Lab 5
Heuristic Search

Oct 12, 2023

Department of Computing
The Hong Kong Polytechnic University

Today's Arrangement

- Assignment 1 Check
- Heuristic Search basics
  - Blind search vs Heuristic search
  - A* Search
- Exercises
- Assignment 2

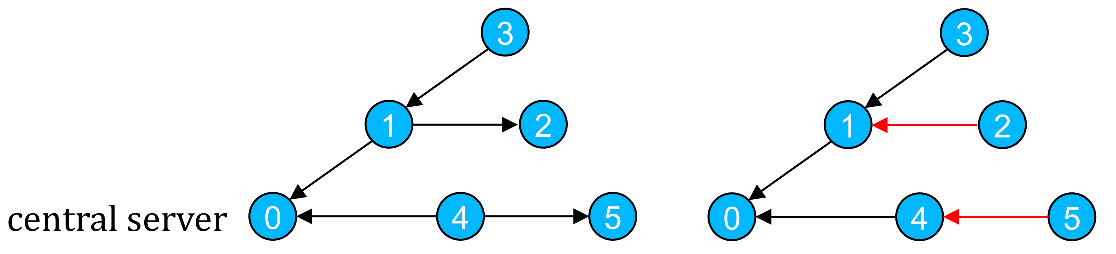
Assignment 1 Check

Assignment 1

- The Information Center has set up a network to allow data uploads between servers. This network can be represented by a set of permitted directed transmissions, where a permitted directed transmission from server a_i to server b_i can be formulated as $[a_i, b_i]$. It's given that $0 \leq a_i, b_i < n, a_i \neq b_i$. There are n $(n \geq 2)$ servers in the network with with server 0 being the central server. The number of directed transmissions is less than n. (There is no cycle in the network.)
- The Information Center plans to update this network. The objective is to ensure the central server (0) can receive data uploads from every other server in the network. Your task is to reorient some of these permitted transmissions so that the central server can receive data from other servers. Return the minimum number of transmission changes required.
- If it's not feasible to enable the central server to receive data from other servers by adjusting the directions of existing permitted transmissions, then return -1.

Assignment 1

- Example 1:

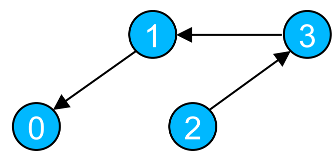


- Demo input: transmissions = [[1, 0], [3, 1], [1, 2], [4,0], [4,5]], num_servers = 6
- Expected output: 2
- Transmission changes are demonstrated in red lines to ensure the central server can receive data from all the other servers.

Assignment 1

- Example 2:

central server



- Demo input: transmissions = [[1, 0], [3, 1], [2, 3]], num_servers = 4
- Expected output: 0
- No transmission route changes are needed, the central server can already receive data from all other servers.

Assignment 1

- Please use the following Python template for submission.
- Your program will be tested on other test cases.

```
import networkx as nx
from typing import List

def solve(transmissions: List[List[int]], num_servers: int) -> int:
    """Your solution to the problem goes in this function.
    Args:
    transmissions (List[List[int]]): The permitted transmission
        between servers,
    e.g., [[1, 0], [3, 1], [1, 2], [4,0], [4,5]]
    num_servers (int): The number of servers in the network,
        e.g., 6
    Returns:
    int: the minimum number of transmission changes required,
        e.g., 2
    """
    return -2
```

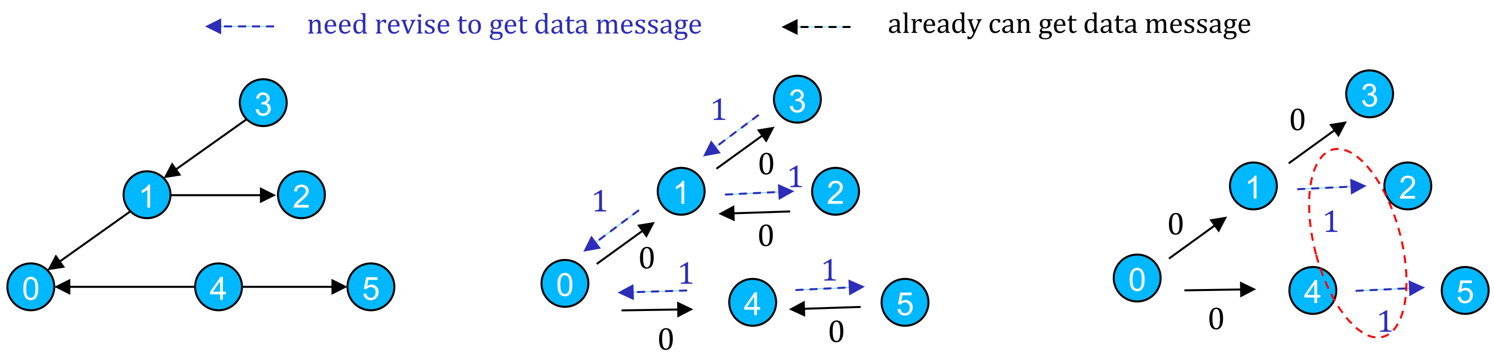
```
# test case 1
transmissions = [[1, 0], [3, 1], [1, 2], [4,0], [4,5]]
num_servers = 6
answer = 2
result = solve(transmissions, num_servers)
assert result == answer, f"Test case 1: expected {answer}, got {result}"
print('Passed test case 1...')

# test case 2
transmissions = [[1, 0], [3, 1], [2, 3]]
num_servers = 4
answer = 0
result = solve(transmissions, num_servers)
assert result == answer, f"Test case 2: expected {answer}, got {result}"
print('Passed test case 2...')
```

Answer

- You can paste your question directly into https://genai.polyu.edu.hk/GPT35 or query GPT through code.
- The expected learning outcomes are:
  1) Practice using ChatGPT to abstract real-world scenarios into algorithmic challenges. (the most important thing is to understand the problem)
  2) Practice having ChatGPT generate understandable code for you.
  3) Practice tailor the code to meet the input and output format requirements.
  4) Practice debugging the code using test cases (you may need to add your own test cases and the most important thing is to understand the algorithm).

Answer

- One of the potential solutions



original graph　　weighted graph　　Dijkstra's algorithm

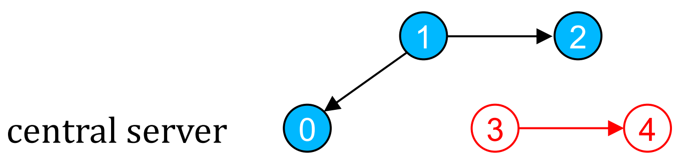
Steps of the algorithm:

- Determine the shortest paths from node 0 to each of the other servers. Before that, we need to:
  - Add reverse edges with a weight of 0 for all edges, indicating no change is needed, for the data delivery.
  - Assign a weight of 1 to all original edges, indicating a change is needed for the data delivery.
- Next, collect edges in the shortest paths and their associated weights (i.e., 1+1=2, indicating change cost)

This idea works because "there is no cycles in the original graph." There will be only one path to each node if the central server can receive data from the server.

Answer

- One of the often-overlooked scenarios

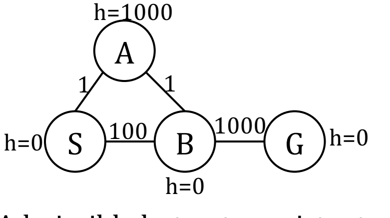


- Demo input: transmissions = [[1, 0], [0, 2], [3, 4]], num_servers=5
- Expected output: -1
- It's not feasible to enable the central server to receive data from other servers by adjusting the directions of existing permitted transmissions.

Heuristic Search basics

Blind vs Heuristic search

- Blind search (Uninformed search): Strategies with no information about the search space, other than distinguishing the goal state from all the others.
- Heuristic Search (Informed search): Strategies that know how far the current state is from the goal state.
  - Heuristic function: Ranks alternatives based on available information.
  - Admissible heuristics: Never overestimate the cost of reaching the goal state.
  - Consistent (monotonic) heuristics: Never overestimate the cost of reaching any internal state.
  - A consistent heuristic is always admissible, but an admissible heuristic does not have to be consistent.



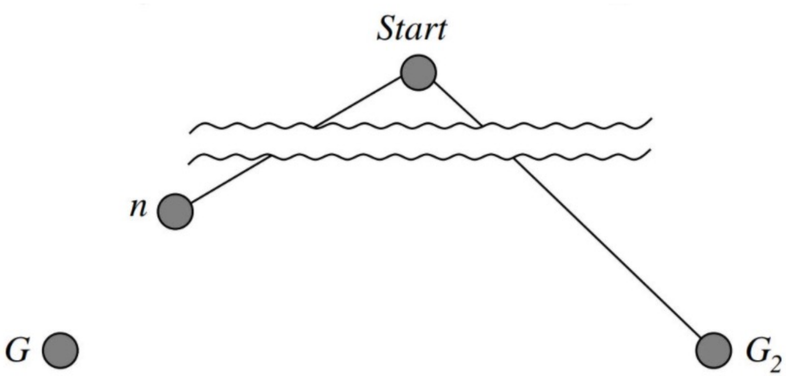
Admissible but not consistent

A* Search

- A quick demo: https://qiao.github.io/PathFinding.js/visual/
- A* Search extends the path that minimizes the Evaluation function: $f(n) = g(n) + h(n)$.
  - $g(n)$: the cost so far to reach the node n.
  - $h(n)$: the estimated cost to goal from the node n.
  - $f(n)$: the estimated total cost of path through the node n to goal.
- Implementation of A* Search:
  - Set "OPEN": Contains those nodes that are candidates for expanding.
  - Set "CLOSED": Contains those nodes that have already been expanded.
  - Each node keeps a pointer to its parent node so that we can determine how it was found.
  - Each node keeps the current best cost from the start to reach the node.

A* Search

- A* Optimality
  - The tree-search version of A* is optimal if h(n) is admissible. (Multiple goals)



A simple proof:
Assume:
- G: Optimal, G_2: Suboptimal
- A* return G_2 before G
- n is the previous node of G on the path from Start to G

Then:
- $g(G) < g(G_2)$ [G is the Optimal]
- $g(G)+h(G) < g(G_2)+h(G_2)$ [h is admissible, so $h(G) = h(G_2) = 0$]
- $g(n)+h(n) <= g(G)+h(G)$ [h is admissible]

Thus:
- $g(n)+h(n) < g(G_2)+h(G_2)$

Which means A* cannot return G_2 before n and G

  - The graph-search version of A* is optimal if h(n) is consistent.
    - Or, with an admissible but inconsistent heuristic, A∗ requires some ways to ensure optimality (such as allow reopening of the closed nodes).
- Learn more about the A* Optimality:
  - Artificial Intelligence: A Modern Approach, 4th US ed. (Section 3.5)

A* Search Pseudocode

Initially, OPEN only contains the start node S, CLOSED is an empty set. Do the following repeatedly:

- If the node N with the lowest rank (by the evaluation function f()) in OPEN is the GOAL:
  - Reconstruct the path reversely from goal to start by following parent pointers
- Else do:
  - Remove N from OPEN and add it to CLOSED
  - For each neighbor M of N:
    - cost = g(N) + distance(N, M)
    - If M is not in either OPEN or CLOSED:
      - Set g(M) to cost
      - Add M to OPEN
      - Set the rank of M in OPEN by the evaluation function f()
      - Set M's parent to N
    - If M is in OPEN and cost is less than g(M)
      - Update the g(M), the rank, and the parent
    - If M is in CLOSED and cost is less than g(M) (This should never happen if you have a consistent admissible heuristic.)
      - Remove M from CLOSED and re-add it to OPEN

Reference: https://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html

A* Search With NetworkX

- astar_path(G, source, target, heuristic=None, weight='weight')
  - Returns a list of nodes in a shortest path between source and target using the A* ("A-star") algorithm.
  - There may be more than one shortest path. This returns only one.
- Check it via official document:

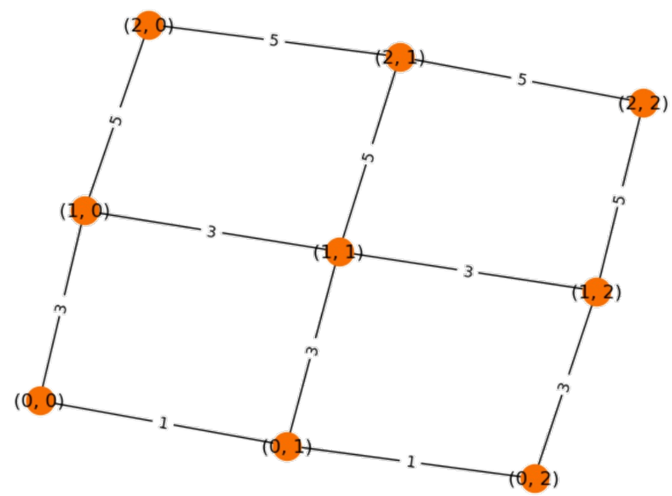
  https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.shortest_paths.astar.astar_path.html

Exercise 1

Exercise 1: A* Search on a grid graph

This is a 3*3 grid graph. The weight of each edge represents the cost moving from one node to another.

Find the shortest path from (0,0) to (2,2) by A* Search.



Requirements:

- You should design a heuristic function for this problem.

Code to generate the above grid graph:

```
G = nx.grid_graph(dim=[3, 3])  # nodes are two-tuples (x,y)
nx.set_edge_attributes(G, {e: e[1][0] * 2 + 1 for e in G.edges()}, "cost")
pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True, node_color="#f86e00")
edge_labels = nx.get_edge_attributes(G, "cost")
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)
plt.show()
```

Exercise 1: A* Search on a grid graph

Example prompt:

Write a Python script that uses the package networkx to find a shortest path by following below instructions.

1. Create a graph by the following code.

```
G = nx.grid_graph(dim=[3, 3])  # nodes are two-tuples (x,y)
nx.set_edge_attributes(G, {e: e[1][0] * 2 + 1 for e in G.edges()}, "cost")
```

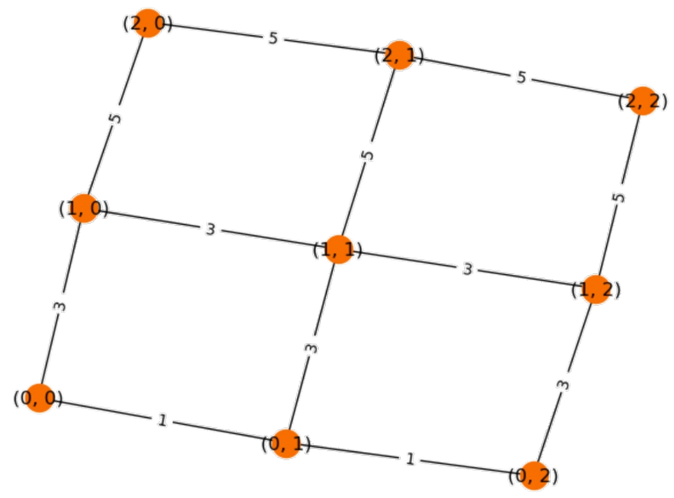
2. Use the Manhattan distance between nodes as the heuristic function and the distance between each pair of adjacent nodes is 1.

3. Apply A* Search to find the shortest path.

Exercise 1: A* Search on a grid graph

Resulting code and answer:

```
import networkx as nx
from heapq import heappop, heappush

# Create the graph
G = nx.grid_graph(dim=[3, 3])
nx.set_edge_attributes(G, {e: e[1][0] * 2 + 1 for e in G.edges()}, "cost")

# Heuristic function using Manhattan distance
def heuristic(node, target):
    return abs(target[0] - node[0]) + abs(target[1] - node[1])

# A* algorithm to find the shortest path
def astar_search(graph, start, target):
    # Priority queue to store nodes to be explored
    queue = []
    heappush(queue, (0, start))

    # Dictionary to store the cost of reaching a node
    cost = {start: 0}

    # Dictionary to store the parent node of each node
    parent = {start: None}

    # A* algorithm
    while queue:
        current_cost, current_node = heappop(queue)

        if current_node == target:
            break

        for neighbor in graph.neighbors(current_node):
            new_cost = cost[current_node] + graph.edges[current_node, neighbor]['cost']
            if neighbor not in cost or new_cost < cost[neighbor]:
                cost[neighbor] = new_cost
                priority = new_cost + heuristic(neighbor, target)
                heappush(queue, (priority, neighbor))
                parent[neighbor] = current_node

    # Reconstruct the path from start to target
    path = []
    current_node = target
    while current_node != start:
        path.append(current_node)
        current_node = parent[current_node]
    path.append(start)
    path.reverse()

    return path

# Test the A* search
start = (0, 0)
target = (2, 2)
shortest_path = astar_search(G, start, target)
print("Shortest Path:", shortest_path)
```

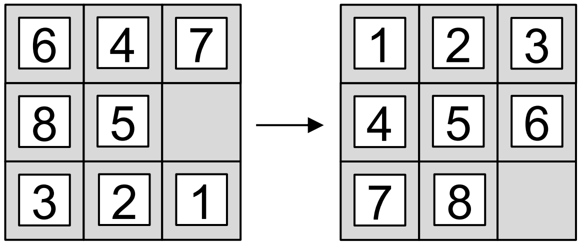



```
Shortest Path: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2)]
```

Exercise 2

Exercise 2: 8-Puzzle Problem

- 8-Puzzle Problem:
  - 3-by-3 grid (9 squares) with eight numbered tiles and one empty space.
  - Slide four adjacent (left, right, above, and below) tiles into the empty space.
  - Objective: Rearrange the tiles so that the numbers are displayed in the goal state.

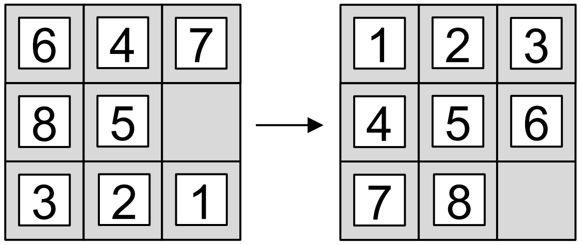


Initial State Goal State

- Task: Solve the 8-Puzzle Problem using A* Search
  - Use proper representation for the 3-by-3 grid and the labels (numbers)
  - Try different heuristic functions (all heuristic functions below are admissible).
    - h1: Number of misplaced tiles
    - h2: Sum of Euclidean distances of the tiles from their goal positions
    - h3: Sum of Manhattan distances of the tiles from their goal positions
    - h4: Number of tiles out of row + Number of tiles out of column

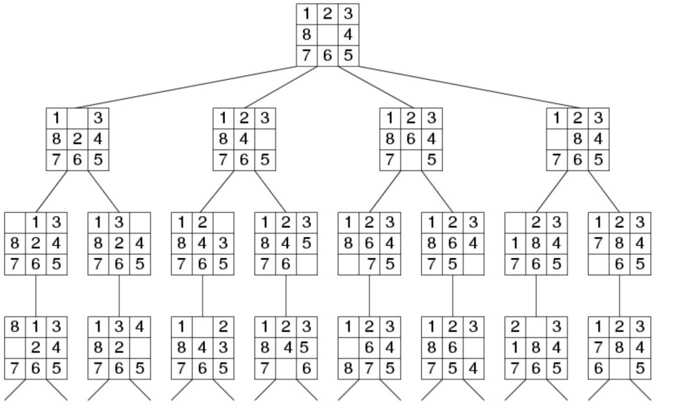
Exercise 2: 8-Puzzle Problem

- 8-Puzzle Problem:
  - 3-by-3 grid (9 squares) with eight numbered tiles and one empty space.
  - Slide four adjacent (left, right, above, and below) tiles into the empty space.
  - Objective: Rearrange the tiles so that the numbers are displayed in the goal state.



Initial State Goal State

- Problem formulation: A* search on a tree
  - Each state represents one kind of positions of tiles
  - Cost from a state to one of its neighbors is 1



Example of search space of 8-Puzzle Problem (first 3 layers)

Exercise 2: 8-Puzzle Problem

- Example prompt (part 1 / 2):

Write a Python script that solves the 8-Puzzle Problem by A* Search. Follow the requirements below:

Requirement 1: Use a tuple instead of a list to store each state because lists are unhashable.

Requirement 2: You need to define several versions of heuristic functions. Follow the name and definition as follows.

h1: Number of misplaced tiles

h2: Sum of Euclidean distances of the tiles from their goal positions

h3: Sum of Manhattan distances of the tiles from their goal positions

h4: Number of tiles out of row + Number of tiles out of column

Requirement 3: If a solution is found after several searches, print the resulting path found by A* Search. If there is no solution, print "No path found".

Exercise 2: 8-Puzzle Problem

- Example prompt (part 2 / 2):

Requirement 4: Record the number of states being examined and print the number at last.

Requirement 5: Use the first heuristic function h1 and the following initial and goal states as a test example. Besides, follow the style to output the states of the found path.

Initial state ("_" is for the empty space):
6 4 7
8 5 _
3 2 1

Goal state ("_" is for the empty space):
1 2 3
4 5 6
7 8 _

Requirement 6: Take your time with an answer, test the availability, and avoid errors before output.

Exercise 2: 8-Puzzle Problem

- Resulting code (part 1 / 3):

```python
import heapq

# Define the initial and goal states
INITIAL_STATE = ((6, 4, 7), (8, 5, '_'), (3, 2, 1))
GOAL_STATE = ((1, 2, 3), (4, 5, 6), (7, 8, '_'))

# Define the heuristic functions
def h1(state):
    # Number of misplaced tiles
    count = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != GOAL_STATE[i][j]:
                count += 1
    return count

def h2(state):
    # Sum of Euclidean distances of the tiles from their goal positions
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != '_':
                x1, y1 = i, j
                x2, y2 = divmod(state[i][j] - 1, 3)
                distance += ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5
    return distance
```

```python
def h3(state):
    # Sum of Manhattan distances of the tiles from their goal positions
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != '_':
                x1, y1 = i, j
                x2, y2 = divmod(state[i][j] - 1, 3)
                distance += abs(x1 - x2) + abs(y1 - y2)
    return distance

def h4(state):
    # Number of tiles out of row + Number of tiles out of column
    count = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != '_':
                x1, y1 = i, j
                x2, y2 = divmod(state[i][j] - 1, 3)
                count += abs(x1 - x2) + abs(y1 - y2)
    return count
```

```python
# Define the Node class to represent each state
class Node:
    def __init__(self, state, g, h, parent=None):
        self.state = state
        self.g = g
        self.h = h
        self.f = g + h
        self.parent = parent

    def __lt__(self, other):
        return self.f < other.f
```

Exercise 2: 8-Puzzle Problem

- Resulting code (part 2 / 3):

```python
# Define the A* Search function
def astar_search(initial_state, goal_state, heuristic):
    # Define the movements for shifting the empty space
    movements = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    movements_text = ['right', 'left', 'down', 'up']

    # Initialize the open and closed sets
    open_set = []
    closed_set = set()
    heapq.heappush(open_set, Node(initial_state, 0, heuristic(initial_state)))

    # Initialize the number of states examined
    states_examined = 0
```

```python
    while open_set:
        # Pop the node with the lowest f value from the open set
        current_node = heapq.heappop(open_set)
        current_state = current_node.state

        # Add the current state to the closed set
        closed_set.add(current_state)

        # Check if the current state is the goal state
        if current_state == goal_state:
            # Construct the path from the root node to the goal node
            path = []
            while current_node:
                path.append(current_node.state)
                current_node = current_node.parent

            # Reverse the path to get the correct order
            path.reverse()

            # Print the path
            for node in path:
                print_state(node)
                print()

            # Print the number of states examined
            print("Number of states examined:", states_examined)

            return

        # Generate the next states
        for movement, movement_text in zip(movements, movements_text):
            new_state = move(current_state, movement)
            if new_state is not None and new_state not in closed_set:
                # Calculate the cost and heuristic values for the new state
                g = current_node.g + 1
                h = heuristic(new_state)

                # Create a new node for the new state
                new_node = Node(new_state, g, h, current_node)

                # Add the new node to the open set
                heapq.heappush(open_set, new_node)

                # Increment the number of states examined
                states_examined += 1

    # If no path is found, print "No path found"
    print("No path found")

    # Print the number of states examined
    print("Number of states examined:", states_examined)
```

Exercise 2: 8-Puzzle Problem

- Resulting code (part 3 / 3):

```
# Define the function to move the empty space
def move(state, movement):
    x, y = find_empty_space(state)
    dx, dy = movement

    # Check if the movement is valid
    if 0 <= x + dx < 3 and 0 <= y + dy < 3:
        # Create a new state by swapping the empty space with a tile
        new_state = list(map(list, state))
        new_state[x][y], new_state[x + dx][y + dy] = new_state[x + dx][y + dy], new_state[x][y]
        return tuple(map(tuple, new_state))

    return None

# Define the function to find the empty space
def find_empty_space(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == '_':
                return i, j

# Define the function to print the state
def print_state(state):
    for row in state:
        print(*row)

# Run the A* Search with the first heuristic function
print("Using heuristic function h1:")
astar_search(INITIAL_STATE, GOAL_STATE, h1)
```

Exercise 2: 8-Puzzle Problem

- Output of the code (h1: Number of misplaced tiles)

```
Using heuristic function h1:
6 4 7      4 _ 7      7 _ 1      7 1 5      1 5 2      1 2 3
8 5 _      6 5 1      4 6 5      _ 4 2      7 4 3      4 5 _
3 2 1      8 3 2      8 3 2      8 6 3      8 _ 6      7 8 6

6 4 7      4 7 _      7 1 _      _ 1 5      1 5 2      1 2 3
8 5 1      6 5 1      4 6 5      7 4 2      7 4 3      4 5 6
3 2 _      8 3 2      8 3 2      8 6 3      _ 8 6      7 8 _

6 4 7      4 7 1      7 1 5      1 _ 5      1 5 2
8 5 1      6 5 _      4 6 _      7 4 2      _ 4 3
3 _ 2      8 3 2      8 3 2      8 6 3      7 8 6

6 4 7      4 7 1      7 1 5      1 5 _      1 5 2
8 5 1      6 _ 5      4 6 2      7 4 2      4 _ 3
_ 3 2      8 3 2      8 3 _      8 6 3      7 8 6

6 4 7      4 7 1      7 1 5      1 5 2      1 _ 2
_ 5 1      _ 6 5      4 6 2      7 4 _      4 5 3
8 3 2      8 3 2      8 _ 3      8 6 3      7 8 6

_ 4 7      _ 7 1      7 1 5      1 5 2      1 2 _
6 5 1      4 6 5      4 _ 2      7 4 3      4 5 3
8 3 2      8 3 2      8 6 3      8 6 _      7 8 6

Number of states examined: 221849
```

The efficiency of h1

The found solution (optimal): 32 states in the path (at least 31 steps to solve the problem)

Exercise 2: 8-Puzzle Problem

- Output of the code (h2: Sum of Euclidean distances of the tiles from their goal positions)

```
Using heuristic function h2:
6 4 7      4 _ 7      4 _ 1      4 1 2      1 _ 2      1 2 3
8 5 _      6 5 1      6 7 2      _ 7 3      4 8 3      4 5 _
3 2 1      8 3 2      8 5 3      6 8 5      7 6 5      7 8 6

6 4 7      4 7 _      4 1 _      4 1 2      1 2 _      1 2 3
8 5 1      6 5 1      6 7 2      7 _ 3      4 8 3      4 5 6
3 2 _      8 3 2      8 5 3      6 8 5      7 6 5      7 8 _

6 4 7      4 7 1      4 1 2      4 1 2      1 2 3
8 5 1      6 5 _      6 7 _      7 8 3      4 8 _
3 _ 2      8 3 2      8 5 3      6 _ 5      7 6 5

6 4 7      4 7 1      4 1 2      4 1 2      1 2 3
8 5 1      6 5 2      6 7 3      7 8 3      4 8 5
_ 3 2      8 3 _      8 5 _      _ 6 5      7 6 _

6 4 7      4 7 1      4 1 2      4 1 2      1 2 3
_ 5 1      6 5 2      6 7 3      _ 8 3      4 8 5
8 3 2      8 _ 3      8 _ 5      7 6 5      7 _ 6

_ 4 7      4 7 1      4 1 2      _ 1 2      1 2 3
6 5 1      6 _ 2      6 7 3      4 8 3      4 _ 5
8 3 2      8 5 3      _ 8 5      7 6 5      7 8 6
```

```
Number of states examined: 68764
```

The efficiency of h2

The found solution (optimal): 32 states in the path (at least 31 steps to solve the problem)

Exercise 2: 8-Puzzle Problem

- Output of the code (h3: Sum of Manhattan distances of the tiles from their goal positions)

```
Using heuristic function h3:
6 4 7      4 _ 7      4 7 1      4 1 2      4 1 2      1 2 3
8 5 _      6 5 1      _ 6 2      8 7 _      5 8 3      4 5 _
3 2 1      8 3 2      8 5 3      5 6 3      7 _ 6      7 8 6

6 4 7      4 7 _      4 7 1      4 1 2      4 1 2      1 2 3
8 5 1      6 5 1      8 6 2      8 7 3      5 _ 3      4 5 6
3 2 _      8 3 2      _ 5 3      5 6 _      7 8 6      7 8 _

6 4 7      4 7 1      4 7 1      4 1 2      4 1 2
8 5 1      6 5 _      8 6 2      8 7 3      _ 5 3
3 _ 2      8 3 2      5 _ 3      5 _ 6      7 8 6

6 4 7      4 7 1      4 7 1      4 1 2      _ 1 2
8 5 1      6 5 2      8 _ 2      8 _ 3      4 5 3
_ 3 2      8 3 _      5 6 3      5 7 6      7 8 6

6 4 7      4 7 1      4 _ 1      4 1 2      1 _ 2
_ 5 1      6 5 2      8 7 2      _ 8 3      4 5 3
8 3 2      8 _ 3      5 6 3      5 7 6      7 8 6

_ 4 7      4 7 1      4 1 _      4 1 2      1 2 _
6 5 1      6 _ 2      8 7 2      5 8 3      4 5 3
8 3 2      8 5 3      5 6 3      _ 7 6      7 8 6
```

```
Number of states examined: 16868
```

The efficiency of h3

The found solution (optimal): 32 states in the path (at least 31 steps to solve the problem)

Exercise 2: 8-Puzzle Problem

- Output of the code (h4: Number of tiles out of row + Number of tiles out of column)

```
Using heuristic function h3:
6 4 7      4 _ 7      4 7 1      4 1 2      4 1 2      1 2 3
8 5 _      6 5 1      _ 6 2      8 7 _      5 8 3      4 5 _
3 2 1      8 3 2      8 5 3      5 6 3      7 _ 6      7 8 6

6 4 7      4 7 _      4 7 1      4 1 2      4 1 2      1 2 3
8 5 1      6 5 1      8 6 2      8 7 3      5 _ 3      4 5 6
3 2 _      8 3 2      _ 5 3      5 6 _      7 8 6      7 8 _

6 4 7      4 7 1      4 7 1      4 1 2      4 1 2
8 5 1      6 5 _      8 6 2      8 7 3      _ 5 3      Number of states examined: 16868
3 _ 2      8 3 2      5 _ 3      5 _ 6      7 8 6

6 4 7      4 7 1      4 7 1      4 1 2      _ 1 2
8 5 1      6 5 2      8 _ 2      8 _ 3      4 5 3
_ 3 2      8 3 _      5 6 3      5 7 6      7 8 6

6 4 7      4 7 1      4 _ 1      4 1 2      1 _ 2
_ 5 1      6 5 2      8 7 2      _ 8 3      4 5 3
8 3 2      8 _ 3      5 6 3      5 7 6      7 8 6

_ 4 7      4 7 1      4 1 _      4 1 2      1 2 _
6 5 1      6 _ 2      8 7 2      5 8 3      4 5 3
8 3 2      8 5 3      5 6 3      _ 7 6      7 8 6
```

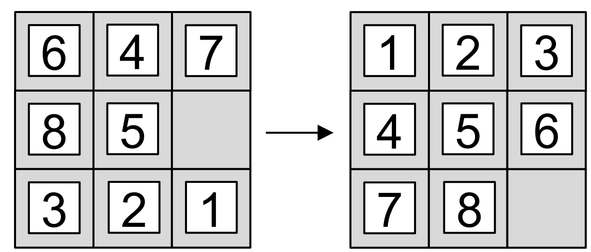
The efficiency of h4

The found solution (optimal): 32 states in the path (at least 31 steps to solve the problem)

Exercise 2: 8-Puzzle Problem

Comparison:

- The heuristic functions:
  - h1: Number of misplaced tiles
  - h2: Sum of Euclidean distances of the tiles from their goal positions
  - h3: Sum of Manhattan distances of the tiles from their goal positions
  - h4: Number of tiles out of row + Number of tiles out of column
- Observations of the values of different heuristic functions :
  - h3 > h2 > h1
  - h4 > h1
- Efficiency (in this case only):
  - h1: 221849 states examined
  - h2: 68764 states examined
  - h3: 16868 states examined
  - h4: 16868 states examined
- Conclusion:
  - Intuitively, among the admissible heuristics, the larger, the more efficient. (Could be verified by experiments.)



Initial State　　Goal State

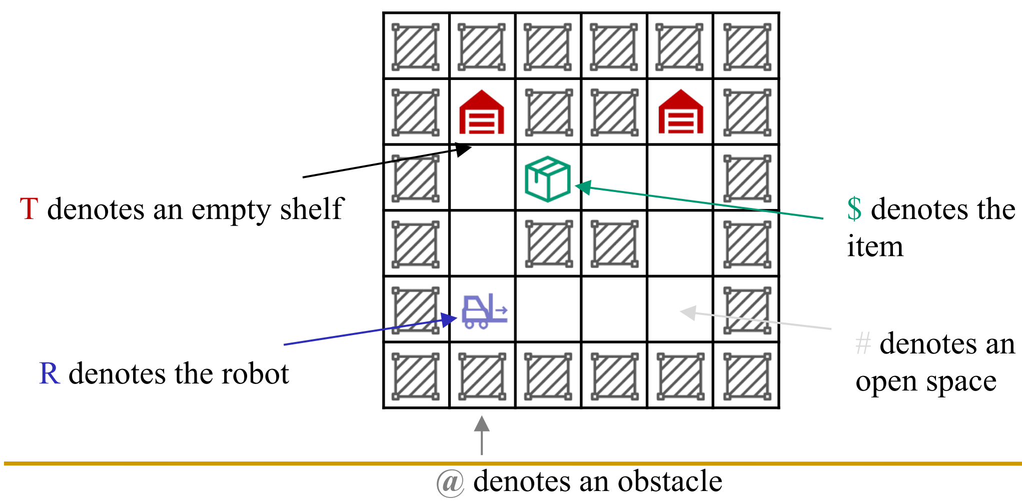
Reference: https://cse.iitk.ac.in/users/cs365/2009/ppt/13jan_Aman.pdf

Assignment 2

Assignment 2

In a warehouse, the robot is used for material handling. Your task is to help the robot to place the item on one of the available shelves. Here, the warehouse layout is represented by an $m \times n$ grid of characters, where the grid includes "R", "#", "@", "$", "T", and :

- "R" stands for the robot.
- "$" denotes the item that needs to be moved. There's only one item on the grid.
- "#" denotes an open space, which both the robot and the item can move onto.
- "@" denotes an obstacle. Neither the robot nor the item can move onto it.
- "T" denotes an empty shelf where the item can be placed. If there are multiple empty shelves, the robot can place the item on any of them. Besides, the robot can move through the empty shelf "T".

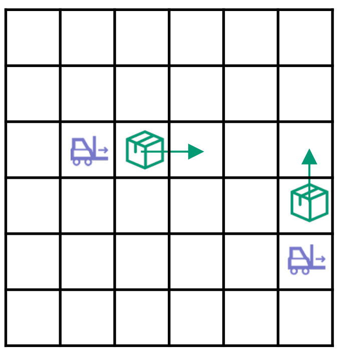


@	@	@	@	@	@
@	T	@	@	T	@
@	#	$	#	#	@
@	#	@	@	#	@
@	R	#	#	#	@
@	@	@	@	@	@

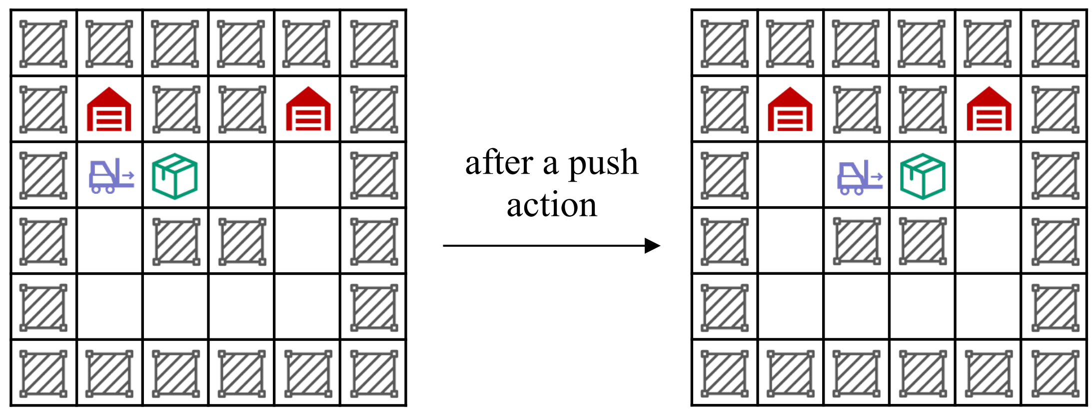
Assignment 2

Available actions of the robot are as follows:

- “Move”: The robot can move up, down, left or right on the grid to reach adjacent open spaces ("#").
- “Push”: The robot can push the item by standing next to it and moving in the direction of the item. The robot cannot move through the item.



The robot can push the item by standing next to it and moving in the direction of the item

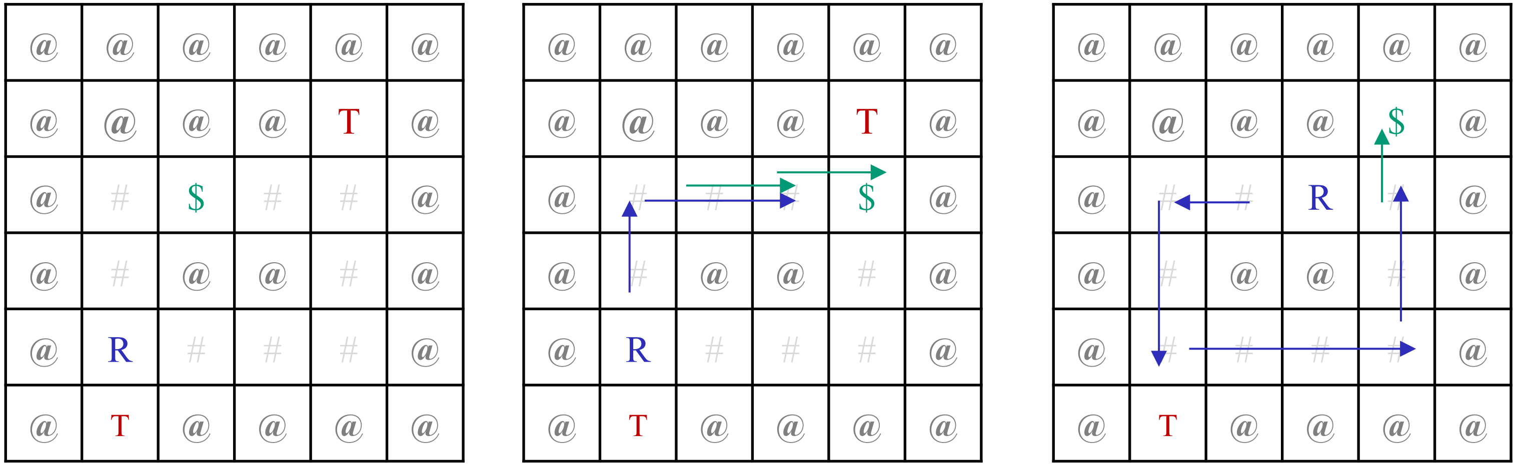


You should return the minimum number of pushes required for the robot to move the item to an empty shelf. If it's impossible to place the item on any shelf, return -1.

- Hint 1: Consider calculating the minimum pushes required to reach different target shelves separately.
- Hint 2: Consider utilizing A* search and heuristically use the Manhattan distance between the item and the target shelf to guide the search.

Assignment 2

- Example 1: a pushing action the robot's movement path



- Demo input: grid = [["@", "@", "@", "@", "@", "@"], ["@", "@", "@", "@", "T", "@"], ["@", "#", "$", "#", "#", "@"], ["@", "#", "@", "@", "#", "@"], ["@", "R", "#", "#", "#", "@"], ["@", "T", "@", "@", "@", "@"]]
- Expected output: 3

Assignment 2

- Example 2:

@	T	@	@	@	@
@	#	@	@	@	@
@	#	#	#	$	@
@	#	@	@	@	@
@	R	#	#	T	@
@	@	@	@	@	@

@	T	@	@	@	@
@	#	@	@	@	@
@	#	#	#	$	@
@	#	@	@	@	@
@	R	#	#	T	@
@	@	@	@	@	@

The robot cannot move the item to the left because it cannot access the @.

- Demo input: grid = [["@, "T", "@", "@", "@", "@"], ["@, "#", "@", "@", "@", "@"], ["@, "#", "#", "#", "$", "@"], ["@, "#", "@", "@", "@", "@"], ["@, "R", "#", "#", "T", "@"], ["@, "@", "@", "@", "@", "@"]]
- It's impossible to place the item on any shelf.
- Expected output: -1

Assignment 2

- Please use the following Python template for submission. (You can copy the code below from lab5-Exercise.ipynb)
- Your program will be tested on other test cases.

```
from typing import List
def solve(grid: List[List[str]]) -> int:
    """Your solution to the problem goes in this function.
    Args:
        grid (List[List[str]]): The warehouse layout, e.g., [["@", "@", "@"], ["@", "R", "$"], ["@", "@", "T"]]
    Returns:
        int: the minimum number of pushes required for the robot to move the item to an empty shelf.
    """
    # Your solution here
    return -2
```

Assignment 2

- Please use the following Python template for submission. (You can copy the code below from lab5-Exercise.ipynb)
- Your program will be tested on other test cases.

```
# test case 1
grid1 = [["@", "@", "@", "@", "@", "@"],
["@", "@", "@", "@", "T", "@"],
["@", "#", "$", "#", "#", "@"],
["@", "#", "@", "@", "#", "@"],
["@", "R", "#", "#", "#", "@"],
["@", "T", "@", "@", "@", "@"]]
answer1 = 3
result1 = solve(grid1)
assert result1 == answer1, f"Test case 1: expected {answer1}, got {result1}"
print('Passed test case 1...')

# test case 2
grid2 = [["@", "T", "@", "@", "@", "@"],
["@", "#", "@", "@", "@", "@"],
["@", "#", "#", "#", "$", "@"],
["@", "#", "@", "@", "@", "@"],
["@", "R", "#", "#", "T", "@"],
["@", "@", "@", "@", "@", "@"]]
answer2 = -1
result2 = solve(grid2)
assert result2 == answer2, f"Test case 2: expected {answer2}, got {result2}"
print('Passed test case 2...')
```