Algorithms Preliminaries

ECS 289A – Lecture 3

What is an algorithm

- A procedure designed to perform a certain task, or solve a particular problem
- Algorithms are recipes: ordered lists of steps to follow in order to complete a task
- Abstract idea behind particular implementation in a computer program

Algorithms in Bioinformatics

Theoretical Computer Scientists are contributors to the genomic revolution

- Sequence comparison
- Genome Assembly
- Phylogenetic Trees
- Microarray design (SBH)
- Gene network inference

Algorithm Design

- Recognize the structure of a given problem:
  - Where does it come from?
  - What does it remind of?
  - How does it relate to established problems?
- Build on existing, efficient data structures and algorithms to solve the problem
- If the problem is difficult to solve efficiently, use approximative algorithms

Problems and Solutions

In algorithmic lingo:

- Problems are very specific, general mathematical tasks, that take variables as input and yield variables as output.
- Particularizations (assigning values to the variables) are called instances.
- Problem: Multiply(a,b): Given integers a and b, compute their product a*b.
- Instance: Multiply (13, 243).

Algorithms produce solutions for any given instance of a general problem

```
Multiply(a,b):
0) Let Product = 0
1) Take the k-th rightmost digit of b
and multiply a by it. Attach k-1 zeros
to the right, and add to Product.
2) Repeat Step 1. for all digits of b.
3) Product = a*b
```

Multiply (13, 243) = 3159

Algorithm Analysis

- **Correctness**
  - Exact solutions require a proof of correctness
  - Heuristics: approximate solutions
- **Resource Efficiency (complexity)**
  - **Time:** number of steps to follow to obtain a solution as a function of the input size
  - **Space:** amount of memory required for the algorithm execution
- **Best, Average, and Worst Case** Analysis

ECS289A

Time / Space Complexity

- **Input size**: how many units of constant size does it take to represent the input? This is dependent on the computational model, but can be thought of as the storage size of the input. The input size is usually n.
- Running time: f(n) = const., n, log n, Poly(n), e^n

ECS289A

Big Oh Notation

- Asymptotic upper bound on the number of steps an algorithm takes (in the worst case)
- f(n) = O(g(n)) iff there is a constant c such that for all large n, 0 <= f(n) <= c*g(n)
- More intuitively: f(n) is almost always less than or equal to g(n), i.e. algorithm with t.c. f(n) will almost never take more time than one with t.c. of g(n)

ECS289A

Big Oh, examples

- Const. = O(1)
- 3n = O(n)
- 3n = $O(n^2)$
- log n = O(n)
- Poly(n) = $O(e^n)$

- O(n) time algorithm is called linear
- O(Poly(n)) is polynomial
- $O(e^n)$ is polynomial

ECS289A

Basic Complexity Theory

- Classification of Problems based on the time/space complexity of their solutions
- **Class P:** Problems with polynomial time algorithms t.c. = O(Poly(n))
- **Class NP:** (non-deterministic polynomial) Problems whose solution instances can be verified in Poly(n) time.

ECS289A

Complexity, contd.

- **NP-complete** problems: a polynomial algorithm for one of them would mean all problems in NP are polynomial time
- But, **NO** polynomial time algorithms for NP problems are known
- P ≠ NP? Still unsolved, although strongly suspected true.
- NP complete problems: 3-SAT, Hamiltonian Cycle, Vertex Cover, Maximal Clique, etc. Thousands of NP-complete problems known
- Compendium:

http://www.nada.kth.se/~viggo/problemlist/compendium.html

ECS289A

Why All That?

- Many important problems in the real world tend to be NP-complete
- That means exact solutions are intractable, but for very small instances
- Proving a problem to be NP-complete is just a first step: a good algorist would use good and efficient heuristics

Popular Algorithms

- Sorting
- String Matching
- Graph Algorithms
  - Graph representation: linked lists, incidence matrix
  - Graph Traversal (Depth First and Breadth First)
  - Minimum Spanning Trees
  - Shortest Paths
- Linear Programming

Algorithmic Techniques

- Combinatorial Optimization Problems
  - Find min (max) of a given function under given constraints
- Greedy – best solution locally
- Dynamic Programming – best global solution, if the problem has a nice structure
- Simulated Annealing: if not much is known about the problem. Good general technique

Data Structures

- Once a given problem is digested, algorithm design becomes an engineering discipline: having a big toolbox and matching the tools to the task at hand
- A major part of the toolbox are <u>data structures:</u>

  <u>Data representations allowing efficient performance of basic operations</u>

Basic Opperations

- **Store/Search:**
  - Search(x)
  - Delete(x)
  - Insert(x)
- **Priority:**
  - FindMIN
  - FindMAX
- **Set:**
  - UnionSet
  - FindElement

Basic Data Structures

- Static: arrays and matrices
  - Array of n elements: a[i], 0 <= i <= n-1

1	2	3	4	5
a[1]	a[2]	a[3]	a[4]	a[5]

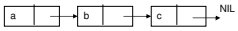
  - Matrix of n*n elements: m[i][j], 0 <= i, j <= n-1

	1	2	3	4
1	m[1][1]	m[1][2]	m[1][3]	m[1][4]
2	m[2][1]	m[2][2]	m[2][3]	m[2][3]
3	m[3][1]	m[3][2]	m[3][3]	m[3][4]

- Basic operations are O(1)

Dynamic Data Structures: linked lists, trees and balanced trees, hash tables

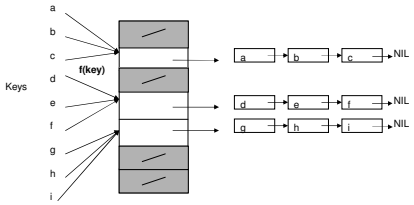
- No static memory allocation: items are added/deleted on the go
- Linked Lists (basic operations are O(n)):



- Trees

Balanced tree: Height is O(logn).
Basic operations are O(log n)

Hash Tables



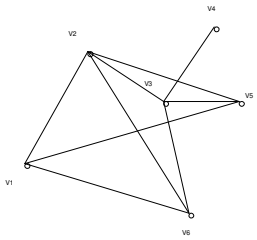
A good hash function f(key) yields constant search time O(1).

Set Data Structures

- Given sets A={1,2,3,4} and B={1,3}
- Operations: Find, Union
- Example:
  - Find(A,3) = yes
  - Find(A,5) = no
  - Find(B,3) = yes
  - Union(A,B) = {1,2,3,4}
- Very efficient: almost linear in the number of union+find operations

Graphs

- Graph G(V, E). V is a set of vertices, E a set of edges

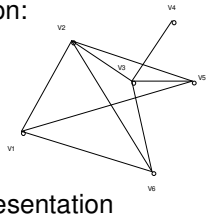


V = {v1, v2, v3, v4, v5, v6}

E = { (v1, v2), (v1, v5), (v1, v6),
(v2, v3), (v2, v5), (v2, v6),
(v3, v4), (v3, v5), (v3, v6) }

- Linked list representation:

v1: v2, v5, v6
v2: v1, v3, v5, v6
v3: v2, v4, v5, v6
v4: v3
v5: v1, v2, v3
v6: v1, v2, v3



- Adjacency Matrix Representation

	V1	V2	V3	V4	V5	V6
V1		1	0	0	1	1
V2	1		1	0	1	1
V3	0	1		1	1	1
V4	0	0	1		0	0
V5	1	1	1	0		0
V6	1	1	1	0	0	

A Greedy Clustering Example

- Clustering is a very important tool in analysis of large quantities of data
- Clustering: Given a number of objects we want to group them based on similarity
- Here we will work out a very simple example: clustering points in a plane by single-link hierarchical clustering

Clustering Points in the Plane

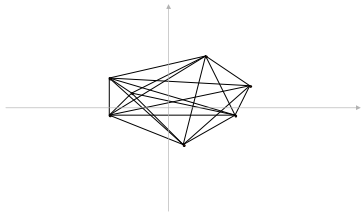
Problem 1: Given ***n*** points $p_1(x_1, y_1), p_2(x_2, y_2), \ldots, p_n(x_n, y_n)$ in a plane, cluster them so that if the distance between two points is less than D they are in the same cluster

Input: D, $p_1(x_1, y_1), p_2(x_2, y_2), \ldots, p_n(x_n, y_n)$

Output: Sets (clusters) of points C_1, C_2, …, C_k.

Algorithm Draft

- Calculate distances between point pairs



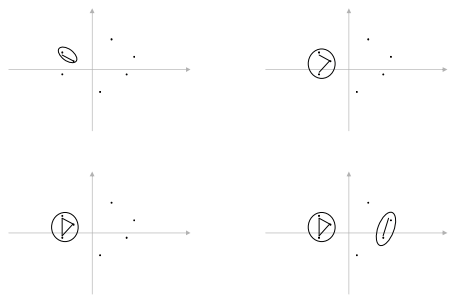
- Sort the distances in ascending order

p_2	p_1	$d_{2,1}$
p_3	p_2	$d_{3,2}$
p_3	p_1	$d_{3,1}$
…	…	…

Sort →

p_7	p_5	$d_{7,5}$
p_3	p_1	$d_{3,1}$
p_4	p_3	$d_{4,3}$
…	…	…

Move through the sorted list of distances and add a new point to a cluster if the distance is < D.



Algorithm in Detail

- Data Structure for the graph: adjacency matrix

p_2	p_1	$d_{2,1}$
p_3	p_2	$d_{3,2}$
p_3	p_1	$d_{3,1}$
…	…	…

- Data Structure for the clusters: Set (Union / Find)

Algorithm in detail

- Calculate distances $O(n^2)$
  - For all pairs i,j calculate d(i,j)
- Sort adjacency table $O(n^2 \log n)$
- Start with n sets, $p_1, p_2, \ldots, p_n$. Build a linked-list representation of a graph:
  - Get the next smallest distance, d(i,j)
  - If d(i,j) >= D done
  - Else Union(Find(p_i),Find(p_j))
- Traverse the graph to find the <u>connected components (DFS)</u>

ECS289A

Algorithm Analysis

- Correctness:
  - All distances less than D are added
  - Clusters contain all points with distance < D to some other point in the cluster
- Time complexity:
  - Bounded above by the sorting step
  - $O(n^2 \log n)$

ECS289A

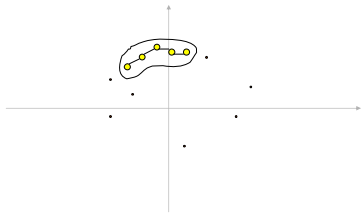
Discussion

- This algorithm is known as Single-Link Hierarchical Clustering
- It is a version of Kruskal's Minimum Spanning Tree Algorithm
- It is fast

ECS289A

Performance on Real Data

- Lousy: Chaining effects



Better Approaches

Problem 2: Given ***n*** points $p_1(x_1, y_1), p_2(x_2, y_2), \ldots, p_n(x_n, y_n)$ in a plane, cluster them so that the distance between any two points in a cluster is less than D

Input: D, $p_1(x_1, y_1), p_2(x_2, y_2), \ldots, p_n(x_n, y_n)$

Output: Sets (clusters) of points C_1, C_2, …, C_k.

- This problem is equivalent to Complete Link Hierarchical Clustering
- It can be shown to be NP-complete, with a direct reduction from the Clique problem
- Also, this clustering approach yields many small clusters
- We will talk about better clustering approaches in this class