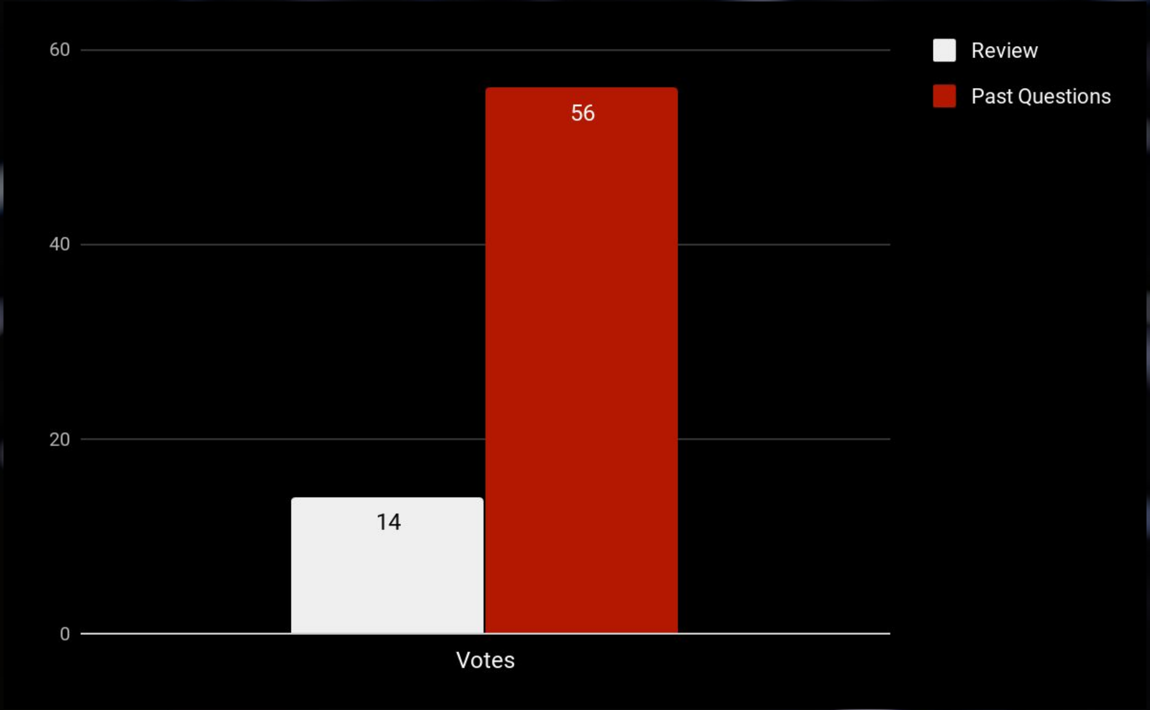
COMP 251 Final Review

Compete McGill x CSUS.

>Compete McGill_

Slides prepared by Imad Dodin and Andre Kaba.

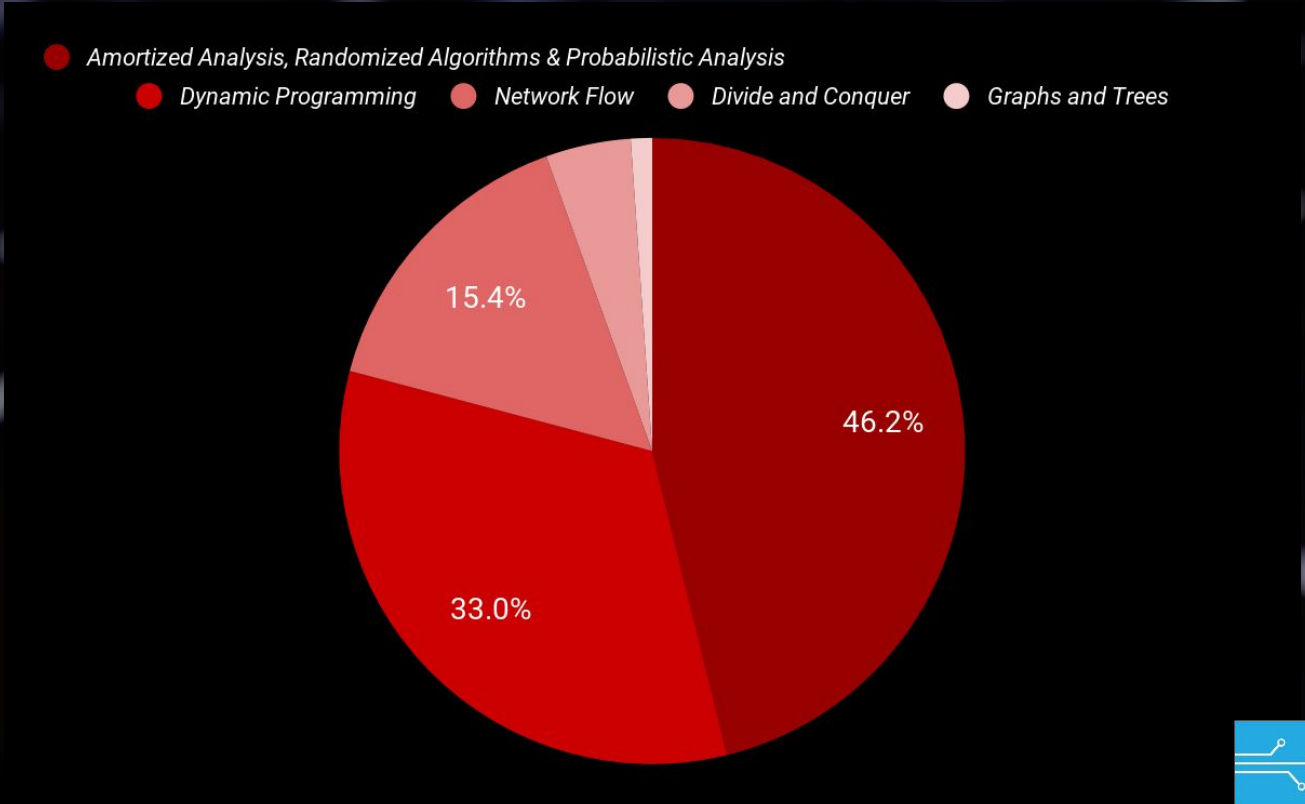
Facebook Survey Responses



>Compete McGill_

CSUS

Facebook Survey Responses (Cont.)



>Compete McGill_

This Session:

- We’ll go more in-depth for **Amortized Analysis (etc.) and Dynamic Programming** (you guys seem to really hate those topics!)
- We’ll do **brief reviews** for everything else, with a focus on solving **problems**!
- For each section:
  - Review → Answer Questions (above link and by hands) → Problem → Answer Questions!

Exam Structure

- → True / False Questions - **10%**
- → Short Answer Questions - **20%**
- → Algorithm Demonstrations - **60% (pt. 1)**
- → Algorithm Pseudo Code - **60% (pt. 2)**
- → Algorithm Proofs (Guided) - **60% (pt. 3)**
- → Multiple CHoice Questions - **10%**

Amortized Analysis

Amortized Analysis

"In an ***amortized analysis,*** we **average** the time required to perform a **sequence** of data-structure operations **over all the operations performed**. WIth amortized analysis, we can show that the **average cost of an operation is small,** if we average over a sequence of operations, even though a **single operation** within the sequence **may be expensive**. Amortized analysis differs from **average-case analysis** in that **probability is not involved**; an amortize**d analysis guarantees the *average performance of each operation in the worst case.*"**

- [CLRS 2009, Page 451]

Amortized Analysis (Cont.)

Key Points:

- Describe a **sequence** of operations **as a whole** in the **worst case.**
- E.g. Consider: **one** operation which is **expensive** (slow), but the rest are **cheap** (fast). With Amortized Analysis, we find that as a whole, the average cost is **small** (fast).
- Not probabilistic! Don't use probability! Probability is not here!
  - Probability is used in **Average-Case Analysis**.

>Compete McGill_

CSUS

Amortized Analysis Techniques

1. Aggregate Analysis
2. Accounting Method

Amortized Analysis: Aggregate Analysis

- "In **aggregate analysis**, we show that for all *n*, a sequence of n operations takes worst-case time *T(n)* in **total**. In the worst cases, the **average cost**, or amortized cost, per operation is therefore **T(n)/n**. Note that this amortized **cost applies to each operation**, even when there are several types of operations in the sequence."
  - [CLRS 2009, Page 452]

Amortized Analysis: Aggregate Analysis (Cont.)

Key Points:

- Total worst-case time is T(n), **for all n.**
- Amortized Cost **for any operation in the sequence is** T(n)/n.
- So, each operation is judged based on all of its peers!
    - We don't treat different types of operations differently!

Aggregate Analysis: Multipop

In stacks we have:

- **S.PUSH(x) : O(1) -** we say it has **cost of 1.**
- **S.POP(): O(1)** - we say it has **cost of 1.**

```
S.MULTIPOP(k)
while not S.EMPTY() and k>0
        S.POP()
        k--
```

Let's also add:

- **S.MULTIPOP(k): O(k)** - but it's **cost is really min(s,k)** where s is the number of objects in the Stack.

Let's consider all of these together as our "Stack Operations (Stack Bois)"

Aggregate Analysis: Multipop (Cont.)

So in a sequence of *n* Stack Bois (operations) we have:

- The worst-case runtime for each operation is O(n) - if it turns out that operation is MULTIPOP and the stack is filled with *n* elements.
- We also have *n* total operations, so our worst case run time is $O(n^2)$.

But that's not an Amortized Analysis: what's wrong with this?

Aggregate Analysis: Multipop (Cont.)

If we consider the sequence of operations as a whole, we find that the previous case is not actually possible.

- **# of Pop Operations (including in MULTIPOP) = # of Push Operations.**
- The sequence of operations therefore takes O(n) total time.
- Our amortized time is then O(n) / n = O(1).
  - This gives us an average worst case runtime per operation.
- The tricky part here was showing the O(n) worst case bound on the sequence of n operations.

Aggregate Analysis: Binary Counter

Consider an array that acts as a binary counter (i.e. represents a binary number):

0	1	2	3	4	5	6
1	0	0	0	0	0	0

$$[1,0,0,0,0,0,0]$$

→ Value is the sum of A[i] * 2^i (e.g. above gives one).

Aggregate Analysis: Binary Counter (Cont.)

We can define an operation for adding one to this Binary Counter Array:

A.INCREMENT()

$i = 0$
while i<*A.length* and A[i] == 1
    A[i] = 0
    $i = i + 1$
If i < A.*length*
    A[i] = 1

Each execution of INCREMENT() takes Θ(k) in the worst case.

A sequence of *n* INCREMENT() operation -s on A=0 (or any A) takes O(nk) in the worst case, where k is the length of A.

Aggregate Analysis: Binary Counter (Cont.)

We can find a tighter bound with Aggregate Analysis.

- → Obviously we do not have k bit flips for each increment operation.
  - ↳ Can we find the no. bit flips a sequence of n increment operations is restricted to? (Similarly to how we did for push / pop operations before?)
- → Try to look at the right hand side.

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Aggregate Analysis: Binary Counter (Cont.)

- The zero-th bit is flipped for every increment operation.
- The first bit is flipped every *second* increment operation.
- The second bit is flipped every *fourth* increment operation.
- The third bit is flipped every *eight* increment operation.

In general we have:

- The i-th bit is flipped $\lfloor n/2^i \rfloor$ times for *n* increment operations.
- By taking sums we find that the total number of bit flips for n operations is 2n.

Aggregate Analysis: Binary Counter (Cont.)

- Worst case runtime for n increment operations is therefore O(n)
- Performing Aggregate Analysis we get that the amortized cost per operation is O(n) / n = O(1).
- Again this is the average worst case runtime per operation.
  - (And furthermore, since we used aggregate analysis this is a representation of **all** the operations in our sequence - we don't treat different operations differently).

Amortized Analysis: Accounting Method

→ "In the ***accounting method*** of amortized analysis, we assign differing charges to different operations, with some operations charged more or less than they actually cost. We call the amount we charge an operation its ***amortized cost.*** When an operation's amortized cost exceeds its actual cost, we assign the different to specific objects in the data structure as ***credit.*** Credit can help pay for later operations whose amortized cost is less than their actual cost."

- [CLRS 2009, Page 456]

>Compete McGill_

Amortized Analysis: Accounting Method (Cont.)

Key Points:

- Assign a cost to each operation - known as ***amortized cost***.
  - Different to the ***average amortized cost per operation*** that we've been calculating up until now - we assign different costs for different operations.
- If we overshoot the ***amortized cost*** (i.e. the cost we assign is bigger than its actual cost, we assign the ***amount we overshot by*** as **credit.**

Accounting Method: Choosing Costs

- We need to choose costs such that **the total amortized cost** of a sequence of operations provides an **upper bound on the total actual cost** of the sequence.
  - **For an arbitrary sequence of operations!**
- This means the sum of amortized costs needs to be **greater than the sum of actual costs:**

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

>Compete McGill_

Accounting Method: Credits

- When we overshoot the amortized cost (i.e. estimate the amortized cost to be larger than the actual cost):
- We can take the difference (the amount we overshot by) as ***credit***.
- For each operation the credit is given by: $\hat{c}_i - c_i$
- And the total credit is given by: $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$
- **The total credit must remain non-negative to ensure an upper bound.**

>Compete McGill_

Accounting Method: Multipop

The **actual cost** for our Stack Bois are:

PUSH	-	**1**
POP	-	**1**
MULTIPOP	-	**min(s,k)**

But we can assign **amortized costs** of (we'll see why next):

PUSH	-	**2**
POP	-	**0**
MULTIPOP	-	**0**

Accounting Method: Multipop (Cont.)

>Compete McGill_

Accounting Method: Multipop (Cont.)

What we're doing:

PUSH	-	**2**
POP	-	**0**
MULTIPOP	-	**0**

- → We can only pop *after* we've pushed.
- → Assigning a cost of **2** to PUSH gives us a credit of **2-1=1**.
- → By assigning a cost of **2** to PUSH, we are keeping a credit of **1** to pay for its eventual **POP** operation (including the **POP**s that take place in **MULTIPOP**).

>Compete McGill_

CSUS

Accounting Method: Multipop (Cont.)

Proving that our assigned costs give an upper bound:

- **2 cases:**
- **Case I:** The element we push is never popped, so we've just overcharged the operation without using the credit - **Upper Bound**
- **Case 2:** The element we push is popped (in **POP** or **MULTIPOP**), so we use the credit - '**Correct Estimation' (Upper Bound)**.

>Compete McGill_

Accounting Method: Binary Counter

When we set a bit to 1, let's charge an **amortized cost** of **2**.

A cost of **1** goes to covering the actual cost of us flipping the bit. The **remaining 1** is considered credit to pay for us if / when we flip the bit back to **0**.

We can consider every **'1'** on our binary counter as having a credit of **1** to pay for it if/when its reset back to **0**.

A.INCREMENT()

$i = 0$
while i<$A.length$ and A[i] == 1
    A[i] = 0
    $i = i + 1$
If i < A.$length$
    A[i] = 1

Accounting Method: Binary Counter (Cont.)

So what is the amortized cost of the whole INCREMENT operation?

- All of the flips from 1→ 0 are already paid for by the credit we placed on our 1's so the while loop has a total amortized cost of 0.
- All we have to consider is the final 'if'-block of the operation, so the amortized cost of an INCREMENT operation is at most 2. (Constant time)
- For a sequence of *n* INCREMENT operations, we have a total amortized cost of O(n).

>Compete McGill_

Dynamic Tables

- We're not always able to predict the size of a table (e.g. HashTable) that we store objects in.
- It's useful to be able to ***dynamically expand / contract*** a table.
- When we **insert** an element - cost is potentially **big**, if it triggers an **expansion**.
- **Amortized Analysis** shows that the **Amortized** Cost of Insertion (and Deletion) is only **O(1)**.

>Compete McGill_

Dynamic Tables

- The **load factor** is, defined to be:

  α = no. items stored / size allocated to table.

- We obviously want to keep the load factor **under 1.**
- We also want to keep the load factor above some constant, **a**.
  - This ensures we haven't allocated *too big* of a table.

>Compete McGill_

Dynamic Tables - Insertion

TABLE-INSERT(T, x)

if *T.size* == 0

    allocate T.*table* with 1 slot

    T.*size* = 1

if *T.num* == *T.size*

    allocate *new-table* with $2 \cdot$ *T.size* slots

    insert all items in *T.table* into *new-table*

    free *T.table*

    T.size = $2 \cdot$ *T.size*

insert *x* into *T.table*

T.num = T.num + 1

Dynamic Table Insertion: Aggregate Analysis

- Disregard allocating tables / freeing memory steps.
  - 1st allocation is a constant cost that's only used once.
  - Allocation and freeing memory upon expansion is dominated by the cost of actually transferring the elements into the new table.
- We are left with just the elementary insertion steps, which are constant in time so we assign them a **cost of 1.**
- The cost of a single Table Insertion is then given by:

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of } 2\,, \\ 1 & \text{otherwise}\,. \end{cases}$$

>Compete McGill_

CSUS

Dynamic Table Insertion: Aggregate Analysis

- The total cost of *n* Insertions is the sum of the individual costs:
- An upper bound for this can be given by:

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n \,,$$

- 3n / n = **3**
  - Our **average amortized cost per operation** (a.k.a O(1))

>Compete McGill_

Dynamic Table Insertion: Accounting Method

- We, again, only consider the elementary insertions.
- Let's start by trying to assign a **cost of 2** to each elementary-insertion.

  1 - *Pays for the actual elementary insertion.*
  1 - *Pays for moving the element when we expand.*

- **Where is the problem?**
- We need to assign an **extra cost of 1** giving us an **amortized cost of 3.**

Questions?

>Compete McGill_

Problem 1: Let's say we added a MULTIPUSH operation with our Stack Bois, which pushes ***k*** elements onto the stack. Would our $O(1)$ bound on the amortized cost of stack operations still hold?

Solution 1:

No. The whole reason that we could lower the time complexity from $O(n^2)$ to $O(n)$ was that the runtime of the MULTIPOP operations was limited by the number of unpopped push operations preceding it. Also, ***more obviously,*** if our stack has no upper bound on its capacity, we could just keep executing **MULTIPUSH** which has a runtime of **$\Theta(k)$**.

Performing ***n*** **MULTIPUSH** operations, gives us a total runtime of **$\Theta(kn)$**. This gives us an amortized cost of **$\Theta(k)$.**

>Compete McGill_

CSUS

Problem 2: We perform a sequence of ***n*** operations wherein the *i*th operation costs *i* if *i* is an exact power of 3, else it is 1. Use aggregate analysis to determine the amortized cost per operation.

Solution 2:

Consider a sequence of ***n*** operations: We have $\lfloor \log_3 n \rfloor + 1$ powers of 3. We can calculate the **total cost** of the operations where i is a power of 3 as the **sum of all powers of 3** in the sequence. This is a **finite geometric sum.**

$$\sum_{i=0}^{\lfloor \log_3 n \rfloor} 3^i = \frac{3^{\lfloor \log_3 n \rfloor + 1} - 1}{3 - 1} < \frac{3n - 1}{2} < 3n$$

Adding the total cost of all other operations (= **n**), we get an **overall total cost of 4n**.

Our **average amortized cost per operation is then O(4n)/n = 0(1).**

Problem 3: Suppose we perform a sequence of stack operations on a stack whose size never exceeds k. After every k operations, we make a copy of the entire stack for backup purposes. Show that the cost of n stack operations, including copying the stack, is $O(n)$ by [the accounting method].

>Compete McGill_

Solution 3:

We know that ***once an element is pushed,*** it may ***potentially be popped*** or ***eventually copied.*** To cover both cases we can assign the following **amortized costs:**

PUSH	-	**3**
POP	-	**0**
COPY	-	**0**

This works because the cost of **3 covers the actual cost of pushing (1)** and gives us a **credit of 2** to **cover popping case and copying case.** *n* operations thus have an amortized cost of ***O(3n) = O(n).***

Randomized Algorithms

>Compete McGill_

Global Min Cut

A Global Min-Cut is a **partition** of a **graph vertices, V** into two subsets (A,B) such that the **number of edges between A** and **B** is minimized.

- We could do this using Network (Flows):
  - **Replace every edge (u,v) with 2 opposing edges (u,v) & (v,u).**
  - **Pick some vertex s, and compute the min s-v cut (as we do in Ford Fulkerson) for each other vertex, v.**
- The problem here is that we're kind of **assuming that it's easier to find s-t cuts than to find global min-cut → This is a wrong assumption!**

Karger's Contraction Algorithm

Contraction(V,E):

while $|V| > 2$:

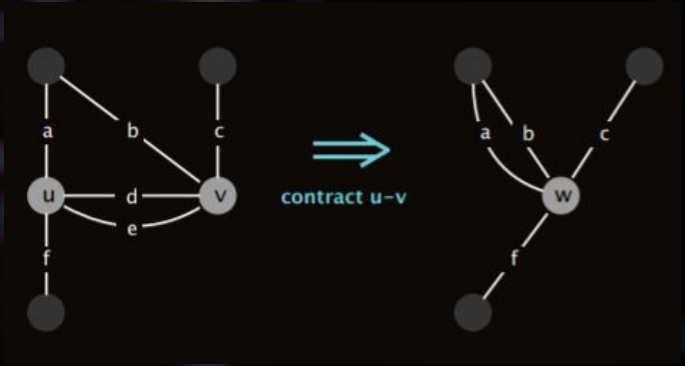
  choose $e \in E$ uniformly at random

  contract edge e

return remaining cut (all nodes contracted to form v_1)

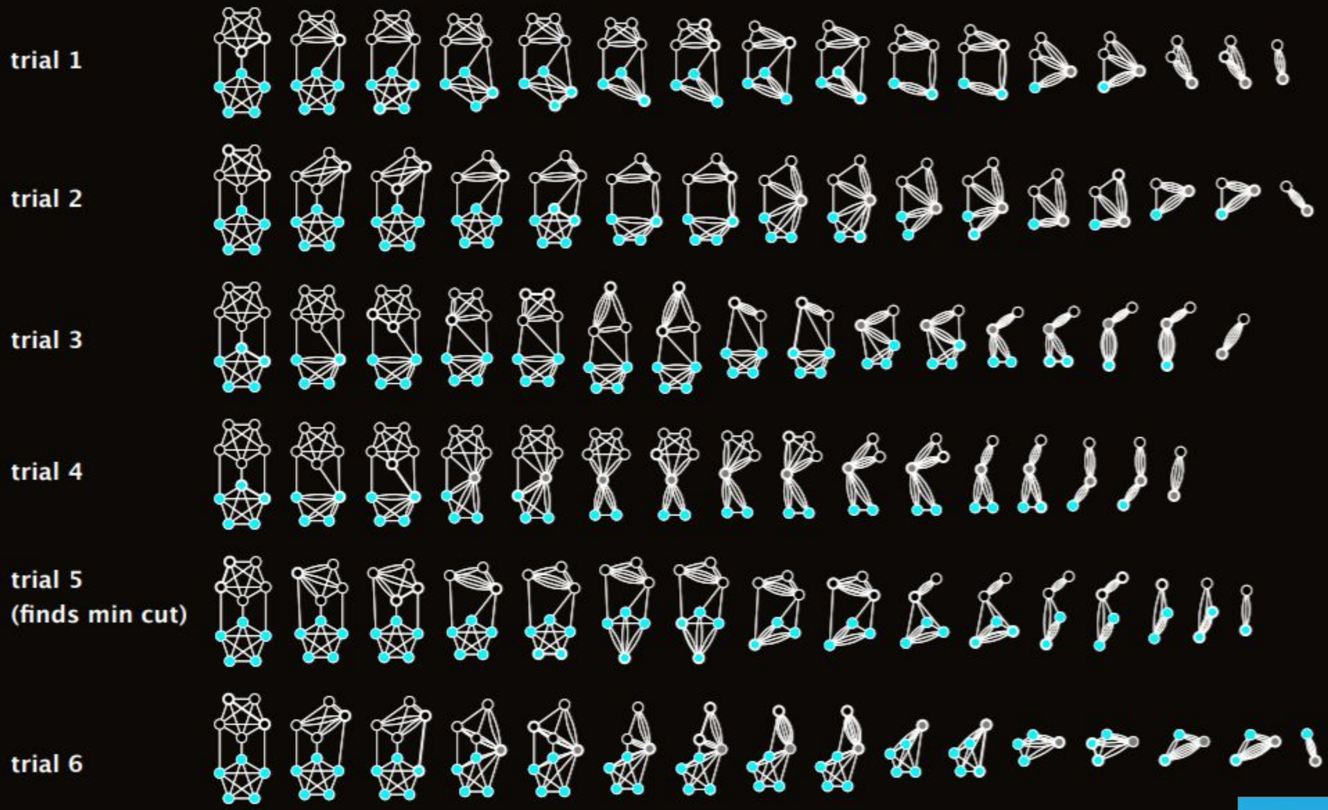
Contracting an edge, e:

- Replace u and v with a single node.
- Delete all self-loops.



>Compete McGill_

Contraction algorithm: example execution



...

Reference: Thore Husfeldt

Contraction Algorithm: Proof I

Prove that the contraction algorithm returns a min cut with a probability ≥ $2/n^2$

Let's say **F*** is the **set of edges in the global min-cut** (the ones that cross)
Let **k = |F*| = size of min cut.**
Note that every node in the graph must have a degree of at least k, or else the min-cut we have would not be a min-cut.
Recall also that the sum of degrees is twice the number of edges.
$2|E| \geq kn \rightarrow |E| \geq kn/2$

Contraction Algorithm: Proof I (Cont.)

Let $\mathbf{E_i}$ be the event that the algorithm **doesn't** contract an edge from **F** in step i. (What we want, *when it does contract - we've f***ed up*)

$$
\begin{aligned}
Pr(E_1) &= 1 - \frac{k}{|E|} \\
&\geq 1 - \frac{2}{n}
\end{aligned}
$$

$$
\begin{aligned}
Pr(E_2|E_1) &\geq 1 - \frac{2}{n-1} \\
Pr(E_i|E_1 \cap E_2 \cap \ldots) &\geq 1 - \frac{2}{n-i+1}
\end{aligned}
$$

>Compete McGill_

CSUS

Contraction Algorithm: Proof I (Cont.)

Probability that the Algorithm ***finds a Global Min-Cut*** = Probability that it ***hasn't contracted an edge that it wasn't meant*** to at every step of the algorithm.

$$
\begin{aligned}
Pr(\text{success}) &= Pr(E_1 \cap E_2 \cap \ldots \cap E_{n-2}) \\
&\geq Pr(E_1)Pr(E_2|E_1)\ldots Pr(E_{n-2}|E_1 \cap \ldots E_{n-3}) \\
&\geq (1-\frac{2}{n})(1-\frac{2}{n-1})\ldots(1-\frac{2}{3}) \\
&\geq 2\frac{(n-2)!}{n!} \geq \frac{2}{n(n-1)},
\end{aligned}
$$

>Compete McGill_

CSUS

Contraction Algorithm: Proof II - Amplification

If we run the algorithm many times, the probability that it succeeds at some point is greater (**its amplified).**

Prove that if we repeat the contraction algorithm $n^2 \ln n$ times, then the probability of failure is ≤ $1/n^2$.

$$\left(1-\frac{2}{n^2}\right)^{n^2\ln n} = \left[\left(1-\frac{2}{n^2}\right)^{\frac{1}{2}n^2}\right]^{2\ln n} \le \left(e^{-1}\right)^{2\ln n} = \frac{1}{n^2}$$

$(1 - 1/x)^x \le 1/e$

>Compete McGill_

Contraction - Crib Sheet Stuff

The overall running time is slow since we perform $\Theta(n^2 \log n)$ iterations with $\Omega(|E|)$ time giving us an overall complexity of **$O(n^2|E|\log n)$**

Early iterations are less at risk of f***ing up than later ones: probability of contracting an edge you weren't meant to hits 50% when $n/\sqrt{2}$ nodes remain. Run contraction algorithm **once until $n/\sqrt{2}$ nodes remain.** Run it **twice on resulting graph** and return the **best of two cuts. $O(n^2\log^3 n)$**

Best known improvement gives runtime of **$O(|E|\log^3 n)$**

Maximum 3-satisfiability

Satisfiability: When a boolean expression evaluates to true.

3-SAT: A boolean expression with 3 literals (elements) per clauses.

Maximum 3-satisfiability: Given a 3-SAT formula, find a truth assignment that satisfies as many clauses as possible.

The idea is we flip a coin for each element in the formula - if it comes up **Heads**, we set the element to **true,** if it is **Tails** we set it to **false**.

$$
\begin{aligned}
C_1 &= x_2 \lor \overline{x_3} \lor \overline{x_4} \\
C_2 &= x_2 \lor x_3 \lor \overline{x_4} \\
C_3 &= \overline{x_1} \lor x_2 \lor x_4 \\
C_4 &= \overline{x_1} \lor \overline{x_2} \lor x_3 \\
C_5 &= x_1 \lor \overline{x_2} \lor \overline{x_4}
\end{aligned}
$$

Maximum 3-satisfiability: Proof III - Expectation

Prove: Given a 3-SAT formula with *k* clauses, the expected number of clauses satisfied by a random assignment is *7k/8*.

We first **define a random variable** that tells us whether each clause is satisfied:

$$Z_j = \begin{cases} 1 & \text{if clause } C_j \text{ is satisfied} \\ 0 & \text{otherwise.} \end{cases}$$

Note that: $\Pr(Z_j=1) = 1 - (½)^3 = ⅞$

$$\begin{aligned} E[Z] &= \sum_{j=1}^{k} E[Z_j] \\ &= \sum_{j=1}^{k} \Pr[\text{clause } C_j \text{ is satisfied}] \\ &= \tfrac{7}{8} k \end{aligned}$$

Maximum 3-satisfiability: Proof III - Expectation

A **corollary** (by-product) of the previous proof is:
For any instance of 3-SAT, there exists a truth assignment that satisfied at least a ⅞ fraction of all clauses.

This is pretty trivial to prove: A Discrete Random Variable is going to attain at least its expected value *some* of the time.

The Probabilistic Method

This is when you use a Random Variable to produce the property you are trying to prove with a positive probability.

Property: *For any instance of 3-SAT, there exists a truth assignment that satisfies at least ⅞ of the clauses.* ***It is a ⅞-approximation algorithm.***

The Probabilistic Method

Pf. Let p_j be probability that exactly j clauses are satisfied; let p be probability that $\geq 7k/8$ clauses are satisfied.

$$\begin{aligned}
\tfrac{7}{8}k \;=\; E[Z] \;&=\; \sum_{j \geq 0} j\,p_j \\
&=\; \sum_{j < 7k/8} j\,p_j \;+\; \sum_{j \geq 7k/8} j\,p_j \\
&\leq\; \left(\tfrac{7k}{8} - \tfrac{1}{8}\right) \sum_{j < 7k/8} p_j \;+\; k \sum_{j \geq 7k/8} p_j \\
&\leq\; \left(\tfrac{7}{8}k - \tfrac{1}{8}\right) \cdot 1 \;+\; k\,p
\end{aligned}$$

Rearranging terms yields $p \geq 1/(8k)$. ▪

>Compete McGill_

Questions?

>Compete McGill_

Problem 1: An execution of the contraction algorithm will find the global min-cut if it never contracts an edge that crosses a global min-cut. Is this assertion true or false?

>Compete McGill_

Solution 1

True, if the global min-cut is unique.

The logic suggested by the question is:

Never contracting an edge that crosses a global min-cut ⇒ A Global Min-Cut is found.

This is conditional on the global min-cut being unique:

If it is not unique then, if we never contract an edge that crosses a global min-cut we won’t contract required edges to find the global min-cut.

Problem 2: Suppose we repeat the contraction algorithm n^3 times. Which of the following statements are correct?

A. We are guaranteed to find the global min cut
B. The probability of failing to find the min cut is less than $1/e^n$
C. The probability of failing to find the min cut is less than $1/e^{n\ln(n)}$
D. The probability of failing to find the min cut is less than $1/e^{n^2}$
E. Repeating the algorithm another *n* times increases the probability of finding the min cut.
F. No matter how many times we repeat the algorithm, there is a chance we fail to find the min cut.

Solution 2

A. We are guaranteed to find the global min cut
B. The probability of failing to find the min cut is less than $1/e^{n}$
C. THe probability of failing to find the min cut is less than $1/e^{n\ln(n)}$
D. The probability of failing to find the min cut is less than $1/e^{n^2}$
E. Repeating the algorithm another *n* times increases the probability of finding the min cut.
F. No matter how many times we repeat the algorithm, there is a chance we fail to find the min cut.

Problem 3: There exists at least one truth assignment that satisfies any 3-SAT formula with 7 clauses - T/F?

>Compete McGill_

Solution 3

The expectation ⅞ * k. For k=7, this gives an expectation of 49/8

This is 6.125 clauses - Recall that we need to have at least some value of our random variable that surpasses the expectation and since the number of clauses satisfied is strictly a natural number, we must have at least some outcome that satisfies 7 clauses (i.e. all the clauses).

So the answer is **True.**

Probabilistic Analysis of Quick Sort

>Compete McGill_

What the f**k is quick sort?

1. Given an array **A=[a1, ..., an]**
2. Pick an element as a pivot **p** (Somehow)
3. Split A into two arrays (Partition step)
    a. A1 = { all elements < **p**}
    b. A2 = { all elements > **p**}
4. Recursively sort **A1**, **A2**
5. Then the sorted array is **A1 + [p] + A2** (+ means array concatenation)

Animation: https://visualgo.net/en/sorting

>Compete McGill_

CSUS

Worst case

Recursion tree for
worst-case partition

$$\underbrace{n \to n-1 \to n-2 \to n-3 \cdots 2 \to 1}_{n}$$

Split off a single element at each level:

$$
\begin{aligned}
T(n) &= T(n-1) + T(0) + \text{PartitionTime}(n) \\
&= T(n-1) + \Theta(n) \\
&= \textstyle\sum_{k=1 \text{ to } n} \Theta(k) \\
&= \Theta(\textstyle\sum_{k=1 \text{ to } n} k\,) \\
&= \Theta(n^2)
\end{aligned}
$$

Randomized quicksort

- **Deterministic Algorithm :** Exactly same **behavior** for different runs for the same input.
- **Randomized Algorithm : Behavior** might differ for different runs for the same input.
  - **Watch out!** Behaviour not output is what matters.

- To make an algorithm randomized we need to introduce a random decision somewhere. For quicksort we can pick the pivot at random!

Terminology

- Random variable **X** is a variable that take values **{x1,...,xn}** with probabilities **{p1, p2, ... ,pn}** such that **p1 + p2 + ... + pn = 1**

- Expectation of a random variable is the weighted average of all the values **X** takes. **E[x]=(x1*P(X=x1) + ... + xn*P(X=xn))**
  - One can think about it as the value we expect from **X** on average

- Indicator variable **I** is a random variable only taking values **{1, 0}** with probabilities **{p, 1-p}.** We also have **E[I]=0*P(I=0)+1*P(I=1)=P(I=1)=p**

>Compete McGill_

CSUS

Analysis:

Notation:

- Let $z_1, z_2, \ldots, z_n$ denote the list items (in sorted order).
- Let $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$.

Let RV $X_{ij} = \begin{cases} 1 & \text{if } z_i \text{ is compared to } z_j \\ 0 & \text{otherwise} \end{cases}$

X_{ij} is an **indicator random variable**.
$X_{ij}=I\{z_i \text{ is compared to } z_j\}$.

Thus, $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$.

Analysis:

$$E[X] = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right]$$

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}]$$

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} P[z_i \text{ is compared to } z_j]$$

Reminder:

$$E[X_{ij}] = 0 \cdot P[X_{ij}=0] + 1 \cdot P[X_{ij}=1]$$
$$= P[X_{ij}=1]$$

Analysis:

So, $P[z_i \text{ is compared to } z_j] = P[z_i \text{ or } z_j \text{ is first pivot from } Z_{ij}]$

$$= P[z_i \text{ is first pivot from } Z_{ij}]$$

$$+ P[z_j \text{ is first pivot from } Z_{ij}]$$

$$\boxed{= \frac{1}{j-i+1} + \frac{1}{j-i+1}}$$

$$= \frac{2}{j-i+1}$$

>Compete McGill_

CSUS

Analysis:

- **zi** will be compared to **zj** if and only if the first element to be picked as the pivot from **[zi, … ,zj]** is either **zi** or **zj**.
- Imagine we are throwing darts:
  - If we hit **zi** or **zj** then **Xij=1**
  - If we hit [**zi+1** … **zj-1**] then **Xij=0**
  - If we hit [**z1** … **zi-1**] or [**zj+1** … **zn**] then we need to try again!

…	…	zi-1	zi	…	…	zj	zj+1	…

Analysis:

$$
\begin{aligned}
E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= \boxed{O(n \lg n).}
\end{aligned}
$$

Substitute $k = j - i$.

$$\sum_{k=1}^{n} \frac{1}{k} = H_n \quad (n^{\text{th}} \text{ Harmonic number})$$

$$H_n = \ln n + O(1)$$

Analysis:

You can think about this like this:

$$\sum_{x=1}^{n} \frac{1}{x} \leq \int_{1}^{n} \frac{1}{x}\, dx$$

Not claiming this is always true.

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$$

Substitute $k = j - i$.

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$

$$= \sum_{i=1}^{n-1} O(\lg n)$$

$$\sum_{k=1}^{n} \frac{1}{k} = H_n \text{ (n}^{\text{th}} \text{ Harmonic number)}$$

$$H_n = \ln n + O(1)$$

$$= \boxed{O(n \lg n).}$$

Dynamic Programming

>Compete McGill_

CSUS

Plan for this part:

- Very quick intro to what is DP.
- Practice final sample problem walkthrough.
- Summarize Jerome's slides.

The idea

1. Split the current problem into multiple smaller sub-problems.
2. Solve all possible sub-problems.
3. Assemble them to build up solutions to the current problem.
   a. Usually, Bottom-up but no necessarily.
4. **Crib sheet stuff**: Dynamic programming is always one of these themes:
   a. Maximization.
   b. Minimization
   c. Counting.

→ Easiest way to learn dynamic programming is to solve problems!

The idea

1. Split the current problem into multiple smaller sub-problems.
2. Solve all possible sub-problems.
3. Assemble them to build up solutions to the current problem.
   a. Usually, Bottom-up but no necessarily.

4. **Crib sheet stuff**: Dynamic programming is always either a
   a. A Maximization problem
   b. Or a Minimization problem
   c. Or a Counting problem

Coin change:

Given a set of m types of coins $C=\{c1,\dots,cm\}$ such that $ci>0$ and a number $k>0$. Give the minimum size of a subset of C, call it S, such that $|S|=k$.

>Compete McGill_

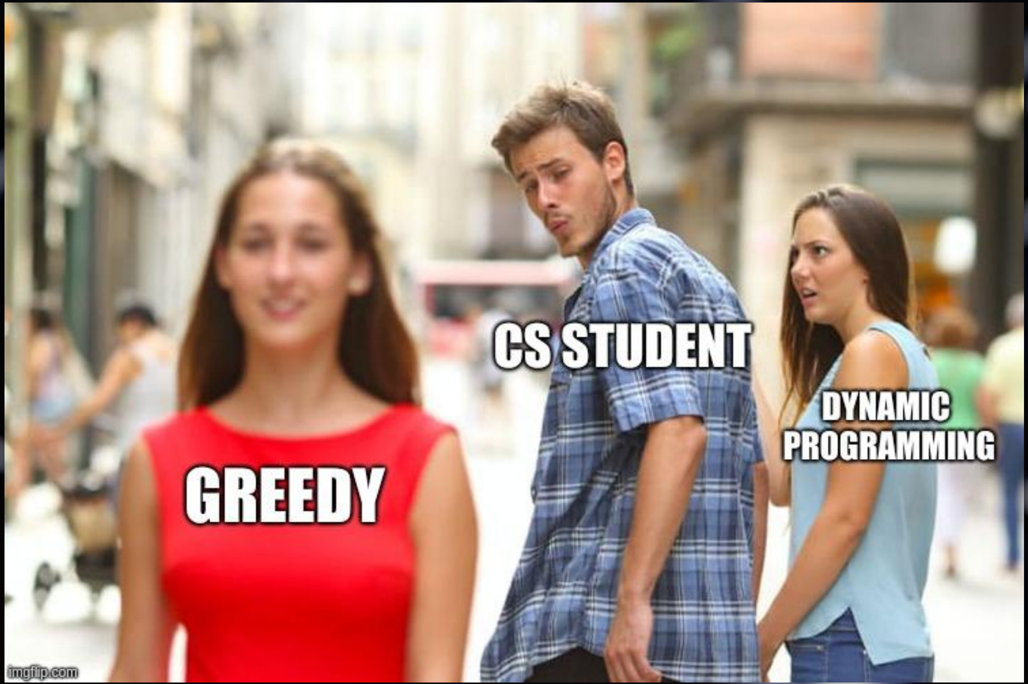
CSUS

Comparison: [Brute force]

1. Try every possible subset
2. Can be pruned and optimized, but still exponential complexity.
3. Example:
   a. C={1, 3, 4}, k=6
   b. {1}, {3}, {4}, {1,1}, {1,3}, {1,4}, {3,1}, **{3,3}**, {3,4}, {4,1}, {4,3}, {4,4}, {1,1,1}, ...

- → Assume we want to try sets of size k, we have 3^k possible sets.
- → The total number of sets to try is (|C| + |C|^2 + ... + |C|^k) so complexity is ~ O(|C|^(k+1))

What about Greedy?



Comparison: [Greedy]

- We want minimum |S| then why not pick the largest coin the fits?
- Example:
  - C={1, 3, 4}, k=6
  - k=6, S={}
  - k=2, S={4}
  - k=1, S={1,4}
  - **k=0, S={1, 1, 4} => solution is 3?**

- Optimal solution is actually 2 (S={3,3})

Comparison: [Dynamic Programming]

- We want to split to subproblems.
- Assume C={1, 3, 4}, k=6
  - Min. number of coins for 6 have to be one of these:
    - 1 + Min. number of coins for 5
    - 1 + Min. number of coins for 3
    - 1 + Min. number of coins for 2
  - We recursively solve for 2, 3, 5 then take the minimum.

Comparison: [Dynamic Programming]

- We want to split to subproblems.
- Assume C={1, 3, 4}, k=6
  - Min. number of coins for 6 have to be one of these:
    - 1 + Min. number of coins for 5
    - 1 + Min. number of coins for 3
    - 1 + Min. number of coins for 2
  - We recursively solve for 2, 3, 5 then take the minimum.

$$F[0] = 0$$
$$F[i] = \min_{i=1\ldots m} \{1 + F[i - c_i]\}$$

>Compete McGill_

CSUS

Example

→ C={1, 3, 4}, k=6

0	1	2	3	4	5	6
0	Infinity	Infinity	Infinity	Infinity	Infinity	Infinity

>Compete McGill_

Example (Cont.)

→ C={1, 3, 4}, k=6

0	1	2	3	4	5	6
0	1	Infinity	Infinity	Infinity	Infinity	Infinity

>Compete McGill_

Example (Cont.)

- C={1, 3, 4}, k=6

0	1	2	3	4	5	6
0	1	2	Infinity	Infinity	Infinity	Infinity

>Compete McGill_

Example (Cont.)

→ C={1, 3, 4}, k=6

0	1	2	3	4	5	6
0	1	2	1	Infinity	Infinity	Infinity

>Compete McGill_

Example (Cont.)

→ C={1, 3, 4}, k=6

0	1	2	3	4	5	6
0	1	2	1	1	Infinity	Infinity

>Compete McGill_

Example (Cont.)

→ C={1, 3, 4}, k=6

0	1	2	3	4	5	6
0	1	2	1	1	2	Infinity

>Compete McGill_

CSUS

Example (Cont.)

→ C={1, 3, 4}, k=6

0	1	2	3	4	5	6
0	1	2	1	1	2	2

>Compete McGill_

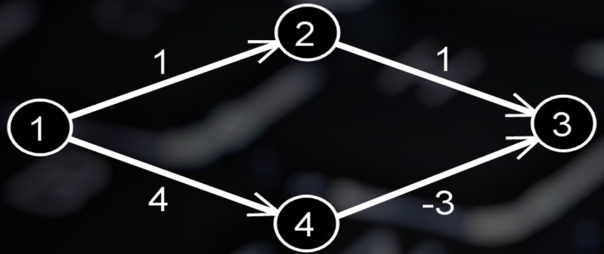
Bellman-Ford Algorithm

>Compete McGill_

CSUS

Shortest path with negative weights

- Given a weighted graph G(V,E) find the shortest path from u to all other vertices in the graph.



- If we have negative weights Dikstra's will probably f**k up.
- If negative cycles exist, then shortest path is ill-defined.

>Compete McGill_

Bellman-ford

Bellman-Ford Algorithm

- Allows negative-weight edges.
- Computes *d*[v] and π[v] for all v ∈ *V*.
- Returns TRUE if no negative-weight cycles reachable from *s*, FALSE otherwise.

If Bellman-Ford has not converged after V(G) - 1 iterations, then there cannot be a shortest path tree, so there must be a negative weight cycle.

Bellman-ford (Cont.)

- The idea is that we find shortest path of **length 1**, then using those we find paths of **length 2,** ..., up to paths of **length |V|-1**

- Shortest paths are constructed from other shortest paths
- After |V|-1 iterations, we should have computed all shortest paths.
  - Because we cannot have a shortest path with length greater than |V|-1
  - If not, then there is **negative cycle!**

>Compete McGill_

Bellman-ford (Cont.)

Bellman-Ford Algorithm

- Can have negative-weight edges.
- Will "detect" **reachable** negative-weight cycles.

```
Initialize(G, s);
for i := 1 to |V[G]| −1 do
    for each (u, v) in E[G] do
        Relax(u, v, w)
for each (u, v) in E[G] do
    if d[v] > d[u] + w(u, v) then
        return false
return true
```

Time Complexity is O(VE).

>Compete McGill_

Table view

Another Look at Bellman-Ford

Note: This is essentially **dynamic programming**.
Let d(i, j) = cost of the shortest path from s to i that is at most j hops.

$$d(i,j) = \begin{cases} 0 & \text{if } i = s \wedge j = 0 \\ \infty & \text{if } i \neq s \wedge j = 0 \\ \min(\{d(k, j-1) + w(k, i) : i \in Adj(k)\} \cup \{d(i, j-1)\}) & \text{if } j > 0 \end{cases}$$

j ↓ \ i →	z (1)	u (2)	v (3)	x (4)	y (5)
0	0	∞	∞	∞	∞
1	0	6	∞	7	∞
2	0	6	4	7	2
3	0	2	4	7	2
4	0	2	4	7	–2

>Compete McGill_

CSUS

Knapsack

>Compete McGill_

CSUS

Definition

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ and has value $v_i > 0$.
- Knapsack has capacity of W.
- Goal: fill knapsack so as to maximize total value.

Ex. $\{1, 2, 5\}$ has value 35.
Ex. $\{3, 4\}$ has value 40.
Ex. $\{3, 5\}$ has value 46 (but exceeds weight limit).

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance
(weight limit W = 11)

Greedy by value. Repeatedly add item with maximum v_i.
Greedy by weight. Repeatedly add item with minimum w_i.
Greedy by ratio. Repeatedly add item with maximum ratio v_i / w_i.

Observation. None of greedy algorithms is optimal.

Idea

- For each item we can either take it or leave it.
  - If we take it then W=W-wi and V=V+vi
  - If not, we just move to the next item
- We need the current weight to be a parameter to our function because otherwise there would be no way of telling if we can fit more items!
- So our solution can be a function **OPT(i, w)**
  - **i** means we are considering items **1, 2, … , i**
  - **w** is the available weight
- Complexity is **O(nW)** which is **pseudo-polynomial**

>Compete McGill_

Dynamic programming solution

Def. $OPT(i, w)$ = max profit subset of items $1, \ldots, i$ with weight limit w.

Case 1. OPT does not select item i.

- OPT selects best of $\{ 1, 2, \ldots, i-1 \}$ using weight limit w.

Case 2. OPT selects item i.

- New weight limit $= w - w_i$.
- OPT selects best of $\{ 1, 2, \ldots, i-1 \}$ using this new weight limit.

optimal substructure property
(proof via exchange argument)

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{ OPT(i-1, w),\ \ v_i + OPT(i-1, w-w_i) \} & \text{otherwise} \end{cases}$$

>Compete McGill_

CSUS

Dynamic programming solution (Cont.)

KNAPSACK (n, W, w_1, ..., w_n, v_1, ..., v_n)

FOR $w = 0$ TO W
  $M[0, w] \leftarrow 0$.

FOR $i = 1$ TO n
  FOR $w = 1$ TO W
  IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w]$.
  ELSE $M[i, w] \leftarrow \max \{ M[i-1, w], v_i + M[i-1, w-w_i] \}$.

RETURN $M[n, W]$.

Table view

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w),\ v_i + OPT(i-1,w-w_i)\} & \text{otherwise} \end{cases}$$

weight limit w

subset of items 1, …, i	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

>Compete McGill_

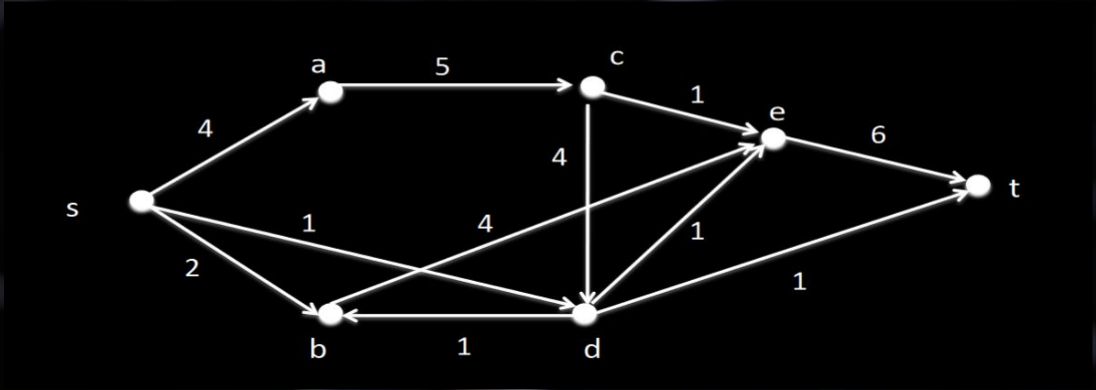
Activity-selection Problem
+
Pairwise sequence alignment

>Compete McGill_

Network Flows

>Compete McGill_

Problem 1: Apply Ford-Fulkerson while choosing the shortest augmenting path at every iteration:



>Compete McGill_

CSUS

Solution 1

See board.

Problem 2: Prove that any maximum s-t flow has a finite optimal solution if and only if there is no directed path from s to t consisting only of infinite capacity arcs.

Solution 2

If G contains a path, P, with infinite capacity for some path from s to t then we can send an infinite amount of flow from s to t using this path. **(No finite and optimal solution)**

Suppose there is no such path. Let S be the set of nodes reachable from s using infinite capacity arcs. Obviously, s is contained within S and t is not (since we don't have an infinite capacity path to t) - S is an s-t cut. There are only finite capacity arcs leaving S, otherwise we can grow S more. The cut given by S has a finite capacity and so there exists a minimum-cut with finite capacity (whether or not it is S). Since the capacity of such a cut is finite, by Max-Flow, Min-Cut, so is the max flow.

Questions?

>Compete McGill_

Good luck!

>Compete McGill_