

CSC373

Review

BUT FIRST...

Course Evaluation

Low response rate

Complete your course evaluations...



Check your e-mail for a link to your evaluations, or log-in to www.portal.utoronto.ca and click the *Course Evals* tab!

Should I expect drumrolls?

Course Evaluation

New response rate

Topics

- Divide and conquer
- Greedy algorithms
- Dynamic programming
- Network flow
- Linear programming
- Complexity
- Approximation algorithms & local search
- Randomized algorithms

Greedy Algorithms

- Greedy algorithm outline

- We want to find the optimal solution maximizing some objective f over a large space of feasible solutions
- Solution x is composed of several parts (e.g. a set)
- Instead of directly computing x ...
 - Consider one element at a time in some greedy ordering
 - Make a decision about that element before moving on to future elements (and without knowing what will happen for the future elements)

Greedy Algorithms

- Proof of optimality outline

- Strategy 1:

- G_i = greedy solution after i steps
- Show that $\forall i$, there is some optimal solution OPT_i s.t. $G_i \subseteq OPT_i$
 - “Greedy solution is promising”
- By induction
- Then the final solution returned by greedy must be optimal

- Strategy 2:

- Same as strategy 1, but more direct
- Consider OPT that matches greedy solution for as many steps as possible
- If it doesn't match in all steps, find another OPT which matches for one more step (contradiction)

Dynamic Programming

- Key steps in designing a DP algorithm
 - “Generalize” the problem first
 - E.g. instead of computing edit distance between strings $X = x_1, \dots, x_m$ and $Y = y_1, \dots, y_n$, we compute $E[i, j]$ = edit distance between i -prefix of X and j -prefix of Y for all (i, j)
 - The right generalization is often obtained by looking at the structure of the “subproblem” which must be solved optimally to get an optimal solution to the overall problem
 - Remember the difference between DP and divide-and-conquer
 - Sometimes you can save quite a bit of space by only storing solutions to those subproblems that you need in the future

Dynamic Programming

- Dynamic programming applies well to problems that have **optimal substructure property**
 - Optimal solution to a problem contains (or can be computed easily given) optimal solution to subproblems.
- **Recall: divide-and-conquer also uses this property**
 - You can think of divide-and-conquer as a special case of dynamic programming, where the two (or more) subproblems you need to solve don't "overlap"
 - So there's no need for memoization
 - In dynamic programming, one of the subproblems may in turn require solution to the other subproblem...

Dynamic Programming

- **Top-Down may be preferred...**
 - ...when not all sub-solutions need to be computed on some inputs
 - ...because one does not need to think of the “right order” in which to compute sub-solutions
- **Bottom-Up may be preferred...**
 - ...when all sub-solutions will anyway need to be computed
 - ...because it is sometimes faster as it prevents recursive call overheads and unnecessary random memory accesses

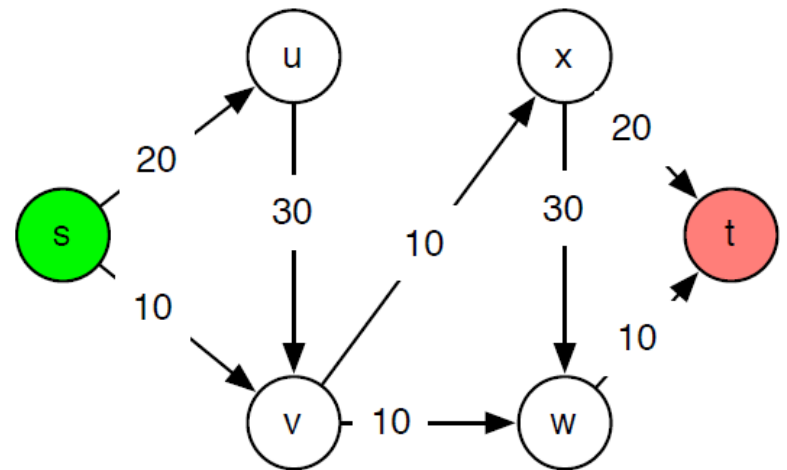
Network Flow

- **Input**

- A directed graph $G = (V, E)$
- Edge capacities $c : E \rightarrow \mathbb{R}_{\geq 0}$
- Source node s , target node t

- **Output**

- Maximum “flow” from s to t



Ford-Fulkerson Algorithm

MaxFlow(G):

// initialize:

Set $f(e) = 0$ for all e in G

// while there is an s-t path in G_f :

While $P = \text{FindPath}(s, t, \text{Residual}(G, f)) \neq \text{None}$:

$f = \text{Augment}(f, P)$

 UpdateResidual(G, f)

EndWhile

Return f

Max Flow - Min Cut

- **Theorem:** In any graph, the value of the maximum flow is equal to the capacity of the minimum cut.
- **Ford-Fulkerson can be used to find the min cut**
 - Find the max flow f^*
 - Let A^* = set of all nodes reachable from s in residual graph G_{f^*}
 - Easy to compute using BFS
 - Then $(A^*, V \setminus A^*)$ is min cut

LP, Standard Formulation

- **Input:** $c, a_1, a_2, \dots, a_m \in \mathbb{R}^n, b \in \mathbb{R}^m$
 - There are n variables and m constraints
- **Goal:**

$$\begin{aligned} & \text{Maximize } c^T x \\ & \text{Subject to } a_1^T x \leq b_1 \\ & \quad a_2^T x \leq b_2 \\ & \quad \vdots \\ & \quad a_m^T x \leq b_m \\ & \quad x \geq 0 \end{aligned}$$

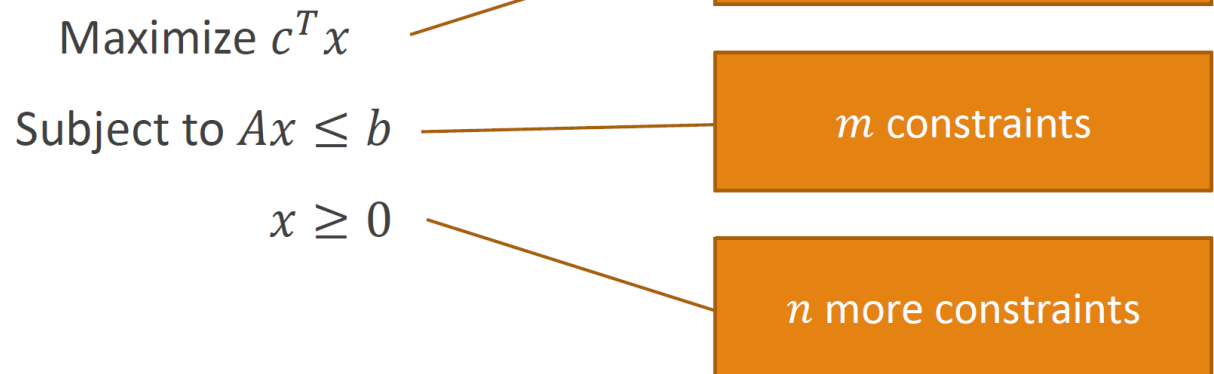
n variables

m constraints

n more constraints

LP, Standard Matrix Form

- **Input:** $c, a_1, a_2, \dots, a_m \in \mathbb{R}^n, b \in \mathbb{R}^m$
 - There are n variables and m constraints
- **Goal:**



Convert to Standard Form

- What if the LP is not in standard form?
 - Constraints that use \geq
 - $a^T x \geq b \Leftrightarrow -a^T x \leq -b$
 - Constraints that use equality
 - $a^T x = b \Leftrightarrow a^T x \leq b, a^T x \geq b$
 - Objective function is a minimization
 - Minimize $c^T x \Leftrightarrow$ Maximize $-c^T x$
 - Variable is unconstrained
 - x with no constraint \Leftrightarrow Replace x by two variables x' and x'' , replace every occurrence of x with $x' - x''$, and add constraints $x' \geq 0, x'' \geq 0$

Duality

Primal LP

$$\max \mathbf{c}^T \mathbf{x}$$

$$\mathbf{A}\mathbf{x} \leq \mathbf{b}$$

$$\mathbf{x} \geq 0$$

Dual LP

$$\min \mathbf{y}^T \mathbf{b}$$

$$\mathbf{y}^T \mathbf{A} \geq \mathbf{c}^T$$

$$\mathbf{y} \geq 0$$

- **Weak duality theorem:**

- For any primal feasible x and dual feasible y , $c^T x \leq y^T b$

- **Strong duality theorem:**

- For any primal optimal x^* and dual optimal y^* , $c^T x^* = (y^*)^T b$

P

- P (polynomial time)
 - The class of all decision problems computable by a TM in polynomial time

NP

- NP (nondeterministic polynomial time)
 - The class of all decision problems for which a YES answer can be verified by a TM in polynomial time given polynomial length “advice” or “witness”.
 - There is a polynomial-time verifier TM V and another polynomial p such that
 - For all YES inputs x , there exists y with $|y| = p(|x|)$ on which $V(x, y)$ returns YES
 - For all NO inputs x , $V(x, y)$ returns NO for every y
 - Informally: “Whenever the answer is YES, there’s a short proof of it.”

co-NP

- co-NP

- Same as NP, except whenever the answer is NO, we want there to be a short proof of it

Reductions

- Problem A is **p-reducible** to problem B if an “oracle” (subrouting) for B can be used to efficiently solve A
 - You can solve A by making polynomially many calls to the oracle and doing additional polynomial computation

NP-completeness

- NP-completeness

- A problem B is NP-complete if it is in NP and every problem A in NP is p-reducible to B
- Hardest problems in NP
- If one of them can be solved efficiently, every problem in NP can be solved efficiently, implying $P=NP$

- Observation:

- If A is in NP, and some NP-complete problem B is p-reducible to A , then A is NP-complete too
 - “If I could solve A , then I could solve B , then I could solve any problem in NP”

Review of Reductions

- If you want to show that problem B is NP-complete
- **Step 1: Show that B is in NP**
 - Some polynomial-size advice should be sufficient to verify a YES instance in polynomial time
 - No advice should work for a NO instance
 - Usually, the solution of the “search version” of the problem works
 - But sometimes, the advice can be non-trivial
 - For example, to **check LP optimality**, one possible advice is the **values of both primal and dual variables**, as we saw in the last lecture

Review of Reductions

- If you want to show that problem B is NP-complete
- **Step 2: Find a known NP-complete problem A and reduce it to B (i.e. show $A \leq_p B$)**
 - This means taking an arbitrary instance of A, and solving it in polynomial time using an oracle for B
 - Caution 1: Remember the direction. You are “reducing known NP-complete problem to your current problem”.
 - Caution 2: The size of the B-instance you construct should be polynomial in the size of the original A-instance
 - This would show that if B can be solved in polynomial time, then A can be as well
 - Some reductions are trivial, some are notoriously tricky...

Approximation Algorithms

- We focus on **optimization problems**
 - Decision problem: “Is there...where... $\geq k$?”
 - E.g. “Is there an assignment which satisfies at least k clauses of a given formula φ ?”
 - Optimization problem: “Find...which maximizes...”
 - E.g. “Find an assignment which satisfies the maximum possible number of clauses from a given formula φ .”
 - Recall that if the decision problem is hard, then the optimization problem is hard too

Approximation Algorithms

- There is a function *Profit* we want to maximize or a function *Cost* we want to minimize
- Given input instance I ...
 - Our algorithm returns a solution $ALG(I)$
 - An optimal solution maximizing *Profit* or minimizing *Cost* is $OPT(I)$
 - Then, the approximation ratio of ALG on instance I is

$$\frac{Profit(OPT(I))}{Profit(ALG(I))} \quad \text{or} \quad \frac{Cost(ALG(I))}{Cost(OPT(I))}$$

Approximation Algorithms

- Approximation ratio of ALG on instance I is

$$\frac{Profit(OPT(I))}{Profit(ALG(I))} \quad \text{or} \quad \frac{Cost(ALG(I))}{Cost(OPT(I))}$$

➤ Note: These are defined to be ≥ 1 in each case.

- 2-approximation = half the optimal profit / twice the optimal cost

- ALG has worst-case c -approximation if for each instance I ...

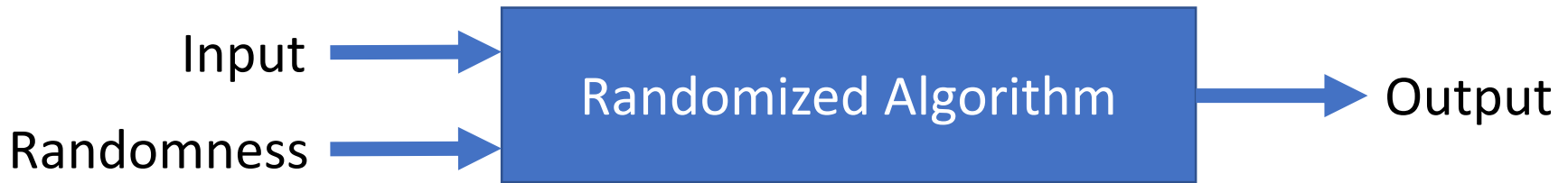
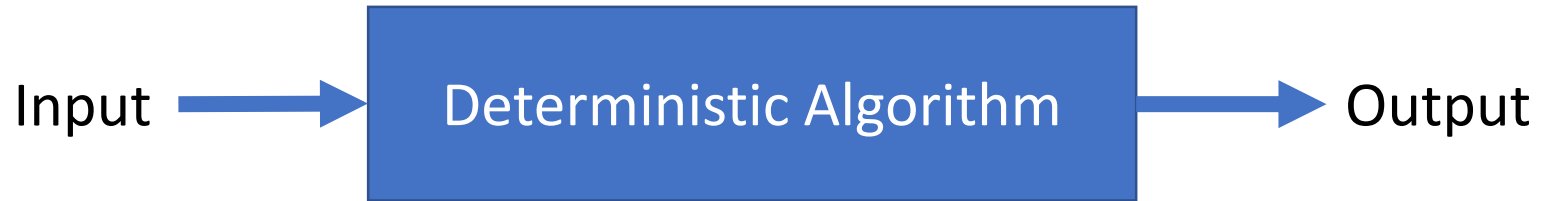
$$Profit(ALG(I)) \geq \frac{1}{c} \cdot Profit(OPT(I)) \quad \text{or}$$

$$Cost(ALG(I)) \leq c \cdot Cost(OPT(I))$$

Local Search

- A heuristic paradigm for solving complex problems
 - Sometimes it might provably return an optimal solution
 - But even if not, it might give a good approximation
- Idea:
 - Start with some solution S
 - While there is a “better” solution S' in the **local neighborhood** of S
 - Switch to S'
- Need to define what is “better” and what is a “local neighborhood”

Randomized Algorithms



Randomized Algorithms

- Running time

- Sometimes, we want the algorithm to *always* take a small amount of time
 - Regardless of both the input and the random coin flips
- Sometimes, we want the algorithm to take a small amount of time *in expectation*
 - Expectation over random coin flips
 - Still regardless of the input

Randomized Algorithms

- Efficiency

- We want the algorithm to return a solution that is, *in expectation*, close to the optimum according to the objective under consideration
 - Once again, the expectation is over random coin flips
 - We want this to hold for every input

Derandomization

- For some problems, it is easy to come up with a very simple randomized approximation algorithm
- Later, one can ask whether this algorithm can be “derandomized”
 - Informally, the randomized algorithm is making some random choices, and sometimes they turn out to be good
 - Can we make these “good” choices deterministically?