```
from collections import deque

# Define the graph as an adjacency list
graph = {
    0: [1, 2],
    1: [0, 3, 4],
    2: [0, 5],
    3: [1],
    4: [1],
    5: [2]
}

# Define the BFS function
def bfs(graph, start_vertex):
    # Initialize the visited set to keep track of visited vertices
    visited = set()

    # Initialize the queue with the starting vertex
    queue = deque([start_vertex])

    # Loop until the queue is empty
    while queue:
        # Dequeue the next vertex from the queue
        current_vertex = queue.popleft()

        # If the current vertex has not been visited yet, print it and mark it as visited
        if current_vertex not in visited:
            print(current_vertex)
            visited.add(current_vertex)

            # Enqueue the neighbors of the current vertex that have not been visited yet
            for neighbor in graph[current_vertex]:
                if neighbor not in visited:
                    queue.append(neighbor)

# Call the BFS function with the graph and a starting vertex
bfs(graph, 0)
```

```
0
1
2
3
4
5
```

```
def dfs(graph, start):
    visited = set()
    stack = [start]

    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            print(vertex)
            stack.extend(neighbor for neighbor in graph[vertex] if neighbor not in visited)

# Example usage:
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

dfs(graph, 'A')
```

```
A
C
F
E
B
D
```

```
import numpy as np

class AStar:
    def __init__(self, start, goal):
        self.start = start
        self.goal = goal
```

```
    def heuristic(self, node):
        return np.sum(np.abs(node - self.goal))

    def neighbors(self, node):
        neighbors = []
        for i in range(-1, 2):
            for j in range(-1, 2):
                if i == 0 and j == 0:
                    continue
                new_node = node.copy()
                new_node[1 + i, 1 + j] = new_node[1, 1]
                new_node[1, 1] = new_node[1 + i, 1 + j]
                neighbors.append(new_node)
        return neighbors

    def search(self):
        open_list = [(self.heuristic(self.start), 0, self.start)]
        closed_list = set()
        came_from = {}
        g_score = {self.start: 0}
        f_score = {self.start: self.heuristic(self.start)}

        while open_list:
            _, current_g, current = min(open_list)
            open_list.remove((current_g, current_g, current))
            closed_list.add(tuple(current.ravel()))

            if np.array_equal(current, self.goal):
                path = [current]
                while current in came_from:
                    current = came_from[current]
                    path.append(current)
                return path[::-1]

            for neighbor in self.neighbors(current):
                tentative_g_score = g_score[current] + 1
                if tuple(neighbor.ravel()) not in g_score or tentative_g_score < g_score[neighbor]:
                    came_from[neighbor] = current
                    g_score[neighbor] = tentative_g_score
                    f_score[neighbor] = tentative_g_score + self.heuristic(neighbor)
                    if neighbor not in open_list:
                        open_list.append((f_score[neighbor], tentative_g_score, neighbor))

        return []
```