**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**КАФЕДРА САПР**

**ОТЧЕТ**

**по курсовой работе**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: Потоки в сетях**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 9301 |  | Синицкая В. А. |
| Преподаватель |  | Тутуева А. В. |

Санкт-Петербург

2021

Оглавление

[1 Постановка задачи 3](#_Toc71741216)

[2 Реализуемые классы и методы 3](#_Toc71741217)

[3 Оценка временной сложности каждого метода 4](#_Toc71741218)

[4 Описание реализованных Unit test 5](#_Toc71741219)

[5 Пример работы программы 5](#_Toc71741220)

[5.1 TEST\_METHOD(TestMethodMaxFlow\_6\_vertexes) 5](#_Toc71741221)

[5.2 TEST\_METHOD(TestMethodMaxFlow\_7\_vertexes) 5](#_Toc71741222)

[6 Листинг 7](#_Toc71741223)

[6.1 Element.h 7](#_Toc71741224)

[6.2 Queue.h 7](#_Toc71741225)

[6.3 Edge.h 10](#_Toc71741226)

[6.4 Vertex.h 10](#_Toc71741227)

[6.5 Graph.h 11](#_Toc71741228)

[6.6 Graph.cpp 12](#_Toc71741229)

[6.7 UnitTest\_EdmondsKarp.cpp 16](#_Toc71741230)

# Постановка задачи

20 вариант => 2 вариант:

Найти максимальный поток в сети, используя алгоритм Эдмондса-Карпа.

# Реализуемые классы и методы

В проекте содержатся классы Element, Queue, Vertex, Edge, Graph, все классы, кроме Graph, реализованы как шаблонные.

Класс Element содержит поля data и next, где data — имеет тип указателя на пользовательский тип данных, next — ссылка на следующий элемент.

Класс Queue представляет собой очередь со стандартным набором полей и функций, а также имеет функцию push\_front, добавляющую Element в начало очереди. При добавлении элементов в функцию передается содержимое, а Element, который будет его содержать, создается новый.

Класс Edge шаблонный по двум типам: по типу названия вершин (в данном случае char, но могут быть и string) и по типу данных пропускного потока (может быть целый, а может быть double, как в данной реализации Graph). Класс имеет 4 поля: название вершин, откуда и куда направлено ребро, поток, проходящий через ребро и пропускная способность ребра.

Класс Vertex подобно классу Edge является шаблонным по двум типам, имеет три поля: название вершины, пометку — указатель на вершину, из которой рассматриваемый на данный момент поток пришел в текущую вершину, и очередь из ребер, которые выходят из текущей вершины.

Класс Graph является дружественным к классам Element, Vertex и Edge. Класс содержит одно поле — очередь из вершин (множество вершин V), каждая из которых хранит список своих ребер (граф хранится как список ребер). Graph имеет:

1. конструктор по умолчанию;
2. функцию clear\_marks, очищающую все пометки в множестве вершин;
3. функцию clear\_flows, очищающую все потоки в графе (на случай, если это будет нужно позже, для решения задачи не используется, может быть удалена или закомментирована);
4. функцию search\_vertex\_on\_name, которая возвращает новый Element, содержащий вершину из множества V с заданным именем, в случае отсутствия искомой вершины возвращает nullptr (это используется);
5. функцию search\_edge, которая возвращает новый Element, содержащий ребро, которое начинается с заданной вершины и заканчивается на вершину с заданным именем. В случае отсутствия искомого ребра генерируется исключение;
6. функцию read\_from\_file, которая считывает информацию о графе из файла с указанным названием и заполняет граф;
7. функцию print, которая печатает ребра в формате: начальная вершина, конечная вершина, поток через ребро/пропускная способность ребра, ребра могут располагаться не в том порядке, в котором они были в файле для считывания;
8. функцию edmonds\_karp, выполняющую алгоритм Эдмондса-Карпа на текущем графе, функция изменяет потоки в настоящем графе и возвращает число, обозначающее найденный максимальный поток в сети. Алгоритм работает, находя кратчайший по количеству ребер путь и пуская по нему максимально возможный поток. Кратчайший путь находится путем поиска в ширину, если таких путей несколько, будет выбран первый по порядку прохода. Если какой-то путь приходит в уже помеченную вершину, то он больше не считается путем, так как он заведомо длиннее какого-то другого уже рассматриваемого на данной итерации пути. Путь восстанавливается из конца в начало по пометкам, после нахождения каждого такого пути изменяются величины потоков вдоль него на максимально возможное количество и искомый максимальный поток увеличивается на данный поток. Когда пути больше нет, алгоритм завершает работу.

# Оценка временной сложности каждого метода

Далее обозначается: V — количество вершин, E — количество ребер графа.

1. clear\_marks — O(V)
2. clear\_flows — O(E)
3. search\_vertex\_on\_name — O(V)
4. search\_edge — O(E)
5. read\_from\_file — O(E)
6. print — O(E)
7. edmonds\_karp — O(V\*E^2)

# Описание реализованных Unit test

Была проверена работа методов print и read\_from\_file, была проверена работа алгоритма Эдмондса-Карпа на графах разного вида, с разными потоками.

# Пример работы программы

Примером работы послужат результаты выполнения некоторых тестов:

## TEST\_METHOD(TestMethodMaxFlow\_6\_vertexes)

На Рис. 5.1.1 показан ввод, записанный в файл in.txt и вывод, записанный в файл out.txt, где можно посчитать максимальный поток как сумму исходящих из S потоков (5) или как сумму входящих в Т потоков (5), максимальный поток проверяется в самом тесте, в файл не выводится.

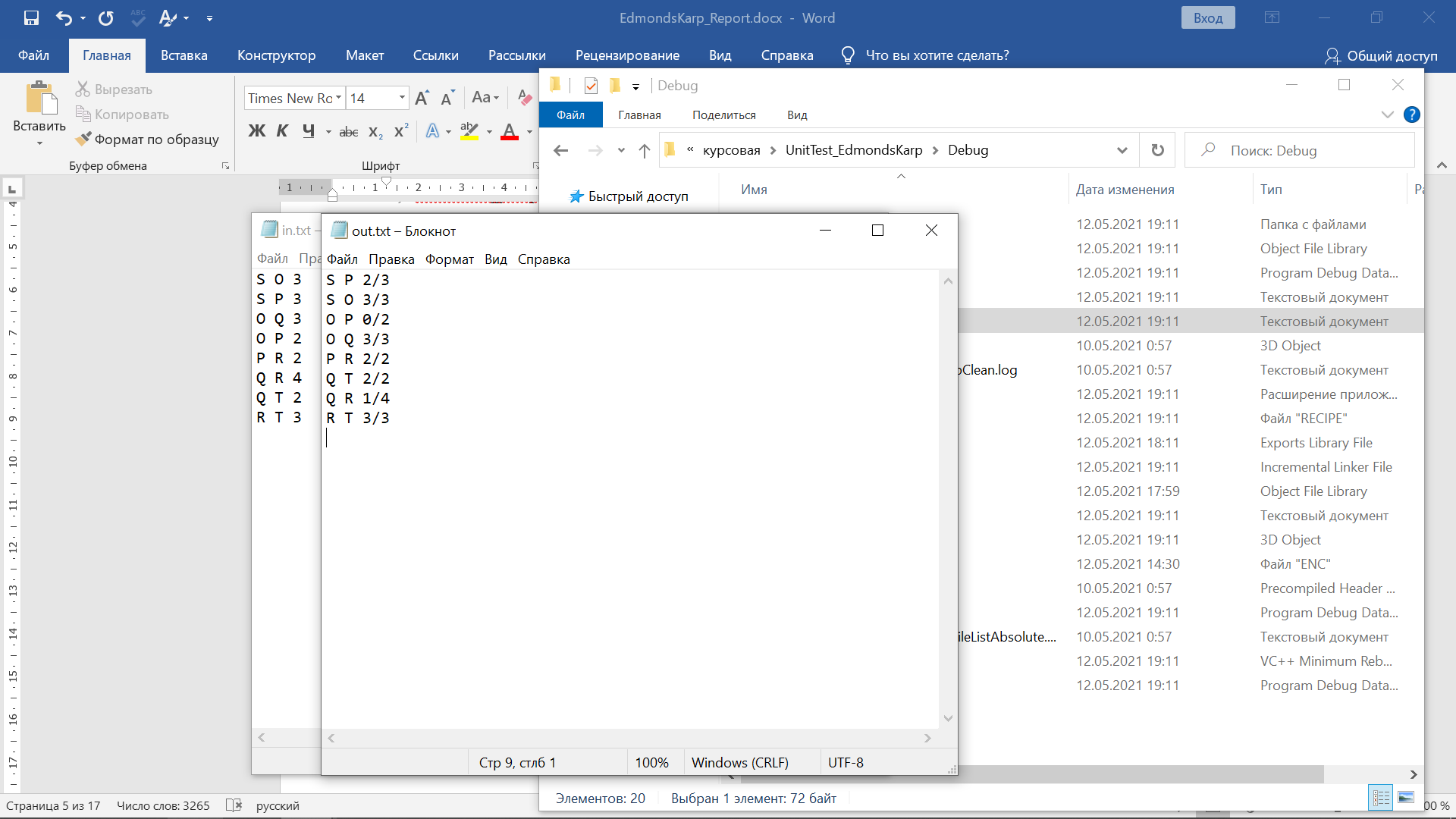


Рис. 5.1.1 — входные и выходные данные для примера 1.

## TEST\_METHOD(TestMethodMaxFlow\_7\_vertexes)

На Рис. 5.2.1 показан результат работы теста TestMethodMaxFlow\_7\_vertexes.

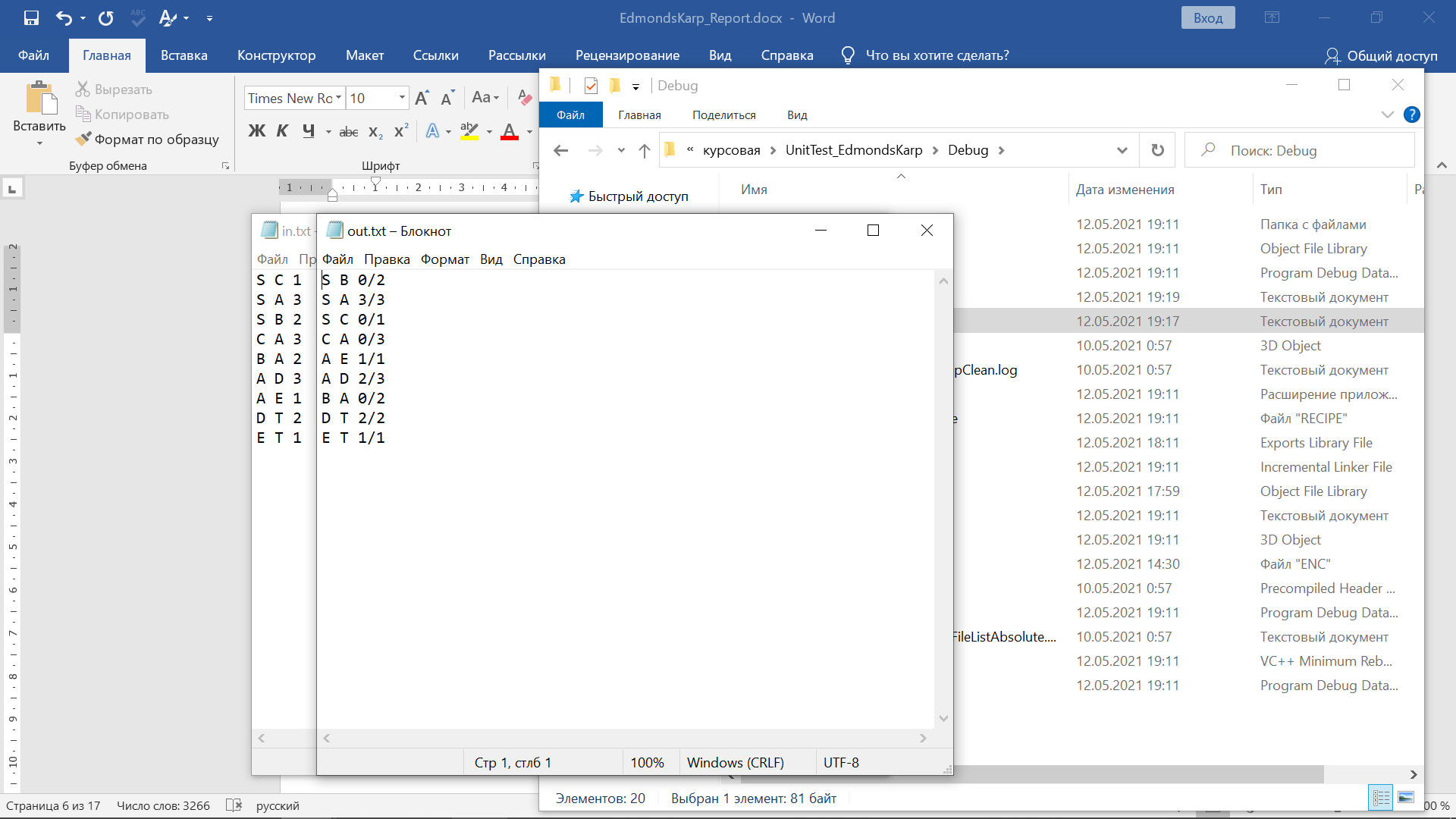


Рис. 5.2.1 — входные и выходные данные для примера 2.

# Листинг

## Element.h

1. #pragma once
2. #pragma once
3. #include <iostream>
4. using namespace std;
5. template <class Value\_Type>
6. class Element
7. {
8. Element<Value\_Type>\* next;
9. Value\_Type\* data;
10. void set\_next(Element<Value\_Type>\* new\_element) { next = new\_element; }
11. void set\_data(Value\_Type\* new\_data) { data = new\_data; }
12. Element()
13. {
14. next = nullptr;
15. data = nullptr;
16. }
17. public:
18. Element(Value\_Type\* data)
19. {
20. set\_data(data);
21. next = nullptr;
22. }
23. Element<Value\_Type>\* get\_next() { return next; }
24. Value\_Type\* get\_data() { return data; }
25. ~Element() {}
26. template <class Value\_Type> friend class Queue;
27. friend class Graph;
28. };

## Queue.h

1. #pragma once
2. #pragma once
3. #include "Element.h"
4. template <class Value\_Type>
5. class Queue
6. {
7. private:
8. Element<Value\_Type>\* head; // the first in queue, leaves first
9. Element<Value\_Type>\* tail; // the last, new in queue
10. void set\_head(Element<Value\_Type>\* new\_data) { head->set\_data(new\_data); }
11. void set\_tail(Element<Value\_Type>\* new\_data) { tail->set\_data(new\_data); }
12. public:
14. Queue()
15. {
16. head = nullptr;
17. tail = nullptr;
18. }
19. bool is\_empty() // returns true if queue is empty
20. {
21. if (tail == nullptr)
22. return true;
23. return false;
24. }
25. int size() // return a number of elements in queue
26. {
27. if (is\_empty())
28. return 0;
29. int size = 1;
30. for (Element<Value\_Type>\* now = tail; now->get\_next() != nullptr; now = now->get\_next())
31. size++;
32. return size;
33. }
34. Element<Value\_Type>\* top() // to know who will leave first
35. {
36. return head;
37. }
39. Element<Value\_Type>\* get\_tail() // to know who is the last
40. {
41. return tail;
42. }
44. void push(Value\_Type\* data) // to add in the end of the queue the element with key-data and return it to user
45. {
46. Element<Value\_Type>\* new\_element = new Element<Value\_Type>(data);
47. if (is\_empty())
48. head = tail = new\_element;
49. else
50. {
51. new\_element->set\_next(tail);
52. tail = new\_element;
53. }
54. }
56. void push\_front(Value\_Type\* data)
57. {
58. Element<Value\_Type>\* new\_element = new Element<Value\_Type>(data);
59. if (is\_empty())
60. head = tail = new\_element;
61. else
62. {
63. head->set\_next(new\_element);
64. head = new\_element;
65. }
66. }
68. Element<Value\_Type>\* pop() // to delete first in queue and return it to user
69. {
70. if (is\_empty())
71. throw out\_of\_range("the queue is empty");
72. Element<Value\_Type>\* to\_delete(head);
73. if (size() == 1)
74. {
75. head = tail = nullptr;
76. }
77. else
78. {
79. for (Element<Value\_Type>\* now(tail); now->get\_next() != nullptr; now = now->get\_next())
80. head = now;
81. head->next = nullptr;
82. }
83. return to\_delete;
84. }
86. ~Queue()
87. {
88. while (!is\_empty())
89. pop();
90. }
91. };

## Edge.h

1. #pragma once
2. #pragma once
3. #include <iostream>
4. using namespace std;
5. template <class name\_type, class weight\_type>
6. class Edge
7. {
8. name\_type from; // name of the vertex from
9. name\_type to; // name of the vertex to
10. weight\_type flow; // the flow that goes through the edge
11. weight\_type capacity; // max flow through the edge
12. public:
13. Edge()
14. {
15. from = ' ';
16. to = ' ';
17. flow = 0;
18. capacity = 0;
19. }
20. Edge(name\_type From, name\_type To, weight\_type Capacity)
21. {
22. from = From;
23. to = To;
24. capacity = Capacity;
25. flow = 0;
26. }
27. name\_type get\_vertex\_from() { return from; }
28. name\_type get\_vertex\_to() { return to; }
29. weight\_type get\_flow() { return flow; }
30. weight\_type get\_capacity() { return capacity; }
31. friend class Graph;
32. };

## Vertex.h

1. #pragma once
2. #pragma once
3. #include "Edge.h"
4. #include "Queue.h"
5. template <class name\_type, class weight\_type>
6. class Vertex
7. {
8. name\_type name; // name of the vertex
9. Vertex\* mark; // mark from where came here (empty when not set)
10. Queue <Edge<name\_type, weight\_type>>\* edges; // edges that starts from this vertex
11. public:
12. Vertex()
13. {
14. name = ' ';
15. mark = nullptr;
16. edges = new Queue <Edge<name\_type, weight\_type>>();
17. }
18. Vertex(name\_type Name)
19. {
20. name = Name;
21. mark = nullptr;
22. edges = new Queue <Edge<name\_type, weight\_type>>();
23. }
24. name\_type get\_name() { return name; }
25. Vertex\* get\_mark() { return mark; }
26. void add\_edge(Edge<char, double>\* e)
27. {
28. edges->push(e);
29. }
30. friend class Graph;
31. };

## Graph.h

1. #pragma once
2. #pragma once
3. #include "Vertex.h"
4. #include <fstream>
6. class Graph
7. {
8. Queue<Vertex<char, double>>\* V; // list of vertexes, every vertex has a list of edges starting from it
10. public:
11. Graph();
13. void clear\_marks(); // clear all marks that have vertexes from V
15. void clear\_flows(); // clear all flows in graph (make every flow = 0)
17. Element<Vertex<char, double>>\* search\_vertex\_on\_name(char); // returns vertex from V with data name
19. Element<Edge<char, double>>\* search\_edge(Vertex<char, double>\*, char); // searches edge on vertexes from and to
21. void read\_from\_file(string); // fill V and E from file with filename
23. void print(string); // prints nodes with flow/capacity to the file with filename
25. double edmonds\_karp(); // returns max flow through the graph from 'S' to 'T'
26. };

## Graph.cpp

1. #include "Graph.h"
3. char source = 'S', target = 'T';
5. Graph::Graph() { V = new Queue<Vertex<char, double>>(); }
7. void Graph::clear\_marks() // clear all marks that have vertexes from V
8. {
9. Element<Vertex<char, double>>\* v = V->get\_tail();
10. while (v != nullptr)
11. {
12. v->get\_data()->mark = nullptr;
13. v = v->get\_next();
14. }
15. }
17. void Graph::clear\_flows() // clear all flows in graph (make every flow = 0)
18. {
19. Element<Vertex<char, double>>\* v = V->get\_tail();
20. while (v != nullptr)
21. {
22. Element<Edge<char, double>>\* e = v->get\_data()->edges->get\_tail();
23. while (e != nullptr)
24. {
25. e->get\_data()->flow = 0;
26. e = e->get\_next();
27. }
28. v = v->get\_next();
29. }
30. }
32. Element<Vertex<char, double>>\* Graph::search\_vertex\_on\_name(char Name) // returns vertex from V with data name
33. {
34. if (!V->is\_empty())
35. {
36. Element<Vertex<char, double>>\* current(V->get\_tail());
37. while (current != nullptr)
38. {
39. Vertex<char, double>\* v = current->get\_data();
40. if (v->get\_name() == Name)
41. return current;
42. current = current->get\_next();
43. }
44. }
45. return nullptr; // returns null when vertex doesn't exist
46. }
48. Element<Edge<char, double>>\* Graph::search\_edge(Vertex<char, double>\* from, char to) // searches edge on vertexes
49. {
50. Element<Edge<char, double>>\* current = from->edges->get\_tail();
51. while (current != nullptr)
52. {
53. if (current->get\_data()->get\_vertex\_to() == to)
54. return current;
55. current = current->get\_next();
56. }
57. throw out\_of\_range("searching edge doesn't exist");
58. }
60. void Graph::read\_from\_file(string filename) // fill V and E from file
61. {
62. ifstream in(filename);
63. if (!in.is\_open())
64. throw out\_of\_range("data file couldn't be open");
65. while (in.peek() != EOF)
66. {
67. char v\_from, v\_to;
68. double max\_flow;
69. in >> v\_from >> v\_to >> max\_flow;
70. Edge<char, double>\* e = new Edge<char, double>(v\_from, v\_to, max\_flow);
71. Element<Vertex<char, double>>\* elem = search\_vertex\_on\_name(v\_from);
73. if (elem == nullptr)
74. {
75. Vertex<char, double>\* v = new Vertex<char, double>(v\_from);
76. v->add\_edge(e);
77. V->push\_front(v);
78. }
79. else
80. {
81. Vertex<char, double>\* v = elem->get\_data();
82. v->add\_edge(e);
83. }
84. elem = search\_vertex\_on\_name(v\_to);
85. if (elem == nullptr)
86. {
87. Vertex<char, double>\* v = new Vertex<char, double>(v\_to);
88. V->push\_front(v);
89. }
90. }
91. in.close();
92. }
94. void Graph::print(string filename) // prints nodes with flow/capacity to the file with filename
95. {
96. ofstream out(filename);
97. if (!out.is\_open())
98. throw out\_of\_range("output file couldn't be open");
100. Element<Vertex<char, double>>\* v = V->get\_tail();
101. while (v != nullptr)
102. {
103. Element<Edge<char, double>>\* e = v->get\_data()->edges->get\_tail();
104. while (e != nullptr)
105. {
106. out << e->get\_data()->get\_vertex\_from() << " " << e->get\_data()->get\_vertex\_to() << " " << e->get\_data()->get\_flow() << "/" << e->get\_data()->get\_capacity() << endl;
107. e = e->get\_next();
108. }
109. v = v->get\_next();
110. }
111. out.close();
112. }
114. double Graph::edmonds\_karp() // returns max flow through the graph from 'S' to 'T'
115. { // changes only flows on edges in the original graph, to clear flows use clear\_flows() function
116. double max\_flow = 0;
117. Queue<Vertex<char, double>>\* queue = new Queue<Vertex<char, double>>();
118. Element<Vertex<char, double>>\* S = search\_vertex\_on\_name(source); // source vertex
119. queue->push(S->get\_data()); // vertex is always the same and changing but elements containing this vertex are not similar
120. Element<Vertex<char, double>>\* current\_vertex = new Element<Vertex<char, double>>();
121. while (!queue->is\_empty()) // when empty path doesn't exist
122. {
123. current\_vertex = queue->pop();
124. Element<Edge<char, double>>\* current\_edge = current\_vertex->get\_data()->edges->get\_tail();
125. while (current\_edge != nullptr) // go through edges of current vertex
126. {
127. if (current\_edge->get\_data()->get\_capacity() - current\_edge->get\_data()->get\_flow() > 0)
128. { // if there is a possibility to push some flow through this edge
129. Element<Vertex<char, double>>\* adding\_vertex = search\_vertex\_on\_name(current\_edge->get\_data()->get\_vertex\_to()); // new to queue
130. if (adding\_vertex->get\_data()->get\_mark() == nullptr) // if it already has a mark we should save it
131. { // it has no mark yet
132. adding\_vertex->get\_data()->mark = current\_vertex->get\_data();
133. queue->push(adding\_vertex->get\_data());
134. if (adding\_vertex->get\_data()->get\_name() == target) // found min path
135. {
136. Queue<Edge<char, double>>\* path = new Queue<Edge<char, double>>();
137. double current\_flow = current\_edge->get\_data()->get\_capacity() - current\_edge->get\_data()->get\_flow();
139. while (current\_vertex->get\_data() != nullptr) // while the edge with source not mentioned
140. {
141. path->push(current\_edge->get\_data());
142. if (current\_flow > current\_edge->get\_data()->get\_capacity() - current\_edge->get\_data()->get\_flow())
143. current\_flow = current\_edge->get\_data()->get\_capacity() - current\_edge->get\_data()->get\_flow();
144. // current edge is the edge from current vertex to adding vertex
145. if (current\_vertex->get\_data()->get\_name() == source)
146. break;
147. adding\_vertex = current\_vertex;
148. current\_vertex = search\_vertex\_on\_name(current\_vertex->get\_data()->get\_mark()->get\_name());
149. current\_edge = search\_edge(current\_vertex->get\_data(), adding\_vertex->get\_data()->get\_name()); // go to previous edge in the path
150. }
151. max\_flow += current\_flow;
152. while (!path->is\_empty()) // change flow through chosen path
153. {
154. Edge<char, double>\* e = path->pop()->get\_data();
155. e->flow = e->flow + current\_flow;
156. }
157. clear\_marks(); // clear all marks, refresh graph
158. queue->~Queue(); // clear queue
159. queue->push(S->get\_data()); // go back to initial position
160. break;
161. }
162. } // it has mark and to save it we go to the next vertex because this path is obviously longer
163. }
164. current\_edge = current\_edge->get\_next();
165. }
167. }
168. clear\_marks();
169. return max\_flow;
170. }

## UnitTest\_EdmondsKarp.cpp

1. #include "pch.h"
2. #include "CppUnitTest.h"
3. #include "../EdmondsKarp/Graph.h"
5. using namespace Microsoft::VisualStudio::CppUnitTestFramework;
7. namespace UnitTestEdmondsKarp
8. {
9. TEST\_CLASS(UnitTestEdmondsKarp)
10. {
11. public:
13. TEST\_METHOD(TestMethodRead)
14. {
15. string input\_file = "in.txt";
16. string output\_file = "out.txt";
17. string input = "S O 3\nS P 3\nO Q 3\nO P 2\nP R 2\nQ R 4\nQ T 2\nR T 3";
18. string expecting\_output = "S P 0/3\nS O 0/3\nO P 0/2\nO Q 0/3\nP R 0/2\nQ T 0/2\nQ R 0/4\nR T 0/3\n";
19. ofstream out(input\_file);
20. out << input;
21. out.close();
22. Graph graph;
23. graph.read\_from\_file(input\_file);
24. graph.print(output\_file);
25. ifstream in(output\_file);
26. string output = "";
27. while (in.peek() != EOF)
28. {
29. char now = in.get();
30. output += now;
31. }
32. in.close();
33. Assert::AreEqual(output, expecting\_output);
34. }
36. TEST\_METHOD(TestMethodMaxFlow\_6\_vertexes)
37. {
38. string input\_file = "in.txt";
39. string output\_file = "out.txt";
40. string input = "S O 3\nS P 3\nO Q 3\nO P 2\nP R 2\nQ R 4\nQ T 2\nR T 3";
41. string expecting\_output = "S P 2/3\nS O 3/3\nO P 0/2\nO Q 3/3\nP R 2/2\nQ T 2/2\nQ R 1/4\nR T 3/3\n";
42. ofstream out(input\_file);
43. out << input;
44. out.close();
45. Graph graph;
46. graph.read\_from\_file(input\_file);
47. double max\_flow = graph.edmonds\_karp();
48. graph.print(output\_file);
49. ifstream in(output\_file);
50. string output = "";
51. while (in.peek() != EOF)
52. {
53. char now = in.get();
54. output += now;
55. }
56. in.close();
57. Assert::AreEqual(output, expecting\_output);
58. Assert::AreEqual(max\_flow, 5.0);
59. }
60. TEST\_METHOD(TestMethodMaxFlow\_2\_vertexes)
61. {
62. string input\_file = "in.txt";
63. string output\_file = "out.txt";
64. string input = "S T 2";
65. string expecting\_output = "S T 2/2\n";
66. ofstream out(input\_file);
67. out << input;
68. out.close();
69. Graph graph;
70. graph.read\_from\_file(input\_file);
71. double max\_flow = graph.edmonds\_karp();
72. graph.print(output\_file);
73. ifstream in(output\_file);
74. string output = "";
75. while (in.peek() != EOF)
76. {
77. char now = in.get();
78. output += now;
79. }
80. in.close();
81. Assert::AreEqual(output, expecting\_output);
82. Assert::AreEqual(max\_flow, 2.0);
83. }
85. TEST\_METHOD(TestMethodMaxFlow\_3\_vertexes)
86. {
87. string input\_file = "in.txt";
88. string output\_file = "out.txt";
89. string input = "S O 1\nO T 4";
90. string expecting\_output = "S O 1/1\nO T 1/4\n";
91. ofstream out(input\_file);
92. out << input;
93. out.close();
94. Graph graph;
95. graph.read\_from\_file(input\_file);
96. double max\_flow = graph.edmonds\_karp();
97. graph.print(output\_file);
98. ifstream in(output\_file);
99. string output = "";
100. while (in.peek() != EOF)
101. {
102. char now = in.get();
103. output += now;
104. }
105. in.close();
106. Assert::AreEqual(output, expecting\_output);
107. Assert::AreEqual(max\_flow, 1.0);
108. }
110. TEST\_METHOD(TestMethodMaxFlow\_triangle)
111. {
112. string input\_file = "in.txt";
113. string output\_file = "out.txt";
114. string input = "S O 2\nO T 1\nS T 3";
115. string expecting\_output = "S T 3/3\nS O 1/2\nO T 1/1\n";
116. ofstream out(input\_file);
117. out << input;
118. out.close();
119. Graph graph;
120. graph.read\_from\_file(input\_file);
121. double max\_flow = graph.edmonds\_karp();
122. graph.print(output\_file);
123. ifstream in(output\_file);
124. string output = "";
125. while (in.peek() != EOF)
126. {
127. char now = in.get();
128. output += now;
129. }
130. in.close();
131. Assert::AreEqual(output, expecting\_output);
132. Assert::AreEqual(max\_flow, 4.0);
133. }
135. TEST\_METHOD(TestMethodMaxFlow\_square)
136. {
137. string input\_file = "in.txt";
138. string output\_file = "out.txt";
139. string input = "S O 2\nO T 1\nS A 2\nA T 3";
140. string expecting\_output = "S A 2/2\nS O 1/2\nO T 1/1\nA T 2/3\n";
141. ofstream out(input\_file);
142. out << input;
143. out.close();
144. Graph graph;
145. graph.read\_from\_file(input\_file);
146. double max\_flow = graph.edmonds\_karp();
147. graph.print(output\_file);
148. ifstream in(output\_file);
149. string output = "";
150. while (in.peek() != EOF)
151. {
152. char now = in.get();
153. output += now;
154. }
155. in.close();
156. Assert::AreEqual(output, expecting\_output);
157. Assert::AreEqual(max\_flow, 3.0);
158. }
160. TEST\_METHOD(TestMethodMaxFlow\_4\_vertexes)
161. {
162. string input\_file = "in.txt";
163. string output\_file = "out.txt";
164. string input = "S O 2\nO A 3\nS A 1\nA T 4";
165. string expecting\_output = "S A 1/1\nS O 2/2\nO A 2/3\nA T 3/4\n";
166. ofstream out(input\_file);
167. out << input;
168. out.close();
169. Graph graph;
170. graph.read\_from\_file(input\_file);
171. double max\_flow = graph.edmonds\_karp();
172. graph.print(output\_file);
173. ifstream in(output\_file);
174. string output = "";
175. while (in.peek() != EOF)
176. {
177. char now = in.get();
178. output += now;
179. }
180. in.close();
181. Assert::AreEqual(output, expecting\_output);
182. Assert::AreEqual(max\_flow, 3.0);
183. }
185. TEST\_METHOD(TestMethodMaxFlow\_2\_triangles)
186. {
187. string input\_file = "in.txt";
188. string output\_file = "out.txt";
189. string input = "S O 4\nO T 2\nS A 1\nA T 3\nS T 3";
190. string expecting\_output = "S T 3/3\nS A 1/1\nS O 2/4\nO T 2/2\nA T 1/3\n";
191. ofstream out(input\_file);
192. out << input;
193. out.close();
194. Graph graph;
195. graph.read\_from\_file(input\_file);
196. double max\_flow = graph.edmonds\_karp();
197. graph.print(output\_file);
198. ifstream in(output\_file);
199. string output = "";
200. while (in.peek() != EOF)
201. {
202. char now = in.get();
203. output += now;
204. }
205. in.close();
206. Assert::AreEqual(output, expecting\_output);
207. Assert::AreEqual(max\_flow, 6.0);
208. }
210. TEST\_METHOD(TestMethodMaxFlow\_7\_vertexes)
211. {
212. string input\_file = "in.txt";
213. string output\_file = "out.txt";
214. string input = "S C 1\nS A 3\nS B 2\nC A 3\nB A 2\nA D 3\nA E 1\nD T 2\nE T 1";
215. string expecting\_output = "S B 0/2\nS A 3/3\nS C 0/1\nC A 0/3\nA E 1/1\nA D 2/3\nB A 0/2\nD T 2/2\nE T 1/1\n";
216. ofstream out(input\_file);
217. out << input;
218. out.close();
219. Graph graph;
220. graph.read\_from\_file(input\_file);
221. double max\_flow = graph.edmonds\_karp();
222. graph.print(output\_file);
223. ifstream in(output\_file);
224. string output = "";
225. while (in.peek() != EOF)
226. {
227. char now = in.get();
228. output += now;
229. }
230. in.close();
231. Assert::AreEqual(output, expecting\_output);
232. Assert::AreEqual(max\_flow, 3.0);
233. }
234. };
235. }