Def. Suchproblem

```
P = <Q, next_states, start, goal>
```

- Q ist Menge der Zustände
- next_states: $Q- > 2^Q$; next_states(q) berechnet Zustände die in einem Schritt von q erreicht werden können
- $start \in Q$; Startzustand
- $goal \in Q$; Zielzustand

Suchalgorithmen

```
   *  - *
  / \ _ *
 /
* - * - *
 \
  \
   *   - *
    \  - *
         \ - *
```

```
def dfs(start, goal, next_states):
        Stack  = [start]
        Parent = { start: start }
        while Stack:
                state = Stack.pop()
                for ns in next_states(state):
                        if ns not in Parent:
                                Parent[ns] = state
                                if ns == goal:
                                        return path_to(goal, Parent)
                                Stack.append(ns)

def path_to(state, Parent):
        Path = [state]
        while state != Parent[state]:
                state = Parent[state]
                Path = [ state ] + Path
        return Path

def bfs(start, goal, next_states):
        Frontier = { start }
        Parent   = { start: start }
        while Frontier:
                NewFrontier = set()
                for s in Frontier:
                        for ns in next_states(s):
                                if ns not in Parent:
                                        NewFrontier.add(ns)
                                        Parent[ns] = s
                                        if ns == goal:
                                                return path_to(goal, Parent)
                Frontier = NewFrontier
```

Sliding Puzzle

```
start = [[0, 3, 4], [1, 7, 8], [2, 6, 5]]

def copy_state(state):
        # return [list(row) for row in state]
        new_state = []
        for row in state:
                new_row = []
                for col in row:
                        new_row.append(col)
                new_state.append(new_row)
        return new_state

def next_states(state):
        row, col   = find_tile(0, state)
        New_States = set()
        if row > 0:
                ns = copy_state(state)
                ns[row-1][col], ns[row][col] = ns[row][col], ns[row-1][col]
                New_States.add(ns)
        if row < len(state) - 1:
                ns = copy_state(state)
                ns[row+1][col], ns[row][col] = ns[row][col], ns[row+1][col]
                New_States.add(ns)
        if col > 0:
                ns = copy_state(state)
                ns[row][col-1], ns[row][col] = ns[row][col], ns[row][col-1]
                New_States.add(ns)
        if col < len(state[0]) - 1:
                ns = copy_state(state)
                ns[row][col+1], ns[row][col] = ns[row][col], ns[row][col+1]
                New_States.add(ns)
        return New_States
```

Missionaries

```
start = (3, 3, True)
goal  = (0, 0, False)

def problem(m, i)
    return 0 < m < i

def no_problem(m, i)
    return not problem(m, i) and not problem(3 - m, 3 - i)

def next_states(state):
    m, i, b = state
    if  b:
        return { (m-mb, i-ib, False) for mb in range(m+1)
                                     for ib in range(i+1)
                                     if 1 <= mb + ib <= 2 and no_problem(m-mb, i-ib)
               }
    else:
        return { (m+mb, i+ib, True) for mb in range(3-m+1)
                                    for ib in range(3-i+1)
                                    if 1 <= mb + ib <= 2 and no_problem(m+mb, i+ib)
               }
```

A*

Heuristik, um zu berechnen, welcher Zustand näher am Endzustand ist

$h : Q \to \R$

Eine Heuristik h ist zulässig g.d.w. $h(s)$ niemals größer ist als die wirkliche Entfernung von s zum $goal$

Eine Heuristik h ist konsistent g.d.w. $h(goal) = 0$ und $h(s_1) \leq 1 + h(s_2)$ für alle $s_2 \in next\_states(s_1)$

```
def manhattan(rowA, colA, rowB, colB):
        return abs(rowA - rowB) + abs(colA - colB)

def manhattan_slide_puzzle(stateA, stateB):
        n = len(stateA)
        result = 0
        for rowA in range(n):
                for colA in range(n):
                        tile = stateA[rowA][colA]
                        if tile != 0: # damit es zulässig ist
                                rowB, colB = find_tile(tile, stateB)
                                result += manhattan(rowA, colA, rowB, colB)
        return result

def a_star(start, goal, next_states, heuristic):
        Visited = set()
        PrioQueue = [ (heuristic(start, goal)), [start] ]
        while PrioQueue:
                _, Path = heapq.heappop(PrioQueue)
                state    = Path[-1]
                if state in Visited:
                        continue
                if state == goal:
                        return Path
                for ns in next_states(state):
                        if ns not in Visited:
                                prio = len(Path) + heuristic(ns, goal)
                                heapq.heappush(PrioQueue, (prio, Path + [ns]))
                Visited.add(state)
```