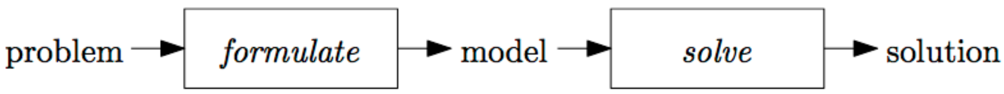
# Lecture 2

# Searching – Problem Formulation

## 2.1 Solving Problems by Searching

To solve problem, we first try to **formulate** the problem or find the model of the problem. Then, we find the solutions from the model. Basically, each obtained solution is a solution in terms of the model. We have more confidence in the solution when our model accurately represents the problem.




**Example 2.1** A rectangle has an area of 56 square centimeters. Its height is 5 centimeters longer than its width. What is the dimension of this rectangle?

Let $x$ be the width. Then, the height is $x + 5$. We set up a quadratic equation

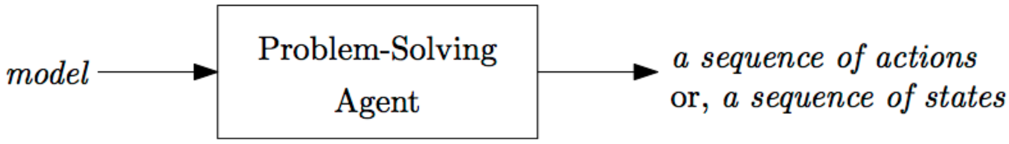
$$\begin{aligned} x(x+5) &= 56 \\ x^2 + 5x - 56 &= 0 \end{aligned}$$

Solving the equation provides the value of $x$.

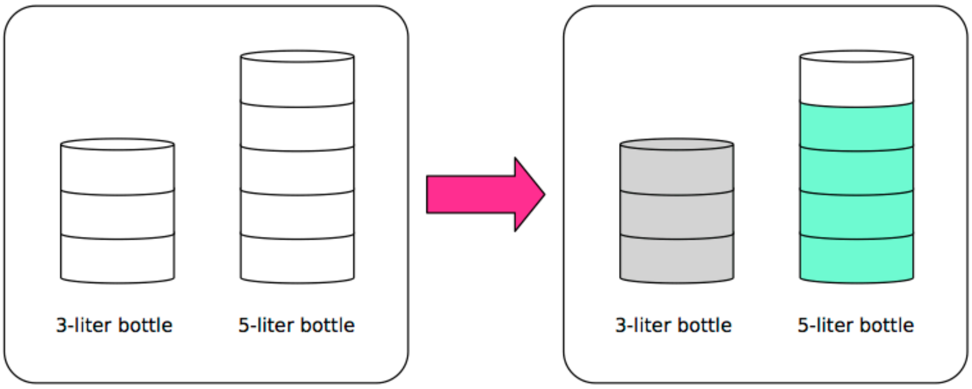
## 2.2 Problem Solving Agents

Given a model of a problem, a problem solving agent finds **a sequence of actions** that lead from an initial state to the goal. The sequence is also called a **solution** which can be **executed** later by the agent. (3.1)

**Searching** is a basic technique to find a solution of problem. It is performed by trying all possible actions iteratively in order to get to the goal.

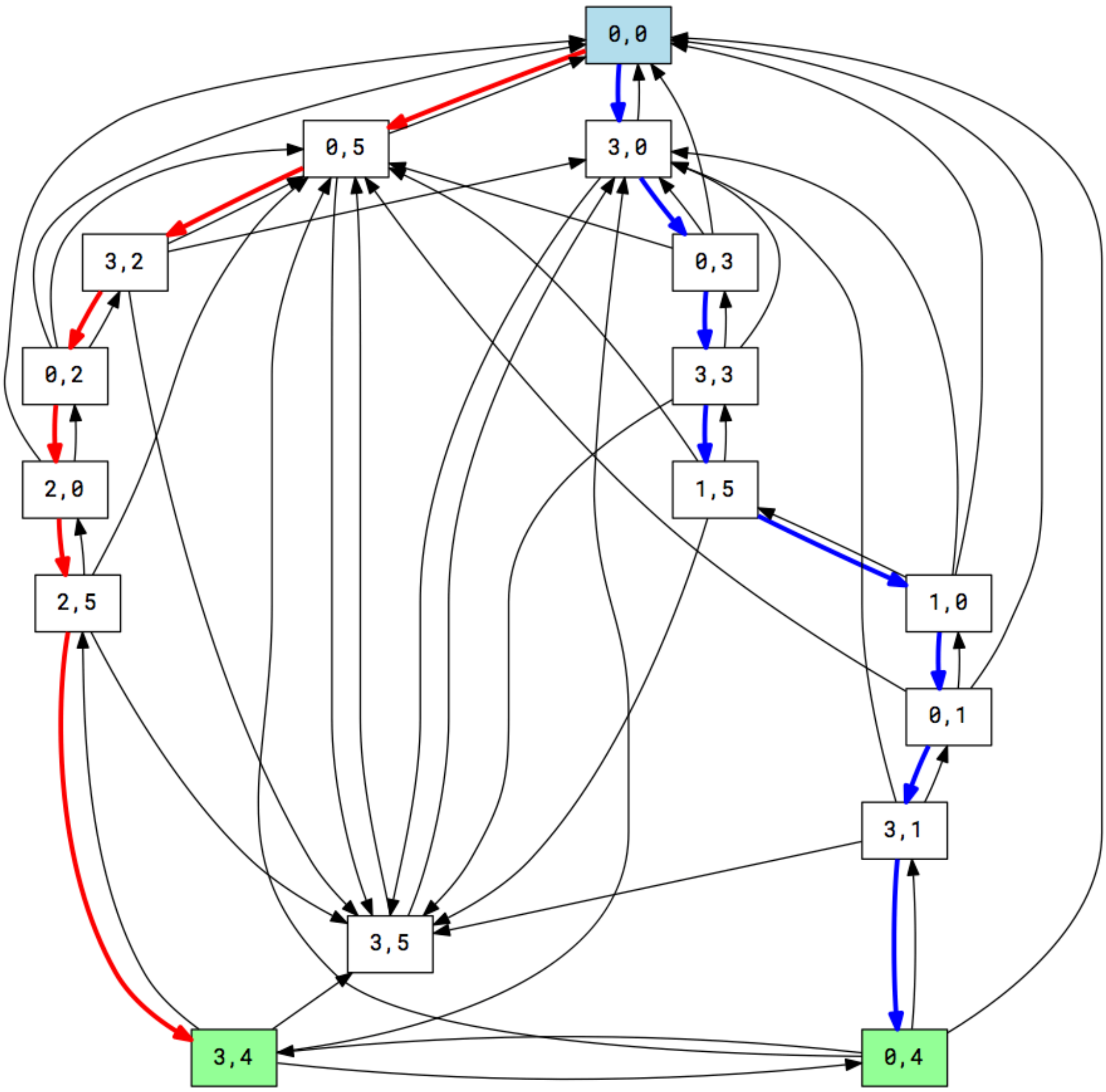



**Example 2.2** We has a 3-liter bottle, a 5-liter bottle, as well as a water faucet. How can we have 4 liters of water in the 5-liter bottle?




To solve this problem, we define a state representing the amount of water in both bottles. Each state composes of two values i.e. the amount of water in the 3-liter bottle, and the amount of water in the 5-liter bottle. For example, we can write `[3,0]` when the 3-liter bottle is full, and the 5-liter is empty.

We can draw a graph showing the changes of amount of water after applying actions. Here, each node of the graph is a state, and each edge shows an action transforming a state to another state.




A *solution* of the problem is a path from the initial state to one of the goal states. One of the solutions from the graph is

[0,0] → [0,5] → [3,2] → [0,2] → [2,0] → [2,5] → [3,4]

### 2.2.1 Problem Formulation

Here are what we need to determine as the *model* of a search problem.

**State** represents a situation of problem. A state contains all necessary information to identify a situation of problem. A state is transformed to another state by applying an action.

**Initial state** is the state that the system starts in.

**Goal test** is the condition to determine whether a given state is a goal state. In real-world problems, more than one state can be considered a goal.

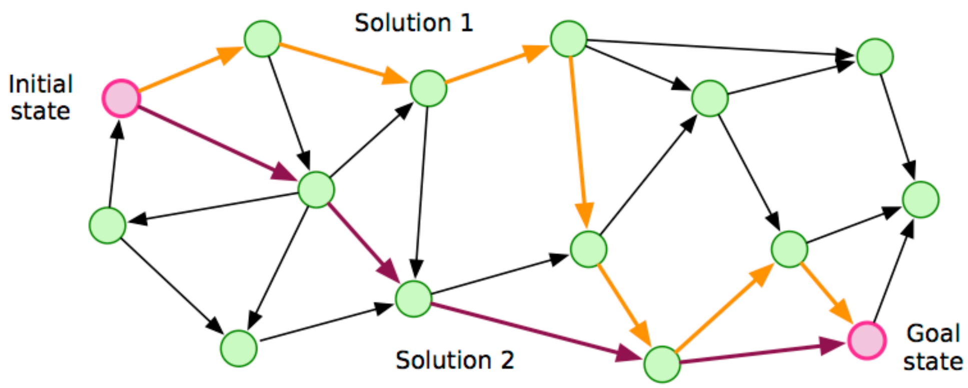
**Actions** are a set of transitions between states. Different states may have different sets of possible actions.

A **successor function** takes a state and returns a set of its possible actions.

If we connect all possible states with the actions, it becomes a graph called '**state space**'. However, it is usually difficult to *explicitly* define the state space in the real-world problems.

**Path cost** represents the cost of solutions since actions may cost differently. The path cost is sometimes computed from the sum of the **step costs** along the path.

$$PathCost = \sum_i StepCost_i$$



**Example 2.3** Formulate the water-measuring problem. Here, we have a 3-liter and 5-liter bottles. We want to measure 4 liters of water.

**State** a tuple `[`$x$`,`$y$`]` where $x$, $y$ show the amount of water in 3- and 5-liter bottles respectively.

**Initial state** a state `[0,0]`

**Goal test** check the amount of the 5-liter bottle is 4, or any state `[`$x$`,4]` where $0 \leq x \leq 3$.

**Actions**

- Empty one of the bottles.
- Fill up one of the bottles.
- Pour water from one bottle to the other bottle.

**Step cost** amount of water changed.

**Exercise 2.1** Write a list of successors of a state `[3,1]`.

### 2.2.2 Implementing the Problem Formulation

To develop a problem-solving agent, we implement the formulation including the *state representation*, the *goal test*, and the *successor function*.

*Successor function* is a function accepting a state and returns a list of successor states with costs.

**Example 2.4** Implementation of the water measuring problem.

```python
""" water.py """
# Define how to represent a state
class State:
    # Each state stores the amount of water in both bottles
    def __init__(self, x, y):
        self.x = x  # amount of water in bottle X
        self.y = y  # amount of water in bottle Y

    def __str__(self):
        # Convert a state into a string
        return "[%d, %d]" % (self.x, self.y)

    def __repr__(self):
        # Representation of a state
        return str(self)

# Define the initial state
def initial_state():
    # We start from the state where both bottles are empty
    return State(0, 0)

# Define how to check if a state is a goal state
def is_goal(s):
    # It is a goal when bottle Y contains 4 liters
    if (s.y == 4):
        return True
    return False

```

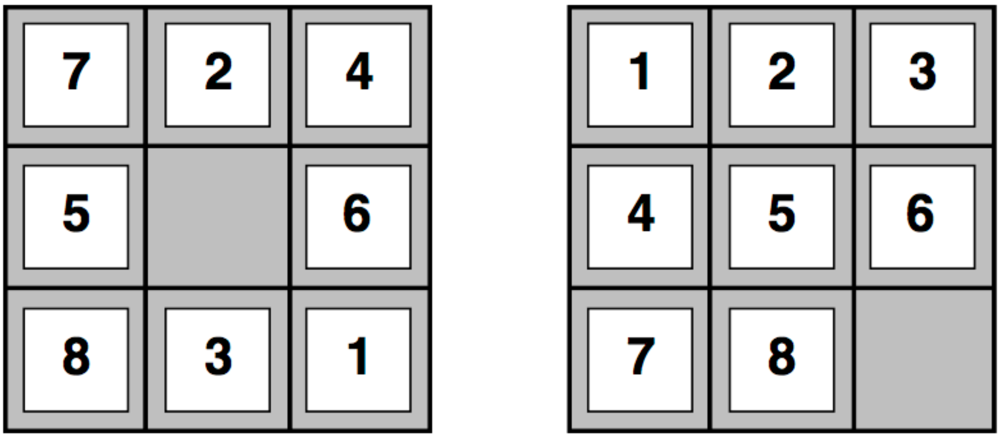
```python
# Define how to generate successors according to the problem
def successors(s):
    # This function returns a list of (state, cost) pairs
    # where state is a successor of s, and
    #       cost  is the cost required to generate this state
    # Case 1: Try to empty the bottle X
    if s.x > 0:
        # State(0, s.y) = a state where X is empty and
        #                 Y remains unchanged
        # Cost = s.x since we throw away the water in X
        yield (State(0, s.y), s.x)

    # Case 2: Try to empty the bottle Y
    if s.y > 0:
        yield (State(s.x, 0), s.y)

    # Case 3: Try to fill up the bottle X
    if s.x < 3:
        yield (State(3, s.y), 3-s.x)

    # Case 4: Try to fill up the bottle Y
    if s.y < 5:
        yield (State(s.x, 5), 5-s.y)

    # Case 5: Try to pour water from X to Y
    t = 5-s.y   # available space of Y
    if s.x > 0 and t > 0:
        if s.x > t:
            # Pour until Y is full
            yield (State(s.x-t, 5), t)
        else:
            # Pour until X is empty
            yield (State(0, s.y+s.x), s.x)

    # Case 6: Try to pour water from Y to X
    t = 3-s.x   # available space of X
    if s.y > 0 and t > 0:
        if s.y > t:
            # Pour until X is full
            yield (State(3, s.y-t), t)
        else:
            # Pour until Y is empty
            yield (State(s.x+s.y, 0), s.y)
```

We can then create a Python script to test the implementation.

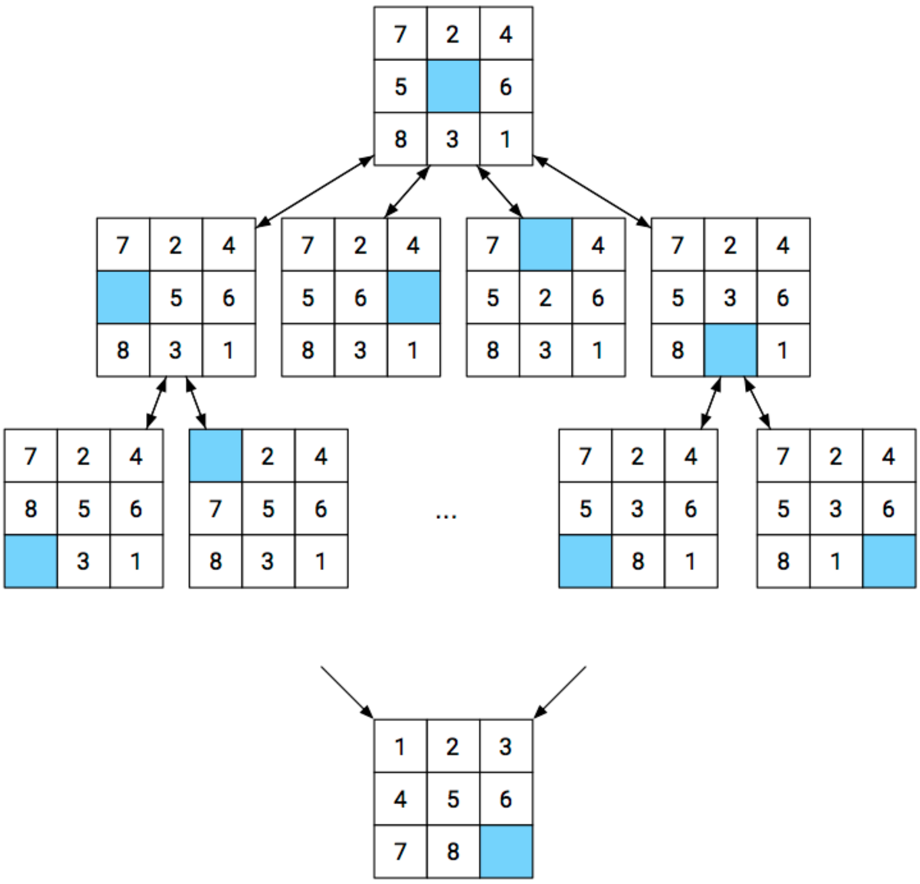
```python
""" water_sample.py """
import water

s1 = water.initial_state()
print("initial state =", s1)
print("is_goal(s1) =", water.is_goal(s1))
print("successors(s1) =", end=" ")
for s in water.successors(s1):
    print(s, end=" ")
print()
print("-----------------")

s2 = water.State(3, 4)
print("s2 =", s2)
print("is_goal(s2) =", water.is_goal(s2))
print("successors(s2) =", end=" ")
for s in water.successors(s2):
    print(s, end=" ")
print()
```

```
initial state = [0, 0]
is_goal(s1) = False
successors(s1) = ([3, 0], 3) ([0, 5], 5)
-----------------
s2 = [3, 4]
is_goal(s2) = True
successors(s2) = ([0, 4], 3) ([3, 0], 4) ([3, 5], 1) ([2, 5], 1)
```

**Exercise 2.2** 8-puzzle is a sliding puzzle with the objective to order the tiles in order by making sliding moves. If we formulate this problem as a search problem, explain how to represent a *state* and the possible *actions*.




**Start State** **Goal State**




State space

*(This page is intentionally left blank.)*

**Example 2.5** Implementation of the 8-puzzle problem.

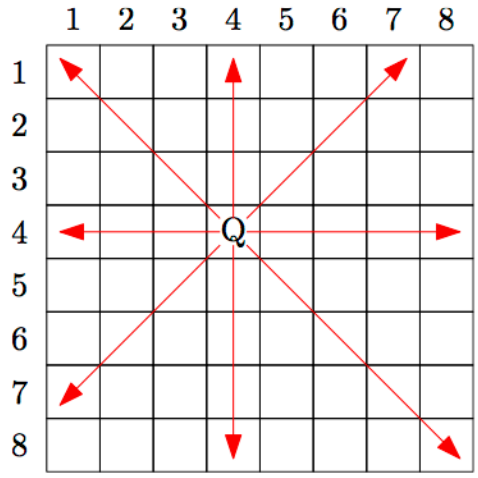
```python
""" eight_puzzle.py """
import copy
class State:
    def __init__(self, b, r, c):
        # b   = a list showing the tile locations
        # r,c = the row and column of the blank
        self.b = b
        self.r = r
        self.c = c

    def __str__(self):
        return str(self.b)

    def __repr__(self):
        return str(self)

    def pretty_print(self):
        print("+---"*3 + "+")
        for i in range(9):
            print("|", self.b[i], end=" ")
            if i % 3 == 2:
                print("|")
                print("+---"*3 + "+")

    def move_blank_to(self, new_r, new_c):
        tmp = self.b[self.r*3 + self.c]
        self.b[self.r*3 + self.c] = self.b[new_r*3 + new_c]
        self.b[new_r*3 + new_c] = tmp
        self.r = new_r
        self.c = new_c

def initial_state():
    b = [7, 2, 4, 5, 0, 6, 8, 3, 1]
    r = 1
    c = 1
    return State(b, r, c)

def is_goal(s):
    return s.b == [1, 2, 3, 4, 5, 6, 7, 8, 0]

```

```
def is_valid_location(r, c):
    if r >= 0 and r <= 2 and c >= 0 and c <= 2:
        return True
    return False

def successors(s):
    # Case 1: Try to move the blank up
    new_r = s.r-1
    new_c = s.c
    if is_valid_location(new_r, new_c):
        t = copy.deepcopy(s)
        t.move_blank_to(new_r, new_c)
        yield (t, 1)

    # Case 2: Try to move the blank down
    new_r = s.r+1
    new_c = s.c
    if is_valid_location(new_r, new_c):
        t = copy.deepcopy(s)
        t.move_blank_to(new_r, new_c)
        yield (t, 1)

    # Case 3: Try to move the blank to the left
    new_r = s.r
    new_c = s.c-1
    if is_valid_location(new_r, new_c):
        t = copy.deepcopy(s)
        t.move_blank_to(new_r, new_c)
        yield (t, 1)

    # Case 4: Try to move the blank to the right
    new_r = s.r
    new_c = s.c+1
    if is_valid_location(new_r, new_c):
        t = copy.deepcopy(s)
        t.move_blank_to(new_r, new_c)
        yield (t, 1)
```

```python
""" eight_puzzle_sample.py """
import eight_puzzle as p
s1 = p.initial_state()
print("s1 is")
s1.pretty_print()
print("Successors of s1 are")
for t,c  in p.successors(s1):
    t.pretty_print()
```

```
s1 is
+---+---+---+
| 7 | 2 | 4 |
+---+---+---+
| 5 | 0 | 6 |
+---+---+---+
| 8 | 3 | 1 |
+---+---+---+
Successors of s1 are
+---+---+---+
| 7 | 0 | 4 |
+---+---+---+
| 5 | 2 | 6 |
+---+---+---+
| 8 | 3 | 1 |
+---+---+---+
+---+---+---+
| 7 | 2 | 4 |
+---+---+---+
| 5 | 3 | 6 |
+---+---+---+
| 8 | 0 | 1 |
+---+---+---+
+---+---+---+
| 7 | 2 | 4 |
+---+---+---+
| 0 | 5 | 6 |
+---+---+---+
| 8 | 3 | 1 |
+---+---+---+
+---+---+---+
| 7 | 2 | 4 |
+---+---+---+
| 5 | 6 | 0 |
+---+---+---+
| 8 | 3 | 1 |
+---+---+---+
```

**Exercise 2.3** 8-queens is a puzzle on placing eight queens on an $8 \times 8$ chess board. Explain how to represent a *state*, the *goal test*, and the *actions*.



**Example 2.6** There are many ways to formulate the 8-queens problem. Here is another formalation.

**State** an 8-tuple representing the row number that a queen is placed in each column; 0 means no queen in that column; no attacking between any pair of queens is allowed. For example,

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | Q | | | | | | | |
| 2 | | | | | | | | |
| 3 | | | | | | | | |
| 4 | | | Q | | | | | |
| 5 | | | | | | | | |
| 6 | | | | | | | | |
| 7 | | Q | | | | | | |
| 8 | | | | | | | | |

[ 1, 7, 4, 0, 0, 0, 0, 0 ]

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | Q | | |
| 2 | Q | | | | | | | |
| 3 | | | | | Q | | | |
| 4 | | Q | | | | | | |
| 5 | | | | | | | | Q |
| 6 | | | Q | | | | | |
| 7 | | | | | | | Q | |
| 8 | | | | Q | | | | |

[ 2, 4, 6, 8, 3, 1, 7, 5 ]

**Initial state** an empty chess board i.e. `[0,0,0,0,0,0,0,0]`

**Goal test** a board with 8 queens on the board

**Actions** add a queen to any row in the leftmost empty column. This new queen must not attacked any other queen.

Any formulations can be used to find a solution for the 8-queen problem.

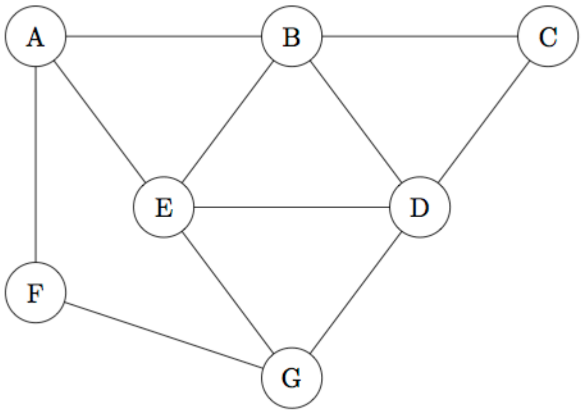
In the 1st formulation, there are $64 \cdot 63 \cdot 62 \cdot 61 \cdot 60 \cdot 59 \cdot 58 \cdot 57 \approx 1.8 \times 10^{14}$ possible states. In the 2nd formulation, there are only 2,057 states.

**Example 2.7** Implementation of the 8-queens problem.

```
import copy

# Set size of the board
N = 8
class NQueen:
    def __init__(self):
        self.b = [0]*N
        self.n = 0

    def __str__(self):
        return str(self.b)

    def __repr__(self):
        return str(self)

    def pretty_print(self):
        print("+---"*N + "+")
        for i in range(1, N+1):
            row = ""
            for j in range(N):
                if self.b[j] == i:
                    row += "| Q "
                else:
                    row += "|   "
            row += "|"
            print(row)
            print("+---"*N + "+")

def initial_state():
    s = NQueen()
    s.b[s.n] = 2;
    s.n += 1;
    return s

def is_goal(s):
    if s.n == N:
        return True
    return False

```

```python
def attack(r1, c1, r2, c2):
    # return True when the queen at row r1, column c1 attacks
    # the queen at row r2, column c2
    if r1 == r2 or c1 == c2:
        # Two queens are on the same row or column
        return True
    if (r1-r2) == (c1-c2) or (r1-r2) == (c2-c1):
        # Two queens are on the same diagonal line
        return True
    return False

def is_okay_to_add(s, q):
    # return True when adding a queen into a state s at row q
    # does not cause any attack
    for i in range(s.n):
        if attack(s.b[i], i, q, s.n):
            return False
    return True

def successors(s):
    # Try to place a queen on the next column
    for i in range(1, N+1):
        if is_okay_to_add(s, i):
            t = copy.deepcopy(s)
            t.b[t.n] = i
            t.n += 1
            yield (t, 1)
```

**Exercise 2.4** From an undirected graph shown below, we want to color the nodes of the graph using four colors, i.e. red, green, blue, and yellow. Formulate this graph-coloring problem as a search problem.



## 2.3 Searching for Solutions

Problem-solving agents find a solution by conducting search. Here is the search algorithm.

1. Start from the initial state
2. Apply all possible actions to the state, generate the set of successors, and keep them in a data structure.
3. Choose one of the successors out of the data structure.
4. Iteratively follow step 2 and 3 until the goal test is satisfied.
5. Trace back to obtain the sequence of actions from the goal state to the initial state

# References

Russell, S. and Norvig, P. (2010). Artificial Intelligence: A Modern Approach (3rd edition). Pearson/Prentice Hall.
Michalewicz, F. and Fogel, D. B. (1998). How to Solve It: Modern Heuristics. Springer.