JICSCI803
Algorithms and Data Structures
March to June 2020

Highlights of Lecture 09

backtracking
branch and bound

Backtracking

Backtracking is a general algorithm strategy for finding all (or some) solutions to some computational problems, notably **constraint satisfaction problems**, 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 can be applied only for problems which admit the concept of a "partial candidate solution" and a **relatively quick test of whether it can possibly be completed to a valid solution**. It is useless, for example, for locating a given value in an unordered table. When it is applicable, however, backtracking is often much faster than brute force enumeration of all complete candidates, since it can eliminate a large number of candidates with a single test.

Backtracking

The classic textbook example of the use of backtracking is the **eight queens puzzle**, that asks for all arrangements of eight chess queens on a standard chessboard so that no queen attacks any other. In the common backtracking approach, the partial candidates are arrangements of *k* queens in the first *k* rows of the board, all in different rows and columns. Any partial solution that contains two mutually attacking queens can be abandoned.

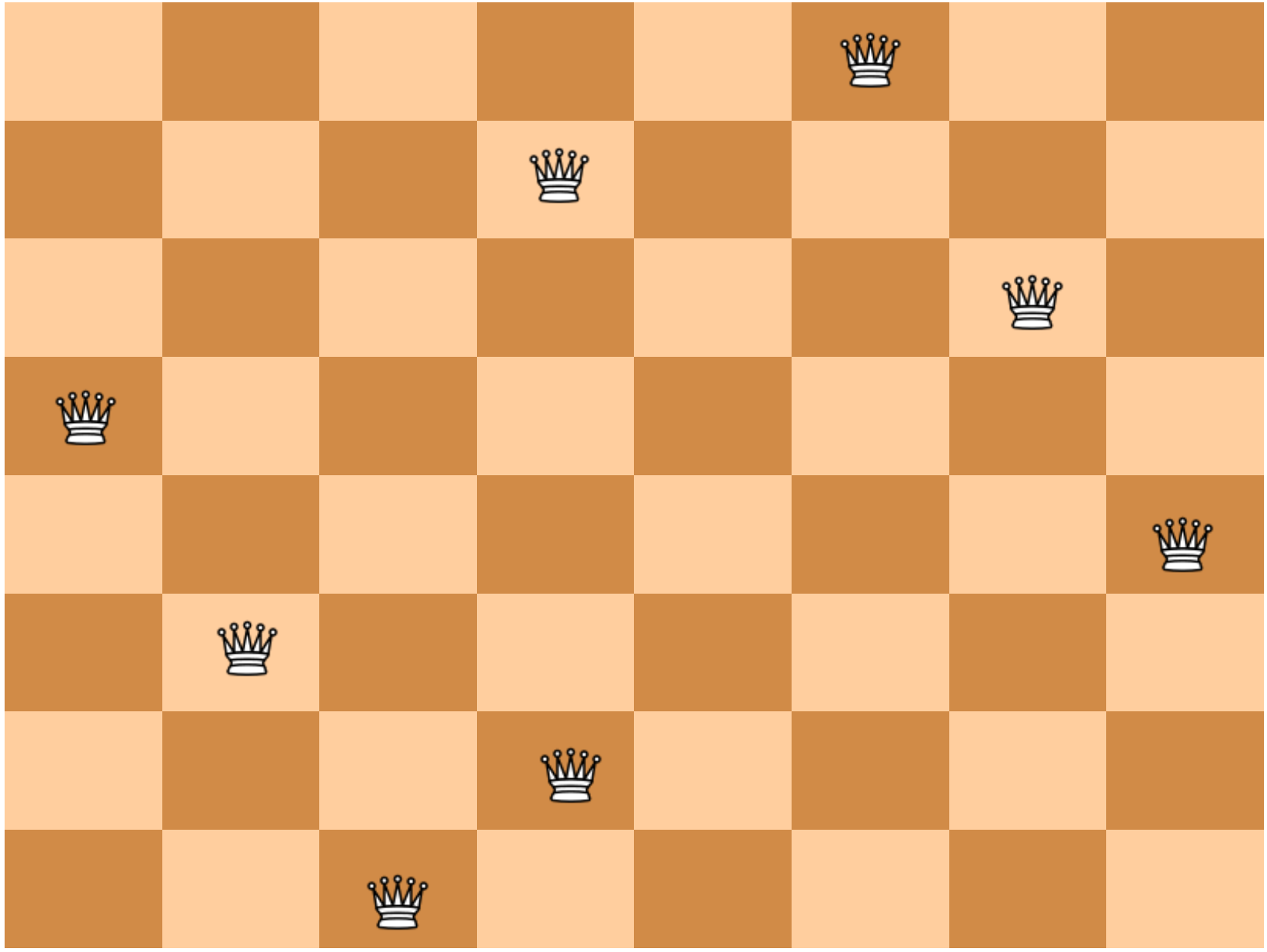
The Eight Queens problem:

• The **eight queens puzzle** is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten 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* non-attacking queens on an *n*×*n* chessboard, for which solutions exist for all natural numbers *n* with the exception of *n*=2 and *n*=3.

Is it a solution of The Eight Queens Problem?



The Eight Queens Problem: evaluation function

- We wish to place 8 queens on a chessboard in such a way that no queen threatens another.

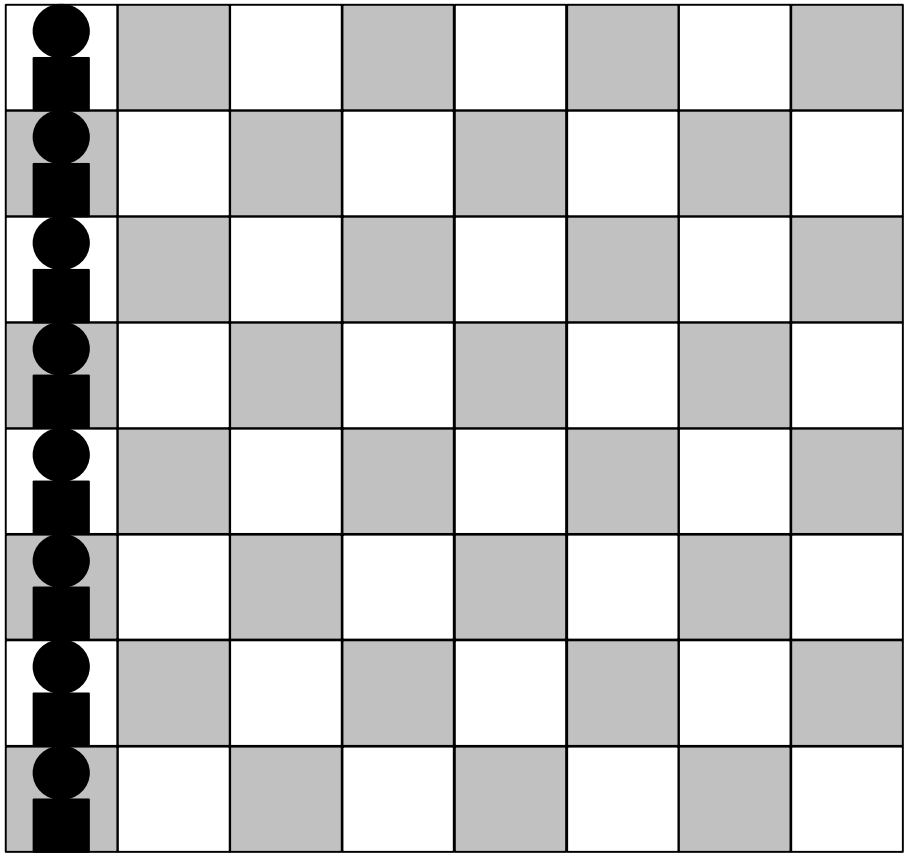
- In other words, no two queens may be in the same row, column or diagonal.

- Let us assume we have a function solution(x) which returns True when x is a solution and False when it is not.

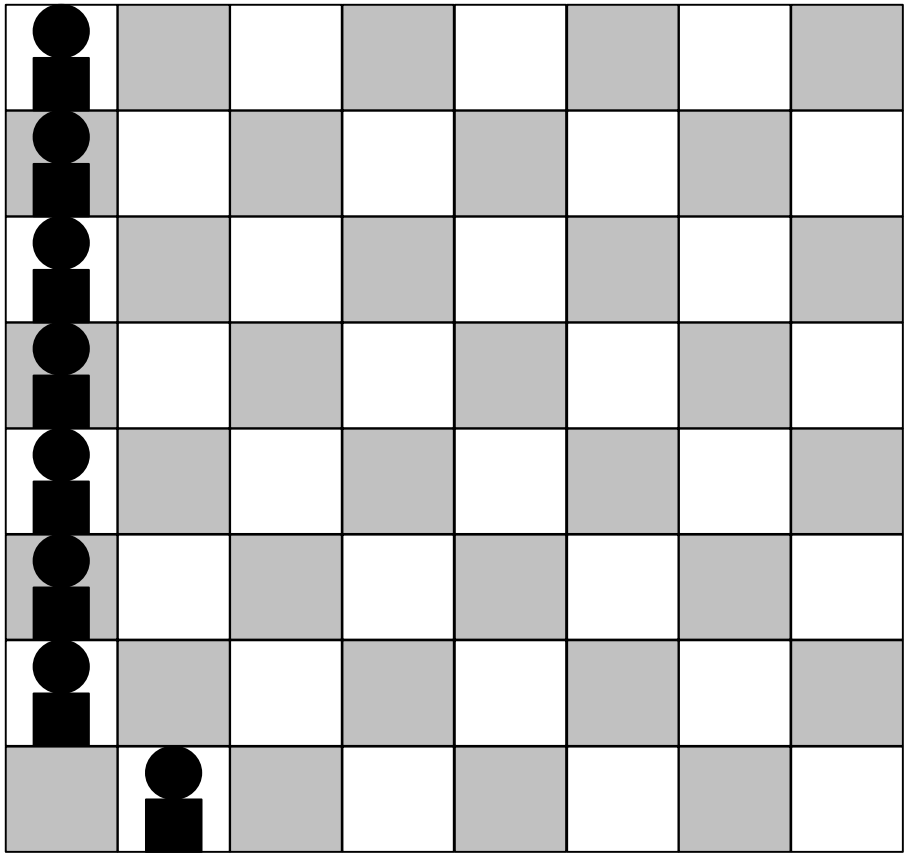
The Eight Queens Problem:
Brute Force Algorithm

```
program queens1
  for i1 = 1 to 8 do
    for i2 = 1 to 8 do
        …
          for i8 = 1 to 8 do
              sol = [i1, i2, …, i8]
              if solution(sol) then
                 write sol
                 stop
  write “no solution”
```

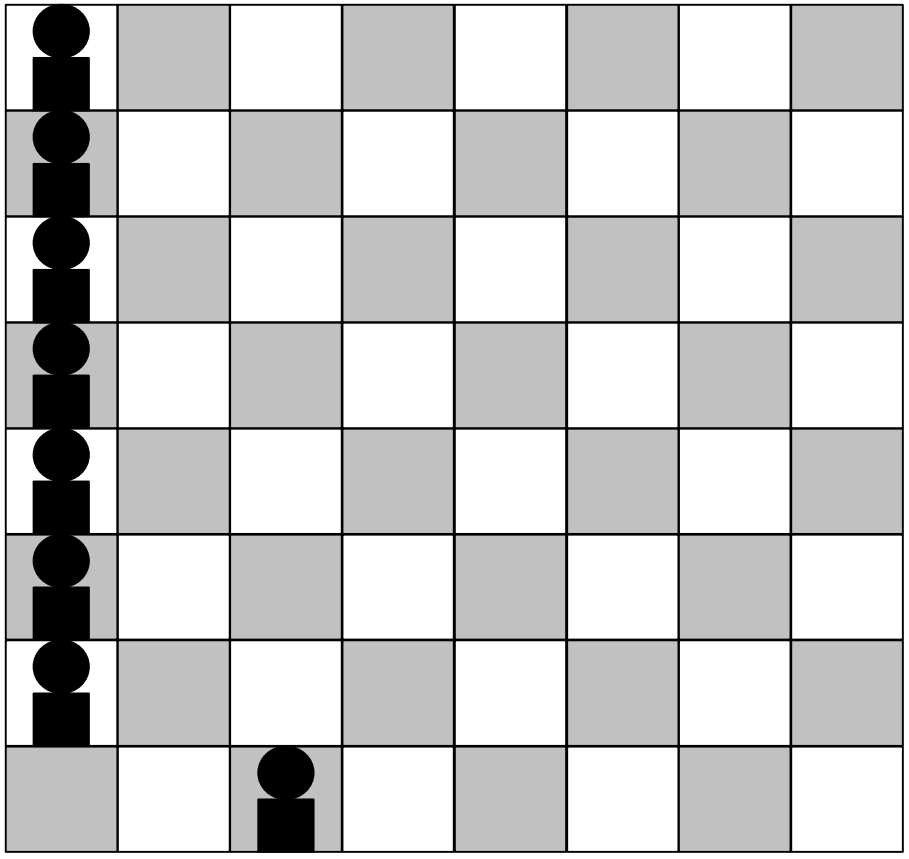
The Eight Queens problem



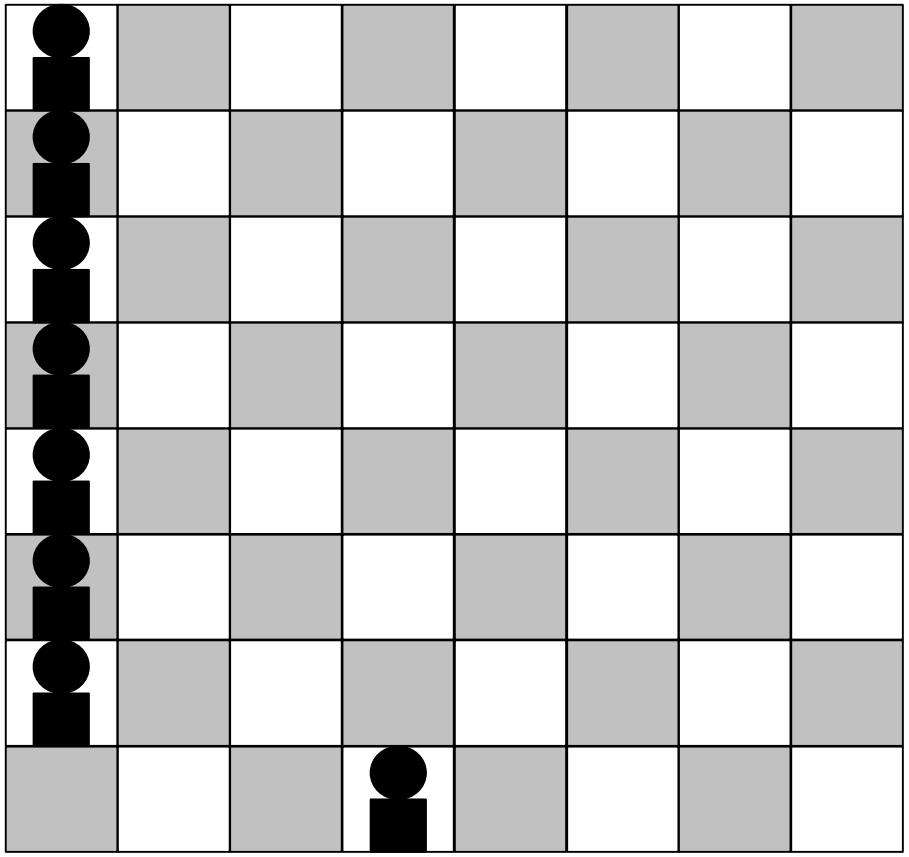
The Eight Queens problem



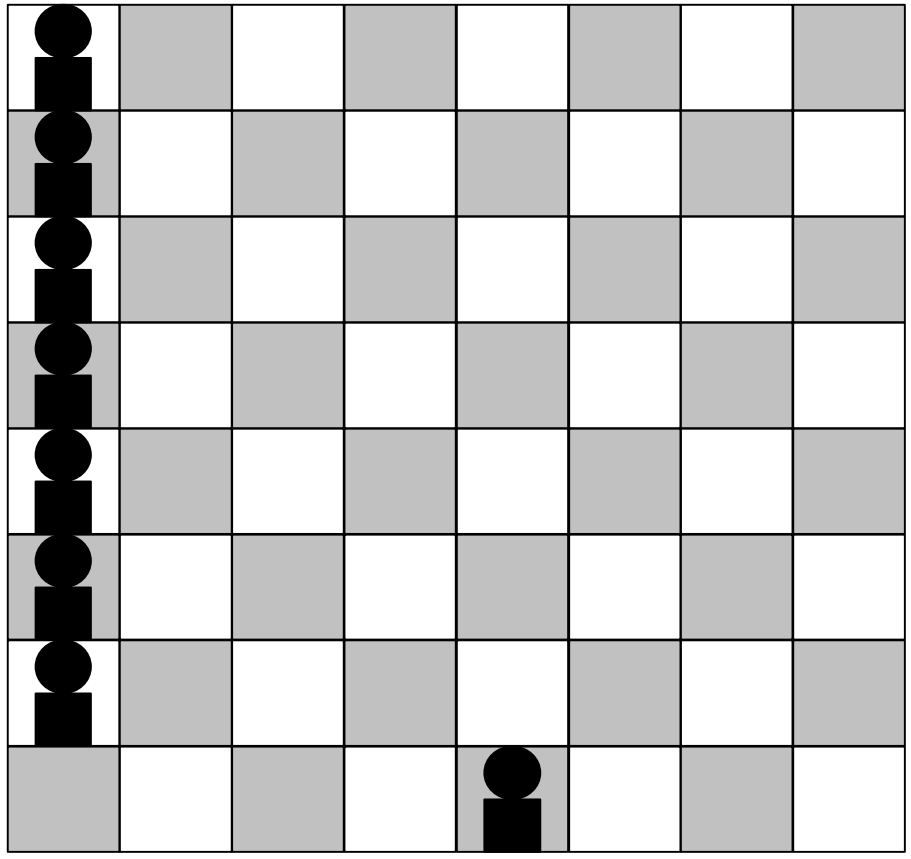
The Eight Queens problem



The Eight Queens problem

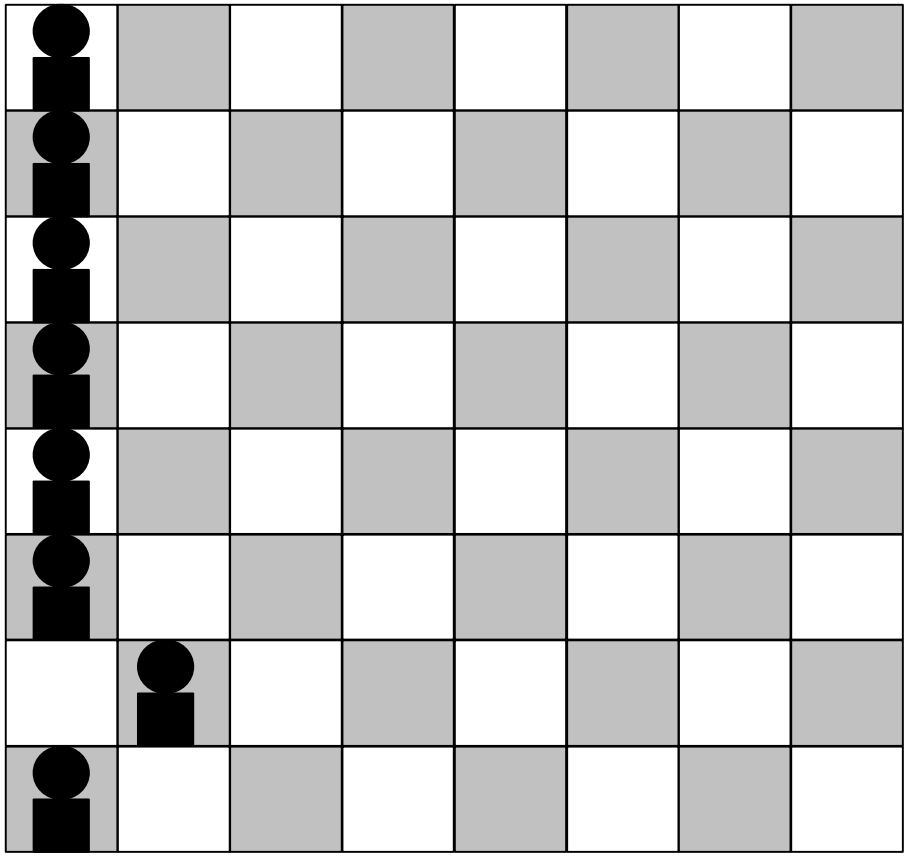


The Eight Queens problem

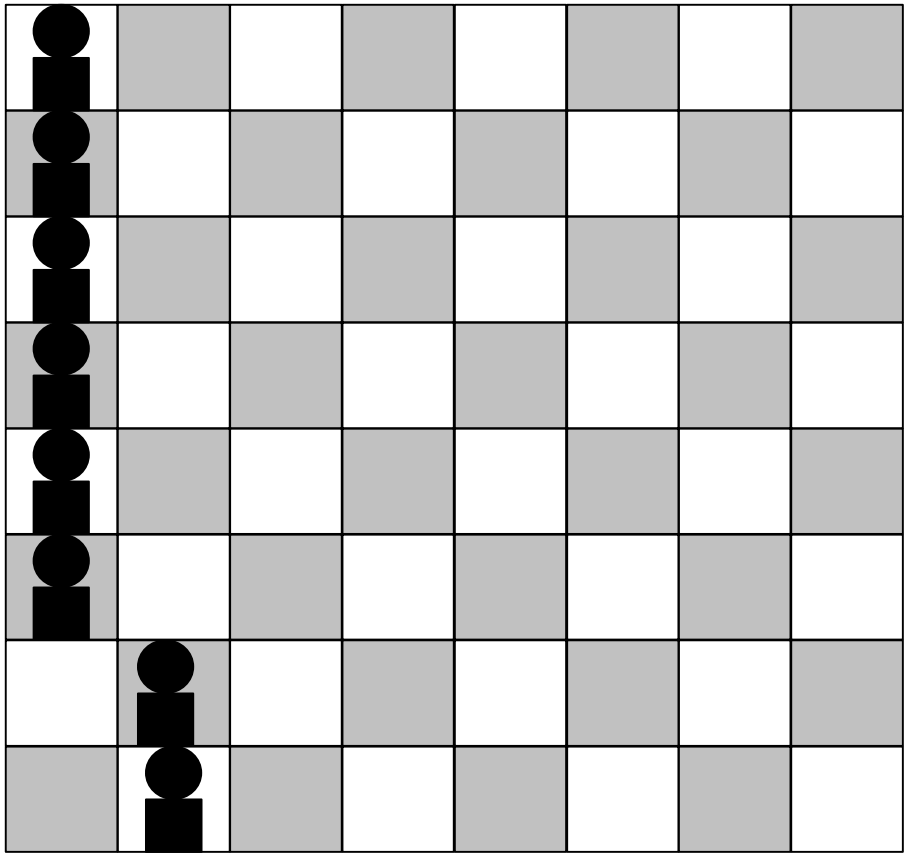


etc…

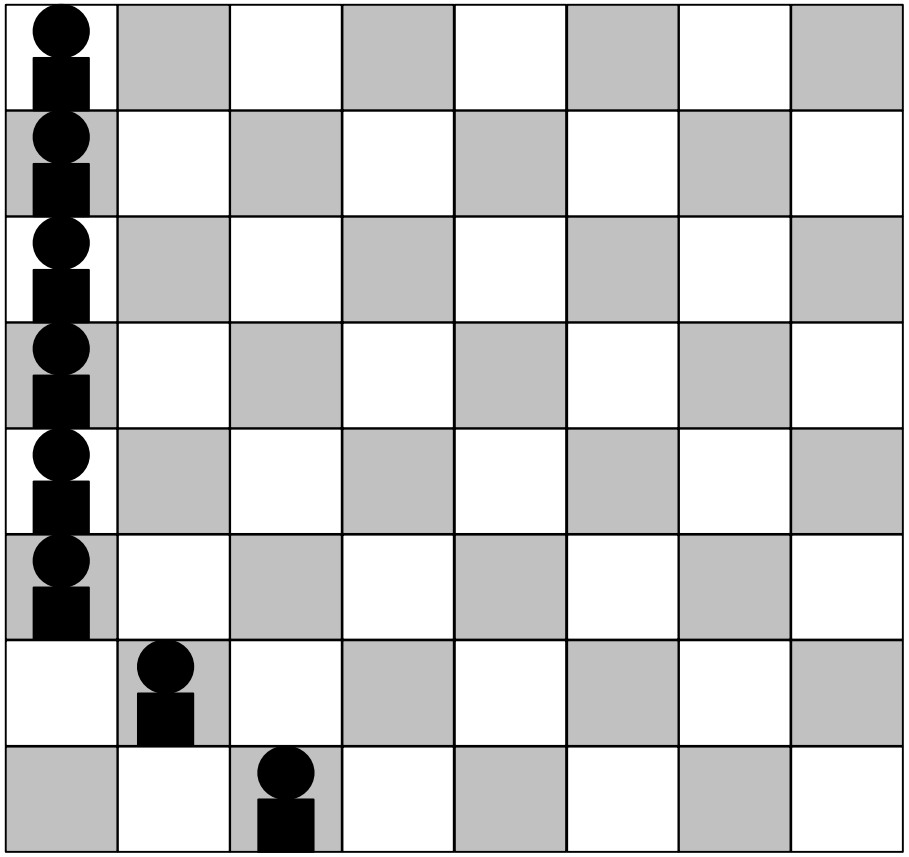
The Eight Queens problem



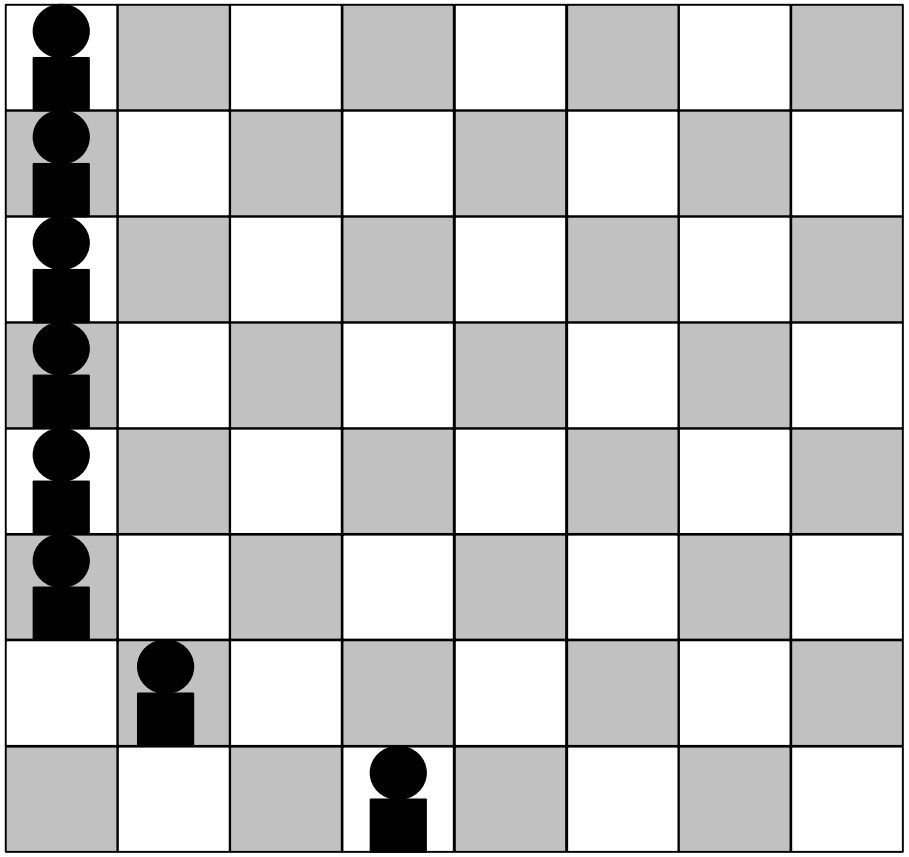
The Eight Queens problem



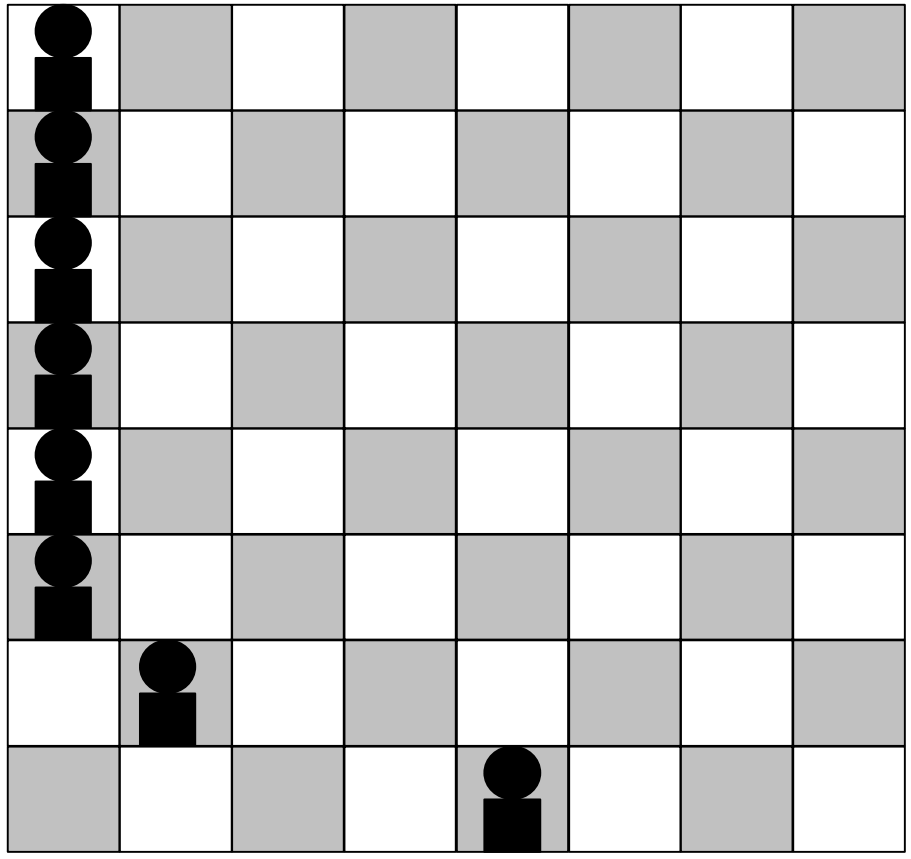
The Eight Queens problem



The Eight Queens problem



The Eight Queens problem

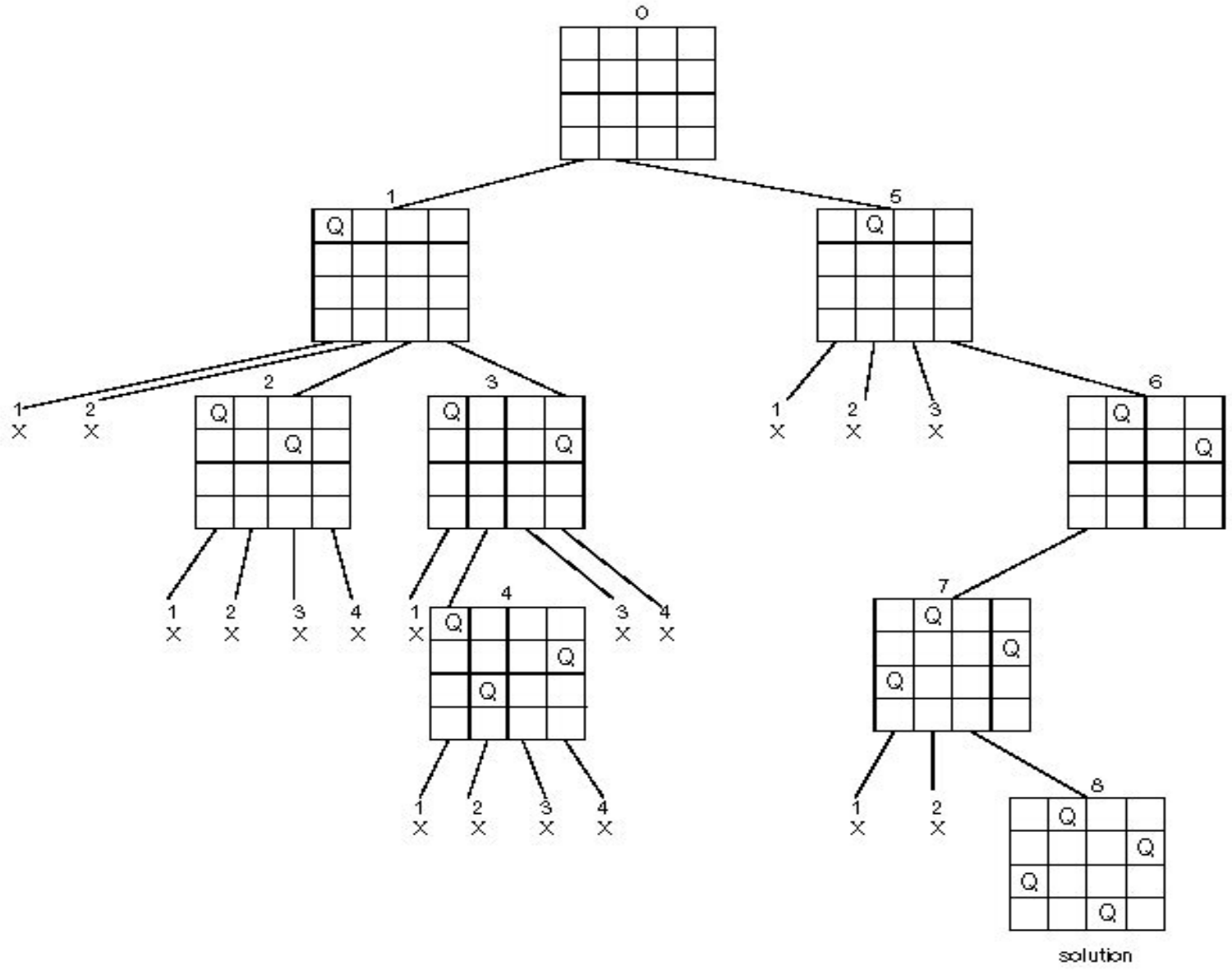


etc…

The Eight Queens problem

- There are 8x8x8x8x8x8x8x8 = 8^8 = 16,777,216 possible arrangements.
- In fact the algorithm stops after 1,299,852 positions because a solution is found.
- The algorithm queens1, by using an array, implicitly prevents two queens being in the same row.
- By requiring each element of sol[] to have a different value, from 1 to 8, we can implicitly require each queen to be in a different column.
- Each possible sol is now a permutation of 1...8.

State-space of the four-queens problem

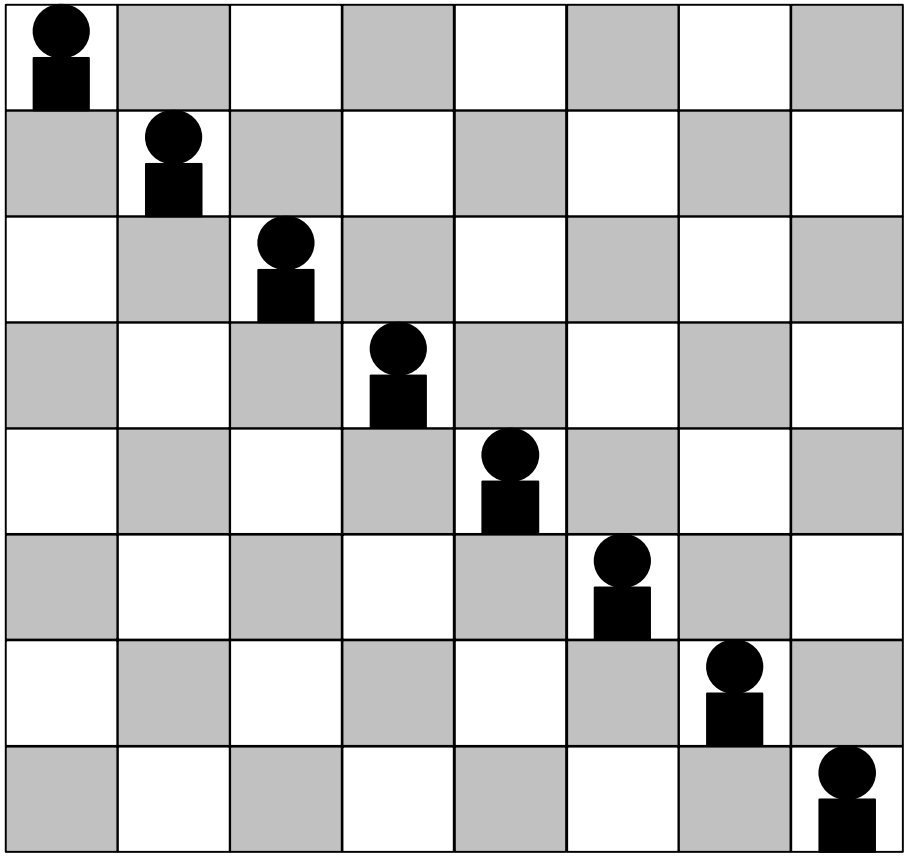


The Eight Queens Problem

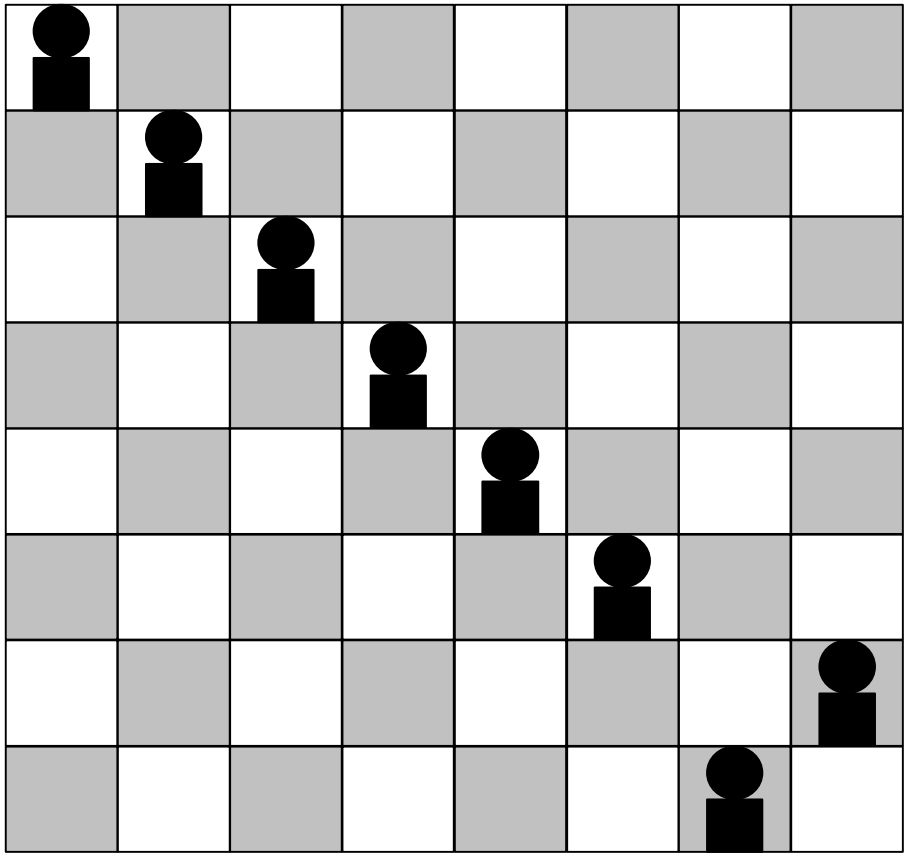
```
program queens2
    sol = [1, 2, 3, 4, 5, 6, 7, 8]
    final = [8, 7, 6, 5, 4, 3, 2, 1]
    while sol ≠ final and not solution(sol) do
        sol = next_permutation(sol)
    if solution(sol) then
        write sol
    else
        write “no solution”
```

Sol[i] = j means that a queen is at row i and col j

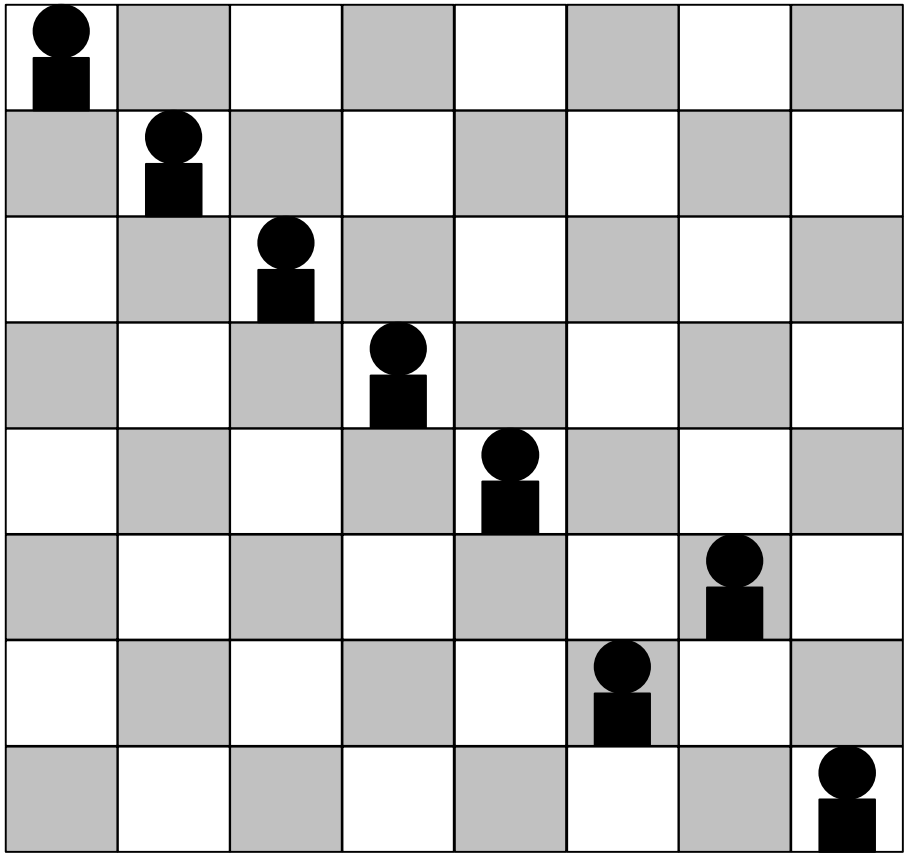
The Eight Queens problem



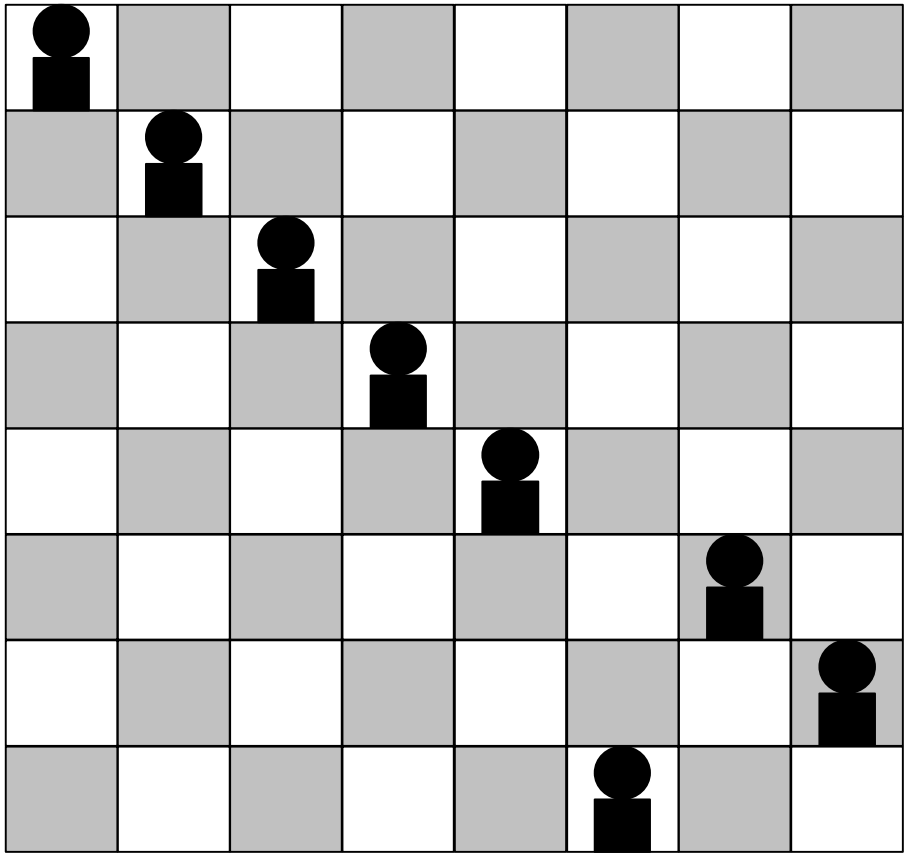
The Eight Queens problem



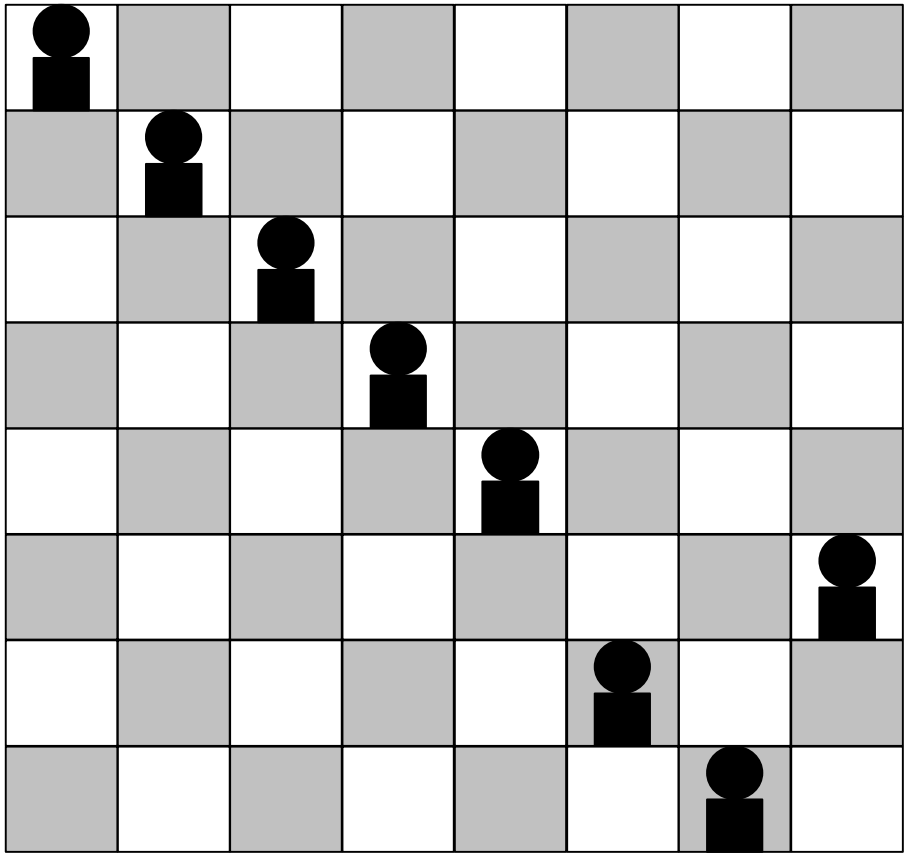
The Eight Queens problem



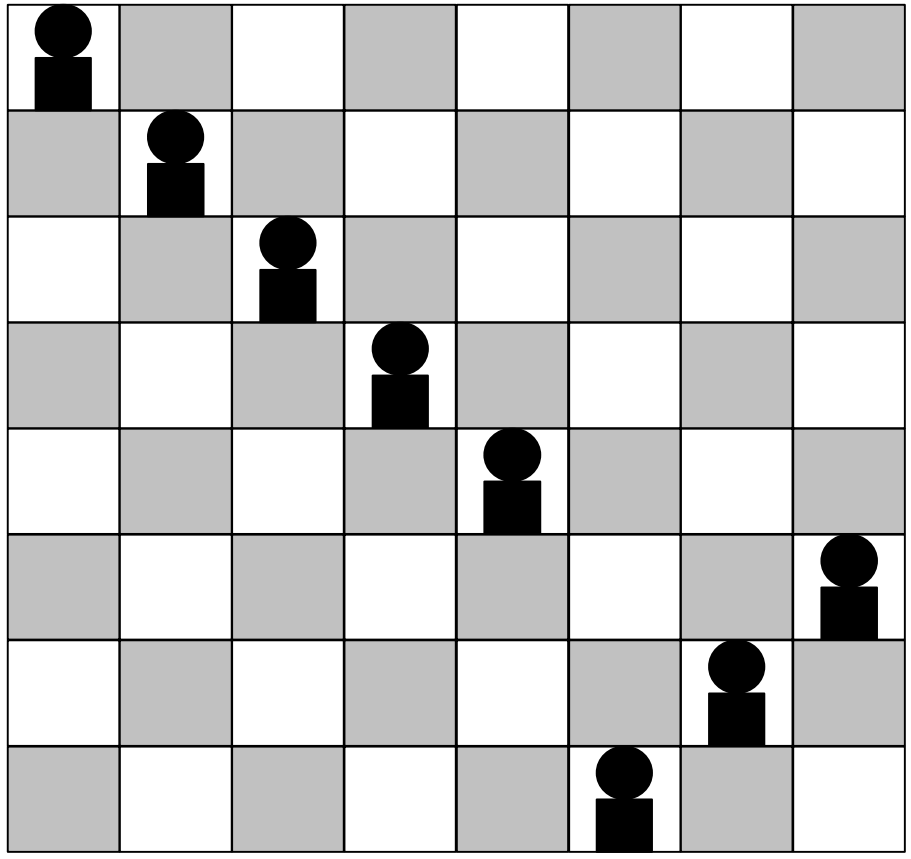
The Eight Queens problem



The Eight Queens problem



The Eight Queens problem



etc.

The Eight Queens problem

– This approach results in only 8! = 40320 positions we have to test.
– In fact, the algorithm stops after 2830 positions have been tested.
– It is harder to get each position than in queens1. e.g. **`next_permutation is not easy.`**
– There are fewer positions to check.
– Is there a better approach to the problem?

The Eight Queens problem

– We note that both queens1 and queens2 only evaluate a potential solution after all eight queens are placed.
– What if we instead evaluate partial positions as we create them?
– Let us restate the problem as a **tree searching problem**.
– We say a vector V[1..*k*] of integers between 1 and 8 is *k*-promising if none of the *k* queens in positions (1, V[1]), (2,V[2]),…, (*k*, V[*k*]) threatens any of the others.

The Eight Queens Problem

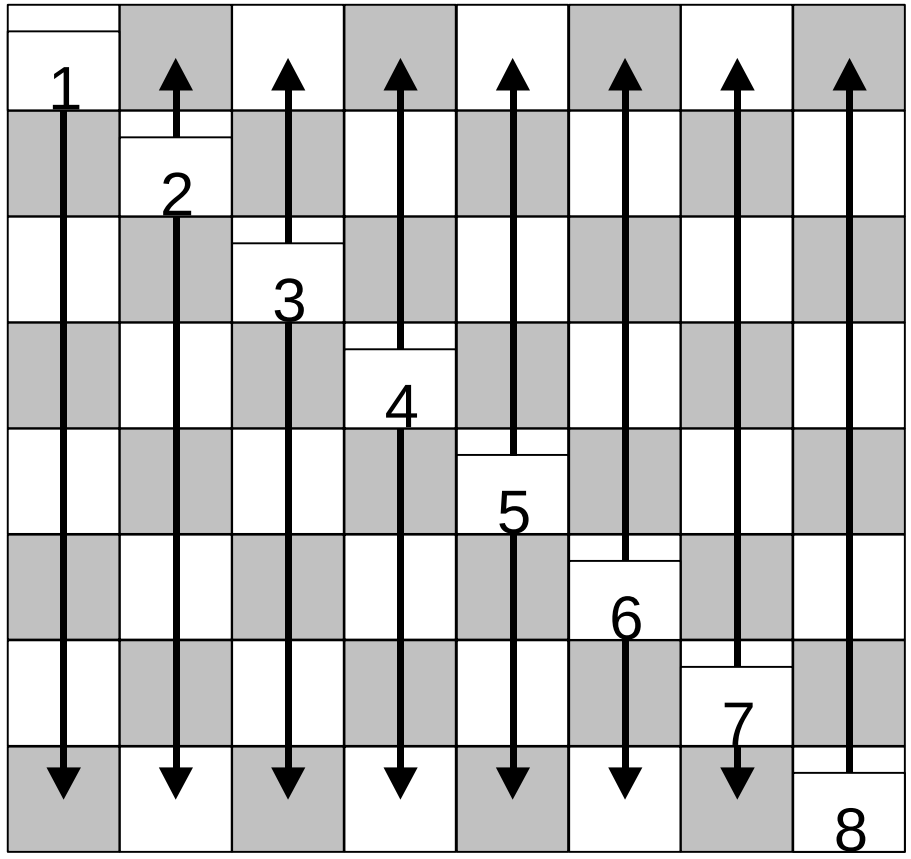
– Let *N* be the set of *k*-promising vectors, $0 \le k \le 8$.
– Let $G=\langle N, A\rangle$ be the directed graph such that $(u, v)\in A$ iff $\exists\ 0 \le k < 8$, such that
  •*u is k-promising*
  •*v is (k+1)-promising*
  •*u[i] = v[i], 1 ≤ i ≤ k*
– This graph is a tree.
– Its root is the empty vector corresponding to $k = 0$.
– We do not generate the tree explicitly.

The Eight Queens Problem

- At each step we try to add another queen to the board.
- We can only add a queen if there is a free column in which the queen can be placed such that no queens lie in the 45° and 135° diagonals from the new queen.
- We can test this by maintaining three sets col, diag45 and diag135 which note the columns and diagonals already occupied by queens.

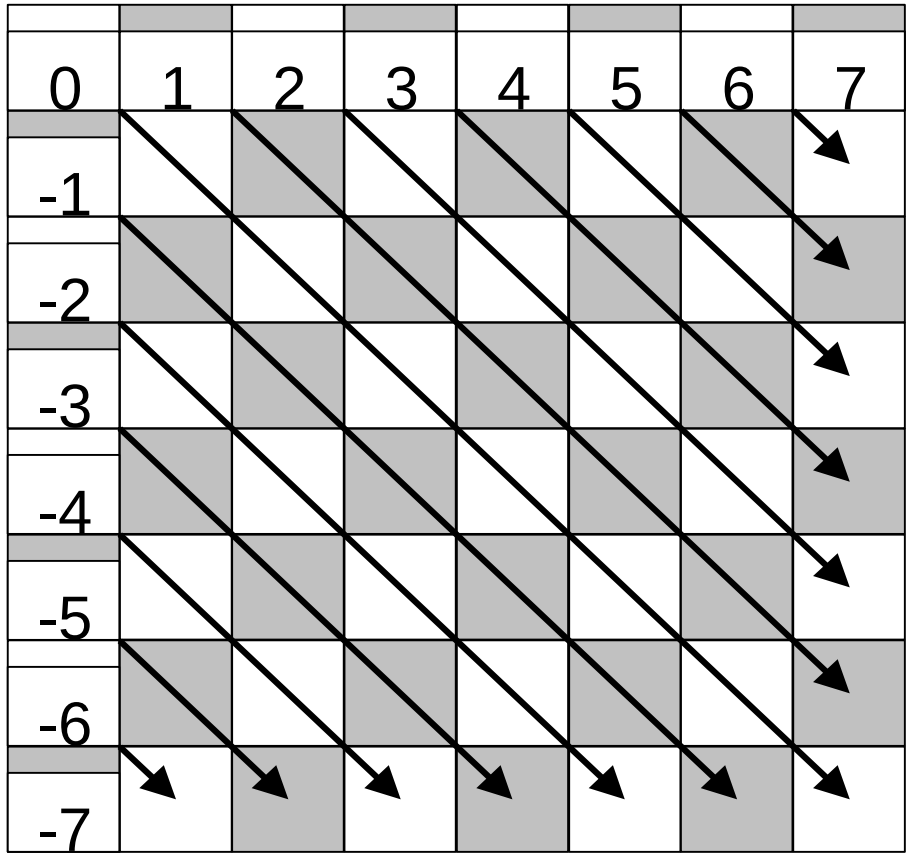
The Eight Queens problem

col



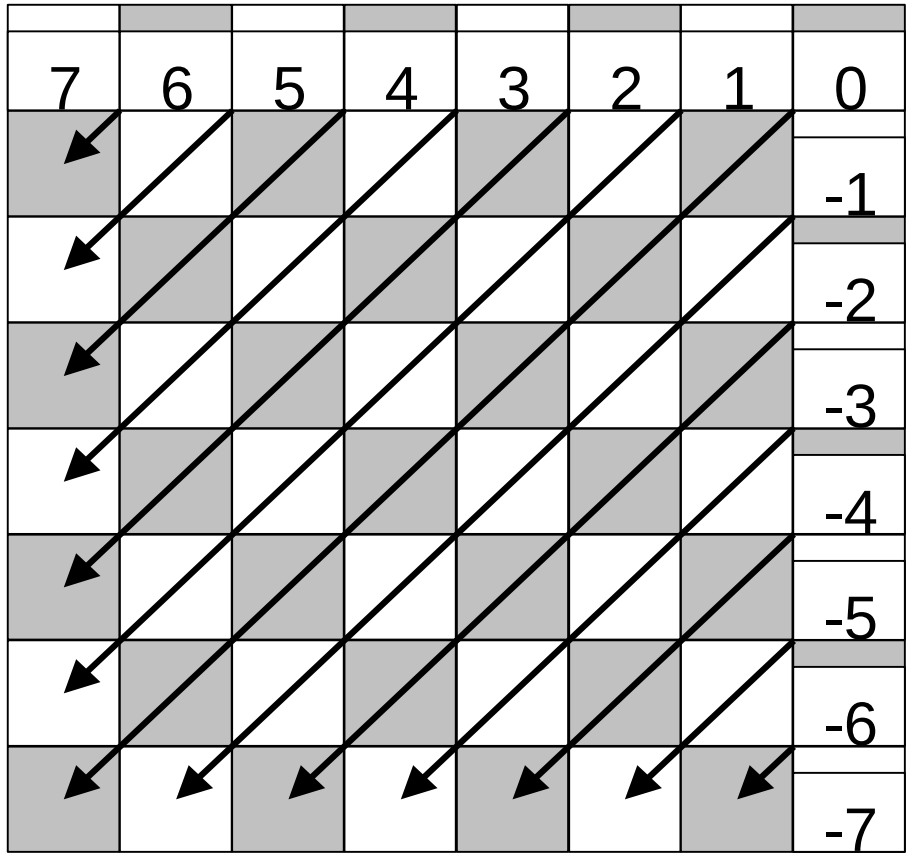
The Eight Queens problem

diag45



The Eight Queens problem

diag135



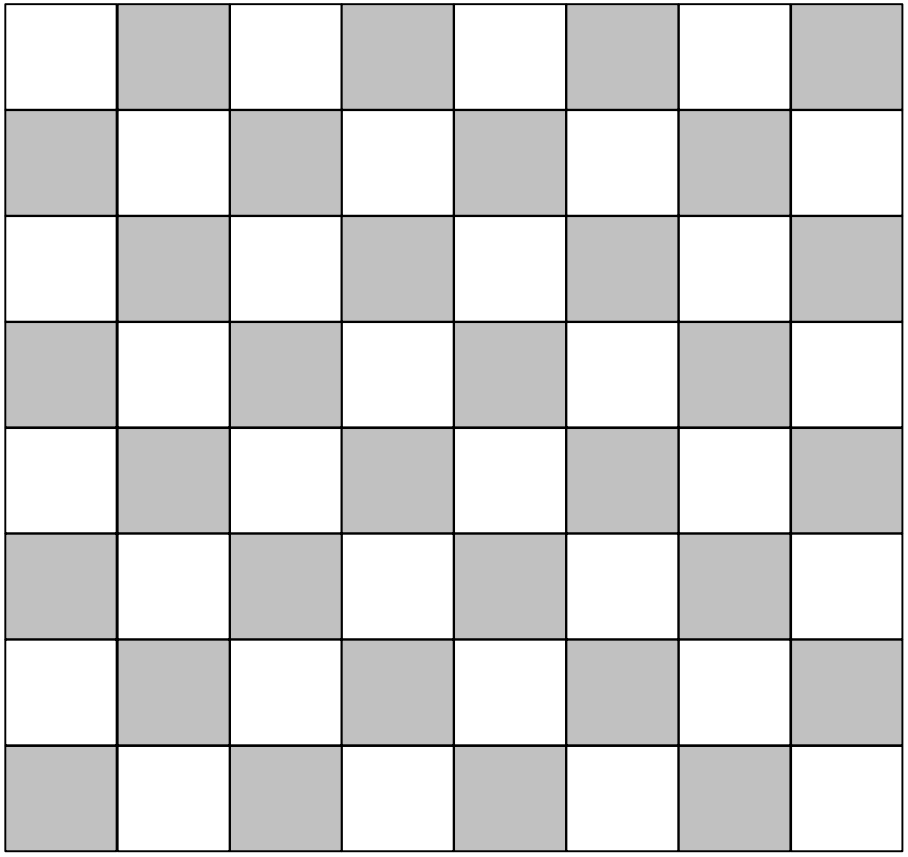
The Eight Queens problem

– If a queen is in location (i, j)
  - sol[i] = j
  - sol[i] $\in$ col
  - sol[i] –i $\in$ diag45
  - 9 –(sol[i] + i) $\in$ diag135
– We can now write a procedure to find and print all the solutions to the eight queens problem.
– We simply call queens(0, ϕ, ϕ, ϕ).

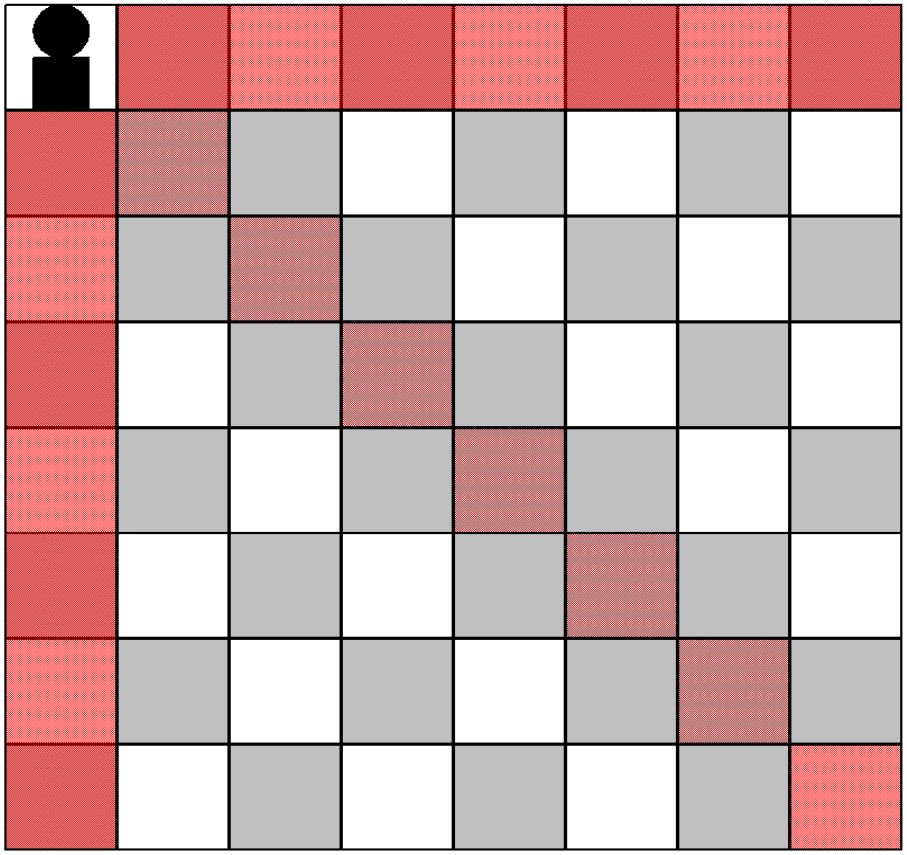
The Eight Queens Problem

```
procedure queens(k, col, diag45, diag135)
  if k = 8 then
    write sol
  else
    for j = 1 to 8 do
     if j∈col and j –(k + 1)∈diag45 and
        9 –(j + (k + 1))∈diag135 then
          sol[k + 1] = j
          queens(k + 1, col∪{j},
          diag45∪{j –(k + 1)},
          diag135∪{9 –(j + (k + 1))}
```

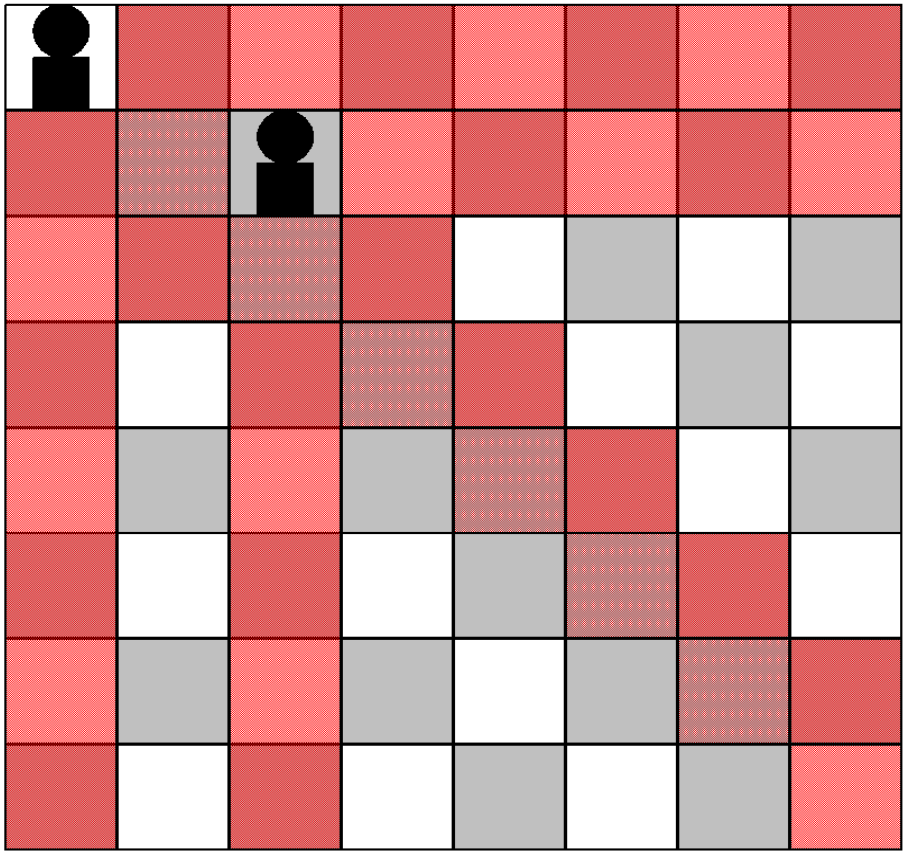
The Eight Queens problem



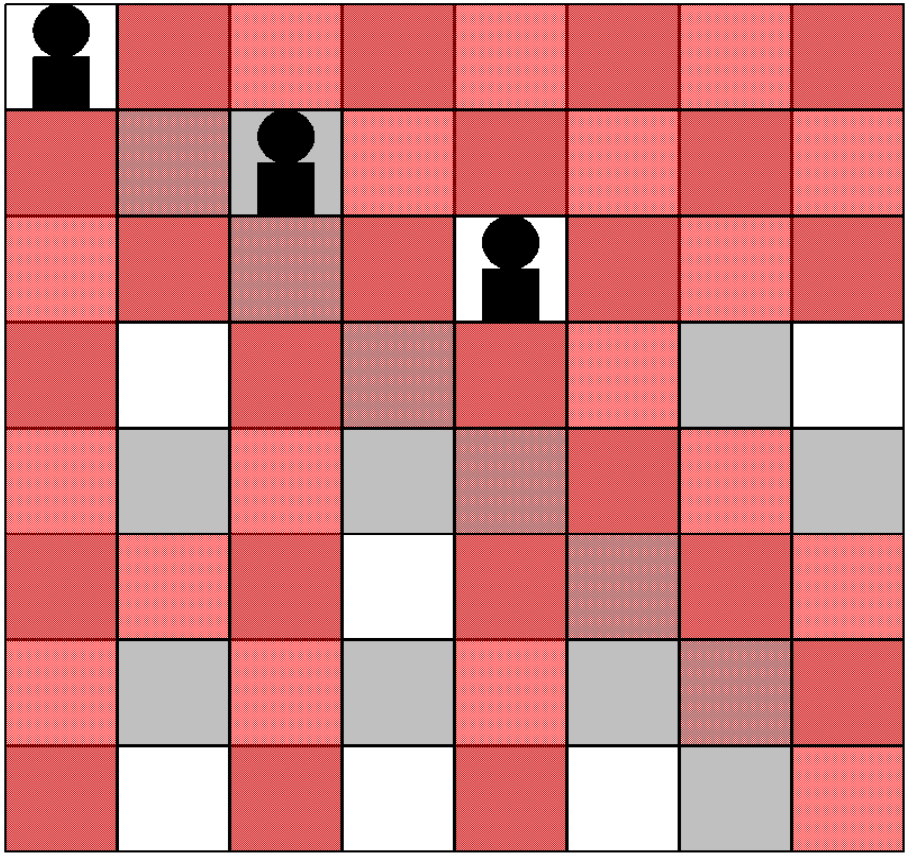
The Eight Queens problem



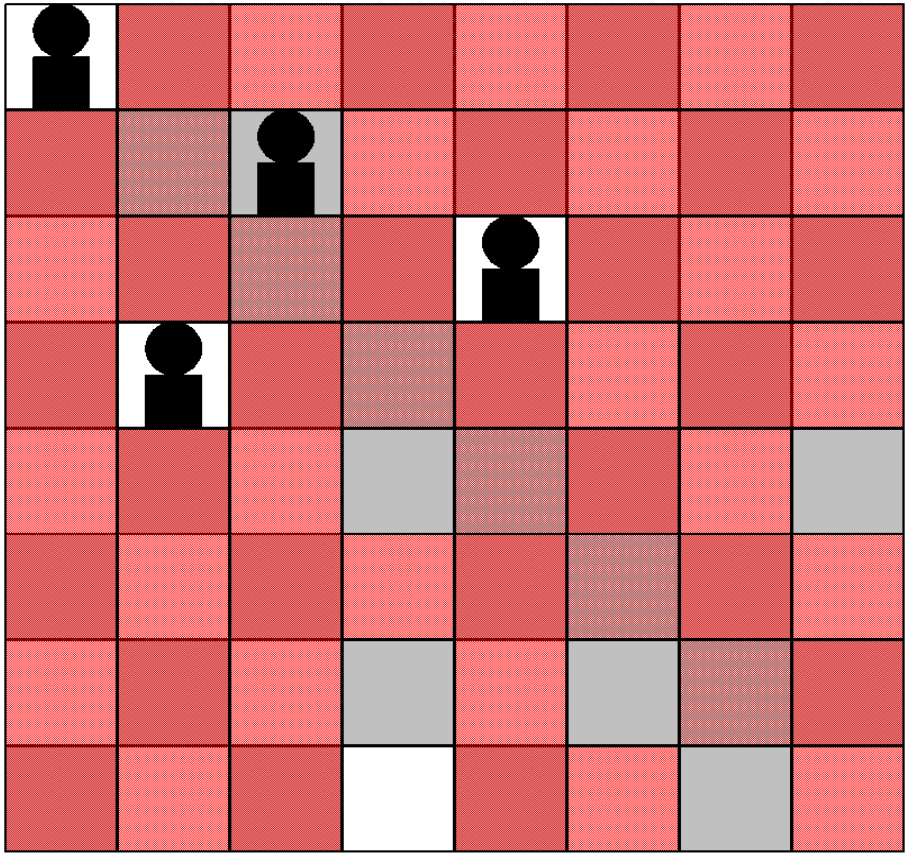
The Eight Queens problem



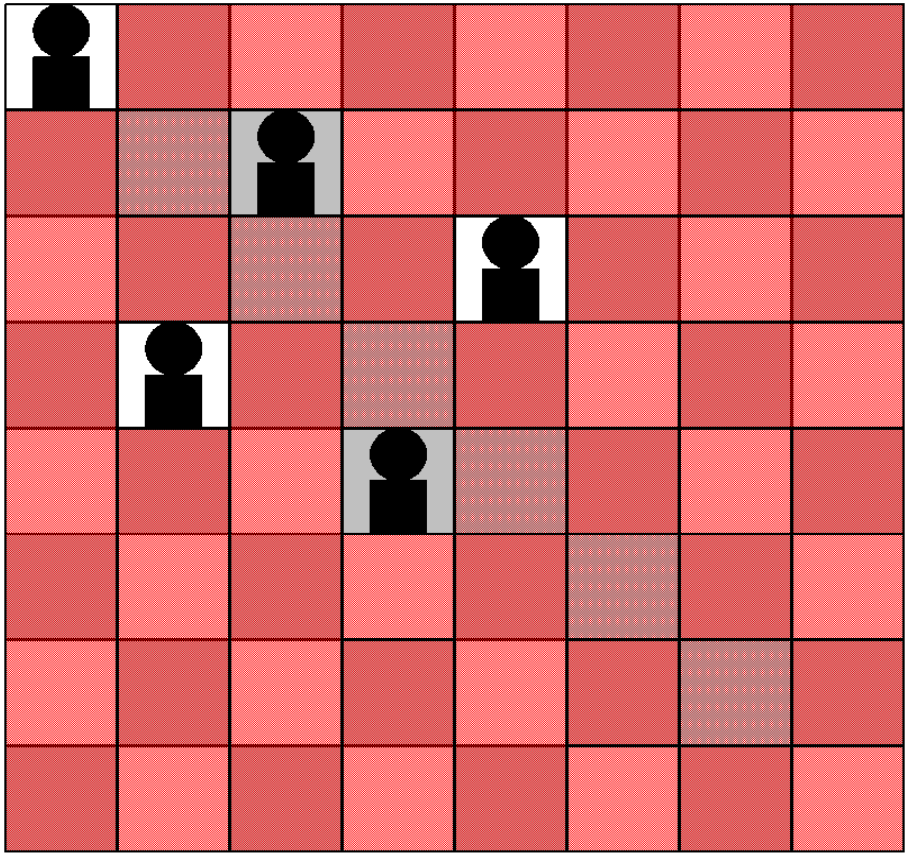
The Eight Queens problem



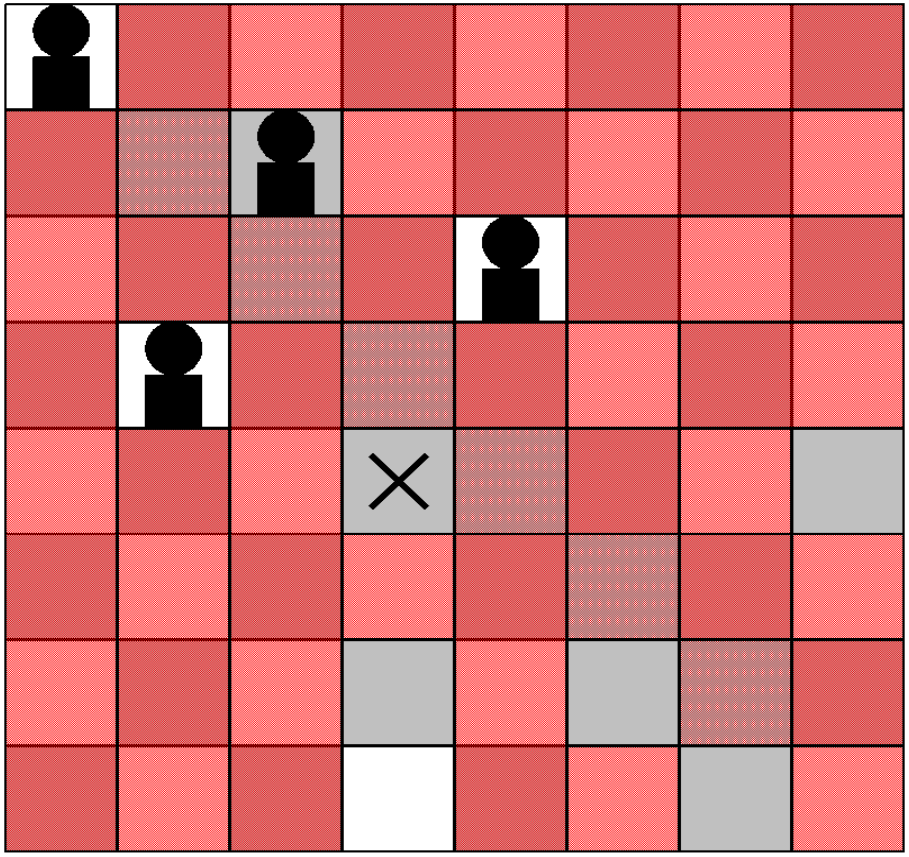
The Eight Queens problem



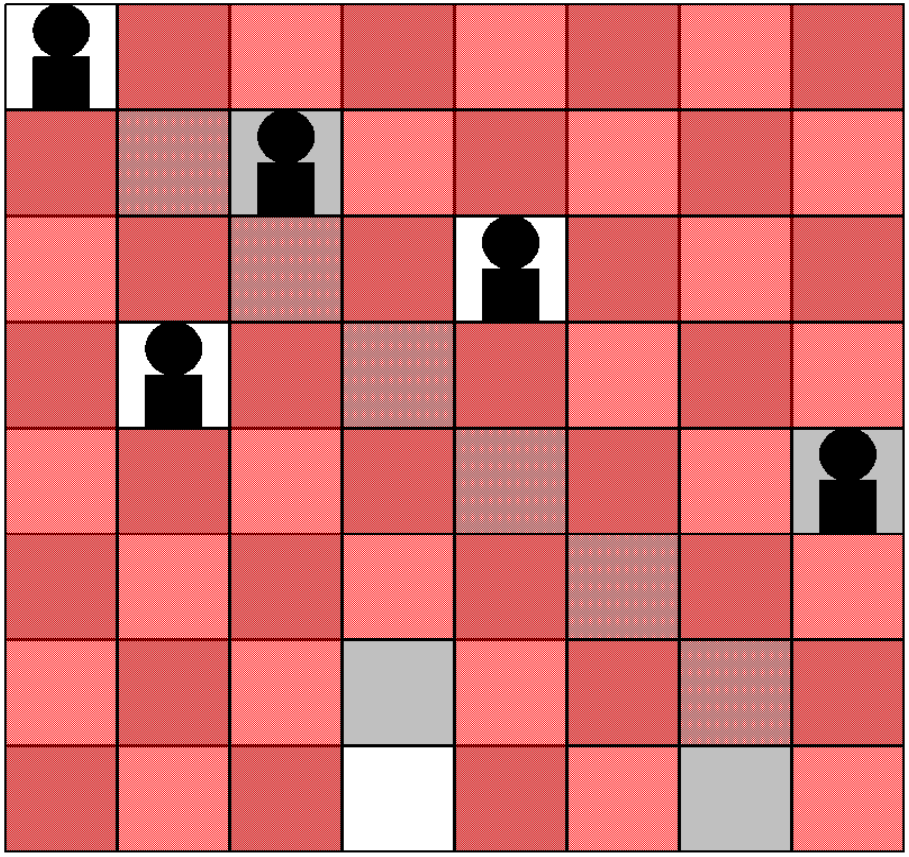
The Eight Queens problem



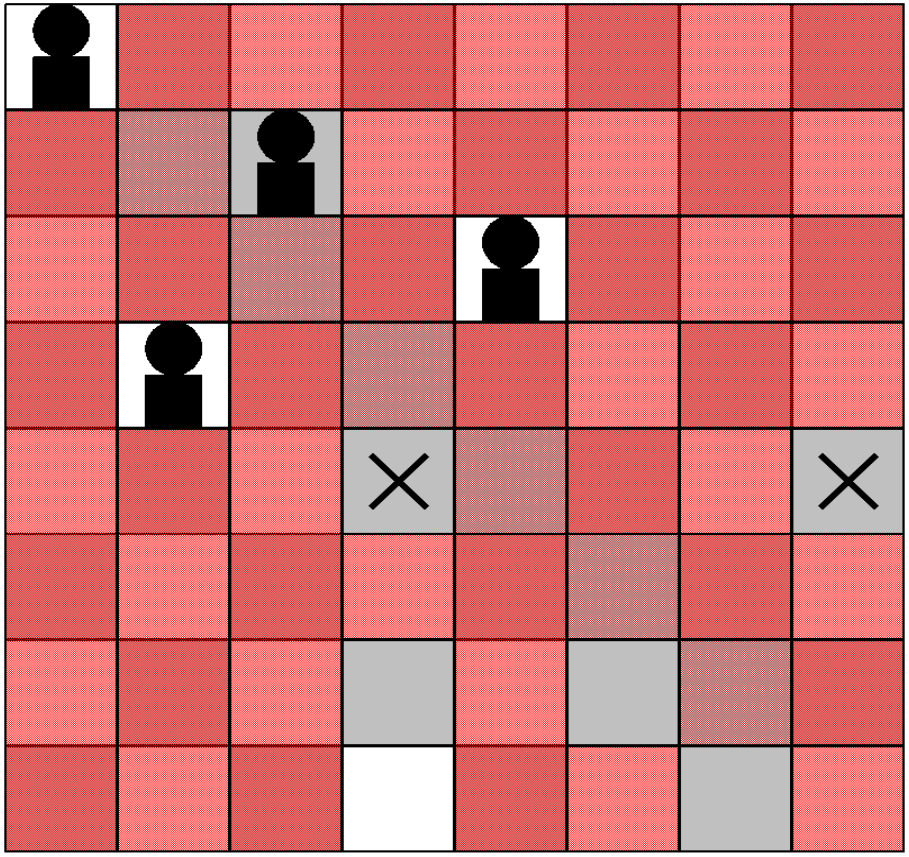
The Eight Queens problem



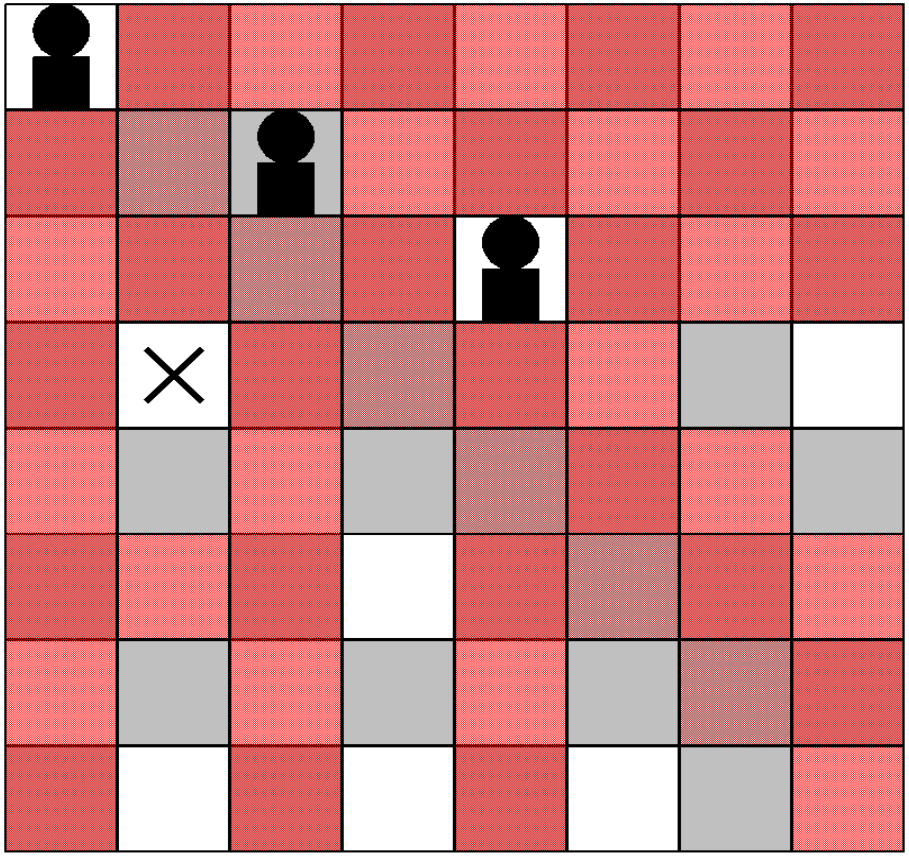
The Eight Queens problem



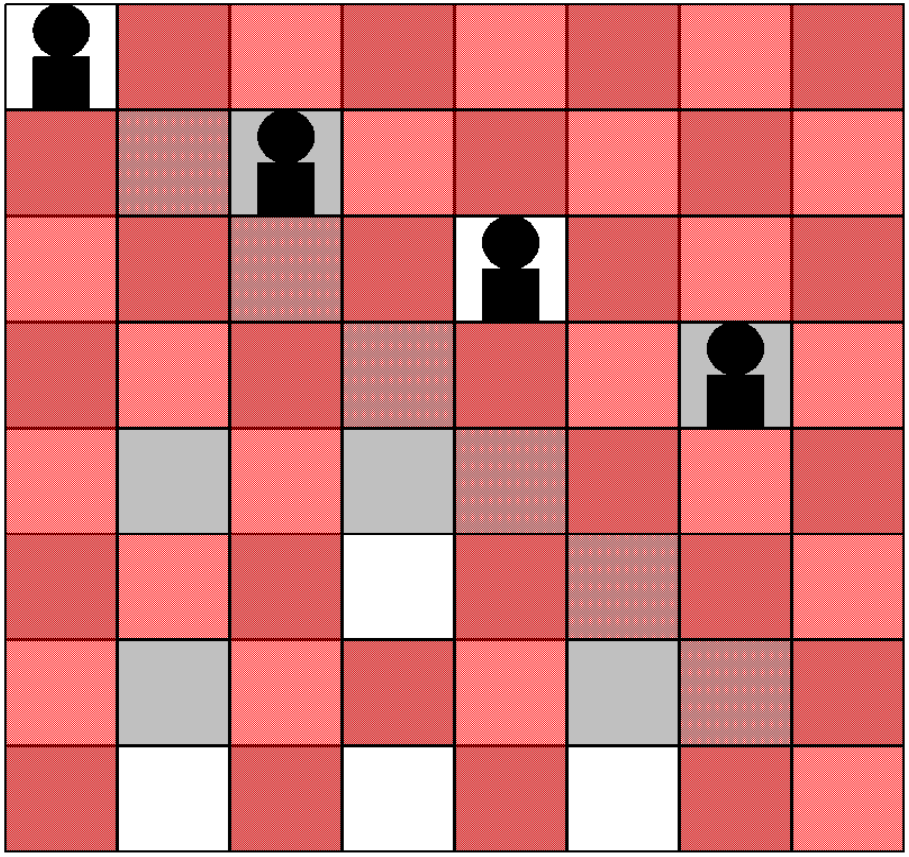
The Eight Queens problem



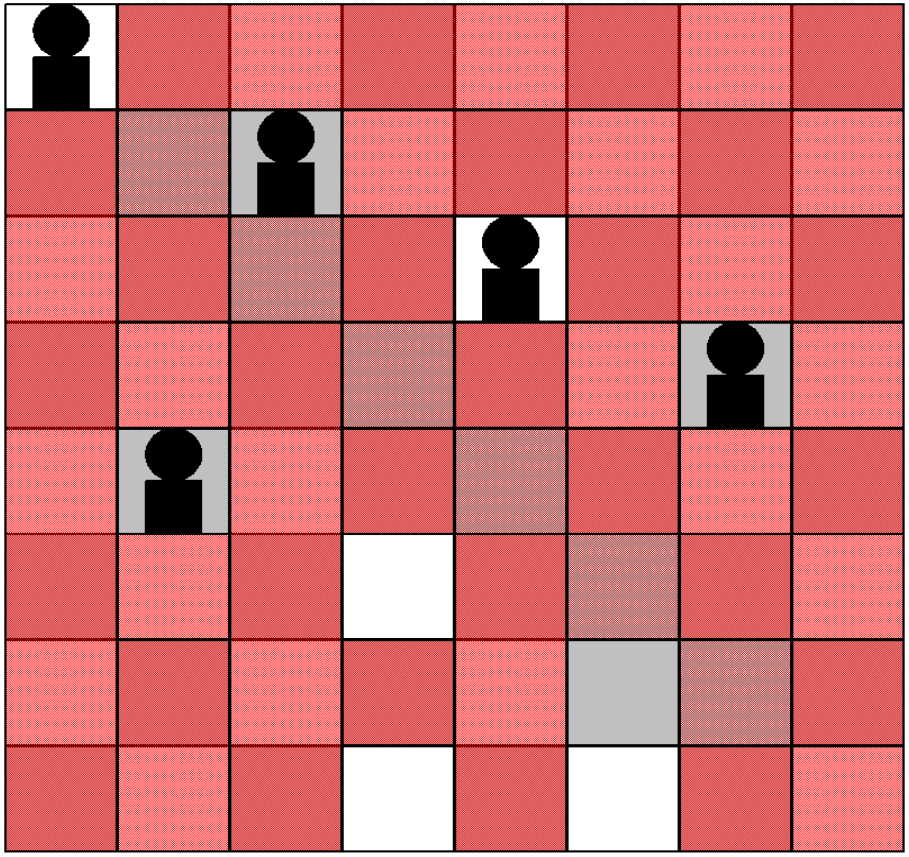
The Eight Queens problem



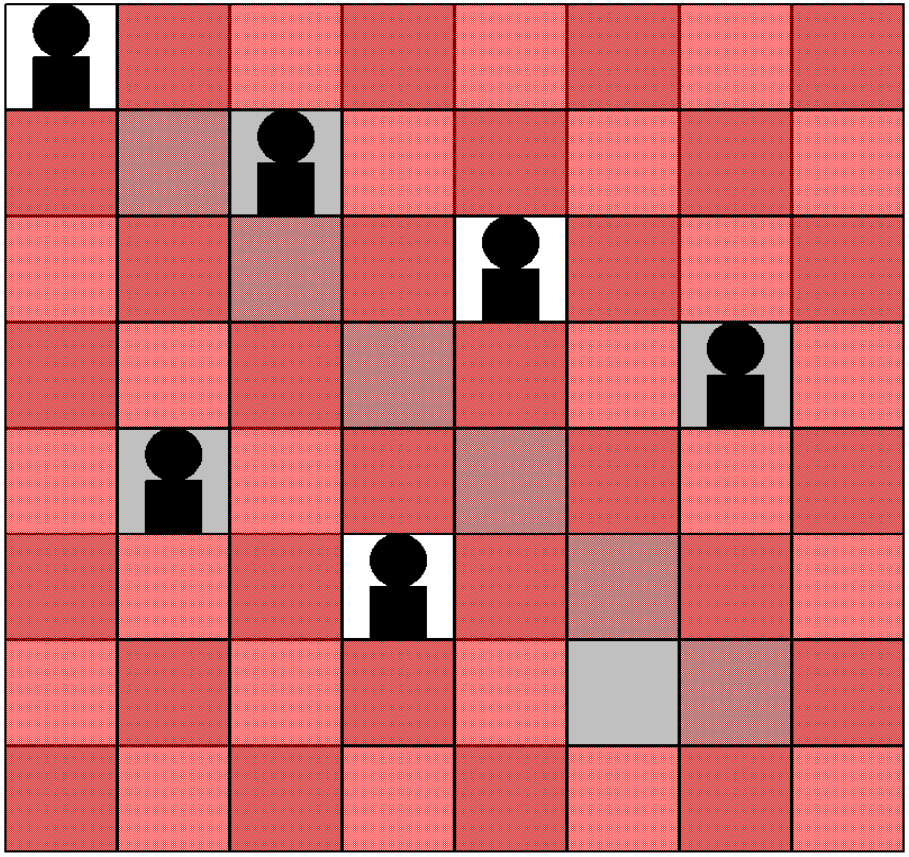
The Eight Queens problem



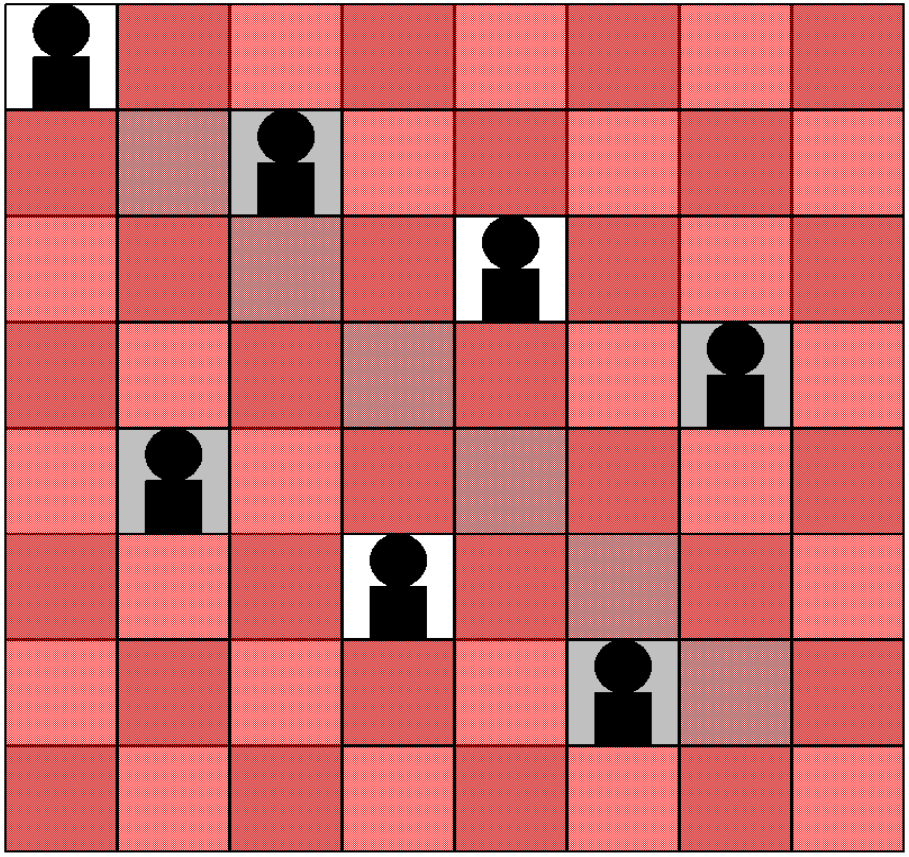
The Eight Queens problem



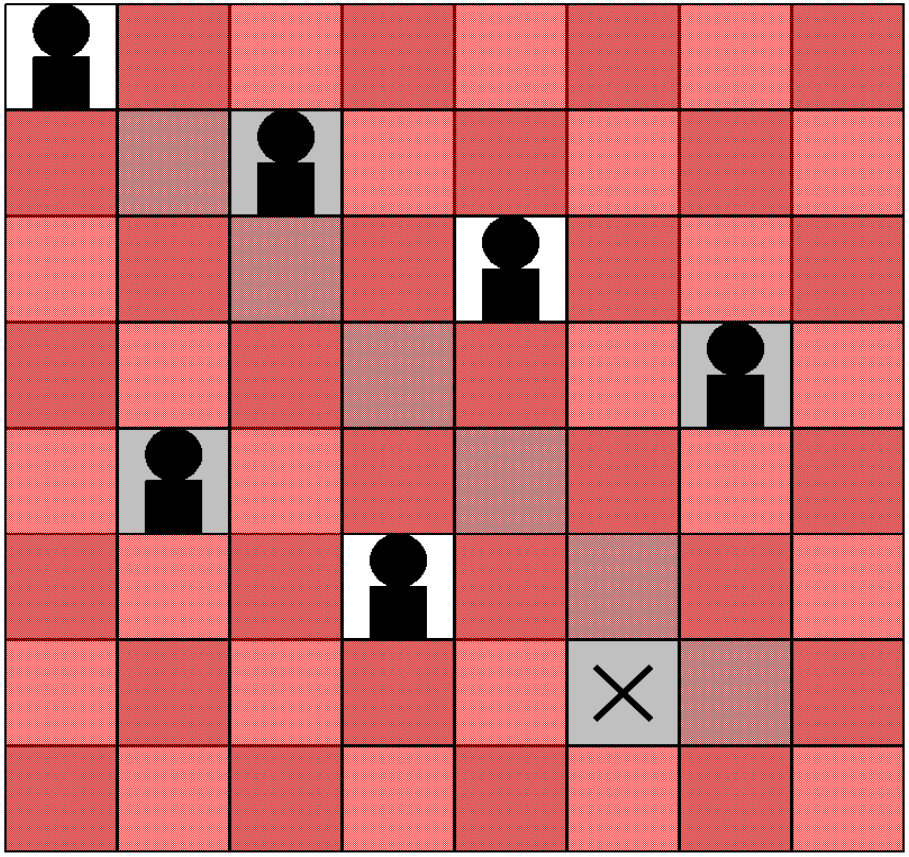
The Eight Queens problem



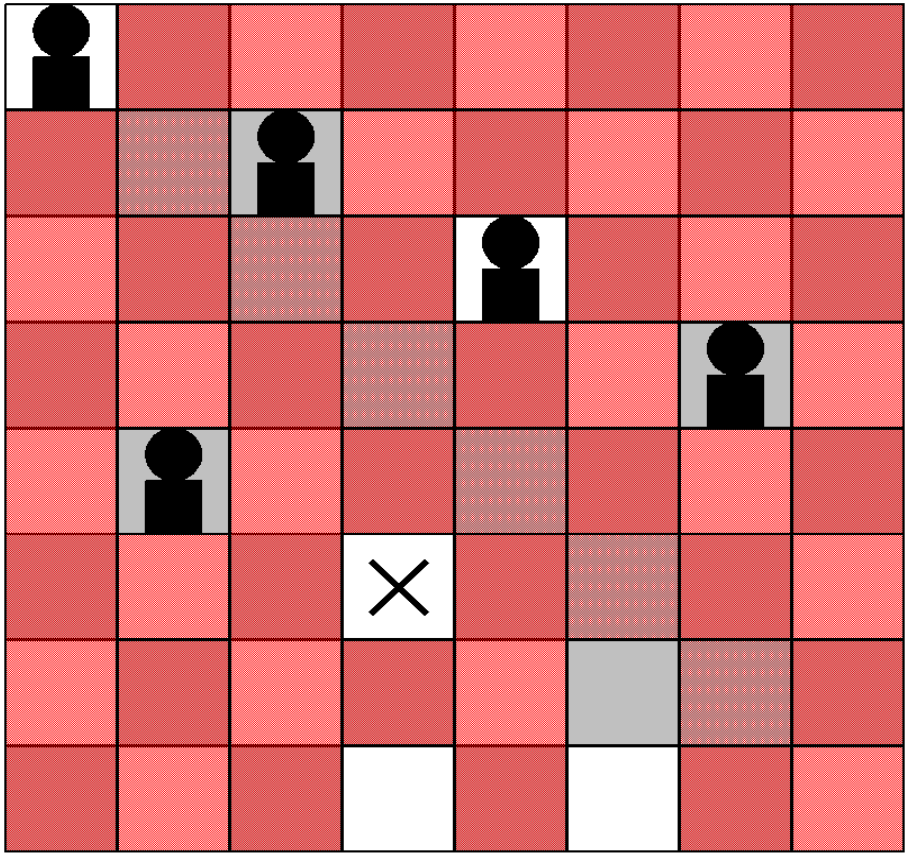
The Eight Queens problem



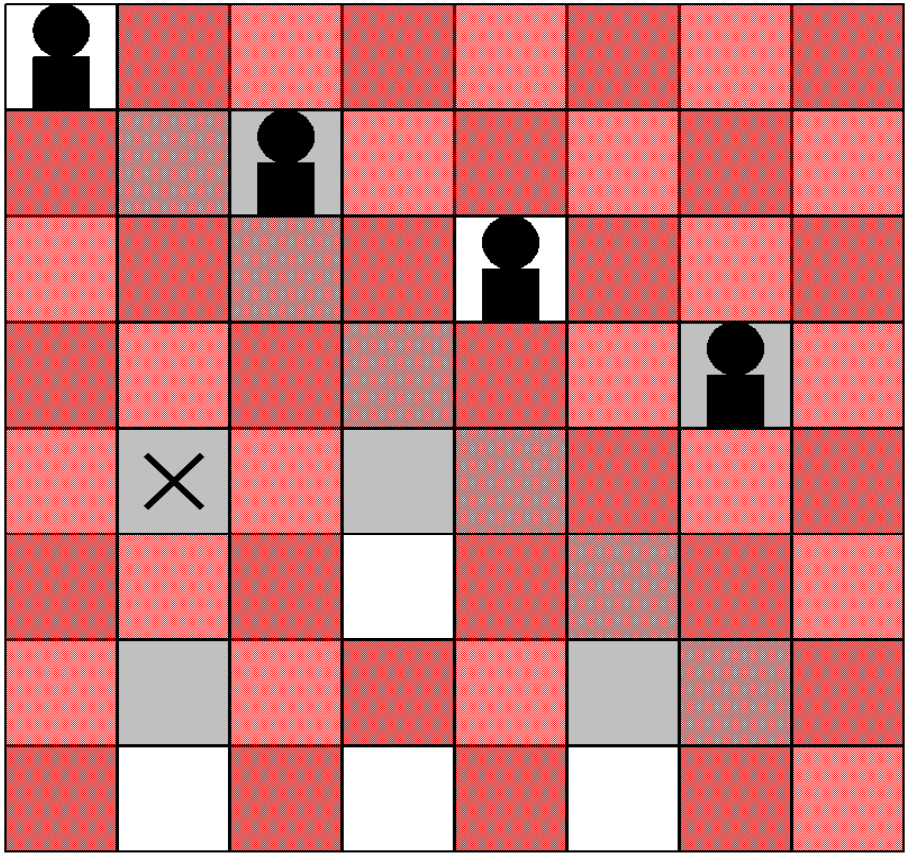
The Eight Queens problem



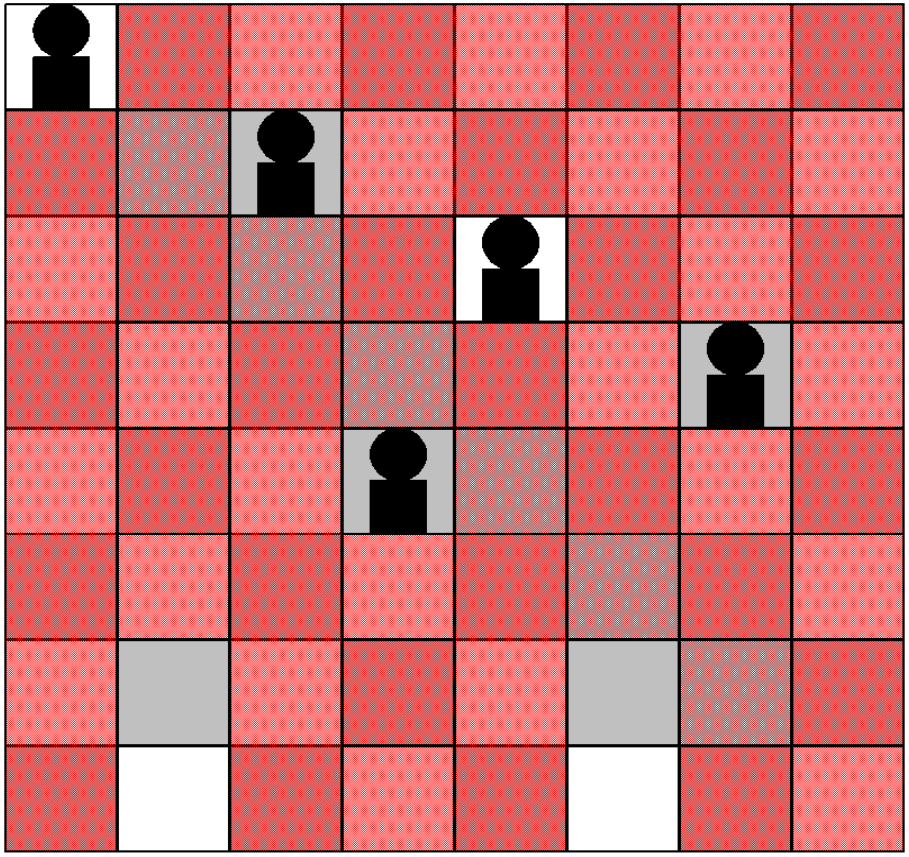
The Eight Queens problem



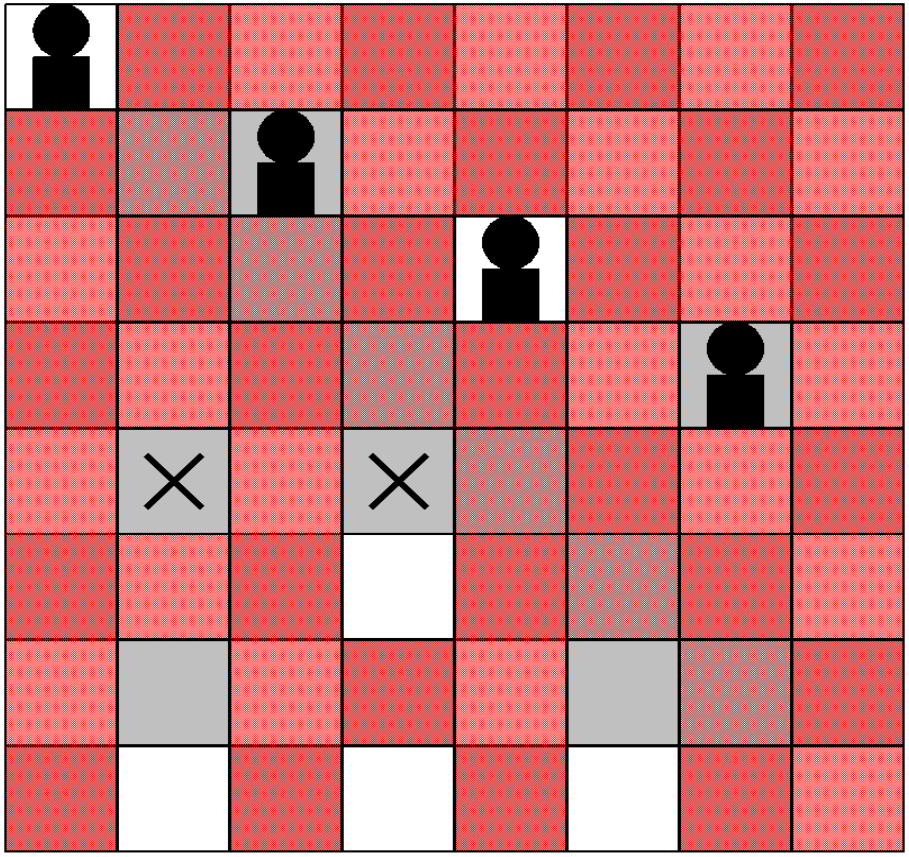
The Eight Queens problem



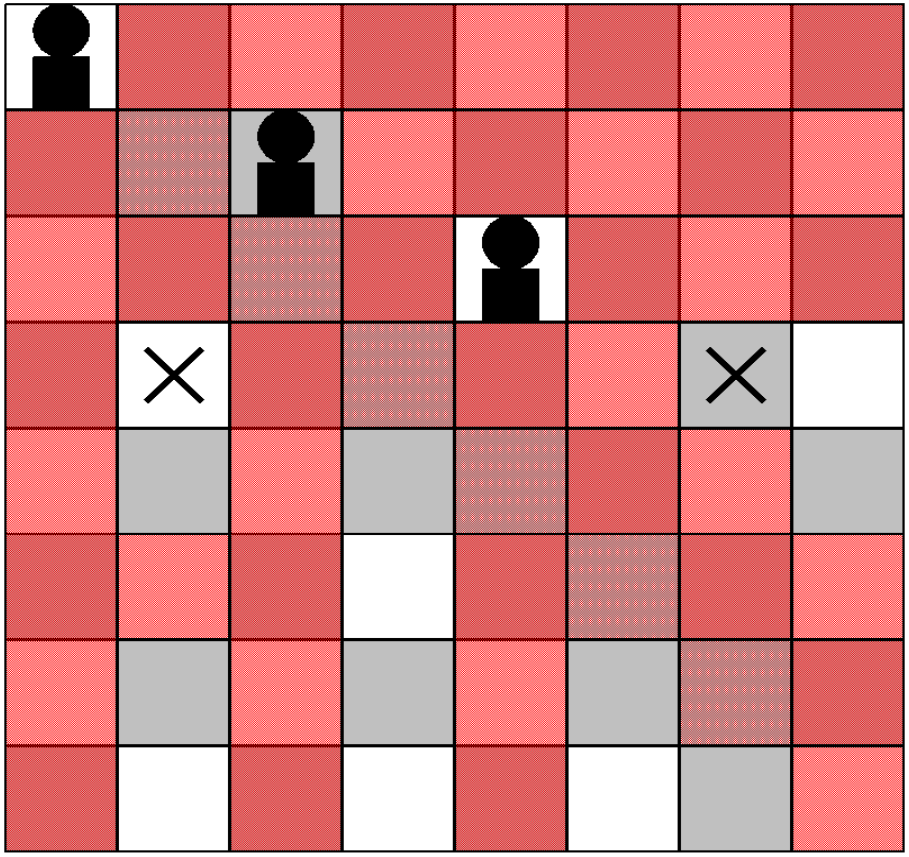
The Eight Queens problem



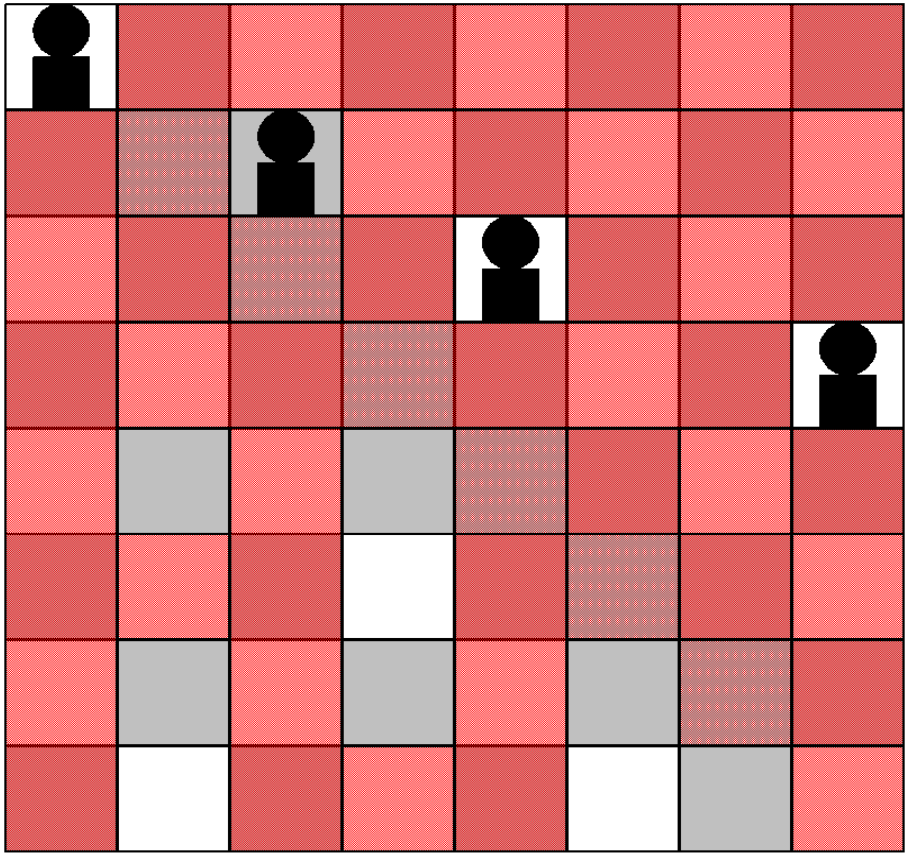
The Eight Queens problem



The Eight Queens problem



The Eight Queens problem



- …etc

The Eight Queens problem

- This approach is far more efficient than the previous ones.
- It can be readily generalised to solve the n queens problem.
- The approach used in this algorithm can be stated as a general approach to problem solving –***backtracking***.

Backtracking

```
Procedure backtrack(v[1 .. k])
   if v is a solution then
      write v
   {else}
      for each (k + 1)-promising vector w[1 .. k + 1]
                              where w[1 .. k] = v[1 .. k] do
                     backtrack(w[1 .. k + 1])
```

– The **`else`** should only be present if a solution cannot be a part of a larger solution.

Branch and bound

The assignment problem

- A set of n agents are assigned n tasks.
- Each agent performs exactly one task.
- If agent i is assigned task *j* then a cost c_{ij} is associated with this combination.
- The problem is to minimise the total cost C.

The assignment problem

– For example suppose agents a, b and c are assigned tasks 1, 2 and 3 with the following cost matrix:

	1	2	3
a	4	7	3
b	2	6	1
c	3	9	4

– If we allot task 1 to agent a, task 2 to agent b and task 3 to agent c the total cost is 4 + 6 + 4 = 14

The assignment problem

– For example suppose agents a, b and c are assigned tasks 1, 2 and 3 with the following cost matrix:

	1	2	3
a	**4**	7	3
b	2	**6**	1
c	3	9	**4**

– If we allot task 1 to agent a, task 2 to agent b and task 3 to agent c the total cost is 4 + 6 + 4 = 14

The assignment problem

– For example suppose agents a, b and c are assigned tasks 1, 2 and 3 with the following cost matrix:

	1	2	3
a	4	**7**	3
b	2	6	**1**
c	**3**	9	4

– If we allot task 1 to agent c, task 2 to agent a and task 3 to agent b, the total cost is 3 + 7 + 1 = 11

The assignment problem

A company is assembling a team to carry out a series of operations. There are 4 members of the team: A, B, C and D, and 4 operations to be carried out. Each team member can carry out exactly one operation. All 4 operations must be carried out successfully for the overall project to succeed, however the probability of a particular team member succeeding in a particular operation varies, as shown in the table below.

		Operation			
		1	2	3	4
Team member	A	0.9	0.8	0.9	0.85
	B	0.7	0.6	0.8	0.7
	C	0.85	0.7	0.85	0.8
	D	0.75	0.7	0.75	0.7

If the team members were assigned to operations in the order ABCD, then the overall probability of successful completion of the project is (0.9)(0.6)(0.85)(0.7) = 0.3213.

The assignment problem

A company is assembling a team to carry out a series of operations. There are 4 members of the team: A, B, C and D, and 4 operations to be carried out. Each team member can carry out exactly one operation. All 4 operations must be carried out successfully for the overall project to succeed, however the probability of a particular team member succeeding in a particular operation varies, as shown in the table below.

		Operation			
		1	2	3	4
Team member	A	0.9	0.8	0.9	0.85
	B	0.7	0.6	0.8	0.7
	C	0.85	0.7	0.85	0.8
	D	0.75	0.7	0.75	0.7

If the team members were assigned to operations in the order ABCD, then the overall probability of successful completion of the project is (0.9)(0.7)(0.85)(0.7) = 0.37485.

The assignment problem

- The assignment problem has many applications:
  - Buildings and sites –the cost of erecting building i on site j.
  - Trucks and destinations –the cost of sending truck *i* to destination *j*.
  - etc.
- In general, with *n* agents and *n* jobs there are *n*! possible assignments.
- This is too many to consider, even for relatively small values of *n*.

The assignment problem

– Suppose we have to solve the following instance

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

– Note the assignment a $\Rightarrow$1, b $\Rightarrow$ 2, c $\Rightarrow$ 3, d $\Rightarrow$ 4 has cost 73.

The assignment problem

- This cost of 73 provides an upper bound on the solution.
- The minimal cost must be ≤ 73
- Can we get a lower bound?

The assignment problem

– This cost of 73 provides an upper bound on the solution.

– The minimal cost must be ≤73

–Can we get a lower bound?

- Task 1 must cost at least 11

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

The assignment problem

– This cost of 73 provides an upper bound on the solution.

– The minimal cost must be ≤ 73

–Can we get a lower bound?

- Task 1 must cost at least 11
- Task 2 must cost at least 12

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

The assignment problem

– This cost of 73 provides an upper bound on the solution.

– The minimal cost must be ≤ 73

–Can we get a lower bound?

- Task 1 must cost at least 11
- Task 2 must cost at least 12
- Task 3 must cost at least 13

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

The assignment problem

– This cost of 73 provides an upper bound on the solution.

– The minimal cost must be ≤ 73

–Can we get a lower bound?

- Task 1 must cost at least 11
- Task 2 must cost at least 12
- Task 3 must cost at least 13
- Task 4 must cost at least 22

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

The assignment problem

– This cost of 73 provides an upper bound on the solution.

– The minimal cost must be ≤ 73

–Can we get a lower bound?

- Task 1 must cost at least 11
- Task 2 must cost at least 12
- Task 3 must cost at least 13
- Task 4 must cost at least 22

– The minimal cost must be ≥58

– 58 ≤C ≤73

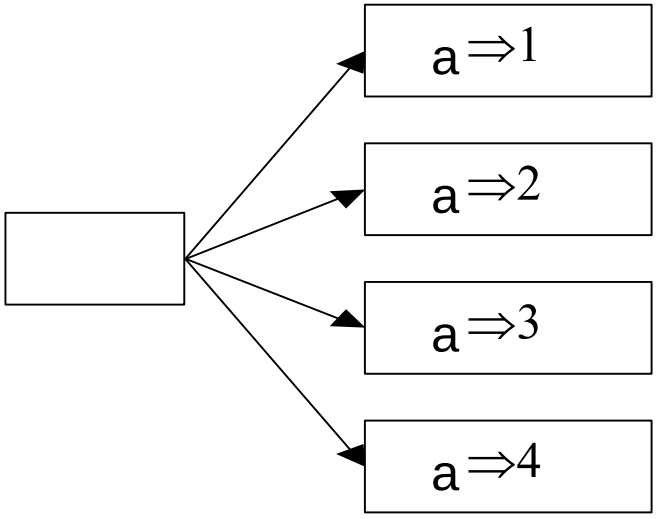
	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

The assignment problem

– The solution proceeds as follows:
  • We explore a tree whose nodes correspond to partial assignments.
  • At the root of the tree, no assignments have been made.
  • At each level we fix the assignment of one more agent.
  • At each node we calculate a bound on the costs that can be achieved by completing this sub tree.
  • We prune any sub tree with too high a minimum cost bound.

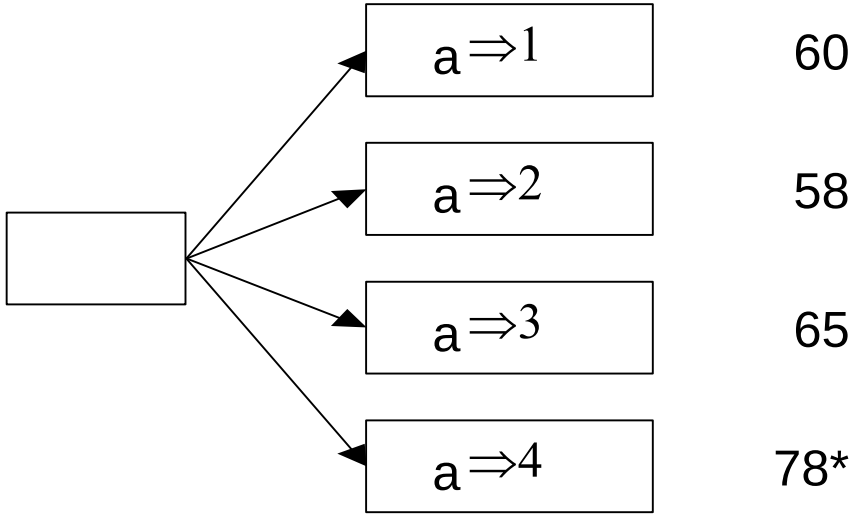
The assignment problem: 58≤C ≤73

– Assign agent a



The assignment problem: 58≤C ≤73

– Assign agent a, calculate lower bounds from here
– a⇒4 is clearly not a possible solution



	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

	1	2	3	4
a				
b		15	13	22
c		17	19	23
d		14	20	28

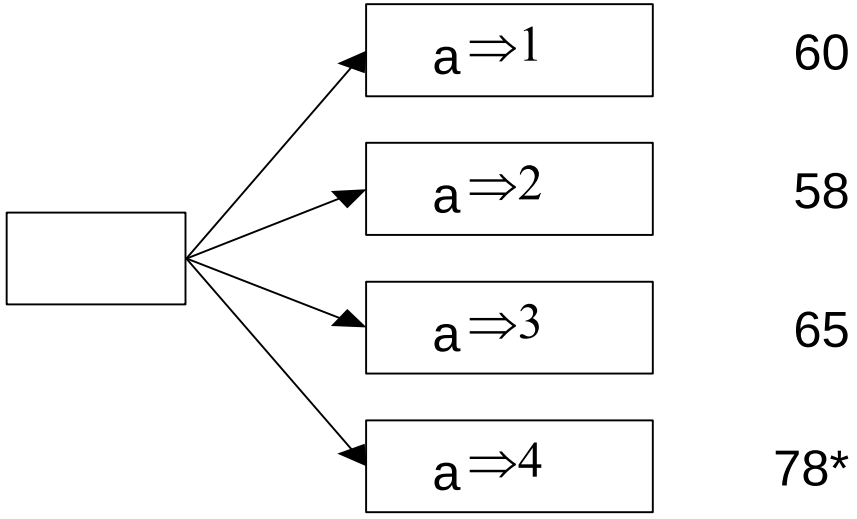
	1	2	3	4
a				
b	14		13	22
c	11		19	23
d	17		20	28

	1	2	3	4
a				
b	14	15		22
c	11	17		23
d	17	14		28

	1	2	3	4
a				
b	14	15	13	
c	11	17	19	
d	17	14	20	

The assignment problem: 58≤C ≤73

– Assign agent a, calculate lower bounds from here
– A ⇒4 is clearly not a possible solution



	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

	1	2	3	4
a	11			
b		15	13	22
c		17	19	23
d		14	20	28

14 13 22

	1	2	3	4
a		12		
b	14		13	22
c	11		19	23
d	17		20	28

11 13 22

	1	2	3	4
a			18	
b	14	15		22
c	11	17		23
d	17	14		28

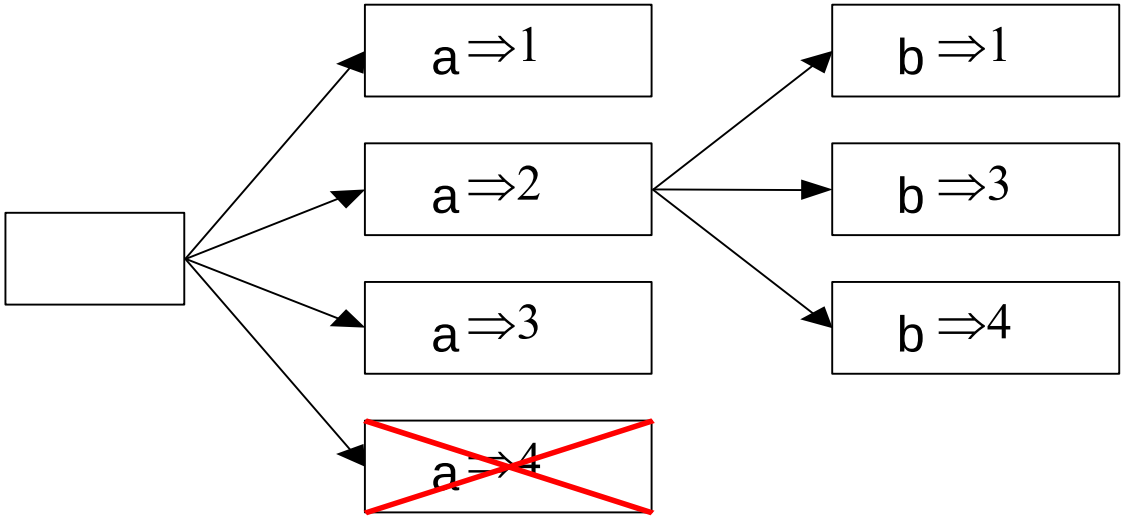
11 14 22

	1	2	3	4
a				40
b	14	15	13	
c	11	17	19	
d	17	14	20	

11 14 13

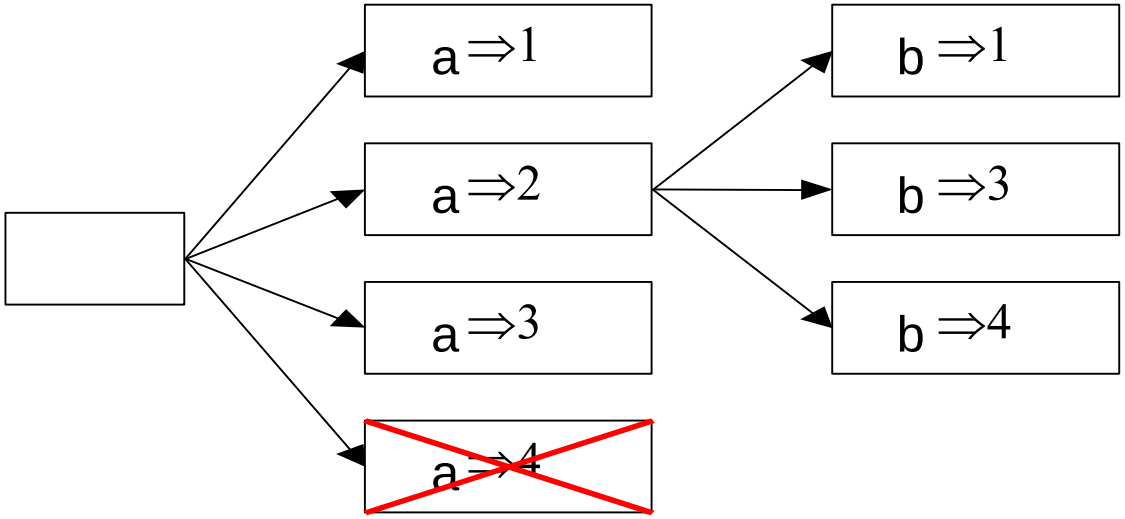
The assignment problem: 58≤C ≤73

– Eliminate a ⇒4, try a ⇒2



The assignment problem: 58≤C ≤73

– Eliminate a ⇒4, try a ⇒2, calculate lower bounds



68

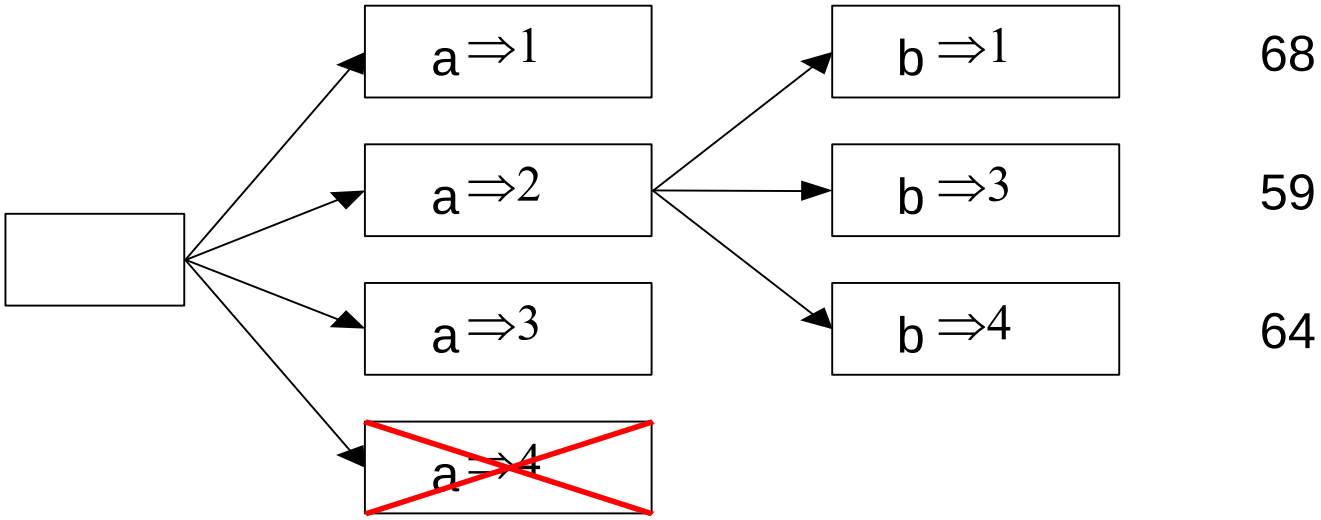
59

64

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

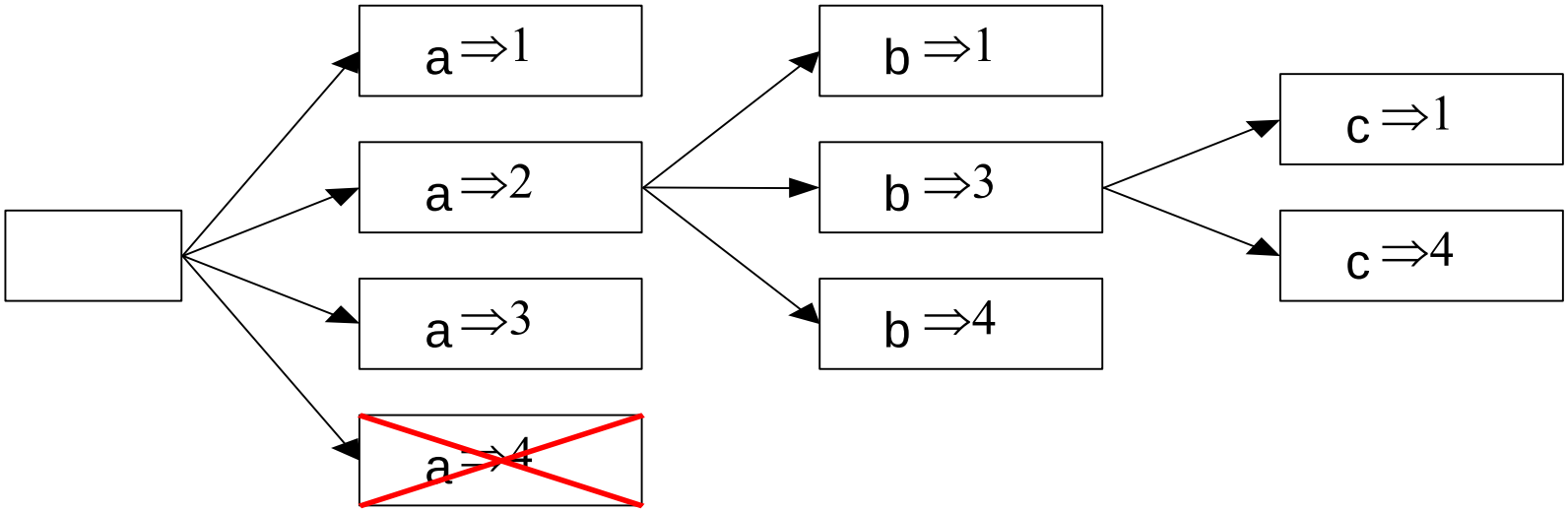
The assignment problem: 58≤C ≤73

– Eliminate a ⇒4, try a ⇒2, calculate lower bounds



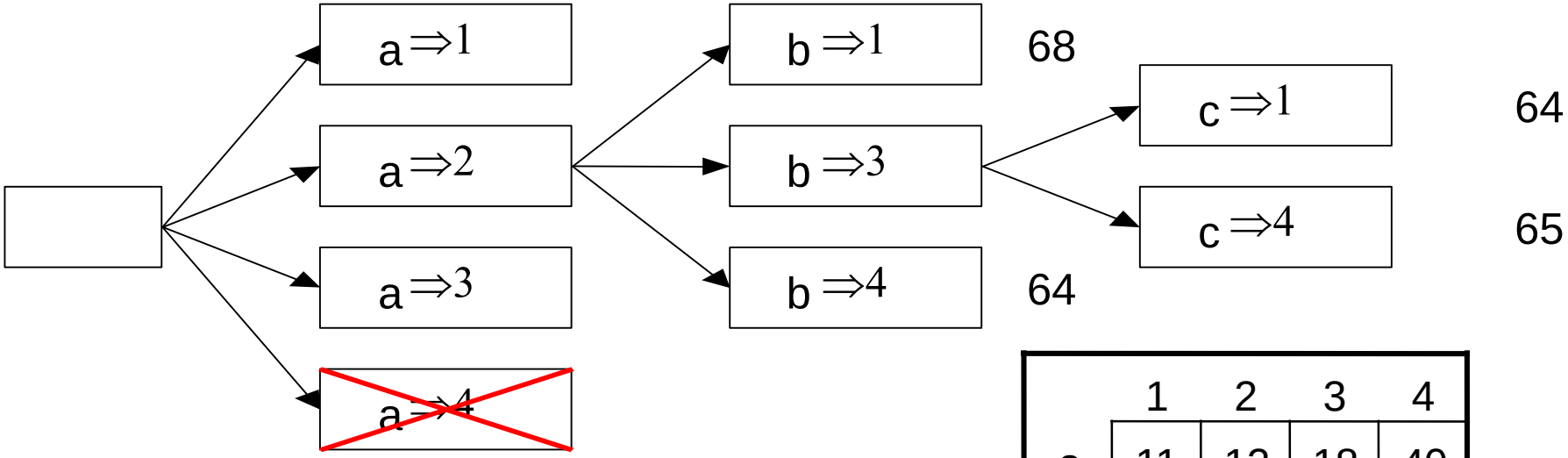
– Proceed with a⇒2, b ⇒ 3

The assignment problem: 58≤C ≤73



The assignment problem: 58≤C ≤73

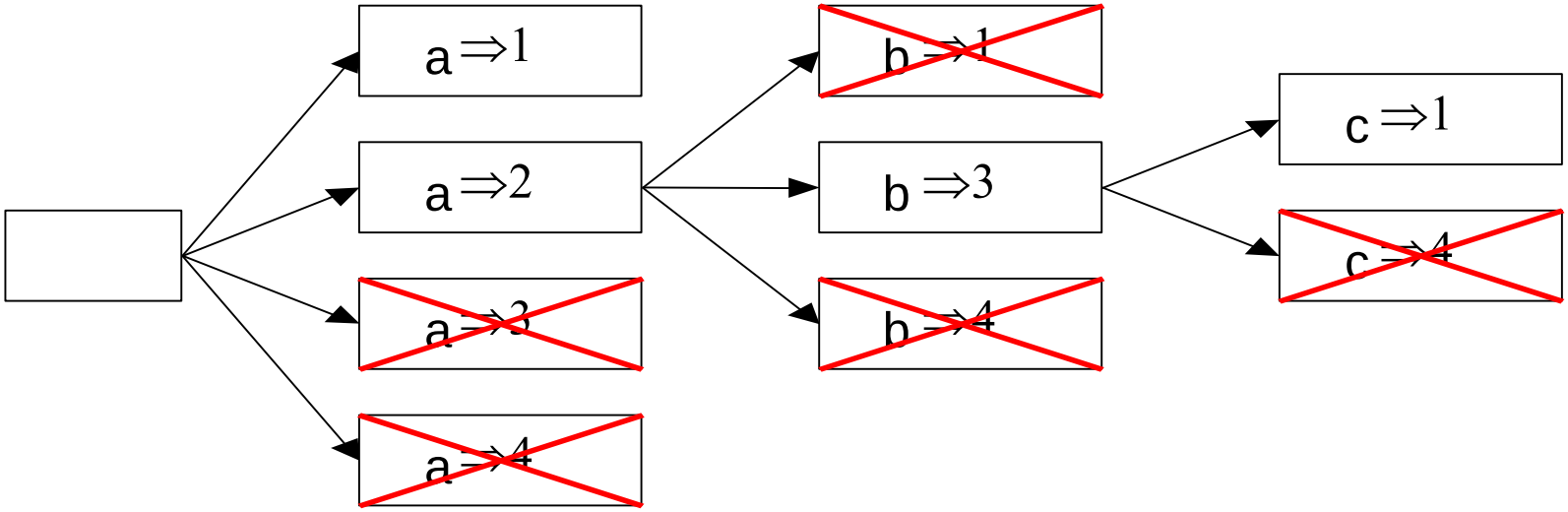
– These are final costs so 64 is a new upper bound



	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

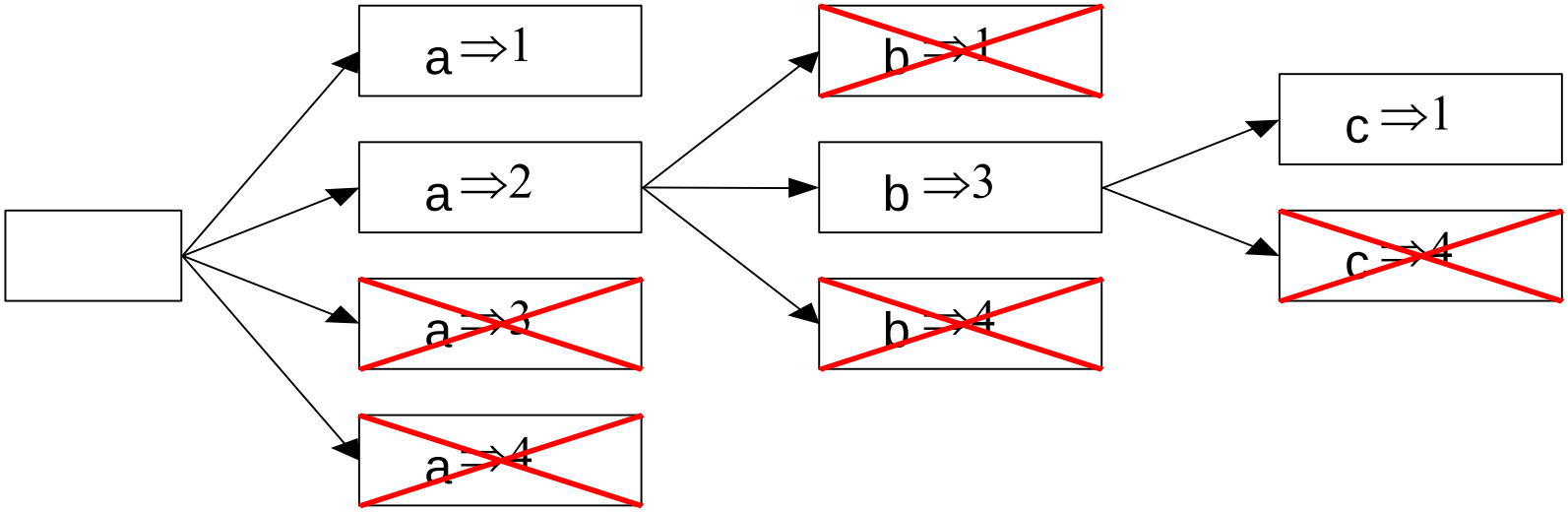
The assignment problem: 58≤C ≤64

– Eliminate any sub tree with min cost > 64



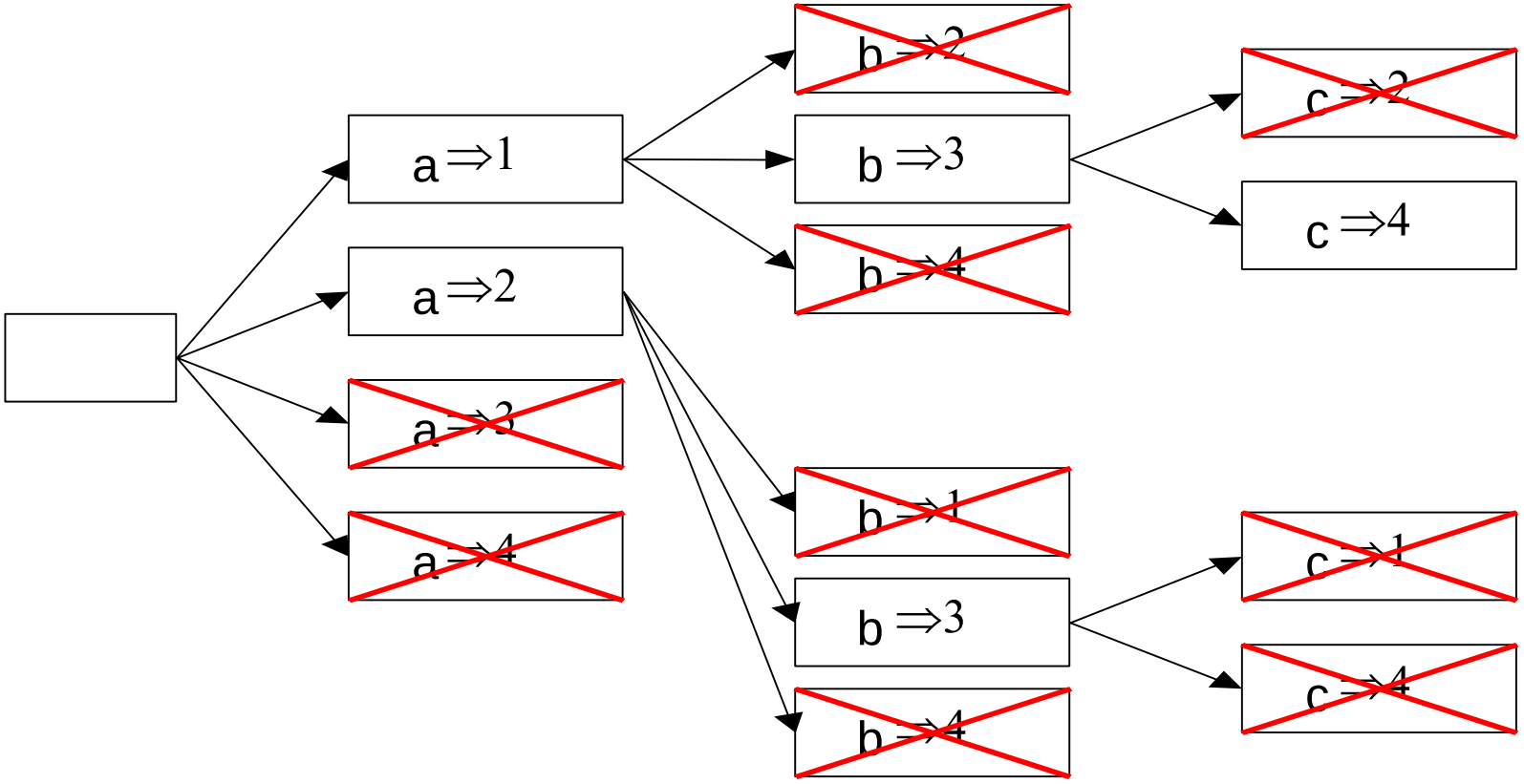
The assignment problem: 58≤C ≤64

– Node a = 1 is the only one left worth exploring



– After two more steps we get to…

The assignment problem: 58≤C ≤64



61

– With a solution a=1, b=3, c=4, d=2; cost = 61

The assignment problem

Select one element in each row of the cost matrix *C* so that:

- **no two selected elements are in the same column; and**
- **the sum is minimized**

For example:

	Job 1	*Job* 2	*Job* 3	*Job* 4
Person *a*	9	2	7	8
Person *b*	6	4	3	7
Person *c*	5	8	1	8
Person *d*	7	6	9	4

Lower bound: Any solution to this problem will have total cost of at least:

Assignment problem: lower bounds

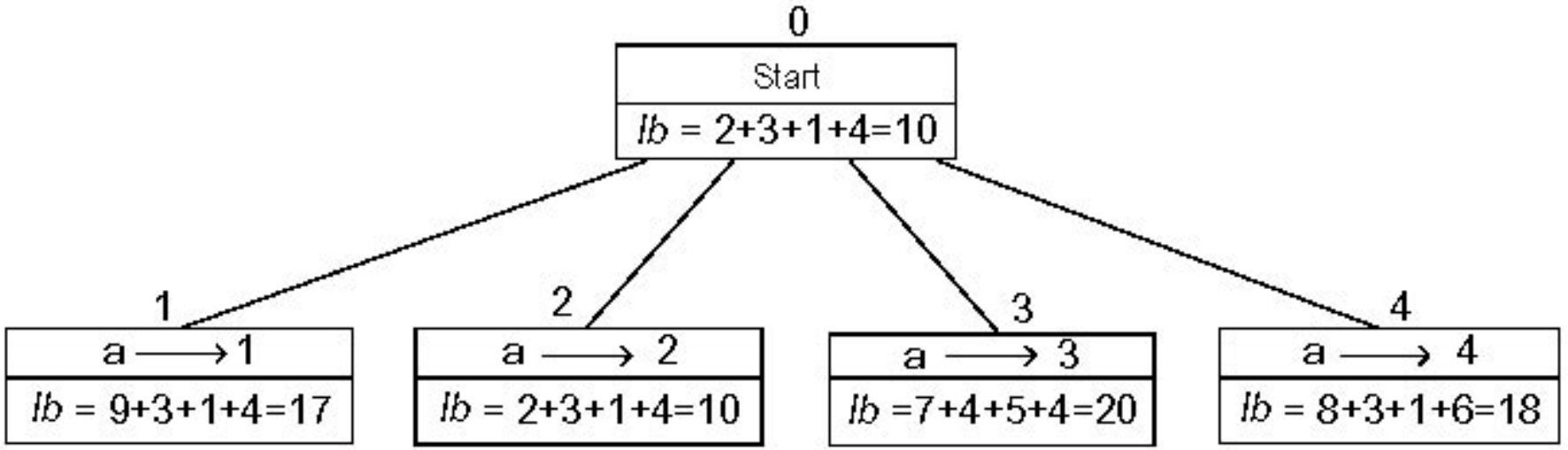


Figure 11.5 Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person a and the lower bound value, lb, for this node.

State-space levels 0, 1, 2

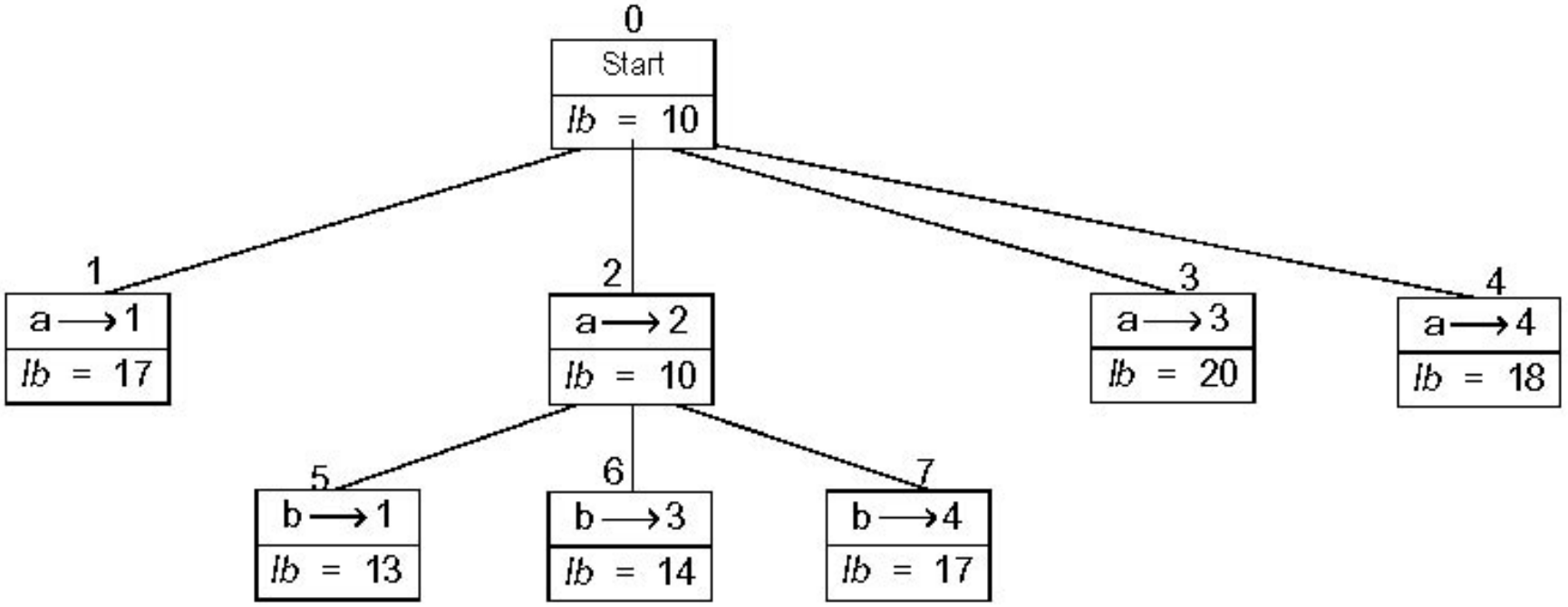


Figure 11.6 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm

Complete state-space

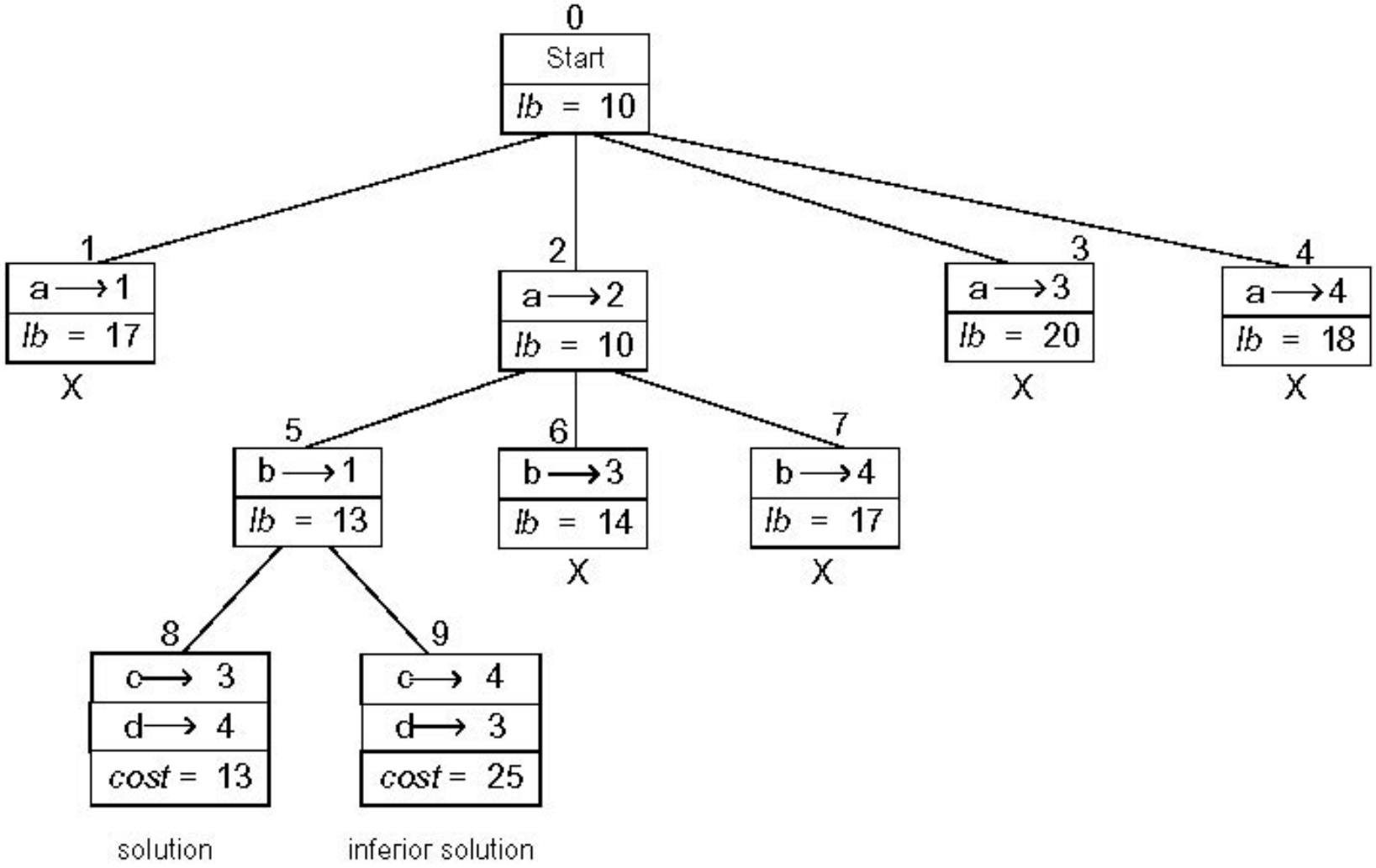


Figure 11.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm

Branch and bound

– The basic idea behind branch and bound algorithms is shown in the preceding example:

•Proceed to explore the sample space of possible solutions.

•At each stage, prune off any part of the sample space that cannot lead to a solution better than the best one found so far.

•If we have not found any solution so far we can still prune off any part of the sample space for which the best possible solution is worse than the worst possible solution.

Depth- vs. Breadth-First

– In the example, we follow the most promising path to its end before we look at other branches.
– This is called a ***depth first*** strategy.
– If we look at all of the branches and select the most promising at each step, perhaps jumping from branch to branch, we have what is called a ***breadth first*** strategy.
– Generally, the breadth first strategy will find a solution more quickly (it will require less of the solution space to be explored) but will involve more housekeeping for the program.

Backtracking vs Branch-and-bound

- Similarity
  - – Construct state-space tree whose nodes reflect specific choices
  - – Cut off a branch if it cannot lead to a solution
- Difference
  - – Branch-and-bound:
    - Optimization problem----–Compute a bound on possible values of the objective function
    - Depth-first or Breadth-first rule to generate state-space tree
    - Generate all the children of the most promising node
    - Completely searches the state-space tree to find the optimal solution
  - – Backtracking:
    - Non-optimization problem----– Compute a feasibility function
    - Depth-first rule to generate state-space tree
    - Generate a single child of the last promising node
    - Searches the state-space tree until it finds a solution

Discussions

1. What type of problems can be solved by branch and bound strategy?
2. Can Traveling Salesman Problem be solved by branch and bound strategy?

Homework

1. Implement the backtracking algorithm for solving the eight queen problem. (Require to print a solution).
2. Write the algorithm for solving the assignment problem by using branch and bound strategy.