Dijkstra’s algorithm

DFS  running

Stack time O(|V|+|E|)

dfs(v) { // v is a vertes

for each x ∈ V

x.visited = false

S←(v) // stack, first in, last out

D←() // list for output

while S not empty

v in S.pop\_back()

if v is not visited

D.push\_back(v)

v.visited←true

for each x, st (v,x) in E and ! x.visited

s.push\_back(x)

}

d[s] ←0

for each v in V-{s}

do d[v] ←inf //initialization

S←empty

Q←V //Q is a priority queue maintaing V-S

while Q != empty

do u←Extract-Min(Q)

S←S and {u}

for each v in adj(u)

do if d[v]>d[u]+w(u,v)//relaxation step

then d[v]=d[u]+w(u,v)

P

D

K=1

K=2

K=0

K=5

K=3

K=4

Warshalls Algorithm

for k←1 to n //n=|V|(num of

do for i←1 to n vertices)

do for j←1 to n

do if cij > cik + ckj

then cij ← cik + ckj

runs in O(n3)

BFS running time O(|V|+|E|)

Queue

Bfs(V0)

for each x in V

x.state←white

v0.d←0

v0.p←nil  // parent node in BFS

Q← (v0)  // queue

D← ()  // list of output

while Q not empty

v←Q.pop\_front()

D.push\_back(v)

v.state←red

for each x st (v,x) in E and x.state==white

Q.push\_back(x)

x.state←green

x.d←v.d+1

x.p←v

}

Topological sort

ordering vertices in a DAG

TopSort(G) {

Q← () //queue for indegree zero vertices

T← () //out put list

for each v in V in G

v.ingree=ComputeInDegree(v)

if v.ingree==0

Q.push\_back(v)

while !Q.empty() do

v←Q.pop\_back()

T.push\_back(v)

for each w, st (v,w) in E

w.indegree--

if w.indegree==0 then

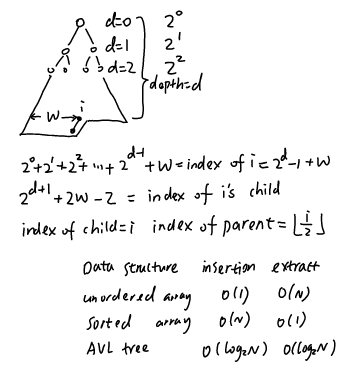
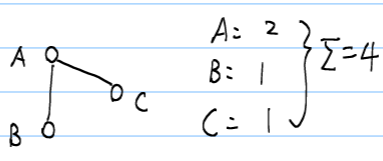
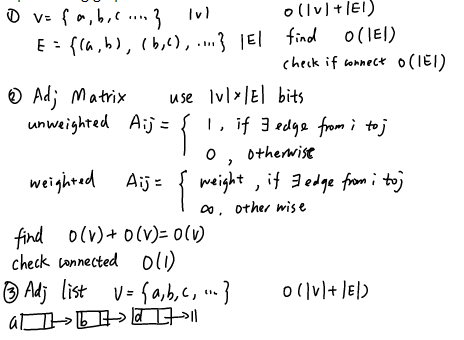
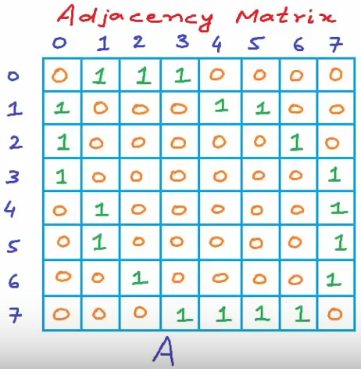
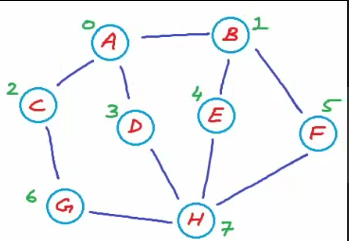
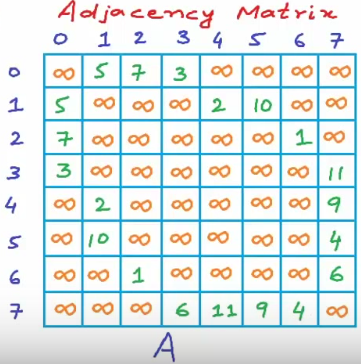
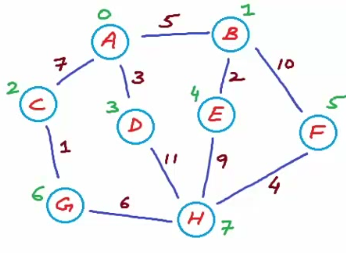
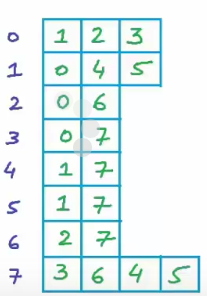
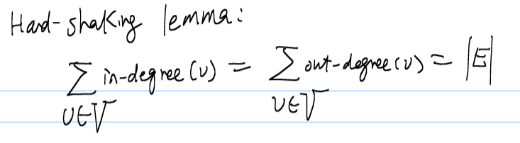
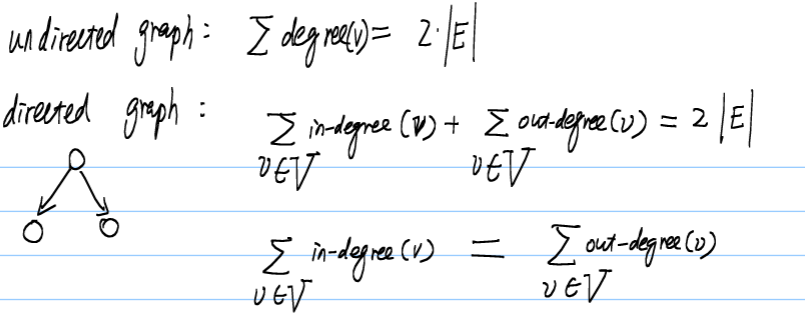
Q.push\_back(w)

if T.size()<|V|

error

return T

}



Quadratic Probing

h(k,i) = (h’(k) + i^2) mod m

Adv: faster run time because of fewer collisions

Disadv: will cost a lot of time to find a good secondary hash function

Hash

ASCII code -sum/weighted integer/Radix -128 integer

Resolve collision with Chaining

Insert O(1)

Search O(|T[h(x)]|) size of chain at h(x)

Deletion same as search

Hash function

Resolve collision with Open dressing include

m is the table size

h’(k) = k mod m

Linear probing

h(k,i) = (h’(k) + i ) mod m

= ((k mod m) + i ) mod m

Adv: simple idea; Removals are clean; used when memory is of concern

Disadv: unevenly distributed keys(long list, many empty space); cost memory

Double probing

h2(k) = R + (k mod R)题目给, R a prime smaller than table size will work well

h(k.i) = (h’(k) + i\*h2(k) ) mod m

Adv: easy to implement; low overhead

Disadv: primary clustering (keys tend to cluster)

BFS的连接点就是里面的children个数

Representation

Graph

G=(V,E) , V = vertex/node, E = edge

In-degree: num of income edge,

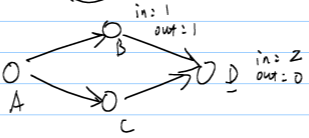
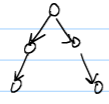
out-degree=num of outcome edge,

Degree = in+out



cycle is a path of length at least one such that V1==Vn

DAG: Directed Acyclic Graph, tree, there is only one path between any two nodes



Hand-shaking lemma

Heap

FindMin: O(1)

FindMax: O(N), number of leaf nodes

Build a heap: O(N)

insert: compare with parent, bubble it up, O(nlogn)

Heap\_insert(H,key) {

Heap\_size(H) ←Heap\_size(H)+1

i←Heap\_size(H) // i is index

while i>1 and H[parent(i)]>key

do H[i] ←H[parent(i)] // parent=[i/2]

i←parent(i)

H[i] ←key

}

ExatractMin: swap with last, swap,

Heap\_Extraction(H) {

min←H[i]

H[1] ←H[Heap\_size(H)] // last item→root

Heap\_size(H) --

Heapify(H,1) // 1 is index

return min

}

Heapify(H, i) {

l←Left(i) //i\*2

r←Right //i\*2+1

//determine the smallest of l,r,i

if l<=Heap\_size[H] and H[l]<H[i]

smallest<-l

else

smallest<-i

if r<=Heap\_size[H] and H[r]<H[smallest]

then smallest←r

if smallest != i

then swap(H[i],H[smallest])

Heapify(H,smallest)

}

Heap\_Sort (H){

Build-Heap(H); // O(N)

for (i←length[H] down to 2) // O(N)

do swap(H[1], H[i]); // O(N)

heapsize(H)--; // O(N)

heapify(H,1); // O(NlogN)

}