Lecture notes for CSCI 405 Fall 2024

Kameron Decker Harris

October 21, 2024

Textbook: "Introduction to Algorithms" by Cormen, Leiserson, Rivest, and Stein (4th ed). Note that we recently changed to the new edition. Please let me know if page/formula numbers are incorrect in my materials.

Contents

1 Introduction to class

Welcome to class!

Go over syllabus: culture and environment, communication, schedule

Icebreaker: 2 truths and 1 lie

- Break into groups of 5 (count off numbers 1–7 for group of 35)
- Members have 3 minutes to write down truths and lies about themselves
- One at a time, people reveal the 3 things
- Group has 30 s to vote on what the lie is
- Afterwards, the actual lie is revealed

After the game finishes, go around the class to do full introductions and reveal 1 thing about each person.

Open-ended question 1: What makes a "good" algorithm? (Think-pair-share)

We will talk about the ethics of algorithms in this class. This is the main topic for your essay-writing.

Open-ended question 2: Is an efficient algorithm always good? (Think-pair-share)

Example of efficiency not leading to good things: **Jevons paradox**. William Stanley Jevons was a British economist and mathematician in 19th century. He proposed that increased efficiency of coal furnaces actually led to *more* coal consumption. From "The Coal Question," 1865, chapter VI:

"It is wholly a confusion of ideas to suppose that the economical use of fuel is equivalent to a diminished consumption. The very contrary is the truth."

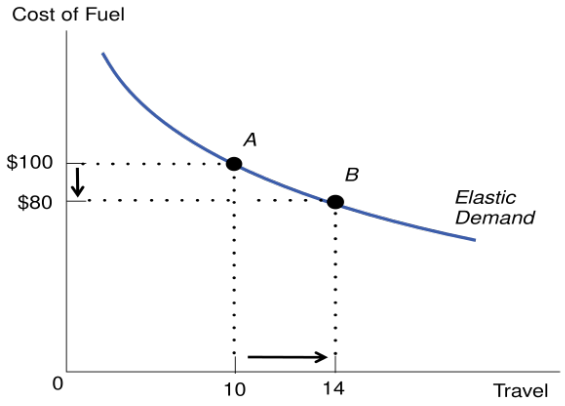


Figure 1: Elastic Demand: A 20% increase in efficiency causes a 40% increase in travel. Fuel consumption increases and the Jevons paradox occurs. (Wiki: Lawrencekhoo)

Other examples: (1) If fuel is cheaper, people tend to travel more, or (2) increasing road capacity makes congestion worse (Downs-Thomson paradox). Depending on supply/demand curve, the paradox may or may not occur.

Discuss: Do you think that can happen with algorithms?

It is worth thinking about algorithms from multiple perspectives:

- Practical
- Mathematical/theoretical

- Cultural
- Environmental
- Economical

There are also multiple ethical frameworks through which we may address human questions.

Further reading/discussion:

- Read the syllabus. Quiz Friday
- Introduction to Philosophy: Ethics. Free textbook.
- Wirth's law (Niklaus Wirth): "software is getting slower more rapidly than hardware is becoming faster"
- Andy and Bill's law (Intel CEO Andy Grove, Bill Gates): "new software will always consume any increase in computing power that new hardware can provide"

2 Chapter 14: Dynamic programming

Reading: Sections 14.1–14.4

Lecture Notes for Chapter 14: Dynamic Programming

Dynamic Programming

- Not a specific algorithm, but a technique (like divide-and-conquer).
- Developed back in the day when "programming" meant "tabular method" (like linear programming). Doesn't really refer to computer programming.
- Used for optimization problems:
  - Find *a* solution with *the* optimal value.
  - Minimization or maximization. (We'll see both.)

Four-step method

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Rod cutting

How to cut steel rods into pieces in order to maximize the revenue you can get? Each cut is free. Rod lengths are always an integer number of inches.

Input: A length n and table of prices p_i, for $i = 1, 2, \ldots, n$.

Output: The maximum revenue obtainable for rods whose lengths sum to n, computed as the sum of the prices for the individual rods.

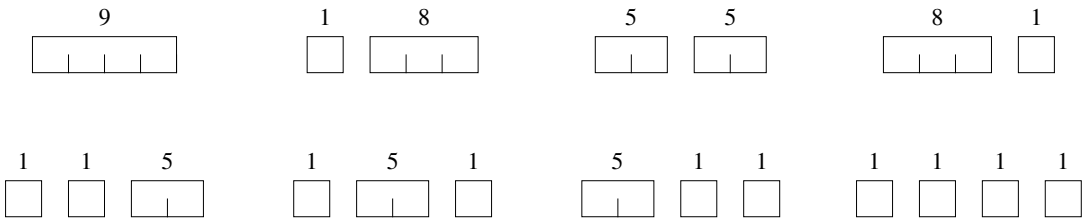
If p_n is large enough, an optimal solution might require no cuts, i.e., just leave the rod as n inches long.

Example: *[Using the first* 8 *values from the example in the textbook.]*

length i	1	2	3	4	5	6	7	8
price p_i	1	5	8	9	10	17	17	20

Can cut up a rod in 2^{n-1} different ways, because can choose to cut or not cut after each of the first $n-1$ inches.

Here are all 8 ways to cut a rod of length 4, with the costs from the example:



The best way is to cut it into two 2-inch pieces, getting a revenue of $p_2 + p_2 = 5 + 5 = 10$.

Let r_i be the maximum revenue for a rod of length i. Can express a solution as a sum of individual rod lengths.

Can determine optimal revenues r_i for the example, by inspection:

i	r_i	optimal solution
1	1	1 (no cuts)
2	5	2 (no cuts)
3	8	3 (no cuts)
4	10	$2 + 2$
5	13	$2 + 3$
6	17	6 (no cuts)
7	18	$1 + 6$ or $2 + 2 + 3$
8	22	$2 + 6$

Can determine optimal revenue r_n by taking the maximum of

- p_n: the revenue from not making a cut,
- $r_1 + r_{n-1}$: the maximum revenue from a rod of 1 inch and a rod of $n-1$ inches,
- $r_2 + r_{n-2}$: the maximum revenue from a rod of 2 inches and a rod of $n-2$ inches, ...
- $r_{n-1} + r_1$.

That is,

$$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\} \ .$$

Optimal substructure: To solve the original problem of size n, solve subproblems on smaller sizes. After making a cut, two subproblems remain. The optimal solution to the original problem incorporates optimal solutions to the subproblems. May solve the subproblems independently.

Example: For $n = 7$, one of the optimal solutions makes a cut at 3 inches, giving two subproblems, of lengths 3 and 4. Need to solve both of them optimally. The optimal solution for the problem of length 4, cutting into 2 pieces, each of length 2, is used in the optimal solution to the original problem with length 7.

A simpler way to decompose the problem: Every optimal solution has a leftmost cut. In other words, there's some cut that gives a first piece of length i cut off the left end, and a remaining piece of length $n - i$ on the right.

- Need to divide only the remainder, not the first piece.
- Leaves only one subproblem to solve, rather than two subproblems.
- Say that the solution with no cuts has first piece size $i = n$ with revenue p_n, and remainder size 0 with revenue $r_0 = 0$.
- Gives a simpler version of the equation for r_n:

  $$r_n = \max\{p_i + r_{n-i} : 1 \le i \le n\} \ .$$

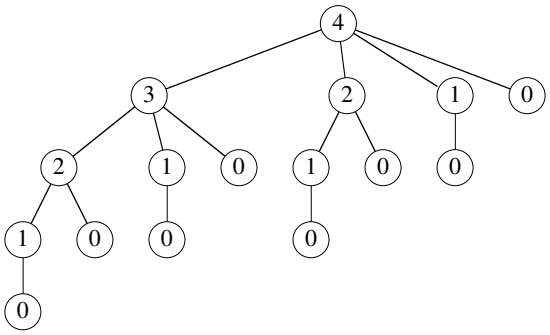
Recursive top-down solution

Direct implementation of the simpler equation for r_n.
The call CUT-ROD(p, n) returns the optimal revenue r_n:

```
CUT-ROD(p, n)
  if n == 0
      return 0
  q = -∞
  for i = 1 to n
      q = max{q, p[i] + CUT-ROD(p, n - i)}
  return q
```

This procedure works, but it is terribly *inefficient*. If you code it up and run it, it could take more than an hour for $n = 40$. Running time approximately doubles each time n increases by 1.

Why so inefficient?: CUT-ROD calls itself repeatedly, even on subproblems it has already solved. Here's a tree of recursive calls for $n = 4$. Inside each node is the value of n for the call represented by the node:



Lots of repeated subproblems. Solves the subproblem for size 2 twice, for size 1 four times, and for size 0 eight times.

Exponential growth: Let $T(n)$ equal the number of calls to CUT-ROD with second parameter equal to n. Then

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \,, \\ 1 + \displaystyle\sum_{j=0}^{n-1} T(j) & \text{if } n \geq 1 \,. \end{cases}$$

Summation counts calls where second parameter is $j = n - i$.
Solution to recurrence is $T(n) = 2^n$.

Dynamic-programming solution

Instead of solving the same subproblems repeatedly, arrange to solve each subproblem just once.
Save the solution to a subproblem in a table, and refer back to the table whenever we revisit the subproblem.
"Store, don't recompute" $\Rightarrow$ time-memory trade-off.
Can turn an exponential-time solution into a polynomial-time solution.
Two basic approaches: top-down with memoization, and bottom-up.

Top-down with memoization

Solve recursively, but store each result in a table.
To find the solution to a subproblem, first look in the table. If the answer is there, use it. Otherwise, compute the solution to the subproblem and then store the solution in the table for future use.

Memoizing is remembering what has been computed previously. *["Memoizing," not "memorizing."]*

Memoized version of the recursive solution, storing the solution to the subproblem of length i in array entry $r[i]$:

```
MEMOIZED-CUT-ROD(p, n)
  let r[0:n] be a new array          // will remember solution values in r
  for i = 0 to n
      r[i] = -∞
  return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
  if r[n] ≥ 0                        // already have a solution for length n?
      return r[n]
  if n == 0
      q = 0
  else q = -∞
      for i = 1 to n                 // i is the position of the first cut
          q = max{q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r)}
  r[n] = q                           // remember the solution value for length n
  return q
```

Bottom-up

Sort the subproblems by size and solve the smaller ones first. That way, when solving a subproblem, have already solved the smaller subproblems needed.

```
BOTTOM-UP-CUT-ROD(p, n)
  let r[0:n] be a new array        // will remember solution values in r
  r[0] = 0
  for j = 1 to n                   // for increasing rod length j
      q = -∞
      for i = 1 to j               // i is the position of the first cut
          q = max{q, p[i] + r[j - i]}
      r[j] = q                     // remember the solution value for length j
  return r[n]
```

Running time

Both the top-down and bottom-up versions run in $\Theta(n^2)$ time.

- Bottom-up: Doubly nested loops. Number of iterations of inner **for** loop forms an arithmetic series.
- Top-down: MEMOIZED-CUT-ROD solves each subproblem just once, and it solves subproblems for sizes $0, 1, \ldots, n$. To solve a subproblem of size n, the **for** loop iterates n times $\Rightarrow$ over all recursive calls, total number of iterations forms an arithmetic series. *[Actually using aggregate analysis, which Chapter 16 covers.]*

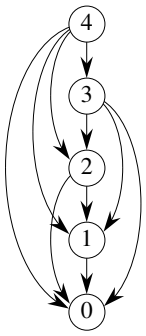
Subproblem graphs

How to understand the subproblems involved and how they depend on each other.

Directed graph:

- One vertex for each distinct subproblem.
- Has a directed edge (x, y) if computing an optimal solution to subproblem x *directly* requires knowing an optimal solution to subproblem y.

Example: For rod-cutting problem with $n = 4$:



Can think of the subproblem graph as a collapsed version of the tree of recursive calls, where all nodes for the same subproblem are collapsed into a single vertex, and all edges go from parent to child.

Subproblem graph can help determine running time. Because each subproblem is solved just once, running time is sum of times needed to solve each subproblem.

- Time to compute solution to a subproblem is typically linear in the out-degree (number of outgoing edges) of its vertex.
- Number of subproblems equals number of vertices.

When these conditions hold, running time is linear in number of vertices and edges.

Reconstructing a solution

So far, have focused on computing the *value* of an optimal solution, rather than the *choices* that produced an optimal solution.

Extend the bottom-up approach to record not just optimal values, but optimal choices. Save the optimal choices in a separate table. Then use a separate procedure to print the optimal choices.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  let r[0:n] and s[1:n] be new arrays
  r[0] = 0
  for j = 1 to n                    // for increasing rod length j
      q = -∞
      for i = 1 to j                // i is the position of the first cut
          if q < p[i] + r[j - i]
              q = p[i] + r[j - i]
              s[j] = i              // best cut location so far for length j
      r[j] = q                      // remember the solution value for length j
  return r and s
```

Saves the first cut made in an optimal solution for a problem of size i in $s[i]$.

To print out the cuts made in an optimal solution:

```
PRINT-CUT-ROD-SOLUTION(p, n)
  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  while n > 0
      print s[n]          // cut location for length n
      n = n - s[n]        // length of the remainder of the rod
```

Example: For the example, EXTENDED-BOTTOM-UP-CUT-ROD returns

i	0	1	2	3	4	5	6	7	8
$r[i]$	0	1	5	8	10	13	17	18	22
$s[i]$		1	2	3	2	2	6	1	2

A call to PRINT-CUT-ROD-SOLUTION$(p, 8)$ calls EXTENDED-BOTTOM-UP-CUT-ROD to compute the above r and s tables. Then it prints 2, sets n to 6, prints 6, and finishes (because n becomes 0).

Matrix-chain multiplication

Problem: Given a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, compute the product $A_1 A_2 \cdots A_n$ using standard matrix multiplication (not Strassen's method) while minimizing the number of scalar multiplications.

How to parenthesize the product to minimize the number of scalar multiplications?

Suppose multiplying matrices A and B: $C = A \cdot B$. *[The textbook has a procedure to compute $C = C + A \cdot B$, but it's easier in a lecture situation to just use $C = A \cdot B$.]* The matrices must be compatible: number of columns of A equals number of rows of B. If A is $p \times q$ and B is $q \times r$, then C is $p \times r$ and takes pqr scalar multiplications.

Example: $A_1 : 10 \times 100$, $A_2 : 100 \times 5$, $A_3 : 5 \times 50$. Compute $A_1 A_2 A_3$, which is 10×50.

- Try parenthesizing by $((A_1 A_2) A_3)$. First perform $10 \cdot 100 \cdot 5 = 5000$ multiplications, then perform $10 \cdot 5 \cdot 50 = 2500$, for a total of 7500.
- Try parenthesizing by $(A_1 (A_2 A_3))$. First perform $100 \cdot 5 \cdot 50 = 25{,}000$ multiplications, then perform $10 \cdot 100 \cdot 50 = 50{,}000$, for a total of 75,000.
- The first way is 10 times faster.

Input to the problem: Let A_i be $p_{i-1} \times p_i$. The input is the sequence of dimensions $\langle p_0, p_1, p_2, \ldots, p_n \rangle$.

Note: Not actually multiplying matrices. Just deciding an order with the lowest cost.

Counting the number of parenthesizations

Let $P(n)$ denote the number of ways to parenthesize a product of n matrices. $P(1) = 1$.

When $n \geq 2$, can split anywhere between A_k and A_{k+1} for $k = 1, 2, \ldots, n-1$. Then have to split the subproducts. Get

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \,, \\ \displaystyle\sum_{k=1}^{n-1} P(k) P(n-k) & \text{if } n \geq 2 \,. \end{cases}$$

The solution is $P(n) = \Omega(4^n / n^{3/2})$. *[The textbook does not prove the solution to this recurrence.]* So brute force is a bad strategy.

Step 1: Structure of an optimal solution

Let $A_{i:j}$ be the matrix product $A_i A_{i+1} \cdots A_j$.

If $i < j$, then must split between A_k and A_{k+1} for some $i \leq k < j \Rightarrow$ compute $A_{i:k}$ and $A_{k+1:j}$ and then multiply them together. Cost is

cost of computing $A_{i:k}$
\+ cost of computing $A_{k+1:j}$
\+ cost of multiplying them together .

Optimal substructure: Suppose that optimal parenthesization of $A_{i:j}$ splits between A_k and A_{k+1}. Then the parenthesization of $A_{i:k}$ must be optimal. Otherwise, if there's a less costly way to parenthesize it, you'd use it and get a parenthesization of $A_{i:j}$ with a lower cost. Same for $A_{k+1:j}$.

Therefore, to build an optimal solution to $A_{i:j}$, split it into how to optimally parenthesize $A_{i:k}$ and $A_{k+1:j}$, find optimal solutions to these subproblems, and then combine the optimal solutions. Need to consider all possible splits.

Step 2: A recursive solution

Define the cost of an optimal solution recursively in terms of optimal subproblem solutions.

Let $m[i, j]$ be the minimum number of scalar multiplications to compute $A_{i:j}$. For the full problem, want $m[1, n]$.

If $i = j$, then just one matrix $\Rightarrow m[i, i] = 0$ for $i = 1, 2, \ldots, n$.

If $i < j$, then suppose the optimal split is between A_k and A_{k+1}, where $i \le k < j$. Then $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_i p_j$.

But that's assuming you know the value of k. Have to try all possible values and pick the best, so that

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \ , \\ \min \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j : i \le k < j\} & \text{if } i < j \ . \end{cases}$$

That formula gives the cost of an optimal solution, but not how to construct it. Define $s[i, j]$ to be a value of k to split $A_{i:j}$ in an optimal parenthesization. Then $s[i, j] = k$ such that $m[i, j] = m[i, k] + m[k_1, j] + p_{i-1} p_k p_j$.

Step 3: Compute the optimal costs

Could implement a recursive algorithm based on the above equation for $m[i, j]$. *Problem:* It would take exponential time.

There are not all that many subproblems: just one for each i, j such that $1 \le i \le j \le n$. There are $\binom{n}{2} + n = \Theta(n^2)$ of them. Thus, a recursive algorithm would solve the same subproblems over and over.

In other words, this problem has overlapping subproblems.

Here is a tabular, bottom-up method to solve the problem. It solves subproblems in order of increasing chain length. The variable $l = j - i + 1$ indicates the chain length.

```
MATRIX-CHAIN-ORDER(p, n)
  let m[1:n, 1:n] and s[1:n − 1, 2:n] be new tables
  for i = 1 to n                          // chain length 1
      m[i, i] = 0
  for l = 2 to n                          // l is the chain length
      for i = 1 to n − l + 1              // chain begins at A_i
          j = i + l − 1                   // chain ends at A_j
          m[i, j] = ∞
          m[i, j] = ∞
          for k = i to j − 1
              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
              if q < m[i, j]
                  m[i, j] = q             // remember this cost
                  s[i, j] = k             // remember this index
  return m and s
```

All n chains of length 1 are initialized so that $m[i, i] = 0$ for $i = 1, 2, \ldots, n$. Then $n - 1$ chains of length 2 are computed, then $n - 2$ chains of length 3, and so on, up to 1 chain of length n.

[We don't include an example here because the arithmetic is hard for students to process in real time.]

Time: $O(n^3)$, from triply nested loops. Also $\Omega(n^3) \Rightarrow \Theta(n^3)$.

Step 4: Construct an optimal solution

With the s table filled in, recursively print an optimal solution.

```
PRINT-OPTIMAL-PARENS(s, i, j)
  if i == j
      print "A"_i
  else print "("
      PRINT-OPTIMAL-PARENS(s, i, s[i, j])
      PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
      print ")"
```

Initial call is PRINT-OPTIMAL-PARENS$(s, 1, n)$

Longest common subsequence

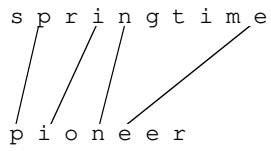
[The textbook has the section on elements of dynamic programming next, but these lecture notes reserve that section for the end of the chapter so that it may refer to two more examples of dynamic programming.]

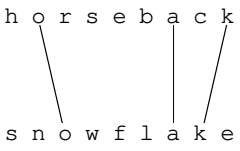
Problem: Given two sequences, $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$. Find a subsequence common to both whose length is longest. A subsequence doesn't have to be consecutive, but it has to be in order.

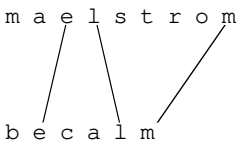
[To come up with examples of longest common subsequences, search the dictionary for all words that contain the word you are looking for as a subsequence. On a UNIX system, for example, to find all the words with `pine` as a subsequence, use the command `grep '.*p.*i.*n.*e.*' dict`, where `dict` is your local dictionary. Then check if that word is actually a longest common subsequence. Working C code for finding a longest common subsequence of two strings appears at http://www.cs.dartmouth.edu/~thc/code/lcs.c The comments in the code refer to the second edition of the textbook, but the code is correct.]

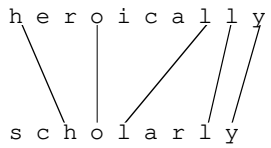
Examples

[The examples are of different types of trees.]









Brute-force algorithm:

For every subsequence of X, check whether it's a subsequence of Y.

Time: $\Theta(n2^m)$.

- 2^m subsequences of X to check.
- Each subsequence takes $\Theta(n)$ time to check: scan Y for first letter, from there scan for second, and so on.

Step 1: Characterize an LCS

Notation:

X_i = prefix $\langle x_1, \ldots, x_i \rangle$

Y_i = prefix $\langle y_1, \ldots, y_i \rangle$

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$ and $z_k \neq x_m$, then Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$ and $z_k \neq y_n$, then Z is an LCS of X and Y_{n-1}.

Proof

1. First show that $z_k = x_m = y_n$. Suppose not. Then make a subsequence $Z' = \langle z_1, \ldots, z_k, x_m \rangle$. It's a common subsequence of X and Y and has length $k + 1 \Rightarrow Z'$ is a longer common subsequence than $Z \Rightarrow$ contradicts Z being an LCS.

   Now show Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}. Clearly, it's a common subsequence. Now suppose there exists a common subsequence W of X_{m-1} and Y_{n-1} that's longer than $Z_{k-1} \Rightarrow$ length of $W \geq k$. Make subsequence W' by appending x_m to W. W' is common subsequence of X and Y, has length $\geq k + 1$ $\Rightarrow$ contradicts Z being an LCS.
2. If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y. Suppose there exists a subsequence W of X_{m-1} and Y with length $> k$. Then W is a common subsequence of X and $Y \Rightarrow$ contradicts Z being an LCS.
3. Symmetric to 2. ■ (theorem)

Therefore, an LCS of two sequences contains as a prefix an LCS of prefixes of the sequences.

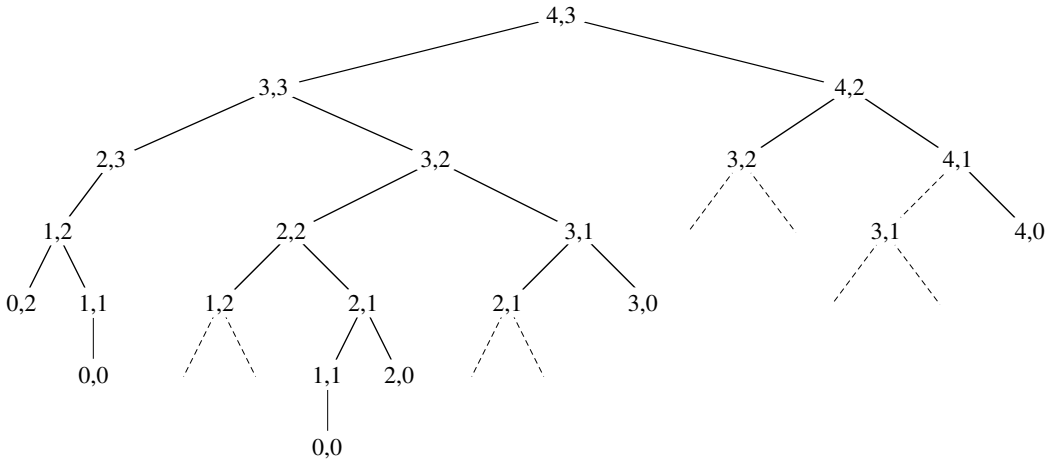
Step 2: Recursively define an optimal solution

Define $c[i, j]$ = length of LCS of X_i and Y_j. Want $c[m, n]$.

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i-1, j], c[i, j-1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

Again, could write a recursive algorithm based on this formulation.

Try with $X = \langle a, t, o, m \rangle$ and $Y = \langle a, n, t \rangle$. Numbers in nodes are values of i, j in each recursive call. Dashed lines indicate subproblems already computed.



- Lots of repeated subproblems.
- Instead of recomputing, store in a table.

Step 3: Compute the length of an LCS

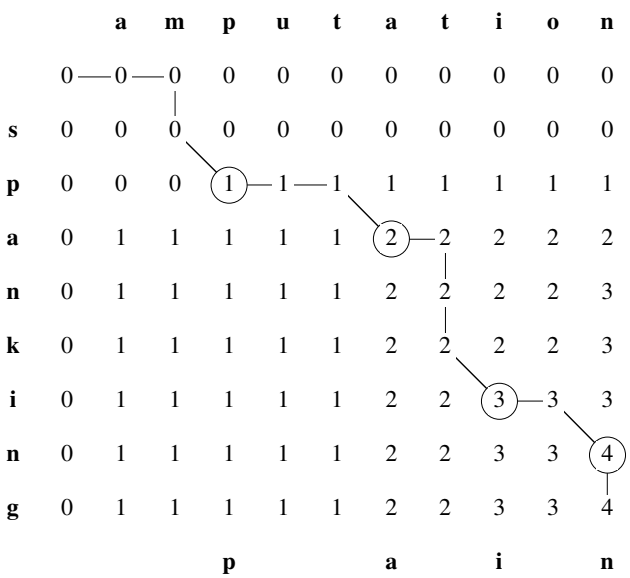
```
LCS-LENGTH(X, Y, m, n)
  let b[1:m, 1:n] and c[0:m, 0:n] be new tables
  for i = 1 to m
      c[i, 0] = 0
  for j = 0 to n
      c[0, j] = 0
  for i = 1 to m          // compute table entries in row-major order
      for j = 1 to n
          if x_i == y_j
              c[i, j] = c[i − 1, j − 1] + 1
              b[i, j] = "↖"
          else if c[i − 1, j] ≥ c[i, j − 1]
                  c[i, j] = c[i − 1, j]
                  b[i, j] = "↑"
          else c[i, j] = c[i, j − 1]
                  b[i, j] = "←"
  return c and b
```

```
PRINT-LCS(b, X, i, j)
  if i == 0 or j = 0
      return              // the LCS has length 0
  if b[i, j] == "↖"
      PRINT-LCS(b, X, i − 1, j − 1)
      print x_i           // same as y_j
  elseif b[i, j] == "↑"
      PRINT-LCS(b, X, i − 1, j)
  else PRINT-LCS(b, X, i, j − 1)
```

- Initial call is PRINT-LCS(b, X, m, n).
- $b[i, j]$ points to table entry whose subproblem was used in solving LCS of X_i and Y_j.
- When $b[i, j] = \nwarrow$, LCS extended by one character. So longest common subsequence = entries with $\nwarrow$ in them.

Demonstration

What do `spanking` and `amputation` have in common? *[Show only $c[i, j]$.]*



Answer: `pain`.

Time

$\Theta(mn)$

Improving the code

Don't really need the b table. $c[i, j]$ depends only on $c[i-1, j-1]$, $c[i-1, j]$, and $c[i, j-1]$. Given $c[i, j]$, can determine in constant time which of the three values was used to compute $c[i, j]$. *[Exercise 14.4-2.]*

Or, if only need the length of an LCS, and don't need to construct the LCS itself, can get away with storing only one row of the c table plus a constant amount of additional entries. *[Exercise 14.4-4.]*

Optimal binary search trees

- Given sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys, sorted ($k_1 < k_2 < \cdots < k_n$).
- Want to build a binary search tree from the keys.
- For k_i, have probability p_i that a search is for k_i.
- Want BST with minimum expected search cost.
- Actual cost = # of items examined.

  For key k_i, cost = $\text{depth}_T(k_i) + 1$, where $\text{depth}_T(k_i)$ = depth of k_i in BST T.

E [search cost in T]

$$= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i$$

$$= \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=1}^{n} p_i$$

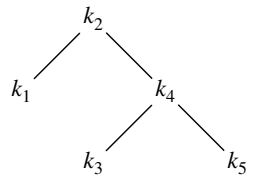
$$= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i \qquad \text{(since probabilities sum to 1)} \qquad (*)$$

[Keep equation (∗) on board.]

[Similar to optimal BST problem in the textbook, but simplified here: we assume that all searches are successful. Textbook has probabilities of searches between keys in tree.]

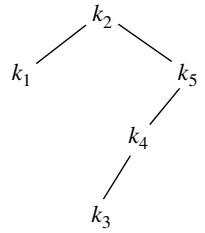
Example

i	1	2	3	4	5
p_i	.25	.2	.05	.2	.3



i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	.25
2	0	0
3	2	.1
4	1	.2
5	2	.6
		1.15

Therefore, E [search cost] = 2.15.



i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	.25
2	0	0
3	3	.15
4	2	.4
5	1	.3
		1.10

Therefore, E [search cost] = 2.10, which turns out to be optimal.

Observations

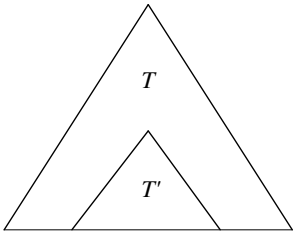
- Optimal BST might not have smallest height.
- Optimal BST might not have highest-probability key at root.

Build by exhaustive checking?

- Construct each n-node BST.
- For each, put in keys.
- Then compute expected search cost.
- But there are $\Omega(4^n/n^{3/2})$ different BSTs with n nodes.

Step 1: The structure of an optimal binary search tree

Consider any subtree of a BST. It contains keys in a contiguous range $k_i, \ldots, k_j$ for some $1 \le i \le j \le n$.

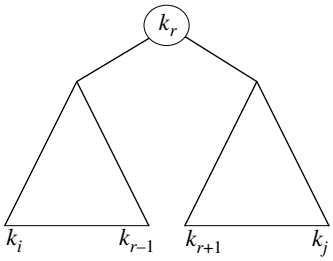


If T is an optimal BST and T contains subtree T' with keys $k_i, \ldots, k_j$, then T' must be an optimal BST for keys $k_i, \ldots, k_j$.

Proof Cut and paste. ■

Use optimal substructure to construct an optimal solution to the problem from optimal solutions to subproblems:

- Given keys $k_i, \ldots, k_j$ (the problem).
- One of them, k_r, where $i \le r \le j$, must be the root.
- Left subtree of k_r contains $k_i, \ldots, k_{r-1}$.
- Right subtree of k_r contains $k_{r+1}, \ldots, k_j$.



- If
  - you examine all candidate roots k_r, for $i \le r \le j$, and
  - you determine all optimal BSTs containing $k_i, \ldots, k_{r-1}$ and containing $k_{r+1}, \ldots, k_j$,

  then you're guaranteed to find an optimal BST for $k_i, \ldots, k_j$.

Step 2: Recursive solution

Subproblem domain:

- Find optimal BST for $k_i, \ldots, k_j$, where $i \geq 1$, $j \leq n$, $j \geq i-1$.
- When $j = i-1$, the tree is empty.

Define $e[i, j]$ = expected search cost of optimal BST for $k_i, \ldots, k_j$.

If $j = i-1$, then $e[i, j] = 0$.

If $j \geq i$,

- Select a root k_r, for some $i \leq r \leq j$.
- Make an optimal BST with $k_i, \ldots, k_{r-1}$ as the left subtree.
- Make an optimal BST with $k_{r+1}, \ldots, k_j$ as the right subtree.
- Note: when $r = i$, left subtree is $k_i, \ldots, k_{i-1}$; when $r = j$, right subtree is $k_{j+1}, \ldots, k_j$. These subtreees are empty.

When a subtree becomes a subtree of a node:

- Depth of every node in subtree goes up by 1.
- Expected search cost increases by

$$w(i, j) = \sum_{l=i}^{j} p_l \qquad \text{(refer to equation } (*)\text{)} .$$

If k_r is the root of an optimal BST for $k_i, \ldots, k_j$:

$$e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j)) .$$

But $w(i, j) = w(i, r-1) + p_r + w(r+1, j)$.

Therefore, $e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j)$.

This equation assumes that we already know which key is k_r.

We don't.

Try all candidates, and pick the best one:

$$e[i, j] = \begin{cases} 0 & \text{if } j = i-1 , \\ \min \{e[i, r-1] + e[r+1, j] + w(i, j) : i \leq r \leq j\} & \text{if } i \leq j . \end{cases}$$

Could write a recursive algorithm...

Step 3: Computing the expected search cost of an optimal binary search tree

As "usual," store the values in a table:

$$e[\ \underbrace{1 : n+1}_{\substack{\text{can store} \\ e[n+1, n]}}\ ,\ \underbrace{0 : n}_{\substack{\text{can store} \\ e[1, 0]}}\ ]$$

- Will use only entries $e[i, j]$, where $j \geq i-1$.

- Will also compute

  $root[i, j]$ = root of subtree with keys $k_i, \ldots, k_j$, for $1 \le i \le j \le n$.

One other table: don't recompute $w(i, j)$ from scratch every time we need it. (Would take $\Theta(j - i)$ additions.)

Instead:

- Table $w[1:n+1, 0:n]$
- $w[i, i-1] = 0$ for $1 \le i \le n$
- $w[i, j] = w[i, j-1] + p_j$ for $1 \le i \le j \le n$

Can compute all $\Theta(n^2)$ values in $O(1)$ time each.

```
OPTIMAL-BST(p, q, n)
  let e[1 : n + 1, 0 : n], w[1 : n + 1, 0 : n], and root[1 : n, 1 : n] be new tables
  for i = 1 to n + 1            // base cases
      e[i, i − 1] = 0
      w[i, i − 1] = 0
  for l = 1 to n
      for i = 1 to n − l + 1
          j = i + l − 1
          e[i, j] = ∞
          w[i, j] = w[i, j − 1] + p_j
          for r = i to j            // try all possible roots r
              t = e[i, r − 1] + e[r + 1, j] + w[i, j]
              if t < e[i, j]        // new minimum?
                  e[i, j] = t
                  root[i, j] = r
  return e and root
```

First **for** loop initializes e, w entries for subtrees with 0 keys.

Main **for** loop:

- Iteration for l works on subtrees with l keys.
- Idea: compute in order of subtree sizes, smaller (1 key) to larger (n keys).

For example at beginning:

	e \ j	0	1	2	3	4	5
i	1	0	.25	.65	.8	1.25	2.10
	2		0	.2	.3	.75	1.35
	3			0	.05	.3	.85
	4				0	.2	.7
	5					0	.3
	6						0

p_i

		j					
	w	0	1	2	3	4	5
	1	0	.25	.45	.5	.7	1.0
	2		0	.2	.25	.45	.75
	3			0	.05	.25	.55
i	4				0	.2	.5
	5					0	.3
	6						0

		j				
	root	1	2	3	4	5
	1	1	1	1	2	2
	2		2	2	2	4
i	3			3	4	5
	4				4	5
	5					5

Time

$O(n^3)$: for loops nested 3 deep, each loop index takes on $\leq n$ values. Can also show $\Omega(n^3)$. Therefore, $\Theta(n^3)$.

Step 4: Construct an optimal binary search tree

[Exercise 14.5-1 asks to write this pseudocode.]

```
CONSTRUCT-OPTIMAL-BST(root)
  r = root[1, n]
  print "k"_r "is the root"
  CONSTRUCT-OPT-SUBTREE(1, r − 1, r, "left", root)
  CONSTRUCT-OPT-SUBTREE(r + 1, n, r, "right", root)

CONSTRUCT-OPT-SUBTREE(i, j, r, dir, root)
  if i ≤ j
      t = root[i, j]
      print "k"_t "is" dir "child of k"_r
      CONSTRUCT-OPT-SUBTREE(i, t − 1, t, "left", root)
      CONSTRUCT-OPT-SUBTREE(t + 1, j, t, "right", root)
```

Elements of dynamic programming

Mentioned already:

- optimal substructure
- overlapping subproblems

Optimal substructure

- Show that a solution to a problem consists of making a choice, which leaves one or more subproblems to solve.
- Suppose that you are given this last choice that leads to an optimal solution. *[We find that students often have trouble understanding the relationship between optimal substructure and determining which choice is made in an optimal solution. One way that helps them understand optimal substructure is to imagine that the dynamic-programming gods tell you what was the last choice made in an optimal solution.]*
- Given this choice, determine which subproblems arise and how to characterize the resulting space of subproblems.
- Show that the solutions to the subproblems used within the optimal solution must themselves be optimal. Usually use cut-and-paste:
  - Suppose that one of the subproblem solutions is not optimal.
  - *Cut* it out.
  - *Paste* in an optimal solution.
  - Get a better solution to the original problem. Contradicts optimality of problem solution.

That was optimal substructure.

Need to ensure that you consider a wide enough range of choices and subproblems that you get them all. *[The dynamic-programming gods are too busy to tell you what that last choice really was.]* Try all the choices, solve all the subproblems resulting from each choice, and pick the choice whose solution, along with subproblem solutions, is best.

How to characterize the space of subproblems?

- Keep the space as simple as possible.
- Expand it as necessary.

Examples

Rod cutting

- Space of subproblems was rods of length $n - i$, for $1 \le i \le n$.
- No need to try a more general space of subproblems.

Matrix-chain multiplication

- Suppose we had tried to constrain the space of subproblems to parenthesizing $A_1 A_2 \cdots A_j$.
- An optimal parenthesization splits at some matrix A_k.
- Get subproblems for $A_1 \cdots A_k$ and $A_{k+1} \cdots A_j$.
- Unless we could guarantee that $k = j - 1$, so that the subproblem for $A_{k+1} \cdots A_j$ has only A_j, then this subproblem is *not* of the form $A_1 A_2 \cdots A_j$.
- Thus, needed to allow the subproblems to vary at both ends—allow both i and j to vary.

Longest commmon subsequence

- Space of subproblems for $\langle x_1, \ldots, x_i \rangle$ and $\langle y_1, \ldots, y_j \rangle$ was just $\langle x_1, \ldots, x_{i-1} \rangle$ and $\langle y_1, \ldots, y_{j-1} \rangle$.
- No need to try a more general space of subproblems.

Optimal binary search trees

- Similar to matrix-chain multiplication.
- Suppose we had tried to constrain space of subproblems to subtrees with keys $k_1, k_2, \ldots, k_j$.
- An optimal BST would have root k_r, for some $1 \le r \le j$.
- Get subproblems $k_1, \ldots, k_{r-1}$ and $k_{r+1}, \ldots, k_j$.
- Unless we could guarantee that $r = j$, so that subproblem with $k_{r+1}, \ldots, k_j$ is empty, then this subproblem is *not* of the form $k_1, k_2, \ldots, k_j$.
- Thus, needed to allow the subproblems to vary at "both ends," i.e., allow both i and j to vary.

Optimal substructure varies across problem domains:

1. *How many subproblems* are used in an optimal solution.
2. *How many choices* in determining which subproblem(s) to use.

- Rod cutting:
  - 1 subproblem (of size $n - i$)
  - n choices
- Matrix-chain multiplication:
  - 2 subproblems ($A_i \cdots A_k$ and $A_{k+1} \cdots A_j$)
  - $j - i$ choices for A_k in $A_i, A_{i+1}, \ldots, A_{j-1}$. Having found optimal solutions to subproblems, choose from among the $j - i$ candidates for A_k.
- Longest common subsequence:
  - 1 subproblem
  - Either
    - 1 choice (if $x_i = y_j$, LCS of X_{i-1} and Y_{j-1}), or
    - 2 choices (if $x_i \ne y_j$, LCS of X_{i-1} and Y, and LCS of X and Y_{j-1})
- Optimal binary search tree:
  - 2 subproblems ($k_i, \ldots, k_{r-1}$ and $k_{r+1}, \ldots, k_j$)
  - $j - i + 1$ choices for k_r in $k_i, \ldots, k_j$. Having found optimal solutions to subproblems, choose from among the $j - i + 1$ candidates for k_r.

Informally, running time depends on (# of subproblems overall) $\times$ (# of choices).

- Rod cutting: $\Theta(n)$ subproblems, $\le n$ choices for each
  $\Rightarrow O(n^2)$ running time.
- Matrix-chain multiplication: $\Theta(n^2)$ subproblems, $O(n)$ choices for each
  $\Rightarrow O(n^3)$ running time.

- Longest common subsequence: $\Theta(mn)$ subproblems, ≤ 2 choices for each $\Rightarrow \Theta(mn)$ running time.
- Optimal binary search tree: $\Theta(n^2)$ subproblems, $O(n)$ choices for each $\Rightarrow O(n^3)$ running time.

Can use the subproblem graph to get the same analysis: count the number of edges.

- Each vertex corresponds to a subproblem.
- Choices for a subproblem are vertices that the subproblem has edges going to.
- For rod cutting, subproblem graph has n vertices and $\leq n$ edges per vertex $\Rightarrow O(n^2)$ running time.
  In fact, can get an exact count of the edges: for $i = 0, 1, \ldots, n$, vertex for subproblem size i has out-degree $i \Rightarrow$ # of edges $= \sum_{i=0}^{n} i = n(n+1)/2$.
- Subproblem graph for matrix-chain multiplication has $\Theta(n^2)$ vertices, each with degree $\leq n - 1$
  $\Rightarrow O(n^3)$ running time.

Dynamic programming uses optimal substructure *bottom up*.

- *First* find optimal solutions to subproblems.
- *Then* choose which to use in optimal solution to the problem.

When we look at greedy algorithms, we'll see that they work *top down*: *first* make a choice that looks best, *then* solve the resulting subproblem.

Don't be fooled into thinking optimal substructure applies to all optimization problems. It doesn't.

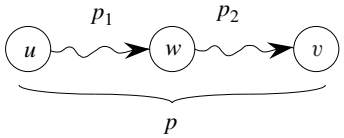
Here are two problems that look similar. In both, we're given an *unweighted, directed graph* $G = (V, E)$.

- V is a set of *vertices*.
- E is a set of *edges*.

And we ask about finding a ***path*** (sequence of connected edges) from vertex u to vertex v.

- ***Shortest path***: find a path $u \rightsquigarrow v$ with fewest edges. Must be ***simple*** (no *cycles*), since removing a cycle from a path gives a path with fewer edges.
- ***Longest simple path***: find a *simple* path $u \rightsquigarrow v$ with most edges. If didn't require simple, could repeatedly traverse a cycle to make an arbitrarily long path.

Shortest path has optimal substructure.



- Suppose p is shortest path $u \rightsquigarrow v$.
- Let w be any vertex on p.
- Let p_1 be the portion of p going $u \rightsquigarrow w$.
- Then p_1 is a shortest path $u \rightsquigarrow w$.

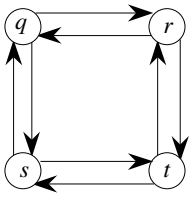
Proof Suppose there exists a shorter path p_1' going $u \rightsquigarrow w$. Cut out p_1, replace it with p_1', get path $u \overset{p_1'}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$ with fewer edges than p. ■

Therefore, can find shortest path $u \rightsquigarrow v$ by considering all intermediate vertices w, then finding shortest paths $u \rightsquigarrow w$ and $w \rightsquigarrow v$.

Same argument applies to p_2.

Does longest path have optimal substructure?

- It seems like it should.
- It does *not*.



Consider $q \rightarrow r \rightarrow t$ = longest path $q \rightsquigarrow t$. Are its subpaths longest paths?

No!

- Subpath $q \rightsquigarrow r$ is $q \rightarrow r$.
- Longest simple path $q \rightsquigarrow r$ is $q \rightarrow s \rightarrow t \rightarrow r$.
- Subpath $r \rightsquigarrow t$ is $r \rightarrow t$.
- Longest simple path $r \rightsquigarrow t$ is $r \rightarrow q \rightarrow s \rightarrow t$.

Not only isn't there optimal substructure, but can't even assemble a legal solution from solutions to subproblems.

Combine longest simple paths:

$q \rightarrow s \rightarrow t \rightarrow r \rightarrow q \rightarrow s \rightarrow t$

Not simple!

In fact, this problem is NP-complete (so it probably has no optimal substructure to find.)

What's the big difference between shortest path and longest path?

- Shortest path has ***independent*** subproblems.
- Solution to one subproblem does not affect solution to another subproblem of the same problem.
- Longest simple path: subproblems are *not* independent.
- Consider subproblems of longest simple paths $q \rightsquigarrow r$ and $r \rightsquigarrow t$.
- Longest simple path $q \rightsquigarrow r$ uses s and t.
- Cannot use s and t to solve longest simple path $r \rightsquigarrow t$, since if you do, the path isn't simple.
- But you *have* to use t to find longest simple path $r \rightsquigarrow t$!

- Using resources (vertices) to solve one subproblem renders them unavailable to solve the other subproblem.

  [For shortest paths, for a shortest path $u \overset{p_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$, no vertex other than w can appear in p_1 and p_2. Otherwise, get a cycle.]

Independent subproblems in our examples:

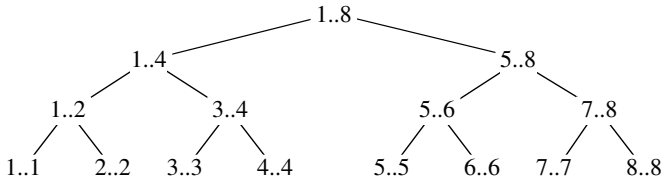
- Rod cutting and longest common subsequence
  - 1 subproblem $\Rightarrow$ automatically independent.
- Matrix-chain multiplication
  - $A_i \cdots A_k$ and $A_{k+1} \cdots A_j \Rightarrow$ independent.
- Optimal binary search tree
  - $k_i, \ldots, k_{r-1}$ and $k_{r+1}, \ldots, k_j \Rightarrow$ independent.

Overlapping subproblems

These occur when a recursive algorithm revisits the same problem over and over.

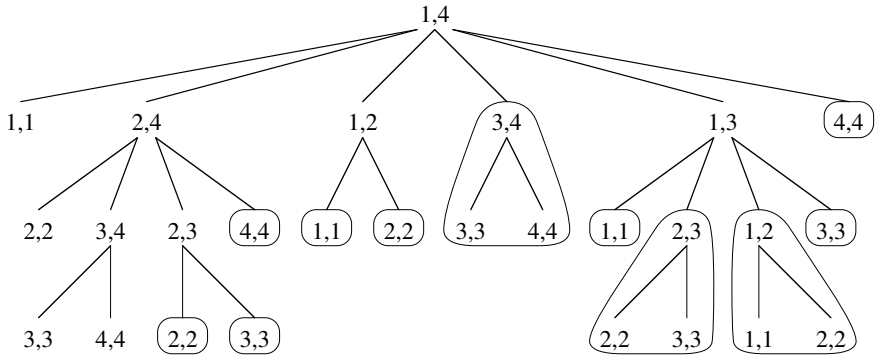
Good divide-and-conquer algorithms usually generate a brand new problem at each stage of recursion.

Example: merge sort



Alternative approach to dynamic programming: ***memoization***

- "Store, don't recompute."
- Make a table indexed by subproblem.
- When solving a subproblem:
  - Lookup in table.
  - If answer is there, use it.
  - Else, compute answer, then store it.
- For matrix-chain multiplication:



Each node has the parameters i and j. Computations performed in highlighted subtrees are replaced by a single table lookup if computing recursively with memoization.

- In bottom-up dynamic programming, we go one step further. Determine in what order to access the table, and fill it in that way.

3 Chapter 15: Greedy algorithms

Reading: 15.1–15.3

Lecture Notes for Chapter 15: Greedy Algorithms

[The fourth edition removed the starred sections on matroids and task scheduling (an application of matroids). These sections were replaced by a new, unstarred section covering offline caching, which had been the subject of Problem 16-5 in the third edition.]

Chapter 15 overview

Similar to dynamic programming.

Used for optimization problems.

Idea

When you have a choice to make, make the one that looks best *right now*. Make a *locally optimal choice* in hope of getting a *globally optimal solution*.

Greedy algorithms don't always yield an optimal solution. But sometimes they do. We'll see a problem for which they do. Then we'll look at some general characteristics of when greedy algorithms give optimal solutions. We then study two other applications of the greedy method: Huffman coding and offline caching. *[Later chapters use the greedy method as well: minimum spanning tree, Dijkstra's algorithm for single-source shortest paths, and a greedy set-covering heuristic.]*

Activity selection

n ***activities*** require *exclusive* use of a common resource. For example, scheduling the use of a classroom.

Set of activities $S = \{a_1, \ldots, a_n\}$.

a_i needs resource during period $[s_i, f_i)$, which is a half-open interval, where $s_i =$ start time and $f_i =$ finish time.

Goal

Select the largest possible set of nonoverlapping (***mutually compatible***) activities.

Could have many other objectives:

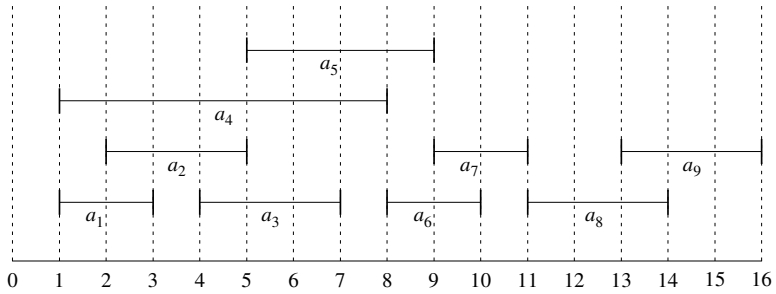
- Schedule room for longest time.
- Maximize income rental fees.

Assume that activities are sorted by finish time: $f_1 \le f_2 \le f_3 \le \cdots \le f_{n-1} \le f_n$.

Example

S sorted by finish time: *[Leave on board]*

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



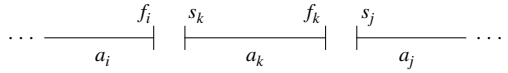
Maximum-size mutually compatible set: $\{a_1, a_3, a_6, a_8\}$.

Not unique: also $\{a_1, a_3, a_6, a_9\}$, $\{a_1, a_3, a_7, a_8\}$, $\{a_1, a_3, a_7, a_9\}$, $\{a_1, a_5, a_7, a_8\}$, $\{a_1, a_5, a_7, a_9\}$, $\{a_2, a_5, a_7, a_8\}$, $\{a_2, a_5, a_7, a_9\}$.

Optimal substructure of activity selection

$S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\}$ *[Leave on board]*

$=$ activities that start after a_i finishes and finish before a_j starts .



Activities in S_{ij} are compatible with

- all activities that finish by f_i, and
- all activities that start no earlier than s_j.

Let A_{ij} be a maximum-size set of mutually compatible activities in S_{ij}.

Let $a_k \in A_{ij}$ be some activity in A_{ij}. Then we have two subproblems:

- Find mutually compatible activities in S_{ik} (activities that start after a_i finishes and that finish before a_k starts).
- Find mutually compatible activities in S_{kj} (activities that start after a_k finishes and that finish before a_j starts).

Let

$A_{ik} = A_{ij} \cap S_{ik} =$ activities in A_{ij} that finish before a_k starts ,

$A_{kj} = A_{ij} \cap S_{kj} =$ activities in A_{ij} that start afer a_k finishes .

Then $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$

$\Rightarrow |A_{ij}| = |A_{ik}| + |A_{kj}| + 1$.

Claim

Optimal solution A_{ij} must include optimal solutions for the two subproblems for S_{ik} and S_{kj}.

Proof of claim Use the usual cut-and-paste argument. Will show the claim for S_{kj}; proof for S_{ik} is symmetric.

Suppose we could find a set A'_{kj} of mutually compatible activities in S_{kj}, where $|A'_{kj}| > |A_{kj}|$. Then use A'_{kj} instead of A_{kj} when solving the subproblem for S_{ij}. Size of resulting set of mutually compatible activities would be $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A|$. Contradicts assumption that A_{ij} is optimal. ■ (claim)

One recursive solution

Since optimal solution A_{ij} must include optimal solutions to the subproblems for S_{ik} and S_{kj}, could solve by dynamic programming.

Let $c[i, j] =$ size of optimal solution for S_{ij}. Then

$$c[i, j] = c[i, k] + c[k, j] + 1 \ .$$

But we don't know which activity a_k to choose, so we have to try them all:

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \ , \\ \max \{c[i, k] + c[k, j] + 1 : a_k \in S_{ij}\} & \text{if } S_{ij} \neq \emptyset \ . \end{cases}$$

Could then develop a recursive algorithm and memoize it. Or could develop a bottom-up algorithm and fill in table entries.

Instead, we will look at a greedy approach.

Making the greedy choice

Choose an activity to add to optimal solution *before* solving subproblems. For activity-selection problem, we can get away with considering only the greedy choice: the activity that leaves the resource available for as many other activities as possible.

Question: Which activity leaves the resource available for the most other activities? Answer: The first activity to finish. (If more than one activity has earliest finish time, can choose any such activity.)

Since activities are sorted by finish time, just choose activity a_1.

That leaves only one subproblem to solve: finding a maximum size set of mutually compatible activities that start after a_1 finishes. (Don't have to worry about activities that finish before a_1 starts, because $s_1 < f_1$ and no activity a_i has finish time $f_i < f_1 \Rightarrow$ no activity a_i has $f_i \leq s_1$.)

Since have only subproblem to solve, simplify notation:

$$S_k = \{a_i \in S : s_i \geq f_k\} = \text{activities that start after } a_k \text{ finishes} \ .$$

Making greedy choice of $a_1 \Rightarrow S_1$ remains as only subproblem to solve. *[Slight abuse of notation: referring to S_k not only as a set of activities but as a subproblem consisting of these activities.]*

By optimal substructure, if a_1 is in an optimal solution, then an optimal solution to the original problem consists of a_1 plus all activities in an optimal solution to S_1.

But need to prove that a_1 is always part of some optimal solution.

Theorem

If S_k is nonempty and a_m has the earliest finish time in S_k, then a_m is included in some optimal solution.

Proof Let A_k be an optimal solution to S_k, and let a_j have the earliest finish time of any activity in A_k. If $a_j = a_m$, done. Otherwise, let $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be A_k but with a_m substituted for a_j.

Claim

Activities in A'_k are disjoint.

Proof of claim Activities in A_k are disjoint, a_j is first activity in A_k to finish, and $f_m \le f_j$. ■ (claim)

Since $|A'_k| = |A_k|$, conclude that A'_k is an optimal solution to S_k, and it includes a_m. ■ (theorem)

So, don't need full power of dynamic programming. Don't need to work bottom-up.

Instead, can just repeatedly choose the activity that finishes first, keep only the activities that are compatible with that one, and repeat until no activities remain.

Can work top-down: make a choice, then solve a subproblem. Don't have to solve subproblems before making a choice.

Recursive greedy algorithm

Start and finish times are represented by arrays s and f, where f is assumed to be already sorted in monotonically increasing order.

To start, add fictitious activity a_0 with $f_0 = 0$, so that $S_0 = S$, the entire set of activities.

Procedure RECURSIVE-ACTIVITY-SELECTOR takes as parameters the arrays s and f, index k of current subproblem, and number n of activities in the original problem.

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
  m = k + 1
  while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
      m = m + 1
  if m ≤ n
      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
  else return ∅
```

Initial call

RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$.

Idea

The **while** loop checks $a_{k+1}, a_{k+2}, \ldots, a_n$ until it finds an activity a_m that is compatible with a_k (need $s_m \geq f_k$).

- If the loop terminates because a_m is found ($m \leq n$), then recursively solve S_m, and return this solution, along with a_m.
- If the loop never finds a compatible a_m ($m > n$), then just return empty set.

Go through example given earlier. Should get $\{a_1, a_3, a_6, a_8\}$.

Time

$\Theta(n)$—each activity examined exactly once, assuming that activities are already sorted by finish times.

Iterative greedy algorithm

Can convert the recursive algorithm to an iterative one. It's already almost tail recursive.

```
GREEDY-ACTIVITY-SELECTOR(s, f, n)
  A = {a_1}
  k = 1
  for m = 2 to n
      if s[m] ≥ f[k]          // is a_m in S_k?
          A = A ∪ {a_m}       // yes, so choose it
          k = m               // and continue from there
  return A
```

Go through example given earlier. Should again get $\{a_1, a_3, a_6, a_8\}$.

Time

$\Theta(n)$, if activities are already sorted by finish times.

For both the recursive and iterative algorithms, add $O(n \lg n)$ time if activities need to be sorted.

Elements of the greedy strategy

The choice that seems best at the moment is the one we go with.

What did we do for activity selection?

1. Determine the optimal substructure.

2. Develop a recursive solution.
3. Show that if you make the greedy choice, only one subproblem remains.
4. Prove that it's always safe to make the greedy choice.
5. Develop a recursive greedy algorithm.
6. Convert it to an iterative algorithm.

At first, it looked like dynamic programming. In the activity-selection problem, we started out by defining subproblems S_{ij}, where both i and j varied. But then found that making the greedy choice allowed us to restrict the subproblems to be of the form S_k.

Could instead have gone straight for the greedy approach: in our first crack at defining subproblems, use the S_k form. Could then have proven that the greedy choice a_m (the first activity to finish), combined with optimal solution to the remaining compatible activities S_m, gives an optimal solution to S_k.

Typically, we streamline these steps:

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, combining an optimal solution to the remaining subproblem with the greedy choice gives an optimal solution to the original problem.

No general way to tell whether a greedy algorithm is optimal, but two key ingredients are

1. greedy-choice property and
2. optimal substructure.

Greedy-choice property

Can assemble a globally optimal solution by making locally optimal (greedy) choices.

Dynamic programming

- Make a choice at each step.
- Choice depends on knowing optimal solutions to subproblems. Solve subproblems *first*.
- Solve *bottom-up* (unless memoizing).

Greedy

- Make a choice at each step.
- Make the choice *before* solving the subproblems.
- Solve *top-down*.

Typically show the greedy-choice property by what we did for activity selection:

- Look at an optimal solution.
- If it includes the greedy choice, done.
- Otherwise, modify the optimal solution to include the greedy choice, yielding another solution that's just as good.

Can get efficiency gains from greedy-choice property.

- Preprocess input to put it into greedy order.
- Or, if dynamic data, use a priority queue.

Optimal substructure

Just show that optimal solution to subproblem and greedy choice $\Rightarrow$ optimal solution to problem.

Greedy vs. dynamic programming

The knapsack problem is a good example of the difference.

0-1 knapsack problem

- n items.
- Item i is worth $\$v_i$, weighs w_i pounds.
- Find a most valuable subset of items with total weight $\leq W$.
- Have to either take an item or not take it—can't take part of it.

Fractional knapsack problem

Like the 0-1 knapsack problem, but can take fraction of an item.

Both have optimal substructure.

But the fractional knapsack problem has the greedy-choice property, and the 0-1 knapsack problem does not.

To solve the fractional problem, rank items by value/weight: v_i/w_i. Let $v_i/w_i \geq v_{i+1}/w_{i+1}$ for all i. Take items in decreasing order of value/weight. Will take all of the items with the greatest value/weight, and possibly a fraction of the next item.

```
FRACTIONAL-KNAPSACK(v, w, W)
  load = 0
  i = 1
  while load < W and i ≤ n
      if w_i ≤ W − load
          take all of item i
      else take (W − load)/w_i of item i
      add what was taken to load
      i = i + 1
```

Time: $O(n \lg n)$ to sort, $O(n)$ thereafter.

Greedy doesn't work for the 0-1 knapsack problem. Might get empty space, which lowers the average value per pound of the items taken.

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i/w_i	6	5	4

$W = 50$.

Greedy solution:

- Take items 1 and 2.
- value = 160, weight = 30.

Have 20 pounds of capacity left over.

Optimal solution:

- Take items 2 and 3.
- value = 220, weight = 50.

No leftover capacity.

Huffman codes

Goal: Compress a data file made up of characters. You know how often each character appears in the file—its ***frequency***. Each character is represented by some bit sequence: a ***codeword***. Use as few bits as possible to represent the file.

Fixed-length code: All codewords have the same number of bits. For $n \geq 2$ characters, need $\lceil \lg n \rceil$ bits.

Variable-length code: Represent different characters with differing numbers of bits. In particular, give frequently occurring characters shorter codewords and infrequently occurring characters longer codewords.

Example: For a data file of 100,000 characters:

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

For a fixed-length code, need 3 bits per character. For 100,000 characters, need 300,000 bits. For this variable-length code, need

$$\begin{aligned} 45{,}000 \cdot 1 &= 45{,}000 \\ +\ 13{,}000 \cdot 3 &= 39{,}000 \\ +\ 12{,}000 \cdot 3 &= 36{,}000 \\ +\ 16{,}000 \cdot 3 &= 48{,}000 \\ +\ 9{,}000 \cdot 4 &= 36{,}000 \\ +\ 5{,}000 \cdot 4 &= 20{,}000 \\ \hline &= 224{,}000 \text{ bits} \end{aligned}$$

Prefix-free codes

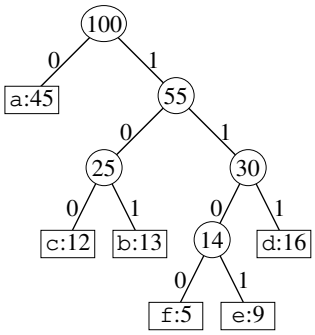
No codeword is also a prefix of any other codeword. *[Called "prefix codes" in earlier editions of the book. Changed to "prefix-free codes" in the fourth edition because each codeword is free of prefixes of other codes.]* A prefix-free code can always achieve the optimal compression.

Encoding: Just concatenate codewords for each character in the file. ***Example:*** To encode `face`: $1100 \cdot 0 \cdot 100 \cdot 1101 = 110001001101$, where $\cdot$ is concatenation.

Decoding: Since no codeword is a prefix of any other codeword, just process bits until you get a match. Then discard the bits and go from the rest of the compressed file. ***Example:*** If encoding is 100011001101, get a match on $100 =$ `c`. That leaves 011001101. Get a match on $0 =$ `a`. That leaves 11001101. Get a match on $1100 =$ `f`. That leaves 1101. Get a match on $1101 =$ `e`. So the encoded file represents `cafe`.

Binary tree representation

Use a binary tree whose leaves are the characters. The codeword for a character is given by the simple path from the root down to that character's leaf, where going left is 0 and going right is 1.



Here, each leaf has its character and frequency (in thousands). Each internal node holds the sum of the frequencies of the leaves in its subtree.

An optimal code is always given by a full binary tree: each internal node has 2 children $\Rightarrow$ if C is the alphabet for the characters, then the tree has $|C|$ leaves and $|C| - 1$ internal nodes.

How to compute the number of bits to encode a file for alphabet C given tree T: For each character $c \in C$, denote its frequency by $c.freq$. Denote the depth of c in T by $d_T(c)$, which equals the length of c's codeword. Then the number of bits to encode the file, the ***cost*** of T, is

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c) \ .$$

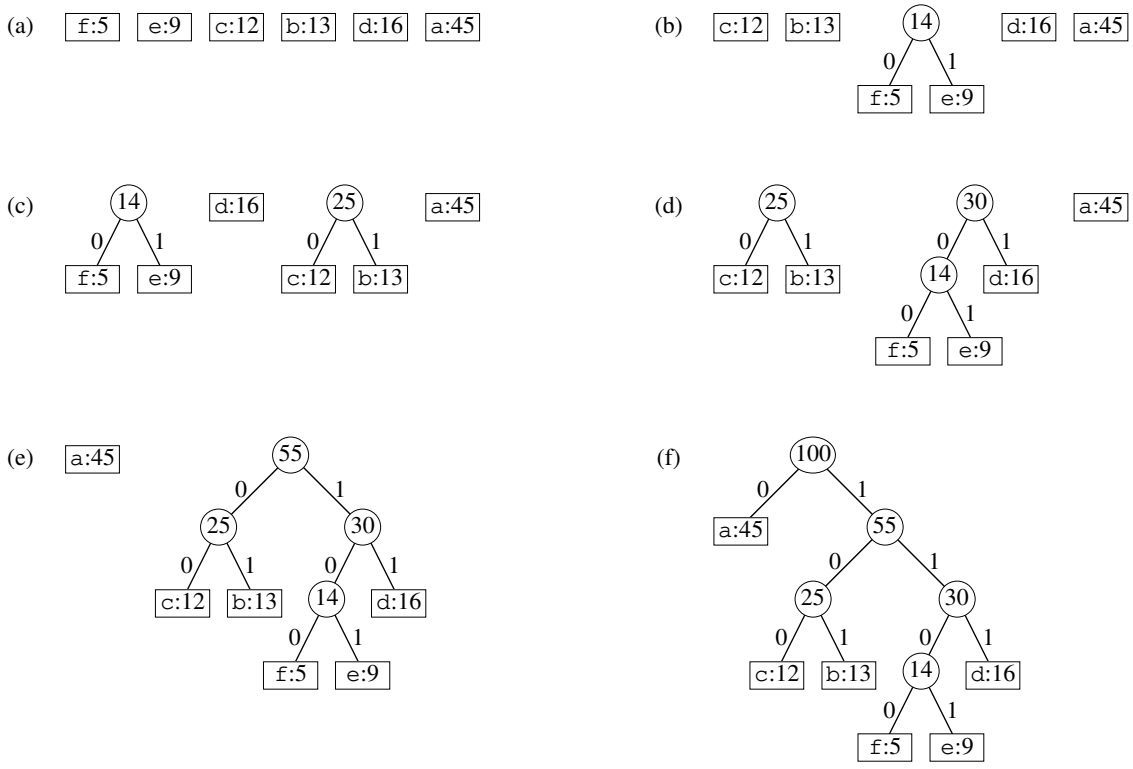
Constructing a Huffman code

[Named after David Huffman.] The algorithm builds tree T bottom-up. It repeatedly selects two nodes with the lowest frequency and makes them children of

a new node whose frequency is the sum of the two nodes' frequencies. It uses a min-priority queue Q keyed on the *freq* attribute, which all nodes have.

```
HUFFMAN(C)
  n = |C|
  Q = C
  for i = 1 to n − 1
      allocate a new node z
      x = EXTRACT-MIN(Q)
      y = EXTRACT-MIN(Q)
      z.left = x
      z.right = y
      z.freq = x.freq + y.freq
      INSERT(Q, z)
  return EXTRACT-MIN(Q)      // the root of the tree is the only node left
```

Example: Using the frequencies from before:



Running time: Let $n = |C|$. The running time depends on how the min-priority queue Q is implemented. If with a binary min-heap, can initialize Q in $O(n)$ time. The **for** loop runs $n - 1$ times, and each INSERT and EXTRACT-MIN call takes $O(\lg n)$ time $\Rightarrow O(n \lg n)$ time in all.

Correctness

Show the greedy-choice and optimal-substructure properties.

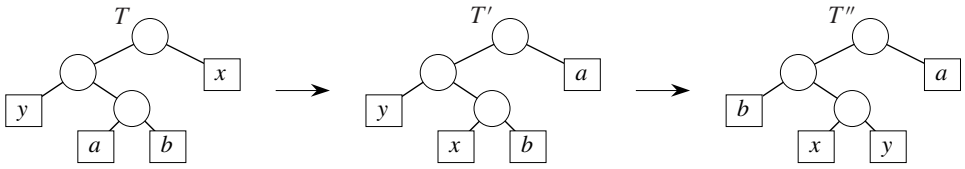
Lemma (Greedy-choice property)
For alphabet C, let x and y be the two characters with the lowest frequencies. Then there exists an optimal prefix-free code for C where the codewords for x and y have the same length and differ only in the last bit.

Proof Given a tree T for some optimal prefix-free code, modify it so that x and y are sibling leaves of maximum depth. Then the codewords for x and y will have the same length and differ in the last bit.

Let a, b be two characters that are sibling leaves of maximum depth in T. Assume wlog that $a.freq \le b.freq$ and $x.freq \le y.freq$. Must have $x.freq \le a.freq$ and $y.freq \le b.freq$.

Could have $x.freq = a.freq$ or $y.freq = b.freq$. If $x.freq = b.freq$, then $a.freq = b.freq = x.freq = y.freq$ (Exercise 15.3-1), and the lemma is trivially true. So assume that $x.freq \ne b.freq \Rightarrow x \ne b$.

In T: exchange a and x, producing T'.
In T': exchange b and y, producing T''.
In T'', x and y are sibling leaves of maximum depth.



Claim
$B(T') \le B(T)$. (Exchanging a and x does not increase the cost.)

Proof of claim

$$\begin{aligned}
&B(T) - B(T') \\
&\quad = \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\
&\quad = x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\
&\quad = x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\
&\quad = (a.freq - x.freq)(d_T(a) - d_T(x)) \\
&\quad \ge 0 \,.
\end{aligned}$$

The last line follows because $x.freq \le a.freq$ and a is a maximum-depth leaf $\Rightarrow$ $d_T(a) \ge d_T(x)$. ■ (claim)

Similarly, $B(T'') \le B(T')$ because exchanging y and b doesn't increase the cost. Therefore, $B(T'') \le B(T') \le B(T)$. T is optimal $\Rightarrow B(T) \le B(T'') \Rightarrow B(T'') = B(T) \Rightarrow T''$ is optimal, and x and y are sibling leaves of maximum depth. ■

The lemma shows that to build up an optimal tree, can begin with the greedy choice of merging the two characters with lowest frequency. Greedy because the cost of a merger is the sum of the frequencies of its children and the cost of a tree equals the sum of the costs of its mergers (Exercise 15.3-4).

Lemma (Optimal-substructure property)

For alphabet C, let x, y be the two characters with minimum frequency. Let $C' = (C - \{x, y\}) \cup z$ for a new character z with $z.freq = x.freq + y.freq$. Let T' be a tree representing an optimal prefix-free code for C', and T be T' with the leaf for z replaced by an internal node with children x and y. Then T represents an optimal prefix-free code for C.

Proof $c \in C - \{x, y\} \Rightarrow d_T(c) = d_{T'}(c) \Rightarrow c.freq \cdot d_T(c) = c.freq \cdot d_{T'}(c)$. $d_T(x) = d_T(y) = d_{T'}(z) + 1 \Rightarrow$

$$\begin{aligned} x.freq \cdot d_T(x) + y.freq \cdot d_T(y) &= (x.freq + y.freq)(d_{T'}(z) + 1) \\ &= z.freq \cdot d_{T'}(z) + (x.freq + y.freq) , \end{aligned}$$

so that $B(T) = B(T') + x.freq + y.freq$, which is equivalent to $B(T') = B(T) - x.freq - y.freq$.

Now suppose T doesn't represent an optimal prefix-free code for C. Then $B(T'') < B(T)$ for some optimal tree T''. By the previous lemma, without loss of generality, T'' has x and y as siblings. Replace the common parent of x and y by a leaf z with $z.freq = x.freq + y.freq$ and call the resulting tree T'''. Then,

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T') , \end{aligned}$$

so that T' was not optimal, a contradiction. ■

Theorem

HUFFMAN produces an optimal prefix-free code.

Proof The greedy-choice and optimal-substructure properties both apply. ■

Offline caching

In a computer, a ***cache*** is memory that is smaller but faster than main memory. It holds a small subset of what's in main memory. Caches store data in ***blocks***, also known as ***cache lines***, usually 32, 64, or 128 bytes. *[We use the term* blocks *in this discussion, rather than* cache lines.*]*

A program makes a sequence of memory requests to blocks. Each block usually has several requests to some data that it holds.

The cache size is limited to k blocks, starting out empty before the first request. Each request causes either 0 or 1 block to enter the cache, and either 0 or 1 block to be evicted. A request for block b may have one of three outcomes:

1. b is already in the cache due to some previous request $\Rightarrow$ ***cache hit***. The cache remains unchanged.
2. b is not already in the cache, but the cache is not yet full (contains $< k$ blocks). b goes into the cache, so that the cache now contains one more block than before the request.

4 Chapter 5 & 9: Randomized algorithms

Reading: Chapter 5; Sections 9.1, 9.2

4.1 Class notes

4.1.1 Introduction

Randomized algorithms (like dynamic or greedy) refer to a general design technique. Unfortunately, analysis of these algorithms requires many tools from probability. We could spend the whole course on randomized algorithms. We will cover a selection of topics beyond what was shown in 305.

Topics to cover:

- Random permutations (Ch. 5)
- QuickSelect (Ch. 9: 9.1, 9.2)
- Filtering: Bloom, CountMinSketch

Overall, I am indebted to Jeff Erickson for much of this material: `https://jeffe.cs.illinois.edu/teaching/algorithms/`

Background knowledge that I assume:

- Definition of random variables (r.v.'s)
- Independence
- Expectation, linearity
- Conditional probabilities
- Indicator random variables
- Coin flips (Bernoulli r.v.)
- Balls in bins (Binomial r.v.)
- QuickSort with random pivots
- Hashing, uniform random analysis

An example of things you should be comfortable computing:

- What is the probability that an item hashes into slot 1 when there are m total slots?

  Answer: $1/m$.

- Let $X = I\{\text{item hashes into slot 10}\}$, an indicator variable, compute $\mathbb{E}[X]$.

  Answer: $\mathbb{E}[X] = \Pr(\text{item hashes into slot 10}) = 1/m$ if there are m slots. This random variable is like a biased coin flip.

- What is the distribution of chain lengths in a hash table with m slots and n items?

  Answer: This is like placing n balls (items) into m bins (slots). The number of balls in a given bin is a Binomial random variable where we perform n trials and have a probability $1/m$ of success (landing in that bin). If X_i for $i = 1, \ldots, m$ is the count of balls in bin i, then $X_i = \text{Binom}(n, 1/m)$. The X_i are *not* independent; the joint distribution of all chain lengths is known as the multinomial distribution.

Pass out handout "randomized 1." The first side covers von Neumann's de-biasing algorithm

Runtime? Let T be the number of coin tosses before the algorithm exits. Then

$$\mathbb{E}[T] = \mathbb{E}[T|x \neq y]\Pr[x \neq y] + \mathbb{E}[T|x = y]\Pr[x = y].$$

The two coins to come up differently with probability $2p(1-p) = \Pr[x \neq y]$, which implies $\Pr[x = y] = 1 - 2p(1-p)$. When they come up same, the algorithm exits after those 2 tosses. When they come up different, the algorithm runs again after those 2 tosses. Putting this all together, we get

$$\mathbb{E}[T] = 4p(1-p) + (2 + \mathbb{E}[T])(1 - 2p(1-p)).$$

Going through the algebra, we can solve for $\mathbb{E}[T] = \frac{1}{p(1-p)}$.

4.1.2 Random permutations

Often in CS we want to generate combinatorial objects. Enumerating all of these is usually intractible because there are too many. Luckily, we can usually do better if we want just a *random sample* of a combinatorial object.

An example would be random permutations of $\{1, \ldots, n\}$. We know there are $n!$ of these, which is more than exponential, too many to enumerate for even moderate n. Any correct algorithm for generating these permutations must output a given permutation with probablity $1/n!$.

A basic algorithm would be to generate a random integer between 1 and $n!$, then to somehow map that number onto the given permutation. Since $\log(n!) = \Theta(n \log n)$, this could require $\Theta(n \log n)$ many digits in any number base. If you are working with `int` or similar finite-precision data types, you are going to run into trouble pretty quickly.

There are instead better methods that give $O(n)$ performance and directly output the permutation as a list of numbers. These methods work like an old-school lottery:

1. Place the n numbers into an urn.
2. Draw numbers from the urn and output the sequence.

This is called "drawing lots." There are n choices for the first number, $n-1$ for the second, etc. and thus $n!$ possible outcomes. This proves that drawing lots produces a correct permutation.

One possible way to implement this on the computer is

```
FisherYates(n):
  for i = 1 to n
    Chosen[i] = False
  for i = n down to 1
    repeat
      r = Random(1, n)
    until not Chosen[r]
    R[i] = r
    Chosen[r] = True
  return R
```

(According to Jeff Erickson, whose exposition I am following, this is the actual Fisher-Yates algorithm not the one that is presented in the book.) Since the algorithm implements lot-casting, it is correct. However, the inner "repeat" loop may need to generate multiple random draws before it finds an item that is not collected. This becomes more common as i decreases. So what is the runtime?

This is an example of the *coupon collector's problem*. You are collecting coupons at random, and you'd like to collect all n of them. Every coupon you receive is one of the n with equal probability $1/n$. (Maybe you are collecting Pokémon cards in an ideal world where each card occurs with equal frequency in the decks.) Let $T(n)$ be the number of coupons you need to accumulate to get all n unique coupons, and call $T_i(n)$ the time it takes to see the ith unique card. Then

$$T(n) = \sum_{i=1}^{n} T_i(n)$$

Assume we've seen $i-1$ unique cards and we are waiting for the ith new card. The next card is new with probability $p = \frac{n-i+1}{n}$. We need to keep drawing cards until this is new, which is like flipping coins until you see the first head. The expected time for this is $1/p = \frac{n}{n-i+1}$. (Prove it. You can also treat this as a geometric random variable, see Section C.4.) Using this fact, we get that

$$\mathbb{E}[T(n)] = \sum_{i=1}^{n} \mathbb{E}[T_i(n)] = \sum_{i=1}^{n} \frac{n}{n-i+1} = \sum_{j=1}^{n} \frac{n}{j} = nH_n.$$

This expression uses the nth Harmonic number $H_n := \sum_{i=1}^{n} 1/i = \Theta(\log n)$ (see Section A.1). Finally, this implies that the expected number of coupons we need to collect to get all n is $\Theta(n \log n)$.

Interpretation: You have to collect slightly more—a logarithmic factor—than the number of coupons to "catch 'em all."

Here is a better algorithm (called Fisher-Yates in our book but apparently due to Durstenfeld):

```
SelectionShuffle(A[1:n]):
  for i = n down to 1
    swap A[i] with A[Random(1, i)]
```

Here we can see, again, that there are n choices for the first swap, $n-1$ for the second, etc. So SelectionShuffle generates $n!$ possible permutations with equal probability. Another way to prove its correctness is to observe it's actually a case of lot-casting: We swap A[i] with a random element from $1, \ldots, i$. So, A[1:i] represents the urn of available elements. After the swap, the subarray A[i:n] plays the role of the lots that have already been chosen (they aren't modified again). By this mapping to the problem of drawing lots, we show its correctness.

See the book for a different kind of direct proof. Exercise: Show that SelectionShuffle is the same as Randomly-Permute presented in the book.

4.1.3 QuickSelect

Chapter 9 covers the randomized algorithm QuickSelect but refers to it as Randomized-Select.

The selection problem takes in an unsorted array of size n and an integer $1 \leq k \leq n$. The goal is to output the kth smallest element of the array, also called the *kth order statistic*.

For example, $k = 1$ refers to the minimum of the array, $k = n$ is the maximum, and $k = \lfloor n/2 \rfloor$ is the median (technically, if n is even we can also take $k = n/2 + 1$ as the upper median). These are all useful for summarizing data, because they capture the ranges and middle of the dataset. Percentile p of a discrete dataset can be directly computed using the order statistic $\lfloor np \rfloor$.

A simple way to compute all order statistics is to sort the array using a comparison sort, requiring $O(n \log n)$ work. But we can do better if we want just one. For instance, min and max can be computed in $O(n)$ time. QuickSelect is an algorithm similar to QuickSort—it partitions the array based off of a random pivot—that can acheive $O(n)$ in expectation.

```
QuickSelect(A[1:n], k) :
  r = Partition(A[1:n], Random(1, n))
  if k < r
    return QuickSelect(A[1:r - 1], k)
  else if k > r
    return QuickSelect(A[r + 1:n], k - r)
  else
    return A[k]
```

How does it work? We pick some random index to be the pivot element. After calling partition, elements `A[1:r-1]` have values smaller than the pivot value `A[r]`, and elements `A[r+1:n]` have values larger than

`A[r]`. If $r = k$, we got lucky and the pivot is our order statistic! Otherwise, it's either in the left or right set of remaining elements. We just keep going until we find it.

Runtime? Proving this is kind of tricky (see the book), but here's intuition:

Suppose that each pivot is in the second or third quartiles if the elements were sorted, i.e. in the "middle half." Then at least 1/4 of the remaining elements are ignored in all future recursive calls, implying that at most 3/4 of the elements are still in play. Since Partition takes $O(n)$ time, the recurrence would be $T(n) \leq T(3n/4) + O(n)$, which has solution $O(n)$.

What if the pivot is not always in the middle half? Probability that it is in the middle half is 1/2. View selecting a pivot in the middle half as a fair coin flip. Then the number of trials before a success is a geometric distribution with expected value 2. So that half the time, 1/4 of the elements go out of play, and the other half of the time, as few as one element (worst case) goes out of play. But that just doubles the running time, so still expect $O(n)$.

4.1.4 Tail inequalities

So far we've been focused on analysis of the expected performance of algorithms. For instance, we expect $nH_n \leq n(\log n + 1)$ coupons are needed to get them all. How much worse can it be? Can we guarantee that we get all coupons with 99% probability?

This leads to the study of tail/concentration inequalities, bounds on the probability that a random variable ends up in the tail of its distribution.

Markov's inequality. Let Z be a non-negative integer random variable. For any real number $z > 0$, we have $\Pr[Z \geq z] \leq \mathbb{E}[Z]/z$.

Proof. Suppose we plot $\Pr[Z \geq z]$ as a function of z, as in Fig. 2. Because Z takes only integer values, the resulting curve consists of horizontal and vertical line segments. By splitting the region between this curve and the coordinate axes into horizontal rectangles, we see that the area of this region is exactly $\sum_z \Pr[Z \geq z] = \sum_z z\Pr[Z = z] = E[Z]$, because the function $\Pr[Z \geq z]$ only steps down when $\Pr[Z = z] > 0$. On the other hand, for any particular value of z, the rectangle with width z and height $\Pr[Z \geq z]$ (shaded darker in the figure) fits entirely under the curve. We conclude that $z\Pr[Z \geq z] \leq E[Z]$.

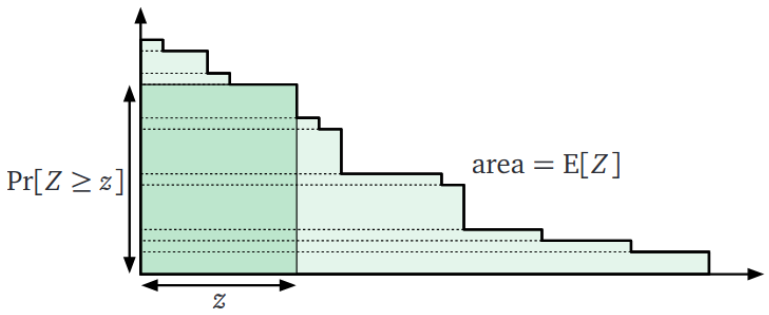


Figure 2: Markov's inequality.

Applying Markov's inequality to the coupon collector, we want to guarantee that we get all coupons with probability 99%. Equivalently, we fail with probability 1%. Then we want $Z = T(n)$ and $\Pr[T(n) \geq$ something$] \leq 1\%$. Seeing that we have $\mathbb{E}[Z]$ on the rhs of Markov's inequality, let's use $z = c\mathbb{E}[T(n)]$ to get $\Pr[T(n) \geq c\mathbb{E}[T(n)]] \leq 1/c$. This means we should take $c = 1/0.01 = 100$ to acheive the guarantee. Plugging in $n = 50$, we have that $\mathbb{E}[T(n)] = 225$, and so getting $100 \times 225 = 22,500$ coupons will guarantee that we get all 50.

This is pessimistic!

In reality, Markov's inequality is a very weak tail bound. Stronger inequalities that use the independence of random variables (Chebyshev's inequality) or moment bounds can be used to do better.

For coupon collecting, this direct computation (via wikipedia) is nice: Let Z_i^r indicate that the ith coupon was not picked in r trials. Then

$$\Pr[Z_i^r] = \left(1 - \frac{1}{n}\right)^r \leq e^{-r/n}$$

Setting $r = cn \log n$ gives $\Pr[Z_i^r] \leq n^- c$. For $T(n) \geq cn \log n$ to occur, we need Z_i^r to occur for some i between 1 and n. This means

$$\Pr[T(n) \geq cn \log n] = \Pr[Z_1^r \text{ or } \ldots Z_n^r] \leq nP[Z_1^r] \leq n^{1-c}.$$

This is a lot better than we had before. Let's plug in $c = 100$ from before. Then the probability of failure is less than n^{-99}, extremely small! Or, taking $n = 50$, we can solve for c to give us 1% failure and we need $50^{1-c} \leq 0.01$ or $c > 2.2$. Taking $c = 3$ gives 0.04% failure probability. Why did the above analysis do so much better? We had an exponential bound on each random event and we used independence!

Note: When the error probability falls off like $1/n^c$ we say the algorithm is correct *with high probability.*

4.1.5 Streaming and filtering algorithms

A data stream is an extremely long sequence S of items from some universe U that can be read only once, in order. Good examples of data streams include the sequence of packets that pass through a network router, the sequence of posts on a social media site, the sequence of all bids on the New York Stock Exchange, and the sequence of humans passing through the Shinjuku Railway Station in Tokyo. Standard algorithms are not appropriate for data streams; there is simply too much data to store, and it arrives too quickly for any complex computations.

Basic streaming algorithm:

```
DoSomething(S):
  ⟨⟨initialize⟩⟩
  while S is not done
    x = next item in S
    ⟨⟨do something fast with x⟩⟩
  return ⟨⟨something⟩⟩
```

As an example, say we want to estimate the number of times that an arbitrary item $x \in U$ has appeared in the stream so far. We will use the *Count-Min Sketch* algorithm.

The data structure consists of an $d \times m$ array of counters, initialized to 0, and d hash functions $h_1, \ldots, h_d : U \to \{1, \ldots, m\}$. (Technically, these hash functions should be drawn independently and uniformly at random from a family of hash functions $\mathcal{H}$ and be 2-uniform, meaning that for any two keys and any two hash functions, the probability of collision at any slot is $1/m^2$. This is possible to acheive!) There are two functions we use for our streaming algorithm:

```
CMIncrement(x):
  for i = 1 to d
    j = h_i(x)
    Count[i, j] = Count[i, j] + 1

CMEstimate(x):
  est = Inf
  for i = 1 to d
    j = h_i(x)
    est = min(est, Count[i, j])
  return est
```

Every time a new x arrives from the stream, we call `CMIncrement(x)`. When we want to estimate the count of x, we call `CMEstimate(x)`. Either function, incrementing or estimating, takes $O(d)$ time, and we use $O(md)$ space to store the table.

In words: Increment hashes the item d times and ups those counters. Estimate takes the minimum value stored in any counter and returns that.

We'll now show that this provides accurate estimates for our items. To specify the accuracy we want, we set a tolerance ϵ and an error rate δ. Let f_x be the true frequency of x among the items that we've seen, and let $\hat{f}_x$ be the value returned by the call to `CMEstimate`. Because of collisions, the counters may overestimate the true frequencies, but they are guaranteed to have been incremented every time x was seen. So $f_x \leq \hat{f}_x$; our estimate is never too small. We want to guarantee the following error bound:

$$\Pr[\hat{f}_x > f_x + \epsilon N] < \delta,$$

where N is the total length of the stream seen so far. This means we never overestimate too much, as controlled by the tolerance ϵ. (Error is additive for any item, so rare items may have relatively worse error than more frequent ones.)

For items $x \neq y$ and hash i, let $X_{i,x,y} = I[h_i(x) = h_i(y)]$ indicate a collision of those items at some slot. By 2-uniformity of our hash functions,

$$\mathbb{E}[X_{i,x,y}] = \Pr[h_i(x) = h_i(y)] = \frac{1}{m}.$$

Let $X_{i,x} = \sum_{y \neq x} X_{i,x,y} f_y$ be the total number of collisions with x in row i of the table. Then, if $j = h_i(x)$ is the slot, `Count[i,j]` $= f_x + X_{i,x} \geq f_x$. On the other hand, we have by linearity of expectation and $\sum_{y \neq x} f_y \leq N$,

$$\mathbb{E}[X_{i,x}] = \sum_{y \neq x} \mathbb{E}[X_{i,x,y}] f_y \leq \frac{N}{m}.$$

Since we take the minimum over all rows when constructing our estimate $\hat{f}_x$,

$$\begin{aligned}
\Pr[\hat{f}_x > f_x + \epsilon N] &= \Pr[X_{i,x} > \epsilon N \text{ for all } i] \\
&= \Pr[X_{1,x} > \epsilon N]^d \\
&\leq \left(\frac{\mathbb{E}[X_{1,x}]}{\epsilon N} \right)^d \\
&\leq \left(\frac{N/m}{\epsilon N} \right)^d = (m\epsilon)^{-d}
\end{aligned}$$

The first inequality in the chain follows from Markov.

Only now do we have the information we need to choose our m and d!

Setting $m = \lceil e/\epsilon \rceil$ and $d = \lceil \log(1/\delta) \rceil$ gives $\Pr[\hat{f}_x > f_x + \epsilon N] \leq (1/e)^{\log(1/\delta)} = \delta$.

To conclude, we get by with $O(\frac{1}{\epsilon} \log \frac{1}{\delta})$ space and $O(\log \frac{1}{\delta})$ time.

4.2 Ch. 5 material

Lecture Notes for Chapter 5: Probabilistic Analysis and Randomized Algorithms

[This chapter introduces probabilistic analysis and randomized algorithms. It assumes that the student is familiar with the basic probability material in Appendix C.

The primary goals of these notes are to

- *explain the difference between probabilistic analysis and randomized algorithms,*
- *present the technique of indicator random variables, and*
- *give another example of the analysis of a randomized algorithm (permuting an array in place).*

These notes omit the starred Section 5.4.]

The hiring problem

Scenario

- You are using an employment agency to hire a new office assistant.
- The agency sends you one candidate each day.
- You interview the candidate and must immediately decide whether or not to hire that person. But if you hire, you must also fire your current office assistant —even if it's someone you have recently hired.
- Cost to interview is c_i per candidate (interview fee paid to agency).
- Cost to hire is c_h per candidate (includes cost to fire current office assistant + hiring fee paid to agency).
- Assume that $c_h > c_i$.
- You are committed to having hired, at all times, the best candidate seen so far. Meaning that whenever you interview a candidate who is better than your current office assistant, you must fire the current office assistant and hire the candidate. Since you must have someone hired at all times, you will always hire the first candidate that you interview.

Goal

Determine what the price of this strategy will be.

Pseudocode to model this scenario

Assumes that the candidates are numbered 1 to n and that after interviewing each candidate, you can determine if they're better than the current office assistant. Uses a dummy candidate 0 that is worse than all others, so that the first candidate is always hired.

```
HIRE-ASSISTANT(n)
  best = 0          // candidate 0 is a least-qualified dummy candidate
  for i = 1 to n
      interview candidate i
      if candidate i is better than candidate best
          best = i
          hire candidate i
```

Cost

If n candidates, and you hire m of them, the cost is $O(nc_i + mc_h)$.

- Have to pay nc_i to interview, no matter how many you hire.
- So we focus on analyzing the hiring cost mc_h.
- mc_h varies with each run—it depends on the order in which you interview the candidates.
- This is a model of a common paradigm: need to find the maximum or minimum in a sequence by examining each element and maintaining a current "winner." The variable m denotes how many times we change our notion of which element is currently winning.

Worst-case analysis

In the worst case, you hire all n candidates.

This happens if each one is better than all who came before. In other words, if the candidates appear in increasing order of quality.

If you hire all n, then the cost is $O(c_i n + c_h n) = O(c_h n)$ (since $c_h > c_i$).

Probabilistic analysis

In general, you have no control over the order in which candidates appear.

We could assume that they come in a random order:

- Assign a rank to each candidate: $rank(i)$ is a unique integer in the range 1 to n.
- The ordered list $\langle rank(1), rank(2), \ldots, rank(n)\rangle$ is a permutation of the candidate numbers $\langle 1, 2, \ldots, n\rangle$.
- The list of ranks is equally likely to be any one of the $n!$ permutations.
- Equivalently, the ranks form a ***uniform random permutation***: each of the possible $n!$ permutations appears with equal probability.

Essential idea of probabilistic analysis

Use knowledge of, or make assumptions about, the distribution of inputs.

- The expectation is over this distribution.
- This technique requires that we can make a reasonable characterization of the input distribution.

Randomized algorithms

Might not know the distribution of inputs, or might not be able to model it computationally.

Instead, use randomization within the algorithm in order to impose a distribution on the inputs.

For the hiring problem

Change the scenario:

- The employment agency sends you a list of all n candidates in advance.
- On each day, you randomly choose a candidate from the list to interview (but considering only those not yet interviewed).
- Instead of relying on the candidates being presented in a random order, take control of the process and enforce a random order.

What makes an algorithm randomized

An algorithm is ***randomized*** if its behavior is determined in part by values produced by a ***random-number generator***.

- RANDOM(a, b) returns an integer r, where $a \le r \le b$ and each of the $b - a + 1$ possible values of r is equally likely.
- In practice, RANDOM is implemented by a ***pseudorandom-number generator***, which is a deterministic method returning numbers that "look" random and pass statistical tests.

Indicator random variables

A simple yet powerful technique for computing the expected value of a random variable. Provides an easy way to convert a probability to an expectation.

Helpful in situations in which there may be dependence.

Given a sample space and an event A, define the ***indicator random variable***

$$\mathrm{I}\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs ,} \\ 0 & \text{if } A \text{ does not occur .} \end{cases}$$

Lemma

For an event A, let $X_A = \mathrm{I}\{A\}$. Then $\mathrm{E}[X_A] = \Pr\{A\}$.

Proof Letting $\overline{A}$ be the complement of A, we have

$$\begin{aligned}
\mathrm{E}[X_A] &= \mathrm{E}[\mathrm{I}\{A\}] \\
&= 1 \cdot \Pr\{A\} + 0 \cdot \Pr\{\overline{A}\} \quad \text{(definition of expected value)} \\
&= \Pr\{A\} \ .
\end{aligned}$$

■ (lemma)

Simple example

Determine the expected number of heads from one flip of a fair coin.

- Sample space is $\{H, T\}$.
- $\Pr\{H\} = \Pr\{T\} = 1/2$.
- Define indicator random variable $X_H = \mathrm{I}\{H\}$. X_H counts the number of heads in one flip.
- Since $\Pr\{H\} = 1/2$, lemma says that $\mathrm{E}[X_H] = 1/2$.

Slightly more complicated example

Determine the expected number of heads in n coin flips.

- Let X be a random variable for the number of heads in n flips.
- Could compute $\mathrm{E}[X] = \sum_{k=0}^{n} k \cdot \Pr\{X = k\}$. In fact, this is what the book does in equation (C.41).
- Instead, use indicator random variables.
- For $i = 1, 2, \ldots, n$, define $X_i = \mathrm{I}\{\text{the } i\text{th flip results in event } H\}$.
- Then $X = \sum_{i=1}^{n} X_i$.
- Lemma says that $\mathrm{E}[X_i] = \Pr\{H\} = 1/2$ for $i = 1, 2, \ldots, n$.
- Expected number of heads is $\mathrm{E}[X] = \mathrm{E}[\sum_{i=1}^{n} X_i]$.
- ***Problem:*** We want $\mathrm{E}[\sum_{i=1}^{n} X_i]$. We have only the individual expectations $\mathrm{E}[X_1], \mathrm{E}[X_2], \ldots, \mathrm{E}[X_n]$.
- ***Solution:*** Linearity of expectation (equation (C.24)) says that the expectation of the sum equals the sum of the expectations. Thus,

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n} 1/2 \\
&= n/2 \ .
\end{aligned}$$

- Linearity of expectation applies even when there is dependence among the random variables. *[Not an issue in this example, but it can be a great help. The hat-check problem of Exercise 5.2-5 is a problem with lots of dependence.]*

Analysis of the hiring problem

Assume that the candidates arrive in a random order.

Let X be a random variable that equals the number of times you hire a new office assistant.

Define indicator random variables $X_1, X_2, \ldots, X_n$, where

$X_i = \mathrm{I}\{\text{candidate } i \text{ is hired}\}$.

Useful properties:

- $X = X_1 + X_2 + \cdots + X_n$.
- Lemma $\Rightarrow \mathrm{E}[X_i] = \Pr\{\text{candidate } i \text{ is hired}\}$.

Need to determine $\Pr\{\text{candidate } i \text{ is hired}\}$.

- Candidate i is hired if and only if candidate i is better than each of candidates $1, 2, \ldots, i-1$.
- Assumption that the candidates arrive in random order $\Rightarrow$ candidates $1, 2, \ldots, i$ arrive in random order $\Rightarrow$ any one of these first i candidates is equally likely to be the best one so far.
- Thus, $\Pr\{\text{candidate } i \text{ is the best so far}\} = 1/i$.
- Which, by the lemma, implies $\mathrm{E}[X_i] = 1/i$.

Now compute $\mathrm{E}[X]$:

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n} 1/i \\
&= \ln n + O(1) \quad \text{(equation (A.9): the sum is a harmonic series)} \ .
\end{aligned}$$

Thus, the expected hiring cost is $O(c_h \ln n)$, which is much better than the worst-case cost of $O(c_h n)$.

Randomized algorithms

Instead of assuming a distribution of the inputs, impose a distribution.

The hiring problem

For the hiring problem, the algorithm is deterministic:

- For any given input, the number of times you hire a new office assistant will always be the same.

- The number of times you hire a new office assistant depends only on the input.
- In fact, it depends only on the ordering of the candidates' ranks that it is given.
- Some rank orderings will always produce a high hiring cost. Example: $\langle 1, 2, 3, 4, 5, 6\rangle$, where each candidate is hired.
- Some will always produce a low hiring cost. Example: any ordering in which the best candidate is the first one interviewed. Then only the best candidate is hired.
- Some may be in between.

Instead of always interviewing the candidates in the order presented, what if you first randomly permuted this order?

- The randomization is now in the algorithm, not in the input distribution.
- Given a particular input, we can no longer say what its hiring cost will be. Each run of the algorithm can result in a different hiring cost.
- In other words, in each run of the algorithm, the execution depends on the random choices made.
- No particular input always elicits worst-case behavior.
- Bad behavior occurs only if you get "unlucky" numbers from the random-number generator.

Pseudocode for randomized hiring problem

RANDOMIZED-HIRE-ASSISTANT(n)
  randomly permute the list of candidates
  HIRE-ASSISTANT(n)

Lemma

The expected hiring cost of RANDOMIZED-HIRE-ASSISTANT is $O(c_h \ln n)$.

Proof After permuting the input array, we have a situation identical to the probabilistic analysis of deterministic HIRE-ASSISTANT. ■

Randomly permuting an array

Goal

Produce a uniform random permutation (each of the $n!$ permutations is equally likely to be produced).

Non-goal: Show that for each element $A[i]$, the probability that $A[i]$ moves to position j is $1/n$.

The following procedure permutes the array $A[1:n]$ in place (i.e., no auxiliary array is required).

RANDOMLY-PERMUTE(A, n)
  for $i = 1$ **to** n
    swap $A[i]$ with $A[\text{RANDOM}(i, n)]$

Idea

- In iteration i, choose $A[i]$ randomly from $A[i:n]$.
- Will never alter $A[i]$ after iteration i.

Time

$O(1)$ per iteration $\Rightarrow$ $O(n)$ total.

Correctness

Given a set of n elements, a ***k-permutation*** is a sequence containing k of the n elements. There are $n!/(n-k)!$ possible k-permutations. (On page 1180 in Appendix C.)

Lemma

RANDOMLY-PERMUTE computes a uniform random permutation.

Proof Use a loop invariant:

> **Loop invariant:** Just prior to the ith iteration of the **for** loop, for each possible $(i-1)$-permutation, subarray $A[1:i-1]$ contains this $(i-1)$-permutation with probability $(n-i+1)!/n!$.

Initialization: Just before first iteration, $i = 1$. Loop invariant says that for each possible 0-permutation, subarray $A[1:0]$ contains this 0-permutation with probability $n!/n! = 1$. $A[1:0]$ is an empty subarray, and a 0-permutation has no elements. So, $A[1:0]$ contains any 0-permutation with probability 1.

Maintenance: Assume that just prior to the ith iteration, each possible $(i-1)$-permutation appears in $A[1:i-1]$ with probability $(n-i+1)!/n!$. Will show that after the ith iteration, each possible i-permutation appears in $A[1:i]$ with probability $(n-i)!/n!$. Incrementing i for the next iteration then maintains the invariant.

Consider a particular i-permutation $\pi = \langle x_1, x_2, \ldots, x_i \rangle$. It consists of an $(i-1)$-permutation $\pi' = \langle x_1, x_2, \ldots, x_{i-1} \rangle$, followed by x_i.

Let E_1 be the event that the algorithm actually puts π' into $A[1:i-1]$. By the loop invariant, $\Pr\{E_1\} = (n-i+1)!/n!$.

Let E_2 be the event that the ith iteration puts x_i into $A[i]$.

We get the i-permutation π in $A[1:i]$ if and only if both E_1 and E_2 occur $\Rightarrow$ the probability that the algorithm produces π in $A[1:i]$ is $\Pr\{E_2 \cap E_1\}$.

Equation (C.16) $\Rightarrow \Pr\{E_2 \cap E_1\} = \Pr\{E_2 \mid E_1\} \Pr\{E_1\}$.

The algorithm chooses x_i randomly from the $n-i+1$ possibilities in $A[i:n]$ $\Rightarrow \Pr\{E_2 \mid E_1\} = 1/(n-i+1)$. Thus,

$$\begin{aligned}
\Pr\{E_2 \cap E_1\} &= \Pr\{E_2 \mid E_1\} \Pr\{E_1\} \\
&= \frac{1}{n-i+1} \cdot \frac{(n-i+1)!}{n!} \\
&= \frac{(n-i)!}{n!} \, .
\end{aligned}$$

Termination: The loop terminates, since it's a **for** loop iterating n times. At termination, $i = n + 1$, so we conclude that $A[1:n]$ is a given n-permutation with probability $(n-n)!/n! = 1/n!$. ■ (lemma)

4.3 QuickSelect intro, 4th edition

Lecture Notes for Chapter 9: Medians and Order Statistics

Chapter 9 overview

- ***i th order statistic*** is the ith smallest element of a set of n elements.
- The ***minimum*** is the first order statistic ($i = 1$).
- The ***maximum*** is the nth order statistic ($i = n$).
- A ***median*** is the "halfway point" of the set.
- When n is odd, the median is unique, at $i = (n + 1)/2$.
- When n is even, there are two medians:
  - The ***lower median***, at $i = n/2$, and
  - The ***upper median***, at $i = n/2 + 1$.
  - We mean lower median when we use the phrase "the median."

The ***selection problem***:

Input: A set A of n distinct numbers and a number i, with $1 \le i \le n$.

Output: The element $x \in A$ that is larger than exactly $i - 1$ other elements in A. In other words, the ith smallest element of A.

Easy to solve the selection problem in $O(n \lg n)$ time:

- Sort the numbers using an $O(n \lg n)$-time algorithm, such as heapsort or merge sort.
- Then return the ith element in the sorted array.

There are faster algorithms, however.

- First, we'll look at the problem of selecting the minimum and maximum of a set of elements.
- Then, we'll look at a simple general selection algorithm with a time bound of $O(n)$ in the average case.
- Finally, we'll look at a more complicated general selection algorithm with a time bound of $O(n)$ in the worst case.

Minimum and maximum

We can easily obtain an upper bound of $n-1$ comparisons for finding the minimum of a set of n elements.

- Examine each element in turn and keep track of the smallest one.
- This is the best we can do, because each element, except the minimum, must be compared to a smaller element at least once.

The following pseudocode finds the minimum element in array $A[1:n]$:

```
MINIMUM(A, n)
  min = A[1]
  for i = 2 to n
      if min > A[i]
          min = A[i]
  return min
```

The maximum can be found in exactly the same way by replacing the $>$ with $<$ in the above algorithm.

Simultaneous minimum and maximum

Some applications need both the minimum and maximum of a set of elements.

- For example, a graphics program may need to scale a set of (x, y) data to fit onto a rectangular display. To do so, the program must first find the minimum and maximum of each coordinate.

A simple algorithm to find the minimum and maximum is to find each one independently. There will be $n-1$ comparisons for the minimum and $n-1$ comparisons for the maximum, for a total of $2n-2$ comparisons. This will result in $\Theta(n)$ time.

In fact, at most $3\lfloor n/2 \rfloor$ comparisons suffice to find both the minimum and maximum:

- Maintain the minimum and maximum of elements seen so far.
- Don't compare each element to the minimum and maximum separately.
- Process elements in pairs.
- Compare the elements of a pair to each other.
- Then compare the larger element to the maximum so far, and compare the smaller element to the minimum so far.

This leads to only 3 comparisons for every 2 elements.

Setting up the initial values for the min and max depends on whether n is odd or even.

- If n is even, compare the first two elements and assign the larger to max and the smaller to min. Then process the rest of the elements in pairs.
- If n is odd, set both min and max to the first element. Then process the rest of the elements in pairs.

Analysis of the total number of comparisons

- If n is even, do 1 initial comparison and then $3(n-2)/2$ more comparisons.

$$\begin{aligned}\#\text{ of comparisons} &= \frac{3(n-2)}{2} + 1 \\ &= \frac{3n-6}{2} + 1 \\ &= \frac{3n}{2} - 3 + 1 \\ &= \frac{3n}{2} - 2\,.\end{aligned}$$

- If n is odd, do $3(n-1)/2 = 3\lfloor n/2 \rfloor$ comparisons.

In either case, the maximum number of comparisons is $\leq 3\lfloor n/2 \rfloor$.

Selection in expected linear time

Selection of the ith smallest element of the array A can be done in $\Theta(n)$ time.

The function RANDOMIZED-SELECT uses RANDOMIZED-PARTITION from the quicksort algorithm in Chapter 7. RANDOMIZED-SELECT differs from quicksort because it recurses on one side of the partition only.

```
RANDOMIZED-SELECT(A, p, r, i)
  if p == r
      return A[p]        // 1 ≤ i ≤ r − p + 1 when p == r means that i = 1
  q = RANDOMIZED-PARTITION(A, p, r)
  k = q − p + 1
  if i == k
      return A[q]        // the pivot value is the answer
  elseif i < k
      return RANDOMIZED-SELECT(A, p, q − 1, i)
  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

After the call to RANDOMIZED-PARTITION, the array is partitioned into two subarrays $A[p:q-1]$ and $A[q+1:r]$, along with a ***pivot*** element $A[q]$.

- The elements of subarray $A[p:q-1]$ are all $\leq A[q]$.
- The elements of subarray $A[q+1:r]$ are all $> A[q]$.
- The pivot element is the kth element of the subarray $A[p:r]$, where $k = q-p+1$.
- If the pivot element is the ith smallest element (i.e., $i = k$), return $A[q]$.
- Otherwise, recurse on the subarray containing the ith smallest element.
  - If $i < k$, this subarray is $A[p:q-1]$, and we want the ith smallest element.
  - If $i > k$, this subarray is $A[q+1:r]$ and, since there are k elements in $A[p:r]$ that precede $A[q+1:r]$, we want the $(i-k)$th smallest element of this subarray.

Analysis

Worst-case running time

$\Theta(n^2)$, because we could be extremely unlucky and always recurse on a subarray that is only one element smaller than the previous subarray.

Expected running time

RANDOMIZED-SELECT works well on average. Because it is randomized, no particular input brings out the worst-case behavior consistently.

Analysis assumes that the recursion goes as deep as possible: until only one element remains.

Intuition: Suppose that each pivot is in the second or third quartiles if the elements were sorted—in the "middle half." Then at least $1/4$ of the remaining elements are ignored in all future recursive calls $\Rightarrow$ at most $3/4$ of the elements are still ***in play***: somewhere within $A[p:r]$. RANDOMIZE-PARTITION takes $\Theta(n)$ time to partition n elements $\Rightarrow$ recurrence would be $T(n) = T(3n/4) + \Theta(n) = \Theta(n)$ by case 3 of the master method.

What if the pivot is not always in the middle half? Probability that it is in the middle half is $1/2$. View selecting a pivot in the middle half as a Bernoulli trial with probability of success $1/2$. Then the number of trials before a success is a geometric distribution with expected value 2. So that half the time, $1/4$ of the elements go out of play, and the other half of the time, as few as one element (the pivot) goes out of play. But that just doubles the running time, so still expect $\Theta(n)$.

Rigorous analysis:

- Define $A^{(j)}$ as the set of elements still in play (within $A[p:r]$) after j recursive calls (i.e., after jth partitioning). $A^{(0)}$ is all the elements in A.
- $|A^{(j)}|$ is a random variable that depends on A and order statistic i, but not on the order of elements in A.
- Each partitioning removes at least one element (the pivot) $\Rightarrow$ sizes of $A^{(j)}$ strictly decrease.
- jth partitioning takes set $A^{(j-1)}$ and produces $A^{(j)}$.
- Assume a 0th "dummy" partitioning that produces $A^{(0)}$.
- jth partitioning is ***helpful*** if $|A^{(j)}| \le (3/4)|A^{(j-1)}|$. Not all partitionings are necessarily helpful. Think of a helpful partitioning as a successful Bernoulli trial.

Lemma

A partitioning is helpful with probability $\ge 1/2$.

Proof

- Whether or not a partitioning is helpful depends on the randomly chosen pivot.
- Define "middle half" of an n-element subarray as all but the smallest $\lceil n/4 \rceil - 1$ and greatest $\lceil n/4 \rceil - 1$ elements. That is, all but the first and last $\lceil n/4 \rceil - 1$ if the subarray were sorted.

- Will show that if the pivot is in the middle half, then that pivot leads to a helpful partitioning and that the probability that the pivot is in the middle half is $\geq 1/2$.
- No matter where the pivot lies, either all elements $>$ pivot or all elements $<$ pivot, and the pivot itself, are not in play after partitioning $\Rightarrow$ if the pivot is in the middle half, at least the smallest $\lceil n/4 \rceil - 1$ or greatest $\lceil n/4 \rceil - 1$ elements, plus the pivot, will not be in play after partitioning $\Rightarrow \geq \lceil n/4 \rceil$ elements not in play.
- Then, at most $n - \lceil n/4 \rceil = \lfloor 3n/4 \rfloor < 3n/4$ elements in play $\Rightarrow$ partitioning is helpful. ($n - \lceil n/4 \rceil = \lfloor 3n/4 \rfloor$ is from Exercise 3.3-2.)
- To find a lower bound on the probability that a randomly chosen pivot is in the middle half, find an upper bound on the probability that it is not:

$$\begin{aligned}\frac{2(\lceil n/4 \rceil - 1)}{n} &\leq \frac{2((n/4+1)-1)}{n} \quad \text{(inequailty (3.2))} \\ &= \frac{n/2}{n} \\ &= 1/2\,.\end{aligned}$$

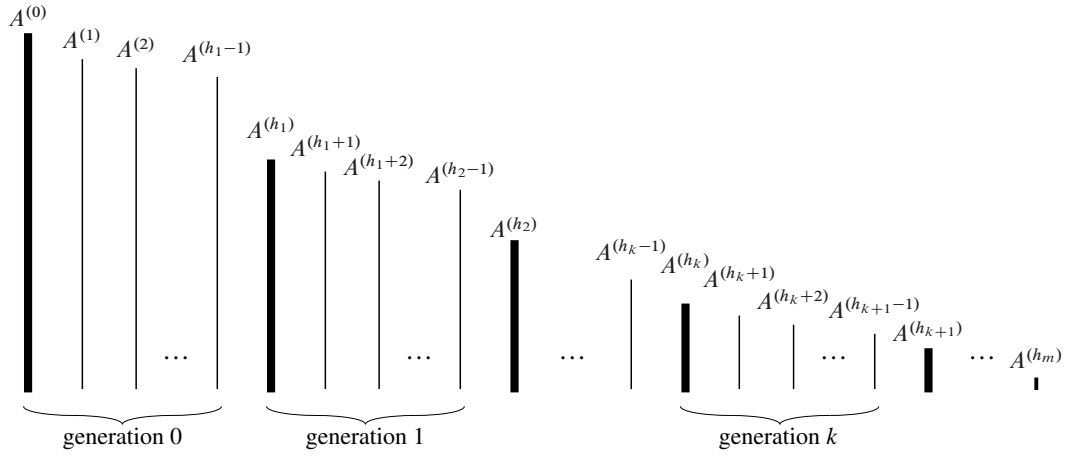
- Since the pivot has probability $\geq 1/2$ of falling into the middle half, a partitioning is helpful with probability $\geq 1/2$. ■ (lemma)

Theorem

The expected running time of RANDOMIZED-SELECT is $\Theta(n)$.

Proof

- Let the sequence of helpful partitionings be $\langle h_0, h_1, \ldots, h_m \rangle$. Consider the 0th partitioning as helpful $\Rightarrow h_0 = 0$. Can bound m, since after at most $\lceil \log_{4/3} n \rceil$ helpful partitionings, only one element remains in play.
- Define $n_k = |A^{(h_k)}|$ and $n_0 = |A^{(0)}|$, the original problem size. $n_k = |A^{(h_k)}| \leq (3/4)|A^{(h_k-1)}| = (3/4)\,n_{k-1}$ for $k = 1, 2, \ldots, m$.
- Iterating gives $n_k \leq (3/4)^k n_0$.
- Break up sets into m "generations." The sets in generation k are $A^{(h_k)}$, $A^{(h_k+1)}$, $\ldots$, $A^{(h_{k+1}-1)}$, where $A^{(h_k)}$ is the result of a helpful partitioning and $A^{(h_{k+1}-1)}$ is the last set before the next helpful partitioning.



[Height of each line indicates the set of the set (number of elements in play). Heavy lines are sets $A^{(h_k)}$, resulting from helpful partitionings and are first within their generation. Other lines are not first within their generation. A generation may contain just one set.]

- If $A^{(j)}$ is in the kth generation, then $|A^{(j)}| \le |A^{(h_k)}| = n_k \le (3/4)^k n_0$.
- Define random variable $X_k = h_{k+1} - h_k$ as the number of sets in the kth generation $\Rightarrow$ kth generation includes sets $A^{(h_k)}, A^{(h_k+1)}, \ldots, A^{(h_k+X_k-1)}$.
- By previous lemma, a partitioning is helpful with probability $\ge 1/2$. The probability is even higher, since a partitioning is helpful even if the pivot doesn't fall into middle half, but the ith smallest element lies in the smaller side. Just use the $1/2$ lower bound $\Rightarrow \mathrm{E}[X_k] \le 2$ for $k = 0, 1, \ldots, m-1$ (by equation (C.36), expectation of a geometric distribution).
- The total running time is dominated by the comparisons during partitioning. The jth partitioning takes $A^{(j-1)}$ and compares the pivot with all the other $|A^{(j-1)}| - 1$ elements $\Rightarrow$ jth partitioning makes $< |A^{(j-1)}|$ comparisons.
- The total number of comparisons is less than

$$\begin{aligned}
\sum_{k=0}^{m-1} \sum_{j=h_k}^{h_k+X_k-1} |A^{(j)}| &\le \sum_{k=0}^{m-1} \sum_{j=h_k}^{h_k+X_k-1} |A^{(h_k)}| \\
&= \sum_{k=0}^{m-1} X_k \, |A^{(h_k)}| \\
&\le \sum_{k=0}^{m-1} X_k \left(\frac{3}{4}\right)^k n_0 \, .
\end{aligned}$$

- Since $\mathrm{E}[X_k] \le 2$, the expected total number of comparisons is less than

$$\begin{aligned}
\mathrm{E}\left[\sum_{k=0}^{m-1} X_k \left(\frac{3}{4}\right)^k n_0\right] &= \sum_{k=0}^{m-1} \mathrm{E}\left[X_k \left(\frac{3}{4}\right)^k n_0\right] \quad \text{(linearity of expectation)} \\
&= n_0 \sum_{k=0}^{m-1} \left(\frac{3}{4}\right)^k \mathrm{E}[X_k] \\
&\le 2n_0 \sum_{k=0}^{m-1} \left(\frac{3}{4}\right)^k \\
&< 2n_0 \sum_{k=0}^{\infty} \left(\frac{3}{4}\right)^k \\
&= 8n_0 \quad \text{(infinite geometric series)} \, .
\end{aligned}$$

- n_0 is the size of the original array A $\Rightarrow$ an $O(n)$ upper bound on the expected running time. For the lower bound, the first call of RANDOMIZED-PARTITION examines all n elements $\Rightarrow$ $\Theta(n)$. ■ (theorem)

Therefore, we can determine any order statistic in linear time on average, assuming that all elements are distinct.

4.4 3rd edition proof

Analysis

Worst-case running time

$\Theta(n^2)$, because we could be extremely unlucky and always recurse on a subarray that is only 1 element smaller than the previous subarray.

Expected running time

RANDOMIZED-SELECT works well on average. Because it is randomized, no particular input brings out the worst-case behavior consistently.

The running time of RANDOMIZED-SELECT is a random variable that we denote by $T(n)$. We obtain an upper bound on $\mathrm{E}[T(n)]$ as follows:

- RANDOMIZED-PARTITION is equally likely to return any element of A as the pivot.
- For each k such that $1 \le k \le n$, the subarray $A[p..q]$ has k elements (all $\le$ pivot) with probability $1/n$. *[Note that we're now considering a subarray that includes the pivot, along with elements less than the pivot.]*
- For $k = 1, 2, \ldots, n$, define indicator random variable

  $X_k = \mathrm{I}\{\text{subarray } A[p..q] \text{ has exactly } k \text{ elements}\}$.
- Since $\Pr\{\text{subarray } A[p..q] \text{ has exactly } k \text{ elements}\} = 1/n$, Lemma 5.1 says that $\mathrm{E}[X_k] = 1/n$.
- When we call RANDOMIZED-SELECT, we don't know if it will terminate immediately with the correct answer, recurse on $A[p..q-1]$, or recurse on $A[q+1..r]$. It depends on whether the ith smallest element is less than, equal to, or greater than the pivot element $A[q]$.
- To obtain an upper bound, we assume that $T(n)$ is monotonically increasing and that the ith smallest element is always in the larger subarray.
- For a given call of RANDOMIZED-SELECT, $X_k = 1$ for exactly one value of k, and $X_k = 0$ for all other k.
- When $X_k = 1$, the two subarrays have sizes $k-1$ and $n-k$.
- For a subproblem of size n, RANDOMIZED-PARTITION takes $O(n)$ time. *[Actually, it takes $\Theta(n)$ time, but $O(n)$ suffices, since we're obtaining only an upper bound on the expected running time.]*
- Therefore, we have the recurrence

$$\begin{aligned} T(n) &\le \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n)) \\ &= \sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n) \; . \end{aligned}$$

- Taking expected values gives

$$\begin{aligned} &\mathrm{E}[T(n)] \\ &\le \mathrm{E}\left[\sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)\right] \end{aligned}$$

$$
\begin{aligned}
&= \sum_{k=1}^{n} \mathrm{E}\left[X_k \cdot T(\max(k-1, n-k))\right] + O(n) \qquad \text{(linearity of expectation)} \\
&= \sum_{k=1}^{n} \mathrm{E}\left[X_k\right] \cdot \mathrm{E}\left[T(\max(k-1, n-k))\right] + O(n) \quad \text{(equation (C.24))} \\
&= \sum_{k=1}^{n} \frac{1}{n} \cdot \mathrm{E}\left[T(\max(k-1, n-k))\right] + O(n) \; .
\end{aligned}
$$

- We rely on X_k and $T(\max(k-1, n-k))$ being independent random variables in order to apply equation (C.24).
- Looking at the expression $\max(k-1, n-k)$, we have

$$
\max(k-1, n-k) = \begin{cases} k-1 & \text{if } k > \lceil n/2 \rceil \; , \\ n-k & \text{if } k \le \lceil n/2 \rceil \; . \end{cases}
$$

  - If n is even, each term from $T(\lceil n/2 \rceil)$ up to $T(n-1)$ appears exactly twice in the summation.
  - If n is odd, these terms appear twice and $T(\lfloor n/2 \rfloor)$ appears once.

- Either way,

$$
\mathrm{E}\left[T(n)\right] \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} \mathrm{E}\left[T(k)\right] + O(n) \; .
$$

- Solve this recurrence by substitution:
  - Guess that $T(n) \le cn$ for some constant c that satisfies the initial conditions of the recurrence.
  - Assume that $T(n) = O(1)$ for $n <$ some constant. We'll pick this constant later.
  - Also pick a constant a such that the function described by the $O(n)$ term is bounded from above by an for all $n > 0$.
  - Using this guess and constants c and a, we have

$$
\begin{aligned}
\mathrm{E}\left[T(n)\right] &\le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an \\
&= \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) + an \\
&= \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\,\lfloor n/2 \rfloor}{2} \right) + an \\
&\le \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2} \right) + an \\
&= \frac{2c}{n} \left(\frac{n^2-n}{2} - \frac{n^2/4 - 3n/2 + 2}{2} \right) + an \\
&= \frac{c}{n} \left(\frac{3n^2}{4} + \frac{n}{2} - 2 \right) + an
\end{aligned}
$$

$$\begin{aligned} &= c\left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n}\right) + an \\ &\le \frac{3cn}{4} + \frac{c}{2} + an \\ &= cn - \left(\frac{cn}{4} - \frac{c}{2} - an\right) . \end{aligned}$$

- To complete this proof, we choose c such that

$$\begin{aligned} cn/4 - c/2 - an &\ge 0 \\ cn/4 - an &\ge c/2 \\ n(c/4 - a) &\ge c/2 \\ n &\ge \frac{c/2}{c/4 - a} \\ n &\ge \frac{2c}{c - 4a} . \end{aligned}$$

- Thus, as long as we assume that $T(n) = O(1)$ for $n < 2c/(c-4a)$, we have $\mathrm{E}[T(n)] = O(n)$.

Therefore, we can determine any order statistic in linear time on average.

Selection in worst-case linear time

We can find the ith smallest element in $O(n)$ time *in the worst case*. We'll describe a procedure SELECT that does so.

SELECT recursively partitions the input array.

- ***Idea:*** Guarantee a good split when the array is partitioned.
- Will use the deterministic procedure PARTITION, but with a small modification. Instead of assuming that the last element of the subarray is the pivot, the modified PARTITION procedure is told which element to use as the pivot.

SELECT works on an array of $n > 1$ elements. It executes the following steps:

1. Divide the n elements into groups of 5. Get $\lceil n/5 \rceil$ groups: $\lfloor n/5 \rfloor$ groups with exactly 5 elements and, if 5 does not divide n, one group with the remaining n mod 5 elements.
2. Find the median of each of the $\lceil n/5 \rceil$ groups:
   - Run insertion sort on each group. Takes $O(1)$ time per group since each group has ≤ 5 elements.
   - Then just pick the median from each group, in $O(1)$ time.
3. Find the median x of the $\lceil n/5 \rceil$ medians by a recursive call to SELECT. (If $\lceil n/5 \rceil$ is even, then follow our convention and find the lower median.)
4. Using the modified version of PARTITION that takes the pivot element as input, partition the input array around x. Let x be the kth element of the array after partitioning, so that there are $k-1$ elements on the low side of the partition and $n-k$ elements on the high side.

5 Chapter 20: Basic graph algorithms

Reading: Chapter 20

Lecture Notes for Chapter 20: Elementary Graph Algorithms

Graph representation

Given graph $G = (V, E)$. In pseudocode, represent vertex set by $G.V$ and edge set by $G.E$.

- G may be either directed or undirected.
- Two common ways to represent graphs for algorithms:
  1. Adjacency lists.
  2. Adjacency matrix.

When expressing the running time of an algorithm, it's often in terms of both $|V|$ and $|E|$. In asymptotic notation—and *only* in asymptotic notation—we'll drop the cardinality. Example: $O(V + E)$ really means $O(|V| + |E|)$.

[The introduction to Part VI talks more about this.]

Adjacency lists

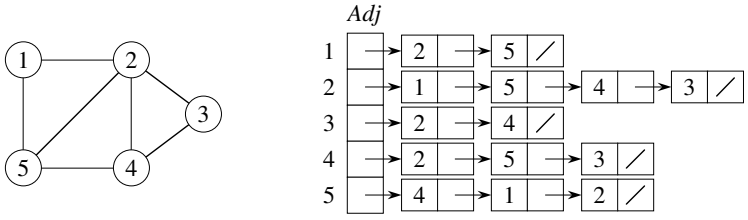
Array Adj of $|V|$ lists, one per vertex.

Vertex u's list has all vertices v such that $(u, v) \in E$. (Works for both directed and undirected graphs.)

In pseudocode, denote the array as attribute $G.Adj$, so will see notation such as $G.Adj[u]$.

Example

For an undirected graph:



If edges have *weights*, can put the weights in the lists.

Weight: $w : E \to \mathbb{R}$

We'll use weights later on for spanning trees and shortest paths.

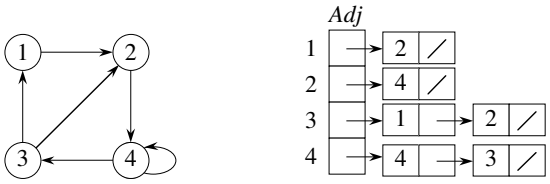
Space: $\Theta(V + E)$.

Time: to list all vertices adjacent to u: $\Theta(\text{degree}(u))$.

Time: to determine whether $(u, v) \in E$: $O(\text{degree}(u))$.

Example

For a directed graph:



Same asymptotic space and time.

Adjacency matrix

$|V| \times |V|$ matrix $A = (a_{ij})$

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \ , \\ 0 & \text{otherwise} \ . \end{cases}$$

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

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

Space: $\Theta(V^2)$.

Time: to list all vertices adjacent to u: $\Theta(V)$.

Time: to determine whether $(u, v) \in E$: $\Theta(1)$.

Can store weights instead of bits for weighted graph.

We'll use both representations in these lecture notes.

Representing graph attributes

Graph algorithms usually need to maintain attributes for vertices and/or edges. Use the usual dot-notation: denote attribute d of vertex v by $v.d$.

Use the dot-notation for edges, too: denote attribute f of edge (u, v) by $(u, v).f$.

Implementing graph attributes

No one best way to implement. Depends on the programming language, the algorithm, and how the rest of the program interacts with the graph.

If representing the graph with adjacency lists, can represent vertex attributes in additional arrays that parallel the Adj array, e.g., $d[1 : |V|]$, so that if vertices adjacent to u are in $Adj[u]$, store $u.d$ in array entry $d[u]$.

But can represent attributes in other ways. Example: represent vertex attributes as instance variables within a subclass of a `Vertex` class.

Breadth-first search

Input: Graph $G = (V, E)$, either directed or undirected, and ***source vertex*** $s \in V$.

Output:

- $v.d =$ distance (smallest # of edges) from s to v, for all $v \in V$.
- $v.\pi$ is v's ***predecessor*** on a shortest path (smallest # of edges) from s.
  (u, v) is last edge on shortest path $s \rightsquigarrow v$.
  Predecessor subgraph contains edges (u, v) such that $v.\pi = u$.
  The predecessor subgraph forms a tree, called the ***breadth-first tree***.

Later, we'll see a generalization of breadth-first search, with edge weights. For now, we'll keep it simple.

[Omitting colors of vertices. Used in book to reason about the algorithm.]

Intuition

Breadth-first search expands the frontier between discovered and undiscovered vertices uniformly across the breath of the frontier.

Discovers vertices in waves, starting from s.

- First visits all vertices 1 edge from s.
- From there, visits all vertices 2 edges from s.
- Etc.

Use FIFO queue Q to maintain wavefront.

- $v \in Q$ if and only if wave has visited v but has not come out of v yet.
- Q contains vertices at a distance k, and possibly some vertices at a distance $k + 1$. Therefore, at any time Q contains portions of two consecutive waves.

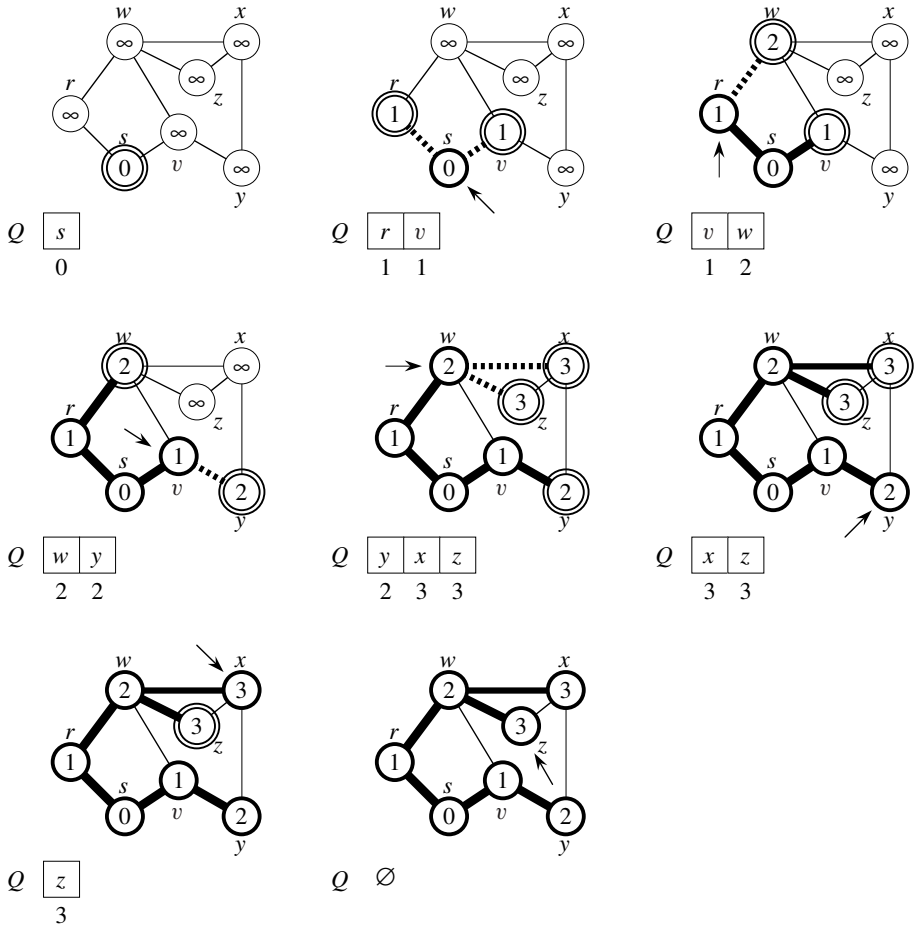
```
BFS(V, E, s)
  for each vertex u ∈ V − {s}
      u.d = ∞
      u.π = NIL
  s.d = 0
  Q = ∅
  ENQUEUE(Q, s)
  while Q ≠ ∅
      u = DEQUEUE(Q)
      for each vertex v in G.Adj[u]   // search the neighbors of u
          if v.d == ∞                 // is v being discovered now?
              v.d = u.d + 1
              v.π = u
              ENQUEUE(Q, v)           // v is now on the frontier
      // u is now behind the frontier.
```

[In the book, the test for whether v is being newly discovered uses the colors. Checking whether $v.d$ is finite or infinite works just as well, since once v is discovered it gets a finite d value. Can also check for whether $v.\pi$ equals NIL*.]*

Example

BFS on an undirected graph: *[There is a more detailed, colorized example in book. Go through this example, showing how vertices are discovered and Q is updated]* .

- Arrows point to the vertex being visited.
- Edges drawn with heavy lines are in the predecessor subgraph.
- Dashed lines go to newly discovered vertices. They are drawn with heavy lines because they are also now in the predecessor subgraph.
- Double-outline vertices have been discovered and are in Q, waiting to be visited.
- Heavy-outline vertices have been discovered, dequeued from Q, and visited.



Can show that Q consists of vertices with d values.

$k \quad k \quad k \quad \ldots \quad k \quad k+1 \quad k+1 \quad \ldots \quad k+1$

- Only 1 or 2 values.
- If 2, differ by 1 and all smallest are first.

Since each vertex gets a finite d value at most once, values assigned to vertices are monotonically increasing over time.

[Actual proof of correctness is a bit trickier. See book.]

BFS may not reach all vertices.

Time $= O(V + E)$.

- $O(V)$ because every vertex enqueued at most once.
- $O(E)$ because every vertex dequeued at most once and edge (u, v) is examined only when u is dequeued. Therefore, every edge examined at most once if directed, at most twice if undirected.

To print the vertices on a shortest path from s to v:

```
PRINT-PATH(G, s, v)
  if v == s
      print s
  elseif v.π == NIL
      print "no path from" s "to" v "exists"
  else PRINT-PATH(G, s, v.π)
      print v
```

Depth-first search

Input: $G = (V, E)$, directed or undirected. No source vertex given.
Output:

- 2 ***timestamps*** on each vertex:
  - $v.d =$ ***discovery time***
  - $v.f =$ ***finish time***

  These will be useful for other algorithms later on.
- $v.\pi$ is v's predecessor in the ***depth-first forest*** of ≥ 1 ***depth-first trees***. If $u = v.\pi$, then (u, v) is a ***tree edge***.

Methodically explores *every* edge.

- Start over from different vertices as necessary.

As soon as a vertex is discovered, explore from it.

- Unlike BFS, which puts a vertex on a queue so that it's explored from later.

As DFS progresses, every vertex has a ***color***:

- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)

Discovery and finish times:

- Unique integers from 1 to $2|V|$.
- For all v, $v.d < v.f$.

In other words, $1 \leq v.d < v.f \leq 2|V|$.

Pseudocode

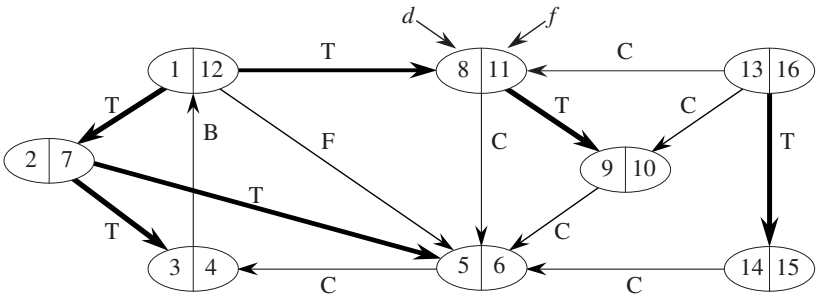
Uses a global timestamp *time*.

```
DFS(G)
  for each vertex u ∈ G.V
      u.color = WHITE
      u.π = NIL
  time = 0
  for each vertex u ∈ G.V
      if u.color == WHITE
          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
  time = time + 1                  // white vertex u has just been discovered
  u.d = time
  u.color = GRAY
  for each vertex v in G.Adj[u]    // explore each edge (u, v)
      if v.color == WHITE
          v.π = u
          DFS-VISIT(G, v)
  time = time + 1
  u.f = time
  u.color = BLACK                  // blacken u; it is finished
```

Example

[Go through this example of DFS on a directed graph, adding in the d and f values as they're computed. Show colors as they change. Don't put in the edge types yet, except that the tree edges are drawn with heavy lines.]



Time $= \Theta(V + E)$.

- Similar to BFS analysis.
- Θ, not just O, since guaranteed to examine every vertex and edge.

Each depth-first tree is made of edges (u, v) such that u is gray and v is white when (u, v) is explored.

Theorem (Parenthesis theorem)
[Proof omitted.]

For all u, v, exactly one of the following holds:

1. $u.d < u.f < v.d < v.f$ or $v.d < v.f < u.d < u.f$ (i.e., the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint) and neither of u and v is a descendant of the other.
2. $u.d < v.d < v.f < u.f$ and v is a descendant of u. (v is discovered after and finished before u.)
3. $v.d < u.d < u.f < v.f$ and u is a descendant of v. (u is discovered after and finished before v.)

So $u.d < v.d < u.f < v.f$ (v is both discovered and finished after u) *cannot* happen.

Like parentheses:

- OK: ( ) [ ] ([ ]) [( )]
- Not OK: ([ )] [( ])

Corollary

v is a proper descendant of u if and only if $u.d < v.d < v.f < u.f$.

Theorem (White-path theorem)

[Proof omitted.]

v is a descendant of u if and only if at time $u.d$, there is a path $u \rightsquigarrow v$ consisting of only white vertices. (Except for u, which was *just* colored gray.)

Classification of edges

- ***Tree edge:*** in the depth-first forest. Found by exploring (u, v).
- ***Back edge:*** (u, v), where u is a descendant of v.
- ***Forward edge:*** (u, v), where v is a descendant of u, but not a tree edge.
- ***Cross edge:*** any other edge. Can go between vertices in same depth-first tree or in different depth-first trees.

[Now label the example from above with edge types.]

In an undirected graph, there may be some ambiguity since (u, v) and (v, u) are the same edge. Classify by the first type above that matches.

Theorem

[Proof omitted.]

A DFS of an *undirected* graph yields only tree and back edges. No forward or cross edges.

Topological sort

Directed acyclic graph (dag)

A directed graph with no cycles.

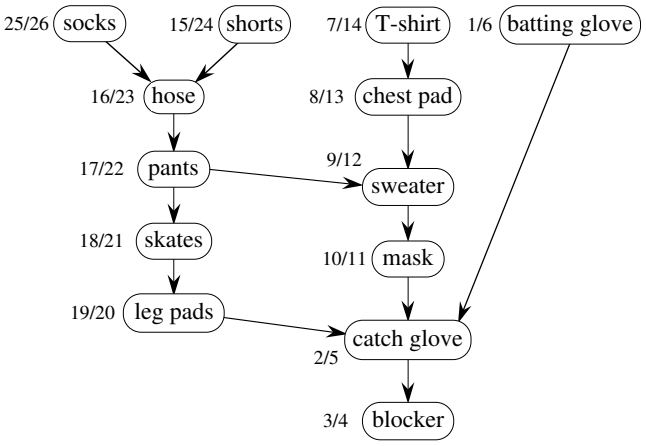
Good for modeling processes and structures that have a ***partial order:***

- $a > b$ and $b > c \Rightarrow a > c$.
- But may have a and b such that neither $a > b$ nor $b > c$.

Can always make a ***total order*** (either $a > b$ or $b > a$ for all $a \neq b$) from a partial order. In fact, that's what a topological sort will do.

Example

Dag of dependencies for putting on goalie equipment for ice hockey: *[Leave on board, but show without discovery and finish times. Will put them in later.]*

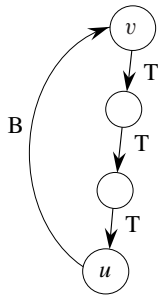


Lemma

A directed graph G is acyclic if and only if a DFS of G yields no back edges.

Proof $\Rightarrow$: Show that back edge $\Rightarrow$ cycle.

Suppose there is a back edge (u, v). Then v is ancestor of u in depth-first forest.



Therefore, there is a path $v \rightsquigarrow u$, so $v \rightsquigarrow u \rightarrow v$ is a cycle.

$\Leftarrow$: Show that cycle $\Rightarrow$ back edge.

Suppose G contains cycle c. Let v be the first vertex discovered in c, and let (u, v) be the preceding edge in c. At time $v.d$, vertices of c form a white path $v \rightsquigarrow u$ (since v is the first vertex discovered in c). By white-path theorem, u is descendant of v in depth-first forest. Therefore, (u, v) is a back edge. ■ (lemma)

Topological sort of a dag: a linear ordering of vertices such that if $(u, v) \in E$, then u appears somewhere before v. (Not like sorting numbers.)

TOPOLOGICAL-SORT(G)
  call DFS(G) to compute finish times $v.f$ for all $v \in G.V$
  output vertices in order of *decreasing* finish times

Don't need to sort by finish times.

- Can just output vertices as they're finished and understand that we want the *reverse* of this list.
- Or put them onto the *front* of a linked list as they're finished. When done, the list contains vertices in topologically sorted order.

Time

$\Theta(V + E)$.

Do example. *[Now write discovery and finish times in goalie equipment example.]*

Order:

26 socks
24 shorts
23 hose
22 pants
21 skates
20 leg pads
14 t-shirt
13 chest pad
12 sweater
11 mask
6 batting glove
5 catch glove
4 blocker

Correctness

Just need to show if $(u, v) \in E$, then $v.f < u.f$.
When edge (u, v) is explored, what are the colors of u and v?

- u is gray.
- Is v gray, too?
  - *No*, because then v would be ancestor of u.
    $\Rightarrow (u, v)$ is a back edge.
    $\Rightarrow$ contradiction of previous lemma (dag has no back edges).
- Is v white?
  - Then becomes descendant of u.
    By parenthesis theorem, $u.d < v.d < \underline{v.f < u.f}$.
- Is v black?
  - Then v is already finished.
    Since exploring (u, v), u is not yet finished.
    Therefore, $v.f < u.f$. ■

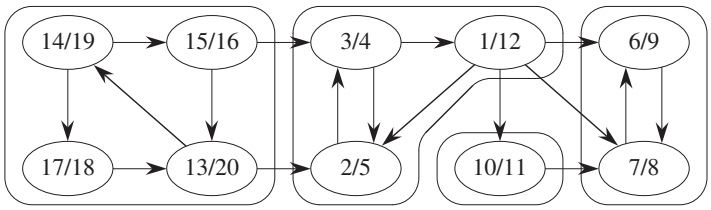
Strongly connected components

Given directed graph $G = (V, E)$.

A ***strongly connected component*** (***SCC***) of G is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

Example

[Don't show discovery/finish times yet.]



Algorithm uses $G^{\mathrm{T}} =$ ***transpose*** of G.

- $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$, $E^{\mathrm{T}} = \{(u, v) : (v, u) \in E\}$.
- G^{T} is G with all edges reversed.

Can create G^{T} in $\Theta(V + E)$ time if using adjacency lists.

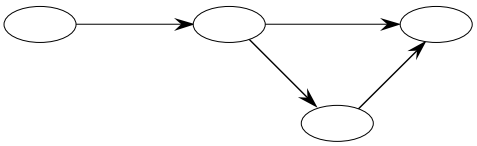
Observation

G and G^{T} have the *same* SCC's. (u and v are reachable from each other in G if and only if reachable from each other in G^{T}.)

Component graph

- $G^{\mathrm{SCC}} = (V^{\mathrm{SCC}}, E^{\mathrm{SCC}})$.
- V^{SCC} has one vertex for each SCC in G.
- E^{SCC} has an edge if there's an edge between the corresponding SCC's in G.

For our example:



Lemma

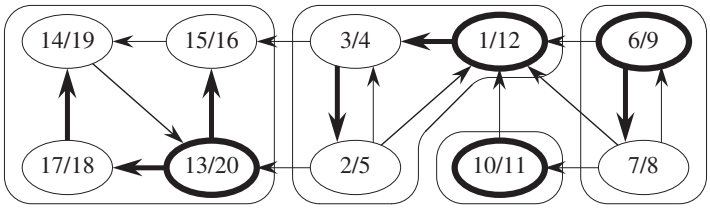
G^{SCC} is a dag. More formally, let C and C' be distinct SCC's in G, let $u, v \in C$, $u', v' \in C'$, and suppose there is a path $u \rightsquigarrow u'$ in G. Then there cannot also be a path $v' \rightsquigarrow v$ in G.

Proof Suppose there is a path $v' \rightsquigarrow v$ in G. Then there are paths $u \rightsquigarrow u' \rightsquigarrow v'$ and $v' \rightsquigarrow v \rightsquigarrow u$ in G. Therefore, u and v' are reachable from each other, so they are not in separate SCC's. ■ (lemma)

SCC(G)
  call DFS(G) to compute finish times $u.f$ for each vertex u
  create G^{T}
  call DFS(G^{T}), but in the main loop, consider vertices in order of decreasing $u.f$
      (as computed in first DFS)
  output the vertices in each tree of the depth-first forest formed in second DFS
      as a separate SCC

Example:

1. Do DFS in G. *[Now show discovery and finish times in G.]*
2. G^{T}.
3. DFS in G^{T}. *[Discovery and finish times are from first DFS in G. Roots in second DFS in G^{T} are drawn with heavy outlines, tree edges in second DFS are drawn with heavy lines.]*



Time: $\Theta(V + E)$.

How can this possibly work?

Idea

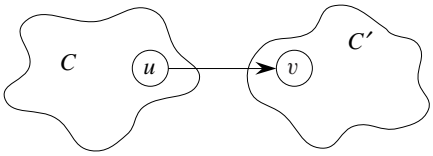
By considering vertices in second DFS in decreasing order of finish times from first DFS, visiting vertices of the component graph in topological sort order.

To prove that it works, first deal with 2 notational issues:

- Will be discussing $u.d$ and $u.f$. These always refer to *first* DFS.
- Extend notation for d and f to sets of vertices $U \subseteq V$:
  - $d(U) = \min\{u.d : u \in U\}$ (earliest discovery time in U)
  - $f(U) = \max\{u.f : u \in U\}$ (latest finish time in U)

Lemma

Let C and C' be distinct SCC's in $G = (V, E)$. Suppose that there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$.



Then $f(C) > f(C')$.

Proof Two cases, depending on which SCC had the first discovered vertex during the first DFS.

- If $d(C) < d(C')$, let x be the first vertex discovered in C. At time $x.d$, all vertices in C and C' are white. Thus, there exist paths of white vertices from x to all vertices in C and C'.

  By the white-path theorem, all vertices in C and C' are descendants of x in depth-first tree.

  By the parenthesis theorem, $x.f = f(C) > f(C')$.
- If $d(C) > d(C')$, let y be the first vertex discovered in C'. At time $y.d$, all vertices in C' are white and there is a white path from y to each vertex in C' $\Rightarrow$ all vertices in C' become descendants of y. Again, $y.f = f(C')$.

  At time $y.d$, all vertices in C are white.

  By earlier lemma, since there is an edge (u, v), we cannot have a path from C' to C.

  So no vertex in C is reachable from y.

  Therefore, at time $y.f$, all vertices in C are still white.

  Therefore, for all $w \in C$, $w.f > y.f$, which implies that $f(C) > f(C')$.

■ (lemma)

Corollary
Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E^{\mathrm{T}}$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.

Proof $(u, v) \in E^{\mathrm{T}} \Rightarrow (v, u) \in E$. Since SCC's of G and G^{T} are the same, $f(C') > f(C)$. ■ (corollary)

Corollary
Let C and C' be distinct SCC's in $G = (V, E)$, and suppose that $f(C) > f(C')$. Then there cannot be an edge from C to C' in G^{T}.

Proof It's the contrapositive of the previous corollary. ■

Now we have the intuition to understand why the SCC procedure works.

The second DFS, on G^{T}, starts with an SCC C such that $f(C)$ is maximum. The second DFS starts from some $x \in C$, and it visits all vertices in C. The corollary says that since $f(C) > f(C')$ for all $C' \neq C$, there are no edges from C to C' in G^{T}.

Therefore, the second DFS visits *only* vertices in C.

Which means that the depth-first tree rooted at x contains *exactly* the vertices of C.

The next root chosen in the second DFS is in SCC C' such that $f(C')$ is maximum over all SCC's other than C. DFS visits all vertices in C', but the only edges out of C' go to C, *which we've already visited.*

Therefore, the only tree edges will be to vertices in C'.

The process continues.

Each root chosen for the second DFS can reach only

- vertices in its SCC—get tree edges to these,
- vertices in SCC's *already visited* in second DFS—get *no* tree edges to these.

Visiting vertices of $(G^{\mathrm{T}})^{\mathrm{SCC}}$ in reverse of topologically sorted order.

[The book has a formal proof.]

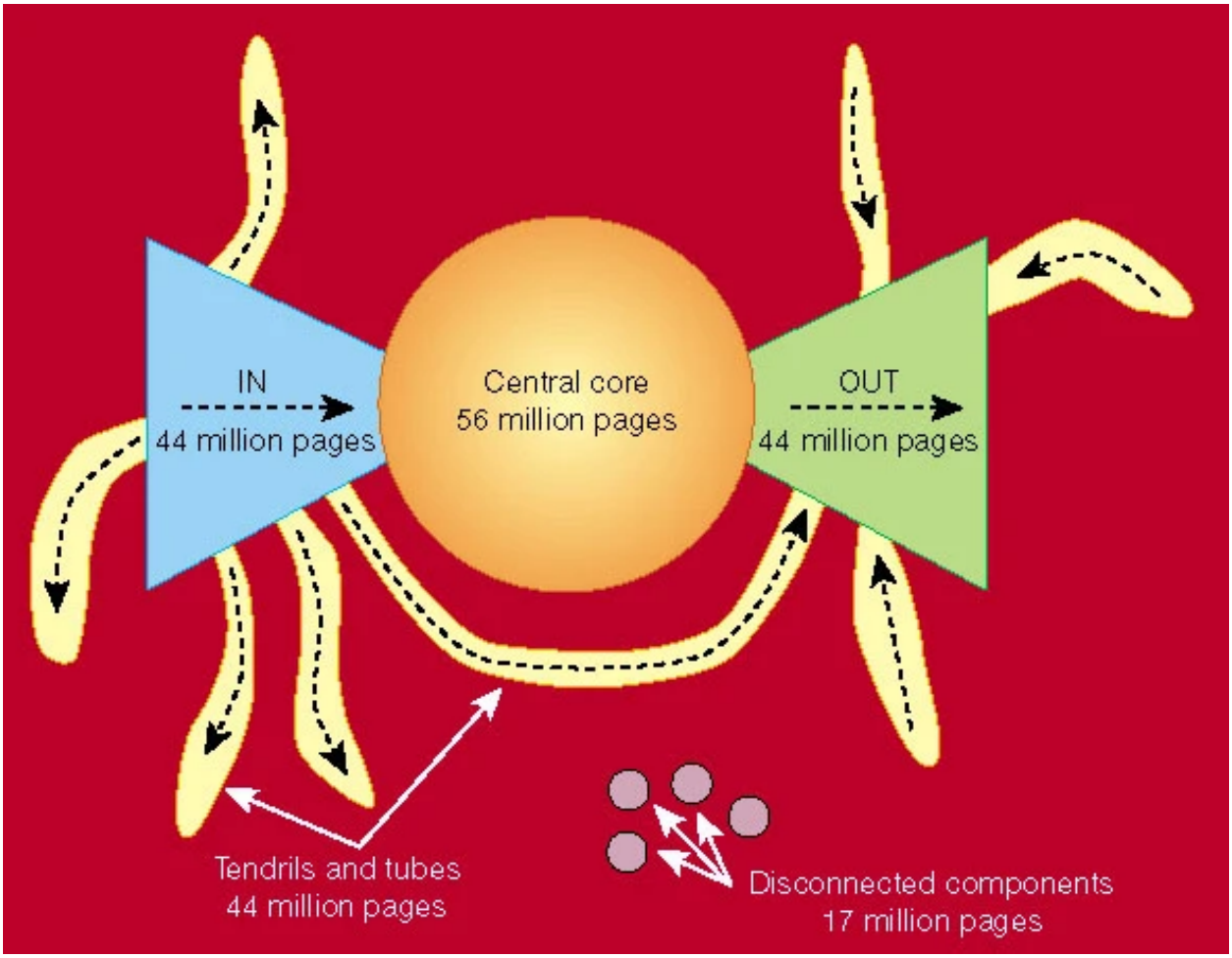


Figure 3: The Web circa 2000

6 Chapter 21: MSTs

Reading: Chapter 21

Lecture Notes for Chapter 21: Minimum Spanning Trees

Chapter 21 overview

Problem

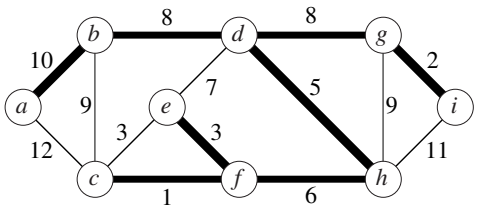
- A town has a set of houses and a set of roads.
- A road connects 2 and only 2 houses.
- A road connecting houses u and v has a repair cost $w(u, v)$.
- ***Goal:*** Repair enough (and no more) roads such that
  1. everyone stays connected: can reach every house from all other houses, and
  2. total repair cost is minimum.

Model as a graph:

- Undirected graph $G = (V, E)$.
- ***Weight*** $w(u, v)$ on each edge $(u, v) \in E$.
- Find $T \subseteq E$ such that
  1. T connects all vertices (T is a ***spanning tree***), and
  2. $w(T) = \sum_{(u,v)\in T} w(u, v)$ is minimized.

A spanning tree whose weight is minimum over all spanning trees is called a ***minimum spanning tree***, or ***MST***.

Example of such a graph *[Differs from Figure 21.1 in the textbook. Edges in the MST are drawn with heavy lines.]* :



In this example, there is more than one MST. Replace edge (e, f) in the MST by (c, e). Get a different spanning tree with the same weight.

Growing a minimum spanning tree

Some properties of an MST:

- It has $|V| - 1$ edges.
- It has no cycles.
- It might not be unique.

Building up the solution

- Build a set A of edges.
- Initially, A has no edges.
- As edges are added to A, maintain a loop invariant:

  Loop invariant: A is a subset of some MST.

- Add only edges that maintain the invariant. If A is a subset of some MST, an edge (u, v) is ***safe*** for A if and only if $A \cup \{(u, v)\}$ is also a subset of some MST. So add only safe edges.

Generic MST algorithm

GENERIC-MST(G, w)
  $A = \emptyset$
  while A does not form a spanning tree
    find an edge (u, v) that is safe for A
    $A = A \cup \{(u, v)\}$
  return A

Use the loop invariant to show that this generic algorithm works.

Initialization: The empty set trivially satisfies the loop invariant.

Maintenance: Since only safe edges are added, A remains a subset of some MST.

Termination: The loop must terminate by the time it considers all edges. All edges added to A are in an MST, so upon termination. A is a spanning tree that is also an MST.

Finding a safe edge

How to find safe edges?

Let's look at the example. Edge (c, f) has the lowest weight of any edge in the graph. Is it safe for $A = \emptyset$?

Intuitively: Let $S \subset V$ be any proper subset of vertices that includes c but not f (so that f is in $V - S$). In any MST, there has to be one edge (at least) that connects S with $V - S$. Why not choose the edge with minimum weight? (Which would be (c, f) in this case.)

Some definitions: Let $S \subset V$ and $A \subseteq E$.

- A ***cut*** $(S, V - S)$ is a partition of vertices into disjoint sets V and $S - V$.
- Edge $(u, v) \in E$ ***crosses*** cut $(S, V - S)$ if one endpoint is in S and the other is in $V - S$.
- A cut ***respects*** A if and only if no edge in A crosses the cut.
- An edge is a ***light edge*** crossing a cut if and only if its weight is minimum over all edges crossing the cut. For a given cut, there can be > 1 light edge crossing it.

Theorem

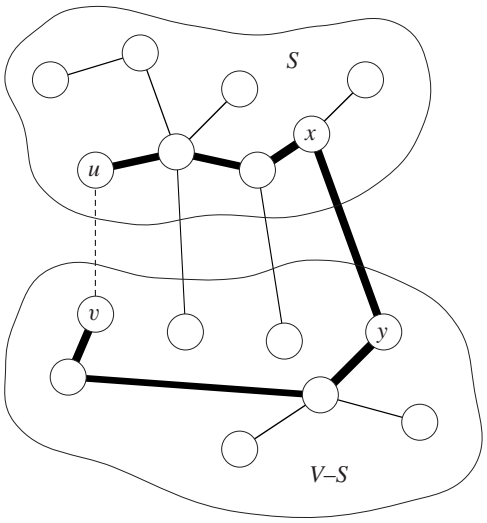
Let A be a subset of some MST, $(S, V - S)$ be a cut that respects A, and (u, v) be a light edge crossing $(S, V - S)$. Then (u, v) is safe for A.

Proof Let T be an MST that includes A.

If T contains (u, v), done.

So now assume that T does not contain (u, v). Construct a different MST T' that includes $A \cup \{(u, v)\}$.

Recall: a tree has unique path between each pair of vertices. Since T is an MST, it contains a unique path p between u and v. Path p must cross the cut$(S, V - S)$ at least once. Let (x, y) be an edge of p that crosses the cut. From how we chose (u, v), must have $w(u, v) \leq w(x, y)$.



[Except for the dashed edge (u, v), all edges shown are in T. A is some subset of the edges of T, but A cannot contain any edges that cross the cut $(S, V - S)$, since this cut respects A. Edges with heavy lines are the path p.]

Since the cut respects A, edge (x, y) is not in A.

To form T' from T:

- Remove (x, y). Breaks T into two components.
- Add (u, v). Reconnects.

So $T' = T - \{(x, y)\} \cup \{(u, v)\}$.

T' is a spanning tree.

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\leq w(T) , \end{aligned}$$

since $w(u, v) \leq w(x, y)$. Since T' is a spanning tree, $w(T') \leq w(T)$, and T is an MST, then T' must be an MST.

Need to show that (u, v) is safe for A:

- $A \subseteq T$ and $(x, y) \notin A \Rightarrow A \subseteq T'$.
- $A \cup \{(u, v)\} \subseteq T'$.
- Since T' is an MST, (u, v) is safe for A. ■ (theorem)

So, in GENERIC-MST:

- A is a forest containing connected components. Initially, each component is a single vertex.
- Any safe edge merges two of these components into one. Each component is a tree.
- Since an MST has exactly $|V| - 1$ edges, the **for** loop iterates $|V| - 1$ times. Equivalently, after adding $|V|-1$ safe edges, we're down to just one component.

Corollary

If $C = (V_C, E_C)$ is a connected component in the forest $G_A = (V, A)$ and (u, v) is a light edge connecting C to some other component in G_A (i.e., (u, v) is a light edge crossing the cut $(V_C, V - V_C)$), then (u, v) is safe for A.

Proof Set $S = V_C$ in the theorem. ■ (corollary)

This idea naturally leads to the algorithm known as Kruskal's algorithm to solve the minimum-spanning-tree problem.

Kruskal's algorithm

$G = (V, E)$ is a connected, undirected, weighted graph. $w : E \to \mathbb{R}$.

- Starts with each vertex being its own component.
- Repeatedly merges two components into one by choosing the light edge that connects them (i.e., the light edge crossing the cut between them).
- Scans the set of edges in monotonically increasing order by weight.
- Uses a disjoint-set data structure to determine whether an edge connects vertices in different components.

```
MST-KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  create a single list of the edges in G.E
  sort the list of edges into nondecreasing order by weight w
  for each edge (u, v) taken from the sorted list in order
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

Run through the above example to see how Kruskal's algorithm works on it:

(c, f) : safe
(g, i) : safe
(e, f) : safe
(c, e) : reject
(d, h) : safe
(f, h) : safe
(e, d) : reject
(b, d) : safe
(d, g) : safe
(b, c) : reject
(g, h) : reject
(a, b) : safe

At this point, there is only one component, so that all other edges will be rejected. *[Could add a test to the main loop of* KRUSKAL *to stop once* $|V| - 1$ *edges have been added to* A*.]*

Get the heavy edges shown in the figure.

Suppose (c, e) had been examined *before* (e, f). Then would have found (c, e) safe and would have rejected (e, f).

Analysis

Initialize A:	$O(1)$
First **for** loop:	$\|V\|$ MAKE-SETs
Sort E:	$O(E \lg E)$
Second **for** loop:	$O(E)$ FIND-SETs and UNIONs

- Assuming the implementation of disjoint-set data structure, already seen in Chapter 19, that uses union by rank and path compression:

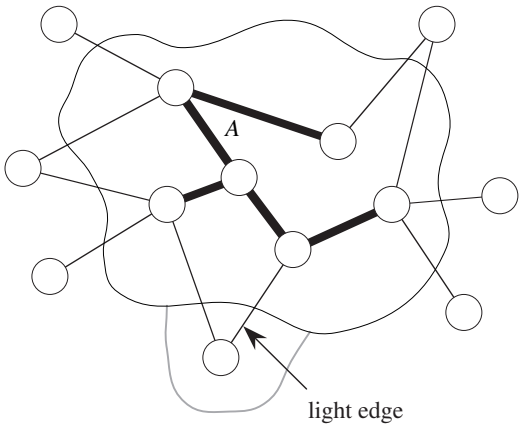
  $$O((V + E)\,\alpha(V)) + O(E \lg E)\ .$$

- Since G is connected, $|E| \geq |V| - 1 \Rightarrow O(E\,\alpha(V)) + O(E \lg E)$.
- $\alpha(|V|) = O(\lg V) = O(\lg E)$.
- Therefore, total time is $O(E \lg E)$.

- $|E| \le |V|^2 \Rightarrow \lg|E| = O(2\lg V) = O(\lg V)$.
- Therefore, $O(E \lg V)$ time. (If edges are already sorted, $O(E\,\alpha(V))$, which is almost linear.)

Prim's algorithm

- Builds one tree, so A is always a tree.
- Starts from an arbitrary "root" r.
- At each step, find a light edge connecting A to an isolated vertex. Such an edge must be safe for A. Add this edge to A.



[Edges of A are drawn with heavy lines.]

How to find the light edge quickly?

Use a priority queue Q:

- Each object is a vertex *not* in A.
- $v.key$ is the minimum weight of any edge connecting v to a vertex in A. $v.key = \infty$ if no such edge.
- $v.\pi$ is v's parent in A.
- Maintain A implicitly as $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$.
- At completion, Q is empty and the minimum spanning tree is $A = \{(v, v.\pi) : v \in V - \{r\}\}$.

```
MST-PRIM(G, w, r)
  for each vertex u ∈ G.V
      u.key = ∞
      u.π = NIL
  r.key = 0
  Q = ∅
  for each vertex u ∈ G.V
      INSERT(Q, u)
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)          // add u to the tree
      for each vertex v in G.Adj[u]    // update keys of u's non-tree neighbors
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              v.key = w(u, v)
              DECREASE-KEY(Q, v, w(u, v))
```

Loop invariant: Prior to each iteration of the **while** loop,

1. $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$.
2. The vertices already placed into the minimum spanning tree are those in $V - Q$.
3. For all vertices $v \in Q$, if $v.\pi \neq$ NIL, then $v.key < \infty$ and $v.key$ is the weight of a light edge $(v, v.\pi)$ connecting v to some vertex already placed into the minimum spanning tree.

Do example from the graph on page 21-1. *[Let a student pick the root.]*

Analysis

Depends on how the priority queue is implemented:

- Suppose Q is a binary heap.

Initialize Q and first **for** loop:	$O(V \lg V)$
Decrease key of r:	$O(\lg V)$
while loop:	$\lvert V\rvert$ EXTRACT-MIN calls $\Rightarrow O(V \lg V)$
	$\leq \lvert E\rvert$ DECREASE-KEY calls $\Rightarrow O(E \lg V)$
Total:	$O(E \lg V)$

- Suppose DECREASE-KEY could take $O(1)$ *amortized* time.

  Then $\leq \lvert E\rvert$ DECREASE-KEY calls take $O(E)$ time altogether $\Rightarrow$ total time becomes $O(V \lg V + E)$.

  In fact, there is a way to perform DECREASE-KEY in $O(1)$ amortized time: Fibonacci heaps, mentioned in the introduction to Part V.

7 Chapter 22: Single-Source Shortest Paths

Reading: Chapter 22 intro, 22.1, 22.3, 22.5 (skip DAGs and difference constraints)

Lecture Notes for Chapter 22: Single-Source Shortest Paths

Shortest paths

How to find the shortest route between two points on a map.

Input:

- Directed graph $G = (V, E)$
- Weight function $w : E \to \mathbb{R}$

Weight of path $p = \langle v_0, v_1, \ldots, v_k \rangle$

$$= \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

= sum of edge weights on path p .

Shortest-path weight u to v:

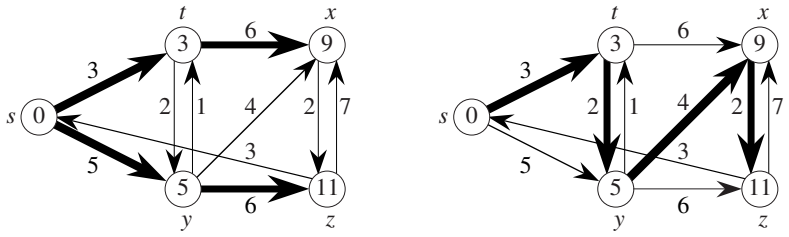
$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there exists a path } u \rightsquigarrow v \text{ ,} \\ \infty & \text{otherwise .} \end{cases}$$

Shortest path u to v is any path p such that $w(p) = \delta(u, v)$.

Example

shortest paths from s

[δ values appear inside vertices. Heavy edges show shortest paths.]



This example shows that a shortest path might not be unique.

It also shows that when we look at shortest paths from one vertex to all other vertices, the shortest paths are organized as a tree.

Can think of weights as representing any measure that

- accumulates linearly along a path, and
- we want to minimize.

Examples: time, cost, penalties, loss.

Generalization of breadth-first search to weighted graphs.

Variants

- ***Single-source:*** Find shortest paths from a given ***source*** vertex $s \in V$ to every vertex $v \in V$.
- ***Single-destination:*** Find shortest paths to a given destination vertex.
- ***Single-pair:*** Find shortest path from u to v. No way known that's better in worst case than solving single-source.
- ***All-pairs:*** Find shortest path from u to v for all $u, v \in V$. We'll see algorithms for all-pairs in the next chapter.

Negative-weight edges

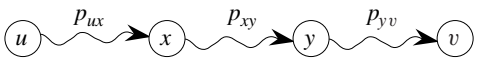
OK, as long as no negative-weight cycles are reachable from the source.

- If we have a negative-weight cycle, we can just keep going around it, and get $w(s, v) = -\infty$ for all v on the cycle.
- But OK if the negative-weight cycle is not reachable from the source.
- Some algorithms work only if there are no negative-weight edges in the graph. We'll be clear when they're allowed and not allowed.

Optimal substructure

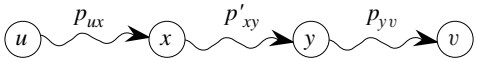
Lemma
Any subpath of a shortest path is a shortest path.

Proof Cut-and-paste.



Suppose this path p is a shortest path from u to v.
Then $\delta(u, v) = w(p) = w(p_{ux}) + w(p_{xy}) + w(p_{yv})$.

Now suppose there exists a shorter path $x \overset{p'_{xy}}{\rightsquigarrow} y$.
Then $w(p'_{xy}) < w(p_{xy})$.
Construct p':



Then

$$\begin{aligned} w(p') &= w(p_{ux}) + w(p'_{xy}) + w(p_{yv}) \\ &< w(p_{ux}) + w(p_{xy}) + w(p_{yv}) \\ &= w(p) \,. \end{aligned}$$

Contradicts the assumption that p is a shortest path. ■ (lemma)

Cycles

Shortest paths can't contain cycles:

- Already ruled out negative-weight cycles.
- Positive-weight $\Rightarrow$ we can get a shorter path by omitting the cycle.
- 0-weight: no reason to use them $\Rightarrow$ assume that our solutions won't use them.

Output of single-source shortest-path algorithm

For each vertex $v \in V$:

- $v.d = \delta(s, v)$.
  - Initially, $v.d = \infty$.
  - Reduces as algorithms progress. But always maintain $v.d \geq \delta(s, v)$.
  - Call $v.d$ a ***shortest-path estimate***.
- $v.\pi$ = predecessor of v on a shortest path from s.
  - If no predecessor, $v.\pi = \text{NIL}$.
  - π induces a tree—***shortest-path tree***.
  - We won't prove properties of π in lecture—see text.

Initialization

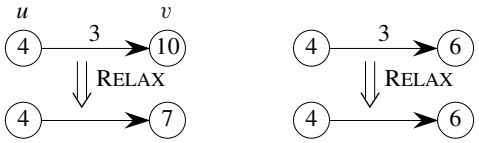
All the shortest-paths algorithms start with INITIALIZE-SINGLE-SOURCE.

```
INITIALIZE-SINGLE-SOURCE(G, s)
  for each vertex v ∈ G.V
      v.d = ∞
      v.π = NIL
  s.d = 0
```

Relaxing an edge (u, v)

Can the shortest-path estimate for v be improved by going through u and taking (u, v)?

```
RELAX(u, v, w)
  if v.d > u.d + w(u, v)
      v.d = u.d + w(u, v)
      v.π = u
```



For all the single-source shortest-paths algorithms we'll look at,

- start by calling INITIALIZE-SINGLE-SOURCE,
- then relax edges.

The algorithms differ in the order and how many times they relax each edge.

Shortest-paths properties

[The textbook states these properties in the chapter introduction and proves them in a later section. You might elect to just state these properties at first and prove them later.]

Based on calling INITIALIZE-SINGLE-SOURCE once and then calling RELAX zero or more times.

Triangle inequality: For all $(u, v) \in E$, we have $\delta(s, v) \le \delta(s, u) + w(u, v)$.

Proof Weight of shortest path $s \rightsquigarrow v$ is $\le$ weight of any path $s \rightsquigarrow v$. Path $s \rightsquigarrow u \rightarrow v$ is a path $s \rightsquigarrow v$, and if we use a shortest path $s \rightsquigarrow u$, its weight is $\delta(s, u) + w(u, v)$. ■

Upper-bound property: Always have $v.d \ge \delta(s, v)$ for all v. Once $v.d$ gets down to $\delta(s, v)$, it never changes.

Proof Initially true.

Suppose there exists a vertex such that $v.d < \delta(s, v)$.

Without loss of generality, v is first vertex for which this happens.

Let u be the vertex that causes $v.d$ to change.

Then $v.d = u.d + w(u, v)$.

So,

$$\begin{aligned} v.d &< \delta(s, v) \\ &\le \delta(s, u) + w(u, v) \quad \text{(triangle inequality)} \\ &\le u.d + w(u, v) \quad (v \text{ is first violation}) \\ \Rightarrow v.d &< u.d + w(u, v)\ . \end{aligned}$$

Contradicts $v.d = u.d + w(u, v)$.

Once $v.d$ reaches $\delta(s, v)$, it never goes lower. It never goes up, since relaxations only lower shortest-path estimates. ■

No-path property: If $\delta(s, v) = \infty$, then $v.d = \infty$ always.

Proof $v.d \geq \delta(s,v) = \infty \Rightarrow v.d = \infty$. ■

Convergence property: If $s \rightsquigarrow u \rightarrow v$ is a shortest path, $u.d = \delta(s,u)$, and edge (u,v) is relaxed, then $v.d = \delta(s,v)$ afterward.

Proof After relaxation:

$$\begin{aligned} v.d &\leq u.d + w(u,v) && \text{(RELAX code)} \\ &= \delta(s,u) + w(u,v) \\ &= \delta(s,v) && \text{(lemma—optimal substructure)} \end{aligned}$$

Since $v.d \geq \delta(s,v)$, must have $v.d = \delta(s,v)$. ■

Path-relaxation property: Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from $s = v_0$ to v_k. If the edges of p are relaxed, *in the order*, $(v_0,v_1), (v_1,v_2), \ldots, (v_{k-1},v_k)$, even intermixed with other relaxations, then $v_k.d = \delta(s,v_k)$.

Proof Induction to show that $v_i.d = \delta(s,v_i)$ after (v_{i-1},v_i) is relaxed.

Basis: $i = 0$. Initially, $v_0.d = 0 = \delta(s,v_0) = \delta(s,s)$.

Inductive step: Assume $v_{i-1}.d = \delta(s,v_{i-1})$. Relax (v_{i-1},v_i). By convergence property, $v_i.d = \delta(s,v_i)$ afterward and $v_i.d$ never changes. ■

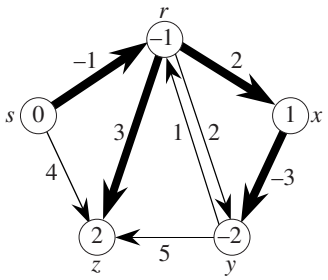
The Bellman-Ford algorithm

- Allows negative-weight edges.
- Computes $v.d$ and $v.\pi$ for all $v \in V$.
- Returns TRUE if no negative-weight cycles reachable from s, FALSE otherwise.

```
BELLMAN-FORD(G, w, s)
  INITIALIZE-SINGLE-SOURCE(G, s)
  for i = 1 to |G.V| − 1
      for each edge (u, v) ∈ G.E
          RELAX(u, v, w)
  for each edge (u, v) ∈ G.E
      if v.d > u.d + w(u, v)
          return FALSE
  return TRUE
```

Time: $O(V^2 + VE)$. The first **for** loop makes $|V| - 1$ passes over the edges, and each pass takes $\Theta(V + E)$ time. We use O rather than Θ because sometimes $< |V| - 1$ passes are enough (Exercise 22.1-3).

Example



Values you get on each pass and how quickly it converges depends on order of relaxation.

But guaranteed to converge after $|V| - 1$ passes, assuming no negative-weight cycles.

Proof Use path-relaxation property.

Let v be reachable from s, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from s to v, where $v_0 = s$ and $v_k = v$. Since p is acyclic, it has $\leq |V| - 1$ edges, so that $k \leq |V| - 1$.

Each iteration of the **for** loop relaxes all edges:

- First iteration relaxes (v_0, v_1).
- Second iteration relaxes (v_1, v_2).
- kth iteration relaxes (v_{k-1}, v_k).

By the path-relaxation property, $v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$. ■

How about the TRUE/FALSE return value?

- Suppose there is no negative-weight cycle reachable from s.

  At termination, for all $(u, v) \in E$,

  $$\begin{aligned} v.d &= \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) \quad \text{(triangle inequality)} \\ &= u.d + w(u, v) \ . \end{aligned}$$

  So BELLMAN-FORD returns TRUE.

- Now suppose there exists negative-weight cycle $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$, reachable from s.

  Then $\displaystyle\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$.

  Suppose (for contradiction) that BELLMAN-FORD returns TRUE.

  Then $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$.

  Sum around c:

  $$\begin{aligned} \sum_{i=1}^{k} v_i.d &\leq \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i) \end{aligned}$$

Each vertex appears once in each summation $\sum_{i=1}^{k} v_i.d$ and $\sum_{i=1}^{k} v_{i-1}.d \Rightarrow$

$$0 \le \sum_{i=1}^{k} w(v_{i-1}, v_i) \ .$$

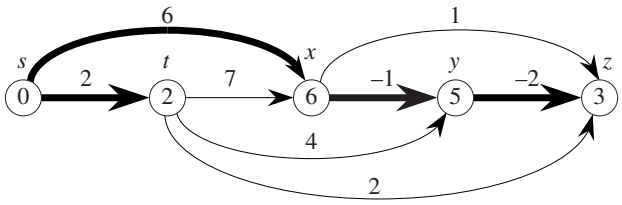
Contradicts c being a negative-weight cycle. ■

Single-source shortest paths in a directed acyclic graph

Since a dag, we're guaranteed no negative-weight cycles.

DAG-SHORTEST-PATHS(G, w, s)
  topologically sort the vertices of G
  INITIALIZE-SINGLE-SOURCE(G, s)
  for each vertex $u \in G.V$, taken in topologically sorted order
    for each vertex v in $G.Adj[u]$
      RELAX(u, v, w)

Example



Time

$\Theta(V + E)$.

Correctness

Because vertices are processed in topologically sorted order, edges of *any* path must be relaxed in order of appearance in the path.
$\Rightarrow$ Edges on any shortest path are relaxed in order.
$\Rightarrow$ By path-relaxation property, correct. ■

Dijkstra's algorithm

No negative-weight *edges*.

Essentially a weighted version of breadth-first search.

- Instead of a FIFO queue, uses a priority queue.
- Keys are shortest-path weights ($v.d$).
- Can think of waves, like BFS.

- A wave emanates from the source.
- The first time that a wave arrives at a vertex, a new wave emanates from that vertex.
- The time it takes for the wave to arrive at a neighboring vertex equals the weight of the edge. (In BFS, each wave takes unit time to arrive at each neighbor.)

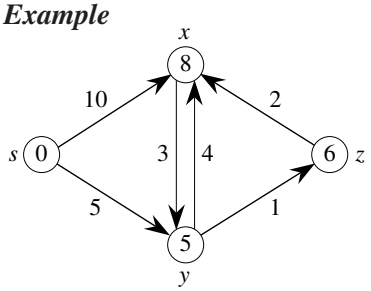
Have two sets of vertices:

- S = vertices whose final shortest-path weights are determined,
- Q = priority queue = $V - S$.

```
DIJKSTRA(G, w, s)
  INITIALIZE-SINGLE-SOURCE(G, s)
  S = ∅
  Q = ∅
  for each vertex u ∈ G.V
      INSERT(Q, u)
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      S = S ∪ {u}
      for each vertex v in G.Adj[u]
          RELAX(u, v, w)
          if the call of RELAX decreased v.d
              DECREASE-KEY(Q, v, v.d)
```

- Looks a lot like Prim's algorithm, but computing $v.d$, and using shortest-path weights as keys.
- Dijkstra's algorithm can be viewed as greedy, since it always chooses the "lightest" ("closest"?) vertex in $V - S$ to add to S.

Example



Order of adding to S: s, y, z, x.

Correctness

We will show that at the start of each iteration of the **while** loop, $v.d = \delta(s, v)$ for all $v \in S$. The algorithm terminates when $S = V$, so that $v.d = \delta(s, v)$ for all $v \in V$.

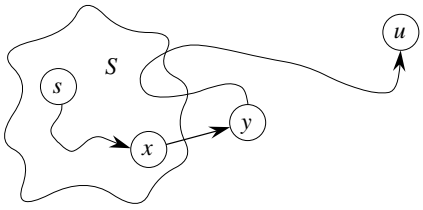
The proof is by induction on the number of iterations of the **while** loop, i.e., on $|S|$. The bases are for $|S| = 0$, so that $S = \emptyset$ and the claim is trivially true, and for $|S| = 1$, so that $S = \{s\}$ and $s.d = \delta(s, s) = 0$.

Inductive hypothesis: $v.d = \delta(s, v)$ for all $v \in S$.

Inductive step: The algorithm extracts vertex u from $V - S$. Because the algorithm adds u into S, we need to show that $u.d = \delta(s, u)$ at that time. If there is no path from s to u, then we are done, by the no-path property.

If there is a path from s to u:

- Let y be the first vertex on a shortest path from s to u that is *not* in S.
- Let $x \in S$ be the predecessor of y on that shortest path.
- Could have $y = u$ or $x = s$.



- y appears no later than u on the shortest path and all edge weights are nonnegative $\Rightarrow \delta(s, y) \le \delta(s, u)$.
- How we chose $u \Rightarrow u.d \le y.d$ at the time u is extracted from $V - S$.
- Upper-bound property $\Rightarrow \delta(s, u) \le u.d$.
- $x \in S \Rightarrow x.d = \delta(s, x)$. Edge (x, y) was relaxed when x was added into S. Convergence property $\Rightarrow$ set $y.d = \delta(s, y)$ at that time.
- Thus, we have $\delta(s, y) \le \delta(s, u) \le u.d \le y.d$ and $y.d = \delta(s, y) \Rightarrow \delta(s, y) = \delta(s, u) = u.d = y.d$.
- Hence, $u.d = \delta(s, u)$. Upper-bound property $\Rightarrow u.d$ doesn't change afterward. ■

Analysis

$|V|$ INSERT and EXTRACT-MIN operations.
$\le |E|$ DECREASE-KEY operations.

Like Prim's algorithm, depends on implementation of priority queue.

- If binary heap, each operation takes $O(\lg V)$ time $\Rightarrow O(E \lg V)$.
- If a Fibonacci heap:
  - Each EXTRACT-MIN takes $O(1)$ amortized time.
  - There are $\Theta(V)$ INSERT and EXTRACT-MIN operations, taking $O(\lg V)$ amortized time each.
  - Therefore, time is $O(V \lg V + E)$.

Difference constraints

Special case of linear programming.

Given a set of inequalities of the form $x_j - x_i \le b_k$.

8 Chapter 23: All-Pairs Shortest Paths

Reading: Chapter 23 intro, 23.1, 23.2 (focus on Floyd-Warshall algorithm)

Lecture Notes for Chapter 23: All-Pairs Shortest Paths

Chapter 23 overview

Given a directed graph $G = (V, E)$, weight function $w : E \to \mathbb{R}$, $|V| = n$. Assume that the vertices are numbered $1, 2, \ldots, n$.

Goal: create an $n \times n$ matrix $D = (d_{ij})$ of shortest-path distances, so that $d_{ij} = \delta(i, j)$ for all vertices i and j.

Could run BELLMAN-FORD once from each vertex:

- $O(V^2E)$—which is $O(V^4)$ if the graph is ***dense*** ($E = \Theta(V^2)$).

If no negative-weight edges, could run Dijkstra's algorithm once from each vertex:

- $O(VE \lg V)$ with binary heap—$O(V^3 \lg V)$ if dense,
- $O(V^2 \lg V + VE)$ with Fibonacci heap—$O(V^3)$ if dense.

We'll see how to do in $O(V^3)$ in all cases, with no fancy data structure.

Shortest paths and matrix multiplication

Assume that G is given as adjacency matrix of weights: $W = (w_{ij})$, with vertices numbered 1 to n.

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \ , \\ \text{weight of edge } (i, j) & \text{if } i \neq j, (i, j) \in E \ , \\ \infty & \text{if } i \neq j, (i, j) \notin E \ . \end{cases}$$

Won't worry about predecessors—see book.

Will use dynamic programming at first.

Optimal substructure

Recall: subpaths of shortest paths are shortest paths.

Recursive solution

Let $l_{ij}^{(r)}$ = weight of shortest path $i \rightsquigarrow j$ that contains $\le r$ edges.

- $r = 0$
  $\Rightarrow$ there is a shortest path $i \rightsquigarrow j$ with $\le r$ edges if and only if $i = j$
  $$\Rightarrow l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \ , \\ \infty & \text{if } i \ne j \ . \end{cases}$$
- $r \ge 1$
  $$\Rightarrow l_{ij}^{(r)} = \min\left\{l_{ij}^{(r-1)},\ \min\left\{l_{ik}^{(r-1)} + w_{kj} : 1 \le k \le n\right\}\right\}$$
  (k ranges over all possible predecessors of j)
  $$= \min\left\{l_{ik}^{(r-1)} + w_{kj} : 1 \le k \le n\right\} \qquad (\text{since } w_{jj} = 0 \text{ for all } j) \ .$$
- Observe that when $r = 1$, must have $l_{ij}^{(1)} = w_{ij}$.

  Conceptually, when the path is restricted to at most 1 edge, the weight of the shortest path $i \rightsquigarrow j$ must be w_{ij}.

  And the math works out, too:
  $$l_{ij}^{(1)} = \min\left\{l_{ik}^{(0)} + w_{kj} : 1 \le k \le n\right\}$$
  $$= l_{ii}^{(0)} + w_{ij} \qquad (l_{ii}^{(0)} \text{ is the only non-}\infty \text{ among } l_{ik}^{(0)})$$
  $$= w_{ij} \ .$$

All simple shortest paths contain $\le n - 1$ edges
$\Rightarrow \delta(i, j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \cdots$

Compute a solution bottom-up

Compute $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$.

Start with $L^{(1)} = W$, since $l_{ij}^{(1)} = w_{ij}$.

Go from $L^{(r-1)}$ to $L^{(r)}$:

EXTEND-SHORTEST-PATHS$(L^{(r-1)}, W, L^{(r)}, n)$
```
// Assume that the elements of L^(r) are initialized to ∞.
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            l_ij^(r) = min{l_ij^(r), l_ik^(r-1) + w_kj}
```

Compute each $L^{(r)}$:

SLOW-APSP$(W, L^{(0)}, n)$
```
let L = (l_ij) and M = (m_ij) be new n × n matrices
L = L^(0)
for r = 1 to n − 1
    M = ∞      // initialize M
    EXTEND-SHORTEST-PATHS(L, W, M, n)
    L = M
return L
```

Time

- EXTEND-SHORTEST-PATHS: $\Theta(n^3)$.
- SLOW-ALL-APSP: $\Theta(n^4)$.

Observation

EXTEND-SHORTEST-PATHS is like matrix multiplication. Make the following substitutions in the equation $l_{ij}^{(r)} = \min\{l_{ik}^{(r-1)} + w_{kj} : 1 \le k \le n\}$:

$l^{(r-1)} \rightarrow a$
$w \rightarrow b$
$l^{(r)} \rightarrow c$
$\min \rightarrow +$
$+ \rightarrow \cdot$

You get $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$, which is the equation for computing c_{ij} in matrix multiplication.

Making these changes to EXTEND-SHORTEST-PATHS and replacing ∞ (identity for min) with 0 (identity for $+$) gives a procedure for matrix multiplication:

```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
```

So, we can view EXTEND-SHORTEST-PATHS as just like matrix multiplication!

Why do we care?

Because our goal is to compute $L^{(n-1)}$ as fast as we can. Don't need to compute *all* the intermediate $L^{(1)}, L^{(2)}, L^{(3)}, \ldots, L^{(n-2)}$.

Suppose we had a matrix A and we wanted to compute A^{n-1} (like calling EXTEND-SHORTEST-PATHS $n-1$ times).

Could compute $A, A^2, A^4, A^8, \ldots$

If we knew $A^r = A^{n-1}$ for all $r \ge n-1$, could just finish with A^r, where r is the smallest power of 2 that is $\ge n-1$ ($r = 2^{\lceil \lg(n-1) \rceil}$).

```
FASTER-ALL-PAIRS-SHORTEST-PATHS(W, n)
  let L and M be new n × n matrices
  L = W
  r = 1
  while r < n − 1
      M = ∞      // initialize M
      EXTEND-SHORTEST-PATHS(L, L, M, n)      // compute M = L^2
      r = 2r
      L = M      // ready for the next iteration
  return L
```

OK to overshoot, since products don't change after $L^{(n-1)}$.

Time

$\Theta(n^3 \lg n)$.

Floyd-Warshall algorithm

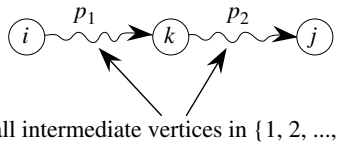
A different dynamic-programming approach.

For path $p = \langle v_1, v_2, \ldots, v_l \rangle$, an ***intermediate vertex*** is any vertex of p other than v_1 or v_l.

Let $d_{ij}^{(k)}$ = shortest-path weight of any path $i \rightsquigarrow j$ with all intermediate vertices in $\{1, 2, \ldots, k\}$.

Consider a shortest path $i \overset{p}{\rightsquigarrow} j$ with all intermediate vertices in $\{1, 2, \ldots, k\}$:

- If k is not an intermediate vertex, then all intermediate vertices of p are in $\{1, 2, \ldots, k-1\}$.
- If k is an intermediate vertex:



Recursive formulation

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0\,, \\ \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\} & \text{if } k \geq 1\,. \end{cases}$$

Have $d_{ij}^{(0)} = w_{ij}$ because can't have intermediate vertices $\Rightarrow \leq 1$ edge.

Want $D^{(n)} = \left(d_{ij}^{(n)}\right)$, since all vertices numbered $\leq n$.

Compute bottom-up

Compute in increasing order of k:

```
FLOYD-WARSHALL(W, n)
  D^(0) = W
  for k = 1 to n
      let D^(k) = (d_ij^(k)) be a new n × n matrix
      for i = 1 to n
          for j = 1 to n
              d_ij^(k) = min{d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1)}
  return D^(n)
```

Can drop superscripts. (See Exercise 23.2-4 in text.)

Time

$\Theta(n^3)$.

Computing predecessors

Can compute predecessor matrix Π while computing the D matrices. Let $\Pi^{(k)} = \left(\pi_{ij}^{(k)}\right)$ for $k = 0, 1, \ldots, n$.

Define $\pi_{ij}^{(k)}$ recursively. For $k = 0$, a shortest path from i to j has no intermediate vertices:

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty \,, \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty \,. \end{cases}$$

For $k \geq 1$:

- If shortest path from i to j has k as an intermediate vertex, then it's $i \rightsquigarrow k \rightsquigarrow j$ where $k \neq j$. Choose j's predecessor to be the predecessor of j on a shortest path from k to j with all intermediate vertices $< k$: $\pi_{ij}^{(k)} = \pi_{kj}^{(k-1)}$.
- Otherwise, shortest path from i to j does not have k as an intermediate vertex. Keep the same predecessor as shortest path from i to j with all intermediate vertices $< k$: $\pi_{ij}^{(k)} = \pi_{ij}^{(k-1)}$.

Transitive closure

Given $G = (V, E)$, directed.

Compute $G^* = (V, E^*)$.

- $E^* = \{(i, j) : \text{there is a path } i \rightsquigarrow j \text{ in } G\}$.

Could assign weight of 1 to each edge, then run FLOYD-WARSHALL.

- If $d_{ij} < n$, then there is a path $i \rightsquigarrow j$.
- Otherwise, $d_{ij} = \infty$ and there is no path.

Simpler way

Substitute other values and operators in FLOYD-WARSHALL.

- Use unweighted adjacency matrix
- min $\rightarrow \vee$ (OR)
- $+ \rightarrow \wedge$ (AND)
- $t_{ij}^{(k)} = \begin{cases} 1 & \text{if there is path } i \rightsquigarrow j \text{ with all intermediate vertices in } \{1, 2, \ldots, k\} \,, \\ 0 & \text{otherwise} \,. \end{cases}$
- $t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i, j) \notin E \,, \\ 1 & \text{if } i = j \text{ or } (i, j) \in E \,. \end{cases}$
- $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$.

```
TRANSITIVE-CLOSURE(G, n)
  let T^(0) = (t_ij^(0)) be a new n × n matrix
  for i = 1 to n
      for j = 1 to n
          if i == j or (i, j) ∈ G.E
              t_ij^(0) = 1
          else t_ij^(0) = 0
  for k = 1 to n
      let T^(k) = (t_ij^(k)) be a new n × n matrix
      for i = 1 to n
          for j = 1 to n
              t_ij^(k) = t_ij^(k−1) ∨ (t_ik^(k−1) ∧ t_kj^(k−1))
  return T^(n)
```

Time

$\Theta(n^3)$, but simpler operations than FLOYD-WARSHALL.

Johnson's algorithm

Idea

If the graph is sparse, it pays to run Dijkstra's algorithm once from each vertex.

If we use a Fibonacci heap for the priority queue, the running time is down to $O(V^2 \lg V + VE)$, which is better than FLOYD-WARSHALL's $\Theta(V^3)$ time if $E = o(V^2)$.

But Dijkstra's algorithm requires that all edge weights be nonnegative.

Donald Johnson figured out how to make an equivalent graph that *does* have all edge weights ≥ 0.

Reweighting

Compute a new weight function $\hat{w}$ such that

1. For all $u, v \in V$, p is a shortest path $u \rightsquigarrow v$ using w if and only if p is a shortest path $u \rightsquigarrow v$ using $\hat{w}$.
2. For all $(u, v) \in E$, $\hat{w}(u, v) \geq 0$.

Property (1) says that it suffices to find shortest paths with $\hat{w}$. Property (2) says we can do so by running Dijkstra's algorithm from each vertex.

How to come up with $\hat{w}$?

Lemma shows it's easy to get property (1):

9 PageRank

PageRank

Ryan Tibshirani
Data Mining: 36-462/36-662

January 22 2013

Optional reading: ESL 14.10

Information retrieval with the web

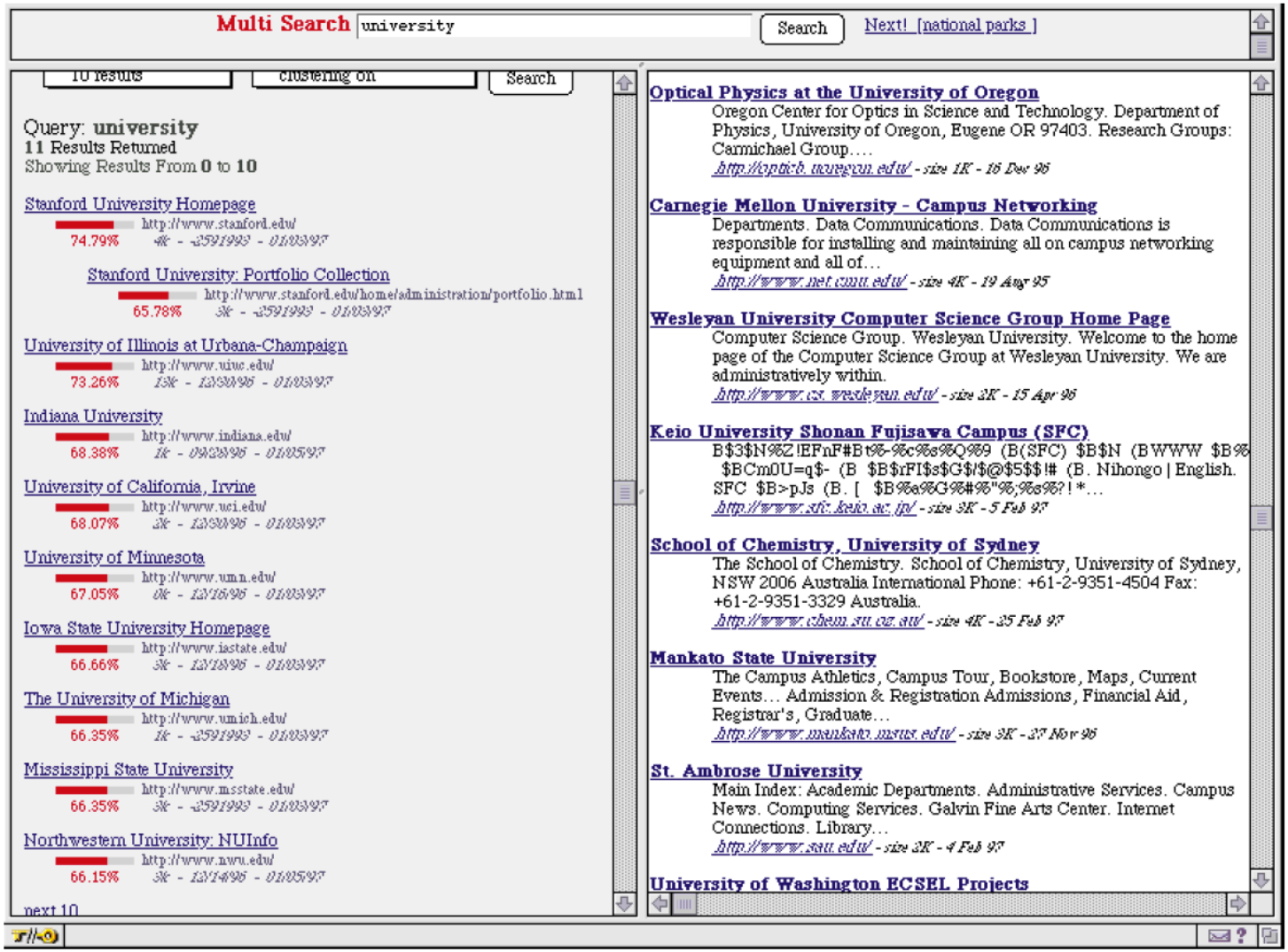
Last time: information retrieval, learned how to compute similarity scores (distances) of documents to a given query string

But what if documents are webpages, and our collection is the whole web (or a big chunk of it)? Now, two problems:

- Techniques from last lectures (normalization, IDF weighting) are computationally infeasible at this scale. There are about 30 billion webpages!
- Some webpages should be assigned more priority than others, for being more important

Fortunately, there is an underlying structure that we can exploit: links between webpages

Web search before Google



(From Page et al. (1999), "The PageRank Citation Ranking: Bringing Order to the Web")

PageRank algorithm

PageRank algorithm: famously invented by Larry Page and Sergei Brin, founders of Google. Assigns a *PageRank* (score, or a measure of importance) to each webpage

Given webpages numbered $1, \ldots n$. The PageRank of webpage i is based on its linking webpages (webpages j that link to i), but we don't just count the number of linking webpages, i.e., don't want to treat all linking webpages equally

Instead, we weight the links from different webpages

- Webpages that link to i, and have high PageRank scores themselves, should be given more weight
- Webpages that link to i, but link to a lot of other webpages in general, should be given less weight

Note that the first idea is circular! (But that's OK)

BrokenRank (almost PageRank) definition

Let $L_{ij} = 1$ if webpage j links to webpage i (written $j \to i$), and $L_{ij} = 0$ otherwise

Also let $m_j = \sum_{k=1}^{n} L_{kj}$, the total number of webpages that j links to

First we define something that's almost PageRank, but not quite, because it's broken. The BrokenRank p_i of webpage i is

$$p_i = \sum_{j \to i} \frac{p_j}{m_j} = \sum_{j=1}^{n} \frac{L_{ij}}{m_j} p_j$$

Does this match our ideas from the last slide? Yes: for $j \to i$, the weight is p_j/m_j—this increases with p_j, but decreases with m_j

BrokenRank in matrix notation

Written in matrix notation,

$$p = \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{pmatrix}, \quad L = \begin{pmatrix} L_{11} & L_{12} & \dots & L_{1n} \\ L_{21} & L_{22} & \dots & L_{2n} \\ \vdots & & & \\ L_{n1} & L_{n2} & \dots & L_{nn} \end{pmatrix},$$

$$M = \begin{pmatrix} m_1 & 0 & \dots & 0 \\ 0 & m_2 & \dots & 0 \\ \vdots & & & \\ 0 & 0 & \dots & m_n \end{pmatrix}$$

Dimensions: p is $n \times 1$, L and M are $n \times n$

Now re-express definition on the previous page: the BrokenRank vector p is defined as $p = LM^{-1}p$

Eigenvalues and eigenvectors

Let $A = LM^{-1}$, then $p = Ap$. This means that p is an eigenvector of the matrix A with eigenvalue 1

Great! Because we know how to compute the eigenvalues and eigenvectors of A, and there are even methods for doing this quickly when A is large and sparse (why is our A sparse?)

But wait ... do we know that A has an eigenvalue of 1, so that such a vector p exists? And even if it does exist, will be unique (well-defined)?

For these questions, it helps to interpret BrokenRank in terms of a Markov chain

BrokenRank as a Markov chain

Think of a Markov Chain as a random process that moves between states numbered $1, \ldots n$ (each step of the process is one move). Recall that for a Markov chain to have an $n \times n$ transition matrix P, this means $\mathrm{P}(\text{go from } i \text{ to } j) = P_{ij}$

Suppose $p^{(0)}$ is an n-dimensional vector giving initial probabilities. After one step, $p^{(1)} = P^T p^{(0)}$ gives probabilities of being in each state (why?)

Now consider a Markov chain, with the states as webpages, and with transition matrix A^T. Note that $(A^T)_{ij} = A_{ji} = L_{ji}/m_i$, so we can describe the chain as

$$\mathrm{P}(\text{go from } i \text{ to } j) = \begin{cases} 1/m_i & \text{if } i \to j \\ 0 & \text{otherwise} \end{cases}$$

(Check: does this make sense?) This is like a random surfer, i.e., a person surfing the web by clicking on links uniformly at random

Stationary distribution

A stationary distribution of our Markov chain is a probability vector p (i.e., its entries are ≥ 0 and sum to 1) with $p = Ap$

I.e., distribution after one step of the Markov chain is unchanged. Exactly what we're looking for: an eigenvector of A corresponding to eigenvalue 1

If the Markov chain is strongly connected, meaning that any state can be reached from any other state, then stationary distribution p exists and is unique. Furthermore, we can think of the stationary distribution as the of proportions of visits the chain pays to each state after a very long time (the ergodic theorem):

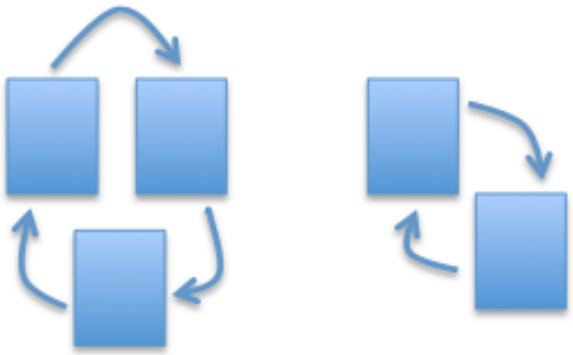
$$p_i = \lim_{t \to \infty} \frac{\#\text{ of visits to state } i \text{ in } t \text{ steps}}{t}$$

Our interpretation: the BrokenRank of p_i is the proportion of time our random surfer spends on webpage i if we let him go forever

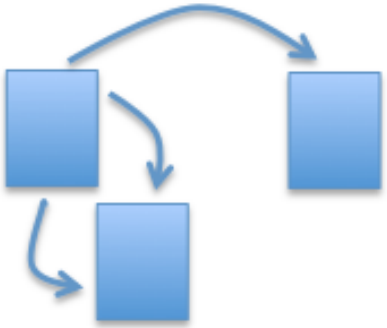
Why is BrokenRank broken?

There's a problem here. Our Markov chain—a random surfer on the web graph—is not strongly connected, in three cases (at least):

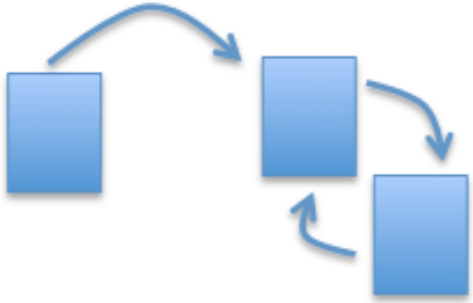
Disconnected components



Dangling links



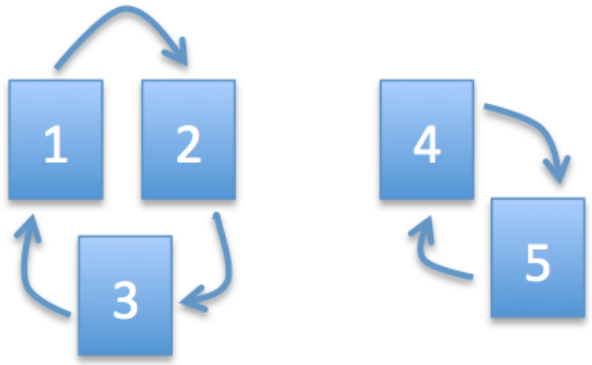
Loops



Actually, even for Markov chains that are not strongly connected, a stationary distribution always exists, but may nonunique

In other words, the BrokenRank vector p exists but is ambiguously defined

BrokenRank example



Here $A = LM^{-1} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$

(Check: matches both definitions?)

Here there are two eigenvectors of A with eigenvalue 1:

$$p = \begin{pmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \\ 0 \\ 0 \end{pmatrix} \quad \text{and} \quad p = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \frac{1}{2} \\ \frac{1}{2} \end{pmatrix}$$

These are totally opposite rankings!

PageRank definition

PageRank is given by a small modification of BrokenRank:

$$p_i = \frac{1-d}{n} + d\sum_{j=1}^{n} \frac{L_{ij}}{m_j} p_j,$$

where $0 < d < 1$ is a constant (apparently Google uses $d = 0.85$)

In matrix notation, this is

$$p = \left(\frac{1-d}{n} E + dLM^{-1}\right)p,$$

where E is the $n \times n$ matrix of 1s, subject to the constraint $\sum_{i=1}^{n} p_i = 1$

(Check: are these definitions the same? Show that the second definition gives the first. Hint: if e is the n-vector of all 1s, then $E = ee^T$, and $e^T p = 1$)

PageRank as a Markov chain

Let $A = \frac{1-d}{n}E + dLM^{-1}$, and consider as before a Markov chain with transition matrix A^T

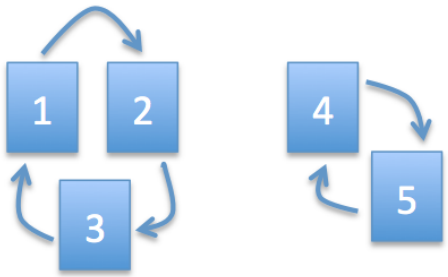
Well $(A^T)_{ij} = A_{ji} = (1-d)/n + dL_{ji}/m_i$, so the chain can be described as

$$\mathrm{P}(\text{go from } i \text{ to } j) = \begin{cases} (1-d)/n + d/m_i & \text{if } i \to j \\ (1-d)/n & \text{otherwise} \end{cases}$$

(Check: does this make sense?) The chain moves through a link with probability $(1-d)/n + d/m_i$, and with probability $(1-d)/n$ it jumps to an unlinked webpage

Hence this is like a random surfer with random jumps. Fortunately, the random jumps get rid of our problems: our Markov chain is now strongly connected. Therefore the stationary distribution (i.e., PageRank vector) p is unique

PageRank example



With $d = 0.85$, $A = \frac{1-d}{n} E + dLM^{-1}$

$$= \frac{0.15}{5} \cdot \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} + 0.85 \cdot \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 0.03 & 0.03 & 0.88 & 0.03 & 0.03 \\ 0.88 & 0.03 & 0.03 & 0.03 & 0.03 \\ 0.03 & 0.88 & 0.03 & 0.03 & 0.03 \\ 0.03 & 0.03 & 0.03 & 0.03 & 0.88 \\ 0.03 & 0.03 & 0.03 & 0.88 & 0.03 \end{pmatrix}$$

Now only one eigenvector of A with eigenvalue 1: $p = \begin{pmatrix} 0.2 \\ 0.2 \\ 0.2 \\ 0.2 \\ 0.2 \end{pmatrix}$

Computing the PageRank vector

Computing the PageRank vector p via traditional methods, i.e., an eigendecomposition, takes roughly n^3 operations. When $n = 10^{10}$, $n^3 = 10^{30}$. Yikes! (But a bigger concern would be memory ...)

Fortunately, much faster way to compute the eigenvector of A with eigenvalue 1: begin with any initial distribution $p^{(0)}$, and compute

$$\begin{aligned} p^{(1)} &= Ap^{(0)} \\ p^{(2)} &= Ap^{(1)} \\ &\vdots \\ p^{(t)} &= Ap^{(t-1)}, \end{aligned}$$

Then $p^{(t)} \to p$ as $t \to \infty$. In practice, we just repeatedly multiply by A until there isn't much change between iterations

E.g., after 100 iterations, operation count: $100n^2 \ll n^3$ for large n

Computation, continued

There are still important questions remaining about computing the PageRank vector p (with the algorithm presented on last slide):

1. How can we perform each iteration quickly (multiply by A quickly)?
2. How many iterations does it take (generally) to get a reasonable answer?

Broadly, the answers are:

1. Use the sparsity of web graph (how?)
2. Not very many if A large spectral gap (difference between its first and second largest absolute eigenvalues); the largest is 1, the second largest is $\leq d$

(PageRank in R: see the function `page.rank` in package `igraph`)

A basic web search

For a basic web search, given a query, we could do the following:

1. Compute the PageRank vector p once (Google recomputes this from time to time, to stay current)
2. Find the documents containing all words in the query
3. Sort these documents by PageRank, and return the top k (e.g., $k = 50$)

This is a little too simple ... but we can use the similarity scores learned last time, changing the above to:

3. Sort these documents by PageRank, and keep only the top K (e.g., $K = 5000$)
4. Sort by similarity to the query (e.g., normalized, IDF weighted distance), and return the top k (e.g., $k = 50$)

Google uses a combination of PageRank, similarity scores, and other techniques (it's proprietary!)

Variants/extensions of PageRank

A precursor to PageRank:

- Hubs and authorities: using link structure to determine "hubs" and "authorities"; a similar algorithm was used by Ask.com (Kleinberg (1997), "Authoritative Sources in a Hyperlinked Environment")

Following its discovery, there has been a huge amount of work to improve/extend PageRank—and not only at Google! There are many, many academic papers too, here are a few:

- Intelligent surfing: pointing surfer towards textually relevant webpages (Richardson and Domingos (2002), "The Intelligent Surfer: Probabilistic Combination of Link and Content Information in PageRank")
- TrustRank: pointing surfer away from spam (Gyongyi et al. (2004), "Combating Web Spam with TrustRank")
- PigeonRank: pigeons, the real reason for Google's success (`http://www.google.com/onceuponatime/technology/pigeonrank.html`)

Recap: PageRank

PageRank is a ranking for webpages based on their importance. For a given webpage, its PageRank is based on the webpages that link to it; it helps if these linking webpages have high PageRank themselves; it hurts if these linking webpages also link to a lot of other webpages

We defined it by modifying a simpler ranking system (BrokenRank) that didn't quite work. The PageRank vector p corresponds to the eigenvector of a particular matrix A corresponding to eigenvalue 1. Can also be explained in terms of a Markov chain, interpreted as a random surfer with random jumps. These jumps were crucial, because they made the chain strongly connected, and guaranteed that the PageRank vector (stationary distribution) p is unique

We can compute p by repeatedly multiplying by A. PageRank can be combined with similarity scores for a basic web search

Next time: clustering

Not quite as easy as apples with apples and oranges with oranges

10 Chapter 24: Network Flows

Reading: Chapter 24

Lecture Notes for Chapter 24: Maximum Flow

Chapter 24 overview

Network flow

[The third and fourth editions treat flow networks differently from the first two editions. The concept of net flow is gone, except that we do discuss net flow across a cut. Skew symmetry is also gone, as is implicit summation notation. The third and fourth editions count flows on edges directly. We find that although the mathematics is not quite as slick as in the first two editions, the approach in the newer editions matches intuition more closely, and therefore students tend to pick it up more quickly.]

Use a graph to model material that flows through conduits.

Each edge represents one conduit, and has a ***capacity***, which is an upper bound on the ***flow rate*** = units/time.

Can think of edges as pipes of different sizes. But flows don't have to be of liquids. The textbook has an example where a flow is how many trucks per day can ship hockey pucks between cities.

Want to compute the maximum rate that material can be shipped from a designated ***source*** to a designated ***sink***.

Flow networks

$G = (V, E)$ directed.

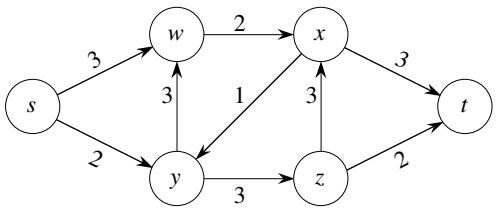
Each edge (u, v) has a ***capacity*** $c(u, v) \geq 0$.

If $(u, v) \notin E$, then $c(u, v) = 0$.

If $(u, v) \in E$, then reverse edge $(v, u) \notin E$. (Can work around this restriction.)

Source vertex s, ***sink*** vertex t, assume $s \rightsquigarrow v \rightsquigarrow t$ for all $v \in V$, so that each vertex lies on a path from source to sink.

Example: *[Edges are labeled with capacities.]*



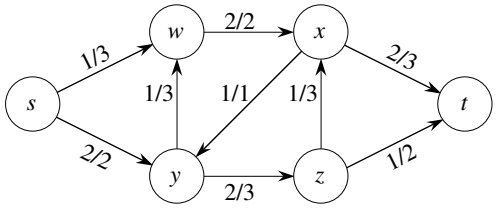
Flow

A function $f : V \times V \to \mathbb{R}$ satisfying

- ***Capacity constraint:*** For all $u, v \in V, 0 \le f(u,v) \le c(u,v)$,
- ***Flow conservation:*** For all $u \in V - \{s,t\}$, $\underbrace{\sum_{v \in V} f(v,u)}_{\text{flow into } u} = \underbrace{\sum_{v \in V} f(u,v)}_{\text{flow out of } u}$.

  Equivalently, $\sum_{v \in V} f(u,v) - \sum_{v \in V} f(v,u) = 0$.

[Add flows to previous example. Edges here are labeled as flow/capacity. Leave on board.]



- Note that all flows are $\le$ capacities.
- Verify flow conservation by adding up flows at a couple of vertices.
- Note that all flows $= 0$ is legitimate.

Value of flow f $= |f|$
$= \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$
$=$ flow out of source $-$ flow into source .

In the example above, value of flow $f = |f| = 3$.

Antiparallel edges

Definition of flow network does not allow both (u,v) and (v,u) to be edges. These edges would be ***antiparallel***.

What if really need antiparallel edges?

- Choose one of them, say (u,v).
- Create a new vertex v'.
- Replace (u,v) by two new edges (u,v') and (v',v), with $c(u,v') = c(v',v) = c(u,v)$.
- Get an equivalent flow network with no antiparallel edges.

Multiple sources and sinks

If you need multiple sources $s_1, \ldots, s_m$, create a single ***supersource*** s with edges (s, s_i) for $i = 1, \ldots, m$ and infinite capacity on each edge.

Same idea for multiple sinks: create a single ***supersink*** with an infinite-capacity edge from each sink to the supersink.

Now there are just one source and one sink.

Cuts

A ***cut*** (S, T) of flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$.

- Similar to cut used in minimum spanning trees, except that here the graph is directed, and require $s \in S$ and $t \in T$.

For flow f, the ***net flow*** across cut (S, T) is

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u) \ .$$

Capacity of cut (S, T) is

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v) \ .$$

A ***minimum cut*** of G is a cut whose capacity is minimum over all cuts of G.

Asymmetry between net flow across a cut and capacity of a cut: For capacity, count only capacities of edges going from S to T. Ignore edges going in the reverse direction. For net flow, count flow on all edges across the cut: flow on edges going from S to T minus flow on edges going from T to S.

In previous example, consider the cut $S = \{s, w, y\}, T = \{x, z, t\}$.

$$\begin{aligned} f(S, T) &= \underbrace{f(w, x) + f(y, z)}_{\text{from } S \text{ to } T} - \underbrace{f(x, y)}_{\text{from } T \text{ to } S} \\ &= 2 + 2 - 1 \\ &= 3 \ . \end{aligned}$$

$$\begin{aligned} c(S, T) &= \underbrace{c(w, x) + c(y, z)}_{\text{from } S \text{ to } T} \\ &= 2 + 3 \\ &= 5 \ . \end{aligned}$$

Now consider the cut $S = \{s, w, x, y\}, T = \{z, t\}$.

$$\begin{aligned} f(S, T) &= \underbrace{f(x, t) + f(y, z)}_{\text{from } S \text{ to } T} - \underbrace{f(z, x)}_{\text{from } T \text{ to } S} \\ &= 2 + 2 - 1 \\ &= 3 \ . \end{aligned}$$

$$\begin{aligned} c(S, T) &= \underbrace{c(x, t) + c(y, z)}_{\text{from } S \text{ to } T} \\ &= 3 + 3 \\ &= 6 \ . \end{aligned}$$

Same flow as previous cut, higher capacity.

Lemma

For any cut (S, T), $f(S, T) = |f|$.
(Net flow across the cut equals value of the flow.)
[Leave on board.]

[This proof is much more involved than the proof in the first two editions. You might want to omit it, or just give the intuition that no matter where you cut the pipes in a network, you'll see the same flow volume coming out of the openings.]

Proof Rewrite flow conservation: for any $u \in V - \{s, t\}$,

$$\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) = 0 \, .$$

Take definition of $|f|$ and add in left-hand side of above equation, summed over all vertices in $S - \{s\}$. Above equation applies to each vertex in $S - \{s\}$ (since $t \notin S$ and obviously $s \notin S - \{s\}$), so just adding in lots of 0s:

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{u \in S - \{s\}} \left(\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) \right) \, .$$

Expand right-hand summation and regroup terms:

$$\begin{aligned}
|f| &= \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{u \in S - \{s\}} \sum_{v \in V} f(u, v) - \sum_{u \in S - \{s\}} \sum_{v \in V} f(v, u) \\
&= \sum_{v \in V} \left(f(s, v) + \sum_{u \in S - \{s\}} f(u, v) \right) - \sum_{v \in V} \left(f(v, s) + \sum_{u \in S - \{s\}} f(v, u) \right) \\
&= \sum_{v \in V} \sum_{u \in S} f(u, v) - \sum_{v \in V} \sum_{u \in S} f(v, u) \, .
\end{aligned}$$

Partition V into $S \cup T$ and split each summation over V into summations over S and T:

$$\begin{aligned}
|f| &= \sum_{v \in S} \sum_{u \in S} f(u, v) + \sum_{v \in T} \sum_{u \in S} f(u, v) - \sum_{v \in S} \sum_{u \in S} f(v, u) - \sum_{v \in T} \sum_{u \in S} f(v, u) \\
&= \sum_{v \in T} \sum_{u \in S} f(u, v) - \sum_{v \in T} \sum_{u \in S} f(v, u) \\
&\qquad + \left(\sum_{v \in S} \sum_{u \in S} f(u, v) - \sum_{v \in S} \sum_{u \in S} f(v, u) \right) \, .
\end{aligned}$$

Summations within parentheses are the same, since $f(x, y)$ appears once in each summation, for any $x, y \in V$. These summations cancel:

$$\begin{aligned}
|f| &= \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u) \\
&= f(S, T) \, .
\end{aligned}$$

■ (lemma)

Corollary

The value of any flow $\leq$ capacity of any cut.
[Leave on board.]

Proof Let (S, T) be any cut, f be any flow.

$$\begin{aligned}
|f| &= f(S,T) && \text{(lemma)}\\
&= \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u) && \text{(definition of } f(S,T))\\
&\le \sum_{u\in S}\sum_{v\in T} f(u,v) && (f(v,u) \ge 0)\\
&\le \sum_{u\in S}\sum_{v\in T} c(u,v) && \text{(capacity constraint)}\\
&= c(S,T)\ . && \text{(definition of } c(S,T)) \quad \blacksquare \text{ (corollary)}
\end{aligned}$$

Therefore, maximum flow $\le$ capacity of minimum cut.

Will see a little later that this is in fact an equality.

The Ford-Fulkerson method

Residual network

Given a flow f in network $G = (V, E)$.

Consider a pair of vertices $u, v \in V$.

How much additional flow can be pushed directly from u to v?
That's the ***residual capacity***,

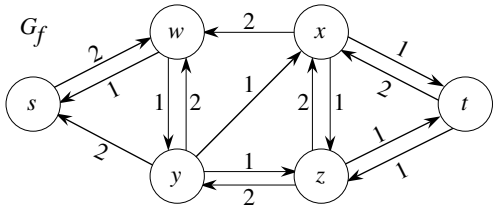
$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E\ , \\ f(v,u) & \text{if } (v,u) \in E\ , \\ 0 & \text{otherwise (i.e., } (u,v), (v,u) \notin E)\ . \end{cases}$$

The ***residual network*** is $G_f = (V, E_f)$, where

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}\ .$$

Each edge of the residual network can admit a positive flow.

For our example:



Every edge $(u, v) \in E_f$ corresponds to an edge $(u, v) \in E$ or $(v, u) \in E$.

Therefore, $|E_f| \le 2\,|E|$.

Residual network is similar to a flow network, except that it may contain antiparallel edges ((u, v) and (v, u)). Can define a flow in a residual network that satisfies the definition of a flow, but with respect to capacities c_f in G_f.

Given flows f in G and f' in G_f, define $(f \uparrow f')$, the ***augmentation*** of f by f', as a function $V \times V \to \mathbb{R}$:

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

for all $u, v \in V$.

Intuition: Increase the flow on (u,v) by $f'(u,v)$ but decrease it by $f'(v,u)$ because pushing flow on the reverse edge in the residual network decreases the flow in the original network. Also known as ***cancellation***.

Lemma
Given a flow network G, a flow f in G, and the residual network G_f, let f' be a flow in G_f. Then $f \uparrow f'$ is a flow in G with value $|f \uparrow f'| = |f| + |f'|$.

[See textbook for proof. It has a lot of summations in it. Probably not worth writing on the board.]

Augmenting path

A simple path $s \rightsquigarrow t$ in G_f.

- Admits more flow along each edge.
- Like a sequence of pipes through which can squirt more flow from s to t.

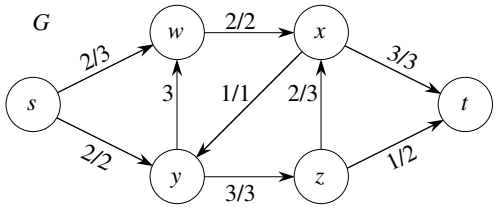
How much more flow can be pushed from s to t along augmenting path p? That is the ***residual capacity*** of p:

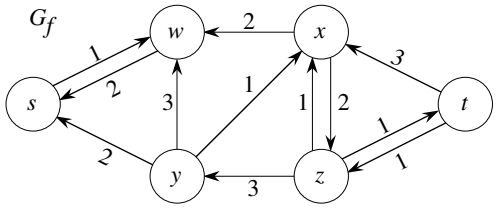
$c_f(p) = \min\{c_f(u,v) : (u,v) \text{ is on } p\}$.

For our example, consider the augmenting path $p = \langle s, w, y, z, x, t \rangle$.

Minimum residual capacity is 1.

After pushing 1 additional unit along p: *[Continue from G left on board from before. Edge (y,w) has $f(y,w) = 0$, which we omit, showing only $c(y,w) = 3$.]*





Observe that G_f now has no augmenting path. Why? No edges cross the cut $(\{s,w\}, \{x,y,z,t\})$ in the forward direction in G_f. So no path can get from s to t.

Claim that the flow shown in G is a maximum flow.

Lemma

Given flow network G, flow f in G, residual network G_f. Let p be an augmenting path in G_f. Define $f_p : V \times V \to \mathbb{R}$:

$$f_p(u,v) = \begin{cases} c_f(p) & \text{if } (u,v) \text{ is on } p \ , \\ 0 & \text{otherwise} \ . \end{cases}$$

Then f_p is a flow in G_f with value $|f_p| = c_f(p) > 0$.

Corollary

Given flow network G, flow f in G, and an augmenting path p in G_f, define f_p as in lemma. Then $f \uparrow f_p$ is a flow in G with value $|f \uparrow f_p| = |f| + |f_p| > |f|$.

Theorem (Max-flow min-cut theorem)

The following are equivalent:

1. f is a maximum flow.
2. G_f has no augmenting path.
3. $|f| = c(S,T)$ for some cut (S,T).

Proof

(1) $\Rightarrow$ (2): Show the contrapositive: if G_f has an augmenting path, then f is not a maximum flow. If G_f has augmenting path p, then by the above corollary, $f \uparrow f_p$ is a flow in G with value $|f| + |f_p| > |f|$, so that f was not a maximum flow.

(2) $\Rightarrow$ (3): Suppose G_f has no augmenting path. Define

$S = \{v \in V : \text{there exists a path } s \rightsquigarrow v \text{ in } G_f\}$,

$T = V - S$.

Must have $t \in T$; otherwise there is an augmenting path.
Therefore, (S,T) is a cut.
Consider $u \in S$ and $v \in T$:

- If $(u,v) \in E$, must have $f(u,v) = c(u,v)$; otherwise, $(u,v) \in E_f \Rightarrow v \in S$.
- If $(v,u) \in E$, must have $f(v,u) = 0$; otherwise, $c_f(u,v) = f(v,u) > 0 \Rightarrow (u,v) \in E_f \Rightarrow v \in S$.
- If $(u,v), (v,u) \notin E$, must have $f(u,v) = f(v,u) = 0$.

Then,

$$\begin{aligned} f(S,T) &= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{v \in T} \sum_{u \in S} f(v,u) \\ &= \sum_{u \in S} \sum_{v \in T} c(u,v) - \sum_{v \in T} \sum_{u \in S} 0 \\ &= c(S,T) \ . \end{aligned}$$

By lemma, $|f| = f(S,T) = c(S,T)$.

(3) $\Rightarrow$ (1): An earlier corollary says that the value of any flow is $\leq$ the capacity of any cut, so that $|f| \leq c(S,T)$.
Therefore, $|f| = c(S,T) \Rightarrow f$ is a max flow. ■ (theorem)

Ford-Fulkerson algorithm

Keep augmenting flow along an augmenting path until there is no augmenting path.

Represent the flow attribute using the usual dot-notation, but on an edge: $(u, v).f$.

```
FORD-FULKERSON(G, s, t)
  for each edge (u, v) ∈ G.E
      (u, v).f = 0
  while there is an augmenting path p in G_f
      c_f(p) = min {c_f(u, v) : (u, v) is in p}
      for each edge (u, v) in p        // augment f by c_f(p)
          if (u, v) ∈ G.E
              (u, v).f = (u, v).f + c_f(p)
          else (v, u).f = (v, u).f − c_f(p)
  return f
```

Analysis

If capacities are all integer, then each augmenting path raises $|f|$ by ≥ 1. If max flow is f^*, then need $\leq |f^*|$ iterations $\Rightarrow$ time is $O(E\,|f^*|)$.
[Handwaving—see textbook for better explanation.]

Note that this running time is *not* polynomial in input size. It depends on $|f^*|$, which is not a function of $|V|$ and $|E|$.

If capacities are rational, can scale them to integers.

If irrational, FORD-FULKERSON might never terminate!

Edmonds-Karp algorithm

Do FORD-FULKERSON, but compute augmenting paths by BFS of G_f. Augmenting paths are shortest paths $s \rightsquigarrow t$ in G_f, with all edge weights $= 1$.

Edmonds-Karp runs in $O(VE^2)$ time.

To prove, need to look at distances to vertices in G_f.

Let $\delta_f(u, v) =$ shortest path distance u to v in G_f, with unit edge weights.

Lemma

For all $v \in V - \{s, t\}$, $\delta_f(s, v)$ increases monotonically with each flow augmentation.

Proof Suppose there exists $v \in V - \{s, t\}$ such that some flow augmentation causes $\delta_f(s, v)$ to decrease. Will derive a contradiction.

Let f be the flow before the first augmentation that causes a shortest-path distance to decrease, f' be the flow afterward.

Let v be a vertex with minimum $\delta_{f'}(s, v)$ whose distance was decreased by the augmentation, so that $\delta_{f'}(s, v) < \delta_f(s, v)$.

Let a shortest path s to v in $G_{f'}$ be $s \rightsquigarrow u \rightarrow v$, so that $(u, v) \in E_{f'}$ and $\delta_{f'}(s, v) = \delta_{f'}(s, u) + 1$. (Or $\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1$.)

Since $\delta_{f'}(s, u) < \delta_{f'}(s, v)$ and how we chose v, we have $\delta_{f'}(s, u) \geq \delta_f(s, u)$.

Claim

$(u, v) \notin E_f$.

Proof of claim If $(u, v) \in E_f$, then

$$\begin{aligned}\delta_f(s, v) &\le \delta_f(s, u) + 1 \quad \text{(triangle inequality)}\\ &\le \delta_{f'}(s, u) + 1\\ &= \delta_{f'}(s, v)\ ,\end{aligned}$$

contradicting $\delta_{f'}(s, v) < \delta_f(s, v)$. ■ (claim)

How can $(u, v) \notin E_f$ and $(u, v) \in E_{f'}$?

The augmentation must increase flow v to u.

Since Edmonds-Karp augments along shortest paths, the shortest path s to u in G_f has (v, u) as its last edge.

Therefore,

$$\begin{aligned}\delta_f(s, v) &= \delta_f(s, u) - 1\\ &\le \delta_{f'}(s, u) - 1\\ &= \delta_{f'}(s, v) - 2\ ,\end{aligned}$$

contradicting $\delta_{f'}(s, v) < \delta_f(s, v)$.

Therefore, v cannot exist. ■ (lemma)

Theorem

Edmonds-Karp performs $O(VE)$ augmentations.

Proof Suppose p is an augmenting path and $c_f(u, v) = c_f(p)$. Then call (u, v) a ***critical*** edge in G_f, and it disappears from the residual network after augmenting along p.

≥ 1 edge on any augmenting path is critical.

Will show that each of the $|E|$ edges can become critical $\le |V|/2$ times.

Consider $u, v \in V$ such that either $(u, v) \in E$ or $(v, u) \in E$. Since augmenting paths are shortest paths, when (u, v) becomes critical the first time, $\delta_f(s, v) = \delta_f(s, u) + 1$.

Augment flow, so that (u, v) disppears from the residual network. This edge cannot reappear in the residual network until flow from u to v decreases, which happens only if (v, u) is on an augmenting path in $G_{f'}$: $\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1$. ($f'$ is flow when this occurs.)

By lemma, $\delta_f(s, v) \le \delta_{f'}(s, v) \Rightarrow$

$$\begin{aligned}\delta_{f'}(s, u) &= \delta_{f'}(s, v) + 1\\ &\ge \delta_f(s, v) + 1\\ &= \delta_f(s, u) + 2\ .\end{aligned}$$

Therefore, from the time (u, v) becomes critical to the next time, distance of u from s increases by ≥ 2. Initially, distance to u is ≥ 0, and augmenting path can't have s, u, and t as intermediate vertices.

Therefore, until u becomes unreachable from source, its distance is $\leq |V| - 2$
$\Rightarrow$ after (u, v) becomes critical the first time, it can become critical
$\leq (|V| - 2)/2 = |V|/2 - 1$ times more
$\Rightarrow$ (u, v) can become critical $\leq |V|/2$ times.

Since $O(E)$ pairs of vertices can have an edge between them in residual network, total # of critical edges during execution of Edmonds-Karp is $O(VE)$. Since each augmenting path has ≥ 1 critical edge, have $O(VE)$ augmentations. ■ (theorem)

Use BFS to find each augmenting path in $O(E)$ time $\Rightarrow$ $O(VE^2)$ time.

Can get better bounds. *[Push-relabel algorithms in the first three editions of the textbook give $O(V^3)$. The two sections on push-relabel algorithm were dropped from the fourth edition but are available from the MIT Press website for the book.]*

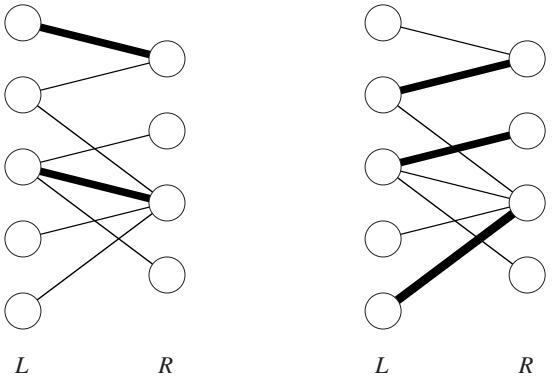
Maximum bipartite matching

Example of a problem that can be solved by turning it into a flow problem.

$G = (V, E)$ (undirected) is ***bipartite*** if there is a partition of the vertices $V = L \cup R$ such that all edges in E go between L and R.

A ***matching*** is a subset of edges $M \subseteq E$ such that for all $v \in V$, ≤ 1 edge of M is incident on v. (Vertex v is ***matched*** if an edge of M is incident on it; otherwise ***unmatched***).

Maximum matching: a matching of maximum cardinality. (M is a maximum matching if $|M| \geq |M'|$ for all matchings M'.)



matching maximum matching

[Edges in matchings are drawn with heavy lines.]

Problem

Given a bipartite graph (with the partition), find a maximum matching.

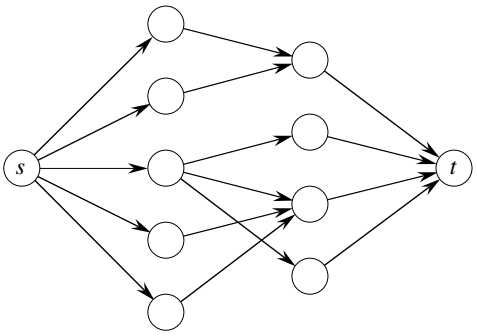
Application

Matching planes to routes.

- $L =$ set of planes.
- $R =$ set of routes.
- $(u, v) \in E$ if plane u can fly route v.
- Want maximum # of routes to be served by planes.

Given G, define flow network $G' = (V', E')$.

- $V' = V \cup \{s, t\}$.
- $E' = \{(s, u) : u \in L\} \cup \{(u, v) : (u, v) \in E\} \cup \{(v, t) : v \in R\}$.
- $c(u, v) = 1$ for all $(u, v) \in E'$.



Each vertex in V has ≥ 1 incident edge $\Rightarrow |E| \geq |V|/2$.

Therefore, $|E| \leq |E'| = |E| + |V| \leq 3|E|$.

Therefore, $|E'| = \Theta(E)$.

Find a max flow in G'. Textbook shows that it will have integer values for all (u, v).

Use edges (u, v) such that $u \in L$ and $v \in R$ that carry flow of 1 in matching.

Textbook proves that this method produces a maximum matching.

[The next chapter, Chapter 25, has a better algorithm (Hopcroft-Karp) to find a maximum matching, as well as other algorithms based on bipartite matchings.]

Following are notes from Jeff Erikson (who has a good textbook)

10.4 Ford and Fulkerson's augmenting-path algorithm

Ford and Fulkerson's proof of the Maxflow-Mincut Theorem immediately suggests an algorithm to compute maximum flows: Starting with the zero flow, repeatedly augment the flow along **any** path from s to t in the residual graph, until there is no such path.

This algorithm has an important but straightforward corollary:

Integrality Theorem. *If all capacities in a flow network are integers, then there is a maximum flow such that the flow through every edge is an integer.*

Proof: We argue by induction that after each iteration of the augmenting path algorithm, all flow values and all residual capacities are integers.

- Before the first iteration, all flow values are 0 (which is an integer), and all residual capacities are the original capacities, which are integers by definition.
- In each later iteration, the induction hypothesis implies that the capacity F of the augmenting path is an integer, so augmenting changes the flow on each edge, and therefore the residual capacity of each edge, by an integer.

In particular, each iteration of the augmenting path algorithm increases the value of the flow by a positive integer. It follows that the algorithm eventually halts and returns a maximum flow. □

If every edge capacity is an integer, then conservatively, the Ford-Fulkerson algorithm halts after at most $|f^*|$ iterations, where f^* is the actual maximum flow. In each iteration, we can build the residual graph G_f and perform a whatever-first-search to find an augmenting path in $O(E)$ time. Thus, in this setting, the algorithm runs in $\boldsymbol{O(E|f^*|)}$ time in the worst case.

Jack Edmonds and Richard Karp observed that this running time analysis is essentially tight. Consider the 4-node network in Figure 10.7, where X is some large integer. The maximum flow in this network is clearly $2X$. However, Ford-Fulkerson might alternate between pushing one unit of flow along the augmenting path $s \to u \to v \to t$ and then pushing one unit of flow along the augmenting path $s \to v \to u \to t$, leading to a running time of $\Theta(X) = \Omega(|f^*|)$.

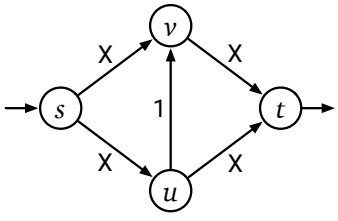


Figure 10.7. Edmonds and Karp's bad example for the Ford-Fulkerson algorithm.

Ford and Fulkerson's algorithm is usually fast in practice, and it is always fast when the maximum flow value $|f^*|$ is small, but without further constraints on the augmenting paths, this is *not* an efficient algorithm in worst case. Edmonds and Karp's bad example network can be described using only $O(\log X)$ bits; thus, the running time of Ford-Fulkerson is actually *exponential* in the input size.

♥Irrational Capacities

But what if the capacities are *not* integers? If we multiply all the capacities by the same (positive) constant, the maximum flow increases everywhere by the same constant factor. It follows that if all the edge capacities are *rational*, then the Ford-Fulkerson algorithm eventually halts, although still in exponential time (in the number of bits used to described the input).

However, if we allow *irrational* capacities, the algorithm can actually loop forever, always finding smaller and smaller augmenting paths. Worse yet, this infinite sequence of augmentations may not even converge to the maximum flow, or even to a significant fraction of the maximum flow! The smallest network that exhibits this bad behavior was discovered by Uri Zwick in 1993.[2]

Consider the six-node network shown in Figure 10.8. Six of the nine edges have some large integer capacity X, two have capacity 1, and one has capacity $\phi = (\sqrt{5}-1)/2 \approx 0.618034$, chosen so that $1-\phi = \phi^2$. To prove that the Ford-Fulkerson algorithm can get stuck, we can watch the residual capacities of the three horizontal edges as the algorithm progresses. (The residual capacities of the other six edges will always be at least $X-3$.)

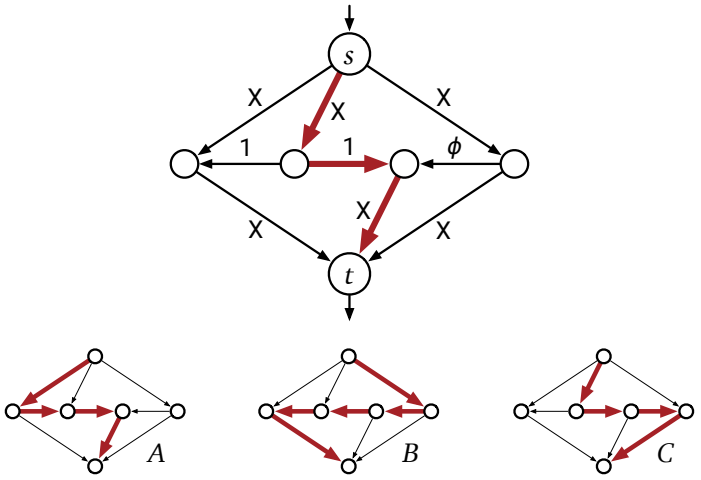


Figure 10.8. Uri Zwick's non-terminating flow example, and three augmenting paths.

Suppose the Ford-Fulkerson algorithm starts by choosing the central augmenting path, shown at the top of Figure 10.8. The three horizontal edges, in

[2] In 1962, Ford and Fulkerson described a more complex network, with 10 vertices and 48 edges, with the same bad behavior.

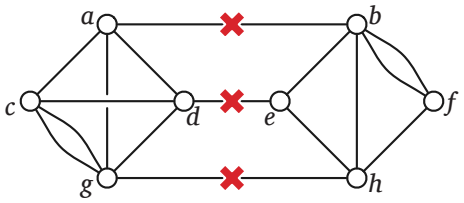
> **Jaques:** *But, for the seventh cause; how did you find the quarrel on the seventh cause?*
> **Touchstone:** *Upon a lie seven times removed:–bear your body more seeming, Audrey:–as thus, sir. I did dislike the cut of a certain courtier's beard: he sent me word, if I said his beard was not cut well, he was in the mind it was: this is called the Retort Courteous. If I sent him word again 'it was not well cut,' he would send me word, he cut it to please himself: this is called the Quip Modest. If again 'it was not well cut,' he disabled my judgment: this is called the Reply Churlish. If again 'it was not well cut,' he would answer, I spake not true: this is called the Reproof Valiant. If again 'it was not well cut,' he would say I lied: this is called the Counter-cheque Quarrelsome: and so to the Lie Circumstantial and the Lie Direct.*
> **Jaques:** *And how oft did you say his beard was not well cut?*
> **Touchstone:** *I durst go no further than the Lie Circumstantial, nor he durst not give me the Lie Direct; and so we measured swords and parted.*
>
> — William Shakespeare, *As You Like It*, Act V, Scene 4 (1600)

13 Randomized Minimum Cut

13.1 Setting Up the Problem

This lecture considers a problem that arises in robust network design. Suppose we have a connected multigraph[1] G representing a communications network like the UIUC telephone system, the Facebook social network, the internet, or Al-Qaeda. In order to disrupt the network, an enemy agent plans to remove some of the edges in this multigraph (by cutting wires, placing police at strategic drop-off points, or paying street urchins to 'lose' messages) to separate it into multiple components. Since his country is currently having an economic crisis, the agent wants to remove as few edges as possible to accomplish this task.

More formally, a *cut* partitions the nodes of G into two nonempty subsets. The *size* of the cut is the number of *crossing edges*, which have one endpoint in each subset. Finally, a *minimum* cut in G is a cut with the smallest number of crossing edges. The same graph may have several minimum cuts.



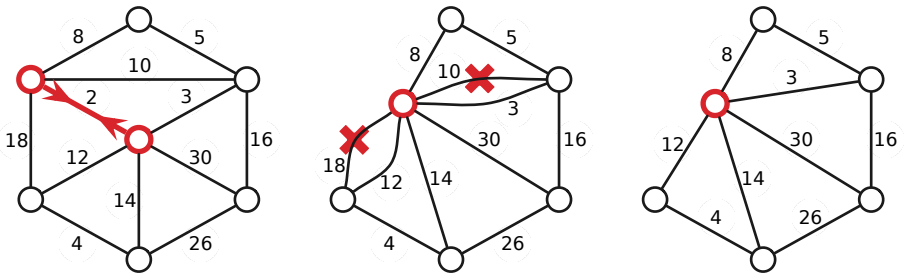
A multigraph whose minimum cut has three edges.

This problem has a long history. The classical deterministic algorithms for this problem rely on *network flow* techniques, which are discussed in another lecture. The fastest such algorithms (that we will discuss) run in $O(n^3)$ time and are fairly complex; we will see some of these later in the semester. Here I'll describe a relatively simple randomized algorithm discovered by David Karger when he was a Ph.D. student.[2]

[1] A multigraph allows multiple edges between the same pair of nodes. Everything in this lecture could be rephrased in terms of simple graphs where every edge has a non-negative weight, but this would make the algorithms and analysis slightly more complicated.

[2] David R. Karger*. Random sampling in cut, flow, and network design problems. Proc. 25th STOC, 648–657, 1994.

Karger's algorithm uses a primitive operation called ***edge contraction***. Suppose u and v are vertices that are connected by an edge in some multigraph G. To contract the edge $\{u, v\}$, we create a new node called uv, replace any edge of the form $\{u, w\}$ or $\{v, w\}$ with a new edge $\{uv, w\}$, and then delete the original vertices u and v. Equivalently, contracting the edge shrinks the edge down to nothing, pulling the two endpoints together. The new contracted graph is denoted $G/\{u, v\}$. We don't allow self-loops in our multigraphs; if there are multiple edges between u and v, contracting any one of them deletes them all.



A graph G and two contracted graphs $G/\{b, e\}$ and $G/\{c, d\}$.

Any edge in an n-vertex graph can be contracted in $O(n)$ time, assuming the graph is represented as an adjacency list; I'll leave the precise implementation details as an easy exercise.

The correctness of our algorithms will eventually boil down the following simple observation: For any cut in $G/\{u, v\}$, there is cut in G with exactly the same number of crossing edges. In fact, in some sense, the 'same' edges form the cut in both graphs. The converse is not necessarily true, however. For example, in the picture above, the original graph G has a cut of size 1, but the contracted graph $G/\{c, d\}$ does not.

This simple observation has two immediate but important consequences. First, contracting an edge cannot decrease the minimum cut size. More importantly, contracting an edge increases the minimum cut size if and only if that edge is part of *every* minimum cut.

13.2 Blindly Guessing

Let's start with an algorithm that tries to *guess* the minimum cut by randomly contracting edges until only two vertices remain.

```
GuessMinCut(G):
  for i ← n downto 2
      pick a random edge e in G
      G ← G/e
  return the only cut in G
```

Because each contraction requires $O(n)$ time, this algorithm runs in $O(n^2)$ time. Our earlier observations imply that as long as we never contract an edge that lies in every minimum cut, our algorithm will actually guess correctly. But how likely is that?

Suppose G has only one minimum cut—if it actually has more than one, just pick your favorite—and this cut has size k. Every vertex of G must lie on at least k edges; otherwise, we could separate that vertex from the rest of the graph with an even smaller cut. Thus, the number of incident vertex-edge pairs is at least kn. Since every edge is incident to exactly two vertices, G must have at least $kn/2$ edges. That implies that if we pick an edge in G uniformly at random, the probability of picking an edge in the minimum cut is at most $2/n$. In other words, the probability that we don't screw up on the very first step is at least $1 - 2/n$.

Once we've contracted the first random edge, the rest of the algorithm proceeds recursively (with independent random choices) on the remaining $(n-1)$-node graph. So the overall probability $P(n)$ that GUESSMINCUT returns the true minimum cut is given by the recurrence

$$P(n) \geq \frac{n-2}{n} \cdot P(n-1)$$

with base case $P(2) = 1$. We can expand this recurrence into a product, most of whose factors cancel out immediately.

$$P(n) \geq \prod_{i=3}^{n} \frac{i-2}{i} = \frac{\prod_{i=3}^{n}(i-2)}{\prod_{i=3}^{n} i} = \frac{\prod_{j=1}^{n-2} j}{\prod_{i=3}^{n} i} = \boxed{\frac{2}{n(n-1)}}$$

13.3 Blindly Guessing Over and Over

That's not very good. Fortunately, there's a simple method for increasing our chances of finding the minimum cut: run the guessing algorithm many times and return the smallest guess. Randomized algorithms folks like to call this idea ***amplification***.

```
KargerMinCut(G):
  mink ← ∞
  for i ← 1 to N
      X ← GuessMinCut(G)
      if |X| < mink
          mink ← |X|
          minX ← X
  return minX
```

Both the running time and the probability of success will depend on the number of iterations N, which we haven't specified yet.

First let's figure out the probability that KARGERMINCUT returns the actual minimum cut. The only way for the algorithm to return the wrong answer is if GUESSMINCUT fails N times in a row. Since each guess is independent, our probability of success is at least

$$1 - \left(1 - \frac{2}{n(n-1)}\right)^N \leq 1 - e^{-2N/n(n-1)},$$

by The World's Most Useful Inequality $1 + x \leq e^x$. By making N larger, we can make this probability arbitrarily close to 1, but never equal to 1. In particular, if we set $\boldsymbol{N = c\binom{n}{2}\ln n}$ for some constant c, then KARGERMINCUT is correct with probability at least

$$1 - e^{-c \ln n} = 1 - \frac{1}{n^c}.$$

When the failure probability is a polynomial fraction, we say that the algorithm is correct *with high probability*. Thus, KARGERMINCUT computes the minimum cut of any n-node graph in $\boldsymbol{O(n^4 \log n)}$ time with high probability.

If we make the number of iterations even larger, say $N = n^2(n-1)/2$, the success probability becomes $1 - e^{-n}$. When the failure probability is exponentially small like this, we say that the algorithm is correct with *very* high probability. In practice, very high probability is usually overkill; high probability is enough. (Remember, there is a small but non-zero probability that your computer will transform itself into a kitten before your program is finished.)

13.4 Not-So-Blindly Guessing

The $O(n^4 \log n)$ running time is actually comparable to some of the simpler flow-based algorithms, but it's nothing to get excited about. But we can improve our guessing algorithm, and thus decrease the number of iterations in the outer loop, by exploiting the observation we made earlier:

> *As the graph shrinks, the probability of contracting an edge in the minimum cut increases.*

At first the probability is quite small, only $2/n$, but near the end of execution, when the graph has only three vertices, we have a $2/3$ chance of screwing up!

A simple technique for working around this increasing probability of error was developed by David Karger and Cliff Stein.[3] Their idea was to group the first several random contractions a "safe" phase, so that the cumulative probability of screwing up is relatively small and a "dangerous" phase, which is much more likely to screw up.

The safe phase shrinks the graph from n nodes to about $n/2$ nodes, using a sequence of about $n/2$ random contractions.[4] Following our earlier analysis, the probability that *none* of these safe contractions touches the minimum cut is at least

$$\prod_{i=n/2+1}^{n} \frac{i-2}{i} = \frac{(n/2)(n/2-1)}{n(n-1)} = \frac{n-2}{4(n-1)} \approx \frac{1}{4}.$$

To get around the danger of the dangerous phase, we use amplification: we run the dangerous phase *four times* and keep the best of the four answers. Naturally, we treat the dangerous phase recursively, so we actually obtain a recursion tree with degree 4, which expands as we get closer to the base case, instead of a single path. More formally, the algorithm looks like this:

```
Contract(G, m):
  for i ← n downto m
      pick a random edge e in G
      G ← G/e
  return G
```

```
BetterGuess(G):
  if n < 1000
      use brute force
  else
      H ← Contract(G, n/α)
      X1 ← BetterGuess(H, α)
      X2 ← BetterGuess(H, α)
      X3 ← BetterGuess(H, α)
      X4 ← BetterGuess(H, α)
      return min{X1, X2, X3, X4}
```

At first glance, it may look like we are performing exactly the same BetterGuess computation four times in a row, but remember that Contract (and therefore BetterGuess) is randomized. Each recursive call to BetterGuess contracts a different, independently chosen random set of edges. Thus, X_1, X_2, X_3, and X_4 are almost always four completely different cuts!

Why four recursive calls, and not two or six? We're facing a tradeoff between the speed of the algorithm and its probability of success. More recursive calls makes the algorithm more likely to succeed but slower; fewer recursive calls makes the algorithm faster but *considerably* more likely to fail. Four recursive calls turns out to be the right balance, both guaranteeing a fast algorithm and a reasonably large success probability.

[3] David R. Karger* and Cliff Stein. An $\tilde{O}(n^2)$ algorithm for minimum cuts. Proc. 25th STOC, 757–765, 1993.

[4] I'm deliberately simplifying the analysis here to expose more intuition. More formally, if we contract from n to $\lceil n/2 \rceil + 1$ nodes, the probability that no minimum cut edge is contracted is strictly greater than $1/2$.

BETTERGUESS correctly returns the minimum cut unless (1) none of the edges of the minimum cut are CONTRACTed and (2) *all four* recursive calls return the incorrect cut for H. Let $P(n)$ denote the probability that BETTERGUESS returns a minimum cut of an n-node graph. Then for each index i, the probability that X_i is the minimum cut of H is $P(n/2)$. Because the four recursive calls are independent, we obtain the following recurrence for $P(n)$, with base case $P(n) = 1$ for all $n \leq 8$.

$$\begin{aligned}
P(n) &= \Pr[\text{mincut in } H \text{ is mincut in } G] \cdot \Pr[\text{some } X_i \text{ is mincut in } H] \\
&\geq \frac{1}{4}\Big(1 - \Pr[\text{no } X_i \text{ is mincut in } H]\Big) \\
&\geq \frac{1}{4}\Big(1 - \Pr[X_1 \text{ is not mincut in } H]^4\Big) \\
&\geq \frac{1}{4}\Big(1 - \Big(1 - \Pr[X_1 \text{ is mincut in } H]\Big)^4\Big) \\
&\geq \frac{1}{4}\left(1 - \left(1 - P\left(\frac{n}{2}\right)\right)^4\right) \\
&= \frac{1}{4} - \frac{1}{4}\left(1 - P\left(\frac{n}{2}\right)\right)^4
\end{aligned}$$

Using a series of transformations, Karger and Stein prove that $P(n) = \Omega(1/\log n)$. I've included a proof at the end of this note.

For the running time, we get a simple recurrence that is easily solved using recursion trees:

$$T(n) = O(n^2) + 4T\left(\frac{n}{2}\right) = O(n^2 \log n)$$

So all this splitting and recursing has slowed down the guessing algorithm slightly, but the probability of failure is *exponentially* smaller!

Let's express the lower bound $P(n) = \Omega(1/\log n)$ explicitly as $P(n) \geq \alpha/\ln n$ for some constant α. (Karger and Stein's proof implies $\alpha > 2$). If we call BETTERGUESS $N = c\ln^2 n$ times, for some new constant c, the overall probability of success is at least

$$1 - \left(1 - \frac{\alpha}{\ln n}\right)^{c\ln^2 n} \geq 1 - e^{-(c/\alpha)\ln n} = 1 - \frac{1}{n^{c/\alpha}}.$$

By setting c sufficiently large, we can bound the probability of failure by an arbitrarily small polynomial function of n. In other words, we now have an algorithm that computes the minimum cut with high probability in only $\boldsymbol{O(n^2 \log^3 n)}$ time!

*13.5 Solving the Karger-Stein recurrence

Recall the following recursive inequality for the probability that BETTERGUESS successfully finds a minimum cut of an n-node graph:

$$P(n) \geq \frac{1}{4} - \frac{1}{4}\left(1 - P\left(\frac{n}{2}\right)\right)^4$$

Let $\overline{P}(n)$ be the function that satisfies this recurrence with equality; clearly, $P(n) \geq \overline{P}(n)$.

We can solve this rather ugly recurrence for $\overline{P}(n)$ through a series of functional transformations. First, consider the function $p(k) = \overline{P}(2^k)$; this function satisfies the recurrence

$$p(k) = \frac{1}{4} - \frac{1}{4}(1 - p(k-1))^4,$$

Next, define $d(k) = 1/p(k)$; this new function obeys the reciprocal recurrence

$$d(k) = \frac{4}{1-\left(1-\frac{1}{d(k-1)}\right)^4} = \frac{4\,d(k-1)^4}{d(k-1)^4-(d(k-1)-1)^4}$$

Straightforward algebraic manipulation[5] implies the inequalities

$$x < \frac{4x^4}{x^4-(x-1)^4} \le x+3$$

for all $x \ge 1$. Thus, as long as $d(k-1) \ge 1$, we have the simple recursive inequalities

$$d(k-1) < d(k) \le d(k-1)+3.$$

Our original base case $P(2) = 1$ implies the base case $d(1) = 1$. It immediately follows by induction that $d(k) \le 3k-2$, and therefore

$$P(n) \ge \overline{P}(n) = p(\lg n) = \frac{1}{d(\lg n)} \ge \frac{1}{3\lg n - 2} = \Omega\left(\frac{1}{\log n}\right),$$

as promised. Whew!

★★★ Galton-Watson process?

Exercises

1. (a) Suppose you had an algorithm to compute the minimum spanning tree of a graph in $O(m)$ time, where m is the number of edges in the input graph.[6] Use this algorithm as a subroutine to improve the running time of GuessMinCut from $O(n^2)$ to $O(m)$.

   (b) Describe and analyze an algorithm to compute the *maximum-weight edge* in the minimum spanning tree of a graph with m edges in $O(m)$ time. *[Hint: We can compute the median edge weight in $O(m)$ time. How do you tell quickly if that's too big or too small?]*

   (c) Use your algorithm from part (b) to improve the running time of GuessMinCut from $O(n^2)$ to $O(m)$.

2. Suppose you are given a graph G with weighted edges, and your goal is to find a cut whose total weight (not just number of edges) is smallest.

   (a) Describe and analyze an algorithm to select a random edge of G, where the probability of choosing edge e is proportional to the weight of e.

   (b) Prove that if you use the algorithm from part (a), instead of choosing edges uniformly at random, the probability that GuessMinCut returns a minimum-weight cut is still $\Omega(1/n^2)$.

[5] otherwise known as Wolfram Alpha

[6] In fact, there *is* a randomized algorithm—due to Philip Klein, David Karger, and Robert Tarjan—that computes the minimum spanning tree of any graph in $O(m)$ *expected* time. The fastest deterministic algorithm known in 2015 runs in $O(m\alpha(m))$ time, where $\alpha(\cdot)$ is the inverse Ackermann function.

(c) What is the running time of your modified GUESSMINCUT algorithm?

3. Prove that GUESSMINCUT returns the *second* smallest cut in its input graph with probability $\Omega(1/n^3)$. (The second smallest cut could be significantly larger than the minimum cut.)

4. Consider the following generalization of the BETTERGUESS algorithm, where we pass in a real parameter $\alpha > 1$ in addition to the graph G.

```
BETTERGUESS(G, α):
  if n < 1000
      use brute force
  else
      H ← CONTRACT(G, n/α)
      X_1 ← BETTERGUESS(H, α)
      X_2 ← BETTERGUESS(H, α)
      X_3 ← BETTERGUESS(H, α)
      X_4 ← BETTERGUESS(H, α)
      return min{X_1, X_2, X_3, X_4}
```

Assume for this question that the input graph G has a unique minimum cut.

(a) What is the running time of the modified algorithm, as a function of n and α? *[Hint: Consider the cases $\alpha < 2$, $\alpha = 2$, and $\alpha > 2$ separately.]*

(b) What is the probability that CONTRACT$(G, n/\alpha)$ does not contract any edge in the minimum cut in G? Give both an exact expression involving both n and α, and a simple approximation in terms of just α. *[Hint: When $\alpha = 2$, the probability is approximately $1/4$.]*

(c) Estimate the probability that BETTERGUESS(G, α) returns the minimum cut in G, by adapting the solution to the Karger-Stein recurrence. *[Hint: Consider the cases $\alpha < 2$, $\alpha = 2$, and $\alpha > 2$ separately.]*

(d) Suppose we iterate BETTERGUESS(G, α) until we are guaranteed to see the minimum cut with high probability. What is the running time of the resulting algorithm? For which value of α is this running time minimized?

(e) Suppose we modify BETTERGUESS(G, α) further, to make k recursive calls instead of four. What is the best choice of α, as a function of k? What is the resulting running time? What is the best choice for k and α?

11 Complexity: Chapter 34, 35

Reading: These notes is all you need, courtesy of Steven Skiena (who also wrote a good textbook). Our CLRS textbook gets into a lot more detail than we need at this level.

The Theory of NP-Completeness

Several times this semester we have encountered problems for which we couldn't find efficient algorithms, such as the traveling salesman problem.

We also couldn't prove exponential-time lower bounds for these problems.

The theory of NP-completeness, developed by Stephen Cook and Richard Karp, provides the tools to show that all of these problems were really the same problem.

The Main Idea

Suppose I gave you the following algorithm to solve the *bandersnatch* problem:

Bandersnatch(G)

> Convert G to an instance of the Bo-billy problem Y.
> Call the subroutine Bo-billy on Y to solve this instance.
> Return the answer of Bo-billy(Y) as the answer to G.

Such a translation from instances of one type of problem to instances of another type such that answers are preserved is called a *reduction*.

Topic: Problems and Reductions

What is a Problem?

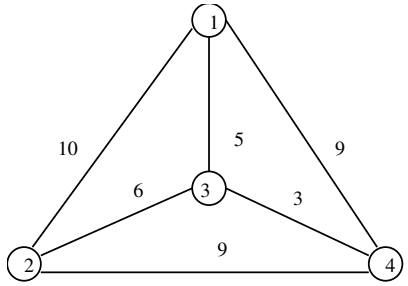
A *problem* is a general question, with parameters for the input and conditions on what is a satisfactory answer or solution.

Example: The Traveling Salesman

Problem: Given a weighted graph G, what tour $\{v_1, v_2, ..., v_n\}$ minimizes $\Sigma_{i=1}^{n-1} d[v_i, v_{i+1}] + d[v_n, v_1]$.

What is an Instance?

An instance is a problem with the input parameters specified.
TSP instance: $d[v_1, d_2] = 10$, $d[v_1, d_3] = 5$, $d[v_1, d_4] = 9$, $d[v_2, d_3] = 6$, $d[v_2, d_4] = 9$, $d[v_3, d_4] = 3$



Solution: $\{v_1, v_2, v_3, v_4\}$ cost= 27

Decision Problems

A problem with answers restricted to yes and no is called a *decision problem*.

Most interesting optimization problems can be phrased as decision problems which capture the essence of the computation.

For convenience, from now on we will talk *only* about decision problems.

The Traveling Salesman Decision Problem

Given a weighted graph G and integer k, does there exist a traveling salesman tour with cost $\leq k$?

Using binary search and the decision version of the problem we can find the optimal TSP solution.

Problem of the Day

Suppose we are given a subroutine which can solve the traveling salesman decision problem in, say, linear time. Give an efficient algorithm to find the actual TSP tour by making a polynomial number of calls to this subroutine.

Satisfiability

Consider the following logic problem:

Instance: A set V of variables and a set of clauses C over V.

Question: Does there exist a satisfying truth assignment for C?

Example 1: $V = v_1, v_2$ and $C = \{\{v_1, \overline{v}_2\}, \{\overline{v}_1, v_2\}\}$

A clause is satisfied when at least one literal in it is true. C is satisfied when $v_1 = v_2 =$true.

Not Satisfiable

Example 2: $V = v_1, v_2$,

$$C = \{\{v_1, v_2\}, \{v_1, \overline{v}_2\}, \{\overline{v}_1\}\}$$

Although you try, and you try, and you try and you try, you can get no satisfaction.

There is no satisfying assigment since v_1 must be false (third clause), so v_2 must be false (second clause), but then the first clause is unsatisfiable!

3-Satisfiability

Instance: A collection of clause C where each clause contains *exactly* 3 literals, boolean variable v.

Question: Is there a truth assignment to v so that each clause is satisfied?

Note that this is a more restricted problem than SAT. If 3-SAT is NP-complete, it implies SAT is NP-complete but not visa-versa, perhaps long clauses are what makes SAT difficult? After all, 1-Sat is trivial!

3-SAT is NP-Complete

To prove it is complete, we give a reduction from $Sat \propto 3-Sat$. We will transform each clause independantly based on its *length*.
Suppose the clause C_i contains k literals.

- If $k = 1$, meaning $C_i = \{z_1\}$, create two new variables v_1, v_2 and four new 3-literal clauses:

  $\{v_1, v_2, z_1\}, \{v_1, \overline{v}_2, z_1\}, \{\overline{v}_1, v_2, z_1\}, \{\overline{v}_1, \overline{v}_2, z_1\}$.

  Note that the only way all four of these can be satisfied is if z is true.

- If $k = 2$, meaning $\{z_1, z_2\}$, create one new variable v_1 and two new clauses: $\{v_1, z_1, z_2\}$, $\{\overline{v}_1, z_1, z_2\}$
- If $k = 3$, meaning $\{z_1, z_2, z_3\}$, copy into the 3-SAT instance as it is.
- If $k > 3$, meaning $\{z_1, z_2, ..., z_n\}$, create $n - 3$ new variables and $n - 2$ new clauses in a chain: $\{v_i, z_i, \overline{v}_i\}$, ...

Why does the Chain Work?

If none of the original variables in a clause are true, there is no way to satisfy all of them using the additional variable:

$$(F, F, T), (F, F, T), \ldots, (F, F, F)$$

But if any literal is true, we have $n - 3$ free variables and $n - 3$ remaining 3-clauses, so we can satisfy all clauses.

$(F, F, T), (F, F, T), \ldots, (\mathbf{F}, \mathbf{T}, \mathbf{F}), \ldots, (T, F, F), (T, F, F)$

Any SAT solution will also satisfy the 3-SAT instance and any 3-SAT solution sets variables giving a SAT solution, so the problems are equivallent.

4-Sat and 2-Sat

A slight modification to this construction would prove 4-SAT, or 5-SAT,... also NP-complete.
However, it breaks down when we try to use it for 2-SAT, since there is no way to stuff anything into the chain of clauses.

Topic: Vertex Cover

The Power of 3-SAT

Now that we have shown 3-SAT is NP-complete, we may use it for further reductions. Since the set of 3-SAT instances is smaller and more regular than the SAT instances, it will be easier to use 3-SAT for future reductions.
Remember the direction of the reduction!

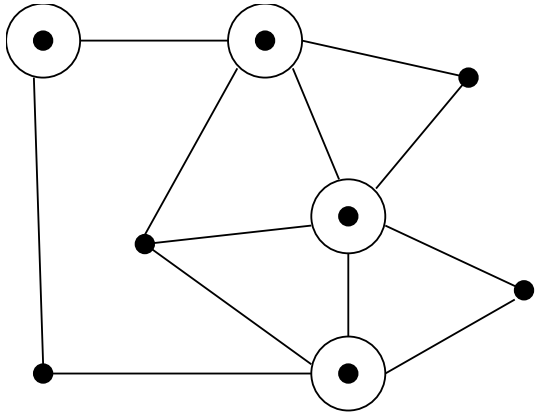
$$Sat \propto 3 - Sat \propto X$$

A Perpetual Point of Confusion

Note carefully the direction of the reduction.
We must transform *every* instance of a known NP-complete problem to an instance of the problem we are interested in.
If we do the reduction the other way, all we get is a slow way to solve x, by using a subroutine which probably will take exponential time.

Vertex Cover

Instance: A graph $G = (V, E)$, and integer $k \leq V$
Question: Is there a subset of at most k vertices such that every $e \in E$ has at least one vertex in the subset?

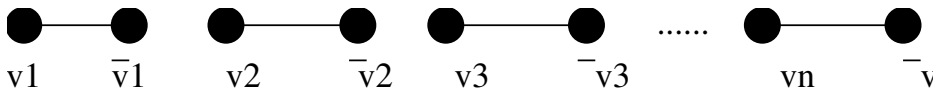


Here, four of the eight vertices suffice to cover.
It is easy to find *a* vertex cover of a graph: just take all the vertices. The hard part is to cover with as small a set as possible.

Vertex cover is NP-complete

To prove completeness, we show reduce 3-SAT to VC. From a 3-SAT instance with n variables and c clauses, we construct a graph with $2n + 3c$ vertices.

For each variable, we create two vertices connected by an edge:



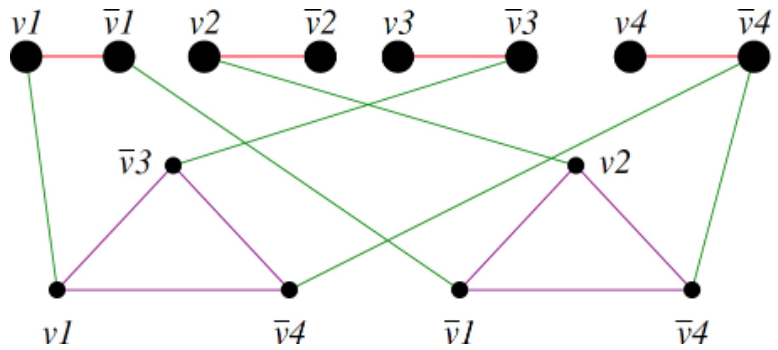
To cover each of these edges, at least n vertices must be in the cover, one for each pair.

Clause Gadgets

For each clause, we create three new vertices, one for each literal in each clause. Connect these in a triangle.

At least two vertices per triangle must be in the cover to take care of edges in the triangle, for a total of at least $2c$ vertices.

Finally, we will connect each literal in the flat structure to the corresponding vertices in the triangles which share the same literal.



Claim: G has a VC of size $n + 2c$ iff S is Satisfiable

Any cover of G must have at least $n + 2c$ vertices. To show that our reduction is correct, we must show that:

Every satisfying truth assignment gives a cover.

Select the n vertices cooresponding to the true literals to be in the cover.

Since it is a satisfying truth assignment, at least one of the three cross edges associated with each clause must already be covered - pick the other two vertices to complete the cover.∎

Every vertex cover gives a satisfying truth assignment

Every vertex cover must contain n first stage vertices and $2c$ second stage vertices. Let the first stage vertices define the truth assignment.

To give the cover, at least one cross-edge must be covered, so the truth assignment satisfies.

Topic: Clique and Independent Set

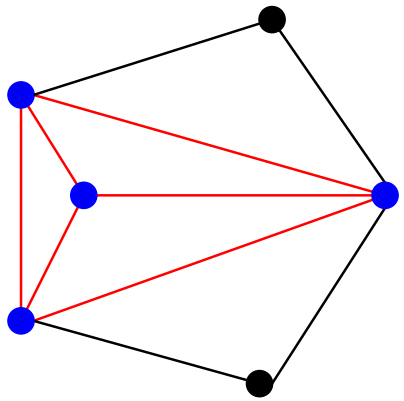
Starting from the Right Problem

As you can see, the reductions can be very clever and very complicated. While theoretically any NP-complete problem can be reduced to any other one, choosing the correct one makes finding a reduction much easier.

$$3 - Sat \propto VC$$

Maximum Independent Set

Instance: A graph $G = (V, E)$ and integer $j \leq v$.
Question: Does the graph contain an independent of j vertices, ie. is there a subset of v of size j such that no pair of vertices in the subset defines an edge of G?



Example: this graph contains an independent set of size 2.
Recall that the movie star scheduling problem was a version of maximum independent set.

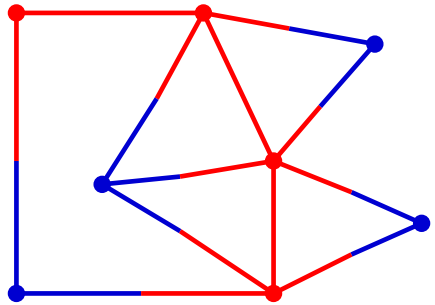
Proving Graph Problems Hard

When talking about graph problems, it is most natural to work from a graph problem - the only NP-complete one we have is vertex cover!

If you take a graph and find its vertex cover, the remaining vertices form an independent set, meaning there are no edges between any two vertices in the independent set.

Why? If there were such an edge the rest of the vertices could not have been a vertex cover.

Maximum Independent Set is NP-Complete

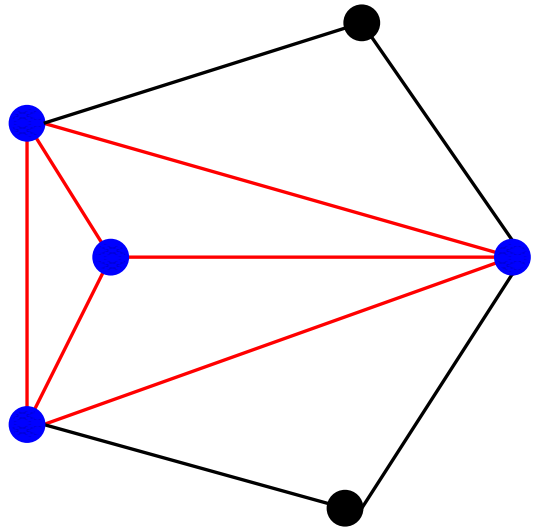


The smallest vertex cover gives the biggest independent set, and so the problems are equivallent: delete the subset of vertices in one from V to get the other!

Thus finding the maximum independent set is NP-complete!

Maximum Clique

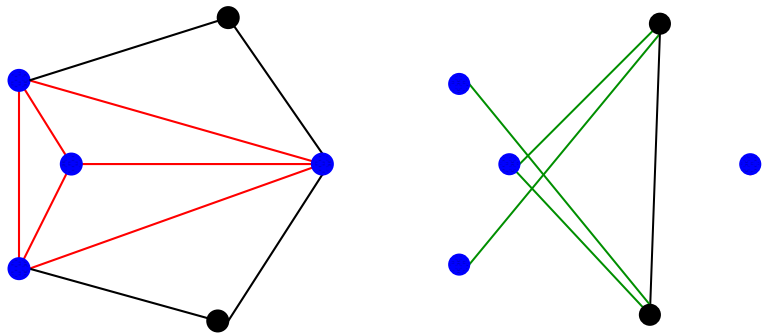
Instance: A graph $G = (V, E)$ and integer $j \leq v$.
Question: Does the graph contain a clique of j vertices, ie. is there a subset of v of size j such that every pair of vertices in the subset defines an edge of G?



Example: this graph contains a clique of size 5.

From Independent Set

In an independent set, there are no edges between two vertices. In a clique, there are always between two vertices. Thus if we complement a graph (have an edge iff there was no edge in the original graph), a clique becomes an independent set and an independent set becomes a clique!



P versus NP

- A problem is in NP if a given answer can be checked in polynomial time.
- A problem is in P if it can be solve in time polynomial in the size of the input.

Satisfiability is in NP, since we can guess an assignment of (true, false) to the literals and check it in polynomial time.
The precise distinction between P or NP is somewhat technical, requiring formal language theory and Turing machines to state correctly.
But the real issue is the difference between finding solutions or verifying them.

Classifying Example Problems

- In P – Is there a path from s to t in G of length less than k.
- In NP – Is there a TSP tour in G of length less than k. Given the tour, it is easy to add up the costs and convince me it is correct.
- *Not* in NP – How many TSP tours are there in G of length less than k. Since there can be an exponential number of them, we cannot count them all in polynomial time.

Don't let this issue confuse you – the important idea here is of reductions as a way of proving hardness.

Other NP-complete/hard Problems

- Bin Packing - how many bins of a given size do you need to hold n items of variable size?
- Chromatic Number - how many colors do you need to color a graph?
- $N \times N$ checkers - does black have a forced win from a given position?

Open: Graph Isomorphism, Factoring Integers.

Polynomial or Exponential?

Just changing a problem a little can make the difference between it being in P or NP-complete:

	P	NP-complete
	Shortest Path	Longest Path
	Eulerian Circuit	Hamiltonian Circuit
	Edge Cover	Vertex Cover

The first thing you should do when you suspect a problem might be NP-complete is look in Garey and Johnson, *Computers and Intractability*.

Is $P = NP$**?**

This remains the greatest open problem in Computer Science.

Some will say it is true for $N = 1$ or $P = 0$. :-)

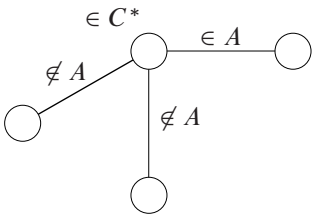
Theorem

APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.

Proof Already shown that it runs in polynomial time. Need to show that it produces a vertex cover, and that the size of the vertex cover it produces is within a factor of 2 of optimal.

The procedure produces a vertex cover because it loops until every edge in E' has been covered by some vertex in C.

Now let C^* be an optimal vertex cover. Need to show that $|C| \le 2|C^*|$. Let A be the set of edges chosen in the **while** loop. To cover the edges in A, any vertex cover—including C^*—must include at least one endpoint of each edge in A. No two edges in A share an endpoint (once we pick an edge, all other edges incident on its endpoints are removed from E').



For every vertex in C^*, there is at most one edge of A, so that $|C^*| \ge |A|$.

Each time APPROX-VERTEX-COVER chooses an edge, neither of the endpoints can be in C. That means each edge chosen puts two vertices into C, so that $|C| = 2|A|$. Therefore,

$$\begin{aligned} |C| &= 2|A| \\ &\le 2|C^*| \ . \end{aligned}$$

■

How did this proof work when we don't even know the size of an optimal vertex cover? We got a lower bound on its size ($|C^*| \ge |A|$) and then found an upper bound on the size of the approximate vertex cover that was within a factor of 2 of the lower bound.

Traveling-salesperson problem

Also known as TSP.

Input: A complete undirected graph $G = (V, E)$ with a nonnegative cost $c(u, v)$ for each edge $(u, v) \in E$.

Output: A ***tour*** (a hamiltonian cycle—a cycle that visits every vertex) with minimum total cost.

Also the optimization version of an NP-complete problem.

With the triangle inequality

Assume that the triangle inequality holds:

$c(u, w) \le c(u, v) + c(v, w)$

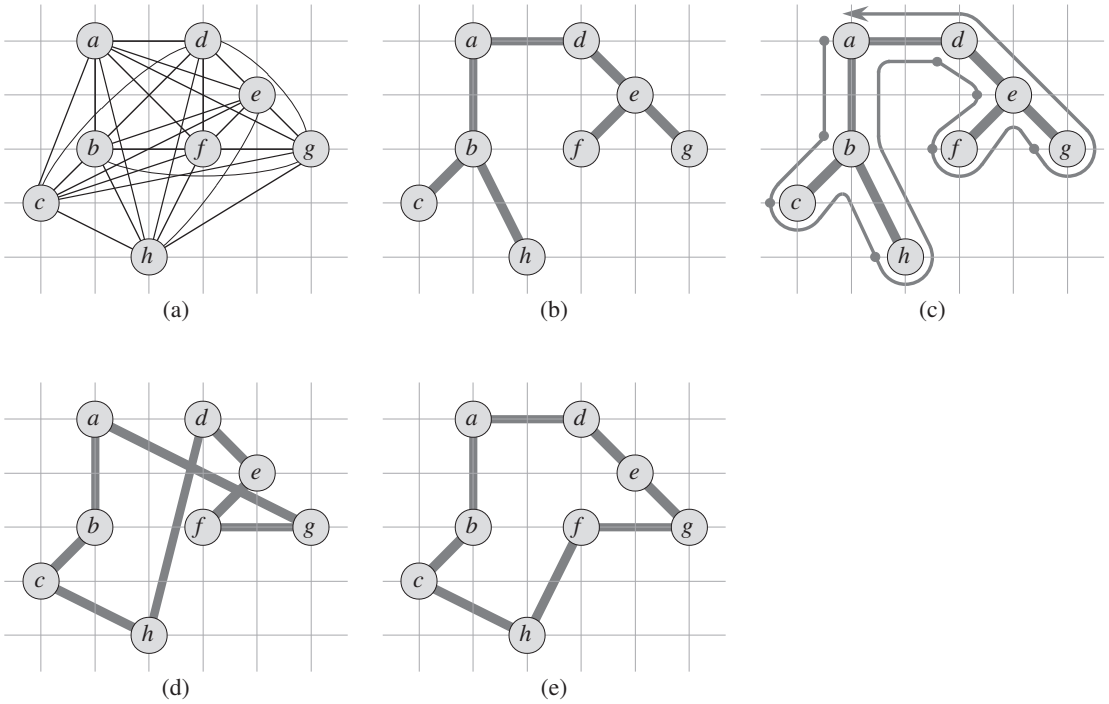
for all $u, v, w \in V$. The triangle inequality does not always hold, but it does when the vertices are points in the plane and the costs are euclidean distances between vertices. (Other cost functions can satisfy the triangle inequality.)

TSP is NP-complete even when the triangle inequality holds. The book shows that if the triangle inequality does not hold, then there is no polynomial-time approximation algorithm with a constant approximation ratio unless P = NP. So we'll concentrate on when the triangle inequality holds.

Idea: Construct a minimum spanning tree, do a preorder walk of the tree, and construct the TSP tour in the order in which each vertex is first visited during the preorder walk.

APPROX-TSP(G, c)
  select a vertex $r \in G.V$ to be a root vertex
  call MST-PRIM(G, c, r) to construct a minimum spanning tree T
  perform a preorder walk of T, and make a list H of vertices,
      ordered according to when first visited
  return the list H as the tour

Example: Using euclidean distance.



(a) A complete undirected graph.

(b) A minimum spanning tree with root a.

(c) A preorder walk of the spanning tree, with a dot next to each vertex the first time it's visited.

(d) The tour obtained by visiting the vertices in that order. Its total cost is approximately 19.074.

(e) An optimal tour with cost approximately 14.715.

Time: Easy to make Prim's algorithm run in $O(V^2)$ time. The rest is $\Theta(V)$, so total running time is $O(V^2)$.

Theorem

APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the traveling-salesperson problem with the triangle inequality.

Proof Already shown that it runs in polynomial time. It's obvious that it produces a tour. Need to show that the cost of the tour is at most 2 times the cost of an optimal tour. For any subset $A \subseteq E$ of edges, let $c(A) = \sum_{(u,v)\in A} c(u,v)$ be the total cost of the edges in A.

Let H^* be an optimal tour. Delete any edge from H^*, and get a spanning tree T^* whose cost is no greater than the cost of H^*, since all edge weights are nonnegative. Since the minimum spanning tree T computed in APPROX-TSP has a cost at most that of T^*, have

$$c(T) \le c(T^*) \le c(H^*) .$$

Instead of the preorder walk, think of the ***full walk*** of T, which lists each vertex whenever it's visited, including when returning from a visit to a subtree. In the example above, a full walk of T visits vertices in the order

$$a, b, c, b, h, b, a, d, e, f, e, g, e, d, a .$$

The full walk, call it W, visits every edge exactly twice, so $c(W) = 2c(T)$. *[We're extending the cost notation to allow edges to appear mulitple times—twice in this case—in the summation.]* So now we have

$$c(W) = 2c(T) \le 2c(H^*) .$$

The full walk W is not a tour, since it visits some vertices more than once. Instead, remove vertices from W to make it a tour by leaving only the first instance of each vertex in W. By the triangle inequality, the total length cannot increase by removing a vertex: if the full walk contains vertices u, v, w in order, then by removing v, we have $c(u,w) \le c(u,v) + c(v,w)$. What we get is the tour H, where $c(H) \le c(W)$. And now

$$\begin{aligned} c(H) &\le c(W) \\ &\le 2c(H^*) . \end{aligned}$$

■

[This algorithm is nowhere near the best approximation algorithm for the traveling-salesperson problem. Much better approximations are possible.]

Without the triangle inequality

[This material assumes that students understand that there is a polynomial-time algorithm for the hamiltonian-cycle problem if and only if P = NP. *Otherwise, you need to present some background material on* P = NP.*]*

If we cannot assume that the cost function satisfies the triangle inquality, then we cannot find good approximate tours in polynomial time unless P = NP.

Theorem

If P ≠ NP, then for any constant $\rho \geq 1$, there is no polynomial-time ρ-approximation algorithm for the general traveling-salesperson problem.

Proof

Idea: Proof by contradiction. Assume that a polynomial-time ρ-approximation algorithm exists. Given an instance of the hamiltonian-cycle problem, create in polynomial time an instance of TSP that has a small optimal value if the original graph has a hamiltonian cycle, but a high optimal value if the original graph does not have a hamiltonian cycle. Then use the ρ-approximation algorithm to get an approximate bound on the cost of the TSP tour. This bound is low if and only if the original graph has a hamiltonian cycle. Then we have a way to solve the hamiltonian-cycle problem in polynomial time.

Given an undirected graph $G = (V, E)$ as an instance of the hamiltonian-cycle problem, suppose that there is a ρ-approximation algorithm for TSP. Create an instance $G' = (V, E')$ of TSP. G' is a complete graph:

$E' = \{(u, v) : u, v \in V \text{ and } u \neq v\}$.

Without loss of generality, assume that ρ is an integer, rounding up if necessary. Define integer edge costs in E':

$$c(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E \ , \\ \rho\,|V| + 1 & \text{if } (u, v) \notin E \ . \end{cases}$$

Easy to create G' and c in polynomial time.

We will show that G has hamiltonian cycle if and only if G' with cost c has a tour with cost $|V|$, which means that G has hamiltonian cycle if and only if the approximation algorithm A would find a tour with cost at most $\rho\,|V|$.

If G has a hamiltonian cycle H, then the edges of H each have cost 1 in G', and so G' has a tour with cost $|V|$. The approximation algorithm would find a tour with cost at most $\rho\,|V|$.

If G does not have a hamiltonian cycle, then any tour of G' must use some edge not in E. This edge has cost $\rho\,|V| + 1$. The tour of G' has at most $|V| - 1$ edges with cost 1, and so the cost of the tour is at least

$(\rho\,|V| + 1) + (|V| - 1) = \rho\,|V| + |V|$.

Therefore, the lowest tour cost that the approximation algorithm A could give is $\rho\,|V| + |V|$, which is greater than $|V|$.

So if A gives a tour with cost at most $\rho\,|V|$, then G has a hamiltonian cycle, and if A gives a tour with cost at least $\rho\,|V| + |V|$, then G does not contain a hamiltonian cycle. Thus, there is no polynomial-time ρ-approximation algorithm for the general TSP unless P = NP. ■