МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Юсипов И.И.
Калякулина А.И.
Иванченко М.В.

АЛГОРИТМЫ БИОИНФОРМАТИКИ

Учебное пособие (лекционный материал)

2018

УДК _____
ББК _____
П44

Юсипов И.И., Калякулина А.И., Иванченко М.В. Алгоритмы биоинформатики: Учебное пособие (лекционный материал). Нижний Новгород: Нижегородский госуниверситет, 2018. 112 с.

Рецензент: д.ф.-м.н. Канаков О.И.

Предметом рассмотрения настоящего курса являются основные задачи биоинформатики, которые могут быть сформулированы как задачи поиска, выравнивания, а также другие модельные задачи на строках и графах. Цель курса состоит в изучении соответствующих методов и алгоритмов биоинформатики. Особое внимание уделяется формированию у студентов навыков реализации и применения рассматриваемых методов к решению конкретных задач биоинформатики. Задачей дисциплины является теоретическое и практическое освоение вышеупомянутых методов и алгоритмов. Курс лекций предназначен для студентов института ИТММ ННГУ, специализирующихся в области математического моделирования.

Для успешного усвоения материала необходимо предварительное изучение дисциплины “Языки программирования”, “Основы объектно-ориентированного программирования” и “Введение в алгоритмы биоинформатики”.

Сборник лекций основан на курсах, читаемых Н.И. Вяххи [1], и работах П.А. Певзнера [2, 3].

Ответственный за выпуск:

председатель методической комиссии ИИТММ ННГУ,

Грезина А.В., к.ф.-м.н., доцент

УДК _____
ББК _____

СОДЕРЖАНИЕ

Лекция 1. Секвенирование ДНК.

Определение порядка нуклеотидов в геноме или секвенирование генома представляет собой фундаментальную задачу в биоинформатике. Геномы различаются по длине; геном человека составляет примерно 3 миллиарда нуклеотидов, геном *Amoeba dubia*, аморфного одноклеточного организма, примерно в 200 раз длиннее. Этот одноклеточный организм конкурирует с редким японским цветком *Paris japonica* за звание вида с самым длинным геномом.

Первый секвенированный геном, принадлежащий бактериофагу φX174 (то есть вирусу, который охотится на бактерии), имел только 5 386 нуклеотидов и был расшифрован в 1977 году Фредериком Сангером. Спустя четыре десятилетия после получения Нобелевской премии за данное открытие, секвенирование генома вырвалось на передний план исследований биоинформатики, поскольку стоимость секвенирования резко упала. Из-за снижения стоимости секвенирования теперь есть тысячи секвенированных геномов, в том числе многих млекопитающих.

К концу 1980-х годов биологи проводили секвенирование вирусов, содержащих сотни тысяч нуклеотидов, но идея секвенирования бактериальных (не говоря уже о человеческих) геномах оставалась нереализуемой, как экспериментально, так и вычислительно. Действительно, генерация одного прочтения в конце 1980-х годов стоила более доллара, что оценивало последовательности генома млекопитающих в миллиардах. Поэтому были созданы ДНК-массивы с целью дешево генерировать композицию *k*-меров генома, хотя и с меньшей длиной прочтения *k*, чем исходная технология секвенирования ДНК. Например, в то время как сложная методика секвенирования Сангера сгенерировала прочтения длиной 500 нуклеотидов в 1988 году, изобретатели ДНК-матрицы первоначально стремились к созданию прочтений длиной всего 10.

Матрицы ДНК работают следующим образом. Сначала синтезируются все 4^k возможных *k*-меров и присоединяются к массиву ДНК. Массив представляет собой сетку, на которой каждому *k*-меру присваивается уникальное местоположение. Затем флуоресцентно маркируется одноцепочечный фрагмент ДНК (с неизвестной последовательностью) и наносится раствор, содержащий эту помеченную ДНК, на матрицу ДНК. *k*-меры во фрагменте ДНК гибридизуют (связывают) с их обратными комплементарными *k*-мерами на матрице. Все, что нужно сделать, это использовать спектроскопию для анализа того, какие участки на решетке излучают флуоресценцию; обратные дополнения *k*-меров, соответствующие этим сайтам, должны принадлежать (неизвестному) фрагменту ДНК. Таким образом, набор флуоресцентных *k*-

меров в массиве показывает состав фрагмента ДНК. См. Рисунок 1.1 для иллюстрации массива ДНК.

AAA	AGA	CAA	CGA	GAA	GGA	TAA	TGA
AAC	AGC	CAC	CGC	GAC	GGC	TAC	TGC
AAG	AGG	CAG	CGG	GAG	GGG	TAG	TGG
AAT	AGT	CAT	CGT	GAT	GGT	TAT	TGT
ACA	ATA	CCA	CTA	GCA	GTA	TCA	TTA
ACC	ATC	CCC	CTC	GCC	GTC	TCC	TTC
ACG	ATG	CCG	CTG	GCG	GTG	TCG	TTG
ACT	ATT	CCT	CTT	GCT	GTT	TCT	TTT

Рис. 1.1. ДНК-массив, содержащий все возможные 3-меры. 10 флуоресцентно помеченных 3-меров (ACC, ACG, CAC, CCG, CGC, CGT, GCA, GTT, TAC, TTA) представляют собой композицию 3-меров строки CGCACGTTACCG длиной 12 нуклеотидов. Этот массив ДНК не содержит информации о множественности 3-меров.

Сначала мало кто верил, что массивы ДНК будут работать, потому что и биохимическая проблема синтеза миллионов коротких фрагментов ДНК, и алгоритмическая проблема восстановления последовательности казалась слишком сложной. В 1988 году в журнале *Science* появилась заметка о том, что, учитывая объем работы, необходимой для синтеза ДНК-массива, «использование ДНК-массивов для секвенирования просто заменило бы одну ужасающую задачу другой». Это оказалось лишь отчасти верным. В середине 1990-х годов ряд компаний усовершенствовали технологии для разработки больших ДНК-массивов, однако, массивы ДНК в конечном итоге не смогли реализовать идею, которая мотивировала их изобретателей, потому что корректность гибридизации ДНК с массива была слишком низкой и значение k было слишком маленьким.

В то время как исходная цель (секвенирование ДНК) была вне досягаемости, появились два новых неожиданных приложения ДНК-массивов. Сегодня массивы используются для измерения уровня экспрессии генов, а также для анализа генетических вариаций. Эти неожиданные приложения превратили массивы ДНК в многомиллиардную индустрию, включающую компанию *Hyseq*, основанную Радоже Дрманаком, одним из изобретателей ДНК-массивов.

После основания *Hyseq* Дрманак не отказался от своей мечты изобрести альтернативную технологию секвенирования ДНК. В 2005 году он основал *Complete Genomics*, одну из первых компаний Секвенирования Нового Поколения *Next Generation Sequencing* (NGS). *Complete Genomics*, *Illumina*, *Life Technologies* и другие компании NGS впоследствии разработали технологию, позволяющую дешево генерировать почти все *k*-меры генома. Хотя эти технологии сильно отличаются от технологии массивов ДНК, предложенной в 1988 году, интеллектуальное наследие ДНК-массивов по-прежнему можно признать в подходах NGS, что свидетельствует о том, что хорошие идеи не умирают, даже если они терпят неудачу вначале.

Подобно ДНК-массивам, технологии NGS сначала генерировали миллионы довольно коротких подверженных ошибкам прочтений (длиной около 20 нуклеотидов), революция NGS началась в 2005 году. За несколько лет компании NGS смогли увеличить длину прочтений и на порядок улучшить точность. Кроме того, *Pacific Biosciences* и *Oxford Nanopore Technologies* уже генерируют считываемые с ошибками прочтения, содержащие тысячи нуклеотидов. Каким бы ни было будущее, последние события в NGS уже произвели революцию в геномике, и биологи готовятся собирать геномы всех видов млекопитающих на Земле.

Чтобы секвенировать геном, нужно устранить некоторые практические препятствия. Самым большим препятствием является тот факт, что биологам до сих пор не хватает технологии для считывания нуклеотидов генома от начала до конца. Лучшее, что они могут получить, это последовательность гораздо более коротких фрагментов ДНК, называемых прочтениями. Причины, по которым исследователи могут секвенировать небольшие фрагменты ДНК, но не длинные геномы, требуют отдельного обсуждения.

Традиционный метод секвенирования геномов описывается следующим образом: исследователи берут небольшой образец ткани или крови, содержащий миллионы клеток с идентичной ДНК, используют биохимические методы для разложения ДНК на фрагменты, а затем упорядочивают эти фрагменты для получения прочтений (рисунок 1.2).

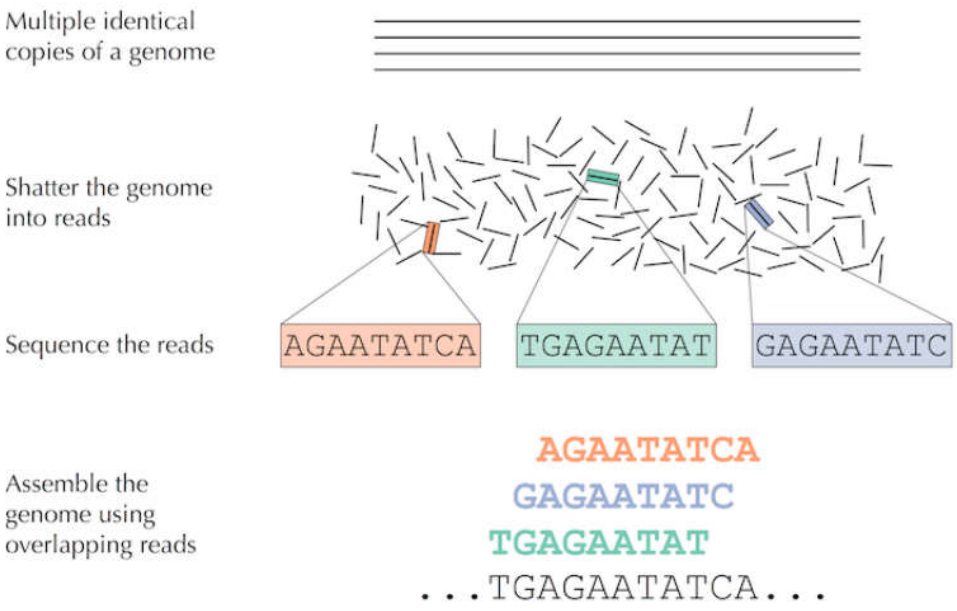


Рис. 1.2. В секвенировании ДНК многие идентичные копии генома разрушены в случайных местах, чтобы сгенерировать короткие прочтения, которые затем секвенируются и собираются в нуклеотидную последовательность генома.

Трудность состоит в том, что исследователи не знают, в какой части генома находятся данные прочтения, и поэтому они должны использовать перекрывающиеся чтения для восстановления генома.

Несмотря на то, что исследователи упорядочили множество геномов, гигантский геном, подобный *Amoeba dubia*, по-прежнему остается недоступным для современных технологий секвенирования. Биологи могут легко генерировать достаточное количество прочтений для анализа большого генома, но сборка этих прочтений по-прежнему представляет собой серьезную вычислительную задачу.

Поскольку ДНК двухцепочечна, не известно априори, какой из цепочек принадлежит данное прочтение, это означает, что заранее неизвестно, нужно использовать прочтение или его обратное дополнение при сборке определенной цепи генома. Современные машины для секвенирования не идеальны, и прочтение, которое они генерируют, часто содержит ошибки. Ошибки секвенирования усложняют сборку генома, потому что они мешают идентифицировать все перекрывающиеся прочтения. Некоторые районы генома не могут быть охвачены никакими прочтениями, что делает невозможным восстановление всего генома.

Так как прочтения, создаваемые современными секвенсорами, часто имеют одинаковую длину, можно предположить, что прочтение является k-мером для некоторого значения k. Для начала будем предполагать идеальную и нереалистичную ситуацию, когда все прочтения поступают из одной и той же линии, не имеют ошибок и имеют полный охват, поэтому каждая подстрока k-

мера генома генерируется как прочитанная. Позже будет показано, как ослабить эти предположения для более реалистичных наборов данных.

Теперь можно определить вычислительную проблему моделирования сборки генома. Для заданной строки *Text* ее набор *k*-меров $Composition_k(Text)$ представляет собой совокупность всех *k*-меров *Text* (включая повторяющиеся). Например,

$$Composition_3(\text{TATGGGGTGC}) = \{\text{ATG, GGG, GGG, GGT, GTG, TAT, TGC, TGG}\}.$$

Обратите внимание, что *k*-меры перечислены в лексикографическом порядке (т.е. как они появились бы в словаре), а не в порядке их появления в TATGGGGTGC, потому что правильное упорядочение сгенерированных прочтений неизвестно.

Задача 1.1. Создать набор *k*-меров для заданной строки.

Вход: строка *Text* и целое число *k*.

Выход: $Composition_k(Text)$, где *k*-меры расположены в лексикографическом порядке.

Пример входа:

5

CAATCCAAC

Пример выхода:

CAATC

AATCC

ATCCA

TCCAA

CCAAC

Решение задачи создания композиции – это простое упражнение, но для моделирования сборки генома необходимо решить обратную задачу.

Задача. Восстановить строку из ее композиции *k*-меров.

Вход: целое число *k* и набор *k*-меров *Pattern*.

Выход: Строка *Text*, с композицией *k*-меров, эквивалентной набору *Pattern* (если такая строка существует).

Рассмотрим следующий пример композиции 3-меров:

```
AAT    ATG    GTT    TAA    TGT
```

Наиболее естественным способом решения проблемы восстановления строк является «соединение» пары *k*-меров, в том случае, если они перекрываются в *k-1* символах. В приведенном выше примере можно видеть, что строка должна начинаться с TAA, потому что нет 3-мера, оканчивающегося на TA. Это означает, что следующий 3-мер в строке должен начинаться с AA. Существует только один 3-мер, удовлетворяющий этому условию, AAT:

```
TAA
 AAT
```

В свою очередь, AAT может быть расширен только ATG, который может быть расширен только TGT и т.д., что приводит к восстановлению TAATGTT:

```
TAA
 AAT
  ATG
   TGT
    GTT
TAATGTT
```

Рассмотрим дополнительную композицию 3-меров:

AAT ATG ATG ATG CAT CCA GAT GCC GGA GGG GTT TAA TGC TGG TGT

Если снова начать с TAA, то следующий 3-мер в строке должен начинаться с AA, и есть только один такой 3-мер, AAT. В свою очередь, AAT может быть расширен только ATG.

```
TAA
 AAT
  ATG
TAATG
```

ATG может быть расширен либо TGC, либо TGG, либо TGT. Нужно решить, какой из этих 3-меров выбрать. Выберем TGT:

```
TAA
 AAT
  ATG
   TGT
TAATGT
```

После TGT единственный выбор – GTT:

```
TAA
 AAT
  ATG
   TGT
```

```
    GTT
TAATGTT
```

К сожалению, на GTT придется остановиться, потому что ни один из 3-меров не начинается с TT. Можно было бы попытаться расширить TAA влево, но ни один 3-мер в композиции не заканчивается на TA.

Если как хороший шахматист думать на несколько шагов вперед, то не нужно расширять ATG TGT до конца генома. Учитывая эту мысль, сделаем шаг назад, расширив ATG TGC:

```
TAA
 AAT
  ATG
   TGC
TAATGC
```

Продолжая процесс, получаем следующую сборку:

```
TAA
 AAT
  ATG
   TGC
    GCC
     CCA
      CAT
       ATG
        TGG
         GGA
          GAT
           ATG
            TGT
             GTT
TAATGCCATGGATGTT
```

Однако эта сборка неверна, потому что для составления строки было использовано только 14 из 15 3-меров (пропущен GGG), что делает восстановленный геном на один нуклеотид короче.

Трудность в сборке этого смоделированного генома возникает из-за того, что ATG повторяется три раза в композиции 3-меров, что заставляет рассматривать три варианта TGG, TGC и TGT, с помощью которых можно расширить ATG. Повторные подстроки в геноме не являются серьезной проблемой, когда имеется всего 15 прочтений, но с миллионами прочтений при наличии повторов сложнее «смотреть в будущее» и строить правильную сборку.

Если следовать идее об обнаружении источника репликации в бактериальных геномах, уже известно, насколько маловероятны многократные повторения

нуклеотидов в случайно генерируемой последовательности. Вы также знаете, что реальные геномы - это не случайные. Действительно, приблизительно половина человеческого генома составлена из повторов, например, последовательность Alu длиной в 300 нуклеотидов повторяется более миллиона раз, причем каждый раз вставляются / удаляются / замещаются только несколько нуклеотидов.

Для дальнейшей работы стоит напомнить некоторые базовые сведения из теории графов.

На рисунке 1.3 показана шахматная доска размером 4×4 с удаленными четырьмя угловыми квадратами. Рыцарь может двигаться на два шага в любом из четырех направлений (налево, направо, вверх и вниз), за которым следует один шаг в перпендикулярном направлении. Например, рыцарь на квадрате 1 может двигаться до квадрата 7 (два вниз и один налево), квадрат 9 (два вниз и один направо) или квадрат 6 (два направо и один вниз).

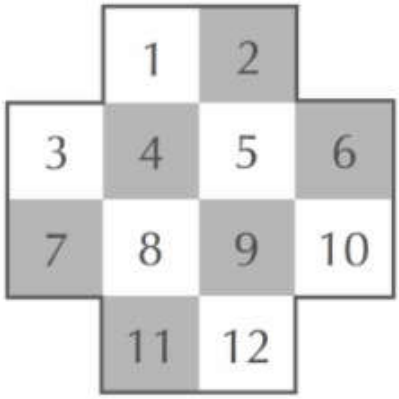


Рис. 1.3. Шахматная доска размером 4×4 с удаленными четырьмя угловыми квадратами.

Далее представим каждый из 12 квадратов на шахматной доске как узел. Два узла соединены ребром, если рыцарь может перемещаться между ними за один шаг. Например, узел 1 соединен с узлами 6, 7 и 9. Соединение узлов таким образом создает «Рыцарский граф», состоящий из 12 узлов и 16 ребер.

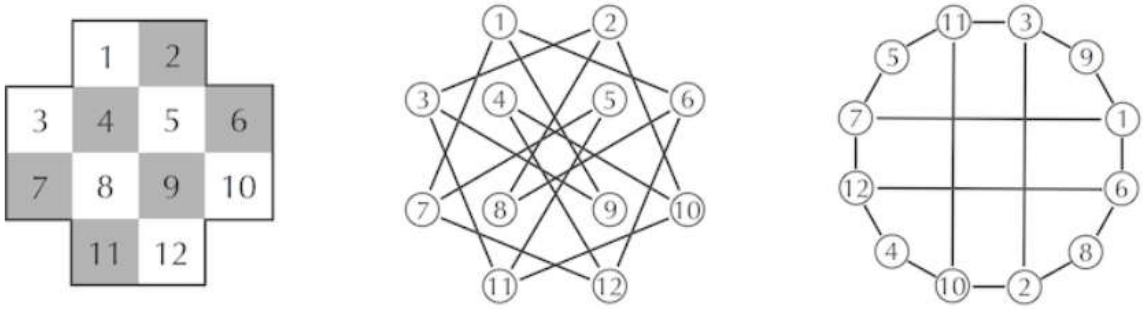


Рис. 1.4. (слева) Гипотетическая шахматная доска с квадратами. (В середине) Рыцарский граф представляет каждый квадрат узлом и соединяет два узла ребром, если

рыцарь может перемещаться между соответствующими квадратами всего за один ход. (Справа) Эквивалентное представление Рыцарского графа.

Можно описать граф по его набору узлов и ребер, где каждое ребро записывается как пара узлов, с которыми он соединяется. Граф в середине рисунка 92 описывается набором узлов:

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

и набором ребер:

```
{1,6}  {1,7}  {1,9} {2,3}  {2,8} {2,10} {3,9}  {3,11}
{4,10} {4,12} {5,7} {5,11} {6,8} {6,12} {7,12} {10,11}
```

Способ рисования графа не имеет значения; два графа с одинаковыми узлами и наборами ребер эквивалентны, даже если конкретные изображения, представляющие граф, различны. Важно только то, какие узлы связаны, а какие нет. Поэтому граф справа на рисунке 1.4 идентичен графу в середине.

Путь в графе – это последовательность ребер, где каждое следующее ребро начинается в узле, где заканчивается предыдущее ребро. Например, путь 8→6→1→9 в Рыцарском графе начинается с узла 8, заканчивается в узле 9 и состоит из 3 ребер. Пути, которые начинаются и заканчиваются в одном и том же узле, называются циклами. Цикл 3→2→10→11→3 начинается и заканчивается в узле 3 и состоит из 4 ребер.

Узлы и ребра графа, показанные справа на рисунке 1.4, показывают цикл, который посещает каждый узел на Рыцарском графе один раз и описывает последовательность движений рыцаря, посещающего каждый квадрат ровно один раз.

Число ребер, инцидентных данному узлу v, называется степенью v. К примеру, узел 1 в Рыцарском графе имеет степень 3, а узел 5 имеет степень 2. Сумма степеней всех 12 узлов в этом случае равна 32 (8 узлов имеют степень 3, а 4 узла имеют степень 2), что вдвое больше ребер на графе.

Многие проблемы биоинформатики анализируют ориентированные графы, в которых каждое ребро направлено от одного узла к другому, как показано стрелками на рисунке 1.5. Каждый узел в ориентированном графе характеризуется полустепенью захода (количество входящих ребер) и полустепенью исхода (количество исходящих ребер).

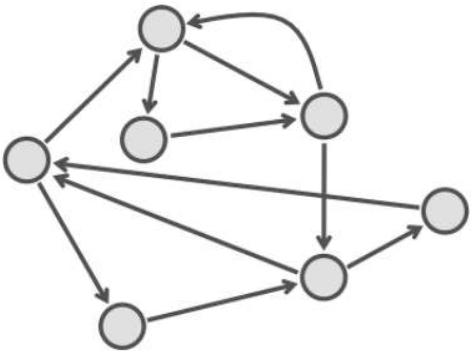


Рис. 1.5. Ориентированный граф.

Неориентированный граф является связным, если каждый из двух узлов имеет путь, соединяющий их. Несвязные графы могут быть разбиты на непересекающиеся компоненты связности.

Лекция 2. Граф де Брёйна.

Повторения в геноме требуют некоторого способа заглянуть вперед, чтобы увидеть правильную сборку заранее. Вернемся к приведенному ранее примеру, строка TAATGCCATGGGATGTT является решением задачи восстановления строки для сбора 15 3-меров, как показано ниже. Для каждого интервала строки между вхождениями ATG используется новый цвет.

```
TAA
 AAT
  ATG
   TGC
    GCC
     CCA
      CAT
       ATG
        TGG
         GGG
          GGA
           GAT
            ATG
             TGT
              GTT
TAATGCCATGGGATGTT
```

На рисунке 2.1 последовательные 3-меры в TAATGCCATGGGATGTT соединены вместе, чтобы сформировать путь генома этой строки.

Рис. 2.1. Пятнадцать цветных 3-меров, составляющих TAATGCCATGGGATGTT, соединяются в путь генома в соответствии с их порядком в геноме.

Задача 2.1. Восстановить строку из пути генома.

Вход: последовательность *k*-меров $Pattern_1$, ..., $Pattern_n$ такая, что последние *k-1* символов $Pattern_i$ равны первым *k-1* символам $Pattern_{i+1}$ при $1 \leq i \leq n-1$.

Выход: строка *Text* длины *k+n-1* такая, что *i*-й *k*-мер в *Text* равен $Pattern_i$ (для $1 \leq i \leq n$).

Пример входа:

ACCGA

CCGAA

CGAAG

GAAGC

AAGCT

Пример выхода:

ACCGAAGCT

Реконструировать геном из пути генома легко: по мере движения слева направо, каждый следующий 3-мер «TAATGCCATGGGATGTT» добавляет один новый символ в геном. К сожалению, для построения пути генома необходимо заранее знать геном.

Часто используются термины префикс и суффикс для обозначения первых *k-1* нуклеотидов и последних *k-1* нуклеотидов *k*-мера соответственно. Например, *Prefix*(TAA) = TA и *Suffix*(TAA) = AA. Заметим, что суффикс одного 3-мера в пути генома равен префиксу следующего 3-мера. Например, *Suffix*(TAA) = *Prefix*(AAT) = AA в пути генома для TAATGCCATGGGATGTT.

Это наблюдение предлагает метод построения пути генома строки из его композиции *k*-меров: можно использовать стрелку для соединения любого *k*-мера *Pattern* с *k*-мером *Pattern'*, если суффикс *Pattern* равен префиксу *Pattern'*.

Если строго следовать правилу соединения двух 3-меров стрелкой каждый раз, когда суффикс одного равен префиксу другого, то получится соединение всех последовательных 3-меров в TAATGCCATGGGATGTT. Однако, тот факт, что геном неизвестен заранее, вынуждает также соединять многие другие пары 3-

меров. Например, каждое из трех вхождений ATG должно быть соединено с TGC, TGG и TGT, как показано на рисунке 2.2.

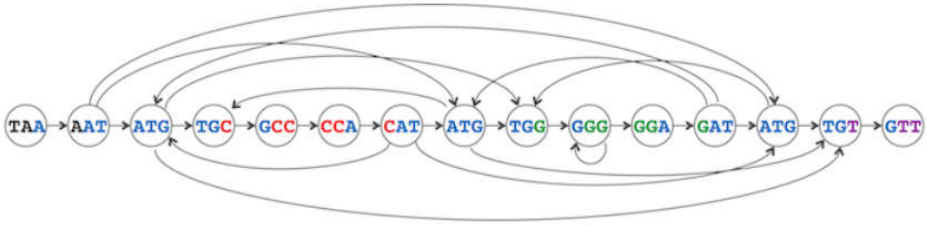


Рис. 2.2. Граф, показывающий все соединения между узлами, представляющими композицию 3-меров TAATGCCATGGGATGTT. Этот граф имеет 15 узлов и 25 ребер. Геном можно восстановить, пройдя по горизонтальным стрелкам от TAA до GTT.

На рисунке 2.2 представлен пример графа или сети узлов, связанных ребрами. Этот граф является примером ориентированного графа, ребра которого имеют направление и представлены стрелками (в отличие от неориентированных графов, ребра которых не имеют направлений).

Геном все еще можно проследить, следуя горизонтальному пути от TAA до GTT. Но в последовательности геномов неизвестно заранее, как правильно упорядочить прочтения. Поэтому предлагается располагать 3-меры лексикографически, что дает граф перекрытия, показанный на рисунке 2.3. Путь генома исчез.

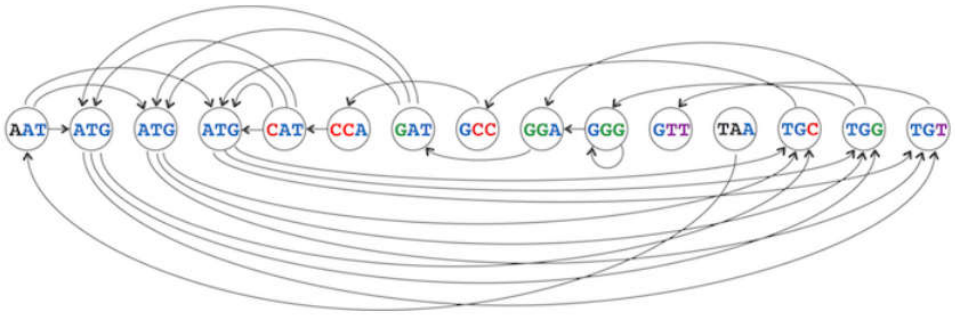


Рис. 2.3. Граф с рисунка 53, с лексикографически упорядоченными 3-мерами. Путь по графу, представляющий правильную сборку, увидеть более сложно.

Путь генома, возможно, исчез для обнаружения невооруженным глазом, но он все равно должен быть там, поскольку произошло только перемещение узлов графа.

Действительно, на рисунке 2.4 показан путь генома, описывающий TAATGCCATGGGATGTT. Несмотря на то, что найти такой путь в настоящее время так же сложно, как и собрать геном вручную, граф тем не менее дает хороший способ визуализации отношений перекрытия между прочтениями.

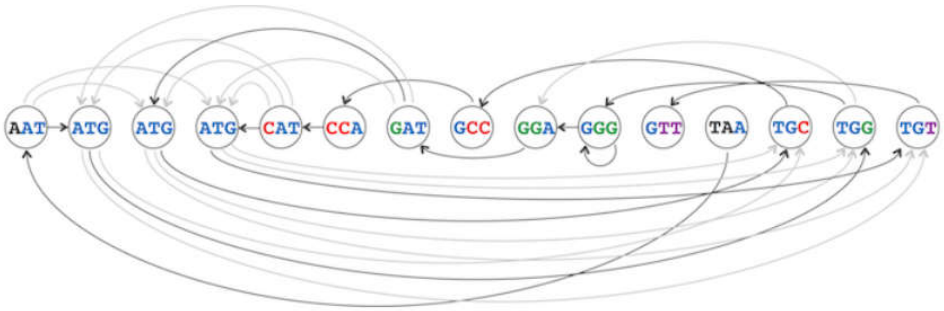
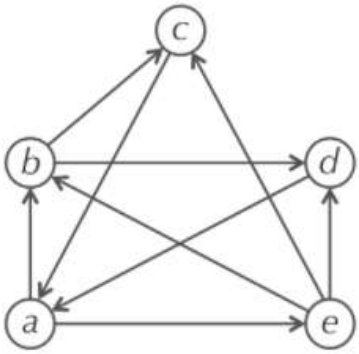


Рис. 2.4. Путь генома, определяющий TAATGCCATGGGATGTT, выделенный в графе перекрытий.

Чтобы обобщить построение приведенного выше графа на произвольный набор *k*-меров *Patterns*, формируем узел для каждого *k*-мера в *Patterns* и соединяем *k*-меры *Pattern* и *Pattern'* направленным ребром, если *Suffix*(*Pattern*) равен *Prefix*(*Pattern'*). Полученный граф называется графом перекрытий для этих *k*-меров и обозначается *Overlap*(*Patterns*).

Существует два стандартных способа представления графа. Для ориентированного графа с n узлами матрица смежности $(A_{i,\ j})$ размера $n \times n$ определяется следующим правилом: $A_{i,\ j}=1$, если направленное ребро соединяет узел i с узлом j и $A_{i,\ j}=0$ в противном случае. Другим (более эффективным по памяти) способом представления графа является использование списка смежности, для которого перечисляются все узлы, соединенные с данным узлом.



Adjacency matrix						Adjacency list
	a	*b*	*c*	*d*	*e*	
a	0	1	0	0	1	*a* is adjacent to *b* and *e*
b	0	0	1	1	0	*b* is adjacent to *c* and *d*
c	1	0	0	0	0	*c* is adjacent to *a*
d	1	0	0	0	0	*d* is adjacent to *a*
e	0	1	1	1	0	*e* is adjacent to *b*, *c*, and *d*

Рис. 2.5. Способы представления графа в виде матрицы смежности и списка связности.

Задача 2.2. Построить граф перекрытий для коллекции *k*-меров.

Вход: коллекция *k*-меров *Patterns*.

Выход: граф перекрытий *Overlap*(*Patterns*).

Пример входа:

ATGCG

GCATG

CATGC

AGGCA

GGCAT

Пример выхода:

AGGCA -> GGCAT

CATGC -> ATGCG

GCATG -> CATGC

GGCAT -> GCATG

Теперь известно, что для решения проблемы восстановления строк нужно искать путь в графе перекрытий, который посещает каждый узел ровно один раз. Такой путь называется гамильтоновым путем, названный в честь ирландского математика Уильяма Гамильтона. Как показано на рисунке 2.6, граф может иметь более одного гамильтонова пути.

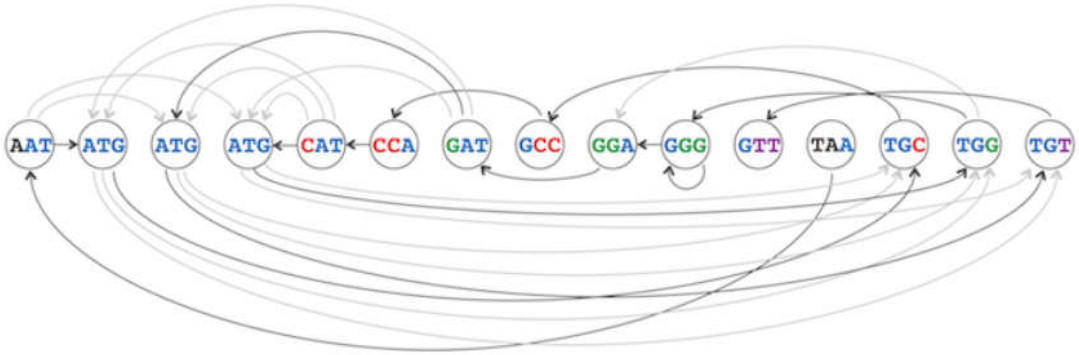


Рис. 2.6. В дополнение к гамильтонову пути, который восстанавливает TAATGCCATGGGATGTT, другой гамильтонов путь в графе перекрытий восстанавливает геном TAATGGGATGCCATGTT. Эти два генома отличаются положениями CC и GG, но имеют одинаковую композицию 3-меров.

Задача. Построить гамильтонов путь в графе.

Вход: ориентированный граф.

Выход: путь, который посещает каждый узел в графе ровно один раз (если такой путь существует).

Для того, чтобы решить проблему Гамильтонова пути, сначала нужно познакомиться с работами Николаса де Брёйна, голландского математика. В 1946 году де Брёйн заинтересовался в решении чисто теоретической проблемы, описанной ниже. Бинарная строка – это строка, состоящая только из 0 и 1; двоичная строка является *k*-универсальной, если она содержит каждый бинарный *k*-мер ровно один раз. Например, 0001110100 является 3-

универсальной строкой, так как она содержит каждый из восьми двоичных 3-меров (000, 001, 011, 111, 110, 101, 010 и 100) ровно один раз.

Поиск *k*-универсальной строки эквивалентен решению проблемы восстановления строки, когда композиция *k*-меров представляет собой совокупность всех двоичных *k*-меров. Таким образом, задача нахождения *k*-универсальной строки эквивалентна нахождению гамильтонова пути в графе перекрытий, сформированном на всех двоичных *k*-мерах (см. Рисунок 2.7). Хотя гамильтонов путь в данном случае легко найти вручную, де Брёйну было интересно построить *k*-универсальные строки для любых значений *k*. Например, чтобы найти 20-универсальную строку, нужно рассмотреть граф с более чем миллионом узлов. Абсолютно неясно, как найти гамильтонов путь в таком огромном графе, или даже проверить, существует ли такой путь.

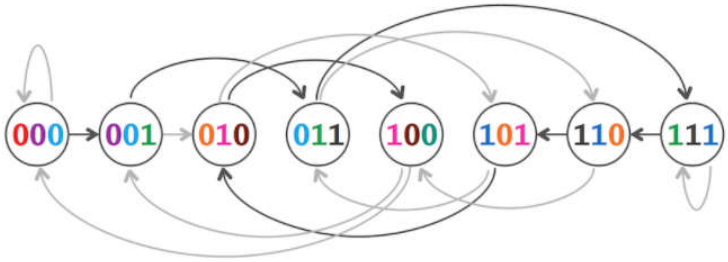


Рис. 2.7. Гамильтонов путь в графе перекрытий всех двоичных 3-меров.

Вместо поиска гамильтоновых путей на огромных графах де Брёйн разработал совершенно другой (и несколько неинтуитивный) способ представления композиции *k*-меров с использованием графа. Позже в этой главе будет показано, как он использовал этот метод для построения универсальных строк.

Представим геном TAATGCCATGGGATGTT как последовательность его 3-меров:

TAA AAT ATG TGC GCC CCA CAT ATG TGG GGG GGA GAT ATG TGT GTT

На этот раз вместо приписывания этих 3-меров узлам, назначим их ребрам, как показано на рисунке 2.8.

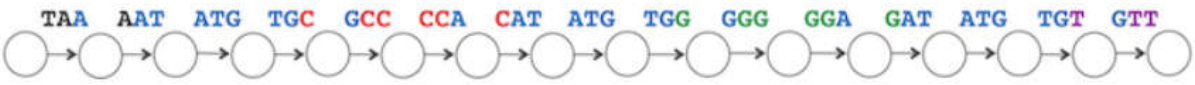


Рис. 2.8. Геном TAATGCCATGGGATGTT представлен как путь, с ребрами (а не узлами), помеченными 3-мерами.

Можно восстановить геном, следуя этому пути слева направо, добавляя на каждом шаге новый нуклеотид. Поскольку каждая пара последовательных ребер представляет собой последовательные 3-меры, которые перекрываются в двух нуклеотидах, можно маркировать каждый узел этого графа 2-мером, представляющим перекрывающиеся нуклеотиды по обе стороны узла.

Например, узел с входящим ребром CAT и исходящим ребром ATG помечен как AT.

Рис. 2.9. Геном TAATGCCATGGGATGTT, представленный как путь с ребрами (а не узлами), помеченными 3-мерами и узлами, помеченными 2-мерами.

Здесь нет ничего нового, пока не начать склеивать идентично помеченные узлы. На рисунке 2.10 происходит смещение трех узлов AT ближе друг к другу, пока они не склеятся в один узел.

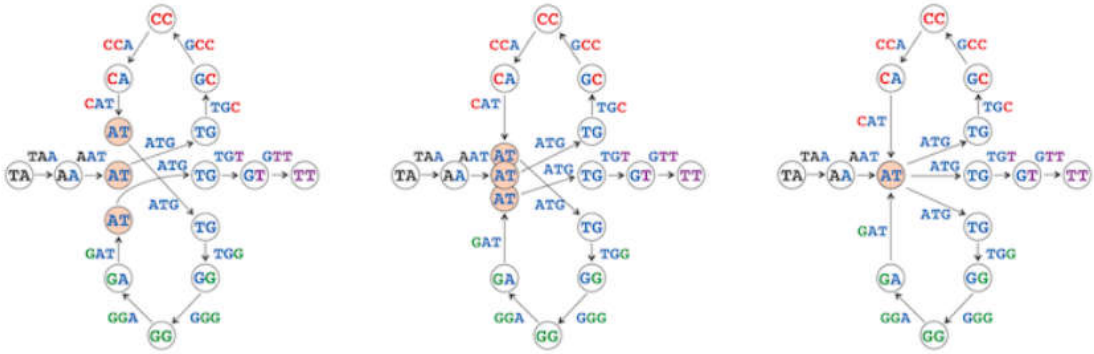


Рис. 2.10. Смещение трех узлов, помеченных AT ближе (слева) и ближе (в середине) друг к другу, чтобы в конечном итоге склеить их в один узел (справа). Путь с 16 узлами преобразован в граф с 14 узлами.

Есть также три узла, помеченные TG, которые также можно приблизить друг к другу (рисунок 2.11), пока они не склеятся в один узел.

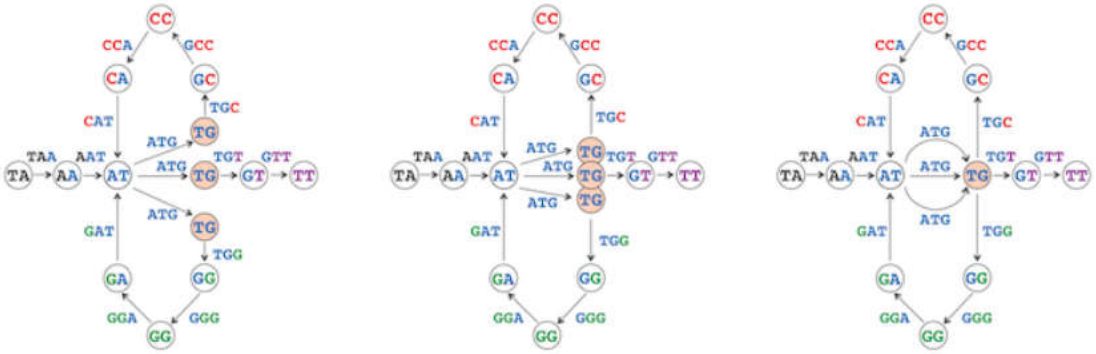


Рис. 2.11. Смещение трех узлов, помеченных GT ближе (слева) и ближе (в середине) друг к другу, чтобы в конечном итоге склеить их в один узел (справа). Путь с 16 узлами преобразован в граф с 12 узлами.

Наконец, склеим два узла, помеченных GG (GG и GG), который создает специальный тип ребра, называемый петлей, соединяющим GG с самим собой. Число узлов в полученном графике (справа на рисунке 2.12) сократилось с 16 до 11, а количество ребер оставалось неизменным. Этот граф называется

графом де Брёйна строки TAATGCCATGGGATGTT, обозначаемым $DeBruijn_3$(TAATGCCATGGGATGTT). Стоит обратить внимание, что приведенный ниже граф де Брёйна имеет три разных ребра, соединяющих AT с TG (представляющие три копии повторного ATG).

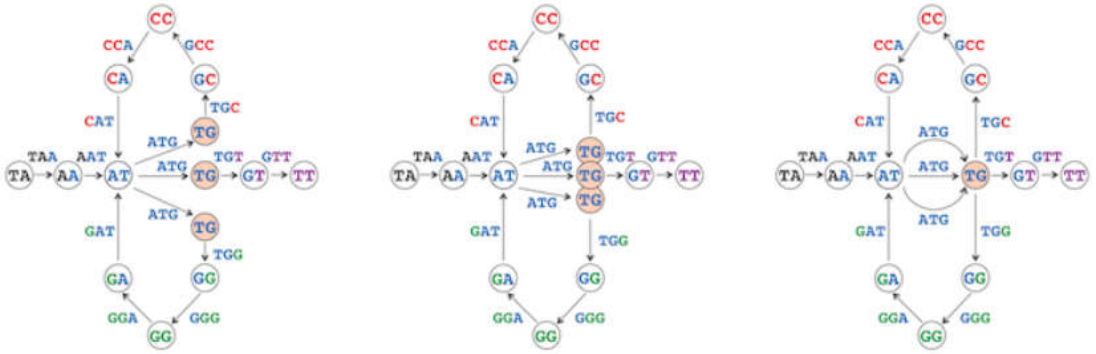


Рис. 2.12. Смещение двух узлов, помеченных GG ближе (слева) и ближе (в середине) друг к другу, чтобы в конечном итоге склеить их в один узел (справа). Путь с 16 узлами преобразован в граф $DeBruijn_3$(TAATGCCATGGGATGTT) с 11 узлами.

С учетом генома *Text*, $PathGraph_k(Text)$ – это путь, состоящий из $|Text| - k + 1$ ребра, где *i*-е ребро этого пути помечено *i*-м *k*-мером в *Text*, а *i*-й узел пути помечен *i*-м (*k-1*)-мером в *Text*. Граф де Брёйна $DeBruijn_k(Text)$ формируется путем склеивания идентично помеченных узлов в $PathGraph_k(Text)$.

Задача 2.3. Построить граф де Брёйна для строки.

Вход: целое число *k* и строка *Text*.

Выход: $DeBruijn_k(Text)$.

Пример входа:

4

AAGATTCTCTAAGA

Пример выхода:

AAG -> AGA,AGA

AGA -> GAT

ATT -> TTC

CTA -> TAA

CTC -> TCT

GAT -> ATT

TAA -> AAG

TCT -> CTA,CTC

TTC -> TCT

Несмотря на то, что нужно склеивать узлы для формирования графа де Брёйна, его ребра не изменяются, поэтому путь от TA до TT, восстанавливающий геном, все еще находится в $DeBruijn_3$(TAATGCCATGGGATGTT) (см. Рисунок 2.13), хотя этот путь стал «запутанным» после склеивания. Решение проблемы восстановления строки сводится к нахождению пути в графе де Брёйна, который проходит по каждому ребру ровно один раз. Такой путь называется Эйлеровым путем в честь математика Леонарда Эйлера.

Задача. Построить эйлеров путь в графе.

Вход: ориентированный граф.

Выход: путь, который посещает каждое ребро в графе ровно один раз (если он существует).

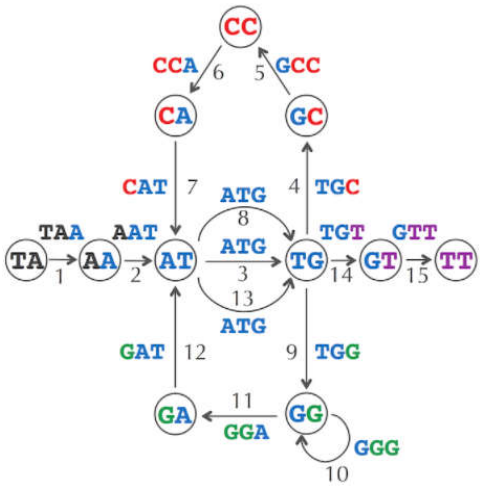


Рис. 2.13. Путь от TA до TT, описывающий геном TAATGCCATGGGATGTT, стал «запутанным» на графе де Брёйна. Нумерация 15 ребер пути указывает на эйлеров путь, восстанавливающий геном.

Теперь есть альтернативный способ решения проблемы восстановления строки, которая сводится к нахождению эйлерова пути на графе де Брёйна. Но чтобы построить граф де Брёйна в геноме, требуется склеивать узлы $PathGraph_k(Text)$. Однако, построение этого графа требует знания правильного упорядочения k-меров в $Text$.

На рисунке 2.14 представлена коллекция 3-меров TAATGCCATGGGATGTT как граф $CompositionGraph_3$(TAATGCCATGGGATGTT). Как и в графе де Брёйна, каждый 3-мер приписывается направленному ребру с префиксом, обозначающим первый узел ребра, и суффиксом, обозначающим второй узел ребра. Тем не менее, ребра этого графа изолированы, а это означает, что никакие два ребра не имеют общего узла.

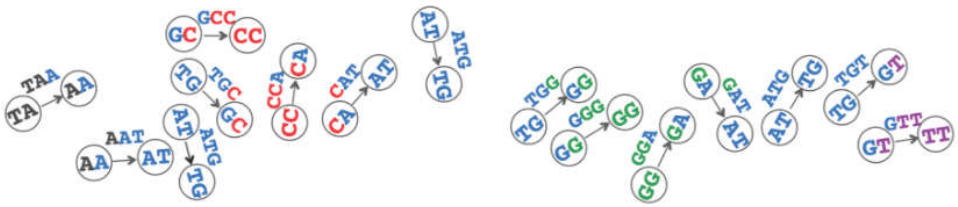


Рис. 2.14. *CompositionGraph*$_3$(TAATGCCATGGGATGTT) представляет собой композицию 3-меров TAATGCCATGGGATGTT с использованием изолированного направленного ребра для каждого 3-мерного.

На рисунке 2.15 показано, как *CompositionGraph*$_3$(TAATGCCATGGGATGTT) изменяется после склеивания узлов с одной и той же меткой. Пятнадцать изолированных ребер в *CompositionGraph*$_3$(TAATGCCATGGGATGTT) складываются в путь *PathGraph*$_3$(TAATGCCATGGGATGTT). Последующие операции склеивания выполняются так же, как при склеивании узлов *PathGraph*$_3$(TAATGCCATGGGATGTT), в результате чего получается граф *DeBruijn*$_3$(TAATGCCATGGGATGTT). Таким образом, можно построить граф де Брёйна из композиции 3-меров этого генома, не зная генома.

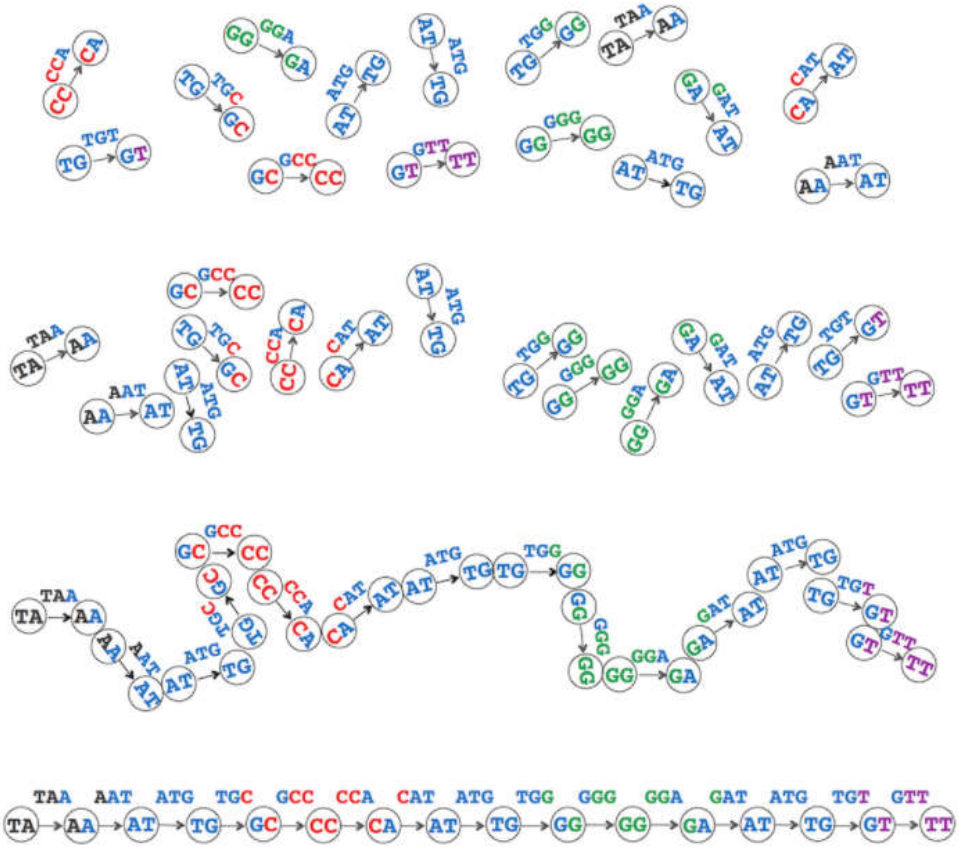


Рис. 2.15. Преобразование *CompositionGraph*$_3$(TAATGCCATGGGATGTT) в *DeBruijn*$_3$(TAATGCCATGGGATGTT).

Для произвольной строки *Text* определим *CompositionGraph*$_k$(*Text*) как граф, состоящий из |*Text*| - *k* + *1* изолированных ребер, где ребра помечены *k*-мерами

в *Text*; каждое ребро, помеченное *k*-мером, соединяет узлы, помеченные префиксом и суффиксом этого *k*-мера. Граф $CompositionGraph_k(Text)$ – это совокупность изолированных ребер, представляющих *k*-меры в композиции *k*-меров *Text*. Это означает, что можно построить *CompositionGraph*(*Text*) из композиции *k*-меров *Text*. Склеивание узлов с одинаковой меткой в $CompositionGraph_k(Text)$ создает $DeBruijn_3(Text)$. Таким образом, можно построить граф де Брёйна генома, не зная геном.

Для данной коллекции *k*-меров *Patterns* (где некоторые *k*-меры могут появляться несколько раз), определим *CompositionGraph*(*Patterns*) как граф с |*Patterns*| изолированными ребрами. Каждое ребро помечено *k*-мером из *Patterns*, а начальный и конечный узлы ребра помечены префиксом и суффиксом метки *k*-мера этого ребра. Затем определим *DeBruijn*(*Patterns*) путем склеивания идентично помеченных узлов в *CompositionGraph*(*Patterns*), что дает следующий алгоритм.

```
DeBruijn(Patterns)
    represent every k-mer in Patterns as an isolated edge between its prefix and suffix
    glue all nodes with identical labels, yielding the graph DeBruijn(Patterns)
    return DeBruijn(Patterns)
```

Построение графа де Брёйна путем склеивания тождественно помеченных узлов поможет позже, когда введется обобщенное понятие графа де Брёйна для других приложений. Опишем еще один способ построения графов де Брёйна без склеивания.

Для заданной коллекции *k*-меров *Patterns*, узлы $DeBruijn_k(Patterns)$ – это все уникальные (*k-1*)-меры, которые являются префиксами или суффиксами в *Patterns*. Например, дана следующая коллекция 3-меров:

AAT ATG ATG ATG CAT CCA GAT GCC GGA GGG GTT TAA TGC TGG TGT

Тогда набор из одиннадцати уникальных 2-меров, встречающихся в качестве префикса или суффикса 3-меров в этой коллекции, выглядит следующим образом:

AA AT CA CC GA GC GG GT TA TG TT

Для каждого *k*-мера в *Patterns* соединим его префиксный узел с его суффиксным узлом направленным ребром для создания *DeBruijn*(*Patterns*). Этот процесс создает тот же граф де Брёйна, который был раньше (рисунок 2.16).

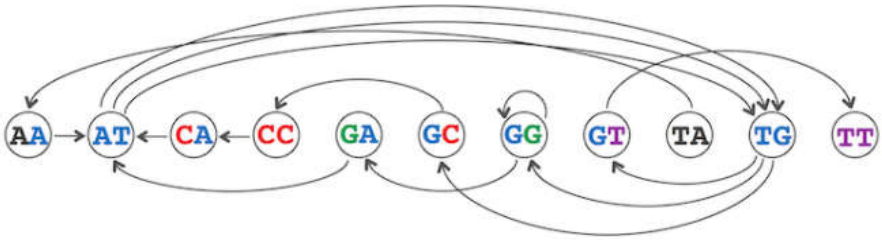


Рис. 2.16. Граф $DeBruijn_3$(TAATGCCATGGGATGTT).

Задача 2.4. Построить граф де Брёйна для коллекции *k*-меров.

Вход: коллекция *k*-меров *Patterns*.

Выход: Список смежности графа *DeBruijn*(*Patterns*).

Пример входа:

GAGG

CAGG

GGGG

GGGA

CAGG

AGGG

GGAG

Пример выхода:

AGG -> GGG

CAG -> AGG,AGG

GAG -> AGG

GGA -> GAG

GGG -> GGA,GGG

Теперь есть два пути решения проблемы восстановления строки. Можно либо найти гамильтонову траекторию в графе перекрытий (рисунок 2.17, сверху), либо найти эйлеров путь в графе де Брёйна (рисунок 2.17, внизу).

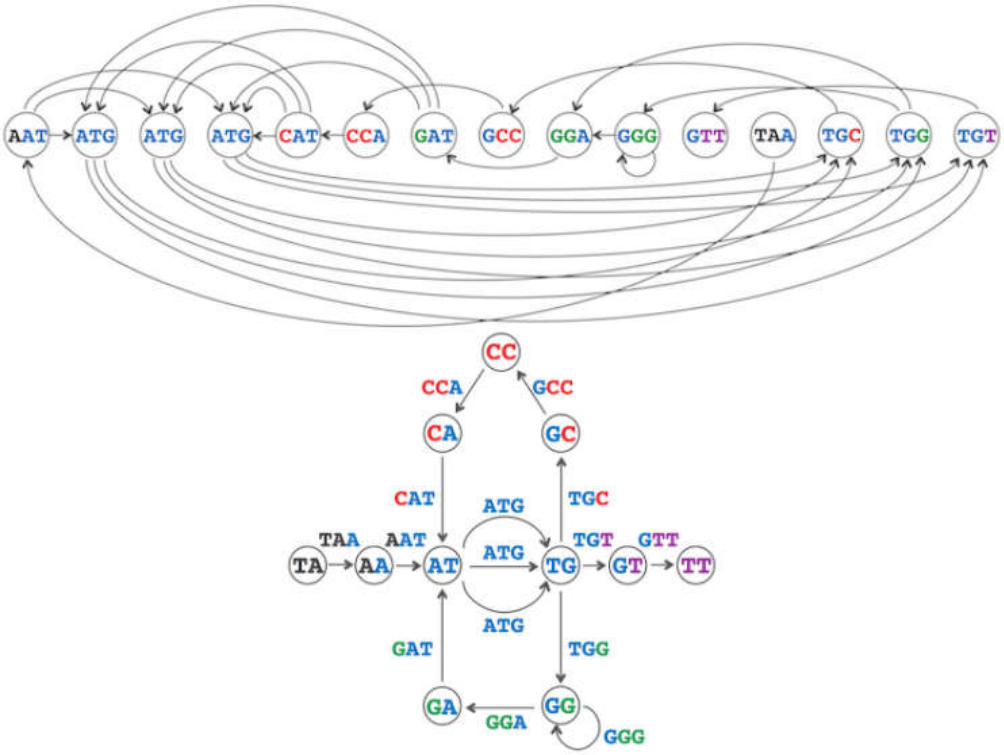


Рис. 2.17. Граф перекрытия и граф де Брёйна для генома TAATGCCATGGGATGTT.

Лекция 3. Эйлеров путь. Эйлеров цикл.

Вернемся в 1735 год и прусский город Кенигсберг. Этот город, который сегодня является Калининградом, Россия, включает в себя берега реки Прегель и два речных острова; семь мостов соединяли эти четыре разные части города, как показано на рисунке 3.1. Жители Кенигсберга наслаждались прогулками, и задали естественный вопрос: можно ли выйти из дома, пересечь каждый мост ровно один раз и вернуться домой. Их вопрос стал известен как Задача о семи кёнигсбергских мостах.

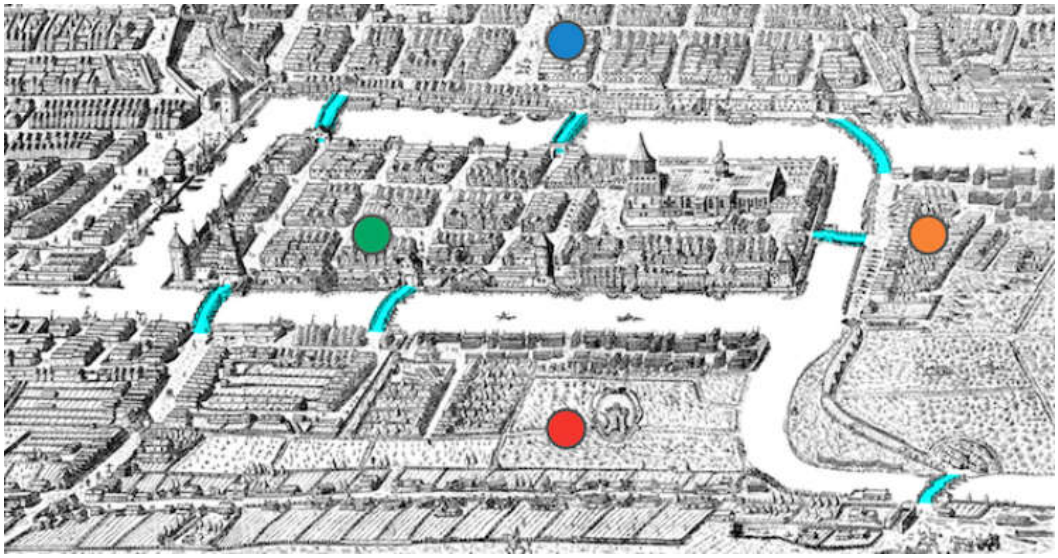


Рис. 3.1. Карта Кенигсберга, адаптированная из иллюстрации 1613 года Йоахима Беринга. Город состоял из четырех районов, представленных голубой, красной, желтой и зеленой точками. Семь мостов, соединяющих разные части города, для наглядности выделены голубым цветом.

В 1735 году Леонард Эйлер нарисовал граф, показанный на рисунке 3.2, который называют Кенигсбергом; узлы этого графа представляют четыре района города, а его ребра представляют собой семь мостов, соединяющих разные районы. Обратите внимание, что ребра Кенигсберга не ориентированы, что означает, что они могут быть пройдены в любом направлении.

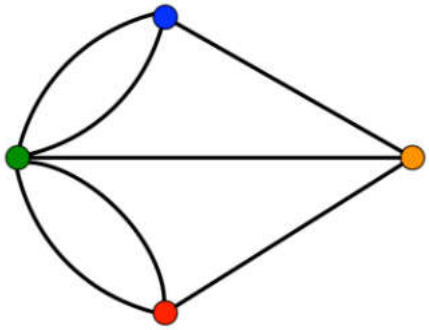


Рис. 3.2. Граф Кенигсберг.

Ранее эйлеров путь был определен как путь в графе, проходящий по каждому ребру графа ровно один раз. Цикл, который проходит по каждому ребру графа ровно один раз, называется эйлеровым циклом, говорят, что граф, содержащий такой цикл, является эйлеровым. Обратите внимание, что эйлеров цикл в Кенигсберге немедленно предоставит жителям города прогулку, которую они хотели. Теперь можно переопределить Задачу о семи кёнигсбергских мостах как пример следующей более общей задачи.

Задача. Построить эйлеров цикл в графе.

Вход: граф.

Выход: эйлеров цикл на этом графе, если таковой существует.

Эйлер решил Задачу о семи кёнигсбергских мостах, показав, что ни одна прогулка не может пересекать каждый мост ровно один раз (т.е. граф Кенигсберг не является эйлеровым). Тем не менее его реальный вклад и причина, почему он рассматривается как основоположник математической теории графов, состоит в том, что он доказал теорему, определяющую, когда граф будет иметь эйлеров цикл. Его теорема подразумевает эффективный алгоритм построения эйлерова цикла на любом эйлеровом графе, даже на том, который имеет миллионы ребер. Более того, этот алгоритм можно легко распространить на алгоритм построения эйлерова пути (в графе, имеющем такой путь), что позволит решить задачу восстановления строки, используя граф де Брёйна.

С другой стороны, оказывается, что не найден эффективный алгоритм, решающий проблему Гамильтонова пути. Поиск такого алгоритма или доказательство того, что эффективный алгоритм не существует для этой проблемы, лежит в основе одного из фундаментальных неотвеченных вопросов в информатике. Ученые в области компьютерных классифицируют алгоритм как полиномиальный, если его время работы может быть ограничено полиномом от длины входных данных. С другой стороны, алгоритм является экспоненциальным, если его время выполнения на некоторых наборах данных является экспоненциальным от длины входных данных. Хотя алгоритм Эйлера является полиномиальным, проблема Гамильтонова пути принадлежит специальному классу задач, для которых все попытки разработать полиномиальный алгоритм потерпели неудачу.

Можно задаться вопросом, существует ли простой результат, приводящий к быстрому алгоритму для задачи Гамильтонова цикла. Основная задача состоит в том, что, хоть обычно руководствуются теоремой Эйлера при решении задачи Эйлерова цикла, аналогичное простое условие для задачи Гамильтонова цикла остается неизвестным. Всегда можно исследовать все проходы по графу полным перебором и остановиться, когда будет найден гамильтонов цикл. Проблема с этим методом полного перебора заключается в том, что для графа из тысяч узлов может быть больше проходов, чем атомов во вселенной.

В течение многих лет проблема гамильтонова цикла ускользала от всех попыток решить ее. После долгих лет бесплодных усилий ученые в области компьютерных наук стали задаваться вопросом, не является ли эта проблема неразрешимой, т.е. что их неспособность найти полиномиальный алгоритм объясняется не отсутствием понимания, а тем, что такой алгоритм решения проблемы гамильтонова цикла не существует. В 1970-х годах ученые обнаружили еще тысячи алгоритмических задач с той же судьбой, что и

проблема гамильтонова цикла. Несмотря на то, что они могут показаться простыми, никто не смог найти быстрые алгоритмы для их решения. Большое подмножество этих проблем, в том числе проблема гамильтонова цикла, называется NP-полными задачами.

Все NP-полные задачи эквивалентны друг другу: любая NP-полная задача может быть преобразована в любую другую NP-полную задачу за полиномиальное время. Таким образом, если найти быстрый алгоритм для одной NP-полной задачи, автоматически можно будет использовать этот алгоритм для разработки быстрого алгоритма для любой другой NP-полной задачи. Проблема эффективного решения NP-полных задач или доказательство того, что они неразрешимы, настолько фундаментальна, что она была названа одной из семи «Проблем тысячелетия» Института математики Клэя в 2000 году. Если найти эффективный алгоритм для любой NP-полной задачи или показать, что одна из этих задач неразрешима, то Институт Клэя присудит приз в размере миллиона долларов.

До сих пор была решена только одна из семи проблем тысячелетия; в 2003 году Григорий Перельман доказал гипотезу Пуанкаре. Будучи истинным математиком, Перельман отказался от премии в миллион долларов, полагая, что чистота и красота математики – это выше любого приза.

Несмотря на то, что NP-полные проблемы наиболее привлекали внимание, существует множество различных классов вычислительных задач. Например, проблема P является NP-трудной, если существует некоторая NP-полная проблема, которая может быть сведена к P за полиномиальное время. Поскольку NP-полная проблема может быть приведена к другой такой же за полиномиальное время, NP-трудные проблемы включают в себя все NP-полные проблемы, и, как следствие, любая NP-полная проблема может быть сведена к любой NP-трудной задаче за полиномиальное время. Однако, не всякая NP-трудная проблема является NP-полной (что означает, что первые «как минимум также трудны» для решения, как вторые). Одним из примеров NP-трудной проблемы, которая не является NP-полной, является проблема коммивояжера, в которой дан граф с взвешенными ребрами, и требуется создать гамильтонов цикл с минимальным общим весом.

Далее сосредоточимся на подходе построения графа де Брёйна к сборке генома.

В течение первых двух десятилетий после изобретения методов секвенирования ДНК, биологи собирали геномы с использованием графов перекрытия, поскольку они не понимали, что Задача о семи кёнигсбергских мостах являлась ключом к сборке ДНК. Действительно, графы перекрытия использовались для сборки генома человека. Биоинформатикам понадобилось некоторое время, чтобы понять, что граф де Брёйна, первоначально

построенный для решения строго теоретической проблемы, имел отношение к сборке генома. Более того, когда граф де Брёйна дошел до биоинформатики, он считался экзотической математической концепцией с ограниченным практическим применением. Сегодня граф де Брёйна стал доминирующим подходом для сборки генома.

Рассмотрим метод Эйлера для решения задачи поиска Эйлерова цикла. Эйлер работал с неориентированными графами, такими как Кенигсберг, однако, рассмотрим аналог его алгоритма для ориентированных графов, чтобы его метод применить к сборке генома.

Рассмотрим муравья Лео, который идет по ребрам эйлерова цикла. Каждый раз, когда Лео входит в узел этого графа по ребру, он может покинуть этот узел по второму, не использованному ранее ребру. Таким образом, для того, чтобы граф был эйлеровым, количество входящих ребер в любом узле должно быть равно количеству исходящих ребер в этом узле. Определим полустепени захода и исхода узла v (обозначим соответственно $in(v)$ и $out(v)$) как число ребер, входящих в v и исходящих из v. Узел v сбалансирован, если $in(v)=out(v)$, а граф сбалансирован, если все его узлы сбалансированы. Поскольку Лео всегда должен иметь возможность покинуть узел по неиспользованному ребру, любой эйлеров граф должен быть сбалансирован. На рисунке 3.3 показаны сбалансированный и несбалансированный граф.

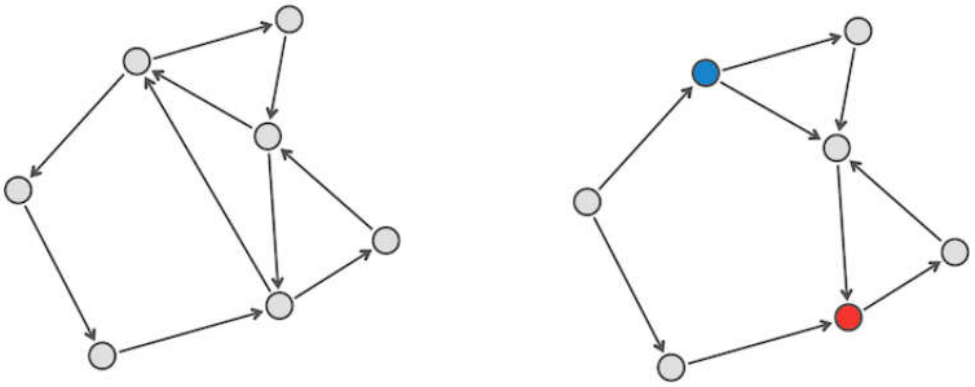


Рис. 3.3. Сбалансированный (левый) и несбалансированный (правый) ориентированные графы. Для (несбалансированного) синего узла v, $in(v) = 1$ и $out(v) = 2$, тогда как для (несбалансированного) красного узла w $in(w) = 2$ и $out(w) = 1$.

Граф на рисунке 3.4 сбалансирован, но не является эйлеровым, потому что он несвязный, что означает, что в некоторые узлы нельзя попасть из других узлов. На любом несвязном графе невозможно найти эйлеров цикл. Напротив, говорят, что ориентированный граф сильно связан, если в любой узел можно попасть из любого другого узла. Таким образом, эйлеровы графы должны быть сбалансированными и сильно связными. Теорема Эйлера утверждает, что эти два условия достаточны, чтобы гарантировать, что произвольный граф

является эйлеровым. В результате это означает, что можно определить, является ли граф эйлеровым, не находя при этом никаких циклов.

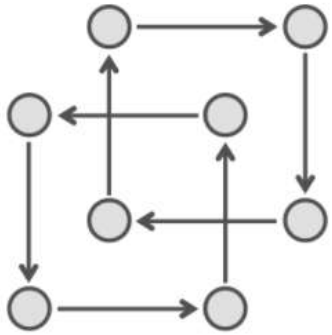


Рис. 3.4. Сбалансированный несвязный граф.

Теорема Эйлера: любой сбалансированный, сильно связный ориентированный граф является эйлеровым.

Докажем теорему Эйлера, предположив, что имеется произвольный сбалансированный и сильно связный ориентированный граф и показав, что этот граф должен иметь эйлеров цикл.

Пусть *Graph* – произвольный сбалансированный и сильно связный ориентированный граф. Чтобы доказать, что *Graph* имеет эйлеров цикл, поместим Лео в любой узел v_0 графа (зеленый узел на рисунке 3.5) и пусть он случайно перемещается по графу при условии, что он не может пройти по одному и тому же ребру дважды.

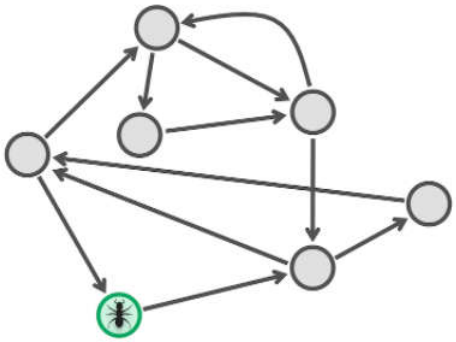


Рис. 3.5. Лео начинает с зеленого узла v_0 и проходит через сбалансированный и сильно связный граф.

Если Лео невероятно удачлив, то он будет проходить по каждому ребру ровно один раз и вернётся к v_0. Однако вероятность того, что Лео «застрянет» где-нибудь, прежде чем он сможет закончить цикл Эйлера, означает, что он достигает узла и не находит неиспользованных ранее ребер, выходящих их этого узла.

Единственным узлом, где Лео может «застрять», является исходный узел v_0. Причина в том, что граф сбалансирован – если Лео входит в любой узел,

отличный от v_0 (через входящее ребро), тогда он всегда сможет выйти через неиспользуемое исходящее ребро. Единственным исключением из этого правила является начальный узел v_0, так как Лео использовал одно из выходящих ребер v_0 при первом перемещении. Теперь, поскольку Лео вернулся к v_0, результатом его прогулки был цикл, который назовем $Cycle_0$ (см. Рисунок 3.6).

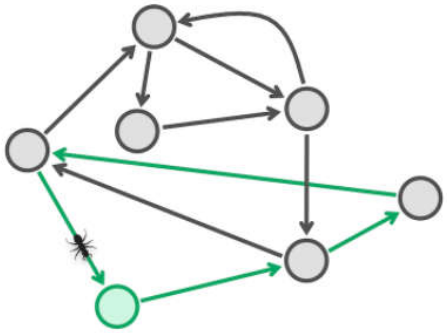


Рис. 3.6. Лео создает цикл $Cycle_0$ (образованный зелеными ребрами), когда он «застревает» в зеленом узле v_0. В этом случае он еще не посетил все ребра графа.

Если $Cycle_0$ является эйлеровым, проход закончен. В противном случае, поскольку *Graph* сильно связан, некоторый узел в $Cycle_0$ должен иметь неиспользованные входящие и исходящие ребра. Назовем этот узел v_1, пусть теперь Лео начнет с v_1 вместо v_0 и пройдет по $Cycle_0$ (таким образом, вернется к v_1), как показано на рисунке 3.7.

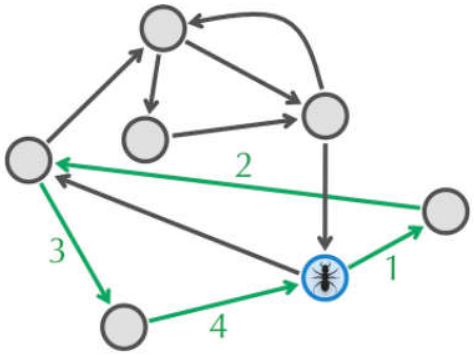


Рис. 3.7. Начиная с нового узла v_1 (показан синим цветом), Лео перемещается по $Cycle_0$, возвращаясь к v_1. Обратите внимание, что теперь v_1, в отличие от v_0, имеет неиспользованные исходящие и входящие ребра.

Лео, как и прежде, в конце концов застрянет в v_1, в том месте, где он начал. Однако, теперь у этого узла появляются неиспользованные ребра, и поэтому он может продолжать идти от v_1, каждый раз используя новое ребро. Тот же аргумент, что и раньше, подразумевает, что Лео должен в конечном итоге «застрять» в v_1. Результатом прогулки Лео является новый цикл, $Cycle_1$ (см. Рисунок 3.8), который больше, чем $Cycle_0$.

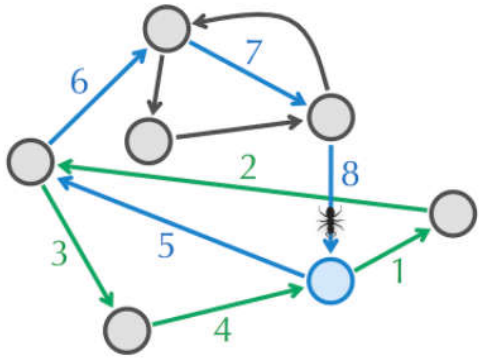


Рис. 3.8. После прохождения ранее построенного зеленого цикла $Cycle_0$, Лео продолжает идти и, в конечном счете, проходит по более крупному циклу $Cycle_1$, образованному как зеленым, так и синим циклами, объединенными в один.

Если $Cycle_1$ является эйлеровым циклом, то Лео выполнил свою работу. В противном случае выбираем узел v_2 в $Cycle_1$, который имеет неиспользованные входящие и исходящие ребра (красный узел на рисунке 3.9). Начиная с v_2, Лео проходит по $Cycle_1$, вернувшись к v_2, как показано справа на рисунке 3.9. Впоследствии он будет беспорядочно ходить, пока не «застрянет» в v_2, создав еще больший цикл, который назовем $Cycle_2$.

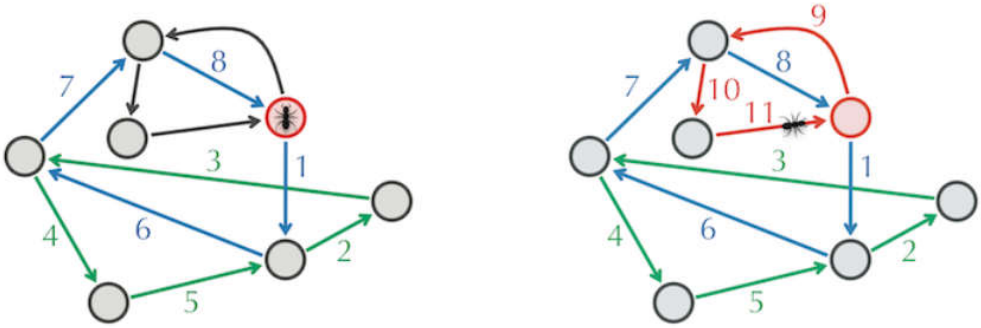


Рис. 3.9. (слева) Начиная с нового узла v_2 (показан красным), Лео сначала перемещается вдоль ранее построенного $Cycle_1$ (показан как зеленые и синие ребра). (Справа). После завершения прохода через $Cycle_1$ Лео продолжает беспорядочно ходить по графу и, наконец, создает цикл Эйлера.

На рисунке 3.9 $Cycle_2$ является эйлеровым циклом, хотя это не относится к произвольному графу. В общем случае, на каждой итерации Лео генерирует все больше и больше циклов, и поэтому гарантируется, что рано или поздно некоторый $Cycle_m$ будет проходить по всем ребрам в *Graph*. Этот цикл должен быть эйлеровым, и Лео закончил.

Доказательство теоремы Эйлера дает пример того, что математики называют конструктивным доказательством, которое не только доказывает желаемый результат, но и предоставляет метод построения требуемого объекта. Движения Лео отслеживаются до тех пор, пока он неизбежно не произведет эйлеров цикл в сбалансированном и сильно связанном графе *Graph*, как показано в следующем псевдокоде.

```
EulerianCycle(Graph)
  form a cycle Cycle by randomly walking in Graph (don't visit the same edge twice!)
  while there are unexplored edges in Graph
    select a node newStart in Cycle with still unexplored edges
    form Cycle' by traversing Cycle (starting at newStart) and then randomly walking
    Cycle ← Cycle'
  return Cycle
```

Теперь можно решить проблему эйлерова цикла для любого графа. Это может быть не очевидно, но хорошая реализация *EulerianCycle* будет работать за линейное время. Чтобы достичь этого ускорения, нужно использовать эффективную структуру данных, чтобы поддерживать текущий цикл, который создает Лео, а также список неиспользованных ребер, инцидентных каждому узлу, и список узлов текущего цикла, которые имеют неиспользованные ребра.

Задача 3.1. Реализовать *EulerianCycle*.

Вход: список смежности эйлерова ориентированного графа.

Выход: эйлеров цикл данного графа.

Пример входа:

0 -> 3

1 -> 0

2 -> 1,6

3 -> 2

4 -> 2

5 -> 4

6 -> 5,8

7 -> 9

8 -> 7

9 -> 6

Пример выхода:

6->8->7->9->6->5->4->2->1->0->3->2->6

Теперь можно проверить, имеет ли направленный граф эйлеров цикл, но непонятно, применимо ли это к эйлерову пути. Рассмотрим граф де Брёйна слева на рисунке 3.10, который имеет эйлеров путь, но не имеет эйлерова цикла, потому что узлы TA и TT не сбалансированы. Однако, можно преобразовать этот эйлеров путь в эйлеров цикл, добавив единственное ребро, соединяющее TT и TA, как показано на рисунке 3.10.

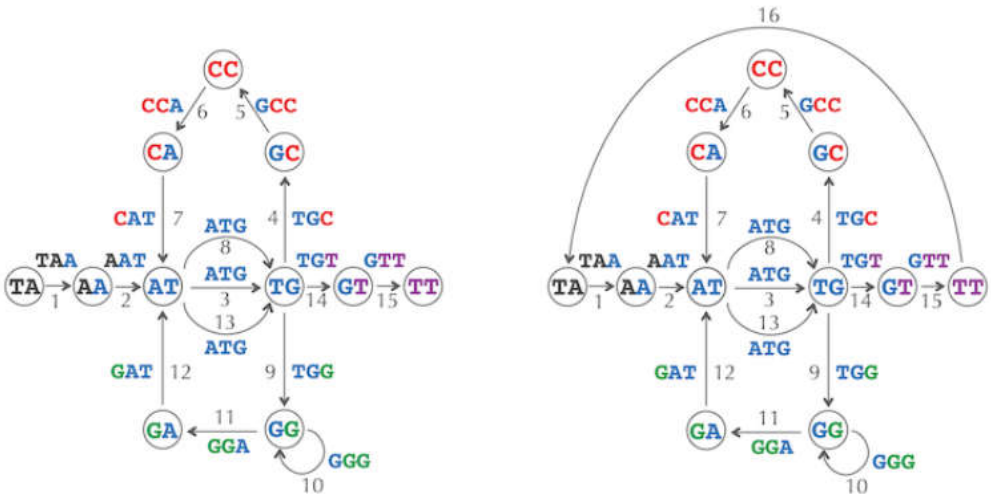


Рис. 3.10. Преобразование эйлерова пути (слева) в эйлеров цикл (справа) путем добавления ребра.

В более общем плане рассмотрим граф, который не имеет эйлерова цикла, но имеет эйлеров путь. Если эйлеров путь в этом графе соединяет узел *v* с другим узлом *w*, то граф почти сбалансирован, что означает, что все его узлы, кроме *v* и *w*, сбалансированы. В этом случае добавление дополнительного ребра из *w* в *v* преобразует эйлеров путь в эйлеров цикл. Таким образом, почти сбалансированный граф имеет эйлеров путь тогда и только тогда, когда добавление ребра между его неуравновешенными узлами делает граф сбалансированным и сильно связанным.

Аналог теоремы Эйлера для неориентированных графов означает, что в Кенигсберге нет эйлерова пути.

Задача 3.2. Найти эйлеров путь в графе.

Вход: список смежности эйлерова ориентированного графа, который имеет эйлеров путь.

Выход: эйлеров путь данного графа.

Пример входа:

0 -> 2

1 -> 3

2 -> 1

3 -> 0,4

6 -> 3,7

7 -> 8

8 -> 9

9 -> 6

Пример выхода:

6->7->8->9->6->3->0->2->1->3->4

Теперь есть метод для сборки генома, так как проблема восстановления строк сводится к поиску эйлерова пути на графе де Брёйна, полученном из прочтений.

Задача 3.3. Решить проблему восстановления строки.

Вход: целое число *k*, соответствующий список *k*-меров *Patterns*.

Выход: строка *Text* с композицией *k*-меров, эквивалентной *Patterns*.

Пример входа:

4

CTTA

ACCA

TACC

GGCT

GCTT

TTAC

Пример выхода:

GGCTTACCA

Теперь, когда известно, как использовать граф де Брёйна для решения проблемы восстановления строк, также можно построить *k*-универсальную строку для любого значения *k*.

Следует отметить, что де Брёйн был заинтересован в построении *k*-универсальных циклических строк. Например, 00011101 представляет собой 3-универсальную циклическую строку, так как она содержит каждый из восьми двоичных 3-меров (000, 001, 011, 111, 110, 101,010 и 100) ровно один раз (см. Рисунок 3.11).

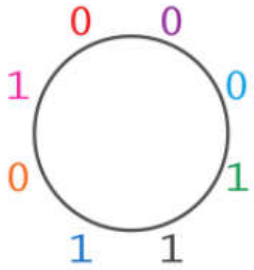


Рис. 3.11. Циклическая 3-универсальная строка 00011101 содержит каждый из двоичных 3-меров ровно один раз.

Задача. Найти *k*-универсальную циклическую строку.

Вход: целое число *k*.

Выход: *k*-универсальная циклическая строка.

Как и для линейных строк, проблема построения *k*-универсальной циклической строки является конкретным случаем более общей проблемы, которая требует восстановления циклической строки с учетом ее композиции *k*-меров. Эта проблема моделирует сборку циклического генома, содержащего одну хромосому, подобно геномам большинства бактерий. Известно, что можно восстановить циклическую цепочку из ее композиции *k*-меров, найдя эйлеров цикл на графе де Брёйна, построенный из этих *k*-меров. Поэтому можно построить *k*-универсальную циклическую двоичную цепочку, найдя эйлеров цикл на графе де Брёйна, построенном из совокупности всех двоичных *k*-меров (см. Рисунок 3.12).

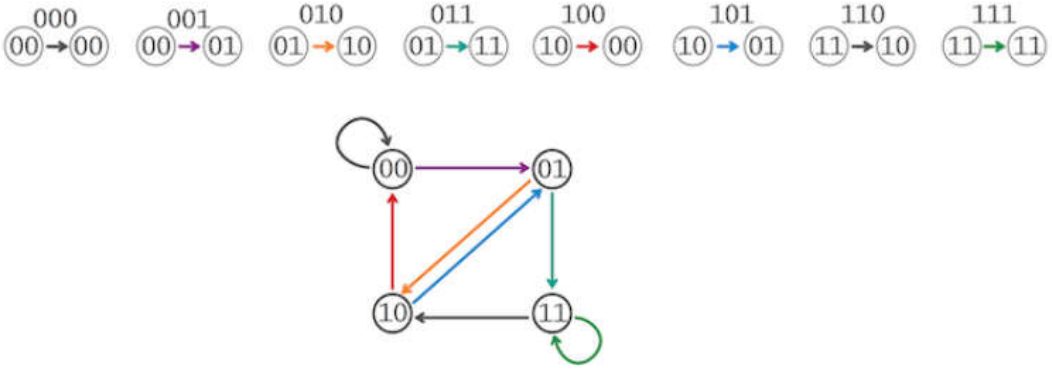


Рис. 3.12. (сверху) Граф, состоящий из восьми изолированных направленных ребер, по одному для каждого двоичного 3-мера. Узлы каждого ребра соответствуют префиксу и суффиксу 3-мера. (снизу) Склеивание идентично помеченных узлов в графе сверху приводит к графу де Брёйна, содержащему четыре узла. Эйлеров цикл, проходящий по ребрам 000 → 001 → 011 → 111 → 110 → 101 → 010 → 000, дает 3-универсальную циклическую строку 00011101.

Несмотря на то, что поиск 20-универсальной циклической строки сводится к нахождению эйлерова цикла в графе с более чем миллионом ребер, есть быстрый алгоритм для решения этой проблемы. Пусть $BinaryStrings_k$ – множество всех 2^k двоичных *k*-меров. Единственное, что нужно сделать, это решить задачу построения *k*-универсальной циклической строки – найти эйлеров цикл в $DeBruijn(BinaryStrings_k)$. Строит обратить внимание, что узлы этого графа представляют все возможные бинарные (*k-1*)-меры. Ориентированное ребро связывает (*k-1*)-мер *Pattern* с (*k-1*)-мером *Pattern*' в этом графе, если существует *k*-мер, чей префикс является *Pattern*, а суффикс – *Pattern*'.

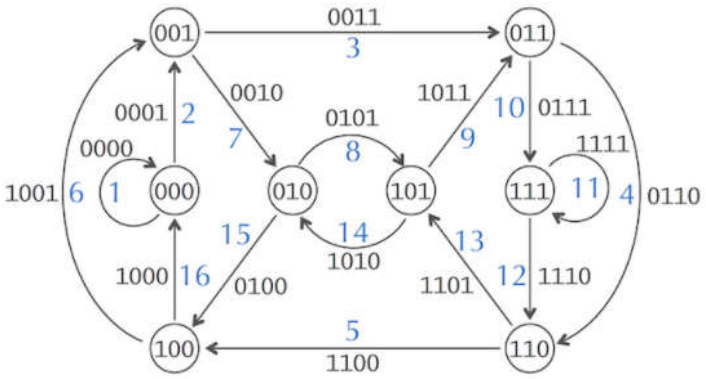


Рис. 3.13. Эйлеров цикл, составляющий циклическую 4-универсальную строку 0000110010111101 в *DeBruijn*(*BinaryStrings*$_4$).

Задача 3.4. Найти *k*-универсальную циклическую строку.

Вход: целое число *k*.

Выход: *k*-универсальная циклическая строка.

Пример входа:

4

Пример выхода:

0000110010111101

Лекция 4. Сборка генов из пар прочтений.

Раньше была описана идеализированная форма сборки генома. Далее обсудим ряд практически мотивированных тем, которые помогут оценить современные методы, используемые современными сборщиками геномов.

Сборка чтения, взятая из случайно генерируемого текста, тривиальна, так как случайные строки не должны иметь длинных повторов. Более того, графы де Брёйна становятся все менее и менее запутанными при увеличении длины прочтения (см. Рисунок 4.1). Как только длина прочтения превышает длину всех повторов в геноме (при условии, что прочтения не имеют ошибок), граф де Брёйна превращается в путь. Однако, несмотря на многие попытки, биологи еще не выяснили, как генерировать длинные и точные прочтения. Самые точные на сегодняшний день технологии секвенирования генерируют прочтения, длина которых составляет всего около 300 нуклеотидов, что слишком мало, чтобы охватить большинство повторов даже в коротких бактериальных геномах.

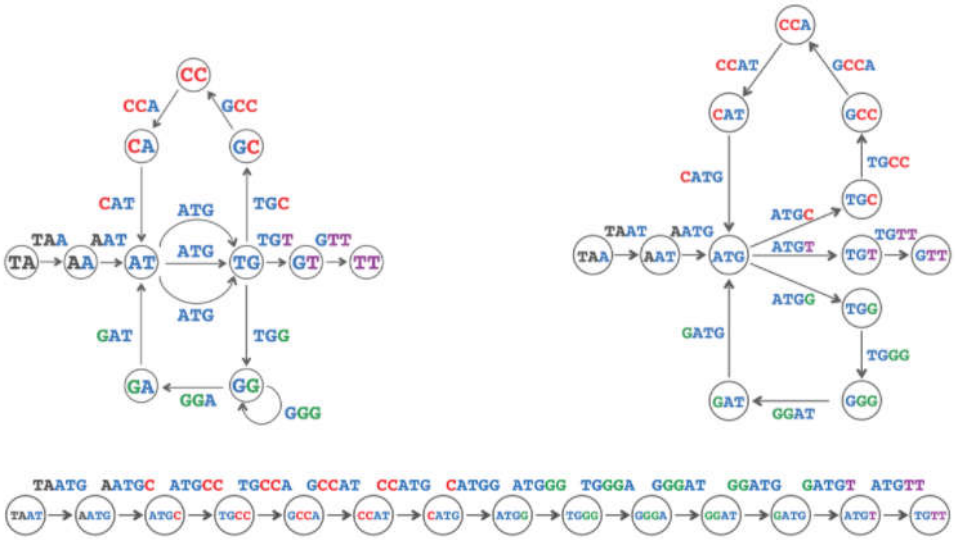


Рис. 4.1. Граф *DeBruijn*$_4$(TAATGCCATGGGATGTT) (вверху справа) менее запутан, чем граф *DeBruijn*$_3$(TAATGCCATGGGATGTT) (вверху слева). Граф *DeBruijn*$_5$(TAATGCCATGGGATGTT) – это путь (внизу).

Ранее было показано, что строка TAATGCCATGGGATGTT не может быть однозначно восстановлена из ее композиции 3-меров, поскольку другая строка (TAATGGGATGCCATGTT) имеет ту же композицию.

Увеличение длины прочтения поможет определить правильную сборку, но поскольку увеличение длины прочтения представляет собой сложную экспериментальную проблему, биологи предложили непрямой способ увеличения длины прочтения путем генерации пар прочтений, которые представляют собой пары, разделенные фиксированным расстоянием d в геноме (см. рисунок 4.2). Можно подумать о паре прочтений как о длинном чтении с пропусками длины $k + d + k$, чьи первый и последний k-меры известны, а средний сегмент длины d неизвестен. Тем не менее, пары прочтений содержат больше информации, чем k-меры, и поэтому нужно иметь возможность использовать их для улучшения сборок. Если бы можно было определить нуклеотиды в среднем сегменте пары прочтений, можно было бы увеличить длину прочтения от k до $2 \cdot k + d$.

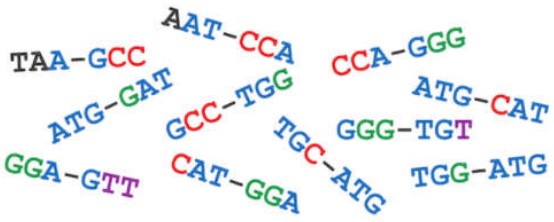


Рис. 4.2. Пары прочтений, отобранные из TAATGCCATGGGATGTT и сформированные с помощью прочтений длины 3, разделенных пробелом длины 1. Простым, но

неэффективным способом собрать эти пары прочтений, является построение графа де Брёйна отдельных прочтений (3-меров) в пределах пары.

Пусть *Reads* – это совокупность всех *2N k*-мерных прочтений, взятых из *N* пар прочтений. Обратите внимание, что пара прочтений $Read_1$ и $Read_2$, образованная *k*-мером соответствует двум ребрам в графе де Брёйна $DeBruijn_k(Reads)$. Поскольку эти прочтения разделены расстоянием *d* в геноме, в $DeBruijn_k(Reads)$ должен быть путь длины $k + d$, соединяющий узел в начале ребра, соответствующий $Read_1$, с узлом в начале ребра, соответствующим $Read_2$, как показано на рисунке 4.3. Если есть только один путь длины $k+d+1$, соединяющий эти узлы, или если все такие пути описывают одну и ту же строку, тогда можно преобразовать пару прочтений, образованную $Read_1$ и $Read_2$ в виртуальное прочтение длины $2 \cdot k + d$, начинающееся как $Read_1$, проходящее этот путь и заканчивающееся $Read_2$.

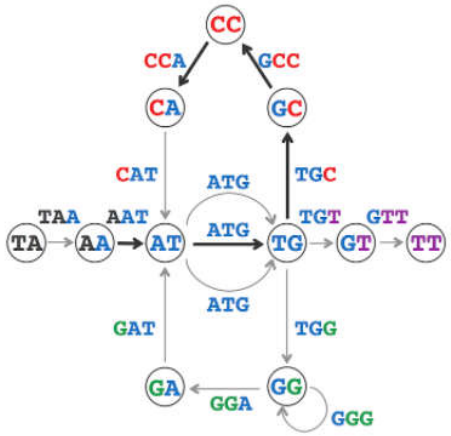


Рис. 4.3. Каждый путь длины 3 + 1 = 4 между ребрами, обозначенными AAT и CCA, выдает AATGCCA. Таким образом, пара прочтений AAT-CCA может быть преобразована в длинное виртуальное прочтение AATGCCA, показанное выделенным путем. Есть три таких пути, потому что есть три возможных варианта ребер, помеченных как ATG.

Например, рассмотрим граф де Брёйна на рисунке 4.3 который генерируется из всех прочтений, присутствующих в парах прочтений с рисунка 4.2. Существует единственная строка, записанная путями длиной $k + d + 1 = 5$ между ребрами, обозначенными AAT и CCA, в паре прочтений AAT-CCA. Таким образом, из двух коротких прочтений длины *k* создано длинное виртуальное чтение длины $2 \cdot k + d$, что позволяет вычислить то, что исследователи все еще не могут достичь экспериментально. После предварительной обработки графа де Брёйна для создания длинных виртуальных чтений можно просто построить граф де Брёйна из этих длинных чтений и использовать его для сборки генома.

Хотя идея преобразования пар прочтений в длинные виртуальные прочтения используется во многих программах сборки, при этом было использовано

оптимистичное предположение: «Если существует только один путь длины *k+d+1*, соединяющий эти узлы, или если все такие пути обозначают одну и ту же строку». На практике это предположение ограничивает применение метода длинного виртуального прочтения для сборки пар, поскольку сильно повторяющиеся геномные области часто содержат несколько путей одинаковой длины между двумя ребрами, обозначающих разные строки (см. рисунок 4.4). Если это так, то нельзя однозначно преобразовать пару прочтений в длинное прочтение. Вместо этого опишем альтернативный подход к анализу пар прочтений.

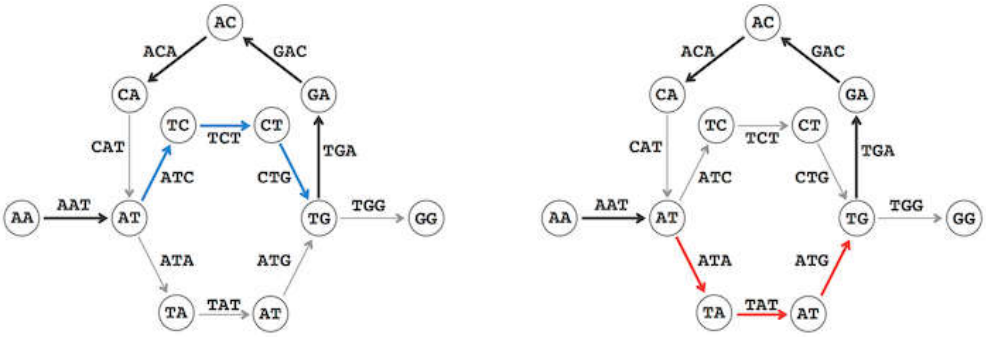
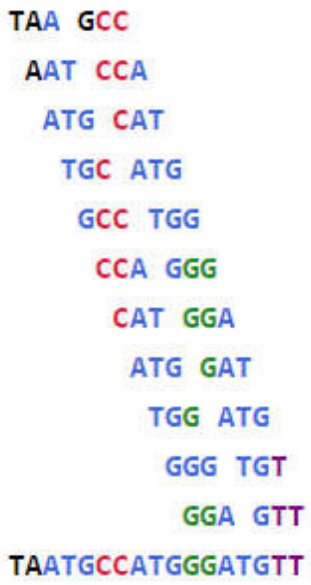


Рис. 4.4. (Слева) Выделенный путь в $DeBruijn_3$(AATCTGACATATGG) описывает длинное виртуальное прочтение AATCTGACA, которое является подстрокой AATCTGACATATGG. (Справа) Выделенный путь в том же графе описывает длинное виртуальное прочтение AATATGACA, которое не встречается в AATCTGACATATGG.

Для данной строки *Text* (*k, d*)-мер представляет собой пару *k*-меров в *Text*, разделенных расстоянием *d*. ($Pattern_1|Pattern_2$) обозначает (*k, d*)-мер, чьими *k*-мерами являются $Pattern_1$ и $Pattern_2$. Например, (ATG | GGG) является (3,4)-мером в TAATGCCATGGGATGTT. Композиция (*k, d*)-меров в *Text*, обозначаемая $PairedComposition_{k,d}(Text)$, представляет собой совокупность всех (*k, d*)-меров в *Text* (включая повторяющиеся (*k, d*)-меры). Например, здесь $PairedComposition_{3,1}$(TAATGCCATGGGATGTT):



Поскольку порядок (3,1)-меров неизвестен, перечислим их в соответствии с лексикографическим порядком 6-меров, образованных конкатенированными 3-мерами:

(AAT|CCA) (ATG|CAT) (ATG|GAT) (CAT|GGA) (CCA|GGG) (GCC|TGG) (GGA|GTT) (GGG|TGT) (TAA|GCC) (TGC|ATG) (TGG|ATG)

Стоит заметить: хоть в композиции 3-меров этой строки есть повторяющиеся 3-меры, в ее композиции пар нет повторных (3,1)-меров. Кроме того, хоть TAATGCCATGGGATGTT и TAATGGGATGCCATGTT имеют одинаковую композицию 3-меров, они имеют разные композиции (3,1)-меров. Таким образом, если можно сгенерировать композицию (3,1)-меров этих строк, то можно будет различать их. Но остается вопрос, как можно восстановить строку из ее композиции (*k, d*)-меров, и можно ли адаптировать подход графа де Брёйна для этой цели.

Задача. Восстановить строку из ее композиции пар.

Вход: набор парных *k*-меров *PairedReads* и целое число *d*.

Выход: строка *Text* с композицией (*k, d*)-меров, равной *PairedReads* (если такая строка существует).

Для данного (*k, d*)-мера ($a_1...a_k|b_1...b_k$), определим его префикс как (*k-1, d+1*)-мер ($a_1...a_{k-1}|b_1...b_{k-1}$), а его суффикс как (*k-1, d+1*)-мер ($a_2...a_k|b_2...b_k$). Например, *Prefix*((GAC | TCA)) = (GA | TC) и *Suffix*((GAC | TCA)) = (AC | CA).

Стоит заметить, что для последовательных (*k, d*)-меров, появляющихся в *Text*, суффикс первого (*k, d*)-мера равен префиксу второго (*k, d*)-мера. Например, для последовательных (*k, d*)-меров (TAA|GCC) и (AAT|CCA) в строке TAATGCCATGGGATGTT: *Suffix*((TAA|GCC))=*Prefix*((AAT|CCA))=(AA|CC).

Для заданной строки *Text* построим граф $PathGraph_{k,d}(Text)$, представляющий путь, образованный $|Text|-(k+d+k)+1$ ребрами, соответствующими всем (k, d)-мерам в *Text*. Помечаем ребра в этом пути как (k, d) и обозначаем начальный и конечный узлы ребра своим префиксом и суффиксом соответственно. На рисунке 4.5 показан $PathGraph_{3,1}$(TAATGCCATGGGATGTT).

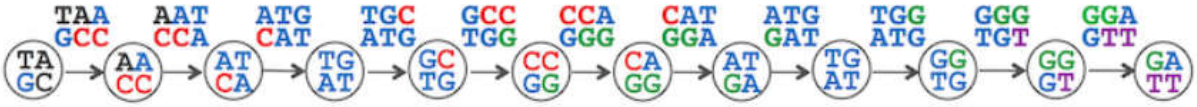


Рис. 4.5. $PathGraph_{3,1}$(TAATGCCATGGGATGTT). Каждый (3,1)-мер отображен в виде двухстрочного выражения для экономии места.

Парный граф де Брёйна, обозначенный $DeBruijn_{k,\,d}(Text)$, формируется путем склеивания идентично помеченных узлов в $PathGraph_{k,\,d}(Text)$; на рисунке 4.6 показан процесс построения парного графа де Брёйна $DeBruijn_{3,1}$(TAATGCCATGGGATGTT). Парный граф де Брейна менее «запутан», чем граф де Брёйна, построенный из отдельных прочтений.

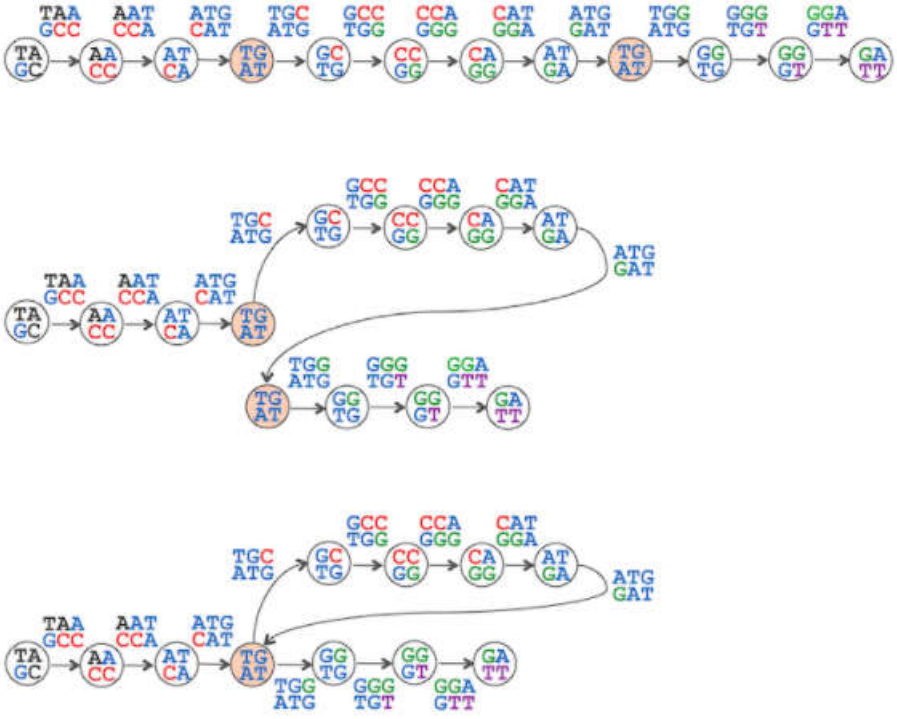


Рис. 4.6. (Сверху) $PathGraph_{3,1}$(TAATGCCATGGGATGTT) образован 11 ребрами и 12 узлами. Только два из этих узлов имеют одну и ту же метку (TG | AT). (В середине) Сближение двух идентично обозначенных узлов в процессе подготовки к склеиванию. (Снизу). Парный граф де Брёйна $DeBruijn_{3,1}$(TAATGCCATGGGATGTT) получается из $PathGraph_{3,1}$(TAATGCCATGGGATGTT) путем склеивания узлов с меткой (TG | AT). Этот парный граф де Брёйна имеет единственный эйлеров путь для TAATGCCATGGGATGTT.

Определим $CompositionGraph_{k,\,d}(Text)$ как граф, состоящий из $|Text|-(k+d+k)+1$ изолированных ребер, помеченных как (k, d) в *Text*, и узлов, помеченных

префиксами и суффиксами этих меток. На рисунке 4.7 показан граф $CompositionGraph_{3,1}$(TAATGCCATGGGATGTT). Склеивание идентично обозначенных узлов в $PairedCompositionGraph_{k,d}$(Text) приводит к точно тому же графу де Брёйна, как склейка тождественно помеченных узлов в $PathGraph_{k,d}(Text)$. На практике *Text* неизвестна, но можно сформировать $CompositionGraph_{k,d}(Text)$ непосредственно из композиции (*k, d*)-меров *Text*, а шаг склейки приведет к парному графику де Брёйна этой композиции. Геном можно восстановить, следуя эйлерову пути на этом графе де Брёйна.

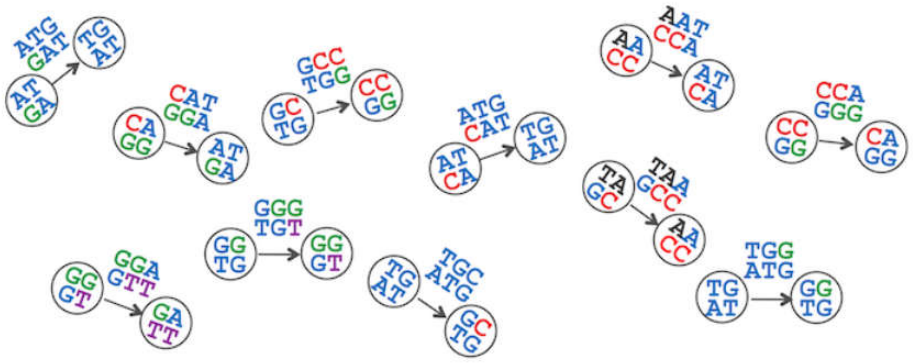


Рис. 4.7. Граф $CompositionGraph_{3,1}$(TAATGCCATGGGATGTT) представляет собой набор изолированных ребер. Каждое ребро помечено (3, 1)-мером в TAATGCCATGGGATGTT; стартовый узел ребра помечен префиксом (3,1)-мера, а конечный узел ребра помечен суффиксом этого (3, 1)-мера. Склеивание идентично помеченных узлов дает парный граф де Брёйна с предыдущего рисунка.

Ранее было показано, что каждое решение задачи восстановления строки соответствует эйлерову пути на графе де Брёйна, построенном из композиции *k*-меров. Аналогично, каждое решение задачи восстановления строки из ее композиции пар соответствует эйлерову пути на парном графе де Брёйна, построенном из композиции (*k, d*)-меров.

Также было показано, что каждый эйлеров путь на графе де Брёйна, построенный из композиции *k*-меров, решает задачу восстановления строки. Так ли это для парного графа де Брёйна?

Не каждый эйлеров путь на парном графе де Брёйна, построенный из композиции (*k, d*)-меров, решает задачу восстановления строки из ее композиции пар.

Задача 4.1. Восстановить строку из ее композиции пар.

Вход: целые числа *k* и *d*, набор парных *k*-меров *PairedReads*.

Выход: строка *Text* с композицией (*k, d*)-меров, равной *PairedReads* (если такая строка существует).

Обсуждение сборки генома до сих пор основывалось на различных предположениях. Соответственно, применение графов де Брёйна к реальным

данным секвенирования не является простой процедурой. Далее будут описаны практические проблемы, возникающие с современными технологиями секвенирования, и некоторые вычислительные методы, которые были разработаны для решения этих задач. В этом обсуждении сначала для простоты предположим, что прочтения считываются как *k*-меры вместо пар прочтений.

Во-первых, для данного *k*-мера генома, определим его охват как число прочтений, которым принадлежит этот *k*-мер. Считается, что машина секвенирования может генерировать все *k*-меры, присутствующие в геноме, но это предположение об «идеальном покрытии *k*-мерами» не выполняется на практике. Например, популярная технология секвенирования *Illumina* генерирует записи длиной около 300 нуклеотидов, но эта технология по-прежнему пропускает множество присутствующих в геноме 300-меров (даже если средний охват очень высок), и почти все прочтения, которые она генерирует, имеют ошибки секвенирования.

В левой части рисунка 4.8 показаны четыре прочтения 10-мера, которые захватывают не все 10-меры из примера генома. Однако, если предпринять противоречивый шаг по разрыву этих прочтений на более короткие 5-меры (рис. 4.8, справа), то эти 5-меры составляют идеальное покрытие. Этот подход разрыва прочтений используется многими современными сборщиками.

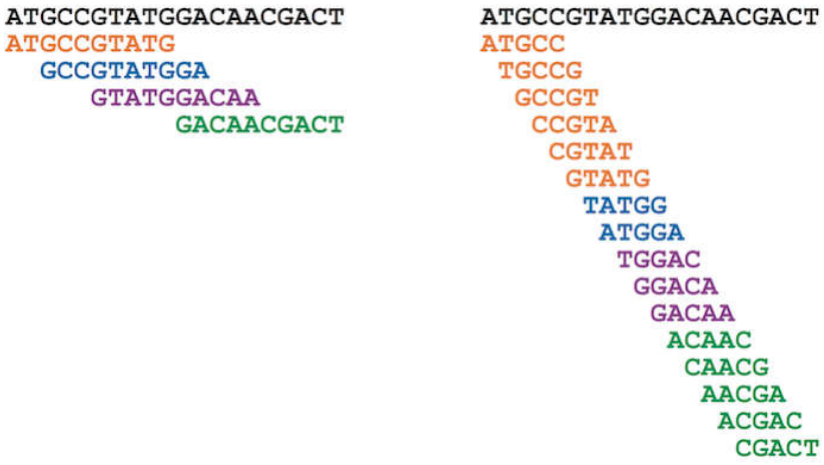


Рис. 4.8. Разрыв прочтений 10-меров (слева) на 5-меры приводит к полному охвату генома 5-мерами (справа).

Подход разрыва прочтений должен содержать практический компромисс. С одной стороны, чем меньше значение *k*, тем больше вероятность того, что покрытие *k*-мерами будет идеально. С другой стороны, меньшие значения *k* приводят к более запутанному графу де Брёйна, что затрудняет получение гена из этого графа.

Даже после разрыва прочтений, большинство сборок по-прежнему имеют пробелы в покрытии *k*-мерами. Это приводит к тому, что граф де Брёйна имеет

пропущенные ребра, и поэтому поиск эйлерова пути терпит неудачу. В этом случае биологи часто собирают контиги (длинные смежные сегменты генома), а не целые хромосомы. Например, типичная задача бактериального секвенирования может приводить к возникновению около ста контигов, длиной от нескольких тысяч до нескольких сотен тысяч нуклеотидов. Для большинства геномов порядок этих контигов вдоль генома остается неизвестным. Биологи предпочли бы знать всю последовательность генома, но стоимость упорядочения контигов в окончательную сборку и закрытие пробелов с использованием более дорогих экспериментальных методов часто является непомерно высокой.

Однако, можно выводить контиги из графа де Брёйна. Путь в графе называется не ветвящимся, если $in(v) = out(v) = 1$ для каждого промежуточного узла v этого пути, то есть для каждого узла, за исключением, возможно, начального и конечного узлов пути. Максимальный неветвящийся путь – это не ветвящийся путь, который не может быть расширен до более длинного не ветвящегося пути. Эти пути интересны, потому что строки нуклеотидов, которые они собирают, должны присутствовать в любой сборке с данной композицией k-меров. По этой причине контиги соответствуют строкам, записанным максимальными неветвящимися путями на графе де Брёйна. Например, приведенный на рисунке 4.9 граф де Брёйна, построенный для композиции 3-меров TAATGCCATGGGATGTT, имеет 9 максимальных неветвящихся путей, которые описывают контиги TAAT, TGTT, TGCCAT, ATG, ATG, ATG, TGG, GGG и GGAT. На практике у биологов нет другого выбора, кроме как разбить геномы на контиги, даже в случае идеального покрытия (как на рисунке 4.9), поскольку повторы не позволяют вывести единственный эйлеров путь.

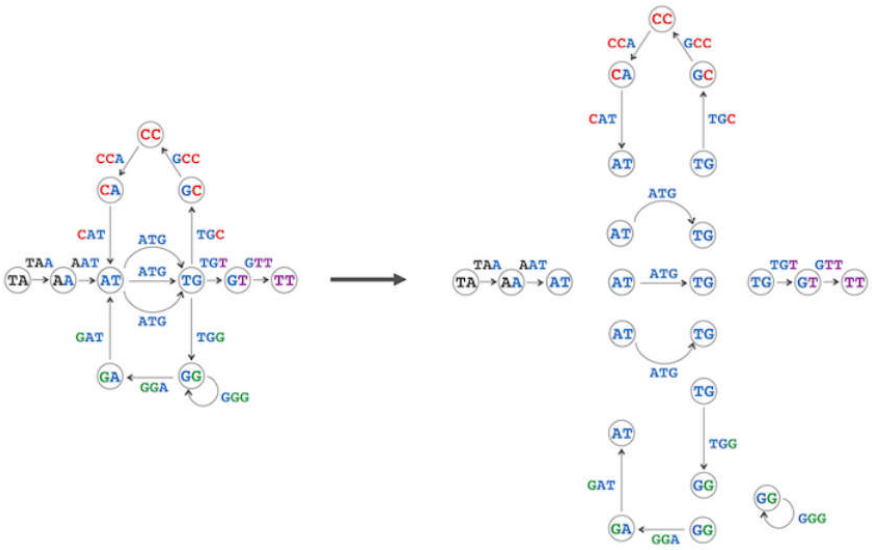


Рис. 4.9. Разбиение графа де Брёйна на 9 максимальных неветвящихся путей, представляющих контиги TAAT, TGTT, TGCCAT, ATG, ATG, ATG, TGG, GGG и GGAT.

Задача 4.2. Создать контиги из коллекции прочтений (с несовершенным покрытием).

Вход: коллекция *k*-меров *Patterns*.

Выход: все контиги в *DeBruijn*(*Patterns*).

Пример входа:

ATG

ATG

TGT

TGG

CAT

GGA

GAT

AGA

Пример выхода:

AGA

ATG

ATG

CAT

GAT

TGGA

TGT

Ошибочные прочтения представляют собой еще одну проблему для реальных проектов секвенирования. Добавление единственного ошибочного прочтения CGTACGGACA (с одной ошибкой, которая неправильно считывает t как C) к набору «сломанных» 5-мерных прочтений, приводит к ошибочным 5-мерам CGTAC, GTACG, TACGG, ACGGA и CGGAC. Эти ошибки приводят к ошибочному пути от узла CGTA к узлу GGAC в графе де Брёйна на рисунке 4.10, а это означает, что, даже если сгенерировано корректное прочтение CGTATGGACA, то будет два пути, связывающие CGTA с GGAC на графе де Брёйна. Эта структура называется «пузырьком», который определяется как два коротких непересекающихся пути (например, короче некоторой пороговой длины), соединяющих одну и ту же пару узлов в графе де Брёйна.

Рис. 4.10. Правильный путь CGTA → GTAT → TATG → ATGG → TGGA → GGAC вместе с неправильным путем CGTA → GTAC → TACG → ACGG → CGGA → GGAC образуют «пузырек» на графе де Брёйна, что затрудняет определение верного пути.

Существующие алгоритмы удаляют пузырьки с графов де Брёйна. Практическая задача состоит в том, что, поскольку почти все прочтения имеют ошибки, графы де Брёйна имеют миллионы пузырьков (см. рисунок 4.11).

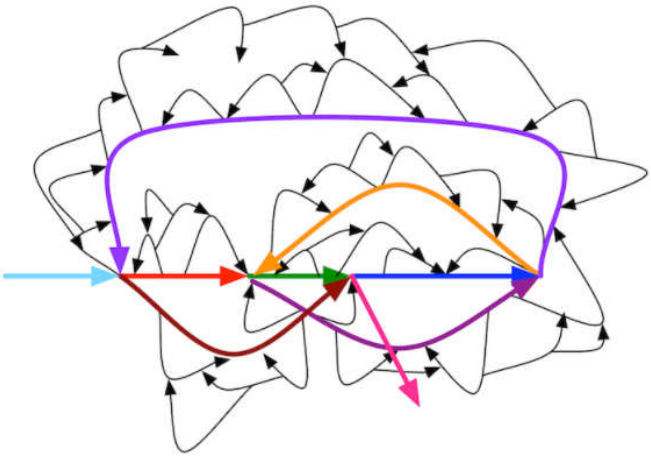


Рис. 4.11. Иллюстрация графа де Брёйна со многими пузырьками. Удаление пузырьков должно оставлять только окрашенные пути.

Удаление пузырьков иногда удаляет правильный путь, тем самым вводя ошибки, а не удаляя их. Хуже того, в геноме с неточными повторами, где повторяющиеся регионы отличаются одним нуклеотидом или некоторым другим небольшим изменением, из двух повторных копий также генерируются пузырьки на графе де Брёйна, потому что одна из копий может быть ошибочной версией другой. Применение удаления пузырьков в эти регионы вводит ошибки сборки. Таким образом, современные сборщики геномов пытаются отличить пузырьки, вызванные ошибками секвенирования (которые должны быть удалены) от пузырьков, вызванных изменениями (которые следует сохранить).

В то время как структура графа де Брёйна требует знания кратности каждого k-мера в геноме (т.е. число раз, когда появляется k-мер), эта информация не является легко доступной из прочтения. Однако, множественность k-мера в геноме часто оценивается с использованием его охвата. В самом деле, k-меров, которые появляются t раз в геноме, как ожидается, будет примерно в t раз больше, чем k-меров, которые появляются только один раз. Охват варьируется

в зависимости от генома, и это условие часто нарушается. В результате существующие сборщики часто собирают повторяющиеся области в геномах, не зная точного количества раз, когда каждый *k*-мер из этой области встречается в геноме.

Лекция 5. Выравнивание биологических последовательностей.

После публикации Уотсона и Крика о структуре двойной спирали ДНК в 1953 году физик Джордж Гамов основал «RNA Tie Club» (клуб галстуков РНК) для известных ученых. Членство в этом клубе означал галстук, расшитый двойной спиралью. Оно было ограничено двадцатью обычными членами (один для каждой аминокислоты), а также четырьмя почетными членами (по одному на каждый нуклеотид). Гамов хотел, чтобы RNA Tie Club имел больше, чем только социальную функцию, собирая высшие научные умы, он надеялся расшифровать сообщение, скрытое в ДНК, определив, как РНК превращается в аминокислоты. Действительно, Сидней Бреннер и Фрэнсис Крик год спустя обнаружили, что аминокислоты транслируются из кодонов (т. е. триплетов нуклеотидов).

В RNA Tie Club было восемь лауреатов Нобелевской премии, но вся эта интеллектуальная мощь не помогла им предпринять дальнейшие шаги по расшифровке генетического кода. В 1961 году Маршалл Ниренберг провел следующий эксперимент: он синтезировал нити РНК, состоящие только из урацила (...UUUUUUUUUUUUU...), добавлял рибосомы и аминокислоты и продуцировал пептид, состоящий только из фенилаланина (...PhePhePhePhe...). Таким образом, Ниренберг пришел к выводу, что кодон UUU РНК кодирует аминокислотный фенилаланин. После успеха Ниренберга, Хар Гобинд Корана синтезировал нить РНК ...UCUCUCUCUCUC... и продемонстрировал, что она транслируется в ...SerLeuSerLeu... После этих открытий остальная часть рибосомного генетического кода была выяснена быстро.

Почти четыре десятилетия спустя Мохамед Марахиель решил решить гораздо более сложную задачу взлома нерибосомального кода. Как было сказано ранее, бактерии и грибы производят антибиотики и другие нерибосомальные пептиды (NRP) без какой-либо зависимости от рибосомы и генетического кода. Вместо этого эти организмы производят NRP, используя гигантский белок под названием NRP-синтетаза:

DNA → RNA → NRP synthetase → NRP

NRP-синтетаза, которая кодирует антибиотик *Tyrocidine B1* длиной 10 аминокислот, включает 10 сегментов, называемых аденилирующими

доменами (A-доменами); каждый A-домен имеет длину около 500 аминокислот и ответственен за добавление одной аминокислоты к *Tyrocidine B1*.

Ранее RNA Tie Club спросил: «Как РНК кодирует аминокислоту». Теперь Марахиель решил ответить на гораздо более сложный вопрос: «Как каждый A-домен кодирует аминокислоту».

Марахиель знал аминокислотные последовательности некоторых A-доменов, а также аминокислоты, которые они добавляют к растущему пептиду. Ниже приведены три из этих A-доменов (взятых из трех разных бактерий), которые кодируют аспарагиновую кислоту (*Asp*), орнитин (*Orn*) и валин (*Val*) соответственно:

```
YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGITYIKLTPSLFHTIVNTASFAFDANFESLRLIVLGGEKIIPIDVIAFRKMYGHTEFINHYGPTEATIGA
AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYIYEQKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS
IAFDASSWEIYAPLLNGGTVVCIDYYTTIDIKALEAVFKQHHIRGAMLPPALLKQCLVSAPTMISSLEILFAAGDRLSSQDAILARRAVGSGVYNAYGPTENTVLS
```

Марахиель предположил, что, поскольку A-домены имеют одинаковую функцию (добавление аминокислоты к растущему пептиду), разные A-домены должны иметь сходные части. A-домены также должны иметь разные части для включения различных аминокислот. Тем не менее, только три консервативных столбца (показаны красным ниже) являются общими для трех последовательностей и, вероятно, возникли случайно:

```
YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGITYIKLTPSLFHTIVNTASFAFDANFESLRLIVLGGEKIIPIDVIAFRKMYGHTEFINHYGPTEATIGA
AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYIYEQKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS
IAFDASSWEIYAPLLNGGTVVCIDYYTTIDIKALEAVFKQHHIRGAMLPPALLKQCLVSAPTMISSLEILFAAGDRLSSQDAILARRAVGSGVYNAYGPTENTVLS
```

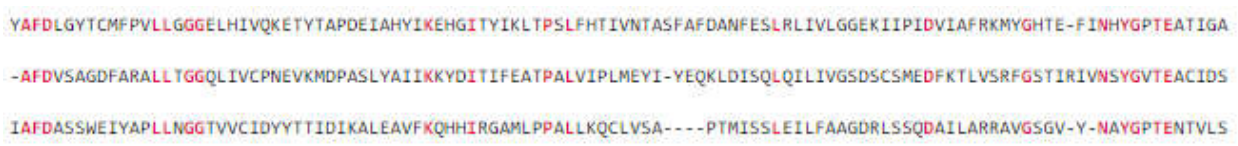
Если сдвинуть вторую последовательность на одну аминокислоту вправо, добавив символ пробела (-) в начало последовательности, тогда найдется 11 общих столбцов.

```
YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGITYIKLTPSLFHTIVNTASFAFDANFESLRLIVLGGEKIIPIDVIAFRKMYGHTEFINHYGPTEATIGA
-AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYIYEQKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS
IAFDASSWEIYAPLLNGGTVVCIDYYTTIDIKALEAVFKQHHIRGAMLPPALLKQCLVSAPTMISSLEILFAAGDRLSSQDAILARRAVGSGVYNAYGPTENTVLS
```

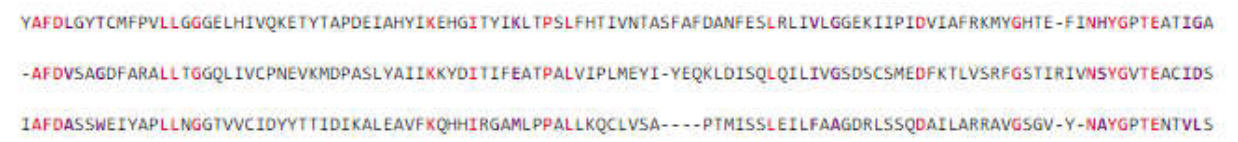
Добавление еще нескольких символов пробела показывает 14 общих столбцов:

```
YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGITYIKLTPSLFHTIVNTASFAFDANFESLRLIVLGGEKIIPIDVIAFRKMYGHTEFINHYGPTEATIGA
-AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYI-YEQKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS
IAFDASSWEIYAPLLNGGTVVCIDYYTTIDIKALEAVFKQHHIRGAMLPPALLKQCLVSA----PTMISSLEILFAAGDRLSSQDAILARRAVGSGVYNAYGPTENTVLS
```

и еще больший сдвиг дает 19 общих столбцов:



Оказывается, что красные столбцы представляют собой консервативное ядро, общее для многих А-доменов. Теперь, когда Марахиель знал, как правильно выравнивать А-домены, он предположил, что некоторые из оставшихся изменяющихся столбцов должны кодировать *Asp*, *Orn* и *Val*. Он обнаружил, что нерибосомальный код определяется 8 нерибосомальными сигнатурами длиной в 6 аминокислот, которые показаны ниже как фиолетовые столбцы.



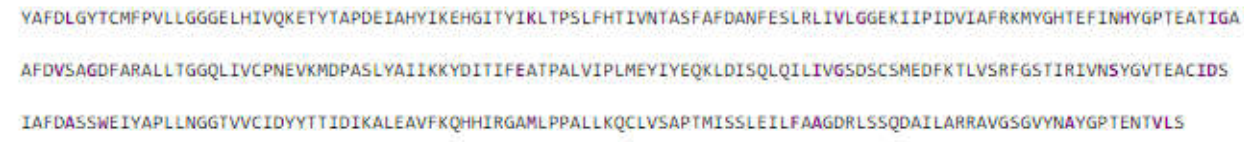
Фиолетовые столбцы определяют сигнатуры LTKVGHIG, VGEIGSID и AWMFAAVL, кодирующие *Asp*, *Orn* и *Val* соответственно:

LTKVGHIG → Asp

VGEIGSID → Orn

AWMFAAVL → Val

Важно отметить, что без первоначального построения консервативного ядра Марахиель не смог бы вывести нерибосомальный код, поскольку 24 аминокислоты в вышеперечисленных сигнатурах не выстраиваются в исходное выравнивание:



Даже после определения консервативного ядра может быть интересно, почему Марахиель выбрал именно эти фиолетовые столбцы. Почему в сигнатурах должно быть 8 аминокислот, а не 5 или 3. Через пятнадцать лет после открытия Марахиеля нерибосомальный код до сих пор не полностью понят.

Взлом Марахиелем нерибосомального кода является лишь одной из многих биологических проблем, которым помогло сравнение последовательностей. Еще один пример силы сравнения последовательностей был установлен в 1983 году, когда Рассел Дулиттл сравнивал новый секвенированный ген фактора роста тромбоцитов (PDGF) со всеми другими известными в то время генами. Дулиттл ошеломил биологов, исследующих рак, когда показал, что PDGF очень похож на последовательность гена, известного как v-sis. Сходство двух генов вызывало недоумение, поскольку их функции сильно различались; ген PDGF кодирует белок, стимулирующий рост клеток, тогда как v-sis является

онкогеном или геном в вирусах, который вызывает раковую трансформацию инфицированных клеток человека. После открытия Дулиттла ученые предположили, что некоторые формы рака могут быть вызваны «хорошим» геном, который делает правильные вещи в неподходящее время.

Однако остается вопрос, как алгоритмически сравнивать последовательности. Возвращаясь к примеру A-домена, вставка пробелов для выявления консервативного ядра, была необоснованной. Непонятно, какой алгоритм нужно использовать, чтобы решить, куда вставить символы пробела, или как количественно определить лучшее выравнивание трех последовательностей.

Для простоты будем сравнивать только две последовательности. Расстояние Хэмминга, которое считает несоответствия в двух строках, строго предполагает, что *i*-й символ одной последовательности поставлен против *i*-го символа другой последовательности. Однако, поскольку биологические последовательности подвержены вставкам и удалениям, часто бывает, что *i*-й символ одной последовательности соответствует символу в совершенно другом положении в другой последовательности. Поэтому целью является найти наиболее подходящее соответствие символов.

Например, ATGCATGC и TGCATGCA не имеют совпадающих позиций, поэтому их расстояние Хэмминга равно 8:

```
ATGCATGC
TGCATGCA
```

Однако эти строки имеют 7 совпадающих позиций, если их по-разному выровнять:

```
ATGCATGC-
-TGCATGCA
```

Строки ATGCTTA и TGCATTAA имеют более тонкое сходство:

```
ATGC-TTA-
-TGCATTAA
```

Эти примеры приводят к постулированию понятия хорошего выравнивания, которое соответствует максимально возможному числу символов.

Можно думать о максимизации количества совпадающих символов в двух строках как об игре с одним человеком. На каждом шаге есть два варианта. Можно удалить первый символ из каждой последовательности, и в этом случае заработать очко, если символы совпадают; наоборот, можно удалить первый символ из любой из двух последовательностей, и в этом случае не получить очков, но подготовить позицию, чтобы заработать больше очков в последующих ходах. Цель – максимизировать количество очков.

На рисунке 5.1 показан один из способов игры в эту игру для последовательностей ATGTTATA и ATCGTCC, с результатом 4 очка. Каждый раз, когда удаляется символ слева, он добавляется к растущему выравниванию ATGTTATA и ATCGTCC справа. Когда только один символ удаляется в очередь, он выравнивается с символом пробела.

Рис. 5.1. Выравнивание последовательностей ATGTTATA и ATCGTCC в форме игры.

Теперь определим выравнивание последовательностей v и w как двухстроковую матрицу так, чтобы первая строка содержала символы v (по порядку), вторая строка содержала символы w (по порядку), а символы пробела могут быть вставлены в обе строки, если два символа пробела не выровнены друг относительно друга. Ниже приведен пример выравнивания ATGTTATA и ATCGTCC с рисунка 5.1.

```
A T - G T T A T A
A T C G T - C - C
```

Выравнивание представляет собой один из возможных сценариев, по которым v может превратиться в w. Столбцы, содержащие одну и ту же букву в обеих строках, называются совпадениями и представляют собой консервативные нуклеотиды, тогда как столбцы, содержащие разные буквы, называются промахами и представляют собой однонуклеотидные замены. Столбцы,

содержащие символы пробела, называются индели: столбец, содержащий пробел в верхней строке выравнивания, называется вставкой, так как подразумевается вставка символа при преобразовании *v* в *w*; столбец, содержащий символ пробела в нижней строке выравнивания, называется удалением, так как он указывает на удаление символа при преобразовании *v* в *w*. Выравнивание выше имеет четыре совпадения, два промаха, одну вставку и два удаления.

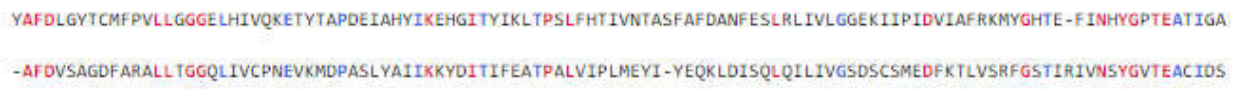
Совпадения в выравнивании двух строк определяют общую подпоследовательность двух строк или последовательность символов, входящих в одном и том же порядке (не обязательно последовательно) в обе строки. Например, выравнивание выше указывает, что ATGT является общей подпоследовательностью ATGTTATA и ATCGTCC. Таким образом, выравнивание двух строк, максимизирующих количество совпадений, соответствует самой длинной общей подпоследовательности этих строк. Две строки могут иметь более одной наиболее длинной общей подпоследовательности.

Задача. Найти самую длинную общую подпоследовательность двух строк.

Вход: две строки.

Выход: самая длинная общая подпоследовательность этих строк.

Если ограничить рассмотрение двумя A-доменами, кодирующими *Asp* и *Orn*, то в дополнение к 19 совпадениям, которые были найдены ранее, можно найти еще 10 совпадений (показаны синим), что дает общую подпоследовательность длины 29.



Ни один из алгоритмических подходов, которые были рассмотрены до сих пор, не поможет решить проблему наибольшей общей подпоследовательности, поэтому, сначала нужно рассмотреть другую проблему, которая может показаться совершенно не связанной с выравниванием последовательностей.

Представьте, что вы являетесь туристом в Мидтауне Манхэттена, и вы хотите посетить как можно больше достопримечательностей на своем пути от перекрестка 59-й улицы и 8-й авеню до перекрестка 42-й улицы и 3-й авеню (рис. 5.2). Однако, вы ограничены во времени, и на каждом перекрестке вы можете пойти только на юг (↓) или на восток (→). Можно выбирать из множества разных путей через карту, но ни один путь не будет посещать все достопримечательности. Задача поиска пути, который посещает большинство достопримечательностей, называется задачей Манхэттенского туриста.

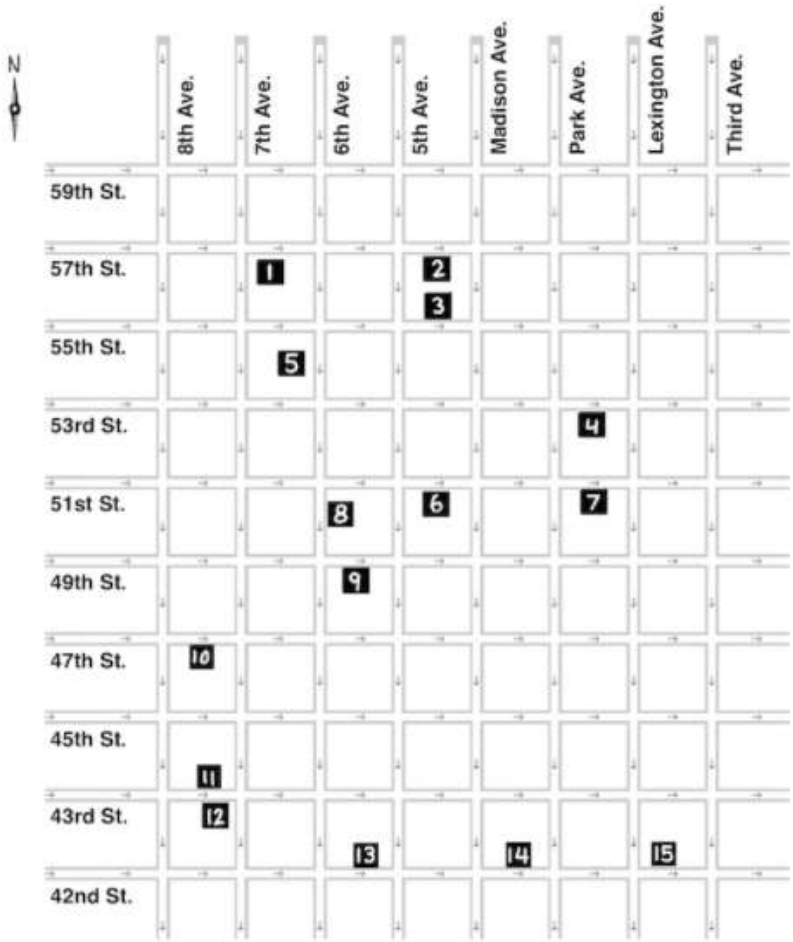


Рис. 5.2. Упрощенная схема Мидтауна Манхэттена. Турист стартует на пересечении 59-й улицы и 8-й авеню в северо-западном углу и заканчивает на пересечении 42-й улицы и 3-й авеню в юго-восточном углу, путешествуя только на юг или восток между перекрестками. Представленные достопримечательности: Карнеги-холл (1), Tiffany & Co. (2), здание Sony (3), Музей современного искусства (4), отель Four Seasons (5), собор Св. Патрика (6) (7), Radio City Music Hall (8), Рокфеллер-центр (9), здание Paramount (10), здание New York Times (11), Таймс-сквер (12), General Society of Mechanics and Tradesmen (13), Центральный вокзал (14) и здание Крайслер (15).

Представим карту Манхэттена как ориентированный граф *ManhattanGraph*, в котором каждое пересечение моделируется как узел, а каждый блок между двумя пересечениями – как направленное ребро, указывающее направление движения (↓ или →), как показано на рисунке 5.3. Затем назначим каждому направленному ребру вес, равный количеству достопримечательностей вдоль соответствующего блока. Стартовый (синий) узел называется узлом-источником, а конечный (красный) узел называется узлом-стоком. Суммарный вес вдоль пути от источника к стоку дает количество достопримечательностей по этому пути; поэтому для решения Манхэттенской туристической проблемы нужно найти путь с максимальным весом, соединяющий источник со стоком (также называемый самым длинным путем) в *ManhattanGraph*.

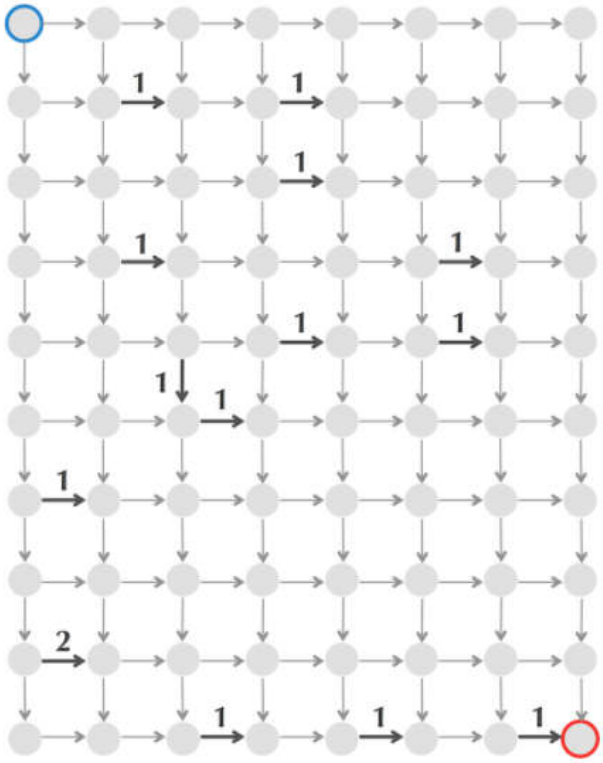


Рис. 5.3. Ориентированный граф *ManhattanGraph*, в котором вес каждого ребра определяется как количество достопримечательностей вдоль этого городского квартала (веса, равные 0, не показаны).

Можно смоделировать любую прямоугольную сетку улиц, используя аналогичный ориентированный граф; на рисунке 5.4 показан граф для гипотетического города с большим количеством достопримечательностей. В отличие от декартовой плоскости, оси этой сетки ориентированы вниз и вправо. Таким образом, синему узлу-источнику присваиваются координаты (*0, 0*), а красном узлу-стоку присваиваются координаты (*n, m*).

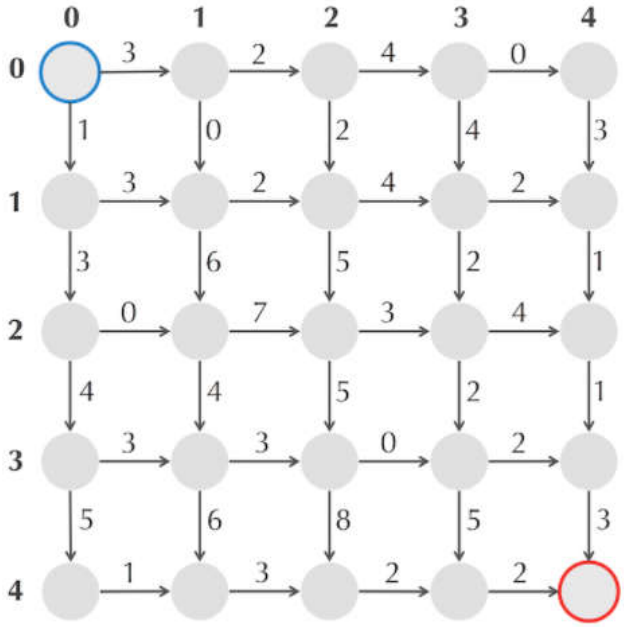


Рис. 5.4. Сетка города размером $n \times m$, представленная в виде графа с взвешенными ребрами для $n=m=4$. Нижний левый узел имеет координаты (4, 0), а верхний правый узел имеет координаты (0, 4).

Это обсуждение подразумевает следующее обобщение исходной задачи.

Задача. Найти самый длинный путь в прямоугольном городе.

Вход: взвешенная прямоугольная сетка $n \times m$ с $n+1$ строками и $m+1$ столбцами.

Выход: самый длинный путь от источника $(0,0)$ к стоку (n, m) в сетке.

Применение подхода полного перебора к решению задачи о Манхэттенском туристе нецелесообразно, потому что количество путей огромно. Разумный жадный подход мог бы выбирать между двумя возможными направлениями на каждом узле (→ или ↓) в зависимости от того, сколько достопримечательностей можно увидеть, если переместиться только на один блок на юг или на один блок на восток. Например, на рисунке 5.5 начнем движение на восток от $(0, 0)$, а не на юг, потому что горизонтальное ребро имеет вес 3, в то время как вертикальное ребро имеет вес 1. К сожалению, эта жадная стратегия может пропустить самый длинный путь в долгосрочной перспективе.

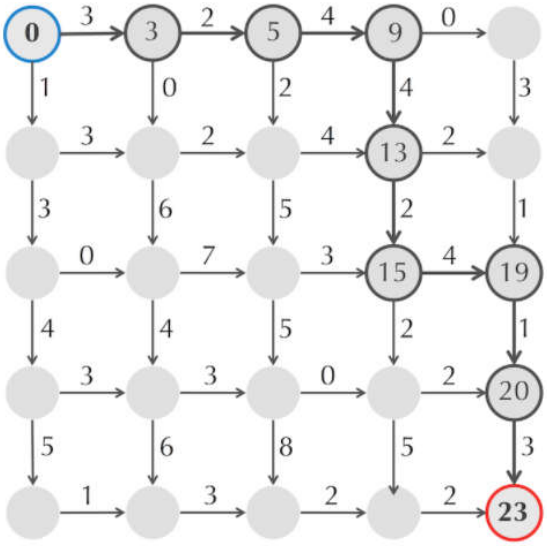


Рис. 5.5. Выделенный путь через этот граф, который найден жадным алгоритмом, не является самым длинным путем.

В действительности, улицы в Мидтауне Манхэттена не образуют идеальной прямоугольной сетки, потому что Бродвей-авеню пересекает сетку по диагонали, но сеть улиц все еще может быть представлена ориентированным графом. На самом деле, задача о Манхэттенском туристе является частным случаем более общей проблемы нахождения самого длинного пути в произвольном ориентированном графе, например, показанного на рисунке 5.6.

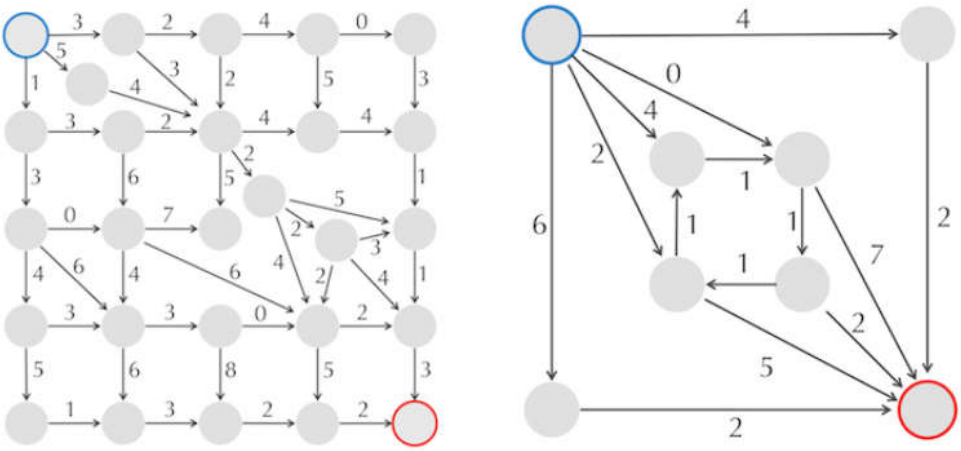


Рис. 5.6. Ориентированные графы, соответствующие гипотетическим нерегулярным городским сетям.

Задача. Найти самый длинный путь между двумя узлами в ориентированном взвешенном графе.

Вход: ориентированный взвешенный граф с источником и стоком.

Выход: самый длинный путь от источника к стоку в ориентированном графе.

Если ориентированный граф содержит направленный цикл (например, четыре центральных ребра веса 1 на рисунке 5.6 справа), то турист мог бесконечно проходить этот цикл, снова и снова пересматривая одни и те же достопримечательности и создавая путь огромной длины. По этой причине графы, которые будут рассмотрены в этой главе, не содержат направленных циклов; такие графы называются ориентированными ациклическими графами (DAG – directed acyclic graph).

Задача. Найти самый длинный путь между двумя узлами в ориентированном DAG.

Вход: взвешенный DAG с источником и стоком.

Выход: самый длинный путь от источника к стоку в DAG.

Добавим два массива целых чисел к выравниванию ATGTTATA и ATCGTCC. Массив [0 1 2 2 3 4 5 6 7 8] содержит количество символов ATGTTATA, использованных до данного столбца в выравнивании. Аналогично, массив [0 1 2 3 4 5 5 6 6 7] содержит количество символов ATCGTCC, использованных до данного столбца.

```
0   1   2   2   3   4   5   6   7   8
  A   T   -   G   T   T   A   T   A
  A   T   C   G   T   -   C   -   C
0   1   2   3   4   5   5   6   6   7
```

Добавим третий массив [↘ ↘ → ↘ ↘ ↓ ↘ ↓ ↘], записывающий, соответствует каждый столбец совпадению/промаху (↘ / ↘), вставке (→) или удалению (↓).

```
0   1   2   2   3   4   5   6   7   8
  A   T   -   G   T   T   A   T   A
  A   T   C   G   T   -   C   -   C
0   1   2   3   4   5   5   6   6   7
    ↘   ↘   →   ↘   ↘   ↓   ↘   ↓   ↘
```

Этот массив соответствует пути от источника к стоку на прямоугольной сетке *8×7*. *I*-й узел этого пути состоит из *i*-го элемента [0 1 2 2 3 4 5 6 7 8] и *i*-го элемента [0 1 2 3 4 5 5 6 6 7]:

(0. 0) ↘ (1, 1) ↘ (2, 2) → (2, 3) ↘ (3, 4) ↘ (4, 5) ↓ (5, 5) ↘ (6, 6) ↓ (7, 6) ↘ (8, 7)

Этот путь показан на рисунке 5.7. В дополнение к горизонтальным и вертикальным ребрам также были добавлены диагональные ребра, соединяющие (i, j) с $(i+1, j+1)$.

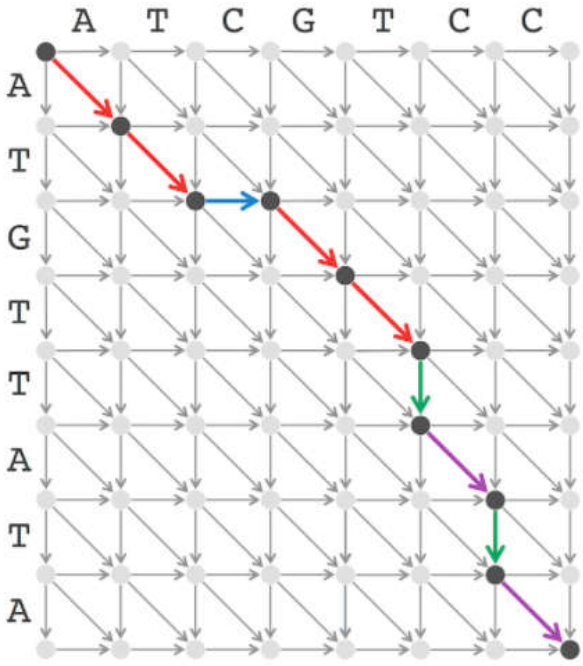


Рис. 5.7. Путь в графе выравнивания ATGTTATA и ATCGTCC.

DAG на рисунке 5.7 называется графом выравнивания строк *v* и *w*, и обозначается *AlignmentGraph*(*v, w*), цветом показан путь от источника к стоку в DAG, обозначающий путь выравнивания. Каждое выравнивание *v* и *w* можно рассматривать как набор инструкций для построения уникального пути выравнивания в *AlignmentGraph*(*v, w*), где каждое соответствие/промах, вставка и удаление соответствуют ребру ↘ / ↘, → и ↓ , соответственно. Кроме того, этот процесс обратим, поскольку можно преобразовать каждый путь выравнивания в уникальное выравнивание.

Поиск самой длинной общей подпоследовательности двух строк эквивалентен поиску выравнивания этих строк, максимизирующих количество совпадений. На рисунке 5.8 выделены все диагональные грани *AlignmentGraph* (*ATGTTATA, ATCGTCC*), соответствующие совпадениям. Если присвоить вес 1 всем этим ребрам и 0 всем другим ребрам, то задача поиска самой длинной подпоследовательности эквивалентна поиску самого длинного пути в этом взвешенном DAG. Таким образом, нужно разработать алгоритм для поиска самого длинного пути в DAG, но для этого нужно больше знать о динамическом программировании, мощной алгоритмической парадигме, которая используется для решения тысяч проблем из разных научных областей.

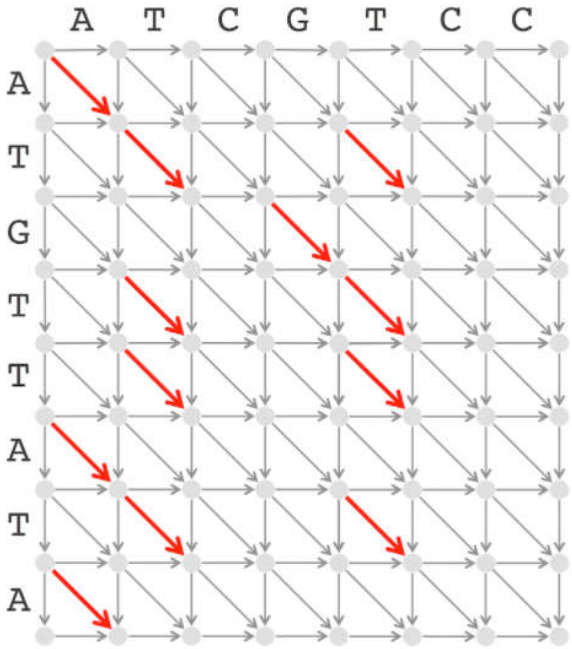


Рис. 5.8. *AlignmentGraph*(*ATGTTATA, ATCGTCC*), где все ребра веса 1 обозначены красным цветом (остальные ребра имеют вес 0). Эти ребра соответствуют потенциальным совпавшим символам в выравнивании двух строк.

Лекция 6. Введение в динамическое программирование. Задача о размене монет.

Представьте, что человек покупает учебник в книжном магазине за 69,24 доллара, наличными он заплатил 70 долларов. Он должен получить сдачу 76 центов, и кассир должен принять решение о том, давать ли ему гору из 76 монет по 1 центу или всего четыре монеты (25 + 25 + 25 + 1 = 76). Получить сдачу в этом примере легко, но он показывает более общую проблему – как кассир может сдать сдачу, используя наименьшее количество монет.

Различные валюты имеют разные возможные номиналы монет. В США номиналами монет являются (100, 50, 25, 10, 5, 1); в Римской Республике они были (120, 40, 30, 24, 20, 10, 5, 4, 1). Эвристика, используемая кассирами во всем мире для выдачи сдачи, которая называется *GreedyChange*, итерационно выбирает монеты наибольшего возможного достоинства.

```
GREEDYCHANGE(money)
  change ← empty collection of coins)
  while money > 0
    coin ← largest denomination that is less than or equal to money
    add a coin with denomination coin to the collection of coins change
    money ← money − coin
  return change
```

Допустим, нужно сдать 48 единиц валюты (называемых денариями) в Древнем Риме. *GreedyChange* возвращает пять монет (48 = 40 + 5 + 1 + 1 + 1), но можно сдать сдачу, используя только две монеты (48 = 24 + 24). Таким образом, *GreedyChange* не оптимален для некоторых номиналов.

Поскольку *GreedyChange* некорректен, нужно разработать другой подход. Можно представить монеты из d произвольных номиналов массивом целых чисел $Coins = (coin_1, \ldots, coin_d)$, где значения $coin_i$ приведены в порядке убывания. Массив из d положительных целых чисел $(change_1, \ldots, change_d)$ с количеством монет $change_1+\ldots+change_d$ разменивает целое число *money* (для номиналов *Coins*), если

$$coin_1 \cdot change_1 + \cdots + coin_d \cdot change_d = money$$

Например, для римских номиналов *Coins* = (120, 40, 30, 24, 20, 10, 5, 4, 1) оба набора (0, 1, 0, 0, 0, 0, 0, 1, 0, 3) и (0, 0, 0, 2, 0, 0, 0, 0, 0) разменивают *money*=48.

Рассмотрим проблему поиска минимального количества монет, необходимых для размена. Пусть *MinNumCoins*(*money*) обозначает минимальное количество монет, необходимых для размена *money* для данного набора номиналов (например, для римских номиналов, *MinNumCoins*(48) = 2).

Задача. Найти минимальное количество монет, необходимых для размена.

Вход: целое число *money* и массив *Coins* из d положительных целых чисел.

Выход: минимальное количество монет с номиналами *Coins*, которые разменивают *money*.

Поскольку жадное решение, используемое римскими кассирами для решения проблемы размена, неверно, рассмотрим другой подход. Предположим, что нужно разменять 76 динариев, и имеются только монеты трех наименьших номиналов: *Coins* = (5, 4, 1). Минимальный набор монет на сумму 76 денариев должен быть одним из следующих:

- минимальный набор монет общей стоимостью 75 денариев, плюс монета в 1 динарий;

- минимальный набор монет общей стоимостью 72 денария, плюс монета в 4 динария;
- минимальный набор монет общей стоимостью 71 денарий, плюс монета в 5 динариев.

Для номиналов $Coins = (coin_1, \ldots, coin_d)$, $MinNumCoins(money)$ равно минимуму из d чисел:

$$MinNumCoins(money) = \min \begin{Bmatrix} MinNumCoins(money - coin_1) + 1, \\ \ldots, \\ MinNumCoins(money - coin_d) + 1 \end{Bmatrix}.$$

Получается рекуррентное соотношение или уравнение для $MinNumCoins(money)$ в терминах $MinNumCoins(m)$ для меньших значений m.

Классическим примером рекуррентного решения является задача о Ханойских башнях. Головоломка «Ханойские башни» состоит из трех вертикальных колышков и нескольких дисков разного размера, каждый с отверстием в центре, чтобы он соответствовал колышкам. Диски изначально складываются на левый колышек (колышек 1), так что диски увеличиваются в размере сверху вниз (см. Рисунок 6.1). За раз можно передвигать диск между позициями, с целью перемещения всех дисков с левой позиции (колышек 1) на правую (колышек 3). Однако, не разрешается размещать больший диск поверх меньшего диска.

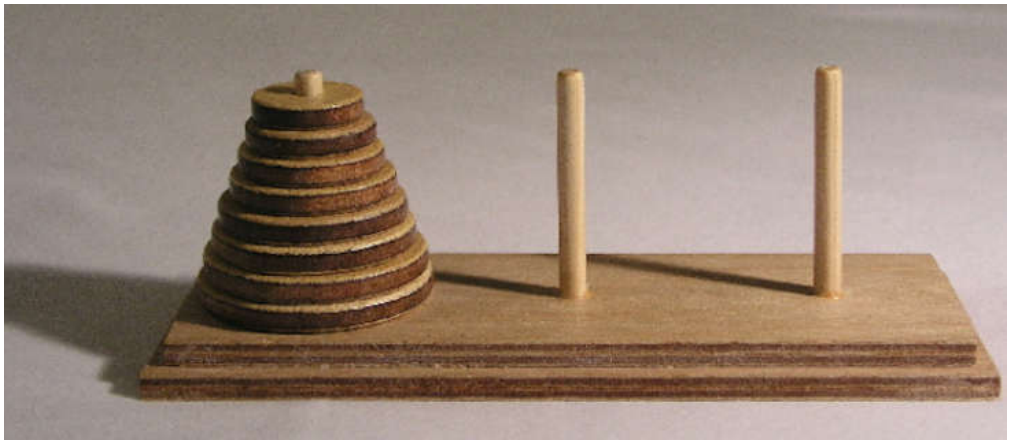


Рис. 6.1. Головоломка «Ханойские башни».

Задача. Решить задачу Ханойских башен для любого количества дисков.

Вход: целое число n.

Выход: последовательность ходов, которая разрешит головоломку «Ханойские башни» с n дисками.

Рассмотрим, сколько шагов требуется для решения головоломки «Ханойские башни» для четырех дисков. Первое важное замечание состоит в том, что рано

или поздно придется переместить самый большой диск в нужное положение. Однако, чтобы переместить самый большой диск, сначала нужно перенести все три самых маленьких диска с первого колышка. Кроме того, эти три самых маленьких диска должны быть на одном и том же колышке, потому что самый большой диск нельзя разместить поверх другого диска. Таким образом, сначала нужно перенести три верхних диска на средний колышек (7 ходов), затем переместить самый большой диск на правый колышек (1 шаг), затем снова переместить три самых маленьких диска со среднего колышка на самый большой диск на правом колышке (еще 7 ходов), всего 15 ходов.

В общем случае пусть T(*n*) обозначает минимальное количество шагов, необходимых для решения головоломки «Ханойские башни» с *n* дисками. Чтобы переместить *n* дисков с левого колышка вправо, сначала нужно перенести *n-1* самых маленьких дисков с левого колышка на средний (T(*n-1*)), затем переместить самый большой диск вправо (1 шаг) и, наконец, переместить *n-1* самых маленьких дисков из середины вправо (T(*n-1*)). Это дает рекуррентное соотношение

$$T(n) = 2T(n-1) + 1$$

Теперь есть рекурсивный алгоритм для перемещения *n* дисков из левого положения в правое. Будут использованы три переменные (каждая из которых имеет различное значение от 1, 2 и 3), чтобы обозначить три положения (начальное, конечное и промежуточное): *startPeg*, *destinationPeg* и *transitPeg*. Эти три переменные всегда представляют разные положения, и поэтому сумма *startPeg* + *destinationPeg* + *transitPeg* всегда равна 1+2+3 = 6. *HanoiTowers*(*n, startPeg, destinationPeg*) перемещает *n* дисков из *startPeg* в *destinationPeg* (с использованием *transitPeg* в качестве временного адресата).

```
HANOITOWERS(n, startPeg, destinationPeg)
    if n = 1
        Move top disk from startPeg to destinationPeg
        return
    transitPeg = 6 - startPeg - destinationPeg
    HANOITOWERS(n - 1, startPeg, transitPeg)
    Move top disk from startPeg to destinationPeg
    HANOITOWERS(n - 1, transitPeg, destinationPeg)
    return
```

Несмотря на то, что этот алгоритм может показаться простым, перемещение 100-дисковой башни потребует больше шагов, чем число атомов во Вселенной. Быстрый рост количества ходов, требуемых *HanoiTowers*, объясняется тем, что каждый раз, когда *HanoiTowers* вызывается для *n* дисков, он вызывает себя дважды для *n-1*, что, в свою очередь, влечет четыре вызова для *n-2* и т.д. Например, вызов *HanoiTowers*(*4, 1, 3*) приводит к вызовам *HanoiTowers*(*3, 1, 2*)

и *HanoiTowers*(*3, 2, 3*); эти вызовы, в свою очередь, вызывают *HanoiTowers*(*2, 1, 3*), *HanoiTowers*(*2, 3, 2*), *HanoiTowers*(*2, 2, 1*) и *HanoiTowers*(*2, 1, 3*).

Для задачи размена монет можно получить следующий рекурсивный алгоритм (т.е. алгоритм, который может вызывать самого себя), который решает проблему размера путем вычисления *MinNumCoins*(*m*) для меньших и меньших значений *m*. В этом алгоритме |*Coins*| означает число номиналов в *Coins*.

```
RecursiveChange(money, Coins)
  if money = 0
    return 0
  MinNumCoins ← ∞
  for i ← 1 to |Coins|
    if money ≥ coin_i
      NumCoins ← RecursiveChange(money − coin_i, Coins)
      if NumCoins + 1 < MinNumCoins
        MinNumCoins ← NumCoins + 1
  return MinNumCoins
```

RecursiveChange может показаться эффективным, но он непрактичен, поскольку он пересчитывает оптимальную комбинацию монет для заданного значения *money* снова и снова. Например, когда *money* = 76 и *Coins* = (5, 4, 1), *MinNumCoins*(70) вычисляется шесть раз, пять из которых показаны на рисунке 6.2. Это может показаться не проблемой, но *MinNumCoins*(30) будет вычисляться миллиарды раз.

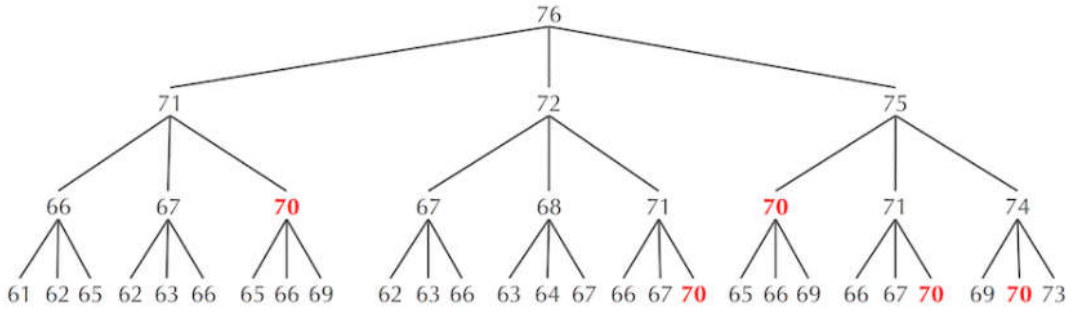


Рис. 6.2. Дерево, иллюстрирующее вычисление *MinNumCoins*(76), для номиналов *Coins* = (5, 4, 1). Ребра этого дерева представляют рекурсивные вызовы *RecursiveChange* для разных входных значений, пять из шести вычислений *MinNumCoins*(70) выделены красным цветом. Когда *MinNumCoins*(70) вычисляется в шестой раз (соответствует траектории 76 → 75 → 74 → 73 → 72 → 71 → 70), *RecursiveChange* уже вызывается сотни раз.

Чтобы избежать множества рекурсивных вызовов, необходимых для вычисления *MinNumCoins*(*money*), можно использовать динамическое программирование. Было бы полезно знать все значения *MinNumCoins*(*money* – $coin_i$) к моменту вычисления *MinNumCoins*(*money*). Вместо трудоемких

вызовов *RecursiveChange* (*money* – $coin_i$, *Coins*) можно было бы искать значения *MinNumCoins*(*money* – $coin_i$) в массиве и таким образом вычислять *MinNumCoins*(*money*), используя только |*Coins*| сравнений.

Ключом к динамическому программированию является шаг, который может показаться нелогичным. Вместо вычисления *MinNumCoins*(m) для каждого значения m от 76 до m = 1 с помощью рекурсивных вызовов инвертируем алгоритм и вычислим *MinNumCoins*(m) от m=1 до 76, сохраняя все полученные значения в массиве так, чтобы что нужно было вычислить *MinNumCoins*(m) только один раз для каждого значения m. *MinNumCoins*(m) по-прежнему вычисляется по тому же рекуррентному соотношению:

$$MinNumCoins(m) = \min \begin{Bmatrix} MinNumCoins(m-5)+1, \\ MinNumCoins(m-4)+1, \\ MinNumCoins(m-1)+1 \end{Bmatrix}.$$

Например, предположим, уже вычислено *MinNumCoins*(m) для $m<7$, *MinNumCoins*(6) на единицу больше минимума из *MinNumCoins*(6-5)=1, *MinNumCoins*(6-4)=2 и *MinNumCoins*(6 - 1)=1. Таким образом, *MinNumCoins*(6) равно 1+1=2. В свою очередь, *MinNumCoins*(7) на единицу больше минимума из *MinNumCoins*(7 - 5) = 2, *MinNumCoins*(7 - 4) = 3 и *MinNumCoins*(7 - 1) = 2. В результате *MinNumCoins*(7) равно 2+1=3. И так далее, в результате получается следующая таблица.

m	0	1	2	3	4	5	6	7	8	9	10	11	12
MinNumCoins(m)	0	1	2	3	1	1	2	3	2	2	2	3	3

Нужно обратить внимание, что *MinNumCoins*(2) используется при вычислении как *MinNumCoins*(6), так и *MinNumCoins*(7), но вместо того, чтобы использовать вычислительные ресурсы, вынуждая вычислять это значение с нуля, найдем предварительно вычисленное значение в массиве. Следующий алгоритм динамического программирования вычисляет *MinNumCoins*(*money*) за время $O(money \cdot |Coins|)$.

```
DPCHANGE(money, Coins)
 MinNumCoins(0) ← 0
 for m ← 1 to money
   MinNumCoins(m) ← ∞
     for i ← 1 to |Coins|
       if m ≥ coin_i
         if MinNumCoins(m - coin_i) + 1 < MinNumCoins(m)
           MinNumCoins(m) ← MinNumCoins(m - coin_i) + 1
 output MinNumCoins(money)
```

Задача 6.1. Реализовать *DPChange*.

Вход: целое число *money* и массив $Coins = (coin_1, \ldots, coin_d)$.

Выход: минимальное количество монет с номиналами *Coins*, которые разменивают *money*.

Пример входа:

40

50,25,20,10,5,1

Пример выхода:

2

Теперь можно реализовать алгоритм, решающий задачу о Манхэттенском туристе. Следующий псевдокод вычисляет длину самого длинного пути к узлу (i, j) на прямоугольной сетке, основываясь на наблюдении, что единственный способ достичь узла (i, j) в задаче о Манхэттенском туристе – либо двигаться на юг (↓) из $(i - 1, j)$ или на восток (→) из $(i, j - 1)$.

```
SouthOrEast(i, j)
  if i = 0 and j = 0
    return 0
  x ← −∞, y ← −∞
  if i > 0
    x ← SouthOrEast(i − 1, j) + weight of the vertical edge into (i, j)
  if j > 0
    y ← SouthOrEast(i, j − 1) + weight of the horizontal edge into (i, j)
  return max{x, y}
```

Аналогично *RecursiveChange*, *SouthOrEast* страдает от огромного количества рекурсивных вызовов, и нужно пересмотреть этот алгоритм с точки зрения динамического программирования. Чтобы найти длину самого длинного пути из источника $(0, 0)$ в сток (n, m), сначала найдем длины самых длинных путей от источника ко всем узлам (i, j) в сетке, медленно расширяясь от источника. На первый взгляд, можно подумать, что придется решать $n \times m$ различных задач вместо одной. Однако *SouthOrEast* также решает все эти меньшие проблемы, как *RecursiveChange* и *DPChange* вычисляют *MinNumCoins*(m) для всех значений $m<money$. Идеей динамического программирования является решение меньших задач один раз, а не миллиарды раз.

В дальнейшем будем обозначать длину самого длинного пути от $(0, 0)$ до (i, j) как $s_{i,j}$. Вычислить $s_{0,j}$ (для $0 \leq j \leq m$) легко, так как можно достичь $(0, j)$ только двигаясь вправо (→) и не имея никакой гибкости в выборе пути. Таким образом, $s_{0,j}$ представляет собой сумму весов первых j горизонтальных ребер, выходящих из источника. Аналогично, $s_{i,0}$ – сумма весов первых i вертикальных ребер, идущих от источника. См. Рисунок 6.3.

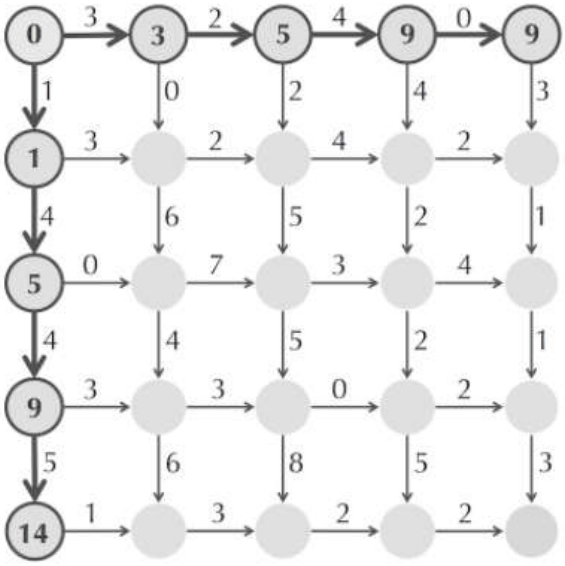


Рис. 6.3. Вычислить $s_{i,0}$ и $s_{0,j}$ легко, поскольку есть только один путь от источника до $(i,0)$ и только один путь от источника до $(0,j)$.

Для $i>0$ и $j>0$ единственным способом достижения узла (i, j) является переход вниз от узла $(i-1, j)$ или перемещение вправо от узла $(i, j-1)$. Таким образом, $s_{i,j}$ можно вычислить как максимум из двух значений:

- $s_{i-1,j}$ + вес вертикального ребра от $(i - 1, j)$ до (i, j)
- $s_{i,j-1}$ + вес горизонтального ребра от $(i, j - 1)$ до (i, j)

Теперь, когда вычислены вычислили $s_{0,1}$ и $s_{1,0}$, можно вычислить $s_{1,1}$. В (1, 1) можно прийти, перемещаясь вниз из (0, 1) или вправо из (1, 0). Следовательно, $s_{1,1}$ – максимум двух значений:

- $s_{0,1}$ + вес вертикального ребра от (0, 1) до (1, 1) = 3 + 0 = 3
- $s_{1,0}$ + вес горизонтального ребра от (1, 0) до (1, 1) = 1 + 3 = 4

Поскольку цель – найти самый длинный путь от (0, 0) до (1, 1), заключаем, что $s_{1,1} = 4$. Поскольку было выбрано горизонтальное ребро от (1, 0) до (1, 1) самый длинный путь через (1, 1) должен использовать это ребро, который выделено на рисунке 107 слева. Подобная логика позволяет вычислить остальную часть значений в столбце 1; для каждого $s_{i,1}$ выделяется выбранное ребро, ведущее в $(i, 1)$, как показано на рисунке 6.4 справа.

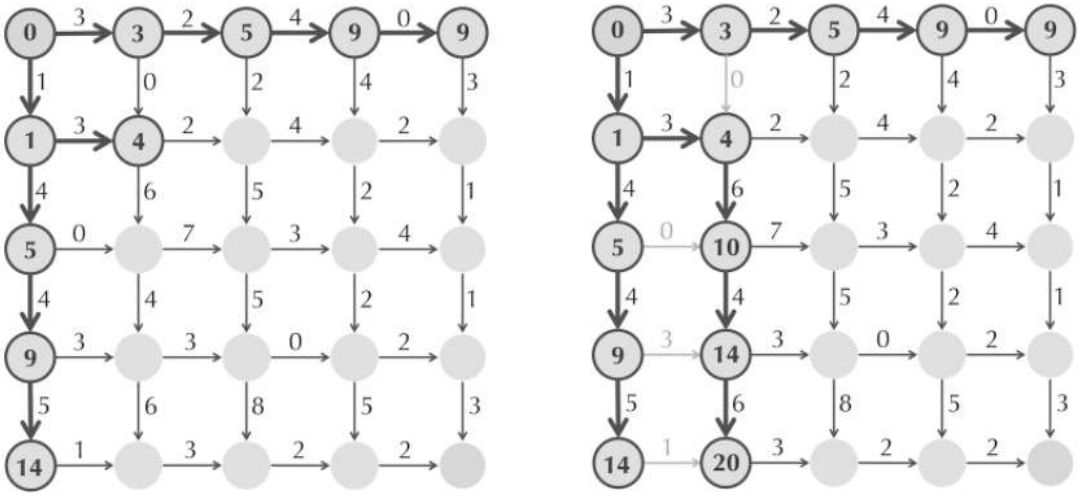


Рис. 6.4. (слева). Вычисление $s_{1,1}$ использует горизонтальное ребро от (1, 0), которое выделено. (справа) Вычисление всех значений $s_{i,1}$ в столбце 1.

Продолжая столбец за столбцом, как показано на рисунке 6.5, можно вычислить каждую оценку $s_{i,j}$ за один проход по графу, в конечном счете вычисляя $s_{4,4} = 34$.

Для каждого узла (i, j) выделим ребро, ведущее в (i, j), которое было использовано для вычисления $s_{i,j}$. Однако, при вычислении $s_{3,3}$:

$$s_{3,3} = \max\begin{cases} s_{2,3} + weight\ of\ the\ vertical\ edge\ from\ (2,3)\ to\ (3,3) = 20 + 2 = 22 \\ s_{3,2} + weight\ of\ the\ horizontal\ edge\ from\ (3,2)\ to\ (3,3) = 22 + 0 = 22 \end{cases}$$

Чтобы достичь (3, 3), можно использовать либо горизонтальное, либо вертикальное входящее ребро, и поэтому оба этих ребра выделены в графе на рисунке 108.

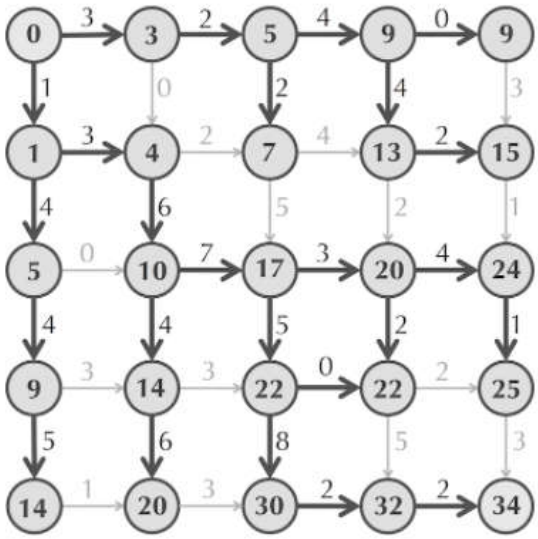


Рис. 6.5. Граф, отображающий все оценки $s_{i,j}$.

Теперь есть схема алгоритма динамического программирования для нахождения длины самого длинного пути в задаче о Манхэттенском туристе, называемой *ManhattanTourist*. В следующем псевдокоде, $down_{i,j}$ и $right_{i,j}$ – соответствующие веса вертикальных и горизонтальных ребер, входящих в узел (i, j). Матрицы, содержащие $(down_{i,j})$ и $(right_{i,j})$, обозначаются как *Down* и *Right* соответственно.

```
MANHATTANTOURIST(n, m, Down, Right)
  s_{0,0} ← 0
  for i ← 1 to n
    s_{i,0} ← s_{i-1,0} + down_{i,0}
  for j ← 1 to m
    s_{0,j} ← s_{0,j-1} + right_{0,j}
  for i ← 1 to n
    for j ← 1 to m
      s_{i,j} ← max{s_{i-1,j} + down_{i,j}, s_{i,j-1} + right_{i,j}}
  return s_{n,m}
```

Задача 6.2. Найдите длину самого длинного пути в задаче о Манхэттенском туристе.

Вход: целые числа n и m, матрица размера $n \times (m+1)$ *Down* и матрица размера $(n+1) \times m$ *Right*. Две матрицы разделены символом -.

Выход: длина самого длинного пути из источника $(0, 0)$ в сток (n, m) на прямоугольной сетке $n \times m$, ребра которой определяются матрицами *Down* и *Right*.

Пример входа:

```
4 4
1 0 2 4 3
4 6 5 2 1
4 4 5 2 1
5 6 8 5 3
-
3 2 4 0
3 2 4 2
0 7 3 3
3 3 0 2
1 3 2 2
```

Пример выхода:

34

После решения задачи о Манхэттенском туристе с помощью динамического программирования, можно адаптировать *ManhattanTourist* для графов выравнивания с диагональными ребрами.

Различные приложения задачи выравнивания намного сложнее, чем задача поиска самой длинной общей подпоследовательности, и требуют создания DAG с соответствующим образом выбранным весом ребер, чтобы смоделировать специфику биологической проблемы. Вместо того чтобы рассматривать каждое последующее приложение выравнивания как новую задачу, можно создать универсальный алгоритм динамического программирования, который найдет самый длинный путь в любом DAG. Более того, многие проблемы биоинформатики не имеют никакого отношения к выравниванию, но они также могут быть решены как приложения задачи поиска самого длинного пути в DAG.

Для заданного узла b в DAG, пусть s_b обозначает длину самого длинного пути от источника до b. Узел a называется предшественником (predecessor) узла b, если есть ребро, соединяющее a и b в DAG; полустепень захода узла равна числу его предшественников. Оценка s_b узла b с индексом k вычисляется как максимум из k элементов:

$$s_b = \max_{all\ predecessors\ a\ of node\ b} \{s_a + weight\ of\ edge\ from\ a\ to\ b\}.$$

Например, на графе, показанном на рисунке 6.6, узел (1,1) имеет три предшественника. До (1, 1) можно добраться, перемещаясь прямо от (1, 0), вниз от (0, 1) или по диагонали от (0, 0), Предполагая, что $s_{0,0}$, $s_{0,1}$, и $s_{1,0}$ уже вычислены, можно вычислить $s_{1,1}$ как максимум трех значений.

$$s_{1,1} = \max \begin{cases} s_{0,1} + \text{weight of edge} \downarrow \text{connecting } (0,1) \text{ to } (1,1) = 3+0=3 \\ s_{1,0} + \text{weight of edge} \rightarrow \text{connecting } (1,0) \text{ to } (1,1) = 1+3=4 \\ s_{0,0} + \text{weight of edge} \searrow \text{connecting } (0,0) \text{ to } (1,1) = 0+5=5 \end{cases}$$

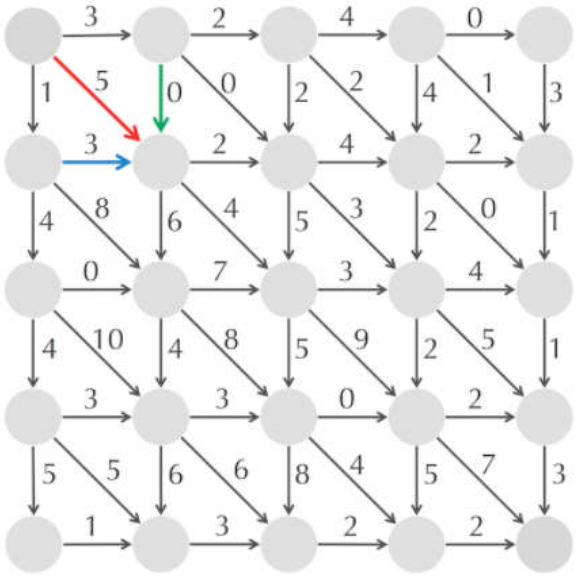


Рис. 6.6. В DAG узел (1, 1) имеет три предшественника ((0, 0), (0, 1) и (1, 0)), которые используются при вычислении $s_{1,1}$.

Аналогичный аргумент может быть применен к графу выравнивания (рисунок 6.7) для вычисления длины LCS между последовательностями v и w. Так как в этом случае все ребра имеют вес 0, за исключением диагональных ребер веса 1, которые представляют совпадения ($v_i = w_j$), получается следующая рекурсия для вычисления длины LCS.

$$s_{i,j} = \max \begin{cases} s_{i-1,j} & +0 \\ s_{i,j-1} & +0 \\ s_{i-1,j-1} & +1 \text{ if } v_i = w_j \end{cases}$$

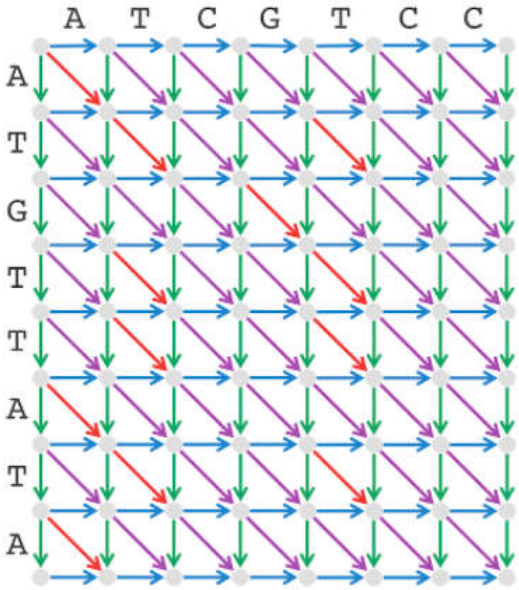


Рис. 6.7. Граф выравнивания последовательностей ATGTTATA и ATCGTCC.

Использование динамического программирования для нахождения длины самого длинного пути в DAG заключается в том, что нужно принять решение о порядке посещения узлов при вычислении значений s_b в соответствии с рекурсией

$$s_b = \max_{all\ predecessors\ a\ of node\ b} \{s_a + weight\ of\ edge\ from\ a\ to\ b\}.$$

Это упорядочение узлов важно, так как к тому времени, когда достигнем узла b, значения s_a для всех его предшественников должны быть уже вычислены. Для прямоугольных сеток порядок, в котором вычисляются $s_{i,j}$, гарантировал, что узел никогда не будет рассмотрен перед посещением всех его предшественников.

Чтобы найти самый длинный путь в произвольном DAG, сначала нужно упорядочить узлы DAG, чтобы каждый узел шёл после всех своих предшественников. Формально упорядочение узлов $(a_1,...,a_k)$ в DAG называется топологическим упорядочением, если каждое ребро (a_i, a_j) DAG связывает узел с меньшим индексом с узлом с большим индексом, т.е., $i < j$.

Причина, по которой *ManhattanTourist* способен найти самый длинный путь на прямоугольной сетке, заключается в том, что его псевдокод неявно использует порядок, аналогичный проходу «по столбцу», показанному на рисунке 6.8 слева. Топологическое упорядочение «по строкам», показанное на рисунке 6.8 справа, дает еще одно топологическое упорядочение прямоугольной сетки.

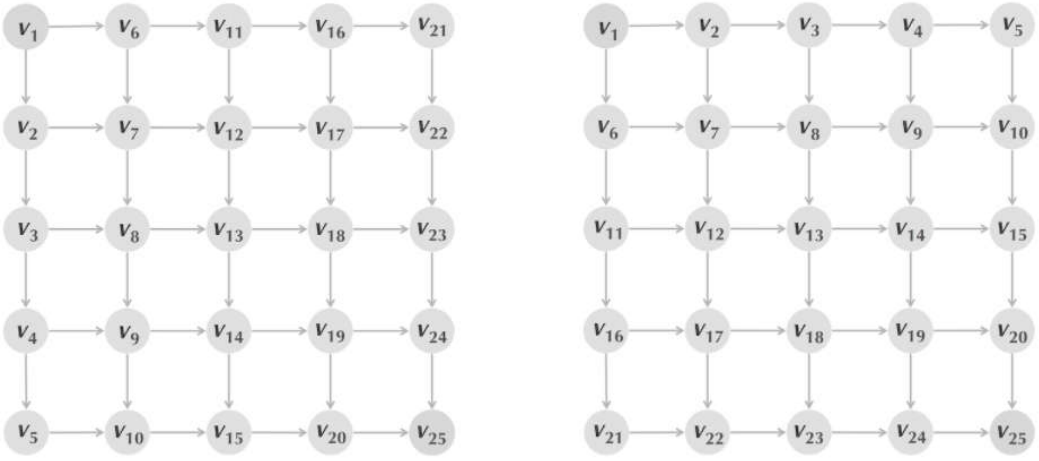


Рис. 6.8. Столбцовое (слева) и строковое (правые) топологические упорядочения прямоугольной сетки.

Любой DAG имеет топологическое упорядочение и это топологическое упорядочение может быть построено за время, пропорциональное числу ребер в графе.

Первые приложения топологического упорядочения были результатом крупных проектов управления в попытке запланировать последовательность задач на основе их зависимостей. В этих проектах задачи представлены узлами,

а ребро соединяет узел *a* с узлом *b*, если задача *a* должна быть завершена до того, как задача *b* будет запущена.

Следующий алгоритм построения топологического упорядочения основан на наблюдении, что каждый DAG имеет по крайней мере один узел без входящих ребер. Будем отмечать один из этих узлов как v_1, а затем удалять этот узел из графа вместе со всеми его исходящими ребрами. Полученный граф также является DAG, который, в свою очередь, должен иметь узел без входящих ребер; маркируем этот узел v_2 и снова удаляем его из графа вместе с его исходящими ребрами. Результирующий алгоритм продолжается до тех пор, пока все узлы не будут удалены, создавая топологический порядок v_1, ..., v_n. Этот алгоритм работает за время, пропорциональное количеству ребер во входном DAG.

```
TopologicalOrdering(Graph)
  List ← empty list
  Candidates ← set of all nodes in Graph with no incoming edges
  while Candidates is non-empty
    select an arbitrary node a from Candidates
    add a to the end of List and remove it from Candidates
    for each outgoing edge from a to another node b
      remove edge (a, b) from Graph
      if b has no other incoming edges
        add b to Candidates
  if Graph has edges that have not been removed
    return "the input graph is not a DAG"
  else return List
```

Задача 6.3. Реализовать *TopologicalOrdering*.

Вход: список смежности графа (с узлами, представленными целыми числами).

Выход: топологическое упорядочение этого графа.

Пример входа:

0 -> 1

1 -> 2

3 -> 1

4 -> 2

Пример выхода:

0, 3, 4, 1, 2

Как только имеется топологическое упорядочение, можно вычислить длину самого длинного пути от источника к стоку, посетив узлы DAG в порядке, определяемом топологическим упорядочением, что достигается в

соответствии со следующим алгоритмом. Для простоты предположим, что исходный узел является единственным узлом с индексом 0 в *Graph*.

```
LongestPath(Graph, source, sink)
  for each node b in Graph
    s_b ← -∞
  s_source ← 0
  topologically order Graph
  for each node b in Graph (following the topological order)
    s_b ← max_{all predecessors a of node b} {s_a + weight of edge from a to b}
  return s_sink
```

Поскольку каждое ребро участвует только в одном уровне рекурсии, время выполнения *LongestPath* пропорционально количеству ребер в DAG *Graph*.

Теперь можно эффективно вычислить длину самого длинного пути в произвольном DAG, но пока неизвестно, как преобразовать *LongestPath* в алгоритм, который будет строить этот самый длинный путь.

Лекция 7. Поиск с возвратом.

Каждое ребро, выбранное *ManhattanTourist*, выделено в графе на рисунке 7.1.

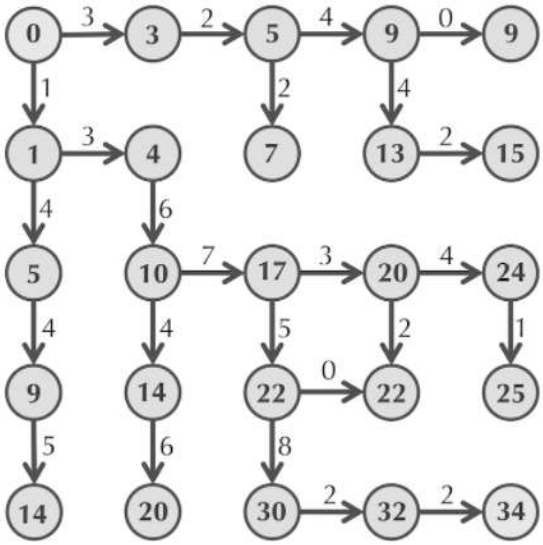


Рис. 7.1. Ребра в графе, выбранные *ManhattanTourist*.

Чтобы сформировать самый длинный путь, нужно найти путь от источника к стоку, образованный выделенными ребрами (может существовать более одного такого пути). Однако, если нужно пройти от источника к стоку вдоль выделенных ребер, можно достичь тупика, например, узла (1, 2). Напротив, каждый путь от стока ведет к источнику, если возвращаться назад в

направлении, противоположном каждому выделенному ребру, как показано на рисунке 7.2.

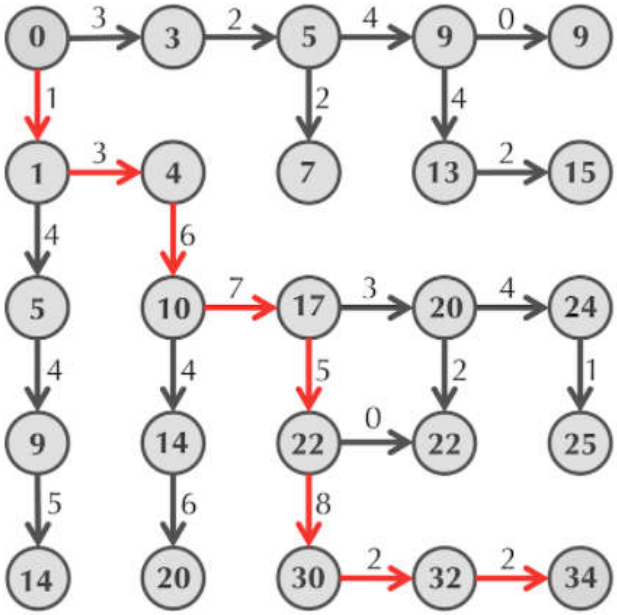


Рис. 7.2. Показанные ребра от стока к источнику создают красный путь в DAG, который выделяет самый длинный путь от источника к стоку.

Можно использовать эту идею возврата для построения LCS строк *v* и *w*. Если присвоить единичные веса ребрам в *AlignmentGraph*(*v*, *w*), соответствующим совпадениям, и нулевые веса всем остальным ребрам, то $s_{|v|,|w|}$ дает длину LCS. Следующий алгоритм поддерживает запись того, какое ребро использовалось для вычисления каждого значения $s_{i,j}$, используя указатели обратного отслеживания, которые принимают одно из трех значений ↓, → или ↘. Указатели обратного отслеживания сохраняются в матрице *Backtrack*.

```
LCSBacktrack(v, w)
  for i ← 0 to |v|
    s_{i,0} ← 0
  for j ← 0 to |w|
    s_{0,j} ← 0
  for i ← 1 to |v|
    for j ← 1 to |w|
      s_{i,j} ← max{s_{i-1,j}, s_{i,j-1}, s_{i-1,j-1} + 1 (if v_i = w_j)}
      if s_{i,j} = s_{i-1,j}
        Backtrack_{i,j} ← "↓"
      else if s_{i,j} = s_{i,j-1}
        Backtrack_{i,j} ← "→"
      else if s_{i,j} = s_{i-1,j-1} + 1 and v_i = w_j
        Backtrack_{i,j} ← "↘"
  return Backtrack
```

Теперь нужно найти путь от источника к стоку, образованной выделенными ребрами. Следующий алгоритм решает задачу самой длинной общей подпоследовательности, используя информацию из *Backtrack*. *OutputLCS*(*Backtrack*, *v*, *i*, *j*) выводит LCS между *i*-префиксом *v* и *j*-префиксом *w*. Первоначальным вызовом, выводящим LCS *v* и *w*, является *OutputLCS*(*Backtrack*, *v*, |*v*|, |*w*|).

```
OutputLCS(Backtrack, v, i, j)
  if i = 0 or j = 0
    return
  if backtrack_{i,j} = "↓"
    OutputLCS(Backtrack, v, i - 1, j)
  else if Backtrack_{i,j} = "→"
    OutputLCS(Backtrack, v, i, j - 1)
  else
    OutputLCS(Backtrack v, i - 1, j - 1)
    output v_i
```

Задача 7.1. Используя *OutputLCS*, решить задачу поиска самой длинной общей подпоследовательности.

Вход: две строки *s* и *t*.

Выход: самая длинная общая подпоследовательность *s* и *t*. (Примечание: может существовать несколько решений, и в этом случае можно вывести любое из них).

Пример входа:

AACCTTGG

ACACTGTGA

Пример выхода:

AACTGG

Метод поиска с возвратом может быть обобщен для построения самого длинного пути в любом DAG. Всякий раз, когда вычисляется s_b как

$$s_b = \max_{all\ predecessors\ a\ of node\ b} \{s_a + weight\ of\ edge\ from\ a\ to\ b\},$$

нужно сохранить предшественника b, который использовался при вычислении s_b, чтобы потом можно было вернуться назад. Теперь можно использовать поиск с возвратом, чтобы найти самый длинный путь в произвольном DAG.

Задача 7.2. Решить задачу поиска самого длинного пути в DAG.

Вход: целое число, представляющее узел-источник графа, целое число, представляющее узел-сток графа, список ребер в графе. Обозначение ребра как 0->1:7 показывает, что ребро соединяет узел 0 с узлом 1 с весом 7.

Выход: длина самого длинного пути в графе, самый длинный путь. (Если существует несколько таких путей, можно вернуть любой).

Пример входа:

0

4

0->1:7

0->2:4

2->3:2

1->4:1

3->4:3

Пример выхода:

9

0->2->3->4

Нужно напомнить выравнивание Марахиеля кодировки А-доменов для *Asp* и *Orn*, у которой было 19+10 совпадений:

YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGITYIKLTPSLFHTIVNTASFAFDANFESLRLIVLGGEKIIPIDVIAFRKMYGHTE-FINHYGPTEATIGA

-AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYI-YEQKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS

Можно построить выравнивание, имеющее больше совпадений за счет введения большего количества инделей. Тем не менее, чем больше инделей

добавляется, тем менее биологически релевантным становится выравнивание, поскольку оно все больше и больше расходится с биологически правильным выравниванием, найденным Марахиелем. Ниже приведено выравнивание с максимальным количеством совпадений, представляющих LCS длиной 19 + 8 + 19 = 46. Это выравнивание настолько велико, что его нельзя поместить в одну строку.

```
YAFDL--G-YTCMFP--VLL-GGGELHIV---Q-K-E--T-YTAPDEIAHYIK----EHGITYI---KLTPSL-FHT
-AFDVSAGD----FARA-LLTGG-QL-IVCPNEVKMDPASLY-A---I---IKKYD----IT-IFEA--TPALV---

 IVNTASFAFDANFE-----S-LR-LIVLGG-----EKIIPIDVIAFRK-M---YGHTEFI---NHYGPTEATIGA
 IPLMEYIY-----EQKLDISQLQILIV-GSDSCSME-----D---F-KTLVSRFGST--IRIVNSYGVTEACIDS
```

Ниже выделены фиолетовые аминокислоты, представляющие нерибосомальные сигнатуры. Несмотря на то, что эти сигнатуры сгруппированы в восемь консервативных столбцов в выравнивании Марахиеля, только пять из этих столбцов «выжили» в выравнивании LCS, что делает невозможным вывод нерибосомальных сигнатур:

```
YAFDL--G-YTCMFP--VLL-GGGELHIV---Q-K-E--T-YTAPDEIAHYIK--EHGITYI---KLTPSL-FHT
-AFDVSAGD----FARA-LLTGG-QL-IVCPNEVKMDPASLY-A---I---IKKYD--IT-IFEA--TPALV---

IVNTASFAFDANFE-----S-LR-LIVLGG-----EKIIPIDVIAFRK-M---YGHTEFI---NHYGPTEATIGA
IPLMEYIY-----EQKLDISQLQILIV-GSDSCSME-----D---F-KTLVSRFGST--IRIVNSYGVTEACIDS
```

Необоснованные совпадения, скрывающие реальный эволюционный сценарий, появились потому, что ничто не мешало вводить чрезмерное количество инделей при построении LCS. Значит, нужен способ наложения штрафов и промахов. Во-первых, обратимся к инделям. В дополнение к присвоению соответствиям премии +1, наделим индели штрафом в -4. Лучшее по оценке выравнивание A-доменов приближается к биологически правильному выравниванию, причем шесть столбцов соответствуют правильно выровненным сигнатурам.

```
YAFDLGYTCMFP-VLL-GGGELHIV-QKETYTAPDEI-AHYIKEHGITYI-KLTPSLFHTIVNTASFAFDANFE
-AFDVS-AGDFARALLTGG-QL-IVCPNEVKMDPASLYA-IIKKYDIT-IFEATPAL--VIPLME-YIYEQKLD

          -S-LR-LIVLGGEKIIPIDVIAFRKM---YGHTE-FINHYGPTEATIGA
          ISQLQILIV-GSDSC-SME--DFKTLVSRFGSTIRIVNSYGVTEACIDS
```

Чтобы обобщить модель оценки выравнивания, по-прежнему присуждаем +1 для соответствий, но также наказываем промахи некоторой положительной константой μ (штраф промаха) и индели некоторой положительной константой σ (штраф инделя). В результате оценка выравнивания равна следующему выражению:

$$\#\text{совпадений} - \mu \cdot \#\text{промахов} - \sigma \cdot \#\text{инделей}$$

Например, для параметров $\mu = 1$ и $\sigma = 2$ для следующего выравнивания получится оценка +1 + 1 - 2 + 1 + 1 - 2 - 1 - 2 - 1 = -4.

```
A T - G T T A T A
A T C G T - C - C
```

Биологи дополнительно уточнили эту функцию стоимости, чтобы учесть, что некоторые мутации могут быть более вероятными, чем другие, что требует различия в штрафах для промахов и инделей в зависимости от конкретных символов. Расширим k-буквенный алфавит, чтобы включить символ пробела, а затем построим $(k+1)\times(k+1)$ матрицу, содержащую оценку выравнивания каждой пары символов. Матрица оценки для сравнения последовательностей ДНК ($k = 4$), когда все промахи подвергаются штрафу μ, а все индели наказываются как σ, показана ниже.

	A	C	G	T	-
A	+1	-μ	-μ	-μ	-σ
C	-μ	+1	-μ	-μ	-σ
G	-μ	-μ	+1	-μ	-σ
T	-μ	-μ	-μ	+1	-σ
-	-σ	-σ	-σ	-σ	

Матрицы для сравнения последовательностей ДНК обычно определяются только параметрами μ и σ, однако, матрицы оценки для сравнения последовательностей белков оценивают различные мутации по-разному. Пример – матрица оценки PAM_{250}, показана на рисунке 7.3.

	A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y	–
A	2	-2	0	0	-3	1	-1	-1	-1	-2	-1	0	1	0	-2	1	1	0	-6	-3	-8
C	-2	12	-5	-5	-4	-3	-3	-2	-5	-6	-5	-4	-3	-5	-4	0	-2	-2	-8	0	-8
D	0	-5	4	3	-6	1	1	-2	0	-4	-3	2	-1	2	-1	0	0	-2	-7	-4	-8
E	0	-5	3	4	-5	0	1	-2	0	-3	-2	1	-1	2	-1	0	0	-2	-7	-4	-8
F	-3	-4	-6	-5	9	-5	-2	1	-5	2	0	-3	-5	-5	-4	-3	-3	-1	0	7	-8
G	1	-3	1	0	-5	5	-2	-3	-2	-4	-3	0	0	-1	-3	1	0	-1	-7	-5	-8
H	-1	-3	1	1	-2	-2	6	-2	0	-2	-2	2	0	3	2	-1	-1	-2	-3	0	-8
I	-1	-2	-2	-2	1	-3	-2	5	-2	2	2	-2	-2	-2	-2	-1	0	4	-5	-1	-8
K	-1	-5	0	0	-5	-2	0	-2	5	-3	0	1	-1	1	3	0	0	-2	-3	-4	-8
L	-2	-6	-4	-3	2	-4	-2	2	-3	6	4	-3	-3	-2	-3	-3	-2	2	-2	-1	-8
M	-1	-5	-3	-2	0	-3	-2	2	0	4	6	-2	-2	-1	0	-2	-1	2	-4	-2	-8
N	0	-4	2	1	-3	0	2	-2	1	-3	-2	2	0	1	0	1	0	-2	-4	-2	-8
P	1	-3	-1	-1	-5	0	0	-2	-1	-3	-2	0	6	0	0	1	0	-1	-6	-5	-8
Q	0	-5	2	2	-5	-1	3	-2	1	-2	-1	1	0	4	1	-1	-1	-2	-5	-4	-8
R	-2	-4	-1	-1	-4	-3	2	-2	3	-3	0	0	0	1	6	0	-1	-2	2	-4	-8
S	1	0	0	0	-3	1	-1	-1	0	-3	-2	1	1	-1	0	2	1	-1	-2	-3	-8
T	1	-2	0	0	-3	0	-1	0	0	-2	-1	0	0	-1	-1	1	3	0	-5	-3	-8
V	0	-2	-2	-2	-1	-1	-2	4	-2	2	2	-2	-1	-2	-2	-1	0	4	-6	-2	-8
W	-6	-8	-7	-7	0	-7	-3	-5	-3	-2	-4	-4	-6	-5	2	-2	-5	-6	17	0	-8
Y	-3	0	-4	-4	7	-5	0	-1	-4	-1	-2	-2	-5	-4	-4	-3	-3	-2	0	10	-8
–	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	

Рис. 7.3. Матрица оценки PAM_{250} для выравнивания белка. Здесь штраф за индели установлен равным 8.

Мутации нуклеотидной последовательности гена часто изменяют аминокислотную последовательность транслированного белка. Некоторые из этих мутаций ухудшают функциональные способности белка, что делает их достаточно редкими событиями в молекулярной эволюции. Такие аминокислоты как *Asn*, *Asp*, *Glu* и *Ser* являются наиболее «изменчивыми», тогда как *Cys* и *Trp* являются наименее изменчивыми. Знание вероятности каждой возможной мутации позволяет биологам создавать матрицы аминокислотных оценок для выравнивания биологически обоснованных последовательностей, в которых разные замены штрафуются по-разному. Элемент матрицы выравнивания аминокислот $score(i, j)$ обычно отражает то, как часто i-я аминокислота заменяет j-ю аминокислоту в выравниваниях связанных белковых последовательностей. В результате оптимальные выравнивания аминокислотных последовательностей могут иметь очень мало совпадений, но все же представляют собой биологически адекватные выравнивания.

Если известен большой набор парных выравниваний связанных последовательностей (например, содержащих по меньшей мере 90% аминокислот), то вычисление *score*(*i, j*) основано на подсчете, сколько раз соответствующие аминокислоты были выровнены. Тем не менее, нужно заранее знать матрицу оценки, чтобы построить этот набор стартовых выравниваний – уловка-22.

Уловка-22 – это ситуация, возникающая в результате логического парадокса между взаимоисключающими правилами и процедурами. В этой ситуации индивид, попадающий под действие таких норм, не может их контролировать, так как попытка нарушить эти установки автоматически подразумевает их соблюдение.

Термин был введен американским писателем Джозефом Хеллером в одноименном романе, опубликованном в 1961 году. В основе сюжета лежит стремление главного героя, пилота бомбардировщика капитана Йоссариана, обойти так называемую «Уловку-22» – абсурдные бюрократические ограничения, которые не дают ему право вернуться домой, отлетав свою норму боевых вылетов в период Итальянской кампании против войск Германии и Италии в 1944 году.

Впервые в произведении этот термин упоминается персонажем романа доктором Дейникой, армейским психиатром:

– Конечно, ловушка, – ответил Дейника. – И называется она «уловка двадцать два». «Уловка двадцать два» гласит: «Всякий, кто пытается уклониться от выполнения боевого долга, не является подлинно сумасшедшим».

Да, это была настоящая ловушка. «Уловка двадцать два» разъясняла, что забота о себе самом перед лицом прямой и непосредственной опасности является проявлением здравого смысла. Орр был сумасшедшим, и его можно было освободить от полетов. Единственное, что он должен был для этого сделать, — попросить. Но как только он попросит, его тут же перестанут считать сумасшедшим и заставят снова летать на задания. Орр сумасшедший, раз он продолжает летать. Он был бы нормальным, если бы захотел перестать летать; но, если он нормален, он обязан летать. Если он летает, значит, он сумасшедший и, следовательно, летать не должен; но, если он не хочет летать, — значит, он здоров и летать обязан.

Схожий принцип был отражен в упоминаемом Бертраном Расселом парадоксе брадобрея:

Пусть в некой деревне живёт брадобрей, который бреет всех жителей деревни, которые не бреются сами, и только их. Бреет ли брадобрей сам себя?

Правильное выравнивание очень похожих последовательностей настолько очевидно, что оно может быть сконструировано даже с помощью примитивной

схемы подсчета, которая не учитывает различные вероятности мутаций (например, +1 для совпадений и -1 для промахов и инделей), тем самым разрешая головоломку. После создания этих очевидных выравниваний можно использовать их для вычисления новой матрицы оценок, которую можно использовать итеративно, чтобы сформировать все менее очевидные выравнивания.

Это упрощенное описание скрывает некоторые детали. Например, вероятность того, что *Ser* мутирует в *Phe* у видов, которые разделились 1 миллион лет назад, меньше вероятности той же мутации у видов, которые разделились 100 миллионов лет назад. Это наблюдение подразумевает, что матрицы оценки для сравнения белков должны зависеть от сходства организмов и скорости эволюции интересующих белков. На практике белки, которые биологи используют для создания первоначального выравнивания, чрезвычайно похожи, причем 99% их аминокислот сохраняются (например, большинство белков, имеющихся у людей и шимпанзе). Считается, что последовательности, которые на 99% похожи, расходятся на 1 единицу PAM («PAM» означает «принятая точка мутации»). Можно думать о единицах PAM как о количестве времени, в течение которого «средний» белок мутирует 1% его аминокислот.

Матрица подсчета PAM_1 определяется из многих парных выравниваний подобных на 99% белков. Для заданного набора парных выравниваний, пусть M(*i, j*) – количество раз, когда *i*-я и *j*-я аминокислоты появляются в одном столбце, деленное на общее количество раз, когда *i*-я аминокислота появляется во всех последовательностях. Пусть *f*(*j*) – частота *j*-й аминокислоты в последовательностях или количество раз, которое она появляется во всех последовательностях, разделенное на объединенные длины двух последовательностей. Запись (*i, j**)-го элемента матрицы PAM_1 определяется как:

$$\log\left[\frac{M(i,j)}{f(j)}\right]$$

Для большего количества *n* единиц PAM матрица PAM_n вычисляется на основе наблюдения, что матрица M^n (результат умножения M саму на себя *n* раз) содержит эмпирические вероятности, что одна аминокислота мутирует в другую за *n* единиц PAM. Таким образом, (*i, j*)-й элемент матрицы PAM_n задается формулой

$$\log\left[\frac{M^n(i,j)}{f(j)}\right]$$

Этот подход предполагает, что частоты аминокислот *f*(*j*) остаются неизменными во времени и что мутационные процессы в интервале 1 единицы PAM действуют последовательно в течение длительного периода времени. При

больших n результирующие матрицы PAM часто позволяют находить связанные белки, даже если выравнивание имеет мало совпадений.

Лекция 8. От глобального к локальному выравниванию.

Теперь можно изменить граф выравнивания, чтобы решить обобщенную форму задачи выравнивания, которая принимает матрицу оценки в качестве входных данных.

Задача. Найти выравнивание с наивысшей оценкой для двух строк, как определено матрицей оценки.

Вход: две строки и матрица выравнивания *Score*.

Выход: выравнивание строк, чья оценка выравнивания (как определено *Score*) максимизируется по всем выравниваниям строк.

Чтобы решить проблему глобального выравнивания, все равно нужно найти самый длинный путь в графе выравнивания после обновления весов ребер, чтобы отразить значения в матрице оценки. Поскольку удаления соответствуют вертикальным ребрам (↓), вставки соответствуют горизонтальным ребрам (→), а совпадения / промахи соответствуют диагональным ребрам (↘ / ↘), можно получить следующую рекурсию для $s_{i,j}$, длины самого длинного пути от $(0, 0)$ до (i, j):

$$s_{i,j} = \max \begin{cases} s_{i-1,j} & + score(v_i, -) \\ s_{i,j-1} & + score(-, w_j) \\ s_{i-1,j-1} & + score(v_i, w_j) \end{cases}$$

Когда награда за совпадение равна +1, штраф за промах равен μ, а штраф за индель равен σ, рекурсия для выравнивания может быть записана следующим образом:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} & - \sigma \\ s_{i,j-1} & - \sigma \\ s_{i-1,j-1} & + 1 \quad \text{if } v_i = w_j \\ s_{i-1,j-1} & - \mu \quad \text{if } v_i \neq w_j \end{cases}$$

Пример графа выравнивания показан на рисунке 8.1.

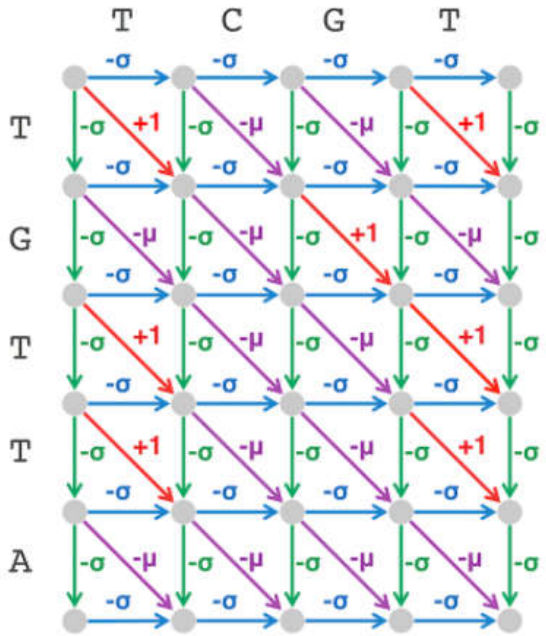


Рис. 8.1. *AlignmentGraph*(TGTTA, TCGT) с каждым ребром, окрашенным в зависимости от того, соответствует ли ребро совпадению, промаху, вставке или удалению.

Задача 8.1. Решить задачу глобального выравнивания.

Вход: две строки белка, записанные в алфавите однобуквенных аминокислот.

Выход: максимальная оценка выравнивания этих строк, выравнивание, достигающее этого максимального балла. Используется матрица оценки $BLOSUM_{62}$ и штраф за индель $\sigma = 5$.

Пример входа:

PLEASANTLY

MEANLY

Пример выхода:

8

PLEASANTLY

-MEA--N-LY

	A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y
A	4	0	-2	-1	-2	0	-2	-1	-1	-1	-1	-2	-1	-1	-1	1	0	0	-3	-2
C	0	9	-3	-4	-2	-3	-3	-1	-3	-1	-1	-3	-3	-3	-3	-1	-1	-1	-2	-2
D	-2	-3	6	2	-3	-1	-1	-3	-1	-4	-3	1	-1	0	-2	0	-1	-3	-4	-3
E	-1	-4	2	5	-3	-2	0	-3	1	-3	-2	0	-1	2	0	0	-1	-2	-3	-2
F	-2	-2	-3	-3	6	-3	-1	0	-3	0	0	-3	-4	-3	-3	-2	-2	-1	1	3
G	0	-3	-1	-2	-3	6	-2	-4	-2	-4	-3	0	-2	-2	-2	0	-2	-3	-2	-3
H	-2	-3	-1	0	-1	-2	8	-3	-1	-3	-2	1	-2	0	0	-1	-2	-3	-2	2
I	-1	-1	-3	-3	0	-4	-3	4	-3	2	1	-3	-3	-3	-3	-2	-1	3	-3	-1
K	-1	-3	-1	1	-3	-2	-1	-3	5	-2	-1	0	-1	1	2	0	-1	-2	-3	-2
L	-1	-1	-4	-3	0	-4	-3	2	-2	4	2	-3	-3	-2	-2	-2	-1	1	-2	-1
M	-1	-1	-3	-2	0	-3	-2	1	-1	2	5	-2	-2	0	-1	-1	-1	1	-1	-1
N	-2	-3	1	0	-3	0	1	-3	0	-3	-2	6	-2	0	0	1	0	-3	-4	-2
P	-1	-3	-1	-1	-4	-2	-2	-3	-1	-3	-2	-2	7	-1	-2	-1	-1	-2	-4	-3
Q	-1	-3	0	2	-3	-2	0	-3	1	-2	0	0	-1	5	1	0	-1	-2	-2	-1
R	-1	-3	-2	0	-3	-2	0	-3	2	-2	-1	0	-2	1	5	-1	-1	-3	-3	-2
S	1	-1	0	0	-2	0	-1	-2	0	-2	-1	1	-1	0	-1	4	1	-2	-3	-2
T	0	-1	-1	-1	-2	-2	-2	-1	-1	-1	-1	0	-1	-1	-1	1	5	0	-2	-2
V	0	-1	-3	-2	-1	-3	-3	3	-2	1	1	-3	-2	-2	-3	-2	0	4	-3	-1
W	-3	-2	-4	-3	1	-2	-2	-3	-3	-2	-1	-4	-4	-2	-3	-3	-2	-3	11	2
Y	-2	-2	-3	-2	3	-3	2	-1	-2	-1	-1	-2	-3	-1	-2	-2	-2	-1	2	7

Рис. 8.2. Матрица оценки $BLOSUM_{62}$ для выравнивания белка.

Анализ гомеобоксных генов является примером задачи, для которой глобальное выравнивание может не выявить биологически значимых сходств. Эти гены регулируют эмбриональное развитие и присутствуют в большом разнообразии видов, от мух до людей. Гомеобоксные гены длинны, они сильно различаются между видами, но существуют последовательности из примерно 60 аминокислот в каждом гене, называемые гомеодоменом, которые очень консервативны. Рассмотрим мышиный и человеческий гомеодомены:

Mouse

...ARRSRTHFTKFQTDILIEAFEKNRFPGIVTREKLAQQTGIPESRIHIWFQNRRARHPDGP...

...ARQKQTFITWTQKNRLVQAFERNPFPDTATRKKLAEQTGLQESRIQMWFQKQRSLYLKKS...

Human

Непосредственный вопрос заключается в том, как найти этот консервативный сегмент в гораздо более длинных генах и игнорировать остальные области, которые мало похожи. Глобальное выравнивание ищет сходство между двумя строками по всей их длине; однако при поиске гомеодоменов находятся более мелкие локальные области сходства и не выравнивают целые строки. Например, глобальное выравнивание между двумя последовательностями ниже имеет 22 совпадения, 18 индексов и 2 промаха, что приводит к результату 22 - 18 - 2 = 2 (если $\sigma = \mu = 1$).

GCC-C-AGTC-TATGT-CAGGGGGCACG--A-GCATGCACA-

GCCGCC-GTCGT-T-TTCAG----CA-GTTATGT-T-CAGAT

Однако эти последовательности могут быть выровнены по-разному (с 17 совпадениями и 32 инделями) на основе высококонсервативного интервала, представленного подстроками CAGTCTATGTCAG и CAGTTATGTTCAG:

```
---G----C-----C--CAGTCTATG-TCAGGGGGCACGAGCATGCACA
GCCGCCGTCGTTTTCAGCAGT-TATGTTCAG-----A------T-----
```

У этого выравнивания меньше совпадений и более низкая оценка 17 - 32 = -15, хотя консервативная область выравнивания дает оценку 12 - 2 = 10.

На рисунке 8.3 показаны два пути, соответствующие этим двум различным выравниваниям. Верхний путь, соответствующий второму выравниванию, проигрывает, потому что он содержит много сильно оштрафованных инделей по обе стороны от диагонали, соответствующих консервативному интервалу. В результате глобальное выравнивание выводит биологически нерелевантный нижний путь. Вопрос состоит в том, как можно исключить биологически несоответствующие выравнивания, когда присутствуют местные сходства.

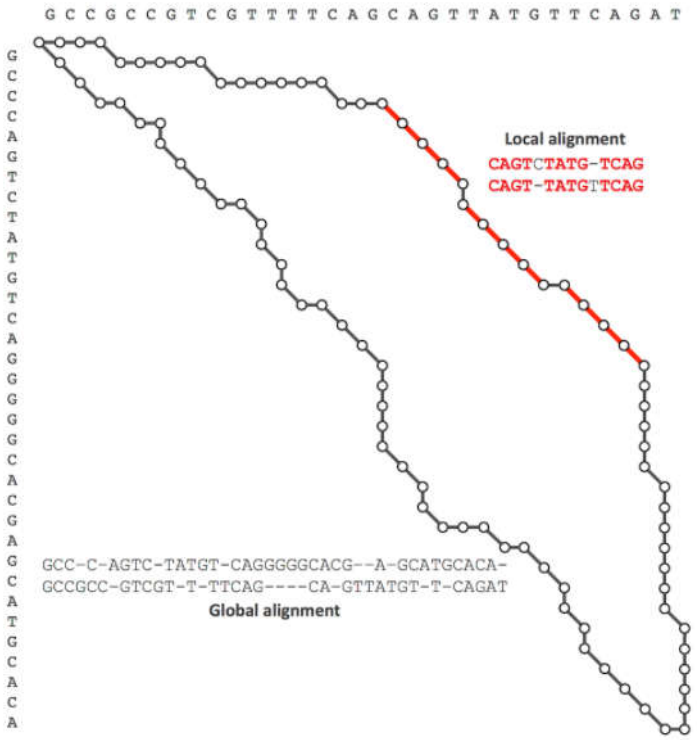


Рис. 8.3. Глобальное и локальное выравнивания двух цепочек ДНК, которые имеют высококонсервативный интервал. Соответствующее выравнивание, которое захватывает этот интервал (верхний путь), проигрывает некорректному выравниванию (нижний путь), поскольку на первый накладываются тяжелые штрафные санкции.

Когда биологически значимые сходства присутствуют в некоторых частях последовательностей v и w и отсутствуют у других, биологи пытаются игнорировать глобальное выравнивание и вместо этого выравнивают подстроки v и w, что дает локальное выравнивание двух строк. Проблема

нахождения подстрок, которые максимизируют оценку глобального выравнивания по всем подстрокам *v* и *w*, называется проблемой локального выравнивания.

Задача. Найти локальное выравнивание с максимальной оценкой для двух строк.

Вход: строки *v* и *w*, матрица выравнивания *Score*.

Выход: подстроки *v* и *w*, чья оценка глобального выравнивания (согласно *Score*) максимальна среди всех подстрок *v* и *w*.

Прямой способ решения проблемы локального выравнивания – найти самый длинный путь, соединяющий каждую пару узлов в графе выравнивания (а не только те, которые соединяют источник и сток, как в задаче поиска глобального выравнивания), а затем выбрать путь с максимальным весом по всем этим самым длинным путям.

Для более быстрого решения можно представить себе «бесплатную поездку на такси» от источника (*0,0*) до узла, представляющего начальный узел консервативного (красного) интервала на рисунке 8.4. Можно также представить себе «бесплатную поездку на такси» от конечного узла консервативного интервала до стока. Если бы такие поездки были доступны, можно было бы бесплатно добраться до стартового узла консервативного интервала, вместо того, чтобы вносить большие штрафы, как при глобальном выравнивании. Затем можно проходить по консервативному интервалу до его конечного узла, накапливая положительные оценки совпадений. Наконец, можно взять еще одну бесплатную поездку от конечного узла консервативного интервала к стоку. Полученная оценки этой поездки равна оценки выравнивания только консервативных интервалов.

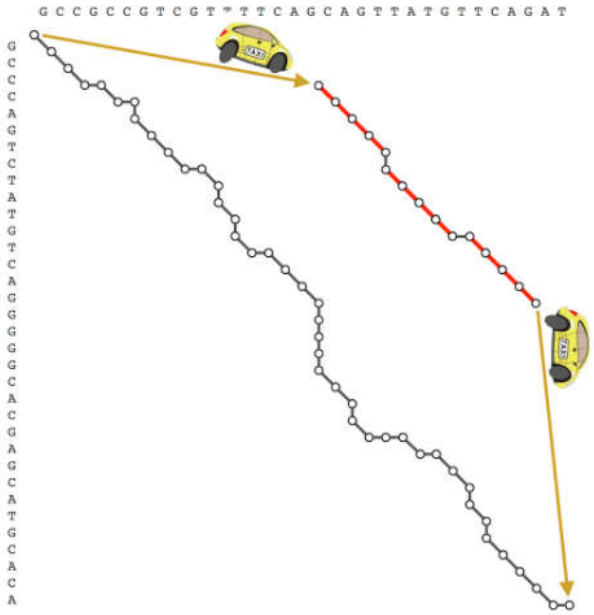


Рис. 8.4. Изменение рисунка 115 путем добавления ребер «бесплатных поездок на такси» (с весом 0), соединяющих источник с начальным узлом консервативного интервала и соединяющих конечный узел консервативного интервала со стоком. Эти новые ребра позволяют оценивать только локальное выравнивание, содержащее консервативный интервал.

Соединение источника (*0, 0*) с каждым другим узлом путем добавления ребра нулевого веса и соединения каждого узла со стоком (*n, m*) ребром нулевого веса приведет к тому, что DAG идеально подходит для решения проблемы локального выравнивания, как показано на рисунке 8.5. Из-за бесплатных поездок больше не нужно строить длинный путь между каждой парой узлов в графе – самый длинный путь от источника до стока дает оптимальное локальное выравнивание.

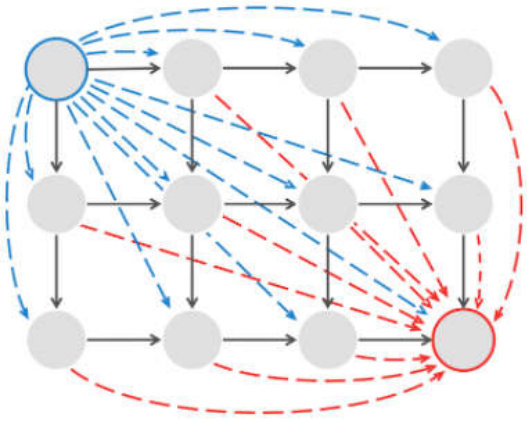


Рис. 8.5. Алгоритм локального выравнивания вводит ребра с нулевым весом (показаны синими пунктирными линиями), соединяющие источник (*0, 0*) с каждым другим узлом графа выравнивания, а также ребра с нулевым весом (показаны красными пунктирными линиями), соединяющими каждый узел с узлом-стоком.

Общее число ребер в приведенном на рисунке 8.5 графе равно $O(|v|\cdot|w|)$, что все еще мало. Поскольку время нахождения самого длинного пути в DAG определяется количеством ребер в графе, результирующий алгоритм локального выравнивания будет быстрым. Что касается вычисления значений $s_{i,j}$, добавление ребер нулевого веса из $(0, 0)$ в каждый узел сделало узел-источник $(0, 0)$ предшественником каждого узла (i, j). Следовательно, теперь есть четыре ребра, входящие в (i, j), что добавляет только одно новое слагаемое к рекуррентному соотношению:

$$s_{i,j} = \max \begin{cases} 0 \\ s_{i-1,j} + score(v_i, -) \\ s_{i,j-1} + score(-, w_j) \\ s_{i-1,j-1} + score(v_i, w_j) \end{cases}$$

Кроме того, поскольку узел (n, m) теперь имеет любой другой узел в качестве предшественника, $s_{n,m}$ будет равен наибольшему значению $s_{i,j}$ по всему графу выравнивания.

Задача 8.2. Решить задачу локального выравнивания.

Вход: две строки белка, записанные в алфавите однобуквенных аминокислот.

Выход: максимальная оценка выравнивания этих строк, выравнивание, достигающее этого максимального балла. Используется матрица оценки PAM_{250} и штраф за индель $\sigma = 5$.

Пример входа:

MEANLY

PENALTY

Пример выхода:

15

EANL-Y

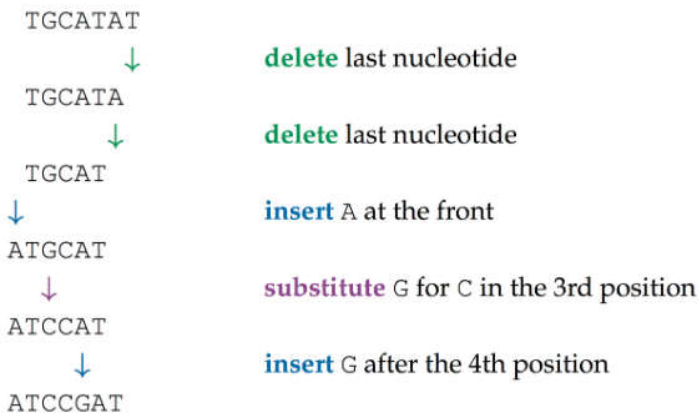
ENALTY

Рассмотрим ещё несколько задач, связанных с выравниванием последовательностей.

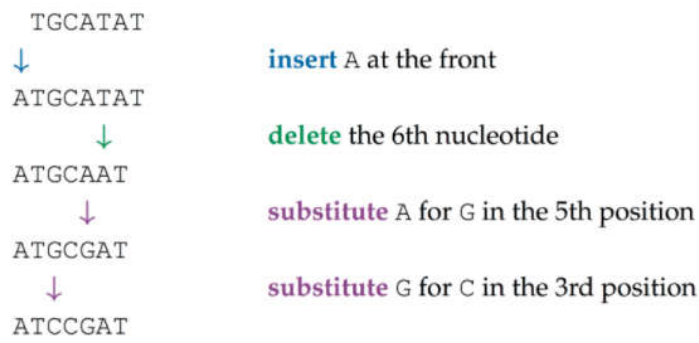
Задача 1: Расстояние редактирования.

В 1966 году Владимир Левенштейн ввел понятие расстояния редактирования между двумя строками как минимальное количество операций редактирования, необходимых для преобразования одной строки в другую. Здесь операции редактирования – это вставка, удаление или замена одного

символа. Например, TGCATAT можно преобразовать в ATCCGAT с помощью пяти операций редактирования, подразумевая, что расстояние редактирования между этими строками не более 5.



Фактически, расстояние редактирования между TGCATAT и ATCCGAT равно 4:



Левенштейн представил расстояние редактирования, но не описал алгоритм его вычисления.

Задача 8.3. Найти расстояние редактирования между двумя строками.

Вход: две строки.

Выход: расстояние редактирования между этими строками.

Пример входа:

PLEASANTLY

MEANLY

Пример выхода:

5

Задача 2: Фитирующее выравнивание.

Предположим, что нужно сравнить приблизительно 20 000 аминокислотных NRP-синтетаз *Bacillus brevis* с длиной приблизительно в 600 аминокислот с

Streptomyces roseosporus, бактерией, которая продуцирует антибиотик *Daptomycin*. Есть надежда найти область в более длинной последовательности белка *v*, которая имеет большое сходство со всей более короткой последовательностью *w*. Глобальное выравнивание не будет работать, поскольку оно пытается выровнять всю *v* со всей *w*; локальное выравнивание не будет работать, поскольку оно пытается выровнять подстроки *v* и *w*. Таким образом, у есть четкое приложение для выравнивания, называемое проблемой фитирующего выравнивания.

«Фитирование» *w* к *v* требует поиска подстроки *v'* в *v*, которая максимизирует глобальную оценку выравнивания между *v* и *w* среди всех подстрок v. К примеру, лучшие глобальное, локальное и фитирующее выравнивания *v*=GTAGGCTTAAGGTTA и *w*=TAGATA показаны ниже (с штрафами за промахи и индели, равными 1).

Global	Local	Fitting
GTAGGCTTAAGGTTA -TAG----A---T-A	GTAGGCTTAAGGTTA TAGATA	GTAGGCTTAAGGTTA TAGA--TA

Стоит обратить внимание, что оптимальное локальное выравнивание (с оценкой 3) не является допустимым фитирующим выравниванием. С другой стороны, оценка оптимального глобального выравнивания (6 - 9 = -3) меньше, чем оценка лучшего фитирующего выравнивания (5 - 1 - 2 = +2).

Задача 8.4. Построить фитирующее выравнивание двух строк с лучшей оценкой.

Вход: две нуклеотидные строки *v* и *w*, где *v* имеет длину не более 1000 и *w* имеет длину не более 100.

Выход: фитирующее выравнивание с лучшей оценкой для *v* и *w*. Совпадение дает +1, штрафы за промахи и индели равны 1.

Пример входа:

GTAGGCTTAAGGTTA

TAGATA

Пример выхода:

2

TAGGCTTA

TAGA--TA

Задача 3: Перекрывающееся выравнивание.

Перекрывающиеся прочтения затрудняли сборку генома ошибками в прочтениях. Выравнивание концов гипотетических прочтений, показанных ниже, дает возможность найти перекрытия между прочтениями, подверженным ошибкам.

ATGCATGCCGG

TGCCGGAAAC

Перекрывающееся выравнивание строк $v=v_1...v_n$ и $w=w_1...w_m$ является глобальным выравниванием суффикса v с префиксом w. Оптимальное перекрывающееся выравнивание строк v и w максимизирует оценку глобального выравнивания между i-суффиксом v и j-префиксом w (то есть между $v_i...v_n$ и $w_1...w_j$) для всех i и j.

Задача 8.5. Построить перекрывающееся выравнивание двух строк с лучшей оценкой.

Вход: две строки v и w, где каждая имеет длину не более 1000.

Выход: оценка оптимального перекрывающегося выравнивания v и w, выравнивания суффикса v' строки v и префикса w' строки w для достижения этой максимальной оценки. Совпадение дает +1, штрафы промаха и инделя равны 2.

Пример входа:

PAWHEAE

HEAGAWGHEE

Пример выхода:

1

HEAE

HEAG

Введение штрафов за промахи и индели может привести к более биологически адекватным глобальным выравниваниям. Однако, даже с этой более надежной моделью выравнивание А-доменов, которое было построено ранее (с штрафом за индели $\sigma = 4$), все еще обнаруживает только 6 из 8 консервативных фиолетовых столбцов, соответствующих нерибосомальным сигнатурам:

YAFDLGYTCMFP-VLL-GGGELHIV-QKETYTAPDEI-AHYIKEHGITYI-KLTPSLFHTIVNTASFAFDANFE
-AFDVS-AGDFARALLTGG-QL-IVCPNEVKMDPASLYA-IIKKYDIT-IFEATPAL--VIPLME-YIYEQKLD

-S-LR-LIVLGGEKIIPIDVIAFRKM---YGHTE-FINHYGPTEATIGA
ISQLQILIV-GSDSC-SME--DFKTLVSRFGSTIRIVNSYGVTEACIDS

В ранее определенной линейной модели оценивания, если σ является штрафом за вставку или удаление одного символа, то $\sigma \cdot k$ является штрафом за вставку

или удаление интервала из k символов. Эта модель приводит к неадекватной оценке биологических последовательностей. Мутации часто вызывают ошибки в репликации ДНК, которые вставляют или удаляют весь интервал из k нуклеотидов как одно событие, а не как k независимых вставок или удалений. Таким образом, наложение штрафа на такой индель размером $\sigma \cdot k$ представляет собой чрезмерное наказание. Например, выравнивание справа более адекватно, чем выравнивание слева, но в настоящее время они получают одинаковую оценку.

```
GATCCAG GATCCAG
GA-C-AG GA--CAG
```

Разрыв представляет собой непрерывную последовательность пробелов в строке выравнивания. Одним из способов более точного оценивания разрывов является определение аффинного штрафа за промежуток длины k следующего вида: $\sigma + \varepsilon \cdot (k - 1)$, где σ – штраф за начало разрыва, накладываемый на первый символ в разрыве, а ε – штраф за продолжение разрыва, накладываемый на каждый дополнительный символ в промежутке. Обычно выбирают ε меньше, чем σ, чтобы аффинный штраф за разрыв длины k был меньше штрафа за k независимых однонуклеотидных инделей ($\sigma \cdot k$). Например, если $\sigma = 5$ и $\varepsilon = 1$, то выравнивание слева наказывается как $2\sigma = 10$, тогда как тогда как выравнивание справа наказывается только $\sigma + \varepsilon = 6$.

Задача. Построить глобальное выравнивание двух строк с лучшей оценкой (с аффинными штрафами).

Вход: две строки, матрица оценок *score*, числа σ и ε.

Выход: глобальное выравнивание этих строк с наилучшей оценкой, определяемое матрицей оценок *score*, а также штрафами на начало и продолжение разрыва σ и ε.

На рисунке 8.6 показано, как аффинные штрафы за разрыв могут быть смоделированы в графе выравнивания введением нового «длинного» ребра для каждого разрыва.

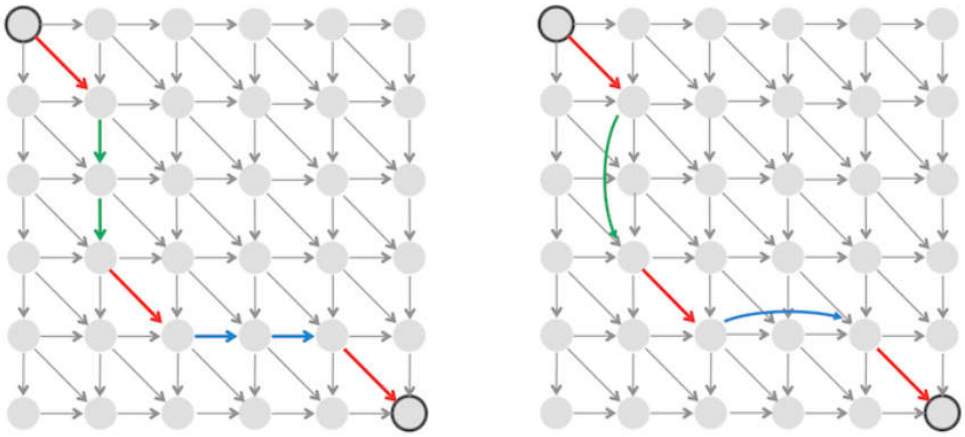


Рис. 8.6. Представление разрывов в графе выравнивания слева как «длинных» ребер вставок и удаления в графе выравнивания справа. Для разрыва длины k вес соответствующего длинного ребра равен $\sigma + \varepsilon \cdot (k - 1)$.

Поскольку заранее неизвестно, где должны располагаться разрывы, нужно добавить ребра, учитывающие все возможные разрывы. Таким образом, аффинные штрафы могут быть распределены путем добавления всех возможных вертикальных и горизонтальных ребер в графе выравнивания для представления всех возможных разрывов. В частности, добавляем ребра, соединяющие (i, j) с обоими узлами $(i + k, j)$ и $(i, j + k)$ с весами $\sigma + \varepsilon \cdot (k - 1)$ для всех возможных размеров разрыва k, как показано на рисунке 8.7. Для двух последовательностей длины n число ребер в результирующем графе выравнивания, моделирующих аффинные разрывы, увеличивается от $O(n^2)$ до $O(n^3)$.

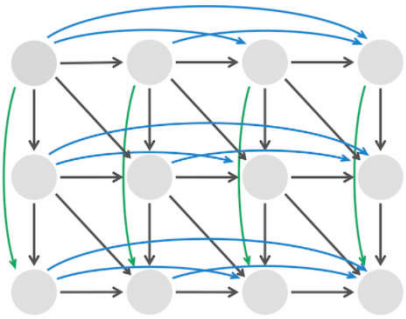


Рис. 8.7. Добавление ребер, соответствующих инделям для всех возможных размеров разрыва, добавляет большое количество ребер к графу выравнивания.

Прием уменьшения количества ребер в DAG для проблемы выравнивания с помощью аффинных штрафов заключается в увеличении числа узлов. С этой целью построим граф выравнивания на трех уровнях; для каждого узла (i, j) построим три разных узла: $(i, j)_{\text{lower}}$, $(i, j)_{\text{middle}}$ и $(i, j)_{\text{upper}}$. Средний уровень будет содержать диагональные ребра веса $score(v_i, w_j)$, представляющие совпадения и промахи. Нижний уровень будет содержать только вертикальные ребра с весом $-\varepsilon$, чтобы представлять продолжения разрыва в v, а верхний уровень

будет иметь только горизонтальные ребра с весами -ε, чтобы представлять продолжения разрыва в w (см. Рисунок 8.8).

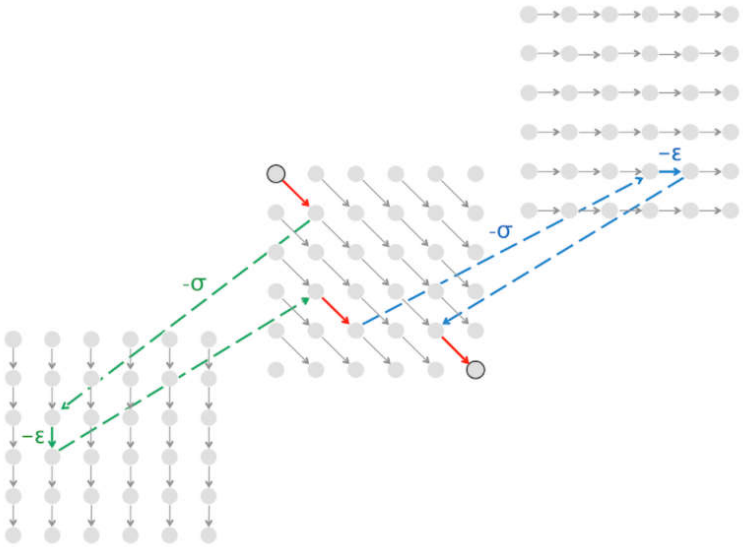


Рис. 8.8. Создание трехуровневого графа выравнивания с аффинными штрафами. Нижний уровень соответствует продолжениям разрыва в v, средний уровень соответствует совпадениям и промахам, а верхний уровень соответствует продолжениями разрыва в w.

В настоящее время нет способа передвигаться между разными уровнями. Чтобы решить эту проблему, нужно добавить ребра, ответственные за открытие и закрытие разрывов. Чтобы моделировать открытие разрыва, соединяем каждый узел $(i, j)_{\text{middle}}$ с обоими $(i + 1, j)_{\text{lower}}$ и $(i, j + 1)_{\text{upper}}$; назначаем вес этим ребрам, равный -σ. Закрытие разрыва не несет штрафа, поэтому вводим нулевые веса, соединяющие узлы $(i, j)_{\text{lower}}$ и $(i, j)_{\text{upper}}$ с соответствующим узлом $(i, j)_{\text{middle}}$ на среднем уровне. В результате разрыв длины k начинается и заканчивается на среднем уровне и штрафуется на -σ для первого символа, -ε для каждого последующего символа и на 0, чтобы закрыть разрыв, создавая общей штраф $\sigma + \varepsilon \cdot (k - 1)$.

Только что построенный DAG может быть сложным, но он использует только $O(n \cdot m)$ ребер для последовательностей длины n и m, а самый длинный путь на этом графе по-прежнему создает оптимальное выравнивание с аффинными штрафами. Трехуровневый граф выравнивания преобразуется в систему трех рекуррентных соотношений, показанных ниже. Здесь $lower_{i,j}$, $middle_{i,j}$ и $upper_{i,j}$ являются длинами самых длинных путей от исходного узла до $(i, j)_{\text{lower}}$, $(i, j)_{\text{middle}}$ и $(i, j)_{\text{upper}}$ соответственно:

$$lower_{i,j} = \max \begin{cases} lower_{i-1,j} - \epsilon \\ middle_{i-1,j} - \sigma \end{cases}$$

$$upper_{i,j} = \max \begin{cases} upper_{i,j-1} - \epsilon \\ middle_{i,j-1} - \sigma \end{cases}$$

$$middle_{i,j} = \max \begin{cases} lower_{i,j} \\ middle_{i-1,j-1} + score(v_i, w_j) \\ upper_{i,j} \end{cases}$$

Задача 8.6. Решить задачу выравнивания с аффинными штрафами.

Вход: две аминокислотные строки *v* и *w*, где каждая имеет длину не более 100.

Выход: оценка оптимального выравнивания *v* и *w*, выравнивание *v* и *w*, достигающее этой максимальной оценки. Используется матрица оценок $BLOSUM_{62}$, штраф за начало разрыва равен 11, штраф за продолжение разрыва равен 1.

Пример входа:

PRTEINS

PRTWPSEIN

Пример выхода:

8

PRT---EINS

PRTWPSEIN-

Лекция 9. Эффективное по используемой памяти выравнивание последовательностей. Множественное выравнивание.

Чтобы ввести фитирующее выравнивание, был использован пример выравнивания NRP-синтетазы размером 20000 аминокислот *Bacillus brevis* с длинным А-доменом длиной в 600 аминокислот *Streptomyces roseosporus*. Однако память, необходимая для хранения матрицы динамического программирования, существенна.

Время выполнения алгоритма динамического программирования для выравнивания двух последовательностей длин *n* и *m* пропорционально количеству ребер в их графе выравнивания, которое равно $O(n \cdot m)$. Память,

требуемая этим алгоритмом, также $O(n \cdot m)$, так как нужно сохранить указатели обратного отслеживания. Можно построить выравнивание в $O(n)$ пространстве за счет удвоения времени выполнения (что означает, что время выполнения все еще $O(n \cdot m)$).

Алгоритм «разделяй и властвуй» часто работает, когда решение большой проблемы может быть построено из решений меньших экземпляров задачи. Такая стратегия применяется в два этапа. Фаза разделения разделяет экземпляр задачи на более мелкие экземпляры и решает их; фаза властвования сводит малые решения к решению исходной задачи.

В качестве примера алгоритма «разделяй и властвуй» рассмотрим задачу сортировки списка целых чисел. Начнем с проблемы слияния, в которой требуется объединить два отсортированных списка $List_1$ и $List_2$ в один отсортированный список. Приведенный ниже алгоритм *Merge* объединяет два отсортированных списка в один отсортированный список за $O(|List_1|+|List_2|)$, итерационно выбирая наименьший оставшийся элемент в $List_1$ и $List_2$ и перемещая его в растущий отсортированный список.

```
MERGE(List1, List2)
  SortedList ← empty list
  while both List1 and List2 are non-empty
    if the smallest element in List1 is smaller than the smallest element in List2
      move the smallest element from List1 to the end of SortedList
    else
      move the smallest element from List2 to the end of SortedList
  move any remaining elements from either List1 or List2 to the end of SortedList
  return SortedList
```

На рисунке 9.1 проиллюстрирован процесс объединения отсортированных списков (2, 5, 7, 8) и (3, 4, 6) в отсортированный список (2, 3, 4, 5, 6, 7, 8).

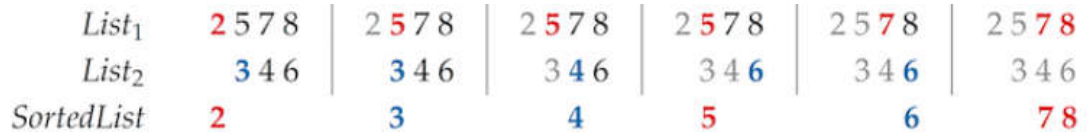


Рис. 9.1. Процесс объединения отсортированных списков.

Merge был бы полезен для сортировки произвольного списка, если бы было известно, как разделить произвольный (несортированный) список на два уже отсортированных полуразмерных списка. Может показаться, что это вернуло нас к началу, за исключением того, что теперь нужно сортировать два небольших списка вместо одного большого. Однако, сортировка двух меньших списков является предпочтительной вычислительной проблемой. Рассмотрим алгоритм *MergeSort*, который делит несортированный список на две части, а

затем рекурсивно вызывает каждую меньшую задачу сортировки перед объединением отсортированных списков.

```
MERGESORT(List)
  if List consists of a single element
    return List
  FirstHalf ← first half of List
  SecondHalf ← second half of List
  SortedFirstHalf ← MERGESORT(FirstHalf)
  SortedSecondHalf ← MERGESORT(SecondHalf)
  SortedList ← MERGE(SortedFirstHalf, SortedSecondHalf)
  return SortedList
```

На рисунке 9.2 показано дерево рекурсии *MergeSort*, состоящее из $\log_2(n)$ уровней, где n – размер исходного несортированного списка. На нижнем уровне нужно объединить два отсортированных списка примерно из $n/2$ элементов каждый, для этого требуется время $O(n/2+n/2)=O(n)$. На следующем более высоком уровне нужно объединить четыре списка из $n/4$ элементов, требуя времени $O(n/4+n/4+n/4+n/4)=O(n)$. Этот шаблон можно обобщить: i-й уровень содержит $2i$ списков, каждый из которых имеет примерно $n/2i$ элементов и требует времени $O(n)$ для слияния. Поскольку в дереве рекурсии всего $\log_2(n)$ уровней, *MergeSort* требует $O(n\cdot\log_2(n))$ времени выполнения, что обеспечивает ускорение по сравнению с наивными алгоритмам сортировки, работающими за $O(n^2)$.

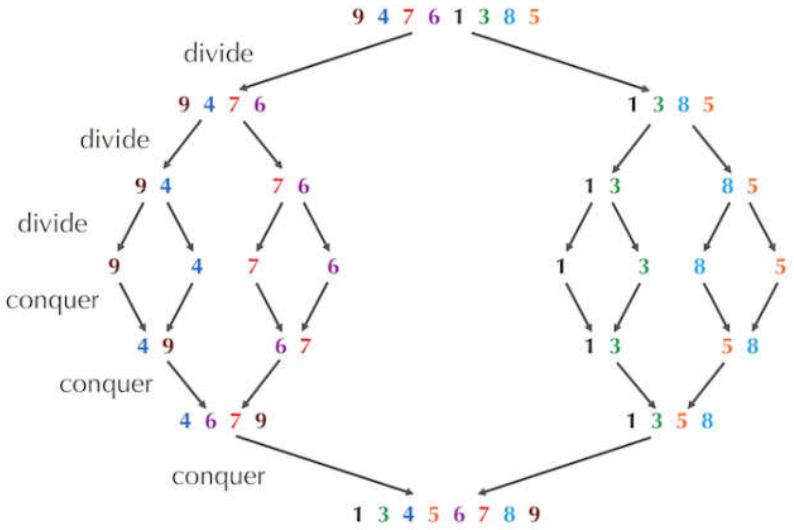


Рис. 9.2. Дерево рекурсии для сортировки списка из 8 элементов с помощью *MergeSort*. Шаги «разделения» (верхние) состоят из $\log_2(8) = 3$ уровней, где входной список разбивается на меньшие и меньшие подсписки. Шаги «властвования» (нижние) состоят из того же количества уровней, где отсортированные подсписки объединяются вместе.

Прежде чем приступить к алгоритму «разделяй и властвуй» для линейного по памяти выравнивания, нужно обратить внимание, что если нужно только вычислить оценку выравнивания, а не получить само выравнивание, то

требуемое пространство можно легко уменьшить удвоением количества узлов в одном столбце графа выравнивания, что есть O(n). Это сокращение получается из наблюдения, что единственными значениями, необходимыми для вычисления оценок выравнивания в столбце j, являются оценки выравнивания в столбце j-1. Поэтому оценки выравнивания в столбцах перед столбцом j-1 могут быть отброшены при вычислении оценок выравнивания для столбца j, как показано на рисунке 9.3. К сожалению, поиск самого длинного пути требует хранения всей матрицы указателей обратного отслеживания, что и требует квадратичного пространства. Идея техники сокращения пространства заключается в том, что не нужно хранить указатели обратного отслеживания, если можно потратить больше времени.

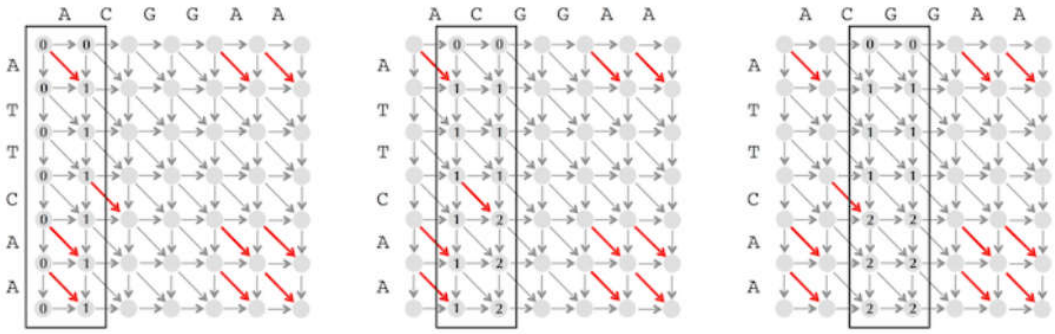


Рис. 9.3. Вычисление оценки выравнивания LCS путем хранения оценок всего за два столбца графа выравнивания.

Для заданных строк $v=v_1...v_n$ и $w=w_1...w_m$, определим $middle=\lfloor m / 2 \rfloor$. Средний столбец *AlignmentGraph*(v, w) – это столбец, содержащий все узлы (i, $middle$) для $0 \le i \le n$. Самый длинный путь от источника до стока в графе выравнивания должен где-то пересекать средний столбец, и первая задача – выяснить, где именно, используя только O(n) память. Обращаемся к узлу, где самый длинный путь пересекает средний столбец как средний узел (разные самые длинные пути могут иметь разные средние узлы, а заданный самый длинный путь может иметь более одного среднего узла.). На рисунке 9.4 $middle$ = 3, а оптимальный путь выравнивания пересекает средний столбец в (единственном) среднем узле (4, 3).

Главное наблюдение заключается в том, что можно найти средний узел самого длинного пути, не создавая этот путь в графе выравнивания. Путь от источника к стоку классифицируется как i-путь, если он пройдет через средний столбец в строке i. К примеру, путь, выделенный на рисунке 9.4, представляет собой 4-путь. Для каждого i между 0 и n требуется найти длину самого длинного i-пути (обозначенного *Length*(i)), потому что наибольшее значение *Length*(i) по всем i покажет средний узел.

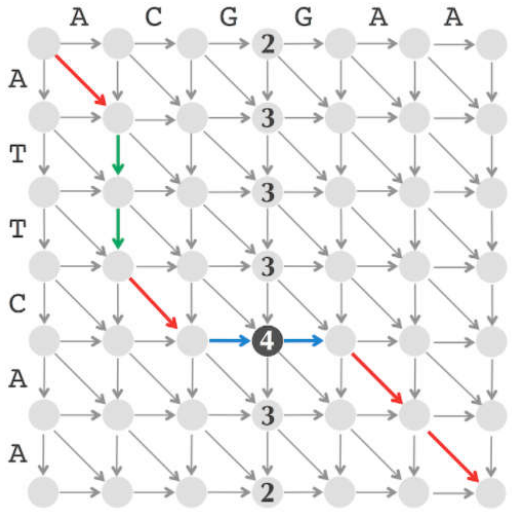


Рис. 9.4. Граф выравнивания ATTCAA и ACGGAA с выделенным путем LCS. Число внутри каждого узла (i, *middle*) в среднем столбце равно *Length*(*i*). Узел, максимизирующий *Length*(*i*) является средним узлом (могут существовать многочисленные средние узлы); средний узел в этом графе окрашен в черный цвет.

Пусть *FromSource*(*i*) обозначает длину самого длинного пути от источника, заканчивающегося в (*i, middle*) и *ToSink*(*i*) обозначает длину самого длинного пути от (*i, middle*) до стока. Тогда *Length*(*i*)=*FromSource*(*i*)+*ToSink*(*i*) и поэтому нужно вычислить *FromSource*(*i*) и *ToSink*(*i*) для каждого *i*.

Значение *FromSource*(*i*) равно $s_{i,middle}$, которое может быть вычислено в линейной памяти. Таким образом, значения *FromSource*(*i*) для всех *i* хранятся как $s_{i,middle}$ в среднем столбце графа выравнивания, а их вычисление требует рассмотрения половины графа выравнивания от столбца 0 до среднего столбца. Поскольку нужно исследовать примерно половину ребер графа выравнивания для вычисления *FromSource*(*i*), говорят, что время выполнения, необходимое для вычисления всех значений *FromSource*(*i*), пропорционально половине «области» графа выравнивания или $n \cdot m / 2$ (см. Рисунок 9.5).

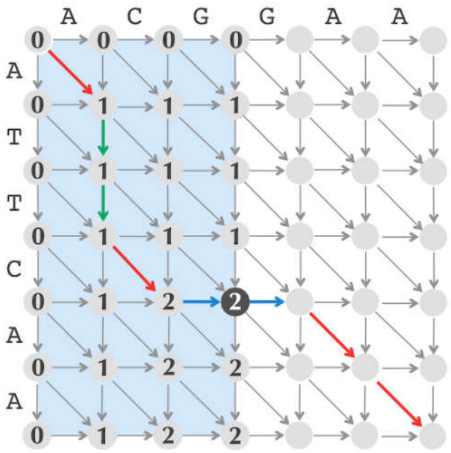


Рис. 9.5. Вычисление *FromSource*(*i*) для всех *i* может быть выполнено за O(*n*) по памяти и за время, пропорциональное $n \cdot m / 2$.

Вычисление *ToSink*(*i*) эквивалентно поиску самого длинного пути от стока до (*i*, *middle*), если все направления ребер изменить на противоположные. Вместо того, чтобы разворачивать ребра, можно перевернуть строки $v=v_1...v_n$ и $w=w_1...w_m$ и найти $s_{n\text{-}i,m\text{-}middle}$ в графе выравнивания для $v_n...v_1$ и $w_m...w_1$. Таким образом, вычисление *ToSink*(*i*) аналогично вычислению *FromSource*(*i*) и может также выполняться за O(*n*) по памяти за время, пропорциональное $n \cdot m / 2$, или половине области графика выравнивания (см. Рисунок 9.6). В общем случае можно вычислить все значения *Length*(*i*)=*FromSource*(*i*)+*ToSink*(*i*) в линейной памяти за время, пропорциональное $n \cdot m / 2 + n \cdot m / 2 = n \cdot m$, что является общей площадью графа выравнивания.

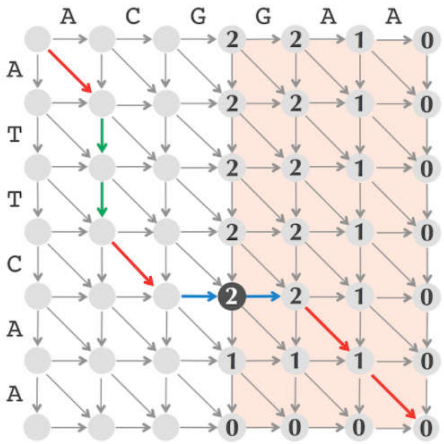


Рис. 9.6. Вычисление *ToSink*(*i*) для всех *i* может быть выполнено за O(*n*) по памяти и за время, пропорциональное $n \cdot m / 2$; это требует изменения направления всех ребер и использования стока в качестве источника.

На данный момент уже потрачено время O($n \cdot m$) (вся область графа выравнивания) для получения очень малого количества информации.

Как только будет найдет средний узел, автоматически можно узнать два прямоугольника, через которые должен пройти самый длинный путь по обе стороны от среднего узла. Как показано на рисунке 9.7, один из этих прямоугольников состоит из всех узлов выше и слева от среднего узла, тогда как другой прямоугольник состоит из всех узлов ниже и справа от среднего узла. Таким образом, площадь двух выделенных прямоугольников составляет половину общей площади графа выравнивания.

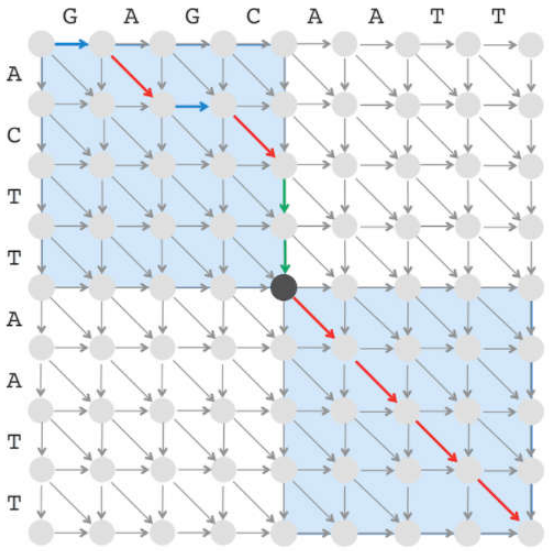


Рис. 9.7. Средний узел (показан черным) определяет два выделенных прямоугольника и иллюстрирует, что оптимальный путь должен проходить внутри этих прямоугольников. Поэтому можно исключить оставшиеся части графа выравнивания из соображений оптимального пути выравнивания.

Теперь можно разделить проблему нахождения самого длинного пути от (*0, 0*) до (*n, m*) на две подзадачи: поиск самого длинного пути от (*0,0*) до среднего узла; и поиск самого длинного пути от среднего узла до (*n, m*). Шаг «властвования» находит два средних узла в меньших прямоугольниках, который может быть выполнен за время, пропорциональное сумме площадей этих прямоугольников или $n \cdot m / 2$ (рисунок 9.8). Заметим, что теперь восстановлены три узла оптимального пути.

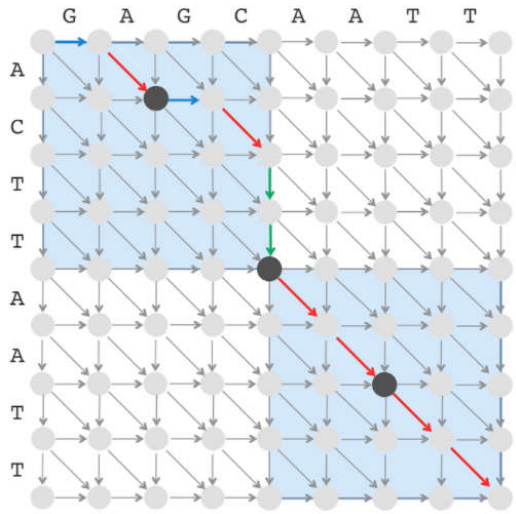


Рис. 9.8. Поиск средних узлов (показаны как еще два черных круга) в ранее идентифицированных прямоугольниках.

На следующей итерации произойдет «разделение и властвование», чтобы найти четыре средних узла за время, равное сумме площадей еще меньших

синих прямоугольников, которые имеют общую площадь $n \cdot m / 4$ (см. Рисунок 9.9). Теперь восстановлены почти все узлы оптимального пути выравнивания.

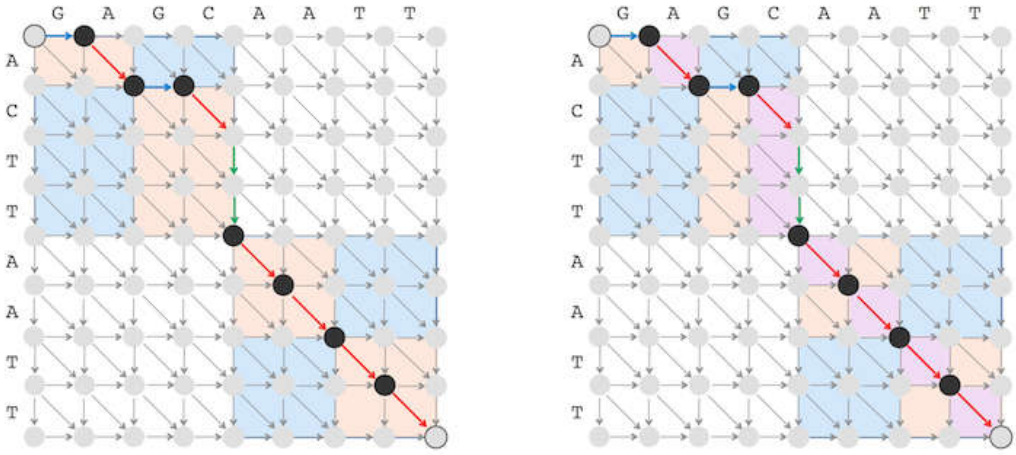


Рис. 9.9. Поиск средних узлов (выделенных как черные круги) в ранее обозначенных синих прямоугольниках.

В общем, на каждом новом шаге перед заключительным, количество найденных средних узлов удваивается, сокращая вдвое время выполнения, необходимое для поиска средних узлов. Исходя из этого, найдем средние узлы всех прямоугольников (и, следовательно, построим все выравнивание) за время, равное

$$n \cdot m + n \cdot \frac{m}{2} + n \cdot \frac{m}{4} + \cdots < 2 \cdot n \cdot m = O(n \cdot m).$$

Таким образом, пришли к линейному алгоритму, который требует только линейной памяти.

Можно использовать более элегантный подход, основанный на нахождении среднего ребра или ребра в оптимальном пути выравнивания, начинающегося со среднего узла (для данного среднего узла может существовать более одного среднего ребра). Как только среднее ребро найдено, снова известны два прямоугольника, через которые самый длинный путь должен проходить по обе стороны от среднего ребра. Но теперь эти два прямоугольника занимают менее половины площади графика выравнивания (см. Рисунок 9.10), что является преимуществом выбора среднего ребра вместо среднего узла.

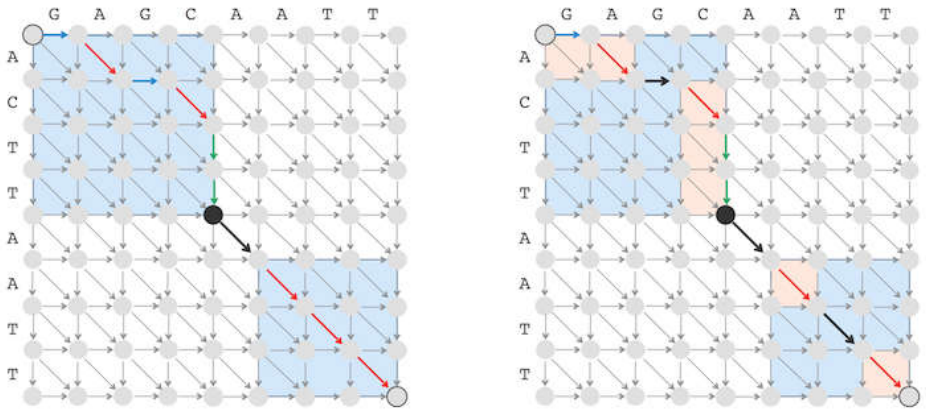


Рис. 9.10. (слева) Среднее ребро (выделено жирным) начинается со среднего узла (черный круг). Оптимальный путь проходит внутри первого выделенного прямоугольника, пересекает среднее ребро и затем перемещается во второй выделенный прямоугольник. Можно исключить оставшиеся части графа выравнивания, которые занимают более половины площади, из дальнейшего рассмотрения. (справа) Поиск средних ребер (выделены полужирным) в ранее выделенных прямоугольниках.

Задача 9.1. Найти среднее ребро найдите в графе выравнивания в линейной памяти.

Вход: две аминокислотные строки.

Выход: среднее ребро графа выравнивания в виде «(*i, j*) (*k, l*)», где (*i, j*) соединяется с (*k, l*). Используется матрица оценок $BLOSUM_{62}$ и (линейный) штраф за индели, равный 5.

Пример входа:

PLEASANTLY

MEASNLY

Пример выхода:

(4, 3) (5, 4)

Псевдокод ниже для *LinearSpaceAlignment* описывает, как рекурсивно найти самый длинный путь в графе выравнивания, построенный для подстроки $v_{top+1}...v_{bottom}$ строки v и подстроки $w_{left+1}...w_{right}$ строки w. *LinearSpaceAlignment* вызывает функцию *MiddleNode*(*top, bottom, left, right*), которая возвращает координату i среднего узла (i, j), определяемую последовательностями $v_{top+1}...v_{bottom}$ и $w_{left+1}...w_{right}$. *LinearSpaceAlignment* также вызывает *MiddleEdge*(*top, bottom, left, right*), который возвращает →, ↓ или ↘ в зависимости от того, является ли среднее ребро горизонтальным, вертикальным или диагональным. Линейное по памяти выравнивание строк v и w строится путем вызова *LinearSpaceAlignment*(*0, n, 0, m*). Случай *left = right* описывает выравнивание пустой строки со строкой $v_{top+1}...v_{bottom}$, которая

тривиально вычисляется как оценка разрыва, образованного *bottom* – *top* вертикальными ребрами.

```
LinearSpaceAlignment(top, bottom, left, right)
  if left = right
    return alignment formed by bottom − top vertical edges
  if top = bottom
    return alignment formed by right − left horizontal edges
  middle ← ⌊(left + right)/2⌋
  midNode ← MiddleNode(top, bottom, left, right)
  midEdge ← MiddleEdge(top, bottom, left, right)
  LinearSpaceAlignment(top, midNode, left, middle)
  output midEdge
  if midEdge = "→" or midEdge = "↘"
    middle ← middle + 1
  if midEdge = "↓" or midEdge ="↘"
    midNode ← midNode + 1
  LinearSpaceAlignment(midNode, bottom, middle, right)
```

Задача 9.2. Реализовать *LinearSpaceAlignment* для решения задачи глобального выравнивания для большого набора данных.

Вход: две длинные (10000 аминокислотных) белковые строки, записанные в алфавите однобуквенных аминокислот.

Выход: максимальная оценка выравнивания этих строк, выравнивание, достигающее этой максимальной оценки. Используется матрица оценок $BLOSUM_{62}$ и (линейный) штраф за индели, равный 5.

Пример входа:

PLEASANTLY

MEANLY

Пример выхода:

8

PLEASANTLY

-MEA--N-LY

Аминокислотные последовательности белков, выполняющих одну и ту же функцию, вероятно, будут схожими, но эти сходства могут оказаться неуловимыми в случае отдаленных видов. Если сходство последовательностей слабое, попарное выравнивание не может идентифицировать биологически связанные последовательности. Однако одновременное сравнение многих последовательностей часто позволяет найти сходства, которые не удалось выявить при сопоставлении пар последовательностей.

Теперь можно использовать анализ пар последовательностей для создания алгоритма сравнения нескольких последовательностей. В рассмотренном трехмерном выравнивании A-доменов было обнаружено 19 консервативных столбцов:

```
YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGITYIKLTPSLFHTIVNTA
-AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYI
IAFDASSWEIYAPLLNGGTVVCIDYYTTIDIKALEAVFKQHHIRGAMLPPALLKQCLVSA

SFAFDANFESLRLIVLGGEKIIPIDVIAFRKMYGHTE-FINHYGPTEATIGA
-YEQKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS
----PTMISSLEILFAAGDRLSSQDAILARRAVGSGV-Y-NAYGPTENTVLS
```

Однако сходства между A-доменами не ограничиваются этими 19 столбцами, так как можно найти 10 + 9 + 12 = 31 полуконсервативных столбцов, каждый из которых имеет две совпадающие аминокислоты:

```
YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGITYIKLTPSLFHTIVNTA
-AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYI
IAFDASSWEIYAPLLNGGTVVCIDYYTTIDIKALEAVFKQHHIRGAMLPPALLKQCLVSA

SFAFDANFESLRLIVLGGEKIIPIDVIAFRKMYGHTE-FINHYGPTEATIGA
-YEQKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS
----PTMISSLEILFAAGDRLSSQDAILARRAVGSGV-Y-NAYGPTENTVLS
```

Множественное выравнивание строк v_1, ..., v_t, также называемое t-сторонним выравниванием, задается матрицей, имеющей t строк, где i-я строка содержит символы v_i по порядку с включенными пробелами. Предполагается, что ни один столбец в множественном выравнивании не содержит только символов пробела. В трехстороннем выравнивании ниже выделен самый популярный символ в каждом столбце использованием букв верхнего регистра:

```
  A  T  -  G  T  T  a  T  A
  A  g  C  G  a  T  C  -  A
  A  T  C  G  T  -  C  T  c
0  1  2  2  3  4  5  6  7  8
0  1  2  3  4  5  6  7  7  8
0  1  2  3  4  5  5  6  7  8
```

Матрица множественного выравнивания представляет собой обобщение парной матрицы выравнивания на более чем две последовательности. Три массива, показанные ниже этого выравнивания, хранят соответствующее количество символов в ATGTTATA, AGCGATCA и ATCGTCTC, встречающихся до заданной позиции. Вместе эти три массива соответствуют траектории в трехмерной сетке:

(0,0,0) → (1,1,1) → (2,2,2) → (2,3,3) → (3,4,4) → (4,5,5) → (5,6,5) → (6,7,6) → (7,7,7) → (8,8,8)

Поскольку граф выравнивания для двух последовательностей представляет собой квадратную сетку, граф выравнивания для трех последовательностей представляет собой кубическую сетку. Каждый узел в трехстороннем графе выравнивания имеет до семи входящих ребер, как показано на рисунке 9.11.

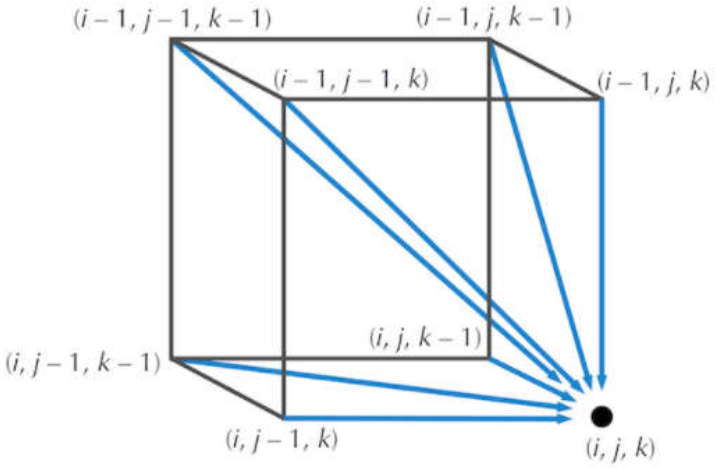


Рис. 9.11. Узел в графе для трехстороннего выравнивания.

Оценка множественного выравнивания определяется как сумма оценок для столбцов выравнивания (или, то же самое, веса ребер в пути выравнивания), при этом оптимальное выравнивание максимизирует этот показатель. В случае аминокислотного алфавита можно использовать очень общий метод подсчета, который определяется t-мерной матрицей, содержащей 21^t записей и описывающей оценки всех возможных комбинаций из t символов (представляющих 20 аминокислот и символ пробела). Интуитивно, более консервативные столбцы должны вознаграждаться более высокими баллами.

Задача. Найти множественное выравнивание с максимальной оценкой для заданной матрицы подсчета.

Вход: коллекция из *t* строк и t-мерная матрица *Score*.

Выход: множественное выравнивание этих строк, с максимальной оценкой (определенной матрицей *Score*).

Прямой алгоритм динамического программирования, примененный к t-мерному графу выравнивания, решает проблему множественного выравнивания для *t* строк. Для трех последовательностей *v*, *w* и *u* определяется $s_{i,j,k}$ как длина самого длинного пути от источника (*0, 0, 0*) до узла (*i, j, k*) в графе выравнивания. Рекуррентное соотношение для $s_{i,j,k}$ в трехмерном случае аналогично соотношению для парного выравнивания:

$$s_{i,j,k} = \max \begin{cases} s_{i-1,j,k} & + score(v_i, -, -) \\ s_{i,j-1,k} & + score(-, w_j, -) \\ s_{i,j,k-1} & + score(-, -, u_k) \\ s_{i-1,j-1,k} & + score(v_i, w_j, -) \\ s_{i-1,j,k-1} & + score(v_i, -, u_k) \\ s_{i,j-1,k-1} & + score(-, w_j, u_k) \\ s_{i-1,j-1,k-1} & + score(v_i, w_j, u_k) \end{cases}$$

В случае t последовательностей длины n граф выравнивания состоит из приблизительно n^t узлов, и каждый узел имеет до 2^t-1 входящих ребер, что дает общее время выполнения $O(n^t \cdot 2^t)$. По мере роста t алгоритм динамического программирования становится непрактичным. Было предложено много эвристик для субоптимальных множественных выравниваний для устранения этой проблемы.

В задаче множественного поиска наиболее длинных общих подпоследовательностей столбец матрицы выравнивания равен 1, если все символы столбца одинаковы, равен 0, если хотя бы один символ отличается.

Задача 9.3. Решить задачу множественного поиска наиболее длинных общих подпоследовательностей.

Вход: три строки ДНК длиной не более 10.

Выход: длина самой длинной общей подпоследовательности этих трех строк, множественное выравнивание трех строк, соответствующих такому выравниванию.

Пример входа:

ATATCCG

TCCGA

ATGTACTG

Пример выхода:

3

ATATCC-G-

---TCC-GA

ATGTACTG-

Стоит обратить внимание, что множественное выравнивание

```
AT-GTTaTA
AgCGaTC-A
ATCGT-CTc
```

индуцирует три попарных выравнивания:

```
AT-GTTaTA      AT-GTTaTA      AgCGaTC-A
AgCGaTC-A      ATCGT-CTc      ATCGT-CTc
```

Вопрос состоит в том, можно ли работать в противоположном направлении, комбинируя оптимальные попарные выравнивания в множественное выравнивание.

К сожалению, не всегда можно сочетать оптимальные парные выравнивания с множественным выравниванием, потому что некоторые парные выравнивания могут быть несовместимыми (как показано тремя попарными выравниваниями ниже).

```
CCCCTTTT--------      --------CCCCTTTT      TTTTGGGG--------
--------TTTTGGGG      GGGGCCCC--------      --------GGGGCCCC
```

Первое попарное выравнивание подразумевает, что CCCC встречается до TTTT в множественном выравнивании, построенном из этих трех парных выравниваний. Третье попарное выравнивание подразумевает, что TTTT встречается до GGGG в множественном выравнивании. Но второе попарное выравнивание подразумевает, что GGGG встречается до CCCC в множественном выравнивании. Таким образом, CCCC должен встретиться до TTTT, который должен встретиться до GGGG, который должен встретиться до CCCC, противоречие.

Чтобы избежать несовместимости, некоторые алгоритмы множественного выравнивания пытаются жадно построить множественное выравнивание из парных выравниваний, которые необязательно являются оптимальными. Жадная эвристика начинается с выбора двух строк, имеющих наивысшую оценку парного выравнивания (среди всех возможных пар строк), а затем использует это парное выравнивание как строительный блок для итеративного добавления одной строки за раз к растущему множественному выравниванию. На первом шаге будут выровнены две самые близкие строки, потому что они часто дают наилучшие шансы построить надежное множественное выравнивание. По этой же причине на каждом этапе выбирается строка, имеющая максимальную оценку против текущего выравнивания. Однако, вопросом является, что значит выровнять строку против выравнивания других строк.

Выравнивание нуклеотидных последовательностей с k столбцами может быть представлено в виде матрицы размера $4 \times k$, подобной той, которая показана на

рисунке 9.12, которая содержит нуклеотидные частоты каждого столбца (выравнивание аминокислот представлено профильными матрицами размера $20 \times k$). Жадная эвристика множественного выравнивания добавляет строку к текущему выравниванию, создавая парное выравнивание между строкой и профилем текущего выравнивания. В результате проблема построения множественного выравнивания t последовательностей сводится к построению $t - 1$ попарных выравниваний.

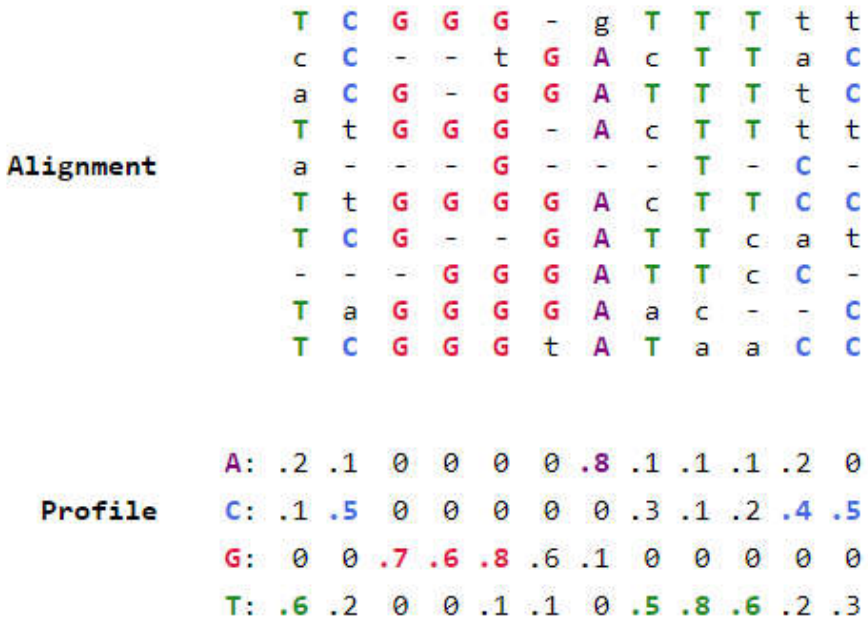


Рис. 9.12. Матрица профиля для множественного выравнивания из 10 последовательностей. Каждый столбец матрицы профиля в сумме дает (1 – частота символа пробела). Самые популярные нуклеотиды в каждом столбце показаны как буквы в верхнем регистре.

Хотя жадные алгоритмы множественного выравнивания хорошо работают для подобных последовательностей, их производительность ухудшается для разнородных последовательностей, потому что жадные методы могут быть введены в заблуждение парным выравниванием. Если первые две последовательности, выбранные для построения множественного выравнивания, выровнены таким образом, который несовместим с оптимальным множественным выравниванием, тогда ошибка в этом начальном парном выравнивании будет распространяться вплоть до окончательного множественного выравнивания.

Список литературы

[1] http://bioinformaticsinstitute.ru/teachers/vyahhi

[2] N.C. Jones, P.A. Pevzner. Introduction to Bioinformatics Algorithms. The MIT Press, Cambridge, MA 2004.

[3] P.A. Pevzner., P. Compeau. Bioinformatics Algorithms: An Active Learning Approach, Active Learning Publishers, 2014.

Игорь Ильясович Юсипов
Алёна Игоревна Калякулина
Михаил Васильевич Иванченко

АЛГОРИТМЫ БИОИНФОРМАТИКИ

Учебное пособие (лекционный материал)

Федеральное государственное автономное образовательное учреждение
высшего образования “Нижегородский государственный
университет им. Н.И. Лобачевского”.
603950, Нижний Новгород, пр. Гагарина, 23.