Всестороннее изучение структур данных с использованием C# 2.0

Часть VI: Эффективное представление множеств

Шестая часть этого цикла статей изучает то, как реализовать математическую конструкцию множество. Множество это неупорядоченная коллекция уникальных элементов, которые можно пересчитать и сравнить с другими множествами различными способами. В этой статье мы рассмотрим структуры данных, с помощью которых можно реализовать как обыкновенные множества, так и непересекающиеся множества.

1 Введение

Одной из наиболее основных математических конструкций является множество (*set*), которое представляет собой неупорядоченный набор уникальных объектов. Объекты, содержащиеся в множестве называются элементами множества. Формально множества обозначаются заглавными наклонными буквами, элементы которых заключены в фигурные скобки (`{...}`). Примеры этой нотации приведены ниже:

```
S = { 1, 3, 5, 7, 9 }
T = { Scott, Jisun, Sam }
U = { -4, 3.14159, Todd, x }
```

В математике множества обычно состоят из чисел, как это приведено для множества *S*, которое содержит нечетные положительные числа меньше 10. Однако отметьте, что элементы множества могут быть чем угодно—числами, людьми, строками, символами, переменными и так далее. Пусть *T*, например, содержит имена людей; множество *U* содержит смесь чисел, имен и переменных.

В этой статье мы начнем с базового знакомства с множествами, включая нотацию и операции, которые можно выполнять над множествами. Вслед за этим мы изучим как эффективно реализовать структуру данных множества. Статья завершится изучением непересекающихся множеств и наилучшей структуры данных для их представления.

2 Основы множеств

Для начала вспомним, что множество это просто набор элементов. Оператор «является элементом из», обозначаемый $x \in S$, обозначает, что x является элементом множества S. Например, если множество S содержит нечетные положительны числа меньше 10, то $1 \in S$. Читая такую нотацию вы произносите "1 является элементом S." Кроме того, что 1 является элементом S, мы также имеем $3 \in S$, $5 \in S$, $7 \in S$ и $9 \in S$. Оператор «не является элементом из», обозначаемый $x \notin S$, означает, что x не является элементом множества S.

Число уникальных элементов в множестве называется мощностью множества (*cardinality*). Множество {1, 2, 3} имеет мощность 3, так же как и множество {1, 1, 1, 1, 1, 1, 1, 2, 3} (так как оно содержит только три уникальных элемента). Множество может вообще не

содержать элементов. Такое множество называется пустым множеством (*empty set*), и обозначается {} или □, и имеет мощность равную 0.

При первом знакомстве с множествами разработчики очень часть подразумевают, что они равноценны коллекциям, таким как Список (List). Однако существуют некие изысканные отличия. Список (List) это *упорядоченная* коллекция элементов. Каждый элемент в Списке имеет ассоциированный с ним порядковый индекс, который и определяет порядок. Также в Списке могут быть идентичные элементы.

Comment [A.A.1]: В математике, упорядоченная коллекция элементов, которая позволяет иметь идентичные элементы называется списком (*list*). Два списка, L_1 и L_2 считаются равными если, и только если, для всего диапазона элементов в списке i-ый элемент списка L_1 равен i-му элементу списка L_2.

Множество, с другой стороны, *неупорядочено* и имеет только *уникальные* элементы. Так как множества неупорядочены, то элементы множества могут быть перечислены в любом порядке. То есть, множества {1, 2, 3} и {3, 1, 2} рассматриваются как эквивалентные. Также, любые идентичные элементы в множестве рассматриваются как избыточные. Множество {1, 1, 1, 2, 3} и множество {1, 2, 3} являются эквивалентными. Два множества являются эквивалентными, если они содержат одинаковые элементы. (Эквивалентность множеств обозначается с помощью знака равенства =; если S и T эквивалентны, то это записывается как *S* = *T*.)

Обычно, элементы, которые принадлежат множеству ограничены неким пространством значений (*universe*). Пространство значений это множество всех возможных значений, которые могут появляться в множестве. Например, мы можем быть заинтересованы работать с множествами пространство значений которых содержит только целые числа. Ограничивая пространство значений до целых чисел, мы тем самым не можем иметь множества, которые содержат нецелочисленные элементы, такие как 8.125 или Sam.

2.1 Операции отношения над множествами

Существует набор операций отношения, которые хорошо известны и используются с целыми числами. Некоторые из наиболее часто используемых, особенно в языках программирования, включают <, <=, =, !=, > и >=. Операторы отношения определяют находится ли операнд с левой стороны от знака операции отношения в отношении к операнду с правой стороны на основании критерия определенного этим оператором отношения. Операторы отношения возвращают логическое значение «истина» ("true") или «ложь» ("false"), указывающее выполняется ли заданное отношение между операндами. Например, *x* < *y* возвращает «истину», если *x* меньше, чем *y*, и «ложь» в противном случае. (Конечно же смысл «меньше чем» зависит от типа данных *x* и *y*.)

Операторы отношения подобные <, <=, =, !=, > и >= обычно используются с числами. Множества, как мы это уже видели, используют оператор отношения = для индикации того, что два множества эквивалентны (и возможно использование != для обозначения того, что два множества не эквивалентны), однако операторы отношения <, <=, > и >= не определены для множеств. Как, например, определить, является ли множество {1, 2, 3} меньше, чем множество {Scott, 3.14159}?

Вместо обозначений < и <=, множества используют следующие операции отношения: подмножество (*subset*) и точное подмножество (*proper subset*), обозначаемые □ и □, соответственно. (Более ранние тексты используют обозначение □ для подмножества и □ для точного подмножества.) *S* является подмножеством *T*—обозначаемое как *S* □ *T*—если каждый элемент множества *S* принадлежит множеству *T*. То есть, *S* является подмножеством *T* если оно содержится в *T*. Если *S* = {1, 2, 3}, а *T* = {0, 1, 2, 3, 4, 5}, то *S* □ *T* так как каждый элемент *S* – 1, 2 и 3—являются элементами *T*. *S* является точным

Comment [A.A.2]: Так как □ аналогично <=, то имело бы смысл существование оператора отношения аналогичного >=. Этот оператор отношения называется надмножеством (*superset*), и обозначается □; точное надмножество (*proper superset*) обозначается □. Так же как и в случае операторов <= и >=, *S* □ *T* если и только если *T* □ *S*.

подмножеством *T*—обозначаемое как *S* □ *T*—если *S* □ *T* и *S* □ *T*. То есть, если *S* = {1, 2, 3} и *T* = {1, 2, 3}, то *S* □ *T* так как каждый элемент *S* является также элементом *T*, однако *S* □ *T* так как *S* = *T*. (Отметьте аналогию между операциями отношения < и <= для чисел и операциями отношения □ и □ для множеств.)

Используя новый оператор подмножества мы можем более формально определить эквивалентность множеств. Имея множества *S* и *T*, *S* = *T* если и только если *S* □ *T* и *T* □ *S*. Другими словами, *S* и *T* являются эквивалентными если и только если каждый элемент *S* содержится в *T*, и каждый элемент *T* содержится в *S*.

2.2 Операции над множествами

Так же как и операторы отношения, многие из операций применимые к числам не применимы так же и к множествам. Общие операции над числами включают сложение, умножение, вычитание, возведение в степень и так далее. Для множеств определяются четыре основных операции:

1. **Объединение:** объединение двух множеств, обозначаемое *S* □ *T*, сродни сложению чисел. Оператор объединения возвращает множество, которое содержит все элементы из *S* и все элементы из *T*. Например, {1, 2, 3} □ {2, 4, 6} даст в результате {1, 2, 3, 2, 4, 6}. (Повторяющуюся величину 2 можно удалить, чтобы получить более сжатый ответ {1, 2, 3, 4, 6}.) Формально, *S* □ *T* = {*x* : *x* □ *S* или *x* □ *T*}. Другими словами, результат объединения *S* и *T* представляет собой множество, которое содержит элемент *x* если *x* находится либо в *S* либо в *T*.
2. **Пересечение:** пересечение двух множеств, обозначаемое *S* □ *T*, это множество элементов, которые содержит как *S* так и *T*. Например, {1, 2, 3} □ {2, 4, 6} даст в результате {2}, так как только этот единственный элемент встречается и в множестве {1, 2, 3} и в множестве {2, 4, 6}. Формально, *S* □ *T* = {*x* : *x* □ *S* и *x* □ *T*}. Другими словами, результат пересечения *S* и *T* представляет собой множество, которое содержит элемент *x* если *x* находится и в *S* **и в** *T*.
3. **Разница:** разница двух множеств, обозначаемая *S* – *T*, представляет собой все элементы *S*, которые не содержатся в *T*. Например, {1, 2, 3} – {2, 4, 6} равно {1, 3}, так как 1 и 3 это те элементы *S*, которые не содержатся в *T*. Формально, *S* – *T* = {*x* : *x* □ *S* и *x* □ *T*}. Другими словами, результат вычитания множества *T* из множества *S* представляет собой множество, которое содержит элемент *x* если *x* содержится в *S* и не содержится в *T*.
4. **Дополнение:** Ранее мы обсуждали как обычно множества ограничены известным пространством значений, таким например, как целые числа. Дополнение множества, обозначаемое *S*', равно *U* – *S*. (*U* это пространство допустимых значений.) Если пространство значений представляет собой целые числа от 1 до 10 и *S* = {1, 4, 9, 10}, то *S*' = {2, 3, 5, 6, 7, 8}. Дополнение множества сродни отрицанию числа. Так же как и двойное отрицание числа даст вам первоначальное число, то есть --*x* = *x* – выполнение операции дополнения дважды даст вам первоначальное множество – *S*'' = *S*.

При изучении новых операций очень важно понять природу этих операций. Некоторые из вопросов, которые вы должны задать себе при изучении любых операций:

- **Является ли операция коммутативной?** Оператор *op* является коммутативным, если *x op y* эквивалентно *y op x*. В отношении чисел операция сложения является коммутативной, в то время как операция деления таковой не является.

- **Является ли операция ассоциативной?** То есть, имеет ли значение порядок операций. Если оператор *op* ассоциативен, то *x op* (*y op z*) эквивалентно (*x op y*) *op z*. Опять таки, в отношение чисел операция сложения ассоциативна, а операция деления – нет.

Для множеств операции объединения и пересечения являются коммутативными и ассоциативными. *S* □ *T* эквивалентно *T* □ *S*, и *S* □ (*T* □ *V*) эквивалентно (*S* □ *T*) □ *V*. Разница множеств, однако, не является ни коммутативной ни ассоциативной. (Чтобы увидеть, что разница не является коммутативной, давайте рассмотрим, что {1, 2, 3} – {3, 4, 5} = {1, 2}, но {3, 4, 5} – {1, 2, 3} = {4, 5}.)

2.3 Конечные множества и бесконечные множества

Все примеры множеств, которые мы рассмотрели до сих пор были конечными множествами. Конечное множество это множество, которое имеет конечное число элементов. Хотя это и может показаться алогичным, на первый взгляд, существуют множества с бесконечным числом элементов. Множество целых положительных чисел, например, является бесконечным множеством, так как нет ограничения для количества элементов в множестве.

В математике существует несколько бесконечных множеств, которые так часто используются, что даже получили специальные символы для их обозначения. Это:

N = {0, 1, 2, ...}

Z = {..., -2, -1, 0, 1, 2, ...}

Q = {a/b: a □ Z, b □ Z, и b □ 0}

R = *множестве вещественных чисел*

N это множество натуральных чисел (*natural number*), или положительных целых чисел больших или равных 0. Z это множество целых чисел (*integers*). Q это множество рациональных чисел (*rational numbers*), представляющие собой числа, которые можно выразить в виде дроби, состоящей из двух целых чисел. И, наконец, R это множество вещественных чисел (*real numbers*), которые все являются рациональными числами плюс также иррациональные числа (например, число □ или квадратный корень из 2).

Бесконечные множества, конечно же, не могут быть записаны полностью, так как вы никогда не закончите перечислять их элементы, но вместо этого их можно компактно описать с помощью математических обозначений, как например:

S = {*x* : *x*□ N и *x* > 100}

Здесь *S* представляет собой множество натуральных чисел больших 100.

В этой статье мы исследуем структуры данных для представления конечных множеств. В то время как бесконечные множества имеют свое применение в математике, в

компьютерных программах использование бесконечных множеств практически встречается крайне редко

Comment [A.A.3]: Вычислени е мощности конечного множества достаточно просто. Необходимо просто сосчитать количество элементов в множестве. Но как вычислить мощность бесконечного множества? Обсуждение этого вопроса выходит за рамки данной статьи, однако необходимо понимать, что существуют различные типы мощности для бесконечных множеств. например, множество положительных целых чисел имеет такую же мощность как и множество **всех** целых чисел, однако множество вещественных чисел имеет большую мощность, чем множество всех целых чисел.

2.4 Множества в языках программирования

C++, C#, Visual Basic .NET и Java не предоставляют соответствующих языковых конструкций для работы с множествами. Если вы желаете использовать множества вам придется создать свой собственный класс с соответствующими методами, свойствами и логикой. (Мы все это сделаем в следующем разделе.) Однако, в прошлом были созданы языки программирования, которые предлагали множество в качестве базовой конструкции языка. Pascal, например, предлагает конструкцию set, которую можно использовать для создания множеств в явно определенном пространстве значений (universe). Для работы с множествами Pascal предоставляет оператор `in` для определения принадлежности элемента множеству. Операторы `+`, `*` и `-` используются для объединения, пересечения и разности множеств, соответственно. Приведенный ниже код на языке Pascal иллюстрирует синтаксис, используемый для работы с множествами:

```
/* объявляет переменную с именем possibleNumbers, чье пространство значений
  это множество целых чисел в диапазоне от 1 до 100... */
var
  possibleNumbers = set of 1..100;

...

/* Присваивает множество {1, 45, 23, 87, 14} переменной possibleNumbers */
possibleNumbers := [1, 45, 23, 87, 14];

/* Присваивает possibleNumbers объединение possibleNumbers и {3, 98} */
possibleNumbers := possibleNumbers + [3, 98];

/* Проверить является ли число 4 элементом множества возможных чисел... */
if 4 in possibleNumbers then write("4 is in the set!");
```

Другие ранние языки программирования предлагали более мощные семантические и синтаксические средства для работы с множествами. Язык SETL (сокращение для SET Language) был создан в 70-х годах и предлагал работу с множествами как основным типом данных. В отличие от Паскаля, работая с множествами в SETL вы не ограничены пространством значений множества.

3 Реализация эффективной структуры данных множества

В этом разделе мы построим класс, который обеспечивает функциональность и возможности множества. При создании такой структуры данных, первый вопрос, на который мы должны ответить, это то каким образом мы будем хранить элементы множества. Это решение повлияет на асимптотическое время выполнения операций над структурой данных множества. (Помните, что операции, которые нам нужно выполнять над множеством включают: объединение, пересечение, разницу множеств, подмножество, и принадлежность элемента множеству.)

Чтобы проиллюстрировать насколько метод хранения элементов множества может повлиять на время выполнения операций над множеством давайте представим, что мы создали класс множества, использующий в качестве внутренней структуры данных ArrayList для того, чтобы хранить элементы множества. Тогда, если у нас есть два

множества S_1 и S_2, которые мы хотим объединить (где множество S_1 содержит m элементов, а множество S_2 содержит n элементов), нам понадобится выполнить следующие шаги:

1. Создать новое множество T, которое будет содержать результат объединения S_1 и S_2.
2. Пройти по всем элементам множества S_1 и добавить их в T.
3. Пройти по всем элементам множества S_2. Если элемент еще не существует в T, то добавить его к множеству T.

Сколько шагов потребует операция объединения? Шаг (2) потребует m шагов по m элементам множества S_1. Шаг (3) потребует n шагов, и для каждого элемента в S_2, нам требуется определить присутствует ли данный элемент в T. Чтобы определить существует ли элемент в не отсортированном списке необходимо линейно просмотреть весь список. Поэтому, для каждого из n элементов в S_2 мы должны произвести поиск среди m элементов из T. Это приведет к квадратичному времени выполнения операции объединения порядка $O(m * n)$.

Причина, по которой операция объединения с помощью списков требует квадратичного времени состоит в том, что проверка существования элемента требует линейного времени. То есть, чтобы определить существует ли заданный элемент в множестве, список, с помощью которого реализовано множество, должен быть просмотрен с полным перебором. Если мы сможем снизить время выполнения операции «элемент из» до константы, то тем самым мы сможем улучшить время выполнения операции объединения до линейного $O(m + n)$. Вспомните из второй части нашей серии статей, что Хеш-таблицы и Словари обеспечивают константное время выполнения для определения существует ли элемент в структуре данных. Из этого следует, что Хеш-таблица или Словарь будут более предпочтительным выбором для хранения элементов множества, чем Список.

Если мы потребуем, чтобы пространство значений множества было известно, то тогда можно реализовать более эффективную структуру данных с использованием битовых массивов. Предположим, что пространство значений состоит из элементов e_1, e_2, ..., e_k. Тогда мы можем обозначить множество с помощью k-элементного битового массива; если i-й бит равен 1, то элемент e_i находится в множестве; если, с другой стороны, i-й бит равен 0, то элемент e_i не находится в множестве. Представление множеств в идее битовых массивов не только позволяет существенно сэкономить оперативную память, но также позволяет эффективно реализовать операции над множествами с использованием простых побитовых операций. Например, определение того существует ли элемент e_i в множестве требует константного времени так как требуется проверить только один i-й бит массива. Объединение двух множеств это просто побитовая операция OR (ИЛИ) битовых массивов множеств; пересечение двух множеств это побитовая операция AND (И) битовых массивов множеств. Разница множеств и нахождение подмножества также могут быть выражены в терминах побитовых операций.

Comment [A.A.4]: Битовый массив это компактный массив состоящий из 1 и 0, и обычно реализуемый как массив целых чисел. Так как целое число в .NET Framework имеет 32 бита, то битовый массив может хранить 32 битовых значения в одном элементе массива целых чисел.

Битовые операции это операции выполняемые над отдельными битами, а не над целым числом. Существуют как бинарные битовые операции так и унарные битовые операции. Битовые операторы AND (И) и OR (ИЛИ) являются бинарными, получая для работы два бита и возвращая в качестве результата один бит. Битовый оператор AND возвращает 1 только в том случае когда оба входных операнда равны 1, в противном случае он возвращает 0. Битовый оператор OR возвращает 0 только в том случае, если оба операнда

Давайте рассмотрим как реализовать множество используя битовые операции языка C#.

3.1 Создание класса PascalSet

Необходимо понимать, что для реализации класса множества с использованием эффективных побитовых операций, требуется знать пространство значений множества. Это похоже на то как Паскаль использует множества, поэтому в честь языка программирования Паскаль мы назовем этот класс множества `PascalSet`. `PascalSet` ограничивает пространство значений диапазоном целых чисел или символов (так же как это сделано в языке программирования Паскаль). Этот диапазон может быть задан в конструкторе класса `PascalSet`.

```
public class PascalSet : ICloneable, ICollection, IEnumerable
{
   // Приватные переменные класса
   private int lowerBound, upperBound;
   private BitArray data;

   public PascalSet(int lowerBound, int upperBound)
   {
      // удостовериться, что нижняя граница меньше или равна верхней
      if (lowerBound > upperBound)
         throw new ArgumentException("The set's lower bound cannot be
            greater than its upper bound.");

      this.lowerBound = lowerBound;
      this.upperBound = upperBound;

      // Создать битовый массив
      data = new BitArray(upperBound - lowerBound + 1);
   }

   ...
}
```

Поэтому для создания `PascalSet` чье пространство значений является множеством целых чисел между -100 и 250, можно использовать следующий синтаксис:

```
PascalSet mySet = new PascalSet(-100, 250);
```

3.1.1 Реализация операций над множеством

`PascalSet` реализует стандартный набор операций над множеством—объединение, пересечение и разницу—а также стандартные операции отношения—подмножество, точное подмножество, надмножество и точное надмножество. Операции объединения, пересечения и разницы множество возвращают новый экземпляр `PascalSet`, который содержит результат объединения, пересечения или разницы множеств соответственно. Нижеследующий код метода `Union(`*`PascalSet`*`)` иллюстрирует это поведение:

```
public virtual PascalSet Union(PascalSet s)
{
    if (!AreSimilar(s))
        throw new ArgumentException("Attempting to union two dissimilar
           sets.  Union can only occur between two sets with the same
universe.");

    // выполнить битовое OR для объединения this.data и s.data
    PascalSet result = (PascalSet)Clone();
    result.data.Or(s.data);
```

```
    return result;
}

public static PascalSet operator +(PascalSet s, PascalSet t)
{
   return s.Union(t);
}
```

Метод `AreSimilar(PascalSet)` определяет имеет ли переданное значение `PascalSet` одинаковые верхнюю и нижнюю границы с экземпляром `PascalSet`. Поэтому, объединение (а также пересечение и разность множеств) может быть выполнено только для двух множеств с одинаковым пространством значений. Если два экземпляра `PascalSets` имеют одинаковое пространство значений, то создается новый экземпляр `PascalSet`—`result`—который является точной копией экземпляра `PascalSet` чей метод `Union()` был вызван. Содержимое этого клонированного `PascalSet` изменяется с использованием битовой операции OR, передавая ей содержимое множеств которые должны быть объединены. Заметьте, что класс `PascalSet` также переопределяет оператор `+` для выполнения операции объединения (так же как это реализовано в языке программирования Паскаль).

Аналогично, класс **PascalSet** предоставляет методы для пересечения множеств и разности множеств, а также переопределенные операторы `*` и `-`, соответственно.

3.1.2 Перечисление элементов множества PascalSet

Так как множества представляют собой неупорядоченные наборы элементов, то будет иметь смысл, чтобы класс `PascalSet` реализовал интерфейс `IList`, так как коллекция, которая реализует интерфейс `IList` подразумевает, что список имеет некоторый порядок. Так как `PascalSet` это набор элементов, то также имеет смысл реализовать интерфейсы `ICollection` и `IEnumerable`. А так как `PascalSet` реализует `IEnumerable`, то необходимо предоставить метод `GetEnumerator()` который возвращает экземпляр `IEnumerator` позволяя тем самым разработчику выполнять проход по элементам множества. Этот метод просто проходит по внутреннему массиву `BitArray`, возвращая соответствующее значение для тех битов, которые установлены в 1.

```
public IEnumerator GetEnumerator()
{
    int totalElements = Count;
    int itemsReturned = 0;
    for (int i = 0; i < this.data.Length; i++)
    {
        if (itemsReturned >= totalElements)
            break;
        else if (this.data.Get(i))
            yield return i + this.lowerBound;
    }
}
```

Чтобы перечислить элементы PascalSet, вы можете просто использовать предложение `foreach`. Следующий фрагмент кода демонстрирует создание двух экземпляров PascalSet и затем полного перечисления элементов их объединения:

```
PascalSet a = new PascalSet(0, 255, new int[] {1, 2, 4, 8});
PascalSet b = new PascalSet(0, 255, new int[] { 3, 4, 5, 6 });

foreach(int i in a + b)
```

```
MessageBox.Show(i);
```

Этот код отобразит семь окон сообщения одно за другим, которые показывают значения 1, 2, 3, 4, 5, 6 и 8.

4 Поддержка наборов непересекающихся множеств

Следующий раз, когда вы будете выполнять поиск с помощью Google отметьте, что возле каждого найденного результата присутствует ссылка "Similar Pages." Если вы перейдете по этой ссылке, то Google покажет вам список URL, который относятся к найденному результату, с которого вы перешли. Хотя мы и не знаем, как Google определяет какие ссылки являются относящимися к найденному результату, одним из подходов к решению такой задачи мог бы быть следующий:

- Пусть x будет веб страницей для которой мы желаем найти относящиеся к ней другие страницы.
- Пусть S_1 будет множеством веб страниц, на которые имеется ссылка со страницы x.
- Пусть S_2 будет множеством веб страниц, на которые имеются ссылки со страниц множества S_1.
- Пусть S_3 будет множеством веб страниц, на которые имеются ссылки со страниц множества S_2.
- ...
- Пусть S_k будет множеством веб страниц, на которые имеются ссылки со страниц множества S_{k-1}.

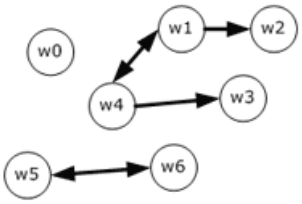
Все страницы в множествах S_1, S_2, и до S_k связаны со страницей x. Вместо того, чтобы находить список веб страниц относящихся к данной, по запросу, мы можем оптимизировать эту операцию создав множество таких веб страниц для всех веб страниц сразу, и сохранить этот результат в базе данных или другом постоянном хранилище данных. Затем, когда пользователь переходит по ссылке "Similar Pages" мы просто извлекаем информацию из базы данных и отображаем ее на экране.

Google поддерживает базу данных всех известных ему веб страниц. Каждая из этих веб страниц имеет множество связей (ссылок). Мы можем вычислить множество связанных веб страниц используя следующий алгоритм:

1. Для каждой веб страницы в базе данных создать множество и поместить эту единственную страницу в это множество. После завершения первого шага алгоритма если мы имеем n веб страниц в базе данных, то мы получим n одноэлементных множеств.
2. Для веб страницы x из базы данный, найти все те веб страницы, которые с ней непосредственно связаны. Назовем эти страницы S. Для каждого элемента p из S, произведем объединение множества, содержащего p с множеством с элементом x.
3. Повторить шаг 2 для всех веб страниц из базы данных.

После завершения шага 3 все веб страницы в базе данных будут разбиты на три группы. Рисунок 1 показывает графическое представление этого алгоритма в действии.

Imagine we have seven Web pages in the database, w0 through w6. The graph to the right shows how these pages link to one another.



Step 1

First, place each Web page in a singleton set. (Sets are denoted by color.)

Step 2

Union w0 and its neighbors. (w0 has no neighbors)

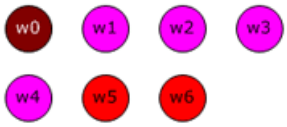
Next, union w1 and its neighbors (w2 and w4).

Next, union w2 and its neighbors (w1).

Next, union w3 and its neighbors (w3 has no neighbors).

Next, union w4 and its neighbors (w1 and w3).

Next, union w5 and its neighbors (w6).



Next, union w6 and its neighbors (w5).

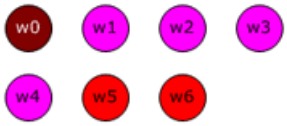


Рисунок 1. Графическое представление алгоритма группирования связанных Веб-страниц.

Представьте, что у нас есть семь страниц в базе данных, от w0 до w6. Граф с правой стороны показывает как эти страницы связаны друг с другом.

ШАГ 1. Сперва разместить каждую веб страницу в отдельное множество. (Множества обозначены различными оттенками серого.)

ШАГ 2. Объединить w0 и его соседей (w0 не имеет соседей).

Далее, объединить w1 и его соседей (w2 и w4).

Далее, объединить w2 и его соседей (w1).

Далее, объединить w3 и его соседей (w3 не имеет соседей).

Далее, объединить w4 и его соседей (w1 и w3).

Далее, объединить w5 и его соседей (w6)

Далее, объединить w6 и его соседей (w5)

Изучая рисунок 1, отметьте, что в конце присутствуют три разделения:

- w0
- w1, w2, w3 и w4
- w5 и w6

Поэтому, когда пользователь нажимает на ссылку "Similar Pages" для w2, то он увидит связи к w1, w3 и w4; нажимая на ссылку "Similar Pages" для w6 будет показана только ссылка на w5.

Заметьте, что при решении этой частной задачи использовалась только одна операция над множествами—объединение. Более того, все веб страницы разделяются на непересекающиеся множества (*disjoint sets*). Пусть дано произвольное число множеств, эти множества называются непересекающимися (*disjoint*) если они не имеют общих, или одинаковых, элементов. Например, {1,2,3} и {4,5,6} являются непересекающимися, в то время как {1,2,3} и {2,4,6} нет, потому что у них есть один общий элемент 2. На всех стадиях, показанных на Рисунке 1, каждое из множеств, содержащее веб страницы, является непересекающимся. То есть никогда не будет такого случая, чтобы веб страница содержалась более чем в одном множестве в данный момент времени.

При работе с непересекающимися множествами данным образом нам часто бывает необходимо знать какому из непересекающихся множеств принадлежит тот или иной элемент. Чтобы идентифицировать каждое множество мы выбираем произвольным образом представителя множества (*representative*). Представитель это элемент непересекающегося множества, который уникально идентифицирует все множество целиком. С использованием понятия представителя теперь можно определить для двух произвольных элементов находятся ли они в одном и том же непересекающемся множестве просто взглянув имеют ли они одного и того же представителя.

Структура данных непересекающихся множеств должна предоставлять два следующих метода:

- **GetRepresentative(element):** это метод получает элемент в качестве входного параметра и возвращает представителя элемента.
- **Union(element, element):** этот метод получает два элемента. Если элементы из одного и того же непересекающегося множества, то `Union()` ничего не делает. Если же, однако, два элемента из различных непересекающихся множеств, то `Union()` комбинирует два непересекающихся множества в одно множество.

Проблема с которой мы сталкиваемся теперь состоит в том как эффективно поддерживать несколько непересекающихся множеств, в такой среде где эти множества часто сливаются из двух множеств в одно. Существует две основные структуры данных которые можно использовать для решения этой проблемы—одна использует последовательность связанных списков, а другая набор деревьев.

4.1 Поддержка непересекающихся множеств с помощью связанных списков

В четвертой части нашей серии статей мы потратили несколько минут на повторное ознакомление со связанными списками. Вспомните, что связанные списки представляют собой множество узлов, которые обычно имеют одну ссылку на своих ближайших соседей. Рисунок 2 показывает связанный список с четырьмя элементами.

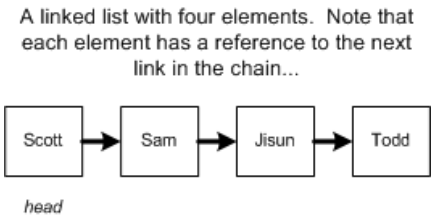


Рисунок 2. Связанный список из четырех элементов

Связанный список состоящий из четырех элементов. Отметьте, что каждый элемент имеет ссылку на следующий элемент в цепочке...

Голова

Для структуры данных непересекающихся множеств одно множество представляется в виде модифицированного связанного списка. Кроме ссылки на своего соседа каждый узел в связанном списке непересекающихся множеств также имеет ссылку на представителя множества (set's representative). Как показано на рисунке 3, *все* узлы в связанном списке указывают на один и тот же узел, как их представителя, который, по соглашению, является головой списка. (На рисунке 3 показано представление непересекающихся множеств в виде связанного списка. Заметьте, что для каждого из непересекающихся множеств существует соответствующий связанный список и что узлы связанного списка содержат элементы данного непересекающегося множества.)

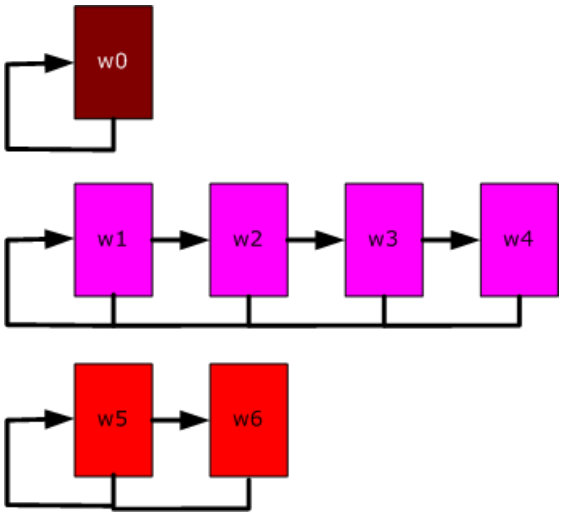


Рисунок 3. Представление непересекающихся множеств в виде связанных списков из финальной фазы алгоритма проанализированного на Рисунке 1.

Так как каждый элемент множества имеет прямую ссылку на представителя множества, то метод `GetRepresentative(`*`element`*`)` выполняется за константное время. (Чтобы понять почему, рассмотрим, что независимо от количества элементов в множестве всегда потребуется ровно одна операция для нахождения представителя данного элемента, так как для этого требуется всего лишь проверка указателя на представителя элемента.)

Используя подход со связанными списками комбинирование двух непересекающихся множеств в одно требует добавления одного связанного списка к концу другого, а также обновления ссылок на представителя списка для каждого из добавленных узлов. Процесс объединения двух непересекающихся множеств представлен на Рисунке 4.

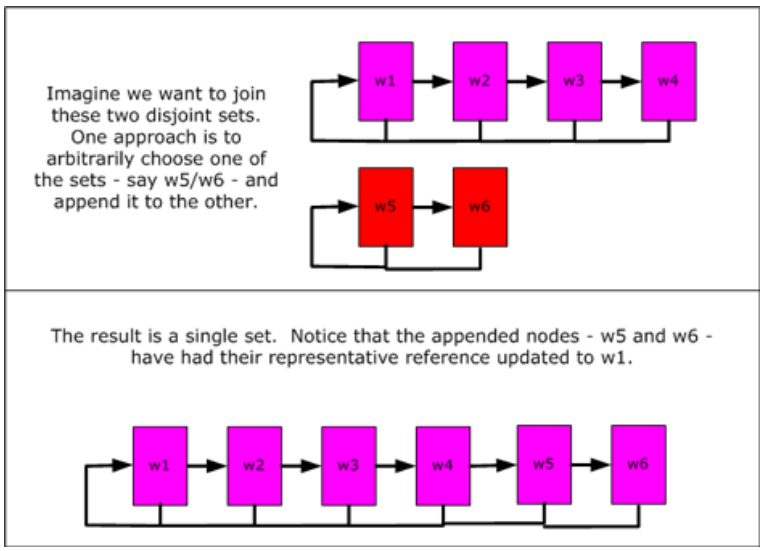


Рисунок 4. Процесс объединения двух непересекающихся множеств

Представьте, что мы хотим объединить эти два непересекающиеся множества. Один из подходов будет состоять в том, чтобы произвольным образом выбрать одно из множеств – скажем w5/w6 – и добавить его в конец другого.

Результат объединения представляет собой одно множество. Заметьте, что добавленные в конец узлы – w5 и w6 – обновили свои ссылки на представителя на узел w1.

При объединении двух непересекающихся множеств корректность алгоритма не нарушается от того, какое из двух множеств добавляется в конец другого. Однако, время выполнения операции может измениться. Представьте, что наш алгоритм объединения случайным образом выбирает один из двух связанных списков для добавления. В том случае, если нам все время не везет, представьте, что будет всегда выбран для добавления более длинный список из двух. Это может негативно повлиять на время выполнения операции объединения, так как нам необходимо просмотреть все элементы добавляемого связанного списка для обновления их ссылки на представителя списка. То есть, представьте, что мы сделали n непересекающихся множеств, S_1 до S_n. Каждое множество будет иметь один элемент. Затем мы произведем n - 1 объединений, соединяя все n множеств в одно большое множество с n элементами.

Суммируя число операций, которые должны быть выполнены на каждом шаге мы найдем, что вся последовательность шагов—n операций создания множества и n-1 объединений—требует квадратичного времени—$O(n^2)$.

Этот наихудший случай времени выполнения может проявиться потому, что существует вероятность, что объединение будет выбирать длинный список для добавления к короткому. Добавление длинного списка требует, чтобы обновилось большее число ссылок на представителей узлов. Лучший подход состоит в том, чтобы отслеживать размер каждого множества и затем, при объединении двух множеств, добавлять меньший из двух связанных списков. Время выполнения при использовании такого улучшенного подхода будет составлять $O(\text{n } \log_2 n)$.

Чтобы оценить улучшения полученные от $O(\text{n } \log_2 n)$ в сравнении с $O(n^2)$, рассмотрим рисунок 5, который показывает показатель роста n^2 черным цветом и показатель роста $\text{n } \log_2 n$ серым цветом. Для небольших значений n эти два графика сравнимы, но при как только n превышает 32, то $\text{n } \log_2 n$ растет гораздо медленнее, чем n^2. Например, выполнение 64 объединений потребует более 4,000 операций при использовании простого связанного списка, в то время как при использовании оптимизированного связанного списка это потребует всего лишь 384 операции. Эти различия становятся более заметными при возрастании n.

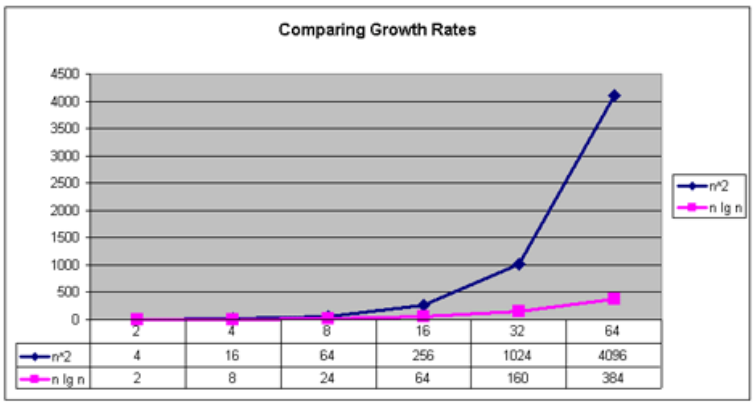


Рисунок 5. Сравнений функций роста n^2 и $n \log_2$

Сравнение показателей роста

4.2 Поддержка непересекающихся множеств с помощью леса

Непересекающиеся множества также можно реализовать с использованием леса (*forest*). Лес это множество деревьев (вспомнили?). Вспомните также, что в случае связанных списков представителем множества был элемент головы списка. В случае реализации с помощью леса, каждое множество реализуется в виде дерева и представителем множества является корень дерева.

С использованием связанных списков нахождение представителя для данного элемента было быстрым так как каждый узел имел ссылку на его представителя. Однако, в случае подхода с использованием связанных список операция объединения длилась дольше, так как это требовало обновления ссылок на представителя для всех узлов, которые были добавлены к первому списку. Подход с использованием леса нацелен на ускорение операции объединения за счет увеличения стоимости нахождения представителя множества для данного элемента.

Подход с использованием леса реализует каждое из непересекающихся множеств в виде дерева, в котором в качестве представителя используется корень дерева. Чтобы объединить вместе два множества одно дерево присоединяется к другому в качестве потомка. Рисунок 6 иллюстрирует этот подход графически.

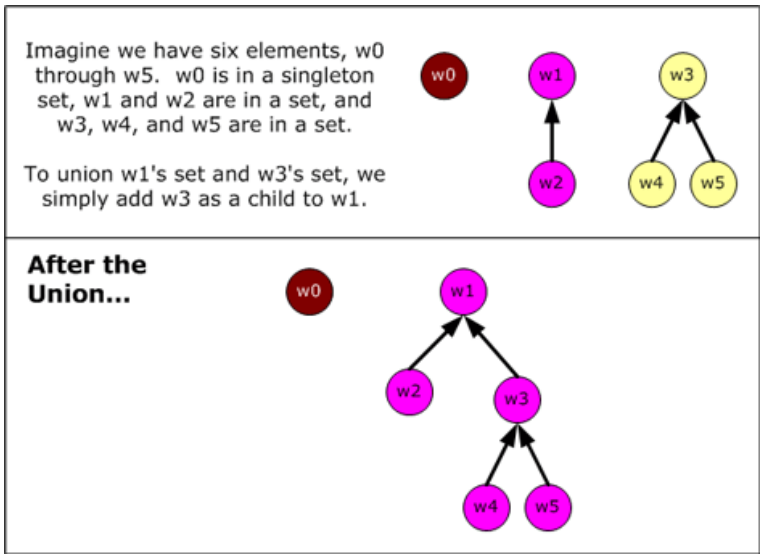


Рисунок 6. Объединение двух множеств

Представьте, что мы имеем шесть элементов, от w0 до w5. w0 это множество состоящее из одного элемента, w1 и w2 находятся в одном множестве, w3, w4 и w5 также находятся в одном множестве.

Чтобы объединить множество w1 с множеством w3 мы просто добавляем w3 в качестве потомка w1.

После выполнения операции объединения

Чтобы объединить два множества требуется константное время, так как только одному узлу требуется обновить ссылку на представителя. (На Рисунке 6, чтобы объединить множества w1 и w3, все что мы должны сделать это обновить ссылку w3, чтобы она указывала на w1—узлы w4 и w5 не требуют никакой модификации.)

Сравнивая с реализацией с помощью связанных списков подход с использованием леса улучшил время необходимое для объединения двух непересекающихся множеств, однако ухудшил время нахождения представителя множества. Единственный способ которым мы можем найти представителя множества для данного элемента это пройти вверх по дереву множества до тех пор пока не будет найден корень. Представьте, что нам нужно найти представителя узла w5 (после того, как множества w1 и w3 были объединены). Нам необходимо пройти вверх по дереву до тех пор, пока не будет достигнут корень—сперва к w3, а затем к w1. Отсюда, нахождение представителя множества требует времени пропорционального глубине дерева, а не константного времени, как это было в случае представления с помощью связанных списков.

Подход с использованием леса предлагает две оптимизации, которые будучи применены обе, приводят к линейному времени выполнения операций с *n* непересекающимися множествами, означая тем самым, что каждая операция в среднем требует константного времени. Эти две оптимизации называются «объединение по рангу» и «сжатие пути». Чего мы пытаемся избежать в этих двух оптимизациях так это чтобы последовательность операций объединения не генерировала высокое и тонкое дерево. Как обсуждалось в

третьей статье цикла отношение высоты дерева к его ширине обычно влияет на время выполнения операций над деревом. В идеальном случае дерево должно быть как можно более широким.

4.2.1 Оптимизация объединением по рангу

Объединение по рангу сродни оптимизации связанных списков, когда более короткий список добавляется в конец более длинного связанного списка. А именно, объединение по рангу поддерживает ранг для каждого корня множества, что обеспечивает верхнюю границу высоты дерева. При объединении двух множеств множество с меньшим рангом добавляется в качестве потомка к корню множества с большим рангом. Объединение по рангу помогает удостовериться, что наши деревья будут широкими. Однако, даже с использованием объединения по рангу мы все еще можем получить высокие хотя и широкие деревья. Рисунок 7 показывает изображение дерева, которое может быть сформировано последовательностью операций объединения, которые используют только оптимизацию объединения по рангу. Проблема состоит в том, что для листьев дерева в правой его части все еще необходимо выполнить достаточное число операций для нахождения представителя множества.

Comment [A.A.5]: Подход с использованием леса при реализации только оптимизации объединения по рангу имеет такое же время выполнения как и представления с использованием оптимизированного связанного списка.

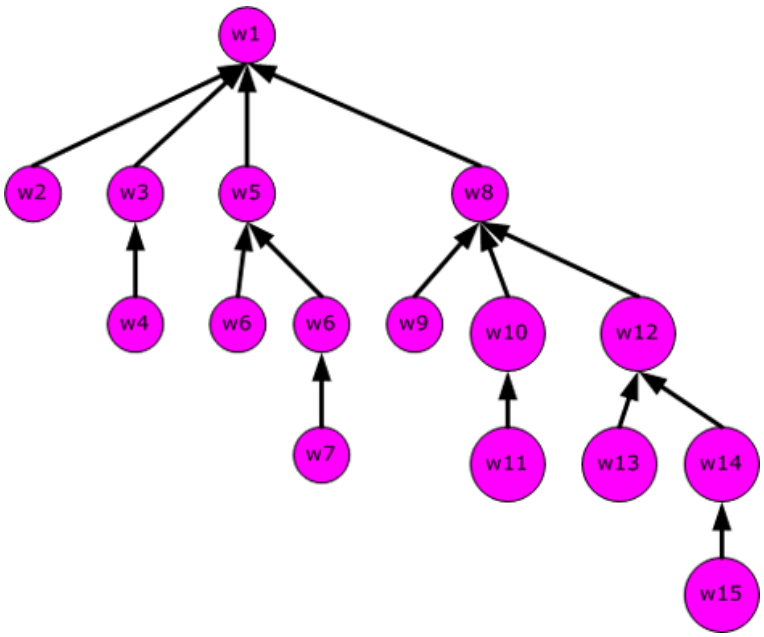


Рисунок 7. Дерево, которое может быть сформировано последовательностью операций объединения, которые используют только оптимизацию объединения по рангу

4.2.2 Оптимизация сжатия пути

Так как высокое дерево делает операцию поиска представителя множества дорогостоящей, то в идеале мы бы хотели иметь широкие и плоские деревья. Оптимизация сжатия пути работает над уплощением дерева. Как мы уже обсуждали ранее, как только запрашивается операция поиска представителя элемента, алгоритм начинает прохождение вверх по дереву до корня. Способ, которым выполняется оптимизация

сжатия пути в этом алгоритме, состоит в том, что при прохождении вверх по дереву у каждого посещенного узла обновляется ссылка на родителя – она обновляется, указывая на корень дерева.

Чтобы понять как это уплощение работает рассмотрим дерево на Рисунке 7. Теперь представьте, что вам необходимо найти представителя множества для узла w13. Алгоритм начнет с узла w13, перейдет вверх к w12, затем к w8, и в конце к w1, возвращая w1 в качестве представителя. Используя сжатие пути этот алгоритм также произведет побочный эффект для узлов w13 и w12 настраивая их указатель на корень—w1. Рисунок 8 показывает как будет выглядеть дерево после того, как было произведено сжатие пути.

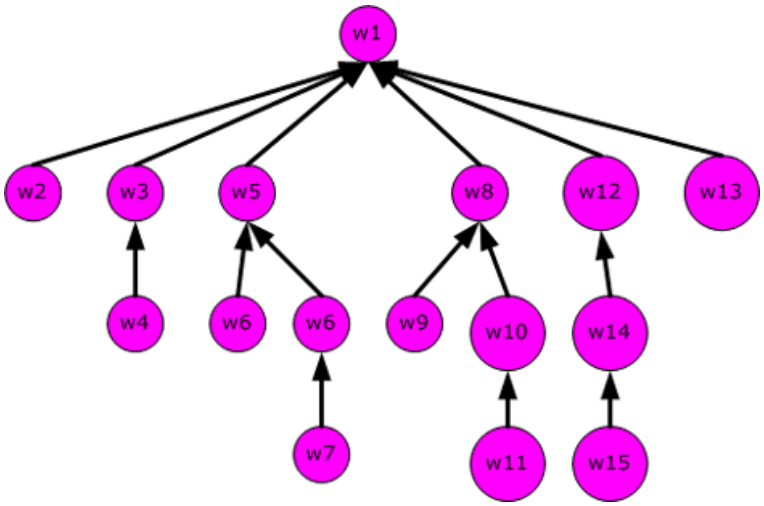


Рисунок 8. Дерево после операции сжатия пути

Сжатие пути имеет некие накладные расходы при первом поиске представителя, но дает существенный выигрыш в последующих поисках представителя. То есть, после того как было выполнено сжатие пути, нахождение представителя множества для узла w13 требует одного шага, так как w13 является потомком корня. На Рисунке 7, до сжатия пути, нахождение представителя узла w13 требовало три шага. Основная идея, присутствующая здесь, состоит в том, что вы платите за улучшения только один раз, а выигрыш от улучшения получаете каждый раз при выполнении проверки в будущем.

При применении алгоритмов объединения по рангу и сжатия пути, время, требуемое для выполнения n операций с непересекающимися множествами является линейным. То есть, подход с использованием леса, и использование двух оптимизаций имеет время выполнения $O(n)$. Вам придется поверить на слово так как формальное доказательство временной сложности достаточно сложное и длинное и потребует нескольких страниц печатного текста.

Это была последняя статья из серии…

Игорь Изварин