Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский университет ИТМО»

Факультет программной инженерии и компьютерной техники

Отчет по блоку №4 (timus)

По Алгоритмам и структурам данных

Выполнил:

Ступин Тимур Русланович

Группа № P3108

Поток № 1.3

Преподаватель:

Тараканов Денис Сергеевич

Санкт-Петербург 2025

Содержание

Задача 1

Решение

Заметим, что если известна максимальная допустимая длинна кабеля, то проверить является ли она подходящей можно выполнив простой обход по графу (например в глубину), игнорируя рёбра, вес которых больше выбранного максимума. Если в ходе обхода все вершины будут посещены то данное ограничение на длину является допустимым, иначе нет. Также отметим, что если некоторое ограничение на длину кабеля является подходящим, то любое большее ограничение является подходящим. Напротив, если некоторое ограничение на длину не подходит, то и любое меньшее ограничение также не будет подходить. Из этого следует, что выбор оптимального значения ограничения на длину можно выполнить с помощью бинарного поиска.

Для восстановления ответа достаточно выполнить ещё один обход графа, с найденным ограничением, запомнив все пройденные рёбра. После чего вывести полученный список рёбер.

Сложность алгоритма

- По времени: Сложность бинарного поиска $O(\log A)$, где A - максимальное значение длинны кабеля. При этом на каждом шаге бинарного поиска происходит обход графа, который имеет сложность $O(N + M)$. Таким образом итоговая сложность будет $O((N + M) \log A)$
- По памяти: В памяти хранится граф в виде списков смежности. Таким образом сложность будет $O(N + M)$

Итого:

- По времени: $O((N + M) \log A)$
- По памяти: $O(N + M)$

Код

```cpp
#include <bits/stdc++.h>
using namespace std;

struct weight_edge {
  int u;
  int w;
};

struct edge {
  int u;
  int v;
};

void dfs(const int v, const int c, vector<bool>& used,
 ↪  vector<vector<weight_edge>>& g) {
  used[v] = true;
  for (auto [u, w] : g[v]) {
    if (w <= c && !used[u]) {
      dfs(u, c, used, g);
    }
  }
}

bool is_all_visited(vector<bool>& used) {
  for (const auto a : used) {
    if (!a) {
      return false;
    }
  }
  return true;
}

bool check(const int c, vector<vector<weight_edge>>& g) {
  const int n = static_cast<int>(g.size());
  vector used(n, false);
  dfs(0, c, used, g);
  return is_all_visited(used);
}

void create_ans(
    const int v, const int c, vector<edge>& ans, vector<bool>& used,
     ↪  vector<vector<weight_edge>>& g
) {
  used[v] = true;
  for (auto [u, w] : g[v]) {
    if (w <= c && !used[u]) {
      ans.push_back({u, v});
      create_ans(u, c, ans, used, g);
    }
```

```
  }
}

int main() {
  int n, m;
  cin >> n >> m;

  vector<vector<weight_edge>> g(n);

  for (int i = 0; i < m; i++) {
    int u, v, w;
    cin >> u >> v >> w;
    u--;
    v--;
    g[u].push_back({v, w});
    g[v].push_back({u, w});
  }

  int l = 0, r = 1e9;
  while (r - l > 1) {
    const int c = (l + r) / 2;
    if (check(c, g)) {
      r = c;
    } else {
      l = c;
    }
  }
  cout << r << endl;

  vector<edge> ans;
  vector used(n, false);
  create_ans(0, r, ans, used, g);

  cout << ans.size() << endl;
  for (const auto [u, v] : ans) {
    cout << u + 1 << " " << v + 1 << endl;
  }
}
```

Задача 2

Решение

Заметим, что сама по себе задача является обычной задачей поиска минимального пути в графе, которая может быть решена обычным алгоритмом Дейкстры. Однако сложность заключается в том, что граф не задан явно и для определения вершин смежных с данной нужно выполнить некоторые вычисления. Отметим, что с учётом доступных преобразований, из некоторого номера `a` можно получить $10 \cdot 10 = 100$ номеров путём замены одной цифры и $C_{10}^2 = 45$ номеров путём обмена двух цифр местами. Таким образом, имея некоторый номер, можно достаточно быстро получить список всех потенциальных соседей, из которых достаточно выбрать те номера, которые реально есть в графе. Для этого можно сохранить все номера в хэш таблицу, что позволит проверять наличие некоторого номера за константу.

Само решение является реализацией алгоритма Дейкстры с оптимизацией за счёт использования `set`. Единственной особенностью является то, что для каждой выбранной вершины, относительно которой будет происходить релаксации, происходит генерация списка смежных вершин на основе её номера.

Восстановление ответа происходит стандартно, через массив переходов.

Сложность алгоритма

- По времени: Сложность алгоритма Дейкстры с оптимизацией поиска минимума составляет $O(M \log N)$ при этом в данном случае число рёбер можно оценить как N, что даёт сложность $O(N \log N)$. Время, затрачиваемое на генерацию рёбер не учитывается, так как считается константным.
- По памяти: В памяти хранятся вершины и вспомогательные массивы, каждый из которых не превосходит по размеру количество вершин, что даёт сложность $O(N)$.

Итого:

- По времени: $O(N \log N)$
- По памяти: $O(N)$

Код

```cpp
#include <bits/stdc++.h>
using namespace std;

constexpr int INF = 1e9;

struct edge {
  int u;
  int w;
};

void generate_swap_edges(
    const int v,
    vector<edge>& edges,
    const unordered_map<string, int>& mode_map,
    const vector<int>& weights,
    const vector<string>& nodes
) {
  for (int i = 0; i < 10; i++) {
    for (int j = i + 1; j < 10; j++) {
      if (nodes[v][i] == nodes[v][j]) {
        continue;
      }

      string uuid = nodes[v];
      swap(uuid[i], uuid[j]);

      auto it = mode_map.find(uuid);
      if (it == mode_map.end()) {
        continue;
      }

      const int u = it->second;
      const int w = weights[i];

      edges.push_back({u, w});
    }
  }
}

void generate_replace_edges(
    const int v,
    vector<edge>& edges,
    const unordered_map<string, int>& node_map,
    const vector<int>& weights,
    const vector<string>& nodes
) {
  for (int i = 0; i < 10; i++) {
    for (char c = '0'; c <= '9'; c++) {
      if (nodes[v][i] == c) {
```

```
        continue;
      }

      string uuid = nodes[v];
      uuid[i] = c;

      auto it = node_map.find(uuid);
      if (it == node_map.end()) {
        continue;
      }

      const int u = it->second;
      const int w = weights[i];

      edges.push_back({u, w});
    }
  }
}

int main() {
  int n;
  cin >> n;

  vector<int> weights(10);
  for (int i = 0; i < 10; i++) {
    cin >> weights[i];
  }

  unordered_map<string, int> node_map;
  vector<string> nodes(n);
  for (int i = 0; i < n; i++) {
    string uuid;
    cin >> uuid;
    node_map[uuid] = i;
    nodes[i] = uuid;
  }

  vector<vector<edge>> g(n);
  for (int i = 0; i < n; i++) {
    generate_swap_edges(i, g[i], node_map, weights, nodes);
    generate_replace_edges(i, g[i], node_map, weights, nodes);
  }

  vector d(n, INF);
  set<pair<int, int>> d_s;
  d[0] = 0;
  d_s.insert(make_pair(0, 0));

  vector<int> h(n);
  h[0] = -1;
```

```
100
101   for (int k = 0; k < n; k++) {
102     if (d_s.empty()) {
103       break;
104     }
105     const int cur_node = d_s.begin()->second;
106     d_s.erase(d_s.begin());
107     for (auto [u, w] : g[cur_node]) {
108       if (d[cur_node] + w < d[u]) {
109         d_s.erase(make_pair(d[u], u));
110         d[u] = d[cur_node] + w;
111         d_s.insert(make_pair(d[u], u));
112         h[u] = cur_node;
113       }
114     }
115   }
116   if (d[n - 1] == INF) {
117     cout << -1 << endl;
118     return 0;
119   }
120   cout << d[n - 1] << endl;
121   vector<int> ans;
122   int ptr = n - 1;
123   while (ptr != -1) {
124     ans.push_back(ptr + 1);
125     ptr = h[ptr];
126   }
127   reverse(ans.begin(), ans.end());
128   cout << ans.size() << endl;
129   for (const int a : ans) {
130     cout << a << " ";
131   }
132   cout << endl;
133 }
```