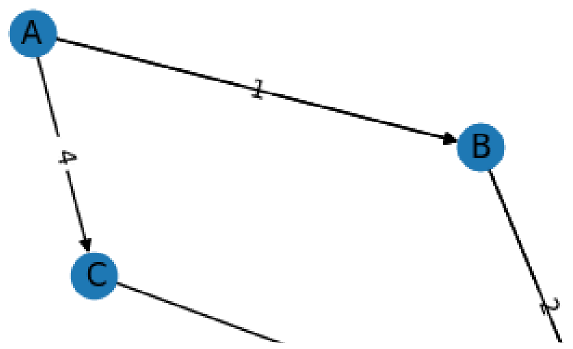
```
import matplotlib.pyplot as plt
import networkx as nx
from collections import deque
import heapq

graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'D': 2},
    'C': {'D': 3},
    'D': {'E': 1},
    'E': {}
}

# DFS function
def dfs(graph, start, goal, path=None, visited=None):
    if path is None: path = [start]
    if visited is None: visited = set()
    visited.add(start)
    if start == goal: return path
    for n in graph[start]:
        if n not in visited:
            r = dfs(graph, n, goal, path + [n], visited)
            if r: return r
    return None

def bfs(graph, start, goal):
    queue = deque([[start]])
    visited = set([start])
    while queue:
        path = queue.popleft()
        node = path[-1]
        if node == goal:
            return path
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(path + [neighbor])
    return None

def ucs(graph, start, goal):
    queue = [(0, [start])]
    visited = set()
    while queue:
        cost, path = heapq.heappop(queue)
        node = path[-1]
        if node == goal:
            return path, cost
        if node not in visited:
            visited.add(node)
            for neighbor, edge_cost in graph[node].items():
                if neighbor not in visited:
                    total_cost = cost + edge_cost
                    heapq.heappush(queue, (total_cost, path + [neighbor]))
    return None, float('inf')

path = dfs(graph, 'A', 'E')
print("DFS Path:", path)
```

```
G = nx.DiGraph(graph);
p = nx.spring_layout(G)
nx.draw(G, p, with_labels=True)
nx.draw_networkx_edge_labels(G, p, edge_labels={(u,v):w for u,d in graph.items
if path:
   nx.draw(G, p, nodelist=path, edgelist=list(zip(path, path[1:])))
plt.show()
```

```
DFS Path: ['A', 'B', 'D', 'E']
```



```
import random

class Environment:
    def __init__(self):
        self.states = ["clean", "dirty"]
        self.current_state = random.choice(self.states)

    def get_state(self):
        return self.current_state

    def perform_action(self, action):
        if action == "clean":
            self.current_state = "clean"
        elif action == "move":
            self.current_state = random.choice(self.states)
        elif action == "do nothing":
            pass
```

```
class SimpleReflexAgent:
    def act(self, percept):
        if percept == "dirty":
            return "clean"
        return "do nothing"


class ModelBasedReflexAgent:
    def __init__(self):
        self.model = "clean"

    def act(self, percept):
        self.model = percept
        if self.model == "dirty":
            return "clean"
        return "move"


class GoalBasedAgent:
    def act(self, percept):
        if percept == "dirty":
            return "clean"
        return "move"


class UtilityBasedAgent:
    def utility(self, state):
        return 1 if state == "clean" else -1

    def act(self, percept):
        if self.utility(percept) < 0:
            return "clean"
        return "relax"


class LearningAgent:
    def __init__(self):
        self.knowledge = {}

    def act(self, percept):
        if percept not in self.knowledge:
            self.knowledge[percept] = "clean"
        return self.knowledge[percept]


def simulate(agent, steps=5):
    env = Environment()
    for i in range(steps):
        s = env.get_state()
        a = agent.act(s)
        env.perform_action(a)
        print(f"Step {i+1}: {s} → {a} → {env.get_state()}")



print("\n--- Simple Reflex Agent ---")
simulate(SimpleReflexAgent())

print("\n--- Model-Based Reflex Agent ---")
```

```
simulate(ModelBasedReflexAgent())

print("\n--- Goal-Based Agent ---")
simulate(GoalBasedAgent())

print("\n--- Utility-Based Agent ---")
simulate(UtilityBasedAgent())

print("\n--- Learning Agent ---")
simulate(LearningAgent())
```

```
--- Simple Reflex Agent ---
Step 1: clean → do nothing → clean
Step 2: clean → do nothing → clean
Step 3: clean → do nothing → clean
Step 4: clean → do nothing → clean
Step 5: clean → do nothing → clean

--- Model-Based Reflex Agent ---
Step 1: dirty → clean → clean
Step 2: clean → move → dirty
Step 3: dirty → clean → clean
Step 4: clean → move → clean
Step 5: clean → move → clean

--- Goal-Based Agent ---
Step 1: dirty → clean → clean
Step 2: clean → move → clean
Step 3: clean → move → clean
Step 4: clean → move → clean
Step 5: clean → move → dirty

--- Utility-Based Agent ---
Step 1: dirty → clean → clean
Step 2: clean → relax → clean
Step 3: clean → relax → clean
Step 4: clean → relax → clean
Step 5: clean → relax → clean

--- Learning Agent ---
Step 1: clean → clean → clean
Step 2: clean → clean → clean
Step 3: clean → clean → clean
Step 4: clean → clean → clean
Step 5: clean → clean → clean
```

```
pit(3,1).
pit(3,3).
pit(4,4).
wumpus(1,3).
gold(2,3).
agent(1,1).

adjacent(X1,Y1,X2,Y2):-
    (   X2 is X1+1 , Y2 is Y1
        ;   X2 is X1-1 , Y2 is Y1
        ;   X2 is X1 , Y2 is Y1+1
        ;   X2 is X1 , Y2 is Y1-1
        ),
    X2>=1,Y2>=1,X2=<4,Y2=<4.

    stench(X,Y):- wumpus(X1,Y1),adjacent(X1,Y1,X,Y).
    breeze(X,Y):- pit(X1,Y1),adjacent(X1,Y1,X,Y).
    glitter(X,Y):-gold(X,Y).
    safe(X,Y):- \+pit(X,Y),\+wuwpus(X,Y).
    move(X, Y, X1, Y1) :- adjacent(X, Y, X1, Y1), safe(X1, Y1).
    grab(X, Y) :- gold(X, Y).
    shoot(X, Y) :- stench(X, Y).




    % ?- move(1, 1, X, Y).
    % ?- stench(X,Y).
    % ?- breeze(X,Y).
    % ?- glitter(X,Y).
```