Chris Hodapp
chodapp3@gatech.edu

2017-01-11, CS6505 (Computability, Complexity, & Algorithms)

CLRS Ch. 2 (CLRS = 'Introduction to Algorithms', Cormen, Leiserson, Rivest, Stein)

- Insertion sort
- Showing correctness: One approach is loop invariants. Must show:
  - Init: Invariant is true prior to 1st iter.
  - Maintenance: If it's true at start of iteration, it must be true at end.
  - Termination: Invariant tells something useful at loop's termination.
- Note relation to induction.
- Analyzing algorithm: Predicting resources (usually time, but sometimes memory, bandwidth, hardware...)
  - Typically, running time relative to input size. Both depend a lot on context, but loosely, # of steps vs. # of items.
  - Worst-case & average-case analysis
    Upper bound
  - Note that for some algorithms or applications, almost all inputs will be "worst-case" inputs.
- Order of growth: Ignore constant-factors, lower-order terms
- Designing algorithms:
  - Incremental approach (as in insertion sort)
  - Divide & conquer (typical for recursive algorithms)
    - Break into subproblems, solve subproblems, & combine into solution
    - e.g. merge sort
- p39, recursion tree example

2017-01-12

Pryby lecture: Intro to Algorithm Design & Analysis, part 1

- Sorting problem
  - input: sequence of $n$ numbers $(a_0, a_1, \ldots a_{n-1})$
  - output: a permutation of the input, $(a'_0, a'_1, \ldots a'_{n-1})$ s.t. $a'_0 \leq a'_1 \leq \ldots \leq a'_{n-1}$

N.B. We use 0-based indexing but CLRS uses 1-based.

- But what makes a "good" sorting algorithm?
  - fast
  - doesn't use a lot of space (obviously we need space for elements though)

N.B. CLRS says that checking initialization in 'for' loop should be done after variable value is set but before checking loop's condition or doing anything else.

2017-01-12

Pyby intro, part 1 (cont.)

- Some bad ones: bogosort, bozosort, worstsort
- But for a microcontroller, for instance, perhaps we will accept a drop in speed in exchange for fixed memory usage
- Prerequisites of this:
  - Asymptotic notation - $O(n^2), \Omega, \Theta$
  - Mathematical induction

  ] See mini-lessons on these two

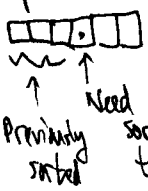
- Insertion sort

Design / Analysis

1. Basic idea of algorithm
2. Implement in pseudocode
3. Prove algorithm is correct
4. Analysis of performance

] He recommends following this for other algorithms too

- Basic idea:




eg. 1 4 2 5 ↓

→ 1 2 4 5 ↓

→ 1 2 4 5

- For 'new' element: compare with all previously-sorted elements, ~~left-to-right~~ starting at end, until we reach a smaller element
- Pseudocode (he does it Python-like):

```
def insertion_sort(A):
    n = len(A)
    for i in range(1,n):
        j = i-1
        while j≥0 and A[j] > A[j+1]:
            A[j], A[j+1] = A[j+1], A[j]
            j -= 1
```

This is in-place sort, by the way.

- Correctness proof: We use the loop invariant.
  - Invariant: On iteration $i$, sublist $A[0:i]$ consists of the numbers $(A[0], \ldots A[i-1])$ - the original first $i$ elements - but in sorted order.

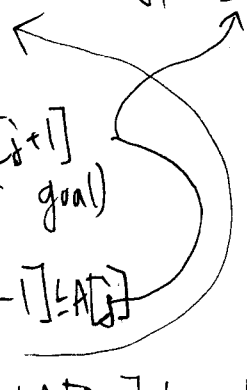
  ⇒ Iteration $n$ means $A[0:n]$ contains $A$ in sorted order.

Chris Hodapp
chodapp3@gatech.edu

2017-01-12 (b), CS6505 (Computability, Complexity, & Algorithms)

Pyguy intro, part 1 (cont.)

- That's a definition under which the function is correct (provided we can prove it)
- First: Prove invariant at beginning of 1st iteration. ] Base case
  $i=0$, $A[0:i]$ is just 1st element of $A$, which is trivially true.
- Inductive step: If invariant holds at $i$, it holds at $i+1$.
  $A[0:i]$ is first $i$ elements of $A$ in sorted order
  $A[i]$ has not been touched before $i^{th}$ iteration.
  'while' loop only affects elements to the left of $A[i+1]$
  Need to show: At end of iteration $i$, $A[0,i+1]$ is first $i+1$ elements of $A$ in sorted order.
  - N.B. You need loop invariants for each loop, so this includes the inner 'for' loop.
    - He ignores it for now! At start of iteration $j$, $A[0:j+1]$ and $A[j+1:i+1]$ are sorted.
    - Inner loop terminates at: $j=-1$, or $A[j] \leq A[j+1]$
    - If $j=-1$, $A[0,i+1]$ is sorted (our goal)
    - If $A[j] \leq A[j+1]$:
      $A[j]$: $A[0] \leq A[1] \leq \ldots \leq A[j-1] \leq A[j]$
      $A[j+1]$: $A[j+1] \leq A[j+2] \leq \ldots$
      But then $A[0] \leq A[1] \leq \ldots \leq A[j] \leq A[j+1] \leq \ldots A[i]$
      So $A[0,i+1]$ is sorted!



- Thus by induction, it appears to be proven correct.
- Performance of insertion sort (relative to input size)
  - Assume 1 CPU, RAM, & count basic arithmetic ops as single steps
  - Usually # of items in input (sometimes # of bits) & # of operations as a function of input size

- Insertion sort has various constant-size operations inside loops, so we need to count the iterations sometimes
  - We do $i$ iterations inside inner loop, $O(i)$ ops ($O(i) \leq ci$)
  - So total ops in for-loop: $c \cdot 1 + c \cdot 2 + c \cdot 3 + \ldots c(n-1) = \frac{c}{2}(n^2 - n)$
    $\rightarrow O(n^2)$
- But, what about special cases of input?
  - Already sorted: Inner loop is $\Theta(1)$, run $n$ times. So $\Theta(n)$ total (best case - algorithm must at least look at $n$ elements. All sorting algos are $\Omega(n)$ - best case)
  - Sorted in opposite order: Worst case, still $\Theta(n^2)$
  - What is average-case? Still $\Theta(n^2)$. (Average might as well be worst-case. Not all algorithms are this way.)
- So... Insertion sort is okay if we know input's already sorted or mostly sorted. Outside of that, much better algorithms exist. (For small lists, it's fine though. It's easy to do and has very low overhead; it's faster than "better" algorithms much of the time for that range of $n$.)

Pryby intro, part 2

- Divide & Conquer
  - Another approach to algorithm design
  - Divide, Conquer (Recurse), Combine
  - How might we sort a list this way?
    1. Split input into two sublists about 1/2 size
    2. Recursively sort sublists
    3. Merge sorted sublists
       - Start at "front" elements of each, and add the smaller of the two, removing it from sublist

] Merge Sort

Chris Hodapp
chodapp3@gatech.edu

2017-01-16, CS6505 (Computability, Complexity, & Algorithms)

Prgbuy intro, part 2 (cont.)

```
def merge_sort(A, l, r):
    n = r - l
    if n > 1:
        merge_sort(A, l, l + n/2)
        merge_sort(A, l + n/2, r)
        merge(A, l, l + n/2, r)
```

N.B. n/2 is floor division

merge(..., ..., ..., ...)
left index, midpoint index (between sublists), right index

```
def merge(A, l, m, r):
    ~~n1 = m - l~~
    ~~n2 = r - m~~
    L = A[l:m] + [float('inf')]
    R = A[m:r] + [float('inf')]
    i, j = 0
    while i+j < r-l:
        if L[i] <= R[j]:
            A[l+i+j] = L[i]
            i += 1
        else:
            A[l+i+j] = R[j]
            j += 1
```

actually not needed

lengths of left & right sublist

- Pythonic ∞ (sentinel)
- Indexing is like in Python too
- Note that i+j always increments
- This is in-place

- Showing correctness
  - First, for while loop: $S_{i,j}$: Before iteration i, j,
    a) A[l:l+i+j] contains i+j smallest elements of L & R in sorted order
    b) L[i] & R[j] are the smallest elements of L & R, respectively, not yet copied back into A.
  - Init: Prove $S_{00}$ is true (before loop)
    - A[l:l] is empty, thus trivially true
  - Maintenance: $S_{i,j} \Rightarrow S_{i+1,j}$ & $S_{i,j+1}$
    - Without loss of generality, assume $L[i] \leq R[j]$; assume all of $S_{i,j}$ is true for inductive step

Pryby into, part 2 (cont.)

(a)
- $L[i] \le R[j] \Rightarrow L[i]$ is smallest of L not in A
  After copy ($A[l+i+j] = L[i]$), $A[l:l+i+j+1]$ will have $i+j+1$ smallest elements of L & R (note $L[i] \le R[j]$).
  Since (per $S_{i,j}$) A was already sorted, adding new element that is next-smallest (at end of A) keeps A sorted.

(b)
- Since L & R were sorted already, $L[i+1]$ is smallest element of $L[i+1:]$, thus smallest element of A not copied yet.

- Next: Check <u>termination</u>.
  - Loop terminates only at $i+j \ge r-l$, at end of iteration.
  - Property at start of iteration is true at end ($S_{i,j}$)
  - Thus, $S_{i,j}$ is a loop invariant.
- When $i+j = r-l$, $S_{i,j}$ says $A[l:l+i+j]$ has L&R sorted, but $A[l:l+i+j] = A[l:l+r-l] = A[l:r]$ and it contains none of the sentinels.
- Thus merge function is right.
- Next: merge_sort. Since recursive, use induction.
  - Base case: $r-l \le 1$, then A is empty or has one element, so trivially, it's sorted
  - Inductive step:

Strong induction

suppose merge_sort works correctly for <u>all</u> $n < r-l$. Then the recursive calls to merge_sort function correctly (they're called on $n < r-l$), and merge (as proven) combines two sorted sublists into single sorted list. Q.E.D.

- Analysis:
  Let $T(n)$ = performance of merge_sort on list of n elements.
  - Recursive calls to merge_sort are $T(n/2)$.
  - merge is $\Theta(r-l)$ due to loop, thus $\Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & n \le 1 \\ 2 \cdot T(n/2) + \Theta(n) & \end{cases} \rightarrow \text{Recurrence relation}$$

Chris Hodapp
chodapp3@gatech.edu

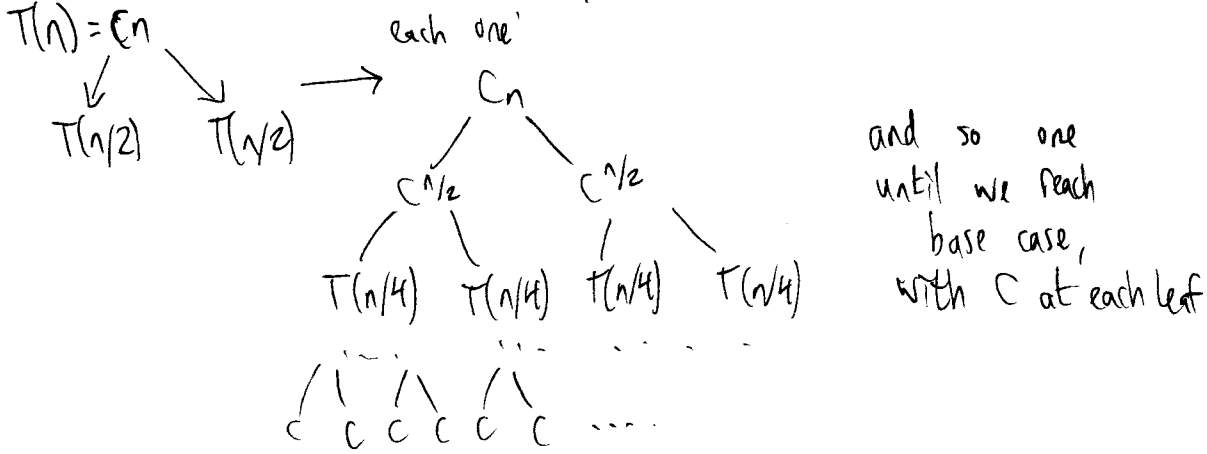
2017-01-16b, CS6505 (Computability, Complexity, & Algorithms)

Pryby intro, part 2 (cont.)

- In general for divide & conquer:
  - $a$ equally-sized subproblems
  - $n/b$ in size
  - $C(n)$ combine time
  - $D(n)$ divide time

$$\rightarrow T(n) = \begin{cases} \Theta(1) & n \leq n_0 \\ a \cdot T(n/b) + C(n) + D(n), & n > n_0 \end{cases}$$

- One heuristic here: Assume $n$ is power of 2




Cn is constant time part of each iter

and so one until we reach base case, with C at each leaf

- So each level has $Cn$ total, so...
  $Cn + Cn + \dots + Cn = d \cdot Cn$ where $d$ is tree depth
  - We divide $n$ by 2 $d$ times until it equals 1, so:
    $n/2^d = 1 \rightarrow d = \log_2 n \rightarrow T(n) = Cn \log_2 n$
  - We can show a similar lower bound, then $T(n) = \Theta(n \log n)$.
- This isn't a formal proof, but the answer is right.
- Note that runtime here depends only on $n$, not on contents of list. Merge sort has a fairly constant runtime & call pattern. However, it does create many temporary lists (it needs $\Theta(n)$ auxiliary space, while $\Theta(1)$ for insertion sort).

Chris Hodapp
chodapp3@gatech.edu

2017-01-16c CS6505 (Computability, Complexity, & Algorithms)

CLRS, Ch. 3 (Growth of Functions)

- Growth of Functions
- Merge sort is $\Theta(n \log n)$ worst-case, insertion sort is $\Theta(n^2)$.
- Usually we only want approximate running time (exact time may be intractable and not even useful).
- When we want only the order of growth, it is asymptotic efficiency we study - running time increase vs. size of input in the limit.
- Typically we take $n \in \mathbb{N}$ (natural numbers) but sometimes generalize to $\mathbb{R}$ or restrict to a subset
- The notation herein applies more generally to functions.
- $T(n)$ = Worst-case running time function ($T(n) = \Theta(n^2)$ for insertion sort)

"big theta" p44, p45

- $\Theta(g(n)) = \{ f(n)$ : there exist constants $c_1, c_2$, and $n_0 > 0$ such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0 \}$ - Typ. $n_0$ integer
  - Thus $\Theta$ bounds function to within constant factors (and it is a set)
  - To be pedantic, we'd write $f(n) \in \Theta(g(n))$, but really we write $f(n) = \Theta(g(n))$ most of the time
  - We say that $g(n)$ is an asymptotically tight bound for $f(n)$.
  - Note that all $f(n)$ must be asymptotically nonnegative ($\ge 0$ for large $n$), and so must $g(n)$ (or else $\Theta$ is empty).

Typ. $F: \mathbb{N} \to \mathbb{R}$, $f \ge 0$ for large $n$

This convention is used elsewhere

- Example: $\frac{1}{2}n^2 - 3n = \Theta(n^2)$ but how?
  - Need $c_1, c_2, n_0$ such that $c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2 \quad \forall n \ge n_0$
    $= c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2$
    - LHS is true for any $c_1 \le 1/14$ if $n \ge 7$.
      RHS " " $c_2 \ge 1/2$ if $n \ge 1$.
      Thus one choice is: $c_1 = 1/14$, $c_2 = 1/2$, $n_0 = 7$.
- More generally, if $f(n) = an^2 + bn + c$ ($a > 0$), let $c_1 = \frac{a}{4}$, $c_2 = \frac{7a}{4}$, $n_0 = 2\max\left(\frac{|b|}{a}, \sqrt{|c|/a}\right)$ for $g(n) = n^2$.

Not all literature uses $O/\Theta$ this way. Sometimes they use $O$ for both bounds.

- $\Theta$ bounds above and below. If only an asymptotic upper bound, use $O$. (big-oh): $\mapsto c \in \mathbb{R}, n_0 \in \mathbb{N}$
  $O(g(n)) = \{ f(n)$ : there exist constants $c, n_0 > 0$ s.t. $0 \le f(n) \le g(n) \ \forall n \ge n_0 \}$.
  (Note that $\Theta(g(n)) \subseteq O(g(n))$ then.)
  - Curiously: $an + b = O(n^2)$ for $a > 0$. It doesn't say how tight the upper bound is.

Also, $f = \Omega(g) \Leftrightarrow g = O(f)$

CLRS, Ch. 3 (cont.)

- $\Omega$ (big-omega) provides an asymptotic lower bound:
  $\Omega(g(n)) = \{ f(n)$ : there exist $c, n_0 > 0$ s.t. $0 \le cg(n) \le f(n)$ $\forall n \ge n_0 \}$
- Thus (theorem 3.1): $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n))$ & $f(n) = \Omega(g(n))$
  - Usually we use this to prove $\Theta$ given $O$ & $\Omega$, not the reverse
- Insertion sort has running time of both $\Omega(n)$ & $O(n^2)$, and asymptotically, these bounds are as tight as possible.
  - Worst-case running time is $\Omega(n^2)$ - for reverse-sorted input.
- What if we see something like: $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$?
  - Loosely: then $\Theta(n)$ is just some function $f(n)$ in the set $\Theta(n)$, e.g. $f(n) = 3n + 1$. We use this when the exact function isn't really relevant.
- Or: $2n^2 + \Theta(n) = \Theta(n^2)$ - then, loosely, for any specific choice in the $\Theta$ on the LHS, the $\Theta$ on the RHS can be chosen to make the equality hold.
- o-notation (little-oh) is for bounds that aren't asymptotically tight.

Also, $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$

  $o(g(n)) = \{ f(n)$ : for any constant $c > 0$, there exists constant $n_0 > 0$ s.t. $0 \le f(n) < cg(n)$ $\forall n \ge n_0 \}$
  - N.B. for $O(\ldots)$, bound holds for some $c$ but for $o(\ldots)$ it is for all $c$.
  - $2n = o(n^2)$ but $2n^2 \neq o(n^2)$
- $\omega$-notation (little-omega) is similar counterpart for $\Omega$.

$\lim_{n \to \infty} \frac{f(n)}{g(n)} \to \infty$

  $f(n) \in \omega(g(n)) \Leftrightarrow g(n) \in o(f(n))$ or, more explicitly
  $\omega(g(n)) = \{ f(n)$ : for any $c > 0$, there exists $n_0 > 0$ s.t. $0 \le cg(n) < f(n)$ $\forall n \ge n_0 \}$
  - $n^2/2 = \omega(n)$, but $n^2/2 \neq \omega(n^2)$
- $o$: $f(n)$ is insignificant relative to $g(n)$ as $n \to \infty$
- $\omega$: " " arbitrarily large " " " "
- See p51 for some relational properties - transitivity, reflexivity, symmetry, transpose symmetry
- $f(n)$ is asymptotically smaller than $g(n)$ if $f(n) = o(g(n))$; likewise, asymptotically larger for $\omega$ rather than $o$.

Chris Hodapp
chodapp3@gatech.edu

2017-01-16d, CS6505 (Computability, Complexity, & Algorithms)

CLRS, Ch. 3 (cont.)

- We can treat $O$ like $\leq$, $\Omega$ like $\geq$, $\Theta$ like $=$, $o$ like $<$, $\omega$ like $>$, e.g. $f(n) = O(g(n)) \rightarrow a \leq b$ - except that not all functions are asymptotically comparable.
- Monotonicity:
  - Monotonically increasing: $m \leq n \Rightarrow f(m) \leq f(n)$
  - Monotonically decreasing: $m \leq n \Rightarrow f(m) \geq f(n)$
  - Strictly increasing: $m < n \Rightarrow f(m) > f(n)$
  - Strictly decreasing: $m < n \Rightarrow f(m) < f(n)$
- $\lceil x \rceil$, $\lfloor x \rfloor$ - ceil (greatest integer $\geq x$), floor (least integer $\leq x$)
  - N.B. Both are monotonically increasing
- $a \bmod n = a - n\lfloor a/n \rfloor \rightarrow 0 \leq a \bmod n < n$
- Polynomial $a_0 n^0 + a_1 n^1 + \dots a_d n^d$: asymptotically positive iff $a_d > 0$ then $p(n) = \Theta(n^d)$; monotonically increasing
  - $f(n)$ is polynomially bounded if $f(n) = O(n^k)$ for constant $k$
- Since $\lim_{n \to \infty} n^b / a^n = 0$, $n^b = o(a^n)$. That is: Exponential (base $> 1$) grows faster than any polynomial.
- Logarithms: $\lg = \log_2$, $\ln = \log_e$, $\lg^k n = (\lg n)^k$, $\lg \lg n = \lg(\lg n)$
  - Polylogarithmically bounded: $f(n) = O(\lg^k n)$
    but $\lg^b n = o(n^a)$ - positive polynomial grows faster than any polylogarithmic function
- Factorials
  - Weak upper bound: $n! \leq n^n$ (factorial has $n$ terms each $\leq n$)
  - Better bound: Stirling's approximation, $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + \Theta\left(\frac{1}{n}\right)\right)$
- Notation $f^{(i)}(n) = f(n)$ applied iteratively $i$ times to $n$, where $f: \mathbb{R} \rightarrow \mathbb{R}$
  e.g. if $f(n) = 2n$, then $f^{(i)}(n) = 2^i n$
- Fibonacci functions & iterated logarithm: See pp58-59.

---

Pryby, Asymptotic Notation mini-lesson

- Goal: Compare two functions, find which is asymptotically larger, give a tight bound on asymptotic growth.
- Example: $\frac{1}{2}n^2 - 3n = O(n^2) \rightarrow 0 < \frac{1}{2}n^2 - 3n \leq \frac{1}{2}n^2$ for all $n \geq 7$
  ($f(n) > 0$ for $n \geq 6$; $c = \frac{1}{2}$, $g(n) = n^2$, $n_0 = 7$)

Pryby asymptotic notation (mini-lesson) cont.

- Other $n_0$ & $c$ could work too.
- $7n = O(n^2)$ - $0 < 7n \leq 7n^2 \quad \forall n \geq 1$ ($n_0$; $c$; $g(x)$)
  - Also $O(n^3)$ and so on. Its only an upper bound.
- $f(n) = 10n^3 + 100n^2 + 1000n + 10000 = O(n^3)$
  What can we multiply $n^3$ by to make it larger than all of $f(n)$?
  If $n=10$ then $f(n) = 4 \cdot 10^4$ and $g(n) = 10^3$.
  $40\,g(n) = 40 \cdot 10^3 = 4 \times 10^4$, then $g(n) = 10n^3 + 10n^3 + 10n^3 + 10n^3$ too. Each term of $g(n)$ then is greater than or equal to corresponding terms of $f(n)$ - $10n^3, 100n^2, 1000n, 10000$ - for $n \geq 10$.
  Thus, $n_0 = 10$ $c = 40$ (one answer)
- If $f = \Theta(g)$, they have "asymptotically equal growth"
- If $f(n)$ = constant, we say $f = \Theta(1)$. We're not concerned with anything that "grows" slower than $\Theta(1)$, but it can occur, e.g. $\Theta(\frac{1}{n})$ decay.
- For constant $c$, since $\log_b(cn) = \log_b(n) + \log_b c$ (Also true.), $\log_b(cn) = \Theta(\log_b n)$
  $\log_b(n^k) = k \log_b n = \Theta(\log_b n)$ again, & likewise for any polynomial.
- Basically, the log base is irrelevant (it's just a constant factor)
- Consider $n^2 + 2n \log n - 2(n^0) = \Theta(n^2)$
  1. Find largest power of $n$.
  2. If tie, use power of $\log n$ to break tie
- Or: $n^2 \log^2 n + 300\,n^2 - n^2 \log^2 n = \Theta(n^2 \log^2 n)$
  - $n^2$ tied, so take $n^2 \log^2 n$ term
- Or: $n^{2.000001} + n^2 \log^{1000000} n = \Theta(n^{2.000001})$
- We ignore iterated logs here
- To find dominant term in exponentials:
  1. Find exponent with largest base.
  2. Break tie w/ polynomial factor.

  - $3^n n^2 + 2^n n^3 = \Theta(3^n n^2)$
  - $3^n n^2 + 3^n n^3 = \Theta(3^n n^3)$

Note that higher exponents don't matter:
$2^{2n} + 3^{n+3} = 4^n + 27 \cdot 3^n = \Theta(4^n)$

Chris Hodapp
chodapp3@gatech.edu

2017-01-16, CS6505 (Computability, Complexity, & Algorithms)

Prybg, asymptotic notation mini-lesson (cont.)

- Factorials are common, especially with permutations & combinations in algorithm.
- $\log(n!) = \Theta(n \log n)$ - Stirling's approx. easily shows this.

---

Prybg induction mini-lesson (cont.)

- Mathematical induction - $S_0 \Rightarrow S_1 \Rightarrow S_2 \ldots$
  - Suppose $c$ is a positive integer.
  - If $(x-1)$ is a positive integer, then so is $x$.
  - Then $c+1$ is a positive integer. So is $c+2$, $c+3$, $c+4 \ldots$
  $\Rightarrow$ $c+n$ is positive integer $\forall n > 0$.
- Example: Sum of first $n$ positive integers is $\frac{n(n+1)}{2}$ for all $n \geq 0$.
  1. Sum of first 0 is 0 (no numbers), which is $\frac{0(0+1)}{2} = \frac{n(n+1)}{2}$ - Base Case
  2. If sum of first $(n-1)$ pos. ints is $\frac{(n-1)n}{2}$, then the sum of first $n$ pos. ints. is $\frac{n(n+1)}{2}$. To show: $\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$ - Inductive Hypothesis

$$\sum_{i=1}^{n} i = n + \sum_{i=1}^{n-1} i = n + \frac{(n-1)n}{2} = \frac{(n-1)n + 2n}{2} = \frac{n^2 - n + 2n}{2} = \frac{n^2+n}{2} = \frac{n(n+1)}{2}$$ - thus, proven.

- "Strong" induction: $S_0 \wedge S_1 \wedge \ldots \wedge S_{n-1} \Rightarrow S_n$ (all cases up to $n-1$, imply $S_n$)
  - "Standard" induction is just $S_{n-1} \Rightarrow S_n$
  - They're actually equivalent. However, strong induction can be easier to prove if $S_n$ doesn't have an easy proof from just $S_{n-1}$.
- Consider: Every integer $n \geq 2$ is a product of prime numbers.
  - Base case: 2 is prime, so trivially a product of itself
  - Inductive step: Given $n > 2$, suppose every integer from 2 to $n-1$ (inclusive) is a product of primes.
    - $n$ is prime: $n$ trivially is product of primes (as itself)
    - $n$ is not prime: $n = a \cdot b$, where $a, b$ are ints between 2 & $n-1$. But with strong induction, $a$ & $b$ must both be products of primes as they're between 2 & $n-1$, so $a = p_1 p_2 \ldots p_k$, $b = q_1 q_2 \ldots q_\ell$. Thus, $n = p_1 p_2 \ldots p_k q_1 q_2 \ldots q_\ell$ - again product of primes!

Pryby, Induction mini-lesson (cont.)

- Another: For all $n \geq 12$, we may make $n$ by nonnegative multiples of 3 & 7.
  - Base case(s): $12 = 4(3)$.
    $13 = 2(3) + 7$
    $14 = 2(7)$
  - We can easily get to 15 by adding another 3 to 12, and likewise $13 \to 16$ and $14 \to 17$ in the same manner, and from there to any other number.

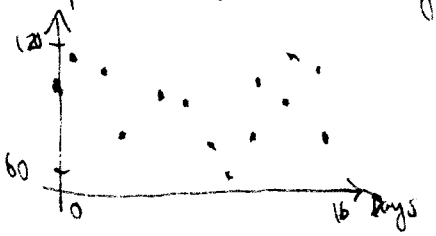
We may have multiple base cases. ~~Note that from 12→13 we subtract two 3s and add a 7 (thus +1), and from 12→14 we replace 2 3s with one 7~~

- All integers are a multiple of 3, multiple of 3 + 1, or multiple of 3 + 2.
- Inductive step, more formally:
  If claim is true for $(n-3)$, $n \geq 15$, then claim is true for $n$.

2017-01-18

Pryby, Divide & Conquer (part 1)

- Recall steps: Divide
  Conquer
  Combine
- Example: Maximum Subarray Problem




- maybe that's stock price on each day
- Want to maximize profit - $P_{sale} - P_{purchase}$
  - but simply picking the lowest ~~point~~ point to purchase, and highest point after, doesn't work. Likewise, neither does picking highest price to sell, then looking backwards to lowest point to buy.
- Brute force: Consider all $(i, j)$, $i < j$, for $n$ days, which is $\binom{n}{2} = \frac{1}{2}n(n-1) = \Theta(n^2)$
- But suppose we compute a table of daily changes:

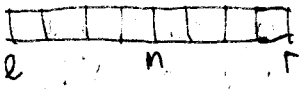
| 13 | -3 | -25 | 20 | -3 | -16 | ... |
|---|---|---|---|---|---|---|
| day 0→1 | 1→2 | 2→3 | 3→4 | 4→5 | | |

Chris Hodapp
chodapp3@gatech.edu

2017-01-18, CS6505 (Computability, Complexity, & Algorithms)

Dyby. Divide & Conquer, part 1 (cont.)

- But to use this, we still must iterate through $\binom{n}{2}$ pairs and compute the sum of each subarray → $O(n^3)$
  - Though judicious adding gets that to $O(n^2)$
- We can get it to $o(n^2)$ though (that's little-oh) (what?)
- We want the subarray $A[l:r]$ which has the largest sum of its elements. If we divide into $A[l:m]$ and $A[m:r]$...




Max subarray is either inside $A[l:m]$, inside $A[m:r]$, or it crosses m. The first two cases can be handled recursively. The last case has two sub-cases:

- Max subarray of form $A[i:m]$ - "anchored" at right
- Max subarray of form $A[m:j]$ " " left

then concatenate them!


→ Exercise for us

This combination is: 1) Computable in $\Theta(r-l)$ time (but why?) 2) Guaranteed to be the max subarray crossing m

- Assuming that's true, how do we turn this to a full algorithm?

```
def max_subarray(A, l, r):
    if r - l == 1:
        return (l, r, A[l])
    m = (l + r) / 2                          (Floor division)
    (l1, r1, s1) = max_subarray(A, l, m)
    (l2, r2, s2) = max_subarray(A, m, r)
    (l3, r3, s3) = max_crossing_subarray(A, l, m, r)
    return (li, ri, si) s.t. si = max(s1, s2, s3)
```

Base

Divide & Conquer

Combine?

Returns:
$(l', r', s)$ s.t. $A[l':r']$ is max subarray of $A[l:r]$ & elements add up to s

- Not described here (just assumed)

2016-01-18

Prgby, Divide & Conquer part 1 (cont.)

- Correctness: Since it's recursive, use induction.
  - Base case: Single-element array is its own max, trivially
  - Inductive step (use strong induction):
    Suppose max_subarray(A, l, r) returns max subarray for $r-l \leq k$, and suppose $r-l = k+1$.
    A[l:m] & A[m:r] are subarrays of length $\leq k$. (Easily shown.)
    (That's for $k \geq 2$. $k=1$ is base case.) Thus, the first two max_subarray calls are the max subarrays for A[l:m] & A[m:r]. Assume max_crossing_subarray gives max subarray not contained in A[l:m] & A[m:r]
    All subarrays are in one of those 3 places, the max of the max of these 3 must give us the overall max. QED.
- So (the whole point) does this actually give a performance improvement?

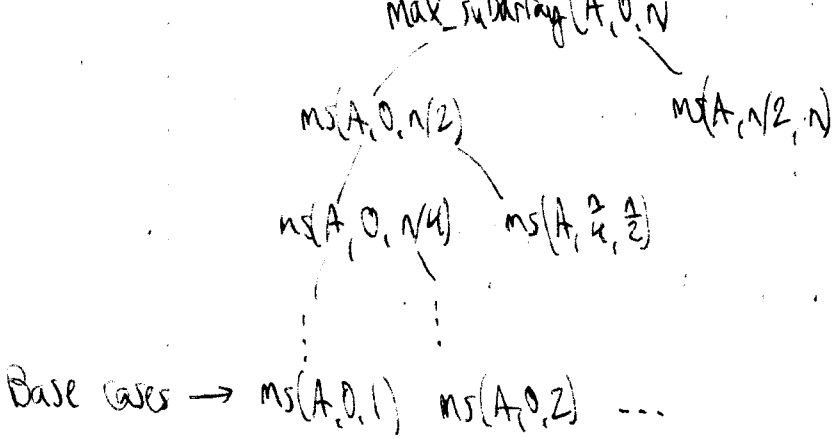
$$T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \quad \text{- base case} \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n>1 \quad \text{- recursing} \end{cases}$$

$n = r-l$

↑ Recursion ↑ max_crossing_subarray (so ignore $\Theta(1)$ stuff)

This is asymptotically the same as merge sort, thus, $\Theta(n \log n)$.

- Memory usage
  - A little trickier, but mostly linear like mergesort.
  - It's $\Theta(1)$ per call that must persist during recursion




but we only need the memory while the recursive call is still active, thus, tree depth tells how many units of memory.

$\Theta(\log n)$ levels → $\Theta(\log n)$ memory!

Chris Hodapp
chodapp3@gatech.edu

2016-01-18c, CS6505 (Computability, Complexity, & Algorithms)

Prydz; Divide & Conquer, part 1 (cont.)

- Another algorithm: Integer multiplication
  - We assume prior that +, −, *, /, %, <<, >> are $\Theta(1)$ but this isn't quite right.
  - If we consider the number of bits, for instance, addition is more like $\Theta(\log a + \log b)$; for bitshift, $\Theta(n)$ for n bit shift
  - What about multiplication?
    - Consider the algorithm you learned in grade school for multiplying multi-digit numbers by hand...

```
  123
  456
 ---
  738      - 3 single-digit multiplies, additions
 6150      - Shift by 1 (i.e. x10), 3 single-digit x/+
49200      - Shift by 2 (x100),
56088      - Add 3 numbers
```

    - So, something like $\Theta(mn)$ assuming m & n digits and multiplication table available for all single-digit multiplies
    - Thus, $\Theta(n^2)$ for same number of digits in operands. This was thought to be the best possible asymptotic behavior through the 1950s....
      Until Kolmogorov tried to prove it and his student Karatsuba found a way that was $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$

It's not specific to base 2.

  - Karatsuba's method:
    x, y product can be achieved by 3 multiplications of smaller numbers
    x & y have n bits, and we can do this with 3 multiplications half-size as original.

$$x = \underbrace{x_1 \cdot 2^{n/2}}_{\text{Quotient}} + \underbrace{x_0}_{\text{Remainder}}, \quad y = \underbrace{y_1 \cdot 2^{n/2} + y_0}_{\text{Likewise}}$$

    - Quotient is just a bitshift to the right by n/2 $(\Theta(n))$

$$xy = (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) = \underbrace{x_1 y_1 2^n}_{\text{multiply \& bitshift}} + \underbrace{(x_1 y_0 + x_0 y_1) 2^{n/2}}_{\text{2 multiplies, add, \& bitshift}} + \underbrace{x_0 y_0}_{\text{multiply}}$$

2016-01-18

Prydog, Divide & Conquer, part 1 (cont.)

- Note that after ~~multiply~~ bitshifts, $x_1$ & $y_1$ are $\leq n/2$ bits.
- But this is still 4 multiplications, so (after some work) still $\Theta(n^2)$.
- Let $z_2 = x_1 y_1$, $z_1 = x_1 y_0 + x_0 y_1$, $z_0 = x_0 y_0$
  but then $z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0$
  → Fewer multiplies (at cost of some more linear operations).

- So say... $T(n) = \begin{cases} 3T(n/2) + \Theta(n) & n \geq 4 \\ \Theta(1) & n < 4 \end{cases}$

  $\Theta(1)$: ↑ Basecase
  $3T(n/2)$: 3 recursive multiplies (half-size)
  $\Theta(n)$: Linear ops (shifts, adds)

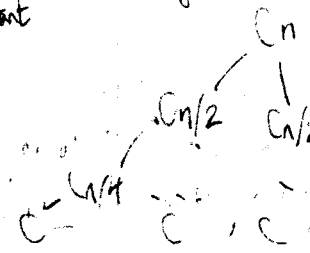
→ $T(n) = \Theta(n^{\log_2 3})$

Left as an exercise: Correctness proof (use induction on # of bits)

```
def karatsuba(x,y):                                   T(n)
    n = max(size(x), size(y))
    if n < 4:
        return (grade school multiplication)          Θ(1)
    x1 = x >> (n >> 1)
    x0 = x % (2 << (n >> 1))
    y1 = y >> (n >> 1)                                Θ(n)
    y0 = y % (2 << (n >> 1))
    z2 = karatsuba(x1, y1)                            T(n/2)
    z0 = karatsuba(x0, y0)                            T(n/2)
    a, b = x0+x1, y0+y1                               Θ(n)
    z1 = karatsuba(a,b) - z0 - z2                     T(n/2)
    p1 = z2 << n                                      Θ(n)
    p2 = z1 << (n >> 1)                               Θ(n)
    return p2 + p1 + z0
```

Why $n^{\log_2 3}$?

C = constant




1 term
3 terms
$3^d$ terms

$T(n) = 3T(n/2) + \Theta(n)$

→ Level $d$ has $3^d$ terms of $\frac{Cn}{2^d}$,

so $Cn\left(\frac{3}{2}\right)^d$, $d = \log_2 n$

So $\sum_{d=0}^{\log_2 n} Cn\left(\frac{3}{2}\right)^d = Cn \sum \left(\frac{3}{2}\right)^d$

$= Cn\frac{\left(\frac{3}{2}\right)^{\log_2 n + 1} - 1}{\frac{3}{2} - 1}$

$= \frac{C}{2} n \left(\left(\frac{3}{2}\right)^{\log_2 n + 1} - 1\right)$

$= \frac{3}{2} Cn \, 3^{\log_2 n}$

but $3 = 2^{\log_2 3}$

so $= \frac{3}{2} Cn \left(2^{\log_2 3}\right)^{\log_2 n}$

$= \frac{3}{2} Cn \left(2^{\log_2 n}\right)^{\log_2 3}$

$= \left(\frac{3n}{2}\right)^{\log_2 3}$

close enough

Chris Hodapp
chodapp3@gatech.edu

2016-01-18 d, CS6505 (Computability, Complexity, & Algorithms)

Prybyl, Divide & Conquer part 1 (cont.)

- Left as exercise: Compute auxiliary space required
- Next: Master theorem, to avoid all this $n^{\log_2 3}$ derivation work

---

Part 2

2016-01-19

- Master Theorem (see also CLRS ch. 4)
  - Let $a \geq 1$, $b \geq 1$ be constants $f(n)$ an eventually nonnegative function, and $T(n)$ be defined as $\begin{cases} \Theta(1), & n \leq n_0 \\ aT(n/b) + f(n), & n > n_0 \end{cases}$
  - Then:
    - If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
    - If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$
    - If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$ and $af(n/b) \leq cf(n)$ for some $c < 1$ & large enough $n$, then $T(n) = \Theta(f(n))$. ← regularity condition
      - Note that regularity condition isn't always necessary, but $f(n) = \Omega(\ldots)$ and regularity are always sufficient for $T(n) = \Theta(f(n))$.
  - Intuitively:
    - If $F$ is asymptotically smaller than $n^{\log_b a}$, $T(n)$ still grows like $n^{\log_b a}$. If $F$ is asymptotically equivalent to $n^{\log_b a}$, $T(n)$ has the extra $\log n$ factor. If $f(n)$ is asymptotically larger, $T(n)$ should grow like $F(n)$. (With technicalities.)
    - But, $f(n)$ must be either polynomially smaller or larger, hence the $-\varepsilon$, $+\varepsilon$, equivalent to $1/n^{\varepsilon}$ or $n^{\varepsilon}$ factor on $\log_b a$. If it were a factor of $1/\log n$ instead (i.e. $f(n) = O(n^{\log_b a}/(\log n))$) then perhaps $T(n)$ does grow faster than $f(n)$ but we can't use this theorem to show it.

2017-01-18

Pryby, Divide & Conquer, part 2 (cont.)

- Examples:
  - We already did Karatsuba's algorithm, $T(n) = \begin{cases} \Theta(1), & n \le 3 \\ 3T(n/2) + Cn, & n > 3 \end{cases}$
  - Master Theorem now applies: $a=3$, $b=2$, $f(n) = Cn$, and look at $f(n)$:
    $Cn = O(n^{\log_2 3})$ already, but if we say $\varepsilon = 0.1$,
    $Cn = O(n^{\log_2 3 - \varepsilon}) \approx O(n^{1.485})$ which is true since $Cn = Cn^1$
    → 1st part holds, $T(n) = \Theta(n^{\log_2 3})$
  - Merge sort & max subarray, $T(n) = \begin{cases} \Theta(1), & n \le 1 \\ 2T(n/2) + Cn, & n > 1 \end{cases}$
    $a=2, b=2, f(n) = Cn$, so $\Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$, and so trivially $f(n) = \Theta(n)$. Thus, $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$.
  - Consider: $T(n) = \begin{cases} \Theta(1), & n \le 3 \\ 3T(n/4) + n \log n, & n > 3 \end{cases}$
    $a=3$, $b=4$, $f(n) = n \log n$; how does $f(n)$ relate to $O(n^{\log_b a - \varepsilon})$, $\Theta(n^{\log_b a})$, $\Omega(n^{\log_b a + \varepsilon})$?
    $n^{\log_b a} = n^{\log_4 3} \approx n^{0.8}$, so if we pick $\varepsilon = 0.1$, $n^{\log_4 3 + \varepsilon} < n^1$ still,
    thus $n \log n = \Omega(n^{\approx 0.9})$. Next, $af(n/b) = 3\frac{n}{4} \log \frac{n}{4}$,
    so... $af(n/b) \le cf(n)$
    $3\frac{n}{4} \log \frac{n}{4} \le Cn \log n$ and since $\log \frac{n}{4} < \log n$ we can do $c = \frac{3}{4}$
    $\frac{3}{4} n \log \frac{n}{4} \le \frac{3}{4} n \log n$ - that is then true.
  - Thus, 3rd case holds: $T(n) = \Theta(n \log n)$.

Not really needed for homework or exam

- Proof of the Master Theorem

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ aT(n/b) + f(n), & n > 1 \end{cases}$$

1: Assume $n = b^k$ for $k \ge 0$

2: Extend to all natural numbers (see CLRS, it's not given here)

- $n = b^i$ for $i \ge 1$.

Claim 1: Let $a \ge 1$, $b > 1$ be const, $f(n)$ nonneg. defined on $b^i$, $i \ge 1$, & define $T(n)$ as above, for $n = b^i$. Then,

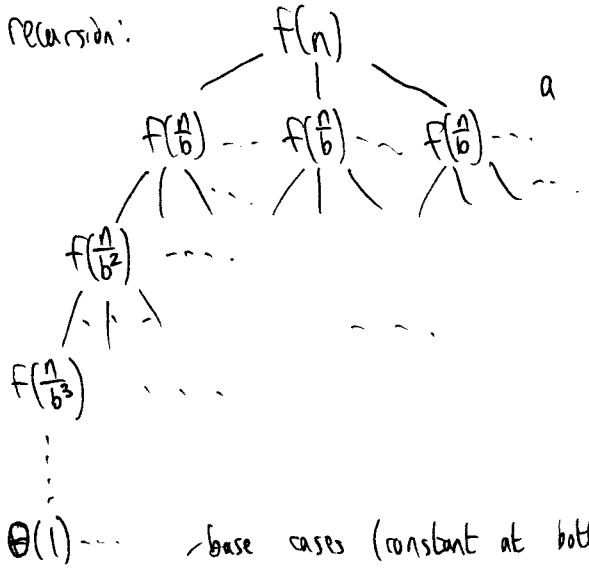
$$T(n) = \Theta(n^{\log_b a}) + \sum_{d=0}^{\log_b n - 1} a^d f\left(\frac{n}{b^d}\right)$$

Chris Hodapp
chodapp3@gatech.edu

2017-01-19(b), CS6505 (Computability, Complexity, & Algorithms)

Prybys, Divide & Conquer, part 2 (cont.)

- Consider recursion:




N.B. a need not be integer. Our tree is just a heuristic. Also, T(n) can be other things than time.

At level $d$, $a^d$ instances of $f(n/b^d)$ exist. Last level $d = \log_b n$.
Bottom has $a^d \Theta(1)$, but $a^d = a^{\log_b n} = b^{\log_b a \cdot \log_b n} = n^{\log_b a}$
$\rightarrow \Theta(n^{\log_b a})$ just at bottom.
Other levels are: $\sum_{d=0}^{\log_b n - 1} a^d f(n/b^d)$, so $T(n) = \Theta(n^{\log_b a}) + \sum a^d f(n/b^d)$
(first sum: Division & Combination steps; $\Theta(n^{\log_b a})$ + ...: Conquer steps)

- So really, this is sort of asking whether the 'divide/combine' parts are more or less than the 'conquer' parts.
- Thus, 1st claim is shown

Claim 2: $a \geq 1$, $b > 1$, $n = b^k$, $k \geq 0$, $f(n)$ nonneg., $g(n) = \sum_{d=0}^{\log_b n - 1} a^d f(n/b^d)$
Then...
i - If $f(n) = O(n^{\log_b a - \varepsilon})$, $\varepsilon > 0$, then $g = O(n^{\log_b a})$
ii - If $f(n) = \Theta(n^{\log_b a})$, then $g(n) = \Theta(n^{\log_b a} \log n)$
Regularity condition → iii - If $af(n/b) \leq cf(n)$ for some $c < 1$ & large enough $n$, then $g(n) = \Theta(f(n))$.

(i) If $f(n) = O(n^{\log_b a - \varepsilon})$ then $f(n/b^d) = O\left(\left(\frac{n}{b^d}\right)^{\log_b a - \varepsilon}\right)$

$$\Rightarrow g(n) = O\left(\sum_d a^d \left(\frac{n}{b^d}\right)^{\log_b a - \varepsilon}\right) = \cdots = O\left(n^{\log_b a - \varepsilon} \underbrace{\left(\frac{b^{\varepsilon \log_b n} - 1}{b^\varepsilon - 1}\right)}_{= \Theta(n^3)}\right) = O(n^{\log_b a})$$

($b^{\varepsilon \log_b n} = n^3$; cancels $n^{-\varepsilon}$ above)

Q. E.D. (for i)

For more practice: Look at top order for Divide & Conquer examples and try to solve their asymptotic runtimes

Pf by Divide & Conquer, part 2 (cont.)

(ii) $f(n) = \Theta(n^{\log_b a})$, $F(n/b^d) = \Theta\left(\left(\frac{n}{b^d}\right)^{\log_b a}\right) \Rightarrow g(n) = \Theta\left(\sum_d a^d \frac{n^{\log_b a}}{b^{d \log_b a}}\right)$

$= \overset{\text{algebra}}{\cdots} = \Theta\left(n^{\log_b a} \sum_{d=0}^{\log_b n - 1} 1\right) = \Theta(n^{\log_b a} \cdot \log_b n)$ - QED (case ii).

(iii) $af(n/b) \le cf(n)$ for $n \ge n_0$, $c < 1$

$f(n/b) \le \frac{c}{a} f(n)$ - replace $n$ with $n/b$ (is that allowed?)

$f(n/b^2) \le \frac{c}{a} f(n/b) \le \frac{c^2}{a^2} f(n)$ by first inequality

then same trend follows for $f(n/b^3)$, $f(n/b^4)$, etc. (Use induction)

$\rightarrow f(n/b^d) \le \left(\frac{c}{a}\right)^d f(n)$

$g(n) \le \sum_d a^d \left(\frac{c}{a}\right)^d f(n) = \sum_{d=0}^{\log_b n - 1} c^d f(n)$ (assuming big enough $n$)

$= f(n) \sum \cdots c^d \le f(n) \sum_{d=0}^{\infty} c^d$ and since $c < 1$ this converges, as an infinite geometric sum

- If we take this to infinite sum, that must exceed the finite sum $\sum_{d=0}^{\log_b n - 1} \cdots$ of above

So... If $g(n) \le f(n)\left(\frac{1}{1-c}\right)$, $g(n) = O(f(n))$. (That's big-oh).

But $g(n) \ge \sum_{d=0}^{0} a^d f(n/b^d)$ - note sum over just 0

- since $g(n) \ge a^0 f(n) = f(n) \rightarrow g(n) = \Omega(f(n))$.

$g = O(f(n))$ & $g = \Omega(f(n)) \Leftrightarrow \boxed{g = \Theta(f(n))}$ as claimed.

- Overall proof:

i) $T(n) = \Theta(n^{\log_b a}) + O(n^{\log_b a}) = \Theta(n^{\log_b a})$

ii) $T(n) = \Theta(n^{\log_b a}) + \underbrace{\Theta(n^{\log_b a} \log n)}_{\text{Dominates other}} = \Theta(n^{\log_b a} \log n)$

iii) $T(n) = \Theta(n^{\log_b a}) + \Theta(f(n))$. We didn't say $f(n) = \Omega(n^{\log_b a + \epsilon})$ because it turns out the regularity condition implies it.

so $T(n) = \Theta(f(n))$

but $-\epsilon$ is still needed in case i.

Chris Hodapp
chodapp3@gatech.edu

2017-01-19c, CS6505 (Computability, Complexity, & Algorithms)

CLRS, Ch. 4 (Divide & Conquer)

- Three methods to solve recurrences:
  - Substitution method (basically, guess and then check with induction)
  - Recursion-tree method
  - Master method
- For recurrences like $T(n) \leq 2T(n/2) + \Theta(n)$ (note $\leq$, not $=$), we'd get only an upper bound, thus $O$, not $\Theta$. (Likewise, if $T(n) \leq \dots$, lower bound, thus $\Omega$)
- We tend to ignore boundary conditions here, e.g. that merge-sort really has $\lfloor n/2 \rfloor$ & $\lceil n/2 \rceil$ size subproblems; they usually amount only to constant-factor changes
- Matrix multiply:

  Simple way: $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$ for square matrices $(n \times n)$ A & B

  - So each $c_{ij}$ is $n$ multiplies, $n$ adds, but $C = AB$ has $n \times n$ elements, so product is $\Theta(n^3)$.
  - Strassen's algorithm runs in $\Theta(n^{\log 7}) \approx \Theta(n^{2.81})$ &c
  - Suppose $n$ is a power of 2. Divide A & B each into four $n/2 \times n/2$ matrices: $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$, $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$, $C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$

    then: $\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$

    $\rightarrow \begin{cases} C_{11} = A_{11}B_{11} + A_{12}B_{21} \\ C_{12} = A_{11}B_{12} + A_{12}B_{22} \\ C_{21} = A_{21}B_{11} + A_{22}B_{21} \\ C_{22} = A_{21}B_{12} + A_{22}B_{22} \end{cases}$

    → Note: 4 adds, 8 multiplies (matrix multiplies, at least)
  - Suppose we apply this recursively, using scalar multiply on $(x)$ as our base case.

    $T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases}$ ($\Theta(n^2)$ is for matrix additions)

    which is again $\Theta(n^3)$.

$S_1 = B_{12} - B_{22}$ $S_5 = A_{11} + A_{22}$ $S_9 = A_{11} - A_{21}$

$S_2 = A_{11} + A_{12}$ $S_6 = B_{11} + B_{22}$ $S_{10} = B_{11} + B_{12}$

$S_3 = A_{21} + A_{22}$ $S_7 = A_{12} - A_{22}$

$S_4 = B_{21} - B_{11}$ $S_8 = B_{21} + B_{22}$

$P_1 = A_{11} S_5$ $P_5 = S_5 S_6$

$P_2 = S_2 B_{22}$ $P_6 = S_7 S_8$

$P_3 = S_3 B_{11}$ $P_7 = S_9 S_{10}$

$P_4 = A_{22} S_4$

$C_{11} = P_5 + P_4 - P_2 + P_6$

$C_{12} = P_1 + P_2$

$C_{21} = P_3 + P_4$

$C_{22} = P_5 + P_1 - P_3 - P_7$

Details given elsewhere

- Strassen's method does only 7 recursive multiplications, not 8, thus making the recursion tree "less bushy"
  1. Divide $A, B, C$ into $\frac{n}{2} \times \frac{n}{2}$ submatrices ($\Theta(1)$ by index calculation)
  2. Create $S_1, S_2, S_3 \dots S_{10}$, each $\frac{n}{2} \times \frac{n}{2}$ and the sum or difference of all matrices from step 1 ($\Theta(n^2)$ for all 10)
  3. Compute 7 matrix products $P_1, P_2, \dots P_7$, each $\frac{n}{2} \times \frac{n}{2}$.
  4. Compute $C_{11}, C_{12}, C_{21}, C_{22}$ with adding/subtracting $P_1 \dots P_7$ ($\Theta(n^2)$).
- Then... $T(n) = \begin{cases} \Theta(1), & n=1 \\ 7T(n/2) + \Theta(n^2), & n>1 \end{cases} \rightarrow T(n) = \Theta(n^{\log 7})$
- Coppersmith & Winograd's method is $O(n^{2.376})$ - fastest known (best lower bound is $\Omega(n^2)$ due simply to there being $n^2$ elements)
- Section 4.3, substitution method, gives some practical hints worth reviewing

2017-01-23

Prylog mini-lesson, probability

- Sample space $S$, outcome = element of $S$, event = subset of $S$
- Probability: function $\mathbb{P}: 2^S \rightarrow [0,1]$ s.t. $\mathbb{P}(S) = 1$
  - $2^S$ = power set
  - For every finite, countable seq. of events $A_0, A_1, A_2 \dots$ s.t. $A_i \cap A_j = \emptyset$ if $i \neq j$, $\mathbb{P}(\bigcup_i A_i) = \sum_i \mathbb{P}(A_i)$ ] pairwise mutually exclusive (disjoint)
- $A^c = S \setminus A$ = complement of A (parts of sample space excluding A); $\mathbb{P}(A^c) = 1 - \mathbb{P}(A)$ - since $\mathbb{P}(A \vee A^c) = \mathbb{P}(A) + \mathbb{P}(A^c)$, so $\mathbb{P}(A^c) = \mathbb{P}(A \vee A^c) - \mathbb{P}(A)$ but $\mathbb{P}(A \vee A^c) = \mathbb{P}(S) = 1$
- $\mathbb{P}(\emptyset) = 0$ since $\emptyset^c = S$, $S^c = \emptyset$, so $\mathbb{P}(\emptyset) = 1 - \mathbb{P}(S) = 0$
- Monotonicity: If $A \subseteq B$ then $\mathbb{P}(A) \leq \mathbb{P}(B)$ ($B = (B \setminus A) \vee A$, $\mathbb{P}(B) = \mathbb{P}(B \setminus A) + \mathbb{P}(A)$ but since $\mathbb{P}$ is in $[0,1]$ $\mathbb{P}(B)$ is $\geq \mathbb{P}(A)$)
- Union bound: For two events, $\mathbb{P}(A \vee B) \leq \mathbb{P}(A) + \mathbb{P}(B)$, even if events aren't exclusive. (Note $B = (B \setminus A) \vee (B \cap A)$ - B is the parts outside of A and inside A. The rest follows from this.). $\mathbb{P}(A \vee B) = \mathbb{P}(A) + \mathbb{P}(B) - \mathbb{P}(A \cap B)$. - Inclusion/exclusion principle, similar to set theory, also generalizes to more than 2

Chris Hodapp
chodapp3@gatech.edu

2017-01-23(6), CS6505 (Computability, Complexity, & Algorithms)

Pruby mini-lesson, probability (cont.)

- Further: $\mathbb{P}(\bigcup_i A_i) \leq \sum \mathbb{P}(A_i)$
- $\mathbb{P}(\text{elementary event}) = \frac{1}{4}$ for $S = \{HH, HT, TH, TT\}$ - 2 coin flips
  $\mathbb{P}(\#H \geq 1) = \mathbb{P}(\{HH, HT, TH\}) = \mathbb{P}(\{HH\}) + \mathbb{P}(\{HT\}) + \mathbb{P}(\{TH\}) = \frac{3}{4}$
  or: $\mathbb{P}(\#H \geq 1) = \mathbb{P}((\#H = 0)^c) = 1 - \mathbb{P}(\#H = 0) = 1 - \mathbb{P}(\{TT\}) = \frac{3}{4}$
  - Two equivalent ways; one or the other may be easier
- We work mainly in <u>discrete</u> probability spaces, finite or countably infinite. Continous spaces are also a thing, but this often must swap $\sum$ for $\int$.
- Uniform prob. distribution: $\mathbb{P}(\{s\})$ is same $\forall s \in S$
  - This applies only for <u>finite</u> spaces. If infinite, individual events aren't really well-defined, either they're all 0, and violate definition because $\sum / \int = 0$, or they're all $> 0$, and then do the same because $\sum / \int \to \infty$.
  - For uniform: $\mathbb{P}(A) = \frac{|A|}{|S|}$ for all $A \subseteq S$
- If $S$ = coin flipping space with $n$ fair coins...
  $2^n$ outcomes, each with prob. $1/2^n$
  $\mathbb{P}(k \text{ heads}) = \frac{\binom{n}{k}}{2^n}$ (likewise $k$ tails)
  - If you need more specific outcomes than you can use other properties to create them by adding & subtracting w.th various $k$

---

Pruby lecture, Probabilistic Analysis

(To read: CLRS Ch.5)

- The goal here is to explain how to analyze algorithms based on expected probabilities of inputs, and from this determine average-case runtime rather than best/worst case.
- This is then applied to <u>randomized</u> algorithms which rely on the random number generator to perform some task
- Or, it can be used to estimate probability of an error occurring in a randomized algorithm

Prereqs: Random variable
Probability & Independence

2017-01-23

Pruby Lecture, Probab. Lite Analysis
mini-lesson, Conditional Prob. & Independnce

- Conditional probability
  - e.g. Suppose a ball is underneath 2 of 4 shells (→ Red, Green, Blue, White) and we don't know which
  - $S = \{RG, RB, RW, GB, GW, BW\}$, uniform prob. – 6 choices, each $1/6$
    ($RG$ = ball under Red & Green shell)
  - Suppose we learn that no shell is under green. What is $\mathbb{P}(B \text{ has a ball})$?
    - We can eliminate $RG, GB, GW$ from $S$, so $RB, RW, BW$ each have $1/3$ now; $RB$ & $BW$ have blue shell, so it's $2/3$ now.
    - Or: $\mathbb{P}(B \text{ has ball} \mid G \text{ doesn't have ball}) = 2/3$
  - That's conditional probability! $\mathbb{P}(A|B) = A \text{ given } B = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}$, $\mathbb{P}(B) > 0$
  - In some ways, it's like restricting our sample space, then finding probability in that new space
  - $\mathbb{P}(A|B^c) = \mathbb{P}(A \cap B^c) / \mathbb{P}(B^c)$
  - Suppose $\mathbb{P}(A|B) = \mathbb{P}(A)$; then $\mathbb{P}(A) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)} \rightarrow \mathbb{P}(A)\mathbb{P}(B) = \mathbb{P}(A \cap B)$
    (Definition of independence of A & B)

  - $A_0, A_1, A_2, \ldots$ are pairwise independent if $\mathbb{P}(A_i) \cdot \mathbb{P}(A_j) = \mathbb{P}(A_i \cap A_j)$ whenever $i \neq j$.
  - $A_0, A_1, A_2 \ldots$ are mutually independent if for every subset $\{A_{i_1}, A_{i_2}, A_{i_3} \ldots\}$, we have $\mathbb{P}\left(\bigcap_j A_{i_j}\right) = \prod_j \mathbb{P}(A_{i_j})$
  - Consider 2 coins:
    $S = \{HH, HT, TH, TT\}$
    $A_1 = \{HH, HT\}$ – 1st coin heads
    $A_2 = \{HH, TH\}$ – 2nd coin heads
    $A_3 = \{HT, TH\}$ – Opposites
    - Probability of $A_1, A_2, A_3 = \frac{1}{2}$
    - $\mathbb{P}(A_1 \cap A_2) = \frac{1}{4}$
    - $\mathbb{P}(A_1 \cap A_3) = \frac{1}{4}$
    - $\mathbb{P}(A_2 \cap A_3) = \frac{1}{4}$
    → so they're pairwise independent (all these unions equal the product)
    - But $\mathbb{P}(A_1 \cap A_2 \cap A_3) = 0$ – nothing intersects all 3
      - So they are not mutually independent.

Chris Hodapp
chodapp3@gatech.edu

2017-01-23c, CS6505 (Computability, Complexity, & Algorithms)

Pryby mini-lesson, Random Variables

- Random variables & expectations
  - Given space $S$, random variable $X: S \to \mathbb{R}$.
    e.g. Sum of values rolled on 2 dice.
    $S = \{(1,1), (1,2), (1,3) \dots, (2,1), \dots \dots (6,1) \dots (6,6)\}$.
    Fair dice, so $P(\{s\}) = 1/36$.
    $X(s)$ = sum of dice, so $X((1,1)) = 2$.
  - We focus mainly on discrete random variables in this class, but continous random variables also come up.
  - For random var. $X$ and $x \in \mathbb{R}$:
    - Notation "$X = x$" $= \{s \in S : X(s) = x\}$
    - e.g. $X = 3$ in our dice example is $\{(1,2), (2,1)\}$ since no other dice combinations produce 3.
    - $P(X=3) = \sum_{s \in S : X(s)=x} P(\{s\})$
      - Which is easy in a uniform space: just count up the samples for $X(s)=x$ and divide by $|S|$.
    - $P(X=4)$
      $= P(\{(1,3)\}) + P(\{(2,2)\}) + P(\{(3,1)\}) = \frac{3}{36} = \frac{1}{12}$.
    - If we let $x$ vary, $P(X=x)$ is a function of it, then we get $f_X(x) = P(X=x)$ - the PDF (Probability Density Function) of $X$.
  - So, of course, we can define various random variables on space $S$.
  - But if $X, Y$ are random vars on $S$, the <u>joint PDF</u> of $X$ & $Y$ is $f_{X,Y}(x,y) = P(X=x \text{ and } Y=y)$

    $\sum_x P(X=x \text{ and } Y=y_0) = P(Y=y_0)$

    $\sum_y P(X=x_0 \text{ and } Y=y) = P(X=x_0)$

    ] Sometimes useful

  - $P(X=x \mid Y=y) = \dfrac{P(X=x \text{ and } Y=y)}{P(Y=y)}$
    - Can use conditionals on random variables

2017-01-23

Pryby mini-lesson, random variables (cont.)

- $X$ & $Y$ (random vars) are independent if for every $x$ & $y$, $\mathbb{P}(X=x \text{ and } Y=y) = \mathbb{P}(X=x)\cdot\mathbb{P}(Y=y)$.
  (Usually only applied where $X$ & $Y$ actually produce $x$ & $y$).
- Main tool in probabilistic analysis: Expected Value
  - It's our formal tool to compute the average of a random var.
    $\mathbb{E}X = \sum_x x\cdot\mathbb{P}(X=x)$ ; sometimes written $\mu_X$ ($\mu$ = mean)
    - So for the 2 dice example: $\mathbb{E}X = 2\cdot\frac{1}{36} + 3\cdot\frac{2}{36} + \dots + 11\cdot\frac{2}{36} + 12\cdot\frac{1}{36}$
      $= 7$
- Expected value is a linear operator. $\mathbb{E}[aX] = a\mathbb{E}X$
  Scalar multiply & add pass through $\mathbb{E}[X+Y] = \mathbb{E}X + \mathbb{E}Y$
  - Thus, we can redo $\mathbb{E}X$ for dice example as $X = X_1 + X_2$
    so $\mathbb{E}X = \mathbb{E}X_1 + \mathbb{E}X_2 = 2\mathbb{E}X_1$ since ($X_1$: die 1 value, $X_2$: die 2 value)
    $\mathbb{P}(X_1=x) = \mathbb{P}(X_2=x)\ \forall x$.
    so $\mathbb{E}X = 2\sum_{x=1}^{6}\frac{x}{6} = \frac{2}{6}(1+2+3+4+5+6) = \frac{21}{3} = 7$

- If $X$ & $Y$ are independent random vars, $\mathbb{E}[XY] = \mathbb{E}X\,\mathbb{E}Y$
- Law of Total Expectation:
  - Suppose random vars. $X, Y$
  - $\mathbb{E}[X|Y]$ is conditional expectation $= \sum_x x\,\mathbb{P}(X=x|Y=y)$
    (also $\mathbb{E}[X|Y=y]$ - it's a function of $y$ →)
  - $\mathbb{E}_Y[\mathbb{E}[X|Y]] = \mathbb{E}X$ - that's aforementioned law
    ↑ E.V. with respect to $Y$

$$= \mathbb{E}_Y\left[\sum_x x\,\mathbb{P}(X=x|Y=y)\right] = \sum_y\left[\sum_x x\,\mathbb{P}(X=x|Y=y)\right]\mathbb{P}(Y=y) = \sum_x\sum_y (x\dots)$$

$$= \sum_x x\sum_y \mathbb{P}(X=x|Y=y)\,\mathbb{P}(Y=y) = \underbrace{\sum_x x\sum_y \mathbb{P}(X=x \text{ and } Y=y) = \sum_x x\,\mathbb{P}(X=x)}_{\text{Recall last page}}$$

$$= \mathbb{E}X$$

Chris Hodapp
chodapp3@gatech.edu

2017-01-23d, CS6505 (Computability, Complexity, & Algorithms)

Pryby lecture, Probabilistic Analysis

- Motivation: "hiring problem"
  - $c_i$ = cost to interview — assume $c_i < c_h$
  - $c_h$ = cost to hire
- Assume candidate has some linear rating (interviewing returns this)
- Goal: Estimate cost of hiring best candidate

```
def hire_best(A):
    best_q = -1
    for in range(0, n):
        this_q = A[i].interview
        if this_q > best_q:
            A[i].hire()
            best_q = this_q
```

  - A is list of n candidates
  - A[i].interview() - returns qualification (int > 0)
  - A[i].hire() - fires old, hires new
  - That is, just iterate over all candidates, interview each, hire the next-best at each opportunity
- cost(n) = $c_i n + X c_h$ where X is number of hired candidates (or C(n))
- Worst-case is easy: A is in increasing order, then X = n since A[i].hire() is called at every iteration, then $C(n) = \Theta((c_i + c_h)n)$
- Best-case is too: A is in decreasing order, then it's called only at first iteration, then X = 1 & $C(n) = \Theta(c_i n + c_h)$. (Actually, best just needs to be first.)
- But, what is average case?
  - rank(i) = candidate with $i^{th}$ lowest qualification - rank(n) = best, rank(1) = worst
  - Worst = [rank(1), rank(2) ... rank(n)]
  - Best = [rank(n), ... (anything else after)]
  - n! orderings exist, of course. To estimate average case, we assume uniform distribution of those n!, thus any ordering has probability $1/n!$. (Note that we don't require uniform)
  - We want: $\mathbb{E}[C(n)] = \mathbb{E}[c_i n + c_h X] = c_i n + c_h \mathbb{E}X$
  - We'll apply a technique: decomposition into indicator random variables.

$$\mathbb{1}_A = \begin{cases} 1, & A \text{ occurs} \\ 0, & A \text{ doesn't occur} \end{cases}$$

- indicator function

Assume each candidate has distinct qualifications.

- Then note that $X = \sum_{i=0}^{n-1} 1\!\!1_{A_i}$ where $A_i$ = candidate $i$ is hired

so: $\mathbb{E}X = \sum_{i=0}^{n-1} \mathbb{E}(1\!\!1_{A_i}) = \sum_{i=0}^{n-1} \mathbb{P}(A_i)$ - That is, just the sum of the probability that we hire each candidate.

$\mathbb{E}1\!\!1_A = 1 \cdot \mathbb{P}(A) + 0 \cdot \mathbb{P}(A^c) = \mathbb{P}(A)$

- So then, what is $\mathbb{P}(A_i)$?:
  - N.B. It is the probability that $A_i$ is more qualified than $A_0, A_1, A_2 \ldots A_{i-1}$.
- Candidate 0 hired: $\mathbb{P}(A_0) = 1$ - we always hire 1st
  Candidate 1 hired: $\mathbb{P}(A_1) = \frac{1}{2}$ if it is equally likely that $A_1$ is more qualified (which we'd expect with uniform distribution
  Candidate 2 hired: $\mathbb{P}(A_2) = \frac{1}{3}$ for same reason
→ $\frac{1}{i+1}$ of the orderings will have a candidate be the best among the first $(i+1)$ candidates.

Thus $\mathbb{E}X = \sum \frac{1}{i+1} = \Theta(\log n)$ - harmonic sum

→ $\mathbb{E}[C(n)] = \Theta(c_1 n + c_h \log n)$.

- We revisit insertion sort in the homework - note that best-case is $\Theta(n)$ & worst-case is $\Theta(n^2)$.

---

Udacity, Lesson 16 (Verifying Polynomial Identities through Analysis of Min Cut)

- Verifying Polynomial Identities
  - Input: Representations of polynomials $A(x)$ & $B(x)$ of degree $d$.
  - Goal: Decide if $A(x) \equiv B(x)$
  - Very simple method: Pick a random integer between 1 & $100d$, see if $A(x) = B(x)$ there. By Fundamental theorem of algebra, unless $A(x) - B(x) = 0$, it can have only $d$ roots. Chance of picking one is $\leq \frac{d}{100d} = \frac{1}{100}$.
    - If $A(x) \equiv B(x)$ this will always return true. If not, it will sometimes return true.

Chris Hodapp
chodapp3@gatech.edu

2017-01-23e, CS6505 (Computability, Complexity, & Algorithms)

Udacity, Lesson 16 (Randomized Algorithms), cont.

- Discrete Probability spaces
- Repeated trials (back to polynomials):
  - Suppose we iteratively run the last method on more random numbers.
  - Then, for $k$ iterations, chance of a false negative is $\frac{1}{100^k}$
- Monte Carlo & Las Vegas
  - The method above is a "one-sided Monte Carlo algorithm":
    If $A \equiv B$, then $\Pr[\text{polyequal}(A,B,d)] = 1$
    If $A \not\equiv B$, then $\Pr[\text{polyequal}(A,B,d)] \leq 100^{-k}$ ← Mistakes only here
  - Interestingly, if we pick our random samples with replacement, then (ignoring issues of precision) we can reduce that $100^{-k}$ to 0. (Las Vegas Algorithm, uses randomization, but that has an effect only on running time, not correctness.)
- Randomized Quicksort

```
def quicksort(A):
    if len(A) < 2:
        return A
    pivot = choice(A)        - Uniformly random
    AL = [x for x in A if x < pivot]
    AR = [x for x in A if x > pivot]
    return quicksort(AL) + pivot + quicksort(AR)
```

Middle of sorted list

- If pivot is at middle, then this is $\Theta(n \log n)$ because it halves list each time.
- If pivot is uniformly the first or last sorted item, then this is $n^2$ because AL will only reduce A by one, or AR
- So, what is quicksorts <u>average</u> case performance?
- Suppose we quicksort a list w/ elements $a_1 < a_2 < a_3 \ldots < a_n$. Let $E_{ijk}$ be event that $a_i$ is separated from $a_j$ by the $k$th choice of a pivot. Let $X_{ij} = 1$ if algo. compares $a_i$ & $a_j$; else 0.

$E[\dots] = \mathbb{E}\dots$

$Pr(\dots) = \mathbb{P}(\dots)$

2017-01-23

Udacity 16 (Randomized Algorithms), cont.

- Then $X_{ij}$ counts how many comparisons are actually done.
- Claim: $E[X_{ij}] = \frac{2}{j-i+1}$

Proof:

$$E[X_{ij}] = 1 \cdot Pr(X_{ij} = 1) + 0 \cdot Pr(X_{ij} = 0)$$

$$= \sum_k Pr(X_{ij} = 1 \cap E_{ijk})$$

$$= \sum_k Pr(X_{ij} = 1 \mid E_{ijk}) \, Pr(E_{ijk}), \quad \underline{Pr[E_{ijk}] > 0}$$

Thus pivot must occur between $i$ & $j$

- $i$ & $j$ are separated (if at all) by one pivot
- Note that all $E_{ijk}$ are disjoint then. (Law of Total Probability)

- But $a_i$ is only compared to $a_j$ when one is chosen as pivot, so $Pr(X_{ij} = 1 \mid E_{ijk}) = \frac{2}{j-i+1}$

2 For $i$ or $j$, $j-i+1$ for total elements between $i$ & $j$.

So $E[X_{ij}] = \sum_k \frac{2}{j-i+1} Pr(E_{ijk})$ (for $Pr(E_{ijk}) > 0$)

but because some pivot exists this must just be $\sum 1$ once $\frac{2}{j-i+1}$ is factored out, thus $\boxed{\frac{2}{j-i+1}}$

By linearity:

$$\sum_{i<j} E[X_{ij}] = \sum_{i<j} \frac{2}{j-i+1} \leq \sum_{i=1}^{n} \sum_{j=1}^{n} \frac{2}{j} = \Theta(n \log n)$$

note $\leq$

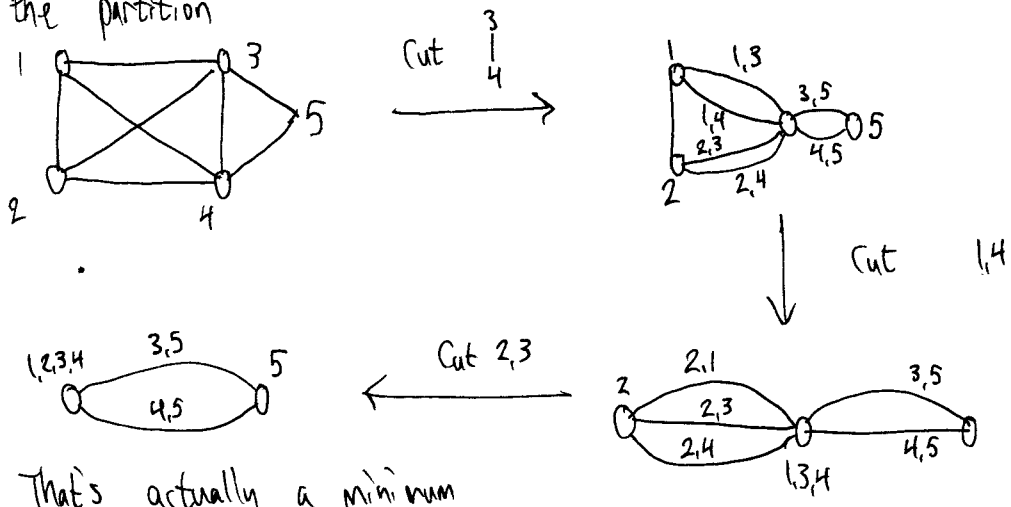
- So, quicksort with random pivots (uniformly) compares $O(n \log n)$ elements in expectation. (But, this doesn't tell us what the distribution. Quicksort happens to perform fairly narrowly but look at "concentration bounds" to actually explain this.)

Chris Hadapp
chadapp3@gatech.edu

2017-01-23f, CS6505 (Computability, Complexity, & Algorithms)

Udacity 16, (Randomized Algorithms), cont.

- A Minimum Cut Algorithm
  - Given: A connected graph $G = (V, E)$
    Output: Set $S$ of minimum size such that $(V, E - S)$ has two connected components
    ($S$ = minimum cut set).
  - This differs from max flow where we are aiming to separate two particular vertices. Here, any vertices may be separated, and all edges are equal weight.
  - One algorithm: Randomly choose an edge. Collapse it. When only 2 vertices remain, the edges left tell one cut set, and the vertices combined to each tell the partition




That's actually a minimum cut set, but isn't guaranteed to be.

- Theorem: This algorithm outputs a min-cut set with probability at least $\frac{2}{n(n-1)}$.
- Corollary: The smallest set of those returned by $\frac{n(n-1)}{2} \ln 1/\epsilon$ callstrans is a min. cut set with probability $1 - \epsilon$.
  - Proof: Each run is independent, so probability that all fail is at most $\left(1 - \frac{2}{n(n-1)}\right)^{\frac{n(n-1)}{2} \ln 1/\epsilon} \leq \exp\left(-\ln 1/\epsilon\right) = \epsilon$

Note that $(1 - x) \leq \exp(-x)$

2017-01-23

Udacity 16 (Randomized Algorithms), cont.

- Proof of theorem:
  Let $C$ be a min. cut set. (→ There can be many. Pick one.) Let $E_i$ = event that edge chosen in iteration $i$ is not in $C$.

$$\Pr\left(\bigcap_{i=1}^{n-2} E_i\right) = \Pr\left(E_{n-2} \middle| \bigcap_{i=1}^{n-3} E_i\right) \cdots \Pr(E_2 | E_1) \Pr(E_1)$$

↑ If we avoided $C$ at iteration $i$, conditioned on avoiding at all other iterations prior

- Claim: $\Pr\left(E_{j+1} \middle| \bigcap_{i=1}^{j} E_i\right) \geq \frac{n-j-2}{n-j}$

- Warm-up: $\Pr(E_1) \geq \frac{n-2}{n}$
  Let $k = |C|$. Then every vertex must have degree at least $k$, (If $< k$, then we could define a smaller cut set by cutting all the edges on that vertex.)
  Also: $|E| \geq \frac{nk}{2}$
  - $n$ vertices, each vertex has degree $\geq k$. An edge connects 2 vertices, hence $nk$ is an overestimate by a factor of 2 - thus $\frac{nk}{2}$.
  - So, $\Pr[E_1] = 1 - |C|/|E| \geq 1 - \frac{k}{kn/2} = 1 - \frac{2}{n} = \boxed{\frac{n-2}{n}}$

- This generalizes...
  Given $\bigcap_{i=1}^{j} E_i$, $C$ is still a min-cut set. Let $k = |C|$, then, $|E| \geq \frac{k(n-j)}{2}$

Still not sure why $C$ is still min-cut

$$\Pr\left(E_{j+1} \middle| \bigcap_{i=1}^{j} E_i\right) = 1 - \frac{|C|}{|E|} \geq 1 - \frac{k}{(n-j)k/2} = 1 - \frac{2}{n-j} = \frac{n-j-2}{n-j}$$

- Substituting into $\Pr\left(\bigcap_{i=1}^{n-2} E_i\right)$ above:

$$\geq \frac{1}{3} \cdot \frac{2}{4} \cdot \cdots \cdot \frac{n-3}{n-1} \cdot \frac{n-2}{n} = \frac{2}{n(n-1)}$$ (telescoping product)

Chris Hodapp
chodapp3@gatech.edu

2017-01-24, CS6505 (Computability, Complexity, & Algorithms

CLRS, Chapter 5 (Probabilistic Analysis & Randomized Algorithms)

- Uniform random permutation: All n! permutations appear with equal probability.
- N.B. Even if we don't know anything about distribution of inputs we can often still randomize them in some way and enforce that they then follow some distribution.
- Book uses $I\{A\}$ for indicator random variable
  - Also (Lemma 5.1): For sample space S & event $A \in S$, let $X_A = I\{A\}$, Then $E[X_A] = Pr\{A\}$.
  - They're very useful for things like repeated trials, especially in conjunction with $E[\ldots]$'s linearity.
- For many randomized algorithms, no particular input produces worst-case behavior.
- Randomized algorithm ≠ probabilistic algorithm. input
  - The latter tends to make assumptions on [redacted], then determine average-case performance. The former makes no assumptions on input, but typically randomizes it purposely.
- Ignored so far: How do we randomly permute an array?
  - A common way: Assign all n elements a random key. Sort them by that key. This produces a uniform random permutation (proof is on p125).
  - In-place randomization of all n elements in order, also works (p126)
  - N.B. The condition $\forall i$ $A[i]$ is equally likely (with $p = 1/n$) to move to any position $j$ is not sufficient to show a uniform random permutation.
- A k-permutation on n elements = a sequence containing k of the n elements with no repetitions. ($n!/(n-k)!$ of them exist)
- Birthday Paradox, 5.4.1 (p130)

N.B. "We will not be using memoization in this course... "dynamic programming" for us will always mean the bottom-up, nonrecursive method."

2017-02-06

Udacity, 9 (Dynamic Programming)

- Here we focus on the class P, that is, all problems decidable in polynomial time
- Dynamic Programming:
  - It's a divide & conquer method, not really an algorithm
  - It's not really 'dynamic'. That term was added to make it sound more exciting. It's also only 'programming' in the older sense of any sort of tabular calculation.
- See: CLRS, and Algorithms (Dasgupta, Papadimitriou, Vazirani)
- Optimal Similar Substructure
  - Consider how we break a hard problem into smaller & smaller subproblems until we've reached trivial ones
  - But, suppose that it also has a lot of repeated pieces, such that recursion would be wasteful (e.g. Fibonacci if we solve it naively)
    - Two ways around this:
      - Memoization (store pieces already computed)
      - Do subproblems in right order (this is always possible because the dependencies have to form DAG) so that edges always point same direction
        - This tends to be more direct than recursion & memoization
- Example: Sequence Alignment/Edit Distance

  e.g. X: A G A G C T A G
  Y: G T T G A C A

  → [A] GA [-] G [-] C [T] A G
  [-] G T [T] G [A] C [-] A

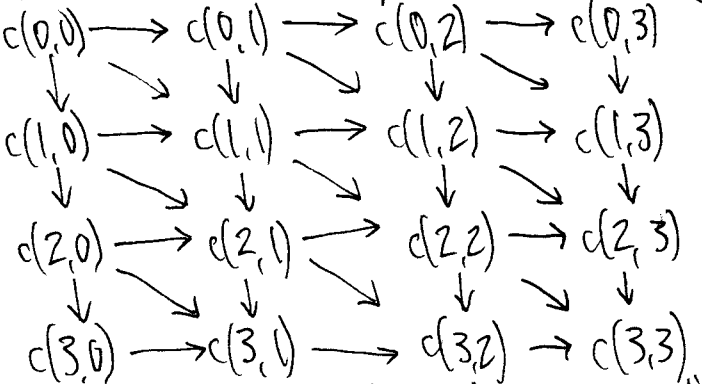
  - Consider seqs. $X = (x_1, x_2, \dots x_m)$ & $Y = (y_1, y_2, \dots y_n)$ over alphabet $\Sigma$.
  Alignment is subset $A \subseteq \{1 \dots m\} \times \{1 \dots n\}$ s.t. for any $(i,j)$ and $(i',j')$: $i \neq i'$, $j \neq j'$, $i < i' \Rightarrow j < j'$.
  Cost(A) = # of unmatched characters $(n+m-2|A|)$
  = # of insertions or deletions - see boxes above
  $+ \sum \alpha(x_i, y_i)$ - typo. $= 0$ if $x_i = y_i$
- What's the 'substructure' to solve this optimally with DP?

Chris Hodapp
chodapp3@gatech.edu

2017-02-06(b), CS6505 (Computability, Complexity, & Algorithms)

Udacity 9 (Dynamic Programming), cnt.

- Prefix substructure
  - Let $c(i,j)$ be minimum cost of aligning $x_1, \ldots x_i$ with $y_1 \ldots y_j$. Goal is to find $c(m,n)$.
  - Case 1: Match the last element of sequence, e.g. A G T C | G / G T T C | A
    Then: $c(m,n) = c(m-1, n-1) + \alpha(x_m, y_n)$
  - Case 2: Leave $x_m$ unmatched. e.g. A G T C | G / G T T C A | -
    Then: $c(m-1, n) + 1$
  - Case 3: Leave $y_n$ unmatched - $c(m, n-1) + 1$ similarly.
- So $c(m,n) = \min\{c(m-1, n-1) + \alpha(x_m, y_m), c(n-1, n) + 1, c(m, n-1), + 1\}$ and nothing here is special to m or n. We can replace m & n with i & j, and we have the base case $c(0,j) = j$, $c(i,0) = i$.
- So this gives us a recursive definition. However, if we implemented it as such, it would recompute a lot of things redundantly.
- But look at the subproblems and how they depend on each other:



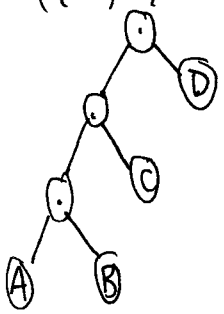

  - Each square (besides bases) depends on west, north, & northwest neighbors
  - This is just a DAG and we can linearize it. One easy way is scanline order after base cases. That is, do 1st row & 1st column, then scan left-to-right and go to next row.
  - Further, once we compute it we can retrace the minimum of each 3 neighbors in order to find the actual alignment.
  - This ends up as $\Theta(mn)$.

2017-02-06

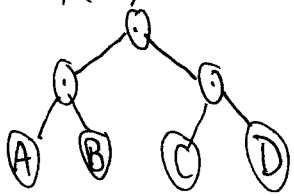
Udacity 9 (Dynamic Programming), cont.

- DP depends on that optimal minimum substructure.
- Next: Chain Matrix Multiplication
  - Given matrices $A_1 A_2 \dots A_n$ of sizes $m_0 \times m_1$, $m_1 \times m_2$, ... $m_{n-1} \times m_n$, compute $A_1 A_2 \dots A_n$ (product) efficiently.
  - Note that matrix multiply is associative: $(AB)C = A(BC)$.
    - they're equal but the number of operations may differ
    - e.g. A is $50 \times 20$, B is $20 \times 50$, C is $50 \times 1$.
      $(AB)C$ is $50 \cdot 20 \cdot 50 + 50 \cdot 50 = 52,500$ operations
      $A(BC)$ is $20 \cdot 50 \cdot 1 + 50 \cdot 20 \cdot 1 = 2,000$
  - Consider expression trees:

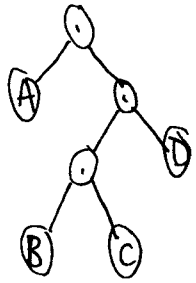
$((AB)C)D$:




$(AB)(CD)$




$A((BC)D)$




  - This suggests trying each possibility of 'partitioning' left & right.
  - Let $C(i,j)$ be minimum cost of multiplying $A_i \dots A_j$.

$$C(i,j) = \min_{i \le k < j} \left[ C(i,k) + C(k+1,j) + m_{i-1} m_k m_j \right]$$

minimum cost of: Subtree k, plus cost of combining those two matrices.

with $C(i,i) = 0 \ \forall i$ for base case. (It's just matrix i by itself)

  - And again, this is a recursive formula, but actually implementing it would recompute various things.
  - Helpfully, $C(i,i)$ is the base cases (diagonals), and the rest is the problems 'above' this.

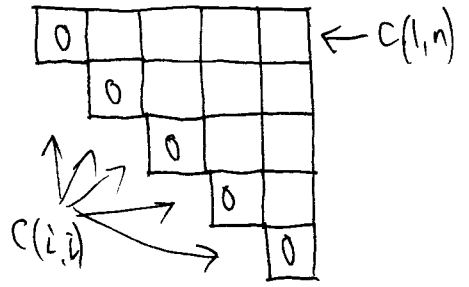
N.B. $i \le k < j$ (≤: less than or equal; <: just less than)

Chris Hodapp
chodapp3@gatech.edu

2017-02-06c, CS6505 (Computability, Complexity, & Algorithms)

Udacity 9, Dynamic Programming, cont.

eg.




In general, every entry depends on elements to the left & downwards. The simplest way to linearize is probably to diagonalize.

```
eg. for s = 1 to n-1
      for i = 1 to n-s
        j = i + s
        Compute C(i,j)
    return C(1,n) - the final cost
```

- To actually produce optimal tree, we need to keep track of which k minimizes (recall C(i,j) formula)
- Theorem: An algorithm exists which, given the dimensions of n matrices gives the expression tree that requires the fewest (scalar) multiplications in $\Theta(n^3)$ time.
- Last example: All Pairs Shortest Path
  - Given graph $G=(V,E)$ and $w: E \to R$ (costs), find "shortest" path between u & v for all $(u,v) \in V \times V$, i.e minimize $\sum_{e \in p} w(e)$.
  - 'Shortest' is sort of figurative. Weights can be negative.
  - Dijkstra's algorithm can find shortest from one vertex to all others in $\Theta(|V| \log |V| + |E|)$ but requires $w \geq 0$.
  - Bellman-Ford can do it in $\Theta(|V||E|)$ for all weights (incl. negative)
  - However, if we want the shortest path for every pair of vertices, then we'd have to run those algorithms $|V|$ times each - so $\Theta(|V|^2 \log |V| + |V||E|)$ & $\Theta(|V|^2 |E|)$ respectively.
  - The former's not awful; the latter is, especially for dense graphs.
- We can do better with DP though.
- The substructure we can exploit: Subpaths of shortest paths are themselves shortest paths.

"single source problem"

2017-02-06

Udacity 9, dynamic programming (cont.)

- Simple example: Suppose we have a shortest path between u & v below:

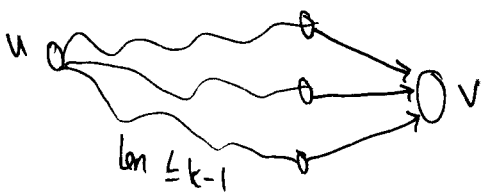

(squiggly lines = paths, straight lines = edges)

- The above subpath between x & y must be a shortest path too. (Why? If it were not, and there were some other shorter path, we could swap this out in the path between u & v and then create a shorter path - contradicting that the path was the shortest.)
- But by its own this isn't really enough (how do we build a shortest path in the first place?)
- So... recurse on path-length.

Let $\delta(u,v,k)$ be the weight of the shortest path that uses at most $k$ edges. Then...

$$\delta(u,v,k) = \min\left\{\delta(u,v,k-1),\ \min_{x \in V(v)} \delta(u,x,k-1) + w(x,v)\right\}$$

Intuitively:




Shortest path of these

→ $\Theta(|V|^3 \log|V|)$ matrix mult. (?) algorithm - see CLRS but we take a somewhat faster approach.

- We instead recurse on potential intermediate vertices

Assume vertices = 1...n

Let $\delta(u,v,k)$ be the min. weight of a path from u to v using only 1...k as intermediate vertices, so...

$$\delta(u,v,k) = \min\left\{\delta(u,v,k-1),\ \delta(u,k,k-1) + \delta(k,v,k-1)\right\}$$

Basically: min (use k, don't use k)

$$\delta(u,v,0) = \begin{cases} w(u,v) & \text{if } (u,v) \in E \\ \infty & \text{otherwise} \end{cases}$$

- Base case just uses weights

Chris Hodapp
chodapp3@gatech.edu

2017-02-06d, CS6505 (Computability, Complexity, & Algorithms)

Udacity 9, Dynamic Programming, cont.

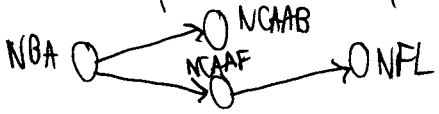
- This is the basis for the Floyd-Warshall Algorithm
- Again, it's recursive but not efficient to compute recursively (too much recomputation)
- We start by building up an $n \times n$ table...

```
for i = 1 to n
    for j = 1 to n
        d[i][j] = { w(i,j)   if (i,j) ∈ E
                  { 0        if i = j
                  { ∞        otherwise
```

- Now, add additional intermediate vertices, one by one:

```
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            d[i,j] = min { d[i][j],
                         { d[i][k] + d[k][j] }
```

- To get the actual paths, we keep a predecessor table, $\Pi$:
  - Initialize with $\Pi[i][j] = \begin{cases} i & \text{if } i,j \in E \\ \text{Null} & \text{otherwise} \end{cases}$
  - Update above; either leave it alone (if $d[i][j]$ is minimum) or set to the predecessor of the other half of the path from $k$ to $j$ (i.e. $\Pi[k][j]$ I think)
- Theorem: Floyd-Warshall Algorithm finds shortest path in a weighted graph for all pairs of vertices in $\Theta(|V|^3)$ time
- Its optimal similar substructure: Optimal paths via restricted set of vertices $1 \ldots k$.
  That is what lets us produce the recurrence.
- Transitive closure: Consider relation $R$ over a set $A$, i.e. $R \subseteq A \times A$
  e.g. sports preferences: NBA > NCAAB, NBA > NCAAF, NCAAF > NFL
- These form a DAG as well:



2017-02-06

- but given this we'd like to infer that the person likes the NBA over the NFL (for instance)
- This is basically equivalent to what we just did! set edge weights all to 1 and run Floyd-Warshall!
  - But we don't need all this information. We don't care how many relations we had to apply transitively, or what they are - just whether they exist. We can simplify a bit:

Init:
```
For i := 1 to n
    for j := 1 to n
        t[i][j] = { 1  if (a_i, a_j) ∈ R
                  { 0  otherwise
```

Then:
```
For k := 1 to n
    for i := 1 to n
        for j := 1 to n
            t[i][j] = t[i][j] ∨ (t[i][k] ∧ t[k][j])
```

Unless P=NP of course.

- DP won't yield a polynomial-time solution to every problem with a recursive formulation. (eg. satisfiability)
- However, keep it in mind as a possible technique anytime you see a problem that can be solved with overlapping subproblems.

2017-02-07

CLRS Ch. 15 (DP)

- DP typically is applied to:
  - Optimization problems (which may lack a unique solution)
  - Problems where subproblems overlap
- Steps (typically):
  1. Characterize an optimal solution's structure
  2. Recursively define the value of an optimal solution.
  3. Compute value of an optimal solution, typically bottom-up
  4. Construct an optimal solution from computed information. (Omit this if we need only optimal value, not solution)

Chris Hodapp
chodapp3@gatech.edu

2017-02-07(b), CS6505 (Computability, Complexity, & Algorithms)

CLRS 15 (DP), cont.

- Step 4 sometimes requires keeping additional info at step 3
- Example: Rod-cutting problem
  - Suppose we have a steel rod $n$ inches long and can cut it (in increments of an inch) to any number of pieces. Pieces of each length have a different price, e.g.

| length $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| price $p_i$ | 1 | 5 | 8 | 9 | 10 | 17 | 17 | 20 | 24 | 30 |

  ... and we want to maximize total revenue. Where do we make cuts to do this? (e.g. for $n=10$, it's no cuts, then $\sum p = 30$)
- Naively, we can cut in $2^{n-1}$ different ways (each inch is a possible cut).
- For $n = i_1 + i_2 + \dots + i_k$ ($k-1$ cuts) revenue is $r_n = p_{i_1} + p_{i_2} + \dots + p_{i_k}$.
  But note... (if $r_n$ = optimal revenue)

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$$

  - $p_n$: No cuts
  - $r_1 + r_{n-1}$: Cut at 1", and cut the rest optimally
  - $r_2 + r_{n-2}$: Cut at 2", and cut the rest optimally
  - ... (Optimal cut at $r_i$ and at $r_{n-i}$)

  - Once we make 1 cut we treat the two subproblems as independent
- This exhibits optimal substructure: optimal solutions to a problem incorporate optimal solutions to related subproblems.
- We may simplify a little: $r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$.

Under this, we have just one related subproblem, not two. We treat the "left" part as something we can't subdivide further, while we do subdivide the right. This still covers all cases.

- However, a naive recursive implementation is $\Theta(2^n)$.
- Dynamic programming is a time-memory trade-off. It stores more in memory, but might turn an exponential problem into a polynomial-time one. (If # of distinct subproblems is polynomial in input size, and we can solve each one in polynomial time.)

Side question: Do things like BLAS & LAPACK do optimal multiplication of a chain of matrices or have a means of computing it?

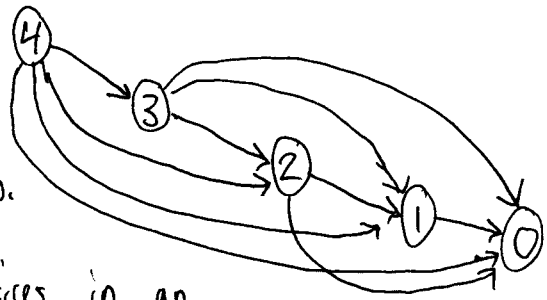
CLRS 15, DP (cont.)

2017-02-07

- The basis of DP is simply saving results so that they need not be recomputed. Top-down with memoization is one way (and we may apply this to the naive recursive solution); typically one just adds an array or hash table.
  - The bottom-up method is another way (the only one this class is concerned with).
    - Solve "smaller" subproblems first, and build up. Solve each subproblem only once.

See p366 for pseudocode, & p369

- Subproblem graph for rod-cutting with $n = 4$:
  - Each edge $a \to b$ means that to solve subproblem $a$ we need subproblem $b$.




  - To do the bottom-up method here we must consider the subproblem graph's vertices in an order that solves subproblems before those that depend on them. (eg. reverse topological sort)
  - DP typically is linear in the number of vertices & edges.
- p370 (15.2) is repeat of matrix-chain multiplication from lectures
- N.B. Optimal substructure suggests that DP applies - but also suggests that greedy algorithms might (see ch. 16)
  - In DP, look for a sort of "cut-and-paste" in an optimal solution: If you assume an optimal solution contains non-optimal subsolutions, could you produce a "better" solution (a contradiction) by cutting out the non-optimal solution & pasting in an optimal one (if you had it)?
- We covered weighted shortest path in the lectures. Note that a somewhat similar problem - unweighted longest simple path - does not exhibit optimal substructure. Further, we can't even build up optimal subsolutions into a 'legal' solution (in this case, we assume simple paths - no repeating vertices).
  What makes it so different is that in shortest path, subproblems are independent. Solution to one subproblem can't affect solution to another.

Chris Hodapp
chodapp3@gatech.edu

2017-02-07c, CS6505 (Computability, Complexity, & Algorithms)

CLRS, 15 (DP), cont.

- It sounds a little contradictory to say that for dynamic programming subproblems must be both overlapping & independent. It's not: Independent just means they don't 'share resources' while overlapping means that different problems might produce the exact same subproblem.
- It helps to store a table of the 'choice' made in each subproblem in order to reconstruct the optimal solution (which is typically computed anyway but not necessarily stored at each step).
- Bottom-up DP usually outperforms memoized top-down (despite their asymptotic complexities generally being identical) by a constant factor; it has less overhead due to no recursion.
  - Also, the regular pattern of table access in a given DP solution may lend itself to further optimization
  - ~~Memoization~~ Memoized can be faster if bottom-up solves subproblems that aren't actually needed.
- p390, sec. 15.4: Longest common subsequence problem (covered in lecture)
- Another example: Optimal binary search tree
  - We're looking things up in a tree, and they all have different probabilities; we want to minimize the number of nodes we must visit in order to find each item (given their frequencies).
- Suppose we have sorted sequence $K = \langle k_1, k_2, \ldots k_n \rangle$ and each key $k_i$ has a probability $p_i$ that a search will be for that key. There will also be $(n+1)$ dummy keys not in $K$ which will be searched for, $d_0, d_1, \ldots d_n$. $d_0$ is all values $< k_1$, through $d_n$ which is all values $> k_n$; $d_i$ is all values between $k_i$ & $k_{i+1}$.
- Each dummy key $d_i$ has probability $q_i$. (In example, dummy keys are for 'unsuccessful' searches?)

So: $\sum p_i + \sum q_i = 1$.

Other side note:
Bellman (as in Bellman equation and as in Curse of Dimensionality) is credited with the systematic study first of dynamic programming in 1955.

2017-02-07

CLRS, Ch. 15 (DP), cont.

- Expected cost of a search in T is then...

$$\sum_{i=1}^{n}\left(\left(\text{depth}_T(k_i)+1\right)\cdot p_i\right)+\sum_{i=0}^{n}\left(\left(\text{depth}_T(d_i)+1\right)\cdot q_i\right) = 1+\sum_{i=1}^{n}\text{depth}_T(k_i)\,p_i+\sum_{i=0}^{n}\text{depth}_T(d_i)\,q_i$$

due to definition on last page (also we assume a +1 on each search)

sec 15.5

- An <u>optimal binary search tree</u> minimizes that expected cost. This doesn't always mean a tree with the smallest height.
- An exhaustive search is $\Omega\left(4^n/n^{3/2}\right)$...
- Goal: label nodes of any n-node binary tree with $k_1, k_2, \ldots k_n$ to make a binary search tree, then add dummies as leaves.
- What optimal substructure can we exploit?
  - Any <u>subtree</u> of an optimal binary tree must itself be optimal (over the specific subset of keys & dummy keys)
  - This is true for any key $k_r$ which is taken as the root of left & right subtrees, provided we take all keys of those subtrees
- This has similarities to matrix-chain multiplication, and it runs in $\Theta(n^3)$.

Pryby lecture, Greedy Algorithms (part 1)

2017-02-10

- Optimization algorithms often involve many choices at each step, & DP tries to enumerate every choice. This sometimes is overkill.
- <u>Greedy algorithms</u> pick the choice that looks best (locally), and sometimes this produces a globally optimum solution
- Goals of this section:
  - Design a greedy algorithm
  - Prove that a problem demonstrates <u>greedy-choice</u> & optimal substructure
  - Demonstate that problem can be reframed as a <u>matroid</u> (giving an almost automatic greedy solution)
- This assumes we've already watched DP lectures, and familiarity with linear algebra (linear independence) & graph theory (trees, cycles, forests, weighted graphs)

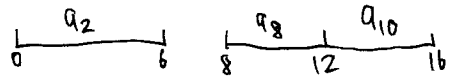
Chris Hodapp
chodapp3@gatech.edu

2017-02-10(b), CS6505 (Computability, Complexity, & Algorithms)

Prybys lecture, Greedy Algorithms, 1 (cont.)

- Example: Activity selection problem
  - Schedule competing activities that require exclusive use of a resource
  - $S = \{a_0, \ldots a_{n-1}\}$ - activities, e.g. jobs for a processor
  - Each activity has start time $s_i$, finish time $f_i$ (for $a_i$). $0 \leq s_i < f_i$.
  - $a_i$ & $a_j$ are <u>compatible</u> if their times dont overlap - iff $[s_i, f_i) \cap [s_j, f_j) = \emptyset$ or iff $s_i \geq f_j$ or $s_j \geq f_i$
  - Goal: Select max-size set of <u>mutually-compatible</u> activities.
  - Assume that $f_0 \leq f_1 \leq \ldots f_{n-1}$ - theyre ordered by finish time (this is a common assumption on input. If its not true, the extra $\Theta(n \log n)$ usually doesnt matter)
  - e.g.

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $s_i$ | 1 | 3 | 0 | 5 | 3 | 5 | 6 | 8 | 8 | 2 | 12 |
| $f_i$ | 4 | 5 | 6 | 7 | 9 | 9 | 10 | 11 | 12 | 14 | 16 |

    - $\{a_2, a_8, a_{10}\}$ is mutually compatible: 

      but is not a maximum.
    - $\{a_0, a_3, a_7, a_{10}\}$ is mutually compatible & larger; $\{a_1, a_3, a_8, a_{10}\}$ too (these end up being maximum solutions)
  - Try a dynamic programming approach first. What is optimal substructure?
  - Let $S_{ij}$ = activities starting after $f_i$ & finishing at or before $s_j$. (basically, those compatible with $a_i$ & $a_j$).
    Let $A_{ij}$ = max-sized set of mutually compatible activities in $S_{ij}$.
    $a_k \in A_{ij}$, $A_{ik} = A_{ij} \cap S_{ik}$, $A_{kj} = A_{ij} \cap S_{kj}$ - this suggests splitting $A_{ij}$

    So: $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$, since all disjoint
    - This follows the cut-and-paste structure too: If we can construct a 'larger' $A_{ik}$ or $A_{kj}$ we can contradict that $A_{ij}$ is optimal.
    - So with DP we'd let $k$ vary through all possibilities, and pick the best, as a recurrence:

2017-02-10

Pryby lecture, greedy algorithms, 1 (cont)

$$c[i][j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max_{k \in S_{ij}} (c[i][k] + c[k][j] + 1), & S_{ij} \neq \emptyset \end{cases}$$

- DP could probably do this in $\Theta(n^3)$, depending on $S_{ij}$ computation
- For a greedy strategy we might notice $S_{ij}$ is a bit:
  - Let $S_k$ = activities starting after $f_k$
  - What if we look only at the activity finishing first, and ignore how long it actually runs?

```
def select_activities(S):
    n = len(S)
    A = {S[0]}
    prev_finished = 0
    for i in range(1, n):
        if S[i].s >= S[prev_finished].f:
            A = A ∪ {S[i]}
            prev_finished = i
    return A
```

  - Assume $S$ is list sorted by finish time (increasing)
  - Index activity finished most recently!
- Proposed strategy is wrong. E.g. $A_1$, if set operations are $\Theta(1)$.
- A loop invariant easily shows that it is not true maximum compatible activities, but it's only a maximum set!
- We want to know: when is the greedy choice actually contained in an optimal solution?
- Consider $\{a_k\} \cup A_k$ where $A_k$ = a max. mutually compatible subset in $S_k$.
  We know that solution of size $m$ is an optimal solution in $S_k$.
  But it's not guaranteed that $a_k$ = activity finishing first = actually in any optimum solution (?)
- Proof:
  $A_k$ = max sized subset of mutually compatible activities in $S_k$.
  Let $a_k$ = activity in $S_k$ with earliest finishing time
  Let $a_j$ = a.t. in $A_k$ with earliest finishing time
  - If $a_j = a_k$ then done, have greedy choice property in $A_k$.

2017-02-10, CS6505 (Computability, Complexity, & Algorithms)

- We already know that all of $A_k$ (ignoring $a_\ell$ & $a_j$) has to be compatible.
  Since $a_j$ had the earliest finish time in $A_k$, nothing else in $A_k$ is "before" $a_j$. However, since $a_\ell$ must finish at or before $a_j$ does (note that $a_\ell$ is the earliest of $S_k$ and $A_k \subseteq S_k$), and since we've removed $a_j$, this must leave 'space' for $a_\ell$ - so it is also compatible. Thus, $A'_k$ is still mutually compatible.
- But since $A_k$ & $\{a_\ell\}$ are disjoint, $|A'_k| = |A_k| + 1 - 1 = |A_k|$. And $A_k$ is already a maximum set, thus if $A'_k$ is the same size, it also is a maximum set, and it contains $a_\ell$, the greedy choice.
- That is, maybe we can make an optimal set without the greedy choice, but can always show one with it. (See CLRS sec. 16.1.)

- Designing greedy algorithms (16.2)
  - We don't need to use DP first to get to a greedy algorithm.
  - Steps will typically follow:
    1. Rephrase in terms of making a choice & solving a single subproblem. That subproblem should look like the original but with one fewer choices.
    2. Prove greedy-choice property: An optimal solution always exists that makes the greedy choice, that is, greedy choice is always safe.
    3. Demonstrate optimal substructure property: Show that having made the greedy choice, what remains is a subproblem with the property that if we combine the optimal solution to that subproblem with the greedy choice, we arrive at an optimal solution to the original problem.
       e.g. above: Show that $\{a_\ell\} \cup A_\ell$ is optimal solution to $S_k$.

Typically a bit difficult to show

2017-02-10

Pryby lecture, greedy algorithms, 1 (cont.)

- Example: Knapsack problem
  - n different items, each has value $v_i$ & weight $w_i$.
    $S = \{a_0, a_1, \ldots a_{n-1}\}$.
  - Goal is to maximize $\sum v$ but $\sum w \leq W$ (total weight)
  - This is the 0-1 knapsack problem (can't divide an item)
    - It's NP-hard; a greedy algorithm can't solve it.
  - If we relax this and say items can be divided, it is solvable by greedy algorithm. ('Fractional knapsack problem')
  - Weight & value both scale. If we take $1/2$ of $a_0$, value is $\frac{1}{2}v_0$ & weight is $\frac{1}{2}w_0$.

1. Rephrase for greedy choice

   Let $q_i = v_i / w_i$. Greedy choice is: take as much of $a_i$ as possible, where $a_i$ has highest $q_i$. (Intuitively, we want to get as much value as we can for the least weight.)

   Subproblem: $W - w_i$, $S \setminus a_i$ (less weight available & remove $a_i$)

   For pseudocode first assume they're sorted by decreasing $q$ value.

```
def frac_knapsack(S, W):
    n = len(S)
    amt = [0] * n                  - This holds the amount of items 1...n
    wt_rem = W                     - Weight remaining
    for i in range(0, n):
        amt[i] = min(wt_rem, S[i].w)    - Take as much as possible
        wt_rem -= amt[i]                  (recall that S is
    return amt                             in decreasing q_i order)
```

   $\Theta(n)$ in time & memory

- Next is to prove it actually is correct (steps 2 & 3).
  - 2: Prove that there is an optimal solution containing the greedy choice

Chris Hodapp
chodapp3@gatech.edu

2017-02-10d, CS6505 (Computability, Complexity, & Algorithms)

Prof by lecture, greedy algorithms, 1, cont.

- That is, an optimum solution to $(S, W)$ such that $\underbrace{\min(W, w_0)}_{=m_0}$ is chosen. ↑ of $a_0$

  Suppose that we only select weight $m$ of $a_0$, $m < m_0$. Then exchange $m_0 - m$ of any other items in the bag for $m_0 - m$ of $a_0$.

  Then total value changes by $\geq \overbrace{q_0(m_0 - m)}^{\text{Added}} - \overbrace{q_1(m_0 - m)}^{\text{Removed}}$
  (it's $\geq$ because $q_1$ is the most that could be subtracted)

  But this amount $> 0$ because $q_0 > q_1$. Therefore, this exchange increased the value - which contradicts that the solution's optimal.

  Therefore, $m_0$ ($\min(W, m_0)$) must be optimal.

This is actually stronger than greedy choice property. We proved that all optimal solutions are greedy.

- Optimal substructure property
  - That is, prove that taking $m_0$ of $a_0$ & and then the optimal solution to the resulting subproblem gives an optimal solution to the original problem.
    (subproblem is $(S', W')$, $S' = \{a_1, \ldots a_{n-1}\}$, $W' = W - m_0$; original is just $(S, W)$).
  - Suppose Optimal ~~sol~~ solution to subproblem has solution $V$. (For $(S', W')$)
    $V + q_0 m_0$ = value of combining greedy choice with optimal solution.
    Now suppose we have a better way to do this, with $V' > V + q_0 m_0$.
    Our work in (2) implies that solution ~~sol~~ giving $V'$ must have $m_0$ of $a_0$.
    This implies that there is a solution to $(S', W')$ with value $V' - q_0 m_0$. But by above inequality, this is $> V$, but if that's the case, $V$ isn't the optimal value, which is a contradiction. ↳ For $(S', W')$.
    Thus, $V + q_0 m_0$ is the max value for $(S, W)$.

2017-02-13

Pryby lecture, greedy algorithms, part 2

- Matroids (Following CLRS sec. 16.4, 16.5; we ignore Huffman codings in 16.3)
  - Describes many situations where greedy choice is optimal (but not all, e.g. activity selection can't be cast as matroid)
  - Matroid is: an ordered pair $M = (S, \mathcal{I})$, where $S$ is a finite set (more generally it can be infinite for a matroid but we ignore that here), and $\mathcal{I}$ is a subset of $\mathcal{P}(S)$ - that is, a collection of subsets of $S$. (It's the "independent" subsets) - $\mathcal{I}$ is LaTeX \mathcal{I}
    1- $\mathcal{I}$ is nonempty. (Or, check that $\emptyset \in \mathcal{I}$)
    2- $\mathcal{I}$ is hereditary (or monotonic): If $B \in \mathcal{I}$ and $A \subseteq B \Rightarrow A \in \mathcal{I}$ (implies that $\emptyset \in \mathcal{I}$)

This implies something about maximum sizes of I too

    3- Exchange property: If $A, B \in \mathcal{I}$ and $|A| < |B|$ then there is some $x \in B \setminus A$ such that $A \cup \{x\} \in \mathcal{I}$.
  - Some intuition: Let $S = \{$rows of matrix $A\}$. Let $\mathcal{I} = \{$linearly independent sets of rows of $A\}$.
    - That's a prototypical matroid; it's just matrices.
    - Hereditary property holds: Removing rows (which is what $A \subseteq B$ suggests) can't ever violate linear independence, only adding them.
    - Exchange property: Given a linearly independent set of rows $A$, and a larger one $B$, there is always a row in $B$ (& not $A$) which can be added to $A$ and still have it be linearly independent (or: that is linearly independent from $A$).

(Full proof left as an exercise to the reader)

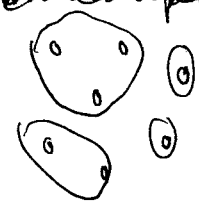
  - Another example unrelated to matrices:
    - Consider an undirected graph $G = (V, E)$. Let $S = E$. Say that $\mathcal{I}$ is the edges that don't form a cycle.
      - $\mathcal{I} = \{A \subseteq E : A$ is acyclic$\} \Leftrightarrow G[A]$ is a forest → or $G_A = (V, A)$ is better notation for subgraph
    - Claim: $M_G = (S_G, \mathcal{I}_G)$ is a matroid (the graphic matroid)
    - 1st property holds: $\emptyset$ trivially is acyclic (no edges)
    - 2nd property holds: Removing an edge never makes a forest cyclic

Chris Hodapp
chodapp3@gatech.edu

2017-02-13(6). CS6505 (Computability, Complexity, & Algorithms)

Pryby lecture, greedy algorithms, part 2 (cont.)

- 3rd property is (as always) a bit trickier:
  - Given a set of edges that is acyclic, and a larger set that is too, some edge exists in the larger set (& not the smaller) which can be added to the first without making it cyclic. $G_A$ $G_B$
  - Suppose $A, B \in \mathcal{I}$ so that $(V, A)$ & $(V, B)$ are forests, and $|B| > |A|$.
  - Lemma: A forest $F = (V_F, E_F)$ contains exactly $|V_F| - |E_F|$ connected components. Starting fact: A tree with $v$ vertices has $v-1$ edges (not proven here).
    Suppose $F$ has $t$ conn. comps. and conn. comp. $i$ has $v_i$ vertices and $e_i$ edges.
    $$|E_F| = \sum_{i=1}^{t} e_i = \sum_{i=1}^{t} (v_i - 1) = |V_F| - t, \quad t = |V_F| - |E_F|.$$
    $\rightarrow \sum v_i = |V_F|$
  - So, $G_A$ has $|V| - |A|$ connected components, $G_B$ has $|V| - |B|$ connected components. But since $|B| > |A|$, $|V| - |B| < |V| - |A|$.
  - The pigeonhole principle sort of applies here. ~~B has more connected components than A, one of them is going to span 2 connected components of A.~~
  - Example:



Suppose A has 4 connected components like this. You can't draw lines to make 3 connected components without spanning two of A's components.

  - So at least one of B's connected components is in two of A's.
  - If we take that edge from B and add it to A, then it can't create a cycle. (It connects two acyclic components of A, but there is still no way to "get back" from one back to the other on just that single edge.)
  - Thus, that new set of edges must be a forest, and we've shown the exchange property.

2017-02-13

Pryby lecture, greedy algorithms 2, cont.

- If $M=(S,\mathcal{I})$, $A\in\mathcal{I}$, then $x\notin A$ is an <u>extension</u> of $A$ iff $A\cup\{x\}\in\mathcal{I}$. ($\{x\}\in\mathcal{I}\Rightarrow x$ is an extension of $\emptyset$)
- $A\in\mathcal{I}$ is <u>maximal</u> iff it has no extension.
  All maximal independent sets in $S$ have the same size.
- Note that for undirected graph example, a maximal independent set is called: a <u>spanning tree</u>. (So all spanning trees must have same number of edges)
- A matroid $M=(S,\mathcal{I})$ is weighted if $w: S\to\mathbb{R}^+$, and $w: 2^S\to\mathbb{R}_{\geq 0}$ is defined by $w(A)=\sum_{x\in A} w(x)$ - that is, we extend $w$ to power set of $S$
- Many greedy solutions can be cast as finding maximum weight of independent subset in a weighted matroid.
  - That is: Goal is to find $A\in\mathcal{I}$ s.t. $w(A)$ is as large as possible, i.e. an <u>optimal</u> independent subset. (It will be a maximal-size independent subset.)
- Consider a graph $G=(V,E)$, $w_G: E\to\mathbb{R}^+$ (graph weight, <u>not</u> matroid weight)
  Matroid $M_G=(S_G,I_G)$, let $w(e)=b-w_G(e)$ - $b$ is pretty arbitrary but we added all edge weights together (our goal is to <u>minimize</u> hence negation but must be $>0$)
- We have an algorithm that can do this for any weighted matroid (it's a greedy algorithm) → list

```
def greedy_matroid(M, w):
    A = ∅
    n = len(M.S)
    sort M.S, decreasing by w(S[i])
    for i in range(0, n):
        x = S[i]
        if A ∪ {x} is in M.I:  A = A ∪ {x}
    return A
```

- We'll use M.S & M.I → set; w is the matroid weight function
- More likely we'll have 'in M.I' as a function

$O(n \log n)$ + $n$ iters + $O(n\,F(n))$ where $F$ is time to check independence

→ $O(n\log n + nF(n))$

Chris Hodapp
chodapp3@gatech.edu

2017-02-13c, CS6505 (Computability, Complexity, & Algorithms)

Pryby lecture, greedy algorithms 2, cont.

- Since $\emptyset$ is independent & the for-loop has invariant that $A$ is independent, the algorithm always returns an independent set.
- But, does it always return largest possible weight? That is: If we make greedy choice, how do we determine that some optimal solution contains it?
- Let $x$ be 1st element of $S$ such that $\{x\} \in \mathcal{I}$. If such an $x$ exists, then there is an optimal set of $S$ containing $x$. (Otherwise $\emptyset$ is optimal)
  - Proof: Assume such an $x$ exists. Let $B$ be an optimal set in $S$. Assume $x \notin B$. No element of $B$ has greater weight than $x$.
    - Why: If $y \in B$ has $w(y) > w(x)$, then $\{y\} \in \mathcal{I}$ but this would contradict that $x$ is 1st element with $\{x\} \in \mathcal{I}$ (note that we sorted $M.S$ in algorithm)
    - Let $A = \{x\} \in \mathcal{I}$. Use exchange property to extend $A$ until $|A| = |B|$ & $A \in \mathcal{I}$. Note that they must differ in one element ($x$). $w(A) = w(x) + w(B) - w(z)$, where $z$ is in $B$ and not $A$. But since $w(x) \geq w(z)$, $w(A) \geq w(B)$.
    - But $B$ is an optimal subset so $w(A)$ can't be greater than $w(B)$.
    - Thus, $w(A) = w(B)$. $A$ contains $x$ and is optimal.
  - Also: The optimal substructure property.
    Let $x$ be 1st element chosen. Then subproblem of finding optimal subset containing $x$ reduces to finding opt. subset of $S'$ in the weighted matroid $(S', \mathcal{I}')$, where $S' = \{y \in S : \{x, y\} \in \mathcal{I}\}$ & $\mathcal{I}' = \{B \subseteq S \setminus \{x\} : B \cup \{x\} \in \mathcal{I}\}$ ←
    - Basically: If you throw away $x$ and all that it would be independent with, and make everything independent in remaining set that would be if you added $x$ back, it has the identical structure to original matroid.
  - Proof: If $A$ is an optimal subset of $S$ containing $x$, then $A' = A \setminus \{x\} \in \mathcal{I}'$

"contraction of $M$ by $x$"

2017-02-13

Prof's lecture, greedy algorithms, 2, cont.

- Conversely, any set $A' \in \mathcal{I}'$ yields independent set $A = A' \cup \{x\} \in \mathcal{I}$.
- In both cases, $w(A) = w(x) + w(A')$. So an optimal solution in $M$ yields optimal solution in $M'$, and vice versa.
- Claim: If $x \in S$ is <u>not</u> an extension of $\emptyset$, then $x$ is not an extension of any independent subset of $S$.
  - Proof: If $x$ is ext. of $A \in \mathcal{I}$ then $A \cup \{x\} \in \mathcal{I}$, so $\{x\} \in \mathcal{I} \Rightarrow x$ is ext. of $\emptyset$, which is a contradiction.

  (So our algorithm, if it skips over the first few elements, doesn't skip anything we could use.)
- So... this algorithm is a very standard solution we can use in many cases. It doesn't work in every case, and it's also sometimes less efficient than a more direct solution.
- Consider CLRS 16.5, task scheduling problem.
  - He sees this as a very good example of how to break a problem down into matroid terms
  - Tasks: $S = \{a_0, a_1, a_2 \dots a_{n-1}\}$, each taking unit time

    Schedule = permutation of $S$

    $0 \to 1 \to 2 \dots \to n$

    1st task, 2nd task, last task
  - $a_i$ has deadline $1 \le d_i \le n$ & penalty $p_i > 0$ for failure to meet deadline
  - Goal: Find schedule minimizing penalty
  - Fact: Schedule can be arranged into a "canonical" form with all "on-time" tasks preceding all late tasks & with on-time tasks sorted in increasing order by deadline.
  - Example:

| | $a_0$ | $a_1$ | $a_2$ | $a_3$ |
|---|---|---|---|---|
| $d_i$ | 3 | 1 ~~8~~ | 4 | 1 ~~2~~ |

$\to$

| | $a_1$ | $a_0$ | $a_2$ | $a_3$ | |
|---|---|---|---|---|---|
| | 1 | 3 | 4 | 1 | Deadline |
| | On-time | | | Late | |

Task

$a_1$ & $a_3$ can't both be done

Note that if they're already late, when they're done doesn't matter.
  - Say that $A \subseteq S$ is independent if they can be scheduled so that none of the tasks are late. Let $\mathcal{I}$ = independent sets of tasks.

Chris Hodapp
chodapp3@gatech.edu

2017-02-13 d, CS6505 (Computability, Complexity, & Algorithms)

Prof by lecture, greedy algorithms, 2, cont.

- Lemma: For $t=0,1,\dots n$, let $N_t(A)$ be the # of tasks in A whose deadline is t or earlier.
  - So for our example, $N_0(A)=0$, $N_1(A)=2$, $N_2(A)=2$, $N_3(A)=3$, $N_4(A)=4$
  - Then these are equivalent:
    a) A is independent (can schedule so none are late)
    b) $N_t(A) \le t$ for all t (all tasks can be done)
    c) If task scheduled in incr. order by $d_i$, no task is late.

Not proven here (it's in CLRS)

- (b) is most useful to us.
- Claim: $(S, \mathcal{I})$ is a matroid.
  - Property 1 is trivially true
  - Property 2 is true: If we remove a task, then existing tasks don't become late
  - Property 3 as usual takes longer:
    Let $A, B \in \mathcal{I}$, $|B| > |A|$.
    Let $k$ = largest $t$ s.t. $N_t(B) \le N_t(A)$.
    $N_0(B) = 0 = N_0(A)$.
    $N_n(B) = |B| > |A| = N_n(A)$ $\Rightarrow 0 \le k \le n-1$
    So $N_k(B) \le N_k(A)$ but $N_{k+1}(B) \not\le N_{k+1}(A)$
    So... more tasks with deadline $k+1$ in B than A.
    Pick $x \in B \setminus A$ with deadline $k+1$. We show $\underbrace{A \cup \{x\}}_{A'} \in \mathcal{I}$.
    For all $t \le k$, $N_t(A') = N_t(A) \le t$
    (Why: $x$ had deadline $k+1$, and we already know A is independent.)
    For $t > k$, $N_t(A') \le N_t(B)$ because $N_t(A') = N_t(A) + 1$ and $N_t(B) \le t$.
    So for all $t$, $N_t(A') \le t \Leftrightarrow A' \in \mathcal{I}$.
- So now all we need is to define a weight function.
  - ~~We can do it similarly. Find the maximum penalty.~~
  - We can just use $w(a_i) = p_i$ and this maximizes the penalty that we do not incur. (since we don't incur any penalty for what's on time)

?

2017-02-13

- We can check independence in $\Theta(n)$ so overall it's $\Theta(n \log n + n^2) = \Theta(n^2)$.
- Don't worry about the long proofs in these lectures. In an exam, he might give a simpler problem that turns directly to a matroid problem, and ask for a way to get its time (via the given algorithm) to some amount by how you compute independence.

CLRS, 16 (Greedy Alg.)

- Choice made in greedy algorithm can depend on choices made so far, but cannot depend on future choices or subproblem solutions.
- Dynamic programming solves the subproblems first.
- Greedy algorithm makes first choice before subproblems. (i.e. top-down).

- Ch. 6, Dynamic Programming (Dasgupta, Papadimitriou, & Vazirani)
- A DAG can always be linearized: arranged in a line so all edges go left-to-right. (Topological Sorting)

2017-02-19

Pryby lecture, Amortized Analysis

- See also CLRS 17.3
- In amortized analysis, we average the time required over all operations performed.
- Intuition: Even if a function is very expensive to call, perhaps it's very rarely called.
- Consider a stack with constant-time push & pop. (S.push(x), S.pop - & S.pop throws error if empty)
- Add another operation, S.multipop(k) which removes top $k' = \min(k, \text{len}(s))$. We could do this as multiple pop calls. That would be $\Theta(k')$ to run, and also $O(\text{len}(s))$ - always $\leq \text{len}(s)$.
- Consider n pushes, pops, or multipops (starting from empty stack). What's worst case for these?

Chris Hodapp
chodapp3@gatech.edu

2017-02-19(b), CS6505 (Computability, Complexity, & Algorithms)

Prof by lecture, Amortized Analysis, cont.

- len(S) $\leq n$ at all times (S starts empty & we do at most n pushes)
- If we do $\Theta(n)$ multipops, $O(n)$ time per op $\rightarrow O(n^2)$ worst-case
  - This is correct but not tight.
  - We ignore the state of the stack in this estimate
- Observation: Each object pushed onto stack can be popped at most once. (per push)
  $\rightarrow$ total # of pops $\leq$ total # of pushes $\leq 2n$
- So total time is then $O(n)$, noting constant-time push & pop
  $\rightarrow$ Average time per operation $= O(n)/n = O(1)$
- Objective:
  - Estimate average amortized runtime on a data structure. We'll use the potential method (CLRS 17.3).
  - Learn to design structures & algorithms for a specific amortized runtime
- Prerequisites: Stacks, queues, heaps, data structures in general, asymptotic notation
- Potential method:
  - Analogy is that data structure stores potential energy as it is operated on; large, disruptive operations use up stored potential.
  - Imagine we do $n$ operations from state $D_0$.
    States progress $D_0 \rightarrow D_1 \rightarrow D_2 \rightarrow \dots D_n = f$.
    Define potential function $\Phi(D)$ = potential of state $D$.
    $c_i$ = actual cost of $i$th operation.
    Amortized cost of op $i$ is: $\hat{c}_i = c_i + \overbrace{\Phi(D_i) - \Phi(D_{i-1})}^{\text{'change in complexity'}}$
    $\hat{c}_i$ is much more evened-out compared with $c_i$.
  - But... $\underbrace{\sum_{i=1}^{n} \hat{c}_i}_{\hat{C}} = \underbrace{\sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))}_{C} = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$
    (the telescoping sum)
  - and if $\Phi(D_n) \geq \Phi(D_0)$ then $\hat{C} \geq C \rightarrow \hat{C}$ is a valid upper bound of $C$

Prybg lecture, Amortized analysis, cont. 2017-02-19

- Typically we define $\Phi$ this way so it's useful
- Often for convenience, just set $\Phi(D_0)=0$ & all other $\Phi(D_i) \geq 0$.
- Consider multipop stack again:

| | |
|---|---|
| push | $c_i = 1$ |
| pop | $c_i = 1$ |
| multipop(k) | $c_i = k' = \min(\text{len}(S), k)$ |

Let $\Phi(D)$ = # of items on stack in state $D$; let $\Phi(D_0)=0$ (so $\Phi(D_i)$, $i \geq 0$, always is positive - what we need for legitimate upper bound).

- If $i$ is push then $\Phi(D_i) - \Phi(D_{i-1}) = 1$ and $\hat{c}_i = c_i + \Delta\Phi_i = 2$ here

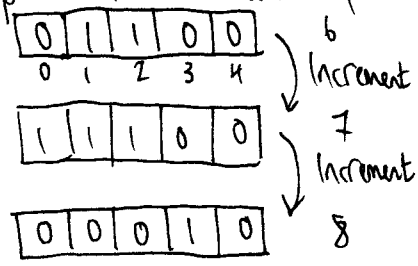
Ignore empty stack

- If $i$ is a pop, then $\Phi(D_i) - \Phi(D_{i-1}) = -1 \rightarrow \hat{c}_i = 0$
- In effect, we "paid" for a pop in advance by doing a push first.
- If $i$ is a multipop, $\Phi(D_i) - \Phi(D_{i-1}) = -k'$, and $c_i = k' \rightarrow \hat{c}_i = 0$ again
- So $\hat{C} = \sum_{i=1}^{n} \hat{c}_i \leq 2n$ regardless of operation.
  $C \leq \hat{C} \leq 2n \rightarrow C = O(n)$. → average cost is constant-time.

Same as in reinforcement learning.

- Note here that we never needed to compute $\Phi(D_i)$ directly - only in its change. This is the case in general: we only want initial value, & deltas.
- Another example: k-bit binary counter
  - k cells, 0 or 1 in each cell; initialized to all 0s.
  - low bit: A[0], high bit: A[k-1] (i.e. $1^{\text{s}}$ place, $2^{k-1}$ place)
  - So total value is $\sum_{i=0}^{k-1} A[i] \cdot 2^i$
  - Only operation: Increment (and reset to 0 if all 1s)

e.g.



Chris Hodapp
chodapp3@gatech.edu

2017-02-19c, CS6505 (Computability, Complexity, & Algorithms)

Pryby lecture, Amortized analysis, cont.

```
def increment(A):
    i = 0
    while i < len(A) and A[i] == 1:
        A[i] = 0
        i += 1
    if i < len(A):
        A[i] = 1
```

Cost (assignments to A) of increment(A) $\leq k$, so cost $= O(k)$

Thus, $n$ increments has total cost $O(nk)$ but this isn't very tight (we don't touch every bit, every time)

Let $t_i$ = #bits set to 0 on $i$th call to increment

then: $c_i \leq t_i + 1$ (count 'while' loop, then final $A[i] = 1$)

$\Phi(D)$ = # bits equalling 1 in state $D$; $\Phi(D_0) = 0$ & $\Phi(D) \geq 0 \ \forall D$

If $\Phi(D_i) = 0$ for $i > 0$ then we must have reset all bits,

so here $\Phi(D_i) - \Phi(D_{i-1}) = 0 - k = -k = -t_i$

If $\Phi(D_i) > 0$ then $\Phi(D_i) = \Phi(D_{i-1}) - t_i + 1$ ← but (we didn't carry) last bit was set to 1; ↑ this many were set to 0

so $\Phi(D_i) - \Phi(D_{i-1}) = -t_i + 1$.

So in both cases, $\Phi(D_i) - \Phi(D_{i-1}) \leq -t_i + 1$

$\hat{c}_i = c_i + \Delta\Phi_i \leq (t_i + 1) + (1 - t_i) = 2 \Rightarrow C \leq \hat{C} \leq 2n \Rightarrow C = O(n)$

So average cost per increment is constant. (Regardless of bit count.)

- Suppose we don't start with an empty data structure.

e.g. $\Phi(D_0) = b_0$, and say $\Phi(D_n) = b_n$.

What if $b_n < b_0$ here? Then $\Phi(D_n) - \Phi(D_0) < 0$ and bounds aren't valid.

But $\hat{C} = C + \Phi(D_n) - \Phi(D_0) \rightarrow C = \hat{C} - \Phi(D_n) + \Phi(D_0) = \hat{C} - b_n + b_0$

$\hat{C} - b_n + b_0$ also $\leq \hat{C} + b_0$, so $C \leq \hat{C} + b_0$ - still an upper bound.

- In our case, as $\hat{C} = 2n$ still:

$C \leq 2n + b_0$ but $b_0 \leq k$ (at most $k$ bits can be 1)

$\rightarrow \boxed{C \leq 2n + k}$

If $k \leq \alpha n$, $C \leq (2 + \alpha)n = O(n)$. (Basically, if $n = \Omega(k)$)

- That is, if the number of operations is asymptotically larger than the number of bits.

2017-03-01

Pryby lectures, graph algorithms

- Why graphs: They're incredibly versatile. Anytime you have a set of objects and relations between pairs of them, you can use a graph. Also: Graph questions are really popular in interviews.
- See: CLRS, Ch. 22 (and appendix B).

$G=(V,E)$

- How do we represent graphs? 2 ways are common:
  - adjacency lists - list of vertices <u>adjacent</u> to each vertex
  - adjacency matrix:
    Number all $n$ vertices and build an $n \times n$ matrix ($n = |V|$).
    Say, $A_{n \times n}$. $A[i][j] = \begin{cases} 1 & \text{if } v_j \in N(v_i), \ N = \text{neighbors} \\ 0 & \text{if not} \end{cases}$

  Note that the above works equally for directed & undirected graphs.
- Adjacency list needs $\Theta(|V|+|E|)$ memory. Each vertex needs a key, and each edge is represented (twice for undirected)
  - But to query if 2 vertices are adjacent is $O(|V|)$ assuming we can look up in constant-time, as we still have to search edges (which can be $\leq |V|-1$ worst-case for a vertex)
- Adjacency matrix is $\Theta(|V|^2)$ memory. $O(1)$ time to compute adjacency.
  - Matrix is symmetric for undirected graph.
- For a <u>sparse</u> graph, $|E| = o(|V|^2)$, that is, asymptotically less than $|V|^2$. (Note that $\binom{|V|}{2} \approx \frac{|V|^2}{2}$ edges are possible)
  In this case, an adjacency list occupies lower asymptotic amount of memory.
- <u>Dense</u> graph: $|E| = \Theta(|V|^2)$. In this case, asymptotic list has no memory advantage over a matrix.
- Both formats extend easily to <u>weighted</u> graphs.
- See also: <u>Spectral graph theory</u> (applying linear algebra to graphs); e.g. eigenvalues can tell you if graph is connected or not.

CLRS 22.2

- 1st algorithm: Breadth-first search. (BFS)
  - "Search" or "graph traversal" algorithms
  - Goal is to reach every vertex in graph in a particular order.
  - In BFS, we reach all neighbors of vertex, then proceed to neighbors of those neighbors, and so on.
  - In DFS (depth-first search), we explore entire path first, backtrack, change path, and so on

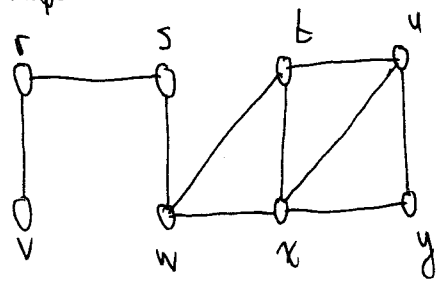
Chris Hodapp
chodapp3@gatech.edu

2017-03-01 (b) CS6505 (Computability, Complexity, & Algorithms)

Pryby lectures, graph algorithms, cont.

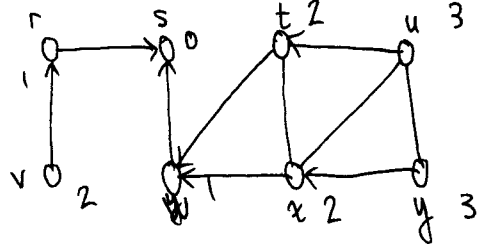
- BFS & DFS both must keep track of vertices already visited, and do so by "coloring" vertices:
  - white = initial color (undiscovered)
  - gray = "discovered" vertex
  - black = "finished" vertex (all neighbors discovered).
- White & black vertices are never adjacent. Gray always sits between.
- We also keep track of predecessor (parent, ancestor) vertex to child/descendant vertex.
  If we turn these to a tree, we get the order in which we searched.
- Example

See algorithm text (~1900 seconds)




- BFS algorithm given: Put start vertex 's' into queue.
  Set 's' gray, all others white.
  - While queue still has elements:
    - Get vertex 'u' from queue.
    - All white vertices adjacent to 'u', color gray & put in queue.
    - Color 'u' black.
- Can also do something in loop above, e.g. set distance at each vertex (i.e minimum # of edges from s). If we encode a predecessor when we set each vertex gray, that gives a tree or, in effect, a path from each vertex back to 's' in minimum steps.

Result:




(#s = distance, arrow is predecessor)

2017-03-01

Pruby lectures, graph algorithms, cont.

- As a byproduct of this we have a minimum spanning tree. (Note "a", not "the". Tree differs because we traverse adjacency list in any order, but distances are always the same.)
- $O(|V|)$ queue operations
  $O(|E|)$ time looking at neighbors
  $\Theta(|V|)$ setup time
  $\} \rightarrow O(|V|+|E|)$
- Tree we build is formally the "predecessor subgraph". It is a theorem that the predecessor subgraph is a tree.
  - Formally:
    $G_p = (V_p, E_p)$, $V_p = \{v \in V : v.pred \neq None \text{ or } v = s\}$,
    $E_p = \{(v.pred, v) : v \in V_p \setminus \{s\}\}$
  - Theorem: $G_p$ consists of precisely those vertices reachable from $s$, and for each $v \in V_p$, there is a unique path from $s$ to $v$ which is also a shortest path from $s$ to $v$ in $G$.
    $\Rightarrow G_p$ is a tree in $G$ (the breadth-first tree).

(CLRS 22.3)

- Next: DFS, Depth-First Search
  - Proceeds until <u>all</u> vertices in $G$ are discovered.
  - Better for discovering subcomponents of graph
  - Predecessor subgraph: $G_p = (V, E_p)$, $E_p = \{(v.pred, v) : v.pred \neq None\}$, is a <u>forest</u> (depth-first forest). It's not guaranteed to be connected, so it can't be a tree, but it's <u>made</u> of depth-first trees.
  - Like BFS, uses colors. It adds a "timestamp" for discovery time and another for finishing time, for each vertex.
    Times are in $1, \ldots, 2|V|$ and are all unique.
    When we color gray, that's discovery time; black, finishing time.
  - Their DFS version is recursive; it calls discover(...) on every vertex, but each discover(...) call calls others.
  - As in DFS, results aren't unique; they depend on order that neighbors are traversed.
  - Running time is, again, $\Theta(|V|+|E|)$.

It could be made iterative, though.

Chris Hodapp
chodapp3@gatech.edu

2017-03-01c, CS6505 (Computability, Complexity, & Algorithms)

Prof by lectures, graph algorithms, cont.

- Interesting properties:
  - Predecessor subgraph of DFS is a forest.
  - Parenthesis Theorem: (interval)
    Let $I_u = [u.d, u.f]$, $I_v = [v.d, v.f]$ where .d = discovery, .f = finishing time
    Then exactly one of the below is true:
    a) $I_u \supset I_v$ & u is an ancestor of v in the DFF
    b) $I_u \subset I_v$ & u is a descendant of v in the DFF
    c) $I_u \cap I_v = \emptyset$ & u, v have no ancestor/descendant relationship
  - White-path theorem: v is descendant of u in DFF iff at time u.d there is a path of white vertices from u to v

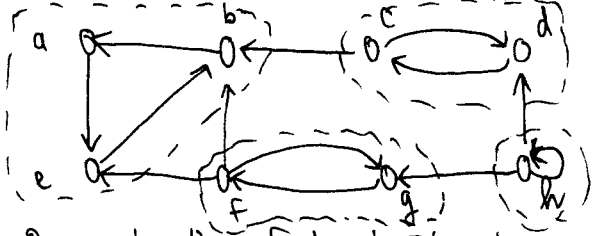
Proofs in the text (Theorems are useful for other proofs too)

- 4 kinds of edges exist in DFF:
  1) tree edges: edges of G in the DFF. (u,v) is a tree edge iff v is discovered by following edge (u,v) from u.
  2) back edges: nontree edges of G (u,v) where u is descendant of v in DFF
  3) forward edges: nontree edges of G (u,v) where u is ancestor of v in DFF.
  4) cross edge: all other edges of G (self loops, e.g. , are always this)

These don't exist in an undirected graph

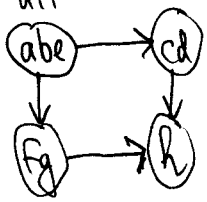
- We can classify edges when we traverse adjacency list in DFS
- DFS is useful for topological sort (CLRS 224), e.g. on a DAG
  - Topo-sort means: Sorting vertices such that u precedes v iff u is reachable from v in graph.
  - We simply sort by finishing time, descending.
- Lemma to prove this: Digraph is acyclic iff a DFS on G yields no back edges.

- Strongly Connected Components (22.5)
  - Very common: Algorithm decomposes graph to strongly connected components, runs operation, recomposes parts back together
  - Definition: If $G=(V,E)$ is a digraph, then $G^T=(V,E^T)$ where $E^T=\{(v,u):(u,v)\in E\}$ (Just flip all the arrows in edges.)




SCCs are circled

- One interesting feature: Strongly-connected components of $G$ & $G^T$ are same.
- See text or ~ 7800 seconds in video for algorithm for this.
  - It involves DFS($G$), then DFS($G^T$) but with main loop considering vertices in topological order (i.e. decreasing finish time)
  - Then, each tree in the DFF is a separate SCC.
- The <u>component graph</u> is what we get if we collapse every SCC into a vertex, and collapse together all the edges between them, e.g. for the graph above:




  - The component graph is a DAG & the algorithm makes use of this.
- Also: If two vertices are mutually reachable, they're in same connected component.
- DFS($G^T$) considering vertices in decreasing finish time is the same as topologically sorting the component graph.

2017-03-06

CLRS, Ch. 23 (Min. Spanning Tree)

- Consider a problem like: Connecting a number of pins on a PCB or ASIC together, but using the minimum amount of wire. We can treat this as an undirected graph where $V$ = pins, $E$ = possible ways to connect them, weight $w(u,v)$ = wire needed to connect $u$ & $v$, then minimizing:

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$ where $T \subseteq E$ is an acyclic subset (therefore a tree).

  - If $T$ connects all vertices, and is acyclic, it is a <u>spanning tree</u>. (Spans all vertices, or spans $G$)
- We can solve in two ways: (Both greedy algorithms)
  Kruskal's algorithm, $\Theta(|E| \log |V|)$
  Prim's algorithm, $\Theta(|E| + |V| \log |V|)$. (Better for $|V| \ll |E|$)
- Both work by growing a minimum spanning tree one edge at a time.
  Loop invariant: Prior to each iteration, $A$ is a subset of some minimum spanning tree. (We find edge $(u,v)$ that can do this such that $A \cup \{(u,v)\}$ is still a subset of some min. spanning tree)

$(u,v)$ = "safe edge"

Chris Hodapp
chodapp3@gatech.edu

2017-03-06(b), CS6505 (Computability, Complexity, & Algorithms)

CLRS, Ch. 23 (cont.)

- How do we find safe edges though?
  - Theorem 23.1:
    - Let $G=(V,E)$ be connected, undirected graph w/ real-valued weights ($w: E \to \mathbb{R}$). $A$ is subset of $E$ included in some min. spanning tree for $G$; let $(S, V-S)$ be any cut of $G$ respecting $A$; let $(u,v)$ be a light edge crossing $(S, V-S)$. Then, $(u,v)$ is safe for $A$.
  - A <u>cut</u> $(S, V-S)$ of $G$ is a partition of $V$.
  - An edge <u>crosses</u> that cut if one endpoint is in $S$, other in $V-S$.
  - A cut <u>respects</u> a set $A$ of edges if no edge in $A$ crosses cut.
  - Edge is a <u>light edge</u> crossing a cut if weight is minimum of any edge crossing cut. (Also: Edge is a light edge satisfying a property if its weight is minimum of all that satisfy that property.)
  - Corollary 23.2: Take $G$ & $A$ as in 23.1. Let $C=(V_C, E_C)$ be a connected component in forest $G_A=(V,A)$. If $(u,v)$ is a light edge connecting $C$ to some other component in $G_A$, then $(u,v)$ is safe for $A$. (Cut $(V_C, V-V_C)$ respects $A$, and $(u,v)$ is a light edge for this cut. $\therefore$ $(u,v)$ is safe for $A$.)
- Section 23.2: The algorithms of Kruskal & Prim
- Kruskal:
  - Set $A$: Forest whose vertices are all those of the given graph.
  - Safe edge added to $A$: Least-weight edge in the graph that connects two distinct components.

Component = tree here?

- Prim:
  - Set $A$: Single tree
  - Safe edge added to $A$: Least-weight edge connecting tree to vertex not in tree

2017-03-07

CLRS Ch. 23 (cont.)

- Kruskal's algorithm relies on a disjoint-set data structure, each set containing vertices in one tree of current forest. Its running time depends on what data structure we use.
  - See MST-Kruskal, pg 631
  - 'for' loop does $O(|E|)$ operations of union & find set, first 'for' loop does $|V|$ operations of making a set, which is $O((|V|+|E|)\alpha(|V|))$, where $\alpha$ is very slowly growing function in 21.4.
  - Since G is connected, $|E| \geq |V| - 1$ - so disjoint-set ops are $O(|E|\alpha(|V|))$.
    $\alpha(|V|) = O(\log |V|) = O(\log |E|) \rightarrow O(|E| \log |E|)$.
    $|E| < |V|^2$, so $\log |E| = O(\log |V|)$, so... $O(|E| \log |V|)$ like we said.
- Prim's algorithm:
  - Similar to Dijkstra's algorithm for finding shortest paths
  - Edges in set A always form a single tree. It starts at an arbitrary root & grows until tree spans all V.
  - Each step adds light edge to A which connects A to isolated vertex. (i.e. vertex for which no edge of A is incident)
  - So, to implement efficiently we need a fast way to select a new edge to add to the tree that A's edges form.
  - It keeps a min-priority queue with a 'key' attribute (the minimum weight of any edge connecting v to a vertex in the tree) for each vertex <u>not</u> in the tree
  - Loosely: Vertices in A define a cut of the graph, and it looks for a light edge crossing that cut (and adds it to A).
  - Running time depends on how we implement min-priority queue.
    - If Q is a binary min-heap (see Ch. 6): can do setup in $O(V)$.
    - While loop runs $|V|$ times, extract-min is $O(\log |V|) \rightarrow O(|V| \log |V|)$.
    - Next 'for' loop runs $O(E)$ times total; we can check membership in Q in constant-time by setting some bit per-vertex & updating when removed from Q. It also has an assignment that is $O(\log |V|)$ with binary heap, so total is $O(V \log V + E \log V) = O(E \log V)$ (Same as Kruskal.)
    - If we use Fibonacci heaps (Ch. 19), extract-min is $O(\log |V|)$, and decrease-key (for assignment) is $O(1)$ amortized $\rightarrow O(E + V \log V)$

Chris Hodapp
chodapp3@gatech.edu

2017-03-07 (b), CS6505 (Computability, Complexity, & Algorithms)

Pryby lectures, graph algorithms, 2 of 2

- This covers CLRS Ch. 23 & 24.
- $G=(V,E)$, undirected & connected & weighted graph.
  - This always has a spanning tree! We're trying to find the minimum-weight one. (MST = minimum spanning tree)
  - The general strategy is to grow set A from $\{\}$ to a spanning tree, one edge at a time.
  - We add a "safe" edge, one we may add to A without violating invariant
  - Another definition of cut: $X, Y \subseteq V$ s.t. $X \cup Y = V$ & $X \cap Y = \emptyset$
    - We're typically interested in edges that cross the cut.
- If $(X,Y)$ is a cut respecting A & e is a light edge crossing $(X,Y)$, e is safe for A.
  - Also: $G_A = (V,A)$ is a forest (it won't necessarily be connected but it will have no cycles). If e is safe for A, e connects distinct components of A.
- We must do this process $|V|-1$ times: every tree has that many edges if it spans $|V|$ vertices
- This leads to a rule: If we find a light edge connecting two connected components, that edge is safe for the first component. (Same as Corollary 23.2)
- See disjoint-set, CLRS 21.1
  - MakeSet(u) - just produce singleton set, $\{u\}$
  - FindSet(u) - return which set u is in
  - union(u,v) - merge $\{u\}, \{v\}$ to $\{u,v\}$ - or set with u & set with v, merged to one set

```
def kruskalMST(G):                    (G is adjacency list & weights)
    A = []
    for u in G.V:                     ] Every vertex sits initially as
        makeSet(u)                    ]   its own set.
    E = edges of G
    sort E ascending by weight
    for e in E:
        u, v = endpoints of e
        if findSet(u) != findSet(v):  ] Are u & v in the same component?
            A.append(e)               ] If not: Merge them & add e to
            union(u, v)               ]   our MST
    return A
```

|V|

O(|E| log |E|)

3|E| total (2 FindSet, 1 union)

- From 21.3 of CLRS:
  - m ops on n elements has $O(m \cdot \alpha(n))$ time.
    $\alpha(n)$ is $\leq 4$ for all "practical" values of n
  - So our amortized time for any operation is $\alpha(n)$.
- Then: $O(|V|\,\alpha(|V|)) + O(|E| \log |E|) + O(3|E|\,\alpha(|V|))$
  but log dominates $\alpha$ ... so:
  $O(|E| \log |E|)$ but $\log |E| < 2 \log |V|$
  so $\boxed{O(|E| \log |V|).}$

That is: Sorting edges is most of the work in Kruskal.

---

- Prim's algorithm:

```
def primMST(G, r):
    A = []
    Q = MinPriorityQueue(G.V)          → That is all keys init to ∞
    Q.decreaseKey(r, 0)                → Root key = 0
    for u in G.V:
        u.pred = None                  → 'pred': what is light edge connecting
        u.inQ = True                       u to tree A?
    while not Q.isEmpty():             → 'inQ': is u in Q? (Gives us constant-time checks)
        u = Q.extractMin()
        u.inQ = False
        if u != r:
            A.append({u, u.pred})
```

Chris Hodapp
chodapp3@gatech.edu

Ryby lectures, graph algorithms 2/2, cont.

2017-03-07c, CS6505 (Computability, Complexity, & Algorithms)

```
for (v,w) in G.adj[u]:
    if v.inQ and w < v.key:
        v.pred = u
        Q.decreasekey(v, w)
return A
```

→ For all neighbors of u still in Q, decrease weight if it is smaller to take (u,v), and use that edge

Note loop invariant: $A = \{\{u, u.pred\} : v \in V \setminus \{r\} \setminus Q\}$

- Also: Vertices already in tree spanned by A are those in $V \setminus Q$ (i.e. in V but not Q)
- $\forall v \in Q$, if v.pred ≠ None, then $v.k < \infty$ and is the weight of a light edge connecting v to the tree

- Running time depends on priority queue implementation
  - If binary heap:
    - Init: $|V| \times$ extractMin $= O(|V| \log |V|)$
    - Inner for loop is, in total, $2|E| \times$ decreasekey $= \boxed{O(|E| \log |V|)}$ which dominates the other two.
  - If a Fibonacci heap, decreasekey is $O(1)$ amortized
    → $O(|V| \log |V| + |E|)$ amortized time
    - This is better than Kruskal (or Prim w/ binary heap) for $|E| = O(|V| \log |V|)$ - a sufficiently sparse graph
- Pairing-based heaps are more common in practice
  - $O(\log |V|)$ amortized for extractMin
  - $o(\log |V|)$ for decrease key (something like $O(2^{2\sqrt{\log \log |V|}})$)
  - May still outperform Kruskal or binary heap for sparse enough graph

These are rare in practice because of large constant factor.

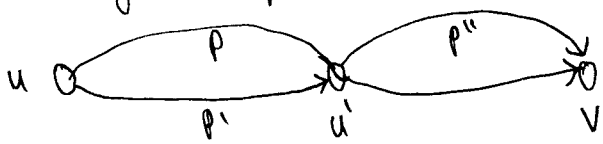
- Next: CLRS Ch. 24, Single-source shortest path
  - This uses digraphs, not undirected
  - Given weighted digraph $G = (V, E)$ & $w: E \to \mathbb{R}$, we want path P in G (note that vertices can't repeat).
  - $w(P) = \sum_{e \in P} w(e)$ - weight of path

'walk' can repeat vertices
'path' cannot

2017-03-07

Pryby lectures, graph algorithms, 2/2 (cont.)

- For $u, v \in V$: $\delta(u,v) = \min\{w(p): p \text{ is path from } u \text{ to } v\}$ or $\infty$ if no path from $u$ to $v$
- We already covered DP & Floyd-Warshall (all-pairs shortest path) which is different, but related.
- What we want here is $\delta(s,u)$ for all $u \in V$, given some source $s$ and/or the actual paths that produced the minimum for each $u \in V$
- These rely on optimal substructure property:




$p$ = shortest path from $u$ to $v$ (with intermediate vertex $u'$)
→ $p'$ ($u$ to $u'$) and $p''$ ($u'$ to $v$) are also shortest paths

- $w(e) < 0$ is okay here - but negative cycles are not (shortest path is not well-defined there). We say $\delta(u,v)$ for those cases $= -\infty$

- Let $G=(V,E)$, and define $u.pred$ for each $u \in V$.
  $G_p = (V_p, E_p)$: $V_p = \{v: v.pred \neq None\} \cup \{s\}$
  $E_p = \{(v.pred, v): v \in V_p\}$
  ] Predecessor subgraph ("shortest-paths tree")

- We use the "relaxation technique":
  - Test whether we can reduce the total cost by going from $s$ to $u$ to $v$, rather than $s$ to $v$ directly, given estimates we have of $u$'s distance & path length.

```
def initializeSSSP(G,s):
    for u in G.V:
        u.d = ∞
        u.pred = None
    s.d = 0
```

] Initially nothing is reachable (hence ∞ & no pred)

→ $s$ can always reach itself, trivially

Chris Hodapp
chodapp3@gatech.edu

2017-03-07d, CS6505 (Computability, Complexity, & Algorithms)

Pryby lectures, graph algorithms, 2/2 (cont.)

```
def relax(u,v,w):
    if v.d > u.d + w:
        v.d = u.d + w
        v.pred = u
```

(Assume adjacency list is used, with weights alongside, i.e. G.adj[i] is list of (j, weight))

- 1st algorithm with this: Bellman-Ford
- This handles negative weights and detects negative cycles

```
def bellmanFordSSSP(G,s):
    initializeSSSP(G,s)
    n = len(G.V)
    for i in range(n-1):
        for u in G.V:
            for (v,w) in G.adj[u]:
                relax(u,v,w)
    for u in G.V:
        for (v,w) in G.adj[u]:
            if v.d > u.d + w:
                return False
    return True
```

|V|-1 passes. Each pass: Look at each vertex. Look at each edge of it, relax that edge.

Check for negative-weight cycles. The part above makes it impossible to ever relax an edge further - unless such a cycle exists. Hence, this just checks if any edge can still be relaxed.

Proof of correctness given in text

Unreachable vertices stay at ∞ & no pred.

- Since 1st 'for' is |V|-1 passes and each looks at every edge, then 2nd 'for' does basically the same
  For one pass: $O(|V||E|)$.
  - That's as bad as $O(|V|^3)$ for a dense graph, but not awful overall.

No cycles = no negative weight cycles

- Interestingly with a DAG a linear-time algorithm exists:

```
def dagSSSP(G,s):
    X = topoSort(G)          - O(|V|+|E|)
    initializeSSSP(G,s)      - O(|V|)
    for u in X:              - O(|E|)
        for (v,w) in G.adj[u]: relax(u,v,w)
```

→ Total $O(|V|+|E|)$

Relax all edges (only one pass)

2017-03-07

ryby lectures, graph algorithms, 2/2 (cont.)

- See text for full correctness proof. It uses the path relaxation property: If $P = (s, v_1, \ldots v_k)$ is shortest path from $s$ to $v_k$ and we relax $P$'s edges in order (even if separated by other relaxations), then $v_k.d = \delta(s, v_k)$ after all the relaxations.
  - This, combined with the fact that our algorithm goes over vertices in topological order, implies we need just 1 pass.
- Last algorithm: Dijkstra's algorithm.
  - Better asymptotic time than Bellman-Ford but edge-weights can't be $< 0$.

```
def dijkstraSSSP(G,s):
    initializeSSSP(G,s)
    S = []
    Q = minPriorityQueue(G.V)        - We use distance estimate as key
    while Q nonempty:
        u = Q.extractMin()           - So at each iteration we get
        S.add(u)                       vertex that is least-far
        for (v,w) in G.adj[u]:         from source
            relax(u,v,w)               (and remove from queue), &
                                       relax all edges from it
```

We look at S later

← Implicit decreaseKey

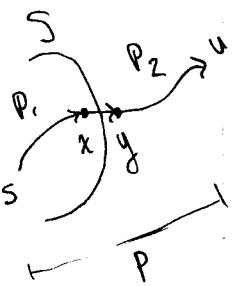
- Running time: $|V|$ inserts
  $|V|$ extractMin
  $O(|E|)$ relaxations → $O(|E|)$ decreaseKeys
  - But, priority-queue implementation matters.
    - Naïve way: Store distance in $v_{th}$ element of array for $v_{th}$ vertex. Then insert is constant-time, decreaseKey is constant-time, but extractMin requires that we traverse array ($O(|V|)$):
      $O(|V| + |V|^2 + |E|) = O(|V|^2)$ - which is just $O(|E|)$ for a dense graph (it's the size of our adjacency list)
    - For a sparser graph we'd want something else.
    - Binary heap: Ignore inserts (we could do all at once), and extractMin & decreaseKey are $\log |V|$, so $O((|V| + |E|) \log |V|)$ which is better if $|E| = o\left(\frac{|V|^2}{\log |V|^2}\right)$.

Chris Hodapp
chodapp3@gatech.edu

2017-03-07e, CS6505 (Computability, Complexity, & Algorithms

Pryby lectures, graph algorithms, 2/2 (cont.)

- Or, use a Fibonacci heap. Then overall: $O(|V| \log |V| + |E|)$.
  (Fibonacci heap was actually developed for Dijkstra's algorithm, due to there being many more decreaseKeys calls.)
- Pairing-based heaps can also improve, and in practice are better.
- Proof is given here because it offers some useful methods
- Loop invariant: At the start of each iteration of the while loop, $v.d = \delta(s,v)$ for each $v \in S = \{\text{vertices already extracted from } Q\}$
  - Proof: Suffices to show $u.d = \delta(s,u)$ when we extract $u$ from $Q$.
  - Init: $S$ starts out empty - so trivially, true for all $v \in S$
  - Maintenance: $\forall v \in S$ already, $v.d = \delta(s,v)$
    For contradiction: $u.d \neq \delta(s,u)$ & is the first vertex for which this is the case.
    $u \neq s$ because $s$ is first vertex added to $S$ & $s.d$ starts out at $0 = \delta(s,s)$.
    Thus, $S \neq \emptyset$ when we add $u$ to $S$, also, there is some path from $s$ to $u$ (otherwise, $\delta(s,u) = \infty$ & $u.d = \infty$ which contradicts $u.d \neq \delta(s,u)$.)
    Say that path, $P$, is shortest path from $s$ to $u$.
    Let $y$ be the first vertex in $P$ not in $S$. (Could be $u$.)
    Let $x$ be verter immediately before $x$ in $P$. (Could be $s$.)
    $P_1$ is path from $s$ to $x$, $P_2$ is $y$ to $u$ - we've split $P$ to $P_1$ & $P_2$. (and edge $x$ to $y$).
    Since $P$ is shortest path, so too must be $P_1$ & $P_2$ (and trivially, $x$ to $y$). $P_2$ might be empty.




  - Claim: $y.d = \delta(s,y)$ when $u$ is added to $S$.
    - $x$ is in $S$, and when we added it to $S$, it must have been true that $x.d = \delta(s,x)$. (Also, $u$ is the First vertex violating this.)
    - We relaxed the edge $(x,y)$. After this, $y.d \leq x.d + w(x,y)$.
      $= \delta(s,x) + w(x,y)$.

- The subpath from $s$ to $y$ must be a shortest path (since $p$ is). Thus, $\delta(s,x) + w(x,y) = \delta(s,y)$.
  But since $y$ is along $p$ and weights are nonnegative,
  $\delta(s,y) \leq \delta(s,u) \leq u.d$ ($\delta(s,u)$ is a lower bound on $u.d$)
  and $y.d \leq \delta(s,y)$
  But we chose $u.d$ before $y.d$ in the priority queue,
  therefore $u.d \leq y.d$. $\Rightarrow$ $y.d = \delta(s,y) = \delta(s,u) = u.d$
  This is a contradiction (we assumed $u.d \neq \delta(s,u)$.)
  $\therefore$ $u.d = \delta(s,u)$

Termination: After last iteration, $Q$ is empty $\Rightarrow$ $S = V$.
Thus, $u.d = \delta(s,u)$ $\forall u \in V$

2017-03-09

CLRS Ch. 24 (Single Source Shortest Paths)

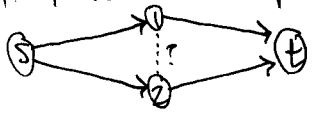
- Edge weights can apply to model pretty much anything that accumulates linearly along a path.
- Note that BFS - which we already covered - handles unweighted graphs.
- SSSP can also solve:
  - Single-destination shortest path (just flip all edges & start at destination)
  - Single-pair shortest path (this is a smaller problem, but it's asymptotically the same as just picking the source & destination you want from a full SSSP solution)
  - All-pairs shortest path (see Ch. 25)
- N.B. Since we can't have negative cycles, & non-negative cycles are pointless, we're concerned with simple paths that are acyclic. Acyclic path has $\leq |V|$ vertices, $\leq |V|-1$ edges, for $G=(V,E)$.
- In the relaxation technique, the distance attribute isn't just an estimate, but an upper bound on distance.
- 24.4: A special case of linear programming can be turned into SSSP and then solved in polynomial time (rather than, say, simplex algorithm, which is not polynomial).

Chris Hodapp
chodapp3@gatech.edu

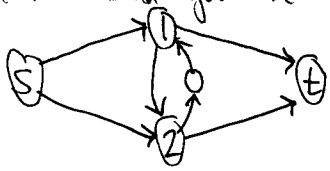
2017-03-10, CS6505 (Computability, Complexity, & Algorithms)

Udacity, L1 (Max Flow)

- "Network" here means nearly anything we can model with a graph, and "flow" is movement from some source to a destination. Typically, we want to maximize that flow.
- Hence: maximum flow.
- Plan:
  1. Flow networks, capacities, residual networks, etc.
  2. Ford-Fulkerson method for finding maximal flow; max-flow min-cut theorem
  3. Scaling algorithms & Edmonds-Karp
- See: Kleinberg & Tardos, Algorithm Design
- Start with flow network, $G=(V,E)$, $(u,v)\in E \Rightarrow (v,u)\notin E$ (no antiparallel edges)
  - Source $s$, sink $t$; all other vertices are internal
  - $\forall v \in V$, $(v,s), (t,v) \notin E$ (no edges go in source or out of sink)
- Capacity: (of a flow network)
  $c: V\times V \to \mathbb{N}\cup\{0\}$; $c(u,v)=0$ if $(u,v)\notin E$
- Flow:
  $f: V\times V \to [0,\infty)$ where:
  1) $f(u,v) \leq c(u,v)$ - flow can never exceed capacity
  2) $f(u,v) f(v,u)$ - one direction must be 0
  3) For all internal vertices: $f^{in}(u) = f^{out}(u)$ ] Flow is conserved except at source/sink
  where $f^{in}(u) = \sum_{v\in V} f(v,u)$, $f^{out}(u) = \sum_{v\in V} f(u,v)$
- Flow value:
  $v(F) = f^{out}(s) = f^{in}(t)$ (i.e. out of source, or into sink)
- We've specified capacities must be integers. If they're rational, we can just multiply by some factor to turn them all to integers.
- Also, we can cope with antiparallel edges. Suppose we have:



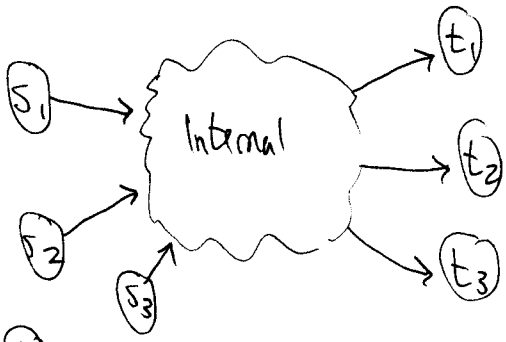

and between 1 & 2 we don't know which way flow should go. We can turn this to:



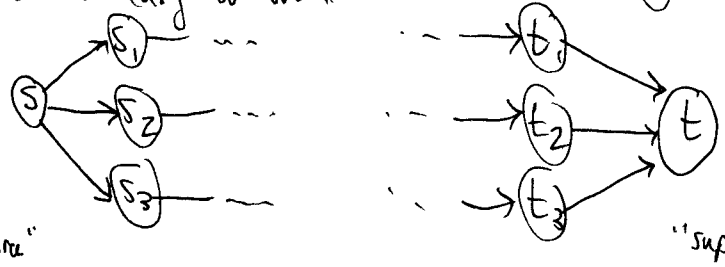
Udacity 11 (cont.)

2017-03-10

- Multiple sources or sinks, e.g.




- This is easy to work around:




"supersource" "supersink"

- Add source & sink tying together all sources and sinks, and give them all capacity of $\infty$.

- So, how do we actually find a maximum flow?
- Typically we take an incremental approach, improving suboptimal flow
- Residual Capacity: Capacity that's not 'used up' in a flow network
  - Given flow f over G:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \\ f(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise} \end{cases}$$

- Can 'send back' this amount (or 'unsend')

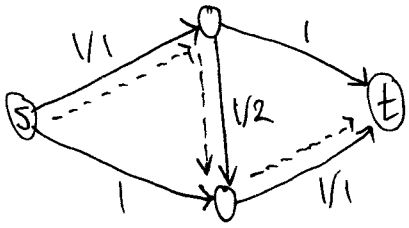
  - Residual graph: $V_f = V$

$(G_F = (V_F, E_F))$ $\quad E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$

  - Usually a sparse graph!

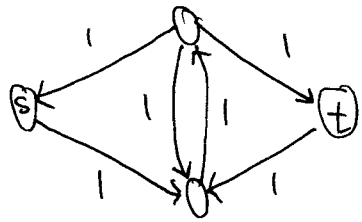
- Example:

f & c over G: (i.e. 1/2 = flow 1, capacity 2)




dotted line = flow network

Residual graph ($c_f$ over $G_f$):



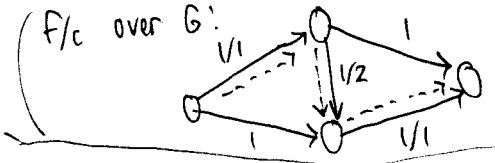

- Note in residual graph that lower left & top right edges stayed the same (we used none of their capacity), but top left & bottom right switched direction (we used all their capacity, but could 'send back' same amount on the flow), and middle edge is now 2 edges (still have capacity 1, but can 'send back' 1 too)

Chris Hodapp
chodapp3@gatech.edu

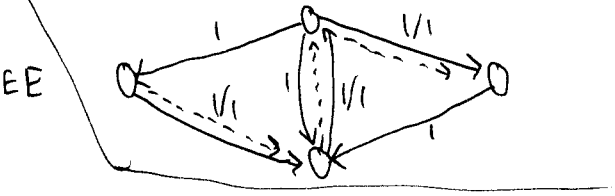
2017-03-10b, CS6505 (Computability, Complexity, & Algorithms)

Udacity 11, cont. (Max Flow)

- Augmentations
  - Often we'll start with a suboptimal flow, then <u>augment</u> it with flow in residual network.
  - Let $f$ be a flow in $G$:

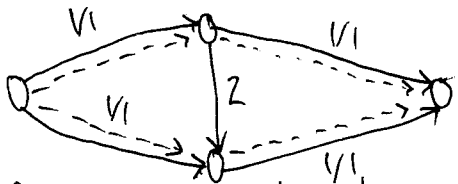



Let $f'$ be a flow in $G_f$:




Let $(f+f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$

$(f+f')/c$ over $G$:




Claim: $(f+f')$ is a flow in $G$ with value $v(f+f') = v(f) + v(f')$

- Ford-Fulkerson
  - MaxFlow Algorithm: (Method, really)
    1. Init $f = 0$ (Start with no flow.)
    2. While a path exists from $s$ to $t$ in $G_f$:
       - Calculate $b = \min_{e \in p} c_f(e)$
       - Set $f = f + f'$, where $f'(u,v) = \begin{cases} b & \text{if } (u,v) \in p \\ 0 & \text{otherwise} \end{cases}$
    3. Return $f$

- Analysis:
  1. $f$ is a flow (0 is a flow, and $f+f'$ is always a flow in $G$)
  2. It must terminate. Each augmenting flow, $f'$, must be $> 0$, but is an integer; once we reduce it to 0 (which must happen for all) then no path exists & it terminates.
  3. $O(|E|)$ time per iteration
- But... Is it actually returning max-flow? We can't augment it, but could some other pattern have done better?
- It terminates, but in what actual time?
- How do we prove that it avoids local optima?
  - Unlike in greedy algorithms, we can't show that it makes no mistakes. Instead we show that it corrects for them.

2017-03-10

Udacity 11. (Cont)

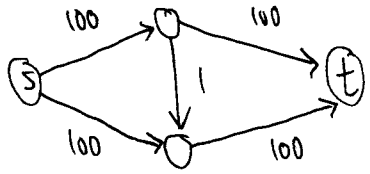
- To do so, we introduce the notion of <u>minimum cut</u>.
- Note that we start at 0 flow & incrementally augment but don't know how high this must go.
- We do know, though, that it can't exceed $\sum_{v \in V} c(s,v)$, i.e. the capacity from the source.
  - Separating s & t forms a <u>cut</u> that limits flow (?) and we may also have other cuts that reduce this further.
  - This reduces the range of possible flows.
- An <u>s-t cut</u> of a flow network is a partition of the vertices $(A,B)$ s.t. $s \in A$, $t \in B$, and $B = V - A$.
  (N.B. Vertices need not be connected)
  - Lemma: Let $f$ be an s-t flow and $(A,B)$ an s-t cut. Then:
    $$v(f) = f^{out}(A) - f^{in}(A) = f^{in}(B) - f^{out}(B)$$
    (Proof by flow conservation is easy.)
  - The <u>capacity</u> of an s-t cut $(A,B)$ is $c(A,B) = \sum_{(u,v) \in A \times B} c(u,v)$ → Capacities between A & B
    - Lemma: Let $f$ be a flow and $(A,B)$ be an s-t cut in a flow network. Then $v(f) \leq c(A,B)$ - pretty intuitive.
- Max-Flow Min-Cut Theorem:
  - The following are equivalent:
    1. $f$ is a maximum flow in $G$.
    2. $G_f$ has no augmenting paths
    3. there exists an s-t cut $(A,B)$ such that $v(f) = c(A,B)$
  - Corollary: Ford-Fulkerson algorithm returns a maximum flow
  - Proof of (1) ⇒ (2): Suppose not. Let $f'$ be an augmenting flow. Then $f + f'$ is a flow and $v(f+f') = v(f) + v(f') > v(f)$ - a contradiction since (1) tells us $f$ is a maximum.
  - Proof of (2) ⇒ (3): Let $A$ be the set of vertices reachable in $G_f$ from $s$; $B = V - A$.
    $(u,v) \in A \times B \Rightarrow f(u,v) = c(u,v)$ - since $v \in B$, $u$ is reachable from $s$ but $v$ is not, thus, capacity must be saturated.
    $(u,v) \notin B \times A \Rightarrow f(u,v) = 0$ (any 'backwards' edge must be unused) ?

Chris Hodapp
chodapp3@gatech.edu

2017-03-10c, CS6505 (Computability, Complexity, & Algorithms)

Udacity 11, cont. (Max Flow)

- Recall $v(f) = f^{out}(A) - f^{in}(A)$ but $f^{in}(A) = 0$ (there is no flow back into A) so $v(f) = f^{out}(A) = \sum_{(u,v) \in A \times B} c(u,v) = c(A,B)$
- Proof of ③ ⇒ ①:
  - If not true, then cut capacity wouldn't be an upper bound (contradiction).
    - That is, since $v(f) = c(A,B)$, $v'(f) > c(A,B)$ violates that definition so no $v'(f) > c(A,B)$, $v'(f) > v(f)$ exists.
- But...




- Floyd-Fulkerson can use up to 200 augmenting paths if we pick less optimal paths

- Certain augmentations are preferable. Key idea: Prefer heavier paths.
- Scaling Algorithm:
  1. Init $F = 0$, init $\Delta = \max_{v \in V} c(s,v)$
  2. While $\Delta \geq 1$:
     While $\exists$ path $p$ in $G_f(\Delta)$
     - use $p$ to augment $F$

     Set $\Delta = \Delta / 2$
  3. Return $F$

→ $G_f(\Delta) = (V, \{(u,v) : c_f(u,v) \geq \Delta\})$ ($G_f(1) = G_f$)

- Analysis:
  $O(\log(\max_{v \in V} c(s,v)))$ outer loop iterations (can only halve $\Delta$ that much)
  Again $O(|E|)$ in inner loop iterations
- But how many inner loop iterations?
- Claim: The maximum number of iterations for a scaling phase is $\leq 2|E|$
  - Lemma: If $G_f(\Delta)$ has no s-t paths, then $\exists$ s-t cut $(A,B)$ s.t. $c(A,B) \leq v(f) + |E|(\Delta - 1)$

Proved in lecture

- Proof: Base case $\Delta = C$ is trivial.
  Let $f$ be the flow after the scaling phase $\Delta$. Let $g$ be the flow after the previous scaling ($2\Delta$ or $2\Delta + 1$).
  $v(f) \leq v(f^*) \leq \underbrace{c(A,B) \leq v(g) + |E|(2\Delta)}_{\text{By our lemma}}$

$v(f^*)$ = value of max flow

2017-03-11

Udacity 11 (cont.)

Let $k$ be the number of iterations

$k\Delta \leq v(f) - v(g) \leq 2|E|\Delta \Rightarrow k \leq 2|E|$

↑ Each iteration increases the flow by at least $\Delta$ (thus a lower bound)

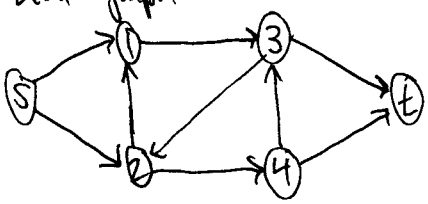
$\Rightarrow$ Scaling algorithm returns a maximum flow in $O(|E|^2 \log C)$ where $C = \max_{v \in V} c(s,v)$

- Preferring shorter paths (not heavier ones) also is viable.
- Edmonds-Karp
  1. Init $F=0$
  2. While $\exists$ s-t path in $G_f$ ] $O(|E||V|)$ iterations
     - Find a shortest s-t path $p$
     - Let $b = \min_{(u,v) \in p} c_f(u,v)$ } Again $O(|E|)$ time
     - Augment $f$ by $b$ over $p$
  3. Return $f$

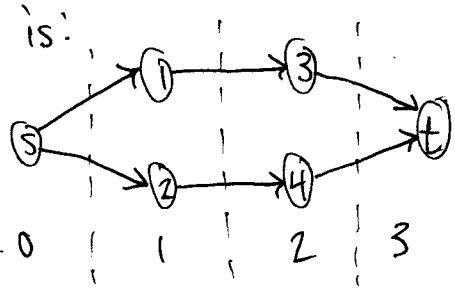
→ Returns max-flow in $O(|E|^2|V|)$ time.

- Level graph: Let $\ell(v)$ be the shortest path distance from $s$ to $v$ in $G$. The level graph of $G$ is the subgraph with edges $(u,v)$ s.t. $\ell(v) = \ell(u)+1$

e.g. Level graph of:




is:




- Observations:
  - Augmenting a shorter path only creates a longer one.
  - Shortest path distance must increase every $|E|$ iterations. Every time we augment a path we remove an edge from the level graph (the edge whose capacity we saturated).
  - Only $|V|$ possible shortest path lengths.
- Note:
  - Now "strongly polynomial" (?)
  - Capacities now can be things besides integers (we've eliminated the dependence on them)

Chris Hodapp
chodapp3@gatech.edu

2017-03-11(b), CS6505 (Computability, Complexity, & Algorithms)

Udacity 11 (cont.)

- One more refinement: Dinic's Algorithm
  - His insight: Shortest path computation can be recycled
  1. Init $f=0$
  2. Repeat:
     - Build level graph $L$ from $G_f$, and let $k$ be the shortest path length
     - While $\exists$ positive s-t path $p$ in $L$ of length $k$: augment $f$ by $p$
     - until no more s-t paths exist in $G_f$
  3. Return $f$

$O(|V|)$ phases

$O(|E||V|)$ phases

Note this actually builds a level graph, not just uses one for proof (Do BFS save only forward edges.)

But in step 2 we need not rebuild level graph every time; we may simply delete a 'blocking vertex' from old one.

~~$O(|E|^2|V|)$~~

$\Rightarrow O(|E|^2|V|)$, an improvement over Edmonds-Karp.

(See CLRS 26.4 & 26.5 ?)

- See also: "push-relabel" algorithms which perform best in practice versus algorithms based on augmenting flow.

2017-03-12

CLRS 26 (Maximum Flow)

- Notes on flow networks:
  - Intuitively, the source produces something at the exact same rate the sink consumes it.
  - We typically assume a connected network.
  - Most of the rules of flow follow from intuition.
- "A minimum cut of a network is a cut whose capacity is minimum over all cuts of the network."
- Maximum flow = capacity of minimum cut

See Sipser Ch. 3 - 4

2017-03-22

Udacity 1, Languages & Computability

- Functions
  - Relation: Set of ordered pairs. (Each: (domain, range).)
  - Function is special type of relation: Each element of domain is paired with exactly one element of the range.
- At least... we generally start there, but then look at examples like $f(x) = 3x + 2$ and understand it more as "what you do to $x$ to get $f(x)$". This is more a <u>Computability</u> definition, and it differs from the set theory definition above.
- First, before we ask what's computable, what is computation?
  - Say that a computer turns input to output...
  - <u>Alphabet</u>: Finite set of symbols, e.g. $\Sigma = \{0, 1\}$, $\Gamma = \{A, C, G, T\}$
  - <u>String</u>: Finite sequence of symbols over some alphabet, e.g. $s = 1010$, $x = AACAG$, $y = \epsilon$ (empty string)
  - If we say that input is a string and output is just one bit, we can refer to all strings the computer accepts
  - <u>Language</u>: a set of strings (any set) over some alphabet
  - Usual set ops apply - union, intersection, complement
    - Complement is over all else that the alphabet allows
  - Also concatenation: $AB = \{xy \mid x \in A \text{ and } y \in B\}$
    e.g. if $A = \{0, 10\}$, $B = \{0, 11\}$, $AB = \{00, 011, 100, 1011\}$
    We typically write $A^k$ for A concatenated k times. $A^0 = \{\epsilon\}$ too.

Note similarity to regexps

- If we want any number of concatenations, use Kleene star: $A^* = \bigcup_{i=0}^{\infty} A^i$ or Kleene plus to exclude empty string: $A^+ = \bigcup_{i=1}^{\infty} A^i$.
  - Note that this is still only finite-length strings (finite but unbounded)
- Some functions aren't computable; we show countable vs. uncountable for this.
  - A set is countable if we can number every element in it <u>and</u> eventually cover the entire set. With even numbers, it's easy; with rationals, a little more difficult. With reals, it's impossible.
- Formally: A set is countable if it is finite or there is a one-to-one correspondence with the natural numbers.
- Theorem: For any finite alphabet $\Sigma$ the set of strings $\Sigma^*$ is countable.

Chris Hodapp
chodapp3@gatech.edu

2017-03-22(b), CS6505 (Computability, Complexity, & Algorithms)

Udacity 1, Languages & Countability (cont.)

- Proof: Recall $\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i$
  Let $N_k$ = size of set of all strings of size $\leq k = |\Sigma^0| + \ldots + |\Sigma^k|$
  Then assign numbers:
  $\{1\} \longleftrightarrow \Sigma^0$
  $\{N_0+1, \ldots N_1\} \longleftrightarrow \Sigma^1$
  $\{N_{k-1}+1, \ldots, N_k\} \longleftrightarrow \Sigma^k$ ...
- e.g. if $\Sigma = \{0,1\}$ (i.e. binary):

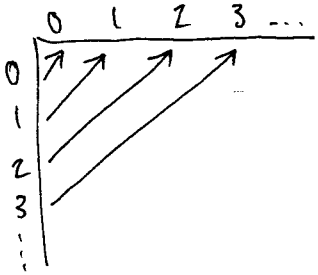
| $\varepsilon$ | 0 | 1 | 00 | 01 | 10 | 11 | 000 | ... |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | | ... |
| Empty | Length 1 | | Length 2 | | | | | |

- Same argument: Countable union of finite sets is countable.
  - Actually can replace 'Finite' with 'Countable'
  - Suppose $S_0, S_1, \ldots$ are disjoint (WLOG) and $S_k = \{x_{k0}, x_{k1}, \ldots\}$
  - Then make a grid:

| | 0 | 1 | 2 | 3 | ... |
|---|---|---|---|---|---|
| 0 | $x_{00}$ | $x_{01}$ | $x_{02}$ | $x_{03}$ | ... |
| 1 | $x_{10}$ | $x_{11}$ | $x_{12}$ | $x_{13}$ | ... |
| 2 | $x_{20}$ | $x_{21}$ | $x_{22}$ | $x_{23}$ | ... |
| 3 | $x_{30}$ | $x_{31}$ | $x_{32}$ | $x_{33}$ | ... |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋱ |

Note that row $k$ is elements of $S_k$.

  - We can't go row-by-row as we could with a finite set (we'd never 'finish' the row and get to the next. But we can do...




  - Each diagonal is finite.
  - A union of these sets is also a union of the diagonals!

$$\bigcup_{i=0}^{\infty} S_i = \bigcup_{k=0} \{x_{ij} \mid i+j=k\}$$

  - then just enumerate based on indices.
- Same argument can show that rationals are countable.

2017-03-22

Udacity 1, cont.

- Theorem: The set of languages over a finite alphabet is uncountable.
- Proof: Suppose not. Let $L_1, L_2, \ldots$ be the set of languages over an alphabet $\Sigma$. Let $x_1, x_2, \ldots$ be the strings in $\Sigma^*$.
  Build a table where rows correspond to the languages, and the columns correspond to the strings $(x_1, x_2, \ldots)$.
  Put a 1 in the table if the string is in the language, otherwise 0.
  Now consider $\{x_i \mid x_i \notin L_i\}$, a language of strings not in $L_i$.
  Note that all we've done is look along this table's diagonal, and reverse it. This language must be $L_k$ for some $k$.

"diagonalization trick"

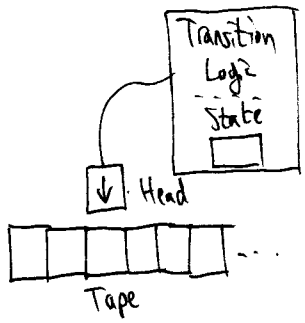
  But is $x_k$ in $L_k$?
  If $x_k \in L_k$ then that value along the diagonal must be 1 - but if that's the case, being in $L_k$ means $x_k \notin L_k$.
  If $x_k \notin L_k$ then that diagonal's same value must be 0 - but then by $L_k$'s definition $x_k \in L_k$.
  Thus, contradiction.
- Consequences
  - Let $\Sigma$ be the set of ASCII characters.
    $\{w \mid w$ represents a valid Python (or whatever) program$\} \subseteq \Sigma^*$
    Thus, set of valid Python programs is countable.
  - But for any language $L$, let $F_L(x) = \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{otherwise} \end{cases}$
    All such functions are distinct (?) so
    $\{F_L \mid L$ is a language over $\Sigma\}$ is uncountable. Thus, the set of functions from $\Sigma^*$ to $\{0,1\}$ is uncountable.
    - But if the programs are countable, this must mean that functions exist (uncountably many of them) that cannot be computed.

Chris Hodapp
chodapp3@gatech.edu

2017-03-22c, CS6505 (Computability, Complexity, & Algorithms)

Udacity 2, Turing Machines

- Notation on Turing Machine:
  1. Finite set of states $Q$
  2. Input alphabet $\Sigma$ ($\sqcup$ not allowed - blank symbol)
  3. Tape alphabet $\Gamma$ (includes $\sqcup$)
  4. Transition Function:
     $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$
     Input — Output & transition of tape




- Tape has a start but no end.
  5. Start state $q_0$
  6. Accept state $q_{accept}$
  7. Reject state $q_{reject}$

- Example: Testing oddness

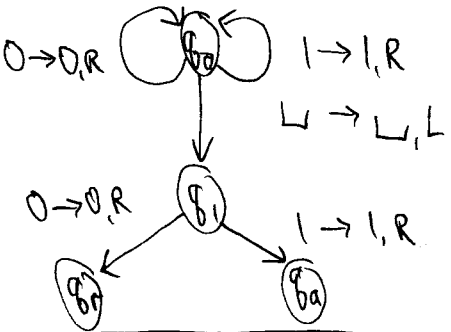
$Q = \{q_0, q_1, q_a, q_r\}$ ($q_0$: start, $q_a$: accept, $q_r$: reject)

$\Sigma = \{0, 1\}$

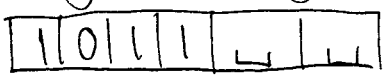
$\Gamma = \{0, 1, \sqcup\}$

$$\delta = \begin{cases} q_0, 0 \rightarrow q_0, 0, R \\ q_0, 1 \rightarrow q_0, 1, R \\ q_0, \sqcup \rightarrow q_1, \sqcup, L \\ q_1, 0 \rightarrow q_r, 0, R \\ q_1, 1 \rightarrow q_a, 1, R \end{cases}$$

- Or:




- Input:




- We eventually end up in $q_a$.

2017-03-22

Udacity 2 (cont.)

- A "configuration" is a triple of state, tape content, & head position.
  - A computation is a sequence of configurations.
- Notation like $q_0 101$ is: content to left of head (blank here), state, right of head, so odd checking goes like:
  $q_0 101$, $1 q_0 01$, $10 q_0 1$, $101 q_0 \sqcup$, $10 q_1 1$, $101 q_a \sqcup$
- See 'Equality Testing' example in Udacity
- Language Deciders
  - A TM decides a language L iff it accepts every $x \in L$ and it rejects every $x \notin L$
  - But...


This does not decide L, $L = \{w \in \{0,1\}^* \mid w \text{ contains a } 1\}$, because it never terminates on a string without a 1.

- But a TM recognizes a language L iff it accepts every $x \in L$ and does not accept any $x \notin L$. (Above example does recognize L.)
- Define $L(M) = \{x \mid M \text{ accepts } x\}$ as the language of machine M.

---

Sipser, Ch. 3

- Church-Turing Thesis
- Turing Machine
  - Infinite tape for unlimited memory; tape head that can read & write symbols and move around.
  - Tape initially contains input and elsewhere is blank.
  - 'accept' & 'reject' are designated halting states
- p126: Example of matching $B = \{w\#w \mid w \in \{0,1\}^*\}$ (formal: p132)
- Configuration $C_1$ 'yields' $C_2$ if TM can go from $C_1$ to $C_2$ in single step
- Turing-recognizable = recursively enumerable language
- Turing-decidable = recursive language
- p131: Recognizing $A = \{0^{2^n} \mid n \geq 0\}$
- p134: Deciding $C = \{a^i b^j c^k \mid i \times j = k \ \& \ i,j,k \geq 1\}$

Chris Hodapp
chodapp3@gatech.edu

2017-03-24, CS6505 (Computability, Complexity, & Algorithms)

Sipser, Ch. 3 (cont.)

I have a different edition of Sipser than the homework refers to...

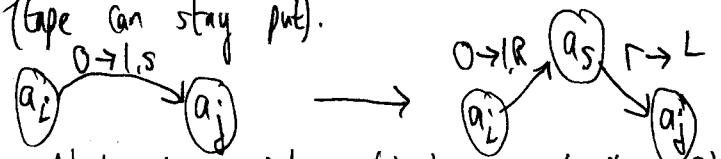
- TM Variants
  - TM is a very robust model, in that for almost any change, we can modify it to an equivalent 'normal' TM
  - To show equivalence, we need only show we can simulate one by the other.
  - e.g. multitape Turing machine
    - Several tapes, each with a head for reading & writing
    - We can simulate on a normal TM:
      - Make a new symbol - say, # - which sits in between the contents of each tape, which we've put onto one tape.
      - Make dotted symbols for each, and let a dotted symbol represent the head position of the respective tape.
      - Shift entire tape to the right if some head moves past a #.
  - Nondeterministic TM: $\delta: Q \times \Gamma \to P(Q \times \Gamma \times \{L, R\})$
    - A normal TM may simulate this by trying every possible branch of a given state (via BFS - not DFS)
- The question of solvability of some problems had to wait until the modern conception of 'algorithm' existed
- Church-Turing thesis is Church's $\lambda$-calculus & TMs are equivalent
- Consider $D = \{p \mid p$ is a polynomial with an integral root$\}$ (Hilbert's Tenth Problem); D is not decidable, but is Turing-recognizable. We can just check integers in order (alternating positive & negative), then we'll always find a root if one exists, but will never halt if none do.
  - See Matijasevič's theorem

~~2017-03-21~~ 2017-04-01

~~Sipser Ch. 3 (cont.)~~

Udacity lesson 3 (Church-Turing Thesis)

- We covered Church-Turing Thesis (that everything computable can be computed by a TM) last lesson.
- Church had an alternate method of computation, the $\lambda$-calculus, which is equivalent.
- This is a <u>thesis</u>, not a theorem, because it's not something we could prove or disprove.
- TM $\leftrightarrow$ Multi-tape TM $\leftrightarrow$ Random access memory model
- A language is "computable" iff a TM can compute it.
- Simulating machines:
  - Example: <u>Stay-Put Machines</u> have $\{L, R, S\}$ movement rather than just $\{L, R\}$ (tape can stay put).
  - Equivalent because...




For any transition that stays put. (Just move L then R)

- Equivalent because:
  1. Accept, reject, & loop on same inputs
  2. Have same tape contents on halt (when relevant)
- Note that other properties - # of states, tape alphabet, # steps - don't matter for equivalence

→ Re-read this in text

- Multitape TM has $k$ tapes & $\delta: Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k$
  - That is, transition is based on contents of all $k$ tapes and can write & read all $k$.
  - We can simulate this with one tape by copying all inputs & separating
- Another variant: We can let cells of tape be written once; we can make it doubly-infinite; more importantly to intuition on computation, we can show that a RAM/register model is equivalent.
  - eg. registers $R_0, R_1, R_2, R_3$, RAM storage $T[1], T[2], \ldots$ program itself is stored separately (as a series of instructions) and Program Counter register stores position there.
  - We have a 'read' & 'write' instruction which read/write to/from address $j \leftrightarrow R_0$; store/load; jump/halt which modify program counter
  - Final value of $R_0$ determines accept/reject (at halt)

Chris Hodapp
chodapp3@gatech.edu

2017-04-01 (b), CS6505 (Computability, Complexity, & Algorithms)

Udacity 3 (Church-Turing)

- Then random access TM consists of:
  1. Number of registers $k \in \mathbb{N}$
  2. Program $\Pi = (\pi_1, \pi_2, \ldots \pi_p)$ where each $\pi_i$ is instruction
- Configuration consists of:
  1. Program counter value $C \in \{0 \ldots p\}$
  2. Register values $R_0, \ldots R_{k-1} \in \mathbb{N}_0$
  3. RAM contents $T: \mathbb{N}_1 \to \mathbb{N}_0$
     (Since we can only have reached a finite number of addresses this can have only a finite representation)
- RAM & TM are equivalent
  - Its easier to show that a multitape TM (One tape per register, one each for scratch, RAM, and I/O). is equivalent
- There are still problems that a TM <u>cannot</u> solve. Why we care: If a TM can't solve it, no currently implemented computer (or implementable, really) can either.

Udacity 4 (Universality)

- Turing realized that the program itself could be viewed as data for other computation.
- How would we encode a TM for another TM to simulate it?
  - Let $M$ be a TM with states $Q = \{q_0, \ldots q_{n-1}\}$ & tape alphabet $\Gamma = \{a_1, \ldots a_m\}$. Define $i$ & $j$ such that $2^i \geq |Q|$ and $2^j \geq |\Gamma|$. Encode $q_k$ as the string $qw$, where $w$ is the binary representation of $k$; eg. if $|Q| = 6$ then $q_3$ is $q011$. Encode $a_k$ as the string $aw$, where $w$ is the binary rep. of $k$. eg. if $|\Gamma| = 10$ then encode $a_5 = *$ as $a0101$.
- Universal Turing Machine.
  - Task: Given input string $\langle w \rangle \# \langle M \rangle$, loop, accept, or reject as $M$ would on $W$.
    $\langle M \rangle$ is stored like (Input state, input symbol, output state, output symbol, direction)
    eg. $(q011, a01, q100, a10, R)$.

2017-04-01

Udacity 4 (Universality)

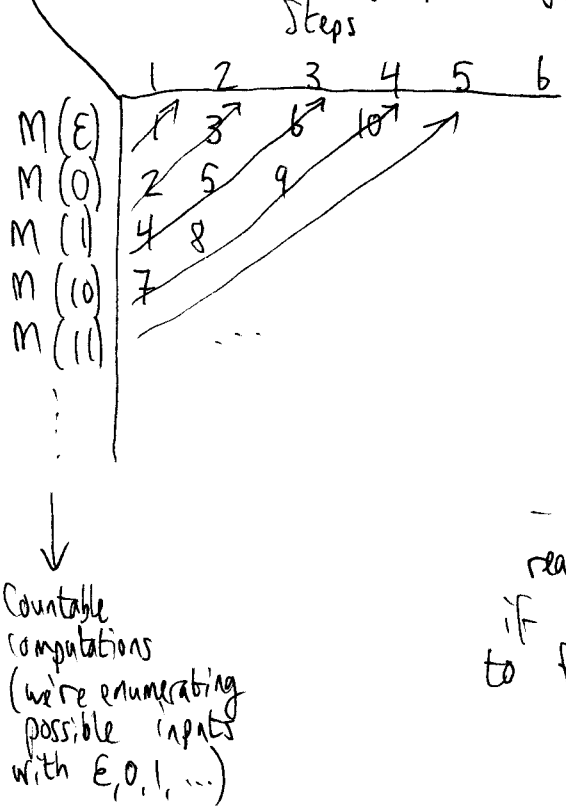
1. Copy ⟨M⟩ to 2nd tape. Copy initial state to 3rd tape. Rewind.
2. Search for correct tuple in 2nd tape. (Find state transition?) Halt if no match found. Otherwise: Apply change.
3. Repeat 2.

- Thus: TMs are reprogrammable since we can pass a TM as input to the universal machine above, and it can simulate them.
- This suggests that programs are just data.
  So... Data can be interpreted as a program (possibly-invalid)
- However, at this point we ignore the <u>specifics</u> of TMs and focus on the higher level.
- Recall:
  TM <u>recognizes</u> language $L$ iff it accepts every $x \in L$, doesn't accept any $x \notin L$. TM <u>decides</u> $L$ if it accepts every $x \in L$ & <u>rejects</u> every $x \notin L$.
  - We aren't interested here in whether a TM recognizes/decides a language $L$, but rather whether one exists that does.
  - Language is recognizable/decidable if a TM exists that recognizes/decides it. Decidable implies recognizable.
  - Decidable = computable = recursive
  - Recognizable = recursively-enumerable = Turing-acceptable = semi-decidable (different notation seen in different contexts)
    - If $L$ is recognizable, and its complement also is, then the language is decidable (but with pitfalls!)
- Suppose $M_1$ recognizes $L$, $M_2$ recognizes $\bar{L}$ (complement). We can run both $M_1$ & $M_2$, but alternate between them with some limited number of steps in each, waiting for $M_1$ or $M_2$ to accept.
  A string must be in either $L$ or $\bar{L}$, so either $M_1$ or $M_2$ must recognize it in finite steps.
- <u>Dovetailing</u> trick: Run a countable set of computations at once.
  - similar to diagonalization

Chris Hodapp
chodapp3@gatech.edu

2017-04-01c, CS6505 (Computability, Complexity, & Algorithms)

Udacity 4 (Universality, cont.)




- Thus we always run every computation for an increasing number of steps, but still finite.
- We can halt as soon as any computation accepts.
- We're guaranteed to eventually reach an answer with this method if some computation is guaranteed to finish

Sipser, Ch.4 (Decidability)

- It is perhaps strange to speak of what <u>cannot</u> be solved with an algorithm when our main concern is how to solve things, but it's important to know the limits. For instance: We may run into problems that need solving but which can't be solved algorithmically without being simplified or altered.
- Decidable languages
  - We start at languages because some terminology for this is already developed; e.g. the <u>acceptance problem</u> for DFAs - testing whether a particular finite automaton accepts a given string - can be expressed as a language $A_{DFA}$:
    $A_{DFA} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$.
    Then: $\langle B, w\rangle$ is member of $A_{DFA} \leftrightarrow$ DFA B accepts input w.
  - Theorem 4.1: $A_{DFA}$ is decidable.
  - The same applies to NFA (nondeterministic finite automata) and to regular expressions.

2017-04-01

Sipser, Ch. 4 (cont.)

- $E_{DFA} = \{\langle A \rangle \mid A \text{ is a DFA } \& \ L(A) = \emptyset\}$ ] Decidable (Emptiness testing)
  - Does the DFA accept any strings at all?
- $EQ_{DFA} = \{\langle A, B \rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$
  - Whether two languages recognize same language (decidable)
- $A_{CFG} = \{\langle G, w \rangle \mid G \text{ is a CFG that generates string } w\}$
  - Decidable
- $E_{CFG}$ is also decidable (analogue of $E_{DFA}$)
- Theorem 4.8: Every context-free language is decidable.
- Halting Problem
  - The general problem of software verification reduces to this.
  - By analogy with the earlier examples: $A_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM \& accepts } w\}$ (does a TM accept some input string?) → Undecidable
    However $A_{TM}$ is Turing-recognizable. (Recognizers are more powerful than deciders.)
  - The halting problem is concerned with whether we can detect programs that don't halt.
- Proof is by diagonalization (from Cantor who observed that sets that could be put into 1:1 correspondence had same cardinality).
  - $f: A \to B$ (or bijective)
  - f is onto if it hits every element of B (or surjective)
  - f is a correspondence if both one-to-one & onto (every element of A maps to a unique element of B, and likewise for B to A)
- A & B are same size if we can find a correspondence.
  - A is countable if it is finite, or has same size as $\mathbb{N}$.
  - Diagonalization shows that set of rationals is countable too
- Uncountable sets can't be put to correspondence with $\mathbb{N}$. (e.g. $\mathbb{R}$).
- Each TM recognizes a single language
- Set of languages is uncountable, but TMs are countable - thus some languages aren't Turing-recognizable.
- Note that $\mathcal{B}$, the set of infinite binary sequences, is uncountable. We can show uncountability by 1:1 correspondence with $\mathcal{B}$.
  - e.g. we can show every language has a characteristic sequence in $\mathcal{B}$; if $\Sigma^* = \{s_1, s_2, s_3, \ldots\}$ then $i$'th bit of $L$'s char. seq. is 1 if $s_i \in A$ & 0 if $s_i \notin A$.

Chris Hodapp
chodapp3@gatech.edu

2017-04-02, CS6505 (Computability, Complexity, & Algorithms)

Sipser, Ch. 4 (cnt.)

- Sketch of why $A_{TM}$ is undecidable:
  Assume $A_{TM}$ is decidable. Call the decider $H$, that is, for TM '$M$' and input $w$, say that $H$ accepts & halts if $M$ accepts $w$, and $H$ rejects & halts if $M$ fails to accept $w$. $H$ is a TM, also.
  - Now construct $D$, a TM which incorporates $H$.
    $D$ calls $H$ using $w = \langle M \rangle$ - that is, $M$'s own description - and returns the opposite (i.e. if $H$ accepts, $D$ rejects, & vice versa).
  - What if we then call $D$ with input $\langle D \rangle$?
  - Note that... $D(\langle M \rangle) = \begin{cases} \text{accept} & \text{if } M \text{ does not accept } \langle M \rangle \\ \text{reject} & \text{if } M \text{ accepts } \langle M \rangle \end{cases}$
  - So... $D(\langle D \rangle) = \begin{cases} \text{accept} & \text{if } D \text{ does not accept } \langle D \rangle \\ \text{reject} & \text{if } D \text{ accepts } \langle D \rangle \end{cases}$
  - This is a contradiction, so neither $D$ nor $H$ can exist.
- It's not immediately obvious but this is a diagonalization proof.
- Look at describing a machine like:

| | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ ... |
|---|---|---|---|
| $M_1$ | accept | reject | reject ... |
| $M_2$ | accept | accept | accept ... |
| $M_3$ | reject | reject | reject ... |
| ⋮ | ⋮ | ⋮ | ⋱ |

  - The left might describe $H$ using input $\langle M_i, \langle M_j \rangle \rangle$ for row $i$, col. $j$.
- Now we build $D$ by reversing the diagonal of this table (where input is $M_i$ & $\langle M_i \rangle$.) But $D$ must also be in this table since it's a TM.
- $A_{TM}$ is not Turing-decidable, but is Turing-recognizable. Turing-unrecognizable languages exist too though.
- For any undecidable language, either it or its complement are not Turing-recognizable.
  - Co-Turing-recognizable = complement is Turing-recognizable
  - Turing-recognizable & co-Turing-recognizable $\leftrightarrow$ Turing decidable

2017-04-03

Sipser, Ch. 5 (Reducibility)

- Ch. 4 established TM as a model of general computation and showed several things solvable on it - and one (the halting problem) that is unsolvable.
- This chapter introduces the primary method for proving unsolvability: reducibility.
- Reduction: Way of converting one problem to another such that solving second problem can solve first one.
- If A reduces to B, we can use solution to B to solve A.
  - eg. Problem of measuring area of rectangle reduces to measuring length & width. Problem of solving linear equation reduces to problem of inverting a matrix.
- Reducibility helps with decidability, but also complexity theory
- If A is reducible to B & B is decidable, so is A.
- If A is undecidable & reducible to B, B is undecidable.
- From language theory:
  $A_{TM}$ relates to $HALT_{TM}$, the problem of determining whether a TM halts on a given input (halting problem)
  - We can prove that $HALT_{TM}$ is undecidable by showing that $A_{TM}$ is reducible to $HALT_{TM}$ & $A_{TM}$ is undecidable.
- Consider $E_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$
  - This can be proven undecidable by a similar method - show that decidability of $E_{TM}$ implies decidability of $A_{TM}$.
- Rice's theorem: Determining any property of the languages recognized by Turing Machines is undecidable.
- $EQ_{TM} = \{\langle M_1, M_2 \rangle \mid M_1 \text{ & } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$ - that is, equality of two TMs - is also undecidable.
  (solving $EQ_{TM}$ lets you solve $E_{TM}$)
- Computation history: Sequence of configurations that the machine goes through as it processes the input; complete record of computation for an input.

Chris Hodapp
chodapp3@gatech.edu

2017-04-03(b), CS6505 (Computability, Complexity, & Algorithms)

Sipser, Ch. 5 (Reducibility), cont.

- If $M$ is a TM & $w$ an input string, an <u>accepting computation history</u> is seq. of configurations $C_1 C_2 \dots C_\ell$ where $C_1$ is start of $M$ on $w$, $C_\ell$ is an accepting config, and each $C_i$ follows from $C_{i-1}$ per rules of $M$.
- <u>Rejecting computation history</u> is similar, but $C_\ell$ is a rejecting config.
- Computation history must be finite. If infinite, $M$ doesn't halt on $w$ - so no accepting/rejecting seq. can exist.
- <u>Linear bounded automaton</u> - restricted TM where tape head can't move off part of tape with input. (LBAs)
  - That is, constant memory
  - Deciders for $A_{DFA}, A_{CFG}, E_{DFA}, E_{CFG}$ are all LBAs; decidable languages that an LBA can't decide are difficult to make, but exist
  - But $A_{LBA} = \{\langle M, w\rangle \mid M \text{ is an LBA that accepts string } w\}$ is decidable. Note that for a given input length $n$ the number of configurations is fixed. (Lemma 5.8: With $q$ states, $g$ symbols, & tape length $n$, there are $qng^n$ configs.) Proof of decidability is based on this.
  - However, emptiness problem $E_{LBA}$ is undecidable.
- Mapping Reducibility
  - One simple way of doing reductions is <u>mapping reducibility</u>; loosely, it means that a computable function can turn an instance of problem $A$ to one of problem $B$, and thus we can solve $A$ if we can solve $B$.
  - <u>Computable function</u>: $f: \Sigma^* \to \Sigma^*$ is computable if some TM, $M$, on every input $w$, halts with just $f(w)$ on its tape.
  - Language $A$ is <u>mapping reducible</u> to language $B$ ($A \leq_m B$) if computable $f: \Sigma^* \to \Sigma^*$, for all $w$, $w \in A \Leftrightarrow f(w) \in B$. ($f$ is the <u>reduction</u> from $A$ to $B$).
    - That is, it converts questions about membership testing in $A$ to membership testing in $B$.

"B is as hard as A"

N.B. $f$ doesn't need to be 1:1, onto, or anything like that - just a function!

2017-04-04

(Thm 5.22)
- If $A \leq_m B$ and $B$ is decidable, so is $A$.
- If $A \leq_m B$ and $A$ is undecidable, so is $B$.
- Also: $A \leq_m B$ & $B$ is Turing-recognizable, so is $A$.
  $A \leq_m B$ & $A$ is not Turing-recognizable, neither is $B$.

---

Udacity, 5 (Undecidability)
- The Foundation for much of this lesson:
  $L = \{\langle M \rangle \mid M \text{ doesn't accept } \langle M \rangle\}$ is not recognizable (easily shown by diagonalization) → $L$ is undecidable.
  Thus $\overline{L} = D_{TM} = \{\langle M \rangle \mid M \text{ accepts } \langle M \rangle\}$ is undecidable.
- Dumaflache
  1. Halts on all inputs
  2. Can interpret Dumaflache descriptions & simulate them
  3. Can invert result of interpreted Dumaflache
- But consider...

```
def invD(⟨D⟩):
    if D(⟨D⟩) accepts, reject
    if D(⟨D⟩) rejects, accept
```

What is invD(⟨invD⟩)?
Contradiction means dumaflache can't exist.

- e.g. $B = \{\langle M \rangle \mid M \text{ accepts something}\}$
  - How do we show that this is undecidable?
  - One way: Reduce $\{\langle M \rangle \mid M \text{ accepts } \langle M \rangle\}$ to it, i.e. $D_{TM} \leq_m B$
    ($D_{TM}$ = diagonal language over TM)
  - e.g.

```
def R(⟨M⟩):
    def N(x):
        return M(⟨M⟩)
    return ⟨N⟩
```

- Reduction R never runs machine N - just writes it
- Note that N ignores its input.
  If M(⟨M⟩) doesn't accept, N accepts nothing. If it does, N accepts everything.
- So if we have a decider for B, we have a decider for $D_{TM}$, which is undecidable.

Chris Hodapp
chodapp3@gatech.edu

2017-04-04(b), CS6505 (Computability, Complexity, & Algorithms)

Udacity 5 (cont.)

- $H_{TM} = \{\langle M \rangle \mid M \text{ halts on } \vec{\varepsilon}\}$ is undecidable. ($\vec{\varepsilon}$ → empty string)
  - How: Show it's at least as hard as $D_{TM}$.
    i.e. $D_{TM} \leq_m H_{TM}$.

```
def R(⟨M⟩):
    def N(x):
        if M(⟨M⟩) rejects, loop
        accept
    return ⟨N⟩
```

?

- $S_{TM} = \{\langle M \rangle \mid \text{accepts exactly one string}\}$ – undecidable
  - How: Reduce $\{\langle M \rangle \mid M \text{ halts on } \varepsilon\}$ to it, i.e. $H_{TM} \leq_m S_{TM}$

```
def R(⟨M⟩):
    def N(x):
        M('')
        return x == ''
    return ⟨N⟩
```

- N loops if M('') does → $L(N) = \emptyset$
  Otherwise, it accepts just '' → $L(N) = \{\varepsilon\}$.

- Rice's theorem: $L = \{\langle M \rangle \mid L(M) \in P\}$ is undecidable.
  Key points:
  1. Membership in L depends only on $L(M)$. (Not on the machine's implementation.)
  2. Language can't be trivial, either including or excluding every TM:
     Assume $\exists \langle M_1 \rangle \in L$ & $\langle M_2 \rangle \notin L$
  - Case: $\emptyset \notin P$ . just reduce from Halting Problem
  - Case: $\emptyset \in P$ . switch $M_1$ for $M_2$ in reduction, & use Halting Problem's complement
- Theorem:
  Let L be a subset of strings representing TMs, where:
  1. If $M_1$ & $M_2$ recognize the same language, then either $\langle M_1 \rangle \langle M_2 \rangle \in L$ or $\langle M_1 \rangle, \langle M_2 \rangle \notin L$.
  2. $\exists M_1, M_2$ s.t. $\langle M_1 \rangle \in L$ & $\langle M_2 \rangle \notin L$.

  Then L is undecidable.

2017-04-10

Udacity 6 (P & NP)

- Computability is good, but it doesn't tell us if something is effectively or quickly computable
- This leads to P & NP
  - Example of NP: Finding 'cliques' in an undirected graph.
    - This is hard, but easy to check.
- P: Problems <u>solvable</u> in polynomial time. (e.g. bipartite matching)
- NP: Problems <u>verifiable</u> in polynomial time.
- It's generally believed that P occupies a part of NP, but not all of it - that is, problems exist that can be verified efficiently, but not solved efficiently. This is unproven however.
- NPC: NP complete, hardest problems in NP
  If any polynomial-time solution could be found to any NPC problem, we could convert all NPC problems to it in polynomial time and then P would equal NP.
- Shortest path = polynomial
- Traveling Salesman = NP-complete (so is longest simple path)
- More examples:

| <u>Polynomial</u> | <u>NP-complete</u> |
|---|---|
| Shortest path | Longest simple path |
| Vertex cover in bipartite graphs | Vertex cover in general graphs |
| Linear programming | Integer programming |
| Eulerian Cycle (touches each edge once) | Hamiltonian Cycle (touches each vertex once) |
| 2-CNF SAT | 3-CNF SAT |

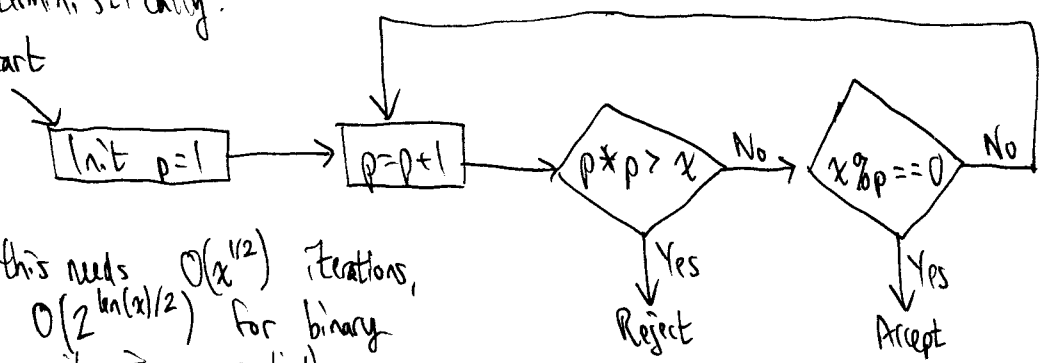
- They can be hard to tell apart.
- But be aware when you see an NP complete problem:
  You will typically choose between exact solutions that scale badly, and approximate solutions that might be polynomial.
- Let $M$ be a TM (any kind) halting on all inputs.
  Running time of $M$ is $f: \mathbb{N} \to \mathbb{N}$ where...
  $f(n) = \max\{m \mid m$ is the # of steps taken by $M$ on string of length $n\}$

Chris Hodapp
chodapp3@gatech.edu

2017-04-10(6), Computability, Complexity, & Algorithms (CS6505)

Udacity 6, P&NP (cont.)

- We mostly use asymptotic analysis here though. (See prior notes...)
- Then define $P = \{ L \mid L$ is a language recognized by an $O(n^k)$ time deterministic Turing machine for some $k \in \mathbb{N} \}$
  - 'language', to make it more formal, though it's really problems we want
  - Time $O(n^k)$ is as in 'running time' on prior page
- In practice, k is usually low (much higher k tends to be intractable).
- P is the same for different types of TM (can translate in polynomial time)
- P is closed under composition of algorithms (if A is $\in P$ and be calls A a polynomial number of times then $B \in P$)
- Problem: How we encode a problem into a language can determine whether or not it's in P!
  - We deal with this by ignoring "unreasonable representations"
- We gave one definition of NP, but we also define NP as the class of problems solvable on a nondeterministic TM in polynomial time
- In a nondeterministic TM, multiple successor configurations can exist at each state. The only change in definition is that $\delta : Q \times \Gamma \to \underbrace{\{S \mid S \subseteq Q \times \Gamma \times \{L, R\}\}}_{\text{Power set}}$ and that NTM accepts if any valid sequence of configurations reaches accepting state, and rejects if every branch does.
- Consider task of identifying composite numbers, i.e. $\{x \mid x = str(p * q)$ for $p, q > 1\}$
  - Deterministically:




- But this needs $O(x^{1/2})$ iterations, or $O(2^{len(x)/2})$ for binary (thus it is exponential)

2017-04-10

Udacity 6, P&NP, cont.

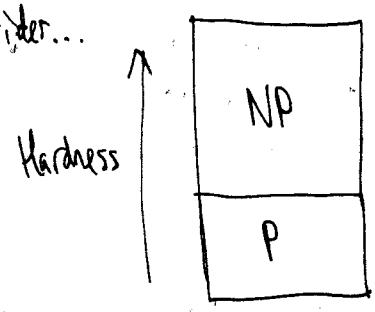
- Nondeterministic: By simultaneously trying various p values (not going to copy it here) it can do $O(\log x) = O(\text{len}(x))$
- But what's running time on an NTM?
  - We use the maximum length on all steps.
- Then: $NP = \{L \mid L$ is a language recognized by an $O(n^k)$ time nondeterministic TM for some $k \in \mathbb{N}\}$.
  - Any polynomial-time recognizer over a language can easily be turned into a polynomial-time decider by adding a timeout. (All accepting configurations are bounded).
- Def: A verifier for a language L is an algorithm V, where $L = \{w \mid V$ accepts $\langle w, c \rangle$ for some string c, and V halts in $O(|w|^k)$ steps$\}$
  → $NP = \{L \mid L$ has a polytime verifier$\}$
- A verifier can't run every path in the NTM - but it can run the specific accepting path, if the 'certificate' c supplies an accepting path by way of what branches to take
- From the other direction: If we have the verifier, we can create a nondeterministic TM which begins by doing a lot of branching to create every possible c, then deterministically (in each branch) simulates verifier and in one (or more) paths finds an accepting configuration.

Chris Hodapp
chodapp3@gatech.edu

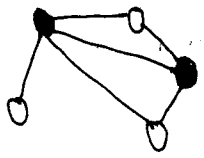
2017-04-12, CS6505 (Computability, Complexity, & Algorithms)

Udacity 7, NP-Completeness

- Consider...




- We can show that a problem is intractable by...
  1. Showing it's <u>not</u> in P. (This would prove $P \neq NP$, so good luck)
  2. Reduce another intractable method to it. (Less convincing)
  3. Reduce an NP-Complete problem to it.
- We focus on #3. We are concerned only with reductions that can be done in polynomial time.
- Def: Language A is <u>polynomial-time reducible</u> to language B (i.e. $A \leq_p B$) if there is a <u>polynomial-time function</u> $f: \Sigma^* \to \Sigma^*$ where for every w, $w \in A \Leftrightarrow f(w) \in B$.
  - f is then a polynomial time reduction of A to B.
- Then if B is in P, and we can convert any $x \in A$ to B with f in polynomial time, we can do both in polynomial time
- Theorem: If $A \leq_p B$ and $B \in P$, then $A \in P$.
  Also, $A \notin P \to B \notin P$.
- Independent Set / Clique:
  - Def: Given graph $G=(V,E)$, $S \subseteq V$ is an <u>independent set</u> if there are no edges between vertices in S.
  - Note that independent set $\Leftrightarrow$ clique in <u>complement</u> graph
  - Identifying if graph G has an independent graph of size $k \leq |V|$ is in NP
- Def: Given graph $G=(V,E)$, $S \subseteq V$ is a <u>vertex cover</u> if every edge is incident on a vertex in S. e.g.:



Darkened vertices are vertex cover.

  - Decision question: Has G a vertex cover of size $k \leq |V|$? (This is in NP)
- Independent Set $\leq_p$ Vertex Cover

2017-04-12

Udacity 7, NP-Completeness (cont.)

- S is an independent set iff there are no edges within S ⇔
  S is " " " iff every edge is within S ⇔
  S is " " " iff every edge is incident on V−S ⇔
  V−S is a vertex cover iff S is an independent set
- Corollary: G has an independent set ≥k iff G has vertex cover ≤|V|−k
- So: $(G,k) \xrightarrow{f} (G, |V|-k)$

- Theorem: If $A \leq_p B$ & $B \leq_p C$, then $A \leq_p C$. (Reducibility is transitive)
- NP-Completeness:
  - Language is NP-Complete if:
    1. $L \in NP$
    2. For all other $L' \in NP$, $L' \leq_p L$
  - NP-Complete is the "hardest" in NP. Nothing higher can still be in NP
- Theorem (Cook-Levin): SAT is NP-complete.
- CNF Satisfiability
  - CNF Boolean formula: e.g. $(x \vee y) \wedge (\overline{x} \vee z)$ (OR, AND, NOT, OR)
    - Variables: e.g. $\{x, y, z\}$
    - Literals: e.g. $x$, $y$
    - Clauses: Disjunction (OR) of literals (Clause, Clause)
    - CNF formula (conjunctive normal form): Conjunction of clauses
  - A formula is <u>satisfiable</u> if a truth assignment exists such that the formula evaluates to true.
- Decision problem: Given a formula in CNF, is there a satisfying assignment?
- See parts 18, 19, 20, 21, 22, 23 for a brief proof of Cook-Levin using NTMs
- Many problems can be shown NP-complete by reducing satisfiability to them

Chris Hodapp
chodapp3@gatech.edu

2017-04-12b, CS6505 (Computability, Complexity, & Algorithms)

Idacity 7, NP Completeness (cont.)
Sipser, Ch. 7 (Time Complexity)

- Problem of determining whether a graph has a Hamiltonian path (path from s to t which goes through every node exactly once) can be polynomially reduced from 3SAT (i.e. satisfiability of 3CNF formula with k clauses, $(a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k)$
  - See p314
- Likewise, subset-sum problem (p320).