Министерство образования и науки Российской Федерации федеральное государственное
автономное образовательное учреждение высшего образования
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Факультет «Программной инженерии и компьютерной техники.»

Алгоритмы и структуры данных

Лабораторная работа №4
Timus

Выполнил
Григорьев Давид Владимирович
Группа: P3215
Преподаватели
Косяков Михаил Сергеевич
Тараканов Денис Сергеевич

Санкт-Петербург 2025

Содержание

1 1329. Галактическая история

Пояснение к примененному алгоритму

Для эффективного определения иерархических отношений между узлами ориентированного ациклического дерева используется обход в глубину (DFS) с фиксацией времен входа (`in_time`) и выхода (`out_time`) для каждого узла. Эти метки времени позволяют быстро определять, является ли один узел предком другого. При обработке запросов:

- Если $\texttt{in\_time}[a] \leq \texttt{in\_time}[b]$ и $\texttt{out\_time}[a] \geq \texttt{out\_time}[b]$, то узел a является предком узла b.
- Если $\texttt{in\_time}[b] \leq \texttt{in\_time}[a]$ и $\texttt{out\_time}[b] \geq \texttt{out\_time}[a]$, то узел b является предком узла a.
- В остальных случаях узлы a и b не связаны иерархически.

Сложность по времени:

- $O(N)$ — обход в глубину по всем узлам дерева, где каждый узел посещается ровно один раз.
  - Обход DFS: $O(N)$
  - Обработка L запросов: $O(L)$

Сложность по памяти:

- $O(N)$ — хранение структуры дерева в виде списка дочерних узлов (`children`).
- $O(N)$ — хранение массивов `in_time` и `out_time`.
- Итого: $O(N)$ (доминирует размер структуры дерева).

Код алгоритма

```cpp
#include <iostream>
#include <stack>
#include <unordered_map>
#include <vector>

int main() {
  int N;
  std::cin >> N;

  std::unordered_map<int, std::vector<int>> children;
  int root = -1;

  for (int i = 0; i < N; ++i) {
    int id = 0;
    int parent_id = 0;
    std::cin >> id >> parent_id;

    if (parent_id == -1) {
      root = id;
    } else {
      children[parent_id].push_back(id);
    }
```

```
}

// Initialize in_time and out_time arrays
const int max_nodes = 40001;
std::vector<int> in_time(max_nodes, 0);
std::vector<int> out_time(max_nodes, 0);
int time = 0;

// DFS traversal using a stack
std::stack<std::pair<int, bool>> dfs_stack;
dfs_stack.emplace(root, false);

while (!dfs_stack.empty()) {
  auto current = dfs_stack.top();
  dfs_stack.pop();

  int node = current.first;
  bool visited = current.second;

  if (!visited) {
    in_time[node] = time++;
    dfs_stack.emplace(node, true);
    // Push children in reverse order to maintain correct DFS order
    for (auto it = children[node].rbegin(); it != children[node].rend(); ++it) {
      dfs_stack.emplace(*it, false);
    }
  } else {
    out_time[node] = time++;
  }
}

int L;
std::cin >> L;

std::vector<std::string> results;
for (int i = 0; i < L; ++i) {
  int a = 0;
  int b = 0;
  std::cin >> a >> b;

  int a_in = in_time[a];
  int a_out = out_time[a];
  int b_in = in_time[b];
  int b_out = out_time[b];

  if (a_in <= b_in && a_out >= b_out) {
    results.emplace_back("1");
  } else if (b_in <= a_in && b_out >= a_out) {
    results.emplace_back("2");
  } else {
    results.emplace_back("0");
  }
}

// Output all results
```

```
  for (const std::string& result : results) {
    std::cout << result << '\n';
  }

  return 0;
}
```

2 1450. Российские газопроводы

Пояснение к примененному алгоритму

Для эффективного поиска маршрута с максимальной суммарной прибыльностью в DAG (ориентированном ациклическом графе) используется топологическая сортировка. Алгоритм состоит из следующих шагов:

- **Топологическая сортировка графа** с помощью алгоритма Кахана:
  - Подсчитывается количество входящих рёбер (`in_degree_count`) для каждой вершины.
  - Вершины с нулевой степенью входящих рёбер добавляются в очередь и обрабатываются последовательно.
  - После обработки вершины её соседи получают уменьшение счётчика входящих рёбер, и те, у кого счётчик достигает нуля, добавляются в очередь.
- **Инициализация массива расстояний** (`longest_path_distances`):
  - Все элементы заполняются значением минус бесконечности (`unreachable`).
  - Расстояние до стартовой вершины S устанавливается равным 0.
- **Обработка вершин в топологическом порядке**:
  - Для текущей вершины обновляются расстояния до всех её соседей через операцию max.
  - Если путь до соседа через текущую вершину дает большее значение, чем текущее, расстояние обновляется.
- **Проверка достижимости** конечной вершины F:
  - Если расстояние до F осталось равным `unreachable`, выводится «No solution».
  - Иначе возвращается значение `longest_path_distances[F]`.

Сложность по времени:

- $O(V + E)$ — время работы топологической сортировки и обработки всех рёбер.
  - Топологическая сортировка: $O(V + E)$ (каждая вершина и ребро обрабатываются один раз).
  - Обработка вершин в топологическом порядке: $O(V + E)$ (каждое ребро проверяется один раз).

Сложность по памяти:

- $O(V + E)$ — хранение графа в виде списка смежности (`adjacency_list`).
- $O(V)$ — массивы `in_degree_count`, `topological_ordering` и `longest_path_distances`.
- Итого: $O(V + E)$ (доминирует размер графа).

Код алгоритма

```cpp
#include <algorithm>
#include <iostream>
#include <limits>
#include <queue>
#include <utility>
#include <vector>

using VertexIndex = int;
using EdgeWeight = int;

namespace {

// Computes a topological ordering of the graph using Kahn's algorithm
std::vector<VertexIndex> ComputeTopologicalOrdering(
    VertexIndex total_nodes,
    const std::vector<std::vector<std::pair<VertexIndex, EdgeWeight>>>& adjacency_list,
    const std::vector<int>& in_degree_count
) {
  std::vector<int> in_degree_copy = in_degree_count;

  std::queue<VertexIndex> queue;
  std::vector<VertexIndex> topological_ordering;

  for (VertexIndex node = 0; node < total_nodes; ++node) {
    if (in_degree_copy[node] == 0) {
      queue.push(node);
    }
  }

  while (!queue.empty()) {
    VertexIndex current_node = queue.front();
    queue.pop();
    topological_ordering.push_back(current_node);

    for (const auto& edge : adjacency_list[current_node]) {
      VertexIndex neighbor_node = edge.first;
      in_degree_copy[neighbor_node]--;
      if (in_degree_copy[neighbor_node] == 0) {
        queue.push(neighbor_node);
      }
    }
  }

  return topological_ordering;
}

}  // namespace

int main() {
  VertexIndex total_nodes = 0;
  VertexIndex total_edges = 0;
  std::cin >> total_nodes >> total_edges;
```

```cpp
std::vector<std::vector<std::pair<VertexIndex, EdgeWeight>>> adjacency_list(total_node
std::vector<int> in_degree_count(total_nodes, 0);

// Read and add edges to the graph
for (VertexIndex i = 0; i < total_edges; ++i) {
  VertexIndex start_node = 0;
  VertexIndex end_node = 0;
  EdgeWeight edge_weight = 0;
  std::cin >> start_node >> end_node >> edge_weight;

  // Convert to 0-based indexing
  start_node -= 1;
  end_node -= 1;

  adjacency_list[start_node].emplace_back(end_node, edge_weight);
  in_degree_count[end_node]++;
}

VertexIndex source_node = 0;
VertexIndex destination_node = 0;
std::cin >> source_node >> destination_node;

// Convert to 0-based indexing
source_node -= 1;
destination_node -= 1;

std::vector<VertexIndex> topological_ordering =
    ComputeTopologicalOrdering(total_nodes, adjacency_list, in_degree_count);

// Initialize the longest path distances with negative infinity
const EdgeWeight unreachable = std::numeric_limits<EdgeWeight>::min();
std::vector<EdgeWeight> longest_path_distances(total_nodes, unreachable);
longest_path_distances[source_node] = 0;

for (VertexIndex current_node : topological_ordering) {
  if (longest_path_distances[current_node] == unreachable) {
    continue;
  }

  for (const auto& edge : adjacency_list[current_node]) {
    VertexIndex neighbor_node = edge.first;
    EdgeWeight edge_weight = edge.second;

    longest_path_distances[neighbor_node] = std::max(
        longest_path_distances[neighbor_node], longest_path_distances[current_node] +
    );
  }
}

if (longest_path_distances[destination_node] == unreachable) {
  std::cout << "No solution" << '\n';
} else {
  std::cout << longest_path_distances[destination_node] << '\n';
}
```

```
  return 0;
}
```