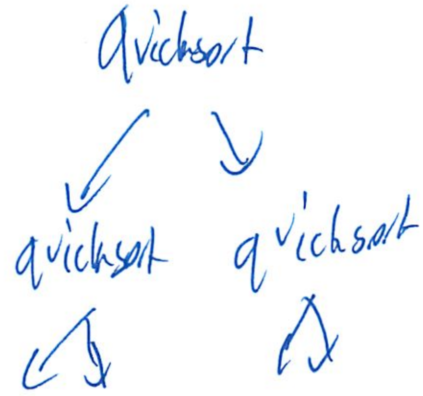
6.006 Quiz 2 Review

4/24

Sorting

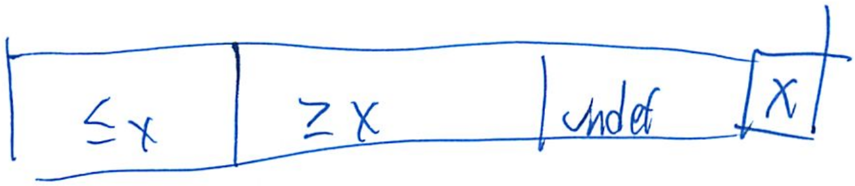
Quicksort

- pivot – usually last, but middle is better
- lesser + greater
  - before pivot, after pivot
- in place or not
- best pratical choice often
- divide + conquer



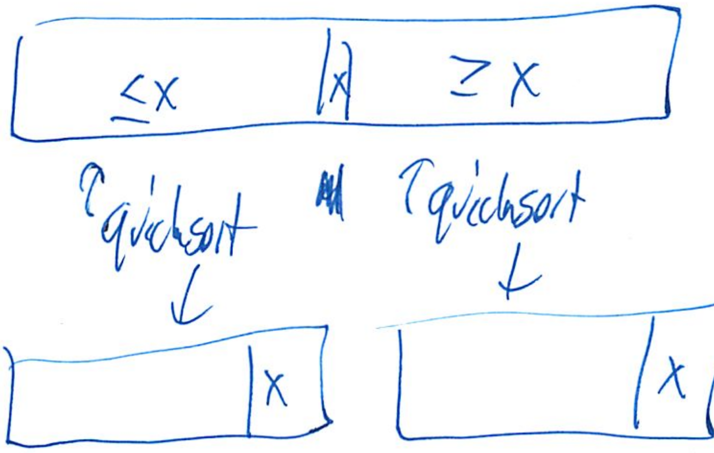
- partition exam in place
  - select last el as a pivot
  - into < pivot and > pivot

②

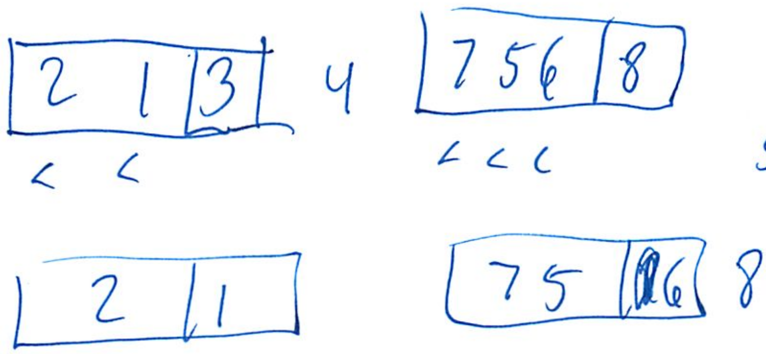


Swap + expand region

Then



So book example



Seems worthless – but not always

1 2 3 4 5 6 7 8 ✓ Done

Done w/ long dissection → not typically on it

(3) Comparison sort → generic name for all the sorts we saw

<u>Merge</u> sort - that one in the start of book

Split up then reassemble

~~heap sort~~

<u>insertion</u> - the niece sorting method

place card into the order

<u>heap sort</u>

$0 \leq A.heap\text{-}size \leq A.lenght$

parent
↳ $i/2$

left
↳ $2i$

right ↳ $2i+1$

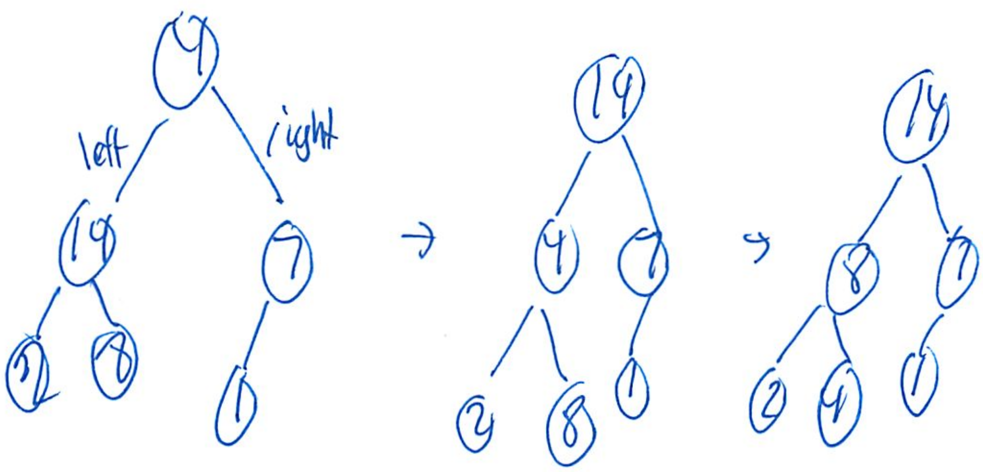
max heapify → $O(\lg n)$ to maintain a heap

heapsort → heap the whole thing $O(n \lg n)$

④

Max-heapify – assumes left and right are max heaps

A[i] can be smaller than child



Complex actual cases

but it floats down

So all we saw so far used comparison sort

decision trees $O(n \lg n)$ is best

Can beat that in certain cases (when data #s)

<u>Counting sort</u>

$O(n)$ scan through list → store # of times you see

↳ actually $O(n+k)$

(5)

Hard to rememebe

1. How it actually works
2. What happens (externally / black box)
3. How to simulate
4. Where it's good at
5. Running time

If want stable

↓

Radix sorting

digit by digit

$O(n + 2^r)$ per pass

$r = \lg n$ gives $O(n)$ per pass

or $\Theta\left(\frac{nb}{\log n}\right)$ total

⑥

Back from the old days

cards w/ 80 cols w/ 12 rows

Splits into 1 of 12 bins

Then just sort that bin

(sounds like map reduce)

but only 12 col

Start w/ ~~most~~ least sig digit

↳ so can combine all the stacks

– otherwise need a lot of piles

$O(d(n+k))$

d — digits

$(n+k)$ — sort stabally

n = # of #s

k = # of possible values

d = digits

When d = constant, $k = O(n)$

(7)

So what stable sort does it use?

WP say's O(k+n)

So sort I guess just use 10 buckets

$O(k+n)$

k: buckets (10)

n: #s

Since this is not all that mem efficent & in place quick sort is better

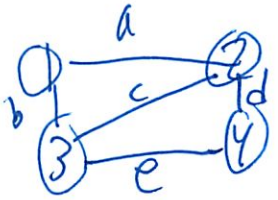
Oh the running time they gave in class was overly complex ...

8

Graphs, Representation + Search

The main (~ ≈80%) topic of exam

V = vertices
E = edges



4 ways to represent

- Adj list

  1 → 2 3
  2 → 1 3 4
  3 → 1 2 4
  4 → 2 3

- Incidence list

  list edges instead

  1 → a, b
  2 → a, c, d
  3 → b, c, e
  4 → d, e

(9)

Adj matrix

	1	2	3	4
1	1	1	1	0
2	1	1	1	1
3	1	1	1	1
4	0	1	1	1

A^2 = # of lenght 2 paths b/w vertices

Implicit representation

Symmetric if undirected

~~"this is like the other~~ Adj(u) returns results when you need it

Or a OO ~~a~~ ~~[illegible]~~ node.adj

Adj list $\Theta(n + m\log n)$

What is m?

Book just says $O(V+E)$

Why is this data conflicting?

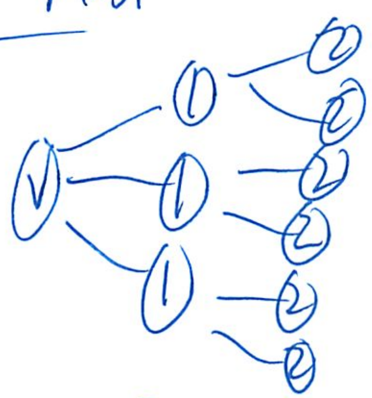
~~A~~ Book one so much simpler

⑩

Diff actions have diff costs

 └ pretty self explanitory

Breath First



? level   greedy   can find shortest path

DFS

like exploring a maze

when get stuck → back track

ie left hand rule

---

form a DFS "forest" from several connected components "trees"

Colors   start → white
discovered → gray
finished → black

timestamps → discovered (grayed)
↳ finished (blacked)

Can write w/ parentheses structure

Edges

(I hate these!)

1. Tree edges ie normal edges
2. Back edges go back to an ancestor
   ie self loops
   was in that last hw

directed only:

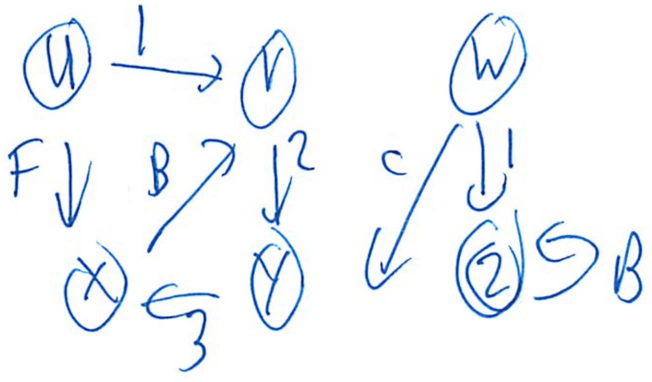
3. Forward edges non tree edges
   connecting to a decendent
4. Cross edges All other edges
   - same tree and not ancestor

(12)

I don't get 3 and 4

3. Is if we backtrack all the way and then search from there can get a forward edges.

4. Is from another connected component back ~~to~~ over to another connected component we saw before all



13

Can redraw w/ all forward edges ↓
and back edges ↗

Topo Sort

Only on a DAG
Run DFS
When finished put on front of list

① → ② → ③ ← order of finished

O(V+E) is BFS, DFS, Topo
- visit each vertex and each edge once

Strongly Connected Components

Decompose a directed graph
Use 2 DFSes
Run seperatly on each connected component

(14)

1. Do DFS(G) for finishing times
2. Compute G^T
   - ie change arrow direction
3. Call DFS(G^T) in order ↓ $v.f$
4. Output each tree as sep strongly connected component

(I don't get how this works
Oh I see it - all the ones in that tree from the last finishing)

---

* Path from each vertex to every other vertex

← can see easily visually

Can make a component graph
- will be a DAG

(15)

Directed (i, j)

Undirected {i, j}

BFS tracks a queue of what you saw

Augment BFS w/ levels

Possible is about ~~diff~~ would it change BFS

Which level is what

[M appears to be edges] ✓

⟷ Symmetric

↺ reflexive

→ → transitive

(their connected components chap seems same)

(16)

Rem DFS stack

Connected component - individual
linear line w/ good counting
(missed how to do)
Just run ~~fill~~ BFS, DFS till returns
Then restart at a unmarked ~~matrix~~ vertex (duh)

Topo Reverse -> un sort

More Shortest Paths

L single source
to multiple destination

relaxation -> if new path shorter than current
~~path~~ distance on record -> pick that

Bellman-Ford

- edges can be ⊖
- longer than Dijstra

~~At every~~ .

~~For each # as # of vertic~~

For as many times as # of vertices -1
  for every edge
    try to relax

If ⊖ weight cycle → return false

O(VE)

In a DAG

Topo-sort it

then for each vertex (once)
  Relax edges that leave the vertex

Dijstra

faster, but no Θ

keep a queue

extract a min

mark as final

relax all neighbors

add them to the queue

Both Bellman Ford + Dijstra are for weighted graphs

Dijstra use hash table instead of list to make it faster

2 tables → estimated and final

[Bellman ford just uses 1 table

Min Heap is better for the queue

Fib queue is the best - out of scope

$O([V+E] \lg V)$ for min heap

↑ extract min
V such op ↖ heap must be rebuilt

E such op for
decrease key $O(\lg V)$
↑ heap rebuilt

Deque "deck"
double linked list
double size of list if needed
aggregate analysis (or whatever its called)

Problem shortest 2 path comprised of 2 edges
Greedy never gets you shortest path
↑ keep these things in mind

Adj list V^3 ← 3 lookups

Or look at ~~all~~ each in E^2

? E^2V

~~No~~ No is just E^2

perhaps E^2+V

But every edge can only be paired w/ V out

↑ How are we supposed to see this?

O(EV)

For each node find smallest in and smallest out

need reverse adj → O(V+E) to build

?did we ever answer this?

~~Best 2 edge~~ Make list of stuff we learned — I think I got this now...

But what would this be

- find shortest edge going in
- Then just jump to that
  - sort, well find min (E)
- or for each vertex find min in + out

$V + 2E \to V + E$

but the find min can be like E

So $V + E^2$

The problem is what they consider
Describe it of course
There is no answer

---

for 3 lenght path

now consider else

best in → edge → best edge out

(22)

Bellman ford to count 3?

Well its a all pairs shortest path
review that chap

Chap 25

Nieve old algs V times

$O(V^3 + VE) = O(V^3)$

~~O(...)~~ $O(V^2 + E \log V)$
$O(V E \log V)$; I guess that is > than V2

Predicessor matrix $\Pi = \pi_{ij}$

predecessor subgraph

(23)

Matrix multiplication

$O(V^4)$ ok this is that chap it was following

Uses DP

1. Char. optimal sol
   (what does this mean?)
2. Recursively define value of optimal sol
3. Compute optimal bottom up

1. Shown all subparts of shortest paths = shortest path

Graph has at most m edges

$i \xrightarrow{p'} k \rightarrow j$

↑ m-1 edges

$\delta(i,j) = \delta(i,k) + w_{kj}$

(24)

2. So $l_{ij}(m)$ is min weight path from $i \to j$
w/ at most m sol

∞ if no path

Compute it as min of

$l_{ij}(m-1)$ or $\min_{1 \le k \le n} l_{ik}(m-1) + w_{kj}$

[oh todays lectures!
why was this alluded to before?

So either the m-1 path
or the m path
└ by looking at all predicessors
of j=k

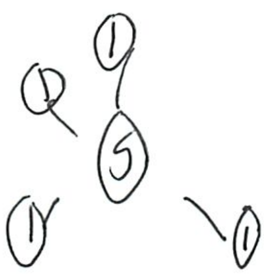
3. Compute bottom up

W/ matrices $L^{(1)}, L^{(2)}, L^{(3)}, \ldots L^{(n-1)}$

? actual shortest length path

$O(n^3)$ w/ 3 for loops

So



expanding from each

Very hard to visualize

All spread out from every one

Like matrix multiplication

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Still $O(n^3)$

Show all pairs shortest path $\Theta(n^4)$

L gets ya $L^{(m)}$

$L^{(5)} = L^{(4)} \cdot W$

↑ $L^{(1)}$ = weights

remember L is min weight

Repeated Squaring

Floyd - Warshall $\Theta(V^3)$

Yeah this is next lecture

That problem ^from recitation was a special case shortest

Yeah I don't get how got (V+E)

need to do some find min

- binary heap nlg n
- or BST nlgn

actually can find while processing → insertion sort if <

27

But then finding in + out w/ same node

(Or can you do that addition w/in V+E

Find V → O(1)

Scan list O(E)

Or keep min of each

- as part of building

Wait min left min right

but must share

So V² to try each combo

Find smallest O(1)

Then scan through V(E) for shared

Oh no its direct left + right for that node

Oh I think I see that

Took me forever to solve

(28)

So scan ~~elan~~ left for smallest path

E edges + V vertices

recorded smallest edge for each V

~~#~~ Reverse $O(V+E)$

Scan right for smallest path

same

Then scan vertices $O(V)$ doing addition

pick smallest - by making a note

$O(V+E)$

I should do more pratice

Problem: Find triple such that

29

So looking for a loop DFS

Or # of lenght 3 paths that loop

Nieve $O(V^3)$ doing matrix

Or BFS to find triangles up to level 3

$O(V(V+E))$

? What about the DFS was never covered

Problem Find # of shortest paths

Do BFS see which levels # shows up
w/ repeats

But BFS is occurance of paths, not vertex

He does adj matrix

Vo $V^{2.32}$ to compute any A^i

(30)

¿Can you use that $S \Lambda S^{-1}$ trick?

Can repeated square

Can solve linear w/ Dynamic Programming

$$sp(x) = \sum_{pred\ p_i} sp(p_i)$$

¿Work backwards?

Huristics

Random graph

- most time in last levels
- so double ended

BFS or Dijstra

Or use crow flys distances in planar graphs

↳ or other huristic

(31)

addition to previous: annotate w/ # of shortest paths to each vertex

---

Problem # shortest paths weighted (# ≥ 0)

Dijstra

Look at prev node

$$nsp(t) = \sum_{\text{prev } p_i} nsp(p_i)$$

This was that other chap we hadn't done yet...

Remember # of paths → not shortest...

Going level by level

Save results in look up table (memozation)

Dynamic Programming

exponential → polynomial

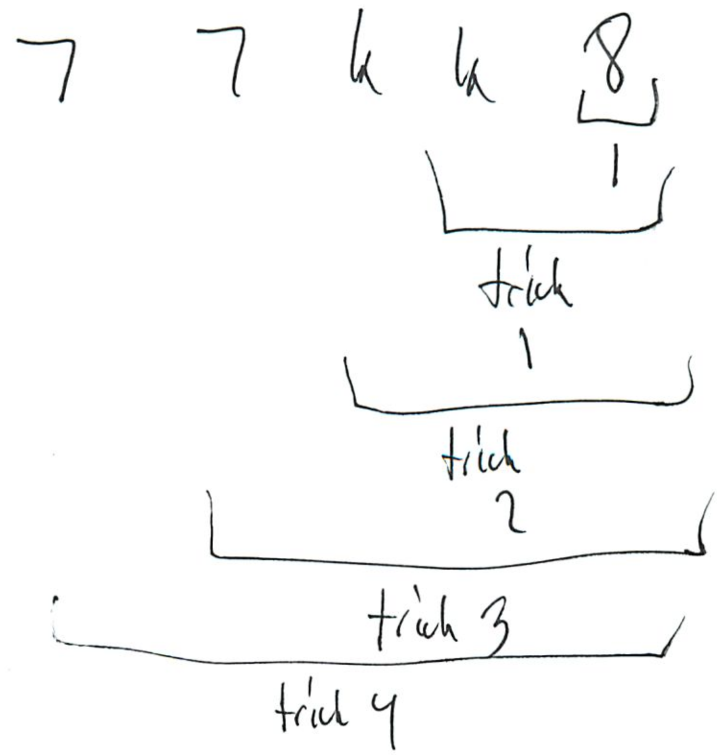
Subproblems made of subproblems

(I think I got this - but no practice)

Review crazy 8s

So this runs backwards??

└ why?



? is this it - wish there was a picture?

(33)

Basically like all pairs shortest path

↳ no that is 2 sep things!

One subproblem = O(n)

Example

[10 5 3 4]

left of 4, < 4

pick 4 → max(lis[3] + 1) or

max lis(10, 5, 3)

lis[3] = 1
base case

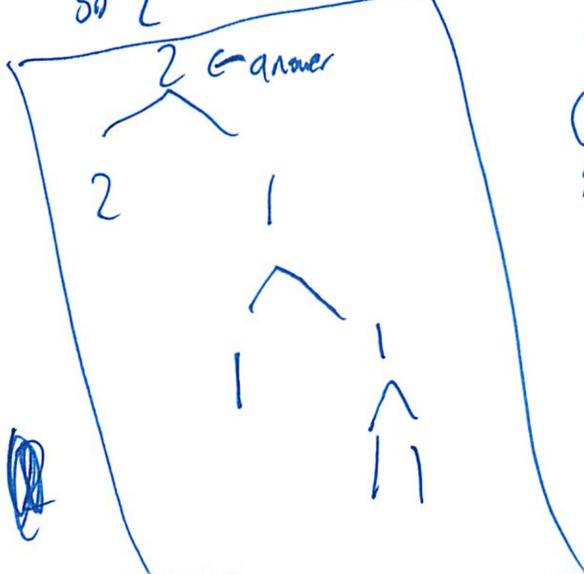
(lis[5] +1) or lis(10, 5)

left of 3, < 3
empty set

0 + 1
so 1

lis[7+1 or lis[10]
1 1

so 2



makes more sense now

What to memoize?

dp[i] = longest ↑ sub seq that ends w/ x[i]

? Oh this is diff subproblem

(that might have been confusing)

do local comparisons

Often < O(n)

Usually involves some special ~~subprob~~ data structure

heaps
BSTs
hashing
one value

dp = length of longest ~~sub problem so far~~ that ends here

? but how is that made of prev ans?
I guess as we move ⟵

(35) Blast from the Past

BSTs

key

< >

inorder walk
left
print $O(n)$
right

Search
go left or right $O(h) \to O(\log n)$ on balanced

min/max
go all the way left or right $O(h)$

insert
walk to place in tree $O(h)$
place node

delete
crazy complicated $O(h)$

(36)

<u>Balanced</u> cotdates as needed

note

167 views

Quiz 2 topics

Quiz 2 will cover everything up to lecture 18 (last thursday, the 19th - intro to dynamic programming), including:

- All the quiz 1 topics (especially BSTs and hashing, two of the most recycled ideas)
- Counting and radix sort
- graph representation, DFS, DAGs and topological sort
- finding paths: BFS, Dijkstra, Bellman-Ford, A*, bidirectional Dijkstra
- Introductory dynamic programming (memoization like Fibonacci, simple DP like the crazy eights problem)

Again, there won't be a cheat sheet allowed. huristic

#quiz2 #pin

follow 14 like 0

5 days ago by Jeff Wu 3 edits

followup discussions, *for lingering questions and comments*

Resolved Unresolved

Anonymous (5 days ago) - Not sure if this was covered already, but will we be allowed to use a cribsheet for the exam? IIRC the instructors mentioned in a previous post about exam 1 that it would be considered for exam 2.

Anonymous (4 days ago) - Are we going to be tested on all-pairs shortest path algorithms?

Anonymous (3 days ago) - Such as Floyd-Warshall, etc.

Jeff Wu **(Instructor)** (2 days ago) - No cribsheet. Sorry.

You won't be tested on Floyd-Warshall either. But we did cover an all-pairs shortest path algorithm via dynamic programming in the last lecture (lecture 18), so you might expect to see that.

Write a reply...

Resolved Unresolved

Anonymous (4 days ago) - Will the grades for Quiz 2 be out before Drop Date, considering that Quiz 2 is just one day before Drop Date?

Jeff Wu **(Instructor)** (3 days ago) - Unfortunately, I don't think so.

Anonymous (1 day ago) - What about those of us who were sick and missed exam 1, for whom exam 2 is worth 40% of the grade? For us, there is currently very little information on what grade we will get. Will it be possible for us to drop it late, if we do badly on exam 2?

Write a reply...

Resolved Unresolved

Anonymous (1 day ago) - Do we need to know about Minimum Spanning Trees, Kruskal and Prim Algorithms, and Constraint Graphs?

Anonymous (1 day ago) - Reason I'm asking is because I believe Minimum Spanning Trees was mentioned briefly in my recitation a few weeks ago.

Victor Pontis (1 day ago) - Definitely not.

Jeff Wu **(Instructor)** (1 day ago) - No.

Write a reply...

Resolved Unresolved

Anonymous (1 day ago) - If you're going to have lecture 19 on the exam can you post some more detailed notes about what that entails?

The slides from lecture 19 (and all the lectures for that matter) are not great study resources...

Anonymous (1 day ago) - Lecture 18 not 19.

Jeff Wu **(Instructor)** (1 day ago) - Know how the Fibonacci memoization works. Understand the crazy eights problem well. Also the all-pairs shortest path algorithm. Does that help?

Sorry if the slides aren't great. Feel free to go to office hours to ask about that lecture. Recitation on Wednesday will also review that lecture.

Write a reply...

Resolved **Unresolved**

Anonymous (15 hours ago) - What key properties do we need to know about DAGs? For example we were expected to know about how coloring relates to even and odd cycles on a previous exam. Any specifics would be helpful!

Anonymous (14 hours ago) - *previous pset

Jeff Wu **(Instructor)** (12 hours ago) - The pset question about coloring wasn't just about DAGs...

DAGs are important because they're a special case of directed graphs which comes up often, and for which many problems (B-F/Dijkstra, for example) have faster solutions than more general graphs. The ability to topologically sort in linear time is important.

Write a reply...

Resolved **Unresolved**

Jancarlo Perez (8 hours ago) - Do we need to know about Strongly Connected Components for the quiz?

Thanks.

Jeff Wu **(Instructor)** (28 minutes ago) - Yes - you should know their definition, at least. You do NOT need to know how to find them

(But the algorithm is here if you're curious: http://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm).

Jancarlo Perez (22 minutes ago) - Thank you

Write a reply...

Resolved **Unresolved**

Anonymous (7 hours ago) - Yeah, I was wondering if All-Pairs Shortest Paths are covered as well.

Chelsea Finn (7 hours ago) - Read above: "Know how the Fibonacci memoization works. Understand the crazy eights problem well. Also the all-pairs shortest path algorithm. Does that help?"

Anonymous (7 hours ago) - Crap, my bad. Thanks Chelsea.

Write a reply...

$(V^2 + E) \to (V^2)$ min prority queue

$((V+E) \lg V)$ binary heap

$(V \lg V + E)$ fib heap

dijstra

Sorting

Counting Sort

Counting sort can sort n integers in the range 0 to k in $O(n+k)$ time. Say the unsorted n integers are stored in array A. Counting sort works as follows:

1. Initialize counting array C, where $C[i]$ will contain the number of times the element i occurs in A. At initialization, $C[i] = 0$ for all i. Also initialize sorted array B, where B will contain all the elements in A in sorted order.

2. Iterate through A, incrementing $C[i]$ by 1 for each value i seen in A. At the end of this step, $C[i]$ = number of times element i was found in A

3. Iterate through C, setting $C[i] = C[i-1] + C[i]$ for each i in C. At the end of this step, $C[i]$ = number of elements less than or equal to i that were found in A

4. Iterate through A backwards, placing element $A[i]$ into $B[C[A[i]]]$ and decrementing $C[A[i]]$ by 1 for each i in A. At the end of this step, B will contain all the elements in A in sorted order

Radix Sort

Radix sort can sort n integers in base k with at most d digits in $O(d(n+k))$ time. It does this by using counting sort to sort the n integers by digits, starting from the least significant digit (i.e. ones digit for integers) to the most significant digit. Each counting sort will take $O(n+k)$ time since there are n elements and the elements are all integers in the range 0 to k since we're in base k. Since the maximum number of digits in these n integers is d, we will have to execute counting sort d times to finish the algorithm. This is how we get a $O(d(n+k))$ running time for radix sort.

The running time of radix sort depends on the base k that the integers are represented in. Large bases result in slower counting sorts, but fewer counting sorts since the number of digits in the elements decrease. On the other hand, small bases result result in faster counting sorts, but more digits and consequently more counting sorts.

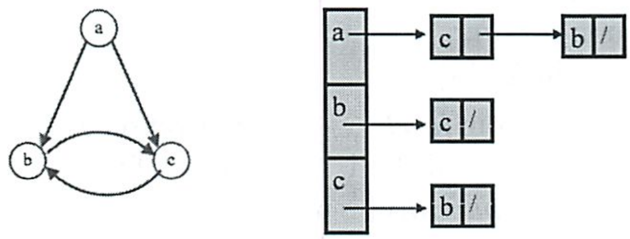
Let's find the optimal base k for radix sort. Say we are sorting n integers in the range 0 to $u-1$. The maximum number of digits in an element will be $\log_k u$ for some base k. To minimize running time, we will want to minimize $O((n+k)\log_k u)$. It turns out that to minimize running time, the best k to choose is $k = n$, in which case the running time of radix sort would be $O(n \log_n u)$. Note that if $u = n^{O(1)}$, the running time of radix sort turns out to be $O(n)$, giving us a linear time sorting algorithm if the range of integers we're sorting is polynomial in the number of integers we're sorting.

Graph Representation

The two main graph representations we use when talking about graph problems are the **adjacency list** and the **adjacency matrix.** It's important to understand the tradeoffs between the two representations. Let $G = (V, E)$ be our graph where V is the set of vertices and E is the set of edges where each edge is represented as a tuple of vertices.

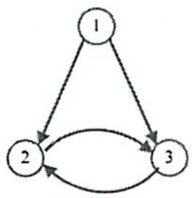
Adjacency List

An adjacency list is a list of lists. Each list corresponds to a vertex u and contains a list of edges (u, v) that originate from u. Thus, an adjacency list takes up $\Theta(V + E)$ space.



Adjacency Matrix

An adjacency matrix is a $|V| \times |V|$ matrix of bits where element (i, j) is 1 if and only if the edge (v_i, v_j) is in E. Thus an adjacency matrix takes up $\Theta(|V|^2)$ storage (note that the constant factor here is small since each entry in the matrix is just a bit).



1	2	3	
0	1	1	1
0	0	1	2
0	1	0	3

Comparison

The worst case storage of an adjacency list is when the graph is **dense**, i.e. $E = \Theta(V^2)$. This gives us the same space complexity as the adjacency matrix representation. The $\Theta(V + E)$ space complexity for the general case is usually more desirable, however. Furthermore, adjacency lists give you the set of adjacent vertices to a given vertex quicker than an adjacency matrix $O(\text{neighbors})$ for the former vs $O(V)$ for the latter. In the algorithms we've seen in class, finding the neighbors of a vertex has been essential.

BFS

BFS (breadth first search) is an algorithm to find the shortest paths from a given vertex in an unweighted graph. It takes $\Theta(V + E)$ time.

```
BFS(V,Adj,s)
 level={s: 0}; parent = {s: None}; i=1
 frontier=[s]                      #previous level, i-1
 while frontier
     next=[]                       #next level, i
     for u in frontier
        for v in Adj[u]
            if v not in level      #not yet seen
                  level[v] = i     #level of u+1
                  parent[v] = u
                  next.append(v)
     frontier = next
     i += 1
```

DFS

DFS (depth first search) is an algorithm that explores an unweighted graph. DFS is useful for many other algorithms, including finding strongly connected components, topological sort, detecting cycles. DFS does not necessarily find shortest paths. It also runs in $\Theta(V + E)$ time.

- *parent* = {s: None}
- call ***DFS-visit*** (V, Adj, s)

```
def DFS-visit (V, Adj, u)
    for v in Adj[u]
        if v not in parent            #not yet seen
            parent[v] = u
            DFS-visit (V, Adj, v)        #recurse!
```

Edge Classification

We classify the edges in the resulting DFS tree as one of the following four types:

1. **Tree edge** - an edge that is traversed during the search.
2. **Back edge** - an edge (u, v) that goes from a node u to an ancestor of it in the DFS tree.
3. **Forward edge** - an edge (u, v) that goes from a node u to a descendant of it in the DFS tree.
4. **Cross edge** - any other edge in the original graph not classified as one of the above three types.

Selected Past Test Questions

You are at an airport in a foreign city and would like to choose a hotel that has the maximum number of shortest paths from the airport (so that you reduce the risk of getting lost). Suppose you are given a city map with unit distance between each pair of directly connected locations. Design an $O(V+E)$-time algorithm that finds the number of shortest paths between the airport (the source vertex s) and the hotel (the target vertex t).

If a topological sort exists for the vertices in a directed graph, then a DFS on the graph will produce no back edges.

Other Important Topics

We did not have time to cover all possible topics regarding Graphs/BFS/DFS at the review session. You should also review anything else in the lecture/recitation notes. For example:

- Beginning/Finishing times for DFS
- Topological sort
- BFS queue vs DFS stack
- Rubik's cube graph
- Proofs of correctness and runtime
- DAG's
- Connected components

For shortest path problem, we are given a weight function $W: E \to \mathbb{R}$ where each edge is assigned a weight. We also assign a weight of infinity for edges that don't exist.

Our goal is to find a path from source S to all other vertices s.t. the sum of the weights of edges along the path is minimal.

1. Bellman - Ford.

```
Relax (u,v)
    if d[v] > d[u] + w(u,v)
        d[v] ← d[u] + w(u,v)
        π[v] ← u.
```

d[v] = current calculated shortest distance from s to v.

w(u,v) = edge weights of (u,v)

π[v] = the current parent of v that lead to the shortest path.

```
Initialize (V, E, s).
    for v ∈ V
        d[v] ← ∞
        π(v) = null.
        d[s] = 0
        π[s] = s.
```

```
Bellman - Ford (V, E, s)
    initialize (V, E, s)
        for i = 1 : |V|-1
            for each edge (u,v) ∈ E
                Relax (u,v).
    for each edge (u,v) ∈ E
        if d[v] > d[u] + w(u,v)
            return False
```

} this returns that ∃ a negative weight cycle.

Because the graph can only contain positive weight cycle, we would not have a cycle in our shortest path, so the shortest path have at most $|V|-1$ edges, so after $|V|-1$ iterations of relaxation, we would have the shortest path.

Running time: $O(VE)$, $(V-1)$ iterations, each iteration we relax $|E|$ edges,

Dijkstra:

In fact we can be more selective on the edges that we relax on each iteration.

2 Dijkstra:

```
Dijkstra (V, E, s)
    initialize (V, E, s).                Q is a min priority queue.
    push all v ∈ V → Q.
    while Q not empty
        u = Q.pop ;
        for every v s.t. (u, v) ∈ E.
            relax (u, v).
```

So each time we only relax edges whose starting point currently have the smallest $d[u]$ value.

Runtime: $O(V \cdot (\text{extract-min}) + E \cdot (\text{decrease-key}))$.

we only look through each vertex once and relax each edge once problem 1: if we increase each edge weights by 1, we still find the shortest path.

Ans. False,

problem 2: if all edges in graph have distinct weights, shortest path are distinct.

Ans. False,

problem 3: Given graph $G = (V, E, w)$. given $\delta(s, u)$ for all $u \in V$ but we are not given $\pi(u)$ for any u, how to find the shortest path from s to a given t.

ans: start with t, one of $v \in V$ s.t. $\delta(v) + w(v, t) = \delta(t)$, then recursively work on v, the running time is $O(V+E)$ since we hit each edge and vertex at most once.

Note: A shortest path should not contain a cycle, for ex. if there exist a zero-weight cycle, the shortest path should ignore it.

problem 4: modified shortest path, if all we care is to minimize the maximum edge weight along a path, how to find shortest path.

answer: change relax function.

if $d[u] > w(v,u)$ and $d(v)$,

$$d(u) = \max\{w(v,u), d(v)\}$$

Contents

1 Depth First Search: Characterizing Nodes and Edges

1.1 Discovery and Finishing Times

Discovery Time: The discovery time $d[v]$ is the number of nodes discovered or finished before first seeing v (call to DFS-VISIT).

Finishing Time: The finishing time $f[v]$ is the number of nodes discovered or finished before finishing the expansion of v (return from DFS-VISIT).

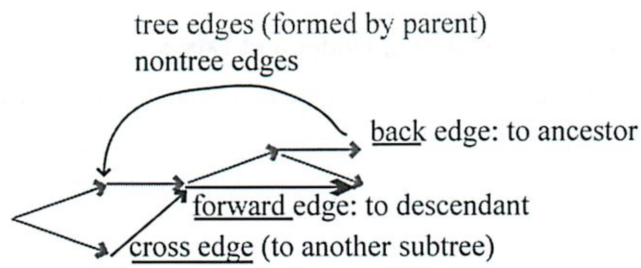


Figure 1: Edge classifications

For two nodes u and v, either $[d[u], f[u]] \subset [d[v], f[v]]$ (or vice versa). or the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are disjoint.

Proof: If u is a descendent of v in the search then $d[v] < d[u]$. Moreover, we must return from DFS-VISIT(u) before we return from DFS-VISIT(v) so $f[u] > f[v]$. In this case $[d[u], f[u]] \subset [d[v], f[v]]$.

If u is not a descendent of v and v is not a descendent of u then we must either finish expanding v before we discover u or finish u before discovering v (since if we discover u while expanding v then u is a descendent of v). In this case $[d[u], f[u]]$ and $[d[v], f[v]]$ are disjoint.

1.2 Edge Classifications

When doing a DFS, we think about four types of edges:

- *Tree edges*: Edges traversed in the search. If the edge is (u, v) then, when we first saw edge (u, v), we expanded v. If there is a path of tree edges from w to s then w is an *ancestor* of s and $[d[s], f[s]] \subset [d[w], f[w]]$.
- *Back edges*: A non-tree edge leading from a node u to a node v where there is a path from v to u consisting of tree edges. If there is a back edge (u, v) then v is an ancestor of u so $[d[u], f[u]] \subset [d[v], f[v]]$.
- *Forward edges*: A non-tree or -back edge leading from a node u to a node v where there is a path of tree edges from u to v. Here u is an ancestor of v so $[d[v], f[v]] \subset [d[u], f[u]]$.
- *Cross edges*: Edges that are not tree, back, or forward edges. If (u, v) is a cross edge then $[d[u], f[u]]$ and $[d[v], f[v]]$ will be disjoint.

Examples of edge types are shown in Figure 1

1.3 Node Coloring

During depth first search, a node can be in three states:

- Never been seen (White)

- Currently on the stack (Gray)
- Already popped off the stack and fully expanded (Black)

The color of the node when we see it tells us a lot about the structure of the search to this point. Assume we are expanding node v and considering child u

- u is white:
  - We will expand u right now
  - u is a *descendent* of v
  - v is an *ancestor* of u
  - v can reach u (possibly u cannot reach v)
  - (v, u) is a *tree edge*
  - We discovered u after v and must finish expanding it before we finish expanding v so $[d[u], f[u]] \subset [d[v], f[v]]$.
- u is gray:
  - u is currently on the stack, therefore it is currently being expanded
  - u is an *ancestor* of v
  - v is a *descendent* of u
  - u can reach v **and** v can reach u
  - (v, u) is a *back edge*
  - There is a cycle in the graph involving v and u
  - We started expanding v during the expansion of u so v was discovered after u and must be finished before u: $[d[v], f[v]] \subset [d[u], f[u]]$.
- u is black
  - The graph must be directed
  - v is an ancestor of u **or** u is not an ancestor of v and v is not an ancestor of u
  - (v, u) is either a forward edge or a cross edge (which can be determined by starting and finishing times)
  - Either $[d[u], f[u]] \subset [d[v], f[v]]$ (forward edge) or $[d[u], f[u]]$ and $[d[v], f[v]]$ are disjoint.

Undirected Graphs have only tree and back edges: Let (u, v) be an edge not traversed during DFS. Then when we saw edge (u, v) we must have already pushed v onto the stack (since we do not expand v). Moreover, if we are currently visiting u then clearly we had not visited u when we pushed v onto the stack. Therefore, we cannot yet have finished v because there is a path from v to u (along edge (u, v) among others). Thus v is an ancestor of u and (u, v) is a back edge.

2 Topological Sort

Recall: We must sort vertices such that if u can reach v then u is sorted before v. We run DFS on a DAG and then sort by decreasing finish times. Given Section 1.1, it's clear why it works:

Assume u can reach v. While expanding u, we must see v. When we see v, it is either white, in which case v is a descendent of u and we have $f[v] < f[u]$ or it is black, in which case $f[v] < f[u]$ since $f[u]$ has yet to be assigned. Note that v cannot be gray since the graph is acyclic. Therefore if u can reach v, u will have a higher finishing time than v and be sorted first.

3 Graph Representations and Transformations

3.1 Implicit Representation

Sometimes we don't want to actually build the graph using an adjacency matrix or lists.

Example: An infinite grid. This cannot possibly be constructed... but that doesn't mean you can't view it as a graph! Given a point (x, y) on the grid, we can define its neighbors using an adjacency function

Adj(x, y)
1 **return** $[(x-1, y), (x+1, y), (x, y-1), (x, y+1)]$

That's really all we need! Why is this useful? Because you may still care about things like the shortest path in the grid from one point to another. Even though you cannot possibly represent the graph, you can still do Dijkstra's early-termination algorithm on it!

3.2 Graph Transformations

Problem: Assume we add 1 to every weight in a graph. Can this change the shortest path from u to v? What if we multiply every weight by a positive constant a?

Solution: After adding 1 to every weight the path lengths change by:

$$w'(p) = w(p) + |p|$$

where $w'(p)$ is the cost of p after we add 1 to every weight, $w(p)$ is the cost before and $|p|$ is the length of the path. Therefore, adding 1 to every weight can change a path. Assume we have one path from s to v with three edges, each with weight 1 and another path with only one edge of weight 4. Then the shortest path from s to v with unmodified weights is along the 3-edge path (length 3) while the shortest path from s to v with modified weights is along the one edge (length 4).

If we multiply each weight by 1 then

$$w'(p) = aw(p)$$

Therefore if $w(p_1) < w(p_2)$, $w'(p_1) < w'(p_2)$ and the shortest path remains the same.

Fall 2009 Quiz 2 Problem 5: Given an undirected, weighted graph G, we have some subset of edges $R \subset E$ that are considered "rough". Give an algorithm to find the shortest point from a vertex s to all other vertices that uses *at most* one rough edge.

Solution: Create a new *directed* graph $G' = (V', E')$. For every vertex $v \in V$, add two vertices v_r and v_s to V' (so $|V| = 2|V'|$). For each smooth edge (u, v), add the edges (u_s, v_s), (v_s, u_s), (u_r, v_r) and (v_r, u_r) to E' (remember (u, v) was undirected). For each rough edge (u, v), add the edges (u_s, v_r) and (v_s, u_r) to E'. Run Dijkstra on G' from s_s. Then $\delta(s, v) = \min(d[v_r], d[v_s])$.

In this graph, both the smooth (subscript s) and rough (subscript r) clusters have only smooth edges. However, rough edges are only in the graph as a path from the smooth to the rough cluster. Once you have traversed a rough edge to the rough cluster, there is no path back. Therefore, if we start at s_s and finish at v_r, we have traversed exactly one rough edge. If we start at s_s and end at v_s, we traversed no rough edges.

4 Shortest Path Theorems

You can cite any of these during the quiz so you should definitely know them! Knowing their proofs will help you understand why all the shortest path algorithms work. All are proved in CLRS or in the notes from Recitation 15.

- **Subpath Theorem** Let $\{v_1, v_2, ..., v_n\}$ be a shortest path from v_1 to v_n. Then any subsequence of this path from v_i to v_j is a shortest path from v_i to v_j.
- **Triangle Inequality** $\forall u, v, x \in V$, we have $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$.
- **Upper Bound Property** We always have $d[v] \geq \delta(s, v)$ and if we ever find $d[v] = \delta(s, v)$, $d[v]$ never changes.
- **Path Relaxation Property** Assume we have a graph G with no negative cycles. Let $p = \langle v_0, v_1, ..., v_j \rangle$ be a shortest path from v_0 to v_j. Any sequence of calls to RELAX that includes, in order, the relaxations of $(v_0, v_1), (v_1, v_2), ..., (v_{j-1}, v_j)$ produces $d[v_j] = \delta(v_0, v_j)$ after all of these relaxations and at all times afterwards. Note that this property holds regardless of what other relaxation calls are made before, during, or after these relaxations.

5 Bellman-Ford

5.1 Things to Know

- You cannot be sure that $d[v] = \delta(s, v)$ until the algorithm has finished.
- Bellman-Ford returns FALSE if it finds a negative cycle.
- The running time is $O(|V||E|)$. This is (much) worse than Dijkstra's algorithm.
- The running time of Bellman-Ford on a DAG is only $O(|E| + |V|)$. See below.
- The proof for why this works is in the book and also in the notes for Recitation 15. Knowing this proof is an excellent way of understanding how the algorithm works.

5.2 On a DAG

Bellman-Ford takes forever because we must relax all edges for every possible path in order. However, on a DAG, we can figure out the order of relaxation easily! Here's what you do:

1. Topologically sort the graph $O(|E| + |V|)$
2. Run *one* iteration of Bellman-Ford taking the vertices in topological order $O(|E| + |V|)$
3. Note: We know there are no cycles, so we don't need to do the negative cycle check at the end!

Assume $p = \langle (s = v_0, v_1), (v_2, v_3), ..., (v_{n-1}, v_n) \rangle$ is a shortest path. Now consider edge (v_i, v_{i+1}). Then v_i is sorted after all $v_{j<i}$ and before all $v_{j>i}$ so we relax all edges $(v_0, v_1), ..., (v_{i-1}, v_i)$ *before* (v_i, v_{i+1}) and all edges $(v_{i+1}, v_{i+2}, ..., (v_{n-1}, v_n)$ after (v_i, v_{i+1}). So by the path relaxation property (look it up in CLRS or Recitation 15), we will report the shortest path!

6 Dijkstra's Algorithm

6.1 Things to Know

- When a vertex v pops off the priority queue, $d[v] = \delta(s, v)$, the shortest path from s to v
- When a vertex v pops off the priority queue, no vertex u will pop off the priority queue at any point later in the algorithm with $d[u] < d[v]$.
- Once a vertex v has popped off the queue, we will never change $d[v]$.
- The running time of Dijkstra depends on your priority queue implementation. If you use a Fibonacci heap, the running time is $O(|E| + |V| \log |V|)$. If you use a binary heap, the running time is $O((|E| + |V|) \log |V|)$.
- See the paper or Recitation 17 for speedups.

Fall 2008 Final Problem 9: Assume we have a directed graph $G = (V, E)$ with non-negative edge weights. We wish to find the shortest path from a vertex $s \in V$ to a vertex $t \in V$ with one caveat: While traversing a path from s to t you may set one edge weight of your choosing to zero. Given an algorithm for finding the shortest path with this caveat.

Solution: First run Dijkstra's algorithm to find the shortest path from s to every other vertex in the graph. Then run Dijkstra's on the transpose graph to find the shortest path from every vertex in the graph to t. Now iterate through the edges (u, v) calculating the path cost from s to t if we set that edge to zero weight:

$$w(s \to t) = \min(\delta(s, t), \delta(s, u) + \delta(u, v))$$

Choose to set the edge to zero that minimizes $w(s \to t)$. Note that if the path cost from s to t is non-zero then $w(s \to t)$ should be less than $\delta(s, t)$.

7 Example Problems

Fall 09 Quiz 2 Problem 3: Consider a graph $G = (V, E)$ that has both directed and undirected edges. There are no cycles in G that use only directed edges. Give an algorithm to assign each undirected edge a direction so that the completely directed graph has no cycles.

Solution: First topologically sort the graph using only the directed edges. Create an array so that for each vertex you store its number in the short. Then, for each undirected edge, draw the edge in the direction that goes from the vertex with the lower sort number to the higher sort number.

Fall 2008 Problem 3a: Given a directed graph G, you would like to get from s to t stopping at u if not too inconvient where "too inconvenient" means the shortest path that stops at u is more than 10% longer than the shortest path from s to t. Give an algorithm for returning the shortest path from s to t that stops at u if convenient.

Solution: Run Dijkstra's algorithm once from s and once from u. The shortest path from s to t is found in doing Dijkstra's algorithm from s. The shortest path from s to t through u is the shortest path from s to u plus the shortest path from u to t.

8 More Example Problems

Problem: Give an algorithm to find the shortest path containing an *even* number of edges in the directed graph with non-negative weights. Your algorithm should have the same running time as Dijkstra's.

Solution: Create a new graph G' as follows: for each $v \in V$, add two vertices v_{red} and v_{blue} to V' (so $|V'| = 2|V|$). Then for every edge (u, v) in E, add the edges $(u_{\text{red}}, v_{\text{blue}})$ and $(u_{\text{blue}}, v_{\text{red}})$ to E' (so $|E'| = 2|E|$). Run Dijkstra on G' starting at s_{red} and report the distance from s to v as the distance from s_{red} to v_{red}.

Every time you traverse an edge in G' you change the color of your cluster. Therefore, if you begin at a red vertex and end at a red vertex you must have traversed an even number of edges. Prove the correctness rigorously yourself as an exercise in how Dijkstra works!

Critical Edges: You are given a graph $G = (V, E)$ a weight function $w : E \to \Re$, and a source vertex s. Assume $w(e) \geq 0$ for all $e \in E$.

We say that an edge e is upwards critical if by increasing $w(e)$ by any $\epsilon > 0$ we increase the shortest path distance from s to some vertex $v \in V$.

We say that an edge e is downwards critical if by decreasing $w(e)$ by any $\epsilon > 0$ we decrease the shortest path distance from s to some vertex $v \in V$ (however, by definition, if $w(e) = 0$ then e is not downwards critical, because we can't decrease its weight below 0).

1. Claim: an edge (u, v) is downwards critical if and only if there is a shortest path from s to v that ends at (u, v), and $w(u, v) > 0$. Prove the claim above.

Solution: First, note that if (u, v) is on any shortest path, then because subpaths of shortest paths are shortest paths, (u, v) will also be on a shortest path to v.

Second, we prove that (u, v) is downwards critical implies (u, v) is on the shortest path from s to v.

Proof by contradiction: Assume (u, v) is downwards critical, but it is not on the shortest path from s to v. Then $\delta[s, v] < \delta[s, u] + w(u, v)$, so let $\epsilon = (\delta[s, u] + w(u, v) - \delta[s, v])/2$ is positive. If we decrease $w(u, v)$ by ϵ, we'll only be changing the cost of the paths to v going through (u, v), so the cost of the minimum path will stay the same. By the choice of ϵ, the best path going through (u, v) will still cost more than the minimum path. So the minimum path cost doesn't change when $w(u, v)$ is decreased by ϵ. Contradiction.

Third, we prove that (u, v) is on the shortest path from s to v implies (u, v) is downwards critical.

If (u, v) is on a shortest path to v, then decreasing its weight by any $\epsilon > 0$ decreases the cost of that path. We know that no other path through v had a lower cost than $w(u, v)$, so the path containing (u, v) is still the shortest path to v. So by decreasing the weight of (u, v), the weight of the shortest path to v is decreased, which means (u, v) is downwards critical.

2. Make a claim similar to the one above, but for upwards critical edges, and prove it.

Solution: Claim: (u, v) is upwards critical if and only if all the shortest paths from s to v end at (u, v).

First, we again start by noting that if (u, v) is on all shortest paths to any particular node, then it must also be on all shortest paths to v.

Second, we prove that if (u, v) is upwards critical then all the shortest paths from s to v end at (u, v).

If (u, v) is upwards critical, then increasing $w(u, v)$ by $\epsilon > 0$ must increase the cost of all shortest paths from s to v, otherwise the minimum cost to get from s to v would stay the same. Increasing $w(u, v)$ only impacts the paths containing (u, v), therefore (u, v) must be contained on all shortest paths to v. Since all the edges have positive weights, (u, v) must be the last edge on any the shortest path from s to v.

Third, we prove that if all the shortest paths from s to v end at (u, v) then (u, v) is upwards critical.

If all the shortest paths from s to v include (u, v), then increasing $w(u, v)$ by any $\epsilon > 0$ increases the cost of all these paths. Therefore, the minimum cost to get from s to v is increased, so (u, v) is upwards critical.

3. Using the claims from the previous two parts, give an $O(E \log V)$ time algorithm that finds all downwards critical edges and all upwards critical edges in G.

Solution: Run Dijkstra using binary heaps as a priority queue (binary trees or Fibonacci heaps are also acceptable data structures here). Save the results in $d[v]$ and $\pi[v]$.

Iterate through all edges, and report an edge (u, v) as downwards critical if $d[u] + w(u, v) = d[v]$. This is correct because the edges satisfying the condition must belong to the shortest paths from s to v. While doing this, compute $dc[v]$ = the number of downwards critical edges coming into v.

Iterate through all the vertices, and report $(\pi[v], v)$ as upwards critical if $dc[v] = 1$. This is correct because the check implies that $(\pi[v], v)$ is the only edge coming into v that is on a shortest path from s to v.

Running time analysis: all vertices are reachable from v, so it must be that $V = O(E)$. Then the running time of Dijkstra is $O((V+E)logV) = O(ElogV)$. Reporting downwards critical edges takes $O(E)$, because we do $O(1)$ work per iteration over all the edges. Reporting upwards critical edges takes $O(V)$, because we do $O(1)$ work per iteration over all the vertices. So the total running time is $O(ElogV + E + V) = O(ElogV)$

9 Radix and Counting Sorts

9.1 Counting Sort

Counting sort can beat comparison sort bound *because* it doesn't use comparisons. In order not to use comparisons, we must have a little bit of "extra" knowledge. Namely: given an array A to sort, we need to be able to map every element that might appear in A uniquely to integers in $[1, k]$ where k is a small integer.

Input: A array to be sorted

Output: B sorted array of elements of A

Pass 1: Create array C of length k. $C[i]$ stores the number of times i appears in A. For each i, $C[A[i]] = C[A[i]] + 1$

Pass 2: For each entry i in C, put $C[i]$ i's in B.

This takes $O(n+k)$.

Problem: B is not *stably* sorted. Two equal keys may swap their relative orders. We would like to avoid that.

New algorithm: Create C', which stores in $C'[i]$ number of numbers in A less than or equal to i. Fill in C' after filling in C just by keeping running total.

Now, for $j = length[A]$ downto 1, place $A[j]$ at $C'[A[j]]$ in B and decrement $C'[A[j]]$. Now the sorting is stable.

Example: We know we are sorting numbers from 1 to 8

$$
\begin{aligned}
A &= [2,7,5,3,5,4] \\
C &= [0,1,1,1,2,0,1,0] \\
C' &= [0,1,2,3,5,5,6,6]
\end{aligned}
$$

Stepping through the second pass (0 in B indicates nothing there yet):

1.

$$
\begin{aligned}
B &= [0,0,0,0,0,0] \\
C' &= [0,1,2,3,5,5,6,6]
\end{aligned}
$$

2.

$$
\begin{aligned}
B &= [0,0,4,0,0,0] \\
C' &= [0,1,2,2,5,5,6,6]
\end{aligned}
$$

3.

$$
\begin{aligned}
B &= [0,0,4,0,5,0] \\
C' &= [0,1,2,2,4,5,6,6]
\end{aligned}
$$

4.

$$
\begin{aligned}
B &= [0,3,4,0,5,0] \\
C' &= [0,1,1,2,4,5,6,6]
\end{aligned}
$$

5.

$$
\begin{aligned}
B &= [0,3,4,5,5,0] \\
C' &= [0,1,1,2,3,5,6,6]
\end{aligned}
$$

6.

$$
\begin{aligned}
B &= [0,3,4,5,5,7] \\
C' &= [0,1,1,2,3,5,5,6]
\end{aligned}
$$

7.

$$
\begin{aligned}
B &= [2,3,4,5,5,7] \\
C' &= [0,0,1,2,3,5,5,6]
\end{aligned}
$$

9.2 Radix Sort

Digit-by-digit sort of list of numbers. Sort on least-significant digit first using a *stable* sort.

Why least significant? Example: Most significant

$$
\begin{array}{ccccc}
33 & & 33 & & 52 \\
55 & \rightarrow & 55 & \rightarrow & 33 \\
52 & & 52 & & 55
\end{array}
$$

Oops!

Why do we need a stable sort? Because we don't want to mess up orderings caused by earlier digits in later digits!

Example: Sort 33, 55, 52

$$
\begin{array}{ccccc}
33 & & 52 & & 33 \\
55 & \rightarrow & 33 & \rightarrow & 52 \\
52 & & 55 & & 55
\end{array}
$$

We can only guarantee that 52 comes before 55 if we can guarantee the sort on the second digit is stable!

Running Time: Sorting n words of b bits each: Each word has b/r 2^r-base digits. Example: 32-bit word is has 4 8-bit digits. Each counting sort takes $O(n + 2^r)$, we do b/r sorts. Choose $r = \log n$, gives running time $O(nb/\log n)$.

Correctness: By induction on number of digits. Do it for practice!

Quiz 2

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- When the quiz begins, write your name on every page of this quiz booklet.
- You have 120 minutes to earn **120** points. Do not spend too much time on any one problem. Read them all through first, and attack them in the order that allows you to make the most progress.
- This quiz booklet contains 19 pages, including this one. Two extra sheets of scratch paper are attached. Please detach them before turning in your quiz at the end of the exam period.
- This quiz is closed book. You may use **one** handwritten, $8\frac{1}{2}'' \times 11''$ or A4 crib sheets (both sides). No calculators or programmable devices are permitted. No cell phones or other communications devices are permitted.
- Write your solutions in the space provided. If you need more space, write on the back of the sheet containing the problem. Pages may be separated for grading.
- Do not waste time and paper rederiving facts that we have studied. It is sufficient to cite known results.
- Show your work, as partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Be neat.
- Good luck!

Problem	Parts	Points	Grade	Grader		Problem	Parts	Points	Grade	Grader
1	8	24				6	–	10		
2	6	24				7	–	10		
3	4	12				8	–	10		
4	–	10				9	–	10		
5	–	10								
						Total		120		

Name: ______________________________

Athena username: ______________________________

Recitation:	Nick **WF10**	Nick **WF11**	Tianren **WF12**	David **WF1**	Joe **WF2**	Joe **WF3a**	Michael **WF3b**

Problem 1. True or false [24 points] (8 parts)

For each of the following questions, circle either T (True) or F (False). **Explain your choice.** (Your explanation is worth more than your choice of true or false.)

(a) T F Instead of using counting sort to sort digits in the radix sort algorithm, we can use any valid sorting algorithm and radix sort will still sort correctly.

(b) T F The depth of a breadth-first search tree on an undirected graph $G = (V, E)$ from an arbitrary vertex $v \in V$ is the diameter of the graph G. (The ***diameter*** d of a graph is the smallest d such that every pair of vertices s and t have $\delta(s, t) \leq d$.)

(c) T F Every directed acyclic graph has exactly one topological ordering. has valid topological orderings $[a, b, c]$ or $[a, c, b]$. As another example, $G = (V, E) = (\{a, b\}, \{\})$ has valid topological orderings $[a, b]$ or $[b, c]$.

(d) T F Given a graph $G = (V, E)$ with positive edge weights, the Bellman-Ford algorithm and Dijkstra's algorithm can produce different shortest-path trees despite always producing the same shortest-path weights.

(e) T F Dijkstra's algorithm may not terminate if the graph contains negative-weight edges.

(f) T F Consider a weighted directed graph $G = (V, E, w)$ and let X be a shortest s-t path for $s, t \in V$. If we double the weight of every edge in the graph, setting $w'(e) = 2w(e)$ for each $e \in E$, then X will still be a shortest s-t path in (V, E, w').

(g) T F If a depth-first search on a directed graph $G = (V, E)$ produces exactly one back edge, then it is possible to choose an edge $e \in E$ such that the graph $G' = (V, E - \{e\})$ is acyclic.

(h) T F If a directed graph G is cyclic but can be made acyclic by removing one edge, then a depth-first search in G will encounter exactly one back edge.

Problem 2. Short answer [24 points] (6 parts)

(a) What is the running time of RADIX-SORT on an array of n integers in the range $0, 1, \ldots, n^5 - 1$ when using base-10 representation? What is the running time when using a base-n representation?

(b) What is the running time of depth-first search, as a function of $|V|$ and $|E|$, if the input graph is represented by an adjacency matrix instead of an adjacency list?

(c) Consider the directed graph where vertices are reachable tic-tac-toe board positions and edges represent valid moves. What are the in-degree and out-degree of the following vertex? (It is O's turn.)

X	O	X
	O	
	X	

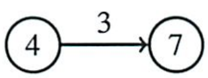
(d) If we modify the RELAX portion of the Bellman-Ford algorithm so that it updates $d[v]$ and $\pi[v]$ if $d[v] \geq d[u] + w(u, v)$ (instead of doing so only if $d[v]$ is strictly greater than $d[u] + w(u, v)$), does the resulting algorithm still produce correct shortest-path weights and a correct shortest-path tree? Justify your answer.

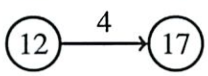
(e) If you take 6.851, you'll learn about a priority queue data structure that supports EXTRACT-MIN and DECREASE-KEY on integers in $\{0, 1, \ldots, u-1\}$ in $O(\lg \lg u)$ time per operation. What is the resulting running time of Dijkstra's algorithm on a weighted direct graph $G = (V, E, w)$ with edge weights in $\{0, 1, \ldots, W-1\}$?

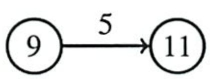
(f) Consider a weighted, directed acyclic graph $G = (V, E, w)$ in which edges that leave the source vertex s may have negative weights and all other edge weights are non-negative. Does Dijkstra's algorithm correctly compute the shortest-path weight $\delta(s, t)$ from s to every vertex t in this graph? Justify your answer.

Problem 3. You are the computer [12 points] (4 parts)

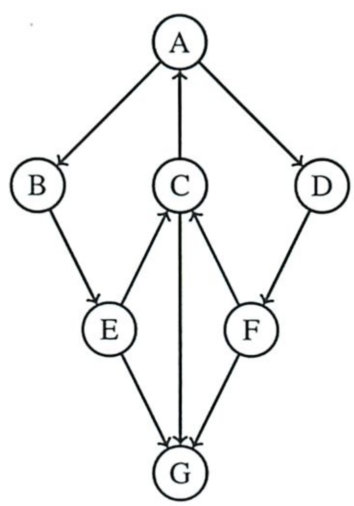
(a) What is the result of relaxing the following edges?

(i) 

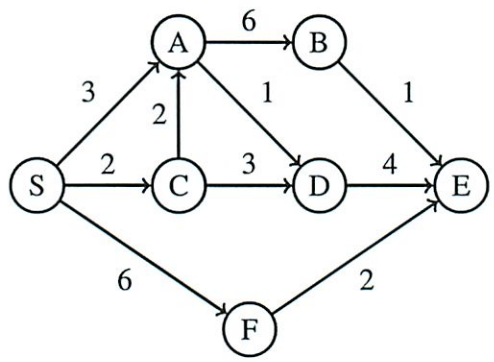
(ii) 

(iii) 

(b) Perform a depth-first search on the following graph starting at A. Label every edge in the graph with T if it's a tree edge, B if it's a back edge, F if it's a forward edge, and C if it's a cross edge. To ensure that your solution will be exactly the same as the staff solution, assume that whenever faced with a decision of which node to pick from a set of nodes, pick the node whose label occurs earliest in the alphabet.



(c) Run Dijkstra's algorithm on the following directed graph, starting at vertex S. What is the order in which vertices get removed from the priority queue? What is the resulting shortest-path tree?



(d) Radix sort the following list of integers in base 10 (smallest at top, largest at bottom). Show the resulting order after each run of counting sort.

Original list	First sort	Second sort	Third sort
583			
625			
682			
243			
745			
522			

Problem 4. Burgers would be great right about now [10 points]

Suppose that you want to get from vertex s to vertex t in an unweighted graph $G = (V, E)$, but you would like to stop by vertex u if it is possible to do so without increasing the length of your path by more than a factor of α.

Describe an efficient algorithm that would determine an optimal s-t path given your preference for stopping at u along the way if doing so is not prohibitively costly. (It should either return the shortest path from s to t or the shortest path from s to t containing u, depending on the situation.)

If it helps, imagine that there are burgers at u.

Problem 5. How I met your midterm [10 points]

Ted and Marshall are taking a roadtrip from Somerville to Vancouver (that's in Canada). Because it's a 52-hour drive, Ted and Marshall decide to switch off driving at each rest stop they visit; however, because Ted has a better sense of direction than Marshall, he should be driving both when they depart and when they arrive (to navigate the city streets).

Given a route map represented as a weighted undirected graph $G = (V, E, w)$ with positive edge weights, where vertices represent rest stops and edges represent routes between rest stops, devise an efficient algorithm to find a route (if possible) of minimum distance between Somerville and Vancouver such that Ted and Marshall alternate edges and Ted drives the first and last edge.

Problem 6. Just reverse the polarity already [10 points]

Professor Kirk has managed to get himself lost in his brand new starship. Furthermore, while boldly going places and meeting strange new, oddly humanoid aliens, his starship's engines have developed a strange problem: he can only make "transwarp jump" to solar systems at distance exactly 5 from his location.

Given a starmap represented as an unweighted undirected graph $G = (V, E)$, where vertices represent glorious new solar systems to explore and edges represent transwarp routes, devise an efficient algorithm to find a route (if possible) of minimum distance from Kirk's current location s to the location t representing Earth, that Kirk's ship will be able to follow. Please hurry—Professor Kirk doesn't want to miss his hot stardate!

Problem 7. The price is close enough [10 points]

As part of a new game show, contestants take turns making several integer guesses between 0 and 1,000,000 (inclusive). In scoring each round, the show's host, Professor Piotrik Kellmaine, needs to know which two guesses were closest to each other. Provide an asymptotically time-optimal algorithm that answers this question, argue that it is correct, and give and explain its time complexity.

Problem 8. Call it the scenic route [10 points]

In the ***longest path problem***, we're given a weighted directed graph $G = (V, E, w)$, a source $s \in V$, and we're asked to find the longest simple path from s to every vertex in G. For a general graph, it's not known whether there exists a polynomial-time algorithm to solve this problem. If we restrict G to be acyclic, however, this problem can be solved in polynomial time. Give an efficient algorithm for finding the longest paths from s in a weighted directed acyclic graph G, give its runtime, and explain why your solution doesn't work when G is not acyclic.

Problem 9. Rated M for "Masochistic" [10 points]

You're playing the hit new platform video game, *Mega Meat Man*, and are having trouble getting through Level 6006. You've decided to model the level as a directed graph, where each vertex represents a platform you can reach, and each edge represents a jump you can try to make. After extensive experimentation, you've labeled each edge with the probability (a number in $[0, 1]$) that you can successfully make the jump. Unfortunately, if you fail to make any jump, you instantly die, and have to start over. Describe an efficient algorithm to find a path from the start platform s to the goal platform t that maximizes the probability of a successful traversal.

SCRATCH PAPER

SCRATCH PAPER

Quiz 2 Solutions

Problem 1. True or false [24 points] (8 parts)

For each of the following questions, circle either T (True) or F (False). **Explain your choice.** (Your explanation is worth more than your choice of true or false.)

(a) T F Instead of using counting sort to sort digits in the radix sort algorithm, we can use any valid sorting algorithm and radix sort will still sort correctly.

Solution: False. Need stable sort.

(b) T F The depth of a breadth-first search tree on an undirected graph $G = (V, E)$ from an arbitrary vertex $v \in V$ is the diameter of the graph G. (The ***diameter*** d of a graph is the smallest d such that every pair of vertices s and t have $\delta(s, t) \leq d$.)

Solution: False. An arbitrary vertex could lay closer to the 'center' of the graph, hence the BFS depth will be underestimating the diameter. For example, in graph $G = (V, E) = (\{a, v, b\}, \{(a, v), (v, b)\})$, a BFS from v will have depth 1 but the graph has diameter 2.

(c) T F Every directed acyclic graph has exactly one topological ordering.

Solution: False. Some priority constraints may be unspecified, and multiple orderings may be possible for a given DAG. For example a graph $G = (V, E) = (\{a, b, c\}, \{(a, b), (a, c)\})$ has valid topological orderings $[a, b, c]$ or $[a, c, b]$. As another example, $G = (V, E) = (\{a, b\}, \{\})$ has valid topological orderings $[a, b]$ or $[b, c]$.

(d) T F Given a graph $G = (V, E)$ with positive edge weights, the Bellman-Ford algorithm and Dijkstra's algorithm can produce different shortest-path trees despite always producing the same shortest-path weights.

Solution: True. Both algorithms are guaranteed to produce the same shortest-path weight, but if there are multiple shortest paths, Dijkstra's will choose the shortest path according to the greedy strategy, and Bellman-Ford will choose the shortest path depending on the order of relaxations, and the two shortest path trees may be different.

(e) T F Dijkstra's algorithm may not terminate if the graph contains negative-weight edges.

Solution: False. It always terminates after $|E|$ relaxations and $|V|+|E|$ priority queue operations, but may produce incorrect results.

(f) T F Consider a weighted directed graph $G = (V, E, w)$ and let X be a shortest s-t path for $s, t \in V$. If we double the weight of every edge in the graph, setting $w'(e) = 2w(e)$ for each $e \in E$, then X will still be a shortest s-t path in (V, E, w').

Solution: True. Any linear transformation of all weights maintains all relative path lengths, and thus shortest paths will continue to be shortest paths, and more generally all paths will have the same relative ordering. One simple way of thinking about this is unit conversions between kilometers and miles.

(g) T F If a depth-first search on a directed graph $G = (V, E)$ produces exactly one back edge, then it is possible to choose an edge $e \in E$ such that the graph $G' = (V, E - \{e\})$ is acyclic.

Solution: True. Removing the back edge will result in a graph with no back edges, and thus a graph with no cycles (as every graph with at least one cycle has at least one back edge). Notice that a graph can have two cycles but a single back edge, thus removing *some* edge that disrupts that cycle is insufficient, you have to remove specifically the back edge. For example, in graph $G = (V, E) = (\{a, b, c\}, \{(a, b), (b, c), (a, c), (c, a)\})$, there are two cycles $[a, b, c, a]$ and $[a, c, a]$, but only one back edge (c, a). Removing edge (b, c) disrupts one of the cycles that gave rise to the back edge ($[a, b, c, a]$), but another cycle remains, $[a, c, a]$.

(h) T F If a directed graph G is cyclic but can be made acyclic by removing one edge, then a depth-first search in G will encounter exactly one back edge.

Solution: False. You can have multiple back edges, yet it can be possible to remove one edge that destroys all cycles. For example, in graph $G = (V, E) = (\{a, b, c\}, \{(a, b), (b, c), (b, a), (c, a)\})$, there are two cycles ($[a, b, a]$ and $[a, b, c, a]$) and a DFS from a in G returns two back edges ((b, a) and (c, a)), but a single removal of edge (a, b) can disrupt both cycles, making the resulting graph acyclic.

Problem 2. Short answer [24 points] (6 parts)

(a) What is the running time of RADIX-SORT on an array of n integers in the range $0, 1, \ldots, n^5 - 1$ when using base-10 representation? What is the running time when using a base-n representation?

Solution: Using base 10, each integer has $d = \log n^5 = 5 \log n$ digits. Each COUNTING-SORT call takes $\Theta(n + 10) = \Theta(n)$ time, so the running time of RADIX-SORT is $\Theta(nd) = \Theta(n \log n)$.

Using base n, each integer has $d = \log_n n^5 = 5$ digits, so the running time of RADIX-SORT is $\Theta(5n) = \Theta(n)$.

2 points were awarded for correct answers on each part. A point was deducted if no attempt to simplify running times were made (e.g. if running time for base-10 representation was left as $\Theta(\log_{10} n^5(n + 10))$

Common mistakes included substituting n^5 as the base instead of 10 or n. This led to $\Theta(n^5)$ and $\Theta(n^6)$ runtimes

(b) What is the running time of depth-first search, as a function of $|V|$ and $|E|$, if the input graph is represented by an adjacency matrix instead of an adjacency list?

Solution: DFS visits each vertex once and as it visits each vertex, we need to find all of its neighbors to figure out where to search next. Finding all its neighbors in an adjacency matrix requires $O(V)$ time, so overall the running time will be $O(V^2)$.

2 points were docked for answers that didn't give the tightest runtime bound, for example $O(V^2 + E)$. While technically correct, it was a key point to realize that DFS using an adjacency matrix doesn't depend on the number of edges in the graph.

(c) Consider the directed graph where vertices are reachable tic-tac-toe board positions and edges represent valid moves. What are the in-degree and out-degree of the following vertex? (It is O's turn.)

X	O	X
	O	
	X	

Solution: There were three possible vertices that could have pointed into this board position:

	O	X
	O	
	X	

X	O	
	O	
	X	

X	O	X
	O	

And there are four possible vertices that could have pointed out from this board position as O has four spaces to move to. In-degree is 3, out-degree is 4.

(d) If we modify the RELAX portion of the Bellman-Ford algorithm so that it updates $d[v]$ and $\pi[v]$ if $d[v] \geq d[u] + w(u, v)$ (instead of doing so only if $d[v]$ is strictly greater than $d[u] + w(u, v)$), does the resulting algorithm still produce correct shortest-path weights and a correct shortest-path tree? Justify your answer.

Solution: No. There exists a zero-weight cycle, then it is possible that relaxing an edge will mess up parent pointers so that it is impossible to recreate a path back to the source node. The easiest example is if we had a vertex v that had a zero-weight edge pointing back to itself. If we relax that edge, v's parent pointer will point back to itself. When we try to recreate a path from some vertex back to the source, if we go through v, we will be stuck there. The shortest-path tree is broken. 1 point was awarded for mentioning that shortest-path weights do get preserved, but also thinking the tree was correct..

(e) If you take 6.851, you'll learn about a priority queue data structure that supports EXTRACT-MIN and DECREASE-KEY on integers in $\{0, 1, \ldots, u-1\}$ in $O(\lg \lg u)$ time per operation. What is the resulting running time of Dijkstra's algorithm on a weighted direct graph $G = (V, E, w)$ with edge weights in $\{0, 1, \ldots, W-1\}$?

Solution: The range of integers that this priority queue data structure (van Emde Boas priority queue) will be from 0 to $|V|(W-1)$. This is because the longest possible path will go through $|V|$ edges of weight $W-1$. Almost the entire class substituted the wrong value for u. Dijkstra's will call EXTRACT-MIN $O(V)$ times and DECREASE-KEY $O(E)$ times. In total, the runtime of Dijkstra's using this new priority queue is $O((|V| + |E|) \lg \lg(|V|w))$

2 points were deducted for substituted the wrong u, but understanding how to use the priority queue's runtimes to get Dijkstra's runtime

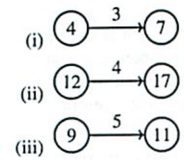
(f) Consider a weighted, directed acyclic graph $G = (V, E, w)$ in which edges that leave the source vertex s may have negative weights and all other edge weights are non-negative. Does Dijkstra's algorithm correctly compute the shortest-path weight $\delta(s, t)$ from s to every vertex t in this graph? Justify your answer.

Solution: Yes, For the correctness of Dijkstra, it is sufficient to show that $d[v] = \delta(s, v)$ for every $v \in V$ when v is added to s. Given the shortest $s \rightsquigarrow v$ path and given that vertex u precedes v on that path, we need to verify that u is in S. If $u = s$, then certainly u is in S. For all other vertices, we have defined v to be the vertex not in S that is closest to s. Since $d[v] = d[u] + w(u, v)$ and $w(u, v) > 0$ for all edges except possibly those leaving the source, u must be in S since it is closer to s than v.

It was not sufficient to state that this works because there are no negative weight cycles. Negative weight edges in DAGs can break Dijkstra's in general, so more justification was needed on why in this case Dijkstra's works.

Problem 3. You are the computer [12 points] (4 parts)

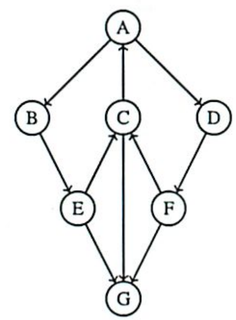
(a) What is the result of relaxing the following edges?



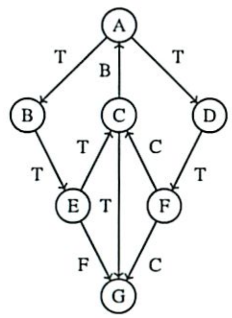
Solution: 7, 16, 11 for the new value of the right vertex

one point for each edge

(b) Perform a depth-first search on the following graph starting at A. Label every edge in the graph with T if it's a tree edge, B if it's a back edge, F if it's a forward edge, and C if it's a cross edge. To ensure that your solution will be exactly the same as the staff solution, assume that whenever faced with a decision of which node to pick from a set of nodes, pick the node whose label occurs earliest in the alphabet.



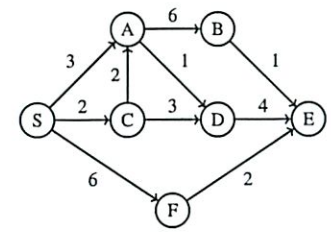
Solution:



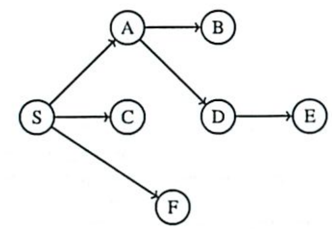
-1 for minor errors in labeling, sometimes resulting from incorrect choice of which node to visit

-2 for major errors in labeling

(c) Run Dijkstra's algorithm on the following directed graph, starting at vertex S. What is the order in which vertices get removed from the priority queue? What is the resulting shortest-path tree?



Solution: Dijkstra will visit the vertices in the following order: S, C, A, D, F, E, B. Dijkstra will relax the edge from D to E before the edge from F to E, since D is closer to S than F is. As a result, the parent of each node is:



-1 for minor errors, such as a missing vertex in the ordering of vertices removed from the priority queue or an incorrect edge in the shortest-path tree

-2 for major errors, such as not providing the shortest-path tree (some people mistakenly provided the shortest-path length in the tree)

(d) Radix sort the following list of integers in base 10 (smallest at top, largest at bottom). Show the resulting order after each run of counting sort.

Original list	First sort	Second sort	Third sort
583			
625			
682			
243			
745			
522			

Solution:

Original list	First sort	Second sort	Third sort
583	682	522	243
625	522	625	522
682	583	243	583
243	243	745	625
745	625	682	682
522	745	583	745

-1 for minor errors

-2 for major errors, such as not using a stable sort for the individual sorts.

Problem 4. Burgers would be great right about now [10 points]

Suppose that you want to get from vertex s to vertex t in an unweighted graph $G = (V, E)$, but you would like to stop by vertex u if it is possible to do so without increasing the length of your path by more than a factor of α.

Describe an efficient algorithm that would determine an optimal s-t path given your preference for stopping at u along the way if doing so is not prohibitively costly. (It should either return the shortest path from s to t or the shortest path from s to t containing u, depending on the situation.)

If it helps, imagine that there are burgers at u.

Solution: Since the graph is unweighted, one can use BFS for the shortest paths computation. We run BFS twice, once from s and once from u. The shortest path from s to t containing u is composed of the shortest path from s to u and the shortest path from u to t. We can now compare the length of this path to the length of the shortest path from s to t, and choose the one to return based on their lengths. The total running time is $O(V + E)$.

An alternative is to use Dijkstra algorithm. This works, but the algorithm becomes slower. Same for Bellman-Ford.

Problem 5. How I met your midterm [10 points]

Ted and Marshall are taking a roadtrip from Somerville to Vancouver (that's in Canada). Because it's a 52-hour drive, Ted and Marshall decide to switch off driving at each rest stop they visit; however, because Ted has a better sense of direction than Marshall, he should be driving both when they depart and when they arrive (to navigate the city streets).

Given a route map represented as a weighted undirected graph $G = (V, E, w)$ with positive edge weights, where vertices represent rest stops and edges represent routes between rest stops, devise an efficient algorithm to find a route (if possible) of minimum distance between Somerville and Vancouver such that Ted and Marshall alternate edges and Ted drives the first and last edge.

Solution: There are two correct and efficient ways to solve this problem. The first solution makes a new graph G'. For every vertex u in G, there are two vertices u_M and u_T in G': these represent reaching the rest stop u when Marshall (for u_M) or Ted (for u_T) will drive next. For every edge (u, v) in G, there are two edges in G': (u_M, v_T) and (u_T, v_M). Both of these edges have the same weight as the original.

We run Dijkstra's algorithm on this new graph to find the shortest path from Somerville$_T$ to Vancouver$_M$ (since Ted drives *to* Vancouver, Marshall would drive next if they continued). This guarantees that we find a path where Ted and Marshall alternate, and Ted drives the first and last segment. Constructing this graph takes linear time, and running Dijkstra's algorithm on it takes $O(V \log V + E)$ time with a Fibonacci heap (it's just a constant factor worse than running Dijkstra on the original graph).

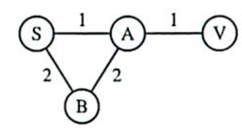
The second correct solution is equivalent to the first, but instead of modifying the graph, we modify Dijkstra's algorithm. Dijkstra's algorithm will store two minimum distances and two parent pointers for each vertex u: the minimum distance d_{odd} using an odd number of edges, and the minimum distance d_{even} using an even number of edges, along with their parent pointers π_{odd} and π_{even}. (These correspond to the minimum distance and parent pointers for u_T and u_M in the previous solution). In addition, we put each vertex in the priority queue twice: once with d_{odd} as its key, and once with d_{even} as its key (this corresponds to putting both u_T and u_M in the priority queue in the previous solution).

When we relax edges in the modified version of Dijkstra, we check whether $v.d_{\text{odd}} > u.d_{\text{even}} + w(u, v)$, and vice versa. One important detail is that we need to initialize Somerville.d_{odd} to ∞, not 0. This algorithm has the same running time as the previous one.

A correct but less efficient algorithm used Dijkstra, but modified it to traverse two edges at a time on every step except the first, to guarantee a path with an odd number of edges was found. Many students incorrectly claimed this had the same running time as Dijkstra's algorithm; however, computing all the paths of length 2 (this is the ***square*** of the graph G) actually takes a total of $O(VE)$ time, whether you compute it beforehand or compute it for each vertex when you remove it from Dijkstra's priority queue. This solution got 5 points.

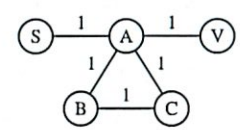
The most common mistake on the problem was to augment Dijkstra (or Bellman-Ford) by keeping track of either the shortest path's edge count for each vertex, or the parity of the number edges in

the shortest path. This is insufficient to guarantee that the shortest odd-edge-length path is found, and this solution got 2 points. Here is an example of a graph where the algorithm fails: once the odd-edge-count path of weight 1 to A is found, Dijkstra will ignore the even-edge-count path of weight 4 to A since it has greater weight. As a result, the odd-edge-count path to V will be missed entirely.



Another common mistake was to use Dijkstra, and if the path Dijkstra found had an even number of edges, to attempt to add or remove edges until a path with an odd number of edges was obtained. In general, there is no guarantee the shortest path with an odd number of edges is at all related to the shortest path with an even number of edges.

Some algorithms ran Dijkstra, and if Dijkstra found a path with an even number of edges, removed some edge or edges from the graph and re-ran Dijkstra. This algorithm fails on the following graph, where the shortest path with an odd number of edges uses *all* the edges and vertices (note that we visit A twice; the first time, Ted drives to A, and the second time, Marshall drives to A):



One last common mistake was to attempt to use Breadth-First Search to label each vertex as an odd or even number of edges from Somerville (or sometimes to label them as odd, even, or both). This does not help: the smallest-weight path with an odd number of edges could go through any particular vertex after having traversed an odd or even number of edges, and BFS will not correctly predict which. These solutions got 0 points.

Algorithms which returned the correct answer but with exponential running time got *at most* 2 points.

Problem 6. Just reverse the polarity already [10 points]

Professor Kirk has managed to get himself lost in his brand new starship. Furthermore, while boldly going places and meeting strange new, oddly humanoid aliens, his starship's engines have developed a strange problem: he can only make "transwarp jump" to solar systems at distance exactly 5 from his location.

Given a starmap represented as an unweighted undirected graph $G = (V, E)$, where vertices represent glorious new solar systems to explore and edges represent transwarp routes, devise an efficient algorithm to find a route (if possible) of minimum distance from Kirk's current location s to the location t representing Earth, that Kirk's ship will be able to follow. Please hurry—Professor Kirk doesn't want to miss his hot stardate!

Solution: In general, the idea is to convert $G = (V, E)$ into a graph $G' = (V, E')$ representing all the feasible transwarp jumps that Kirk can make, i.e., with an edge (u, v) if there is a simple path in G from u to v of length exactly 5. (Note that this definition is the notion of "distance" in the problem, as clarified during the quiz.) Once we have such a graph G', we simply run breadth-first search on G' from s, and follow parent pointers from t to recover the shortest route (if there one) for Kirk to follow. The running time of this breadth-first search is $O(V + E') = O(V^2)$.

The central question is how to compute G'. The best solutions we know run in $O(V^3)$ time. There are two ways to achieve this bound.

The first $O(V^3)$ algorithm is a modification of breadth-first search from every vertex. For each vertex v, we construct the set $N_1(v)$ of all neighbors of v. Next we construct the set $N_2(v)$ of all vertices reachable by a path of length 2 from v, by taking the union of $N_1(u)$ for each $u \in N_1(v)$. Then we construct $N_3(v)$, $N_4(v)$, and $N_5(v)$ similarly. Constructing $N_1(v)$ costs $O(V)$ time, while constructing $N_k(v)$ for $k \in \{2, 3, 4, 5\}$ costs $O(v^2)$ time. The key here is that we remove duplicate vertices in each set $N_k(v)$, so each such set has size $O(V)$. Because we do this for every vertex v, we spend $O(v^3)$ time total. Finally we set $E' = \{(v, w) : w \in N_5(v)\}$.

The second $O(V^3)$ algorithm is to compute the adjacency matrix A, and compute $A^5 = A \cdot A \cdot A \cdot A \cdot A$. Each matrix multiplication costs $O(V^3)$ time, for a total of $O(V^3)$ time. The nonzero entries in this matrix correspond to the edges in G'.

A simpler $O(V^4E) = O(V^6)$ algorithm is much simpler: for each vertex v_0, for each neighbor v_1 of v_0, for each neighbor v_2 of v_1, for each neighbor v_3 of v_2, for each neighbor v_4 of v_3, for each neighbor v_5 of v_4, add the edge (v_0, v_5) to E'. The first two loops cost a factor of $O(E)$, and the next four loops cost a factor of $O(V^4)$.

The grading scheme was as follows. A $O(V^3)$ solution was worth a nominal value of 10/10. A $O(V^4)$ solution was worth a nominal value of 9/10. Very few students achieved such solutions. A $O(V^4E)$ or $O(V^6)$ solution was worth a nominal value of 7/10. These nominal values were adjusted according to clarity, quality, and/or errors. The idea of computing a graph like G' was worth a nominal value of 4/10. Executing this idea by performing a depth-5 BFS or DFS was worth a nominal value of 5/10. Simply running BFS and focusing on the layers divisible by 5 was worth a nominal value of 1/10.

Problem 7. The price is close enough [10 points]

As part of a new game show, contestants take turns making several integer guesses between 0 and 1,000,000 (inclusive). In scoring each round, the show's host, Professor Piotrik Kellmaine, needs to know which two guesses were closest to each other. Provide an asymptotically time-optimal algorithm that answers this question, argue that it is correct, and give and explain its time complexity.

Solution: algorithm: We first radix sort the input n guesses using base 10. Then we go through the list of n sorted integers and compare adjacent ones to see which pair of adjacent integers are closest to each other, and output that pair of gusses.

correctness: We see that the closest pair of guesses have to be adjacent to each other in the sorted list because or else there will be some integers in between them making them not the closest pair. In other word, say a and b are the closest pair, then if $a < c < b$, we see $b - c$ and $c - a$ are less than $b - a$, therefore contradicting the fact that a and b are the closest pair of guesses.

runtime: radix sort takes $O(7 \cdot (n + 10))$ time if we take base 10 which is $O(n)$ time. Going through the list once and compare all adjacent pairs only take $O(n)$ time because there are only $n - 1$ pairs we have to compare and find the minimum absolute difference between them. So the total running time is $O(n)$.

grading: one point is taken off for not mentioning the base of radix sort or using counting sort instead because 1000000 is a relatively big constant factor in the case of this problem. three points are taken off if students did not present an explanation on how to iterate through the sorted list to find the min difference.
three points are given if the student gave the naive algorithm which takes all $\frac{(n)(n-1)}{2}$ pairs and find the minimum. four points are given for students who choose a sorting algorithm that takes $O(n \lg n)$ time.

Problem 8. Call it the scenic route [10 points]

In the ***longest path problem***, we're given a weighted directed graph $G = (V, E, w)$, a source $s \in V$, and we're asked to find the longest simple path from s to every vertex in G. For a general graph, it's not known whether there exists a polynomial-time algorithm to solve this problem. If we restrict G to be acyclic, however, this problem can be solved in polynomial time. Give an efficient algorithm for finding the longest paths from s in a weighted directed acyclic graph G, give its runtime, and explain why your solution doesn't work when G is not acyclic.

Solution: **Algorithm:** We map this to a single-source shortest paths problem by creating a new graph, G', with the same vertices and edges as G but whose weight function is the negative of the original.

Now we can run the single-source shortest paths algorithm for DAG's shown in class to find the shortest paths in $\Theta(V + E)$. This algorithm relaxes the edges of G' in topologically sorted order only once. See class notes to see why this works for finding the shortest paths in a DAG.

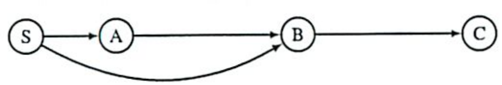
We could alternatively use Bellman Ford here, although that will give us a suboptimal runtime.

Runtime: Creating G' is a simple process and only requires $\Theta(V + E)$ time to iterate over all the edges and vertices to create our new graph and weight function. Topologically sorting the edges takes $\Theta(V + E)$ since topological sort is done using a modification of the DFS algorithm. Finally, relaxing all the edges once only takes $\Theta(E)$ time. Thus, the runtime is $\Theta(V + E)$ overall.

Why G needs to be acyclic: We can't use the single-source shortest paths algorithm for DAG's if G is not acyclic since we would no longer have a DAG. But, even assuming we used Bellman Ford, which can handle negative weight cycles, we would still be in trouble. The main reason we need G to be acyclic is that we're looking for the longest **simple** path (i.e. no vertex is repeated). Negative weight cycles in G' wouldn't be much of an issue if we didn't restrict our paths to be simple. Simply detecting them and marking those paths as infinite is easy to do in asymptotically the same time as Bellman Ford.

Grading: Overall, 6 points were given to the algorithm and 4 points were given to the explanation of why we need G to be acyclic.

Many students tried BFS or DFS, both of only work on unweighted graphs. Another large portion of students attempted to use Dijkstra or a modified Dijkstra algorithm. The problem with a Dijkstra approach is that Dijkstra for shortest paths relies on the fact that once we visit a vertex, we wont ever find a shorter path to that vertex. This requires non neg. edge weights, however. So, in the longest path problem, we would need all neg. edge weights in order to be able to have a similar invariant. But, there's nothing in the problem statement that allows us to make this assumption. If you're still skeptical, here's a counterexample to the Dijkstra approaches seen:



Dijkstra will not find the longest path from S to C.

Otherwise, the majority of students did not give an adequate explanation to why the graph needs to be acyclic. We were mainly looking for some comment about the problem specifying **simple** paths, since that's at the heart of the matter. Any solution that didn't mention this, or touch on something close to this, lost credit.

Finally, while Bellman Ford is a correct approach, it is not optimal. Only a point was docked for this.

Problem 9. Rated M for "Masochistic" [10 points]

You're playing the hit new platform video game, *Mega Meat Man*, and are having trouble getting through Level 6006. You've decided to model the level as a directed graph, where each vertex represents a platform you can reach, and each edge represents a jump you can try to make. After extensive experimentation, you've labeled each edge with the probability (a number in [0, 1]) that you can successfully make the jump. Unfortunately, if you fail to make any jump, you instantly die, and have to start over. Describe an efficient algorithm to find a path from the start platform s to the goal platform t that maximizes the probability of a successful traversal.

Solution: Intuitively, we'd like to maximize $\prod_i p_i$ over the vertices in the path we take from s to t. Since the log function is monotonic, this is the same as maximizing $\sum_i \log p_i$, which is the same as minimizing $-\sum_i \log p_i = \sum_i (-\log p_i)$. Therefore, if we create an auxiliary graph in which the weight w of each edge is replaced with $-\log w$, the shortest s-t path is the maximum probability path. Additionally, $p_i \in [0, 1] \implies \log p_i \leq 0 \implies -\log p_i \geq 0$, so all edge weights are nonnegative. The negative logarithm goes to ∞ as p_i goes to 0, which suits us just fine; if we never make the jump, we should never try that path. Because all of our edge weights are nonnegative, we can use Dijkstra to find the shortest s-t path; the creation of the auxiliary graph takes $O(|E|)$ time, so the total time complexity of the algorithm is $O(|E| + |V| \lg |V|)$ (if we use a Fibonnaci heap).

Solutions that did not provide the time complexity of the algorithm or that used a less efficient algorithm, solutions that did not convincingly argue that edge weights were nonnegative (while using Dijkstra), and solutions that did not convincingly argue that the shortest s-t path in the auxiliary graph corresponded to the solution to the original problem lost points. Some solutions tried to modify Dijkstra instead of reducing the problem given to a standard shortest-path problem. This itself did not cause any loss of credit, but often led to mistakes (subtle or otherwise) or a lack of clarity.

Quiz 2

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- When the quiz begins, write your name on every page of this quiz booklet.
- You have 120 minutes to earn 120 points. Do not spend too much time on any one problem. Read them all through first, and attack them in the order that allows you to make the most progress.
- This quiz is closed book. You may use **two** $8\frac{1}{2}'' \times 11''$ or A4 crib sheet (both sides). No calculators or programmable devices are permitted. No cell phones or other communications devices are permitted.
- Write your solutions in the space provided. If you need more space, write on the back of the sheet containing the problem. Pages may be separated for grading.
- Do not waste time and paper rederiving facts that we have studied. It is sufficient to cite known results.
- When writing an algorithm, a **clear** description in English will suffice. Pseudo-code is not required.
- When asked for an algorithm, your algorithm should have the time complexity specified in the problem with a correct analysis. If you cannot find such an algorithm, you will generally receive partial credit for a slower algorithm **if you analyze your algorithm correctly**.
- Show your work, as partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. **This quiz is shorter than the first, so we expect you to take the time to write clear and thorough solutions.**
- Good luck!

Problem	Parts	Points	Grade	Grader
1	2	2		
2	4	38		
3	2	20		
4	1	20		
5	3	20		
6	1	20		
Total		120		

Name: ______________________________

Friday Recitation:	Aleksander **11 AM**	Arnab **12 PM**	Alina **3 PM**	Matthew **4 PM**

Problem 1. What is Your Name? [2 points] (2 parts)

(a) [1 point] Flip back to the cover page. Write your name there.

(b) [1 point] Flip back to the cover page. Circle your recitation section.

Problem 2. Short Answer [38 points] (4 parts)

(a) [9 points] Give an example of a graph such that running Dijkstra on it would give incorrect distances.

(b) [9 points] Give an efficient algorithm to sort n dates (represented as month-day-year and all from the 20^{th} century), and analyze the running time.

(c) [10 points] Give an $O(V + E)$-time algorithm to remove all the cycles in a directed graph $G = (V, E)$. Removing a cycle means removing an edge of the cycle. If there are k cycles in G, the algorithm should only remove $O(k)$ edges.

(d) [10 points] Let $G = (V, E)$ be a weighted, directed graph with exactly one negative-weight edge and no negative-weight cycles. Give an algorithm to find the shortest distance from s to all other vertices in V that has the same running time as Dijkstra.

Problem 3. Path Problems [20 points] (2 parts)

We are given a directed graph $G = (V, E)$, and, for each edge $(u, v) \in E$, we are given a probability $f(u, v)$ that the edge may fail. These probabilities are independent. The reliability $\pi(p)$ of a path $p = (u_1, u_2, \ldots u_k)$ is the probability that no edge fails in the path, i.e.
$\pi(p) = (1 - f(u_1, u_2)) \cdot (1 - f(u_2, u_3)) \ldots \cdot (1 - f(u_{k-1}, u_k))$. Given a graph G, the edge failure probabilities, and two vertices $s, t \in V$, we are interested in finding a path from s to t of maximum reliability.

(a) [10 points] Propose an efficient algorithm to solve this problem. Analyze its running time.

(b) [10 points] You tend to be risk-averse and in addition to finding a most reliable simple path from s to t, you also want to find a next-most reliable simple path, and output these two paths. Propose an algorithm to solve the problem, argue its correctness, and give its asymptotic running time.

Problem 4. Flight Plans [20 points]

When an airline is compiling flight plans to all destinations from an airport it serves, the flight plans are plotted through the air over other airports in case the plane needs to make an emergency landing. In other words, flights can be taken only along pre-defined edges between airports. Two airports are adjacent if there is an edge between them. The airline also likes to ensure that all the airports along a flight plan will be no more than three edges away from an airport that the airline regularly serves.

Given a graph with V vertices representing all the airports, the subset W of V which are served by the airline, the distance $w(u, v)$ for each pair of adjacent airports u, v, and a base airport s, give an algorithm which finds the shortest distance from s to all other airports, with the airports along the path never more than 3 edges from an airport in W.

Problem 5. Tree Searches [20 points] (3 parts)

In this problem we consider doing a depth first search of a perfect binary search tree B. In a perfect binary search tree a node p can have either 2 or 0 children (but not just one child) with the usual requirement that any node in the left subtree of p is less than p and node in the right subtree is greater than p. In addition, all nodes with no children (leaves) must be at the same level of the tree. To make B into a directed graph, we consider the nodes of B to be the vertices of the graph. For each node p, we draw a directed edge from p to its left child and from p to its right child. An example of a perfect binary search tree represented as a graph is shown in Figure 1.

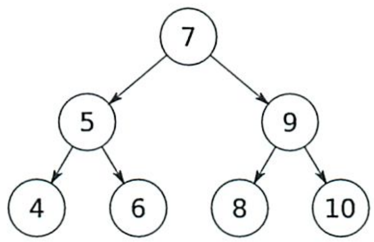


Figure 1: An example of a perfect binary search tree represented as a directed graph.

(a) [6 points] We structure our adjacency function such that at a node p, we first run DFS-VISIT on the left child of p and then on the right child. When we have finished expanding a node (i.e. just before we return from DFS-VISIT), we print the node. What is the first node printed? What is the last node printed? Give a short defense of your answer.

(b) [7 points] Does DFS print out the nodes of the tree in increasing or decreasing order? If yes, give a proof. If no, give a small counter example where the algorithm fails to print out the nodes in increasing or decreasing order and show the output of DFS on your example.

(c) [7 points] Recall that usually when doing depth first search, we use the *parent* structure to keep track of which vertices have been visited. During the search, if a vertex v is in *parent*, the search will not run DFS-VISIT(v) again. Aspen Tu declares that *parent* is unnecessary when doing a DFS of B. She says that whenever the algorithm checks if a vertex v is in *parent*, the answer is always false. Do you agree with Aspen? If you do, prove that she is correct. If you do not, give a small counter-example where a depth first search through B will see a vertex twice. Remember, B is a directed graph.

Problem 6. Computing Minimum Assembly Time [20 points]

As you might have heard, NASA is planning on deploying a new generation of space shuttles. Part of this project is creating a schedule according to which the prototype of the space shuttle will be assembled.

The assembly is broken down into atomic actions – called *jobs* – that have to be performed to build the prototype. Each job has a *processing time* and a (possibly empty) set of *required jobs* that need to be completed before this job can start – we will refer to this set as *precedence constraint.* Given such specification, we call an assembly schedule *valid* if it completes all the jobs and all the precedence constraints are satisfied.

Now, as the plan of the whole undertaking is being finalized, NASA has to compute the *minimum assembly time* of the prototype. This time is defined as the minimum, taken over all the valid assembly schedules, of the time that passes since the processing of the first scheduled job starts until the processing of the last job finishes. (Note that we allow jobs to be processed in parallel, as long as their precedence constraints are satisfied.)

As the prototype assembly is an immensely complex task, can you help NASA by designing an algorithm that computes the minimum assembly time efficiently? Prove the correctness of your algorithm and analyze its running time in terms of the number of jobs n and the total length of the required jobs lists m.

Formally, the assembly is presented as a list of n jobs $J_1, \ldots, J_n$, and each job J has a specified processing time, and the set of required jobs. We assume that there always is at least one valid assembly schedule corresponding to the given specification.

Example:

Job:	Processing time:	Required jobs:
J_1	1	$\{J_6, J_7\}$
J_2	6	$\emptyset$
J_3	4	$\{J_2, J_5\}$
J_4	2	$\{J_2, J_3\}$
J_5	3	$\emptyset$
J_6	5	$\emptyset$
J_7	7	$\emptyset$

Here, $n = 7$ and $m = 6$.

Solution: The minimum assembly time is 12.

(The corresponding schedule starts jobs J_2, J_5, J_6, J_7 at time 0, J_3 at time 6, J_1 at time 7, and J_4 at time 10.)

SCRATCH PAPER

SCRATCH PAPER

Quiz 2 Solutions

Problem 1. What is Your Name? [2 points] (2 parts)

(a) [1 point] Flip back to the cover page. Write your name there.

(b) [1 point] Flip back to the cover page. Circle your recitation section.

Problem 2. Short Answer [38 points] (4 parts)

(a) [9 points] Give an example of a graph such that running Dijkstra on it would give incorrect distances.

Solution: Below is one example of such a graph. There needs to be a vertex u such that when it is extracted, the distance to it is not the weight of the shortest path. But this alone is not enough: there needs to be a vertex v adjacent to u whose shortest path is through u. Since the edges from u get relaxed only once, then even though the distance to u could later be updated to the correct shortest distance, the distance to v will not be. Dijkstra will also yield incorrect distances for a graph with a negative-weight cycle.

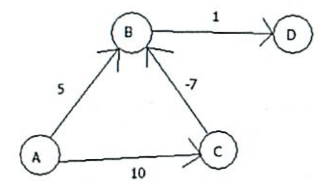


Figure 1: A gets extracted first, after which edges (A, B) and (A, C) are relaxed, and the distances are $d[A] = 0, d[B] = 5, d[C] = 10$. B is extracted next, leading to edge (B, D) being relaxed, and $d[D]$ becomes 6. D is extracted next, but it has no edges to relax. Finally, C is extracted, relaxing edge (C, B) and making $d[B] = 3$. The shortest path to D has weight 4, but $d[D] = 6$.

(b) [9 points] Give an efficient algorithm to sort n dates (represented as month-day-year and all from the 20^{th} century), and analyze the running time.

Solution: Use radix sort. First sort by day using counting sort with an array of size 31, then sort by month using counting sort with an array of size 12, and finally sort by year using counting sort with an array of size 100, where the counter in slot i corresponds to year $1900 + i$. The running time of radix sort is $\Theta(d(n + k))$. In this case, $d = 3$ and k is maximum 100, so the running time is $\Theta(n)$.

(c) [10 points] Give an $O(V + E)$-time algorithm to remove all the cycles in a directed graph $G = (V, E)$. Removing a cycle means removing an edge of the cycle. If there are k cycles in G, the algorithm should only remove $O(k)$ edges.

Solution: Do a DFS of the graph, and at the end, remove all the back edges. As you traverse the graph, you can check whether the edge you are trying to relax goes to a node that has been seen but is not yet finished, and if so, then it is a back edge and you can store it in a set. After the DFS, remove all the edges that are in this set.

(d) [10 points] Let $G = (V, E)$ be a weighted, directed graph with exactly one negative-weight edge and no negative-weight cycles. Give an algorithm to find the shortest distance from s to all other vertices in V that has the same running time as Dijkstra.

Solution: Let's say the negative-weight edge is (u, v). First, remove the edge and run Dijkstra from s. Then, check if $d_s[u] + w(u, v) < d_s[v]$. If not, then we're done. If yes, then run Dijkstra from v, with the negative-weight edge still removed. Then, for any node t, its shortest distance from s will be $min(d_s[t], d_s[u] + w(u, v) + d_v[t])$.

Problem 3. Path Problems [20 points] (2 parts)

We are given a directed graph $G = (V, E)$, and, for each edge $(u, v) \in E$, we are given a probability $f(u, v)$ that the edge may fail. These probabilities are independent. The reliability $\pi(p)$ of a path $p = (u_1, u_2, \ldots u_k)$ is the probability that no edge fails in the path, i.e. $\pi(p) = (1 - f(u_1, u_2)) \cdot (1 - f(u_2, u_3)) \ldots \cdot (1 - f(u_{k-1}, u_k))$. Given a graph G, the edge failure probabilities, and two vertices $s, t \in V$, we are interested in finding a path from s to t of maximum reliability.

(a) [10 points] Propose an efficient algorithm to solve this problem. Analyze its running time.

Solution: Since the logarithm is a monotonic increasing function, maximizing the reliability $\pi(p) = (1 - f(u_1, u_2))(1 - f(u_1, u_2)) \cdots (1 - f(u_{k-1}, u_k))$ of a path is equivalent to maximizing $\log \pi(p) = \log(1 - f(u_1, u_2)) + \log(1 - f(u_1, u_2)) + \cdots + \log(1 - f(u_{k-1}, u_k))$, equivalent to minimizing $-\log \pi(p)$. Assign each edge a weight $w(u, v) = -\log(1 - f(u, v))$. These weights are all non-negative - and so we can apply Dijkstra.

Alternatively, simply modify the Dijkstra's algorithm (appropriately defining and initializing $d[u]$, replacing extract-min by extract-max, and using the relaxation step "if $d[v] < d[u](1 - f(u, v))$, then $d[v] = d[u](1 - f(u, v))$". These modifications work since $0 \leq f(u, v) \leq 1$ for all edge $(u, v) \in E$.

(b) [10 points] You tend to be risk-averse and in addition to finding a most reliable simple path from s to t, you also want to find a next-most reliable simple path, and output these two paths. Propose an algorithm to solve the problem, argue its correctness, and give its asymptotic running time.

Solution: We are not asking for a most efficient algorithm, simply a correct one. First notice that if the graph has no more than one simple path from s to t, then the problem has no next-most reliable simple path, and our algorithm should indicate this. We first found a most reliable simple path from s to t, if one exists. A next most reliable simple path must differ from it by at least one edge. So repeatedly resolve the problem after removing each edge of the initial path from G, one at a time, and chose among all these solutions the one that maximizes the reliability (if for each edge removal s is not connected to t anymore, the algorithm output "no next-most reliable path from s to t). This algorithm works since it will find a next-most reliable simple path that has to differ from the first one. It takes up to $k \leq n - 1$ iterations of Dijkstra, where k is the number of edges in the initial most reliable path.

Problem 4. Flight Plans [20 points]

When an airline is compiling flight plans to all destinations from an airport it serves, the flight plans are plotted through the air over other airports in case the plane needs to make an emergency landing. In other words, flights can be taken only along pre-defined edges between airports. Two airports are adjacent if there is an edge between them. The airline also likes to ensure that all the airports along a flight plan will be no more than three edges away from an airport that the airline regularly serves.

Given a graph with V vertices representing all the airports, the subset W of V which are served by the airline, the distance $w(u, v)$ for each pair of adjacent airports u, v, and a base airport s, give an algorithm which finds the shortest distance from s to all other airports, with the airports along the path never more than 3 edges from an airport in W.

Solution: As written, this problem asked that all nodes in the paths be within 3 edges of a node in W. So, we can solve this in two steps. First, we eliminate the nodes that are further than 3 edges away from a node in W. The most efficient way to do this is to create a supernode connected to all nodes in W and then run BFS to only four levels, eliminating all nodes not encountered. Alternately, using BFS the algorithm could run BFS as normal but start with a queue filled with all nodes in W. Other slower options include running BFS from every node in W or running Bellman-Ford with edge weights of 1 and nodes in W with starting weight 0. After eliminating the nodes which are further than 3 edges from a node in W, we can just run Dijkstra as normal. The running time of BFS in $O(V + E)$ is overtaken by Dijkstra's running time of $O(E + V \log V)$ giving that as the total. Solutions suggesting using multiple runs of BFS but skipping already-visited nodes from previous runs were invalid because the already-visited node may be reached at a shorter number of edges from a node in W, allowing more of its children to be included in the graph. Similarly, an algorithm just using Dijkstra but also tracking the distance from a node in W while running Dijkstra fails because there may have been a longer earlier path which would have run through a node in W.

Another interpretation of this question which was also accepted was that every node in the path must be within 3 edges of a node in W which is also in the same path. In this case, valid solutions used a graph transformation, making copies of each node for edge counts away from W, with each edge linking either to a higher-distance node or back to 0 if the node was in W. Similarly, another valid solution to this interpretation was to keep track of the shortest path thus far for each valid edge count away from W.

Problem 5. Tree Searches [20 points] (3 parts)

In this problem we consider doing a depth first search of a perfect binary search tree B. In a perfect binary search tree a node p can have either 2 or 0 children (but not just one child) with the usual requirement that any node in the left subtree of p is less than p and node in the right subtree is greater than p. In addition, all nodes with no children (leaves) must be at the same level of the tree. To make B into a directed graph, we consider the nodes of B to be the vertices of the graph. For each node p, we draw a directed edge from p to its left child and from p to its right child. An example of a perfect binary search tree represented as a graph is shown in Figure 2.

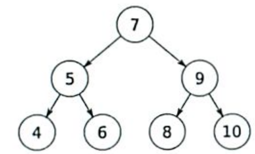


Figure 2: An example of a perfect binary search tree represented as a directed graph.

(a) [6 points] We structure our adjacency function such that at a node p, we first run DFS-VISIT on the left child of p and then on the right child. When we have finished expanding a node (i.e. just before we return from DFS-VISIT), we print the node. What is the first node printed? What is the last node printed? Give a short defense of your answer.

Solution: The first node printed will be the smallest node in the tree because DFS goes all the way down the tree before finally returning from a DFS-VISIT so that the first node it prints is in the bottom row. Since DFS first visits the left children, this will be the leftmost node in the bottom row. Since the bottom row is full, this node is the left child of its parent (which was the left child of its parent, etc) so it is the smallest node in the tree.

The last node printed will be the root node since this is the last node DFS finishes. Since the tree is perfectly binary, the root node is also the median of the tree.

Grading:

- 6/6: For something like the above answer. If you just did it on the example tree you received full credit provided you gave an explanation that showed you understood the order in which DFS expands node.
- 3/6: If got only one of the two right.

(b) [7 points] Does DFS print out the nodes of the tree in increasing or decreasing order? If yes, give a proof. If no, give a small counter example where the algorithm fails to print out the nodes in increasing or decreasing order and show the output of DFS on your example.

Solution: For the tree shown in Figure 2, DFS prints out

$$4, 6, 5, 8, 10, 9, 7$$

Grading:

- 7/7: For any counter-example
- 5/7: For a counter example that was not a perfect binary search tree.
- 4/7: If you showed a counter-example where DFS does not print out the nodes in order, but gave the wrong order or failed to give the output.
- 2/7: If you said DFS does not print out the nodes in order but gave no counter-example.
- 1-2/7: If you said DFS prints out the nodes in order, but gave a reasonable justification for why you might think that.
- 0/7: If you said DFS prints out the nodes in order and gave no justification.

(c) [7 points] Recall that usually when doing depth first search, we use the *parent* structure to keep track of which vertices have been visited. During the search, if a vertex v is in *parent*, the search will not run DFS-VISIT(v) again. Aspen Tu declares that *parent* is unnecessary when doing a DFS of B. She says that whenever the algorithm checks if a vertex v is in *parent*, the answer is always false. Do you agree with Aspen? If you do, prove that she is correct. If you do not, give a small counter-example where a depth first search through B will see a vertex twice. Remember, B is a directed graph.

Solution: Aspen is correct. Each node has only one incoming edge. When doing a DFS on a directed graph, we traverse each edge only once. Therefore, we can only see each node once in the search. You could also say that the search only produces tree edges and seeing a node twice requires a cross, forward, or back edge.

If you assumed the search did not start from the root then you do need the parent structure. This answer with correct justification also received full credit.

This question may have been confusingly worded. *Every* seen node is put into the *parents* data structure; not just the parents on the current path. This should have been clear from context, but naming the structure *parents* was a little misleading. It was done this way because that is how it was shown in lecture in class.

Grading:

- 7/7: For a correct answer with a good justification.
- 5/7: For saying the tree is a DAG but making it clear that you thought *parent* just stored current parents on the path. This is not true and we were trying to use the notation from class, but it was misleading.
- 4/7: For saying the tree is a DAG. This *does not* show you will not see a node twice. DAGs can have forward edges.
- 3/7 if you showed an example in which DFS would see a node twice that was not a perfect balanced binary search tree.
- 2/7: For the correct answer with no or very little justification.
- 1-3/7: For the wrong answer, but a justification that shows some knowledge of how DFS works.
- 0/7: For the wrong answer and no justification.

Problem 6. Computing Minimum Assembly Time [20 points]

As you might have heard, NASA is planning on deploying a new generation of space shuttles. Part of this project is creating a schedule according to which the prototype of the space shuttle will be assembled.

The assembly is broken down into atomic actions – called *jobs* – that have to be performed to build the prototype. Each job has a *processing time* and a (possibly empty) set of *required jobs* that need to be completed before this job can start – we will refer to this set as *precedence constraint.* Given such specification, we call an assembly schedule *valid* if it completes all the jobs and all the precedence constraints are satisfied.

Now, as the plan of the whole undertaking is being finalized, NASA has to compute the *minimum assembly time* of the prototype. This time is defined as the minimum, taken over all the valid assembly schedules, of the time that passes since the processing of the first scheduled job starts until the processing of the last job finishes. (Note that we allow jobs to be processed in parallel, as long as their precedence constraints are satisfied.)

As the prototype assembly is an immensely complex task, can you help NASA by designing an algorithm that computes the minimum assembly time efficiently? Prove the correctness of your algorithm and analyze its running time in terms of the number of jobs n and the total length of the required jobs lists m.

Formally, the assembly is presented as a list of n jobs $J_1, \ldots, J_n$, and each job J has a specified processing time, and the set of required jobs. We assume that there always is at least one valid assembly schedule corresponding to the given specification.

Example:

Job:	Processing time:	Required jobs:
J_1	1	$\{J_6, J_7\}$
J_2	6	$\emptyset$
J_3	4	$\{J_2, J_5\}$
J_4	2	$\{J_2, J_3\}$
J_5	3	$\emptyset$
J_6	5	$\emptyset$
J_7	7	$\emptyset$

Here, $n = 7$ and $m = 6$.

Solution: The minimum assembly time is 12.

(The corresponding schedule starts jobs J_2, J_5, J_6, J_7 at time 0, J_3 at time 6, J_1 at time 7, and J_4 at time 10.)

Solution: We start by augmenting the set of our jobs with two dummy jobs J_0 and J_{n+1} that have processing times equal zero. Furthermore, we make all the original jobs require J_0 to be

completed, and we define the set of required jobs of J_{n+1} to contain all the rest of the jobs. In this way, J_0 can be thought of as a "start" job and J_{n+1} as a "finish" job.

Next, we create a dependency graph that has one vertex per each job in our (augmented) set. For two jobs J_i, J_j, we put a directed edge from vertex J_i to vertex J_j if J_i is required by J_j. We set the weight of this edge (J_i, J_j) to be equal to the processing time of J_i. Note that the fact that there must exist at least one valid assembly schedule means that there is no circular precedence constraints and thus the dependency graph has to be a DAG.

It is easy to see that the minimum assembly time is just the *maximum* length of J_0-J_{n+1} path in this dependency graph.

This length can be computed by negating all the weights of the edges and finding the J_0-J_{n+1} distance $\delta(J_0, J_{n+1})$ in resulting graph – the minimum assembly time will be equal to $(-\delta(J_0, J_{n+1}))$. Since this graph is a DAG, we can compute this distance by running an appropriately modified version of Bellman-Ford that works in $O(m + n)$ time. (This version of Bellman-Ford was presented both in the lecture and in the recitations.) The total running time of this algorithm will be $O(m + n)$, as desired.

Quiz 2 Solutions

Problem 1. True or False [30 points] (10 parts)

For each of the following questions, circle either T (True) or F (False). There is no penalty for incorrect answers.

(a) T F [3 points] For *all* weighted graphs and all vertices s and t, Bellman-Ford starting at s will *always* return a shortest path to t.

Solution: FALSE. If the graph contains a negative-weight cycle, then no shortest path exists.

(b) T F [3 points] If all edges in a graph have distinct weights, then the shortest path between two vertices is unique.

Solution: FALSE. Even if no two edges have the same weight, there could be two *paths* with the same weight. For example, there could be two paths from s to t with lengths $3 + 5 = 8$ and $2 + 6 = 8$. These paths have the same length (8) even though the edges $(2, 3, 5, 6)$ are all distinct.

(c) T F [3 points] For a directed graph, the absence of back edges with respect to a BFS tree implies that the graph is acyclic.

Solution: FALSE. It is true that the absence of back edges with respect to a *DFS* tree implies that the graph is acyclic. However, the same is not true for a BFS tree. There may be cross edges which go from one branch of the BFS tree to a lower level of another branch of the BFS tree. It is possible to construct a cycle using such cross edges (which decrease the level) and using forward edges (which increase the level).

(d) T F [3 points] At the termination of the Bellman-Ford algorithm, even if the graph has a negative length cycle, a correct shortest path is found for a vertex for which shortest path is well-defined.

Solution: TRUE. If the shortest path is well defined, then it cannot include a cycle. Thus, the shortest path contains at most $V - 1$ edges. Running the usual $V - 1$ iterations of Bellman-Form will therefore find that path.

(e) T F [3 points] The depth of any DFS tree rooted at a vertex is at least as much as the depth of any BFS tree rooted at the same vertex.

Solution: TRUE. Since BFS finds paths using the fewest number of edges, the BFS depth of any vertex is at least as small as the DFS depth of the same vertex. Thus, the DFS tree has a greater or equal depth.

(f) T F [3 points] In bidirectional Dijkstra, the first vertex to appear in both the forward and backward runs must be on the shortest path between the source and the destination.

Solution: FALSE. When a vertex appears in both the forward and backward runs, it may be that there is another vertex (on a different path) which is further away from the source but substantially closer to the destination. (This was covered in recitation.)

(g) T F [3 points] There is no edge in an undirected graph that jumps more than one level of any BFS tree of the graph.

Solution: TRUE. If such an edge existed, it would provide a shorter path to some node than the path found by BFS (in terms in the number of edges). This cannot happen, as BFS always finds the path with the fewest edges.

(h) **T F** [3 points] In an unweighted graph where the distance between any two vertices is at most T, any BFS tree has depth at most T, but a DFS tree might have larger depth.

Solution: TRUE. Since all vertices are connected by a path with at most T edges, and since BFS always finds the path with the fewest edges, the BFS tree will have depth at most T. A DFS tree may have depth up to $V - 1$ (for example, in a complete graph).

(i) **T F** [3 points] BFS takes $O(V + E)$ time irrespective of whether the graph is presented with an adjacency list or with an adjacency matrix.

Solution: FALSE. With an adjacency matrix representation, visiting each vertex takes $O(V)$ time, as we must check all N possible outgoing edges in the adjacency matrix. Thus, BFS will take $O(V^2)$ time using an adjacency matrix.

(j) **T F** [3 points] An undirected graph is said to be *Hamiltonian* if it has a cycle containing all the vertices. Any DFS tree on a Hamiltonian graph must have depth $V - 1$.

Solution: FALSE. If a graph has a Hamiltonian cycle, then it is *possible*, depending on the ordering of the graph, that DFS will find that cycle and that the DFS tree will have depth $V - 1$. However, DFS is not guaranteed to find that cycle. (Indeed, finding a Hamiltonian cycle in a graph is NP-complete.) If DFS does not find that cycle, then the depth of the DFS tree will be less than $V - 1$.

Problem 2. Neighborhood Finding in Low-Degree Graphs [20 points]

Suppose you are given an adjacency-list representation of an N-vertex graph undirected G with non-negative edge weights in which every vertex has at most 5 incident edges. Give an algorithm that will find the K closest vertices to some vertex v in $O(K \log K)$ time.

Solution: We use a modified version of Dijkstra's algorithm for shortest paths. Suppose that we were to run Dijkstra's algorithm from v until we visited the $(K + 1)$-st vertex (i.e. v plus K more). Then, these K vertices (not including v) would be the vertices we want.

However, we must make a modification. In the version of Dijkstra presented in class, we create a binary heap (or Fibonacci heap) and initialize the distance of all N vertices to ∞. We can't do that here, as that would require $O(N)$ time to initialize and as subsequent heap operations would take $O(\log N)$ time instead of $O(\log K)$ time. Thus, we start with an empty heap. Then, when we relax an edge, we insert the destination of the edge into the heap if it isn't already there.

The total time for this algorithm can be determined by the number of operations we perform. As each vertex has degree at most 5 and as we visit K vertices, we perform at most $5K$ `Inserts`, $5K$ `DecreaseKeys`, and K `ExtractMins`. Since we perform at most $5K$ `Inserts`, the size of the heap is at most $5K$ and all heap operations take $O(\log 5K) = O(\log K)$ time. Thus, our modified Dijkstra takes $O(5K \log K + 5K \log K + K \log K) = O(K \log K)$. (Using a Fibonacci heap results in the same asymptotic runtime.)

[Note: Many students lost points on this problem for not explaining how the heap needs to start empty and how it never grows beyond $5K$ elements.]

Problem 3. Word Chain [15 points] (3 parts)

A word chain is a simple word game, the object of which is to change one word into another through the smallest possible number of steps. At each step a player may perform one of four specific actions upon the word in play — either *add a letter*, *remove a letter* or *change a letter* without switching the order of the letters in play, or *create an anagram* of the current word (an anagram is a word with exactly the same number of each letter). The trick is that each new step must create a valid, English-language word. A quick exaple would be FROG → FOG → FLOG → GOLF.

(a) [5 points] Give an $O(L)$-time algorithm for deciding if two English words of length L are anagrams.

Solution: Iterate through each of the L letters in each word, tracking the frequency of each letter. If both words have the same counts, the words are anagrams. This is similar to counting sort and runs in $O(L+S)$ time where S is the size of the alphabet.

(b) [2 points] Give an $O(L)$-time algorithm for deciding whether two words differ by one letter (added/removed/changed).

Solution: Iterate through each of the words comparing each letter as you go. If the letters do not match and one word is longer, then move to the next letter in that word. If a mismatch is found twice, return false, otherwise return true at the end.

Where the previous part asked about identifying anagrams, this asks about the one-off changes other than anagram listed above. Solutions which looked for a one-off anagram were not accepted.

(c) [8 points] Suppose you are given a dictionary containing N English words of length at most L and two particular words. Give an $O(N^2 \cdot L)$-time algorithm for deciding whether there is a word chain connecting the two words.

Solution: Construct a graph with each word in the dictionary being a node. For each node, create an edge to another node if the function from either a or b return true. Then use BFS on this graph to determine if one word can be reached from the other. Building the graph takes L time for each comparison, done comparing each node to each other node, N^2 time. This takes longer than BFS, so total time is $O(N^2 \cdot L)$.

Problem 4. Approximate Diameter [15 points]

The *diameter* of a weighted undirected graph $G = (V, E)$ is the maximum distance between any two vertices in G, i.e. $\Delta(G) = \max_{u,v \in V} \delta(u, v)$ where $\Delta(G)$ is the diameter of G and $\delta(u, v)$ is the weight of a shortest path between vertices u and v in G. Assuming that all edge weights in G are non-negative, give an $O(E + V \log V)$-time algorithm to find a value D that satisfies the following relation: $\Delta(G)/2 \le D \le \Delta(G)$. You must prove that the value of D output by your algorithm indeed satisfies the above relation.

Hint: For any arbitrary vertex u, what can you say about $\max_{v \in V} \delta(u, v)$?

Solution: We run Dijkstra's algorithm for single source shortest paths (using a Fibonacci heap) with an arbitrarily selected vertex u as the source. Since the vertices are removed from the heap in non-decreasing order of distance from u, the distance from u to the last vertex in the heap is $\max_{v \in V} \delta(u, v)$. Thus, we can find $\max_{v \in V} \delta(u, v)$ in $O(m + n \log n)$ time. We output $\max_{v \in V} \delta(u, v)$ as D. Since $\Delta \ge \delta(u, v)$ for all $u, v \in V$, $D \le \Delta$. Further,

$$\begin{aligned}
D &= \max_{v \in V} \delta(u, v) \\
&\ge \max_{v_1, v_2 \in V} \frac{\delta(u, v_1) + \delta(u, v_2)}{2} \\
&= \max_{v_1, v_2 \in V} \frac{\delta(v_1, u) + \delta(u, v_2)}{2} \quad \text{(since the graph is undirected)} \\
&\ge \max_{v_1, v_2 \in V} \frac{\delta(v_1, v_2)}{2} \quad \text{(by triangle inequality of } \delta) \\
&= \frac{\Delta}{2}.
\end{aligned}$$

Problem 5. Triple Testing [20 points]

Consider the following problem: given sets A,B,C, each comprising N integers in the range $-N^k \ldots N^k$ for some constant $k > 1$, determine whether there is a triple $a \in A$, $b \in B$, $c \in C$ such that $a+b+c=0$. Give a *deterministic* (e.g. no hashing) algorithm for this problem that runs in time $O(N^2)$.

Solution: Perhaps the simplest solution involved Radix-Sort. We start by generating a set D of all pairs $a+b, a \in A, b \in B$; this takes $O(N^2)$-time. Then we sort D in $O(N^2)$ time using Radix Sort; this is possible since after adding $2N^k$ to each element in D, all elements in D become integers in the range $0 \ldots 4N^k$, where k is constant. Let $D'[1 \ldots N^2]$ be the sorted array. Now, for each $c \in C$, we check whether $-c \in D$; this can be done in $O(\log N)$ time per element c using binary search on D'. Since $|C| = N$, we can perform all checks in $O(N \log N)$ time. Overall, the running time is $O(N^2)$.

There are many variants of the above solution. Here are common examples:

- Instead of searching for $-c$, $c \in C$, in D', one can search for $-d$, $d \in D$, in a sorted version of C. This solution is correct, but takes $O(N^2 \log N)$ time, so it received only a partial credit.
- To find collisions between elements in D and the inverses of elements in C, one can (i) label each element to denote whether it comes from D or is the inverse of an element in C, (ii) perform Radix Sort on the union of D and the inverses of the elements in C and (iii) check if the sorted array contains any consecutive elements that are equal, and have different labels. This takes $O(N^2)$ time.
- One can use hashing to check whether an element $-c$, $c \in C$, belongs to D. Unfortunately, this involves using hash functions, which are either randomized (as in universal hashing) or heuristic (there are some sets on which the hash function has bad performance). As such, hashing was explicitly disallowed by the problem statement. Still, solutions involving hashing received some partial credit.

Overall, the best way to approach this was problem was to use Radix Sort (or some other form of sorting). Some people attempted to map the problem into some shortest paths problem, where the elements of A , B and C are used as edge weights. However, finding a, b, c such that $a+b+c=0$ would typically require finding a path of length 0, which is quite different from finding the *shortest* path.

Problem 6. Number of Shortest Paths [20 points]

You are at an airport in a foreign city and would like to choose a hotel that has the maximum number of shortest paths from the airport (so that you reduce the risk of getting lost). Suppose you are given a city map with unit distance between each pair of directly connected locations. Design an $O(V+E)$-time algorithm that finds the number of shortest paths between the airport (the source vertex s) and the hotel (the target vertex t).

Solution: First we view the map as an (unweighted) undirected graph with locations as vertices and two locations are connected if and only if there is a unit-distance connection between the two locations. Then we do a breadth-first search (BFS) starting from the airport. We augment the data structure so that each vertex has an additional field paths to count the number of shortest paths from the root to that vertex. Initially we set $\text{paths}(s) = 1$ for the root vertex s (that is, the airport) and $\text{paths}(v) = 0$ for all other vertices. After each vertex (say v) is explored during BFS, we check all the neighbors of v and set $\text{paths}(v)$ to be the sum of the paths of its neighbor nodes whose level is one less than the level of v. That is (recall that, for every $v \in V(G)$, $N(v)$ denotes the set of vertices adjacent to the vertex v),

$$\text{paths}(v) = \sum_{w \in N(v) \text{ and } \text{level}(w) = \text{level}(v) - 1} \text{paths}(w).$$

The correctness of the algorithm follows from the fact that all the shortest paths from the root node s to a node t at level k must have length k, and each of the shortest path is of the form $\langle s, v_1, \ldots, v_k = t \rangle$, where node v_i is some node at level i in the BFS tree and $1 \le i \le k$. The only modification to the original BFS is about the counter $\text{paths}(v)$ for every vertex v, it is clear that initialization takes $O(V)$ time and updating the counter at any vertex v takes $O(|N(v)|)$ time. Note that $\sum_{v \in V(G)} |N(v)| = 2E$ and an ordinary BFS takes time $O(V+E)$, therefore the total running time of our modified BFS is $O(V+E) + O(V) + O(E) = O(V+E)$.

Quiz 2

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- When the quiz begins, write your name on every page of this quiz booklet.
- You have 120 minutes to earn 120 points. Do not spend too much time on any one problem. Read them all through first, and attack them in the order that allows you to make the most progress.
- This quiz is closed book. You may use **two** $8\frac{1}{2}'' \times 11''$ or A4 crib sheet (both sides). No calculators or programmable devices are permitted. No cell phones or other communications devices are permitted.
- Write your solutions in the space provided. If you need more space, write on the back of the sheet containing the problem. Pages may be separated for grading.
- Do not waste time and paper rederiving facts that we have studied. It is sufficient to cite known results.
- When writing an algorithm, a **clear** description in English will suffice. Pseudo-code is not required.
- When asked for an algorithm, your algorithm should have the time complexity specified in the problem with a correct analysis. If you cannot find such an algorithm, you will generally receive partial credit for a slower algorithm **if you analyze your algorithm correctly**.
- Show your work, as partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Be neat.
- Good luck!

Problem	Parts	Points	Grade	Grader
1	10	30		
2	1	20		
3	3	15		
4	1	15		
5	1	20		
6	1	20		
Total		120		

Name: ______________________________

Friday	Zuzana	Debmalya	Ning	Matthew	Alina	Alex
Recitation:	**10 AM**	**11 AM**	**12 PM**	**1 PM**	**2 PM**	**3 PM**

Problem 1. True or False [30 points] (10 parts)

For each of the following questions, circle either T (True) or F (False). There is no penalty for incorrect answers.

(a) T F [3 points] For *all* weighted graphs and all vertices s and t, Bellman-Ford starting at s will *always* return a shortest path to t.

(b) T F [3 points] If all edges in a graph have distinct weights, then the shortest path between two vertices is unique.

(c) T F [3 points] For a directed graph, the absence of back edges with respect to a BFS tree implies that the graph is acyclic.

(d) T F [3 points] At the termination of the Bellman-Ford algorithm, even if the graph has a negative length cycle, a correct shortest path is found for a vertex for which shortest path is well-defined.

(e) T F [3 points] The depth of any DFS tree rooted at a vertex is at least as much as the depth of any BFS tree rooted at the same vertex.

(f) T F [3 points] In bidirectional Dijkstra, the first vertex to appear in both the forward and backward runs must be on the shortest path between the source and the destination.

(g) T F [3 points] There is no edge in an undirected graph that jumps more than one level of any BFS tree of the graph.

(h) T F [3 points] In an unweighted graph where the distance between any two vertices is at most T, any BFS tree has depth at most T, but a DFS tree might have larger depth.

(i) T F [3 points] BFS takes $O(V + E)$ time irrespective of whether the graph is presented with an adjacency list or with an adjacency matrix.

(j) T F [3 points] An undirected graph is said to be *Hamiltonian* if it has a cycle containing all the vertices. Any DFS tree on a Hamiltonian graph must have depth $V - 1$.

Problem 2. Neighborhood Finding in Low-Degree Graphs [20 points]

Suppose you are given an adjacency-list representation of an N-vertex graph undirected G with non-negative edge weights in which every vertex has at most 5 incident edges. Give an algorithm that will find the K closest vertices to some vertex v in $O(K \log K)$ time.

Problem 3. Word Chain [15 points] (3 parts)

A word chain is a simple word game, the object of which is to change one word into another through the smallest possible number of steps. At each step a player may perform one of four specific actions upon the word in play — either *add a letter*, *remove a letter* or *change a letter* without switching the order of the letters in play, or *create an anagram* of the current word (an anagram is a word with exactly the same number of each letter). The trick is that each new step must create a valid, English-language word. A quick exaple would be FROG $\rightarrow$ FOG $\rightarrow$ FLOG $\rightarrow$ GOLF.

(a) [5 points] Give an $O(L)$-time algorithm for deciding if two English words of length L are anagrams.

(b) [2 points] Give an $O(L)$-time algorithm for deciding whether two words differ by one letter (added/removed/changed).

(c) [8 points] Suppose you are given a dictionary containing N English words of length at most L and two particular words. Give an $O(N^2 \cdot L)$-time algorithm for deciding whether there is a word chain connecting the two words.

Problem 4. Approximate Diameter [15 points]

The *diameter* of a weighted undirected graph $G = (V, E)$ is the maximum distance between any two vertices in G, i.e. $\Delta(G) = \max_{u,v \in V} \delta(u, v)$ where $\Delta(G)$ is the diameter of G and $\delta(u, v)$ is the weight of a shortest path between vertices u and v in G. Assuming that all edge weights in G are non-negative, give an $O(E + V \log V)$-time algorithm to find a value D that satisfies the following relation: $\Delta(G)/2 \leq D \leq \Delta(G)$. You must prove that the value of D output by your algorithm indeed satisfies the above relation.
Hint: For any arbitrary vertex u, what can you say about $\max_{v \in V} \delta(u, v)$?

Problem 5. Triple Testing [20 points]

Consider the following problem: given sets A,B,C, each comprising N integers in the range $-N^k \dots N^k$ for some constant $k > 1$, determine whether there is a triple $a \in A$, $b \in B$, $c \in C$ such that $a+b+c=0$. Give a *deterministic* (e.g. no hashing) algorithm for this problem that runs in time $O(N^2)$.

Problem 6. Number of Shortest Paths [20 points]

You are at an airport in a foreign city and would like to choose a hotel that has the maximum number of shortest paths from the airport (so that you reduce the risk of getting lost). Suppose you are given a city map with unit distance between each pair of directly connected locations. Design an $O(V+E)$-time algorithm that finds the number of shortest paths between the airport (the source vertex s) and the hotel (the target vertex t).

SCRATCH PAPER

SCRATCH PAPER

6.006 Exam 2 Notes

4/25

DP

subseq = longest up to here

So there were 2 ways dp[i] was 2nd

max was 1st

(2,7) (0,0) (7,0) (0,3) (0,6) (7,6)

x ✓ x ✓ ✓ ↑

↜ ; but how to see recursively call

need like a keep or skip though

If football w/ prev ↑ 1

Otherwise keep looking

(is actually fairly diff

(should practice more...)

Also confusing since 2

Try to think the other way

dp[i] ~~fast~~ highest football ending there

└ is football, add 1

Or is not skip

Compare n times

for n times - wait that works!

② Improve

1. Memoize prev results – how
   └ Should be in previous

Don't have to get to n

When compute longest score → remove
Starting from back - will be right
But if multiple use it

00   03   33
      30

Even though not shorter

anything special w/ 2, 3, 7   2·3=6
                               ↑ so either way

00 — 20 — 40 — 60
   \ 30 /

remember linear
   - so only 1 seq
this is greedy
any tricks w/ that?

(0,0) (2,0) (4,0) (3,0) (6,0)

2

not 3

So previous is wrong - don't want greedy

(3)

So 2 2 2 = 33

~~3232~~

2 2 3 = 7

So branch on 7

and 3

- since 2 would be longest

This is n^3 - though

no 2n - scanning list

still

Back to part b)

Do we have a cheat/short cut?

So the 2 sol were

[< and less] + 1 or [all except last]

and then it was the dp[i]

BST of everything less than

Which way did it build?

Start →

calc football of 1st = 1

Calc football of 2nd

if 1 → 2 = 2

if 1 ↛ 2 = 1

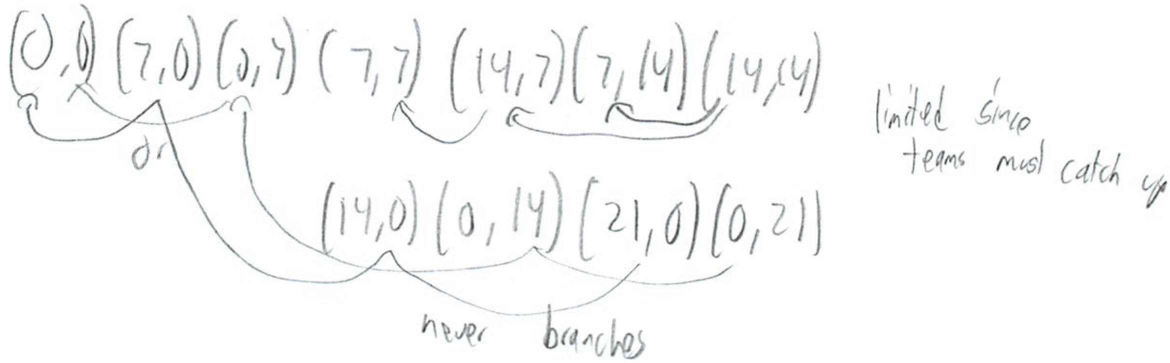
(4)

Then use previous results

Same branching problem

L oh ugh forget about out of order

but this can branch every $\frac{n}{2}$



So basically weight = time

flight = time

but also waiting = time

So must explore from every node

(or dij

but could be surprises - since time is in nodes too

Need full BFS

So

earliest flight

then next [illegible] flights

no repeats

So BFS $O(V+E)$

Can we beat linear scan for time?

~~binary~~ BST

but have to build ... just for 1 lookup

Introduction to Algorithms
Massachusetts Institute of Technology
Profs. Constantinos Daskalakis and Silvio Micali

April 25, 2012
6.006
Quiz 2

Quiz 2

You will have 2 hours to complete this exam. No notes or other resources are allowed. Unless otherwise specified, full credit will only be given to a correct answer which is described clearly and concisely.

Do not discuss this exam with anyone who has not yet taken it.

Problem	Points	Grade	Initials
Name	1	1	HP
1	24	21	$
2	18	15	$
3	12	10	$
4	25	11	SK
5	20	14	HP
Total	100	72	RM

Name: [1 point] Michael Plasmeier

R01	R02	R03	R04	R05	R06	R07
WF10	WF11	WF12	WF1	WF2	WF3	WF3
Shaunak	Shaunak	Alan	Jeff	Rafael	Henrique	Dragos

Problem 1. True/False [24 points]

Note: Correct answers are worth 2 points, blanks are worth 0 points, and incorrect answers are worth -3 points. You will not be graded on any explanation.

(a) Depth-first search can be modified to check if there are cycles in an undirected graph.

Circle: **True** or **False**

(b) Breadth-first search can be modified to check if there are cycles in an undirected graph.

Circle: **True** or **False**

(c) If we represent a graph with $|V|$ vertices and $\Theta(|V|)$ edges as an adjacency matrix, the worst-case running time of breadth-first search is $\Theta(|V|^2)$.

Circle: **True** or **False**

(d) In this problem, suppose that G is a directed graph and that u and v are vertices of this graph such that there is a path from u to v in G but no path from v to u.

i. Any depth-first search in G that discovers both u and v must discover u before it discovers v. [1]

Circle: **True** or **False**

ii. Any depth-first search in G that discovers both u and v must finish u before it finishes v. [2]

Circle: **True** or **False**

iii. Any depth-first search in G that discovers u and ***later*** discovers v must finish u before it finishes v.

Circle: **True** or **False**

[1] In the terminology of CLRS, the discovery time is the time it is colored grey.

[2] The finishing time is, in the terminology of CLRS, the time in which a vertex is colored black.

(e) The strongly-connected components of a directed graph are preserved if you reverse every edge — that is, if you replace every edge (u, v) of the graph with the edge (v, u).[3]

Circle: **True** or **False**

(f) We can use Dijkstra's algorithm to find the shortest path between two vertices in a graph with arbitrary edge weights.

Circle: **True** or **False**

(g) The worst-case running time of A* is asymptotically better than the worst-case running time of Dijkstra's algorithm.

Circle: **True** or **False**

(h) Suppose that s and t are vertices in a weighted graph G that does not contain negative cycles, and suppose that there is a path from s to t. We run Bellman-Ford on G with starting vertex s.

i. If there is a shortest path from s to t consisting of k edges, then after the k^{th} iteration, then Bellman-Ford's estimate of the distance to t will be correct.

Circle: **True** or **False**

ii. If Bellman-Ford's estimate of the distance to t is correct after the k^{th} iteration, then there is a shortest path from s to t consisting of at most k edges.

Circle: **True** or **False**

(i) If we draw out the full recursion tree of a problem that can be sped up by memoization, the same subproblem might appear multiple times in the tree.

Circle: **True** or **False**

[3] Recall that u and v are in the same strongly connected component if there is a path from u to v and a path from v to u.

Prob i

No negative credit, guess anyway

Problem 2. Short Answers [18 points]

(a) [6 points] Mark the entries of the following table that correspond to properties that are true of counting and radix sort, as described in lecture.

5

Property	Counting sort	Radix sort
Can be implemented so it is stable	✓	✓
Can be implemented so it is in-place	X	X
Sorts n integers in the range $\{0, 1, \ldots, n^c\}$ in $O(n)$ time, for any constant $c > 0$.	X	✓

linear - but $O(\#)$ n+d

(b) [8 points] On which of the following undirected graphs does bi-directional breadth-first search perform asymptotically better than regular breadth-first search? Circle the numbers of **all that apply**.

Kishore said in recitation

i. A path graph on n vertices, in which s and t are connected by a path of length $n-1$ (and there are no other edges). n vs n/2

ii. A complete graph, in which there is an edge between every pair of vertices. less exponential growth

iii. A star graph, in which s, t, and $n-3$ other vertices are all connected to a central nth vertex (and there are no other edges). n v 2

iv. A balanced binary tree on n vertices in which s and t are leaves.

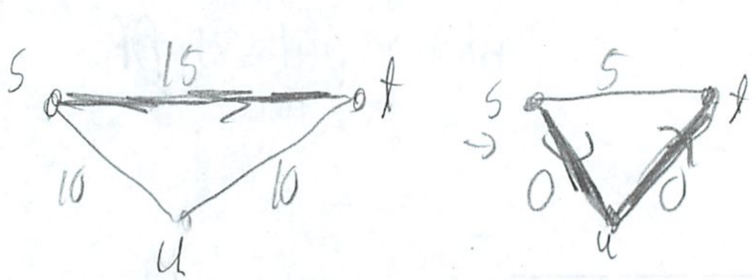
6

(c) [4 points] Ben Bitdiddle thinks it is possible to find s-t shortest paths on any weighted graph using the following algorithm:

1. Find the minimum weight, m, of any edge.
2. Subtract m from the weight of every edge - that is, let $w'(i,j)$ equal $w(i,j) - m$.
3. Run Dijkstra on the transformed graph. No!

Draw a three-vertex directed graph with vertices s, t, and u on which Ben's algorithm does **not** find the shortest path from s to t. Label the vertices and assign **non-negative** weights to the edges to construct your counterexample.

4

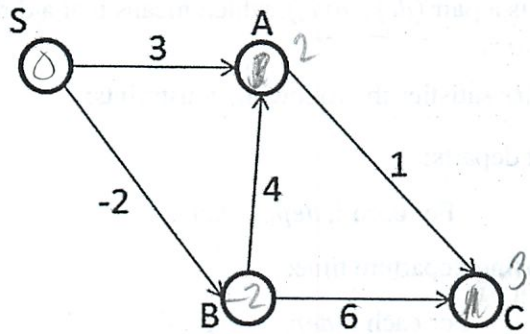


Goes a diff path.

Not the original shortest path

Problem 3. Bellman-Ford [12 points]

In this problem, you must run Bellman-Ford manually on the directed graph provided below, starting at the source vertex S. In each iteration, the edges will be relaxed in the following order: BC, AC, BA, SA, and SB.



(a) [8 points] Fill in the table with the distance estimates for each vertex after each iteration. Note that **all** the edges are relaxed in each iteration. For example, after the first iteration, you should find that the distance estimate for B is -2. Yes duh

Vertex	Iteration 0	Iteration 1	Iteration 2	Iteration 3
S	0	0	0	0
A	∞	3	2	2
B	∞	-2	-2	-2
C	∞	∞	4	3

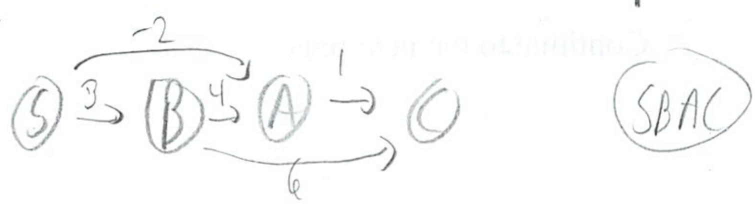
8

(b) [4 points] In the worst case, the Bellman-Ford algorithm runs for $|V| - 1$ iterations, where $|V|$ is the number of vertices. However, for this particular graph, there exists an ordering of the edges such that for **any** edge weights, the Bellman-Ford algorithm will terminate after a **single** iteration.

Give one such edge ordering, and briefly explain why it works.

Since it is a DAG, has special rules 2

Topo sort it to get it in proper form



Only run through once since once you leave a node you can't come back to it later. Same ans as before.

Problem 4. Optimal Travel Plans [25 points]

In this problem, your goal is to determine a sequence of flights between airports which will get you from your current location to a target destination as quickly as possible.

Each airport is represented by a vertex on a directed graph $G = (V, E)$. Each directed edge $e = (u, v)$ in the graph has an associated array, $e.flights$. The array $e.flights$ has at most k entries. The i^{th} entry in the array is a pair (dep_i, arr_i), which means that a direct flight leaves u at time dep_i and arrives at v at time arr_i.

For each edge e, the array $e.flights$ satisfies the following constraints:

1. No flight can arrive before it departs:

$$\text{For each } i,\ dep_i < arr_i.$$

2. The array is sorted by increasing departure time:

$$\text{For each } i,\ dep_i < dep_{i+1}.$$

3. Flights that depart later also arrive later: not like an Amazing Race.

$$\text{For each } i,\ arr_i < arr_{i+1}.$$

Given a source node s and an initial starting time, your goal is to determine a sequence of flights that can be taken to reach a target node t as early as possible. Of course, in order for you to take a flight, you must be at airport that the flight departs from by the time that it departs.

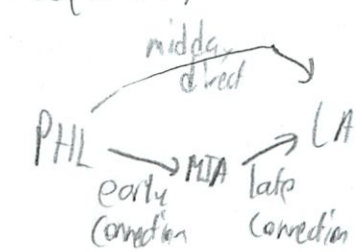
Continue to the next page.

(a) [20 points] Design and analyze an efficient algorithm to compute a sequence of flights which arrives at t as early as possible. Be sure to explicitly state your algorithm's running time in terms of $|V|$, $|E|$, and k. known, fixed 5

8

Note: You will receive 6 points for a blank answer to this question. You will get more than 6 points for progress towards a correct solution, but any other text will count against you.

We esentially have a cost in a node (terminal) now.
We can't use Dijstra since its too greedy



So must explore every node - so BFS,
Say we start at my home PHL, Explore the first flight out of PHL to everywhere. From those places take their next flights elsewhere that depart after you arrive already! Do not fly somewhere we have already explored (since we could have gotten there earlier). Stop when you arrive at t.

Must modify Dijkstra to solve this well correctly. BFS will not respect edge costs.

This is standard BFS $O(V+E)$ except we linearly scan a table (k) for each $V \in$ | We know $E \leq k$ $V \leq k$

So $O(kV+E)$. Visit each node at most once and only traverse an edge at most once as BFS.

(b) [5 points] Suppose now that we now have additional geographic information about the graph. In particular, we model the Earth as a plane, and we determine the location (x_u, y_u) of each airport in the plane. We also know that the speed of any plane is bounded by a top speed, s.

3

Explain how to use a heuristic to speed up your search algorithm in practice. Be sure to explicitly define any heuristic functions that you use.

Note: Use at most four sentences. You should only need half of this page.

We Use the planar distance to calculate the "as the crow flys" distance between (X_a, Y_a) and (X_b, Y_b) [which is close but not exact to how an airplane flys)

We then divide this distance by the planes top speed (which it normally flys at) to find the flight lenght

$$\lambda = \text{flight lenght} = \frac{\text{crows distance}}{\text{speed}} = \frac{\sqrt{(X_a - X_b)^2 + (Y_a - Y_b)^2}}{\text{speed}}$$ ✓

We use this to get a lower band on how much time to fly to t from the next airport we are considering So we compare actual next flight + huristic and pick lowest.

for i j (at i ? considering j's) ; $i \to j$; $j \to t$

Not true:

In practice this has us go towards t, not in all directions

Not s t but s t

Problem 5. Longest Football Subsequence [20 points]

In the game of football, teams can score 2, 3, or 7 points at a time. A *football sequence* is a sequence of valid scores for the two teams in a football game. That is, it is a sequence of pairs of nonnegative integers (a_i, b_i) that satisfies the following properties:

1. The initial scores are 0:

$$(a_0, b_0) = (0, 0).$$

2. Exactly one team scores at a time:

$$\text{For all } i, \text{ either } a_{i+1} = a_i \text{ or } b_{i+1} = b_i, \text{ but not both.}$$

3. Teams score in the correct increments:

$$\text{If } a_{i+1} \neq a_i, \text{ then } a_{i+1} - a_i \text{ is either 2, 3 or 7, and}$$

$$\text{If } b_{i+1} \neq b_i, \text{ then } b_{i+1} - b_i \text{ is either 2, 3 or 7}$$

For example, the following sequence is a football sequence:

$$(0, 0), (0, 3), (0, 5), (7, 5), (7, 12), (9, 12).$$

In this problem, your goal is to determine the length of the longest football subsequence of a given n-element sequence S of pairs of nonnegative integers (x_i, y_i). (Note that your subsequence may include non-consecutive elements of S, as long as their relative order is preserved.)

For example, if

$$S = (2, 7), (0, 0), (7, 0), (0, 3), (0, 6), (7, 6)$$

then the longest football subsequence is

$$(0, 0), (0, 3), (0, 6), (7, 6),$$

so your algorithm should return 4.

Continue to the next page.

(a) [10 points] Give a simple dynamic programming algorithm which finds the size of the largest football subsequence in time $O(n^2)$. Briefly prove your algorithm's runtime and argue its correctness.

Note: You will receive 3 points for a blank answer to this question. You will get more than 3 points for progress towards a correct solution, but any other text will count against you.

7/10

Work from the back forwards

For each integer i, calculate dp[i]

dp[i] = Start at i and count back j times ($j := n-i$)

If jump is football, add 1

If jump is not football, add 0

should find max because there could be many j's

You should only return if you find a (0,0)

keep special track of largest dp[i]

So each for each item on the list (n), scan the whole list (n) = $O(n^2)$

Example (2,7), (0,0), (7,0), (0,3), (0,6), (7,6)

3 2 1 — draw arrow where add 1

2 1 — is actually stupid, we already know

1

1

max = 3

We're not memoizing → so not DP – but can memoize to save more time

is greedy

See pg 12

(b) [10 points] Design and analyze the most efficient algorithm you can for this problem. Be sure to explicitly state your algorithm's running time in terms of n, and briefly argue its correctness. Nieve sol (10%) = Same as previous :) O(n²)

Note: You will receive 3 points for a blank answer to this question. You will get more than 3 points for progress towards a correct solution, but any other text will count against you.

7/10

Move left to right →
Calc dp[i] of each and memoize
1st one → dp[1] = 1 → dp = 0 unless it is (0,0) in which case it is 1.
Then for next → if football w/ previous =
∟ it is dp[previous] + 1 = dp[i] ← from memoized
∟ otherwise dp[i] = 1
} Should take max
O(n) since scan through list once
But same branching problem as before – just forwards here

222 = 33
223 = 7

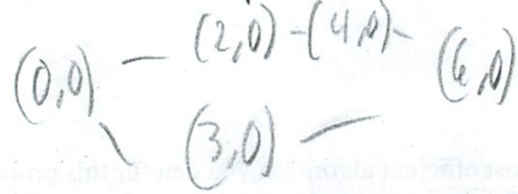
Plus other team

So might have to scan 4n

Still O(n)

(100 Simple

But there is a problem



Arranged like (0,0)(2,0)(4,0)(3,0)(6,0) The greedy algorithm would be wrong

The only 2 cases: 2 2 2 = 33
223 = 7

So when you come across either 7 or two 3s in a row, branch (included it and don't include it) for the next 3 football jumps (where you add 1) which are not +7 but are a permutation of 222 (for 2×3s) or 223 (for 7)

This could be 2n worst case → still n^2

Correct, since we are exploring every possible football seq by checking the entire list and branching where needed

Also branching from teams scoring out of order (going backward)
This is limited (ie not polynomical) since teams must be caught up in order to branch. They must catch up again in order to re-branch - so you just merge them when the catch up
Another 2n only
So total 2·2n = 4n possible branches
(0,0)(7,0)(0,7)(7,7)(14,7)(7,14)(14,14)

email Peter

Quiz 2

You will have 2 hours to complete this exam. No notes or other resources are allowed. Unless otherwise specified, full credit will only be given to a correct answer which is described clearly and concisely.

Do not discuss this exam with anyone who has not yet taken it.

Problem	Points	Grade	Initials
Name	1		
1	24		
2	18		
3	12		
4	25		
5	20		
Total	100		

Name: [1 point] ______________________________

R01	R02	R03	R04	R05	R06	R07
WF10	WF11	WF12	WF1	WF2	WF3	WF3
Shaunak	Shaunak	Alan	Jeff	Rafael	Henrique	Dragos

Problem 1. True/False [24 points]

Note: Correct answers are worth 2 points, blanks are worth 0 points, and incorrect answers are worth -3 points. You will not be graded on any explanation.

(a) Depth-first search can be modified to check if there are cycles in an undirected graph.

True

(b) Breadth-first search can be modified to check if there are cycles in an undirected graph.

True

(c) If we represent a graph with $|V|$ vertices and $\Theta(|V|)$ edges as an adjacency matrix, the worst-case running time of breadth-first search is $\Theta(|V|^2)$.

True

(d) In this problem, suppose that G is a directed graph and that u and v are vertices of this graph such that there is a path from u to v in G but no path from v to u.

i. Any depth-first search in G that discovers both u and v must discover u before it discovers v. [1]

False

ii. Any depth-first search in G that discovers both u and v must finish u before it finishes v. [2]

False

iii. Any depth-first search in G that discovers u and ***later*** discovers v must finish u before it finishes v.

False

[1] In the terminology of CLRS, the discovery time is the time it is colored grey.

[2] The finishing time is, in the terminology of CLRS, the time in which a vertex is colored black.

(e) The strongly-connected components of a directed graph are preserved if you reverse every edge — that is, if you replace every edge (u, v) of the graph with the edge (v, u).[3]

True

(f) We can use Dijkstra's algorithm to find the shortest path between two vertices in a graph with arbitrary edge weights.

False

(g) The worst-case running time of A* is asymptotically better than the worst-case running time of Dijkstra's algorithm.

False

(h) Suppose that s and t are vertices in a weighted graph G that does not contain negative cycles, and suppose that there is a path from s to t. We run Bellman-Ford on G with starting vertex s.

i. If there is a shortest path from s to t consisting of k edges, then after the k^{th} iteration, then Bellman-Ford's estimate of the distance to t will be correct.

True

ii. If Bellman-Ford's estimate of the distance to t is correct after the k^{th} iteration, then there is a shortest path from s to t consisting of at most k edges.

False

(i) If we draw out the full recursion tree of a problem that can be sped up by memoization, the same subproblem might appear multiple times in the tree.

True

[3]Recall that u and v are in the same strongly connected component if there is a path from u to v and a path from v to u.

Problem 2. Short Answers [18 points]

(a) [6 points] Mark the entries of the following table that correspond to properties that are true of counting and radix sort, as described in lecture.

Solution:

Property	Counting sort	Radix sort
Can be implemented so it is stable	X	X
Can be implemented so it is in-place		
Sorts n integers in the range $\{0, 1, \ldots, n^c\}$ in $O(n)$ time, for any constant $c > 0$.		X

Note: We gave everyone full credit for the in-place question. This is because, while what we did in classl

(b) [8 points] On which of the following undirected graphs does bi-directional breadth-first search perform asymptotically better than regular breadth-first search? Circle the numbers of **all that apply**.

i. A path graph on n vertices, in which s and t are connected by a path of length $n - 1$ (and there are no other edges).

ii. A complete graph, in which there is an edge between every pair of vertices.

iii. A star graph, in which s, t, and $n - 3$ other vertices are all connected to a central nth vertex (and there are no other edges).

iv. A balanced binary tree on n vertices in which s and t are leaves.

(c) [4 points] Ben Bitdiddle thinks it is possible to find s-t shortest paths on any weighted graph using the following algorithm:

1. Find the minimum weight, m, of any edge.
2. Subtract m from the weight of every edge - that is, let $w'(i, j)$ equal $w(i, j) - m$.
3. Run Dijkstra on the transformed graph.

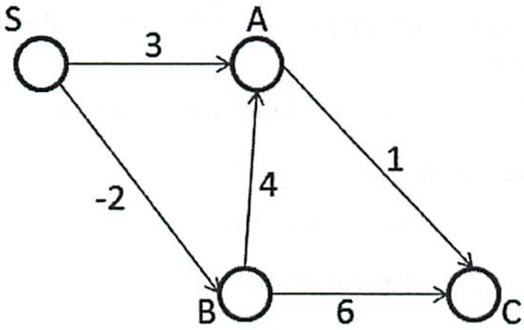
Draw a three-vertex directed graph with vertices s, t, and u on which Ben's algorithm does **not** find the shortest path from s to t. Label the vertices and assign **non-negative** weights to the edges to construct your counterexample.

Solution:

Any graph where the shortest $s \rightarrow t$ path goes directly from s to t,, yet goes through u after transforming. For example, have $w(s, u) = w(u, t) = 2$ and $w(s, t) = 3$.

Problem 3. Bellman-Ford [12 points]

In this problem, you must run Bellman-Ford manually on the directed graph provided below, starting at the source vertex S. In each iteration, the edges will be relaxed in the following order: BC, AC, BA, SA, and SB.



(a) [8 points] Fill in the table with the distance estimates for each vertex after each iteration. Note that **all** the edges are relaxed in each iteration. For example, after the first iteration, you should find that the distance estimate for B is -2.

Solution:

Vertex	Iteration 0	Iteration 1	Iteration 2	Iteration 3
S	0	0	0	0
A	∞	3	2	2
B	∞	-2	-2	-2
C	∞	∞	4	3

(b) [4 points] In the worst case, the Bellman-Ford algorithm runs for $|V| - 1$ iterations, where $|V|$ is the number of vertices. However, for this particular graph, there exists an ordering of the edges such that for **any** edge weights, the Bellman-Ford algorithm will terminate after a **single** iteration.

Give one such edge ordering, and briefly explain why it works.

Solution: Since the graph is a DAG, we can take any ordering which will process all of the edges of a path in the correct order. In particular, a sequence works if and only if for every node, the incoming edges are relaxed before any of the outgoing edges. There are actually many such sequences.

Problem 4. Optimal Travel Plans [25 points]

In this problem, your goal is to determine a sequence of flights between airports which will get you from your current location to a target destination as quickly as possible.

Each airport is represented by a vertex on a directed graph $G = (V, E)$. Each directed edge $e = (u, v)$ in the graph has an associated array, $e.flights$. The array $e.flights$ has at most k entries. The i^{th} entry in the array is a pair (dep_i, arr_i), which means that a direct flight leaves u at time dep_i and arrives at v at time arr_i.

For each edge e, the array $e.flights$ satisfies the following constraints:

1. No flight can arrive before it departs:

   For each i, $dep_i < arr_i$.

2. The array is sorted by increasing departure time:

   For each i, $dep_i < dep_{i+1}$.

3. Flights that depart later also arrive later:

   For each i, $arr_i < arr_{i+1}$.

Given a source node s and an initial starting time, your goal is to determine a sequence of flights that can be taken to reach a target node t as early as possible. Of course, in order for you to take a flight, you must be at airport that the flight departs from by the time that it departs.

Continue to the next page.

(a) [20 points] Design and analyze an efficient algorithm to compute a sequence of flights which arrives at t as early as possible. Be sure to explicitly state your algorithm's running time in terms of $|V|$, $|E|$, and k.

Note: You will receive 6 points for a blank answer to this question. You will get more than 6 points for progress towards a correct solution, but any other text will count against you.

Solution:

We solve this problem by running a slightly modified variant of Dijkstra's algorithm (using a fibonacci heap, for optimal running time.) In our algorithm, for each node v we will keep track of $d[v]$ which is earliest time for which we know it is possible to arrive at v, when we leave from s at $t = 0$. (We initialize $d[v] = \infty$ for all $v \neq s$ and $d[s] = 0$.) We now run Dijkstra's algorithm using these d values. The only difference is in how we relax an edge.

Consider relaxing edge $e = (u, v)$. Because of the properties of e.flights, we know that leaving u for v as early as possible will be at least as good as having a longer layover in u and leaving for the direct $u \to v$ flight later. Therefore, we look for the smallest i^* such that the corresponding depart_{i^*} in e.flights is greater than or equal to $d[u]$. We can use binary search to find this i in time $O(\log k)$.

To relax the edge $e = (u, v)$, we set $d[v]$ to be the minimum of the current $d[v]$ and $d[v] + \text{arrive}_{i^*}$.

The correctness of this algorithm follows from the correctness of Dijkstra's algorithm. The running time, using a fibonacci heap for the Dijkstra priority queue, is $O(V \log V + E \log k)$. (The $O(\log k)$ term comes from needing to find the earlier flight to take on a given edge.)

Partial credit was given for the $O(V \log V + kE)$ solution which did a linear scan through the departure times. Also, note that using binary heaps instead of a fibonacci heap gives running time $O(\log kE \log V)$.

Note that this problem was a modified (harder) version of an idea from

http://www.csl.mtu.edu/cs2321/www/newLectures/30_More_Dijkstra.htm

Notes on grading:

- **Getting runtimes using binary heaps instead of Fibonacci heaps lost you no points.**
- **Not getting the binary search step caused you to lose 2 points.**
- **There were other solutions which involved modifying the graph, which got a worse running-time, and thus 16 or 12 points, depending on whether the transformation obtains Ek or VEk edges in the new graph.**

(b) [5 points] Suppose now that we now have additional geographic information about the graph. In particular, we model the Earth as a plane, and we determine the location (x_u, y_u) of each airport in the plane. We also know that the speed of any plane is bounded by a top speed, s.
Explain how to use a heuristic to speed up your search algorithm in practice. Be sure to explicitly define any heuristic functions that you use.

Note: Use at most four sentences. You should only need half of this page.

Solution: We can use a heuristic of $h(u) = \sqrt{(x_u - x_t)^2 + (y_u - y_t)^2}/c$, which is clearly a lower bound on the minimum time it will take to get from u to t. We modify our Dijkstra algorithm analogously to A^*: instead of sorting our queue by d values, we sort by $d + h$ values.

Problem 5. Longest Football Subsequence [20 points]

In the game of football, teams can score 2, 3, or 7 points at a time. A *football sequence* is a sequence of valid scores for the two teams in a football game. That is, it is a sequence of pairs of nonnegative integers (a_i, b_i) that satisfies the following properties:

1. The initial scores are 0:

$$(a_0, b_0) = (0, 0).$$

2. Exactly one team scores at a time:

For all i, either $a_{i+1} = a_i$ or $b_{i+1} = b_i$, but not both.

3. Teams score in the correct increments:

If $a_{i+1} \neq a_i$, then $a_{i+1} - a_i$ is either 2, 3 or 7, and

If $b_{i+1} \neq b_i$, then $b_{i+1} - b_i$ is either 2, 3 or 7

For example, the following sequence is a football sequence:

$$(0,0), (0,3), (0,5), (7,5), (7,12), (9,12).$$

In this problem, your goal is to determine the length of the longest football subsequence of a given n-element sequence S of pairs of nonnegative integers (x_i, y_i). (Note that your subsequence may include non-consecutive elements of S, as long as their relative order is preserved.)

For example, if

$$S = (2,7), (0,0), (7,0), (0,3), (0,6), (7,6)$$

then the longest football subsequence is

$$(0,0), (0,3), (0,6), (7,6),$$

so your algorithm should return 4.

Continue to the next page.

(a) [10 points] Give a simple dynamic programming algorithm which finds the size of the largest football subsequence in time $O(n^2)$. Briefly prove your algorithm's runtime and argue its correctness.

Note: You will receive 3 points for a blank answer to this question. You will get more than 3 points for progress towards a correct solution, but any other text will count against you.

Solution:

Let $C[i]$ be the length of the longest football subsequence which ends at $S[i]$. We now do a scan of S from left to right to compute the C values. To compute $C[i]$, we need only look at the values between $C[1], \ldots, C[i-1]$ and look at the maximum value $C[j]$ for which we could append $S[i]$ after $S[j]$. The correctness of this algorithm follows trivially by induction: Given that we have computed the first $i-1$ values of C correctly, it follows that any football subsequence ending with $S[i]$ is constructed by taking a football subsequence ending before i and (if valid) appending $S[i]$ to the end. (If $S[i] = (0,0)$, then we can instead have the length-1 subsequence consisting of $S[i]$. Since football subsequences are increasing, $(0,0)$ can only appear at the beginning of a subsequence.)

This gives us the following pseudocode algorithm:

- Initialize $C[i] = -\infty$ for all i
- For $i = 1$ to n:
  - Let c^* be the maximum $C[j]$ value for $j < i$ such that $S[i]$ can immediately follow $S[j]$ in a valid football sequence (that is, the first components are equal and the second component increases by 2, 3, or 7, or instead the second components are equal and the first component increases by 2, 3, or 7.) If no such j exists, set c^* to $-\infty$.
  - If $S[i] == (0,0)$, set $C[i] \leftarrow 1$.
  - Otherwise, set $C[i] \leftarrow c^* + 1$.
- Return $\max\{\max_i C[i], 0\}$.

Notice that in the final step, we return $\max\{\max_i C[i], 0\}$. This deals with the case that $(0,0)$ does not appear in S.

Our algorithm has running time $O(n + 1 + 2 + 3 + \cdots + (n-1)) = O(n^2)$.

(b) [10 points] Design and analyze the most efficient algorithm you can for this problem. Be sure to explicitly state your algorithm's running time in terms of n, and briefly argue its correctness.

Note: You will receive 3 points for a blank answer to this question. You will get more than 3 points for progress towards a correct solution, but any other text will count against you.

Solution:

We can solve this problem by doing a single linear scan through S. As before, we let $C[i]$ be the length of the longest football subsequence ending at $S[i]$. We note that, for $S[i]$ to appear in a sequence, there are at most 6 possible terms that could immediately proceed $S[i]$ (corresponding to subtracting 2, 3, or 7 from either the first or second component.)

As we scan S, we will hash each pair (a, b) for which we have found an j with $S[j] = (a, b)$ and $C[j] > 0$. The value of (a, b) will be the length of the longest football subsequence we have found thus far which ends at (a, b). We will compute $C[i]$ by searching for all six possible proceeding (a, b) pairs in the hash table, and setting $C[i]$ to be 1 more than the maximum of the corresponding C values for these six pairs. We give pseudocode for this algorithm below. Correctness follows from a trivial induction argument, similar to that above.

- Create a hash table d.
- For $i = 1$ to n:
  - Denote by (a, b) the term $S[i]$.
  - If $(a, b) == (0, 0)$, set $C[i] \leftarrow 1$ and $d[(0, 0)] \leftarrow 1$.
  - Else:
    * Let P be the set of the (at most 6) pairs which could possibly proceed (a, b) in a football sequence. (i.e., P consists of the terms $(a - 7, b), (a - 3, b), (a - 2, b), (a, b - 7), (a, b - 3), (a, b - 2)$ in which both values are nonnegative.)
    * If there exists a $p \in P$ for which $d.has\_key(p)$, let c^* be $\max_{p \in P} d[p]$ (if any p is not in d, initialize its value to $-\infty$). Set $C[i] \leftarrow c^*$ and $d[(a, b)] \leftarrow \max\{d[(a, b)], c^* + 1\}$.
    * Otherwise, set $c[i] \leftarrow -\infty$.
- Return $\max\{\max_i C[i], 0\}$.

By resizing our hash table appropriately, the amortized running time for our hash operations will be $O(1)$ per operation. Therefore, each iteration of the inner loop takes $O(1)$ time, and therefore the overall running time is $O(n)$. (Notice that there are several slight variations of this algorithm which also achieve $O(n)$ running time.)

25 6.006 ▾ 14 Search or add a post... + New Post Michael Pl... Join our team

Views: Filter: All »

▾ PINNED

Instr Note Quiz 2 stats

Instr Note Preliminary Quiz 2 solution... 6

Instr Note No Recitation Tomorrow 4

Instr Note Late/wrong pset submissions 1

▾ TODAY

Why is problem 1.h.ii on the quiz false? 4

pset 6 prob 2 5

▾ YESTERDAY

Quiz 2 Grade distribution 4

When the test cases for pset 6 up? 7

▾ THIS WEEK

bottom up vs. top down DP 3

Question about today's lecture 4
- 1 Unresolved Followup

quiz 2 solutions 4

Can 6.006 help the USPS?

spring 2009 question 2d 1

hamiltonian cycle 3

fall 2008 question 2a 2

Grades before Drop Day? 5

General question about Why use com... 4

Bi-directional Djikstra's algorithm 4

Instr Note UPDATE: Change to gradin... 5

Fall 2011 Quiz 1f 2

Runtime of auxiliary graph by changin... 2

2010 Fall question 6 5

Can a graph have a positive weight cy... 3

Quiz 2 Spring 2008 3a 5

memorizing runtimes 2

spring 2009 problem 5 (c) 10

Instr Note UPDATED!! Regarding the t...

Spring 2011 Problem 6 2

Bidirectional Dijkstra 2

pset 6 clarification 2

A* heuristics 5

Spring 2011 Problem 5 3

How to take the test 3

Spring 10 Problem 2 2

Which sorts are stable sorts? 3

Spring 2010 Q1 (j) 2

Detecting Cycles 4

BFS/DFS Trees 6

Spring 2010 Q2 T/F 4
- An instructor thinks this is a good question

Spring 2010 Q2 #5 solution 3

Fall 2011 Recitation 19 Notes 2

Quiz 2 Wording 6
- 1 Unresolved Followup

Note History:

note 12

Quiz 2 stats

Quiz 2 grades are up now.

Prior to regrades:

STATISTICS (223 tests)
Mean 74.7
Median 75.5
Stdev 11.87
Max 98

HISTOGRAM

0 - 9	0
10 - 19	0
20 - 29	0
30 - 39	1
40 - 49	1
50 - 59	22
60 - 69	48
70 - 79	77 ←
80 - 89	44
90 - 99	30

Grade 72/100

(very happy :))

Quiz 1 stats: @268

#quiz2 #statistics #pin

follow 3 like 0 more ▾ 18 hours ago by Jeff Wu 1

followup discussions, *for lingering questions and comments*

● Resolved ○ Unresolved

Anonymous (1 hour ago) - Just clarification - that doesn't include those that dropped, rig

Jeff Wu **(Instructor)** (1 hour ago) - It includes everyone who took the test, including those v might have dropped (I don't think very many did though). The Quiz 1 stats also include those who dropped, and don't account for regrades.

Write a reply...

Click to start a new followup discussion

Average Response Time: 1 hr

Special Mentions: Shaunak Kishore **answered** Why is problem 1.h.ii ... in...

Online Now | Tl 5 |