**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»**

**Вариант 2**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 8301 |  | Рожков В. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

Цель работы

Реализовать программу принимающую список ребер из файла, представляющий собой граф. Далее следует рассчитать максимальный поток в заданном графе методом Эдмондса-Карпа.

Описание реализуемых и вспомогательных классов

Класс Network, содержит поля: int VertexCount (количество верщин), int SourceVertex (исток), int SinkVertex (сток), int\*\* Capacity (пропускная способность), int\*\* FlowPassed (поток), int\* ParList (вершина, связанная с текущей), int\* CurrentPathC (поток, приходящий в текущую вершину), List<int>\* Graph (граф с пропускными способностями рёбер).

Класс содержит следующие методы:

* Конструктор – получает на вход файл, в котором список строк, обрабатывает их и в результате выдает список Map’ов, где каждый Map это одна вершина.
* Деструктор – вызывает метод clear (на основе обычного удаления двоичного дерева)
* BFS(int Source, int Sink) – функция поиска кратчайшего пути в графе, где остаточная пропускная способность ещё не равна нулю. Возвращает максимальный поток кратчайшего пути.
* max\_flow() – функция возвращает максимальный поток транспортной сети.
* EdmondsKarp(int Source, int Sink) – функция вычисляет максимальный поток методом Эдмондса-Карпа.

Оценка временной сложности алгоритмов

* BFS(int Source, int Sink) – O(E)
* max\_flow() – O(V\*E2)
* EdmondsKarp(int Source, int Sink) – O(V\*E2)

Описание реализованных unit-тестов

Реализованные мною тесты проверяют ситуации c 6 вершинами и с 20. Они проверяют ситуацию с одним ребром из стока в сток, а также когда есть не только ребро из истока в сток, но и другие рёбра. Тесты проверяют ситуации, когда отсутствует исток или сток, а также, когда введена отрицательная пропускная способность. Также они проверяют корректность обработки исключительных ситуаций, например, когда пользователь не ввел одну из позиций, либо ввёл её некорректно.

**Обоснование выбора используемых структур данных**

Мap я использую для того, чтобы индивидуализировать вершины индексами. Я использую данную структуру, т.к. она позволяет не сохранять повторяющиеся данные и даёт быстрый доступ к своим элементам. List я использую для хранения графа и пропускных способностей. В структуре List удобный функционал в отличие от обычного массива, нам не нужно хранить размер массива, а также мы можем быстро добавлять и удалять элементы, без затраты времени на их перезапись в новый массив. Структуру Queue я использую для перебора путей потоков. Эта структура удобна тем, что не требует данных о количестве элементов, а также об их индексах.

Примеры работы программы

|  |
| --- |
| Примеры |
|  |

Листинг

|  |
| --- |
| Network.h |
| 1. #pragma once 2. #include<fstream> 3. #include"Map.h" 4. #include"Queue.h" 5. #include<string> 6. #include"Algorithm.h" 7. class Network { 8. public: 9. ~Network() { 10. for (int i = 0; i < VertexCount; ++i) { 11. Graph[i].clear(); 12. delete[]Capacity[i]; 13. delete[]FlowPassed[i]; 14. } 15. delete[]ParList; 16. delete[]CurrentPathC; 17. } 18. Network(ifstream& file) { 19. Map<char, int>\* Map\_from\_char\_to\_number = new Map<char, int>(); 20. Validators(file, Map\_from\_char\_to\_number); 21. file.clear(); 22. file.seekg(ios::beg); 23. while (!file.eof()) { 24. string s1; 25. int vert1, vert2, cap; 26. getline(file, s1); 27. vert1 = Map\_from\_char\_to\_number->find(s1[0]); 28. vert2 = Map\_from\_char\_to\_number->find(s1[2]); 29. Capacity[vert1][vert2] = stoi(s1.substr(4)); 30. Graph[vert1].push\_back(vert2); 31. Graph[vert2].push\_back(vert1); 32. } 33. } 34. void Validators(ifstream& file, Map<char, int>\*& Map\_from\_char\_to\_number) { 35. int vert1, vert2, cap; 36. VertexCount = 0; 37. Map\_from\_char\_to\_number = new Map<char, int>(); 38. int str\_num = 1; 39. while (!file.eof()) { 40. string s1; 41. getline(file, s1); 42. if (s1.size() >= 5) { //greater than or equal to 5, because this is the minimum possible input(two letters, two spaces,one digit) 43. if (!((s1[0] >= 'A' && s1[0] <= 'Z') && (s1[1] == ' '))) { 44. throw std::exception(string(("Error entering the first character in the string or missing a space after it. Check the correctness of the input in the file and correct these errors in the line under the number: " + to\_string(str\_num))).c\_str()); 45. } 46. if (!((s1[2] >= 'A' && s1[2] <= 'Z') && (s1[3] == ' '))) { 47. throw std::exception(string(("Error entering the second character in the string or missing a space after it. Check the correctness of the input in the file and correct these errors in the line under the number: " + to\_string(str\_num))).c\_str()); 48. } 49. string cur; 50. for (int i = 4; i < s1.size(); ++i) { 51. if (s1[i] >= '0' && s1[i] <= '9') 52. cur += s1[i]; 53. else { 54. throw std::exception(string(("Error entering the third character (bandwidth) in the string or the presence of a space after it.Please note that the bandwidth cannot be negative. Check that you entered the file correctly and correct these errors in the line number: " + to\_string(str\_num))).c\_str()); 55. } 56. } 57. if (!Map\_from\_char\_to\_number->find\_is(s1[0])) { 58. Map\_from\_char\_to\_number->insert(s1[0], VertexCount); 59. ++VertexCount; 60. } 61. if (!Map\_from\_char\_to\_number->find\_is(s1[2])) { 62. Map\_from\_char\_to\_number->insert(s1[2], VertexCount); 63. ++VertexCount; 64. } 65. } 66. else 67. { 68. throw std::exception(string(("A data-entry error. Check the correctness of the input in the file and correct these errors in the line under the number: " + to\_string(str\_num))).c\_str()); 69. } 70. ++str\_num; 71. } 72. if (Map\_from\_char\_to\_number->find\_is('S')) 73. SourceVertex = Map\_from\_char\_to\_number->find('S'); 74. else { 75. throw std::exception("Source is missing"); 76. } 77. if (Map\_from\_char\_to\_number->find\_is('T')) 78. SinkVertex = Map\_from\_char\_to\_number->find('T'); 79. else { 80. throw std::exception("Sink is missing"); 81. } 82. Graph = new List<int>[VertexCount]; 83. FlowPassed = new int\* [VertexCount]; 84. Capacity = new int\* [VertexCount]; 85. ParList = new int[VertexCount]; 86. CurrentPathC = new int[VertexCount]; 87. for (int i = 0; i < VertexCount; ++i) { 88. FlowPassed[i] = new int[VertexCount]; 89. Capacity[i] = new int[VertexCount]; 90. for (int j = 0; j < VertexCount; ++j) { 91. FlowPassed[i][j] = 0; 92. Capacity[i][j] = 0; 93. } 94. } 95. } 97. int BFS(int Source, int Sink) //breadth first search 98. { 99. for (int i = 0; i < VertexCount; ++i) { 100. ParList[i] = -1; 101. CurrentPathC[i] = 0; 102. } 103. Queue<int> q; //declare queue vector 104. q.enqueue(Source); 105. ParList[Source] = -1; 106. CurrentPathC[Source] = 999; 107. while (!q.empty()) //if q is not empty 108. { 109. int CurrNode = q.dequeue(); 110. for (int i = 0; i < Graph[CurrNode].get\_size(); i++) 111. { 112. int to = Graph[CurrNode].at(i); 113. if (ParList[to] == -1) 114. { 115. if (Capacity[CurrNode][to] - FlowPassed[CurrNode][to] > 0) 116. { 117. ParList[to] = CurrNode; 118. CurrentPathC[to] = min(CurrentPathC[CurrNode], Capacity[CurrNode][to] - FlowPassed[CurrNode][to]); 119. if (to == Sink) 120. { 121. return CurrentPathC[Sink]; 122. } 123. q.enqueue(to); 124. } 125. } 126. } 127. } 128. return 0; 129. } 130. int max\_flow() { 131. int max = EdmondsKarp(SourceVertex, SinkVertex); 132. return max; 133. } 134. int EdmondsKarp(int Source, int Sink) 135. { 136. int MaxFlow = 0; 137. while (true) 138. { 139. int Flow = BFS(Source, Sink); 140. if (Flow == 0) 141. { 142. break; 143. } 144. MaxFlow += Flow; 145. int CurrNode = Sink; 146. while (CurrNode != Source) 147. { 148. int PrevNode = ParList[CurrNode]; 149. FlowPassed[PrevNode][CurrNode] += Flow; 150. FlowPassed[CurrNode][PrevNode] -= Flow; 151. CurrNode = PrevNode; 152. } 153. } 154. return MaxFlow; 155. } 156. private: 157. int SourceVertex; 158. int SinkVertex; 159. List<int>\* Graph; 160. int\*\* Capacity; 161. int\*\* FlowPassed; 162. int\* ParList; 163. int\* CurrentPathC; 164. int VertexCount; 165. }; |

|  |
| --- |
| UnitTestNetwork.cpp |
| 1. #include "pch.h" 2. #include "CppUnitTest.h" 3. #include"../Network.h" 4. using namespace Microsoft::VisualStudio::CppUnitTestFramework; 5. namespace UnitTestNetwork 6. { 7. TEST\_CLASS(UnitTestNetwork) 8. { 9. public: 11. TEST\_METHOD(TestMethod\_Ñorrect\_output\_for\_6\_vertexes) 12. { 13. ifstream input("D:\\Учеба\\АиСд\\Edmonds\_Karp\\UnitTestNetwork\\Input1.txt"); 14. Network Flows(input); 15. Assert::AreEqual(Flows.max\_flow(), 5); 16. } 17. TEST\_METHOD(TestMethod\_Exception\_entering\_the\_first\_character) { 18. try { 19. ifstream input("D:\\Учеба\\АиСд\\Edmonds\_Karp\\UnitTestNetwork\\Input2.txt"); 20. Network Flows(input); 21. } 22. catch (exception& ex) { 23. Assert::AreEqual(ex.what(), "Error entering the first character in the string or missing a space after it. Check the correctness of the input in the file and correct these errors in the line under the number: 2"); 24. } 25. } 26. TEST\_METHOD(TestMethod\_Exception\_entering\_the\_second\_character) { 27. try { 28. ifstream input("D:\\Учеба\\АиСд\\Edmonds\_Karp\\UnitTestNetwork\\Input3.txt"); 29. Network Flows(input); 30. } 31. catch (exception& ex) { 32. Assert::AreEqual(ex.what(), "Error entering the second character in the string or missing a space after it. Check the correctness of the input in the file and correct these errors in the line under the number: 2"); 33. } 34. } 35. TEST\_METHOD(TestMethod\_Exception\_entering\_the\_third\_number\_Flows) { 36. try { 37. ifstream input("D:\\Учеба\\АиСд\\Edmonds\_Karp\\UnitTestNetwork\\Input4.txt"); 38. Network Flows(input); 39. } 40. catch (exception& ex) { 41. Assert::AreEqual(ex.what(), "Error entering the third character (bandwidth) in the string or the presence of a space after it.Please note that the bandwidth cannot be negative. Check that you entered the file correctly and correct these errors in the line number: 2"); 42. } 43. } 44. TEST\_METHOD(TestMethod\_Exception\_empty\_string) { 45. try { 46. ifstream input("D:\\Учеба\\АиСд\\Edmonds\_Karp\\UnitTestNetwork\\Input5.txt"); 47. Network Flows(input); 48. } 49. catch (exception& ex) { 50. Assert::AreEqual(ex.what(), "A data-entry error. Check the correctness of the input in the file and correct these errors in the line under the number: 2"); 51. } 52. } 53. TEST\_METHOD(TestMethod\_Ñorrect\_output\_for\_6\_vertexes\_and\_edge\_from\_source\_to\_sink) 54. { 55. ifstream input("D:\\Учеба\\АиСд\\Edmonds\_Karp\\UnitTestNetwork\\Input6.txt"); 56. Network Flows(input); 57. Assert::AreEqual(Flows.max\_flow(), 25); 58. } 59. TEST\_METHOD(TestMethod\_Ñorrect\_output\_for\_2\_vertexes\_edges\_from\_source\_to\_sink) 60. { 61. ifstream input("D:\\Учеба\\АиСд\\Edmonds\_Karp\\UnitTestNetwork\\Input7.txt"); 62. Network Flows(input); 63. Assert::AreEqual(Flows.max\_flow(), 20); 64. } 65. TEST\_METHOD(TestMethod\_Exception\_there\_is\_a\_path\_from\_the\_vertex\_to\_itself) { 66. try 67. { 68. ifstream input("D:\\Учеба\\АиСд\\Edmonds\_Karp\\UnitTestNetwork\\Input8.txt"); 69. Network Flows(input); 70. } 71. catch (exception& ex) { 72. Assert::AreEqual(ex.what(), "The path from the vertex to itself is impossible in the string under the number: 2"); 73. } 74. } 75. TEST\_METHOD(TestMethod\_Сorrect\_output\_for\_20\_vertexes) 76. { 77. ifstream input("D:\\Учеба\\АиСд\\Edmonds\_Karp\\UnitTestNetwork\\Input9.txt"); 78. Network Flows(input); 79. Assert::AreEqual(Flows.max\_flow(), 19); 80. } 81. TEST\_METHOD(TestMethod\_Exception\_missing\_is\_source) { 82. try 83. { 84. ifstream input("D:\\Учеба\\АиСд\\Edmonds\_Karp\\UnitTestNetwork\\Input10.txt"); 85. Network Flows(input); 86. } 87. catch (exception& ex) { 88. Assert::AreEqual(ex.what(), "Source is missing"); 89. } 90. } 91. TEST\_METHOD(TestMethod\_Exception\_missing\_is\_sink) { 92. try 93. { 94. ifstream input("D:\\Учеба\\АиСд\\Edmonds\_Karp\\UnitTestNetwork\\Input11.txt"); 95. Network Flows(input); 96. } 97. catch (exception& ex) { 98. Assert::AreEqual(ex.what(), "Sink is missing"); 99. } 100. } 101. }; 102. } |

Вывод

В данной лабораторной работе я познакомилась с алгоритмом поиска максимального потока в транспортной сети методом Эдмондса-Карпа и смогла применить его в нахождении максимального потока в данной сети, а также закрепила свои навыки в объектно-ориентированном программировании на языке C++.