# 实验报告：图搜索BFS算法及存储优化

**胡延伸 PB22050983**

---

## 一、实验内容

本实验要求实现图的广度优先搜索（BFS）算法，并选择合适的存储方式（邻接矩阵或邻接表）存储图数据。实验包括以下部分：

1. 针对 `data.txt` 数据构建无向图，选择合适的方式存储，并以节点A为起始点输出BFS遍历过程。
2. 针对 `twitter_small` 数据集（有向图），选择合适的存储方式存储数据，并输出BFS的遍历时间。

---

## 二、实验目的

1. 掌握图的两种存储方式：邻接矩阵和邻接表。
2. 掌握广度优先搜索（BFS）算法并理解其执行过程。
3. 根据图的特征（节点数和边密度）选择合适的存储方式，优化算法性能。
4. 分析BFS在不同规模图上的性能表现。

---

## 三、算法设计思路

### 1. 存储方式选择

- **邻接矩阵**：适合节点数少但边密集的图，时间复杂度为 `O(V^2)`。
- **邻接表**：适合节点数多但边稀疏的图，时间复杂度为 `O(V + E)`。

针对实验数据：

1. `data.txt`：节点数少，边密集（无向图），适合使用邻接矩阵存储。
2. `twitter_small`：节点数多，边稀疏（有向图），适合使用邻接表存储。

### 2. 广度优先搜索（BFS）算法

- 使用一个队列存储当前层的节点。
- 依次访问队列中的节点，并将其相邻的未访问节点加入队列。
- 遍历过程中记录访问顺序或统计执行时间。

---

## 四、源码 + 注释

### 1. `data.txt` 示例代码（邻接矩阵 + BFS）

```
class ADJMatrix:
    def __init__(self, n):
        # 初始化邻接矩阵
        self.graph = [[0 for _ in range(n)] for _ in range(n)]
        self.vertices_to_integer = {}
        self.integer_to_vertices = {}

    def addEdge(self, u, v):
        # 添加无向边
        self.graph[u][v] = 1
        self.graph[v][u] = 1

    def BFS(self, s):
        # BFS算法
        visited = [False] * len(self.graph)
        queue = [s]
        visited[s] = True

        while queue:
            s = queue.pop(0)
            print(self.integer_to_vertices[s], end=" ")
            for i in range(len(self.graph[s])):
                if self.graph[s][i] == 1 and not visited[i]:
                    queue.append(i)
                    visited[i] = True


# 读取data.txt并构建邻接矩阵
```

```
with open('data.txt', 'r') as f:
    vertices = f.readline().strip().split(',')
    vertices_to_integer = {v: i for i, v in enumerate(vertices)}
    integer_to_vertices = {i: v for v, i in vertices_to_integer.items()}
    edges = f.readlines()

# 初始化邻接矩阵
graph_ADJMatrix = ADJMatrix(len(vertices))
graph_ADJMatrix.vertices_to_integer = vertices_to_integer
graph_ADJMatrix.integer_to_vertices = integer_to_vertices
for edge in edges:
    u, v = edge.strip().split('-')
    graph_ADJMatrix.addEdge(vertices_to_integer[u], vertices_to_integer[v])

# 从A节点开始BFS
print("BFS traversal starting from A:")
graph_ADJMatrix.BFS(vertices_to_integer['A'])
```

## 2. `twitter_small` 示例代码（邻接表 + BFS）

```
import time

class ADJList:
    def __init__(self):
        # 初始化邻接表
        self.graph = {}

    def addEdge(self, u, v):
        # 添加有向边
        if u not in self.graph:
            self.graph[u] = []
        self.graph[u].append(v)

    def BFS(self, s, print_vertices=True):
        # BFS算法
        visited = {vertex: False for vertex in self.graph.keys()}
        queue = [s]
        visited[s] = True

        while queue:
            s = queue.pop(0)
```

```
            if print_vertices:
                print(s, end=" ")
            for i in self.graph.get(s, []):
                if not visited[i]:
                    queue.append(i)
                    visited[i] = True


# 读取twitter_small数据集并构建邻接表
with open('twitter_small.txt', 'r') as f:
    edges = f.readlines()
    graph_twitter_small = ADJList()
    for edge in edges:
        u, v = map(int, edge.strip().split())
        graph_twitter_small.addEdge(u, v)

# 测试BFS性能
start_node = list(graph_twitter_small.graph.keys())[0]
time_start = time.time()
graph_twitter_small.BFS(start_node, print_vertices=False)
time_end = time.time()

print("\nTime used for BFS on twitter_small: ", time_end - time_start)
```

---

## 五、算法测试结果

### 1. `data.txt` 测试结果

**输入数据：**

```
A,B,C,D,E,F,G,H,I
A-B
A-C
B-D
B-E
B-G
C-H
C-G
C-F
```

```
D-E
D-G
E-H
D-H
D-I
E-I
F-H
F-I
H-G
H-I
```

**运行结果：**

```
A B C D E G F H I
```

**分析：** BFS按照层次顺序访问节点，结果符合预期。

---

### 2. `twitter_small` 测试结果

**输入数据：**

- 节点数：81306
- 边数：1768149

**运行结果：**

```
Time used for BFS on twitter_small:  0.4179840087890625
```

**分析：** 得益于邻接表的存储方式，BFS在稀疏图上的执行时间较短。

---

## 六、实验过程中遇到的困难及收获

### 困难

1. **数据规模较大**：`twitter_small`数据集规模较大，如何选择合适的数据结构存储成为关键。
2. **边的方向**：在处理有向图时，需要特别注意边的方向，仅添加单向边。

### 收获

1. 深刻理解了邻接矩阵和邻接表的优劣及适用场景。
2. 通过BFS的实现，掌握了图搜索算法的核心思想。
3. 学会了如何通过实验数据特征优化存储方式，提升算法性能。

---

## 七、总结

本实验通过对`data.txt`和`twitter_small`两个数据集的测试，验证了存储方式对图搜索性能的影响。实验结果表明：

- **邻接矩阵**适合小规模、边密集的图，而
- **邻接表**适合大规模、边稀疏的图。

广度优先搜索是一种高效的图遍历算法，结合合适的存储方式可以显著提升算法效率。