Problem Transformations

CSC 411

Richard Kelley

Last Time

- Divide and Conquer
  - Karatsuba Multiplication
  - Strassen Multiplication

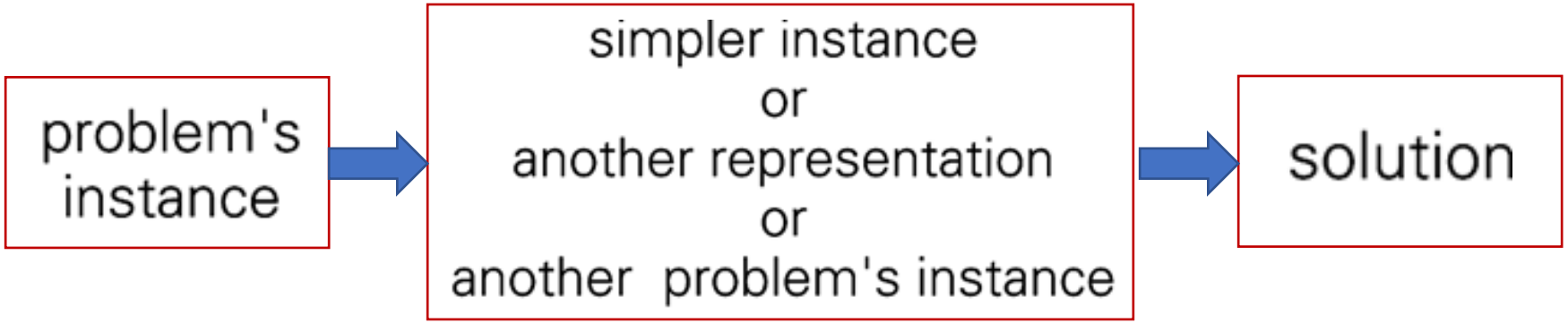
Today

- Problem Transformation
  - Presorting
  - Gaussian Elimination
  - Graph Connectivity via Linear Algebra
- Transformations and Reductions
  - Decision Problems
  - Languages
  - Complexity Classes
  - Completeness

“Transform and Conquer”

- This group of techniques solves a problem by a ***transformation***
  - to a simpler/more convenient instance of the same problem (*instance simplification*)
  - to a different representation of the same instance (*representation change*)
  - to a different problem for which an algorithm is already available (*problem reduction*)

Transform and Conquer strategy



- Suppose you have 3 problems:
  - searching
  - computing the median (selection problem)
  - checking if all elements are distinct (element uniqueness)

- Are these similar problems?

Instance simplification - Presorting

- Solve a problem's instance by transforming it into another simpler or easier instance of the same problem

Presorting

Many problems involving lists are easier when list is sorted.

- searching
- computing the median (selection problem)
- checking if all elements are distinct (element uniqueness)

Also:

- Topological sorting helps solving some problems for dags.
- Presorting is used in many geometric algorithms.

How fast can we sort ?

- Efficiency of algorithms involving sorting depends on efficiency of sorting.

Theorem: $\lceil \log_2 (n!) \rceil \approx n \log_2 n$ comparisons are necessary in the worst case to sort a list of size n by any comparison-based algorithm.

Note: About $n \log_2 n$ comparisons are also sufficient to sort array of size n (by Mergesort).

Searching with presorting

- Problem: Search for a given K in A[0..n-1]

- Presorting-based algorithm:
  - Stage 1 Sort the array by an efficient sorting algorithm
  - Stage 2 Apply binary search

- **Efficiency?**
  - $\Theta(n \log n) + O(\log n) = \Theta(n \log n)$

- Good or bad?
  - Why do we have our dictionaries, telephone directories, etc. sorted?

Finding Element Uniqueness

- Given a list *A* of *n* orderable elements, determine if there are any duplicates of any element

Brute Force:

```
for each x ∈ A
  for each y ∈ {A – x}
      if x = y return not unique
return unique
```

Presorting:

```
Sort A
for i ← 1 to n-1
   if A[i] = A[i+1] return not unique
return unique
```

Runtime?

Efficiency of Element Uniqueness Algorithms

- Brute force algorithm
  - Compare all pairs of elements
  - Efficiency: $O(n^2)$

- Presorting-based algorithm
  - Stage 1: sort by efficient sorting algorithm (e.g. mergesort)
  - Stage 2: scan array to check pairs of adjacent elements
  - Efficiency: $\Theta(n \log n) + O(n) = \Theta(n \log n)$

- Another algorithm?
  - Hashing (do you own research if you are interested.)

To sort or not to sort?

- You are given n telephone bills and m checks sent to pay the bills (n ≥ m)
  - Assuming that telephone numbers are written on the checks, find out who failed to pay.
  - For simplicity, you may also assume that only one check is written for a particular bill and that it covers the bill in full.

- Brute-force algorithm:
  - O(mn)
- Presorting by telephone number:
  - O($n \log n$)+O($m \log m$)+O(*a merging-like scan of the two sorted lists*)
    = O($n \log n$)

Instance simplification – Gaussian Elimination

- **Given:** A system of *n* linear equations in *n* unknowns with an arbitrary coefficient matrix.

- **Transform to:**
  - An equivalent system of *n* linear equations in *n* unknowns with an upper triangular coefficient matrix.

- Backward substitution
  - Solve the latter by substitutions starting with the last equation and moving up to the first one.

Gaussian Elimination

- This is an example of transform and conquer through representation change

- Consider a system of two linear equations:

  $A_{11}x + A_{12}y = B_1$

  $A_{21}x + A_{22}y = B_2$

- To solve this we can rewrite the first equation to solve for x:

  $A_{11}x = (B_1 - A_{12}y) \rightarrow x = (B_1 - A_{12}y) / A_{11}$

- And then substitute x in the second equation to solve for y. After we solve for y, we can then solve for x:

  $A_{21}(B_1 - A_{12}y) / A_{11} + A_{22}y = B_2 \rightarrow y = ?$

Gaussian Elimination

- In many applications we need to solve a system of n equations with n unknowns, e.g.:

$$A_{11}x_1 + A_{12}x_2 + \ldots + A_{1n}x_n = B_1$$
$$A_{21}x_1 + A_{22}x_2 + \ldots + A_{2n}x_n = B_2$$
$$\ldots$$
$$A_{n1}x_1 + A_{n2}x_2 + \ldots + A_{nn}x_n = B_n$$

- If n is a large number it is very cumbersome to solve these equations using the substitution method.

- Fortunately there is a more elegant algorithm to solve such systems of linear equations: Gaussian elimination
  - Named after Carl Gauss

Gaussian Elimination

The idea is to transform the system of linear equations into an equivalent one that eliminates coefficients so we end up with a triangular matrix.

$A_{11}x_1 + A_{12}x_2 + \ldots + A_{1n}x_n = B_1$
$A_{21}x_1 + A_{22}x_2 + \ldots + A_{2n}x_n = B_2$
$\ldots$
$A_{n1}x_1 + A_{n2}x_2 + \ldots + A_{nn}x_n = B_n$

versus

$A'_{11}x_1 + A'_{12}x_2 + \ldots + A'_{1n}x_n = B'_1$
$\mathbf{0}x_1 + A'_{22}x_2 + \ldots + A'_{2n}x_n = B'_2$
$\ldots$
$\mathbf{0}x_1 + \mathbf{0}x_2 + \ldots + A'_{nn}x_n = B'_n$

In matrix form we can write this as: $Ax = B \rightarrow A'x = B'$

$$A = \begin{bmatrix} A_{11} & A_{12} & \ldots & A_{1n} \\ A_{21} & A_{22} & \ldots & A_{2n} \\ \ldots & & & \\ A_{n1} & A_{n2} & \ldots & A_{nn} \end{bmatrix} \quad B = \begin{bmatrix} B_1 \\ B_2 \\ \ldots \\ B_n \end{bmatrix}$$

Gaussian Elimination

- Why transform?

$$
\begin{aligned}
&A_{11}x_1 + A_{12}x_2 + \ldots + A_{1n}x_n = B_1 \\
&A_{21}x_1 + A_{22}x_2 + \ldots + A_{2n}x_n = B_2 \\
&\ldots \\
&A_{n1}x_1 + A_{n2}x_2 + \ldots + A_{nn}x_n = B_n
\end{aligned}
\quad \Rightarrow \quad
\begin{aligned}
&A'_{11}x_1 + A'_{12}x_2 + \ldots + A'_{1n}x_n = B'_1 \\
&\mathbf{0}x_1 + A'_{22}x_2 + \ldots + A'_{2n}x_n = B'_2 \\
&\ldots \\
&\mathbf{0}x_1 + \mathbf{0}x_2 + \ldots + A'_{nn}x_n = B'_n
\end{aligned}
$$

- The matrix with zeros in the lower triangle (it is called an upper triangular matrix) is easier to solve.

  We can solve the last equation first, substitute into the second to last, etc. working our way back to the first one.

Example of Gaussian Elimination

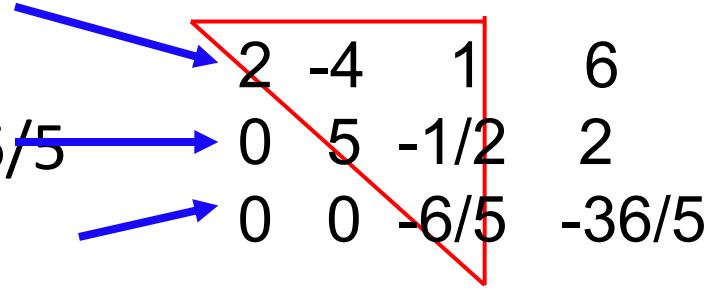
Solve $2x_1 - 4x_2 + x_3 = 6$
$3x_1 - x_2 + x_3 = 11$
$x_1 + x_2 - x_3 = -3$

Coefficient matrix

2	-4	1	6
3	-1	1	11
1	1	-1	-3

Gaussian elimination

	2	-4	1	6
row2 – (3/2)*row1	0	5	-1/2	2
row3 – (1/2)*row1	0	3	-3/2	-6
row3 – (3/5)*row2	0	0	-6/5	-36/5



Gaussian Elimination Example

Solve

$$2x_1 - x_2 + x_3 = 1$$
$$4x_1 + x_2 - x_3 = 5$$
$$x_1 + x_2 + x_3 = 0$$

Coefficient matrix

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

subtract 2*row1

subtract ½*row1

Gaussian elimination

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & \frac{3}{2} & \frac{1}{2} & -\frac{1}{2} \end{bmatrix}$$

subtract ½*row2

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{bmatrix}$$

Example of Gaussian Elimination

Solve $2x_1 - 4x_2 + x_3 = 6$
$3x_1 - x_2 + x_3 = 11$
$x_1 + x_2 - x_3 = -3$

Gaussian elimination

$$\begin{array}{cccc} 2 & -4 & 1 & 6 \\ 0 & 5 & -1/2 & 2 \\ 0 & 0 & -6/5 & -36/5 \end{array}$$

Backward substitution

$x_3 = (-36/5) / (-6/5) = 6$

$x_2 = (2+(1/2)*6) / 5 = 1$

$x_1 = (6 - 6 + 4*1)/2 = 2$

Pseudocode and Efficiency of Gaussian Elimination

Stage 1: Reduction to the upper-triangular matrix

for $i \leftarrow 1$ to n-1 do

    for $j \leftarrow i$+1 to n do
        for $k \leftarrow i$ to n+1 do
            $A[\, j, k] \leftarrow A[\, j, k] - A[\, i, k] * A[\, j, i\,] / A[\, i, i\,]$

Stage 2: Backward substitution

for $j \leftarrow n$ downto 1 do

    $t \leftarrow 0$

    for $k \leftarrow j$ +1 to n do

        $t \leftarrow t + A[\, j, k\,] * x[k]$
    $x[\, j\,] \leftarrow (A[\, j, n+1] - t) / A[\, j, j\,]$

Efficiency: $\Theta(n^3) + \Theta(n^2) = \Theta(n^3)$

Graph Connectivity

- Input: An undirected graph G = (V,E)
- Question: Is the graph connected?
- We are going to reduce this to a *linear algebra* question.

The Laplacian Matrix

- Let A be the *adjacency matrix* of G.
- Let D be the *degree matrix* of a graph G.
  - Diagonal matrix whose (i,i) entry is the degree of node i.
  - Off-diagonal entries are 0.
- Given G, define the *Laplacian matrix* L as D - A.
  - L_{ii} = degree of vertex i
  - L_{ij} = -1 if i and j are adjacent nodes in the graph
  - L_{ij} = 0 otherwise

Demo!

Connectivity and the Spectrum

- The **spectrum** of A, denoted σ(A), is the set of its eigenvalues.
- An ***eigenvalue*** of A is a number λ such that there is a nonzero v with
  - $Av = \lambda v$
- The (algebraic) ***multiplicity*** of an eigenvalue λ is the number of times λ appears as a root of the polynomial $\det(A - \lambda I)$.
- Amazing Fact: ***The multiplicity of the eigenvalue 0 of the Laplacian equals the number of connected components of the graph G!***

Example Implementation

```
import numpy as np

def num_connected_components_laplacian(adj: np.ndarray, *, tol: float = 1e-9) -> int:
    A = np.asarray(adj)
    if A.ndim != 2 or A.shape[0] != A.shape[1]:
        raise ValueError(f"adj must be a square 2D array; got shape {A.shape}")

    # Symmetrize defensively (in case of small floating-point asymmetries)
    A = 0.5 * (A + A.T)

    # Degree matrix diagonal: sum of weights incident to each vertex
    deg = A.sum(axis=1)
    L = np.diag(deg) - A

    # For real symmetric matrices, eigvalsh is stable and returns sorted eigenvalues.
    eigvals = np.linalg.eigvalsh(L)

    # Count eigenvalues close to 0 (L is PSD; tiny negative values can occur numerically)
    return int(np.sum(np.abs(eigvals) <= tol))
```

Transformations and Reductions

Transformations in Algorithms

- We have already been using transformations this semester.
- Examples:
  - Presort an array → simplify searching or duplicate detection
  - Gaussian elimination → transform a linear system into triangular form
  - Reduce 2D problems to 1D problems
- We often solve a problem by transforming it into a different problem that is easier to solve.
- It will be helpful to formalize this process a bit.
  - This is a prelude to the discussion on NP-completeness at the end of the semester.

Formalizing “Problem”

- In this course we have solved problems like:
  - Sort an array
  - Find a path
  - Solve Ax = b
- But to compare problems mathematically, we need a uniform format.
- This motivates the idea of ***decision problem***.
- A ***decision problem*** is a problem whose answer is either YES or NO.

A Problem Is a Set

- Take a decision problem:
  - “Does graph G have a path from s to t?”
- For some inputs the answer is YES.
- For others, the answer is NO.
- So we can divide all possible inputs into two groups:
  - YES instances
  - NO instances
- That set completely determines the problem.
  - If you know exactly which inputs produce YES, you know the problem.

Languages = Sets of Strings

- To reason formally, we encode every input as a binary string.
  - We call such a set a ***language***.
- With this we can define a ***decision problem***
  - A language $L \subseteq \{0,1\}^*$
  - Such that $x \in L$ <-> the answer on input x is YES.
- There is no difference between
  - A decision problem, and
  - The set of inputs for which the answer is YES.

Examples of Decision Problems

- “Does this graph contain a path from s to t?”
- “Is this number prime?”
- “Does this Boolean formula have a satisfying assignment?”

Why use decision problems?

- Every optimization problem has a decision version.
  - How?
- YES/NO problems can be represented as sets of inputs.

- Using the decision problem abstraction lets us compare problems precisely using transformations.

Transformations Between Problems

- Let A and B be decision problems.
- We say A ***reduces*** to B if there is a function f such that:
  - $x \in A \Leftrightarrow f(x) \in B$.
- Intuition: to solve A:
  - Transform x into f(x).
  - Solve B on f(x).
- Examples
  - We presort before searching.
  - We perform Gaussian elimination before back substitution.
- A reduction is just the formalized idea of transformation.

Not All Transformations Are Equal

- In algorithm design:
  - An O(n) preprocessing step is fine
  - An $O(n^3)$ preprocessing step might not be
- In complexity theory, we formalize this:
  - ***Polynomial-time reduction***
    - f is computable in polynomial time.
  - ***Log-space reduction***
    - f is computable using O(log n) memory.
- The allowed power of the transformation determines the notion of "hardness."

What Does “Polynomial Time” Mean?

- An algorithm runs in **polynomial time** if there exists a constant k >= 0 such that, for every input of length n, the running time is bounded by $O(n^k)$.
- Polynomial-time algorithms are considered “efficient” because:
  - They scale reasonably as input size grows.
  - They are stable under composition.
  - They correspond to what is computationally feasible in practice (at least in principle).

Classes of Problems

- We don’t just study individual problems.
- We group problems by the resources required to solve them.
- A ***complexity class*** is a collection of decision problems that can be solved under some resource bound.
- Two examples
  - The class P
    - All decision problems ***solvable*** in polynomial time.
  - The class NP
    - All decision problems whose YES answers can be ***verified*** in polynomial time.
- Once we define classes, we can ask how they are related. For instance, is P = NP?

Completeness: Hardest Problems in a Class

- A problem L* is ***complete*** for a complexity class C if
  - It is in C, and
  - Every problem in C reduces to it.
- That means L* is as hard as any problem in the class.
  - If we can solve L* efficiently, we can solve everything in C efficiently.

Historical Note

- The Halting Problem was understood to be “complete” for undecidable problems.
- By the 1960s, researchers knew:
  - Many natural problems seem hard — but they are decidable.
- The question shifted:
  - Which efficiently solvable problems are hardest? (This corresponds to P.)
  - Which inefficiently solvable problems are hardest? (This corresponds to NP.)
- In 1971, Stephen Cook proved:
  - A problem called SAT is complete for NP.
- This introduced NP-completeness.
  - Related to the question of P vs. NP.