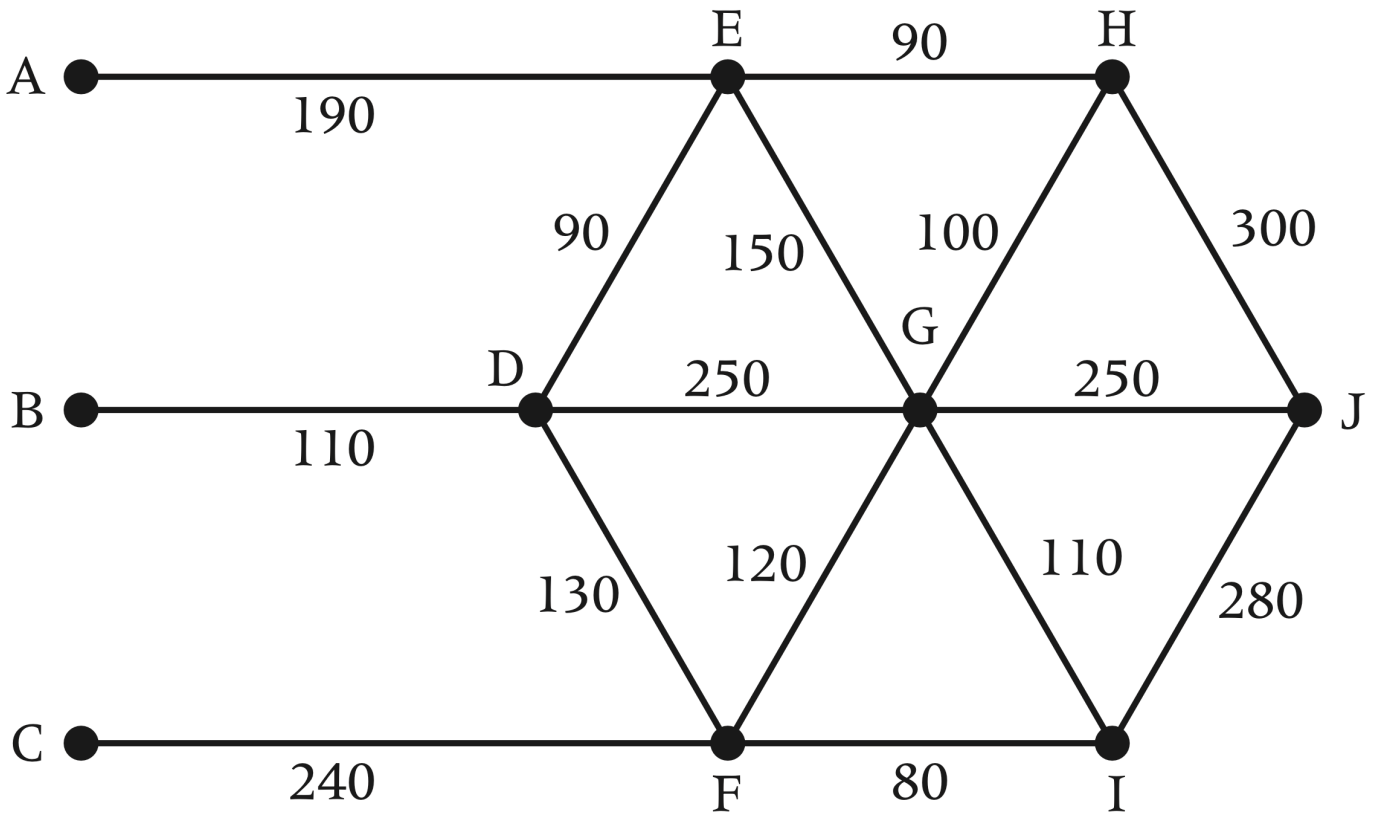
Assignment Shortest Path and Search Based Optimization

Part 1 - Dijkstra's Shortest Path Algorithm

Three different fire stations are located at points A, B, and C. If a fire breaks out at point J, from which station is the travel time to reach the fire the shortest? Use Dijkstra's algorithm. Trial-and-error answers will not be accepted.



Part 2 - Exploring A* Pathfinding Heuristics & Parameters

The A* algorithm is a best-first search algorithm that finds the least-cost path from a start node to a goal node. It uses a heuristic function h(n) to estimate the cost from node n to the goal, combined with the actual cost g(n) from the start to node n.

The algorithm's effectiveness heavily depends on the heuristic function chosen. Different heuristics can dramatically affect:

- **Search efficiency** (nodes explored)

- **Path optimality** (shortest path found)
- **Computational time**
- **Memory usage**

Objective

- Use the provided A* demo to experiment with heuristics and parameters.
- Observe how different configurations influence the search process and performance.

Run de Astar_assignment.py file to start the demo.

The demo makes use of the **pygame** library.

To install pygame:

pip

```
pip install pygame
```

Anaconda

```
conda install conda-forge::pygame
```

Task 1: Implement Different Heuristics

Modify the provided demo to support multiple heuristic functions. Implement the following:

1. **Euclidean Distance** (already implemented)

```
def euclidean_heuristic(pos1, pos2):
    return math.sqrt((pos1[0] - pos2[0])**2 + (pos1[1] - pos2[1])**2)
```

2. **Manhattan Distance**

```
def manhattan_heuristic(pos1, pos2):
    return abs(pos1[0] - pos2[0]) + abs(pos1[1] - pos2[1])
```

3. **Chebyshev Distance**

```
def chebyshev_heuristic(pos1, pos2):
    return max(abs(pos1[0] - pos2[0]), abs(pos1[1] - pos2[1]))
```

4. **Weighted Euclidean** (experiment with weights 0.5, 1.0, 1.5, 2.0)

```python
def weighted_euclidean_heuristic(pos1, pos2, weight=1.0):
    return weight * math.sqrt((pos1[0] - pos2[0])**2 + (pos1[1] - pos2[1])**2)
```

Task 2: Performance Comparison

For each heuristic, run the algorithm on the same set of test scenarios and collect the following metrics:

Metrics to Track:

- Path length found
- Execution time
- Whether optimal path was found
- Number of nodes explored

Test Scenarios: make sure there is always a possible path from start to finish

1. **Empty Grid**: Start (5,5) → End (45,25)
2. **Simple Barrier**: Single vertical wall in the middle
3. **Maze**: Complex maze pattern
4. **Scattered Obstacles**: Random obstacles covering about 20% of grid

Task 3: Design Obstacle Patterns

Create and test the following obstacle configurations:

1. **Linear Barriers**
    - Single horizontal wall
    - Single vertical wall
    - Diagonal wall
    - L-shaped barrier
2. **Geometric Shapes**
    - Hollow rectangle/square
    - Filled circle/ellipse
    - Star pattern
    - Spiral pattern
3. **Maze Types**
    - Simple corridor maze
    - Room-and-corridor layout
    - Open maze with multiple paths

Task 4: Pattern Impact Analysis

For each obstacle pattern, analyze:

- Which heuristic performs best (fewest nodes explored)?
- Which patterns cause the most/least exploration?
- Are there patterns where certain heuristics fail or excel, explain?

Questions to Answer:

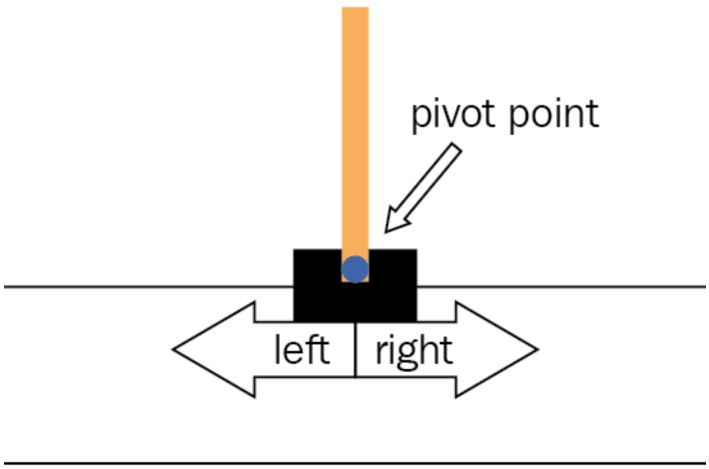
1. Why does Manhattan distance work better/worse in certain scenarios?
2. When might you prefer Chebyshev distance over Euclidean?
3. How does increasing the heuristic weight affect exploration vs. optimality?

Part 3 - Search Based Optimization - Cart Pole

Objective

You will use search based optimization algorithms in order to solve the OpenAI gym cartpole-V1 environment.

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.



More information about the Cartpole environment:

https://gymnasium.farama.org/environments/classic_control/cart_pole/

Information about how to install the Gymnasium environment in Python:

Pip

```
pip install gymnasium
```

Anaconda

```
conda install conda-forge::gymnasium
```

For this assignment you can use the IDE of your choice (Pycharm, VS Code, Cursor, Jupyter notebook, ...).

1. Random action based control

Run the cartpole for 1000 episodes. Control the cartpole by choosing random actions.

- For each episode register the number of successful timesteps.
- Plot a histogram of the number of successful timesteps. You can use the Seaborn distplot function for visualizing this histogram.
- What is the average number of successful timesteps you achieve by taking random actions?

2. Angle based action control

Use the information about the angle of the pole to decide on the best action to take. If the pole is falling to the right, push the cart to the right. If the pole is falling to the left, push the cart to the left.

Again, run the cartpole for 1000 episodes.

- For each episode register the number of successful timesteps.
- Plot a histogram of the number of successful timesteps. You can use the Seaborn distplot function for visualizing this histogram.
- What is the average number of successful timesteps you achieved over these 1000 episodes?

3. Random search based control

First define a vector of weights. Each of the weights corresponds to a one cartpole observation variable. The action will be based on the outcome of a linear combination of weights and observations.

$$\text{evaluation} = [w_1, w_2, w_3, w_4] \cdot \begin{bmatrix} \text{cart position} \\ \text{cart velocity} \\ \text{pole angle} \\ \text{angular velocity} \end{bmatrix}$$

Choose the action based on the following rule:

$$\text{Action} = \begin{cases} evaluation < 0 & \text{push left} \\ evaluation \geq 0 & \text{push right} \end{cases}$$

- Run 1000 iteration. For each iteration set the weights to a random value (in a realistic range) and evaluate these weights by running 20 episodes. The achieved reward for one iteration (and thus a weight vector) is the average reward over the 20 episodes. Store the history of iterations (average reward and the weight vector).
- Which weight vector yields the highest average reward?
- Now test these best weights over 1000 episodes. What is the average reward you have achieved?
- Can you read from the weights which observation variables are important and which ones are not important in deciding which action to take?
- Make a 3D graph (scatterplot) showing which weight vectors achieve an average reward larger than 100 and which ones do not. To plot this graph you will put the 3 most important weights on the x,y,z axis. A red color indicates an average reward higher or equal than 100, black indicates an average reward lower than 100.

4. Hill climbing

Implement the hill climbing algorithm to find a good weight vector. You start with a random weight vector. Each iteration you slightly change the weight vector by adding some noise (can be taken from a normal distribution with mean 0 and standard deviation sigma. If the new weight vector results in a higher reward (for example over 20 episodes) then you can keep the new weight vector. Otherwise you stick to the old one.

- Run 1000 iteration. For each iteration set the weights to a random value (in a realistic range) and evaluate these weights by running 20 episodes. The achieved reward for one iteration (and thus a weight vector) is the average reward over the 20 episodes. Store the history of iterations (average reward and the weight vector).
- Which weight vector yields the highest average reward?
- Now test these best weights over 1000 episodes. What is the average reward you have achieved? Plot the histogram of the rewards.

5. Simulated annealing

Now change the hill climbing algorithm to simulated annealing. Also here you will run 1000 iterations with 20 episodes per iteration.

- Set a starting temperature and a cooling rate.
- Randomly change the weights and evaluate the current reward Rt.
- If the current reward Rt is higher than the previous reward Rt-1 keep the new weights (hill climbing). $p = e^{-\frac{\Delta}{T}}$ where $\Delta = R_{t-1} - R_t$
- Gradually lower the temperature.
- Run 1000 iteration. For each iteration set the weights to a random value (in a realistic range) and evaluate these weights by running 20 episodes. The achieved reward for one iteration (and thus a weight vector) is the average reward over the 20 episodes. Store the history of iterations (average reward and the weight vector).

- Which weight vector yields the highest average reward?
- Now test these best weights over 1000 episodes. What is the average reward you have achieved? Plot the histogram of the rewards.

1.5 Adaptive noise scaling

Modify the simulated annealing algorithm in the following way:

Reduce the standard deviation of the distribution we sample the noise from when the new reward is higher than the previous reward. Otherwise increase the standard deviation. For example you can half or double the standard deviation. It's up to you to find a good value for the scaling factor.

- Run 1000 iteration. For each iteration set the weights to a random value (in a realistic range) and evaluate these weights by running 20 episodes. The achieved reward for one iteration (and thus a weight vector) is the average reward over the 20 episodes. Store the history of iterations (average reward and the weight vector).
- Which weight vector yields the highest average reward?
- Now test these best weights over 1000 episodes. What is the average reward you have achieved? Plot the histogram of the rewards.

1.6 Extensions and conclusions

- How do these different search algorithms compare to or differ from each other in terms of reward and computational efficiency?
- Does it make sense to increase the number of observation variables by deriving new observations from the existing ones? Think of it as feature expansion in machine learning.
- Consider the MountainCar-v0 environment. Argument whether or not it is possible to solve by means of these search based optimization techniques.