Discrete Optimization

Constraint Programming: Part IX

Goals of the lecture

- Search in constraint programming
  - introduction
  - active research area

Search in constraint programming

- Key idea
  - use feasibility information for branching

Search in constraint programming

- Key idea
  - use feasibility information for branching
- First-fail principle
  - try first where you are the most likely to fail

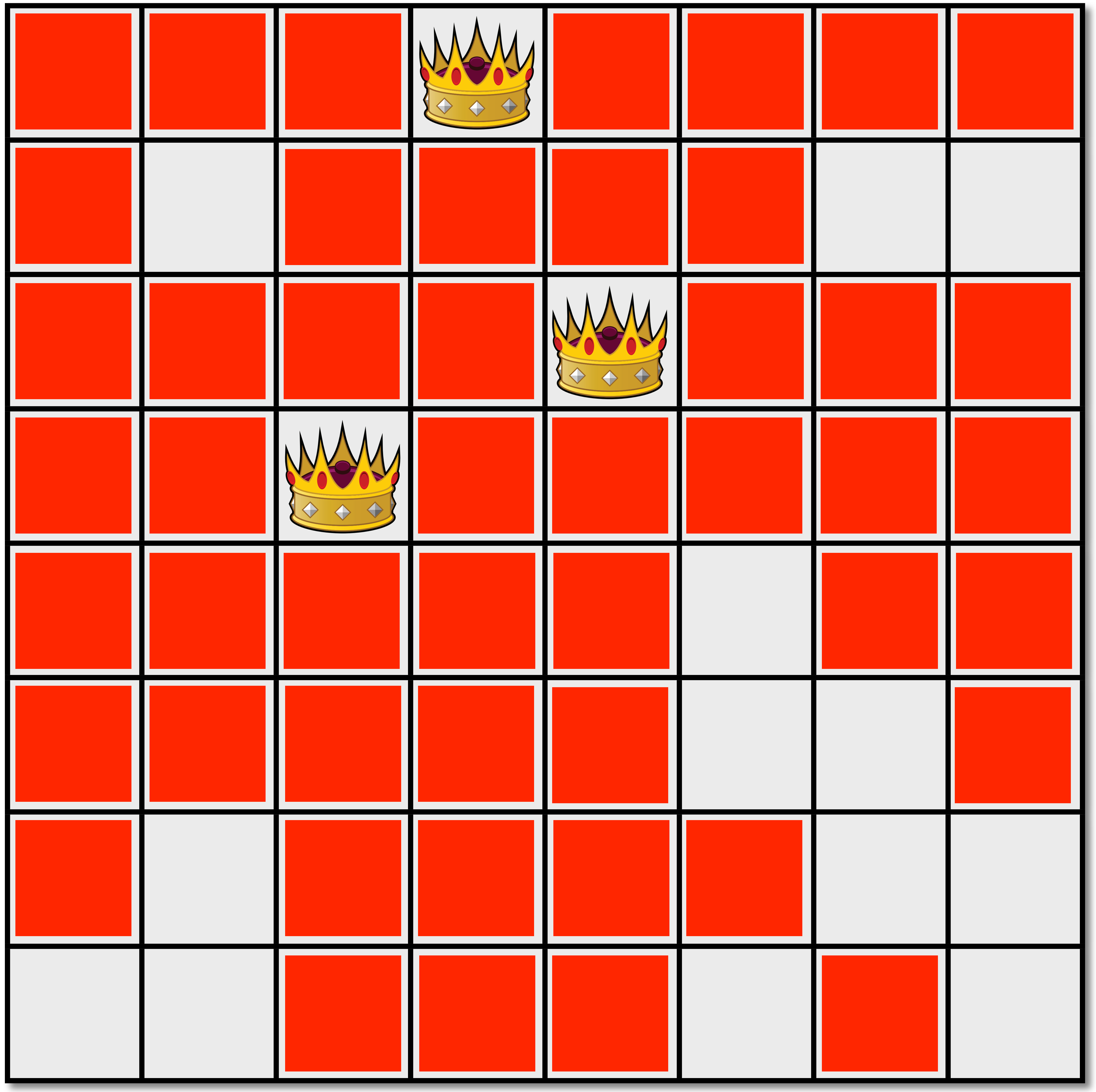
Search in constraint programming

- Key idea
  - use feasibility information for branching
- First-fail principle
  - try first where you are the most likely to fail
- Why the first-fail principle?
  - do not spend time doing easy stuff first and avoid redoing the difficult part

Search in constraint programming

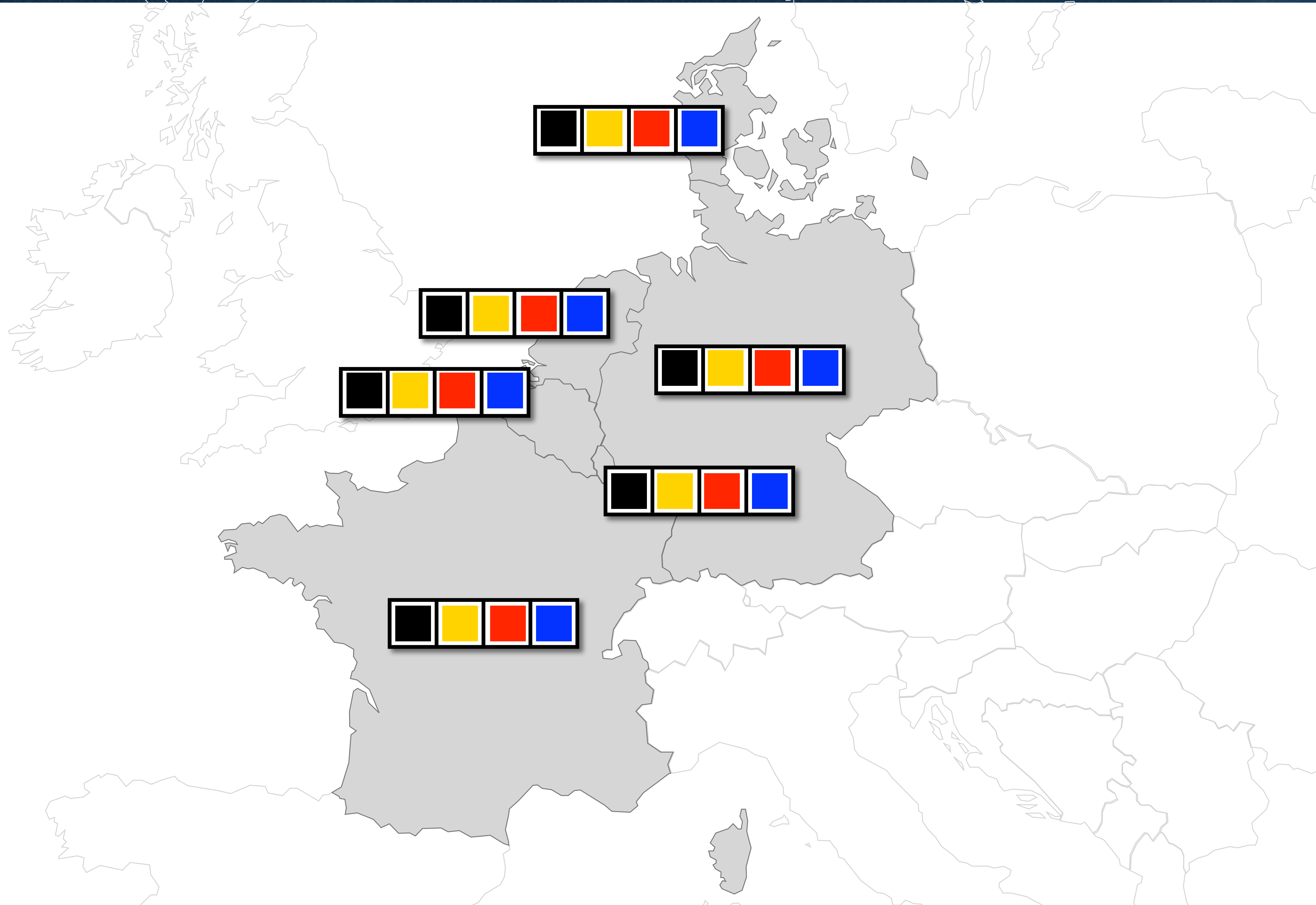
- Key idea
  – use feasibility information for branching
- First-fail principle
  – try first where you are the most likely to fail
- Why the first-fail principle?
  – do not spend time doing easy stuff first and avoid redoing the difficult part
- The ultimate goal
  – creating small search trees

How to approximate the first-fail principle?



Coloring a Map

Coloring a Map



Euler Knight

- The problem
  - use a knight to visit all positions of a chessboard exactly once

- Puzzle with links
  - vehicle routing problems

Euler Knight

```
range Board = 1..64;
var{int} jump[i in Board] in Knightmoves(i);
solve {
   circuit(jump);
}
```

Euler Knight

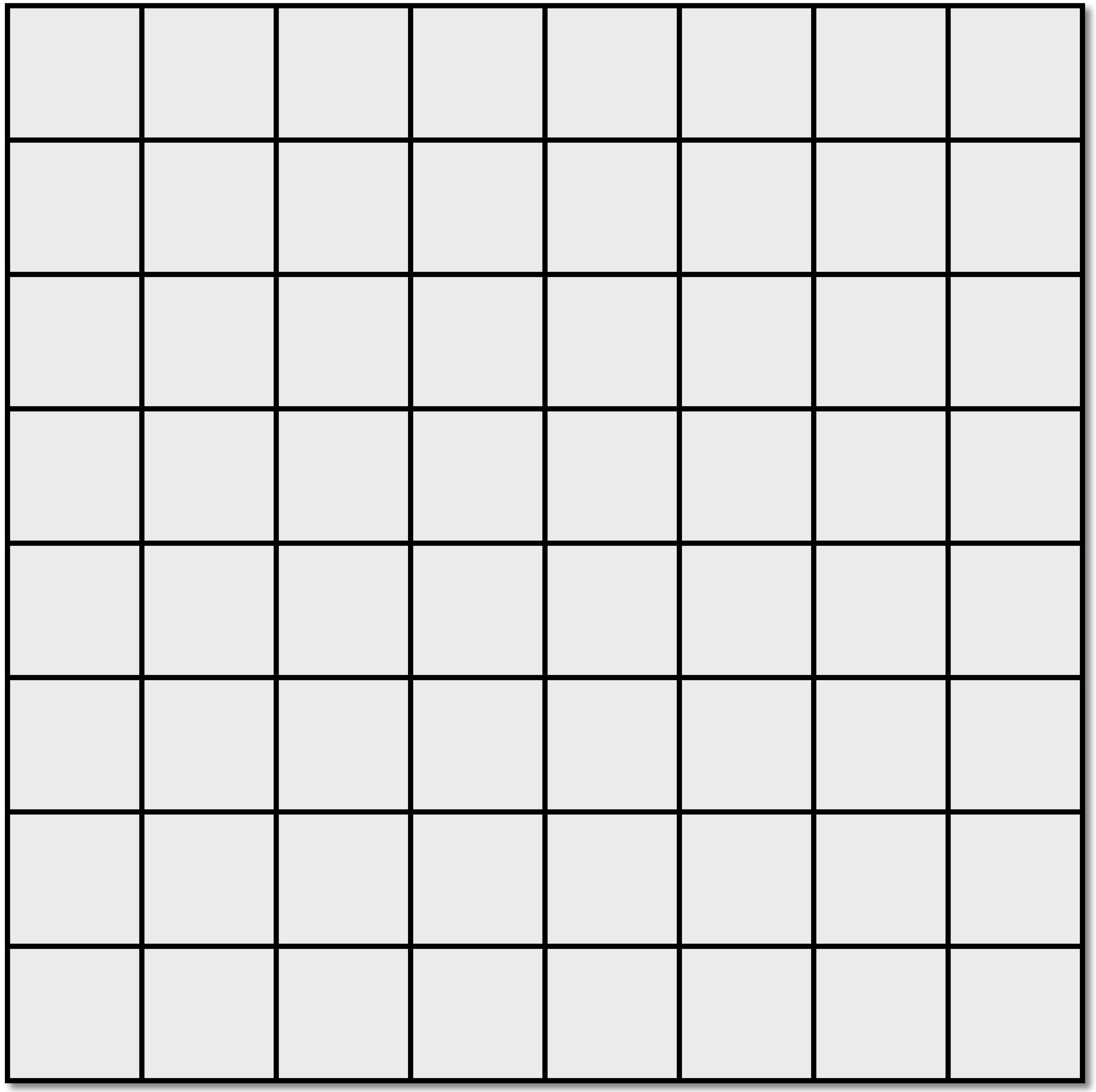
```
range Board = 1..64;
var{int} jump[i in Board] in Knightmoves(i);
solve {
   circuit(jump);
}
```

```
function set{int} Knightmoves(int i) {
   set{int} S;
   if (i % 8 == 1)
      S = {i-15,i-6,i+10,i+17};
   else if (i % 8 == 2)
      S = {i-17,i-15,i-6,i+10,i+15,i+17};
   else if (i % 8 == 7)
      S = {i-17,i-15,i-10,i+6,i+15,i+17};
   else if (i % 8 == 0)
      S = {i-17,i-10,i+6,i+15};
   else
      S = {i-17,i-15,i-10,i-6,i+6,i+10,i+15,i+17};
   return filter(v in S) (v >= 1 && v <= 64);
}
```

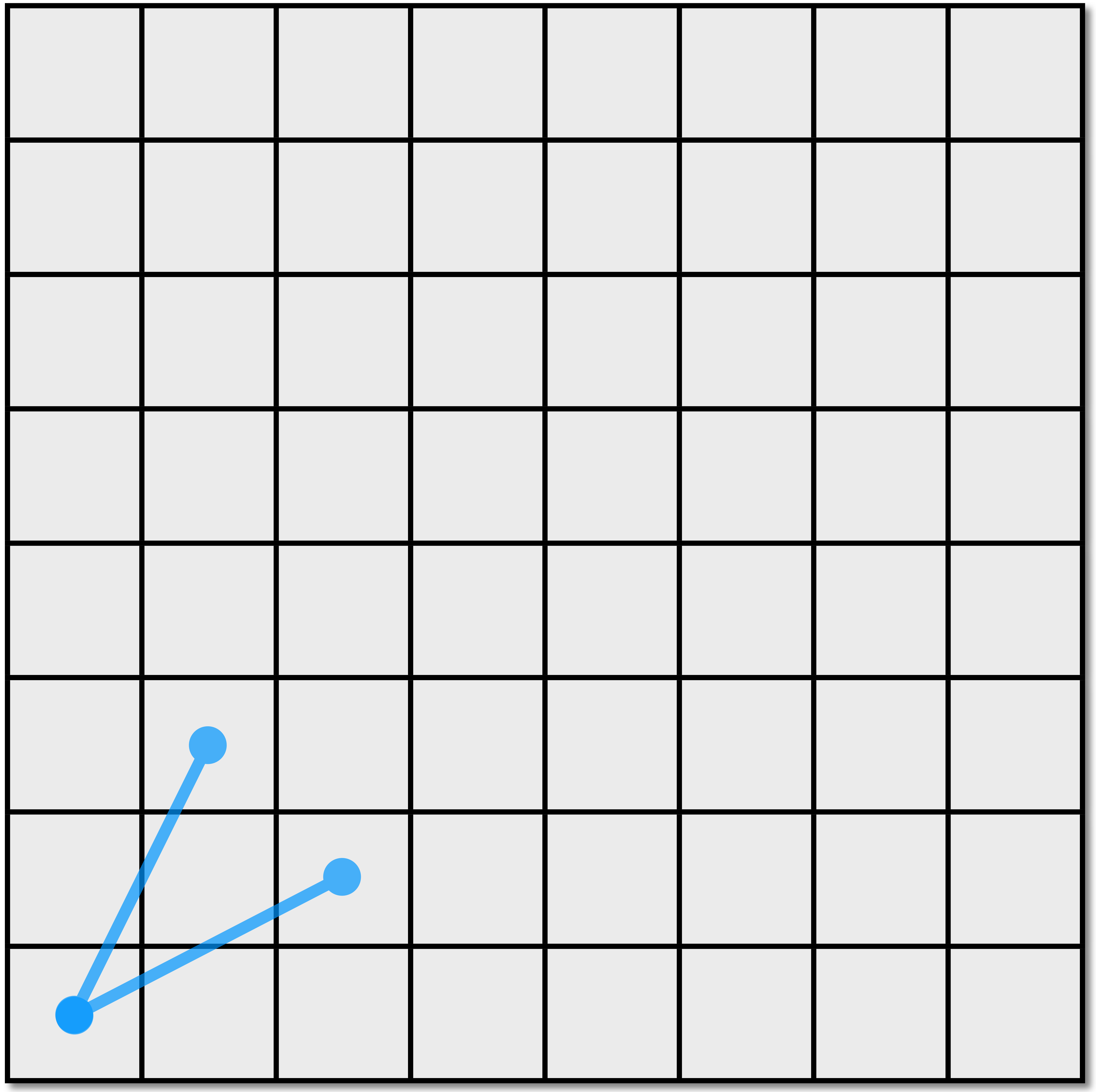
Euler Knight

- First-fail principle on the Euler Knight
  – where do we start?

Euler Knight



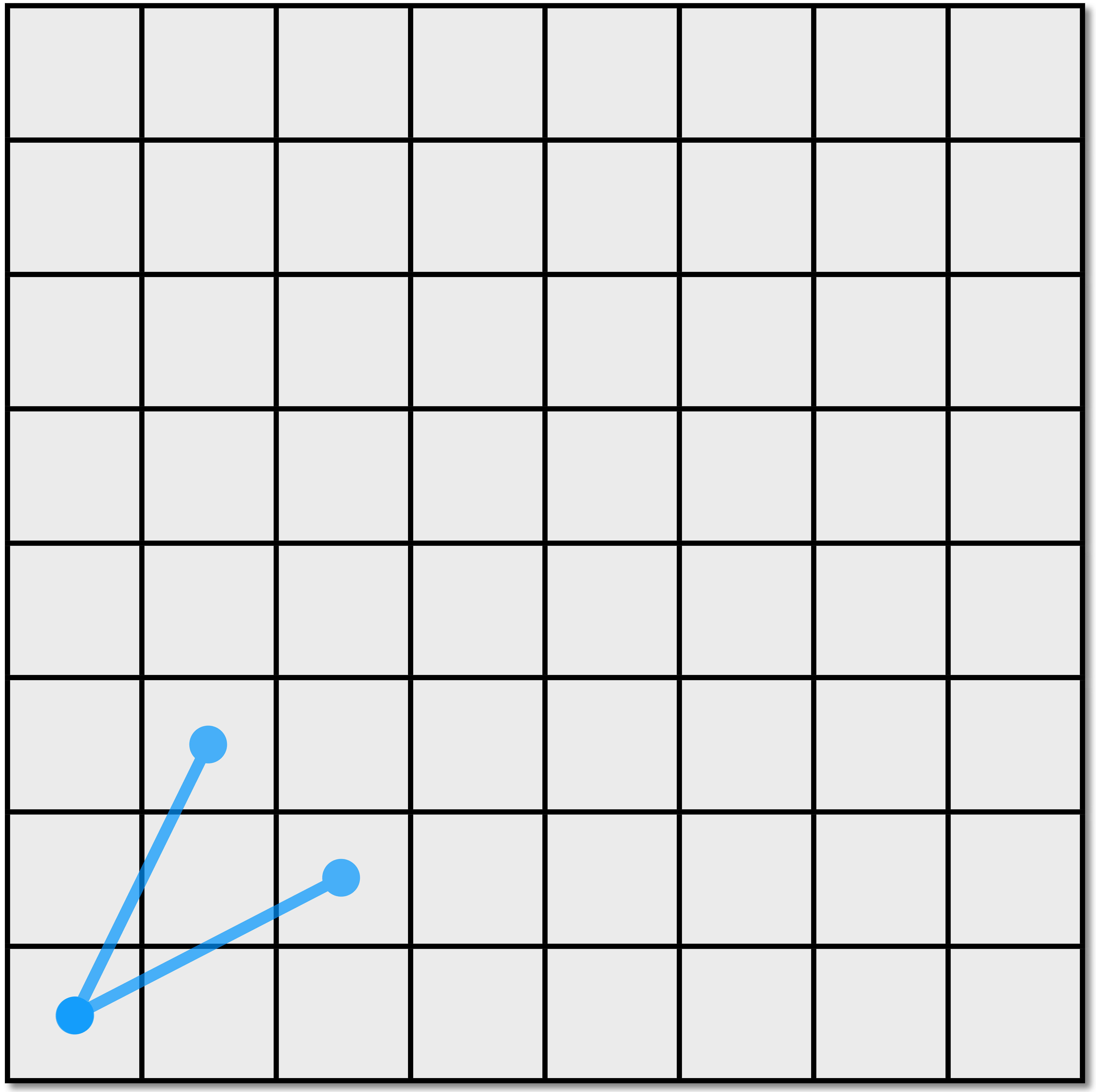
Euler Knight



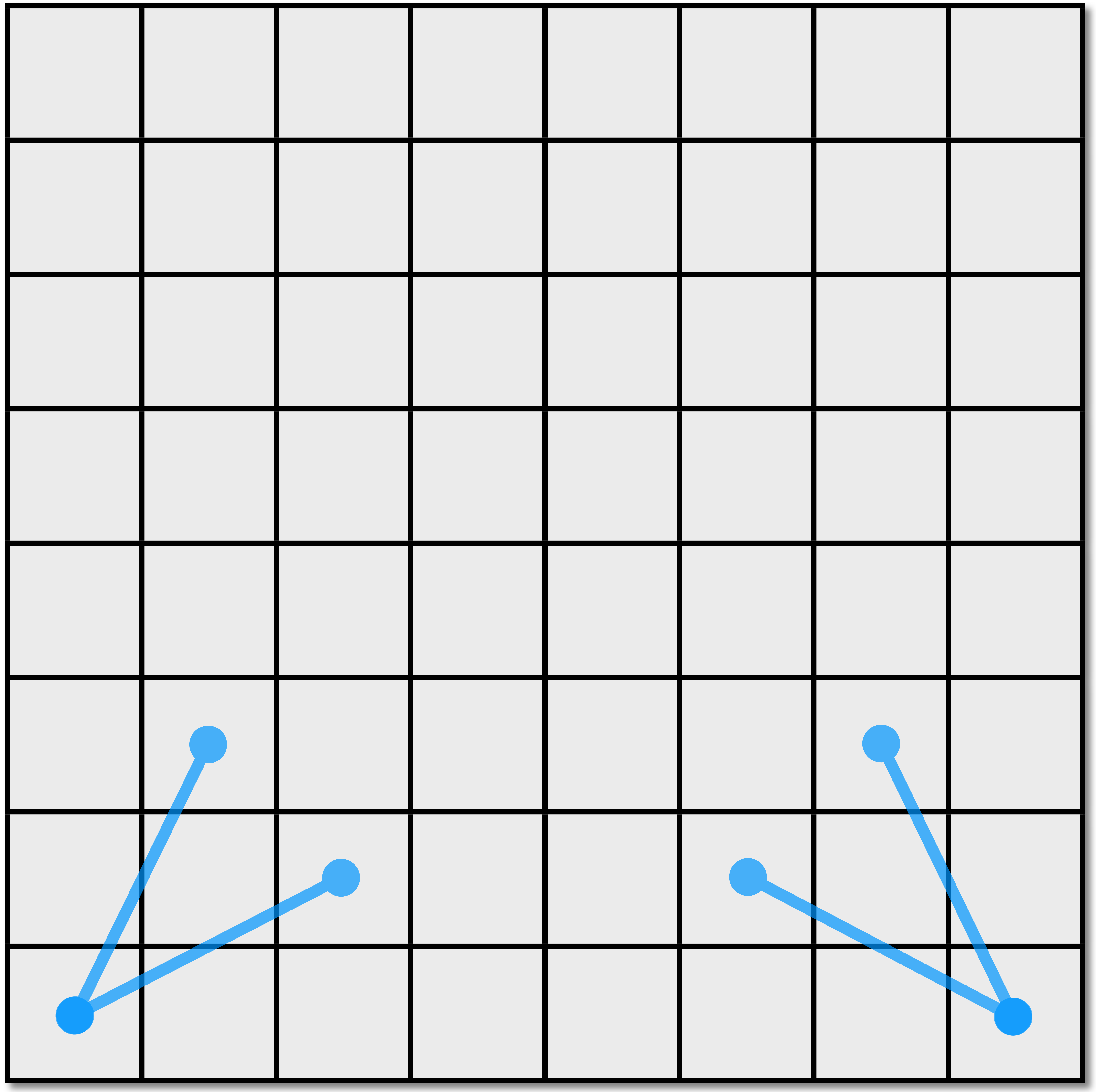
Euler Knight

- First-fail principle on the Euler Knight
  - where do we go next?

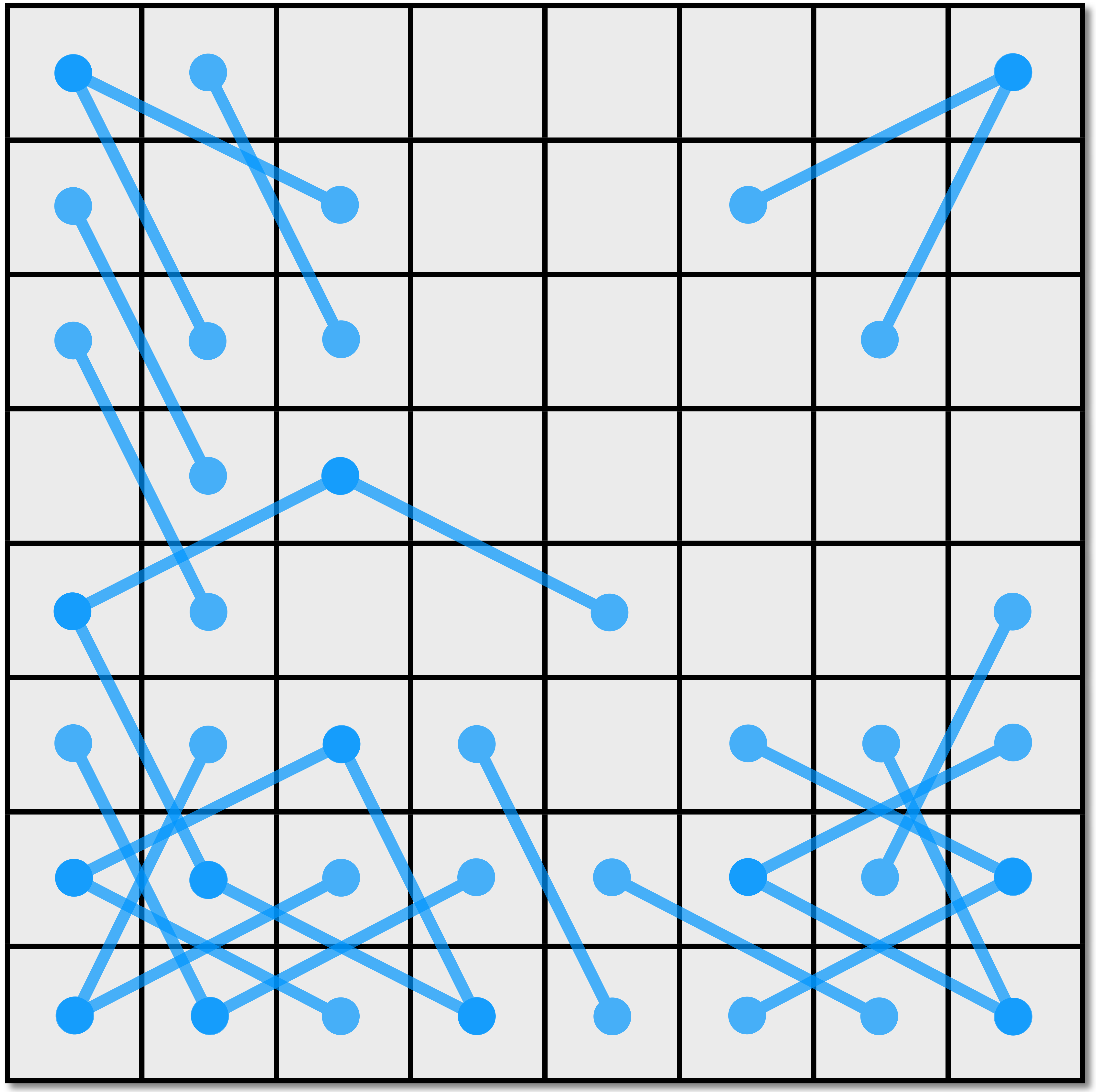
Euler Knight



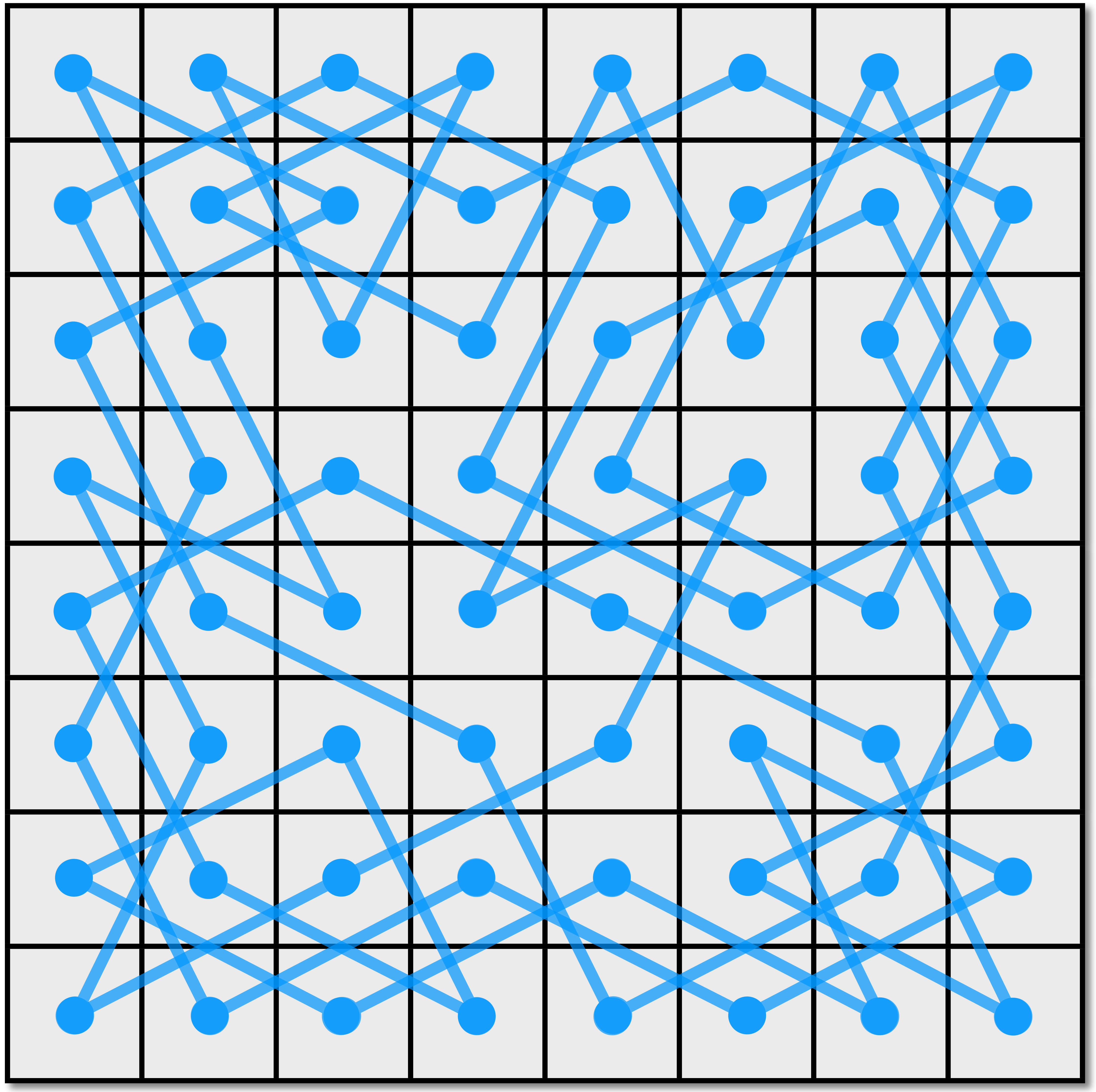
Euler Knight



Euler Knight



Euler Knight



A search procedure for the 8-queens problem

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
      tryall(v in R)
         row[r] = v;
}
```

A search procedure for the 8-queens problem

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
      tryall(v in R)
         row[r] = v;
}
```

iterate over all queens

A search procedure for the 8-queens problem

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
      tryall(v in R)
         row[r] = v;
}
```

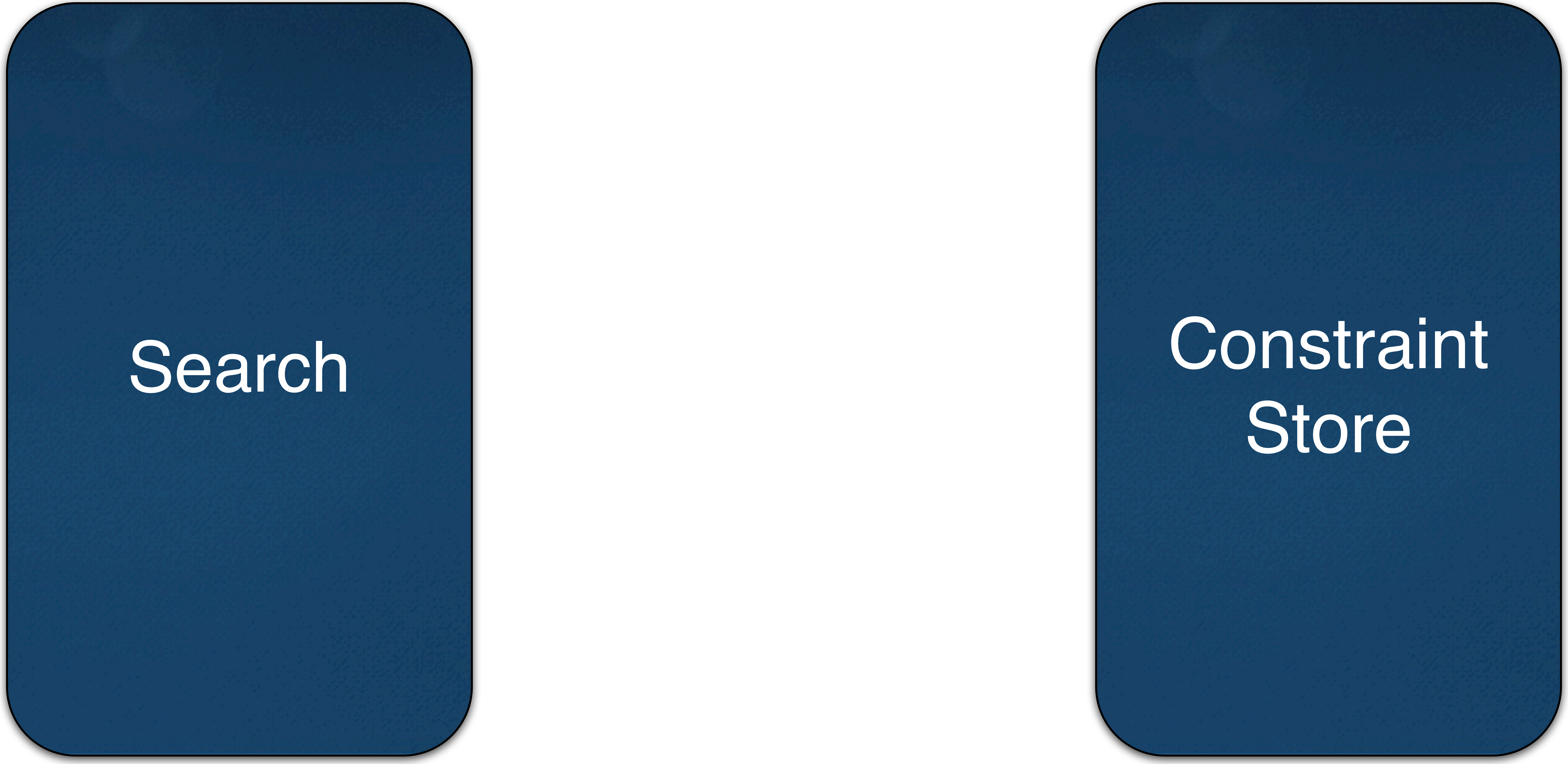
nondeterministically explore all values

A search procedure for the 8-queens problem

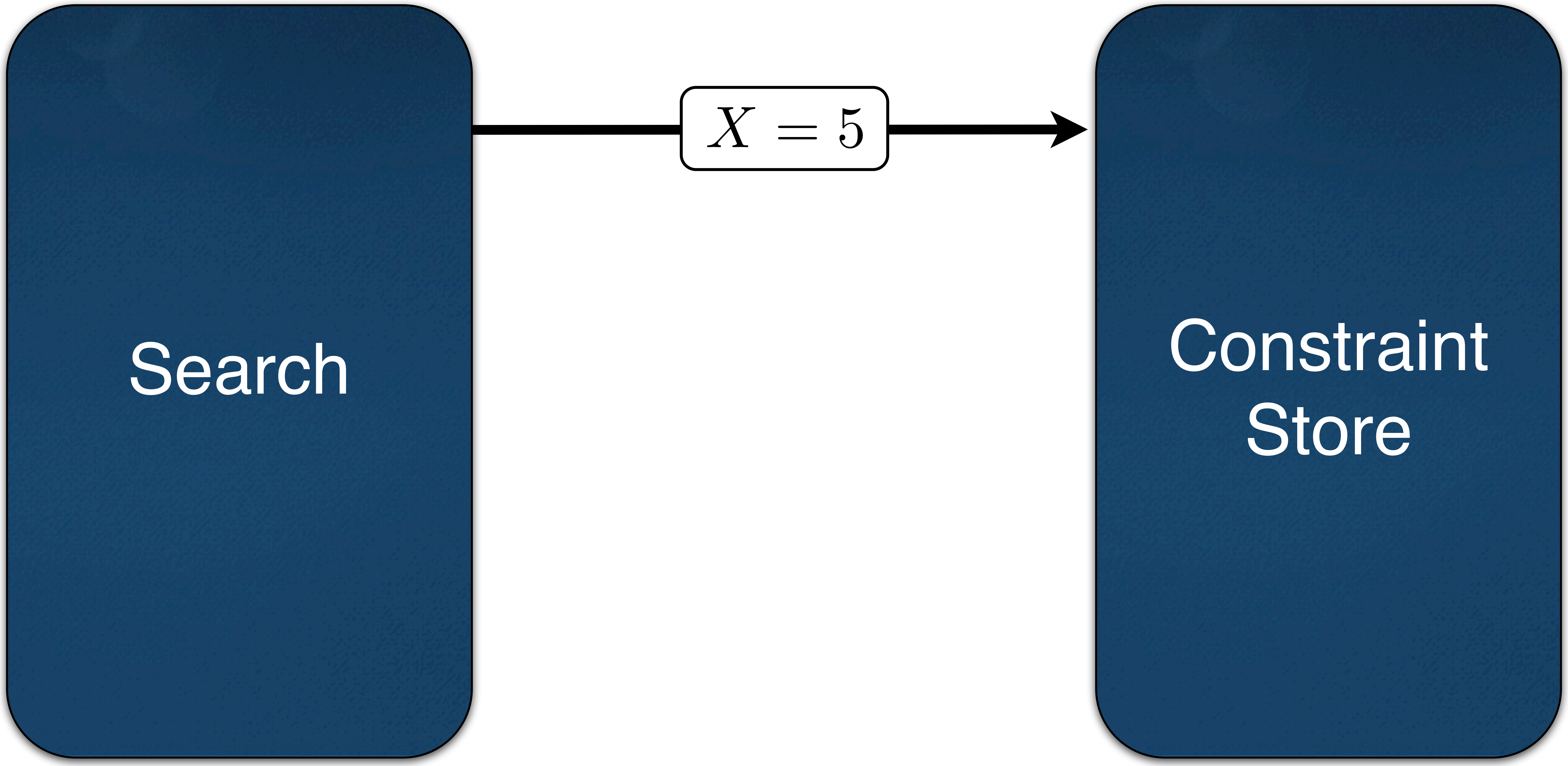
```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
      tryall(v in R)
         row[r] = v;
}
```

add a constraint to the constraint store

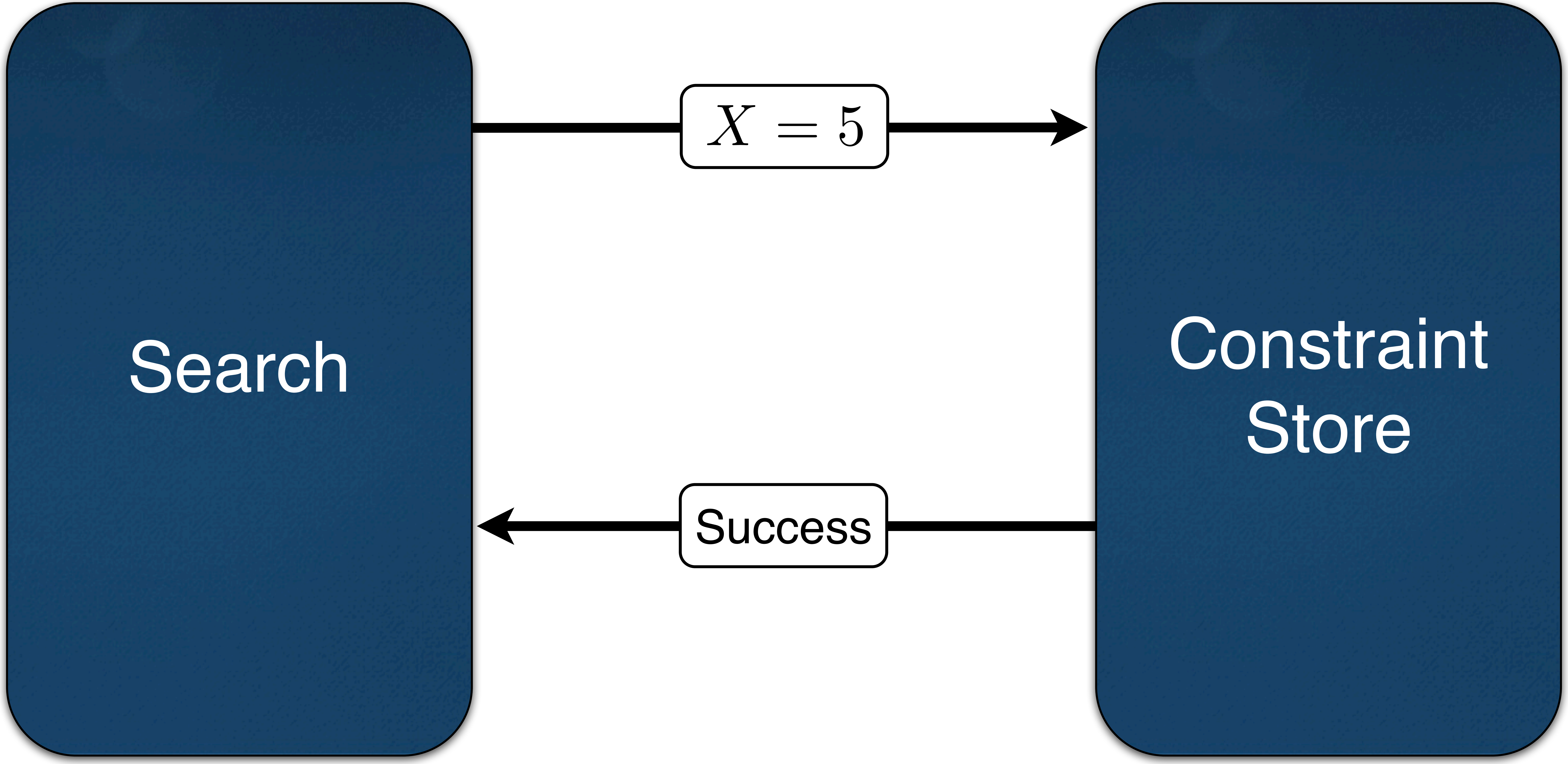
Computational Paradigm



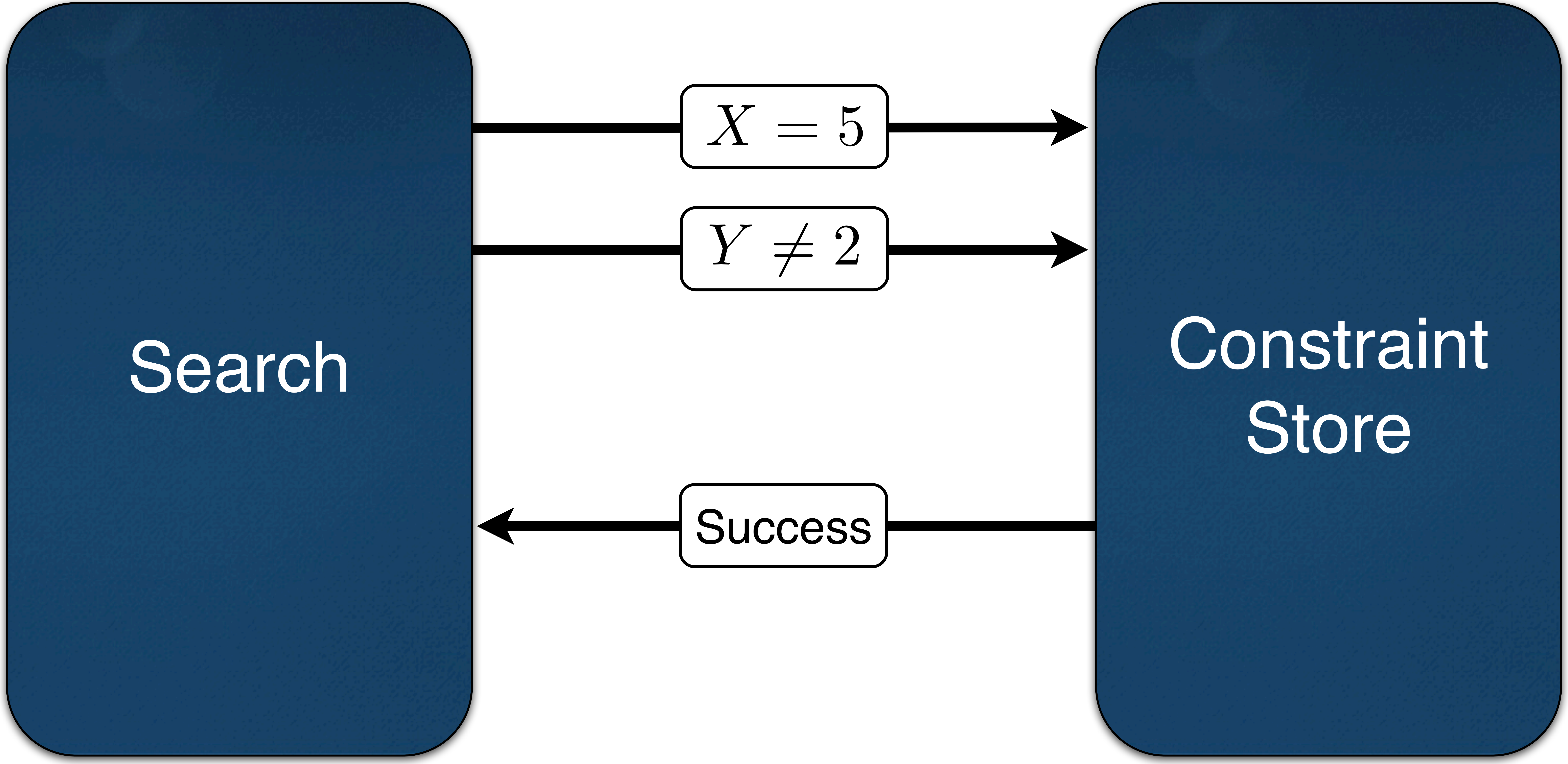
Computational Paradigm



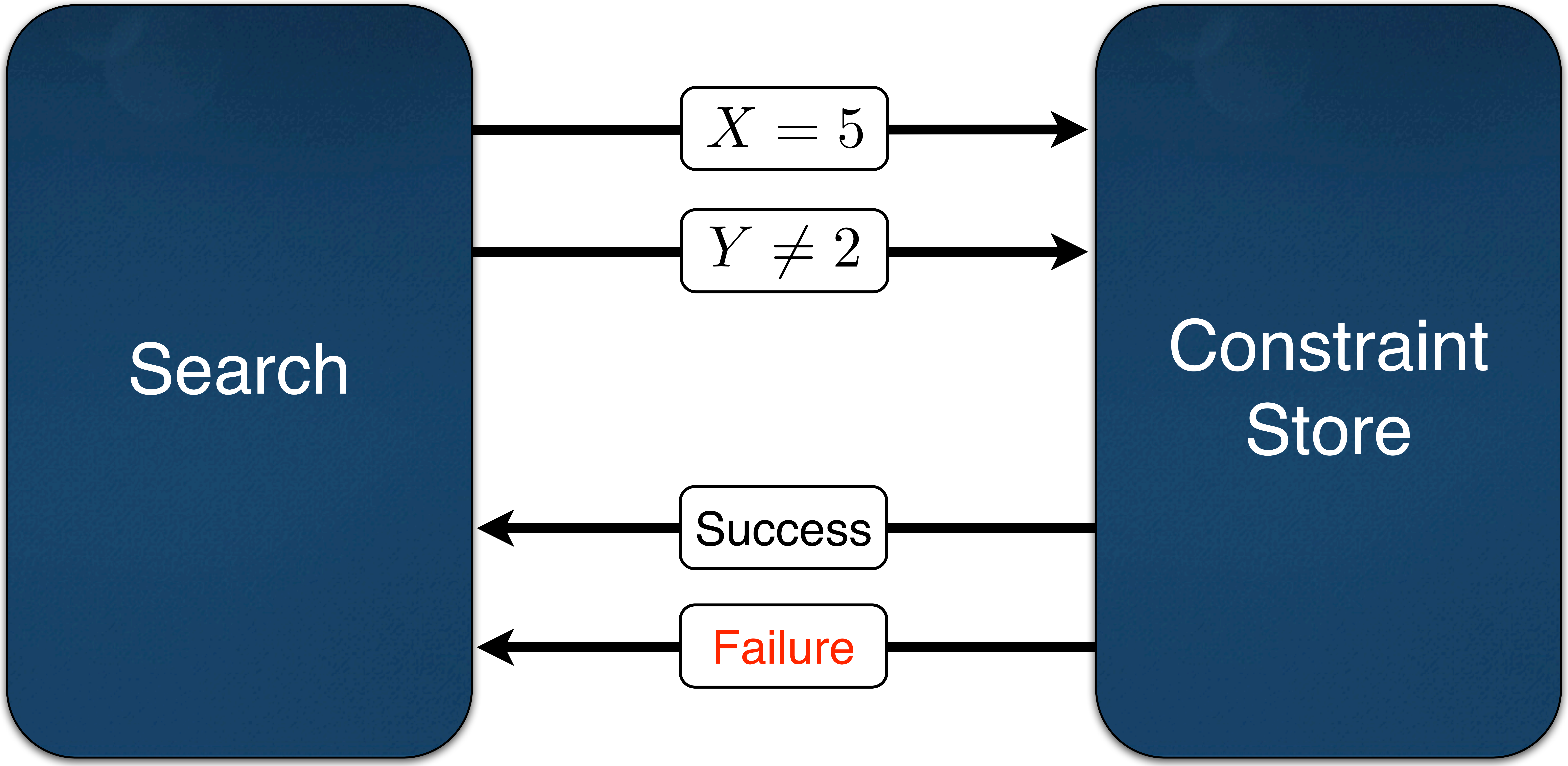
Computational Paradigm



Computational Paradigm



Computational Paradigm



Nondeterministic choice

- When a constraint fails
  - that is, when adding a constraint to the constraint store returns a failure
- the solver goes back to the last tryall
  - and assigns a value that has not been tried before
  - if no such value is left, the system backtracks to an earlier nondeterministic instruction

Understanding nondeterministic computations

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
      tryall(v in R)
         row[r] = v;
}
```

Understanding nondeterministic computations

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
      tryall(v in R)
         row[r] = v;
}
```

iterate over all queens

Understanding nondeterministic computations

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   tryall(v in R) row[1] = v;
   tryall(v in R) row[2] = v;
   tryall(v in R) row[3] = v;
   tryall(v in R) row[4] = v;
   tryall(v in R) row[5] = v;
   tryall(v in R) row[6] = v;
   tryall(v in R) row[7] = v;
   tryall(v in R) row[8] = v;
}
```

Understanding nondeterministic computations

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
      tryall(v in R)
         row[r] = v;
}
```

Understanding nondeterministic computations

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
     tryall(v in R)
         row[r] = v;
}
```

nondeterministically explore all values

Understanding nondeterministic computations

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
     try row[r] = 1;
     |   row[r] = 2;
     |   row[r] = 3;
     |   row[r] = 4;
     |   row[r] = 5;
     |   row[r] = 6;
     |   row[r] = 7;
     |   row[r] = 8;
     endtry;
}
```

Understanding nondeterministic computations

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
     try row[r] = 1;
     |   row[r] = 2;
     |   row[r] = 3;
     |   row[r] = 4;
     |   row[r] = 5;
     |   row[r] = 6;
     |   row[r] = 7;
     |   row[r] = 8;
     endtry;
}
```

Searching in constraint programming

- variable/value labeling
- value/variable labeling
- domain splitting
- focusing on the objective
- symmetry breaking during search
- randomization and restarts

Variable/Value Labeling

- Two steps
  - choose the variable to assign next
  - choose the value to assign

Variable/Value Labeling

- Two steps
  - choose the variable to assign next
  - choose the value to assign
- First-fail principle
  - choose the variable with the smallest domain

Variable/Value Labeling

- Two steps
  - choose the variable to assign next
  - choose the value to assign
- First-fail principle
  - choose the variable with the smallest domain
- The variable ordering is dynamic
  - reconsider the selection after each choice

Variable/Value Labeling

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R) by row[r].getSize()
      tryall(v in R)
         row[r] = v;
}
```

Variable/Value Labeling

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R) by row[r].getSize()
      tryall(v in R)
         row[r] = v;
}
```

select first the variable with the smallest domain

Variable/Value Labeling

- Two steps
  - choose the variable to assign next
  - choose the value to assign
- First-fail principle
  - choose the variable with the smallest domain
  - choose the most constrained variable

Variable/Value Labeling

- Two steps
  - choose the variable to assign next
  - choose the value to assign
- First-fail principle
  - choose the variable with the smallest domain
  - choose the most constrained variable
- Use a lexicographic criterion
  - first the domain size
  - next the proximity to the middle of the board

Lexicographic ordering

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
     by (row[r].getSize(),abs(r-n/2))
      tryall(v in R)
         row[r] = v;
}
```

Dynamic orderings for variable and value choices

```
range R = 1..8;
range C = 1..8;
var{int} row[C] in R;
var{int} col[R] in C
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
   forall(i in C,j in C: i < j) {
      col[i] ≠  col[j];
      col[i] ≠  col[j] + (j - i);
      col[i] ≠  col[j] - (j - i);
   }
   forall(r in R,c in C)
      (row[c] = r) <=> (col[r] = c);
}
using {
   forall(r in R) by row[r].getSize()
      tryall(v in R) by col[v].getSize()
         row[r] = v;
}
```

Variable/Value Labeling

- Variable ordering
  - choose the most constrained variable
  - e.g., smallest domain, variable that fails often

Variable/Value Labeling

- Variable ordering
  – choose the most constrained variable
  – e.g., smallest domain, variable that fails often
- Value ordering
  – often choose a value that leaves as many options as possible to the other variables
  – this helps finding a solution

Feasibility versus Optimality

- strong focus on feasibility branching
  - the pruning algorithm provides a lot of information

Feasibility versus Optimality

- strong focus on feasibility branching
  - the pruning algorithm provides a lot of information
- may also focus on the objective function
  - does not change the way search works

The ESDD Deployment Problem

- Generalized quadratic assignment problem
  - f: the communication frequency matrix
  - h: the distance matrix (in hops)
  - x: the assignment vector (decision variables)
  - C: sets of components
  - Sep: separation constraints
  - Col: colocation constraints
  - objective function

$$\min_{x \in \mathbb{N}^n} \sum_{a \in C} \sum_{b \in C} f_{a,b} \cdot h_{x_a, x_b}$$

The CP Model

```
minimize
    sum(a in C,b in C: a != b) f[a,b]*h[x[a],x[b]]
subject to {
   forall(S in Col,c1 in S,c2 in S: c1 < c2)
     x[c1] = x[c2];
   forall(S in Sep)
     alldifferent(all(c in S) x[c]);
}
using {
 while (!bound(x))
   selectMax(i in C:!x[i].bound(),j in C)(f[i,j])
      tryall(n in N) by (min(l in x[j].memberOf(l)) h[n,l])
         x[i] = n;
}
```

The CP Model

```
minimize
    sum(a in C,b in C: a != b) f[a,b]*h[x[a],x[b]]
subject to {
   forall(S in Col,c1 in S,c2 in S: c1 < c2)
     x[c1] = x[c2];
   forall(S in Sep)
     alldifferent(all(c in S) x[c]);
}
using {
 while (!bound(x))
   selectMax(i in C:!x[i].bound(),j in C)(f[i,j])
      tryall(n in N) by (min(l in x[j].memberOf(l)) h[n,l])
         x[i] = n;
}
```

select a component i to assign that has the largest communication frequency

The CP Model

```
minimize
    sum(a in C,b in C: a != b) f[a,b]*h[x[a],x[b]]
subject to {
   forall(S in Col,c1 in S,c2 in S: c1 < c2)
     x[c1] = x[c2];
   forall(S in Sep)
     alldifferent(all(c in S) x[c]);
}
using {
 while (!bound(x))
   selectMax(i in C:!x[i].bound(),j in C)(f[i,j])
      tryall(n in N) by (min(l in x[j].memberOf(l)) h[n,l])
         x[i] = n;
}
```

try the possible value starting first with those minimizing the number of hops to j