Algorithmic thinking

General Process:

1. understanding the problem

2. formulating the problem

3. developing an algorithm

4. implementing the algorithm

5. running it on the data

Brute Force – graph degree distribution

(1) The small-world problem

if I take any two individuals in the world, what is the shortest path that connects these two people in terms of who knows who relationship.

input: graph

output: a distribution of the realized distances

(2) graphs representation

undirected graph

V(finite set of nodes) e.g. V = {0, 1, 2, 3}

E(a set of undirected edges, an edge is a set of two nodes) e.g. E = {{0,1}, {1,2}}

directed graph

V is in the same case as for undirected graph

E is now set of directed edges e.g. E = {(0,1), (1,2)} (direction: 0 to 1, 1 to 2)

common representation of graph:

using V&E will lead to a lot of redundancy facing large graphs

common representation:

**adjacency list**- a list with a node column and a “set of all adjacent nodes”

|  |  |
| --- | --- |
| nodes | Set of all adjacent nodes |
| 0 | 1,3 (order does not matter) |
| 1 | 0,2 |
| 2 | 1,3 |
| 3 | 0,2 |

**adjacency matrices**- n\*n matrices, A[i,j] = 1 (i to j for directed; connected for undirected); 0 (not)

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |

if the graph is sparse, adjacency list is better; otherwise if it is dense, adjacency matrices is better.

(3) degrees of nodes

undirected: edges connected to a node

directed: in-degree (neighbors are connected by an edge from them to the node)/ out-degree (vice versa)

distance: smallest k for which there is a path of length k between these two nodes

simple path: does not repeat any nodes in the path

(4) Brute Force Algorithm

For a graph with n nodes

for every two nodes

found = False

for k = 1 to n-1:

if there is a path of length k between these two\*:

found = True

distance = k

break

if found = False

distance = infinity

\*to check whether there is a path of length k between two nodes:

try every possible subset of k-1 nodes, and try every arrangement or permutation of these subsets between the two nodes.

Brute Force Algorithm is easy to understand, easy to argue about correctness, easy to implement, but the major problem is the efficiency.

(5) Algorithm efficiency

time- when we analyze the running time, we have to make assumptions about representations of the data and implementation details (e.g. adjacency list/ matrices)

space- how much memory does it take?

measuring efficiency (focus on time efficiency)

for graph, input size can be expressed by N nodes, M edges

measure the number of operations be executed as a function of M and N

assume basic operations like addition, division, indexing and calling a function takes fixed amount of time which is independent of the input size

focus on worst-case analysis

(6) Measuring efficiency for Brute Force:

what is the input size: number of nodes N, number of edges M

assume a adjacency matrix is used

worst-case: when nodes I and J have no path between them (not connected)

1 k ← 1

2 while k < |V| do

3 ul ← i

4 uk ← j

5 for each subset V’ of V with size k-1 do

6 for each permutation u1,…uk-1 of the elements of V’ do

7 IsPath ← True

8 for L ← 0 to k-1 do

9 if {ul, ul+1} ∉ E then

10 IsPath ← False

11 if IsPath = True then

12 return k

13 k = k + 1

14 return infinity

look at individual operations:

line 1, 3, 4, 7, 9, 10, 11, 12, 13, 14: one operation each

line 2: the size of graph V is N, so k will take number of 1, 2, … N-1; loop N-1 times

line 5\*: subset with size of k-1 of set with size of n: nC(k-1)

line 6\*\*: (k-1)!

line 8: iterate k times

\*the number of subsets of size k of a set of size n is nCk = n!/((n-k)!k!)

\*\*the number of permutations of a list of n elements is n!

counting operations:

line 1 and 14 are outside the loop: 2

2-13 loop: (n-1) times

line 3, 4, 13 are executed once every time in the loop: 3 \* (n-1)

5-12 loop:

line 7, 11 will be executed once

line 12 will never be executed in worst case

running time =

n = 2: 13

n = 5: 500

n = 10: >13 million

extremely inefficient

rough estimation of running time:

total number of subsets of size n set: 2n-1,thus lower bound is 2n-1

thus grows extremely fast

Breadth First Search (BFS) – computer resilience

(1) Asymptotic Notations

put upper and lower bounds on functions

give us an idea of how fast the function grows

upper bound:

for the running time function f(n):

we want a function g(n),

there exist two constant C>0, n0>= 0,

where f(n) <= C\*g(n) for any n>=n0

→**f(n) = O(g(n))**

lower bound:

we want a function g(n),

there exist two constant C>0, n0>= 0,

where f(n) >= C\*g(n) for any n>=n0

→**f(n) = Ω(g(n))**

if there exist C1 and C2,

where f(n) <= C1\*g(n) and f(n) >= C2\*g(n) for any n>=n0

(or say, if f(n) = O(g(n)) and f(n) = Ω(g(n)))

→**f(n) = Θ(g(n))**

we often want upper/lower bound as tight as possible

(2) Big O

upper bound

the running time will not be worse than this

we say f(n) = O(g(n)) if

there exist C>0 , n0>= 0,

such that f(n) <= C\*g(n) for any n>=n0

to find the upper bound:

for f(n), we first ignore lower term

e.g. f(n) = 1000 + 17n + 2n2

f(n) = 1000 + 17n + 2n^2 <= 1000n2 + 17n2 + 2n2=1019n2 (n>=1)

thus for g(n) = n2, f(n) = O(g(n))

(3) Breadth First Search (BFS)

denote dj as the distance between node 0 and node j

layer 0: node 0: d0 = 0

layer 1: each neighbor node i of node 0: di = 1

layer 3: each neighbor node of nodes in layer 1(excluding layer 0 and 1 nodes): dj = 2

order:

initialize every distance dj to infinity

first compute distance for neighbor nodes of 0: e.g. 1,3,4

then compute distance for neighbor of 1, 3, 4

to keep track of the nodes: use a queue

always add element at the tail, and extract element at the head

enqueue(Q,x) put x at the tail

dequeue(Q) get element at the head out of the queue

every time to compute distance for a neighbor, check whether it has been computed

(4) Pseudocode

1 initialize Q to an empty queue

2 for each j ∈V do

3 dj ← infinity

4 di ← 0 (i=0)

5 enqueue (Q, i)

6 while Q is not empty do

7 j ← dequeue(Q)

8 for each neighbor h of j do

9 if dh = infinity do

10 dh ← dj + 1

11 enqueue(Q, h)

12 return d

\* assumptions:

the graph is given by adjacency list

the graph is connected (there is a path from i(the initial node) to any other node)

N nodes and M edges

efficiency:

1 O(1)

2 O(N)

3 O(1)

4 O(1)

5 O(1)

6 O(N)

every node is going to be added once and every iteration exactly one node is going to be removed

7 O(1)

8 O(N) worst case: N-1 neighbor

9 O(1)

10 O(1)

11 O(1)

12 O(1)

running time: 1-5: O(N), 6-12:O(N2) → O(N2)

every edge will be looked at twice, that is 2M times total in the loop during 6 to 12

so we can also say, 6-12: O(M)

the running time is O(N+M)

for O(N2) and O(N+M)

when the graph is really tense, M is on the order of N2

when it is really sparse, O(M+N) will be a more accurate representation

normally m>n, thus O(N+M) can be simplified as O(M)

(5) Distance distribution

algorithm to compute distance distribution (for the small world problem)

compute distances between every two nodes

input: a graph G=(V,E)

output: frequency of each possible distances (N-1 at most)

f[0 … |V|-1] (f[d] is the frequency of distance d)

for d ← 0 to |V|-1 do

f[d] ← 0

for each i∈V do

dist ← BFS(G, i)

for each j∈V do

f[distj] ← f[distj]+1

return f

efficiency: O(MN)

when the graph is very tense, at worse, it will be O(N3)

Sorting & Searching – map clustering

two most common problems: sorting and searching

Sorting:

partial order: not all items in one sets are comparable to each other

sorting problem: given a list of items, we want to sort them in certain order, and we need to switch the order of the satellite data together with the main key

(1) brute force algorithm:

loop over all permutations and find the one satisfies the specific order

n! permutations, really inefficient

(2) simple quadratic algorithm:

for a list L (to be sorted in ascending order)

create another list L’

loop over L to find the minimum value and add into L’

then remove the minimum from L

loop again to find the minimum in current L …

**O(n2)**

(3) MergeSort

**O(nlogn)**

sort a number list in ascending order

divide and conquer approach: break a problem into subproblems, then solve each of them

when we have two sorted sublists, we know how to merge them together in a big sorted list

compare the first element in each of them and put the smaller one in big list

compare the second element in the element-removed list and another element

…

subproblem for sorting problem:

divide a list into two halves and sort each of them, and merge them

to sort each of them, divide the half into two quarters and sort them and merge them

…

when each sublist only contains 1 element, it’s easy to merge, and already sorted- this is the simplest instance of the problem

**⬆recursive algorithm**

MergeSort(L[0…n-1])

MergeSort(L[0…(n-1)/2])

MergeSort(L[(n-1)/2+1,…,n-1])

merge the two sublists- O(n)

total operation time :

T(n) = 2T(n/2) + O(n) ——recurrence

T(n) = 2(2T(n/4)+O(n/2)) + O(n)

T(1) = O(1)

T(n) = O(nlogn) ——best we can do for comparison based sorting algorithm

**The Master Theorem:**

Given T(n) = aT(n/b) + f(n), T(1) = c,

where a >= 1, b>=2, c>0

if f(n) = O(nd), d>=0, then

(4) Linear vs. binary search

to find one item in a list

LinearSearch:

loop through each item in the list, to see whether match

this algorithm works on any list (sorted or unsorted)

**O(n)**

BinarySearch:

assumptions: already sorted list

going to the middle of the list, and ask the one looking for is before/after

then choose the right half and do again

when meet only one item branch: check whether it is the one looking for

T(n) = T(n/2) + O(1) (divide into to halves with O(1) and take one of them)

T(1) = O(1)

→**T(n) = O(logn)** (actually the height of the tree)

for those unsorted list: O(nlogn) + O(logn) = O(nlogn)

therefore, for a unsorted list, it’s worse than linear search

Dynamic programming

(1) The RNA secondary structure problem

RNA secondary structure: RNA moves/loops on itself

RNA sequences: looks like a string of combination of letters AUCG

every letter in the string could be one of A,U,C,G

recursively: base case is one of AUCG, then add one on to them

input: a sequence like AAUUACCG

output: prediction of how the sequence is going to fold

set of pairs like {(i,j), (k,l), (m,n), …}

rule1: A must go with U, C must go with G (complementarity)

rule2: some letters in the middle do not match with each other (create a loop)

letters separated by at least 4 positions can be matched (no sharp turns)

rule3: cannot match the same letter more than once

rule4: for i<k<j<l in a sequence, cannot match (i,j) and (k,l), but (i,l), (k,j) is ok (cannot have crossing situations)

\* empty set will always satisfy these rules, but we want the set with a maximum possible number of such pairs that satisfy these constrains.

(2) Dynamic programming

B = b1b2b3…bn

bi ∈ {A, U, C, G}

OPT(i,j): the maximum numbers of pairs of indices that you can put in the set that satisfy the four requirements of feasibility

e.g.

OPT(1, n) is the maximum number of pairs of indices that you can match together to satisfy the four requirements in the feasibility of the problem when you look at the entire sequence. So it is exactly the size of pairs that the biologist is interested in.

for every letter: part of the solution (matched)/ outside the solution (not matched)

if position n is not part of the solution: OPT(1, n) = OPT(1, n-1)

if position n pairs with position t (t<n-4): OPT(1, n) =1 + OPT(t+1, n-1) + OPT(1, t-1)

**OPT(i, j) = max{OPT(i, j-1), maxt(1 + OPT(t+1, j-1) + OPT(i, t-1))}**

(3) DP algorithm

Initialize OPT(i, j) <- 0 whenever i >= j-4

for k <- 5, 6, …, n-1

for i <- 1, 2, …, n-k

j <- i+k

OPT(i, j) = max{OPT(i, j-1), maxt(1 + OPT(t+1, j-1) + OPT(i, t-1))}

return OPT(1, n)

store OPT(i, j) as a matrix, and whenever needed, retrieve the value from the matrix

we solve sub-problem first and store the value in a matrix, then solve other problems using the result of the sub-problems

if the sub-problem’s result is used several times afterwards, then the dynamic algorithm will be efficient.

running time:

k loop: O(n)

i loop: O(n)

OPT(i,j) computation: loop t is O(n), so the whole is O(n)+1 ~ O(n)

running time = O(n3)

faster then recursive algorithm, in which OPT(i,j) will call the function over again

backtracking:

we need to know exactly which two of the numbers will be paired

from OPT(1,n), compare with OPT(1, n-1), we know n is or is not part of the solution, then go back to (1, n-1) and …

(4) DP v.s. recursive implementation

e.g. extract a subset with size of k from the set {1, 2, …, n}

recursive implementation:

for n: two status- included or excluded

included: k-1 elements from {1, 2, …, n-1}

excluded: k elements from {1, 2, …, n-1}

thus Cnk = Cn-1k-1 + Cn-1k

base case: Cn0=1, C0k=0

this problem then transit to “compute\_nchoosek(n,k)”:

if n>= 0 and k = 0:

return 1

elif n = 0 and k > 0:

return 0

else:

return compute\_nchoosek(n-1,k) + compute\_nchoosek(n-1,k-1)

dynamic programming:

two nested loop for n and k

know exactly each Cnk

solving from smaller problems to big problems

(5) Sequence alignment

evolution from a same ancestor

mutation:

1. changes a letter- pair substitution

2. insertion/deletion: some part get deleted or some other part inserted

global alignment problem

input: two sequences X and Y from different existent animals

output: an alignment of X and Y –

allow insert dash in either of them in any position

definition of feasible alignment

the two sequences must be of the same length

don’t put a dash at the same position for both sequences

removing the dashes in the alignment, you get the original sequence

optimal alignment- employ a scoring scheme/matrix

5\*5 matrix (ACGT- \* ACGT-)

for each blank in the matrix: score for different matches

at each position of alignment: two nucleotides or a nucleotide+ a dash

same nucleotides: 10

different nucleotides: 4

dash + nucleotide: -4

for each alignment, returns a score, and the higher the better

local pairwise sequence alignment

try every substring of X and of Y and globally align them using scoring matrix

we have one optimal score, but for the score there may be several alignments

B(m,n) = number of global alignments of two sequences with lengths m and n

B(0, 0) = 0

B(0, 1) = 1

B(0, 2) = 1

B(1, 1) = 3

B(1, 2) = 2 + 3 = 5

B(2, 2) = 1 + 3\*2 + 4C2 = 13

B(2, 3) = 3 + 4\*3 + 5C2 = 25

B(1, 3) = 3 + 4 = 7

B(m,n) = B(m-1,n) + B(m,n-1)+B(m-1,n-1)