**Authors:**

Ke Zhang, Yiyang Zhou

**Run this project:**

This project requires python3. Use python main.py to run.

In main.py, specify input file in SRC\_PATH and output location in DEST\_PATH.

**Outputs:**

Output1:

1 2 3

4 0 5

6 7 8

9 10 11

12 13 14

15 16 17

18 19 20

21 22 23

24 25 26

1 2 3

4 13 5

6 7 8

9 10 11

15 12 14

24 16 17

18 19 20

21 0 23

25 22 26

6

22

D W S D E N

6 6 6 6 6 6 6

Output2:

1 2 3

4 0 5

6 7 8

9 10 11

12 13 14

15 16 17

18 19 20

21 22 23

24 25 26

1 10 2

4 5 3

6 7 8

9 13 11

21 12 14

15 16 17

18 0 20

24 19 22

25 26 23

13

43

E N W D S W D S E E N W N

13 13 13 13 13 13 13 13 13 13 13 13 13 13

Output3:

1 2 3

4 0 5

6 7 8

9 10 11

12 13 14

15 16 17

18 19 20

21 22 23

24 25 26

0 2 3

1 7 14

6 8 5

12 9 10

4 13 11

21 16 17

18 19 20

22 25 23

15 24 26

16

58

S E N D N W W S D E S W U N U N

16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16

**Source code:**

**main.py**

from agent import Agent

from task\_environment import TaskEnvironment

SRC\_PATH = "test/Input1.txt"

DEST\_PATH = "outputs/Output1.txt"

def main():

    task\_environment = TaskEnvironment(SRC\_PATH)

    agent = Agent(task\_environment)

    solutionFound = False

    while agent.frontier:

        node = agent.popFrontier()[1]

        if task\_environment.isGoal(node):

            solutionFound = True

            break

        for child in agent.expand(node):

            agent.pushFrontier(child)

    if solutionFound:

        agent.output(DEST\_PATH)

    else:

        print("No Solution!")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**interface.py**

from typing import List, Tuple

Position = Tuple[int, int, int] # (x, y, z)

Cube = List[Position] # indices: num, values: position, length: CUBE\_LENGTH

CUBE\_LENGTH = 27

DIRECTION = {

  'UP': (0, 0, -1),

  'DOWN': (0, 0, 1),

  'NORTH': (0, -1, 0),

  'SOUTH': (0, 1, 0),

  'WEST': (-1, 0, 0),

  'EAST': (1, 0, 0)

}

**agent.py**

import heapq

from typing import List, Optional

from interface import DIRECTION, Cube, Position

class Agent:

  def \_\_init\_\_(self, task\_environment):

    self.task\_environment = task\_environment

    # keys: Cube state

    # values: [g value, h value, parent Cube, last move direction]

    self.node\_history = {

      hash(tuple(task\_environment.init\_state)): [0, task\_environment.calculateHeuristic(task\_environment.init\_state), None, None]

    }

    self.node\_count = 0

    self.frontier = [(0, task\_environment.init\_state)] # priority queue: (f=g+h, Cube)

  ## FUNCTIONS TO EXPAND NODES ##

  def act(self, cur\_state: Cube, direction: str) -> Optional[Cube]:

    cur\_state = cur\_state[:]

    parent\_hash = hash(tuple(cur\_state))

    offsets = DIRECTION[direction]

    tmp\_tuple = tuple((cur\_state[0][i] + offsets[i] for i in range(len(offsets))))

    if any([e > 2 or e < 0 for e in tmp\_tuple]):

      return None

    index = cur\_state.index(tmp\_tuple)

    cur\_state[index], cur\_state[0] = cur\_state[0], cur\_state[index]

    if hash(tuple(cur\_state)) in self.node\_history.keys():

      return None

    self.node\_count += 1

    self.gValue = self.node\_history[parent\_hash][0] + 1

    self.node\_history[hash(tuple(cur\_state))] = [self.gValue, self.task\_environment.calculateHeuristic(cur\_state), parent\_hash, direction]

    return cur\_state

  def expand(self, node):

    children = []

    for direction in DIRECTION.keys():

      child = self.act(node, direction)

      if child is not None:

        children.append(child)

    return children

  ## FRONTIER MANIPULATION

  def popFrontier(self) -> (int, Cube):

    return heapq.heappop(self.frontier)

  def pushFrontier(self, newNode: Cube):

    record = self.node\_history[hash(tuple(newNode))]

    heapq.heappush(self.frontier, (record[0] + record[1], newNode))

  ## OUTPUT HANDLERS ##

  # calculate for the best solution: the number of steps, total nodes expnded, actions and f values along the path

  def generateSolutions(self):

    solution\_fValues, solution\_actions = [], []

    cur\_hash = hash(tuple(self.task\_environment.goal\_state))

    root\_hash = hash(tuple(self.task\_environment.init\_state))

    while cur\_hash != root\_hash:

      gValue, hValue, cur\_hash, last\_move = self.node\_history[cur\_hash]

      solution\_actions.append(last\_move[0])

      solution\_fValues.append(str(gValue + hValue))

    hValue = self.node\_history[root\_hash][1]

    solution\_fValues.append(str(hValue))

    output = []

    output.append(str(len(solution\_actions)))

    output.append(str(self.node\_count))

    output.append(' '.join(solution\_actions[::-1]))

    output.append(' '.join(solution\_fValues[::-1]))

    return '\n'.join(output)

  # write the result to a file

  def output(self, path):

    f = open(path, "w")

    f.write(self.task\_environment.input\_file + '\n\n')

    f.write(self.generateSolutions())

    f.close()

**task\_environment.py**

from typing import List, Tuple

from interface import CUBE\_LENGTH, Cube, Position

class TaskEnvironment:

  def \_\_init\_\_(self, path):

    f = open(path, "r")

    self.input\_file = f.read()

    f.close()

    self.init\_state, self.goal\_state = self.deserialize(self.input\_file)

  # order = x + 3y + 9z

  def orderToPosition(self, order: int) -> Tuple[int, int, int]:

    x = order % 3

    order //= 3

    y = order % 3

    order //= 3

    z = order % 3

    return (x, y, z)

  def deserialize(self, input\_file: str) -> tuple[Cube, Cube]:

    init\_state = [-1] \* 27

    goal\_state = [-1] \* 27

    nums = input\_file.split()

    for i, num in enumerate(nums[:CUBE\_LENGTH]):

      init\_state[int(num)] = self.orderToPosition(i)

    for i, num in enumerate(nums[CUBE\_LENGTH:]):

      goal\_state[int(num)] = self.orderToPosition(i)

    return init\_state, goal\_state

  def manhattan\_distance(self, pos1: Position, pos2: Position):

    distance = 0

    for i in range(len(pos1)):

      distance += abs(pos1[i] - pos2[i])

    return distance

  def calculateHeuristic(self, state: Cube) -> int:

    heuristic = 0

    for i in range(1, CUBE\_LENGTH):

      heuristic += self.manhattan\_distance(state[i], self.goal\_state[i])

    return heuristic

  def isGoal(self, state: Cube) -> bool:

    return self.calculateHeuristic(state) == 0