Week 2: Convex Hull

- **Note: Due-date has been extended to Thursday 5pm**
- **Please submit on moodle; only the files convex_hull.py, and sorting.py.**
- This week, we will learn how to implement ADTs (Abstract Data Types) using Python. Then, we will implement a program to compute the convex hull of points on a plane. The algorithm for Convex Hull is known as the Graham Scan and has a time complexity of O(n log n) (it uses sorting as a sub-routine and has an additional complexity of O(n) on top).
- Please note that you are not allowed to look online for implementation / pseudo-code of the algorithm.

Algorithm

- The first step in this algorithm is to find the point with the lowest y-coordinate. If the lowest y-coordinate exists in more than one point in the set, the point with the lowest x-coordinate out of the candidates should be chosen. Call this point "P0".
- Next, the set of points must be sorted in increasing order of the angle they and the point "P0" make with the x-axis. Any general-purpose sorting algorithm is appropriate for this; we will use python's inbuilt sort routine for this (see convex_hull.py).
- The algorithm proceeds by considering each of the points in the sorted array in sequence. For each point, it is first determined whether traveling from the two points immediately preceding this point constitutes making a left turn or a right turn. If a right turn, the second-to-last point is not part of the convex hull, and lies 'inside' it. The same determination is then made for the set of the latest point and the two points that immediately precede the point found to have been inside the hull, and is repeated until a "left turn" set is encountered, at which point the algorithm moves on to the next point in the set of points in the sorted array minus any points that were found to be inside the hull; there is no need to consider these points again. (If at any stage the three points are collinear, we will discard points strictly within the line segment of the other two.
- This process will eventually return to the point at which it started, at which point the algorithm is completed and the stack now contains the points on theconvex hull in counterclockwise order.

Week 3 - Strassen's Matrix Multiplication

This week, we will implement matrix multiplication using the divide-and-conquer approach, due to Strassen. The algorithm is as described in the theory course. You may also refer here for a good overview of the algorithm. As always, you **may not** refer online for pseudo-code.

Instructions

Copy the files in ~rajsekar/courses/CS2810/week3/ to your directory and code your implementation in matrix.py. The file already contains a function called mult_mat that takes two arrays as input. The files are also available at here.

This assignment is due by the end of the class (4:40pm, Feb 1, 2016).

ADTs (Abstract Data Types) using Python. Then, we will implement a program to compute the convex hull of points on a plane. The algorithm for Convex Hull is known as the Graham Scan and has a time complexity of O(n log n) (it uses sorting as a sub-routine and has an additional complexity of O(n) on top).

Please note that you are not allowed to look online for implementation / pseudo-code of the algorithm.

Week 4: Shortest Path

Today's assignment will be to implement a min-heap, and use the data structure to implement Dijkstra's shortest path algorithm. The program should receive input in the following format.

Input / Output

- The first line of the input contains two integers, say n, m. n is the number of vertices in the graph, and m is the number of edges
- The next m lines, each, contains a list of 3 integers, u, v, d. This means that the edge connecting u and v has length / distance d.
- The next line is a number k, specifying the number of test cases for this graph. The next k lines each specifies two integers, u(i), v(i), 1 <= i <= k. For each such pair, the program should output the shortest distance between vertex u(i) and v(i) followed by a new line. In total, the output of the program is only k lines, each with a single integer: the shortest distance between the corresponding vertices. **Do not print anything else.**

Example Input:

5 5
1 2 1
2 3 1
3 5 1
1 4 1
4 5 1
2
1 5
4 5

Example Output:

2
1

Notes:

- Use "arr = map(int, raw_input().split())" to split a line of integers from the standard input into a python list. Thus, arr[0] above will contain the first integer in the list. If you know that the line contains 4 integers, then you may also say "a,b,c,d = map(int, raw_input().split())".

    - You may use the list representation of heap, and use integer comparison to heapify.
    - The time complexity of Dijkstra's algorithm is O(m log n) when using min-heaps, where m is the number of edges, and n is the number of vertices. Your implementation should also behave similarily.

Week 5 - Union Find

- This week, we will study the Disjoint-Set data structure and the Union-Find algorithm. The data structure is designed to maintain a collection of disjoint sets over a universe of elements. The key operations supported by the structure are: (1) union(A, B): combines the sets A and B to form A-union-B; (2) representative(x): returns the (unique) set S from the collection that contains x.
- The typical method to implement this structure is construct a data-type, called Node, for an element. A Node also contains an attribute 'parent' which is used as follows. Each node is initialized as a single-element set containing itself by setting the parent attribute to point to the element itself. Thus, initially the universe is regarded as a collection of (disjoint) single-element sets.
- When x needs to be added to another set Y, we simply set the parent of x to some element in Y. Now, each set Y contains one unique element whose parent is itself. To compute the union of X and Y, we compute the representatives (root elements) of X and Y and set one of them as the parent of the other. Now, every element in X U Y has a single new representative. Thus, to check if X and Y are distinct (disjoint) sets, we simply compare the root elements. These procedures are together termed Union-Find algorithms.

- As is, the procedures are very simple but not asymptotically fast. It is not too hard to construct a sequence of Union operation so that finding the root element of many of the elements takes O(n) time, where n is the number of elements in the universe. However, too simple heuristics together speedup the computation: (1) path compression: while finding the root of an element x, we also set the parent of x, along with all its predecessors to directly point to the actual root. (2) rank based union: while combining sets X and Y, we check the number of nodes contained in X and Y, and set the parent of the root of the smaller set to point to the root of Y. Together, these two heuristics improve the performance of the data structure. It is known (proven by Robert Tarjan in 1975) that for m operations, the improved ADT takes only time O(m \alpha(m)) where \alpha is the inverse-ackermann function (an extremely slow growing function).
- Today's assignment is to implement the two improvements to an implementation of the union-find ADT. As before, the source files are available at ~rajsekar/courses/CS2810/week5/ and also attached here. Upload the file union_find.py after implementing the two heuristics within.

Long Assignment - Graph Vizualization

- **Assignments to groups can be found here**
- The goal in this assignment is to build a framework to describe a graph algorithm. Towards this, each group (**upto 3 people**) should pick an algorithm involving a graph or a tree (spanning tree, shortest path, connectivity, heaps, search trees, etc.) and design an animated description of the algorithm along with annotation about the proof of its correctness.
- The animated description can be as rudimentary as a set of still views of the algorithm in action, or a fluid animation potraying the dynamics of the system. At the core should be a module that can display the current state of the algorithm. In particular, this would involve implementing a "dismantled" version of the algorithm.
- You can use graphviz library to display / save the slides. It is installed in ~rajsekar/sys/bin, so you can add the directory to PATH and try the software. Check out http://www.graphviz.org/Documentation.php for further references.
- Note that we encourage using open-source libraries if you can install it on the dcf machines (./configure; make; make install way). The larger problem to think about is in designing a framework to generate technical diagrams to describe the working of an algorithm. The design of your code should fit the larger picture.

March 14 - Hashing

- Hash tables are an important data-structure as they are quite simple, while at the same time very efficient in storing a collection of elements. In many cases, they outperform more sophisticated data-structures such as the search trees (AVL, RB, etc.).
- The core of a hash table is a hash function that maps a universe U to a much smaller range [p] = {0, ..., p-1}. The parameter p is usually chosen to match the expected size of the collection to store. The key idea is that if the hash-function is able to randomly / evenly distribute the universe of elements into the range, then insertion and deletion can simply be implemented as an array of array. Where the ith array stores all the elements in the collection whose hash value is i. Usually, the hash-function is picked at random to ensure this property (as otherwise, in theory, we could have a bad collection where all the elements map to the same element).
- In this assignment, we will work with a "pairwise-independent" hash family. The family has a parameter p and the set of functions are h(a, b): x -> ax + b (mod p). Here, a, b are picked at random from {0, ..., p-1} except a can't take 0. It is known (and not too hard to prove) that in this case, in expectation this is a pretty good hash-function.
- Implement a hash table based on this hash function. We also wish to follow software design paradigms and thus, you are provided with the interface of this data-structure (see htable.py, num_htable.py). All the related files can be found at ~rajsekar/courses/CS2810/Mar14/
- See below for the statistics of an example implementation. Note that the average time / op is in fact constant (independent of the size of the collection) and the max time / op is logarithmic (similar to AVL, RB trees).

March 21: Random Graph Models

- In this assignment, we generate random graphs and study their properties. The most commonly studied is the one proposed by Edgar Gilbert, denoted $G(n,p)$, in which every possible edge occurs independently with probability $0 < p < 1$. Another model we experiment is where we pick d random neighbors v_1, ..., v_d of a vertex u and add edges (u, v_1), (u, v_2), ..., (u, v_d).

- In each of these models, we wish to study:
    - the maximum degree
    - the diameter of the graph
    - connectivity
- Plot each of these quantities against the number of vertices (in log-scale if appropriate).

March 28: Search Trees

- In this assignment, we implement a search tree data-structure and measure its performance via simulation. The task may be broken into the following modules:
    - Trees:
    -
    - This module should provide a Tree abstraction. Design an interface that facilitates its use as a search tree while also keeping a count of the total operations made. This may be done by passing a counter object (from a Counter class that allows to increment, reset, and read the count) to the constructor of the Tree class.
    - Search Trees:
    - You may pick either AVL or RB trees to implement. The interface needs to simply allow one to insert, delete and query for an element. Assume that the elements may be compared
    - In fact, in standard libraries that implement data-structures for totally-ordered set, only the less-than operator is assumed to be defined. Everything else is derived from this (eg. a == b iff !(a < b) and !(b < a)).
    - Simulation:
    - To simulate, pick a number n and simulate m = O(n) random insertions and deletions. Since we expect the average time per op. to be log(n), pick n as a geometric series, floor(5 * 1.2^k) for k = 1, 2, … . You should initialize a counter per simulation and plot the average value against log(n).
- Submit your source directory as a zip file. You may zip a file using tar and gzip by running the command: tar cfz <zipfilename.tar.gz> <directory>/*.py . You should be in the parent directory when running this command.

April 4: String Searching Algorithms

- A string matching algorithm looks for a string W inside another (larger) string S. The simplest algorithm here simply starts at each index i less than the length of S and checks if W is in the string starting at the index i. The complexity of this algorithm is O(n.k) where n = |S|, and k = |W|.
- The Knuth-Morris-Pratt algorithm is a sophisticated version of this that runs in time O(n + k) and proceeds by first pre-processing the search string W into a finite-state automaton F(W). Now, the string S is fed to this automaton in order to check if W exists in S. This is a very relevant algorithm when searching for a large string inside another string (eg. a large DNA sub-sequence within the human genome).
- This week, we will study and implement the KMP algorithm for string matching.

April 18: AVL / RB Trees

- Extend the binary search trees from a previous class to implement rotation and balancing operations. Plot the same statistics. Extra question: can you find a statistic on random data which is w(log n) with usual BSTs but is O(log n) for AVL trees?