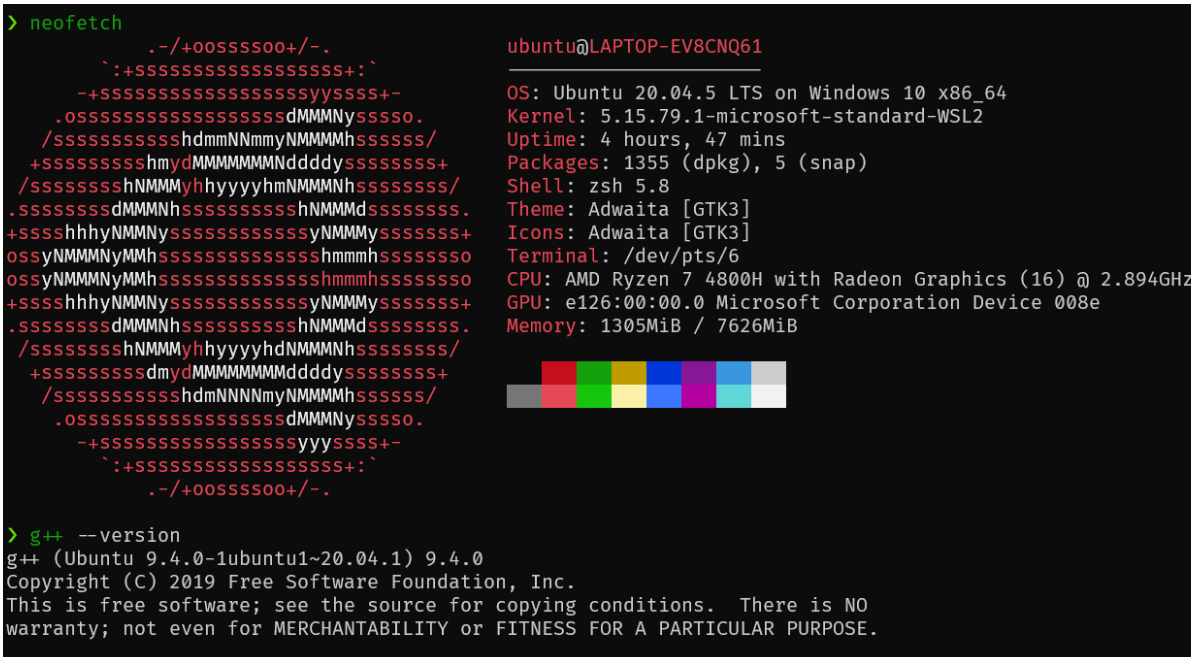
算法实验 4 实验报告

PB20000180 刘良宇

实验设备和环境



编译 `gen_input` 并运行：

```
g++ ./src/gen_input.cpp -o gen_input
./gen_input
```

编译 `main` 并运行：

```
g++ -O2 ./src/main.cpp -o main
./mian
```

实验内容和要求

实现 Johnson 算法，对一个有向图求出所有点对的最短路径及其长度。

方法和步骤

数据生成

首先按照要求随机生成指定数量的边，随后不断使用 Bellman-Ford 算法判断图中是否存在负环。如果不存在，即可保存数据。如果存在，则找出一个可松弛边，不断寻找它的前驱，直到发现环。这个时候就找到了负环，从上面删除一条边，重复 Bellman-Ford 算法。

具体实现上，设计一个检测并消除负环的 detectNegativeCycle 函数，如果存在负环，删除一条边。该函数的返回值是本次运行 Bellman-Ford 算法是否检测到负环。

```cpp
/**
 * @brief 检测负环，如果有则删除
 *
 * @param nodes 图
 * @return true 有负环
 * @return false 没有负环
 */
bool detectNegativeCycle(std::vector<std::unordered_map<int, int>> &nodes) {
    // 首先运行一遍 Bellman-Ford 算法，以第一个点为源点
    auto n = nodes.size() - 1;
    // d 需要初始化为 inf 数组，除了源点为 0
    auto d = std::vector<int>(n + 1, std::numeric_limits<int>::max() / 2);
    d[1] = 0;
    // 前驱数组
    auto pi = std::vector<int>(n + 1, 0);

    // 循环 |V| - 1 次，松弛每一条边
    for (auto i = 1; i <= n - 1; i++) {
        for (auto u = 1; u <= n; u++) {
            for (auto [v, weight] : nodes[u]) {
                if (d[v] > d[u] + weight) {
                    d[v] = d[u] + weight;
                    pi[v] = u;
                }
            }
        }
    }

    // 判断是否有负环
    auto found = false;
    auto path = std::unordered_set<int>{};
    auto runner = 0;
    for (auto u = 1; u <= n; u++) {
        for (auto [v, weight] : nodes[u]) {
            if (d[v] > d[u] + weight) {
                found = true;
                path.insert(v);
                runner = v;
            }
        }
    }

    // 如果没找到，不需要进一步处理了
    if (!found)
        return false;

    // 否则不断寻找前驱，直到重复
    while ((runner = pi[runner]) != 0) {
        if (path.count(runner) == 0) {
            path.insert(runner);
        } else {
            auto pre = pi[runner];
            // 删去边 <pre, runner>
```

```
            nodes[pre].erase(runner);
            break;
        }
    }

    return true;
}
```

这样，在 main 函数内部就可以简单的调用它，完成负环的消除：

```
// delete negative weight cycle
while (detectNegativeCycle(nodes))
    ;
```

堆优 Dijkstra 算法

使用堆优化，使得复杂度来到 $O(e \log e)$。

```
struct dist {
    int dis, u;
    bool operator>(const dist &a) const { return dis > a.dis; }
};

std::vector<int> dijkstra(std::vector<std::unordered_map<int, int>> &nodes,
                          int src) {
    auto n = nodes.size() - 1;
    auto d = std::vector<int>(n + 1, std::numeric_limits<int>::max() / 2);
    d[src] = 0;

    // 前驱数组，即为返回的对象
    auto pi = std::vector<int>(n + 1, 0);

    // 优先队列
    auto p_q = std::priority_queue<dist, std::vector<dist>, std::greater<>>{};
    p_q.push({0, src});

    // 记录已确定的集合
    auto visited = std::unordered_set<int>{};

    while (!p_q.empty()) {
        int u = p_q.top().u;
        p_q.pop();
        if (visited.count(u) != 0)
            continue;
        visited.insert(u);
        auto &adjs = nodes[u];
        for (auto [v, weight] : adjs) {
            if (d[v] > d[u] + weight) {
                d[v] = d[u] + weight;
                pi[v] = u;
                p_q.push({d[v], v});
            }
        }
    }
    return pi;
}
```

Johnson 算法

首先虚设结点，运行 Bellman-Ford 算法。随后根据结果更新边权，消除负权重的边。接下来以每个点为源点运行 Dijkstra 算法即可。

```
/**
 * @brief 运行 johnson 算法，返回构造完毕的矩阵
 *
 * @param nodes 需要执行算法的图
 */
std::vector<std::vector<int>>
johnson(std::vector<std::unordered_map<int, int>> nodes) {
    // 首先运行一遍 Bellman-Ford 算法，以第 0 个点为源点（虚拟结点）
    auto n = nodes.size();
    // d 需要初始化为 inf 数组，除了源点为 0
    auto d = std::vector<int>(n, std::numeric_limits<int>::max() / 2);
    d[0] = 0;

    // 循环 |V| - 1 次，松弛每一条边
    for (auto i = 1; i <= n - 1; i++) {
        for (auto u = 0; u < n; u++) {
            for (auto [v, weight] : nodes[u]) {
                if (d[v] > d[u] + weight) {
                    d[v] = d[u] + weight;
                }
            }
        }
    }

    n--;
    // 边权重新设定为 w + h_u - h_v
    nodes[0].clear();
    for (auto u = 1; u <= n; u++) {
        auto &map = nodes[u];
        for (auto &[v, weight] : map) {
            weight += d[u] - d[v];
        }
    }

    // 以每个点为起点，运行 n 次 dijkstra 算法
    // dijkstra 返回的是 src 到 dst 最短路径上 dst 前面的点
    std::vector<std::vector<int>> result{{}};
    for (auto u = 1; u <= n; u++) {
        result.push_back(dijkstra(nodes, u));
    }

    return result;
}
```

时间统计及输出

输出时因为需要输出具体路径，所以要根据前驱数组还原。

```
// 执行 johnson 算法
gettimeofday(&t1, nullptr);
auto johnson_result = johnson(nodes);
```

```cpp
gettimeofday(&t2, nullptr);

// johnson 算法的结果保存到输出文件
auto output_fstream = std::ofstream(output_filename);
for (auto u = 1; u <= n; u++) {
    for (auto v = 1; v <= n; v++) {
        if (u == v)
            continue;
        auto pre_v = johnson_result[u][v];
        if (pre_v == 0) {
            // 不连通
            output_fstream << "Null!" << std::endl;
            continue;
        }
        // 循环找 v 的前驱，直到找到 u
        auto path = std::vector<int>{v};
        auto length = 0;
        while (path.back() != u) {
            length += nodes[pre_v][path.back()];
            path.push_back(pre_v);
            pre_v = johnson_result[u][pre_v];
        }
        // 此时 path 逆向保存了 u 到 v 的路径，输出即可
        output_fstream << '(';
        while (path.size() > 1) {
            auto back = path.back();
            output_fstream << back << ',';
            path.pop_back();
        }
        output_fstream << v << ' ' << length << ')' << std::endl;
    }
}
```

代码实际统计的仅为执行 Johnson 算法的时间。

结果与分析

以 `input12.txt` 为例：

```
1 7 40
2 22 46
3 20 -3
4 13 10
5 4 49
6 26 18
7 6 21
8 5 4
9 23 -2
10 18 13
11 15 -7
12 7 8
13 10 5
14 4 12
15 16 18
```

```
16 7 9
17 14 40
18 25 39
19 24 -10
20 6 41
21 2 19
22 1 33
23 15 10
24 11 19
25 1 48
26 4 45
27 10 33
```

观察输出中第一个结点到其他结点的最短路径：

```
Null!
Null!
(1,7,6,26,4 124)
Null!
(1,7,6 61)
(1,7 40)
Null!
Null!
(1,7,6,26,4,13,10 139)
Null!
Null!
(1,7,6,26,4,13 134)
Null!
Null!
Null!
Null!
(1,7,6,26,4,13,10,18 152)
Null!
Null!
Null!
Null!
Null!
Null!
(1,7,6,26,4,13,10,18,25 191)
(1,7,6,26 79)
Null!
```

可以发现是正确的。

复杂度分析

输出 `time.txt` 如下（`#` 后是注释内容）：

```
0.051ms   # n = 27,  e = 54,   f = nelne/20000 = 0.02907969369945356
0.016ms   # n = 27,  e = 27,   f = nelne/20000 = 0.01201332537658578
0.44ms    # n = 81,  e = 160,  f = nelne/20000 = 0.32887126322715193
0.432ms   # n = 81,  e = 162,  f = nelne/20000 = 0.3337971955545967
6.936ms   # n = 243, e = 709,  f = nelne/20000 = 5.654334885498202
4.71ms    # n = 243, e = 486,  f = nelne/20000 = 3.6528943303270025
84.101ms  # n = 729, e = 2811, f = nelne/20000 = 81.36726884258756
35.571ms  # n = 729, e = 1770, f = nelne/20000 = 48.25017953737498
```

理论复杂度：$O(ne\log e)$

我们尝试用理论时间复杂度线性拟合，如上所示，$f = ne\ln e/20000$ 就是一个简单的拟合。

已经可以看出来十分接近，说明实际运行时间符合理论时间复杂度。