МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Лабораторна робота №2-3

з дисципліни «Програмне забезпечення комп'ютерних систем - 2»

на тему: «Формування черги задач»

Виконав:
студент 5-го курсу ФІОТ
групи ІО – 82мп
Барабаш Т.А.

Перевірила:
доцент
Русанова О. В.

Київ - 2019

ЛАБОРАТОРНА РОБОТА №2-3

Формування черги задач

Мета: Реалізувати перший етап планування шляхом формування черг обчислювальних робіт.

I. Завдання

Номер	Складність	Метод формування черг	Використанні характеристики графа
1	4	У порядку спадання пронормованої суми критичних по часу і по кількості вершин шляхів до кінця графа задачі.	Критичний шлях графа та вершин по часу.
11	4	У порядку спадання зв'язності вершин, а при рівних значеннях – в порядку зростання критичного по кількості вершин шляхів від початку графа задачі.	Критичний шлях по кількості вершин та їх зв'язність

II. Хід роботи

Алгоритм №1:

$\mathrm{Pr}_i = \frac{\boldsymbol{T}\text{кр}_{ik}}{\boldsymbol{T}\text{кр}_{\text{графу}}} + \frac{\boldsymbol{N}\text{кр}_{ik}}{\boldsymbol{N}\text{кр}_{\text{графу}}}$, де $\mathbf{T}\text{кр}_{\mathbf{ik}}$- сума ваг вершин найдовшого шляху до кінцевих вершин, $\boldsymbol{T}\text{кр}_{\text{графу}}$ – максимальне значення $\mathbf{T}\text{кр}_{\mathbf{ik}}$, $\boldsymbol{N}\text{кр}_{\boldsymbol{ik}}$ – максимальна кількість вершин на шляху до кінцевих вершин графу, $\boldsymbol{N}\text{кр}_{\text{графу}}$ - максимальне значення $\mathbf{N}\text{кр}_{\mathbf{ik}}$

1. Знайти кінцеві вершини графу.
2. Для кожної із вершин графу знайти усі шляхи до кінцевих вершин графу. Обрати найдовший шлях та обчислити його вартість. Крім того, прийняти $\boldsymbol{N}\text{кр}_{\boldsymbol{ik}}$ як кількість вершин у шляху.
3. Знайти найдовший шлях та прийняти його за $\boldsymbol{T}\text{кр}_{\text{графу}}$, а $\boldsymbol{N}\text{кр}_{\text{графу}}$ обрати як максимальне значення $\boldsymbol{N}\text{кр}_{\boldsymbol{ik}}$.
4. Обчислити за формулою вище пріоритети для кожної із вершин та відсортувати їх у порядку спадання.

Алгоритм №11:

1. Для кожної вершини обчислити кількість вхідних та вихідних дуг.
2. Відсортувати отримані значення у порядку спадання.
3. У випадку якщо значення однакові, знайти початкові вершини та використовуючи частину алгоритму №1 (пп. 2-3), обчислити критичне по кількості вершин від початку графу $\boldsymbol{N}\text{кр}_{\boldsymbol{i}_{\text{п}}}$. Відсортувати отримані значення у порядку зростання.

III. Код програми

```
class Queue {
  constructor(method, data) {
    this._method = method;
    this._data = data;
  }

  run() {
    let result;
    switch (this._method) {
      case 1:
        result = this._run1();
        break;
      case 11:
        result = this._run11();
        break;
      default:
        throw new Error(`Unsupported algo`);
    }
    return result;
  }

  _toMatrix(reverse) {
    const { nodes, edges } = this._data;
    const matrix = [];
    edges.forEach(edge => {
      if (!reverse)
        matrix[edge.from] = {
          ...matrix[edge.from],
          [edge.to]: nodes.get(edge.to).weight
        };
      else
        matrix[edge.to] = {
          ...matrix[edge.to],
          [edge.from]: nodes.get(edge.from).weight
        };
    });
    return matrix;
  }

  _toEdgesArray(reverse) {
    const { edges } = this._data;
    const graph = [];
    edges.forEach(edge => {
      graph.push(!reverse ? [edge.from, edge.to] : [edge.to, edge.from]);
    });
    return graph;
  }

  _toUndirectedMatrix() {
    const { nodes, edges } = this._data;
    const matrix = [];
    edges.forEach(edge => {
      matrix[edge.to] = {
        ...matrix[edge.to],
        [edge.from]: nodes.get(edge.from).weight
      };
      matrix[edge.from] = {
        ...matrix[edge.from],
        [edge.to]: nodes.get(edge.to).weight
      };
    });
    return matrix;
  }

  _getPaths(from, to, reverse) {
    return paths({ graph: this._toEdgesArray(reverse), from, to });
  }

  _BFS(reverse = false) {
    const { nodes } = this._data;
    const allNodes = Object.keys(nodes._data);
    const matrixNodes = Object.keys(this._toMatrix(reverse));
    const endNodes = allNodes.filter(item => !matrixNodes.includes(item));
```

```
    const weights = {};
    const maxPaths = {};
    matrixNodes.forEach(i => {
      i = parseInt(i);
      let maxWeight = 0;
      let maxPath = [];
      endNodes.forEach(j => {
        j = parseInt(j);
        const paths = this._getPaths(i, j, reverse);
        paths.forEach(path => {
          const weight = path.reduce(
            (sum, node) => sum + nodes.get(node).weight,
            reverse ? -nodes.get(i).weight : 0
          );
          if (maxWeight < weight) {
            maxWeight = weight;
            maxPath = path;
          }
        });
        weights[j] = !reverse ? nodes.get(j).weight : 0;
        maxPaths[j] = [j];
      });
      weights[i] = maxWeight;
      maxPaths[i] = maxPath;
    });
    return { weights, maxPaths };
  }

  _run1() {
    const { nodes } = this._data;
    const { weights, maxPaths } = this._BFS();
    const graphWeight = Math.max(...Object.values(weights));
    const lengthes = Object.values(maxPaths).map(i => i.length);
    const graphLength = Math.max(...lengthes);
    let result = [];
    for (let i = 0; i < nodes.length; i++) {
      const value = weights[i] / graphWeight + lengthes[i] / graphLength;
      result.push({ number: i, value, names: [`Pr`] });
    }
    const sortedResult = result.sort((a, b) => b.value - a.value);
    return sortedResult;
  }

  _run11() {
    const adjMatrix = this._toUndirectedMatrix();
    const { nodes } = this._data;
    const allNodes = Object.keys(nodes._data);
    const { weights } = this._BFS(true);
    const result = allNodes.map(node => {
      return {
        number: node,
        value: adjMatrix[node] ? Object.keys(adjMatrix[node]).length : 0,
        weights: weights[node],
        names: [`Sv`, `Ткр.поч.`]
      };
    });
    const sortedResult = result.sort((a, b) => {
      if (a.value === b.value) {
        return a.weights - b.weights;
      }
      return b.value - a.value;
    });
    return sortedResult;
  }
}

function paths({ graph = [], from, to }, path = []) {
  const linkedNodes = memoize(nodes.bind(null, graph));
  return explore(from, to);

  function explore(currNode, to, paths = []) {
    path.push(currNode);
    for (let linkedNode of linkedNodes(currNode)) {
      if (linkedNode === to) {
        let result = path.slice();
        result.push(to);
        paths.push(result);
```

```
        continue;
      }
      if (
        !hasEdgeBeenFollowedInPath({
          edge: {
            from: currNode,
            to: linkedNode
          },
          path
        })
      ) {
        explore(linkedNode, to, paths);
      }
    }
    path.pop();
    return paths;
  }
}
function nodes(graph, node) {
  return graph.reduce((p, c) => {
    c[0] === node && p.push(c[1]);
    return p;
  }, []);
}
function hasEdgeBeenFollowedInPath({ edge, path }) {
  var indices = allIndices(path, edge.from);
  return indices.some(i => path[i + 1] === edge.to);
}
function allIndices(arr, val) {
  var indices = [],
    i;
  for (i = 0; i < arr.length; i++) {
    if (arr[i] === val) {
      indices.push(i);
    }
  }
  return indices;
}
function memoize(fn) {
  const cache = new Map();
  return function() {
    var key = JSON.stringify(arguments);
    var cached = cache.get(key);
    if (cached) {
      return cached;
    }
    cached = fn.apply(this, arguments);
    cache.set(key, cached);
    return cached;
  };
}

export default Queue;
```