SOFTENG 250 Summary Notes (LN 1-10)

Edward Zhang

June 21, 2017

Gale-Shapely Algorithm - Stable Matching Problem **LN1**

- Using the company - applicants matching example, the company ranks all applicants in order of preference and the applicants do the same for companies.
- Initially all companies and applicants are free.

```
while there is a free company (c)
        set x to be c's highest ranked applicant to whom c hasn't made an offer yet
        if x is free then (c, x) becomes a job pair
        else this means x is already in an job pair with c2
                if x prefers c2 to c then c remains free
                else x prefers c to c2 then (c, x) becomes a job pair and c2 becomes free
        endif
endwhile
return the set of pairs we generated
```

- Due to the dynamically changing size of eg. free companies, a (doubly) linked list is appropriate.
- Running time is linear to the size of the input (where the size of the input is number of companies $\times$ number of applicants).

MergeSort - Array Sorting **LN2**

- The Merge algorithm combines two sorted lists in one pass. It keeps a pointer on each list and appends the smaller of the two onto the output list, then advances that pointer.
- MergeSort recursively splits the input array in half until size ≤ 1 then recursively combines them using Merge, runs in $O(n \log(n))$ time.

FindMin - Finds minimum value of array **LN2**

- Trivial. Makes one pass through the array while storing the current minimum. Runs in linear time.

Closest Pair (Brute Force) - Finds the two closest points in space **LN2**

- Loop through all points. For each point, loop through all points except itself and compute the distance between them. If this is lower than the currently stored minimum distance, store these points as the new closest pair.
- Runs in $O(n^2)$ time, as there is a nested forloop, each of size n.

Disjoint Subset - Given a set of subsets, Find two subsets with 0 overlap **LN2**

- Loop through all subsets. For each subset, loop through all other subsets. For each point in the first subset, check if it belongs in the second subset. If no points belong in the second subset then output these two subsets as disjoint.
- If each set has a max size of n, the inner loop makes n iterations. The loop for the second subsets loops n times. The loop for the outer subsets loops n times too. Hence $O(n^3)$.

Euclidean Algorithm - Find Greatest Common Denominator **LN2**

- First we divide the larger number by the smaller one and find the remainder. Then we divide the smaller number by the remainder then the old remainder by the new remainder until the remainder $= 0$.
- Let b be the larger of the two numbers, and c the smaller one.

```
while remainder > 0
        find r = x%c (remainder/mod)
        set x = c and c = r
endwhile
return c
```

Efficiency and Running Time — LN3

"An algorithm is efficient if its running time
is bounded by a polynomial."

$O(a)$ represents a asymptotic upper bound, whereas $\Omega(b)$ represents a asymptotic lower bound. $\Theta(c)$ represents a asymptotically tight bound. This is true if $a = b = c$.

Log functions are growing functions with a *very slow growth rate.* This function grows much slower than any polynomial. We can see the slow growth rate if we calculate the rise/run. $\frac{\log_b(n)}{n}$ as n approaches infinity is zero, hence the steepness approaches zero. For running time analysis, the base of the log is not considered.

Polynomial bounds are of the form $a_0 + a_1n + a_2n^2 + ... + a_nn^k$. The tight bound for this is $\Theta(n^k)$. Running times can also be bounded by functions like n^x where $x > 0$ is *not necessarily a integer.* For instance, you can have an algorithm with a running time bounded by $n^{0.001}$ or $n^{0.57}$. Note that logarithms like $n\log(n)$ are still polynomial since they're upper bounded by $O(n^2)$. Further examples of polynomial bounds (from the test) include;

- $O(n)$, since this is essentially $O(n^1)$
- $O(\sqrt{n^3})$, as this is $O(n^{\frac{1}{3}})$
- $\Theta(n^2 \times \log^3(n))$, as n^2 is the dominant value
- $O(n^4 + n^3.5 + \log^2(n))$, still bound by n^4
- $O(1)$, basically $O(n^0)$

An exponential function is in the form of r^n, where $r > 1$. Just as log functions are upper bounded by polynomials, polynomials are upper bounded by exponentials: $n^x = O(r^n)$. However unlike Logarithmic functions, we cannot ignore the size of r.

Some algorithms run in sub-linear time. How can the running time be smaller than the time required to read the input of size n? Sub-linear time algorithms *do not read the entire input.* A good example is Binary Search, below.

Binary Search - Find if x is in a sorted array — LN3

- Brute force completes this in linear time. However Binary Search is faster, given a sorted array.
- Find the mid point. If it's x, we're done. Otherwise, compare the midpoint to x to determine whether x is in the upper half or lower half of the array. This halves the search space. We now find the mid point again etc. etc. until x is found or if we run out of entries to check.
- At each step, the search space is halved. Therefore after k steps the search space is 0.5^kn. Therefore, the number of times we halve the segments is $\leq \log(n)$, thus the running time is $O(\log(n))$, which is sublinear time.

Graph Definitions — LN4

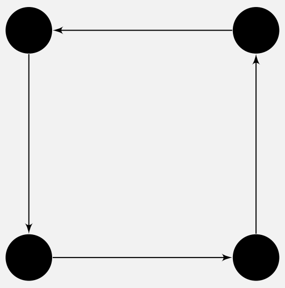
- *Undirected Graphs* are also just simply known as *graphs.* They are the most basic type of graphs and consist of points *(vertices)* and lines *(edges).* Edges that connect a vertex to itself are disallowed. Multiple edges between any two vertices are also disallowed.
- *Vertices* are points on the graph to which edges connect. The set of vertices of a graph are written like $V = \{0, 1, 2, 3, ..., 9\}$
- *Edges* are the lines that connect multiple vertices. For example a series of edges between vertices 0 and 1, 0 and 3 and 2 and 3 can be written as $E = \{\{0, 1\}\{0, 3\}\{2, 3\}\}$
- *Degrees* are the number of edges connected to a vertex; so a vertex with 4 edges connect to it would have a degree of 4.

- *Digraphs (Directed Graphs)* still consists of points & edges, however the edges have specific directions (ie. they're arrows). For example a series of edges from vertices 0 to 1, 0 to 3 and 2 to 3 can be written as $E = \{(0,1)(0,3)(2,3)\}$
- *Complete* graphs are graphs where every vertex is directly connected to every other vertex.
- *A path* is a sequence of vertices such that $\{v_0, v_1\}, \{v_1, v_2\}, \{v_{n-1}, v_n\}$ etc., thus forming a *path* from 0 to n. n is regarded as the length of this path. Paths are reversible (ie. starting at n and ending at 0) (except for in digraphs).
- *Connectivity*: Vertices u and v are connected if there is a path from u to v. Note every vertex is connected to itself. A *component* of vertex v is the collection of all vertices that are connected to v. A entire graph G is connected if all pairs of u and v are connected.
- *Strong Connectivity*: Where vertex u has a path to v and v also has a path back to u. Only relevant for digraphs.
- *Cycles* are paths where no vertices are repeated except the start and finish being the same vertex.
- *Trees* are graphs that are connected and contain no cycles. We call vertices in a tree nodes.
- Rooting a tree is when we select a vertex in a tree and "pull it up" and let the rest of the vertices "dangle down". This vertex is the root of the tree and we move down from the root to the leaves. The parent of a node v is u where we encounter u first then v on it's path from r.

Graph Arrangements **LN4**

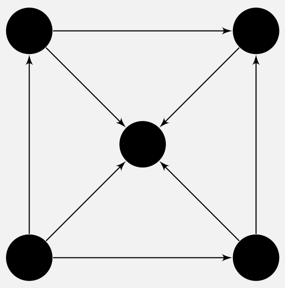
Note that these examples are directed, however undirected versions exist too.

Cycles



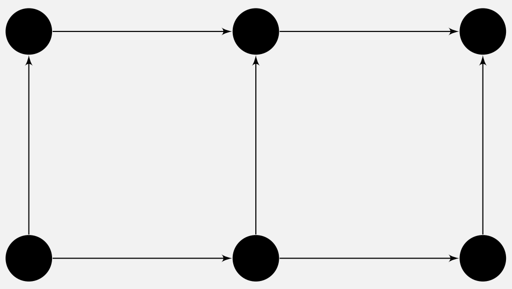
A cycle of length n has n vertices and n edges, arranged as $(0,1)(1,2),(2,3),...,(n-1,0)$.

Wheels

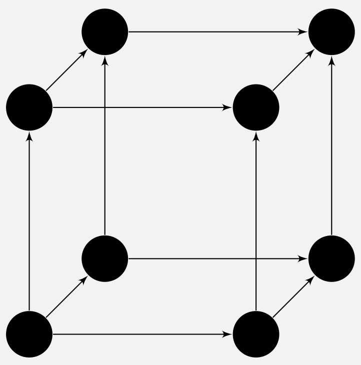


A wheel of size n (where $n > 2$) has n vertices and $2n$ edges, arranged as $(1,2),(2,3),...,(n-1,1)$ and $(1,0),(2,0),(n,0)$ (0 is in the center)

Grids



Cubes



BFS algorithm - Construct the component of s **LN5**

- We start at s and we define L_0 (Layer 0) as just s. We then proceed to store all vertices adjacent to members of L_0 (ie. vertices adjacent to s) into L_1. Then we store all vertices adjacent to all members of L_1 into L_2 and so on and so forth.
- We can implement BFS using the Queue data structure (FIFO). We pop the oldest vertex from the queue and find all of it's adjacent vertices. We then put these vertices into the back of the queue. We will need a separate array to store

vertices that we've already explored to avoid duplicates.

- Generates a BFS-Tree, runs in $O(n+m)$ time, where n is the number of vertices and m is the number of edges.

DFS algorithm - Construct the component of s **LN5**

- We start at s and we "walk" along the edges until we reach a dead end. From there we backtrack until we find a vertex that has unexplored edges and we then "walk" along those edges, and son on and so forth.
- We can implement BFS using the Stack data structure (LIFO). We pop a vertex from the Stack and replace it with all of it's adjacent vertices. Now we pop one of these vertices and put all of it's adjacent vertices etc. We will need a separate array to store vertices that we've already explored to avoid duleplicates.
- Generates a DFS-Tree, also runs in $O(n+m)$ time.

Strong Connectivity - Check if digraph G is strongly connected **LN6**

- Run BFS on G, then reverse all vertices in G and construct BFS again. If both BFS outcomes are identical, then G is strongly connected.

DAGs and Topological Order **LN6**

Consider a set of tasks that represent the set of courses needed to complete a SE degree with dependency relations between these tasks (ie. We must complete course a before doing b): This can be represented by a directed acyclic graph (or DAG). There have a different structure from acyclic undirected graphs (which are just trees). A topological order would represent an order the courses can be completed in without creating any conflicts (ie. once we get to course c in the order all of it's prerequisites have already been completed earlier on in the order).

In-degrees and out-degrees are like degrees for undirected graphs except in-degrees represent edges going into a vertex and out-degrees represent edges coming out of a vertex.

Topological Order - Finds the topological order of a DAG G **LN6**

- By definition a DAG must have at least one vertex with an in-degree of zero (otherwise there will be a cycle)
- While G is not empty, find a vertex with an in-degree of zero and append it to the output, then delete this vertex from G. Now find the next vertex with an in-degree of zero etc.
- Using a basic implementation, running time is n^2 as it takes n time to *find* and delete a vertex from the graph and the while loop makes n iterations.
- However, if when a vertex v is removed we decrease the in-degree of those it is adjacent to while simultaneously checking to see of the in-degree of these have decreased to 0, we can achieve a better running time since we don't have to check n every iteration, instead just every outbound edge of v. This means that each edge is only checked once, and thus the running time is $O(n+m)$ with the implementation.

Greedy Algorithms & Scheduling Problems **LN7**

Greedy Algorithms build the solution step by step and are defined as;

"An algorithm that makes the locally optimal choice at each stage
in order to find the global optimum"

We define requests in a scheduling problem in the following form;

"We would like to use your resource for the period
starting at time s and finishing at time f."

- Interval Scheduling: Given a set of n requests, compute the largest size subset of compatible requests (compatible = non-overlapping), ie. fulfil as many requests as possible without overlap.
- Interval Colouring: Given a set of n requests, fulfil all requests while allocating as few resources as possible (where each resource can only fulfil one request concurrently.)

- Minimising Lateness: Requests instead state a length and a deadline. We only have one resource and must satisfy all requests - find a sequence of requests and their scheduled intervals that minimises lateness (finishing time past deadline).

Earliest Finishing Time - Interval Scheduling **LN7**

- A very simple algorithm based on the selection rule of "earliest finishing time"
- While the input is not empty, append the request with the earliest finishing time to the output, then delete said request and all requests that overlap it from the input. Now pick the request with the earliest finishing time again etc.
- Works in a single pass provided the requests are sorted by finishing time. Sorting the requests takes $O(n \log n)$ and thus this algorithm is $O(n \log n)$.

Greedy Colouring - Interval Colouring **LN7**

- Sort all request intervals by their start times: $I_1, I_2, I_3, ..., I_n$
- Assign I_1 resource 1. Advance to I_2. Check all existing resources to see if they are free (in this case only resouce 1) - assign the first free resource, or, if there aren't any, allocate a new resource - resource 2. Advance to I_3. Check all resources (1 and 2) and see if they are free... etc.
- We make n iterations as we visit each request. At each iteration we check every previous resource to see if they overlap. Therefore running time is $O(n^2)$.

Earliest Deadline First - Minimising Lateness **LN8**

- Another greedy algorithm - use the selection rule of "earliest deadline".
- First sort requests by deadline. For each request assign a start time equal to the finishing time of the previous request and a finishing time of start time + length.
- This algorithm abides by two properties - if r_1 has an earlier deadline than r_2, then r_1 is scheduled earlier than r_2 and the schedule has no idle time. Note however that an optimal schedule doesn't necessarily have these two properties.
- Works in a single pass provided the requests are sorted by deadline. Sorting the requests takes $O(n \log n)$ and thus this algorithm is $O(n \log n)$.

Optimal Caching **LN8**

Assume we have data stored in fast memory C and also in the main, but much slower memory M. The storage of C is much smaller than M. Therefore C acts as a *cache* for M.

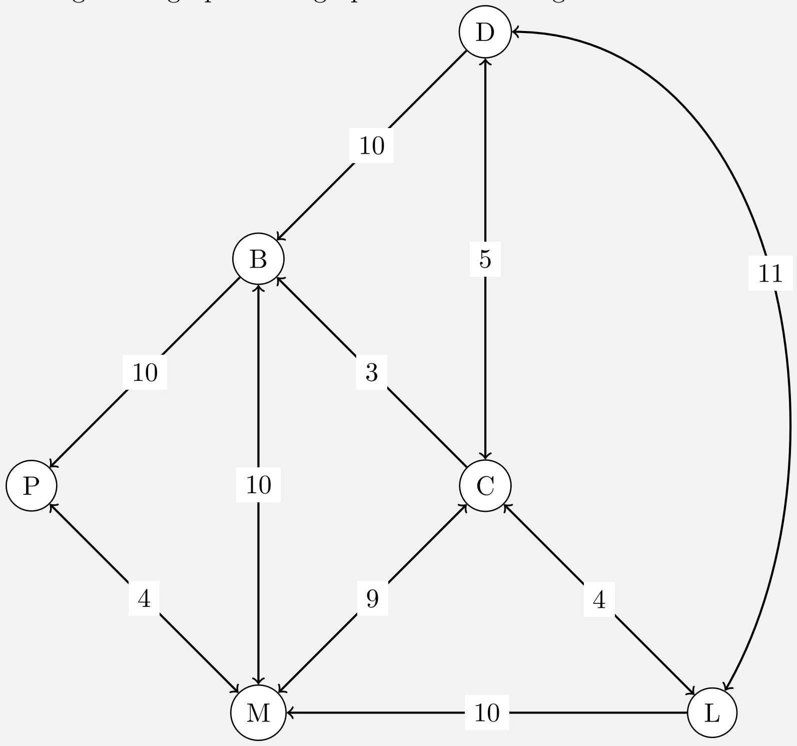
We want caching to be as effective as possible: when we want to access a piece of data, the data should already be in the cache as often as possible. If we have to retrieve data from the main memory then this is a cache miss. We want to minimise cache misses.

Farthest in Future - Cache Maintenance **LN8**

- Assume a full cache to begin with
- When we need to access item d that isn't already in the cache, evict the item in the cache that is required the farthest into the future and insert d.
- This algorithm has the property that it only evicts an item from the cache when necessary (ie. it produces a *reduced schedule*). It is ok to make an unnecessary eviction earlier and still have an optimal solution, however the number of cache misses is unaffected (and hence farthest in future still produces an optimal solution).

Weighted Digraphs **LN9**

A weighted digraph is a digraph where each edge has an associated weight. An example of a weighted digraph;



- We always assume the weights of all edges are non-negative
- The *weight* of a path is the sum of the weights of it's edges
- The *shortest path-distance* from u to v is the minimum weight path from u to v, denoted by $\delta(u, v)$
- Given a path P, a *subpath* is any continuous segment of P.
- The triangle equality states that $\delta(x, z)$ must $\leq \delta(x, y) + \delta(y, z)$
- The *shortest path problem* involves finding the shortest path from source vertex s to *all other vertices.*

Dijkstra's Algorithm - Shortest Path Problem **LN9**

- The algorithm computes an path-distance estimate to all vertices, denoted $d(v)$.
- A set F of vertices is maintained which stores the shortest path-distances that have already been computed.
- Each iteration the algorithm picks the vertex not in F with the smallest path-distance estimate v and puts it in F - the intention is that this must now be the shortest path distance.
- Once v is added to F, all estimates not in F that are adjacent to v are updated. This is called *relaxation.*
- Let G be the input graph and s be the starting vertex.

```
set d(s) = 0 and d(v) = ∞ where v is every other vertex, F = {s} and R = {every vertex that's not s}
        foreach v that has an edge outgoing from s, set d(v) to the weight of the edge between s and v
        while R contains a vertex with a distance < ∞
                select the vertex v in R with the lowest distance
                add v to F
                Relaxation: Update every vertex connected to a edge outgoing from v with new distance
                delete v from R
        endwhile
```

- In order to construct the shortest path from s to u, we find the edge (a, u) where a is in F and is responsible for u being put in F. We then do the same for a until we end up at s.
- Dijkstra's runs in $O(e \log(v))$ time (where v is the number of vertices and e is the number of edges.) when using a heap data structure (see below)..

Priority Queue **LN9**

A priority queue data structure maintains a set S of elements, where each v in S has a key - no two keys represent the same element but two elements may have the same key. The lower the value of the key, the higher the priority. The priority queue supports 3 operations - insert (add element), delete (remove element) and select (returns the element of the highest priority (ie. lowest key)).

Heap **LN9**

Priority Queue data structures can be implemented using a heap, allowing for all operations to be completed in $O(\log n)$ time. A heap is a balanced binary (ie. 2 children nodes per parent) tree. Heaps are defined by the property that parents of any elements must have a key of equal or smaller value than their children. (ie. lowest keys at the top and largest at the bottom) This is called Heap order. We do insertion and deletion in a heap using HeapifyUp and HeapifyDown respectively.

Heapify Up - Heap Insertion **LN9**

- Before running Heapify Up we insert the new entry into a leaf (ie. at the bottom of) of the existing tree.
- If the parent of the new entry has a key value lower than the new entry, do nothing
- If the parent has a higher key, swap the parent and the new entry. Then, run heapify up again on the entry and it's new parent.
- Since the height of the tree is $\log n$, the running time of the insert operation is $O(\log n)$.

Heapify Down - Heap Deletion **LN9**

- Before running Heapify Down we insert the entry at the end of the tree into the "hole" left by the removed entry. If the key is too small for it's position just run Heapify Up. Otherwise, we need to run heapify Down.
- Intuitive. Check the new entry and it's two children and perform (or don't do anything) such that the entry out of the 3 with the smallest key is at "top".
- Need to descend at most $\log n$, so deletion is at most $O(\log n)$.

Minimum Spanning Tree **LN10**

Problem is intuitive: Given a weighted, undirected graph, find a tree that spans all vertices (ie. allows you to access all vertices) with the lowest possible total weight. A real life example would be if you were building a communications network between n locations and you want to make sure they can all communicate to each other for the lowest cost of constructing the network.

Prim - Minimum Spanning Tree **LN10**

- The assumption is made that no two edges have the same weight
- Start with a subset V that only contains the starting vertex, s.
- Find an edge e with the smallest weight such that one side of it is in V and the other side is not.
- Add the vertex at the other end of the edge to V and e to E (E stores accepted edges)
- Running time is similar to Dijkstras, $e \log(v)$ ... why?

Kruskal's - Minimum Spanning Tree **LN10**

- Start with just the vertices (and no edges) and add edges back one by one, in ascending order of weight. Skip edges that cause cycles.
- Running time is the same as Dijkstras and Prim, $e \log(v)$.