Module 6

BackTracking
N-Queen Problem

Backtracking

- Suppose you have to make a series of *decisions,* among various *choices,* where
  - You don't have enough information to know what to choose
  - Each decision leads to a new set of choices
  - Some sequence of choices (possibly more than one) may be a solution to your problem
- **Backtracking** is a methodical way of trying out various sequences of decisions, until you find one that "works"

Introduction

- Backtracking is used to solve problems in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion.
- Backtracking is a modified depth-first search of a tree.
- Backtracking involves only a tree search.
- Backtracking is the procedure whereby, after determining that a node can lead to nothing but dead nodes, we go back (“backtrack”) to the node’s parent and proceed with the search on the next child.
- The term "backtrack" was coined by American mathematician D. H. Lehmer in the 1950s.

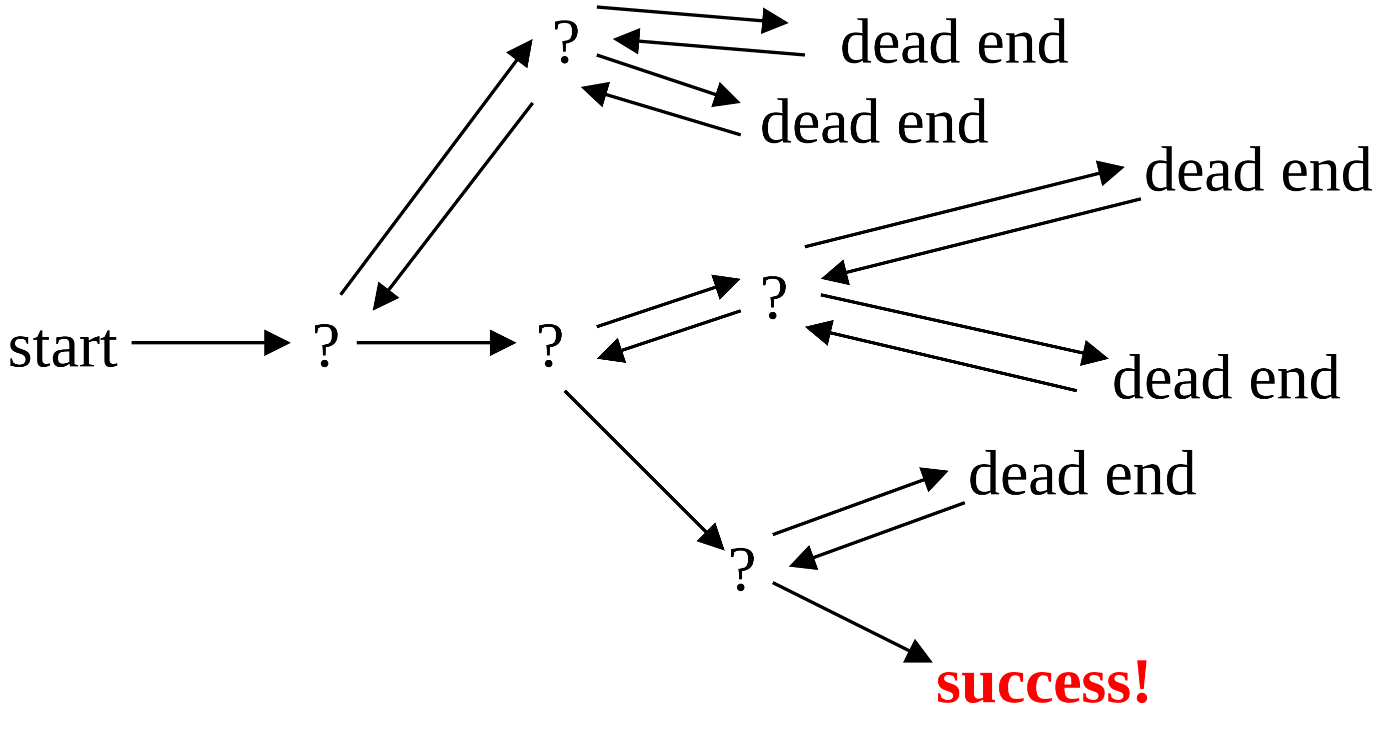
Introduction …

- We call a node nonpromising if when visiting the node we determine that it cannot possibly lead to a solution. Otherwise, we call it promising.
- In summary, backtracking consists of
  - Doing a depth-first search of a state space tree,
  - Checking whether each node is promising, and, if it is nonpromising, backtracking to the node's parent.
- This is called pruning the state space tree, and the subtree consisting of the visited nodes is called the pruned state space tree.

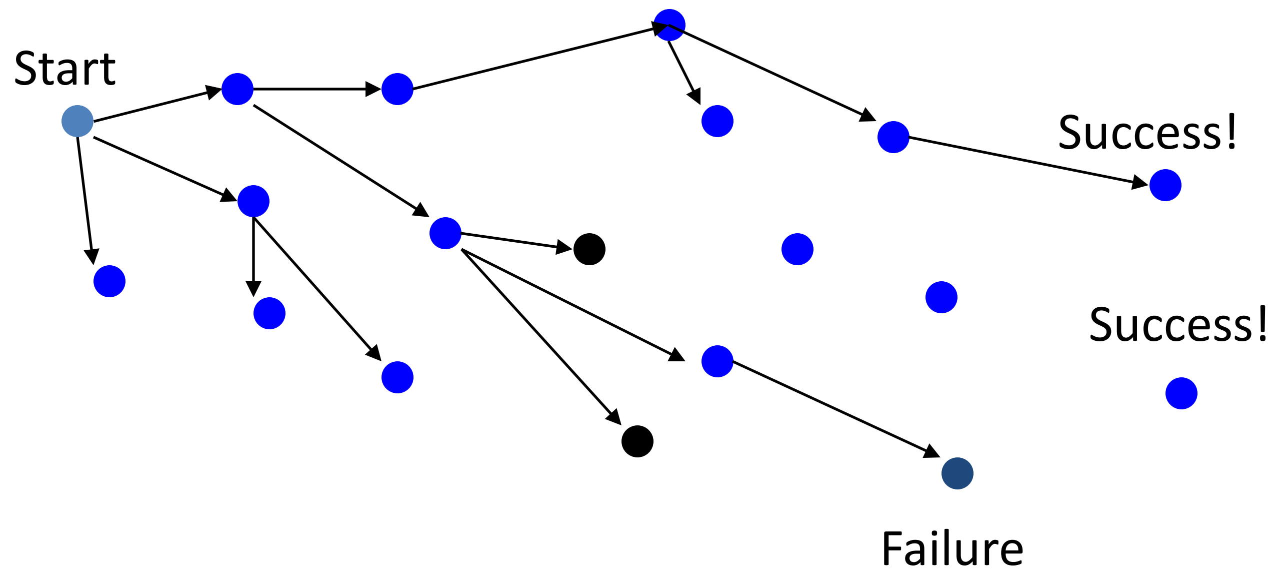
state space tree

- Construct the state space tree:
  - Root represents an initial state
  - Nodes reflect specific choices made for a solution's components.
    - Promising and nonpromising nodes
    - Leaves
- Explore the state space tree using depth-first search
- "Prune" non-promising nodes
  - dfs stops exploring subtree rooted at nodes leading to no solutions and...
  - "backtracks" to its parent node

Backtracking (animation)



Backtracking



Problem space consists of states (nodes) and actions (paths that lead to new states). When in a node can can only see paths to connected nodes

If a node only leads to failure go back to its "parent" node. Try other alternatives. If these all lead to failure then more backtracking may be necessary.

BACK TRACKING

- **Backtracking** is a general algorithm for finding all (or some) solutions to some computational problem, that *incrementally builds candidates to the solutions*, and abandons each partial candidate *‘c’* ("backtracks") as soon as it determines that *‘c’* cannot possibly be completed to a valid solution.

- Backtracking is an important tool for solving constraint satisfaction problems, such as *crosswords, verbal arithmetic, Sudoku, and many other puzzles.*

Control Abstraction

```
Algorithm Backtrack(k)
// This schema describes the backtracking process using
// recursion. On entering, the first k - 1 values
// x[1], x[2], ..., x[k - 1] of the solution vector
// x[1 : n] have been assigned. x[ ] and n are global.
{
    for (each x[k] ∈ T(x[1], ..., x[k - 1]) do
    {
        if (B_k(x[1], x[2], ..., x[k]) ≠ 0) then
        {
            if (x[1], x[2], ..., x[k] is a path to an answer node)
                then write (x[1 : k]);
            if (k < n) then Backtrack(k + 1);
        }
    }
}
```

```
Algorithm IBacktrack(n)
// This schema describes the backtracking process.
// All solutions are generated in x[1 : n] and printed
// as soon as they are determined.
{
    k := 1;
    while (k ≠ 0) do
    {
        if (there remains an untried x[k] ∈ T(x[1], x[2], ...,
            x[k − 1]) and B_k(x[1], ..., x[k]) is true) then
        {
            if (x[1], ..., x[k] is a path to an answer node)
                then write (x[1 : k]);
            k := k + 1; // Consider the next set.
        }
        else k := k − 1; // Backtrack to the previous set.
    }
}
```

N-Queens Problem

- Try to place N queens on an N * N board such that none of the queens can attack another queen.
- Remember that queens can move horizontally, vertically, or diagonally any distance.
- Let’s consider the 8 queen example…

WHAT IS 8 QUEEN PROBLEM?

- The **eight queens puzzle** is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens attack each other.
- Thus, a solution requires that no two queens share the same row, column, or diagonal.
- The eight queens puzzle is an example of the more general ***n*-queens problem** of placing *n* queens on an *n*×*n* chessboard, where solutions exist for all natural numbers *n* with the exception of *1, 2 and 3.*
- The solution possibilities are discovered only up to *23 queen*.

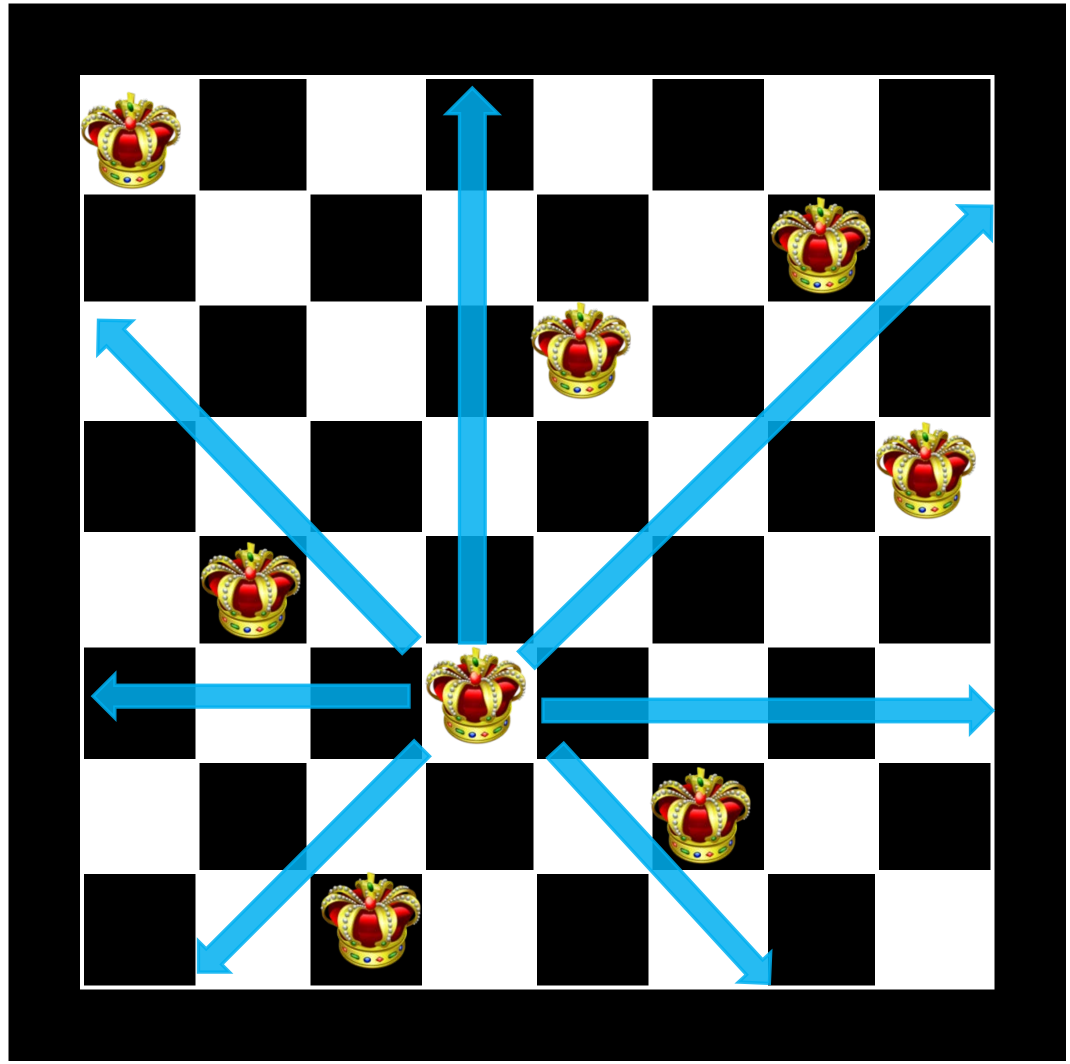
Formulation :

States: any arrangement of 0 to 8 queens on the board

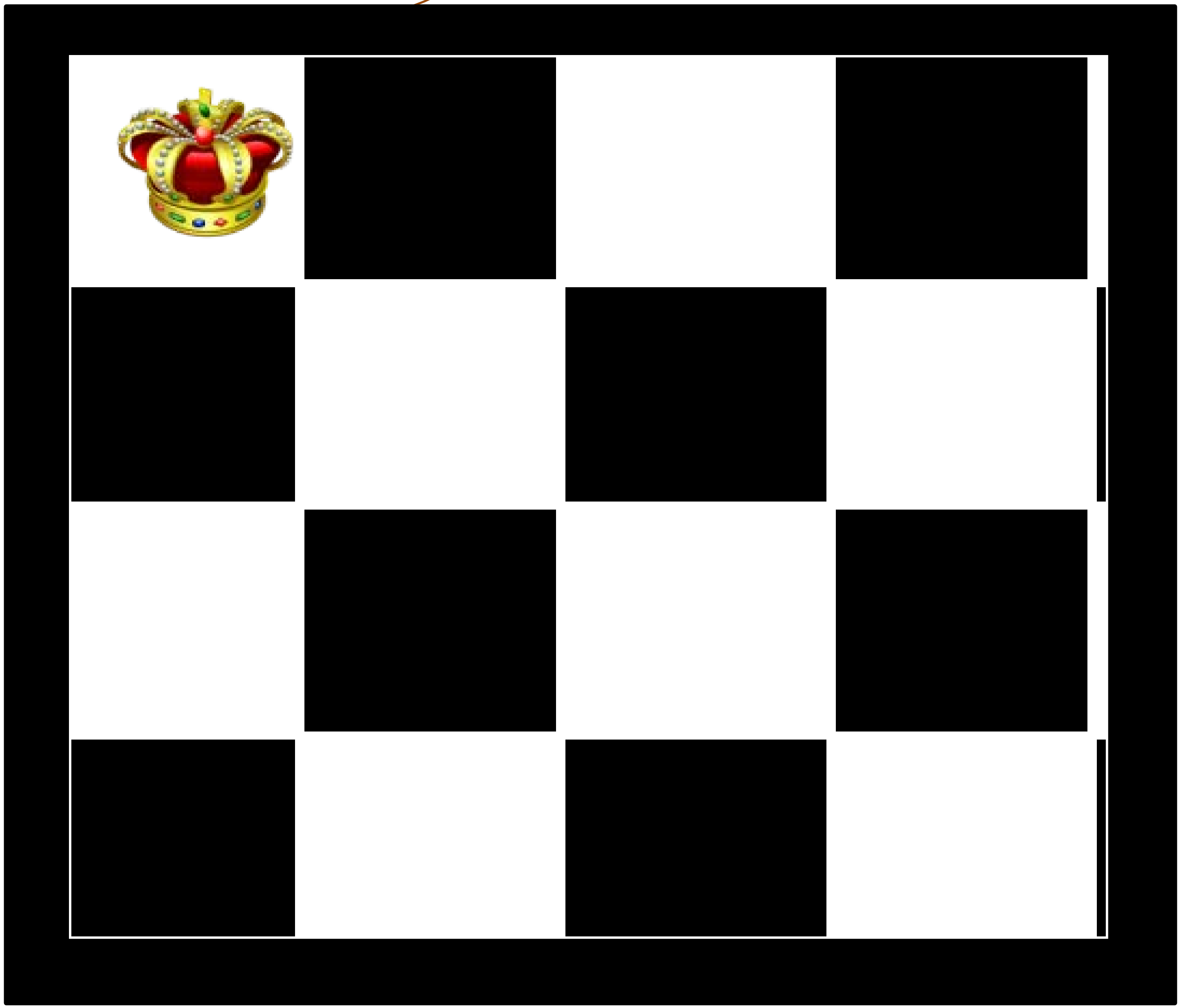
Initial state: 0 queens on the board

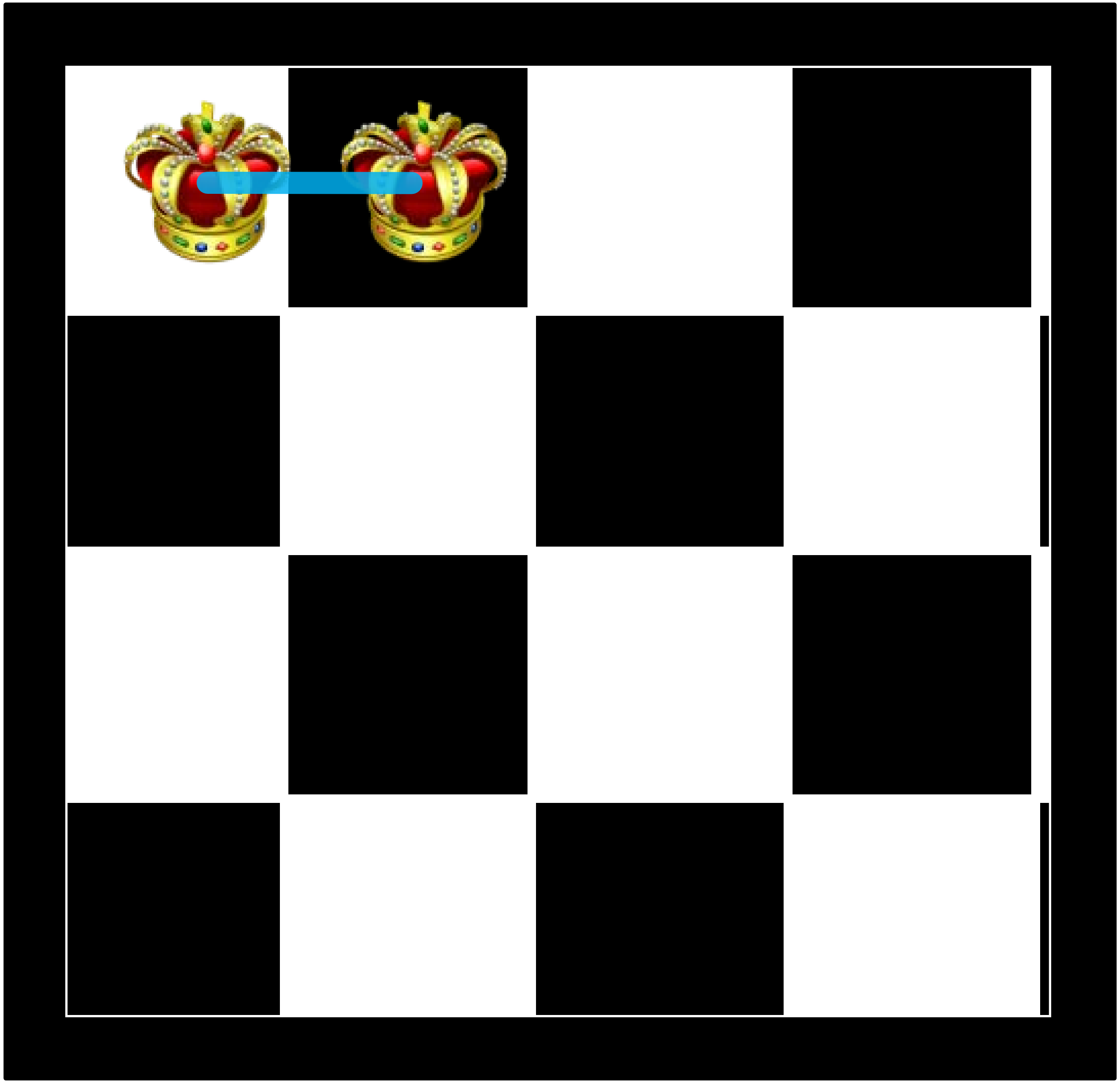
Successor function: add a queen in any square

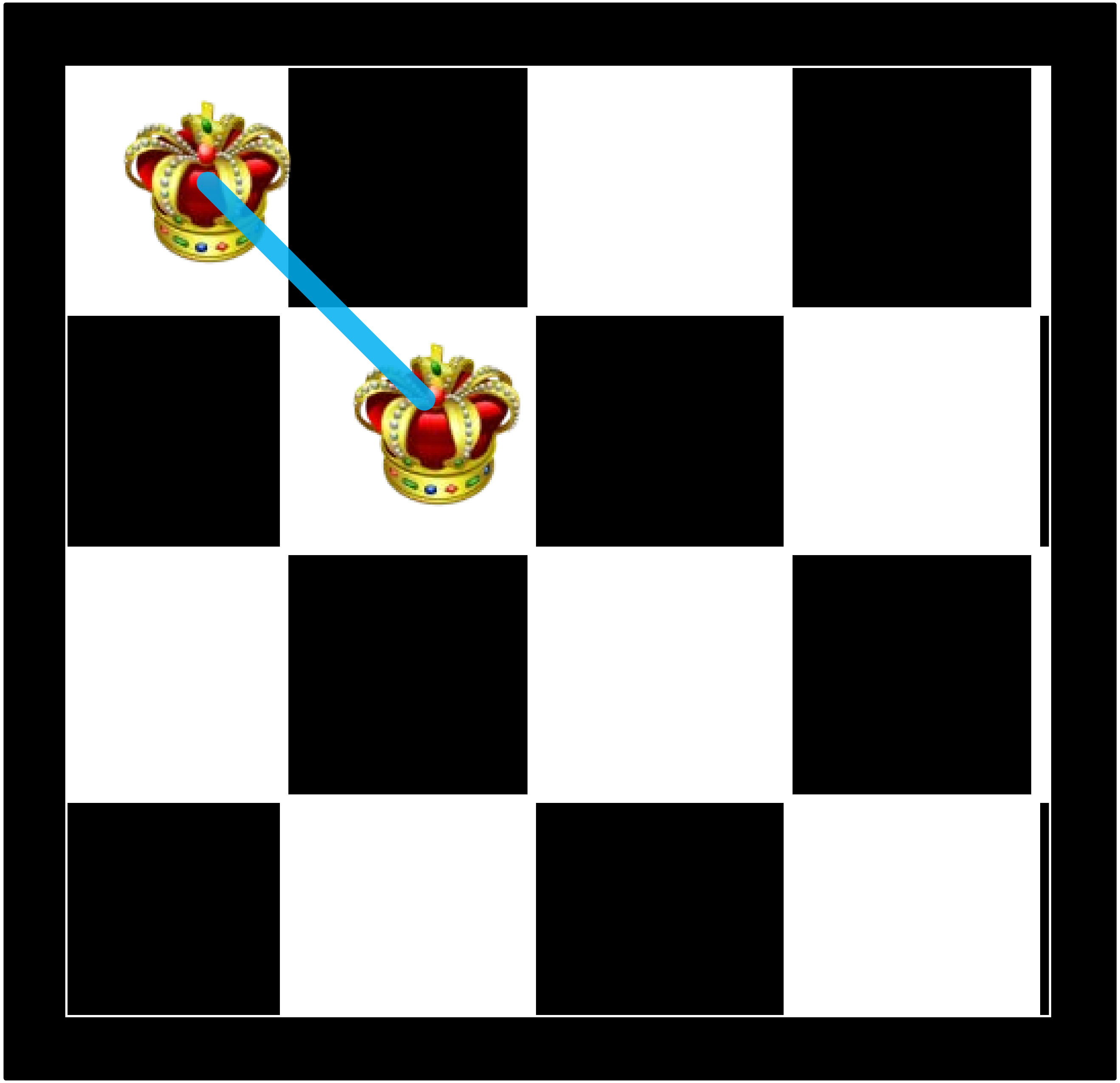
Goal test: 8 queens on the board, none attacked

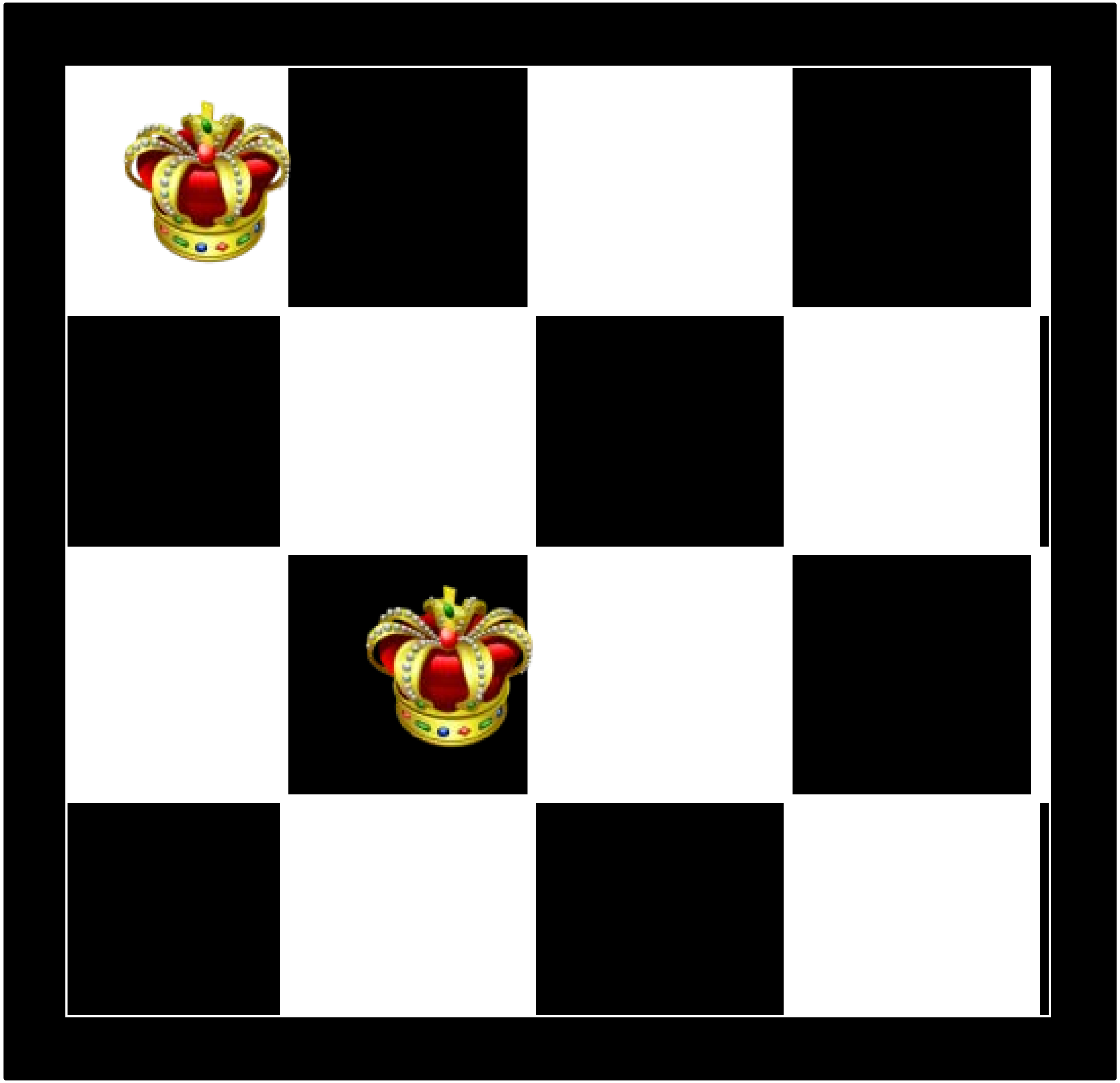


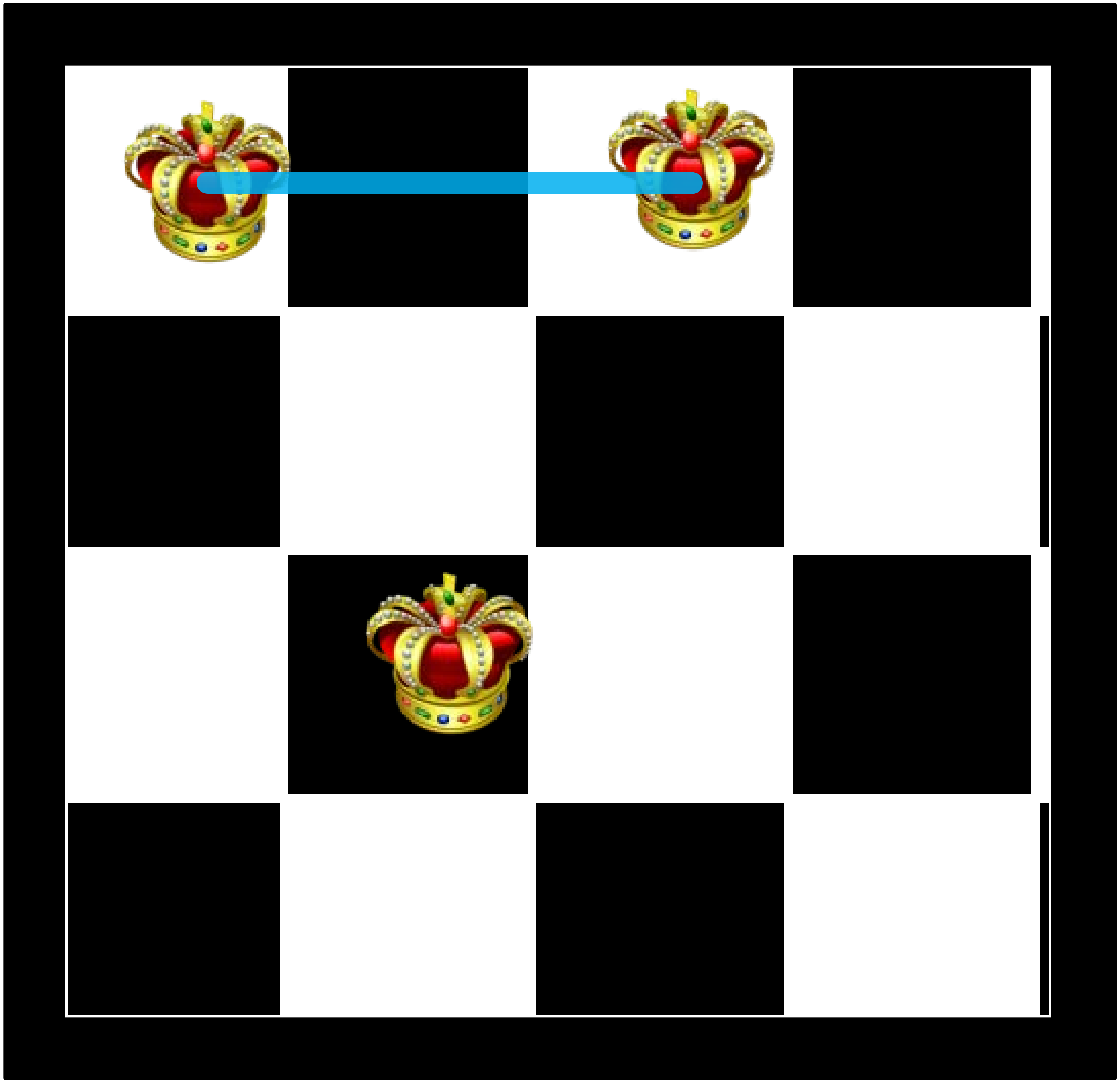
BACKTRACKING DEMO FOR 4 QUEENS

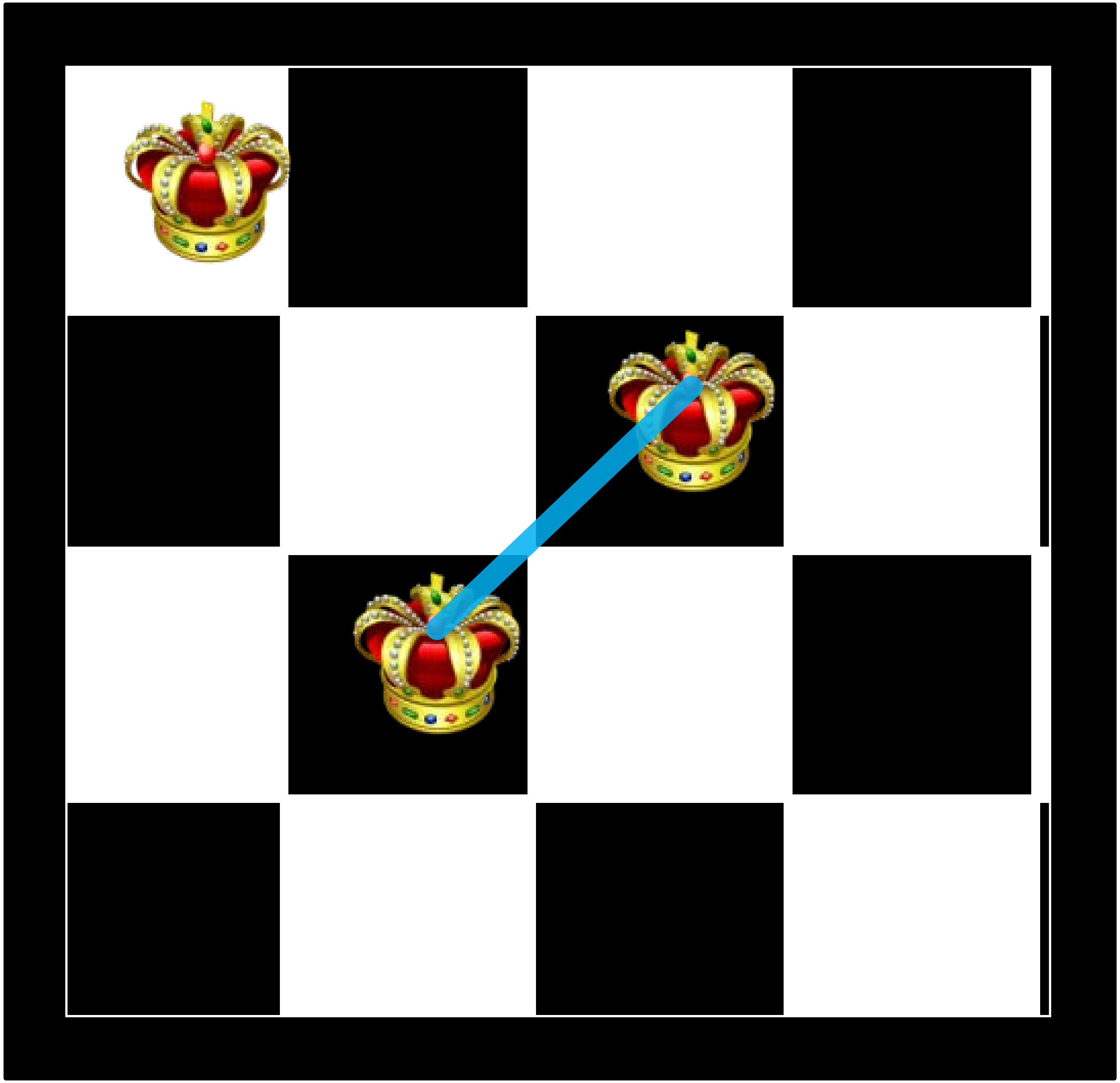


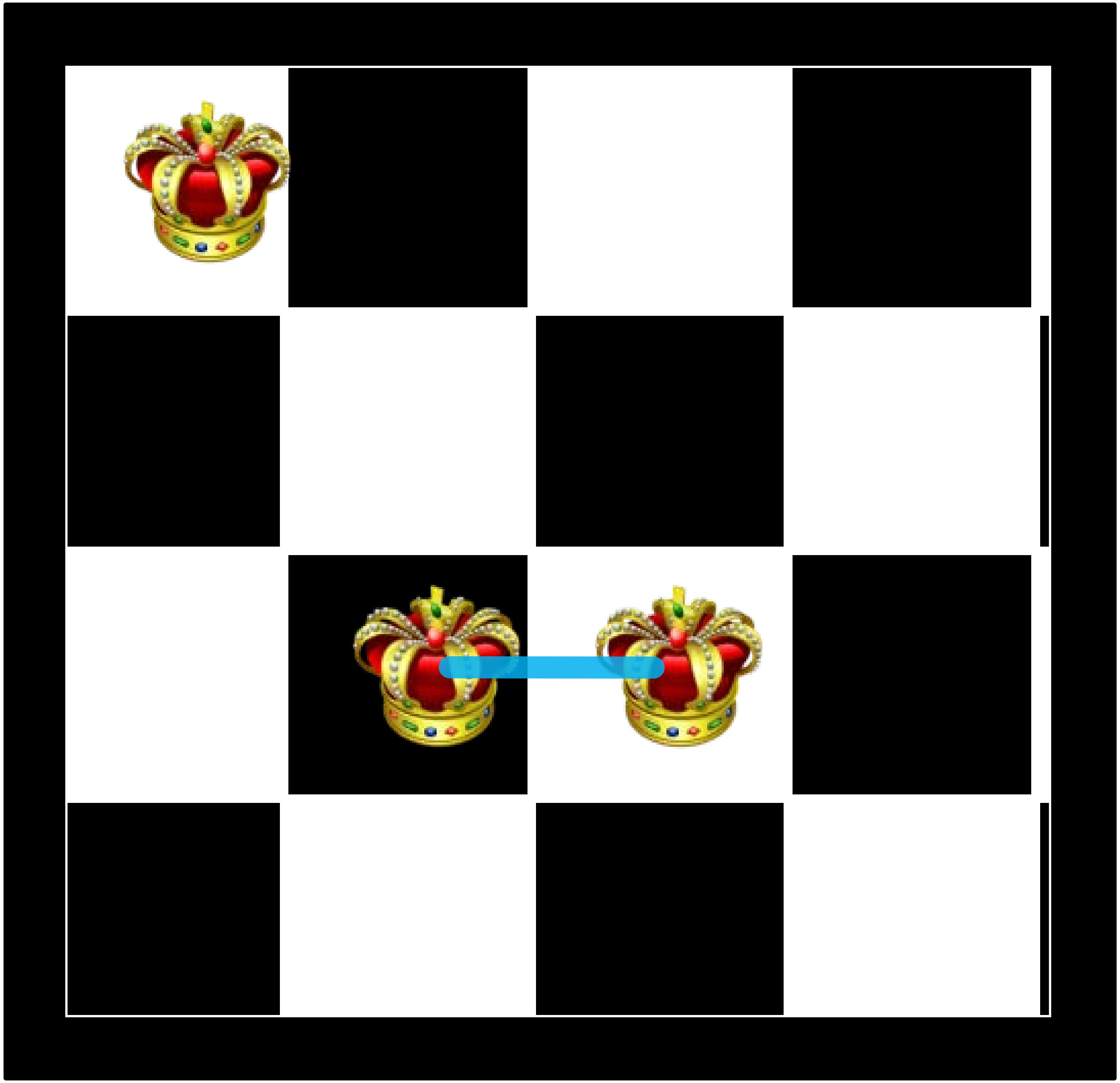


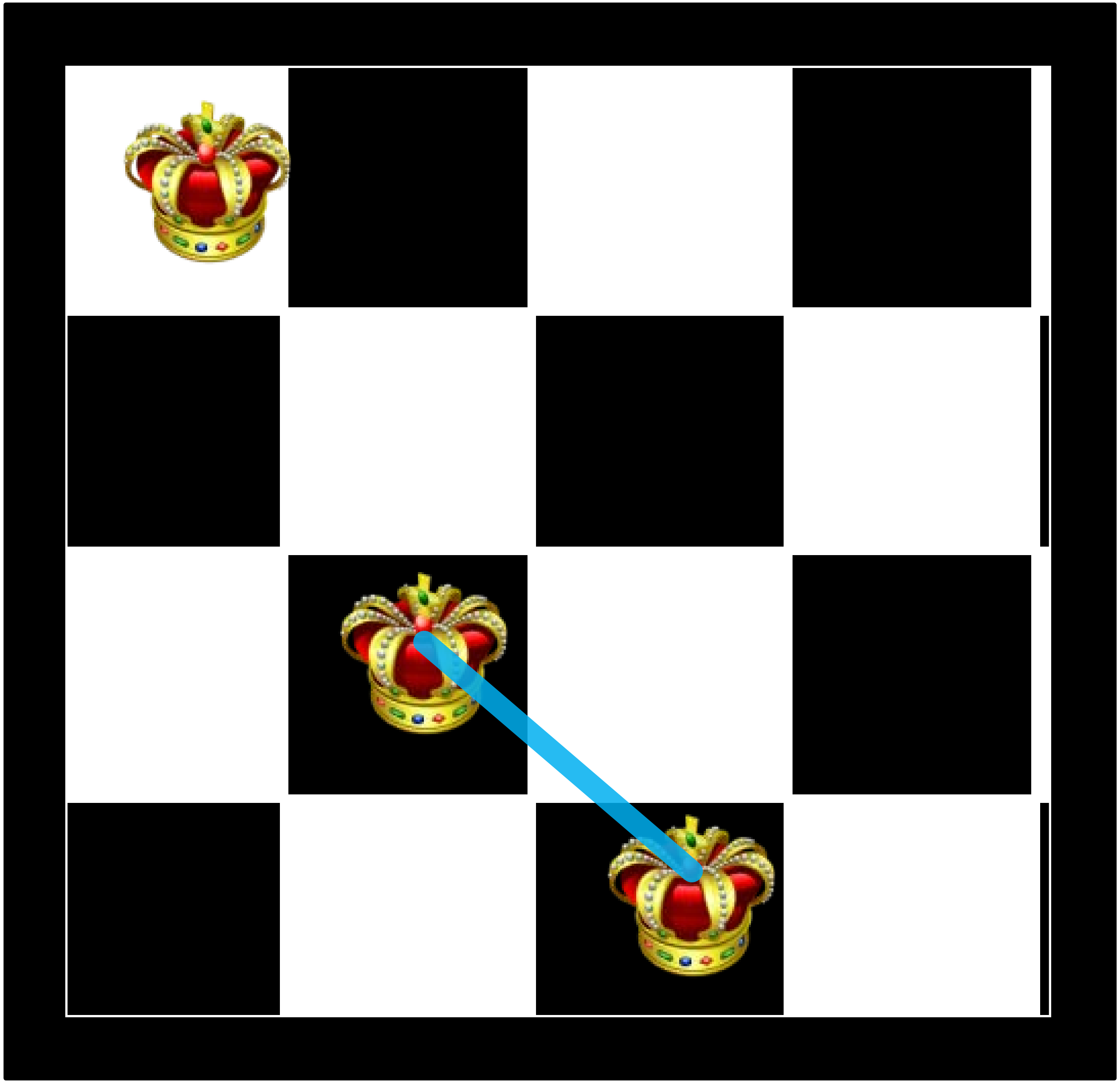


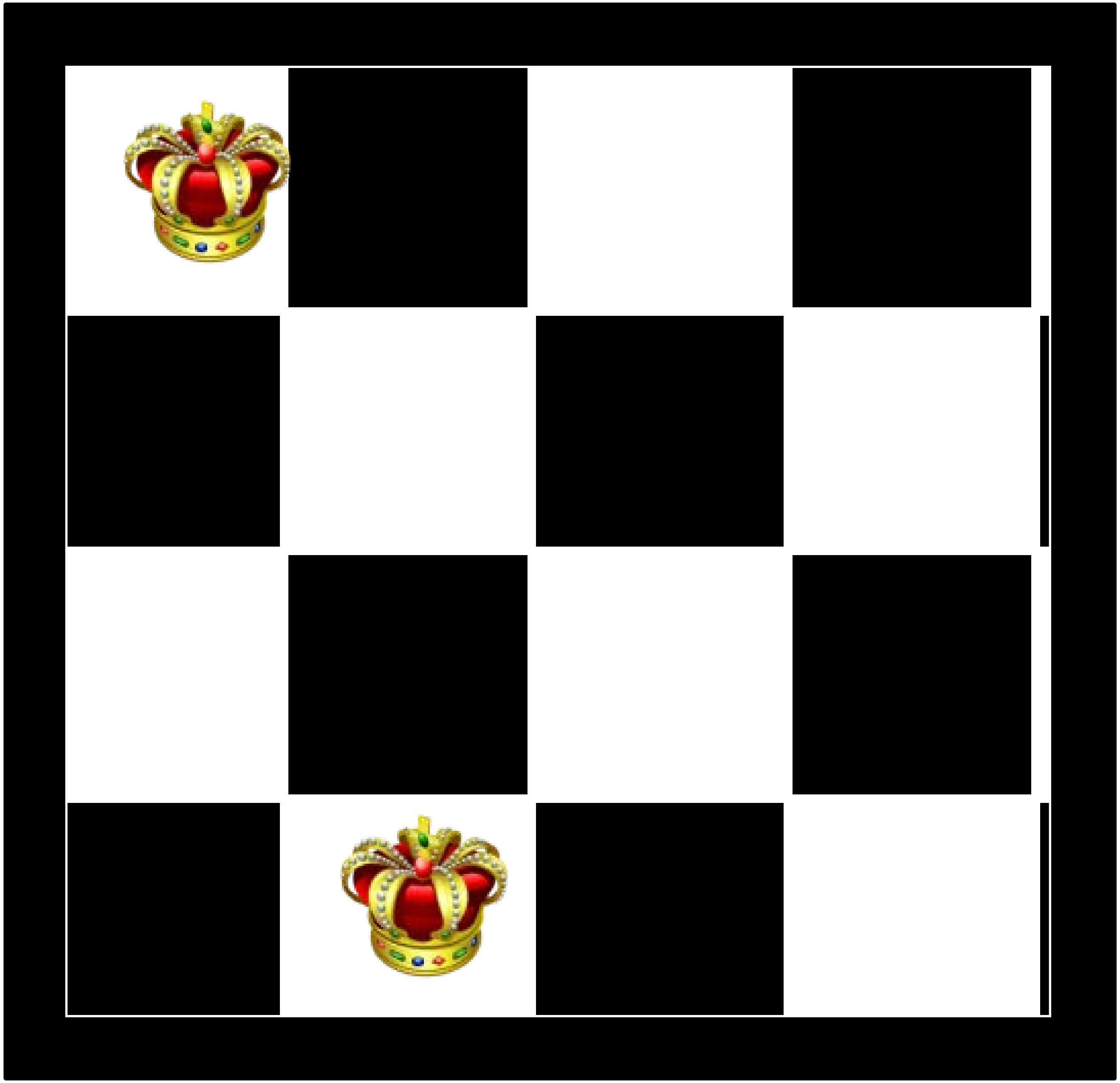


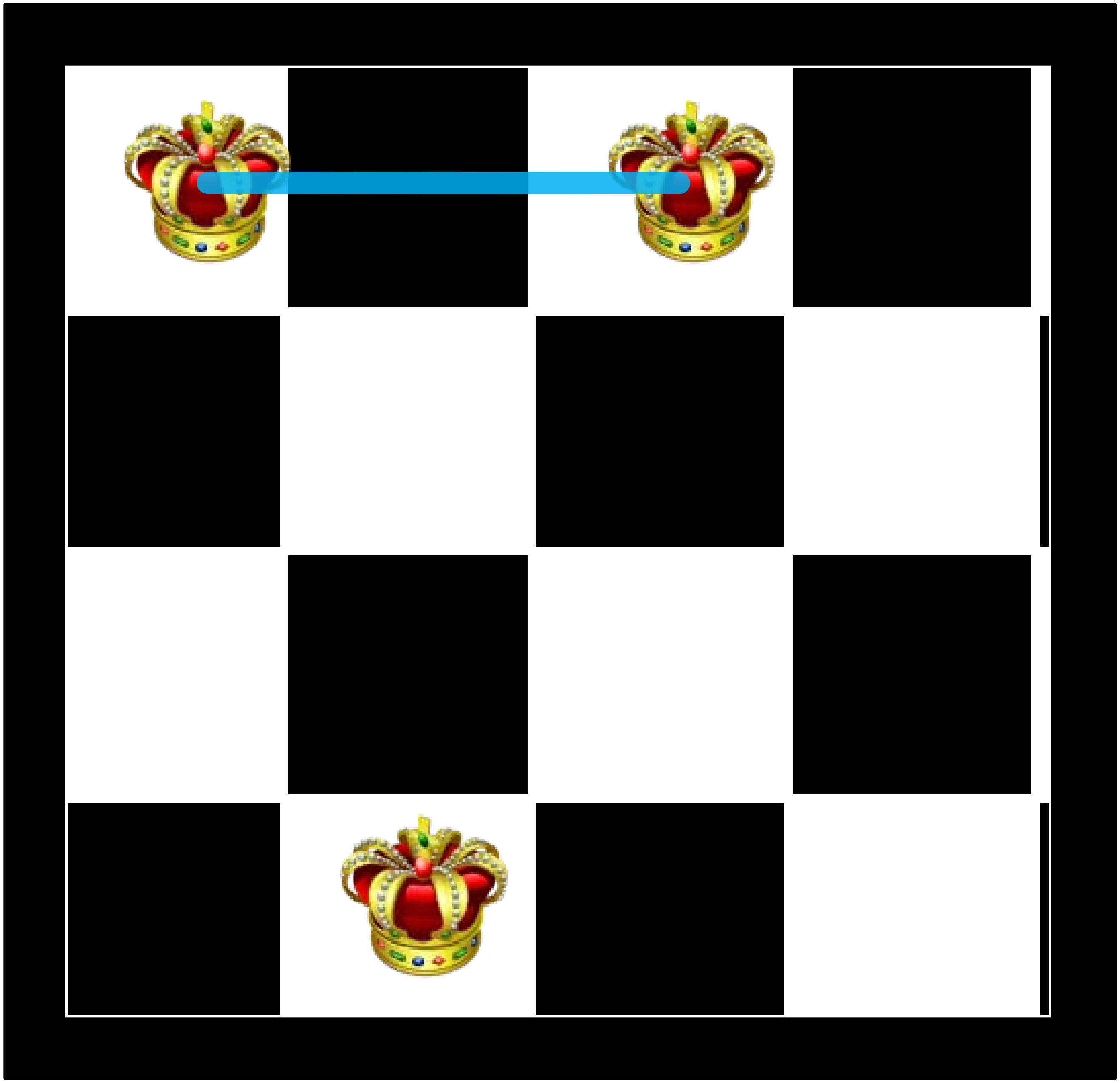


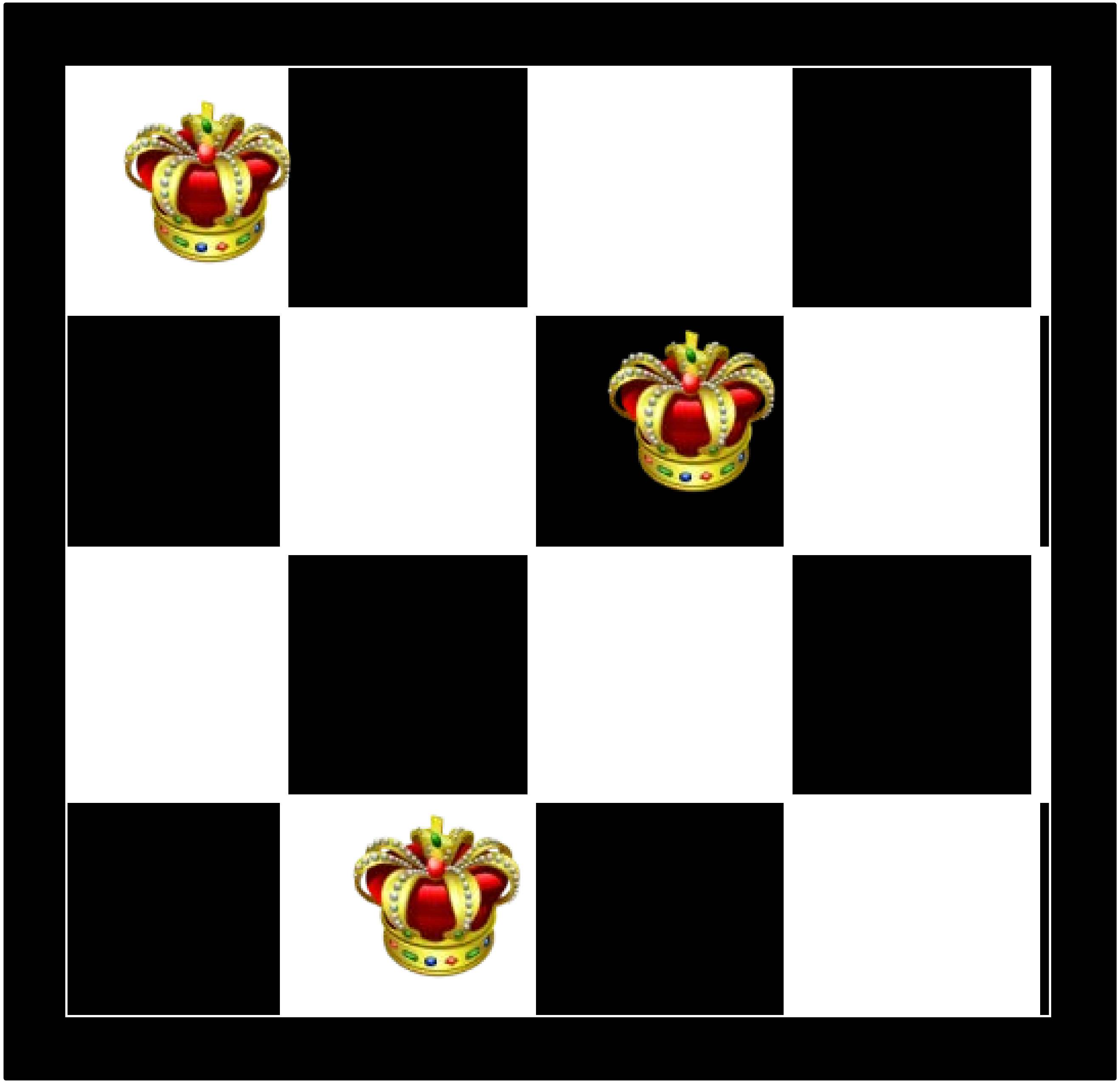


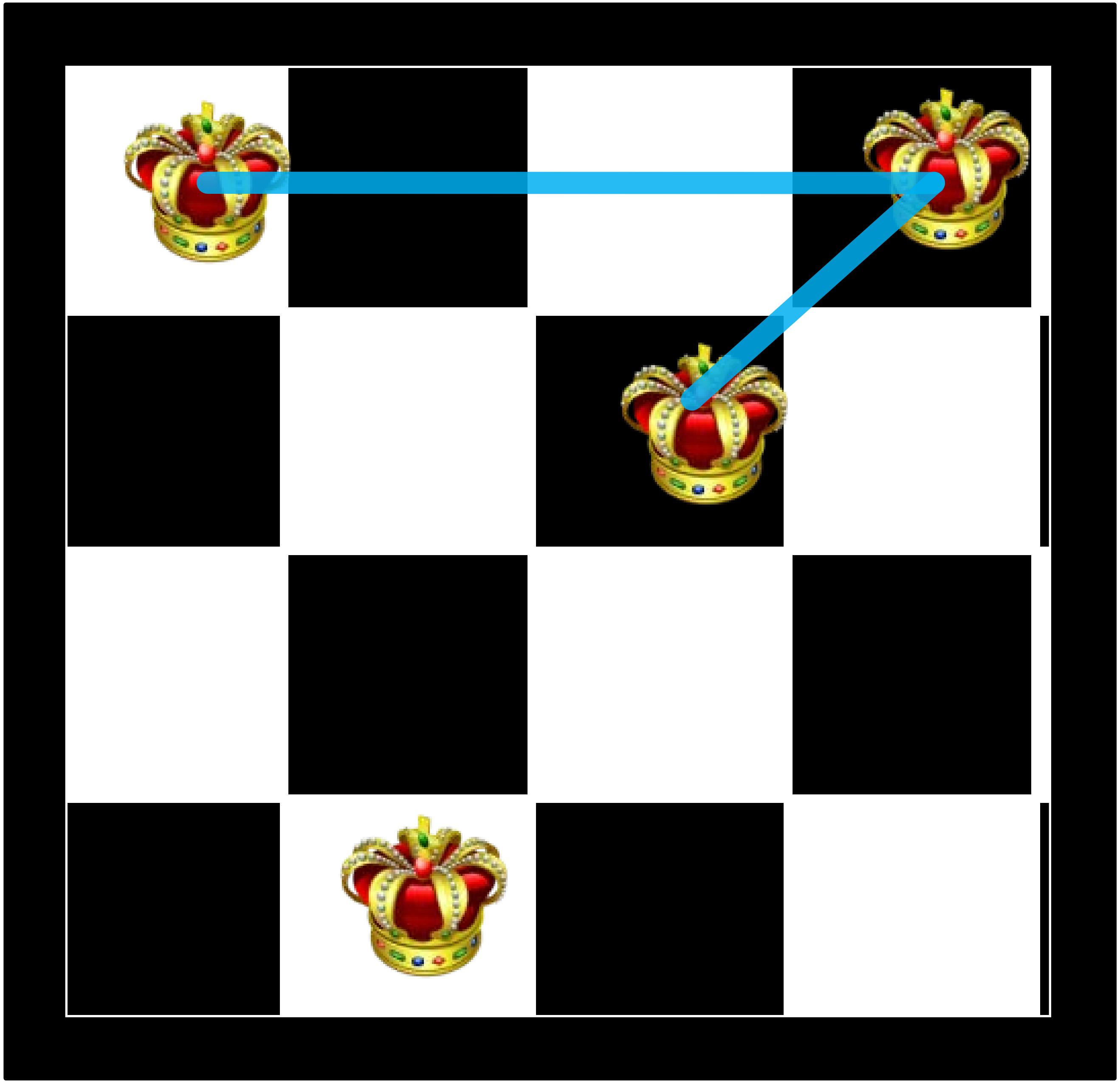


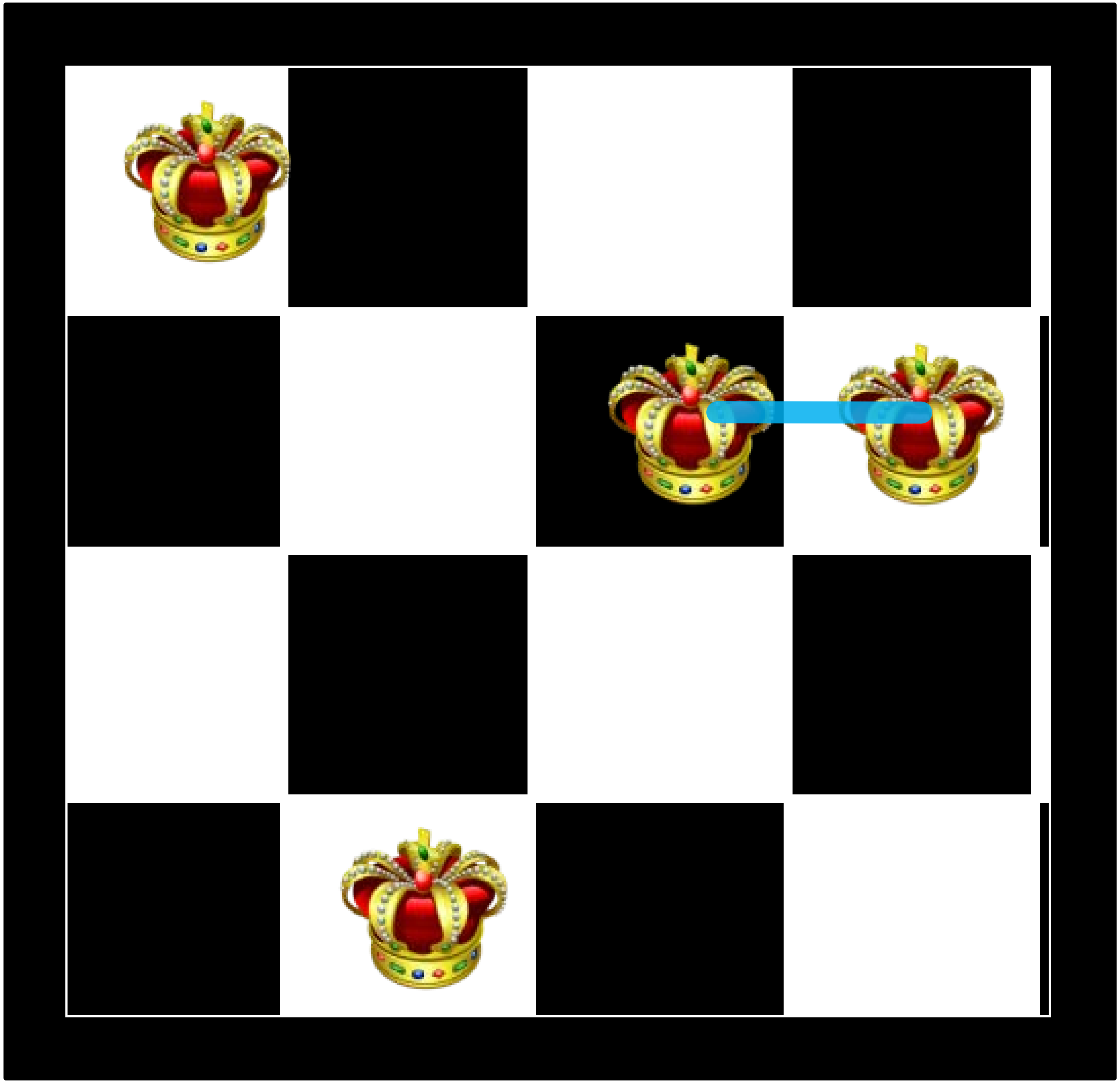


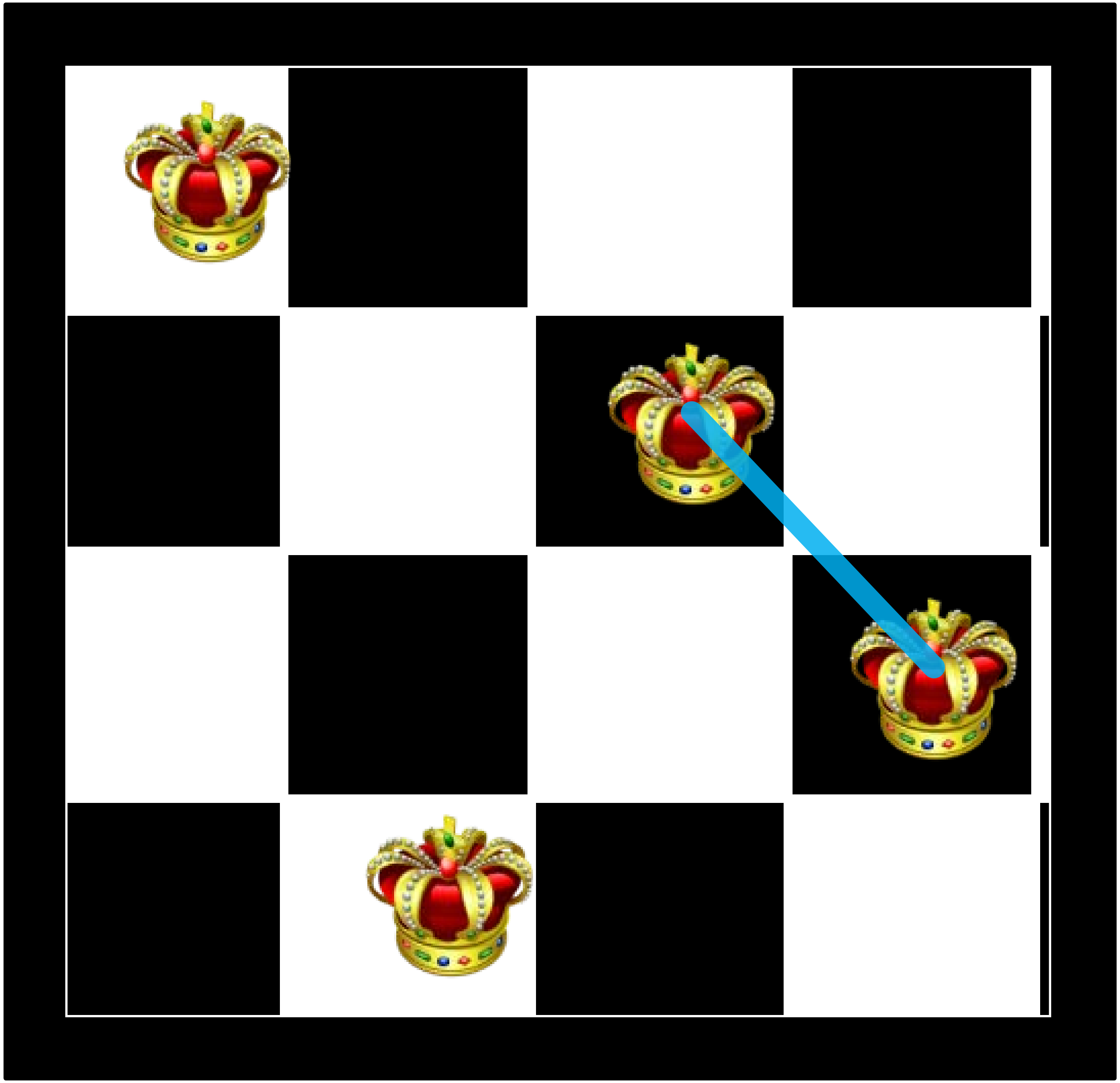


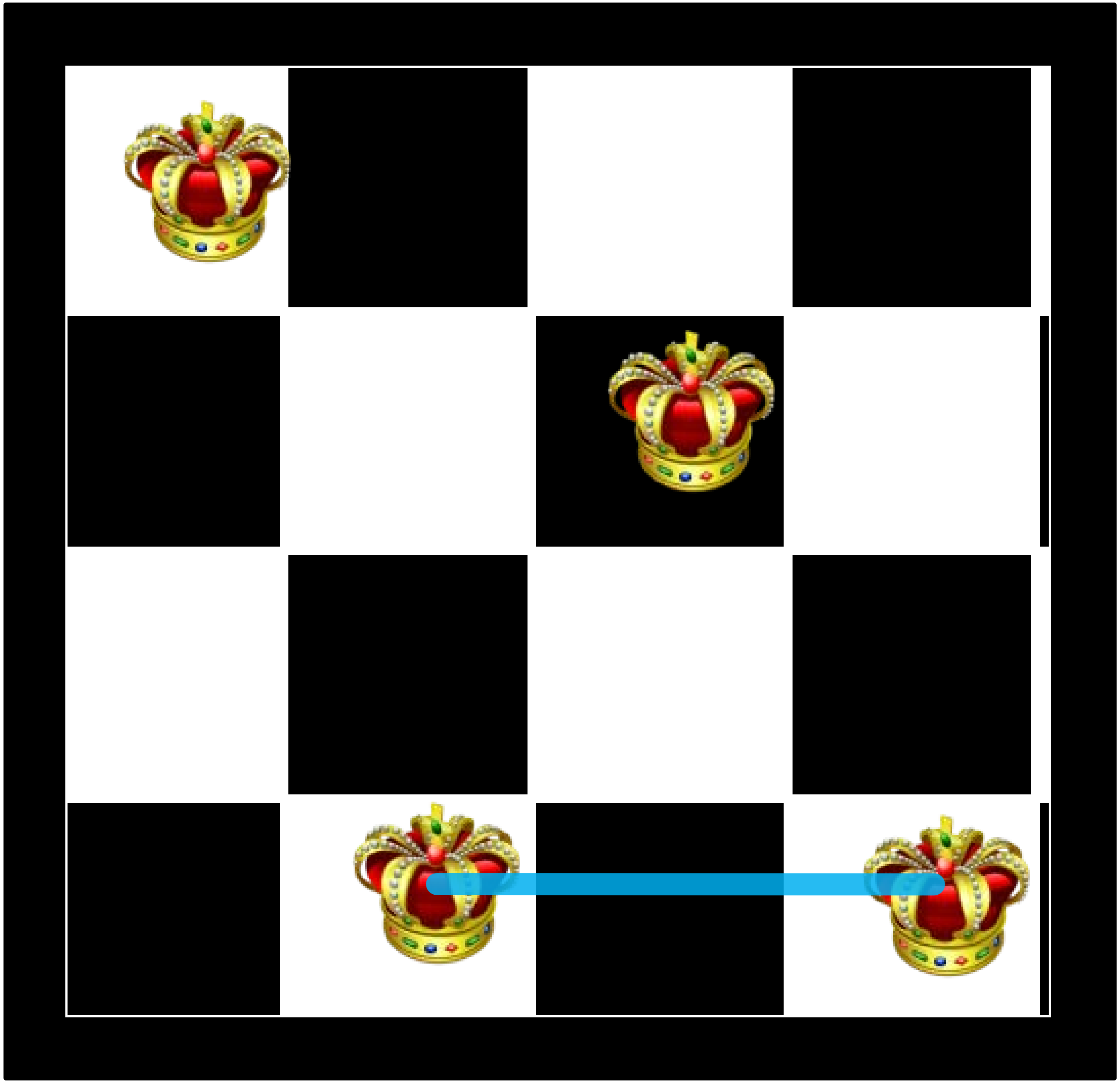


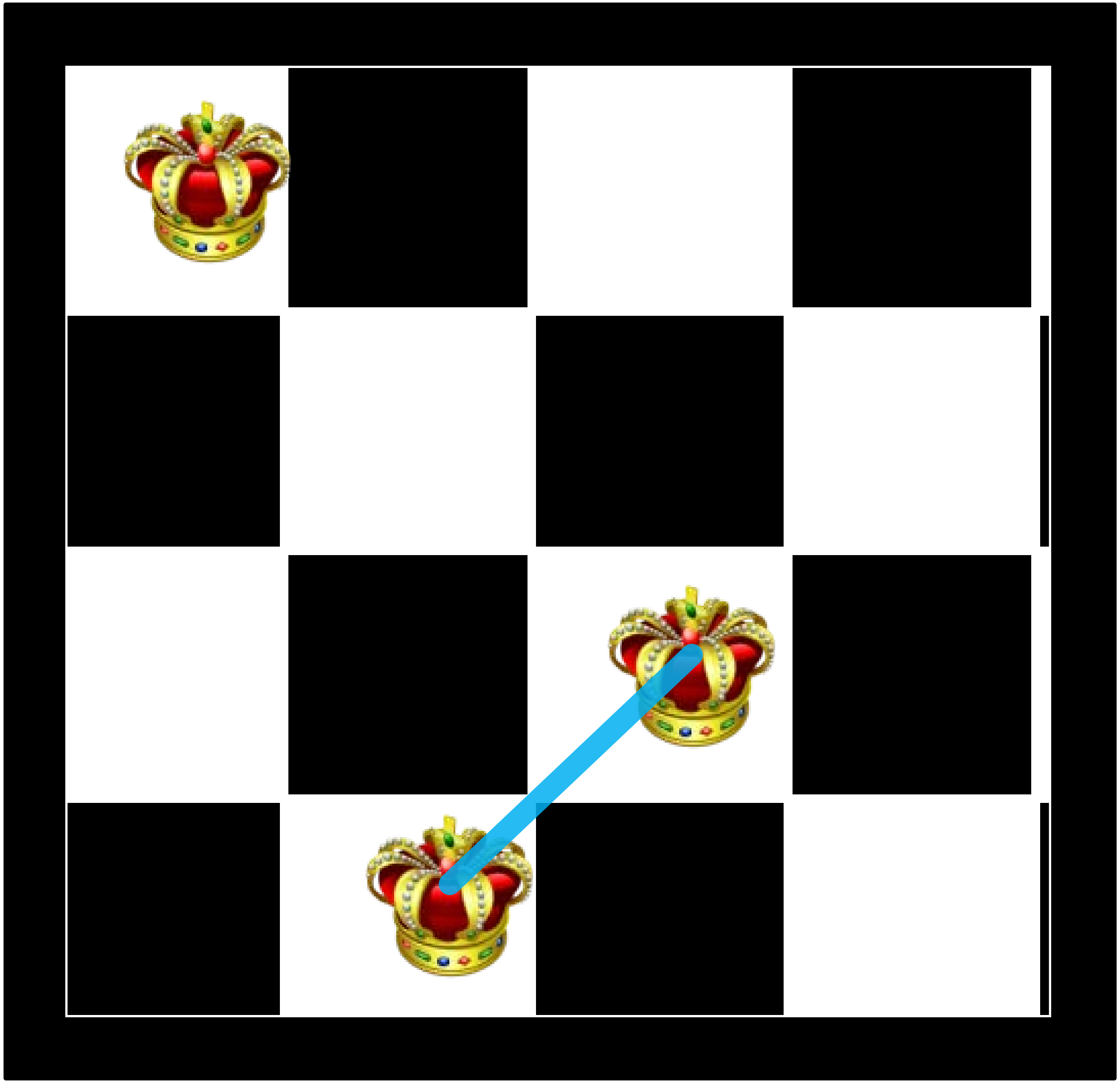


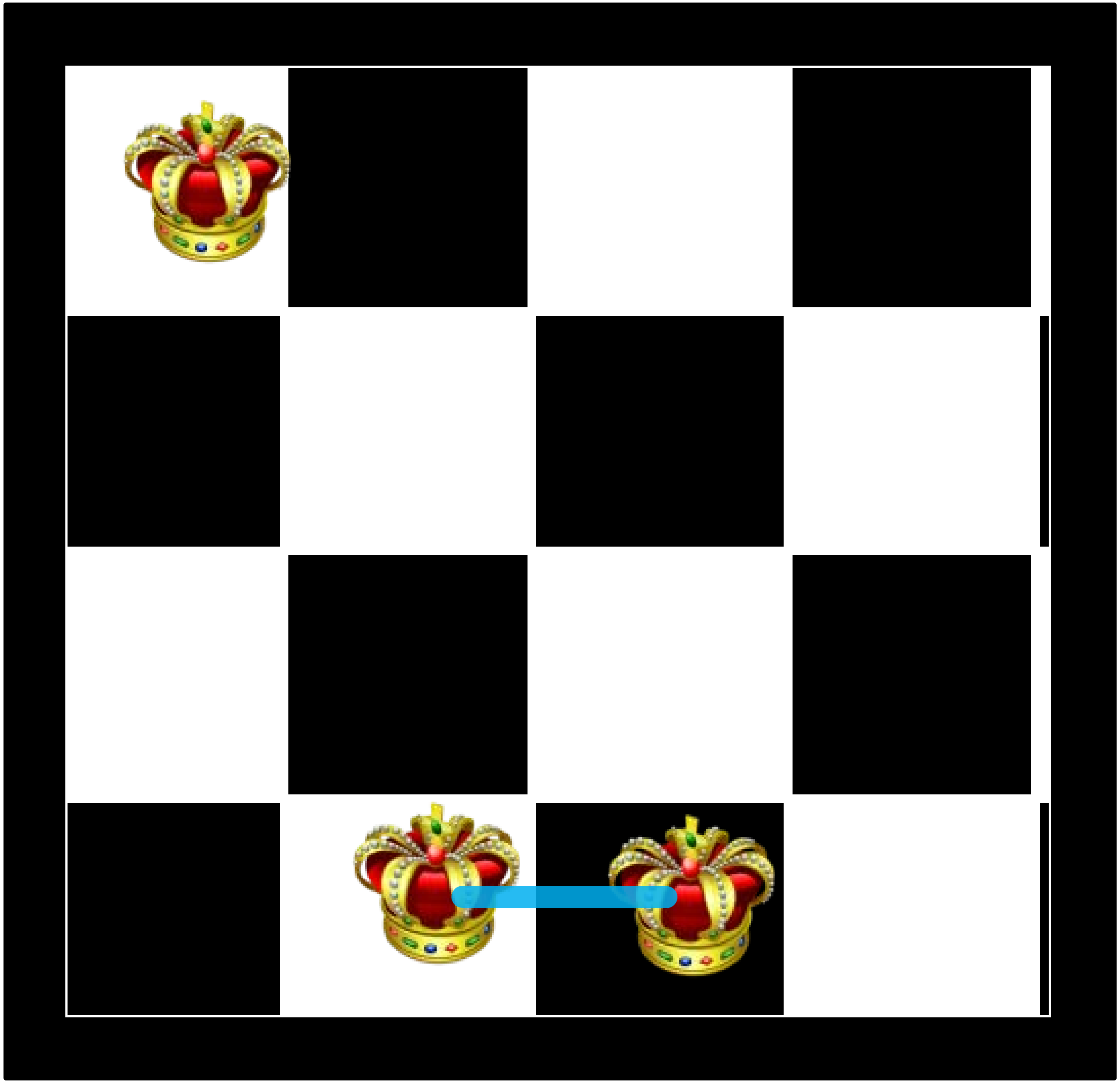


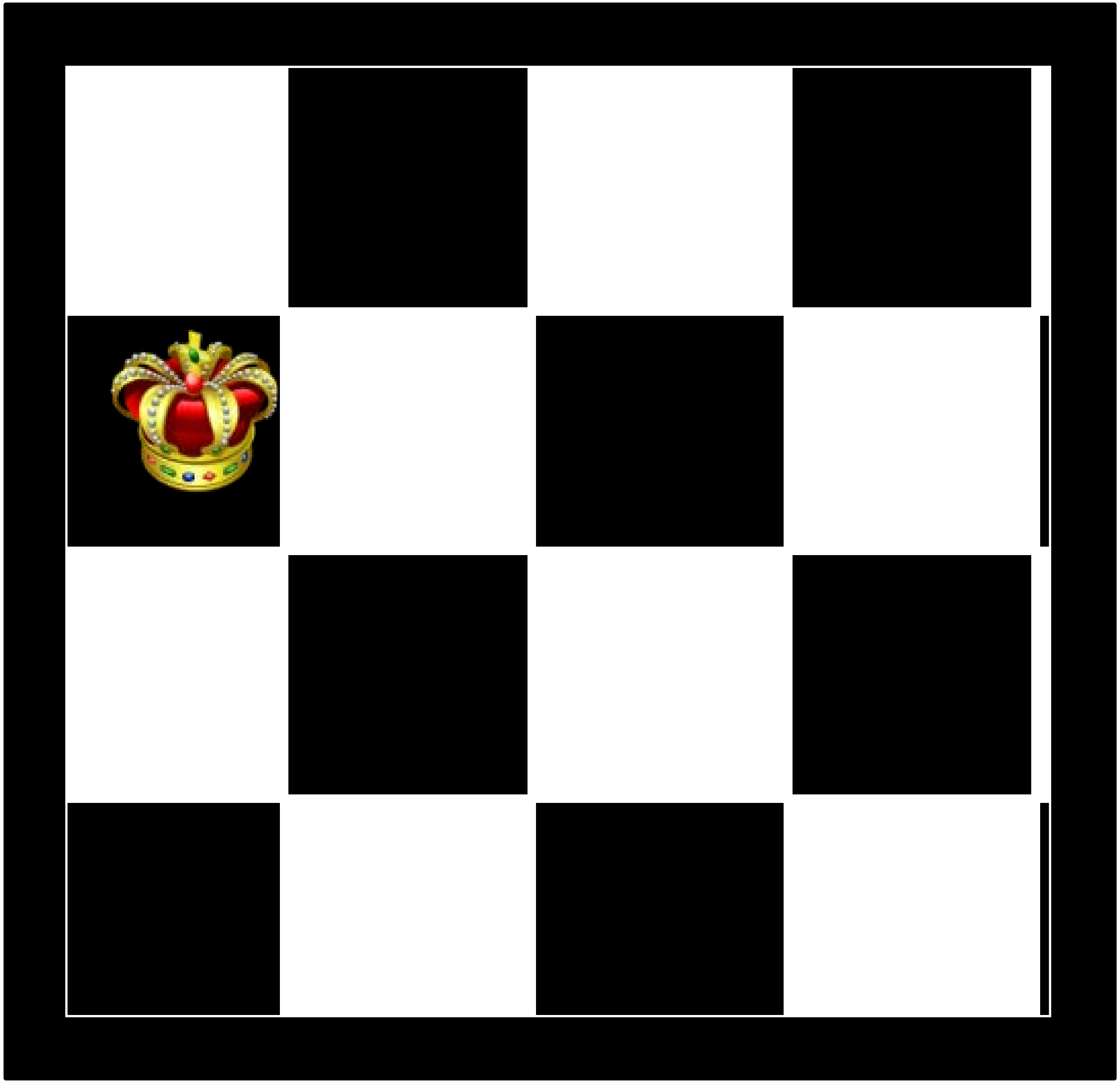


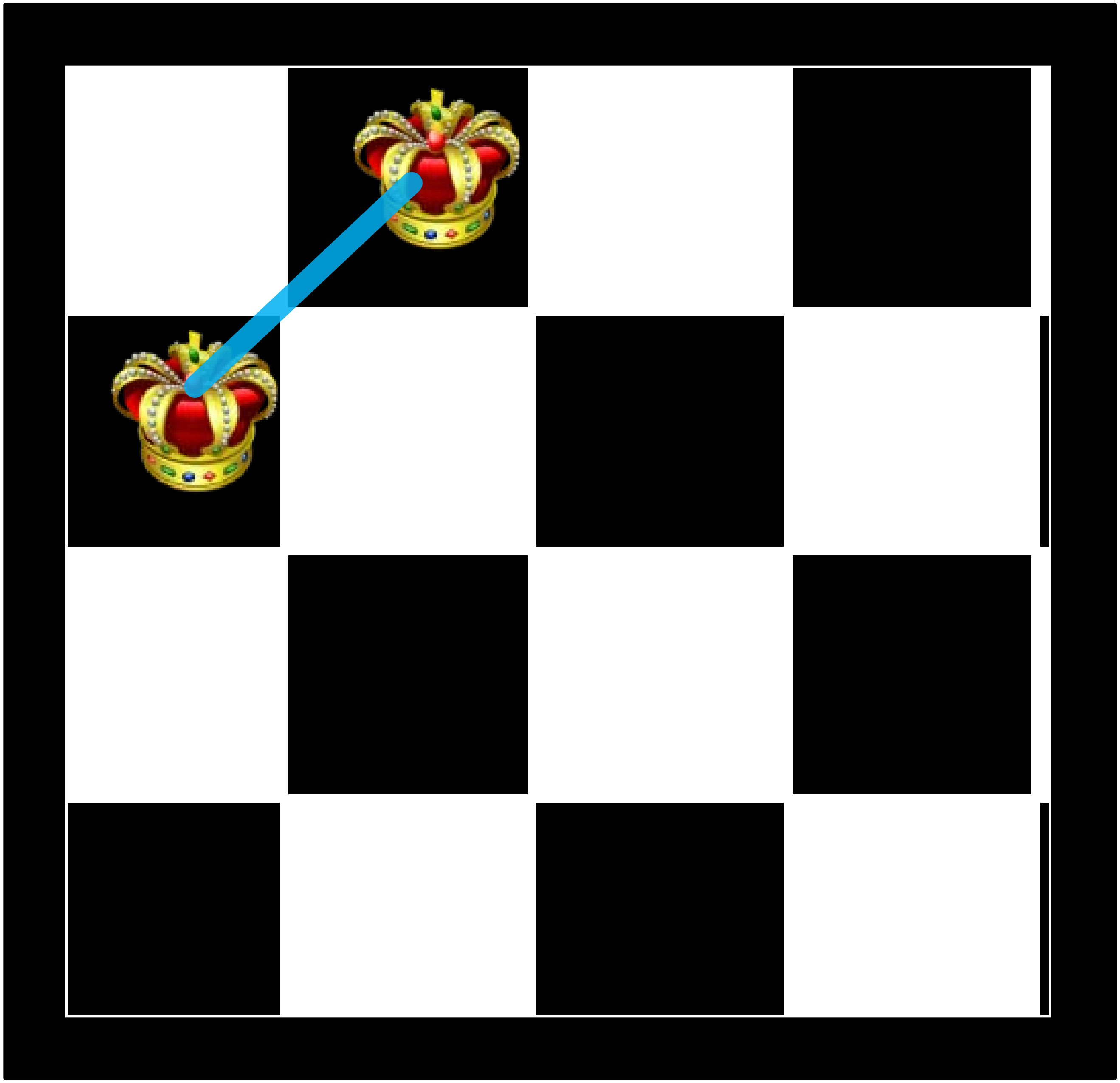


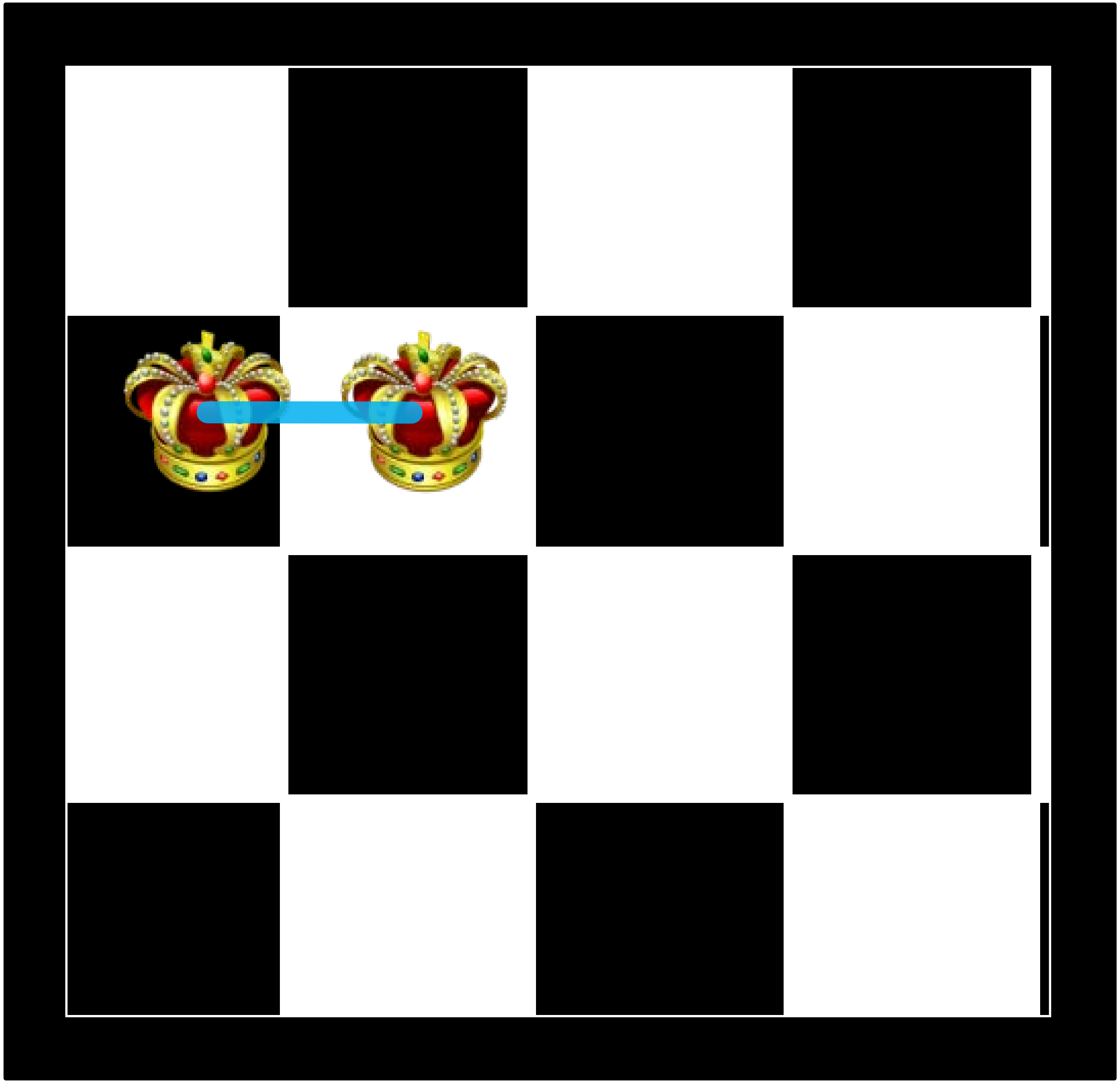


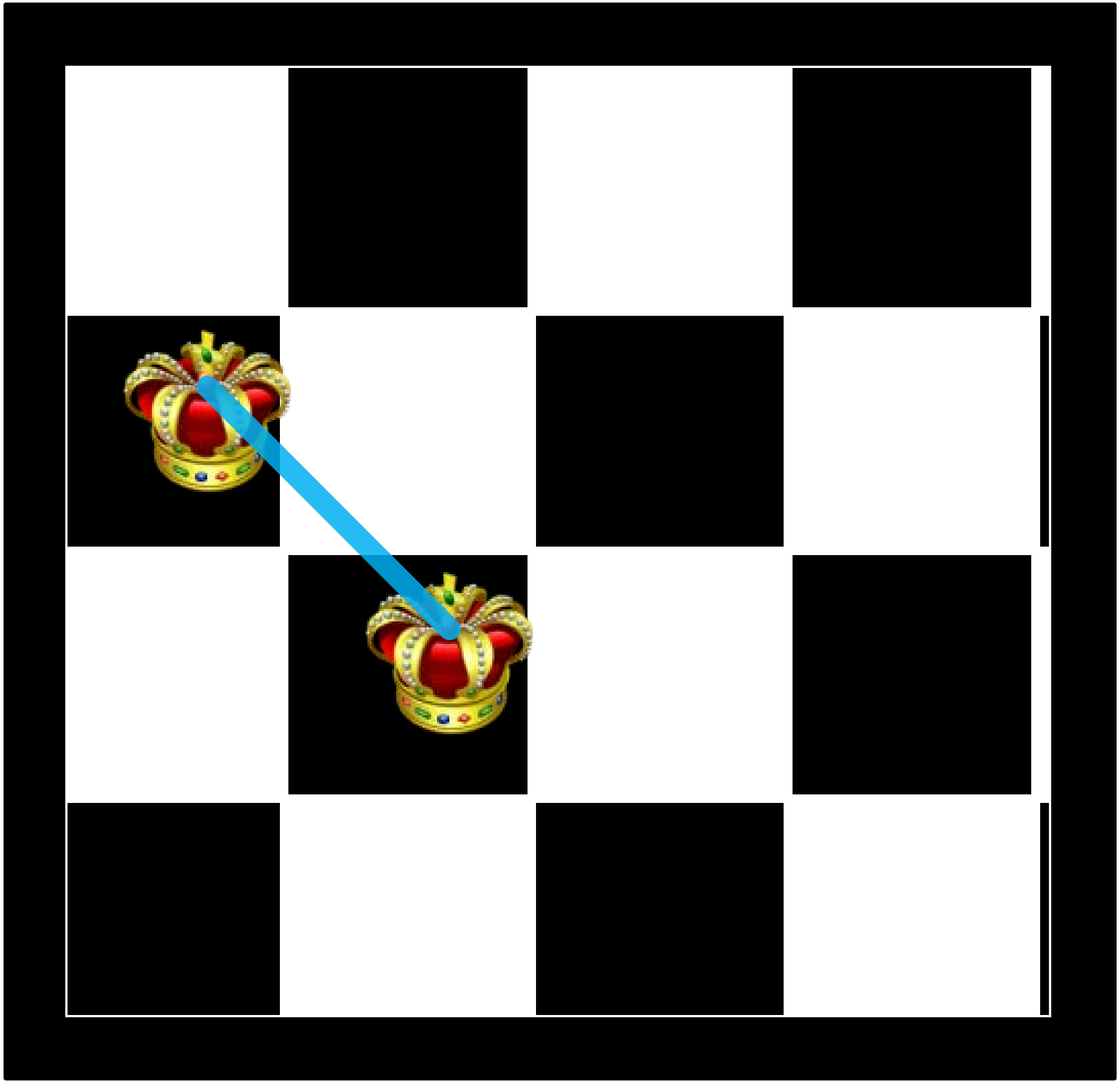


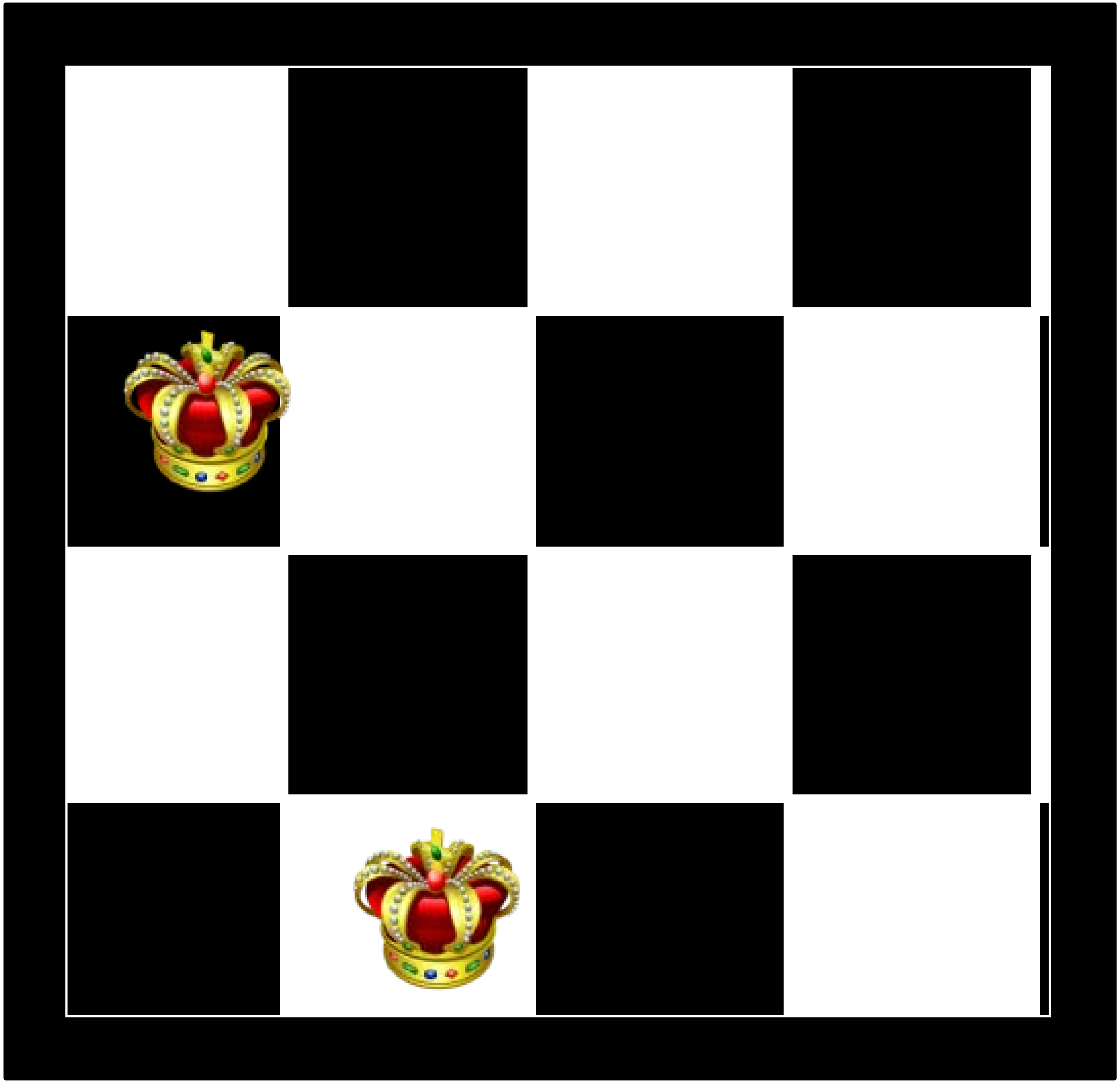


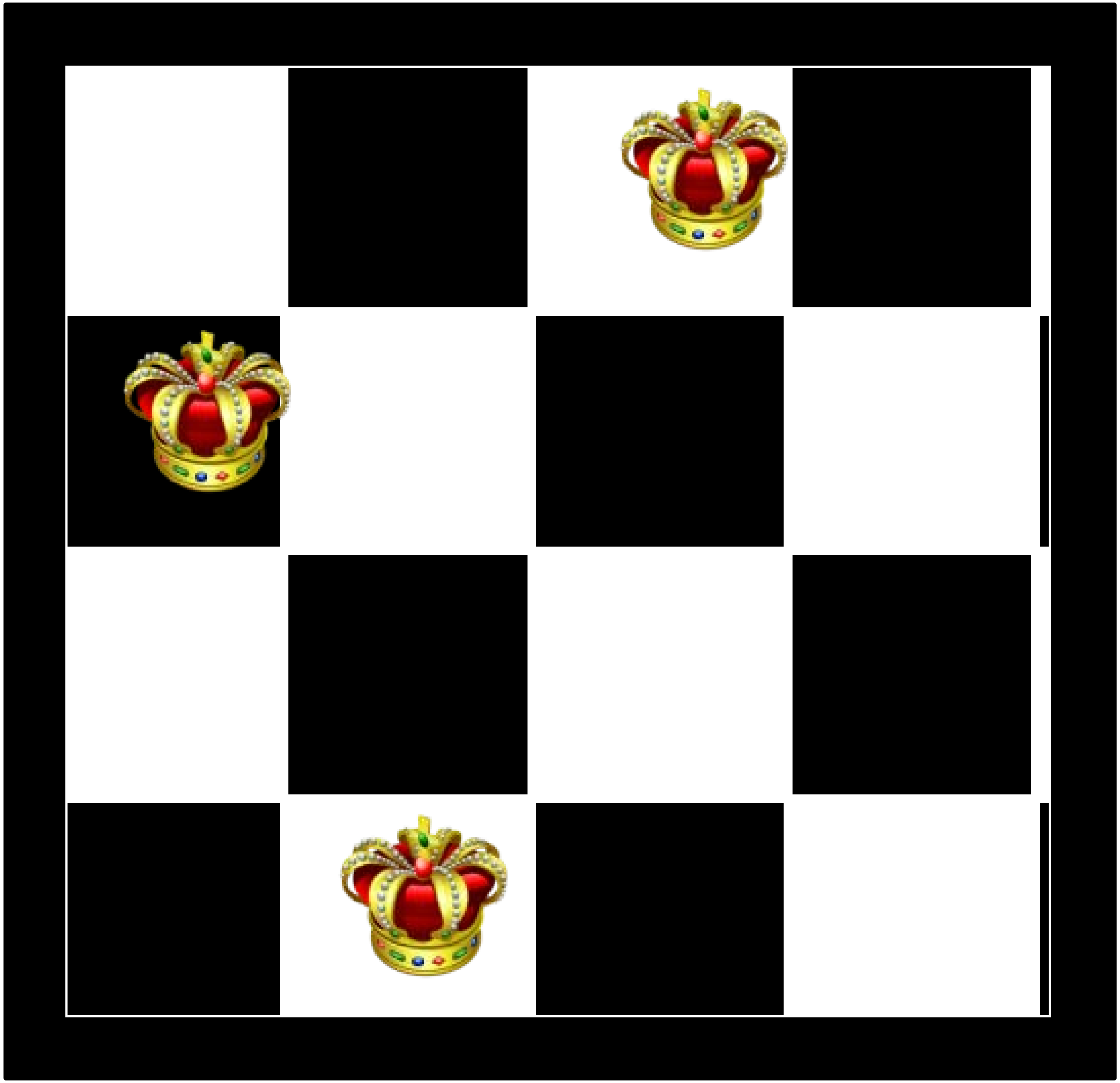


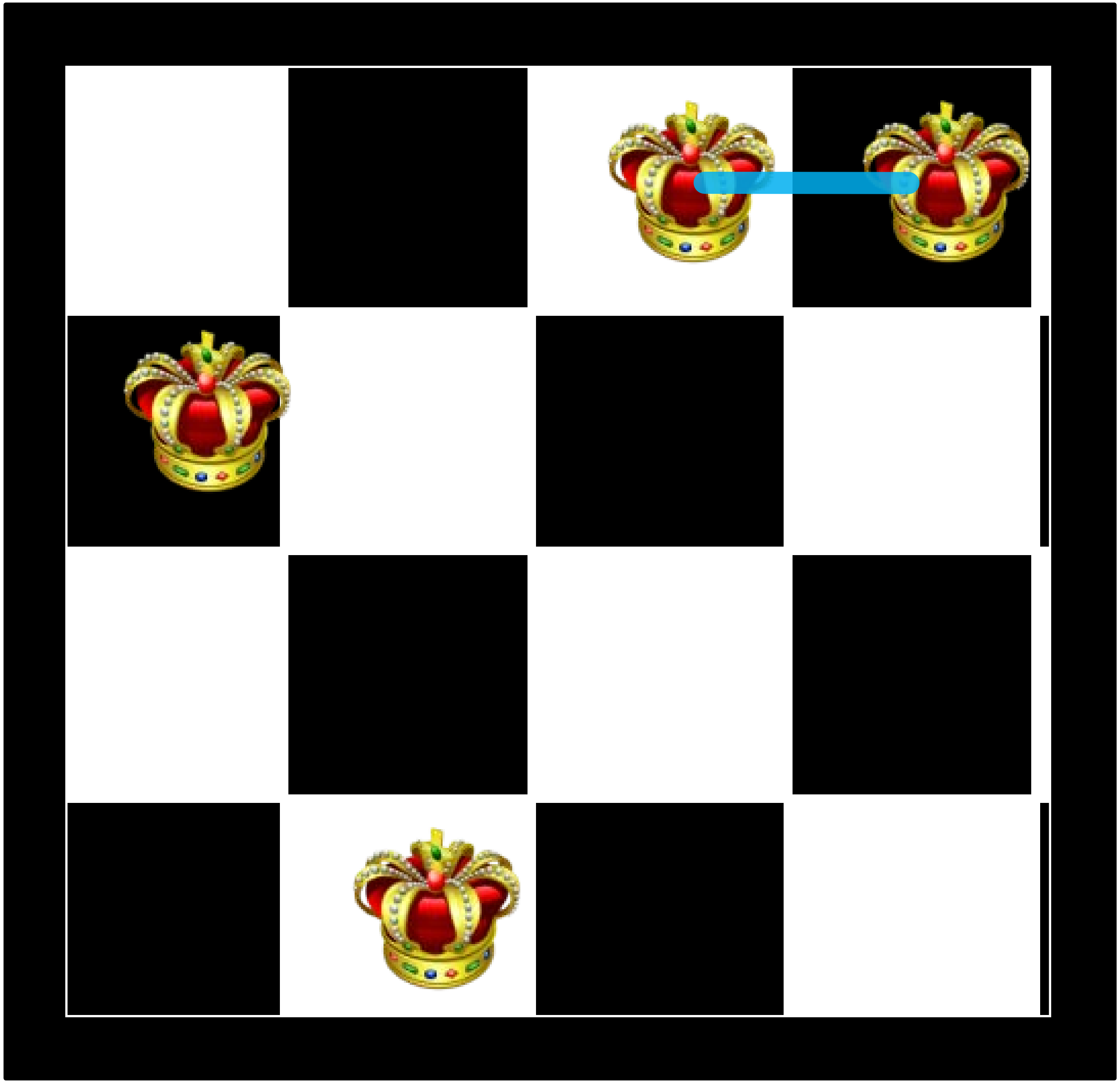


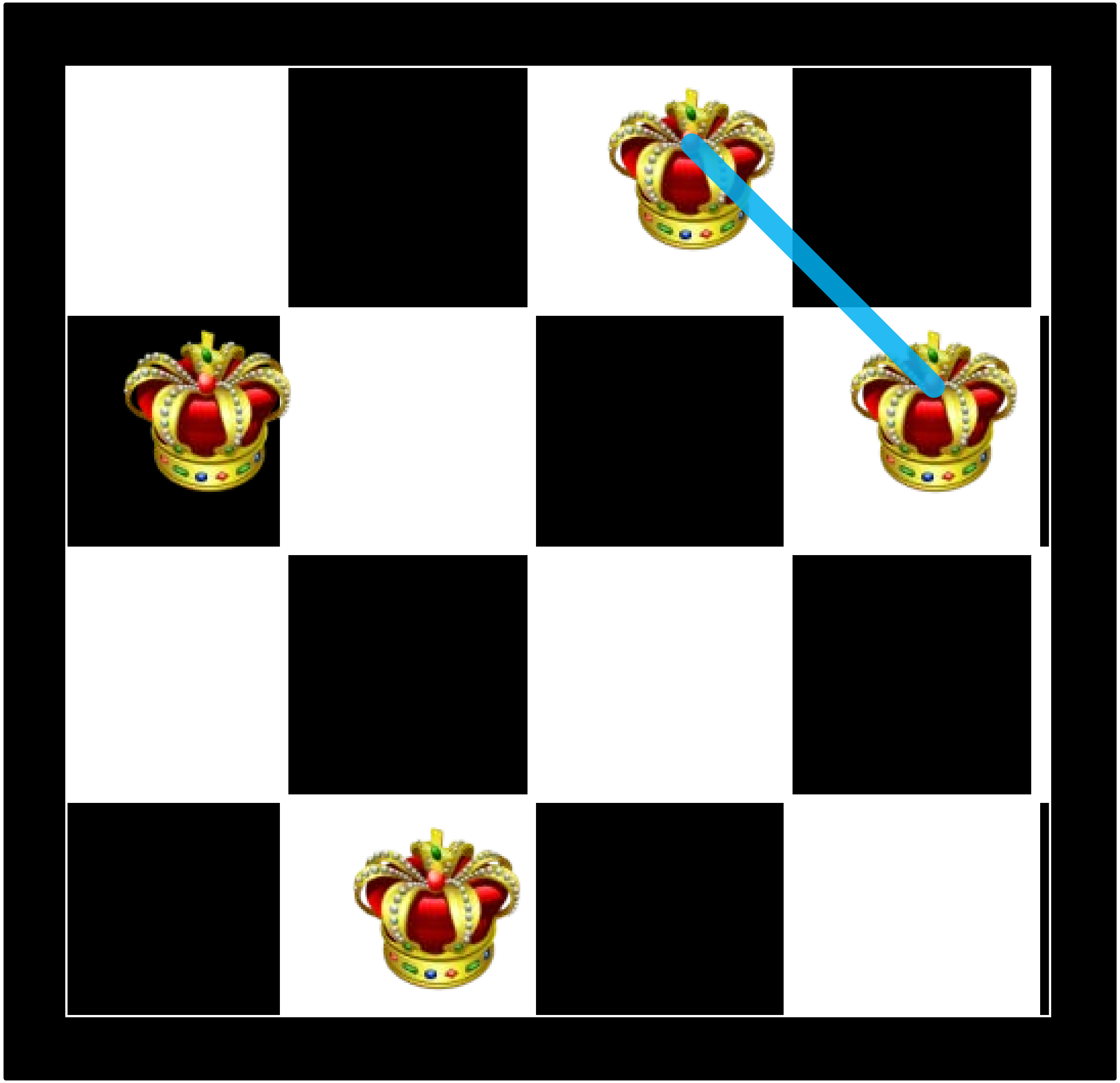













1 UNIQUE
SOLUTION

Example of a backtrack solution to the 4 –queens problem

1			

(a)

1			
.	.	2	

(b)

1			
		2	
.	.	.	.

(c)

1			
			2
.	3		

(d)

1			
			2
	3		
.	.	.	.

(e)

	1		

(f)

	1		
.	.	.	2

(g)

	1		
			2
3			
.	.	4	

(h)

Solutions of 4queen problems

	1		
			2
3			
		4	

		1	
2			
			3
	4		

```
Algorithm NQueens(k, n)
// Using backtracking, this procedure prints all
// possible placements of n queens on an n × n
// chessboard so that they are nonattacking.
{
    for i := 1 to n do
    {
        if Place(k, i) then
        {
            x[k] := i;
            if (k = n) then write (x[1 : n]);
            else NQueens(k + 1, n);
        }
    }
}
```

```
Algorithm Place(k, i)
// Returns true if a queen can be placed in kth row and
// ith column. Otherwise it returns false. x[ ] is a
// global array whose first (k - 1) values have been set.
// Abs(r) returns the absolute value of r.
{
    for j := 1 to k - 1 do
        if ((x[j] = i) // Two in the same column
             or (Abs(x[j] - i) = Abs(j - k)))
                // or in the same diagonal
            then return false;
    return true;
}
```

- To check whether they are on the same diagonal, let chessboard be represented by an array a[1..n][1..n].
- Every element with same diagonal that runs from upper left to lower right has same “row-column” value. E.g., consider the element a[4][2]. Elements a[3][1], a[5][3], a[6][4], a[7][5] and a[8][6] have row-column value 2.
- every element from same diagonal that goes from upper right to lower left has same “row+column” value. The elements a[1][5], a[2][4], a[3][3], a[5][1] have same “row+column” value as that of element a[4][2] which is 6.

- Hence two queens placed at (i,j) and (k,l) have same diagonal iff $i - j = k - l$ or $i + j = k + l$

- i.e., $j - l = i - k$ or $j - l = k - i$

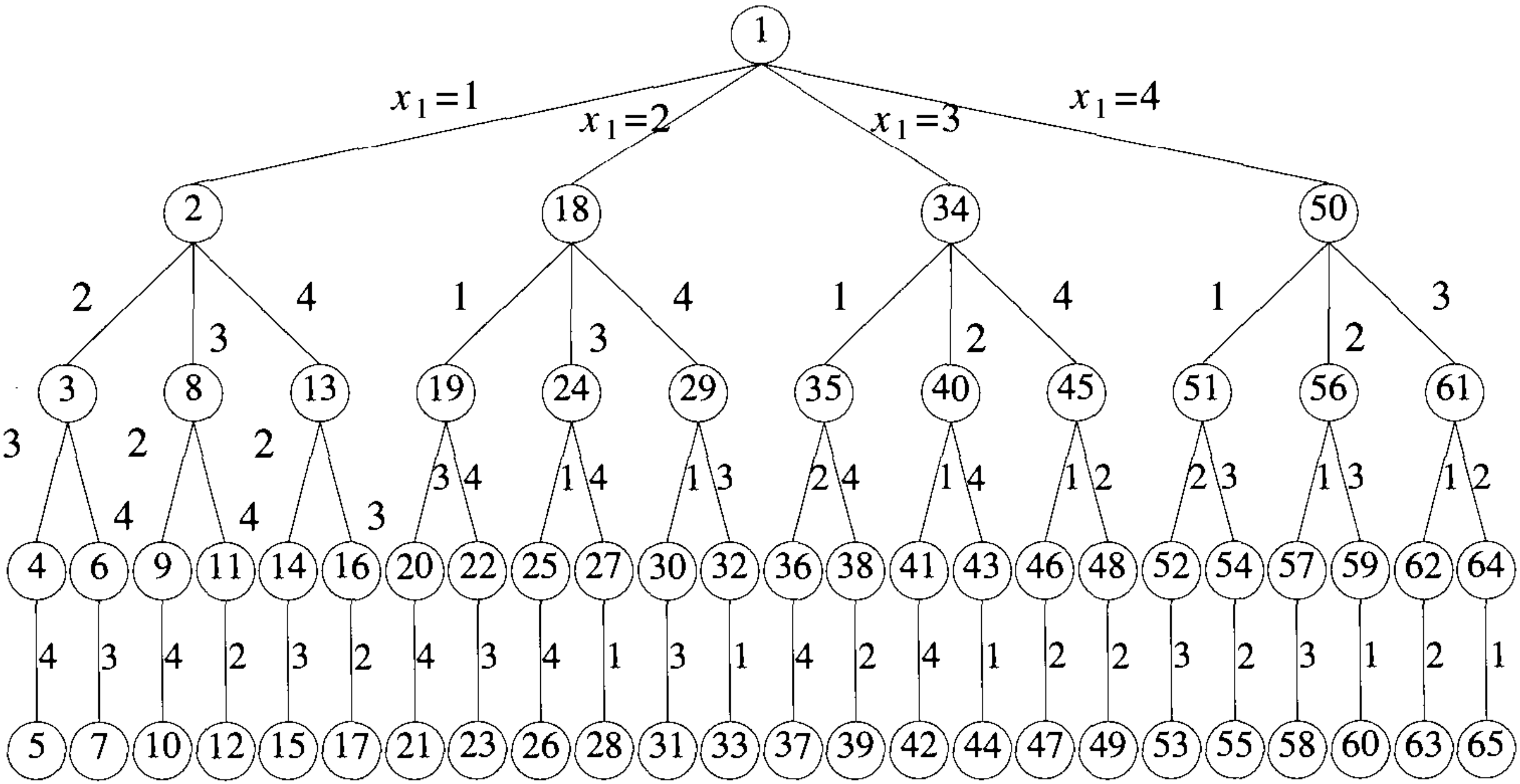
- $\mathbf{|j - l| = |i - k|}$

1	2	3	4	5	6
2					
3					
4	(i,j)				
5					
6					
7					

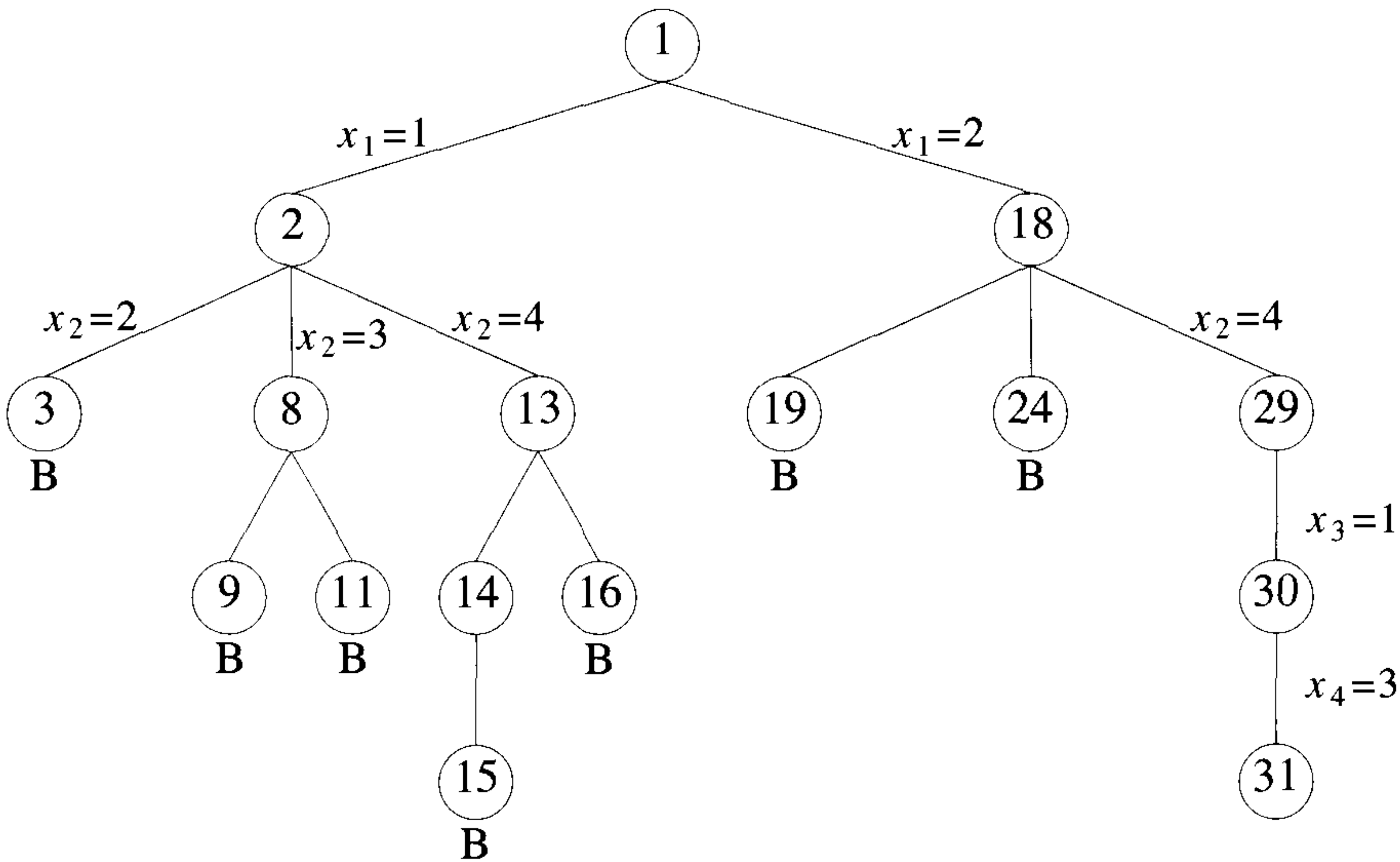
- **State Space Tree:** The tree organization of the solution space is referred to as the **state space tree**.
  - Each node in state space tree defines a problem state.
  - All paths from root to other nodes define state space of problem.
- **Solution states** are those problem states for which the path from root to **s** defines a tuple in solution space.
- **Answer states** are those solution states s for which path from root to s **defines** a tuple that is member of solution (satisfies implicit constraints).

- **Live node:** A node which has been generated and all of whose children are not yet been generated.
- **E node**: A live node whose children are currently been generated (node being expanded).
- **Dead node:** A generated node with all its children expanded

Tree Organization of the 4-queens solution space. Nodes are numbered as in depth first search



Portion of the tree is generated during backtracking



Counting Solutions

Order (“N”)	Total Solutions	Unique Solutions	Exec time
1	1	1	< 0 seconds
2	0	0	< 0 seconds
3	0	0	< 0 seconds
4	2	1	< 0 seconds
5	10	2	< 0 seconds
6	4	1	< 0 seconds
7	40	6	< 0 seconds
8	92	12	< 0 seconds
9	352	46	< 0 seconds
10	724	92	< 0 seconds
11	2,680	341	< 0 seconds
12	14,200	1,787	< 0 seconds
13	73,712	9,233	< 0 seconds
14	365,596	45,752	0.2s

15	2,279,184	285,053	1.9 s
16	14,772,512	1,846,955	11.2 s
17	95,815,104	11,977,939	77.2 s
18	666,090,624	83,263,591	9.6 m
19	4,968,057,848	621,012,754	75.0 m
20	39,029,188,884	4,878,666,808	10.2 h
21	314,666,222,712	39,333,324,973	87.2 h
22	2,691,008,701,644	336,376,244,042	31.9
23	24,233,937,684,440	3,029,242,658,210	296 d
24	227,514,171,973,736	28,439,272,956,934	?
25	2,207,893,435,808,352	275,986,683,743,434	?
26	22,317,699,616,364,044	2,789,712,466,510,289	?

(s = seconds   m = minutes   h = hours   d = days)

END