(Finishing up separate chaining hash tables)

Note: We could use fancier data structures to store all the keys that

collide to the same array index (e.g., search tree or even a "sub" hash table),

but if we have a well-designed hash table, so that the table size, hash

function, and actual keys encountered work well together, we expect the

number of collisions to be small and it might not be worth the complexity.

(C++ programming details discussed on pp. 197-199 deferred to lab.)

So how long do we expect the linked lists to be?

Definition: the LOAD FACTOR of a hash table, denoted lambda, is the number

of elements in the table, N, divided by the table size, M.

So in the example above, lambda = 10/10 = 1.

Informally, a good hash function for separate chaining should

distribute the elements evenly among the table entries.

More formally: any key should be equally likely to hash to any of the M locations

in the hash table. This is the ideal.

Impractical to check in practice since the probability distribution on the

keys is usually not known.

Example: Suppose the symbol table in a compiler is implemented with a hash

table (symbol table is a data structure that keeps information about all the

user-defined names, e.g., variables, in a program). The compiler writer

cannot know in advance which variable names will appear in each program to

be compiled.

Heuristics are used to approximate this ideal condition: try something that

seems reasonable and run experiments to see how it works.

Also, try to exploit application-specific information. For the symbol table

example, take into account the kinds of variable names that people often

choose and try to avoid collisions on these names.

Assume we have an ideal hash function for chaining.

Fact: The average length of each linked list is N/M = lambda.

Therefore, the average running times are:

\* insert: O(1), assuming no check for duplicates

\* unsuccessful search: O(1 + lambda)

\* successful search: O(1 + lambda/2) since on average the key being sought

is in the middle of the linked list

\* remove: same as search

For these times to be O(1), lambda must be O(1), so N must be O(M), i.e.,

the maximum number of keys in the hash table must be no more than a constant

multiple of the table size.

\*\* General Rule: Make the table size about as large as the number of elements

expected, so lambda is about 1.

Section 5.4: Hash Tables without Linked Lists / Probing Hash Tables / Open

Addressing

---------------------------------------------------------------------------

With this scheme, there are no linked lists. Instead, all keys are stored

in the table proper.

If there is a collision when trying to insert a key, you have to

"probe" the table, i.e., check whether a table entry is empty,

repeatedly until finding an entry that is empty. This will only work

if the load factor is at most 1.

You must pick a pattern that you will use to probe the table.

(1) linear probing:

To insert key x, start at h(x), then check h(x)+1, h(x)+2, h(x)+3,...,

wrapping around to check 0, 1, 2,... if necessary, until finding an

empty entry.

Example: M = 10, h(x) = x mod 10.

Insert 89, 18, 49, 58, 69.

0 1 2 3 4 5 6 7 8 9

-----------------------------------------------

89

18

49

58

69

How about searching for x? Start at h(x), then keep probing in the same

pattern as for insert until either finding x or reaching an empty entry (not

found).

We'll talk about remove later.

Problem with linear probing: Leads to build-up of "clusters"

(contiguous group of full entries). If an insert probe sequence

begins in a cluster:

- it takes a while to get out of the cluster to find an empty slot, and

- inserting the new element just makes the cluster even bigger

Claim: The expected number of probes for linear probing is

about (1 + 1/(1-lambda)^2)/2 for insertions and unsuccessful search and

about (1 + 1/(1-lambda))/2 for successful searches.

Calculation is beyond the scope of our class but we can do a sanity check:

If lambda is very small (table is almost empty), then this is close to 1.

As lambda approaches 1, this grows without bound.

Of course, there is no point in continuing once all slots have been checked.

Rule of Thumb: Don't use linear probing if table is expected to be more than

half full.

(2) Quadratic Probing: Similar to linear probing but offset for continuing

the search keeps growing with every unsuccessful probe to avoid clustering.

Specifically, the offset for the i-th probe is i^2 ("quadratic").

If we are searching for key x, where h(x) = i, the probe sequence is:

A[i]

A[(i+1) mod M]

A[(i+4) mod M]

A[(i+9) mod M]

A[(i+16) mod M]

etc.

Example:

0 1 2 3 4 5 6 7 8 9

-----------------------------------------------

89

18

49

58

69

58: probe sequence is 8 (full), 8+1 = 9 (full), (8+4)%10 = 2

69: probe sequence is 9 (full) (9+1)%10 = 0 (full), (9+4)%10 = 3

It's particularly important for M to be prime to make sure that an empty entry

can be found (for insert) if there is one.

Counter-example: Suppose M = 6. If a probe sequence starts at A[0], then

A[5] is never probed (can be proved with some number theory that there is no

i such that 0 + i\*i = 5 mod 6).

In fact, there is no guarantee of finding an empty cell once the load factor

is more than 1/2, and if the table size is not prime this problem can occur

even with a lower load factor.

However, we do have this guarantee:

Theorem 5.1: If quadratic probing is used and M is an odd prime, then

a new element x can always be inserted if lambda < 1/2.

Proof: We'll show that the first ceiling(M/2) entries probed are all distinct

indices. Since at most floor(M/2) entries are filled, we'll find an empty one.

(Since M is odd, floor(M/2) < ceiling(M/2).)

Suppose in contradiction there exist distinct i and j with

0 <= i, j <= ceiling(M/2)-1 but

h(x) + i^2 = h(x) + j^2 (mod M).

Then:

i^2 = j^2 (mod M)

i^2 - j^2 = 0 (mod M)

(i-j)(i+j) = 0 (mod M)

Thus (i-j)(i+j) is a multiple of M. Since M is prime, either (i-j) or (i+j)

(or both) is a multiple of M: there are no other factors that can be combined

to produce M as could be done with, say, 5\*4 = 20 = 2\*10.

Case 1: (i-j) is a multiple of M. That is, (i-j) = 0 (mod M).

So the difference between i and j is a multiple of M.

But since i != j and both i and j are between 0 and ceiling(M/2)-1, this is

not possible.

Case 2: (i+j) is a multiple of M. That is, (i+j) = 0 (mod M).

That is, i+j is 0 (which is impossible since i and j are nonnegative and i != j),

or M or 2M or 3M,... (which is impossible since i and j are at most ceiling(M/2)-1).

Since both cases are impossible, the proof by contradiction is complete.

QED

Even with quadratic probing (or variations where the probe offset is dependent

on the key), we get a form of clustering where the entries of the table are

filled in a fixed pattern. The problem is that two keys that hash to the same

initial entry will follow exactly the same probe sequence.

To get around the problems with quadratic probing:

(3) Double hashing.

One hash function h(x) is used to determine where to start probing for key x.

A second hash function h'(x) is used to determine the probe sequence (e.g.,

the offset).

If the hash functions are chosen properly, different keys that have the same

starting place will have different probe increments.

If h(x) = i and h'(x) = z, the probe sequence is:

A[i]

A[(i+z) mod M]

A[(i+2z) mod M]

A[(i+3z) mod M]

A[(i+4z) mod M]

etc.

Must ensure that h'(x) is nonzero. A common choice is

h'(x) = R - (x mod R), where R is a prime < M.

Example: h(x) = x mod 10, h'(x) = 7 - (x mod 7)

0 1 2 3 4 5 6 7 8 9

-----------------------------------------------

89

18

49

58

69

60

49: h(49) = 9, h'(49) = 7; probe sequence is 9, 6.

58: h(58) = 8, h'(58) = 5; probe sequence is 8, 3.

69: h(69) = 9, h'(69) = 1; probe sequence is 9, 0.

60: h(60) = 0, h'(60) = 3; probe sequence is 0, 3, 6, 9, 2.

Here's why having a non-prime table size, as in this example, is bad.

Suppose next we want to insert 23: h(23) = 3, h'(23) = 5, so probe sequence

is 3, 8, 3, 8, 3, 8, forever.

Theoretically, a good (IDEAL) hash function for open addressing should

satisfy this property:

\*\* Each key should be equally likely to have each permutation of {0,1,...,M-1}

as its probe sequence. \*\*

This is even harder to achieve in practice than the ideal property for chaining.

Double hashing gets closer than linear probing or quadratic probing.

A good approximation is often achieved with this version of double hashing:

Let M be prime, h(x) = x mod M, h'(x) = 1 + (x mod(M-2)).

Claims: Assuming an ideal hash function for open addressing, average

case running times are:

\* insert: O(1/(1-lambda))

\* unsuccessful search: O(1/(1-lambda))

\* successful search: O((1/lambda)\*lg (1/(1-lambda)))

Proofs are out of scope of our class. Sanity check:

Intuitively the time for search should increase as lambda increases, which

is true with these formulas. At the extreme, when n = M-1, the formula

1/(1-lambda) = M, meaning you will search the entire table before discovering

the key is not there.

-----------------------------

How to remove an element from a hash table implemented with open addressing?

Same issues for linear probing, quadratic probing, and double hashing.

First cut: Find x and replace it with some indication of "empty".

Problem: Recall this example:

Example: M = 10, h(x) = x mod 10.

Insert 89, 18, 49, 58, 69.

0 1 2 3 4 5 6 7 8 9

-----------------------------------------------

89

18

49

58

69

Suppose we delete 18 and then search for 58. We'll think that 58 is not in the

table, which is incorrect.

Instead, replace deleted elements with a special "tombstone" symbol. During a

search, if you encounter one of these, then keep searching until either finding

the key or hitting a truly empty entry.

Section 5.5: Rehashing

-----------------------

For both separate chaining and open addressing: What can you do if the load

factor starts getting too big and performance is degrading?

Familiar solution: Allocate a hash table that is twice as big as the current

one and "rehash" all the keys from the current table to put them into the new one.

As usual, every once in a while an operation will be very slow, but the expense

is amortized over many subsequently fast operations so the average time for

operations is still constant.

Section 5.6: Hash Tables in the STL

------------------------------------

To be discussed in lab.

Section 5.8: Universal Hashing (also see powerpoint slides)

-------------------------------

For separate chaining, the constant time performance of operations on

a hash table depends on the interaction between the hash function and

the actual keys that are to be stored ensuring that the keys are

distributed uniformly among the array entries. This is a fragile situation,

as it puts constraints on the distribution of the input keys, which can be

error-prone. If we mis-calculate, then the performance can be bad.

Can we design a more robust scheme? We'd like a scheme where the probability

of good performance is under the control of the algorithm, not the inputs.

With UNIVERSAL HASHING, we choose the hash function to be used for the execution

initially at random from a set (or "family") H of different hash functions.

All the hash functions in H use the same table size M.

H must ensure that for all keys x and y, the number of hash functions h in H

for which h(x) = h(y) is at most |H|/M.

This implies that the probability of a collision between any two keys is

at most 1/M, no matter which two keys we are considering!

Thus the average length of each list is at most 1 + N/M, for ANY set of N keys.

From Cormen et al. textbook: "randomization guarantees that no single

input will always evoke worst-case behavior. Because we randomly

select the hash function, the algorithm can behave differently on each

execution, even for the same input, guaranteeing good average-case

performance FOR ANY INPUT."

The remaining question is whether such a family of hash functions exists.

The answer is yes. Here's an example.

Theorem: The hash family H = {H\_{a,b}(x) = ((ax+b) mod p) mod M, where

1 <= a <= p-1 and 0 <= b <= p-1} is universal.

The proof uses probability and number theory to show that for any two keys

x and y, the number of hash functions h in H (i.e., the number of choices for

a and b) for which h(x) = h(y) is at most |H|/M. We will skip the proof.

So to choose the random hash function at the beginning of the execution

that will use a hash table,

- choose M appropriately (e.g., M should be O(N), where N is the number of

keys that will ever be stored in the hash table)

- choose a prime p larger than the largest input key you would ever expect

to encounter

- uniformly at random choose a between 1 and p-1

- uniformly at random choose b between 0 and p-1

Example: N = 100, M = 120, keys are chosen from {1,...,10000}, p = 10007.

A couple of the possible hash functions are:

- H\_{3,8}(x) = ((3x+8) mod 10007) mod 120

- H\_{876,5001}(x) = ((876x+5001) mod 10007) mod 120

With universal hashing, collisions can still occur, but their likelihood is

independent of the actual keys being handled.

Section 5.7.1: Perfect Hashing for Worst-Case O(1) Access

----------------------------------------------------------

So far, we've been discussing the average time for hash table operations.

But what if we want the WORST-CASE time to be really good, in particular,

constant?

We'll discuss a way to do that in the special case when all the keys to

be stored in the hash table are known in advance. This assumption makes

sense for applications such as compiler symbol tables (to hold the reserved

words in a programming language), or set of file names on a CD-ROM, and

IP address lookup in a network router. The result is a lookup table: we

only do searches on it, no inserts or deletes.

We also want the amount of space to be O(N), where N is the number of keys.

The idea:

\* First, use hash function h, to decide which table entry to put x into

(namely, entry h(x))

\* All the keys that hash to the same table entry are stored in another

hash table, instead of a linked list.

\* Each entry j in the main hash table has an associated second-level

hash table with its own hash function, h\_j.

\* If item x hashes to location j in the main hash table, then h\_j

is used to decide where to store x in the second-level hash table

(namely, entry h\_j(x))

(See Figure 5.24 on p. 214.)

Goal 1: Make sure that we NEVER have collisions with any of the h\_j's.

To choose the top-level hash function h:

Choose a hash function from the universal hash family H\_{p,N},

where p is a prime number greater than any key value.

Note that the table size is the same as the number of keys to be stored.

Check how many collisions are produced. If the number is "too big"

(specified below), then try again by choosing another hash function

from the universal family.

A probabilistic analysis can be used to show that you don't have to try

too many times before finding a hash function with not too many collisions.

For each entry j in the main hash function (0 <= j < N):

Choose a hash function from the universal hash family H\_{p,(n\_j)^2}, where

n\_j is the number of keys that need to be stored in the hash table

for entry j (i.e., the number of keys that collide at entry j using

the top-level hash function). Note the square!

If ANY collisions are produced, then try again by choosing another hash

function from the universal family.

A probabilistic analysis can be used to show that you don't have to try

too many times before finding a hash function with NO collisions.

(Relies heavily on the square.)

Goal 2: Make sure the total space used by all the hash tables is O(N).

This is where we need the top level hash function to have a certain property.

If all the keys hash to the same entry, then the second-level hash table for

that entry needs to have N^2 entries, and we have Theta(N^2) space, not

Theta(N). So we need the keys to be somewhat spread out by the top-level

hash function. In particular, we need the sum of all the sizes of the

second-level hash tables to be O(N). So the check that is done on the

top-level hash function h is, letting n\_j be the number of keys that h

hashes to entry j:

Sum\_{j = 0}^{N-1} (n\_j)^2 <= 4N

Example (from Cormen et al. textbook):

- Keys are {10, 22, 37, 40, 52, 60, 70, 72, 75}, so N = 9.

- Top level hash function is h(x) = ((3\*x + 42) mod 101) mod 9.

- T is top level hash table:

- 10 hashes to T[0]

- 60, 72, and 75 hash to T[1]

- 70 hashes to T[5]

- 40, 52, 22, and 37 hash to T[7]

- secondary hash tables for T[0] and T[5] have size 1^2 = 1 and

hash functions h\_0(x) = h\_5(x) = ((1\*x + 0) mod 101) mod 1 = 0.

- secondary hash table for T[2], call it S\_2, has size 3^2 = 9 and

hash function h\_2(x) = (10\*x + 18) mod 101) mod 9

- 60 hashes to S\_2[3]

- 72 hashes to S\_2[4]

- 75 hashes to S-2[7]

- secondary hash table for T[7], call it S\_7, has size 4^2 = 16 and

hash function h\_7(x) = (23\*x + 88) mod 101) mod 16

- 40 hashes to S\_7[7]

- 52 hashes to S\_7[8]

- 22 hashes to S\_7[9]

- 37 hashes to S\_7[14]

Comments:

Note that constructing the perfect hash table is a probabilistic algorithm.

It can be shown that the expected running time is polynomial though (requires,

on average, constant number of attempts to get the top level hash function

to have the required property, and a constant number of attempts to find

a collision-free hash function for each of the second-level hash tables).

Remember, this is a pre-processing step that is just done once.

After the table is set up, all lookups take exactly two probes.

(skip Sections 5.7.2, 5.7.3, and 5.9)

Summary:

-------

\* Hash tables implement search, insert and remove on a set of data times in

constant average time, subject to carefully chosen

- table size

- hash function

- load factor

\* For separate chaining, load factor should be close to 1

\* For open addressing (probing):

- load factor should not exceed 1/2, as performance degrades rapidly as load

factor approaches 1

\* Can do rehashing (resize table) to control load factor

\* In the static case, when all the keys are known in advance, constant worst-case

lookup time is achievable (using perfect hashing and universal hash families).

\* Hard to know which hash table implementation is best: simulation results provide

conflicting guidance, performance depends on type of items, hardware, and

programming language

\* When comparing hash tables to (balanced) binary search trees:

+ BSTs provide faster findMin, findMax, predecessor, successor: logarithmic

vs. linear

+ BSTs provide faster way to list all items in sorted order (or find all those

in a certain range): linear vs. Omega(n log n)

- BSTs have slower (average) time for search, insert delete: logarithmic

vs. constant

\* Important applications of hash tables:

- compiler symbol table: store program identifiers (e.g., variable names)

- graph algorithms: map real-world names of objects that will be the nodes

of the graph into {1,2,3,...,n} for use by the algorithms

- online spelling checkers (look up each word in the file to be checked in a

precomputed hash table made out of the dictionary)

- software caches (Internet browser)

- hardware caches (computer memory)

==============================

Chapter 6: Priority Queues (Heaps)

Let's revisit stacks and queues. In some ways, they are very similar: when

you remove an element from the collection, you get a very specific one:

- stack: you get the newest element (pop)

- queue: you get the oldest element (dequeue)

So stack prioritizes the newest element while queue prioritizes the oldest

element.

We can generalize this to create a priority queue: each element inserted

into the collection has some kind of priority, and when we remove an

element, we get one with the "highest" priority.

It's up to the user to define the priority and what is considered highest.

Example application: Elements are people waiting in the emergency room

of a hospital. Priority is how sick (injured) the person is. Highest

priority is the sickest person.

Note that this is NOT the same as stack or queue, as the sickest person

is not necessarily the one who has been waiting the longest nor the person

who just arrived.

For simplicity, let's assume that the elements in the priority queue are

integers and the highest priority is the SMALLEST element.

Priority Queue Abstract Data Type:

---------------------------------

State: set of elements each with an associated priority drawn from

a totally ordered set.

Operations:

- insert (sometimes called enqueue): add a new element (together with its

priority) into the set

- deleteMin (sometimes called extractMin): remove and return the element

in the set with the highest priority (usually smallest value)

- (findMin, size, empty, sometimes others)

Important: In the basic priority queue definition, NO WAY TO REMOVE ANY

ELEMENT OTHER THAN MIN!

Applications:

------------

- selection (find k-th smallest element)

- sorting

- discrete event simulation

- some important graph algorithms

- operating systems (choose which program to run next)

- network routing (choose which packet to send next)

- etc.

Implementations:

---------------

Attempt #1: linked list

- insert: O(1)

- deleteMin: O(n)

Attempt #2: linked list but keep elements sorted by priority

- insert: O(n)

- deleteMin: O(1)

Attempt #3: balanced binary search tree:

- insert: O(log n)

- deleteMin: O(log n)

Attempt #4: hash table

- insert: O(1)

- deleteMin: Omega(n)

Attempt #5: a new data structure called "binary heap"

- insert: O(log n)

- deleteMin: O(log n)

\*\*\* Advantage of binary heap over balanced binary search tree will

be simplicity and constant factors; ultimately we won't need

to use any links (pointers) \*\*\*

Note: Sometimes "binary heap" is called just "heap", but don't confuse

this use of the word "heap" with the area of memory for dynamically

allocated objects when a program is running.

Definition of binary heap (has 2 parts):

(Part 1) The structure is a "complete binary tree",

which is a binary tree that is completely filled except possibly the

bottom level which is filled from left to right.

Examples of complete binary trees:

\* \* \* \* \* \* \* etc.

/ / \ / \ / \ / \ / \

\* \* \* \* \* \* \* \* \* \* \*

/ / \ / \ / / \ / \

\* \* \* \* \* \* \* \* \* \*

Fact: A complete binary tree of height h has between 2^h and 2^{h+1} - 1

nodes.

Fact: The height of a complete binary tree with n nodes is ceiling(log n),

which is O(log n).

Observation: Since complete binary trees have such a regular structure,

we can implement them simply using an array, with no need for pointers:

We go across level by level in the tree, starting at index 1 (so leave

index 0 empty).

A

/ \

B C

/ \ / \

D E F G

/ \ /

H I J

index: 0 1 2 3 4 5 6 7 8 9 10 11 12 ...

value: - A B C D E F G H I J - - ...

Fact: in the array representation of a complete binary tree, for

the node at index i,

- its left child is at index 2\*i

- its right child is at index 2\*i+1

- its parent is at index floor(i/2)

We will draw heaps as trees (more intuitive) but remember that we will

actually implement them as arrays (or vectors).

(Part 2) "Heap Ordering Property": To complete the definition of

binary heap, we require that, for each node in the heap, the data

stored in it is SMALLER (has higher priority) than the data stored in

its children.

IMPORTANT: No restriction is put on the relative order of the data in

sibling nodes. Smaller value could be on the left or on the right.

Consequence of heap ordering property: smallest value in the entire

heap is at the root.

Examples:

13 this is a heap: complete binary tree

/ \ with heap ordering property

21 16

/ \ / \

24 31 19 68

/ \ /

65 26 32

13 this is not a heap: complete binary tree

/ \ but 6 is smaller than its parent 21

21 16

/ \ / \

6 31 19 68

/ \ /

65 26 32

insert(x) algorithm:

----------------

Here is pseudocode in terms of the tree (recall that references to

parent, left child and right child can be translated into array indices):

1. Add a new node to the heap containing x so that result is

still a complete binary tree

2. Adjust elements in the heap to maintain heap-order property.

How to do step 1: Add a new node at the bottom level of the tree as

far to the left as possible. If the bottom level is already full, then

start a new, lower, level and add the node all the way to the left on

that level. Takes O(1) time.

How to do step 2: "Bubble up" x to a proper place in the heap:

let cur be node just inserted (holding x)

while cur != root and key stored in cur < key stored in parent(cur) do

swap data in cur and parent(cur)

cur := parent(cur) // move up the tree

endwhile

Convince yourself that this restores the heap-ordering property.

Running Time:

- In worst case, we have to bubble all the way up to the root from the

bottom level.

- There are O(log n) levels in the heap.

- The work done at each level is O(1)

- Total is O(log n).

Example:

13 13

/ \ insert 14 / \ swap 14 and 31

21 16 => 21 16 =>

/ \ / \ / \ / \

24 31 19 68 24 31 19 68

/ \ / / \ / \

65 26 32 65 26 32 14

13 13

/ \ swap 14 and 21 / \

21 16 => 14 16

/ \ / \ / \ / \

24 14 19 68 24 21 19 68

/ \ / \ / \ / \

65 26 32 31 65 26 32 31

(Textbook presents an optimization that reduces the number of assignments

required by a constant factor by avoiding swaps.)

deleteMin algorithm:

--------------------

As mentioned above, the element to be removed is stored at the root.

But we cannot just remove the root as it will partition the heap into

two trees. Instead:

1. Change the tree structure to have one fewer nodes:

The only way to do this is to remove the rightmost node on the bottom

level.

2. What do we do with the data element stored in the node being removed?

Put it in the root which has lost its data.

Then "bubble down" to restore the heap-ordering property.

How to bubble down?

let cur be root

while cur is not a leaf and

key stored in cur > key stored in at least one child do

let sc be child of cur with smaller data

swap data in cur with data in sc // don't swap with larger child!

cur := sc

endwhile

Convince yourself this restores the heap-ordering property.

Running Time:

- In worst case, we have to bubble all the way down to the bottom level

from the root

- There are O(log n) levels in the heap.

- The work done at each level is O(1)

- Total is O(log n).

Example:

13 31

/ \ deleteMin / \ swap 31 and 14

14 16 => 14 16 =>

/ \ / \ / \ / \

19 21 18 68 19 21 18 68

/ \ /\ / \ /

65 26 32 31 65 26 32

14 14

/ \ swap 31 and 19 / \ swap 31 and 26

31 16 => 19 16 =>

/ \ / \ / \ / \

19 21 18 68 31 21 18 68

/ \ / / \ /

65 26 32 65 26 32

14

/ \

19 16

/ \ / \

26 21 18 68

/ \ /

65 31 32

Other operations:

----------------

Finding any key other than the minimum one in a heap takes linear time

in the worst case. But if you have access to an element in the heap

through other means (maybe you have another data structure that keeps

pointers to the heap elements), we can do the following operations

in logarithmic time:

- decreaseKey: change the data as indicated and bubble up as needed

to restore the heap ordering property

- increaseKey: change the data as indicated and bubble down as needed

to restore the heap ordering property

- remove: one approach is to decrease the key of this node to the smallest

possible (highest priority) so it becomes the root and then do deleteMin.

How to create a heap out of a SET of items

-----------------------------------------

If you are given a sequence of N items one by one to put into a heap, you

can use the insert algorithm given above N times. The worst-case running

time will be

Theta(sum\_{i=1}^{N} (log i))

= Theta(log 1 + log 2 + ... + log(N/2) + log(N/2+1) + ... + log N)

>= Theta(log(N/2+1) + ... + log N) (only keep last N/2 terms)

> Theta(log(N/2) + ... + log(N/2) (replace each log argument with N/2)

= Theta((N/2)log(N/2))

= Theta(N log N).

However, if we are given all the N items initially, we can create a heap

faster, in Theta(N) time! Here's how:

1. Put the items into an array (vector) in arbitrary order (remember how

an array represents a complete binary tree, where the left child of index i

is index 2\*i, the right child is 2\*i+1, the parent is floor(i/2)

// process all the nodes going level by level starting just above the

// leaves and ending at the root; in each level go RIGHT to LEFT

2. for i = floor(N/2) down to 1 do

bubble down the item at index i it is at its proper place

(cf. deleteMin algorithm)

This algorithm is called BUILD-HEAP.

Example (cf. pp. 256-257):

150

/ \

80 40

/ \ / \

30 10 70 110

/ \ / \ / \ / \

100 20 90 60 50 120 140 130

Since N = 15 and floor(15/2) = 7, the indexing of the tree nodes causes

us to consider the nodes in order: 110, 70, 10, 30, 40, 80, 150.

CORRECTNESS OF BUILD-HEAP

Why do we end up with a heap?

Initially, every subtree consisting of a single leaf is a heap.

After we've processed all the nodes at height 1 (parents of leaves),

every subtree consisting of two sibling leaves and their parent is a heap.

Etc. until reaching the entire tree.

Worst-case running time: In the worst case, the bottom level of the

tree is full of nodes and each node we consider has to be bubbled down

all the way to a leaf. So the running time is proportional to the sum

of the heights of all the nodes in the tree. Let h be the height of

the tree.

height number of nodes at this height

--------------------------------------------

h 1 (root)

h-1 2

h-2 4

...

h-i 2^i

...

1 2^{h-1}

0 2^h (leaves)

Note that N = 2^{h+1}-1, which means h is Theta(log N).

Now calculate the sum of all the heights:

sum\_{i=0}^h 2^i\*(h-i)

This can be proved by induction to be 2^{h+1} - 1 - (h+1).

(The textbook gives a non-inductive proof.)

Plugging in to get the formula in terms of N instead h, we get:

2^{h+1} - 1 - (h+1) = N - h - 1 = N - Theta(log N) = Theta(N).

Applications of Priority Queues

------------------------------

#1: The selection problem: given a list of N elements and integer k,

find the k-th smallest element in the list.

1. Put all the elements of the list into a priority queue, where higher

priority means smaller.

2. Call deleteMin k times and return the last element obtained.

Running time with the heap implementation: Use build-heap for step 1,

which takes O(N) time. Step 2 takes O(k\*log N) time. Total is

O(N + k\*log N).

\*\* NOTE DEPENDENCE ON k AS WELL AS N! \*\*

If k is very small relative to N, say a constant (e.g., find the smallest

element or the 2nd smallest element), then the total time for selection

is O(N).

But if k is around N/2 (e.g., find the median element), then the total

time is O(N log N).

What if k is very large, say around N minus a constant (e.g., find the

largest or second largest element)? Then a better idea would be for the

priority queue to have higher priority mean larger (instead of smaller).

..............

#2: Discrete Event simulation: Suppose we want to study some kind of

real world scenario where events happen at various times, some of which

might cause other events to happen in the future. Typically, events

occur with some probability and the time between events can also be

described with a random distribution. In simple cases, behavior might

be analyzable through mathematics, but sometimes it's too complicated,

so instead, the behavior is simulated and observed.

The key observation is that the times we associate with events can be

anything we want as long as the sequence is correct. For instance,

if event A, occurring at 3:00 causes event B to occur at 4:00, there is

no reason why our simulation needs to wait an hour to simulate B.

The simulation can just go along simulating event after event in time order.

So why not just use a queue, where the events go at the back every time

they are generated? Because events might not be generated in the order

in which they are to be executed.

Example: Suppose we are simulating a networked computer system.

Events are:

- x sends a message to y

- z receives a message from w

A message send event is scheduled to occur every minute.

Once a message is sent, the receiving time is calculated using a random delay;

it's possible that message m1 is sent after message m2 but m1 is received

before m2.

So instead, put the events in a priority queue where the priority is the

smallest scheduled time.

========================================

Chapter 7: Sorting

-------------------

This chapter covers several sorting algorithms.

For simplicity, assume we are sorting integers (easy to extend to general

objects with keys that are comparable)

Sorting problem:

- input: An array (vector) A of n integers

- output: A permutation of the input so that A[i] <= A[i+1] for all i

Properties of sorting algorithms:

- running time

- how much extra space is needed (besides the input array); if only a

constant amount of extra space is needed, we say it is "in place"

- is it "stable" (do elements with the same keys stay in the same relative

order after sorting)?

- complexity of code

- complexity of analysis

https://www.toptal.com/developers/sorting-algorithms has a fun animation.

Insertion Sort:

--------------

Idea is construct increasingly larger prefixes of the array A that are

in sorted order by inserting the next element into its proper place in

the prefix.

Example:

Input: 10 4 6 1 3 7

10 | 4 6 1 3 7

4 < 10 so shift 10, front of array, insert 4

4 10 | 6 1 3 7

6 < 10 so shift 10, 6 > 4 so insert 6

4 6 10 | 1 3 7

1 < 10 so shift 10, 1 < 6 so shift 6, 1 < 4 so shift 4,

front of array, insert 1

1 4 6 10 | 3 7

3 < 10 so shift 10, 3 < 6 so shift 6, 3 < 4 so shift 4,

3 > 1 so insert 3

1 3 4 6 10 | 7

7 < 10 so shift 10, 7 > 6 so insert 7

1 3 4 6 7 10

Pseudocode:

for i = 1 to n-1 do

cur = A[i]

j = i-1

// start at right edge of sorted prefix and move left, looking for

// correct location of cur

while (j >= 0) // don't fall off left edge of A

and (A[j] > cur) // haven't yet found place for cur

A[j+1] = A[j] // shift right

j--

endwhile

A[j+1] = cur // put cur in its proper place

endfor

Running time: The worst case is when A is in reverse sorted order.

Then to place A[1], we have to shift 1 element,

to place A[2], we have to shift 2 elements,

...

to place A[n-1], we have to shift n-1 elements.

Total time is Theta(1 + 2 + ... + n-1 + n) = Theta(n^2).

Note that best case scenario is when A is in sorted order, and no shifting

has to be done and it takes Theta(n) time.

So maybe insertion sort has good "average" time (i.e., assume each

permutation of the input is equally likely)! Unfortunately, not...

Theorem: Any algorithm (including insertion sort) that sorts n elements

by exchanging adjacent elements requires Omega(n^2) time on average.

(Assume A has no duplicates.)

Proof: An "inversion" in an array A is an ordered pair (i,j) such that

i < j but A[i] > A[j].

Example: 10 4 6 1 3 7 has nine inversions:

(10,4), (10,6), (10,1), (10,3), (10,7),

(4,1), (4,3)

(6,1), (6,3)

1 3 4 6 7 10 has zero inversions.

Note that each swap (of adjacent elements) decreases the number of inversions

in the array by at most one.

Letting n be the size of A, we now show that the average number of

inversions is n(n-1)/4, which is Omega(n^2).

For an input array chosen at random, the probability that any pair of

elements form an inversion is 1/2. Since there are n(n-1)/2 pairs,

the average number of pairs that are inversions is half that, which is

n(n-1)/4.

QED

Note that this theorem is a lower bound that holds for an entire class

of algorithms, not just insertion sort.

(Skip Section 7.4 on Shellsort, a generalization of insertion sort where

items that are far away are swapped with each other, instead of just

neighbors. Using different sequences of gaps between the swaps gives

specific algorithms with different running times, including Theta(n^2),

Theta(n^{3/2}), Theta(n^{4/3}).)

Heapsort:

--------

Use a priority queue:

let PQ be an initially empty priority queue

for i = 0 to n-1 do

PQ.insert(A[i])

endfor

for i = 0 to n-1 do

A[i] = PQ.deleteMin()

endfor

Correctness should be clear.

Running time depends on how the priority queue is implemented.

With a binary heap, we have n inserts and n deleteMins, each O(log n),

so we get O(n log n) worst-case time.

Can we speed this up? We could use Build-heap to accomplish the first for

loop in Theta(n) time. But the second for loop will still take Theta(n log n)

time.

However, this approach wastes space: there is the input array A, and

then the array for the binary heap.

We can optimize this to only use A plus a constant amount of additional

space. We'll use a max-heap instead of the min-heap we've been using so

far, so the highest priority element is the largest element, and we have

operation deleteMax to get the root instead of deleteMin.

\*\* note that when we bubble down in a max-heap, we have to swap the

item with its LARGER child (not smaller) \*\*

Phase 1:

Do Build-heap on A

Phase 2:

Store the heap in a prefix of the array and the sequence of numbers

already sorted in the suffix of the array.

- Initially the heap encompasses the entire array.

- One by one, the numbers are removed from the prefix and inserted

into the suffix.

Example: (- indicates entry 0 of the array which is not used)

- 3 7 2 1 4 initial state of A (list of numbers to be sorted)

- 7 4 2 1 3 max-heap resulting from running build-heap

- 7 4 2 1 3 | heap is in A[1:5]; sorted list is empty

- 4 3 2 1 | 7 heap is in A[1:4]; sorted list is in A[5]

- 3 1 2 | 4 7 heap is in A[1:3]; sorted list is in A[4:5]

- 2 1 | 3 4 7 heap is in A[1:2]; sorted list is in A[3:5]

- 1 | 2 3 4 7 heap is in A[1]; sorted list is in A[2:5]

- | 1 2 3 4 7 heap is empty; sorted list is in A[1:5]

It's possible to prove that the average case running time of heapsort

(assuming every permutation of the input is equally likely) is Theta(n log n).

Mergesort:

---------

A very common and useful algorithm design technique is "Divide and Conquer",

which is based on recursion:

1. Divide: break the input problem up into smaller problem(s).

(If the input problem is sufficiently small, then just solve it

directly; this is the base of the recursion.)

2. Recur: Recursively solve the smaller subproblem(s) from Step 1.

3. Conquer: Take the solution(s) obtained in Step 2 and combine them

to get the solution to the original problem.

A famous example of divide and conquer is the sorting algorithm called

"mergesort":

Let S be the input sequence, with n elements.

1. Divide: if |S| < 2, then S is already sorted so return S.

otherwise, let S1 be a sequence containing the first half of S

and S2 be a sequence containing the second half of S.

(To handle the case when n is odd, put ceiling(n/2) elements

in S1 and floor(n/2) elements in S2.)

2. Recur: Recursively sort S1 and S2.

3. Conquer: Merge the sorted sequences S1 and S2 into a single sorted

sequence.

How to merge two sorted sequences? Assume array (vector) implementation.

Example execution of merge algorithm:

S1 = [2, 5, 8]

S2 = [3, 9, 10]

result is 2, 3, 5, 8, 9, 10.

(continuation of mergesort topic)

Algorithm merge(S1,S2,S):

input: sorted sequences S1 and S2 and empty sequence S

output: sorted sequence S containing elements from S1 and S2

i := 0 // "pointer" into S1

j := 0 // "pointer" into S2

while i < size(S1) and j < size(S2) do

if S1[i] <= S2[j] then

append S1[i] to end of S

i++

else

append S2[j] to end of S

j++

endif

endwhile

// either we've run through all of S1 or all of S2

while i < size(S1) // copy rest of S1 to end of S

append S1[i] to end of S

i++

endwhile

while j < size(S2) // copy rest of S2 to end of S

append S2[j] to end of S

j++

endwhile

Example execution of merge algorithm:

S1 = [2, 5, 8]

S2 = [3, 9, 10]

result is 2, 3, 5, 8, 9, 10.

Running time for merge:

- the body of each while loop takes O(1) time per iteration

- each iteration of any of the loops copies one element from either S1

or S2 into S

- each element in S1 and S2 is copied exactly once.

- Therefore overall time is O(size(S1)+size(S2)).

Back to mergesort:

Algorithm mergesort(S) returns T:

input: sequence S

output: sorted sequence T containing the elements of S

if size(S) <= 1 then return a copy of S // base case

let S1 and S2 be initially empty sequences

for i = 1 to ceiling(size(S)/2) do

append S[i] to end of S1

endfor

for i = ceiling(size(S)/2)+1 to size(S) do

append S[i] to end of S2

endfor

T1 := mergesort(S1)

T2 := mergesort(S2)

merge(T1,T2,T)

return T

Example execution of mergesort:

Suppose input is [85, 24, 63, 45, 17, 31, 96, 50]

< do example >

Running time of mergesort:

- Assume n, the size of the input sequence, is a power of 2, say n = 2^k,

so k = log\_2 n.

- Consider one recursive call, say, when the input size is m.

- The nonrecursive work done is in the divide step and the conquer step.

- The divide step copies the input array into S1 and S2: O(m) time.

- The conquer step merges T1 and T2: O(m) time since size(T1) + size(T2) = m.

So each recursive call takes O(m) time, excluding the time taken by its

own recursive calls.

There is one recursive call on an input of size n => n total time

2 recursive calls on inputs of size n/2 => n total time

4 recursive calls on inputs of size n/4 => n total time

...

k recursive calls on inputs of size n/(2^k) = 1 => n total time

Adding all this up is O(k\*n) = O(n log n) since k = log\_2 n.

We can also use a "recurrence" to describe the running time of mergesort.

Let T(n) be the running time of mergesort on an input of size n.

- T(n) = b if n <= 1 for some constant b

- T(n) = 2\*T(n/2) + c\*n if n > 1

because there are two recursive calls, each on an input of size n/2

and the nonrecursive work (splitting and merging) takes O(n) time

This is a called a recurrence equation since T is defined in terms of

T (albeit with a smaller argument). But a recurrence equation doesn't

give us an intuitive idea of behavior of the function. We'd prefer a

"closed form" for T(n).

Here is an ad-hoc way of "solving" this recurrence (getting a closed form

version). There are more general ways, e.g., the master theorem, covered

in CSCE 222 and CSCE 411.

Basically, just keep expanding until you have enough examples to guess

the pattern and then solve.

T(n) = 2\*T(n/2) + cn

= 2\*[2\*T(n/4) + c(n/2)] + cn plug into T(n/2)

= (2^2)\*T(n/4) + 2cn

= (2^2)[2\*T(n/8) + c(n/4)] + 2cn plug into T(n/4)

= (2^3)\*T(n/8) + 3cn

...

= (2^i)\*T(n/(2^i)) + i\*cn

...

= (2^k)\*T(n/(2^k)) + k\*cn k = log\_2 n, this is where we stop

Plugging in log\_2 n for k and simplifying, we get:

T(n) = n\*T(1) + (log\_2 n)\*cn

= n\*b + (log\_2 n)\*cn

= O(n log n).

Main drawback of the algorithm as presented: uses a lot of extra space,

is not "in-place". The textbook discusses some ways to optimize the space

usage.

Quicksort

--------

Quicksort is another recursive sorting algorithm.

Basic idea:

1. Choose an item x in the input, called the pivot

2. Partition all the items into

S1: all items < x

S2: all items = x

S3: all items > x

3. Recursively sort S1

4. Recursively sort S3

5. Concatenate S1 followed by S2 followed by S3

We also need a base case: in theory it can be a sequence of size 0 or 1

(no changes needed).

To have any kind of advantage over previous sorting algorithms, we

need to do these steps very carefully and efficiently. In particular:

\* choose the pivot carefully (ideally, get equal-sized subproblems)

\* do the partitioning efficiently (in-place)

There are numerous variations on these details.

Choosing the pivot:

#1: Pick first element in the input array.

+ simple and fast

- leads to very slow runtime in certain cases, including when

the input is in sorted or reverse sorted order

#2: Choose an element at random.

+ leads to fast runtime on average on EVERY input

- time-consuming to call a random number generator

#3: Median of 3: take the median of the first, last, and middle element

+ leads to fast runtime in many cases

+ simple and fast

The book uses median of 3, like this:

input:

- A, array of of numbers

- left, index of left end of segment of A to be partitioned

- right, index of right end of segment of A to be partitioned

center = (left + right)/2

if A[center] < A[left] then swap A[center] and A[left]

if A[right] < A[left] then swap A[left] and A[right]

if A[right] < A[center] then swap A[center] and A[right]

Example:

60 80 40 90 0 65 30 50 20 70 10 55

left center right

- don't swap 60 and 65

- swap 60 and 55:

55 80 40 90 0 65 30 50 20 70 10 60

left center right

- swap 60 and 65:

55 80 40 90 0 60 30 50 20 70 10 65

left center right

So A[center] = 60 is the pivot to be returned.

It will be convenient if we finish by swapping A[center] with A[right-1]

(get the pivot out of the way; later we will put it back):

55 80 40 90 0 10 30 50 20 70 60\* 65

left center right

Next we do the partitioning (continuing to follow textbook):

Use counters i and j to scan the array. Initially i = left+1 and j = right-2:

55 80 40 90 0 10 30 50 20 70 60\* 65

left i center j right

Move i right, skipping over all elements < 60.

Move j left, skipping over all elements > 60.

If i and j cross each other, then stop.

Otherwise swap A[i] and A[j] and repeat.

55 80 40 90 0 10 30 50 20 70 60\* 65

i j

Swap 80 and 20:

55 20 40 90 0 10 30 50 80 70 60\* 65

i j

Move i and j:

55 20 40 90 0 10 30 50 80 70 60\* 65

i j

Swap 90 and 50:

55 20 40 50 0 10 30 90 80 70 60\* 65

i j

Move i and j:

55 20 40 50 0 10 30 90 80 70 60\* 65

j i

i and j have crossed, so stop.

Final step: swap A[i] with pivot, which has been waiting patiently in

A[right-1]:

55 20 40 50 0 10 30 60\* 80 70 90 65

left j i right

Then the two recursive calls operate on A[left,i-1] and A[i+1,right].

(continuing quicksort partitioning)

Final step: swap A[i] with pivot, which has been waiting patiently in

A[right-1]:

55 20 40 50 0 10 30 60\* 80 70 90 65

left j i right

Then the two recursive calls operate on A[left,i-1] and A[i+1,right].

What to do if we have duplicate elements? The subtlety is whether

i should stop when it finds an element greater than the pivot or

when it finds an element greater than OR EQUAL to the pivot.

Similarly, should j stop when it finds an element less than the pivot or

when it finds an element less than OR EQUAL to the pivot.

It turns out that we'll get better performance due to more equal-sized

subproblems if we have both i and j NOT skip over elements equal to the

pivot, even they it may cause more swapping.

Example:

6 6 6 6 6 6 6 6

left i j right

6 6 6 6 6 6 6 6

left i j right

6 6 6 6 6 6 6 6

left i,j right

6 6 6 6 6 6 6 6

left j i right

Splitting this array segment into A[left,i-1] and A[i+1,right]

gives (almost) equal sized subproblems (4 and 3).

Note about base case for quicksort: In practice, it's better to switch

to insertion sort once the problem size is down to around 10.

Analysis:

--------

Describe the running time with a recurrence. Let T(N) be the time on

an input array of N elements.

T(N) = T(i) + T(N-i-1) + cN

where i is the size of the first subarray and N-i-1 is the size of the

second subarray. Why?

- We make two recursive calls, one on the first subarray, whose size is i,

and another on the second subarray, whose size is N-i-1

- The partitioning takes time linear in N

- Choosing the pivot takes constant time, so we can include it in

the constant c for the partitioning

Worst-case: when the two subarrays are maximally unequal in size,

so i = 0. This happens when the pivot is the smallest element.

T(N) = T(N-1) + cN

= T(N-2) + c(N-1) + cN

= T(N-3) + c(N-2) + c(N-1) + cN

...

= T(N-j) + c(N + N-1 + N-2 + ... + N-(j-1))

...

= T(1) + c\*Sum\_{i=2}^N i

= Theta(N^2)

Best-case: when the two subarrays are the same size. This happens

when the pivot is the median.

T(N) = 2\*T(N/2) + cN

This is exactly the same recurrence as for mergesort, so we have

T(N) = Theta(N log N)

Fact: "Average case" running time can be shown to be Theta(N log N),

where "average" means either

(i) every permutation of the inputs is equally likely or

(ii) the pivot is chosen at random

Linear Time Algorithms for Selection:

-------------------------------------

Quicksort can be modified to solve the selection problem (find the k-th

smallest element in a sequence S) in O(N) expected time:

QuickSelect(S,k):

1. if |S| = 1 then return the one element in S // base case

2. pick a pivot element v in S

3. partition S - {v} into S1 (elements less than v) and S2 (elements

greater than v)

4. if k <= |S1| then return QuickSelect(S1,k)

if k = 1 + |S1| then return v (the pivot is the k-th smallest)

else return QuickSelect(S2,k - |S1| - 1)

Example:

Suppose after partitioning around pivot 25, we have:

10 20 15 | 25 | 80 40 50

S1 = 10, 20, 15 and S2 = 80, 40, 50

If k = 2, we continue the search in S1, looking for the 2nd smallest element,

which will turn out to be 20.

If k = 4, we return 25.

If k = 5, we continue the search in S2, looking for the smallest element,

which will turn out to be 40.

Quickselect only makes one recursive call, instead of two as in Quicksort.

Worst-case running time: Similar situation as for Quicksort, when the

partitioning is maximally unequal, and we have to continue searching in

the larger partition. This gives Theta(N^2).

However, the \*average\* running time can be proved to be only Theta(N),

which beats quicksort's average time.

Note: There exists a variation of quickselect that ensures Theta(N) time

in the worst case! (That is, without using any randomizaton or any

assumptions on the input.) It has a complicated, but still linear time,

algorithm for choosing the pivot which ensures that the two partitions

are closely balanced in size.

General Lower Bound for Sorting

-------------------------------

The fastest worst-case algorithms we have for sorting so far are heapsort

and mergesort, at Theta(N log N). Can we do better?

Unfortunately, no, at least not if we restrict ourselves to "comparison-

based algorithms". These are algorithms that only compare the elements

being sorted but don't depend on the specific values of the elements.

All the sorting algorithms we've seen so far are comparison-based.

Theorem: Any sorting algorithm that uses only comparisons between

elements requires Omega(N log N) comparisons on some input of size N.

This implies that the worst-case running time is Omega(N log N).

The proof relies on the concept of a "decision tree", which is an abstraction

of how a SPECIFIC algorithm works on ALL INPUTS OF A SPECIFIC SIZE.

- each non-leaf decision tree node corresponds to the execution of a

comparison of two inputs (e.g., checking for <, >, =, <= or >=)

- each non-leaf node has 2 children, one for the case when the comparison

is true and one for the case when the comparison is false

- each leaf represents the correct sorted order for the path starting at

the root and ending at that leaf

Example for insertion sort for N = 3: Suppose input is [a0,a1,a2].

Comparisons refer to actual elements, not to their array locations.

a0 ≤ a1?

Y/ \N

a1 ≤ a2? a0 ≤ a2?

Y/ \N Y/ \N

a0a1a2 a0 ≤ a2? a1a0a2 a1 ≤ a2?

Y/ \N Y/ \N

a0a2a1 a2a0a1 a1a2a0 a2a1a0

(continuing lower bound for sortin)

In a decision tree, there must be at least one leaf for each

permutation of the input, otherwise there would be a situation that is

not correctly sorted.

Number of permutations of N keys is N!.

Idea: Since there must be a lot of leaves in the decision tree, but each

node has only two children, the tree cannot be too shallow.

So we get a lower bound on the number of comparisons that must be done

on some input, which is a lower bound on the total time required for that

input.

Details for proof of main theorem:

Since the decision tree has at least N! leaves, its depth must be at least

log\_2 (N!).

(Why? Because maximum number of leaves in a binary tree with depth d is 2^d.)

log\_2 (N!) = log\_2 (N\*(N-1)\*(N-2)\*...\*2\*1)

= log\_2 N + log\_2(N-1) + log\_2(N-2) + ... + log\_2(2) + log\_1(1)

>= (N/2)\*log\_2(N/2) (same argument we saw in a previous lecture)

= Omega(N log N).

QED

Decision-tree-based proofs can be used to prove additional lower bounds:

\* N-1 comparisons are necessary to find the smallest item.

\* N + ceil(log N) - 2 comparisons are necessary to find the two smallest

items.

\* ceil(3N/2) - O(log N) comparisons are necessary to find the median.

Linear Time Sorting Algorithms

------------------------------

To beat the N log N lower bound for sorting, we have to do something

different with the elements besides just comparing their relative values.

We'll see two ways to do this, bucket sort and radix sort.

Bucket Sort:

-----------

Suppose the N elements to be sorted are nonnegative integers and each one

is less than some value M.

input: array A of N integers to be sorted

array count, size M, initially every element is 0

for i = 0 to N-1 do

increment count[x] by 1, where x = A[i]

endfor

output consists of count[0] copies of 0, followed by count[1] copies of 1,

... followed by count[M-1] copies of M-1

(In this algorithm, we have one bucket for each possible input value.

Bucket sort can also be used when the input is assumed to be distributed

uniformly at random in [0,1), and the interval is divided into N equal-sized

subintervals; not very many elements are expected to be in the same

bucket and they can be sorted very quickly, resulting in constant expected

time. Cf. Cormen et al. textbook.)

Let's think more about how we are able to get around the N log N lower bound.

\* We have extra information about the nature of the input!

Nonnegative integers less than M.

\* This basically lets us do an M-way comparison in constant time, which is

NOT the model used by the lower bound (which assumed 2-way comparison

can be done in constant time)

Running time and space are both O(N + M). If M is O(N), then this is O(N).

But what if M is much much larger than N? See next algorithm...

Radix Sort:

----------

Suppose we have 10 numbers in the range 0 to 999.

Using basic bucket sort would take time proportional to 1000 but there are only

10 numbers.

Instead, let's do bucket sort on each DIGIT, starting with least significant.

Example:

064 008 216 512 027 729 000 001 343 125

After bucket sort on ones digits (N = 10, M = 10):

000 001 512 343 064 125 216 027 008 729

^ ^ ^ ^ ^ ^ ^ ^ ^ ^

After bucket sort on tens digits (N = 10, M = 10): Note use of stability!

000 001 008 512 216 125 027 729 343 064

^ ^ ^ ^ ^ ^ ^ ^ ^ ^

After bucket sort on hundreds digits (N = 10, M = 10): Note use of stability!

000 001 008 027 064 125 216 343 512 729

^ ^ ^ ^ ^ ^ ^ ^ ^ ^

In this example we had three passes, doing bucket sort in each pass.

Note that the range of numbers is 0 to 10^3 - 1.

We can generalize this example to sorting N numbers in the range

0 to b^p - 1:

- p passes, doing bucket sort in each pass

- i-th pass focuses on i-th least significant "digit" of the number

when considering the base b representation

Running time is O(p\*(N+b). For appropriate choices of p and b, this

can be much faster than O(N + b^p).

Why is this called "radix" sort? "Radix" means the base of a number system;

the algorithm represents each number in base b, then uses the resulting

"digits" in the bucket sort passes.

\*\* It's essential for the correctness of radix sort that the "subroutine"

sort used in each pass be stable, as bucket sort is.

Application of radix sort:

-------------------------

Sort a set of N ASCII strings, all with the same length L.

Let each character by a "digit", do bucket sort on each position from

right to left. L passes, each one takes O(N + 256), so total is

O(LN), since 256 is a constant.

What to do if the strings have different lengths?

1. Sort the strings according to length. Assuming even the longest string

isn't terribly long relative to N, the number of strings, this can be done

with bucket sort in O(N) time.

2. For each string length, use radix sort.

Running time is linear in the total number of characters in all the strings.

Radix Sort with Counting Sort:

-----------------------------

A variation of radix sort uses a different sorting algorithm for each

digit: "counting sort" instead of bucket sort.

Counting sort works like this:

As before, we assume that we have N numbers to be sorted and each

number is in the range 0 to M-1.

Idea: for each number x, determine how many elements are less than x

(e.g., 17) and then place x in the correct position in the output

(e.g., the 18th position).

If there can be duplicate inputs, we need to modify this idea.

input: array A[1..N] of integers, each less than M \*\*start indexing at 1\*\*

output: array B[1..N], sorted permutation of A \*\*start indexing at 1\*\*

initialize array count[0..M-1] to all 0's \*\*start indexing at 0\*\*

for i = 1 to N do

count[A[i]]++

endfor

// now count[i] contains the number of elements equal to i

for i = 1 to M-1 do

count[i] = count[i] + count[i-1]

endfor

// now count[i] contains the number of elements <= i

for i = N downto 1 do

x = A[i]

B[count[x]] = x // put x in its proper place

count[x]-- // handles possibility of multiple copies of x

endfor

Example: M = 6, N = 8, A = [2, 5, 3, 0, 2, 3, 0, 3]

after first for loop:

count = [2, 0, 2, 3, 0, 1]

after second for loop:

count = [2, 2, 4, 7, 7, 8]

execution of third for loop:

i = N = 8:

x = A[N] = 3

count[3] = 7

put 3 in B[7]

decrement count[3] by 1

B = [-, -, -, -, -, -, 3, -]

count = [2, 2, 4, 6, 7, 8]

i = N-1 = 7:

x = A[N-1] = 0

count[0] = 2

put 0 in B[2]

decrement count[0] by 1

B = [-, 0, -, -, -, -, 3, -]

count = [1, 2, 4, 6, 7, 8]

i = N-2 = 6:

x = A[N-2] = 3

count[3] = 6

put 3 in B[6]

decrement count[3] by 1

B = [-, 0, -, -, -, 3, 3, -]

count = [1, 2, 4, 5, 7, 8]

etc. At the end,

B = [0, 0, 2, 2, 3, 3, 3, 5]

Running Time of Counting Sort: O(N+M), which is O(N) if M is O(N).

Same as for bucket sort. It is also stable.

Summary:

-------

"Sorting is one of the oldest and most well-studied problems in computing."

(textbook)

There are many many variations that have been studied. For instance,

the textbook presents algorithms for "external sorting", which means that

the input to be sorted is so big that it will not all fit in memory,

and instead is stored on disk -- you have to read in subsets of the

input into memory where it can be directly addressed.

The choice of the best sorting algorithm depends on

- desired properties

- size of input

- underlying environment

Chapter 9: Graph Algorithms

A graph is a way of representing relationships that exist between pairs of

objects. Each object is a "vertex" of the graph, and relationships are

indicated by "edges" between pairs of vertices.

Graphs are used almost everywhere: social networks, communication networks,

transportation networks, biological applications,...

Terminology:

------------

- graph G = (V,E), set V of vertices (or "nodes"), set E of edges where an

edge is a pair of vertices

- directed graph: edges are ORDERED pairs, (u,v)

Edges model asymmetric relationships between vertices, such as parent-child

relationship or a software call graph.

- undirected graph: edges are UNORDERED pairs, technically {u,v}

but often written (u,v)

Edges model symmetric relationships between vertices, such as two

people being friends with each other, or a 2-way road between two towns.

- weighted graph: a number, weight or cost, is associated with each edge

- loop: edge (v,v); mostly we'll consider loopless graphs

- vertex w is adjacent to vertex v: (v,w) is an edge

(sometimes w is said to be v's "neighbor")

- path: sequence of vertices w\_1, w\_2, w\_3, ... w\_N such that each

(w\_i,w\_{i+1}) is an edge

- length of path: number of EDGES in the path (e.g., N-1)

path from w to itself has length 0

- simple path: all vertices are distinct (except possibly first and last)

- cycle: path of length >= 1 with w\_1 = w\_N

- DAG: directed acyclic graph (has no cycles)

- if there is a path from every vertex to every other vertex, we say:

graph is CONNECTED in the undirected case

graph is STRONGLY CONNECTED in the directed case

- complete graph: there is an edge between every pair of vertices.

This is the graph with the maximum number of edges:

- |V|\*(|V|-1)/2 for undirected graph

- |V|\*(|V|-1) for directed graph

- both are Theta(|V|^2)

How to Measure Time and Space for Graph Algorithms:

-------------------------------------------------

Use both |V| (number of vertices) and |E| (number of edges).

|E| is never more than Theta(|V|^2) but it can be much smaller for some graphs.

So we want to be as general as possible.

Adjacency Matrix Representation of a graph:

------------------------------------------

Use a 2-dimensional array A, with one column for each vertex and one row

for each vertex.

Set A[u,v] to true if (u,v) is an edge, otherwise false.

Example: (\* means arrowhead)

1 -------\* 2

/ \ / \

/ \ / \

\* \* \* \*

3 \*------ 4 -------\* 5

\ / \ /

\ / \ /

\* \* \* \*

6 \*------- 7

1 2 3 4 5 6 7

1 F T T T F F F

2 F F F T T F F

3 F F F F F T F

4 F F T F T T T

5 F F F F F F T

6 F F F F F F F

7 F F F F F T F

Note: If the graph is undirected, we don't need half of the array, since

A[u,v] is the same as A[v,u].

Note: If the graph is weighted, then A[u,v] can be used to hold the weight

of the edge (u,v).

Advantages:

\* very simple

\* can determine whether two vertices are adjacent in constant time

Disadvantages:

\* lots of space, Theta(|V|^2). If graph is "dense" (meaning |E| = Theta(|V|^2),

this is fine, but if graph is not dense ("sparse"), this is wasteful.

\* takes Theta(|V|) time to find all neighbors of a vertex, even if there are

very few

Adjacency List Representation of a Graph:

----------------------------------------

First cut idea:

- have an array (vector) with one entry for each vertex

- the array entry at index i is (a pointer to) a linked list of all the

vertices adjacent to i

Example for graph drawn above:

1 : 2 -> 3 -> 4

2 : 4 -> 5

3 : 6

4 : 3 -> 6 -> 7

5 : 4 -> 7

6 : (empty)

7 : 6

Note: If graph is undirected, then there are two entries for edge {u,v},

one for v in u's linked list, and one for u in v's linked list.

Advantages:

\* linear space, O(|V| + |E|)

\* can find all neighbors of a vertex in time linear in the number of

neighbors

Disadvantages:

\* more complicated to code

\* takes Theta(degree) time to determine whether two vertices are adjacent

(the "degree" of a graph is the maximum number of neighbors that any

vertex has)

Variations on the idea:

\* Use a different data structure than array for all the vertices; for

example, a map

\* Use a different data structure than a linked list for holding adjacent

vertices; for example, a map

Topological Sort:

----------------

This is a concept that is defined only for directed acyclic graphs (DAGs).

Definition: Given a DAG, a TOPOLOGICAL SORT is an ordering of the vertices

such that if (u,v) is an edge, then u comes before v in the ordering.

Applications include

- scheduling (what order to take your classes in while respecting prerequisites)

- PERT charts (project timeline)

- makefiles (what order to compile files in)

- spreadsheets (which cells depend on which other cells)

etc.

Example: (same graph from above, which happens to be a DAG)

1 -------\* 2

/ \ / \

/ \ / \

\* \* \* \*

3 \*------ 4 -------\* 5

\ / \ /

\ / \ /

\* \* \* \*

6 \*------- 7

A topological sort is: 1, 2, 4, 3, 5, 7, 6.

Not necessarily unique; in this example, we could swap 3 and 5.

Topological Sort Algorithm #1:

-----------------------------

Idea: repeatedly find a vertex with no incoming edges, print it and

remove it, until no more vertices.

More details: Suppose graph is input into an adjacency list and the

"in-degree" (number of incoming edges) is stored for each vertex.

for i = 1 to |V| do

// find vertex with no incoming edges

for v = 1 to |V| do

if indegree(v) = 0 then

output v // next vertex in the topological sort ordering

break // break out of inner for loop; remember v

endif

endfor

// update indegrees for v's neighbors

for each w in v's adjacency list do

indegree(w)--

endfor

endfor

Running time:

- Outer for loop is done |V| times

- First inner for loop is done at most |V| times during each iteration of

outer for loop, for a total time of Theta(|V|^2)

- Second inner for loop is done different number of times, depending on

how many neighbors v has:

outdegree(v1) + outdegree(v2) + ... + outdegree(vn)

for a TOTAL time of Theta(|E|) (not per iteration)

Total is Theta(|V|^2 + |E|) = Theta(|V|^2) since |E| is O(|V|^2).

Can we do better? Yes!

Topological Sort Algorithm #2:

-----------------------------

input: graph G = (V,E) using adjacency list representation

let Q be an empty FIFO queue

for each v in V do

if indegree(v) = 0 then Q.enqueue(v)

endfor

while Q is not empty do

v := Q.dequeue()

output v // the order of dequeueing is the topological sort order

for each w adjacent to v do

decrement indegree(w)

if indegree(w) = 0 then Q.enqueue(w)

endfor

endwhile

Example: run this algorithm on the input graph from above

(note that this is not exactly the same as the graph in the textbook).

Work it out yourself. Start like this:

vertex indegree

-------------------

1 0

2 1

3 2

4 2

5 2

6 3

7 2

Q: 1

Running time:

\* Convince yourself that the indegrees of all the vertices can be

computed in O(|V|+|E|) time, assuming adjacency list representation.

\* Recall that queue operations take constant time.

\* First for loop takes O(|V|) time.

\* While loop is done |V| times since each vertex is enqueued once and

dequeued once.

\* The dequeue and the output take constant time per iteration, so O(|V|)

time altogether.

\* the for loop takes time proportional to the number of neighbors of v;

this can be different for different vertices, but altogether it is

O(|E|) time.

\* Total is O(|V|+|E|).

Single-Source Shortest-Path Problem (SSSP):

-------------------------------------------

Given a graph G and two vertices v (the source) and w in G, find the "shortest"

path from v to w.

What does "shortest" mean?

- If the graph is unweighted, then it means the path from v to w with the

minimum number of edges

- If the graph is weighted, then it means the path from v to w such that the

sum of the weights on all the edges of the path is minimum

Applications: fastest driving route, cheapest plane ticket, etc.

In fact, it's usually just as easy to find the shortest paths from v to ALL the

other vertices.

SSSP Algorithm for Unweighted Graphs:

------------------------------------

This algorithm is also known as "breadth-first search" (BFS).

It can be viewed as a way to visit all the vertices of the graph.

Idea: Find all the vertices at distance 0 from v, then all the vertices

at distance 1 from v, then all the vertices at distance 2 from v, etc.

BFS:

input: unweighted directed graph G = (V,E), and source vertex s in V

for each v in V do

dist(v) := infinity

endfor

dist(s) := 0

let Q be an initially empty queue

Q.enqueue(s)

while (Q is not empty) do

v := Q.dequeue()

for each vertex w adjacent to v do

if dist(w) = infinity then // w is not yet visited

dist(w) := dist(v) + 1 // level, or distance from s

parent(w) := v // to keep track of the actual path

Q.enqueue(w)

endif

endfor

endwhile

Output is the collection of dist and parent variables.

Example: (not exactly the same as the graph used before)

1 -------\* 2

\* \ / \

/ \ / \

/ \* \* \*

3 \*------ 4 -------\* 5 source is vertex 3

\ / \ /

\ / \ /

\* \* \* \*

6 \*------- 7

vertex dist parent

1 infty nil

2 infty nil

3 0 3 (indicates 3 is the source)

4 infty nil

5 infty nil

6 infty nil

7 infty nil

Q: 3

iterations of while loop:

- 3 is dequeued, dist is set to 1 and parent is set to 3 for 1 and 6,

Q: 1, 6

- 1 is dequeued, dist is set to 2 and parent is set to 1 for 2 and 4,

Q: 6, 2, 4

- 6 is dequeued, 6 has no outgoing neighbors

Q: 2, 4

- 2 is dequeued, dist is set to 3 and parent is set to 2 for 5

Q: 4, 5

- 4 is dequeued, dist is set to 4 and parent is set to 4 for 7

Q: 5, 7

- 5 is dequeued, 5 has no unvisited outgoing neighbors

- 7 is dequeued, 7 has no unvisited outgoing neighbors

Correctness: One can use induction to prove that after iteration i of the

while loop, all vertices at distance 1 from the source have the correct values

for dist and parent. Note that if a vertex v is unreachable from the source,

it has distance infinity at the end, which is appropriate.

What about for undirected graphs? The algorithm works the same way, except

we don't have to worry about the neighbors being "outgoing". We could still

have unreachable vertices, if the graph is disconnected.

Definition: If G is a connected UNdirected graph, a SPANNING TREE of G

is a subgraph that contains all the vertices of G, is connected and

has no cycles. (The definition can be extended to directed graphs,

but we won't consider that in this class.)

Fact: The BFS algorithm's parent variables define a spanning tree of the

input graph (if it is connected), called a "breadth-first spanning tree".

Running Time: If G is represented using an adjacency list, the running time

is O(|V|+|E|), by essentially the argument as for the topological sort

algorithm.

Note use of queue ADT!

Depth-first search:

------------------

We'll jump ahead to to another way of searching all the vertices in a graph,

called depth-first search (DFS), to give you the background for PA 5.

Then we'll come back to SSSP for weighted graphs.

\*\* DFS does NOT give you shortest path info! We'll see applications for DFS

in a bit. \*\*

Idea is to go as far away from the source as possible in the search; once we

get stuck then we backtrack as little as possible. That is, go as deep in

the graph as possible at all times.

The DFS algorithm is recursive, top level call has parameter s (the source).

Initially, for each vertex v:

- visited(v) := false

- parent (v) := nil (somehow indicate that s isn't supposed to have a parent)

DFS(v):

visited(v) := true

for each vertex w adjacent to v do

if visited(w) is false then

parent(w) := w

DFS(w)

endif

endfor

Example on an UNdirected graph, assume neighbors are considered in alphabetical

order:

A

/ | \

/ | \

/ | \

B -- D E Suppose we start at A

\ | /

\ | /

\ | /

\ C

DFS(A):

visited(A) := true

parent(B) := A

DFS(B):

visited(B) := true

parent(C) := B

DFS(C):

visited(C) := true

parent(D) := C

DFS(D):

visited(D) := true

(D has no unvisited neighbors)

end of recursive call DFS(D), pop back to recursive call DFS(C)

parent(E) := C

DFS(E):

visited(E) := true

(E has no unvisited neighbors)

end of recursive call DFS(E), pop back to recursive call DFS(C)

end of recursive call DFS(C), pop back to recursive call DFS(B)

end of recursive call DFS(B), pop back to recursive call DFS(A)

end of recursive call DFS(A), done

DFS Notes (continued):

Running Time: If the graph is represented using the adjacency list representation,

the DFS algorithmm takes O(|V|+|E|) time. The reason is that there is one

recursive call for each vertex. The amount of (non-recursive) work done during

each recursive call differs, depending on how many neighbors the current vertex

has. As we've seen with previous graph algorithms, though, the total amount

of work done in all the for loops is proportional to the total number of edges.

Fact: The DFS algorithm's parent variables define a spanning tree of the

input graph (if it is connected), called a "depth-first spanning tree".

Now let's see what happens when we run the DFS algorithm on a directed graph.

1 -------\* 2

\* \ / \

/ \ / \

/ \* \* \*

3 \*------ 4 -------\* 5 start at vertex 3, consider neighbors in

\ / \ / numeric order

\ / \ /

\* \* \* \*

6 \*------- 7

We end up with the spanning tree 3 -> 1 -> 2 -> 4 -> 5 -> 7 -> 6 (it's actually

a chain).

What happens if we start at 2? We get spanning tree with edges

(2,4), (4,3), (3,1), (3,6), (4,5), (5,7).

What happens if we start at 5? We get 5 -> 7 -> 6. So we weren't able to

reach all the vertices.

What happens if we start at 6? We get nowhere!

The problem is that this graph is not strongly connected, so depending on where

we start, we might not be able to reach all the vertices.

This is analogous to the situation in undirected graphs that are not connected:

we cannot get to the vertices that are not reachable from the starting vertex.

To get around this problem, we can put a non-recursive "wrapper" around the

recursive DFS, which can also take care of the initialization of the visited

and parent variables:

for each vertex v in V do

visited(v) := false

parent(v) := nil

endfor

for vertex v = 1 through |V| do

if visited(v) is false then

parent(v) := v // mark v as a root of a tree by being its own parent

call recursive DFS(v) // same pseudocode as above

endif

endfor

Fact: Using this wrapper ensures that every vertex is eventually visited.

The parent variables form a "forest", collection of trees. Altogether this is

called a "spanning forest".

Applications of DFS:

-------------------

For undirected graphs:

- used in an algorithm to identify "articulation points" of a graph,

vertices whose removal would disconnect the graph. Finding such points is

useful when designing and analyzing computer networks and transportation

networks.

- used in an algorithm to find an "Euler circuit" in a graph if it has one,

a path in the graph that uses every edge exactly once. Useful for planning

routes for mail carriers, garbage pickup, etc.

For directed graphs:

- used in an algorithm to find the "strongly connected components" of a graph,

partitions of the graph into the minimum number of subgraphs, each of which

is strongly connected. Useful preprocessing step before applying some

algorithms that only work on strongly connected graphs; used in algorithms

for formal verification of program correctness (details beyond scope of this

class).

Because DFS is so fast (linear time), it helps in finding efficient algorithms

for the problems above.

(Come back to these applications if time)

Returning to SSSP problem, but for weighted graphs:

--------------------------------------------------

Famous algorithm due to Dijkstra, a Dutch computer scientist who came up

the algorithm in 1956. It took him 20 minutes while sitting at a cafe without

pencil or paper!

Restriction: NO NEGATIVE WEIGHTS!

Dijkstra's algorithm uses ideas similar to those in the BFS algorithm (for

the unweighted case).

Dijkstra:

input: weighted directed graph G = (V,E,w) with no negative edge weights,

and source vertex s in V

for each v in V do

known(v) := false // like "visited"

dist(v) := infinity

parent(v) := nil

endfor

dist(s) := 0

while (set of unknown vertices is not empty) do

v := unknown vertex with smallest distance

known(v) := true

for each vertex w adjacent to v do

if known(w) = false then // w is not yet visited

if (dist(v) + wt(v,w) < dist(w)) then // see if v gives w a shorter path

dist(w) := dist(v) + wt(v,w)

parent(w) := v // to keep track of the actual path

endif

endif

endfor

endwhile

Example:

2

A -------\* B source is A, consider neighbors

\* \ / \ in alphabetical order

4 / \ 1 3 / \ 10

/ 2 \* \* 2 \*

C \*------ D -------\* E

\ / \ /

5 \ 8 / \ 4 / 6

\* \* \* \*

F \*------- G

1

vertex known dist parent

A F 0 -

B F % - % means infinity

C F % -

D F % -

E F % -

F F % -

G F % -

iteration 1:

- A is unknown vertex with smallest distance. visited(A) := true

- update A's (outgoing) neighbors:

dist(B) := 0+2, parent(B) := A

dist(D) := 0+1, parent (D) := B

iteration 2:

- D is unknown vertex with smallest distance. visited(D) := true

- update D's (outgoing) neighbors:

dist(C) := 1+2, parent(C) := D

dist(E) := 1+2, parent(E) := D

dist(F) := 1+8, parent(F) := D

dist(G) := 1+4, parent(D) := D

iteration 3:

- B is unknown vertex with smallest distance. visited(B) := true

- update B's (outgoing) neighbors:

D is already known.

E is unknown but cost of going through B is 2+10, which is worse than 3

so E is not adjusted

iteration 4:

- E is unknown vertex with smallest distance. visited(E) := true

- update E's (outgoing) neighbors:

G is unknown but cost of going through E is 3+6, which is worse than 5

so G is not adjusted

iteration 5:

- C is unknown vertex with smallest distance. visited(C) := true

- update C's (outgoing) neighbors:

dist(F) := 3+5 (better than 9), parent(F) := C

iteration 6:

- G is unknown vertex with smallest distance. visited(G) := true

- update G's (outgoing) neighbors:

dist(F) := 5+1 (better than 8), parent(F) := G

iteration 7:

- F is unknown vertex with smallest distance. visited(F) := true

- update F's (outgoing) neighbors) - doesn't have any

Output:

vertex known dist parent

A T 0 A (or something to indicate A is root/source)

B T 2 A

C T 3 D

D T 1 A

E T 3 D

F T 6 F

G T 5 D

Correctness: Can be proved by induction on the iterations of the while loop

that the distance and parent variables for all the known vertices are correct.

Depends crucially on the assumption of no negative weights.

Running time of Dijkstra's algorithm: Depends on how the set of unknown

vertices and their distances is manipulated.

Simple approach: Keep an array with the data and scan through it to find

unknown vertex with smallest distance.

- Initialization: O(|V|)

- while loop is done |V| times since each iteration reduces the number of

unknown vertices by 1

- finding unknown vertex with smallest distance takes O(|V|) time (scan

the array), so total time is O(|V|^2)

- in each iteration of while loop, the for loop is done once for each

neighbor and the body of the for loop takes O(1) time, so total time is O(|E|).

Grand total is O(|V|^2 + |E|) = O(|V|^2).

For dense graphs, this is the best you could have and has virtue of simplicity.

But if graph is sparse, we can do better by keeping the unknown vertices in

a priority queue, using the distance as the key. We can rephrase the algorithm as:

let Q be a min priority queue using dist as the key

for each v in V do

if v = s then dist(v) := 0 else dist(v) := infinity

Q.insert(v)

parent(v) := nil

endfor

while (Q is not empty) do

v := Q.deleteMin()

for each vertex w adjacent to v do

if (dist(v) + wt(v,w) < dist(w)) then // see if v gives w a shorter path

dist(w) := dist(v) + wt(v,w)

Q.decrease-key(w,dist(w))

parent(w) := v // to keep track of the actual path

endif

endfor

endwhile

We do |V| inserts on the priority queue, |V| deleteMins on the priority queue,

and |E| decreaseKey's on the priority queue. Using a binary heap implementation

of the priority queue, each of these operations takes O(log|V|) time, for

a total of O(|V|\*log|V| + |E|\*log|V|) = O(|E|\*log|V|), since in a connected

graph, the number of edges is at least the number of vertices minus 1.

What if there are negative weights? If the graph has a negative weight cycle,

then the shortest path problem is not well-defined. The Bellman-Ford algorithm,

which uses some similar ideas to Dijkstra's, will find the shortest paths if

there is no negative weight cycle, and will let you know if there is a negative

weight cycle. Drawback is that it takes O(|V|\*|E|) time.

What if you want to solve the all-pairs shortest path (APSP) problem?

One solution is to repeat Dijkstra's algorithm once with each vertex as the

source: running time is O(|V|\*|E|\*log|V|).

There is another algorithm, using adjacency MATRIX representation, due

to Floyd (and Warshall and...), that r uns in O(|V|^3) time, which is simpler

and is faster for dense graphs.

Minimum Spanning Trees (MST)

----------------------------

Recall the definition from earlier of a spanning tree for an undirected, and

unweighted, graph. For an undirected graph that does have weights, sometimes

we want to find the spanning tree with the minimum total weight (add the weights

on all the edges of the tree).

Example:

1

A ------- B

G: | |

4 | | 2

| |

D ------- C

3

G has four spanning trees, each with a different weight. The one with minimum

weight, 6, is {(A,B),(B,C),(C,D)}.

Uses of MSTs:

- cheapest way to connect objects when you are constrained to use a subset of

given connections, e.g., provide utilities to all the houses in a neighborhood

but the utility lines have to along the roads.

- data clustering: complete graph whose vertices are points in the

plane (or space of any dimension), edge weights are Euclidean

distances between the points. Compute an MST, discard the k-1 most

expensive edges and the resulting connecting components form k

clusters of "nearby" points.

- efficient approximation algorithm for the traveling salesman problem

Prim's MST Algorithm:

--------------------

Intuition: start at any vertex and grow the tree around that vertex. At

all times, the edges we have chosen form a tree, eventually the tree reaches

all the vertices. The edge to be chosen at each step is the minimum weight

outgoing edge, that is, an edge with one endpoint in the current tree and

the other endpoint not yet in the tree.

This algorithm is very similar to Dijkstra's, but the MST problem is not the

same as the SSSP problem. For one thing, there is no notion of a source vertex

for MST. Refer back to the graph above and note that the MST is NOT a shortest

path tree when considering D as the source.

Example:

2

A -------- B

/ \ / \

4 / \ 1 3 / \ 10

/ 2 \ / 7 \

C ------- D -------- E

\ / \ /

5 \ 8 / \ 4 / 6

\ / \ /

F ------- G

1

Suppose we start at A. We'll add edges in this order:

(A,D) since weight 1 is the smallest among all outgoing edges of the current

tree, which consists just of vertex A.

Next we have two choices, (D,C) and (A,B) both have weight 2, which is smallest

among all edges that are outgoing from the tree.

Break the tie by choosing (A,B).

(D,C)

(D,G)

(G,F)

(G,E)

Prim's MST:

input: weighted connected undirected graph G = (V,E,w)

for each v in V do

dist(v) := infinity // smallest weight of any edge from v to current tree

parent(v) := nil // ultimately, (v,parent(v)) will be an edge in the MST

known(v) := false // like "visited"

endfor

choose any v in V

dist(v) := 0

while (set of unknown vertices is not empty) do

v := unknown vertex with smallest dist variable

known(v) := true

for each vertex w adjacent to v do

if known(w) = false then // w is not yet visited

if (wt(v,w) < dist(w)) then // see if v gives w a shorter path

dist(w) := wt(v,w)

parent(w) := v // to keep track of the edges

endif

endif

endfor

endwhile

output: all edges of the form (v,parent(v)), v in V // edges of the MST

Very similar to Dijkstra's: key difference is in the innermost updating of

dist. For MST,

"if (wt(v,w) < dist(w)) then dist(w) := wt(v,w)"

while for SSSP,

"if (dist(v) + wt(v,w) < dist(w)) then dist(w) := dist(v) + wt(v,w)"

Correctness can be proved by showing inductively that after the i-th iteration

of the while loop, the current tree is a subtree of some MST of the input graph.

Running time analysis and improvement using a priority queue is the same

as for Dijkstra's:

- Simple approach of keeping an array and scanning through it to find unknown

vertex with smallest distance gives O(|V|^2) time, which is good for dense

graphs.

- For sparse graphs, keep the unknown vertices in a priority queue, using the

distance as the key. If the priority queue is implemented with a binary heap,

the run time is O(|E|\*log|V|).

Kruskal's MST Algorithm:

-----------------------

A different approach to finding an MST of a graph is to add edges in increasing

order of weights, as long as adding the current edge does not cause a cycle.

Once |V|-1 edges have been added, stop.

Example:

2

A -------- B

/ \ / \

4 / \ 1 3 / \ 10

/ 2 \ / 7 \

C ------- D -------- E

\ / \ /

5 \ 8 / \ 4 / 6

\ / \ /

F ------- G

1

Since |V| = 7, we stop after including 6 edges.

edge weight include?

(A,D) 1 Y

(F,G) 1 Y

(A,B) 2 Y

(C,D) 2 Y

(B,D) 3 N

(A,C) 4 N

(D,G) 4 Y

(C,F) 5 N

(E,G) 6 Y

Obviously, we need to sort the edges, which will take O(|E|\*log|E|) time,

unless we have extra information about the edge weights and can use a

non-comparison-based sort.

How do we test whether adding the next edge in the sorted order causes a cycle?

I'll let you all think about ways to do that. It turns out there is a clever

data structure (which we won't cover) called "disjoint sets" that can do

this even faster than the time taken for (comparison-based) sorting of the

edges by weight. So the end result is O(|E|\*log|E|), which equals O(|E|\*log|V|)

since |E| is O(|V|^2), asymptotically same as Prim's.