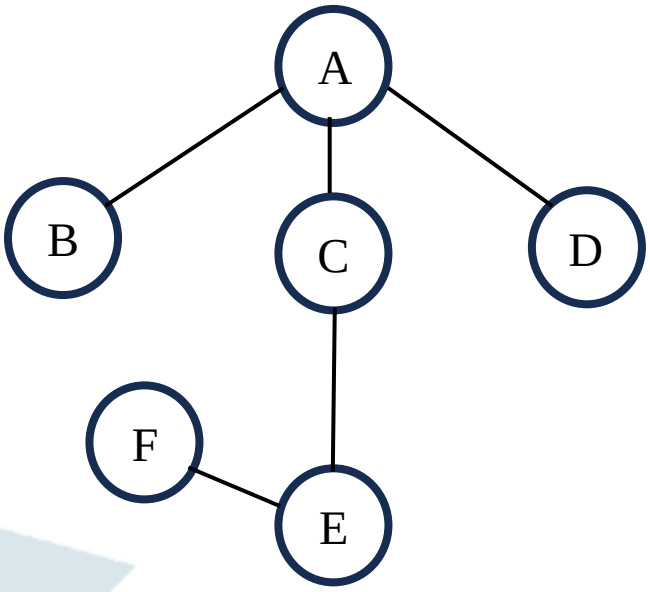
Introduction to Artificial Intelligence
Lecture 4: A* Algorithm

Revision Exercises

Exercise 1:
Convert the following graph to a dictionary in python.



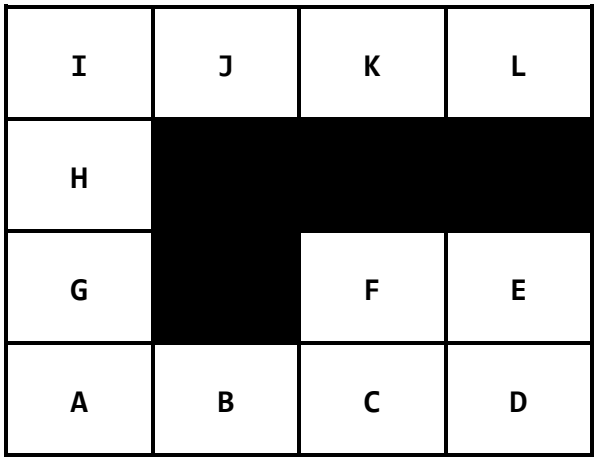
Solution:

This is undirected graph.

```
graph={ "A":["B","C","D"],
        "B":["A"],
        "C":["A","E"],
        "D":["A"],
        "E":["C","F"],
        "F":["E"]
        }
```

Exercise 2:

Convert the following maze into a dictionary in python.



Solution:

First of all, we should imagine this maze as undirected graph, where each two adjacent squares are neighbors in the graph.

```
graph={ "A":["B","G"],
        "B":["A","C"],
        "C":["F","D","B"],
        "D":["C","E"],
        "E":["F","D"],
        "F":["C","E"],
        "G":["A","H"],
        "H":["G","I"],
        "I":["H","J"],
        "J":["I","K"],
        "K":["J","L"],
        "L":["K"]
        }
```

A* Algorithm

A* is an informed search algorithm that aims to find the shortest path between a start node and a goal node. To achieve that, it considers both the cost to reach a node from the start node (g(n)) and the heuristic estimate of the cost from that node to the goal (h(n)).
It selects the path that minimizes the following function:

$$f(n) = g(n) + h(n)$$

where:

- g(n) is the actual cost of the path from the starting point to node n.
- h(n) is the estimated cost of the path from node n to the destination node, as computed by a heuristic.

A* Approach:

- For the start node, compute g, h, and f = g + h.
- Start with a frontier that contains the start node.
- Start with an empty explored set.
- Repeat:
  - If the frontier is empty, then no solution.
  - Remove the node with the lowest f-value from the frontier.
  - If the node is the goal state, return the solution.
  - Add the node to the explored set.
  - Expand the node and compute g, h, and f for each neighbor.
  - For each neighbor node:
    - If it is not in the frontier or explored set, add it to the frontier.
    - If it is already in the frontier and the new g-cost is lower than its current g-cost, update its g-cost and f-value.

To achieve this, the frontier in A* is represented as a priority queue.

A* algorithm in python:

We have discussed Greedy Best-First Search (gbfs) earlier and we had implanted it in python.
To turn Greedy Best-First Search code into A*, we only have to change two main ideas.

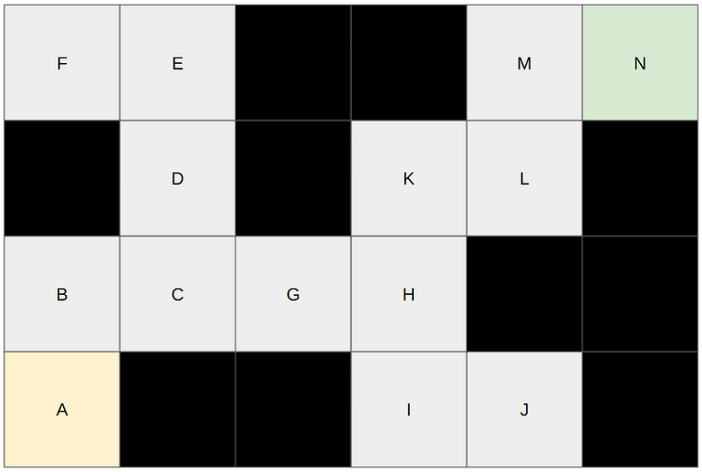
- The first change is how you choose which node to expand next (which available node to take from the priority queue). Greedy uses only the heuristic value $h(n)$ as the priority of the node, so it always picks the node that ***looks*** closest to the goal. A* uses $f(n) = g(n) + h(n)$, which means it cares about both the distance already traveled (g) from the start node to the current node and the estimated distance still left to reach the goal (h). This makes A* search for the truly shortest path, not just the path that looks good at the moment (locally).
- The second change is how nodes in the frontier are handled. In Greedy Best-First Search, once a node is added to the frontier, it is never updated. In contrast, A* updates the priority of a node if a cheaper path (lower total cost) to that node is found, even after it has been pushed into the frontier. This is similar to Dijkstra's algorithm, where a node's cost can be improved if a better path is discovered.

These two changes—using $g(n) + h(n)$ instead of only $h(n)$, and allowing priority update when a cheaper path appears—are what make A* more accurate and optimal than Greedy.

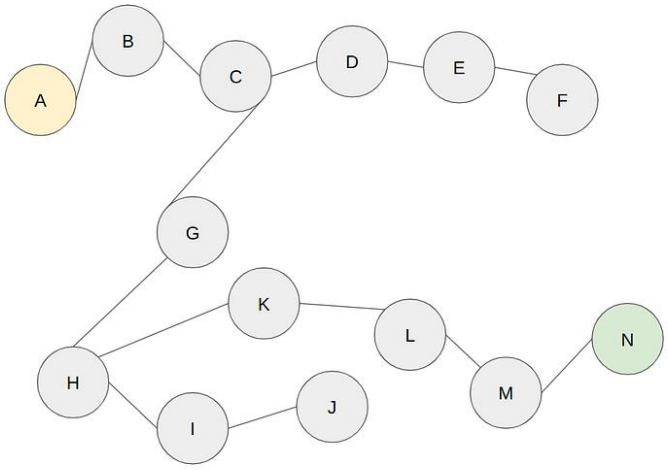
Python code:

```
from queue import PriorityQueue

def a_star_search(graph, start_node, end_node, heuristic):

    solution = []
    frontier = PriorityQueue()
    visited = []

    # cost from start to each node
    g_cost = {start_node: 0}

    # f(n) = g(n) + h(n)
    f_start = g_cost[start_node] + heuristic[start_node]
    frontier.put((f_start, start_node))

    while not frontier.empty():

        _, selected_node = frontier.get()

        if selected_node in visited:
            continue

        visited.append(selected_node)
        solution.append(selected_node)

        if selected_node == end_node:
            break

        for neighbour in graph[selected_node]:
            if neighbour in visited:
                continue

            # cost from start to neighbour through selected_node
            tentative_g = g_cost[selected_node] + 1   # each step costs 1

            # If neighbour has no g_cost yet, give it a very large value
            if neighbour not in g_cost:
                old_cost = 999999
            else:
                old_cost = g_cost[neighbour]

            # Only update if this path is better
            if tentative_g < old_cost:
                g_cost[neighbour] = tentative_g
                f_value = tentative_g + heuristic[neighbour]
                frontier.put((f_value, neighbour))

    return solution
```

Exercise:
For the following maze, apply A* algorithm to figure out whether there is a path from A to N of not.



Solution:
We first **convert the maze into a graph**, where each open cell becomes a node and edges connect neighboring cells that can be moved between.



Then we should convert this graph to a dictionary in order to use it in the code. We also define the heuristic values, and we call the previously defined function as follow:

```
Maze = {
    "A": ["B"],
    "B": ["C"],
    "C": ["D", "G"],
    "D": ["E"],
    "E": ["F"],
    "F": [],
    "G": ["H"],
    "H": ["K", "I"],
    "K": ["L"],
    "L": ["M"],
    "M": ["N"],
    "N": [],
    "I": ["J"],
    "J": [],
}
heuristic = {
    "A": 8,
    "B": 7,
    "C": 6,
    "D": 5,
    "E": 4,
    "F": 5,
    "G": 5,
    "H": 4,
    "K": 3,
    "L": 2,
    "M": 1,
    "N": 0,
    "I": 5,
    "J": 4,
}
start_state = "A"
goal_state = "N"

s=a_star_search(Maze, start_state, goal_state,heuristic)
if (s[-1]=="N"):
```

```
    print("there is a path from A to N")
else:
    print("there is NOT any path from A to N")
```

Output:
there is a path from A to N
Homework:
If there is a path between the start node the goal node, how can we adjust the code to print the path between these two nodes.
Note: "*solution*" in the code is not the path, but it is the searching order through the graph.