Final Exam Revision

Design of Algorithms [COMP20007]
Semester 1, 2016

Data Structure Operations

Big-O of each data structure (type): (average) (worst)

- Array
    - Access: O(1), O(1)
    - Search: O(n), O(n)
    - Insert: O(n), O(n)
    - Delete O(n), O(n)

- Linked-list
    - Access: O(n), O(n)
    - Search: O(n), O(n)
    - Insert:
        - Append - O(n), O(n)
        - Prepend - O(1), O(1)
    - Delete:
        - Front - O(1), O(1)
        - Back - O(n), O(n)

- Hash Table
    - Access - can't be accessed…
    - Search: O(1), O(n)
    - Insert: O(1), O(n)
    - Delete O(1), O(n)

Divide and Conquer

- Multi-module program
- Solve each sub-problems via each sub-program
- Source -> Assembler -> Object

- Then many objects linked into one big program

Big-O Notation

Master's Theorem

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

- $log_b(a)$ in relationship to d
  - if $log_b(a) > d$ then $T(n) = O(n^{log_b a})$
  - if $log_b(a) = d$ then $T(n) = O(n^d log(n))$
  - if $log_b(a) < d$ then $T(n) = O(n^d)$

Recurrence

- Finding big-O notation through the algorithm's recurrence
- Identify Master's Theorem problems constants:
  - a is the number of sub-problems
  - $\frac{n}{b}$ is the size of sub-problems
  - n^d is the steps of operations of each recurrence
- After that then refer to Master's Theorem to calculate the big-O notation

***** State your assumption when doing the recurrence**

Graph

Formal Definition: graph is a set of vertices and edges

$$G = (V, E)$$

- V = a set of vertices
- E = a set of edges

Strongly Connected, Contected and Complete Graph

- Strongly connected is usually associated with directed graphs (one way edges):
  - There is a route between every two nodes.
- Connected is usually associated with undirected graphs (two way edges):
  - There is a path between every two nodes.
- Complete graphs are undirected graph where there is an edge between every pair of nodes.

Applications

1. Traffic flow
2. Finding the shortest path
3. Maze-solving

Topological Sort

Topological ordering of a directed graph is a linear ordering of its vertices such that for every directed `edge uv` from `vertex u` to `vertex v`, u comes before v in the ordering.

Pre and post number to find topological sorted list. From

Kahn's Algorithm (Kahn's Sort)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges
while S is non-empty do
    remove a node n from S
    add n to tail of L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    return error (graph has at least one cycle)
else
    return L (topologically sorted order)
```

If the graph is a DAG, a solution will be contained in the list L (the solution is not necessarily unique). Otherwise, the graph must have at least one cycle and therefore a topological sorting is impossible.

Since Kahn's algorithm is based on *breath-first search*, the data stucture used here is a *stack*.

Recursive DFS Algorithm

```
L ← Empty list that will contain the sorted nodes
while there are unmarked nodes do
    select an unmarked node n
    visit(n)


function visit(node n)
    if n has a temporary mark then stop (not a DAG)
    if n is not marked (i.e. has not been visited yet) then
```

```
        mark n temporarily
        for each node m with an edge from n to m do
            visit(m)
        mark n permanently
        add n to head of L
```

On each visit, each node gets appended to list L. When the same position is visited (marked) then the algorithm knows that the graph is cyclic, it terminates.

- Both of the algorithms have a linear time complexity $O(|V| + |E|)$

Depth-first Search algorithms

- Basic algorithms
- Search through the depth of the graph
- Adaptation to compute cyclicity, contectedness and colouring
- DFS Transversal
  - Pre and post numbers
  - Checking for cyclicity
  - Connectivity
  - Strongly Connected Components (SCC)
    - Sources and Sinks of the graph

Kosaraju’s Algorithm

Finding Strongly Connecetd Components through transposed graph

If strong components are to be represented by appointing a separate root vertex for each component, and assigning to each vertex the root vertex of its component, then Kosaraju’s algorithm can be stated as follows.

1. For each vertex u of the graph, mark u as unvisited. Let L be empty.
2. For each vertex u of the graph do Visit(u), where Visit(u) is the recursive subroutine:
   - If u is unvisited then:
     - Mark u as visited.
     - For each out-neighbour v of u, do Visit(v).
     - Prepend u to L.
   - Otherwise do nothing.
3. For each element u of L in order, do Assign(u,u) where Assign(u,root) is the recursive subroutine:
   - If u has not been assigned to a component then:
     - Assign u as belonging to the component whose root is root.
     - For each in-neighbour v of u, do Assign(v,root).
   - Otherwise do nothing.

Kosaraju with DFS

1. Perform a DFS of G and number the vertices in order of completion of the recursive calls.
2. Construct a new directed graph Gr by reversing the direction of every arc in G.
3. Perform a DFS on Gr starting the search from the highest numbered vertex according to the numbering assigned at step 1. If the DFS does not reach all vertices, start the next DFS from the highest numbered remaining vertex.
4. Each tree in the resulting spanning forest is a strong component of G.

Distance in graphs

- Breath-first Search algorithm
- Definition of path
- Dijkstra's Algorithm
  - Find the shortest node from the origin and add into paths
  - Priority Queue implementation

Dijkstar Algorithm

Finding the shortest path within a graph.

```
initialize Q
foreach node v:
    add v to Q with priority ∞
update priority of source node s in Q to 0
while Q not empty:
    v = element of Q with lowest priority
    remove v from Q
    foreach edge (v, u):
        dist(u) = min(dist(v) + wvu, dist(v))
        update priority of u in Q
```

Djikstar Algorithm is a greedy algorithm which heavily depends on priority queue: the shortest path from the origin goes first. Due to this fact, Djikstar complexity rely on the queue implementation.

Priority queue implementation:

- Array
- Heap
- List

Bellman-Ford Algorithm

Slower than Dijkstra’s but capable of handling negative edges

So, the idea behind Bellman-Ford Algorithm is if we run the algorithms through the graph repeatedly, the distance from a vertex to another won’t be changed if there is no negative cycle on the graph.

The algorithm is different from Dijkstra’s as Djikstra “greedily' choose the shortest path from the starting point and update the distance, Bellman-Ford update the distance on all of the edges.

```
function BellmanFord(list vertices, list edges, vertex source)
    ::distance[],predecessor[]

    // This implementation takes in a graph, represented as
    // lists of vertices and edges, and fills two arrays
    // (distance and predecessor) with shortest-path
    // (less cost/distance/metric) information

    // Step 1: initialize graph
    for each vertex v in vertices:

        // At the beginning , all vertices have a weight of infinity
        distance[v] := inf
        predecessor[v] := null  // And a null predecessor

    // Except for the Source, where the Weight is zero
    distance[source] := 0

    // Step 2: relax edges repeatedly
    for i from 1 to size(vertices)-1:
        for each edge (u, v) with weight w in edges:
            if distance[u] + w < distance[v]:
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            error "Graph contains a negative-weight cycle"
    return distance[], predecessor[]
```

Minimum Spanning Tree

A minimum spanning tree is a spanning tree of a connected, undirected graph. It connects all the vertices together with the minimal total weighting for its edges

Prim’s Algorithm

Prim's Algorithm have simiar approach to Dijkstra's algorithm. The two are different by the fact that Dijkstra choose the vertex that is the *closest to the known region* while Prim choose the one that is the *closest to the current vertex*.

1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
2. Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
3. Repeat step 2 (until all vertices are in the tree).

Kruskal's Algorithm

Kruskal's Algorithm of finding a minimal spanning tree is based on *disjointed set* data structure. Kruskal approach is very similar to Prim's but Kruskal use a `disjointed set` data structure to implement his algorithms.

1. Create a set S containing all the edges in the graph
2. Remove an edge with minimun weight from S
3. If the removed edge connect to two different set, add it into a set F
4. Repeat steps 2-3 (untll all S is empty)

Finding K-th Smallest element

Given the array is sorted, through the method of pivoting the array into 3 parts

Let the kth-smallest item is `d`

```
Choose a pivot p and split into L
Look at size of L:
    if L >= k
        d is in L
    if L = k-1
        then p is d
    if L < k-1
        d in R: k-(L+1)th-smallest in R
```

Map/Dictionary

- create empty map
- insert (key, value)
- delete (key, value)
- find(key)

Hash Tables

- h(key) = key mod m
- Insert(key, data): A[h(key)] = data - $\Theta(1)$
- Search(key): return A[h(key)] - $\Theta(1)$
- Delete(key): A[h(key)] = NULL - $\Theta(1)$

Problems

COLLISION - How to handle collisions?

- Make m better
- Make $h()$ better

*** Even if $m \geq n$, can still get collisions
e.g. Birthday Paradox

Methods of Handling Collision

- Seperate Chaining
  - Linked List
  - BBST
- Open Addressing
  - Linear Probing
  - Double Hashing
  - Cuckoo Hashing

Crytographic Hash

Hashing function is still a hashing function.

- Often compared by its length

Features of crytographic hash function

- Collision resistant
- Second preimage resistant
- Preimage resistant

Application of crytographic hash

- Most digital signature algorithms sign a hash of the message
- Anti-Virus
  - Check hash of the virus to detect
  - Check hash of the state that the computer should be
- Storing Username/Password
  - Hashing a password when loging in to see whether or not it matches the username provided

Digital Signature

- Alice generates
  - a public key (which she tells everyone) - for verification
  - a private key (which she keeps secret) - for signing
- Digital signature is a link between a particular message and a particular key
  - `Sign(message, private_key) = Signature`
    - A string of bits that Alice appends to her message
  - `Verify(message, signature, public key)`
    - Allows Bob to check that Alice's public key was used to sign
- Hence, without Alice's private key, it's impossible to do anything with the message. If you do, the verification process will fail.
- e.g. RSA, DSA

RSA hash & sign

If RSA hash is easy to invert, what can they do?

1. Reverse engineering from signature, using public key to compute the digest
2. If the collision is found:
   - Manipulate the content of the message:
   - e.g. "A owns B $10" and "A owns B $100" where both of the message have the same signature

Certificate

Special kind of message

- The message says "XYZ is the public key of So-and-so"
- Signer is someone you know their public key
- Sometimes come in chain

Example: Google certificate transparency logs

- Certificate is the way trusted authority (like Google or Verisign) attests to the public key of the website you're visiting

    - You can check the authority’s digital signature on a certificate (and your browser does) but sometimes certificates need to be revoked
        - Because the person lost the key
        - Because the certificate was issued by mistake
        - Because the key was exposed

Data Compression and Coding

How is data encoded in most computers?

- ASCII (American Standard Code for Information Interchange)
- Unicode and the ISO/IEC 10646 (UTF-8, UTF-16, UTF-32)
- 8-bit ASCII (ISO-8859-1 (Latin 1))

Cost in bits per symbol

- The data to be stored is called a `message`
- Each letter to be encoded is called a `symbol`
- Each bit pattern for each symbol is called a `codeword`
- A collection of codewords forms a `code`
- The cost of a code can be measured in `bits-per-symbol`
    - e.g. ASCII always has the cost of 8 bps

Wasted bits?

- For English text, ASCII always has the first bit as 0

Unary code

To store n, output n 1-bits then a 0-bit

- The best to compress a symbol is the *Entropy* of the symbol
- If a symbol occurs with probability p, the best possible encoding for that symbol is $-log_2(p)$
- The inverse of this is that a codeword of length x implies an expected probability of occurrence of 2^{-x} or $2^{-length}$

Static vs semi-static codes

- ASCII and unary are example of static code, every symbols have the same length of codeword, regardless of message
- A *semi-static* code knows the probability of occurrence of each symbol in the message

- Shannon’s entropy theorem: the best bps for a message is

$$H(p) = -\sum_{i=1}^{n} p_i log_2 p_i$$

How to find best semi-static codewords

- Divide and Conquer
  - Shanon-Fano algorithm
    - Divide the possibility by half (left/right) and join recursively to from a tree
  - Huffman algorithm
    - Join two symbols that have the lowest possibility to form a tree recursively

Alistair’s in-array implementation of Huffman

- Pass One:
  - Turn n weights into $n-1$ internal node weights and then $n-2$ internal parent pointers.
- Pass Two:
  - Turn $n-2$ internal parent pointers in to $n-1$ internal node depths, using $A[i] \leftarrow A[A[i]]+1$.
- Pass Three:
  - Turn $n-1$ internal node depths in to n leaf depths.

Secret Sharing

Suppose you want to hide a secret s

- It might be a number or a string of bits
- Can you give a,b to two other people so that
- Neither person alone has any information about s
  - a or b alone reveals nothing
- But both people working together can find s
  - Some computation on a, b recovers s

Ans: give one person rand r.
Give the other person r XOR s (bitwise).
Recover s by computing r XOR (r XOR s).

Think of s as a number in range [0,n-1]. Give one person r in the range *** and n
Givetheotherpersons+rmodn ***andn Recover s by computing (s+r) – r mod n

Dynamic Programming

Each recursive call adds result to the table which increase the performance time as the algorithm doesn't need to re-calculate the same value again. Becareful of this case as Dynamic Programming is really close to distinct memory.

1. Assign a table
2. Assign known value
3. If compute new value, add that to the table
4. Repeat 3 recursively until the task is done

Each step 3 calls reduce the running time of the algorithm as the access time of an array is $O(1)$.