**Санкт-Петербургский национальный исследовательский университет**

**информационных технологий, механики и оптики**

**Кафедра информатики и прикладной математики**

Алгоритмы и структуры данных

Лабораторная работа №4

Вариант 3

Выполнил: Гхази Даниэль

Группа P3218

Преподаватель: Зинчик А.А.

2017 г.

**Текст задания**

1. Самостоятельно задать пропускные способности дуг и построить максимальный поток в транспортной сети.

2. Найти минимальный разрез сети и проверить справедливость теоремы Форда – Фалкерсона.



**Текст программы**

#include "../Headers/findMaxFlow.h"

int findMaxFlow(GraphParameters \*graphParameters, std::vector <std::vector <Edge> > &listsOfEdges)

{

int maxFlow = 0;

for (int i = 1; i < listsOfEdges.size(); i++)

{

for (int j = 0; j < listsOfEdges[i].size(); j++)

{

listsOfEdges[i][j].flow = 0;

}

}

// Cycles until there are no enlargeable paths.

while (true)

{

// Keeps information about visited vertices of graph. Used in DFS algorithm.

std::vector<Visitation> visitations(graphParameters->numberOfVertices + 1, UNVISITED);

// Keeps unfinished paths found during DFS, including one completed.

std::vector<ElementOfPath> prior;

bool pathFound = false;

// A is first vertex in variant 3. It does not have parent, so second parameter is empty edge.

ElementOfPath emptyEdge;

ElementOfPath firstEdge;

firstEdge.thisVertex = A;

DFS(firstEdge, emptyEdge, listsOfEdges, visitations, prior, &pathFound, 0);

if (!pathFound) return maxFlow;

// Keeps current enlargeable path, which is used to increase the flow.

std::vector<ElementOfPath> path;

path = getPath(prior);

// Keeps pointers to edges in listsOfEdges, so we can increase and decrease flow in them directly.

std::vector<Edge \*> edgesInPath;

int minFlow = findMinFlow(listsOfEdges, path, edgesInPath);

std::vector<VertexName> wasInPath;

// Decreasing flow in backward edges and increasing in forward.

for (int i = 0; i < edgesInPath.size(); i++)

{

if (std::find(wasInPath.begin(), wasInPath.end(), edgesInPath[i]->vertex) != wasInPath.end())

{

if (edgesInPath[i]->capacity == 0) edgesInPath[i]->flow += minFlow;

else edgesInPath[i]->flow -= minFlow;

}

else if (edgesInPath[i]->capacity == 0)

{

edgesInPath[i]->flow -= minFlow;

wasInPath.push\_back(edgesInPath[i]->vertex);

}

else

{

edgesInPath[i]->flow += minFlow;

wasInPath.push\_back(edgesInPath[i]->vertex);

}

}

maxFlow += minFlow;

}

}

// If enlargeable path is found returns true,

// else returns false and algorithm is terminated.

void DFS(ElementOfPath currentVertex, ElementOfPath previousVertex, std::vector <std::vector <Edge> > &listsOfEdges,

std::vector <Visitation> &visitations, std::vector <ElementOfPath> &prior, bool \*pathFound, int forMinCutVersion)

{

if (visitations[currentVertex.thisVertex] == VISITED) return;

visitations[currentVertex.thisVertex] = VISITED;

prior.push\_back(previousVertex);

currentVertex.from = previousVertex.thisVertex;

// F is the last vertex in variant 3.

if (currentVertex.thisVertex == F)

{

prior.push\_back(currentVertex);

\*pathFound = true;

return;

}

// Cycling through each edge.

// If capacity = 0, the edge is backward, so we should check if we can return some flow back, redirecting it later.

// Else edge is forward, so we check space left for extra flow.

for (int i = 0; i < listsOfEdges[currentVertex.thisVertex].size(); i++)

{

Edge currentEdge = listsOfEdges[currentVertex.thisVertex][i];

if (currentEdge.capacity == 0)

{

if (currentEdge.flow > 0)

{

ElementOfPath nextElement;

nextElement.capacity = listsOfEdges[currentVertex.thisVertex][i].capacity;

nextElement.thisVertex = listsOfEdges[currentVertex.thisVertex][i].vertex;

DFS(nextElement, currentVertex, listsOfEdges, visitations, prior, pathFound, forMinCutVersion);

}

}

else if (currentEdge.capacity - currentEdge.flow + forMinCutVersion > 0)

{

ElementOfPath nextElement;

nextElement.capacity = listsOfEdges[currentVertex.thisVertex][i].capacity;

nextElement.thisVertex = listsOfEdges[currentVertex.thisVertex][i].vertex;

DFS(nextElement, currentVertex, listsOfEdges, visitations, prior, pathFound, forMinCutVersion);

}

// Used for fast interruption of all higher level function calls after path was found.

if (\*pathFound) return;

}

}

// Extracting complete path and returning it.

std::vector<ElementOfPath> getPath(std::vector <ElementOfPath> &prior)

{

std::vector<ElementOfPath> path;

path.push\_back(prior.back());

while (path.back().from != NOT\_A\_VERTEX)

{

for (int i = prior.size() - path.size() - 1; i > 0; i--)

{

if (prior[i].thisVertex == path.back().from)

{

path.push\_back(prior[i]);

break;

}

}

}

std::reverse(path.begin(), path.end());

return path;

}

int findMinFlow(std::vector <std::vector <Edge> > &listsOfEdges, std::vector<ElementOfPath> &path,

std::vector<Edge \*> &edgesInPath)

{

int minFlow = INT32\_MAX;

// Index of current element of path.

int index = 0;

outOfForLoop: while (++index < path.size())

{

for (int j = 0; j < listsOfEdges[path[index].from].size(); j++)

{

if ((path[index].thisVertex == listsOfEdges[path[index].from][j].vertex) &&

(path[index].capacity == listsOfEdges[path[index].from][j].capacity))

{

edgesInPath.push\_back(&listsOfEdges[path[index].from][j]);

// Adding connected backward/forward edge to change its flow later.

if (path[index].thisVertex != F)

{

for (int k = 0; k < listsOfEdges[path[index].thisVertex].size(); k++)

{

if (listsOfEdges[path[index].thisVertex][k].vertex == path[index].from)

{

if ((path[index].capacity == 0) && (listsOfEdges[path[index].thisVertex][k].capacity != 0))

{

edgesInPath.push\_back(&listsOfEdges[path[index].thisVertex][k]);

break;

}

else if (listsOfEdges[path[index].thisVertex][k].capacity == 0)

{

edgesInPath.push\_back(&listsOfEdges[path[index].thisVertex][k]);

break;

}

}

}

}

if (listsOfEdges[path[index].from][j].capacity == 0)

{

if (listsOfEdges[path[index].from][j].flow < minFlow)

{

minFlow = listsOfEdges[path[index].from][j].flow;

}

}

else if (listsOfEdges[path[index].from][j].capacity - listsOfEdges[path[index].from][j].flow < minFlow)

{

minFlow = listsOfEdges[path[index].from][j].capacity - listsOfEdges[path[index].from][j].flow;

}

goto outOfForLoop;

}

}

}

return minFlow;

}

**Вывод**

В результате выполнения лабораторной работы были реализованы алгоритмы поиска максимального потока в транспортной сети и минимального разреза.

При реализации алгоритма поиска минимального разреза удобно пользоваться полученным значениям максимально потока. Были трудности с поиском алгоритма для нахождения минимального разреза. Алгоритма Каргера позволяет найти минимальный разрез лишь с определенной долей вероятности.

Было выяснено, что все ребра в минимальном разрезе должны иметь поток = вместимости ребра. Таким образом, можно сильно уменьшить количество ребер, которые необходимо просмотреть для нахождения минимального разреза.

Учитывая небольшой размер графа в данной лабораторной работе, был использован алгоритм перебора всех возможных комбинаций ребер с потоком = вместимости.