NUS | Computing
National University of Singapore

CS3243 Tutorial 5
Constraint Satisfaction Problems

Gu Zhenhao

March 1, 2023

Notes

- **Project 2** coming up!
- No tutorial assignment next week :-D, but there's still a tutorial.
- Don't worry too much about the Midterm exam.
  - I cannot finish it in 1 hour either :-)
  - The ability to apply the algorithms you learned is more important than exam results.

Local Search

How to find the next best state?

Tutorial 4, Problem 3

$f(\text{state}) = 4.$

2	3	
1	8	4
7	6	5

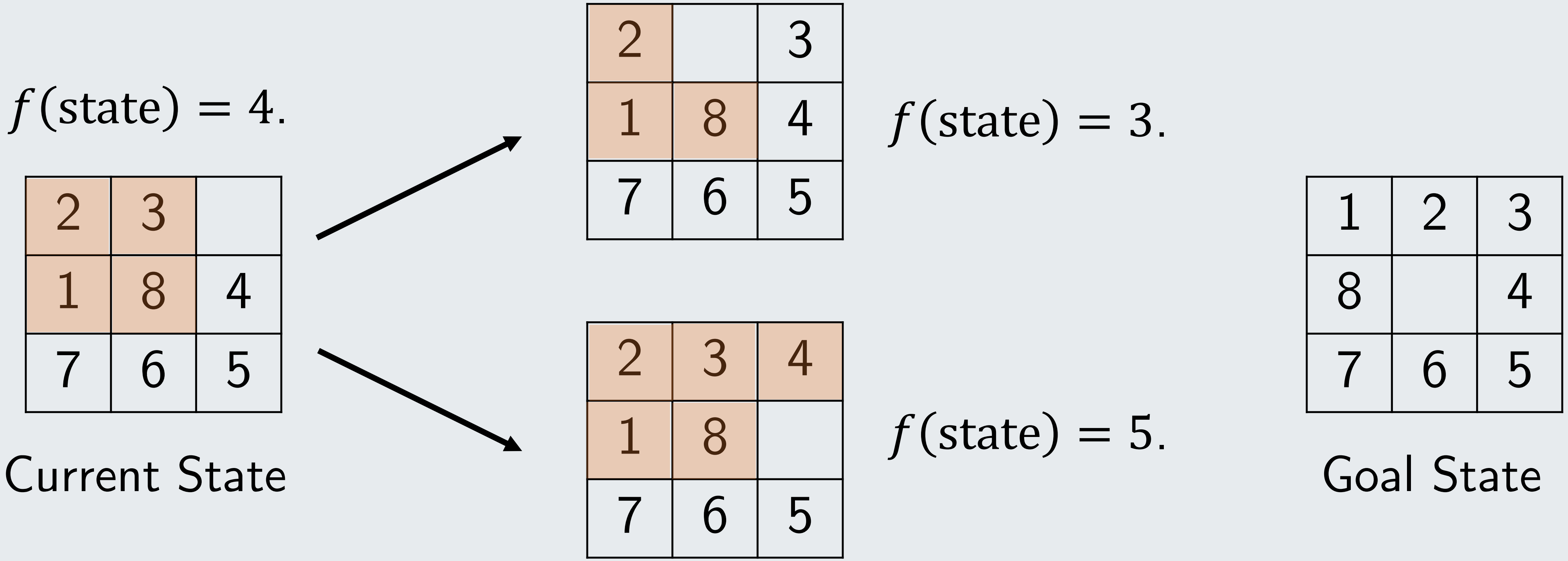
Current State

1	2	3
8		4
7	6	5

Goal State

$f(\text{state}) = \text{number of mismatched tiles.}$

Tutorial 4, Problem 3



$f(\text{state}) = \text{number of mismatched tiles.}$

Building CSPs

How to formulate Constraint Satisfaction Problems?

Building Constraint Satisfaction Problems

Constraint Satisfaction Problem is a special case of search problem, where

- **State**: A set of variables, each assigned a value,
  - **Initial State**: No assignment to any variable,
  - **Goal State**: each variable is assigned a value, satisfying all ***constraints***.
- **Action**: assigning a value to a variable.

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Building Constraint Satisfaction Problems

Constraint Satisfaction Problem is defined by 3 components:

- A set of **variables**, $X = \{X_1, \dots, X_n\}$,
- A set of **domains**, $D = \{D_1, \dots, D_n\}$.
- A set of **constraints** C that specify allowable combination of values of the variables.

Note. C should be expressed as formulas depending on $X_1, \dots, X_n$.

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Problem 2

Goal: Assign one professor to each course. Each professor can teach one class at a time.

Classes:

- C_1: 8:00 AM – 9:00 AM.
- C_2: 8:30 AM – 9:30 AM.
- C_3: 9:00 AM – 10:00 AM.
- C_4: 9:00 AM – 10:00 AM.
- C_5: 9:30 AM – 10:30 AM.

Professors:

- Prof T: available for C_3, C_4.
- Prof J: available for C_2, C_3, C_4, C_5.
- Prof B: available for C_1, C_2, C_3, C_4, C_5.

Variable	Domain
C_1	
C_2	
C_3	
C_4	
C_5	

Constraints:

Problem 2

Goal: Assign one professor to each course. Each professor can teach one class at a time.

Classes:

- C_1: 8:00 AM – 9:00 AM.
- C_2: 8:30 AM – 9:30 AM.
- C_3: 9:00 AM – 10:00 AM.
- C_4: 9:00 AM – 10:00 AM.
- C_5: 9:30 AM – 10:30 AM.

Professors:

- **Prof T**: available for C_3, C_4.
- **Prof J**: available for C_2, C_3, C_4, C_5.
- **Prof B**: available for C_1, C_2, C_3, C_4, C_5.

Variable	Domain
C_1	$\{B\}$
C_2	$\{J, B\}$
C_3	$\{T, J, B\}$
C_4	$\{T, J, B\}$
C_5	$\{J, B\}$

Constraints:

- $C_1 \neq C_2$.
- $C_3 \neq C_4$.
- $C_2 \neq C_4$.
- $C_2 \neq C_3$.
- $C_4 \neq C_5$.
- $C_3 \neq C_5$.

Possible Assignment:

$C_1 = B,\ C_2 = J,\ C_3 = B,\ C_4 = T,\ C_5 = J.$

Problem 3.a

- A set of people $N = \{1, \dots, n\}$, and a set of items $G = \{g_1, \dots, g_m\}$.
- **Constraints**: each person is assigned at most one item, and each item is assigned to at most one person.

Define:

$$x_{i,j} = \begin{cases} 1 & \text{if item } j \text{ is assigned to person } i \\ 0 & \text{otherwise} \end{cases}$$

then the constraints can be written as

$$\text{for each person } i\text{: } \sum_{j=1}^{m} x_{i,j} \leq 1, \qquad \text{for each item } j\text{: } \sum_{i=1}^{n} x_{i,j} \leq 1$$

Problem 3.b

- A set of people $N = \{1, \dots, n\}$, and a set of items $G = \{g_1, \dots, g_m\}$.
- Divide N into k groups $N_1, \dots, N_k$, and divide G into l groups $G_1, \dots, G_l$.
- **Constraints**: each group of people N_p can take $\leq \lambda_{pq}$ items from group G_q.

Define:

$$x_{i,j} = \begin{cases} 1 & \text{if item } j \text{ is assigned to person } i \\ 0 & \text{otherwise} \end{cases}$$

then the constraints can be written as

$$\text{for all } p, q\text{: } \sum_{i \in N_p} \sum_{j \in G_q} x_{i,j} \leq \lambda_{pq}$$

Problem 3.c

- A set of people $N = \{1, \dots, n\}$, and a set of items $G = \{g_1, \dots, g_m\}$.
- Each people i has a function $u_i(g_j) > 0$ that shows how i prefers item g_j.
- Suppose i is assigned item g_j, i' is assigned item $g_{j'}$, then i ***envies*** i' if $u_i(g_j) < u_i(g_{j'})$.
- **Constraints**: No people envies any other people.

The constraint can be written as

$$\text{for all } i, i' \in N: \quad \text{if } x_{i,j} = 1 \text{ and } x_{i',j'} = 1, \text{then } u_i(g_j) \geq u_i(g_{j'})$$

Problem 3.c

- A set of people $N = \{1, \dots, n\}$, and a set of items $G = \{g_1, \dots, g_m\}$.
- Each people i has a function $u_i(g_j) > 0$ that shows how i prefers item g_j.
- Suppose i is assigned item g_j, i' is assigned item $g_{j'}$, then i ***envies*** i' if $u_i(g_j) < u_i(g_{j'})$.
- **Constraints**: No people envies any other people.

The constraint can be written as

$$\text{for all } i, i' \in N: \quad (x_{i,j} = 1 \wedge x_{i',j'} = 1) \Rightarrow u_i(g_j) \geq u_i(g_{j'})$$

Problem 3.c

- A set of people $N = \{1, \ldots, n\}$, and a set of items $G = \{g_1, \ldots, g_m\}$.
- Each people i has a function $u_i(g_j) > 0$ that shows how i prefers item g_j.
- Suppose i is assigned item g_j, i' is assigned item $g_{j'}$, then i ***envies*** i' if $u_i(g_j) < u_i(g_{j'})$.
- **Constraints**: No people envies any other people.

The constraint can be written as

$$\text{for all } i \in N \text{ and all other items } j': \quad \sum_{i' \in N \setminus \{i\}} x_{i',j'} \geq 1 \Rightarrow u_i(g_j) \geq u_i(g_{j'})$$

Solving CSPs

How to find solutions to Constraint Satisfaction Problems?

Backtracking Search

Intuition: **DFS**, but (i) has early failure detection, and (ii) implemented in a recursive manner.

function BACKTRACKING-SEARCH(*csp*) **returns** a solution, or failure
    return BACKTRACK({ }, *csp*)

function BACKTRACK(*assignment*, *csp*) **returns** a solution, or failure
    if *assignment* is complete **then return** *assignment*
    var ← SELECT-UNASSIGNED-VARIABLE(*csp*)
    for each *value* **in** ORDER-DOMAIN-VALUES(*var*, *assignment*, *csp*) **do**
        if *value* is consistent with *assignment* **then**
            add {*var* = *value*} to *assignment*
            inferences ← INFERENCE(*csp*, *var*, *value*)
            if *inferences* ≠ *failure* **then**
                add *inferences* to *assignment*
                result ← BACKTRACK(*assignment*, *csp*)
                if *result* ≠ *failure* **then**
                    return *result*
        remove {*var* = *value*} and *inferences* from *assignment*
    return *failure*

Try another path if we **detect** failure.

Otherwise, continue exploring current path.

Variable Ordering

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
   return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment then
         add {var = value} to assignment
         inferences ← INFERENCE(csp, var, value)
         if inferences ≠ failure then
            add inferences to assignment
            result ← BACKTRACK(assignment, csp)
            if result ≠ failure then
               return result
      remove {var = value} and inferences from assignment
   return failure
```

Minimum-remaining-values (MRV) **heuristic**: choose the variable with fewest legal values left in domain.

Intuition: variables that are most "constrained" are likely to cause failure.

Effects:

- Detect failure faster.
- Avoid pointless search on other variables if this path is likely to fail.

Variable Ordering

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
     if value is consistent with assignment then
        add {var = value} to assignment
        inferences ← INFERENCE(csp, var, value)
        if inferences ≠ failure then
          add inferences to assignment
          result ← BACKTRACK(assignment, csp)
          if result ≠ failure then
            return result
     remove {var = value} and inferences from assignment
  return failure
```

***Degree* heuristic**: choose the variable involved in the most constraints on other unassigned variables.

Intuition: Assigning values to high-degree variable eliminates a lot of values in other domains.

Effects:

- Reduce the branching factor.

Common practice: Use MRV to select variables, and the degree heuristic as tiebreaker.

Value Ordering

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
     if value is consistent with assignment then
        add {var = value} to assignment
        inferences ← INFERENCE(csp, var, value)
        if inferences ≠ failure then
           add inferences to assignment
           result ← BACKTRACK(assignment, csp)
           if result ≠ failure then
              return result
     remove {var = value} and inferences from assignment
  return failure
```

Least-constraining-value **heuristic**: choose the value that rules out fewest values in domain of neighbouring variables.

Intuition: Leave maximum flexibility for subsequent assignments.

Effects:

- Minimizes chances to fail.

Question: Why do we choose variables to detect failures faster, and choose values that avoids failure?

Inference

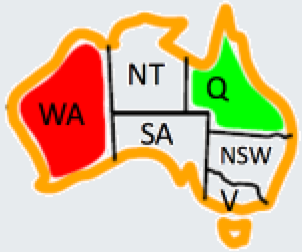
```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
   return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment then
         add {var = value} to assignment
         inferences ← INFERENCE(csp, var, value)
         if inferences ≠ failure then
            add inferences to assignment
            result ← BACKTRACK(assignment, csp)
            if result ≠ failure then
               return result
      remove {var = value} and inferences from assignment
   return failure
```

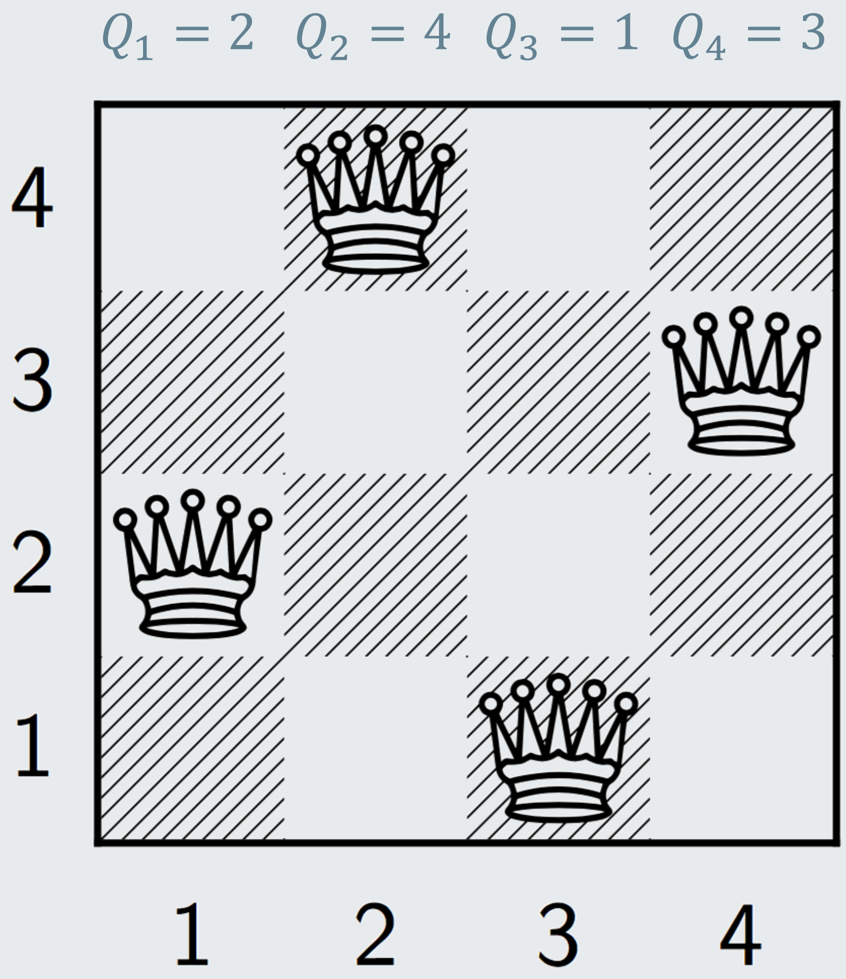
Forward checking: check if current assignment will eliminate all candidate values in unassigned variables.

Effects:

- Detects failures faster.
- Avoid exploring false paths.



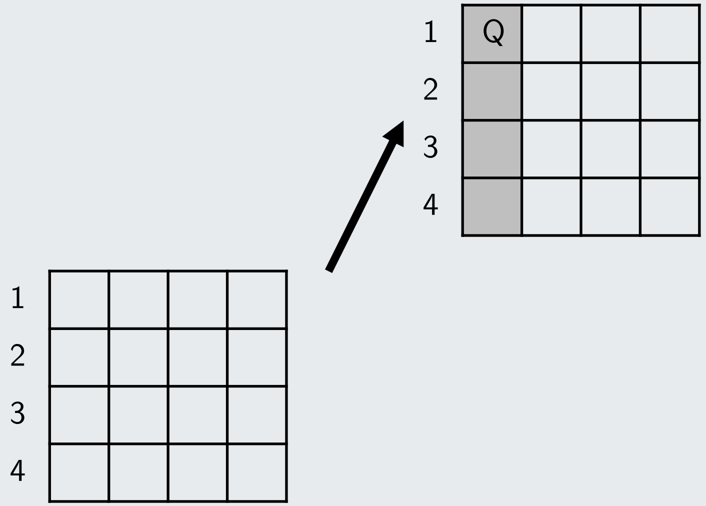
Question 1



- **Variables**: 4 queens, Q_1, Q_2, Q_3 and Q_4, where queen Q_i is on the i-th column.
- **Domains**: Each queen can be assigned to one of the rows $\{1,2,3,4\}$.
- **Constraints**: No queens are threatening each other.

Assign variables in the order Q_1, Q_2, Q_3, Q_4.

Problem 1



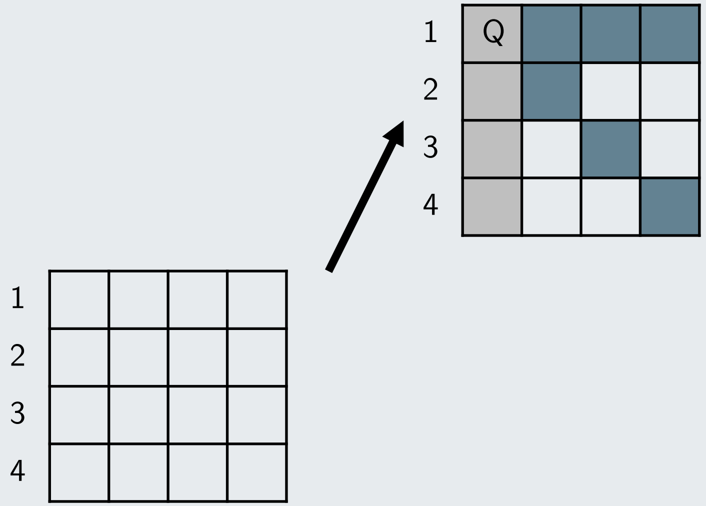
Assigned Variable

Eliminated Values

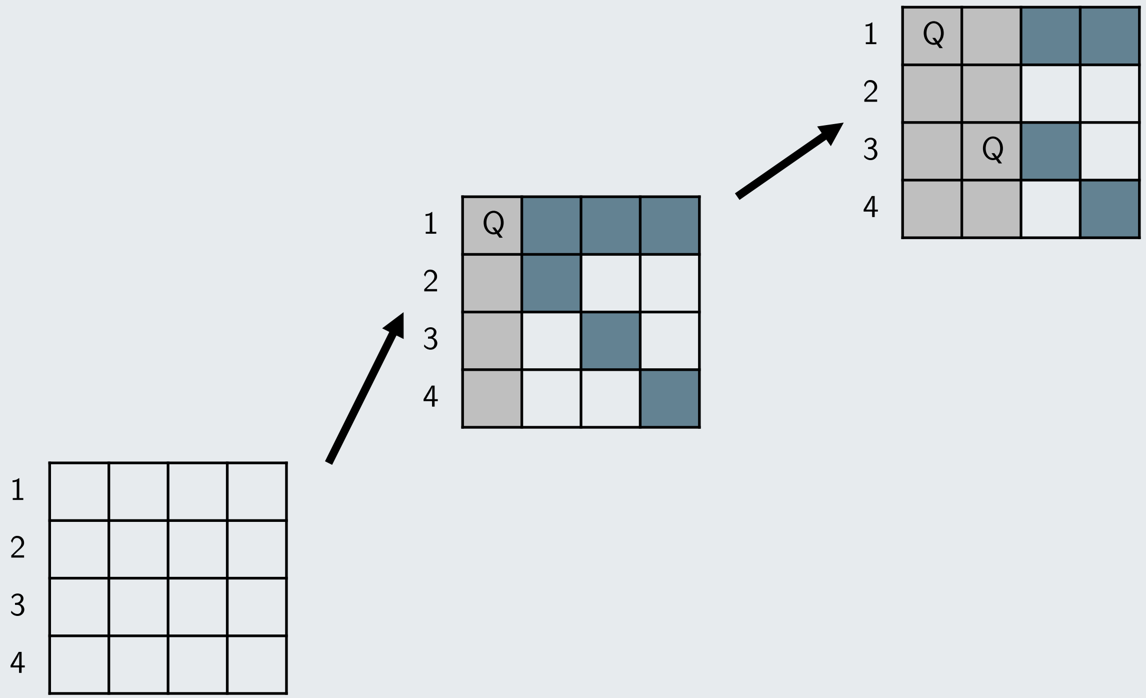
Problem 1

Assigned Variable

Eliminated Values



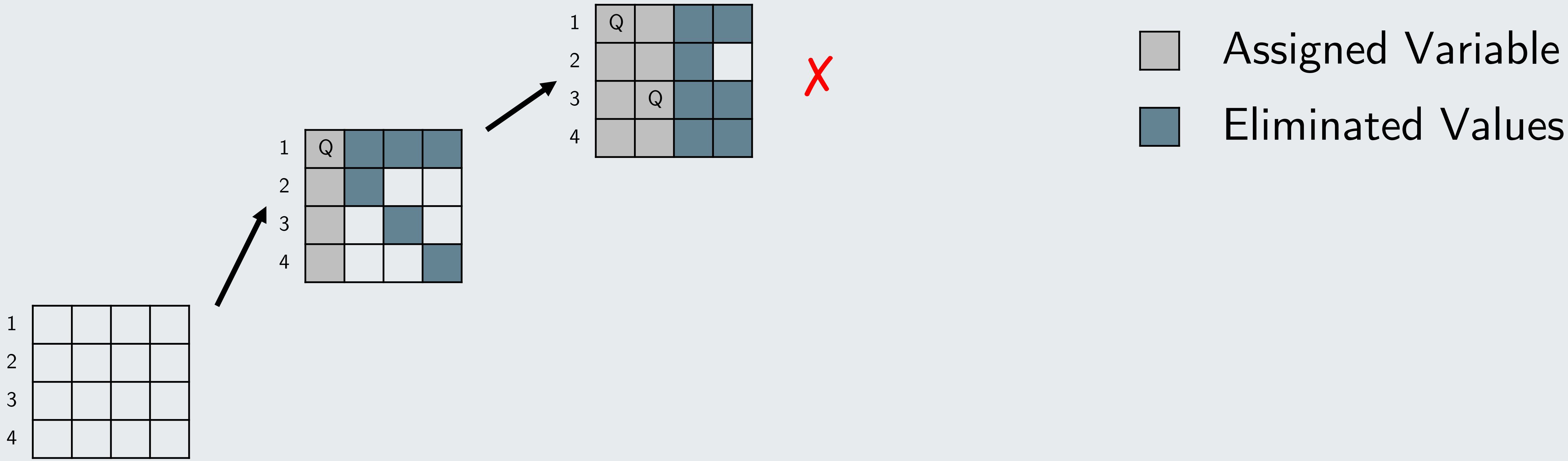
Problem 1



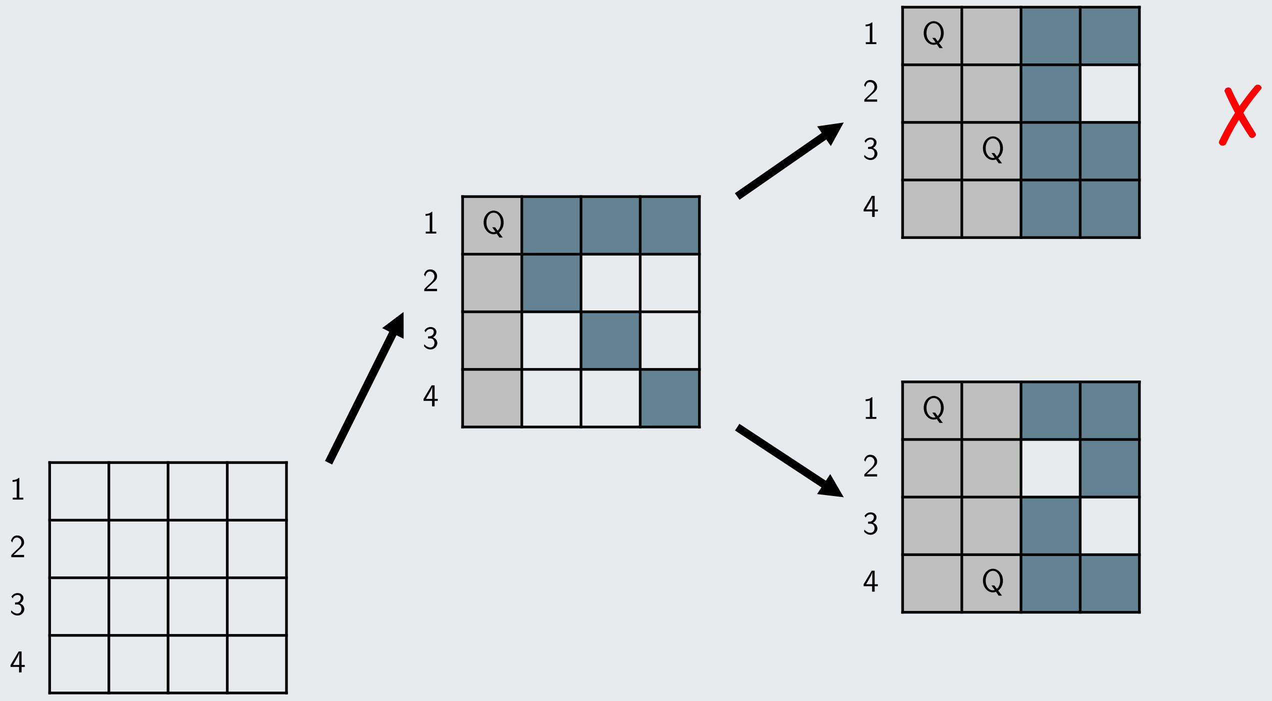
Assigned Variable

Eliminated Values

Problem 1



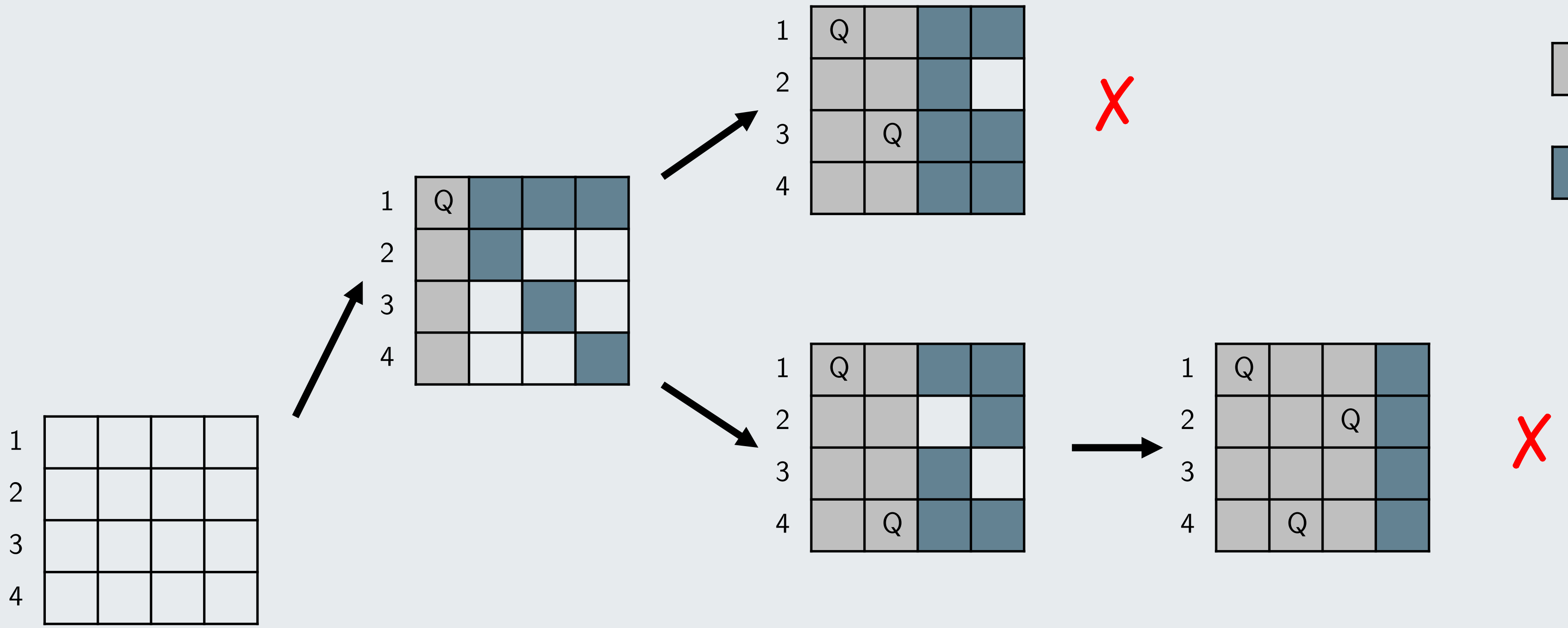
Problem 1



Assigned Variable

Eliminated Values

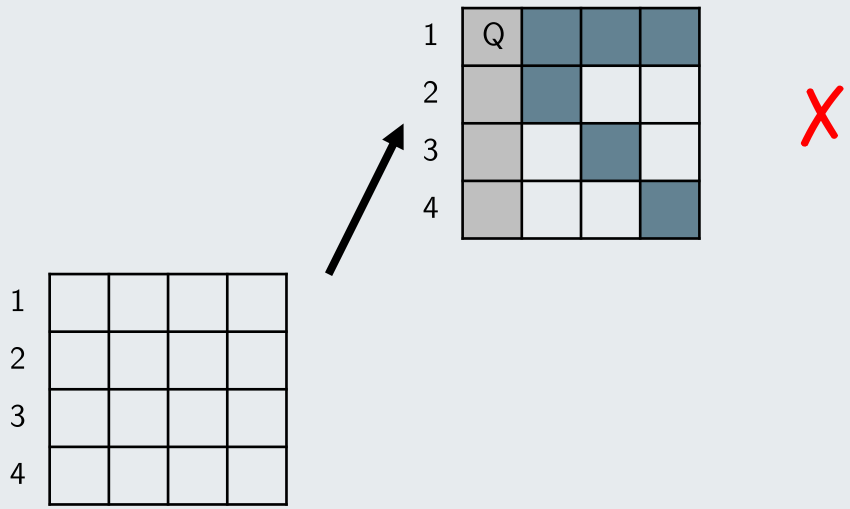
Problem 1



Assigned Variable

Eliminated Values

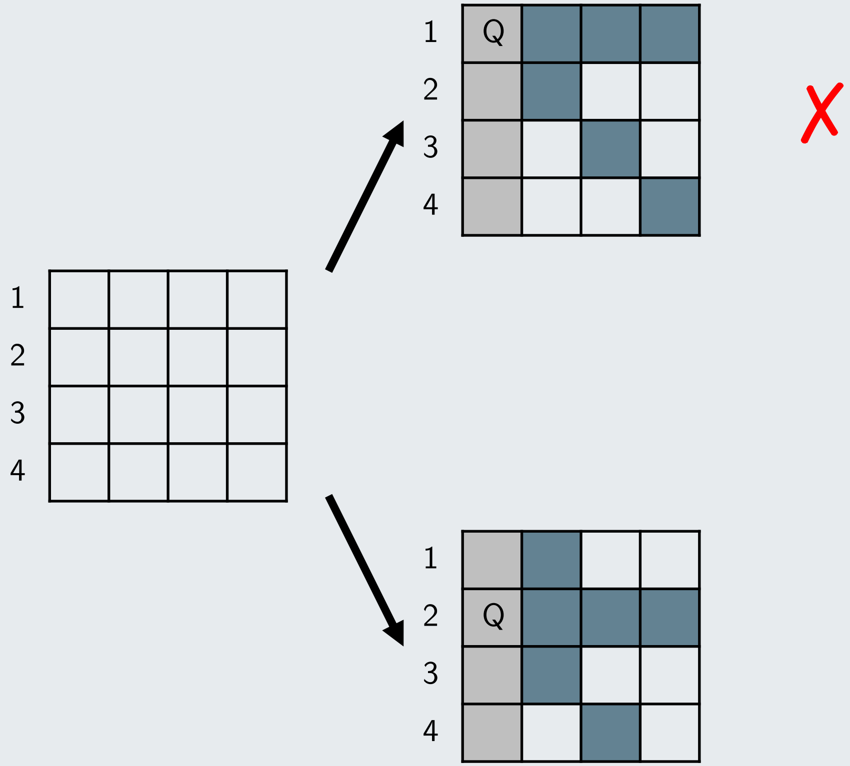
Problem 1



Assigned Variable

Eliminated Values

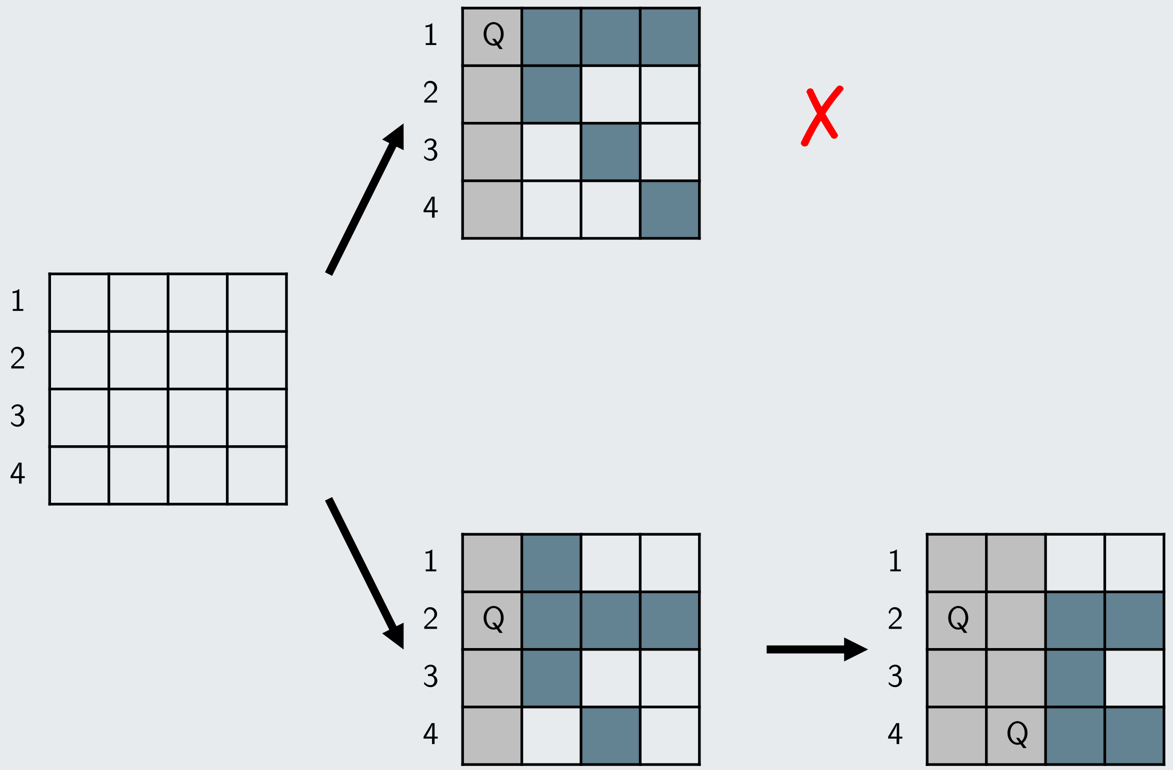
Problem 1



Assigned Variable

Eliminated Values

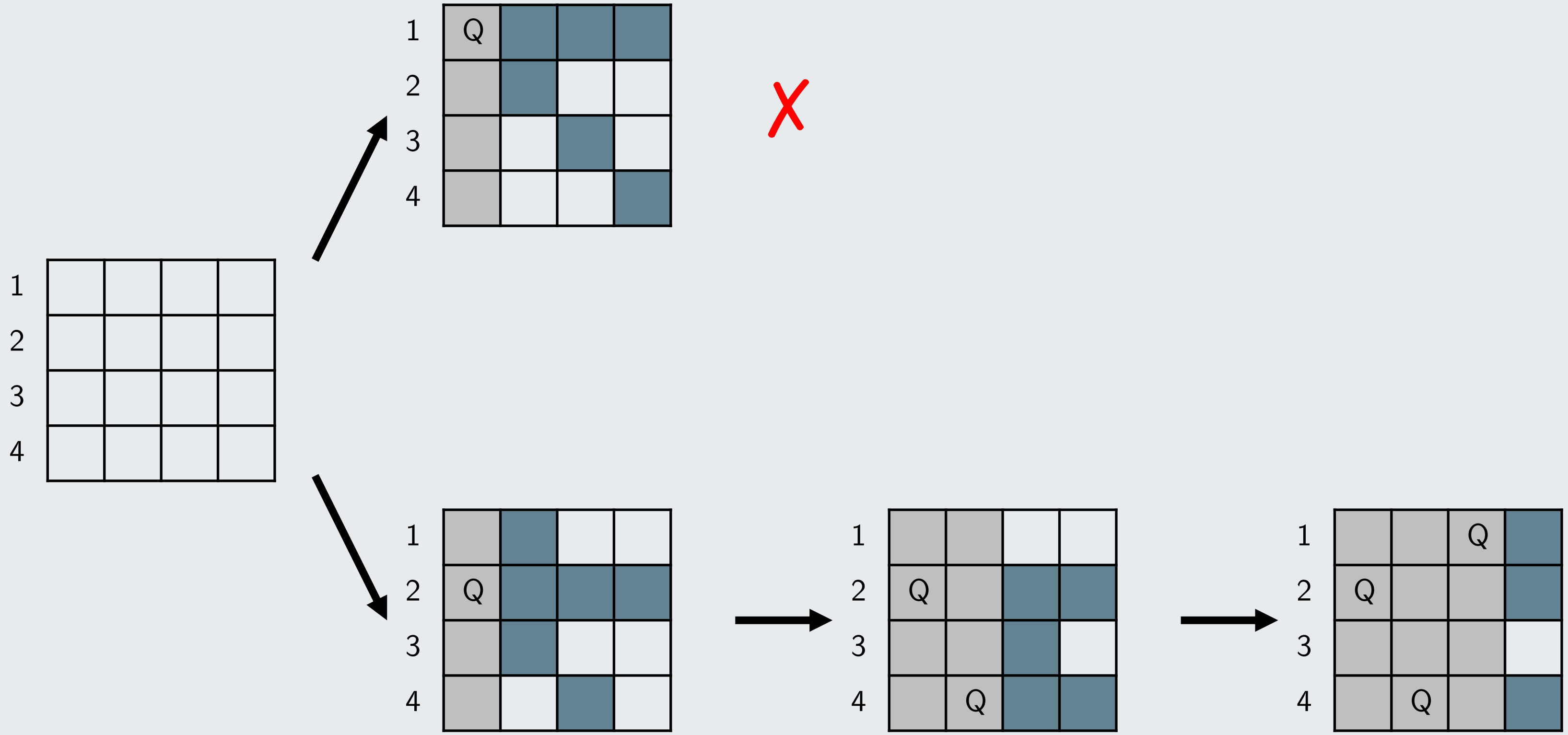
Problem 1



Assigned Variable

Eliminated Values

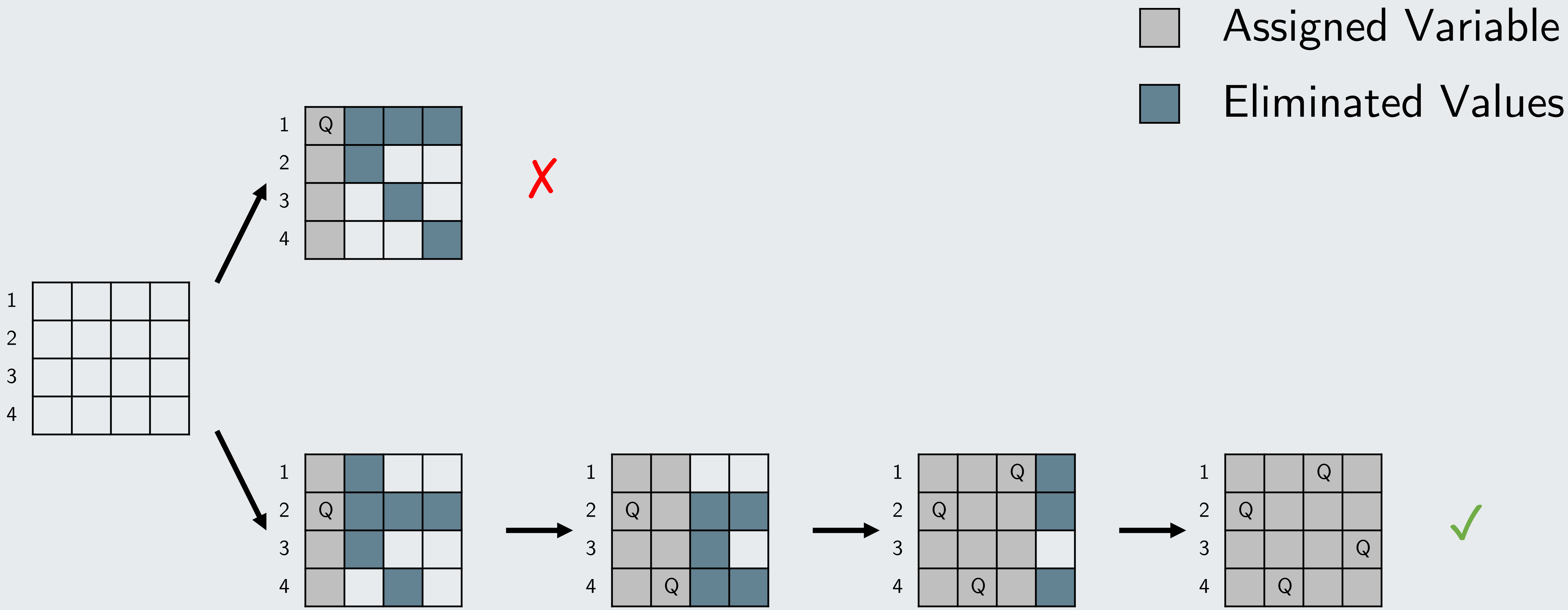
Problem 1



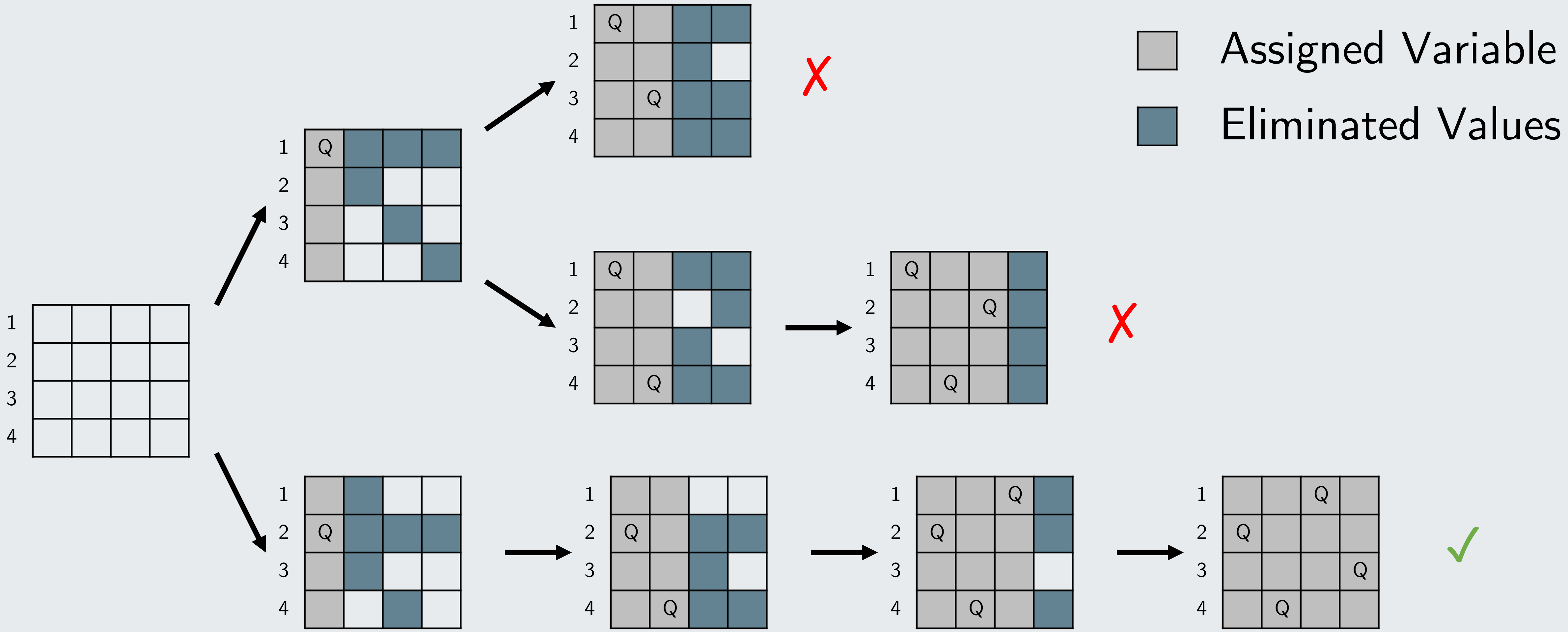
Assigned Variable

Eliminated Values

Problem 1



Problem 1



Inference

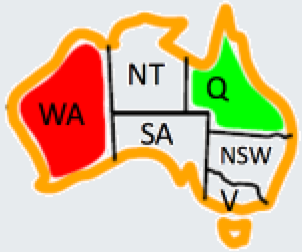
```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
   return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment then
         add {var = value} to assignment
         inferences ← INFERENCE(csp, var, value)
         if inferences ≠ failure then
            add inferences to assignment
            result ← BACKTRACK(assignment, csp)
            if result ≠ failure then
               return result
      remove {var = value} and inferences from assignment
   return failure
```

AC-3: Repeatedly check all arcs and eliminate all invalid pairs of assignment.

Pros: Good at pruning away a large part of search tree.

Cons: $O(dc^3)$ time, where d is domain size and c is binary constraints.



AC-3

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REVISE(csp, X_i, X_j) then
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true
```

```
function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

Things to note:

- When revising (X_i, X_j), we only delete invalid values in domain of X_i.
- If domain of X_i is updated, we revise neighbours of X_i (excluding X_j) again.

End of File

Thank you very much for your attention!

References

- D. Ler, “Constraint Satisfaction Problems”, 2023. [Online].
- S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach," 3rd ed., Prentice Hall, 2010.