Всестороннее изучение структур данных с использованием C# 2.0

Часть V: От деревьев к графам

Граф, также как и дерево, представляет собой набор узлов и ребер, но не имеет правил, которые определяют связи между узлами. В пятой статье этой серии статей мы изучим все о графах, одной из наиболее многогранных структур данных.

1 Введение

В первой и второй частях нашей серии статей мы рассмотрели линейные структуры данных — массив, Список, Очередь, Стек, Хеш-таблицу и Словарь. В третьей части мы начали исследование деревьев. Вспомните, что деревья состоят из множества узлов (*nodes*), где все узлы соединяются с другими узлами. Эти соединения называются ребрами (*edges*). Как мы уже обсуждали, существует много правил говорящих о том, как эти соединения могут производиться. Например, все узлы в дереве за исключением одного—корня—должны иметь в точности один родительский (*parent*) узел, в то время как все узлы могут иметь произвольное число потомков. Эти простые правила обеспечивают выполнение следующих трех предпосылок:

1. Начиная с любого узла можно достичь любой другой узел в дереве.
2. В дереве нет циклов (*cycles*). Цикл существует если, например, начиная с некоторого узла v, существует путь через узлы v_1, v_2, ..., v_k который затем возвращается к v.
3. Число ребер в дереве в точности на единицу меньше, чем число узлов.

В третьей части мы сфокусировали наше внимание на бинарных деревьях (*binary trees*), которые являются частным случаем деревьев. Бинарные деревья это деревья, чьи узлы имеют не более двух потомков.

И в пятой части нашей серии статей мы собираемся рассмотреть графы (*graphs*). Графы состоят из множества узлов и ребер, так же как и деревья, но в графах нет специфических правил для связей между узлами. В графах нет понятия корневого узла, также нет понятий предков и потомков. Граф просто представляет собой набор взаимосвязанных узлов.

Comment [T1]: Необходимо понимать, что все деревья являются графами. Дерево это специальный случай графа, в котором все узлы достижимы из начального узла и в котором нет циклов.

На Рисунке 1 показано три примера графов. Отметьте, что графы, в отличие от деревьев, могут иметь множества узлов, которые не связаны с другими множествами узлов. Например, граф (a) имеет два отдельных несвязанных множества узлов. Графы также могут иметь циклы. Граф (b) имеет несколько циклов. Один из них это путь от узла v_1 к v_2 к v_4 и обратно к v_1. Другой цикл идет от узла v_1 к v_2 к v_3 к v_5 к v_4 и обратно к v_1. (Циклы также есть в графе (a).) Граф (c) не содержит никаких циклов, и поэтому является деревом.

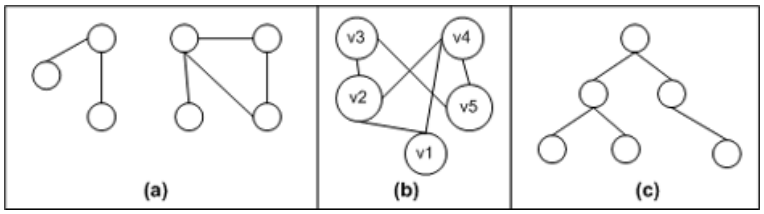

Рисунок 1. Три примера графов

Многие из проблем окружающего нас мира могут быть смоделированы с использованием графов. Например, поисковые машины моделируют Интернет в виде графа, в котором веб-страницы представляют собой узлы графа, а связи между веб-страницами его ребра. Такие программные продукты как Microsoft MapPoint которые могут строить маршруты движения от одного города к другому используют графы, моделируя города в качестве узлов графа и дороги, соединяющие города, как ребра графа.

2 Изучение различных классов ребер

Графы, в их простейшем определении, представляют собой набор узлов и ребер, однако существует несколько разновидностей ребер:

1. Направленные и ненаправленные ребра;
2. Взвешенные и невзвешенные ребра.

При обсуждении использования графов для моделирования проблемы очень важно явно указать о том, какой граф используется. То ли это граф, чьи ребра направлены и взвешены, то ли это граф, чьи ребра ненаправлены и взвешены? В двух следующих разделах мы обсудим различия между направленными и ненаправленными ребрами, а также взвешенными и невзвешенными ребрами.

2.1 Направленные и ненаправленные ребра

Ребра графа обеспечивают связи между узлами. По умолчанию ребро рассматривается как двунаправленное. То есть, если существует ребро между узлами *v* и *u*, то предполагается, что можно пройти как от узла *v* к узлу *u* так и от узла *u* к узлу *v*. Графы с двунаправленными ребрами называются ненаправленными графами (*undirected graphs*), так как они не имеют явного направления ребер.

Однако, для некоторых проблем ребро должно предполагать только одно направление соединения от одного узла к другому. Например, моделируя Интернет в виде графа, ссылка от веб-сраницы *v* к веб-странице *u* будет предполагать, что ребро между узлами *v* и *u* будет однонаправленным. То есть, имеется возможность пройти от узла *v* к узлу *u*, но не от узла *u* к узлу *v*. Графы, которые имеют однонаправленные ребра называются направленными графами (*directed graphs*).

При рисовании графа двунаправленные ребра изображаются как прямые линии, так как это показано на Рисунке 1. Однонаправленные ребра изображаются в виде стрелок, показывающих направление ребер. Рисунок 2 показывает направленный граф, в котором узлами являются веб-страницы определенного веб-сайта и направленное ребро от узла *u* к узлу *v* указывает, что существует ссылка от веб-страницы *u* к веб-странице *v*. Отметьте,

что в том случае когда существуют обе ссылки, как от *u* к *v* так и от *v* к *u*, то используются две стрелки—одна от узла *v* к узлу *u* и другая от узла *u* к узлу *v*.

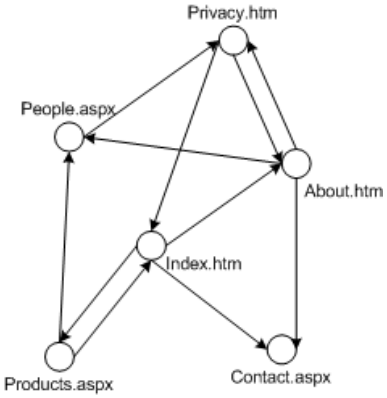

Рисунок 2. Модель страниц, составляющих веб-сайт

2.2 Взвешенные и невзвешенные ребра

Обычно графы используются для моделирования набора «вещей» и взаимоотношений между этими «вещами». Например, граф на Рисунке 2 моделирует страницы веб-сайта и их перекрестные ссылки (гиперссылки). Иногда, однако, важно связать некую стоимость соединения от одного узла к другому.

Карта может быть очень легко смоделирована в виде графа, в котором города являются узлами, а дороги между городами – ребрами. Если мы желаем определить кратчайший путь от одного города к другому, то нам сперва необходимо определить стоимость путешествия от одного города к другому. Логическое решение будет состоять в том, чтобы дать каждому ребру вес (*weight*), такой как, например, расстояние в милях от одного города к другому.

Рисунок 3 показывает граф, который представляет несколько городов в южной Калифорнии. Стоимость любого определенного пути от одного города к другому представляет собой сумму стоимостей ребер составляющих путь. Тогда кратчайшим путем будет такой путь, стоимость которого минимальна. На Рисунке 3, например, путешествие от Сан Диего до Санта Барбары будет иметь длину 210 миль, если ехать через Риверсайд, затем к Барстоу И затем обратно к Санта Барбаре. Кратчайший путь, однако, будет составлять 100 миль до Лос Анжелеса и затем еще 30 миль до Санта Барбары.

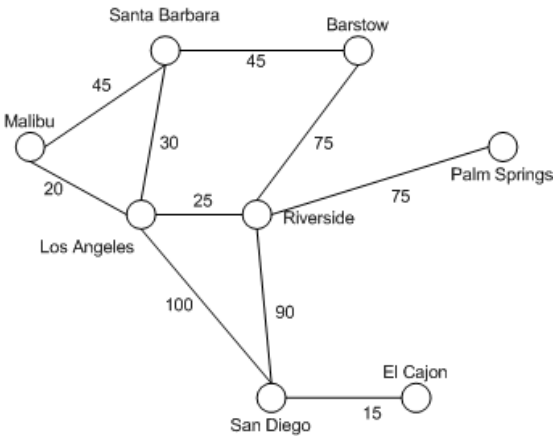

Рисунок 3. Граф городов Калифорнии с ребрами, на которых указаны расстояния в милях

Необходимо понимать, что направленность и взвешенность ребер являются ортогональными. Иными словами граф может иметь одну из четырех организаций ребер:

- Направленный, взвешенные ребра
- Направленный, невзвешенные ребра
- Ненаправленный, взвешенные ребра
- Ненаправленный, невзвешенные ребра

Граф на Рисунке 1 представляет собой ненаправленный граф с невзвешенными ребрами. Рисунок 2 показывает направленный граф с невзвешенными ребрами, и Рисунок 3 показывает ненаправленный граф со взвешенными ребрами.

2.3 Разреженные и плотные графы

В то время как граф может иметь ноль или несколько ребер, обычно граф имеет больше ребер, чем узлов. Какое максимальное числе ребер может иметь граф у которого n узлов? Это зависит от того, является ли граф направленным или нет. Если граф направленный, то каждый узел может ребро к каждому узлу. То есть, все n узлов могут иметь $n - 1$ ребер, давая в общем $n * (n - 1)$ ребер, что приблизительно равно n^2.

Comment [T2]: В этой статье я предполагаю, что узлы не могут иметь ребер к самим себе. В общем случае, однако, графы позволяют иметь ребра от узла v обратно к узлу v. Если таким ребрам позволено существовать в графе, то общее число ребер в направленном графе будет равно n^2.

Если граф является ненаправленным, то тогда один узел, скажем v_1, может иметь ребро к любому другому узлу, или $n - 1$ ребер. Следующий узел, скажем v_2, может иметь по крайней мере $n - 2$ ребер, так как уже существует ребро от v_2 к v_1. Третий узел, v_3, может иметь не более $n - 3$ ребер, и так далее. Поэтому, для n узлов, в графе будет существовать не более $(n - 1) + (n - 2) + ... + 1$ ребер. Суммируя, мы получим $[n * (n-1)] / 2$, или, как вы уже могли предположить, в точности в два раза меньше, чем ребер у ориентированного графа.

Если граф имеет существенно меньше ребер, чем n^2, то граф называется разреженным (*sparse*). Например, граф с n узлами и n ребрами, или даже с $2n$ ребрами будет называться разреженным. Граф с числом ребер близким к максимально возможному числу называется плотным (*dense*).

При использовании графов в алгоритмах очень важно знать отношение числа узлов к числу ребер. Как мы это увидим далее в статье асимптотическое время выполнения операций выполняемых над графом, обычно, выражается в терминах числа узлов и ребер в графе.

3 Создание класса графов

В то время как граф является широко распространенной структурой данных, которая используется при моделировании различных проблем, .NET Framework не имеет встроенной структуры данных для реализации графов. Частично это связано с тем, что эффективная реализация класса `Graph` зависит от множества факторов специфичных для каждой конкретной решаемой задачи. Например, графы обычно моделируются одним из двух способов:

- Список смежности
- Матрица смежности

Эти две техники отличаются в том, как хранятся узлы и ребра внутри класса `Graph`. Давайте изучим оба подхода и взвесим все «за и против» каждого из них.

3.1 Представление графа с использованием списка смежности

В Части III мы создали базовый класс `Node` для представления узлов. Этот базовый класс был расширен для построения специализированных классов узлов для классов `BinaryTree`, `BST` и `SkipList`. Так как каждый узел в графе имеет произвольное число соседей, то может показаться убедительным, что мы можем просто использовать класс `Node` для представления узла графа, так как класс `Node` состоит из значения и произвольного числа экземпляров `Node`. Однако, хотя этот базовый класс и является шагом в правильном направлении, в нем все еще не хватает некоторых возможностей, таких как ассоциирование веса со связями между соседними узлами. Один из вариантов состоит в том, чтобы создать класс `GraphNode`, который наследуется от базового класса `Node` и расширяет его, добавляя требуемые возможности. Тогда каждый класс `GraphNode` будет отслеживать всех своих соседей в свойстве `Neighbors` базового класса.

Класс `Graph` содержит `NodeList`, который хранит множество экземпляров `GraphNode`, которые и представляют узлы графа. То есть граф представляется в виде множества узлов и каждый узел обслуживает список своих соседей. Такое представление называется списком смежности (*adjacency list*) и показано на Рисунке 4.

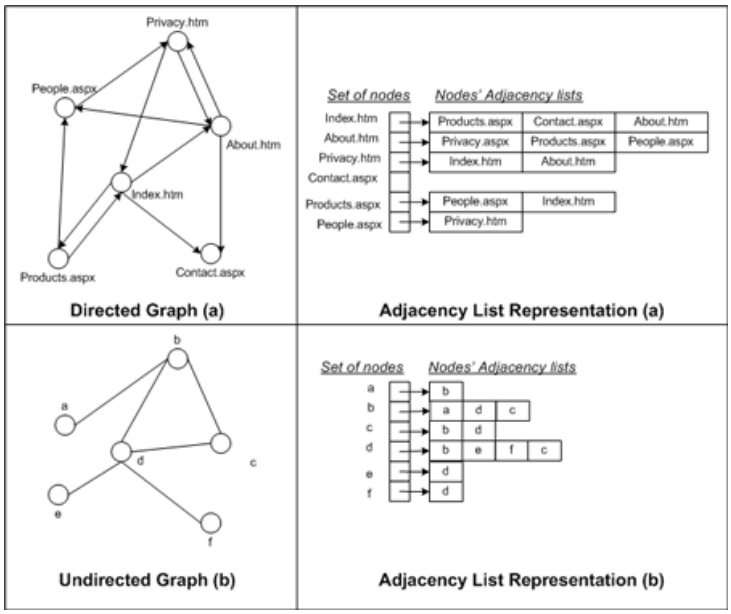

Рисунок 4. Список смежности представленный в графическом виде

Отметьте, что в случае ненаправленного графа представление в виде списка смежности дублирует информацию о ребрах. Например, в представлении списка смежности (b) на Рисунке 4, узел *a* имеет *b* в своем списке смежности, а также узел *b* имеет узел *a* в своем списке смежности.

Каждый узел имеет в точности столько экземпляров `GraphNode` в своем списке смежности, сколько он имеет соседних узлов. Поэтому список смежности является очень экономным представлением графа — вы никогда не храните данных больше, чем нужно. В частности, для графа с *V* узлами и *E* ребрами, представление графа с помощью списка смежности потребует $V + E$ экземпляров `GraphNode` для направленного графа и $V + 2E$ экземпляров `Node` для ненаправленного графа.

Хотя это и не показано на Рисунке 4, списки смежности также могут быть использованы для представления взвешенных графов. Единственное дополнение состоит в том, что для каждого списка смежности узла *n* каждый экземпляр класса `GraphNode` должен хранить стоимость ребра от узла *n*.

Единственный недостаток списка смежности состоит в том, что определение того, существует ли ребро от узла *u* к *v* требует поиска в списке смежности *u*. Для плотных графов *u* будет иметь много экземпляров `GraphNode` в своем списке смежности. Определение того существует ли ребро между двумя узлами, таким образом, требует линейного времени для списка смежности плотных графов. К счастью, при использовании графов нам нет необходимости определять существует ли ребро между двумя узлами. Чаще всего нам потребуется просто перечислить все ребра для данного узла.

3.2 Представление графа с использованием матрицы смежности

Альтернативный метод представления графа состоит в использовании матрицы смежности (*adjacency matrix*). Для графа с *n* узлами, матрица смежности представляет собой двумерный массив размера *n* x *n*. Для взвешенного графа элемент массива (*u*, *v*) будет содержать стоимость ребра между узлами *u* и *v* (или, возможно -1 если ребра между узлами *u* и *v* не существует). Для невзвешенного графа массив будет представлять собой массив булевых значений, в котором ИСТИНА в элементе (*u*, *v*) обозначает ребро от узла *u* к *v* и ЛОЖЬ обозначает отсутствие ребра.

Рисунок 5 показывает, как выглядит матрица смежности графа.

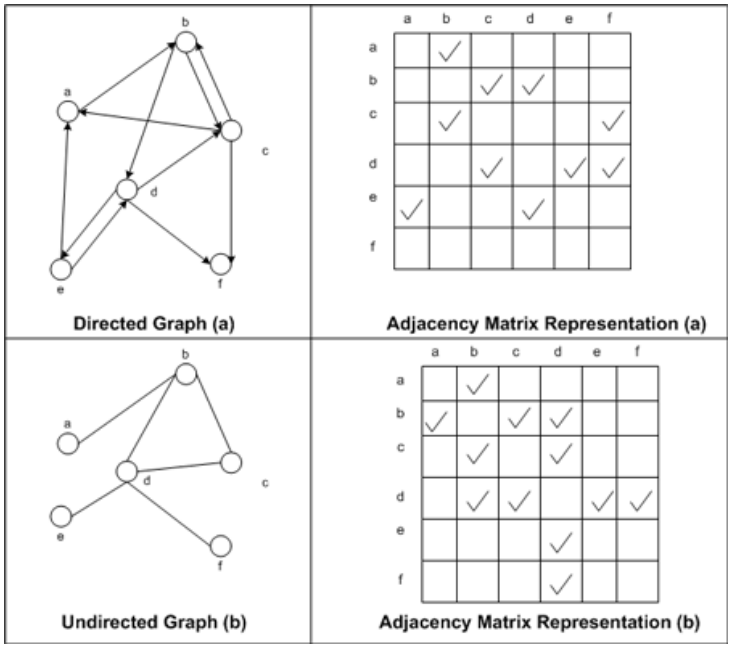

Рисунок 5. Матрица смежности представленная в графическом виде

Заметьте, что ненаправленные графы имеют симметричную матрицу смежности относительно ее диагонали. То есть, если граф имеет ребро от узла *u* к узлу *v*, то в матрице смежности будет присутствовать два соответствующих элемента: (*u*, *v*) и (*v*, *u*).

Так как определение того, существует ли ребро между двумя узлами, это просто извлечение данных из массива, то эту операцию можно выполнить за константное время. Недостаток матриц смежности состоит в их неэффективном использовании памяти. Матрица смежности требует массив с n^2 элементами, поэтому для разреженных графов большая часть матрицы смежности будет пустой. А также, для ненаправленных графов половина матрицы смежности будет хранить повторяющуюся информацию.

В то время, как и матрица смежности и список смежности являются достаточными для внутреннего представления графа в нашем классе `Graph`, давайте двинемся вперед, используя модель списка смежности. Этот подход выбран потому, что он является логическим развитием классов `BinaryTreeNode` и `BinaryTree`, которые были созданы ранее, а также потому, что имеется возможность реализации путем расширения класса

Node используемого в качестве базового класса для структур данных, изученных нами ранее.

3.3 Создание класса GraphNode

Класс GraphNode представляет собой один узел графа, и является производным от базового класса Node, изученного в Части III данной серии статей. Класс GraphNode расширяет базовый класс, предоставляя доступ к свойству Neighbors, а также добавляя новое свойство Cost. Свойство Cost имеет тип List<int>; для взвешенных графов Cost[i] можно использовать для задания стоимости, ассоциированной при движении от GraphNode к Neighbors[i].

```
public class GraphNode<T> : Node<T>
{
    private List<int> costs;

    public GraphNode() : base() { }
    public GraphNode(T value) : base(value) { }
    public GraphNode(T value, NodeList<T> neighbors) : base(value, neighbors)
{ }

    new public NodeList<T> Neighbors
    {
        get
        {
            if (base.Neighbors == null)
                base.Neighbors = new NodeList<T>();

            return base.Neighbors;
        }
    }

    public List<int> Costs
    {
        get
        {
            if (costs == null)
                costs = new List<int>();

            return costs;
        }
    }
}
```

Как показано в коде класса GraphNode, класс предоставляет два свойства:

- **Neighbors:** обеспечивает глобальный доступ к защищенному свойству базового класса. Neighbors имеет тип NodeList<T>.
- **Costs:** List<int> хранит веса из GraphNode к определенному соседнему узлу.

3.4 Построение класса Graph

Вспомните, что с использованием техники списка смежности граф поддерживает список своих узлов. А каждый узел, в свою очередь, имеет список смежных узлов. Поэтому, создавая класс Graph нам необходимо иметь список экземпляров GraphNode. Это множество узлов обслуживается экземпляром класса NodeList. (Мы изучили класс NodeList в Части III; этот класс использовался в классах BinaryTree и BST, а также был

расширен при создании класса SkipList.) Класс Graph предоставляет свой набор узлов посредством метода Nodes.

В дополнение, класс Graph имеет набор методов для добавления узлов, направленных и ненаправленных, а также взвешенных и невзвешенных ребер между узлами. Метод AddNode() добавляет узел к графу, а методы AddDirectedEdge() и AddUndirectedEdge() позволяют связать взвешенное или невзвешенное ребро с двумя узлами.

Кроме методов для добавления ребер, класс Graph также имеет метод Contains(), который возвращает логическое значение, указывающее, существует ли определенное значение в графе или нет. Присутствует также метод Remove(), который удаляет GraphNode и все ребра из графа. Соответствующий код класса Graph приведен ниже (некоторые из переопределенных методов для добавления ребер и узлов были удалены из кода для краткости):

```
public class Graph<T> : IEnumerable<T>
{
    private NodeList<T> nodeSet;

    public Graph() : this(null) {}
    public Graph(NodeList<T> nodeSet)
    {
        if (nodeSet == null)
            this.nodeSet = new NodeList<T>();
        else
            this.nodeSet = nodeSet;
    }

    public void AddNode(GraphNode<T> node)
    {
        // добавляет узел в граф
        nodeSet.Add(node);
    }

    public void AddNode(T value)
    {
        // добавляет узел в граф
        nodeSet.Add(new GraphNode<T>(value));
    }

    public void AddDirectedEdge(GraphNode<T> from, GraphNode<T> to, int cost)
    {
        from.Neighbors.Add(to);
        from.Costs.Add(cost);
    }

    public void AddUndirectedEdge(GraphNode<T> from, GraphNode<T> to, int
cost)
    {
        from.Neighbors.Add(to);
        from.Costs.Add(cost);

        to.Neighbors.Add(from);
        to.Costs.Add(cost);
    }

    public bool Contains(T value)
    {
        return nodeSet.FindByValue(value) != null;
    }
```

```
    public bool Remove(T value)
    {
        // сперва удалить узел из множества узлов
        GraphNode<T> nodeToRemove = (GraphNode<T>)
nodeSet.FindByValue(value);
        if (nodeToRemove == null)
            // узел не найден
            return false;

        // в противном случае узел найден
        nodeSet.Remove(nodeToRemove);

        // пройти по каждому узлу в наборе узлов, удаляя ребра к данному узлу
        foreach (GraphNode<T> gnode in nodeSet)
        {
            int index = gnode.Neighbors.IndexOf(nodeToRemove);
            if (index != -1)
            {
                // удалить ссылку на узел и его стоимость
                gnode.Neighbors.RemoveAt(index);
                gnode.Costs.RemoveAt(index);
            }
        }

        return true;
    }

    public NodeList<T> Nodes
    {
        get
        {
            return nodeSet;
        }
    }

    public int Count
    {
        get { return nodeSet.Count; }
    }
}
```

3.5 Использование класса Graph

Вплоть до этого места мы создали все необходимые классы для структуры данных нашего графа. Вскоре мы переключим наше внимание на некоторые алгоритмы работы с графами, такие как построение минимального остовного дерева и поиск кратчайшего пути от заданного узла ко всем другим узлам, но перед тем как мы это сделаем давайте изучим как использовать класс `Graph` в приложениях написанных на языке C#.

Как только мы создали экземпляр класса `Graph`, следующей задачей будет добавление узлов `Node` к графу. Это включает в себя вызов метода `AddNode()` для каждого узла, который необходимо добавить к графу. Давайте создадим граф, представленный на Рисунке 2. Нам необходимо начать с добавления шести узлов. Для каждого из этих узлов свойство `Key` будет хранить имя файла веб-страницы; мы оставим свойство `Data` пустым (`null`), хотя по идее там можно хранить содержимое файла, или набор ключевых слов описывающих содержимое веб-страницы.

```
Graph<string> web = new Graph<string>();
web.AddNode("Privacy.htm");
```

```
web.AddNode("People.aspx");
web.AddNode("About.htm");
web.AddNode("Index.htm");
web.AddNode("Products.aspx");
web.AddNode("Contact.aspx");
```

Далее, нам необходимо добавить ребра. Так как данный граф является направленным невзвешенным графом, мы будем использовать метод `AddDirectedEdge(u, v)` класса `Graph` для добавления ребра от узла *u* к узлу *v*.

```
web.AddDirectedEdge("People.aspx", "Privacy.htm");  // People -> Privacy

web.AddDirectedEdge("Privacy.htm", "Index.htm");    // Privacy -> Index
web.AddDirectedEdge("Privacy.htm", "About.htm");    // Privacy -> About

web.AddDirectedEdge("About.htm", "Privacy.htm");    // About -> Privacy
web.AddDirectedEdge("About.htm", "People.aspx");    // About -> People
web.AddDirectedEdge("About.htm", "Contact.aspx");   // About -> Contact

web.AddDirectedEdge("Index.htm", "About.htm");      // Index -> About
web.AddDirectedEdge("Index.htm", "Contact.aspx");   // Index -> Contacts
web.AddDirectedEdge("Index.htm", "Products.aspx");  // Index -> Products

web.AddDirectedEdge("Products.aspx", "Index.htm");  // Products -> Index
web.AddDirectedEdge("Products.aspx", "People.aspx");// Products -> People
```

После выполнения этих операций `web` представляет собой граф, показанный на Рисунке 2. После того как мы построили граф мы, естественно, захотим получить ответы на некоторые вопросы. Например, для только что построенного графа мы захотим узнать ответ, "Каково минимальное количество ссылок, по которым должен пройти пользователь, чтобы достичь любой веб-страницы, начиная с домашней страницы (`Index.htm`)?" Чтобы дать ответы на такие вопросы нам придется использовать алгоритмы, работающие с графами. В следующем разделе мы изучим два наиболее известных алгоритма для взвешенных графов: построение минимального остовного дерева и нахождение кратчайшего пути от заданного узла ко всем остальным.

4 Некоторые общие алгоритмы работы с графами

Так как графы являются такими структурами данных, которые можно использовать для моделирования проблем реального мира, то существует неисчислимое множество алгоритмов созданных для нахождения решений различных задач. Для углубления нашего понимания графов давайте посмотрим на два наиболее изученных приложения графов: нахождение минимального остовного дерева и нахождение кратчайшего пути от узла-источника ко всем остальным узлам.

4.1 Проблема минимального остовного дерева

Представьте, что вы работаете в телефонной компании, и, что ваша задача состоит в предоставлении телефонных линий в поселок с 10 домами, каждый из которых помечен от H1 до H10. В частности это предполагает необходимость прокладки отдельного кабеля к каждому дому. То есть кабель должен проходить через дома H1, H2, и так далее вплоть до H10. Из-за географических препятствий—холмов, деревьев, рек, и прочего — прокладывать кабель от одного дома к дому может не быть оптимальным.

Рисунок 6 показывает эту проблему изображенную в виде графа. Каждый узел представляет собой дом, а ребра это те линии, которыми один дом может быть соединен с другим. Веса ребер представляют собой расстояния между домами. Ваша задача состоит в том, чтобы соединить все дома, использовав при этом минимальное количество телефонного провода

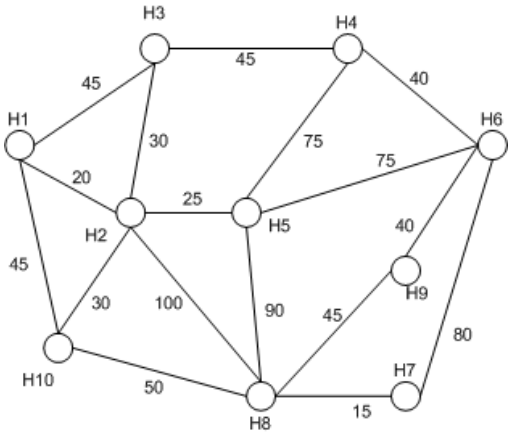

Рисунок 6. Графическое представление поселка с 10 домами и телефонными линиями

Для связного ненаправленного графа существует такое подмножество ребер, которые соединяют все узлы и не образуют циклов. Такое подмножество ребер формирует дерево (потому что оно состоит из количества ребер на единицу меньшего, чем количество узлов и является ацикличным), и называется остовным деревом (*spanning tree*). Для данного графа существует несколько остовных деревьев. На Рисунке 7 показано два правильных остовных дерева для графа с Рисунка 6. (Ребра, формирующие остовное дерево выделены жирными линиями.)

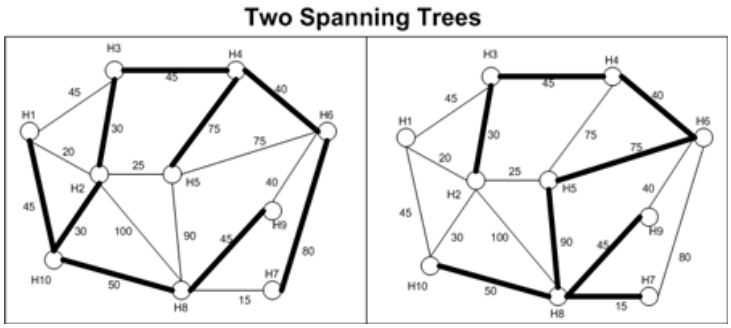

Рисунок 7. Множество остовных деревьев основанные на Рисунке 6

Для графов со взвешенными ребрами различные остовные деревья будут иметь различные связанные с ними стоимости, где стоимость представляет собой сумму весов ребер входящих в остовное дерево. Тогда минимальным остовным деревом (*minimum spanning tree*) будет остовное дерево с минимальной стоимостью.

Существует два основных подхода к решению проблемы минимального остовного дерева. Один из подходов состоит в построении остовного дерева путем выбора ребер с минимальным весом, с учетом того, что выбранное ребро не создает цикла среди уже выбранных ребер. Этот подход показан на Рисунке 8.

Comment [T3]: Первый подход к решению задачи был открыт Джозефом Крускалом (Joseph Kruskal) в 1956 в Белл Лабс (Bell Labs). Вторая техника была изобретена в 1957 году Робертом Примом (Robert Prim), также исследователем из Белл Лабс. Имеется изобилие информации по этим двум алгоритмам в Интернет, включая апплеты Java демонстрирующие работу алгоритмов графически (http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/kruskal/Kruskal.shtml | http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/PrimApp.shtml), а также исходный код алгоритмов на различных языках программирования.

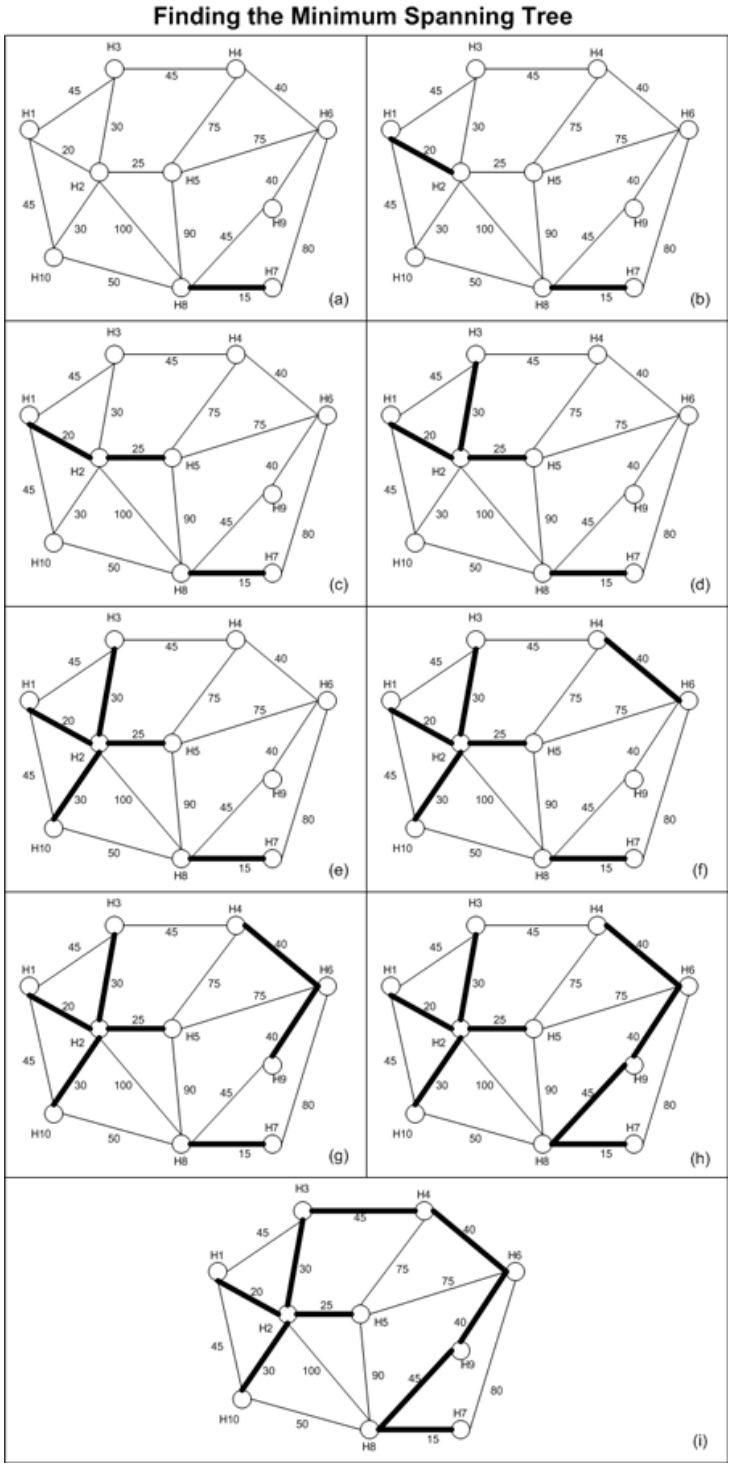

Рисунок 8. Минимальное остовное дерево, которое использует ребра с минимальным весом

Другой подход строит остовное дерево путем деления узлов графа на два непересекающихся множества: узлы, которые находятся в остовном дереве и узлы, которые в него пока еще не добавлены. На каждой итерации ребро с наименьшим весом, которое соединяет узлы остовного дерева с узлом не находящимся в остовном дереве, добавляется к остовному дереву. Чтобы начать работу алгоритма выбирается начальный узел произвольным образом. Рисунок 9 иллюстрирует этот подход в действии, используя H1 в качестве стартового узла. (На Рисунке 9 те узлы, которые входят в множество узлов остовного дерева, закрашены светло-желтым цветом.)

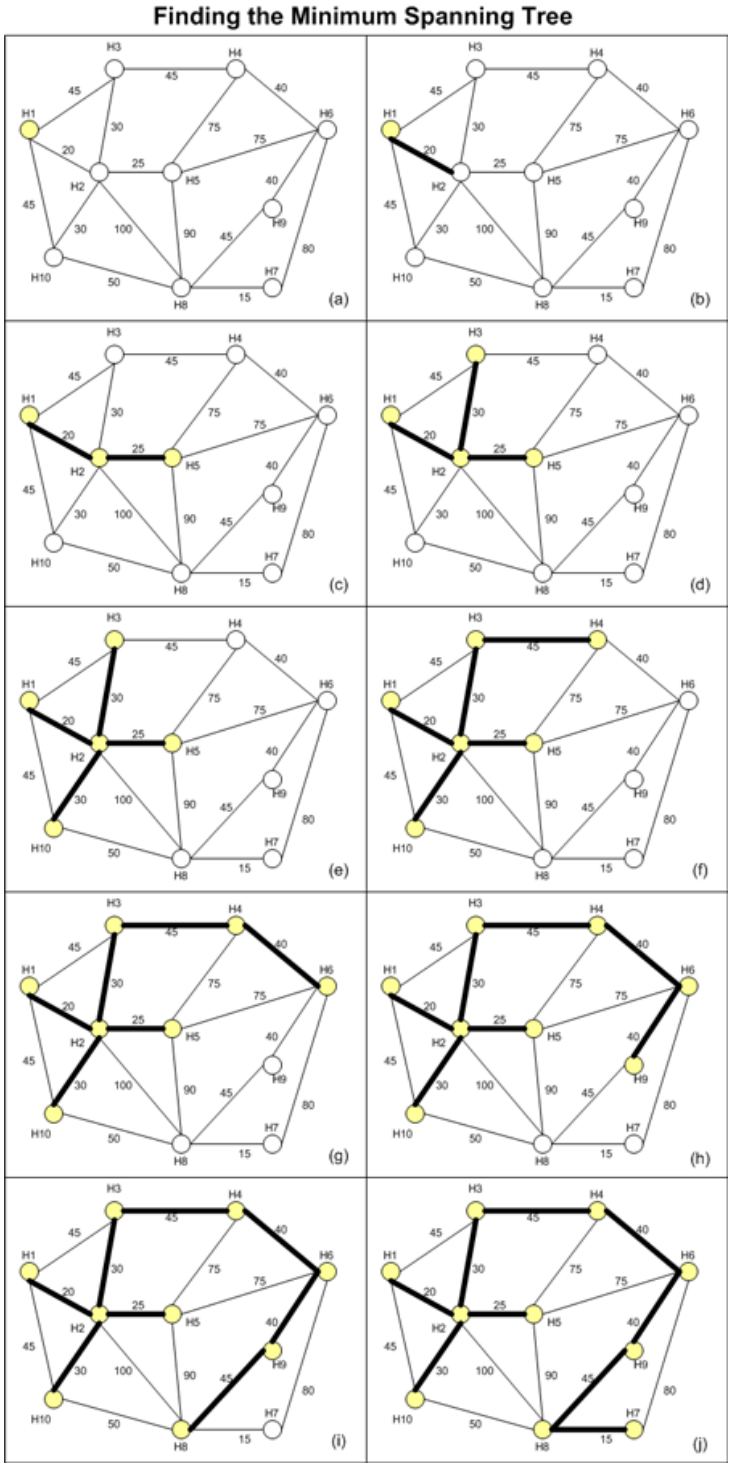

Рисунок 9. Метод Прима для нахождения минимального остовного дерева

Отметьте, что техники, проиллюстрированные на Рисунках 8 и 9, привели к построению одного и того же минимального остовного дерева. Если граф имеет только одно минимальное остовное дерево, то оба эти подхода ведут к одному и тому же решению. Если же граф имеет несколько минимальных остовных деревьев, то эти два подхода могут привести к различным результатам (однако, оба результата будут правильными).

4.2 Вычисление кратчайшего пути из заданного источника

При перелете из одного города в другой, головная боль возникает из-за необходимости найти путь, который требует минимального количества пересадок—кому понравиться лететь из Нью Йорка в Лос Анжелес, сперва перелетев из Нью Йорка в Чикаго, затем из Чикаго в Денвер и только потом из Денвера в Лос Анжелес? Наверное, большинство людей предпочло бы лететь прямо из Нью Йорка в Лос Анжелес.

Предположим, однако, что вы не такой человек, и что вы желаете сберечь свои деньги больше, чем свое время, и что вы больше всего заинтересованы в нахождении самого дешевого маршрута, не зависимо от количества пересадок. Это может означать перелет из Нью Йорка в Майами, затем из Майами в Даллас, затем из Далласа в Финикс, из Финикса в Сан Диего и наконец из Сан Диего в Лос Анжелес.

Мы можем решить эту задачу, моделируя существующие маршруты перелетов и их стоимости в виде направленного взвешенного графа. Рисунок 10 показывает такой граф.

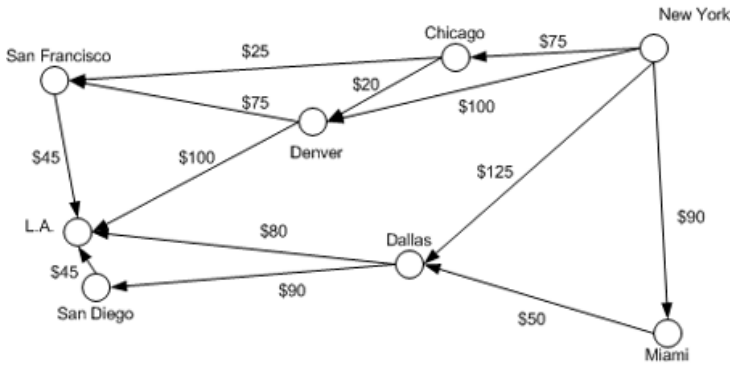

Рисунок 10. Моделирование возможных рейсов самолетов основанное на стоимости билета

Что нам интересно было бы знать, так это самый дешевый путь из Нью Йорка в Лос Анжелес. Инспектируя граф мы быстро определим, что такой путь существует из Нью Йорка в Чикаго, затем в Сан Франциско и наконец в Лос Анжелес, но, чтобы компьютер смог решить поставленную задачу, нам необходимо сформулировать алгоритм.

Один из наиболее заметных ученых нашего времени – Эдсгер Дейкстра (Edsger Dijkstra) – изобрел наиболее широко используемый алгоритм для поиска кратчайшего пути от заданного узла ко всем остальным узлам во взвешенном направленном графе. Этот алгоритм, называемый алгоритмом Дейкстры, работает с использованием двух таблиц, каждая из которых имеет по одной записи на каждый узел. Это таблицы:

- таблица расстояний, которая хранит самое последнее лучшее расстояние от узла-источника ко всем другим узлам.

- таблица маршрута, которая, для каждого узла *n*, указывает, какоц узел использовался, чтобы достичь *n* с учетом наименьшего расстояния.

Первоначально в таблице расстояний каждая строка имеет некоторое произвольное большое значение (например, положительную бесконечность) за исключением начального узла, который имеет расстояние к самому себе равное 0. Все строки таблицы маршрута имеют значение `null`. Также поддерживается набор узлов *Q*, которые необходимо проверить; в начале работы алгоритма этот набор содержит все узлы графа.

Алгоритм работает выбирая (и удаляя) узлы из *Q*, которые имеют наименьшее значение в таблице расстояний. Пусть такой узел называется *n* и значение в таблице расстояний для *n* будет *d*. Для каждого из ребер узла *n*, выполняется проверка, чтобы удостовериться, является ли *d* плюс стоимость от *n* к соседнему узлу меньше, чем значение для этого соседнего узла в таблице расстояний. Если это так, то найден более оптимальный путь для достижений соседнего узла и в этом случае таблица расстояний и таблица пути соответствующим образом обновляются.

Чтобы пояснить этот алгоритм давайте, применим его к графу, представленному на Рисунке 10. Так как мы хотим найти самый дешевый путь из Нью Йорка в Лос Анжелес, то мы используем Нью Йорк в качестве отправного узла. Наша таблица расстояний в начальном состоянии содержит значение бесконечности для всех других городов и значение 0 для Нью Йорка. Таблица маршрута содержит `null` во всех строках, и *Q* содержит все узлы (смотрите Рисунок 11).

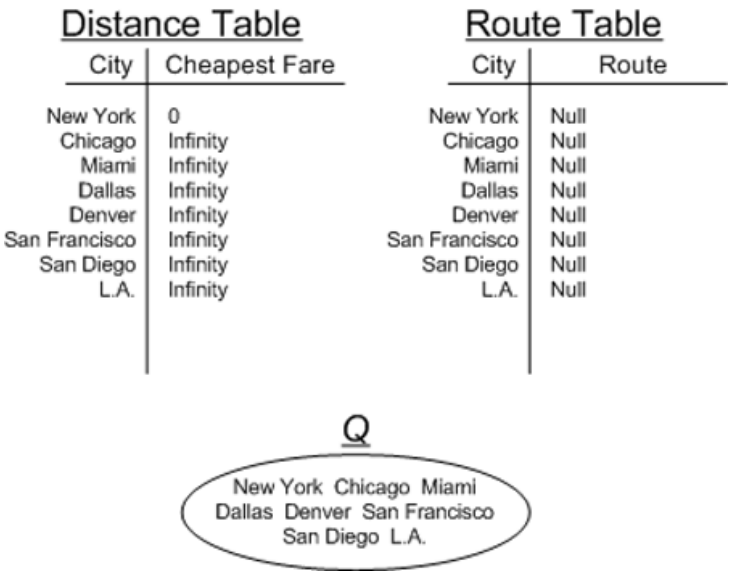

Рисунок 11. Таблица расстояний и таблица маршрутов для определения самого дешевого пути (cheapest fare)

Мы начинаем извлекая город из *Q* который имеет наименьшее значение в таблице расстояний—Нью Йорк. Затем мы исследуем каждый соседний узел Нью Йорка и проверяем, чтобы стоимость полета из Нью Йорка к этому соседнему городу была меньше, чем лучшая известная на данный момент стоимость, а именно стоимость из

таблицы расстояний. После этой первой проверки мы удалили Нью Йорк из *Q* и обновили таблицы расстояний и маршрута для городов Чикаго, Денвер, Майами и Даллас.

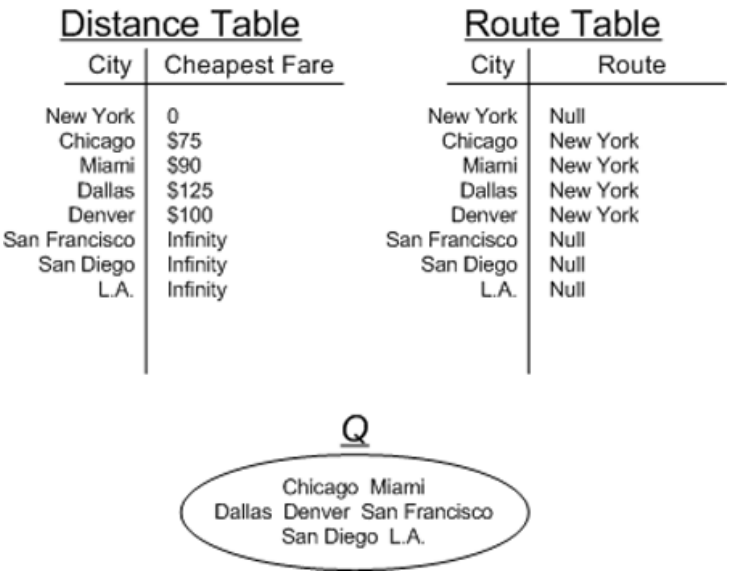

Рисунок 12. Шаг 2 в процессе определения самого дешевого пути (cheapest fare)

Следующий шаг итерации дает нам самый дешевый город из *Q*, Чикаго, и затем проверяем его ближайших соседей, чтобы определить есть ли лучшая стоимость. В частности мы попытаемся определить есть ли лучший маршрут, чтобы добраться до Сан Франциско или Денвера. Очевидно, что стоимость перелета до Сан Франциско из Чикаго—$75 + $25—меньше, чем бесконечность, поэтому необходимо обновить строку Сан Франциско. Отметьте также, что стоимость перелета из Чикаго в Денвер меньше, чем стоимость перелета из Нью Йорка в Денвер ($75 + $20 < $100), поэтому строка Денвера также подлежит обновлению. На Рисунке 13 показаны значения в таблицах и содержимое множества *Q* после обработки города Чикаго.

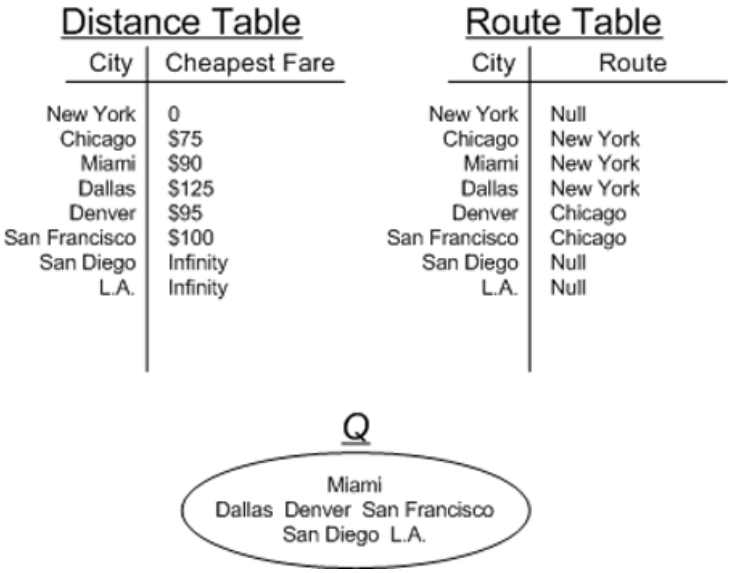

Рисунок 13. Состояние таблиц после завершения третьего шага процесса

Этот процесс продолжается до тех пор пока в *Q* не останется узлов. Рисунок 14 *Q* исчерпалось.

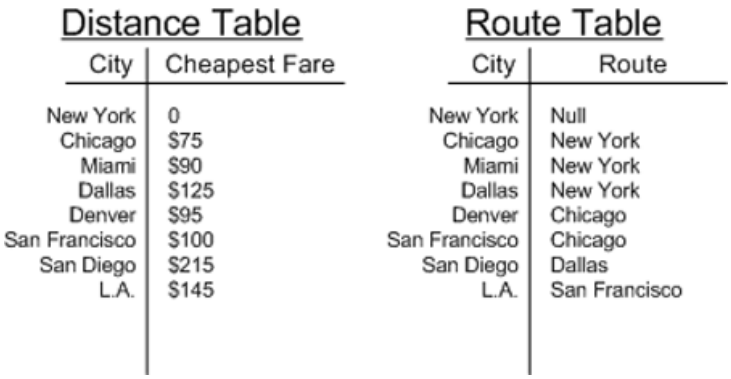

Рисунок 14. Окончательный результат определения самого дешевого пути (cheapest fare)

После того, как мы полностью исчерпали содержимое набора *Q*, таблица расстояний будет содержать наименьшие стоимости перелета из Нью Йорка в каждый город. Чтобы определить путь перелета для прибытия в Лос Анжелес необходимо начать с рассмотрения строки Лос Анжелес в таблице маршрута и проследить его обратно до Нью Йорка. То есть, строка таблицы для Лос Анжелеса содержит Сан Франциско, означая, что самый дешевый перелет в Лос Анжелес будет из Сан Франциско. Строка в таблице маршрута для Сан Франциско содержит Чикаго, означая, что вы доберетесь до Сан Франциско через Чикаго. И наконец, строка в таблице маршрута для Чикаго содержит Нью ЙОрк. Собирая все это вместе мы видим, что самый дешевый перелет лежит из Нью

Йорка в Чикаго, затем в Сан Франциско и затем в Лос Анжелес, и его стоимость составляет $145.

4 Заключение

Графы это широко используемые структуры данных, поскольку их можно использовать для моделирования многих проблем из реального мира. Граф состоит из набора узлов с произвольным числом связей, или ребер, с другими узлами. Эти ребра могут быть либо направленными, либо ненаправленными, а также либо взвешенными, либо невзвешенными.

В этой статье мы изучили основы графов и построили класс `Graph`. Этот класс аналогичен классу `BinaryTree`, который мы создали в Части III, различие состоит только в том, что вместо ссылок на два ребра, класс `GraphNodes` может иметь произвольное число ссылок. Это подобие не является неожиданным, поскольку деревья являются специальным случаем графов.

В дополнение к созданию класса `Graph`, мы также рассмотрели два алгоритма для работы с графами, проблему минимального остовного дерева и вычисление кратчайшего пути от узла источника ко всем другим узлам в направленном взвешенном графе. Хотя мы и не рассмотрели пример исходного кода для реализации этих алгоритмов. В Интернет доступно много хороших примеров такого кода.

В следующей статье нашей серии, Части VI, мы рассмотрим, как эффективно использовать непересекающиеся множества. Непересекающиеся множества это набор двух или более множеств, которые не имеют общих элементов. Например, в алгоритме Прима для нахождения минимального остовного дерева узлы графа могут быть разделены на два непересекающихся множества: множество узлов, входящих в построенное на данный момент остовное дерево и множество узлов, еще не входящих в остовное дерево.

Игорь Изварин