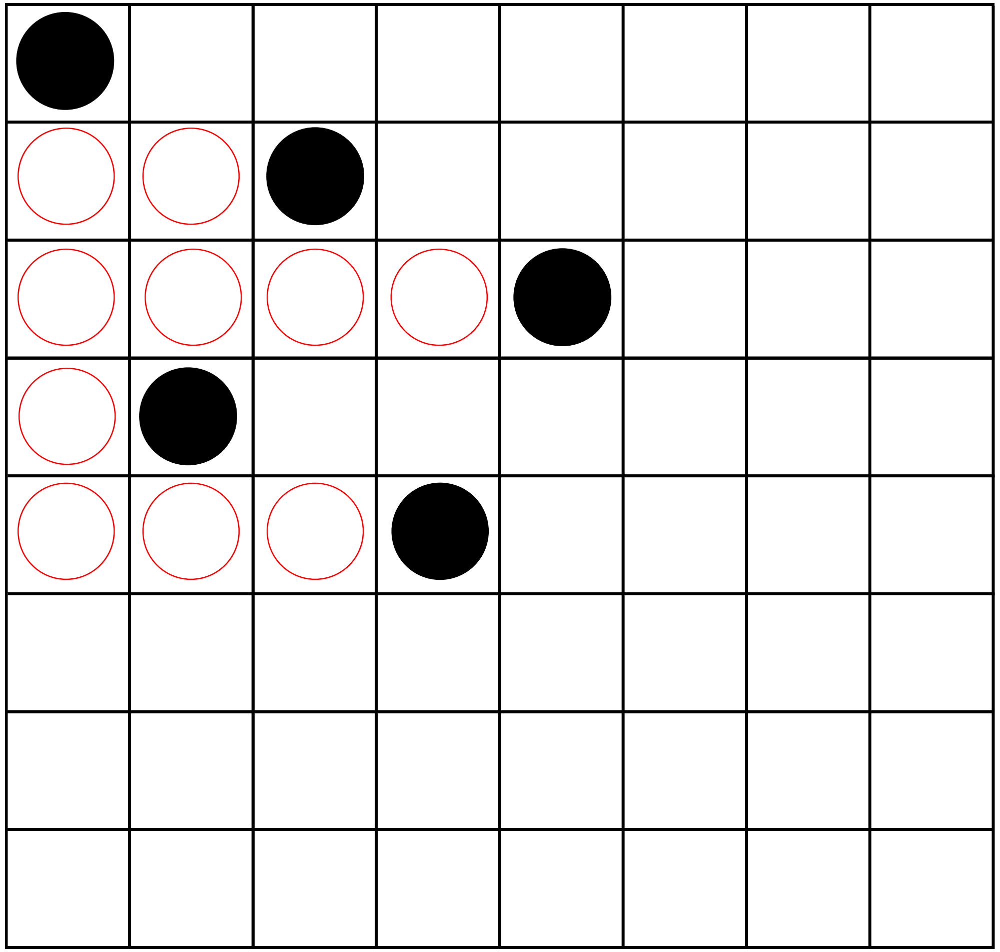
Tests 9 + 2 = 11 Backtracks 0

Backtracking



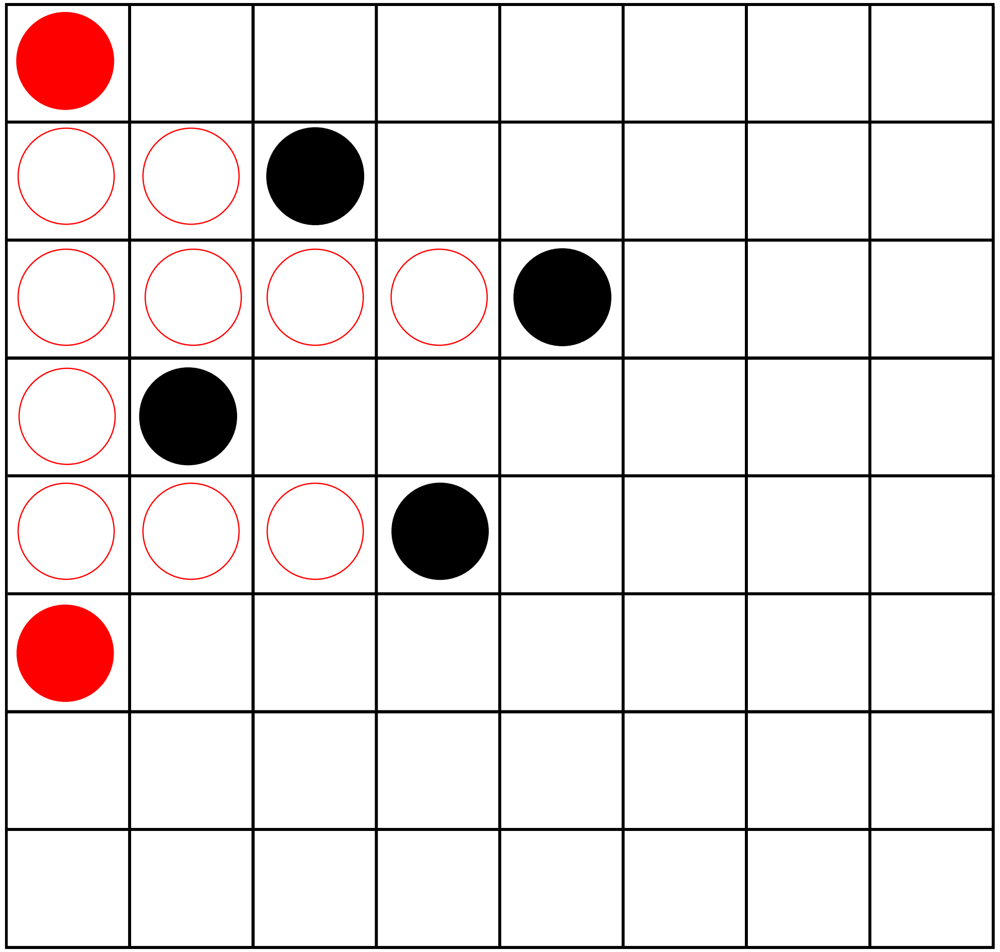
Tests 11 + 1 + 3 = 15 Backtracks 0

Backtracking



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

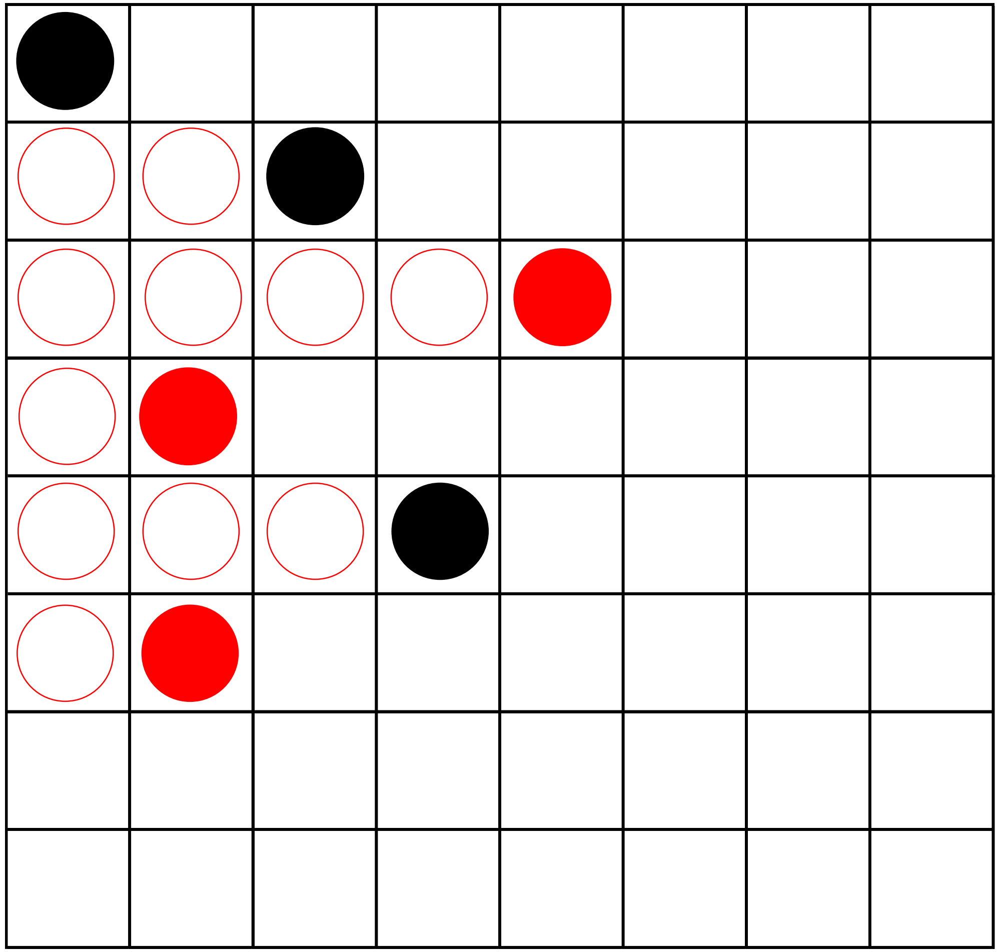
Backtracking



Tests 26 + 1 = 27

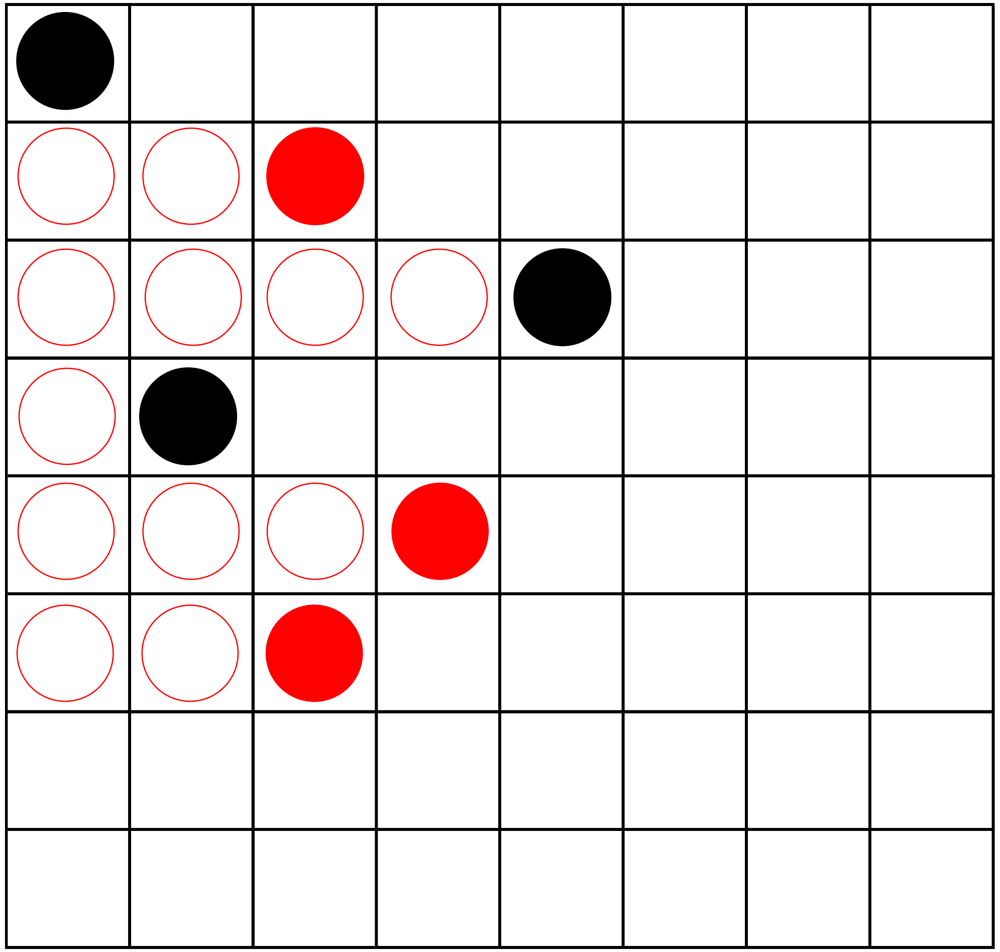
Backtracks 0

Backtracking



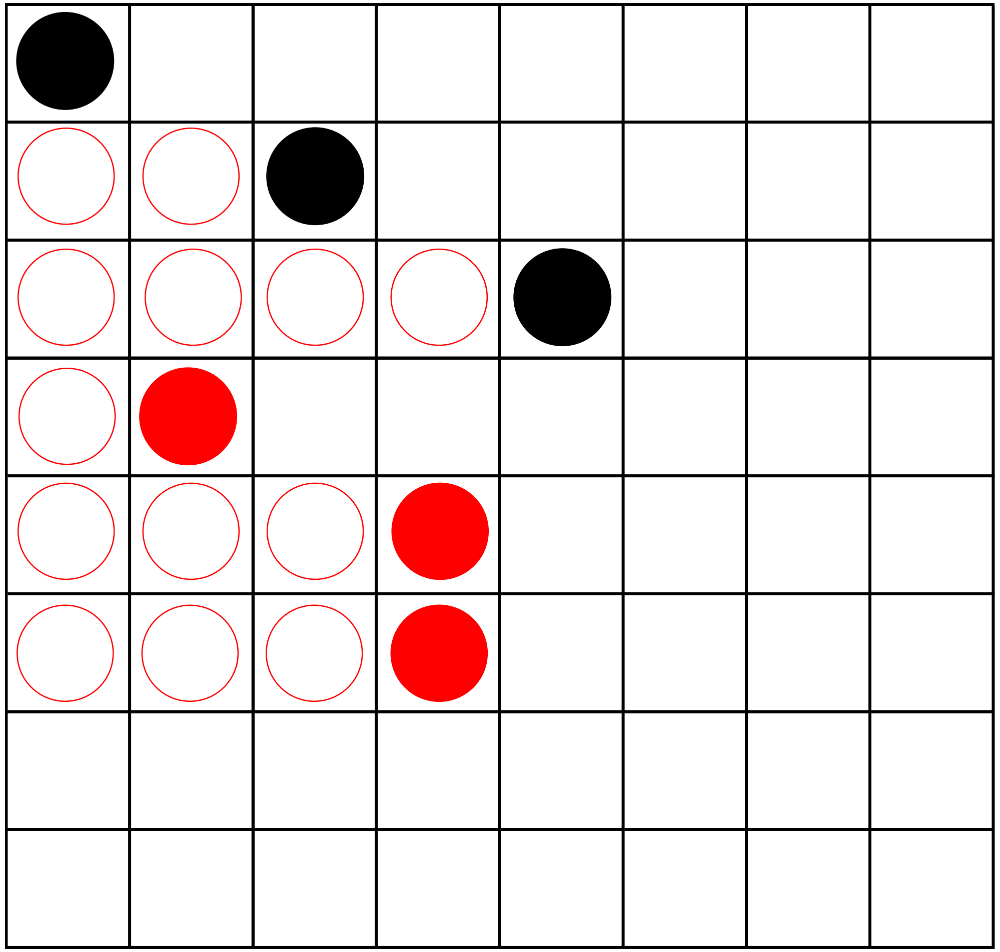
Tests 27 + 3 = 30 Backtracks 0

Backtracking



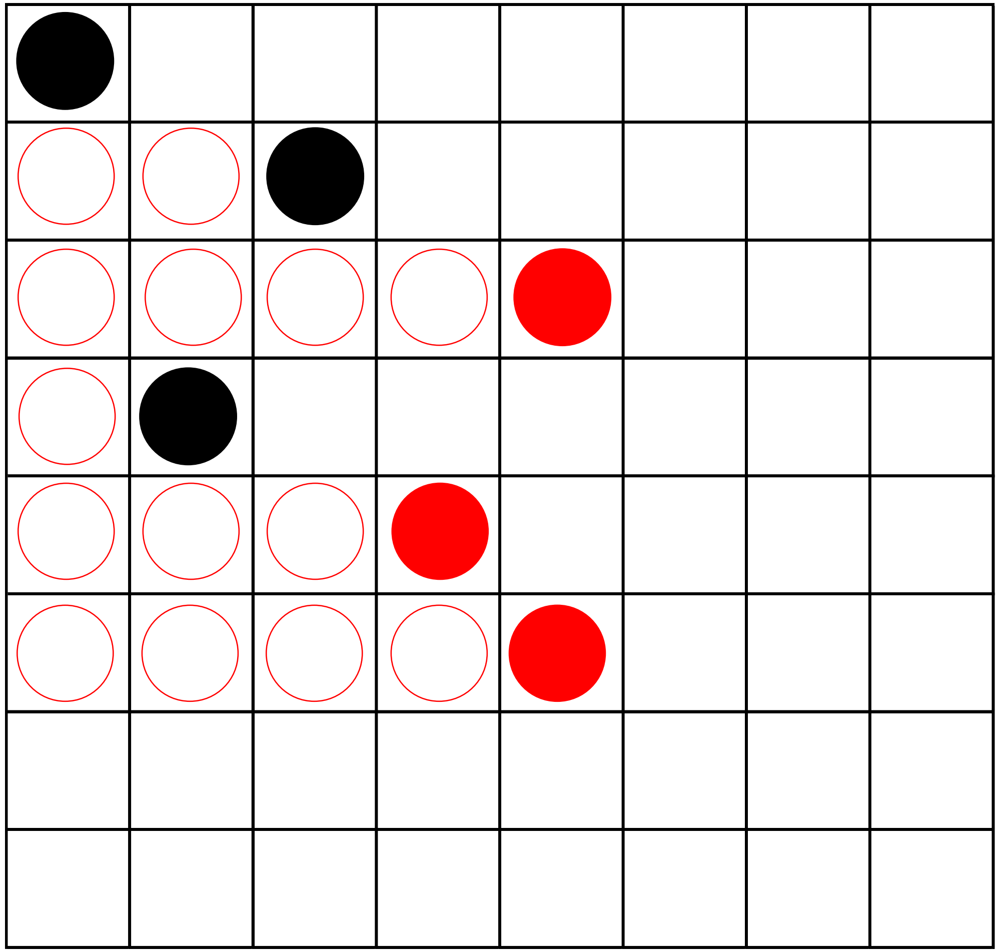
Tests 30 + 2 = 32 Backtracks 0

Backtracking



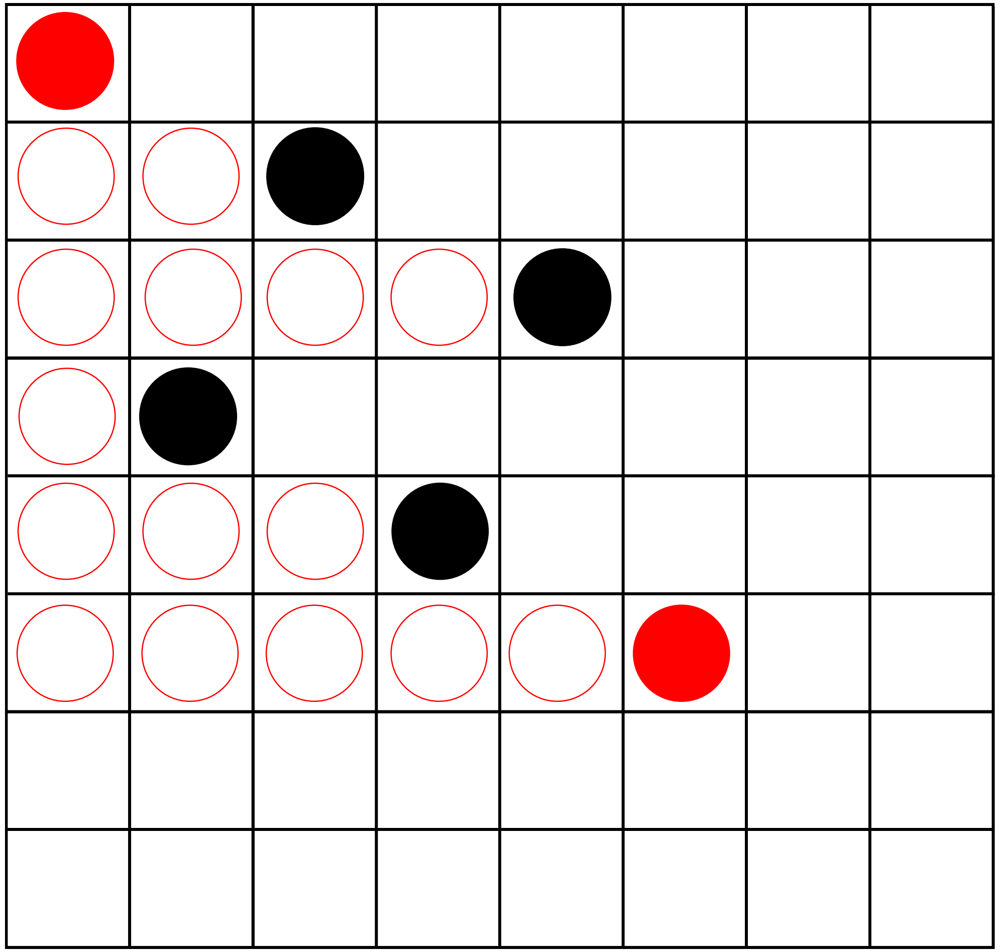
Tests 32 + 4 = 36 Backtracks 0

Backtracking



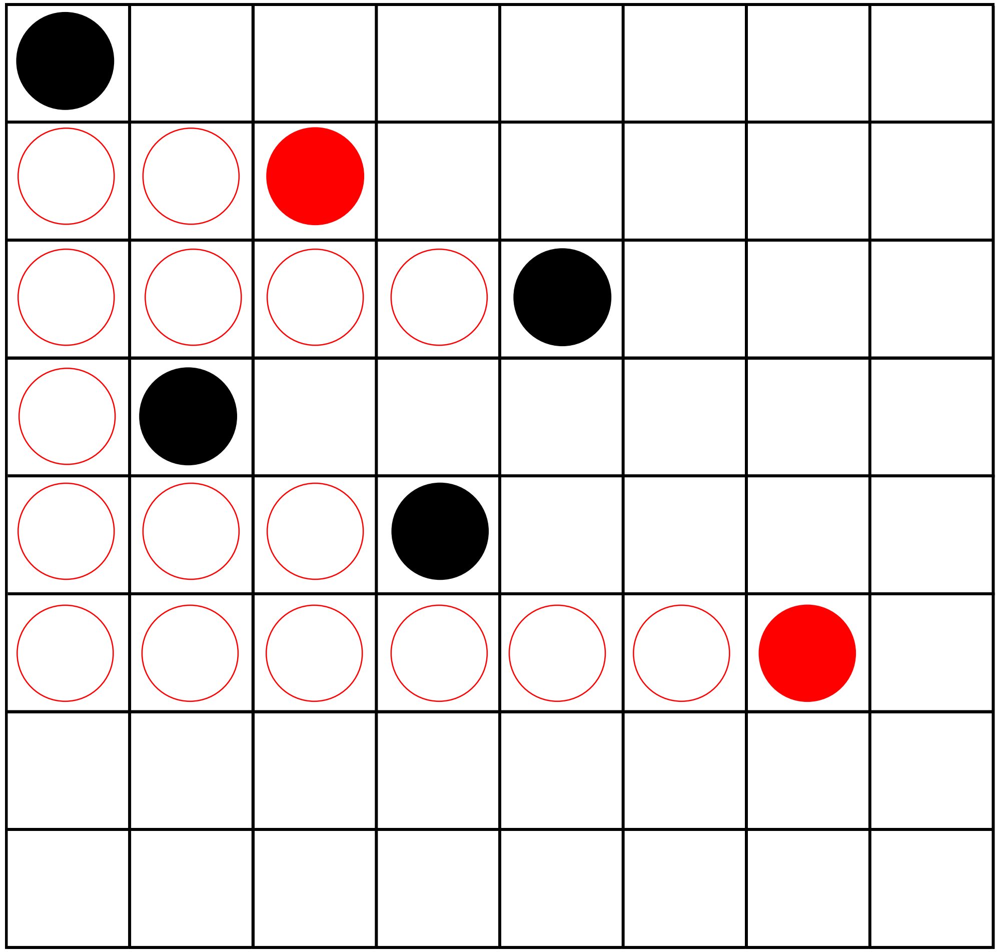
Tests 36 + 3 = 39 Backtracks 0

Backtracking



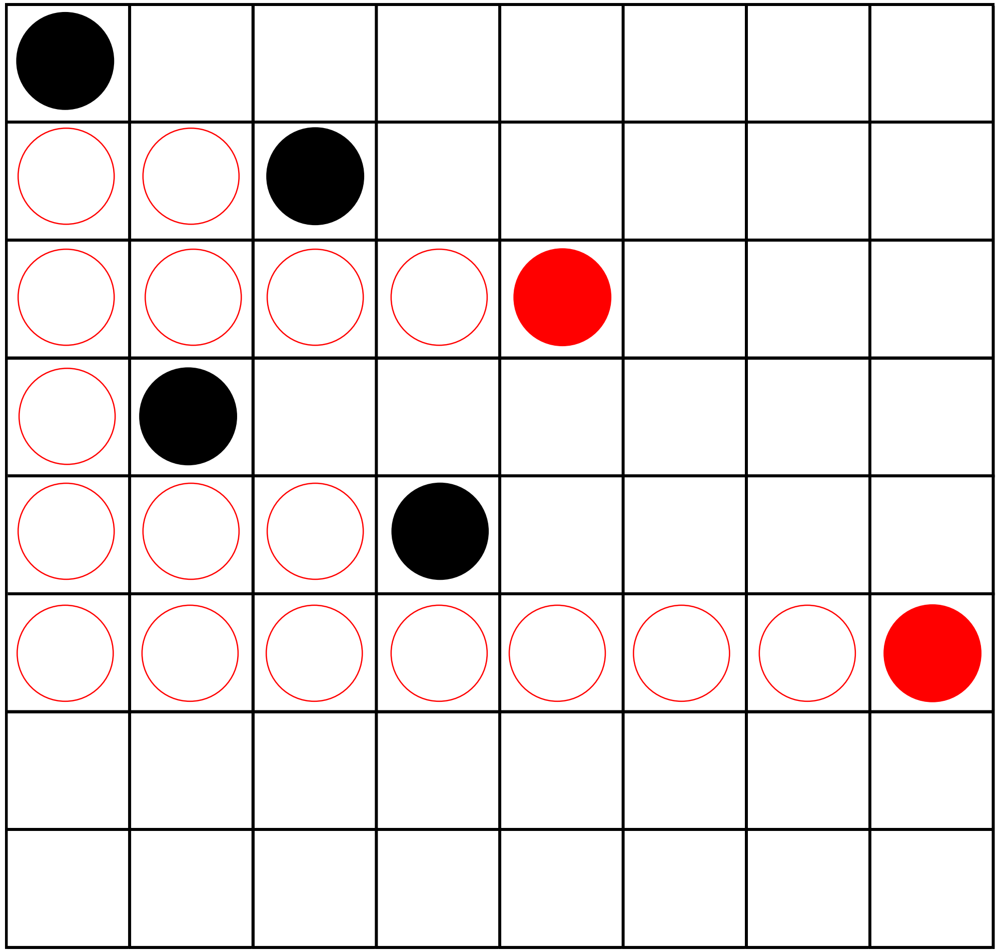
Tests 39 + 1 = 40 Backtracks 0

Backtracking



Tests 40 + 2 = 42 Backtracks 0

Backtracking



Tests 42 + 3 = 45 Backtracks 0

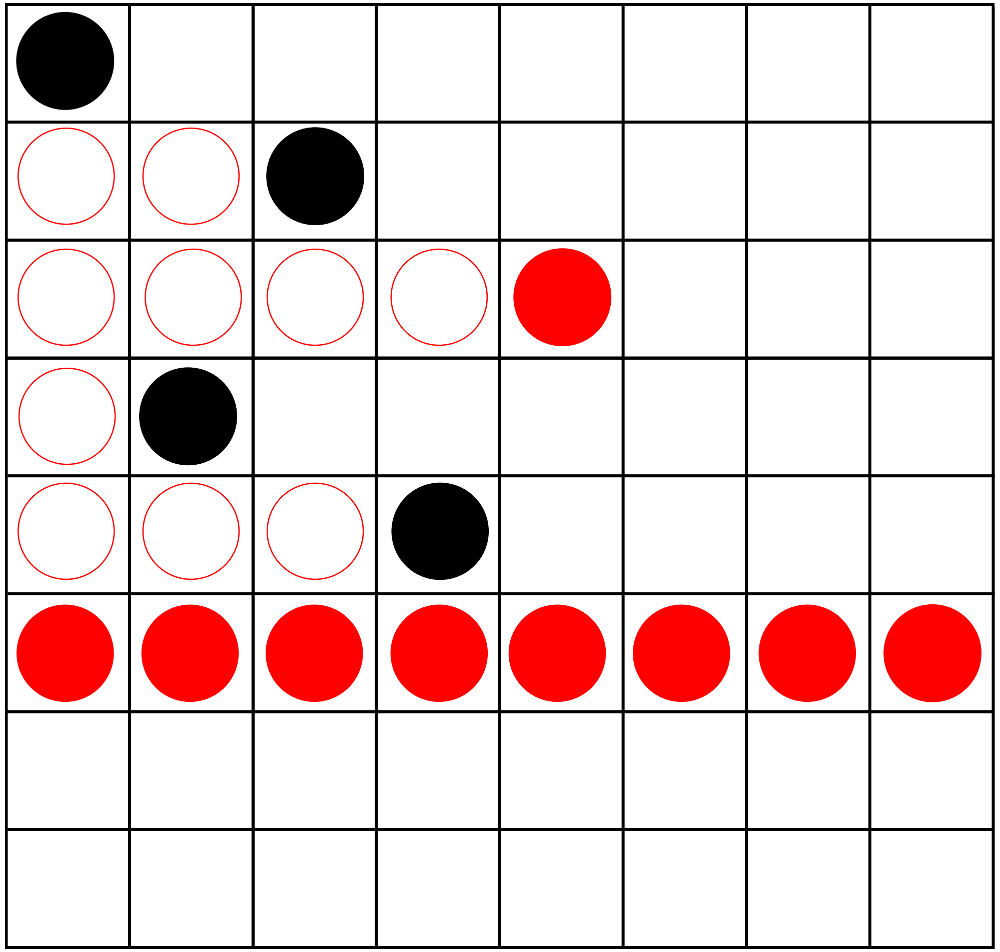
Backtracking

Fails

6

Backtracks

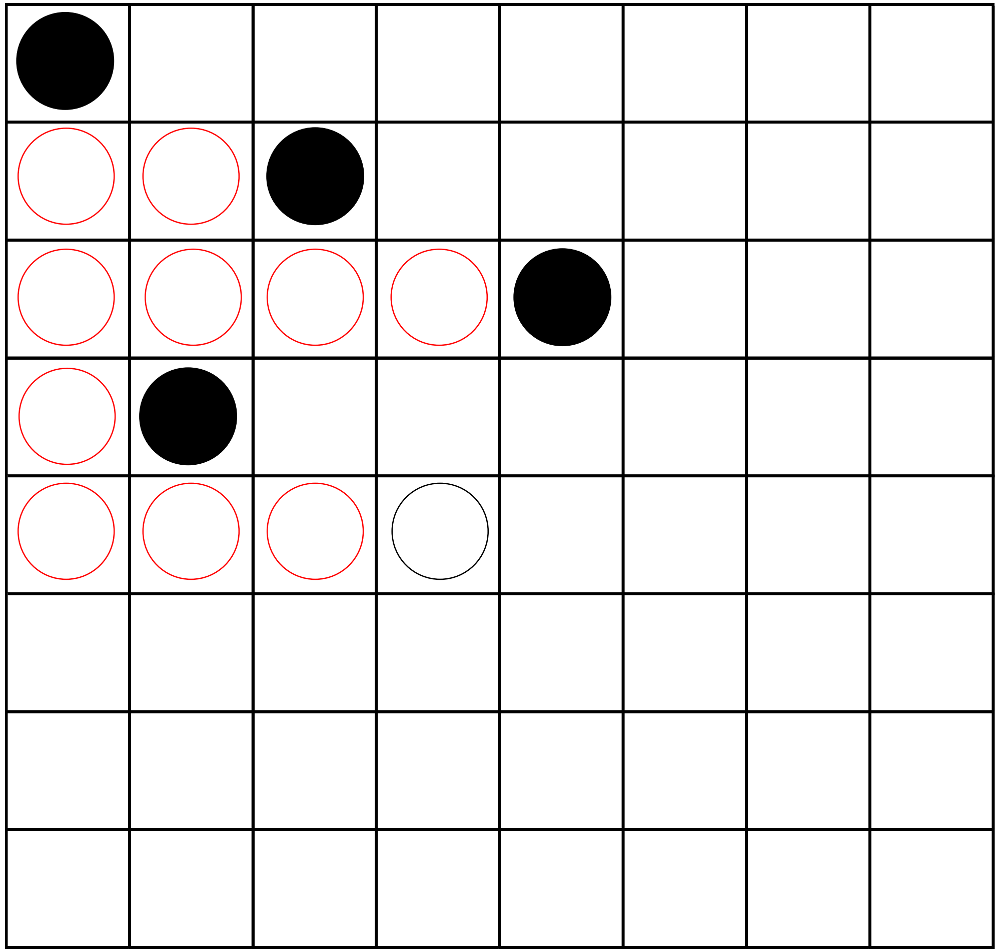
5



Tests 45

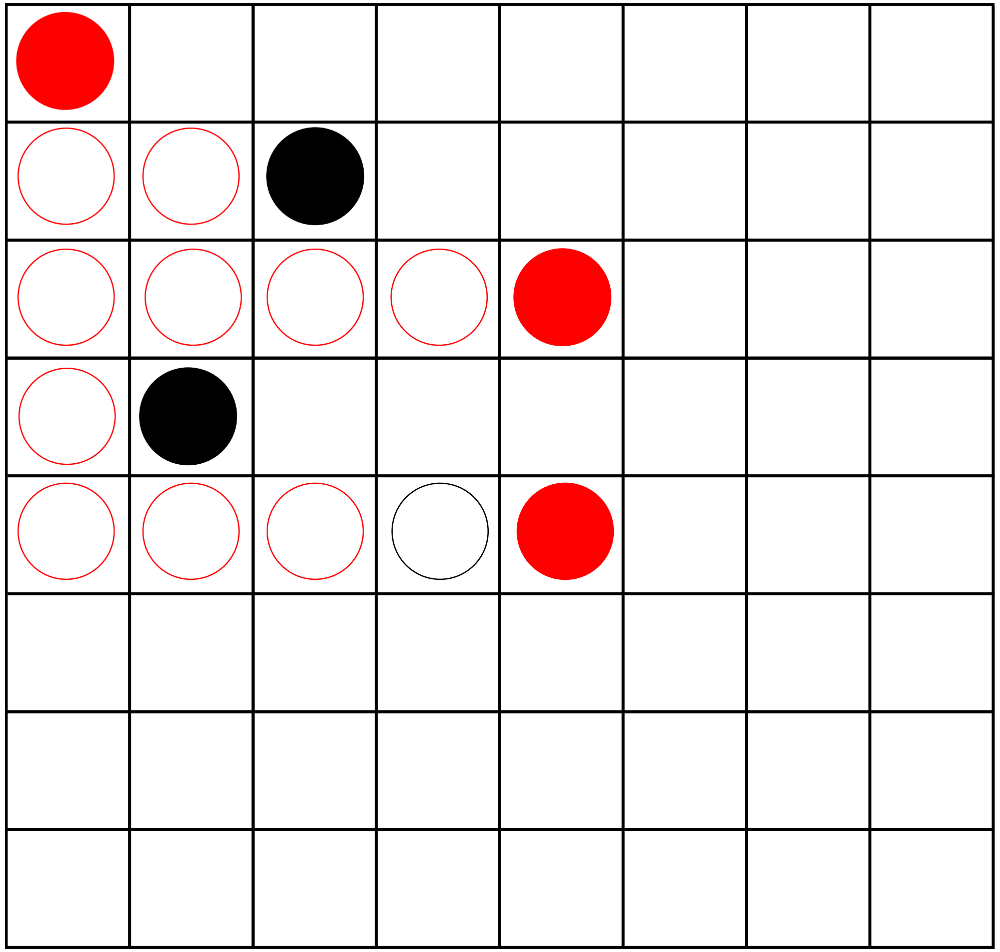
Backtracks 0

Backtracking



Tests 45 Backtracks 1

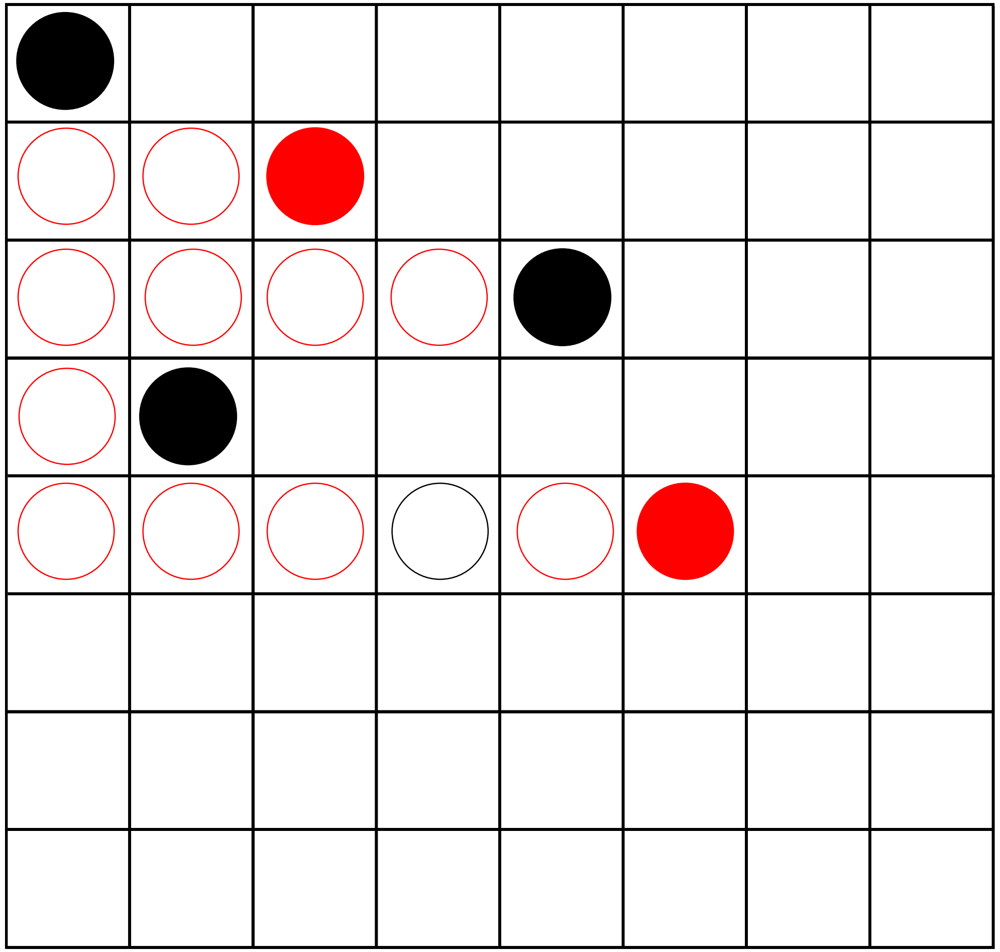
Backtracking



Tests 45 + 1 = 46

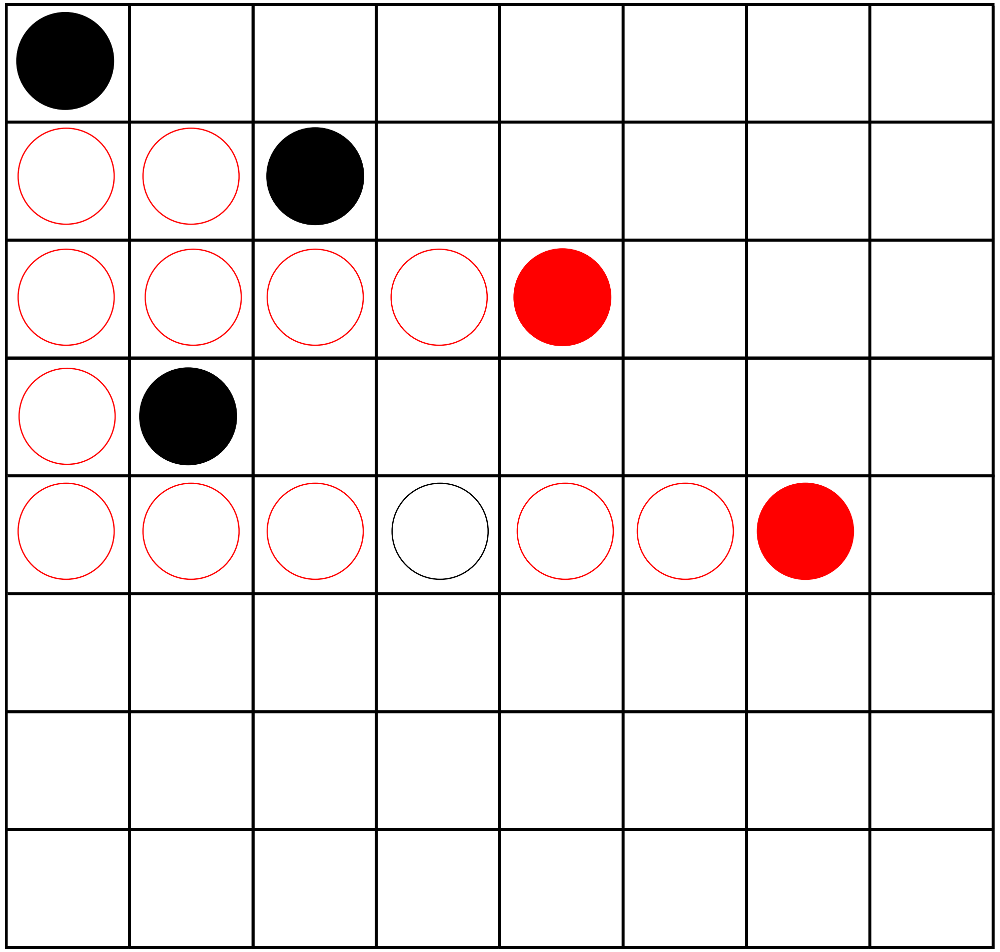
Backtracks 1

Backtracking



Tests 46 + 2 = 48 Backtracks 1

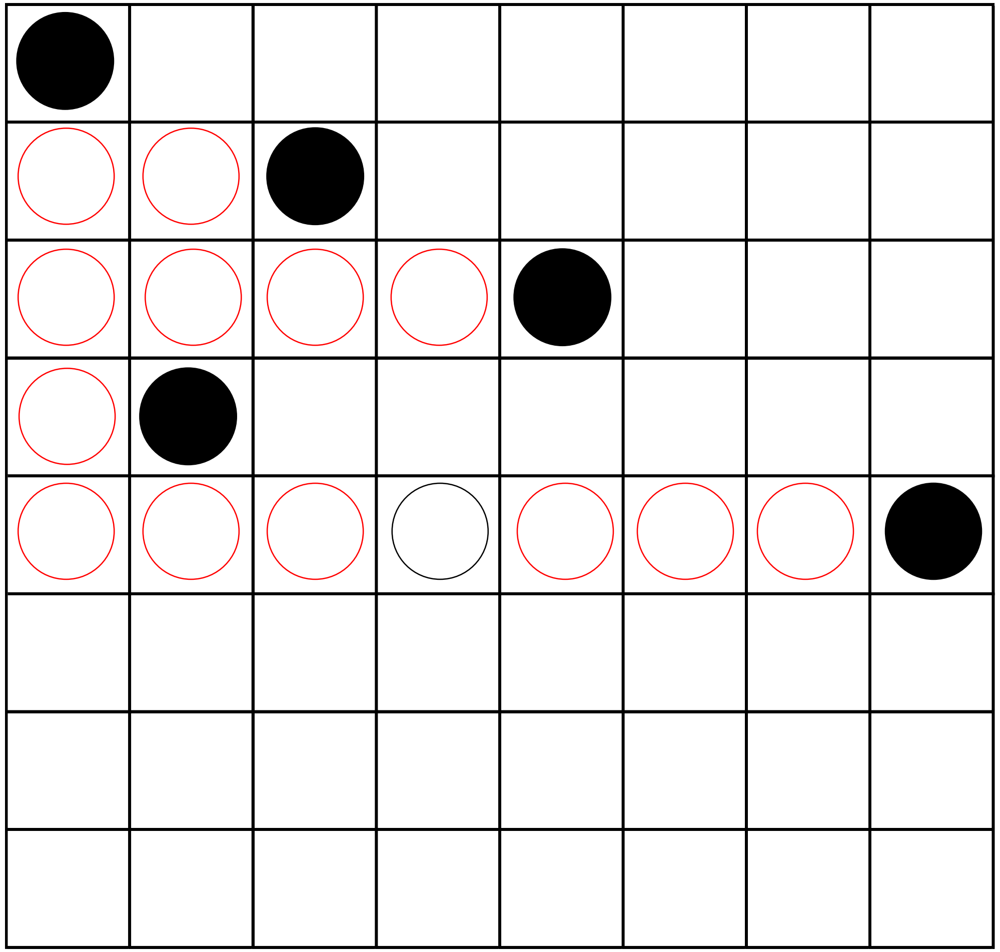
Backtracking



Tests 48 + 3 = 51

Backtracks 1

Backtracking



Tests 51 + 4 = 55 Backtracks 1

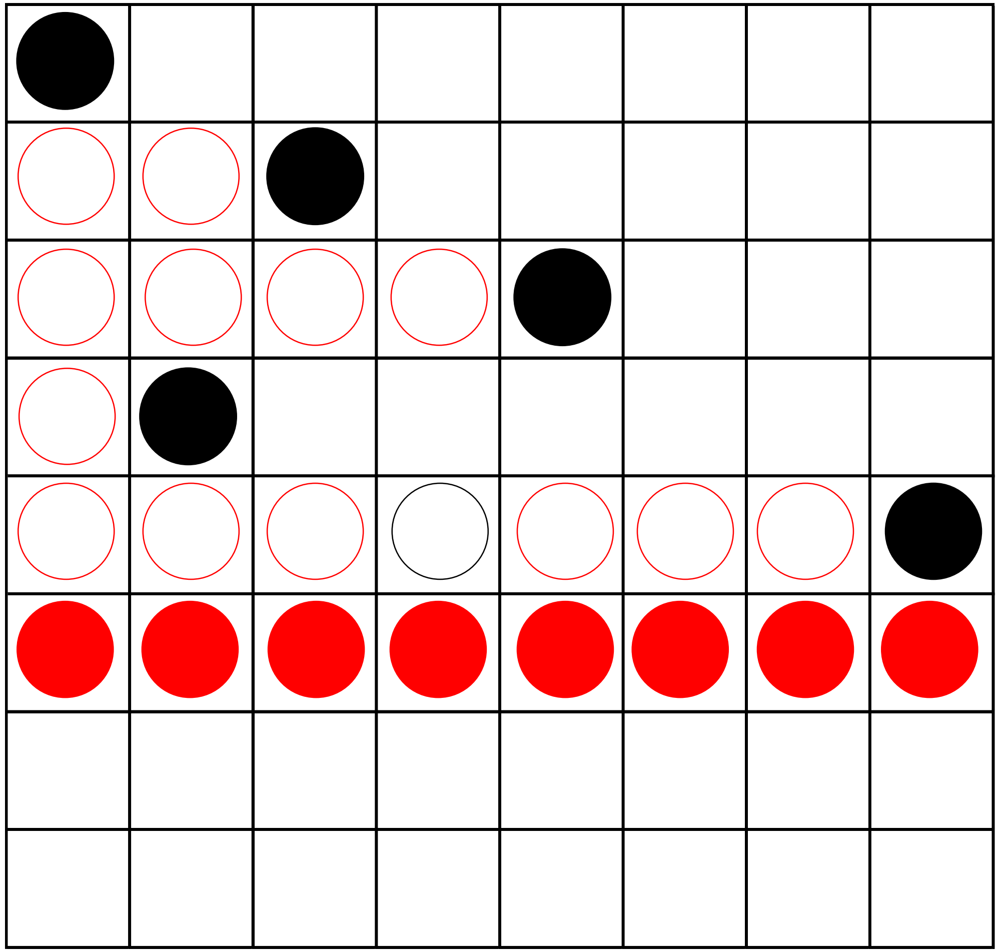
Backtracking

Fails

6

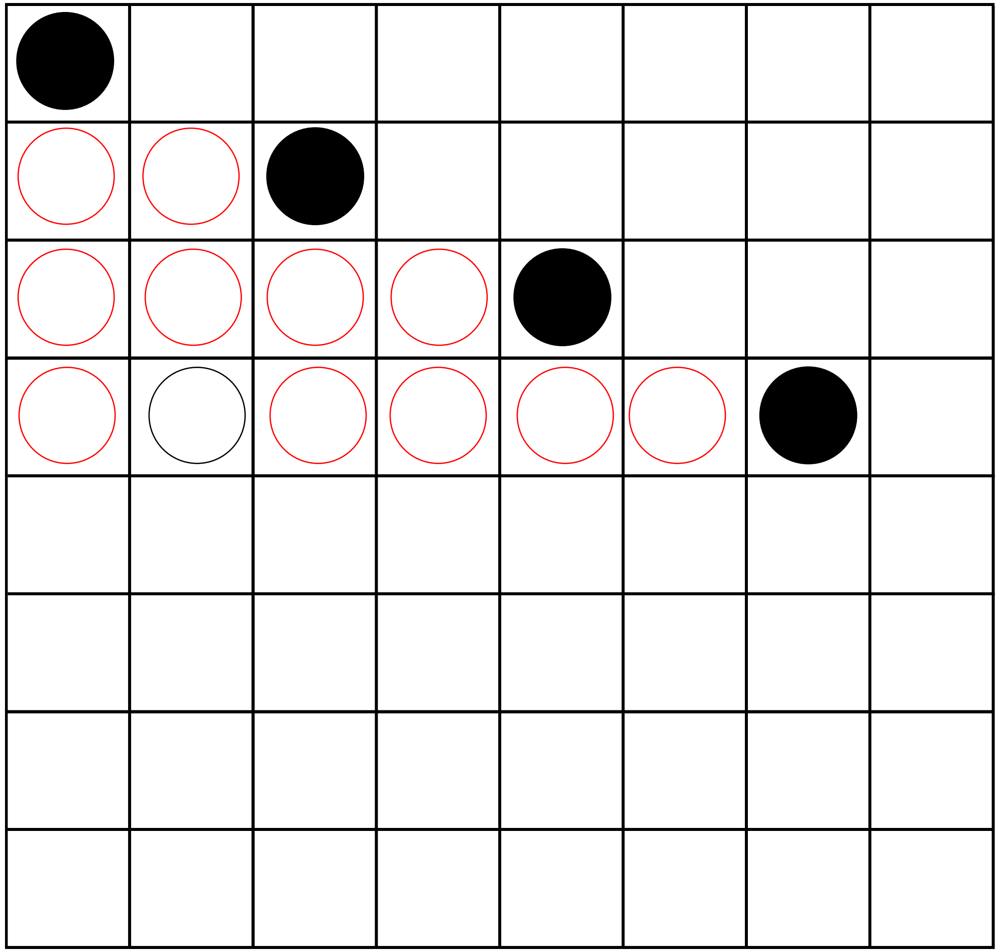
Backtracks

5 and 4



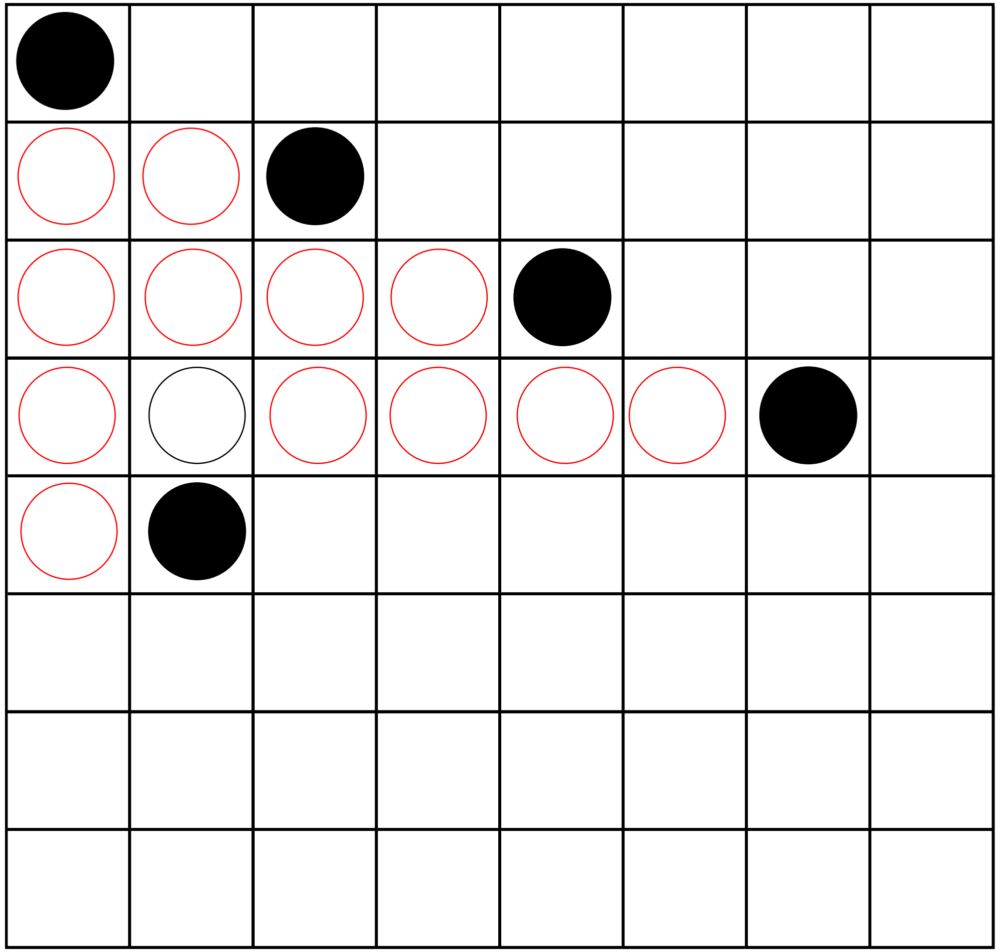
Tests 55+1+3+2+4+3+1+2+3 = 74 Backtracks 1

Backtracking



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

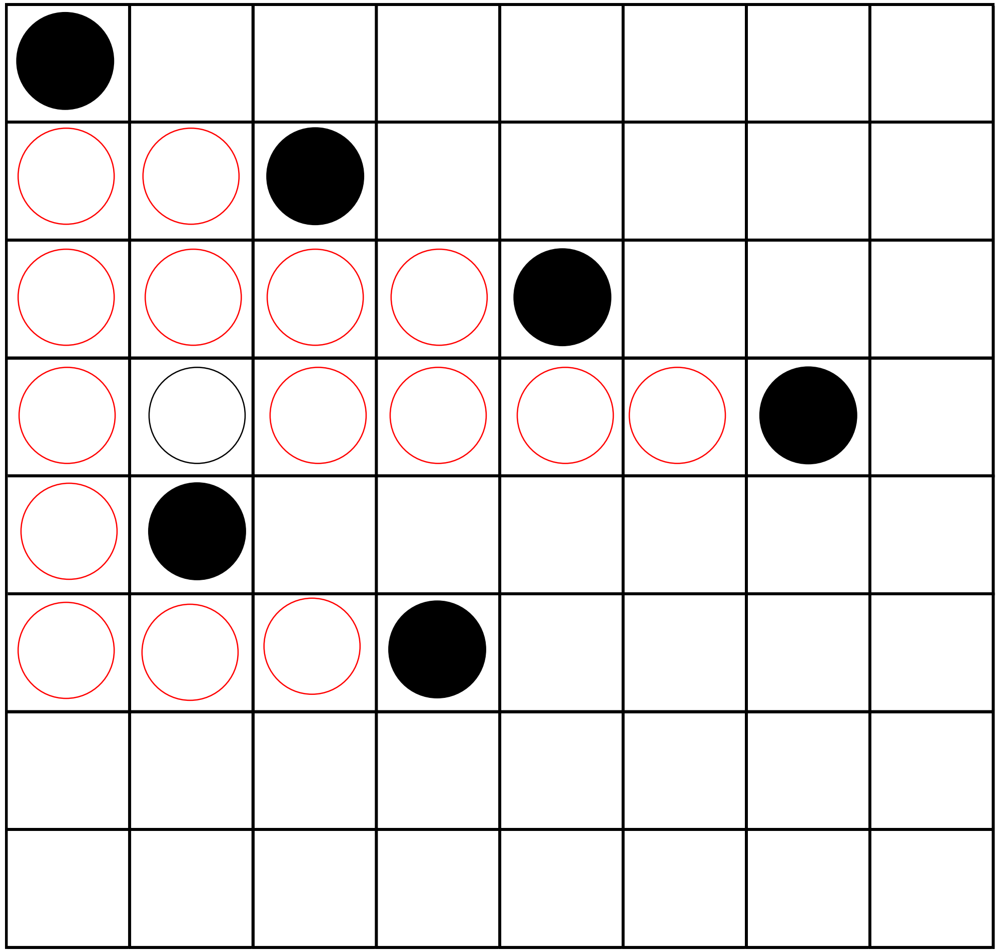
Backtracking



Tests 85 + 1 + 4 = 90

Backtracks 3

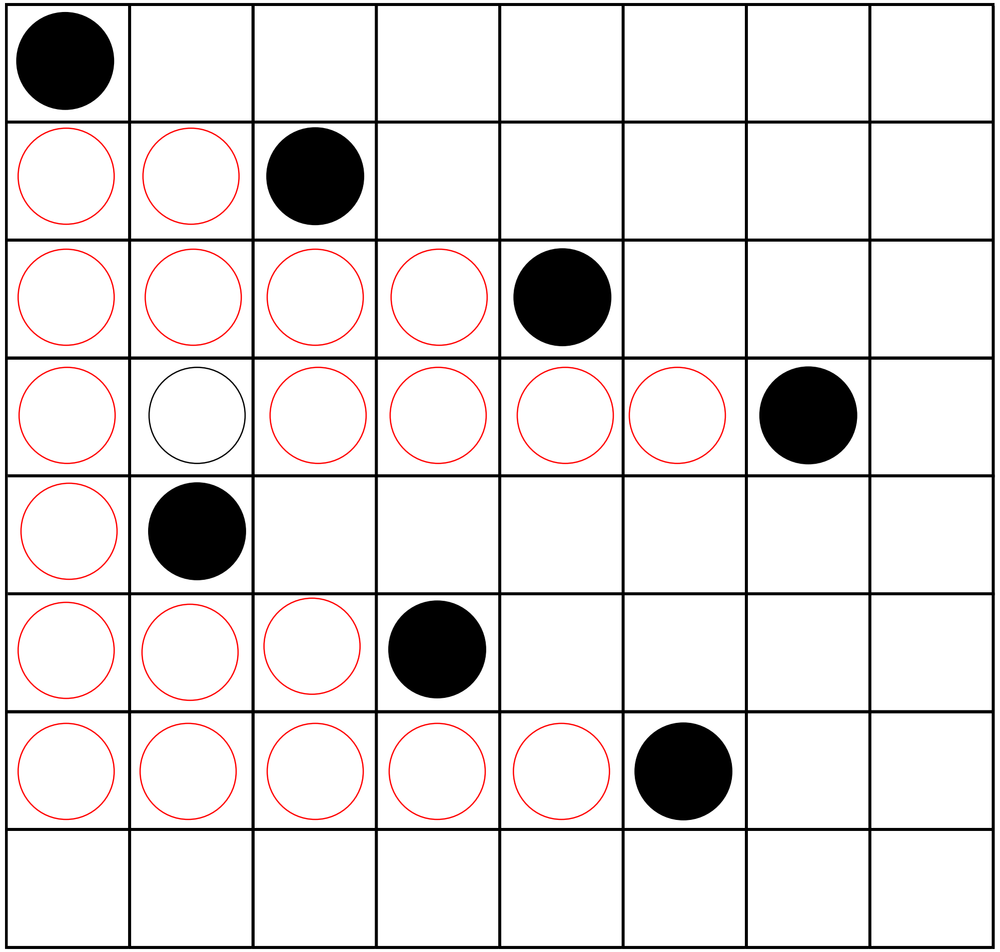
Backtracking



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

Backtracks 3

Backtracking



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

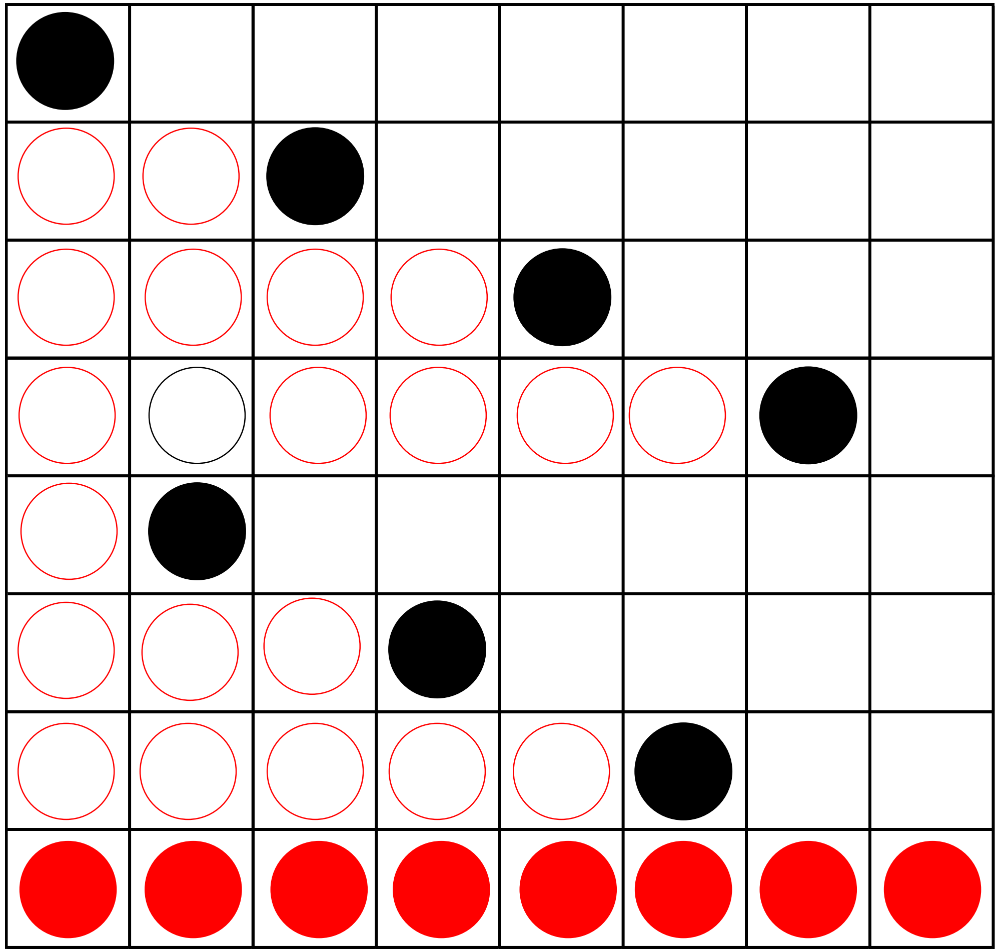
Backtracking

Fails

8

Backtracks

7



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

Backtracks 3+1=4

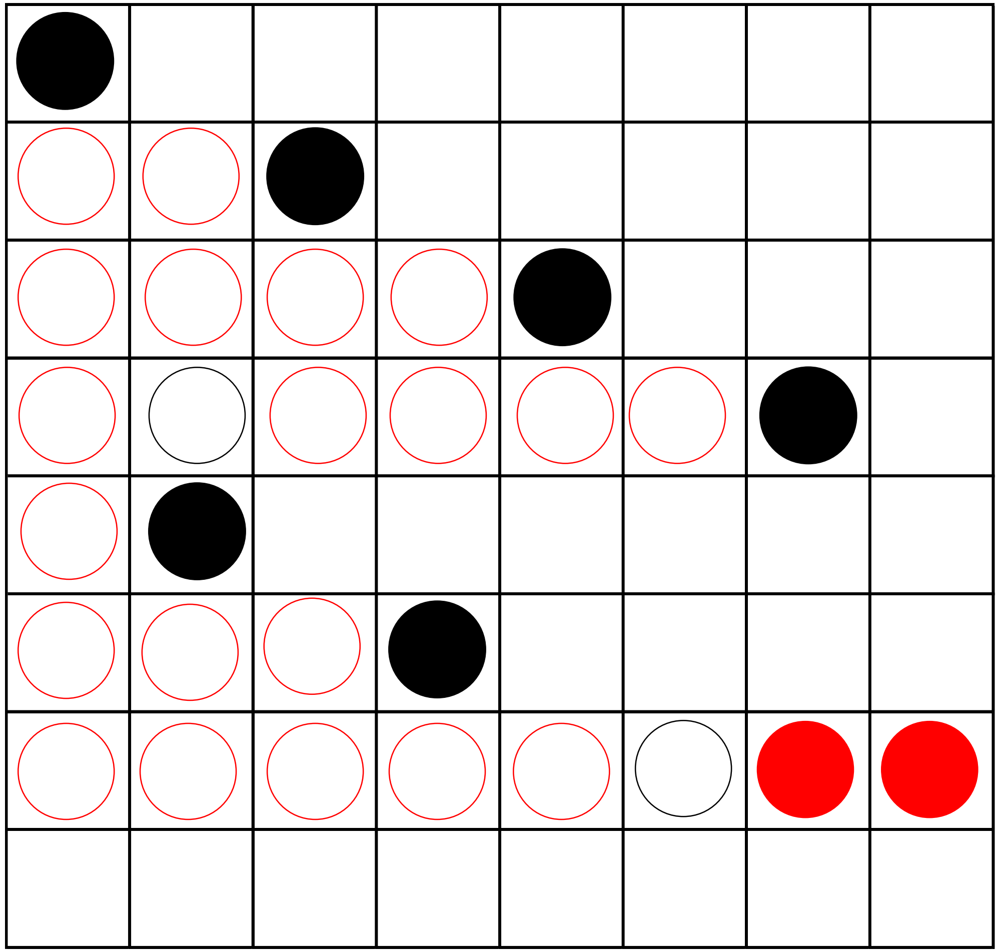
Backtracking

Fails

7

Backtracks

6



Tests 150+1+2= 153

Backtracks 4+1=5

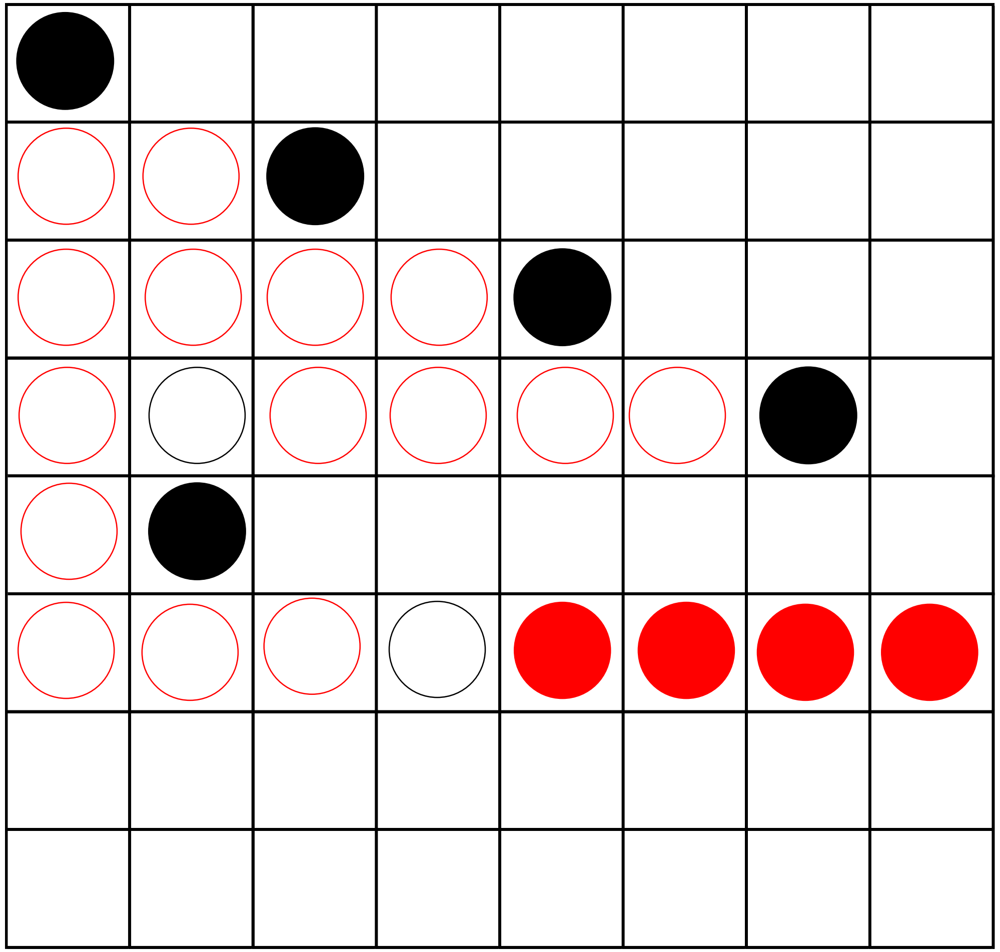
Backtracking

Fails

6

Backtracks

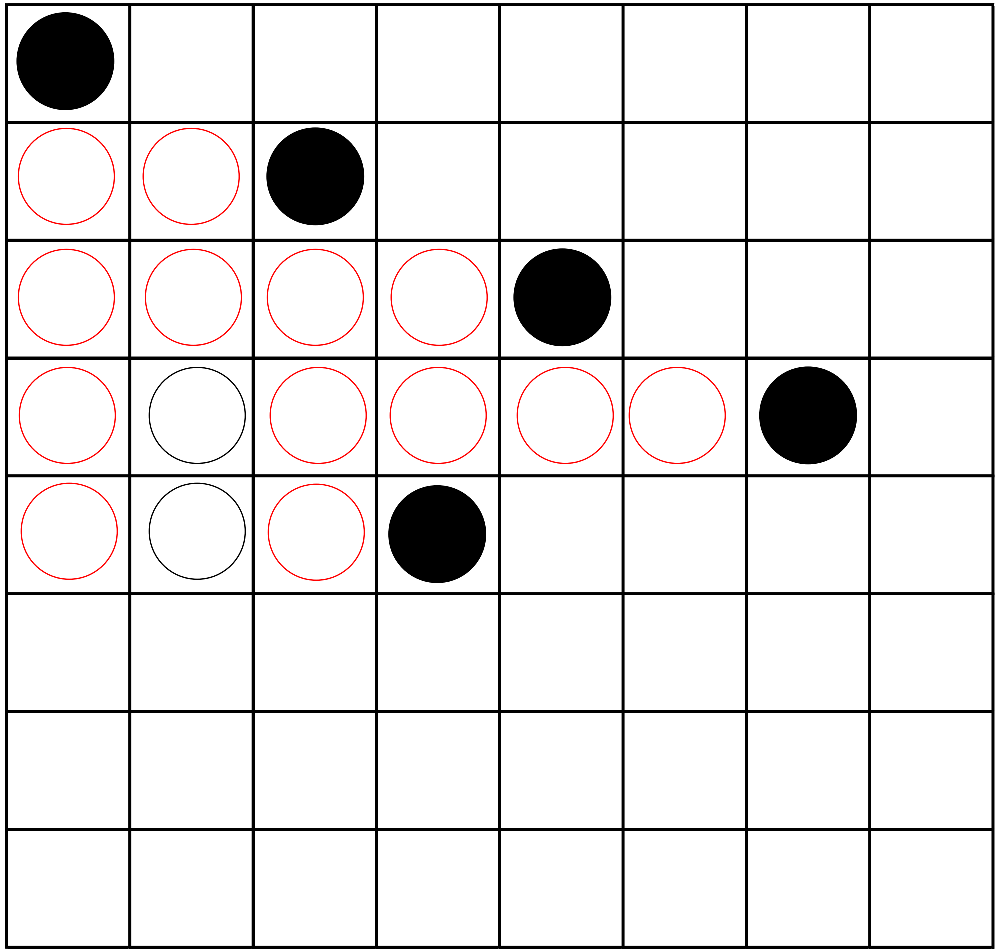
5



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

Backtracks 5+1=6

Backtracking



Tests 162+2+4= 168

Backtracks 6+1=7

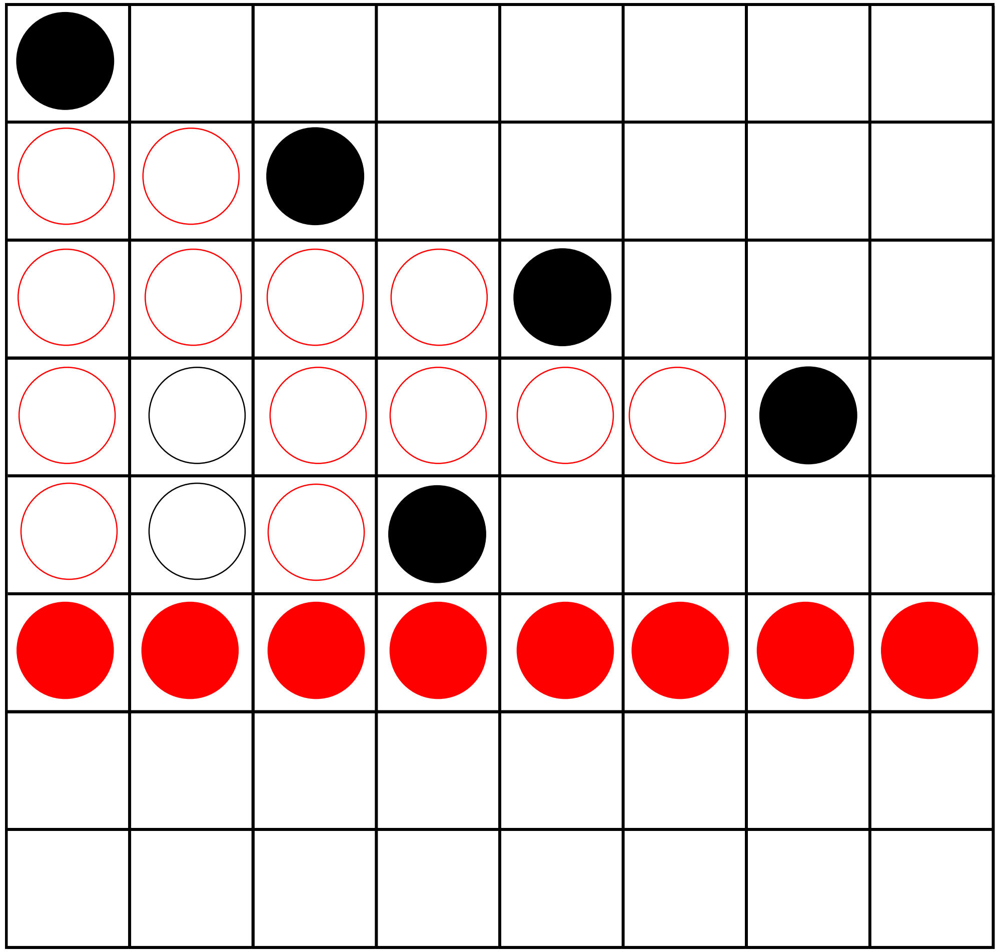
Backtracking

Fails

6

Backtracks

5



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

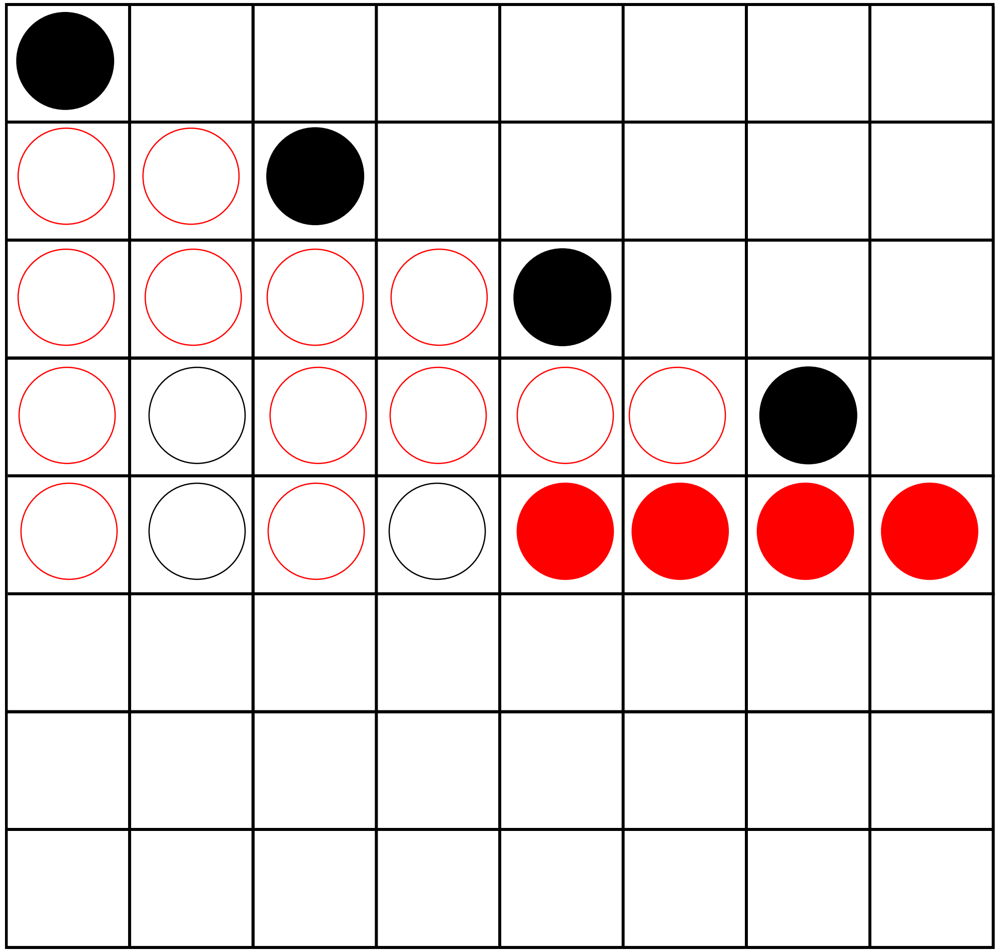
Backtracking

Fails

5

Backtracks

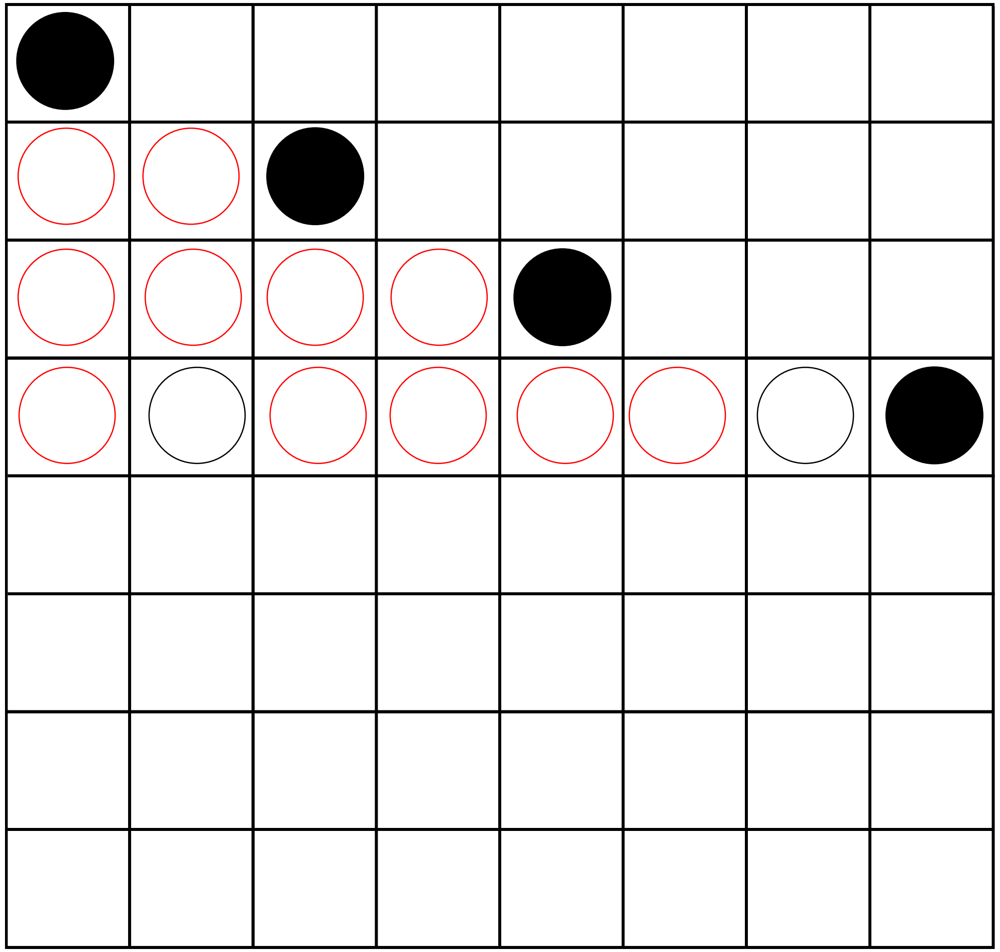
4



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

Backtracks 7+1=8

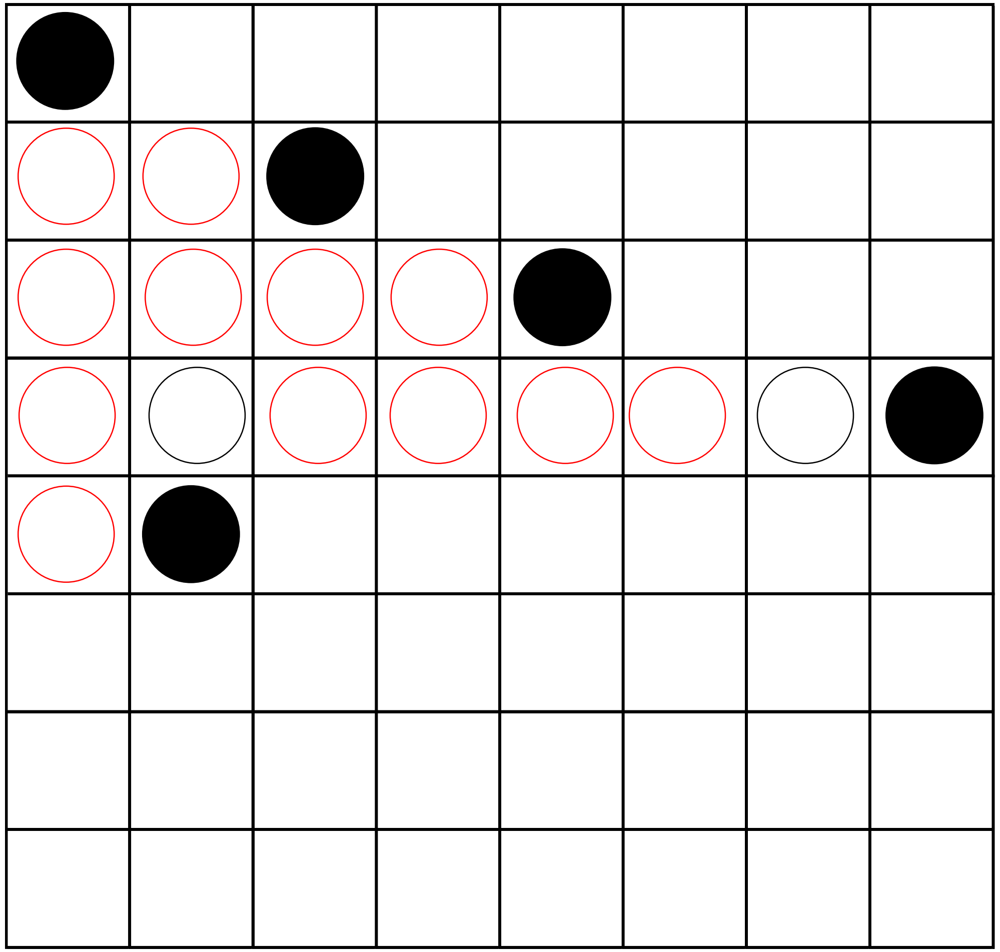
Backtracking



Tests 198 + 3 = 201

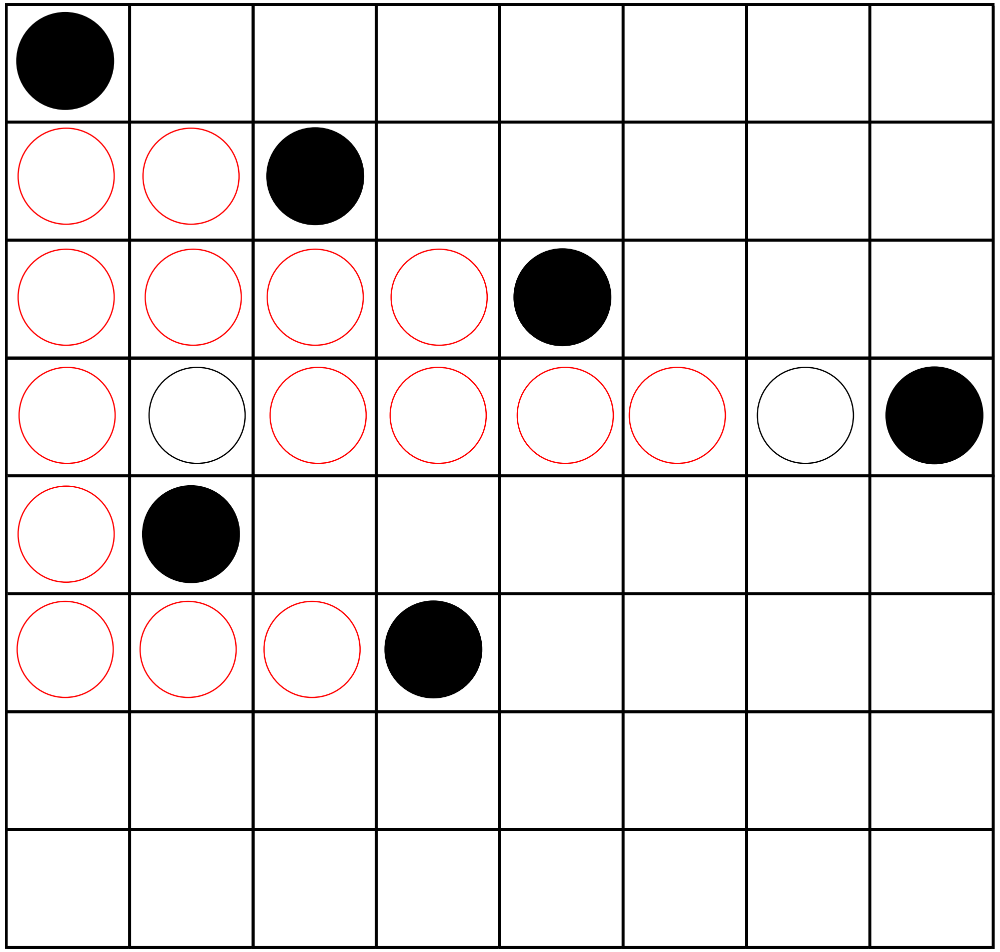
Backtracks 8+1=9

Backtracking



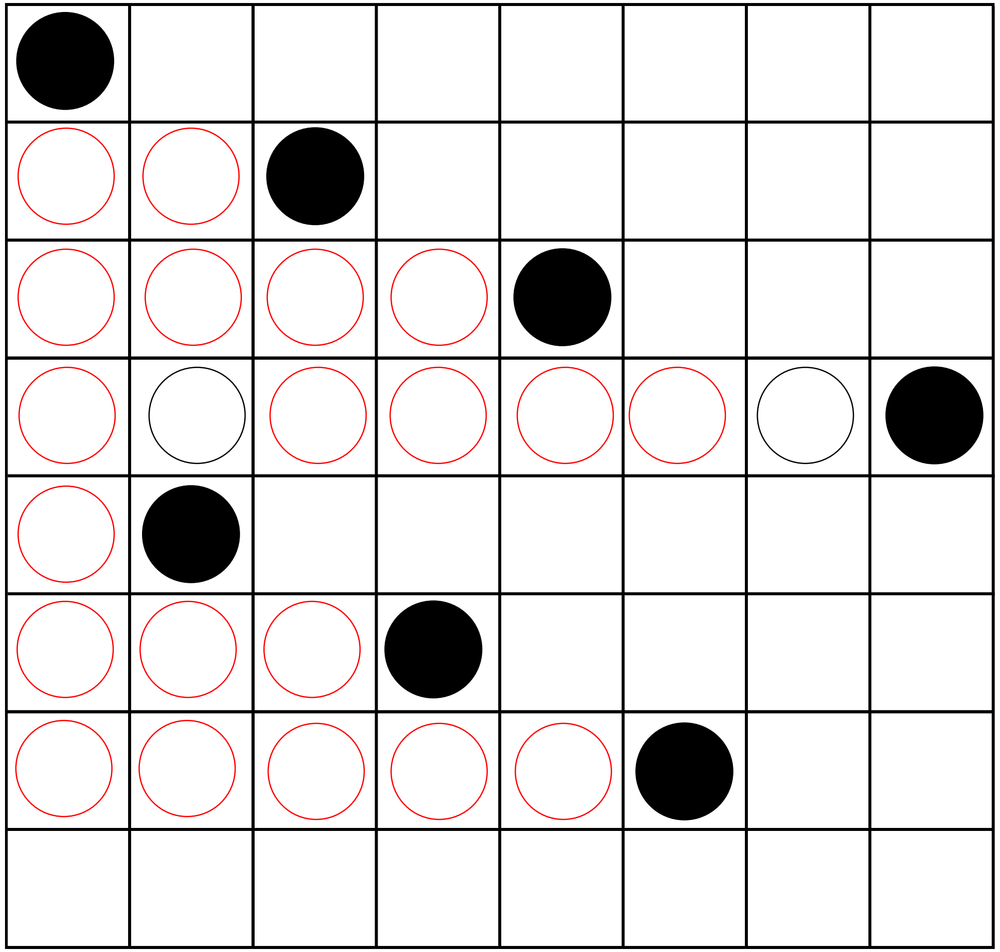
Tests 201+1+4 = 206 Backtracks 9

Backtracking



Tests 206+1+3+2+5 = 217 Backtracks 9

Backtracking



Tests 217+1+5+2+5+3+6 = 239 Backtracks 9

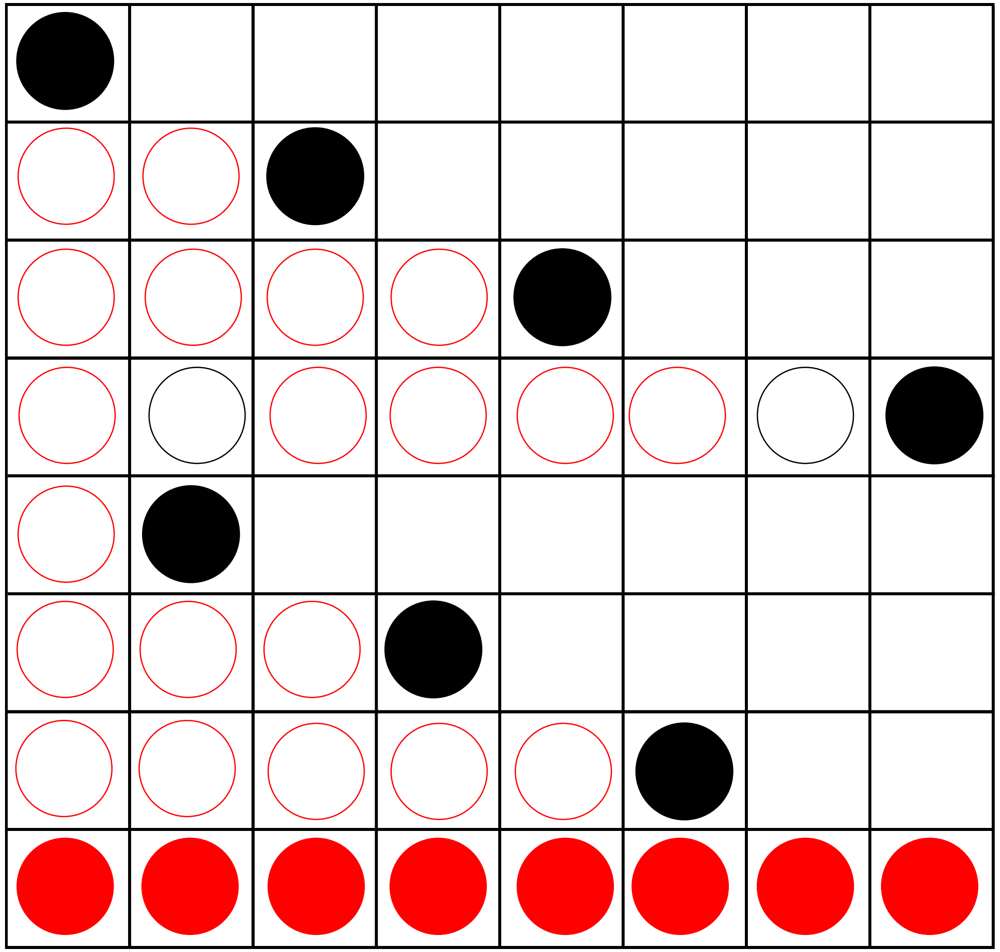
Backtracking

Fails

8

Backtracks

7



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

Backtracks 9+1=10

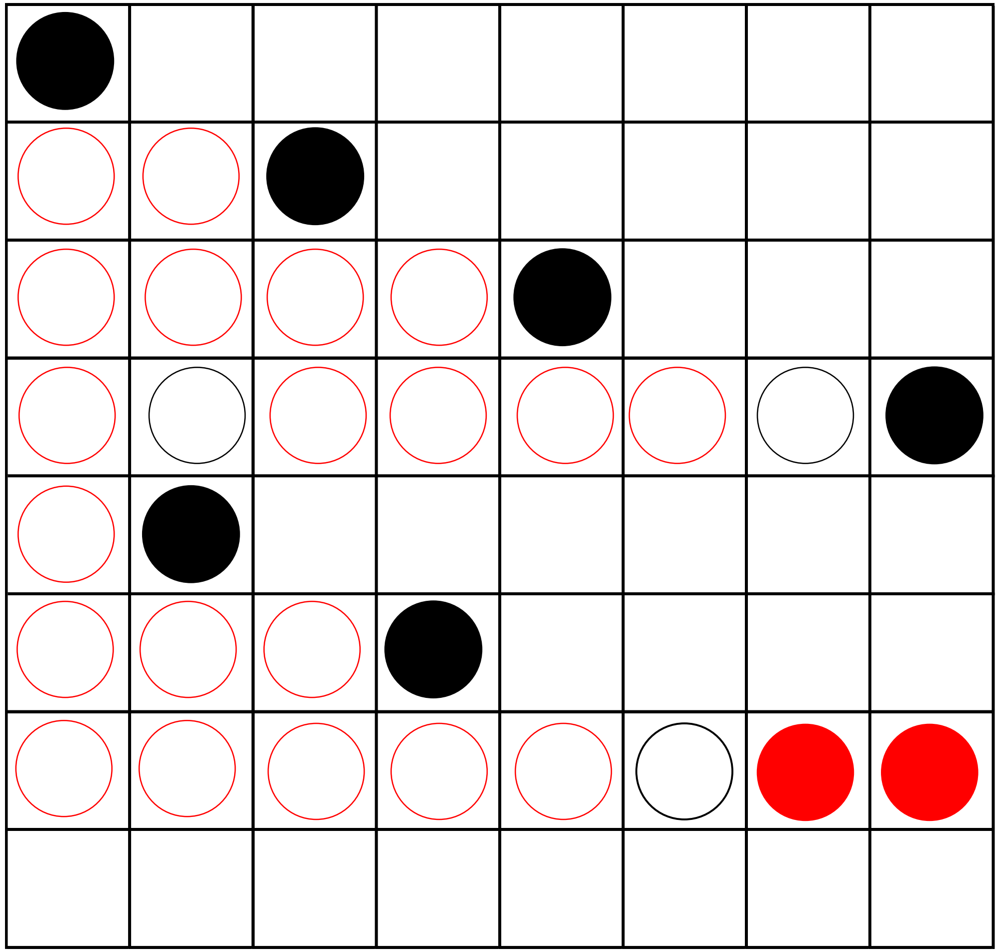
Backtracking

Fails

7

Backtracks

6



Tests 274+1+2= 277

Backtracks 10+1=11

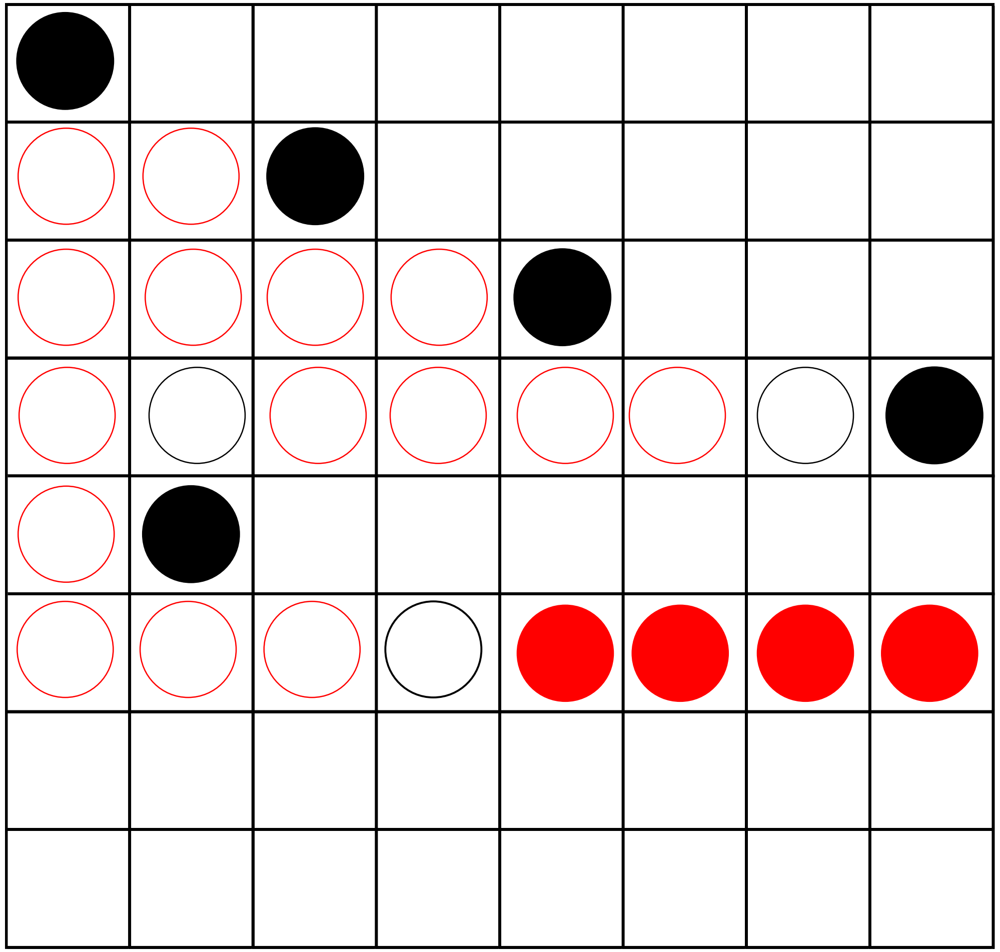
Backtracking

Fails

6

Backtracks

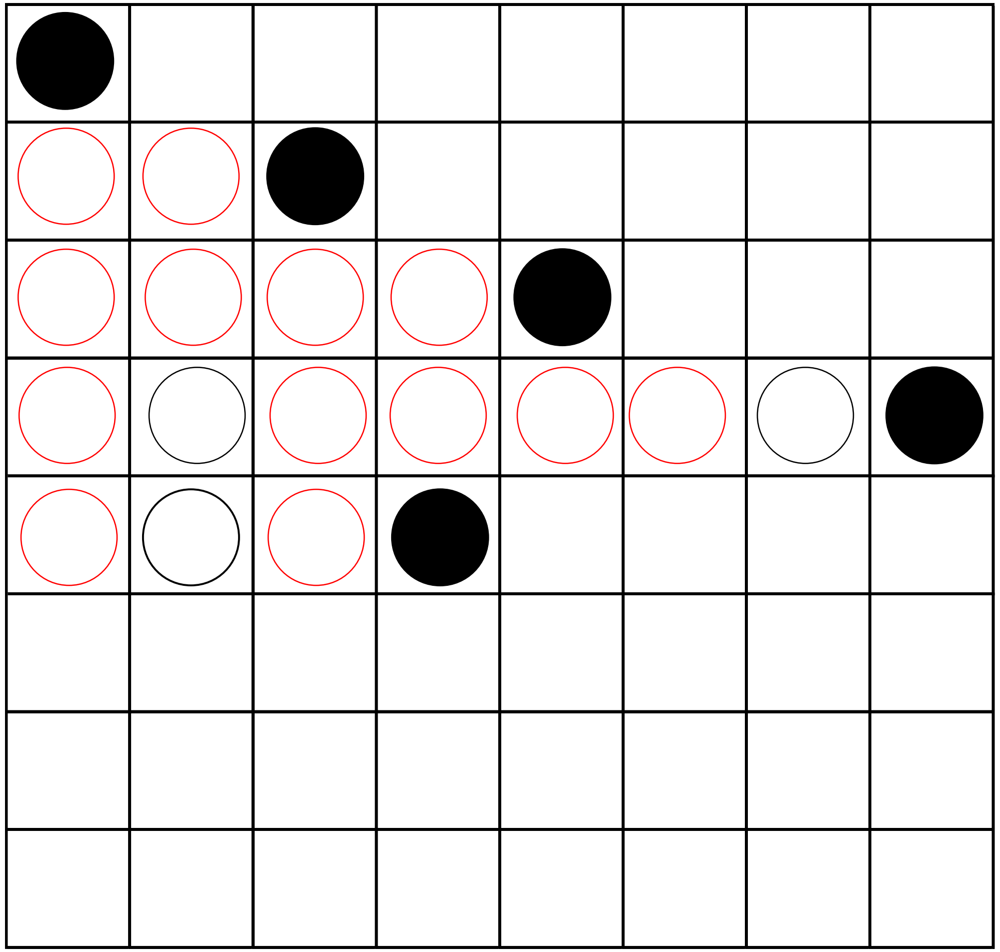
5



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

Backtracks 11+1=12

Backtracking



Tests 286+2+4= 292

Backtracks 12

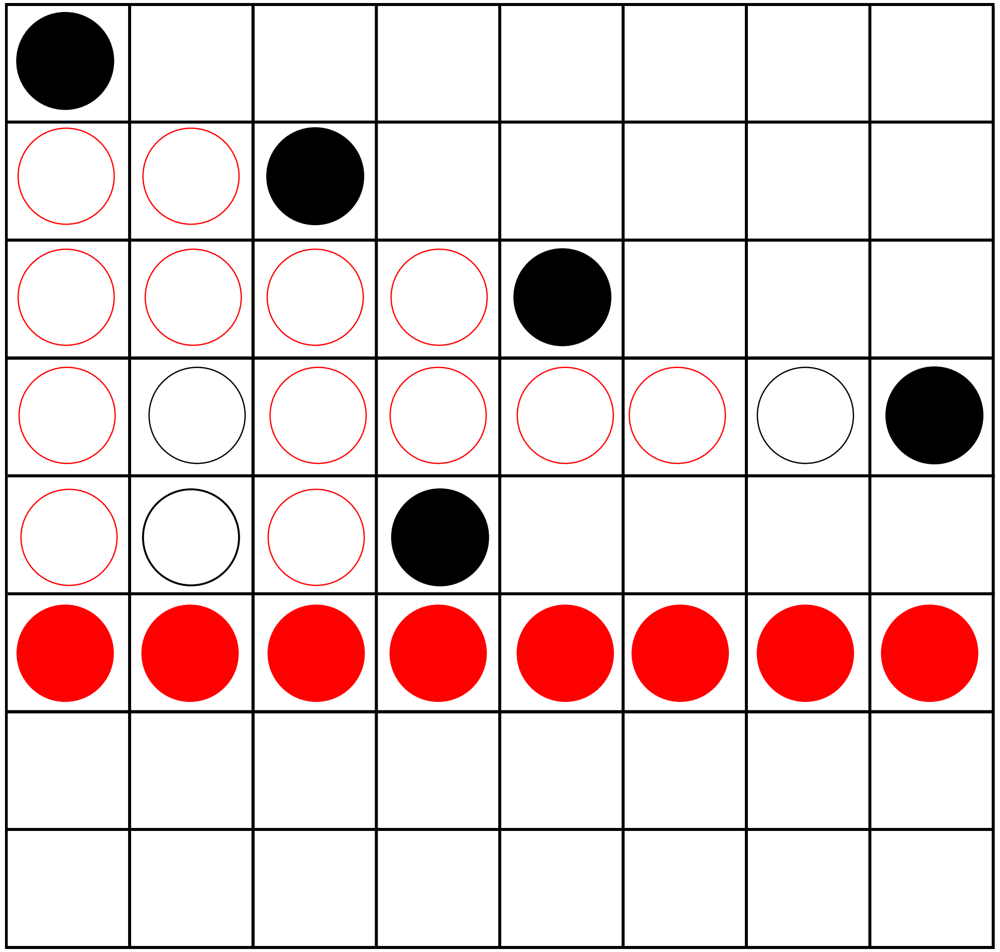
Backtracking

Fails

6

Backtracks

5



Tests 292+1+3+2+5+3+1+2+3= 312

Backtracks 12+1=13

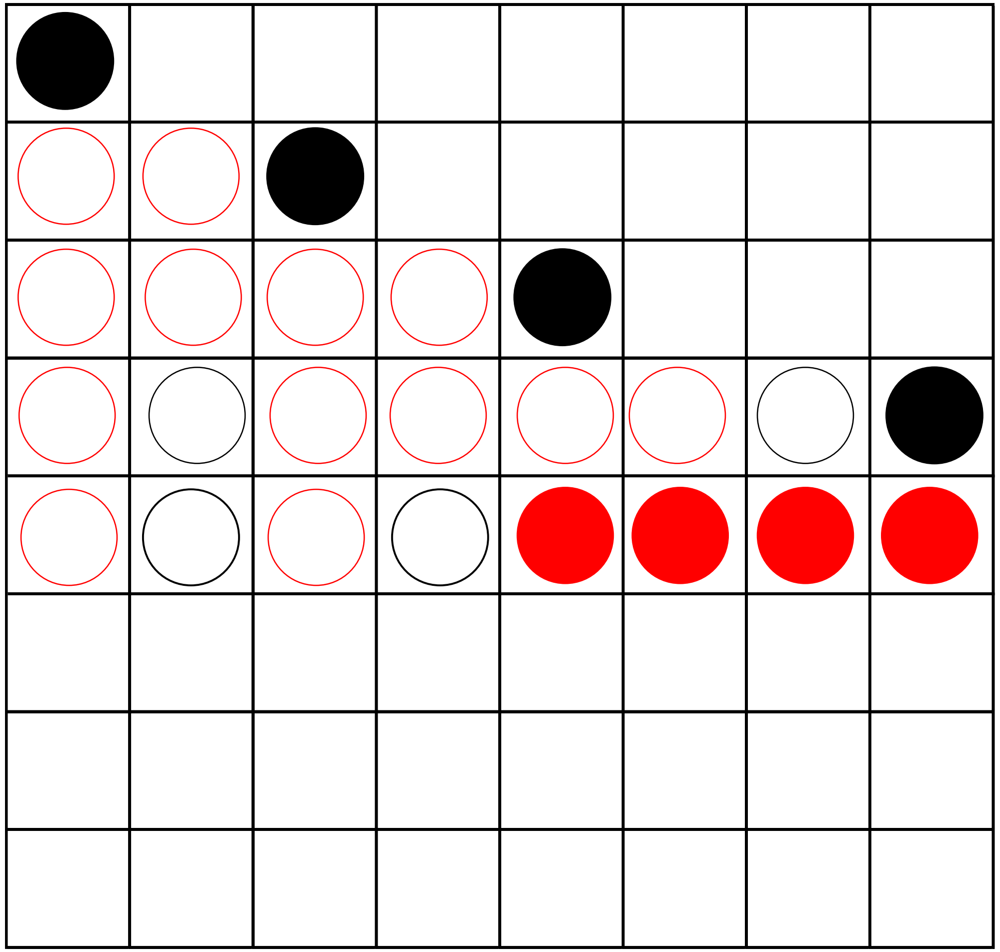
Backtracking

Fails

5

Backtracks

4 and 3



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

Backtracks 13+2=15

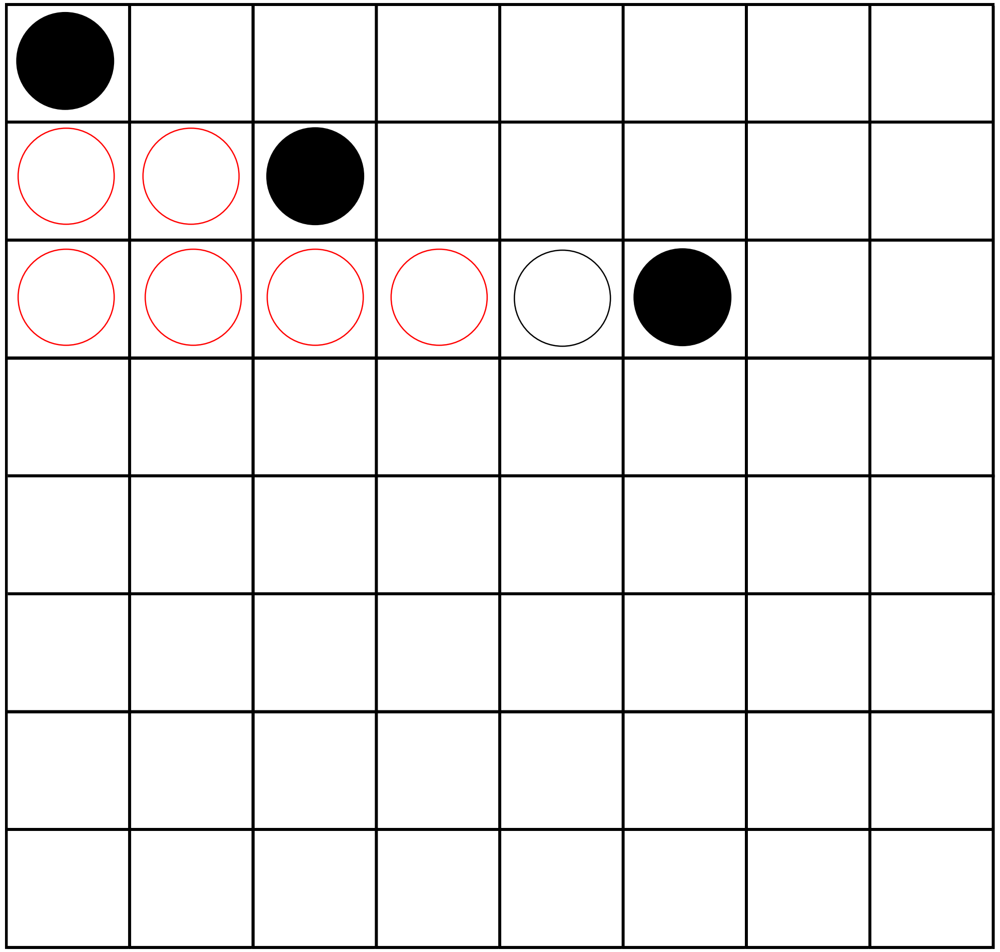
Backtracking

$X_1=1$

$X_2=3$

$X_3=5$

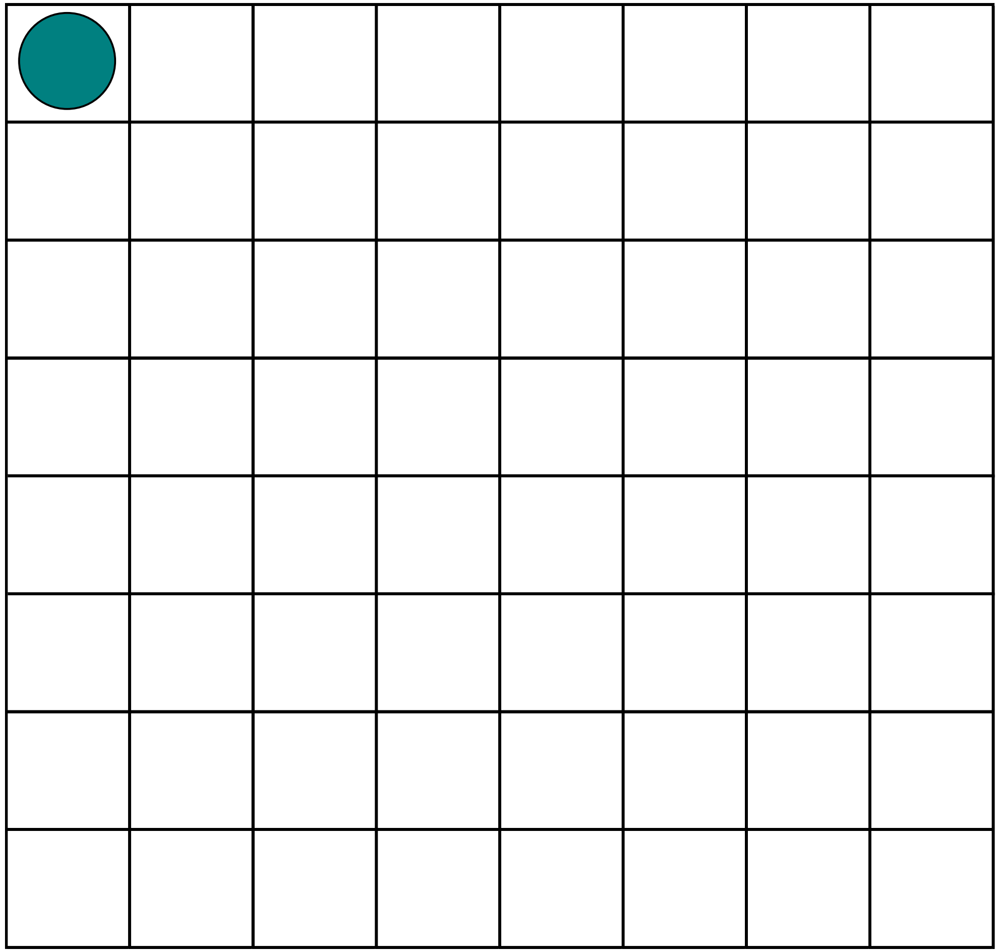
Impossible



Tests 322 + 2 = 324

Backtracks 15

Forward Checking



Tests 0

Backtracks 0

Forward Checking

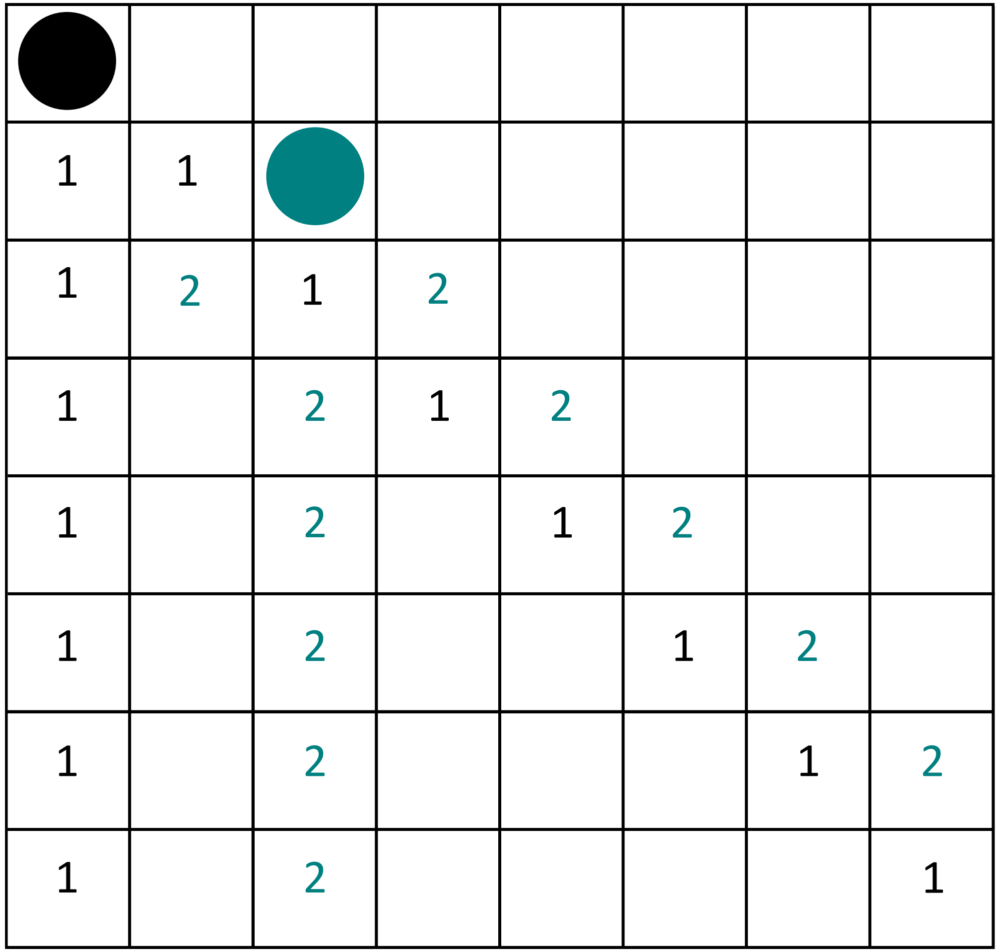
`Q1 #\= Q2,  L1+Q1 #\= L2+Q2,  L1+Q2 #\= L2+Q1.`

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

Tests 8 * 7 = 56

Backtracks 0

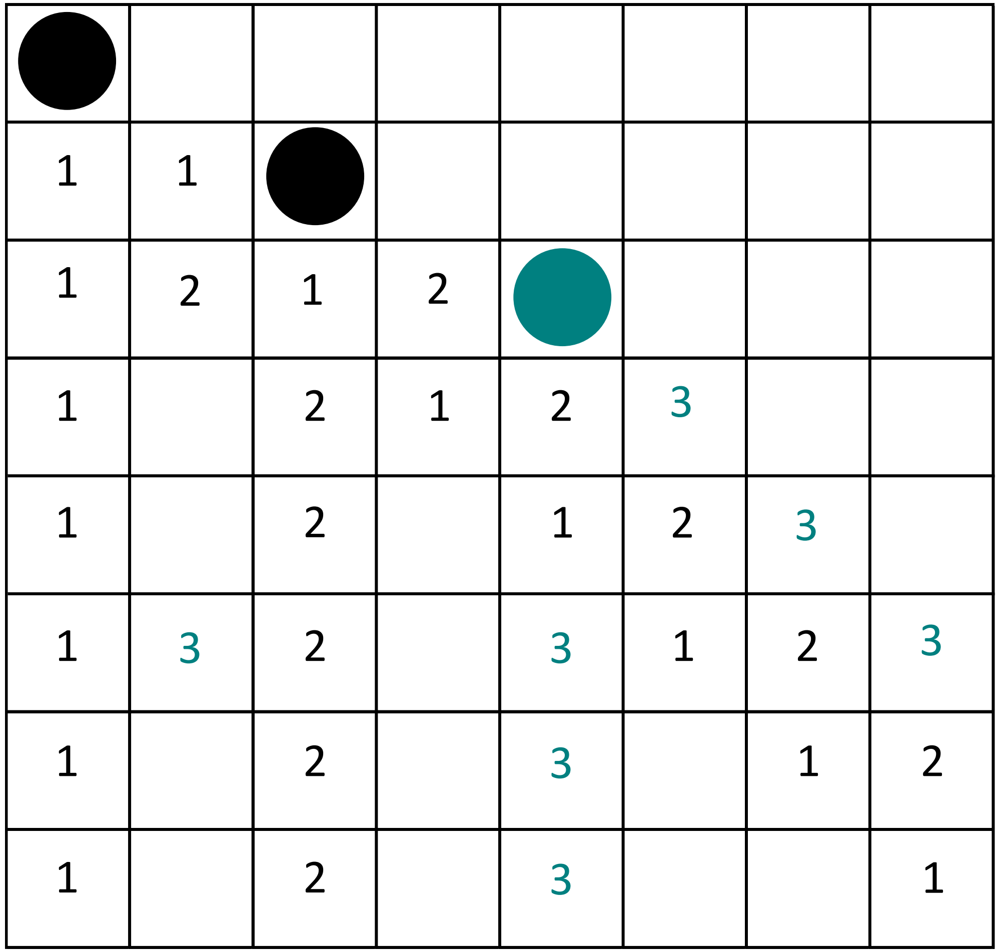
Forward Checking



Tests 56 + 6 * 6 = 92

Backtracks 0

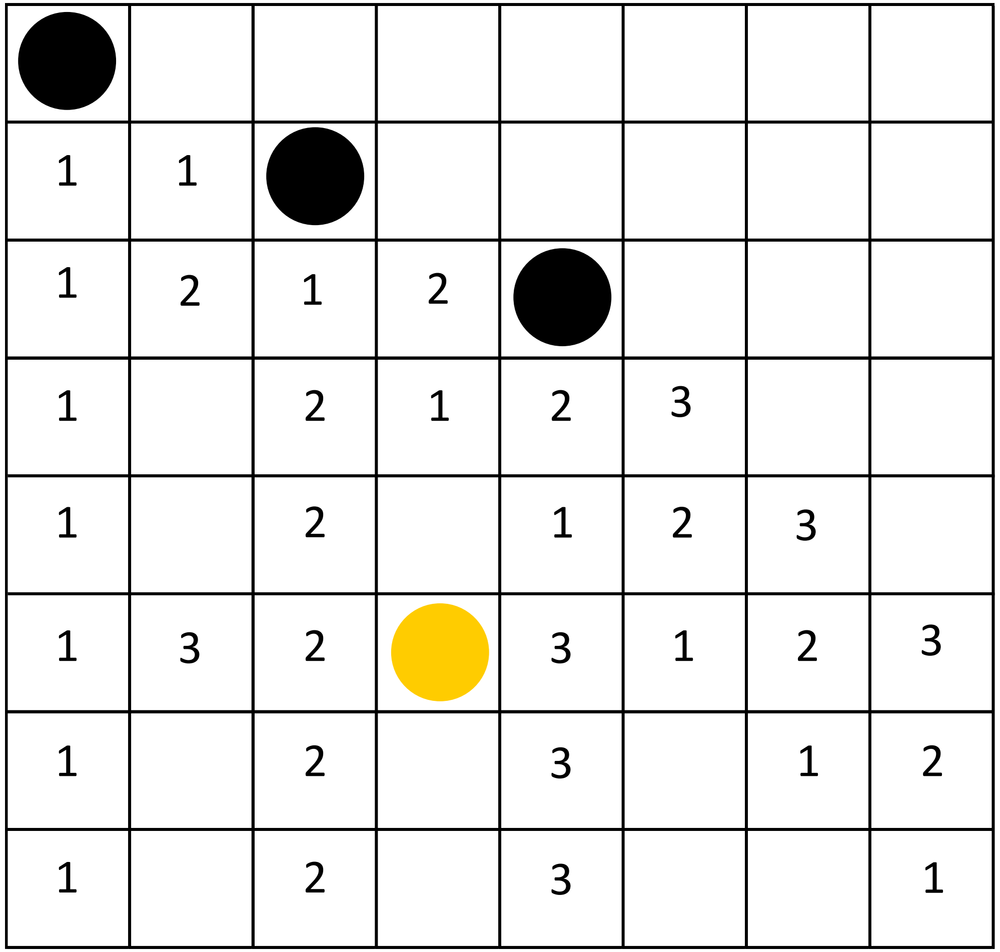
Forward Checking



Tests 92 + 4 * 5 = 112 Backtracks 0

Forward Checking

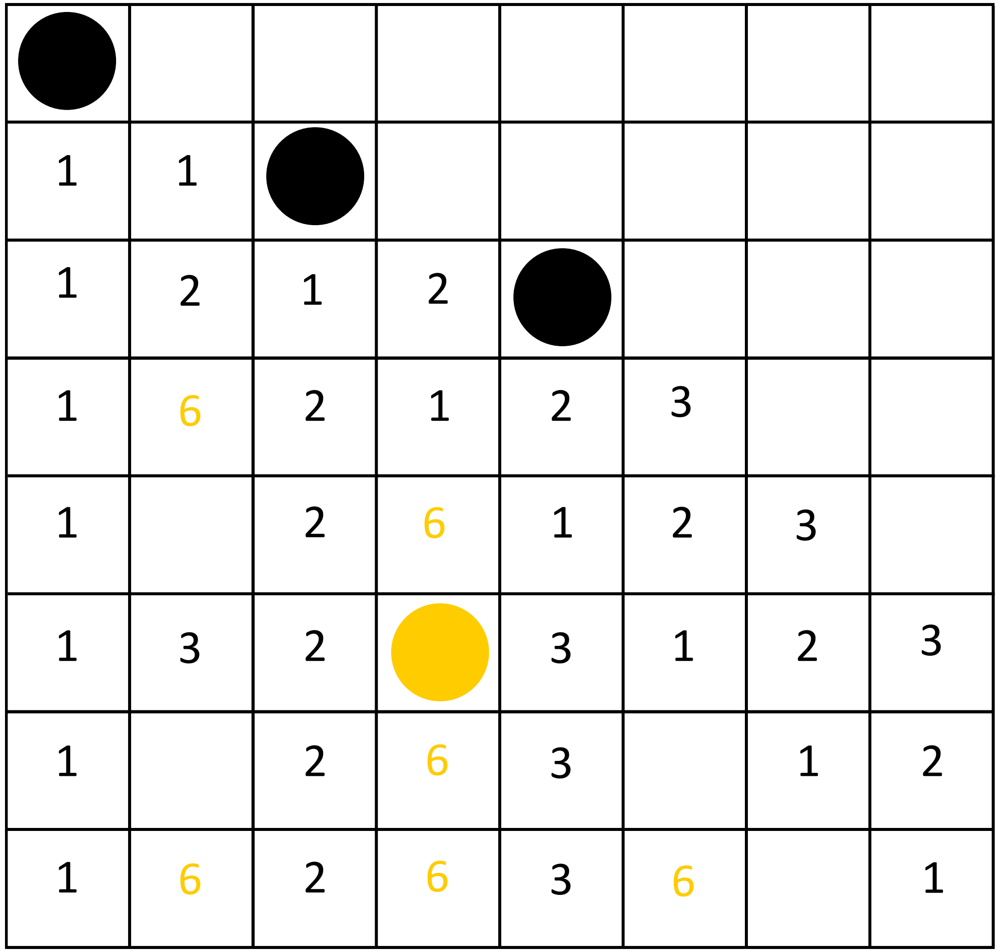
X_6 can only take value 4



Tests 92 + 4 * 5 = 112

Backtracks 0

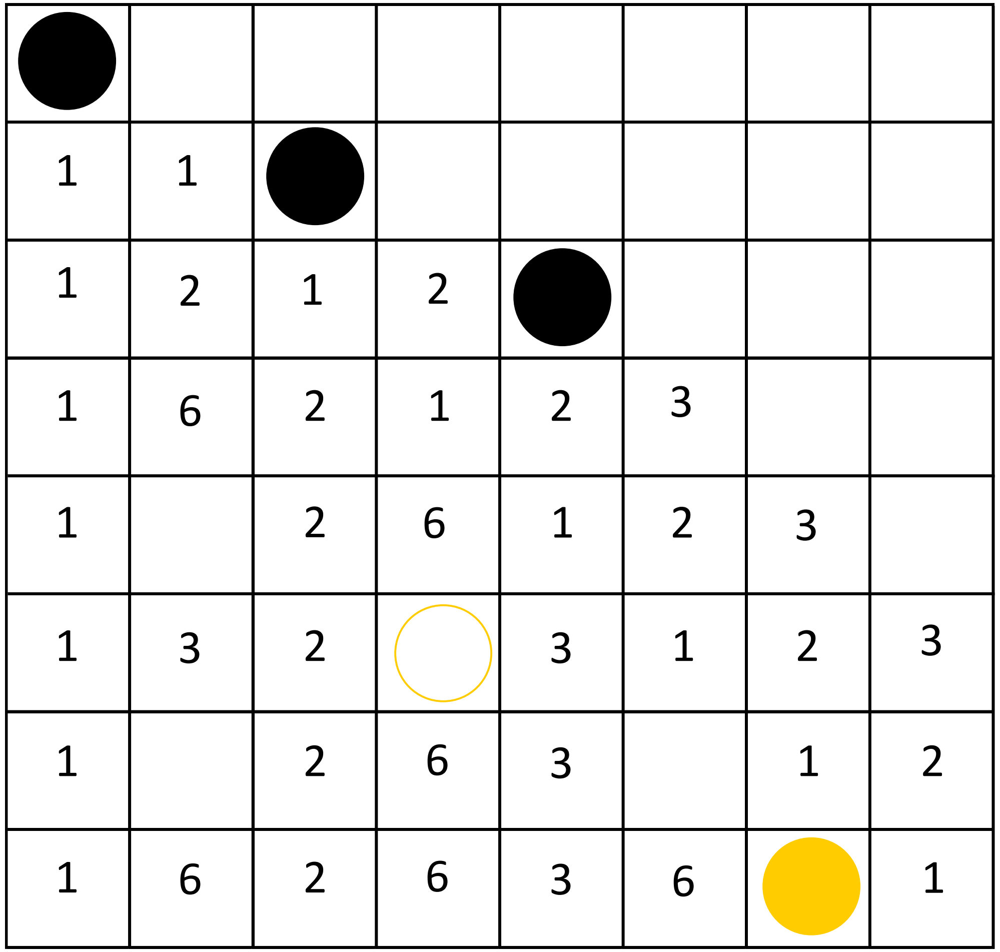
Forward Checking



Tests 112+3+3+3+4 = 125 Backtracks 0

Forward Checking

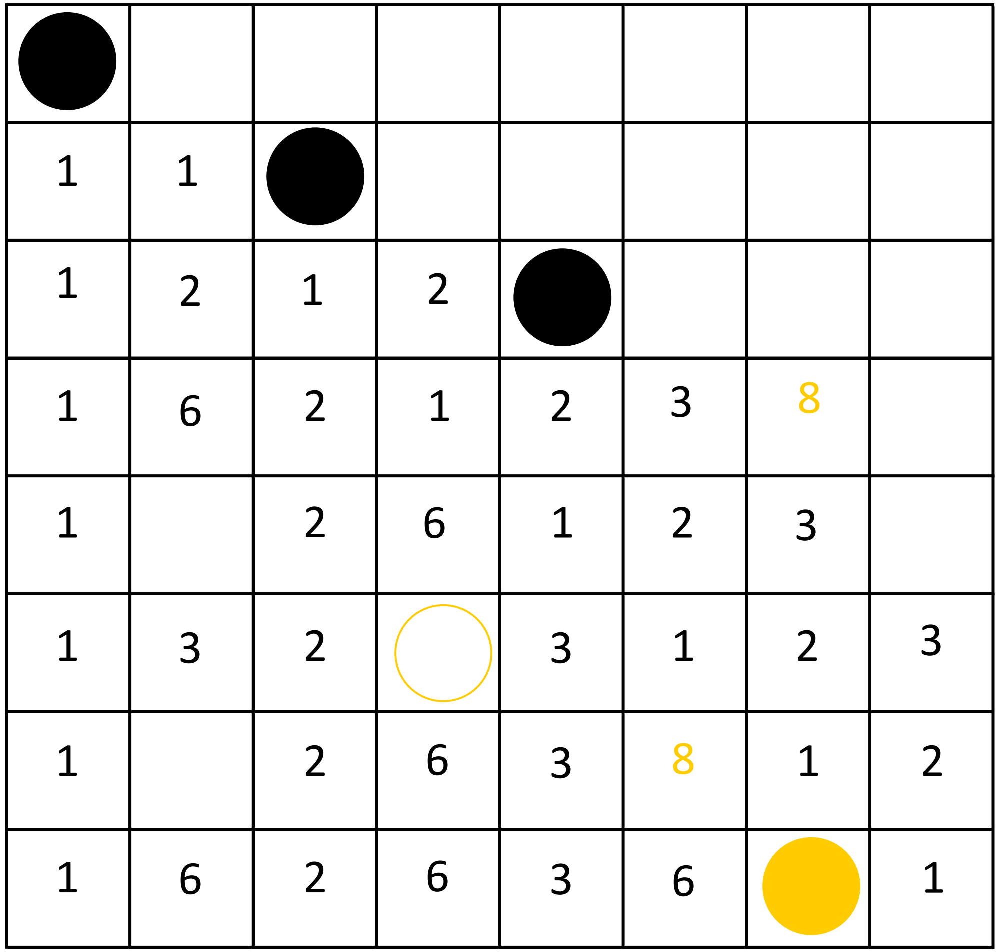
X_8 can only take value 7



Tests 125

Backtracks 0

Forward Checking



Tests 125+2+2+2=131

Backtracks 0

Forward Checking

X_4 can 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	6	3	6		1

Tests 131

Backtracks 0

Forward Checking

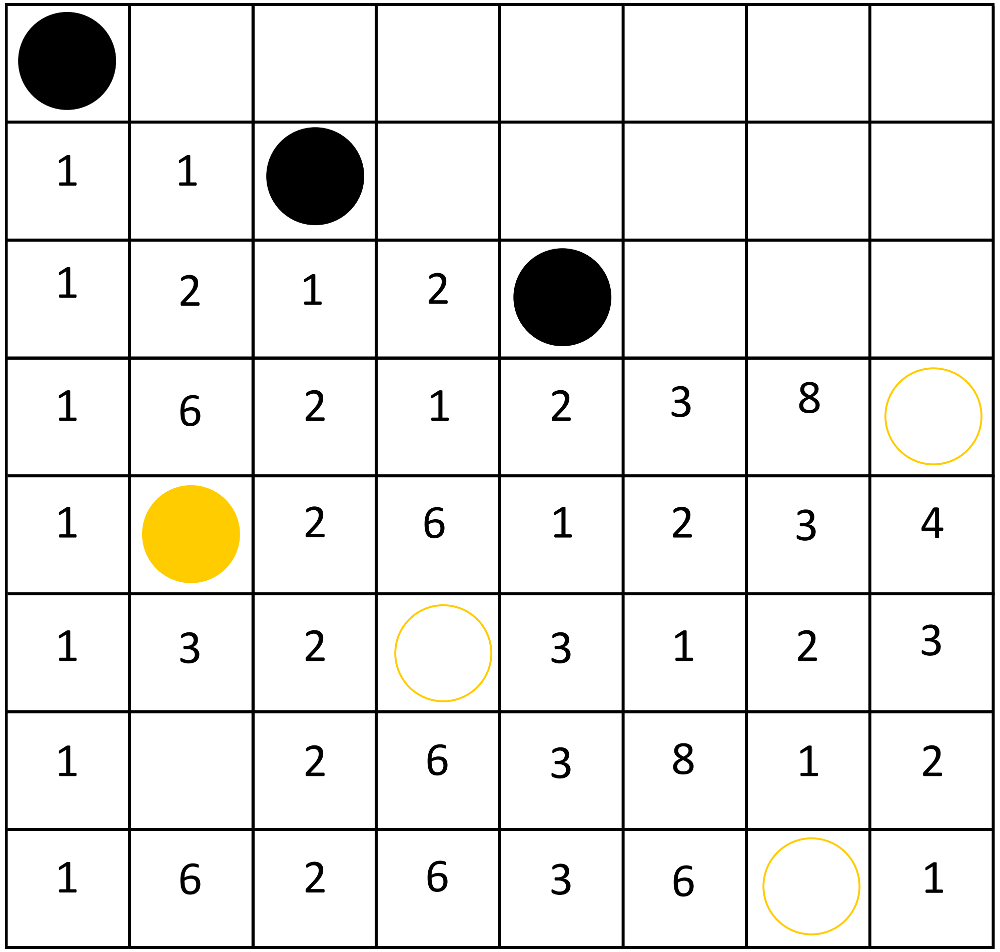
1	1						
1	2	1	2				
1	6	2	1	2	3	8	
1		2	6	1	2	3	4
1	3	2		3	1	2	3
1		2	6	3	8	1	2
1	6	2	6	3	6		1

Tests 131+2+2=135

Backtracks 0

Forward Checking

X_5 can only take value 2



Tests 135

Backtracks 0

Forward Checking

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	6	3	6		1

Tests 135+1=136 Backtracks 0

Forward Checking

●							
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	6	3	6	○	1

Tests 136 Backtracks 0

Forward Checking

Fails

7

Backtracks

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	6	3	6		1

Tests 136

Backtracks 0+1=1

Forward Checking

$X_1=1$

$X_2=3$

$X_3=5$

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

(15)

Tests 136

Backtracks 1