# 18-462/662 Homework #1

**Carnegie Mellon University**
Department of Electrical and Computer Engineering
*Due date: February 4th, 5pm*

*Note: NO late homework will be accepted.*

## Part I: Theoretical Questions (40 points – 1 point for each question unless stated otherwise)

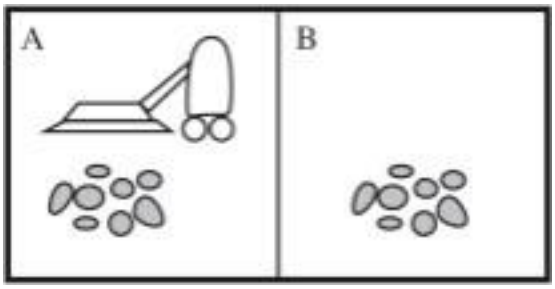
1. For each of the following assertions, say whether it is true or false and support your answer with examples or counterexamples where appropriate.
    a. An agent that senses only partial information about the state cannot be perfectly rational.
    b. There exist task environments in which no pure reflex agent can behave rationally.
    c. There exists a task environment in which every agent is rational.
    d. The input to an agent program is the same as the input to the agent function.
    e. Every agent function is implementable by some program/machine combination.
    f. Suppose an agent selects its action uniformly at random from the set of possible actions. There exists a deterministic task environment in which this agent is rational.
    g. It is possible for a given agent to be perfectly rational in two distinct task environments.
    h. Every agent is rational in an unobservable environment.
    i. A perfectly rational poker-playing agent never loses.

2. For each of the following activities, give a PEAS description of the task environment and characterize its properties. [*Fully vs Partially Observable, Single agent vs Multiagent, Deterministic vs Stochastic, Episodic vs Sequential, Static vs Dynamic, Discrete vs Continuous, Known vs Unknown*]
    a. Playing soccer.
    b. Exploring the subsurface oceans of Titan.
    c. Shopping for used AI books on the Internet.
    d. Playing a tennis match.
    e. Practicing tennis against a wall.
    f. Performing a high jump.
    g. Knitting a sweater.
    h. Bidding on an item at an auction.

3. This question explores the differences between agent functions and agent programs.

   a. Can there be more than one agent program that implements a given agent function? Give an example, or show why one is not possible.
   b. Are there agent functions that cannot be implemented by any agent program?
   c. Given a fixed machine architecture, does each agent program implement exactly one agent function?
   d. Given an architecture with n bits of storage, how many different possible agent programs are there?
   e. Suppose we keep the agent program fixed but speed up the machine by a factor of two. Does that change the agent function?

4. Implement a performance-measuring environment simulator for the vacuum-cleaner world depicted in following figure. Your implementation should be modular so that the sensors, actuators, and environment characteristics (*size, shape, dirt placement, etc*.) can be changed easily. **(6 points)**




A vacuum-cleaner world with just two locations.

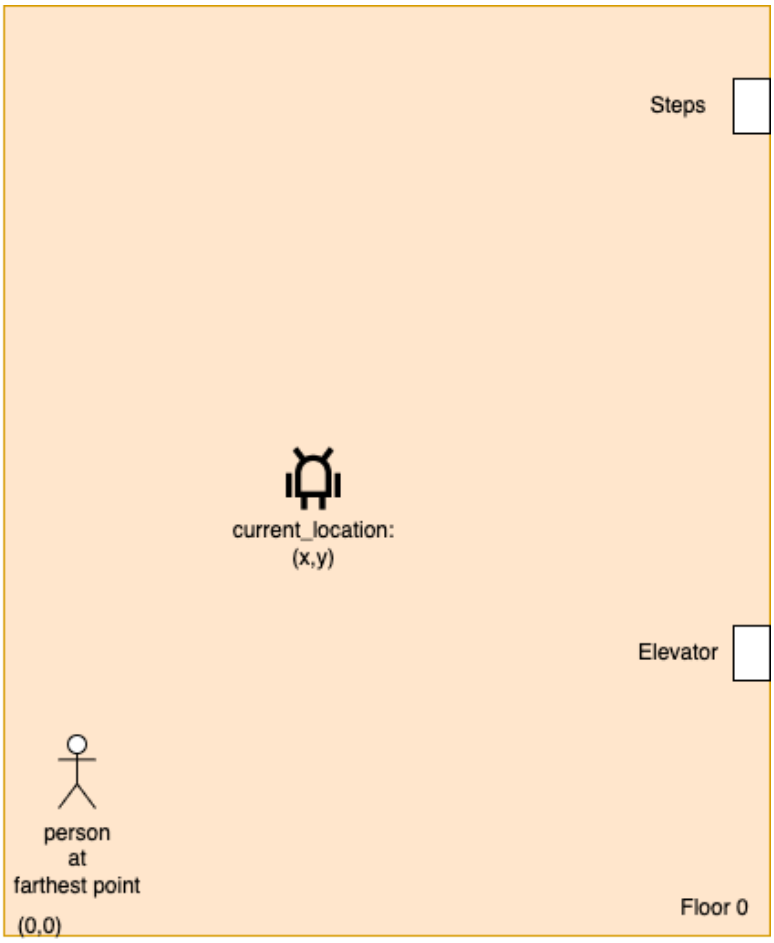
5. Consider a modified version of the vacuum environment in the Question 4, in which the geography of the environment—its extent, boundaries, and obstacles—is unknown, as is the initial dirt configuration. (The agent can go *Up* and *Down* as well as *Left* and *Right)* **(12 points – 3 points each)**

   a. Can a simple reflex agent be perfectly rational for this environment? Explain.
   b. Can a simple reflex agent with a *randomized* agent function outperform a simple reflex agent? Design such an agent and measure its performance on several environments.
   c. Can you design an environment in which your randomized agent will perform poorly? Show your results.
   d. Can a reflex agent with state outperform a simple reflex agent? Design such an agent and measure its performance on several environments. Can you design a rational agent of this type?

# Part II: Python Problems (60 points)

## 1. Reflex Agent Implementation (10 points)

You work for a company that designs autonomous robots for security purposes. Your latest project is to develop a reflex agent for a security robot that is tasked with securing a large, multi-floor building. The robot is equipped with sensors that detect any intruders, fires, gas leaks, and floods. It needs to take appropriate action based on the specific situation to secure the building.




The robot is able to move between floors using the elevators and staircases. It moves at a speed of 3 m/s. It needs to choose the quickest route to reach the location of the emergency and take appropriate action.

Implement a reflex agent for the security robot using the following rules (in order of priority):

1. If an intruder is detected, the robot should emit an alarm and move to the location of the intruder to secure the area.
2. If a fire is detected, the robot should call the fire department and evacuate the building.
3. If a gas leak is detected, the robot should turn off the gas supply and evacuate the building.
4. If a flood is detected, the robot should turn off the electricity supply and evacuate the building.
5. If multiple emergencies are detected, the robot should prioritize the emergencies as follows: intruders, fires, gas leaks, and floods, in that order.
6. If no emergency is detected, the robot should continue patrolling the building.

When moving between floors, the robot should choose the quickest route using the elevators or staircases. The elevator takes 30 seconds to move between floors, while using the staircases takes 60 seconds. The robot should choose the quickest route based on the current floor and the location(floor) of the emergency.

The floor plan for all the floors in the building is the same. A Man sitting on the farthest corner of the floor facing North, Would have to walk 1km straight and then turn 90 degrees right to walk for another 1km to find the steps. Although, if from the location of the steps, he walks 500m towards West, then 0.9km towards South and further 500 m East, he would reach the elevator.
Write a Python function security_robot that returns a string that describes the action taken by the robot and the quickest route:

1. "Emit alarm and move to location of intruder on floor X using Y" if an intruder is detected. The X should be the floor number where the intruder is located.
2. "Call fire department and evacuate building on floor X using Y" if a fire is detected. The X should be the floor number where the fire is located.
3. "Turn off gas supply and evacuate building on floor X using Y" if a gas leak is detected. The X should be the floor number where the gas leak is located.
4. "Turn off electricity supply and evacuate building on floor X using Y" if a flood is detected. The X should be the floor number where the flood is located.
5. "Continue patrolling the building on floor X" if no emergency is detected. The X should be the current floor of the robot.

The Y should be the quickest route chosen based on the speed of the robot, the distance to each route (steps/elevator) and the time taken to move between floors using the route.

If multiple emergencies are detected, the function should return the action for the highest priority emergency. The floor number should be the floor with the highest priority emergency.

For example: If an intruder is detected at 5th floor and the robot is currently 20m away from stairs and 100m away from elevator, your agent should give the following message:

"Emit alarm and move to location of intruder on floor 5 using elevator"

Points to keep in mind:

- Unless specified, the reflex agent here would mean a simple reflex agent.
- The location of the robot at the time will be floor 0 and the parameter current_floor refers to the floor of emergency.
- You can assume the distance / time taken to travel from the steps/elevators to the exact location of emergency on the floor would be 0. (which means the bot needs to reach just the floor of emergency and not the exact location of emergency on that floor).
- The floor is rectangular in shape and the robot may move in any direction from its initial position (current_location) within the floor.
- The floor plan details and other specifications mentioned in the question are just for reference and visualizing and do not specify the path a robot should take.

**2. Model-Based Agent in a Frozen Lake (10 points)**

See `hw1_model_based_agent.py` on Canvas.


In this section, we will use the Frozen Lake OpenAI Gym environment and build a model-based agent that interacts with it. First, make sure you have the necessary installations (run these commands in the terminal: `pip install gym==0.26.2`, `pip install pygame`).

Run the file `hw1_model_based_agent.py` as is. You should be able to view an animation of our agent traveling for a few steps before falling through a hole in the ice. The environment then resets and the game is repeated. If this step does not work, follow any terminal instructions and install the recommended libraries. Note: don't use Jupyter Notebook or Google Colab for this problem.

Your goal in this section is to create a set of condition-action rules so that the agent moves freely through the environment without ever falling through the ice after learning the environment. The environment is partially observable, meaning that the agent doesn't know which squares contain holes or ice until it has walked on them. Keep in mind that your solution should work for ANY (8x8) map that it is tested on, though we have provided a map for you to get started with.

Notes:

- **Do NOT use information from the provided `map` in your solution.** Your agent should work in any environment. Further, your agent does not have access to `map`, and it should keep track of its own map of the environment that it adds to at each iteration of the for-loop. You CAN assume however that the `map` will

always be 8x8.

- Your agent should keep track of where it has been before, where the holes and ice spots are and after a learning process, it should have full knowledge of the environment. Once it has full knowledge of the environment, it should move through the environment and never fall through any holes.
- Once the agent has fallen through a hole in the environment, it shouldn't ever fall through that hole again.
- FrozenLake-v1 docs: https://www.gymlibrary.dev/environments/toy_text/frozen_lake/
- You might want to make use of an explore-exploit ratio while your agent is building the map.

What we should see in your submission:

- Different functions for handling two cases: 1) when the environment is not yet fully known, and 2) when the environment is fully known and the agent no longer needs to incrementally build its knowledge of the environment.
- When we run your Python file, the agent must explore its surroundings and the game must reset every time the agent discovers a new hole. When incrementally building the map, the agent should prefer to travel to squares it hasn't seen before. Keep in mind that the agent needs to sometimes travel to squares it has already seen before in order to be able to travel to new ones. The following functions will be useful: `numpy.random.randint()` and `numpy.random.uniform()`.
- The "incremental map-building" process does not need to be completed within a certain number of iterations. We will check the logic in your code instead. ● **Document your solution**
    - Add comments to your code to help us understand your thought process. Add function declarations (including a description and the types/meaning of the inputs and outputs).
    - Add images or videos that help demonstrate how your solution works. ○ A description of any design choices you made in your code. What makes your code unique and better than other solutions?

**3. Utility-Based Agent Implementation in a Frozen lake (10 points)**

See `hw1_utility_based_agent.py` on Canvas.


This part builds off of the previous model-based agent problem. The difference here is that there is an objective: reach the reward at the other end of the frozen lake using the shortest possible feasible (i.e., no holes) path. Since with this form of intelligent agent, the world is fully-observable, feel free to use the provided `map` to create an adjacency matrix to build your graph. Keep in mind that your solution should work for ANY (8x8) map that it is tested on. (The start location of the agent will always be the same but the locations of the holes and the reward will not always be the same. The map will always be 8x8.)

What we should see in your submission:

- A data structure that encodes your graph (with nodes and edges).
- An implementation of Dijkstra's algorithm that is invariant to changes in the start location of the agent and to changes in the location of the reward.
- When we run your Python file, the agent must travel from its start location to the reward location using the shortest path. It must not fall into any of the holes. Note that we will test your solution on a map different from the one provided in the .py file.
- **Document your solution**
  - Add comments to your code to help us understand your thought process. Add function declarations (including a description and the types/meaning of the inputs and outputs).

- Add images or videos that help demonstrate how your solution works.
- A description of any design choices you made in your code. What makes your code unique and better than other solutions?

**4. Utility-Based Personalized Recommendation Agent (10 points)**

Design a utility-based agent that recommends products to users based on their preferences and past interactions. The goal is to calculate utility scores for products based on specific
factors and recommend the top n products that maximize the user's satisfaction. See Skeleton code: hw1_product.py on canvas

What we should see in your submission:

1. Implement a function recommend_products(user_data, products, n):
   - user_data: A dictionary containing user preferences, past ratings, and interaction history.

     {

     "ratings": {"product1": 5, "product2": 3},

     "preferences": {"category": ["electronics",

     "books"]}

     }
   - products: A list of dictionaries, where each dictionary represents a product with attributes like category, price, and rating.

     [

     {"id": "product1", "category": "electronics", "price": 499, "rating": 4.5},

     {"id": "product2", "category": "books", "price": 15, "rating": 4.0}

     ]
   - n: The number of recommendations to return.
2. Compute utility scores for each product based on similarity to user preferences and past ratings. Justify your choices in your submission.
3. Sort the products by their utility scores and return the top n recommendations
4. Provide a clear explanation of:
   - The logic behind your utility calculation.
   - Why you chose specific factors and their weights.
   - Examples or test cases showing the input, the calculated utility scores, and the final recommendations.

**5. Warehouse Robot (10 points)**

For this problem, imagine you are in charge of a warehouse agent that tries to place stuffs onto stacks with minimal cost. However, the true scenario is complex and contains too many variables. For the sake of an interesting problem, let's restrict the problem to have the following settings.

- Each item you are rectangular or cube like objects of the same width and depth, but varying height varying from unit 1 to 10.
- The weight is directly tied to the height and has range from 1 unit to 10 unit. (If weight is mentioned in the problem).
- You have fixed height limit in stacks. The stacks have varying height limit.
- The number of stacks is from 3 to 5.
- There is only one robot, and you can only place thing onto stack sequentially.

Your warehouse bot is in dire need for some smart algorithms to place items to stacks. Here are some questions for you to think about and write some simple code under very controlled settings. For the following four questions, you are asked to find the total cost of placing all the items onto the stack (plus in the last question, we will remove one item, so the total cost of adding and removing).

1. To get you started, let's begin by the simplest case where we ignore the weights. The cost of placing an item onto the stack equals to the existing height of the stack prior to placing the current item. You are given a simple construct in the file *warehousebotwithbubbles.py*. Please start your implementation in function *question1*.

2. Now let's take this problem up a notch. We modify the cost to be more realistic: the cost of placing an item onto the stack equals to the existing height of the stack prior to placing the current item times the weight (height, since they are tied in our setting) of the placing item. Please implement your code in function *question2*.

3. In reality, the items are coming in unpredictable order. You next move should be trying to come up with some method trying to cope with this unpredictability. The cost function is still the same from question 2. Please implement your code in function *question3*.

4. One last step of the question. Now, let's consider the case where we not only need to add items to our stacks, we also want to remove them. To make your life easier, we now assume that there is only one stack, and after all the items are placed onto the stack, only one item will be removed. Also, you will be given a list of items to add and one item to be removed (look at the function signature). All of the items are

known for you to decide your placement method. We also provide the item to be removed.

However, the goal for this question is different. We want you to find a solution that can achieve the minimal combined costs of placing all the items on the stack plus removing that one item. The cost of removing one item is provided as follows: the cost of removing all the items above the item of interest (inclusive the item want to remove) plus the cost of placing the items above the item of interest (exclusive the item want to remove) back to the stack.

Here to make your life easier, we make some critical assumptions:

- The cost of removing a group of items from top of the stack equals to (the sum of heights of the items removing) times the height of stack beneath the group of items.
- The cost of adding a group of items to the stack equals to (the sum of heights of the items adding) times the height of the stack prior to the addition of items.

Please show your implementation in *question4*.

**Note: Some of these problems are very difficult to find the optimal solution (if that exists). Although we love to see your fancy Dynamic Programming Algorithms or Integer Linear Programming Algorithms, you don't have to do that. You will be given full score in this problem as long as your cost reported equals or beats the greedy algorithms for these problems. Therefore, relax and try to begin with the most intuitive ways to solve these problems.**

In the provided code, we provide the construct for Item and Stack. Also, we have the function signature layed out. In the bottom, we also have a simple test case for you to do some sanity check with some targets in the comments. We have some additional testcases to grade your performance.

# 6. Airport Path Finding Assignment

## Objective

Develop a program that finds the shortest path between two airports considering aircraft range limitations using both Dijkstra's and A* algorithms.

## Input Data

- File: airport.geojson (contains airport data with coordinates)
- User inputs:
  - Departure airport IATA code
  - Destination airport IATA code
  - Maximum aircraft range in kilometers

## Requirements

### 1. Data Processing

- Parse airport.geojson to extract:
  - Coordinates [longitude, latitude]
  - IATA codes
  - Airport names

### Constraints

- Aircraft must refuel at airports when the next leg exceeds maximum range
- All flight segments must be shorter than or equal to the maximum range
- Invalid routes (segments > max_range) should be excluded from path finding

### 2. Distance Calculation

Implement the Haversine formula to compute the distance between two airports using their latitude and longitude :

***def haversine_distance(lon1, lat1, lon2, lat2):***
*R = 6371  # Earth's radius in kilometers*

```
    # Convert to radians
    lat1, lon1, lat2, lon2 = map(math.radians, [lat1, lon1, lat2, lon2])

    # Haversine formula
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = math.sin(dlat/2)**2 + math.cos(lat1) * math.cos(lat2) * math.sin(dlon/2)**2
    c = 2 * math.asin(math.sqrt(a))

    return R * c  # Distance in kilometers
```

### 3. Algorithm Implementation

Implement both:

```
def dijkstra(airports, start, end,
    max_range): """
    Args:
        airports: Dict of airport data
        start: Starting airport IATA
        code
        end: Destination airport IATA code
        max_range: Maximum flight range in kilometers
    Returns:
        (path, distance): Tuple of airport list and total distance
    """

def astar(airports, start, end,
    max_range): """
    Same interface as dijkstra()
    Use straight-line distance as heuristic
    Consider max_range constraint
    """
```

### 4. Program Interface

```
def find_route(departure_iata, destination_iata, max_range,
    algorithm='dijkstra'): """
    Args:
        departure_iata: Starting airport code
        destination_iata: Destination airport code
        max_range: Maximum flight range in kilometers
        algorithm: 'dijkstra' or 'astar'
```

"""

## Example Usage

find_route("CDG", "NRT", 5000, "dijkstra")

## Example Output

**Using Dijkstra's Algorithm:**

Departure: CDG (Charles de Gaulle Int'l)
Destination: NRT (Narita Int'l)
Algorithm: dijkstra
Total Distance: 9825.7 km
Route:
1. CDG (Charles de Gaulle Int'l)
2. SGC (Surgut)
3. VVO (Vladivostok Int'l)
4. NRT (Narita Int'l)

**Using A* Algorithm:**

Departure: CDG (Charles de Gaulle Int'l)
Destination: NRT (Narita Int'l)
Algorithm: astar
Total Distance: 9825.7 km
Route:
1. CDG (Charles de Gaulle Int'l)
2. SGC (Surgut)
3. VVO (Vladivostok Int'l)
4. NRT (Narita Int'l)