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
Базовые задачи

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

Санкт-Петербург 2025

Содержание

1 Задача M. Цивилизация

Основная идея

Для поиска кратчайшего пути на карте с различными типами рельефа используется алгоритм Дейкстры. Алгоритм поддерживает приоритетную очередь для выбора следующей вершины с минимальной стоимостью достижения.

- Инициализация: устанавливаем расстояние до начальной вершины равным 0, а до всех остальных — бесконечности.
- В приоритетную очередь добавляется начальная вершина с нулевой стоимостью.
- Пока очередь не пуста:
  - Извлекается вершина с минимальной стоимостью достижения.
  - Для каждой смежной вершины рассчитывается новая стоимость достижения.
  - Если новая стоимость меньше текущей, обновляются расстояние и предшественник, а вершина добавляется в очередь.
- Восстановление пути: начинаем с конечной вершины и двигаемся по предшественникам до начальной точки.

Сложность по времени:

- $O(N\,M\,\log(N\,M))$ — где N и M — размеры карты, $\log(N\,M)$ от количества операций с приоритетной очередью.
  - Обработка каждой вершины: $O(\log(N\,M))$
  - Обработка каждого ребра: $O(\log(N\,M))$

Сложность по памяти:

- $O(N\,M)$ — хранение карты, расстояний и предшественников.
  - Хранение карты: $O(N\,M)$
  - Хранение расстояний: $O(N\,M)$
  - Хранение предшественников: $O(N\,M)$

Код алгоритма

```
/*

Мы должны использовать алгоритм Дейкстры для поиска кратчайшего пути

легче всего получать инцидентные вершины прямо с матрицы, не создавая
вектор инцидентных вершин.


*/

#include <algorithm>
#include <cassert>
#include <cstddef>
#include <cstdint>
```

```cpp
#include <iostream>
#include <queue>
#include <string>
#include <utility>
#include <vector>

using Coordinates = std::pair<size_t, size_t>;
enum VertexType : int8_t { Uninitialized = 0, Field = 1, Forest = 2, Water = -1 };
using MapPoint = std::pair<VertexType, Coordinates>;

using Distance = size_t;
#define DISTANCE_MAX SIZE_MAX
using PathNode = std::pair<Distance, Coordinates>;

namespace {
VertexType CharToVertexType(char chr) {
  switch (chr) {
    case '.':
      return VertexType::Field;
    case 'W':
      return VertexType::Forest;
    case '#':
      return VertexType::Water;
    default:
      assert(false);
      return VertexType::Uninitialized;
  }
}

Distance VertexTypeToDistance(VertexType vertex) {
  return static_cast<Distance>(vertex);
}

std::vector<MapPoint> GetAdjVertices(
    const Coordinates& pos, const std::vector<std::vector<VertexType>>& map
) {
  std::vector<MapPoint> res;

  // get left
  if (pos.first != 0) {
    VertexType type = map[pos.second][pos.first - 1];
    if (type != VertexType::Water) {
      res.emplace_back(type, std::make_pair(pos.first - 1, pos.second));
    }
  }

  // get up
  if (pos.second != 0) {
    VertexType type = map[pos.second - 1][pos.first];
    if (type != VertexType::Water) {
      res.emplace_back(type, std::make_pair(pos.first, pos.second - 1));
    }
  }

  // get right
```

```
  if (pos.first != map[0].size() - 1) {
    VertexType type = map[pos.second][pos.first + 1];
    if (type != VertexType::Water) {
      res.emplace_back(type, std::make_pair(pos.first + 1, pos.second));
    }
  }

  // get down
  if (pos.second != map.size() - 1) {
    VertexType type = map[pos.second + 1][pos.first];
    if (type != VertexType::Water) {
      res.emplace_back(type, std::make_pair(pos.first, pos.second + 1));
    }
  }

  return res;
}
void Dijkstra(
    const Coordinates& start_pos,
    const std::vector<std::vector<VertexType>>& map,
    const Coordinates& destination,
    std::vector<std::vector<Distance>>& distances,
    std::vector<std::vector<Coordinates>>& predecessors
) {
  size_t rows = map.size();
  size_t cols = map[0].size();

  // Initialize distances to infinity and predecessors to (-1,-1)
  distances.assign(rows, std::vector<Distance>(cols, DISTANCE_MAX));
  predecessors.assign(rows, std::vector<Coordinates>(cols, Coordinates{-1, -1}));

  // Get start row and column
  distances[start_pos.second][start_pos.first] = 0;
  std::priority_queue<PathNode, std::vector<PathNode>, std::greater<>> paths_to_go;
  paths_to_go.emplace(0, start_pos);

  while (!paths_to_go.empty()) {
    PathNode current_path_node = paths_to_go.top();
    paths_to_go.pop();
    Coordinates current_pos = current_path_node.second;

    Distance distance_to_current_node = distances[current_pos.second][current_pos.first]

    if (current_pos == destination) {
      // early exit
      return;
    }

    for (auto adj : GetAdjVertices(current_path_node.second, map)) {
      Coordinates adj_coords = adj.second;
      Distance distance_to_adj_from_current = distances[adj_coords.second][adj_coords.fi
      Distance new_distance = distance_to_current_node + VertexTypeToDistance(adj.first)

      if (distance_to_adj_from_current > new_distance) {
        distances[adj_coords.second][adj_coords.first] = new_distance;
```

```
        predecessors[adj_coords.second][adj_coords.first] = current_pos;
        paths_to_go.emplace(new_distance, adj_coords);
      }
    }
  }
}

}  // namespace

int main() {
  Coordinates map_size;
  std::cin >> map_size.first >> map_size.second;

  Coordinates start_pos;
  std::cin >> start_pos.second >> start_pos.first;

  Coordinates destination;
  std::cin >> destination.second >> destination.first;

  // zero_based_indexing
  destination.first--;
  destination.second--;

  // zero_based_indexing
  start_pos.first--;
  start_pos.second--;

  std::vector<std::vector<VertexType>> map(
      map_size.first, std::vector<VertexType>(map_size.second)
  );

  // skip one line
  {
    std::string _;
    std::getline(std::cin, _);
  }

  for (size_t i = 0; i < map_size.first; i++) {
    std::string map_line_str;
    std::getline(std::cin, map_line_str);
    for (size_t char_index = 0; char_index < map_size.second; char_index++) {
      char current_char = map_line_str[char_index];
      map[i][char_index] = CharToVertexType(current_char);
    }
  }

  // Initialize distances and predecessors
  std::vector<std::vector<Distance>> distances;
  std::vector<std::vector<Coordinates>> predecessors;

  Dijkstra(start_pos, map, destination, distances, predecessors);

  Distance min_time = distances[destination.second][destination.first];

  if (min_time == DISTANCE_MAX) {
```

```
    std::cout << -1 << '\n';
    return 0;
  }

  std::cout << min_time << '\n';

  // Reconstruct path
  std::vector<char> path;
  Coordinates current = destination;

  while (current != start_pos) {
    // Get predecessor for current
    Coordinates pred = predecessors[current.second][current.first];
    if (pred.first == static_cast<size_t>(-1)) {
      break;  // invalid
    }

    int diff_col = (int)current.first - (int)pred.first;
    int diff_row = (int)current.second - (int)pred.second;

    if (diff_col > 0) {
      path.push_back('E');
    } else if (diff_col < 0) {
      path.push_back('W');
    } else if (diff_row > 0) {
      path.push_back('S');
    } else if (diff_row < 0) {
      path.push_back('N');
    }

    current = pred;
  }

  // Reverse path to get start to end
  std::reverse(path.begin(), path.end());

  // Output the directions
  for (char c : path) {
    std::cout << c;
  }
  std::cout << '\n';

  return 0;
}
```

2 Задача N. Свинки-копилки

Основная идея

Алгоритм поиска минимального количества разбиваемых копилок

Для решения задачи используется построение неориентированного графа и поиск его компонент связности. Каждая копилка представляет собой вершину графа, а ребро между вершинами u и v появляется, если ключ от одной копилки находится в другой. Минимальное количество разбиваемых копилок равно количеству компонент связности графа.

- **Построение графа:**
  - Для каждой копилки i определяется номер копилки j, в которой хранится её ключ.
  - Добавляется неориентированное ребро между вершинами i и j.
- **Поиск компонент связности:**
  - Инициализируется массив посещённых вершин.
  - Для каждой непосещённой вершины запускается BFS/DFS, который помечает все достижимые вершины как посещённые.
  - Количество запусков BFS/DFS соответствует числу компонент связности.

Сложность по времени:

- $O(N)$ — построение графа с N рёбрами ($E = N$).
- $O(N)$ — обход всех вершин через BFS/DFS.
- Итоговая сложность: $O(N)$, где N — количество копилок.

Сложность по памяти:

- $O(N)$ — хранение списка смежности для графа.
- $O(N)$ — массив посещённых вершин.
- Итого: $O(N)$ (доминирует размер графа и вспомогательных структур).

Код алгоритма

```
/*

количество разбитых копилок - количество висячих вершин в графе
upd: не совсем так, любая начальная точка это сломанная копилка.

а если мы ее сломали, то в любом случае можно попробовать пооткрывать ключом и другие.

еще стоит начинать разбитие копилок, с тех, которые могут открыть больше всего других.
*/

#include <algorithm>
#include <climits>
#include <cstddef>
#include <functional>
#include <iostream>
```

```cpp
#include <queue>
#include <utility>
#include <vector>

using BankIndex = uint;

namespace {
void Bfs(
    BankIndex starting_bank,
    std::vector<bool>& visited,
    const std::vector<std::vector<BankIndex>>& adj_banks
) {
  std::queue<BankIndex> banks_to_visit;
  visited[starting_bank] = true;

  banks_to_visit.push(starting_bank);
  while (!banks_to_visit.empty()) {
    BankIndex current_bank = banks_to_visit.front();
    banks_to_visit.pop();

    for (BankIndex adj_bank : adj_banks[current_bank]) {
      if (!visited[adj_bank]) {
        banks_to_visit.push(adj_bank);
        visited[adj_bank] = true;
      }
    }
  }
}
}  // namespace

int main() {
  size_t number_of_banks = 0;
  std::cin >> number_of_banks;

  std::vector<std::vector<BankIndex>> adj_banks(number_of_banks);

  for (BankIndex current_bank = 0; current_bank < number_of_banks; current_bank++) {
    BankIndex bank_with_a_key_to_current_bank = 0;
    std::cin >> bank_with_a_key_to_current_bank;

    // zero based indexing
    bank_with_a_key_to_current_bank--;
    // adj_banks[bank_with_a_key_to_current_bank].emplace_back(current_bank);
    adj_banks[current_bank].emplace_back(bank_with_a_key_to_current_bank);
  }

  std::vector<std::pair<size_t, BankIndex>> breaking_order;
  breaking_order.reserve(number_of_banks);
  for (BankIndex current_bank = 0; current_bank < number_of_banks; current_bank++) {
    breaking_order.emplace_back(adj_banks[current_bank].size(), current_bank);
  }
  std::sort(breaking_order.begin(), breaking_order.end(), std::greater<>());

  std::vector<bool> visited(number_of_banks);
  size_t broken_banks_to_get_starting_key = 0;
```

```
  for (auto starting_bank_pair : breaking_order) {
    BankIndex starting_bank = starting_bank_pair.second;
    if (visited[starting_bank]) {
      continue;
    }
    // if its not visited then we have to break it to use the key
    broken_banks_to_get_starting_key++;
    Bfs(starting_bank, visited, adj_banks);
  }

  std::cout << broken_banks_to_get_starting_key << '\n';

  return 0;
}
```

3 Задача O. Долой списывание!

Пояснение к примененному алгоритму

Для проверки двудольности графа используется обход в ширину (BFS). Основные шаги:

- Граф представляется в виде списка смежности для эффективного перебора соседних вершин.
- Инициализируется массив цветов вершин (`student_color`), где каждый элемент может быть неопределенным, черным или красным.
- Последовательно обрабатываются все компоненты связности графа:
  - Выбирается стартовая вершина. Если она ещё не окрашена, ей присваивается цвет (например, черный).
  - Запускается BFS: вершины обрабатываются по уровням, соседние окрашиваются в противоположный цвет.
  - При обнаружении ребра между вершинами одного цвета граф признается недвудольным.

Сложность по времени:

- $O(N + M)$ — каждая вершина и ребро обрабатываются ровно один раз.
  - Обработка вершин: $O(1)$ на добавление/удаление из очереди BFS.
  - Обработка рёбер: $O(1)$ на проверку цвета смежной вершины.

Сложность по памяти:

- $O(N)$ — массив цветов `student_color`.
- $O(N + M)$ — список смежности `adjacent_students`.
- $O(N)$ — очередь BFS для хранения вершин на текущем уровне.
- Итого: $O(N + M)$ (доминируют размеры списка смежности и массива цветов).

Код алгоритма

```
/*

"Требуется определить, сможет ли он разделить лкшат на две группы так,
чтобы любой обмен записками осуществлялся от лкшонка одной группы лкшонку другой группы.

то есть нужно проверить граф на двудольность.

главное условий двудольности: не должно быть ребер, которые находятся в одной доли.

можно воспользоваться bfs, причем каждый раз все вершины на одном уровне будут всегда ок
один цвет если это не так, то мы знаем, что граф не двудольный.

граф может быть не связным, поэтому мы пройдемся по каждой вершине как по начальной.

*/

#include <cassert>
```

```cpp
#include <cstddef>
#include <cstdint>
#include <iostream>
#include <queue>
#include <vector>

using StudentIndex = uint;

enum class Color : uint8_t { Undefined = 0, Black, Red };

namespace {
bool IsVisited(StudentIndex student, const std::vector<Color>& student_color) {
  return student_color[student] != Color::Undefined;
}
Color ChooseNextColor(Color color) {
  assert(color != Color::Undefined);

  return static_cast<Color>(3 - static_cast<uint8_t>(color));
}

bool Bfs(
    StudentIndex starting_student,
    const std::vector<std::vector<StudentIndex>>& adjacent_students,
    std::vector<Color>& student_color
) {
  // if we are here, this means that we are inside another connected component,
  // so we can color the first vertex anything we want, since it wont affect other compo
  std::queue<StudentIndex> students_to_visit;
  students_to_visit.push(starting_student);
  student_color[starting_student] = Color::Black;

  while (!students_to_visit.empty()) {
    StudentIndex current_student = students_to_visit.front();
    Color current_student_color = student_color[current_student];
    students_to_visit.pop();
    Color next_color = ChooseNextColor(current_student_color);

    for (StudentIndex adj_student : adjacent_students[current_student]) {
      if (!IsVisited(adj_student, student_color)) {
        students_to_visit.push(adj_student);
        student_color[adj_student] = next_color;
      } else {
        if (student_color[adj_student] == current_student_color) {
          return false;
        }
      }
    }
  }

  return true;
}

bool IsBipartite(
    const std::vector<std::vector<StudentIndex>>& adjacent_students, size_t number_of_st
) {
```

```cpp
  // also can be used as a visited list
  std::vector<Color> student_color(number_of_students);

  for (StudentIndex starting_student = 0; starting_student < number_of_students;
       starting_student++) {
    if (IsVisited(starting_student, student_color)) {
      continue;
    }

    if (!Bfs(starting_student, adjacent_students, student_color)) {
      return false;
    }
  }

  return true;
}

}  // namespace

int main() {
  size_t number_of_students = 0;
  std::cin >> number_of_students;

  size_t number_of_pairs = 0;
  std::cin >> number_of_pairs;

  std::vector<std::vector<StudentIndex>> adjacent_students(
      number_of_students, std::vector<StudentIndex>()
  );

  for (size_t i = 0; i < number_of_pairs; i++) {
    StudentIndex first = 0;
    StudentIndex second = 0;
    std::cin >> first >> second;

    // zero based indexing
    first--;
    second--;

    adjacent_students[first].emplace_back(second);
    adjacent_students[second].emplace_back(first);
  }

  bool res = IsBipartite(adjacent_students, number_of_students);

  if (res) {
    std::cout << "YES" << '\n';
  } else {
    std::cout << "NO" << '\n';
  }
}
```

4 Задача P. Авиаперелёты

Пояснение к примененному алгоритму

Алгоритм решения задачи P. Авиаперёты

Для определения минимального размера топливного бака, позволяющего самолёту летать между любыми городами, используется бинарный поиск по ответу. Условие достижимости всех городов проверяется с помощью BFS, учитывая рёбра с весом не более текущего значения `mid`.

- Инициализация границ бинарного поиска:
  - `low` = 0 (минимально возможное значение).
  - `high` = максимальный вес в матрице (начальный верхний предел).
- На каждой итерации бинарного поиска:
  - Вычисляется `mid` = (`low` + `high`) / 2.
  - Проверяется, является ли граф сильно связным при использовании рёбер с весом ≤ `mid`.
    - Проверка осуществляется двумя BFS:
      - В оригинальной матрице (прямые рёбра).
      - В транспонированной матрице (обратные рёбра).
    - Если оба BFS покрывают все вершины, граф сильно связен.
  - Если граф сильно связен:
    - Обновляется `high` = `mid` - 1.
    - Текущий `mid` сохраняется как возможный ответ.
  - Иначе:
    - Обновляется `low` = `mid` + 1.
- Алгоритм завершается, когда `low` > `high`. Возвращается последнее сохранённое значение `answer`.

Сложность по времени:

- $O(\log(\texttt{max\_weight}) \cdot n^2)$, где n — количество городов, `max_weight` — максимальный вес в матрице.
  - Каждая итерация бинарного поиска ($O(\log(10^9)) \approx 30$ итераций) требует $O(n^2)$ времени на проверку связности.
  - Проверка связности включает два BFS, каждый из которых работает за $O(n + e)$, где e — количество рёбер ≤ `mid`. В полном графе $e = O(n^2)$, но в коде проверка реализована через цикл по всем вершинам, что даёт $O(n^2)$ на каждую проверку.

Сложность по памяти:

- $O(n^2)$ — хранение матрицы смежности.
- $O(n)$ — вспомогательные структуры данных (`visited`, очередь BFS).
- Итого: $O(n^2)$ (доминирует матрица смежности).

Код алгоритма

```cpp
#include <algorithm>
#include <climits>
#include <cstddef>
#include <iostream>
#include <queue>
#include <vector>

/*

This problem has a full graph with weighed edges

approach is to for each vertex leave the smallest weighted vertex

but then there is a chance that there will be floating vertexes left out

We need to find smallest possible subtree by sum, then take its max edge

Why tree?

Suppose that we have a loop inside a graph,
then to travel to each city of this loop we need to
consider just the max edge of this graph

we dont care about the sum, we just need the least volume fuel tank

then each loop inside a graph can be resolved to a tree,
hence the left out graph of optimal travels can't have a loop inside of it.

then graph of optimal travels is a tree, constructed from a full graph. a sub tree.

we need to use primes algorithm to create a least sum subtree inside this full graph.

нам выгодно использовать использовать такую структуру данных, чтобы быстро находить мини
возможное ребро внутри вершины, при этом нужно учесть что мы будем хранить все использов

алгоритм Прима не сработал, потому что матрица не симметрична.

будем делать бин поиск по ответу

dfs оказался слишком медленным, попробую bfs

- спросить почему uint16_t и uint32_t не поместились а обычные инты поместились

*/

using CityIndex = uint;
using EdgeWeight = int;

namespace {

bool BfsAllVerticesWithEdgesLowerThan(
    EdgeWeight upper_bound, const std::vector<std::vector<EdgeWeight>>& matrix, bool tra
) {
```

```cpp
  auto vertices_count = static_cast<CityIndex>(matrix.size());

  std::vector<bool> visited(vertices_count, false);
  std::queue<CityIndex> vertices_to_visit;
  vertices_to_visit.push(0);
  visited[0] = true;
  size_t visited_count = 1;

  while (!vertices_to_visit.empty()) {
    CityIndex current_city = vertices_to_visit.front();
    vertices_to_visit.pop();

    for (CityIndex adj_city = 0; adj_city < vertices_count; adj_city++) {
      if (visited[adj_city] || adj_city == current_city) {
        continue;
      }
      EdgeWeight weight =
          transpose ? matrix[current_city][adj_city] : matrix[adj_city][current_city];

      if (weight > upper_bound) {
        continue;
      }

      visited[adj_city] = true;
      visited_count++;
      vertices_to_visit.push(adj_city);
      if (vertices_count == visited_count) {
        return true;
      }
    }
  }
  return visited_count == vertices_count;
}

bool IsStronglyConnected(
    EdgeWeight upper_bound, const std::vector<std::vector<EdgeWeight>>& matrix
) {
  return BfsAllVerticesWithEdgesLowerThan(upper_bound, matrix, false) &&
         BfsAllVerticesWithEdgesLowerThan(upper_bound, matrix, true);
}
}  // namespace

int main() {
  CityIndex number_of_cities = 0;
  std::cin >> number_of_cities;

  std::vector<std::vector<EdgeWeight>> matrix(
      number_of_cities, std::vector<EdgeWeight>(number_of_cities)
  );

  EdgeWeight max_weight = 0;
  for (CityIndex i = 0; i < number_of_cities; ++i) {
    for (CityIndex j = 0; j < number_of_cities; ++j) {
      std::cin >> matrix[i][j];
      if (i != j) {
```

```
        max_weight = std::max(max_weight, matrix[i][j]);
      }
    }
  }

  EdgeWeight high = max_weight;
  EdgeWeight low = 0;

  EdgeWeight answer = 0;

  while (low <= high) {
    EdgeWeight mid = (high + low) / 2;
    if (IsStronglyConnected(mid, matrix)) {
      answer = mid;
      high = mid - 1;
    } else {
      low = mid + 1;
    }
  }

  std::cout << answer << '\n';

  return 0;
}
```