Programación Científica y Algoritmos Básicos

José R. Dorronsoro
Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
28049 Madrid, Spain

Description

- Lecturers: Ana González (coordinator), José Dorronsoro
- Contents:
  - Introduction to Algorithms and Data Structures
  - Dynamic Programming. Levenshtein Distances.
  - Eulerian Paths and DNA Sequencing
  - Algorithm design techniques applied to motif search and sequence alignment
- **First Part**:
  - First three items are a fast review of Chapter 1 in Jones & Pevzner's **An Introduction to Bioinformatics Algorithms**
  - A mix of theoretical classes, lab assignments, problems
  - Evaluation: miniproject + take home

Outline

Algorithms

Wirth's Equation

- Actually is the title of his book Algorithms + Data Structures = Programs
  - Very good book with examples written in Pascal
- Just add input and output to the equation to get useful software
- But first we have to define and clarify what we mean by **Algorithms** and by **Data Structures**

Algorithms

- Many definitions, none too precise
- Adapting from Wikipedia

  a set of rules that precisely define a sequence of operations to perform some task and which eventually stop
- Usually written in **pseudocode**:
  - Intermediate between natural language and computer code
  - Not necessarily directly understandable by a computer but close by
  - Simple Python often fine in “simple" tasks
- Three building blocks:
  - **Sequential blocks**
  - **Selections**
  - **Repetitions** or **Loops**

Sequences, Selections and Loops

- **Sequential blocks**: blocks of (ordinary) sentences that execute sequentially in their entirety
  - Sentences may have just straight computations or several calls to functions
  - Order of execution according to gravity's law
  - In Python: blocks made of sentences with same indentation
- **Selections**: sentences where execution branches to different blocks according to some condition
  - In Python: `if condition:`, `elif condition:`, `else:`
- **Repetitions** or **loops**: a sentence block is repeated while some condition holds
  - In Python: `while condition:` and also `for` iterations

First Example: Euclid's Algorithm

- Computes the g.c.d. of two positive numbers a, b by repeatedly computing $r = a\%b$ and replacing a by b and b by r while $r > 0$
- In Python:

```
def euclid_gcd(a, b):
    while b > 0:
        #print(a, b)
        r = a % b
        a = b
        b = r

    return a
```

- Or more concise (pythonic?):

```
def euclid_gcd(a, b):
    while b > 0:
        a, b = b, a % b

    return a
```

But ... Watch Out!!

- In the last version of Euclid's algorithm:
  - What happens if a is 0?
  - What happens if a, b or both are negative?
- Wirth's equation is nice and broadly correct but programs also need quite a bit of **exception handling**, i.e.
  - Detect exceptional situations
  - Tell the program what to do when they arise
- Argument errors are easy to prevent and handle
- Execution exceptions, i.e., things going wrong during execution are harder to detect and prevent
- Programming should be quite **defensive**

Data Structures

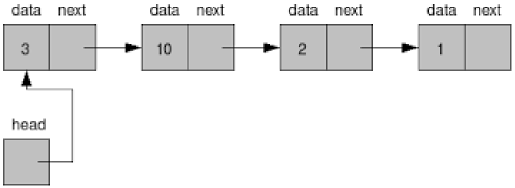
- Algorithms work on data
- Single variables are OK for simple algorithms
- **Data structures**: ways to organize more complex data for more advanced algorithms
- Simplest data structures: **strings, lists, arrays**
- More advanced: **dictionaries, sets**
- All of them built in in Python (arrays better through Numpy)
- Advanced structures: **linked lists, trees, graphs**
  - Available in Python through imported modules

Strings, Lists, Arrays and Dicts

- String, list and array elements are accessed through **indices**
  - Similar at first sight (a string is a list of characters?) but in fact quite different
  - Our arrays will be defined and handled by the `numpy` module
- Dicts are made up of `key:value` pairs
- All are Python **objects** with
  - **attributes**: variables with object information and
  - **methods**: functions that act on the object's content
- `dir(object)` lists all of them
  - Try `dir(1)`

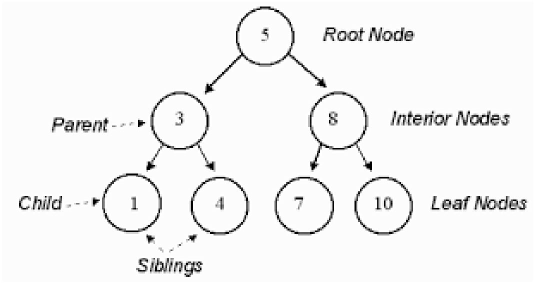
Linked Lists

- Made up of individual **nodes** with fields `data`, `next`
  - `data` contains the node's info
  - `next` points to the next node
- They are **dynamic versions** of arrays
- Useful when the number of nodes and/or their location is not known in advance



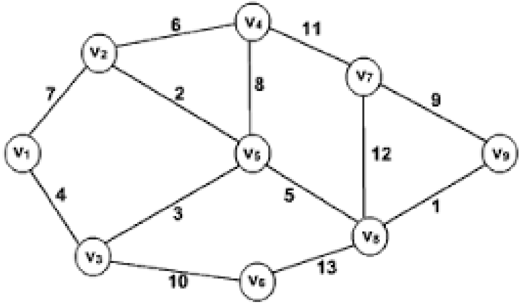
Trees

- Hold data nodes organized in a **hierarchical** way with a single **root** node and the other ones having a **parent** and perhaps **children**



Graphs

- Made up of **nodes** or **vertices** connected by **edges**
- Possibly the most general data structure: can represent road maps, social networks, protein interactions, ...



Algorithm Design

Writing Algorithms

- Algorithm writing (and programming in general) is usually done on an ad–hoc basis
- It is a **creative act**: must follow programming rules but also requires **imagination, creativity and experience**
    - The same happens with ordinary writing: we cannot fill an empty page just with grammar rules
    - Programming also requires hard work, lots of practice and, also, **quite a bit of algorithm reading**
- Sometimes we can take advantage of general **design techniques**
    - Derived from long experience in problem solving and algorithm analysis
    - Cannot be applied as automatic rules of thumb
    - But may have a wide range of applicability
- We will consider three here: **greedy** algorithms, **divide and conquer** (a.k.a. **recursive**) algorithms and **dynamic programming**

The Change Problem

- Assume we have work as supermarket cashiers and our clients want change in as few coins as possible
- How can we proceed?
- Simplest idea: give at each step the largest coin smaller than the amount that remains to change
- Example: how to give change of 3,44 euros?
    - Easy: one 2 euro coin, one 1 euro coin, two 20 cent coins, two 2 cent coins
- Have to write down the algorithm but the general idea is **greedy**:
    - We try to minimize **globally** the total number of coins
    - Using **locally** at each step the largest coin possible to minimize the amount still to change

The Greedy Change Algorithm

- We will work with an ordered list of coin values, say

  `1, 2, 5, 10, 20, 50, 100, 200`

  and save the number of coins of each type used in a dict

```
def change(c):
    """
    docstring: write it always!!!
    """
    assert c >= 0, "change for positive amounts only"

    l_coin_values = [1, 2, 5, 10, 20, 50, 100, 200]
    d_change = {}

    for coin in sorted(l_coin_values)[ : : -1]:
        d_change[ coin ] = c // coin
        c = c % coin

    return d_change
```

- Exercise: pass the coin list as a parameter and add an appropriate assert

Does It Work?

- At first sight yes, but ...
- Try it to give change of 7 maravedís with coin values `1, 3, 4, 5`
- What is the answer of the algorithm?
  - Correct answer: just 2 coins, one of 4 maravedís and one of 3
- This often happens with greedy algorithms
  - They are very natural but may give wrong results!!
  - Later we will give a correct algorithm using **Dynamic Programming**, another general algorithm design technique

The Towers of Hanoi

- We are given a set of 64 gold disks of different sizes piled on peg A in increasing sizes, and two other empty pegs B, C
- We want to move the first stack to B one disk at a time using C as an auxiliary peg obeying the rule:
    - **No disk may be placed on top of a smaller disk**
- Easy for 2 disks, not too hard for 3. But for 4 ...???
- Simple **recursive** solution: for N disks
    - Move the first $N-1$ disks from peg A to C using B as the auxiliary peg
    - Move the remaining disk from A to B
    - Move the $N-1$ remaining disks from C to B using A as the auxiliary peg
- Very easy to program

Printing Disk Moves

- The following Python code prints the disk moves required:

```
def hanoi(n_disks, a=1, b=2, c=3):
    assert n_disks > 0, "n_disks at least 1"

    if n_disks == 1:
        print("move disk from %d to %d" % (a, b))
    else:
        hanoi(n_disks - 1, a, c, b)
        print("move disk from %d to %d" % (a, b))
        hanoi(n_disks - 1, c, b, a)
```

- But watch out the running times even for small `n_disks`
- In fact the general Hanoi problem is **extremely costly** even for moderate disk numbers

When Will The World End?

- According to a legend, there is large room in a Kashi Vishwanath temple with three posts where 64 golden disks are placed and Brahmin priests keep on moving them
- And the world will end when the last disk move is done
- Q: How many moves will be needed?
- To get an idea
  - Count first the number of moves for small values of N
  - Modify the previous Hanoi code to get a function that returns the number of moves
  - And then decide whether to start worrying
- By the way, the legend is false:
  - The whole thing is a mathematical game devised by Edouard Lucas

Divide And Conquer

- Recursive algorithms often derive from a **Divide and Conquer** strategy:
  - Divide a problem P in M subproblems P_m
  - Solve these separately getting solutions S_m
  - Combine these solutions in a solution S of P
- Two subproblems in Hanoi:
  - P_1 is the subproblem of moving $N - 1$ disks from A to C using B
  - P_2 is the subproblem of moving $N - 1$ disks from C to B using A
  - And we combine the moves according to the Python code
- Efficient algorithms if subproblems are substantially smaller
  - Not the case in Hanoi
- Another clearer example is **binary search** (we'll see it in the exercises)

Algorithm Efficiency

Things To Keep In Mind

- First of all **algorithms must be correct**
  - A fast but wrong algorithm is useless
- It is also desirable that they do not require (much) extra memory
  - The `hanoi` function fulfills this: only its parameters are used
  - Something to watch out in Bioinformatics
- It is also highly desirable that they are as **fast** as possible
  - But ... an algorithm must “read" its inputs
  - If there are many and they are large, the algorithm will likely be slow
  - But desirable execution times should not be far above from the same “order of magnitude" than its inputs' size
- How do measure execution times?

Estimating Execution Times I

- First of all, **forget about just measuring actual times**
  - They depend on the language, the machine, the programmer and, of course, the inputs
  - They are thus too context–dependent to allow meaningful **generalizations**
- We focus instead on **abstract times** measured by counting the **key operations** the algorithm performs on a given input
- For iterative algorithms we usually look for the key operation on the innermost loop
  - Counting how many times these key operations are performed will give us a good estimate of the time their algorithms will take
- This way, the cost of the change algorithm is given by the length of the coin list

Estimating Execution Times II

- The analysis of recursive algorithms is (much) harder
- We can see that possible key operations may be
  - `print("move disk from %d to %d" % (a, b))` for Hanoi
- But while they appear explicitly in the code, they also take place **inside** the recursive calls
- This results in **recurrent estimates** for the cost of recursive algorithms that are often tricky to write and to solve
- Let's try to develop some general strategies on much simpler, loop–based algorithms

Matrix Multiplication

- Well known algorithm: $c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$ (and quite costly!!)
- Simple (and quite bad) Python code:

```
def matrix_multiplication(m_1, m_2):
    """ ..."""
    n_rows, n_interm, n_columns = \
        m_1.shape[0], m_2.shape[0], m_2.shape[1]

    m_product = np.zeros( (n_rows, n_columns) )

    for p in range(n_rows):
        for q in range(n_columns):
            for r in range(n_interm):
                m_product[p, q] += m_1[p, r] * m_2[r, q]

    return m_product
```

- Key operation: clearly `m_1[p, r] * m_2[r, q]`
- How many? Assuming square matrices with N rows/columns, obviously $N \times N \times N = N^3$
  - Substantially larger than problem size $N^2 + N^2 = 2N^2$

Linear Search

- Perhaps the simplest algorithm to search a key in a list:
  - Just compare the key agains the list's elements until a match (if any) is found
- Simple Python code:

```
def linear_search(key, l_ints):
    """ ..."""
    for i, val in enumerate(l_ints):
        if val == key:
            return i

    return None
```

- Key operation: clearly `if val == key:`
- Finding `l_ints[0]` requires just one key operation
- Finding `l_ints[-1]` requires `N = len(l_ints)` key operations
- If `key` is not in `l_ints`, `linear_search` will also require `N` key operations

The o, O and Θ Notations I

- It seems that the cost in key operations is given by some function $f(N)$ of input size N
  - We have $f_{MM}(N) = N^3$, $f_{LS}(N) = N$
- We can thus **compare two algorithms** A, B by **comparing their cost functions** f_A, f_B
- We assume that the cost functions are positive and increasing (this should be the case with the abstract execution times of algorithms)
- Given such a pair f, g, we say that $f = o(g)$ if $\frac{f(N)}{g(N)} \to 0$ when $N \to \infty$
  - The growth of f is “clearly" smaller than that of g
- Also, $f = O(g)$ if we can find C and N_C s. t. $f(N) \leq Cg(N)$ if $N \geq N_C$
  - g will be bigger than f eventually ($N \geq N_C$) and with help of C

The o, O and Θ Notations II

- Finally, $f = \Theta(g)$ if $f = O(g)$ and $g = O(f)$
- (Very) **Informally** we will understand the preceding as
  - $f < g$ when $f = o(g)$
  - $f \leq g$ when $f = O(g)$
  - $f \simeq g$ when $f = \Theta(g)$ (and, hence, $g = \Theta(f)$)
- O and Θ may be quite messy to check in general, but if $\lim_{N\to\infty} \frac{f(N)}{g(N)} = L \neq 0$, then $f = \Theta(g)$
- In the preceding examples
  - $N^2 = o(N^3)$ and also, but much less precise, $N^2 = O(N^3)$
  - $\frac{N}{2} = \Theta(N)$ and, also, $N = \Theta\left(\frac{N}{2}\right)$

Complexity of an Algorithm

- From the preceding it seems that given an algorithm A with input I we can “measure" abstract execution times as follows
  - We can identify a **key operation** in A and estimate its abstract work on I by the number $n_A(I)$ of times A executes it on I
  - We can assign a **size** N to the input I
  - We can find a **function** $f_A(N)$ so that $n_A(I) = O(f_A(N))$
- In some cases we will be able to refine this to $n_A(I) = \Theta(f_A(N))$
- We say that $f_A(N)$ gives (is?) the **complexity** of A over entries of size N
- We have $n_{MM}(A, B) = N^3$ with N matrix dimension.
- For Hanoi we have $n_{hanoi}(N) = 2^N - 1$
- For successful searches we have $n^e_{LS}(N) \leq N$, but sometimes $n^e_{LS}(N) = 1$
  - So we say that the **worst** case of LS is $W(N) = N$, as always $n_{LS}(k, l\_ints) \leq N$

From Abstract Times to Real Times

- In IPython the magic command `%timeit` allows to estimate execution times of simple functions

```
a = np.ones((100, 100)); b = np.eye(100)
%timeit -n 10 -r 1 matrix_multiplication(a, b)
```

- If it gives us a time estimation of, say, 1 second, what can we expect for matrices with dimensions 500?
  - Since $500 = 5 \times 100$, $500^3 = 125 \times 100^3$, i.e., 125 times bigger
  - Thus, we should expect `%timeit` to report about 125 seconds
- This is not %100 precise, but gives a ball park estimate
- Hence, `matrix_multiplication` is quite costly but `hanoi` is truly awful
- And watch out: straight Python code can be quite slow because of language overheads
  - Heavy duty libraries such as `numpy`, `pandas`, `scipy` or `sklearn` do their work in C or C++ compiled code

Revisiting The Change Problem

Back To Giving Change

- The key trick is to decompose the problem into increasing subproblems and to get a formula to go from one subproblem to the next
- Assume we want to change an amount C with $v_1 = 1, \ldots, v_N$ coin denominations
- Let $n(i, c)$ be the minimum number of coins to change an amount c using only the first i coins
  - What we want is $n(N, C)$ which we will get from the easier to understand $n(i, c)$
  - Observe that $n(1, c) = c, n(i, 0) = 0$
- Depending on whether or not coin i enters the change of c we have the following equations

$$\begin{aligned} n(i, c) &= n(i-1, c) && \text{if coin } i \textbf{ doesn't enter} \text{ the change,} \\ &= 1 + n(i, c - v_i) && \text{if coin } i \textbf{ enters} \text{ the change} \end{aligned}$$

The DP Change Algorithm

- Therefore, we arrive at

$$n(i, c) = \min\{n(i-1, c), 1 + n(i, c - v_i)\}$$

  with a $O(1)$ cost
  - And the algorithm is (more or less obviously) correct
- Thus, we simply have to fill the DP matrix but with a cost

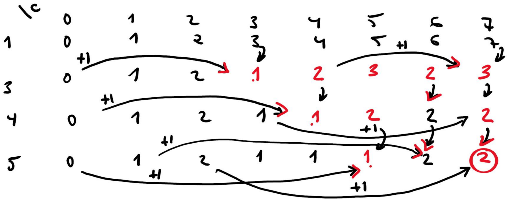
$$N \times C \times O(1) = O(N \times C) = O(N \times 2^{\lg C})$$

  exponential on the size $\lg C$ of C
- In fact, the change problem is **NP–complete**
  - Nice podcast at **BBC's In Our Time**
- But recall there is a linear complexity algorithm for the Euro coin system

Giving DP Change Algorithm

- Giving change via DP of 7 with coin values `1, 3, 4, 5`

cambio de 7 con monedas {1, 3, 4, 5}



\c	0	1	2	3	4	5	6	7
1	0	1	2	3	4	5	6	7
3	0	1	2	1	2	3	2	3
4	0	1	2	1	1	2	2	2
5	0	1	2	1	1	1	2	$\boxed{2}$

DP String Algorithms

Editing Strings

- Assume we have two strings and want to edit, i.e., transform one string `s_1` into another `s_2` according to the following allowable operations
  - Change one character in either string
  - Insert a character in either string
  - Delete a character from either string
- Notice that removing a delete a character from string `s_1` is equivalent to adding a character to string `s_2`
- Also we can keep one of the strings fixed and do all the change/add/delete operations on the other

The Edit Distance

- The **Edit Distance** `d(s_1, s_2)` between two strings `s_1` into another `s_2` is the minimum number of edit operations that we have to make to turn, say, `s_2` into `s_1`
- For instance, the edit distance between `unnecessarily` and `unescessaraly` is 3:

```
unne cessarily
  a d      c
un escessaraly
```

  as we just add to the second string an `'n'` , delete an `'s'` and change `'a'` into `'i'`
- We can conceptually substitute the change operation by an insert + delete combination

Approximate String Searching

- These arrangements of character plus blanks are sometimes known as **alignment matrices**
    - By convention, the bottom string is the "target" sequence
- The edit distance can also be used for **approximate string searches**: given a string `s` find another `t` in a string list so that their edit distance is minimal
- Below we will penalize these mismatches with a $+1$ scoring penalty
- More generally one can consider scoring matrices δ_{s_i,t_j} associated to particular character mismatches
- Sequence alignment with general scoring penalties are very important in **DNA sequence comparison**
    - Some more details later on and in Jones and Pevzner, Section 6.7

A Dynamic Programming Solution

- Given the full strings S and T with M and N characters respectively, consider the substrings

$$S_i = [s_1, \dots, s_{i-1}, s_i], \quad T_j = [t_1, \dots, t_{j-1}, t_j]$$

- If $d_{i,j}$ is the edit distance between S_i and T_j, we want to find $d_{M,N} = \mathrm{dist}(S, T)$
- Observe that if $s_i = t_j$, then $d_{i,j} = d_{i-1,j-1}$
- And if $s_i \neq t_j$ we have three options
  - Change t_j into s_i; then $d_{i,j} = 1 + d_{i-1,j-1}$
  - Delete t_j from T_j; then $d_{i,j} = 1 + d_{i,j-1}$
  - Delete s_i from S_i; then $d_{i,j} = 1 + d_{i-1,j}$

Filling The DP Matrix

- We thus arrive to the following equations for the Edit Distance problem

$$\begin{aligned} d_{i,j} &= d_{i-1,j-1} & \text{if } s_i = t_j; \\ &= 1 + \min\{d_{i-1,j-1}, d_{i,j-1}, d_{i-1,j}\} & \text{if } s_i \neq t_j \end{aligned}$$

  which result in an easy to apply algorithm
- Example: find the edit distance between `biscuit` and `suitcase`
- The cost is clearly $O(M \times N)$, no longer NP–complete but, still, costly
- And an initial memory cost is also $O(M \times N)$, just awful, as problem size is $O(M + N)$
  - But, in fact, quite easy to alleviate if we only care about $d_{M,N}$

An Example

- Example: find the edit distance between `biscuit` and `suitcase`

	∅	b	i	s	c	u	i	t
∅	0	1	2	3	4	5	6	7
s	1	1	2	↘ 2	3	4	5	6
u	2	2	2	3	3	↘ 3	4	5
i	3	3	↘ 2	3	4	4	↘ 3	4
t	4	4	3	3	4	5	4	↘ 3
c	5	5	4	4	↘ 3	4	5	4
a	6	6	5	5	4	4	5	5
s	7	7	6	↘ 5	5	5	5	6
e	8	8	7	6	6	6	6	6

Alignment with general penalties

- Sequence alignment with general scoring penalties is very important in DNA sequence comparison
- We can consider different penalties for switching s_i or t_j into the other or for inserting one or the other
    - For switching we may assume a penalty of α_{s_i,t_j}
    - For inserting we may assume a character independent penalty of δ
- We know talk about **alignment costs** $c_{m,n}$ and the new DP penalty equations are

$$c_{i,j} \quad = \quad \min\left\{\alpha_{s_i,t_j} + c_{i-1,j-1},\ \ \delta + c_{i,j-1},\ \ \delta + c_{i-1,j}\right\}$$

  where we assume $\alpha_{c,c} = 0$
- This (plus quite a bit of Biochemistry) is at the core of the **Smith–Waterman** and **Needleman–Wunsch** sequence alignment methods
    - More biology–oriented details in Jones and Pevzner, Sections 6.4–6.7

The Longest Common Substring

- Given again strings S, T, we want to find the longest (non necessarily consecutive) common substring (LCS) to both
- As before, let first $\ell_{i,j}$ be the length of the LCS between S_i and T_j. We have now:

$$\begin{aligned} \ell_{i,j} &= 1 + \ell_{i-1,j-1} && \text{if } s_i = t_j; \\ &= \max\{\ell_{i,j-1}, \ell_{i-1,j}\} && \text{if } s_i \neq t_j \end{aligned}$$

  - Again this results in an easy to apply algorithm with cost $O(M \times N)$
  - Example: find the LCS between `biscuit` and `suitcase`
- Good (and easy) exercise: write down a Python function to find the length of the LCS
- Very good exercise: modify the previous Python function so that it gives one of the possible LCS

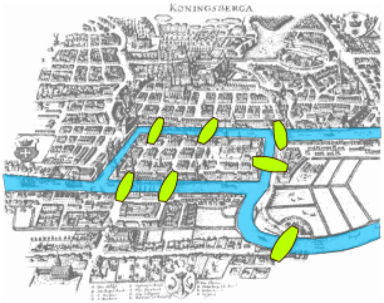
Other Problems With DP Solutions

- There is a large number of problems, essentially different and often useful in practice, that can be solved via DP algorithms
- We list some of them:
  - Find the optimal ordering to multiply N matrices
  - Find optimal binary search trees
  - Find RNA's Secondary Structure
  - Cut a rod of a lenght L in pieces that get as much money as possible when sold
  - Finding the longest palindromic substring of a given string
  - Fit Least Squares segments to data
  - Speech recognition through the Viterbi algorithm
- And many more!

Eulerian Paths

The Bridges of Königsberg

- The bridges of Königsberg (East Prussia) over the Pregel river circa 1700:



- The problem: find a promenade that crosses all bridges but only once
- Tool: a **graph** representation of the problem

Graphs

- Graph: Pair $G = (V, E)$ of a set V of vertices (nodes) and a set E of edges (u, v) with $u, v \in V$
- Edges imply direction: in (u, v) we go from u to v
- In general, graphs are **directed**
- **Undirected** graphs: $(u, v) \in E$ iff $(v, u) \in E$
- **Multigraphs**: graphs with multiple copies of an edge (u, v) or with self–edges (u, u)
- Standard representation: adjacency lists

An Example

- For a graph such as

$$\begin{tikzcd}
1 \arrow[rr] \arrow[dr] & & 2 \arrow[dl] & \\
& 3 \arrow[rr] & & 4 \arrow[ul]
\end{tikzcd}$$

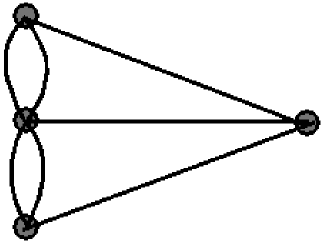
  its adjacency list is

```
v   l_adj
-   -----
1 - 2 - 3
2 - 3
3 - 4
4 - 2
```

- We define $out(w) = \#$edges from w, $in(w) = \#$edges into w
- In undirected graphs edges appear “twice"
  - We draw them only once without arrows
  - Then $out(u) = in(u)$ for all $u \in V$ and we just talk of $deg(u)$

The (Multi) Graph of Königsberg

- We can depict the bridges of Königsberg as an **undirected multigraph** (i.e., we allow for multiple bridges/edges between two nodes)



- The problem: find a path that passes **through all edges but only once**
- Such a path in a multigraph is called an **Eulerian path** (EP)

Eulerian Paths on Undirected Graphs

- Let $\pi = \{(u = u_0, u_1), \ldots, (u_{K-1}, u_K = v \neq u)\}$ be such an EP
  - When $u = v$ we talk of a circuit
- If $w \neq u, v$ is a node in π, each time we enter w we subtract 1 from $deg(w)$ and also when we leave w;
  - Since at the end we have passed through all the edges of w, we must have at the beginning $deg(w)$ even
- Similarly each time we pass through u inside π we subtract 2 from $deg(u)$; moreover, we also subtract 1 when we start from u
  - Thus, we must also have $deg(u)$ odd
- Similarly each time we pass through v we subtract 2 from $deg(v)$ and we also subtract 1 when we end π at v
  - Thus, we must also have $deg(v)$ odd
- Thus, **a necessary condition to have an EP is that $deg(w)$ is even for all w except the first node u and the final one v of π**

Euler's Insight

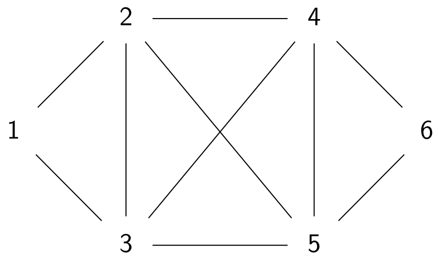
- Since all the nodes in the previous multigraph have odd degrees, Euler concluded that **no Eulerian path is possible in Königsberg**
- In fact, Euler also proved that the condition is sufficient:
  If $deg(w)$ is even for all nodes w of an undirected graph G, except for two ones u, v, then there is an Eulerian path in G that either starts at u and ends in v or viceversa
- A similar result holds for Eulerian **circuits**:
  There is an Eulerian circuit in an undirected graph G if and only if $deg(w)$ is even for all nodes w

The Bridges of Madrid?

- Madrid has a river with many bridges but no islands
- Does Madrid have a (quite long) Euclidean path? Or perhaps a circuit?
- More generally, think of a city with a river with N bridges and no islands
- Does it have any Eulerian promenade?
- Easy exercise: look at the multigraph and count the degree of its nodes

How to Find an EC

- Assuming an EP exists, the basic idea is start a walk until we cannot go on and re–start the walk if needed
- Example:



- Cost? We simply take out edges, so it is $O(|E|)$

Walking to Find an EC

- The steps are

g	AdL$_1$
2	1 ~~−~~ 2 ~~−~~ 3
4	2 ~~−~~ 1 ~~−~~ 3 − 4 − 5
4	3 ~~−~~ 1 − 2 − 4 − 5
4	4 − 2 − 3 − 5 − 6
4	5 − 2 − 3 − 4 − 6
2	6 − 4 − 5

g	AdL$_2$
2	2 ~~−~~ 4 ~~−~~ 5
2	3 ~~−~~ 4 ~~−~~ 5
4	4 ~~−~~ 2 ~~−~~ 3 − 5 − 6
4	5 ~~−~~ 2 − 3 − 4 − 6
2	6 − 4 − 5

$\boxed{1}$ all gr are even ⇒ there is an EC

$$\tilde{\pi} = \pi_1 : \quad 1^1 - 2^{\boxed{2}} - 3^2 - 1^0$$

not an EC ⇒ we re-start on the residual graph fr

$\boxed{2}$ $\pi_2 : \ 2^1 - 4^2 - 3^0 - 5^2 - 2^0$

and we stich it with $\tilde{\pi} = \pi_1$

$$\tilde{\pi} : \begin{array}{cccccccc} 1^0 - & 2^0 & & & 2^0 & - 3^0 - 1^0 \\ & \| & \boxed{2} & & \| & \\ & 4 & = 3^0 = & & 5^2 & \end{array}$$

⇒ still not EC

⇒ re-start from

$\boxed{3}$ $\pi_3 : \ 4^1 - 5^0 - 6^0 - 4^0$ and stitching wit

$$\tilde{\pi} : \begin{array}{ccccc} 1 - 2 - & 4 & & 4 & - 3 - 5 - 2 - 3 - 1 \\ & \| & & \| & \\ & 5 & = & 6 & \end{array}$$

⇒ EC !!!

Hamiltonian Paths

- If G is an undirected connected graph, a **Hamiltonian path** (HP) is a path on G that visits each node **only once**
  - There is a similar definition for Hamiltonian circuits
- Finding HPs may be trivial in some cases, as for complete graphs
- There are also sufficient conditions for special graphs
- But for general graphs, while finding ECs has an $O(|E|)$ cost, finding HPs is much costlier
- In fact, finding HPs in general graphs is another example of an **NP–complete** problem
- Well, and so what??

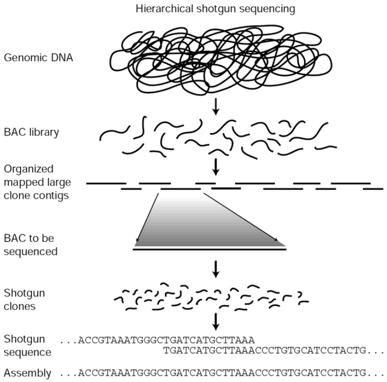
Hamiltonian and Eulerian DNA Sequencing

DNA Sequencing

- **Note:** this is a very, very light description of one of the many approaches to DNA Sequencing
  - See DNA sequencing at 40: past, present and future (Nature, Oct. 2017) for a very recent review
- Goal: decompose a gene into a sequence of four letters $\{A, C, G, T\}$
- **Shotgun sequencing** broadly follows a four step process:
  - Break the gene into random short “reads" of 100–500 bases
  - Identify read subsequences by hybridizing them on a DNA microarray
  - Reconstruct each read from these subsequences
  - Reconstruct the entire gene from the reads
- First two steps: biochemistry
- Third step: Hamiltonian or (better) Eulerian paths
  - The approach proposed in An Eulerian path approach to DNA fragment assembly (Pevzner, Tang, Waterman; PNAS, vol. 98, 2001)
- Fourth step: compute the Shortest Superstring Problem (a version of the Traveling Salesman Problem (TSP) , another NP problem)
  - Plus more algorithms and a lot of biochemistry

Shotgun Sequencing

- Idealized hierarchical shotgun sequencing strategy

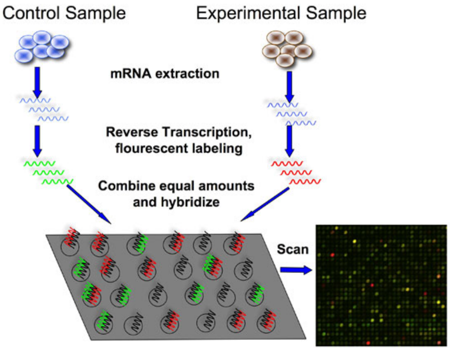


Hierarchical shotgun sequencing
Genomic DNA
BAC library
Organized mapped large clone contigs
BAC to be sequenced
Shotgun clones
Shotgun sequence
...ACCGTAAATGGGCTGATCATGCTTAAA
TGATCATGCTTAAACCCTGTGCATCCTACTG...
Assembly
...ACCGTAAATGGGCTGATCATGCTTAAACCCTGTGCATCCTACTG...

From Nature

Sequencing by Hybridization

- Scheme of the process:


Control Sample
Experimental Sample
mRNA extraction
Reverse Transcription, flourescent labeling
Combine equal amounts and hybridize
Scan

From *bitesizebio.com/7206/introduction-to-dna-microarrays*

Microarray Hybridization

- Put all the posible lenght ℓ probes, i.e., DNA subsequences of a fixed lenght ℓ, into the spots of a microarray
- Put a drop of fluorescently labeled DNA into each microspot of the array
- The DNA fragment hybridizes with those microspots that are complementary to a certain substring of length ℓ of the fragment
- This way we get all possible lenght ℓ subsequences that make the fragment but they are **unordered**
    - They follow the order in the microarray but **not the one in the sequence**

ℓ–mers and the Spectrum

- We call the sequence on each one of the probes an ℓ**–mer**
- The ℓ**–spectrum** $sp(S, \ell)$ of a sequence S is the set of all the ℓ–mers from S
- For instance, `S = [TATGGTGC]` we have

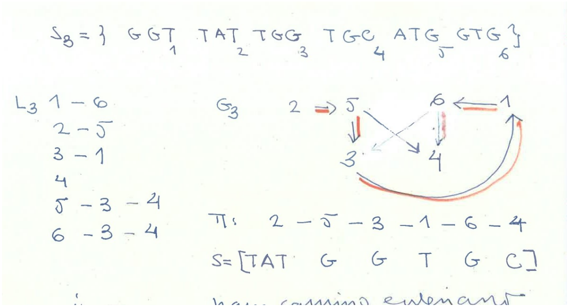
  `sp(S, 3)= {TAT, ATG, TGG, GGT, GTG, TGC}`
- After hybridization, the hybridized probes in the microarray give us **an unordered version of** $sp(S, \ell)$ that we have to reorder to recover S
- Define the **overlap** $\omega(s_1, s_2)$ between two ℓ–mers s_1, s_2 as the **longest length of a suffix of** s_1 **that is also a prefix of** s_2
  - Clearly have $\omega(s_1, s_2) \leq \ell - 1$
- Now, **if** s_2 **follows** s_1 **in** S**, we must have** $\omega(s_1, s_2) = \ell - 1$

Sequencing by Hamiltonian Paths

- We can reconstruct the sequence S by finding an **ordering** $s_{i_1}, \dots, s_{i_K}$ **of** $sp(S, \ell)$ **such that** $\omega(s_{i_j}, s_{i_{j+1}}) = \ell - 1$
- This suggests to define the graph $G_\ell(S) = (V_\ell, E_\ell)$ where
  - $V_\ell = sp(S, \ell)$ and
  - $(s, s') \in E_\ell$ iff $\omega(s, s') = \ell - 1$
- Notice that reconstructing S is equivalent to **pass once through all the nodes of** $G_\ell(S)$
- In other words, **we can reconstruct** S **by finding a Hamiltonian path in** $G_\ell(S)$

Sequencing by Hamiltonian Paths II

- **Example:** consider `S = [TATGGTGC]` and the unordered 3–spectrum `sp(S, 3)= {GGT, TAT, TGG, TGC, ATG, GTG}`
- By inspection, the adjacency list and graph, the HC and the recovered sequence are



Sequencing by Eulerian Paths

- The obvious problem of HP sequencing is the lack of efficient algorithms to solve the HP problem
- The alternative is to try to **have ℓ–mers on the edges** instead of on nodes
- If $s \in sp(S, \ell)$ and s_1 is its $\ell - 1$ prefix and s_2 its $\ell - 1$ suffix, we can consider s as the edge connecting nodes s_1 and s_2
    - Now we have $\omega(s_1, s_2) = \ell - 2$
- We define now the graph $G_{\ell-1} = (V_{\ell-1}, E_{\ell-1})$ where
    - $V_{\ell-1} = sp(S, \ell - 1)$ and
    - $(s, s') \in E_{\ell-1}$ iff they are respectively **prefix and suffix of an $s \in sp(S, \ell)$**
- Notice that now **reconstructing S is equivalent to pass once over all the edges of $G_{\ell-1}$**
- In other words, **we can reconstruct S by finding a Eulerian path in $G_{\ell-1}$**

Eulerian Paths on Directed Graphs

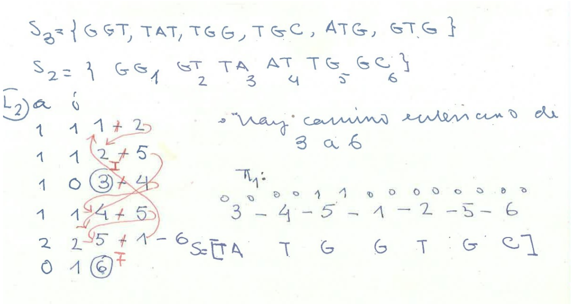
- However, notice that $G_{\ell-1}$ is a **directed** graph and we have to adapt the Eulerian path theory to these graphs
- In an directed graph $G(V, E)$ we have to distinguish between incident and adjacent edges
- For any $u \in V$, we say that (u, v) is an **adjacent** edge for u and an **incident** edge for v
- Recall that the **indegree** $in(u)$ of u is the number of incoming edges to u
- And the **outdegree** $out(u)$ is the number of outgoing edges from u

Eulerian Paths on Directed Graphs II

- Let $\pi = \{(u = u_0, u_1), \ldots, (u_{K-1}, u_K = v)\}$, $v \neq u$, be an Eulerian path on G
- If $w \neq u$ is a node in π, each time we enter w we subtract 1 from $in(w)$ and also from $out(w)$ when we leave w
  - Since at the end we have passed through all the edges of w, **we must have at the beginning** $in(w) = out(w)$
- Similarly each time we enter u inside π we subtract 1 from $in(u)$ and also from $out(u)$ when we leave it
  - Moreover, when we start we subtract 1 from $out(u)$
  - We must thus have $out(u) = 1 + in(u)$ **at the starting node**
- Similarly, we must have $in(v) = 1 + ou(v)$ **at the ending node** Thus, we must also have $in(u) = out(u)$
- The above are in fact, **necessary and sufficient conditions to have an EP in a directed** G
- For circuits, i.e., $v = u$, these conditions are $in(w) = out(w)$ for all nodes

Eulerian Sequencing

- Essentially the same $O(|E|)$ algorithm we saw for undirected graphs can be applied to directed ones
- Thus we can efficiently sequence genomic reads
- **Example:** consider again `S = [TATGGTGC]` and `sp(S, 2)= {TA, AT, TG, GG, GT, GC}`
- Applying the Euler algorithm we obtain



$S_3 = \{GGT, TAT, TGG, TGC, ATG, GTG\}$
$S_2 = \{GG_1, GT_2, TA_3, AT_4, TG_5, GC_6\}$
L_2 a i
1 1 1 → 2
1 1 2 → 5
1 0 3 → 4
1 1 4 → 5
2 2 5 → 1 − 6
0 1 6
I
7
• hay camino euleriano de 3 a 6
π_1:
3 − 4 − 5 − 1 − 2 − 5 − 6
S=[TA T G G T G C]