Please fill in the course evaluation!

bu.campuslabs.com/courseeval

Rate your Professor!

https://www.ratemyprofessors.com/professor/2880902
Info about final exam

date: Tuesday, June 29, 10-12 pm in CAS 224

format: similar to midterm and practice final

content:
  • cumulative over the entire semester, includes lecture, lab, hw
  • everything that we covered, including proofs

tools:
  • closed book
  • 2 cheat sheets: one from midterm, one for the second half
  • double sided, hand-written. Can’t use typed pseudocode
  • no electronics
  • must be handed back at the end of the exam

No collaboration!!!!
Max-flow formulation: proof of correctness

Theorem. Max size of a matching in $G = \text{value of max flow in } G'$. 

Pf. size of any matching is $\leq$ value of the max flow 

- Given a max matching $M$ of cardinality $k$. 
- Consider flow $f$ that sends 1 unit along each of $k$ paths. 
- $f$ is a flow, and has value $k$. □
Max-flow formulation: proof of correctness

Theorem. Max size of a matching in $G = \text{value of max flow in } G'$.

Pf. size of the max matching $\geq$ value of any flow

- Let $f$ be a max flow in $G'$ of value $k$.
- Consider $M = \text{set of edges from } L \text{ to } R \text{ with } f(e) = 1$.
  - each node in $L$ and $R$ participates in at most one edge in $M$
  - $|M| = k$: consider cut $(L \cup \{s\}, R \cup \{t\})$
Assume we have an efficient algorithm A to find the maximum flow in a network. Can we use A to solve other problems?

Assume we want to solve problem P. Is there a one-to-one correspondence between the solution to P and the max flow in an appropriate graph?

Reduction (informal): Turn problem P into an instance of network flow. Find the max flow. Find the solution to P with help of the max flow.
Reductions to Network Flow — steps

Bipartite matching. Given a bipartite graph $G = (L \cup R, E)$, find a max matching.

1. Define a corresponding instance of the network flow problem.
   1. define a directed graph
   2. identify the source and sink
   3. assign capacities to edges
2. Find the max flow
3. Based on the max flow solution find a solution to the bipartite matching problem
   1. In fact, the $\text{val}(f) = \text{size of the max matching}$
   2. we can identify the edges in the matching by the flow value on each edge

This approach is useful if steps 1. and 3. are polynomial.
**Edge-disjoint paths**

**Def.** Two paths are edge-disjoint if they have no edge in common.

**Disjoint path problem.** Given a digraph $G = (V, E)$ and two nodes $s$ and $t$, find the max number of edge-disjoint $s \leadsto t$ paths.

**Ex.** Communication networks.

How can we reduce this to the maximum flow problem?

![Diagarm showing a digraph G with two edge-disjoint paths](image)
Edge-disjoint paths
Question. Select all true statements.

A. The value of the maximum flow is at least as much as the number of edge-disjoint st-paths.

B. Every edge with flow can be assigned to one of the edge-disjoint paths.

C. The capacity of the min-cut is an upper bound on the number of edge-disjoint paths.

D. If the graph contains at most k edge-disjoint st-paths, then we can find k edges whose removal would disconnect s from t.
Edge-disjoint paths

Max-flow formulation. Assign unit capacity to every edge.

Theorem. Max number edge-disjoint $s \sim t$ paths equals value of max flow.

Pf. $\leq$
Edge-disjoint paths

Max-flow formulation. Assign unit capacity to every edge.

Theorem. Max number edge-disjoint $s \rightarrow t$ paths equals value of max flow.

Pf. \( \leq \)

- Suppose there are \( k \) edge-disjoint $s \rightarrow t$ paths \( P_1, \ldots, P_k \).
- Set \( f(e) = 1 \) if \( e \) participates in some path \( P_j \); else set \( f(e) = 0 \).
- Since paths are edge-disjoint, \( f \) is a flow of value \( k \).
Edge-disjoint paths

Max-flow formulation. Assign unit capacity to every edge.

**Theorem.** Max number edge-disjoint $s \leadsto t$ paths equals value of max flow.

**Pf.** $\geq$
Edge-disjoint paths

Max-flow formulation. Assign unit capacity to every edge.

Theorem. Max number edge-disjoint \( s \sim t \) paths equals value of max flow.

Pf.  

- Suppose max flow value is \( k \).
- Integrality theorem \( \Rightarrow \) there exists 0–1 flow \( f \) of value \( k \).
- Consider edge \((s, u)\) with \( f(s, u) = 1 \).
  - by flow conservation, there exists an edge \((u, v)\) with \( f(u, v) = 1 \)
  - continue until reach \( t \), always choosing a new edge
- Produces \( k \) edge-disjoint paths.  

\[ \]
Network connectivity

Def. A set of edges $F \subseteq E$ disconnects $t$ from $s$ if every $s \sim t$ path uses at least one edge in $F$.

- Thus removing the edges in $F$ would disconnect $s$ from $t$.

Network connectivity. Given a digraph $G = (V, E)$ and two nodes $s$ and $t$, find the min number of edges whose removal disconnects $t$ from $s$. 
Menger’s theorem

Theorem. [Menger 1927] The max number of edge-disjoint $s \sim t$ paths equals the min number of edges whose removal disconnects $t$ from $s$.

Pf. Using the max-flow formulation Menger’s theorem states the same as the Max Flow Min Cut theorem.
Vertex-disjoint paths

Def. Two paths are vertex-disjoint if they have no vertex in common.
  • note that this implies that they also don’t have edges in common

Vertex disjoint paths problem. Given a digraph $G = (V, E)$ and two nodes $s$ and $t$, find the max number of vertex-disjoint $s \rightarrow t$ paths.
Multiple sources or sinks

Problem: Directed graph $G$ with capacities $c(e)$ has multiple sources $s_1, s_2, \ldots, s_k$ and multiple sinks $t_1, t_2, \ldots, t_l$. Find the max total flow from the sources to the sinks.
**Examples of hard (?) problems**

<table>
<thead>
<tr>
<th>Polynomial — P</th>
<th>Problems in NP, not known to be P or NP-Complete</th>
<th>NP-Complete</th>
<th>Undecidable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shortest paths in graphs</td>
<td>Factoring</td>
<td>Longest paths in graphs</td>
<td>Halting problem</td>
</tr>
<tr>
<td>Interval Scheduling</td>
<td>Graph Isomorphism</td>
<td>Traveling Salesperson</td>
<td>Busy Beaver numbers</td>
</tr>
<tr>
<td>Bipartite Matching</td>
<td></td>
<td>3-D Matching</td>
<td></td>
</tr>
<tr>
<td>Maximum flow</td>
<td></td>
<td>3-SAT</td>
<td></td>
</tr>
<tr>
<td>Detecting cycles</td>
<td></td>
<td>Graph Coloring</td>
<td></td>
</tr>
<tr>
<td>Longest Common Subsequence</td>
<td></td>
<td>Independent Set</td>
<td></td>
</tr>
<tr>
<td>sorting</td>
<td></td>
<td>Hamiltonian Paths</td>
<td></td>
</tr>
</tbody>
</table>
Alan Turing and computational complexity

Alan Turing (1912 - 1954)

launched “computer science” as a field

best known for

- theory of computability
- “universal” computers (a.k.a. Turing machines — take CS332!)
- code breaking
- AI
- statistics
- ...

Algorithm: finite set of simple, unambiguous instructions

- think: Python program on a machine with infinite memory

Turing: Let’s think about algorithms systematically

- formalized algorithms via Turing machines
Complexity classes

Computability
- understand what can/cannot be computed (in finite time) even in principle
  - example: problems that we know an algorithm for can
  - example: halting problem cannot

Complexity
- understand which problems can/cannot be solved in *reasonable amount* of time

  - *Polynomial* problems: problems for which there is a polynomial time algorithm.
  - *Exponential* problems: problems that can be solved in finite time, but we can prove that any algorithm to solve it requires exponential time
    - nxn chess: given a board position in an nxn generalization of chess, can black guarantee a win?
    - Go game
Some well-known complexity classes

Polynomial (P): problems that can be solved in polynomial time
  • How would you “prove” that a problem is polynomial?

NP: problems whose answer is hard to find but a potential solution to it is easy to verify whether it’s indeed a solution

example:
  • longest paths problem - Q: is there a path of length k? proof: list the edges in the path
  • Hamiltonian cycle - proof: cycle itself
  • Factorization - Q; what are the prime factors of n? proof: list the factors
Complexity classes

**Polynomial problems:** problems for which there is a polynomial time algorithm.

**Exponential problems:** problems that can be solved in finite time, but we can prove that any algorithm to solve it requires exponential time.

Bad news: a huge number of problems have defied classification into either group.

Instead we want to at least understand which problems are “similar”:

**Complexity class:** a set (class) of problems that are provable equally complex.

- polynomial problems form a complexity class
- exponential problems form a complexity class

• how do we define “equally complex”?
Defining complexity classes

NP: problems whose answers are *easy to verify* given a *hint*

Observation: both the shortest and longest path problem are in NP, yet they don’t seem to be “equally hard”.

- What are some other problems similar in complexity to the shortest paths problem?
- What are some other problems similar in complexity to the longest paths problem?

Going forward: how to define “similarity”
- tool called polynomial-time reduction
We have an efficient algorithm (FF) to find the maximum flow in a network. Can we use it to solve other problems?

Assume we want to solve problem P.

Reduction (informal): Turn problem P into an instance of network flow. Find the max flow. Find the solution to P with help of the max flow.
NP-complete

Goal. Identify sets of problems of similar difficulty that are not in P.

A problem Y is **NP-complete** if:
1. Y is in NP
2. *every* problem X in NP reduces to Y.

**Corollary:** every NP-complete problem reduces to every other NP-C problem. They are *all polynomially equivalent.*

**Corollary:** if we find a polynomial solution for one NP-C problem, then all of them are solvable in polynomial time.
Does there exist at least 1 NP-complete problem?

- the previous definitions say that if there is one, then there are many. But how do we find one?
The “first” NP-complete problem

**Theorem.** \textsc{Circuit-Sat} \(\in\) NP-complete. [Cook 1971, Levin 1973]

**Summary**

It is shown that any recognition problem solved by a polynomial-time-bounded nondeterministic Turing machine can be solved deterministically in polynomial time. This is done by defining the complexity of polynomials as a notion of polynomial-time complexity, and then using this notion to show that the complexity of polynomials is equal to the complexity of Turing machines. The main result is then obtained by showing that the complexity of polynomials is equal to the complexity of Turing machines.

**Definition**

Let \(\mathcal{S}\) be a set of strings. A polynomial \(P(p, q, ..., z)\) is said to be \(\mathcal{S}\)-reducible if there exists a polynomial \(Q(p, q, ..., z)\) such that \(Q(p, q, ..., z) = \mathcal{S}(p, q, ..., z)\).

**Theorem**

Let \(\mathcal{S}\) be a set of strings. A polynomial \(P(p, q, ..., z)\) is said to be \(\mathcal{S}\)-reducible if there exists a polynomial \(Q(p, q, ..., z)\) such that \(Q(p, q, ..., z) = \mathcal{S}(p, q, ..., z)\).

**Proof**

The proof is by contradiction. Suppose that \(P(p, q, ..., z)\) is not \(\mathcal{S}\)-reducible. Then there exists a polynomial \(Q(p, q, ..., z)\) such that \(Q(p, q, ..., z) \neq \mathcal{S}(p, q, ..., z)\).

Talk at STOC by prof. Levin in honor of the 50th anniversary: https://www.cs.bu.edu/fac/lnd/expo/stoc21/vid.mp4
Circuit satisfiability

**CIRCUIT-SAT.** Given a combinational circuit built from AND, OR, and NOT gates, is there a way to set the circuit inputs so that the output is 1?

![Diagram of a circuit with AND, OR, and NOT gates]

**yes:** 1 0 1

**hard-coded inputs**

**variable inputs**
How to study

• Know the definitions
  - be able to identify them on examples
• Trace all algorithms on specific input
• Be able to compute their running times
  - be able to put running times in order of growth
• Review lab problems & problems from lecture
  - make a list of tricks we used
• Review format of solution
  - DP and network flow applications
• Review proofs
Divide and Conquer

• Break down problem into smaller instances
• Find solution to large instance by first recursively solving the smaller ones
  • instance means solving the same problem with a different (here shorter) input

• Note that instances of the same size have different inputs
  • e.g. in mergeSort we make recursive calls on arrays of length n/2 but each n/2
    array holds different numbers

• We use recurrence equations to compute the running time (= number of
  computational steps) of the algorithm
• Goal is to find a closed-form math formula
  • in general we consider one mathematical operation or one read/write
    operation a computational step. The exception being the lecture we spent on
    integer multiplication
Divide and Conquer - write recurrences

What is the recurrence corresponding to the following algorithm description?

Some algorithm takes as input an array of length n. It divides the data into 5 parts, and makes recursive calls to 4 of those parts. It combines the results in $O(n)$ steps.

A. $T(n) = 4T\left(\frac{n}{5}\right) + O(n)$

B. $T(n) = O\left(n^{\log_4 5}\right)$

C. $T(n) = 5T\left(\frac{n}{4}\right) + O(n)$

D. $T(n) = 5T\left(\frac{n}{5}\right) + O(1)$
D & C - some common recurrences and rules

\[ T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n) \] - mergeSort

\[ T(n) = T\left(\frac{n}{2}\right) + O(1) = O(\log n) \] - binary search

If the formula is of the form \[ T(n) = qT\left(\frac{n}{2}\right) + cn \] where \( q > 2 \)
then we always have \( O\left( n^{\log_2 q} \right) \)

Apply this to \( T(n) = 8T\left(\frac{n}{2}\right) + cn = O(?) \)

A. \( O(\log n) \)
B. \( O(n) \)
C. \( O(n \log n) \)
D. \( O(n^2) \)
E. \( O(n^3) \)
D & C - telescoping method

\[ T(n) = 9T\left(\frac{n}{3}\right) + O(n) \]

trick: when solving the recurrence use \( cn \) instead of \( O(n) \)
D & C - recurrence tree method

\[ T(n) = 4T\left(\frac{n}{2}\right) + O(n) \]
DP - structure, layout of solution

Subproblems correspond to the solution on a smaller input
- “smaller input” is a fixed subset of elements
- may be described by 1, 2, … variables
- always make clear what a solution to a subproblem means

Recursive formula
- find the “last” choice (corresponding to the highest index) to get the opt solution
- may be a max/min over multiple cases
- recursive call is always made to (at least one) lower index

Implementation
- use memoization table
- running time: (size of table) \( \times \) (operations to compute one entry of the table)

backtrack solution:
- find the actual elements in min/max solution
- find what choice was made, e.g. what was the min/max
- recurse on the index corresponding to the choice
- can store info on choice in an extra table while computing opt, not required
knapsack with infinite resources

We are given the info on n items, each with value $v_i$ and weight $w_i$. We are also given a maximum capacity $W$. We have an infinite supply of each item available.

Decide how many of each item to select to maximize the total value while not exceeding the weight limit $W$.

OPT($i$,$w$) =

What is the size of the corresponding DP table?

Select the best recursive formula:

A. $OPT(i, w) = \max\{OPT(i - 1,w); v_i + OPT(i - 1,w - w_i)\}$
B. $OPT(i, w) = \max\{OPT(i - 1,w); v_i + OPT(i, w - w_i)\}$
C. $OPT(i, w) = \max\{OPT(i - 1), v_i + OPT(i)\}$
Flow

Input: weighted directed graph G, source s, sink t, capacity on each edge c(u,v)

Flow: a value \( f(u,v) \) assigned to each edge

- flow constraints
  - when updating an existing flow we have to make sure that the constraints are still met
  - a flow is given as an input, if we know its value on the edges

Value of flow: total flow from s to t

- max flow: maximum achievable
  - the value of the max flow is a unique number
  - the amount of flow on each edge in a max flow is not unique

FF: be comfortable running it

Residual graph:

- be able to draw it, when you are given the flow on each edge (may not be max)
Flow - MFMC

Value of flow: total flow from s to t
  • max flow: maximum achievable

st-cut: subset of nodes connected to s that doesn’t contain t
edges in cut: edges directed from nodes in A to V-A

capacity of cut: total capacity of edges in the cut
  • not flow values
  • min-cut: the cut with the minimum sum
    • not always unique
    • be able to find min-cut given the max flow
    • be able to verify whether a flow is maximum

MFMC: The value of the max flow = capacity of the min-cut
  • intuition: edges in the cut are a bottleneck to the max flow
How many min-cuts do you see in this graph?

A. 1  
B. 2  
C. 3  
D. 4
Flow apps as reductions

examples:
• max bipartite matching
• unique paths
• advertising
• word problems

Solution structure:
1. define corresponding flow graph
   1. specify nodes, edges, capacities
2. find flow
3. relate that back to solution of original problem
Asymptotic growth

big-Oh:
intuition: \( f(n) = O(g(n)) \) if \( g \) is an “upper bound” on \( f \)
  - always look for best upper bound

• combinatoric def: \( f(n) = O(g(n)) \) iff there exist constant \( c>1 \) and \( n_0 \), such that for every \( n> n_0 \) \( f(n) \leq cg(n) \) is true.
• analytic def: \( f(n) = O(g(n)) \) iff \( \limsup_{n\to\infty} \frac{f(n)}{g(n)} = 0 \)
  - the two definitions are equivalent

Write the definition for \( \Omega \) and \( \Theta \)
  • Know the rules for ordering functions (see relevant slide)
Analysis of an iterative algorithm

Algorithm 1: arrayIter\((A, B)\)

\[
\begin{array}{l}
/* A, B are arrays of ints */ \\
/* this algorithm is nonsense ;) */ \\
1 \quad n \leftarrow \text{len}(A); \\
2 \quad m \leftarrow \text{len}(B); \\
3 \quad \textbf{for} \ i = 1 \ \textbf{to} \ n \ \textbf{do} \\
4 \quad \quad \textbf{for} \ j = 1 \ \textbf{to} \ m \ \textbf{do} \\
5 \quad \quad \quad A[i] = A[i] \times B[j]; \\
6 \quad \textbf{return} \ A \\
\end{array}
\]

Algorithm 1: arrayIter2\((A)\)

\[
\begin{array}{l}
/* A are arrays of ints */ \\
/* this algorithm is nonsense ;) */ \\
1 \quad n \leftarrow \text{len}(A); \\
2 \quad \textbf{for} \ i = 1 \ \textbf{to} \ n \ \textbf{do} \\
3 \quad \quad \textbf{for} \ j = i + 1 \ \textbf{to} \ n \ \textbf{do} \\
4 \quad \quad \quad A[i] = A[i] + A[j]; \\
5 \quad \textbf{return} \ A \\
\end{array}
\]
Analysis of iterative graph algorithms

Algorithm 1: deleteEdge($G$)

/* $G$ is the adjacency list if a graph */
1 for $u$ in $G$ do
2     for $v$ in $G[u]$ do
3         if $u > v$ then
4             del $G[u][v]$;
5 return $G$

Algorithm 1: reverseEdge($G$)

/* $G$ is the adjacency list if a graph */
1 for $u$ in $G$ do
2     for $v$ in $G[u]$ do
3         if $u > v$ then
4             del $G[u][v]$;
5             add $G[v][u]$;
6 return $G$
Graph data structures

Adjacency list:
• implement as hash table of hash tables
• hash table:
  • can be used for (non)numerical indices: H[id]
  • can access index in O(1)
    • notation for nested table: G[u][v], for v in G[u]
• iterate over entire adj. list takes O(n+m)
Graph data structures

How would you solve the following using an adjacency list?

Problem: Given a directed graph \( G \), fix some random order of its nodes.

- Delete all back-edges (from higher to lower index) in this graph.
  - What kind of graph is this? How can we find the source nodes?
- Instead of deleting edges, reverse the direction of the back edges (and leave forward edges unchanged). How should we update the adjacency list?
BFS and DFS and applications

Output
• tree/ parent list
• can use to backtrack paths from s
• depends on random choices

BFS:
• find paths with min number of edges
• can use to explore weighted graphs but not to find minimum weighted paths

DFS:
• doesn’t find shortest paths!
• What is it used for?
• connectivity
  • is there a (directed) path between nodes u and v?
  • is the graph connected

applications:
• connected components
• strong connectedness
• decide whether DAG and topological order
Scheduling and partitioning

greedy algorithm:
• makes choices based on local information, doesn’t look at the global structure.
• Once a decision has been made, it’s never changed.

algorithm:
• need to fix (and state) what order to iterate over elements
• sorting or selection order is part of the running time
• fine to order on the fly, e.g. the way the next edge is selected in Prim’s

proof:
• most often we assume that there is some optimal solution
• we could find it using brute force
• we prove that the output of the greedy algorithm is at least as good as the optimal.
Priority queues

Data structure containing keys and corresponding values <key, value>
• keys are priorities
• we used it in applications where lower key = higher priority

implementation:
• many implementations
• we assume binary min-heap
• running times:
  • insert, update (DECREASE-KEY), EXTRACT-MIN: O(log n)
    • if n elements are stored (edges are O(m)!!)
  • PEEK (= read key and value of minimum key): O(1)
Dijkstra’s

Find shortest paths in a weighted graph
• negative edge weights are not allowed

minimum path = minimum sum of weights
• in case of unweighted graphs we may assume that the edge weights are one.
  Then number of edges = weight of paths

uses PQ for efficiency