COMP20007 Design of Algorithms

Week 7 Workshop

Tutorial

1. Negative edge weights Dijkstra's algorithm, unmodified, can't handle some graphs with negative edge weights. Your friend has come up with a modified algorithm for finding shortest paths in a graph with negative edge weights:

1. Find the largest negative edge weight, call this weight $-w$.
2. Add w to the weight of all edges in the graph. Now, all edges have non-negative weights.
3. Run Dijkstra's algorithm on the resulting non-negative-edge-weighted graph.
4. For each path found by Dijkstra's algorithm, compute its true cost by subtracting w from the weight of each of its edges.

Will your friend's algorithm work?

2. Master Theorem The Master Theorem states that if we have a recurrence relation $T(n)$ such that

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^d),$$
$$T(1) = c,$$

then,

$$T(n) \in \begin{cases} \Theta\left(n^d\right) & \text{if } a < b^d \\ \Theta\left(n^d \log n\right) & \text{if } a = b^d \\ \Theta\left(n^{\log_b(a)}\right) & \text{if } a > b^d \end{cases}.$$

Note: a similar formula holds for O and Ω as well.

Apply the Master Theorem to find the time complexities of the following recurrence relations (in Big-Theta terms).

(a) $T(n) = 9T\left(\frac{n}{3}\right) + n^3,\ T(1) = 1$

(b) $T(n) = 64T\left(\frac{n}{4}\right) + n + \log n,\ T(1) = 1$

(c) $T(n) = 2T\left(\frac{n}{2}\right) + n,\ T(1) = 1$

(d) $T(n) = 2T\left(\frac{n}{2}\right),\ T(1) = 1$

3. Mergesort Time Complexity Mergesort is a divide-and-conquer sorting algorithm made up of three steps (in the recursive case):

1. Sort the left half of the input (using mergesort)
2. Sort the right half of the input (using mergesort)
3. Merge the two halves together (using a merge operation)

Construct a recurrence relation to describe the runtime of mergesort sorting n elements. Explain where each term in the recurrence relation comes from.

Use the Master Theorem to find the time complexity of Mergesort in Big-Theta terms.

4. Lower bound for the Closest Pairs problem The closest pairs problem takes n points in the plane and computes the Euclidean distance between the closest pair of points.

The algorithm provided in lectures to solve the closest pairs problem applies the divide and conquer strategy and has a time complexity of $O(n \log n)$.

The *element distinction problem* takes as input a collection of n elements and determines whether or not all elements are distinct. It has been proved that if we disallow the usage of a hash table[1] then this problem cannot be solved in less than $n \log n$ time (*i.e.*, this class of problems is $\Omega(n \log n)$).

Describe how we could use the closest pair algorithm from class to solve the element distinction problem (where the input is a collection of floating point numbers), and hence explain why this proves that the closest pair problem must not be able to be solved in less that $n \log n$ time (and is thus $\Omega(n \log n)$).

[1] Excluding the use of a hash table is done to conform to the *algebraic decision tree* model of computation, where we cannot use the values of the elements to index into memory, and only allows us to compute and compare the elements themselves (and simple functions of these elements).

Computer Lab

For today's computer lab we've provided a graph module, a list module and a naive priority queue module (later in the semester we'll use a min-heap to re-implement this module).

Your task will be to implement a number of graph algorithms using the modules provided.

We have already implemented a DFS in the `graphalgs` module for you as an example.

You should read through the provided code and understand the example provided. Then you should implement the following graph algorithms in the graph algorithms package.

We've provided a number of input graphs for you to test your algorithm with.

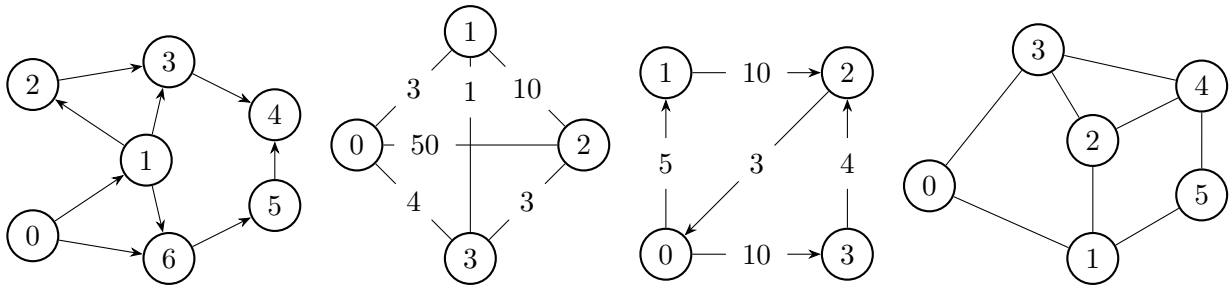

Figure: The graphs represented in `graph-01.txt`, `graph-02.txt`, `graph-03.txt` and `graph-04.txt`.

1. BFS We've written a `dfs` function which returns an array with the order in which nodes are explored using a depth-first search.

Write your own algorithm with the same behaviour, except this time for a **breadth-first search**. You may find using an iterative aproach easier for this task.

Your algorithm should have the signature `int *bfs(Graph *graph);`

2. Dijkstra's Algorithm Write a function which performs Dijkstra's algorithm on the input graph. Remember that Dijkstra's algorithm is a *single source shortest paths* (SSSP) algorithm which finds the shortest paths from a given start node to every other node.

Your function should take a pointer to an array `dist` which has length n and store the distances from the start node to each other node, or -1 if there is no path from the start node to that node.

Your function signature should be `void dijkstras(Graph *graph, int start, int *dist);`

(*Challenge*) Modify your function to store the paths from the start node to each other node in an array of linked lists or an array of arrays. Which data struture will be easier to work with?

3. Prim's Algorithm (Optional) Write a function which performs Prim's algorithm on the input graph. Remember that Prim's algorithm computes the *minimum spanning tree* for a connected undirected graph.

Your function should take in pointers to two arrays `from` and `to` which have length $n-1$ and fill them so that the edges that make up the minimum spanning tree are:

(`from[0]`, `to[0]`), ..., (`from[n-1]`, `to[n-1]`)

Your function should return whether or not the algorithm was succesful.

The signature should be `bool prims(Graph *graph, int *from, int *to);`