**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по лабораторной работе №2**

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

|  |  |  |
| --- | --- | --- |
| Студент гр. 1302 |  | Новиков Г.В. |
| Преподаватель |  | Родионова Е.А. |

Санкт-Петербург

2023

Постановка задачи

**Алгоритм Дейкстры**

Реализовать алгоритм Дейстры с помощью выбранной базовой структуры данных, двоичной кучи и Фибоначчиевой кучи (кучи реализовать самостоятельно). Сравнить временные затраты на выполнение данных реализаций алгоритма для графов низкой/средней/высокой связности. Отчет должен содержать графики и выводы по проделанной работе.

Теория

**Алгоритм Дейкстры**

Основная идея алгоритма. На каждом шаге отыскивается вершина, кратчайшее расстояние до которой уже найдено. Например, на первом шаге это будет вершина — конец наименьшего ребра, инцидентного источнику. Пометки остальных вершин модифицируются через эту вершину, после чего ее можно не рассматривать. Алгоритм 2.18 (алгоритм Дейкстры). Обозначим W — множество вершин, кратчайшее расстояние до которых уже найдено, и назовем его множеством обработанных вершин.

*Инициализация:*

ЦИКЛ ПО i от 1 до n

D(i) := P(s; i);

G(i) := s

КЦ

*Основной алгоритм:*

W := {s}

ЦИКЛ-ПОКА V \ W 6= ∅ (есть необработанные вершины)

найти вершину w ∈ V \ W с наименьшей пометкой D(w)

перенести найденную вершину в множество обработанных:

W := W ∪ {w}

модифицировать пометки необработанных вершин через w:

ЦИКЛ ПО всем вершинам u ∈ V \ W, смежным с w

D(u) := min{D(u); D(w) + P(w; u)}

КЦ

КЦ

**Двоичная куча**

Двоичная куча — такое двоичное дерево, для которого выполнены три условия:

1. Значение в любой вершине не больше, чем значения её потомков.
2. Глубина всех листьев (расстояние до корня) различается не более чем на 1 слой.
3. Последний слой заполняется слева направо без «дырок».

Удобная структура данных для сортирующего дерева — массив *A*, у которого первый элемент, *A*[1] — элемент в корне, а потомками элемента *A*[*i*] являются *A*[2*i*] и *A*[2*i*+1] (при нумерации элементов с первого). При нумерации элементов с нулевого, корневой элемент — *A*[0], а потомки элемента *A*[*i*] — *A*[2*i*+1] и *A*[2*i*+2]. При таком способе хранения условия 2 и 3 выполнены автоматически.

Высота кучи определяется как высота двоичного дерева. То есть она равна количеству рёбер в самом длинном простом пути, соединяющем корень кучи с одним из её листьев. Высота кучи есть Θ(log⁡�)ϴ(logN), где *N* — количество узлов дерева.

**Фибоначчиева куча**

Куча Фибоначчи — набор из подвешенных деревьев удовлетворяющих свойству: каждый предок не больше своих детей (если дерево на минимум). Это означает, что минимум всей кучи это один из корней этих деревьев. Одно из главных преимуществ Фибоначчиевой кучи — гибкость её структуры из-за того, что на деревья не наложены никакие ограничения по форме. Например, Фибоначчиева куча может состоять хоть из деревьев в каждом из которых по одному элементу. Такая гибкость позволяет выполнять некоторые операции лениво, оставляя работу более поздним операциям.

Оценка сложности

n — количество вершин в графе

m — количество рёбер в графе

**Алгоритм Дейкстры:**

С бинарной кучей – O(mlogn)

С кучей Фибоначчи – O(nlogn + m)

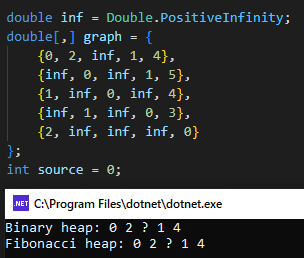
**Двоичная куча:**

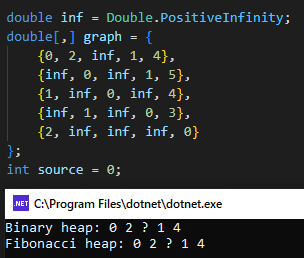
|  |  |
| --- | --- |
| **Метод** | **Сложность** |
| Search | O(n) |
| Sift\_up, Sift\_down | O(logn) |
| ExtractMin | O(logn) |
| Insert | O(logn) |
| Delete | O(logn) |

**Фибоначчиева куча:**

|  |  |
| --- | --- |
| **Метод** | **Сложность** |
| Search | O(n) |
| ExtractMin | O(max\_degree) = O(logn) |
| Insert | O(1) |
| DecreaseKey | O(1) |

Пример работы

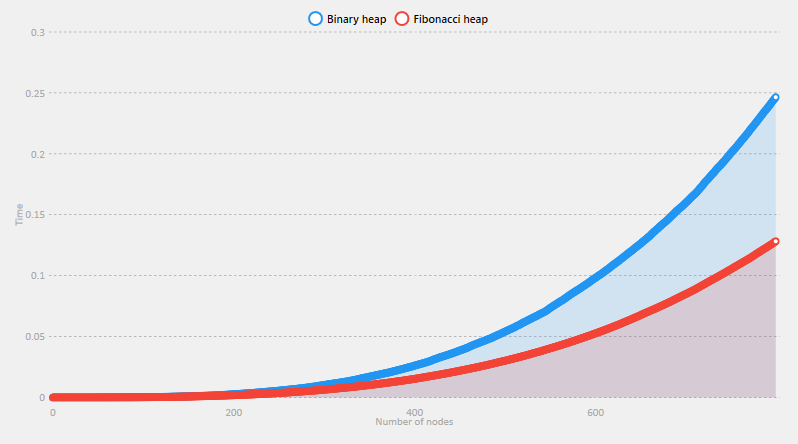




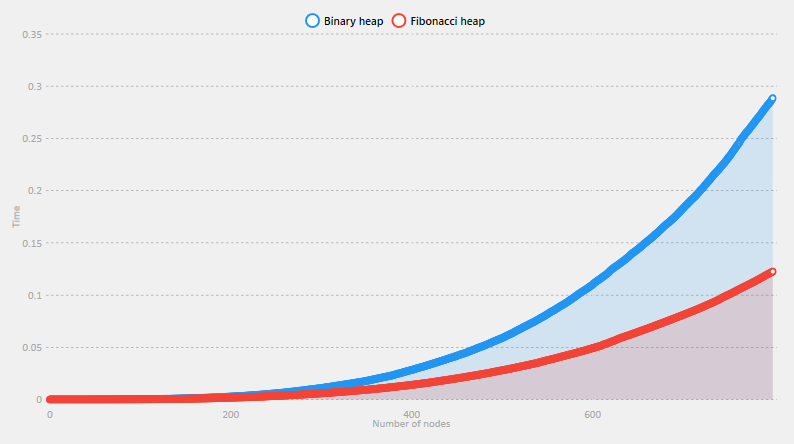
Графики

Максимальное количество узлов в графе для всех тестов - 800

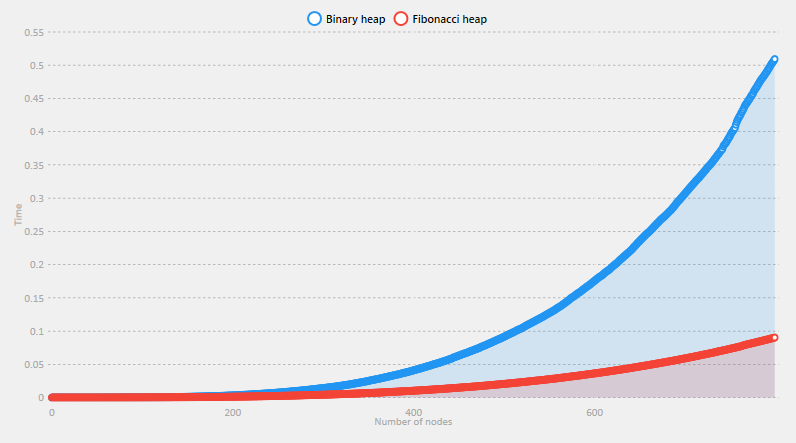
*Алгоритм Дейкстры на графе слабой связности:*



*Алгоритм Дейкстры на графе средней связности:*



*Алгоритм Дейкстры на графе сильной связности:*

**

**Вывод по графикам:** как мы видим, во всех случаях алгоритм на куче Фибоначчи работает быстрее. Скорость алгоритма на бинарной куче сильно зависит от связности графа, в то время как скорость алгоритма на куче Фибоначчи примерно одинакова для графов разной связности.

Листинг

|  |  |  |
| --- | --- | --- |
| **Program.cs:**  namespace lab2;  class Program {  static void Main(string[] args)  {  // adjacency matrix  double inf = Double.PositiveInfinity;  double[,] graph = {  {0, 2, inf, 1, 4},  {inf, 0, inf, 1, 5},  {1, inf, 0, inf, 4},  {inf, 1, inf, 0, 3},  {2, inf, inf, inf, 0}  };  int source = 3;  Dijkstra algorithm = new Dijkstra();  algorithm.SetMatrix(graph);  double[] result = algorithm.Run(source, HeapType.Binary);  Console.Write("Binary heap: ");  for (int i = 0; i < result.Length; i++) {  Console.Write(Convert.ToString(result[i]) + ' ');  }  Console.WriteLine();  result = algorithm.Run(source, HeapType.Fibonacci);  Console.Write("Fibonacci heap: ");  for (int i = 0; i < result.Length; i++) {  Console.Write(Convert.ToString(result[i]) + ' ');  }  Console.WriteLine();  PlotDataCreator dataCreator = new PlotDataCreator("data.txt");  Console.WriteLine("Running...");  dataCreator.CreatePlotData(90, 800);  Console.WriteLine("Success!");  Console.ReadLine();  }  } | **BinHeap.cs:**  namespace lab2;  class BinHeap {  private List<Node> elements = new List<Node>();  public List<Node> Elements {  set { elements = value; }  get { return elements; }  }  public int Length {  get { return elements.Count; }  }  public BinHeap(Node root) {  Insert(root);  }  public void Print() {  elements.ForEach(el => Console.Write(Convert.ToString(el.Value) + ' '));  Console.WriteLine();  }  public int Search(int id) {  for (int i = 0; i < Length; i++) {  if (Elements[i].Id == id) return i;  }  return -1;  }  public int GetParentIndex(int el\_index) {  if (el\_index == 0) return -1;  return (el\_index - 1) / 2;  }  public int[] GetChildrenIndices(int el\_index) {  int[] indices = new int[2] {el\_index \* 2 + 1, el\_index \* 2 + 2};  if (indices[0] >= Length) {  indices[0] = -1;  indices[1] = -1;  } else if (indices[1] >= Length) {  indices[1] = -1;  };  return indices;  }  public void Swap(int first, int second) {  Node tmp = elements[first];  elements[first] = elements[second];  elements[second] = tmp;  }  public void Sift\_up(int index) {  while (Length > index && index > 0) {  int parent\_index = GetParentIndex(index);  if (elements[index].Value < elements[parent\_index].Value) {  Swap(index, parent\_index);  } else {  return;  }  index = parent\_index;  }  }  public void Sift\_down(int index) {  int[] children\_indices = GetChildrenIndices(index);  while (children\_indices[0] != -1) {  int max\_child\_index = children\_indices[0];  if (children\_indices[1] != -1 && elements[children\_indices[0]].Value < elements[children\_indices[1]].Value) {  max\_child\_index = children\_indices[1];  }  if (elements[index].Value > elements[max\_child\_index].Value) {  Swap(index, max\_child\_index);  } else {  return;  }  index = max\_child\_index;  children\_indices = GetChildrenIndices(index);  }  }  public Node? ExtractMin() {  if (Length == 0) return null;  Node min = Elements[0];  Delete(0);  return min;  }  public void Insert(Node node) {  elements.Add(node);  Sift\_up(Length - 1);  }  public void Delete(int index) {  Swap(index, Length - 1);  elements.RemoveAt(Length - 1);  Sift\_down(index);  Sift\_up(index);  }  } | **FibonacciHeap.cs:**  namespace lab2;  class FibonacciHeap {  private LinkedList roots = new LinkedList();  private int length = 0;  private FibNode? min;  private int max\_degree = 0;  public LinkedList Roots {  get { return roots; }  }  public FibNode? Min {  get { return min; }  set { min = value; }  }  public int MaxDegree {  get { return max\_degree; }  }  public int Length {  get { return length; }  }  public FibonacciHeap(FibNode root) {  Insert(root);  Min = root;  }  private List<FibNode> GetNodes(FibNode? root) {  List<FibNode>? nodes = new List<FibNode>();  if (root is null) return nodes;  nodes.Add(root);  FibNode? child = root.Children.First;  while (child is not null) {  nodes.AddRange(GetNodes(child) ?? new List<FibNode>());  child = child.Next;  }  return nodes;  }  public List<FibNode> GetAllNodes() {  List<FibNode>? nodes = new List<FibNode>();  FibNode? root = Roots.First;  while (root is not null) {  nodes.AddRange(GetNodes(root) ?? new List<FibNode>());  root = root.Next;  }  return nodes;  }  public void Print() {  FibNode? root = Roots.First;  Console.Write("HEAP ROOTS: ");  while (root is not null) {  Console.Write($"({root.Id}, {root.Value}) ");  root = root.Next;  }  List<FibNode> nodes = GetAllNodes();  foreach (FibNode node in nodes) {  Console.Write($"\nnode ({node.Id}, {node.Value}): ");  FibNode? child = node.Children.First;  while (child is not null) {  Console.Write($"({child.Id}, {child.Value}) ");  child = child.Next;  }  }  Console.WriteLine("\n");  }  public void Insert(FibNode node) {  roots.AddLast(node);  if (Min is null || node.Value < Min.Value) {  Min = node;  }  if (node.Degree > MaxDegree) max\_degree = node.Degree;  length++;  }  public FibNode? ExtractMin() {  FibNode? min = Min;  if (min is null) return null;  FibNode? child = min.Children.First;  FibNode? next\_child;  while (child is not null) {  next\_child = child.Next;  min.Children.Remove(child);  Roots.AddLast(child);  if (child.Degree > MaxDegree) max\_degree = child.Degree;  child = next\_child;  }  Roots.Remove(min);  Min = null;  length--;  Compress();  return min;  }  public FibNode Merge(FibNode node1, FibNode node2) {  if (node2.Value < node1.Value) {  FibNode tmp = node1;  node1 = node2;  node2 = tmp;  }  Roots.Remove(node2);  node1.AddChild(node2);  if (node1.Degree > MaxDegree) {  max\_degree = node1.Degree;  }  return node1;  }  public void Compress() {  if (Roots.Count == 0) {  Min = null;  return;  }  FibNode?[] roots\_by\_degrees = new FibNode[MaxDegree + Convert.ToInt32(Math.Log(Roots.Count, 2)) + 1];  FibNode? node = Roots.First;  FibNode? next;  while (node is not null) {  next = node.Next;  while (roots\_by\_degrees[node.Degree] is not null) {  node = Merge(roots\_by\_degrees[node.Degree], node);  roots\_by\_degrees[node.Degree - 1] = null;  }  roots\_by\_degrees[node.Degree] = node;  node = next;  }  max\_degree = 0;  node = Roots.First;  while (node is not null) {  if (node.Degree > max\_degree) max\_degree = node.Degree;  if (Min is null || node.Value < Min.Value) Min = node;  node = node.Next;  }  }  public FibNode? Search(int id, LinkedList roots) {  FibNode? root = roots.First;  while (root is not null) {  if (root.Id == id) return root;  FibNode? node = Search(id, root.Children);  if (node is not null) return node;  root = root.Next;  }  return null;  }  public void DecreaseKey(FibNode node, double value) {  if (node is null) return;  if (node.Value < value) throw new ArgumentException("value is bigger than node.Value");  node.Value = value;  FibNode? parent = node.Parent;  if (parent is not null && value < parent.Value) {  CutOut(node);  }  if (value < Min.Value) Min = node;  }  private void CutOut(FibNode node) {  FibNode? parent = node.Parent;  if (parent is not null) {  parent.Children.Remove(node);  Roots.AddLast(node);  if (node.Degree > max\_degree) max\_degree = node.Degree;  node.Parent = null;  node.LostChild = false;  if (!parent.LostChild) {  parent.LostChild = true;  }  else CutOut(parent);  }  }  } |
| **Dijkstra.cs:**  namespace lab2;  enum HeapType { Binary, Fibonacci }  class Dijkstra {  private double[,]? graph;  public void SetMatrix(double[,]? graph) {  if (graph is not null && graph.GetLength(0) != graph.GetLength(1)) throw new InvalidDataException("Graph adjacency matrix is not square");  if (graph is not null && graph.Length == 0) graph = null;  this.graph = graph;  }  public double[] Run(int start\_index, HeapType heap\_type) {  if (graph is null) throw new InvalidDataException("Graph is not initialized. Run SetMatrix first");  double[] distances;  if (heap\_type == HeapType.Binary) {  distances = RunWithBinHeap(start\_index);  }  else {  distances = RunWithFibonacciHeap(start\_index);  }  return distances;  }  private double[] RunWithBinHeap(int start\_index) {  Node start = new Node(start\_index, 0);  BinHeap heap = new BinHeap(start);  double[] distances = new double[graph.GetLength(0)];  for (int i = 0; i < distances.Length; i++) {  distances[i] = Double.PositiveInfinity;  }  distances[start\_index] = 0;  Node? current = heap.ExtractMin();  while (current is not null) {  for (int i = 0; i < graph.GetLength(1); i++) {  if (distances[current.Id] + graph[current.Id,i] < distances[i]) {  distances[i] = distances[current.Id] + graph[current.Id,i];  int node\_id = heap.Search(i);  if (node\_id == -1) heap.Insert(new Node(i, distances[i]));  else heap.Elements[node\_id].Value = distances[i];  }  }  current = heap.ExtractMin();  }  return distances;  }  private double[] RunWithFibonacciHeap(int start\_index) {  FibNode start = new FibNode(start\_index, 0);  FibonacciHeap heap = new FibonacciHeap(start);  double[] distances = new double[graph.GetLength(0)];  for (int i = 0; i < distances.Length; i++) {  distances[i] = Double.PositiveInfinity;  }  distances[start\_index] = 0;  FibNode? current = heap.ExtractMin();  while (current is not null) {  for (int i = 0; i < graph.GetLength(1); i++) {  if (distances[current.Id] + graph[current.Id,i] < distances[i]) {  distances[i] = distances[current.Id] + graph[current.Id,i];  FibNode? node = heap.Search(i, heap.Roots);  if (node is null) heap.Insert(new FibNode(i, distances[i]));  else heap.DecreaseKey(node, distances[i]);  }  }  // Console.WriteLine($"({current.Id}, {current.Value})");  current = heap.ExtractMin();  }  return distances;  }  } | **LinkedList.cs:**  namespace lab2;  class LinkedList {  private FibNode? first;  private FibNode? last;  private int count = 0;  public FibNode? First {  get { return first; }  set { first = value; }  }  public FibNode? Last {  get { return last; }  set { last = value; }  }  public int Count {  get { return count; }  set { count = value; }  }  public void AddLast(FibNode node) {  if (Object.ReferenceEquals(node.List, this)) throw new InvalidOperationException("Node is already in this list");  if (First is null) {  First = node;  Last = node;  }  else {  Last.Next = node;  node.Previous = Last;  Last = node;  }  node.List = this;  Count++;  }  public void Remove(FibNode node) {  if (!Object.ReferenceEquals(node.List, this)) throw new InvalidOperationException("Node does not belong this list");  if (node.Previous is not null && node.Next is not null) {  node.Next.Previous = node.Previous;  node.Previous.Next = node.Next;  }  else if (node.Next is not null) {  First = node.Next;  node.Next.Previous = null;  }  else if (node.Previous is not null) {  Last = node.Previous;  node.Previous.Next = null;  }  else {  First = null;  Last = null;  }  node.Parent = null;  node.Next = null;  node.Previous = null;  node.List = null;  Count--;  }  } | **Node.cs:**  namespace lab2;  class Node {  private int id;  private double value = Double.PositiveInfinity;  public int Id {  set { id = value; }  get { return id; }  }  public double Value {  set { this.value = value; }  get { return value; }  }  public Node(int id, double value) {  this.id = id;  this.Value = value;  }  }  class FibNode : Node {  private LinkedList children = new LinkedList();  private bool lost\_child = false;  private FibNode? parent;  private FibNode? next;  private FibNode? previous;  private LinkedList? list;  public LinkedList Children {  get { return children; }  }  public LinkedList? List {  get { return list; }  set { list = value; }  }  public bool LostChild {  get { return lost\_child; }  set { lost\_child = value; }  }  public int Degree {  get { return Children.Count; }  }  public FibNode? Parent {  get { return parent; }  set { parent = value; }  }  public FibNode? Next {  get { return next; }  set { next = value; }  }  public FibNode? Previous {  get { return previous; }  set { previous = value; }  }  public void AddChild(FibNode child) {  child.Parent = this;  Children.AddLast(child);  }  public FibNode(int id, double value) : base(id, value) {}  } |

Ссылка на код

[GitHub](https://github.com/Gregory-hub/leti/tree/master/Algorithms%20and%20Data%20Structures/sem2/lab2/lab2)