In [4]:
```
!pip install simpleai
```

```
Collecting simpleai
  Downloading simpleai-0.8.3.tar.gz (94 kB)
Using legacy 'setup.py install' for simpleai, since package 'wheel' is not instal
led.
Installing collected packages: simpleai
    Running setup.py install for simpleai: started
    Running setup.py install for simpleai: finished with status 'done'
Successfully installed simpleai-0.8.3
```

```
WARNING: You are using pip version 21.2.3; however, version 25.0.1 is available.
You should consider upgrading via the 'C:\Users\smain\AppData\Local\Programs\Pyth
on\Python310\python.exe -m pip install --upgrade pip' command.
```

In [10]:
```python
import simpleai.search as ss

class CustomProblem(ss.SearchProblem):
    """Problème de recherche basé sur la génération de la chaîne cible avec l'al

    def set_target(self, target_string):
        self.target_string = target_string

    def actions(self, cur_state):
        """Retourne la liste des actions possibles (ajouter un caractère)."""
        if len(cur_state) < len(self.target_string):
            alphabets = 'abcdefghijklmnopqrstuvwxyz'
            return list(set(alphabets + ' ' + alphabets.upper()))
        return []

    def result(self, cur_state, action):
        """Concatène l'état actuel avec l'action pour obtenir le nouvel état."""
        return cur_state + action

    def is_goal(self, cur_state):
        """Vérifie si l'objectif est atteint (la chaîne est construite)."""
        return cur_state == self.target_string

    def heuristic(self, cur_state):
        """Calcule une heuristique basée sur le nombre de caractères incorrects
        dist = sum(1 if cur_state[i] != self.target_string[i] else 0 for i in ra
        diff = len(self.target_string) - len(cur_state)
        return dist + diff

def run_greedy_search(input_string, initial_state=""):
    """Exécute la recherche gloutonne et affiche le chemin trouvé."""
    if not input_string.strip():
        raise ValueError("❌ ERREUR : La chaîne cible ne peut pas être vide !")

    # 📌 Initialisation du problème
    problem = CustomProblem()
    problem.set_target(input_string)
    problem.initial_state = initial_state

    # 📌 Exécution de l'algorithme glouton
    print("\n🔍 Recherche en cours...")
    output = ss.greedy(problem)

    # 📌 Affichage du résultat
```

```
    if output:
        print("\n✅ Chaîne cible :", input_string)
        print("\n🔍 Chemin vers la solution :")
        for step, item in enumerate(output.path()):
            print(f"Étape {step}: {item}")
    else:
        print("❌ Aucune solution trouvée.")

# 📌 EXÉCUTION DU CODE AVEC UNE CHAÎNE PAR DÉFAUT
run_greedy_search("Artificial Intelligence")
```

```
🔍 Recherche en cours...

✅ Chaîne cible : Artificial Intelligence

🔍 Chemin vers la solution :
Étape 0: (None, '')
Étape 1: ('A', 'A')
Étape 2: ('r', 'Ar')
Étape 3: ('t', 'Art')
Étape 4: ('i', 'Arti')
Étape 5: ('f', 'Artif')
Étape 6: ('i', 'Artifi')
Étape 7: ('c', 'Artific')
Étape 8: ('i', 'Artifici')
Étape 9: ('a', 'Artificia')
Étape 10: ('l', 'Artificial')
Étape 11: (' ', 'Artificial ')
Étape 12: ('I', 'Artificial I')
Étape 13: ('n', 'Artificial In')
Étape 14: ('t', 'Artificial Int')
Étape 15: ('e', 'Artificial Inte')
Étape 16: ('l', 'Artificial Intel')
Étape 17: ('l', 'Artificial Intell')
Étape 18: ('i', 'Artificial Intelli')
Étape 19: ('g', 'Artificial Intellig')
Étape 20: ('e', 'Artificial Intellige')
Étape 21: ('n', 'Artificial Intelligen')
Étape 22: ('c', 'Artificial Intelligenc')
Étape 23: ('e', 'Artificial Intelligence')
```

Le bloc de code précédent implémente une recherche gloutonne pour générer une chaîne de caractères cible en construisant progressivement la solution optimale.

- 🔹 Classe CustomProblem :

Définit un problème où chaque état est une chaîne partiellement construite. Les actions possibles ajoutent une lettre de l'alphabet. L'heuristique estime le coût en fonction des erreurs et des caractères manquants. 🔹 Fonction run_greedy_search :

Initialise le problème avec une chaîne cible. Utilise l'algorithme de recherche gloutonne pour atteindre la solution. Affiche le chemin suivi pour construire la chaîne. 🔹 Exécution : Génère la chaîne "Artificial Intelligence" en minimisant les erreurs à chaque étape.

In [11]:
```
from simpleai.search import CspProblem, backtrack, \
        min_conflicts, MOST_CONSTRAINED_VARIABLE, \
        HIGHEST_DEGREE_VARIABLE, LEAST_CONSTRAINING_VALUE
```

```
# Constraint that expects all the different variables
# to have different values
def constraint_unique(variables, values):
    # Check if all the values are unique
    return len(values) == len(set(values))

# Constraint that specifies that one variable
# should be bigger than other
def constraint_bigger(variables, values):
    return values[0] > values[1]

# Constraint that specifies that there should be
# one odd and one even variables in the two variables
def constraint_odd_even(variables, values):
    # If first variable is even, then second should
    # be odd and vice versa
    if values[0] % 2 == 0:
        return values[1] % 2 == 1
    else:
        return values[1] % 2 == 0

if __name__=='__main__':
    variables = ('John', 'Anna', 'Tom', 'Patricia')

    domains = {
        'John': [1, 2, 3],
        'Anna': [1, 3],
        'Tom': [2, 4],
        'Patricia': [2, 3, 4],
    }

    constraints = [
        (('John', 'Anna', 'Tom'), constraint_unique),
        (('Tom', 'Anna'), constraint_bigger),
        (('John', 'Patricia'), constraint_odd_even),
    ]

    problem = CspProblem(variables, domains, constraints)

    print('\nSolutions:\n\nNormal:', backtrack(problem))
    print('\nMost constrained variable:', backtrack(problem,
            variable_heuristic=MOST_CONSTRAINED_VARIABLE))
    print('\nHighest degree variable:', backtrack(problem,
            variable_heuristic=HIGHEST_DEGREE_VARIABLE))
    print('\nLeast constraining value:', backtrack(problem,
            value_heuristic=LEAST_CONSTRAINING_VALUE))
    print('\nMost constrained variable and least constraining value:',
            backtrack(problem, variable_heuristic=MOST_CONSTRAINED_VARIABLE,
            value_heuristic=LEAST_CONSTRAINING_VALUE))
    print('\nHighest degree and least constraining value:',
            backtrack(problem, variable_heuristic=HIGHEST_DEGREE_VARIABLE,
            value_heuristic=LEAST_CONSTRAINING_VALUE))
    print('\nMinimum conflicts:', min_conflicts(problem))
```

```
Solutions:

Normal: {'John': 1, 'Anna': 3, 'Tom': 4, 'Patricia': 2}

Most constrained variable: {'Anna': 1, 'Tom': 2, 'John': 3, 'Patricia': 2}

Highest degree variable: {'John': 1, 'Anna': 3, 'Tom': 4, 'Patricia': 2}

Least constraining value: {'John': 1, 'Anna': 3, 'Tom': 4, 'Patricia': 2}

Most constrained variable and least constraining value: {'Anna': 1, 'Tom': 2, 'Jo
hn': 3, 'Patricia': 2}

Highest degree and least constraining value: {'John': 1, 'Anna': 3, 'Tom': 4, 'Pa
tricia': 2}

Minimum conflicts: {'John': 2, 'Anna': 3, 'Tom': 4, 'Patricia': 3}
```

Le bloc de code précédent précédentmet en place un problème de satisfaction de contraintes (CSP) et utilise des algorithmes de recherche pour trouver des solutions conformes aux contraintes définies.

- Définition des contraintes :

constraint_unique : Tous les individus doivent avoir des valeurs uniques.
constraint_bigger : Tom doit avoir une valeur supérieure à Anna. constraint_odd_even : John et Patricia doivent avoir une parité opposée. 🔹 Configuration du problème :

Quatre variables (John, Anna, Tom, Patricia) avec des domaines de valeurs possibles. Liste des contraintes qui régissent les relations entre ces variables. 🔹 Résolution avec différentes heuristiques :

backtrack : Algorithme de retour arrière standard. MOST_CONSTRAINED_VARIABLE : Choisit en priorité la variable avec le moins d'options disponibles.
HIGHEST_DEGREE_VARIABLE : Sélectionne la variable avec le plus de contraintes.
LEAST_CONSTRAINING_VALUE : Privilégie les valeurs ayant le moins d'impact sur les autres variables. min_conflicts : Algorithme de résolution par conflits minimaux.

In [12]:
```python
from simpleai.search import CspProblem, backtrack

# Define the function that imposes the constraint
# that neighbors should be different
def constraint_func(names, values):
    return values[0] != values[1]

if __name__=='__main__':
    # Specify the variables
    names = ('Mark', 'Julia', 'Steve', 'Amanda', 'Brian',
             'Joanne', 'Derek', 'Allan', 'Michelle', 'Kelly')

    # Define the possible colors
    colors = dict((name, ['red', 'green', 'blue', 'gray']) for name in names)

    # Define the constraints
    constraints = [
        (('Mark', 'Julia'), constraint_func),
```

```
        (('Mark', 'Steve'), constraint_func),
        (('Julia', 'Steve'), constraint_func),
        (('Julia', 'Amanda'), constraint_func),
        (('Julia', 'Derek'), constraint_func),
        (('Julia', 'Brian'), constraint_func),
        (('Steve', 'Amanda'), constraint_func),
        (('Steve', 'Allan'), constraint_func),
        (('Steve', 'Michelle'), constraint_func),
        (('Amanda', 'Michelle'), constraint_func),
        (('Amanda', 'Joanne'), constraint_func),
        (('Amanda', 'Derek'), constraint_func),
        (('Brian', 'Derek'), constraint_func),
        (('Brian', 'Kelly'), constraint_func),
        (('Joanne', 'Michelle'), constraint_func),
        (('Joanne', 'Amanda'), constraint_func),
        (('Joanne', 'Derek'), constraint_func),
        (('Joanne', 'Kelly'), constraint_func),
        (('Derek', 'Kelly'), constraint_func),
    ]

    # Solve the problem
    problem = CspProblem(names, colors, constraints)

    # Print the solution
    output = backtrack(problem)
    print('\nColor mapping:\n')
    for k, v in output.items():
        print(k, '==>', v)
```

```
Color mapping:

Mark ==> red
Julia ==> green
Steve ==> blue
Amanda ==> red
Brian ==> red
Joanne ==> green
Derek ==> blue
Allan ==> red
Michelle ==> gray
Kelly ==> gray
```

Ce code résout un problème de coloration de graphe en utilisant un problème de satisfaction de contraintes (CSP) et l'algorithme de backtracking. L'objectif est d'attribuer une couleur à chaque individu tout en respectant la contrainte que deux voisins ne doivent pas avoir la même couleur.

D'abord, une liste de personnes est définie, représentant les nœuds du graphe. Chacune d'elles peut être colorée avec l'une des quatre couleurs disponibles : rouge, vert, bleu ou gris. Ensuite, des contraintes sont spécifiées pour indiquer quelles personnes sont voisines et ne doivent donc pas avoir la même couleur.

Une fonction de contrainte est créée pour vérifier que deux individus liés ne partagent pas la même couleur. Une fois le problème formulé, l'algorithme de backtracking est utilisé pour trouver une solution. Il attribue les couleurs tout en respectant les contraintes

et génère une solution optimale, qui est ensuite affichée sous forme de correspondance entre chaque individu et sa couleur attribuée.

In [13]:
```
from simpleai.search import astar, SearchProblem

# Class containing methods to solve the puzzle
class PuzzleSolver(SearchProblem):
    # Action method to get the list of the possible
    # numbers that can be moved in to the empty space
    def actions(self, cur_state):
        rows = string_to_list(cur_state)
        row_empty, col_empty = get_location(rows, 'e')

        actions = []
        if row_empty > 0:
            actions.append(rows[row_empty - 1][col_empty])
        if row_empty < 2:
            actions.append(rows[row_empty + 1][col_empty])
        if col_empty > 0:
            actions.append(rows[row_empty][col_empty - 1])
        if col_empty < 2:
            actions.append(rows[row_empty][col_empty + 1])

        return actions

    # Return the resulting state after moving a piece to the empty space
    def result(self, state, action):
        rows = string_to_list(state)
        row_empty, col_empty = get_location(rows, 'e')
        row_new, col_new = get_location(rows, action)

        rows[row_empty][col_empty], rows[row_new][col_new] = \
                rows[row_new][col_new], rows[row_empty][col_empty]

        return list_to_string(rows)

    # Returns true if a state is the goal state
    def is_goal(self, state):
        return state == GOAL

    # Returns an estimate of the distance from a state to
    # the goal using the manhattan distance
    def heuristic(self, state):
        rows = string_to_list(state)

        distance = 0

        for number in '12345678e':
            row_new, col_new = get_location(rows, number)
            row_new_goal, col_new_goal = goal_positions[number]

            distance += abs(row_new - row_new_goal) + abs(col_new - col_new_goal

        return distance

# Convert list to string
def list_to_string(input_list):
    return '\n'.join(['-'.join(x) for x in input_list])
```

```
# Convert string to list
def string_to_list(input_string):
    return [x.split('-') for x in input_string.split('\n')]

# Find the 2D location of the input element
def get_location(rows, input_element):
    for i, row in enumerate(rows):
        for j, item in enumerate(row):
            if item == input_element:
                return i, j

# Final result that we want to achieve
GOAL = '''1-2-3
4-5-6
7-8-e'''

# Starting point
INITIAL = '''1-e-2
6-3-4
7-5-8'''

# Create a cache for the goal position of each piece
goal_positions = {}
rows_goal = string_to_list(GOAL)
for number in '12345678e':
    goal_positions[number] = get_location(rows_goal, number)

# Create the solver object
result = astar(PuzzleSolver(INITIAL))

# Print the results
for i, (action, state) in enumerate(result.path()):
    print()
    if action == None:
        print('Initial configuration')
    elif i == len(result.path()) - 1:
        print('After moving', action, 'into the empty space. Goal achieved!')
    else:
        print('After moving', action, 'into the empty space')

    print(state)
```

```
Initial configuration
1-e-2
6-3-4
7-5-8

After moving 2 into the empty space
1-2-e
6-3-4
7-5-8

After moving 4 into the empty space
1-2-4
6-3-e
7-5-8

After moving 3 into the empty space
1-2-4
6-e-3
7-5-8

After moving 6 into the empty space
1-2-4
e-6-3
7-5-8

After moving 1 into the empty space
e-2-4
1-6-3
7-5-8

After moving 2 into the empty space
2-e-4
1-6-3
7-5-8

After moving 4 into the empty space
2-4-e
1-6-3
7-5-8

After moving 3 into the empty space
2-4-3
1-6-e
7-5-8

After moving 6 into the empty space
2-4-3
1-e-6
7-5-8

After moving 4 into the empty space
2-e-3
1-4-6
7-5-8

After moving 2 into the empty space
e-2-3
1-4-6
7-5-8
```

```
After moving 1 into the empty space
1-2-3
e-4-6
7-5-8

After moving 4 into the empty space
1-2-3
4-e-6
7-5-8

After moving 5 into the empty space
1-2-3
4-5-6
7-e-8

After moving 8 into the empty space. Goal achieved!
1-2-3
4-5-6
7-8-e
```

Ce code implémente une résolution du puzzle du taquin (8-puzzle) à l'aide de l'*algorithme A (A-star)**. Il définit une classe PuzzleSolver qui hérite de SearchProblem pour modéliser le problème sous forme de recherche d'état.

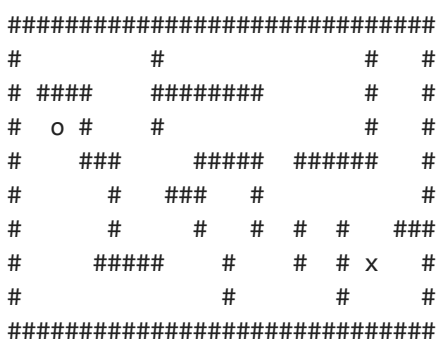
Le puzzle est représenté sous forme de chaînes de caractères converties en listes 2D pour faciliter les manipulations. À chaque étape, l'algorithme identifie les mouvements possibles d'une case vide (e) en échangeant sa position avec une case adjacente.

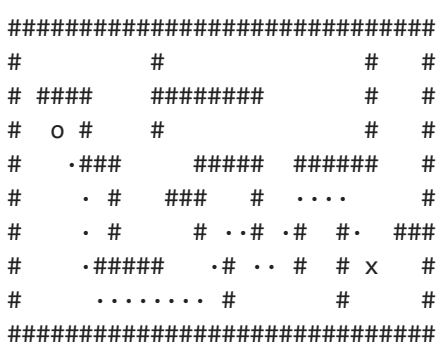
La fonction heuristique utilise la distance de Manhattan pour estimer la distance entre l'état actuel et l'objectif. Le programme exécute ensuite l'algorithme A* pour trouver le chemin optimal vers la solution. Enfin, il affiche les états intermédiaires jusqu'à l'obtention de la configuration finale correcte.

In [14]:
```python
import math
from simpleai.search import SearchProblem, astar

# Class containing the methods to solve the maze
class MazeSolver(SearchProblem):
    # Initialize the class
    def __init__(self, board):
        self.board = board
        self.goal = (0, 0)

        for y in range(len(self.board)):
            for x in range(len(self.board[y])):
                if self.board[y][x].lower() == "o":
                    self.initial = (x, y)
                elif self.board[y][x].lower() == "x":
                    self.goal = (x, y)

        super(MazeSolver, self).__init__(initial_state=self.initial)

    # Define the method that takes actions
    # to arrive at the solution
    def actions(self, state):
        actions = []
```

```
        for action in COSTS.keys():
            newx, newy = self.result(state, action)
            if self.board[newy][newx] != "#":
                actions.append(action)

        return actions

    # Update the state based on the action
    def result(self, state, action):
        x, y = state

        if action.count("up"):
            y -= 1
        if action.count("down"):
            y += 1
        if action.count("left"):
            x -= 1
        if action.count("right"):
            x += 1

        new_state = (x, y)

        return new_state

    # Check if we have reached the goal
    def is_goal(self, state):
        return state == self.goal

    # Compute the cost of taking an action
    def cost(self, state, action, state2):
        return COSTS[action]

    # Heuristic that we use to arrive at the solution
    def heuristic(self, state):
        x, y = state
        gx, gy = self.goal

        return math.sqrt((x - gx) ** 2 + (y - gy) ** 2)

if __name__ == "__main__":
    # Define the map
    MAP = """
    ##############################
    #         #              #   #
    # ####    ########       #   #
    #  o #    #              #   #
    #    ###     #####  ######   #
    #      #   ###   #           #
    #      #     #   #  #  #   ###
    #     #####    #    #  # x   #
    #              #       #     #
    ##############################
    """

    # Convert map to a list
    print(MAP)
    MAP = [list(x) for x in MAP.split("\n") if x]

    # Define cost of moving around the map
    cost_regular = 1.0
```

```
    cost_diagonal = 1.7

    # Create the cost dictionary
    COSTS = {
        "up": cost_regular,
        "down": cost_regular,
        "left": cost_regular,
        "right": cost_regular,
        "up left": cost_diagonal,
        "up right": cost_diagonal,
        "down left": cost_diagonal,
        "down right": cost_diagonal,
    }

    # Create maze solver object
    problem = MazeSolver(MAP)

    # Run the solver
    result = astar(problem, graph_search=True)

    # Extract the path
    path = [x[1] for x in result.path()]

    # Print the result
    print()
    for y in range(len(MAP)):
        for x in range(len(MAP[y])):
            if (x, y) == problem.initial:
                print('o', end='')
            elif (x, y) == problem.goal:
                print('x', end='')
            elif (x, y) in path:
                print('·', end='')
            else:
                print(MAP[y][x], end='')

        print()
```





Ce code implémente un solveur de labyrinthe basé sur l'algorithme A* pour trouver le chemin optimal entre un point de départ (o) et une destination (x).

Le labyrinthe est représenté sous forme de tableau 2D, où # représente des obstacles. L'algorithme explore les mouvements possibles (haut, bas, gauche, droite, et diagonales) en évitant les murs.

L'heuristique utilisée est la distance euclidienne entre la position actuelle et la cible. À la fin, le programme affiche le labyrinthe résolu, où · marque le chemin optimal entre o et x.